PROCESS:

When you execute a program on your UNIX system, the system creates a special environment for that program. This environment contains everything needed for the system to run the program as if no other program were running on the system.

Whenever you issue a command in UNIX, it creates, or starts, a new process. When you tried out the ls command to list directory contents, you started a process. A process, in simple terms, is an instance of a running program.

When you start a process (run a command), there are two ways you can run it −

- ✓ Foreground Processes
- ✓ Background Processes

FOREGROUND PROCESSES:

By default, every process that you start runs in the foreground. It gets its input from the keyboard and sends its output to the screen.

Example: $ls ch*.doc

When a program is running in foreground and taking much time, we cannot run any other commands (start any other processes) because prompt would not be available until program finishes its processing and comes out.

BACKGROUND PROCESSES:

A background process runs without being connected to your keyboard. If the background process requires any keyboard input, it waits.

The advantage of running a process in the background is that you can run other commands; you do not have to wait until it completes to start another!

The simplest way to start a background process is to add an ampersand (&) at the end of the command.

Example: $ls ch*.doc &

PROCESS IDENTIFIERS:

The process identifier – normally referred to as the process ID or just PID, is a number used by most operating system kernels such as that of UNIX, Mac OS X or Microsoft Windows — to uniquely identify an active process.

This number may be used as a parameter in various function calls, allowing processes to be manipulated, such as adjusting the process's priority or killing it altogether.

In Unix-like operating systems, new processes are created by the fork() system call. The PID is returned to the parent enabling it to refer to the child in further function calls. The parent may, for example, wait for the child to terminate with the waitpid() function, or terminate the process with kill().

There are three IDs associated with every process, the ID of the process itself (the PID), its parent process's ID (the PPID) and its process group ID (the PGID). Every UNIX process has a unique PID in the range 0 to 30000.

There are two tasks with specially distinguished process IDs:

- ***swapper or sched*** has process ID 0 and is responsible for paging, and is actually part of the kernel rather than a normal user-mode process.
- Process ID 1 is usually the ***init*** process primarily responsible for starting and shutting down the system.

The operating system tracks processes through a five digit ID number known as the PID or process ID. Each process in the system has a unique PID. PIDs eventually repeat because all the possible numbers are used up and the next PID rolls or start over.

At any one time, no two processes with the same PID exist in the system because it is the PID that UNIX uses to track each process. You can see this happen with the ls command.

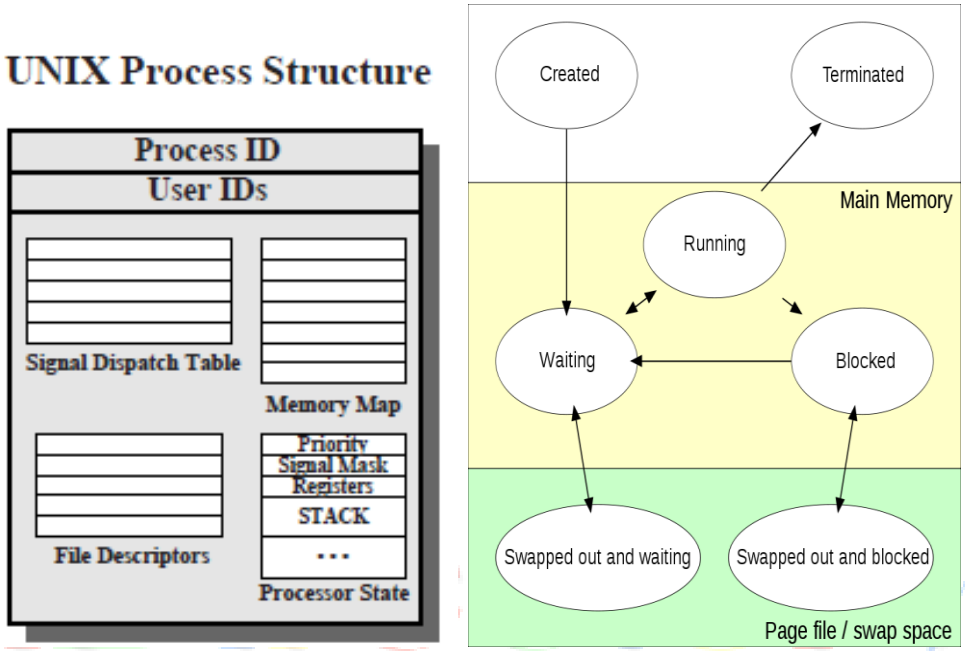
Processes are created in UNIX in an especially simple manner. The fork system call creates an exact copy of the original process. The forking process is called the parent process. The new process is called the child process. The parent and child each have their own, private memory images. If the parent subsequently changes any of its variables, the changes are not visible to the child, and vice versa.

Open files are shared between parent and child. ***Example for creating a process:***

```
pid = fork( );              /*      if the fork succeeds, pid > 0 in the parent     */
if (pid < 0){
        handle_error( );    /* fork failed (e.g., memory or some table is full) */
} else if (pid > 0) {
                            /*      parent code goes here. /*/
} else {
                            /*      child code goes here. /*/
}
```

PROCESS STRUCTURE:

The kernel maintains a process structure for every running process. This structure contains the information that the kernel needs to manage the process, such as various process descriptors (process-id, process group-id, session-id, and so on), a list of open files, a memory map and possibly other process attributes.



The exact information present in any process structure will vary from one implementation to another, but all process structures include:

- Process id
- Parent Process id (pointer to parent's process structure)
- Pointer to list of children of the process
- Process priority for scheduling, statistics about CPU usage and last priority
- Process State
- Signal information (Signals pending, signal mask etc…)

The process structure is located in kernel memory. Different versions of UNIX store it in different ways. In BSD UNIX kernel maintains two lists of process structures called *zombie* list and *allproc* list. Zombies are processes that have terminated but that cannot be destroyed. Zombie processes have their structure on the zombie list. The allproc list contains those that are not zombies.

ZOMBIE PROCESS:

On UNIX and Unix-like computer operating systems, a zombie process or defunct process is a process that has completed execution (via the exit system call) but still has an entry in the process table: it is a process in the "Terminated state".

This occurs for child processes, where the entry is still needed to allow the parent process to read its child's exit status: once the exit status is read via the wait system call, the zombie's entry is removed from the process table and it is said to be acquired.

A child process always first becomes a zombie before being removed from the resource table. In most cases, under normal system operation zombies are immediately waited on by their parent and then acquired by the system – ***processes that stay zombies for a long time are generally an error and cause a resource leak***.

OVERVIEW OF ZOMBIE PROCESS:

When a process ends via exit, all of the memory and resources associated with it are deallocated so they can be used by other processes. However, the process's entry in the process table remains.

The parent can read the child's exit status by executing the wait system call, whereupon the zombie is removed. The wait call may be executed in sequential code, but it is commonly executed in a handler for the SIGCHLD signal, which the parent receives whenever a child has died (terminated).

After the zombie is removed, its process identifier (PID) and entry in the process table can then be reused. However, if a parent fails to call wait, the zombie will be left in the process table, causing a resource leak.

Zombies can be identified in the output from the UNIX ps command by the presence of a "Z" in the "STAT" column. Zombies that exist for more than a short period of time typically indicates a bug in the parent program, or just an uncommon decision to not reap (acquire) children.

If the parent program is no longer running, zombie processes typically indicate a bug in the operating system.

