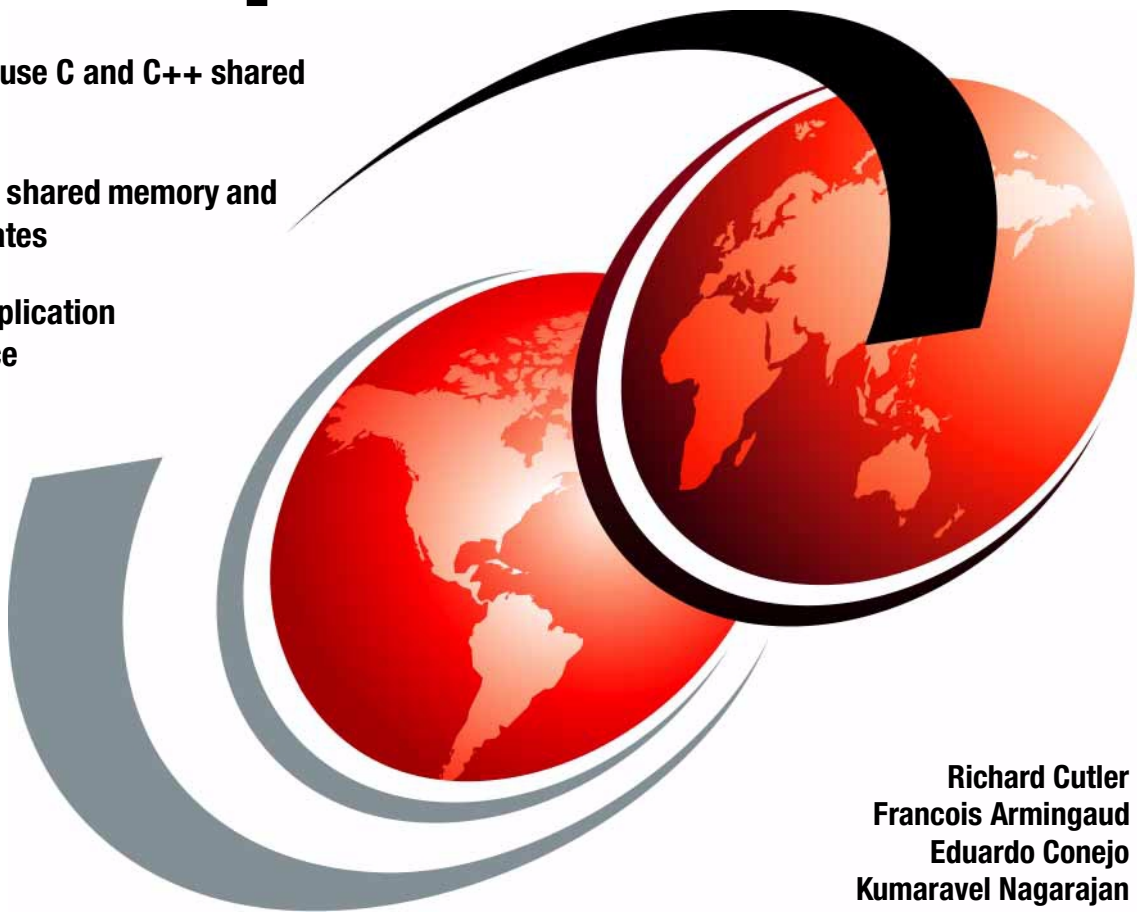
C and C++ Application Development on AIX

- Create and use C and C++ shared libraries
- Understand shared memory and C++ templates
- Improve application performance



Richard Cutler
Francois Armingaud
Eduardo Conejo
Kumaravel Nagarajan

ibm.com/redbooks

Redbooks

SG24-5674-00

International Technical Support Organization

C and C++ Application Development on AIX

September 2000

Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix A, “Special notices” on page 241.

First Edition (September 2000)

This edition applies to VisualAge C++ Professional for AIX, Version 5, Program Number 5765-E26, and IBM C for AIX, Program Number 5765-E32, for use with the AIX Operating System, Program Number 5765-C34.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. JN9B Building 003 Internal Zip 2834
11400 Burnet Road
Austin, Texas 78758-3493

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Figures

Tables

Preface

This IBM Redbook is intended to help experienced UNIX application developers who are new to the AIX operating system. The book explains, in detail, the features of the AIX operating system and IBM compilers that are most often misunderstood by developers, primarily C++ templates and the creation and use of shared libraries and shared memory segments.

In addition to discussing the topics that can sometimes prove troublesome, this book explores some of the features of AIX for measuring and improving application performance, such as profiling and threading, along with memory facilities.

This book contains a brief history of the different IBM C and C++ compiler products that are available for AIX, along with installation, configuration, and co-existence tips.

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization Austin, Center.

Richard Cutler is an AIX and RS/6000 Technical Specialist at the ITSO, Austin Center. Before joining the ITSO, he worked in the RS/6000 Technical Center in the UK where he assisted customers and independent software vendors to port their applications to AIX.

Francois Armingaud is a consultant in IBM France. He entered IBM in 1978 after six years of teaching Computer Science in France (Ecole des Mines de Nancy, Ecole des Mines de Paris) and abroad (Ecole Polytechnique of Algiers). He has 12 years of experience in the AIX/Unix field since 1987. He holds a degree in Engineering from Ecole des Mines de Nancy. His areas of expertise include AIX, Linux, C/C++ programming, and system and network administration.

Eduardo Conejo is an I/T Specialist in IBM Brazil. He as a degree in Mechanical Engineering from Universidade Estadual de Campinas - UNICAMP. He joined IBM in 1997, and has 10 years of industry experience. His specialist subjects include real-time programming and multi-threaded applications.

Kumaravel Nagarajan is an I/T Specialist in IBM India. He has over three years experience in C and C++ development on UNIX systems. He has an

Engineering degree in Electrical and Electronics and a post-graduate degree in Management (Systems). His areas of expertise include shared libraries, threading concepts, and AIX system administration.

Thanks to the following people for their invaluable contributions to this project:

Mark Changfoot
IBM Toronto

Rene Matteau
IBM Toronto

Paul Pacholski
IBM Toronto

Derek Truong
IBM Toronto

John Owczarzak, Editor
ITSO Austin

Comments welcome

Your comments are important to us!

We want our Redbooks to be as helpful as possible. Please send us your comments about this or other Redbooks in one of the following ways:

- Fax the evaluation form found in “IBM Redbooks review” on page 259 to the fax number shown on the form.
- Use the online evaluation form found at `ibm.com/redbooks`
- Send your comments in an Internet note to `redbook@us.ibm.com`

Chapter 1. C and C++ compilers on AIX

Over the years, IBM has offered a variety of compiler products to perform the compilation of C and C++ programs on AIX. These C and C++ compilers have evolved over time and can be tracked by their different version numbers. The C and C++ compilers introduced new compiler features to take advantage of the functionality included in new releases of the AIX operating system.

1.1 Compiler product similarities

All of the IBM C and C++ compiler products for AIX Version 4 share some similar characteristics, in particular, the way the products are installed on the system and the configuration options available when using the products.

1.1.1 Multiple command line drivers

Each compiler product, with the exception of VisualAge C++ Version 4.0, has multiple command line driver interfaces available, each causing a different set of default arguments to be used. For example, the C compiler products provide commands, such as cc, xlc, c89, cc_r, and so on. These commands are all links to a single compiler core, which uses a specific set of options depending on the name of the command used to invoke it.

In addition to the default invocation commands provided when the compiler is installed, the system administrator can create new commands, which result in the compiler being invoked with a customized set of default options. This feature is controlled by the compiler configuration file, which lists the options to be used for each invocation command. The exact name of the configuration file differs between the compiler products, but generally has a name of the form /etc/*comp*.cfg, where *comp* indicates the compiler product that uses the configuration file.

1.1.1.1 Finding the compiler drivers

The earlier versions of the compiler products automatically created symbolic links in /usr/bin for each invocation command supplied by the compiler. For example, this means that if a user has the directory, /usr/bin, as part of their PATH environment variable (which it is by default), they need only type cc on the command line to invoke the `/usr/bin/cc` command.

The later versions of the compiler products are designed to co-exist with earlier versions, and, as a consequence, they do not create the symbolic links in /usr/bin when they are installed. This means that a user may have trouble

invoking the compiler on a system that only has a new version compiler product installed. There are two solutions available in this instance:

- When logged in as the root user, invoke the `replaceCSET` command supplied with the compiler. This will create appropriate symbolic links in /usr/bin to the compiler driver programs.
- Alter the PATH environment variable to add the directory that contains the compiler driver programs. For example:

```
PATH=/usr/vac/bin:$PATH; export PATH
```

The second solution should be used if two compilers are installed on a system since it allows each user to choose which version of the compiler they wish to use. If the system only has one compiler installed, it makes sense to use the first solution. If required, the root user can reverse the action of the `replaceCSET` command by using the `restoreCSET` command, which is also supplied with the compiler. The exact location of the `replaceCSET` and `restoreCSET` commands will depend on the version of the compiler you are using.

1.1.2 Installation directory

The main components of the compiler product are installed on the system in the /usr file system. The exact directory used depends on the compiler product. Table 1 on page 7 shows the location of this directory for C compiler products. Table 3 on page 11 shows the location of this directory for C++ compiler products.

1.2 IBM C compilers

This section describes the various IBM C compilers for AIX. The details provided are limited to compatibility issues of the various compilers with the different versions of the AIX operating system. This information can help in deciding which version of a compiler product should be used in a given situation, based on the version of AIX that will be used. The information can also help avoid problems when upgrading a compiler product to a newer version.

AIX Version 3.2 included a bundled C compiler as part of the operating system. This compiler was known as XLC Version 1.3. When AIX Version 4.1 was introduced, the compiler product was unbundled from the operating system and offered for sale as a separately orderable product. This redbook covers the C and C++ compiler products for AIX Version 4.1 and later versions.

1.2.1 IBM C for AIX, Version 3

The IBM C for AIX Version 3 product was the first C compiler product from IBM for AIX Version 4.1. The *Version 3* in the name of the product specifies the version number of the compiler product rather than the version of the AIX operating system the compiler is compatible with.

Initially released as Version 3.1.0, this compiler evolved over time with the addition of Program Temporary Fixes (PTFs) to become C for AIX Version 3.1.4, which was supported on AIX Version 4.1 and AIX Version 4.2. This compiler was not supported on AIX Version 3.2 and is not supported on AIX Version 4.3.

C programs written using C Set ++ Version 2 and XL C Version 1.3 on AIX Version 3.2 are source compatible with C for AIX Version 3, with some exceptions to detect invalid programs or areas where results are undefined. These exceptions are documented in the product README file.

The compiler product itself is installed in /usr/lpp/xlC, and symbolic links are created in /usr/bin for the command line driver programs, for example, /usr/bin/cc and /usr/bin/c89. The default PATH environment variable means that most users need only type cc on the command line to invoke the /usr/bin/cc driver program, which, in turn, is a symbolic link to the driver /usr/lpp/xlC/bin/xlC. The compiler configuration file is /etc/xlC.cfg.

The C for AIX Version 3 compiler product uses the NetLS licensing system to control usage of the product.

This product has been withdrawn from marketing and is no longer available for purchase. Support for this product has also been discontinued. Current users of this product are encouraged to upgrade to the IBM C for AIX Version 5.0 compiler product, described in Section 1.2.5, “IBM C for AIX, Version 5.0” on page 6.

1.2.2 IBM C for AIX, Version 4.1

A new version of the C compiler product was introduced with AIX Version 4.3.0. The compiler product, IBM C for AIX Version 4.1, had a number of new features, including new optimization routines for improved execution performance, new inter-procedural analysis tools, precompiled headers for improved compiler performance, improved memory management, and improved prototyping of programs. This version of the compiler was supported on AIX Version 4.1.4, AIX Version 4.2 and AIX Version 4.3.

C programs written using either Version 2 or 3 of IBM C Set ++ for AIX, Version 3 of IBM C for AIX, or the XL C compiler component of AIX Version 3.2 are source compatible with C for AIX Version 4.1, with some exceptions to detect invalid programs or areas where results are undefined.

Two important configuration differences introduced with this version of the compiler are:

1. The compiler product is now installed under /usr/vac, rather than /usr/lpp/xlC.
2. The installation process does not create symbolic links to the driver programs from the /usr/bin directory. This is because the C compiler has been designed to co-exist on a system that already has the previous version of the C compiler or a version of the CSet++ compiler installed.

If the system does not have another version of the compiler installed, the symbolic links in the /usr/bin directory can be created by invoking the supplied command `/usr/vac/bin/replaceCSET`, which, as the name implies, replaces the symbolic links to the CSet driver programs.

The compiler product also includes the `/usr/vac/bin/restoreCSET` command, which can be used to reverse the actions of the `replaceCSET` command.

Alternatively, if multiple versions of the compiler exist, or if the user does not want to create symbolic links in /usr/bin, the setting of the PATH environment variable can be used to determine which compiler product is used.

For example, setting the PATH environment variable as follows:

```
PATH=/usr/vac/bin:$PATH; export PATH
```

will result in the C for AIX Version 4.1 compiler being used when the `cc` command is invoked.

The compiler configuration file is /etc/vac.cfg.

The C for AIX Version 4.1 compiler uses the License Use Management (LUM) licensing system to control usage of the product. Refer to Section 1.5, “Activating the compilers” on page 16 for information on configuring the licence system.

This product has been withdrawn from marketing and is no longer available for purchase. Support for this product has also been discontinued. Current users of this product are encouraged to upgrade to the IBM C for AIX Version

5.0 compiler product, described in Section 1.2.5, “IBM C for AIX, Version 5.0” on page 6.

1.2.3 IBM C for AIX, Version 4.3

The IBM C for AIX Version 4.3 compiler product was introduced shortly after the release of AIX Version 4.3.0 and the 64-bit hardware models of the RS/6000 family. This version of the compiler was similar to the IBM C for AIX Version 4.1 compiler, except that it added support for creating and debugging 64-bit application binaries for use on the 64-bit hardware. This version of the compiler is installed under /usr/vac, and uses the /etc/vac.cfg configuration file. If C for AIX Version 4.1 is already installed, installing C for AIX Version 4.3 will overwrite and upgrade the previous version.

The C for AIX Version 4.3 compiler uses the LUM licensing system to control usage of the product. Refer to Section 1.5, “Activating the compilers” on page 16 for information on configuring the licence system.

This product has been withdrawn from marketing and is no longer available for purchase. Support for this product has also been discontinued. Current users of this product are encouraged to upgrade to the IBM C for AIX Version 5.0 compiler product, described in Section 1.2.5, “IBM C for AIX, Version 5.0” on page 6.

1.2.4 IBM C for AIX, Version 4.4

The IBM C for AIX Version 4.4 compiler product is an improved version of the previously released C for AIX Version 4.3. The main enhancement is that this compiler was designed to exploit the RS/6000 Symmetric Multi-Processing (SMP) architecture. It supports automatic parallellization of a C program as well as explicit parallellization through a set of directives that enable the user to parallelize selected sections of the application program. This version of the C compiler is supported only by AIX Version 4.1.5 or later.

C programs written using either Version 2 or Version 3 of IBM C Set ++ for AIX, the XL C compiler component of AIX Version 3.2, or previous versions of the C for AIX Version 4.x compilers, are source compatible with C for AIX Version 4.4, with some exceptions to detect invalid programs or areas where results are undefined.

The compiler is installed under /usr/vac, and uses the /etc/vac.cfg configuration file. If a previous version of C for AIX 4.x is installed, installing C for AIX Version 4.4 will overwrite and upgrade the previous version.

The C for AIX Version 4.4 compiler uses the LUM licensing system to control usage of the product. Refer to Section 1.5, “Activating the compilers” on page 16 for information on configuring the licence system.

This product will be withdrawn from marketing as of August 16th 2000, and will not be available for purchase after that date. Support for this product will be discontinued as of January 31st 2001. Current users of this product are encouraged to upgrade to the IBM C for AIX Version 5.0 compiler product, described in Section 1.2.5, “IBM C for AIX, Version 5.0” on page 6.

1.2.5 IBM C for AIX, Version 5.0

The C for AIX Version 5.0 compiler is the latest IBM C compiler product available for AIX. It extends the existing symmetric multi-processing (SMP) support available with C for AIX Version 4.4 by supporting the OpenMP industry specification. OpenMP provides a model for parallel programming that allows a program to be portable across shared memory architectures from different vendors by using a common set of application program interfaces. The compiler generates highly-optimized code for all RS/6000 processors and can provide run-time address checking to detect memory errors.

This compiler is supported only by IBM AIX Version 4.2.1 or later. Also, note that 64-bit applications will run only on AIX Version 4.3 and later when running on 64-bit hardware.

C programs written using Version 3 or Version 4 of IBM C for AIX are source compatible with IBM C for AIX, Version 5.0. C programs written using either Version 2 or 3 of IBM Set ++ for AIX or the XL C compiler component of AIX Version 3.2 are source compatible with IBM C for AIX, Version 5.0 with exceptions to detect invalid programs or areas where results are undefined.

This version of the compiler is installed under /usr/vac and uses the /etc/vac.cfg configuration file. If C for AIX Version 4.x is installed on a system, installing C for AIX Version 5.0 will overwrite and upgrade the previous version.

The C for AIX Version 5.0 compiler uses the LUM licensing system to control usage of the product. Refer to Section 1.5, “Activating the compilers” on page 16 for information on configuring the licence system.

1.2.6 C compiler summary

Table 1 summarizes the various versions of IBM C compiler products for AIX.

Table 1. IBM C compilers for AIX

Compiler	Installation directory	Configuration file	Supported AIX levels	Licensing Method	Drivers in /usr/bin
C for AIX, Version 3	/usr/lpp/xlC	/etc/xlC.cfg	4.1, 4.2	NetLS	Yes
C for AIX, Version 4.1	/usr/vac	/etc/vac.cfg	4.1.4, 4.2, 4.3	LUM	No
C for AIX, Version 4.3	/usr/vac	/etc/vac.cfg	4.1.5, 4.2, 4.3	LUM	No
C for AIX, Version 4.4	/usr/vac	/etc/vac.cfg	4.2, 4.3	LUM	No
C for AIX, Version 5.0	/usr/vac	/etc/vac.cfg	4.2, 4.3	LUM	No

1.3 IBM C++ Compilers

This section describes the various IBM C++ Compilers for AIX. The details provide here are, again, limited to compatibility issues of the various compilers with the different versions of the AIX operating system. This information can help decide which C++ Compiler product to use for a particular project, based on the target version of AIX, and the nature of the C++ source code being compiled.

1.3.1 IBM C Set ++ for AIX, Version 3

The IBM C Set ++ for AIX, Version 3 product was the first C++ compiler product from IBM for AIX, Version 4.1. The *Version 3* in the name of the product specifies the version of the compiler product rather than the version of the AIX operating system the compiler is compatible with. The C Set ++ for AIX Version 3 product is, in effect, an extension of the C for AIX Version 3 compiler. An alternative view is that the C for AIX Version 3 compiler is a subset of the C Set ++ for AIX compiler.

This compiler was initially released as Version 3.1.0, and evolved over time with the addition of Program Temporary Fixes (PTFs) to become C Set ++ for AIX Version 3.1.4, which was supported on AIX Version 4.1 and AIX Version 4.2. It was not supported on AIX Version 3.2 and is not supported on AIX Version 4.3.

C++ programs written using C Set ++ Version 2 on AIX Version 3.2 are source compatible with C Set ++ for AIX Version 3, with some exceptions to detect

invalid programs or areas where results are undefined. These exceptions are documented in the product README file.

The compiler product itself is installed in /usr/lpp/xlC, and symbolic links are created in /usr/bin for the command line driver programs, for example, /usr/bin/xlC. The default PATH environment variable means that most users need only type xlC on the command line to invoke the /usr/bin/xlC driver program, which, in turn, is a symbolic link to the driver /usr/lpp/xlC/bin/xlC. The compiler configuration file is /etc/xlC.cfg.

The C Set ++ for AIX Version 3 compiler product uses the NetLS licensing system to control usage of the product.

This product has been withdrawn from marketing and is no longer available for purchase. Support for this product has also been discontinued. Current users of this product are encouraged to upgrade to IBM VisualAge C++ Professional for AIX Version 5.

1.3.2 IBM C and C++ Compilers, Version 3.6

The official name of this product is IBM C and C++ compilers for AIX, OS/2, and Windows NT. The product is part of a family of related compilers, with versions available for each of the mentioned platforms. The product is sometimes referred to as the *Portapak* compiler, or more commonly, C Set ++ Version 3.6. The AIX Version of the product is the follow on to C Set ++ Version 3 for AIX.

The product offered a number of facilities to assist in the development of cross-platform applications, where the same source code is used on multiple platforms. The product includes a rich set of IBM class libraries, memory management routines, graphical debuggers, and resource tools for creating and compiling resources and converting between platform formats. The C compiler component of the product can produce either 32-bit or 64-bit executables when used on AIX, Version 4.3.

The product is supported on AIX, Version 4.1.4 or later.

C++ programs written using Version 3 of C Set ++ for AIX, and earlier, are source compatible with the C++ compiler of C Set ++ for AIX, Version 3.6.

As with the other post Version 3.1 compiler products, the compiler command drivers are not created in /usr/bin when the product is installed.

The installation directory is /usr/ibmcxx, and the configuration file is /etc/ibmcxx.cfg.

The product uses the LUM license management system to control usage of the product. Refer to Section 1.5, “Activating the compilers” on page 16 for information on configuring the licence system.

This product has been withdrawn from marketing and is no longer available for purchase. Support for this product has also been discontinued. Current users of this product are encouraged to upgrade to IBM VisualAge C++ Professional for AIX Version 5.

1.3.3 IBM VisualAge C++ Professional for AIX, Version 4

VisualAge C++ Professional for AIX, Version 4 is a powerful rapid application development (RAD) tool for building C and C++ applications. This heterogeneous RAD environment provides:

- Tools, including a graphical debugger
- Visual Builder, Data Access Builder
- Incremental compiler and linker
- A rich set of class libraries
- Online help and a powerful full-text search engine

VisualAge C++ Professional provides a standards-compliant C++ compiler. Its incremental development environment and visual programming tools improve programmer productivity.

This product features an incremental compiler and linker and, as such, is not ideally suited for use when working with existing application code that uses Makefiles or for a development environment that maintains a single source tree for multiple platforms and uses Makefiles. For this reason, the product includes a copy of the IBM CSet ++, Version 3.6 compiler for use in a batch compile environment.

This compiler product runs on IBM AIX, Version 4.1.5, or later, for RS/6000.

C++ programs written using Version 3.6 of IBM C and C++ compilers, and earlier, are source compatible with the C++ compiler component of VisualAge C++ Professional for AIX, Version 4.

Note

As described above, this version of VisualAge features an incremental compiler. The implications of this for productivity and the code are impressive, but if the application is moving from a batch environment, do spend time with the application to adapt to the VisualAge products. For example, makefiles cannot be processed directly by the incremental compiler.

But, once the migration is done, then the advantages of VisualAge products are very impressive. This then would reduce the amount of time and memory required to do each build as well as the time spent on rebuilding when some changes are made to the source files.

This product is expected to be withdrawn from marketing in late 2000, and will no longer be available for purchase. Support for this product will be discontinued as of January 31st 2001. Current users of this product are encouraged to upgrade to IBM VisualAge C++ Professional for AIX Version 5.

1.3.4 IBM VisualAge C++ Professional for AIX, Version 5

VisualAge C++ Version 5.0 features a fully incremental compiler and a new batch compiler. The Integrated Development Environment (IDE) operates with the incremental compiler when used in the AIX Common Desktop Environment (CDE). The batch compiler is run from the command line and is suitable for use in a development environment that uses Makefiles. Both compilers support the latest ANSI/ISO C++ language standard and the latest version (Version 5) of the IBM Open Class library.

The main differences between Version 4 and Version 5 of this product are:

- Version 5 supports multiple codestores in a single project.
- Version 5 is a single product featuring both batch and incremental compilers.

The graphical interface of Version 5 has been redesigned with a host of helpful features. Version 5 has improved optimization techniques and provides the programmer with effective and efficient ways handling of C++ object code. Also, this product allows the developer to carry out performance analysis to determine the applications usage of system resources.

This product is supported on IBM AIX Version 4.2.1 and later versions for RS/6000 hardware.

C++ programs written using Version 4 of IBM VisualAge Professional for AIX, and IBM C and C++ compilers, Version 3.6, and earlier, are source compatible with the VisualAge C++ Professional for AIX, Version 5.

Since the product features a batch compiler in addition to the incremental compiler, there are situations where one is more suitable than the other. Table 2 offers advice on choosing between them.

Table 2. Recommendation based on the code to be maintained

Type of project	IBM recommends
Writing new code	Incremental compiler
Maintaining projects developed with VisualAge C++ Version 4.0	Incremental compiler
Maintaining existing code from a batch environment	Batch compiler
Porting existing code from another IBM platform	Incremental compiler
Porting existing code from a non-IBM platform	Batch compiler
Developing applications for deployment on multiple flavors of UNIX	Batch compiler
Developing applications using C OpenMP support, SMP explicit directives or automatic parallellization	Batch compiler

The C compiler component of VisualAge C++ Professional, Version 5 is provided by the IBM C for AIX, Version 5 compiler.

1.3.5 C++ Compiler summary

Table 3 summarizes the various IBM C++ compiler products for AIX.

Table 3. C++ compiler products

Compiler	Installation directory	Configuration file	Supported AIX levels	Licensing Method	Drivers in /usr/bin
C Set ++ for AIX Version 3	/usr/lpp/xlC	/etc/xlC.cfg	4.1, 4.2	NetLS	Yes
IBM C and C ++ compilers Version 3.6	/usr/ibmcxx	/etc/ibmcxx.cfg	4.1.4, 4.2, 4.3	LUM	No
VisualAge C++ Professional for AIX, Version 4	/usr/vacpp	/etc/vacpp.cfg	4.1.5, 4.2, 4.3	LUM	No

Compiler	Installation directory	Configuration file	Supported AIX levels	Licensing Method	Drivers in /usr/bin
VisualAge C++ Professional for AIX, Version 5	/usr/vacpp	/etc/vacpp.cfg /etc/vac.cfg	4.2.1, 4.3	LUM	No

1.4 Installation of compiler products

The installation of the latest compiler products (C for AIX, Version 5 and VisualAge C++ Professional for AIX, Version 5) is a very simple task. There are a number of steps that need to be performed to end up with correctly installed and working compilers.

1.4.1 Install compiler filesets

The first step in the installation process is to install the compiler product filesets onto the system. The filesets to be installed will vary, depending on the compiler product and the desired configuration.

1.4.1.1 Selecting required filesets

The compiler products are delivered on CD-ROM media and are accompanied with a licence certificate for the number of licences purchased. The CD-ROM media includes the compiler filesets along with a number of other filesets, some of which are optionally installable, and some of which are co-requisites of the compiler filesets and are installed automatically. Table 4 on page 13 lists the main packages on the C for AIX, Version 5 CD-ROM

media, and Table 5 lists the main packages on the VisualAge C++ Professional for AIX, Version 5 CD-ROM media.

Table 4. C for AIX, Version 5 packages

Package name	Description
IMNSearch	Search engine for HTML documentation
idebug	Debugger with graphical user interface
memdbg	Memory debugging tools
vac	C compiler
xlC	C++ library (required by compiler executables)
xlsmp	Parallelization runtime component

Table 5. VisualAge C++ Professional for AIX, Version 5 packages

Package name	Description
IMNSearch	Search engine for HTML documentation
idebug	Debugger with graphical user interface
ipfx	Information Presentation tool (used for viewing manuals)
memdbg	Memory debugging tools
vac	C compiler
vacpp.Dt	Desktop integration
vacpp.cmp.batch	Batch (command line) C++ compiler
vacpp.cmp.incremental	Incremental C++ compiler
vacpp.cmp.C	C compiler integration
vacpp.dax	Data access builder
vacpp.ioc	IBM Open Class Library
vacpp.lic	Licence files
vacpp.memdbg	C++ memory debugging tools
vacpp.rescmp	Resource compiler
vacpp.vb	Visual Builder
vatools	Additional C++ development tools
xlC.adt	Additional C++ header files

Package name	Description
xlC.rte	C++ libraries
xlsmp	Parallelization runtime component

In all cases, the target AIX system should already have the bos.adt.include fileset installed, which contains the system provided header files. The other filesets in the bos.adt package contain useful tools and utilities often used during application development; so, it is a good idea to install the entire package. Neither the bos.adt package or bos.adt.include fileset is installed by default when installing AIX on a machine. If your system does not have the filesets installed, you will need to locate your AIX installation media and install them prior to installing the compilers since these filesets are AIX version specific and are not supplied on the compiler CD-ROM product media.

When installing the C for AIX, Version 5 product, installing the vac.C fileset will automatically install the minimum of additional required filesets. The additional filesets you may wish to install are the documentation filesets.

When installing the VisualAge C++ Professional for AIX, Version 5 product, the choice of filesets will depend on whether you wish to install the batch (command line) C++ compiler, incremental C++ compiler, C compiler, or a combination of the three.

For simple C++ command line compiles, installing the vacpp.cmp.batch fileset will automatically include the minimum required filesets. Additional filesets can be selected, depending on the type of development work being done, such as vacpp.vb for installing the components used for building applications using the Visual Builder component. Regardless of whether you are using the incremental of batch compiler, ensure that the vacpp.lic fileset is installed, as this contains the licence files required when activating the compiler.

Regardless of the product or required configuration, the filesets can be installed using one of two methods discussed in the following sections.

1.4.1.2 Install using Web-based System Manager

If your system has a graphical user interface, the filesets can be installed using the `wsm` command. The procedure is as follows:

1. Log in as the root user.
2. Insert the product CD in the CD-ROM drive.
3. Start the software installation taskguide with the following command:

```
# wsm install
```

4. From the Software drop-down menu, select **New Software (Install/Update) > Install Additional Software (Custom).**
5. In the Install Additional Software dialog, select the CD-ROM device as the software source. Then, select to install specific software available from the software source.
6. Select the **Browse** button to generate a list of software on the media.
7. Select the desired filesets from the dialog. Press and hold down the **Control** button while pressing the **mouse** button to select one or more additional objects.
8. Select the **OK** button once you have selected the desired filesets to return to the Software Install dialog.
9. Select the **OK** button to start the install.
10. Select the **YES** button to continue with the install. A popup window will appear and show the output of the installation process.
11. Select the **Close** button once the installation has completed.

1.4.1.3 Install using SMIT

If your system does not have a graphical user interface, or you do not wish to use a Web-based System Manger, you can install the required filesets using the smit command as follows:

1. Log in as the root user.
2. Insert the product CD in the CD-ROM drive.
3. Start the SMIT dialog with the following command:

```
# smit install_latest
```

4. Press the **F4** key to generate a list of possible input devices.
5. Select the CD-ROM device.
6. Press the **F4** key to generate a list of available filesets.
7. Select the required filesets by highlighting them and then pressing the **F7** key.
8. Press the **Enter** key once the required filesets have been selected.
9. Press the **Enter** key to start the install.
10. Press the **Enter** key to continue the install.
11. Press the **F10** key to exit once the installation has completed.

Note

The compiler products can not be used immediately after installation. Prior to invoking the compiler, a product licence must be enrolled with the License Use Management (LUN) system.

1.5 Activating the compilers

Once you have installed the desired compiler filesets onto the system, the next step in the process is to enroll a licence for the product into the LUM system. This section describes the process of configuring a LUM server and enrolling a product licence. If you already have a LUM environment enabled, you may go directly to Section 1.7, “Enrolling a product license” on page 20.

1.5.1 What is LUM

IBM License Use Management Runtime, referred to hereafter as License Use Management (LUM), contains the tools needed in an end user environment to manage product licenses and to get up-to-date information about license usage.

LUM is the replacement for the iFOR/LS and Net/LS systems that were used in previous versions of AIX and with previous versions of the IBM compilers.

The LUM runtime is included with AIX, Version 4.3 and higher and is automatically installed. A comprehensive description of the functionality of LUM can be found in the LUM online documentation supplied on the AIX 4.3 product media in the ifor_ls.html.en_US.base.cli fileset. The documentation fileset is not automatically installed when installing AIX; so, you will have to obtain your AIX installation media in order to install it.

For AIX, Version 4.1 and AIX, Version 4.2 systems, you must obtain the runtime and documentation filesets and manually install them. These filesets are available from several resources:

- On the Web at:

  `http://www.software.ibm.com/is/lum`

- Anonymous FTP from the server ftp.software.ibm.com. Log in with the userid *anonymous* and enter your e-mail address as the password. Change directory to software/lum/aix. The LUM run-time installation images (ARK) are contained in the sub-directory “ark”. The LUM documentation is contained in the sub-directory “doc”.

1.5.2 Configuring LUM

After installing the LUM runtime images, normally one or more LUM license servers need to be configured. No license server needs to be configured if the licensed product supplies a simple nodelock license certificate. Both the C for AIX, Version 5 and VisualAge C++ Professional for AIX, Version 5 compiler products supply a simple nodelock license certificate.

The simplest method of licensing the latest compiler products is to use the simple nodelock license certificate. When this is done, there is no need to configure a LUM server; however, the installation of the certificate in large numbers of machines can be cumbersome.

If you wish to use the simple nodelock certificate, you can skip directly to Section 1.7, “Enrolling a product license” on page 20. If you wish to use the additional functionality available when using a license server, then the first step is to decide which server type is best suited for your environment.

There are two types of license servers:

1. Concurrent nodelock license server
2. Concurrent network license server

A *concurrent nodelock license server* supports concurrent nodelock product licenses. A concurrent nodelock license is local to the node where the LUM enabled product has been installed. It allows a limited number of simultaneous users to invoke the enabled licensed product on the local system.

A *concurrent network license server* supports concurrent network product licenses. A concurrent network license is a network license that can temporarily grant a user on a client system the authority to run a LUM enabled product.

Either or both of the above license servers may be configured on a single system. The number of concurrent users for the product is specified during the enrollment of the product license certificate described in Section 1.7, “Enrolling a product license” on page 20.

The advantage of using a concurrent nodelock license server is that the server is installed on the same machine as the compiler, and, therefore, users can obtain compiler licenses even if the machine is temporarily disconnected from the network. The disadvantage, however, is that installation of licenses is cumbersome in environments with a large number of client machines.

The main advantage of using a central network license server is that the administration of product licenses is very simple. The disadvantage is that client machines must be able to contact the license server in order to use the licensed products.

Configuring LUM requires answering several questions on how you would like to set up the LUM environment. It is recommended that users read the LUM documentation supplied with the AIX product media prior to configuring LUM.

A LUM server can be configured in several different ways. You can issue commands on the command line with appropriate arguments to configure the LUM server. You can issue a command that starts a dialog and asks a number of questions to determine the appropriate configuration, or you can configure the server using a graphical user interface.

1.5.2.1 Configuring a nodelock server

For small numbers of client machines (typically 10 or less), using a nodelock license server on each machine is the simplest method of configuring LUM.

Log in as the root user and perform the following commands to configure a machine as a nodelock license server:

```
# /var/ifor/i4cfg -a n -S a
# /var/ifor/i4cfg -start
```

The first command configures the local machine as a nodelock license server and sets the option that the LUM daemons should be started automatically when the system boots. The second command starts the LUM daemons.

1.5.2.2 Using the interactive configuration tool

As an alternative to using the above commands, you can use the /var/ifor/i4config interactive configuration script to perform the same actions.

1. Log in as userid `root` on the system where the license server will be installed.
2. Enter `cd /var/ifor`. If this directory does not exist, then LUM has not been installed.
3. Invoke the LUM configuration tool by entering the command, `./i4config`. This is the command line version of the LUM configuration tool.
4. Answer the LUM configuration questions as appropriate. The answers to the configuration questions are dependent on the LUM environment you wish to create.

The following are typical answers to the configuration questions of LUM in order to configure both concurrent nodelock and concurrent network license servers on a single system. You may change the various answers accordingly to suit your preferred system environment. For details on configuring LUM, please read the documentation that comes with LUM.

- Select 4 “Central Registry (and/or Network and/or Nodelock) License Server” on the first panel.
- Answer y to “Do you want this system be a Network License Server too?”
- Select 2 “Direct Binding only” as the mechanism to locate a license server.
- Answer n to “Do you want to change the Network License Server ip port number?”
- Answer n to “Do you want to change the Central Registry License Server ip port number?”
- Answer n to “Do you want to disable remote administration of this Network License Server?”
- Answer y to “Do you want this system be a Nodelock License Server too?”
- Select 1 “Default” as the desired server(s) logging level.
- Enter blank to accept the default path for the default log file(s).
- Answer y to “Do you want to modify the list of remote License Servers this system can connect to in direct binding mode (both for administration purposes and for working as Network License Client)?”
- Select 3 “Create a new list” to the direct binding list menu.
- Enter the hostname, without the domain, of the system you are configuring LUM when prompted for the “Server network name(s).”
- Answer n to “Do you want to change the default ip port number?”
- Answer y to “Do you want the License Server(s) automatically start on this system at boot time?”
- Answer y to continue the configuration setup and write the updates to the i4ls.ini file.
- Answer y to “Do you want the License Server(s) start now?”

Both concurrent nodelock and concurrent network license servers should now be configured on your system.

For more information on configuring and using LUM, refer to the LUM documentation supplied with AIX. As an alternative, the LUM manual, *Using*

License Use Management Runtime for AIX, SH19-4346, can be viewed online in PDF format at the following URL:

```
ftp://ftp.software.ibm.com/software/lum/aix/doc/V4.5.5/lumusgaix.pdf
```

1.6 Activating the LUM server

After configuring and starting the LUM server, you can enroll product licenses. Before attempting to enroll a license, you must first ensure that the LUM daemons are active. This can be done with the following command:

```
# /var/ifor/i4cfg -list
```

Depending on the type of LUM server configured, the output will be similar to the following:

```
i4cfg Version 4.5 AIX -- LUM Configuration Tool
(c) Copyright 1995-1998, IBM Corporation, All Rights Reserved
US Government Users Restricted Rights - Use, duplication or disclosure
restricted by GSA ADP Schedule Contract with IBM Corp.
Subsystem         Group            PID     Status
 i4llmd           iforls           22974   active
```

If no subsystem is listed as active, then start them with the following command:

```
# /var/ifor/i4cfg -start
```

The only daemon that must be active is the Nodelock License Server Subsystem (i4llmd) daemon. The other daemons that may be active depending on your configuration are as follows:

- License Sever Subsystem (i4lmd)
- Central Registry Subsystem (i4gdb)
- Global Location Broker Data Cleaner Subsystem (i4glbcd)

1.7 Enrolling a product license

After LUM has been installed and configured on your system, the product license certificates can be enrolled with the LUM license server. Three LUM product license certificates are provided with each of the latest compiler products:

1. Concurrent nodelock license certificate
2. Concurrent network license certificate

3. Simple nodelock license certificate

You should enrol the appropriate license certificate for the type of LUM environment you have configured.

The locations of the license certificates for the compiler products are detailed in Table 6.

Table 6. License certificate locations

Compiler	License Certificate Type	Location
C for AIX Version 5	Concurrent Network	/usr/vac/cforaix_c.lic
	Concurrent Nodelock	/usr/vac/cforaix_cn.lic
	Simple Nodelock	/usr/vac/cforaix_n.lic
VisualAge C++ Professional for AIX Version 5	Concurrent Network	/usr/vacpp/vacpp_c.lic
	Concurrent Nodelock	/usr/vacpp/vacpp_cn.lic
	Simple Nodelock	/usr/vacpp/vacpp_n.lic

1.7.1 Enrolling a concurrent license

To enroll a Concurrent Network or Concurrent Nodelock license certificate, perform the following steps:

1. Log in as root on the system where the license server is installed.
2. Invoke the LUM configuration tool by entering the LUM Basic License Tool command as follows:

```
/var/ifor/i4blt
```

The i4blt tool contains both a graphical user interface and a command line interface. Note that the LUM daemons must be running before starting the i4blt tool. Refer to Section 1.6, “Activating the LUM server” on page 20 for information on how to check the status of the LUM daemons.

If the X11 runtime (X11.base.rte fileset) has been installed on your system, the GUI version of the tool will be invoked. Otherwise, the command line version will be invoked, and an error will occur since the appropriate command line parameters were not specified.

The following are the instructions for both interfaces using the i4blt tool:

- Enrolling using the graphical user interface:
  - Select the **Products** pull-down and click on **Enroll Product** item.

- Click on the **Import** button. The Import panel should be displayed.
- In the Filter entry prompt, enter /usr/vacpp/*.lic if you are enrolling a license for VisualAge C++ or /usr/vac/*.lic if you are enrolling a license for C for AIX, and press **Enter**. This will show the various product license files in the Files panel. The three license files for the product, as detailed in Table 6, should be displayed.
- Select either the *prod*_c.lic or *prod*_cn.lic (where *prod* is either vacpp or cforaix) license by clicking on the entry.
- Click **OK**. The Enroll Product panel should be re-displayed with information regarding the product indicated.
- Click on the **OK** button of the Enroll Product panel. The Enroll Licenses panel should be displayed.
- Fill in the information on the Administrator Information portion of the panel (optional.)
- Fill in the number of valid purchased licenses of the product under Enrolled Licenses in the Product information portion of the panel. (mandatory.)
- Click on the **OK** button of the Enroll Licenses panel. The product should be successfully enrolled. You may terminate the i4blt tool.

• Enrolling using the command line:

- From the required product license file, as detailed in Table 6, extract the `i4blt` command from the top of the file.
- Replace number_of_lics from the command with the number of valid purchased licenses of the product (mandatory.)
- Replace admin_name with the name of the administrator (optional.)
- Invoke this command as root from /var/ifor. The product should be successfully enrolled.

1.7.2 Enrolling a simple nodelock license

Read the instructions at the top of the simple nodelock license certificate file. In general, this type of license will be installed when no LUM system has been configured. This means enrolling the license is simply a case of placing the indicated license information line into the /var/ifor/nodelock LUM nodelock file.

1.8 Invoking the compilers

Once a compiler product license has been enrolled, you are now ready to use the compilers. As mentioned in Section 1.1.1.1, “Finding the compiler drivers” on page 1, the compiler drivers are not installed in a directory that is searched with the default PATH environment variable. There are a number of methods of resolving this issue:

- If you do not have a previous version of the compiler installed, as the root user invoke the replaceCSET script supplied with the compiler.
- Add the directory containing the compiler drivers to the default PATH environment variable set in the /etc/environment configuration file.
- Add the directory containing the compiler drivers to the PATH environment variable in each users .profile shell configuration file.
- Change the Makefiles used in your development environment to configure the compiler macro to use the absolute path. For example:

```
CC=/usr/vac/bin/cc
```

Using the replaceCSET script is the preferred option since it resolves the problem for all users after a simple single action by the root user.

1.8.1 Default compiler drivers

The Version 5 compiler products include a number of default compiler configurations in the /etc/vac.cfg compiler configuration file. The default C++ command line driver is /usr/vacpp/bin/xlC. The three main C compiler command line drivers are as follows:

/usr/vac/bin/cc Extended mode C compiler.

/usr/vac/bin/xlc ANSI C compiler, using UNIX header files.

/usr/vac/bin/c89 ANSI C compiler, using ANSI C header files.

There are a number of additional command line drivers available, each one based on the basic cc, xlc and xlC drivers described above. They are described in Table 7.

Table 7. Compiler driver extensions

Command extension	Meaning
_r	Use the UNIX98 threads libraries
_r7	Use the POSIX Draft 7 threads libraries
_r4	Use the POSIX Draft 4 (DCE) threads libraries

Command extension	Meaning
128	Enable 128 bit double precision floating point values and use appropriate libraries.
128_r	Enable 128 bit double precision floating point values and use the UNIX98 threads libraries.
128_r7	Enable 128 bit double precision floating point values and use the POSIX Draft 7 threads libraries.
128_r4	Enable 128 bit double precision floating point values and use the POSIX Draft 4 (DCE) threads libraries.

For example, to compile an ANSI C program using Draft 7 of the POSIX threads standard, use the xlc_r7 compiler driver. To compile a C++ program that uses 128 bit floating point values, use the xlC128 compiler driver.

1.9 Online documentation

The Version 5 compilers come with online documentation that is written in HTML format. The default configuration makes it very easy to view the online documentation on the machine on which it is installed.

1.9.1 Viewing locally

The procedure for viewing the documentation installed on the local machine depends on a number of factors, including which compiler product is installed and whether you are using the AIX Common Desktop Environment.

1.9.1.1 C compiler documentation

The C for AIX Version 5 compiler documentation is written in HTML format. The HTML files are located in the /usr/vac/html directory. To view the documentation, start the Netscape browser supplied with the AIX Bonus Pack and point it at the following file:

```
/usr/vac/html/en_US/doc/index.htm
```

Before starting Netscape, ensure that the environment variable, SOCKS_NS, is not set. For the search facility to work correctly, the browser must not have proxy handling enabled for the localhost port. To disable proxy handling for the local host when using Netscape:

1. Start the browser, then select **Edit->Preferences** from the menu.
2. Double-click **Advanced** in the Category tree.
3. Click **Proxies** in the Advanced subtree.

4. Click **View** at the Manual Proxy Configuration selection.
5. Type the following in the “Do not use proxy servers for domains beginning with:” box:

   ```
   localhost:49213
   ```

   If there are other entries in the box, separate the new entry with a comma.
6. Click **OK**, then click **OK** to exit the Preferences window.

1.9.1.2 C++ compiler documentation

The VisualAge C++ Professional for AIX, Version 5 compiler documentation is written in HTML format. The HTML files are stored in a single file in ZIP format. The files are viewed using an HTML browser, which uses a cgi-bin script to extract and view the required files. There is no need to manually unpack the ZIP file.

If you are using the AIX CDE interface, the C++ compiler documentation can be started by double-clicking on the **Help Homepage** icon in the VisualAge C++ Professional folder of the Application Manager.

If you are not using the AIX CDE interface, or are logged in remotely from another X11 capable display, then use the following command:

```
/usr/vacpp/bin/vacpphelp
```

The command starts the default Netscape browser (which is supplied on the AIX Bonus Pack media) with the correct URL.

1.9.2 Viewing remotely

By default, it is not possible to view the online documentation from a remote machine. It can be done in a simple way by logging in to the machine that has the documentation installed, set the DISPLAY environment variable to use a remote X11 display, then view the documentation by invoking the same command used to view locally.

A better solution, particularly in larger environments or where remote clients do note have X11 capabilities, is to configure the machine to allow remote viewing of the documentation. This can be performed as shown in the following sections.

1.9.2.1 Configuring the HTTP server

Suppose the machine that has the documentation filesets installed has a fully qualified domain name of docs.ibm.com. The following example demonstrates

the steps performed on that machine to allow remote clients to view the compiler documentation using their HTML browser:

1. Log in as the root user.
2. Perform the following command:

   ```
   cp /etc/IMNSearch/httpdlite/httpdlite.conf /etc/IMNSearch/httpdlite/vacpp.conf
   ```

3. Edit /etc/IMNSearch/httpdlite/vacpp.conf, and make the following changes:
   a. Change the HostName line from:

      ```
      HostName localhost
      ```

      to:

      ```
      HostName docs.ibm.com
      ```

      If the HostName line is not present, or has a comment symbol (#) at the start of the line, then simply add the following line to the file:

      ```
      HostName docs.ibm.com
      ```

   b. Change the Port line from:

      ```
      Port 49213
      ```

      to:

      ```
      Port 49214
      ```

   c. If the version of IMNSearch.rte.httpdlite installed on your machine is greater than 2.0.0.0, you will need to add one or more Allow lines to specify which hosts are permitted to access the Web server. The Allow statement has the following syntax:

      ```
      Allow network-ip network-mask
      ```

      A client is only granted access if the following rule is met: (& is a bitwise AND operation)

      ```
      client-ip & network-mask == network-ip & network-mask
      ```

      For example, if you wanted machines with an address, such as 9.x.x.x, to be able to access the help server, you would add the following statement to vacpp.conf:

      ```
      Allow 9.0.0.0 255.0.0.0
      ```

   d. Save the file and exit the editor.
4. Edit the file /etc/inittab. There is a line that executes the `httpdlite` command with a filename argument. The line is as follows:

   ```
   httpdlite:2:once:/usr/IMNSearch/httpdlite/httpdlite -r
   /etc/IMNSearch/httpdlite/httpdlite.conf >/dev/console 2>&1
   ```

Make a copy of this line immediately below the original line. In the new line:

a. Change the first field from httpdlite to httpdlite2.

b. Change the part of the line that reads httpdlite.conf to vacpp.conf

The result should be as follows:

```
httpdlite2:2:once:/usr/IMNSearch/httpdlite/httpdlite -r
/etc/IMNSearch/httpdlite/vacpp.conf >/dev/console 2>&1
```

Save the file and exit from the editor.

