IV-Unit

Processes and Signals

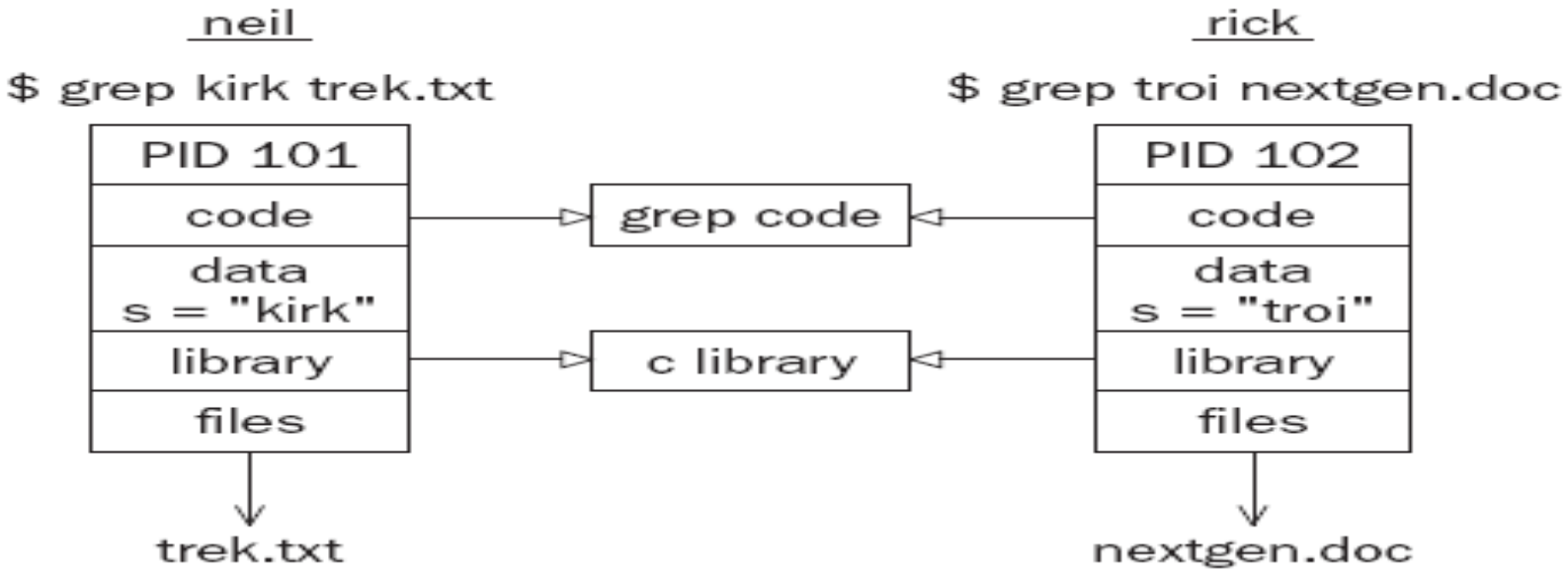
What Is a Process

- The UNIX standards, specifically IEEE Std 1003.1, 2004 Edition, defines a process as “an address space with one or more threads executing within that address space, and the required system resources for those threads.”
- A proces as just a program that is running
- A multitasking operating system such as Linux lets many programs run at once. Each instance of a running program constitutes a process.
- As a multiuser system, Linux allows many users to access the system at the same time. Each user can run many programs, or even many instances of the same program, at the same time

- The system itself runs other programs to manage system resources and control user access.
- a program — or process — that is running consists of program code, data, variables (occupying system memory), open files (file descriptors), and an environment.
- Typically, a Linux system will share code and system libraries among processes so that there's only one copy of the code in memory at any one time.

Process Structure

- Let's have a look at how a couple of processes might be arranged within the operating system.
- If two users, neil and rick, both run the grep program at the same time to look for different strings in different files, the processes being used might look like Figure below.



- If you could run the ps command as in the following code quickly enough and before the searches had finished, the output might contain something like this:

$ **ps -ef (e for all processes and f for full details)**

```
UID   PID   PPID C   STIME   TTY TIME      CMD
rick  101    96   0  18:24    tty2 00:00:00  grep troi nextgen.doc
neil  102    92   0  18:24    tty4 00:00:00  grep kirk trek.txt
```

- The C column shows utilization time.
- Each process is allocated a unique number, called a *process identifier or PID. This is usually a positive* integer between 2 and 32,768.
- When a process is started, the next unused number in sequence is chosen and the numbers restart at 2 so that they wrap around.

- The number 1 is typically reserved for the special init process, which manages other processes.
- Here you see that the two processes started by neil and rick have been allocated the identifiers 101 and 102.
- The program code that will be executed by the grep command is stored in a disk file.
- Normally, a Linux process can't write to the memory area used to hold the program code, so the code is loaded into memory as read-only.
- The system libraries can also be shared. Thus, there need be only one copy of printf, for example, in memory, even if many running programs call it. This is a more sophisticated, but similar, scheme to the way dynamic link libraries (DLLs) work in Windows.

- As you can see in the preceding diagram, an additional benefit is that the disk file containing the executable program grep is smaller because it doesn't contain shared library code.
- This might not seem like much saving for a single program, but extracting the common routines for (say) the standard C library saves a significant amount of space over a whole operating system.
- Of course, not everything that a program needs to run can be shared. For example, the variables that it uses are distinct for each process.
- In this example, you see that the search string passed to the grep command appears as a variable, s, in the data space of each process. These are separate and usually can't be read by other processes. The files that are being used in the two grep commands are also different:

- The processes have their own set of file descriptors used for file access.
- Additionally, a process has its own stack space, used for local variables in functions and for controlling function calls and returns. It also has its own environment space, containing environment variables that may be established solely for this process to use, as you saw with putenv and getenv
- process must also maintain its own program counter, a record of where it has gotten to in its execution, which is the *execution thread*.
- *processes can* have more than one thread of execution.
- On many Linux systems, and some UNIX systems, there is a special set of "files" in a directory called /proc.
- These are special in that rather than being true files they allow you to "look inside" processes while they are running as if they were files in directories.
- Finally, because Linux, like UNIX, has a virtual memory system that pages code and data out to an area of the hard disk, many more processes can be managed than would fit into the physical memory.

The Process Table

- The Linux *process table is like a data structure describing all of the processes that are currently loaded* with, for example, their PID, status, and command string, the sort of information output by ps.
- The operating system manages processes using their PIDs, and they are used as an index into the process table.
- The table is of limited size, so the number of processes a system will support is limited.
- Early UNIX systems were limited to 256 processes. More modern implementations have relaxed this restriction considerably and may be limited only by the memory available to construct a process table entry.

Viewing Processes

- The ps command shows the processes you're running, the process another user is running, or all the processes on the system.

Here is more sample output:

```
$ ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root       433   425  0 18:12 tty1     00:00:00 [bash]
rick       445   426  0 18:12 tty2     00:00:00 -bash
rick       456   427  0 18:12 tty3     00:00:00 [bash]
root       467   433  0 18:12 tty1     00:00:00 sh /usr/X11R6/bin/startx
root       474   467  0 18:12 tty1     00:00:00 xinit /etc/X11/xinit/xinitrc --
root       478   474  0 18:12 tty1     00:00:00 /usr/bin/gnome-session
root       487     1  0 18:12 tty1     00:00:00 gnome-smproxy --sm-client-id def
root       493     1  0 18:12 tty1     00:00:01 [enlightenment]
root       506     1  0 18:12 tty1     00:00:03 panel --sm-client-id default8
```

- This shows information about many processes, including the processes involved with the Emacs editor under X on a Linux system.
- For example, the TTY column shows which terminal the process was started from, TIME gives the CPU time used so far, and the CMD column shows the command used to start the process. Let's take a closer look at some of these

```
neil       655   428  0 18:24 tty4     00:00:00 -bash
```

- The initial login was performed on virtual console number 4. This is just the console on this machine.
- The shell program that is running is the Linux default, bash.

```
root       467   433  0 18:12 tty1     00:00:00 sh /usr/X11R6/bin/startx
```

The X Window System was started by the command `startx`. This is a shell script that starts the X server and runs some initial X programs.

```
root       717   716 13 18:28 pts/0    00:00:01 emacs
```

This process represents a window in X running Emacs. It was started by the window manager in response to a request for a new window. A new pseudo terminal, `pts/0`, has been assigned for the shell to read from and write to.

```
root       512     1  0 18:12 tty1     00:00:01 gnome-help-browser --sm-client-i
```

This is the GNOME help browser started by the window manager.

