File Access and Directory System Calls

There are many UNIX utilities that allow us to manipulate directories and files. `cd, ls, rm, cp, mkdir` ***etc.*** are examples we have (hopefully) already met.

We will now see how to achieve similar tasks from within a C program.

Directory handling functions: `<unistd.h>`

This basically involves calling appropriate functions to traverse a directory hierarchy or inquire about a directories contents.

`int chdir(char *path)` -- changes directory to specified path string.

Example: C emulation of UNIX's `cd` command:

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

main(int argc,char **argv)
   {
                              if (argc < 2)
                                            {  printf(``Usage: %s
                                                                                    >\
                                                                                     <pathname    n'',argv[0]);
                                                                                    exit(1);
                                             }
                             if (chdir(argv[1]) != 0)
                                                                                          \
                                            { printf(``Error in chdir  n'');
                                                                                    exit(1);
                                            }
              }
```

`char *getwd(char *path) -- get the full pathname of the current working directory. path is a pointer to a string where the pathname will be returned. getwd returns a pointer to the string or NULL if an error occurs.`

Scanning and Sorting Directories: `<sys/types.h>,<sys/dir.h>`

Two useful functions (On BSD platforms and **NOT** in multi-threaded application) are available

`scandir(char *dirname, struct direct **namelist, int (*select)(),`
`int (*compar)())` -- reads the directory dirname and builds an array of pointers to directory entries or -1 for an error. `namelist` is a pointer to an array of structure pointers.

`(*select))()` is a pointer to a function which is called with a pointer to a directory entry (defined in <sys/types> and should return a non zero value if the directory entry should be included in the array. If this pointer is NULL, then all the directory entries will be included.

The last argument is a pointer to a routine which is passed to `qsort` (see `man qsort`) -- a built in function which sorts the completed array. If this pointer is NULL, the array is not sorted.

`alphasort(struct direct **d1, **d2)` -- alphasort() is a built in routine which will sort the array alphabetically.

Example - a simple C version of UNIX `ls` utility

```
#include <sys/types.h>
#include <sys/dir.h>
#include <sys/param.h>
#include <stdio.h>

#define FALSE 0
#define TRUE !FALSE

extern  int alphasort();

char pathname[MAXPATHLEN];

main()   { int count,i;
                              struct direct **files;
                              int file_select();

                              if (getwd(pathname) == NULL )
                                                                                    \
                                            { printf("Error getting path  n");
                                                                          exit(0);
                                            }
                                                                                      \
                              printf("Current Working Directory = %s  n",pathname);
```

```
                        count =
                          scandir(pathname, &files, file_select, alphasort);

                        /* If no files found, make a non-selectable menu item */
                        if              (count <= 0)

                                        {               printf(``No files in this directory  n'');
                                                       exit(0);
                                        }

                        printf(``Number of files = %d  n'',count);
                        for (i=1;i<count+1;++i)
                                         printf(``%s  '',files[i-1]->d_name);

                        printf(``  n''); /* flush buffer */
                }


int file_select(struct direct   *entry)

                {if ((strcmp(entry->d_name, ``.'') == 0) ||
                                        (strcmp(entry->d_name, ``..'') == 0))
                                         return (FALSE);
                        else
                                                return (TRUE);
                }
```

scandir returns the current directory (.) and the directory above this (..) as well as all files so we need to check for these and return FALSE so that they are not included in our list.

Note: scandir and alphasort have definitions in sys/types.h and sys/dir.h.
MAXPATHLEN and getwd definitions in sys/param.h

We can go further than this and search for specific files: Let's write a modified file_select() that only scans for files with a .c, .o or .h suffix:

```
int file_select(struct direct   *entry)

                {char *ptr;
                         char *rindex(char *s, char c);

                        if ((strcmp(entry->d_name, ``.'')== 0) ||
                                        (strcmp(entry->d_name, ``..'') == 0))
                                         return (FALSE);

                         /* Check for filename extensions */
                         ptr = rindex(entry->d_name, '.')
                         if ((ptr != NULL) &&
                                         ((strcmp(ptr, ``.c'') == 0)
                                         ||
                                            (strcmp(ptr, ``.h'') == 0)
                                         ||
                                            (strcmp(ptr, ``.o'') == 0) ))
                                                        return (TRUE);
                         else
                                         return(FALSE);
                }
```

```
NOTE: rindex() is a string handling function that returns a pointer to the last occurrence of character c
in string s, or a NULL pointer if c does not occur in the string. (index() is similar function but assigns
a pointer to 1st occurrence.)

The function struct direct *readdir(char *dir) also exists in <sys/dir.h>> to return a given directory dir
listing.
```

File Manipulation Routines: unistd.h, sys/types.h, sys/stat.h

There are many system calls that can applied directly to files stored in a directory.

File Access

`int access(char *path, int mode)` -- determine accessibility of file.

`path` points to a path name naming a file. `access()` checks the named file for accessibility according to `mode`, defined in `#include <unistd.h>`:

R_OK
- test for read permission

W_OK
- test for write permission

X_OK

- test for execute or search permission

F_OK

- test whether the directories leading to the file can be searched and the file exists.

`access()` returns: 0 on success, -1 on failure and sets `errno` to indicate the error. See `man` pages for list of errors.

errno

`errno` is a special system variable that is set if a system call cannot perform its set task.

To use `errno` in a C program it must be declared via:

```
extern int errno;
```

It can be manually reset within a C program other wise it simply retains its last value.

`int chmod(char *path, int mode)` change the mode of access of a file. specified by `path` to the given `mode`.

`chmod()` returns 0 on success, -1 on failure and sets `errno` to indicate the error. Errors are defined in `#include <sys/stat.h>`

The access mode of a file can be set using predefined macros in `sys/stat.h` -- see `man` pages -- or by setting the mode in a a 3 digit octal number.

The rightmost digit specifies owner privileges, middle group privileges and the leftmost other users privileges.

For each octal digit think of it a 3 bit binary number. Leftmost bit = read access (on/off) middle is write, right is executable.

So 4 (octal 100) = read only, 2 (010) = write, 6 (110) = read and write, 1 (001) = execute.

so for access mode 600 gives user read and write access others no access. 666 gives everybody read/write access.

NOTE: a UNIX command `chmod` also exists

File Status

Two useful functions exist to inquire about the files current status. You can find out how large the file is (`st_size`) when it was created (`st_ctime`) ***etc.*** (see `stat` structure definition below. The two functions are prototyped in `<sys/stat.h>`

```
int stat(char *path, struct stat *buf),
int fstat(int fd, struct
stat *buf)
```

`stat()` obtains information about the file named by path. Read, write or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable.

`fstat()` obtains the same information about an open file referenced by the argument descriptor, such as would be obtained by an `open` call (Low level I/O).

`stat(), and fstat()` return 0 on success, -1 on failure and sets `errno` to indicate the error. Errors are again defined in `#include <sys/stat.h>`

`buf` is a pointer to a stat structure into which information is placed concerning the file. A stat structure is define in `#include <sys/types.h>`, as follows

```
struct stat {
          mode_t   st_mode;     /* File mode (type, perms) */
          ino_t    st_ino;      /* Inode number */
          dev_t    st_dev;      /* ID of device containing */
                                /* a directory entry for this file */
          dev_t    st_rdev;     /* ID of device */
                                /* This entry is defined only for */
                                /* char special or block special files */
          nlink_t  st_nlink;    /* Number of links */
          uid_t    st_uid;      /* User ID of the file's owner */
          gid_t    st_gid;      /* Group ID of the file's group */
          off_t    st_size;     /* File size in bytes */
          time_t   st_atime;    /* Time of last access */
          time_t   st_mtime;    /* Time of last data modification */
          time_t   st_ctime;    /* Time of last file status change */
                                /* Times measured in seconds since */
                                /* 00:00:00 UTC, Jan. 1, 1970 */
          long     st_blksize;  /* Preferred I/O block size */
          blkcnt_t st_blocks;   /* Number of 512 byte blocks allocated*/
}
```

File Manipulation:stdio.h, unistd.h

There are few functions that exist to delete and rename files. Probably the most common way is to use the `stdio.h` functions:

```
int remove(const char *path);
int rename(const char *old, const char *new);
```

Two system calls (defined in `unistd.h`) which are actually used by `remove()` and `rename()` also exist but are probably harder to remember unless you are familiar with UNIX.

`int unlink(cons char *path)` -- removes the directory entry named by `path`

`unlink()` returns 0 on success, -1 on failure and sets `errno` to indicate the error. Errors listed in `#include <sys/stat.h>`

A similar function `link(const char *path1, const char *path2)` creates a linking from an existing directory entry `path1` to a new entry `path2`

Creating Temporary FIles:<stdio.h>

Programs often need to create files just for the life of the program. Two convenient functions (plus some variants) exist to assist in this task. Management (deletion of files etc) is taken care of by the Operating System.

The function `FILE *tmpfile(void)` creates a temporary file and opens a corresponding stream. The file will automatically be deleted when all references to the file are closed.

The function `char *tmpnam(char *s)` generate file names that can safely be used for a temporary file. Variant functions `char *tmpnam_r(char *s)` and `char *tempnam(const char *dir, const char *pfx)` also exist

NOTE: There are a few more file manipulation routines not listed here see `man` pages.

Exercises

Exercise 12675

Write a C program to emulate the `ls -l` UNIX command that prints all files in a current directory and lists access privileges etc. DO NOT simply `exec ls -l` from the program.

Exercise 12676

Write a program to print the lines of a file which contain a word given as the program argument (a simple version of `grep` UNIX utility).

Exercise 12677

Write a program to list the files given as arguments, stopping every 20 lines until a key is hit.(a simple version of `more` UNIX utility)

Exercise 12678

Write a program that will list all files in a current directory and all files in subsequent sub directories.

Exercise 12679

Write a program that will only list subdirectories in alphabetical order.

Exercise 12680

Write a program that shows the user all his/her C source programs and then prompts interactively as to whether others should be granted read permission; if affirmative such permission should be granted.

Exercise 12681

Write a program that gives the user the opportunity to remove any or all of the files in a current working directory. The name of the file should appear followed by a prompt as to whether it should be removed.

Time Functions

In this chapter we will look at how we can access the clock time with UNIX system calls.

There are many more time functions than we consider here - see `man` pages and standard library function listings for full details. In this chapter we concentrate on applications of timing functions in C

Uses of time functions include:

- telling the time.
- timing programs and functions.
- setting number seeds.

Basic time functions

Some of thge basic time functions are prototypes as follows:

`time_t time(time_t *tloc)` -- returns the time since `00:00:00 GMT, Jan. 1, 1970`, measured in seconds.

If `tloc` is not NULL, the return value is also stored in the location to which tloc points.

`time()` returns the value of time on success.

On failure, it returns `(time_t) -1`. `time_t` is typedefed to a long (int) in <sys/types.h> and <sys/time.h> header files.

`int ftime(struct timeb *tp)` -- fills in a structure pointed to by `tp`, as defined in <sys/timeb.h>:

```
   struct timeb
                  { time_t time;
                                          unsigned short millitm;
                                          short timezone;
                                          short dstflag;
                                        };
```

The structure contains the time since the epoch in seconds, up to 1000 milliseconds of more precise interval, the local time zone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that Day light Saving time applies locally during the appropriate part of the year.

On success, `ftime() returns no useful value. On failure, it returns -1.`

```
Two other functions defined ***etc.*** in #include <time.h>

char *ctime(time_t *clock),
char *asctime(struct tm *tm)

ctime() converts a long integer, pointed to by clock, to a 26-character string of the form produced by
asctime(). It first breaks down clock to a tm structure by calling localtime(), and then calls asctime() to
convert that tm structure to a string.

asctime() converts a time value contained in a tm structure to a 26-character string of the form:

   Sun Sep 16 01:03:52 1973

asctime() returns a pointer to the string.
```

Example time applications

we mentioned above three possible uses of time functions (there are many more) but these are very common.

Example 1: Time (in seconds) to perform some computation

This is a simple program that illustrates that calling the time function at distinct moments and noting the different times is a simple method of timing fragments of code:

```
/* timer.c */

#include <stdio.h>
#include <sys/types.h>
#include <time.h>

main()
  {  int i;
                              time_t t1,t2;

                              (void) time(&t1);
                              for (i=1;i<=300;++i)
                                \

                                 printf(``%d %d %d  n'',i, i*i, i*i*i);
                                (void) time(&t2);
                                  \

                                printf(``  n Time to do 300 squares and
                                    \

                              cubes= %d seconds  n'', (int) t2-t1);
                }
```

Example 2: Set a random number seed

We have seen a similar example previously, this time we use the `lrand48()` function to generate of number sequence:

```
/* random.c */
#include <stdio.h>
#include <sys/types.h>
#include <time.h>

main()
  { int i;
                              time_t t1;

                              (void) time(&t1);
                              srand48((long) t1);
                              /* use time in seconds to set seed */
                              printf(``5 random numbers
                                                  \
                                (Seed = %d):  n'',(int) t1);
                              for (i=0;i<5;++i)
                                printf(``%d '', lrand48());
                                        \   \
                              printf(``  n  n''); /* flush print buffer */
                }
```

```
lrand48() returns non-negative long integers uniformly distributed over the interval (0, 2**31).

A similar function drand48() returns double precision numbers in the range [0.0,1.0).

srand48() sets the seed for these random number generators. It is important to have different seeds when we
call the functions otherwise the same set of pseudo-random numbers will generated. time() always provides a
unique seed.
```

Exercises

Exercise 12708

Write a C program that times a fragment of code in milliseconds.

Exercise 12709

Write a C program to produce a series of floating point random numbers in the ranges (a) 0.0 - 1.0 (b) 0.0 - n where n is any floating point value. The seed should be set so that a unique sequence is guaranteed.

Process Control: `<stdlib.h>,<unistd.h>`

A ***process*** is basically a single running program. It may be a ``system'' program (***e.g*** login, update, csh) or program initiated by the user (textedit, dbxtool or a user written one).

When UNIX runs a process it gives each process a unique number - a process ID, `pid`.

The UNIX command `ps` will list all current processes running on your machine and will list the pid.

The C function `int getpid()` will return the pid of process that called this function.

A program usually runs as a single process. However later we will see how we can make programs run as several separate communicating processes.

Running UNIX Commands from C

We can run commands from a C program just as if they were from the UNIX command line by using the `system()` function. **NOTE:** this can save us a lot of time and hassle as we can run other (proven) programs, scripts ***etc.*** to do set tasks.

`int system(char *string)` -- where string can be the name of a unix utility, an executable shell script or a user program. System returns the exit status of the shell. System is prototyped in `<stdlib.h>`

Example: Call `ls` from a program

```
main()
                                        \
{ printf(``Files in Directory are: n'');
                system(``ls -l'');
}
```

```
system is a call that is made up of 3 other system calls: execl(), wait() and fork() (which are prototyed
in <unistd>)
```

execl()

`execl` has 5 other related functions -- see `man` pages.

`execl` stands for ***execute*** and ***leave*** which means that a process will get executed and then terminated by `execl`.

It is defined by:

```
execl(char *path, char *arg0,...,char *argn, 0);
```

The last parameter must always be 0. It is a ***NULL terminator***. Since the argument list is variable we must have some way of telling C when it is to end. The NULL terminator does this job.

where `path` points to the name of a file holding a command that is to be executed, `argo` points to a string that is the same as path (or at least its last component.

`arg1 ... argn` are pointers to arguments for the command and 0 simply marks the end of the (variable) list of arguments.

So our above example could look like this also:

```
main()
                                      \
{ printf(``Files in Directory are: n'');
                execl(`/bin/ls'',``ls'', ``-l'',0);
}
```

fork()

`int fork()` turns a single process into 2 identical processes, known as the ***parent*** and the ***child***. On success, fork() returns 0 to the child process and returns the process ID of the child process to the parent process. On failure, fork() returns -1 to the parent process, sets errno to indicate the error, and no child process is created.

NOTE: The child process will have its own unique PID.

The following program illustrates a simple use of fork, where two copies are made and run together (multitasking)

```
main()
{ int return_value;
                                                \
                printf(``Forking process  n'');
                fork();
                printf(``The process id is %d
                                          \
                  and return value is %d  n",
                  getpid(), return_value);
                execl(``/bin/ls/'',``ls'',``-l'',0);
                                                  \
                printf(``This line is not printed  n'');
}
```

The Output of this would be:

```
Forking process
The process id is 6753 and return value is 0
The process id is 6754 and return value is 0
two lists of files in current directory
```

NOTE: The processes have unique ID's which will be different at each run.

It also impossible to tell in advance which process will get to CPU's time -- so one run may differ from the next.

When we spawn 2 processes we can easily detect (in each process) whether it is the child or parent since fork returns 0 to the child. We can trap any errors if fork returns a -1. ***i.e.***:

```
int pid; /* process identifier */

pid = fork();
if ( pid < 0 )
                                    \
                { printf(``Cannot fork!!  n'');
                              exit(1);
                }
if ( pid == 0 )
                { /* Child process */ ..... }
else
                { /* Parent process pid is child's pid */
                .... }
```

wait()

`int wait (int *status_location)` -- will force a parent process to wait for a child process to stop or terminate. `wait()` return the pid of the child or -1 for an error. The exit status of the child is returned to `status_location`.

exit()

`void exit(int status)` -- terminates the process which calls this function and returns the exit `status` value. Both UNIX and C (forked) programs can read the status value.

By convention, a status of 0 means ***normal termination*** any other value indicates an error or unusual occurrence. Many standard library calls have errors defined in the `sys/stat.h` header file. We can easily derive our own conventions.

A complete example of forking program is originally titled `fork.c`:

```
/* fork.c - example of a fork in a program */
/* The program asks for UNIX commands to be typed and inputted to a string*/
/* The string is then "parsed" by locating blanks etc. */
/* Each command and sorresponding arguments are put in a args array */
/* execvp is called to execute these commands in child process */
/* spawned by fork() */

/* cc -o fork fork.c */

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

main()
{
    char buf[1024];
    char *args[64];

    for (;;) {
        /*
         * Prompt for and read a command.
         */
        printf("Command: ");

        if (gets(buf) == NULL) {
            printf("\n");
            exit(0);
        }

        /*
         * Split the string into arguments.
         */
        parse(buf, args);

        /*
         * Execute the command.
         */
        execute(args);
    }
}

/*
 * parse--split the command in buf into
 *         individual arguments.
 */
parse(buf, args)
char *buf;
char **args;
{
    while (*buf != NULL) {
        /*
         * Strip whitespace.  Use nulls, so
         * that the previous argument is terminated
         * automatically.
```

```
         */
        while ((*buf == ' ') || (*buf == '\t'))
            *buf++ = NULL;

        /*
         * Save the argument.
         */
        *args++ = buf;

        /*
         * Skip over the argument.
         */
        while ((*buf != NULL) && (*buf != ' ') && (*buf != '\t'))
            buf++;
    }

    *args = NULL;
}

/*
 * execute--spawn a child process and execute
 *           the program.
 */
execute(args)
char **args;
{
    int pid, status;

    /*
     * Get a child process.
     */
    if ((pid = fork()) < 0) {
        perror("fork");
        exit(1);

       /* NOTE: perror() produces a short  error  message  on  the  standard
           error describing the last error encountered during a call to
           a system or library function.
       */
    }

    /*
     * The child executes the code inside the if.
     */
    if (pid == 0) {
        execvp(*args, args);
        perror(*args);
        exit(1);

       /* NOTE: The execv() vnd execvp versions of execl() are useful when the
          number  of  arguments is unknown in advance;
          The arguments to execv() and execvp()  are the name
          of the file to be executed and a vector of strings  contain-
          ing  the  arguments.   The last argument string must be fol-
          lowed by a 0 pointer.

          execlp() and execvp() are called with the same arguments  as
          execl()  and  execv(),  but duplicate the shell's actions in
          searching for an executable file in a list  of  directories.
          The directory list is obtained from the environment.
        */
    }

    /*
     * The parent executes the wait.
     */
    while (wait(&status) != pid)
        /* empty */ ;
}
```

Exerises

Exercise 12727

Use `popen()` to pipe the `rwho` (UNIX command) output into `more` (UNIX command) in a C program.

Interprocess Communication (IPC), Pipes

We have now began to see how multiple processes may be running on a machine and maybe be controlled (spawned by `fork()` by one of our programs.

In numerous applications there is clearly a need for these processes to communicate with each exchanging data or control information. There are a few methods which can accomplish this task. We will consider:

- Pipes
- Signals
- Message Queues
- Semaphores
- Shared Memory
- Sockets

In this chapter, we will study the piping of two processes. We will study the others in turn in subsequent chapters.

Piping in a C program: <`stdio.h`>

Piping is a process where the input of one process is made the input of another. We have seen examples of this from the UNIX command line using `|`.

We will now see how we do this from C programs.

We will have two (or more) `forked` processes and will communicate between them.

We must first open a ***pipe***

UNIX allows two ways of opening a pipe.

`popen()` -- Formatted Piping

`FILE *popen(char *command, char *type)` -- opens a pipe for I/O where the command is the process that will be connected to the calling process thus creating the ***pipe***. The type is either ``r'' - for reading, or ``w'' for writing.

`popen()` returns is a stream pointer or NULL for any errors.

A pipe opened by `popen()` should always be closed by `pclose(FILE *stream)`.

We use `fprintf()` and `fscanf()` to communicate with the pipe's `stream`.

`pipe()` -- Low level Piping

`int pipe(int fd[2])` -- creates a pipe and returns two file descriptors, `fd[0], fd[1]`. `fd[0]` is opened for reading, `fd[1]` for writing.

`pipe()` returns 0 on success, -1 on failure and sets `errno` accordingly.

The standard programming model is that after the pipe has been set up, two (or more) cooperative processes will be created by a fork and data will be passed using `read()` and `write()`.

Pipes opened with `pipe()` should be closed with `close(int fd)`.

Example: Parent writes to a child

```
int pdes[2];

pipe(pdes);
if ( fork() == 0 )
  { /* child */
                              close(pdes[1]);
                              read( pdes[0]); /* read from parent */
                              .....
                }
else
                {             close(pdes[0]);
                              write( pdes[1]); /* write to child */
                              .....
                }
```

An futher example of piping in a C program is `plot.c and subroutines and it performs as follows:`

- `The program has two modules plot.c (main) and plotter.c.`
- `The program relies on you having installed the freely` ***gnuplot*** `graph drawing program in the directory /usr/local/bin/ (in the listing below at least) -- this path could easily be changed.`
- `The program plot.c calls` ***gnuplot***
- `Two Data Stream is generated from Plot`
  - $y = sin(x)$
  - $y = sin(1/x)$

- 2 Pipes created -- 1 per Data Stream.
- °***Gnuplot*** produces ``live'' drawing of output.

The code listing for plot.c is:

```
/* plot.c - example of unix pipe. Calls gnuplot graph drawing package to draw
   graphs from within a C program. Info is piped to gnuplot */
/* Creates 2 pipes one will draw graphs of y=0.5 and y = random 0-1.0 */
/* the other graphs of y = sin (1/x) and y = sin x */

/* Also user a plotter.c module */
/* compile: cc -o plot plot.c plotter.c */

#include "externals.h"
#include <signal.h>

#define DEG_TO_RAD(x) (x*180/M_PI)

double drand48();
void quit();

FILE *fp1, *fp2, *fp3, *fp4, *fopen();

main()
{   float i;
    float y1,y2,y3,y4;

    /* open files which will store plot data */
    if ( ((fp1 = fopen("plot11.dat","w")) == NULL) ||
         ((fp2 = fopen("plot12.dat","w")) == NULL) ||
          ((fp3 = fopen("plot21.dat","w")) == NULL) ||
           ((fp4 = fopen("plot22.dat","w")) == NULL) )
            { printf("Error can't open one or more data files\n");
              exit(1);
            }

signal(SIGINT,quit); /* trap ctrl-c call quit fn */
StartPlot();
y1 = 0.5;
srand48(1); /* set seed */
for (i=0;;i+=0.01) /* increment i forever use ctrl-c to quit prog */
  { y2 =  (float) drand48();
    if (i == 0.0)
       y3 = 0.0;
   else
       y3 = sin(DEG_TO_RAD(1.0/i));
    y4 = sin(DEG_TO_RAD(i));

    /* load files */
    fprintf(fp1,"%f %f\n",i,y1);
    fprintf(fp2,"%f %f\n",i,y2);
    fprintf(fp3,"%f %f\n",i,y3);
    fprintf(fp4,"%f %f\n",i,y4);

    /* make sure buffers flushed so that gnuplot */
    /*  reads up to data file */
    fflush(fp1);
    fflush(fp2);
    fflush(fp3);
    fflush(fp4);

    /* plot graph */
    PlotOne();
    usleep(250); /* sleep for short time */
  }
}

void quit()
{  printf("\nctrl-c caught:\n Shutting down pipes\n");
   StopPlot();

   printf("closing data files\n");
   fclose(fp1);
   fclose(fp2);
   fclose(fp3);
   fclose(fp4);

   printf("deleting data files\n");
   RemoveDat();
}
```

The plotter.c module is as follows:

```
/* plotter.c module */
/* contains routines to plot a data file produced by another program  */
/* 2d data plotted in this version                                    */
/**********************************************************************/

#include "externals.h"

static FILE *plot1,
       *plot2,
       *ashell;
```

```
static char *startplot1 = "plot [] [0:1.1]'plot11.dat' with lines,
            'plot12.dat' with lines\n";

static char *startplot2 = "plot 'plot21.dat' with lines,
            'plot22.dat' with lines\n";

static char *replot = "replot\n";
static char *command1= "/usr/local/bin/gnuplot> dump1";
static char *command2= "/usr/local/bin/gnuplot> dump2";
static char *deletefiles = "rm plot11.dat plot12.dat plot21.dat plot22.dat";
static char *set_term = "set terminal x11\n";

void
StartPlot(void)
 { plot1 = popen(command1, "w");
   fprintf(plot1, "%s", set_term);
   fflush(plot1);
   if (plot1 == NULL)
      exit(2);
   plot2 = popen(command2, "w");
   fprintf(plot2, "%s", set_term);
   fflush(plot2);
   if (plot2 == NULL)
      exit(2);
 }

void
RemoveDat(void)
 { ashell = popen(deletefiles, "w");
   exit(0);
 }

void
StopPlot(void)
 { pclose(plot1);
   pclose(plot2);
 }

void
PlotOne(void)
 { fprintf(plot1, "%s", startplot1);
   fflush(plot1);

   fprintf(plot2, "%s", startplot2);
   fflush(plot2);
 }

void
RePlot(void)
 { fprintf(plot1, "%s", replot);
   fflush(plot1);
 }
```

The header file externals.h contains the following:

```
/* externals.h */
#ifndef EXTERNALS
#define EXTERNALS

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* prototypes */
void StartPlot(void);
void RemoveDat(void);
void StopPlot(void);
void PlotOne(void);
void RePlot(void);
#endif
```

Exercises

Exercise 12733

Setup a two-way pipe between parent and child processes in a C program. i.e. both can send and receive signals.

IPC:Interrupts and Signals: <`signal.h`>

In this section will look at ways in which two processes can communicate. When a process terminates abnormally it usually tries to send a signal indicating what went wrong. C programs (and UNIX) can trap these for diagnostics. Also user specified communication can take place in this way.

Signals are software generated interrupts that are sent to a process when a event happens. Signals can be synchronously generated by an error in an application, such as `SIGFPE` and `SIGSEGV`, but most signals are asynchronous. Signals can be posted to a process when the system detects a software event, such as a user entering an interrupt or stop or a kill request from another process. Signals can also be come directly from the OS kernel when a hardware event such as a bus error or an illegal instruction is encountered. The system defines a set of signals that can be posted to a process. Signal delivery is analogous to hardware interrupts in that a signal can be blocked from being delivered in the future. Most signals cause termination of the receiving process if no action is taken by the process in response to the signal. Some signals stop the receiving process and other signals can be ignored. Each signal has a default action which is one of the following:

- The signal is discarded after being received
- The process is terminated after the signal is received
- A core file is written, then the process is terminated
- Stop the process after the signal is received

Each signal defined by the system falls into one of five classes:

- Hardware conditions
- Software conditions
- Input/output notification
- Process control
- Resource control

Macros are defined in `<signal.h>` header file for common signals.

These include:

SIGHUP 1 /* hangup */
SIGINT 2 /* interrupt */
SIGQUIT 3 /* quit */
SIGILL 4 /* illegal instruction */
SIGABRT 6 /* used by abort */
SIGKILL 9 /* hard kill */
SIGALRM 14 /* alarm clock */
SIGCONT 19 /* continue a stopped process */
SIGCHLD 20 /* to parent on child stop or exit */

Signals can be numbered from 0 to 31.

Sending Signals -- `kill(), raise()`

There are two common functions used to send signals

`int kill(int pid, int signal)` - a system call that send a `signal` to a process, `pid`. If pid is greater than zero, the signal is sent to the process whose process ID is equal to pid. If pid is 0, the signal is sent to all processes, except system processes.

`kill()` returns 0 for a successful call, -1 otherwise and sets `errno` accordingly.

`int raise(int sig)` sends the signal sig to the executing program. `raise()` actually uses `kill()` to send the signal to the executing program:

```
kill(getpid(), sig);
```

There is also a UNIX command called kill that can be used to send signals from the command line - see `man` pages.

NOTE: that unless caught or ignored, the `kill` signal terminates the process. Therefore protection is built into the system.

Only processes with certain access privileges can be killed off.

Basic rule: ***only processes that have the same user can send/receive messages***.

The `SIGKILL` signal cannot be caught or ignored and will always terminate a process.

For example`kill(getpid(),SIGINT);` would send the interrupt signal to the id of the calling process.

This would have a similar effect to `exit()` command. Also `ctrl-c` typed from the command sends a `SIGINT` to the process currently being.

`unsigned int alarm(unsigned int seconds)` -- sends the signal `SIGALRM` to the invoking process after seconds seconds.

Signal Handling -- `signal()`

An application program can specify a function called a signal handler to be invoked when a specific signal is received. When a signal handler is invoked on receipt of a signal, it is said to catch the signal. A process can deal with a signal in one of the following ways:

- The process can let the default action happen
- The process can block the signal (some signals cannot be ignored)
- the process can catch the signal with a handler.

Signal handlers usually execute on the current stack of the process. This lets the signal handler return to the point that execution was interrupted in the process. This can be changed on a per-signal basis so that a signal handler executes on a special stack. If a process must resume in a different context than the interrupted one, it must restore the previous context itself

Receiving signals is straighforward with the function:

`int (*signal(int sig, void (*func)()))()` -- that is to say the function `signal()` will call the `func` functions if the process receives a signal `sig`. Signal returns a pointer to function `func` if successful or it returns an error to `errno` and -1 otherwise.

`func()` can have three values:

`SIG_DFL`
-- a pointer to a system default function `SID_DFL()`, which will terminate the process upon receipt of `sig`.

`SIG_IGN`
-- a pointer to system ignore function `SIG_IGN()` which will disregard the `sig` action (UNLESS it is `SIGKILL`).

A function address
-- a user specified function.

`SIG_DFL and SIG_IGN` are defined in `signal.h` (standard library) header file.

Thus to ignore a `ctrl-c` command from the command line. we could do:

```
signal(SIGINT, SIG_IGN);
```

TO reset system so that `SIGINT` causes a termination at any place in our program, we would do:

```
signal(SIGINT, SIG_DFL);
```

So lets write a program to trap a `ctrl-c` but not quit on this signal. We have a function `sigproc()` that is executed when we trap a `ctrl-c`. We will also set another function to quit the program if it traps the `SIGQUIT` signal so we can terminate our program:

```
#include <stdio.h>

void sigproc(void);

void quitproc(void);

main()
{ signal(SIGINT, sigproc);
                signal(SIGQUIT, quitproc);

                printf(``ctrl-c disabled use ctrl-\ to quit \n'');
                for(;;); /* infinite loop */}

void sigproc()
{               signal(SIGINT, sigproc); /*  */
                /* NOTE some versions of UNIX will reset signal to default
                after each call. So for portability reset signal each time */

                printf(``you have pressed ctrl-c \n'');
}

void quitproc()

{               printf(``ctrl-\ pressed to quit \n'');
                exit(0); /* normal exit status */
}
```

`sig_talk.c` -- complete example program

Let us now write a program that communicates between child and parent processes using `kill() and signal()`.

`fork()` creates the child process from the parent. The `pid` can be checked to decide whether it is the child (== 0) or the parent (pid = child process id).

The parent can then send messages to child using the pid and `kill()`.

The child picks up these signals with `signal()` and calls appropriate functions.

An example of communicating process using signals is `sig_talk.c`:

```
/* sig_talk.c --- Example of how 2 processes can talk */
/* to each other using kill() and signal() */
/* We will fork() 2 process and let the parent send a few */
/* signals to it`s child  */

/* cc sig_talk.c -o sig_talk  */

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

void sighup(); /* routines child will call upon sigtrap */
void sigint();
void sigquit();

main()
{ int pid;

  /* get child process */

   if ((pid = fork()) < 0) {
        perror("fork");
        exit(1);
    }

   if (pid == 0)
     { /* child */
       signal(SIGHUP,sighup); /* set function calls */
       signal(SIGINT,sigint);
       signal(SIGQUIT, sigquit);
       for(;;); /* loop for ever */
     }
  else /* parent */
     {  /* pid hold id of child */
       printf("\nPARENT: sending SIGHUP\n\n");
       kill(pid,SIGHUP);
       sleep(3); /* pause for 3 secs */
       printf("\nPARENT: sending SIGINT\n\n");
       kill(pid,SIGINT);
       sleep(3); /* pause for 3 secs */
       printf("\nPARENT: sending SIGQUIT\n\n");
       kill(pid,SIGQUIT);
       sleep(3);
     }
}

void sighup()

{  signal(SIGHUP,sighup); /* reset signal */
   printf("CHILD: I have received a SIGHUP\n");
}

void sigint()

{  signal(SIGINT,sigint); /* reset signal */
   printf("CHILD: I have received a SIGINT\n");
}

void sigquit()
{ printf("My DADDY has Killed me!!!\n");
  exit(0);
}
```

Other signal functions

There are a few other functions defined in `signal.h`:

`int sighold(int sig)` -- adds `sig` to the calling process's signal mask

`int sigrelse(int sig)` -- removes `sig` from the calling process's signal mask

`int sigignore(int sig)` -- sets the disposition of `sig` to `SIG_IGN`

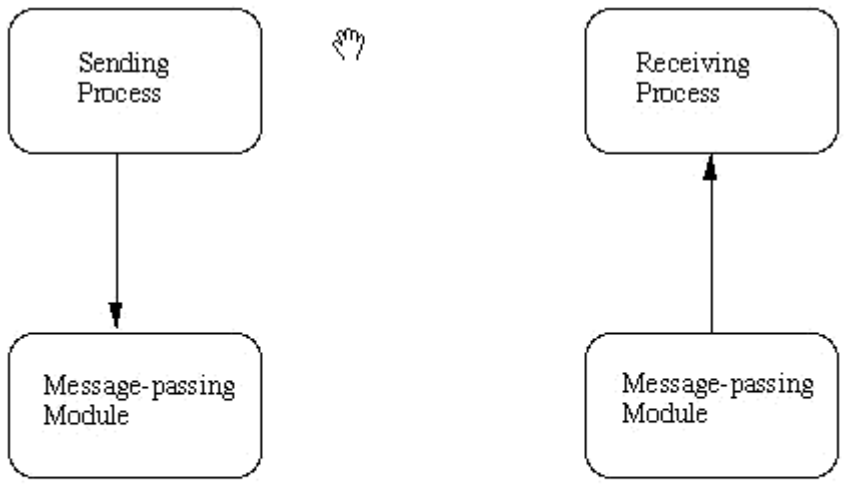
`int sigpause(int sig)` -- removes `sig` from the calling process's signal mask and suspends the calling process until a signal is received

---

IPC:Message Queues:<sys/msg.h>

The basic idea of a ***message queue*** is a simple one.

Two (or more) processes can exchange information via access to a common system message queue. The ***sending*** process places via some (OS) message-passing module a message onto a queue which can be read by another process (Figure 24.1). Each message is given an identification or `type` so that processes can select the appropriate message. Process must share a common `key` in order to gain access to the queue in the first place (subject to other permissions -- see below).



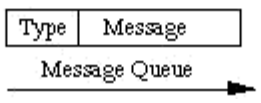


Fig. 24.1 Basic Message Passing IPC messaging lets processes send and receive messages, and queue messages for processing in an arbitrary order. Unlike the file byte-stream data flow of pipes, each IPC message has an explicit length. Messages can be assigned a specific type. Because of this, a server process can direct message traffic between clients on its queue by using the client process PID as the message type. For single-message transactions, multiple server processes can work in parallel on transactions sent to a shared message queue.

Before a process can send or receive a message, the queue must be initialized (through the `msgget` function see below) Operations to send and receive messages are performed by the `msgsnd()` and `msgrcv()` functions, respectively.

When a message is sent, its text is copied to the message queue. The `msgsnd()` and `msgrcv()` functions can be performed as either blocking or non-blocking operations. Non-blocking operations allow for asynchronous message transfer -- the process is not suspended as a result of sending or receiving a message. In blocking or synchronous message passing the sending process cannot continue until the message has been transferred or has even been acknowledged by a receiver. IPC signal and other mechanisms can be employed to implement such transfer. A blocked message operation remains suspended until one of the following three conditions occurs:

- The call succeeds.
- The process receives a signal.
- The queue is removed.

Initialising the Message Queue

The `msgget()` function initializes a new message queue:

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

It can also return the message queue ID (`msqid`) of the queue corresponding to the key argument. The value passed as the `msgflg` argument must be an octal integer with settings for the queue's permissions and control flags.

The following code illustrates the `msgget()` function.

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

...


key_t key; /* key to be passed to msgget() */
int msgflg /* msgflg to be passed to msgget() */
int msqid; /* return value from msgget() */

...
key = ...
msgflg = ...

if ((msqid = msgget(key, msgflg)) == &ndash;1)
  {
    perror("msgget: msgget failed");
    exit(1);
   } else
    (void) fprintf(stderr, &ldquo;msgget succeeded");
```

...

IPC Functions, Key Arguments, and Creation Flags: <sys/ipc.h>

Processes requesting access to an IPC facility must be able to identify it. To do this, functions that initialize or provide access to an IPC facility use a `key_t key` argument. (`key_t` is essentially an `int` type defined in <sys/types.h>

The `key` is an arbitrary value or one that can be derived from a common seed at run time. One way is with `ftok()` , which converts a filename to a key value that is unique within the system. Functions that initialize or get access to messages (also semaphores or shared memory see later) return an ID number of type int. IPC functions that perform read, write, and control operations use this ID. If the key argument is specified as `IPC_PRIVATE`, the call initializes a new instance of an IPC facility that is private to the creating process. When the `IPC_CREAT` flag is supplied in the flags argument appropriate to the call, the function tries to create the facility if it does not exist already. When called with both the `IPC_CREAT` and `IPC_EXCL` flags, the function fails if the facility already exists. This can be useful when more than one process might attempt to initialize the facility. One such case might involve several server processes having access to the same facility. If they all attempt to create the facility with `IPC_EXCL` in effect, only the first attempt succeeds. If neither of these flags is given and the facility already exists, the functions to get access simply return the ID of the facility. If `IPC_CREAT` is omitted and the facility is not already initialized, the calls fail. These control flags are combined, using logical (bitwise) OR, with the octal permission modes to form the flags argument. For example, the statement below initializes a new message queue if the queue does not exist.

```
msqid = msgget(ftok("/tmp",
key), (IPC_CREAT | IPC_EXCL | 0400));
```

The first argument evaluates to a `key` based on the string ("/tmp"). The second argument evaluates to the combined permissions and control flags.

Controlling message queues

The `msgctl()` function alters the permissions and other characteristics of a message queue. The owner or creator of a queue can change its ownership or permissions using `msgctl()` Also, any process with permission to do so can use `msgctl()` for control operations.

The `msgctl()` function is prototypes as follows:

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

The `msqid` argument must be the ID of an existing message queue. The `cmd argument` is one of:

`IPC_STAT`
-- Place information about the status of the queue in the data structure pointed to by `buf`. The process must have read permission for this call to succeed.

`IPC_SET`
-- Set the owner's user and group ID, the permissions, and the size (in number of bytes) of the message queue. A process must have the effective user ID of the owner, creator, or superuser for this call to succeed.

`IPC_RMID`
-- Remove the message queue specified by the `msqid` argument.

The following code illustrates the `msgctl()` function with all its various flags:

```
#include<sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
...
if (msgctl(msqid, IPC_STAT, &buf) == -1) {
perror("msgctl: msgctl failed");
exit(1);
}
...
if (msgctl(msqid, IPC_SET, &buf) == -1) {
perror("msgctl: msgctl failed");
exit(1);
}
...
```

Sending and Receiving Messages

The `msgsnd()` and `msgrcv()` functions send and receive messages, respectively:

```
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);
```

The `msqid` argument **must** be the ID of an existing message queue. The `msgp` argument is a pointer to a structure that contains the type of the message and its text. The structure below is an example of what this user-defined buffer might look like:

```
 struct mymsg {
      long      mtype;    /* message type */
      char mtext[MSGSZ]; /* message text of length MSGSZ */
}
```

The `msgsz` argument specifies the length of the message in bytes.

The structure member `msgtype` is the received message's type as specified by the sending process.

The argument `msgflg` specifies the action to be taken if one or more of the following are true:

- The number of bytes already on the queue is equal to `msg_qbytes`.
- The total number of messages on all queues system-wide is equal to the system-imposed limit.

These actions are as follows:

- If (`msgflg & IPC_NOWAIT`) is non-zero, the message will not be sent and the calling process will return immediately.
- If (`msgflg & IPC_NOWAIT`) is 0, the calling process will suspend execution until one of the following occurs:
    - The condition responsible for the suspension no longer exists, in which case the message is sent.
    - The message queue identifier `msqid` is removed from the system; when this occurs, `errno` is set equal to `EIDRM` and -1 is returned.
    - The calling process receives a signal that is to be caught; in this case the message is not sent and the calling process resumes execution.

Upon successful completion, the following actions are taken with respect to the data structure associated with `msqid`:

- `msg_qnum` is incremented by 1.
- `msg_lspid` is set equal to the process ID of the calling process.
- `msg_stime` is set equal to the current time.