5. Reboot the system or run the following command to start the second copy of the ICS lite server:

```
/usr/IMNSearch/httpdlite/httpdlite -r
/etc/IMNSearch/httpdlite/vacpp.conf >/dev/console 2>&1
```

The steps described above configure an instance of an HTTP server to respond on a specific port number to requests to access compiler documentation.

The following sections detail the additional steps required to configure the documentation for each compiler product to be served by the HTTP server.

1.9.2.2 Configuring the C++ documentation

The following steps are required to enable the online documentation for the VisualAge C++ Professional for AIX, Version 5 compiler to be served by the HTTP server:

1. Log in as the root user.
2. Change directory to /var/vacpp/en_US
3. Edit the file hgssrch.htm, and change the line:

   ```
   <form ACTION="http://localhost:49213/cgi-bin/vacsrch.exe" METHOD="POST">
   ```

   to:

   ```
   <form ACTION="http://docs.ibm.com:49214/cgi-bin/vacsrch.exe" METHOD="POST">
   ```

   Then, save the file and exit the editor.

4. Issue the following command:

   ```
   /usr/IMNSearch/cli/imndomap -u "VACENUS"
   "http://docs.ibm.com:49214/cgi-bin/vahwebx.exe/en_US/vacpp/Extract/0/"
   "VisualAge C++"
   ```

5. Users can point their browser at the following URL to browse and search the documentation:

```
http://docs.ibm.com:49214/cgi-bin/vahwebx.exe/en_US/vacpp/Extract/0/index.htm
```

1.9.2.3 Configuring the C compiler documentation

The following steps are required to enable the online documentation for the C for AIX, Version 5 compiler to be served by the HTTP server:

1. Log in as the root user.
2. Change directory to /usr/docsearch/html.
3. Perform the following command:

   ```
   ln -s /usr/vac/html/en_US/doc vac_doc
   ```

4. Edit the file /usr/vac/html/en_US/doc/hgssrch.htm and change the line:

   ```
   <form ACTION="http://localhost:49213/cgi-bin/caixsrch.exe" METHOD="POST">
   ```

   to:

   ```
   <form ACTION="http://docs.ibm.com:49214/cgi-bin/caixsrch.exe" METHOD="POST">
   ```

   Then, save the file and exit the editor.

5. Issue the following command:

   ```
   /usr/IMNSearch/cli/imndomap -u "CENUS"
   "http://docs.ibm.com:49214/vac_doc/" "C for AIX"
   ```

6. Users can point their browser at the following URL to browse and search the documentation:

   ```
   http://docs.ibm.com:49214/vac_doc/index.htm
   ```

1.10 Additional developer resources

IBM maintains many Web sites that provide useful information for developers using the AIX platform. The most important ones are described in the following sections.

1.10.1 AIX operating system documentation

The online documentation for the AIX operating system can be viewed at the following URL:

```
http://www.rs6000.ibm.com/library
```

The site contains up-to-date versions of the HTML documentation supplied with the AIX product media.

As new releases of the AIX operating system become available, they generally add new functionality. As a developer, you might wish to use some

of the new functionality, but the decision to do so may also be based on the minimum level of AIX required to use a particular feature. The IBM Redbook, *AIX Version 4.3 Differences Guide*, SG24-2104, is updated with each new release of AIX, and contains information on when particular features were introduced.

1.10.2 Compiler product information

The latest compiler products both have support Web sites that contain useful hints, tips, frequently asked questions, and links to other useful Web sites. The support page for the VisualAge C++ Professional for AIX, Version 5 compiler is:

`http://www-4.ibm.com/software/ad/vacpp/support.html`

The support page for the C for AIX, Version 5 compiler is:

`http://www-4.ibm.com/software/ad/caix/support.html`

Information on the availability of IBM products for the AIX operating system, along with details of when support for products will be withdrawn, is available on the following Web site:

`http://www.ibm.com/servers/aix/products/ibmsw/list/`

1.10.3 PartnerWorld for developers

PartnerWorld for Developers is a worldwide program supporting developers who build solutions using IBM technologies. The program covers all IBM platforms, not just AIX. Its Web site contains a lot of useful information for the AIX developer, including white papers, sample code, and technology articles. It can be located on the Web at the following URL:

`http://www.developer.ibm.com`

Chapter 2. Shared memory

The Inter-Process Communication (IPC) facilities in the UNIX operating system are used by processes to communicate with each other and to synchronize their activities. Semaphores, signals, and message queues are common methods of inter-process communication.

A memory mapping mechanism, also provided by the UNIX system, allows programs to exchange data and synchronization by accessing the same memory address space. A beneficial side-effect is that this mechanism provides a very fast channel for data exchange when compared with other kinds of IPC methods that are kernel related, which means ultimately that they trigger system call operations.

The memory mapping technique can be used for:

Mapping files

Memory mapped files provide a mechanism for a process to access files by directly incorporating file data into the process address space. The use of mapped files can significantly reduce I/O data movement since the file data does not have to be copied into process data buffers, as is done by the read and write subroutines. When more than one process maps the same file, its contents are shared among them, therefore, providing a low-overhead mechanism by which processes can synchronize and communicate.

Mapping regions of memory

Mapped memory regions, also called shared memory areas, can serve as a large pool for exchanging data among processes. The available mechanics do not provide locks or access control among the processes. Therefore, processes using shared memory areas must set up a signal or semaphore control method to prevent access conflicts and to keep one process from changing data that another is using. Shared memory areas can be most beneficial when the amount of data to be exchanged between processes is too large to transfer with messages or when many processes maintain a common large database.

Figure 1 on page 32 shows two generic processes, where process B reads the raw data from the input file, executes some transformations on this data, and sends it to the process A through an IPC mechanism, such as a pipe or message queue.

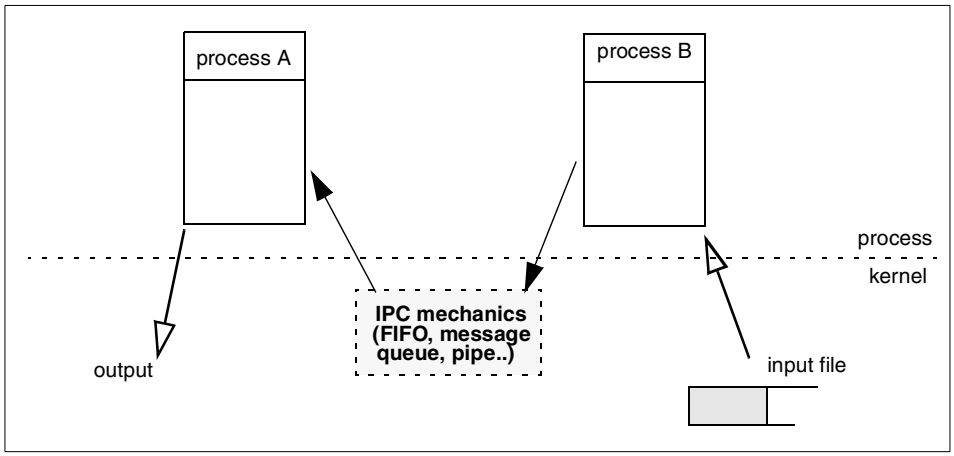


Figure 1. IPC communication thorough kernel

By comparison, Figure 2 shows a more efficient implementation, where both processes exchange data through a shared memory region. Another great improvement, regarding performance, in this implementation is mapping the input file that incorporates raw data directly into the process data space.

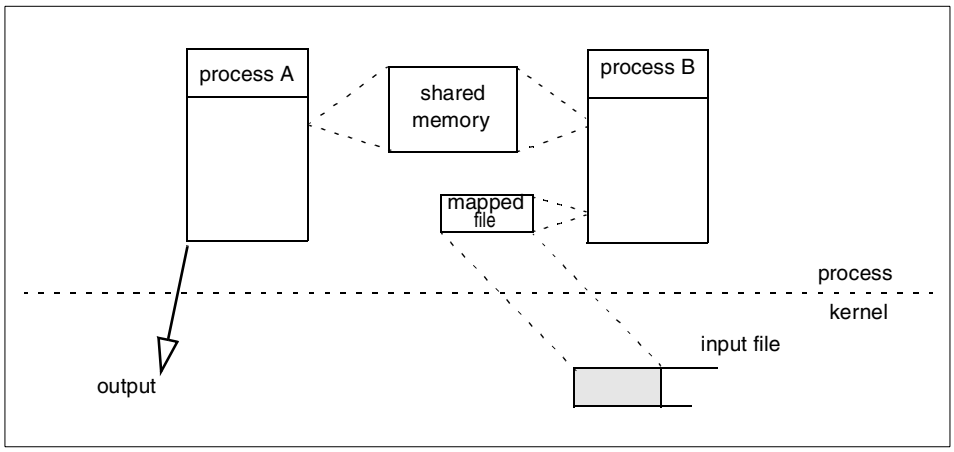


Figure 2. IPC communication through shared memory

2.1 Program address space

The UNIX system employs a memory management scheme that uses software to extend the capabilities of the physical hardware. Because the address space does not correspond one-to-one with real memory, the

address space (and the way the system makes it correspond to real memory) is called *virtual memory*.

The subsystems of the kernel and the hardware that cooperate to translate the virtual addresses to physical addresses make up the memory management subsystem. The actions the kernel takes to ensure that processes share main memory fairly comprise the *memory management policy*.

2.1.1 The physical address space of a 32-bit system

The hardware provides a continuous range of virtual memory, `0x0000000000000` to `0xFFFFFFFFFFFFF`, for accessing data, which gives a total addressable space with more than 1,000 terabytes. This range of memory requires a 52 bit representation. As the memory access instructions generate addresses of 32 bits, a segment::offset mechanism is implemented to provide the mapping of virtual to physical addresses. Figure 3 shows this.

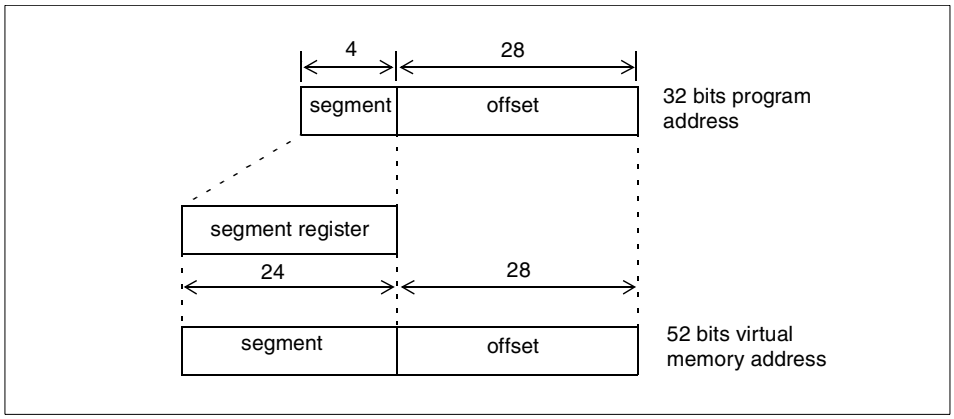


Figure 3. 32 bits segment register addressing

This addressing scheme provides access to 16 segments of up to 256 MB each. Each segment register contains a 24-bit segment ID that becomes a prefix to the 28-bit offset, which together form the virtual memory address.

The process space is a 32-bit address space; that is, programs use 32-bit pointers. However, each process or interrupt handler can address only the system-wide virtual memory space (segment) whose segment IDs are in the segment register. A process can access more than 16 segments by changing registers rapidly.

2.1.2 Segment Register addressing

The system kernel loads some segment registers in the conventional way for all processes, implicitly providing the memory addressability needed by most processes. These registers include two kernel segments and a shared library text segment that are shared by all processes and whose contents are read-only to non-kernel programs. There is also a read-only segment for the machine code instructions (text) of a process, which is shared with other processes executing the same program. There is also a private shared library data segment that contains read-write library data, and a read-write segment that is private to the process and contains the processes stack, initialized data, and memory heap. The remaining segment registers may be utilized using *memory mapping* techniques to provide more memory, or through memory access to files according to access permissions imposed by the kernel. Figure 4 shows a schematic representation of the segment registers used in addressing in a 32 bit environment.

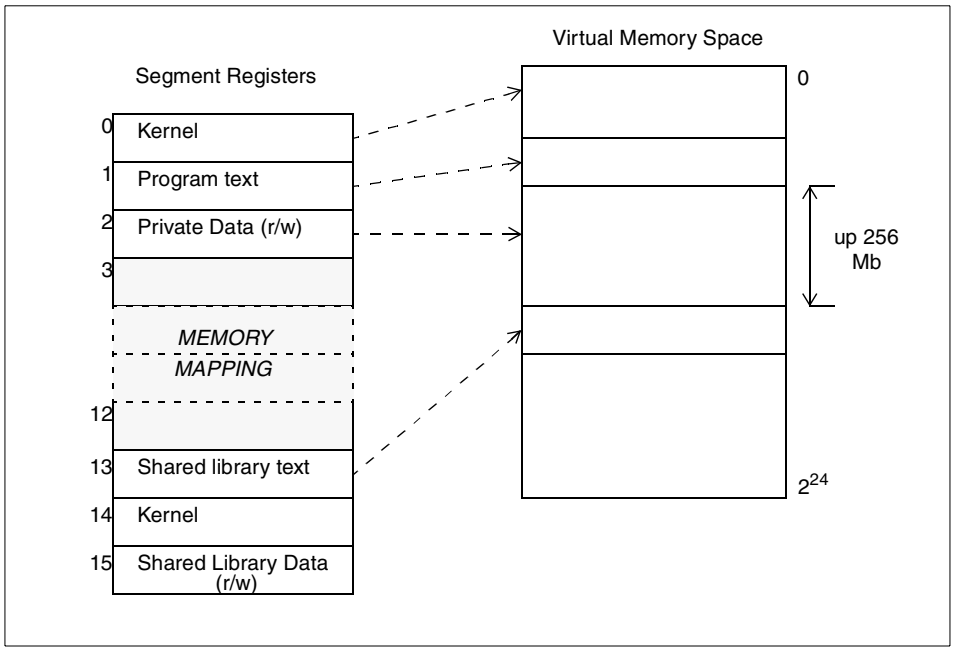


Figure 4. 32 bit process-view of system virtual memory space

2.2 Memory mapping mechanics

AIX Version 4.3 supports the following two interfaces for memory mapping:

- The shmap interfaces
- The mmap interfaces

Both mechanisms address the same kind of applications, but normally shmap is used to create and use shared memory segments from within a program, and the mmap model is mostly used for mapping existing files into the process address space, although it can be used for creating shared memory regions as well as mapping files.

2.2.1 The shmap interfaces

AIX, Version 4 supports the shmap implementation of memory mapping through the following functions:

shmat Attaches a shared memory segment to a process.

shmctl Controls shared memory operations.

shmget Gets or creates a shared memory segment.

shmdt Detaches a shared memory segment from a process.

disclaim Removes a mapping from a specified address range within a shared memory segment.

ftok Provides the key that the shmget subroutine uses to create the shared segment.

The shmat subroutine attaches a shared memory segment or a mapped file to the address space of the calling process.

Some limits apply to shared memory capacity according to the version of AIX in use. The limits are shown in Table 8.

Table 8. Limitations of shmap on AIX

	AIX 4.2.1	**AIX 4.3.1**	**AIX 4.3.2**
Maximum segment size	256 MB	2 GB	2 GB
Maximum number of shared IDs	4096	131072	131072
Maximum address space for mapping	2.75 GB	2.75 GB	2.75 GB

In versions of AIX up to Version 4.2.0, there was a limitation on the number of memory segments that could be attached using the shmat subroutine. A process could attach no more than 10 shared memory regions at any one

time since each attached region consumed a segment register, regardless of the size of the memory region being attached. This limitation sometimes caused problems for developers who were porting applications to AIX from other platforms that allowed more than 10 shared memory regions at a time, and the architecture of the application required this functionality. In the past, significant modifications would be required to the application to re-engineer the design to work correctly with 10 or fewer shared memory regions.

For processes on AIX, Version 4.2.1 and later releases, an extended shmat capability is available. This capability allows a process to attach more than 11 shared memory regions at any one time. This capability is enabled by setting an environment variable as follows:

```
EXTSHM=ON
```

The segments can be of size from 1 byte to 256 MB, and they are attached into the address space for the size of the segment. No source code modifications are required to use this functionality; so, developers porting applications to AIX are no longer restricted by the number of simultaneously attached shared memory regions.

Note

If a single shared memory region larger than 256 MB is used, the system automatically works as if the EXTSHM environment variable is set, even if it was not defined.

When this functionality is enabled, the attached shared memory regions are tracked using an extra level of indirection rather than consuming a segment register for each attached region.

2.2.1.1 Mapped memory with shmat

In the following, we show a step-by-step process for creating and using mapped memory, or shared memory, segments:

1. Create a key to uniquely identify the shared segment. Use the ftok subroutine to create the key. For example, to create the key, mykey, using a project ID in the variable proj (integer type) and a file name of null_file, use a statement, such as:

   ```
   mykey = ftok( null_file, proj );
   ```

2. Either:

a. Create a shared memory segment with the shmget subroutine. For example, to create a shared segment that contains 4096 bytes and assign its ID to an integer variable, mem_id, use a statement, such as:

```
mem_id = shmget(mykey, 4096, IPC_CREAT | o666 );
```

b. Get a previously created shared segment with the shmget subroutine. For example, to get a shared segment that is already associated with the key, mykey, and assign the ID to an integer variable, mem_id, use a statement, such as:

```
mem_id = shmget( mykey, 4096, IPC_ACCESS );
```

3. Attach the shared segment to the process with the shmat subroutine. For example, to attach a previously created segment, use a statement, such as:

```
ptr = shmat( mem_id );
```

   In this example, the variable `ptr` is a pointer to a structure that defines the fields in the shared segment. Use this template structure to store and retrieve data in the shared segment. This template should be the same for all processes using the segment.

4. Work with the data in the segment.

5. Detach from the segment using the shmdt subroutine:

```
shmdt( ptr );
```

6. If the shared segment is no longer needed, remove it from the system with the shmctl subroutine:

```
shmctl( mem_id, IPC_RMID, ptr );
```

Next, we show a very simple example where one program, *crshmen.c*, creates and sets some data in a shared memory segment, and a second program, *rdshmen.c*, reads it.

The crshmen.c code is as follows:

```
#include <sys/shm.h>       /* include the proper headers */
#include <stdio.h>
                           /* defines the struct type for*/
                           /* using in shared memory     */
typedef struct idex_t{
                     char  name[80];
                     char  version[10];
                     int   year;
               } idex;
main()
{
```

```
key_t   mykey;             /* key identifier variable    */
idex   *ptr;               /* pointer to the struct      */
int     mem_id;            /* memory ID                  */
int     seg_id = 1;        /* project identifier         */

                           /* creates the key            */
mykey = ftok( null_file, seg_id);

                           /* creates the shared segment */
mem_id = shmget(mykey, sizeof( idex), IPC_CREAT | S_IROT | S_IWOTH );

                           /* attach it to the process   */
    ptr = shmat( mem_id, NULL, 0);

                           /* handle the data as need    */
    strcpy( ptr->name, "C&C++ Application Development on AIX" );
    strcpy( ptr->version, "1.0");
    ptr->year = 2000;
                           /* dettach from the segment   */
    shmdt( ptr );
}
```

And, next the rdshmen.c code:

```
#include <sys/shm.h>       /* include the proper headers */
#include <stdio.h>
                           /* defines the struct type for*/
                           /* using in shared memory     */
                           /* it must be the same as when*/
                           /* segment was created        */
typedef struct idex_t{
                     char  name[80];
                     char  version[10];
                     int   year;
               } idex;
main()
{
key_t   mykey;             /* key identifier variable    */
idex   *ptr;               /* pointer to the struct      */
int     mem_id;            /* memory ID                  */
int     seg_id = 1;        /* project identifier         */

                           /* creates the key that matchs*/
                           /* to the original segment    */
    mykey = ftok( null_file, seg_id);

                           /* gets access to the segment */
```

```
    mem_id = shmget( mykey, sizeof( idex), IPC_ACCESS );

                          /* attach it to the process   */
    ptr = shmat( mem_id, NULL, 0);

                          /* handle the data as need    */
    printf(" Book name: %s\n", ptr->name);
    printf("   version: %s\n", ptr->version);
    printf("      year: %d\n", ptr->year);
                          /* dettach from the segment   */
    shmdt( ptr );
}
```

2.2.1.2 Mapped files with shmat

The creation of a mapped data file is a two-step process. First, you create the mapped file. Then, because the shmat subroutine does not provide for it, you must program a method for detecting the end of the mapped file.

1. To create the mapped data file:
   a. Open (or create) the file and save the file descriptor:

      ```
      if( ( fildes = open( filename , 2 ) ) < 0 )
                {
                        printf( "cannot open file\n" );
                        exit(1);
                }
      ```

   b. Map the file to a segment with the shmat subroutine:

      ```
      file_ptr=shmat (fildes, 0, SHM_MAP);
      ```

      The SHM_MAP constant is defined in the /usr/include/sys/shm.h file. This constant indicates that the file is a mapped file. Include this file and the other shared memory header files in a program with the following directives:

      ```
      #include <sys/shm.h>
      ```

2. To detect the end of the mapped file:
   a. Use the lseek subroutine to go to the end of file:

      ```
      eof = file_ptr + lseek(fildes, 0, 2);
      ```

      This example sets the value of `eof` to an address that is 1 byte beyond the end of file. Use this value as the end-of-file marker in the program.

   b. Use file_ptr as a pointer to the start of the data file, and access the data as if it were in memory:

      ```
      while ( file_ptr < eof)
      ```

```
{
        .
        .
        .
        (references to file using file_ptr)
}
```

c. Close the file when the program is finished working with it:

```
close (fildes );
```

2.2.2 The mmap functions

AIX Verions 4 supports the mmap version of memory mapping through the following functions:

mmap Maps an object file into virtual memory.

madvise Advises the system of a process' expected paging behavior.

mincore Determines residency of memory pages.

mprotect Modifies the access protections of memory mapping.

msync Synchronizes a mapped file with its underlying storage device.

munmap Unmaps a mapped memory region.

The mmap subroutines map a file or anonymous memory region by establishing a mapping between a process-address space and a file system object. This implies that in the mmap model there is always a file object associated on the file system, even if it is only mapping a memory segment.

2.2.2.1 Mapped memory with mmap

In the following we show the three step process for creating and using mapped memory, or shared memory, segments:

1. As, in fact, there is no file to attach when we just want to map an amount of memory, just create a shared memory segment with the mmap subroutine. For example, to create a shared starting at address addr, using len bytes of size, with the access permissions defined by prot and 0 bytes of offset, use a statement, such as:

   ```
   ptr = mmap( addr, len, prot, MAP_ANONYMOUS, -1, 0)
   ```

   This memory region pointed by ptr can be shared only with the descendants of the current process.

2. Work with the data in the segment.
3. Detach the segment from the address space using the munmap subroutine:

```
munmap( addr, len);
```

2.2.2.2 Mapped files with mmap

When we are, in fact, mapping files, there are three more steps in the process compared with mapping memory. The steps are as follows:

1. Create a file descriptor for a file system object using:

   ```
   fp = open( pathname, permissions);
   ```

2. Determine the file length by using the lseek system call. For example:

   ```
   len = lseek(fildes, 0, 2)
   ```

3. Map the file into the process address space with the mmap subroutine. For example, to map the file for the file descriptor fp, starting at address addr, using len bytes of size, with the access permissions defined by prot and 0 bytes of offset, use a statement, such as:

   ```
   ptr = mmap(addr, len, prot, MAP_FILE, fp, 0)
   ```

   This specifies the creation of a new mapped file region by mapping the file associated with the fp file descriptor. The mapped region can extend beyond the end of the file, both at the time when the mmap subroutine is called and while the mapping persists. This situation could occur if a file with no contents was created just before the call to the mmap subroutine, or if a file was later truncated.

4. The file descriptor can be closed by using:

   ```
   close( fp);
   ```

5. Work with the data in the segment.

6. Detach from the segment using the munmap subroutine:

   ```
   munmap( addr, len);
   ```

The mmap services are specified by various standards and commonly used as the file-mapping interface of choice in other operating system implementations. However, the system's implementation of the mmap subroutine may differ from other implementations. The mmap subroutine incorporates the following modifications:

- Mapping into the process private area is not supported.
- Mappings are not implicitly unmapped. An mmap operation that specifies MAP_FIXED will fail if a mapping already exists within the range specified.
- For private mappings, the copy-on-write semantic makes a copy of a page on the first write reference.
- Mapping of I/O or device memory is not supported.

- Mapping of character devices or use of an mmap region as a buffer for a read-write operation to a character device is not supported.
- The madvise subroutine is provided for compatibility only. The system takes no action on the advice specified.
- The mprotect subroutine allows the specified region to contain unmapped pages. In operation, the unmapped pages are simply skipped over.
- The OSF/AES-specific options for default exact mapping and for the MAP_INHERIT, MAP_HASSEMAPHORE, and MAP_UNALIGNED flags are not supported.

Note

A file system object should not be simultaneously mapped using both the mmap and shmat subroutines. Unexpected results may occur when references are made beyond the end of the object.

2.2.3 Comparison of shmat and mmap

Both the mmap and shmat services provide the capability for multiple processes to map the same region of an object such that they share addressability to that object. However, the mmap subroutine extends this capability beyond that provided by the shmat subroutine by allowing a relatively unlimited number of such mappings to be established. While this capability increases the number of mappings supported per file object or memory segment, it can prove inefficient for applications in which many processes map the same file data into their address space.

The mmap subroutine provides a unique object address for each process that maps to an object. The software accomplishes this by providing each process with a unique virtual address, known as an alias. The shmat subroutine allows processes to share the addresses of the mapped objects.

Because only one of the existing aliases for a given page in an object has a real address translation at any given time, only one of the mmap mappings can make a reference to that page without incurring a page fault. Any reference to the page by a different mapping (and thus a different alias) results in a page fault that causes the existing real-address translation for the page to be invalidated. As a result, a new translation must be established for it under a different alias. Processes share pages by moving them between these different translations.

For applications in which many processes map the same file data into their address space, this toggling process may have an adverse affect on

performance. In these cases, the shmat subroutine may provide more efficient file-mapping capabilities.

The overall indications for using shmat are:

- For 32-bit applications, eleven or fewer files are mapped simultaneously, and each is smaller than 256 MB.
- When mapping files larger than 256 MB.
- When mapping shared memory regions that need to be shared among unrelated processes (no parent-child relationship).
- When mapping entire files.

And, the indications for using mmap are as follows:

- Portability of the application is a concern.
- Many files are mapped simultaneously.
- Only a portion of a file needs to be mapped.
- Page-level protection needs to be set on the mapping.
- Private mapping is required.

2.3 Process private data

As we discussed in early sections, the system kernel loads some segment registers for all processes. Here, we are especially interested on the process private data segment. It is the third segment register and is next to the text of the process as shown in Figure 4 on page 34.

As a segment, its size can reach up to 256 MB and it holds the *heap, stack* and *initialized data* area for the application as shown in Figure 5 on page 44.

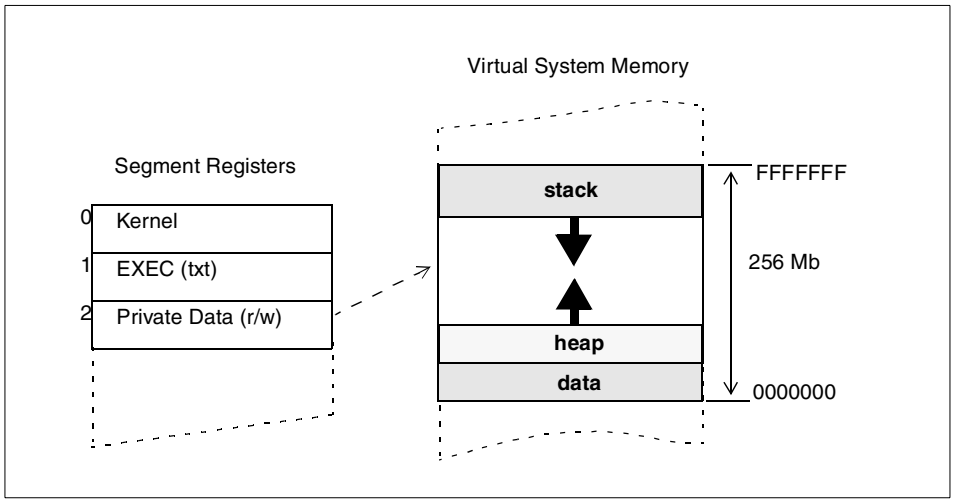


Figure 5. Default process private data area

There are three general situations where increasing the size of this area can be necessary:

- When the application has a huge initialized data area.
- When the processes memory heap grows very large due to large use of the malloc subroutine
- The stack grows very large, for example, when using recursive functions.

The AIX linker can be used to change the segment limit by using the following argument at link time:

`-bMAXDATA:`*`number`*

where *`number`* indicates how many extra segments to use for the R/W private data during the program load. By default, its value is 0, which means to use only one data segment (256 MB). Using numbers that are multiples of 0x10000000 (0x10000000, 0x20000000.....,0x80000000) reserves segment number 2 for the stack, and segment 3, and following segments, for the data and heap.

For example, using in a compilation an argument such as:

```
-bMAXDATA:0x10000000
```

gives us a 256 MB segment for the stack plus a 256 MB segment shared between the heap and static data. This situation is shown in Figure 6 on page 45.

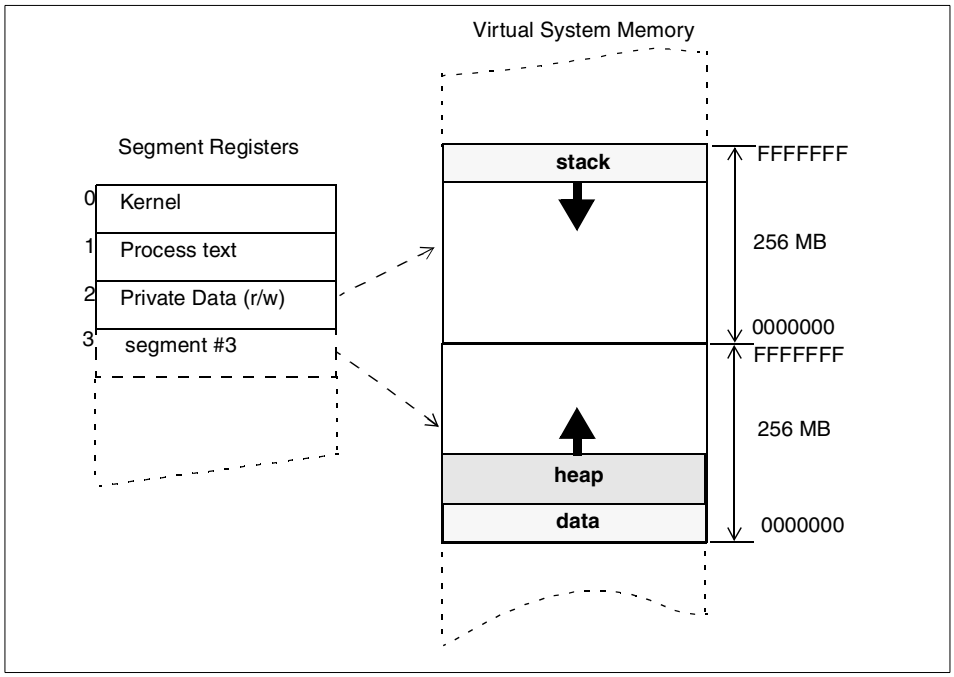


Figure 6. Extended process private data area

2.3.1 Example

The following example program contains very simple code that displays the positioning of different data types within the private data segment:

```
#include <stdio.h>

#define num_vector    3
#define size_vector   1024

/* creates static data */
char static_vector1[ size_vector];
char static_vector2[ size_vector];
char static_vector3[ size_vector];

void function( int data1, int data2, int data3)
{
  printf(" Stack Data\n");
  printf("              data 3 address: %p \n", &data3);
  printf("              data 2 address: %p \n", &data2);
  printf("              data 1 address: %p \n", &data1);
}
```

```
main()
{
int i;
char * heap_vector[ num_vector];

                                /* creates dynamica data */
    for( i = 0; i < num_vector; i++)
       heap_vector[i] = (char *)malloc( size_vector);

                                /* display stack data    */
    function( 1, 2, 3);
                                /* display dynamica data */
    printf(" Heap Data\n");
    for( i = 1; i <= num_vector; i++)
       printf("              data %d address: %p\n", (num_vector - i + 1),
                                        &heap_vector[num_vector - i][0]);

                                /* display static data   */
    printf(" Static Data\n");
    printf("              data 3 address: %p\n",  &static_vector3);
    printf("              data 2 address: %p\n",  &static_vector2);
    printf("              data 1 address: %p\n",  &static_vector1);

}
```

Here, its output is shown for a default compilation, where the second segment contains the heap, stack, and data. When a 32-bit virtual address is displayed in hexadecimal format, the first digit indicates the segment number. The addresses in the default output all start with 0x2, indicating they are in segment number 2:

```
Stack Data
              data 3 address: 2ff22bc0
              data 2 address: 2ff22bbc
              data 1 address: 2ff22bb8
Heap Data
              data 3 address: 200096e8
              data 2 address: 200092d8
              data 1 address: 20008ec8
Static Data
              data 3 address: 20000b18
              data 2 address: 20000f18
              data 1 address: 20001318
```

When the same program is compiled with the -bmaxdata:0x10000000 option, the output is as follows:

```
Stack Data
              data 3 address: 2ff22bc0
              data 2 address: 2ff22bbc
              data 1 address: 2ff22bb8
Heap Data
              data 3 address: 300096e8
              data 2 address: 300092d8
              data 1 address: 30008ec8
Static Data
              data 3 address: 30000b18
              data 2 address: 30000f18
              data 1 address: 30001318
```

From the addresses printed, it can be seen that the stack is still in segment 2; whereas, the data and heap are now in segment 3. Note that, in both cases, the static data are queued in the reverse order as they are declared.

Note

The following shell commands can patch programs to use large data without relinking them. This can be useful when the source code of the program is not available to relink:

```
/usr/bin/echo '\0200\0\0\0'|dd of=executable_file_name bs=4 count=1
seek=19 conv=notrunc
```

The echo string generates the binary value 0x80000000. The `dd` command seeks to the proper offset in the executable file and modifies the o_maxdata field.

Debugging programs with large data is similar to debugging other programs. The `dbx` command can debug these large programs actively or from a core dump. A full core dump should not be performed because programs with large data areas produce large core dumps, which consume large amounts of file system space.

Chapter 3. AIX shared libraries

Facilities for the creation and use of shared libraries are found on many operating systems. The AIX operating system is no exception and provides a large number of useful tools to aid in the creation, development, testing, and debugging of shared libraries and applications that use them.

Developers porting code to the AIX operating system from other platforms may, at first, be troubled by the different implementation methods that are available. AIX, Version 4.3 contains shared library features that are broadly compatible with other UNIX operating systems. Previous versions of AIX did not contain all of these features; so, the method you choose will be based on the exact version of AIX in use.

The AIX operating system provides facilities for the creation and use of dynamically bound shared libraries. Dynamic binding allows external symbols referenced in user code and defined in a shared library to be resolved by the loader at run time.

The shared library code is not present in the executable image on disk. Shared code is loaded into memory once in the system shared library segment and shared by all processes that reference it. The advantages of shared libraries are:

- Less disk space is used because the shared library code is not included in the executable programs.
- Less memory is used because the shared library code is only loaded once.
- The time taken to start an application may be reduced because the shared library code may already be in memory.
- Performance may be improved because fewer page faults will be generated when the shared library code is already in memory. However, there is a performance cost in calls to shared library routines of one to eight instructions.

This chapter introduces the developer to shared libraries and their implementation in AIX.

3.1 Terminology

When discussing shared libraries, it is very important to understand the terminology used since there are many terms with similar names but different meanings.

3.1.1 Static library

A static library is a collection of object files in a single ar format archive. The library can be used during the linking phase of creating an executable. The object files in the library that contain symbols referenced by the main application are extracted from the library and incorporated into the resulting executable file. The library is used only during the linking phase and is not relevant at runtime. The executable file that is created is sufficient to run the program.

3.1.2 Shared library

Shared libraries and shared objects, (normally called Dynamically Loaded Libraries, or DLLs in Windows terminology) are terms used to refer to object code components that are handled in a special way.

Shared libraries are used in two stages when creating an executable. At link time, the link editor (the `ld` command) searches the specified library to resolve all undefined symbols that are referenced in the main application code. If a shared library contains the referenced symbols, the loader section of the XCOFF header of the created executable contains a reference to the shared library. Unlike using the static library, the object files containing the referenced symbols are not incorporated into the executable. Refer to Figure 7 on page 51 for a graphical representation. At runtime, the system loader (the kernel component that starts new processes) reads the header information of the executable and attempts to locate and load any referenced shared libraries. Assuming all the referenced shared libraries are found, the executable can be started. This process is known as dynamic linking.

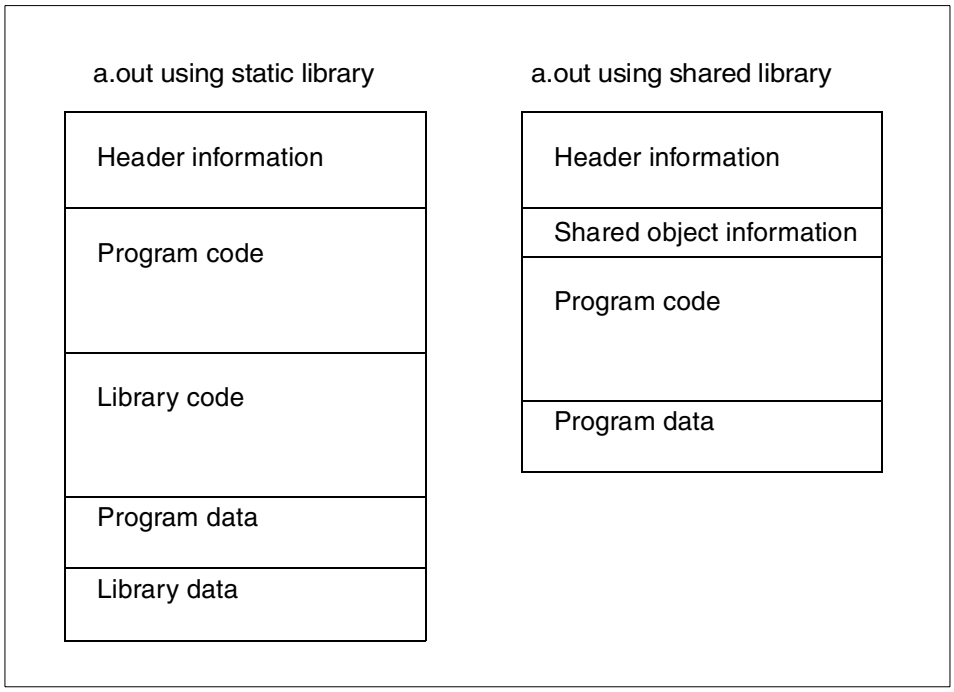


Figure 7. Executables created using static library and shared library

Figure 7 shows the difference between two executables created using the same main application code. One is created using a static version of the library, the other with a shared object version of the same library.

The object code for shared libraries that get loaded into system memory when starting an executable can then be shared by all subsequent executables that use the library. The benefit of this is that only one copy of the object code of a shared library is stored in system memory at any given time, with all the executing programs sharing the same copy. Thus, dynamic linking uses far less memory to run programs. Additionally, the executable files are much smaller, thus potentially also saving disk space.

The AIX operating system supports dynamic linking. Developers moving code to AIX often have problems, however, as the implementation specifics are slightly different from other platforms.

In the UNIX world, the terms shared library and shared object are generally used interchangeably. On the AIX system, there is a distinct difference between the two terms.

shared object A shared object is a single object file that has the SRE (Shared REusable) bit set in the XCOFF header. A shared object normally has a name of the form, *filename*.o. In other words, it is a regular file with a *.o* (lower case O) extension to indicate it is an object file. The SRE bit indicates that the file is handled in a special way by the linker.

shared library A shared library refers to an ar format archive library, where one or more of the members is a shared object. Note that the library can also contain normal, non-shared object files, which are handled in the normal way by the linker. A shared library normally has a name of the form, lib*name*.a. This allows the linker to search for libraries specified with the -lname option on the command line.

AIX Version 4.2.1 introduced support for a new type of shared object, commonly found on other UNIX systems, such as Solaris and HP-UX. Shared files of the new format generally have a name of the form, lib*name*.so. Although the name incorporates the term *lib*, the file is, in fact, a shared object (as indicated by the *.so* filename extension) rather than a shared library, since it is a single object file rather than an ar format archive. The benefit of this type of shared object is that, in common with a true shared library, it can be specified on the compiler or linker command line with the -l*name* option and searched for with the -L *directory* option when the -brtl option is being used.

In addition to the use of shared libraries and shared objects with the compile and link commands, a program may choose to explicitly control their use with the dlopen() family of subroutines.

3.2 Creating a shared library

The method used to create a shared library depends on the type you wish to create.

3.2.1 Traditional AIX shared object

A traditional AIX shared object is a single object file created by a call to the linker (`ld`) command. Normally, the shared object is created from multiple object files that are linked together; however, it is also possible to create a shared object from a single object file. Although the linker is the component that actually does the work, it is normal to create the shared object using the compiler command line since the compiler, in turn, calls the linker once it has performed any processing it is capable of. The benefit of using this method to

create the shared object is that default linker options are automatically used and do not need to be specified on the command line.

Creating a traditional AIX shared object normally involves the use of an export file. An export file is a text file containing a list of symbols. It is used to control which symbols are visible outside the shared object. The symbols not specified in the export file are only visible to other routines within the shared object. The use of export files allows a developer to create a shared object that has a well defined interface. Only the symbols listed in the export file can be referenced by executables and other shared objects that are linked with the object. In addition, creating a shared object may involve the use of an import file. An import file is a text file that lists the names of symbols that the shared object may reference. It allows the object to be created without the source of those symbols being available. This may be required in the situation where two shared objects have dependencies on each others symbols. Export files are normally identified by using a .exp extension to the filename. When the run-time linker (discussed in Section 3.4, “Run-time linking” on page 66) is not used, all symbols must be accounted for when the module is linked. The undefined symbols must be listed in the module’s import list or be deferred. Symbols are deferred if they are listed as being defined by #! in the import list.

If you are creating a shared object and want all symbols to be exported, then you do not need to use an export file. You can use the -bexpall linker option, which will automatically export all global symbols (except imported symbols, unreferenced symbols defined in archive members, and symbols beginning with an underscore). Additional symbols may be exported by listing them in an export list. Any symbol with a leading underscore will not be exported by this option. These symbols must be listed in an exports list to be exported.

If the shared object supplies symbols that are used by another shared object, then you still have to create an exports file, as this is used as an import file when creating the dependent shared object.

3.2.1.1 Single shared object

The scenario described in this section for creating a shared object uses the following source code files:

The file source1.c is as follows:

```
/* source1.c : First shared library source */
void private(void)
{
printf(“private\n”);
}
```

```
int addtot(int a , int b)
{
int c;
c = a+b;
return c;
}
```

The file source2.c is as follows:

```
/* source2.c : Second shared library source */
#include <stdio.h>
int disptot(int a)
{
printf("The total is : %d \n",a);
}
```

The process of creating the shared object is as follows:

1. Create the object files that will be combined together to create the shared object. This is achieved using the -c option of the compiler. For example:

   ```
   cc -c source1.c
   cc -c source2.c
   ```

2. Create an export file that lists the symbol names that should be visible outside the shared object. In this example, the symbols addtotal and displaytotal are the names of the functions that will be called by the main application. The symbol names can also include variable names in addition to function names. The libadd.exp export file is as follows:

   ```
   #!
   addtot
   disptot
   ```

3. Create the shared object with the following command:

   ```
   cc -o shrobj.o source1.o source2.o -bE:libadd.exp -bM:SRE -bnoentry
   ```

   The -bE:libadd.exp option uses the file libadd.exp as an export file that lists the names of the symbols that should be exported. The -bM:SRE flag marks the resultant object file, shrobj.o, as a shared reusable object. The -bnoentry flag indicates that there is no entry point (main function) in the object file.

The `dump` command can be used to list the symbols that are exported (and imported) by the shared object. For example:

```
# dump -Tv shrobj.o

shrobj.o:
```

```
                        ***Loader Section***

                        ***Loader Symbol Table Information***
[Index]      Value      Scn     IMEX Sclass   Type           IMPid Name

[0]      0x00000000    undef      IMP     DS EXTref   libc.a(shr.o) printf
[1]      0x2000020c    .data      EXP     DS SECdef         [noIMid] addtot
[2]      0x20000218    .data      EXP     DS SECdef         [noIMid] disptot
```

The important fields to examine are the IMEX, IMPid, and Name entries. A value of EXP in the IMEX field indicates that this symbol is being exported from the object. In this case, the Name field gives the name of the symbol being exported, and the IMPid field is not used.

A value of IMP in the IMEX field means that the symbol listed in the Name field is being imported into the object. In this case, the IMPid indicates the target shared object that the symbol will be imported from. In the case of a shared object that is contained in an ar format library, both the library name and object name will be displayed. In the example shown above, the symbol printf is being imported from the shared object shr.o, which is contained in the libc.a archive library.

3.2.1.2 Interdependent shared objects

The process for creating interdependent shared objects is similar to the process of creating a single shared object but requires the use of an import file. Suppose there are two shared objects, shr1.o and shr2.o, and each references symbols in the other. When creating the first shared object (shr1.o), the second shared object may not exist. This means that when the command to create the first shared object is executed, there will be unresolved symbols since, at this point, the second shared object does not exist. This problem is overcome with the use of an import file. An import file is similar to the export file used when creating a shared object. In fact, in most cases, it is possible to use the same file for both purposes.

Consider the following files for use in this example scenario:

The file source1.c is as follows:

```
/* source1.c : First shared library source */
int function1(int a)
{
int c;
c = a + function2(a);
return c;
}
```

```
int function3(int a)
{
int c;
c = a / 2;
return c;
}
```

The file source2.c is as follows:

```
/* source2.c : Second shared library source */
int function2(int a)
{
int c;
c = function3(a + 5);
return c;
}
```

In this example, each source file needs to be made into a separate, shared object. Note that the resulting shared objects are interdependent, since:

- Subroutine function1 in source1.c calls function2 in source2.c.
- Subroutine function2 in source2.c calls function3 in source1.c.

As with the simple example, each shared object requires an export file to define which symbols will be exported. With a slight modification, the export file for each shared object can also be used as the import file for other shared objects that use the exported symbols. The slight change does not effect the file when used as an export file. The modification is to add the name of the library and shared object that contains the symbols. In the example, the export file (libone.exp) for the first shared object is:

```
#!libone.a(shr1.o)
function1
function3
```

The export file (libtwo.exp) for the second shared object is:

```
#!libtwo.a(shr2.o)
function2
```

When used as an export file, the line starting with the #! symbol sequence is ignored. When used as an import file, the information following the #! sequence details the location of the symbols contained in the file. The format of the entry is *libraryname*(*membername*). The libraryname component can be just the name of the library (as it is in the example), or it may include a relative or absolute pathname component, for example:

```
#!/data/lib/libone.a(shr1.o)
```

Any pathname component listed is used when attempting to load the shared object at runtime to resolve the symbols.