To remove zombies from a system, the SIGCHLD signal can be sent to the parent manually, using the kill command. If the parent process still refuses to reap the zombie, and if it would be fine to terminate the parent process, the next step can be to remove the parent process. When a process loses its parent, init becomes its new parent. "***init***" periodically executes the wait system call to reap any zombies with init as parent.

ORPHAN PROCESS:

An orphan process is a computer process whose parent process has finished or terminated, though it remains running itself. In a Unix-like operating system any orphaned process will be immediately adopted by the special init system process: the kernel sets the parent to ***init***. This operation is called re-parenting and occurs automatically.

Even though technically the process has the "init" process as its parent, it is still called an orphan process since the process that originally created it no longer exists. In other systems orphaned processes are immediately terminated by the kernel. A process can be orphaned unintentionally, such as when the parent process terminates or crashes or a network connection is disconnected.

The process group mechanism in most Unix-like operating systems can be used to help protect against accidental orphaning, where in coordination with the user's shell will try to terminate all the child processes with the "hangup" signal (SIGHUP), rather than letting them continue to run as orphans. More precisely, as part of job control, when the shell exits, because it is the "session leader" (its session id equals its process id), the corresponding login session ends, and the shell sends SIGHUP to all its jobs (internal representation of process groups).

It is sometimes desirable to intentionally orphan a process, usually to allow a long-running job to complete without further user attention, or to start an indefinitely running service or agent; such processes (without an associated session) are known as daemons, particularly if they are indefinitely running.

In multitasking computer operating systems, a daemon is a computer program that runs as a background process, rather than being under the direct control of an interactive user. Traditionally, the process names of a daemon end with the letter d, for clarification that the process is, in fact, a daemon, and for differentiation between a daemon and a normal computer program. For example, syslogd is the daemon that implements the system logging facility.

In a UNIX environment, the parent process of a daemon is often, but not always, the init process. A daemon is usually either created by a process forking a child process and then immediately exiting, thus causing init to adopt the child process, or by the init process directly launching the daemon.

FORK:

In computing, particularly in the context of the UNIX operating system, fork is an operation whereby a process creates a copy of itself. It is usually a system call, implemented in the kernel. Fork is the primary (and historically, only) method of process creation on Unix-like operating systems.

An existing process can create a new one by calling the fork function.

#include <unistd.h>
pid_t fork(void);

Returns: 0 in child, process ID of child in parent, 1 on error

The new process created by fork is called the child process. This function is called once but returns twice. The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child.

The reason the child's process ID is returned to the parent is that a process can have more than one child, and there is no function that allows a process to obtain the process IDs of its children. The reason fork returns 0 to the child is that a process can have only a single parent, and the child can always call ***getppid*** to obtain the process ID of its parent. (Process ID 0 is reserved for use by the kernel, so it's not possible for 0 to be the process ID of a child.)

Both the child and the parent continue executing with the instruction that follows the call to fork. The child is a copy of the parent. For example, the child gets a copy of the parent's data space, heap, and stack. Note that this is a copy for the child; the parent and the child do not share these portions of memory. The parent and the child share the text segment.

Example of fork function:

```
#include <stdio.h>
#include <process.h>
int glob = 6; /* external variable in initialized data */
char buf[] = "a write to stdout\n";
int main(void)
{
int var; /* automatic variable on the stack */
pid_t pid;
var = 88;
if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
err_sys("write error");
printf("before fork\n"); /* we don't flush stdout */
if ((pid = fork()) < 0) {
err_sys("fork error");
} else if (pid == 0) { /* child */
glob++; /* modify variables */
var++;
} else {
```

```
sleep(2); /* parent */
}
printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
exit(0);
}
```

Output:

```
$ ./a.out
a write to stdout
before fork
pid = 430, glob = 7, var = 89 child's variables were changed
pid = 429, glob = 6, var = 88 parent's copy was not changed

$ ./a.out > temp.out
$ cat temp.out
a write to stdout
before fork
pid = 432, glob = 7, var = 89
before fork
pid = 431, glob = 6, var = 88
```


VFORK:

The function vfork has the same calling sequence and same return values as fork. But the semantics of the two functions differ.

However, since vfork() was introduced, the implementation of fork() has improved drastically, most notably with the introduction of `copy-on-write', where the copying of the process address space is transparently faked by allowing both processes to refer to the same physical memory until either of them modify it.

The vfork function is intended to create a new process when the purpose of the new process is to exec a new program. The vfork function creates the new process, just like fork, without copying the address space of the parent into the child, as the child won't reference that address space; the child simply calls exec (or exit) right after the vfork.

Instead, while the child is running and until it calls either exec or exit, the child runs in the address space of the parent. This optimization provides an efficiency gain on some paged virtual-memory implementations of the UNIX System.

Another difference between the two functions is that vfork guarantees that the child runs first, until the child calls exec or exit. When the child calls either of these functions, the parent resumes. (***This can lead to deadlock if the child depends on further actions of the parent before calling either of these two functions.***)

Example of vfork function:

```
int glob = 6; /* external variable in initialized data */
int main(void)
{
int var; /* automatic variable on the stack */
pid_t pid;
var = 88;
printf("before vfork\n"); /* we don't flush stdio */
if ((pid = vfork()) < 0) {
err_sys("vfork error");
} else if (pid == 0) { /* child */
glob++; /* modify parent's variables */
var++;
_exit(0); /* child terminates */
}
/* Parent continues here. */
printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
exit(0);
}
```


Output:

```
$ ./a.out
before vfork
pid = 29039, glob = 7, var = 89
```

EXIT:

There are eight ways for a process to terminate. ***Normal termination*** occurs in 5 ways:

1. Return from main
2. Calling exit
3. Calling _exit or _Exit
4. Return of the last thread from its start routine
5. Calling pthread_exit from the last thread

Abnormal termination occurs in 3 ways:

6. Calling abort
7. Receipt of a signal
8. Response of the last thread to a cancellation request

1. Executing a return from the main function. This is equivalent to calling exit.

2. Calling the exit function.

3. Calling the _exit or _Exit function. _Exit provides a way for a process to terminate without running exit handlers or signal handlers. Whether or not standard I/O streams are flushed depends on the implementation.

    a. On UNIX systems, _Exit and _exit are synonymous and do not flush standard I/O streams. The _exit function is called by exit and handles the UNIX system-specific details.

4. Executing a return from the start routine of the last thread in the process. The return value of the thread is not used as the return value of the process, however. When the last thread returns from its start routine, the process exits with a termination status of 0.

5. Calling the pthread_exit function from the last thread in the process. As with the previous case, the exit status of the process in this situation is always 0, regardless of the argument passed to pthread_exit.

The three forms of abnormal termination are as follows:

6. Calling abort. This is a special case of the next item, as it generates the SIGABRT signal.

7. When the process receives certain signals. The signal can be generated by the process itself for example, by calling the abort function by some other process, or by the kernel. Examples of signals generated by the kernel include the process referencing a memory location not within its address space or trying to divide by 0.

8. The last thread responds to a cancellation request. By default, cancellation occurs in a deferred manner: one thread requests that another be canceled, and sometime later, the target thread terminates.

Regardless of how a process terminates, the same code in the kernel is eventually executed. This kernel code closes all the open descriptors for the process, releases the memory that it was using, and the like.

For any of the preceding cases, we want the terminating process to be able to notify its parent how it terminated. For the three exit functions (exit, _exit, and _Exit), this is done by passing an exit status as the argument to the function.

