sun®
microsystems

System Services Overview

Part Number: 800-1753-10
Revision A, of 9 May 1988

Sun equipment generates, uses, and can radiate radio frequency energy and if not installed and used in accordance with the instructions manual, may cause interference to radio communications. It has been tested and found to comply with the limits for a Class A computing device pursuant to Subpart J of Part 15 of FCC Rules, which are designed to provide reasonable protection against such interference when operated in a commercial environment. Operation of Sun equipment in a residential area is likely to cause interference in which case the user at his own expense will be required to take whatever measures may be required to correct the interference.

Contents

Figures

1

Introduction

1

Introduction

Facilities available to a SunOS user process can be logically divided into groups: facilities implemented by code running in the operating system, kernel code in libraries, and other system facilities implemented by the system in cooperation with a *server* or *daemon* process. Such server processes are described in greater detail, and the standard system daemons are introduced, in the *Network-Based Services* section of this manual. For a more extended discussion, see the *Network Services* chapter of *Network Programming*.

Facilities implemented in the kernel itself are those which define the SunOS virtual machine in which processes runs. Like many real machines, the SunOS virtual machine has memory management hardware, an interrupt facility, timers and counters. The SunOS virtual machine also allows access to files and other objects through a set of *descriptors*. Each descriptor resembles a device controller, and supports a set of operations. Like devices on real machines, some of which are internal to the machine and some of which are external, parts of the descriptor machinery are built-in to the operating system, while other parts are often implemented in server processes on other machines.

System abstractions described in this manual are:

Directories
A directory is an object in the file system name space. Operations on files and other named objects in a file system are always specified relative to a working directory.

Files
Files are used to store uninterpreted sequence of bytes on which random access `read()` and `write()` operations can be made. Pages from files or devices may also be mapped into process address space.

Communications Domains
A communications domain represents an interprocess communications environment, such as socket-based communications facilities, communications in the Internet or the resource sharing protocols and access rights of a resource sharing system on a local network.

Sockets
A socket is an endpoint of communication and the focal point for Inter-Process Communication (IPC) in a communications domain. Sockets may be created in pairs, or given names and used to rendezvous with other

sockets in a communications domain, accepting connections from these sockets or exchanging messages with them. These operations model a labeled or unlabeled communications graph, and can be used in a wide variety of communications domains. Sockets can have different *types* to provide different semantics of communication, increasing the flexibility of the model.

Terminals and other devices
Devices include terminals, providing input editing and interrupt generation and output flow control and editing, magnetic tapes, disks and other peripherals. They often support the generic `read()` and `write()` operations as well as a number of `ioctl()`'s. (A much more substantive discussion of devices can be found in the *Writing Device Drivers* manual).

Processes
Process identifiers provide the handles needed to schedule, control, execute and debug programs.

Lightweight Processes
Lightweight Processes provide an efficient, user-level facility for managing multiple program threads within one real SunOS process.

1.1. Manual Organization

The first few chapters of this manual discuss two key introductory issues — the relationship of SunOS to AT&T System V and the role of server (daemon) processes in SunOS.

There follows a long chapter — *SunOS Kernel Interface* — which introduces the bulk of the important kernel interfaces. Memory-related interfaces, are also dealt with in the course of the detailed discussion contained in the following *Memory Management* chapter. There follows another detailed section on Lightweight Processes.

1.2. Notation and Types

The notation used to describe system calls is a variant of a C language function declaration, consisting of a prototype call followed by declaration of parameters and results. An additional keyword `result`, not part of the normal C language, is used to indicate which of the declared entities receive results. As an example, consider the `read()` call, as described in section *3.8.1*.

```
cc = read(fd, buf, nbytes);
    result int cc;
    int fd, nbytes;
    result char *buf;
```

The first line shows how the `read()` routine is called, with three parameters. As shown on the second line `cc` is an integer and `read()` also returns information in the parameter `buf`.

Description of all error conditions arising from each system call is not provided here; they appear in the `intro(2)` manual page of the *SunOS Reference Manual*. All error codes also appear in the index to the *SunOS Reference Manual*. In particular, when accessed from the C language, many calls return a characteristic –1 value when an error occurs, returning the error code in the global variable errno. Since some calls return –1 as a legitimate value, you may have to check errno to determine if the return value is genuine or an error. Other languages may present errors in different ways.

A number of system standard types are defined in the <sys/types.h> include file and used in the specifications here and in many C programs. These include caddr_t giving a memory address, off_t giving a file offset, and a set of unsigned types u_char, u_short, u_int and u_long, shorthand names for unsigned char, unsigned short, and so on.

2

System V Compatibility

2

System V Compatibility

SunOS contains almost all AT&T System V functionality and features. This chapter *very briefly* introduces the few remaining incompatibilities and lists the major new system-level functions that are available to users.

For an introduction to STREAMS and STREAM-related systems facilities, see the *Writing Device Drivers* manual.

For an introduction to System V compatible shared memory, see the *Memory Management* section of this manual.

(For a more complete discussion of System V functionality, one which includes libraries and applications as well as system facilities, see the *System V Programming* section of the *Programming Utilities and Libraries* manual).

2.1. BSD/System V Incompatibilities

The following systems calls suffer incompatibilities between BSD and System V variants. SunOS supports both incompatible versions.

`getpgrp()/setpgrp()`:

The BSD version of `getpgrp()` takes a single argument, which is interpreted as a process ID; `getpgrp()` returns the process group ID of that process. The System V version of `getpgrp()` takes no arguments, and returns the process group ID of the current process. There is no way to get the BSD behavior of `getpgrp()` in all cases using the same code in both environments. However, the BSD `getpgrp()` is rarely called with an argument other than zero or the current process's process ID; since the behavior of the System V `getpgrp()` is the same as that of the BSD version when an argument of zero or the current process's process ID is provided, and since the System V `getpgrp()` ignores any extra argument passed to it, calling `getpgrp()` with a zero argument will give the same behavior in both environments.

The BSD version of `setpgrp()` takes two arguments, which are interpreted as a process ID and a process group ID, respectively. It sets the process group ID of the process specified by the first argument to the value of the second argument. The System V version of `setpgrp()` takes no arguments. It sets the process group ID of the current process equal to its process ID, detaches the current process from its controlling terminal, arranges that the next terminal that it opens that is not already a controlling terminal will become the controlling terminal for the new process group, and arranges that when the current process exits, a `SIGHUP` will be sent to all processes in the new process group. There is no way to get the BSD behavior of `setpgrp()` in the System V environment, and there is no

`open()`:

In the BSD environment, opening a file with the `O_NDELAY` bit set in the open mode does not leave the file descriptor returned, nor the object referred to by that file descriptor, in non-blocking mode. In the System V environment it does. Furthermore, the form of no-delay mode selected by the `O_NDELAY` flag differs between the two environments, as described below under `read()` and `write()`. The `O_NDELAY` bit has a different value in the two environments; the BSD behavior is available in both environments if the `FNDELAY` bit is used, and the System V behavior is available in both environments if the `FNBIO` bit is used.

`read()`:

In the BSD environment, a `read()` on a file descriptor or from an object in no-delay mode will return –1 and set `errno` to `EWOULDBLOCK` if no data is available. In the System V environment, the `read()` will return a count of 0, which is indistinguishable from end-of-file, unless the `read()` is on a STREAMS device other than a terminal, in which case the `read()` will return –1 and set `errno` to `EAGAIN`. Again, the BSD behavior is available in both environments if the `FNDELAY` bit is used in an `fcntl()` `F_SETFL` call, and the System V behavior is available in both environments if the `FNBIO` bit is used in an `fcntl()` `F_SETFL` call.

`write()`:

As with `read()`, the two environments differ in how a `write()` in no-delay mode indicates that there is no buffer space available to store the data to be written, and as with `read()`, the BSD or System V behavior is available in both environments if the `FNDELAY` or `FNBIO` bits are used.

`fcntl()`:

In the BSD environment, using the `F_SETFL` `fcntl()` call to turn the `O_NDELAY` flag on turns it on for the object referred to by the file descriptor given as the first argument to `fcntl()`, so that all other file descriptors referring to that object will also act as if they were in no-delay mode, although an `F_GETFL` `fcntl()` call will not indicate that no-delay mode is on. In the System V environment, it only turns no-delay mode on for the file descriptor given as the first argument to `fcntl()` and file descriptors created from that file descriptor by `dup()`. Furthermore, the form of no-delay mode selected by the `O_NDELAY` flag differs between the two environments, as described above under `read()` and `write()`. As described above, the `FNDELAY` flag can be used to select BSD-style no-delay mode in either environment, and the `FNBIO` flag can be used to select System V-style no-delay mode in either environment.

2.2. Major System V Derived Facilities

(See the *System V Programming* section of the *Programming Utilities and Libraries* manual for a more detailed discussions of these System V derived facilities).

System V introduced first-in-first-out (FIFO) files, which are also called named pipes. This allows processes to open this special file, using it for communication just like a pipe, but between possibly unrelated processes. FIFO files are created using the `mknod()` system call.

The System V shared-memory facility provides common areas in memory for sharing data between processes. Facilities are also provided to control access to shared memory, and to synchronize updating by multiple processes. Shared memory is useful for database applications, among other things.

The System V semaphore facility provides a process synchronization mechanism, which can be used to schedule processes that modify shared system resources. Resources are locked for updating by one process at a time, and update ordering is supported.

The System V message queue facility provides another form of inter-process communication. Messages are a convenient way for unrelated processes to share data. Since only the message is shared, processes can remain independent of each other's internal data structures. The message queue facility is used to establish one or more queues for inter-process communication. Messages are placed on specific queues for subsequent retrieval. A control data structure is included to allow restricted access to individual message queues.

3

SunOS Kernel Interface

3

SunOS Kernel Interface

3.1. Processes and Protection

Host and Process Identifiers

Each SunOS host has associated with it a 32-bit host id, and a host name of up to MAXHOSTNAMELEN characters (as defined in <sys/param.h>). The host name is accessed and modified with the calls:

```
getdomainname(name, namelength);
     char *name;
     int namelength;

setdomainname(name, namelength);
     char *name;
     int namelength;

hostid = gethostid();
     result long hostid;

sethostname(name, len);
     char *name;
     int len;

gethostname(buf, buflen);
     result char *buf;
     int buflen;
```

getdomainname() returns the name of the domain for the current processor. setdomainname() sets the domain of the current processor to name.

The *buf* containing the host name returned by gethostname() is null-terminated (if space allows).

On each host runs a set of *processes*. Each process is largely independent of other processes, having its own protection domain, address space, timers, and an independent set of references to system or user implemented objects.

Each process in a host is named by an integer called the *process id*. This number is in the range 1-30000. A process can discover its process id with the getpid() routine:

```
pid = getpid();
    result int pid;
```

On each SunOS host this identifier is guaranteed to be unique; in a multi-host environment, the (hostid, process id) pairs are guaranteed unique.

Creating and Terminating Processes

A new process is usually created by copying that mappings that define the address space of a *parent* process, thus making a logical duplicate of the parent. (See the *Memory Management* chapter for a description of mapping).

```
pid = fork();
    result int pid;
```

The `fork()` call returns twice, once in the parent process, where *pid* is the process identifier of the child, and once in the child process where *pid* is 0.

Since `execve()` (see below) specifies `MAP_PRIVATE` on all the mappings it performs, parent and child effectively have copy-on-write access to a single set of objects. Any `MAP_SHARED` mappings in the parent are also `MAP_SHARED` in the child, providing the opportunity for both parent and child to operate on a common object. The parent-child relationship induces a hierarchical structure on the set of processes in the system.

A process may terminate by executing an `exit()` call:

```
exit(status);
    int status;
```

returning 8 bits of exit status to its parent.

When a child process exits or terminates abnormally, the parent process receives information about any event which caused termination of the child process. A second call provides a non-blocking interface and may also be used to retrieve information about resources consumed by the process during its lifetime.

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

pid = wait(astatus);
    result int pid;
    result union wait *astatus;

pid = wait3(astatus, options, arusage);
    result int pid;
    result union waitstatus *astatus;
int options; result struct rusage *arusage;
```

A process can overlay itself with the memory image of another program, passing the newly created process a set of parameters, using the call:

```
execve(name, argv, envp)
    char *name, **argv, **envp;
```

execve() specifies MAP_PRIVATE on the mappings which overlay the old address space. execve() performs this operation by performing the internal equivalent of an mmap() to the file containing the program. The text and initialized data segments are mapped to the file, and the program's uninitialized data and stack areas are mapped to unnamed objects in the system's virtual memory. The boundaries of the mappings it establishes are recorded as representing the traditional "segments" of a UNIX process's address space.

The text segment is mapped with only PROT_READ and PROT_EXECUTE protections, so that write references to the text produce segmentation violations. The data segment is mapped as writable; however any page of initialized data that does not get written may be shared among all the processes running the program.

The specified *name* must be a file which is in a format recognized by the system, either a binary executable file or a ASCII file which causes the execution of a specified interpreter program (usually sh(1) or csh(1)) to process its contents.

User and Group Ids

Each process in the system has associated with it two user-id's: a *real user id* and an *effective user id*, both non-negative 16 bit integers. (Note: a user may change their *effective user id*, for example with the su(1) command, but this does *not* change their *real user id*). Each process has an *real accounting group id* and an *effective accounting group id* and a set of *access group id's*. The group id's are non-negative 16 bit integers. Each process may be in several different access groups, with the maximum concurrent number of access groups a system compilation parameter, the constant NGROUPS in the file <sys/param.h>, guaranteed to be at least 8.

The real and effective user ids associated with a process are returned by:

```
ruid = getuid();
    result int ruid;

euid = geteuid();
    result int euid;
```

the real and effective accounting group ids by:

```
rgid = getgid();
    result int rgid;

egid = getegid();
    result int egid;
```

and the access group id set is returned by a getgroups() call:

```
ngroups = getgroups(gidsetsize, gidset);
    result int ngroups, gidset[gidsetsize];
    int gidsetsize;
```

The user and group id's are assigned at login time using the `setreuid()`, `setregid()`, and `setgroups()` calls:

```
setreuid(ruid, euid);
    int ruid, euid;

setregid(rgid, egid);
    int rgid, egid;

setgroups(gidsetsize, gidset);
    int gidsetsize, gidset[gidsetsize];
```

The `setreuid()` call sets both the real and effective user-id's, while the `setregid()` call sets both the real and effective accounting group id's. Unless the caller is the super-user, *ruid* must be equal to either the current real or effective user-id, and *rgid* equal to either the current real or effective accounting group id. The `setgroups()` call is restricted to the super-user.

Process Groups and System Terminals

Each process in the system is also normally associated with a *process group*. The group of processes in a process group is sometimes referred to as a *job* and manipulated by high-level system software (such as the shell). The current process group of a process is returned by the `getpgrp()` call:

```
pgrp = getpgrp(pid);
    result int pgrp;
    int pid;
```

The process group associated with a process may be changed by the `setpgrp()` call:

```
setpgrp(pid, pgrp);
    int pid, pgrp;
```

Newly created processes are assigned process id's distinct from all processes and process groups, and the same process group as their parent. A normal (unprivileged) process may set its process group equal to its process id. A privileged process may set the process group of any process to any value.

When a process is in a specific process group it may receive software interrupts affecting the group, causing the group to suspend or resume execution or to be interrupted or terminated. In particular, every system terminal has a process group and only processes which are in the process group of a terminal may read from the terminal, allowing arbitration of terminals among several different jobs. A process can examine the process group of a terminal via the `ioctl()` call:

```
ioctl(fd, TIOCGPGRP, pgrp);
    int fd;
    result int *pgrp;
```

A process may change the process group of any terminal which it can write by the `ioctl()` call:

```
ioctl(fd, TIOCSPGRP, pgrp);
    int fd, *pgrp;
```

The terminal's process group may be set to any value. Thus, more than one terminal may be in a process group.

Control Terminal

Each process in the system is usually associated with a *control terminal*, accessible through the file `/dev/tty`. A newly created process inherits the control terminal of its parent. A process may be in a different process group than its control terminal, in which case the process does not receive software interrupts affecting the control terminal's process group.

You can arrange for a process to be detached from the control terminal, via this code sequence:

```
if ((i = open("/dev/tty"), O_RDONLY) >= 0)
    (void)ioctl(i, TIOCNOTTY, (char *)0);
```

3.2. Signals

The system defines a set of *signals* that may be delivered to a process. Signal delivery resembles the occurrence of a hardware interrupt: the signal is blocked from further occurrence, the current process context is saved, and a new one is built. A process may specify the *handler* to which a signal is delivered, or specify that the signal is to be *blocked* or *ignored*. A process may also specify that a *default* action is to be taken when signals occur.

Some signals will cause a process to exit when they are not caught. This may be accompanied by creation of a `core` image file, containing the current memory image of the process for use in post-mortem debugging. A process may choose to have signals delivered on a special stack, so that sophisticated software stack manipulations are possible.

All signals have the same *priority*. If multiple signals are pending simultaneously, the order in which they are delivered to a process is implementation specific. Signal routines execute with the signal that caused their invocation *blocked*, but other signals may yet occur. Mechanisms are provided whereby critical sections of code may protect themselves against the occurrence of specified signals.

Signal Types

The signals defined by the system fall into one of five classes: hardware conditions, software conditions, input/output notification, process control, or resource control. The set of signals is defined in the file <signal.h>.

Hardware signals are derived from exceptional conditions which may occur during execution. Such signals include SIGFPE representing floating point and other arithmetic exceptions, SIGILL for illegal instruction execution, SIGSEGV for addresses outside the currently assigned area of memory, and SIGBUS for accesses that violate memory protection constraints. Other, more cpu-specific hardware signals exist, such as SIGIOT, SIGEMT, and SIGTRAP.

Software signals reflect interrupts generated by user request: SIGINT for the normal interrupt signal; SIGQUIT for the more powerful quit signal, that normally causes a core image to be generated; SIGHUP and SIGTERM that cause graceful process termination, either because a user has "hung up", or by user or program request; and SIGKILL, a more powerful termination signal which a process cannot catch or ignore. Programs may define their own asynchronous events using SIGUSR1 and SIGUSR2. Other software signals (SIGALRM, SIGVTALRM, SIGPROF) indicate the expiration of interval timers.

A process can request notification via a SIGIO signal when input or output is possible on a descriptor, or when a *non-blocking* operation completes. A process may request to receive a SIGURG signal when an urgent condition arises.

A process may be *stopped* by a signal sent to it or the members of its process group. The SIGSTOP signal is a powerful stop signal, because it cannot be caught. Other stop signals SIGTSTP, SIGTTIN, and SIGTTOU are used when a user request, input request, or output request respectively is the reason for stopping the process. A SIGCONT signal is sent to a process when it is continued from a stopped state. Processes may receive notification with a SIGCHLD signal when a child process changes state, either by stopping or by terminating.

Exceeding resource limits may cause signals to be generated. SIGXCPU occurs when a process nears its CPU time limit and SIGXFSZ warns that the limit on file size creation has been reached.

Signal Handlers

A process has a handler associated with each signal. The handler controls the way the signal is delivered. The call

