System Programming for Solaris 2.3

Department of Computer Science
124 Latrobe Street, Melbourne 3001, Australia.
Email: `steveh@rmit.edu.au`

This document was preprared using the TEX and LATEX systems.

Contents

Chapter 1

Processes

1.1 Introduction

A *process* is a portion of executable code that resides in the computers core memory or temporary swap space. A *program* is not a process. A program is defined as the executable image of one or more processes stored in a file.

Each process has a *context*, which defines the *state* of a process at a certain time. A process context consists of:

- The executable code for a process (`text`).
- The memory required to store data for a process (`data`).
- The stack for the process (`stack`).
- A process region table, which keeps track of the various pages of virtual memory belonging to the process.
- The register values of the CPU for the process.
- Other housekeeping information stored in the process table (`proctab`).

Each process has a unique identification number, call the PID (process identification number). The PID is often used to index into a processes `proctab` by the kernel. The PID can also be used for controlling a process by executing user programs.

1.1.1 Process States

Process states (and contexts) change during the lifetime of an executing process. After a process has been created, it undergoes state transitions which can be depicted by a State transition Graph (See Figure 1.1). All processes start their existence in the *created* state and terminate in the *zombie* state.

The possible states that a process may be in are:

1. User Running – The process is executing in user mode.
2. Kernel Running – The process is executing a system call.
3. Runnable in Memory – The process is not running but is ready for the kernel to schedule it.
4. Sleeping in Memory – The process is sleeping in memory.

5. Swapped but Runnable – The process has been temporarily swapped out to disk, but it ready to execute.

6. Sleeping and Swapped – The process has been moved to a temporary file and is sleeping.

7. Preempted – The process was returning from kernel mode execution to user mode execution when the kernel scheduler decided to let a higher priority process execute.

8. Created – The process is newly created but not ready for execution yet.

9. Zombie – The process has terminated but its context has not yet been destroyed.

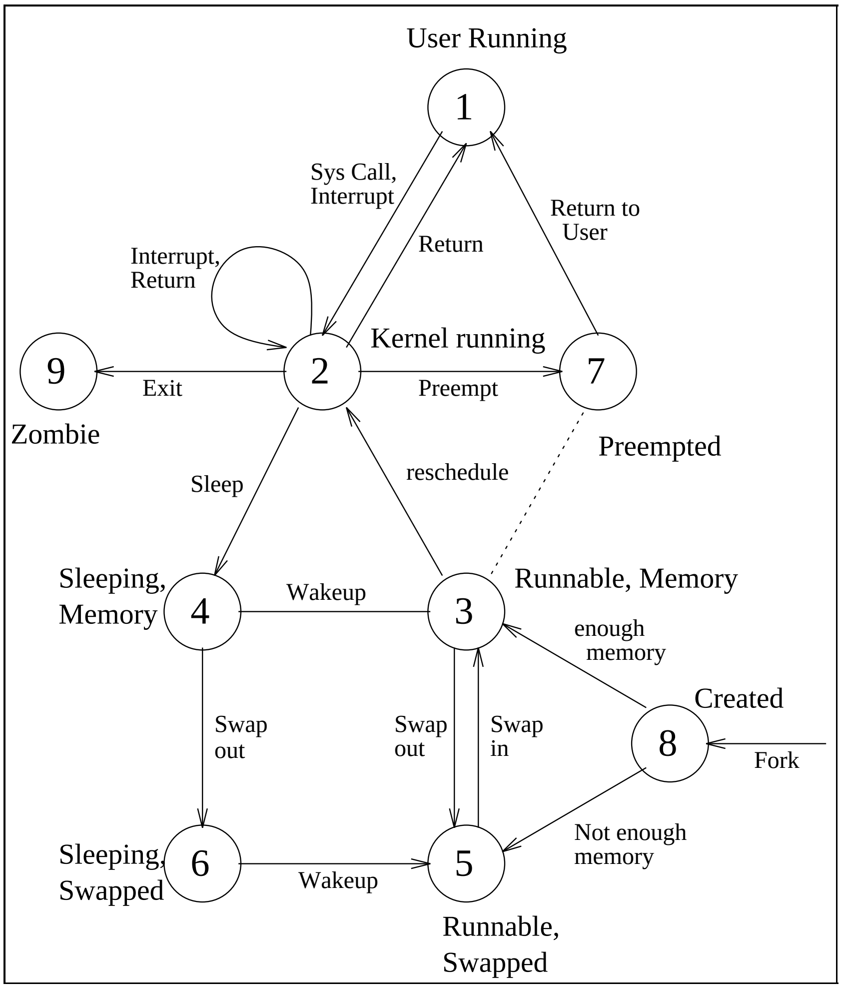

Figure 1.1: Process State Transition Diagram

1.2 Manipulating Processes

Process manipulation is performed via system calls to the operating system. Every process maniuplation system call, with the exception of `fork()` and `wait()`, require a PID value to be passed as an argument.

1.2.1 Process Creation

Process creation under UNIX involves the duplication of an existing process context. One notable exception to this rule is the `init` process (PID = 0). The `init` process is the first process to execute when a UNIX system is booted, and as such there are no other existing processes to duplicate a context from.

Once a process context has been duplicated, the context can be changed to execute a new program or to execute a different set of subroutines.

The `fork()` System Call

To create a new process in UNIX the `fork()` system call is used. `fork()` creates a new context based on the context of the calling process. The `fork()` call is unusual in that it returns *twice*: It returns in both the process calling `fork()` and in the newly created process.

The synopsis for `fork()` is as follows:

```
#include <unistd.h>

pid_t fork(void);
pid_t vfork(void);
```

If `fork()` is sucessful, it returns a number of type *pid_t* which is greater than 0 and represents the PID of the newly created *child* process. In the child process, `fork()` returns 0. If `fork()` fails then its return value will be less than 0. `vfork()` is a more efficient version of `fork()`, which does not duplicate the entire parent context. `vfork()` is suitable for use with `exec()`, which will be described later.

A trivial example of `fork()` follows. Here, the parent process prints "Hello" to `stdout`, and the new child process prints "World.". Note that the order of printing is *not* guaranteed. Without some method of synchronising the processes execution, "Hello" may or may not be printed before "World.".

```
#include <unistd.h>
#include <stdio.h>

char string1[] = "Hello";
char string2[] = "World.\n";

int main(void)
{
        pid_t PID;

        PID = fork();
        if (PID == 0)                   /* In the child process? */
                printf("%s", string2);
        else                            /* In the parent process */
                printf("%s", string1);
        exit(0);                        /* Executed by both processes */
}
```

The `exec()` Family of System Calls

Often we wish to spawn a different process as a child of the process that is executing. In order to accomplish this, we must first create a new process using `fork()` and then *replace* the image of the child process with a new process image. The image of a new process is created by the operating system from an executable binary file stored on disk.

The exec() family of system calls replace the image of the calling process with the image of a different process stored on disk. The synopsis of the exec() family of system calls follows:

```
#include <unistd.h>

extern char **environ;

int execl( const char *path, const char *arg, ...);
int execlp( const char *file, const char *arg, ...);
int execle( const char *path, const char *arg , ..., char *const envp[]);
int exect( const char *path, char *const argv[]);
int execv( const char *path, char *const argv[]);
int execvp( const char *file, char *const argv[]);
```

Refer to the exec manual page for a detailed discription of these functions. We shall only discuss execlp(). The listing below shows how execlp() is used to execute the UNIX ls program as a child of the parent process.

```
#include <unistd.h>
#include <stdio.h>

int main(void)
{
        pid_t PID;

        PID = vfork();
        if (PID == 0)                       /* In the child process? */
                execlp("/bin/ls", "");  /* Execute ls as the child */
        wait((int *) 0);                    /* Wait for the child */
        printf("done!\n");
        exit(0);
}
```

When the child process executes execlp() its PID does not change, and the operating system still recognises the child process as belonging to the parent process. The child process terminates its execution as soon as ls has finished executing. Should execlp() (or any other member of the exec() family) fail to create a new process image then they will return a number less than 0.

The wait() System Call

In the above example for execl(), the wait() system call is being used to force the parent process to *wait* until its child process has terminated before it resumes execution. Hence "done!" is always going to be printed to the terminal *after* the output of ls has been displayed.

The synopsis for wait() is as follows:

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status)
pid_t waitpid(pid_t pid, int *status, int options);
```

The wait() system call suspends execution of the current process until a child of that process has terminated. If the child exits before wait() is executed then wait returns immediately. wait() accepts a single call-by-reference argument in which `wait()` stores the child processes exit value. If the argument to `wait()` is 0 then no attempt to return the childs exit status is made.

The `waitpid()` system call suspends execution of the current process until the child with the specified PID terminates. This is useful if the current process has spawned multiple children but is only required to wait on a particular one. Refer to the on-line UNIX `wait` manual page for further details of `waitpid()`.

1.2.2 Process Priority Control

From a programmers point of view, UNIX processes have priorities that range from -20 (*highest* priority) to +19 (*lowest* priority). The default priority for all user processes is 0. Users can only *decrease* the priority of an executing process (unless the priority number of the process is already greater than 0). Users can not increase the priority of a process if the priority number is 0 (ie: the user is not allowed to specify a negative priority number). Be carefull not to get confused about priority numbers; the *lower* the number the higher the likelyhood that a process will be scheduled.

The *root* user is capable of specifying negative priority numbers, and hence is able to give processes higher priorities than users.

The `nice` UNIX utility is used for specifying the priority of a process. Refer to `nice` in section 1 of the on-line manual pages for details of this command.

Programmers should be aware that priorities for processes specified with `nice` are *not* the exact priority numbers used by the UNIX scheduler. Internally, the UNIX kernel calculates the *real* priority of a process dynamically, based on how much CPU time the process has already been given (internal priority values decay over time). Dynamic priority calculation prevents the starvation of lower priority processes. All things considered, the programmer should view process priorities specified by `nice` as *desired* priorities and not *actual* priorities. For this reason, standard UNIX processes are not appropriate for many real–time processing applications.

The `nice()` System Call

The `nice()` system call is used to change the priority of the currently executing process. It takes a single argument `inc` which is a number between 0 and +19 for user-executed processes and between -20 and +19 for root executed processes. `nice()` returns 0 if successful and -1 if an error occures.

The synopsis for `nice()` follows:

```
#include <unistd.h>

int nice(int inc);
```

BSD Compatibility Functions for Priorities

Solaris 2.3 supports a BSD compatibility library, which includes the `setpriority()` and `getpriority()` system calls. These system calls allow priorities of *groups* of processes to be specified, as well as the priority of all processes belonging to a user to be set simultaneously.

Although these calls are considerably more powerful and flexible than `nice`, the programmer should not utilise them if:

1. Portability is an issue.
2. Non-BSD system libraries are also being used.
3. A multi-threaded application is being developed.

1.2.3 Process Termination

A number of conditions can cause a program to terminate:

1. There is no more code to execute (the end of `main()` is reached).
2. The `exit()` function is executed.
3. the `abort()` function is executed.
4. A signal is sent to the process which causes it to terminate.

Normal program termination occures when either of the first two conditions are satisfied. The `abort()` system call is used when a process detects an error condition which it can not recover from.

The `exit()` System Call

The purpose of the `exit()` system call is to gracefully terminate the currently executing process. A *status* number is returned by `exit()` to the parent of the terminating process. The status value is used to indicate if the terminating process was successful or not. Typically negative status values indicate an error occured, while 0 indicates successful execution.

The synopsis for `exit()` follows:

```
#include <stdlib.h>

void exit(int status);
```

The use of `exit()` to terminate a process is not required, but is encouraged to ensure that the status value of the terminating process is explicitly set. `exit()` *never* returns to the calling process, so it's return type is `void`. Any open streams and files belonging to the terminating process are automatically flushed and closed during `exit()`.

The `abort()` System Call

The `abort()` system call is similar to exit in use, except that no user-defined status is returned to the parent of the terminating process. Internally, `abort` generates a signal which terminates the process. The SIGABORT signal, generated by `abort()`, can neither be blocked or ignored. Signals are discussed later in this chapter.

The synopsis for `abort()` follows:

```
#include <stdlib.h>

void abort(void);
```

1.2.4 Suspending a Process

Sometimes it is necessary to suspend the execution of a process until some externel event occurs. This can be accomplished with the `pause()` system call. `pause()` always returns -1. The synopsis for `pause()` follows:

```
#include <unistd.h>

int pause(void);
```

If process needs to be suspended for a certain amount of time, the `sleep()` system call can be used. `sleep()` takes a single argument which specifies the number of seconds to suspend the process. The synopsis for `sleep()` follows:

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
```

1.3 Process Groups

A parent process and it's children are all associated together. The kernel keeps track of the association by using process groups. By default, every process executed from a UNIX login shell belongs to the same group. Each process group has a unique identifier, called the Group ID (GID). The GID for a group is determined by the PID of the controlling process of the group (usually a login shell).

By default, processes inherit the GID of their parent. Thus a group of processes is a heirarchical structure, with the controlling process at the root of the heirarchy. The controlling process of a group is also known as the *session leader*.

The `setpgrp()` System Call

The `setpgrp()` system call creates a new process group. The `setpgid()` system call adds a process to a process group.

The synopsis for `setpgrp()` follows:

```
#include <sys/types.h>
#include <unistd.h>

pid_t setpgrp(void);
int setpgid(pid_t pid, pid_t pgid);
```

If the process calling `setpgrp()` is not already a session leader, the process becomes one by setting its GID to the value of its PID. `setpgid()` sets the process group ID of the process with PID `pid` to `pgid`. If `pgid` is equal to `pid` then the process becomes the group leader. If `pgid` is not equal to `pid`, the process becomes a member of an existing process group.

The `getpid()` Family of System Calls

The synopsis for `getpid()` family of system calls follows:

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);
pid_t getpgrp(void);
pid_t getppid(void);
pid_t getpgid(pid_t pid);
```

The `getpid()` system call returns the PID of the calling process. `getpgrp()` returns the process group ID (GID) of the calling process. `getppid()` returns the parent process PID of the calling process. `getpgid()` returns the process group ID of the process whose process ID is equal to `pid`, or the process group ID of the calling process, if `pid` is equal to 0. If successful, these functions return the correct PID or GID. If an error occures -1 will be returned.

Below is an example of using `setpgrg()`, `getpid()` and `getpgrp()`. The example also introduces *signals*, which will be discussed in the next section.

```
#include <signal.h>

int main(void)
{
        register int i;

        setpgrp();
        for (i = 0; i < 10; ++i)
        {
                if (fork() == 0)
                {                                       /* In the child process */
                        if (i & 1)
                                setpgrp();
                        printf("pid = $d, gid = %d\n", grepid(), getpgrp());
                        pause();
                }
        }
        kill(0, SIGINT);
}
```

In the code above the process resets its GID and then spawns 10 children. Each child initially has the same GID as their parent, but the processes created during the odd iterations create their own GIDs using `setpgrp()`. The execution of the children is then suspended. Once all of the children have been created the parent sends a termination signal to every process in its group and then exits. The five "odd" processes will not be terminated because they do not belong to the parents group anymore.

1.4 Signals

Signals inform processes of the occurence of asychronous events. Every type of signal has a *handler* which is a function. All signals have default handlers which may be replaced with user-defined handlers. The default signal handlers for each process usually terminate the process or ignore the signal, but this is not always the case.

Signals may be sent to a process from another process, from the kernel, or from devices such as terminals. The `^C`, `^Z`, `^S` and `^Q` terminal commands all generate signals which are sent to the foreground process when pressed.

The delivery of signals to a process is handled by the kernel. Signals are checked for whenever a process is being rescheduled, put to sleep, or re-executing in user mode after a system call. Figure 1.2 shows when signals are tested for and handled during state changes.

1.4.1 Signal Types

There are many types of signals. The list below describes the most commonly encountered signals:

`SIGHUP` Hangup. This signal is sent to all processes attached to a control terminal when that terminal is disconnected. This signal is also sent to all processes belonging to a process group when the group controlling process is terminated.

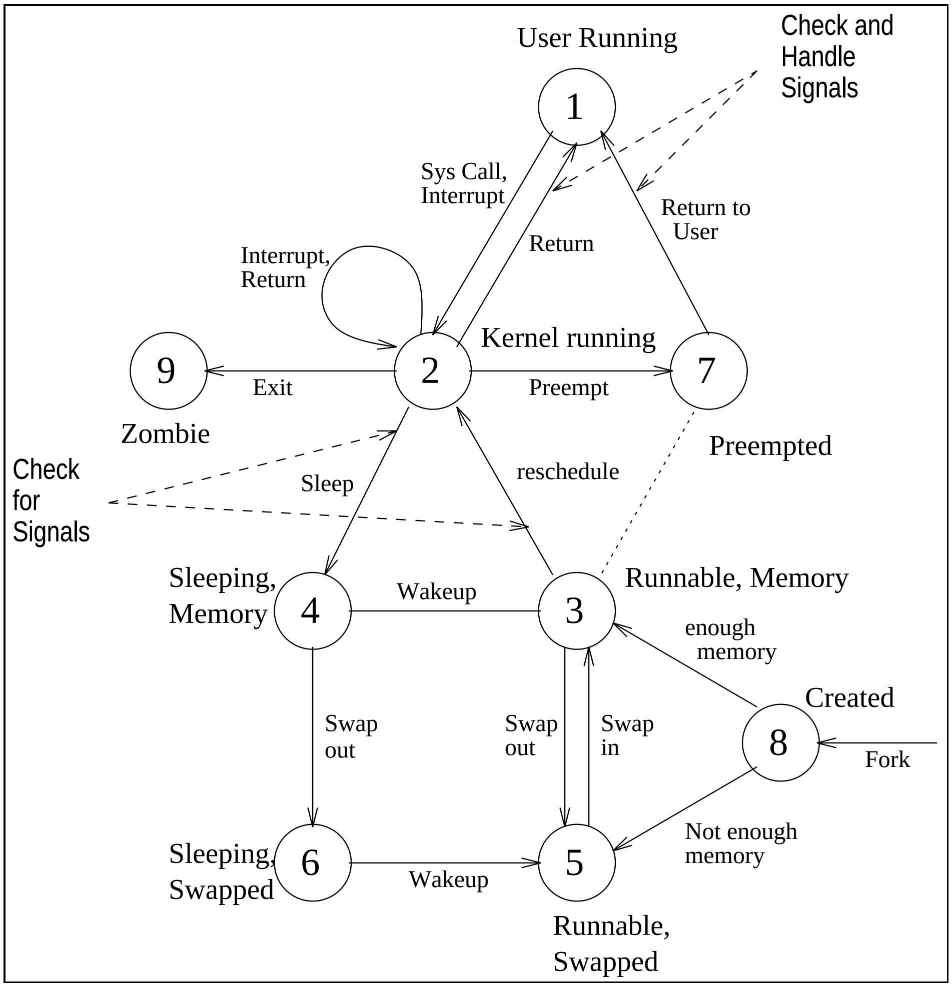

Figure 1.2: Checking and Handling Signals in the Process State Diagram

`SIGINT` Interrupt. This signal is sent to all processes associated with a terminal when `^C` is pressed.

`SIGQUIT` Quit. Sent by the kernel to a process that is to be abnormally terminated. Generates a core file for the terminating process.

`SIGILL` Illegal instruction. This signal is sent by the kernel to a process which is trying to execute invalid code.

`SIGTRAP` Trace trap. Used by debuggers and `ptrace()`.

`SIGFPE` Floating-point exception. This signal is sent by the kernel to a process that generates an FP exception such as overflow or divide-by-zero.

`SIGKILL` Kill. Allows one process to send a signal to terminate another process. `SIGKILL` can not be blocked nor caught.

`SIGSYS` Bad argument to system call. This signal is sent by the kernel to a process which has made a system call with an inappropriate argument. Usually the system call would return -1, but sometimes the kernel is not able to handle the condition, hence this signal is sent.

`SIGPIPE` Broken pipe. This signal is generated by the kernel when a process tries to write to a pipe that has no reader.

SIGALRM Alarm clock. This signal is sent by the kernel to a process which had previously set up delay using `alarm()`

SIGTERM Software termination signal. A user-definable signal that is usually used for terminating a process.

SIGUSR1 User signal number one. Another user-definable signal.

SIGUSR2 User signal number two. Yet another user-definable signal.

SIGCLD Death of a child. Sent to a parent process by the kernel when one of the parent child processes termiates. This isgnal is used by the `wait()` system call.

SIGSEGV Segmentation violation. This signal is generated by the kernel whenever a process tries to access memory out side of its virtual address space. The default action for this signal is to generate a core file and terminate the process.

1.4.2 Signal Handlers

User written processes can *catch* the majority of signals by installing a user-defined handler. This is accomplished by the `signal()` system call. User processes can also generate signals to be sent to other processes using the `kill()` system call.

The `signal()` System Call

The `signal()` system call installs a new signal handler for a particular signal type. `signal()` takes two arguments, the first of which is the type of signal, and the second is the address of the new signal handler. `signal()` returns the address of the old signal handler. It is wise to store the return address. The synopsis for `signal()` is given below:

```
#include <signal.h>
#include <unistd.h>

void (*signal(int signum, (void *handler)(int))))(int);
```

It is also possible to re-install the default signal handler or ignore the signal with `signal()`. Instead of specifying the address of a signal handler, one may specify one of the following symbols defined in `signal.h`:

SIG_DFL Use the default signal handler.

SIG_IGN Ignore the signal.

It is important to note that under UNIX System V (which Solaris is based on) once a signal handler has been installed, it is only valid for the receipt of a single signal. After the signal has been caught and handled, the default signal handler for the sent signal is automatically reinstated. Thus if the user-defined signal handler is to be used multiple times, it is necessary to reinstate the user defined signal handler inside the handler itself.

The `kill()` System Call

To transmit a signal to another process the `kill()` system call is used. The synopsis for `kill()` follows:

```
#include <signal.h>

int kill(pid_t pid, int sig);
```

`pid` identifies the set of processes to receive the signal, and `sig` is the type of signal to be sent. The following list shows the correspondence betweens values of `pid` and sets of processes.

- If `pid` is positive, then `pid` is the PID of a particular process.
- If `pid` is 0, then the kernel sends the signal to all processes in the senders group.
- If `pid` is -1, then the signal is sent to all processes with the senders user ID.
- If `pid` is less than -1, then the signal is sent to the process group with GID equal to the absolute of `pid`.

`kill()` returns a negative number if the signal could not be sent, otherwise it return 0.

1.4.3 Signals in Action

In the example code for setting process groups we saw the use of `kill()` for transmitting `SIGINT` to a process group. The five "odd" processes which created their own process groups do not receive the signal.

The code below illustrates the use of `signal()`, and also pinpoints a potential problem with SYSV signal handling:

```
#include <unistd.h>
#include <stdio.h>
#include <signal.h>

sigcatcher(void)
{
        printf("PID %d caught one\n", getpid());
        signal(SIGINT, sigcatcher);
}

main(void)
{
        int ppid;

        signal(SIGINT, sigcatcher);
        if (fork() == 0)
        {
                sleep(5);         /* in the child */
                ppid = getppid();
                while (1)
                        if (kill(ppid, SIGINT) == -1)
                                exit();
        }
        nice(10);                 /* parent runs with lower priority */
        while (1);
}
```

In this example, it is possible for the following sequence of events to occur:

1. The child sends `SIGINT` to the parent.
2. The parent process catches the signal, but is then preempted by the kernel.
3. The child executes again, sending another signal to it's parent.

4. The parent receives the second signal, but has not had a chance to reinstate the user-defined handler, thus the default action for the signal is made and the parent process exits.

The likelihood of these events occuring is increased by the fact that the parent process executes at a lower priority than the child. This is called a *race condition*, and it is the programmers responsibilty to avoid code that might generate a race.

BSD versions of UNIX handle signals in a more sensible way. After a signal has been delievered to a process, the default handler for that signal is *not* reinstalled, and so race conditions can not be generated. Solaris supports BSD signals via the BSD compatibility library. The BSD Compatibility library should be avoided however if portability and maintenance are high priority considerations.

1.5 Exercises

1. Rewrite the "Hello World" program so that "Hello" is *always* printed before "World". Note that you must still use `fork()`.
2. Write a UNIX menu program for "dummies" that allows the commands `ls`, `vi` and `mail` to be executed. You must allow optional arguments to be provided to the commands.
3. Write a utility which displays the hierarchy of processes currently associated with the command shell. (The parent of your utility will be the command shell itself)
4. Write some code that traps the `SIGINT` signal and asks the user if they really wish to terminate the process. If the user *does* wish to terminate the process the perform he appropriate action, otherwise reinstate the signal handler.

Chapter 2

Files and The File System

2.1 Introduction

Anyone familiar with C programming should already have a good grasp of the ideas behind the UNIX file system. We will not cover file streams or the `FILE` data structure in this course because the probability of being redundant is extremely high.

We will cover file *descriptors*, and the system calls that operate on them. In Section 2.2 we cover basic file descriptor functions. In Section 2.3 we look at the UNIX directory structure and the system calls which manipulate it.

2.2 File Manipulation

A *file descriptor* is a small integer value that represents an open file. File descriptors are created with the `open()` and `creat()` system calls. All of the system calls we will investigate here use descriptors to indicate which file is being affected. Each process has it's own *file descriptor table* from which the `open()`, `creat()` and `dup()` system calls obtain new descriptors.

The `open()` System Call

The synopsis for `open()` is:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *path, int  oflag,  /*  mode_t  mode  */ ...);
```

The `open()` system call opens a file for reading or writing. It takes two or three arguments. The `path` argument is the path of the file to be opened. The `mode` argument specifies the access permissions to the file for the user, group, and others using the UNIX octal permission bits. `oflag` allows various options to be set by ORing the following constants together:

`O_RDONLY` Open a file for reading only.

`O_WRONLY` Open a file for writing only.

`O_RDWR` Open a file for reading and writing.

`O_NDELAY` Don't block on reading or writing.

`O_APPEND` Write to the end of the file.

`O_CREAT` Create the file if it doesn't exist.

`O_EXCL` Open the file *only* if it doesn't exist (with `O_CREAT`).

`O_TRUNC` Truncate the file if it exists.

`open()` returns a valid file descriptor if successful or -1 otherwise.

The `creat()` System Call

The synopsis for `creat()` is:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int creat(const char *path, mode_t mode);
```

`creat()` creates a new regular file or prepares to rewrite an existing file named by `path`. The `mode` argument specifies the access permissions to the file for the user, group, and others using the UNIX octal permission bits. `creat()` returns a valid file descriptor if successful or -1 otherwise.

The `close()` System Call

The synopsis for `close()` is:

```
#include <unistd.h>

int close(int fildes);
```

`close()` takes a single argument which is the descriptor of the file to close. It returns 0 if successful or -1 otherwise. If `close()` is successful, the file descriptors entry in the file descriptor table is marked free.

The `read()` System Call

The synopsis for `read()` is:

```
#include <sys/types.h>
#include <unistd.h>

size_t read(int fildes, void *buf, size_t nbyte)
```

The `read()` system call reads in `nbyte` bytes into `buf` from the file represented by `fildes`. `read()` returns the number of bytes successfully read from the file.

The `write()` System Call

The synopsis for `write()` is:

```
#include <unistd.h>

ssize_t write(int fildes, const void *buf, size_t nbyte);
```

The `write()` system call writes `nbyte` bytes from `buf` to the file represented by `fildes`. `write()` returns the number of bytes successfully written to `fildes`.

The `lseek()` System Call

The synopsis for `lseek()` is:

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek(int fildes, off_t offset, int whence);
```

The `lseek()` system call moves the file pointer for `fildes` by `offset` bytes from `whence`. `whence` is determined from one of the following three constants:

`SEEK_SET` The pointer is set to `offset` bytes from the start of the file.

`SEEK_CUR` The pointer is set to the current position plus `offset` bytes.

`SEEK_END` The pointer is set to the size of the file plus `offset` bytes.

If `lseek()` is successful is returns the new file pointer location relative to the start of the file.

The `unlink()` System Call

The synopsis for `unlink()` is:

```
#include <unistd.h>

int unlink(const char *path);
```

The `unlink()` system call removes the file indicated by `path` from the file system. The deletion of the file is irreversible. `unlink()` will remove symbolic links, but should not be used to remove directories. `unlink()` returns 0 if successful and -1 otherwise. If `unlink()` fails, it is most likely due to an ownership permission problem.

The `link()` System Call

The synopsis for `link()` is:

```
#include <unistd.h>

int link(const char *existing, const char *new);
```

The `link()` system call creates a new directory entry for a file that already exists. This system call effectively allows a file to have more than one name. `link()` returns 0 if successful and -1 otherwise.

The chmod() System Call

The synopsis for chmod() is:

```
#include <sys/types.h>
#include <sys/stat.h>

int chmod(const char *path, mode_t mode);
int fchmod(int fildes, mode_t mode);
```

The chmod() system call changes the file permissions of the file indicated by path according to the value of mode. The fchmod() call is identical to chmod() except that fildes indicates which file to change permissions on.