The following code illustrates `msgsnd()` and `msgrcv()`:

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

...

int msgflg; /* message flags for the operation */
struct msgbuf *msgp; /* pointer to the message buffer */
int msgsz; /* message size */
long msgtyp; /* desired message type */
int msqid /* message queue ID to be used */

...

msgp = (struct msgbuf *)malloc((unsigned)(sizeof(struct msgbuf)
- sizeof msgp->mtext + maxmsgsz));

if (msgp == NULL) {
(void) fprintf(stderr, "msgop: %s %d byte messages.\n",
"could not allocate message buffer for", maxmsgsz);
exit(1);

...
msgsz = ...
msgflg = ...

if (msgsnd(msqid, msgp, msgsz, msgflg) == -1)
perror("msgop: msgsnd failed");
...
msgsz = ...
msgtyp = first_on_queue;
msgflg = ...
if (rtrn = msgrcv(msqid, msgp, msgsz, msgtyp, msgflg) == -1)
perror("msgop: msgrcv failed");
...
```

POSIX Messages: <`mqueue.h`>

The POSIX message queue functions are:

`mq_open()` -- Connects to, and optionally creates, a named message queue.

`mq_close()` -- Ends the connection to an open message queue.

`mq_unlink()` -- Ends the connection to an open message queue and causes the queue to be removed when the last process closes it.

`mq_send()` -- Places a message in the queue.

`mq_receive()` -- Receives (removes) the oldest, highest priority message from the queue.

`mq_notify()` -- Notifies a process or thread that a message is available in the queue.

`mq_setattr()` -- Set or get message queue attributes.

The basic operation of these functions is as described above. For full function prototypes and further information see the UNIX `man pages`

Example: Sending messages between two processes

The following two programs should be compiled and run ***at the same time*** to illustrate basic principle of message passing:

`message_send.c`
-- Creates a message queue and sends one message to the queue.

`message_rec.c`
-- Reads the message from the queue.

`message_send.c` -- creating and sending to a simple message queue

The full code listing for `message_send.c` is as follows:

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

#define MSGSZ     128


/*
 * Declare the message structure.
 */

typedef struct msgbuf {
         long    mtype;
         char    mtext[MSGSZ];
         } message_buf;

main()
{
    int msqid;
    int msgflg = IPC_CREAT | 0666;
    key_t key;
    message_buf sbuf;
    size_t buf_length;

    /*
     * Get the message queue id for the
     * "name" 1234, which was created by
     * the server.
     */
    key = 1234;

(void) fprintf(stderr, "\nmsgget: Calling msgget(%#lx,\
%#o)\n",
key, msgflg);

if ((msqid = msgget(key, msgflg )) < 0) {
    perror("msgget");
    exit(1);
}
else
 (void) fprintf(stderr,"msgget: msgget succeeded: msqid = %d\n", msqid);


/*
 * We'll send message type 1
 */

sbuf.mtype = 1;

(void) fprintf(stderr,"msgget: msgget succeeded: msqid = %d\n", msqid);

(void) strcpy(sbuf.mtext, "Did you get this?");

(void) fprintf(stderr,"msgget: msgget succeeded: msqid = %d\n", msqid);

buf_length = strlen(sbuf.mtext) + 1 ;



/*
 * Send a message.
 */
if (msgsnd(msqid, &sbuf, buf_length, IPC_NOWAIT) < 0) {
   printf ("%d, %d, %s, %d\n", msqid, sbuf.mtype, sbuf.mtext, buf_length);
    perror("msgsnd");
```

```
        exit(1);
    }

   else
      printf("Message: \"%s\" Sent\n", sbuf.mtext);

    exit(0);
}
```

The essential points to note here are:

- The Message queue is created with a basic `key` and message flag `msgflg = IPC_CREAT | 0666` -- create queue and make it read and appendable by all.
- A message of type (`sbuf.mtype`) 1 is sent to the queue with the message ``Did you get this?"

`message_rec.c` -- receiving the above message

The full code listing for `message_send.c`'s companion process, `message_rec.c` is as follows:

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

#define MSGSZ     128

/*
 * Declare the message structure.
 */

typedef struct msgbuf {
    long    mtype;
    char    mtext[MSGSZ];
} message_buf;


main()
{
    int msqid;
    key_t key;
    message_buf  rbuf;

    /*
     * Get the message queue id for the
     * "name" 1234, which was created by
     * the server.
     */
    key = 1234;

    if ((msqid = msgget(key, 0666)) < 0) {
        perror("msgget");
        exit(1);
    }


    /*
     * Receive an answer of message type 1.
     */
    if (msgrcv(msqid, &rbuf, MSGSZ, 1, 0) < 0) {
        perror("msgrcv");
        exit(1);
    }

    /*
     * Print the answer.
     */
    printf("%s\n", rbuf.mtext);
    exit(0);
}
```

The essential points to note here are:

- The Message queue is opened with `msgget` (message flag `0666`) and the ***same*** `key` as `message_send.c`.
- A message of the ***same*** type 1 is received from the queue with the message ``Did you get this?" stored in `rbuf.mtext`.

Some further example message queue programs

The following suite of programs can be used to investigate interactively a variety of massage passing ideas (see exercises below).

The message queue **must** be initialised with the `msgget.c` program. The effects of controlling the queue and sending and receiving messages can be investigated with `msgctl.c` and `msgop.c` respectively.

`msgget.c`: Simple Program to illustrate `msget()`

```
/*
 * msgget.c: Illustrate the msgget() function.
 * This is a simple exerciser of the msgget() function. It prompts
 * for the arguments, makes the call, and reports the results.
 */

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

extern void  exit();
extern void  perror();

main()
{
 key_t  key;  /* key to be passed to msgget() */
 int  msgflg,  /* msgflg to be passed to msgget() */
   msqid;  /* return value from msgget() */

 (void) fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
  "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");
 (void) fprintf(stderr, "IPC_PRIVATE == %#lx\n", IPC_PRIVATE);
 (void) fprintf(stderr, "Enter key: ");
 (void) scanf("%li", &key);
 (void) fprintf(stderr, "\nExpected flags for msgflg argument
are:\n");
 (void) fprintf(stderr, "\tIPC_EXCL =\t%#8.8o\n", IPC_EXCL);
 (void) fprintf(stderr, "\tIPC_CREAT =\t%#8.8o\n", IPC_CREAT);
 (void) fprintf(stderr, "\towner read =\t%#8.8o\n", 0400);
 (void) fprintf(stderr, "\towner write =\t%#8.8o\n", 0200);
 (void) fprintf(stderr, "\tgroup read =\t%#8.8o\n", 040);
 (void) fprintf(stderr, "\tgroup write =\t%#8.8o\n", 020);
 (void) fprintf(stderr, "\tother read =\t%#8.8o\n", 04);
 (void) fprintf(stderr, "\tother write =\t%#8.8o\n", 02);
 (void) fprintf(stderr, "Enter msgflg value: ");
 (void) scanf("%i", &msgflg);

 (void) fprintf(stderr, "\nmsgget: Calling msgget(%#lx,
%#o)\n",
 key, msgflg);
 if ((msqid = msgget(key, msgflg)) == -1)
 {
  perror("msgget: msgget failed");
  exit(1);
 } else {
  (void) fprintf(stderr,
   "msgget: msgget succeeded: msqid = %d\n", msqid);
  exit(0);
 }
}
```

`msgctl.c` Sample Program to Illustrate `msgctl()`

```
/*
 * msgctl.c:  Illustrate the msgctl() function.
 *
 * This is a simple exerciser of the msgctl() function.  It allows
 * you to perform one control operation on one message queue.  It
 * gives up immediately if any control operation fails, so be
careful
 * not to set permissions to preclude read permission; you won't
be
 * able to reset the permissions with this code if you do.
 */
#include   <stdio.h>
#include   <sys/types.h>
#include   <sys/ipc.h>
#include   <sys/msg.h>
#include   <time.h>

static void do_msgctl();
extern void exit();
extern void perror();
static char warning_message[] = "If you remove read permission
for \
     yourself, this program will fail frequently!";

main()
{
 struct msqid_ds buf;     /* queue descriptor buffer for IPC_STAT
          and IP_SET commands */
 int    cmd,  /* command to be given to msgctl() */
     msqid;  /* queue ID to be given to msgctl() */

 (void fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
  "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
```

```
 (void) fprintf(stderr, "\totherwise, decimal.\n");

 /* Get the msqid and cmd arguments for the msgctl() call. */
 (void) fprintf(stderr,
  "Please enter arguments for msgctls() as requested.");
 (void) fprintf(stderr, "\nEnter the msqid: ");
 (void) scanf("%i", &msqid);
 (void) fprintf(stderr, "\tIPC_RMID = %d\n", IPC_RMID);
 (void) fprintf(stderr, "\tIPC_SET = %d\n", IPC_SET);
 (void) fprintf(stderr, "\tIPC_STAT = %d\n", IPC_STAT);
 (void) fprintf(stderr, "\nEnter the value for the command: ");
 (void) scanf("%i", &cmd);

 switch (cmd) {
  case IPC_SET:
   /* Modify settings in the message queue control structure.
*/
   (void) fprintf(stderr, "Before IPC_SET, get current
values:");
   /* fall through to IPC_STAT processing */
  case IPC_STAT:
   /* Get a copy of the current message queue control
    * structure and show it to the user. */
   do_msgctl(msqid, IPC_STAT, &buf);
   (void) fprintf(stderr, ]
   "msg_perm.uid = %d\n", buf.msg_perm.uid);
   (void) fprintf(stderr,
   "msg_perm.gid = %d\n", buf.msg_perm.gid);
   (void) fprintf(stderr,
   "msg_perm.cuid = %d\n", buf.msg_perm.cuid);
   (void) fprintf(stderr,
   "msg_perm.cgid = %d\n", buf.msg_perm.cgid);
   (void) fprintf(stderr, "msg_perm.mode = %#o, ",
   buf.msg_perm.mode);
   (void) fprintf(stderr, "access permissions = %#o\n",
   buf.msg_perm.mode & 0777);
   (void) fprintf(stderr, "msg_cbytes = %d\n",
       buf.msg_cbytes);
   (void) fprintf(stderr, "msg_qbytes = %d\n",
       buf.msg_qbytes);
   (void) fprintf(stderr, "msg_qnum = %d\n", buf.msg_qnum);
   (void) fprintf(stderr, "msg_lspid = %d\n",
       buf.msg_lspid);
   (void) fprintf(stderr, "msg_lrpid = %d\n",
       buf.msg_lrpid);
   (void) fprintf(stderr, "msg_stime = %s", buf.msg_stime ?
   ctime(&buf.msg_stime) : "Not Set\n");
   (void) fprintf(stderr, "msg_rtime = %s", buf.msg_rtime ?
   ctime(&buf.msg_rtime) : "Not Set\n");
   (void) fprintf(stderr, "msg_ctime = %s",
       ctime(&buf.msg_ctime));
   if (cmd == IPC_STAT)
    break;
   /*  Now continue with IPC_SET. */
   (void) fprintf(stderr, "Enter msg_perm.uid: ");
   (void) scanf ("%hi", &buf.msg_perm.uid);
   (void) fprintf(stderr, "Enter msg_perm.gid: ");
   (void) scanf("%hi", &buf.msg_perm.gid);
   (void) fprintf(stderr, "%s\n", warning_message);
   (void) fprintf(stderr, "Enter msg_perm.mode: ");
   (void) scanf("%hi", &buf.msg_perm.mode);
   (void) fprintf(stderr, "Enter msg_qbytes: ");
   (void) scanf("%hi", &buf.msg_qbytes);
   do_msgctl(msqid, IPC_SET, &buf);
   break;
 case IPC_RMID:
  default:
   /* Remove the message queue or try an unknown command. */
   do_msgctl(msqid, cmd, (struct msqid_ds *)NULL);
   break;
 }
 exit(0);
}

/*
 * Print indication of arguments being passed to msgctl(), call
 * msgctl(), and report the results. If msgctl() fails, do not
 * return; this example doesn't deal with errors, it just reports
 * them.
 */
static void
do_msgctl(msqid, cmd, buf)
struct msqid_ds    *buf;  /* pointer to queue descriptor buffer */
int    cmd,  /* command code */
    msqid;  /* queue ID */
{
 register int rtrn;  /* hold area for return value from msgctl()
*/

 (void) fprintf(stderr, "\nmsgctl: Calling msgctl(%d, %d,
%s)\n",
   msqid, cmd, buf ? "&buf" : "(struct msqid_ds *)NULL");
 rtrn = msgctl(msqid, cmd, buf);
 if (rtrn == -1) {
  perror("msgctl: msgctl failed");
```

```
  exit(1);
 } else {
  (void) fprintf(stderr, "msgctl: msgctl returned %d\n",
      rtrn);
 }
}
```

`msgop.c`: Sample Program to Illustrate `msgsnd()` and `msgrcv()`

```
/*
 * msgop.c: Illustrate the msgsnd() and msgrcv() functions.
 *
 * This is a simple exerciser of the message send and receive
 * routines. It allows the user to attempt to send and receive as
many
 * messages as wanted to or from one message queue.
 */

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

static int ask();
extern void exit();
extern char *malloc();
extern void perror();

char first_on_queue[] = "-> first message on queue",
 full_buf[] = "Message buffer overflow. Extra message text\
      discarded.";

main()
{
 register int    c;  /* message text input */
 int    choice; /* user's selected operation code */
 register int    i;  /* loop control for mtext */
 int    msgflg; /* message flags for the operation */
 struct msgbuf     *msgp;  /* pointer to the message buffer */
 int    msgsz;  /* message size */
 long    msgtyp;  /* desired message type */
 int    msqid,  /* message queue ID to be used */
     maxmsgsz, /* size of allocated message buffer */
     rtrn;  /* return value from msgrcv or msgsnd */
 (void) fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
  "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");
 /* Get the message queue ID and set up the message buffer. */
 (void) fprintf(stderr, "Enter msqid: ");
 (void) scanf("%i", &msqid);
 /*
  * Note that <sys/msg.h> includes a definition of struct
msgbuf
  * with the mtext field defined as:
  *   char mtext[1];
  * therefore, this definition is only a template, not a
structure
  * definition that you can use directly, unless you want only
to
  * send and receive messages of 0 or 1 byte. To handle this,
  * malloc an area big enough to contain the template - the size
  * of the mtext template field + the size of the mtext field
  * wanted. Then you can use the pointer returned by malloc as a
  * struct msgbuf with an mtext field of the size you want. Note
  * also that sizeof msgp->mtext is valid even though msgp
isn't
  * pointing to anything yet. Sizeof doesn't dereference msgp,
but
  * uses its type to figure out what you are asking about.
  */
 (void) fprintf(stderr,
  "Enter the message buffer size you want:");
 (void) scanf("%i", &maxmsgsz);
 if (maxmsgsz < 0) {
  (void) fprintf(stderr, "msgop: %s\n",
    "The message buffer size must be >= 0.");
  exit(1);
 }
 msgp = (struct msgbuf *)malloc((unsigned)(sizeof(struct
msgbuf)
    - sizeof msgp->mtext + maxmsgsz));
 if (msgp == NULL) {
  (void) fprintf(stderr, "msgop: %s %d byte messages.\n",
    "could not allocate message buffer for", maxmsgsz);
  exit(1);
 }
 /* Loop through message operations until the user is ready to
  quit. */
 while (choice = ask()) {
  switch (choice) {
  case 1: /* msgsnd() requested: Get the arguments, make the
```

```
 call, and report the results. */
(void) fprintf(stderr, "Valid msgsnd message %s\n",
 "types are positive integers.");
(void) fprintf(stderr, "Enter msgp->mtype: ");
(void) scanf("%li", &msgp->mtype);
if (maxmsgsz) {
 /* Since you've been using scanf, you need the loop
    below to throw away the rest of the input on the
    line after the entered mtype before you start
    reading the mtext. */
 while ((c = getchar()) != '\n' && c != EOF);
 (void) fprintf(stderr, "Enter a %s:\n",
     "one line message");
 for (i = 0; ((c = getchar()) != '\n'); i++) {
  if (i >= maxmsgsz) {
   (void) fprintf(stderr, "\n%s\n", full_buf);
   while ((c = getchar()) != '\n');
   break;
  }
  msgp->mtext[i] = c;
 }
 msgsz = i;
} else
 msgsz = 0;
(void) fprintf(stderr,"\nMeaningful msgsnd flag is:\n");
(void) fprintf(stderr, "\tIPC_NOWAIT =\t%#8.8o\n",
 IPC_NOWAIT);
(void) fprintf(stderr, "Enter msgflg: ");
(void) scanf("%i", &msgflg);
(void) fprintf(stderr, "%s(%d, msgp, %d, %#o)\n",
  "msgop: Calling msgsnd", msqid, msgsz, msgflg);
 (void) fprintf(stderr, "msgp->mtype = %ld\n",
     msgp->mtype);
 (void) fprintf(stderr, "msgp->mtext = \"");
 for (i = 0; i < msgsz; i++)
  (void) fputc(msgp->mtext[i], stderr);
  (void) fprintf(stderr, "\"\n");
  rtrn = msgsnd(msqid, msgp, msgsz, msgflg);
  if (rtrn == -1)
   perror("msgop: msgsnd failed");
  else
   (void) fprintf(stderr,
      "msgop: msgsnd returned %d\n", rtrn);
  break;
case 2: /* msgrcv() requested: Get the arguments, make the
     call, and report the results. */
 for (msgsz = -1; msgsz < 0 || msgsz > maxmsgsz;
    (void) scanf("%i", &msgsz))
  (void) fprintf(stderr, "%s (0 <= msgsz <= %d): ",
      "Enter msgsz", maxmsgsz);
 (void) fprintf(stderr, "msgtyp meanings:\n");
 (void) fprintf(stderr, "\t 0 %s\n", first_on_queue);
 (void) fprintf(stderr, "\t>0 %s of given type\n",
  first_on_queue);
 (void) fprintf(stderr, "\t<0 %s with type <= |msgtyp|\n",
     first_on_queue);
 (void) fprintf(stderr, "Enter msgtyp: ");
 (void) scanf("%li", &msgtyp);
 (void) fprintf(stderr,
     "Meaningful msgrcv flags are:\n");
 (void) fprintf(stderr, "\tMSG_NOERROR =\t%#8.8o\n",
     MSG_NOERROR);
 (void) fprintf(stderr, "\tIPC_NOWAIT =\t%#8.8o\n",
     IPC_NOWAIT);
 (void) fprintf(stderr, "Enter msgflg: ");
 (void) scanf("%i", &msgflg);
   (void) fprintf(stderr, "%s(%d, msgp, %d, %ld, %#o);\n",
       "msgop: Calling msgrcv", msqid, msgsz,
       msgtyp, msgflg);
   rtrn = msgrcv(msqid, msgp, msgsz, msgtyp, msgflg);
   if (rtrn == -1)
    perror("msgop: msgrcv failed");
   else {
    (void) fprintf(stderr, "msgop: %s %d\n",
        "msgrcv returned", rtrn);
    (void) fprintf(stderr, "msgp->mtype = %ld\n",
        msgp->mtype);
    (void) fprintf(stderr, "msgp->mtext is: \"");
    for (i = 0; i < rtrn; i++)
     (void) fputc(msgp->mtext[i], stderr);
    (void) fprintf(stderr, "\"\n");
   }
   break;
  default:
   (void) fprintf(stderr, "msgop: operation unknown\n");
   break;
  }
 }
 exit(0);
}

/*
 * Ask the user what to do next. Return the user's choice code.
 * Don't return until the user selects a valid choice.
 */
```

```
static
ask()
{
 int response; /* User's response. */

 do {
  (void) fprintf(stderr, "Your options are:\n");
  (void) fprintf(stderr, "\tExit =\t0 or Control-D\n");
  (void) fprintf(stderr, "\tmsgsnd =\t1\n");
  (void) fprintf(stderr, "\tmsgrcv =\t2\n");
  (void) fprintf(stderr, "Enter your choice: ");

  /* Preset response so "^D" will be interpreted as exit. */
  response = 0;
  (void) scanf("%i", &response);
 } while (response < 0 || response > 2);

 return(response);
}
```

Exercises

Exercise 12755

Write a 2 programs that will both send and messages and construct the following dialog between them

- (Process 1) Sends the message "Are you hearing me?"
- (Process 2) Receives the message and replies "Loud and Clear".
- (Process 1) Receives the reply and then says "I can hear you too".

Exercise 12756

Compile the programs `msgget.c`, `msgctl.c` and `msgop.c` and then

- investigate and understand fully the operations of the flags (access, creation ***etc.*** permissions) you can set interactively in the programs.
- Use the programs to:
  - Send and receive messages of two different message `types`.
  - Place several messages on the queue and inquire about the state of the queue with `msgctl.c`. Add/delete a few messages (using `msgop.c` and perform the inquiry once more.
  - Use `msgctl.c` to alter a message on the queue.
  - Use `msgctl.c` to delete a message from the queue.

Exercise 12757

Write a ***server*** program and two ***client*** programs so that the ***server*** can communicate privately to ***each client*** individually via a ***single*** message queue.

Exercise 12758

Implement a ***blocked*** or ***synchronous*** method of message passing using signal interrupts.

IPC:Semaphores

Semaphores are a programming construct designed by E. W. Dijkstra in the late 1960s. Dijkstra's model was the operation of railroads: consider a stretch of railroad in which there is a single track over which only one train at a time is allowed. Guarding this track is a semaphore. A train must wait before entering the single track until the semaphore is in a state that permits travel. When the train enters the track, the semaphore changes state to prevent other trains from entering the track. A train that is leaving this section of track must again change the state of the semaphore to allow another train to enter. In the computer version, a semaphore appears to be a simple integer. A process (or a thread) waits for permission to proceed by waiting for the integer to become 0. The signal if it proceeds signals that this by performing incrementing the integer by 1. When it is finished, the process changes the semaphore's value by subtracting one from it.

Semaphores let processes query or alter status information. They are often used to monitor and control the availability of system resources such as shared memory segments.

Semaphores can be operated on as individual units or as elements in a set. Because System V IPC semaphores can be in a large array, they are extremely heavy weight. Much lighter weight semaphores are available in the threads library (see `man semaphore` and also Chapter 30.3) and POSIX semaphores (see below briefly). Threads library semaphores must be used with mapped memory . A semaphore set consists of a control structure and an array of individual semaphores. A set of semaphores can contain up to 25 elements.

In a similar fashion to message queues, the semaphore set must be initialized using `semget()`; the semaphore creator can change its ownership or permissions using `semctl()`; and semaphore operations are performed via the `semop()` function. These are now discussed below:

Initializing a Semaphore Set

The function `semget()` initializes or gains access to a semaphore. It is prototyped by:

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

When the call succeeds, it returns the semaphore ID (`semid`).

The `key` argument is a access value associated with the semaphore ID.

The `nsems` argument specifies the number of elements in a semaphore array. The call fails when `nsems` is greater than the number of elements in an existing array; when the correct count is not known, supplying 0 for this argument ensures that it will succeed.

The `semflg` argument specifies the initial access permissions and creation control flags.

The following code illustrates the semget() function.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
...
key_t key; /* key to pass to semget() */
int semflg; /* semflg to pass tosemget() */
int nsems; /* nsems to pass to semget() */
int semid; /* return value from semget() */

...

key = ...
nsems = ...
semflg = ... ...
if ((semid = semget(key, nsems, semflg)) == -1) {
               perror("semget: semget failed");
               exit(1); }
else
   ...
```

Controlling Semaphores

`semctl()` changes permissions and other characteristics of a semaphore set. It is prototyped as follows:

```
int semctl(int semid, int semnum, int cmd, union semun arg);
```

It must be called with a valid semaphore ID, `semid`. The `semnum` value selects a semaphore within an array by its index. The `cmd` argument is one of the following control flags:

GETVAL
-- Return the value of a single semaphore.

SETVAL
-- Set the value of a single semaphore. In this case, arg is taken as arg.val, an int.

GETPID
-- Return the `PID` of the process that performed the last operation on the semaphore or array.

GETNCNT
-- Return the number of processes waiting for the value of a semaphore to increase.

GETZCNT
-- Return the number of processes waiting for the value of a particular semaphore to reach zero.

GETALL
-- Return the values for all semaphores in a set. In this case, `arg` is taken as `arg.array`, a pointer to an array of unsigned shorts (see below).

SETALL
-- Set values for all semaphores in a set. In this case, `arg` is taken as `arg.array`, a pointer to an array of unsigned shorts.

IPC_STAT
-- Return the status information from the control structure for the semaphore set and place it in the data structure pointed to by `arg.buf`, a pointer to a buffer of type `semid_ds`.

IPC_SET
-- Set the effective user and group identification and permissions. In this case, `arg` is taken as `arg.buf`.

IPC_RMID
-- Remove the specified semaphore set.

A process must have an effective user identification of owner, creator, or superuser to perform an `IPC_SET` or `IPC_RMID` command. Read and write permission is required as for the other control commands. The following code illustrates `semctl ()`.

The fourth argument `union semun arg` is optional, depending upon the operation requested. If required it is of type `union semun`, which must be ***explicitly*** declared by the application program as:

```
          union semun {
               int val;
               struct semid_ds *buf;
               ushort *array;
          } arg;
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

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

int i;
int semnum = ....;
int cmd = GETALL; /* get value */



...
i = semctl(semid, semnum, cmd, arg);
if (i == -1) {
               perror("semctl: semctl failed");
  exit(1);
 }
else
...
```

Semaphore Operations

`semop()` performs operations on a semaphore set. It is prototyped by:

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

The `semid` argument is the semaphore ID returned by a previous `semget()` call. The `sops` argument is a pointer to an array of structures, each containing the following information about a semaphore operation:

- The semaphore number
- The operation to be performed
- Control flags, if any.

The `sembuf` structure specifies a semaphore operation, as defined in `<sys/sem.h>`.

```
struct sembuf {
        ushort_t        sem_num;        /* semaphore number */
        short           sem_op;         /* semaphore operation */
        short           sem_flg;        /* operation flags */
};
```

The `nsops` argument specifies the length of the array, the maximum size of which is determined by the `SEMOPM` configuration option; this is the maximum number of operations allowed by a single semop() call, and is set to 10 by default. The operation to be performed is determined as follows:

- A positive integer increments the semaphore value by that amount.
- A negative integer decrements the semaphore value by that amount. An attempt to set a semaphore to a value less than zero fails or blocks, depending on whether `IPC_NOWAIT` is in effect.
- A value of zero means to wait for the semaphore value to reach zero.

There are two control flags that can be used with `semop()`:

`IPC_NOWAIT`
-- Can be set for any operations in the array. Makes the function return without changing any semaphore value if any operation for which `IPC_NOWAIT` is set cannot be performed. The function fails if it tries to decrement a semaphore more than its current value, or tests a nonzero semaphore to be equal to zero.

`SEM_UNDO`
-- Allows individual operations in the array to be undone when the process exits.

This function takes a pointer, `sops`, to an array of semaphore operation structures. Each structure in the array contains data about an operation to perform on a semaphore. Any process with read permission can test whether a semaphore has a zero value. To increment or decrement a semaphore requires write permission. When an operation fails, none of the semaphores is altered.

The process blocks (unless the `IPC_NOWAIT` flag is set), and remains blocked until:

- the semaphore operations can all finish, so the call succeeds,
- the process receives a signal, or
- the semaphore set is removed.

Only one process at a time can update a semaphore. Simultaneous requests by different processes are performed in an arbitrary order. When an array of operations is given by a `semop()` call, no updates are done until all operations on the array can finish successfully.

If a process with exclusive use of a semaphore terminates abnormally and fails to undo the operation or free the semaphore, the semaphore stays locked in memory in the state the process left it. To prevent this, the `SEM_UNDO` control flag makes `semop()` allocate an undo structure for each semaphore operation, which contains the operation that returns the semaphore to its previous state. If the process dies, the system applies the operations in the undo structures. This prevents an aborted process from leaving a semaphore set in an inconsistent state. If processes share access to a resource controlled by a semaphore, operations on the semaphore should not be made with `SEM_UNDO` in effect. If the process that currently has control of the resource terminates abnormally, the resource is presumed to be inconsistent. Another process must be able to recognize this to restore the resource to a consistent state. When performing a semaphore operation with `SEM_UNDO` in effect, you must also have it in effect for the call that will perform the reversing operation. When the process runs normally, the reversing operation updates the undo structure with a complementary value. This ensures that, unless the process is aborted, the values applied to the undo structure are cancel to zero. When the undo structure reaches zero, it is removed.

NOTE: Using `SEM_UNDO` inconsistently can lead to excessive resource consumption because allocated undo structures might not be freed until the system is rebooted.

The following code illustrates the `semop()` function:

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

...
int i;
int nsops; /* number of operations to do */
int semid; /* semid of semaphore set */
struct sembuf *sops; /* ptr to operations to perform */
...

if ((semid = semop(semid, sops, nsops)) == -1)
{
        perror("semop: semop failed");
 exit(1);
}
else
(void) fprintf(stderr, "semop: returned %d\n", i);
...
```

POSIX Semaphores: <semaphore.h>

POSIX semaphores are much lighter weight than are System V semaphores. A POSIX semaphore structure defines a single semaphore, not an array of up to twenty five semaphores. The POSIX semaphore functions are:

`sem_open()` -- Connects to, and optionally creates, a named semaphore

`sem_init()` -- Initializes a semaphore structure (internal to the calling program, so not a named semaphore).

`sem_close()` -- Ends the connection to an open semaphore.

`sem_unlink()` -- Ends the connection to an open semaphore and causes the semaphore to be removed when the last process closes it.

`sem_destroy()` -- Initializes a semaphore structure (internal to the calling program, so not a named semaphore).

`sem_getvalue()` -- Copies the value of the semaphore into the specified integer.

`sem_wait(), sem_trywait()` -- Blocks while the semaphore is held by other processes or returns an error if the semaphore is held by another process.

`sem_post()` -- Increments the count of the semaphore.

The basic operation of these functions is essence the same as described above, except note there are more specialised functions, here. These are not discussed further here and the reader is referred to the online `man` pages for further details.

`semaphore.c`: Illustration of simple semaphore passing

```
/* semaphore.c --- simple illustration of dijkstra's semaphore analogy
 *
 *   We fork() a  child process so that we have two processes running:
 *   Each process communicates via a semaphore.
 *   The respective process can only do its work (not much here)
 *   When it notices that the semaphore track is free when it returns to 0
 *   Each process must modify the semaphore accordingly
 */

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

 union semun {
              int val;
              struct semid_ds *buf;
```

```
              ushort *array;
          };

main()
{ int i,j;
  int pid;
  int semid; /* semid of semaphore set */
  key_t key = 1234; /* key to pass to semget() */
  int semflg = IPC_CREAT | 0666; /* semflg to pass to semget() */
  int nsems = 1; /* nsems to pass to semget() */
  int nsops; /* number of operations to do */
  struct sembuf *sops = (struct sembuf *) malloc(2*sizeof(struct sembuf));
  /* ptr to operations to perform */

  /* set up semaphore */

  (void) fprintf(stderr, "\nsemget: Setting up seamaphore: semget(%#lx, %\
%#o)\n",key, nsems, semflg);
   if ((semid = semget(key, nsems, semflg)) == -1) {
        perror("semget: semget failed");
        exit(1);
      } else
        (void) fprintf(stderr, "semget: semget succeeded: semid =\
%d\n", semid);


  /* get child process */

   if ((pid = fork()) < 0) {
        perror("fork");
        exit(1);
    }

if (pid == 0)
     { /* child */
       i = 0;


       while (i  < 3) {/* allow for 3 semaphore sets */

       nsops = 2;

       /* wait for semaphore to reach zero */

       sops[0].sem_num = 0; /* We only use one track */
       sops[0].sem_op = 0; /* wait for semaphore flag to become zero */
       sops[0].sem_flg = SEM_UNDO; /* take off semaphore asynchronous  */


       sops[1].sem_num = 0;
       sops[1].sem_op = 1; /* increment semaphore -- take control of track */
       sops[1].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore */

       /* Recap the call to be made. */

       (void) fprintf(stderr,"\nsemop:Child  Calling semop(%d, &sops, %d) with:", semid, nsops);
       for (j = 0; j < nsops; j++)
        {
          (void) fprintf(stderr, "\n\tsops[%d].sem_num = %d, ", j, sops[j].sem_num);
          (void) fprintf(stderr, "sem_op = %d, ", sops[j].sem_op);
          (void) fprintf(stderr, "sem_flg = %#o\n", sops[j].sem_flg);
        }

       /* Make the semop() call and report the results. */
        if ((j = semop(semid, sops, nsops)) == -1) {
               perror("semop: semop failed");
               }
          else
            {
              (void) fprintf(stderr, "\tsemop: semop returned %d\n", j);

              (void) fprintf(stderr, "\n\nChild Process Taking Control of Track: %d/3 times\n", i+1);
              sleep(5); /* DO Nothing for 5 seconds */

               nsops = 1;

              /* wait for semaphore to reach zero */
              sops[0].sem_num = 0;
               sops[0].sem_op = -1; /* Give UP COntrol of track */
              sops[0].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore, asynchronous  */


              if ((j = semop(semid, sops, nsops)) == -1) {
                     perror("semop: semop failed");
                     }
                 else
                    (void) fprintf(stderr, "Child Process Giving up Control of Track: %d/3 times\n", i+1);
              sleep(5); /* halt process to allow parent to catch semaphor change first */
             }
      ++i;
    }

   }
else /* parent */
   {  /* pid hold id of child */
```

```
 i = 0;


while (i  < 3) { /* allow for 3 semaphore sets */

nsops = 2;

/* wait for semaphore to reach zero */
sops[0].sem_num = 0;
sops[0].sem_op = 0; /* wait for semaphore flag to become zero */
sops[0].sem_flg = SEM_UNDO; /* take off semaphore asynchronous  */


sops[1].sem_num = 0;
sops[1].sem_op = 1; /* increment semaphore -- take control of track */
sops[1].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore */

/* Recap the call to be made. */

(void) fprintf(stderr,"\nsemop:Parent Calling semop(%d, &sops, %d) with:", semid, nsops);
for (j = 0; j < nsops; j++)
 {
   (void) fprintf(stderr, "\n\tsops[%d].sem_num = %d, ", j, sops[j].sem_num);
   (void) fprintf(stderr, "sem_op = %d, ", sops[j].sem_op);
   (void) fprintf(stderr, "sem_flg = %#o\n", sops[j].sem_flg);
 }

/* Make the semop() call and report the results. */
 if ((j = semop(semid, sops, nsops)) == -1) {
                perror("semop: semop failed");
                }
           else
             {
                (void) fprintf(stderr, "semop: semop returned %d\n", j);

                (void) fprintf(stderr, "Parent Process Taking Control of Track: %d/3 times\n", i+1);
                sleep(5); /* Do nothing for 5 seconds */

                 nsops = 1;

                /* wait for semaphore to reach zero */
                sops[0].sem_num = 0;
                 sops[0].sem_op = -1; /* Give UP COntrol of track */
                sops[0].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore, asynchronous  */

                if ((j = semop(semid, sops, nsops)) == -1) {
                        perror("semop: semop failed");
                        }
                   else
                      (void) fprintf(stderr, "Parent Process Giving up Control of Track: %d/3 times\n",
i+1);
                sleep(5); /* halt process to allow child to catch semaphor change first */
              }
        ++i;

      }

     }
}
```

The key elements of this program are as follows:

- After a semaphore is created with as simple key `1234`, two prcesses are forked.
- Each process (parent and child) essentially performs the same operations:
  - Each process accesses the same semaphore ***track*** (`sops[].sem_num = 0`).
  - Each process waits for the ***track*** to become free and then attempts to take control of ***track***

This is achieved by setting appropriate `sops[].sem_op` values in the array.

- Once the process has control it sleeps for 5 seconds (in reality some processing would take place in place of this simple illustration)
- The process then gives up control of the ***track*** `sops[1].sem_op = -1`
- an additional sleep operation is then performed to ensure that the other process has time to access the semaphore before a subsequent (same process) semaphore read.

Note: There is no synchronisation here in this simple example an we have no control over how the OS will schedule the processes.

Some further example semaphore programs

The following suite of programs can be used to investigate interactively a variety of semaphore ideas (see exercises below).

The semaphore **must** be initialised with the `semget.c` program. The effects of controlling the semaphore queue and sending and receiving semaphore can be investigated with `semctl.c` and `semop.c` respectively.

`semget.c`: Illustrate the `semget()` function

```
/*
 * semget.c: Illustrate the semget() function.
 *
 * This is a simple exerciser of the semget() function. It prompts
 * for the arguments, makes the call, and reports the results.
*/

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

extern void    exit();
extern void    perror();

main()
{
 key_t  key;   /* key to pass to semget() */
 int  semflg;   /* semflg to pass to semget() */
 int  nsems;   /* nsems to pass to semget() */
 int  semid;   /* return value from semget() */

 (void) fprintf(stderr,
  "All numeric input must follow C conventions:\n");
 (void) fprintf(stderr,
  "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");
 (void) fprintf(stderr, "IPC_PRIVATE == %#lx\n", IPC_PRIVATE);
 (void) fprintf(stderr, "Enter key: ");
 (void) scanf("%li", &key);

 (void) fprintf(stderr, "Enter nsems value: ");
 (void) scanf("%i", &nsems);
 (void) fprintf(stderr, "\nExpected flags for semflg are:\n");
 (void) fprintf(stderr, "\tIPC_EXCL = \t%#8.8o\n", IPC_EXCL);
 (void) fprintf(stderr, "\tIPC_CREAT = \t%#8.8o\n",
IPC_CREAT);
 (void) fprintf(stderr, "\towner read = \t%#8.8o\n", 0400);
 (void) fprintf(stderr, "\towner alter = \t%#8.8o\n", 0200);
 (void) fprintf(stderr, "\tgroup read = \t%#8.8o\n", 040);
 (void) fprintf(stderr, "\tgroup alter = \t%#8.8o\n", 020);
 (void) fprintf(stderr, "\tother read = \t%#8.8o\n", 04);
 (void) fprintf(stderr, "\tother alter = \t%#8.8o\n", 02);
 (void) fprintf(stderr, "Enter semflg value: ");
 (void) scanf("%i", &semflg);
 (void) fprintf(stderr, "\nsemget: Calling semget(%#lx, %
     %#o)\n",key, nsems, semflg);
 if ((semid = semget(key, nsems, semflg)) == -1) {
  perror("semget: semget failed");
  exit(1);
 } else {
  (void) fprintf(stderr, "semget: semget succeeded: semid =
%d\n",
   semid);
  exit(0);
 }
}
```

`semctl.c`: Illustrate the `semctl()` function

```
/*
 * semctl.c:   Illustrate the semctl() function.
 *
 * This is a simple exerciser of the semctl() function. It lets you
 * perform one control operation on one semaphore set. It gives up
 * immediately if any control operation fails, so be careful not
to
 * set permissions to preclude read permission; you won't be able
to
 * reset the permissions with this code if you do.
 */

#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/sem.h>
#include    <time.h>

struct semid_ds semid_ds;

static void   do_semctl();
static void   do_stat();
extern char   *malloc();
extern void   exit();
extern void   perror();

char    warning_message[] = "If you remove read permission\
    for yourself, this program will fail frequently!";

main()
{
 union semun    arg;    /* union to pass to semctl() */
 int    cmd,    /* command to give to semctl() */
```

```
    i,    /* work area */
    semid,    /* semid to pass to semctl() */
    semnum;    /* semnum to pass to semctl() */

(void) fprintf(stderr,
   "All numeric input must follow C conventions:\n");
(void) fprintf(stderr,
   "\t0x... is interpreted as hexadecimal,\n");
(void) fprintf(stderr, "\t0... is interpreted as octal,\n");
(void) fprintf(stderr, "\totherwise, decimal.\n");
(void) fprintf(stderr, "Enter semid value: ");
(void) scanf("%i", &semid);

(void) fprintf(stderr, "Valid semctl cmd values are:\n");
(void) fprintf(stderr, "\tGETALL = %d\n", GETALL);
(void) fprintf(stderr, "\tGETNCNT = %d\n", GETNCNT);
(void) fprintf(stderr, "\tGETPID = %d\n", GETPID);
(void) fprintf(stderr, "\tGETVAL = %d\n", GETVAL);
(void) fprintf(stderr, "\tGETZCNT = %d\n", GETZCNT);
(void) fprintf(stderr, "\tIPC_RMID = %d\n", IPC_RMID);
(void) fprintf(stderr, "\tIPC_SET = %d\n", IPC_SET);
(void) fprintf(stderr, "\tIPC_STAT = %d\n", IPC_STAT);
(void) fprintf(stderr, "\tSETALL = %d\n", SETALL);
(void) fprintf(stderr, "\tSETVAL = %d\n", SETVAL);
(void) fprintf(stderr, "\nEnter cmd: ");
(void) scanf("%i", &cmd);

/* Do some setup operations needed by multiple commands. */
switch (cmd) {
 case SETVAL:
 case GETNCNT:
 case GETZCNT:
  /* Get the semaphore number for these commands. */
  (void) fprintf(stderr, "\nEnter semnum value: ");
  (void) scanf("%i", &semnum);
  break;
 case GETALL:
 case SETALL:
  /* Allocate a buffer for the semaphore values. */
  (void) fprintf(stderr,
   "Get number of semaphores in the set.\n");
  arg.buf = &semid_ds;
  do_semctl(semid, 0, IPC_STAT, arg);
  if (arg.array =
   (ushort *)malloc((unsigned)
    (semid_ds.sem_nsems * sizeof(ushort)))) {
   /* Break out if you got what you needed. */
   break;
  }
  (void) fprintf(stderr,
   "semctl: unable to allocate space for %d values\n",
   semid_ds.sem_nsems);
  exit(2);
}

/* Get the rest of the arguments needed for the specified
   command. */
switch (cmd) {
 case SETVAL:
  /* Set value of one semaphore. */
  (void) fprintf(stderr, "\nEnter semaphore value: ");
  (void) scanf("%i", &arg.val);
  do_semctl(semid, semnum, SETVAL, arg);
  /* Fall through to verify the result. */
  (void) fprintf(stderr,
   "Do semctl GETVAL command to verify results.\n");
case GETVAL:
 /* Get value of one semaphore. */
 arg.val = 0;
 do_semctl(semid, semnum, GETVAL, arg);
 break;
case GETPID:
 /* Get PID of last process to successfully complete a
    semctl(SETVAL), semctl(SETALL), or semop() on the
    semaphore. */
 arg.val = 0;
 do_semctl(semid, 0, GETPID, arg);
 break;
case GETNCNT:
 /* Get number of processes waiting for semaphore value to
    increase. */
 arg.val = 0;
 do_semctl(semid, semnum, GETNCNT, arg);
 break;
case GETZCNT:
 /* Get number of processes waiting for semaphore value to
    become zero. */
 arg.val = 0;
 do_semctl(semid, semnum, GETZCNT, arg);
 break;
case SETALL:
 /* Set the values of all semaphores in the set. */
 (void) fprintf(stderr,
     "There are %d semaphores in the set.\n",
```