By default, the `ps` program shows only processes that maintain a connection with a terminal, a console, a serial line, or a pseudo terminal. Other processes run without needing to communicate with a user on a terminal. These are typically system processes that Linux uses to manage shared resources. You can use `ps` to see all such processes using the `-e` option and to get "full" information with `-f`.

System Processes

- Here are some of the processes running on another Linux system. The output has been abbreviated for clarity.
- In the following examples you will see how to view the status of a process. The STAT output from ps provides codes indicating the current status.
- Common codes are given in the following table.

STAT Code	Description
S	Sleeping. Usually waiting for an event to occur, such as a signal or input to become available.
R	Running. Strictly speaking, "runnable," that is, on the run queue either executing or about to run.
D	Uninterruptible Sleep (Waiting). Usually waiting for input or output to complete.
T	Stopped. Usually stopped by shell job control or the process is under the control of a debugger.
Z	Defunct or "zombie" process.
N	Low priority task, "nice."
W	Paging. (Not for Linux kernel 2.6 onwards.)
s	Process is a session leader.
+	Process is in the foreground process group.
l	Process is multithreaded.
<	High priority task.

```
$ ps ax
  PID TTY      STAT   TIME COMMAND
    1 ?        Ss     0:03 init [5]
    2 ?        S      0:00 [migration/0]
    3 ?        SN     0:00 [ksoftirqd/0]
    4 ?        S<     0:05 [events/0]
    5 ?        S<     0:00 [khelper]
    6 ?        S<     0:00 [kthread]
  840 ?        S<     2:52 [kjournald]
  888 ?        S<s    0:03 /sbin/udevd --daemon
 3069 ?        Ss     0:00 /sbin/acpid
 3098 ?        Ss     0:11 /usr/sbin/hald --daemon=yes
 3099 ?        S      0:00 hald-runner
 8357 ?        Ss     0:03 /sbin/syslog-ng
 8677 ?        Ss     0:00 /opt/kde3/bin/kdm
 9119 ?        S      0:11 konsole [kdeinit]
 9120 pts/2    Ss     0:00 /bin/bash
 9151 ?        Ss     0:00 /usr/sbin/cupsd
 9457 ?        Ss     0:00 /usr/sbin/cron
 9479 ?        Ss     0:00 /usr/sbin/sshd -o PidFile=/var/run/sshd.init.pid
```

- Here you can see one very important process indeed.
- 1 ? Ss 0:03 init [5]
- In general, each process is started by another process known as its *parent process*.
- *A process so started* is known as a *child process. When Linux starts, it runs a single program, the prime ancestor and process* number 1, init.
- This is, if you like, the operating system process manager and the grandparent of all processes. Other system processes you'll meet soon are started by init or by other processes started by init.

- One such example is the login procedure. init starts the getty program once for each serial terminal or dial-in modem that you can use to log in. These are shown in the ps output like this:

9619 tty2 Ss+ 0:00 /sbin/mingetty tty2

- The getty processes wait for activity at the terminal, prompt the user with the familiar login prompt, and then pass control to the login program, which sets up the user environment and finally starts a shell.
- When the user shell exits, init starts another getty process.
- You can see that the ability to start new processes and to wait for them to finish is fundamental to the system.

Process Scheduling

- One further ps output example is the entry for the ps command itself:

21475 pts/2 R+ 0:00 ps ax

- This indicates that process 21475 is in a run state (R) and is executing the command ps ax.
- Thus the process is described in its own output! The status indicator shows only that the program is ready to run, not necessarily that it's actually running.
- On a single-processor computer, only one process can run at a time, while others wait their turn. These turns, known as time slices, are quite short and give the impression that programs are running at the same time.
- The R+ just shows that the program is a foreground task not waiting for other processes to finish or waiting for input or output to complete.
- That is why you may see two such processes listed in ps output. (Another commonly seen process marked as running is the X display server.)

- The Linux kernel uses a process scheduler to decide which process will receive the next time slice.
- It does this using the process priority Processes with a high priority get to run more often, whereas others, such as low-priority background tasks, run less frequently.
- With Linux, processes can't overrun their allocated time slice. They are preemptively multitasked so that they are suspended and resumed without their cooperation.
- Older systems, such as Windows 3.*x*, *generally* require processes to yield explicitly so that others may resume.
- In a multitasking system such as Linux where several programs are likely to be competing for the same resource, programs that perform short bursts of work and pause for input are considered better behaved than those that hog the processor by continually calculating some value or continually querying the system to see if new input is available.

- Well-behaved programs are termed *nice programs, and* in a sense this “niceness” can be measured. The operating system determines the priority of a process based on a “nice” value, which defaults to 0, and on the behavior of the program
- Programs that run for long periods without pausing generally get lower priorities. Programs that pause while, for example, waiting for input, get rewarded. This helps keep a program that interacts with the user responsive;
- while it is waiting for some input from the user, the system increases its priority, so that when it’s ready to resume, it has a high priority.
- You can set the process nice value using nice and adjust it using renice. The nice command increases the nice value of a process by 10, giving it a lower priority.

- You can view the nice values of active processes using the –l or –f (for long output) option to ps.
- The value you are interested in is shown in the NI (nice) column.

```
$ ps -l
  F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
000 S   500  1259  1254  0  75   0 -   710 wait4  pts/2    00:00:00 bash
000 S   500  1262  1251  0  75   0 -   714 wait4  pts/1    00:00:00 bash
000 S   500  1313  1262  0  75   0 -  2762 schedu pts/1    00:00:00 emacs
000 S   500  1362  1262  2  80   0 -   789 schedu pts/1    00:00:00 oclock
000 R   500  1363  1262  0  81   0 -   782 -      pts/1    00:00:00 ps
```

Here you can see that the `oclock` program is running (as process 1362) with a default nice value. If it it would have been allocated a nice value of +10. If you adjust this value with the command

```
$ renice 10 1362
1362: old priority 0, new priority 10
```

the clock program will run less often. You can see the modified nice value with `ps` again:

```
$ ps -l
  F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
000 S   500  1259  1254  0  75   0 -   710 wait4  pts/2    00:00:00 bash
000 S   500  1262  1251  0  75   0 -   714 wait4  pts/1    00:00:00 bash
000 S   500  1313  1262  0  75   0 -  2762 schedu pts/1    00:00:00 emacs
000 S   500  1362  1262  0  90  10 -   789 schedu pts/1    00:00:00 oclock
000 R   500  1365  1262  0  81   0 -   782 -      pts/1    00:00:00 ps
```

The status column now also contains `N` to indicate that the nice value has changed from the default.

```
$ ps x
  PID TTY      STAT   TIME COMMAND
 1362 pts/1    SN     0:00 oclock
```

The `PPID` field of `ps` output indicates the parent process ID, the PID of either the process that caused this process to start or, if that process is no longer running, `init` (PID 1).