```
#include <signal.h>

struct sigvec {
    int (*sv_handler)();
    int sv_mask;
    int sv_flags;
};

sigvec(signo, sv, osv)
    int signo;
    struct sigvec *sv;
    result struct sigvec *osv;
```

assigns interrupt handler address sv_handler to signal *signo*. Each handler address specifies either an interrupt routine for the signal, that the signal is to be ignored, or that a default action (usually process termination) is to occur if the signal occurs. The constants `SIG_IGN` and `SIG_DFL` used as values for `sv_handler` cause ignoring or defaulting of a condition.

NOTE *There are two things that must be done to reset a signal handler from within a signal handler. Resetting the routine that catches the signal, which*

```
signal(n, SIG_DFL);
```

does, is only the first. It's also necessary to unblock the blocked signal, which is done with `sigsetmask()` *or* `sigblock()`*. The way to think of signals is as hardware interrupts. Just resetting the vector for the interrupt is not enough, you also have to lower the processor priority level.*

The `sv_mask` and `sv_onstack` values specify the signal mask to be used when the handler is invoked; it implicitly includes the signal which invoked the handler. Signal masks include one bit for each signal; the mask for a signal *signo* is provided by the macro `sigmask`(*signo*), from `<signal.h>`. `sv_flags` specifies whether system calls should be restarted if the signal handler returns and whether the handler should operate on the normal run-time stack or a special signal stack (see below). If *osv* is non-zero, the previous signal vector is returned. It also specifies whether the signal action is to be reset to `SIG_DFL`, and if the signal is to be blocked by setting a bit to the signal mask, when the signal handler is called. This latter behavior is the default; the former is for backward compatibility with the signal mechanisms of some other versions of the UNIX system (V7, BSD4.1, System V, etc.).

When a signal condition arises for a process, the signal is added to a set of signals pending for the process. If the signal is not currently *blocked* by the process then it will be delivered. The process of signal delivery adds the signal to be delivered and those signals specified in the associated signal handler's `sv_mask` to a set of those *masked* for the process, saves the current process context, and places the process in the context of the signal handling routine. The call is arranged so that if the signal handling routine exits normally the signal mask will be restored and the process will resume execution in the original context. If the process wishes to resume in a different context, then it must arrange to restore the signal mask itself.

The mask of *blocked* signals is independent of handlers for delays. It delays the delivery of signals much as a raised hardware interrupt priority level delays hardware interrupts. Preventing an interrupt from occurring by changing the handler is analogous to disabling a device from further interrupts.

The signal handling routine `sv_handler` is called by a C call of the form

```
(*sv_handler)(signo, code, scp, addr);
    int signo, code;
    struct sigcontext *scp;
    char *addr;
```

The *signo* gives the number of the signal that occurred, while *code*, is a parameter of certain signals that provides additional detail. The *scp* parameter is a pointer to a machine-dependent structure containing the information for restoring the context from before the signal. *addr* is additional address information.

Sending Signals

A process can send a signal to another process or group of processes with the calls:

```
kill(pid, signo);
    int pid, signo;

killpgrp(pgrp, signo);
    int pgrp, signo;
```

Unless the process sending the signal is privileged, it must have the same effective user id as the process receiving the signal.

Signals can also be sent from from a terminal device to the process group associated with the terminal. See `kill(1)`.

Protecting Critical Sections

To block a section of code against one or more signals, a `sigblock()` call may be used to add a set of signals to the existing mask, returning the old mask:

```
oldmask = sigblock(mask);
    result long oldmask;
    long mask;
```

The old mask can then be restored later with `sigsetmask()`,

```
oldmask = sigsetmask(mask);
    result long oldmask;
    long mask;
```

The `sigblock()` call can be used to read the current mask by specifying an empty *mask*.

It is possible to check conditions with some signals blocked, and then to pause waiting for a signal and restoring the mask, by using:

```
sigpause(mask);
    long mask;
```

Signal Stacks

Applications that maintain complex or fixed size stacks can use the call

```
struct sigstack {
    caddr_t ss_sp;
    int ss_onstack;
};

sigstack(ss, oss)
    struct sigstack *ss;
    result struct sigstack *oss;
```

to provide the system with a stack based at *ss_sp* for delivery of signals. The value *ss_onstack* indicates whether the process is currently on the signal stack, a notion maintained in software by the system.

When a signal is to be delivered, the system checks whether the process is on a signal stack. If not, then the process is switched to the signal stack for delivery, with the return from the signal arranged to restore the previous stack.

If the process wishes to take a non-local exit from the signal routine, or run code from the signal stack that uses a different stack, a `sigstack()` call should be used to reset the signal stack.

3.3. Timers

Real Time

The system's notion of the current Greenwich time and the current time zone is set and returned by the calls:

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

settimeofday(tvp, tzp);
    struct timeval *tp;
    struct timezone *tzp;

gettimeofday(tp, tzp);
    result struct timeval *tp;
    result struct timezone *tzp;
```

where the structures are defined in `<sys/time.h>` as:

```
struct timeval {
    long    tv_sec; /* seconds since Jan 1, 1970 */
    long    tv_usec;     /* and microseconds */
};

struct timezone {
    int tz_minuteswest; /* of Greenwich */
    int tz_dsttime; /* type of dst correction to apply */
};
```

The precision of the system clock is hardware dependent. Earlier versions of the UNIX system contained only a 1-second resolution version of this call, which remains as a library routine:

```
time(tvp)
    result long *tvp;
```

or

```
tv = time((long *)0);
    result long tv;
```

returning only the `tv_sec` field from the `gettimeofday()` call.

Interval Time

The system provides each process with three interval timers, defined in `<sys/time.h>`:

```
#define ITIMER_REAL      0    /* real time intervals */
#define ITIMER_VIRTUAL   1    /* virtual time intervals */
#define ITIMER_PROF      2    /* user and system virtual time */
```

The `ITIMER_REAL` timer decrements in real time. It could be used by a library routine to maintain a wakeup service queue. A `SIGALRM` signal is delivered when this timer expires.

The `ITIMER_VIRTUAL` timer decrements in process virtual time. It runs only when the process is executing. A `SIGVTALRM` signal is delivered when it expires.

The `ITIMER_PROF` timer decrements both in process virtual time and when the system is running on behalf of the process. It is designed to be used by processes to statistically profile their execution. A `SIGPROF` signal is delivered when it expires.

A timer value is defined by the `itimerval` structure:

```
struct itimerval {
    struct  timeval it_interval;    /* timer interval */
    struct  timeval it_value;    /* current value */
};
```

and a timer is set or read by the call:

```
getitimer(which, value);
    int which;
    result struct itimerval *value;

setitimer(which, value, ovalue);
    int which;
    struct itimerval *value;
    result struct itimerval *ovalue;
```

The third argument to `setitimer()` specifies an optional structure to receive the previous contents of the interval timer. A timer can be disabled by specifying a timer value of 0.

The system rounds argument timer intervals to be not less than the resolution of its clock. This clock resolution can be determined by loading a very small value into a timer and reading the timer back to see what value resulted.

The alarm() system call of earlier versions of the UNIX system is provided as a library routine using the ITIMER_REAL timer. The process profiling facilities of earlier versions of the UNIX system remain because it is not always possible to guarantee the automatic restart of system calls after receipt of a signal. The profil() call arranges for the kernel to begin gathering execution statistics for a process:

```
profil(buf, bufsize, offset, scale);
    result char *buf;
    int bufsize, offset, scale;
```

This begins sampling of the program counter, with statistics maintained in the user-provided buffer.

3.4. Descriptors

Each process has access to resources through *descriptors*. Each descriptor is a handle allowing the process to reference objects such as files, devices and communications links.

The Reference Table

Rather than allowing processes direct access to descriptors, the system introduces a level of indirection, so that descriptors may be shared between processes. Each process has a *descriptor reference table*, containing pointers to the actual descriptors. The descriptors themselves thus have multiple references, and are reference counted by the system.

Each process has a fixed size descriptor reference table, where the size is returned by the getdtablesize() call:

```
nds = getdtablesize();
    result int nds;
```

and guaranteed to be at least 20. The entries in the descriptor reference table are referred to by small integers; for example if there are 20 slots they are numbered 0 to 19.

Descriptor Properties

Each descriptor has a logical set of properties maintained by the system and defined by its *type*. Each type supports a set of operations; some operations, such as reading and writing, are common to several abstractions, while others are unique. Generic operations applying to many of these types are described in *3.8*. Naming contexts, files and directories are described in *3.9*. Section *4.1*. describes communications domains and sockets. Terminals and (structured and unstructured) devices are described in *3.10*.

Managing Descriptor References

A duplicate of a descriptor reference may be made by doing

```
new = dup(old);
    result int new;
    int old;
```

returning a copy of descriptor reference *old* indistinguishable from the original. The *new* chosen by the system will be the smallest unused descriptor reference slot. A copy of a descriptor reference may be made in a specific slot by doing

```
dup2(old, new);
    int old, new;
```

The dup2() call causes the system to deallocate the descriptor reference currently occupying slot *new*, if any, replacing it with a reference to the same descriptor as old. This deallocation is also performed by:

```
close(old);
    int old;
```

Multiplexing Requests

The system provides a standard way to perform synchronous and asynchronous multiplexing of operations.

Note: Operations are said to be multiplexed when they are interleaved in real time on the same device or communications channel. For example, I/O streams A and B are multiplexed if B begins before A is completed.

Synchronous multiplexing is performed by using the select() call to examine the state of multiple descriptors simultaneously, and to wait for state changes on those descriptors. Sets of descriptors of interest are specified as bit masks, as follows:

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

nds = select(nd, in, out, except, tvp);
    result int nds;
    int nd;
    fd_set *in, *out, *except;
    struct timeval *tvp;

FD_ZERO(&fdset);
FD_SET(fd, &fdset);
FD_CLR(fd, &fdset);
FD_ISSET(fd, &fdset);
    int fs;
    fs_set fdset;
```

The select() call examines the descriptors specified by the sets *in*, *out* and *except*, replacing the specified bit masks by the subsets that select true for input, output, and exceptional conditions respectively (*nd* indicates the number of file descriptors specified by the bit masks). If any descriptors meet the following criteria, then the number of such descriptors is returned in *nds* and the bit masks are updated.

- A descriptor selects for input if an input oriented operation such as `read()` or `receive()` is possible, or if a connection request may be accepted (see *Accepting Connections)* in section *4.1.1*.
- A descriptor selects for output if an output oriented operation such as `write()` or `send()` is possible, or if an operation that was "in progress", such as connection establishment, has completed (see section *3.8.3*.
- A descriptor selects for an exceptional condition if a condition that would cause a `SIGURG` signal to be generated exists (see section *3.2.1)* or other device-specific events have occurred.

If none of the specified conditions is true, the operation waits for one of the conditions to arise, blocking at most the amount of time specified by `tvp`. If `tvp` is given as 0, the `select()` waits indefinitely

Options affecting I/O on a descriptor may be read and set by the call:

```
dopt = fcntl(d, cmd, arg);
    result int dopt;
    int d, cmd, arg;

/* Interesting values for cmd */
#define F_DUPFD     0    /* Return new descriptor */
#define F_SETFD     1    /* Set close-on-exec flag */
#define F_GETFD     2    /* Set close-on-exec flag */
#define F_SETFL     3    /* Set descriptor options */
#define F_GETFL     4    /* Set descriptor options */
#define F_SETOWN    5    /* Set descriptor owner (pid/pgrp) */
#define F_GETOWN    6    /* Set descriptor owner (pid/pgrp) */
```

The `F_SETFL` *cmd* may be used to set a descriptor in non-blocking I/O mode and/or enable signaling when I/O is possible. `F_SETOWN` may be used to specify a process or process group to be signaled when using the latter mode of operation or or when urgent indications arise.

Operations on non-blocking descriptors will either complete immediately, note an error `EWOULDBLOCK`, partially complete an input or output operation returning a partial count, or return an error `EINPROGRESS` noting that the requested operation is in progress. A descriptor which has signaling enabled will cause the specified process and/or process group be signaled, with a `SIGIO` for input, output, or in-progress operation complete, or a `SIGURG` for exceptional conditions.

For example, when writing to a terminal using non-blocking output, the system will accept only as much data as there is buffer space for and return; when making a connection on a *socket*, the operation may return indicating that the connection establishment is "in progress". The `select()` facility can be used to determine when further output is possible on the terminal, or when the connection establishment attempt is complete.

3.5. Resource Controls

Process Priorities

The system gives CPU scheduling priority to processes that have not used CPU time recently. This tends to favor interactive processes and processes that execute only for short periods. It is possible to determine the priority currently assigned to a process, process group, or the processes of a specified user, or to alter this priority using the calls:

```
#define PRIO_PROCESS    0    /* process */
#define PRIO_PGRP       1    /* process group */
#define PRIO_USER       2    /* user id */

prio = getpriority(which, who);
    result int prio;
    int which, who;

setpriority(which, who, prio);
    int which, who, prio;
```

The value *prio* is in the range –20 to 20. The default priority is 0; lower priorities cause more favorable execution. The `getpriority()` call returns the highest priority (lowest numerical value) enjoyed by any of the specified processes. The `setpriority()` call sets the priorities of all of the specified processes to the specified value. Only the super-user may lower priorities.

Resource Utilization

`getrusage()` returned information about currently consumed resources in a structure defined in `<sys/resource.h>`:

```
#define RUSAGE_SELF 0        /* usage by this process */
#define RUSAGE_CHILDREN -1   /* usage by all children */

getrusage(who, rusage);
    int who;
    result struct rusage *rusage;

struct rusage {
    struct  timeval ru_utime;    /* user time used */
    struct  timeval ru_stime;    /* system time used */
    int ru_maxrss;  /* maximum core resident set size: kbytes */
    int ru_ixrss;   /* integral shared memory size (kbytes*sec) */
    int ru_idrss;   /* unshared data memory size */
    int ru_isrss;   /* unshared stack memory size */
    int ru_minflt;  /* page-reclaims */
    int ru_majflt;  /* page faults */
    int ru_nswap;   /* swaps */
    int ru_inblock; /* block input operations */
    int ru_oublock; /* block output operations */
    int ru_msgsnd;  /* messages sent */
    int ru_msgrcv;  /* messages received */
    int ru_nsignals;/* signals received */
    int ru_nvcsw;   /* voluntary context switches */
    int ru_nivcsw;  /* involuntary context switches */
};
```

The *who* parameter specifies whose resource usage is to be returned. The resources used by the current process, or by all the terminated children of the current process may be requested.

Resource Limits

The resources of a process for which limits are controlled by the kernel are defined in <sys/resource.h>, and controlled by the getrlimit() and setrlimit() calls:

```
#define RLIMIT_CPU       0     /* cpu time in milliseconds */
#define RLIMIT_FSIZE     1     /* maximum file size */
#define RLIMIT_DATA      2     /* maximum data segment size */
#define RLIMIT_STACK     3     /* maximum stack segment size */
#define RLIMIT_CORE      4     /* maximum core file size */
#define RLIMIT_RSS       5     /* maximum resident set size */

#define RLIM_NLIMITS     6

#define RLIM_INFINITY    0x7fffffff

struct rlimit {
    int rlim_cur;    /* current (soft) limit */
    int rlim_max;    /* hard limit */
};

getrlimit(resource, rlp);
    int resource;
    result struct rlimit *rlp;

setrlimit(resource, rlp);
    int resource;
    struct rlimit *rlp;
```

Only the super-user can raise the maximum limits. Other users may only alter *rlim_cur* within the range from 0 to *rlim_max* or (irreversibly) lower *rlim_max*.

3.6. System Operation Support

The call

```
swapon(blkdev, size);
    char *blkdev;
    int size;
```

specifies a device to be made available for paging and swapping. It can be run only by a privileged user.

The call

```
reboot(how);
    int how;
```

halts or reboots a machine. It too can be run only by a privileged user. The user may request a reboot by specifying *how* as RB_AUTOBOOT, or that the machine be halted with RB_HALT. These constants are defined in <sys/reboot.h>.

Accounting

The system optionally keeps an accounting record in a file for each process that exits on the system. The format of this record is beyond the scope of this document. The accounting may be enabled to a file *name* by doing

```
acct(path);
    char *path;
```

If *path* is null, then accounting is disabled. Otherwise, the named file becomes the accounting file.

3.7. Memory Management

For a synopses of the SunOS memory-management interface, see the *Memory Mapping Interface* section of the *Memory Management* chapter.

3.8. Generic I/O Operations

All filesystem descriptors support the operations `read()`, `write()` and `ioctl()`. We describe the basics of these common primitives here, as well as the `sync()` and `fsync()` primitives. Similarly, the mechanisms whereby normally synchronous operations may occur in a non-blocking or asynchronous fashion are common to all system-defined abstractions and are described here.

read() and write()

The `read()` and `write()` system calls can be applied to communications channels, files, terminals and devices. They have the form:

```
cc = read(fd, buf, nbytes);
    result int cc;
    int fd, nbytes;
    result caddr_t buf;

cc = write(fd, buf, nbytes);
    result int cc;
    int fd, nbytes;
    caddr_t buf;
```

The `read()` call transfers as much data as possible from the object defined by *fd* to the buffer at address *buf* of size *nbytes*. The number of bytes transferred is returned in *cc*, which is −1 if a return occurred before any data was transferred because of an error or use of non-blocking operations.

The `write()` call transfers data from the buffer to the object defined by *fd*. Depending on the type of *fd*, it is possible that the *write* call will accept some portion of the provided bytes; in this case the user should resubmit the other bytes in a later request. Error returns because of interrupted or otherwise incomplete operations are possible.

Scattering of data on input or gathering of data for output is also possible using an array of input/output vector descriptors. The type for the descriptors is defined in `<sys/uio.h>` as:

```
struct iovec {
    caddr_t iov_msg;        /* base of a component */
    int iov_len;            /* length of a component */
};
```

The calls using an array of descriptors are:

```
cc = readv(fd, iov, iovlen);
    result int cc;
    int fd, iovlen
    struct iovec *iov;
cc = writev(fd, iov, iovlen);
    result int cc;
    int fd, iovlen;
    struct iovec *iov;
```

Here *iovlen* is the count of elements in the *iov* array. It cannot exceed 16.

Input/Output Control

Control operations on an object are performed by the `ioctl()` operation:

```
ioctl(fd, request, buffer);
    int fd, request;
    caddr_t buffer;
```

This operation causes the specified *request* to be performed on the object *fd*. The *request* parameter specifies whether the argument buffer is to be read, written, read and written, or is not needed, and also the size of the buffer, as well as the request. Different descriptor types and subtypes within descriptor types may use distinct `ioctl()` requests. For example, operations on terminals control flushing of input and output queues and setting of terminal parameters; operations on disks cause formatting operations to occur; operations on tapes control tape positioning.

The names for basic control operations are defined in `<sys/ioctl.h>`.

Non-Blocking and Asynchronous Operations

A process that wishes to do non-blocking operations on one of its descriptors sets the descriptor in non-blocking mode as described in section *3.4.4*. Thereafter the `read()` call will return a specific `EWOULDBLOCK` error indication if there is no data to be `read()`. The process may `select()` the associated descriptor to determine when a read is possible.

Output attempted when a descriptor can accept less than is requested will either accept some of the provided data, returning a shorter than normal length, or return an error indicating that the operation would block. More output can be performed as soon as a `select()` call indicates the object is writable.

Operations other than data input or output may be performed on a descriptor in a non-blocking fashion. These operations will return with a characteristic error indicating that they are in progress if they cannot complete immediately. The descriptor may then be select()'ed for write() to find out when the operation has been completed. When select() indicates the descriptor is writable, the operation has completed. Depending on the nature of the descriptor and the operation, additional activity may be started or the new state may be tested.

File Caches

The call

```
fsync(fd);
    int fd;
```

moves all modified data and attributes of the file referenced by *fd* to a permanent storage device. When the fsync() call returns, all in-memory modified copies of buffers for the associated file have been written to disk. This call is different from sync().

The call

```
sync();
```

schedules input/output to clean all system buffer caches.

3.9. File System

The file system abstraction provides access to a hierarchical file system structure. The file system contains directories (each of which may contain other subdirectories) as well as files and references to other objects such as devices and inter-process communications sockets.

Each file is organized as a linear array of bytes. No record boundaries or system related information is present in a file. Files may be read and written in a random-access fashion. The user may read the data in a directory as though it were an ordinary file to determine the names of the contained files, but only the system may write into the directories. The file system stores only a small amount of ownership, protection and usage information with a file.

Naming

The file system calls take *pathname* arguments. These consist of a zero or more component *filenames* separated by / characters, where each filename is up to 255 ASCII characters excluding null and "/".

Each process always has two naming contexts: one for the root directory of the file system and one for the current working directory. These are used by the system in the filename translation process. If a pathname begins with a /, it is called a full pathname and interpreted relative to the root directory context. If the pathname does not begin with a / it is called a relative pathname and interpreted relative to the current directory context.

The system limits the total length of a pathname to 1024 characters.

The filename ".." in each directory refers to the parent directory of that directory. The parent directory of the root of the file system is always that directory.

The calls

```
chdir(path);
    char *path;

chroot(path);
    char *path;
```

change the current working directory and root directory context of a process. Only the super-user can change the root directory context of a process.

Creation and Removal

The file system allows directories, files and special devices, to be created and removed from the file system.

Directory Creation and Removal

A directory is created with the `mkdir()` system call:

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

where the mode is defined as for files (see below). Note that, in SunOS, `mkdir()` supports both the Berkeley and the SystemV group ID semantics. If the set-group-id bit on a directory is set, objects created within that directory are assigned the group ID of their parent directory, as in the BSD UNIX system. If the parent directory group ID bit is clear, objects created within it are assigned the group ID of the creating process, as in System V.

Directories are removed with the `rmdir()` system call:

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

A directory must be empty if it is to be deleted.

File Creation

Files are created with the `open()` system call,

```
fd = open(path, oflag, mode);
    result int fd;
    int oflag, mode;
    char *path;
```

The *path* parameter specifies the name of the file to be created. The *oflag* parameter must include `O_CREAT` from below to cause the file to be created. The protection for the new file is specified in *mode.* The protection for the new file is specified in *mode.* Bits for *oflag* are defined in `<sys/file.h>`:

```
#define O_RDONLY     000      /* open for reading */
#define O_WRONLY     001      /* open for writing */
#define O_RDWR       002      /* open for read & write */
#define O_NDELAY     004      /* non-blocking open */
#define O_APPEND     010      /* append on each write */
#define O_CREAT      01000    /* open with file create */
#define O_TRUNC      02000    /* open with truncation */
#define O_EXCL       04000    /* error on create if file exists */
```

One of O_RDONLY, O_WRONLY and O_RDWR should be specified, indicating what types of operations are desired to be performed on the open file. The operations will be checked against the user's access rights to the file before allowing the open() to succeed. Specifying O_APPEND causes writes to automatically append to the file. The flag O_CREAT causes the file to be created if it does not exist, owned by the current user and the group of the containing directory. The protection for the new file is specified in *mode*. The file mode is used as a three digit octal number. Each digit encodes read access as 4, write access as 2 and execute access as 1, or'ed together. The 700 bits describe owner access, the 070 bits describe the access rights for processes in the same group as the file, and the 007 bits describe the access rights for other processes.

If the open specifies to create the file with O_EXCL and the file already exists, then the open() will fail without affecting the file in any way. This provides a simple exclusive access facility. If the file exists but is a symbolic link, the open will fail regardless of the existence of the file specified by the link.

Creating References to Devices

The file system allows entries which reference peripheral devices. Peripherals are distinguished as *block* or *character* devices according by their ability to support block-oriented operations. Devices are identified by their 'major' and 'minor' device numbers. The major device number determines the kind of peripheral it is, while the minor device number indicates one of possibly many peripherals of that kind. Structured devices have all operations performed internally in 'block' quantities while unstructured devices often have a number of special ioctl() operations, and may have input and output performed in varying units. The mknod() call creates special entries:

```
mknod(path, mode, dev);
    char *path;
    int mode, dev;
```

where *mode* is formed from the object type and access permissions. The parameter *dev* is a configuration dependent parameter used to identify specific character or block I/O devices.

File and Device Removal

A reference to a file or special device may be removed with the unlink() call,

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

The caller must have write access to the directory in which the file is located for this call to be successful.

Reading and Modifying File Attributes

Detailed information about the attributes of a file system may be obtained with the calls:

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

statfs(path, buf);
    char *path;
    result struct statfs *buf;

fstatfs(fd, buf);
    int fd;
    result struct statfs *buf;
```

The statfs() structure includes the file system type, file system block size, total blocks in the file system, free blocks, free blocks available to non superuser, total file nodes in the file system, free file nodes in the file system, and the file system ID.

Directory entries can be obtained in a filesystem-independent format by using the getdents() call:

```
getdents(uap)
    register struct a {
        int fd;
        char *buf;
        unsigned count;
    } *uap;
```

Detailed information about the attributes of a file may be obtained with the calls:

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

stat(path, stb);
    char *path;
    result struct stat *stb;

fstat(fd, stb);
    int fd;
    result struct stat *stb;
```

The stat() structure includes the file type, protection, ownership, access times, size, and a count of hard links. If the file is a symbolic link, then the status of the link itself (rather than the file the link references) may be found using the lstat() call:

```
lstat(path, stb);
    char *path;
    result struct stat *stb;
```

Newly created files are assigned the user id of the process that created it and the group id of the directory in which it was created. The ownership of a file may be changed by either of the calls

```
chown(path, owner, group);
    char *path;
    int owner, group;
fchown(fd, owner, group);
    int fd, owner, group;
```

In addition to ownership, each file has three levels of access protection associated with it. These levels are owner relative, group relative, and global (all users and groups). Each level of access has separate indicators for read permission, write permission, and execute permission. The protection bits associated with a file may be set by either of the calls:

```
chmod(path, mode);
    char *path;
    int mode;
fchmod(fd, mode);
    int fd, mode;
```

where *mode* is a value indicating the new protection of the file as listed above in the *File Creation* section.

Three additional bits exist: the 04000 'set-user-id' bit can be set on an executable file to cause the effective user-id of a process which executes the file to be set to the owner of that file; the 02000 bit has a similar effect on the effective group-id. The 01000 bit causes an image of an executable program to be saved longer than would otherwise be normal; this 'sticky' bit is a hint to the system that a program is heavily used.

Finally, the access and modify times on a file may be set by the call:

```
utimes(path, tvp);
    char *path;
    struct timeval *tvp[2];
```

This is particularly useful when moving files between media, to preserve relationships between the times the file was modified.

Links and Renaming

Links allow multiple names for a file to exist. Links exist independently of the file linked to.

Two types of links exist, *hard* links and *symbolic* links. A hard link is a reference counting mechanism that allows a file to have multiple names within the same file system. Symbolic links cause string substitution during the pathname interpretation process.

Hard links and symbolic links have different properties. A hard link insures the target file will always be accessible, even after its original directory entry is removed; no such guarantee exists for a symbolic link. Symbolic links can span file systems boundaries.

The following calls create a new link, named *path2*, to *path1*:

```
link(path1, path2);
    char *path1, *path2;

symlink(path1, path2);
    char *path1, *path2;
```

The `unlink()` primitive may be used to remove either type of link.

If a file is a symbolic link, the 'value' of the link may be read with the `readlink()` call,

```
len = readlink(path, buf, bufsize);
    result int len;
    int bufsize;
    result char *path, *buf;
```

This call returns, in *buf*, the null-terminated string substituted into pathnames passing through *path*.

Atomic renaming of file system resident objects is possible with the `rename()` call:

```
rename(oldname, newname);
    char *oldname, *newname;
```

where both *oldname* and *newname* must be in the same file system. If *newname* exists and is a directory, then it must be empty.

Extension and Truncation

Files are created with zero length and may be extended simply by writing or appending to them. While a file is open the system maintains a pointer into the file indicating the current location in the file associated with the descriptor. This pointer may be moved about in the file in a random access fashion. To set the current offset into a file, the `lseek()` call may be used,

```
oldoffset = lseek(fd, offset, type);
    result off_t oldoffset;
    off_t offset;
    int fd, type;
```

where *type* is given in `<sys/file.h>` as one of,

```
#define L_SET    0    /* set absolute file offset */
#define L_INCR   1    /* set file offset relative to current position */
#define L_XTND   2    /* set offset relative to end-of-file */
```

The call

```
lseek(fd, 0, L_INCR)
```

returns the current offset into the file.

Files may have 'holes' in them. Holes are void areas in the linear extent of the file where data has never been written. These may be created by seeking to a location in a file past the current end-of-file and writing. Holes are treated by the system as zero valued bytes.

A file may be truncated (or extended) with either of the calls:

```
truncate(path, length);
    char *path;
    off_t length;

ftruncate(fd, length);
    int fd;
    off_t length;
```

The `truncate()` and `ftruncate()` system calls set the length of a file. If the newly specified length is shorter than the file's current length, the file is shortened. However, if the new length is longer, the file's size is increased to the desired length. When writing a file exclusively through mapped access, `truncate()` and `ftruncate()` are the only alternatives to `MAP_RENAME` operations for growing a file.

Checking Accessibility

A process running with different real and effective user ids may interrogate the accessibility of a file to the real user by using the `access()` call:

```
accessible = access(path, how);
    result int accessible;
    int how;
    char *path;
```

Here *how* is constructed by or'ing the following bits, defined in `<sys/file.h>`:

```
#define F_OK     0     /* file exists */
#define X_OK     1     /* file is executable */
#define W_OK     2     /* file is writable */
#define R_OK     4     /* file is readable */
```

The presence or absence of advisory locks does not affect the result of `access()`.

Locking

The file system provides basic facilities that allow cooperating processes to synchronize their access to shared files. A process may place an advisory `read()` or `write()` lock on a file, so that other cooperating processes may avoid interfering with the process' access. This simple mechanism provides locking with file granularity. More granular locking can be built using the IPC facilities to provide a lock manager. The system does not force processes to obey the locks; they are of an advisory nature only.

Locking is performed after an `open()` call by applying the `flock()` primitive,

```
flock(fd, how);
    int fd, how;
```

where the *how* parameter is formed from bits defined in `<sys/file.h>`:

```
#define LOCK_SH 1    /* shared lock */
#define LOCK_EX 2    /* exclusive lock */
#define LOCK_NB 4    /* don't block when locking */
#define LOCK_UN 8    /* unlock */
```

Successive lock calls may be used to increase or decrease the level of locking. If an object is currently locked by another process when a `flock()` call is made, the caller will be blocked until the current lock owner releases the lock; this may be avoided by including LOCK_NB in the *how* parameter. Specifying LOCK_UN removes all locks associated with the descriptor. Advisory locks held by a process are automatically deleted when the process terminates.

Mounting Filesystems

The call

```
mount(type, dir, flags, data);
    char *type, *dir;
    int flags;
    caddr_t data;
```

extends the UNIX name space. The `mount()` call specifies a block device *type* containing a UNIX file system to be made available starting at *dir*. If *flags* is set then the file system is read-only; writes to the file system will not be permitted and access times will not be updated when files are referenced. *data* is a pointer to a structure which contains the type specific arguments to mount.

The call

```
unmount(dir);
    char *dir;
```

unmounts the file system mounted on *dir*. umount() call will succeed only if the file system is not currently being used.

Disk Quotas

As an optional facility, each file system may be requested to impose limits on a user's disk usage. Two quantities are limited: the total amount of disk space which a user may allocate in a file system and the total number of files a user may create in a file system. Quotas are expressed as *hard* limits and *soft* limits. A hard limit is always imposed; if a user would exceed a hard limit, the operation which caused the resource request will fail. A soft limit results in the user receiving a warning message, but with allocation succeeding. Facilities are provided to turn soft limits into hard limits if a user has exceeded a soft limit for an unreasonable period of time.

To manipulate disk quotas on a file system the quotactl() call is used:

```
#include <ufs/quota.h>

quotactl(cmd, special, uid, addr);
    int cmd, uid;
    char *special;
    caddr_t addr;
```

where *cmd* indicates a command to be applied to the user ID *uid*. *Special* is a pointer to a null-terminated string containing the path name of the block special device for the file system being manipulated. The block special device must be mounted. *Addr* is the address of an optional, command specific, data structure which is copied in or out of the system. The interpretation of *addr* is given with each command.

3.10. Devices

The system uses a collection of device drivers to access attached peripherals. Such devices are generally grouped into two classes: structured devices on which block-oriented input/output operations occur (basically disks and tapes), and unstructured devices (anything else).

Structured Devices

Structured devices include disk and tape drives, and are accessed through a system buffer-caching mechanism, which permits them to be accessed as ordinary files, by means of random-access reads and writes.

The *mount* command in the system allows a structured device containing a file system volume to be accessed through the SunOS

Tape drives also typically provide a structured interface, although this is rarely used.

Unstructured Devices

Unstructured devices are those devices which do not support a randomly accessed block structure.

Communications lines, raster plotters, normal magnetic tape access (in large or variable size blocks), and access to disk drives permitting large block transfers and special operations like disk formatting and labeling all use unstructured device interfaces.

Much more information about device drivers can be found in *Writing Device Drivers.*

3.11. Debugging Support

ptrace() provides a means by which a process may control the execution of another process, and examine and change its memory image. Its primary use is for the implementation of breakpoint debugging.