```
    semid_ds.sem_nsems);
 (void) fprintf(stderr, "Enter semaphore values:\n");
 for (i = 0; i < semid_ds.sem_nsems; i++) {
  (void) fprintf(stderr, "Semaphore %d: ", i);
  (void) scanf("%hi", &arg.array[i]);
 }
 do_semctl(semid, 0, SETALL, arg);
 /* Fall through to verify the results. */
 (void) fprintf(stderr,
  "Do semctl GETALL command to verify results.\n");
case GETALL:
 /* Get and print the values of all semaphores in the
    set.*/
 do_semctl(semid, 0, GETALL, arg);
 (void) fprintf(stderr,
     "The values of the %d semaphores are:\n",
     semid_ds.sem_nsems);
 for (i = 0; i < semid_ds.sem_nsems; i++)
  (void) fprintf(stderr, "%d ", arg.array[i]);
 (void) fprintf(stderr, "\n");
 break;
case IPC_SET:
 /* Modify mode and/or ownership. */
 arg.buf = &semid_ds;
 do_semctl(semid, 0, IPC_STAT, arg);
 (void) fprintf(stderr, "Status before IPC_SET:\n");
 do_stat();
 (void) fprintf(stderr, "Enter sem_perm.uid value: ");
 (void) scanf("%hi", &semid_ds.sem_perm.uid);
 (void) fprintf(stderr, "Enter sem_perm.gid value: ");
   (void) scanf("%hi", &semid_ds.sem_perm.gid);
   (void) fprintf(stderr, "%s\n", warning_message);
   (void) fprintf(stderr, "Enter sem_perm.mode value: ");
   (void) scanf("%hi", &semid_ds.sem_perm.mode);
   do_semctl(semid, 0, IPC_SET, arg);
   /* Fall through to verify changes. */
   (void) fprintf(stderr, "Status after IPC_SET:\n");
  case IPC_STAT:
   /* Get and print current status. */
   arg.buf = &semid_ds;
   do_semctl(semid, 0, IPC_STAT, arg);
   do_stat();
   break;
  case IPC_RMID:
   /* Remove the semaphore set. */
   arg.val = 0;
   do_semctl(semid, 0, IPC_RMID, arg);
   break;
  default:
   /* Pass unknown command to semctl. */
   arg.val = 0;
   do_semctl(semid, 0, cmd, arg);
   break;
 }
 exit(0);
}

/*
 * Print indication of arguments being passed to semctl(), call
 * semctl(), and report the results. If semctl() fails, do not
 * return; this example doesn't deal with errors, it just reports
 * them.
 */
static void
do_semctl(semid, semnum, cmd, arg)
union semun  arg;
int  cmd,
  semid,
  semnum;
{
 register int      i;   /* work area */

 void) fprintf(stderr, "\nsemctl: Calling semctl(%d, %d, %d,
",
    semid, semnum, cmd);
 switch (cmd) {
  case GETALL:
   (void) fprintf(stderr, "arg.array = %#x)\n",
       arg.array);
   break;
  case IPC_STAT:
  case IPC_SET:
   (void) fprintf(stderr, "arg.buf = %#x)\n", arg.buf);
   break;
  case SETALL:
   (void) fprintf(stderr, "arg.array = [", arg.buf);
   for (i = 0;i < semid_ds.sem_nsems;) {
    (void) fprintf(stderr, "%d", arg.array[i++]);
    if (i < semid_ds.sem_nsems)
      (void) fprintf(stderr, ", ");
   }
   (void) fprintf(stderr, "])\n");
   break;
  case SETVAL:
  default:
```

```
   (void) fprintf(stderr, "arg.val = %d)\n", arg.val);
   break;
 }
 i = semctl(semid, semnum, cmd, arg);
 if (i == -1) {
  perror("semctl: semctl failed");
  exit(1);
 }
 (void) fprintf(stderr, "semctl: semctl returned %d\n", i);
 return;
}

/*
 * Display contents of commonly used pieces of the status
structure.
 */
static void
do_stat()
{
 (void) fprintf(stderr, "sem_perm.uid = %d\n",
       semid_ds.sem_perm.uid);
 (void) fprintf(stderr, "sem_perm.gid = %d\n",
       semid_ds.sem_perm.gid);
 (void) fprintf(stderr, "sem_perm.cuid = %d\n",
       semid_ds.sem_perm.cuid);
 (void) fprintf(stderr, "sem_perm.cgid = %d\n",
       semid_ds.sem_perm.cgid);
 (void) fprintf(stderr, "sem_perm.mode = %#o, ",
       semid_ds.sem_perm.mode);
 (void) fprintf(stderr, "access permissions = %#o\n",
       semid_ds.sem_perm.mode & 0777);
 (void) fprintf(stderr, "sem_nsems = %d\n",
semid_ds.sem_nsems);
 (void) fprintf(stderr, "sem_otime = %s", semid_ds.sem_otime ?
      ctime(&semid_ds.sem_otime) : "Not Set\n");
 (void) fprintf(stderr, "sem_ctime = %s",
       ctime(&semid_ds.sem_ctime));
}
```

`semop()` Sample Program to Illustrate `semop()`

```
/*
 * semop.c: Illustrate the semop() function.
 *
 * This is a simple exerciser of the semop() function. It lets you
 * to set up arguments for semop() and make the call. It then
reports
 * the results repeatedly on one semaphore set. You must have read
 * permission on the semaphore set or this exerciser will fail.
(It
 * needs read permission to get the number of semaphores in the set
 * and to report the values before and after calls to semop().)
 */

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

static int      ask();
extern void     exit();
extern void     free();
extern char     *malloc();
extern void     perror();

static struct semid_ds  semid_ds;           /* status of semaphore set */

static char    error_mesg1[] = "semop: Can't allocate space for %d\
        semaphore values. Giving up.\n";
static char   error_mesg2[] = "semop: Can't allocate space for %d\
        sembuf structures. Giving up.\n";

main()
{
 register int    i;   /* work area */
 int    nsops;   /* number of operations to do */
 int    semid;   /* semid of semaphore set */
 struct sembuf    *sops;   /* ptr to operations to perform */

 (void) fprintf(stderr,
    "All numeric input must follow C conventions:\n");
 (void) fprintf(stderr,
    "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");
 /* Loop until the invoker doesn't want to do anymore. */
 while (nsops = ask(&semid, &sops)) {
  /* Initialize the array of operations to be performed.*/
  for (i = 0; i < nsops; i++) {
   (void) fprintf(stderr,
     "\nEnter values for operation %d of %d.\n",
       i + 1, nsops);
   (void) fprintf(stderr,
     "sem_num(valid values are 0 <= sem_num < %d): ",
```

```
   semid_ds.sem_nsems);
 (void) scanf("%hi", &sops[i].sem_num);
 (void) fprintf(stderr, "sem_op: ");
 (void) scanf("%hi", &sops[i].sem_op);
 (void) fprintf(stderr,
   "Expected flags in sem_flg are:\n");
 (void) fprintf(stderr, "\tIPC_NOWAIT =\t%#6.6o\n",
   IPC_NOWAIT);
 (void) fprintf(stderr, "\tSEM_UNDO =\t%#6.6o\n",
   SEM_UNDO);
 (void) fprintf(stderr, "sem_flg: ");
 (void) scanf("%hi", &sops[i].sem_flg);
}

/* Recap the call to be made. */
(void) fprintf(stderr,
   "\nsemop: Calling semop(%d, &sops, %d) with:",
   semid, nsops);
for (i = 0; i < nsops; i++)
{
 (void) fprintf(stderr, "\nsops[%d].sem_num = %d, ", i,
    sops[i].sem_num);
 (void) fprintf(stderr, "sem_op = %d, ", sops[i].sem_op);
 (void) fprintf(stderr, "sem_flg = %#o\n",
    sops[i].sem_flg);
}

/* Make the semop() call and report the results. */
if ((i = semop(semid, sops, nsops)) == -1) {
 perror("semop: semop failed");
  } else {
   (void) fprintf(stderr, "semop: semop returned %d\n", i);
  }
 }
}

/*
 * Ask if user wants to continue.
 *
 * On the first call:
 * Get the semid to be processed and supply it to the caller.
 * On each call:
 *  1. Print current semaphore values.
 *  2. Ask user how many operations are to be performed on the next
 *     call to semop. Allocate an array of sembuf structures
 *     sufficient for the job and set caller-supplied pointer to
that
 *     array. (The array is reused on subsequent calls if it is big
 *     enough. If it isn't, it is freed and a larger array is
 *     allocated.)
 */
static
ask(semidp, sopsp)
int   *semidp;  /* pointer to semid (used only the first time) */
struct sembuf   **sopsp;
{
 static union semun     arg;  /* argument to semctl */
 int     i;  /* work area */
 static int     nsops = 0;  /* size of currently allocated
           sembuf array */
 static int     semid = -1;   /* semid supplied by user */
 static struct sembuf    *sops;     /* pointer to allocated array */

 if (semid < 0) {
  /* First call; get semid from user and the current state of
     the semaphore set. */
 (void) fprintf(stderr,
   "Enter semid of the semaphore set you want to use: ");
 (void) scanf("%i", &semid);
 *semidp = semid;
 arg.buf = &semid_ds;
 if (semctl(semid, 0, IPC_STAT, arg) == -1) {
  perror("semop: semctl(IPC_STAT) failed");
  /* Note that if semctl fails, semid_ds remains filled
     with zeros, so later test for number of semaphores will
     be zero. */
  (void) fprintf(stderr,
    "Before and after values are not printed.\n");
 } else {
  if ((arg.array = (ushort *)malloc(
   (unsigned)(sizeof(ushort) * semid_ds.sem_nsems)))
      == NULL) {
   (void) fprintf(stderr, error_mesg1,
      semid_ds.sem_nsems);
   exit(1);
  }
 }
}
/* Print current semaphore values. */
if (semid_ds.sem_nsems) {
 (void) fprintf(stderr,
     "There are %d semaphores in the set.\n",
     semid_ds.sem_nsems);
 if (semctl(semid, 0, GETALL, arg) == -1) {
  perror("semop: semctl(GETALL) failed");
```

```
  } else {
   (void) fprintf(stderr, "Current semaphore values are:");
   for (i = 0; i < semid_ds.sem_nsems;
    (void) fprintf(stderr, " %d", arg.array[i++]));
   (void) fprintf(stderr, "\n");
  }
 }
 /* Find out how many operations are going to be done in the
next
    call and allocate enough space to do it. */
 (void) fprintf(stderr,
     "How many semaphore operations do you want %s\n",
     "on the next call to semop()?");
 (void) fprintf(stderr, "Enter 0 or control-D to quit: ");
 i = 0;
 if (scanf("%i", &i) == EOF || i == 0)
  exit(0);
 if (i > nsops) {
  if (nsops)
   free((char *)sops);
  nsops = i;
  if ((sops = (struct sembuf *)malloc((unsigned)(nsops *
   sizeof(struct sembuf)))) == NULL) {
   (void) fprintf(stderr, error_mesg2, nsops);
   exit(2);
  }
 }
 *sopsp = sops;
 return (i);
}
```

Exercises

Exercise 12763

Write 2 programs that will communicate **both ways** (***i.e*** each process can read and write) when run concurrently via semaphores.

Exercise 12764

Modify the `semaphore.c` program to handle synchronous semaphore communication semaphores.

Exercise 12765

Write 3 programs that communicate together via semaphores according to the following specifications: `sem_server.c` -- a program that can communicate independently (on different semaphore tracks) with two clients programs. `sem_client1.c` -- a program that talks to `sem_server.c` on one track. `sem_client2.c` -- a program that talks to `sem_server.c` on another track to `sem_client1.c`.

Exercise 12766

Compile the programs `semget.c`, `semctl.c` and `semop.c` and then

- investigate and understand fully the operations of the flags (access, creation ***etc.*** permissions) you can set interactively in the programs.
- Use the prgrams to:
  - Send and receive semaphores of 3 different semaphore `tracks`.
  - Inquire about the state of the semaphore queue with `semctl.c`. Add/delete a few semaphores (using `semop.c` and perform the inquiry once more.
  - Use `semctl.c` to alter a semaphore on the queue.
  - Use `semctl.c` to delete a semaphore from the queue.

IPC:Shared Memory

Shared Memory is an efficeint means of passing data between programs. One program will create a memory portion which other processes (if permitted) can access.

In the Solaris 2.x operating system, the most efficient way to implement shared memory applications is to rely on the `mmap()` function and on the system's native virtual memory facility. Solaris 2.x also supports System V shared memory, which is another way to let multiple processes attach a segment of physical memory to their virtual address spaces. When write access is allowed for more than one process, an outside protocol or mechanism such as a semaphore can be used to prevent inconsistencies and collisions.

A process creates a shared memory segment using `shmget()|`. The original owner of a shared memory segment can assign ownership to another user with `shmctl()`. It can also revoke this assignment. Other processes with proper permission can perform various control functions on the shared memory segment using `shmctl()`. Once created, a shared segment can be attached to a process address space using `shmat()`. It can be detached using `shmdt()` (see `shmop()`). The attaching process must have the appropriate permissions

for `shmat()`. Once attached, the process can read or write to the segment, as allowed by the permission requested in the attach operation. A shared segment can be attached multiple times by the same process. A shared memory segment is described by a control structure with a unique ID that points to an area of physical memory. The identifier of the segment is called the shmid. The structure definition for the shared memory segment control structures and prototypews can be found in <`sys/shm.h`>.

Accessing a Shared Memory Segment

`shmget()` is used to obtain access to a shared memory segment. It is prottyped by:

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

The `key` argument is a access value associated with the semaphore ID. The `size` argument is the size in bytes of the requested shared memory. The `shmflg` argument specifies the initial access permissions and creation control flags.

When the call succeeds, it returns the shared memory segment ID. This call is also used to get the ID of an existing shared segment (from a process requesting sharing of some existing memory portion).

The following code illustrates `shmget()`:

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

...

key_t key; /* key to be passed to shmget() */
int shmflg; /* shmflg to be passed to shmget() */
int shmid; /* return value from shmget() */
int size; /* size to be passed to shmget() */

...

key = ...
size = ...
shmflg) = ...

if ((shmid = shmget (key, size, shmflg)) == -1) {
   perror("shmget: shmget failed"); exit(1); } else {
   (void) fprintf(stderr, "shmget: shmget returned %d\n", shmid);
   exit(0);
  }
...
```

Controlling a Shared Memory Segment

`shmctl()` is used to alter the permissions and other characteristics of a shared memory segment. It is prototyped as follows:

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

The process must have an effective `shmid` of owner, creator or superuser to perform this command. The `cmd` argument is one of following control commands:

`SHM_LOCK`
-- Lock the specified shared memory segment in memory. The process must have the effective ID of superuser to perform this command.

`SHM_UNLOCK`
-- Unlock the shared memory segment. The process must have the effective ID of superuser to perform this command.

`IPC_STAT`
-- Return the status information contained in the control structure and place it in the buffer pointed to by buf. The process must have read permission on the segment to perform this command.

`IPC_SET`
-- Set the effective user and group identification and access permissions. The process must have an effective ID of owner, creator or superuser to perform this command.

`IPC_RMID`
-- Remove the shared memory segment.

The `buf` is a sructure of type `struct shmid_ds` which is defined in <`sys/shm.h`>

The following code illustrates `shmctl()`:

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

...

int cmd; /* command code for shmctl() */
int shmid; /* segment ID */
struct shmid_ds shmid_ds; /* shared memory data structure to
                             hold results */
...
```

```
shmid = ...
cmd = ...
if ((rtrn = shmctl(shmid, cmd, shmid_ds)) == -1) {
    perror("shmctl: shmctl failed");
    exit(1);
   }
...
```

Attaching and Detaching a Shared Memory Segment

`shmat()` and `shmdt()` are used to attach and detach shared memory segments. They are prototypes as follows:

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

```
int shmdt(const void *shmaddr);
```

`shmat()` returns a pointer, `shmaddr`, to the head of the shared segment associated with a valid `shmid`. `shmdt()` detaches the shared memory segment located at the address indicated by `shmaddr`

. The following code illustrates calls to `shmat()` and `shmdt()`:

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

static struct state { /* Internal record of attached segments. */
          int shmid; /* shmid of attached segment */
          char *shmaddr; /* attach point */
          int shmflg; /* flags used on attach */
         } ap[MAXnap]; /* State of current attached segments. */
int nap; /* Number of currently attached segments. */

...

char *addr; /* address work variable */
register int i; /* work area */
register struct state *p; /* ptr to current state entry */
...

p = &ap[nap++];
p->shmid = ...
p->shmaddr = ...
p->shmflg = ...

p->shmaddr = shmat(p->shmid, p->shmaddr, p->shmflg);
if(p->shmaddr == (char *)-1) {
     perror("shmop: shmat failed");
     nap--;
    } else
    (void) fprintf(stderr, "shmop: shmat returned %#8.8x\n",
p->shmaddr);

...
i = shmdt(addr);
if(i == -1) {
    perror("shmop: shmdt failed");
    } else {
  (void) fprintf(stderr, "shmop: shmdt returned %d\n", i);

for (p = ap, i = nap; i--; p++)
  if (p->shmaddr == addr) *p = ap[--nap];
}
...
```

Example two processes comunicating via shared memory:

`shm_server.c, shm_client.c`

We develop two programs here that illustrate the passing of a simple piece of memery (a string) between the processes if running simulatenously:

`shm_server.c`
-- simply creates the string and shared memory portion.

`shm_client.c`
-- attaches itself to the created shared memory portion and uses the string (`printf`.

The code listings of the 2 programs no follow:

`shm_server.c`

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

#define SHMSZ     27
```

```
main()
{
    char c;
    int shmid;
    key_t key;
    char *shm, *s;

    /*
     * We'll name our shared memory segment
     * "5678".
     */
    key = 5678;

    /*
     * Create the segment.
     */
    if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
        perror("shmget");
        exit(1);
    }

    /*
     * Now we attach the segment to our data space.
     */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }

    /*
     * Now put some things into the memory for the
     * other process to read.
     */
    s = shm;

    for (c = 'a'; c <= 'z'; c++)
        *s++ = c;
    *s = NULL;

    /*
     * Finally, we wait until the other process
     * changes the first character of our memory
     * to '*', indicating that it has read what
     * we put there.
     */
    while (*shm != '*')
        sleep(1);

    exit(0);
}
```

shm_client.c

```
/*
 * shm-client - client program to demonstrate shared memory.
 */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

#define SHMSZ     27

main()
{
int shmid;
key_t key;
char *shm, *s;

/*
 * We need to get the segment named
 * "5678", created by the server.
 */
key = 5678;

/*
 * Locate the segment.
 */
if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
    perror("shmget");
    exit(1);
}

/*
 * Now we attach the segment to our data space.
 */
if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
    perror("shmat");
    exit(1);
}

/*
 * Now read what the server put in the memory.
 */
```

```
    for (s = shm; *s != NULL; s++)
        putchar(*s);
    putchar('\n');

    /*
     * Finally, change the first character of the
     * segment to '*', indicating we have read
     * the segment.
     */
    *shm = '*';

    exit(0);
}
```

POSIX Shared Memory

POSIX shared memory is actually a variation of mapped memory. The major differences are to use `shm_open()` to open the shared memory object (instead of calling `open()`) and use `shm_unlink()` to close and delete the object (instead of calling `close()` which does not remove the object). The options in `shm_open()` are substantially fewer than the number of options provided in `open()`.

Mapped memory

In a system with fixed memory (non-virtual), the address space of a process occupies and is limited to a portion of the system's main memory. In Solaris 2.x virtual memory the actual address space of a process occupies a file in the swap partition of disk storage (the file is called the backing store). Pages of main memory buffer the active (or recently active) portions of the process address space to provide code for the CPU(s) to execute and data for the program to process.

A page of address space is loaded when an address that is not currently in memory is accessed by a CPU, causing a page fault. Since execution cannot continue until the page fault is resolved by reading the referenced address segment into memory, the process sleeps until the page has been read. The most obvious difference between the two memory systems for the application developer is that virtual memory lets applications occupy much larger address spaces. Less obvious advantages of virtual memory are much simpler and more efficient file I/O and very efficient sharing of memory between processes.

Address Spaces and Mapping

Since backing store files (the process address space) exist only in swap storage, they are not included in the UNIX named file space. (This makes backing store files inaccessible to other processes.) However, it is a simple extension to allow the logical insertion of all, or part, of one, or more, named files in the backing store and to treat the result as a single address space. This is called mapping. With mapping, any part of any readable or writable file can be logically included in a process's address space. Like any other portion of the process's address space, no page of the file is not actually loaded into memory until a page fault forces this action. Pages of memory are written to the file only if their contents have been modified. So, reading from and writing to files is completely automatic and very efficient. More than one process can map a single named file. This provides very efficient memory sharing between processes. All or part of other files can also be shared between processes.

Not all named file system objects can be mapped. Devices that cannot be treated as storage, such as terminal and network device files, are examples of objects that cannot be mapped. A process address space is defined by all of the files (or portions of files) mapped into the address space. Each mapping is sized and aligned to the page boundaries of the system on which the process is executing. There is no memory associated with processes themselves.

A process page maps to only one object at a time, although an object address may be the subject of many process mappings. The notion of a "page" is not a property of the mapped object. Mapping an object only provides the potential for a process to read or write the object's contents. Mapping makes the object's contents directly addressable by a process. Applications can access the storage resources they use directly rather than indirectly through read and write. Potential advantages include efficiency (elimination of unnecessary data copying) and reduced complexity (single-step updates rather than the read, modify buffer, write cycle). The ability to access an object and have it retain its identity over the course of the access is unique to this access method, and facilitates the sharing of common code and data.

Because the file system name space includes any directory trees that are connected from other systems via NFS, any networked file can also be mapped into a process's address space.

Coherence

Whether to share memory or to share data contained in the file, when multiple process map a file simultaneously there may be problems with simultaneous access to data elements. Such processes can cooperate through any of the synchronization mechanisms provided in Solaris 2.x. Because they are very light weight, the most efficient synchronization mechanisms in Solaris 2.x are the threads library ones.

Creating and Using Mappings

`mmap()` establishes a mapping of a named file system object (or part of one) into a process address space. It is the basic memory management function and it is very simple.

- First `open()` the file, then
- `mmap()` it with appropriate access and sharing options
- Away you go.

mmap is prototypes as follows:

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

caddr_t mmap(caddr_t addr, size_t len, int prot, int flags,
          int fildes, off_t off);
```

The mapping established by mmap() replaces any previous mappings for specified address range. The flags MAP_SHARED and MAP_PRIVATE specify the mapping type, and one of them must be specified. MAP_SHARED specifies that writes modify the mapped object. No further operations on the object are needed to make the change. MAP_PRIVATE specifies that an initial write to the mapped area creates a copy of the page and all writes reference the copy. Only modified pages are copied.

A mapping type is retained across a fork(). The file descriptor used in a mmap call need not be kept open after the mapping is established. If it is closed, the mapping remains until the mapping is undone by munmap() or be replacing in with a new mapping. If a mapped file is shortened by a call to truncate, an access to the area of the file that no longer exists causes a SIGBUS signal.

The following code fragment demonstrates a use of this to create a block of scratch storage in a program, at an address that the system chooses.:

```
int fd;
caddr_t result;
if ((fd = open("/dev/zero", O_RDWR)) == -1)
   return ((caddr_t)-1);

result = mmap(0, len, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
(void) close(fd);
```

Other Memory Control Functions

int mlock(caddr_t addr, size_t len) causes the pages in the specified address range to be locked in physical memory. References to locked pages (in this or other processes) do not result in page faults that require an I/O operation. This operation ties up physical resources and can disrupt normal system operation, so, use of mlock() is limited to the superuser. The system lets only a configuration dependent limit of pages be locked in memory. The call to mlock fails if this limit is exceeded.

int munlock(caddr_t addr, size_t len) releases the locks on physical pages. If multiple mlock() calls are made on an address range of a single mapping, a single munlock call is release the locks. However, if different mappings to the same pages are mlocked, the pages are not unlocked until the locks on all the mappings are released. Locks are also released when a mapping is removed, either through being replaced with an mmap operation or removed with munmap. A lock is transferred between pages on the ``copy-on-write' event associated with a MAP_PRIVATE mapping, thus locks on an address range that includes MAP_PRIVATE mappings will be retained transparently along with the copy-on-write redirection (see mmap above for a discussion of this redirection)

int mlockall(int flags) and int munlockall(void) are similar to mlock() and munlock(), but they operate on entire address spaces. mlockall() sets locks on all pages in the address space and munlockall() removes all locks on all pages in the address space, whether established by mlock or mlockall.

int msync(caddr_t addr, size_t len, int flags) causes all modified pages in the specified address range to be flushed to the objects mapped by those addresses. It is similar to fsync() for files.

long sysconf(int name) returns the system dependent size of a memory page. For portability, applications should not embed any constants specifying the size of a page. Note that it is not unusual for page sizes to vary even among implementations of the same instruction set.

int mprotect(caddr_t addr, size_t len, int prot) assigns the specified protection to all pages in the specified address range. The protection cannot exceed the permissions allowed on the underlying object.

int brk(void *endds) and void *sbrk(int incr) are called to add storage to the data segment of a process. A process can manipulate this area by calling brk() and sbrk(). brk() sets the system idea of the lowest data segment location not used by the caller to addr (rounded up to the next multiple of the system page size). sbrk() adds incr bytes to the caller data space and returns a pointer to the start of the new data area.

Some further example shared memory programs

The following suite of programs can be used to investigate interactively a variety of shared ideas (see exercises below).

The semaphore **must** be initialised with the shmget.c program. The effects of controlling shared memory and accessing can be investigated with shmctl.c and shmop.c respectively.

shmget.c:Sample Program to Illustrate shmget()

```
/*
 * shmget.c: Illustrate the shmget() function.
 *
 * This is a simple exerciser of the shmget() function. It
prompts
 * for the arguments, makes the call, and reports the results.
 */

#include <stdio.h>
```

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

extern void    exit();
extern void    perror();

main()
{
 key_t  key;   /* key to be passed to shmget() */
 int  shmflg;   /* shmflg to be passed to shmget() */
 int  shmid;   /* return value from shmget() */
 int  size;   /* size to be passed to shmget() */

 (void) fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
    "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");

 /* Get the key. */
 (void) fprintf(stderr, "IPC_PRIVATE == %#lx\n", IPC_PRIVATE);
 (void) fprintf(stderr, "Enter key: ");
 (void) scanf("%li", &key);

 /* Get the size of the segment. */
 (void) fprintf(stderr, "Enter size: ");
 (void) scanf("%i", &size);

 /* Get the shmflg value. */
 (void) fprintf(stderr,
    "Expected flags for the shmflg argument are:\n");
 (void) fprintf(stderr, "\tIPC_CREAT = \t%#8.8o\n",
IPC_CREAT);
 (void) fprintf(stderr, "\tIPC_EXCL = \t%#8.8o\n", IPC_EXCL);
 (void) fprintf(stderr, "\towner read =\t%#8.8o\n", 0400);
 (void) fprintf(stderr, "\towner write =\t%#8.8o\n", 0200);
 (void) fprintf(stderr, "\tgroup read =\t%#8.8o\n", 040);
 (void) fprintf(stderr, "\tgroup write =\t%#8.8o\n", 020);
 (void) fprintf(stderr, "\tother read =\t%#8.8o\n", 04);
 (void) fprintf(stderr, "\tother write =\t%#8.8o\n", 02);
 (void) fprintf(stderr, "Enter shmflg: ");
 (void) scanf("%i", &shmflg);

 /* Make the call and report the results. */
 (void) fprintf(stderr,
     "shmget: Calling shmget(%#lx, %d, %#o)\n",
     key, size, shmflg);
 if ((shmid = shmget (key, size, shmflg)) == -1) {
  perror("shmget: shmget failed");
  exit(1);
 } else {
  (void) fprintf(stderr,
     "shmget: shmget returned %d\n", shmid);
  exit(0);
 }
}
```

`shmctl.c`: Sample Program to Illustrate `shmctl()`

```
/*
 * shmctl.c: Illustrate the shmctl() function.
 *
 * This is a simple exerciser of the shmctl() function. It lets you
 * to perform one control operation on one shared memory segment.
 * (Some operations are done for the user whether requested or
not.
 * It gives up immediately if any control operation fails. Be
careful
 * not to set permissions to preclude read permission; you won't
be
 *able to reset the permissions with this code if you do.)
*/

#include   <stdio.h>
#include   <sys/types.h>
#include   <sys/ipc.h>
#include   <sys/shm.h>
#include   <time.h>
static void   do_shmctl();
extern void   exit();
extern void   perror();

main()
{
 int  cmd;  /* command code for shmctl() */
 int  shmid;  /* segment ID */
 struct shmid_ds  shmid_ds;     /* shared memory data structure to
         hold results */

 (void) fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
```

```
    "\t0x... is interpreted as hexadecimal,\n");
(void) fprintf(stderr, "\t0... is interpreted as octal,\n");
(void) fprintf(stderr, "\totherwise, decimal.\n");

/* Get shmid and cmd. */
(void) fprintf(stderr,
    "Enter the shmid for the desired segment: ");
(void) scanf("%i", &shmid);
(void) fprintf(stderr, "Valid shmctl cmd values are:\n");
(void) fprintf(stderr, "\tIPC_RMID =\t%d\n", IPC_RMID);
(void) fprintf(stderr, "\tIPC_SET =\t%d\n", IPC_SET);
(void) fprintf(stderr, "\tIPC_STAT =\t%d\n", IPC_STAT);
(void) fprintf(stderr, "\tSHM_LOCK =\t%d\n", SHM_LOCK);
(void) fprintf(stderr, "\tSHM_UNLOCK =\t%d\n", SHM_UNLOCK);
(void) fprintf(stderr, "Enter the desired cmd value: ");
(void) scanf("%i", &cmd);

switch (cmd) {
 case IPC_STAT:
  /* Get shared memory segment status. */
  break;
 case IPC_SET:
  /* Set owner UID and GID and permissions. */
  /* Get and print current values. */
  do_shmctl(shmid, IPC_STAT, &shmid_ds);
  /* Set UID, GID, and permissions to be loaded. */
  (void) fprintf(stderr, "\nEnter shm_perm.uid: ");
  (void) scanf("%hi", &shmid_ds.shm_perm.uid);
  (void) fprintf(stderr, "Enter shm_perm.gid: ");
  (void) scanf("%hi", &shmid_ds.shm_perm.gid);
  (void) fprintf(stderr,
   "Note: Keep read permission for yourself.\n");
  (void) fprintf(stderr, "Enter shm_perm.mode: ");
  (void) scanf("%hi", &shmid_ds.shm_perm.mode);
  break;
 case IPC_RMID:
  /* Remove the segment when the last attach point is
     detached. */
  break;
 case SHM_LOCK:
  /* Lock the shared memory segment. */
  break;
 case SHM_UNLOCK:
  /* Unlock the shared memory segment. */
  break;
 default:
  /* Unknown command will be passed to shmctl. */
  break;
 }
 do_shmctl(shmid, cmd, &shmid_ds);
 exit(0);
}

/*
 * Display the arguments being passed to shmctl(), call shmctl(),
 * and report the results. If shmctl() fails, do not return; this
 * example doesn't deal with errors, it just reports them.
 */
static void
do_shmctl(shmid, cmd, buf)
int   shmid,   /* attach point */
   cmd;   /* command code */
struct shmid_ds   *buf;   /* pointer to shared memory data structure */
{
 register int    rtrn;  /* hold area */
 (void) fprintf(stderr, "shmctl: Calling shmctl(%d, %d,
buf)\n",
  shmid, cmd);
 if (cmd == IPC_SET) {
  (void) fprintf(stderr, "\tbuf->shm_perm.uid == %d\n",
     buf->shm_perm.uid);
  (void) fprintf(stderr, "\tbuf->shm_perm.gid == %d\n",
     buf->shm_perm.gid);
  (void) fprintf(stderr, "\tbuf->shm_perm.mode == %#o\n",
     buf->shm_perm.mode);
 }
 if ((rtrn = shmctl(shmid, cmd, buf)) == -1) {
  perror("shmctl: shmctl failed");
  exit(1);
 } else {
  (void) fprintf(stderr,
      "shmctl: shmctl returned %d\n", rtrn);
 }
 if (cmd != IPC_STAT && cmd != IPC_SET)
  return;

 /* Print the current status. */
 (void) fprintf(stderr, "\nCurrent status:\n");
 (void) fprintf(stderr, "\tshm_perm.uid = %d\n",
      buf->shm_perm.uid);
 (void) fprintf(stderr, "\tshm_perm.gid = %d\n",
      buf->shm_perm.gid);
 (void) fprintf(stderr, "\tshm_perm.cuid = %d\n",
      buf->shm_perm.cuid);
```

```
 (void) fprintf(stderr, "\tshm_perm.cgid = %d\n",
      buf->shm_perm.cgid);
 (void) fprintf(stderr, "\tshm_perm.mode = %#o\n",
      buf->shm_perm.mode);
 (void) fprintf(stderr, "\tshm_perm.key = %#x\n",
      buf->shm_perm.key);
 (void) fprintf(stderr, "\tshm_segsz = %d\n", buf->shm_segsz);
 (void) fprintf(stderr, "\tshm_lpid = %d\n", buf->shm_lpid);
 (void) fprintf(stderr, "\tshm_cpid = %d\n", buf->shm_cpid);
 (void) fprintf(stderr, "\tshm_nattch = %d\n", buf->shm_nattch);
 (void) fprintf(stderr, "\tshm_atime = %s",
   buf->shm_atime ? ctime(&buf->shm_atime) : "Not Set\n");
 (void) fprintf(stderr, "\tshm_dtime = %s",
   buf->shm_dtime ? ctime(&buf->shm_dtime) : "Not Set\n");
 (void) fprintf(stderr, "\tshm_ctime = %s",
      ctime(&buf->shm_ctime));
}
```

`shmop.c`: Sample Program to Illustrate `shmat()` and `shmdt()`

```
/*
 * shmop.c: Illustrate the shmat() and shmdt() functions.
 *
 * This is a simple exerciser for the shmat() and shmdt() system
 * calls. It allows you to attach and detach segments and to
 * write strings into and read strings from attached segments.
 */

#include   <stdio.h>
#include   <setjmp.h>
#include   <signal.h>
#include   <sys/types.h>
#include   <sys/ipc.h>
#include   <sys/shm.h>

#define   MAXnap  4 /* Maximum number of concurrent attaches. */

static   ask();
static void  catcher();
extern void  exit();
static   good_addr();
extern void  perror();
extern char  *shmat();

static struct state     { /* Internal record of currently attached
segments. */
 int  shmid;   /* shmid of attached segment */
 char  *shmaddr;   /* attach point */
 int  shmflg;   /* flags used on attach */
} ap[MAXnap];     /* State of current attached segments. */

static int    nap;  /* Number of currently attached segments. */
static jmp_buf   segvbuf;   /* Process state save area for SIGSEGV
         catching. */

main()
{
 register int    action;   /* action to be performed */
 char    *addr;   /* address work area */
 register int    i;   /* work area */
 register struct state   *p;    /* ptr to current state entry */
 void   (*savefunc)();  /* SIGSEGV state hold area */
 (void) fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
  "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");
 while (action = ask()) {
  if (nap) {
   (void) fprintf(stderr,
      "\nCurrently attached segment(s):\n");
   (void) fprintf(stderr, " shmid address\n");
   (void) fprintf(stderr, "------ ----------\n");
   p = &ap[nap];
   while (p-- != ap) {
    (void) fprintf(stderr, "%6d", p->shmid);
    (void) fprintf(stderr, "%#11x", p->shmaddr);
    (void) fprintf(stderr, " Read%s\n",
     (p->shmflg & SHM_RDONLY) ?
     "-Only" : "/Write");
   }
  } else
   (void) fprintf(stderr,
    "\nNo segments are currently attached.\n");
  switch (action) {
  case 1:   /* Shmat requested. */
   /* Verify that there is space for another attach. */
   if (nap == MAXnap) {
    (void) fprintf(stderr, "%s %d %s\n",
       "This simple example will only allow",
       MAXnap, "attached segments.");
    break;
   }
```

```
p = &ap[nap++];
/* Get the arguments, make the call, report the
 results, and update the current state array. */
(void) fprintf(stderr,
 "Enter shmid of segment to attach: ");
(void) scanf("%i", &p->shmid);

(void) fprintf(stderr, "Enter shmaddr: ");
(void) scanf("%i", &p->shmaddr);
(void) fprintf(stderr,
 "Meaningful shmflg values are:\n");
(void) fprintf(stderr, "\tSHM_RDONLY = \t%#8.8o\n",
 SHM_RDONLY);
(void) fprintf(stderr, "\tSHM_RND = \t%#8.8o\n",
 SHM_RND);
(void) fprintf(stderr, "Enter shmflg value: ");
(void) scanf("%i", &p->shmflg);

(void) fprintf(stderr,
 "shmop: Calling shmat(%d, %#x, %#o)\n",
 p->shmid, p->shmaddr, p->shmflg);
p->shmaddr = shmat(p->shmid, p->shmaddr, p->shmflg);
if(p->shmaddr == (char *)-1) {
 perror("shmop: shmat failed");
 nap--;
} else {
 (void) fprintf(stderr,
  "shmop: shmat returned %#8.8x\n",
  p->shmaddr);
}
 break;

case 2:   /* Shmdt requested. */
 /* Get the address, make the call, report the results,
  and make the internal state match. */
 (void) fprintf(stderr,
  "Enter detach shmaddr: ");
 (void) scanf("%i", &addr);

 i = shmdt(addr);
 if(i == -1) {
  perror("shmop: shmdt failed");
 } else {
  (void) fprintf(stderr,
   "shmop: shmdt returned %d\n", i);
  for (p = ap, i = nap; i--; p++) {
   if (p->shmaddr == addr)
    *p = ap[--nap];
  }
 }
 break;
case 3: /* Read from segment requested. */
 if (nap == 0)
  break;

 (void) fprintf(stderr, "Enter address of an %s",
  "attached segment: ");
 (void) scanf("%i", &addr);

 if (good_addr(addr))
  (void) fprintf(stderr, "String @ %#x is `%s'\n",
   addr, addr);
 break;

case 4: /* Write to segment requested. */
 if (nap == 0)
 break;

(void) fprintf(stderr, "Enter address of an %s",
 "attached segment: ");
(void) scanf("%i", &addr);

/* Set up SIGSEGV catch routine to trap attempts to
 write into a read-only attached segment. */
savefunc = signal(SIGSEGV, catcher);

if (setjmp(segvbuf)) {
 (void) fprintf(stderr, "shmop: %s: %s\n",
  "SIGSEGV signal caught",
  "Write aborted.");
} else {
 if (good_addr(addr)) {
  (void) fflush(stdin);
  (void) fprintf(stderr, "%s %s %#x:\n",
   "Enter one line to be copied",
   "to shared segment attached @",
   addr);
  (void) gets(addr);
 }
}
(void) fflush(stdin);

/* Restore SIGSEGV to previous condition. */
(void) signal(SIGSEGV, savefunc);
break;
```

```
  }
 }
 exit(0);
 /*NOTREACHED*/
}
/*
** Ask for next action.
*/
static
ask()
{
 int  response;   /* user response */
 do {
   (void) fprintf(stderr, "Your options are:\n");
   (void) fprintf(stderr, "\t^D = exit\n");
   (void) fprintf(stderr, "\t 0 = exit\n");
   (void) fprintf(stderr, "\t 1 = shmat\n");
   (void) fprintf(stderr, "\t 2 = shmdt\n");
   (void) fprintf(stderr, "\t 3 = read from segment\n");
   (void) fprintf(stderr, "\t 4 = write to segment\n");
   (void) fprintf(stderr,
    "Enter the number corresponding to your choice: ");

   /* Preset response so "^D" will be interpreted as exit. */
   response = 0;
   (void) scanf("%i", &response);
 } while (response < 0 || response > 4);
 return (response);
}
/*
** Catch signal caused by attempt to write into shared memory
segment
** attached with SHM_RDONLY flag set.
*/
/*ARGSUSED*/
static void
catcher(sig)
{
 longjmp(segvbuf, 1);
 /*NOTREACHED*/
}
/*
** Verify that given address is the address of an attached
segment.
** Return 1 if address is valid; 0 if not.
*/
static
good_addr(address)
char *address;
{
 register struct state          *p;   /* ptr to state of attached
segment */

 for (p = ap; p != &ap[nap]; p++)
   if (p->shmaddr == address)
    return(1);
 return(0);
}
```

Exercises

Exercise 12771

Write 2 programs that will communicate via shared memory and semaphores. Data will be exchanged via memory and semaphores will be used to synchronise and notify each process when operations such as memory loaded and memory read have been performed.

Exercise 12772

Compile the programs `shmget.c`, `shmctl.c` and `shmop.c` and then

- investigate and understand fully the operations of the flags (access, creation ***etc.*** permissions) you can set interactively in the programs.
- Use the prgrams to:
  - Exchange data between two processe running as `shmop.c`.
  - Inquire about the state of shared memory with `shmctl.c`.
  - Use `semctl.c` to lock a shared memory segment.
  - Use `semctl.c` to delete a shared memory segment.

Exercise 12773

Write 2 programs that will communicate via mapped memory.

IPC:Sockets

Sockets provide point-to-point, two-way communication between two processes. Sockets are very versatile and are a basic component of interprocess and intersystem communication. A socket is an endpoint of communication to which a name can be bound. It has a type and one or more associated processes.

Sockets exist in communication domains. A socket domain is an abstraction that provides an addressing structure and a set of protocols. Sockets connect only with sockets in the same domain. Twenty three socket domains are identified (see `<sys/socket.h>`), of which only the UNIX and Internet domains are normally used Solaris 2.x Sockets can be used to communicate between processes on a single system, like other forms of IPC.

The UNIX domain provides a socket address space on a single system. UNIX domain sockets are named with UNIX paths. Sockets can also be used to communicate between processes on different systems. The socket address space between connected systems is called the Internet domain.

Internet domain communication uses the TCP/IP internet protocol suite.

Socket types define the communication properties visible to the application. Processes communicate only between sockets of the same type. There are five types of socket.

A stream socket
-- provides two-way, sequenced, reliable, and unduplicated flow of data with no record boundaries. A stream operates much like a telephone conversation. The socket type is `SOCK_STREAM`, which, in the Internet domain, uses Transmission Control Protocol (TCP).

A datagram socket
-- supports a two-way flow of messages. A on a datagram socket may receive messages in a different order from the sequence in which the messages were sent. Record boundaries in the data are preserved. Datagram sockets operate much like passing letters back and forth in the mail. The socket type is `SOCK_DGRAM`, which, in the Internet domain, uses User Datagram Protocol (UDP).

A sequential packet socket
-- provides a two-way, sequenced, reliable, connection, for datagrams of a fixed maximum length. The socket type is `SOCK_SEQPACKET`. No protocol for this type has been implemented for any protocol family.

A raw socket
provides access to the underlying communication protocols.

These sockets are usually datagram oriented, but their exact characteristics depend on the interface provided by the protocol.

Socket Creation and Naming

`int socket(int domain, int type, int protocol)` is called to create a socket in the specified domain and of the specified type. If a `protocol` is not specified, the system defaults to a protocol that supports the specified socket type. The socket handle (a descriptor) is returned. A remote process has no way to identify a socket until an address is bound to it. Communicating processes connect through addresses. In the UNIX domain, a connection is usually composed of one or two path names. In the Internet domain, a connection is composed of local and remote addresses and local and remote ports. In most domains, connections must be unique.

`int bind(int s, const struct sockaddr *name, int namelen)` is called to bind a path or internet address to a socket. There are three different ways to call `bind()`, depending on the domain of the socket.

- For UNIX domain sockets with paths containing 14, or fewer characters, you can:
- `#include <sys/socket.h>`
- `...`
- `bind (sd, (struct sockaddr *) &addr, length);`
- If the path of a UNIX domain socket requires more characters, use:
- `#include <sys/un.h>`
- `...`
- `bind (sd, (struct sockaddr_un *) &addr, length);`
- For Internet domain sockets, use
- `#include <netinet/in.h>`
- `...`
- `bind (sd, (struct sockaddr_in *) &addr, length);`

In the UNIX domain, binding a name creates a named socket in the file system. Use `unlink()` or `rm ()` to remove the socket.

Connecting Stream Sockets

Connecting sockets is usually not symmetric. One process usually acts as a server and the other process is the client. The server binds its socket to a previously agreed path or address. It then blocks on the socket. For a `SOCK_STREAM` socket, the server calls `int listen(int s, int backlog)` , which specifies how many connection requests can be queued. A client initiates a connection to the server's socket by a call to `int connect(int s, struct sockaddr *name, int namelen)` . A UNIX domain call is like this:

```
struct sockaddr_un server;
...
connect (sd, (struct sockaddr_un *)&server, length);
```

while an Internet domain call would be:

```
struct sockaddr_in;
...
connect (sd, (struct sockaddr_in *)&server, length);
```

If the client's socket is unbound at the time of the connect call, the system automatically selects and binds a name to the socket. For a `SOCK_STREAM` socket, the server calls accept(3N) to complete the connection.

`int accept(int s, struct sockaddr *addr, int *addrlen)` returns a new socket descriptor which is valid only for the particular connection. A server can have multiple `SOCK_STREAM` connections active at one time.

Stream Data Transfer and Closing

Several functions to send and receive data from a `SOCK_STREAM` socket. These are `write()`, `read()`, `int send(int s, const char *msg, int len, int flags)`, and `int recv(int s, char *buf, int len, int flags)`. `send()` and `recv()` are very similar to `read()` and `write()`, but have some additional operational `flags`.

The flags parameter is formed from the bitwise OR of zero or more of the following:

`MSG_OOB`
: -- Send "out-of-band" data on sockets that support this notion. The underlying protocol must also support "out-of-band" data. Only `SOCK_STREAM` sockets created in the `AF_INET` address family support out-of-band data.

`MSG_DONTROUTE`
: -- The `SO_DONTROUTE` option is turned on for the duration of the operation. It is used only by diagnostic or routing pro- grams.

`MSG_PEEK`
: -- "Peek" at the data present on the socket; the data is returned, but not consumed, so that a subsequent receive operation will see the same data.

A `SOCK_STREAM` socket is discarded by calling `close()`.

Datagram sockets

A datagram socket does not require that a connection be established. Each message carries the destination address. If a particular local address is needed, a call to `bind()` must precede any data transfer. Data is sent through calls to `sendto()` or `sendmsg()`. The `sendto()` call is like a `send()` call with the destination address also specified. To receive datagram socket messages, call `recvfrom()` or `recvmsg()`. While `recv()` requires one buffer for the arriving data, `recvfrom()` requires two buffers, one for the incoming message and another to receive the source address.

Datagram sockets can also use `connect()` to connect the socket to a specified destination socket. When this is done, `send()` and `recv()` are used to send and receive data.

`accept()` and `listen()` are not used with datagram sockets.

Socket Options

Sockets have a number of options that can be fetched with `getsockopt()` and set with `setsockopt()`. These functions can be used at the native socket level (`level = SOL_SOCKET`), in which case the socket option name must be specified. To manipulate options at any other level the protocol number of the desired protocol controlling the option of interest must be specified (see `getprotoent()` in `getprotobyname()`).

Example Socket Programs:`socket_server.c,socket_client`

These two programs show how you can establish a socket connection using the above functions.

`socket_server.c`

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>

#define NSTRS       3           /* no. of strings  */
#define ADDRESS     "mysocket"  /* addr to connect */

/*
 * Strings we send to the client.
 */
char *strs[NSTRS] = {
    "This is the first string from the server.\n",
    "This is the second string from the server.\n",
    "This is the third string from the server.\n"
};

main()
{
```

```
char c;
FILE *fp;
int fromlen;
register int i, s, ns, len;
struct sockaddr_un saun, fsaun;

/*
 * Get a socket to work with.  This socket will
 * be in the UNIX domain, and will be a
 * stream socket.
 */
if ((s = socket(AF_UNIX, SOCK_STREAM, 0)) < 0) {
    perror("server: socket");
    exit(1);
}

/*
 * Create the address we will be binding to.
 */
saun.sun_family = AF_UNIX;
strcpy(saun.sun_path, ADDRESS);

/*
 * Try to bind the address to the socket.  We
 * unlink the name first so that the bind won't
 * fail.
 *
 * The third argument indicates the "length" of
 * the structure, not just the length of the
 * socket name.
 */
unlink(ADDRESS);
len = sizeof(saun.sun_family) + strlen(saun.sun_path);

if (bind(s, &saun, len) < 0) {
    perror("server: bind");
    exit(1);
}

/*
 * Listen on the socket.
 */
if (listen(s, 5) < 0) {
    perror("server: listen");
    exit(1);
}

/*
 * Accept connections.  When we accept one, ns
 * will be connected to the client.  fsaun will
 * contain the address of the client.
 */
if ((ns = accept(s, &fsaun, &fromlen)) < 0) {
    perror("server: accept");
    exit(1);
}

/*
 * We'll use stdio for reading the socket.
 */
fp = fdopen(ns, "r");

/*
 * First we send some strings to the client.
 */
for (i = 0; i < NSTRS; i++)
        send(ns, strs[i], strlen(strs[i]), 0);

    /*
     * Then we read some strings from the client and
     * print them out.
     */
    for (i = 0; i < NSTRS; i++) {
        while ((c = fgetc(fp)) != EOF) {
            putchar(c);

            if (c == '\n')
                break;
        }
    }

    /*
     * We can simply use close() to terminate the
     * connection, since we're done with both sides.
     */
    close(s);

    exit(0);
}
```

socket_client.c

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

```
#include <sys/un.h>
#include <stdio.h>

#define NSTRS       3           /* no. of strings  */
#define ADDRESS     "mysocket"  /* addr to connect */

/*
 * Strings we send to the server.
 */
char *strs[NSTRS] = {
    "This is the first string from the client.\n",
    "This is the second string from the client.\n",
    "This is the third string from the client.\n"
};

main()
{
    char c;
    FILE *fp;
    register int i, s, len;
    struct sockaddr_un saun;

    /*
     * Get a socket to work with.  This socket will
     * be in the UNIX domain, and will be a
     * stream socket.
     */
    if ((s = socket(AF_UNIX, SOCK_STREAM, 0)) < 0) {
        perror("client: socket");
        exit(1);
    }

    /*
     * Create the address we will be connecting to.
     */
    saun.sun_family = AF_UNIX;
    strcpy(saun.sun_path, ADDRESS);

    /*
     * Try to connect to the address.  For this to
     * succeed, the server must already have bound
     * this address, and must have issued a listen()
     * request.
     *
     * The third argument indicates the "length" of
     * the structure, not just the length of the
     * socket name.
     */
    len = sizeof(saun.sun_family) + strlen(saun.sun_path);

    if (connect(s, &saun, len) < 0) {
        perror("client: connect");
        exit(1);
    }

    /*
     * We'll use stdio for reading
     * the socket.
     */
    fp = fdopen(s, "r");

    /*
     * First we read some strings from the server
     * and print them out.
     */
    for (i = 0; i < NSTRS; i++) {
        while ((c = fgetc(fp)) != EOF) {
            putchar(c);

            if (c == '\n')
                break;
        }
    }

    /*
     * Now we send some strings to the server.
     */
    for (i = 0; i < NSTRS; i++)
        send(s, strs[i], strlen(strs[i]), 0);

    /*
     * We can simply use close() to terminate the
     * connection, since we're done with both sides.
     */
    close(s);

    exit(0);
}
```

Exercises

Exercise 12776

Configure the above `socket_server.c` and `socket_client.c` programs for you system and compile and run them. You will need to set up socket `ADDRESS` definition.

Threads: Basic Theory and Libraries

This chapter examines aspects of threads and multiprocessing (and multithreading). We will firts study a little theory of threads and also look at how threading can be effectively used to make programs more efficient. The C thread libraries will then be introduced. The following chapters will look at further thead issues sucj a synchronisation and practical examples.

Processes and Threads

We can think of a **thread** as basically a ***lightweight*** process. In order to understand this let us consider the two main characteristics of a process:

Unit of resource ownership

-- A process is allocated:

- a virtual address space to hold the process image
- control of some resources (files, I/O devices...)

Unit of dispatching

- A process is an execution path through one or more programs:

- execution may be interleaved with other processes
- the process has an execution state and a dispatching priority

If we treat these two characteristics as being independent (as does modern OS theory):

- The unit of resource ownership is usually referred to as a **process** or task. This Processes have:
  - a virtual address space which holds the process image.
  - protected access to processors, other processes, files, and I/O resources.
- The unit of dispatching is usually referred to a **thread** or a lightweight process. Thus a thread:
  - Has an execution state (running, ready, etc.)
  - Saves thread context when not running
  - Has an execution stack and some per-thread static storage for local variables
  - Has access to the memory address space and resources of its process
- all threads of a process share this when one thread alters a (non-private) memory item, all other threads (of the process) sees that a file open with one thread, is available to others

Benefits of Threads vs Processes

If implemented correctly then threads have some advantages of (multi) processes, They take:

- Less time to create a new thread than a process, because the newly created thread uses the current process address space.
- Less time to terminate a thread than a process.
- Less time to switch between two threads within the same process, partly because the newly created thread uses the current process address space.
- Less communication overheads -- communicating between the threads of one process is simple because the threads share everything: address space, in particular. So, data produced by one thread is immediately available to all the other threads.

Multithreading vs. Single threading

Just a we can multiple processes running on some systems we can have multiple threads running:

Single threading

-- when the OS does not recognize the concept of thread

Multithreading

-- when the OS supports multiple threads of execution within a single process

Figure 28.1 shows a variety of models for threads and processes.

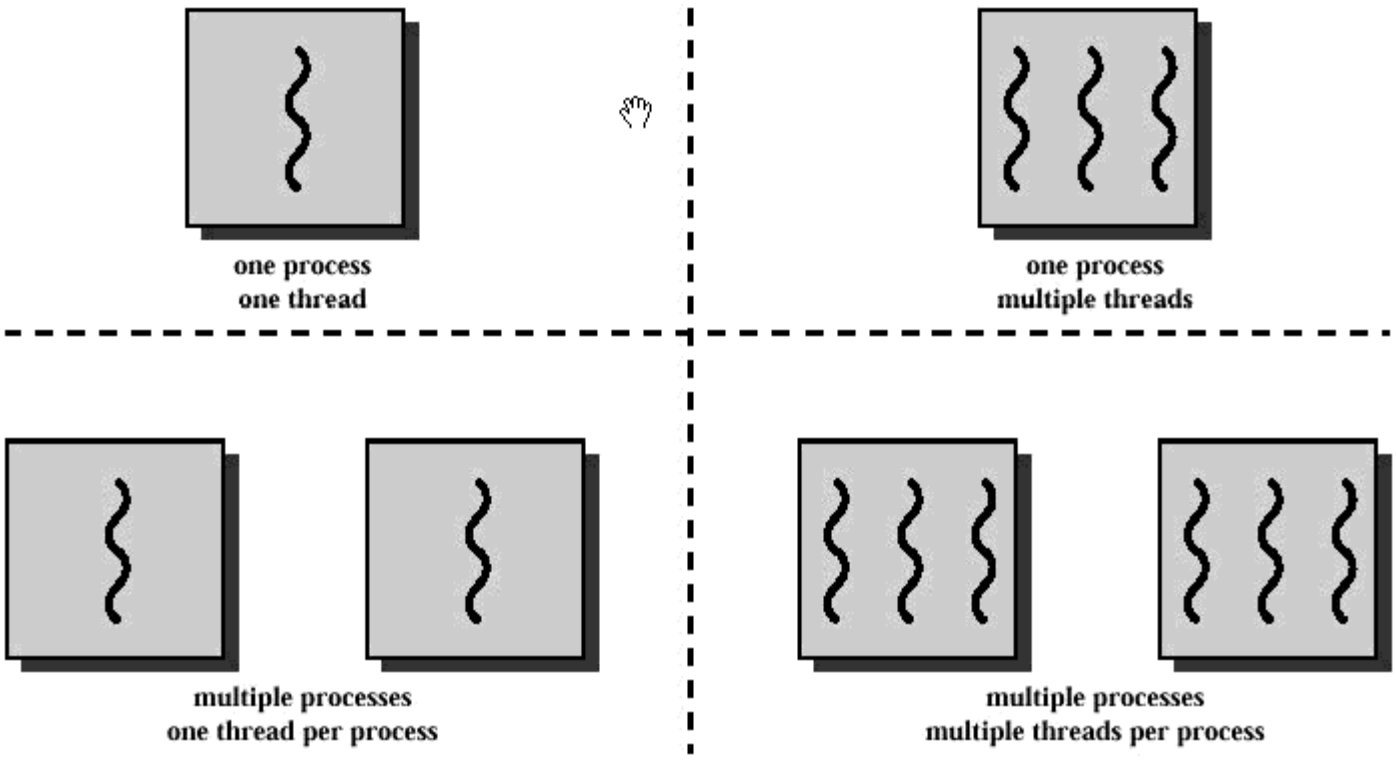


Fig. 28.1 Threads and Processes Some example popular OSs and their thread support is:

MS-DOS
-- support a single user process and a single thread

UNIX
-- supports multiple user processes but only supports one thread per process

Solaris
-- supports multiple threads

Multithreading your code can have many benefits:

- Improve application responsiveness -- Any program in which many activities are not dependent upon each other can be redesigned so that each activity is defined as a thread. For example, the user of a multithreaded GUI does not have to wait for one activity to complete before starting another.
- Use multiprocessors more efficiently -- Typically, applications that express concurrency requirements with threads need not take into account the number of available processors. The performance of the application improves transparently with additional processors. Numerical algorithms and applications with a high degree of parallelism, such as matrix multiplications, can run much faster when implemented with threads on a multiprocessor.
- Improve program structure -- Many programs are more efficiently structured as multiple independent or semi-independent units of execution instead of as a single, monolithic thread. Multithreaded programs can be more adaptive to variations in user demands than single threaded programs.
- Use fewer system resources -- Programs that use two or more processes that access common data through shared memory are applying more than one thread of control. However, each process has a full address space and operating systems state. The cost of creating and maintaining this large amount of state information makes each process much more expensive than a thread in both time and space. In addition, the inherent separation between processes can require a major effort by the programmer to communicate between the threads in different processes, or to synchronize their actions.

Figure 28.2 illustrates different process models and thread control in a single thread and multithreaded application.

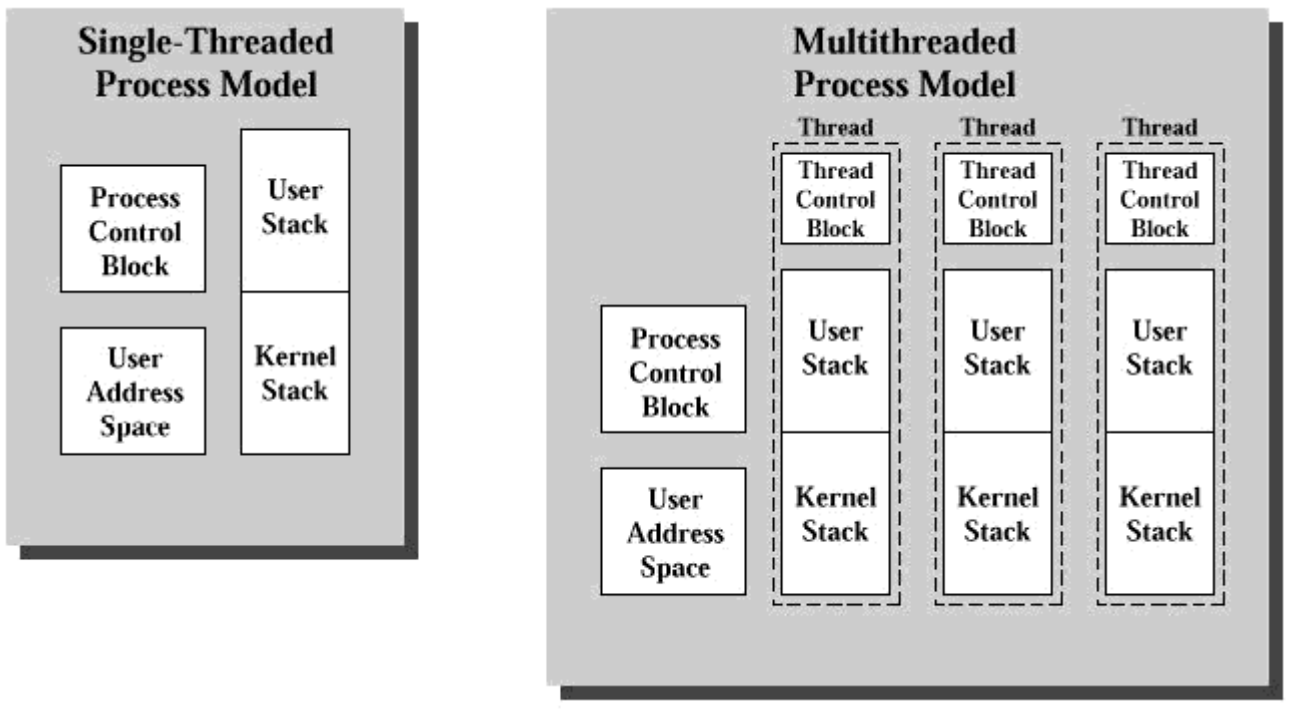


Fig. 28.2 Single and Multi- Thread Applicatiions

Some Example applications of threads

:

Example : A file server on a LAN

- It needs to handle several file requests over a short period
- Hence more efficient to create (and destroy) a single thread for each request
- Multiple threads can possibly be executing simultaneously on different processors

Example 2: Matrix Multiplication

Matrix Multilication essentially involves taking the rows of one matrix and multiplying and adding corresponding columns in a second matrix ***i.e***:

$$\begin{pmatrix} a11 & a12 & a13 \\ a21 & a22 & a23 \\ a31 & a32 & a33 \end{pmatrix} \begin{pmatrix} b11 & b12 & b13 \\ b21 & b22 & b23 \\ b31 & b32 & b33 \end{pmatrix} =$$

$$\begin{pmatrix} a11.b11 + a12.b21 + a13.b31 & a11.b12 + a12.b22 + a13.b32 & a11.b13 + a12.b23 + a13.b33 \\ a21.b11 + a22.b21 + a23.b31 & a21.b12 + a22.b22 + a23.b32 & a21.b13 + a22.b23 + a23.b33 \\ a31.b11 + a32.b21 + a33.b31 & a31.b12 + a32.b22 + a33.b32 & a31.b13 + a32.b23 + a33.b33 \end{pmatrix}$$

Fig. 28.3 Matrix Multiplication (3x3 example) Note that each ***element*** of the resultant matrix can be computed independently, that is to say by a different thread.

We will develop a C++ example program for matrix multiplication later (see Chapter ❑).

Thread Levels

There are two broad categories of thread implementation:

- User-Level Threads -- Thread Libraries.
- Kernel-level Threads -- System Calls.

There are merits to both, in fact some OSs allow access to both levels (***e.g.*** Solaris).

User-Level Threads (ULT)

In this level, the kernel is not aware of the existence of threads -- All thread management is done by the application by using a thread library. Thread switching does not require kernel mode privileges (no mode switch) and scheduling is application specific

Kernel activity for ULTs:

- The kernel is not aware of thread activity but it is still managing process activity
- When a thread makes a system call, the whole process will be blocked but for the thread library that thread is still in the running state
- So thread states are independent of process states

Advantages and inconveniences of ULT

Advantages:

- Thread switching does not involve the kernel -- no mode switching
- Scheduling can be application specific -- choose the best algorithm.
- ULTs can run on any OS -- Only needs a thread library

Disadvantages:

- Most system calls are blocking and the kernel blocks processes -- So all threads within the process will be blocked
- The kernel can only assign processes to processors -- Two threads within the same process cannot run simultaneously on two processors

Kernel-Level Threads (KLT)

In this level, All thread management is done by kernel No thread library but an API (system calls) to the kernel thread facility exists. The kernel maintains context information for the process and the threads, switching between threads requires the kernel Scheduling is performed on a thread basis.

Advantages and inconveniences of KLT

Advantages

- the kernel can simultaneously schedule many threads of the same process on many processors blocking is done on a thread level
- kernel routines can be multithreaded

Disadvantages:

- thread switching within the same process involves the kernel, ***e.g*** if we have 2 mode switches per thread switch this results in a significant slow down.

Combined ULT/KLT Approaches

Idea is to combine the best of both approaches

Solaris is an example of an OS that combines both ULT and KLT (Figure 28.4:

- Thread creation done in the user space
- Bulk of scheduling and synchronization of threads done in the user space
- The programmer may adjust the number of KLTs
- Process includes the user's address space, stack, and process control block
- User-level threads (threads library) invisible to the OS are the interface for application parallelism
- Kernel threads the unit that can be dispatched on a processor
- Lightweight processes (LWP) each LWP supports one or more ULTs and maps to exactly one KLT

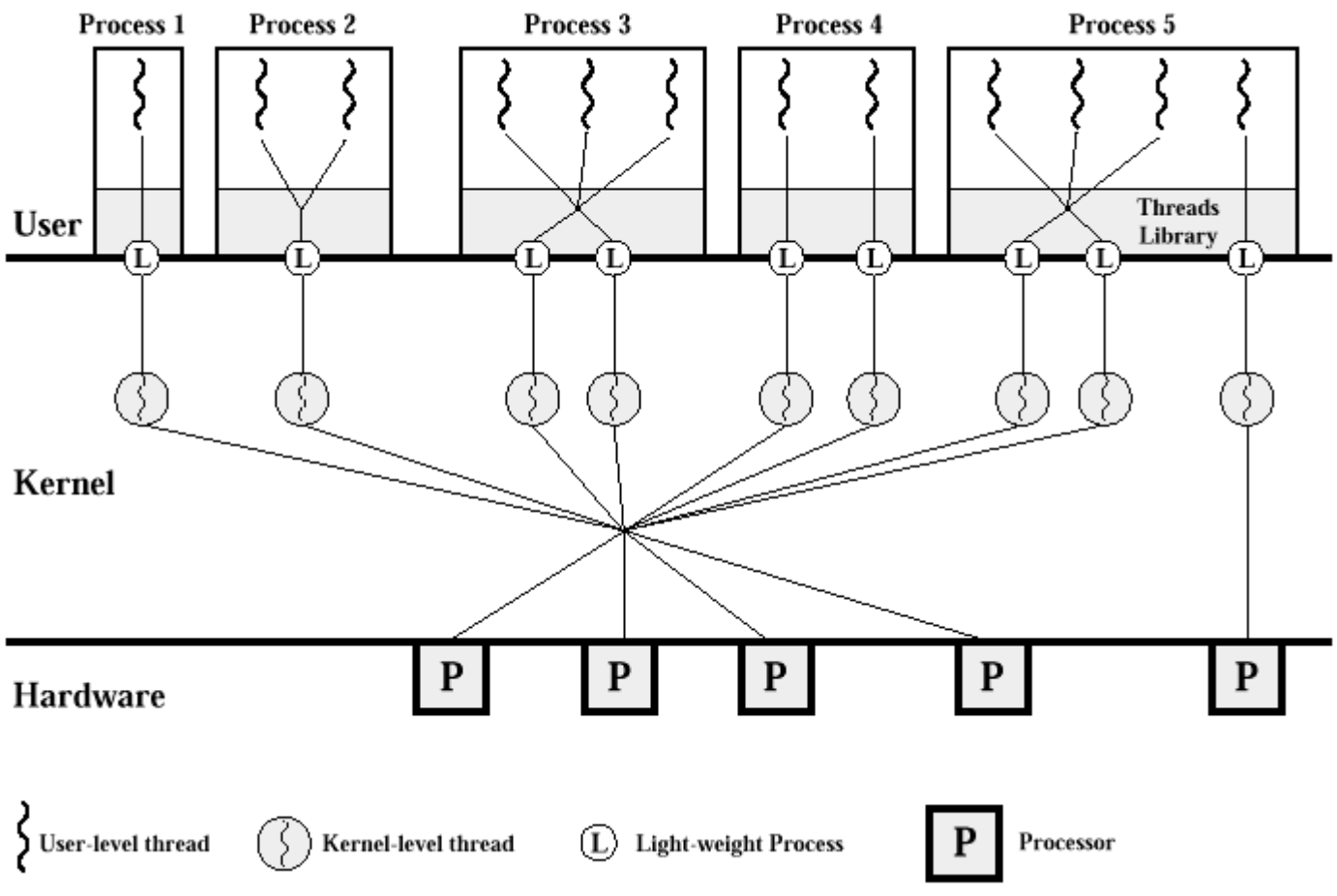


Fig. 28.4 Solaris Thread Implementation

Threads libraries

The interface to multithreading support is through a subroutine library, libpthread for POSIX threads, and libthread for Solaris threads. They both contain code for:

- creating and destroying threads
- passing messages and data between threads
- scheduling thread execution
- saving and restoring thread contexts

The POSIX Threads Library: `libpthread`, <pthread.h>

Creating a (Default) Thread

Use the function `pthread_create()` to add a new thread of control to the current process. It is prototyped by:

```
int pthread_create(pthread\_t *tid, const pthread\_attr\_t *tattr,
void*(*start_routine)(void *), void *arg);
```

When an attribute object is not specified, it is NULL, and the ***default*** thread is created with the following attributes:

- It is unbounded
- It is nondetached
- It has a a default stack and stack size
- It inhetits the parent's priority

You can also create a default attribute object with `pthread_attr_init()` function, and then use this attribute object to create a default thread. See the Section 29.2.

An example call of default thread creation is:

```
#include <pthread.h>
pthread_attr_t tattr;
pthread_t tid;
extern void *start_routine(void *arg);
void *arg;
int ret;
/* default behavior*/
ret = pthread_create(&tid, NULL, start_routine, arg);

/* initialized with default attributes */
ret = pthread_attr_init(&tattr);
/* default behavior specified*/
ret = pthread_create(&tid, &tattr, start_routine, arg);
```

The `pthread_create()` function is called with `attr` having the necessary state behavior. `start_routine` is the function with which the new thread begins execution. When `start_routine` returns, the thread exits with the exit status set to the value returned by `start_routine`.

When `pthread_create` is successful, the ID of the thread created is stored in the location referred to as `tid`.

Creating a thread using a NULL attribute argument has the same effect as using a default attribute; both create a default thread. When tattr is initialized, it acquires the default behavior.

`pthread_create()` returns a zero and exits when it completes successfully. Any other returned value indicates that an error occurred.

Wait for Thread Termination

Use the `pthread_join` function to wait for a thread to terminate. It is prototyped by:

```
int pthread_join(thread_t tid, void **status);
```

An example use of this function is:

```
#include <pthread.h>
pthread_t tid;
int ret;
int status;
/* waiting to join thread "tid" with status */
ret = pthread_join(tid, &status);
/* waiting to join thread "tid" without status */
ret = pthread_join(tid, NULL);
```

The `pthread_join()` function blocks the calling thread until the specified thread terminates. The specified thread must be in the current process and must not be detached. When `status` is not `NULL`, it points to a location that is set to the exit status of the terminated thread when `pthread_join()` returns successfully. Multiple threads cannot wait for the same thread to terminate. If they try to, one thread returns successfully and the others fail with an error of `ESRCH`. After `pthread_join()` returns, any stack storage associated with the thread can be reclaimed by the application.

The `pthread_join()` routine takes two arguments, giving you some flexibility in its use. When you want the caller to wait until a specific thread terminates, supply that thread's ID as the first argument. If you are interested in the exit code of the defunct thread, supply the address of an area to receive it. Remember that `pthread_join()` works only for target threads that are nondetached. When there is no reason to synchronize with the termination of a particular thread, then that thread should be detached. Think of a detached thread as being the thread you use in most instances and reserve nondetached threads for only those situations that require them.

A Simple Threads Example

In this Simple Threads fragment below, one thread executes the procedure at the top, creating a helper thread that executes the procedure fetch, which involves a complicated database lookup and might take some time.

The main thread wants the results of the lookup but has other work to do in the meantime. So it does those other things and then waits for its helper to complete its job by executing `pthread_join()`. An argument, `pbe`, to the new thread is passed as a stack parameter. This can be done here because the main thread waits for the spun-off thread to terminate. In general, though, it is better to `malloc()` storage from the heap instead of passing an address to thread stack storage, which can disappear or be reassigned if the thread terminated.

The source for `thread.c` is as follows:

```
void mainline (...)
```

```
{
struct phonebookentry *pbe;
pthread_attr_t tattr;
pthread_t helper;
int status;
pthread_create(&helper, NULL, fetch, &pbe);
/* do something else for a while */
pthread_join(helper, &status);
/* it's now safe to use result */
}
void fetch(struct phonebookentry *arg)
{
struct phonebookentry *npbe;
/* fetch value from a database */
npbe = search (prog_name)
if (npbe != NULL)
*arg = *npbe;
pthread_exit(0);
}
struct phonebookentry {
char name[64];
char phonenumber[32];
char flags[16];
}
```

Detaching a Thread

The function `pthread_detach()` is an alternative to `pthread_join()` to reclaim storage for a thread that is created with a detachstate attribute set to `PTHREAD_CREATE_JOINABLE`. It is prototyped by:

```
int pthread\_detach(thread\_t tid);
```

A simple example of calling this fucntion to detatch a thread is given by:

```
#include <pthread.h>
pthread_t tid;
int ret;
/* detach thread tid */
ret = pthread_detach(tid);
```

The `pthread_detach()` function is used to indicate to the implementation that storage for the thread `tid` can be reclaimed when the thread terminates. If tid has not terminated, `pthread_detach()` does not cause it to terminate. The effect of multiple `pthread_detach()` calls on the same target thread is unspecified.

`pthread_detach()` returns a zero when it completes successfully. Any other returned value indicates that an error occurred. When any of the following conditions are detected, `pthread_detach()` fails and returns the an error value.

Create a Key for Thread-Specific Data

Single-threaded C programs have two basic classes of data: local data and global data. For multithreaded C programs a third class is added:***thread-specific data (TSD)***. This is very much like global data, except that it is private to a thread.

Thread-specific data is maintained on a per-thread basis. TSD is the only way to define and refer to data that is private to a thread. Each thread-specific data item is associated with a key that is global to all threads in the process. Using the key, a thread can access a pointer `(void *)` that is maintained per-thread.

The function `pthread_keycreate()` is used to allocate a key that is used to identify thread-specific data in a process. The key is global to all threads in the process, and all threads initially have the value `NULL` associated with the key when it is created.

`pthread_keycreate()` is called once for each key before the key is used. There is no implicit synchronization. Once a key has been created, each thread can bind a value to the key. The values are specific to the thread and are maintained for each thread independently. The per-thread binding is deallocated when a thread terminates if the key was created with a destructor function. `pthread_keycreate()` is prototyped by:

```
int pthread_key_create(pthread_key_t *key, void (*destructor) (void *));
```

A simple example use of this function is:

```
#include <pthread.h>
pthread_key_t key;
int ret;
/* key create without destructor */
ret = pthread_key_create(&key, NULL);
/* key create with destructor */
ret = pthread_key_create(&key, destructor);
```

When `pthread_keycreate()` returns successfully, the allocated `key` is stored in the location pointed to by `key`. The caller must ensure that the storage and access to this key are properly synchronized. An optional destructor function, destructor, can be used to free stale storage. When a key has a non-NULL destructor function and the thread has a non-`NULL` value associated with that key, the destructor function is called with the current associated value when the thread exits. The order in which the destructor functions are called is unspecified.

`pthread_keycreate()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, `pthread_keycreate()` fails and returns an error value.

Delete the Thread-Specific Data Key

The function `pthread_keydelete()` is used to destroy an existing thread-specific data key. Any memory associated with the key can be freed because the key has been invalidated and will return an error if ever referenced. (There is no comparable function in Solaris threads.)

`pthread_keydelete()` is prototyped by:

```
int pthread_key_delete(pthread_key_t key);
```

A simple example use of this function is:

```
#include <pthread.h>
pthread_key_t key;
int ret;
/* key previously created */
ret = pthread_key_delete(key);
```

Once a `key` has been deleted, any reference to it with the `pthread_setspecific()` or `pthread_getspecific()` call results in the `EINVAL` error.

It is the responsibility of the programmer to free any thread-specific resources before calling the delete function. This function does not invoke any of the destructors.

`pthread_keydelete()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, `pthread_keycreate()` fails and returns the corresponding value.

Set the Thread-Specific Data Key

The function `pthread_setspecific()` is used to set the thread-specific binding to the specified thread-specific data key. It is prototyped by :

```
int pthread_setspecific(pthread_key_t key, const void *value);
```

A simple example use of this function is:

```
#include <pthread.h>
pthread_key_t key;
void *value;
int ret;

/* key previously created */
ret = pthread_setspecific(key, value);
```

`pthread_setspecific()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, `pthread_setspecific()` fails and returns an error value.

Note: `pthread_setspecific()` does ***not*** free its storage. If a new binding is set, the existing binding must be freed; otherwise, a ***memory leak can occur***.

Get the Thread-Specific Data Key

Use `pthread_getspecific()` to get the calling thread's binding for key, and store it in the location pointed to by value. This function is prototyped by:

```
int pthread_getspecific(pthread_key_t key);
```

A simple example use of this function is:

```
#include <pthread.h>
pthread_key_t key;
void *value;
/* key previously created */
value = pthread_getspecific(key);
```

Global and Private Thread-Specific Data Example

Thread-Specific Data Global but Private

Consider the following code:

```
body() {
...
while (write(fd, buffer, size) == -1) {
if (errno != EINTR) {
fprintf(mywindow, "%s\n", strerror(errno));
```

```
exit(1);
}
}
...
}
```

This code may be executed by any number of threads, but it has references to two global variables, errno and mywindow, that really should be references to items private to each thread.

References to errno should get the system error code from the routine called by this thread, not by some other thread. So, references to errno by one thread refer to a different storage location than references to errno by other threads. The mywindow variable is intended to refer to a stdio stream connected to a window that is private to the referring thread. So, as with `errno`, references to mywindow by one thread should refer to a different storage location (and, ultimately, a different window) than references to mywindow by other threads. The only difference here is that the threads library takes care of errno, but the programmer must somehow make this work for mywindow. The next example shows how the references to mywindow work. The preprocessor converts references to mywindow into invocations of the `mywindow` procedure. This routine in turn invokes `pthread_getspecific()`, passing it the `mywindow_key` global variable (it really is a global variable) and an output parameter, `win`, that receives the identity of this thread's window.

Turning Global References Into Private References Now consider this code fragment:

```
thread_key_t mywin_key;
FILE *_mywindow(void) {
FILE *win;
pthread_getspecific(mywin_key, &win);
return(win);
}
#define mywindow _mywindow()

void routine_uses_win( FILE *win) {
...
}
void thread_start(...) {
...
make_mywin();
...
routine_uses_win( mywindow )
...
}
```

The `mywin_key` variable identifies a class of variables for which each thread has its own private copy; that is, these variables are thread-specific data. Each thread calls `make_mywin` to initialize its window and to arrange for its instance of mywindow to refer to it. Once this routine is called, the thread can safely refer to `mywindow` and, after `mywindow`, the thread gets the reference to its private window. So, references to mywindow behave as if they were direct references to data private to the thread.

We can now set up our initial Thread-Specific Data:

```
void make_mywindow(void) {
FILE **win;
static pthread_once_t mykeycreated = PTHREAD_ONCE_INIT;
pthread_once(&mykeycreated, mykeycreate);
win = malloc(sizeof(*win));
create_window(win, ...);
pthread_setspecific(mywindow_key, win);
}
void mykeycreate(void) {
pthread_keycreate(&mywindow_key, free_key);
}
void free_key(void *win) {
free(win);
}
```

First, get a unique value for the key, `mywin_key`. This key is used to identify the thread-specific class of data. So, the first thread to call `make_mywin` eventually calls `pthread_keycreate()`, which assigns to its first argument a unique key. The second argument is a destructor function that is used to deallocate a thread's instance of this thread-specific data item once the thread terminates.

The next step is to allocate the storage for the caller's instance of this thread-specific data item. Having allocated the storage, a call is made to the `create_window` routine, which sets up a window for the thread and sets the storage pointed to by win to refer to it. Finally, a call is made to `pthread_setspecific()`, which associates the value contained in win (that is, the location of the storage containing the reference to the window) with the key. After this, whenever this thread calls `pthread_getspecific()`, passing the global key, it gets the value that was associated with this key by this thread when it called `pthread_setspecific()`. When a thread terminates, calls are made to the destructor functions that were set up in `pthread_key_create()`. Each destructor function is called only if the terminating thread established a value for the key by calling `pthread_setspecific()`.

Getting the Thread Identifiers

The function `pthread_self()` can be called to return the ID of the calling thread. It is prototyped by:

```
pthread_t pthread_self(void);
```

It is use is very straightforward:

```
#include <pthread.h>
pthread_t tid;
tid = pthread_self();
```

Comparing Thread IDs

The function `pthread_equal()` can be called to compare the thread identification numbers of two threads. It is prototyped by:

```
int pthread_equal(pthread_t tid1, pthread_t tid2);
```

It is use is straightforward to use, also:

```
#include <pthread.h>
pthread_t tid1, tid2;
int ret;
ret = pthread_equal(tid1, tid2);
```

As with other comparison functions, `pthread_equal()` returns a non-zero value when `tid1` and `tid2` are equal; otherwise, zero is returned. When either `tid1` or `tid2` is an invalid thread identification number, the result is unpredictable.

Initializing Threads

Use `pthread_once()` to call an initialization routine the first time `pthread_once()` is called -- Subsequent calls to have no effect. The prototype of this function is:

```
int pthread_once(pthread_once_t *once_control,
void (*init_routine)(void));
```

Yield Thread Execution

The function `sched_yield()` to cause the current thread to yield its execution in favor of another thread with the same or greater priority. It is prototyped by:

```
int sched_yield(void);
```

It is clearly a simple function to call:

```
#include <sched.h>
int ret;
ret = sched_yield();
```

`sched_yield()` returns zero after completing successfully. Otherwise -1 is returned and errno is set to indicate the error condition.

Set the Thread Priority

Use `pthread_setschedparam()` to modify the priority of an existing thread. This function has no effect on scheduling policy. It is prototyped as follows:

```
int pthread_setschedparam(pthread_t tid, int policy,
const struct sched_param *param);
```

and used as follows:

```
#include <pthread.h>
pthread_t tid;
int ret;
struct sched_param param;
int priority;
/* sched_priority will be the priority of the thread */
sched_param.sched_priority = priority;
/* only supported policy, others will result in ENOTSUP */

policy = SCHED_OTHER;
/* scheduling parameters of target thread */
ret = pthread_setschedparam(tid, policy, &param);
```

`pthread_setschedparam()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When either of the following conditions occurs, the `pthread_setschedparam()` function fails and returns an error value.

Get the Thread Priority

`int pthread_getschedparam(pthread_t tid, int policy, struct schedparam *param)` gets the priority of the existing thread.

An example call of this function is:

```
#include <pthread.h>
pthread_t tid;
sched_param param;
int priority;
int policy;
int ret;
/* scheduling parameters of target thread */
ret = pthread_getschedparam (tid, &policy, &param);
/* sched_priority contains the priority of the thread */
```

```
priority = param.sched_priority;
```

`pthread_getschedparam()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, the function fails and returns the error value set.

Send a Signal to a Thread

Signal may be sent to threads is a similar fashion to those for process as follows:

```
#include <pthread.h>
#include <signal.h>
int sig;
pthread_t tid;
int ret;
ret = pthread_kill(tid, sig);
```

`pthread_kill()` sends the signal `sig` to the thread specified by `tid`. `tid` must be a thread within the same process as the calling thread. The `sig` argument must be a valid signal of the same type defined for `signal()` in < `signal.h`> (See Chapter 23)

When `sig` is zero, error checking is performed but no signal is actually sent. This can be used to check the validity of tid.

This function returns zero after completing successfully. Any other returned value indicates that an error occurred. When either of the following conditions occurs, `pthread_kill()` fails and returns an error value.

Access the Signal Mask of the Calling Thread

The function `pthread_sigmask()` may be used to change or examine the signal mask of the calling thread. It is prototyped as follows:

```
int pthread_sigmask(int how, const sigset_t *new, sigset_t *old);
```

Example uses of this function include:

```
#include <pthread.h>
#include <signal.h>
int ret;
sigset_t old, new;
ret = pthread_sigmask(SIG_SETMASK, &new, &old); /* set new mask */
ret = pthread_sigmask(SIG_BLOCK, &new, &old); /* blocking mask */
ret = pthread_sigmask(SIG_UNBLOCK, &new, &old); /* unblocking */
```

how determines how the signal set is changed. It can have one of the following values:

`SIG_SETMASK`
-- Replace the current signal mask with new, where new indicates the new signal mask.

`SIG_BLOCK`
-- Add new to the current signal mask, where new indicates the set of signals to block.

`SIG_UNBLOCK`
-- Delete new from the current signal mask, where new indicates the set of signals to unblock.

When the value of `new` is `NULL`, the value of how is not significant and the signal mask of the thread is unchanged. So, to inquire about currently blocked signals, assign a `NULL` value to the new argument. The old variable points to the space where the previous signal mask is stored, unless it is `NULL`.

`pthread_sigmask()` returns a zero when it completes successfully. Any other returned value indicates that an error occurred. When the following condition occurs, `pthread_sigmask()` fails and returns an errro value.

Terminate a Thread

A thread can terminate its execution in the following ways:

- By returning from its first (outermost) procedure, the threads start routine; see `pthread_create()`
- By calling `pthread_exit()`, supplying an exit status
- By termination with POSIX cancel functions; see `pthread_cancel()`

The `void pthread_exit(void *status)` is used terminate a thread in a similar fashion the `exit()` for a process:

```
#include <pthread.h>
int status;
pthread_exit(&status); /* exit with status */
```

The `pthread_exit()` function terminates the calling thread. All thread-specific data bindings are released. If the calling thread is not detached, then the thread's ID and the exit status specified by status are retained until the thread is waited for (blocked). Otherwise, status is ignored and the thread's ID can be reclaimed immediately.

The `pthread_cancel()` function to cancel a thread is prototyped:

```
int pthread_cancel(pthread_t thread);
```

and called:

```
#include <pthread.h>
pthread_t thread;
int ret;
ret = pthread_cancel(thread);
```

How the cancellation request is treated depends on the state of the target thread. Two functions,

`pthread_setcancelstate()` and `pthread_setcanceltype()` (see `man` pages for further information on these functions), determine that state.

`pthread_cancel()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, the function fails and returns an error value.

Solaris Threads: <`thread.h`>

Solaris have many similarities to POSIX threads,In this sectionfocus on the Solaris features that are not found in POSIX threads. Where functionality is virtually the same for both Solaris threads and for pthreads, (even though the function names or arguments might differ), only a brief example consisting of the correct include file and the function prototype is presented. Where return values are not given for the Solaris threads functions, see the appropriate `man` pages.

The Solaris threads API and the pthreads API are two solutions to the same problem: building parallelism into application software. Although each API is complete in itself, you can safely mix Solaris threads functions and pthread functions in the same program.

The two APIs do not match exactly, however. Solaris threads supports functions that are not found in pthreads, and pthreads includes functions that are not supported in the Solaris interface. For those functions that do match, the associated arguments might not, although the information content is effectively the same.

By combining the two APIs, you can use features not found in one to enhance the other. Similarly, you can run applications using Solaris threads, exclusively, with applications using pthreads, exclusively, on the same system.

