Chapter 3

Models of Parallel Computation

Silicon Graphics, Inc., makes multiprocessor computer systems. You can use any of several programming models to exploit the parallel capabilities of the hardware. This chapter reviews the parallel programming models, supplying enough information that you can select one model. Pointers to more detailed documentation of each model are included. The major topics are:

- "Parallel Hardware and Programming Models" on page 123 provides a quick survey of the programming models and their relationship to the hardware.
- "Using Statement-Level Parallelism" on page 131 discusses using fine-grained parallel execution in Fortran and C.
- "Using POSIX Threads" on page 141 gives an overview of the pthreads implementation for IRIX 6.2.
- "Using Process-Level Parallelism" on page 136 provides an overview of the use of coordinated UNIX processes for parallel execution.
- "Using MPI and PVM" on page 167 compares two interfaces for distributed, process-level, parallelism.

Parallel Hardware and Programming Models

Silicon Graphics makes a variety of multiprocessor systems, including

- The CHALLENGE/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 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.

Most programs that run on a multiprocessor execute as if they 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. Each model is designed around a different set of assumptions about the hardware, and in particular about the memory system available. Each model is implemented using a different library of code linked with your program.

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 possible to restrict and assign the available CPUs of a multiprocessor so that only one CPU is available to execute a given program, even when the program is meant for parallel execution. This can arise as a brief transient condition as IRIX dynamically assigns CPUs to programs, or it can arise through the operator using commands such as *mpadmin* (see the mpadmin(1) reference page).
- It is easier to debug a parallel program by running it in the more predictable environment of a uniprocessor.

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 “Using Statement-Level Parallelism” on page 131).

Memory Systems

For parallel execution, the key memory issue 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 very 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 effect of a single, common memory is that processes running in different CPUs can share pages of 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. 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.

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. As a result:

- 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 through a software socket and 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.

Types of Parallel Models

The IRIX system supports a variety of parallel programming models. You can compare these models 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 3-1.

Table 3-1 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.
Remote Procedure Call (RPC)	Process	Memory copy within node; UDP or TCP network between nodes.

Table 3-1 (continued) Comparing Parallel Models

Model	Granularity	Communication
Portable Virtual Memory (PVM)	Process	Memory copy within node; TCP socket between nodes.
Message-Passing (MPI)	Process	Memory copy within node; special HIPPI interface between nodes.

Process-Level Parallelism

A default UNIX process consists of an address space, a varied set of 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 so as 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 “Distributed Computation Models” on page 130.)

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 concurrently. 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 state values with its parent process (the sproc(2) reference page details how this sharing is specified).

In particular, a lightweight process 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” on page 128). However, a lightweight process differs from a true thread in two significant ways:

- A lightweight process still has a full set of UNIX state values, including its own signal handlers. Some of these values, 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 a context switch between lightweight processes, even when they share the same address space, is time-consuming.

The library support for statement-level parallelism is based on the use of lightweight processes, coordinating their activities through semaphores (see “Statement-Level Parallelism” on page 129 and “Using IRIX Semaphores” on page 45).

Thread-Level Parallelism

A thread is an independent execution state in the context of a program. The concept of a thread is well-known, and the word “thread” is casually used for any kind of independent execution state. However, the best-known formal definition of threads and their operation is found in IEEE standard 1003.1c-1995, “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 process ID, effective user ID, signal handlers, 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. Since a process carries a large amount of state information, 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 almost entirely in the user domain, without kernel assistance. Since threads have less state information, thread scheduling is faster than process scheduling.

The POSIX standard for multithreaded programs (IEEE standard 1003.1c) is supported by IRIX 6.2 after the following patches are applied: 1361, 1367, and 1389. The use of POSIX threads is discussed further under “Using POSIX Threads” on page 141.

In addition, the Silicon Graphics, Inc. implementation of the Ada 95 language includes support for multitasking Ada programs. The current implementation of Ada uses an early version of the pthreads library. The next release of Ada will use 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 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 or a single node in a POWER CHALLENGEarray.

Distributed Computation Models

You can “distribute” a computation by putting parts of the work on different computers. Three models of distributed execution are supported by Silicon Graphics, Inc. systems. Each is a formal, abstract model for distributing a computation across the nodes of a multiple-memory system, without having to reflect the system configuration in the source code. The three programming models are:

- Message-Passing Interface (MPI)
- Portable Virtual Memory (PVM)
- Remote Procedure Call (RPC) interface

Each of the three has its particular strengths and weaknesses.