In the case of an abnormal termination, however, the kernel, not the process, generates a termination status to indicate the reason for the abnormal termination. In any case, the parent of the process can obtain the termination status from either the wait or the waitpid function.

WAIT AND WAITPID:

When a process terminates, either normally or abnormally, the kernel notifies the parent by sending the SIGCHLD signal to the parent. Because the termination of a child is an asynchronous event it can happen at any time while the parent is running this signal is the asynchronous notification from the kernel to the parent.

The parent can choose to ignore this signal, or it can provide a function that is called when the signal occurs: a signal handler. The default action for this signal is to be ignored. For now, we need to be aware that a process that calls wait or waitpid can

- Block, if all of its children are still running
- Return immediately with the termination status of a child, if a child has terminated and is waiting for its termination status to be fetched
- Return immediately with an error, if it doesn't have any child processes

If the process is calling wait because it received the SIGCHLD signal, we expect wait to return immediately. But if we call it at any random point in time, it can block.

Syntax:

```
#include <sys/wait.h>
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
```

Both return: process ID if OK, or 1 on error. The differences between these two functions are as follows:

- The wait function can block the caller until a child process terminates, whereas waitpid has an option that prevents it from blocking.
- The waitpid function doesn't wait for the child that terminates first; it has a number of options that control which process it waits for.

If a child has already terminated and is a zombie, wait returns immediately with that child's status. Otherwise, it blocks the caller until a child terminates. If the caller blocks and has multiple children, wait returns when one terminates. We can always tell which child terminated, because the process ID is returned by the function.

For both functions, the argument *statloc* is a pointer to an integer. If this argument is not a null pointer, the termination status of the terminated process is stored in the location pointed to by the argument. If we don't care about the termination status, we simply pass a null pointer as this argument.

EXEC:

When a process calls one of the exec functions, that process is completely replaced by the new program, and the new program starts executing at its main function. The process ID does not change across an exec, because a new process is not created; exec merely replaces the current process its text, data, heap, and stack segments with a brand new program from disk.

There are six different exec functions, but we'll often simply refer to "the exec function," which means that we could use any of the six functions. These six functions round out the UNIX System process control primitives. With fork, we can create new processes; and with the exec functions, we can initiate new programs. The exit function and the wait functions handle termination and waiting for termination. These are the only process control primitives we need.


```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ... /* (char *)0 */ );
int execv(const char *pathname, char *const argv []);
int execle(const char *pathname, const char *arg0,... /*(char *)0, char *const envp[] */);
int execve(const char *pathname, char *const argv[], char *const envp []);
int execlp(const char *filename, const char *arg0, ... /* (char *)0 */ );
int execvp(const char *filename, char *const argv []);
All six return: 1 on error, no return on success
```

The arguments for these six exec functions are difficult to remember. The letters in the function names help somewhat. The letter **p** means that the function takes a *filename* argument and uses the PATH environment variable to find the executable file. The letter **l** means that the function takes a list of arguments and is mutually exclusive with the letter **v**, which means that it takes an *argv*[] vector. Finally, the letter **e** means that the function takes an *envp*[] array instead of using the current environment.

The first difference in these functions is that the first four take a pathname argument, whereas the last two take a filename argument. When a *filename* argument is specified

- If *filename* contains a slash, it is taken as a pathname.
- Otherwise, the executable file is searched for in the directories specified by the PATH environment variable.

The PATH variable contains a list of directories, called path prefixes that are separated by colons. For example, the *name=value* environment string specifies four directories to search. The last path prefix specifies the current directory. (A zero-length prefix also means the current directory. It can be specified as a colon at the beginning of the *value*, two colons in a row, or a colon at the end of the *value*.)

PATH=/bin:/usr/bin:/usr/local/bin/:.

If either execlp or execvp finds an executable file using one of the path prefixes, but the file isn't a machine executable that was generated by the link editor, the function assumes that the file is a shell script and tries to invoke /bin/sh with the *filename* as input to the shell.

The next difference concerns the passing of the argument list (l stands for list and v stands for vector).

The functions execl, execlp, and execle require each of the command-line arguments to the new program to be specified as separate arguments. We mark the end of the arguments with a null pointer.

For the other three functions (execv, execvp, and execve), we have to build an array of pointers to the arguments, and the address of this array is the argument to these three functions.

The final difference is the passing of the environment list to the new program. The two functions whose names end in an e (execle and execve) allow us to pass a pointer to an array of pointers to the environment strings. The other four functions, however, use the environ variable in the calling process to copy the existing environment for the new program.

We've mentioned that the process ID does not change after an exec, but the new program inherits additional properties from the calling process:

- Process ID and parent process ID
- Real user ID and real group ID
- Supplementary group IDs
- Process group ID
- Session ID
- Controlling terminal
- Time left until alarm clock

- Current working directory
- Root directory
- File mode creation mask
- File locks
- Process signal mask
- Pending signals
- Resource limits
- Values for tms_utime, tms_stime, tms_cutime, and tms_cstime

SIGNALS:

Signals are a technique used to notify a process that some condition has occurred. For example, if a process divides by zero, the signal whose name is SIGFPE (floating-point exception) is sent to the process. The process has three choices for dealing with the signal.

1. Ignore the signal. This option isn't recommended for signals that denote a hardware exception, such as dividing by zero or referencing memory outside the address space of the process, as the results are undefined.

2. Let the default action occur. For a divide-by-zero condition, the default is to terminate the process.

3. Provide a function that is called when the signal occurs (this is called "catching" the signal). By providing a function of our own, we'll know when the signal occurs and we can handle it as we wish.

Many conditions generate signals. Two terminal keys, called the *interrupt key* often the DELETE key or Control-C and the *quit key* often Control-backslash are used to interrupt the currently running process. Another way to generate a signal is by calling the kill function.

We can call this function from a process to send a signal to another process. Naturally, there are limitations: we have to be the owner of the other process (or the superuser) to be able to send it a signal.

SIGNAL FUNCTION:

The simplest interface to the signal features of the UNIX System is the signal function.

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

Returns: previous nature of signal if OK, SIG_ERR on error

The *signo* argument is just the name of the signal (like SIGPWR → power fail/restart, SIGQUIT → terminal quit character, SIGSEGV → invalid memory reference, etc…). The value of *func* is (a) the constant SIG_IGN (telling the system to ignore the signal), (b) the constant SIG_DFL (Default value), or (c) the address of a function to be called when the signal occurs.

UNRELIABLE SIGNALS:

In earlier versions of the UNIX System, signals were unreliable. By this we mean that signals could get lost: a signal could occur and the process would never know about it. Also, a process had little control over a signal: a process could catch the signal or ignore it. Sometimes, we would like to tell the kernel to block a signal: don't ignore it, just remember if it occurs, and tell us later when we're ready.

One problem with these early versions is that the action for a signal was reset to its default each time the signal occurred.

Another problem with these earlier systems is that the process was unable to turn a signal off when it didn't want the signal to occur. All the process could do was ignore the signal. There are times when we would like to tell the system "prevent the following signals from occurring, but remember if they do occur."

INTERRUPTED SYSTEM CALLS

A characteristic of earlier UNIX systems is that if a process caught a signal while the process was blocked in a "slow" system call, the system call was interrupted. The system call returned an error and ***errno*** was set to ***EINTR***. This was done under the assumption that since a signal occurred and the process caught it, there is a good chance that something has happened that should wake up the blocked system call.

Here, we have to differentiate between a system call and a function. It is a system call within the kernel that is interrupted when a signal is caught.