To use the Solaris threads functions described in this chapter, you must link with the Solaris threads library `-lthread` and include the `<thread.h>` in all programs.

Unique Solaris Threads Functions

Let us begin by looking at some functions that are unique to Solaris threads:

- Suspend Thread Execution
- Continue a Suspended Thread
- Set Thread Concurrency Level
- Get Thread Concurrency

Suspend Thread Execution

The function `thr_suspend()` immediately suspends the execution of the thread specified by a target thread, (`tid` below). It is prototyped by:

```
int thr_suspend(thread_t tid);
```

On successful return from `thr_suspend()`, the suspended thread is no longer executing. Once a thread is suspended, subsequent calls to `thr_suspend()` have no effect. Signals cannot awaken the suspended thread; they remain pending until the thread resumes execution.

A simple example call is as follows:

```
#include <thread.h>

thread_t tid; /* tid from thr_create() */
/* pthreads equivalent of Solaris tid from thread created */
/* with pthread_create() */
pthread_t ptid;
int ret;
ret = thr_suspend(tid);
/* using pthreads ID variable with a cast */
ret = thr_suspend((thread_t) ptid);
```

Note: `pthread_t tid` as defined in pthreads is the same as `thread_t` tid in Solaris threads. `tid` values can be used interchangeably either by assignment or through the use of casts.

Continue a Suspended Thread

The function `thr_continue()` resumes the execution of a suspended thread. It is prototypes as follows:

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

Once a suspended thread is continued, subsequent calls to `thr_continue()` have no effect.

A suspended thread will ***not*** be awakened by a signal. The signal stays pending until the execution of the thread is resumed by `thr_continue()`.

`thr_continue()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, `thr_continue()` The following code fragment illustrates the use of the function:

```
thread_t tid; /* tid from thr_create()*/
/* pthreads equivalent of Solaris tid from thread created */
/* with pthread_create()*/
pthread_t ptid;
int ret;
ret = thr_continue(tid);
/* using pthreads ID variable with a cast */
ret = thr_continue((thread_t) ptid)
```

Set Thread Concurrency Level

By default, Solaris threads attempt to adjust the system execution resources (LWPs) used to run unbound threads to match the real number of active threads. While the Solaris threads package cannot make perfect decisions, it at least ensures that the process continues to make progress. When you have some idea of the number of unbound threads that should be simultaneously active (executing code or system calls), tell the library through `thr_setconcurrency(int new_level)`. To get the number of threads being used, use the function `thr_getconcurrencyint(void)`:

`thr_setconcurrency()` provides a hint to the system about the required level of concurrency in the application. The system ensures that a sufficient number of threads are active so that the process continues to make progress, for example:

```
#include <thread.h>
int new_level;
int ret;

ret = thr_setconcurrency(new_level);
```

Unbound threads in a process might or might not be required to be simultaneously active. To conserve system resources, the threads system ensures by default that enough threads are active for the process to make progress, and that the process will not deadlock through a lack of concurrency. Because this might not produce the most effective level of concurrency, `thr_setconcurrency()` permits the application to give the threads system a hint, specified by `new_level`, for the desired level of concurrency. The actual number of simultaneously active threads can be larger or smaller than `new_level`. Note that an application with multiple compute-bound threads can fail to schedule all the runnable threads if `thr_setconcurrency()` has not been called to adjust the level of execution resources. You can also affect the value for the desired concurrency level by setting the `THR_NEW_LW` flag in `thr_create()`. This effectively increments the current level by one.

`thr_setconcurrency()` a zero when it completes successfully. Any other returned value indicates that an error occurred. When any of the following conditions are detected, `thr_setconcurrency()` fails and returns the corresponding value to errno.

Readers/Writer Locks

Readers/Writer locks are another unique feature of Solaris threads. They allow simultaneous read access by many threads while restricting write access to only one thread at a time.

When any thread holds the lock for reading, other threads can also acquire the lock for reading but must wait to acquire the lock for writing. If one thread holds the lock for writing, or is waiting to acquire the lock for writing, other threads must wait to acquire the lock for either reading or writing. Readers/writer locks are slower than mutexes, but can improve performance when they protect data that are not frequently written but that are read by many concurrent threads. Use readers/writer locks to synchronize threads in this process and other processes by allocating them in memory that is writable and shared among the cooperating processes (see mmap(2)) and by initializing them for this behavior. By default, the acquisition order is not defined when multiple threads are waiting for a readers/writer lock. However, to avoid writer starvation, the Solaris threads package tends to favor writers over readers. Readers/writer locks must be initialized before use.

Initialize a Readers/Writer Lock

The function `rwlock_init()` initialises the readers/writer lock. it is prototypes in <`synch.h`> or <`thread.h`> as follows:

```
int rwlock_init(rwlock_t *rwlp, int type, void * arg);
```

The readers/writer lock pointed to by `rwlp` and to set the lock state to unlocked. `type` can be one of the following

`USYNC_PROCESS`
-- The readers/writer lock can be used to synchronize threads in this process and other processes.

`USYNC_THREAD`
-- The readers/writer lock can be used to synchronize threads in this process, only.

Note: that `arg` is currently ignored.

`rwlock_init()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, the function fails and returns the corresponding value to `errno`.

Multiple threads must not initialize the same readers/writer lock simultaneously. Readers/writer locks can also be initialized by allocation in zeroed memory, in which case a type of `USYNC_THREAD` is assumed. A readers/writer lock must not be reinitialized while other threads might be using it.

An example code fragment that initialises Readers/Writer Locks with Intraprocess Scope is as follows:

```
#include <thread.h>

rwlock_t rwlp;
int ret;
/* to be used within this process only */
ret = rwlock_init(&rwlp, USYNC_THREAD, 0);
Initializing Readers/Writer Locks with Interprocess Scope
#include <thread.h>
rwlock_t rwlp;
int ret;
/* to be used among all processes */
ret = rwlock_init(&rwlp, USYNC_PROCESS, 0);
```

Acquire a Read Lock

To acquire a read lock on the readers/writer lock use the `rw_rdlock()` function:

```
int rw_rdlock(rwlock_t *rwlp);
```

The readers/writer lock pointed to by `rwlp`. When the readers/writer lock is already locked for writing, the calling thread blocks until the write lock is released. Otherwise, the read lock is acquired.

`rw_rdlock()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, the function fails and returns the corresponding value to `errno`.

A function `rw_tryrdlock(rwlock_t *rwlp)` may also be used to attempt to acquire a read lock on the readers/writer lock pointed to by rwlp. When the readers/writer lock is already locked for writing, it returns an error. Otherwise, the read lock is acquired. This function returns zero after completing successfully. Any other returned value indicates that an error occurred.

Acquire a Write Lock

The function `rw_wrlock(rwlock_t *rwlp)` acquires a write lock on the readers/writer lock pointed to by `rwlp`. When the readers/writer lock is already locked for reading or writing, the calling thread blocks until all the read locks and write locks are released. Only one thread at a time can hold a write lock on a readers/writer lock.

`rw_wrlock()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

Use `rw_trywrlockrwlock_t *rwlp)` to attempt to acquire a write lock on the readers/writer lock pointed to by rwlp. When the readers/writer lock is already locked for reading or writing, it returns an error.

`rw_trywrlock()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

Unlock a Readers/Writer Lock

The function `rw_unlock(rwlock_t *rwlp)` unlocks a readers/writer lock pointed to by `rwlp`. The readers/writer lock must be locked and the calling thread must hold the lock either for reading or writing. When any other threads are waiting for the readers/writer lock to become available, one of them is unblocked.

`rw_unlock()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

Destroy Readers/Writer Lock State

The function `rwlock_destroy(rwlock_t *rwlp)` destroys any state associated with the readers/writer lock pointed to by `rlwp`. The space for storing the readers/writer lock is not freed.

`rwlock_destroy()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

Readers/Writer Lock Example

The following example uses a bank account analogy to demonstrate readers/writer locks. While the program could allow multiple threads to have concurrent read-only access to the account balance, only a single writer is allowed. Note that the `get_balance()` function needs the lock to ensure that the addition of the checking and saving balances occurs atomically.

```
rwlock_t account_lock;
float checking_balance = 100.0;
float saving_balance = 100.0;
...
rwlock_init(&account_lock, 0, NULL);
...
float
get_balance() {
float bal;
rw_rdlock(&account_lock);
bal = checking_balance + saving_balance;
rw_unlock(&account_lock);
return(bal);
}
void
transfer_checking_to_savings(float amount) {
rw_wrlock(&account_lock);
```

```
checking_balance = checking_balance - amount;
saving_balance = saving_balance + amount;
rw_unlock(&account_lock);
}
```

Similar Solaris Threads Functions

Here we simply list the similar thread functions and their prototype definitions, except where the complexity of the function merits further exposition. .

Create a Thread

The `thr_create()` routine is one of the most elaborate of all the Solaris threads library routines.

It is prototyped as follows:

```
int thr_create(void *stack_base, size_t stack_size,
void *(*start_routine) (void *), void *arg, long flags,
thread_t *new_thread);
```

Thjis function adds a new thread of control to the current process. Note that the new thread does not inherit pending signals, but it does inherit priority and signal masks.

`stack_base` contains the address for the stack that the new thread uses. If `stack_base` is NULL then `thr_create()` allocates a stack for the new thread with at least `stac_size` bytes. `stack_size` Contains the size, in number of bytes, for the stack that the new thread uses. If `stack_size` is zero, a default size is used. In most cases, a zero value works best. If `stack_size` is not zero, it must be greater than the value returned by `thr_min_stack(void)` inquiry function.

There is no general need to allocate stack space for threads. The threads library allocates one megabyte of virtual memory for each thread's stack with no swap space reserved.

`start_routine` contains the function with which the new thread begins execution. When `start_routine` returns, the thread exits with the exit status set to the value returned by `start_routine`

`arg` can be anything that is described by void, which is typically any 4-byte value. Anything larger must be passed indirectly by having the argument point to it.

Note that you can supply only one argument. To get your procedure to take multiple arguments, encode them as one (such as by putting them in a structure).

`flags` specifies attributes for the created thread. In most cases a zero value works best. The value in flags is constructed from the bitwise inclusive OR of the following:

`THR_SUSPENDED`

-- Suspends the new thread and does not execute `start_routine` until the thread is started by `thr_continue()`. Use this to operate on the thread (such as changing its priority) before you run it. The termination of a detached thread is ignored.

`THR_DETACHED`

-- Detaches the new thread so that its thread ID and other resources can be reused as soon as the thread terminates. Set this when you do not want to wait for the thread to terminate. Note - When there is no explicit synchronization to prevent it, an unsuspended, detached thread can die and have its thread ID reassigned to another new thread before its creator returns from `thr_create()`.

`THR_BOUND`

-- Permanently binds the new thread to an LWP (the new thread is a bound thread).

`THR_NEW_LWP`

-- Increases the concurrency level for unbound threads by one. The effect is similar to incrementing concurrency by one with `thr_setconcurrency()`, although `THR_NEW_LWP` does not affect the level set through the `thr_setconcurrency()` function. Typically, `THR_NEW_LWP` adds a new LWP to the pool of LWPs running unbound threads.

When you specify both `THR_BOUND` and `THR_NEW_LWP`, two LWPs are typically created -- one for the bound thread and another for the pool of LWPs running unbound threads.

`THR_DAEMON`

-- Marks the new thread as a daemon. The process exits when all nondaemon threads exit. Daemon threads do not affect the process exit status and are ignored when counting the number of thread exits.

A process can exit either by calling `exit()` or by having every thread in the process that was not created with the `THR_DAEMON` flag call `thr_exit()`. An application, or a library it calls, can create one or more threads that should be ignored (not counted) in the decision of whether to exit. The `THR_DAEMON`l flag identifies threads that are not counted in the process exit criterion.

`new_thread` points to a location (when `new_thread` is not `NULL`) where the ID of the new thread is stored when `thr_create()` is successful. The caller is responsible for supplying the storage this argument points to. The ID is valid only within the calling process. If you are not interested in this identifier, supply a zero value to `new_thread`.

`thr_create()` returns a zero and exits when it completes successfully. Any other returned value indicates that an error occurred. When any of the following conditions are detected, `thr_create()` fails and returns the corresponding value to `errno`.

Get the Thread Identifier

The `int thr_self(void)` to get the ID of the calling thread.

Yield Thread Execution

`void thr_yield(void)` causes the current thread to yield its execution in favor of another thread with the same or greater priority; otherwise it has no effect. There is no guarantee that a thread calling `thr_yield()` will do so.

Signals and Solaris Threads

The following functions exist and operate as do pthreads.

`int thr_kill(thread_t target_thread, int sig)` sends a signal to a thread.

`int thr_sigsetmask(int how, const sigset_t *set, sigset_t *oset)` to change or examine the signal mask of the calling thread.

Terminating a Thread

The `void th_exit(void *status)` to terminates a thread.

The `int thr_join(thread_t tid, thread_t *departedid, void **status)` function to wait for a thread to terminate.

Therefore to join specific threads one would do:

```
#include <thread.h>
thread_t tid;
thread_t departedid;
int ret;
int status;
/* waiting to join thread "tid" with status */
ret = thr_join(tid, &departedid, (void**)&status);
/* waiting to join thread "tid" without status */
ret = thr_join(tid, &departedid, NULL);
/* waiting to join thread "tid" without return id and status */
ret = thr_join(tid, NULL, NULL);
```

When the `tid` is `(thread_t) 0`, then `thread_join()` waits for any undetached thread in the process to terminate. In other words, when no thread identifier is specified, any undetached thread that exits causes `thread_join()` to return.

To join any threads:

```
#include <thread.h>
thread_t tid;
thread_t departedid;
int ret;
int status;
/* waiting to join thread "tid" with status */
ret = thr_join(NULL, &departedid, (void **)&status);
```

By indicating NULL as `thread id` in the `thr_join()`, a join will take place when any non detached thread in the process exits. The departedid will indicate the thread ID of exiting thread.

Creating a Thread-Specific Data Key

Except for the function names and arguments, thread specific data is the same for Solaris as it is for POSIX.

`int thr_keycreate(thread_key_t *keyp, void (*destructor) (void *value))` allocates a key that is used to identify thread-specific data in a process.

`int thr_setspecific(thread_key_t key, void *value)` binds value to the thread-specific data key, key, for the calling thread.

`int thr_getspecific(thread_key_t key, void **valuep)` stores the current value bound to key for the calling thread into the location pointed to by valuep.

In Solaris threads, if a thread is to be created with a priority other than that of its parent's, it is created in `SUSPEND` mode. While suspended, the threads priority is modified using the `int thr_setprio(thread_t tid, int newprio)` function call; then it is continued.

An unbound thread is usually scheduled only with respect to other threads in the process using simple priority levels with no adjustments and no kernel involvement. Its system priority is usually uniform and is inherited from the creating process.

The function `thr_setprio()` changes the priority of the thread, specified by `tid`, within the current process to the priority specified by newprio.

By default, threads are scheduled based on fixed priorities that range from zero, the least significant, to the largest integer. The `tid` will preempt lower priority threads, and will yield to higher priority threads. For example:

```
#include <thread.h>
```

```
thread_t tid;
int ret;
int newprio = 20;
/* suspended thread creation */
ret = thr_create(NULL, NULL, func, arg, THR_SUSPEND, &tid);
/* set the new priority of suspended child thread */
ret = thr_setprio(tid, newprio);
/* suspended child thread starts executing with new priority */

ret = thr_continue(tid);
```

Use int thr_getprio(thread_t tid, int *newprio) to get the current priority for the thread. Each thread inherits a priority from its creator. `thr_getprio()` stores the current priority, `tid`, in the location pointed to by `newprio`.

Example Use of Thread Specific Data:Rethinking Global Variables

Historically, most code has been designed for single-threaded programs. This is especially true for most of the library routines called from C programs. The following implicit assumptions were made for single-threaded code:

- When you write into a global variable and then, a moment later, read from it, what you read is exactly what you just wrote.
- This is also true for nonglobal, static storage.
- You do not need synchronization because there is nothing to synchronize with.

The next few examples discuss some of the problems that arise in multithreaded programs because of these assumptions, and how you can deal with them.

Traditional, single-threaded C and UNIX have a convention for handling errors detected in system calls. System calls can return anything as a functional value (for example, write returns the number of bytes that were transferred). However, the value -1 is reserved to indicate that something went wrong. So, when a system call returns -1, you know that it failed.

Consider the following piece of code:

```
extern int errno;

...

if (write(file_desc, buffer, size) == -1)
  { /* the system call failed */
    fprintf(stderr, "something went wrong, error code = %d\n", errno);
    exit(1);
  }
```

Rather than return the actual error code (which could be confused with normal return values), the error code is placed into the global variable `errno`. When the system call fails, you can look in `errno` to find out what went wrong.

Now consider what happens in a multithreaded environment when two threads fail at about the same time, but with different errors.

- Both expect to find their error codes in `errno`,
- **but** one copy of errno cannot hold both values.a

This global variable approach simply does not work for multithreaded programs. Threads solves this problem through a conceptually new storage class: ***thread-specific data***.

This storage is similar to global storage in that it can be accessed from any procedure in which a thread might be running. However, it is private to the thread: when two threads refer to the thread-specific data location of the same name, they are referring to two different areas of storage.

So, when using threads, each reference to `errno` is thread-specific because each thread has a private copy of `errno`. This is achieved in this implementation by making `errno` a macro that expands to a function call.

Compiling a Multithreaded Application

There are many options to consider for header files, define flags, and linking.

Preparing for Compilation

The following items are required to compile and link a multithreaded program.

- A standard C compiler (`cc, gcc` ***etc***)
- Include files:
  - <thread.h> and <pthread.h>
  - <errno.h>, <limits.h>, <signal.h>, <unistd.h>
- The Solaris threads library (`libthread`), the POSIX threads library (`libpthread`), and possibly the POSIX realtime library (`libposix4`) for semaphores
- MT-safe libraries (`libc, libm, libw, libintl, libnsl, libsocket, libmalloc, libmapmalloc`, and so on)

The include file <thread.h>, used with the `-lthread` library, compiles code that is upward compatible with earlier releases of the Solaris system. This library contains both interfaces: those with Solaris semantics and those with POSIX semantics. To call `thr_setconcurrency()` with POSIX threads, your program needs to include <thread.h>.

The include file <pthread.h>, used with the `-lpthread` library, compiles code that is conformant with the multithreading interfaces defined by the POSIX 1003.1c standard. For complete POSIX compliance, the define flag `_POSIX_C_SOURCE` should be set to a (long)

value , as follows:

```
cc [flags] file... -D_POSIX_C_SOURCE=N (where N 199506L)
```

You can mix Solaris threads and POSIX threads in the same application, by including both <thread.h> and <pthread.h>, and linking with either the `-lthread` or `-lpthread` library. In mixed use, Solaris semantics prevail when compiling with `-D_REENTRANT` flag set $\geq 199506L$ and linking with `-lthread`, whereas POSIX semantics prevail when compiling with `D_POSIX_C_SOURCE` flag set $\geq 199506L$ and linking with `-lpthread`. Defining `_REENTRANT` or `_POSIX_C_SOURCE`

Linking With libthread or libpthread

For POSIX threads behavior, load the `libpthread` library. For Solaris threads behavior, load the `libthread` library. Some POSIX programmers might want to link with `-lthread`to preserve the Solaris distinction between `fork()` and `fork1()`. All that `-lpthread` really does is to make `fork()` behave the same way as the Solaris `fork1()` call, and change the behavior of `alarm()`.

To use libthread, specify `-lthread` last on the `cc` command line.

To use libpthread, specify `-lpthread` last on the `cc` command line.

Do not link a ***nonthreaded*** program with `-lthread` or `-lpthread`. Doing so establishes multithreading mechanisms at link time that are initiated at run time. These ***slow down*** a single-threaded application, waste system resources, and produce misleading results when you debug your code.

Note: For C++ programs that use threads, use the `-mt` option, rather than `-lthread`, to compile and link your application. The `-mt` option links with `libthread` and ensures proper library linking order. (Using `-lthread` might cause your program to crash (core dump).

Linking with -lposix4 for POSIX Semaphores

The Solaris semaphore routines (see Chapter 30.3) are contained in the `libthread` library. By contrast, you link with the `-lposix4` library to get the standard POSIX semaphore routines (See Chapter 25)

Debugging a Multithreaded Program

The following list points out some of the more frequent oversights and errors that can cause bugs in multithreaded programs.

- Passing a pointer to the caller's stack as an argument to a new thread.
- Accessing global memory (shared changeable state) without the protection of a synchronization mechanism.
- Creating deadlocks caused by two threads trying to acquire rights to the same pair of global resources in alternate order (so that one thread controls the first resource and the other controls the second resource and neither can proceed until the other gives up).
- Trying to reacquire a lock already held (recursive deadlock).
- Creating a hidden gap in synchronization protection. This is caused when a code segment protected by a synchronization mechanism contains a call to a function that frees and then reacquires the synchronization mechanism before it returns to the caller. The result is that it appears to the caller that the global data has been protected when it actually has not.
- Mixing UNIX signals with threads -- it is better to use the `sigwait()` model for handling asynchronous signals.
- Forgetting that default threads are created `PTHREAD_CREATE_JOINABLE` and must be reclaimed with `pthread_join()`. **Note**, `pthread_exit()` does not free up its storage space.
- Making deeply nested, recursive calls and using large automatic arrays can cause problems because multithreaded programs have a more limited stack size than single-threaded programs.
- Specifying an inadequate stack size, or using non-default stacks. And, note that multithreaded programs (especially those containing bugs) often behave differently in two successive runs, given identical inputs, because of differences in the thread scheduling order.

In general, multithreading bugs are statistical instead of deterministic. Tracing is usually a more effective method of finding order of execution problems than is breakpoint-based debugging.

Further Threads Programming:Thread Attributes (POSIX)

The previous chapter covered the basics of threads creation using default attributes. This chapter discusses setting attributes at thread creation time.

Note that only pthreads uses attributes and cancellation, so the API covered in this chapter is for POSIX threads only. Otherwise, the functionality for Solaris threads and pthreads is largely the same.

Attributes

Attributes are a way to specify behavior that is different from the default. When a thread is created with `pthread_create()` or when a synchronization variable is initialized, an attribute object can be specified. **Note:** however that the default atributes are usually sufficient for most applications.

Impottant Note: Attributes are specified ***only at thread creation time***; they **cannot** be altered while the thread is **being used**.

Thus three functions are usually called in tandem

- Thread attibute intialisation -- `pthread_attr_init()` create a default `pthread_attr_t tattr`
- Thread attribute value change (unless defaults appropriate) -- a variety of `pthread_attr_*()` functions are available to set individual attribute values for the `pthread_attr_t tattr` structure. (see below).
- Thread creation -- a call to `pthread_create()` with approriate attribute values set in a `pthread_attr_t tattr` structure.

The following code fragment should make this point clearer:

```
#include <pthread.h>

pthread_attr_t tattr;
pthread_t tid;
void *start_routine;
void arg
int ret;

/* initialized with default attributes */
ret = pthread_attr_init(&tattr);

/* call an appropriate functions to alter a default value */
ret = pthread_attr_*(&tattr,SOME_ATRIBUTE_VALUE_PARAMETER);

/* create the thread */
ret = pthread_create(&tid, &tattr, start_routine, arg);
```

In order to save space, code examples mainly focus on the attribute setting functions and the intializing and creation functions are ommitted. These **must** of course be present in all actual code fragtments.

An attribute object is opaque, and cannot be directly modified by assignments. A set of functions is provided to initialize, configure, and destroy each object type. Once an attribute is initialized and configured, it has process-wide scope. The suggested method for using attributes is to configure all required state specifications at one time in the early stages of program execution. The appropriate attribute object can then be referred to as needed. Using attribute objects has two primary advantages:

- First, it adds to code portability. Even though supported attributes might vary between implementations, you need not modify function calls that create thread entities because the attribute object is hidden from the interface. If the target port supports attributes that are not found in the current port, provision must be made to manage the new attributes. This is an easy porting task though, because attribute objects need only be initialized once in a well-defined location.
- Second, state specification in an application is simplified. As an example, consider that several sets of threads might exist within a process, each providing a separate service, and each with its own state requirements. At some point in the early stages of the application, a thread attribute object can be initialized for each set. All future thread creations will then refer to the attribute object initialized for that type of thread. The initialization phase is simple and localized, and any future modifications can be made quickly and reliably.

Attribute objects require attention at process exit time. When the object is initialized, memory is allocated for it. This memory must be returned to the system. The pthreads standard provides function calls to destroy attribute objects.

Initializing Thread Attributes

The function `pthread_attr_init()` is used to initialize object attributes to their default values. The storage is allocated by the thread system during execution.

The function is prototyped by:

```
int pthread_attr_init(pthread_attr_t *tattr);
```

An example call to this function is:

```
#include <pthread.h>
pthread_attr_t tattr;
int ret;
/* initialize an attribute to the default value */
ret = pthread_attr_init(&tattr);
```

The default values for attributes (`tattr`) are:

Attribute	Value	Result
scope	`PTHREAD_SCOPE_PROCESS`	New thread is
		unbound -
		not
		permanently
		attached to
		LWP.
detachstate	`PTHREAD_CREATE_JOINABLE`	Exit status
		and thread are
		preserved
		after the
		thread
		terminates.
stackaddr	`NULL`	New thread
		has
		system-allocated stack
		address.
stacksize	`1` megabyte	New thread
		has
		system-defined
		stack size.
		priority New thread
		inherits
		parent thread
		priority.
inheritsched	`PTHREAD_INHERIT_SCHED`	New thread
		inherits
		parent thread
		scheduling
		priority.
schedpolicy	`SCHED_OTHER`	New thread
		uses
		Solaris-defined
		fixed priority
		scheduling;
		threads run
		until
		preempted by a
		higher-priority
		thread or
		until they
		block or
		yield.

This function zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns an error value (to `errno`).

Destroying Thread Attributes

The function `pthread_attr_destroy()` is used to remove the storage allocated during initialization. The attribute object becomes invalid. It is prototyped by:

```
int pthread_attr_destroy(pthread_attr_t *tattr);
```

A sample call to this functions is:

```
#include <pthread.h>
pthread_attr_t tattr;
int ret;
/* destroy an attribute */
ret = pthread_attr_destroy(&tattr);
```

Attribites are declared as for `pthread_attr_init()` above.

`pthread_attr_destroy()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

Thread's Detach State

When a thread is created detached (`PTHREAD_CREATE_DETACHED`), its thread ID and other resources can be reused as soon as the thread terminates.

If you do not want the calling thread to wait for the thread to terminate then call the function `pthread_attr_setdetachstate()`.

When a thread is created nondetached (`PTHREAD_CREATE_JOINABLE`), it is assumed that you will be waiting for it. That is, it is assumed that you will be executing a `pthread_join()` on the thread. Whether a thread is created detached or nondetached, the process does not exit until all threads have exited.

`pthread_attr_setdetachstate()` is prototyped by:

```
int pthread_attr_setdetachstate(pthread_attr_t *tattr,int detachstate);
```

`pthread_attr_setdetachstate()` returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

An example call to detatch a thread with this function is:

```
#include <pthread.h>
pthread_attr_t tattr;
int ret;
/* set the thread detach state */
ret = pthread_attr_setdetachstate(&tattr,PTHREAD_CREATE_DETACHED);
```

Note - When there is no explicit synchronization to prevent it, a newly created, detached thread can die and have its thread ID reassigned to another new thread before its creator returns from `pthread_create()`. For nondetached (`PTHREAD_CREATE_JOINABLE`) threads, it is very important that some thread join with it after it terminates -- otherwise the resources of that thread are not released for use by new threads. This commonly results in a memory leak. So when you do not want a thread to be joined, create it as a detached thread.

It is quite common that you will wish to create a thread which is detatched from creation. The following code illustrates how this may be achieved with the standard calls to initialise and set and then create a thread:

```
#include <pthread.h>
pthread_attr_t tattr;
pthread_t tid;
void *start_routine;
void arg
int ret;

/* initialized with default attributes */
ret = pthread_attr_init(&tattr);
ret = pthread_attr_setdetachstate(&tattr,PTHREAD_CREATE_DETACHED);
ret = pthread_create(&tid, &tattr, start_routine, arg);
```

The function `pthread_attr_getdetachstate()` may be used to retrieve the thread create state, which can be either detached or joined. It is prototyped by:

```
int pthread_attr_getdetachstate(const pthread_attr_t *tattr, int *detachstate);
```

`pthread_attr_getdetachstate()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

An example call to this fuction is:

```
#include <pthread.h>
```

```
pthread_attr_t tattr;
int detachstate;
int ret;

/* get detachstate of thread */
ret = pthread_attr_getdetachstate (&tattr, &detachstate);
```

Thread's Set Scope

A thread may be bound (`PTHREAD_SCOPE_SYSTEM`) or an unbound (`PTHREAD_SCOPE_PROCESS`). Both these types of types are accessible **only** within a given process.

The function `pthread_attr_setscope()` to create a bound or unbound thread. It is prototyped by:

```
int pthread_attr_setscope(pthread_attr_t *tattr,int scope);
```

Scope takes on the value of either `PTHREAD_SCOP_SYSTEM` or `PTHREAD_SCOPE_PROCESS`.

`pthread_attr_setscope()` returns zero after completing successfully. Any other returned value indicates that an error occurred and an appropriate value is returned.

So to set a bound thread at thread creation on would do the following function calls:

```
#include <pthread.h>
pthread_attr_t attr;
pthread_t tid;
void start_routine;
void arg;
int ret;

/* initialized with default attributes */
ret = pthread_attr_init (&tattr);
/* BOUND behavior */
ret = pthread_attr_setscope(&tattr, PTHREAD_SCOPE_SYSTEM);
ret = pthread_create (&tid, &tattr, start_routine, arg);
```

If the following conditions occur, the function fails and returns the corresponding value.

The function `pthread_attr_getscope()` is used to retrieve the thread scope, which indicates whether the thread is bound or unbound. It is prototyped by:

```
int pthread_attr_getscope(pthread_attr_t *tattr, int *scope);
```

An example use of this function is:

```
#include <pthread.h>

pthread_attr_t tattr;
int scope;
int ret;

/* get scope of thread */
ret = pthread_attr_getscope(&tattr, &scope);
```

If successful the approriate (`PTHREAD_SCOP_SYSTEM` or `PTHREAD_SCOPE_PROCESS`) wil be stored in `scope`.

`pthread_att_getscope()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

Thread Scheduling Policy

The POSIX draft standard specifies scheduling policy attributes of `SCHED_FIFO` (first-in-first-out), `SCHED_RR` (round-robin), or `SCHED_OTHER` (an implementation-defined method). `SCHED_FIFO` and `SCHED_RR` are optional in POSIX, and **only** are supported for ***real time bound threads***.

Howver Note, currently, only the Solaris `SCHED_OTHER` default value is supported in pthreads. Attempting to set policy as `SCHED_FIFO` or `SCHED_RR` will result in the error `ENOSUP`.

The function is used to set the scheduling policy.It is prototyped by:

```
int pthread_attr_setschedpolicy(pthread_attr_t *tattr, int policy);
```

`pthread_attr_setschedpolicy()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

To set the scheduling policy to `SCHED_OTHER` simply do:

```
#include <pthread.h>
pthread_attr_t tattr;
int ret;

/* set the scheduling policy to SCHED_OTHER */
ret = pthread_attr_setschedpolicy(&tattr, SCHED_OTHER);
```

There is a function `pthread_attr_getschedpolicy()` that retrieves the scheduling policy. But, currently, it is not of great use as it can only return the (Solaris-based) `SCHED_OTHER` default value

Thread Inherited Scheduling Policy

The function `pthread_attr_setinheritsched()` can be used to the inherited scheduling policy of a thread. It is prototyped by:

```
int pthread_attr_setinheritsched(pthread_attr_t *tattr, int inherit);
```

An `inherit` value of `PTHREAD_INHERIT_SCHED` (the default) means that the scheduling policies defined in the creating thread are to be used, and any scheduling attributes defined in the `pthread_create()` call are to be ignored. If `PTHREAD_EXPLICIT_SCHED` is used, the attributes from the `pthread_create()` call are to be used.

The function returns zero after completing successfully. Any other returned value indicates that an error occurred.

An example call of this function is:

```
#include <pthread.h>
pthread_attr_t tattr;
int ret;

/* use the current scheduling policy */
ret = pthread_attr_setinheritsched(&tattr, PTHREAD_EXPLICIT_SCHED);
```

The function `pthread_attr_getinheritsched(pthread_attr_t *tattr, int *inherit)` may be used to inquire a current threads scheduling policy.

Set Scheduling Parameters

Scheduling parameters are defined in the `sched_param` structure; **only** priority `sched_param.sched_priority` is supported. This priority is an integer value the higher the value the higher a thread's proiority for scehduling. Newly created threads run with this priority. The `pthread_attr_setschedparam()` is used to set this stucture appropiately. It is prototyped by:

```
int pthread_attr_setschedparam(pthread_attr_t *tattr,
const struct sched_param *param);
```

and returns zero after completing successfully. Any other returned value indicates that an error occurred.

An example call to `pthread_attr_setschedparam()` is:

```
#include <pthread.h>
pthread_attr_t tattr;
int newprio;
sched_param param;


/* set the priority; others are unchanged */
newprio = 30;
param.sched_priority = newprio;

/* set the new scheduling param */
ret = pthread_attr_setschedparam (&tattr, &param);
```

The function `pthread_attr_getschedparam(pthread_attr_t *tattr, const struct sched_param *param)` may be used to inquire a current thread's priority of scheduling.

Thread Stack Size

Typically, thread stacks begin on page boundaries and any specified size is rounded up to the next page boundary. A page with no access permission is appended to the top of the stack so that most stack overflows result in sending a `SIGSEGV` signal to the offending thread. Thread stacks allocated by the caller are used as is.

When a stack is specified, the thread should also be created `PTHREAD_CREATE_JOINABLE`. That stack cannot be freed until the `pthread_join()` call for that thread has returned, because the thread's stack cannot be freed until the thread has terminated. The only reliable way to know if such a thread has terminated is through `pthread_join()`.

Generally, you do not need to allocate stack space for threads. The threads library allocates one megabyte of virtual memory for each thread's stack with no swap space reserved. (The library uses the `MAP_NORESERVE` option of mmap to make the allocations.)

Each thread stack created by the threads library has a red zone. The library creates the red zone by appending a page to the top of a stack to catch stack overflows. This page is invalid and causes a memory fault if it is accessed. Red zones are appended to all automatically allocated stacks whether the size is specified by the application or the default size is used.

Note: Because runtime stack requirements vary, you should be absolutely certain that the specified stack will satisfy the runtime requirements needed for library calls and dynamic linking.

There are very few occasions when it is appropriate to specify a stack, its size, or both. It is difficult even for an expert to know if the right size was specified. This is because even a program compliant with ABI standards cannot determine its stack size statically. Its size is dependent on the needs of the particular runtime environment in which it executes.

Building Your Own Thread Stack

When you specify the size of a thread stack, be sure to account for the allocations needed by the invoked function and by each function called. The accounting should include calling sequence needs, local variables, and information structures.

Occasionally you want a stack that is a bit different from the default stack. An obvious situation is when the thread needs more than one megabyte of stack space. A less obvious situation is when the default stack is too large. You might be creating thousands of threads and not have enough virtual memory to handle the gigabytes of stack space that this many default stacks require.

The limits on the maximum size of a stack are often obvious, but what about the limits on its minimum size? There must be enough stack space to handle all of the stack frames that are pushed onto the stack, along with their local variables, and so on.

You can get the absolute minimum limit on stack size by calling the macro `PTHREAD_STACK_MIN` (defined in <pthread.h>), which returns the amount of stack space required for a thread that executes a `NULL` procedure. Useful threads need more than this, so be very careful when reducing the stack size.

The function `pthread_attr_setstacksize()` is used to set this a thread's stack size, it is prototyped by:

```
int pthread_attr_setstacksize(pthread_attr_t *tattr, int stacksize);
```

The `stacksize` attribute defines the size of the stack (in bytes) that the system will allocate. The size should not be less than the system-defined minimum stack size.

`pthread_attr_setstacksize()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

An example call to set the stacksize is:

```
#include <pthread.h>

pthread_attr_t tattr;
int stacksize;
int ret;


/* setting a new size */
stacksize = (PTHREAD_STACK_MIN + 0x4000);
ret = pthread_attr_setstacksize(&tattr, stacksize);
```

In the example above, size contains the size, in number of bytes, for the stack that the new thread uses. If size is zero, a default size is used. In most cases, a zero value works best. `PTHREAD_STACK_MIN` is the amount of stack space required to start a thread. This does not take into consideration the threads routine requirements that are needed to execute application code.

The function `pthread_attr_getstacksize(pthread_attr_t *tattr, size_t *size)` may be used to inquire about a current threads stack size as follows:

```
#include <pthread.h>

pthread_attr_t tattr;
int stacksize;
int ret;
/* getting the stack size */
ret = pthread_attr_getstacksize(&tattr, &stacksize);
```

The function only returns the minimum stack size (in bytes) allocated for the created threads stack to the variable `stacksize`. **It DOES NOT RETURN the actual stack size** so use the function with care.

You may wish to specify the base adress of thread's stack. The function `pthread_attr_setstackaddr()` does this task. It is prototyped by:

```
int pthread_attr_setstackaddr(pthread_attr_t *tattr,void *stackaddr);
```

The `stackaddr` parameter defines the base of the thread's stack. If this is set to non-null (NULL is the default) the system initializes the stack at that address.

The function returns zero after completing successfully. Any other returned value indicates that an error occurred.

This example shows how to create a thread with both a custom stack address and a custom stack size.

```
#include <pthread.h>

pthread_attr_t tattr;
pthread_t tid;
int ret;
void *stackbase;
int size = PTHREAD_STACK_MIN + 0x4000;
stackbase = (void *) malloc(size);
/* initialized with default attributes */
ret = pthread_attr_init(&tattr);
/* setting the size of the stack also */
ret = pthread_attr_setstacksize(&tattr, size);
/* setting the base address in the attribute */
ret = pthread_attr_setstackaddr(&tattr, stackbase);
/* address and size specified */
```

```
ret = pthread_create(&tid, &tattr, func, arg);
```

The function `pthread_attr_getstackaddr(pthread_attr_t *tattr,void * *stackaddr)` can be used to obtain the base address for a current thread's stack address.

Further Threads Programming:Synchronization

When we multiple threads running they will invariably need to communicate with each other in order ***synchronise*** their execution. This chapter describes the synchronization types available with threads and discusses when and how to use synchronization.

There are a few possible methods of synchronising threads:

- Mutual Exclusion (Mutex) Locks
- Condition Variables
- Semaphores

We wil frequently make use of ***Synchronization objects***: these are variables in memory that you access just like data. Threads in different processes can communicate with each other through synchronization objects placed in threads-controlled shared memory, even though the threads in different processes are generally invisible to each other.

Synchronization objects can also be placed in files and can have lifetimes beyond that of the creating process.

Here are some example situations that require or can profit from the use of synchronization:

- When synchronization is the only way to ensure consistency of shared data.
- When threads in two or more processes can use a single synchronization object jointly. Note that the synchronization object should be initialized by only one of the cooperating processes, because reinitializing a synchronization object sets it to the unlocked state.
- When synchronization can ensure the safety of mutable data.
- When a process can map a file and have a thread in this process get a record's lock. Once the lock is acquired, any other thread in any process mapping the file that tries to acquire the lock is blocked until the lock is released.
- Even when accessing a single primitive variable, such as an integer. On machines where the integer is not aligned to the bus data width or is larger than the data width, a single memory load can use more than one memory cycle. While this cannot happen on the SPARC architectures, portable programs cannot rely on this.

Mutual Exclusion Locks

Mutual exclusion locks (mutexes) are a comon method of serializing thread execution. Mutual exclusion locks synchronize threads, usually by ensuring that only one thread at a time executes a critical section of code. Mutex locks can also preserve single-threaded code.

Mutex attributes may be associated with every thread. To change the default mutex attributes, you can declare and initialize an mutex attribute object and then alter specific values much like we have seen in the last chapter on more general POSIX attributes. Often, the mutex attributes are set in one place at the beginning of the application so they can be located quickly and modified easily.

After the attributes for a mutex are configured, you initialize the mutex itself. Functions are available to initialize or destroy, lock or unlock, or try to lock a mutex.

Initializing a Mutex Attribute Object

The function `pthread_mutexattr_init()` is used to initialize attributes associated with this object to their default values. It is prototyped by:

```
int pthread_mutexattr_init(pthread_mutexattr_t *mattr);
```

Storage for each attribute object is allocated by the threads system during execution. `mattr` is an opaque type that contains a system-allocated attribute object. The possible values of mattr's scope are `PTHREAD_PROCESS_PRIVATE` (the default) and `PTHREAD_PROCESS_SHARED`.The default value of the pshared attribute when this function is called is `PTHREAD_PROCESS_PRIVATE`, which means that the initialized mutex can be used within a process.

Before a mutex attribute object can be reinitialized, it must first be destroyed by `pthread_mutexattr_destroy()` (see below). The `pthread_mutexattr_init()` call returns a pointer to an opaque object. If the object is not destroyed, a memory leak will result. `pthread_mutexattr_init()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

A simple example of this function call is:

```
#include <pthread.h>

pthread_mutexattr_t mattr;
```

```
int ret;

/* initialize an attribute to default value */

ret = pthread_mutexattr_init(&mattr);
```

Destroying a Mutex Attribute Object

The function `pthread_mutexattr_destroy()` deallocates the storage space used to maintain the attribute object created by `pthread_mutexattr_init()`. It is prototyped by:

```
int pthread_mutexattr_destroy(pthread_mutexattr_t *mattr);
```

which returns zero after completing successfully. Any other returned value indicates that an error occurred.

The function is called as follows:

```
#include <pthread.h>

pthread_mutexattr_t mattr;
int ret;

/* destroy an attribute */
ret = pthread_mutexattr_destroy(&mattr);
```

The Scope of a Mutex

The scope of a mutex variable can be either process private (intraprocess) or system wide (interprocess). The function `pthread_mutexattr_setpshared()` is used to set the scope of a mutex atrribute and it is prototype as follows:

```
int pthread_mutexattr_setpshared(pthread_mutexattr_t *mattr, int pshared);
```

If the mutex is created with the `pshared` (POSIX) attribute set to the `PTHREAD_PROCESS_SHARED` state, and it exists in shared memory, it can be shared among threads from more than one process. This is equivalent to the `USYNC_PROCESS` flag in `mutex_init()` in Solaris threads. If the mutex `pshared` attribute is set to `PTHREAD_PROCESS_PRIVATE`, only those threads created by the same process can operate on the mutex. This is equivalent to the `USYNC_THREAD` flag in `mutex_init()` in Solaris threads.