The Linux scheduler decides which process it will allow to run on the basis of priority. Exact implementations vary, of course, but higher-priority processes run more often. In some cases, low-priority processes don't run at all if higher-priority processes are ready to run.

Starting New Processes

- You can cause a program to run from inside another program and thereby create a new process by using the system library function.

#include <stdlib.h>

int system (const char *string);

- The system function runs the command passed to it as a string and waits for it to complete. The command is executed as if the command
- $ **sh -c string**
- has been given to a shell. system returns 127 if a shell can't be started to run the command and -1 if another error occurs.
- Otherwise, system returns the exit code of the command.

- You can use system to write a program to run ps. Though this is not tremendously useful in and of itself, you'll see how to develop this technique in later examples.
- Actually worked for the sake of simplicity in the example.)

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

int main()
{
    printf("Running ps with system\n");
    system("ps ax");
    printf("Done.\n");
    exit(0);
}
```

When you compile and run this program, `system1.c`, you get something like the following:

```
$ ./system1
Running ps with system
  PID TTY      STAT   TIME COMMAND
    1 ?        Ss     0:03 init [5]
...

 1262 pts/1    Ss     0:00 /bin/bash
 1273 pts/2    S      0:00 su -
 1274 pts/2    S+     0:00 -bash
 1463 pts/2    SN     0:00 oclock
 1465 pts/1    S      0:01 emacs Makefile
 1480 pts/1    S+     0:00 ./system1
 1481 pts/1    R+     0:00 ps ax
Done.
```

- Because the system function uses a shell to start the desired program, you could put it in the background by changing the function call in system1.c to the following:

```
system("ps ax &");
```

When you compile and run this version of the program, you get something like

```
$ ./system2
Running ps with system
  PID TTY      STAT   TIME COMMAND
    1 ?        S      0:03 init [5]
...
Done.
$   1274 pts/2    S+     0:00 -bash
 1463 pts/1    SN     0:00 oclock
 1465 pts/1    S      0:01 emacs Makefile
 1484 pts/1    R      0:00 ps ax
```

How It Works

- In the first example, the program calls system with the string "ps ax", which executes the ps program.
- The program returns from the call to system when the ps command has finished. The system function can be quite useful but is also limited.

- Because the program has to wait until the process started by the call to system finishes, you can't get on with other tasks.
- In the second example, the call to system returns as soon as the shell command finishes. Because it's a request to run a program in the background, the shell returns as soon as the ps program is started, just as would happen if you had typed
- $ **ps ax &**
- at a shell prompt. The system2 program then prints Done. and exits before the ps command has had a chance to finish all of its output.
- The ps output continues to produce output after system2 exits and in this case does not include an entry for system2.
- This kind of process behavior can be quite confusing for users.
- To make good use of processes, you need finer control over their actions. Let's look at a lowerlevel interface to process creation, exec

Replacing a Process Image:

- There is a whole family of related functions grouped under the exec heading.
- They differ in the way that they start processes and present program arguments.
- An exec function replaces the current process with a new process specified by the path or file argument.
- You can use exec functions to "hand off" execution of your program to another.
- For example, you could check the user's credentials before starting another application that has a restricted usage policy. The exec functions are more efficient than system because the original program will no longer be running after the new one is started.

```
#include <unistd.h>

char **environ;

int execl(const char *path, const char *arg0, ...,  (char *)0);
int execlp(const char *file, const char *arg0, ...,  (char *)0);
int execle(const char *path, const char *arg0, ...,  (char *)0, char *const
envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execve(const char *path, char *const argv[], char *const envp[]);
```

- These functions belong to two types. execl, execlp, and execle take a variable number of arguments ending with a null pointer.
- execv and execvp have as their second argument an array of strings.
- In both cases, the new program starts with the given arguments appearing in the argv array passed to main.

- The base of each is **exec** (execute), followed by one or more letters:
- **e** – An array of pointers to environment variables is explicitly passed to the new process image.
- **l** – Command-line arguments are passed individually (a **l**ist) to the function.
- **p** – Uses the PATH environment variable to find the file named in the *file* argument to be executed.
- **v** – Command-line arguments are passed to the function as an array (**v**ector) of pointers.
- path :The argument specifies the path name of the file to execute as the new process image. Arguments beginning at *arg0* are pointers to arguments to be passed to the new process image. The *argv* value is an array of pointers to arguments

- arg0:The first argument *arg0* should be the name of the executable file. Usually it is the same value as the *path* argument.
- envp:Argument *envp* is an array of pointers to environment settings. The *exec* calls named ending with an *e* alter the environment for the new process image by passing a list of environment settings through the *envp* argument. This argument is an array of character pointers; each element (except for the final element) points to a null-terminated string defining an environment variable.

- These functions are usually implemented using execve, though there is no requirement for it to be done this way.
- The functions with names suffixed with a p differ in that they will search the PATH environment variable
- To find the new program executable file. If the executable isn't on the path, an absolute filename, including directories, will need to be passed to the function as a parameter.
- The global variable environ is available to pass a value for the new program environment.
- Alternatively, an additional argument to the functions execle and execve is available for passing an array of strings to be used as the new program environment.
- If you want to use an exec function to start the ps program, you can choose from among the six exec family functions.

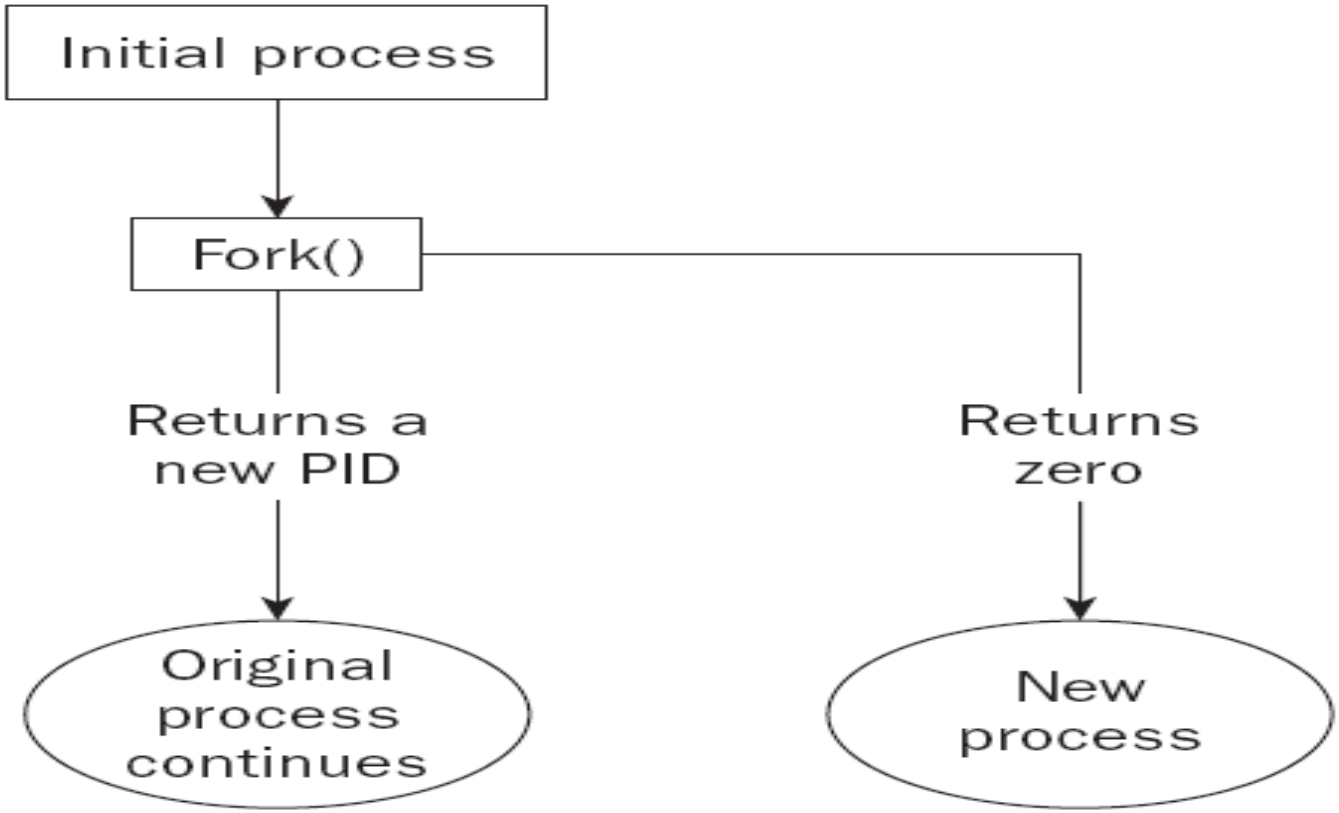
Duplicating a Process Image

- To use processes to perform more than one function at a time, you can either use threads, or create an entirely separate process from within a program, as init does, rather than replace the current thread of execution, as in the exec case.
- You can create a new process by calling fork. This system call duplicates the current process, creating a new entry in the process table with many of the same attributes as the current process.
- The new process is almost identical to the original, executing the same code but with its own data space, environment, and file descriptors.

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

pid_t  fork(void);
```