The commands used to create the shared objects and create the libraries are as follows:

```
cc -c source1.c
cc -o shr1.o source1.o -bE:libone.exp -bI:libtwo.exp -bM:SRE -bnoentry
ar rv libone.a shr1.o
cc -c source2.c
cc -o shr2.o source2.o -bE:libtwo.exp -bI:libone.exp -bM:SRE -bnoentry
ar rv libtwo.a shr2.o
```

Note the use of the file libone.exp as an export file when creating the first shared library and as an import file when creating the second. If the file is not used when creating the second shared library, the creation of the shared object will fail with an error message complaining of unresolved symbols:

```
cc -o shr2.o source2.o -bE:libtwo.exp  -bM:SRE -bnoentry
ld: 0711-317 ERROR: Undefined symbol: .function3
ld: 0711-345 Use the -bloadmap or -bnoquiet option to obtain more
information.
```

A single import file can be used to list symbols that are imported from different modules. The import file is just a concatenation of the individual export files for each of the shared objects. Using the example import files shown above, suppose that a new shared object, libthree.a, was to be created, and it imports symbols from both libone.a and libtwo.a. The import file used to create the new shared object might be as follows:

```
#!libone.a(shr1.o)
function1
function3
* a comment line starts with the asterix symbol
* blank lines are ignored

#!libtwo.a(shr2.o)
function2
```

As the example illustrates, although it is possible to create interdependent shared objects, from a design and implementation point of view, it is much simpler to create shared objects that are as self-contained as possible.

3.2.2 New style shared object

Creating a new style shared object (lib*name*.so) does not require the use of export files; however, by default, all symbols are visible to executables that are linked with the object.

3.2.2.1 Single shared object

Using the same source code as used in Section 3.2.1.1, “Single shared object” on page 53, the following command is used to create a new style shared object:

```
cc -G -o libsimple.so source1.o source2.o
```

Note that the -G option implicitly enables a number of other default linker options, including one that exports all symbols. This makes things simple when creating the shared object since you do not need to maintain a file listing the symbols you want to be exported. The effect of this can be seen in the output of the `dump` command when used on the resulting shared object:

```
# dump -Tv libsimple.so

libsimple.so:

                        ***Loader Section***

                        ***Loader Symbol Table Information***
[Index]      Value      Scn     IMEX Sclass   Type           IMPid Name

[0]     0x00000000    undef      IMP     DS EXTref   libc.a(shr.o) printf
[1]     0x20000204    .data      EXP     DS SECdef         [noIMid] addtot
[2]     0x20000210    .data      EXP     DS SECdef         [noIMid] privatefn
[3]     0x2000021c    .data      EXP     DS SECdef         [noIMid] disptot
```

Although the manual pages for the compilers state that the -G option is passed directly to the linker, the compiler itself does, in fact, perform additional processing. This can be detected since replacing `cc` with `ld` in the example shown above results in an error:

```
ld -G -o libsimple.so source1.o source2.o
ld: 0711-327 WARNING: Entry point not found: __start
ld: 0711-244 ERROR: No csects or exported symbols have been saved.
```

Even resolving the warning message about the entry point by using the -bnoentry linker option does not solve the problem. There is still a warning that no exported symbols have been saved. Essentially, this means the shared object has not exported any symbols.

The reason the command works when the compiler is invoked with the -G option can be seen when we additionally use the -v option to get more information about what the compiler is actually doing:

```
cc -v -G -o libsimple.so source1.o source2.o
exec: /usr/vac/bin/CreateExportList(/usr/vac/bin/CreateExportList,
/tmp/xlcSEMY4Qie,-f,/tmp/xlcSFMY4Qid,NULL)
exec: /bin/ld(ld,-bM:SRE,-bnoentry,-bpT:0x10000000,-bpD:0x20000000,
-olibsimple.so,source1.o,source2.o,-lc,-bE:/tmp/xlcSEMY4Qie,NULL)
unlink: /tmp/xlcW0MY4Qia
unlink: /tmp/xlcW1MY4Qib
unlink: /tmp/xlcW2MY4Qic
unlink: /tmp/xlcSEMY4Qie
unlink: /tmp/xlcSFMY4Qid
```

The important thing to notice is that the compiler is using a shell script called CreateExportList to create an export list file on the fly for the specified input files.

3.2.2.2 Creating an export list

You can use the /usr/vac/bin/CreateExportList shell script supplied with the C for AIX, Version 5 compiler to automatically generate the symbols that should be included in an export list. It can save a considerable amount of time if you want to use the traditional AIX method for creating shared objects as described in Section 3.2.1, “Traditional AIX shared object” on page 52, or if you want to use an export list in conjunction with the -G option to create a new style shared object that does not export all symbols.

The simplest way to use the command is as follows:

1. Compile all of the source files that will be included in the shared object.
2. Create a single file that lists the names of all of the object files that will be included in the shared object. For example, create a file called objectlist that contains the following lines:

   ```
   source1.o
   source2.o
   ```

3. Invoke the `CreateExportList` command as follows:

   ```
   /usr/vac/bin/CreateExportList exportfile -f objectlist
   ```

   where `exportfile` is the name of the export file you want to create, and `objectlist` is the file that contains the list of object file names.

4. Edit the resulting export file to include the #!pathname(member) line at the start.

5. Edit the resulting export file to remove the symbol names you wish to keep private within the shared object.

3.2.2.3 Interdependent shared objects

The creation of interdependent shared objects using the libname.so style requires the use of import files so that the linker can resolve the externally referenced symbols.

If using an export file for a new style shared object as an import file when creating another shared object, the location specified does not need the (member) entry since the file itself is the shared object. Using the example described in Section 3.2.2.1, “Single shared object” on page 58, the export file produced would have the following line inserted as the first line in the file:

```
#!libsimple.so
```

3.2.3 Importing symbols from the main program

When creating either traditional or new style shared objects, it is possible for the object to resolve a symbol that is provided in the main program rather than a shared object. There are two steps required to ensure that this works correctly.

The first step is to use an import file when creating the shared object that lists the symbols to be imported from the main routine. The symbols should be listed under the module name as follows:

```
#!.
```

The special module name of “.” (dot) indicates that the symbols will be imported from the main program. The status of the symbols in the shared object can be checked using the `dump -Tv` command as described in Section 3.5.3.2, “The dump -Tv command” on page 75.

Link the application using the shared objects as normal. The linker will automatically detect that the shared objects import symbols from the main routine and will automatically export them if they exist. If a shared object tries to import a symbol that does not exist in the main routine, then the link stage will fail.

3.2.4 Initialization and termination routines

Optional shared object initialization and termination routines can be specified when creating the shared object. You can use one or the other, or both. The routines may be useful for initializing dynamic data structures or reading configuration information. The initialization routines are called by the program

startup code and are performed before the application main routine is started. Termination routines are called when the program makes a graceful exit. They will not be called if the program exits due to receipt of a signal.

The -binitfini linker option is used to specify the names of the routines along with a priority number. The priority is used to indicate the order that the routines should be called in when multiple shared objects with initialization or termination routines are used.

3.3 Using a shared library

Once you have created the required shared libraries, you can then proceed to use them when linking applications. There are a number of linker options that effect the way in which the shared libraries are used.

The most important point to remember about using shared libraries is that the way the application is linked will determine how the shared libraries will be searched for at runtime.

3.3.1 On the compile line

When using shared libraries to create an executable, there are a number of methods that can be used to specify the library on the command line. The method used will depend on the type of shared object being used.

As far as the linker is concerned, there are three types of shared objects that it can handle:

- An archive library that contains object files with the SRE bit set.
- A new style shared object of the form lib*name*.so.
- An individual object file with the SRE bit set, for example, shr1.o.

In all cases, the shared object can be specified directly on the command line using either an absolute or relative pathname. If the shared object is in the same directory as the current working directory, then no path component needs to be specified since the current directory is searched by default.

If the shared object is a single object file, then the absolute or relative pathname is the only way to include it on the command line.

If the shared object is part of an archive library, then the -l and -L linker options can be used to search for the library. If the shared object is a new style shared object, then the -brtl linker option must be used. This enables the runtime linker, described in Section 3.4, “Run-time linking” on page 66, and

also allows these shared objects to be specified on the command line using the -l and -L options. If you want to use the new style shared object naming conventions, but do not want to use run-time linking, then specify the -brtl and -bnortllib options when linking the main application. This will mean that you must build the new style shared objects using export and import files, if required. You should use the compiler with the -G option to create the shared objects, not the ld -G method described in Section 3.4, “Run-time linking” on page 66.

The -l option is used to specify the name of the library without the .a or .so extension and without the lib prefix. For example, the shared objects libone.a and libtwo.so, would be specified on the command line as -lone -ltwo.

The -L option is used to specify a directory that should be searched for the libraries specified with the -l option. The /usr/lib and /lib directories are automatically added to the end of the list of directories to be searched. The list of directories specified with the -L option (along with the default /usr/lib and /lib entries) is included in the header section of the resulting executable. This path is used to search for the directories at runtime. Refer to Section 3.5.3.1, “The dump -H command” on page 72 for details on how the use of pathnames, and the -L option can have an impact on how the system loader searches for the shared objects at runtime.

If your application development directory structure does not match the directory structure used when your application is installed in a production environment, then, potentially, you need to adjust the arguments used with the linker to ensure that the resulting executables have the desired library search path.

For example, consider an application that has a development source code tree as shown in Figure 8 on page 63.

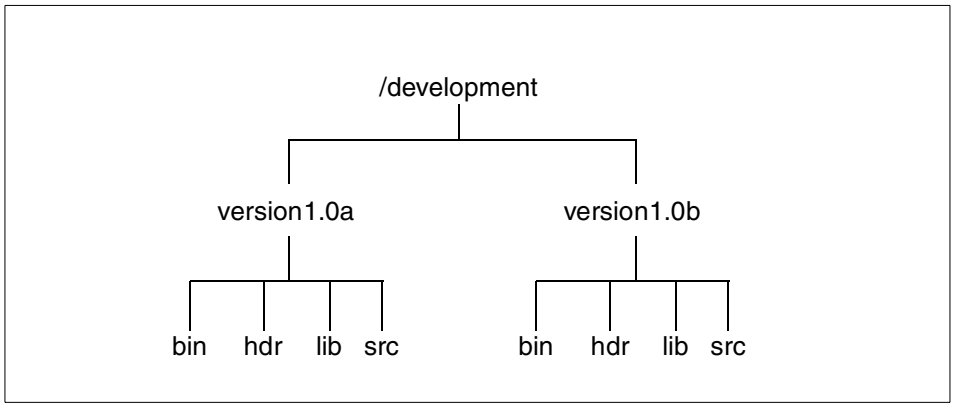


Figure 8. Sample development directory structure

Consider the application file, main.c, being compiled and linked in the directory, /development/version1.0b/src, and using shared libraries stored in the directory, /development/version1.0b/lib. There are a number of options that can be used to specify the libraries, depending on how the resulting executable will be deployed.

When the application is installed in a production environment, for example, after being installed on a customer machine, the directory structure may be different. The method to use when compiling the executables will depend on the degree of freedom the customer is permitted when installing the application. For example, some products specify that the executables and libraries must be installed in a specific directory, such as /opt/productname. Some products allow the binaries and libraries to be installed in any directory structure.

If the libraries for the product will be installed in a specific directory, then you can either:

- Create the shared libraries and then copy them to the same directory structure to be used when the product is installed in a production environment. In this case, you use the -L option to find the shared libraries. For example:

  ```
  cc -o ../bin/app1 main.c -llibone -L/product/lib
  ```

- Create the shared libraries, but leave them in the development directory structure. When compiling the applications, use absolute pathnames to specify the shared libraries along with the -bnoipath linker option to prevent the pathname being included in the header section of the final

executable. At the same time, use the -L option to specify the directory where the libraries will exist on a production system. For example:

```
cc -o ../bin/app1 main.c -bnoipath ../lib/libone.a -L/product/lib
```

If your product allows the executables and libraries to be installed in any directory structure, then you need to use the LIBPATH environment variable to search for shared objects. Some other UNIX platforms use the LD_LIBRARY_PATH variable for a similar purpose.

The order of libraries and objects specified on the command line is not important unless run-time linking is being used. See Section 3.4, “Run-time linking” on page 66 for more information.

3.3.2 Searching at runtime

The LIBPATH environment variable is only needed when shared libraries exist in a different directory to that specified in the header section of the executable. The variable is a colon separated list of directory names. If it is set, the directories specified in the LIBPATH environment variable are searched for the required shared objects before the list of directories specified in the header section of the executable. The exception to this case is when a user other than root is attempting to run a setuid or setgid executable. In this case, only the directories listed in the header section of the executable are searched; the LIBPATH variable is ignored, even if set.

If a relative or absolute pathname is used to specify a shared object when the application is compiled, and the -bnoipath option is not specified, then the system loader will only look for the shared object using the exact pathname specified at link time for that object. Even if a shared object with the same name exists in a directory searched as part of the LIBPATH or INDEX 0 path included in the header section, it will be ignored.

If a shared object can not be found by the system loader when trying to start an executable, an error message similar to the following will be seen:

```
exec(): 0509-036 Cannot load program ex1 because of the following errors:
      0509-022 Cannot load library libone.so.
      0509-026 System error: A file or directory in the path name does not
exist.
```

The missing objects will be listed with 0509-022 error messages. Use the `find` command to search the system for the missing shared objects. If the object is found, try setting the LIBPATH environment variable to include the directory that contains the shared object and restart the application. Also, ensure that

the object or library has read permission for the user trying to start the application.

A similar error message is produced when the system loader finds the specified shared objects, but not all of the required symbols can be resolved. This can happen when an incompatible version of a shared object is used with an executable. The error message is similar to the following:

```
exec(): 0509-036 Cannot load program ./ex1 because of the following errors:
      0509-023 Symbol func1 in ex1 is not defined.
      0509-026 System error: Cannot run a file that does not have a valid
format.
```

The unresolved symbols are listed in the 0509-023 message lines. Note the name of the symbol, and use the `dump -Tv` command to determine which shared object the executable expects to resolve the symbol from. For example:

```
# dump -Tv ex1 | grep function1
[4]     0x00000000    undef      IMP     DS EXTref libone.a(shr1.o) func1
```

This indicates that the executable is expecting to resolve the symbol func1 from the shared object shr1.o which is an archive member of libone.a. This information can help you start the problem determination process.

3.3.3 Shared or non-shared

AIX, Version 4.3 supports the use of the -bdynamic and -bstatic linker options to determine how a shared object should be treated by the linker.

These options are toggles and can be used repeatedly in the same link line. When -bdynamic is in effect, shared objects are used in the usual way. If you use the -bstatic option, remember to specify -bdynamic as the last option on the link line to ensure that the system libraries are treated as shared objects by the linker. If this is not done, and the system libraries are treated as normal archive libraries, the executable produced will be larger than normal. In addition, it will have the disadvantage that it may not work on future versions of AIX since it is hardcoded with a specific version of system libraries.

When the -bstatic option is in effect, shared objects are treated as regular files. Additionally, when -brtl is specified, and -bdynamic is in effect, the -l flag will search for files ending in ‘.so’ as well as those ending in ‘.a’. Do refer to the following examples:

```
cc -o main.o -bstatic -lx -Lnewpath -bdynamic
```

In this example, libx.a is treated as a regular archive file, even if it contains shared objects. The -bdynamic ensures that the system libraries, such as libc.a, are processed as shared objects.

```
cc -o main main.o -brtl -lx -Lpath1 -Lpath2
```

Search for the object specified by -lx in the following order:

1. path1/libx.so
2. path1/libx.a
3. path2/libx.so
4. path2/libx.a

3.3.4 Lazy loading

AIX, Version 4.3 supports the use of the -blazy option to implement lazy loading. Lazy loading is a mechanism for deferring the loading of modules until one of its functions is required to be executed. By default, the system loader automatically loads all of the module's dependants at the same time. By linking a module with the -blazy option, the module is loaded only when a function within it is called for the first time. Note that lazy loading works only if the run-time linker is not enabled. Also, only the modules referenced for their function can be lazy loaded.

3.4 Run-time linking

As shown in the examples above, generally, references to the symbols in the shared objects are bound at link time. That is, the output module associates an imported symbol with its definition in a specific object. The source of the definition can be seen by using the `dump -Tv` command on the executable or shared object. Refer to Section 3.5.3, "The dump command" on page 72 for more details.

At load time, the definition in the specified shared object is used even if other shared objects export the same symbol.

Programs can be modified to use the run-time linker, therefore, allowing some symbols to be rebound at load time. To create a program that uses the run-time linker, link the program with the -brtl option. The way that shared modules are linked affects the rebinding of symbols.

To build shared objects enabled for run-time linking, use the -G flag and build the shared object with the `ld` command rather than the compiler `cc`, `xlc`, or `xlC`

commands. The -G linker option enables the combination of options described in Table 9.

Table 9. The -G option

Option	Description
-berok	Enables creation of the object file, even if there are unresolved references
-brtl	Enables runtime linking. All shared objects listed on the command line (those that are not part of an archive member) are listed in the output file. The system loader loads all such shared modules when the program runs, and the symbols exported by these shared objects may be used by the runtime linker.
-bsymbolic	Assigns this attribute to most symbols exported without an explicit attribute.
-bnortllib	Removes a reference to the runtime linker libraries. This means that the module built with -G option (which contains the -bnortllib option) will be enabled for runtime linking, but the reference to the runtime linker libraries will be removed. Note that the runtime libraries should be referenced to link the main executable only.
-bnoautoexp	Prevent automatic exportation of any symbol.
-bM:SRE	Build this module to be shared and reusable.

The function of the -G option to the `compiler` command is very similar in function to the -G option to the linker (`ld`) command, but there is a very subtle, yet important, difference when it comes to creating shared objects for use with runtime linking.

The important difference is the way the two options impact the handling of unresolved symbols. The following source code files will be used to demonstrate the difference.

File source1.c is used to make libone.so. The source code is as follows:

```
/* source1.c - demo of difference between cc -G and ld -G */
#include <stdio.h>
void function1(int a)
{
   printf("In function1\n");
   function2(a);
}
```

File source2.c is used to make libtwo.so. The source code is as follows:

```
/* source2.c - demo of difference between cc -G and ld -G */
```

```
#include <stdio.h>
void function2(int a)
{
   printf("In function2\n");
}
```

If the compiler command is used to create libone.so, initially it fails with an error message complaining about the unresolved symbol function2:

```
cc -G -o libone.so source1.c
ld: 0711-317 ERROR: Undefined symbol: .function2
ld: 0711-345 Use the -bloadmap or -bnoquiet option to obtain more
information.
```

We can solve this immediate problem in one of two ways. We can supply an import file that resolves the symbol function2 to the shared object libtwo.so. However, if we do this, the shared object, libone.so, will be created with a reference to libtwo.so in the header section. This means we have resolved the symbol function2 at link time, which is not what we want. Alternatively, we can add the -berok option to the command line, which allows errors in the output file. If we do this, then the symbol function2 is unresolved at link time, which is what we want. We can then create libtwo.so, and then link both libraries with the following main.c program:

```
/* main.c - demonstrate how cc -G differs from ld -G */
int main(int argc, char ** argv)
{
   function1(45);
}
```

using the following command:

```
cc -o example main.c -brtl -L`pwd` libone.so libtwo.so
```

Note the use of the -brtl option, which is required to enable the run-time linker. If we try and run the example program, an error message is produced:

```
# ./example
in function 1
Segmentation fault(coredump)
```

It can be seen from the output that the program has managed to start and get as far as the printf statement in function1. It then experiences a fatal error when trying to call function2. If we look at the header information for libone.so with the `dump -Tv` command, we can check the status of the symbols:

```
# dump -Tv libone.so

libone.so:
```

```
                        ***Loader Section***

                        ***Loader Symbol Table Information***
[Index]      Value      Scn     IMEX Sclass   Type           IMPid Name

[0]      0x00000000    undef      IMP     DS EXTref   libc.a(shr.o) printf
[1]      0x200001e8    .data      EXP     DS SECdef        [noIMid] function1
[2]      0x00000000    undef      IMP     DS EXTref        [noIMid] function2
```

It can be seen that the symbol function2 is marked as undef, which is what we expect. However, the problem is that the IMPid is marked as [noIMid], which means that the shared object does not know where to resolve the symbol function2. If we use the `ld` command to create the shared object, instead of the compiler, then the result is slightly different. Create the shared object with the following commands:

```
cc -c source1.c
ld -G -o libone.so source1.o -bnoentry -bexapall -lc
```

The -bnoentry and -bexpall options are described previously. The -lc option is required to link the C library to resolve the printf function. If we look at the symbol information in the header section with the `dump -Tv` command:

```
# dump -Tv libone.so

libone.so:

                        ***Loader Section***

                        ***Loader Symbol Table Information***
[Index]      Value      Scn     IMEX Sclass   Type           IMPid Name

[0]      0x00000000    undef      IMP     DS EXTref   libc.a(shr.o) printf
[1]      0x00000008    .data      EXP     DS SECdef        [noIMid] function1
[2]      0x00000000    undef      IMP     DS EXTref             .. function2
```

the difference is the IMPid for the symbol function2. The shared object now thinks it will resolve the symbol from the special module called ".." (dot dot). This indicates that the symbol will be resolved by the run-time linker. If we create libtwo.so using the same method, then the example program works correctly.

The run-time linker is called by the program startup code before entering the application's main routine.

When using run-time linking, the order of specifying libraries and objects on the command line is important. This is because the list of libraries and objects will be searched in sequence to resolve symbols that are imported from the special ".." module. In addition, all of the shared objects specified on the command line will be included in the header section of the resulting executable. Using the example described above, the main program only calls the routine function1, which is in libone.so. Using the traditional style AIX link time symbol resolution, this would mean that the resulting executable would only reference libone.so in the header section. If this were the case, when the run-time linker is called, the shared object, libtwo.so, would not be present, and so the symbol resolution of function2, which is called from function1, would fail.

Another advantage of using run-time linking is that developers do not need to maintain a list of module interdependencies and import/export lists. By using the -bexpall option, all shared objects can export all symbols, and the run-time linker can be used to resolve the inter-module dependencies.

3.4.1 Rebinding system defined symbols

The shared libraries shipped with the AIX operating system are not enabled for run-time linking, but they can be enabled by using the `rtl_enable` command. For example, if a program defines its own version of the malloc routine, and wants to run in such a way that the routines in the libc.a shared objects also use the user defined version of malloc, then a new instance of libc.a must be first created. This can be done as follows:

```
rtl_enable -o /usr/local/lib/libc.a /lib/libc.a
```

Then, the program must be relinked:

```
cc .... mymalloc.o -L /usr/local/lib -brtl -bE:myexports
```

In this example, mymalloc.o defines malloc and the export file myexports causes the symbol malloc to be exported from the main program. Calls to malloc from within libc.a will now go to the malloc routine defined in mymalloc.o.

3.5 Developing shared libraries

The way a shared library is used in a development environment is somewhat different to that in a production environment. In a development environment, the library may be constantly changed and altered so that new versions can be tested. On large systems, multiple users may be working with their own

version of the shared library. There are a number of things to be aware of to make the development environment for shared libraries easier to use.

If the system has multiple versions of a shared library, then you need to be careful that your program uses the version of the library that you want. This can be achieved with the use of the -L option on the command line and the use of the LIBPATH environment variable.

When an application is started, the system loader reads the loader section of the header of the executable file. It reads the dependency information for any shared objects the executable requires and attempts to load the code for those shared objects into the system shared library segment if they are not already loaded. Shared objects that are loaded into the system shared library segment have an attribute called the use count. Each time an application program that uses the shared object is started, the use count is incremented. When an application terminates, the use count for any shared objects it was using is decreased by one. When the use count for a shared object in the system shared library segment reaches zero, the shared object is not unloaded, but instead, remains in memory. This is done to reduce the overhead of starting any more applications that use the shared object since they will not have to load the object into the system shared segment.

3.5.1 The genkld command

The `genkld` command is used to list the shared objects that are loaded in the system shared library segment. The output of the command can contain multiple duplicate entries and be quite lengthy; so, it is best to filter the output using the `sort` command or by performing a grep for the shared object you are investigating. For example:

```
# genkld | sort -u

      d00005c0           19f26f /usr/lib/libc.a/shr.o
      d01a00f8              87a /usr/lib/libcrypt.a/shr.o
      d01a7100             78b4 /usr/lib/libi18n.a/shr.o
      d01af100            137fe /usr/lib/libiconv.a/shr4.o
      d01c3100            124f1 /usr/lib/libodm.a/shr.o
      d01d6100             bc4c /usr/lib/libcfg.a/shr.o
      d01e2880            19583 /usr/lib/libsm.a/shr.o
      d01fc100            262fc /usr/lib/liblvm.a/shr.o
      d02230f8             1624 /usr/lib/libpthreads_compat.a/shr.o
```

The command can only be executed by the root user or a user in the system group. The three columns show the virtual address of the object within the

system segment, the size of the object, and the name of the file that was loaded.

3.5.2 The slibclean command

The `slibclean` command can be used by the root user to unload all shared objects with a use count value of zero from the system shared library segment. This command is useful in an environment when shared libraries are under development. You can run the `slibclean` command followed by the `genkld` command to ensure that the shared objects under development are not loaded in the system shared library segment. This means that any application started after this will automatically use the latest version of the shared objects since the system loader will search for and load them. It also prevents multiple versions of the same objects existing in the system segment.

During the development of shared objects, you may sometimes see an error message similar to the following when creating a new version of an existing shared object:

```
# make libone.so
        cc -O -c source1.c
        cc -berok -G -o libone.so source1.o
ld: 0711-851 SEVERE ERROR: Output file: libone.so
        The file is in use and cannot be overwritten.
make: 1254-004 The error code from the last command is 12.
```

The error message means that the shared object in question has been loaded into the system shared library segment. The file is marked as in use, even if the use count is zero. Running the `slibclean` command will unload all of the unused shared objects from the system. An alternative (and simpler) method of avoiding this problem is to use the `rm -f` command to remove the shared object before creating it.

3.5.3 The dump command

The `dump` command is used to examine the header information of executable files and shared objects. The main options that are useful when working with shared libraries are the -H option and the -Tv options.

3.5.3.1 The dump -H command

The `dump -H` command is used to determine which shared objects an executable or shared object depends on for symbol resolution at runtime. The interesting information is in the last section of output and has the title, ***Import File Strings***. Sample output is as follows:

```
# dump -H example
```

example:

```
                    ***Loader Section***
                  Loader Header Information
VERSION#         #SYMtableENT     #RELOCent        LENidSTR
0x00000001       0x00000006       0x0000000e       0x00000047

#IMPfilID        OFFidSTR         LENstrTBL        OFFstrTBL
0x00000003       0x00000158       0x00000019       0x0000019f


                    ***Import File Strings***
INDEX  PATH                              BASE               MEMBER
0      /tmp/addlib/old/complex:/usr/lib:/lib
1                                        libc.a             shr.o
2                                        libone.a           shr1.o
```

The number of INDEX entries will depend on how many shared objects the target depends on for symbol resolution. The INDEX 0 entry is a colon separated list of directories. If the LIBPATH environment variable is not set when the executable is started, the directories listed in the INDEX 0 entry are searched for by the shared objects mentioned in subsequent entries. The directories in the entry are those used with the -L option when the object was linked. The /usr/lib and /lib entries are always present. If you want these directories to be searched first, you need to add them explicitly to the linker command line and ensure that they appear before any other -L options. Using the example shown above, altering the -L options on the `link` command line to be -L/usr/lib -L/lib -L/tmp/addlib/old/complex would result in an INDEX 0 entry of:

```
0      /usr/lib:/lib:/tmp/addlib/old/complex:/usr/lib:/lib
```

The format of the other entries is as follows:

Index The index number of the entry in the Import File Strings section.

Path Optional pathname component of the shared object. A pathname will be present if a pathname was used when the shared object was specified on the `link` command line. The -bnoipath linker option can be used to prevent the pathname used on the command line from appearing in this portion of the entry. The -bipath option is the default. The option effects all shared objects listed on the command line.

Base The name of the archive library containing the shared object, or the name of the shared object itself if it is a new style shared object.

Member The name of the shared object if it is contained in an archive library.

Some examples of different `link` commands are appended below, along with the Import File Strings section of the output of the `dump -H` command on the resulting executables. This demonstrates the relationship between the way the shared objects are specified on the command line and the entries in the Import File Strings section of the executable header.

This sample shows the use of the absolute pathname on the link line. The following command:

```
cc -o example main.c /tmp/addlib/old/complex/libone.a
```

results in an Import File Strings section of:

```
                        ***Import File Strings***
INDEX  PATH                            BASE               MEMBER
0      /usr/lib:/lib
1                                      libc.a             shr.o
2      /tmp/addlib/old/complex         libone.a           shr1.o
```

This sample shows how to suppress the absolute pathname used on the link line. The following command:

```
cc -o example main.c -bnoipath /tmp/addlib/old/complex/libone.a
```

results in an Import File Strings section of:

```
                        ***Import File Strings***
INDEX  PATH                            BASE               MEMBER
0      /usr/lib:/lib
1                                      libc.a             shr.o
2                                      libone.a           shr1.o
```

When using a new style shared object, there is no member entry in the output, only a base entry. The following command:

```
cc -brtl -o example main.c -L/tmp/addlib/new/complex -lone
```

results in an Import File Strings section of:

```
                        ***Import File Strings***
INDEX  PATH                            BASE               MEMBER
0      /tmp/addlib/new/complex:/usr/lib:/lib
1                                      libone.so
```

```
2                                   libc.a              shr.o
3                                   librtl.a            shr.o
```

3.5.3.2 The dump -Tv command

The `dump -Tv` command is used to examine the symbol information of a shared object or executable. It lists information on the symbols the object is exporting. It also lists the symbols the object or executable will try and import at load time and, if known, the name of the shared object that contains those symbols. The main columns to examine in the output are headed IMEX, IMPid, and Name.

The IMEX column indicates if the symbol is being imported (IMP) or exported (EXP). The IMPid field contains information on the shared object that the symbol will be imported from. The Name field lists the name of the symbol. For example:

```
# dump -Tv libone.so

libone.so:

                        ***Loader Section***

                        ***Loader Symbol Table Information***
[Index]      Value      Scn     IMEX Sclass   Type           IMPid Name

[0]      0x00000000    undef      IMP     DS EXTref   libc.a(shr.o) printf
[1]      0x00000000    undef      IMP     DS EXTref      libtwo.so function2
[2]      0x20000264    .data      EXP     DS SECdef        [noIMid] function3
[3]      0x20000270    .data      EXP     DS SECdef        [noIMid] function1
```

The output shown above for the libone.so new style shared object indicates that the symbols function1 and function3 are being exported from this object. The object also has two imported symbols on which it depends. The symbol, printf, is being imported from the shared object, shr.o, which is a member of the libc.a archive library. It also imports the symbol function2 from the new style shared object, libtwo.so.

3.5.4 Using a private shared object

When used under normal circumstances, a shared object is loaded into the system global shared object segment. Subsequent executables that use the shared object benefit from the fact that it is already loaded.

In a development environment, particularly on a system with multiple developers, it may be preferable to use a private copy of a shared object. This may be useful when developing and testing new functionality in a shared

object that is specific to a particular version of the application that a single developer is working on.

If the shared object or container has the access permissions modified as detailed below, then when the system loader starts an application that uses this shared object, the shared object text will be loaded into the process private segment rather than the system shared object segment. The shared object data will also be loaded into the process private segment instead of its normal location of the process shared object data segment. This means every application will have its own private copy of the shared object text and data. Applications normally have their own copy of the shared object data and share the text with other applications.

To use a private version of the shared object text and data, modify the access permissions as follows:

- If the shared object is contained in an archive library, remove read-other permission from the archive library.
- If the shared object is a new style shared object, for example libname.so, or a standalone shared object, for example, shrobj.o, then remove read-other permission from the shared object.

The effect of this change can be demonstrated using the following sample code.

The file source1.c is used to make a simple shared object. It contains the following code:

```
struct funcdesc {
int     codeaddr;
int     TOCentry;
int     env;
} * shlibfdesc;

void function1(int a)
{
      shlibfdesc = (struct funcdesc *) function1;
      printf("address of function1 is 0x%p\n",shlibfdesc->codeaddr);
      printf("address of shlibfdesc is 0x%p\n",&shlibfdesc);
}
```

The shared object is linked with a small main application, which contains the following code:

```
struct funcdesc {
int     codeaddr;
int     TOCentry;
```

```
int     env;
} * mainfdesc;
int main(int argc, char ** argv)

{
        mainfdesc = (struct funcdesc *) main;
        printf("address of main is 0x%p\n",mainfdesc->codeaddr);
        printf("address of mainfdesc struct is 0x%p\n",&mainfdesc);
        function1();
}
```

A function pointer in the C language is implemented as a pointer to a structure that contains three entries. The first entry is a pointer to the address of the code, and the second is a pointer to the table of contents entry for the module containing the function. The third entry is a pointer to environment information and is used by certain other languages, such as Pascal.

The shared object is created with the following commands:

```
cc -c source1.c
ld -G -o libone.so source1.o -bexpall -bnoentry -lc
chmod o-r libone.so
```

Note that read-other permission is removed from the shared object. The main routine is then created with the following command:

```
cc -brtl -o example main.c -lone -L.
```

The output from running the program is as follows:

```
./example
address of main is 0x100002f0
address of mainfdesc struct is 0x200027d4
address of function1 is 0x20000150
address of shlibfdesc is 0x2000127c
```

Note that the address of the code for the main routine is in segment 1 as expected, and the data structure mainfdesc is in segment 2. Since the shared object had read-other permission removed, it was loaded into segment 2 by the system loader. This can be seen with the address of function1 and shlibfdesc starting with 0x2.

If read-other permission is restored to the shared object, and the program is invoked again, the result is as follows:

```
chmod o+r libone.so
./example
address of main is 0x100002f0
```

```
address of mainfdesc struct is 0x200007d4
address of function1 is 0xd040f150
address of shlibfdesc is 0xf001f27c
```

The address of the main routine and the mainfdesc structure have not changed. The address of function1 now starts with 0xd. This indicates the code is in segment 13, the system shared object segment. The address of the data object shlibfdesc now starts with 0xf, which indicates it is in segment 15, the process private shared object data segment.

Refer to Section 2.1.2, “Segment Register addressing” on page 34 for a description of the segment registers and the normal use for each segment.

3.6 Programatic control of loading shared objects

The dlopen() family of subroutines is supported on the AIX operating system. The functions include:

- dlopen
- dlclose
- dlsym
- dlerror

When used appropriately they allow a program to dynamically load shared objects into the address space, use functions in the shared object and then unload the shared object when it is no longer required.

3.6.1 The dlopen subroutine

The dlopen function is used to open a shared object, and dynamically map it into the running programs address space. The specification of the function is as follows:

```
#include <dlfcn.h>

void *dlopen (FilePath, Flags);
const char *FilePath;
int Flags;
```

The FilePath parameter is the full path to a shared object, for example shrobj.o, or libname.so. It can also be a pathname to an archive library that includes the required shared object member name in parenthesis, for example /lib/libc.a(shr1.o).

The Flags parameter specifies how the named shared object should be loaded. The Flags parameter must be set to RTLD_NOW or RTLD_LAZY. If the object is a member of an archive library, the Flags parameter must be OR'ed with RTLD_MEMBER.

The subroutine returns a handle to the shared library that gets loaded. This handle is then used to with the dlsym subroutine to reference the symbols in the shared object. On failure, the subroutine returns NULL. If this is the case, the dlerror subroutine can be used to print an error message.

3.6.2 The dlsym subroutine

The dlopen subroutine is used to load the library. If successful, it returns a handle for use with the dlsym routine to search for symbols in the loaded shared object. Once the handle is available, the symbols (including functions and variables) in the shared object can be found easily. For example:

```
lib_func=dlsym(lib_handle, "locatefn");
error=dlerror();
if (error)
{
   fprintf(stderr, "Error:%s \n",error);
   exit(1);
}
```

The dlsym subroutine accepts two parameters. The first is the handle to the shared object returned from the dlopen subroutine. The other is a string representing the symbol to be searched for.

If successful, the dlsym subroutine returns a pointer that holds the address of the symbol that is referenced. On failure, the dlsym subroutine returns NULL. This, again, can be used with the dlerror subroutine to print an error message as shown above.

3.6.3 The dlclose subroutine

The dlclose subroutine is used to remove access to a shared object that was loaded into the processes' address space with the dlopen subroutine. The subroutine takes as its argument the handle returned by dlopen.

3.6.4 The dlerror subroutine

The dlerror subroutine is used to obtain information about the last error that occurred in a dynamic loading routine (that is, dlopen, dlsym, or dlclose). The returned value is a pointer to a null-terminated string without a final newline.

Once a call is made to this subroutine, subsequent calls without any intervening dynamic loading errors will return NULL.

Applications can avoid calling the dlerror subroutine, in many cases, by examining errno after a failed call to a dynamic loading routine. If errno is ENOEXEC, the dlerror subroutine will return additional information. In all other cases, dlerror will return the string corresponding to the value of errno.

Note

The dlerror() subroutine is not thread-safe since the string may reside in a static area that is overwritten when an error occurs.

3.6.5 Using dynamic loading subroutines

In order to use the dynamic loading subroutines, an application must be linked with the libdl.a library. The shared objects used with the dynamic loading subroutines can be traditional AIX shared objects or shared objects that have been enabled for run-time linking with the -G linker option.

When the dlopen subroutine is used to open a shared object, any initialization routines specified with the -binitfini option, as described in Section 3.2.4, “Initialization and termination routines” on page 60, will be called before dlopen returns. Similarly, any termination routines will be called by the dlclose subroutine.

3.6.6 Advantages of dynamic loading

Use of dynamic linking allows several benefits for application developers:

1. The ability to share commonly-used code across many applications, leading to disk and memory savings.
2. It allows the implementation of services to be hidden from applications.
3. It allows the re-implementation of services, for example, to permit bug and performance fixes or to allow multiple implementations selectable at runtime.

3.6.7 Previous dynamic loading interface

AIX, Version 4.1 did not support the dlopen family of dynamic loading subroutines as standard. Instead, it supported the load, loadbind, and unload subroutines, which could be used to perform similar tasks.

The main difference between the load and dlopen interfaces is that the load interface is normally used to load a shared object to resolve deferred

symbols. That is symbols included as part of an import file that has a blank filename specifier (#!). The symbols are referenced throughout the code, but not resolved (and, therefore, can not be used) until the shared object containing the symbols is loaded into the process address space with the load subroutine and the deferred symbols resolved by calling the loadbind routine. The use of deferred imports like this has essentially been taken care of with the introduction of the run-time linker. Use of the run-time linker is much simpler, as there is no need for the program code itself to perform the actions that cause the shared object to be loaded and the symbols resolved.

Programs written in C++ had to use the loadAndInit and terminateAndUnload subroutines instead of the load and unload subroutines to ensure that constructors and destructors for classes in the shared object being loaded or unloaded were called correctly. The loadAndInit and terminateAndUnload routines are part of the C++ runtime library (libC.a), unlike the load, loadbind, and unload routines, which are part of the C runtime library (libc.a).

The load, loadbind, loadAndInit, unload, and terminateAndUnload subroutines are still available and supported in AIX, Version 4.3. For application code portability reasons, however, the dlopen subroutine family should be used in preference.

Constructors for static classes are called when C++ shared objects are loaded using either the loadAndInit or dlopen routines. Similarly, destructors for static classes are called when C++ shared objects are unloaded using either the terminateAndUnload or dlclose routines. Refer to Section 3.8, “Order of initialization” on page 84 for information on specifying the relative order of initialization between shared objects and between modules within a single shared object. Note that the use of dlopen in C++ applications is complicated by the fact that the C++ symbol names are mangled to support function overloading. The symbol name used as the argument to the dlsym routine must be the mangled symbol name.

3.7 Shared objects and C++

The C++ language, although similar in some respects to the C language, offers many additional facilities. One of these is known as *function overloading*, which makes it possible to have multiple functions with the same name but different parameter lists. This feature means it is not possible to use the function name alone as a unique identifier in the symbol table of an object file. For this reason, function names in C++ are *mangled* to produce the symbol name. The mangling uses a code to indicate the number, type, and ordering of parameters to the function.

It is the name mangling feature of C++ that means the process of creating a shared object, that includes object code created by the C++ compiler, is slightly more complicated than when using code produced by the C compiler.

Although it would be possible to create import and export files manually, the process is time consuming since a unique symbol name is required for each instance of an overloaded function.

3.7.1 Creating a C++ shared object

The good news is that the C++ compiler comes with a `makeC++SharedLib` command that performs most of the dirty work behind the scenes. If you are creating a shared object that uses template functions, you should use the -qmkshrobj option. Refer to Section 3.7.3, “The -qmkshrobj option” on page 83 for more information. The interface to the `makeC++SharedLib` command is similar to the linker interface used to create a shared object based on C language code. For example:

```
cc -c source1.c
ld -G -o libone.so source1.o
```

The process of creating a shared object from C++ code is very similar. The first step is to use the C++ compiler to create the individual object modules that will be placed in the shared object, for example:

```
xlC -c cplussource1.C
```

The second step is to use the `makeC++SharedLib` command to create the shared object. The command has many optional arguments, but in its simplest form, can be used as follows:

```
/usr/vacpp/bin/makeC++SharedLib -o shr1.o cplussource1.o
```

The full path name to the command is not required; however, to avoid this, you will have to add the directory in which it is located to your PATH environment variable. The command is located in the /usr/vacpp/bin directory with the VisualAge C++ Professional for AIX, Version 5 compiler. The command is installed in /usr/ibmcxx/bin with the CSet++ 3.6 compiler and in /usr/lpp/xlC/bin with the CSet++ 3.1 compiler.

Use the -G option to the command when you want to create a shared object enabled for use with run-time linking, or one that uses the libname.so format. For example:

```
/usr/vac/bin/makeC++SharedLib -G -o libone.so source.o
```

If the makeC++SharedLib script is used to build the C++ shared libraries and export symbols, make sure that any system libraries required by the shared object are always specified with the -l option (for example, -lX11) and never by name (for example, /usr/lib/libX11.a) on the command line. This allows the system libraries to be simply referenced by the shared object being created and not go through the special C++ related processing.

3.7.2 Generating an exports file

Another very useful option to the `makeC++SharedLib` command is the ability to save the export file that is generated behind the scenes and normally discarded after use. If saved, this export file can then be used as an import file when creating another shared object. The -e expfile option is used to save the export file. Note that the export file produced does not have an object file name field (#!) on the first line; so, one will have to be manually added, if required.

3.7.3 The -qmkshrobj option

VisualAge C++ Professional, Version 5 for AIX supplies the makeC++SharedLib shell script, as do the previous IBM C++ compiler products for AIX. VisualAge, Version 5, however, also allows the use of a new option to the compiler. The new -qmkshrobj option is used to instruct the compiler to create a shared object from previously created object files and archive libraries. It provides similar functionality to the `makeC++SharedLib` command and, in addition, makes it much easier to create shared objects that use template functions. Refer to Section 4.4, “Shared objects with templates” on page 100 for more information.

For example, to create a shared object shr1.o from the files source1.o and source2.o, use the following command:

```
xlC -qmkshrobj -o shr1.o source1.o source2.o
```

The -G option can also be used in conjunction with the -qmkshrobj option to create an object that uses the new style naming convention and is enabled for run-time linking. For example:

```
xlC -G -qmkshrobj -o libshr1.so source1.o source2.o
```

To specify the priority of the shared object, which determines the initialization order of the shared objects used in an application, append the priority number to the -qmkshrobj option. For example, to create the shared object shr1.o, which has an initialization priority of -100, use the following command:

```
xlC -qmkshrobj=-100 -o shr1.o source1.o source2.o
```

If none of the -bexpall, -bE:, -bexport:, or -bnoexpall options are specified, then using the -qmkshrobj option will force the compiler to generate an exports file that exports all symbols. This file can be saved for use as an import file when creating other shared objects, if desired. This is done using the -qexpfile=filename option. For example:

```
xlC -qmkshrobj -qexpfile=shr1.exp -o shr1.o source1.o source2.o
```

3.7.4 Mixing C and C++ object files

In addition to the mangling of symbol names, the C++ language differs from the C language in the way function arguments are passed on the calling stack. The C language pushes arguments onto the stack right to left, which means the left-most argument is top-most on the stack. For various reasons, the C++ language uses right to left instead. This is termed as linkage, and one can speak of a function having C or C++ linkage.

When mixing C and C++ code together, it is necessary to use a linkage block to call a C routine from a C++ routine. This is to prevent the compiler from mangling the name of the C routine, which would result in a symbol name that could not be resolved. For example, to call the C function, foo, from C++ code, the declaration of foo must be in an external linkage block:

```
extern "C" {
void foo(void);
}
class1::class1(int a)
{
   foo();
}
```

If the declaration of foo was not contained in the extern “C” block, the C++ compiler would mangle the symbol name to foo__Fv.

When mixing C and C++ objects within a single shared object, either the `makeC++SharedLib` command (which uses the C++ compiler) or the -qmkshrobj option of the C++ compiler should be used to create the shared object. Do not use the C compiler or the linker since they may not produce the correct result as they are not aware of C++ constructors, destructors, templates, and other C++ language features.

3.8 Order of initialization

There are situations where the order of initialization of data objects within a program is important to the correct operation of the application. A priority can

be assigned to an individual object file when it is compiled. This is done using the -qpriority option. For example:

```
xlC -c zoo.C -qpriority=-50
```

The C++ compiler and the `makeC++SharedLib` command also support options that can be used to indicate the relative order of initialization of shared objects. There is a slight difference in the way the priority is specified when using each command. When using the C++ compiler, the priority is specified as an additional value with the -qmkshrobj option. For example:

```
xlC -qmkshrobj=-100 -o shr1.o source1.o
```

When using the `makeC++SharedLib` command, the priority is specified with the -p option. For example:

```
makeC++SharedLib -p -100 -o shr1.o source1.o
```

Priority values can also be indicated within C++ code by using the priority compiler directive as follows:

```
#pragma priority(value)
```

These values alter the order of initialization of data objects within the object module.

3.8.1 Priority values

Priority values may be any number from -214782623 to 214783647. A priority value of -214782623 is the highest priority. Data objects with this priority are initialized first. A priority value of 214783647 is the lowest priority. Data objects with this priority are initialized last. Priority values from -214783648 to -214782624 are reserved for system use. If no priority is specified, the default priority of 0 is used.

The explanation of priority values uses the example data objects and files shown in Figure 9 on page 86.

