PART FOUR

Models of Parallel Computation

Chapter 10, "Models of Parallel Computation"

Provides an overview of the different models around which you can design a parallel or distributed application in Silicon Graphics systems.

Chapter 11, "Statement−Level Parallelism"

Gives an overview of the use of Power Fortran and Power C to execute do−loops across multiple CPUs.

Chapter 12, "Process−Level Parallelism"

Describes the use of IRIX processes to execute in parallel within one address space or in multiple address spaces.

Chapter 13, "Thread−Level Parallelism"

Describes the use of POSIX threads (IEEE 1003.1c) for parallel execution within a single address space.

Chapter 14, "Message−Passing Parallelism"

Describes two different facilities for distributing an application across multiple host computers: PVM and MPI.

Chapter 10

Models of Parallel Computation

You design a program to perform computations in parallel in order to get higher performance, by bringing more hardware to bear on the problem concurrently. In order to succeed, you need to understand the hardware architecture of the target system, and also the software interfaces that are available.

The purpose of this chapter is to give a high−level overview of parallel programming models and of the hardware that they use. The parallel models are discussed in more detail in following chapters.

Parallel Hardware Models

Silicon Graphics makes a variety of systems:

The O2, Indy, and Indigo workstations have single CPUs. Although they can perform I/O operations in parallel with computing, they can execute only one stream of instructions at a time, and time−share the CPU across all active processes.

The CHALLENGE and Onyx systems (and their POWER versions) are symmetric multiprocessor (SMP) computers. In these systems at least 2, and as many as 36, identical microprocessors access a single, common memory and a common set of peripherals through a high−speed bus.

The OCTANE workstation is a two−CPU SMP.

The POWER CHALLENGEarray comprises 2 or more POWER CHALLENGE systems connected by a high−speed local HIPPI network. Each node in the array is an SMP with 2 to 36 CPUs. Nodes do not share a common memory; communication between programs in different nodes passes through sockets. However, the entire array can be administered and programmed as a single entity.

An Origin2000 system provides nodes each containing two or four CPUs, connected in systems of 2 to 128 nodes by a high−speed connection fabric. All system memory is uniformly addressable, but there is a time penalty for the use of nonlocal memory (see "Using Origin2000 Nonuniform Memory").

Most programs have a single thread of execution that runs as if it were in a uniprocessor, employing the facilities of a single CPU. The IRIX operating system applies CPUs to different programs in order to maximize system throughput.

You can write a program so that it makes use of more than one CPU at a time. The software interface that you use for this is the parallel programming model. The IRIX operating system gives you a variety of such interfaces. Each one is designed around a different set of assumptions about the hardware, especially the memory system.

Each model is implemented using a different library of code linked with your program. In some cases you can design a mixed−model program, but in general this is a recipe for confusion.

Parallel Programs on Uniprocessors

It might seem a contradiction, but it is possible to execute some parallel programs in uniprocessors. Obviously you would not do this expecting the best performance. However, it is easier to debug a parallel program by running it in the more predictable environment of a single CPU, on a multiprocessor or on a uniprocessor workstation. Also, you might deliberately restrict a parallel program to one CPU in order to establish a performance baseline.

Most parallel programming libraries adapt to the available hardware. They run concurrently on multiple CPUs when the CPUs are available (up to some programmer−defined limit). They run on a limited number, or even just one CPU, when necessary. For example, the Fortran programmer can control the number of CPUs used by a MIPSpro Fortran 77 program by setting environment variables before the program starts (see **Chapter 11, "Statement−Level Parallelism"**).

Types of Memory Systems

The key memory issue for parallel execution is this: Can one process access data in memory that belongs to another concurrent process, and if so, what is the time penalty for doing so? The answer depends on the hardware architecture, and determines the optimal programming model.

Single Memory Systems

The CHALLENGE/Onyx system architecture uses a high speed system bus to connect all components of the system.

One component is the physical memory system, which plugs into the bus and is equally available to all other components. Other units that plug into the system bus are I/O adapters, such as the VME bus adapter. CPU modules containing MIPS R4000, R8000, or R10000 CPUs are also plugged into the system bus.

In the CHALLENGE/Onyx architecture, the single, common memory has these features:

There is a single address map; that is, the same word of memory has the same address in every CPU.

There is no time penalty for communication between processes because every memory word is accessible in the same amount of time from any CPU.

All peripherals are equally accessible from any process.

The OCTANE workstation also uses a single, common memory that is accessible from either of its CPUs in the same amount of time.

The effect of a single, common memory is that processes running in different CPUs can share memory and can update the identical memory locations concurrently. For example, suppose there are four CPUs available to a Fortran program that processes a large array of data. You can divide a single DO−loop so that it executes concurrently on the four CPUs, each CPU working in one−fourth of the array in memory.

As another example, IRIX allows processes to map a single segment of memory into the virtual address spaces of two or more concurrent processes (see Chapter 3, "Sharing Memory Between Processes"). Two processes can transfer data at memory speeds, one putting the data into a mapped segment and the other process taking the data out. They can coordinate their access to the data using semaphores located in the shared segment (see Chapter 4, "Mutual Exclusion").

Multiple Memory Systems

In an Array system, such as a POWER CHALLENGEarray, each node is a computer built on the CHALLENGE/Onyx architecture. However, the only connection between nodes is the high−speed HIPPI bus between nodes. The system does not offer a single system memory; instead, there is a separate memory subsystem in each node. The effect is that:

There is not a single address map. A word of memory in one node cannot be addressed at all from another node.

There is a time penalty for some interprocess communication. When data passes between programs in different nodes, it passes over the HIPPI network, which takes longer than a memory−to−memory transfer.

Peripherals are accessible only in the node to which they are physically attached.

Nevertheless, it is possible to design an application that executes concurrently in multiple nodes of an Array. The message−passing interface (MPI) is designed specifically for this.

Hierarchic, Nonuniform Memory Systems

The Origin2000 system uses a memory hierarchy. A certain amount of memory is a physical part of each node. The hardware creates the image of a single system memory. The memory installed in any node can be accessed from any other node as if all memory were local. However, the node number is part of the physical address of a word of memory. There is a multiple−level hierarchy of speed: memory in the same node as the CPU is accessed in the least amount of time, while memory in any other node takes an additional fraction of a microsecond to access. The time penalty depends on the relative location of the two nodes in the system.

These are the results of this design:

There is a single address map. A word of memory can be addressed from any node.

There is a time penalty for some accesses, depending on the node that requests the memory and the node that contains it. However, this time cost is far smaller than the cost of communicating over a socket and a network link.

Peripherals are accessible from any node, but there is a time penalty for access to a peripheral from a node other than the one to which the peripheral is attached.

The implications of these features are explored at more length under "Using Origin2000 Nonuniform Memory".

Parallel Execution Models

You can compare the available models for parallel programming on two features:

granularity
: The relative size of the unit of computation that executes in parallel: a single statement, a function, or an entire process.

communication channel
: The basic mechanism by which the independent, concurrent units of the program exchange data and synchronize their activity.

A summary comparison of the available models is shown in Table 10−1.

Table 10−1 Comparing Parallel Models

Model	Granularity	Communication
Power Fortran, IRIS POWER C	Looping statement (DO or *for* statement)	Shared variables in a single user address space.
Ada95 tasks	Ada Procedure	Shared variables in a single user address space.
POSIX threads	C function	Shared variables in a single user address space.
Lightweight UNIX processes (*sproc()*)	C function	Arena memory segment in a single user address space.
General UNIX processes (*fork()*, *exec()*)	Process	Arena segment mapped to multiple address spaces.
Shared Memory (SHMEM)	Process	Memory copy.
Parallel Virtual Machine (PVM)	Process	Memory copy within node; HIPPI network between nodes.
Message−Passing (MPI)	Process	Memory copy within node; special HIPPI Bypass interface between nodes.

Process−Level Parallelism

A UNIX process consists of an address space, a large set of process state values, and one thread of execution. The main task of the IRIX kernel is to create processes and to dispatch them to different CPUs to maximize the utilization of the system.

IRIX contains a variety of interprocess communication (IPC) mechanisms, which are discussed in Chapter 2, "Interprocess Communication." These mechanisms can be used to exchange data and to coordinate the activities of multiple, asynchronous processes within a single−memory system.

(Processes running in different nodes of an Array must use one of the distributed models; see "Message−Passing Models".)

In traditional UNIX practice, one process creates another with the system call *fork()*, which makes a duplicate of the calling process, after which the two copies execute in parallel. Typically the new process immediately uses the *exec()* function to load a new program. (The fork(2) reference page contains a complete list of the state values that are duplicated when a process is created. The exec(2) reference page details the process of creating a new program image for execution.)

IRIX also supports the system function *sproc()*, which creates a lightweight process. A process created with *sproc()* shares some of its process state values with its parent process (the sproc(2) reference page details how this sharing is specified).

In particular, a process made with *sproc()* does not have its own address space. It continues to execute in the address space of the original process. In this respect, a lightweight process is like a thread (see "Thread−Level Parallelism"). However, a lightweight process differs from a thread in two significant ways:

A lightweight process still has a full set of UNIX state values. Some of these, for example the table of open file descriptors, can be shared with the parent process, but in general a lightweight process carries most of the state information of a process.

Dispatch of lightweight processes is done in the kernel, and has the same overhead as dispatching any process.

The library support for statement−level parallelism is based on the use of lightweight processes (see "Statement−Level Parallelism").

Thread−Level Parallelism

A thread is an independent execution state within the context of a larger program. The concept of a thread is well−known, but the most common formal definition of threads and their operation is provided by POSIX standard 1003.1c, "System Application Program Interface—Amendment 2: Threads Extension."

There are three key differences between a thread and a process:

A UNIX process has its own set of UNIX state information, for example, its own effective user ID and set of open file descriptors.

Threads exist within a process and do not have distinct copies of these UNIX state values. Threads share the single state belonging to their process.

Normally, each UNIX process has a unique address space of memory segments that are accessible only to that process (lightweight processes created with *sproc()* share all or part of an address space).

Threads within a process always share the single address space belonging to their process.

Processes are scheduled by the IRIX kernel. A change of process requires two context changes, one to enter the kernel domain and one to return to the user domain of the next process. The change from the context of one process to the context of another can entail many instructions.

In contrast, threads are scheduled by code that operates largely in the user address space, without kernel assistance. Thread scheduling can be faster than process scheduling.

The POSIX standard for multithreaded programs is supported by IRIX 6.2 with patches 1361, 1367, and 1389 installed, and in all subsequent releases of IRIX.

In addition, the Silicon Graphics implementation of the Ada95 language includes support for multitasking Ada programs—using what are essentially threads, although not implemented using the POSIX library. For a complete discussion of the Ada 95 task facility, refer to the *Ada 95 Reference Manual*, which installs with the Ada 95 compiler (GNAT) product.