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).

A highly tuned, efficient implementation of MPI is included with the Array 2.0 software support for Array systems such as the POWER CHALLENGEarray. MPI is the recommended parallel model for use with Array products.

MPI is discussed in more detail under “Using MPI and PVM” on page 167.

Portable 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.

An implementation of PVM is included with the Array 2.0 software for Silicon Graphics, Inc. Array systems. PVM has a better ability to deal with a heterogenous computer network than MPI does. In every other way, MPI is preferable. When the application runs in the context of a single Array system, an MPI design has better performance.

PVM is discussed in more detail under “Using MPI and PVM” on page 167.

Remote Procedure Call (RPC) Model

RPC is a standard programming interface originally developed at Sun Microsystems, Inc. and used as the basis of Sun’s Network File System (NFS) standard. RPC is used extensively within the IRIX system (and in most current UNIX implementations) to provide NFS and network management services.

The purpose of the RPC interface is to distribute services across a network, so that one program can easily supply a service to all others. An RPC server program registers the services it can provide with RPC. A client program anywhere in the network can issue a remote procedure call for a registered service, and the RPC interface takes care of locating the server program, invoking its service, and returning the result values to the caller.

RPC by itself does not support concurrent execution. A remote procedure call, like a local procedure call, is synchronous; that is, the caller is blocked until the called procedure completes its work. RPC is a method of distributing a computation over a network, not a method of parallel execution. However, RPC can be combined with other parallel execution models. For example, a thread or lightweight process can issue remote procedure calls.

RPC libraries are included in IRIX. For an overview of RPC programming, see the *IRIX Network Programming Guide*. For further details, refer to the rpc(3R) reference page.

Using Statement-Level Parallelism

As noted under “Statement-Level Parallelism” on page 129, you can use statement-level parallelism in three language packages: Fortran 77, Fortran 90, and C. This type of parallelism 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.

The parallel features of all three of these languages are documented in detail in the manuals listed in Table 3-2.

Table 3-2 Documentation for Statement-Level Parallel Products

Manual	Document Number	Contents
IRIS POWER C User's Guide	007-0702-0x0	Use of the IRIS POWER C Analyzer, including all pragmas.
MIPSpro Fortran 77 Programmer's Guide	007-2361-00x	General use of Fortran 77, including parallelizing assertions and directives.
MIPSpro Power Fortran 77 Programmer's Guide	007-2363-00x	Use of the Power Fortran source analyzer to place directives automatically.
MIPSpro Fortran 90 Programmer's Guide	007-2761-001	General use of Fortran 90, including parallelizing assertions and directives.
MIPSpro Power Fortran 90 Programmer's Guide	007-2760-001	Use of the Power Fortran 90 source analyzer to place directives automatically.

In addition to these products from Silicon Graphics, Inc., 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.infomall.org/apri 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 is 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 maintain 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, you 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 each of the languages uses IRIX lightweight processes to implement parallel execution (see "Process-Level Parallelism" on page 127).

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, and one process executes the opening passage of the program. When execution reaches a parallel section, the run-time support unblocks as many processes as necessary. Each one begins to execute the same block of statements. The processes share global variables, while each has 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 section, it returns to the run-time library code, where it picks up another portion of work if any work remains, or simply blocks until the next time it is needed. At the end of the parallel section, all extra processes are blocked 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 that there is one for each CPU in the system. That default is often too high, since at least one of the CPUs is normally dedicated to other work.

The run-time library checks an environment variable, MPC_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 3-3.

Table 3-3 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 3-3 are 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 the Degree of Parallelism" on page 134).

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 so as 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.

Using Process-Level Parallelism

Software products from Silicon Graphics, Inc. use process-level parallelism in order to exploit the power of single-memory multiprocessors. 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” on page 133).

Parallelism in Real-Time Applications

In real-time programs such as aircraft or vehicle simulators, separate processes are used to distribute the work of the simulation 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* (007-2499-00x). 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.

Process Synchronization and Share Groups

IRIX provides a variety of features to make it possible to build an application consisting of multiple, lightweight processes. In general, a lightweight process is one that shares the address space of its parent process (see "Process-Level Parallelism" on page 127). The parent process and the sibling processes that it creates are a *share group*. IRIX provides special services to share groups.

