Processes 2

Overview

When you execute a command, you start a process that is numbered and tracked by the operating system. A flexible feature of the operating system is that processes are generated by other processes. For example, log in to your system running a shell, then use an editor such as `vi`. Take the option of invoking the shell from `vi`. Execute the `ps` command and you will see a display resembling this (which shows the results of a `ps -f` command):

```
UID   PID    PPID   C   STIME     TTY    TIME   COMD
abc   24210     1   0   06:13:14  tty29  0:05   -sh
abc   24631  24210  0   06:59:07  tty29  0:13   vi c2
abc   28441  28358  80  09:17:22  tty29  0:01   ps -f
abc   28358  24631  2   09:15:14  tty29  0:01   sh -i
```

Here, user `abc` has four processes active. When you trace the chain shown in the process ID (PID) and parent process ID (PPID) columns, you see that the shell that was started when user `abc` logged on is process 24210; its parent is the initialization process (process ID 1). Process 24210 is the parent of process 24631, and so on.

The four processes in the example are shell-level commands, but you can start new processes from your own program.

2

Overlooking the case where your program is interactive and contains many choices for the user, it might need to run one or more other programs based on conditions it encounters in its own processing. The reasons why it might not be practical to create one large executable include:

- The load module might get too big to fit in the maximum process size for your system.
- You might not have control over the object code of all the other modules you want to include.

With the "`fork(2)`" `on page 9` and "`exec(2)`" `on page 10` functions you can create a new process (copy of the creating process) and make a process start a new executable in place of the running one.

Functions

These functions are used to control user processes:

Table 2-1 Process Functions

Function Name	Purpose
`fork`	Create a new process

`exec`
`execl`
`execv`
`execle`
`execve`
`execlp`
`execvp`
	Execute a program

`exit` `_exit`	Terminate a process
`wait`	Wait for a child process to stop or terminate
`dladdr`	Translate address to symbolic information
`dlclose`	Close a shared object
`dlerror`	Get diagnostic information
`dlopen`	Open a shared object
`dlsym`	Get the address of a symbol in a shared object

Table 2-1 Process Functions (Continued)

Function Name	Purpose
`setuid` `setgid`	Set user and group IDs
`setpgrp`	Set process group ID
`chdir` `fchdir`	Change working directory
`chroot`	Change root directory
`nice`	Change priority of a process
`getcontext` `setcontext`	Get and set current user context
`getgroups` `setgroups`	Get or set supplementary group access list IDs

`getpid`
`getpgrp`
`getppid`
`getpgid`
	Get process, process group, and parent process IDs

`getuid`
`geteuid`
`getgid`
`getegid`
	Get real user, effective user, real group, and effective group IDs

`pause`	Suspend process until signal
`priocntl`	Control process scheduler
`setpgid`	Set process group ID
`setsid`	Set session ID
`waitid`	Wait for a child process to change state

Spawning new processes

`fork(2)`

The `fork` call creates a new process that is an exact copy of the calling process. The new process is called the child process; the creator is called the parent process. The child gets a new, unique process ID. When the `fork` function has

finished successfully, it returns a 0 to the child process and the child's process ID to the parent. The returned value is how an executable determines whether it is the parent process or the child process.

Leaving out the possibility of named files, the new process created by the `fork` or `exec` function has the three standard files that are automatically opened: `stdin`, `stdout`, and `stderr`. When the parent has buffered output that should appear before output from the child, the buffers must be flushed before the `fork`.

Also, if the parent and the child process both read input from a stream, whatever is read by one process will be lost to the other. That is, once something has been delivered from the input buffer to a process, the pointer has moved on.

Note – An obsolete practice is to use `fork()` and `exec()` to start another executable, then wait for the new process to die. In effect, a second process is created to perform a subroutine call. It is much more efficient to use `dlopen()`, `dlsym()`, and `dlclose()` as described in "Runtime Linking" on page 11.

`exec`*(2)*

`exec` is the name of a family of functions that includes `execl`, `execv`, `execle`, `execve`, `execlp`, and `execvp`. They all transform the calling process into a new process, but with different ways of pulling together and presenting the arguments of the function. For example, `execl` could be used like this

```
execl("/usr/bin/prog2", "prog2", progarg1, progarg2, (char (*)0);
```

The execl argument list is:

`/usr/bin/prog2`	The path name of the new process file.
`prog2`	The name the new process gets in its `argv[0]`.
`progarg1, progarg2`	The arguments to `prog2` as `char` (*)s.
`(char (*)0)`	A null `char` pointer to mark the end of the arguments.