Statement−Level Parallelism

The finest level of granularity is to run individual statements in parallel. This is provided using any of three language products:

MIPSpro Fortran 77 supports compiler directives that command parallel execution of the bodies of DO−loops. The MIPSpro POWER Fortran 77 product is a preprocessor that automates the insertion of these directives in a serial program.

MIPSpro Fortran 90 supports parallelizing directives similar to MIPSpro Fortran 77, and the MIPSpro POWER Fortran 90 product automates their placement.

MIPSpro POWER C supports compiler pragmas that command parallel execution of segments of code. The IRIS POWER C analyzer automates the insertion of these pragmas in a serial program.

In all three languages, the run−time library—which provides the execution environment for the compiled program—contains support for parallel execution. The compiler generates library calls. The library functions create lightweight processes using *sproc()*, and distribute loop iterations among them.

The run−time support can adapt itself dynamically to the number of available CPUs. Alternatively, you can control it—either using program source statements, or using environment variables at execution time—to use a certain number of CPUs.

Statement−level parallel support is based on using common variables in memory, and so it can be used only within the bounds of a single−memory system, a CHALLENGE system or a single node in a POWER CHALLENGEarray system.

Message−Passing Models

One way to design a parallel program is to think of each thread of execution as operating in an independent address space, communicating with other threads by exchanging discrete units of data as messages through an abstract, formal interface for message exchange.

The threads of a program designed in this way can be distributed over different computers. Three message−passing execution models are supported by Silicon Graphics systems. Each defines and implements a formal, abstract model for data exchange. Two of the models allow a computation to be distributed across the nodes of a multiple−memory system, without having to reflect the system configuration in the source code. The programming models are:

Shared Memory Model (SHMEM)

Message−Passing Interface (MPI)

Parallel Virtual Machine (PVM)

All three models are briefly summarized in the following topics, and are discussed in more detail in Chapter 14, "Message−Passing Parallelism."Support for all three is included in the Message−Passing Toolkit (MPT) product. For an overview of MPT, see this URL:

http://www.cray.com/products/software/mpt/mpt.html

Shared Memory (SHMEM) Model

The SHMEM library has been used for some time on Cray systems and is now available for all Silicon Graphics multiprocessors as part of the MPT. A program built on SHMEM is a process−level parallel program. Each process runs in a separate address space. The SHMEM library routines are used to exchange data, and to coordinate execution, between the processes.

SHMEM routines support remote data transfer through *put* operations, which transfer data to a different process, and *get* operations, which transfer data from a different process. Other operations supported include data broadcast and data reduction; barrier synchronization; as well as atomic memory updates such as a fetch−and−increment on remote or local data objects.

SHMEM operations are all memory−to−memory, and as a result have extremely high bandwidth and low latency. However, a SHMEM−based program cannot be distributed across multiple systems.

Message−Passing Interface (MPI) Model

MPI is a standard programming interface for the construction of a portable, parallel application in Fortran 77 or in C, especially when the application can be decomposed into a fixed number of processes operating in a fixed topology (for example, a pipeline, grid, or tree). MPI has wide use on many large computers.

A highly tuned, efficient implementation of MPI is part of the MPT. Within a single system, MPI messages are moved memory−to−memory. Between nodes of an Silicon Graphics Array system, MPI messages are passed over a HIPPI network. Latency and bandwidth are intermediate between memory−to−memory data exchange and socket−based network communication.

Parallel Virtual Machine (PVM) Model

PVM is an integrated set of software tools and libraries that emulates a general−purpose, flexible, heterogeneous, concurrent computing framework on interconnected computers of varied architecture. Using PVM, you can create a parallel application that executes as a set of concurrent processes on a set of computers that can include uniprocessors, multiprocessors, and nodes of Array systems.

Like MPI, PVM has wide use on many types of supercomputer, including Cray systems. An implementation of PVM is included in the MPT. PVM is discussed in more detail under Chapter 14, "Message−Passing Parallelism."

Chapter 11

Statement−Level Parallelism

You can use statement−level parallelism in three language packages: Fortran 77, Fortran 90, and C. This execution model is unique in that you begin with a normal, serial program, and you can always return the program to serial execution by recompiling. Every other parallel model requires you to plan and write a parallel program from the start.

Products for Statement−Level Parallelism

Software support for statement−level parallelism is available from Silicon Graphics and from independent vendors.

Silicon Graphics Support

The parallel features of the three languages from Silicon Graphics are documented in detail in the manuals listed in Table 11−1.

Table 11−1 Documentation for Statement−Level Parallel Products

Manual	Document Number	Contents
C Language Reference Manual	007−0701−*nnn*	Covers all pragmas, including parallel ones.
IRIS Power C User's Guide	007−0702−*nnn*	Use of Power C source analyzer to place pragmas automatically.
MIPSpro Fortran 77 Programmer's Guide	007−2361−*nnn*	General use of Fortran 77, including parallelizing assertions and directives.
MIPSpro Power Fortran 77 Programmer's Guide	007−2363−*nnn*	Use of the Power Fortran source analyzer to place directives automatically.
MIPSpro Fortran 90 Programmer's Guide	007−2761−*nnn*	General use of Fortran 90, including parallelizing assertions and directives.
MIPSpro Power Fortran 90 Programmer's Guide	007−2760−*nnn*	Use of the Power Fortran 90 source analyzer to place directives automatically.

Products from Other Vendors

In addition to these products from Silicon Graphics, the High Performance Fortran (HPF) compiler from the Portland Group is a compiler for Fortran 90 augmented to the HPF standard. It supports automatic parallelization. (Refer to http://www.pgroup.com for more information).

The FORGE products from Applied Parallel Research (APRI) contain a Fortran 77 source analyzer that can insert parallelizing directives, although not the directives supported by MIPSpro Fortran 77. (Refer to http://www.apri.com for more information.)

Creating Parallel Programs

In each of the three languages, the language compiler supports explicit statements that command parallel execution (**#pragma** lines for C; directives and assertions for Fortran). However, placing these statements can be a demanding, error−prone task. It is easy to create a suboptimal program, or worse, a program that is incorrect in subtle ways. Furthermore, small changes in program logic can invalidate parallel directives in ways that are hard to foresee, so it is difficult to modify a program that has been manually made parallel.

For each language, there is a source–level program analyzer that is sold as a separate product (IRIS POWER C, MIPSpro Power Fortran 77, MIPSpro Power Fortran 90). The analyzer identifies sections of the program that can safely be executed in parallel, and automatically inserts the parallelizing directives. After any logic change, you can run the analysis again, so that maintenance is easier.

The source analyzer makes conservative assumptions about the way the program uses data. As a result, it often is unable to find all the potential parallelism. However, the analyzer produces a detailed listing of the program source, showing each segment that could or could not be parallelized, and why. Directed by this listing, you insert source assertions that give the analyzer more information about the program.

The method of creating an optimized parallel program is as follows:

1. Write a complete application that runs on a single processor.
2. Completely debug and verify the correctness of the program in serial execution.
3. Apply the source analyzer and study the listing it produces.
4. Add assertions to the source program. These are not explicit commands to parallelize, but high–level statements that describe the program's use of data.
5. Repeat steps 3 and 4 until the analyzer finds as much parallelism as possible.
6. Run the program on a single–memory multiprocessor.

When the program requires maintenance, you make the necessary logic changes and, simultaneously, remove any assertions about the changed code—unless you are certain that the assertions are still true of the modified logic. Then repeat the preceding procedure from step 2.

Managing Statement–Parallel Execution

The run–time library for all three languages is the same, *libmp*. It is documented in the mp(3) reference page. *libmp* uses IRIX lightweight processes to implement parallel execution (see Chapter 12, "Process–Level Parallelism").

When a parallel program starts, the run–time support creates a pool of lightweight processes using the *sproc()* function. Initially the extra processes are blocked, while one process executes the opening passage of the program. When execution reaches a parallel section, the run–time library code unblocks as many processes as necessary. Each process begins to execute the same block of statements. The processes share global variables, while each allocates its own copy of variables that are local to one iteration of a loop, such as a loop index.

When a process completes its portion of the work of that parallel section, it returns to the run–time library code, where it picks up another portion of work if any work remains, or suspends until the next time it is needed. At the end of the parallel section, all extra processes are suspended and the original process continues to execute the serial code following the parallel section.

Controlling the Degree of Parallelism

You can specify the number of lightweight processes that are started by a program. In IRIS POWER C, you can use *#pragma numthreads* to specify the exact number of processes to start, but it is not a good idea to embed this number in a source program. In all implementations, the run–time library by default starts enough processes so there is one for each CPU in the system. That default is often too high, since typically not all CPUs are available for one program.

The run−time library checks an environment variable, MP_SET_NUM_THREADS, for the number of processes to start. You can use this environment variable to choose the number of processes used by a particular run of the program, thereby tuning the program's requirements to the system load. You can even force a parallelized program to execute on a single CPU when necessary.

MIPSpro Fortran 77 and MIPSpro Fortran 90 also recognize additional environment variables that specify a range of process numbers, and use more or fewer processes within this range as system load varies. (See the *Programmer's Guide* for the language for details.)

At certain points the multiple processes must wait for one another before continuing. They do this by waiting in a busy loop for a certain length of time, then by blocking until they are signaled. You can specify the amount of time that a process should spend spinning before it blocks, using either source directives or an environment variable (see the *Programmer's Guide* for the language for system functions for this purpose).

Choosing the Loop Schedule Type

Most parallel sections are loops. The benefit of parallelization is that some iterations of the loop are executed in one CPU, concurrent with other iterations of the same loop in other CPUs. But how are the different iterations distributed across processes? The languages support four possible methods of scheduling loop iterations, as summarized in Table 11−2

Table 11−2 Loop Scheduling Types

Schedule	Purpose
SIMPLE	Each process executes $\lfloor N/P \rfloor$ *iterations starting at* $Q*(\lfloor N/P \rfloor)$. *First process to finish takes the remainder chunk, if any.*
DYNAMIC	Each process executes *C* iterations of the loop, starting with the next undone chunk unit, returning for another chunk until none are left undone.
INTERLEAVE	Each process executes *C* iterations at $C*Q$, $C*2Q$, $C*3Q$...
GSS	Each process executes chunks of decreasing size, $(N/2P)$, $(N/4P)$,...

The variables used in Table 11−2are as follows:

N Number of iterations in the loop, determined from the source or at run−time.

P Number of available processes, set by default or by environment variable (see "Controlling Degree of Parallelism").

Q Number of a process, from 0 to N–1.

C "Chunk" size, set by directive or by environment variable.