```
#include <signal.h>
#include <sys/ptrace.h>
#include <sys/wait.h>

ptrace(request, pid, addr, data, addr2)
    enum ptracereq request;
    int pid, data;
    char *addr, *addr2;
```

There are five arguments whose interpretation depends on the *request* argument. Generally, *pid* is the process ID of the traced process. A process being traced behaves normally until it encounters some signal whether internally generated like 'illegal instruction' or externally generated like 'interrupt'. See sigvec(2) for the list. Then the traced process enters a stopped state and the tracing process is notified via wait(2). When the traced process is in the stopped state, its memory image can be examined and modified using ptrace(). If desired, another ptrace() request can then cause the traced process either to terminate or to continue, possibly ignoring the signal.

Note that several different values of the *request* argument can make ptrace() return data values — since −1 is a possibly legitimate value, to differentiate between −1 as a legitimate value and −1 as an error code, you should clear the *errno* global error code before doing a ptrace() call, and then check the value of *errno* afterwards.

The value of the *request* argument determines the precise action of the call:

PTRACE_TRACEME

This request is the only one used by the traced process; it declares that the process is to be traced by its parent. All the other arguments are ignored. Peculiar results will ensue if the parent does not expect to trace the child.

PTRACE_PEEKTEXT, PTRACE_PEEKDATA

The word in the traced process's address space at *addr* is returned. *addr* must be even (except on Sun386i machines), the child must be stopped and the input *data* and *addr2* are ignored.

PTRACÉ_PEEKUSER
The word of the system's per-process data area corresponding to *addr* is returned. *Addr* must be a valid offset within the kernel's per-process data structures. This space contains the registers and other information about the process; its layout corresponds to the *user* structure in the system.

PTRACE_POKETEXT, PTRACE_POKEDATA
The given *data* is written at the word in the process's address space corresponding to addr, which must be even (except on Sun386i machines). No useful value is returned. If the instruction and data spaces are separate request PTRACE_PEEKTEXT indicates instruction space while PTRACE_PEEKDATA indicates data space. The PTRACE_POKETEXT request must be used to write into a process's text space even if the instruction and data spaces are not separate.

PTRACE_POKEUSER
The process's system data is written, as it is read with request PTRACE_PEEKUSER. Only a few locations can be written in this way: the general registers, the floating point status and registers, and certain bits of the processor status word.

PTRACE_CONT
The *data* argument is taken as a signal number and the child's execution continues at location *addr* as if it had incurred that signal. Normally the signal number will be either 0 to indicate that the signal that caused the stop should be ignored, or that value fetched out of the process's image indicating which signal caused the stop. If *addr* is (int *)1 then execution continues from where it stopped.

PTRACE_KILL
The traced process terminates.

PTRACE_SINGLESTEP
Execution continues as in request PTRACE_CONT; however, as soon as possible after execution of at least one instruction, execution stops again. The signal number from the stop is SIGTRAP. On Sun machines the T-bit is used and just one instruction is executed.

PTRACE_ATTACH
Attach to the process identified by the *pid* argument and begin tracing it. Process *pid* does not have to be a child of the requester, but the requester must have permission to send process *pid* a signal and the effective userids of the requesting process and process *pid* must match.

PTRACE_DETACH
Detach the process being traced. Process *pid* is no longer being traced and continues its execution. The *data* argument is taken as a signal number and the process continues at location *addr* as if it had incurred that signal.

PTRACE_GETREGS
The traced process's registers are returned in a structure pointed to by the *addr* argument. The registers include the general purpose registers, the program counter and the program status word. The 'regs' structure defined in

<machine/reg.h> describes the data that is returned.

PTRACE_SETREGS

The traced process's registers are written from a structure pointed to by the *addr* argument. The registers include the general purpose registers, the program counter and the program status word. The 'regs' structure defined in <machine/reg.h> describes the data that is set.

PTRACE_READTEXT, PTRACE_READDATA

Read data from the address space of the traced process. If the instruction and data spaces are separate, request PTRACE_READTEXT indicates instruction space while PTRACE_READDATA indicates data space. The *addr* argument is the address within the traced process from where the data is read, the *data* argument is the number of bytes to read, and the *addr2* argument is the address within the requesting process where the data is written.

PTRACE_WRITETEXT, PTRACE_WRITEDATA

Write data into the address space of the traced process. If the instruction and data spaces are separate, request PTRACE_READTEXT indicates instruction space while PTRACE_READDATA indicates data space. The *addr* argument is the address within the traced process where the data is written, the *data* argument is the number of bytes to write, and the *addr2* argument is the address within the requesting process from where the data is read.

As indicated, these calls (except for requests PTRACE_TRACEME and PTRACE_ATTACH) can be used only when the subject process has stopped. The wait() call is used to determine when a process stops; in such a case the 'termination' status returned by *wait* has the value WSTOPPED to indicate a stop rather than genuine termination.

To forestall possible fraud, ptrace() inhibits the set-user-id and set-group-id facilities on subsequent execve(2) calls. If a traced process calls execve(), it will stop before executing the first instruction of the new image showing signal SIGTRAP.

4

SunOS Networking

4

SunOS Networking

4.1. Socket-Based Interprocess Communications

Interprocess Communication Primitives

This chapter introduces the socket-based interprocess communications facilities which SunOS has adapted from BSD. Much more detail about these facilities can be found in part three of *Network Programming*. For an introduction to the networking facilities which Sun has added to its system in the time since socket-based IPC was developed, see the *Network Services* section of this same *Network Programming* manual. (These facilities include the *Network File System*, the *Remote Procedure Call* mechanisms, and the *External Data Representation* standard). For detailed information about AT&T-style STREAMS, see *Writing Device Drivers.*

Communication Domains

The system provides access to an extensible set of communication *domains*. A communication domain is identified by a manifest constant defined in the file `<sys/socket.h>`. Important standard domains supported by the system are the UNIX domain, `AF_UNIX`, for communication within the system, and the "internet" domain for communication in the DARPA internet, `AF_INET`. Other domains can be added to the system.

Socket Types and Protocols

Within a domain, communication takes place between endpoints known as *sockets*. Each socket has the potential to exchange information with other sockets of an appropriate type within the domain.

Each socket has an associated abstract type, which describes the semantics of communication using that socket. Properties such as reliability, ordering, and prevention of duplication of messages are determined by the type. The basic set of socket types is defined in `<sys/socket.h>`:

```
/* Standard socket types */
#define SOCK_DGRAM       1    /* datagram */
#define SOCK_STREAM      2    /* virtual circuit */
#define SOCK_RAW         3    /* raw socket */
#define SOCK_RDM         4    /* reliably-delivered message */
#define SOCK_SEQPACKET   5    /* sequenced packets */
```

The SOCK_DGRAM type models the semantics of datagrams in network communication: messages may be lost or duplicated and may arrive out-of-order. A datagram socket may send messages to and receive messages from multiple peers. The SOCK_RDM type models the semantics of reliable datagrams: messages arrive unduplicated and in-order, the sender is notified if messages are lost. The send() and receive() operations (described below) generate reliable/unreliable datagrams. The SOCK_STREAM type models connection-based virtual circuits: two-way byte streams with no record boundaries. Connection setup is required before data communication may begin. The SOCK_SEQPACKET type models a connection-based, full-duplex, reliable, sequenced packet exchange; the sender is notified if messages are lost, and messages are never duplicated or presented out-of-order. Users of the last two abstractions may use the facilities for out-of-band transmission to send out-of-band data.

SOCK_RAW is used for unprocessed access to internal network layers and interfaces; it has no specific semantics.

Other socket types can be defined.

Each socket may have a concrete *protocol* associated with it. This protocol is used within the domain to provide the semantics required by the socket type. Not all socket types are supported by each domain; support depends on the existence and the implementation of a suitable protocol within the domain. For example, within the "internet" domain, the SOCK_DGRAM type may be implemented by the UDP user datagram protocol, and the SOCK_STREAM type may be implemented by the TCP transmission control protocol, while no standard protocols to provide SOCK_RDM or SOCK_SEQPACKET sockets exist.

Socket Creation, Naming, and Service Establishment

Sockets may be *connected* or *unconnected*. An unconnected socket descriptor is obtained by the socket() call:

```
s = socket(domain, type, protocol);
    result int s;
    int domain, type, protocol;
```

The socket domain and type are as described above, and are specified using the definitions from <sys/socket.h>. The protocol may be given as 0, meaning any suitable protocol. One of several possible protocols may be selected using identifiers obtained from a library routine, getprotobyname().

An unconnected socket descriptor of a connection-oriented type may yield a connected socket descriptor in one of two ways: either by actively connecting to another socket, or by becoming associated with a name in the communications domain and *accepting* a connection from another socket. Datagram sockets need not establish connections before use.

To accept connections or to receive datagrams, a socket must first have a binding to a name (or address) within the communications domain. Such a binding may be established by a bind() call:

```
bind(s, name, namelen);
    int s, namelen;
    struct sockaddr *name;
```

Datagram sockets may have default bindings established when first sending data if not explicitly bound earlier. In either case, a socket's bound name may be retrieved with a getsockname() call:

```
getsockname(s, name, namelen);
    int s;
    result struct sockaddr *name;
    result int *namelen;
```

while the peer's name can be retrieved with getpeername():

```
getpeername(s, name, namelen);
    int s;
    result struct sockaddr *name;
    result int *namelen;
```

Domains may support sockets with several names.

Accepting Connections

Once a binding is made to a connection-oriented socket, it is possible to listen() for connections:

```
listen(s, backlog);
    int s, backlog;
```

The *backlog* specifies the maximum count of connections that can be simultaneously queued awaiting acceptance.

An accept() call:

```
t = accept(s, name, anamelen);
    result int t, *anamelen;
    int s;
    result struct sockaddr *name;
```

returns a descriptor for a new, connected, socket from the queue of pending connections on *s*. If no new connections are queued for acceptance, the call will wait for a connection unless non-blocking I/O has been enabled.

Making Connections

An active connection to a named socket is made by the connect() call:

```
connect(s, name, namelen);
    int s, namelen;
    struct sockaddr *name;
```

Although datagram sockets do not establish connections, the `connect()` call may be used with such sockets to create an *association* with the foreign address. The address is recorded for use in future `send()` calls, which then need not supply destination addresses. Datagrams will be received only from that peer, and asynchronous error reports may be received.

It is also possible to create connected pairs of sockets without using the domain's name space to rendezvous; this is done with the `socketpair()` call[1]:

```
socketpair(domain, type, protocol, sv);
    int domain, type, protocol;
    result int sv[2];
```

Here the returned *sv* descriptors correspond to those obtained with `accept()` and `connect()`.

The call

```
pipe(pv);
    result int pv[2];
```

creates a pair of `SOCK_STREAM` sockets in the UNIX domain, with `pv[0]` only writable and `pv[1]` only readable.

Sending and Receiving Data

Messages may be sent from a socket by:

```
cc = sendto(s, buf, len, flags, to, tolen);
    result int cc;
    int s, len, flags, tolen;
    caddr_t buf, to;
```

if the socket is not connected or:

```
cc = send(s, buf, len, flags);
    result int cc;
    int s, len, flags;
    caddr_t buf;
```

if the socket is connected. The corresponding receive primitives are:

```
msglen = recvfrom(s, buf, len, flags, from, fromlenaddr);
    result int *fromlenaddr;
    result int msglen;
    int s, len, flags;
    result caddr_t buf, from;
```

[1] This release supports `socketpair()` creation only in the "unix" communication domain.

and

```
msglen = recv(s, buf, len, flags);
    result int msglen;
    int s, len, flags;
    result caddr_t buf;
```

In the unconnected case, the parameters *to* and *tolen* specify the destination or source of the message, while the *from* parameter stores the source of the message, and **fromlenaddr* initially gives the size of the *from* buffer and is updated to reflect the true length of the *from* address.

All calls cause the message to be received in or sent from the message buffer of length *len* bytes, starting at address *buf*. The *flags* specify peeking at a message without reading it or sending or receiving high-priority out-of-band messages, as follows:

```
#define MSG_PEEK     0x1 /* peek at incoming message */
#define MSG_OOB 0x2 /* process out-of-band data */
```

Scatter/Gather and Exchanging Access Rights

It is possible to scatter and gather data and to exchange access rights with messages. When either of these operations is involved, the number of parameters to the call becomes large. Thus the system defines a message header structure, in `<sys/socket.h>`, which is used to contain the parameters to the calls:

```
struct msghdr {
    caddr_t msg_name;          /* optional address */
    int msg_namelen;           /* size of address */
    struct  iov *msg_iov;      /* scatter/gather array */
    int msg_iovlen;            /* # elements in msg_iov */
    caddr_t msg_accrights;     /* access rights sent/received */
    int msg_accrightslen;      /* size of msg_accrights */
};
```

Here *msg_name* and *msg_namelen* specify the source or destination address if the socket is unconnected; *msg_name* may be given as a null pointer if no names are desired or required. The *msg_iov* and *msg_iovlen* describe the scatter/gather locations, as described in section *3.8.1*. Access rights to be sent along with the message are specified in *msg_accrights*, which has length *msg_accrightslen*. In the "unix" domain these are an array of integer descriptors, taken from the sending process and duplicated in the receiver.

This structure is used in the operations `sendmsg()` and `recvmsg()`:

```
sendmsg(s, msg, flags);
    int s, flags;
    struct msghdr *msg;

msglen = recvmsg(s, msg, flags);
    result int msglen;
    int s, flags;
    result struct msghdr *msg;
```

Using **read()** and **write()** with Sockets

The normal `read()` and `write()` calls may be applied to connected sockets and translated into `send()` and `receive()` calls from or to a single area of memory and discarding any rights received. A process may operate on a virtual circuit socket, a terminal or a file with blocking or non-blocking input/output operations without distinguishing the descriptor type.

Shutting Down Halves of Full-Duplex Connections

A process that has a full-duplex socket such as a virtual circuit and no longer wishes to read from or write to this socket can give the call:

```
shutdown(s, direction);
    int s, direction;
```

where *direction* is 0 to not read further, 1 to not write further, or 2 to completely shut the connection down. If the underlying protocol supports unidirectional or bidirectional shutdown, this indication will be passed to the peer. For example, a shutdown for writing might produce an end-of-file condition at the remote end.

Socket and Protocol Options

Sockets, and their underlying communication protocols, may support *options*. These options may be used to manipulate implementation specific or protocol-specific facilities. The `getsockopt()` and `setsockopt()` calls are used to control options:

```
getsockopt(s, level, optname, optval, optlen);
    int s, level, optname;
    result caddr_t optval;
    result int *optlen;