`pthread_mutexattr_setpshared()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

A simple example call is:

```
#include <pthread.h>

pthread_mutexattr_t mattr;
int ret;

ret = pthread_mutexattr_init(&mattr);

/* resetting to its default value: private */
ret = pthread_mutexattr_setpshared(&mattr, PTHREAD_PROCESS_PRIVATE);
```

The function `pthread_mutexattr_getpshared(pthread_mutexattr_t *mattr, int *pshared)` may be used to obtain the scope of a current thread mutex as follows:

```
 #include <pthread.h>
pthread_mutexattr_t mattr;
int pshared, ret;

/* get pshared of mutex */ ret =
pthread_mutexattr_getpshared(&mattr, &pshared);
```

Initializing a Mutex

The function `pthread_mutex_init()` to initialize the mutex, it is prototyped by:

```
int pthread_mutex_init(pthread_mutex_t *mp, const pthread_mutexattr_t *mattr);
```

Here, `pthread_mutex_init()` initializes the mutex pointed at by `mp` to its default value if `mattr` is `NULL`, or to specify mutex attributes that have already been set with `pthread_mutexattr_init()`.

A mutex lock must not be reinitialized or destroyed while other threads might be using it. Program failure will result if either action is not done correctly. If a mutex is reinitialized or destroyed, the application must be sure the mutex is not currently in use. `pthread_mutex_init()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

A simple example call is:

```
#include <pthread.h>

pthread_mutex_t mp = PTHREAD_MUTEX_INITIALIZER;
pthread_mutexattr_t mattr;
int ret;
```

```
/* initialize a mutex to its default value */
ret = pthread_mutex_init(&mp, NULL);
```

When the mutex is initialized, it is in an unlocked state. The effect of `mattr` being `NULL` is the same as passing the address of a default mutex attribute object, but without the memory overhead. Statically defined mutexes can be initialized directly to have default attributes with the macro `PTHREAD_MUTEX_INITIALIZER`.

To initialise a mutex with non-default values do something like:

```
/* initialize a mutex attribute */
ret = pthread_mutexattr_init(&mattr);

/* change mattr default values with some function */
 ret = pthread_mutexattr_*();

/* initialize a mutex to a non-default value */
ret = pthread_mutex_init(&mp, &mattr);
```

Locking a Mutex

The function `pthread_mute_lock()` is used to lock a mutex, it is prototyped by:

```
int pthread_mutex_lock(pthread_mutex_t *mp);
```

`pthread_mute_lock()` locks the mutex pointed to by `mp`. When the mutex is already locked, the calling thread blocks and the mutex waits on a prioritized queue. When `pthread_mute_lock()` returns, the mutex is locked and the calling thread is the owner. `pthread_mute_lock()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

Therefor to lock a mutex `mp` on would do the following:

```
#include <pthread.h>
pthread_mutex_t mp;
int ret;

ret = pthread_mutex_lock(&mp);
```

To unlock a mutex use the function `pthread_mutex_unlock()` whose prototype is:

```
int pthread_mutex_unlock(pthread_mutex_t *mp);
```

Clearly, this function unlocks the mutex pointed to by `mp`.

The mutex must be locked and the calling thread **must** be the one that last locked the mutex (***i.e. the owner***). When any other threads are waiting for the mutex to become available, the thread at the head of the queue is unblocked. `pthread_mutex_unlock()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

A simple example call of `pthread_mutex_unlock()` is:

```
#include <pthread.h>

pthread_mutex_t mp;
int ret;

/* release the mutex */
ret = pthread_mutex_unlock(&mp);
```

Lock with a Nonblocking Mutex

The function `pthread_mutex_trylock()` to attempt to lock the mutex and is prototyped by:

```
int pthread_mutex_trylock(pthread_mutex_t *mp);
```

This function attempts to lock the mutex pointed to by `mp`. `pthread_mutex_trylock()` is a nonblocking version of `pthread_mutex_lock()`. When the mutex is already locked, this call returns with an error. Otherwise, the mutex is locked and the calling thread is the owner. `pthread_mutex_trylock()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

The function is called as follows:

```
#include <pthread.h>
pthread_mutex_t mp;

/* try to lock the mutex */
int ret; ret = pthread_ mutex_trylock(&mp);
```

Destroying a Mutex

The function `pthread_mutex_destroy()` may be used to destroy any state associated with the mutex. It is prototyped by:

```
int pthread_mutex_destroy(pthread_mutex_t *mp);
```

and destroys a mutex pointed to by mp.

Note: that the space for storing the mutex is not freed. `pthread_mutex_destroy()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

It is called by:

```
#include <pthread.h>
pthread_mutex_t mp;
int ret;

/* destroy mutex */
ret = pthread_mutex_destroy(&mp);
```

Mutex Lock Code Examples

Here are some code fragments showing mutex locking.

Mutex Lock Example

We develop two small functions that use the mutex lock for different purposes.

- The `increment_count function()` uses the mutex lock simply to ensure an atomic update of the shared variable, `count`.
- The `get_count()` function uses the mutex lock to guarantee that the (`long long`) 64-bit quantity count is read atomically. On a 32-bit architecture, a long long is really two 32-bit quantities.

The 2 functions are as follows:

```
#include <pthread.h>
pthread_mutex_t count_mutex;
long long count;

void increment_count()
   { pthread\_mutex\_lock(&count_mutex);
     count = count + 1;
     pthread_mutex_unlock(&count_mutex);
   }


long long get_count()
   { long long c;
     pthread\_mutex\_lock(&count_mutex);
     c = count;
     pthread_mutex_unlock(&count_mutex);
     return (c);
    }
```

Recall that reading an integer value is an atomic operation because integer is the common word size on most machines.

Using Locking Hierarchies: Avoiding Deadlock

You may occasionally want to access two resources at once. For instance, you are using one of the resources, and then discover that the other resource is needed as well. However, there could be a problem if two threads attempt to claim both resources but lock the associated mutexes in different orders.

In this example, if the two threads lock mutexes 1 and 2 respectively, then a ***deadlock*** occurs when each attempts to lock the other mutex.

Thread 1	**Thread 2**
`/* use resource 1 */`	`/* use resource 2 */`
`pthread_mutex_lock(&m1);`	`pthread_mutex_lock(&m2);`
`/* NOW use resources 2 + 1 */`	`/* NOW use resources 1 + 2 */`
`pthread_mutex_lock(&m2);`	`pthread_mutex_lock(&m1);`
`pthread_mutex_lock(&m1);`	`pthread_mutex_lock(&m2);`

The best way to avoid this problem is to make sure that whenever threads lock multiple mutexes, they do so in the same order. This technique is known as lock hierarchies: order the mutexes by logically assigning numbers to them. Also, honor the restriction that you cannot take a mutex that is assigned n when you are holding any mutex assigned a number greater than n.

Note: The `lock_lint` tool can detect the sort of deadlock problem shown in this example.

The best way to avoid such deadlock problems is to use lock hierarchies. When locks are always taken in a prescribed order, deadlock should not occur. However, this technique cannot always be used :

- sometimes you must take the mutexes in an order other than prescribed.
- To prevent deadlock in such a situation, use `pthread_mutex_trylock()`. One thread must release its mutexes when it discovers that deadlock would otherwise be inevitable.

The idea of ***Conditional Locking*** use this approach:

Thread 1:

```
pthread_mutex_lock(&m1);
pthread_mutex_lock(&m2);

/* no processing */
pthread_mutex_unlock(&m2);
pthread_mutex_unlock(&m1);
```

Thread 2:

```
for (; ;) {
  pthread_mutex_lock(&m2);
  if(pthread_mutex_trylock(&m1)==0)
    /* got it! */
    break;
 /* didn't get it */
 pthread_mutex_unlock(&m2);
 }
/* get locks; no processing */
pthread_mutex_unlock(&m1);
pthread_mutex_unlock(&m2);
```

In the above example, thread 1 locks mutexes in the prescribed order, but thread 2 takes them out of order. To make certain that there is no deadlock, thread 2 has to take mutex 1 very carefully; if it were to block waiting for the mutex to be released, it is likely to have just entered into a deadlock with thread 1. To ensure this does not happen, thread 2 calls `pthread_mutex_trylock()`, which takes the mutex if it is available. If it is not, thread 2 returns immediately, reporting failure. At this point, thread 2 must release mutex 2, so that thread 1 can lock it, and then release both mutex 1 and mutex 2.

Nested Locking with a Singly Linked List

We have met basic linked structues in Section 10.3, when using threads which share a linked list structure the possibility of deadlock may arise.

By nesting mutex locks into the linked data structure and a simple ammendment of the link list code we can prevent deadlock by taking the locks in a prescribed order.

The modified linked is as follows:

```
typedef struct node1 {
     int value;
     struct node1 *link;
     pthread_mutex_t lock;
   } node1_t;
```

Note: we simply ammend a standard singly-linked list structure so that each node containing a mutex.

Assuming we have created a variable `node1_t ListHead`.

To remove a node from the list:

- first search the list starting at ListHead (which itself is never removed) until the desired node is found.
- To protect this search from the effects of concurrent deletions, lock each node before any of its contents are accessed.

  Because all searches start at ListHead, there is never a deadlock because the locks are always taken in list order.

- When the desired node is found, lock both the node and its predecessor since the change involves both nodes.

  Because the predecessor's lock is always taken first, you are again protected from deadlock.

The C code to remove an item from a singly linked list with nested locking is as follows:

```
node1_t *delete(int value)
  { node1_t *prev,
    *current; prev = &ListHead;

   pthread_mutex_lock(&prev->lock);
   while ((current = prev->link) != NULL)
     { pthread_mutex_lock(&current->lock);
       if (current->value == value)
         { prev->link = current->link;
           pthread_mutex_unlock(&current->lock);
```

```
          pthread_mutex_unlock(&prev->lock);
          current->link = NULL; return(current);
        }
      pthread_mutex_unlock(&prev->lock);
      prev = current;
    }
  pthread_mutex_unlock(&prev->lock);
  return(NULL);
}
```

Solaris Mutex Locks

Similar mutual exclusion locks exist for in Solaris.

You should include the <synch.h> or <thread.h>libraries.

To initialize a mutex use `int mutex_init(mutex_t *mp, int type, void *arg))`. `mutex_init()` initializes the mutex pointed to by `mp`. The `type` can be one of the following (note that `arg` is currently ignored).

USYNC_PROCESS
-- The mutex can be used to synchronize threads in this and other processes.

USYNC_THREAD
-- The mutex can be used to synchronize threads in this process, only.

Mutexes can also be initialized by allocation in zeroed memory, in which case a type of `USYNC_THREAD` is assumed. Multiple threads must not initialize the same mutex simultaneously. A mutex lock must not be reinitialized while other threads might be using it.

The function `int mutex_destroy (mutex_t *mp)` destroys any state associated with the mutex pointed to by `mp`. **Note** that the space for storing the mutex is not freed.

To acquire a mutex lock use the function `mutex_lock(mutex_t *mp)` which locks the mutex pointed to by mp. When the mutex is already locked, the calling thread blocks until the mutex becomes available (blocked threads wait on a prioritized queue).

To release a mutex use `mutex_unlock(mutex_t *mp)` which unlocks the mutex pointed to by mp. The mutex must be locked and the calling thread must be the one that last locked the mutex (the owner).

To try to acquire a mutex use mutex_trylock(mutex_t *mp) to attempt to lock the mutex pointed to by `mp`. This function is a nonblocking version of `mutex_lock()`

Condition Variable Attributes

Condition variables can be usedto atomically block threads until a particular condition is true. Condition variables are ***always*** used in conjunction with mutex locks:

- With a condition variable, a thread can atomically block until a condition is satisfied.
- The condition is tested under the protection of a mutual exclusion lock (mutex).
  - When the condition is false, a thread usually blocks on a condition variable and atomically releases the mutex waiting for the condition to change.
  - When another thread changes the condition, it can signal the associated condition variable to cause one or more waiting threads to wake up, acquire the mutex again, and reevaluate the condition.

Condition variables can be used to synchronize threads among processes when they are allocated in memory that can be written to and is shared by the cooperating processes.

The scheduling policy determines how blocking threads are awakened. For the default `SCHED_OTHER`, threads are awakened in priority order. The attributes for condition variables must be set and initialized before the condition variables can be used.

As with mutex locks, The condiotion variable attributes must be initialised and set (or set to `NULL`) before an actual condition variable may be initialise (with appropriat attributes) and then used.

Initializing a Condition Variable Attribute

The function `pthread_condattr_init()` initializes attributes associated with this object to their default values. It is prototyped by:

```
int pthread_condattr_init(pthread_condattr_t *cattr);
```

Storage for each attribute object, `cattr`, is allocated by the threads system during execution. `cattr` is an opaque data type that contains a system-allocated attribute object. The possible values of cattr's scope are `PTHREAD_PROCESS_PRIVATE` and `PTHREAD_PROCESS_SHARED`. The default value of the pshared attribute when this function is called is `PTHREAD_PROCESS_PRIVATE`, which means that the initialized condition variable can be used within a process.

Before a condition variable attribute can be reused, it must first be reinitialized by `pthread_condattr_destroy()`. The `pthread_condattr_init()` call returns a pointer to an opaque object. If the object is not destroyed, a memory leak will result.

`pthread_condattr_init()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When either of the following conditions occurs, the function fails and returns the corresponding value.

A simple example call of this function is :

```
#include <pthread.h>

pthread_condattr_t cattr;
int ret;

/* initialize an attribute to default value */
ret = pthread_condattr_init(&cattr);
```

Destoying a Condition Variable Attribute

The function `pthread_condattr_destroy()` removes storage and renders the attribute object invalid, it is prototyped by:

```
int pthread_condattr_destroy(pthread_condattr_t *cattr);
```

`pthread_condattr_destroy()` returns zero after completing successfully and destroying the condition variable pointed to by `cattr`. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

The Scope of a Condition Variable

The scope of a condition variable can be either process private (intraprocess) or system wide (interprocess), as with mutex locks. If the condition variable is created with the pshared attribute set to the `PTHREAD_PROCESS_SHARED` state, and it exists in shared memory, it can be shared among threads from more than one process. This is equivalent to the `USYNC_PROCESS` flag in `mutex_init()` in the original Solaris threads. If the mutex pshared attribute is set to `PTHREAD_PROCESS_PRIVATE` (default value), only those threads created by the same process can operate on the mutex. Using `PTHREAD_PROCESS_PRIVATE` results in the same behavior as with the `USYNC_THREAD` flag in the original Solaris threads `cond_init()` call, which is that of a local condition variable. `PTHREAD_PROCESS_SHARED` is equivalent to a global condition variable.

The function `pthread_condattr_setpshared()` is used to set the scope of a condition variable, it is prototyped by:

```
int pthread_condattr_setpshared(pthread_condattr_t *cattr, int pshared);
```

The condition variable attribute `cattr` must be initialised first and the value of `pshared` is either `PTHREAD_PROCESS_SHARED` or `PTHREAD_PROCESS_PRIVATE`.

`pthread_condattr_setpshared()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

A sample use of this function is as follows:

```
#include <pthread.h>

pthread_condattr_t cattr;
int ret;

/* Scope: all processes */
ret = pthread_condattr_setpshared(&cattr, PTHREAD_PROCESS_SHARED);

/* OR */
/* Scope: within a process */
ret = pthread_condattr_setpshared(&cattr, PTHREAD_PROCESS_PRIVATE);
```

The function `int pthread_condattr_getpshared(const pthread_condattr_t *cattr, int *pshared)` may be used to obtain the scope of a given condition variable.

Initializing a Condition Variable

The function `pthread_cond_init()` initializes the condition variable and is prototyped as follows:

```
int pthread_cond_init(pthread_cond_t *cv, const pthread_condattr_t *cattr);
```

The condition variable which is initialized is pointed at by `cv` and is set to its default value if `cattr` is `NULL`, or to specific `cattr` condition variable attributes that are already set with `pthread_condattr_init()`. The effect of `cattr` being NULL is the same as passing the address of a default condition variable attribute object, but without the memory overhead.

Statically-defined condition variables can be initialized directly to have default attributes with the macro `PTHREAD_COND_INITIALIZER`. This has the same effect as dynamically allocating `pthread_cond_init()` with null attributes. No error checking is done. Multiple threads must not simultaneously initialize or reinitialize the same condition variable. If a condition variable is reinitialized or destroyed, the application must be sure the condition variable is not in use.

`pthread_cond_init()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

Sample calls of this function are:

```
#include <pthread.h>

pthread_cond_t cv;
pthread_condattr_t cattr;
```

```
int ret;

/* initialize a condition variable to its default value */
ret = pthread_cond_init(&cv, NULL);

/* initialize a condition variable */ ret =
pthread_cond_init(&cv, &cattr);
```

Block on a Condition Variable

The function `pthread_cond_wait()` is used to atomically release a mutex and to cause the calling thread to block on the condition variable. It is protoyped by:

```
int pthread_cond_wait(pthread_cond_t *cv,pthread_mutex_t *mutex);
```

The mutex that is released is pointed to by `mutex` and the condition variable pointed to by `cv` is blocked.

`pthread_cond_wait()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

A simple example call is:

```
#include <pthread.h>

pthread_cond_t cv;
pthread_mutex_t mutex;
int ret;

/* wait on condition variable */
ret = pthread_cond_wait(&cv, &mutex);
```

The blocked thread can be awakened by a `pthread_cond_signal()`, a `pthread_cond_broadcast()`, or when interrupted by delivery of a signal. Any change in the value of a condition associated with the condition variable cannot be inferred by the return of `pthread_cond_wait()`, and any such condition must be reevaluated. The `pthread_cond_wait()` routine always returns with the mutex locked and owned by the calling thread, even when returning an error. This function blocks until the condition is signaled. It atomically releases the associated mutex lock before blocking, and atomically acquires it again before returning. In typical use, a condition expression is evaluated under the protection of a mutex lock. When the condition expression is false, the thread blocks on the condition variable. The condition variable is then signaled by another thread when it changes the condition value. This causes one or all of the threads waiting on the condition to unblock and to try to acquire the mutex lock again. Because the condition can change before an awakened thread returns from `pthread_cond_wait()`, the condition that caused the wait must be retested before the mutex lock is acquired.

The recommended test method is to write the condition check as a while loop that calls `pthread_cond_wait()`, as follows:

```
pthread_mutex_lock();

while(condition_is_false)
  pthread_cond_wait();
pthread_mutex_unlock();
```

No specific order of acquisition is guaranteed when more than one thread blocks on the condition variable. Note also that `pthread_cond_wait()` is a cancellation point. If a cancel is pending and the calling thread has cancellation enabled, the thread terminates and begins executing its cleanup handlers while continuing to hold the lock.

To unblock a specific thread use `pthread_cond_signal()` which is prototyped by:

```
int pthread_cond_signal(pthread_cond_t *cv);
```

This unblocks one thread that is blocked on the condition variable pointed to by `cv`. `pthread_cond_signal()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

You should always call `pthread_cond_signal()` under the protection of the same mutex used with the condition variable being signaled. Otherwise, the condition variable could be signaled between the test of the associated condition and blocking in `pthread_cond_wait()`, which can cause an infinite wait. The scheduling policy determines the order in which blocked threads are awakened. For `SCHED_OTHER`, threads are awakened in priority order. When no threads are blocked on the condition variable, then calling `pthread_cond_signal()`l has no effect.

The folloowing code fragment illustrates how to avoid an infinite problem described above:

```
pthread_mutex_t count_lock;
pthread_cond_t count_nonzero;
unsigned count;

decrement_count()
   { pthread_mutex_lock(&count_lock);

     while (count == 0)
        pthread_cond_wait(&count_nonzero, &count_lock);
     count = count - 1;
     pthread_mutex_unlock(&count_lock);
   }

increment_count()
```

```
   { pthread_mutex_lock(&count_lock);
     if (count == 0)
       pthread_cond_signal(&count_nonzero);
     count = count + 1;
     pthread_mutex_unlock(&count_lock);
    }
```

You can also block until a specified event occurs. The function `pthread_cond_timedwait()` is used for this purpose. It is prototyped by:

```
int pthread_cond_timedwait(pthread_cond_t *cv,
      pthread_mutex_t *mp, const struct timespec *abstime);
```

`pthread_cond_timedwait()` is used in a similar manner to `pthread_cond_wait()`: `pthread_cond_timedwait()` blocks until the condition is signaled or until the time of day, specified by `abstime`, has passed. `pthread_cond_timedwait()` always returns with the mutex, `mp`, locked and owned by the calling thread, even when it is returning an error. `pthread_cond_timedwait()` is also a cancellation point.

`pthread_cond_timedwait()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When either of the following conditions occurs, the function fails and returns the corresponding value.

An examle call of this function is:

```
#include <pthread.h>
#include <time.h>
pthread_timestruc_t to;
pthread_cond_t cv;
pthread_mutex_t mp;
timestruct_t abstime;
int ret;

/* wait on condition variable */

ret = pthread_cond_timedwait(&cv, &mp, &abstime);


pthread_mutex_lock(&m);
to.tv_sec = time(NULL) + TIMEOUT;
to.tv_nsec = 0;

while (cond == FALSE)
  {  err = pthread_cond_timedwait(&c, &m, &to);
     if (err == ETIMEDOUT)
       { /* timeout, do something */
         break;
       }
   }
pthread_mutex_unlock(&m);
```

All threads may be unblocked in one function: `pthread_cond_broadcast()`. This function is prototyped as follows:

```
int pthread_cond_broadcast(pthread_cond_t *cv);
```

`pthread_cond_broadcast()` unblocks all threads that are blocked on the condition variable pointed to by `cv`, specified by `pthread_cond_wait()`. When no threads are blocked on the condition variable, `pthread_cond_broadcast()` has no effect.

`pthread_cond_broadcast()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

Since `pthread_cond_broadcast()` causes all threads blocked on the condition to contend again for the mutex lock, use carefully. For example, use `pthread_cond_broadcast()` to allow threads to contend for varying resource amounts when resources are freed:

```
#include <pthread.h>

pthread_mutex_t rsrc_lock;
pthread_cond_t rsrc_add;
unsigned int resources;

get_resources(int amount)
  { pthread_mutex_lock(&rsrc_lock);
    while (resources < amount)
       pthread_cond_wait(&rsrc_add, &rsrc_lock);

   resources -= amount;
   pthread_mutex_unlock(&rsrc_lock);
  }

add_resources(int amount)
  { pthread_mutex_lock(&rsrc_lock);
    resources += amount;
    pthread_cond_broadcast(&rsrc_add);
    pthread_mutex_unlock(&rsrc_lock);
  }
```

Note: that in `add_resources` it does not matter whether resources is updated first or if `pthread_cond_broadcast()` is called first inside the mutex lock. Call `pthread_cond_broadcast()` under the protection of the same mutex that is used with the condition

variable being signaled. Otherwise, the condition variable could be signaled between the test of the associated condition and blocking in `pthread_cond_wait()`, which can cause an infinite wait.

Destroying a Condition Variable State

The function `pthread_cond_destroy()` to destroy any state associated with the condition variable, it is prototyped by:

```
int pthread_cond_destroy(pthread_cond_t *cv);
```

The condition variable pointed to by `cv` will be destroyed by this call:

```
#include <pthread.h>

pthread_cond_t cv;
int ret;

/* Condition variable is destroyed */
ret = pthread_cond_destroy(&cv);
```

Note that the space for storing the condition variable is not freed.

`pthread_cond_destroy()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, the function fails and returns the corresponding value.

Solaris Condition Variables

Similar condition variables exist in Solaris. The functions are prototyped in <thread.h>.

To initialize a condition variable use `int cond_init(cond_t *cv, int type, int arg)` which initializes the condition variable pointed to by `cv`. The `type` can be one of `USYNC_PROCESS` or `USYNC_THREAD` (See Solaris mutex (Section 30.1.9 for more details). Note that `arg` is currently ignored.

Condition variables can also be initialized by allocation in zeroed memory, in which case a type of `USYNC_THREAD` is assumed. Multiple threads must not initialize the same condition variable simultaneously. A condition variable must not be reinitialized while other threads might be using it.

To destroy a condition variable use `int cond_destroy(cond_t *cv)` which destroys a state associated with the condition variable pointed to by `cv`. The space for storing the condition variable is not freed.

To wait for a condition use `int cond_wait(cond_t *cv, mutex_t *mp)` which atomically releases the mutex pointed to by `mp` and to cause the calling thread to block on the condition variable pointed to by `cv`.

The blocked thread can be awakened by `cond_signal(cond_t *cv)`, `cond_broadcast(cond_t *cv)`, or when interrupted by delivery of a signal or a fork. Use `cond_signal()` to unblock one thread that is blocked on the condition variable pointed to by `cv`. Call this function under protection of the same mutex used with the condition variable being signaled. Otherwise, the condition could be signaled between its test and `cond_wait()`, causing an infinite wait. Use `cond_broadcast()` to unblock all threads that are blocked on the condition variable pointed to by `cv`. When no threads are blocked on the condition variable then `cond_broadcast()` has no effect.

Finally, to wait until the condition is signaled or for an absolute time use `int cond_timedwait(cond_t *cv, mutex_t *mp, timestruct_t abstime)` Use `cond_timedwait()` as you would use `cond_wait()`, except that `cond_timedwait()` does not block past the time of day specified by abstime. `cond_timedwait()` always returns with the mutex locked and owned by the calling thread even when returning an error.

Threads and Semaphores

POSIX Semaphores

Chapter 25 has dealt with semaphore programming for POSIX and System V IPC semaphores.

Semaphore operations are the same in both POSIX and Solaris. The function names are changed from `sema_` in Solaris to `sem_` in pthreads. Solaris semaphore are defined in <thread.h>.

In this section we give a brief description of Solaris thread semaphores.

Basic Solaris Semaphore Functions

To initialize the function `int sema_init(sema_t *sp, unsigned int count, int type, void *arg)` is used. `sema`. `type` can be one of the following):

`USYNC_PROCESS`
: -- The semaphore can be used to synchronize threads in this process and other processes. Only one process should initialize the semaphore.

`USYNC_THREAD`

-- The semaphore can be used to synchronize threads in this process.

`arg` is currently unused.

Multiple threads **must not** initialize the same semaphore simultaneously. A semaphore **must not** be reinitialized while other threads may be using it.

To increment a Semaphore use the function `int sema_post(sema_t *sp)`. `sema_post` atomically increments the semaphore pointed to by `sp`. When any threads are blocked on the semaphore, one is unblocked.

To block on a Semaphore use `int sema_wait(sema_t *sp)`. `sema_wait()` to block the calling thread until the count in the semaphore pointed to by `sp` becomes greater than zero, then atomically decrement it.

To decrement a Semaphore count use `int sema_trywait(sema_t *sp)`. `sema_trywait()` atomically decrements the count in the semaphore pointed to by `sp` when the count is greater than zero. This function is a nonblocking version of `sema_wait()`.

To destroy the Semaphore state call the function `sema_destroy(sema_t *sp)`. `sema_destroy()` to destroy any state associated with the semaphore pointed to by `sp`. The space for storing the semaphore is not freed.

Thread programming examples

This chapter gives some full code examples of thread programs. These examles are taken from a variety of sources:

- The sun workshop developers web page ***http://www.sun.com/workshop/threads/share-code/*** on threads is an excelleny source
- The web page ***http://www.sun.com/workshop/threads/Berg-Lewis/examples.html*** where example from the ***Threads Primer*** Book by D. Berg anD B. Lewis are also a major resource.

Using `thr_create()` and `thr_join()`

This example exercises the `thr_create()` and `thr_join()` calls. There is not a parent/child relationship between threads as there is for processes. This can easily be seen in this example, because threads are created and joined by many different threads in the process. The example also shows how threads behave when created with different attributes and options.

Threads can be created by any thread and joined by any other.

The main thread: In this example the main thread's sole purpose is to create new threads. Threads A, B, and C are created by the main thread. Notice that thread B is created suspended. After creating the new threads, the main thread exits. Also notice that the main thread exited by calling thr_exit(). If the main thread had used the exit() call, the whole process would have exited. The main thread's exit status and resources are held until it is joined by thread C.

Thread A: The first thing thread A does after it is created is to create thread D. Thread A then simulates some processing and then exits, using `thr_exit()`. Notice that thread A was created with the `THR_DETACHED` flag, so thread A's resources will be immediately reclaimed upon its exit. There is no way for thread A's exit status to be collected by a `thr_join()` call.

Thread B: Thread B was created in a suspended state, so it is not able to run until thread D continues it by making the `thr_continue()` call. After thread B is continued, it simulates some processing and then exits. Thread B's exit status and thread resources are held until joined by thread E.

Thread C: The first thing that thread C does is to create thread F. Thread C then joins the main thread. This action will collect the main thread's exit status and allow the main thread's resources to be reused by another thread. Thread C will block, waiting for the main thread to exit, if the main thread has not yet called `thr_exit()`. After joining the main thread, thread C will simulate some processing and then exit. Again, the exit status and thread resources are held until joined by thread E.

Thread D: Thread D immediately creates thread E. After creating thread E, thread D continues thread B by making the `thr_continue()` call. This call will allow thread B to start its execution. Thread D then tries to join thread E, blocking until thread E has exited. Thread D then simulates some processing and exits. If all went well, thread D should be the last nondaemon thread running. When thread D exits, it should do two things: stop the execution of any daemon threads and stop the execution of the process.

Thread E: Thread E starts by joining two threads, threads B and C. Thread E will block, waiting for each of these thread to exit. Thread E will then simulate some processing and will exit. Thread E's exit status and thread resources are held by the operating system until joined by thread D.

Thread F: Thread F was created as a bound, daemon thread by using the `THR_BOUND` and `THR_DAEMON` flags in the `thr_create()` call. This means that it will run on its own LWP until all the nondaemon threads have exited the process. This type of thread can be used when you want some type of "background" processing to always be running, except when all the "regular" threads have exited the process. If thread F was created as a non-daemon thread, then it would continue to run forever, because a process will continue while

there is at least one thread still running. Thread F will exit when all the nondaemon threads have exited. In this case, thread D should be the last nondaemon thread running, so when thread D exits, it will also cause thread F to exit.

This example, however trivial, shows how threads behave differently, based on their creation options. It also shows what happens on the exit of a thread, again based on how it was created. If you understand this example and how it flows, you should have a good understanding of how to use `thr_create()` and `thr_join()` in your own programs. Hopefully you can also see how easy it is to create and join threads.

The source to `multi_thr.c`:

```
#define _REENTRANT
#include <stdio.h>
#include <thread.h>

/* Function prototypes for thread routines */
void *sub_a(void *);
void *sub_b(void *);
void *sub_c(void *);
void *sub_d(void *);
void *sub_e(void *);
void *sub_f(void *);

thread_t thr_a, thr_b, thr_c;

void main()
{
thread_t main_thr;
main_thr = thr_self();
printf("Main thread = %d\n", main_thr);

if (thr_create(NULL, 0, sub_b, NULL, THR_SUSPENDED|THR_NEW_LWP, &thr_b))
        fprintf(stderr,"Can't create thr_b\n"), exit(1);

if (thr_create(NULL, 0, sub_a, (void *)thr_b, THR_NEW_LWP, &thr_a))
        fprintf(stderr,"Can't create thr_a\n"), exit(1);

if (thr_create(NULL, 0, sub_c, (void *)main_thr, THR_NEW_LWP, &thr_c))
        fprintf(stderr,"Can't create thr_c\n"), exit(1);

printf("Main Created threads A:%d B:%d C:%d\n", thr_a, thr_b, thr_c);
printf("Main Thread exiting...\n");
thr_exit((void *)main_thr);
}

void *sub_a(void *arg)
{
thread_t thr_b = (thread_t) arg;
thread_t thr_d;
int i;

printf("A: In thread A...\n");

if (thr_create(NULL, 0, sub_d, (void *)thr_b, THR_NEW_LWP, &thr_d))
        fprintf(stderr, "Can't create thr_d\n"), exit(1);

printf("A: Created thread D:%d\n", thr_d);

/* process
*/
for (i=0;i<1000000*(int)thr_self();i++);
printf("A: Thread exiting...\n");
thr_exit((void *)77);
}

void * sub_b(void *arg)
{
int i;

printf("B: In thread B...\n");

/* process
*/

for (i=0;i<1000000*(int)thr_self();i++);
printf("B: Thread exiting...\n");
thr_exit((void *)66);
}

void * sub_c(void *arg)
{
void *status;
int i;
thread_t main_thr, ret_thr;

main_thr = (thread_t)arg;

printf("C: In thread C...\n");

if (thr_create(NULL, 0, sub_f, (void *)0, THR_BOUND|THR_DAEMON, NULL))
        fprintf(stderr, "Can't create thr_f\n"), exit(1);

printf("C: Join main thread\n");
```

```
if (thr_join(main_thr,(thread_t *)&ret_thr, &status))
        fprintf(stderr, "thr_join Error\n"), exit(1);

printf("C: Main thread (%d) returned thread (%d) w/status %d\n", main_thr, ret_thr, (int) status);

/* process
*/

for (i=0;i<1000000*(int)thr_self();i++);
printf("C: Thread exiting...\n");
thr_exit((void *)88);
}


void * sub_d(void *arg)
{
thread_t thr_b = (thread_t) arg;
int i;
thread_t thr_e, ret_thr;
void *status;

printf("D: In thread D...\n");

if (thr_create(NULL, 0, sub_e, NULL, THR_NEW_LWP, &thr_e))
        fprintf(stderr,"Can't create thr_e\n"), exit(1);

printf("D: Created thread E:%d\n", thr_e);
printf("D: Continue B thread = %d\n", thr_b);
thr_continue(thr_b);
printf("D: Join E thread\n");

if(thr_join(thr_e,(thread_t *)&ret_thr, &status))
        fprintf(stderr,"thr_join Error\n"), exit(1);

printf("D: E thread (%d) returned thread (%d) w/status %d\n", thr_e,
ret_thr, (int) status);

/* process
*/

for (i=0;i<1000000*(int)thr_self();i++);
printf("D: Thread exiting...\n");
thr_exit((void *)55);
}


void * sub_e(void *arg)
{
int i;
thread_t ret_thr;
void *status;

printf("E: In thread E...\n");
printf("E: Join A thread\n");

if(thr_join(thr_a,(thread_t *)&ret_thr, &status))
        fprintf(stderr,"thr_join Error\n"), exit(1);

printf("E: A thread (%d) returned thread (%d) w/status %d\n", ret_thr, ret_thr, (int) status);
printf("E: Join B thread\n");

if(thr_join(thr_b,(thread_t *)&ret_thr, &status))
        fprintf(stderr,"thr_join Error\n"), exit(1);
printf("E: B thread (%d) returned thread (%d) w/status %d\n", thr_b, ret_thr, (int) status);
printf("E: Join C thread\n");

if(thr_join(thr_c,(thread_t *)&ret_thr, &status))
        fprintf(stderr,"thr_join Error\n"), exit(1);

printf("E: C thread (%d) returned thread (%d) w/status %d\n", thr_c, ret_thr, (int) status);

for (i=0;i<1000000*(int)thr_self();i++);

printf("E: Thread exiting...\n");
thr_exit((void *)44);
}


void *sub_f(void *arg)
{
int i;

printf("F: In thread F...\n");

while (1) {
        for (i=0;i<10000000;i++);
        printf("F: Thread F is still running...\n");
        }
}
```

Arrays

This example uses a data structure that contains multiple arrays of data. Multiple threads will concurrently vie for access to the arrays. To control this access, a mutex variable is used within the data structure to lock the entire array and serialize the access to the data.

The main thread first initializes the data structure and the mutex variable. It then sets a level of concurrency and creates the worker threads. The main thread then blocks by joining all the threads. When all the threads have exited, the main thread prints the results.

The worker threads modify the shared data structure from within a loop. Each time the threads need to modify the shared data, they lock the mutex variable associated with the shared data. After modifying the data, the threads unlock the mutex, allowing another thread access to the data.

This example may look quite simple, but it shows how important it is to control access to a simple, shared data structure. The results can be quite different if the mutex variable is not used.

The source to `array.c`:

```
#define _REENTRANT
#include <stdio.h>
#include <thread.h>

/* sample array data structure */
struct {
        mutex_t data_lock[5];
        int     int_val[5];
        float   float_val[5];
        } Data;

/* thread function */
void *Add_to_Value();

main()
{
int i;

/* initialize the mutexes and data */
for (i=0; i<5; i++) {
        mutex_init(&Data.data_lock[i], USYNC_THREAD, 0);
        Data.int_val[i] = 0;
        Data.float_val[i] = 0;
        }

/* set concurrency and create the threads */
thr_setconcurrency(4);

for (i=0; i<5; i++)
    thr_create(NULL, 0, Add_to_Value, (void *)(2*i), 0, NULL);

/* wait till all threads have finished */
for (i=0; i<5; i++)
        thr_join(0,0,0);

/* print the results */
printf("Final Values.....\n");

for (i=0; i<5; i++) {
        printf("integer value[%d] =\t%d\n", i, Data.int_val[i]);
        printf("float value[%d] =\t%.0f\n\n", i, Data.float_val[i]);
        }

return(0);
}


/* Threaded routine */
void *Add_to_Value(void *arg)
{
int inval = (int) arg;
int i;

for (i=0;i<10000;i++){
    mutex_lock(&Data.data_lock[i%5]);
       Data.int_val[i%5] += inval;
       Data.float_val[i%5] += (float) 1.5 * inval;
    mutex_unlock(&Data.data_lock[i%5]);
    }

return((void *)0);
}
```

Deadlock

This example demonstrates how a deadlock can occur in multithreaded programs that use synchronization variables. In this example a thread is created that continually adds a value to a global variable. The thread uses a mutex lock to protect the global data.

The main thread creates the counter thread and then loops, waiting for user input. When the user presses the Return key, the main thread suspends the counter thread and then prints the value of the global variable. The main thread prints the value of the global variable under the protection of a mutex lock.

The problem arises in this example when the main thread suspends the counter thread while the counter thread is holding the mutex lock. After the main thread suspends the counter thread, it tries to lock the mutex variable. Since the mutex variable is already held by the counter thread, which is suspended, the main thread deadlocks.

This example may run fine for a while, as long as the counter thread just happens to be suspended when it is not holding the mutex lock. The example demonstrates how tricky some programming issues can be when you deal with threads.

The source to `susp_lock.c`

```
#define _REENTRANT
#include <stdio.h>
#include <thread.h>

/* Prototype for thread subroutine */
void *counter(void *);

int count;
mutex_t count_lock;

main()
{
char str[80];
thread_t ctid;

/* create the thread counter subroutine */
thr_create(NULL, 0, counter, 0, THR_NEW_LWP|THR_DETACHED, &ctid);

while(1) {
        gets(str);
        thr_suspend(ctid);

        mutex_lock(&count_lock);
        printf("\n\nCOUNT = %d\n\n", count);
        mutex_unlock(&count_lock);

        thr_continue(ctid);
        }

return(0);
}

void *counter(void *arg)
{
int i;

while (1) {
        printf("."); fflush(stdout);

        mutex_lock(&count_lock);
        count++;

        for (i=0;i<50000;i++);

        mutex_unlock(&count_lock);

        for (i=0;i<50000;i++);
        }

return((void *)0);
}
```

Signal Handler

This example shows how easy it is to handle signals in multithreaded programs. In most programs, a different signal handler would be needed to service each type of signal that you wanted to catch. Writing each of the signal handlers can be time consuming and can be a real pain to debug.

This example shows how you can implement a signal handler thread that will service all asynchronous signals that are sent to your process. This is an easy way to deal with signals, because only one thread is needed to handle all the signals. It also makes it easy when you create new threads within the process, because you need not worry about signals in any of the threads.

First, in the main thread, mask out all signals and then create a signal handling thread. Since threads inherit the signal mask from their creator, any new threads created after the new signal mask will also mask all signals. This idea is key, because the only thread that will receive signals is the one thread that does not block all the signals.

The signal handler thread waits for all incoming signals with the sigwait() call. This call unmasks the signals given to it and then blocks until a signal arrives. When a signal arrives, sigwait() masks the signals again and then returns with the signal ID of the incoming signal.

You can extend this example for use in your application code to handle all your signals. Notice also that this signal concept could be added in your existing nonthreaded code as a simpler way to deal with signals.

The source to `thr_sig.c`

```
#define _REENTRANT
#include <stdio.h>
#include <thread.h>
#include <signal.h>
#include <sys/types.h>

void *signal_hand(void *);

main()
{
sigset_t set;

/* block all signals in main thread.  Any other threads that are
   created after this will also block all signals */

sigfillset(&set);

thr_sigsetmask(SIG_SETMASK, &set, NULL);

/* create a signal handler thread.  This thread will catch all
   signals and decide what to do with them.  This will only
   catch nondirected signals.  (I.e., if a thread causes a SIGFPE
   then that thread will get that signal. */

thr_create(NULL, 0, signal_hand, 0, THR_NEW_LWP|THR_DAEMON|THR_DETACHED, NULL);

while (1) {
        /*
        Do your normal processing here....
        */
        }  /* end of while */

return(0);
}

void *signal_hand(void *arg)
{
sigset_t set;
int sig;

sigfillset(&set); /* catch all signals */

while (1) {
        /* wait for a signal to arrive */

        switch (sig=sigwait(&set)) {

          /* here you would add whatever signal you needed to catch */
          case SIGINT : {
                        printf("Interrupted with signal %d, exiting...\n", sig);
                        exit(0);
                        }

          default : printf("GOT A SIGNAL = %d\n", sig);
          } /* end of switch */
        } /* end of while */

return((void *)0);
} /* end of signal_hand */
```

Another example of a signal handler, `sig_kill.c`:

```
/*
*  Multithreaded Demo Source
*
*  Copyright (C) 1995 by Sun Microsystems, Inc.
*  All rights reserved.
*
*  This file is a product of SunSoft, Inc. and is provided for
*  unrestricted use provided that this legend is included on all
*  media and as a part of the software program in whole or part.
*  Users may copy, modify or distribute this file at will.
*
*  THIS FILE IS PROVIDED AS IS WITH NO WARRANTIES OF ANY KIND INCLUDING
*  THE WARRANTIES OF DESIGN, MERCHANTIBILITY AND FITNESS FOR A PARTICULAR
*  PURPOSE, OR ARISING FROM A COURSE OF DEALING, USAGE OR TRADE PRACTICE.
*
*  This file is provided with no support and without any obligation on the
*  part of SunSoft, Inc. to assist in its use, correction, modification or
*  enhancement.
*
*  SUNSOFT AND SUN MICROSYSTEMS, INC. SHALL HAVE NO LIABILITY WITH RESPECT
*  TO THE INFRINGEMENT OF COPYRIGHTS, TRADE SECRETS OR ANY PATENTS BY THIS
*  FILE OR ANY PART THEREOF.
*
*  IN NO EVENT WILL SUNSOFT OR SUN MICROSYSTEMS, INC. BE LIABLE FOR ANY
*  LOST REVENUE OR PROFITS OR OTHER SPECIAL, INDIRECT AND CONSEQUENTIAL
*  DAMAGES, EVEN IF THEY HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH
*  DAMAGES.
*
*  SunSoft, Inc.
*  2550 Garcia Avenue
```

```
*  Mountain View, California  94043
*/

/*
 * Rich Schiavi writes:               Sept 11, 1994
 *
 * I believe the recommended way to kill certain threads is
 * using a signal handler which then will exit that particular
 * thread properly. I'm not sure the exact reason (I can't remember), but
 * if you take out the signal_handler routine in my example, you will see what
 * you describe, as the main process dies even if you send the
 * thr_kill to the specific thread.

 * I whipped up a real quick simple example which shows this using
 * some sleep()s to get a good simulation.
 */

#include <stdio.h>
#include <thread.h>
#include <signal.h>

static  thread_t       one_tid, two_tid, main_thread;
static  void    *first_thread();
static  void    *second_thread();
void            ExitHandler(int);

static  mutex_t        first_mutex, second_mutex;
int     first_active = 1 ;
int     second_active = 1;

main()

{
  int i;
  struct sigaction act;

  act.sa_handler = ExitHandler;
  (void) sigemptyset(&act.sa_mask);
  (void) sigaction(SIGTERM, &act, NULL);

  mutex_init(&first_mutex, 0 , 0);
  mutex_init(&second_mutex, 0 , 0);
  main_thread = thr_self();

  thr_create(NULL,0,first_thread,0,THR_NEW_LWP,&one_tid);
  thr_create(NULL,0,second_thread,0,THR_NEW_LWP,&two_tid);

  for (i = 0; i < 10; i++){
    fprintf(stderr, "main loop: %d\n", i);
    if (i == 5){
      thr_kill(one_tid, SIGTERM);
    }
    sleep(3);
  }
  thr_kill(two_tid, SIGTERM);
  sleep(5);
  fprintf(stderr, "main exit\n");
}

static void *first_thread()
{
  int i = 0;

  fprintf(stderr, "first_thread id: %d\n", thr_self());
  while (first_active){
    fprintf(stderr, "first_thread: %d\n", i++);
    sleep(2);
  }
  fprintf(stderr, "first_thread exit\n");
}

static void *second_thread()
{
  int i = 0;

  fprintf(stderr, "second_thread id: %d\n", thr_self());

  while (second_active){
    fprintf(stderr, "second_thread: %d\n", i++);
    sleep(3);
  }
  fprintf(stderr, "second_thread exit\n");
}

void ExitHandler(int sig)
{
  thread_t id;

  id = thr_self();

  fprintf(stderr, "ExitHandler thread id: %d\n", id);
  thr_exit(0);

}
```

Interprocess Synchronization

This example uses some of the synchronization variables available in the threads library to synchronize access to a resource shared between two processes. The synchronization variables used in the threads library are an advantage over standard IPC synchronization mechanisms because of their speed. The synchronization variables in the threads libraries have been tuned to be very lightweight and very fast. This speed can be an advantage when your application is spending time synchronizing between processes.