The effects of the scheduling types depend on the nature of the loops being parallelized. For example:

The SIMPLE method works well when *N* is relatively small. However, unless *N* is evenly divided by *P*, there will be a time at the end of the loop when fewer than *P* processes are working, and possibly only one.

The DYNAMIC and INTERLEAVE methods allow you to set the chunk size to control the span of an array referenced by each process. You can use this to reduce cache effects. When *N* is very large so that not all data fits in memory, INTERLEAVE may reduce the amount of paging compared to DYNAMIC.

The guided self−scheduling (GSS) method is good for triangular matrices and other algorithms where loop iterations become faster toward the end.

You can use source directives or pragmas within the program to specify the scheduling type and chunk size for particular loops. Where you do not specify the scheduling, the run−time library uses a default method and chunk size. You can establish this default scheduling type and chunk size using environment variables.

Distributing Data

In any statement−level parallel program, memory cache contention can harm performance. This subject is covered under "Dealing With Cache Contention".

When a statement−parallel program runs in an Origin2000 or Onyx2 system, the location of the program's data can affect performance. These issues are covered at length under "Using Origin2000 Nonuniform Memory".

Chapter 12
Process−Level Parallelism

The process is the traditional unit of UNIX execution. The concept of the process (and its relationship to the concept of a thread) are covered under "Process−Level Parallelism". The purpose of this chapter is to review how you can use IRIX processes to perform parallel processing in a single program.

Using Multiple Processes

In general, you can create a new process for each unit of work that your program could do in parallel. The processes can share the address space of the original program, or each can have its own address space. You design the processes so that they coordinate work and share data using any and all of the interprocess communication (IPC) features discussed in Part II, "Interprocess Communication."

Software products from Silicon Graphics use process−level parallelism. For example, the IRIS Performer graphics library normally creates a separate lightweight process to manage the graphics pipe in parallel with rendering work. The run−time library for statement−level parallelism creates a pool of lightweight processes and dispatches them to execute parts of loop code in parallel (see "Managing Statement−Parallel Execution").

Process Creation and Share Groups

The most important system functions you use to create and manage processes are summarized in Table 12−1.

Table 12−1 Commands and System Functions for Process Management

Function Name	Purpose and Operation
npri(1)	Command to run a process at a specified nondegrading priority.
runon(1)	Command to run a process on a specific CPU.
fork(2)	Create a new process with a private address space.
pcreate(3C)	Create a new process with a private address space running a designated program with specified arguments.
sproc(2)	Create a new process in the caller’s address space using a private stack.
sprocsp(2)	Create a new process in the caller’s address space using a preallocated stack area.
prctl(2)	Query and set assorted process attributes.
sysmp(2)	Query multiprocessor status and assign processes to CPUs.
syssgi(2)	Query process virtual and real memory use, and other operations.

You can initiate a program at a specified nondegrading priority (explained under "Process Scheduling") using *npri*. You can initiate a program running on a specific CPU of a multiprocessor using *runon*. Both attributes—the assigned priority and the assigned CPU—are inherited by any child processes that the program creates.

Process Creation

The process that creates another is called the *parent* process. The processes it creates are *child* processes, or siblings. The parent and its children together are a *share group*. IRIX provides special services to share groups. For example, you can send a signal to all processes in a share group.

The *fork()* function is the traditional UNIX way of creating a process. The new process is a duplicate of the parent process, running in a duplicate of the parent's address space. Both execute the identical program text; that is, both processes "return" from the *fork()* call. Your code can distinguish them by the return code, which is 0 in the child process, but in the parent is the new process ID.

The *sproc()* and *sprocsp()* functions create a lightweight process. The difference between these calls is that *sproc()* allocates a new memory segment to serve as the stack for the new process. You use *sprocsp()* to specify a stack segment that you have already allocated—for example, a block of memory that you allocate and lock against paging using *mpin()*.

The *sproc()* calls take as an argument the address of the function that the new process should execute. The new process begins execution in that function, and when that function returns, the process is terminated. Read the sproc(2) reference page for details on the flags that specify which process attributes a child process shares with its parent, and for other comparisons between *fork()* and *sproc()*.

Note: The *sproc()* and *sprocsp()* functions are not available for use in a threaded program (see Chapter 13, "Thread−Level Parallelism"). The pthreads library uses lightweight processes to implement threading, and has to control the creation of processes. Also, when your program uses the MPI library (see Chapter 14, "Message−Passing Parallelism"), the use of *sproc()* and *sprocsp()* can cause problems.

Process Management

Certain system functions give you some control over the processes you create. The *prctl()* function offers a variety of operations. These are some of the most useful:

PR_MAXPROCS	Query the system limit on processes per user (also available from *sysconf(*_SC_CHILD_MAX*)*, see sysconf(2).
PR_MAXPPROCS	Query the maximum number of CPUs that are available to the calling process its children. This reflects both the system hardware and reservations made on CPUs, but does not reflect system load.
PR_GETNSHARE	Query the number of processes in the share group with the calling process.
PR_GETSTACKSIZE	Query the maximum size of the stack segment of the calling process. For the parent process this reflects the system limit (also available from *getrlimit(*RLIMIT_STACK*)*, see getrlimit(2)). For a process started by *sprocs* the size of the allocated stack.
PR_SETSTACKSIZE	Set an upper limit on stack growth for the calling process and for child proces it creates in the future.
PR_RESIDENT	Prevent the calling process from being swapped out. This has no connection t paging, but to swapping out an entire, inactive process under heavy system lo

The *sysmp()* function gives a privileged process information about and control over the use of a multiprocessor. Some of the operations it provides are as follows:

MP_NPROCS	Number of CPUs physically in the system.
MP_NAPROCS	Number of CPUs available to the scheduler; should be the same as *prctl(*PR_MAXPPROCS*)*.
MP_MUSTRUN	Assign the calling process to run on a specific CPU.
MP_MUSTRUN_PID	Assign a specified other process (typically a just−created child proc to run on a specific CPU.
MP_GETMUSTRUN MP_GETMUSTRUN_PID	Query the must−run assignment of the calling process or of a specif process.
MP_RUNANYWHERE MP_RUNANYWHERE_PID	Allow the calling process, or a specified process, to run on any CPU

The *runon* command (see "Process Creation" and runon(1)) initiates the parent process of a program running on a specific CPU. Any child processes also runs on that CPU unless the parent reassigns them to run anywhere, or to run on a different CPU, using *sysmp()*. The use of restricted CPUs and assigned CPUs to get predictable real−time performance is discussed at length in the*REACT Real−Time Programmer's Guide*

The *syssgi()* function has a number of interesting uses but only one of interest for managing processes: *syssgi(*SGI_PROCSZ*)* returns the virtual and resident memory occupancy of the calling process.

Process "Reaping"

A parent process should not terminate while its child processes continue to run. When it does so, the parent process of each child becomes 1, the *init* process. This causes problems if a child process should loop or hang. The functions you use to collect (the technical term is to "reap") the status of child processes are summarized in Table 12−2

Table 12−2 Functions for Child Process Management

Function Name	Purpose and Operation
wait(2)	Function to block until a child stops or terminates, and to receive the cause of its change of status.
waitpid(2)	POSIX extension of *wait()* which allows more selectivity and returns more information.
wait3(2)	BSD extension of *wait()* that allows you to poll for terminated children without suspending.
waitid(2)	Function to suspend until one of a selected set of status changes occurs in one or more child processes.

When the parent process has nothing to do after starting the child processes, it can loop on *wait()* until *wait()* reports no more children exist; then it can exit.

Sometimes it is necessary to handle child termination and other work, and the parent cannot suspend. In this case the parent can treat the termination of a child process as an asynchronous event, and trap it in a signal handler for SIGCLD (see "Catching Signals"). The wait(2) reference page has extensive discussion of the three methods (BSD, SVR4, and POSIX) for handling this situation, with example code for each.

Process Scheduling

There are two different approaches to setting the scheduling priorities of a process, one compatible with IRIX and BSD, the other POSIX compliant.

Controlling Scheduling With IRIX and BSD−Compatible Facilities

The IRIX compatible and BSD compatible scheduling operations are summarized in Table 12−3

Table 12−3 Commands and Functions for Scheduling Control

Function Name	Purpose and Operation
schedctl(2)	Query and set IRIX process scheduling attributes.
getpriority(2)	Return the scheduling priority of a process or share group.
setpriority(2)	Set the priority of a process or process group.
nice(1)	Run a program at a positive or negative increment from normal priority.
renice(1)	Alter the priority of a running process by a positive or negative increment.

For BSD compatibility, use the *nice* and *renice* commands to alter priorities, and within a program use *getpriority()* and *setpriority()* to query and set priorities. These commands and functions use priority numbers ranging from −20 through 0 to +20, with lower arithmetic values having superior access to the CPU.

Only the IRIX *schedctl()* function gives you complete access to a variety of operations related to process scheduling. Some of the key operations are as follows:

NDPRI	Set a nondegrading priority for the calling process (see text).
GETNDPRI	Query the nondegrading priority of the calling process.
SETMASTER	Set the master process of a share group. By default the parent pro is the master process, but it can transfer that honor.
SCHEDMODE, SGS_SINGLE	Cause all processes in the share group to be suspended except the master process (set with SETMASTER).
SCHEDMODE, SGS_GANG	Cause all processes in the share group to be scheduled as a "gang with all running concurrently.
SCHEDMODE, SGS_FREE	Schedule the share group in the default fashion.

A program started interactively inherits a scheduling discipline based on degrading priorities. That is, the longer the process executes without voluntarily suspending, the lower its dispatching priority becomes. This strategy keeps a runaway process from monopolizing the hardware. However, you may have a CPU−intensive application that needs a predictable execution rate. This is the purpose of nondegrading priorities set with *schedctl(*NDPRI*)* or with the *npri* command (see the npri(1) reference page).

There are three bands of nondegrading priorities, designated by symbolic names declared in *sys/schedctl.h*:

A real−time band from NDPHIMAX to NDPHIMIN. System daemons and real−time programs run in this band, which has higher priority than any interactive process.

A normal band from NDPNORMMAX to NDPNORMMIN. These values have the same priority as interactive programs. Processes at these priorities compete with interactive processes, but their priorities do not degrade with time.

A batch band from NDPLOMAX to NDPLOMIN. Processes at these priorities receive available CPU time and are scheduled from a batch queue.

Tip: The IRIX priority numbers are inverted, in the sense that numerically smaller values have superior priority. For example. NDPHIMAX is 30 and NDPHIMIN is 39. However, as long as you declare priority values using symbolic expressions, the numbers work out correctly. For example, the statement

```
#define NDPHIMIDDLE NDPHIMIN+((NDPHIMAX−NDPHIMIN)/2)
```

produces a "middle" value of 35, as it should.

When you create a cooperating group of processes, it is important that they all execute at the same time, provided there are enough CPUs to handle all the members of the group that are ready to run. This minimizes the time that members of the share group spend waiting for each other to release locks or semaphores.