setsockopt(s, level, optname, optval, optlen);
int s, level, optname; caddr_t optval; int optlen;
```

The option *optname* is interpreted at the indicated protocol *level* for socket *s*. If a value is specified with *optval* and *optlen*, it is interpreted by the software operating at the specified *level*. The *level* `SOL_SOCKET` is reserved to indicate options maintained by the socket facilities. Other *level* values indicate a particular protocol which is to act on the option request; these values are normally interpreted as a "protocol number".

UNIX Domain

This section describes briefly the properties of the UNIX communications domain.

Types of Sockets

In the UNIX domain, the `SOCK_STREAM` abstraction provides pipe-like facilities, while `SOCK_DGRAM` provides datagrams — unreliable message-style communications.

Naming

Socket names are strings and the current implementation of the UNIX domain embeds bound sockets in the file system name space; this is a side effect of the implementation.

Access Rights Transmission

The ability to pass UNIX descriptors with messages in this domain allows migration of service within the system and allows user processes to be used in building system facilities.

Internet Domain

This section describes briefly how the Internet domain is mapped to the model described in this section. More information will be found in the *Networking Implementation Notes* section of *Network Programming*.

Socket Types and Protocols

`SOCK_STREAM` is supported by the Internet TCP protocol; `SOCK_DGRAM` by the UDP protocol. Each is layered atop the transport-level Internet Protocol (IP). The Internet Control Message Protocol is implemented atop/beside IP and is accessible via a raw socket.

Socket Naming

Sockets in the Internet domain have names composed of the 32 bit internet address, and a 16 bit port number. Options may be used to provide IP source routing or security options. The 32-bit address is composed of network and host parts; the network part is variable in size and is frequency encoded. The host part may optionally be interpreted as a subnet field plus the host on subnet; this is is enabled by setting a network address mask at boot time.

Access Rights Transmission

No access rights transmission facilities are provided in the Internet domain.

Raw Access

The Internet domain allows the super-user access to the raw facilities of IP. These interfaces are modeled as `SOCK_RAW` sockets. Each raw socket is associated with one IP protocol number, and receives all traffic received for that protocol. This allows administrative and debugging functions to occur, and enables user-level implementations of special-purpose protocols such as inter-gateway routing protocols.

4.2. Network-Based Services

SunOS is considerably more sophisticated than the first versions of the UNIX system. This is true not only in terms of programming environments and tools, though SunOS does include most 4.3BSD enhancements and virtually all AT&T System V.III facilities. SunOS is oriented, at a fundamental level, to networks of closely linked machines. It is *structurally* a network system, and is designed to evolve with the evolution of computer network technology.

SunOS began as 4.2BSD, developed at the University of California at Berkeley from an early version of UNIX. One of the major enhancements made at

Berkeley was the addition of support for network communication. In particular, network services were implemented with special-purpose daemons (server processes) working in close cooperation with the kernel, rather than in the kernel itself. SunOS has continued this line of development. Its expanding domain of network services — from the Network Filing System (NFS) and Network extensible Window System (NeWS) to its Remote Execution Facility (REX) and network naming service (YP) — is uniformly built upon a server-based architecture.

When a network service is added to SunOS, it is added by means of a server process which is executed on all machines providing the service. Each server then communicates with the SunOS kernel and with its peers on other machines as necessary. Sun servers do differ in one very significant way from those which were inherited from Berkeley — they are usually based on Sun's Remote Procedure Call (RPC) mechanism. As a consequence, they automatically benefit from the services provided by RPC and the External Data Representation (XDR), such as the data portability provided by XDR and the modularity of RPC's authentication system.

There are a number of benefits to a server-based approach to the provision of network services:

- The kernel itself remains more manageable in size and complexity, and more clearly delimited in function. Its job is to implement the SunOS virtual machine on the machine that hosts it. It does not negotiate with other machines for the non-local resources that it needs.
- When network services are implemented as independent server processes, they are easily tuned and controlled.
- They can be invoked only when needed (see `inetd`(8)) and thus consume no run-time resources when not in use. And they are easily updated to accommodate protocol and transport changes. Indeed, when such changes are made, multiple versions of the same server can be run simultaneously, thus allowing development to proceed without rendering old applications obsolete.

The overall effect is thus an *extensible* environment in which new network services can be easily added to the system by building upon XDR, RPC, network communications and other services. Network services, then, are analogous to SunOS commands — anyone can add one, and when they do they are effectively extending the "system".

NOTE *See the Network Services section of Network Programming for more information about the fundamental network services.*

4.3. Standard SunOS Server-Based Services

Networking functions contained within the SunOS kernel include the network and transport levels of the system networking support, the network device drivers, the IP and TCP protocol code and the NFS itself. Other network services are provided by server processes:

`/usr/etc/biod`

Block I/O daemon. Used by a NFS client to handle read-ahead and write-behind for blocks in the buffer cache.

`/usr/etc/bootparams`

NFS boot daemon. Provides the information that diskless clients need for booting. If the Yellow Pages aren't available, it consults either the `bootparams` database or `/etc/bootparams`.

`/usr/etc/in.comsat`

Listens to a non-standard UDP socket used for incoming mail notification, as enabled by the `biff` program.

`/usr/etc/rpc.etherd`

`etherd` collects, summarizes and reports statistics on packet traffic for a given network interface.

`/usr/etc/in.fingerd`

`in.fingerd` provides support for the ARPA-standard `finger` command, which displays information about the current users of a given machine.

`/usr/etc/in.ftpd`

File Transfer Protocol daemon. This is the ARPA standard file transfer protocol, rarely used on Suns.

`/usr/etc/inetd`

Opens sockets for all the servers listed in `/etc/inetd.conf`, and then starts them up when requests are made on them.

`/usr/etc/keyserv`

The DES authentication daemon. Generates and stores secret keys and controls access to them. Does the public-key encryption and decryption operations. `keyserv` will not talk to anything but a local root process.

`/usr/etc/rpc.lockd`

The network lock manager daemon. Provides System-V (SVID) compatible advisory file and record locking for both local and NFS mounted files.

`/usr/etc/rpc.mountd`

NSF mount daemon. Handles mount requests for files systems exported over the NFS.

`/usr/etc/in.named`

`named` is the Internet domain name server.

`/usr/etc/nfsd`

Network File System daemon. The real work is done in the kernel by way of a magic system call that never returns.

`/usr/etc/portmap`

Demultiplexes UDPs for Remote Procedure Calls, converting RPC program numbers to UDP port numbers.

`/usr/etc/rarpd`

`rarpd` is a daemon that responds to reverse-arp requests.

`/usr/etc/rpc.rexd`

`rexd` is the Sun RPC server that controls remote program execution.

/usr/etc/in.rexecd

`rexecd` is the server for the `rexec()` routine. It provides remote execution facilities with authentication based on user names and encrypted passwords.

/usr/etc/in.rlogind

Remote Login daemon.

/usr/etc/rmt

Remote magnetic tape access. Used by the remote dump and restore programs to manipulate a tape driver over the network.

/usr/etc/in.routed

Routing table update daemon. Uses a non-standard UDP protocol to update kernel routing tables.

/usr/etc/rpc.rquotad

`rquotad` returns quotas for a user of a local file system which is mounted by a remote machine over the NFS. The results are used by `quota` to display remote file systems user quotas.

/usr/etc/in.rshd

Remote shell daemon. Non-standard TCP protocol to allow remote execution with authentication based on privileged port numbers.

/usr/etc/rpc.rusersd

Remote user daemon. Necessary to support the `rusers` command.

/usr/etc/rpc.rwalld

Remote write-to-all daemon. Handles `rwall` and `shutdown` requests.

/usr/etc/in.rwhod

Remote `who` daemon. Generates broadcasts periodically about the status of logged-in users, and listens to the broadcasts of other servers on the local network and maintains the database that is printed by `rwho`. Not used much in the Sun environment since the protocol involves lots of broadcast packets.

/usr/lib/sendmail

Provides mail transport through the Simple Mail Transfer Protocol (SMTP).

/usr/etc/rpc.sprayd

Spray daemon. Used by the `spray` command for network diagnosis.

/usr/etc/rpc.statd

Remote status daemon. The primary purposes for this server are returning kernel performance statistics for `perfmeter`, and responding to requests from `rup`.

/usr/etc/in.syslog

Reads a datagram (UDP) socket and logs information it receives according to a configuration file.

/usr/etc/in.talkd

Listens on a UDP port, and negotiates talk TCP connections. This protocol doesn't even work between Vaxes and Suns.

/usr/etc/in.telnetd
The ARPA-standard remote terminal service.

/usr/etc/in.tftpd
Trivial file transfer protocol daemon. Can be used for simple, non-authenticated file transfers. Also used to load boot files.

/usr/etc/in.timed
The ARPA-standard time service. Note that this service only provides the system time to clients who request it, and is not a full network synchronization service.

/usr/etc/in.tnamed
The `tnamed` daemon supports the DARPA Name Server Protocol.

/usr/etc/ypbind
`ypbind` remembers information that lets client processes on a single node communicate with some `ypserv` process. It must run on every machine which has YP client processes.

/usr/etc/rpc.yppasswdd
Runs on YP masters only. Supports password change requests for the Yellow Pages password database.

/usr/etc/ypserv
Runs on all YP servers. The `ypserv` daemon's primary function is to look up information in the local Yellow Pages database.

/usr/etc/rpc.ipallocd
(Sun386i only). The `rpc.ipallocd` daemon maps Ethernet addresses to IP addresses, allocating temporary IP addresses when necessary.

/usr/etc/rpc.pnpd
(Sun386i only). The `rpc.pnpd` daemon configures new systems onto a Sun386i network, and distributes configuration information for systems already on the network. It also provides configuration RPCs for diskless clients.

5

Memory Management

5

Memory Management

5.1. The SunOS Virtual Memory System

SunOS includes a complete set of memory-mapping mechanisms, which it uses as the basis of system services like shared libraries and System V compatible shared memory. Process address spaces are composed of a vector of memory pages, each of which can be independently mapped and manipulated. Typically, the system presents the user with mappings that simulate the traditional UNIX process memory environment, but other views of memory are useful as well.

The SunOS memory-management facilities:

- Unify the system's operations on memory.
- Provide a set of kernel mechanisms powerful and general enough to support the implementation of fundamental system services without special-purpose kernel support.
- Maintain consistency with the existing environment, in particular using the UNIX file system as the name space for named virtual-memory objects.

Virtual Memory, Address Spaces and Mapping

The system's *virtual memory* consists of all available physical memory resources. (Examples include local and remote file systems, processor primary memory, swap space and other random-access devices). Named objects in the virtual memory are referenced though the UNIX file system. This does not imply that all file system objects are in the virtual memory, but simply that all named objects in the virtual memory are named in the file system. Some virtual memory objects, such as private process memory and System V shared memory segments, do not have names.

A process's *address space* is defined by mappings onto system virtual-memory objects (usually files). Each mapping is constrained to be sized and aligned with the page boundaries of the system on which the process is executing. Each page may be mapped (or not) independently. Only process addresses which are mapped to some system object are valid, for there is no memory associated with processes themselves—all memory is represented by virtual memory objects.

Each object in the virtual memory has an *object address space* defined by some physical storage. A reference to an object address accesses the physical storage that implements the address within the object. The virtual memory's associated physical storage is thus accessed by transforming process addresses to object addresses, and then to the physical store.

A given process page may map to only one object, although a given object address may be the subject of many process mappings. An important characteristic of a mapping is that the object to which the mapping is made is not affected by the mere *existence* of the mapping. Thus, it cannot, in general, be expected than an object has an "awareness" of having been mapped, or of which portions of its address space are accessed by mappings; in particular, the notion of a "page" is not a property of the object. Establishing a mapping to an object simply provides the *potential* for a process to access or change the object's contents.

The establishment of mappings provides an *access method* that renders an object directly addressable by a process. Applications may find it advantageous to 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.

Networking, Heterogeneity and Coherence

Sun's VM system is designed to fit well with the larger Sun heterogeneous environment. This environment makes extensive use of networking to access file systems—file systems that are now part of the system's virtual memory. Sun's NFS-based networks are not constrained to consist of similar hardware or to be based upon a common operating system; in fact, the opposite is encouraged, for such constraints create serious barriers to accommodating heterogeneity. While a given set of processes may *apply* a set of mechanisms to establish and maintain the properties of various system objects— properties such as page sizes and the ability of objects to synchronize their own use—a given operating system should not *impose* such mechanisms on the rest of the network.

As it stands, the access method view of a virtual memory maintains the potential for a given object (say a text file) to be mapped by systems running Sun's memory management system and also to be accessed by systems for which virtual memory and storage management techniques such as paging are totally foreign, such as PC-DOS. Such systems can continue to share access to the object, each using and providing its programs with the access method appropriate to that system. The unacceptable alternative would be to prohibit access to the object by less capable systems.

Another consideration arises when applications use an object as a communications channel, or otherwise attempt to access it simultaneously. In both of these cases, the object is being shared, and thus the applications must use some synchronization mechanism to guarantee the coherence of their transactions with it. The scope and nature of the synchronization mechanism is best left to the application to decide. For example, file access on systems which do not support virtual memory access methods must be indirect, by way of `read()` and `write()`. Applications sharing files on such systems must coordinate their access using semaphores, file locking or some application-specific protocols. What is required in an environment where mapping replaces *read* and *write* as the access method is that an operation comparable to *fsync* be provided to support atomic update operations.

The nature and scope of synchronization over shared objects is application-defined from the outset. If the system attempted to impose any automatic semantics for sharing, it might prohibit other useful forms of mapped access that have nothing whatsoever to do with communication or sharing. By providing the mechanism to support coherency, and leaving it to cooperating applications to apply the mechanism, the needs of applications are met without erecting barriers to heterogeneity. Note that this design does not prohibit the creation of libraries that provide coherent abstractions for common application needs. Not all abstractions on which an application builds need be supplied by the "operating system".

5.2. System Facilities

Memory Mapping Interface

The applications programmer gains access to the facilities of the VM system through several sets of system calls. This section briefly summarizes these calls. For details, see the *SunOS Reference Manual.*

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

`mmap()` establishes a mapping between the process's address space at an address *addr* for *len* bytes to the object specified by *fd* at offset *off* for *len* bytes. (The value of *paddr* is an implementation-dependent function of the parameter *addr* and the value of *flags*, further described below). A successful call to `mmap()` returns *paddr* as its result. The address ranges covered by [*paddr, paddr* + *len*) and [*off, off* + *len*) must be legitimate for the address space of a process and the object in question, respectively. The mapping established by `mmap()` replaces any previous mappings for the process's pages in the range [*paddr*, *paddr* + *len*).

The parameter *prot* determines whether *read*, *execute*, *write* or some combination of accesses are permitted to the pages being mapped. The values desired are expressed by or'ing the flags values `PROT_READ`, `PROT_EXECUTE`, and `PROT_WRITE`. A *write* access must fail if `PROT_WRITE` has not been set, though its behavior can be influenced by setting `MAP_PRIVATE` in the *flags* parameter.

The *flags* parameter provides other information about the handling of mapped pages.

`MAP_SHARED` and `MAP_PRIVATE` specify the mapping type, and one of them must be specified. If `MAP_SHARED` is specified, write references will change the mapped pages; if `MAP_PRIVATE` is specified, an initial write reference will create a private copy of the mapped pages and redirect the mapping to the copy. The mapping type is retained across a `fork()`. The mapping type only affects the disposition of stores by the calling process – there is no isolation from changes made by other processes. If an application desires such isolation, it should use `read()` to make a copy of the data it wishes to keep protected.

`MAP_FIXED` informs the system that the value of *paddr* must be *addr*, exactly. The use of `MAP_FIXED` is discouraged, as it may prevent an implementation from making the most effective use of system resources. When `MAP_FIXED` is not set, the system uses *addr* as a hint to arrive at *paddr*. The *paddr* so chosen will be an area of the address space that the system deems suitable for a mapping of *len* bytes to the specified object. An *addr* value of zero grants the system complete freedom in selecting *paddr*, subject to constraints described below. A non-zero value of *addr* is taken as a suggestion of a process address near which the mapping should be placed. When the system selects a value for *paddr*, it will never place a mapping at address 0, nor will it replace any extant mapping, nor map into areas considered part of the potential data or stack "segments". The system strives to choose alignments for mappings that maximize the performance of the its hardware resources.

```
msync(addr, len, flags)
    caddr_t addr;
    int len, flags;
```

`msync()` supports applications which require coherency. It causes all modified copies of pages over the range [*addr, addr* + *len*) to be flushed to the objects mapped by those addresses. `msync()` optionally invalidates such cache entries so that further references to the pages will cause the system to obtain them from their permanent storage locations. The *flags* argument provides a bit-field of values which influences `msync()`'s behavior. The bit names and their interpretations are:

MS_ASYNC	Return immediately
MS_INVALIDATE	Invalidate caches

`MS_ASYNC` causes `msync()` to return immediately once all I/O operations are scheduled; normally, `msync()` will not return until all I/O operations are complete. `MS_INVALIDATE` causes all cached copies of data from mapped objects to be invalidated, requiring them to be re-obtained from the object's storage upon the next reference.

```
mprotect(addr, len, prot)
    caddr_t addr;
    int len, prot;
```

`mprotect()` has the effect of assigning protection *prot* to all pages in the range [*addr, addr* + *len*).

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

munmap() has the effect of removing all pages in the range [*addr*, *addr* + *len*) from the address space of the calling process.

```
pagesize = getpagesize();
```

getpagesize() returns the system-dependent size of a memory page.

```
mincore(addr, len, vec)
    caddr_t addr;
    int len;
    char *prot;
```

mincore() determines the residency of the memory pages in the address space covered by mappings in the range [*addr*, *addr* + *len*). The status is returned as a char-per-page in the character array referenced by **vec* (which the system assumes to be large enough to encompass all the pages in the address range)

System V Shared Memory

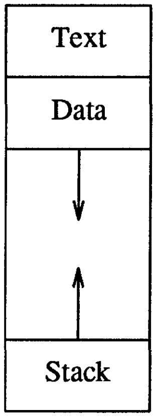
The AT&T *System V Interface Definition (SVID)* defines a number of operations on "shared memory segments". These operations are all supported in SunOS exactly as defined in the SVID.

5.3. Address Space Layout

"Segments"

Traditionally, the address space of a UNIX process has consisted of three segments: one each for write-protected program code (text), a heap of dynamically allocated storage (data), and the process's stack. Text and data segments grow from higher to lower address spaces, while the stack grows from lower to higher addresses. This can be illustrated as follows:

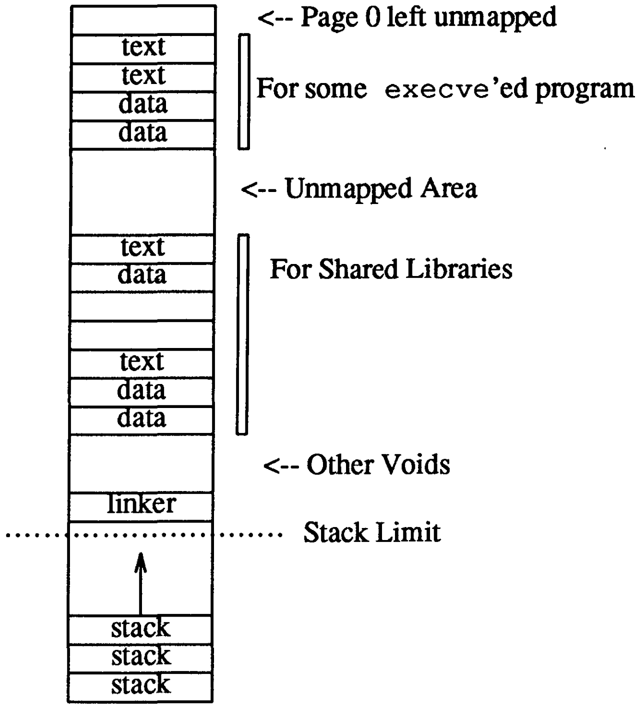
Figure 5-1 *Traditional UNIX Address-space Layout*



Under SunOS, a process's address space is simply a vector of pages and there exists no clear division between different address-space segments. Process text and data spaces are simply groups of pages.[2] There are often multiple text and data "segments", some belonging to specific programs and some belonging to code running in shared libraries. An illustration of one possible layout of an address space is:

[2] For compatibility purposes, the system maintains address ranges that "should" belong to such segments to support operations such as extending or contracting the data segment's "break". These are initialized when a program is initiated with `execve()`.

Figure 5-2 *SunOS Address-space Layout*



Process address spaces are often constructed through dynamic linking when a program is `execve`'ed. As can be seen from the above picture, the system address space is sparsely populated, with data and text pages intermingled. Stack space remains in high address space, and grows down to the stack limit as stack pages are allocated. By convention, page 0 is not used. Link editing can produce errors; for details see the *Shared Libraries* chapter of the *Programming Utilities and Libraries*.

Text, Stack and Data

In SunOS, as in traditional UNIX systems, process memory can be viewed as composed of three logical "segments"— text, data and stack. These segments behave as one would expect, the text segment is read-only and shared, while the data and stack segments are private to the process. The stack expands as necessary to accommodate the process's stack usage.

The process can manipulate its own data and stack segments by calling `brk()` and `sbrk()`:

```
caddr_t brk(addr)
     caddr_t addr;

caddr_t = sbrk(incr);
    int incr;
```

`brk()` sets the system's idea of the lowest data segment location not used by the caller to *addr* (rounded up to the next multiple of the system's page size).

`sbrk()`, the alternate function, adds *incr* bytes to the caller's data space and returns a pointer to the start of the new data area.

6

Lightweight Processes

6

Lightweight Processes

6.1. Introduction

This tutorial provides some examples of how to use the lightweight process library. Although the term "lightweight processes" is often used, it is really a misnomer since the fundamental property of lightweight processes is not that they are somehow "lighter" than ordinary processes, but that a lightweight process represents a thread of control not bound to an address space. If threads appear to operate more efficiently than ordinary SunOS processes, it is because threads communicate via shared memory instead of a filesystem. Because threads can share a common address space, the cost of creating tasks and intertask communication is substantially less than the cost of using more "heavyweight" primitives. The availability of lightweight processes provides an abstraction well-suited to writing programs which react to asynchronous events such as servers. In addition, lightweight processes are useful for simulation programs which model concurrent situations.

Definition

The idea is to provide a process abstraction: a thread is a data type representing a flow of control. A number of operations are available to manipulate threads, including ways to control their scheduling and communication. Lightweight processes exist independently of virtual memory, I/O, resource allocation, and other operating system-supported objects, but are able to smoothly work with these objects.

The lightweight process abstraction for managing asynchrony is superior to the UNIX signal abstraction. Under the UNIX system, a signal causes a sort of context switch (to a new instruction and optionally, to a new location on the stack) but the thread is the same: for example, you can `longjmp()` to the main program (the signal handler and main program can't run in parallel). Critical sections are implemented by disabling interrupts. With lightweight processes, the **only** way to manage an asynchronous activity is via a thread. There are no asynchronous exceptions in a thread. Critical sections are implemented with monitors. There is no need to lock out interrupts, with the concomitant possibility of losing information while in the critical section.

Functionality

The Sun *lightweight process library* provides primitives for manipulating threads, as well as for controlling all events (interrupts and traps) on a processor. The present library is supported for user-level processes only. This means that the time slice given to a process by the operating system is shared by all the threads within that process. Further, LWP objects are not accessible outside of the containing UNIX process. Briefly, the primitives supported by the library include:

- Thread creation, destruction, status gathering, scheduling manipulation, suspend and resume
- Multiplexing the clock (any number of threads can sleep concurrently)
- Individualized context switching (e.g., it is possible to specify that a given set of threads will touch floating point registers and only those threads will context switch these registers)
- Monitors and condition variables to synchronize threads
- Extended rendezvous (message send-receive-reply) between threads
- An exception handling facility that provides both *notify* and *escape* exceptions
- A way to map interrupts into extended rendezvous
- A way to map traps into exceptions
- Utilities to allocate red-zone-protected stacks, and to provide some stack integrity checking for environments that lack sophisticated memory management

Scheduling is by default, priority-based, non-preemptive within a priority. However, sufficient primitives are available that it is possible to write your own scheduler. For example, to provide a round-robin time-sliced scheduler, a high-priority thread may periodically reshuffle the queue of time-sliced threads which are at a lower priority. Although pure coroutine scheduling is possible, it is *not* required and purely preemptive scheduling may be used. Threads currently lack kernel support, so system calls still serialize thread activity, although the *non-blocking I/O library* (*libnbio.a*) mitigates this problem somewhat. When a set of threads are running, it is assumed that they all share memory.

Tutorial Goals

This tutorial provides some practical examples of how to program using lightweight processes. Also included is some discussion of the rationale for the lightweight process primitives. Syntax details of the lightweight process primitives are not supplied in this tutorial, though they can be found in the *SunOS Reference Manual.*

6.2. Threads

The lightweight process mechanism allows several *threads* of control to share the same address space. Each lightweight process is represented by a procedure which will be converted into a thread by the `lwp_create()` primitive. Once created, a thread is an independent entity, with its own stack as supplied by its creator. `lwp_create()` performs a number of actions: a thread context is allocated, the stack is initialized, and the thread is made eligible to run. A collection of threads runs within a single ordinary process. This collection is sometimes called a *pod.*

Lightweight processes (*LWP's* or *threads*) are scheduled by priority. It is always the case that the highest priority non-blocked thread is executing. Threads may block on certain occurrences, such as the arrival of a message or the procurement of a monitor lock. Within a priority, threads execute on a first-come, first-served basis. Thus, if two threads are created at the same priority, they will execute in the order of creation.

Here is an example of how to do something simple with lightweight processes. The program below creates a thread which prints out the "hello world" message and then terminates (by "falling through" the procedure). `main()` becomes a lightweight process as soon as a LWP primitive (here, `pod_setmaxpri()`) is called. Note that `main()` is created with a priority of `MAXPRIO` so that it may set things up as it wishes before allowing other threads to run.