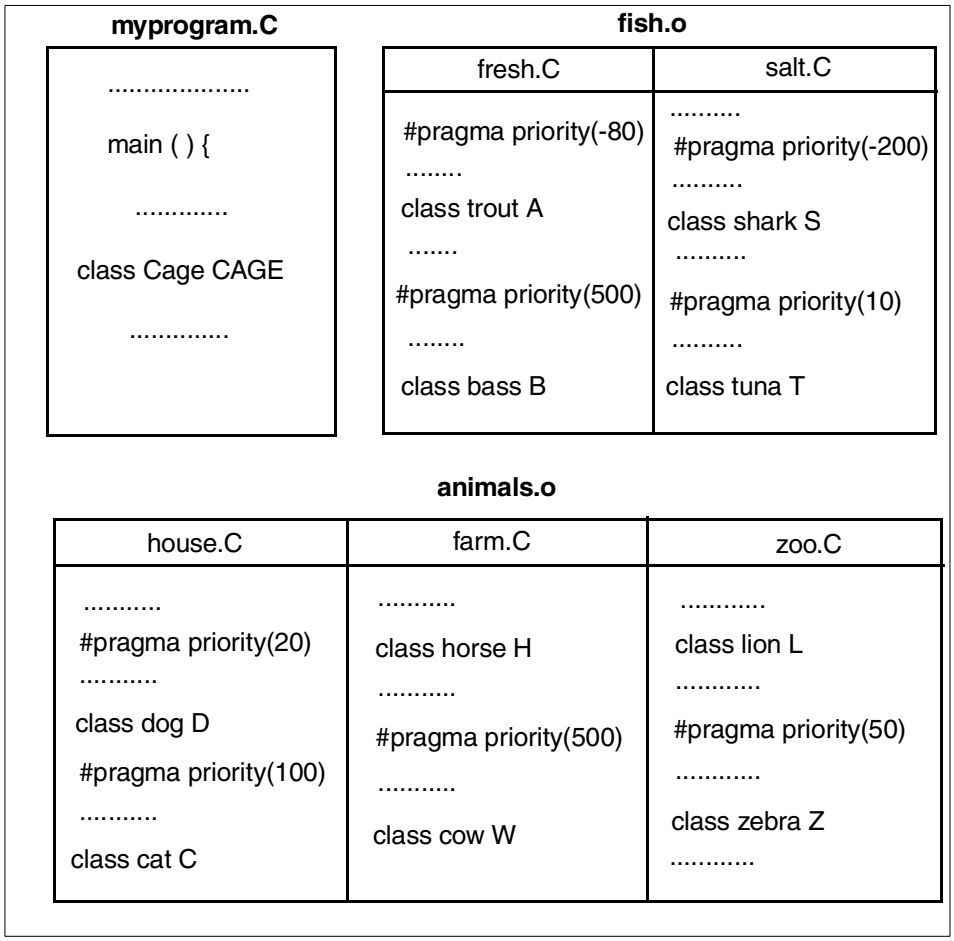


Figure 9. Illustration of objects in fish.o and animals.o

This example shows how to specify priorities when creating shared objects to guarantee the order of initialization. The user should first of all determine the order in which they want the objects to be initialized, both within each file and between shared objects:

1. Develop an initialization order for the objects in house.C, farm.C, and zoo.C:
   a. To ensure that the object lion L in zoo.C is initialized before any other objects in either of the other two files in the shared object animals.o, compile zoo.C using a -qpriority=nn option with nn less than zero so

that data objects have a priority number less than any other objects in farm.C and house.C:

```
xlC zoo.C -c -qpriority=-50
```

b. Compile the house.C and farm.C files without specifying the -qpriority=nn option. This means the priority will default to zero. This means data objects within the files retain the priority numbers specified by their #pragma priority(nn) directives:

```
xlC house.C farm.C -c
```

c. Combine these three files into a shared library. Use the `makeC++SharedLib` command to construct the shared object animals.o with a priority of 40:

```
makeC++SharedLib -o animals.o -p 40 house.o farm.o zoo.o
```

2. Develop an initialization order for the objects in fresh.C and salt.C, and use the #pragma priority(value) directive to implement it:

a. Compile the fresh.C and salt.C files

```
xlC -c fresh.C salt.C
```

b. To assure that all the objects in fresh.C and salt.C are initialized before any other objects, including those in other shared objects and the main application, use `makeC++SharedLib` to construct a shared object fish.o with a priority of -100:

```
makeC++SharedLib -o fish.o -p -100 fresh.o salt.o
```

Because the shared object fish.o has a lower priority number (-100) than animals.o (40), when the files are placed in an archive file with the `ar` command, the objects are initialized first.

3. To create a library that contains the two shared objects, so that the objects are initialized in the order you have specified, use the `ar` command. To produce an archive file, libprio.a, enter the command:

```
ar rv libprio.a animals.o fish.o
```

where:

`libprio.a` is the name of the archive file that will contain the shared library files, and `animals.o` and `fish.o` are the two shared files created with `makeC++SharedLib`.

4. Compile the main program, myprogram.C, that contains the function main to produce an object file, myprogram.o. By not specifying a priority, this file is compiled with a default priority of zero, and the objects in main have a priority of zero:

```
xlC -c myprogram.C
```

5. Produce an executable file, animal_time, so that the objects are initialized in the required order, enter:

```
xlC -o animal_time main.o -lprio -L.
```

When the animal_time executable is run, the order of initialization of objects is as shown in Table 10.

Table 10. Order of initialization of objects in prriolib.a

Object	Priority value	Comment
fish.o	-100	All objects in fish.o are initialized first because they are in a library prepared with `makeC++SharedLib -p -100` (lowest priority number, -p -100 specified for any files in this compilation).
shark S	-100(-200)	Initialized first in fish.o because within file, `#pragma priority(-200)`.
trout A	-100(-80)	`#pragma priority(-80)`
tuna T	-100(10)	`#pragma priority(10)`
bass B	-100(500)	`#pragma priority(500)`
myprog.o	0	File generated with no priority specifications; default is 0.
CAGE	0(0)	Object generated in main with no priority specifications; default is 0.
animals.o	40	File generated with makeC++SharedLib with -p 40.
lion L	40(-50)	Initialized first in file animals.o compiled with `-qpriority=-50`.
horse H	40(0)	Follows with priority of 0 (since -qpriority=nn not specified at compilation and no #pragma priority(nn) directive).
dog D	40(20)	Next priority number (specified by `#pragma priority(20)`).
zebra N	40(50)	Next priority number from `#pragma priority(50)`.

Object	Priority value	Comment
cat C	40(100)	Next priority number from `#pragma priority(100)`.
cow W	40(500)	Next priority number from `#pragma priority(500)`.

3.9 Troubleshooting

The following tips and hints can be used to help linking and loading of C and C++ programs on AIX, Version 4.3.

3.9.1 Link failures

When linking large applications with many libraries, the linker may exit with some strange errors referring to BUMP or indicating that the binder was killed. This may be because of low paging space or because of low resource limits for the user invoking the command. The AIX linker offers a great deal more functionality than traditional UNIX linkers, but it does require a reasonable amount of virtual memory, particularly when linking large applications with many libraries.

If this type of error is encountered, check the paging space available on the machine. In addition, check the resource limits for the user invoking the linker. This can be done with the `ulimit` command.

3.9.1.1 Unresolved symbols

When linking your application with many libraries, particularly those supplied by a third party product, such as a database, it is not unusual during the development cycle to see a linker error warning of unresolved symbols.

The linker supports options that can be used to generate linker log files. These log files can then be analyzed to determine the library or object file that references the unresolved symbol. This can help in tracking interdependent or redundant libraries being used in error.

The -bmap:*filename* option is used to generate an address map. Unresolved symbols are listed at the top of the file, followed by imported symbols.

The -bloadmap:*filename* option is used to generate the linker log file. It includes information on all of the arguments passed to the linker along with the shared objects being read and the number of symbols being imported. If an unresolved symbol is found, the log file produced by the -bloadmap option

lists the object file or shared object that references the symbol. In the case of using libraries supplied by a third party product, you can then search the other libraries supplied by the product in an effort to determine which one defines the unresolved symbol. This is particularly useful when dealing with database products that supply many tens of libraries for use in application development.

3.9.2 Runtime tips

If large parts of the shared libraries are paged in all at once because of C++ calls or many references between libraries, it may be faster to read the library rather than demand-page it into memory. Remove read-other permission from the applications shared libraries and see if the loading performance improves. If it does, then reset the original permissions and set the following environment variable:

```
LDR_CNTRL = PREREAD_SHLIB
```

By using this environment variable, the libraries are read very quickly into the shared memory segment.

Chapter 4. Using C++ templates

Templates are an area of the C++ language that provide a great deal of flexibility for developers. The recent ANSI C++ standard defines the language facilities and features for templates. Unfortunately, the standard does not specify how a compiler should implement templates. This means that there are sometimes significant differences between the methods used to implement templates in compiler products from different vendors.

Developers porting C++ code that uses templates to the AIX platform sometimes have problems with the implementation model. The main problems experienced are:

- Long compile and link times
- Linker warnings of duplicate symbols
- Increase in code and executable size

All of the above problems are generally caused by inefficient use of the AIX implementation of templates. The number of problems experienced will depend on the platform the code is being ported from and the template implementation method used on that platform. Sometimes, the problems can be fixed on AIX by simply adding a few compiler options. In other instances, the code layout needs to be changed in order to utilize the most efficient implementation method on AIX. In most of these rare cases, the code changes are backwards compatible with the original platform the code is being ported from. This is very important for developers who maintain a single source tree that must compile correctly on multiple platforms.

4.1 AIX template implementations

The template mechanism provides a way of defining general container types, such as list, vector, and stack, where the specific type of the elements is left as a parameter. Two types of templates can be defined:

Class templates Specify how individual classes can be constructed.

Function templates Specify how individual functions can be constructed.

Regardless of the type of template being used, the code is essentially split into three parts:

Template declaration This is the part of the source code that declares the template class or function. It does not necessarily contain the definition of the class or function, although

it may optionally do so. For example, a template declaration may describe a Stack template class as shown in Figure 10.

Template definition This portion of code is the definition of the template function itself or the template class member functions. For example, using the Stack class template, this portion of code would define the member functions used to manipulate the stack as shown in Figure 11 on page 93.

Template instance The template instance code is generated by the compiler for each instance of the template. For example, this would be the code to handle a specific instance of the stack template class, such as a stack of integer values.

The difference between the components is that the template declaration must be visible in every compilation unit that uses the template. The template definition and code for each instance of the template need only be visible once in each group of compilation units that are linked together to make a single executable.

```
template <class T> class stack
{
private:
   T* v;
   T* p;
   int sz;
public:
   stack( int );
   ~stack();
   void push( T );
   T pop();
};
```

Figure 10. Stack template declaration

```
template <class T> stack<T>::stack( int s )
{
        v = p = new T[sz=s];
}

template <class T> stack<T>::~stack()
{
        delete [] v;
}

template <class T> void stack<T>::push( T a )
{
        *p++ = a;
}

template <class T> T stack<T>::pop()
{
        T ret = *p;
        p--;
        return ret;
}
```

Figure 11. Stack template member function definition

4.1.1 Generated function bodies

When you use class templates and function templates in your program, the compiler automatically generates function bodies for all template functions that are instantiated. The compiler follows four basic rules to determine when to generate template functions. The compiler applies the rules in the following order:

1. If a template declares a function to have internal linkage, the function must be defined within the same compilation unit. The compiler generates the function with internal linkage, and it is not shared with other compilation units. This is the case if the template class has inline member functions.
2. If a template function is instantiated in a compilation unit, but it is not declared to have internal linkage, the compiler looks for a definition of the function in the same compilation unit. If a definition is found, the compiler generates a function body in the same compilation unit.
3. If a template function is instantiated in a compilation unit, and the function is not defined in the same compilation unit, but certain other conditions are met, the compiler generates the necessary function definitions during a

special prelink phase of the compilation. This is the case when the -qtempinc option is in use.

4. If none of the preceding rules applies, the compiler does not generate the definition of the template function. It must be defined in another compilation unit.

4.2 Simple code layout method

The simplest method of using template code is to include both the declaration and definition of the template in every compilation unit that uses instances of the template. From a code layout point of view, this is very easy since the template declaration and definition can be kept in a single header file. Using the stack example, the code in Figure 10 on page 92 and Figure 11 on page 93, would be combined into a single header file, for example, stack.h, which is then included by every compilation unit that wishes to use the template. Alternatively, the header file for a template declaration can include the source file that contains the template definition. Using the stack template example, the header file, stack.h, would #include the source file, stack.C.

There are a number of disadvantages to using this method. Some of them can be overcome; others can not.

4.2.1 Disadvantages of the simple method

The first disadvantage is that using the header files can become complicated, particularly when other header files need to declare an instance of the template. In order to do this, they must #include the stack.h file, which potentially leads to multiple #include's of the file, resulting in multiple definitions of the member functions. This problem can be fixed with the addition of preprocessor macros in the header file to protect against multiple #include operations. For example:

```
#ifndef stack_h
#define stack_h
....
....declaration and definition of stack template
....
#endif
```

Using the macros shown above, the contents of the header file will only appear once in the compilation unit, regardless of the number of times the file is included. This resolves the problems of multiple definitions within a compilation unit.

4.2.1.1 Template code bloat

The second disadvantage is that the code for each template instance will potentially appear multiple times in the final executable, resulting in the twin problems of large executable size and multiple symbol definition warnings from the linker.

As an example, consider an executable made up of two compilation units, main.C and functions.C. If both compilation units include the stack.h header file, and declare variables of the type stack<int>, then after the first stage of compilation, both object files, main.o and functions.o, will contain the code for the member functions of the stack<int> class. When the system linker parses the object files to create the final executable, it can not remove the duplicate symbols since, by default, each compilation unit is treated as an atomic object by the linker. This results in duplicate symbol linker warnings and a final executable that contains redundant code.

The size of the final executable can be reduced by using the compiler option, -qfuncsect, when compiling all of the source code modules. This option causes the compiler to slightly change the format of the output object files. Instead of creating an object file, which contains a single code section (CSECT) and must be treated by the system linker as an atomic unit, the compiler creates an object file where each function is contained in its own CSECT. This means that the object files created are slightly larger than their default counterparts since they contain extra format information in addition to the executable code. This option does not remove the linker warnings since at link time, there are still multiple symbol definitions. The benefit of this option is that the linker can discard multiple, identical function definitions by discarding the redundant CSECTs, resulting in a smaller final executable. When the -qfuncsect option is not used, the compiler can not discard the redundant function definitions if there are other symbols in the same CSECT that are required in the final executable.

Refer to Section 4.5, “Virtual functions” on page 103 for information on another potential cause of C++ code bloat.

4.2.1.2 Template compile time

The use of the -qfuncsect option reduces the code size of the final executable. It does not resolve the other disadvantage of using this method - that of longer than required compile times. The reason for this is that each compilation unit contains the member functions for the templates that it instantiates. Using an extreme example with the stack class, consider the situation where an application is built from 50 source files, and each source file instantiates a stack<int> template. This means the member functions for

the class are generated and compiled 50 times, yet the result of 49 of those compiles are discarded by the linker since they are not needed. In the example used here, the code for the stack class is trivial; so, in absolute terms, the time saved would be minimal. In real life situations, where the template code is complex, the time savings that can be made when compiling a large application are considerable.

Because of the fact that not all of the disadvantages of the simple template method can be overcome, it is only recommended for use when experimenting with templates. An alternative method can be used, which solves all of the problems of the simple method and scales very well for large applications.

4.3 Preferred template method

The preferred method of template instantiation on AIX basically means letting the compiler decide which template code to instantiate as a final step in the compile and link process. This solves the long compile time disadvantage of the simple template method because the compiler need only compile each template instance once.

This method requires that the declaration and definition of the template are kept in separate files. This is because only the template declaration must be included in every compilation unit that uses the template. If the definition of the template were also in the header file, it would also be included in the source file, and thus compiled, resulting in a situation similar to that in the simple method.

The preferred template model can also benefit from the use of the -qfuncsect compiler option since it means the linker can discard code sections that are not referenced in the final executable.

The template declaration should be left in the header file, as in the simple template method. The definition of the template member functions needs to be in a file with the same basename as the header file but with a .c (lower case C) filename extension.

Note

By default, the file containing the template definition code must have the same name as the template declaration header file, but with a filename extension of .c (lowercase c), even though this extension normally indicates a C language source file. It must also exist in the same directory as the template declaration header file. If the template definition file is not in the same directory, has a different basename, or has a different filename extension (such as .C, .cxx, or .cpp, which are normally used for C++ source files), then the compiler will not detect the presence of the template code to be used with the template declaration header file.

Using the stack template example introduced earlier, the template declaration shown in Figure 10 on page 92 would be in the file stack.h, while the template definition code shown in Figure 11 on page 93 would be in the file stack.c in the same directory. If the template definition code file was named stack.cxx or stack_code.c, then the compiler will not associate the file with the template declaration in the stack.h header file.

The name of the template definition file can be changed, if desired, using the implementation pragma directive as follows:

```
#pragma implementation(string-literal)
```

where `string-literal` is the path name for the template definition file enclosed in double quotes. For example, if the stack template definition code were to be stored in the file, stack_code.cxx, then the stack.h header file would have the following directive:

```
#pragma implementation("stack_code.cxx")
```

Once the structure of the source code has been altered to conform to the required layout, the templates can be used in the preferred way.

4.3.1 The -qtempinc option

The -qtempinc option is used when compiling source code that instantiates templates. When no directory is specified with the option, the compiler will create a directory called tempinc in the current directory. For example:

```
xlC main.C -qtempinc
```

The user may optionally specify the name of a directory to use for storing the information on the templates to be generated. This allows the same tempinc directory to be used when creating an executable that consists of object files that are compiled in different directories. For example:

```
xlC -c file1.C file2.C -qtempinc=../app1/templates
cd ../app1
xlC -o app1 main.C ../src/file1.o ../src/file2.o -qtempinc=./templates
```

The tempinc directory is used to store information about the templates that are required to be generated. When invoked with the -qtempinc option, the compiler collects information about template instantiations and stores the information in the tempinc directory. As the last step of the compilation before linking, the compiler generates the code for the required template instantiations. It then compiles the code and includes it with the other object files and libraries that are passed to the linker to create the final executable.

If the compiler detects a code layout structure that enables the preferred template method to be used, it will automatically enable the -qtempinc option, even if it was not specified on the command line. This causes the template instantiation information to be stored in the tempinc directory. If you want to specify a different directory, you should explicitly use the -qtempinc=dirname option on the command line. If you want to prevent the compiler from automatically generating the template information, which may be the case when creating a shared object, then use the -qnotempinc option. Refer to Section 4.4, “Shared objects with templates” on page 100 for more information on the use of the -qnotempinc option when creating shared objects.

One important point to note about the -qtempinc option is that you should use the same value when compiling all compilation units that will be linked together. In other words, do not compile half of the application with -qtempinc, and the other half with -qtempinc=dirname. Only one tempinc directory can be specified on the final C++ compile line that is used to link the application, which means that half of the template instance information will be missing. If more than one tempinc option is specified on the command line, the last one encountered will prevail.

4.3.2 Contents of the tempinc directory

The compiler generates a file in the tempinc directory for each template header file that has templates instantiated. The file has the same name as the header file, but with a .C (uppercase C) filename extension. The compiler generates the file when it detects the first instantiation of a template that is declared in the header file with the same name. Information on the subsequent instances of the template is added to the file.

As the final step of the compilation before linking, the compiler compiles all of the files in the tempinc directory and passes the object files to the linker along with the user specified files.

The contents of a template information file are as follows:

```
/*0965095125*/#include "/redbooks/examples/C++/stack.h"           1
/*0000000000*/#include "/redbooks/examples/C++/stack_code.cxx"    2
template stack<int>::stack(int);                                  3
template stack<int>::~stack();                                    4
template void stack<int>::push(int);                              5
template int stack<int>::pop();                                   6
```

The line numbers at the end of each line have been added for reference purposes. The code on line 1 includes the header file that declares the template. The comment at the start of the line is a timestamp and is used by the compiler to determine if the header file has changed, which would require the template instance information file to be recompiled.

The code on line 2 includes the template implementation file that corresponds to the header file in line 1. A timestamp consisting of all zeros indicates that the compiler should ignore the timestamp. The file may include other header files that define the classes that are used in template instantiations. For example, if there was a user defined class Box, and the compiler detected and instantiation of stack<Box>, then the header file that defines the class Box would be included in the instance information file.

The subsequent lines in the example shown above cause the individual member functions to be instantiated.

4.3.3 Forcing template instantiation

You can, if you wish, structure your program so that it does not use automatic template instantiation. In order to do this, you must know which template classes and functions need to be instantiated.

The #pragma define directive is used to force the instantiation of a template, even if no reference is made to an instance of the generated template. For example:

```
#pragma define(stack<double>);
```

This, however, means that the template implementation file needs to be included in the compilation units that have the #pragma define directives, which results in the same disadvantages of the simple template method described in Section 4.2, “Simple code layout method” on page 94.

An alternative to this is to manually emulate the process used by the compiler to automatically create the appropriate template instances. Using the stack class as an example, the following compilation unit could be used to force the creation of the desired stack template classes, even though no objects of those types are referenced in the source code:

```
#include "/redbooks/examples/C++/stack.h"                          1
#include "/redbooks/examples/C++/stack_code.cxx"                   2
#include "/redbooks/examples/C++/Box.h" // definition of class Box 3
#pragma define(stack<int>);                                        4
#pragma define(stack<Box>);                                        5
#pragma define(stack<char>);                                       6
#pragma define(stack<short>);                                      7
```

This type of method will be useful when creating shared objects with the `makeC++SharedLib` command. Users of the VisualAge C++ Professional for AIX, Version 5 compiler should use the -qmkshrobj option instead. Refer to Section 4.4, “Shared objects with templates” on page 100 for more information.

4.4 Shared objects with templates

Templates are usually declared in a header file. Each time a template is used, code is generated to instantiate the template with the desired parameters. Most C++ compilers work with a ‘template repository’. No template code is generated at compile time; the compiler just remembers where the template code came from. Then, at link time, as the compiler/linker puts all parts together, it notices which templates actually need to be generated. The code is then produced, compiled, and linked into the application.

This becomes a problem when using templates with shared libraries, where no actual linking takes place. So, one must make sure that the template code is generated when producing the shared library.

Therefore, one should keep track of compilation and inclusion of template instantiations.This would mean that one has to manually keep track of all the template instantiation and address them during the linking phase.

It is here that the VisualAge C++ Professional for AIX, Version 5 compiler has a noticeable improvement over previous versions of C++ compilers for AIX. The compiler, like the makeC++SharedLib command, can be used to create a shared object from object files using the -qmkshrobj option.

This option, together with the -qtempinc option, should be used in preference to the `makeC++SharedLib` command when creating a shared object that uses

templates. The advantage of using these options instead of `makeC++SharedLib` is that the compiler will automatically include and compile the template instantiations in the tempinc directory.

4.4.1 Templates and makeC++SharedLib

The `makeC++SharedLib` command is supplied with all IBM C++ command line compilers for the AIX platform. The command is a shell script that gathers the supplied input and then calls the linker to create the shared object.

When creating a shared object that uses templates, the `makeC++SharedLib` command needs to somehow find information on the templates that are to be instantiated. Because the script calls the linker, and not the compiler, it does not look at the contents of the tempinc directory. This means the method of creating a shared object that uses templates relies on either using the simple template method code layout, as described in Section 4.2, “Simple code layout method” on page 94, or forcing templates to be instantiated, as described in Section 4.3.3, “Forcing template instantiation” on page 99.

The best method to use will depend on the circumstances. Using the simple code layout method means that all the required templates are generated automatically. However, it also comes with the disadvantages of slower compile times and larger code size. Forcing the templates to be instantiated is better from both the code size and compile time aspect, but it does mean that the user needs to maintain files that instantiate the required templates.

Suppose you want to create a shared object from the following two source files, which use the preferred code layout method.

File source1.C contains the following code:

```
#include “stack.h”
stack<int> counter1;
void function1(int a)
{
   counter1.push(a);
}
```

The file source2.C contains the following code:

```
include “stack.h”
stack<int> counter2;
void function2(int a)
{
   counter2.push(a);

}
```

Using the `makeC++SharedLib` command, an attempt is made to create a shared object as follows:

```
xlC -c source1.C source2.C
/usr/vacpp/bin/makeC++SharedLib -o shr1.o -p0 source1.o source2.o
ld: 0711-317 ERROR: Undefined symbol: .stack<int>::stack(int)
ld: 0711-317 ERROR: Undefined symbol: .stack<int>::~stack()
ld: 0711-317 ERROR: Undefined symbol: .stack<int>::push(int)
ld: 0711-345 Use the -bloadmap or -bnoquiet option to obtain more
information.
```

The command failed, and based on the output, it is easy to see that the required template functions have not been instantiated. At this point, note that because the code uses the preferred code layout method, the compiler has, in fact, automatically created the file tempinc/stack.C, which, if compiled, would supply the required template definitions. You can, if you wish, copy this file and make it an explicit compilation unit as part of your source code. In this case, that would mean adding the following command to the sequence:

```
xlC -c tempinc/stack.C -o stack.o
```

The object, file stack.o, would then be passed to the `makeC++SharedLib` command along with source1.o and source2.o.

4.4.2 Templates and -qmkshrobj

Users of the VisualAge C++ Professional for AIX, Version 5 compiler should use the -qmkshrobj option in preference to the `makeC++SharedLib` command when creating a shared object. Because the option is a compiler option, it will automatically look in the tempinc directory (or the directory specified with the -qtempinc=dirname option) for the automatically generated template instance information. Using the same source files as described in the section above, the following commands can be used to create the shared object:

```
xlC -c source1.C source2.C
xlC -qmkshrobj -o shr1.o source1.o source2.o
```

This time the command works correctly since the compiler looks in the tempinc directory. Remember to use the same -qtempinc option (if any) that was used when compiling the modules being used to create the shared object.

This option solves the problems associated with creating shared objects that use template classes and functions. If you want to create a shared object that contains pre-instantiated template classes and functions for use by other developers, then you can create an additional compilation unit that explicitly defines the required templates using the #pragma define directive.

4.5 Virtual functions

In general, when writing C++ code, you should try and avoid the use of virtual functions. They are normally encoded as indirect function calls, which are slower than direct function calls.

Usually, you should not declare virtual functions inline. In most cases, declaring virtual functions inline does not produce any performance advantages. Consider the following code sample:

```
class Base {
public:
   virtual void foo() { /* do something */ }
};

class Derived: public Base {
public:
   virtual void foo() { /* do something else */ }
};

int main() {
   Base*  b = new Derived();
   b->foo(); // not inlined
}
```

In this example, b->foo() is not inlined because it is not known until runtime which version of foo() must be called. This is by far the most common case.

There are cases, however, where you might actually benefit from inlining virtuals; for example, if Base::foo() was a really hot function it would get inlined in the following code:

```
int main() {
   Base  b;
   b.foo();
}
```

If there is a non-inline virtual function in the class, the compiler generates the virtual function table in the first file that provides an implementation for a virtual function; however, if all virtual functions in a class are inline, the virtual table and virtual function bodies will be replicated in each compilation unit that uses the class. The disadvantage to this is that the virtual function table and function bodies are created with internal linkage in each compilation unit. This means that even if the -qfuncsect option is used, the linker can not remove the duplicated table and function bodies from the final executable. This can result in very bloated executable size.

Chapter 5. POSIX threads

This chapter is divided into two main topics. The first one covers the design of multi-threaded applications using POSIX threads in the UNIX system. The second main topic covers the user implementation point of view on AIX systems.

5.1 Designing threaded application with pthreads

As a multi-tasking operating system, UNIX can execute several process simultaneously in a safe environment, where each process has its own address space, resources, and execution flow. This computational model guarantees complete independence among the processes and their data that ultimately gives a stable environment.

In some cases, this traditional UNIX process model is not suitable, especially when the best computational model or the only feasible implementation requires several tasks working on the same data simultaneously. The communication effort among the processes and its control can be very difficult to code or very heavy in computing time. Performance is also a very important reason for using parallel pieces of code, especially in multi-processor systems.

To address this kind of implementation and to ensure a portable code, the Portable Operating System Interface (POSIX) standardized an application programming interface (API) for thread implementation that is called *pthreads*. This API is supported in a wide range of platforms in several levels of evolution. Draft 10 of POSIX pthreads was implemented as the final standard. This standard was, in turn, incorporated into the UNIX98 specification. AIX 4.3.3 supports both Draft 7 of the POSIX pthreads standard and the UNIX98 pthreads standard (which is a superset of the POSIX pthreads standard).

5.1.1 Threads and UNIX processes

A single threaded process can be represented schematically, as shown in Figure 12 on page 106, where a process consists of an address space, resources, and a single flow of control.

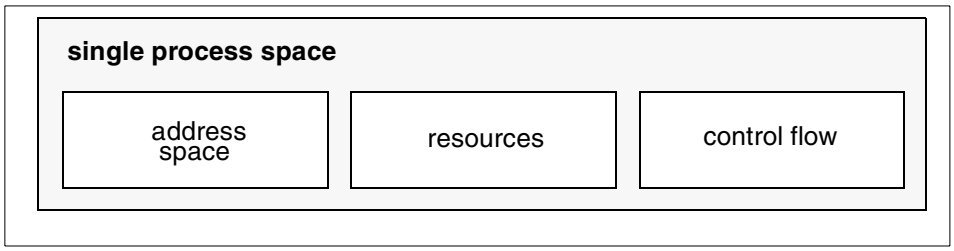


Figure 12. Single-thread process

In Figure 13, we can see a representation of a multi-threaded process. In a multi-threaded process, all threads share the same address space and resources.

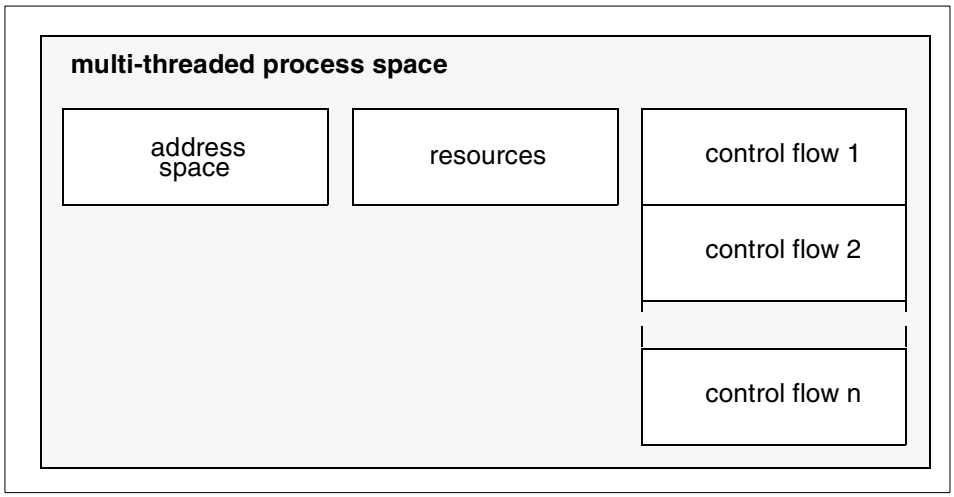


Figure 13. Multi-threaded process

In traditional, non-threaded, process-based systems, a process has a set of properties. In multi-threaded systems, these properties are divided between processes and threads.

A process in a multi-threaded system is the changeable entity. It must be considered as an execution frame. It has all the traditional process attributes, such as:

- Process ID
- Process group ID

- User ID
- Group ID
- Environment, such as current working directory and file descriptors

A process also provides a common address space and common system resources:

- File descriptors
- Signal actions
- Shared libraries
- Inter-process communication tools (such as message queues, pipes, semaphores, and shared memory)

A thread is the schedulable entity. It has only those properties that are required to ensure its independent flow of control. These include the following properties:

- Stack
- Scheduling properties (such as policy or priority)
- Set of pending and blocked signals
- Some thread-specific data

5.1.1.1 Process duplication and threads

On a traditional, single-threaded UNIX system forking a process, calling the fork function creates a new process, known as the child, that is a copy of the calling process, known as the parent.

On a multi-threaded environment, each process has at least one thread. Therefore, duplicating a process ultimately means duplicating a thread.

There are two reasons why AIX programmers may need to fork a program:

- To create a new flow of control within the same program
- To create a new process running a different program

In a multi-threaded program, creating a new flow of control within the same program is provided by the pthreads API. The fork subroutine should, as such, be used only to run new programs.

The fork subroutine duplicates the parent process but duplicates only the calling thread; the child process is a single-threaded process, even if the parent process is multi-threaded. The calling thread of the parent process

becomes the initial thread of the child process; it may not be the initial thread of the parent process. If the initial thread of the child process returns from its entry-point routine, which is equivalent to a single threaded process returning from main, then the child process terminates.

When duplicating the parent process, the fork subroutine also duplicates all the synchronization variables, including their state. Thus, for example, mutexes may be held by threads that no longer exist in the child process, and any associated resources may be inconsistent.

Note

It is strongly recommended to use the fork subroutine only to run new programs and to call one of the exec subroutines as soon as possible after the call to the fork subroutine in the child process.

For new flow of control creation, the *pthread_create* function should be used.

If an application must consist of multiple processes, it is recommended to call the fork subroutine to create the desired child processes before calling any pthreads routines in the parent process.

Unfortunately, the rule explained above does not address the needs of multi-threaded libraries. Application programs may not be aware that a multi-threaded library is in use and will feel free to call any number of library routines between the fork and the exec subroutines. Occasionally, this situation will result in a dead-lock situation.

To address this problem, the pthreads' API provides a function, *pthread_atfork*, that can be used to set three fork handler functions as follow:

Prepare The prepare fork handler is called just before the processing of the fork subroutine begins.

Parent The parent fork handler is called in the parent process just after the processing of the fork subroutine is completed.

Child The child fork handler is called in the child process just after the processing of the fork subroutine is completed.

The pthread_atfork subroutine provides a way for multi-threaded libraries to protect themselves from innocent application programs that call the fork subroutine. It also provides multi-threaded application programs with a

standard mechanism for protecting themselves from calls to the fork subroutine in a library routine or the application itself.

The fork handlers specified by the program can be used to release mutexes and perform general tidy-up duties before and after the fork call.

5.1.2 Lightweight process -LWP

A thread can be implemented in a *kernel-level* or in a *user-level* of abstraction. In the kernel-level, each thread is a kernel entity, which, like processes and interrupt handlers, owns its kernel data structure and can be handled by the system scheduler. A kernel thread is also known as a *lightweight processes*, or LWP.

On the other hand, the *user-level* threads are represented by data structures in the process address space. They are mapped to kernel-level threads in order to be executed. The way this mapping is done is called the thread model. There are three possible thread models, corresponding to three different ways to map user threads to kernel threads.

- M:1 model
- 1:1 model
- M:N model

The mapping of user threads to kernel threads is done using virtual processors. A virtual processor (VP) is a pthreads library entity that is usually implicit. For a user thread, the virtual processor behaves as a CPU for a kernel thread. In the library, the virtual processor is a kernel thread or a structure bound to a kernel thread.

In the M:1 model, all user threads are mapped to one kernel thread; all user threads run on one VP. The mapping is handled by a library scheduler. All user threads programming facilities are completely handled by the library. This model can be used on any system, especially on traditional single-threaded systems. Figure 14 on page 110 illustrates this model.

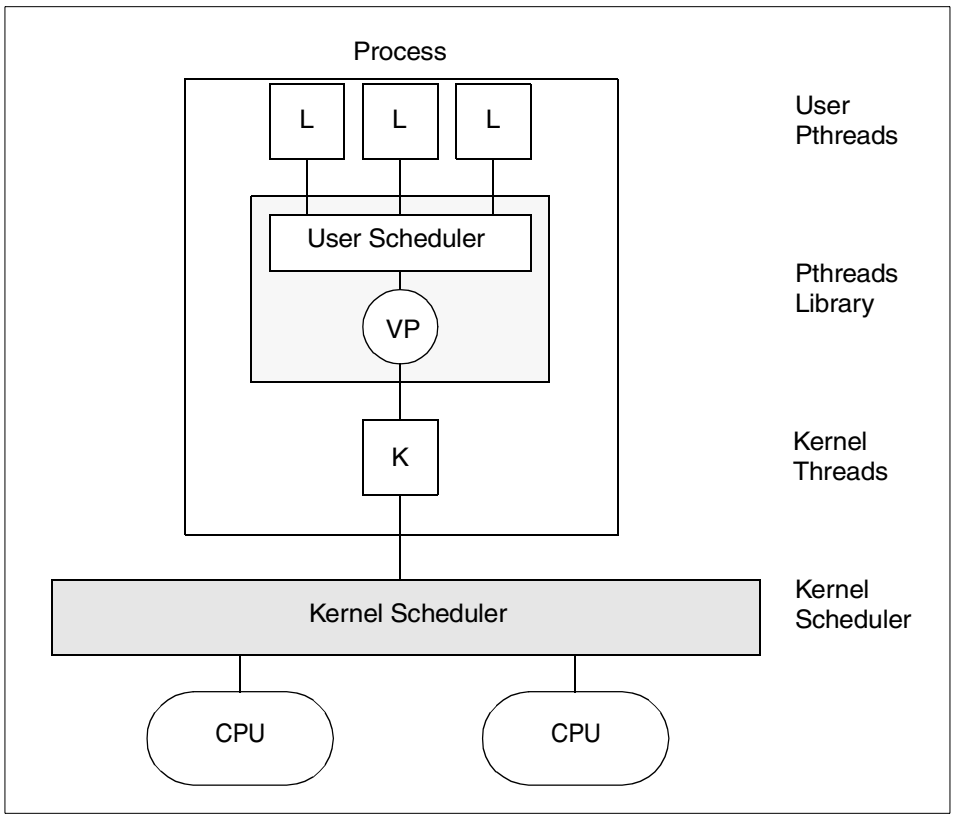


Figure 14. M:1 thread model

In the 1:1 model, each user thread is mapped to one kernel thread; each user thread runs on one VP. Most of the user threads programming facilities are directly handled by the kernel threads. The 1:1 threads model is shown in Figure 15 on page 111.

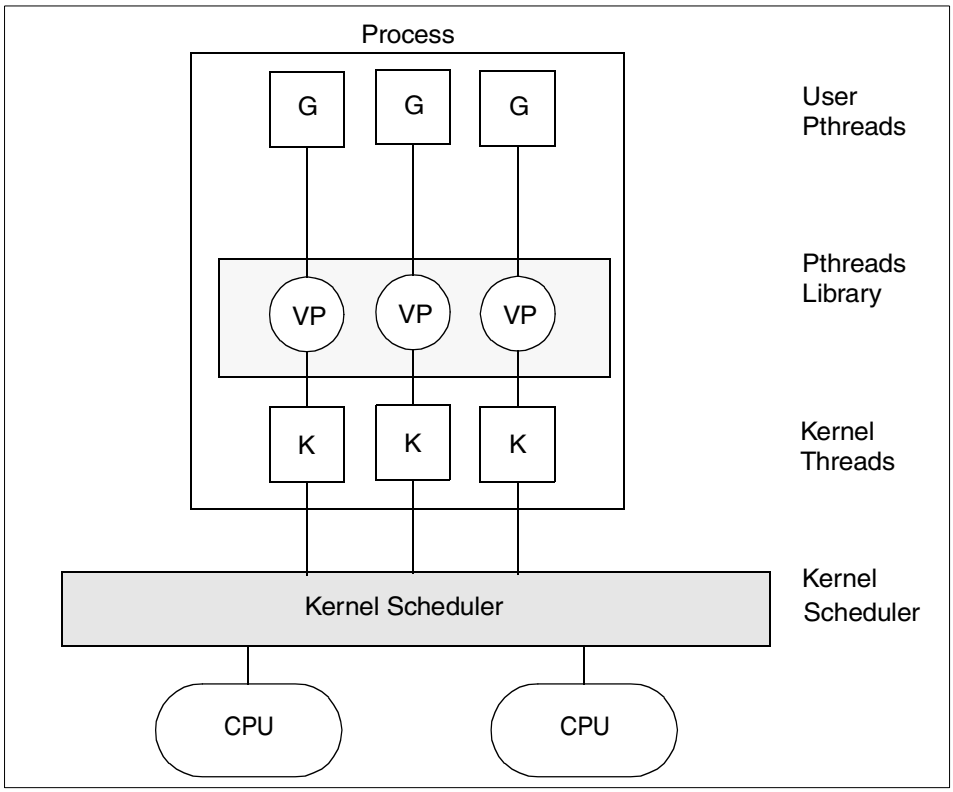


Figure 15. 1:1 thread model

In the M:N model, all user threads are mapped to a pool of kernel threads; all user threads run on a pool of virtual processors. A user thread may be bound to a specific VP, as in the 1:1 model. All multiplexed user threads share the VPs on the pool. This is the most efficient thread model, although it is also the most complex from a library implementation point of view. In this model, the user threads programming facilities are shared between the threads library and the kernel threads. This is shown in Figure 16 on page 112.

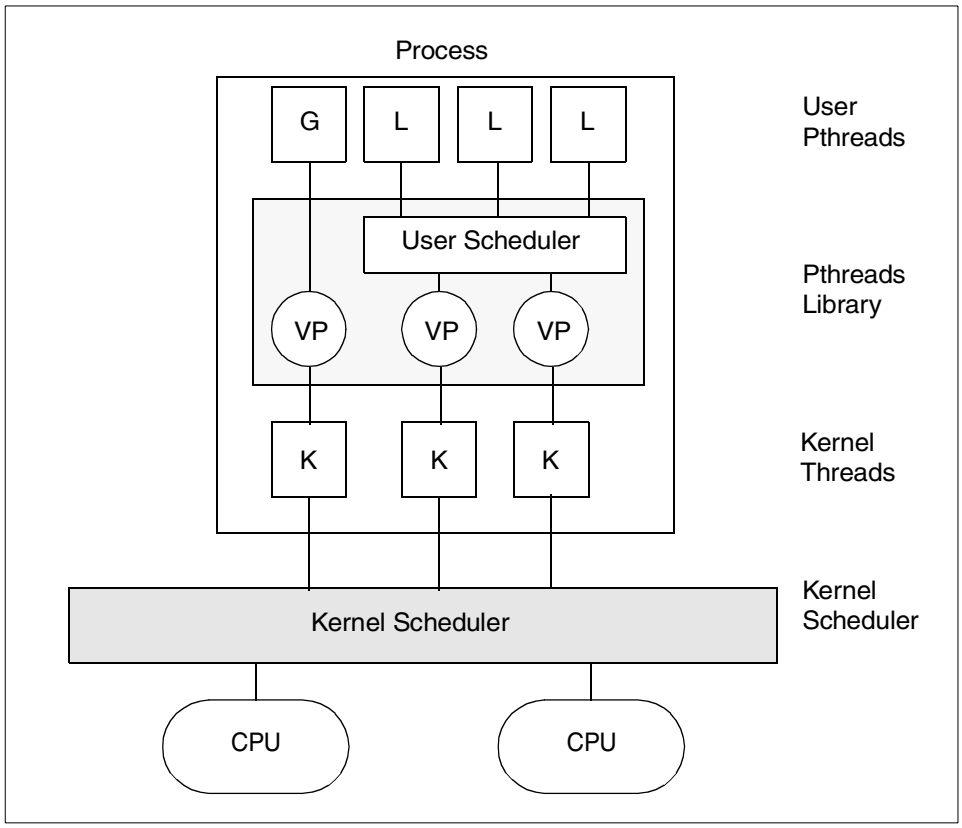


Figure 16. M:N thread model

5.1.3 Thread scheduling

Depending upon the thread model in use, we can determine the contention scope for the threads. In the 1:1 Model, it is called System Contention Scope scheduling, or SCS, which means that all scheduling of the thread is handled by the kernel, and the thread competes with all other threads on the system for CPU time. With both the M:1 and M:N models, the contention scope is known as the Process Contention Scope, or PCS, where the scheduling of a thread is handled on the process level by the thread library, and the threads compete with other threads in the same process for CPU time.

The pthreads library allows the programmer to control the execution scheduling of the threads. The control can be performed in two different ways:

- By setting scheduling attributes when creating a thread.
- By dynamically changing the scheduling attributes of a created thread.

A thread has three scheduling parameters:

Scope The contention scope of a thread is defined by the thread model used in the threads library.

Policy The scheduling policy of a thread defines how the scheduler treats the thread once it gains control of the CPU.

Priority The scheduling priority of a thread defines the relative importance of the work being done by each thread.

The scheduling parameters can be set before the thread's creation or during the thread's execution. In general, controlling the scheduling parameters of threads is important only for threads that are compute-intensive. Thus, the threads library provides default values that are sufficient for most cases.

Controlling the scheduling of a thread is often a complicated task. Because the scheduler can handle all threads system or process-wide, depending on the scope context, the scheduling parameters of a thread can interact with those of all other threads in the process and in the other processes on the system.

On AIX, the threads library provides three scheduling policies:

FIFO First-in first-out (FIFO) scheduling. Each thread has a fixed priority; when multiple threads have the same priority level, they run to completion in FIFO order.

RR Round-robin (RR) scheduling. Each thread has a fixed priority; when multiple threads have the same priority level, they run for a fixed time slice in FIFO order.

Default Default AIX scheduling. Each thread has an initial priority that is dynamically modified by the scheduler according to the thread's activity; thread execution is time-sliced. On other systems, this scheduling policy may be different.

Normally, applications should use the default scheduling policy unless a specific application requires the use of a fixed-priority scheduling policy.

Using the RR policy ensures that all threads having the same priority level will be scheduled equally, regardless of their activity. This can be useful in programs where threads have to read sensors or write actuators.

Using the FIFO policy should be done with great care. A thread running with FIFO policy runs to completion unless it is blocked by some calls, such as performing input and output operations. A high-priority FIFO thread may not be preempted and can affect the global performance of the system. For

example, threads doing intensive calculations, such as inverting a large matrix, should never run with FIFO policy.

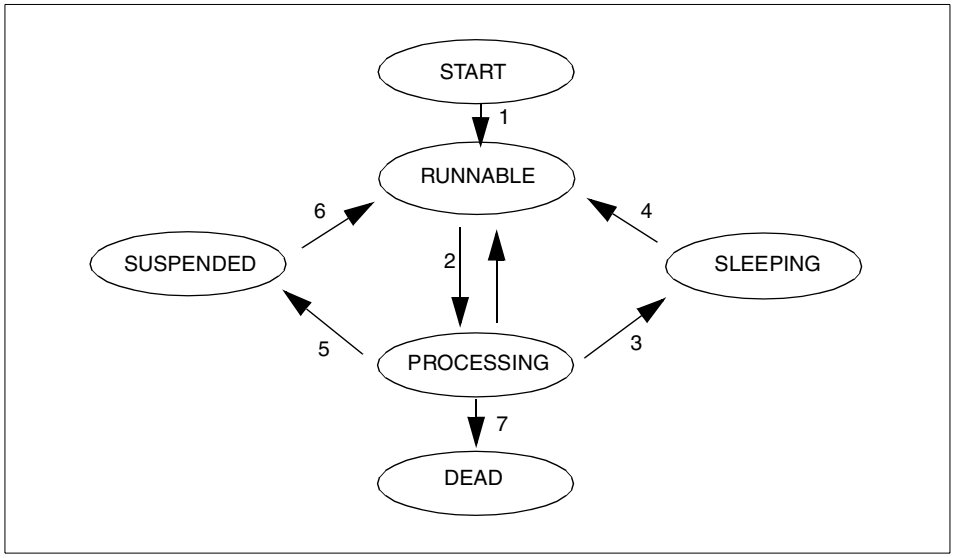


Figure 17. State transitions for a common multiplexed thread

The multiplexed threads run over a state machine as shown in Figure 17. The state transitions are described as follows:

1. At creation time, the system initializes the thread in the RUNNABLE state.
2. When it is mapped to a kernel LWP from the pool, it transitions from RUNNABLE to PROCESSING state when the kernel dispatches the LWP for execution. While in the PROCESSING state, the thread issues kernel calls and remains mapped to the LWP. In the same way, if the kernel call blocks the multiplexed thread, then the LWP will also block. In the next piece of code, the multiplexed thread and its associated LWP will block until the read request completes:

   ```
   read ( file_description, buffer, size);
   ```

3. If during the processing time the thread blocks waiting for a synchronization event, described in the next section, it goes to the SLEEPING state. In the SLEEPING state, it is no longer mapped to a LWP.
4. When a signal wakes up the thread it then transitions from SLEEPING to the RUNNABLE state again.
5. It is also possible for a thread, to transition to the SUSPENDED state and remain there until another thread from the user level resumes it.

6. At the finalization time, the thread transitions from PROCESSING to the DEAD state when it releases its resources. The system will remove the threads data on DEAD state from the process data space.

5.1.3.1 Context switch example

A context switch is not a very easy concept, and it is applied in the same sense for both processes and threads. Essentially, it means that one thread that is in PROCESSING state, which is currently bound to a LWP, must be unbounded to leave room for some other thread. It becomes much more clear when we use an example.