Use *schedctl()* to initiate "gang" scheduling for the share group. IRIX attempts to schedule all processes to execute at the same time, when possible.

Note: Through IRIX 6.2, *schedctl()* also supported a scheduling mode called "deadline scheduling." This scheduling mode is being removed and will not be supported in the future. Do not design a program based on the use of deadline scheduling.

Controlling Scheduling With POSIX Functions

The POSIX compliant functions to control process scheduling are summarized in Table 12−4.

Table 12−4 POSIX Functions for Scheduling

Function Name	Purpose and Operation
sched_getparam(2) sched_setparam(2)	Query and change the POSIX scheduling priority of a process.
sched_getscheduler(2) sched_setscheduler(2)	Query and change the POSIX scheduling policy and priority of a process.
sched_get_priority_max(2) sched_get_priority_min(2)	Query the maximum (most use of CPU) and minimum (least use) priority numbers for use with *sched_getparam()*.
sched_get_rr_interval(2)	Query the timeslice interval of the round−robin scheduling policy.
sched_yield(2)	Let other processes of the same priority execute.

Use the functions *sched_get_priority_max()* and *sched_get_priority_min()* to get the ranges of priority numbers you can use. Use *sched_setparam()* to change priorities. POSIX dispatching priorities are nondegrading. (Note that in a program that links with the pthreads library, these same function names are library functions that return thread scheduling priority numbers unrelated to process scheduling.)

Tip: The POSIX scheduling priority values reported by these functions and declared in *sched.h* are not numerically the same as the bands supported by *schedctl()* and declared in *sys/schedctl.h*. The POSIX numbers are numerically higher for superior priority. However, the POSIX range is functionally (but not numerically) equivalent to the "normal" range supported by *schedctl()* (NDPNORMMAX to NDPNORMMIN).

POSIX scheduling uses one of two scheduling policies, strict FIFO and round−robin, which are described in detail in the sched_setscheduler(2) reference page. The round−robin scheduler, which rotates processes of equal priority on a time−slice basis, is the default. You can query the time−slice interval with *sched_get_rr_interval()*. You can change both the policy and the priority by using *sched_setscheduler()*.

Self−Dispatching Processes

Often, each child process has a particular role to play in the application, and the function that you name to *sproc()* represents that work. The child process stays in that function until it terminates.

Another design is possible. In some applications, you may have to manage a flow of many relatively short activities that should be done in parallel. However, the *sproc()* function has considerable overhead. It is inefficient to continually create and destroy child processes. You do not want to create a new child process for each small activity and destroy it afterward. Instead, you can create a pool containing a small number of processes. When a piece of work needs to be done, you can dispatch one process to do it. The fragmentary code in Example 12−1shows the general approach.

Example 12−1 Partial Code to Manage a Pool of Processes

```
typedef void (*workFunc)(void *arg);
```

```
struct oneSproc {
   struct oneSproc *next;         /* -> next oneSproc ready to run */
   workFunc calledFunc;           /* -> function the sproc is to call
 */
   void *callArg;                 /* argument to pass to the called f
unc */
   usema_t *sprocDone;            /* optional sema to post on complet
ion */
   usema_t *sprocWait;            /* sproc waits for work here */
} sprocList[NUMSPROCS];
usema_t *readySprocs;             /* count represents sprocs ready to
 work */
uslock_t sprocListLock;           /* mutex control of sprocList head
*/
struct oneSproc *sprocList;       /* -> first ready oneSproc */
/*
|| Put a oneSproc structure on the ready list and sleep on it.
|| Called by a child process when its work is done.
*/
void sprocSleep(struct oneSproc *theSproc)
{
    ussetlock(sprocListLock);       /* acquire exclusive rights to s
procList */
    theSproc->next = sprocList;  /* put self on the list */
    sprocList = theSproc;
    usunsetlock(sprocListLock);  /* release sprocList */
    usvsema(readySprocs);           /* notify master, at least 1 on
the list */
    uspsema(theSproc->sprocWait);/* sleep until master posts me */
}
/*
|| Body of a general-purpose child process. The argument, which must
|| be declared void* to match the sproc() prototype, is the oneSproc
|| structure that represents this process.   The contents of that
|| struct, in particular sprocWait, are initialized by the parent.
*/
void childBody(void *theSprocAsVoid)
{
    struct oneSproc *mySproc = (struct oneSproc *)theSprocAsVoid;
    /* here one could establish signal handlers, etc. */
    for(;;)
    {
        sprocSleep(mySproc);      /* wait for work to do */
        mySproc->calledFunc(mySproc->callArg);  /* do the work */
        if (mySproc->sprocDone)   /* if a completion sema is given,
*/
```

```
            usvsema(mySproc->sprocDone); /* ..post it */
        }
}
/*
|| Acquire a oneSproc structure from the ready list, waiting if nece
ssary.
|| Called by the master process as part of dispatching a sproc.
*/
struct oneSproc *getSproc()
{
    struct oneSproc *theSproc;
    uspsema(readySprocs);        /* wait until at least 1 sproc is f
ree */
    ussetlock(sprocListLock);       /* acquire exclusive rights to s
procList */
    theSproc = sprocList;        /* get address of first free oneSpr
oc */
    sprocList = theSproc->next;  /* make next in list, the head of l
ist */
    usunsetlock(sprocListLock);  /* release sprocList */
    return theSproc;
}
/*
|| Start a function going asynchronously. Called by master process.
*/
void execFunc(workFunc toCall, void *callWith, usema_t *done)
{
    struct oneSproc *theSproc = getSproc();
    theSproc->calledFunc = toCall;     /* set address of func to exe
c */
    theSproc->callArg = callWith;      /* set argument to pass */
    theSproc->sprocDone = done;           /* set sema to post on com
pletion */
    usvsema(theSproc->sprocWait);      /* wake up sleeping process *
/
}
```

Parallelism in Real–Time Applications

In real–time programs such as aircraft or vehicle simulators, separate processes are used to divide the work of the simulation and distribute it onto multiple CPUs. In these demanding applications, the programmer frequently uses IRIX facilities to

- reserve one or more CPUs of a multiprocessor for exclusive use by the application
- isolate the reserved CPUs from all interrupts
- assign specific processes to execute on specific, reserved CPUs

These facilities are described in detail in the *REACT Real−Time Programmer's Guide* (007−2499−*nnn*). Also covered in that book is the use of the Frame Scheduler, an alternate process scheduler. The normal process scheduling algorithm of the IRIX kernel attempts to keep all CPUs busy and to keep all processes advancing in a fair manner. This algorithm is in conflict with the stringent needs of a real−time program, which needs to dedicate predictable amounts of hardware capacity to its processes, without regard to fairness.

The Frame Scheduler seizes one or more CPUs of a multiprocessor, isolates them, and executes a specified set of processes on each CPU in strict rotation. The Frame Scheduler has much lower overhead than the normal IRIX scheduler, and it has features designed for real−time work, including detection of overrun (when a scheduled process does not complete its work in the necessary time) and underrun (when a scheduled process fails to execute in its turn).

At this writing there are no real−time applications that use multiple nodes of an Array system.

Chapter 13
Thread−Level Parallelism

IRIX 6.5 conforms to ISO/IEC 9945−1:1996 and UNIX 98; that is, it supports POSIX threads, or pthreads.

This chapter contains the following main topics:

"Overview of POSIX Threads" summarizes the similarities and differences of pthreads and processes.

"Compiling and Debugging a Pthread Application" covers compiling and debugging tools.

"Creating Pthreads" covers the process of creating a pthread with the desired attributes.

"Executing and Terminating Pthreads" discusses how threads initialize themselves and how you synchronize on thread termination.

"Using Thread−Unique Data"tells how to define variables that have a unique value in each thread.

"Pthreads and Signals" discusses the pthread−specific details of signal handling (see"Signals" for the general information).

"Scheduling Pthreads" covers scheduling priorities and policies.

"Synchronizing Pthreads" details the use of mutexes and condition variables.

Overview of POSIX Threads

A *thread* is an independent execution state; that is, a set of machine registers, a call stack, and the ability to execute code. When IRIX creates a process, it also creates one thread to execute that process. However, you can write a program that creates many more threads to execute in the same address space. For a comparison of pthreads to processes, see "Thread−Level Parallelism".

POSIX threads are similar in some ways to IRIX lightweight processes made with *sproc()*. You use pthreads in preference to lightweight processes for two main reasons: portability and performance. A program based on pthreads is normally easier to port from another vendor's equipment than a program that depends on a unique facility such as *sproc()*. Table 13−1summarizes some of the differences between pthreads and lightweight processes.

Table 13−1 Comparison of Pthreads and Processes

Attribute	POSIX Threads	Lightweight Processes	UNIX Processes
Source portability	Standard interface, portable between vendors	*sproc()* is unique to IRIX	*fork()* is a UNIX standard
Creation overhead	Relatively small	Moderately large	Quite large
Block/Unblock (Dispatch) Overhead	Few microseconds	Many microseconds	Many microseconds
Address space	Shared	Shared, or copy on write, or separate	Separate
Memory−mapped files and arenas	Shared	Shared, or copy on write, or separate	Explicit sharing only
Mutual exclusion objects	Mutexes, condition variables, and read−write	IRIX semaphores and locks; POSIX	IRIX semaphores and locks; POSIX

	locks; POSIX semaphores; IRIX semaphores and locks	semaphores	semaphores
Files, pipes, and I/O streams	Shared single−process file table	Shared or separate file table	Separate file table
Signal masks and signal handlers	Each thread has a mask but handlers are shared	Each process has a mask and its own handlers	Each process has a mask and its own handlers
Resource limits	Single−process limits	Single−process limits	Limits apply to each process separately
Process ID	One PID applies to all threads	PID per process plus share−group PID	PID per process

It takes relatively little time to create or destroy a pthread, as compared to creating a lightweight process. Threads share all resources and attributes of a single process (except for the signal mask; see "Pthreads and Signals"). If you want each executing entity to have its own set of file descriptors, or if you want to make sure that one entity cannot modify data shared with another entity, you must use lightweight processes or normal processes.

Compiling and Debugging a Pthread Application

A pthread application is a C or a C++ program that uses some of the POSIX pthreads functions. In order to use these functions, and in order to access the thread−safe versions of the standard I/O macros, you must include the proper header files and link with the pthreads library. You can debug and analyze the compiled program using some of the tools available for IRIX.