See `exec`(2) for more details.

The key point about the `exec` family is that there is no return from a successful execution; the new process overlays the process that makes the `exec` call. The new process also takes over the process ID and other attributes of the old process. If the call to `exec` is unsuccessful, control is returned to your program with a return value of –1. You can check `errno` to learn why it failed.

```
main()
{
    pid_t   pid;

    pid = fork;
    switch (pid) {
        case -1:    /* fork failed */
            perror ("fork");
            exit (1);
        case 0:     /* in new chiild process */
            printf ("In child, my pid is: %d\n", getpid(); );
            do_child_stuff();
            exit (0);
        default:    /* in parent, pid contains PID of child */
            printf ("In parent, my pid is %d, my child is %d\n",
                    getpid(), pid);
            break;
    }

    /* Parent process code */
    ...
}
```

Runtime Linking

An application can extend its address space during execution by binding to additional shared objects. There are several advantages in this delayed binding of shared objects:

- Processing a shared object when it is required, rather than during the initialization of an application, may greatly reduce start-up time. Also, the shared object may not be required during a particular run of the application, for example, help or debugging information.
- The application may choose between a number of different shared objects depending on the exact services required, for example, a networking protocol.
- Any shared objects added to the process address space during execution may be freed after use.

A typical scenario that an application may perform to access an additional shared object is:

- A shared object is located and added to the address space of a running application using `dlopen(3X)`. Any dependencies of shared object are also located and added as this time. For example:

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

main(int argc, char ** argv)
{
    void *  handle;
    .....
    if ((handle = dlopen("foo.so.1", RTLD_LAZY)) == NULL) {
        (void) printf("dlopen: %s\n", dlerror());
        exit (1);
    }
    .....
```

- The added shared object(s) are relocated, and any initialization sections in the new shared object(s) are called.
- The application locates symbols in the added shared object(s) using `dlsym(3X)`. The application can then reference the data or call the functions defined by these new symbols. Continuing the preceding example:

```
if (((fptr = (int (*)())dlsym(handle, "foo")) == NULL) ||
    ((dptr = (int *)dlsym(handle, "bar")) == NULL)) {
    (void) printf("dlsym: %s\n", dlerror());
    exit (1);
}
```

- After the application has finished with the shared object(s) the address space is freed using `dlclose(3X)`. Any termination sections within the shared object(s) being freed are called at this time. For example:

```
if (dlcose (handle) != 0) {
    (void) printf("dlclose: %s\n", dlerror());
    exit (1);
}
```

- Any error conditions that occur as a result of using these runtime linker interface routines can be displayed using `dlerror(3X)`.

The services of the runtime linker are defined in the header file `dlfcn.h` and are made available to an application via the shared library `libdl.so.1`. For example:

```
$ cc -o prog main.c -ldl
```

Here the file `main.c` can refer to any of the `dlopen(3X)` family of routines, and the application `prog` will be bound to these routines at runtime.

For a thorough discussion of application directed runtime linking, see Chapter 3 of *Linker and Libraries Guide*. See `dladdr`(3X), `dlclose`(3X), `dlerror`(3X), `dlopen`(3X), and `dlsym`(3X) for use details.

Process Scheduling

The system scheduler determines when processes run. It maintains process priorities based on configuration parameters, process behavior, and user requests; it uses these priorities to assign processes to the CPU.

Scheduler functions give users varying degrees of control over the order in which certain processes run and the amount of time each process may use the CPU before another process gets a chance.

By default, the scheduler uses a time-sharing policy. A time-sharing policy adjusts process priorities dynamically in an attempt to give good response time to interactive processes and good throughput to CPU-intensive processes.

The scheduler offers an alternate real-time scheduling policy as well. Real-time scheduling allows users to set fixed priorities— priorities that the system does not change. The highest priority real-time user process always gets the CPU as soon as it can be run, even if other system processes are also eligible to be run. A program can therefore specify the exact order in which processes run. You can also write a program so that its real-time processes have a guaranteed response time from the system.

For most SunOS 5.x system environments, the default scheduler configuration works well and no real-time processes are needed: administrators need not change configuration parameters and users need not change scheduler properties of their processes. However, for some programs with strict timing constraints, real-time processes are the only way to guarantee that the timing requirements are met.

For more information, see `priocntl(1)`, `priocntl(2)` and `dispadmin(1M)` of the *man Pages(2): System Calls.*

Error Handling

Functions that do not conclude successfully almost always return a value of –1 to your program. (For a few functions in Section 2 of the *man Pages(2): System Calls*, there are a few calls for which no return value is defined, but these are the exceptions.) In addition to the –1 that is returned to the program, the unsuccessful function places an integer in an externally declared variable, `errno`. In a C program, you can determine the value in `errno` if your program contains the following statement