- Figure below, the call to fork in the parent returns the PID of the new child process.
- The new process continues to execute just like the original, with the exception that in the child process the call to fork returns 0.
- This allows both the parent and child to determine which is which. If fork fails, it returns -1.
- This is commonly due to a limit on the number of child processes that a parent may have (CHILD_MAX), in which case errno will be set to EAGAIN.



- If there is not enough space for an entry in the process table, or not enough virtual memory, the errno variable will be set to ENOMEM.

Ex:

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

int main()
{
     pid_t pid;
     char *message;
     int n;

     printf("fork program starting\n");
     pid = fork();
     switch(pid)
     {
     case -1:
          perror("fork failed");
          exit(1);
     case 0:
          message = "This is the child";
          n = 5;
          break;
     default:
          message = "This is the parent";
          n = 3;
          break;
     }

     for(; n > 0; n--) {
          puts(message);
          sleep(1);
     }
     exit(0);
}
```

- This program runs as two processes. A child is created and prints a message five times.
- The original process (the parent) prints a message only three times. The parent process finishes before the child has printed all of its messages, so the next shell prompt appears mixed in with the output.
- $ **./fork1**
- fork program starting
- This is the child
- This is the parent
- This is the parent
- This is the child
- This is the parent
- This is the child

Waiting for a Process

- When you start a child process with fork, it takes on a life of its own and runs independently.
- Sometimes, you would like to find out when a child process has finished.
- For example, the parent finishes ahead of the child and you get some messy output as the child continues to run.
- You can arrange for the parent process to wait until the child finishes before continuing by calling wait.

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *stat_loc);
```

- The wait system call causes a parent process to pause until one of its child processes is stopped.
- The call returns the PID of the child process. This will normally be a child process that has terminated.
- The status information allows the parent process to determine the exit status of the child process, that is, the value returned from main or passed to exit.
- If stat_loc is not a null pointer, the status information will be written to the location to which it points.
- You can interpret the status information using macros defined in sys/wait.h, shown in the following table.

Macro	Definition
`WIFEXITED(stat_val)`	Nonzero if the child is terminated normally.
`WEXITSTATUS(stat_val)`	If `WIFEXITED` is nonzero, this returns child exit code.
`WIFSIGNALED(stat_val)`	Nonzero if the child is terminated on an uncaught signal.
`WTERMSIG(stat_val)`	If `WIFSIGNALED` is nonzero, this returns a signal number.
`WIFSTOPPED(stat_val)`	Nonzero if the child has stopped.
`WSTOPSIG(stat_val)`	If `WIFSTOPPED` is nonzero, this returns a signal number.

In this Try It Out, you modify the program slightly so you can wait for and examine the child process exit status. Call the new program wait.c.

```
#include <stdlib.h>

int main()
{
     pid_t pid;
     char *message;
     int n;
     int exit_code;

     printf("fork program starting\n");
     pid = fork();
     switch(pid)
     {

     case -1:
          perror("fork failed");
          exit(1);
     case 0:
          message = "This is the child";
          n = 5;
          exit_code = 37;
          break;
     default:
          message = "This is the parent";
          n = 3;
          exit_code = 0;
          break;
     }

     for(; n > 0; n--) {
          puts(message);
          sleep(1);
     }
```

This section of the program waits for the child process to finish.

```
    if (pid != 0) {
        int stat_val;
        pid_t child_pid;

        child_pid = wait(&stat_val);

        printf("Child has finished: PID = %d\n", child_pid);
        if(WIFEXITED(stat_val))
            printf("Child exited with code %d\n", WEXITSTATUS(stat_val));
        else
            printf("Child terminated abnormally\n");
    }
    exit(exit_code);
}
```

- When you run this program, you see the parent wait for the child.

```
$  ./wait
fork program starting

This is the child
This is the parent
This is the parent
This is the child
This is the parent
This is the child
This is the child
This is the child
Child has finished: PID = 1582
Child exited with code 37
$
```

❑ The parent process, which got a nonzero return from the fork call, uses the wait system call to suspend its own execution until status information becomes available for a child process.

❑This happens when the child calls exit; we gave it an exit code of 37.

❑The parent then continues, determines that the child terminated normally by testing the return value of the wait call, and extracts the exit code from the status information.

Zombie Processes

- Using fork to create processes can be very useful, but you must keep track of child processes.
- When a child process terminates, an association with its parent survives until the parent in turn either terminates normally or calls wait.
- The child process entry in the process table is therefore not freed up immediately.
- Although no longer active, the child process is still in the system because its exit code needs to be stored in case the parent subsequently calls wait. It becomes what is known as defunct, or a *zombie process*.

- You can see a zombie process being created if you change the number of messages in the fork example program.
- If the child prints fewer messages than the parent, it will finish first and will exist as a zombie until the parent has finished.

Ex:fork2.c is the same as fork1.c, except that the number of messages printed by the child and parent processes is reversed. Here are the relevant lines of code:

```
switch(pid)
{
case -1:
    perror("fork failed");
    exit(1);
case 0:
    message = "This is the child";
    n = 3;
    break;
default:
    message = "This is the parent";
    n = 5;
    break;
}
```

- If you run the preceding program with ./fork2 & and then call the ps program after the child has finished but before the parent has finished, you'll see a line such as this. (Some systems may say <zombie> rather than <defunct>.)

```
$ ps -al
  F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
004 S     0  1273  1259  0  75   0 -   589 wait4  pts/2    00:00:00 su
000 S     0  1274  1273  0  75   0 -   731 schedu pts/2    00:00:00 bash
000 S   500  1463  1262  0  75   0 -   788 schedu pts/1    00:00:00 oclock
000 S   500  1465  1262  0  75   0 -  2569 schedu pts/1    00:00:01 emacs
000 S   500  1603  1262  0  75   0 -   313 schedu pts/1    00:00:00 fork2
003 Z   500  1604  1603  0  75   0 -     0 do_exi pts/1    00:00:00 fork2 <defunct>
000 R   500  1605  1262  0  81   0 -   781 -      pts/1    00:00:00 ps
```

- If the parent then terminates abnormally, the child process automatically gets the process with PID 1 (init) as parent.
- The child process is now a zombie that is no longer running but has been inherited by init because of the abnormal termination of the parent process.
- The zombie will remain in the process table until collected by the init process. The bigger the table, the slower this procedure. You need to avoid zombie processes, because they consume resources until init cleans them up

- There's another system call that you can use to wait for child processes. It's called waitpid, and you can use it to wait for a specific process to terminate.
- The pid argument specifies the PID of a particular child process to wait for. If it's -1, waitpid will return information for any child process. Like wait, it will write status information to the location pointed to by stat_loc, if that is not a null pointer.
- The options argument allows you to modify the behavior of waitpid. The most useful option is WNOHANG, which prevents the call to waitpid from

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

pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

- Suspending execution of the caller. You can use it to find out whether any child processes have terminated and, if not, to continue. Other options are the same as for wait.
- So, if you wanted to have a parent process regularly check whether a specific child process has terminated, you could use the call waitpid(child_pid, (int *) 0, WNOHANG); This will return zero if the child has not terminated or stopped, or child_pid if it has. waitpid will return -1 on error and set errno.
- This can happen if there are no child processes (errno set to ECHILD),if the call is interrupted by a signal (EINTR), or if the option argument is invalid (EINVAL).