Process Communication and Coordination

IRIX supports a wide range of interprocess communication (IPC) facilities. These are discussed in detail in Chapter 2, "Interprocess Communication." They include:

- The use of shared arenas for common memory (see "Initializing a Shared Arena" on page 36 and the following topics).
- IRIX semaphores ("Using IRIX Semaphores" on page 45), locks ("Using Locks" on page 47) and barriers ("Using Barriers" on page 49) for coordination and mutual exclusion.

  The IRIX semaphores and locks are especially tuned to efficiency in a multiprocessor environment.
- Portable support for interprocess messages, shared memory, and semaphores ("System V IPC Overview" on page 51).

The REACT™/Pro product includes a number of examples of real-time programs that use IRIX IPC features. The *REACT Real-Time Programmer's Guide* includes the source code of additional examples.

Process Creation

The **sproc()** and **sprocsp()** functions create a lightweight process (see the sproc(2) reference page). The difference between the 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()**.

In the traditional **fork()** call, the new process executes the identical program text as the old one; that is, both processes "return" from **fork()** and you distinguish them by the return code, which is 0 in the child process and the new process ID in the parent.

The **sproc()** call differs in that it takes as an argument the address of the function that the new process should execute. Often, each child process has a particular role to play, and the function represents that work.

Another design is possible. In some applications, you may have to manage a flow of many, relatively short, activities which 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 3-1 shows the general approach.

Example 3-1 Partial Code to Manage a Pool of Processes

```
typedef void (*func)(void *arg) workFunc;
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 func */
   usema_t *sprocDone;           /* optional sema to post on completion */
   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 sprocList */
    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 necessary.
|| 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 free */
    ussetlock(sprocListLock);       /* acquire exclusive rights to sprocList */
    theSproc = sprocList;        /* get address of first free oneSproc */
    sprocList = theSproc->next;  /* make next in list, the head of list */
    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 exec */
    theSproc->callArg = callWith;      /* set argument to pass */
    theSproc->sprocDone = done;           /* set sema to post on completion */
    usvsema(theSproc->sprocWait);      /* wake up sleeping process */
}
```

Process Scheduling Features

The IRIX kernel supports special process scheduling rules for share groups. This permits you to increase the efficiency of a parallel program in some cases. The feature is controlled by the **schedctl()** kernel function (detailed in the schedctl(2) reference page).

When **schedctl()** is called with the SCHEDMODE argument, it sets one of three scheduling rules for the share group whose member issues the call:

SGS_FREE — The normal situation, in which each process is scheduled individually.

SGS_SINGLE — All but the master process of the share group are blocked. This permits the master process to perform initialization or error recovery without contention from other members of the group.

SGS_GANG — All processes of the group run concurrently, provided there are sufficient CPUs available.

Under gang scheduling, IRIX tries to run all processes of a share group concurrently. When this is possible (in other words, when there are enough available CPUs in the multiprocessor), gang scheduling can greatly reduce lock conflicts between processes.

Without gang scheduling, one member of the share group can acquire a lock and then be suspended. Another member, attempting to acquire the lock, is also suspended until the first process is dispatched again and releases the lock.

With gang scheduling, when a second member attempts to acquire the lock, the first process is almost certainly executing at the same time, and releases the lock while the second member is still spinning.

Process Management Features

The **prctl()** kernel function provides a variety of process-related management tools (detailed in the prctl(2) reference page). One feature useful for parallel programs is the PR_MAXPPROCS query. This returns the number of different CPUs that the calling process could use for execution. The returned number is 1 when the caller has been assigned to a particular CPU. Otherwise it is the number of unrestricted CPUs in the system. A parent process could use this during initialization to find out the degree of parallelism it can hope to achieve.

The **sysmp()** kernel function provides information about a multiprocessor (detailed in the sysmp(2) reference page). Some of the queries useful to a parallel program include

MP_NPROCS, return number of CPUs in the system, and MP_NAPROCS, return the number of CPUs available for normal process scheduling.

Using POSIX Threads

As noted under "Thread-Level Parallelism" on page 128, 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. Your program must use the functions defined by IEEE standard 1003.1c-1995. Threads of this kind are called *pthreads*, short for POSIX threads.

Comparing Pthreads and Lightweight Processes

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 than a program that depends on a unique facility such as **sproc()**. Table 3-4 summarizes some of the differences between pthreads and lightweight processes.