```
#include <errno.h>
```

The value in `errno` is not cleared on successful calls, so check it only if the function returned –1. See error descriptions in `intro`(2) of the *man Pages(2): System Calls*.

You can use the C language function `perror`(3C) to print an error message on `stderr` based on the value of `errno`.

Signals

Overview

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

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

Not all signals have the same priority. If multiple signals are simultaneously pending and deliverable, the signal with the smallest number will be delivered first. A signal routine usually executes concurrently with the signal that caused its invocation, but other signals can still occur. Mechanisms are provided so that critical sections of code can protect themselves against the occurrence of specified signals.

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

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

The set of signals is defined in the header `<signal.h>`.

The signal functions include:

Table 2-2 Signal Functions

Function Name	Purpose

`sigaction`
`sigset`
`sighold`
`sigrelse`
`sigignore`
	Manage signal (detailed)

`sigaltstack`	Set or get signal alternate stack context
`signal` `sigpause`	Manage signal (simplified)
`sigpending`	Examine signals that are blocked and pending
`sigprocmask`	Change or examine signal mask
`kill`	Send a signal to a process or group of processes
`sigsend` `sigsendset`	Send a signal to a process or group of processes
`sigqueue`	Send a signal with a value to a process
`sigwaitinfosigti` `sigtimedwait`	Receive a value and signal synchronously
`sigsuspend`	Install a signal mask and suspend process until signal

Hardware Signals

Hardware signals are derived from exceptional conditions that can 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 or for accesses that violate memory protection constraints
- `SIGBUS`—for accesses that result in hardware-related errors

Other, more CPU-specific hardware signals exist, such as `SIGIOT`, `SIGEMT`, and `SIGTRAP`.

Software Signals

Software signals reflect interrupts generated by user request:

- `SIGINT`—the normal interrupt signal
- `SIGQUIT`—this more powerful quit signal usually causes a `core` image to be generated
- `SIGHUP` and `SIGTERM`—these signals provide graceful process termination, either because a user has "hung up" or through a user or program request
- `SIGKILL`—a more powerful termination signal that a process cannot catch or ignore
- `SIGUSR1` and `SIGUSR2`—allow programs to define their own asynchronous events
- `SIGRTMIN` through `SIGRTMAX`—a range of signals which allow programs to define their own events

Other software signals (`SIGALRM`, `SIGVTALRM`, `SIGPROF`) indicate the expiration of interval timers.

Notification Signals

A process can request notification with a `SIGPOLL` signal when input or output is possible on a descriptor, or when an operation finishes.

A process can request to receive a SIGURG signal when an urgent condition arises on a communication channel.

Process Control Signals

A process can be notified by a signal sent to it or to the members of its process group.

- SIGSTOP—stops the process; this powerful signal cannot be caught
- SIGTSTP—indicates that a user request stopped the process
- SIGTTIN–indicates that an input request stopped the process
- SIGTTOU—indicates that an output request stopped the process
- SIGCONT—indicates that a process continued from a stopped state
- SIGCHLD—notifies a process that a child process has changed state, either by stopping or by terminating

Resource Limit Signals

Exceeding resource limits can generate signals.

- SIGXCPU occurs when a process nears its CPU time limit
- 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.

Each handler specifies an interrupt routine for the signal, that the signal is to be ignored, or that a default action (usually process termination) takes place if the signal occurs. The constants SIG_IGN and SIG_DFL, used as values for sa_handler, cause ignoring or defaulting of a condition.

Note – To reset a signal handler from within a signal handler, reset the signal handler routine that catches the signal (signal(*n*, SIG_DFL);) and unblock the blocked signal with sigprocmask.

Signal Set Operations

The `sa_mask` field specifies the set of signals to be masked when the handler is invoked; it implicitly includes the signal that invoked the handler.

Five operations are permitted on signal sets.

- `sigemptyset`—empties the signal set
- `sigfillset`—fills the signal set with every signal currently supported
- `sigaddset`—adds specific signals to the set
- `sigdelset`—deletes specific signals from the set
- `sigismember`—tests set membership

Initialize signal sets with a call to `sigemptyset` or `sigfillset`.

Unique Signal Properties

The `sa_flags` field specifies unique properties of the signal. Such properties include:

- whether or not functions should be restarted if the signal handler returns
- whether the signal action should be reset to `SIG_DFL` when it is caught
- whether subsequent occurrences of a signal which is already pending should be queued
- whether the handler should operate on the normal runtime stack or on a particular signal stack.

If *osa* is nonzero, the previous signal action is returned.

Signal Generation

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