Compiling Pthread Source

The header files related to pthreads functions are summarized in Table 13−2

Table 13−2 Header Files Related to Pthreads

Header	Primary Contents
errno.h	System error codes returned by pthreads functions.
pthread.h	Pthread functions and special pthread data types.
sched.h	The *sched_param* structure and related functions used in setting thread priorities.
stdio.h	Standard stream I/O macros, including thread−safe versions.
sys/types.h	IRIX and standard data types.
limits.h	Some POSIX constants such as _POSIX_THREAD_THREADS_MAX.
unistd.h	Constants used when calling *sysconf()* to query POSIX limits (see the sysconf(3) reference page).

It is recommended that the thread−safe options be enabled at compile time using the feature test macro, _POSIX_C_SOURCE (see intro(3) for details). For example, to compile these options, use this command:

```
cc −D_POSIX_C_SOURCE=199506L app.c −llib0 −llib1 ... −lpthread
```

You can use pthreads with a program compiled to any of the supported execution models: *−32* for compatibility with older systems, *−n32* for 64−bit data and 32−bit addressing, or *−64* for 64−bit addressing.

The pthreads functions are defined in the library *libpthread.so*. Link with this library using the *−lpthread* compiler option, which should be the last library on the command line. The compiler chooses the correct library based on the execution model: */usr/lib/libpthread.so*,

/usr/lib32/libpthread.so, and */usr/lib64/libpthread.so.*

Note: A pthread program is a program that links with *libpthread*. Do not link with *libpthread* unless you intend to use the pthread interface, because *libpthread* replaces many standard library functions.

Debugging Pthread Programs

The *dbx* debugger and Workshop Debugger have been extended for use with threaded programs. See the dbx(1M) reference page and the documentation for Workshop Debugger for more details.

Creating Pthreads

You create a pthread by calling *pthread_create()*. One argument to this function is a thread attribute object of type *pthread_attr_t*. You pass a null address to request a thread having default attributes, or you prepare an attribute object to reflect the features you want the thread to have. You can use one attribute object to create many pthreads.

Functions related to attribute objects and pthread creation are summarized in Table 13−3and described in the following sections:

"Initial Detach State"

"Initial Scheduling Scope, Priority, and Policy"

"Thread Stack Allocation"

Table 13−3 Functions for Creating Pthreads

Function	Purpose
pthread_attr_init(3P)	Initialize a *pthread_attr_t* object to default settings.
pthread_attr_setdetachstate(3P)	Set the automatic−detach attribute.
pthread_attr_setinheritsched(3P)	Specify whether scheduling attributes come from the attribute object or are inherited from the creating thread.
pthread_attr_setschedparam(3P)	Set the starting thread priority.
pthread_attr_setschedpolicy(3P)	Set the scheduling policy.
pthread_attr_setscope(3P)	Set the scheduling scope.
pthread_attr_setstacksize(3P)	Set the stack size attribute.
pthread_attr_setguardsize(3P)	Set the stack guard size attribute.
pthread_attr_setstackaddr(3P)	Set the address of memory to use as a stack (when you allocate the stack for the new thread).
pthread_attr_destroy(3P)	Uninitialize a *pthread_attr_t* object.
pthread_create(3P)	Create a new thread based on an attribute object, or with default attributes.

Initial Detach State

Detaching means that the pthreads library frees up resources held by the thread after it terminates (see "Joining and Detaching"). There are three ways to detach a thread:

automatically when the thread terminates

explicitly by calling *pthread_join()*

explicitly by calling *pthread_detach()*

You can use *pthread_attr_setdetachstate()* to specify that a thread should be detached automatically when it terminates. Do this when you know that the thread will not be joined or detached by an explicit function call.

Initial Scheduling Scope, Priority, and Policy

You can specify an initial thread scheduling scope by calling *pthread_attr_setscope()* and passing one of the scope constants (PTHREAD_SCOPE_SYSTEM or PTHREAD_SCOPE_PROCESS) in the *pthread_attr_t* object. By default, process scope is selected and scheduling is performed by the thread runtime, but thread scheduling by the kernel is provided with the system scope attribute. System scope threads run at real−time policy and priority and may be created only by privileged users.

You can specify an initial thread priority in a *struct sched_param* object in memory (the structure is declared in *sched.h*). Set the desired priority in the *sched_priority* field. Pass the structure to *pthread_attr_setschedparam()*.

You can specify an initial scheduling policy by calling *pthread_attr_setschedpolicy()*, passing one of the policy constants SCHED_FIFO or SCHED_RR.

The *pthread_attr_setinheritsched()* function is used to specify, in the attribute object, whether a new thread's scheduling policy and priority should be taken from the attribute object, or whether they should be inherited from the thread that creates the new thread. When you set an attribute object for inheritance, the scheduling policy and priority in the attribute object are ignored.

Scheduling scope, priorities, and policies are described in "Scheduling Pthreads".

Thread Stack Allocation

Each pthread has an execution stack area in memory. By default, *pthread_create()* allocates stack space from dynamic memory, and automatically releases it when the thread terminates.

You use *pthread_attr_setstacksize()* to specify the size of this stack area. You cannot specify a stack size less than a minimum. A pthread process can find the minimum by calling *sysconf()* with _SC_THREAD_STACK_MIN (see the sysconf(3C) reference page).

Threads may overrun their stack area. By default, a thread's stack is created with guard protection, and extra memory is allocated at the overflow end of the stack as a buffer. If an application overflows into this buffer, an exception results (a SIGSEGV signal is delivered to the thread).

The *guardsize* attribute controls the size of the guard area for the created thread's stack and protects against overflow of the stack pointer. The *guardsize* attribute is set using *pthread_attr_setguardsize()*.

Note: Because thread stack space is taken from dynamic memory, the allocation is charged against the process virtual memory limit, not the process stack size limit as you might expect.

Executing and Terminating Pthreads

The functions for managing the progress of a thread are summarized in Table 13−4and described in the following sections:

"Getting the Thread ID"

"Initializing Static Data"

"Setting Event Handlers"

"Terminating a Thread"

"Joining and Detaching"

Table 13−4 Functions for Managing Thread Execution

Function	Purpose
pthread_atfork(3P)	Register functions to handle the event of a *fork()*.
pthread_cancel(3P)	Request cancellation of a specified thread.
pthread_cleanup_push(3P)	Register function to handle the event of thread termination.
pthread_cleanup_pop(3P)	Unregister and optionally call termination handler.
pthread_detach(3P)	Detach a terminated thread.
pthread_exit(3P)	Explicitly terminate the calling thread.
pthread_join(3P)	Wait for a thread to terminate and receive its return value.
pthread_once(3P)	Execute initialization function once only.
pthread_self(3P)	Return the calling thread's ID.
pthread_equal(3P)	Compare two thread IDs for equality.
pthread_setcancelstate(3P)	Permit or block cancellation of the calling thread.
pthread_setcanceltype(3P)	Specify deferred or asynchronous cancellation.
pthread_testcancel(3P)	Permit cancellation to take place, if it is pending.

Getting the Thread ID

Call *pthread_self()* to get the thread ID of the calling thread. A thread can use this thread ID when changing its own scheduling priority, for example (see "Scheduling Pthreads").

Initializing Static Data

Your program may use static data that should be initialized exactly once. The code can be entered by multiple threads, and might be entered concurrently. How can you ensure that only one thread will perform the initialization?

One answer is to create a variable of type *pthread_once_t*, statically initialized to the value PTHREAD_ONCE_INIT. Call *pthread_once(),* passing the addresses of the variable and of an initialization function. The pthreads library ensures that the initialization function is called only once, and that any other threads calling *pthread_once()* for this variable wait until the first thread completes the initialization function. See Example 13−1

Example 13−1 One−Time Initialization

```
pthread_once_t first_time_flag = PTHREAD_ONCE_INIT;
elaborate_struct_t uninitialized; /* thing to initialize */
void elaborate_initializer(void); /* function to do it */
int subroutine(...)
{
   ...
   pthread_once(&first_time_flag, elaborate_initializer);
   ...
}
```

Setting Event Handlers

A thread can establish functions that are called when it terminates and when the process forks.

Call *pthread_cleanup_push()* to register a function that is to be called in the event that the current thread terminates, either by exiting or by cancellation. Call *pthread_cleanup_pop()* to retract this registration and, optionally, to call the handler. These functions are often used in library code, with the push operation done on entry to the library and the pop done upon exit from the library. The push and pop operations are in fact implemented partly as macro code. For this reason, calls to them must be strictly balanced—a pop for each push—and each push/pop pair must appear in a single C lexical scope. A nonstructured jump such as a longjmp (see the setjmp(3) reference page) or goto can cause unexpected results.

Call *pthread_atfork()* to register three handlers related to a UNIX *fork()* call. The first handler executes just before the *fork()* takes place; the second executes just after the *fork()* in the parent process; the third executes just after the *fork()* in the child process.

The *fork()* operation creates a new process with a copy of the calling process's address space, including any locked mutexes or semaphores. Typically, the new process immediately calls *exec()* to replace the address space with a new program. When this is the case, there is no need for *pthread_atfork()* (see the exec(2) and fork(2) reference pages). However, if the new process continues to execute with the inherited address space, including perhaps calls to library code that uses pthreads, it may be necessary for the library code to reinitialize data in the address space of the child process. You can do this in the fork event handlers.

Terminating a Thread

A thread begins execution in the function that is named in the *pthread_create()* call. When it returns from that function, the thread terminates. A thread can terminate earlier by calling *pthread_exit()*. In either case, the thread returns a value of type *void**.

One thread can request early termination of another by calling *pthread_cancel()*, passing the thread ID of the target thread. A thread can protect itself against cancellation using two built−in switches:

> The *pthread_setcancelstate()* function lets you postpone cancellation indefinitely (PTHREAD_CANCEL_DISABLE) or permit cancellation (PTHREAD_CANCEL_ENABLE).
>
> The *pthread_setcanceltype()* function lets you decide when cancellation will take place, if it is allowed at all. Cancellation can happen whenever it is requested (PTHREAD_CANCEL_ASYNCHRONOUS) or only at defined points (PTHREAD_CANCEL_DEFERRED).

When you prevent cancellation by setting PTHREAD_CANCEL_DISABLE, a cancellation request is blocked but remains pending until the thread terminates or changes its cancellation state.

The initial cancellation state of a thread is PTHREAD_CANCEL_ENABLE and the type is PTHREAD_CANCEL_DEFERRED. In this state, a cancellation request is blocked until the thread calls a function that is a defined cancellation point. The functions that are cancellation points are listed in the pthread_setcanceltype(3P) reference page. A thread can explicitly permit cancellation by calling *pthread_testcancel()*.

Joining and Detaching

Sometimes you do not care when threads terminate—your program starts a set of threads, and they continue until the entire program terminates.

In other cases, threads are created and terminated as the program runs. One thread can wait for another to terminated by calling *pthread_join()*, specifying the thread ID. The function does not return until the specified thread terminates. The value the specified thread passed to *pthread_exit()* is returned. At this time, your program can release any resources that you associate with the thread, for example, stack space (see "Thread Stack Allocation").