The chown() System Call

The synopsis for chown() is:

```
#include <unistd.h>
#include <sys/types.h>

int chown(const char *path, uid_t owner, gid_t group);
int fchown(int fildes, uid_t owner, gid_t group);
```

The chown() and fchown() system calls change the owner ID and group ID of the target file. Ownership of a file can only be changed if the user is root or the current owner of the file is changing it's ownership. The chown() and fchown() calls return 0 if successful and -1 otherwise.

The dup() System Call

The synopsis for dup() is:

```
#include <unistd.h>

int dup(int fildes);
```

The dup() system call returns a new file descriptor which has in common with fildes:

- The same open file or pipe.
- The same file pointer.
- The same access mode (O_RDONLY, O_WRONLY or O_RDWR).

dup() always returns the first available file descriptor. dup() will return a valid file descriptor if successful or -1 otherwise.

The fcntl() System Call

The synopsis for fcntl() is:

```
#include <sys/types.h>
#include <fcntl.h>

int fcntl(int fildes, int cmd, /* arg */ ...);
```

The `fcntl()` system call gives programmers more control over open file descriptors. `fildes` is the file descriptor to be operated on, and `cmd` is a function to perform. There are many constants defined in `fcntl.h` which specify commands. Some are listed here:

`F_DUPFD` Duplicate a file descriptor (same as `dup()`).

`F_GETFD` Get the close-on-exec flag.

`F_SETFD` Set the close-on-exec flag.

`F_GETFL` Get the `filedes` status flags.

`F_SETFL` Set the `filedes` status flags.

The `fstat()` System Call

The synopsis for `fstat()` is:

```
#include <sys/types.h>
#include <sys/stat.h>

int stat(const char *path, struct stat *buf);
int fstat(int fildes, struct stat *buf);
```

The `stat()` and `fstat()` system calls allow the programmer to obtain information about a file. `stat()` is used when the file is not currently open, and `fstat()` is used when the file is opened and `fildes` is available. `buf` is a pointer to the `stat` structure defined in `sys/stat.h`:

```
struct  stat {
     dev_t    st_dev;        /* device major and minor numbers */
     ino_t    st_ino;        /* inode number                   */
     mode_t   st_mode;       /* file type and permissions      */
     nlink_t  st_nlink;      /* number of links                */
     uid_t    st_uid;        /* owner user ID                  */
     gid_t    st_gid;        /* owner group ID                 */
     dev_t    st_rdev;       /* used for devices               */
     off_t    st_size;       /* size of the file               */
     time_t   st_atime;      /* last access time               */
     time_t   st_mtime;      /* last modification time         */
     time_t   st_ctime;      /* last time stat info modified   */
};
```

2.3 Directory Manipulation

Directories are special files which contain other files under UNIX. The directories of a file system form a hierarchy, with "/" being the root of the heirarchy. System calls are available for changing, creating, removing and searching directories.

The `chdir()` System Call

The synopsis for `chdir()` is:

```
#include <unistd.h>

int chdir(const char *path)
```

The chdir() system call changes the current working directory to path. It returns 0 if successful or -1 otherwise.

The getcwd() System Call

The synopsis for getcwd() is:

```
#include <unistd.h>

extern char *getcwd(char *buf, size_t size);
```

The getcwd() system call returns the current working directory. If buf is 0, then getcwd() allocates enough dynamic memory to store the path, otherwise the path will be stored in buf. size must be at least two bytes bigger than the required number of characters to store the path. getcwd() returns a pointer to a string which stores the path of the current directory.

The rmdir() System Call

The synopsis for rmdir() is:

```
#include <unistd.h>

int rmdir(const char *path);
```

rmdir() removes the directory specified by path providing that the directory is empty. rmdir() returns 0 if successful and -1 otherwise.

The mkdir() System Call

The synopsis for mkdir() is:

```
#include <sys/types.h>
#include <sys/stat.h>

int mkdir(const char *path, mode_t mode);
```

The mkdir() system call creates a new directory based on path and sets it's access permission bits to mode. mkdir() returns 0 if successful and -1 otherwise.

The ftw() System Call

The synopsis for ftw() is:

```
#include <ftw.h>

int ftw(const char *path, int (*fn) (char *, struct stat *, int), int depth);
```

The ftw() system call provides a file tree walk capability. It takes three arguments: path is the path to start the tree walk at. fn is a user-defined function that is called for every entry found during the file tree walk. depth specifies the number of file descriptors to use during the file tree walk. The more descriptors used the faster the walk will proceed. Be careful in the value you specify for depth – there is only a finite number of file descriptors available per process.

The fn() user-defined function takes three arguments. A template for this function is provided below:

```
int fn(char *name, struct stat *buf, int type);
```

The user-defined function should always return zero unless you wish to terminate the file tree walk. `name` is the name of the directory entry currently under scrutiny. `type` indicates what type of file is currently under scrutiny, and will have one of the following constant values declared in `ftw.h`:

`FTW_F` The object is a file.

`FTW_D` The object is a directory.

`FTW_DNR` The object is an unreadable directory.

`FTW_NS` The object caused `stat()` to fail.

The code below illustrates the use of `ftw()` to list a directory hireracy:

```
/* ftw.c -- file tree walk demonstration */

#include <sys/types.h>
#include <sys/stat.h>
#include <ftw.h>

int list(char *name, struct stat *status, int type)
{
     if (type == FTW_NS)
          return 0;
     if (type == FTW_F)
          printf("%-30s\t0%3o\n", name, status->st_mode & 0777);
     else
          printf("%-30s*\t0%3o\n", name, status->st_mode & 0777);
     return 0;
}

int main(int argc, char *argv[])
{
     int list();

     if (argc <= 1)
          ftw(".", list, 1);
     else
          ftw(argv[1], list, 1);
     exit(0);
}
```

Chapter 3

Interprocess Communication

3.1 Introduction

Interprocess communication is an important part of any time-sharing computer system. Modern day applications demand multiple communicating processes for speed and efficiency. We have already examined one form of interprocess communication when we looked at signals, but signals are limited by both the information they can transmit and the speed of operation.

In Section 3.2 we look at *pipes* for communicating between processes, and in Section 3.3 we investigate *semaphores*, *shared memory* and *messages*.

3.2 Pipes

A pipe is a uni-directional communications channel which couples one process to another. Bi-directional communications is easily accomplished between processes by using two pipes. Data can be written to and read from pipes using the standard `write()` and `read()` I/O routines.

3.2.1 Unnamed Pipes

An *unnamed pipe* is a pipe created between two processes that is valid as long as the process which reads from the pipe exists. An unnamed pipe is created with the `pipe()` system call. The synopsis for `pipe()` is given below:

```
#include <unistd.h>

int pipe(int filedes[2]);
```

The `pipe()` system call takes a signle argument, which is a pointer to an array of two integers. `filedes[0]` is a descriptor for reading from the pipe and `filedes[1]` is a descriptor for writing to the pipe. The `pipe()` call returns 0 if successful and -1 otherwise.

The code below illustrates using a pipe for communicating between two processes. In this simple example a pipe is created, then a child is forked. the child process writes "Hi there" to the pipe. The parent waits for the child to terminate and then reads from the pipe and displays the result.

```
/* pipe.c -- demonstrates a pipe for communication between processes */

#include <stdio.h>
```

```
#include <unistd.h>

#define error(x)        { perror(x); exit(-1); }

main()
{
     int pfds[2];
     int pid;
     char buffer1[] = "Hi there\n";
     char buffer2[] = "          ";

     if (0 > pipe(pfds))
          error("pipe() failed");

     pid = fork();

     if (pid == 0)
          write(pfds[1], buffer1, sizeof buffer1);
     else {
          wait((int *) 0);
          read(pfds[0], buffer2, sizeof buffer2);
          printf("%s", buffer2);
     }
}
```

Pipes have a fixed size. If a process continually writes to a pipe, with no other process reading from it, then eventually the `write()` system call will block until a process reads from the pipe, making more space.

It is possible to stop `write()` from blocking using the `fcntl()` system call with the `O_NDELAY`. The `read()` system call can also be stoped from blocking when there is no data available in the pipe using `fcntl()` with `O_NDELAY`. An example for preventing `write()` from blocking is shown below:

```
#include <fcntl.h>
  .
  .
  .
fcntl(filedes, F_SETFL, O_NDELAY);
```

Clever use of `pipe()`, `dup()`, `fork()` and `exec()` will enable you to execute programs with `stdin` and `stdout` being redirected.

3.2.2 Named Pipes

Unnamed pipes do not exist permanently in the system. They also can only connect processes that share a common ancestry, which severely limits the IPC abailities of pipes. To overcome these problems, *named pipes* were introduced.

Named pipes are really a special kind of file known as a FIFO (first-in, first out). Named pipes are created with the `mknod()` system call. Once a named pipe has been created, it can be opened for reading or writing with the `open()` system call, just like an ordinary file.

The `mknod()` System Call

The `mknod()` system call is used to create normal files, character special files, block special files and FIFOs. `mknod()` takes three arguments. The first argument specifies the `path` of the file to be created. The second argument, `mode`, specifies both the type and access permissions for the file. The type of file to be created is determined by one of these four constants defined in `stat.h`:

`S_IFREG` A regular file.

`S_IFCHR` A character special file.

`S_IFBLK` A block special file.

`S_IFIFO` A FIFO file.

If `S_IFCHR` or `S_IFBLK` is specified then `dev` specifies the major and minor device numbers for a device. We will cover devices later in the course. `mknod()` return 0 on success or -1 otherwise.