Input and Output Redirection

- The example involves a *filter program*— a program that reads from its standard input and writes to its standard output, performing some useful transformation as it does so.
- Here's a very simple filter program, upper.c, that reads input and converts it to uppercase:

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

int main()
{
    int ch;
    while((ch = getchar()) != EOF) {
        putchar(toupper(ch));
    }
    exit(0);
}
```

- When you run the program, it does what you expect:

$ **./upper**

hello THERE

HELLO THERE

^D

$

- You can, of course, use it to convert a file to uppercase by using the shell redirection

$ **cat file.txt**

this is the file, file.txt, it is all lower case.

$ ./**upper < file.txt**

THIS IS THE FILE, FILE.TXT, IT IS ALL LOWER CASE.

- What if you want to use this filter from within another program? This program, use upper.c, accepts a filename as an argument and will respond with an error if called incorrectly.

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

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

    if (argc != 2) {
        fprintf(stderr, "usage: useupper file\n");
           exit(1);
     }

     filename = argv[1];
```

- You reopen the standard input, again checking for any errors as you do so, and then use execl to call upper.

```
if(!freopen(filename, "r", stdin)) {
fprintf(stderr, "could not redirect stdin from file %s\n", filename);
exit(2);
}
execl("./upper", "upper", 0);
```

- Don't forget that execl replaces the current process; if there is no error, the remaining lines are not executed.

```
perror("could not exec ./upper");
exit(3);
}
```

Threads

- Linux processes can cooperate, can send each other messages, and can interrupt one another.
- They can even arrange to share segments of memory between themselves, but they are essentially separate entities within the operating system. They do not readily share variables.
- There is a class of process known as a *thread that is available in many UNIX and Linux systems.*
- *Though* threads can be difficult to program, they can be of great value in some applications, such as multithreaded database servers.
- Programming threads on Linux (and UNIX generally) is not as common as using multiple processes, because Linux processes are quite lightweight, and programming multiple cooperation processes is much easier than programming threads.

Signals

- A *signal is an event generated by the UNIX and Linux systems in response to some condition, upon* receipt of which a process may in turn take some action.
- We use the term *raise to indicate the generation* of a signal, and the term *catch to indicate the receipt of a signal.*
- *Signals are raised by some error conditions*, such as memory segment violations, floating-point processor errors, or illegal instructions.
- They are generated by the shell and terminal handlers to cause interrupts and can also be explicitly sent from one process to another as a way of passing information or modifying behavior.

- In all these cases, the programming interface is the same Signals can be raised, caught and acted upon, or (for some at least) ignored.

Signal Name	Description
SIGABORT	*Process abort
SIGALRM	Alarm clock
SIGFPE	*Floating-point exception
SIGHUP	Hangup
SIGILL	*Illegal instruction
SIGINT	Terminal interrupt
SIGKILL	Kill (can't be caught or ignored)
SIGPIPE	Write on a pipe with no reader
SIGQUIT	Terminal quit
SIGSEGV	*Invalid memory segment access
SIGTERM	Termination
SIGUSR1	User-defined signal 1
SIGUSR2	User-defined signal 2

- Signal names are defined by including the header file signal.h. They all begin with SIG.
- If a process receives one of these signals without first arranging to catch it, the process will be terminated immediately.
- Usually, a core dump file is created. This file, called core and placed in the current directory, is an image of the process that can be useful in debugging.
- Additional signals include,SIGCHLD can be useful for managing child processes. It's ignored by default.

- The remaining signals cause the process receiving them to stop, except for SIGCONT, which causes the process to resume.
- They are used by shell programs for job control and are rarely used by user programs.

Signal Name	Description
SIGCHLD	Child process has stopped or exited.
SIGCONT	Continue executing, if stopped.
SIGSTOP	Stop executing. (Can't be caught or ignored.)
SIGTSTP	Terminal stop signal.
SIGTTIN	Background process trying to read.
SIGTTOU	Background process trying to write.

- Shell and terminal driver are configured normally, typing the interrupt character (usually Ctrl+C) at the keyboard will result in the SIGINT signal being sent to the foreground process, that is, the program currently running.
- This will cause the program to terminate unless it has arranged to catch the signal.
- If you want to send a signal to a process other than the current foreground task, use the kill command.
- This takes an optional signal number or name, and the PID (usually found using the ps command) to send the signal to.
- For example, to send a "hangup" signal to a shell running on a different terminal with PID 512, you would use the command

$ **kill –HUP 512**

- A useful variant of the kill command is killall, which allows you to send a signal to all processes running a specified command.
- Not all versions of UNIX support it, though Linux generally does.
- This is useful when you do not know the PID, or when you want to send a signal to several different processes executing the same command.
- A common use is to tell the inetd program to reread its configuration options. To do this you can use the command

$ **killall –HUP inetd**

- Programs can handle signals using the signal library function.

```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

- This rather complex declaration says that signal is a function that takes two parameters, sig and func.
- The signal to be caught or ignored is given as argument sig. The function to be called when the specified signal is received is given as func.
- This function must be one that takes a single int argument (the signal received) and is of type void.
- The signal function itself returns a function of the same type, which is the previous value of the function set up to handle this signal, or one of these two special values:

SIG_IGN	Ignore the signal.
SIG_DFL	Restore default behavior.

- The function ouch reacts to the signal that is passed in the parameter sig. This function will be called
- when a signal occurs. It prints a message and then resets the signal handling for SIGINT (by default,
- generated by typing Ctrl+C) back to the default behavior.

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

void ouch(int sig)
{
    printf("OUCH! - I got signal %d\n", sig);
    (void) signal(SIGINT, SIG_DFL);
}
```

- The main function has to intercept the SIGINT signal generated when you type Ctrl+C. For the rest of
- the time, it just sits in an infinite loop, printing a message once a second.

```
int main()
{
    (void) signal(SIGINT, ouch);

    while(1) {
        printf("Hello World!\n");
        sleep(1);
    }
}
```

- Typing Ctrl+C (shown as ^C in the following output) for the first time causes the program to react and then continue.
- When you type Ctrl+C again, the program ends because the behavior of SIGINT has returned to the default behavior of causing the program to exit.

```
$ ./ctrlc1
Hello World!
Hello World!
Hello World!
Hello World!
^C
OUCH! - I got signal 2
Hello World!
Hello World!
```

Sending Signals

- A process may send a signal to another process, including itself, by calling kill.
- The call will fail if the program doesn't have permission to send the signal, often because the target process is owned by another user.
- This is the program equivalent of the shell command of the same name.