The *pthread_join()* function also detaches the terminated thread. If your program does not use *pthread_join()*, you must arrange for terminated threads to be detached in some other way. One way is by specifying automatic detachment when the threads are created (see "Initial Detach State"). Another is to call *pthread_detach()* at any time after creating the thread, including after it has terminated.

If your program creates threads and lets them terminate, but does not detach them, resources will be used up and eventually an error will occur when trying to create a thread.

Using Thread−Unique Data

In some designs, especially modules of library code, you need to store data that is both

unique to the calling thread

persistent from one function call to another

Normally, the only data that is unique to a thread is the contents of its local variables on the stack, and these do not persist between calls. However, the pthreads library provides a way to create persistent, thread−unique data. The functions for this are summarized inTable 13−5.

Table 13−5 Functions for Thread−Unique Data

Function	Purpose
pthread_key_create(3P)	Create a key.
pthread_key_delete(3P)	Delete a key.
pthread_getspecific(3P)	Retrieve this thread's value for a key.
pthread_setspecific(3P)	Set this thread's value for a key.

Your program calls *pthread_key_create()* to define a new storage *key*. Once created, a key may be used by all threads to identify a unique key *value.*

Any thread can use *pthread_getspecific()* to retrieve that thread's unique value stored under a key. A thread can fetch only its own value, which is the value stored by this same thread using *pthread_setspecific()*. The initial stored value is NULL.

When you create a key, you can specify a destructor function that is called automatically when a thread terminates. The destructor is called while the key is valid and the key value for the terminating thread is not NULL. The destructor receives the thread's key value as its argument.

Pthreads and Signals

For a general overview of signal concepts and numbers, see "Signals" and the signal(5) reference page. IRIX supports three different signal facilities: BSD signals, SVR4 signals, and POSIX signals.

When you are writing a pthreads program, you should use only the POSIX signal facilities (see "POSIX Signal Facility").

Setting Signal Masks

Each thread has a signal mask that specifies the signals it is willing to receive (see "Signal Blocking and Signal Masks"). In a program that is linked with the pthreads library, this should be changed using *pthread_sigmask()*. Each thread inherits the signal mask of the thread that calls *pthread_create()*. Typically you set an initial mask in the first thread, so that it can be inherited by all other threads.

Note: In IRIX, you can use *sigprocmask()* instead of *pthread_sigmask()*, but it may not be portable to other systems.

When a signal is directed to a specific thread that is blocking the signal, the signal remains pending on the thread until that thread unblocks it. When a signal is directed to a process, it is delivered to the first thread that is not blocking that signal. If all threads block that signal, the signal remains pending on the process until some thread unblocks it or the process terminates.

A thread can find out which signals are pending by calling *sigpending()*. This function returns a mask showing the set of signals pending on the process as a whole or for the calling thread; that is, the signals that could be delivered to the calling thread if they were not blocked.

Setting Signal Actions

When a signal is delivered, some action is taken. You specify what that action should be using the *sigaction()* function. These actions are set on a process−wide basis,*not* individually for each thread. Although each thread has a private signal mask, signal actions are shared with all threads in the process. See "Signal Handling Policies" for details.

Receiving Signals Synchronously

You can design a program to receive signals in a synchronous manner instead of asynchronously. To do this, set a mask that blocks all the signals that are to be received synchronously. Then call one of the following three functions:

sigwait(3)	Suspend until one of a specified set of signals is generated, then return the signal number.
sigwaitinfo(3)	Like *sigwait()*, but returns additional information about the signal.
sigtimedwait(3)	Like *sigwaitinfo()*, but also returns after a specified time has elapsed if no signal is received.

Using these functions you can write a thread that treats signals as a stream of events to be processed. This is generally the safest program model, much easier to work with than the asynchronous model of signal delivery.

Scheduling Pthreads

The pthreads scheduling algorithm is controlled by three variables: a scope, policy, and priority for each thread. These variables are set initially when the thread is created (see "Initial Scheduling Scope, Priority, and Policy"), but policy and priority can be modified while the thread is running.

Contention Scope

The scheduling contention scope of a pthread (see pthread_attr_setscope(3P)) determines the set of threads that it competes against for resources.

System scope threads compete with all other threads on the system and can be created only by privileged users. These threads are used in programs when some form of guaranteed (that is, real−time) response is required. Their scheduling parameters directly affect how the system treats them. In addition to the usual scheduling attributes, they can select a CPU on which to run using the *pthread_setrunon_np()* call.

Process scope threads compete within the process and their scheduling attributes are used by the pthread library to select which threads to run on a pool of kernel entities. The size of the pool is determined dynamically, but may be influenced using the *pthread_setconcurrency()* call.

Process scope threads generally require fewer resources than system scope threads because they can share kernel resources. The kernel entities themselves share a common set of scheduling attributes which privileged users can change using the process scheduling interfaces (see sched_setscheduler(2) and sched_setparam(2)). For further details, see the pthreads(5) reference page.

The functions used in scheduling are summarized in Table 13−6and described in the following sections:

"Scheduling Policy"

"Scheduling Priority"

Table 13−6 Functions for Schedule Management

Function	Purpose
pthread_getschedparam(3P)	Get a thread's policy and priority.
pthread_setschedparam(3P)	Set a thread's policy and priority.
sched_get_priority_max(3C)	Return the maximum priority value.
sched_get_priority_min(3C)	Return the minimum priority value.
sched_yield(2)	Relinquish the processor.
pthread_setconcurrency(3P)	Modify concurrency level.
pthread_getconcurrency(3P)	Check the concurrency level.
pthread_setrunon_np(3P)	Select a CPU to run a system scope thread.
pthread_getrunon_np(3P)	Query a named CPU's affinity.

Scheduling Policy

There are two scheduling policies in this implementation: first−in−first−out (SCHED_FIFO) and the default round−robin (SCHED_RR). SCHED_FIFO and SCHED_RR are similar. The round−robin scheduler ensures that after a thread has used a certain maximum amount of time, it is moved to the end of the queue of threads of the same priority, and can be preempted by other threads.

The details of scheduling are discussed in the pthread_attr_setschedpolicy(3P) reference page.

Scheduling Priority

Threads are ordered by priority values, with a small number representing a low priority, and a larger number representing a higher priority. Threads with higher priorities are chosen to execute before threads with lower priorities.

The *sched_get_priority_max()* and *sched_get_priority_min()* functions return the highest and lowest priority numbers for a given policy. There are at least 32 priority values and the lowest is greater than or equal to 0.

A thread can set another's priority and scheduling policy, using *pthread_setschedparam()*. A simple function to set a specified priority on the current thread is shown in Example 13−2

Example 13−2 Function to Set Own Priority

```
#include <sched.h> /* struct sched_param */
void setMyPriority(int newP)
{
   pthread_t myTid = pthread_self();
   int policy;
   struct sched_param sp;
   (void) pthread_getschedparam(myTID,&policy,&sp);
   sp.sched_priority = newP;
   (void) pthread_setschedparam(myTID,policy,&sp);
}
```

Synchronizing Pthreads

Threads using a common address space must cooperate and coordinate their use of shared variables. IRIX provides many mechanisms for coordinating threads, including:

POSIX semaphores for general coordination and resource management (see "POSIX Facilities for Mutual Exclusion").

POSIX or SVR4 message queues (see Chapter 6, "Message Queues").

POSIX mutex objects, which allow threads to gain exclusive use of a shared variable (see "Mutexes").

POSIX condition variables, which allow a thread to wait when a controlling predicate is false (see "Condition Variables").

POSIX read−write locks, which allow one thread exclusive access to locked data to write it or read access to locked data for several threads (see "Read−Write Locks").

IRIX semaphores and locks.

SVR4 semaphores.

Tip: Synchronization between processes (such as POSIX process−shared mechanisms, IRIX IPC, and SVR4 IPC) is more costly than synchronization between threads (POSIX process−private mechanisms). So where possible, use the process−private mechanisms.

Mutexes

A mutex is a software object that arbitrates the right to modify some shared variable, or the right to execute a critical section of code. A mutex can be owned by only one thread at a time; other threads trying to acquire it wait. Mutexes are intended to be lightweight and owned only for a short time.

Preparing Mutex Objects

When a thread wants to modify a variable that it shares with other threads, or execute a critical section, the thread claims the associated mutex. This can cause the thread to wait until it can acquire the mutex. When the thread has finished using the shared variable or critical code, it releases the mutex. If two or more threads claim the mutex at once, one acquires the mutex and continues, while the others are blocked until the mutex is released.

A mutex has attributes that control its behavior. The pthreads library contains several functions used to prepare a mutex for use. These functions are summarized in Table 13−7.

Table 13−7 Functions for Preparing Mutex Objects

Function	Purpose
pthread_mutexattr_init(3P)	Initialize a *pthread_mutexattr_t* with default attributes.
pthread_mutexattr_destroy(3P)	Uninitialize a *pthread_mutexattr_t*.
pthread_mutexattr_getprotocol(3P)	Query the priority protocol.
pthread_mutexattr_setprotocol(3P)	Set the priority protocol choice.
pthread_mutexattr_getprioceiling(3P)	Query the minimum priority.
pthread_mutexattr_setprioceiling(3P)	Set the minimum priority.
pthread_mutexattr_getpshared(3P)	Query the process−shared attribute.
pthread_mutexattr_setpshared(3P)	Set the process−shared attribute.
pthread_mutexattr_gettype(3P)	Get the mutex type.
pthread_mutexattr_settype(3P)	Set the mutex type.
pthread_mutex_init(3P)	Initialize a mutex object.
pthread_mutex_destroy(3P)	Uninitialize a mutex object.

A mutex must be initialized before use. You can do this in one of three ways:

Static assignment of the constant PTHREAD_MUTEX_INITIALIZER.

Calling *pthread_mutex_init()* passing NULL instead of the address of a mutex attribute object.

Calling *pthread_mutex_init()* passing a *pthread_mutexattr_t* object that you have set up with attribute values.

The first two methods initialize the mutex to default attributes.

Four attributes can be set in a *pthread_mutexattr_t*. You can set the priority inheritance protocol using *pthread_mutexattr_setprotocol()* to one of three values:

PTHREAD_PRIO_NONE	The mutex has no effect on the thread that acquires it. This is the default.
PTHREAD_PRIO_PROTECT	The thread holding the mutex runs at a priority at least as high as th highest priority of any mutex that it currently holds.
PTHREAD_PRIO_INHERIT	The thread holding the mutex runs at a priority at least as high as th highest priority of any thread blocked on that mutex.