Below is some code analogous to the unnamed pipe example, except that it uses named pipes.

```
/* npipe.c -- demonstrates named pipes for interprocess communication */
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

#define error(x)        { perror(x); exit(-1); };

main()
{
     int fd;
     int pid;
     char buffer1[] = "Hi there\n";
     char buffer2[] = "          ";

     mknod("fifo", 010777, 0);
     pid = fork();
     if (pid == 0)
     {
          fd = open("fifo", O_WRONLY);
          write(fd, buffer1, sizeof buffer1);
          close(fd);
     } else {
          fd = open("fifo", O_RDONLY);
          read(fd, buffer2, sizeof buffer1);
          close(fd);
          printf("%s", buffer2);
     }
}
```

3.3 SVID Compilance

3.3.1 Introduction

In 1985 AT&T introduced the *System V Interface Definition (SVID)*. SVID introduced record locking, which we have already examined. It also introduced and standardized some very important interprocess communication facilities, which are described as the *IPC package*. Three sets of IPC facilities were introduced:

1. *Message passing.* The message passing facility allows a process to send and receive messages; a message being in essence an arbitary sequence of bytes.

2. *Semaphores.* Compared with message passing, semaphores provide a rather low-level means of process synchronisation, not suited to the transmission of large amounts of information. Semaphores are extremely efficient however, and are widely used.

3. *Shared memory.* This final IPC facility allows two or more processes to share data contained in specific memory segments. Shared memory represents an extremely efficient method of sharing data between processes, but relies of hardware support. Nearly all modern day computer systems provide the required level of hardware support.

Although these facilities are part of UNIX SYSV, they are not an integral part of the kernel. During installation of UNIX, the system administrator has the option of enabling or disabling SVID IPC features. The `ipcs` shell command can be used to verify which IPC facilities have been installed.

3.3.2 IPC Facility Keys

Before we start looking at each IPC facility in earnest, we will describe the commonalities that these facilities share.

The programming interfaces of semaphores, shared memory and messages are very similar. The most important common feature of these interfaces is the IPC facility key. Numerical keys are used to identify IPC objects. For any process to be able to access an IPC object, it must know the value of the key for that object. IPC keys have the type `key_t` which is defined in `types.h`.

The major problem with keys is avoiding the use of the same key value for different IPC objects. There is no standardized method for key allocation, so it is up to the system programmer to choose a unique one. Good documentation of used keys and the use of the `ipcs` UNIX command for viewing currently used keys should help alleviate the problem.

A routine called `ftok()`, found in most standard C libraries, returns a unique key based on a file system `path` which is specified as an argument. A second argument to `ftok()`, called `id`, allows further levels of uniqueness to be specified (up to 256 levels). The usage of `ftok()` is shown below:

```
#include <sys/types.h>
#include <sys/ipc.h>

key_t key, ftok();
char *path, id;
  .
  .
  .
key = ftok(path, id);
```

The programmer must be wary of the path they choose to use for `ftok()`. If the path is changed, then changed again to reflect it's original state, `ftok()` will return *different* keys. It is probably better to avoid `ftok()` and choose IPC keys yourself.

3.3.3 IPC Operations

There are three classes of IPC operations:

1. *Get operations.* Get operations are used for creating IPC objects and returning *facility identifiers* which are used by the other IPC operations.

2. *Control operations.* Control operations are used for obtaining status information, changing status information, and removing IPC objects.

3. *Specific operations.* These operations are used for manipulating IPC objects, such as sending a message or changing the value of a semaphore.

Permission to access an IPC object must be available before any operations on that object can be performed. Like files, IPC objects have owners, belong to groups, and have read/write permission bits for users, groups and others.

3.3.4 Message Queues

A message is a sequence of bytes to be transmitted from one process to another. Messages are passed between processes via *message queues*. Message queues are created with the `msgget()` system call. Messages can be sent and received by the `msgsnd()` and `msgrcv()` system calls. The `msgctl()` system call serves three purposes: it allows a process to obtain the status of the message queue, to change some of the limits associated with the message queue, or to delete the message queue from the system.

The `msgget()` System Call

The `msgget()` system call takes two arguments: an IPC key that specifies the message queue and a set of flags which determine: (a) if a new or existing queue is to be used, and (b) the permissions for that queue. `msgget()` returns an integer that represents the message queue handle. If `msgget()` fails then it returns -1. The synopsis for `msgget()` is given below:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t key, int msgflg);
```

The `msgflg` parameter determines the exact action performed by `msgget()`. Two constants are of relevance here, both defined in `ipc.h`. They can be ORed together if necessary:

`IPC_CREAT` This tells `msgget()` to create a message queue for the value `key` if one does not already exist. If `IPC_CREAT` is not specified, then a message queue identifier is only returned if the queue already exists.

`IPC_EXCL` This used in conjunction with `IPC_CREAT` will cause `msgget()` to return a message queue identifier only if the message queue did not previously exist.

Along with the two constants documented above, a number representing the read/write permissions for the user, group and others can be specified in the same manner that permissions are specified for files. For example, if we wanted to exclusively create a new message queue so that only the owner of the queue can use it, we would specify `0600|IPC_CREAT|IPC_EXCL` for the `msgflg` argument to `msgget()`.

The `msgsnd()` and `msgrcv()` System Calls

`msgsnd()` and `msgrcv()` send and receive messages respectively. The synopsis for `msgsnd()` and `msgrcv()` follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);

int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);

struct msgp {
     long mtype;
     char mtext[SOMEVALUE];
};
```

`msqid` is the message queue identifier returned by `msgget()`. `msgp` is a pointer to a user-defined data structure which contains the message to be sent or received. `struct msgp` is an example of a user defined message. The `mtype` field allows messages to be categorized or prioritized. `msgsz` is the size of the message to be sent or received in bytes.

The `msgtyp` argument to `msgrcv()` allows for the receipt of messages based on the catergory or priority of the message specified in it's `mtype` field. If `msgtyp` is 0 then the first message on the queue will be retreived. If `msgtyp` is a number greater than 0 then the first message in the queue with that number will be retreived. Finally, if the value of `msgtyp` is negative then the first message with `mtype` value less than or equal to the absolute of `msgtyp` is retreived.

The `msgflg` argument to `msgsnd()` and `msgrcv()` is used to specify control options. For `msgsnd()`, if `IPC_NOWAIT` is specified and there are not sufficient system resources to send the message, `msgsnd()` will fail. If `IPC_NOWAIT` is *not* set, then the calling process will sleep until resources are available. For `msgrcv()`, if `IPC_NOWAIT` is specified and no messages are available to be received then `msgrcv()` will return to the caller immediately, otherwise it will wait.

The `IPC_NOERROR` flag can also be set for `msgrcv()`. This causes messages that are bigger than `msgsz` to be received but truncated. Normally, a message that is larger than `msgsz` would cause `msgrcv()` to fail.

The `msgctl()` System Call

The `msgctl()` system call allows message queue to be removed, modified or queried about its state. `msgctl()` takes three arguments, and returns 0 on success or -1 if an error is detected. The synopsis for `msgctl()` follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl(int msqid, int cmd, /* struct  msqid_ds  *buf  */ ...);
```

`msqid` is the message queue identifier returned by `msgget()`. `buf` is a pointer to a structure which is used for storing control variables for the message queue. Refer to `sys/msg.h` for the details of the `msqid_ds` structure. There are three options for the `cmd` argument, which are described below:

`IPC_STAT` Tells the system to place status information about the queue into `buf`.

`IPC_SET` Allows some of the control variables for a message queue to be changed. The only fields of the `msqid_ds` structure that can be changed are:

`msqid_ds.msg_perm.uid`

`msqid_ds.msg_perm.gid`

msqid_ds.msg_perm.mode

msqid_ds.msg_qbytes

IPC_RMID This removes the queue from the system.

Note that the IPC_SET and IPC_RMID cammoands can only be executed by the owner of the message queue or by the superuser.

An Example of Message Queues

The following code uses messages to implement a client–server system. The server waits for messages from any client in an infinite loop. The server *must* be running before a client is executed. When a client is executed, it sends a message to the server which contains it's PID number. The server then prints to the terminal that it has received a message from a client and then sends its own PID back to the client. The client then displays the servers PID and terminates.

```
/* server.c -- demonstration of messages & client-server programming */

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MSGKEY 75

struct msgform {
    long mtype;
    char mtext[256];
};

int msgid;

main()
{
    struct msgform msg;
    int i, pid, *pint;
    extern cleanup();

    for (i = 0; i < 20; ++i)
        signal(i, cleanup);

    msgid = msgget(MSGKEY, 0777 | IPC_CREAT);

    while (1)
    {
        msgrcv(msgid, &msg, 256, 1, 0);
        pint = (int *) msg.mtext;
        pid = *pint;
        printf("server: receive from pid %d\n", pid);
        msg.mtype = pid;
        *pint = getpid();
        msgsnd(msgid, &msg, sizeof(int), 0);
    }
}

cleanup()
```

```
{
    msgctl(msgid, IPC_RMID, 0);
    exit();
}
```

```
/* client.c -- demonstration of messages & client-server programming */

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MSGKEY 75

struct msgform {
    long mtype;
    char mtext[256];
};

main()
{
    struct msgform msg;
    int msgid, pid, *pint;

    msgid = msgget(MSGKEY, 0777);

    pid = getpid();
    pint = (int *) msg.mtext;
    *pint = pid;
    msg.mtype = 1;

    msgsnd(msgid, &msg, sizeof(int), 0);
    msgrcv(msgid, &msg, 256, pid, 0);
    printf("client: receive from pid %d\n", *pint);
}
```

3.3.5 Shared Memory

Shared memory allows two or more processes to share a physical memory segment. Hardware support is required for shared memory, but most modern computers systems that support virtual paged or segmented memory by definition have the necessary hardware to support shared memory.

In order for a process to use shared memory, the physical memory set aside for use between multilpe processes must first be *attached* to the processes address space. Later, when the process no longer required shared memory, the shared segment is *detached*.

The `shmget()` System Call

The **`shmget()`** system call takes three arguments: a **`key`** associated with the shared memory object, a **`size`** which specifies the required minimum amount of shared memory, and flags which are the same as for the **`msgflg`** argument of the **`msgget()`** system call. The synopsis for **`shmget()`** follows:

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

```
int shmget(key_t key, int size, int shmflg);
```

The shmget() call returns a shared memory identifier if successful or -1 if an error is detected.

The shmat() and shmdt() System Calls

The shmat() routine attaches shared memory to a process while shmdt() detaches the memory when it is no longer required. The synopsis for shmat() and shmdt() follows:

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

void *shmat(int shmid, void *shmaddr, int shmflg);

int shmdt(void *shmaddr);
```

The shmat() system call takes three arguments: shmid is the shared memory identifier returned by a call to shmget(), shmaddr is an address where the programmer would prefer the shared memory segment to reside, and shmflg allows the user to specify whether the shared memory is read-only with the SHM_RDONLY constant. shmat() returns the address of the shared memory segment if successful or (char *) -1 if an error occurs.