Table 3-4 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	Pthread mutexes and condition variables	IRIX semaphores and locks	IRIX semaphores and locks

Table 3-4 (continued) Comparison of Pthreads and Processes

Attribute	POSIX Threads	Lightweight Processes	UNIX Processes
Files, pipes, and I/O streams	Shared single-process file table	Shared or separate file table	Separate
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
Effective user and group IDs	Inherited and unchangeable	Inherited, can be changed	Inherited, can be changed

It takes relatively little time to create or destroy a pthread, as compared to creating a lightweight process. Pthreads are dispatched almost entirely within user-level code, in the pthreads library. Because there are fewer context switches in and out of the kernel, there is less overhead in dispatching a large number of threads than there is in dispatching a similar number of lightweight processes.

On the other hand, threads share all resources and attributes of a single process (except for the signal mask, see “Pthreads and Signals” on page 153). 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 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 3-5.

Table 3-5 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	SGI 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).

Prior to the inclusion of *stdio.h*, be sure that the compiler variables _POSIX1C and _NO_ANSIMODE are defined. These variables are set by default in most compiles. Read the header file */usr/include/standards.h* (which is included by *stdio.h*) to see the logic of standard namespace definition.

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*. (However, you must be sure that the needed version of the library is installed; the *-n32* and *-64* libraries do not install by default.)

Tip: Many names in *libpthread* override names defined in *libc*. The linker displays many warning messages about these overrides. You can silence the warnings with the *-Wl,-woff,85* compile option.

Debugging With dbx

The *dbx* debugger is distributed with the IRIX Developer's Option. Version 7.0 of *dbx* is required to work properly with pthreads.

When debugging a pthreads program, you must set the following *dbx* variables:

- Set $promptonfork to 2
- Set $mp_program to 1

When you set a breakpoint with *dbx*, it is global to all threads. The first thread to reach the breakpoint will trip the breakpoint. This stops execution of the entire process (all threads). If you set the breakpoint in code used by more than one thread, the program could be in a different thread each time it stops. The thread ID is displayed at the stop, as in the display in Example 3-2.

Example 3-2 Debugger Display of Pthread Program

```
(dbx) showthread all
Thread: Start:               State:     Pid:  Location:
0x10000                      COND-WAIT        _SGIPT_sched_block ["xp.c":966]
0x10001 work_thread          RUNNING     1512 FLOCAL_ALIGN ["workfn.c":864]
0x10002 work_thread          RUNNING     1520 FLOCAL_ALIGN ["workfn.c":850]
0x10003 work_thread          RUNNING     1563 FLOCAL_ALIGN ["workfn.c":866]
0x10004>work_thread          RUNNING     1425 thr_tst ["workfn.c":391]
```

You can single-step a threaded program as long as you know that only one thread is executing the code through which you are stepping. When you single-step through code that is executed by more than one thread, confusing results can occur. To single-step, *dbx* sets a breakpoint where the program should stop next. However, breakpoints are global. When you give the *next* command in one thread, the stop can occur in a different thread.

Debugging with the Workshop Debugger

The Workshop Debugger is part of the Developer Magic package. In version 2.6.2 of this package, the debugger is aware of pthreads. The command line view in the debugger main window can be used to set breakpoints and to produce a display similar to the one in Example 3-2.

Breakpoints set with the Workshop Debugger are global to the program and are taken by the next thread to reach them, as with *dbx*.

Creating Pthreads

You create a pthread using **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 with 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 3-6 and described in the following text.

Table 3-6 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 in a *pthread_attr_t* object.
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 in a *pthread_attr_t* object.
pthread_attr_setschedpolicy(3P)	Set the scheduling policy in a *pthread_attr_t* object.
pthread_attr_setstacksize(3P)	Set the stack size attribute in a *pthread_attr_t* object.
pthread_attr_setstackaddr(3P)	Set the address of memory to use as a stack in a *pthread_attr_t* object (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

After a thread has terminated, it can be "detached." Detaching means that the pthreads library deletes its information about the thread, possibly releasing some memory (see "Joining and Detaching" on page 150). 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 detached by an explicit function call.

Initial Scheduling Priority and Policy

Scheduling priorities and policies are described under "Scheduling Pthreads" on page 157. You can specify an initial scheduling policy by calling **pthread_attr_setschedpolicy()**, passing one of the policy constants SCHED_FIFO, SCHED_RR, or SCHED_OTHER.

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()**.

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 these things 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.