If a thread acquires a mutex and then is suspended (for example, because its time slice is up), other threads can be blocked waiting for the mutex. The PTHREAD_PRIO_PROTECT protocol prevents this. Using *pthread_mutexattr_setprioceiling()*, you set a priority higher than normal for the mutex. A thread that acquires the mutex runs at this higher priority while it holds the mutex.

Another problem is that when a low−priority thread has acquired a mutex, and a thread with higher priority claims the mutex and is blocked, a "priority inversion" takes place—a higher−priority thread is forced to wait for one of lower priority. The PTHREAD_PRIO_INHERIT protocol prevents this—when a thread of higher priority blocks, the thread holding the mutex has its priority boosted during the time it holds the mutex.

Tip: PTHREAD_PRIO_NONE uses a faster code path than the other two priority options for mutexes.

By default, only threads within a process share a mutex. Using *pthread_mutexattr_setpshared()*, you can allow any thread (from any process) with access to the mutex memory location to use the mutex. Enable mutex sharing by changing the default PTHREAD_PROCESS_PRIVATE attribute to PTHREAD_PROCESS_SHARED.

Note: The PTHREAD_PRIO_INHERIT attribute is not available with *pthread_mutexattr_setpshared()*.

By default, no error checking is performed on threads that attempt to use a mutex. For example, a thread that attempts to lock a mutex that it already owns deadlocks. Using *pthread_mutexattr_settype()* with PTHREAD_MUTEX_ERRORCHECK allows you to have the lock call return an error instead. If recursive mutexes are required, PTHREAD_MUTEX_RECURSIVE enables recursive mutexes.

Using Mutexes

The functions for claiming, releasing, and using mutexes are summarized in Table 13−8.

Table 13−8 Functions for Using Mutexes

Function	Purpose
pthread_mutex_lock(3P)	Claim a mutex, blocking until it is available.
pthread_mutex_trylock(3P)	Test a mutex and acquire it if it is available, else return an error.
pthread_mutex_unlock(3P)	Release a mutex.
pthread_mutex_getprioceiling(3P)	Query the minimum priority of a mutex.
pthread_mutex_setprioceiling(3P)	Set the minimum priority of a mutex.

To determine where mutexes should be used, examine the memory variables and other objects (such as files) that can be accessed from multiple threads. Create a mutex for each set of shared objects that are used together. Ensure that the code acquires the proper mutex before it modifies the shared objects. You acquire a mutex by calling *pthread_mutex_lock()*, and release it with *pthread_mutex_unlock()*. When a thread must not be blocked, it can use *pthread_mutex_trylock()* to test the mutex and lock it only if it is available.

Condition Variables

A condition variable provides a way in which a thread can wait for an event (or condition) defined by the program, to be satisfied. Condition variables use mutexes to synchronize the wait and wakeup operations.

Preparing Condition Variables

Like mutexes and threads themselves, condition variables are supplied with a mechanism of attribute objects (*pthread_condattr_t* objects) and static and dynamic initializers. (Only the condition variable for the process−shared attribute can be initialized in this implementation.) The functions for initializing one are summarized in Table 13−9.

Table 13–9 Functions for Preparing Condition Variables

Function	Purpose
pthread_condattr_init(3P)	Initialize a *pthread_condattr_t* to default attributes.
pthread_condattr_destroy(3P)	Uninitialize a *pthread_condattr_t*.
pthread_condattr_getpshared(3P)	Get the process–shared attribute.
pthread_condattr_setpshared(3P)	Set the process–shared attribute.
pthread_cond_init(3P)	Initialize a condition variable based on an attribute object.
pthread_cond_destroy(3P)	Uninitialize a condition variable.

A condition variable must be initialized before use. You can do this in one of three ways:

Static assignment of the constant PTHREAD_COND_INITIALIZER.

Calling *pthread_cond_init()* passing NULL instead of the address of an attribute object.

Calling *pthread_cond_init()* passing a *pthread_condattr_t* object that you have set up with attribute values.

The first two methods initialize the variable to default attributes.

By default, only threads within a process share a condition variable. Using *pthread_condattr_setpshared()*, you can allow any thread (from any process) with access to the condition variable memory location to use the condition variable. Enable condition variable sharing by changing the default PTHREAD_PROCESS_PRIVATE attribute to PTHREAD_PROCESS_SHARED.

Using Condition Variables

A condition variable is a software object that represents a test of a Boolean condition. Typically the condition changes because of a software event such as "other thread has supplied data." A thread establishes that it needs to wait by first evaluating the condition. The thread that satisfies the condition signals the condition variable, releasing one or all threads that are waiting.

For example, a thread might acquire a mutex that represents a shared resource. While holding the mutex, the thread finds that the shared resource is not complete. The thread does three things:

Wait, giving up the mutex so that some other thread can renew the shared resource.

Wait until the condition is signalled.

Wake–up, re–acquiring the mutex for the shared resource and rechecking the condition.

These three actions are combined into one using a condition variable. The functions used with condition variables are summarized in Table 13–10.

Table 13–10 Functions for Using Condition Variables

Function	Purpose
pthread_cond_wait(3P)	Wait on a condition variable.
pthread_cond_timedwait(3P)	Wait on a condition variable, returning with an error after a time limit expires.
pthread_cond_signal(3P)	Signal that an awaited event has occurred, releasing at least one waiting thread.
pthread_cond_broadcast(3P)	Signal that an awaited event has occurred, releasing all waiting threads.

The *pthread_cond_wait()* and *pthread_cond_timedwait()* functions require both a condition variable and a mutex that is owned by the calling thread. The mutex is released and the wait begins. When the event is signalled (or the time limit expires), the mutex is reacquired, as if by a call to *pthread_mutex_lock()*.

The POSIX standard explicitly warns that it is possible in some cases for a conditional wait to return before the event has been signalled. For this reason, a conditional wait should always be coded in a loop that tests the shared resource for the needed status. These principles are suggested in the code in Example 13−3, which is modeled after an example in the POSIX 1003.1c standard.

Example 13−3 Use of Condition Variables

```
#include <assert.h>
#include <pthread.h>
typedef int listKey_t;
typedef struct element_s { /* list element */
   listKey_t key;
   struct element_s *next;
   int busyFlag;
   pthread_cond_t notBusy; /* event of no-longer-in-use */
} element_t;
typedef struct listHead_s { /* list head and mutex */
   pthread_mutex_t mutList; /* right to modify the list */
   element_t *head;
} listHead_t;
/*
|| Internal function to find an element in a list, returning NULL
|| if the key is not in the list.
|| A returned element could be in use by another thread (busy).
|| The caller is assumed to hold the list mutex, otherwise
|| the returned value could be made invalid at any time.
*/
static element_t *scanList(listHead_t* lp, listKey_t key)
{
   element_t *ep;
   for (ep=lp->head; (ep) ; ep=ep->next)
   {
      if (ep->key == key) break;
   }
   return ep;
}
/*
|| Public function to find a key in a list, wait until the element
|| is no longer busy, mark it busy, and return it.
*/
element_t *getFromList(listHead_t* lp, listKey_t key)
{
   element_t *ep;
```

```
   pthread_mutex_lock(&lp->mutList); /* lock list against changes */
   while ((ep=scanList(lp,key)) && (ep->busyFlag))
   {
      pthread_cond_wait(&ep->notBusy, &lp->mutList); /* (A) */
   }
   if (ep) ep->busyFlag = 1;
   pthread_mutex_unlock(&lp->mutList);
   return ep;
}
/*
|| Public function to release an element returned by getFromList().
*/
void freeInList(listHead_t* lp, element_t *ep)
{
   assert(ep->busyFlag);
   pthread_mutex_lock(&lp->mutList); /* lock list to prevent races *
/
   ep->busyFlag = 0;
   pthread_cond_signal(&ep->notBusy);
   pthread_mutex_unlock(&lp->mutList);
}
/*
|| Public function to delete a list element returned by getFromList(
).
*/
void deleteInList(listHead_t* lp, element_t *ep)
{
   element_t **epp;
   assert(ep->busyFlag);
   pthread_mutex_lock(&lp->mutList);
   for (epp = &lp->head; ep != *epp; epp = &((*epp)->next))
   { /* finding anchor of *ep in list */ }
   *epp = ep->next; /* remove *ep from list */
   ep->busyFlag = 0;
   pthread_cond_broadcast(&ep->notBusy);
   pthread_mutex_unlock(&lp->mutList);
   pthread_cond_destroy(&ep->notBusy);
   free(ep);
}
```

The functions in Example 13−3implement part of a simple library for managing lists. In a list head, *mutList* is a mutex object that represents the right to modify any part of the list. The elements of a list can be "busy," that is, in use by some thread. An element that is busy has a nonzero *busyFlag* field.

The *getFromList()* function looks up an element in a specified list, makes that element busy, and returns it. The function begins by acquiring the list mutex. This ensures that the list cannot change while the function is searching the list, and makes it legitimate for the function to change the busy flag in an element.

When it finds the element, the function might discover that the element is already busy. In this case, it must wait for the event "element is no longer busy," which is represented by the condition variable *notBusy* in the element. In order to wait for this event, *getFromList()* calls *pthread_cond_wait()* passing its list mutex and the condition variable (point "(A)" in the code). This releases the list mutex so that other threads can acquire the list and do their work on other elements.

When any thread wants to release the use of a list element, it calls *freeInList().* After clearing the busy flag in the list element, *freeInList()* announces that the event "element is no longer busy" has occurred, by calling *pthread_cond_signal()*.

This call releases a thread that is waiting at point "(A)." If there is more than one thread waiting for the same element, the first in priority order is released. The released thread re−acquires the list mutex and resumes execution. The first thing it does is repeat its search of the list for the desired key and, on finding the element again, test it again for busyness. This repetition is needed because it is possible to get spurious returns from a condition variable.

When a thread wants to delete a list element, it gets the list element by calling *getFromList()*. This ensures that the element is busy, so no other thread is using it. Then the thread calls *deleteInList()*. This function changes the list, so it begins by acquiring the list mutex. Then it can safely modify the list pointers. It scans up the list looking for the pointer that points to the target element. It removes the target element from the list by copying its *next* field to replace the pointer to the target element.

With the element removed from the list, *deleteInList()* calls *pthread_cond_broadcast()* to wake up all threads—not just the first thread—that might be waiting for the element to become nonbusy. Each of these threads resumes execution at point "(A)" by attempting to re−acquire the list mutex. However, *deleteInList()* is still holding the list mutex. The mutex is released; then the other threads can resume execution following point "(A)," but this time when they search the list, the desired key is no longer found.

Meanwhile, *deleteInList()* uses *pthread_cond_destroy()* to release any memory that the pthreads library might have associated with the condition variable, before releasing the list element object itself.

Read−Write Locks

A read−write lock is a software object that gives one thread the right to modify some data, or multiple threads the right to read that data. A read−write lock can be owned for write or for read. If acquired for write, only one thread can own it and other threads must wait. If acquired for read, other threads wishing to acquire it for write must wait, but multiple readers can own the lock at the same time.