To support this feature, the system calls are divided into two categories: the "slow" system calls and all the others. The slow system calls are those that can block forever. Included in this category are:

- Reads that can block the caller forever if data isn't present with certain file types (pipes, terminal devices, and network devices)
- Writes that can block the caller forever if the data can't be accepted immediately by these same file types
- Opens that block until some condition occurs on certain file types (such as an open of a terminal device that waits until an attached modem answers the phone)

- The pause function (which by definition puts the calling process to sleep until a signal is caught) and the wait function
- Certain ioctl operations
- Some of the interprocess communication functions

The notable exception to these slow system calls is anything related to disk I/O. Although a read or a write of a disk file can block the caller temporarily (while the disk driver queues the request and then the request is executed), unless a hardware error occurs, the I/O operation always returns and unblocks the caller quickly.

One condition that is handled by interrupted system calls, for example, is when a process initiates a read from a terminal device and the user at the terminal walks away from the terminal for an extended period. In this example, the process could be blocked for hours or days and would remain so unless the system was taken down.

KILL and RAISE:

The kill function sends a signal to a process or a group of processes. The raise function allows a process to send a signal to itself.

#include <signal.h>
int kill(pid_t pid, int signo);
int raise(int signo);
Both return: 0 if OK, 1 on error

The call ***raise(signo);*** is equivalent to the call ***kill(getpid(), signo);***

There are four different conditions for the *pid* argument to kill.

PID Value	Description of Kill Based on PID Value
>0	The signal is sent to the process whose process ID is *pid*.
==0	The signal is sent to all processes whose process group ID equals the process group ID of the sender and for which the sender has permission to send the signal.
<0	The signal is sent to all processes whose process group ID equals the absolute value of *pid* and for which the sender has permission to send the signal.
==1	The signal is sent to all processes on the system for which the sender has permission to send the signal.

ALARM and PAUSE:

The alarm function allows us to set a timer that will expire at a specified time in the future. When the timer expires, the ***SIGALRM*** signal is generated. If we ignore or don't catch this signal, its default action is to terminate the process.

#include <unistd.h>
unsigned int alarm(unsigned int seconds);

Returns: 0 or number of seconds until previously set alarm

The *seconds* value is the number of clock seconds in the future when the signal should be generated. Be aware that when that time occurs, the signal is generated by the kernel, but there could be additional time before the process gets control to handle the signal, because of processor scheduling delays.

If, when we call alarm, a previously registered alarm clock for the process has not yet expired, the number of seconds left for that alarm clock is returned as the value of this function. That previously registered alarm clock is replaced by the new value.

If a previously registered alarm clock for the process has not yet expired and if the *seconds* value is 0, the previous alarm clock is canceled. The number of seconds left for that previous alarm clock is still returned as the value of the function.

Although the default action for ***SIGALRM*** is to terminate the process, most processes that use an alarm clock catch this signal. If the process then wants to terminate, it can perform whatever cleanup is required before terminating.

If we intend to catch ***SIGALRM***, we need to be careful to install its signal handler before calling alarm. If we call alarm first and are sent ***SIGALRM*** before we can install the signal handler, our process will terminate.

The pause function suspends the calling process until a signal is caught.

#include <unistd.h>

int pause(void);

Returns: 1 with errno set to ***EINTR.***

The only time pause returns is if a signal handler is executed and that handler returns. In that case, pause returns 1 with errno set to ***EINTR***.

ABORT:

The abort function causes abnormal program termination. This function sends the ***SIGABRT*** signal to the caller.

```
#include <stdlib.h>

void abort(void);
```

This function never returns.

This function sends the SIGABRT signal to the caller.

SYSTEM:

It is convenient to execute a command string from within a program. For example, assume that we want to put a time-and-date stamp into a certain file.

```
#include <stdlib.h>

int system(const char *cmdstring);
```


If *cmdstring* is a null pointer, system returns nonzero only if a command processor is available. This feature determines whether the system function is supported on a given operating system. Under the UNIX System, system is always available.

Returns:

Because system is implemented by calling fork, exec, and waitpid, there are three types of return values.

1. If either the fork fails or waitpid returns an error other than ***EINTR***, system returns 1 with errno set to indicate the error.
2. If the exec fails, implying that the shell can't be executed, the return value is as if the shell had executed exit(127).
3. Otherwise, all three functions fork, exec, and waitpid succeed, and the return value from system is the termination status of the shell, in the format specified for waitpid.

SLEEP:

This function causes the calling process to be suspended until either

1. The amount of wall clock time specified by *seconds* has elapsed.

2. A signal is caught by the process and the signal handler returns.

#include <unistd.h>

unsigned int sleep(unsigned int seconds);

Returns: 0 or number of unslept seconds

As with an alarm signal, the actual return may be at a time later than requested, because of other system activity. In case 1, the return value is 0. When sleep returns early, because of some signal being caught (case 2), the return value is the number of unslept seconds (the requested time minus the actual time slept).

JOB CONTROL SIGNALS:

Job control is a feature added to BSD around 1980. This feature allows us to start multiple jobs (groups of processes) from a single terminal and to control which jobs can access the terminal and which jobs are to run in the background. Job control requires three forms of support:

1. A shell that supports job control
2. The terminal driver in the kernel must support job control
3. The kernel must support certain job-control signals

A job is simply a collection of processes, often a pipeline of processes.

For example, ***vi main.c*** starts a job consisting of one process in the foreground.

The commands

pr *.c | lpr &

ls all &

start two jobs in the background. All the processes invoked by these background jobs are in the background.

Some of the job control signals are:

- **SIGCHLD** Child process has stopped or terminated.
- **SIGCONT** Continue process, if stopped.
- **SIGSTOP** Stop signal (can't be caught or ignored).
- **SIGTSTP** Interactive stop signal.
- **SIGTTIN** Read from controlling terminal by member of a background process group.
- **SIGTTOU** Write to controlling terminal by member of a background process group.

Except for SIGCHLD, most application programs don't handle these signals: interactive shells usually do all the work required to handle these signals.

When we type the suspend character (usually Control-Z), SIGTSTP is sent to all processes in the foreground process group. When we tell the shell to resume a job in the foreground or background, the shell sends all the processes in the job the SIGCONT signal.

Similarly, if SIGTTIN or SIGTTOU is delivered to a process, the process is stopped by default, and the job-control shell recognizes this and notifies us.

DATA MANAGEMENT:

Data management comprises all the disciplines related to managing data as a valuable resource. Data management refers to several levels of managing data like managing your resource allocation, then of dealing with files that are accessed by many users more or less simultaneously and lastly at one tool provided in most UNIX systems for overcoming the limitations of data files.

We can summarize these in three different ways for managing data:

- ✓ Dynamic memory management: what to do and what UNIX won't let you do.
- ✓ File locking: cooperative locking, locking regions of shared files and avoiding deadlocks.
- ✓ The dbm database: a database library featured in UNIX.

MANAGING MEMORY:

On all computer systems memory is a scarce resource. No matter how much memory is available, it never seems to be enough. It wasn't so long ago that being able to address even a single megabyte of memory was considered more than anyone would ever need, but now sixty four times that is considered a bare minimum for a single−user personal computer, and many systems have much more.

From the earliest versions of the operating system, UNIX has had a very clean approach to managing memory. ***UNIX applications are never permitted to access physical memory directly.***

UNIX has always provided processes with its own memory area. Almost all versions of UNIX also provide memory protection, which guarantees that incorrect (or malicious) programs don't overwrite the memory of other processes or the operating system.