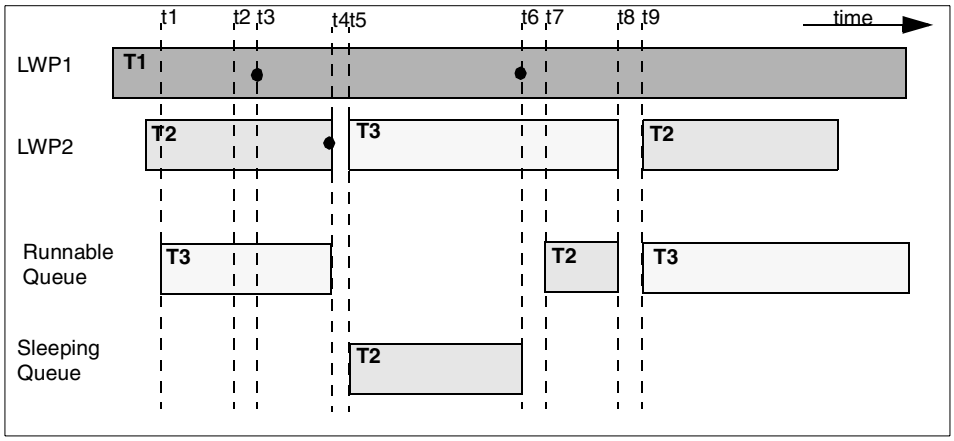


Figure 18. Context switch example

Based on the diagram in Figure 18, suppose there are three threads: T1, T2, and T3 within a single process and only two lightweight process LWP1 and LWP2. This is a typical M:N model.

At the elapsed time, t1 and t2, both threads T1 and T2 are bound respectively to LWP1 and LWP2 and in the PROCESSING state. At the same period of time, T3 is in the RUNNABLE state waiting in a queue for a LWP that can hold it. At that instant, t3, suppose that T1 acquires the rights for a synchronization primitive called L.

As time goes on both threads in the PROCESSING state execute their code flow. At the moment, t4, the thread T2 requests the rights for the L synchronization primitive, the same one that T1 owns. At this moment, the thread T2 will go to the SLEEPING state. As T2 transitions to the SLEEPING state and sleeping queue, it is no longer bound to LWP2. The thread that is topmost in the runnable queue, T3 at this point in time, is bound to LWP2. Here we have a context switch between T2 and T3. During this elapsed time

(t4-t5), the thread library takes care of a lot of local data exchanging in such a way to guarantee that all necessary information for resuming the execution of T2 in the future is safe, and it also restores all T3 information to get it running on the LWP2. We call this context switch-voluntary because T2 goes to SLEEPING state as a result of its own code.

Now, suppose that at this time,instant t6, thread T1 releases the synchronization primitive L. As part of the thread library API, this operation moves T2 from the SLEEPING state to the RUNNABLE state and also moves it to the runnable queue. Then, the thread scheduler interrupts the execution of thread T3, thus moving it to the RUNNABLE state and back to the runnable queue, positioned according its original priority. As LWP2 is unbounded, and as T2 is the topmost thread in the runnable queue, it will be rebound to LWP2 and resumes its execution. This was an involuntary context switch because T3 was just notified that it should leave the LWP2.

5.1.4 Synchronization

Synchronization is a programming method that allows multiple threads to coordinate their data accesses, therefore, avoiding the situation where one thread can change a piece of data at the same time another one is reading or writing the same piece of data. This situation is commonly called a *race condition*.

Consider, for example, a single counter, X, that is incremented by two threads, A and B. If X is originally 1, then by the time threads A and B increment the counter, X should be 3. Both threads are independent entities and have no synchronization between them. Although the C statement X++ looks simple enough to be atomic, the generated assembly code may not be as shown in the following pseudo-assembler code:

```
move    X, REG    /* put the value of X on register */
inc     REG       /* increment register             */
move    REG, X    /* store register value at      X */
```

If both threads are executed concurrently on two CPUs, or if the scheduling makes the threads alternatively execute on each instruction, the following steps may occur:

1. Thread A executes the first instruction and puts X, which is 1, into the thread A register. Then, thread B executes and puts X, which is 1, into the thread B register.
2. Next, thread A executes the second instruction and increments the content of its register to 2. Then, thread B increments its register to 2. Nothing is moved to memory X; so, memory X stays the same.

3. Last, thread A moves the content of its register, which is now 2, into memory X. Then, thread B moves the content of its register, which is also 2, into memory X, overwriting thread A's value.

Note that, in most cases, thread A and thread B will execute the three instructions one after the other, and the result would be 3, as expected. Race conditions are usually difficult to discover because they occur intermittently.

To avoid this race condition, each thread should lock the data before accessing the counter and updating memory X. For example, if thread A takes a lock and updates the counter, it leaves memory X with a value of 2. Once thread A releases the lock, thread B takes the lock and updates the counter, taking 2 as its initial value for X and incrementing it to 3, the expected result.

Basically, there are two ways for implementing how a thread can deal with the situation where it is trying to lock some data that is, in fact, already locked by another thread:

busy/wait This approach is based on the hope that the lock data will be available in a very short period of time. Basically, the thread enters a loop and continuously attempts to get the lock for the data. This model, despite its simplicity normally runs well on multiple CPU machines; otherwise, on a single CPU machine, the thread keeps occupying the CPU, and there is no chance for the other thread, that actually has the lock, to resume execution and free the locked data. This type of lock is also known as a spin lock.

sleep/wait This model is a bit more elaborate but still simple and well understandable. The idea is to put the thread in a SLEEPING state while the required lock is not available. The operating system will reactivate the thread whenever the desired locked data is ready. On SLEEPING state, the thread is not mapped to any LWP and is not consuming CPU, which gives the opportunity for the other threads to run.

We can not say that one implementation approach is better than the other. The decision is very dependent on the problem, and on the machine where it will run, and must be carefully considered at design time.

Another very important issue when dealing with multi-threaded programs is the *deadlock* condition. This is a condition where a thread, for example, locked a data and then attempts to re-lock it before unlocking. Another well known condition of deadlock is when a thread, called *th_a* for example, locks

a data called A and then attempts to lock the data B; but at the same time, another thread, *th_b*, locked the data B and then starts trying to lock the A data. It is a recursive interaction and results in an infinite deadlock.

The pthread library provides a set of synchronization primitives and its proper API. Respecting the formalism of those primitives is a very good way to avoid conflicts when using shared resources. The following section describes, in detail, the most common synchronization methods.

Note

The file descriptors in a process are shared by all of its threads. This can potentially generate data inconsistency when two or more threads are accessing the same file. In this kind of application, a lock mechanizm must be implemented, and the file pointer should be tracked by each thread.

5.1.4.1 Mutex

The mutual exclusion lock (mutex) is the simplest synchronization primitive provided by the pthread library, and many of the other synchronization primitives are built upon it.

It is based on the concept of a resource that only one person can use in a period of time, for example, a chair or a pencil. If one person sits in a chair, no one can sit on it until the first person stands-up. This kind of primitive is quite useful for creating *critical sections*. A critical section is a portion of code that must run atomically because they normally are handling resources, such as file descriptors, I/O devices, or shared data. A critical section is a portion of code delimited by the instructions that lock and unlock a mutex variable. Ensuring that all threads acting on the same resource or shared data obey this rule is a very good practice to avoid trouble when programming with threads. The following program shows a very simple example that complies with this rule:

```
#include <pthread.h>              /* include file for pthreads    */
#include <stdio.h>                /* include file for printf()    */
#define num_threads 10;           /* define the number of threads */
main()                            /* the main thread              */
{
pthread_t th[ num_threads];       /* creates an array for threads */
pthread_mutex_t mutex;            /* defines a mutex variable     */
int i;
...                               /* do other stuff               */
pthread_mutex_init(&mutex, NULL); /* creates the mutex            */
for (i = 0; i < num_thrads; i++)  /* loop to create threads       */
        pthread_create(th + i, NULL, thread_func, NULL);
```

```
...                                 /* do other stuff               */
pthread_mutex_destroy(&mutex);      /* destroys the mutex           */
}

void * thread_func( void *)         /* the request handling thread  */
{
pthread_mutex_lock(&mutex);         /* locks the mutex              */
...                                 /* do all the work              */
pthread_mutex_unlock(&mutex);       /* unlocks the mutex            */
pthread_exit( NULL);                /* finishes the thread          */
}
```

In AIX, mutexes cannot be relocked by the same thread. This may not be the case on other systems. To enhance portability of your programs, assume that the following code fragment will result in a deadlock situation:

```
pthread_mutex_lock(&mutex);
pthread_mutex_lock(&mutex);
```

This kind of deadlock may occur when locking a mutex and then calling a routine that will, itself, attempt to lock the same mutex. For example:

```
pthread_mutex_t mutex;
struct {
        int a;
        int b;
        int c;
} A;
f()
{
        pthread_mutex_lock(&mutex);      /* call 1 */
        A.a++;
        g();
        A.c = 0;
        pthread_mutex_unlock(&mutex);
}

g()
{
        pthread_mutex_lock(&mutex);      /* call 2 */
        A.b += A.a;
        pthread_mutex_unlock(&mutex);
}
```

To avoid this kind of deadlock or data inconsistency, you should use either one of the following locking schemes:

Fine granularity locking Each data atom should be protected by a mutex, locked only by low-level functions. For example, this would result in locking each record of a database. Benefits: High-level functions do not need to care about locking data. Drawbacks: It increases the number of mutexes and great care should be taken to avoid deadlocks.

High-level locking Data should be organized into areas, with each area protected by a mutex; low-level functions do not need to care about locking. For example, this would result in locking a whole database before accessing it. Benefits: There are few mutexes, and thus few risks of deadlocks. Drawbacks: Performance may be degraded, especially if many threads want access to the same data.

5.1.4.2 Condition variables

A condition variable synchronization primitive is provided through a pthreads API. Basically, it permits a thread to suspend its execution waiting for a condition or event to be satisfied by the actions of another thread. Once the condition has been met, the thread will be notified and then resume its execution.

A condition variable is also associated with a shared variable protected by a mutex. Normally, there will be:

- A boolean shared variable representing the condition state.
- A mutex to protect the shared variable.
- The condition variable itself.

The same mutex must be used for the same condition variable, even for different threads. It is possible to bundle the condition, the mutex, and the condition variable in a structure as shown in the following code fragment:

```
struct condition_bundle_t {
        int             condition_predicate;
        pthread_mutex_t  condition_lock;
        pthread_cond_t   condition_variable;
};
```

When waiting for a condition, the subroutines provided by the pthreads API atomically unlock the mutex and block the calling thread. When the condition is signaled, the mutex is relocked, and the condition wait subroutine returns.

Its possible, through a proper API, to define a period of time that the thread is blocked waiting for the condition, and it can resume either by the condition becoming TRUE or by the expiry of the time-out value.

It is important to note that when the subroutine returns without error, the condition may still be false. The reason is that more than one thread may be awoken. The first thread locking the mutex will block all other awoken threads in the condition wait subroutine until the mutex is unlocked. Thus, the predicate may have changed when the second thread gets the mutex and returns from the condition wait subroutine.

In general, whenever a condition wait returns, the thread should reevaluate the condition to determine whether it can safely proceed, should wait again, or should declare a time-out. A return from the condition wait subroutine does not imply that the predicate is either true or false.

It is recommended that a condition wait be enclosed in a while-loop that checks the predicate. The following code fragment provides a basic implementation of a condition wait:

```
pthread_mutex_lock(&condition_lock);
while (condition_predicate == 0)
        pthread_cond_wait(&condition_variable, &condition_lock);
...
pthread_mutex_unlock(&condition_lock);
```

5.1.4.3 Read/Write locks

Despite its simplicity of use, mutex locks are often very costly in some kind of applications, where shared data variables are read more often than written. For this kind of situation, a Read/Write lock, *RWlock*, can be built upon a mutex and condition variable primitive.

Basically, it permits several threads access for reading a shared resource but exclusive access for writing the resource. When a writer releases a lock, other waiting writers will get the lock before any waiting reader.

The following is a very simple implementation example for a RWlock:

```
typedef struct {
        pthread_mutex_t lock;
        pthread_cond_t rcond;
        pthread_cond_t wcond;
        int lock_count;  /* < 0 .. held by writer             */
                         /* > 0 .. held by lock_count readers */
                         /* = 0 .. held by nobody             */
        int waiting_writers;  /* count of waiting writers      */
```

```
} rwlock_t;

void rwlock_init(rwlock_t *rwl)
{
        pthread_mutex_init(&rwl->lock, NULL);
        pthread_cond_init(&rwl->wcond, NULL);
        pthread_cond_init(&rwl->rcond, NULL);
        rwl->lock_count = 0;
        rwl->waiting_writers = 0;
}

void waiting_reader_cleanup(void *arg)
{
        rwlock_t *rwl;

        rwl = (rwlock_t *)arg;
        pthread_mutex_unlock(&rwl->lock);
}

void rwlock_lock_read(rwlock_t *rwl)
{
        pthread_mutex_lock(&rwl->lock);
        pthread_cleanup_push(waiting_reader_cleanup, rwl);
        while ((rwl->lock_count < 0) && (rwl->waiting_writers))
                pthread_cond_wait(&rwl->rcond, &rwl->lock);
        rwl->lock_count++;
        /*
         *  Note that the pthread_cleanup_pop subroutine will
         *  execute the waiting_reader_cleanup routine
         */
        pthread_cleanup_pop(1);
}

void rwlock_unlock_read(rwlock_t *rwl)
{
        pthread_mutex_lock(&rwl->lock);
        rwl->lock_count--;
        if (!rwl->lock_count)
                pthread_cond_signal(&rwl->wcond);
        pthread_mutex_unlock(&rwl->lock);
}

void waiting_writer_cleanup(void *arg)
{
        rwlock_t *rwl;

        rwl = (rwlock_t *)arg;
```

```
        rwl->waiting_writers--;
        if ((!rwl->waiting_writers) && (rwl->lock_count >= 0))
                /*
                         * This only happens if we have been canceled
                         */
                        pthread_cond_broadcast(&rwl->wcond);
                        pthread_mutex_unlock(&rwl->lock);
}

void rwlock_lock_write(rwlock_t *rwl)
{
        pthread_mutex_lock(&rwl->lock);
        rwl->waiting_writers++;
        pthread_cleanup_push(waiting_writer_cleanup, rwl);
        while (rwl->lock_count)
                pthread_cond_wait(&rwl->wcond, &rwl->lock);
        rwl->lock_count = -1;
        /*
         *  Note that the pthread_cleanup_pop subroutine will
         *  execute the waiting_writer_cleanup routine
         */
        pthread_cleanup_pop(1);
}

void rwlock_unlock_write(rwlock_t *rwl)
{
        pthread_mutex_lock(&rwl->lock);
        l->lock_count = 0;
        if (!rwl->wating_writers)
                pthread_cond_broadcast(&rwl->rcond);
        else
                pthread_cond_signal(&rwl->wcond);
        pthread_mutex_unlock(&rwl->lock);
}
```

In this example, readers are just counted. When the count reaches zero, a waiting writer may take the lock. Only one writer can hold the lock. When the lock is released by a writer, another writer is awakened, if there is one. Otherwise, all waiting readers are awakened.

To make this example more useful and safe, some more features must be implemented, such as:

- Checking the lock owner to avoid a thread unlocking a lock that has been locked by another thread.
- An equivalent to the pthread_mutex_trylock subroutine.

- And, finally, some error handling.

5.1.4.4 Semaphores

The main idea behind semaphores is to control access to a set of resources, in the same way a rental car company controls its allocation system. They have a set of cars available and a kind of counter, or semaphore, that shows how many cars are ready in the parking lot. Each time a car is rented, the counter is decremented. Every returned car increases the counter. If a customer requires a car, but the counter is zero, which means no car is available, it must wait until one car becomes available.

This concept is applied on semaphores as a synchronization primitive in traditional UNIX interprocess synchronization facilities. If a semaphore is configured to hold the values 0 or 1, it is called a *binary semaphore* and works in the same way as a mutex. But, if it can reach values greater than 1, they can control a set of resources and are called *counting semaphores*. It is possible to implement interthread semaphores for specific usage.

The following implementation is very basic. Error handling is not performed, but cancellations are properly handled with cleanup handlers whenever required.

A semaphore has the sema_t data type. It must be initialized by the sema_init routine and destroyed with the sema_destroy routine. The semaphore request to lock and unlock operations are respectively performed by the sema_p and sema_v routines:

```
typedef struct {
        pthread_mutex_t lock;
        pthread_cond_t cond;
        int count;
} sema_t;
void sema_init(sema_t *sem)
{
        pthread_mutex_init(&sem->lock, NULL);
        pthread_cond_init(&sem->cond, NULL);
        sem->count = 1;
}

void sema_destroy(sema_t *sem)
{
        pthread_mutex_destroy(&sem->lock);
        pthread_cond_destroy(&sem->cond);
}

void p_operation_cleanup(void *arg)
```

```
{
        sema_t *sem;

        sem = (sema_t *)arg;
        pthread_mutex_unlock(&sem->lock);
}

void sema_p(sema_t *sem)
{
        pthread_mutex_lock(&sem->lock);
        pthread_cleanup_push(p_operation_cleanup, sem);
        while (sem->count <= 0)
                pthread_cond_wait(&sem->cond, &sem->lock);
        sem->count--;
        /*
         *  Note that the pthread_cleanup_pop subroutine will
         *  execute the p_operation_cleanup routine
         */
        pthread_cleanup_pop(1);
}

void sema_v(sema_t *sem)
{
        pthread_mutex_lock(&sem->lock);
        sem->count++;
        if (sem->count <= 0)
                pthread_cond_signal(&sem->cond);
        pthread_mutex_unlock(&sem->lock);
}
```

The counter specifies the number of users that are allowed to take the semaphore. It is never strictly negative; thus, it does not specify the number of waiting users.

5.1.4.5 Joining threads

The joining process is invoked by a thread called the *requester*, and it is addressed to a *target* thread. It simply means that the requester thread blocks until the target one finishes its execution.

This objective is normally best accomplished by the other synchronization primitives, such as mutex and semaphores. The real need for this kind of mechanics is when the requester provided the target thread's stack and must release it after the execution.

It is very important to notice that threads created with the *undetached* attribute must be joined; otherwise, their storage will be kept until the end of

the whole process. Threads created with the *detached* attribute automatically tidy up upon exit, returning their storage to the process heap. Trying to join a detached thread will fail.

5.1.5 Signals and threads

The signal mechanics is part of UNIX systems. It provides a way to handle asynchronous events. Basically, when a process receives a signal, the kernel stops it. Then, a piece of code defined as a *handler* is executed, and after its completion the process resumes at the exact point where it was before being stopped by the kernel. Each handler is assigned to a specific signal through the *signal handler* table. Another table called the *signal mask* defines, which signals the process, it will receive and which it will ignore. Every time a process receives a signal, the kernel checks out its signal mask to determine if it is allowed and then looks in the signal handler table to execute the proper handler.

This powerful feature allows three situations to be dealt with:

Error For example, a division by zero or trying to reach an invalid memory address. Normally, error situations halts the process.

Notification To notify that some situation has changed.

Interruption To force a handler to be executed immediately, despite what code is running.

Basically, we can say that all handlers are in the process scope. This means every thread can assign a handler for a signal, and the latest one overwrites the previous one. But, pthreads address a per thread signal mask that permits control over which signal each thread should receive.

In a single thread model, there is no distinction between these situations. On the other hand, when using multi-thread programming, these three situations are handled in different ways:

Error The signal is delivered to the thread that caused the problem.

Notification The signal is delivered to the process and must be handled on this level.

Interruption The signal is delivered to the process, and there is no way to guarantee that one specific thread will receive the signal, even if it requested it.

Typically, programs may create a dedicated thread to wait for asynchronously generated signals. Such a thread just loops on a sigwait subroutine call and

handles the signals. The following code fragment gives an example of such a signal waiter thread:

```
sigset_t set;
int sig;

sigemptyset(&set);
sigaddset(&set, SIGINT);
sigaddset(&set, SIGQUIT);
sigaddset(&set, SIGTERM);
sigthreadmask(SIG_BLOCK, &set, NULL);

while (1) {
        sigwait(&set, &sig);
        switch (sig) {
                case SIGINT:
                        /* handle interrupts */
                        break;
                case SIGQUIT:
                        /* handle quit */
                        break;
                case SIGTERM:
                        /* handle termination */
                        break;
                default:
                        /* unexpected signal */
                        pthread_exit((void *)-1);
        }
}
```

If more than one thread called the sigwait subroutine, exactly one call returns when a matching signal is sent. There is no way to predict which thread will be awakened.

5.1.6 Software models

In the multi-threaded environment, we have two concepts that tend to be confused:

Concurrency We can say that tasks are concurrent when they can be executed in any order, or maybe at the same time. This is exactly what we have in a multi-threaded process. Each thread can reach the CPU and execute its code. There is no control or a predefined order.

Parallelism Is applied in multiple CPU environments where more than one task is running simultaneously. So, we can conclude

that all parallel programs are concurrent, but not all concurrent programs are parallel.

As a rule, all concerns about synchronizations are about the concurrency, and normally, the parallelism is not an issue for the implementation point of view. In practice, we can say that well designed code, from a concurrency point of view, will run well, or even faster, on a parallel machine.

The next section describes the advantages of using multi-threaded applications in concurrent or parallel environments to address common software models that are well suited to parallel programming techniques.

5.1.6.1 Master/Slave

In the master/slave (sometimes called boss/worker) model, a master entity receives one or more requests, then creates slave entities to execute them. Typically, the master controls how many slaves there are and what each slave does. A slave runs independently of other slaves.

An example of this model is a print job spooler controlling a set of printers. The spooler's role is to ensure that the print requests received are handled in a timely fashion. When the spooler receives a request, the master entity chooses a printer and causes a slave to print the job on the printer. Each slave prints one job at a time on a printer, handling flow control and other printing details. The spooler may support job cancellation or other features that require the master to cancel slave entities or reassign jobs. Figure 19 on page 129 illustrates this model.

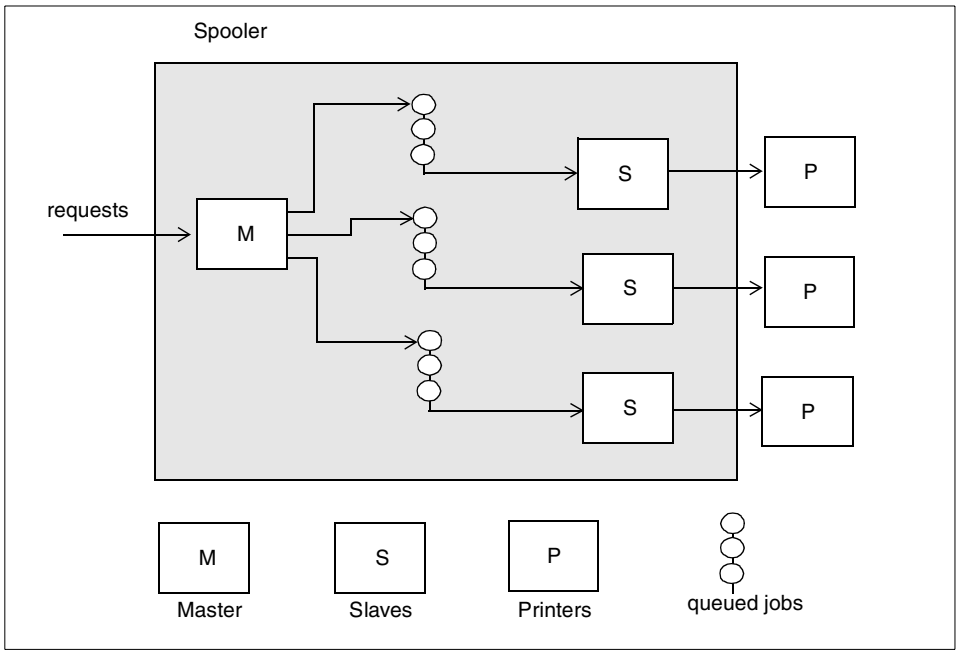


Figure 19. Master/slave print model

5.1.6.2 Divide/Conquer

In the divide-and-conquer (sometimes called simultaneous computation or work crew) model, one or more entities perform the same tasks in parallel. There is no master entity; all entities run in parallel, independently.

An example of a divide-and-conquer model is a parallelized `grep` command implementation, which could be done as follows. The `grep` command first establishes a pool of files to be scanned. It then creates a number of entities. Each entity takes a different file from the pool and searches for the pattern, sending the results to a common output device. When an entity completes its file search, it obtains another file from the pool or stops if the pool is empty.

5.1.6.3 Producer/consumer

The producer/consumer (sometimes called pipelining) model is typified by a production line. An item proceeds from raw components to a final item in a series of stages. Usually, a single worker at each stage modifies the item and passes it on to the next stage. In software terms, an AIX command pipe, such as the `cpio` command, is a good example of a this model.

Figure 20 illustrates a typical producer/consumer model. In this example, the reader entity reads raw data from standard input and passes it to the processor entity, which processes the data and passes it to the writer entity, which writes it to standard output. Parallel programming allows the activities to be performed concurrently: The writer entity may output some processed data while the reader entity gets more raw data.

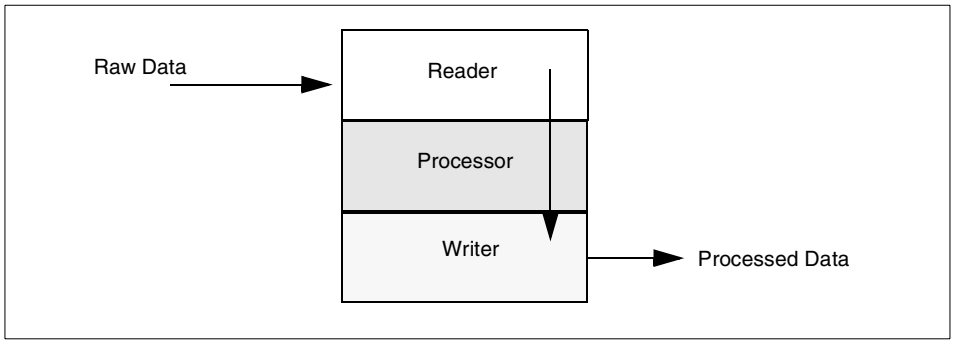


Figure 20. Producer/consumer model

5.1.7 Performance considerations

Multi-threaded applications can improve the application performance when compared to traditional parallel programs. The three most impacting issues between threads and process are listed as follows:

Managing

Managing threads, that is, creating threads and controlling their execution, requires fewer system resources than managing processes. Creating a thread, for example, only requires the allocation of the thread's private data area, usually 64 KB, and two system calls. Creating a process is far more expensive because the entire parent process address space is duplicated.

A context switch between threads in the same process is also much simpler and faster than for a processes context switch between processes.

Communication

Inter-thread communication is far more efficient and easier to use than inter-process communication. Because all threads within a process share the same address space, they do not need to use shared memory. Shared data should be protected from concurrent access using mutexes and other synchronization tools.

Synchronization facilities provided by the threads library allow easy implementation of flexible and powerful synchronization tools. These tools can easily replace traditional inter-process communication facilities, such as message queues.

Multiprocessor

On a multiprocessor system, multiple threads can concurrently run on multiple CPUs. Therefore, multi-threaded programs can run much faster than on a uniprocessor system. They will also be faster than a program using multiple processes because threads require fewer resources and generate less overhead. For example, switching threads in the same process can be faster, especially in the M:N library model where context switches can often be avoided. Finally, a major advantage of using threads is that a single multi-threaded program will work on a uniprocessor system but can naturally take advantage of a multiprocessor system without recompiling.

5.1.7.1 A trade-off

Despite the thread structure and its handling being much lighter when compared to processes, a trade-off consideration must be taken to avoid wasting processing time only by creating and handling unnecessary threads and shared data. There are three basic points to consider, which are as follows:

Overhead of creating/terminating threads

If the application creates and terminates several threads intensively, the overhead for creating, and then later terminating, each thread can became very expensive and degrade performance. It is very important to consider this overhead when designing the application. A good implementation for this situation is creating a pool of threads and reuse them several times. This approach can reduce the creation/termination overhead.

Overhead on context Switch

Normally, multi-threaded programs have more threads than there are CPUs in the system, which can cause constant thread context switching. Again, despite a thread context switch being much lighter than a process one, they consume CPU time and must be avoided whenever possible or at least minimized. Keep in mind that every time a thread blocks waiting for a lock or an I/O operation, it generates a voluntary context switch.

Overhead on data sharing

The multi-thread application must use locks (mutex, condition variables, semaphores, and so on) basically for safety while sharing data and for waiting for some task to be accomplished by another thread. The use of locks impacts the overall application performance in a low level when performing the lock and unlock operations and in a much more considerable level when waiting for locks that are already locked by another thread.

Note

As a rule of thumb, we can say:

1. Use mutex or semaphores only to synchronize access to shared data.
2. Use condition variables to synchronize threads against events, for example, when one thread must wait until another one finishes.

5.2 Implementing threaded applications on AIX

The newest version of AIX, Version 4.3 implements a full compliance to the IEEE POSIX standard for threads APIs, IEEE POSIX 1003.1-1996. Table 11 shows a summary of the evolution of thread support in AIX.

Table 11. AIX POSIX thread conformance

AIX version and release	Threads Version	Thread model
3.2	POSIX Draft 4	M:1
4.1 and 4.2	POSIX Drafts 7 and 4	1:1
4.3	UNIX98, POSIX Drafts 10, 7 and 4	M:N

Basically, we can say that AIX, Version 4, Release 3 provides full support for applications compiled on AIX, Version 3, Release 2 and AIX, Version 4. It also provides compilation support for applications written to the Draft 7 level that are not able to modify their source code to full standards conformance.

5.2.1 Compiling and linking

In AIX, Version 4.3 compiling and linking a multi-threaded application is as simple as compiling a non-threaded application.

Note

Take special attention when code is being ported from another platform or another POSIX Draft.

Table 12 shows all important information about compiler mode, specifically the compiler driver program to use depending on the required pthreads standard.

Table 12. AIX 4.3 C driver programs

C Driver program	Description

xlc_r
cc_r
xlC_r
	All _r-suffixed invocations are functionally similar to their corresponding base compiler invocation, but set the macro name -D_THREAD_SAFE and invoke the added compiler options:
	`-L/usr/lib/threads`
	`-L/usr/lib/dce`
	`-lpthreads`
	`-qthreaded`
	Use the *_r*-suffixed invocations when compiling with the -qsmp compiler option or if you want to create POSIX threaded applications.

xlc_r4 cc_r4 xlC_r4	Use *_r4*-suffixed invocations to provide compatibility between DCE applications written for AIX, Version 3.2.5 and AIX, Version 4. They link your application to the correct AIX, Version 4 DCE libraries, providing compatibility between the latest version of the pthreads library and the earlier versions supported on AIX, Version 3.2.5.
xlc_r7 cc_r7 xlC_r7	Use the *_r7*-suffixed invocations to compile and link applications conforming POSIX “Draft 7” standard. Otherwise, the compiler will, by default, compile and link applications conforming to the current POSIX threads standards.

The *xlc* compiles C source codes with a default language level as *ANSI*, and *cc* compiles C sources with default language level as *extended*. The extended level is suitable for code that does not require full compliance with the ANSI C standard, for example, legacy code. The *xlC* driver is for C++ code.

Note

For multi-threaded applications, you must use one of the _r-suffixed C driver programs.

In the following example, a very simple Makefile is suggested to compile and link a multi-threaded C program called ex1.c:

```
CC              = /usr/vac/bin/xlc_r
CFLAGS          = -g
BIN             = ex1
all:            $(BIN)
clean:
                rm -f $(BIN)
                rm -f *.o
```

Notice that the default path for the C compiler is /usr/vac/bin.

5.2.2 Thread model and tuning

AIX, Version 4.3.1 replaced the previous 1:1 threads implementation model with an M:N version. The M:N model complies with the UNIX 98 pthreads standard, which includes the POSIX pthreads standard. Previous releases of AIX, Version 4 complied with Draft 7 of the POSIX pthreads standard. AIX, Version 4.3.1 is binary compatible with previous releases. The UNIX 98 implementation is the default for application development, but you can use specific compiler drivers, as shown in Table 12 on page 133, to develop new applications using Draft 7 pthreads.

5.2.2.1 Tuning

The M:N pthreads implementation provides several environment variables that can be used to affect application performance. If possible, the application developer should provide a front-end shell script to invoke the binary executable in which the user may specify new values to override the system defaults. The following environment variables can be set by end users and are examined at process initialization time:

AIXTHREAD_SCOPE

This variable can be used to set the contention scope of pthreads created using the default pthread attribute object. It is represented by the following syntax:

```
AIXTHREAD_SCOPE=[P|S]
```

The value `P` indicates process scope, while a value of `S` indicates system scope. If no value is specified, the default pthread attribute

object will use process scope contention, which implies the M:N model.

SPINLOOPTIME

This variable controls the number of times the system will try to get a busy lock without taking a secondary action, such as calling the kernel to yield the processor. This control is really intended for SMP systems, where it is hoped that the lock is held by another actively running pthread and will soon be released. On uniprocessor systems, this value is ignored.

YIELDLOOPTIME

This variable controls the number of times that the system yields the processor when trying to acquire a busy mutex or spin lock before actually going to sleep on the lock. This variable has been shown to be effective in complex applications where multiple locks are in use.

The following environment variables impact the scheduling of pthreads created with process-based contention scope:

AIXTHREAD_MNRATIO

This variable allows the user to specify the ratio of pthreads to kernel threads. It is examined when creating a pthread to determine if a kernel thread should also be created to maintain the correct ratio. It is represented with the following syntax:

```
AIXTHREAD_MNRATIO=p:k
```

where `k` is the number of kernel threads to use to handle `p` pthreads. Any positive integer value may be specified for p and k. These values are used in a formula that employs integer arithmetic, which can result in the loss of some precision when big numbers are specified. If k is greater than p, the ratio is treated as 1:1. If no value is specified, the default ratio depends on the default contention scope. If system scope contention is the default, the ratio is 1:1. If process scope contention is set as the default, the ratio is 8:1.

Note

When migrating threaded applications to AIX from other platforms or previous versions of AIX, the default 8:1 ratio used with the M:N threads model may reduce application performance.
If this is the case, you can either change the source code of the application so that threads are created with the contention scope attribute set to PTHREAD_SCOPE_SYSTEM, set the AIXTHREAD_SCOPE environment variable to the value S, or change the ratio of kernel threads to user threads with the AIXTHREAD_MNRATIO environment variable.

AIXTHREAD_SLPRATIO

This variable is used to determine the number of kernel threads used to support local pthreads sleeping in the library code on a pthread event, for example, attempting to obtain a mutex. It is represented by the following syntax:

```
AIXTHREAD_SLPRATIO=k:p
```

where `k` is the number of kernel threads to reserve for every `p` sleeping pthreads. Notice that the relative positions of the numbers indicating kernel threads and user pthreads are reversed when compared with AIXTHREAD_MNRATIO. Any positive integer value may be specified for p and k. These values are used in a formula that employs integer arithmetic, which can result in the loss of some precision when large numbers are specified. If k is greater than p, the ratio is treated as 1:1. If the variable is not set, a ratio of 1:12 is used. The reason for maintaining kernel threads for sleeping pthreads is that, when the pthread event occurs, the pthread will immediately require a kernel thread to run on. It is more efficient to use a kernel thread that is already available than it is to create a new kernel thread once the event has taken place.

AIXTHREAD_MINKTHREADS

This variable is a manual override to the AIXTHREAD_MNRATIO. It allows you to stipulate the minimum number of active kernel threads. The library scheduler will not reclaim kernel threads below this number.

5.2.3 Pthread creation and handling

This section describes the process for creating and working with a pthread on AIX.

5.2.3.1 A brief description of pthread_attr_t

The attributes of a thread are stored in an opaque object used when creating the thread. The `pthread_attr_t` is a pointer to a structure that must be initialized before being used. The basic code for defining, initializing, and setting values of attributes in the pthread_attr_t object is as follows:

```
#include <pthread.h>       /* must be the first #include file */
...
pthread_attr_t attr;       /* defines a variable somewhere */
                           /* in the code, globally or     */
                           /* locally.                     */
...
pthread_attr_init(&attr);  /* creates and initializes with */
                           /* default variables            */
.....                      /* used for setting non-default values */
pthread_attr_destroy( &attr);/* releases the variable        */
```

Once created, a pthread_attr_t variable contains several attributes, depending on the implementation of POSIX options. In AIX, the pthread_attr_t data type is a pointer to a structure; on other systems, it may be implemented as a structure or another data type. It contains attributes as shown in Table 13.

Table 13. Attributes of pthread_attr_t type

Attribute	Default value
Detachstate	PTHREAD_CREATE_JOINABLE is set as default.
Contention-scope	PTHREAD_SCOPE_PROCESS the default ensures compatibility with implementations that do not support this POSIX option.
Inheritsched	PTHREAD_INHERITSCHED
Schedparam	A sched_param structure which sched_prio field is set to 1, the least favored priority.
Schedpolicy	SCHED_OTHER
Stacksize	PTHREAD_STACK_MIN

Attribute	Default value
Guardsize	PAGESIZE

The resulting attribute object (possibly modified by setting individual attribute values), when used by pthread_create, defines the attributes of the thread created. A single attributes object can be used in multiple simultaneous calls to pthread_create.

For setting values other than the defaults on the pthread_attr_t variable, use the functions detailed in Table 14.

Table 14. Functions for setting pthread_attr_t attributes

Function	Description
pthread_attr_setdetachstate	Sets the detachstate attribute in the attr object.
pthread_attr_setstacksize	Set the value of the stacksize attribute of the thread attributes object attr. This attribute specifies the minimum stack size, in bytes, of a thread created with this attributes object.
pthread_attr_setschedparam	Set the value of the schedparam attribute of the thread attributes object attr. The schedparam attribute specifies the scheduling parameters of a thread created with this attributes object. The sched_priority field of the sched_param structure contains the priority of the thread.
pthread_attr_setstackaddr	Set the value of the stackaddr attribute of the thread attributes object attr. This attribute specifies the stack address of a thread created with this attributes object.

5.2.3.2 A brief description of pthread_create

The creation of a new thread is performed using the *pthread_create* subroutine. This function creates a new thread and makes it runnable. It is defined as:

```
int pthread_create (pthread_t *thread, const pthread_attr_t *attr, void
*(*start_routine) (void), void *arg) ;
```

where the arguments are:

thread A pointer to the new thread's ID variable.

attr Points to a pthread_attr_t variable properly declared and initialized.

start_routine A pointer to the routine that will be executed by the new thread.

arg A pointer to arguments that will be passed to the new thread.

When calling the pthread_create subroutine, you may specify a thread attributes object. If you specify a NULL pointer, the created thread will have the default attributes. Thus, the code fragment:

```
pthread_t thread;
pthread_attr_t attr;
...
pthread_attr_init(&attr);
pthread_create(&thread, &attr, start_routine, NULL);
pthread_attr_destroy(&attr);
```

is equivalent to:

```
pthread_t thread;
...
pthread_create(&thread, NULL, start_routine, NULL);
```

When calling the pthread_create subroutine, you must specify an *entry-point routine*. This routine, provided by your program, is similar to the main routine for a process. It is the user routine executed by the new thread. When the thread returns from this routine, the thread is automatically terminated.

The entry-point routine has one parameter, a void pointer, specified when calling the pthread_create subroutine. You may specify a pointer to some data, such as a string or a structure. The creating thread (the one calling the pthread_create subroutine) and the created thread must agree upon the actual type of this pointer. The entry-point routine returns a void pointer. After the thread termination, this pointer is stored by the threads library unless the thread is detached.

The thread ID of a newly created thread is returned to the creating thread through the thread parameter. A thread ID is an opaque object; its type is pthread_t. In AIX, the pthread_t data type is an integer. On other systems, it may be a structure, a pointer, or any other data type.The caller can use this thread ID to perform various operations on the thread.

Depending on the scheduling parameters, the new thread may start running before the call to the pthread_create subroutine returns. It may even happen

that, when the pthread_create subroutine returns, the new thread has already terminated. The thread ID returned by the pthread_create subroutine through the thread parameter is then already invalid. It is, therefore, important to check for the ESRCH error code returned by threads library subroutines using a thread ID as a parameter.

If the pthread_create subroutine is unsuccessful, no new thread is created, the thread ID in the thread parameter is invalid, and the appropriate error code is returned.

Note

To enhance the portability of programs using the threads library, the thread ID should always be handled as an opaque object. For this reason, thread IDs should be compared using the *pthread_equal* subroutine. Never use the C equality operator (==) because the pthread_t data type may be neither an arithmetic type nor a pointer.

5.2.3.3 Thread specific data

As all threads share the same process address space, they also share the same data space. Thread Specific Data - *TSD* is a POSIX functionality that permits creation of per-thread data. This allows multiple threads to run the same code and access thread-specific data using the same variable names. This makes the design of the code easier since it does not need to be aware of which thread is running.

Thread-specific data may be viewed as a two-dimensional array of values, with keys serving as the row index and thread IDs as the column index as shown in Figure 21 on page 141. A thread-specific data key is an opaque object, of type *pthread_key_t*. The same key can be used by all threads in a process. Although all threads use the same key, they set and access different thread-specific data values associated with that key. Thread-specific data are void pointers. This allows referencing any kind of data, such as dynamically allocated strings or structures.

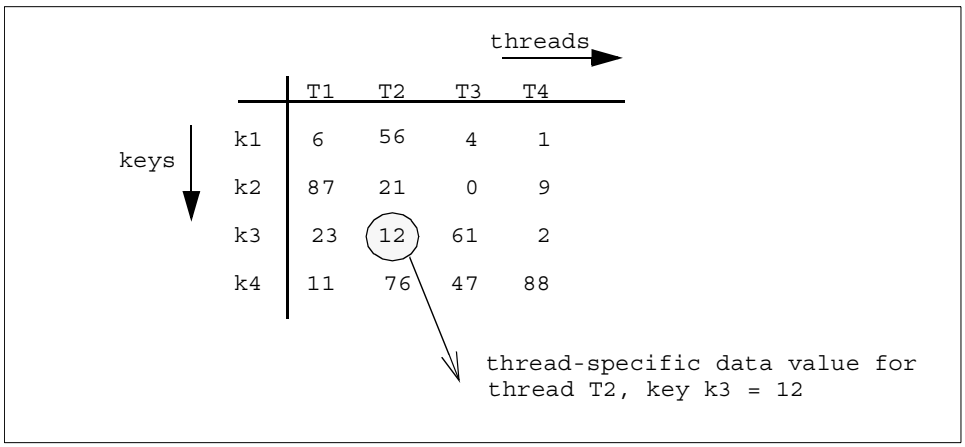


Figure 21. Thread-specific data array

The *pthread_key_create* subroutine creates a thread-specific data key. The key is shared among all threads within the process, but each thread has its own data associated with the key. The thread-specific data is initially set to NULL.

The application is responsible for ensuring that this subroutine is called only once for each requested key. This can be done, for example, by calling the subroutine before creating other threads or by using the one-time initialization facility.

At the key creation time, an optional destructor routine can be specified. If the key specific value is not NULL, that destructor will be called for each thread terminated and detached. Typically, the destructor routine will release the storage thread-specific data. It will receive the thread-specific data as a parameter.

For example, a thread-specific data key may be used for dynamically allocated buffers. A destructor routine should be provided to ensure that the buffer is freed when the thread terminates. The `free` subroutine can be used:

```
pthread_key_create(&key, free);
```

More complex destructors may be used as shown in the following:

```
typedef struct {
        FILE *stream;
        char *buffer;
} data_t;
...
```

```
void destructor(void *data)
{
        fclose(((data_t *)data)->stream);
        free(((data_t *)data)->buffer);
        free(data);
        *data = NULL;
}
```

Note

Although some implementations of the threads library may repeat destructor calls, the destructor routine is called only once in AIX.

Take care when porting code from other systems where a destructor routine can be called several times.

Thread-specific data is accessed using the *pthread_getspecific* and *pthread_setspecific* subroutines. The first one reads the value bound to the specified key and specific to the calling thread; the second one sets the value as shown in the following code example:

```
pthread_key_create(&key, free);
...
private_data = malloc(...);
pthread_setspecific(key, private_data);
...
pthread_getspecific(key, &data);
...
```

5.2.3.4 Cancelling threads

The thread cancellation mechanism allows a thread to terminate the execution of any other thread in the process in a controlled manner. The target thread (that is, the one that's being canceled) can hold cancellation requests pending in a number of ways and perform application-specific cleanup processing when the notice of cancellation is acted upon. When canceled, the thread implicitly calls the pthread_exit((void *)-1) subroutine.

The cancellation of a thread is requested by calling the *pthread_cancel* subroutine. When the call returns, the request has been registered, but the thread may still be running.

The cancelability state and type of a thread determines the action taken upon receipt of a cancellation request:

Disabled cancelability

Any cancellation request is set pending, until the cancelability state is changed or the thread is terminated in another way. A thread should disable cancelability only when performing operations that cannot be interrupted. For example, if a thread is performing some complex file save operations (such as an indexed database) and is canceled during the operation, the files may be left in an inconsistent state. To avoid this, the thread should disable cancelability during the file save operations.

Deferred cancelability

Any cancellation request is set pending until the thread reaches the next cancellation point. This is the *default* cancelability state. This cancelability state ensures that a thread can be cancelled, but limits the cancellation to specific moments in the thread's execution, called *cancellation points*. A thread canceled on a cancellation point leaves the system in a safe state; however, user data may be inconsistent or locks may be held by the canceled thread. To avoid these situations, you may use cleanup handlers or disable cancelability within critical regions.

Asynchronous cancelability

Any cancellation request is acted upon immediately. A thread that is asynchronously canceled while holding resources may leave the process, or even the system, in a state from which it is difficult or impossible to recover.

Cancellation points are points inside of certain subroutines where a thread must act on any pending cancellation request if deferred cancelability is enabled. An explicit cancellation point can also be created by calling the *pthread_testcancel* subroutine. This subroutine simply creates a cancellation point. If deferred cancelability is enabled, and if a cancellation request is pending, the request is acted upon, and the thread is terminated. Otherwise, the subroutine simply returns.

Other cancellation points occur when calling the following subroutines:

- pthread_cond_wait
- pthread_cond_timedwait
- pthread_join

The following code shows a thread where the cancelability is disabled for a set of instructions and then restored using *pthread_setcancelstate* function:

```
void *Thread(void *string)
```

```
{
        int i;
        int o_state;

        /* disables cancelability */
        pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &o_state);

        /* writes five messages */
        for (i=0; i<5; i++)
                printf("%s\n", (char *)string);

        /* restores cancelability */
        pthread_setcancelstate(o_state, &o_state);

        /* writes further */
        while (1)
                printf("%s\n", (char *)string);
        pthread_exit(NULL);
}
```

Table 15 lists the set of functions that contain cancellation points.

Table 15. Cancellation point functions

aio_suspend	close	creat
fcntl	fsync	getmsg
getpmsg	lockf	mq_receive
mq_send	msgrcv	msgsnd
msync	nanosleep	open
pause	poll	pread
pthread_cond_timedwait	pthread_cond_wait	pthread_join
pthread_testcancel	putpmsg	pwrite
read	readv	select
sem_wait	sigpause	sigsuspend
sigtimedwait	sigwait	sigwaitinfo
sleep	system	tcdrain
usleep	wait	wait3
waitid	waitpid	write

writev		

Table 16 shows functions where cancellation points may occur.