This example shows how semaphores from the threads library can be used between processes. Note that this program does not use threads; it is just using the lightweight semaphores available from the threads library.

When using synchronization variables between processes, it is important to make sure that only one process initializes the variable. If both processes try to initialize the synchronization variable, then one of the processes will overwrite the state of the variable set by the other process.

The source to `ipc.c`

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <synch.h>
#include <sys/types.h>
#include <unistd.h>

/* a structure that will be used between processes */
typedef struct {
        sema_t mysema;
        int num;
} buf_t;

main()
{
int     i, j, fd;
buf_t   *buf;

/* open a file to use in a memory mapping */
fd = open("/dev/zero", O_RDWR);

/* create a shared memory map with the open file for the data
   structure that will be shared between processes */
buf=(buf_t *)mmap(NULL, sizeof(buf_t), PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);

/* initialize the semaphore -- note the USYNC_PROCESS flag; this makes
   the semaphore visible from a process level */
sema_init(&buf->mysema, 0, USYNC_PROCESS, 0);

/* fork a new process */
if (fork() == 0) {
        /* The child will run this section of code */
        for (j=0;j<5;j++)
                {
                /* have the child "wait" for the semaphore */

                printf("Child PID(%d): waiting...\n", getpid());
                sema_wait(&buf->mysema);

                /* the child decremented the semaphore */

                printf("Child PID(%d): decrement semaphore.\n", getpid());
                }
        /* exit the child process */
        printf("Child PID(%d): exiting...\n", getpid());
        exit(0);
        }

/* The parent will run this section of code */
/* give the child a chance to start running */

sleep(2);

for (i=0;i<5;i++)
        {
        /* increment (post) the semaphore */

        printf("Parent PID(%d): posting semaphore.\n", getpid());
        sema_post(&buf->mysema);

        /* wait a second */
        sleep(1);
        }

/* exit the parent process */
printf("Parent PID(%d): exiting...\n", getpid());

return(0);
}
```

The Producer / Consumer Problem

This example will show how condition variables can be used to control access of reads and writes to a buffer. This example can also be thought as a producer/consumer problem, where the producer adds items to the buffer and the consumer removes items from the buffer.

Two condition variables control access to the buffer. One condition variable is used to tell if the buffer is full, and the other is used to tell if the buffer is empty. When the producer wants to add an item to the buffer, it checks to see if the buffer is full; if it is full the producer blocks on the `cond_wait()` call, waiting for an item to be removed from the buffer. When the consumer removes an item from the buffer, the buffer is no longer full, so the producer is awakened from the `cond_wait()` call. The producer is then allowed to add another item to the buffer.

The consumer works, in many ways, the same as the producer. The consumer uses the other condition variable to determine if the buffer is empty. When the consumer wants to remove an item from the buffer, it checks to see if it is empty. If the buffer is empty, the consumer then blocks on the `cond_wait()` call, waiting for an item to be added to the buffer. When the producer adds an item to the buffer, the consumer's condition is satisfied, so it can then remove an item from the buffer.

The example copies a file by reading data into a shared buffer (producer) and then writing data out to the new file (consumer). The Buf data structure is used to hold both the buffered data and the condition variables that control the flow of the data.

The main thread opens both files, initializes the `Buf` data structure, creates the consumer thread, and then assumes the role of the producer. The producer reads data from the input file, then places the data into an open buffer position. If no buffer positions are available, then the producer waits via the `cond_wait()` call. After the producer has read all the data from the input file, it closes the file and waits for (joins) the consumer thread.

The consumer thread reads from a shared buffer and then writes the data to the output file. If no buffers positions are available, then the consumer waits for the producer to fill a buffer position. After the consumer has read all the data, it closes the output file and exits.

If the input file and the output file were residing on different physical disks, then this example could execute the reads and writes in parallel. This parallelism would significantly increase the throughput of the example through the use of threads.

The source to `prod_cons.c`:

```
#define _REEENTRANT
#include <stdio.h>
#include <thread.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/uio.h>

#define BUFSIZE 512
#define BUFCNT  4

/* this is the data structure that is used between the producer
   and consumer threads */

struct {
        char buffer[BUFCNT][BUFSIZE];
        int byteinbuf[BUFCNT];
        mutex_t buflock;
        mutex_t donelock;
        cond_t adddata;
        cond_t remdata;
        int nextadd, nextrem, occ, done;
} Buf;

/* function prototype */
void *consumer(void *);

main(int argc, char **argv)
{
int ifd, ofd;
thread_t cons_thr;

/* check the command line arguments */
if (argc != 3)
        printf("Usage: %s <infile> <outfile>\n", argv[0]), exit(0);

/* open the input file for the producer to use */
if ((ifd = open(argv[1], O_RDONLY)) == -1)
        {
        fprintf(stderr, "Can't open file %s\n", argv[1]);
        exit(1);
        }

/* open the output file for the consumer to use */
if ((ofd = open(argv[2], O_WRONLY|O_CREAT, 0666)) == -1)
        {
        fprintf(stderr, "Can't open file %s\n", argv[2]);
        exit(1);
        }

/* zero the counters */
Buf.nextadd = Buf.nextrem = Buf.occ = Buf.done = 0;

/* set the thread concurrency to 2 so the producer and consumer can
   run concurrently */
```

```
thr_setconcurrency(2);

/* create the consumer thread */
thr_create(NULL, 0, consumer, (void *)ofd, NULL, &cons_thr);

/* the producer ! */
while (1) {

        /* lock the mutex */
        mutex_lock(&Buf.buflock);

        /* check to see if any buffers are empty */
        /* If not then wait for that condition to become true */

        while (Buf.occ == BUFCNT)
                cond_wait(&Buf.remdata, &Buf.buflock);

        /* read from the file and put data into a buffer */
        Buf.byteinbuf[Buf.nextadd] = read(ifd,Buf.buffer[Buf.nextadd],BUFSIZE);

        /* check to see if done reading */
        if (Buf.byteinbuf[Buf.nextadd] == 0) {

                /* lock the done lock */
                mutex_lock(&Buf.donelock);

                /* set the done flag and release the mutex lock */
                Buf.done = 1;

                mutex_unlock(&Buf.donelock);

                /* signal the consumer to start consuming */
                cond_signal(&Buf.adddata);

                /* release the buffer mutex */
                mutex_unlock(&Buf.buflock);

                /* leave the while looop */
                break;
                }

        /* set the next buffer to fill */
        Buf.nextadd = ++Buf.nextadd % BUFCNT;

        /* increment the number of buffers that are filled */
        Buf.occ++;

        /* signal the consumer to start consuming */
        cond_signal(&Buf.adddata);

        /* release the mutex */
        mutex_unlock(&Buf.buflock);
        }

close(ifd);

/* wait for the consumer to finish */
thr_join(cons_thr, 0, NULL);

/* exit the program */
return(0);
}


/* The consumer thread */
void *consumer(void *arg)
{
int fd = (int) arg;

/* check to see if any buffers are filled or if the done flag is set */
while (1) {

        /* lock the mutex */
        mutex_lock(&Buf.buflock);

        if (!Buf.occ && Buf.done) {
           mutex_unlock(&Buf.buflock);
           break;
           }

        /* check to see if any buffers are filled */
        /* if not then wait for the condition to become true */

        while (Buf.occ == 0 && !Buf.done)
                cond_wait(&Buf.adddata, &Buf.buflock);

        /* write the data from the buffer to the file */
        write(fd, Buf.buffer[Buf.nextrem], Buf.byteinbuf[Buf.nextrem]);

        /* set the next buffer to write from */
        Buf.nextrem = ++Buf.nextrem % BUFCNT;

        /* decrement the number of buffers that are full */
        Buf.occ--;
```

```
        /* signal the producer that a buffer is empty */
        cond_signal(&Buf.remdata);

        /* release the mutex */
        mutex_unlock(&Buf.buflock);
        }

/* exit the thread */
thr_exit((void *)0);
}
```

A Socket Server

The socket server example uses threads to implement a "standard" socket port server. The example shows how easy it is to use `thr_create()` calls in the place of `fork()` calls in existing programs.

A standard socket server should listen on a socket port and, when a message arrives, fork a process to service the request. Since a `fork()` system call would be used in a nonthreaded program, any communication between the parent and child would have to be done through some sort of interprocess communication.

We can replace the `fork()` call with a `thr_create()` call. Doing so offers a few advantages: `thr_create()` can create a thread much faster then a `fork()` could create a new process, and any communication between the ***server*** and the new thread can be done with common variables. This technique makes the implementation of the socket server much easier to understand and should also make it respond much faster to incoming requests.

The server program first sets up all the needed socket information. This is the basic setup for most socket servers. The server then enters an endless loop, waiting to service a socket port. When a message is sent to the socket port, the server wakes up and creates a new thread to handle the request. Notice that the server creates the new thread as a detached thread and also passes the socket descriptor as an argument to the new thread.

The newly created thread can then read or write, in any fashion it wants, to the socket descriptor that was passed to it. At this point the server could be creating a new thread or waiting for the next message to arrive. The key is that the server thread does not care what happens to the new thread after it creates it.

In our example, the created thread reads from the socket descriptor and then increments a global variable. This global variable keeps track of the number of requests that were made to the server. Notice that a mutex lock is used to protect access to the shared global variable. The lock is needed because many threads might try to increment the same variable at the same time. The mutex lock provides serial access to the shared variable. See how easy it is to share information among the new threads! If each of the threads were a process, then a significant effort would have to be made to share this information among the processes.

The client piece of the example sends a given number of messages to the server. This client code could also be run from different machines by multiple users, thus increasing the need for concurrency in the server process.

The source code to `soc_server.c`:

```
#define _REENTRANT
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <sys/uio.h>
#include <unistd.h>
#include <thread.h>

/* the TCP port that is used for this example */
#define TCP_PORT   6500

/* function prototypes and global variables */
void *do_chld(void *);
mutex_t lock;
int     service_count;

main()
{
        int     sockfd, newsockfd, clilen;
        struct sockaddr_in cli_addr, serv_addr;
        thread_t chld_thr;

        if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
                fprintf(stderr,"server: can't open stream socket\n"), exit(0);

        memset((char *) &serv_addr, 0, sizeof(serv_addr));
        serv_addr.sin_family = AF_INET;
        serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
        serv_addr.sin_port = htons(TCP_PORT);

        if(bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) <
0)
                fprintf(stderr,"server: can't bind local address\n"), exit(0);

        /* set the level of thread concurrency we desire */
        thr_setconcurrency(5);

        listen(sockfd, 5);
```

```
        for(;;){
                clilen = sizeof(cli_addr);
                newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr,
&clilen);

                if(newsockfd < 0)
                        fprintf(stderr,"server: accept error\n"), exit(0);

                /* create a new thread to process the incomming request */
                thr_create(NULL, 0, do_chld, (void *) newsockfd, THR_DETACHED,
&chld_thr);

                /* the server is now free to accept another socket request */
                }
        return(0);
}

/*
        This is the routine that is executed from a new thread
*/

void *do_chld(void *arg)
{
int     mysocfd = (int) arg;
char    data[100];
int     i;

        printf("Child thread [%d]: Socket number = %d\n", thr_self(), mysocfd);

        /* read from the given socket */
        read(mysocfd, data, 40);

        printf("Child thread [%d]: My data = %s\n", thr_self(), data);

        /* simulate some processing */
        for (i=0;i<1000000*thr_self();i++);

        printf("Child [%d]: Done Processing...\n", thr_self());

        /* use a mutex to update the global service counter */
        mutex_lock(&lock);

        service_count++;
        mutex_unlock(&lock);

        printf("Child thread [%d]: The total sockets served = %d\n", thr_self(), service_count);

        /* close the socket and exit this thread */
        close(mysocfd);
        thr_exit((void *)0);
}
```

Using Many Threads

This example that shows how easy it is to create many threads of execution in Solaris. Because of the lightweight nature of threads, it is possible to create literally thousands of threads. Most applications may not need a very large number of threads, but this example shows just how lightweight the threads can be.

We have said before that anything you can do with threads, you can do without them. This may be a case where it would be very hard to do without threads. If you have some spare time (and lots of memory), try implementing this program by using processes, instead of threads. If you try this, you will see why threads can have an advantage over processes.

This program takes as an argument the number of threads to create. Notice that all the threads are created with a user-defined stack size, which limits the amount of memory that the threads will need for execution. The stack size for a given thread can be hard to calculate, so some testing usually needs to be done to see if the chosen stack size will work. You may want to change the stack size in this program and see how much you can lower it before things stop working. The Solaris threads library provides the `thr_min_stack()` call, which returns the minimum allowed stack size. Take care when adjusting the size of a threads stack. A stack overflow can happen quite easily to a thread with a small stack.

After each thread is created, it blocks, waiting on a mutex variable. This mutex variable was locked before any of the threads were created, which prevents the threads from proceeding in their execution. When all of the threads have been created and the user presses Return, the mutex variable is unlocked, allowing all the threads to proceed.

After the main thread has created all the threads, it waits for user input and then tries to join all the threads. Notice that the `thr_join()` call does not care what thread it joins; it is just counting the number of joins it makes.

This example is rather trivial and does not serve any real purpose except to show that it is possible to create a lot of threads in one process. However, there are situations when many threads are needed in an application. An example might be a network port server, where a thread is created each time an incoming or outgoing request is made.

The source to `many_thr.c`:

```
#define _REENTRANT
#include <stdio.h>
#include <stdlib.h>
#include <thread.h>
```

```
/* function prototypes and global varaibles */
void *thr_sub(void *);
mutex_t lock;

main(int argc, char **argv)
{
int i, thr_count = 100;
char buf;

/* check to see if user passed an argument
   -- if so, set the number of threads to the value
      passed to the program */

if (argc == 2) thr_count = atoi(argv[1]);

printf("Creating %d threads...\n", thr_count);

/* lock the mutex variable -- this mutex is being used to
   keep all the other threads created from proceeding   */

mutex_lock(&lock);

/* create all the threads -- Note that a specific stack size is
   given.  Since the created threads will not use all of the
   default stack size, we can save memory by reducing the threads'
   stack size */

for (i=0;i<thr_count;i++) {
        thr_create(NULL,2048,thr_sub,0,0,NULL);
        }

printf("%d threads have been created and are running!\n", i);
printf("Press <return> to join all the threads...\n", i);

/* wait till user presses return, then join all the threads */
gets(&buf);

printf("Joining %d threads...\n", thr_count);

/* now unlock the mutex variable, to let all the threads proceed */
mutex_unlock(&lock);

/* join the threads */
for (i=0;i<thr_count;i++)
        thr_join(0,0,0);

printf("All %d threads have been joined, exiting...\n", thr_count);
return(0);
}

/* The routine that is executed by the created threads */

void *thr_sub(void *arg)
{
/* try to lock the mutex variable -- since the main thread has
   locked the mutex before the threads were created, this thread
   will block until the main thread unlock the mutex */

mutex_lock(&lock);

printf("Thread %d is exiting...\n", thr_self());

/* unlock the mutex to allow another thread to proceed */
mutex_unlock(&lock);

/* exit the thread */
return((void *)0);
}
```

Real-time Thread Example

This example uses the Solaris real-time extensions to make a single bound thread within a process run in the real-time scheduling class. Using a thread in the real-time class is more desirable than running a whole process in the real-time class, because of the many problems that can arise with a process in a real-time state. For example, it would not be desirable for a process to perform any I/O or large memory operations while in realtime, because a real-time process has priority over system-related processes; if a real-time process requests a page fault, it can starve, waiting for the system to fault in a new page. We can limit this exposure by using threads to execute only the instructions that need to run in realtime.

Since this book does not cover the concerns that arise with real-time programming, we have included this code only as an example of how to promote a thread into the real-time class. You must be very careful when you use real-time threads in your applications. For more information on real-time programming, see the Solaris documentation.

This example should be safe from the pitfalls of real-time programs because of its simplicity. However, changing this code in any way could have adverse affects on your system.

The example creates a new thread from the main thread. This new thread is then promoted to the real-time class by looking up the real-time class ID and then setting a real-time priority for the thread. After the thread is running in realtime, it simulates some processing. Since a thread in the real-time class can have an infinite time quantum, the process is allowed to stay on a CPU as long as

it likes. The time quantum is the amount of time a thread is allowed to stay running on a CPU. For the timesharing class, the time quantum (time-slice) is 1/100th of a second by default.

In this example, we set the time quantum for the real-time thread to infinity. That is, it can stay running as long as it likes; it will not be preempted or scheduled off the CPU. If you run this example on a UP machine, it will have the effect of stopping your system for a few seconds while the thread simulates its processing. The system does not actually stop, it is just working in the real-time thread. When the real-time thread finishes its processing, it exits and the system returns to normal.

Using real-time threads can be quite useful when you need an extremely high priority and response time but can also cause big problems if it not used properly. Also note that this example must be run as root or have root execute permissions.

The source to `rt_thr.c`:

```
#define _REENTRANT
#include <stdio.h>
#include <thread.h>
#include <string.h>
#include <sys/priocntl.h>
#include <sys/rtpriocntl.h>

/* thread prototype */
void *rt_thread(void *);

main()
{

/* create the thread that will run in realtime */
thr_create(NULL, 0, rt_thread, 0, THR_DETACHED, 0);

/* loop here forever, this thread is the TS scheduling class */
while (1) {
        printf("MAIN: In time share class... running\n");
        sleep(1);
        }

return(0);
}

/*
        This is the routine that is called by the created thread
*/

void *rt_thread(void *arg)
{
pcinfo_t pcinfo;
pcparms_t pcparms;
int i;

/* let the main thread run for a bit */
sleep(4);

/* get the class ID for the real-time class */
strcpy(pcinfo.pc_clname, "RT");

if (priocntl(0, 0, PC_GETCID, (caddr_t)&pcinfo) == -1)
        fprintf(stderr, "getting RT class id\n"), exit(1);

/* set up the real-time parameters */
pcparms.pc_cid = pcinfo.pc_cid;
((rtparms_t *)pcparms.pc_clparms)->rt_pri = 10;
((rtparms_t *)pcparms.pc_clparms)->rt_tqnsecs = 0;

/* set an infinite time quantum */
((rtparms_t *)pcparms.pc_clparms)->rt_tqsecs = RT_TQINF;

/* move this thread to the real-time scheduling class */
if (priocntl(P_LWPID, P_MYID, PC_SETPARMS, (caddr_t)&pcparms) == -1)
        fprintf(stderr, "Setting RT mode\n"), exit(1);

/* simulate some processing */
for (i=0;i<100000000;i++);

printf("RT_THREAD: NOW EXITING...\n");
thr_exit((void *)0);
}
```

POSIX Cancellation

This example uses the POSIX thread cancellation capability to kill a thread that is no longer needed. Random termination of a thread can cause problems in threaded applications, because a thread may be holding a critical lock when it is terminated. Since the lock was help before the thread was terminated, another thread may deadlock, waiting for that same lock. The thread cancellation capability enables you to control when a thread can be terminated. The example also demonstrates the capabilities of the POSIX thread library in implementing a program that performs a multithreaded search.

This example simulates a multithreaded search for a given number by taking random guesses at a target number. The intent here is to simulate the same type of search that a database might execute. For example, a database might create threads to start searching for a data item; after some amount of time, one or more threads might return with the target data item.

If a thread guesses the number correctly, there is no need for the other threads to continue their search. This is where thread cancellation can help. The thread that finds the number first should cancel the other threads that are still searching for the item and then return the results of the search.

The threads involved in the search can call a cleanup function that can clean up the threads resources before it exits. In this case, the cleanup function prints the progress of the thread when it was cancelled.

The source to `posix_cancel.c`:

```
 #define _REENTRANT
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <pthread.h>

/* defines the number of searching threads */
#define NUM_THREADS 25

/* function prototypes */
void *search(void *);
void print_it(void *);

/* global variables */
pthread_t  threads[NUM_THREADS];
pthread_mutex_t lock;
int tries;

main()
{
int i;
int pid;

/* create a number to search for */
pid = getpid();

/* initialize the mutex lock */
pthread_mutex_init(&lock, NULL);
printf("Searching for the number = %d...\n", pid);

/* create the searching threads */
for (i=0;i<NUM_THREADS;i++)
        pthread_create(&threads[i], NULL, search, (void *)pid);

/* wait for (join) all the searching threads */
for (i=0;i<NUM_THREADS;i++)
        pthread_join(threads[i], NULL);

printf("It took %d tries to find the number.\n", tries);

/* exit this thread */
pthread_exit((void *)0);
}

/*
        This is the cleanup function that is called when
        the threads are cancelled
*/

void print_it(void *arg)
{
int *try = (int *) arg;
pthread_t tid;

/* get the calling thread's ID */
tid = pthread_self();

/* print where the thread was in its search when it was cancelled */
printf("Thread %d was canceled on its %d try.\n", tid, *try);
}

/*
        This is the search routine that is executed in each thread
*/

void *search(void *arg)
{
int num = (int) arg;
int i=0, j;
pthread_t tid;

/* get the calling thread ID */
tid = pthread_self();

/* use the thread ID to set the seed for the random number generator */
srand(tid);

/* set the cancellation parameters --
   - Enable thread cancellation
   - Defer the action of the cancellation
*/

pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
```

```
pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);

/* push the cleanup routine (print_it) onto the thread
   cleanup stack.  This routine will be called when the
   thread is cancelled.  Also note that the pthread_cleanup_push
   call must have a matching pthread_cleanup_pop call.  The
   push and pop calls MUST be at the same lexical level
   within the code */

/* pass address of `i' since the current value of `i' is not
   the one we want to use in the cleanup function */

pthread_cleanup_push(print_it, (void *)&i);

/* loop forever */
while (1) {
       i++;

       /* does the random number match the target number? */
       if (num == rand()) {

              /* try to lock the mutex lock --
                  if locked, check to see if the thread has been cancelled
                 if not locked then continue */

              while (pthread_mutex_trylock(&lock) == EBUSY)
                            pthread_testcancel();

              /* set the global variable for the number of tries */
              tries = i;

              printf("thread %d found the number!\n", tid);

              /* cancel all the other threads */
              for (j=0;j<NUM_THREADS;j++)
                     if (threads[j] != tid) pthread_cancel(threads[j]);

              /* break out of the while loop */
              break;
              }

       /* every 100 tries check to see if the thread has been cancelled
          if the thread has not been cancelled then yield the thread's
          LWP to another thread that may be able to run */

       if (i%100 == 0) {
              pthread_testcancel();
              sched_yield();
              }
       }

/* The only way we can get here is when the thread breaks out
   of the while loop.  In this case the thread that makes it here
   has found the number we are looking for and does not need to run
   the thread cleanup function.  This is why the pthread_cleanup_pop
   function is called with a 0 argument; this will pop the cleanup
   function off the stack without executing it */

pthread_cleanup_pop(0);
return((void *)0);
}
```

Software Race Condition

This example shows a trivial software race condition. A software race condition occurs when the execution of a program is affected by the order and timing of a threads execution. Most software race conditions can be alleviated by using synchronization variables to control the threads' timing and access of shared resources. If a program depends on order of execution, then threading that program may not be a good solution, because the order in which threads execute is nondeterministic.

In the example, `thr_continue()` and `thr_suspend()` calls continue and suspend a given thread, respectively. Although both of these calls are valid, use caution when implementing them. It is very hard to determine where a thread is in its execution. Because of this, you may not be able to tell where the thread will suspend when the call to `thr_suspend()` is made. This behavior can cause problems in threaded code if not used properly.

The following example uses `thr_continue()` and `thr_suspend()` to try to control when a thread starts and stops. The example looks trivial, but, as you will see, can cause a big problem.

Do you see the problem? If you guessed that the program would eventually suspend itself, you were correct! The example attempts to flip-flop between the main thread and a subroutine thread. Each thread continues the other thread and then suspends itself.

Thread A continues thread B and then suspends thread A; now the continued thread B can continue thread A and then suspend itself. This should continue back and forth all day long, right? Wrong! We can't guarantee that each thread will continue the other thread and then suspend itself in one atomic action, so a software race condition could be created. Calling `thr_continue()` on a running thread and calling `thr_suspend()` on a suspended thread has no effect, so we don't know if a thread is already running or suspended.

If thread A continues thread B and if between the time thread A suspends itself, thread B continues thread A, then both of the threads will call `thr_suspend()`. This is the race condition in this program that will cause the whole process to become suspended.

It is very hard to use these calls, because you never really know the state of a thread. If you don't know exactly where a thread is in its execution, then you don't know what locks it holds and where it will stop when you suspend it.

The source to `sw_race.c`

`Tgrep`: Threadeds version of UNIX `grep`

`Tgrep` is a multi-threaded version of `grep`. `Tgrep` supports all but the -w (word search) options of the normal `grep` command, and a few options that are only avaliable to `Tgrep`. The real change from `grep`, is that `Tgrep` will recurse down through sub-directories and search all files for the target string. `Tgrep` searches files like the following command:

```
find <start path> -name "<file/directory pattern>" -exec \ (Line wrapped)
        grep <options> <target> /dev/null {} \;
```

An example of this would be (run from this `Tgrep` directory)

```
% find . -exec grep thr_create /dev/null {} \;
./Solaris/main.c:  if (thr_create(NULL,0,SigThread,NULL,THR_DAEMON,NULL)) {
./Solaris/main.c:          err = thr_create(NULL,0,cascade,(void *)work,
./Solaris/main.c:           err = thr_create(NULL,0,search_thr,(void *)work,
%
Running the same command with timex:
real        4.26
user        0.64
sys         2.81
```

The same search run with `Tgrep` would be

```
% {\tt Tgrep} thr_create
./Solaris/main.c:  if (thr_create(NULL,0,SigThread,NULL,THR_DAEMON,NULL)) {
./Solaris/main.c:          err = thr_create(NULL,0,cascade,(void *)work,
./Solaris/main.c:           err = thr_create(NULL,0,search_thr,(void *)work,
%
Running the same command with timex:
real        0.79
user        0.62
sys         1.50
```

`Tgrep` gets the results almost four times faster. The numbers above where gathered on a SS20 running 5.5 (build 18) with 4 50MHz CPUs.

You can also filter the files that you want `Tgrep` to search like you can with find. The next two commands do the same thing, just `Tgrep` gets it done faster.

```
find . -name "*.c" -exec grep thr_create /dev/null {} \;
and
{\tt Tgrep} -p '.*\.c$' thr_create
```

The -p option will allow `Tgrep` to search only files that match the "regular expression" file pattern string. This option does NOT use shell expression, so to stop `Tgrep` from seeing a file named foobar.cyou must add the "$" meta character to the pattern and escape the real ``." character.

Some of the other `Tgrep` only options are -r, -C, -P, -e, -B, -S and -Z. The -r option stops `Tgrep` from searching any sub-directories, in other words, search only the local directory, but -l was taken. The -C option will search for and print "continued" lines like you find in Makefile. Note the differences in the results of grep and `Tgrep` run in the current directory.

The `Tgrep` output prints the continued lines that ended with the "character. In the case of grep I would not have seen the three values assigned to SUBDIRS, but `Tgrep` shows them to me (Common, Solaris, Posix).

The -P option I use when I am sending the output of a long search to a file and want to see the "progress" of the search. The -P option will print a "." (dot) on stderr for every file (or groups of files depending on the value of the -P argument) `Tgrep` searches.

The -e option will change the way `Tgrep` uses the target string. `Tgrep` uses two different patter matching systems. The first (with out the -e option) is a literal string match call Boyer-Moore. If the -e option is used, then a MT-Safe PD version of regular expression is used to search for the target string as a regexp with meta characters in it. The regular expression method is slower, but `Tgrep` needed the functionality. The -Z option will print help on the meta characters `Tgrep` uses.

The -B option tells `Tgrep` to use the value of the environment variable called TGLIMIT to limit the number of threads it will use during a search. This option has no affect if TGLIMIT is not set. `Tgrep` can "eat" a system alive, so the -B option was a way to run `Tgrep` on a system with out having other users scream at you.

The last new option is -S. If you want to see how things went while `Tgrep` was searching, you can use this option to print statistic about the number of files, lines, bytes, matches, threads created, etc.

Here is an example of the -S options output. (again run in the current directory)

```
% {\tt Tgrep} -S zimzap

----------------- {\tt Tgrep} Stats. --------------------
Number of directories searched:           7
Number of files searched:                 37
```

```
Number of lines searched:                9504
Number of matching lines to target:      0
Number of cascade threads created:       7
Number of search threads created:        20
Number of search threads from pool:      17
Search thread pool hit rate:             45.95%
Search pool overall size:                20
Search pool size limit:                  58
Number of search threads destroyed:      0
Max # of threads running concurrenly:    20
Total run time, in seconds.              1
Work stopped due to no FD's:  (058)      0 Times, 0.00%
Work stopped due to no work on Q:        19 Times, 43.18%
Work stopped due to TGLIMITS: (Unlimited) 0 Times, 0.00%
-------------------------------------------------
%
```

For more information on the usage and options, see the man page `Tgrep`

The `Tgrep.c` source code is:

```
/* Copyright (c) 1993, 1994  Ron Winacott                              */
/* This program may be used, copied, modified, and redistributed freely */
/* for ANY purpose, so long as this notice remains intact.             */

#define _REENTRANT

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <errno.h>
#include <signal.h>
#include <ctype.h>
#include <sys/types.h>
#include <time.h>
#include <sys/stat.h>
#ifdef __sparc
#include <note.h> /* warlock/locklint */
#else
#define NOTE(s)
#endif
#include <dirent.h>
#include <fcntl.h>
#include <sys/uio.h>
#include <thread.h>
#include <synch.h>

#include "version.h"
#include "pmatch.h"
#include "debug.h"

#define PATH_MAX              1024 /* max # of characters in a path name */
#define HOLD_FDS                6  /* stdin,out,err and a buffer */
#define UNLIMITED               99999 /* The default tglimit */
#define MAXREGEXP               10  /* max number of -e options */

#define FB_BLOCK                0x00001
#define FC_COUNT                0x00002
#define FH_HOLDNAME             0x00004
#define FI_IGNCASE              0x00008
#define FL_NAMEONLY             0x00010
#define FN_NUMBER               0x00020
#define FS_NOERROR              0x00040
#define FV_REVERSE              0x00080
#define FW_WORD                 0x00100
#define FR_RECUR                0x00200
#define FU_UNSORT               0x00400
#define FX_STDIN                0x00800
#define TG_BATCH                0x01000
#define TG_FILEPAT              0x02000
#define FE_REGEXP               0x04000
#define FS_STATS                0x08000
#define FC_LINE                 0x10000
#define TG_PROGRESS             0x20000

#define FILET                   1
#define DIRT                    2
#define ALPHASIZ              128

/*
 * New data types
 */

typedef struct work_st {
    char                *path;
    int                 tp;
    struct work_st      *next;
} work_t;

typedef struct out_st {
    char                *line;
    int                 line_count;
```

```
    long                byte_count;
    struct out_st       *next;
} out_t;

typedef struct bm_pattern {     /* Boyer - Moore pattern                */
        short            p_m;           /* length of pattern string     */
        short            p_r[ALPHASIZ]; /* "r" vector                   */
        short           *p_R;           /* "R" vector                   */
        char            *p_pat;         /* pattern string               */
} BM_PATTERN;


/*
 * Prototypes
 */

/* bmpmatch.c */
extern BM_PATTERN *bm_makepat(char *);
extern char *bm_pmatch(BM_PATTERN *, register char *);
extern void bm_freepat(BM_PATTERN *);
/* pmatch.c */
extern char *pmatch(register PATTERN *, register char *, int *);
extern PATTERN *makepat(char *string, char *);
extern void freepat(register PATTERN *);
extern void printpat(PATTERN *);

#include "proto.h"  /* function prototypes of main.c */

void *SigThread(void *arg);
void sig_print_stats(void);

/*
 * Global data
 */

BM_PATTERN      *bm_pat;  /* the global target read only after main */
NOTE(READ_ONLY_DATA(bm_pat))

PATTERN         *pm_pat[MAXREGEXP];  /* global targets read only for pmatch */
NOTE(READ_ONLY_DATA(pm_pat))

mutex_t global_count_lk;
int     global_count = 0;
NOTE(MUTEX_PROTECTS_DATA(global_count_lk, global_count))
NOTE(DATA_READABLE_WITHOUT_LOCK(global_count))  /* see prnt_stats() */

work_t  *work_q = NULL;
cond_t  work_q_cv;
mutex_t work_q_lk;
int     all_done = 0;
int     work_cnt = 0;
int     current_open_files = 0;
int     tglimit = UNLIMITED;    /* if -B limit the number of threads */
NOTE(MUTEX_PROTECTS_DATA(work_q_lk, work_q all_done work_cnt \
                        current_open_files tglimit))

work_t  *search_q = NULL;
mutex_t search_q_lk;
cond_t  search_q_cv;
int     search_pool_cnt = 0;    /* the count in the pool now */
int     search_thr_limit = 0;   /* the max in the pool */
NOTE(MUTEX_PROTECTS_DATA(search_q_lk, search_q search_pool_cnt))
NOTE(DATA_READABLE_WITHOUT_LOCK(search_pool_cnt)) /* see prnt_stats() */
NOTE(READ_ONLY_DATA(search_thr_limit))

work_t  *cascade_q = NULL;
mutex_t cascade_q_lk;
cond_t  cascade_q_cv;
int     cascade_pool_cnt = 0;
int     cascade_thr_limit = 0;
NOTE(MUTEX_PROTECTS_DATA(cascade_q_lk, cascade_q cascade_pool_cnt))
NOTE(DATA_READABLE_WITHOUT_LOCK(cascade_pool_cnt))  /* see prnt_stats() */
NOTE(READ_ONLY_DATA(cascade_thr_limit))

int     running = 0;
mutex_t running_lk;
NOTE(MUTEX_PROTECTS_DATA(running_lk, running))

sigset_t set, oldset;
NOTE(READ_ONLY_DATA(set oldset))

mutex_t stat_lk;
time_t  st_start = 0;
int     st_dir_search = 0;
int     st_file_search = 0;
int     st_line_search = 0;
int     st_cascade = 0;
int     st_cascade_pool = 0;
int     st_cascade_destroy = 0;
int     st_search = 0;
int     st_pool = 0;
int     st_maxrun = 0;
int     st_worknull = 0;
int     st_workfds = 0;
int     st_worklimit = 0;
```

```
int     st_destroy = 0;
NOTE(MUTEX_PROTECTS_DATA(stat_lk, st_start st_dir_search st_file_search \
                         st_line_search st_cascade st_cascade_pool \
                         st_cascade_destroy st_search st_pool st_maxrun \
                         st_worknull st_workfds st_worklimit st_destroy))

int     progress_offset = 1;
NOTE(READ_ONLY_DATA(progress_offset))

mutex_t output_print_lk;
/* output_print_lk used to print multi-line output only */
int     progress = 0;
NOTE(MUTEX_PROTECTS_DATA(output_print_lk, progress))

unsigned int    flags = 0;
int     regexp_cnt = 0;
char    *string[MAXREGEXP];
int     debug = 0;
int     use_pmatch = 0;
char    file_pat[255];  /* file patten match */
PATTERN *pm_file_pat; /* compiled file target string (pmatch()) */
NOTE(READ_ONLY_DATA(flags regexp_cnt string debug use_pmatch \
                    file_pat pm_file_pat))


/*
 * Locking ording.
 */
NOTE(LOCK_ORDER(output_print_lk stat_lk))
/*
 * Main: This is where the fun starts
 */

int
main(int argc, char **argv)
{
    int         c,out_thr_flags;
    long        max_open_files = 0l, ncpus = 0l;
    extern int  optind;
    extern char *optarg;
    NOTE(READ_ONLY_DATA(optind optarg))
    int         prio = 0;
    struct stat sbuf;
    thread_t    tid,dtid;
    void        *status;
    char        *e = NULL, *d = NULL; /* for debug flags */
    int         debug_file = 0;
    int         err = 0, i = 0, pm_file_len = 0;
    work_t      *work;
    int         restart_cnt = 10;

    flags = FR_RECUR;  /* the default */

    thr_setprio(thr_self(),127);  /* set me up HIGH */
    while ((c = getopt(argc, argv, "d:e:bchilnsvwruf:p:BCSZzHP:")) != EOF) {
        switch (c) {
#ifdef DEBUG
        case 'd':
            debug = atoi(optarg);
            if (debug == 0)
                debug_usage();

            d = optarg;
            fprintf(stderr,"tgrep: Debug on at level(s) ");
            while (*d) {
                for (i=0; i<9; i++)
                    if (debug_set[i].level == *d) {
                        debug_levels |= debug_set[i].flag;
                        fprintf(stderr,"%c ",debug_set[i].level);
                        break;
                    }
                d++;
            }
            fprintf(stderr,"\n");
            break;
        case 'f':
            debug_file = atoi(optarg);
            break;
#endif      /* DEBUG */
        case 'B':
            flags |= TG_BATCH;
            if ((e = getenv("TGLIMIT"))) {
                tglimit = atoi(e);
            }
            else {
                if (!(flags & FS_NOERROR))  /* order dependent! */
                    fprintf(stderr,"env TGLIMIT not set, overriding -B\n");
                flags &= ~TG_BATCH;
            }
            break;
        case 'p':
            flags |= TG_FILEPAT;
            strcpy(file_pat,optarg);
            pm_file_pat = makepat(file_pat,NULL);
```

```
    break;
case 'P':
    flags |= TG_PROGRESS;
    progress_offset = atoi(optarg);
    break;
case 'S': flags |= FS_STATS;    break;
case 'b': flags |= FB_BLOCK;    break;
case 'c': flags |= FC_COUNT;    break;
case 'h': flags |= FH_HOLDNAME; break;
case 'i': flags |= FI_IGNCASE;  break;
case 'l': flags |= FL_NAMEONLY; break;
case 'n': flags |= FN_NUMBER;   break;
case 's': flags |= FS_NOERROR;  break;
case 'v': flags |= FV_REVERSE;  break;
case 'w': flags |= FW_WORD;     break;
case 'r': flags &= ~FR_RECUR;   break;
case 'C': flags |= FC_LINE;     break;
case 'e':
    if (regexp_cnt == MAXREGEXP) {
        fprintf(stderr,"Max number of regexp's (%d) exceeded!\n",
                MAXREGEXP);
        exit(1);
    }
    flags |= FE_REGEXP;
    if ((string[regexp_cnt] =(char *)malloc(strlen(optarg)+1))==NULL){
        fprintf(stderr,"tgrep: No space for search string(s)\n");
        exit(1);
    }
    memset(string[regexp_cnt],0,strlen(optarg)+1);
    strcpy(string[regexp_cnt],optarg);
            regexp_cnt++;
            break;
         case 'z':
         case 'Z': regexp_usage();
            break;
         case 'H':
         case '?':
         default : usage();
         }
    }

    if (!(flags & FE_REGEXP)) {
       if (argc - optind < 1) {
            fprintf(stderr,"tgrep: Must supply a search string(s) "
                    "and file list or directory\n");
            usage();
         }
         if ((string[0]=(char *)malloc(strlen(argv[optind])+1))==NULL){
            fprintf(stderr,"tgrep: No space for search string(s)\n");
            exit(1);
         }
         memset(string[0],0,strlen(argv[optind])+1);
         strcpy(string[0],argv[optind]);
         regexp_cnt=1;
         optind++;
    }

    if (flags & FI_IGNCASE)
         for (i=0; i<regexp_cnt; i++)
            uncase(string[i]);

#ifdef __lock_lint
    /*
    ** This is NOT somthing you really want to do. This
    ** function calls are here ONLY for warlock/locklint !!!
    */
    pm_pat[i] = makepat(string[i],NULL);
    bm_pat = bm_makepat(string[0]);
    bm_freepat(bm_pat);  /* stop it from becomming a root */
#else
    if (flags & FE_REGEXP) {
        for (i=0; i<regexp_cnt; i++)
            pm_pat[i] = makepat(string[i],NULL);
        use_pmatch = 1;
    }
    else {
        bm_pat = bm_makepat(string[0]); /* only one allowed */
    }
#endif

    flags |= FX_STDIN;

    max_open_files = sysconf(_SC_OPEN_MAX);
    ncpus = sysconf(_SC_NPROCESSORS_ONLN);
    if ((max_open_files - HOLD_FDS - debug_file) < 1) {
        fprintf(stderr,"tgrep: You MUST have at lest ONE fd "
                "that can be used, check limit (>10)\n");
        exit(1);
    }
    search_thr_limit = max_open_files - HOLD_FDS - debug_file;
    cascade_thr_limit = search_thr_limit / 2;
    /* the number of files that can by open */
    current_open_files = search_thr_limit;

    mutex_init(&stat_lk,USYNC_THREAD,"stat");
```

```
mutex_init(&global_count_lk,USYNC_THREAD,"global_cnt");
mutex_init(&output_print_lk,USYNC_THREAD,"output_print");
mutex_init(&work_q_lk,USYNC_THREAD,"work_q");
mutex_init(&running_lk,USYNC_THREAD,"running");
cond_init(&work_q_cv,USYNC_THREAD,"work_q");
mutex_init(&search_q_lk,USYNC_THREAD,"search_q");
cond_init(&search_q_cv,USYNC_THREAD,"search_q");
mutex_init(&cascade_q_lk,USYNC_THREAD,"cascade_q");
cond_init(&cascade_q_cv,USYNC_THREAD,"cascade_q");

if ((argc == optind) && ((flags & TG_FILEPAT) || (flags & FR_RECUR))) {
    add_work(".",DIRT);
    flags = (flags & ~FX_STDIN);
}
for ( ; optind < argc; optind++) {
    restart_cnt = 10;
    flags = (flags & ~FX_STDIN);
  STAT_AGAIN:
    if (stat(argv[optind], &sbuf)) {
        if (errno == EINTR) { /* try again !, restart */
            if (--restart_cnt)
                goto STAT_AGAIN;
        }
        if (!(flags & FS_NOERROR))
            fprintf(stderr,"tgrep: Can't stat file/dir %s, %s\n",
                    argv[optind], strerror(errno));
        continue;
    }
    switch (sbuf.st_mode & S_IFMT) {
    case S_IFREG :
            if (flags & TG_FILEPAT) {
                if (pmatch(pm_file_pat, argv[optind], &pm_file_len))
                   add_work(argv[optind],FILET);
            }
            else {
                add_work(argv[optind],FILET);
            }
            break;
        case S_IFDIR :
            if (flags & FR_RECUR) {
                add_work(argv[optind],DIRT);
            }
            else {
                if (!(flags & FS_NOERROR))
                    fprintf(stderr,"tgrep: Can't search directory %s, "
                            "-r option is on. Directory ignored.\n",
                            argv[optind]);
            }
            break;
        }
    }