```
#include <lwp/lwp.h>
#include <lwp/stackdep.h>
#define MAXPRIO 10
main(argc, argv)
    int argc;
    char **argv;
{
    thread_t tid;
    int task();

    printf("main here\n");                 /* 1 */
    (void)pod_setmaxpri(MAXPRIO);          /* 2 */
    lwp_setstkcache(1000, 2);              /* 3 */
    lwp_create(&tid, task, MAXPRIO, 0,
      lwp_newstk(), 0);                    /* 4 */
}

task() {
    printf("hello world\n");
    /* now, “fall through” and terminate this thread */
}
```

The command to compile this program (call it foo.c) is:

```
example cc -o foo foo.c -llwp
```

Let's go through this program line by line. We begin by printing a message "main here" at line 1. Then, `pod_setmaxpri()` turns `main()` into a lightweight process (as it's the first LWP primitive to be called). `pod_setmaxpri()` also specifies the maximum scheduling priority: in this case, 10. The range of scheduling priorities 1..10 is now available to the client. If we didn't use `pod_setmaxpri()` the available priority would be just MINPRIO. Now, `main()` is a thread running at a priority of 10, the maximum priority. In other words, `main()` will execute until it explicitly blocks or otherwise yields control to another thread.

`lwp_setstkcache()` initializes a cache of stacks that can be used by subsequent `lwp_newstk()` calls. `lwp_newstk()` will return a stack of *at least* the size specified in the `lwp_setstkcache()` call (here, 1000 bytes), and this stack is red-zone protected. The second argument to `lwp_setstkcache()`

specifies how big the cache should be initially (how many stacks it should contain). Larger numbers will require more memory, but will make cache faults less likely. On a fault, an additional cache of the same size will be allocated. A stack allocated from the stack cache will automatically be freed when the thread that uses it dies. Allocation from this cache is almost as efficient as using statically allocated stacks.

At line 4, we create a new thread. This thread will begin execution at `task()`, have a scheduling priority of 10, use the stack cache for a stack, and take no arguments initially. Even though it will run at the same priority as `main()`, `task()` will not run until `main()` relinquishes control because of the FCFS scheduling policy for threads at the same priority, and `task()` is at the same priority as `main()`. (It is not a good programming practice to rely on the ordering of threads within a priority since this assumption may not hold on a multiprocessor or in the presence of external scheduling). The identity of the new thread is returned in `tid`. This identity may be used in subsequent LWP primitives.

When the `main()` thread "falls through", it terminates. At this point, `task()` will run, print its message, and terminate. The LWP library will notice that no more threads remain, and the program will terminate.

Be careful not to confuse threads with ordinary heavyweight processes. For example, there are no inheritance rules about lightweight processes, and lightweight processes do not have their own set of descriptors.

Stack Issues

Stack Size

A major problem is to determine how big to make the thread stacks. Once this determination is made, you can decide how or if you need protection against exceeding this limit. UNIX presents the same problem to the user, but it rarely causes trouble because the maximum stack length is very big. Allocating large stacks is not a big performance drain because pages are only allocated if actually used. Hence, you can allocate very large stacks fairly casually.

Protecting Against Stack Overflow

`lwp_newstk()` automatically allocates red-zone protected stacks (references beyond the stack limit will generate a SIGSEGV event). There are two ways to ensure stack integrity when not using `lwp_newstk()`. One way is to use the `CHECK()` macro at the beginning of each procedure (before any locals are assigned), in conjunction with the `lwp_checkstkset()` primitive. If the procedure exceeds the thread stack limit, the procedure will return and set a global variable. Another way is to use the `lwp_stkcswset()` primitive. This enables stack checking on context switching. Although this is transparent to the client programs, it may not detect errors until after the stack limit has been exceeded. Thus, with `lwp_stkcswset()`, an error is considered fatal. `CHECK()` detects errors before any damage is done, so error recovery is possible.

It is possible to assign a statically allocated stack to a thread. Thus, in the program above, we could declare a stack as follows, using the macros defined in stackdep.h to declare the stack portably. MINSTACKSZ() is added to include any stack room needed by the LWP library to execute the LWP primitives.

```
#include <lwp/lwpmachdep.h>
#include <lwp/stackdep.h>

stkalign_t stack[1000+MINSTACKSZ];
main()
{
    int task();
    thread_t tid;

    (void)pod_setmaxpri(MAXPRIO);
    lwp_create(&tid, task, MAXPRIO, 0, STKTOP(stack), 0);
}
```

Coroutines

It is possible to use threads as pure coroutines in which one thread explicitly yields control to another. lwp_yield() allows a thread to yield to either a specific thread at the same priority, or the next thread in line at the same priority. Here is an example of three coroutines: main(), coroutine(), and other(). The result should be the numbers 1 through 7 printed in sequence. In the case where a generic yield is done (lwp_yield(THREADNULL)), the current thread goes to the end of its scheduling queue. When a specific yield is done, the specified thread butts in front of the current one at the front of the scheduling queue. Since we are just using coroutines, a single priority (MINPRIO) is sufficient and we do not increase the number of available priorities with pod_setmaxpri().

```
#include <lwp/lwp.h>
#include <lwp/stackdep.h>

thread_t co1;        /* main's tid */
thread_t co2;        /* coroutine's tid */
thread_t co3;        /* other's tid */
main(argc, argv)
    int argc;
    char **argv;
{
    int coroutine(), other();
    lwp_self(&co1);

    lwp_setstkcache(1000, 3);
    lwp_create(&co2, coroutine, MINPRIO, 0,
      lwp_newstk(), 0);
    lwp_create(&co3, other, MINPRIO, 0, lwp_newstk(), 0);
    printf("1\n");
    lwp_yield(THREADNULL);   /* yield to coroutine */
    printf("4\n");
    lwp_yield(co3); /* yield to other */
    printf("6\n");
    exit(0);
}

coroutine() {
    printf("2\n");
    if (lwp_yield(THREADNULL) < 0) {
        lwp_perror("bad yield");
        return;
    }
    printf("7\n");
}

other() {
    printf("3\n");
    lwp_yield(THREADNULL);
    printf("5\n");
}
```

Custom Schedulers

There are three ways to provide scheduling control of threads to the client. One way is to do nothing and simply provide the client a pointer to a thread context which can be scheduled at will. This method suffers from the fact that most clients don't want to be bothered by constructing their own scheduler from scratch. Another way to do it is to provide a single scheduling policy, with very little client control over what runs next. The UNIX system provides such a policy. While this is the simplest (from the point of view of the client) way to go, it makes it difficult to implement policies that take into account the differing response time needs of client threads. We chose to take a middle ground in an effort to avoid these problems. There is a default scheduling policy, but enough primitives are provided that it is possible to construct a wide variety of scheduling policies based on it.

It is possible to custom-build your own scheduler by using the primitives `lwp_suspend()`, `lwp_yield()`, `lwp_resume()`, `lwp_setpri()`, and `lwp_resched()`. `lwp_suspend()` may also be used in debugging, to ensure that a thread is stopped before inspecting it. Here, we give an example of how to build a round-robin time-sliced scheduler. The idea is to have a high priority thread act as a scheduler, with the other threads at a lower priority. This scheduler thread simply sleeps for the desired quantum. When the quantum expires, the scheduler issues a `lwp_resched()` command for the priority of the scheduled threads. This causes a reshuffling of the run queue at that priority.

```
#include <lwp/lwp.h>
#include <lwp/stackdep.h>
#define MAXPRIO 10
main(argc, argv)
    int argc;
    char **argv;
{
    int scheduler(), task(), i;
    (void)pod_setmaxpri(MAXPRIO);
    lwp_setstkcache(1000, 5);
    (void) lwp_create((thread_t *)0, scheduler, MAXPRIO, 0,
      lwp_newstk(), 0);
    for (i = 0; i < 3; i++)
        (void) lwp_create((thread_t *)0, task, MINPRIO, 0,
          lwp_newstk(), 1, i);
    exit(0);
}

scheduler() {
    struct timeval quantum;
    quantum.tv_sec = 0;
    quantum.tv_usec = 10000;
    for(;;) {
        lwp_sleep(&quantum);
        lwp_resched(MINPRIO);
    }
}

/* these tasks are scheduled round-robin, preemptive */
task(arg) {
    for(;;)
        printf("task %d\n", arg);
}
```

Special Context Switching

A thread can pretend to be the only activity executing on its machine even though many threads are running. The LWP library is the entity that provides this illusion. As such, the LWP library provides for *context switches* between threads which cause volatile machine resources to be multiplexed so that each thread operates with its own set of machine resources. In many cases, a context switch requires only that machine registers and the stack be multiplexed. In other cases, floating point state, memory management registers, and even software state may be multiplexed as well. The LWP library allows threads to have differing amounts of switchable state to efficiently allow processes with different resource needs to coexist.

In addition to switchable state, a thread will possess state that is updated by other primitives. This per-thread state includes such information as messages sent to a thread, and monitor locks it holds. The only per-thread state maintained by the library is that used to support the LWP primitives, whereas heavyweight processes entail a considerable amount of per-process state. With threads, this amount of state is much smaller with the intent that only those threads which need to should maintain additional state. Thus, operating-system-specific information such as signal state, accounting information, and file descriptors is not found in the thread context. It is up to the clients to provide as much "weight" as is required.

The reason that special contexts are not directly incorporated into the context of a thread is that not all threads will use these contexts and there is no reason to make a thread pay for something it won't use. The LWP library will allocate a new context buffer for each special context a thread is initialized with, and pass a pointer to this context to the save and restore routines defined for this context. The id of the previous and new threads to use the context are also passed in, in case the save and restore routines maintain per-thread information about a special context. This information could be used, for example, by a memory-management special context to avoid doing work if the previous and current threads access the exact same memory management registers.

To use the special context mechanism, you first define a special context with the `lwp_ctxset()` primitive. This requires that you figure out how to save and restore the state required by your context and provide procedures to do this. In the example below, which context-switches the C-library global `errno`, the routines `__libc_save()` and `__libc_restore()` are provided, and the context they will save into and restore from is of type `libc_ctxt_t`. The routine `libcenable()` is used to define the context, and the global `LibcCtx` remembers the cookie that defines the context.

Once a special context is defined, you may initialize any thread to use the resource multiplexed by the special context by using `lwp_ctxinit()`. The initialization of a given thread to use a special context can be done directly, or, if the resource permits, by catching a trap when the resource is first used by a thread. In the example below, we expect that each thread accessing `errno` will be initialized via `libcset()` to use the special libc context. Threads protected with this special context can read `errno` without fear that another thread can change `errno` (e.g., via a system call) from underneath them. Because this `errno` multiplexing is quite useful, it is available in the routine `lwp_libcset()` which does all of the work for you.

```
typedef struct libc_ctxt_t {
    int libc_errno;
} libc_ctxt_t;
static int LibcCtx;

/* enable libc special contexts */
libcenable()
{
    extern void __libc_save();
    extern void __libc_restore();

    LibcCtx = lwp_ctxset(__libc_save, __libc_restore,
        sizeof (libc_ctxt_t), TRUE);
}

/* set a thread to have libc context */
lwp_libcset(tid)
    thread_t tid;
{
    (void) lwp_ctxinit(tid, LibcCtx);
}

/* routines for saving/restoring global library data. */
```

```
void
__libc_save(cntxt, old, new)
    caddr_t cntxt;
    thread_t old;
    thread_t new;
{
    extern int errno;
#ifdef lint
    old = old;
    new = new;
#endif lint

    ((libc_ctxt_t *)cntxt)->libc_errno = errno;
}

void
__libc_restore(cntxt, old, new)
    caddr_t cntxt;
    thread_t old;
    thread_t new;
{
    extern int errno;
#ifdef lint
    old = old;
    new = new;
#endif lint

    errno = ((libc_ctxt_t *)cntxt)->libc_errno;
}
```

6.3. Messages

Messages vs. Monitors

There are two predominant types of process synchronization in use today: the rendezvous paradigm and the monitor paradigm. The lightweight process package provides both, in part to avoid denying a large number of people their favorite primitives, and in part because each has compelling reasons.

Rendezvous has the advantages that it maps cleanly to Sun interprocess-communications facilities (Sun RPC), can potentially support communication across different address spaces, is higher-level than monitors because both data transmission and synchronization are combined into a single concept, and is a natural way to map asynchronous events into higher-level abstractions since messages are reliable and conditions are not.

The big advantage with monitors are their familiarity to UNIX programmers (via similarity to `sleep()` and `wakeup()` in the kernel), and the efficiency win when protected data is accessed: with rendezvous, a context switch is always required; with monitors, a context switch is only necessary if the monitor lock is busy at the time of access.

Rendezvous Semantics

To use messages, one thread issues a msg_send() and another thread issues a msg_recv(). Whichever thread gets to the corresponding primitive first waits for the other, hence the term *rendezvous*. When the rendezvous takes place, the sender remains blocked until the receiver decides to issue a msg_reply(). Immediately after msg_reply() returns, both threads are unblocked.

It is the responsibility of the sender to provide the buffer space both for a message to be sent to the receiver, and for a reply message from the receiver. Either of these messages may be empty. While the sender is blocked, the receiver has access to the buffers provided by the sender. When the receiver replies, she is undertaking not to use these buffers any more: the transaction is complete. If memory management was used to share address spaces, the sender's buffers would be mapped into the receiver's address space only for the duration of the rendezvous. Because both send and receive buffers are provided by the sender, there is no need for further synchronization to tell the receiver that her reply was accepted by the sender.

Sometimes it is desired to perform a *non-blocking* send in which the sender does not block on a send request. We did not provide this as a primitive because it is easily implemented by using an additional thread to do the send.

Messages and Threads

Messages are sent to threads, and each thread has exactly one queue associated with it to receive messages on. We could have provided message queues (ports) as objects not bound to processes. This would give more flexibility, but would require a more complex selection primitive to really justify the extra functionality. In addition, it would complicate the implementation because we desire to terminate a rendezvous on behalf of the remaining thread should one of the rendezvousing threads be destroyed.

To receive a rendezvous request, a process specifies the identity of the sending thread it wishes to rendezvous with. Optionally, a receiver may specify that *any* sender will do. There is no other form of selection available. If more power is needed, the client can build server processes which act as intelligent ports capable of performing complex selection criteria. Note that the id of the sending thread or agent is supplied to the receiver by the LWP library, so that it is not possible to forge the identity of the sender.

Here is an example of basic message passing. main() creates two threads, sender() and receiver(). Because it has a higher priority, the receiver starts first and blocks, awaiting a rendezvous. Then, the sender runs and prepares a message. However, the sender sleeps for 2 seconds before sending it. In this time, the receiver gave up waiting and tried again, now waiting with infinite patience. The sender wakes up a second later and attempts to rendezvous with the receiver. This rendezvous immediately succeeds, the receiver reads the message, prepares a reply, and replies. At this point, the rendezvous is complete and both sender and receiver are runnable processes. Because the receiver has a higher priority, the message "done receiving" is printed ahead of the "got reply" message. Note that the receiver should *not* touch any of the data mentioned in the send once the reply has been made.

```
#include <lwp/lwp.h>
#include <lwp/stackdep.h>
#include <lwp/lwperror.h>
#define MAXPRIO 10

thread_t c1, c2;

main(argc, argv)
    int argc;
    char **argv;
{
    int sender(), receiver();

    (void)pod_setmaxpri(MAXPRIO);
    lwp_setstkcache(1000, 3);
    lwp_create(&c1, sender, MINPRIO, 0, lwp_newstk(), 0);
    lwp_create(&c2, receiver, MINPRIO+1, 0,
      lwp_newstk(), 0);
    exit(0);
}

sender() {
    char out[20];
    char in[30];
    int i;
    struct timeval wait;

    wait.tv_sec = 2;
    wait.tv_usec = 0;

    for (i = 0; i < 19; i++)
        out[i] = (int)'A' + i;
    out[19] = ' ';
    lwp_sleep(&wait);
    if (msg_send(c2, out, 20, in, 26) == -1) {
        lwp_perror("msg_send");
        return;
    }
    printf("got reply %s\n", in);
}

receiver() {
    int i;
    struct timeval wait;
    char *arg, *res;
    int asz, rsz;
    thread_t sender;

    wait.tv_sec = 1;
    wait.tv_usec = 0;
    /* try one second */
    sender = THREADNULL;        /* take message from anyone */
    if (msg_recv(&sender, &arg, &asz, &res, &rsz, &wait)
      == -1) {
        if (lwp_geterr() != LE_TIMEOUT) {
            lwp_perror("msg_recv");
            return;
        }

        /* wait forever or until message arrives from sender */
        if (msg_recv(&sender, &arg, &asz, &res, &rsz,
          INFINITY) == -1) {
            lwp_perror("msg_recv");
            return;
        }
    }
    printf("got message %s\n", arg);
    for (i = 0; i < rsz - 1; i++)
        res[i] = (int)'B' + i;
    res[rsz - 1] = ' ';
    msg_reply(sender);
    printf("done receiving\n");
}
```

Intelligent Servers

Because the reply can be done at any time, a receiver can receive a number of messages before replying to them. This makes it possible to implement complex servers. In the following example, processes send requests in a random order to a server thread. The server serializes the requests and processes them in the order associated with the request.