In general, the memory allocated to one process can be neither read nor written to by any other process. Almost all versions of UNIX use hardware facilities to enforce this private use of memory.

SIMPLE MEMORY ALLOCATION:

MALLOC:

We allocate memory using the malloc call in the standard C library. It allocates a specified number of bytes of memory. The initial value of the memory is undefined.

Syntax:

#include <stdlib.h>
void *malloc(size_t size);

Return: non-null pointer if OK, NULL on error

Example:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#define A_MEGABYTE (1024 * 1024)
int main()
{
char *some_memory;
int megabyte = A_MEGABYTE;
int exit_code = EXIT_FAILURE;
some_memory = (char *)malloc(megabyte);
if (some_memory != NULL) {
sprintf(some_memory, "Hello World\n");
printf("%s", some_memory);
exit_code = EXIT_SUCCESS;
}
exit(exit_code);
}
```


When we run this program, it outputs:

$ **memory1**
Hello World

How It Works: This program asks the malloc library to give it a pointer to a megabyte of memory. We check to ensure that malloc was successful and then use some of the memory to show that it exists.

When you run the program, you should see Hello World printed out, showing that malloc did indeed return the megabyte of usable memory. We don't check that all of the megabyte is present; we have to put some trust in the malloc code!

CALLOC:

The calloc allocates space for a specified number of objects of a specified size. The space is initialized to all 0 bits.

Syntax:

#include <stdlib.h>
void *calloc(size_t number_of_elements, size_t element_size);

Although calloc allocates memory that can be freed with free, it has rather different parameters. It allocates memory for an array of structures and requires the number of elements and the size of each element as its parameters. The allocated memory is filled with zeros and if calloc is successful, a pointer to the first element is returned.

Like malloc, subsequent calls are not guaranteed to return contiguous space, so you can't enlarge an array created by calloc by simply calling calloc again and expecting the second call to return memory appended to that returned by the first call.

REALLOC:

The realloc increases or decreases the size of a previously allocated area. When the size increases, it may involve moving the previously allocated area somewhere else, to provide the additional room at the end. Also, when the size increases, the initial value of the space between the old contents and the end of the new area is indeterminate.

Syntax:

#include <**stdlib.h**>
void *realloc(void *existing_memory, size_t new_size);

The realloc function changes the size of a block of memory that has been previously allocated. It's passed a pointer to some memory previously allocated by malloc, calloc or realloc and resizes it up or down as requested.

The realloc function may have to move data about to achieve this, so it's important to ensure that once memory has been realloced, you always use the new pointer and never try to access the memory using pointers set up before realloc was called.

Another problem to watch out for is that realloc returns a null pointer if it was unable to resize the memory. This means that, in some applications, you should avoid writing code like this:

my_ptr = malloc(BLOCK_SIZE);
....
my_ptr = realloc(my_ptr, BLOCK_SIZE * 10);

If realloc fails, it will return a null pointer, my_ptr will point to null and the original memory allocated with malloc can no longer be accessed via my_ptr.

It may, therefore, be to your advantage to request the new memory first with malloc and then copy data from the old block to the new block using memcpy before freeing the old block. On error, this would allow the application to retain access to the data stored in the original block of memory, perhaps while arranging a clean termination of the program.

FREE:

Up to now, we've been simply allocating memory and then hoping that when the program ends, the memory we've used hasn't been lost. Fortunately, the UNIX memory management system is quite capable of ensuring that memory is returned to the system when a program ends. However, most programs don't simply want to allocate some memory, use it for a short period, then exit. A much more common use is dynamically using memory as required.

Programs that use memory on a dynamic basis should always release unused memory back to the malloc memory manager using the free call. This allows separate blocks to be remerged and allows the malloc library to look after memory, rather than the application managing it.

If a running program (process) uses and then frees memory, that free memory remains allocated to the process. However, if it's not being used, the UNIX memory manager will be able to page it out from physical memory to swap space, where it has little impact on the use of resources.

Syntax:

#include <stdlib.h>
void free(void *ptr_to memory);

A call to free should only be made with a pointer to memory allocated by a call to malloc, calloc or realloc.

Example:

```
#include <stdlib.h>
#define ONE_K (1024)
int main()
{
char *some_memory;
int exit_code = EXIT_FAILURE;
some_memory = (char *)malloc(ONE_K);
if (some_memory != NULL) {
free(some_memory);
exit_code = EXIT_SUCCESS;
}
exit(exit_code);
}
```

How It Works:

This program simply shows how to call free with a pointer to some previously allocated memory. Important to remember that once you've called free on a block of memory, it no longer belongs to the process. It's not being managed by the malloc library. Never try to read or write memory after calling free on it.

FILE LOCKING:

File locking is a very important part of multiuser, multitasking operating systems. Programs frequently need to share data, usually through files, and it's very important that those programs have some way of establishing control of a file. The file can then be updated in a safe fashion, or a second program can stop itself from trying to read a file that is in a transient state whilst another program is writing to it.

UNIX has several features that we can use for file locking. The simplest method is a technique to create lock files in an atomic way, so that nothing else can happen while the lock is being created. This gives a program a method of creating files that are guaranteed to be unique and could not have been simultaneously created by a different program.

The second method is more advanced and allows programs to lock parts of a file for exclusive access.

CREATING LOCK FILES:

Many applications just need to be able to create a lock file for a resource. Other programs can then check the file to see whether they are permitted to access the resource. Usually, these lock files are in a special place with a name that relates to the resource being controlled.

For example, when a modem is in use, Linux creates a lock file in the /usr/spool/uucp directory. On many UNIX systems, this directory is used to indicate the availability of serial ports:

$ **ls /usr/spool/uucp**
LCK..ttyS1

Remember that lock files act only as indicators; programs need to cooperate to use them. They are termed **advisory**, as opposed to mandatory, locks. To create a file to use as a lock indicator, we use the open system call defined in fcntl.h with the O_CREAT and O_EXCL flags set. This allows us to both checks that the file doesn't already exist and then create it in a single, atomic, operation.

Example:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
int main()
{
int file_desc;
int save_errno;
file_desc = open("/tmp/LCK.test", O_RDWR | O_CREAT | O_EXCL, 0444);
if (file_desc == −1) {
save_errno = errno;
printf("Open failed with error %d\n", save_errno);
}
else {
printf("Open succeeded\n");
}
exit(EXIT_SUCCESS);
}
```

The first time we run the program, the output is,

$ **lock1**
Open succeeded

But the next time we try, we get:

$ **lock1**
Open failed with error 17

How it works:

The program calls open to create a file called /tmp/LCK.test, using the O_CREAT and O_EXCL flags. The first time we run the program, the file didn't exist, so the open call was successful. Subsequent invocations of the program fail, because the file already exists. To get the program to succeed again we'll have to remove the lock file.

On Linux systems at least, error 17 refers to EEXIST, an error that is used to indicate that a file already exists. Error numbers are defined in the header file errno.h or files included by it. In this case, the definition reads:

#define EEXIST 17 /* File exists */

This is an appropriate error for an open(O_CREAT | O_EXCL) failure.

If a program simply needs a resource exclusively for a short period of its execution, often termed a **critical section**, it should create the lock file before entering the critical section and use unlink to delete it afterwards, when it exits the critical section.

LOCKING REGIONS:

Creating lock files is fine for controlling exclusive access to resources such as serial ports, but isn't so good for access to large shared files. Suppose there exists a large file which is written by one program, but updated by many different programs simultaneously.