Table 16. Functions where cancellation can occur

catclose	catgets	catopen
closedir	closelog	ctermid
dbm_close	dbm_delete	dbm_fetch
dbm_nextkey	dbm_open	dbm_store
dlclose	dlopen	endgrent
endpwent	fwprintf	fwrite
fwscanf	getc	getc_unlocked
getchar	getchar_unlocked	getcwd
getdate	getgrent	getgrgid
getgrgid_r	getgrnam	getgrnam_r
getlogin	getlogin_r	popen
printf	putc	putc_unlocked
putchar	putchar_unlocked	puts
pututxline	putw	putwc
putwchar	readdir	readdir_r
remove	rename	rewind
endutxent	fclose	fcntl
fflush	fgetc	fgetpos
fgets	fgetwc	fgetws
fopen	fprintf	fputc
fputs	getpwent	getpwnam
getpwnam_r	getpwuid	getpwuid_r
gets	getutxent	getutxid
getutxline	getw	getwc

getwchar	getwd	rewinddir
scanf	seekdir	semop
setgrent	setpwent	setutxent
strerror	syslog	tmpfile
tmpnam	ttyname	ttyname_r
fputwc	fputws	fread
freopen	fscanf	fseek
fseeko	fsetpos	ftell
ftello	ftw	glob
iconv_close	iconv_open	ioctl
lseek	mkstemp	nftw
opendir	openlog	pclose
perror	ungetc	ungetwc
unlink	vfprintf	vfwprintf
vprintf	vwprintf	wprintf
wscanf		

5.3 Examples

The first multi-threaded program discussed here is very simple. It displays "Hello!" in both English and French for five seconds:

```
#include <pthread.h>    /* include file for pthreads - must be 1st */
#include <stdio.h>      /* include file for printf()           */
#include <unistd.h>     /* include file for sleep()            */
void *Thread(void *string)
{
        while (1)
                printf("%s\n", (char *)string);
        pthread_exit(NULL);
}
int main()
{
        char *e_str = "Hello!";
        char *f_str = "Bonjour !";
```

```
        pthread_t e_th;
        pthread_t f_th;

        int rc;

        rc = pthread_create(&e_th, NULL, Thread, (void *)e_str);
        if (rc)
                exit(-1);
        rc = pthread_create(&f_th, NULL, Thread, (void *)f_str);
        if (rc)
                exit(-1);
        sleep(5);

exit(0);
}
```

Note

In all programs using pthreads, the first non-comment or whitespace line in the code must include the pthread.h header file since it contains macro definitions used in other system header files.

The main thread in the example program shown above creates two threads and waits for five seconds and then exits. Both threads have the same entry-point routine (the *Thread* routine) but a different parameter. The parameter is a pointer to the string that will be displayed.

Note

Calling the *exit* subroutine terminates the entire process, including all its threads. In a multi-threaded program, the exit subroutine should only be used when the entire process needs to be terminated, for example, in the case of an unrecoverable error.

The *pthread_exit* subroutine terminates only the calling thread. It frees any thread-specific data, including the thread's stack.

Now, let‘s take a look at a little more complex and useful example. A matrix multiplication operation is always a bottleneck on programs due to its loop iterations. For a bigger matrix, the computational time can became very long

if processed in a single flow of operation. In the following example, we will show a very simple multi-threaded program for matrix multiplication:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define     SIZE        5

/* creating the variables */
double      A[SIZE][SIZE],
            B[SIZE][SIZE],
            C[SIZE][SIZE];

pthread_t id[SIZE][SIZE];

struct _element {
                  long i,
                       j;
                } element [SIZE][SIZE];

/* This reentrant program computes C[i][j] = A[i][?]XB[?][j] */
void * xprod_thread( void * arg)
{
register double s;
struct   _element *p;
long     i, j, k;
p =  (struct _element *)arg;
s = 0;
i = p->i;
j = p->j;
for (k = 0; k < SIZE; ++k)
     s += A[i][k] * B[k][j];
C[i][j] = s;
pthread_exit( NULL);
}
/* initialize matrix A, B */
void initMat( void)
{
int      i, j;

  for( i = 0; i < SIZE; ++i)
    for( j = 0; j < SIZE; ++j) {
       A[i][j] = i + j;
       B[i][j] = i - j;
    }
}
```

```
/* print the matrix */
void showMat( double m[SIZE][SIZE], int lin, int col)
{
long   i, j;
    for( i = 0; i < lin; ++i)
    {
       printf("\n\t| ");
       for( j = 0; j < col; ++j)
          printf("%5.2lf ", m[i][j]);
       printf("|");
    }
    printf("\n");
}
main(int argc, char * argv[])
{
long     i, j;
/* Initialize the Matrixes */
   initMat();
/* Matrix product */
   for (i = 0; i < SIZE; ++i)
    for (j = 0; j < SIZE; ++j)
    {
       element[i][j].i = i;
       element[i][j].j = j;
       pthread_create ( &id[i][j], NULL, xprod_thread,
                        (void *)&element[i][j]);
     }
/* Let us wait for the computation to be finished */
  for (i = 0; i < SIZE; ++i)
   for (j = 0; j < SIZE; ++j)
   {
       pthread_join (id[i][j], NULL);
   }
/* The computation is over */
/* Lets show the result    */
   printf("\n----- Matrix A ------");
   showMat( A, SIZE, SIZE);
   printf("\n----- Matrix B ------");
   showMat( B, SIZE, SIZE);
   printf("\n----- Matrix C ------");
   showMat( C, SIZE, SIZE);
}
```

There are several other optimized algorithms, but here we are just focused on one multi-threaded implementation. Performance considerations are addressed in Chapter 6, “Making our programs run faster” on page 163.

Here, you can see a sequence of pthreads created. They execute a multiplication line per column on a simple reentrant and thread-safe function xprod_thread. The number of threads created depends on the size of the matrix (#define SIZE 5).

5.3.1 Supported POSIX API

There are very few differences between POSIX Draft 7 and the final standard:

- There are some minor errno differences. The most prevalent is the use of ERSCH to denote the specified pthread could not be found. Draft 7 frequently returns EINVAL for this failure.
- Pthreads are joinable be default. This is a significant change since it can result in a memory leak if ignored.
- Pthreads have process scheduling scope by default.
- The various scheduling policies associated with the mutex locks are slightly different.
- The pthread_yield subroutine has been replaced by sched_yield.

Note

By default, pthreads are joinable, which means their memory space is not released after it terminates. You must join them, using `pthread_join`, or set the detachstate attribute to PTHREAD_CREATE_DETACHED before its creation. Otherwise, you may run out of storage space when creating new threads because each thread takes up an amount of memory.

Table 17 lists all POSIX threads interfaces supported on AIX, Version 4.3.

Table 17. POSIX thread API functions supported on AIX 4.3

pthread_atfork	pthread_attr_destroy
pthread_attr_getdetachstate	pthread_attr_getschedparam
pthread_attr_getstackaddr	pthread_attr_getstacksize
pthread_attr_init	pthread_attr_setdetachstate
pthread_attr_setschedparam	pthread_attr_setstackaddr
pthread_attr_setstacksize	pthread_cancel
pthread_cleanup_pop	pthread_cleanup_push

pthread_detach	pthread_equal
pthread_exit	pthread_getspecific
pthread_join	pthread_key_create
pthread_key_delete	pthread_kill
pthread_mutex_destroy	pthread_mutex_init
pthread_mutex_lock	pthread_mutex_trylock
pthread_mutex_unlock	pthread_mutexattr_destroy
pthread_mutexattr_getpshared	pthread_mutexattr_init
pthread_mutexattr_setpshared	pthread_once
pthread_self	pthread_setcancelstate
pthread_setcanceltype	pthread_setspecific
pthread_sigmask	pthread_testcancel
sigwait	pthread_cond_broadcast
pthread_cond_destroy	pthread_cond_init
pthread_cond_signal	pthread_cond_timedwait
pthread_cond_wait	pthread_condattr_destroy
pthread_condattr_getpshared	pthread_condattr_init
pthread_condattr_setpshared	pthread_create
pthread_attr_getguardsize	pthread_attr_setguardsize
pthread_getconcurrency	pthread_mutexattr_gettype
pthread_mutexattr_settype	pthread_rwlock_destroy
pthread_rwlock_init	pthread_rwlock_rdlock
pthread_rwlock_tryrdlock	pthread_rwlock_trywrlock
pthread_rwlock_unlock	pthread_rwlock_wrlock
pthread_rwlockattr_destroy	pthread_rwlockattr_getpshared
pthread_rwlockattr_init	pthread_rwlockattr_setpshared
pthread_setconcurrency	

AIX, Version 4.3 does not support the interfaces listed in Table 18; the symbols are provided, but they always return an error and set the errno to ENOSYS.

Table 18. POSIX API functions not supported on AIX 4.3

pthread_attr_getinheritsched	pthread_attr_getschedpolicy
pthread_attr_getscope	pthread_attr_setinheritsched
pthread_attr_setschedpolicy	pthread_attr_setscope
pthread_getschedparam	pthread_mutex_getprioceiling
pthread_mutex_setprioceiling	pthread_mutexattr_getprioceiling
pthread_mutexattr_getprotocol	pthread_mutexattr_setprioceiling
pthread_mutexattr_setprotocol	pthread_setschedparam

5.3.2 Thread-safe and reentrant functions

One very important point to take care of when building multi-threaded programs is the resource handling. To avoid getting in trouble, be sure to create only thread-safe and reentrant functions as much as possible. Reentrance and thread-safety are separate concepts: A function can be either reentrant, thread-safe, both, or neither.

Reentrant A reentrant function does not hold static data over successive calls, nor does it return a pointer to static data. All data is provided by the caller of the function. A reentrant function must not call non-reentrant functions.

Thread-safe A thread-safe function protects shared resources from concurrent access by locks. Thread-safety concerns only the implementation of a function and does not affect its external interface. The use of global data is thread-unsafe. It should be maintained per thread or encapsulated so that its access can be serialized.

Reentrant and thread-safe libraries are useful in a wide range of parallel (and asynchronous) programming environments, not just within threads. Thus, it is a good programming practice to always use and write reentrant and thread-safe functions.

Several libraries shipped with the AIX Base Operating System are thread-safe. In the AIX 4.3, the following libraries are thread-safe:

- libc.a - Standard C library

- libbsd.a - Berkeley compatibility library (libbsd.a)
- libm.a - Math Library
- libmsaa.a - SVID (System V Interface Definition) math library profiled
- librts.a - Run-Time Services library
- libodm.a - ODM (Object Data Manager) library
- libs.a - Security functions
- libdes.a - Data Encryption Standard
- libXm.a - Bidirectional Support in Xm
- libXt.a - X11 toolkit library
- libX11.a - X11 run time library

Some of the standard C subroutines are non-reentrant, such as the ctime and strtok subroutines. The reentrant version of the subroutines have the name of the original subroutine with a suffix _r (underscore r).

When writing multi-threaded programs, the reentrant versions of subroutines should be used instead of the original version. For example, the following code fragment:

```
token[0] = strtok(string, separators);
i = 0;
do {
        i++;
        token[i] = strtok(NULL, separators);
} while (token[i] != NULL);
```

should be replaced in a multi-threaded program by the following code fragment:

```
char *pointer;
...
token[0] = strtok_r(string, separators, &pointer);
i = 0;
do {
        i++;
        token[i] = strtok_r(NULL, separators, &pointer);
} while (token[i] != NULL);
```

Thread-unsafe libraries may be used by only one thread in a program. The uniqueness of the thread using the library must be ensured by the programmer; otherwise, the program will have unexpected behavior or may even crash.

The provided POSIX thread-safe functions on AIX are listed in Table 19.

Table 19. Thread-safe functions in AIX 4.3

asctime_r	ctime_r	flockfile
ftrylockfile	funlockfile	getc_unlocked
getchar_unlocked	getgrgid_r	getgrnam_r
getpwnam_r	getpwuid_r	gmtime_r
localtime_r	putc_unlocked	putchar_unlocked
rand_r	readdir_r	strtok_r

Table 20 lists all non thread-safe functions.

Table 20. Non thread-safe functions in AIX 4.3

asctime	catgets	ctime
dbm_clearerr	dbm_close	dbm_delete
dbm_error	dbm_fetch	dbm_firstkey
dbm_nextkey	dbm_open	dbm_store
dirname	drand48	ecvt
encrypt	endgrent	endpwent
endutxent	fcvt	gamma
gcvt	getc_unlocked	getchar_unlocked
getdate	getenv	getgrent
getgrgid	getgrnam	getlogin
getopt	getpwent	getpwnam
getpwuid	getutxent	getutxid
getutxline	getw	gmtime
l64a	lgamma	localtime
lrand48	mrand48	nl_langinfo
ptsname	putc_unlocked	putchar_unlocked
putenv	pututxline	rand

readdir	setgrent	setkey
setpwent	setutxent	strerror
strtok	ttyname	

5.3.3 Inspecting a process and its kernel threads

AIX provides the `ps` command for showing current process status. Setting the appropriates flags, we can also show the kernel threads information. On AIX, Version 4.3, to display information about all processes and kernel threads, enter:

```
ps -emo THREAD
```

The output is similar to:

```
USER   PID  PPID  TID S  C PRI SC   WCHAN   FLAG   TTY BND  CMD
jane  1716 19292    - A 10  60  1       * 260801 pts/7   -  biod
   -     -     - 4863 S  0  60  0 599e9d8   8400     -   -  -
   -     -     - 5537 R 10  60  1 5999e18   2420     -   3  -
luke 19292 18524    - A  0  60  0 586ad84 200001 pts/7   -  -ksh
   -     -     - 7617 S  0  60  0 586ad84    400     -   -  -
luke 25864 31168    - A 11  65  0       - 200001 pts/7   -  -
   -     -     - 8993 R 11  65  0       -      0     -   -  -
```

Where the columns are defined as:

USER The login name of the process owner.
PID The process ID of the process.
PPID The process ID of the parent process.
TID The thread ID of the kernel thread.
S The state of the process or kernel thread.
C The CPU utilization of the process or kernel thread.
PRI The priority of the process or kernel thread.
SC The suspend count of the process or kernel thread.
WCHAN The wait channel of the process or kernel thread.
FLAG The flags of the process or kernel thread.
TTY The controlling terminal of the process.
BND The CPU to which the process or kernel thread is bound.
CMD The command being executed by the process.

In the example output shown above, we can see that process 1716 has two kernel threads, one in SLEEPING state and the other in RUNNING.

5.4 Program parallelization with compiler directives

So far this chapter has discussed the features of the pthreads interface for writing parallel programs. The latest compilers from IBM also support two alternative methods of creating parallel code, both of which involve the use of compiler directives to indicate the purpose of the code. They are:

- IBM Directives
- OpenMP Directives

OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify shared memory parallelism in Fortran and C/C++ programs. Refer to the OpenMP Web site for more information:

`http://www.openmp.org`

In both instances, the compiler uses the given directives to replace sections of code with parallel constructs.

The number of parallel threads generated depends on the run time data. This feature can improve applications performance, especially when taking advantage of multiple CPU machines.

The region of code to be parallelized can be defined automatically by the compiler, for the IBM Directives, or can be explicitly defined by the programer using a proper `#pragma` syntax for both IBM and OpenMP directives.

Note

The VisualAge C++ Professional for AIX, Version 5 compiler only supports IBM and OpenMP directives for parallelization on C language code compilation.

5.4.1 IBM directives

IBM directives for parallelization are based on the possibility of parallelizing *countable loops*. A loop is considered countable when the following rules can be applied:

- There is no branching into or outside of the loop.
- The *incremental expression* (`incr_expr`) is not within a critical section.

Table 21 shows the C language control flow statements and the regular expressions that define when they can be treated as countable loops.

Table 21. Regular expressions for countable loops

C Control of flow	Regular expression

for	`for ([iv]; exit_cond; incr_expr)`
	`  statement`
	
	`for ([iv]; exit_cond; [expr]`
	`{`
	`  [declaration_list]`
	`  [statement_list]`
	`  incr_expr;`
	`  [statement_list]`
	`}`

while	`while (exit_cond)`
	`{`
	`  [declaration_list]`
	`  [statement_list]`
	`  incr_expr;`
	`  [statement_list]`
	`}`

do	`do {`
	`  [declaration_list]`
	`  [statement_list]`
	`  incr_expr;`
	`  [statement_list]`
	`} while (exit_cond)`

where:

exit_cond	iv <= ub
	iv < ub
	iv >= ub
	iv > ub
incr_expr	++iv
	iv++
	--iv
	i--
	iv += incr

iv -= incr

iv = iv + incr

iv = incr + iv

iv = iv - incr

iv Iteration variable. The iteration variable is a signed integer that has either automatic or register storage class, does not have its address taken, and is not modified anywhere in the loop except in *incr_expr*.

incr Loop invariant signed integer expression. The value of the expression is known at compile-time and is not 0. *incr* cannot reference extern or static variables, pointers or pointer expressions, function calls, or variables that have their address taken.

ub Loop invariant signed integer expression. *ub* cannot reference extern or static variables, pointers or pointer expressions, function calls, or variables that have their address taken.

In general, a countable loop is automatically parallelized only if all of the following conditions are met:

- The order in which loop iterations start or end does not affect the results of the program.
- The loop does not contain I/O operations.
- Floating point reductions inside the loop are not affected by round-off error, unless the -qnostrict option is in effect.
- The -qnostrict_induction compiler option is in effect.
- The -qsmp compiler option is in effect without its omp sub option. The compiler must be invoked using a thread-safe compiler mode.

5.4.1.1 The IBM directives syntax

When using IBM directives for explicitly defining parallel portions of code, use the following syntax:

```
#pragma ibm pragma_name_and_args
<countable for|while|do loop>
```

Pragma directives must appear immediately before the section of code to which they apply. For most parallel processing pragma directives, this section of code must be a countable loop, and the compiler will report an error if one is not found.

More than one parallel processing pragma directive can be applied to a countable loop. For example:

```
#pragma ibm independent_loop
#pragma ibm independent_calls
#pragma ibm schedule(static,5)
<countable for|while|do loop>
```

Some pragma directives are mutually-exclusive. If mutually-exclusive pragmas are specified for the same loop, the last pragma specified applies to the loop. In the example below, the `parallel_loop` pragma directive is applied to the loop, and the `sequential_loop` pragma directive is ignored:

```
#pragma ibm sequential_loop
#pragma ibm parallel_loop
```

Other pragmas, if specified repeatedly for a given loop, have an additive effect. For example:

```
#pragma ibm permutation (a,b)
#pragma ibm permutation (c)
```

is equivalent to:

```
#pragma ibm permutation (a,b,c)
```

Table 22 shows all IBM pragma directives supported by the latest compilers.

Table 22. IBM pragma supported by VA C++ compiler, Version 5.0

Pragma	Description
#pragma ibm critical	Instructs the compiler that the statement or statement block immediately following this pragma is a critical section.
#pragma ibm independent_calls	Asserts that specified function calls within the chosen loop have no loop-carried dependencies.
#pragma ibm independent_loop	Asserts that iterations of the chosen loop are independent, and that the loop can therefore be parallelized.
#pragma ibm iterations	Specifies the approximate number of loop iterations for the chosen loop.
#pragma ibm parallel_loop	Explicitly instructs the compiler to parallelize the chosen loop.
#pragma ibm permutation	Asserts that specified arrays in the chosen loop contain no repeated values.

Pragma	Description
#pragma ibm schedule	Specifies scheduling algorithms for parallel loop execution.
#pragma ibm sequential_loop	Explicitly instructs the compiler to execute the chosen loop sequentially.

5.4.2 OpenMP directives

OpenMP directives exploit shared memory parallelism by defining various types of parallel regions. Parallel regions can include both iterative and non-iterative segments of program code.

Pragmas fall into four general categories:

1. The first category of pragmas lets you define parallel regions in which work is done by threads in parallel. Most of the OpenMP directives either statically or dynamically bind to an enclosing parallel region.
2. The second category lets you define how work will be distributed across the threads in a parallel region.
3. The third category lets you control synchronization among threads.
4. The fourth category lets you define the scope of data visibility across threads.

5.4.2.1 The OpenMP syntax

The syntax for using OpenMP directives is as follow:

```
#pragma omp pragma_name_and_args
statement_block
```

Pragma directives generally appear immediately before the section of code to which they apply.

The omp parallel directive is used to define the region of program code to be parallelized. Other OpenMP directives define visibility of data variables in the defined parallel region and how work within that region is shared and synchronized.

For example, the following example defines a parallel region in which iterations of a for loop can run in parallel:

```
#pragma omp parallel {
   #pragma omp for
      for (i=0; i<n; i++)
      ...
```

```
}
```

Next example defines a parallel region in which two or more non-iterative sections of program code can run in parallel:

```
#pragma omp parallel region {
   /* code here is executed by all threads */
   #pragma omp sections {
      /* each section is executed once */
      #pragma omp section
         structured_block_1
            ...
      #pragma omp section
         structured_block_2
            ...
         ....
   }
}
```

Table 23 shows the OpenMP pragmas that are supported by the latest compilers.

Table 23. OpenMP pragmas supported by VA C++ compiler, Version 5.0

Pragma	Description
#pragma omp parallel	Defines a parallel region to be run by multiple threads in parallel. With specific exceptions, all other OpenMP directives work within parallelized regions defined by this directive.
#pragma omp for Preprocessor	Work-sharing construct identifying an iterative for-loop whose iterations should be run in parallel.
#pragma omp parallel for	Shortcut combination of omp parallel and omp for pragma directives used to define a parallel region containing a single for directive.
#pragma omp sections	Work-sharing construct identifying a non-iterative section of code containing one or more subsections of code that should be run in parallel.
#pragma omp parallel sections	Shortcut combination of omp parallel and omp sections pragma directives used to define a parallel region containing a single sections directive.

Pragma	Description
#pragma omp single	Work-sharing construct identifying a section of code that must be run by a single available thread.
#pragma omp master	Synchronization construct identifying a section of code that must be run only by the master thread.
#pragma omp critical	Synchronization construct identifying a statement block that must be executed by a single thread at a time.
#pragma omp barrier	Synchronizes all the threads in a parallel region.
#pragma omp atomic	Identifies a memory location that must be updated atomically and not be exposed to multiple, simultaneous writing threads.
#pragma omp flush Preprocessor	Synchronization construct identifying a point at which the compiler ensures that all threads in a parallel region have the same view of specified objects in memory.
#pragma omp ordered	Identifies a structure block of code that must be executed as a sequential loop.
#pragma omp threadprivate	Defines the scope of selected file-scope data variables as being private to a thread, but file-scope visible within that thread.

Chapter 6. Making our programs run faster

No programmer or user ever complains that their programs are too fast. This chapter covers some of the compiler options and coding techniques that can be used to improve the execution speed of applications. The first part of the chapter deals with measurements because we can claim to improve something only if we measure it before and after our intervention. As we are going to see, code changes and compiler options can sometimes give unexpected results, and this chapter provides some insight in distinguishing good ideas from not-so-good ones.

6.1 Measuring tools

There are two simple ways to time a program:

1. Use the time built-in Korn shell command. For example:

   ```
   $ time a.out
   ```

   This provides information about real (elapsed) time, user time, and system time for a program. There is also a `time` command in the C shell, and a `/usr/bin/time` command (which is different from the `time` command used by either the Korn or C shell) that you can use with other shells.

   On a multiprocessor system, the elapsed time of an application can be considerably smaller than the sum of the user and system times shown if the application is multi-threaded. This is because the user and system times are the sum of the user and system times for all threads.

   The output of each `time` command is slightly different as shown in the three examples in Figure 22 on page 164.

```
$ time a.out

real    0m3.70s
user    0m3.63s
sys     0m0.06s
$ csh
% time a.out
3.6u 0.0s 0:03 100% 11+4874k 0+0io 0pf+0w
% /usr/bin/time a.out

Real   3.61
User   3.53
System 0.07
```

Figure 22. Different versions of time command

Notice that in addition to the format of the output being different, the time results also differ slightly because the load of the system varies a little from one moment to the next.

2. Use the gettimeofday() C function.

   This will only provide elapsed time information, but allows more precise time measurement because you can precisely time a section of the program rather than the entire program. Of course, using the gettimeofday function means that you must have the source code of the application and be able to recompile it.

The examples in this chapter use both timing techniques as described above. In addition, the chapter describes another technique, called profiling, discussed in Section 6.5, “Profiling your programs” on page 171.

To measure the execution time of program sections, consider the following two functions. One is called start_timing() and marks the start of time accounting. The second is called stop_timing(); it measures the elapsed time in microseconds since start_timing() was called and writes it on stdout.

Although the functions produce a result in microseconds, the program is running on a multiuser machine, the load of which is variable, and experience shows that two runs of the same program will give results that can differ by tens of milliseconds and perhaps even more.

Figure 23 on page 165 shows sample implementations of the timing functions. For the convenience of the examples used in this book, the

functions were defined in the header file param.h, although the functions could just as easily be incorporated into a library of utility routines.

```
/* Timing functions : start_timing() and stop_timing()
   THESE FUNCTIONS ARE SUPPOSED TO BE CALLED FROM THE MAIN PROGRAM ONLY
   Do not use them in threads, as they are NOT designed as reentrant */

#include <sys/time.h>

/* Measurement of time : start_timing() and stop_timing(). */

static struct timeval starting, ending;
static long microseconds;

start_timing()
{
   gettimeofday (&starting, NULL);
}

stop_timing()
{
   gettimeofday (&ending, NULL);
   microseconds = (ending.tv_sec - starting.tv_sec) * 1000000
      +ending.tv_usec- starting.tv_usec;
   printf ("%15ld microseconds\n", microseconds);
}
```

Figure 23. Sample timing functions

For an extended discussion on the full suite of performance measuring tools available on the AIX operating system, refer to the following IBM Redbooks:

- *Understanding RS/6000 Performance and Sizing*, SG24-4810
- *RS/6000 Performance Tools in Focus*, SG24-4989

6.2 About the examples

Wherever possible, this chapter uses a simple, but somehow realistic, matrix multiplication program as an example when describing the various compiler options and coding techniques.

However, some optimizations can best be demonstrated by using very specific examples to show how the optimization works in their case. For instance, the effects of function call optimization or malloc() optimization are

best demonstrated on programs that essentially make many function calls or mallocs just for the sake of it. Not only are the examples easier to understand that way, but this increases the signal-to-noise ratio when the optimization effects are measured.

The best results obtained in reducing various bottlenecks, whether in real-life examples or in limited condition examples, are summarized in Section 6.11, “A summary of our best results” on page 238.

Most of the examples used in the book are CPU intensive, with either little or no I/O taking place. This has been done to remove the impact of other factors, such as user reaction time, on the timing results obtained.

6.2.1 What to expect from example timing

Most examples are deliberately constructed to show, in the simplest possible way, how to reduce a particular bottleneck. When a given change reduces execution time by a factor of 5 on an example, you should understand that the time spent by your program on this particular bottleneck will be reduced by a factor of 5, not that your whole program will be five times faster. For example, if your program spends 20 percent of its total time in that bottleneck, a reasonable expectation will be a total execution time reduced by 16 percent. This is shown in Figure 24 on page 167.

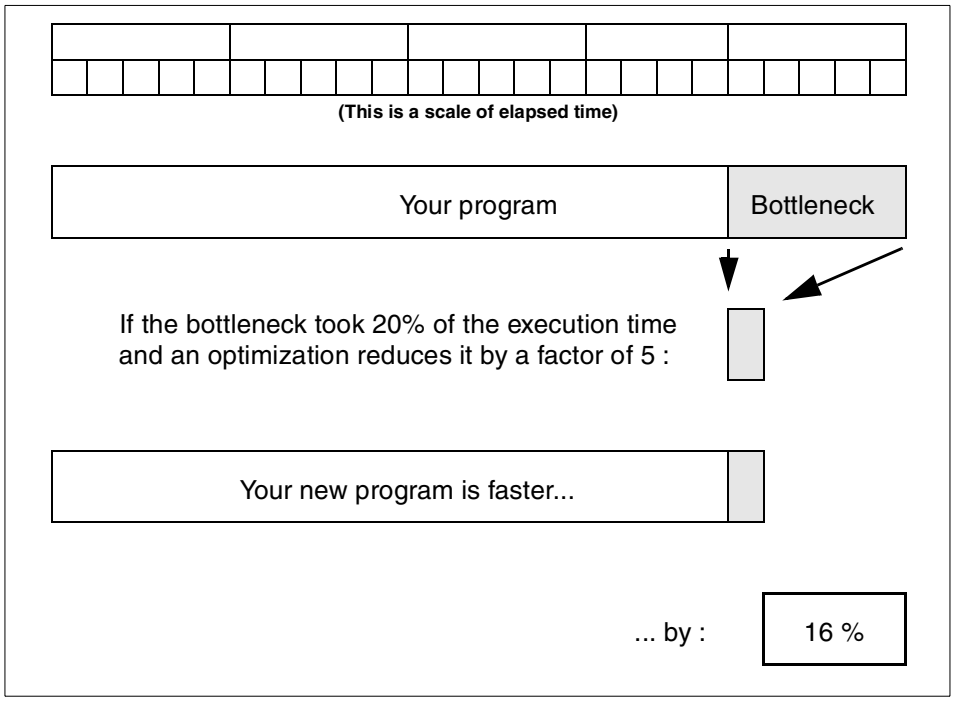


Figure 24. What to expect from an optimization

In cases where the program performs a large amount of I/O, the apparent reduction in execution time may be considerably less due to factors, such as the impact of disk access time and user reaction time. Although the reduction in elapsed time may not be as dramatic as the 16 percent described above, the application should be placing less of a CPU load on the system.

6.2.2 Run the examples on your machine

The sample timings shown in this book were obtained on an RS/6000 Model S70 Advanced server, with the following characteristics:

- 12 RS64 II processors at 262 MHz
- 8 MB of L2 cache per processor
- 1024 MB of main memory
- 1 GB of swap space

If your machine is different, you will obtain different timing results when running the same examples. It is recommended that you run the samples and obtain your own timing results for comparison. Both the raw characteristics of

your machine and the degree to which it is loaded will play a role in the results.

If your machine speed or memory size is very different, you can change some loop or size options (a `SIZE` compile-time variable is available for that in many examples) to obtain results in the meaningful and convenient range going from 100 ms to 30 seconds.

6.3 Timing a typical program

The matrix.c program, shown in Figure 25 on page 169, is a simple program to multiply two NxN matrices. The example will time the multiplication component of the program, the time of which is in N^3 depending on the defined variable SIZE.

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

/* The following program initializes two matrices and multiplies them.
** Its purpose is to test the efficiency of -O and -O3 options
*/

#include "param.h"

main(int argc, char * argv[])
{
    long i, j, k;
    int garbage;
    double A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE];

    /* create matrices A and B */
    for (i=0; i<SIZE; ++i)
        for (j=0; j<SIZE; ++j)
        {
            A[i][j]=i+j;
            B[i][j]=i-j;
        }

    /* Matrix product */
    start_timing();
    for (i=0; i<SIZE; ++i)
        for (j=0; j<SIZE; ++j)
        {
            C[i][j] = 0;
            for (k=0; k<SIZE; ++k)
                C[i][j] += A[i][k]*B[k][j];
        }
    stop_timing();

    /* We are not interested at the result, but the
    ** optimizer should not know it !
    */
    garbage = open("junk", O_RDWR|O_CREAT);
    write(garbage, C, sizeof C);
    close(garbage); unlink("junk");
}
```

Figure 25. The matrix.c sequential matrix multiplication program

The results for multiplying two 500x500 matrices with no compiler optimization options is as follows:

```
$ cc matrix.c -DSIZE=500 -o matrix
$ matrix
       26430222 microseconds
```

6.4 Useful basic compiler options

The first step in getting a program to go faster is to use the options provided by the compiler. The IBM compiler products available on AIX have evolved over many years and, as a result, are capable of generating very well optimized code, assuming you use the correct options.

The first part of this section discusses the basic compiler options that can be used to provide maximum benefit for most programs.

The compiler options of interest are:

-g Generate extra symbolic information for the debugger or the tprof profiler.

-p Generate profiling information for use with the prof profiler.

-pg Generate profiling information for use with the gprof profiler.

-O Optimize program (-O, -O3, -O4, -O5).
-O will attempt to make the best possible use of registers. Further levels will group calculations in order to remove redundancy, move non loop-dependent code outside of loops, and sometimes change the order in which instructions are executed for extra efficiency. For this reason, the -O options should generally not be used with the -g or -p options.

-qalign Choose between CPU-fast memory access (aligned data) or compact representation of data at the expense of a little CPU overhead (unaligned data).

-qfuncsect Generate additional information so that it is possible to avoid some useless or redundant function loads.

-qtune Compile for best results on a given machine type.

-qarch Compile using the instruction set for the defined architecture.

6.5 Profiling your programs

It is not hard to know which part of the matrix.c program is the most time-consuming, but for a real-life program with potentially many thousands of lines of code, it is more difficult to detect.

The technique of profiling a program allows you to discover which parts of your program are consuming the most CPU cycles. This, in turn, allows you to target your initial code optimization efforts to attempt to maximize the improvement.

The AIX, Version 4.3 operating system provides and supports three profiling programs for analyzing your code. Each profiler has a different method of operation and provides a different type of information.

6.5.1 Profiling with tprof

Profiling a program with `tprof` consists of two steps:

1. Compile your program with the -g flag. This adds additional symbol information to the executable, which is used to determine the line of source code being executed at any instant in time.
2. Run your program from the `tprof` profiling environment. For example:

   ```
   # tprof a.out
   ```

Only the root user can perform the second part because `tprof` enables a form of system tracing. When `tprof` is running, the clock tick interrupt handler, which runs 100 times a second, records the process ID and instruction being executed when the interrupt occurred. When the execution of the application has completed, `tprof` disables the system tracing it had enabled. It then generates a number of output files from the data it collected.

The `tprof` command is part of the perfagent.tools fileset supplied with AIX, Version 4.3. The fileset is not installed by default when installing the operating system; so, you will have to use your AIX product media to install it if the command is not present. AIX, Version 4.1 and AIX, Version 4.2 did not supply the perfagent.tools fileset as part of the operating system but as part of the separately orderable Performance Agent Licensed Program Product.

Figure 26 on page 172 shows an example of profiling a program with the `tprof` command. Note that the source code files for the application must be in the directory from which `tprof` is run; otherwise, the output files containing microprofiling information will not be produced.

```
$ cc -g -DSIZE=500 matrix.c
$ tprof a.out
You Must Run Tprof As Root
$ su
root's Password:
# tprof a.out
Starting Trace now
Starting  a.out
Thu Feb 24 10:28:34 2000
System: AIX itsosrv1 Node: 4 Machine: 00017D374C00

       26514787 microseconds
Trace is done now
29.867 secs in measured interval
 * Samples from __trc_rpt2
 * Reached second section of __trc_rpt2
#^D
$ls -lrt | tail -8
-rw-r--r--   1 root     system     36164 Feb 24 10:29 __trc_rpt2
-rw-r--r--   1 fda      project        0 Feb 24 10:29 __tmp.s
-rw-r--r--   1 fda      project       47 Feb 24 10:29 __ldmap
-rw-r--r--   1 fda      project      997 Feb 24 10:29 __a.out.all
-rw-r--r--   1 fda      project     2600 Feb 24 10:29 __tmp.u
-rw-r--r--   1 fda      project      260 Feb 24 10:29 __tmp.k
-rw-r--r--   1 root     system       953 Feb 24 10:29 __t.main_matrix.c
-rw-r--r--   1 root     system       110 Feb 24 10:29 __h.matrix.c
```

Figure 26. Profiling a program with tprof

The `tprof` command produces a number of output files in the current directory, all prefixed with __ (two underscore characters). The files that start with __tmp are temporary files used in producing the microprofiling output files.

For each source file in the application, the `tprof` command produces a file called __h.*filename*, which shows the lines in that file that consumed the most CPU time. Since our example only has a single source file, there is only one __h file, __h.matrix.c, as shown in Figure 27 on page 173. This file contains the essential information for us: Which lines received the most CPU ticks. These are called the *hot lines* (hence the __h). The contents of the file is a hit-parade of the lines where our program spent the most CPU time.

```
Hot Line Profile for ./matrix.c


  Line   Ticks

    32   1868
    31    772
    30      4
    22      2
    19      2
    21      1
    28      1
```

Figure 27. Contents of __h.matrix.c

In addition to the hot lines file for each source file, `tprof` also produces a file for each function in the executable. The files have names of the format __h.*function_filename*. Our simple example only has a main routine; so, the only function file produced by tprof is __t.main_matrix.c as shown in Figure 28 on page 174. This file shows the hot lines in their correct context.

```
Ticks Profile for main in ./matrix.c


  Line   Ticks   Source

   18      -         for (i=0; i<SIZE; ++i)
   19      2             for (j=0; j<SIZE; ++j)
   20      -             {
   21      1                 A[i][j]=i+j;
   22      2                 B[i][j]=i-j;
   23      -             }
   24      -
   25      -         /* Matrix product */
   26      -         start_timing();
   27      -         for (i=0; i<SIZE; ++i)
   28      1             for (j=0; j<SIZE; ++j)
   29      -             {
   30      4                 C[i][j] = 0;
   31    772                 for (k=0; k<SIZE; ++k)
   32   1893                     C[i][j] += A[i][k]*B[k][j];
   33      -             }
   34      -         stop_timing();
   35      -
   36      -         /* We are not interested at the result, but the
   37      -         ** optimizer should not know it !
   38      -         */
   39      -         garbage = open("junk", O_RDWR|O_CREAT);
   40      -         write(garbage, C, sizeof C);
   41      -         close(garbage); unlink("junk");
   42      -     }


2675 Total Ticks for main in ./matrix.c
```

Figure 28. The __t file shows execution time of instructions in the source program

The information tells us that line 32 is the most time-consuming part of our program, followed by line 31, in case we did not know. We can now concentrate on ways to make these lines run faster, as they are responsible for most of the execution time.

How can we make the application run faster? The rest of this section is going to get a little technical, but you can skip it at first reading. It is only intended to

show the burden that program optimization could have been if compiler designers had not built the optimization options in. To take the easy option, skip to Section 6.6, “Optimizing with the -O option” on page 179.

6.5.1.1 Manual optimization example

The first thing to consider is declaring the loop indexes, especially the innermost one, *k*, as a register variable. However, as k is precisely a loop index, the compiler may have already done that automatically. This will be verified.

If the address of `C[i][j]` is computed for every value of k, this is very inefficient. This could be avoided by one of these two means:

1. Compute `p=&C[i][j]` before entering the k loop and replace the use of `C[i][j] +=` with `*p +=` . The pointer variable p would also be declared as a register variable.
2. Another (perhaps more readable) way is to declare a new variable of type `register double s` and replace `C[i][j] +=` with `s +=`. Only when we get out of the k loop, shall we perform the assignment `C[i][j]=s;`

Table 24 shows the results reported by `tprof` for line 32 when trying the three enhancements described above.

Table 24. Trying some enhancements by hand

method	result in ticks	interpretation
original program	1893	
register long k	1879	The compiler had already done that for us.
(*p) +=	3021	Not a good idea
register double s	1581	Slight improvement

An interesting option of the compiler, -qlist, can show you the generated code in assembler format. Even if you do not understand the details of assembler code, this option will sometimes give you an idea of what is going on.

We can also use the profiler to have an idea about the speed of different C arithmetic operations according to the types of their operands. This can be done using the code shown in Figure 29 on page 176.

```
#define test(x) for (k=0; k<100000000; ++k) x;

int i, j, k, m[1000];
long ii, jj, kk, mm[1000];
double x, y, z, t[1000];
long double xx, yy, zz, tt[1000];

main()
{
 test ()
 test ( i = i+1 )
 test ( i = i*j )
 test ( i = k%1000 )
 test ( m[k%1000] = k )
 test ( ii = ii+1 )
 test ( ii = ii*jj )
 test ( mm[k%1000] = (long) k )
 test ( x = x+1. )
 test ( x = x*y)
 test ( t[k%1000] = (double) k)
 test ( xx = xx+1.)
 test ( xx = xx*yy)
 test ( tt[k%1000] = (long double) k)
}
```

Figure 29. profil.c: Timing some typical C instructions

Notice the `test()` line, which will time an empty loop. The corresponding tick count should be subtracted from the other tick counts in order to know the execution time of the computations themselves, not of the loop executing them.

The output from running the program is shown in Figure 30 on page 177.

```
Ticks Profile for main in ./profil.c

  Line   Ticks   Source

   10    348      test ()
   11    537      test ( i = i+1 )
   12    765      test ( i = i*j )
   13    524      test ( i = k%1000 )
   14    653      test ( m[k%1000] = k )
   15    535      test ( ii = ii+1 )
   16    765      test ( ii = ii*jj )
   17    621      test ( mm[k%1000] = (long) k )
   18    803      test ( x = x+1. )
   19    880      test ( x = x*y)
   20    717      test ( t[k%1000] = (double) k)
   21    802      test ( xx = xx+1.)
   22    879      test ( xx = xx*yy)
   23    725      test ( tt[k%1000] = (long double) k)
   24      -     }

9554 Total Ticks for main in ./profil.c
```

Figure 30. The output of profil.c

Table 25 summarizes the actual cost in clock ticks of each type of calculation.

Table 25. Real execution tick count

computation	real tick count
i = i + 1	188
i = i * j	416
i = k % 1000	175
m[k%1000] = k	304
ii = ii + 1	186
ii = ii * jj	416
mm[kk%1000] = (long) k	272
x = x + 1.	454
x = x * y	531

computation	real tick count
t[k%1000] = (double) k	368
xx = xx + 1.	453
xx = xx * yy	530
tt[kk%1000] = (long double) k	376

6.5.2 Other profilers

The nice thing about `tprof` is that it provides profiling at the source-code level (which is sometimes called *microprofiling*) and does not need anything other than the -g option to run. This -g option is routinely used in the development process most of the time because it is needed if you want to run the executable code under control of a debugger.

The not quite so nice thing is that you have to give the root password to whoever wants to use it, and multiplying the number of people having the root password also multiplies the risk of having somebody damaging other people's files by mistake.

For that reason, two other profilers can be used. These profilers give only basic information (mostly about function call statistics), but can be run by any user. The output of these profilers may be sufficient in most cases to determine the areas of code that would benefit most from performance improvement. These profilers need the sources to be recompiled with additional compiler options.

6.5.2.1 prof

The prof profiler is a basic profiler that will provide you the number of milliseconds spent in every function your program called. The application should be compiled with the -p option before using prof.

It does not provide much usable information to use with the matrix.c example, as you can judge by yourself. This is mainly because the application does not make many function calls.

```
$ cc -p matrix.c -DSIZE=100 && prof a.out
Name                 %Time     Seconds     Cumsecs  #Calls   msec/call
.stop_timing          99.1        3.27        3.27
.main                  0.9        0.03        3.30       1     30.
$
```

Figure 31. Using prof

6.5.2.2 gprof

The gprof profiler is used to obtain a call graph of the application. Use of gprof requires a program to be recompiled with the -pg option. The application is run as normal and leaves behind a gmon.out file. The `gprof` command is then used to process the gmon.out file to produce the call graph information.

```
$ cc -pg matrix.c -DSIZE=100
$ a.out
$ gprof
```

Figure 32. Using gprof

Use gprof when you have a lot of functions called and you want to know which ones spent the most time.

6.6 Optimizing with the -O option

The -O compiler option performs some basic optimization of the code that is produced. The result of using -O with the matrix.c program can be seen in Figure 33.

```
    25      -      for (i=0; i<SIZE; ++i)
    26      -      for (j=0; j<SIZE; ++j)
    27      -          { C[i][j] = 0;
    28    373            for (k=0; k<SIZE; ++k) C[i][j] += A[i][k]*B[k][j];
    29      -          }
```

Figure 33. Improvement with optimization

Compared to the results in Figure 28 on page 174, we see that for this program, using the -O option is more efficient that trying to optimize things manually. Moreover, it allows us to preserve program readability by writing

instructions in a “natural” way, readable for human beings, rather than using more efficient, but more obscure, forms as seen in Table 24 on page 175.

People who program in assembler language admit that while it is still possible to do a very good manual optimization for a short program (for example, a device driver), the C optimizer will beat anyone except the very experienced programmer when the source program is more than a few pages. Why is this so? The RS/6000 has a lot of registers and its use of the chromatic algorithm guarantees one of the very best possible register allocation. An excellent low-level language programmer knowing the application (and information the compiler does not have) could do better, but at the expense of program readability and, therefore, maintainability.

Profiling a program, as described in Section 6.5, “Profiling your programs” on page 171, stays, nevertheless, a good way to know where the real program bottlenecks are and act accordingly. For instance, bottlenecks can be good candidates to be considered for multithreading, as we shall see in Section 6.7, “Reworking a program to use multiple processors” on page 206.

Figure 34 on page 181 shows what happens when we apply the -O option to the program profil.c shown in Figure 29 on page 176.

```
Ticks Profile for main in ./profil.c

  Line   Ticks   Source

   10      -      test ()
   11      -      test ( i = i+1 )
   12      -      test ( i = i*j )
   13     77      test ( i = k%1000 )
   14    589      test ( m[k%1000] = k )
   15      -      test ( ii = ii+1 )
   16    191      test ( ii = ii*jj )
   17    572      test ( mm[k%1000] = (long) k )
   18    153      test ( x = x+1. )
   19    153      test ( x = x*y)
   20    547      test ( t[k%1000] = (double) k)
   21    153      test ( xx = xx+1.)
   22    153      test ( xx = xx*yy)
   23    575      test ( tt[k%1000] = (long double) k)
   24      -     }

3163 Total Ticks for main in ./profil.c
```

Figure 34. Optimized profil.c

Some instructions seem to execute in no time at all. What has happened? The reasons for this are:

- The -O option tries to move out of a loop whatever can be computed once and for all outside of the loop.
- If the loop is empty, -O removes it.
- If the contents of a variable are never used, the -O option tries not to compute that variable at all.

Table 26 details the interesting results.

Table 26. Some effects of the -O option

instruction	original count	count with -O	reduction
i = i + 1	188	0	N/A
i = i * j	416	0	N/A
i = k % 1000	175	77	56 %

instruction	original count	count with -O	reduction
ii = ii + 1	186	0	N/A
ii = ii * jj	416	191	54 %
x = x + 1.	454	153	66 %
x = x * y	531	153	71 %
xx = xx + 1.	453	153	66 %
xx = xx * yy	530	153	71 %

The overall effect for the whole program is a that we get down from 9553 tick counts to 3163 tick counts, a 67 percent reduction in elapsed time. Bear in mind, however, that not all instructions were executed.

6.6.1 Optimizing at higher levels

Other compiler options are even more aggressive than -O. According to the compiler basic help, some of these optimizations, from -O on, *have the potential to alter the semantics of a user's program*.

What does altering the semantics mean? Can this level of optimization make the program *compute something other than what we think we asked*? In a sense, yes. For instance, the -O3 option boldly assumes that (a*b)*c is the same thing that a*(b*c). Mathematically, of course, it is perfectly correct since multiplication of scalars is supposed to be a commutative operation. But, if a, b, and c are floating-point variables, this might be only *almost* true, because of rounding errors. Therefore, if you are doing numerical analysis, you should be careful because these roundoffs might accumulate in some way.

The -O3 option also authorizes the compiler to make its computations in a different order than specified in the program if that allows the code to go faster. This means, unfortunately, that tprof's results will in that case be meaningless. For that reason, the -qstrict option allows to the user to turn off that particular authorization.

Let us now compare compilation and execution times of matrix.c for different optimization levels and different values of SIZE. This was done using the command sequence shown in Figure 35 on page 183.

```
for size in 4 10 25 100
do
for options in "" "-O" "-O3 -qstrict" "-O3" "-O4" "-O5"
do
echo "Options : $options"
time /usr/vac/bin/cc_r $1.c -DSIZE=$size -DOPT=\"$options\" $options
time a.out
```

Figure 35. A script to compare optimization options

The results are summarized in Table 27.

Table 27. Comparing optimization levels

Compilation option	Compilation time (seconds)	Execution time (microseconds)			
		SIZE=4	SIZE=10	SIZE=25	SIZE=100
None	0.45	14	159	2 465	187 854
-O	0.48	5	18	258	23 945
-O3 -qstrict	0.57	7	16	256	24 449
-O3	0.58	5	13	284	20 208
-O4	1.02	5	10	212	20 170
-O5	1.06	7	11	228	20 153

We see that the compilation time gets bigger with the optimization level, and that execution times get statistically lower with it, especially for SIZE = 100. Two charts will show us more clearly the amounts of magnitude involved in the two cases. Let us begin with the chart of compilation times as shown in Figure 36 on page 184.