If the shmaddr argument to shmat() is 0, then shmat() automatically chooses a start address for the shared memory segment. If shmaddr is non-zero, then it specifies a perfered memory location for the shared segment. By specifying the SHM_RND flag, the address given by shmaddr will be rounded to the nearest page boundary in memory. The use of a non-zero value for shmaddr is discouraged because it requires the programmer to have intimate knowledge about the layout of the program in memory.

The shmdt() call performs the opposite function of shmat(); that is, it detaches shared memory from a process. It takes a single argument shmaddr which is the address of the shared segment returned by shmat(). shmdt() returns 0 if successful and -1 if an error occurs.

The shmctl() System Call

The shmctl() system call exactly parallels msgctl(), and cmd can take the values IPC_STAT, IPC_SET and IPC_RMID. The synopsis for shmctl() follows:

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

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Refer to sys/shm.h for details of the shmid_ds structure.

An Example of Shared Memory

The following program demonstrates the use of shmget(), shmat() and shmctl(). First a shared memory region of 128K bytes is created with shmget(). Then the process uses shmat() twice to attach the shared region to two different virtual addresses. The second virtual address is for read-only memory. The first 16 words of the shared memory region is then filled with the numbers 0 to 15, and then the contents are read back via the second virtual address and displayed. The program then suspends itself. Any signal sent to the process will cause the shared memory to be destroyed and the process to exit.

```
/* shm1.c -- example of attaching shared memory twice to a process */

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define  SHMKEY 75
#define  K      1024
int shmid;

cleanup()
{
    shmctl(shmid, IPC_RMID, 0);
    exit();
}

main()
{
    int i, *pint;
    char *addr1, *addr2;

    for (i = 0; i < 20; ++i)
        signal(i, cleanup);
    shmid = shmget(SHMKEY, 128 * K, 0777|IPC_CREAT);
    addr1 = (char *) shmat(shmid, 0, 0);
    addr2 = (char *) shmat(shmid, 0, SHM_RDONLY);
    printf("addr1 0x%x addr2 0x%x\n", addr1, addr2);
    pint = (int *) addr1;

    for (i = 0; i < 16; ++i)
        *pint++ = i;
    pint = (int *) addr1;
    *pint = 16;

    pint = (int *) addr2;
    for (i = 0; i < 16; ++i)
        printf("index %d\tvalue %d\n", i, *pint++);

    pause();
}
```

The program below attaches itself to the shared memory region created in the program above and reads the first 16 words, printing out the value of each word. Its output ought to be exactly the same as the output of the program above if the shared memory is working correctly.

```
/* shm2.c -- example of sharing memory between two processes */

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define  SHMKEY 75
#define  K      1024
int shmid;

main()
{
    int i, *pint;
```

```
    char *addr;

    shmid = shmget(SHMKEY, 64 * K, 0777);

    addr = (char *) shmat(shmid, 0, 0);
    pint = (int *) addr;

    while (*pint == 0);

    for (i = 0; i < 16; ++i)
        printf("index %d\tvalue %d\n", i, *pint++);
}
```

3.3.6 Semaphores

The semaphore concept was first put forward by Dutch theoretician, E. W. Dijkstra, as a solution to the problem of process synchronisation. A semaphore can be considered as an integer variable on which two atomically indivisable operations can be performed. The operations are called *wait* and *signal*, the latter not to be confused with the UNIX `signal()`. The C pseudocode below gives the definitions of these operations.

```
void wait(semaphore s)
{
     if (s != 0)
          --s;
     else
          while (s == 0);
}

void signal(semaphore s)
{
     ++s;
}
```

Semaphores are used to ensure that only one process at any time can be utilizing a resource. They can provide *mutual exclusion* between processes and synchronisation.

The implementation of SVID semaphores allows for semaphore *sets*. That is, an IPC semaphore object may contain one or more semaphores. This results in a more complex programming interface, but affords greater flexibility and power.

The semget() System Call

The `semget()` system call takes three arguments: a key for the IPC object, a number `nsems` which specifies the number of semaphores in the set to be created, and some flags. `semget()` returns a semaphore set identifier if successful and -1 if an error is detected. The synopsis for `semget()` follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

Note that the legal values for `semflg` are the same as those for the `msgflg` argument of the `msgget()` system call. The semantics of the flags are also identical.

The `semop()` System Call

The `semop()` system call has less than straight-forward semantics. It is very powerful however. `semid` is a semaphore set identifier returned by the `semget()` call. `nsops` gives the number of semaphores in the set to be operated on. `sops` is an array of `nsops` sembuf structures which determine the operation performed on each individual semaphore. The synopsis for `semop()` follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, size_t nsops);
```

The `sembuf` structure specifies which semaphore to operate on, the operation itself, and some flags. The fields of the `sembuf` structure that the programmer needs to know about are:

`short sem_num` The semaphore to operate on (first semaphore at 0).

`short sem_op` The operation to execute.

`short sem_flg` The flags for the operation.

The behaviour for `semop()` is determined by the value of `sem_op` on a per-semaphore basis. The following pseudocode summarises the possible behaviours:

```
case: sem_op < 0
      if (semval >= ABS(sem_op)
           semval = semval - ABS(sem_op);
      else
           if (sem_flg & IPC_NOWAIT)
                return -1;
           else
                while (semval < ABS(sem_op));
                semval = semval - ABS(sem_op);
           endif
      endif

case: sem_op > 0
      semval = semval + sem_op;

case: sem_op = 0
      if (sem_flg & IPC_NOWAIT and semval != 0)
           return -1;
      else
           while (semval != 0);
      endif
```

It should be clear from this pseudocode that the `IPC_NOWAIT` flag prevents the `semop()` routine from blocking. Another flag, `SEM_UNDO`, should always be specified. It tells the system to adjust ("undo") the semaphore values appropriately when a process terminates.

The `semctl()` System Call

The `semctl()` system call takes four arguments. `semid` is a semaphore set identifier returned by `semget()`. `semnum` identifies a particular semaphore for single semaphore operations. `cmd` indicates which control operation is to be performed. `arg` is a pointer to a union for getting and setting control options. `semctl()` returns -1 if an error is detected, or 0 or a positive integer if successful, depending on `cmd`. The synopsis for `semctl()` follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, /* union semun arg */ ...);

union semun {
    int val;
    struct semid_ds *buf;
    ushort *array;
};
```

The `semctl()` system call supports many different commands. The table below defines and describes al of the command available to `semctl()`:

Standard IPC functions

`IPC_STAT`	Place status information into `arg.stat`
`IPC_SET`	Set ownership/permissions from `arg.stat`
`IPC_RMID`	Remove semaphore set from system

Single semaphore operations

`GETVAL`	Return value of a semaphore
`SETVAL`	Set value of a semaphore
`GETPID`	Get PID of the last process to access a semaphore
`GETNCNT`	Return number of processes waiting `semval` to increase
`GETZCNT`	Return number of processes waiting `semval` to reach 0.

Set-based semaphore operations

`GETALL`	Place all `semvals` into `arg.array`
`SETALL`	Set all `semvals` from `arg.array`

An Example of Semaphores

Below is a set of library routines that illustrate the use of SVID semaphores and hide many of the complexities of the semaphore system calls. Four functions have been created, each of which takes a single argument.

`initsem()` creates a semaphore, taking they key for the semaphore as an argument and returning a semaphore set identifier. `waitsem()` is analoguous to the definition of *wait* given at the start of Section 3.3.6. `postsem()` is analoguous to the definition of *signal*, also given at the start of Section 3.3.6. `destsem()` removes a semaphore from the system.

```
/* semaphore.h -- include file for semaphore library */

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <errno.h>
#include <unistd.h>

extern int errno;

#define SEMPERM 0600
```

```
#define TRUE 1
#define FALSE 0

int initsem(key_t);
int postsem(int);
int waitsem(int);
void destsem(int);


/* semaphore.c -- source code for semaphore library */

#include "semaphore.h"

int initsem(key_t key)          /* create a semaphore */
{
     int status = 0;
     int semid;

     if ((semid = semget(key, 1, SEMPERM|IPC_CREAT|IPC_EXCL)) == -1) {
          if (errno == EEXIST)
               semid = semget(key, 1, 0);
     } else
          status = semctl(semid, 0, SETVAL, 1);

     if (semid == -1 || status == -1) {
          perror("initsem() failed");
          return (-1);
     } else
          return semid;
}

int waitsem(int semid)          /* wait on a semaphore */
{
     struct sembuf p_buf;

     p_buf.sem_num = 0;
     p_buf.sem_op = -1;
     p_buf.sem_flg = SEM_UNDO;

     if (semop(semid, &p_buf, 1) == -1) {
          perror("waitsem() failed");
          exit(1);
     } else
          return (0);
 }

int postsem(int semid)          /* post to a semaphore */
{
     struct sembuf v_buf;

     v_buf.sem_num = 0;
     v_buf.sem_op = 1;
     v_buf.sem_flg = SEM_UNDO;

     if (semop(semid, &v_buf, 1) == -1) {
          perror("postsem() failed");
```

```
            exit(1);
        } else
            return (0);
}

void destsem(int semid)          /* destroy a semaphore */
{
        semctl(semid, 0, IPC_RMID, 0);
}
```

The program below illustrates the use of semaphore library above. It is a two process version of the "Hello World" program that ensures "Hello" is printed before "World" by using semaphores for synchronisation.

```
/* sem_test.c -- demostrates use of the semaphore library */

#include <stdio.h>
#include <time.h>
#include "semaphore.h"

main(void)
{
        int semid;
        int pid;
        key_t key;

        key = ftok("~/.cshrc", 1);
        semid = initsem(key);
        pid = fork();
        if (pid == 0) {
            printf("Hello ");
            postsem(semid);
            exit(0);
        } else {
            waitsem(semid);
            printf("World");
        }
        wait((int *) 0);
        putchar('\n');
        destsem(semid);
        exit(0);
}
```

3.4 Exercises

1. Write a program that uses `pipe()`, `dup()`, `fork()` and `exec()` to redirect the output of the `ps` UNIX command to a file.