Preparing Read−Write Locks

When a thread wants to modify or read data shared by several threads, the thread claims the associated lock. This can cause the thread to wait until it can acquire the lock. When the thread has finished reading or writing the shared data, it releases the lock.

A read−write lock has attributes that control its behavior. The pthreads library contains several functions used to prepare a lock for use. These functions are summarized in Table 13−11.

Table 13−11 Functions for Preparing Read−Write Locks

Function	Purpose
pthread_rwlockattr_init(3P)	Initialize a *pthread_rwlockattr_t* with default

	attributes.
pthread_rwlockattr_destroy(3P)	Uninitialize a *pthread_rwlockattr_t*.
pthread_rwlockattr_getpshared(3P)	Query the process−shared attribute.
pthread_rwlockattr_setpshared(3P)	Set the process−shared attribute.
pthread_rwlock_init(3P)	Initialize a rwlock object based on a *pthread_rwlockattr_t*.
pthread_rwlock_destroy(3P)	Uninitialize a read−write lock object.

A read−write lock must be initialized before use. You can do this in one of three ways:

Static assignment of the constant PTHREAD_RWLOCK_INITIALIZER.

Calling *pthread_rwlock_init()* passing NULL instead of the address of a read−write lock attribute object.

Calling *pthread_rwlock_init()* passing a *pthread_rwlockattr_t* object that you have set up with attribute values.

The first two methods initialize the read−write lock to default attributes.

By default, only threads within a process share a read−write lock. Using *pthread_rwlockattr_setpshared()*, you can allow any thread (from any process) with access to the read−write lock memory location to claim the read−write lock. Enable read−write lock sharing by changing the default PTHREAD_PROCESS_PRIVATE attribute to PTHREAD_PROCESS_SHARED.

Using Read−Write Locks

The functions for claiming, releasing, and using read−write locks are summarized inTable 13−12

Table 13−12 Functions for Using Read−Write Locks

Function	Purpose
pthread_rwlock_wrlock(3P)	Apply a write lock, blocking until it is available.
pthread_rwlock_trywrlock(3P)	Test a write lock and acquire it if it is available, else return an error.
pthread_rwlock_rdlock(3P)	Apply a read lock, blocking until it is available.
pthread_rwlock_tryrdlock(3P)	Test a read lock and acquire it if it is available, else return an error.
pthread_rwlock_unlock(3P)	Release a read or a write lock.

To determine where read−write locks should be used, examine the memory variables and other objects (such as files) that can be accessed from multiple threads. Create a read lock for each set of shared objects that are used together. Ensure that the code acquires the write lock before it modifies the shared objects. You acquire a write lock by calling *pthread_rwlock_wrlock()*, and release it with *pthread_rwlock_unlock()*. A read lock is acquired by calling *pthread_rwlock_rdlock()*, and released with *pthread_rwlock_unlock()*. When a thread must not be blocked, it can use *pthread_rwlock_trywrlock()* or *pthread_rwlock_tryrdlock()* to test the lock and apply it only if it is available.

Chapter 14

Message−Passing Parallelism

In a message−passing model, your parallel application consists of multiple, independent processes, each with its own address space. Each process shares data and coordinates with the others by passing messages, using a formal interface. The formal interface makes the program independent of the medium over which the message go. The processes of the program can be in a single computer, with messages moving via memory−to−memory copy; but it is possible to distribute the program in different machines, with messages passing over a network.

The Message−Passing Toolkit package supports three libraries on which you can build a message−passing application. The Cray Shared−Memory (SHMEM) library supports message passing in a single system. Message−Passing Interface (MPI) and Parallel Virtual Machine (PVM) models support distribution. High−level overviews of these are given under"Message−Passing Models".

Choosing a Message−Passing Model

There are five considerations in choosing among the three message−passing models: compatibility, portability, scope, latency, and bandwidth.

Compatibility	If you are starting with an existing program that uses one of the three models, or i want to reuse code from such a program, or if you personally are highly familiar v of the three, you will likely choose that model in order to minimize development t
Portability	The SHMEM library is portable among all Silicon Graphics/Cray systems, includi IRIX and UNICOS/MK. However, it is not supported on systems of other types. E MPI and PVM are industry standard libraries that are widely available in public−c implementations.
Scope	The SHMEM library can be used only within a single multiprocessor such as Cray or an Origin2000. You can use MPI or PVM to distribute a program across all noc an Silicon Graphics Array, or across a heterogeneous network.
Latency	Latency, the start−up delay inherent in sending any one message of any size, is the shortest in SHMEM. MPI within a single system is a close second (both use memory−to−memory copy). MPI latency across an Array using the Silicon Graphics−proprietary HIPPI Bypas is an order of magnitude greater. MPI or PVM latency using ordinary HIPPI or TCP/IP is greater still.
Bandwidth	The rate at which the bits of a message are sent is the highest in SHMEM and MP a single system. MPI bandwidth over a HIPPI link is next, followed by PVM.

If you require the highest performance within a *single* Cray or Silicon Graphics system, use SHMEM. For the highest performance in an Array system linked with HIPPI, use MPI. Use PVM only when compatibility or portability is an overriding consideration.

Choosing Between MPI and PVM

When your program must be able to use the resources of multiple systems, you choose between MPI and PVM. In many ways, MPI and PVM are similar:

Each is designed, specified, and implemented by third parties that have no direct interest in selling hardware.

- Support for each is available over the Internet at low or no cost.
- Each defines portable, high−level functions that are used by a group of processes to make contact and exchange data without having to be aware of the communication medium.
- Each supports C and Fortran 77.
- Each provides for automatic conversion between different representations of the same kind of data so that processes can be distributed over a heterogeneous computer network.

Another difference between MPI and PVM is in the support for the "topology" (the interconnect pattern: grid, torus, or tree) of the communicating processes. In MPI, the group size and topology are fixed when the group is created. This permits low−overhead group operations. The lack of run−time flexibility is not usually a problem because the topology is normally inherent in the algorithmic design. In PVM, group composition is dynamic, which requires the use of a "group server" process and causes more overhead in common group−related operations.

Other differences are found in the design details of the two interfaces. MPI, for example, supports asynchronous and multiple message traffic, so that a process can wait for any of a list of message−receive calls to complete and can initiate concurrent sending and receiving. MPI provides for a "context" qualifier as part of the "envelope" of each message. This permits you to build encapsulated libraries that exchange data independently of the data exchanged by the client modules. MPI also provides several elegant data−exchange functions for use by a program that is emulating an SPMD parallel architecture.

PVM is possibly more suitable for distributing a program across a heterogeneous network that includes both uniprocessors and multiprocessors, and includes computers from multiple vendors. When the application runs in the environment of a Silicon Graphics Array system, MPI is the recommended interface.

Differences Between PVM and MPI

This section discusses the main differences between PVM and MPI from the programmer’s perspective, focusing mainly on PVM functions that are not available in MPI.

Although to a large extent the library calls of MPI and PVM provide similar functionality, some PVM calls do not have a counterpart in MPI, and vice versa. Additionally, the semantics of some of the equivalent calls are inherently different for the two libraries (owing, for example, to the concept of dynamic groups in PVM). Hence, the process of converting a PVM program into an MPI program can be straightforward or complicated, depending on the particular PVM calls in the program and how they are used. For many PVM programs, conversion is straightforward.

In addition to a message−passing library, PVM also provides the concept of a*parallel virtual machine session*. A user starts this session before invoking any PVM programs; in other words, PVM provides the means to establish a parallel environment from which a user invokes a parallel program.

Additionally, PVM includes a *console*, which is useful for monitoring and controlling the states of the machines in the *virtual machine* and the state of execution of a PVM job. Most PVM console commands have corresponding library calls.

The MPI standard does not provide mechanisms for specifying the initial allocation of processes to an MPI computation and their binding to physical processors. Mechanisms to do so at load time or run time are left to individual vendor implementations. However, this difference between the two paradigms is not, by itself, significant for most programs, and should not affect the port from PVM to MPI.

The chief differences between the current versions of PVM and MPI libraries are as follows:

PVM supports dynamic spawning of tasks, whereas MPI does not.

PVM supports dynamic process groups; that is, groups whose membership can change dynamically at any time during a computation. MPI does not support dynamic process groups.

MPI does not provide a mechanism to build a group from scratch, but only from other groups that have been defined previously. Closely related to groups in MPI are communicators, which specify the communication context for a communication operation and an ordered process group that shares this communication context. The chief difference between PVM groups and MPI communicators is that any PVM task can join/leave a group independently, whereas in MPI all communicator operations are collective.

A PVM task can add or delete a host from the virtual machine, thereby dynamically changing the number of machines a program runs on. This is not available in MPI.

A PVM program (or any of its tasks) can request various kinds of information from the PVM library about the collection of hosts on which it is running, the tasks that make up the program, and a task's parent. The MPI library does not provide such calls.

Some of the collective communication calls in PVM (for instance, *pvm_reduce()*) are nonblocking. The MPI collective communication routines are not required to return as soon as their participation in the collective communication is complete.

PVM provides two methods of signaling other PVM tasks: sending a UNIX signal to another task, and notifying a task about an event (from a set of predefined events) by sending it a message with a user−specified tag that the application can check. A PVM call is also provided through which a task can kill another PVM task. These functions are not available in MPI.

A task can leave/unenroll from a PVM session as many times as it wants, whereas an MPI task must initialize/finalize exactly once.

A PVM task need not explicitly enroll: the first PVM call enrolls the calling task into a PVM session. An MPI task must call *MPI_Init()* before calling any other MPI routine and it must call this routine only once.

A PVM task can be registered by another task as responsible for adding new PVM hosts, or as a PVM resource manager, or as responsible for starting new PVM tasks. These features are not available in MPI.

A PVM task can multicast data to a set of tasks. As opposed to a broadcast, this multicast does not require the participating tasks to be members of a group. MPI does not have a routine to do multicasts.

PVM tasks can be started in debug mode (that is, under the control of a debugger of the user's choice). This capability is not specified in the MPI standard, although it can be provided on top of MPI in some cases.

In PVM, a user can use the *pvm_catchout()* routine to specify collection of task outputs in various ways. The MPI standard does not specify any means to do this.

PVM includes a receive routine with a timeout capability, which allows the user to block on a receive for a user−specified amount of time. MPI does not have a corresponding call.

PVM includes a routine that allows users to define their own receive contexts to be used by subsequent PVM receive routines. Communicators in MPI provide this type of functionality to a limited extent.

On the other hand, MPI provides several features that are not available in PVM, including a variety of communication modes, communicators, derived data types, additional group management facilities, and virtual process topologies, as well as a larger set of collective communication calls.