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

- The kill function sends the specified signal, sig, to the process whose identifier is given by pid.
- It returns 0 on success. To send a signal, the sending process must have permission to do so.
- Normally, this means that both processes must have the same user ID (that is, you can send a signal only to one of your own processes, although the superuser can send signals to any process).
- kill will fail, return -1, and set errno if the signal given is not a valid one (errno set to EINVAL),
- if it doesn't have permission (EPERM), or if the specified process doesn't exist (ESRCH).

- Signals provide you with a useful alarm clock facility. The alarm function call can be used by a process to schedule a SIGALRM signal at some time in the future.

#include <unistd.h>

unsigned int alarm(unsigned int seconds);

- The alarm call schedules the delivery of a SIGALRM signal in seconds In fact, the alarm will be delivered shortly after that, due to processing delays and scheduling uncertainties.
- A value of 0 will cancel any outstanding alarm request.
- Calling alarm before the signal is received will cause the alarm to be rescheduled.
- Each process can have only one outstanding alarm. alarm returns the number of seconds left before any outstanding alarm call would be sent, or -1 if the call fails.

A Robust Signals Interface

- We've covered raising and catching signals using signal because they are very common in older UNIX programs. However, the X/Open and UNIX specifications recommend a newer programming interface for signals that is more robust: sigaction.

#include <signal.h>

int sigaction(int sig, const struct sigaction *act, struct sigaction *oact

- The sigaction structure, used to define the actions to be taken on receipt of the signal specified by sig, is defined in signal.h and has at least the following members:

```
void (*) (int) sa_handler     /*  function, SIG_DFL or SIG_IGN
sigset_t sa_mask              /*  signals to block in sa_handler
int sa_flags                  /*  signal action modifiers
```

- The sigaction function sets the action associated with the signal sig.
- If oact is not null, sigaction writes the previous signal action to the location it refers to.
- If act is null, this is all sigaction does. If act isn't null, the action for the specified signal is set.
- As with signal, sigaction returns 0 if successful and -1 if not.

Signal Sets

- The header file signal.h defines the type sigset_t and functions used to manipulate sets of signals.
- These sets are used in sigaction and other functions to modify process behavior on receipt of signals.

```
#include <signal.h>
int sigaddset(sigset_t *set, int signo);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigdelset(sigset_t *set, int signo);
```

- These functions perform the operations suggested by their names. sigemptyset initializes a signal set to be empty.
- sigfillset initializes a signal set to contain all defined signals.
- sigaddset and sigdelset add and delete a specified signal (signo) from a signal set.
- They all return 0 if successful and -1 with errno set on error.
- The only error defined is EINVAL if the specified signal is invalid.
- The function sigismember determines whether the given signal is a member of a signal set.
- It returns 1 if the signal is a member of the set, 0 if it isn't, and -1 with errno set to EINVAL if the signal is invalid.

#include <signal.h>

int sigismember(sigset_t *set, int signo);

- The process signal mask is set or examined by calling the function sigprocmask. This signal mask is the set of signals that are currently blocked and will therefore not be received by the current process.

#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oset);

- sigprocmask can change the process signal mask in a number of ways according to the how argument.
- New values for the signal mask are passed in the argument set if it isn't null, and the previous signal mask will be written to the signal set oset.

The how argument can be one of the following:

- SIG_BLOCKThe signals in set are added to the signal mask.
- SIG_SETMASKThe signal mask is set from set.
- SIG_UNBLOCK The signals in set are removed from the signal mask.
- If the set argument is a null pointer, the value of how is not used and the only purpose of the call is to fetch the value of the current signal mask into oset.
- If it completes successfully, sigprocmask returns 0, or it returns -1 if the how parameter is invalid, in which case errno will be set to EINVAL.

- If a signal is blocked by a process, it won't be delivered, but will remain pending. A program can determine which of its blocked signals are pending by calling the function sigpending.

#include <signal.h>

int sigpending(sigset_t *set);

- A process can suspend execution until the delivery of one of a set of signals by calling sigsuspend.
- This is a more general form of the pause function you met earlier.

#include <signal.h>

int sigsuspend(const sigset_t *sigmask);

- The sigsuspend function replaces the process signal mask with the signal set given by sigmask and then suspends execution.
- It will resume after the execution of a signal handling function. If the received signal terminates the program, sigsuspend will never return. If a received signal doesn't terminate the program, sigsuspend returns -1 with errno set to EINTR.

sigaction Flags

- The sa_flags field of the sigaction structure used in sigaction may contain the values shown in

The following table to modify signal behavior:

- SA_NOCLDSTOP : Don't generate SIGCHLD when child processes stop.
- SA_RESETHAND : Reset signal action to SIG_DFL on receipt.
- SA_RESTART : Restart interruptible functions rather than error with EINTR.
- SA_NODEFER : Don't add the signal to the signal mask when caught.
- The SA_RESETHAND : flag can be used to automatically clear a signal function when a signal is caught, as we saw before.

What Is a Thread?

- Multiple strands of execution in a single program are called *threads. A more precise definition is that a* thread is a sequence of control within a process.
- All the programs you have seen so far have executed as a single process, although, like many other operating systems, Linux is quite capable of running multiple processes simultaneously.
- Indeed, all processes have at least one thread of execution.
- When a process executes a fork call, a new copy of the process is created with its own variables and its own PID

- This new process is scheduled independently, and (in general) executes almost independently of the process that created it.
- When we create a new thread in a process, in contrast, the new thread of execution gets its own stack (and hence local variables) but shares global variables, file descriptors, signal handlers, and its current directory state with the process that created it.
- The concept of threads has been around for some time, but until the IEEE POSIX committee published some standards, they had not been widely available in UNIX-like operating systems, and the implementations that did exist tended to vary between different vendors.

- With the advent of the POSIX 1003.1c specification, all that changed; threads are not only better standardized, but are also available on most Linux distributions.
- Now that multi-core processors have also become common even in desktop machines, most machines also have underlying hardware support that allows them to physically execute multiple threads simultaneously.
- Previously, with single-core CPUs, the simultaneous execution of threads was just a clever, though very efficient, illusion.
- Linux first acquired thread support around 1996, with a library often referred to as “LinuxThreads.

- ” This was very close to the POSIX standard (indeed, for many purposes the differences are not noticeable) and it was a significant step forward that enabled Linux programmers to use threads for the first time.
- However, there where slight discrepancies between the Linux implementation and the POSIX standard, most notably with regard to signal handling. These limitations were imposed not so much by the library implementation, but more by the limitations of the underlying support from the Linux kernel.

- Various projects looked at how the thread support on Linux might be improved, not just to clear up the slight discrepancies between the POSIX standard and the Linux implementation, but also to improve performance and remove any unnecessary restrictions.
- Much work centered on how user-level threads should map to kernel-level threads. The two principal projects were New Generation POSIX Threads (NGPT) and Native POSIX Thread Library (NPTL). Both projects had to make changes to the Linux kernel to support the new libraries, and both offered significant performance improvements over the older Linux threads.

- In 2002, the NGPT team announced that they did not wish to split the community and would cease adding new features to NGPT, but would continue to work on thread support in Linux, effectively throwing their weight behind the NPTL effort.
- NPTL became the new standard for threads on Linux, with its first mainstream release in Red Hat Linux 9
- Most of the code in this chapter should work with any of the thread libraries, because it is based on the POSIX standard that is common across all the thread libraries. However, you may see some slight differences.
- if you are using an older Linux distribution, particularly if you use ps to look at the examples while they are running.

- Thread ID is represented by the type ‘pthread_t
- Thread would want to know its own thread ID. For this case the following function is used.

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

- **Thread Creation:**

```
#include <pthread.h>
int pthread_create(pthread_t *restrict tidp, const pthread_attr_t
  *restrict attr, void *(*start_rtn)(void), void *restrict arg)
```

The above function requires four arguments:

- The first argument is a pthread_t type address. Once the function is called successfully, the variable whose address is passed as first argument will hold the thread ID of the newly created thread.