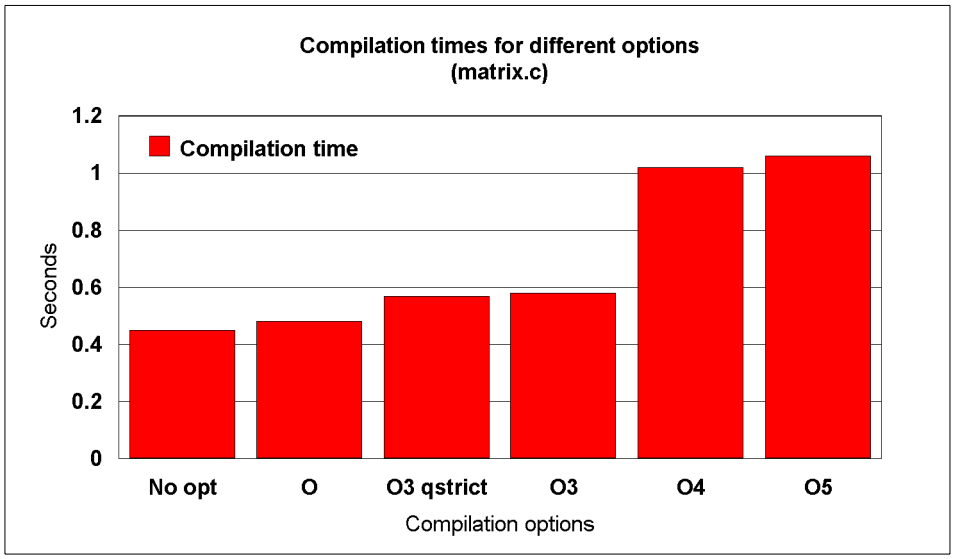


Figure 36. Comparison of compile times

We see that there is a kind of quantum leap involved when we go over -O3. Because of the law of diminishing returns, which states “do the most effective things first”, we expect the quantum leap in execution time to be between using no optimization and using basic optimization with the -O option. The comparison of execution times is shown in Figure 37 on page 185.

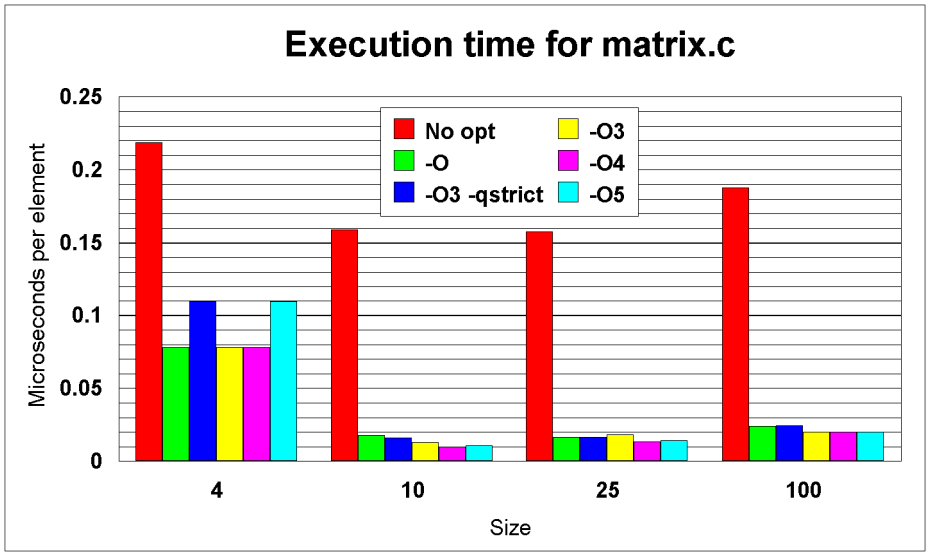


Figure 37. Comparison of execution times

The computation time per element is the ratio of the elapsed time to the number of “multiply and add” operations involved, which is of the order of $SIZE^3$. We see three things on the chart:

1. The time per element can go down to 0.01 microseconds (that is, 10 *nanoseconds*) per computed element. The only way for the RS/6000 to go so fast is having it perform many instructions at the same time using pipelining. This is precisely one of the things the optimizer tries to do, with some success.
2. The time per element first goes down with SIZE, then increases slowly. This is due to the stride effect that we shall see in Section 6.10, “The stride effect” on page 233. Notice that we see the best ordering for SIZE=100, while for a smaller value of SIZE such as 4, the results are not so clear. When your execution time is just five microseconds, any load fluctuation in the system introduces noise in the results.
3. The real execution time quantum leap for this program is clearly between using no optimization and using -O, as we expected.

This is, of course, just an example concerning this particular program. Your mileage may vary. Nevertheless, the general relationship of compilation and execution times compared with optimization level will generally stay similar.

We have now seen the best improvement. What follows will supply only marginal gains, comparatively, but stays worth to be mentioned.

6.6.2 Optimizing further with -qipa

The -qipa option performs *interprocedural analysis*. At compilation time, the compiler examines every procedure to determine which registers are used or unused. This information can be used to prevent the calling subroutine from having to save and restore the registers that are not used by the called procedure.

Let us use the program shown in Figure 38 to demonstrate how the -qipa option works.

```
#include <stdio.h>
#include <stdlib.h>

int f(int i){ return i+1; }

main()
{int i, j, k;
 for (i=0; i<100; ++i)
     { j=i;
       for (k=0; k<1000000; ++k) j=f(j);
       printf ("%d ",j);
     }
}
```

Figure 38. Example code for -qipa

Obviously, this program does nothing very useful, but we understand a lot of time is going to be spent calling the function *f* and, therefore, stacking and restoring contexts.

Let us compile with some different options and time the executions. The command sequence shown in Figure 39 on page 187 demonstrates the effect of the -qipa option on the code shown in Figure 38.

```
$ time xlc -O3 ipa.c

real    0m0.41s
user    0m0.16s
sys     0m0.06s
$ time a.out >/dev/null

real    0m3.08s
user    0m3.05s
sys     0m0.00s
$ time xlc -O3 -qipa ipa.c

real    0m0.72s
user    0m0.21s
sys     0m0.20s
$ time a.out >/dev/null

real    0m0.03s
user    0m0.01s
sys     0m0.00s
```

Figure 39. Effect of the -qipa option

Note that compilation time is longer when using the -qipa option. This is not unexpected since the compiler is performing extra processing. Notice the result, however. Using the extra knowledge of what the called procedure does, the compiler generates an executable that is more than 13 times faster in this example.

This is good news for people who like to program in Smalltalk-like style, with many short procedures calling one another. If they like to program that way, the -qipa option will give them good execution times, nevertheless. This is also good news for people using the GNU SmallEiffel compiler, which also produces a lot of small C procedures that call one another.

The -qipa option accepts additional parameters, some of which are shown in Table 28.

Table 28. Important -qipa optional parameters

Option	Effect
level=0 (or 1, or 2)	From minimal (0) to maximal (2) analysis. The default value is 1, which is intermediate

Option	Effect
lowfreq=toto1,toto2,toto3	Calling the functions toto1, toto2 and toto3 will not occur frequently. Ignore them in analysis if that can make the rest better.

6.6.3 Doing even better with -qinline

Of course, in the preceding example, we could wonder whether it is worth calling such a small procedure at all. When the procedure is so short, why not just copy its text in the main program code? In that way, we not only avoid stacking and restoring registers, but we avoid stacking and restoring a return address as well. In fact, this is already what happens when we use the string manipulation functions defined in the system header file, <string.h>.

This can be done by making f a macro instead of a function, for example:

```
#define f(x) (x+1)
```

However, with real-life functions, the macro could be very difficult to read, including a lot of line breaks with the backslash character. It can also make us lose time by thinking about possible side effects. For exampl, would f(i++) give the correct results?

The compiler provides us with the way to treat the function f as a macro without having to rewrite it. This can be done using the -qinline option, which inlines the function code. The effect of this is shown in Figure 40.

```
$ time xlc -O3 -qinline ipa.c

real    0m0.43s
user    0m0.11s
sys     0m0.08s
$ time a.out >/dev/null

real    0m0.01s
user    0m0.00s
sys     0m0.01s
```

Figure 40. The -qinline option gives the best results here

This, of course, does not mean that -qinline should be preferred every time over -qipa. As we have already said, this example, in fact *all* our examples in this chapter, are *limit situations*, as good examples should be.

Nothing prevents you from using both options because you can use optional additional parameters with -qinline to indicate what you want to be inlined as shown in Table 29.

Table 29. Some inlining options

In-lining options example	Effect
-qinline	Inline every short function.
-qinline=15	Inline every function having less than 15 lines of effective source code. Any other threshold number can be specified.
-qinline+toto:titi:tata	The toto, titi, and tata functions should be inlined.
-qinline-proust:marx:guth	The proust, marx, and guth functions should not be inlined.

Alternatively, the -qipa option also has an inline suboption.

Table 30 resumes and extends the execution time information, as well as compilation time and size of the generated executables, for our example program.

Table 30. Compilation time, size of code, execution times

Options	Compilation time (seconds)	Size of executable	Size of stripped executable	Execution time (seconds)
-O3	0.37	4187	1631	3.08
-O3 -qipa	*0.76*	4153	*1603*	0.03
-O3 -qinline	0.38	4209	1623	0.02

Compared to -qipa, the -qinline option generates a bigger stripped executable. Note that the -qipa option also needs some extra compilation time because this option needs an extensive analysis by the compiler of register allocation in each subprogram. It will allow it to compromise for the best possible combination with register allocation in the calling program. This is a lot of extra work for the compiler, but when it pays off, you will be glad you asked for it.

6.6.4 Space/time trade-off for data

Many RISC machines are sensitive to the alignment of data within memory and give best performance when data is aligned on given boundaries.

This is the default used by the compiler. The library functions shipped with AIX, and dealing with structures, have also been compiled using that option.

However, as often in computer science, “there is no silver bullet”. Aligning data on certain boundaries may be an excellent choice when you deal with one data structure, but not such a good one if you deal with an array of one thousand or one million of data structures.

This is because memory space is used less efficiently, which could lead to more frequent cache misses in the first case, or worse, more page misses, especially if the array of structures is accessed in a sequential way.

Four alignment options are supported by the compiler as detailed in Table 31.

Table 31. Alignment options supported by the compiler

-qalign=	Effect	Use
power (or full)	Uses the POWER alignment rules	Good speed, but will use some memory for padding.
twobyte (or mac68k)	Uses the Apple Macintosh alignment rules	Useful to read a file of records coming from a Mac.
packed	Uses the packed alignment rules	Good memory usage, exact influence on program execution unclear.
natural	Structure members are mapped on their natural boundaries	Biggest memory user. May be efficient for number-crunching applications when we have plenty of memory.

As this may seem a little abstract, let us define a structure mixing, sloppily, a number of variables of different sized types. This is shown in Figure 41.

```
struct t_hollow
       { char c1;
         double dd;
         char c2;
         int ii;
         char c3;
         int  jj;
       }
```

Figure 41. A structure mixing char, int, and double

Now, let us compile and execute the program shown in Figure 42 with different alignment options to see where the compiler puts the structure variables in memory.

```
#include <stdio.h>
#include <stdlib.h>
#include "param.h"

#define o(x) ((char *) &hollow[0].x - (char *) &hollow[0])
#define p(x) printf("%s uses offsets %3d to %3d\n", \
             #x , o(x), o(x)+sizeof hollow[0].x -1);

/* The following structure shoud take very different
   memory sizes according to alignment rules */

struct t_hollow
       { char c1;
         double dd;
         char c2;
         int ii;
         char c3;
         int  jj;
       } hollow[SIZE];

int k;

main()
{
 int pas;
 printf ("Size of array : %10ld, %ld elements of size %ld\n",
         sizeof hollow, SIZE, sizeof hollow[0]);
 p(c1)
 p(dd)
 p(c2)
 p(ii)
 p(c3)
 p(jj)

 pas = ((char *)&hollow[1])-(char *)&hollow[0];
 printf ("Offset next struct :%3ld\n", pas);

 start_timing();

 for (k=0; k<SIZE; ++k)
     {
       /* printf("%10d\n", k);  */
       hollow[k].c1= hollow[k].c2= hollow[k].c3='A';
       hollow[k].dd = 0. ;
       hollow[k].ii = hollow[k].jj = k;
     }

stop_timing();
}
```

Figure 42. A program to investigate how variables in a structure are aligned

The results are shown in Figure 43 on page 192.

```
-w -DSIZE=10000 -qalign=power
Size of array :     280000, 10000 elements of size 28
c1 uses offsets   0 to   0
dd uses offsets   4 to  11
c2 uses offsets  12 to  12
ii uses offsets  16 to  19
c3 uses offsets  20 to  20
jj uses offsets  24 to  27
Offset next struct : 28
           4426 microseconds

-w -DSIZE=10000 -qalign=twobyte
Size of array :     220000, 10000 elements of size 22
c1 uses offsets   0 to   0
dd uses offsets   2 to   9
c2 uses offsets  10 to  10
ii uses offsets  12 to  15
c3 uses offsets  16 to  16
jj uses offsets  18 to  21
Offset next struct : 22
           4303 microseconds

-w -DSIZE=10000 -qalign=packed
Size of array :     190000, 10000 elements of size 19
c1 uses offsets   0 to   0
dd uses offsets   1 to   8
c2 uses offsets   9 to   9
ii uses offsets  10 to  13
c3 uses offsets  14 to  14
jj uses offsets  15 to  18
Offset next struct : 19
           4168 microseconds

-w -DSIZE=10000 -qalign=natural
Size of array :     320000, 10000 elements of size 32
c1 uses offsets   0 to   0
dd uses offsets   8 to  15
c2 uses offsets  16 to  16
ii uses offsets  20 to  23
c3 uses offsets  24 to  24
jj uses offsets  28 to  31
Offset next struct : 32
           3634 microseconds
```

Figure 43. The different layouts for a structure according to alignment options

Table 32 details the offsets of the various structure members depending on the alignment option chosen.

Table 32. Offsets of structure variables according to alignment options

	power	twobyte	packed	natural
c1 (byte)	0	0	0	0
dd (double)	4 to 11	2 to 9	1 to 8	8 to 15

	power	twobyte	packed	natural
c2 (byte)	12	10	9	16
ii (int)	16 to 19	12 to 15	10 to 13	20 to 23
c3 (char)	20	16	14	24
jj (int)	24 to 27	18 to 21	15 to 18	28 to 31
Total size	28	22	19	32

Figure 44 shows a representation of how the structures are laid out in memory.

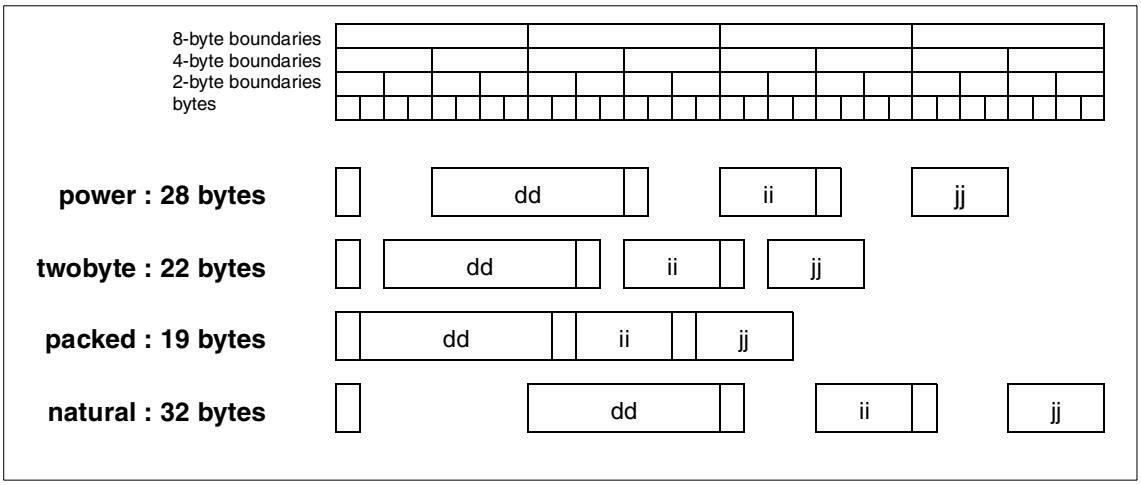


Figure 44. Memory scattering according to alignment options.

We can see from this that no padding occurs between two adjacent members when the -qalign=packed option is used.

For data file portability reasons (between machines of different constructors), it seems tempting to use the compact model. Which are the trade-offs involved?

1. Poor alignment of double variables will necessitate more machine instructions when fetching and storing them.
2. On the other hand, having a more compact data structure can have positive effects on elapsed time because:
   a. The cache may be more efficiently used, as it does not have to store any padding information.

b. A large array of structures will use less pages of virtual memory.

Once again, there is no universal answer. At most, we can see what happens to the execution times of our particular program according to a combination of SIZEs and alignment options used. The results are shown in Table 33.

Table 33. Time measurements for our sample program according to alignment

Compile time options	power	twobyte	packed	natural
SIZE=10K	4 426	4 303	4 168	3 634
SIZE=10K, -O4	2 172	1 898	*1731*	2 212
SIZE=10M	4 427 042	4 371 687	4 219 509	*3 688 052*
SIZE=10M, -O4	1 947 929	1 828 376	*1 660 141*	2 199 061

For this program, the natural alignment gives the best results when optimization is turned off; however, the packed alignment gives better results when optimization is turned on.

Note

Our structure has only *one double* floating-point value, no *long long*, and it is not used for intensive computations. With a lot of large variables misaligned and heavily used for computations, compact could, on the contrary, dramatically slow the program. Always test alignment results on your real program using real data sets before chosing an alignment option.

Now, when looking at Figure 44 on page 193, we see that the placement of variables in our structure is one of the causes for the waste of space, which, on a system that uses cache memory and virtual memory, can cause a waste of CPU cycles. No -O optimization is bold enough to rearrange the disposition of variables within a structure for us because we would then have no control over the data layout in file records; however, this can be done by hand.

The structure shown in Figure 45 on page 195 is very inefficient from a memory utilization point of view.

```
struct t_hollow
       { char c1;
         double dd;
         char c2;
         int ii;
         char c3;
         int  jj;
       }
```

Figure 45. An inherently inefficient structure

What about coding it with biggest variables first? This will not imply any change in the application code other than the structure definition. The result of rearranging is shown in Figure 46.

```
struct t_hollow
       { double dd;        /* 8 bytes */
         int ii;           /* 4 bytes */
         int  jj;          /* 4 bytes */
         char c1, c2, c3;  /* 3*1 byte */
       }
```

Figure 46. Rearranging the structure

This rearrangement should give us more compact results in whichever alignment option is used, except, of course, packed, where the size will remain the same. The comparisons of size are shown in Table 34.

Table 34. Comparing the size of each structure definition

Structure	Size in bytes for alignment			
	power	twobyte	packed	natural
Old	28	22	19	32
New	24	20	19	24

Table 35 compares the execution times of the program when using the different structures with the alignment and optimization options.

Table 35. Comparing the results for each structure definition

Compile options	Structure	power	twobyte	packed	natural
SIZE=10 K	Old	4 426	4 303	4 168	3 634
SIZE=10 K	New	3 997	3 842	4 119	4 045

Compile options	Structure	power	twobyte	packed	natural
SIZE=10 K, -O4	Old	2 172	1 898	1 731	2 212
SIZE=10 K, -O4	New	1 679	1 637	1 647	1 777
SIZE=10 M	Old	4 427 042	4 371 687	4 212 509	3 688 052
SIZE=10 M	New	1 706 133	1 675 191	1 675 272	1 731 576
SIZE=10 M, -O4	Old	1 947 929	1 828 376	1 660 141	2 199 061
SIZE=10 M, -O4	New	1 912 861	1 659 335	1 668 713	1 724 004

We see that designing a structure, with as few alignment holes as possible, sometimes enhances greatly the speed of a program without having to change anything in the code itself.

Whenever you can reduce the space that is wasted in your structures, you can increase your chances of reducing the execution time. For small data sets, this is because more of the meaningful things will be in the cache. For large data sets, this is because a bigger part of your data will be present in the active pages in physical memory.

Figure 47 on page 197 and Figure 48 on page 197 graphically illustrate the difference in execution times when using the different alignment options and the old and redesigned structures. Figure 49 on page 198 shows the padding and layout of the redesigned structure.

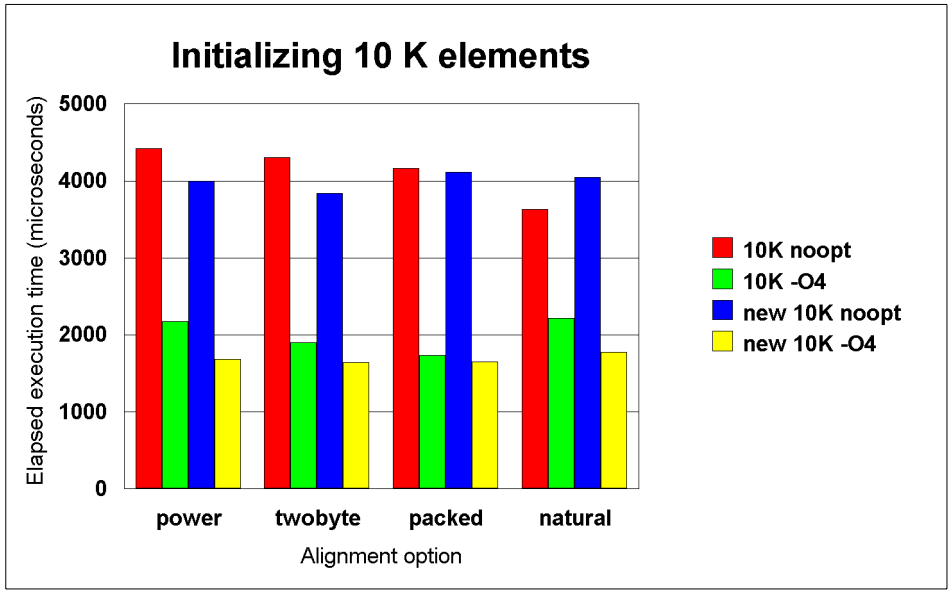


Figure 47. Initializing 10,000 structure elements

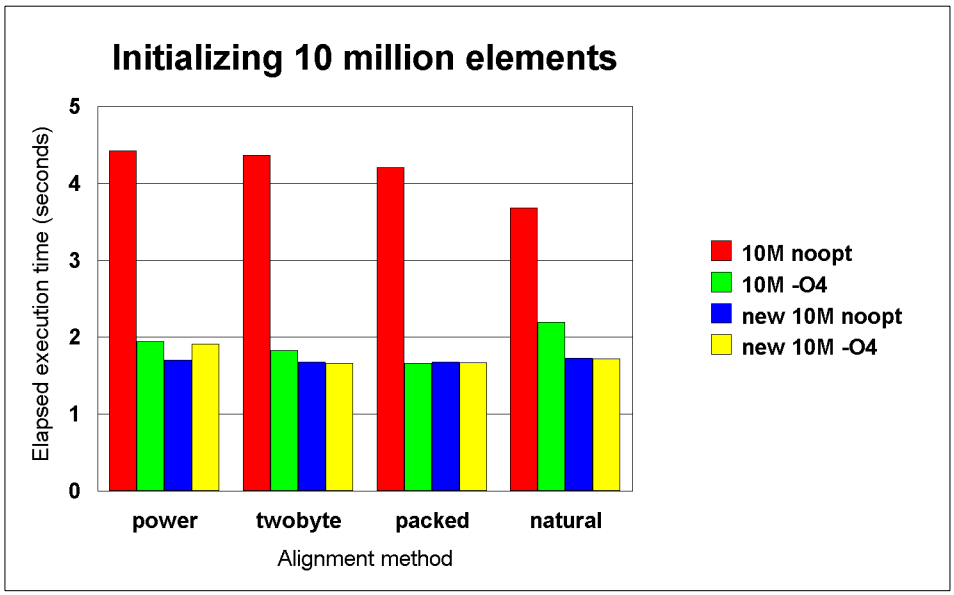


Figure 48. Initializing 10 million elements

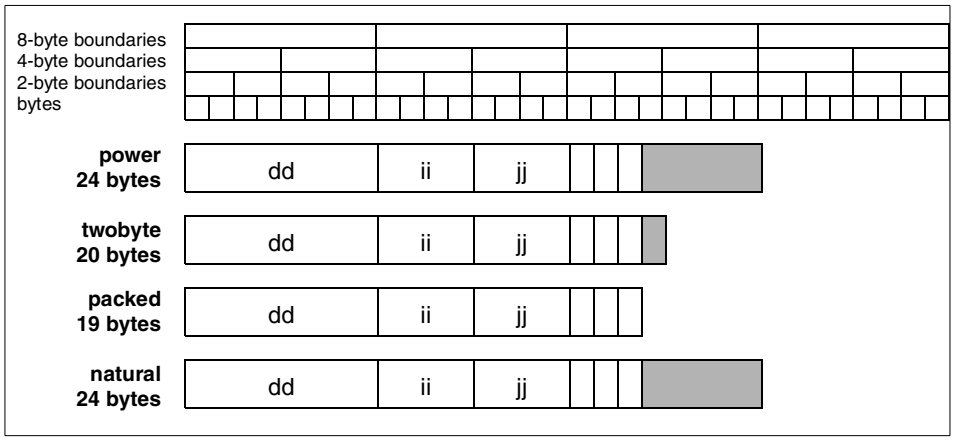


Figure 49. Layout for the reworked structure

The grey space shown in Figure 49 represents the padding between adjacent structures in an array. The compiler performs this padding because the next structure begins with a double dd, and no alignment other than packed can allow dd to start on an odd byte boundary. Power and natural even require it to start on a 8-byte boundary.

Note

Remember that some system functions return pointers to structures and accept pointers to structures as arguments. These functions in the AIX libraries have been compiled with the standard alignment option.

For this reason, if you call these functions, you should never use directly the -qalign option on the compiler line because you would give the compiler a false hint about the structures that are returned.

You can define a different alignment option for your private data structures by using the #pragma align directive.

The #pragma align directive performs the same function as the -qalign compiler option. The difference between the two is that the #pragma version can be used to limit the effect of the alteration in alignment to particular lines of code. For example:

```
#include <stdio.h>
#include <stdlib.h>
/* Structures defined in the header files will have the alignment
```

```
** defined on the compiler command line
*/
#pragma align(packed)
/* structures from now on will use the packed alignment */
struct private_data {
...
...
#pragma align(full)
/* everything from here on will have default alignment */
...
...
```

6.6.5 Light adaptation to a machine with -qtune

Not all RS/6000 systems are created equal. Historically, the instruction set used was POWER (Processing Optimization With Enhanced RISC). Later, POWER gave rise to the PPC (PowerPC) instruction set for chips developed by IBM, Apple, and Motorola. IBM also developed the POWER2 and POWER3 chips. This is a good thing since evolution cannot, and should not, be stopped.

In every one of these cases, some instructions were added in architecture, and some other ones were dropped. For compatibility reasons, some of the suppressed instructions were emulated so that legacy programs could run, nevertheless.

That means there will be more than one way to compile a program for the same result. Some of these ways may be faster on some machines only. The -qtune option gives the compiler a hint to try and make the program faster for a given type of machine. It means the program will *mostly* be used on these machines. The instructions are ordered in such a way as to suit the instruction pipeline on the specified chips.

Unless otherwise specified by a -qarch option, described in 6.6.6, “Heavy adaptation to a machine with -qarch” on page 199, the compiler only generates instructions that are present in all the instruction sets. This is called the *common* architecture and ensures complete binary portability between different types of RS/6000 hardware.

6.6.6 Heavy adaptation to a machine with -qarch

Let us suppose now that we *know* on which machine we shall run the program and wish to use its additional instructions as well. We could lose binary portability, but that would not matter much as long as we have the source code. On the other hand, we get a chance that the generated program can

take profit of the specific fast instructions of this machine and execute in a smaller amount of time.

As the new instructions are not the same in PPC and PWRn architectures, a program compiled for one architecture can be unable to run on another if it made use of those specific instructions. Trying to run it on a such a machine will then give the infamous message:

```
Illegal instruction(coredump)
```

You may consider this as a drawback if you want your programs to run everywhere, and as an advantage if you want them to be somehow protected. For instance, a software vendor could be glad, after all, that its low-cost PPC version could not run on powerful and costly POWERx machines. This way, they could market a “professional” version of its software “optimized for the powerful machines” (and sold at a higher price) by just making a simple recompilation. But, let us not forget that in such a case their maintenance costs will be higher too.

The -qarch option allows us to specify a given architecture for the compilation. The default value is com (common). Let us try to specify different values of -qarch to compile and execute matrix.c in the hope of knowing which type of architecture we are running on. We shall make two different runs in order to have an idea of the role of this option on execution time compared to the random load fluctuations of the system. Figure 50 on page 201 shows the results of this.

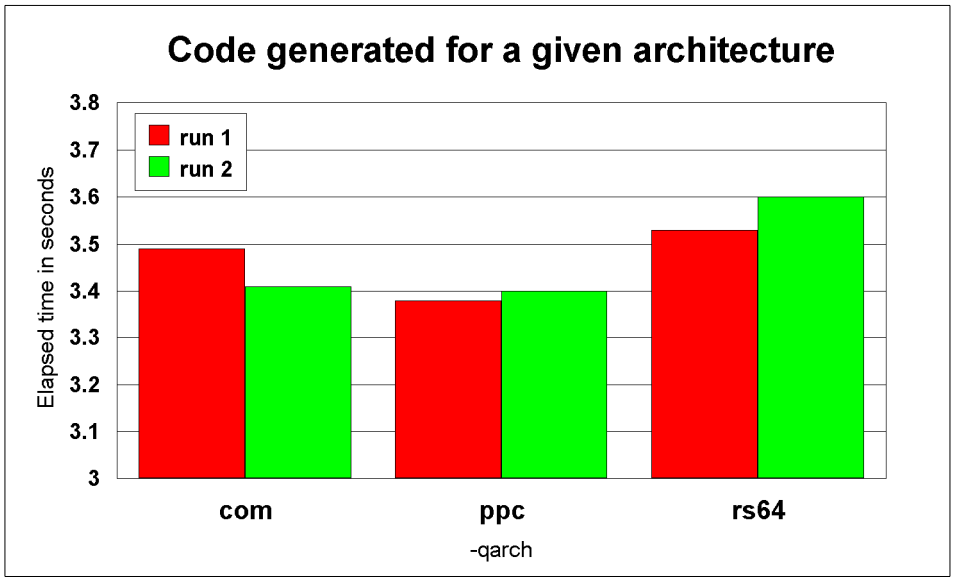


Figure 50. The -qarch option used with matrix.c

What we know for sure now is that our computer architecture is neither pwr2 nor p2sc since the execution of the code aborted with the message, `illegal instruction`, in these two cases. The other results are more controversial for this particular type of program.

However, as processor architectures evolve and get more and more specialized (personal workstation, Web server, file server, number cruncher, and so on), this -qarch option will probably become more and more important in the future.

6.6.7 Combining -qarch and -qtune

The -qarch and -qtune options are not incompatible, and combining them for your machine (or for another machine you intend to run the program on) is supposed to give you the best results. The compiler will warn you if, by mistake, you specify a -qtune option incompatible with your -qarch choice.

6.6.8 Removing redundant code from executables with -qfuncsect

Let us consider the C++ file func1.cpp shown in Figure 51 on page 202.

```
#include <stack>
#include <string>
#include <iostream>

using namespace std;

int func1()
{
   stack<string> stk;
   stk.push("apple");
   stk.push("banana");
   stk.push("orange");
   cout << "size = " << stk.size() << endl;
   cout << "top = " << stk.top() << endl;
   while (!stk.empty()) {
      cout << stk.top() << endl;
      stk.pop();
   }
   return 0;
}
```

Figure 51. A typical C++ function using a stack template

The code for the template is normally coded in the function at compilation time. This means that the compiler sees the code for the stack template function as part of the compilation unit and includes it as part of the output object file. The object file is created with a single CSECT (code section), which is the atomic unit used by the linker when creating the final executable. Now, many other functions coded elsewhere can also have to use this stack template, which means the code for the template will be included in many compilation units and, thus, many object files. For simplicity, we are going to define six other functions using the same template, func2.cpp to func7.cpp, which are just going to be clones of func1.cpp. The calling program for these functions is shown in Figure 52 on page 203.

When the linker is creating the final executable, it sees multiple copies of the stack template code since there is one copy in each object file produced. It can not remove the duplicate entries, however, since each one is within a single CSECT with other code that needs to be included in the final executable. This means the final executable is larger than it needs to be since it has multiple copies of certain routines.

```
using namespace std;
extern int  func1();
extern int  func2();
extern int  func3();
extern int  func4();
extern int  func5();
extern int  func6();
extern int  func7();

int main()
{
   func1();
   func2();
   func3();
   func4();
   func5();
   func6();
   func7();
   return 0;
}
```

Figure 52. stack.cpp: A program calling all the functions.

Let us see what happens when we compile and link this without using -qfuncsect. The script shown in Figure 53 is used to perform the test.

```
set -x
xlC -c *.cpp
ls -l *.o
xlC -w *.o
ls -l a.out
strip a.out
ls -l a.out

xlC -c -qfuncsect *.cpp
ls -l *.o
xlC -w *.o
ls -l a.out
strip a.out
ls -l a.out
```

Figure 53. testfuncsect: A script to test the -qfuncsect option

The result of running the first part of the test is shown in Figure 54. Loader warning messages about duplicate entries have been suppressed for clarity.

```
+ xlC -c func1.cpp func2.cpp func3.cpp func4.cpp func5.cpp func6.cpp
func7.cpp stack.cpp
func1.cpp:
func2.cpp:
func3.cpp:
func4.cpp:
func5.cpp:
func6.cpp:
func7.cpp:
stack.cpp:
+ ls -l func1.o func2.o func3.o func4.o func5.o func6.o func7.o stack.o
-rw-r--r--   1 root     project   157700 Mar 02 10:44 func1.o
-rw-r--r--   1 root     project   157700 Mar 02 10:44 func2.o
-rw-r--r--   1 root     project   157700 Mar 02 10:44 func3.o
-rw-r--r--   1 root     project   157700 Mar 02 10:44 func4.o
-rw-r--r--   1 root     project   157700 Mar 02 10:44 func5.o
-rw-r--r--   1 root     project   157700 Mar 02 10:44 func6.o
-rw-r--r--   1 root     project   157700 Mar 02 10:44 func7.o
-rw-r--r--   1 root     project     1266 Mar 02 10:44 stack.o
+ xlC -w func1.o func2.o func3.o func4.o func5.o func6.o func7.o stack.o
+ ls -l a.out
-rwxr-xr-x   1 root     project   435400 Mar 02 10:44 a.out
+ strip a.out
+ ls -l a.out
-rwxr-xr-x   1 root     project   281479 Mar 02 10:44 a.out
```

Figure 54. Not using the -qfuncsect option

The important things to remember here are:

- Each compiled function takes 157 700 bytes
- The executable takes 435 400 bytes unstripped
- The stripped executable takes 281 479 bytes

Figure 55 on page 205 shows the results of running the second part of the test.

```
+ xlC -c -qfuncsect func1.cpp func2.cpp func3.cpp func4.cpp func5.cpp
func6.cpp func7.cpp stack.cpp
func1.cpp:
func2.cpp:
func3.cpp:
func4.cpp:
func5.cpp:
func6.cpp:
func7.cpp:
stack.cpp:
+ ls -l func1.o func2.o func3.o func4.o func5.o func6.o func7.o stack.o
-rw-r--r--   1 root     project   200371 Mar 02 10:44 func1.o
-rw-r--r--   1 root     project   200371 Mar 02 10:44 func2.o
-rw-r--r--   1 root     project   200371 Mar 02 10:44 func3.o
-rw-r--r--   1 root     project   200371 Mar 02 10:44 func4.o
-rw-r--r--   1 root     project   200371 Mar 02 10:44 func5.o
-rw-r--r--   1 root     project   200371 Mar 02 10:44 func6.o
-rw-r--r--   1 root     project   200371 Mar 02 10:44 func7.o
-rw-r--r--   1 root     project     1266 Mar 02 10:44 stack.o
+ xlC -w func1.o func2.o func3.o func4.o func5.o func6.o func7.o stack.o
+ ls -l a.out
-rwxr-xr-x   1 root     project   239626 Mar 02 10:45 a.out
+ strip a.out
+ ls -l a.out
-rwxr-xr-x   1 root     project   115719 Mar 02 10:45 a.out
```

Figure 55. Using the -qfuncsect option

Using the -qfuncsect option instructs the compiler to generate object code where each function is contained with its own CSECT. This means the object files will be slightly larger since there will be extra formatting information. The benefit of using the option is that the linker can now discard the redundant routines. A comparison of the results is shown in Table 36.

Table 36. Advantages of using -qfuncsect

Component	Size	Size using -qfuncsect	Comments
Compiled function	157 700	200 371	Takes more space on disk because of additional information about different parts.
Executable	435 400	239 626	45% less spaced used by avoiding code duplication.

Component	Size	Size using -qfuncsect	Comments
Stripped executable	281 479	115 719	41% for stripped code.

Use of the -qfuncsect can result in smaller executables, which, in turn, reduces the amount of paging that must be done while the application is running. This, in turn, improves the performance of the application. Any improvement may not be noticeable to the end user once other factors, such as device I/O, are taken into consideration. However, the system as a whole will be performing less work for the same result.

6.7 Reworking a program to use multiple processors

A single CPU has often been a bottleneck in computers having more and more memory to handle as time goes by. AIX, Version 4.3 supports hardware with multiple-CPU configurations, which are becoming more and more commonplace and more and more affordable. Though reworking a program to use multiple processors is certainly not the first thing one would do to optimize a program, we shall discuss enough to show:

1. How surprisingly easy it can be.
2. Which gains can be expected.
3. Which gains should not be expected.

6.7.1 Know your system

Multiprocessor systems are becoming more and more common these days, but do we happen to have one? Using the sysconf() function, the program shown in Figure 56 on page 207 will tell us what we want to know.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <limits.h>

#define w(x) printf("%25.25s %ld\n", #x, sysconf(x));

main()
{
 w(_SC_CLK_TCK)                 /* # of clock ticks/second */
 w(_SC_VERSION)                 /* POSIX version & revision */

 w(_SC_CHILD_MAX)               /* Max # of children per process */

 w(_SC_THREADS)                 /* Are pthreads implemented ? */
 w(_SC_THREAD_PROCESS_SHARED)   /* pshared attribute supported ? */
 w(_SC_THREAD_THREADS_MAX)      /* Max # of threads per process */
 w(_SC_THREAD_STACK_MIN)        /* Minimum stack size for threads */
 w(_SC_THREAD_ATTR_STACKSIZE)   /* Can it be changed ? */

 w(_SC_NPROCESSORS_CONF)        /* Number of processors configured */
 w(_SC_NPROCESSORS_ONLN)        /* Number of processors online */
}
```

Figure 56. Fast inquiry about the system with sysconf()

The results of running the program are shown in Figure 57.

```
              _SC_CLK_TCK 100
               _SC_VERSION 199506
             _SC_CHILD_MAX 262144
               _SC_THREADS 1
_SC_THREAD_PROCESS_SHARED 1
   _SC_THREAD_THREADS_MAX 32767
     _SC_THREAD_STACK_MIN 8192
_SC_THREAD_ATTR_STACKSIZE 1
      _SC_NPROCESSORS_CONF 12
      _SC_NPROCESSORS_ONLN 12
```

Figure 57. The sysconf() results

Our machine has 12 processors, supports threads, and each thread will reserve at least 8 K (that is, 2 pages) for its stack operations.

Note

The number of processes per user (forks, not threads) is limited to 40, by default, to prevent a rogue user from swamping the system. This limitation does not apply to the root user, who can also change the limit for other users using the SMIT command, and selecting the option: `Change / Show Characteristics of Operating System`.

6.7.2 Know your program

If our program is already coded, we assume we know where its bottlenecks lie thanks to profiling as described in Section 6.5, “Profiling your programs” on page 171.

If we consider threading at design time, a better approach, but only if we already have acquired some experience in threading, we assume we have an idea about where these bottlenecks could be.

6.7.3 The gentle art of threading

How are we going to have our C matrix computed by many processors at the same time? A simple way is to have each line of the matrix computed separately, and let the system allocate processors to compute each line as they become available. This design is shown in Figure 58 on page 209.

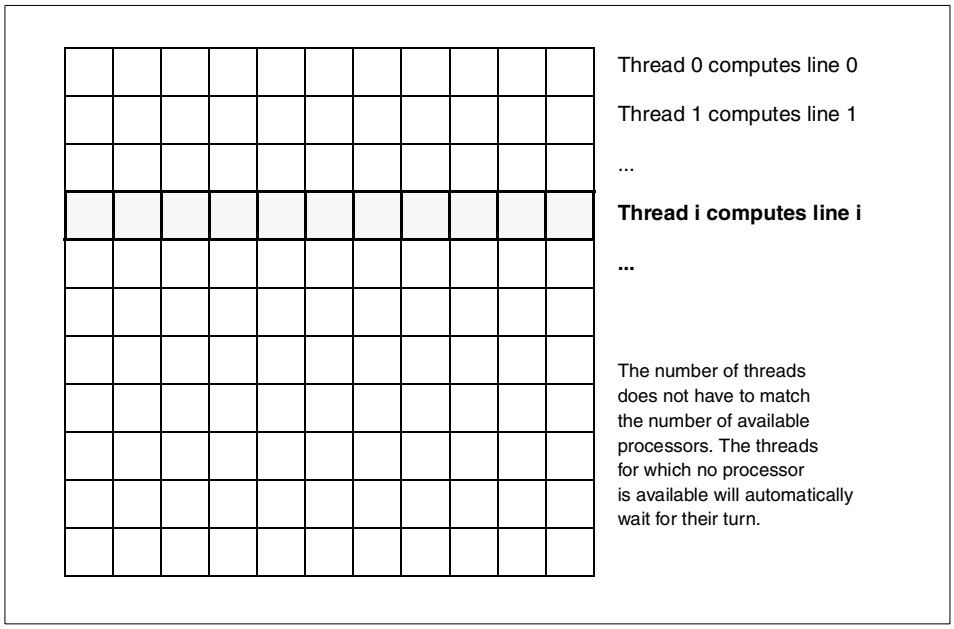


Figure 58. Computing the matrix with many threads

Now, what happens if we have more lines in the matrix than processors in the system? Nothing very tragic: The operating system ensures that every thread will wait quietly for a processor available to process it, even if we have 500 threads. Remember that a thread is a *lightweight* process. Having over 1000 threads is quite possible if we have the care to create them with well-tailored options rather than the default ones.

Of course, this may not be the most efficient practice, but the art of programming is also to compromise between “the three great virtues of a programmer: laziness, impatience, and [pride]", according to Larry Wall, the creator of PERL (he uses the term of *hubris* instead of pride).

If the expected ratio between the number of threads and the number of processors happened to be very high, say 5000 threads per processor, we would consider using a *pool of threads*. The basic idea of a pool of threads is to manage a queue so that the number of running threads has the same order of magnitude as the number of processors available.

Now, let us thread the matrix multiplication of matrix.c.

6.7.3.1 What we start with

The multiplication consists of the three imbricated loops shown in Figure 59.

```
for (i=0; i<SIZE; ++i)
 for (j=0; j<SIZE; ++j)
     { C[i][j] = 0;
       for (k=0; k<SIZE; ++k) C[i][j] += A[i][k]*B[k][j];
     }
```

Figure 59. Core computation

We want to replace it with something like:

```
for (i=0; i<SIZE; ++i) compute_line(i); /* but as a thread */
```

Naturally enough, we can expect `compute_line(i)` to be something like that shown in Figure 60.

```
compute_line(i)
{  int j, k;
   for (j=0; j<SIZE; ++j)
      {  C[i][j] = 0;
         for (k=0; k<SIZE; ++k) C[i][j] += A[i][k]*B[k][j];
      }
}
```

Figure 60. Computing one line of the result matrix

No worry about passing parameters. The i variable (matrix line to compute) is the only thing needed by compute_line() because a threaded procedure already has access to all the global variables defined in its process.

Except, of course, the global variable i which is masked by the parameter i, although normal good coding style would avoid this conflict.

For the same reason, we *must* declare j and k as local variables. If they were global, our computation would be a total mess.

All global variables in the process are accessible to all threads. This is what makes them *lightweight*, but is also what makes them *dangerous*. Hence, the following rules for making threads safe are:

- Any work variable used by a thread should be declared local to that thread.
- When a thread writes to a global variable, you should either make it *safe by design* (it is the case here: Each thread writes in, so to say, its own territory, and no threads can interfere if our design is good) or make it *safe by program* using mutexes.

- When many threads can write to the same variable, remember to declare it as *volatile*. That will tell the compiler that if a variable content was loaded in a register some lines before and has to be used again, the value of the register is *not* reliable and should be reloaded.

6.7.3.2 Calling compute_line() as a thread

Now, we just have to specify that compute_line() should be started as a thread:

```
for (i=0; i<SIZE; ++i) pthread_create (&id[i], NULL, compute_line, i);
```

The variable *id* is an array of type *pthread_t* where the thread identifying numbers will be stored so that we can reference them later in order to wait for their completion before using their result.

6.7.3.3 Declaring what is needed

First, as we use threads, the following line should be inserted *as the first line* in the program:

```
#include <pthread.h>
```

We also need to store the thread IDs in a table:

```
pthread_t id[SIZE];
```

6.7.3.4 Waiting for completion of all threads

Here is the loop to wait for completion of all the threads:

```
for (i=0; i<SIZE; ++i) pthread_join (id[i], NULL);
```

Note that this loop is *not* acknowledging the completion of the threads in the order in which these completions occur, but we just do not care. In this case, whichever the order the threads complete, we cannot use the matrix as long as all the threads are not terminated; so, the order in which we wait for them is irrelevant.

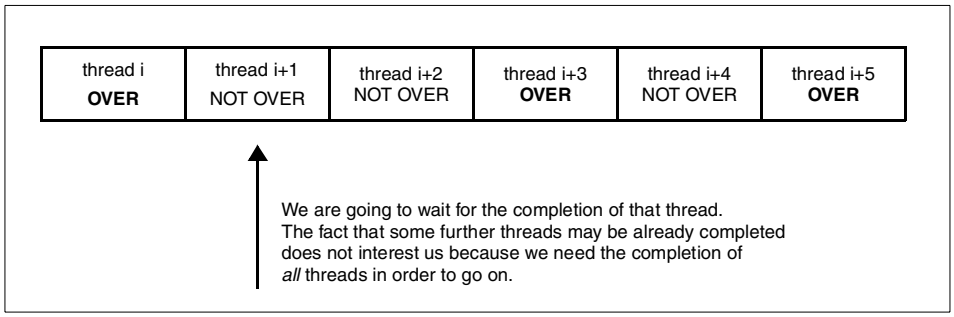


Figure 61. We have to wait for the completion of all threads

Please note also that any computation not using the result matrix can be inserted between the loop launching the threads and the loop waiting for their completion. We shall then have a *working boss* program rather than a *lazy boss* program. Refer to Section 5.1.6, “Software models” on page 127 for a comparison of the different program designs for using threads.

6.7.3.5 Testing the return codes

We shall add a return code test in the thread-launching part, as shown in Figure 62 on page 212, just to be sure everything went well. Otherwise, we shall abort the program and display information detailing where it failed in the hope it will help us in understanding why. Should it occur, that will probably mean there is no more space available in the process address space.

This shortage could be overcome by specifying a *-bmaxdata=0x80000000* at compilation time, but it will be wiser, in such a case, to design the program in a way that creates less threads for the same work. Refer to Section 2.3, “Process private data” on page 43 for information on the use of the -bmaxdata option.

```
for (i=0; i<SIZE; ++i)
{
    rc=pthread_create (&id[i], NULL, compute_line, i);
     if (rc) { printf ("For line [%d][*] ",i);
               perror ("Unable to create thread");
               exit (28);
             }
}
```

Figure 62. Testing the return code at thread creation time

We can also test the result of the pthread_join operation, as shown in Figure 63, though there is much less risk of failure here.