Thread Stack Allocation

Each pthread has an execution stack area in memory. You use **pthread_attr_setstacksize()** to specify the size of this stack area. You cannot specify a stack size less than a minimum. You can learn the minimum by calling **sysconf()** with _SC_THREAD_STACK_MIN (see the sysconf(3C) reference page).

By default, **pthread_create()** allocates stack space of the specified size from dynamic memory. When it does so, the stack space is automatically released when the thread is detached.

You can instead preallocate stack space from any source of dynamic memory such as **malloc()**. When you preallocate stack space, you must:

- Specify the address of the space using **pthread_attr_setstackaddr()**.

  This tells **pthread_create()** not to allocate space.

- Specify the size of the allocated space using **pthread_attr_setstacksize()**.

  This enables **pthread_create()** to initialize the correct starting stack address.

- Free the stack space when the thread terminates (see "Joining and Detaching" on page 150).

There is normally no protection against a thread overrunning the space. If a thread allocates too much automatic data or makes too many nested function calls, it will attempt to modify memory outside the stack space. This might cause a segmentation fault if that memory is not allocated, or it might modify memory used for other purposes.

Tip: When you preallocate stack space, you can create "red zones" around the allocated stacks as follows:

- Allocate the stack memory in multiples of the system page size aligned on page boundaries (see the getpagesize(2) and memalign(3C) reference pages).
- Allocate an extra page of memory above and below each stack area.
- Use the **mprotect()** function to set the protection of the extra pages to PROT_NONE (see the mprotect(2) reference page).

This procedure creates untouchable pages at each end of the stack area. If the thread misuses its stack, it will usually terminate at once with a segmentation fault. (It is still possible for a thread to call a function that allocates more than a page of automatic variables, and so skips over the "red zone" to modify memory beyond it.)

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 (see the getrlimit(2) reference page for information on resource limits).

Executing and Terminating Pthreads

The functions you use to manage the progress of a thread are summarized in Table 3-7 and described in the following topics.

Table 3-7 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.

Table 3-7 (continued) Functions for Managing Thread Execution

Function	Purpose
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

You 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" on page 157).

Initializing Static Data

Your program may use static data that should be initialized, but only 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?

The answer is to create a variable of type *pthread_once_t*, statically initialized to the value PTHREAD_ONCE_INIT. In the module code, you 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 will wait until the first thread completes the call. An example is shown in Example 3-3.

Example 3-3 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 will be called when threads terminate 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. 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 and Being Terminated

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 status switches:

- The **pthread_setcancelstate()** function lets you prevent cancellation entirely (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 state of a thread is PTHREAD_CANCEL_ENABLE and 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(3) reference page. A thread can explicitly permit cancellation by calling **pthread_testcancel()**.

Joining and Detaching

Sometimes you do not care when threads will 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 find out when another has 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” on page 146).

The **pthread_join()** function detaches the terminated thread. If your program does not use **pthread_join()**, and does continue execution after threads have terminated, 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” on page 145). Another is to call **pthread_detach()** at any time after creating the thread, including after it has terminated.

If your program continues for a long time creating threads and letting them terminate, but does not arrange for detaching the completed threads, eventually an error will occur because resources have been used up.

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 in Table 3-8.

Table 3-8 Functions for Thread-Unique Data

Function	Purpose
pthread_key_create(3P)	Create a key (class of thread data)
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. A storage key represents one kind or class of data. Each thread has a unique instance of this class of data, with an initial value of NULL. The returned key value (of type *pthread_key_t*) is used by all threads to store and retrieve data of this class.

Any thread can use **pthread_getspecific()** to retrieve that thread's unique instance of the value stored under this key. A thread can only get its own instance, which is the instance that has been stored by this same thread using **pthread_setspecific()**. Any thread's stored value is NULL until it stores a new value.

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

You create keys by calling **pthread_key_create()**. Keys can be created before any threads are created. However, when you are designing a library module for use from any threaded program, you need to create a key upon first entry to your library code. This is an ideal application for a *pthread_once_t* variable (see "Initializing Static Data" on page 148). The code in Example 3-4 suggests how a threaded module would create a key if necessary, and initialize its contents for the current thread.