- The second argument may contain certain attributes which we want the new thread to contain. It could be priority etc.
- The third argument is a function pointer. This is something to keep in mind that each thread starts with a function and that functions address is passed here as the third argument so that the kernel knows which function to start the thread from.
- As the function (whose address is passed in the third argument above) may accept some arguments also so we can pass these arguments in form of a pointer to a void type. Now, why a void type was chosen? This was because if a function accepts more than one argument then this pointer could be a pointer to a structure that may contain these arguments.

- pthread_join : wait for thread termination

Syntax:

#include <pthread.h>

int pthread_join(pthread_t *thread*, void ***retval*);

- The **pthread_join**() function waits for the thread specified by *thread* to terminate. If that thread has already terminated, then **pthread_join**() returns immediately. The thread specified by *thread* must be joinable.
- If *retval* is not NULL, then **pthread_join**() copies the exit status of the target thread into the location pointed to by *retval*.
- If the target thread was canceled, then **THREAD_CANCELED** is placed in the location pointed to by *retval*.
- On success, **pthread_join**() returns 0; on error, it returns an error number.

Difference between Thread and a Process

- The processes and threads are independent sequences of execution, the typical difference is that threads run in a shared memory space, while processes run in separate memory spaces.
- A process has a self contained execution environment that means it has a complete, private set of basic run time resources purticularly each process has its own memory space. Threads exist within a process and every process has at least one thread.
- On a multiprocessor system, multiple processes can be executed in parallel. Multiple threads of control can exploit the true parallelism possible on multiprocessor systems.
- Threads have direct access to the data segment of its process but a processes have their own copy of the data segment of the parent process.

- Processes are heavily dependent on system resources available while threads require minimal amounts of resource, so a process is considered as heavyweight while a thread is termed as a lightweight process.
- Processes do not share their address space while threads executing under same process share the address space. From the above point its clear that processes execute independent of each other and the synchronization between processes is taken care by kernel only while on the other hand the thread synchronization has to be taken care by the process under which the threads are executing
- Context switching between threads is fast as compared to context switching between processes

Advantages and Drawbacks of Threads

Following are some advantages of using threads:

- ❑ Sometimes it is very useful to make a program appear to do two things at once.
- The classic example is to perform a real-time word count on a document while still editing the text.
- One thread can manage the user's input and perform editing. The other, which can see the same document content, can continuously update a word count variable. The first thread (or even a third one) can use this shared variable to keep the user informed.
- Another example is a multithreaded database server where an apparent single process serves multiple clients, improving the overall data throughput by servicing some requests while blocking others, waiting for disk activity. For a database server, this
- apparent multitasking is quite hard to do efficiently in different

- The performance of an application that mixes input, calculation, and output may be improved by running these as three separate threads.
- While the input or output thread is waiting for a connection, one of the other threads can continue with calculations. A server application processing multiple network connects may also be a natural fit for a multithreaded program.
- Now that multi-cored CPUs are common even in desktop and laptop machines, using multiple threads inside a process can, if the application is suitable, enable a single process to better utilize the hardware resources available.

- In general, switching between threads requires the operating system to do much less work than switching between processes. Thus, multiple threads are much less demanding on resources than multiple processes, and it is more practical to run programs that logically require many threads of execution on single-processor systems. That said, the design difficulties of writing a multithreaded program are significant and should not be taken lightly.

Threads also have drawbacks:

- ❑ Writing multithreaded programs requires very careful design. The potential for introducing subtle timing faults, or faults caused by the unintentional sharing of variables in a ultithreaded

- program is considerable. Alan Cox (the well respected Linux guru) has commented that threads are also known as “how to shoot yourself in both feet at once.

- Debugging a multithreaded program is much, much harder than debugging a single-threaded one, because the interactions between the threads are very hard to control.

- A program that splits a large calculation into two and runs the two parts as different threads will not necessarily run more quickly on a single processor machine, unless the calculation truly allows multiple parts to be calculated simultaneously and the machine it is executing on has multiple processor cores to support true multiprocessing.

- Another problem that may arise is the concurrency problems. Since threads share all the segments (except the stack segment) and can be preempted at any stage by the scheduler than any global variable or data structure that can be left in inconsistent state by preemption of one thread could cause severe problems when the next high priority thread executes the same function and uses the same variables or data structures.

A Simple Threaded Program

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

void *thread_function(void *arg);

char message[] = "Hello World";

int main() {
    int res;
    pthread_t a_thread;
    void *thread_result;

    res = pthread_create(&a_thread, NULL, thread_function, (void *)message);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
    printf("Waiting for thread to finish...\n");
    res = pthread_join(a_thread, &thread_result);
    if (res != 0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }
```

```
    printf("Thread joined, it returned %s\n", (char *)thread_result);
    printf("Message is now %s\n", message);
    exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
    printf("thread_function is running. Argument was %s\n", (char *)arg);
    sleep(3);
    strcpy(message, "Bye!");
    pthread_exit("Thank you for the CPU time");
}
```

When you run this program, you should see the following:

$ **./thread1**

Waiting for thread to finish…

thread_function is running. Argument was Hello World

Thread joined, it returned Thank you for the CPU time

Message is now Bye!

Simultaneous Execution

- The next example shows you how to write a program that checks that the execution of two threads occurs simultaneously. (Of course, if you are using a single-processor system, the CPU would be cleverly switched between the threads, rather than having the hardware simultaneously execute both threads using separate processor cores). Because you haven't yet met any of the thread synchronization functions,this will be a very inefficient program that does what is known as a *polling between the two* threads. Again, you will make use of the fact that everything except local function variables are shared between the different threads in a process.

- The program you create in this section, thread2.c, is created by slightly modifying thread1.c. You add an extra file scope variable to test which thread is running:
- Set run_now to 1 when the main function is executing and to 2 when your new thread is executing.
- In the main function, after the creation of the new thread, add the following code:

```
int print_count1 = 0;

while(print_count1++ < 20) {
    if (run_now == 1) {
        printf("1");
        run_now = 2;
    }
    else {
        sleep(1);
    }
}
```

- If run_now is 1, print "1" and set it to 2. Otherwise, you sleep briefly and check the value again.
- You are waiting for the value to change to 1 by checking over and over again. This is called a *busy wait, although* here it is slowed down by sleeping for a second between checks.
- In thread_function, where your new thread is executing, you do much the same but with the values reversed:

```
int print_count2 = 0;

while(print_count2++ < 20) {
    if (run_now == 2) {
        printf("2");
        run_now = 1;
    }
    else {
        sleep(1);
    }
}
```

- When you run the program, you see the following output. (You may find that it takes a few seconds for
- the program to produce output, particularly on a single-core CPU machine.)

$ **cc -D_REENTRANT thread2.c -o thread2 -lpthread**

$ **./thread2**

12121212121212121212

Waiting for thread to finish…

Thread joined

Synchronization

- In the previous section, you saw that both threads execute together, but the method of switching between them was clumsy and very inefficient. Fortunately, there is a set functions specifically designed to provide better ways to control the execution of threads and access to critical sections of code.
- We look at two basic method here: *semaphores, which act as gatekeepers around a piece of code*.

Synchronization with Semaphores

- Dijkstra, a Dutch computer scientist, first conceived the concept of semaphores. A semaphore is a special type of variable that can be incremented or decremented, but crucial access to the variable is guaranteed to be atomic, even in a multithreaded program.

- semaphore, a *binary semaphore that takes only values 0 or 1*.
- There is also a more general semaphore, a *counting semaphore that takes a wider range of values. Normally,* semaphores are used to protect a piece of code so that only one thread of execution can run it at any one time.
- Occasionally, you want to permit a limited number of threads to execute a given piece of code; for this you would use a counting semaphore. Because counting semaphores are much less common.
- A semaphore is created with the sem_init function, which is declared as follows:

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int
   value);
```

- This function initializes a semaphore object pointed to by sem, sets its sharing option and gives it an initial integer value.
- The pshared parameter controls the type of semaphore.
- If the value of pshared is 0, the semaphore is local to the current process. Otherwise, the semaphore may be shared between processes.
- Here we are interested only in semaphores that are not shared between processes. At the time of writing, Linux doesn't support this sharing, and passing a nonzero value for pshared will cause the call to fail.
- The next pair of functions controls the value of the semaphore and is declared as follows:

#include <semaphore.h>

int sem_wait(sem_t * sem);

int sem_post(sem_t * sem);

- These both take a pointer to the semaphore object initialized by a call to sem_init.
- The sem_post function atomically increases the value of the semaphore by 1.
- *Atomically here means that if* two threads simultaneously try to increase the value of a single semaphore by 1, they do not interfere with each other, as might happen if two programs read, increment, and write a value to a file at the same time.
- If both programs try to increase the value by 1, the semaphore will always be correctly increased in value by 2.