2. Write a program that uses shared memory and semaphores *or* messages to copy a file. Your program should utilise *two* processes: One for reading the source file and another one for writing the destination file.

Chapter 4

Threads

4.1 Introduction

A *thread* is a sequence of executable instructions. The traditional UNIX process can be viewed as a thread. Solaris differs from the traditonal UNIX model of a process because a process consist of *one or more* threads of executable code. The threads of a process in Solaris execute concurrently and share the same address space.

All of the global data belonging to a process is viewed as *shared memory* by threads. That is, every thread in a process can read and write to the processes global data. Local data in a function belonging to a thread is not accessible to other threads, unless the data is declared static. Even then, two threads must be executing the same function before the static data can be shared.

Threads provide several advantages over processes:

1. Creation of a new thread is rapid because it is not necessary to copy any context information.
2. Scheduling and dispatching threads belonging to the same process is rapid because the amount of context information to change is minimal.
3. Shared memory, which is the most efficient form of IPC, is an inherent feature of threads belonging to a signle process.

Each thread belonging to a process has it's own signal mask, `errno` variable, and stack. Individual threads can have different scheduling priorities, and a *concurrency level* can be specified per process, which determines the number of threads that can execute in parallel on a multiprocessor computer.

Threads on Solaris are executed by *Light Weight Processes* (LWP's). An LWP can be used to execute one or more threads, but each LWP can only execute a single thread at a time. The concurrency level for a process is determined by the number of LWP's available to the process.

Unbound threads share a LWP. A *bound* thread has a LWP all to itself. The importantance of bound threads will be discusses in the next chapter, which deals with *real time* processing.

4.2 The Threads Library

The threads library is contained in the file `libthread.a`. To compile a threaded program you must include the library like so:

```
gcc -o thr_example thr_example.c -lthread
```

As an aside, use the `gcc` compiler over Sun's `cc` when it is available. It is a much faster compiler.

While every thread has it's own `errno` variable, the majority of the thread library routines return the `errno` value if an error is detected. If no error is detected, then 0 is returned.

4.2.1 Thread Creation

The `thr_create()` library routine creates a new thread of execution for the calling process. Every thread has it's own *thread identifier*, which is analogous to a processes PID. The synopsis for `thr_create` is given below:

```
#include <thread.h>

int thr_create(void *stk_b, size_t stk_sz, void *(*start)(void *),
     void *arg, long flags, thread_t *new_thread);
```

The `stk_b` and `stl_sz` arguments to `thr_create()` specify the base and size of the new threads stack respectively. If `stk_b` equals `NULL` and `stl_sz` equals 0 then `thr_create()` will automatically allocate a stack of appropriate size the the new thread. `start` is a pointer to a function which acts like `main()` for the thread. When `start` returns the thread temrinates. `arg` allows a single argument to be passed to `start` when the thread first executes. `flags` determines the behaviour of the new thread. Any of the following constants defined in `thread.h` can be ORed together to obtain a particular behaviour:

`THR_SUSPENDED` Creates the thread but does not execute it.

`THR_DETACHED` The thread is created detached.

`THR_BOUND` The new thread is bound to its own LWP.

`THR_NEW_LWP` Causes a new LWP to be added to the pool of LWPs.

`THR_DAEMON` The thread is marked as a daemon. The process will exit when all non-daemon threads exit.

The `new_thread` argument is a pointer to the `thread_t` variable which stores the threads identifier.

4.2.2 Thread Joining

The `thr_join()` library routine is the parallel of the `wait()` system call for processes. `thr_join()` blocks until the thread indicated by `wait_for` terminates. The synopsis for `thr_join()` follows:

```
#include <thread.h>

int thr_join(thread_t wait_for, thread_t *departed, void **status);
```

A *detached* thread, which is created with the `THR_DETACHED` flag set during `thr_create()`, can not be waited on with `thr_join()`. If `wait_for` equals `(thread_t) 0`, then `thr_join()` blocks until any undetached thread terminates. If `departed` is not `NULL`, then the thread identifier of the terminated thread is stored in the location pointed to by `departed`. If `status` is not `NULL` then it points to the exit status value of the terminated thread.

The following code illustrates the used of `thr_create()` and `thr_join()`. It creates a single thread which prints out the numbers 0 to 10 and then exits.

```
/* thr1.c -- demonstrates thread creation and joining */

#include <thread.h>
#include <unistd.h>

void start_routine(int p)
{
    int i;

    for (i = 0; i < p; ++i)
        printf("i = %d\n", i);
}

main(void)
{
    thread_t tid;

    thr_create(NULL, 0, (void *) start_routine, (void *) 10, 0, &tid);
    thr_join(tid, NULL, NULL);
}
```

4.2.3 Thread Termination

The `thr_exit()` library routine is used to terminate a thread. It takes a single argument, which specifies the exit status for the thread. The exist status of a terminated thread can be determined by the threads creator with the `thr_join()` routine. The synopsis for `thr_exit()` is provided below:

```
#include <thread.h>

void thr_exit(void *status);
```

4.2.4 Thread Concurrency

The concurrency level (the number of threads that can be executed concurrently) for a process can be retreived and set with `thr_getconcurrency()` and `thr_setconcurrency()` respectively. `thr_setconcurrency()` is used to add LWPs to the pool of available LWPs for the process. If a thread is created with the `THR_NEW_LWP` flag set then the concurrency level is automatically incremented by 1. The synopsis for these routines is given below:

```
#include <thread.h>

int thr_setconcurrency(int new_level);
int thr_getconcurrency(void );
```

4.2.5 Suspending and Resuming Threads

The `thr_suspend()` and `thr_continue()` library routines are used to suspend and continue thread execution respectively. If the thread is created with the `THR_SUSPENDED` flag, then it can be made to start execution with `thr_continue()`. The synopsis for these routines are provided below:

```
#include <thread.h>

int thr_suspend(thread_t tid);
```

```
int thr_continue(thread_t tid);
```

4.2.6 Thread Priorities

Thread priorities can be retreived and set by `thr_getprio()` and `thr_setprio()` respectively. Unlike standard UNIX process priorities, thread priorities are *fixed*. Thread priorities range from 0 (*lowest* priority) to MAXINT (*highest* priority). The synopsis for these routines is provided below:

```
#include <thread.h>

int thr_setprio(thread_t target_thread, int pri);
int thr_getprio(thread_t target_thread, int *pri);
```

`thr_getprio()` stores the priority of `target_thread` in the location pointed to by `pri`. `thr_setprio()` sets the priority of `target_thread` to `pri`. `target_thread` will preempt lower priority threads, and will yield to higher priority threads.

4.2.7 Thread Synchronisation

The threads library supports four different synchronisation primitives. they are:

- Mutual exclusion locks (*mutex*)
- Read-write locks (*rw*)
- Conditional variables (*cond*)
- Semaphores (*sema*)

We will only discuss mutual exclusion locks here. Refer to section 3T of the UNIX on-linw manual pages for the details of the other synchronisation facilities provided by the threads library.

Mutual exclusion, or *mutex* locks, allow only one thread to access a resource at any time. There are five library routines for manipulating mutex locks. The synopsis for these routines is given below:

```
#include <synch.h>

int mutex_init(mutex_t *mp, int type, void * arg);
int mutex_destroy(mutex_t *mp);
int mutex_lock(mutex_t *mp);
int mutex_trylock(mutex_t *mp);
int mutex_unlock(mutex_t *mp);
```

`mutex_init()` initialises the mutex pointed to by `mp`. `type` may be one of the following constants defined in `synch.h`:

`USYNC_PROCESS` The mutex can be used to synchronise threads across process boundaries.

`USYNC_THREAD` The mutex can only be used by the threads belonging to the process which created it.

`mutex_lock()` blocks until no other thread is executing the critical region. It then locks the region and unblocks. `mutex_trylock()` tries to lock the mutex, but returns immediately if the lock has already been set. `mutex_unlock()` unlocks a locked mutex. `mutex_destroy()` removes the mutex resource from the system.

The code below illustrates the use of mutex locks, `thr_getconcurrency()`, and the `THR_NEW_LWP` flag of `thr_create()`. Four threads are created, each of which increment the global variable `data` eight times. Since `data` must be adjusted in a critical region, a mutex lock is used to ensure exclusion.

```
/* thr2.c -- demonstrates mutexs and thread concurrency */

#include <thread.h>
#include <synch.h>
#include <unistd.h>

int data = 0;       /* shared data */
mutex_t mp;         /* mutual exclusion var */

void routine(void)
{
    int i;

    for (i = 0; i < 8; ++i)
    {
        mutex_lock(&mp);
        data = data + 1;
        mutex_unlock(&mp);
    }
}

main(void)
{
    thread_t tid[4];
    int i;

    printf("current concurrency level = %d\n", thr_getconcurrency());
    mutex_init(&mp, USYNC_THREAD, NULL);

    for (i = 0; i < 4; ++i)
        thr_create(NULL, 0, (void *) routine, NULL, THR_NEW_LWP, &tid[i]);
    printf("new concurrency level = %d\n", thr_getconcurrency());

    for (i = 0; i < 4; ++i)
        thr_join(tid[i], NULL, NULL);

    mutex_destroy(&mp);
    printf("data = %d\n", data);
}
```

4.3 Exercises

1. Write some code to experiment with the threads library semaphores. Try implementing a threaded version of the "Hello world" program that uses semaphores for synchronisation. The threads library routines which you will need to use are `sema_init()`, `sema_destroy()`, `sema_wait()` and `sema_post()`.