Example 3-4 Initializing Thread-Unique Data

```
typedef struct perThread_s {
   ...items of data unique to thread...
} perThread_t;
pthread_key_t perThreadKey; /* key used to find per-thread info */
pthread_once_t makePerThreadKey = PTHREAD_ONCE_INIT;
/*
|| Destructor function, called when any thread exits with a
|| non-NULL value of perThreadKey.
*/
void deletePerThread(void *arg)
{
   free(arg);
}
/*
|| One-time initializing function, called through pthread_once,
|| to create the perThreadKey.
*/
void createPerThreadKey(void)
{
   pthread_key_create(&perThreadKey,deletePerThread);
}
/*
|| Return the address of this thread's instance of perThread_t,
|| Create the struct if necessary. Create the key if necessary.
*/
struct perThreadInfo *getPerThread(void);
{
   perThread_t *ppt;
   int ret;
   pthread_once(&makePerThreadKey, createPerThreadKey);
   ppt = pthread_getspecific(perThreadKey);
   if (NULL==ppt)
```

```
    {
       ppt = (perThread_t*)malloc(sizeof(perThread_t));
       ...initialize fields of ppt->new per-thread struct...
       ret = pthread_setspecific(perThreadKey,(void*)ppt);
       if (ret) perror("pthread_setspecific()");
    }
    return ppt;
}
```

The code in Example 3-4 includes the following functions and global variables:

perThreadKey	The key that represents the class of *perThread_t* structures.
makePerThreadKey	A *pthread_once_t* variable used to ensure that **pthread_key_create()** is called only once.
deletePerThread()	Destructor function, passed to **pthread_key_create()**, called when any thread terminates leaving a non-NULL value under the *perThreadKey* key.
createPerThreadKey()	Function called via **pthread_once()** to create *perThreadKey.*
getPerThread()	Function that can be called from any thread to retrieve that thread's value of *perThreadKey*. If the key itself has not been defined, the function defines it (calling **createPerThreadKey()** by way of **pthread_once()**). If this thread's value of the key is NULL, the function creates and initializes a value, and stores it using **pthread_setspecific()**.

Pthreads and Signals

Signals are an integral part of UNIX programming. For a general overview of signal concepts and numbers, see the signal(5) reference page. IRIX supports three different, partly-compatible, signal facilities: BSD signals, SVR4 signals, and POSIX signals. When you are writing a pthreads program, you must be sure to use only the POSIX signal facilities. Do not mix use of other signal functions in a pthreads program or unpredictable results can follow.

The POSIX signal functions, including new pthreads functions, are summarized in Table 3-9. The functions are listed alphabetically. They are discussed in related groups in the following topics.

Table 3-9 Functions for POSIX Signal Handling

Function	Purpose
kill(2)	Send a signal to a process or process group.
pthread_kill(3P)	Send a signal to a specified thread.
pthread_sigmask(3P)	In a program linked with *libpthread*, examine or change the mask of signals allowed and blocked by the thread.
sigaction(2)	Specify or test the action to be taken by the process when a signal is received asynchronously.
sigpending(2)	Return set of signals pending for the process or the calling thread.
sigprocmask(2)	In a program not linked with *libpthread*, examine or change the mask of signals allowed and blocked by the process.
sigqueue(3)	Queue a signal against a specified process.
sigsetops(3)	Manipulate signal mask objects.
sigsuspend(2)	Unblock selected signals for the calling thread, and wait for a signal.
sigtimedwait(3) sigwait(3) sigwaitinfo(3)	Wait for and receive specified signals.

Signal Generation

Signals can be generated asynchronously by events outside the program, for example when a requested timer expires, or when a privileged process uses the **kill()** system function. Signals can be generated by hardware action, for example when the program references undefined memory.

Signals can be generated by program action as well. The **kill()** function can be used to send a signal to the current process. The **pthread_kill()** function can generate signal directed to a specific thread in the process.

Signals from outside the program, and signals generated using **kill()**, are directed to the process as a whole. Signals directed to the process are delivered to the first thread that permits delivery of that kind of signal. (There is no way to predict which thread this will be.)

Hardware-generated signals, and signals sent with **pthread_kill()**, are directed to one specific thread. The signal can be delivered only to that thread.

Setting Signal Masks

A thread specifies which signals it is willing to receive. In a conventional, single-threaded, process this is done using **sigprocmask()**. However, a program that is linked with the pthreads library must use **pthread_sigmask()** instead.

Each thread has a signal *mask*, a set of bits that correspond to the possible signal numbers (as listed in the signal(5) reference page). A 1-bit in the mask causes that signal to be *blocked*. A blocked signal cannot be delivered to the thread. When that signal is directed to the thread alone, the signal remains pending until the thread unblocks the signal.