- The sem_wait function atomically decreases the value of the semaphore by one, but always waits until the semaphore has a nonzero count first. Thus, if you call sem_wait on a semaphore with a value of 2, the thread will continue executing but the semaphore will be decreased to 1.
- If sem_wait is called on a semaphore with a value of 0, the function will wait until some other thread has incremented the value so that it is no longer 0.
- If two threads are both waiting in sem_wait for the same semaphore to become nonzero and it is incremented once by a third process, only one of the two waiting processes will get to decrement the semaphore and continue; the other will remain waiting. This atomic "test and set" ability in a single function is what makes semaphores so valuable.

- The last semaphore function is sem_destroy. This function tidies up the semaphore when you have finished with it.
- It is declared as follows:

```
#include <semaphore.h>
int sem_destroy(sem_t * sem);
```

A Thread Semaphore

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>

void *thread_function(void *arg);
sem_t bin_sem;

#define WORK_SIZE 1024
char work_area[WORK_SIZE];

int main() {
    int res;
    pthread_t a_thread;
    void *thread_result;

    res = sem_init(&bin_sem, 0, 0);
    if (res != 0) {
        perror("Semaphore initialization failed");
        exit(EXIT_FAILURE);
    }
    res = pthread_create(&a_thread, NULL, thread_function, NULL);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
    printf("Input some text. Enter 'end' to finish\n");
    while(strncmp("end", work_area, 3) != 0) {
        fgets(work_area, WORK_SIZE, stdin);
        sem_post(&bin_sem);
    }
```

```
    printf("\nWaiting for thread to finish...\n");
    res = pthread_join(a_thread, &thread_result);
    if (res != 0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }
    printf("Thread joined\n");
    sem_destroy(&bin_sem);
    exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
    sem_wait(&bin_sem);
    while(strncmp("end", work_area, 3) != 0) {
        printf("You input %d characters\n", strlen(work_area) -1);
        sem_wait(&bin_sem);
          }
          pthread_exit(NULL);
   }
```

- The first important change is the inclusion of semaphore.h to provide access to the semaphore functions.
- Then you declare a semaphore and some variables and initialize the semaphore *before you create* your new thread.
- In the function main, after you have started the new thread, you read some text from the keyboard, load your work area, and then increment the semaphore with sem_post.
- In the new thread, you wait for the semaphore and then count the characters from the input.
- While the semaphore is set, you are waiting for keyboard input. When you have some input, you release the semaphore, allowing the second thread to count the characters before the first thread reads the keyboard again.

- Again both threads share the same work_area array. Again, we have omitted some error checking, such as the returns from sem_wait to make the code samples more succinct and easier to follow. However, in production code you should always check for error returns unless there is a very good reason to omit this check.

Output:

- $ **cc -D_REENTRANT thread3.c –o thread3 -lpthread**
- $ **./thread3**
- Input some text. Enter ‘end’ to finish
- The Wasp Factory
- You input 16 characters
- Iain Banks
- You input 10 characters
- end

Synchronization with Mutexes

- The other way of synchronizing access in multithreaded programs is with *mutexes (short for mutual exclusions), which act by allowing the programmer to "lock" an object so that only one thread can access* it.
- To control access to a critical section of code you lock a mutex before entering the code section and then unlock it when you have finished.
- The basic functions required to use mutexes are very similar to those needed for semaphores. They are declared as follows:

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex, const
    pthread_mutexattr_t *mutexattr);
```

- **int pthread_mutex_lock(pthread_mutex_t *mutex));**
- **int pthread_mutex_unlock(pthread_mutex_t *mutex);**
- **int pthread_mutex_destroy(pthread_mutex_t *mutex);**

- As usual, 0 is returned for success, and on failure an error code is returned, but errno is not set; you must use the return code.
- As with semaphores, they all take a pointer to a previously declared object, in this case a pthread_mutex_t.
- The extra attribute parameter pthread_mutex_init allows you to provide attributes for the mutex, which control its behavior. The attribute type by default is "fast." This has the slight drawback that, if your program tries to call pthread_mutex_lock on a mutex that it has already locked, the program will block.

- Because the thread that holds the lock is the one that is now blocked, the mutex can never be unlocked and
- the program is deadlocked. It is possible to alter the attributes of the mutex so that it either checks for this
- and returns an error or acts recursively and allows multiple locks by the same thread if there are the same
- number of unlocks afterward.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>

void *thread_function(void *arg);
pthread_mutex_t work_mutex; /* protects both work_area and time_to_exit */

#define WORK_SIZE 1024
char work_area[WORK_SIZE];
int time_to_exit = 0;

int main() {
    int res;
    pthread_t a_thread;
    void *thread_result;
    res = pthread_mutex_init(&work_mutex, NULL);
    if (res != 0) {
        perror("Mutex initialization failed");
        exit(EXIT_FAILURE);
    }
    res = pthread_create(&a_thread, NULL, thread_function, NULL);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
```

```
pthread_mutex_lock(&work_mutex);
printf("Input some text. Enter 'end' to finish\n");
while(!time_to_exit) {
    fgets(work_area, WORK_SIZE, stdin);
    pthread_mutex_unlock(&work_mutex);
    while(1) {
        pthread_mutex_lock(&work_mutex);
        if (work_area[0] != '\0') {
            pthread_mutex_unlock(&work_mutex);
            sleep(1);
        }
        else {
            break;
        }
```

```
        }
    }
    pthread_mutex_unlock(&work_mutex);
    printf("\nWaiting for thread to finish...\n");
    res = pthread_join(a_thread, &thread_result);
    if (res != 0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }
    printf("Thread joined\n");
    pthread_mutex_destroy(&work_mutex);
    exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
    sleep(1);
    pthread_mutex_lock(&work_mutex);
    while(strncmp("end", work_area, 3) != 0) {
        printf("You input %d characters\n", strlen(work_area) -1);
        work_area[0] = '\0';
        pthread_mutex_unlock(&work_mutex);
```

```
        sleep(1);
        pthread_mutex_lock(&work_mutex);
        while (work_area[0] == '\0' ) {
            pthread_mutex_unlock(&work_mutex);
            sleep(1);
            pthread_mutex_lock(&work_mutex);
        }
    }
    time_to_exit = 1;
    work_area[0] = '\0';
    pthread_mutex_unlock(&work_mutex);
    pthread_exit(0);
}
```

```
$ cc -D_REENTRANT thread4.c -o thread4 -lpthread
$ ./thread4
Input some text. Enter 'end' to finish
Whit
You input 4 characters
The Crow Road
You input 13 characters
end

Waiting for thread to finish...
Thread joined
```