2. Examine the `islandfind.c` program which is available from the lecturer. It is a program that finds the largest 4-connected region of 1's in a binary matrix. The algorithm, although implemented serially, is inherently parallel in nature. Modify the code so that the sweeps through the matrix

are done in parallel by threads. You will need to use mutex locks or semaphores to synchronise the threads execution.

Chapter 5

Realtime Processing

5.1 Introduction

Solaris provides support for real time processing through *scheduling classes*. There are three types of class, listed here in order of priority of execution:

1. `RT` – Real time class.

2. `sys` – System class.

3. `TS` – Time sharing class.

By default, user programs operate in the `TS` scheduling class. Kernel routines execute in the `sys` scheduling class. Processes inherit their scheduling class from their creators. Scheduling classes can be specified at the user, group or process levels.

It is *not* possible change the scheduling class for a particular thread. The scheduling class for a thread depends entirely on the scheduling class of the process that the thread belongs to.

Since scheduling classes are inherited, and user shells are by default of scheduling class TS, it is necessary to make a system call to convert a user process to the RT scheduling class.

5.1.1 Changing Scheduling Classes

The `priocntl()` system call is used for changing a processes scheduling class, priority and time quantum. These values can also be queried with `priocntl()`. The synopsis for `priocntl()` is as follows:

```
#include <sys/types.h>
#include <sys/procset.h>

#include <sys/priocntl.h>
#include <sys/rtpriocntl.h>
#include <sys/tspriocntl.h>

long priocntl(idtype_t idtype, id_t id, int cmd, /* cmd_struct arg */);
```

`id` is the PID, UID, GID, LWPID or SID of the process(es) to be affected. `idtype` is one of the following, which determines what the value of `id` refers to:

P_PID	Process ID of a single process.
P_PPID	Parent process ID.
P_LWP	LWP ID.
P_PGID	Process group ID.
P_SID	Session ID.
P_CID	Class ID.
P_UID	Effective user ID.
P_GID	Effective group ID
P_ALL	All processes.

The `cmd` argument specifies the operation that `priocntl()` is to perform. The value of `cmd` determines the type of structure that `arg` points to. `cmd` may be one of the following constants defined in `priocntl.h`:

`cmd` argument	`arg` type	function
PC_GETCID	pcinfo_t	get class ID and attributes
PC_GETCLINFO	pcinfo_t	get class name and attributes
PC_SETPARMS	pcparms_t	set class and scheduling parameters
PC_GETPARMS	pcparms_t	get class and scheduling parameters

If successful, `priocntl()` returns the following values:

- `PC_GETCID` and `PC_GETCLINFO` commands return the number of scheduling classes.
- `PC_SETPARMS` returns 0.
- `PC_GETPARMS` returns the PID of the process being queried.

The `priocntl()` routine retuns -1 if an error occurs.

The `PC_GETCID` and `PC_GETCLINFO` commands use the `pcinfo` structure (which is pointed to by the `arg` parameter to `priocntl()`) to send and receive values:

```
typedef struct pcinfo {
    id_t pc_cid;                    /* class ID */
    char pc_clname[PC_CLNMSZ];      /* class name */
    long pc_clinfo[PC_CLINFOSZ];    /* class information */
} pcinfo_t;
```

For the realtime class, `pc_clinfo` contains an `rtinfo` structure which holds the maximum valid realtime priority:

```
typedef struct rtinfo {
     short rt_maxpri;               /* maximum realtime priority */
} rtinfo_t;
```

For the timesharing class, `pc_clinfo` contains an `tsinfo` structure whic holds the maximum end-user timesharing priority:

```
typedef struct tsinfo {
     short rt_maxupri;               /* limits of user priority range */
} tsinfo_t;
```

The `PC_GETPARMS` and `PC_SETPARMS` commands use the `pcparms` structure (which is pointed to by the `arg` parameter to `priocntl()`) to send and receive values:

```
typedef struct pcparms {
    id_t pc_cid;                    /* process class */
    long pc_clparms[PC_CLPARMSZ];   /* class specific info */
} pcparms_t;
```

For the realtime class, `pc_clparms` contains an `rtparms` structure, defined below:

```
typedef struct rtparms {
    short rt_pri;                   /* reatime priority */
    ulong rt_tqsecs;                /* seconds in time quantum */
    long  rt_tqnsecs;               /* additional nsecs in quantum */
} rtparms_t;
```

For the timesharing class, `pc_clparms` contains an `tsparms` structure w holds the scheduler parameters specific to timesharing:

```
typedef struct tsparms {
    short ts_uprilim;               /* user priority limit */
    short ts_upri;                  /* user priority */
} tsparms_t;
```

The code below illustrates how to change a processes scheduling call to realtime, and set the priority of that process to the maximum allowable priority minus one. The executable takes a single command line argument which is the PID of the process to be RT scheduled.

```
/* realtime.c -- change a processes scheduling class to RT */

#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <sys/priocntl.h>
#include <sys/rtpriocntl.h>
#include <sys/tspriocntl.h>

id_t schedinfo(char *, short *);

int main(int argc, char *argv[])
{
    pcparms_t    pcparms;
    rtparms_t    *rtparmsp;
    id_t         pid, rtID;
    short        maxrtpri;

    if ((pid = atoi(argv[1])) <= 0)
    {
        perror("bad pid");
        exit(1);
    }

    /* determine max priority and RT class ID */

    if ((rtID = schedinfo("RT", &maxrtpri)) == -1)
    {
        perror("schedinfo failed for RT");
```

```
        exit(2);
    }

    /* change PID to RT class scheduling, and set priority to maxrtpri - 1 */

    pcparms.pc_cid = rtID;
    rtparmsp = (struct rtparms *) pcparms.pc_clparms;
    rtparmsp->rt_pri = maxrtpri - 1;
    rtparmsp->rt_tqnsecs = RT_TQDEF;

    if (priocntl(P_PID, pid, PC_SETPARMS, &pcparms) == -1)
    {
        perror("PC_SETPARMS failed");
        exit(3);
    }
}

/* schedinfo() -- returns class ID and maximum priority */

id_t schedinfo(char *name, short *maxpri)
{
    pcinfo_t   info;
    tsinfo_t   *tsinfop;
    rtinfo_t   *rtinfop;

    (void) strcpy(info.pc_clname, name);
    if (priocntl(0L, 0L, PC_GETCID, &info) == -1L)
        return -1;
    if (strcmp(name, "TS") == 0)
    {
        tsinfop = (struct tsinfo *) info.pc_clinfo;
        *maxpri = tsinfop->ts_maxupri;
    }
    else if (strcmp(name, "RT") == 0)
    {
        rtinfop = (struct rtinfo *) info.pc_clinfo;
        *maxpri = rtinfop->rt_maxpri;
    }
    else
        return -1;
    return info.pc_cid;
}
```

On yallara, the machine we are using for the course, the RT scheduling class has been disabled. This can be verified with the `priocntl -l` shell command.

5.1.2 Locking Memory

It is often desirable for the memory associated with a realtime process to be *locked*. Locked memory can not be paged or swapped out of physical memory to disk. Locking the memory of a RT class process guarantees a minimum process dispatch latency because all of the processes address space is located in physical memory.

Under Solaris, there is a system-wide limit on how many pages can be locked simultaneously. The limit is determined during the system boot sequence.

There are three system calls that are used for locking memory. `mlock()` requests that one segment of memory be locked. `munlock()` reverses the action of `mlock()`. `mlockall()` allows all of the address mappings of a super-user process to be locked at once. Only processes with super-user priviledges have permission to lock memory. The locks remain in place until they are specifically unlocked or the process terminates.

5.1.3 High Performance I/O

The standard `read()` and `write()` systems calls are *synchronous* operations, at least as far as the process using them is concerned. `read()` and `write()` do not return to the process until their tasks have been completed.

It is not desirable for realtime processes to perform I/O synchronously. Solaris supports *asynchronous* I/O operations via the `aioread()`, `aiowrite()`, `aiocancel()` and `aiowait()` system calls. These routines place the I/O requests on a queue and return immediately. The kernel is then responsible for processing the enqueued I/O requests in a timely fashion. Notification of the completion of asynchronous I/O operations are made to the process via a `SIGIO` signal being generated. Refer to the UNIX on-line manual pages for the calls listed above for more details.

5.1.4 Timers

Often we need a process to execute specific code at regular time intervals when supporting realtime processes. The `getitimer()` and `setitimer()` system calls allow up to four different interval timer types to be created. Solaris supports a timer resolution of 10 milliseconds. Whenever a timer expires a signal is generated to notify the process.

The four different timer types are described below:

`ITIMER_REAL` Decrements the timer in real time. Generates a `SIGALRM` signal.

`ITIMER_VIRTUAL` Decrements the timer only when the process is executing. Generates a `SIGVTALRM` signal.

`ITIMER_PROF` Decrements the time both in process virtual time and when the system is running on behalf of the process. Generates a `SIGPROF` signal.

`ITIMER_REALPROF` Decrements in real time. This is designed for profiling multithreaded programs. Does not generate a signal.

5.2 Exercise

1. Write a clock program that makes use of `getitimer()`, `setitimer()` and `gettimeofday()`.