```
  for (i=0; i<SIZE; ++i)
        {
          rc=pthread_join (id[i], NULL);
          if (rc) { printf ("thread %d could not be joined.\n", i);
                    exit (28);
                  }
        }
```

Figure 63. Testing the return code a thread completion time

6.7.4 Our final program

Here is the mathread3.c program obtained by applying the preceding modifications to matrix.c:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include "param.h"

 double A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE];
 pthread_t id[SIZE];
 int garbage;

/* This reentrant program computes the whole line C[i][*]      */

void compute_line(int i)
{float s;
int j, k;
 for (j=0; j<SIZE; ++j)
 { s=0.;
   for (k=0; k<SIZE; ++k) s+=A[i][k]*B[k][j];
   C[i][j]=s;
 }
}
main(int argc, char * argv[])
{long i;
 int rc;

  /* create matrices A and B */

   for (i=0; i<SIZE; ++i)
```

```
   for (j=0; j<SIZE; ++j) { A[i][j]=i+j; B[i][j]=i-j; }

  /* Matrix product */

   start_timing();

   for (i=0; i<SIZE; ++i)
       {
     rc=pthread_create (&id[i], NULL, compute_line, i);
     if (rc) { printf ("For line [%d][*] ",i);
          perror ("Unable to create thread");
          exit (28);
        }
       }

  /* Let us wait for the computation to be finished */

  for (i=0; i<SIZE; ++i)
       {
    pthread_join (id[i], NULL);
       }
 /* The computation is over */

 stop_timing();

/* We are not interested at the result, but the optimizer should
   not know it !
*/

garbage = open ("junk", O_RDWR|O_CREAT);
write (garbage, C, sizeof C);

}
```

6.7.5 A good example

Let us see the execution time of this program for 500x500 matrices:

```
cc_r mathread3.c -DSIZE=500 && a.out
2114006 microseconds
```

While that may not appear spectacular at the first look, this program is 11.43 times faster (2.11 seconds of execution instead of 24.16 seconds) than matrix.c as timed in Section 6.3, “Timing a typical program” on page 168. As our machine has 12 processors, the increase in speed is not very far from the ideal.

6.7.6 A bad example

What happens, however, if we have to multiply small matrices, typically not 500x500, but 4x4? OpenGL uses a lot of 4x4 matrices as, in fact, do most programs dealing with geometric information:

```
cc_r matrix.c -DSIZE=4 && a.out
13 microseconds
cc_r mathread3.c -DSIZE=4 && a.out
1411 microseconds
```

The message is clear: The computation of the result is so fast in such a case that the overhead of creating four threads, however small, is bigger than the 13 microseconds needed by the whole serial computation. We shall measure by how much in Section 6.8, “Threads versus forks” on page 216, but we already guess it will be around 1400/4 = 350 microseconds.

So, using threads is not a good choice in that case.

Note also that here we had no chance of ever using the 12 available processors anyway. At most, four of them will be used since four user threads will be created, one per line of the matrix.

6.7.7 Deciding when to use threads

You can see why having 1000 threads in the same program is not much of a problem: Not only do threads use a very small amount of memory (each time a thread is created, it gets assigned just an 8 KB stack area within the process), but creating 1000 of them will typically add only 350 milliseconds, or 0.35 seconds, to program execution time. This speed of creation make threads especially attractive for all kinds of telecommunications or client/server applications, too. Even if we have 10,000 new clients per minute, we can create one thread to serve each client without penalizing the system.

Of course, if you are running on a *uniprocessor* machine, do not expect any performance gain by using the threaded solution, as the program is clearly CPU-bound. But, on the other hand, your very same a.out file will be able to use multiple processors on another machine without any recompilation, which can be interesting.

To summarize things:

1. It is a good idea to know the order of magnitude of thread creation time on your machine.

2. It is also a good idea to know the order of magnitude of your computation times. 350 microseconds per thread suggest that a typical thread should do at least 3.5 ms of computation, as a 10% overhead stays acceptable.
3. When threads are concerned, a good solution at a given scale can become not-so-good (one would say *catastrophic* in this example) when scaled differently.

Note

You may have to choose between an optimal solution, which is not scalable, and a non-optimal, but scalable, solution. An alternative is to invest in some extra programming effort so that your program can determine how it should behave at run-time.

So much for the myth according to which the multiplication of CPUs would be the solution to every one of yesterday's problems.

6.8 Threads versus forks

Knowing the typical overheads of a fork() or a pthread_create() can be interesting. Two programs, forks.c shown in Figure 64 on page 217 and threads.c shown in Figure 65 on page 218, will create SIZE forks or threads. These forks and threads do not do anything. We are just, once more, trying to measure a kind of empty shell.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include "param.h"

main()
{
 int i;
 pid_t rc;

 start_timing();

 for (i=0; i<SIZE; ++i)
     { rc = fork();
       /* Did the fork() succeed ? Otherwise get out */
       if (rc<0) { printf("No more processes available.\n");
         exit (28);
       }

       /* Let the child sleep after a symbolic work
          which will not be timed anyway */
       if (rc == 0) { printf("Child %d created\n", getpid());
                 sleep(5);
            printf ("Child %d has finished\n", getpid());
            exit(0);
          }

       /* If we are the parent, do nothing */
     }

 /* No child will ever arrive here because of the exit(0).
    So we are timing the parent, and only the parent. */

 stop_timing(); printf(" for %4d forks.\n", SIZE);
}
```

Figure 64. A program to time fork()

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include "param.h"

void mythread(int i)
{
       /* Let the thread sleep after a symbolic work
          which will not be timed anyway */

       printf("Thread %d created\n", i);
       sleep(5);
       printf ("Thread %d finished\n", i);
       pthread_exit(0);
}

main()
{
 int i, rc;
 pthread_t id[SIZE];

 start_timing();

 for (i=0; i<SIZE; ++i)
     { rc = pthread_create (&id[i], NULL, mythread, i);
       /* Was the thread created ? Otherwise get out */
       if (rc) { printf("Unable to create thread %d.\n", i);
            exit (28);
          }
     }

 /* No thread worker will ever arrive here because of the exit(0).
    So we are timing the boss, and only the boss. */

 stop_timing(); printf (" for %4d threads.\n", SIZE);

 for (i=0; i<SIZE; ++i) pthread_join(id[i], NULL);
}
```

Figure 65. A program to time pthread_create()

We just have to compare the execution times of the two programs for different values of SIZE with the script shown in Figure 66 on page 219.

```
rm results/threadsforks.txt
for size in 10 20 50 100 200 500 1000
do
  echo "Testing size $size"
  cc -w -DSIZE=$size forks.c -o forks
  forks >> results/threadsforks.txt
  sleep 7
  cc_r -w -DSIZE=$size threads.c -o threads
  threads >>results/threadsforks.txt
  sleep 7
  done
```

Figure 66. A script to explore the effect of SIZE

Figure 67 shows the results of tuning the threadsforks script as shown in Figure 66.

```
# threadsforks
Testing size 10
Testing size 20
Testing size 50
Testing size 100
Testing size 200
Testing size 500
Testing size 1000
# grep micro results/threadsforks.txt
           7263 microseconds for   10 forks.
           2996 microseconds for   10 threads.
          14749 microseconds for   20 forks.
           5856 microseconds for   20 threads.
          38940 microseconds for   50 forks.
          14952 microseconds for   50 threads.
          79872 microseconds for  100 forks.
          29344 microseconds for  100 threads.
         164336 microseconds for  200 forks.
          59781 microseconds for  200 threads.
         447253 microseconds for  500 forks.
         146688 microseconds for  500 threads.
        1167343 microseconds for 1000 forks.
         293043 microseconds for 1000 threads.
```

Figure 67. Running the threadsforks script

Figure 68 gives a better understanding of these results. The generic term *subprocess* is used to designate a forked child or a thread.

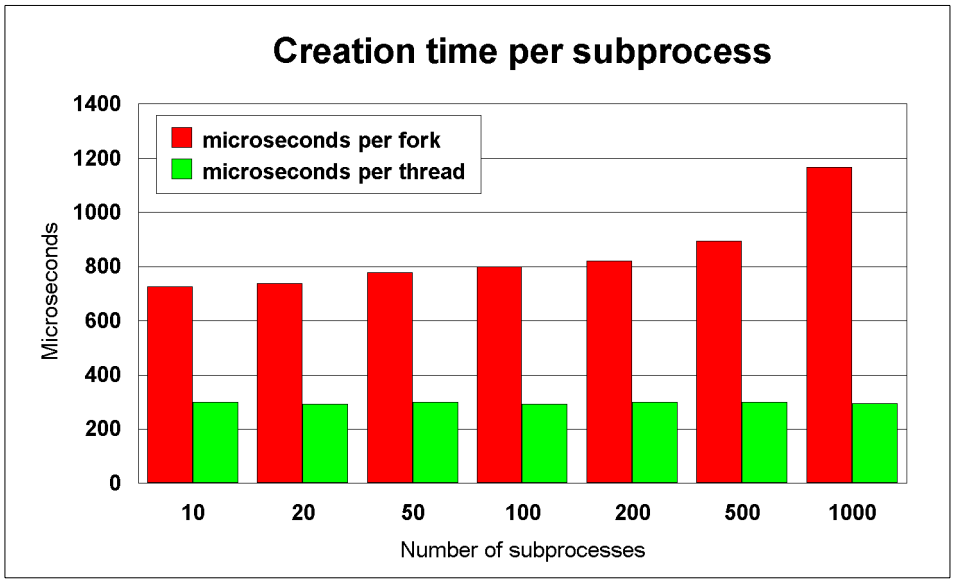


Figure 68. Creation times for threads versus forks

Notice that creating forks is twice slower than creating threads, and things tend to get worse as the number of created subprocesses increase, while thread creation time remains quite stable, around 300 microseconds.

Consider also that threads can work on common data without having to use shared memory, pipes, or some other form of IPC. The most basic C programmer is able to use threads within minutes.

6.8.1 Putting it all together

What happens now if we apply optimization options to our former threaded program, mathread3.c? The results are shown in Table 37.

Table 37. Executing mathread3.c with different optimization options

options	microseconds	real (seconds)	user	sys
(none)	1 798 492	1.95	21.41	0.13
-O	402 348	0.52	4.63	0.11
-O2	395 448	0.52	4.51	0.12

options	microseconds	real (seconds)	user	sys
-O3 -qstrict	341 243	0.45	3.90	0.13
-O3	295 905	0.40	3.31	0.11
-O4	308 554	0.43	3.53	0.12

The execution time for calculating the matrix using the threaded code is reduced from 24 seconds to 0.43 second when the code is optimized. In the preceding sections, we squeezed the elapsed time of our computation by a factor of 55. We can, therefore, expect that jobs formerly taking hours will be reduced considerable by combining the two tools of multithreading and compiler optimization options. Figure 69 graphically illustrates the reduction in execution time.

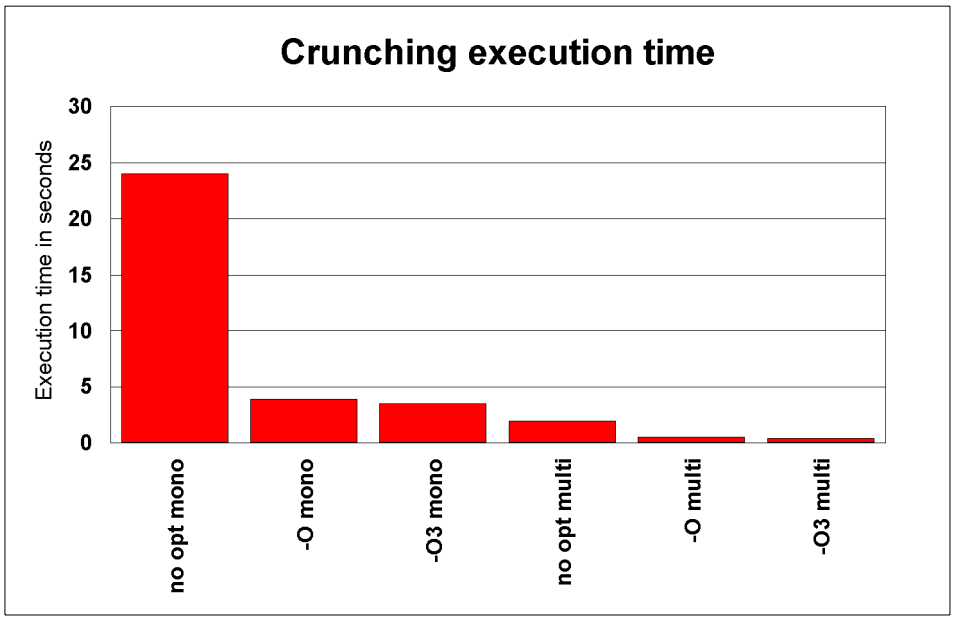


Figure 69. Our progress so far

We have now seen the best. What follows may supply only marginal gains, comparatively, but is, nevertheless, worth being mentioned.

6.8.2 The effects of scope and M:N ratio

Let us continue with the good threading case seen in Section 6.7.5, “A good example” on page 214, and see, without recompiling the executable, how the

run-time parameters, AIXTHREAD_SCOPE and AIXTHREAD_RATIO, affect execution times. These parameters and the M:N ratio are described in detail in Section 5.1.2, “Lightweight process -LWP” on page 109. The script shown in Figure 70 will be used to gather results.

```
#! /bin/ksh
# Run a.out with different THREAD options

export AIXTHREAD_SCOPE=P
for ratio in "1:1" "2:1" "4:1" "8:1" "12:1" "16:1" "12:2" "12:3" "12:4"
do
export AIXTHREAD_MNRATIO=$ratio
a.out
echo " for scope $AIXTHREAD_SCOPE mnratio = $AIXTHREAD_MNRATIO"
done

unset AIXTHREAD_MNRATIO
export AIXTHREAD_SCOPE=S
a.out
echo " for scope S"
```

Figure 70. A script to test various M:N ratios together with the scope

Running this script with the matrix.c program gives the results shown in Figure 71.

```
matrix compiled with option -DSIZE=500

      24164807 microseconds for scope P mnratio = 1:1
      24053571 microseconds for scope P mnratio = 2:1
      24246108 microseconds for scope P mnratio = 4:1
      24083303 microseconds for scope P mnratio = 8:1
      24029279 microseconds for scope P mnratio = 12:1
      24256506 microseconds for scope P mnratio = 16:1

       24227765 microseconds for scope S
```

Figure 71. Effect of M:N ratio when running the matrix.c program

The result is quite normal. The program, matrix.c, remember, does not use multithreading. However, when we run mathread3.c, we get the results shown in Figure 72 on page 223.

```
mathread3 compiled with option -DSIZE=500
        2114006 microseconds for scope P mnratio = 1:1
        2168204 microseconds for scope P mnratio = 2:1
        2121688 microseconds for scope P mnratio = 4:1
        2130196 microseconds for scope P mnratio = 8:1
        2112072 microseconds for scope P mnratio = 12:1
        2131732 microseconds for scope P mnratio = 16:1

        2110639 microseconds for scope S
```

Figure 72. Influence of M:N ratio on mathread3.c

What is happening here? The execution times for the different ratios do not seem to differ at all.

The reason is that *our threads are so fast in executing that they terminate before reaching the end of their time slice and, therefore, never get a chance to be handled by the scheduler after having been launched*. Of course, in these conditions, the choice of the scheduling policy will not change anything.

Rather than writing a more complicated floating-point program (our point is program optimization, not numerical analysis), we are going to cheat a little, and execute each computation 10000 times so that our threads will last longer.

Therefore, we make the changes highlighted in Figure 73 to the compute_line function, which is the routine being run by each thread.

```
void compute_line(int i)
{int j, k, l;
 for (l=0; l<10000; ++l)
 for (j=0; j<SIZE; ++j)
 { s=0.;
   for (k=0; k<SIZE; ++k) s+=A[i][k]*B[k][j];
   C[i][j]=s;
 }
}
```

Figure 73. Slow compute_line()

We shall not use the -O type optimizations here because the optimizer could move the computations out of the loop, and then discover that the loop is empty and remove it. The reason for this example is to show the impact of the M:N ratio on the execution time, rather than trying to reduce the execution time. The SIZE macro is reduced to 20, and the program recompiled. The results are shown in Figure 74 on page 224.

```
        15668986 microseconds for scope P mnratio = 1:1
        15509688 microseconds for scope P mnratio = 2:1
        15752484 microseconds for scope P mnratio = 4:1
        26389433 microseconds for scope P mnratio = 8:1
        36945916 microseconds for scope P mnratio = 12:1
        46210871 microseconds for scope P mnratio = 16:1
        20522753 microseconds for scope P mnratio = 12:2
        15974403 microseconds for scope P mnratio = 12:3
        15413396 microseconds for scope P mnratio = 12:4

        15518422 microseconds for scope S
```

Figure 74. M:N ratio influence on execution time for the modified program

Notice that:

- Elapsed time increases when the number of user threads per system thread is increased.
- Specifying 2:1, or even 4:1, gives better results for this example than the standard AIXTHREAD_RATIO of 8:1.
- Not surprisingly:
  - The results for 12:2 (we could have written it 6:1) fit somewhere between the results for 4:1 and 8:1.
  - The results for 12:3 are similar to those for 4:1.
  - The result when scope=S is one of the best.
- For scope=P, 12:4 (that is, 3:1) seems the best compromise here.

Figure 75 on page 225 shows the impact of the M:N ration on the execution time of this application.

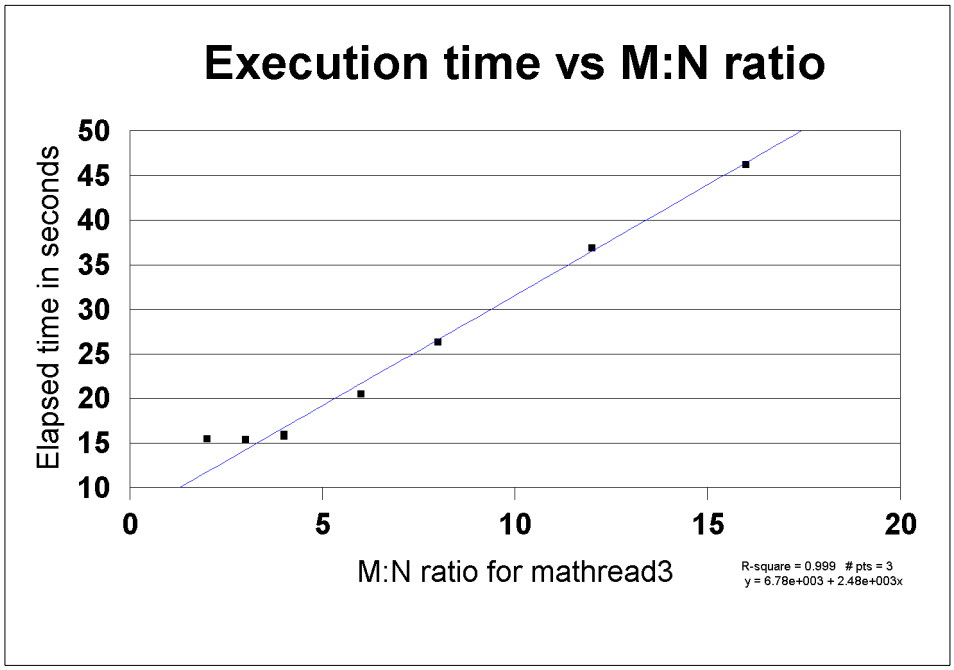


Figure 75. Execution time against M:N ratio for the new program

6.9 Malloc multiheap

By default, the malloc subsystem uses a single heap or free memory pool. Starting with AIX Version, 4.3.3, the malloc routine supports an optional multiheap capability to allow applications to enable the use of multiple heaps of free memory, rather than just one.

The purpose of providing multiple heap capability in the malloc subsystem is to improve the performance of threaded applications running on multiprocessor systems. When the malloc subsystem is limited to using a single heap, simultaneous memory allocation requests received from threads running on separate processors are serialized, meaning that the malloc subsystem can only service one thread at a time. This can have a serious impact on application performance.

With the multiheap capability enabled, the malloc subsystem creates a fixed number of heaps for its use. Each memory allocation request is serviced using one of the available heaps. The malloc subsystem can then process memory allocation requests in parallel as long as the number of threads

simultaneously requesting service is less than or equal to the number of heaps.

6.9.1 Using malloc multiheap

The simplest way of using the malloc multiheap feature is to set the following environment variable:

```
MALLOCMULTIHEAP=true
```

This enables the feature with the default configuration of 32 memory pools. Let us build a program using a lot of threads that make a lot of calls to malloc, then run it using the standard malloc, and then with malloc multiheap enabled. The program does not do anything else other than call malloc, and that is why we can expect severe competition between all the active threads to gain access to the heap. This means the malloc routine will be the bottleneck for all the threads. The program source code is shown in Figure 76 on page 227. Multithreaded C++ programs will potentially also have a large benefit from using the malloc multiheap feature since each the heap must be accessed each time a constructor or destructor is called.

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include "param.h"

void mythread(void)
{ int i;
  struct numberlist
     { double x;
       struct numberlist * next;} *p;
  p = malloc (sizeof(struct numberlist));
  for (i=1; i<10000; ++i)
      { p->x=0.;
    p->next = malloc (sizeof(struct numberlist));
    p=p->next;
    if (!p) { printf("Unable to allocate for i = %d\n.", i);
                  exit (28);
       }
      }
}

main()
{ int i, rc;
  pthread_t id[100];
  start_timing();

  for (i=0; i<100; ++i)
      { rc = pthread_create (&id[i], NULL, mythread, NULL);
       if (rc) { printf("Unable to create thread for i = %d.\n", i);
         exit (28);
       }
      }
/* Let us wait for all those threads to finish */
  for (i=0; i<100; ++i)
       { rc = pthread_join (id[i]);
        if (rc) { printf ("Unable to join thread %d.\n", i);
        exit(28);
       }
  stop_timing();
}
```

Figure 76. A program making a lot of calls to malloc

The results of running this program using the regular malloc routine, and then with the malloc multiheap feature enabled, are as follows:

```
$ time a.out
        8894998 microseconds
real    0m8.96s
user    0m13.65s
sys     1m26.41s
$ export MALLOCMULTIHEAP=true
$ time a.out
         392056 microseconds
real    0m0.45s
user    0m1.59s
```

```
sys     0m2.75s
```

This is a 20:1 improvement in speed. Any comment would be superfluous here.

6.9.2 Parameters of malloc multiheap

The malloc multiheap feature also offers tuning parameters to alter the number of heaps from the default of 32 and alter the algorithm to select the heap to be used.

6.9.2.1 The number of heaps

If it is enabled, the malloc multiheap feature uses 32 heaps by default. If you know you will not use as many processors, or for any other reason, you can ask for any lower number of heaps. Instead of setting the environment variable, MALLOCMULTIHEAP, to a value of true, it is set to a value of heaps:*n*, when *n* is the number of heaps that are desired. As we are fond of measurements, let us see how the number of heaps would affect the elapsed computation time using the example program shown in Figure 76 on page 227.

The following command sequence was used to determine the impact of the number of heaps:

```
unset MALLOCMULTIHEAP
$ for i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
26 27 28 29 30 31 32
> do
> export MALLOCMULTIHEAP=heaps:$i
> time a.out
> done
```

Since the system used for testing has 12 processors, we guess the interesting part will be between 1 and 12 heaps, and the result will not change very much with higher heap numbers.

Figure 77 on page 229 shows the results of running the program with different numbers of malloc heaps.

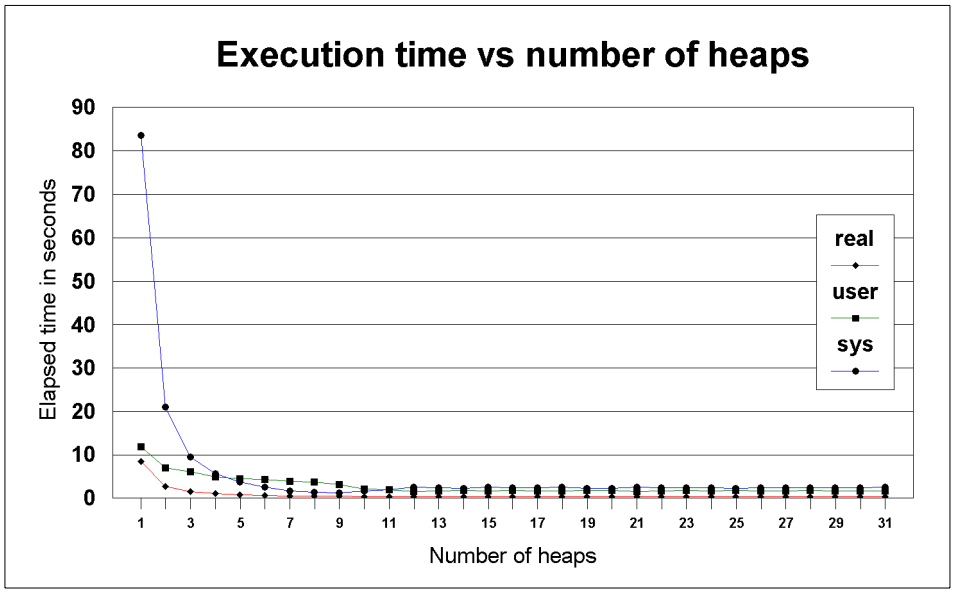


Figure 77. malloc.c: Execution time versus number of heaps

We see that the system time for the single heap version is absolutely huge: 86 seconds for nine seconds of elapsed time, which is on the average seven seconds per processor. Also, the acceleration due to multiprocessing is abnormally small:

```
user time/real time = 13.65/8.96 = 1.52
```

compared to the 11.43 observed in Section 6.7.5, “A good example” on page 214”. Both this system overhead and this poor usage of each processor have the same origin: Processes compete for a bottleneck, which is memory allocation in a unique pool thread, because only one processor can have access to the allocation section at a time; so, one processor is served while the others wait.

Having two heaps already reduces the system times fourfold here, and this system time continues to diminish as the number of heaps increases. Figure 78 on page 230 concentrates on what happens in the range of three to 16 heaps.

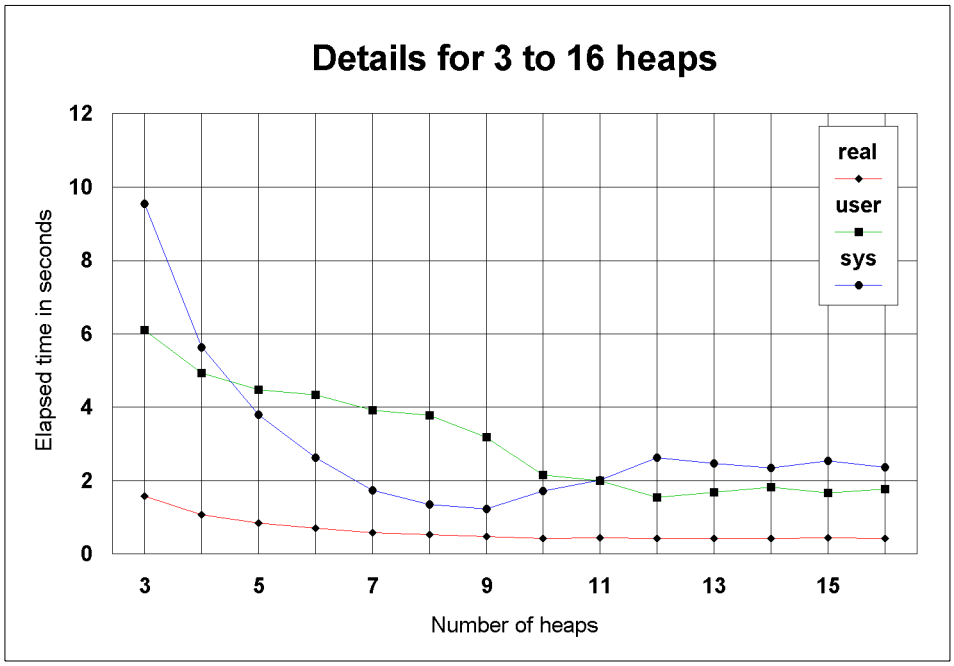


Figure 78. malloc.c: Details from 3 to 16 heaps

We see the system time diminishing up to nine heaps, and then rising again, probably because the overhead of handling more heaps overcomes any waiting overhead that could remain. But, both the user time and the real time, our precious human wait-time that costs much more than machine time, go on diminishing until the number of heaps equal the number of processors.

After that, trends for real and system time stabilize, and only slight random fluctuations remain.

Two interesting variables can be introduced here. The first is the user/real ratio, which is the *degree of parallelism* of our program, and the second the user/sys ratio, which is related to a kind of *good behavior*: Running a job without penalizing the system.

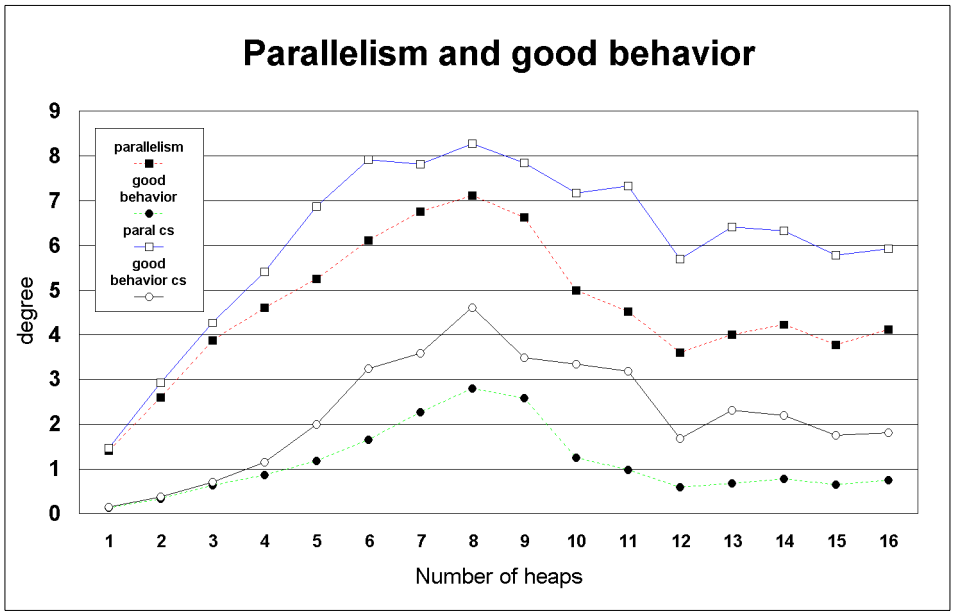


Figure 79. Better parallelism and better behavior with considersize

Figure 79 shows the degrees of parallelism and good behavior as related to the number of heaps (black markers). The white markers ("cs" in the legend) are results obtained using a considersize option that is described in Section 6.9.2.2, "The considersize option" on page 231. Please note that using this option enhances, in our example, both the degrees of parallelism and of good behavior. We are going to see, however, that it is at some extra cost.

Why does the usage of system resources increase? Because the heaps are allocated in a round-robin way. This means all the heaps we are asking for will be used, whether we really need them or not. A more subtle (though more time-consuming) way to allocate space would be to use the *first available* heap instead of the *next* one. The *considersize* option is going to allow us just that.

6.9.2.2 The considersize option

By default, malloc multiheap selects a new, available heap every time a request is made, essentially using round-robin selection. The considersize option will select, instead, the first available heap that has enough free space to handle the request. While somewhat slower in computation time, this option can help reduce both the working set size and the number of sbrk() calls. The considersize option is specified when setting the

MALLOCMULTIHEAP environment variable along with the number of required heaps as follows:

```
MALLOCMULTIHEAP=heaps:4,considersize
```

Table 38 displays the timings of the sample malloc program when changing the number of heaps is used both with and without the considersize option.

Table 38. Impact of the considersize option

Number of heaps	No considersize			Considersize		
	real	user	sys	real	user	sys
1	8.41	11.83	83.54	8.76	12.91	85.32
2	2.71	7.03	21.00	3.00	8.78	22.82
3	1.58	6.11	9.55	1.82	7.77	10.92
4	1.07	4.93	5.65	1.24	6.72	5.80
5	0.85	4.47	3.80	1.01	6.94	3.47
6	0.71	4.34	2.63	0.82	6.50	2.01
7	0.58	3.92	1.73	0.67	5.24	1.46
8	0.53	3.77	1.35	0.58	4.80	1.04
9	0.48	3.15	1.23	0.47	3.69	1.06
10	0.43	2.15	1.72	0.41	2.94	0.88
11	0.44	1.99	2.02	0.36	2.64	0.83
12	0.43	1.55	2.63	0.34	1.94	1.15
13	0.42	1.68	2.46	0.33	2.12	0.92
14	0.43	1.82	2.34	0.33	2.09	0.95
15	0.44	1.66	2.53	0.33	1.91	1.09
16	0.43	1.77	2.36	0.32	1.90	1.05

For a malloc intensive program like our example, the *considersize* options gives clearly better elapsed time results for a large number of heaps (here, for nine and above) at the cost of an increased user time between seven percent and 54 percent.

The elapsed time, where better, is better by two percent (9 heaps) to 34 percent (16 heaps).

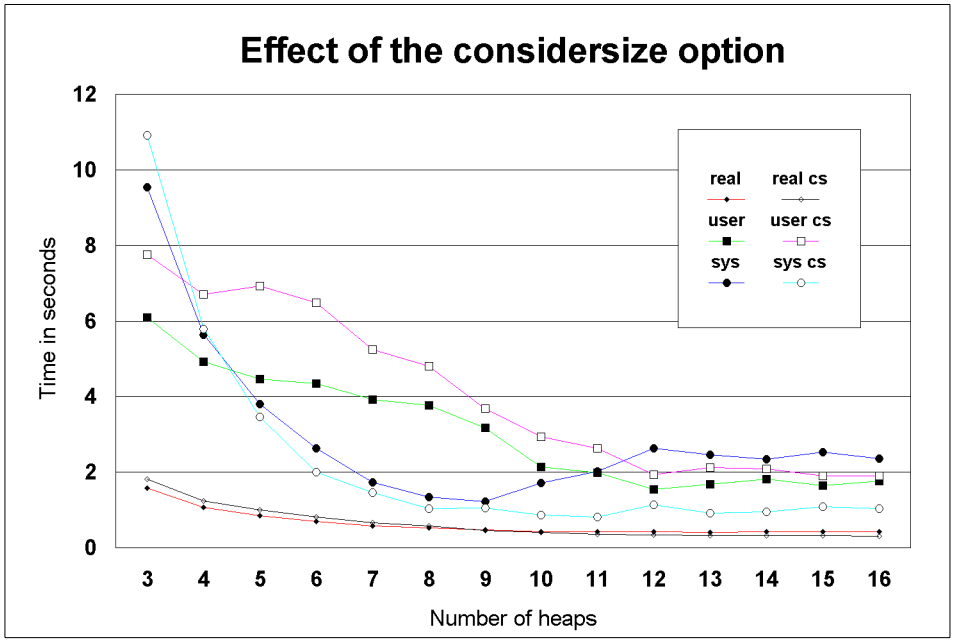


Figure 80. The considersize option enhance data locality

Figure 80 shows that the *considersize* option has indeed a cost when the number of heaps is too small. The additional computations involved make them even more constrained, but, on the other hand, the real time and system overhead are reduced when more heaps are allocated than the optimal level.

As the curve is very asymmetric, overestimating the number of heaps is less penalizing than underestimating it. A reasonable choice is to set the number of heaps equal to the number of processors and use the *considersize* option to be sure we are not using what we do not need.

6.10 The stride effect

Whenever possible, use your data in a sequential manner. In that way:

1. You enhance the number of probable cache hits.
2. You reduce your risks of having to wait for a missing page.

Let us look at the sensitivity of a simple matrix initialization to the stride effect. The sample programs to demonstrate the effect are shown in Figure 81 on page 234 for row major array access and Figure 82 on page 235 for column major array access.

```
#include <stdio.h>
#include <fcntl.h>
#include "param.h"

#define SIZE 1000

long a[SIZE][SIZE];
long i, j, garbage;

main()
{
 start_timing();

 for (i=0; i<SIZE; ++i)
 for (j=0; j<SIZE; ++j)
  a[i][j] = i+j;

 stop_timing();

garbage = open ("junk", O_RDWR+O_CREAT);
write (garbage, a, sizeof a);
}
```

Figure 81. stride.c: accessing a matrix in row-wise order (row by row)

```
#include <stdio.h>
#include <fcntl.h>
#include "param.h"

#define SIZE 1000

long a[SIZE][SIZE];
long i, j, garbage;

main()
{

 start_timing();

 for (j=0; j<=SIZE; ++j)
 for (i=0; i<SIZE; ++i)
  a[i][j] = i+j;

 stop_timing();

 garbage = open ("junk", O_RDWR | O_CREAT);
 write (garbage, a, sizeof a);
}
```

Figure 82. stride2.c: accessing a matrix column-wise order (column by column)

Although the two sample programs look rather similar, their execution times are not:

```
$ time stride
          36599 microseconds
real    0m0.09s
user    0m0.02s
sys     0m0.07s
$ time stride2
         179379 microseconds
real    0m0.24s
user    0m0.16s
sys     0m0.07s
```

To avoid the stride effect, it is not uncommon for numerical analysts to store their matrices into 64x64 submatrices. Multiplying matrixes is equivalent to multiplying their submatrices as one would do with scalars, and for each submatrix multiplication, both operands and result fit in the cache of each

processor, not only giving excellent results, but avoiding any type of cache interference between processors.

6.10.1 An counterintuitive consequence

Let us consider a classical programming problem: The sieve or Eratosthenes, which is used to determine a list of prime numbers. The principle is to cross out all multiples of every prime number, and whatever has not been crossed out at the end of the process is, of course, a prime number.

Now, how should we represent the numbers to be crossed out? With an array of int or with an array of char?

An idea coming from the dark ages of computers (the days before virtual memory was generalized) was that using data aligned on word boundaries was the correct way to make programs faster. This is still true within any active page. But, because of the stride effect, it ceases to be true for large arrays; in that case, choosing word-aligned data can reduce the cache hit as well as increase the probability of swapping pages in and out of memory.

The first version of a sieve program is shown in Figure 83.

```
#include <stdio.h>
#include <stdlib.h>

int k, crible, status[SIZE], primes[SIZE], a[SIZE*SIZE];

get_next_to_crossout()
{while (a[++k])  /* printf("%8ld is not prime.\n", k) */
          ; /* Examine the following integer as divisor */

 /* printf("Selecting %ld as the next prime number.\n", k); */
}

main()
{
 k=2;
 do {
      for (crible=k+k; crible<SIZE*SIZE; crible+=k) a[crible]=1;
      get_next_to_crossout();
    } while (k < SIZE);

 /* print out the prime numbers */

 for (k=2; k<SIZE*SIZE; ++k)
     if (!a[k]) /* printf("%9ld ", k) */ ;

/* for (k=1; k<SIZE; ++k) printf("%5d %5d\n", status[k], a[k]); */
}
```

Figure 83. sieve1.c: The sieve of Eratosthenes

Table 39 shows the results of executing the sieve program with different values of SIZE.

Table 39. sieve1.c: Execution times

SIZE	# of primes	seconds
100	10 000	0.04
200	40 000	0.04
350	122 500	0.06
500	250 000	0.10
1000	1 000 000	0.36
2000	4 000 000	2.35
3500	12 250 000	8.95
5000	25 000 000	19.86

Now, let us replace *int a[SIZE][SIZE]* by *char a[SIZE][SIZE]*, thus, giving program *sieve2.c*. The execution times are reduced as shown in Table 40.

Table 40. sieve2.c: Execution times

SIZE	# of primes	seconds
100	10 000	0.03
200	40 000	0.04
350	122 500	0.05
500	250 000	0.08
1000	1 000 000	0.27
2000	4 000 000	1.09
3500	12 250 000	5.10
5000	25 000 000	12.82

The execution time is reduced by up to 45 percent.

6.11 A summary of our best results

Table 41 summarizes the techniques described in this chapter.

Table 41. The optimization how to

Situation	Possible solutions	Shown bottleneck improvement	Comments
Large overall computation time	-O -O3 -qstrict	67 % on a realistic example	Preserves the possibility to use profiling
Large overall computation time	-O3, -O4, -O5	85 % on a realistic example	Incompatible with profiling
Overhead due to many function calls	-qipa	92 % on a realistic example	Modularity no more enemy of efficiency
Many uses of some small functions	- qinline	Up to 99 % on a limit situation example	More readable and less possible side effects than macros
Large arrays of structures	-qalign	About 30 %	Comparisons need recompilations
We use mostly one type of machine	-qtune		Keeps binary code compatibility
We use ONLY one type of machine	-qarch	About 10 %	Drops binary code compatibility
Using templates in many C++ compilation units	-qfuncsect	40 % code size reduction should reduce paging	Bigger libraries, but smaller executables
Large overall computation time	Multithreading	Up to 91 % on a machine with 12 processors	No recompilation needed to run on uniprocessors
We want a fair share of CPU time	Adjust M:N ratio	Up to 60 % on a realistic example	Trial and error possible with the same executable
Threads compete to get and free heap memory	Malloc multiheap	Up to 90 % on a very specific example	Trial and error possible with the same executable.
Stride situation	Use smallest possible chunks of data	Up to 50 % in a plausible example.	May be more or less, depending on the degree of paging

Appendix A. Special notices

This publication is intended to help application developers using IBM C and C++ compilers on the AIX operating system. The information in this publication is not intended as the specification of any programming interfaces that are provided by the compilers. See the PUBLICATIONS section of the IBM Programming Announcement for IBM C for AIX Version 5, and VisualAge C++ Professional for AIX Version 5 for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers

attempting to adapt these techniques to their own environments do so at their own risk.

Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

The following terms are trademarks of the International Business MachinesCorporation in the United States and/or other countries:

AIX
IBM
PartnerWorld
SP
Redbooks
Redbooks Logo 
C Set ++
Open Class
RS/6000
VisualAge

The following terms are trademarks of other companies:

Tivoli, Manage. Anything. Anywhere.,The Power To Manage., Anything. Anywhere.,TME, NetView, Cross-Site, Tivoli Ready, Tivoli Certified, Planet Tivoli, and Tivoli Enterprise are trademarks or registered trademarks of Tivoli Systems Inc., an IBM company, in the United States, other countries, or both. In Denmark, Tivoli is a trademark licensed from Kjøbenhavns Sommer - Tivoli A/S.

C-bus is a trademark of Corollary, Inc. in the United States and/or other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States and/or other countries.

PC Direct is a trademark of Ziff Communications Company in the United States and/or other countries and is used by IBM Corporation under license.

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States and/or other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through The Open Group.

SET, SET Secure Electronic Transaction, and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

Other company, product, and service names may be trademarks or service marks of others.

Appendix B. Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

B.1 IBM Redbooks

For information on ordering these publications see “How to get IBM Redbooks” on page 247.

- *Understanding RS/6000 Performance and Sizing*, SG24-4810
- *RS/6000 Performance Tools in Focus*, SG24-4989
- *Getting to Know VisualAge C++, Version 4.0*, SG24-5489
- *AIX Version 4.3 Differences Guide*, SG24-2014

B.2 IBM Redbooks collections

Redbooks are also available on the following CD-ROMs. Click the CD-ROMs button at ibm.com/redbooks for information about all the CD-ROMs offered, updates and formats.

CD-ROM Title	Collection Kit Number
IBM System/390 Redbooks Collection	SK2T-2177
IBM Networking Redbooks Collection	SK2T-6022
IBM Transaction Processing and Data Management Redbooks Collection	SK2T-8038
IBM Lotus Redbooks Collection	SK2T-8039
Tivoli Redbooks Collection	SK2T-8044
IBM AS/400 Redbooks Collection	SK2T-2849
IBM Netfinity Hardware and Software Redbooks Collection	SK2T-8046
IBM RS/6000 Redbooks Collection	SK2T-8043
IBM Application Development Redbooks Collection	SK2T-8037
IBM Enterprise Storage and Systems Management Solutions	SK3T-3694

B.3 Other resources

These publications are also relevant as further information sources:

- *Using License Use Management Runtime for AIX*, SH19-4346

B.4 Referenced Web sites

These Web sites are also relevant as further information sources:

- http://www.rs6000.ibm.com/library
- http://www-4.ibm.com/software/ad/vacpp/support.html
- http://http://www-4.ibm.com/software/ad/caix/support.html
- http://www.developer.ibm.com
- ftp://ftp.software.ibm.com/software/lum/aix/doc/V4.5.5/lumusgaix.pdf
- http://www.ibm.com/servers/aix/products/ibmsw/list/

How to get IBM Redbooks

This section explains how both customers and IBM employees can find out about IBM Redbooks, redpieces, and CD-ROMs. A form for ordering books and CD-ROMs by fax or e-mail is also provided.

- **Redbooks Web Site** ibm.com/redbooks

  Search for, view, download, or order hardcopy/CD-ROM Redbooks from the Redbooks Web site. Also read redpieces and download additional materials (code samples or diskette/CD-ROM images) from this Redbooks site.

  Redpieces are Redbooks in progress; not all Redbooks become redpieces and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

- **E-mail Orders**

  Send orders by e-mail including information from the IBM Redbooks fax order form to:

	e-mail address
In United States or Canada	pubscan@us.ibm.com
Outside North America	Contact information is in the “How to Order” section at this site: http://www.elink.ibmlink.ibm.com/pbl/pbl

- **Telephone Orders**

United States (toll free)	1-800-879-2755
Canada (toll free)	1-800-IBM-4YOU
Outside North America	Country coordinator phone number is in the “How to Order” section at this site: http://www.elink.ibmlink.ibm.com/pbl/pbl

- **Fax Orders**

United States (toll free)	1-800-445-9269
Canada	1-403-267-4455
Outside North America	Fax phone number is in the “How to Order” section at this site: http://www.elink.ibmlink.ibm.com/pbl/pbl

This information was current at the time of publication, but is continually subject to change. The latest information may be found at the Redbooks Web site.

IBM Intranet for Employees

IBM employees may register for information on workshops, residencies, and Redbooks by accessing the IBM Intranet Web site at http://w3.itso.ibm.com/ and clicking the ITSO Mailing List button. Look in the Materials repository for workshops, presentations, papers, and Web pages developed and written by the ITSO technical professionals; click the Additional Materials button. Employees may access MyNews at http://w3.ibm.com/ for redbook, residency, and workshop announcements.

IBM Redbooks fax order form

Please send me the following:

Title	Order Number	Quantity

First name	Last name	
Company		
Address		
City	Postal code	Country
Telephone number	Telefax number	VAT number

☐ Invoice to customer number _____

☐ Credit card number _____

Credit card expiration date	Card issued to	Signature

We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.

Index

Symbols

Numerics

A

B

C

D

E

F

G

H

I

J

N

O

P

Q

R

S

T

U

V

W

X

Y

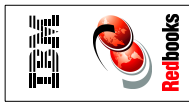
Z

IBM Redbooks review

Your feedback is valued by the Redbook authors. In particular we are interested in situations where a Redbook "made the difference" in a task or problem you encountered. Using one of the following methods, **please review the Redbook, addressing value, subject matter, structure, depth and quality as appropriate.**

- Use the online **Contact us** review redbook form found at **ibm.com**/redbooks
- Fax this form to: USA International Access Code + 1 914 432 8264
- Send your comments in an Internet note to redbook@us.ibm.com

Document Number **Redbook Title**	SG24-5674-00 C and C++ Application Development on AIX
Review	
What other subjects would you like to see IBM Redbooks address?	
Please rate your overall satisfaction:	O Very Good O Good O Average O Poor
Please identify yourself as belonging to one of the following groups:	O Customer O Business Partner O Solution Developer O IBM, Lotus or Tivoli Employee O None of the above
Your email address: The data you provide here may be used to provide you with information from IBM or our business partners about our products, services or activities.	O Please do not use the information collected here for future marketing or promotional contacts or other communications beyond the scope of this transaction.
Questions about IBM's privacy policy?	The following link explains how we protect your personal information. **ibm.com**/privacy/yourprivacy/



C and C++ Application Development on AIX

C and C++ Application Development on AIX

Create and use C and C++ shared libraries

Understand shared memory and C++ templates

Improve application performance

Application development on any platform is a complex subject, particularly if you are unfamiliar with the operating system. Familiar tools may be missing, and things don't work like you expect them to.

This IBM Redbook covers the subject areas that most often cause problems for developers when they migrate their C and C++ applications to AIX. The subjects explained include: Shared libraries, C++ templates, shared memory, compiler products and options, and measuring and improving the performance of applications. The book contains many source code and command examples to illustrate each problem along with recommended solutions.

This book is a must for experienced UNIX application developers who are about to migrate applications to AIX. It is also useful for existing AIX developers who want to increase the efficiency and performance of their applications.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

SG24-5674-00 ISBN 0738417114