    NOTE(COMPETING_THREADS_NOW)  /* we are goinf threaded */

    if (flags & FS_STATS) {
        mutex_lock(&stat_lk);
        st_start = time(NULL);
        mutex_unlock(&stat_lk);
#ifdef SIGNAL_HAND
        /*
        ** setup the signal thread so the first call to SIGINT will
        ** only print stats, the second will interupt.
        */
        sigfillset(&set);
        thr_sigsetmask(SIG_SETMASK, &set, &oldset);
        if (thr_create(NULL,0,SigThread,NULL,THR_DAEMON,NULL)) {
            thr_sigsetmask(SIG_SETMASK,&oldset,NULL);
            fprintf(stderr,"SIGINT for stats NOT setup\n");
        }
        thr_yield(); /* give the other thread time */
#endif /* SIGNAL_HAND */
    }

    thr_setconcurrency(3);

    if (flags & FX_STDIN) {
        fprintf(stderr,"tgrep: stdin option is not coded at this time\n");
        exit(0);                        /* XXX Need to fix this SOON */
        search_thr(NULL);  /* NULL is not understood in search_thr() */
        if (flags & FC_COUNT) {
            mutex_lock(&global_count_lk);
            printf("%d\n",global_count);
            mutex_unlock(&global_count_lk);
        }
        if (flags & FS_STATS) {
            mutex_lock(&stat_lk);
            prnt_stats();
            mutex_unlock(&stat_lk);
        }
        exit(0);
    }

    mutex_lock(&work_q_lk);
    if (!work_q) {
        if (!(flags & FS_NOERROR))
```

```
        fprintf(stderr,"tgrep: No files to search.\n");
    exit(0);
}
mutex_unlock(&work_q_lk);

DP(DLEVEL1,("Starting to loop through the work_q for work\n"));

/* OTHER THREADS ARE RUNNING */
while (1) {
    mutex_lock(&work_q_lk);
    while ((work_q == NULL || current_open_files == 0 || tglimit <= 0) &&
           all_done == 0) {
        if (flags & FS_STATS) {
            mutex_lock(&stat_lk);
            if (work_q == NULL)
                st_worknull++;
            if (current_open_files == 0)
                st_workfds++;
            if (tglimit <= 0)
                st_worklimit++;
            mutex_unlock(&stat_lk);
        }
        cond_wait(&work_q_cv,&work_q_lk);
    }
    if (all_done != 0) {
        mutex_unlock(&work_q_lk);
        DP(DLEVEL1,("All_done was set to TRUE\n"));
        goto OUT;
    }
    work = work_q;
    work_q = work->next;  /* maybe NULL */
    work->next = NULL;
    current_open_files--;
    mutex_unlock(&work_q_lk);

    tid = 0;
    switch (work->tp) {
    case DIRT:
        mutex_lock(&cascade_q_lk);
        if (cascade_pool_cnt) {
            if (flags & FS_STATS) {
                mutex_lock(&stat_lk);
                st_cascade_pool++;
                mutex_unlock(&stat_lk);
            }
            work->next = cascade_q;
            cascade_q = work;
            cond_signal(&cascade_q_cv);
            mutex_unlock(&cascade_q_lk);
            DP(DLEVEL2,("Sent work to cascade pool thread\n"));
        }
        else {
            mutex_unlock(&cascade_q_lk);
            err = thr_create(NULL,0,cascade,(void *)work,
                             THR_DETACHED|THR_DAEMON|THR_NEW_LWP
                             ,&tid);
            DP(DLEVEL2,("Sent work to new cascade thread\n"));
            thr_setprio(tid,64);  /* set cascade to middle */
            if (flags & FS_STATS) {
                mutex_lock(&stat_lk);
                st_cascade++;
                mutex_unlock(&stat_lk);
            }
        }
        break;
    case FILET:
        mutex_lock(&search_q_lk);
        if (search_pool_cnt) {
            if (flags & FS_STATS) {
                mutex_lock(&stat_lk);
                st_pool++;
                mutex_unlock(&stat_lk);
            }
            work->next = search_q;  /* could be null */
            search_q = work;
            cond_signal(&search_q_cv);
            mutex_unlock(&search_q_lk);
            DP(DLEVEL2,("Sent work to search pool thread\n"));
        }
        else {
            mutex_unlock(&search_q_lk);
            err = thr_create(NULL,0,search_thr,(void *)work,
                             THR_DETACHED|THR_DAEMON|THR_NEW_LWP
                             ,&tid);
            thr_setprio(tid,0);  /* set search to low */
            DP(DLEVEL2,("Sent work to new search thread\n"));
            if (flags & FS_STATS) {
                mutex_lock(&stat_lk);
                st_search++;
                mutex_unlock(&stat_lk);
            }
        }
        break;
    default:
        fprintf(stderr,"tgrep: Internal error, work_t->tp no valid\n");
```

```
            exit(1);
        }
        if (err) {  /* NEED TO FIX THIS CODE. Exiting is just wrong */
            fprintf(stderr,"Cound not create new thread!\n");
            exit(1);
        }
    }

 OUT:
    if (flags & TG_PROGRESS) {
        if (progress)
            fprintf(stderr,".\n");
        else
            fprintf(stderr,"\n");
    }
    /* we are done, print the stuff. All other threads ar parked */
    if (flags & FC_COUNT) {
        mutex_lock(&global_count_lk);
        printf("%d\n",global_count);
        mutex_unlock(&global_count_lk);
    }
    if (flags & FS_STATS)
        prnt_stats();
    return(0); /* should have a return from main */
}


/*
 * Add_Work: Called from the main thread, and cascade threads to add file
 * and directory names to the work Q.
 */
int
add_work(char *path,int tp)
{
    work_t      *wt,*ww,*wp;

    if ((wt = (work_t *)malloc(sizeof(work_t))) == NULL)
        goto ERROR;
    if ((wt->path = (char *)malloc(strlen(path)+1)) == NULL)
        goto ERROR;

    strcpy(wt->path,path);
    wt->tp = tp;
    wt->next = NULL;
    if (flags & FS_STATS) {
        mutex_lock(&stat_lk);
        if (wt->tp == DIRT)
            st_dir_search++;
        else
            st_file_search++;
        mutex_unlock(&stat_lk);
    }
    mutex_lock(&work_q_lk);
    work_cnt++;
    wt->next = work_q;
    work_q = wt;
    cond_signal(&work_q_cv);
    mutex_unlock(&work_q_lk);
    return(0);
 ERROR:
    if (!(flags & FS_NOERROR))
        fprintf(stderr,"tgrep: Could not add %s to work queue. Ignored\n",
                path);
    return(-1);
}

/*
 * Search thread: Started by the main thread when a file name is found
 * on the work Q to be serached. If all the needed resources are ready
 * a new search thread will be created.
 */
void *
search_thr(void *arg) /* work_t *arg */
{
    FILE        *fin;
    char        fin_buf[(BUFSIZ*4)];  /* 4 Kbytes */
    work_t      *wt,std;
    int         line_count;
    char        rline[128];
    char        cline[128];
    char        *line;
    register char *p,*pp;
    int            pm_len;
    int         len = 0;
    long        byte_count;
    long        next_line;
    int         show_line;  /* for the -v option */
    register int slen,plen,i;
    out_t       *out = NULL;    /* this threads output list */

    thr_setprio(thr_self(),0);  /* set search to low */
    thr_yield();
    wt = (work_t *)arg; /* first pass, wt is passed to use. */

    /* len = strlen(string);*/  /* only set on first pass */
```

```
while (1) {  /* reuse the search threads */
    /* init all back to zero */
    line_count = 0;
    byte_count = 0l;
    next_line = 0l;
    show_line = 0;

    mutex_lock(&running_lk);
    running++;
    mutex_unlock(&running_lk);
    mutex_lock(&work_q_lk);
    tglimit--;
    mutex_unlock(&work_q_lk);
    DP(DLEVEL5,("searching file (STDIO) %s\n",wt->path));

    if ((fin = fopen(wt->path,"r")) == NULL) {
        if (!(flags & FS_NOERROR)) {
            fprintf(stderr,"tgrep: %s. File \"%s\" not searched.\n",
                    strerror(errno),wt->path);
        }
        goto ERROR;
    }
    setvbuf(fin,fin_buf,_IOFBF,(BUFSIZ*4));  /* XXX */
    DP(DLEVEL5,("Search thread has opened file %s\n",wt->path));
    while ((fgets(rline,127,fin)) != NULL) {
        if (flags & FS_STATS) {
            mutex_lock(&stat_lk);
            st_line_search++;
            mutex_unlock(&stat_lk);
        }
        slen = strlen(rline);
        next_line += slen;
        line_count++;
        if (rline[slen-1] == '\n')
            rline[slen-1] = '\0';
        /*
        ** If the uncase flag is set, copy the read in line (rline)
        ** To the uncase line (cline) Set the line pointer to point at
        ** cline.
        ** If the case flag is NOT set, then point line at rline.
        ** line is what is compared, rline is waht is printed on a
        ** match.
        */
        if (flags & FI_IGNCASE) {
            strcpy(cline,rline);
            uncase(cline);
            line = cline;
        }
        else {
            line = rline;
        }
        show_line = 1;  /* assume no match, if -v set */
        /* The old code removed */
        if (use_pmatch) {
            for (i=0; i<regexp_cnt; i++) {
                if (pmatch(pm_pat[i], line, &pm_len)) {
                    if (!(flags & FV_REVERSE)) {
                        add_output_local(&out,wt,line_count,
                                         byte_count,rline);
                        continue_line(rline,fin,out,wt,
                                      &line_count,&byte_count);
                    }
                    else {
                        show_line = 0;
                    } /* end of if -v flag if / else block */
                    /*
                    ** if we get here on ANY of the regexp targets
                    ** jump out of the loop, we found a single
                    ** match so, do not keep looking!
                    ** If name only, do not keep searcthing the same
                   ** file, we found a single match, so close the file,
                   ** print the file name and move on to the next file.
                    */
                    if (flags & FL_NAMEONLY)
                        goto OUT_OF_LOOP;
                    else
                        goto OUT_AND_DONE;
                } /* end found a match if block */
            } /* end of the for pat[s] loop */
        }
        else {
            if (bm_pmatch( bm_pat, line)) {
                if (!(flags & FV_REVERSE)) {
                    add_output_local(&out,wt,line_count,byte_count,rline);
                    continue_line(rline,fin,out,wt,
                                  &line_count,&byte_count);
                }
                else {
                    show_line = 0;
                }
                if (flags & FL_NAMEONLY)
                    goto OUT_OF_LOOP;
            }
        }
```

```
      OUT_AND_DONE:
        if ((flags & FV_REVERSE) && show_line) {
            add_output_local(&out,wt,line_count,byte_count,rline);
            show_line = 0;
        }
        byte_count = next_line;
    }
  OUT_OF_LOOP:
    fclose(fin);
    /*
    ** The search part is done, but before we give back the FD,
    ** and park this thread in the search thread pool, print the
    ** local output we have gathered.
    */
    print_local_output(out,wt);  /* this also frees out nodes */
    out = NULL; /* for the next time around, if there is one */
ERROR:
    DP(DLEVEL5,("Search done for %s\n",wt->path));
    free(wt->path);
    free(wt);

    notrun();
    mutex_lock(&search_q_lk);
    if (search_pool_cnt > search_thr_limit) {
        mutex_unlock(&search_q_lk);
        DP(DLEVEL5,("Search thread exiting\n"));
        if (flags & FS_STATS) {
            mutex_lock(&stat_lk);
            st_destroy++;
            mutex_unlock(&stat_lk);
            }
            return(0);
        }
        else {
            search_pool_cnt++;
            while (!search_q)
                cond_wait(&search_q_cv,&search_q_lk);
            search_pool_cnt--;
            wt = search_q;  /* we have work to do! */
            if (search_q->next)
                search_q = search_q->next;
            else
                search_q = NULL;
            mutex_unlock(&search_q_lk);
        }
    }
    /*NOTREACHED*/
}

/*
 * Continue line: Speacial case search with the -C flag set. If you are
 * searching files like Makefiles, some lines may have escape char's to
 * contine the line on the next line. So the target string can be found, but
 * no data is displayed. This function continues to print the escaped line
 * until there are no more "\" chars found.
 */
int
continue_line(char *rline, FILE *fin, out_t *out, work_t *wt,
              int *lc, long *bc)
{
    int len;
    int cnt = 0;
    char *line;
    char nline[128];

    if (!(flags & FC_LINE))
        return(0);

    line = rline;
  AGAIN:
    len = strlen(line);
    if (line[len-1] == '\\') {
        if ((fgets(nline,127,fin)) == NULL) {
            return(cnt);
        }
        line = nline;
        len = strlen(line);
        if (line[len-1] == '\n')
            line[len-1] = '\0';
        *bc = *bc + len;
        *lc++;
        add_output_local(&out,wt,*lc,*bc,line);
        cnt++;
        goto AGAIN;
    }
    return(cnt);
}

/*
 * cascade: This thread is started by the main thread when directory names
 * are found on the work Q. The thread reads all the new file, and directory
 * names from the directory it was started when and adds the names to the
 * work Q. (it finds more work!)
 */
void *
```

```
cascade(void *arg)  /* work_t *arg */
{
    char        fullpath[1025];
    int         restart_cnt = 10;
    DIR         *dp;

    char        dir_buf[sizeof(struct dirent) + PATH_MAX];
    struct dirent *dent = (struct dirent *)dir_buf;
    struct stat   sbuf;
    char        *fpath;
    work_t      *wt;
    int         fl = 0, dl = 0;
    int         pm_file_len = 0;

    thr_setprio(thr_self(),64);  /* set search to middle */
    thr_yield();  /* try toi give control back to main thread */
    wt = (work_t *)arg;

    while(1) {
        fl = 0;
        dl = 0;
        restart_cnt = 10;
        pm_file_len = 0;

        mutex_lock(&running_lk);
        running++;
        mutex_unlock(&running_lk);
        mutex_lock(&work_q_lk);
        tglimit--;
        mutex_unlock(&work_q_lk);

if (!wt) {
    if (!(flags & FS_NOERROR))
        fprintf(stderr,"tgrep: Bad work node passed to cascade\n");
    goto DONE;
}
fpath = (char *)wt->path;
if (!fpath) {
    if (!(flags & FS_NOERROR))
        fprintf(stderr,"tgrep: Bad path name passed to cascade\n");
    goto DONE;
}
DP(DLEVEL3,("Cascading on %s\n",fpath));
if (( dp = opendir(fpath)) == NULL) {
    if (!(flags & FS_NOERROR))
        fprintf(stderr,"tgrep: Can't open dir %s, %s. Ignored.\n",
                fpath,strerror(errno));
    goto DONE;
}
while ((readdir_r(dp,dent)) != NULL) {
    restart_cnt = 10;  /* only try to restart the interupted 10 X */

    if (dent->d_name[0] == '.') {
        if (dent->d_name[1] == '.' && dent->d_name[2] == '\0')
            continue;
        if (dent->d_name[1] == '\0')
            continue;
    }

    fl = strlen(fpath);
    dl = strlen(dent->d_name);
    if ((fl + 1 + dl) > 1024) {
        fprintf(stderr,"tgrep: Path %s/%s is too long. "
                "MaxPath = 1024\n",
                fpath, dent->d_name);
        continue;  /* try the next name in this directory */
    }
    strcpy(fullpath,fpath);
    strcat(fullpath,"/");
    strcat(fullpath,dent->d_name);

RESTART_STAT:
    if (stat(fullpath,&sbuf)) {
        if (errno == EINTR) {
            if (--restart_cnt)
                goto RESTART_STAT;
        }
        if (!(flags & FS_NOERROR))
            fprintf(stderr,"tgrep: Can't stat file/dir %s, %s. "
                    "Ignored.\n",
                    fullpath,strerror(errno));
        goto ERROR;
    }

    switch (sbuf.st_mode & S_IFMT) {
    case S_IFREG :
        if (flags & TG_FILEPAT) {
            if (pmatch(pm_file_pat, dent->d_name, &pm_file_len)) {
                DP(DLEVEL3,("file pat match (cascade) %s\n",
                            dent->d_name));
                add_work(fullpath,FILET);
            }
        }
        else {
            add_work(fullpath,FILET);
```

```
                DP(DLEVEL3,("cascade added file (MATCH) %s to Work Q\n",
                           fullpath));
            }
            break;
        case S_IFDIR :
            DP(DLEVEL3,("cascade added dir %s to Work Q\n",fullpath));
            add_work(fullpath,DIRT);
            break;
        }
    }

ERROR:
    closedir(dp);
DONE:
    free(wt->path);
    free(wt);
    notrun();
    mutex_lock(&cascade_q_lk);
    if (cascade_pool_cnt > cascade_thr_limit) {
        mutex_unlock(&cascade_q_lk);
        DP(DLEVEL5,("Cascade thread exiting\n"));
        if (flags & FS_STATS) {
            mutex_lock(&stat_lk);
            st_cascade_destroy++;
            mutex_unlock(&stat_lk);
        }
        return(0); /* thr_exit */
    }
    else {
        DP(DLEVEL5,("Cascade thread waiting in pool\n"));
            cascade_pool_cnt++;
            while (!cascade_q)
                cond_wait(&cascade_q_cv,&cascade_q_lk);
            cascade_pool_cnt--;
            wt = cascade_q;  /* we have work to do! */
            if (cascade_q->next)
                cascade_q = cascade_q->next;
            else
                cascade_q = NULL;
            mutex_unlock(&cascade_q_lk);
        }
    }
    /*NOTREACHED*/
}

/*
 * Print Local Output: Called by the search thread after it is done searching
 * a single file. If any oputput was saved (matching lines), the lines are
 * displayed as a group on stdout.
 */
int
print_local_output(out_t *out, work_t *wt)
{
    out_t       *pp, *op;
    int         out_count = 0;
    int         printed = 0;
    int        print_name = 1;

    pp = out;
    mutex_lock(&output_print_lk);
    if (pp && (flags & TG_PROGRESS)) {
        progress++;
        if (progress >= progress_offset) {
            progress = 0;
            fprintf(stderr,".");
        }
    }
    while (pp) {
        out_count++;
        if (!(flags & FC_COUNT)) {
            if (flags & FL_NAMEONLY) {  /* Pint name ONLY ! */
                if (!printed) {
                    printed = 1;
                    printf("%s\n",wt->path);
                }
            }
            else {  /* We are printing more then just the name */
                if (!(flags & FH_HOLDNAME))  /* do not print name ? */
                    printf("%s :",wt->path);
                if (flags & FB_BLOCK)
                    printf("%ld:",pp->byte_count/512+1);
                if (flags & FN_NUMBER)
                    printf("%d:",pp->line_count);
                printf("%s\n",pp->line);
            }
        }
        op = pp;
        pp = pp->next;
        /* free the nodes as we go down the list */
        free(op->line);
        free(op);
    }
    mutex_unlock(&output_print_lk);
    mutex_lock(&global_count_lk);
    global_count += out_count;
```

```
    mutex_unlock(&global_count_lk);
    return(0);
}

/*
 * add output local: is called by a search thread as it finds matching lines.
 * the matching line, it's byte offset, line count, etc are stored until the
 * search thread is done searching the file, then the lines are printed as
 * a group. This way the lines from more then a single file are not mixed
 * together.
 */
int
add_output_local(out_t **out, work_t *wt,int lc, long bc, char *line)
{
    out_t       *ot,*oo, *op;

    if (( ot = (out_t *)malloc(sizeof(out_t))) == NULL)
        goto ERROR;
    if (( ot->line = (char *)malloc(strlen(line)+1)) == NULL)
        goto ERROR;

    strcpy(ot->line,line);
    ot->line_count = lc;
    ot->byte_count = bc;

    if (!*out) {
        *out = ot;
        ot->next = NULL;
        return(0);
    }
    /* append to the END of the list, keep things sorted! */
    op = oo = *out;
    while(oo) {
        op = oo;
        oo = oo->next;
    }
    op->next = ot;
    ot->next = NULL;
    return(0);
 ERROR:
    if (!(flags & FS_NOERROR))
        fprintf(stderr,"tgrep: Output lost. No space. "
                "[%s: line %d byte %d match : %s\n",
                wt->path,lc,bc,line);
    return(1);
}

/*
 * print stats: If the -S flag is set, after ALL files have been searched,
 * main thread calls this function to print the stats it keeps on how the
 * search went.
 */
void
prnt_stats(void)
{
    float a,b,c;
    float t = 0.0;
    time_t  st_end = 0;
    char    tl[80];

    st_end = time(NULL); /* stop the clock */
    fprintf(stderr,"\n----------------- Tgrep Stats. --------------------\n");
    fprintf(stderr,"Number of directories searched:          %d\n",
           st_dir_search);
    fprintf(stderr,"Number of files searched:                %d\n",
           st_file_search);
c = (float)(st_dir_search + st_file_search) / (float)(st_end - st_start);
fprintf(stderr,"Dir/files per second:                    %3.2f\n",
       c);
fprintf(stderr,"Number of lines searched:                %d\n",
       st_line_search);
fprintf(stderr,"Number of matching lines to target:      %d\n",
       global_count);

fprintf(stderr,"Number of cascade threads created:       %d\n",
       st_cascade);
fprintf(stderr,"Number of cascade threads from pool:     %d\n",
       st_cascade_pool);
a = st_cascade_pool; b = st_dir_search;
fprintf(stderr,"Cascade thread pool hit rate:            %3.2f%%\n",
       ((a/b)*100));
fprintf(stderr,"Cascade pool overall size:               %d\n",
       cascade_pool_cnt);
fprintf(stderr,"Cascade pool size limit:                 %d\n",
       cascade_thr_limit);
fprintf(stderr,"Number of cascade threads destroyed:     %d\n",
       st_cascade_destroy);

fprintf(stderr,"Number of search threads created:        %d\n",
       st_search);
fprintf(stderr,"Number of search threads from pool:      %d\n",
       st_pool);
a = st_pool; b = st_file_search;
fprintf(stderr,"Search thread pool hit rate:             %3.2f%%\n",
       ((a/b)*100));
```

```
    fprintf(stderr,"Search pool overall size:                %d\n",
           search_pool_cnt);
    fprintf(stderr,"Search pool size limit:                  %d\n",
           search_thr_limit);
    fprintf(stderr,"Number of search threads destroyed:      %d\n",
           st_destroy);

    fprintf(stderr,"Max # of threads running concurrenly:    %d\n",
           st_maxrun);
    fprintf(stderr,"Total run time, in seconds.              %d\n",
          (st_end - st_start));

    /* Why did we wait ? */
    a = st_workfds; b = st_dir_search+st_file_search;
    c = (a/b)*100; t += c;
    fprintf(stderr,"Work stopped due to no FD's:  (%.3d)       %d Times, %3.2f%%\n",
          search_thr_limit,st_workfds,c);
    a = st_worknull; b = st_dir_search+st_file_search;
    c = (a/b)*100; t += c;
    fprintf(stderr,"Work stopped due to no work on Q:        %d Times, %3.2f%%\n",
          st_worknull,c);
#ifndef __lock_lint  /* it is OK to read HERE with out the lock ! */
    if (tglimit == UNLIMITED)
        strcpy(tl,"Unlimited");
    else
        sprintf(tl,"   %.3d   ",tglimit);
#endif
    a = st_worklimit; b = st_dir_search+st_file_search;
    c = (a/b)*100; t += c;
    fprintf(stderr,"Work stopped due to TGLIMIT:  (%.9s) %d Times, %3.2f%%\n",
          tl,st_worklimit,c);
    fprintf(stderr,"Work continued to be handed out:         %3.2f%%\n",
           100.00-t);
    fprintf(stderr,"-----------------------------------------------------\n");
}

/*
 * not running: A glue function to track if any search threads or cascade
 * threads are running. When the count is zero, and the work Q is NULL,
 * we can safly say, WE ARE DONE.
 */
void
notrun (void)
{
    mutex_lock(&work_q_lk);
    work_cnt--;
    tglimit++;
    current_open_files++;
    mutex_lock(&running_lk);
    if (flags & FS_STATS) {
        mutex_lock(&stat_lk);
        if (running > st_maxrun) {
            st_maxrun = running;
            DP(DLEVEL6,("Max Running has increased to %d\n",st_maxrun));
        }
        mutex_unlock(&stat_lk);
    }
    running--;
    if (work_cnt == 0 && running == 0) {
        all_done = 1;
        DP(DLEVEL6,("Setting ALL_DONE flag to TRUE.\n"));
    }
    mutex_unlock(&running_lk);
    cond_signal(&work_q_cv);
    mutex_unlock(&work_q_lk);
}

/*
 * uncase: A glue function. If the -i (case insensitive) flag is set, the
 * target strng and the read in line is converted to lower case before
 * comparing them.
 */
void
uncase(char *s)
{
    char        *p;

    for (p = s; *p != NULL; p++)
        *p = (char)tolower(*p);
}


/*
 * SigThread: if the -S option is set, the first ^C set to tgrep will
 * print the stats on the fly, the second will kill the process.
 */

void *
SigThread(void *arg)
{
    int sig;
    int stats_printed = 0;

    while (1) {
        sig = sigwait(&set);
```

```
        DP(DLEVEL7,("Signal %d caught\n",sig));
        switch (sig) {
        case -1:
            fprintf(stderr,"Signal error\n");
            break;
        case SIGINT:
            if (stats_printed)
                exit(1);
            stats_printed = 1;
            sig_print_stats();
            break;
        case SIGHUP:
            sig_print_stats();
            break;
        default:
            DP(DLEVEL7,("Default action taken (exit) for signal %d\n",sig));
            exit(1);  /* default action */
        }
    }
}

void
sig_print_stats(void)
{
    /*
    ** Get the output lock first
    ** Then get the stat lock.
    */
    mutex_lock(&output_print_lk);
    mutex_lock(&stat_lk);
    prnt_stats();
    mutex_unlock(&stat_lk);
    mutex_unlock(&output_print_lk);
    return;
}

/*
 * usage: Have to have one of these.
 */
void
usage(void)
{
    fprintf(stderr,"usage: tgrep <options> pattern <{file,dir}>...\n");
    fprintf(stderr,"\n");
    fprintf(stderr,"Where:\n");
#ifdef DEBUG
    fprintf(stderr,"Debug     -d = debug level -d <levels> (-d0 for usage)\n");
    fprintf(stderr,"Debug     -f = block fd's from use (-f #)\n");
#endif
    fprintf(stderr,"          -b = show block count (512 byte block)\n");
    fprintf(stderr,"          -c = print only a line count\n");
    fprintf(stderr,"          -h = do not print file names\n");
    fprintf(stderr,"          -i = case insensitive\n");
    fprintf(stderr,"          -l = print file name only\n");
    fprintf(stderr,"          -n = print the line number with the line\n");
    fprintf(stderr,"          -s = Suppress error messages\n");
    fprintf(stderr,"          -v = print all but matching lines\n");
#ifdef NOT_IMP
    fprintf(stderr,"          -w = search for a \"word\"\n");
#endif
    fprintf(stderr,"          -r = Do not search for files in all "
                    "sub-directories\n");
    fprintf(stderr,"          -C = show continued lines (\"\\\")\n");
    fprintf(stderr,"          -p = File name regexp pattern. (Quote it)\n");
    fprintf(stderr,"          -P = show progress. -P 1 prints a DOT on stderr\n"
                    "               for each file it finds, -P 10 prints a DOT\n"
                    "               on stderr for each 10 files it finds, etc...\n");
    fprintf(stderr,"          -e = expression search.(regexp) More then one\n");
    fprintf(stderr,"          -B = limit the number of threads to TGLIMIT\n");
    fprintf(stderr,"          -S = Print thread stats when done.\n");
    fprintf(stderr,"          -Z = Print help on the regexp used.\n");
    fprintf(stderr,"\n");
    fprintf(stderr,"Notes:\n");
    fprintf(stderr,"      If you start tgrep with only a directory name\n");
    fprintf(stderr,"      and no file names, you must not have the -r option\n");
    fprintf(stderr,"      set or you will get no output.\n");
    fprintf(stderr,"      To search stdin (piped input), you must set -r\n");
    fprintf(stderr,"      Tgrep will search ALL files in ALL \n");
    fprintf(stderr,"      sub-directories. (like */* */*/* */*/*/* etc..)\n");
    fprintf(stderr,"      if you supply a directory name.\n");
    fprintf(stderr,"      If you do not supply a file, or directory name,\n");
    fprintf(stderr,"      and the -r option is not set, the current \n");
    fprintf(stderr,"      directory \".\" will be used.\n");
    fprintf(stderr,"      All the other options should work \"like\" grep\n");
    fprintf(stderr,"      The -p patten is regexp, tgrep will search only\n");
    fprintf(stderr,"      the file names that match the patten\n");
    fprintf(stderr,"\n");
    fprintf(stderr,"      Tgrep Version %s\n",Tgrep_Version);
    fprintf(stderr,"\n");
    fprintf(stderr,"      Copy Right By Ron Winacott, 1993-1995.\n");
    fprintf(stderr,"\n");
    exit(0);
}

/*
```

```
 * regexp usage: Tell the world about tgrep custom (THREAD SAFE) regexp!
 */
int
regexp_usage (void)
{
    fprintf(stderr,"usage: tgrep <options> -e \"pattern\" <-e ...> "
           "<{file,dir}>...\n");
    fprintf(stderr,"\n");
    fprintf(stderr,"metachars:\n");
    fprintf(stderr,"    . - match any character\n");
    fprintf(stderr,"    * - match 0 or more occurrences of pervious char\n");
    fprintf(stderr,"    + - match 1 or more occurrences of pervious char.\n");
    fprintf(stderr,"    ^ - match at begining of string\n");
    fprintf(stderr,"    $ - match end of string\n");
    fprintf(stderr,"    [ - start of character class\n");
    fprintf(stderr,"    ] - end of character class\n");
    fprintf(stderr,"    ( - start of a new pattern\n");
    fprintf(stderr,"    ) - end of a new pattern\n");
    fprintf(stderr,"    @(n)c - match <c> at column <n>\n");
    fprintf(stderr,"    | - match either pattern\n");
    fprintf(stderr,"    \\ - escape any special characters\n");
    fprintf(stderr,"    \\c - escape any special characters\n");
    fprintf(stderr,"    \\o - turn on any special characters\n");
    fprintf(stderr,"\n");
    fprintf(stderr,"To match two diffrerent patterns in the same command\n");
    fprintf(stderr,"Use the or function. \n"
           "ie: tgrep -e \"(pat1)|(pat2)\" file\n"
           "This will match any line with \"pat1\" or \"pat2\" in it.\n");
    fprintf(stderr,"You can also use up to %d -e expresions\n",MAXREGEXP);
    fprintf(stderr,"RegExp Pattern matching brought to you by Marc Staveley\n");
    exit(0);
}

/*
 * debug usage: If compiled with -DDEBUG, turn it on, and tell the world
 * how to get tgrep to print debug info on different threads.
 */
#ifdef DEBUG
void
debug_usage(void)
{
    int i = 0;

    fprintf(stderr,"DEBUG usage and levels:\n");
    fprintf(stderr,"-------------------------------------------------\n");
    fprintf(stderr,"Level                   code\n");
    fprintf(stderr,"-------------------------------------------------\n");
    fprintf(stderr,"0                 This message.\n");
    for (i=0; i<9; i++) {
        fprintf(stderr,"%d                %s\n",i+1,debug_set[i].name);
    }
    fprintf(stderr,"-------------------------------------------------\n");
    fprintf(stderr,"You can or the levels together like -d134 for levels\n");
    fprintf(stderr,"1 and 3 and 4.\n");
    fprintf(stderr,"\n");
    exit(0);
}
#endif
```

Multithreaded Quicksort

The following example `tquick.c`implements the quicksort algorithm using threads.

```
/*
*  Multithreaded Demo Source
*
*  Copyright (C) 1995 by Sun Microsystems, Inc.
*  All rights reserved.
*
*  This file is a product of SunSoft, Inc. and is provided for
*  unrestricted use provided that this legend is included on all
*  media and as a part of the software program in whole or part.
*  Users may copy, modify or distribute this file at will.
*
*  THIS FILE IS PROVIDED AS IS WITH NO WARRANTIES OF ANY KIND INCLUDING
*  THE WARRANTIES OF DESIGN, MERCHANTIBILITY AND FITNESS FOR A PARTICULAR
*  PURPOSE, OR ARISING FROM A COURSE OF DEALING, USAGE OR TRADE PRACTICE.
*
*  This file is provided with no support and without any obligation on the
*  part of SunSoft, Inc. to assist in its use, correction, modification or
*  enhancement.
*
*  SUNSOFT AND SUN MICROSYSTEMS, INC. SHALL HAVE NO LIABILITY WITH RESPECT
*  TO THE INFRINGEMENT OF COPYRIGHTS, TRADE SECRETS OR ANY PATENTS BY THIS
*  FILE OR ANY PART THEREOF.
*
*  IN NO EVENT WILL SUNSOFT OR SUN MICROSYSTEMS, INC. BE LIABLE FOR ANY
*  LOST REVENUE OR PROFITS OR OTHER SPECIAL, INDIRECT AND CONSEQUENTIAL
*  DAMAGES, EVEN IF THEY HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH
*  DAMAGES.
*
*  SunSoft, Inc.
```

```
*  2550 Garcia Avenue
*  Mountain View, California  94043
*/

/*
 * multiple-thread quick-sort.  See man page for qsort(3c) for info.
 * Works fine on uniprocessor machines as well.
 *
 *  Written by:  Richard Pettit (Richard.Pettit@West.Sun.COM)
 */

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

/* don't create more threads for less than this */
#define SLICE_THRESH   4096

/* how many threads per lwp */
#define THR_PER_LWP       4

/* cast the void to a one byte quanitity and compute the offset */
#define SUB(a, n)      ((void *) (((unsigned char *) (a)) + ((n) * width)))

typedef struct {
  void    *sa_base;
  int      sa_nel;
  size_t   sa_width;
  int    (*sa_compar)(const void *, const void *);
} sort_args_t;

/* for all instances of quicksort */
static int threads_avail;

#define SWAP(a, i, j, width) \
{ \
  int n; \
  unsigned char uc; \
  unsigned short us; \
  unsigned long ul; \
  unsigned long long ull; \
 \
  if (SUB(a, i) == pivot) \
    pivot = SUB(a, j); \
  else if (SUB(a, j) == pivot) \
    pivot = SUB(a, i); \
 \
  /* one of the more convoluted swaps I've done */ \
  switch(width) { \
  case 1: \
    uc = *((unsigned char *) SUB(a, i)); \
    *((unsigned char *) SUB(a, i)) = *((unsigned char *) SUB(a, j)); \
    *((unsigned char *) SUB(a, j)) = uc; \
    break; \
  case 2: \
    us = *((unsigned short *) SUB(a, i)); \
    *((unsigned short *) SUB(a, i)) = *((unsigned short *) SUB(a, j)); \
    *((unsigned short *) SUB(a, j)) = us; \
    break; \
  case 4: \
    ul = *((unsigned long *) SUB(a, i)); \
    *((unsigned long *) SUB(a, i)) = *((unsigned long *) SUB(a, j)); \
    *((unsigned long *) SUB(a, j)) = ul; \
    break; \
  case 8: \
    ull = *((unsigned long long *) SUB(a, i)); \
    *((unsigned long long *) SUB(a,i)) = *((unsigned long long *) SUB(a,j)); \
    *((unsigned long long *) SUB(a, j)) = ull; \
    break; \
  default: \
    for(n=0; n<width; n++) { \
      uc = ((unsigned char *) SUB(a, i))[n]; \
      ((unsigned char *) SUB(a, i))[n] = ((unsigned char *) SUB(a, j))[n]; \
      ((unsigned char *) SUB(a, j))[n] = uc; \
    } \
    break; \
  } \
}

static void *
_quicksort(void *arg)
{
  sort_args_t *sargs = (sort_args_t *) arg;
  register void *a = sargs->sa_base;
  int n = sargs->sa_nel;
  int width = sargs->sa_width;
  int (*compar)(const void *, const void *) = sargs->sa_compar;
  register int i;
  register int j;
  int z;
  int thread_count = 0;
  void *t;
  void *b[3];
  void *pivot = 0;
  sort_args_t sort_args[2];
```

```
thread_t tid;

/* find the pivot point */
switch(n) {
case 0:
case 1:
  return 0;
case 2:
  if ((*compar)(SUB(a, 0), SUB(a, 1)) > 0) {
    SWAP(a, 0, 1, width);
  }
  return 0;
case 3:
  /* three sort */
  if ((*compar)(SUB(a, 0), SUB(a, 1)) > 0) {
    SWAP(a, 0, 1, width);
  }
  /* the first two are now ordered, now order the second two */
  if ((*compar)(SUB(a, 2), SUB(a, 1)) < 0) {
    SWAP(a, 2, 1, width);
  }
  /* should the second be moved to the first? */
  if ((*compar)(SUB(a, 1), SUB(a, 0)) < 0) {
    SWAP(a, 1, 0, width);
  }
  return 0;
default:
  if (n > 3) {
    b[0] = SUB(a, 0);
    b[1] = SUB(a, n / 2);
    b[2] = SUB(a, n - 1);
    /* three sort */
    if ((*compar)(b[0], b[1]) > 0) {
      t = b[0];
      b[0] = b[1];
      b[1] = t;
    }
    /* the first two are now ordered, now order the second two */
    if ((*compar)(b[2], b[1]) < 0) {
      t = b[1];
      b[1] = b[2];
      b[2] = t;
    }
    /* should the second be moved to the first? */
    if ((*compar)(b[1], b[0]) < 0) {
      t = b[0];
      b[0] = b[1];
      b[1] = t;
    }
    if ((*compar)(b[0], b[2]) != 0)
      if ((*compar)(b[0], b[1]) < 0)
        pivot = b[1];
      else
        pivot = b[2];
  }
  break;
}
if (pivot == 0)
  for(i=1; i<n; i++) {
    if (z = (*compar)(SUB(a, 0), SUB(a, i))) {
      pivot = (z > 0) ? SUB(a, 0) : SUB(a, i);
      break;
    }
  }
if (pivot == 0)
  return;

/* sort */
i = 0;
j = n - 1;
while(i <= j) {
  while((*compar)(SUB(a, i), pivot) < 0)
    ++i;
  while((*compar)(SUB(a, j), pivot) >= 0)
    --j;
  if (i < j) {
    SWAP(a, i, j, width);
    ++i;
    --j;
  }
}

/* sort the sides judiciously */
switch(i) {
case 0:
case 1:
  break;
case 2:
  if ((*compar)(SUB(a, 0), SUB(a, 1)) > 0) {
    SWAP(a, 0, 1, width);
  }
  break;
case 3:
  /* three sort */
  if ((*compar)(SUB(a, 0), SUB(a, 1)) > 0) {
```

```
    SWAP(a, 0, 1, width);
  }
  /* the first two are now ordered, now order the second two */
  if ((*compar)(SUB(a, 2), SUB(a, 1)) < 0) {
    SWAP(a, 2, 1, width);
  }
  /* should the second be moved to the first? */
  if ((*compar)(SUB(a, 1), SUB(a, 0)) < 0) {
    SWAP(a, 1, 0, width);
  }
  break;
default:
  sort_args[0].sa_base          = a;
  sort_args[0].sa_nel           = i;
  sort_args[0].sa_width         = width;
  sort_args[0].sa_compar        = compar;
  if ((threads_avail > 0) && (i > SLICE_THRESH)) {
    threads_avail--;
    thr_create(0, 0, _quicksort, &sort_args[0], 0, &tid);
    thread_count = 1;
  } else
    _quicksort(&sort_args[0]);
  break;
}
j = n - i;
switch(j) {
case 1:
  break;
case 2:
  if ((*compar)(SUB(a, i), SUB(a, i + 1)) > 0) {
    SWAP(a, i, i + 1, width);
  }
  break;
case 3:
  /* three sort */
  if ((*compar)(SUB(a, i), SUB(a, i + 1)) > 0) {
    SWAP(a, i, i + 1, width);
  }
  /* the first two are now ordered, now order the second two */
  if ((*compar)(SUB(a, i + 2), SUB(a, i + 1)) < 0) {
    SWAP(a, i + 2, i + 1, width);
  }
  /* should the second be moved to the first? */
  if ((*compar)(SUB(a, i + 1), SUB(a, i)) < 0) {
    SWAP(a, i + 1, i, width);
  }
  break;
default:
  sort_args[1].sa_base          = SUB(a, i);
  sort_args[1].sa_nel           = j;
  sort_args[1].sa_width         = width;
  sort_args[1].sa_compar        = compar;
  if ((thread_count == 0) && (threads_avail > 0) && (i > SLICE_THRESH)) {
    threads_avail--;
    thr_create(0, 0, _quicksort, &sort_args[1], 0, &tid);
    thread_count = 1;
  } else
    _quicksort(&sort_args[1]);
  break;
}
if (thread_count) {
  thr_join(tid, 0, 0);
  threads_avail++;
}
return 0;
}

void
quicksort(void *a, size_t n, size_t width,
          int (*compar)(const void *, const void *))
{
  static int ncpus = -1;
  sort_args_t sort_args;

  if (ncpus == -1) {
    ncpus = sysconf( _SC_NPROCESSORS_ONLN);

    /* lwp for each cpu */
    if ((ncpus > 1) && (thr_getconcurrency() < ncpus))
      thr_setconcurrency(ncpus);

    /* thread count not to exceed THR_PER_LWP per lwp */
    threads_avail = (ncpus == 1) ? 0 : (ncpus * THR_PER_LWP);
  }
  sort_args.sa_base = a;
  sort_args.sa_nel = n;
  sort_args.sa_width = width;
  sort_args.sa_compar = compar;
  (void) _quicksort(&sort_args);
}
```