This might occur if a program is logging some data that is obtained continuously over a long period and is being processed by several other programs. The processing programs can't wait for the logging program to finish–it runs continuously–so they need some way of cooperating to provide simultaneous access to the same file.

We can do this by locking regions of the file, so that a particular section of the file is locked, but other programs may access other parts of the file. UNIX has (at least) two ways to do this (locking region): using the ***fcntl*** system call or the ***lockf*** call.

#include<fcntl.h>

int fcntl(int fildes, int command, ...);

fcntl operates on open file descriptors and, depending on the command parameter, can perform different tasks. The three that we're interested in for file locking are:

- F_GETLK
- F_SETLK
- F_SETLKW → if a conflicting lock is held on the file, then wait for that lock to be released.

When we use these, the third argument must be a pointer to a struct flock, so the prototype is effectively:

int fcntl(int fildes, int command, struct flock *flock_structure);

The flock (file lock) structure is implementation−dependent, but will contain at least the following members:

- short l_type;
- short l_whence;
- off_t l_start;
- off_t l_len;
- pid_t l_pid;

The l_type member takes one of several values, also defined in fcntl.h these are:

Value	Description
F_RDLCK	A shared (or 'read') lock. Many different processes can have a shared lock on the same (or overlapping) regions of the file. If any process has a shared lock then, no process will be able to get an exclusive lock on that region. In order to obtain a shared lock, the file must have been opened with read or read/write access
F_UNLCK	Unlock. Used for clearing locks
F_WRLCK	An exclusive (or 'write') lock. Only a single process may have an exclusive lock on any particular region of a file. Once a process has such a lock, no other process will be able to get any sort of lock on the region. To obtain an exclusive lock, the file must have been opened with write or read/write access

The l_whence, l_start and l_len members define a region, a contiguous set of bytes, in a file. The l_whence must be one of SEEK_SET, SEEK_CUR, SEEK_END (from unistd.h), which correspond to the start, current position and end of a file, respectively. It defines the offset to which l_start, the first byte in the region, is relative.

Normally, this would be SEEK_SET, so l_start is counted from the beginning of the file. The l_len parameter defines the number of bytes in the region. The l_pid parameter is used for reporting the process holding a lock.

Each byte in a file can have only a single type of lock on it at any one time and may be locked for shared access, locked for exclusive access or unlocked.

The F_GETLK Command:

The first command is F_GETLK. This gets locking information about the file that fildes (the first parameter) has open. It doesn't attempt to lock the file. The calling process passes information about the type of lock it might wish to create and fcntl used with the F_GETLK command returns any information that would prevent the lock occurring.

The values used in the flock structure are:

VALUE	DESCRIPTION
l_type	Either F_RDLCK for a shared (read−only) lock or F_WRLCK for an exclusive (write) lock
l_whence	One of SEEK_SET, SEEK_CUR, SEEK_END LCK
l_len	The number of bytes in the file region of interest
l_start	The start byte of the file region of interest
l_pid	The identifier of the process with the lock

A process may use the F_GETLK call to determine the current state of locks on a region of a file. It should set up the flock structure to indicate the type of lock it may require and define the region it's interested in.

The fcntl call returns a value other than −1 if it's successful. If the file already has locks that would prevent a lock request succeeding, it overwrites the flock structure with the relevant information. If the lock would succeed, the flock structure is unchanged. If the F_GETLK call is unable to obtain the information it returns −1 to indicate failure.

If the F_GETLK call is successful (i.e. it returns a value other than −1), the calling application must check the contents of the flock structure to determine whether it was modified. Since the l_pid value is set to the locking process (if one was found), this is a convenient field to check to determine if the flock structure has been changed.

The F_SETLK Command:

This command attempts to lock or unlock part of the file referenced by fildes. The values used in the flock structure (and different from those used by F_GETLK) are:

VALUE	DESCRIPTION
l_type	Either F_RDLCK for a read−only or shared lock; F_WRLCK for an l_type. Either F_RDLCK for a read only, or shared, lock; F_WRLCK for an exclusive, or write, exclusive or write lock; and F_UNLCK to unlock a region
l_pid	Unused

If the lock is successful, fcntl returns a value other than −1; on failure −1 is returned. The function will always return immediately.

The F_SETLKW Command:

This is the same as the F_SETLK command above, except that if it can't obtain the lock, the call will wait until it can. Once this call has started waiting, it will only return when the lock can be obtained or a signal occurs.

USE OF READ AND WRITE WITH LOCKING

When you're using locking on regions of a file, it's very important to use the lower level read and write calls to access the data in the file, rather than the higher level fread and fwrite. This is because fread and fwrite perform buffering of data read or written inside the library, so executing an fread call to read the first 100 bytes of a file may (in fact, almost certainly will) read more than 100 bytes and buffer the additional data inside the library.

If the program then uses fread to read the next 100 bytes, it will actually read data already buffered inside the library and not cause a low level read to pull more data from the file.

To see why this is a problem, consider two programs that wish to update the same file. Let's suppose the file consists of 200 bytes of data, all zeros. The first program starts first and obtains a write lock on the first 100 bytes of the file.

It then uses fread to read in those 100 bytes. However fread will read ahead by up to BUFSIZ bytes at a time, so it actually reads the entire file into memory, but only passes the first 100 bytes back to the program.

The second program then starts. It obtains a write lock on the second 100 bytes of the program. This is successful, since the first program only locked the first 100 bytes. The second program writes twos to bytes 100 to 199, closes the file, and unlocks it and exits.

The first program then locks the second 100 bytes of the file and calls fread to read them in. Because that data was buffered, what the program actually sees is 100 bytes of zeros, not the 100 twos that actually exist in the file. This problem doesn't occur when we're using read and write.

To try out locking, we need two programs, one to do the locking and one to test. The first program does the locking.

Try It Out – Locking a File with fcntl:

1. **Start with the includes and variable declarations: (lock3.c)**

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
const char *test_file = "/tmp/test_lock";
int main()
{
int file_desc;
int byte_count;
char *byte_to_write = "A";
struct flock region_1;
struct flock region_2;
int res;
```

2. **Open a file descriptor:**

```
file_desc = open(test_file, O_RDWR | O_CREAT, 0666);
if (!file_desc) {
fprintf(stderr, "Unable to open %s for read/write\n", test_file);
exit(EXIT_FAILURE);
}
```

3. **Put some data in the file:**

```
for(byte_count = 0; byte_count < 100; byte_count++) {
(void)write(file_desc, byte_to_write, 1);       }
```

4. **Set up region 1 with a shared lock, from bytes 10 to 30:**

```
region_1.l_type = F_RDLCK;
region_1.l_whence = SEEK_SET;
region_1.l_start = 10;
region_1.l_len = 20;
```

5. **Set up region 2 with an exclusive lock, from bytes 40 to 50:**

```
region_2.l_type = F_WRLCK;
region_2.l_whence = SEEK_SET;
region_2.l_start = 40;
```

```
region_2.l_len = 10;
```

6. Now lock the file...

```
printf("Process %d locking file\n", getpid());
res = fcntl(file_desc, F_SETLK, &region_1);
if (res == −1) fprintf(stderr, "Failed to lock region 1\n");
res = fcntl(file_desc, F_SETLK, &region_2);
if (res == −1) fprintf(stderr, "Failed to lock region 2\n");
```

7. ...and wait for a while.