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

The family of functions documented in the sigsetops(3) reference page are used to create and manipulate signal masks, setting and clearing bits.

Each thread inherits the signal mask of the thread that calls **pthread_create()**. You use **pthread_sigmask()** to set the signal mask of the calling thread. Typically you will set an initial mask in the first thread, so that it can be inherited by all other threads.

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

Setting Signal Actions

When a signal is not blocked and is delivered, some action is taken. You specify what that action should be using the **sigaction()** function. You specify an action for each signal number separately. These actions are set on a process-wide basis, *not* individually for each thread. Each thread has a private signal mask, but signal actions are specified for all threads in the process. You choose among the following actions for each signal:

SIG_DFL	Default handling, which depends on the specific signal but is either to ignore the signal or to terminate the process, with or without a dump.
SIG_IGN	Ignore the signal, that is, discard it when it is generated. Certain signals cannot be ignored.
(function address)	Signal is delivered by an asynchronous call to the specified function.

When a signal is delivered to a function, you have the option of specifying a function that receives a *siginfo_t* structure with information about the signal. These and other options are spelled out in the sigaction(2) reference page.

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 arriving 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.

Receiving Signals Asynchronously

When the specified action for a signal is to catch the signal in a function and the signal is not blocked, the signal is delivered asynchronously. That is, the function is called as soon as the signal is generated or unblocked.

You can control the delivery of asynchronous signals more closely using the **sigsuspend()** function. This function takes a signal mask as its argument. It makes that mask the current mask for the thread (unblocking some signals that were blocked). The calling thread is suspended until at least one signal has been delivered. After a signal has been handled, the function restores the previous signal mask and returns.

Scheduling Pthreads

By default, the pthreads library schedules the threads of a process in a round-robin fashion. Much of the scheduling machinery is done in the library, within the context of the user process, without assistance from the IRIX kernel. On a multiprocessor, threads can run concurrently.

The scheduling algorithm is controlled by two parameters: a policy and a priority for each thread. These variables are set initially when the thread is created (see "Initial Scheduling Priority and Policy" on page 146), and can be modified while the thread is running. The functions used in scheduling are summarized in Table 3-10.

Table 3-10 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.

Scheduling Policy

There are two scheduling policies in this implementation: first-in-first-out (SCHED_FIFO) and round-robin (SCHED_RR). (The default SCHED_OTHER behaves the same as SCHED_RR.) SCHED_FIFO and SCHED_RR are similar. The round-robin scheduler ensures that when a thread has used a certain maximum amount of time without blocking, 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(3) reference page.

Scheduling Priority

The queues of runnable threads are ordered by thread priority numbers, 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. There are at least 32 priority values and the lowest is greater than or equal to 0. You can use these functions to set up a system of relative priorities as suggested by the code in Example 3-5.

Example 3-5 Establishing Relative Priority Levels

```
#include <sched.h>
int higherP, mediumP, lowerP;
void setRelativePriorities()
{
   int maxP, minP;
   maxP = sched_get_priority_max();
   minP = sched_get_priority_min();
   mediumP = minP + ((maxP-minP)/2);
   higherP = mediumP+1;
   lowerP = mediumP-1;
}
```

When all threads use one of the three priorities *higherP*, *mediumP*, or *lowerP*, threads that run at *higherP* will always run in preference to threads at the other two priorities.

A thread can set another's priority or scheduling policy, or both, using **pthread_ setschedparam()**. A simple function to set a specified priority on the current thread, returning the previous value, is shown in Example 3-6.

Example 3-6 Function to Set Own Priority

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

```
    if (ret)
    { perror("pthread_setschedparam()"); }
    return oldP;
}
```

Synchronizing Pthreads

Asynchronous threads using a common address space must cooperate and coordinate their use of shared variables. Independent processes coordinate using the mechanisms described in previous chapters: IRIX semaphores and locks and SVR4 semaphores. Threads cannot use these IPC mechanisms. Threads can coordinate using:

- Mutex (mutual exclusion) objects, which allow threads to gain exclusive use of a shared variable
- Condition variables, which allow a thread to wait when a controlling predicate is false.

Threads and Conventional IPC

You cannot use IRIX semaphores, locks, and barriers to coordinate between multiple threads within a single program. Nor can you use SVR4 semaphores for this purpose.

Preparing Mutex Objects

A mutex is a software object that stands for 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.

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 3-11.