```
#include <signal.h>

int
kill(pid_t pid, int sig);


#include <signal.h>

int
sigsend(idtype_t idtype, id_t id, int sig);
```

```
int
sigsendset(procset_t *psp, int sig);
```

or

```
#include <signal.h>

int
sigqueue (pid_t pid, int signo, const union sigval value);
```

Unless the process sending the signal is privileged, its real or effective user ID must be that of the receiving process's real or saved user ID.

Signals can also be sent from a terminal device to the process group or session leader associated with the terminal. See the `termio(7I)` manual page for more information.

Signal Delivery

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 `sa_mask` to a set of those masked for the process
- Saves the current process context
- 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 is restored and the process resumes execution in the original context.

Note – For the process to resume in a different context 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 in the same way that a raised hardware interrupt priority level delays hardware interrupts. Preventing an interrupt from occurring by changing the handler is like disabling a device from further interrupts.

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

```
#include <siginfo.h>
#include <ucontext.h>

(*sa_handler)(int signo, siginfo_t *infop, ucontext_t *ucp);
```

The `signo` field gives the number of the signal that occurred. The `infop` field is either equal to 0 or points to a structure that contains information detailing the reason the signal was generated. This information must be explicitly asked for when the signal action is specified. The `ucp` field is a pointer to a structure containing the process's context before delivery of the signal. It restores the process's context upon return from the signal handler.

To block a section of code against one or more signals, use a `sigprocmask` call to add a set of signals to the existing mask and to return the old mask:

```
#include <signal.h>

int
sigprocmask(int SIG_BLOCK,const sigset_t *mask,sigset_t *omask);
```

The old mask can then be restored later with `sigprocmask`

```
#include <signal.h>

int
sigprocmask(int SIG_UNBLOCK, const sigset_t *mask, sigset_t *omask);
```

Or, the old mask can be reset with

```
#include <signal.h>

int
sigprocmask(int SIG_SETMASK, const sigset_t *mask, sigset_t *omask);
```

The `sigprocmask` call can be used to read the current mask without changing it by specifying a null pointer as its `mask` argument.

You can check conditions with some signals blocked, and then pause to wait for a signal and restore the mask, by using

```
#include <signal.h>

int
sigsuspend(const sigset_t *mask);
```

Applications can receive signals synchronously by using

```
#include <signal.h>

int
sigwaitinfo(const sigset_t *mask, siginfo_t *siginfo);

int
sigtimedwait(const sigset_t *mask,siginfo_t *siginfo,
            const struct timespec *timeout);
```

Programs maintaining complex or fixed-size stacks can use the call

```
#include <signal.h>

int
sigaltstack(const stack_t *ss, stack_t *oss);
```

where the `stack_t` structure contains

```
    int     *ss_sp
    long    ss_size
    int     ss_flags
```

This provides the system with a stack based at `ss_sp` of size `ss_size` for signal delivery. The system automatically adjusts for direction of stack growth. `ss_flags` indicates whether the process is currently on the signal stack and whether or not the signal stack is disabled.

When a signal is to be delivered and the process has requested that it be delivered on the alternate stack (see `sigaction` above), the system checks whether the process is on a signal stack. If it is not, then the process is switched to the signal stack for delivery, with the return from the signal arranged to restore the previous stack.

For a process to take a nonlocal exit from the signal routine, or to run code from the signal stack that uses a different stack, use a `sigaltstack` call to reset the signal stack.

2