```
sleep(60);
printf("Process %d closing file\n", getpid());
close(file_desc);
exit(EXIT_SUCCESS);
}
```

Try It Out – Testing Locks on a File: (lock4.c)

Let's write a program that tests the different sorts of lock that we could need on different regions of a file.

1. As usual, begin with the includes and declarations:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
const char *test_file = "/tmp/test_lock";
#define SIZE_TO_TRY 5
void show_lock_info(struct flock *to_show);
int main()
{
int file_desc;
int res;
struct flock region_to_test;
int start_byte;
```

2. Open a file descriptor:

```
file_desc = open(test_file, O_RDWR | O_CREAT, 0666);
if (!file_desc) {
fprintf(stderr, "Unable to open %s for read/write", test_file);
exit(EXIT_FAILURE);
```

```
}
for (start_byte = 0; start_byte < 99; start_byte += SIZE_TO_TRY) {
```

3. Set up the region we wish to test:

```
region_to_test.l_type = F_WRLCK;
region_to_test.l_whence = SEEK_SET;
region_to_test.l_start = start_byte;
region_to_test.l_len = SIZE_TO_TRY;
region_to_test.l_pid = −1;
printf("Testing F_WRLCK on region from %d to %d\n", start_byte, start_byte +
SIZE_TO_TRY);
```

4. Now test the lock on the file:

```
res = fcntl(file_desc, F_GETLK, &region_to_test);
if (res == −1) {
fprintf(stderr, "F_GETLK failed\n");
exit(EXIT_FAILURE);
}
if (region_to_test.l_pid != −1) {
printf("Lock would fail. F_GETLK returned:\n");
show_lock_info(&region_to_test);
}
else {
printf("F_WRLCK − Lock would succeed\n");
}
```

5. Now repeat the test with a shared (read) lock. Set up the region we wish to test again:

```
region_to_test.l_type = F_RDLCK;
region_to_test.l_whence = SEEK_SET;
region_to_test.l_start = start_byte;
region_to_test.l_len = SIZE_TO_TRY;
region_to_test.l_pid = −1;
printf("Testing F_RDLCK on region from %d to %d\n",
start_byte, start_byte + SIZE_TO_TRY);
```

6. Test the lock on the file again:

```
res = fcntl(file_desc, F_GETLK, &region_to_test);
if (res == −1) {
fprintf(stderr, "F_GETLK failed\n");
```

```
exit(EXIT_FAILURE);
}
if (region_to_test.l_pid != −1) {
printf("Lock would fail. F_GETLK returned:\n");
show_lock_info(&region_to_test);
}
else {
printf("F_RDLCK − Lock would succeed\n");
}
}
close(file_desc);
exit(EXIT_SUCCESS);
}
void show_lock_info(struct flock *to_show) {
printf("\tl_type %d, ", to_show−>l_type);
printf("l_whence %d, ", to_show−>l_whence);
printf("l_start %d, ", (int)to_show−>l_start);
printf("l_len %d, ", (int)to_show−>l_len);
printf("l_pid %d\n", to_show−>l_pid);
}
```


To test out locking, we first need to run the lock3 program, then run the lock4 program to test the locked file. We do this by executing the lock3 program in the background, with the command:

```
$ lock3 &
$ process 1534 locking file
```

The command prompt returns, since lock3 is running in the background and we then immediately run the lock4 program with the command:

```
$ lock4
```

The output we get, with some omissions for shortness, is:

```
Testing F_WRLOCK on region from 0 to 5
F_WRLCK − Lock would succeed
Testing F_RDLOCK on region from 0 to 5
F_RDLCK − Lock would succeed
...
Testing F_WRLOCK on region from 10 to 15
```

```
Lock would fail. F_GETLK returned:
l_type 0, l_whence 0, l_start 10, l_len 20, l_pid 1534
Testing F_RDLOCK on region from 10 to 15
F_RDLCK − Lock would succeed
Testing F_WRLOCK on region from 15 to 20
Lock would fail. F_GETLK returned:
l_type 0, l_whence 0, l_start 10, l_len 20, l_pid 1534
Testing F_RDLOCK on region from 15 to 20
F_RDLCK − Lock would succeed
...
Testing F_WRLOCK on region from 25 to 30
Lock would fail. F_GETLK returned:
l_type 0, l_whence 0, l_start 10, l_len 20, l_pid 1534
Testing F_RDLOCK on region from 25 to 30
F_RDLCK − Lock would succeed
...
Testing F_WRLOCK on region from 40 to 45
Lock would fail. F_GETLK returned:
l_type 1, l_whence 0, l_start 40, l_len 10, l_pid 1534
Testing F_RDLOCK on region from 40 to 45
Lock would fail. F_GETLK returned:
l_type 1, l_whence 0, l_start 40, l_len 10, l_pid 1534
...
Testing F_RDLOCK on region from 95 to 100
F_RDLCK − Lock would succeed
```


COMPETING LOCKS:

Now that we've seen how to test for existing locks on a file, let's see what happens when two programs compete for locks on the same section of the file. We'll use our lock3 program for locking the file in the first place and a new program to try to lock it again. To complete the example, we'll also add some calls for unlocking.

Here's a program, lock5.c, that tries to lock regions of a file that are already locked, rather than testing the lock status of different parts of the file.

Try It Out – Competing Locks

1. After the #includes and declarations, open a file descriptor:

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

```
#include <stdio.h>
#include <fcntl.h>
const char *test_file = "/tmp/test_lock";
int main()
{
int file_desc;
struct flock region_to_lock;
int res;
file_desc = open(test_file, O_RDWR | O_CREAT, 0666);
if (!file_desc) {
fprintf(stderr, "Unable to open %s for read/write\n", test_file);
exit(EXIT_FAILURE);
}
```

2. **The remainder of the program is spent specifying different regions of the file and trying different locking operations on them:**

```
region_to_lock.l_type = F_RDLCK;
region_to_lock.l_whence = SEEK_SET;
region_to_lock.l_start = 10;
region_to_lock.l_len = 5;
printf("Process %d, trying F_RDLCK, region %d to %d\n", getpid(),
(int)region_to_lock.l_start, (int)(region_to_lock.l_start + region_to_lock.l_len));
res = fcntl(file_desc, F_SETLK, &region_to_lock);
if (res == −1) {
printf("Process %d − failed to lock region\n", getpid());
} else {
printf("Process %d − obtained lock region\n", getpid());
}
region_to_lock.l_type = F_UNLCK;
region_to_lock.l_whence = SEEK_SET;
region_to_lock.l_start = 10;
region_to_lock.l_len = 5;
printf("Process %d, trying F_UNLCK, region %d to %d\n", getpid(),
(int)region_to_lock.l_start, (int)(region_to_lock.l_start + res = fcntl(file_desc, F_SETLK,
&region_to_lock);
if (res == −1) {
printf("Process %d − failed to unlock region\n", getpid());
} else {
printf("Process %d − unlocked region\n", getpid());
```

```
}
region_to_lock.l_type = F_UNLCK;
region_to_lock.l_whence = SEEK_SET;
region_to_lock.l_start = 0;
region_to_lock.l_len = 50;
printf("Process %d, trying F_UNLCK, region %d to %d\n", getpid(),
(int)region_to_lock.l_start, (int)(region_to_lock.l_start + res = fcntl(file_desc, F_SETLK,
&region_to_lock);
if (res == −1) {
printf("Process %d − failed to unlock region\n", getpid());
} else {
printf("Process %d − unlocked region\n", getpid());
}
region_to_lock.l_type = F_WRLCK;
region_to_lock.l_whence = SEEK_SET;
region_to_lock.l_start = 16;
region_to_lock.l_len = 5;
printf("Process %d, trying F_WRLCK, region %d to %d\n", getpid(),
(int)region_to_lock.l_start, (int)(region_to_lock.l_start + res = fcntl(file_desc, F_SETLK,
&region_to_lock);
if (res == −1) {
printf("Process %d − failed to lock region\n", getpid());
} else {
printf("Process %d − obtained lock on region\n", getpid());
}
region_to_lock.l_type = F_RDLCK;
region_to_lock.l_whence = SEEK_SET;
region_to_lock.l_start = 40;
region_to_lock.l_len = 10;
printf("Process %d, trying F_RDLCK, region %d to %d\n", getpid(),
(int)region_to_lock.l_start, (int)(region_to_lock.l_start + res = fcntl(file_desc, F_SETLK,
&region_to_lock);
if (res == −1) {
printf("Process %d − failed to lock region\n", getpid());
} else {
printf("Process %d − obtained lock on region\n", getpid());
}
region_to_lock.l_type = F_WRLCK;
region_to_lock.l_whence = SEEK_SET;
```

```
region_to_lock.l_start = 16;
region_to_lock.l_len = 5;
printf("Process %d, trying F_WRLCK with wait, region %d to %d\n", getpid(),
(int)region_to_lock.l_start, (int)(region_to_lock.l_start + res = fcntl(file_desc, F_SETLKW,
&region_to_lock);
if (res == −1) {
printf("Process %d − failed to lock region\n", getpid());
} else {
printf("Process %d − obtained lock on region\n", getpid());
}
printf("Process %d ending\n", getpid());
close(file_desc);
exit(EXIT_SUCCESS);
}
```