Table 3-11 Functions for Preparing Mutex Objects

Function	Purpose
pthread_mutex_init(3P)	Initialize a mutex object based on a *pthread_mutexattr_t*.
pthread_mutex_destroy(3P)	Uninitialize a mutex object.
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 in a *pthread_mutexattr_t*.
pthread_mutexattr_setprotocol(3P)	Set the priority protocol choice in a *pthread_mutexattr_t*.
pthread_mutexattr_getprioceiling(3P)	Query the minimum priority in a *pthread_mutexattr_t*.
pthread_mutexattr_setprioceiling(3P)	Set the minimum priority in a *pthread_mutexattr_t*.

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. Dynamic initialization should be done only once (see "Initializing Static Data" on page 148).

Two attributes can be set in a *pthread_mutexattr_t*. The priority inheritance protocol is the more important. 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.
PTHREAD_PRIO_PROTECT	The thread holding the mutex runs at a priority at least as high as the 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 the highest priority of any thread blocked on that mutex.

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.

When round-robin scheduling is used, and a mutex represents a critical section of code, a second problem can arise. If a thread acquires the mutex, enters the critical section, and then is suspended because its time slice is up, other threads can be blocked needlessly 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. This keeps it at the front of the round-robin queue until it exits the critical section and releases the mutex.

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

Using Mutexes

The functions for claiming, releasing, and using mutexes are summarized in Table 3-12.

Table 3-12 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.

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. However, a condition variable has no useful attributes to initialize in this implementation. The functions for initializing one are summarized in Table 3-13.

Table 3-13 Functions for Preparing Condition Variables

Function	**Purpose**
pthread_cond_init(3P)	Initialize a condition variable based on an attribute object.
pthread_condattr_init(3P)	Initialize a *pthread_condattr_t* to default attributes.
pthread_condattr_destroy(3P)	Uninitialize a *pthread_condattr_t*.

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. Dynamic initialization should be done only once (see “Initializing Static Data” on page 148).

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 needed data.” A thread that wants to wait for that event claims the condition variable, which causes it to wait. The thread that recognizes the event signals the condition variable, releasing one or all threads that are waiting for the event.

A thread holds a mutex that represents a shared resource. While holding the mutex, the thread finds that the shared resource is not complete or not ready. The thread needs to do three things:

- Give up the mutex so that some other thread can renew the shared resource.
- Wait for the event that "resource is now ready for use."
- Re-acquire the mutex for the shared resource.

These three actions are combined into one using a condition variable. The functions used with condition variables are summarized in Table 3-14.

Table 3-14 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 two arguments: a mutex that is owned by the calling thread, and a condition variable. 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 early, 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 3-7, which is modelled after an example in the POSIX 1003.1c standard.

Example 3-7 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 3-7 implement 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 to repeat its search of the list for the desired key and, on finding the element again, to 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 will change 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 will no longer be 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.

Using MPI and PVM

MPI (see "Message-Passing Interface (MPI) Model" on page 130) and PVM (see "Portable Virtual Machine (PVM) Model" on page 130) are two approaches to the same problem: how to distribute a concurrent program across a cluster of computers.

Choosing Between MPI and PVM

Silicon Graphics, Inc. has adopted the MPI interface as the primary and preferred model for distributed applications on Array processors. There are occasions when you may elect to use PVM instead, but in general MPI is strongly recommended for new applications and for applications that are being ported to an Array system.

In many ways, MPI and PVM are similar:

- Each is designed, specified, and implemented by third parties who have no direct interest in selling hardware.
- Support for each is available "on the net" 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.

The primary reason MPI is preferred for Array systems is performance. The design of MPI is such that a highly optimized implementation could be created for the homogenous environment of Silicon Graphics, Inc. Array systems. Under Array 2.0, MPI applications take advantage of a HIPPI "bypass" connection to exchange data with small latencies and high data rates. Specific data rates and latencies are listed in the book *Getting Started With Array Systems*, 007-3058-002.

The PVM implementation for Array systems is not as highly tuned, although still effective for some work.

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 reasons can be 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 heterogenous network that includes both uniprocessors and multiprocessors, and includes computers from multiple vendors. When the application runs in the environment of a Silicon Graphics, Inc. Array system, MPI is the recommended interface.

Porting From PVM to MPI

Because MPI and PVM address similar problems in ways that are conceptually similar, you can consider porting a program from PVM to MPI in order to get better performance on an Array system. A detailed discussion of this process, with examples, appears in Appendix B, “Converting PVM Applications to MPI.”