```
#include <lwp/lwp.h>
#include <lwp/stackdep.h>
thread_t pt;

typedef struct port_msg {
    int order;
    char *msg;
} port_msg;

#define MAXPRIO 10
main(argc, argv)
    int argc;
    char **argv;
{
    int process();
    int port();

    (void)pod_setmaxpri(MAXPRIO);
    lwp_setstkcache(1000, 3);

    /* argument to new thread is order # */
    lwp_create((thread_t *)0, process, MINPRIO, 0,
      lwp_newstk(), 1, 3);
    lwp_create((thread_t *)0, process, MINPRIO, 0,
      lwp_newstk(), 1, 0);
    lwp_create((thread_t *)0, process, MINPRIO, 0,
      lwp_newstk(), 1, 2);
    lwp_create((thread_t *)0, process, MINPRIO, 0,
      lwp_newstk(), 1, 1);
    lwp_create(&pt, port, MAXPRIO, 0, lwp_newstk(), 0);
    exit(0);
}

process(id)
    int id;
{
    port_msg m;
    char buf[10];

    m.order = id;
    m.msg = buf;
    printf("sending %d\n", id);
    msg_send(pt, (char *)&m, sizeof(port_msg), 0, 0);
    printf("%d replied to\n", id);
}

/*
 * collect messages in any order, process them in order
 */
port()
{
    thread_t sender;
    char *arg;
    int asz;
    port_msg *request;
    thread_t senders[4];
    int i;

    for(i = 0; i < 4; i++) {
        /* convenient way to receive from any sender */
        MSG_RECVALL(&sender, &arg, &asz, 0, 0, INFINITY);
        request = (port_msg *)arg;
        printf("got %d\n", request->order);
        senders[request->order] = sender;
    }
    for (i = 0; i < 4; i++) {
        msg_reply(senders[i]);
    }
}
```

6.4. Agents

Some environments will present asynchronous interrupts to the client. For example, on a bare machine, a character typed at a tty can cause an interrupt to randomly steal control away from the executing program. Similarly, a UNIX signal can interrupt the current thread. Because of the random nature of interrupts, it is hard to understand programs that deal with them. The lightweight process library provides a simple way to transform asynchronous events into synchronous ones.

A message paradigm (as opposed to a monitor paradigm) was chosen for mapping interrupts because an interrupt cannot wait for a monitor lock if held by a client. Even if condition variables are used outside of a monitor, it is still necessary to add memory to the condition variable to prevent races (just before the client decides to sleep, an interrupt comes in, causing a condition to be notified, which is missed by the client, who then sleeps, resulting in deadlock). Adding a flag to a condition to prevent this is analogous to converting the condition into a 1-bit message.

With asynchronous interrupts, an event causes a sort of context switch within the *same* thread. With LWP's, a thread must synchronously *rendezvous* with an interrupt. Thus, to have an event do something asynchronously, it is necessary to use a separate thread to handle it. To simulate typical UNIX signal handling, you would create two threads: one thread to represent the main program, and another thread at a higher priority to represent the signal handler. The latter thread would have an *agent* set up to receive signals.

The agent mechanism is provided to map asynchronous events into messages to a lightweight process. A message from an agent looks exactly like a message from another thread. When you create an agent, you also provide a portion of the pod's address space for the agent to store its message. You cannot receive the next message from an agent until you reply to the current one. Because the LWP scheduler is preemptive, when a UNIX signal is mapped into a message, it will cause the highest priority thread blocked on the agent to run next. Client threads which have agents can use all of the LWP library facilities (monitors, condition variables, messages) to synchronize with other threads.

The agent mechanism does its best to process UNIX signals as rapidly as possible. Nonetheless, it is possible that events will be missed because the kernel does not remember more than one signal occurring while a signal is being processed. Furthermore, signals are not delivered for each occurrence of I/O. Therefore, a thread which wakes up from a `SIGIO` agent for example, should not sleep again until `read()` on the descriptor fails, indicating that another SIGIO will be delivered when more I/O is available.

When an interrupt arrives, the LWP library saves only volatile information about the interrupt, and wakes up any threads waiting on the agent. On a bare machine, volatile information would include for example, the character typed in from a tty. Under SunOS, volatile information includes the state normally delivered to a signal handler as well as the identity of the thread running at the time of the event. This volatile information is passed as a message to the client thread.

System Calls

Non-blocking I/O Library

A set of heavyweight processes can execute concurrently in the kernel. For example, three heavyweight processes can concurrently initiate writes to the same device. This is not the case for lightweight threads. Some relief can be provided by marking descriptors asynchronous with `fcntl(2)`. This allows threads to block on `SIGIO` agents and only block on a system call when it is likely to be immediately productive (i.e., without blocking indefinitely). Similarly, a thread can block on a `SIGCHLD` agent instead of blocking on a `wait(2)` system call. However, there is no general solution to the problem of having several threads execute system calls concurrently until the LWP primitives are made available as true system calls operating on a shared set of descriptors. The use of the non-blocking I/O library can help by automatically blocking a thread attempting any I/O until such I/O is likely to succeed immediately. The blocked thread will try the system call again automatically when a SIGIO event occurs.

Using the Non-Blocking IO Library

Here is an example of how to use the non-blocking IO library. We have a procedure *compute_pi* that runs at low priority, and a procedure *reader* that runs at high priority. If we link this program without the non-blocking IO library, the reader will prevent the compute-bound thread from running since the `read()` system call blocks. However, if we link in the non-blocking IO library, the compute-bound procedure will execute until some IO is made available (in this case, by the user typing something at the terminal).

```
#include <lwp/lwp.h>
#include <lwp/stackdep.h>
#define MAXPRIO 10

main(argc, argv)
    int argc;
    char **argv;
{
    int reader();
    thread_t tid;

    pod_setmaxpri(MAXPRIO);
    lwp_setstkcache(3000, 2);
    lwp_create(&tid, reader, MAXPRIO, 0, lwp_newstk(), 0);
    lwp_setpri(SELF, MINPRIO);
    compute_pi();
    exit(0);
}

reader()
{
    char buf[256];
    int cnt;

    for(;;) {
        cnt = read(0, buf, 256);
        buf[cnt] = 0;
        printf("\ngot %s\n", buf);
    }
}

compute_pi()
{
    for(;;) {
        /* compute pi to a zillion places */
    }
}
```

Here is another example of how to use the non-blocking I/O library. The first program is a server which accepts requests over the wire. When a request arrives, a thread is created to handle the request so that accepting and processing the requests can proceed in parallel. The processing of the request consists in sleeping for the amount of time specified in the request message. Note that if the non-blocking I/O library is not linked in, the main program loop prevents any (lower priority) request-processing threads from executing. `lwp_datastk()` is used to put the message on the stack of the newly-created thread. Thus, there is no need to keep the message in *main.*

```
/*
 * sleep server program.
 */
#include <lwp/lwp.h>
#include <lwp/stackdep.h>
#include <lwp/lwperror.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <errno.h>


#define MYPORT  8889
#define MAXPRIO 10
#define BUFSIZE 10

struct message {
    int timeout;
    int msgsize;
    char buf[BUFSIZE];
} message;
extern int errno;

main()
{
    int s;
    struct sockaddr_in addr;
    int len = sizeof(struct sockaddr_in);
    int fromlen;
    int rlen;
    void compute();
    stkalign_t sp;
    caddr_t loc;

    if (pod_setmaxpri(MAXPRIO) < 0) {
        lwp_perror("pod_setmaxpri");
        _exit(1);
    }
    if (lwp_setstkcache(5000, 5) < 0) {
        lwp_perror("lwp_setstkcache");
        _exit(1);
    }
    if ((s = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
    {
        perror("can't get socket");
        _exit(1);
    }
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_family = AF_INET;
    addr.sin_port = MYPORT;
    if (bind(s, (struct sockaddr *)&addr, len) < 0) {
        perror("bind");
        close(s);
        _exit(1);
    }
    if (getsockname(s, (caddr_t)&addr, &len) != 0) {
        perror("can't get name");
        close(s);
        _exit(1);
    }
    for(;;) {
        do {
            fromlen = len;
            rlen = recvfrom(s, (caddr_t)&message,
              sizeof(struct message), 0,
              &addr, &fromlen);
        } while ((rlen == -1) && (errno == EINTR));
        if (rlen == -1) {
            perror("recvfrom");
            _exit(1);
        }
        sp = lwp_datastk(message.buf,
          message.msgsize, &loc);
        lwp_create((thread_t *)0, compute, MINPRIO,
          0, sp, 2, message.timeout, loc);
    }
    exit(0);
}

void
compute(timeout, msg)
    int timeout;
    char *msg;
{
    struct timeval time;
    time.tv_sec = timeout;
    time.tv_usec = 0;

    printf("%s\n", msg);
    lwp_sleep(&time);
    printf("%s slept %d secs\n", msg, timeout);
}
/*
 * program to send a message to the sleep-server.
 * usage: slp <servername> <timeout in seconds> <message>
 */
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <netdb.h>
#include <errno.h>


#define MYPORT  8889
#define BUFSIZE 10

struct messsage {
    int timeout;
    int msgsize;
    char buf[BUFSIZE];
} message;

extern int errno;

main(argc, argv)
    int argc;
    char **argv;
{
    int s;
    struct sockaddr_in addr;
    int len = sizeof(struct sockaddr_in);
    int err;
    struct hostent *hp;
    char *server;


    if (argc != 4) {
        printf("usage: %s server seconds message\n",
          argv[0]);
        exit(2);
    }
    server = argv[1];
    message.timeout = atoi(argv[2]);
    message.msgsize = strlen(argv[3]) + 1;
    bcopy(argv[3], message.buf, message.msgsize);
    if ((hp = gethostbyname(server)) == 0) {
        printf("can't get host name\n");
        exit(1);
    }
    bcopy(hp->h_addr, &addr.sin_addr, hp->h_length);
    addr.sin_family = AF_INET;
    addr.sin_port = MYPORT;

    if ((s = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
    {
        perror("can't get socket");
        exit(1);
    }

    do {
        err = sendto(s, (caddr_t)&message,
          sizeof(message), 0, &addr, len);
    } while ((err == -1) && (errno == EINTR));
    if (err == -1) {
        perror("sendto");
        exit(1);
    }
    exit(0);
}
```

A final example of the non-blocking I/O library illustrates how the `wait(2)` system call can be used. Here, the parent UNIX process forks two children. The children do something (in this case, they just sleep) and terminate with an exit status. The parent would like to reap the children, but does not want to block in the process. The solution is to link in the non-blocking I/O library which lets the parent block without stopping other threads. Behind the scenes, a SIGCHLD agent thread is watching for terminating UNIX processes. If the non-blocking I/O library is not linked in, the *wait* will succeed, but the *otherwork* thread will not get a chance to run. Note that threads using system calls remapped by the non-blocking I/O library automatically receive the C-library special context, so *errno* is not lost across context switches.

```
#include <lwp/lwp.h>
#include <lwp/lwpmachdep.h>
#include <signal.h>

main()
{
    int child;
    union wait stat;
    void otherwork();

    (void)pod_setmaxpri(10);
    (void)lwp_setstkcache(1000, 2);
    (void)lwp_create((thread_t *)0, otherwork,
      MINPRIO, 0, lwp_newstk(), 0);
    if (fork() == 0) {
        sleep(5);
        _exit(7);
    } else if (fork() == 0) {
        sleep(3);
        _exit(5);
    }
    for (;;) {  /* reap children */
        child = wait(&stat);
        printf("%d got %d\n", child, stat.w_retcode);
        if (child == -1) {
            perror("wait");
            break;
        }
    }
    exit(0);
}

void
otherwork()
{
    struct timeval time;
    time.tv_sec = 2;
    time.tv_usec = 0;
    for(;;) {
        printf("otherwork here\n");
        lwp_sleep(&time);
    }
}
```

Examples of Agents

We present two examples of agent use below. The first example shows how a traditional UNIX signal handler can be emulated. Note the use of monitors to protect access to shared state. The second example shows the use of a SIGIO agent.

```
/* Example showing UNIX style signal handling */
#include <lwp/lwp.h>
#include <lwp/stackdep.h>
#include <signal.h>

#define MAXPRIO 10
mon_t mid;
int shared_state;

main(argc, argv)
    int argc;
    char **argv;
{
    int sigint_catch();
    int task();
    int task1();
    (void)pod_setmaxpri(MAXPRIO);
    lwp_setstkcache(3000, 3);
    mon_create(&mid);
    (void) lwp_create((thread_t *)0, sigint_catch, MAXPRIO,
      0, lwp_newstk(), 0);

    /*
     * the signal handler will preempt the main program
     * so we give it the higher priority
     */
    lwp_setpri(SELF, MINPRIO);
    for(;;) {
        /* do other work */;
        mon_enter(mid);
        /* access shared_state */
        mon_exit(mid);
    }
    exit(0);
}

sigint_catch()
{
    eventinfo_t sigmem;
    char *arg;
    int asz;
    thread_t sender;

    agt_create(&sender, SIGINT, (char *)&sigmem);
    for(;;) {
        (void) msg_recv(&sender, &arg, &asz,
          0, 0, INFINITY);
        (void) msg_reply(sender);
        printf("got ^C\n");
        mon_enter(mid);
        /* access shared_state */
        mon_exit(mid);

    }
}

/* Example showing how to process SIGIO */

/*
 * Some points about this code:
 * 1. because the system call could be interrupted, we check for EINTR.
 * In order that errno is accurate, we make sigio_catch a libc thread
 * (else, it may be lost on a context switch).
 *
 * 2. We reset stdin before returning so the shell won't get confused.
 * (It would otherwise get EWOULDBLOCK trying to read stdin, and
 * bomb out with an error).
 */
```

```
#include <lwp/lwp.h>
#include <lwp/stackdep.h>
#include <signal.h>
#include <fcntl.h>
#include <errno.h>
#define TRUE 1
#define MAXPRIO 10

main(argc, argv)
    int argc;
    char **argv;
{
    int sigio_catch();
    thread_t tid;

    (void)pod_setmaxpri(MAXPRIO);
    lwp_setstkcache(3000, 3);
    lwp_create(&tid, sigio_catch, MAXPRIO,
      0, lwp_newstk(), 0);
    lwp_libcset(tid);
    lwp_setpri(SELF, MINPRIO);
    /* do main's work */
}

sigio_catch()
{
    int cnt;
    char buf[256];
    int fd = 0; /* stdin */
    extern int errno;
    int emask, rmask, wmask;
    eventinfo_t agtmemory;
    thread_t sender;
    char *arg;
    int asz;
    int inputbits = 01 << fd;

    /*
     * Enable SIGIO on stdin. When we actually read, it may still
     * return EWOULDBLOCK (SIGINT before SIGIO delivered flushes
     * input leaving nothing to read), so need to read again.
     */
    fcntl(fd, F_SETFL, FASYNC|FNDELAY);
    rmask = inputbits;
    emask = wmask = 0;
    agt_create(&sender, SIGIO, &agtmemory);

    for(;;) {

        /*
         * block pending notification that reading would be useful
         * meanwhile, main can get work done.
         */
        (void) msg_recv(&sender, &arg, &asz,
          0, 0, INFINITY);
        (void) msg_reply(sender);
        select(32, &rmask, &wmask, &emask,
          (struct timevel *)0);
        if (rmask & inputbits) {
            cnt = read(fd, buf, 256);
            if (cnt != -1 || errno != EWOULDBLOCK ||
              errno != EINTR)
                break;
        }
    }
    buf[cnt] = 0;
    printf("\ngot %s\n", buf);
    fcntl(fd, F_SETFL, 0); /* reset stdin so no shell confusion */
}

/*
 * To do simple signal handling within main, we could just write:
 */

main(argc, argv)
    int argc;
    char **argv;
{
    int cnt;
    char buf[256];
    int fd = 0; /* stdin */
    extern int errno;
    int emask, rmask, wmask;
    eventinfo_t agtmemory;
    thread_t sender;
    char *arg;
    int asz;
    int inputbits = 01 << fd;

    (void)pod_setmaxpri(1);
    fcntl(fd, F_SETFL, FASYNC|FNDELAY);
    rmask = inputbits;
    emask = wmask = 0;
    agt_create(&sender, SIGIO, &agtmemory);

    for(;;) {
        (void) msg_recv(&sender, &arg, &asz,
          0, 0, INFINITY);
        (void) msg_reply(sender);
        select(32, &rmask, &wmask, &emask,
          (struct timeval *)0);
        if (rmask & inputbits) {
            cnt = read(fd, buf, 256);
            if (cnt != -1 || errno != EWOULDBLOCK ||
              errno != EINTR)
                break;
        }
    }
    buf[cnt] = 0;
    printf("\ngot %s\n", buf);
    fcntl(fd, F_SETFL, 0);
    exit(0);
}
```

6.5. Monitors and Conditions

The monitor-condition variable paradigm is a familiar one to UNIX kernel programmers because of the analogue to `sleep()` and `wakeup()` in the UNIX kernel.

A monitor implements a *critical section*. This is a reentrant region of code in which access is serialized. As a result, shared data accessed by this code is protected against races that can lead to incorrect interpretations of the data. Once a thread is executing within a monitor, other threads block until that monitor is exited. When thread priorities are equal, they are queued first-come-first-served for access to the monitor. This ensures fair, serial access to the protected data.

As an example, a producer and consumer thread may use a monitor to protect access to a buffer of data being produced or consumed (so that the state of the buffer's "fullness" is consistent). When the producer has filled the buffer, it must wait for the consumer to drain the buffer. This sort of synchronization is provided by *condition variables*. When a thread waits on a condition, it atomically gives up the monitor and blocks pending a *notification*. The result of the notification is that the blocked thread will eventually reacquire the monitor in order to attempt access to the buffer again.

Monitors vs. Interrupt Masking

One goal of lightweight processes is to avoid the use of *sigsetmask's* or other primitives which lock out interrupts to prevent races. By using monitors as a synchronization tool, and by using threads with agents to handle interrupts, the use of interrupt masking can be eliminated, and the risk of dropping interrupts reduced.

Within the LWP library itself, most critical sections are implemented by disabling the scheduler (and *not* by disabling interrupts) for the duration of the critical section. If an interrupt arrives during a critical section, it is processed only to the point of saving the volatile interrupt state. At the end of a critical section, if there are any accumulated events, scheduling decisions are made based upon the agents associated with the events. Interrupts are only masked to ensure that a) the nugget stack is not grown indefinitely by repeated interrupts and b) as a thread is being resumed, to ensure that the new context is loaded atomically. Thus, interrupts are only disabled as a consequence of an interrupt occurring, and never preventively.

Programming with Monitors

Typically, there is some state associated with a condition. When the state acquires a given value, a thread can take some action. Otherwise, it will wait until the state changes. For example, if a buffer is full, a thread writing to the buffer will wait until the state of the buffer indicates that it is no longer full. Another thread reading from the buffer will cooperate by notifying any such waiting thread when the buffer is no longer full. Because the buffer state is accessed by several threads, it is protected by a monitor. Otherwise, a thread could decide to wait for a state change, only to have the state change before the wait can be executed, resulting in deadlock. Therefore, both the waiter and the notifier *must* access the state in a monitor, and the wait primitive (cv_wait) must atomically release the monitor. The typical wait code looks like this:

```
mon_enter(m);
...;
while (!state)
    cv_wait(cv);
...;
mon_exit(m);
```

The while loop is there because if there are several threads waiting in the monitor when the condition is broadcast, all of them wake up, but the first thread to gain entry to the monitor may alter the state, invalidating it for the other awakened threads. In our current example, if two producers are awakened because the buffer is no longer full, the first one may fill the buffer again and wait, leaving the second one to run. The second producer must not add to the buffer now, because it is full again.

Some subtle points about thread scheduling priority should be mentioned. Note that threads queue for monitors and conditions based upon thread priority. No context switch necessarily takes place when a monitor is exited. Thus, a monitor that is repeatedly reentered by a high-priority thread can starve other threads wanting access to the monitor. Care should be taken in assigning priorities to threads using monitors, since a low-priority thread which owns a monitor can still prevent a higher priority thread from accessing that monitor. If a low-priority thread owning a monitor is preempted, it may cause long delays to more important threads needing monitor access.

Monitors and Events

Since events are processed by threads, state manipulated by a thread receiving agent messages can be protected by monitors and condition variables. Thus, after receiving an agent message, a thread may enter a monitor before accessing some global state. Since the LWP library has a large memory for events, no events should be lost if this thread has to block for access to the monitor.

Condition Variables

cv_broadcast() awakens *all* threads blocked on a condition. cv_notify() awakens only a *single* thread blocked on a condition. cv_notify() can result in deadlock states if the awakened thread is not the particular one that should notice a state change and should only be used when it is known that a single other thread is involved. cv_notify() is available because it is more efficient to awaken only a single thread. Note that an awakened thread will be queued to reacquire the monitor. When the thread actually resumes, it will own the monitor it released when it waited for the condition with cv_wait().

Because it is both confusing to the programmer and expensive to implement, no provision for a condition to be shared by several monitors is made. Instead, condition variables are bound to a monitor when they are created. It would be possible to let them be bound when the condition is waited upon, but it would allow the very improbable case of having a waiter awaken in a state testing loop, only to find that his condition was reassigned.

mon_destroy() will remove any conditions bound to the monitor being removed. If mon_destroy() fails because some threads are still waiting on an associated condition, you can use cv_waiters() to see which threads are blocked on conditions associated with the monitor, followed by lwp_destroy() to terminate the blocked threads. After the offending threads are terminated, mon_destroy() should succeed.

Enforcing the Monitor Discipline

Because a thread which forgets to exit a monitor may deadlock the system, it is convenient to use the exception handler mechanism to enforce the enter-exit discipline. The MONITOR() macro enforces this discipline by ensuring that mon_exit() is called when the procedure that embodies the monitor exits. (It is good form to use a single procedure to contain a monitor, viz:)

```
foo() {
    MONITOR(m);
    ...;
}
```

This method ensures that no matter how the procedure is exited (barring longjmp()), the monitor will be exited. That is, if the procedure raises an exception or returns explicitly or implicitly, the monitor is freed.

Nested Monitors

When a thread blocks on a condition while holding several (*nested*) monitor locks, all of the locks except the current one are held. This ensures that the thread does not need to painfully reacquire all of its locks, with the concomitant possibility of deadlock if not all of the locks remain available. If thread T1 holds monitor M1 and wants to acquire monitor M2, and thread T2 holds monitor M2 and wants to acquire monitor M1, deadlock results. One way to avoid this error is to require that the monitors are always acquired in a certain order.

Reentrant Monitors

When a monitor is used to protect a data structure, it may happen, for information hiding reasons, that two different procedures wish to use the same monitor. It may also happen that one of those procedures wishes to use the facilities provided by the other. If these procedures are accessed by the same thread the monitor calls are *reentrant*. If you anticipate such use, you should program your monitors as

```
if (mon_enter(m) < 0) {
    error("bad monitor");
}
```

However, if you wish to catch reentrant monitor use as an error, you should program monitors as:

```
if (mon_enter(m) != 0) {
    error("reentrant monitor");
}
```

Monitor Program Examples

The following is a simple example of monitor use. As described above, we have a producer and a consumer thread, synchronizing with condition variables. To spice it up a bit, we've added some scheduling to make things more realistic.

```
#include <lwp/lwp.h>
#include <lwp/stackdep.h>

thread_t c1, c2, sched;
mon_t m1;
cv_t notempty, notfull;
int cnt = 0;
int in = 0;
int out = 0;
#define MAXBUF  20
char buf[MAXBUF];
#define MAXPRIO 10

main(argc, argv)
    int argc;
    char **argv;
{
    int producer(), consumer();
    int sch();

    (void)pod_setmaxpri(MAXPRIO);
    lwp_setstkcache(3000, 3);
    lwp_create(&c1, producer, MINPRIO+1, 0,
      lwp_newstk(), 0);
    lwp_create(&c2, consumer, MINPRIO, 0,
      lwp_newstk(), 0);
    lwp_create(&sched, sch, MAXPRIO, 0, lwp_newstk(), 0);
    mon_create(&m1);
    cv_create(&notempty, m1);
    cv_create(&notfull, m1);
    exit(0);
}

put(c)   /* add a character to the buffer */
```

```
    char c;
{
    MONITOR(m1);
    while (cnt == MAXBUF) { /* never > MAXBUF chars in buffer */
        printf("waiting on notfull\n");
        cv_wait(notfull);
    }
    buf[in] = c;
    in = (in + 1) % MAXBUF;
    cnt++;
    cv_broadcast(notempty); /* may be a no-op */
}

get(c)
    char *c;
{
    MONITOR(m1);
    while (cnt == 0) { /* never < 0 characters in the buffer */
        printf("waiting on notempty\n");
        cv_wait(notempty);
    }
    *c = buf[out];
    out = (out + 1) % MAXBUF;
    cnt--;
    cv_broadcast(notfull);
}

producer() {
    char c;
    int i;
    int j;

    for(j = 0; j < 500; j++) {
        c = "abcdefghijklmnopqrstuvwxyz"[cnt]; /* produce */
        put(c);
    }
    printf("producer done\n");
}

consumer() {
    char c;
    int i;
    int j;

    for(j = 0; j < 500; j++) {
        get(&c);
        /* consume the character */
    }
    printf("consumer done\n");
}

sch() {
    int k;
    thread_t x = c1;
    struct timeval wait;

    wait.tv_sec = 0;
    wait.tv_usec = 100000;

    for(k = 0; k < 100; k++) {
        lwp_sleep(&wait);
        lwp_setpri(x, MINPRIO);
        if (x == c1)
            x = c2;
        else
            x = c1;
        lwp_setpri(x, MINPRIO+1);
    }
}
```

6.6. Exceptions

The exception primitives can be used to manage synchronous exceptional conditions in a lightweight process. There are no asynchronous exceptions supported by threads because asynchrony can be managed completely with threads and agents, and in a more well-structured fashion. For example, when parsing commands and anticipating an interrupt from the keyboard, you can simply create a thread to parse the command and a thread with an agent to catch the interrupt. When the agent thread catches the interrupt it can simply destroy the parsing thread. This is more elegant than doing a `longjmp()` from a signal handler when an interrupt occurs.

There are several aspects of exceptions. First, you can use *exit_handlers* to be invoked automatically any time a procedure exits. Second, you can provide an exception handler which assumes control anywhere back on the procedure calling chain (*escape exceptions*). Third, you can provide an exception handler which is invoked at the time of an exception and leaves the flow of control alone when it returns (*notification exceptions*). Finally, you can map machine faults (*synchronous traps*) into exceptions. An exception is an event caused by the explicit (or implicit, in the case of synchronous traps) invocation of `exc_raise()`.

When a procedure can exit via a large number of `return` statements or exception raises, it is difficult to monitor the flow of control. Thus, *exit handlers* can be established by `exc_on_exit()` to ensure that a particular action is taken on procedure exit, no matter how the procedure exits. For this reason, no primitive to remove an exit handler is provided, because this provides a way to defeat the whole purpose of exit handlers.

`setjmp()` and `longjmp()` support non-local gotos, but do not give the programmer a disciplined way to invoke them. Pattern-directed handler invocation gives the client an opportunity to establish a set of handlers which are matched by particular patterns. For example, an exception in a memory allocation routine can be raised in such a way that a particular handler (say, a garbage collector) can be explicitly invoked by using a well-known pattern. The `CATCHALL` pattern can be used by a thread either to implement more general sorts of pattern matching (by handling those patterns it wants and discarding those patterns it is not interested in and reraising the exception), or to catch exceptions which must *always* be caught (e.g., a routine which normally allocates some memory permanently and returns should free the memory if an exception occurs).

`exc_notify()` is provided for those exceptions which require an action to be executed on behalf of the exception handler and control to be returned to the raiser of the exception. The handler of a notify exception establishes a function, as well as an argument which can refer to an execution-time environment. By providing a null function, a handler can indicate that only *escape* exceptions (invoked by `exc_raise`()) are to be used.

Exception handling is useful for assisting disciplined use of lightweight process primitives. The `MONITOR()` macro is one example. Another is the `fork()` example discussed in the next section.

Synchronous Traps

Some events are completely synchronous, such as division by zero faults. For such events, it is not logical to allocate a separate thread, since threads are intended to handle asynchronous events. In the lightweight process world, synchronous events appear to be exceptions. Use `agt_trap()` to enable exception mapping for a given event. Note that unhandled exceptions cause termination of the offending thread.

Implementation

One possible way to implement an exception mechanism at the language level would be to use a LWP special context to contain a pointer to the current exception handler for each thread. Using this context, it would be possible to search backwards on the exception chain looking for pattern matches.

Rather than require the client to explicitly pass in a context variable to be used to save and restore exception context, the LWP implementation allocates the context automatically. This is less efficient because by using local variables as contexts, allocation and freeing of the context are free. However, in addition to the more pleasant interface, there are several advantages to the implicit allocation strategy. Because the stack is reset when an exit handler runs, there is no room for local variables to be used by the library code that implements exit handlers (note that the exit handler can make procedure calls of undetermined depth!). This is especially problematic when several exit handlers have been established. Also, if the system being used can't take interrupts on a separate stack, a fair amount of interrupt masking may be required to protect the stack once it is reset.

Exception handling is really a language issue. However, since synchronous traps may be mapped into exceptions, the LWP library itself must be able to access the exception contexts. Thus, the exception handling facility is part of the LWP library and not a separate language facility. In the future, a more flexible interface to `agt_trap()` may be provided so languages can provide their own style of exception handling.

Example of Exception Handling

In the following example, we use the exception handling mechanisms to facilitate a garbage collector. In the event that a resource is exhausted, the client attempts to correct things by notifying the garbage collector. If the next attempt to obtain the resource fails, the client gives up by raising an exception. As an exercise, pretend that the client had resources that needed to be freed as a result of the fatal exception. Use CATCHALL handlers to allow procedures higher up the calling chain to free the resources they allocated.

```
#include <lwp/lwp.h>
#include <lwp/stackdep.h>

#define ATTRIBUTE   9
#define FATAL       7
#define MAXPRIO     10

main(argc, argv)
    char **argv;
{
    int task();

    (void)pod_setmaxpri(MAXPRIO);
    lwp_setstkcache(1000, 3);
    (void) lwp_create((thread_t *)0, task, MINPRIO, 0,
      lwp_newstk(), 0);
    exit(0);
}

task()
{
    int garb_collect();

    /* establish garbage collector for ATTRIBUTE-type resources */
    (void) exc_handle(ATTRIBUTE, garb_collect, ATTRIBUTE);

    /* establish handler for unrecoverable errors */
    if (exc_handle(FATAL, 0, 0) == 0)
        someprocedure();
    else
        abort();
}

someprocedure()
{
    char *r;
    char *getresource();

    r = getresource(ATTRIBUTE);
    /* use resource */
}

char *
getresource(attribute)
```

```
    int attribute;
{
    int (*f)();
    char *resource;
    char *obtain();

    resource = obtain(attribute);        /* try to get resource */
    if (resource == 0)  {                /* couldn't get it */
        (void) exc_notify(attribute);    /* garbage collect */
        resource = obtain(attribute);    /* try again */
        if (resource == 0)               /* still couldn't get it */
            exc_raise(FATAL);            /* give up */
    }
    return(resource);
}

garb_collect(atr)
    int atr;
{
    /*
     * garbage collect resource of type atr such that
     * obtain might succeed if tried again.
     */
}

char *
obtain(atr)
    int atr;
{
    /*
     * try to allocate resource of type atr
     * return 0 if unable to get the resource.
     */
}
```

6.7. Big Example

This example illustrates many of the LWP features: exit handlers, monitors, condition variables, messages, threads. It is a parallel binary tree fringe comparator. Given two binary trees T1 and T2, they have the same fringe if and only if their leaf nodes are equivalent when read left to right.

Part of the program relies on a `fork()` and `join()` mechanism. The idea is that a thread may wish to start some threads and wait for *n* of them to terminate. (To wait for *one* specific thread to die, use *lwp_join*.) Thus, a program could look like:

```
proc() {
     ...;
     tfork(thread1);
     tfork(thread2);
     tfork(thread3);
     join(2);      /* wait for any 2 tforked threads to die */
     ...;
     join(1);      /* wait for last thread to die */
}
```

To make this work, we have `tfork()` create its thread via an intermediary which uses an exit handler (see `exc_on_exit`(3L)) to ensure that the thread calls `die()` when it terminates. `die()` will keep track of the number of terminated threads. Since a `tfork()`'ed thread may be destroyed by another thread, `lwp_destroy()` should be encapsulated by a procedure that calls `die()` as well. This is an illustration of how the exception handling facility can be used to create new protocols (enforced exit actions, for example).

The program begins by declaring two trees (which don't, in this case, have the same fringe). Then, we create three threads: one thread to evaluate each tree, and one thread to compare leaf values and serve as an information exchanger. The two tree evaluators proceed in parallel, sending a message to the comparator containing the leaf value when a leaf is encountered. When the comparator finds a mismatch, it terminates the tree evaluators. When the main program joins successfully, the two evaluators are dead. It then sends a message to the comparator to find out what the results were.

The tree evaluators are simple: they merely recurse down their subtree, pausing to tell the comparator when a leaf is encountered. The comparator is fairly complex. It first receives a message from *either* of the two tree evaluators (which, after all, are running in parallel. As an exercise, add preemptive round-robin scheduling to this program!). Then, it waits for a message from the *other* tree evaluator (else, it could get another value from the same tree evaluator). If the answers disagree, the comparator terminates the evaluators to prevent further (useless and confusing) messages from being sent. Finally, because the two trees being compared may be structurally quite different, one evaluator may finish while the other remains active. As a result, the comparator could do a `msg_recv()` on a non-existent thread. Therefore, we check this condition by noting if `msg_recv()` fails. Just to show that it's possible, this program lints when linted with the LWP lint library!

```
#include <lwp/lwp.h>
#include <lwp/stackdep.h>
#include <lwp/lwperror.h>
#define NULL 0
thread_t cmp, p1, p2;
thread_t driver;
int tfork();
cv_t cv;
mon_t mon;
```

```
int numdead = 0;
typedef struct tree_t {
    int val;
    struct tree_t *left, *right;
} tree_t;
#define TREENULL ((tree_t *) 0)
#define TRUE 1
#define FALSE 0
#define MAXPRIO 10

tree_t t1[] = {
    {0, &t1[1], &t1[2]},
    {1, &t1[3], &t1[4]},
    {4, TREENULL, TREENULL},
    {1, TREENULL, TREENULL},
    {3, TREENULL, &t1[5]},
    {5, TREENULL, TREENULL},
};

tree_t t2[] = {
    {0, &t2[1], &t2[2]},
    {1, TREENULL, TREENULL},
    {2, &t2[3], &t2[4]},
    {3, TREENULL, TREENULL},
    {4, TREENULL, TREENULL},
};

main()
{
    int compare(), parsetree();
    int answer;

    if (pod_setmaxpri(MAXPRIO) == -1)
        lwp_perror("setmaxpri");
    (void)lwp_setstkcache(10000, 5);
    (void)lwp_self(&driver);
    tfork(&cmp, compare, 0);
    tfork(&p1, parsetree, (int)t1);
    tfork(&p2, parsetree, (int)t2);
    join(2);
    (void)msg_send(cmp, (caddr_t)0, 0,
      (caddr_t)&answer, sizeof (answer));
    if (answer)
        (void) printf("same fringe\n");
    else
        (void) printf("not same fringe\n");
    exit(0);
}

compare()
{
    int val1;
    thread_t next;
    thread_t sender;
    int samefringe = TRUE;
    int *resbuf;
    int ressize;
    int *argbuf;
    int argsize;
    int err;

    for(;;) {
        err = MSG_RECVALL(&sender, (caddr_t *)&argbuf,
          &argsize, (caddr_t *)&resbuf,
          &ressize, INFINITY);
        if (err < 0)
            lwp_perror("MSG_RECVALL");
        if (SAMETHREAD(sender, driver)) {
            *resbuf = samefringe;
            (void) msg_reply(driver);
            return;
        }
        vall = *argbuf;
        next = (SAMETHREAD(sender, p1) ? p2 : p1);
        (void) msg_reply(sender);
        err = msg_recv(&next, (caddr_t *)&argbuf,
          &argsize, (caddr_t *)&resbuf,
          &ressize, INFINITY);
        if (err < 0) { /* he died */
            samefringe = FALSE;
            destroy(sender);
        } else {
            samefringe = (*argbuf == vall);
            if (!samefringe) {
                destroy(p1);
                destroy(p2);
            } else
                (void)msg_reply(next);
        }
    }
}

parsetree(t)
    tree_t *t;
{
    if (t == TREENULL)
        return;
    if ((t->left == TREENULL) && (t->right == TREENULL)) {
        /* leaf */
        (void)msg_send(cmp, (caddr_t)&t->val,
          sizeof (int), (caddr_t)0, 0);
    } else {
        parsetree(t->left);
        parsetree(t->right);
    }
}
```

```
tfork(new, adr, arg)
    thread_t *new;
    int (*adr)();
    int arg;
{
    extern void prochelp();
    static int init = 0;

    if (init == 0) {
        init = 1;
        (void)mon_create(&mon);
        (void)cv_create(&cv, mon);
    }
    (void)lwp_create(new, prochelp, MINPRIO, 0,
      lwp_newstk(), 2, adr, arg);
}

void
prochelp(proc, arg)
    int (*proc)();
{
    extern void die();

    (void)exc_on_exit(die, (caddr_t)0);
    proc(arg);
}

void
die()
{
    MONITOR(mon);
    numdead++;
    (void)cv_notify(cv);
}

join(cnt)
{
    MONITOR(mon);
    while (numdead < cnt)
        (void)cv_wait(cv);
    numdead -= cnt;
}

/*  use this instead of lwp_destroy with tfork and join  */
destroy(pid)
    thread_t pid;
{
    die();
    (void)lwp_destroy(pid);
}
```

Index

Notes

Notes

Notes

Notes

Notes

Notes

Notes