If we first run our lock3 program in the background, then immediately run this new program, the output we get is:

```
Process 227 locking file
Process 228, trying F_RDLCK, region 10 to 15
Process 228 − obtained lock on region
Process 228, trying F_UNLCK, region 10 to 15
Process 228 − unlocked region
Process 228, trying F_UNLCK, region 0 to 50
Process 228 − unlocked region
Process 228, trying F_WRLCK, region 16 to 21
Process 228 − failed to lock on region
Process 228, trying F_RDLCK, region 40 to 50
Process 228 − failed to lock on region
Process 228, trying F_WRLCK with wait, region 16 to 21
Process 227 closing file
Process 228 − obtained lock on region
Process 228 ending
```


How It Works:

Firstly, the program attempts to lock a region from bytes 10 to 15 with a shared lock. This region is already locked with a shared lock, but simultaneous shared locks are allowed and the lock is successful.

It then unlocks (its own) shared lock on the region, which is also successful. The program then attempts to unlock the first 50 bytes of the file, even though it doesn't have any locks set. This is also successful because, even though this program had no locks in the first place, the final result of the unlock request is that there are no locks held by this program in the first 50 bytes.

Next, the program attempts to lock the region from bytes 16 to 21, with an exclusive lock. This region is also already locked with a shared lock so, this time, the new lock fails, because an exclusive lock could not be created.

After that, the program attempts a shared lock on the region from bytes 40 to 50. This region is already locked with an exclusive lock, so, again, the lock fails.

Finally, the program again attempts to obtain an exclusive lock on the region from bytes 16 to 21, but, this time, it uses the F_SETLCKW command to wait until it can obtain a lock. There is then a long pause in the output, until the lock3 program, which had already locked the region, closes the file, thus releasing all the locks it had acquired. The lock5 program resumes execution, successfully locking the region, before it also exits.

OTHER LOCK COMMANDS:

There is a second method of locking files: the lockf function. This also operates using file descriptors. It has the prototype:

#include <unistd.h>

int lockf(int fildes, int function, off_t size_to_lock);

It can take the following function values:

- F_ULOCK → Unlock
- F_LOCK → Lock exclusively
- F_TLOCK → Test and lock exclusively
- F_TEST → Test for locks by other processes

The size_to_lock parameter is the number of bytes to operate on, from the current offset in the file. lockf has a simpler interface than the fcntl interface, principally because it has rather less functionality and flexibility. To use the function, you must seek to the start of the region you wish to lock, and then call it with the number of bytes to lock.

Like the fcntl method of file locking, all locks are only advisory; they won't actually prevent reading from or writing to the file. It's the responsibility of programs to check for locks.

The effect of mixing fcntl locks and lockf locks is undefined, so you must decide which type of locking you wish to use and stick to it.

ADVISORY VERSUS MANDATORY LOCKING:

Advisory locking requires cooperation from the participating processes. Suppose process "A" acquires a WRITE lock, and it started writing into the file, and process "B", without trying to acquire a lock, it can open the file and write into it. Here process "B" is the non-cooperating process.

If process "B", tries to acquire a lock, then it means this process is co-operating to ensure the "serialization". Advisory locking will work, only if the participating process is cooperative. Advisory locking sometimes also called as "***unenforced***" locking. With an advisory lock system, processes can still read and write from a file while it's locked.

Advisory locking returns a result indicating whether the lock was obtained or not: processes can ignore the result and do the I/O anyway. You cannot use both mandatory and advisory file locking on the same file at the same time. The mode of a file at the time it is opened determines whether locks on a file are treated as mandatory or advisory.

Mandatory locking doesn't require cooperation from the participating processes. Mandatory locking causes the kernel to check every open, read, and write to verify that the calling process isn't violating a lock on the given file.

Mandatory locking causes the kernel to check every open, read, and write to verify that the calling process isn't violating a lock on the file being accessed. Mandatory locking is sometimes called ***enforcement-mode*** locking.

Cautions about Mandatory Locking:

1. Mandatory locking works only for local files. It is not supported when accessing files through NFS.
2. Mandatory locking protects only the segments of a file that are locked. The remainder of the file can be accessed according to normal file permissions.

Advisory locking is more efficient because a record lock check does not have to be performed for every I/O request.

Selecting Advisory or Mandatory Locking:

For mandatory locks, the file must be a regular file with the set-group-ID bit on and the group execute permission off. If either condition fails, all record locks are advisory.

DEADLOCKS:

No discussion of locking would be complete without a mention of the dangers of deadlocks. Suppose two programs wish to update the same file. They both need to update byte one and byte two at the same time.

Program A chooses to update byte two, then byte one. Program B tries to update byte one first, then byte two. Both programs start at the same time. Program A locks byte two and program B locks byte one.

Program A tries for a lock on byte one; since this is already locked by program B, program A waits. Program B tries for a lock on byte two. Since this is locked by program A, it too waits.

This situation, when neither program is able to proceed, is called a **deadlock** or **deadly embrace**. Most commercial databases detect deadlocks and break them; the UNIX kernel doesn't.

Some external intervention, perhaps forcibly terminating one of the programs, is required to sort out the resulting mess.

Programmers must be cautious of this situation. When you have multiple programs waiting for locks, you need to be very careful to consider if a deadlock could occur. In this example it's quite easy to avoid: both programs should simply lock the bytes they require in the same order, or use a larger region to lock.

DEADLOCK HANDLING:

The UNIX locking facilities provide deadlock detection/avoidance. Deadlocks can happen only when the system is about to put a record locking function to sleep. A search is made to determine whether process A will wait for a lock that B holds while B is waiting for a lock that A holds. If a potential deadlock is detected, the locking function fails and sets errno to indicate deadlock. Processes setting locks using F_SETLK do not cause a deadlock because they do not wait when the lock cannot be granted immediately.