Unix Systems Programming In a Nutshell

Unix presents a huge set of interfaces to the systems programmer. However, much of this complexity can be tamed by understanding several fundamental abstractions and models, as well as by knowing where to look for more detail. It is important to note, however, that Unix does not always conform to these abstractions and models. Furthermore, although the Unix interfaces have the appearance of orthogonality, orthogonality is not always maintained. This document attempts to describe the fundamental concepts and to point out commonly used interfaces. Examples are geared to Linux.

The Different Varieties Of Unix

The evolution of Unix has resulted in a number of different contemporary implementations: BSD, System V, Linux, Mach, … Different implementations have slightly different semantics for their basic interfaces, and often provide additional interfaces. For example, Mach-based Unices have always offered kernel threads, while BSD-based Unices generally have not. If you look at code that is intended to be portable across different Unices, you'll see many #ifdefs that specialize what is done to the specific Unix.

Documentation

Man pages are the basic on-line reference documentation. No matter how primitive your terminal is, you can always get man pages. To get a man page for, for example, the open call, type "man open". If there is both a command and an interface by a given name, man will tell you the command. To have it print the interface you need to specify the appropriate "section" of the manual. For example, "man chmod" will tell you about the chmod command, while "man 2 chmod" will tell you about the chmod interface, on which, not surprisingly, the chmod command is based.

The problem with man is that if you don't know the name of what you're looking for, you're in trouble. If you don't know that the interface to delete files is called "unlink", you could try running "man –k delete" or "apropos delete" and then sifting through the many man pages, but that can be somewhat painful.

Code can sometimes provide examples from which you can generalize. For example, if you look at the source of rm, you will soon discover that it uses unlink. Often, code examples, as well as other documentation can be found on the web. Of course, it helps if you use a powerful search engine such as www.google.com.

Rick Stevens's "Advanced Programming In the Unix Environment" is an excellent.

Everything Tries To Be A File

Unix tries very hard to make all sorts of objects look like files. This means two things: that these objects are named as files in the file system, and that they can be accessed using the same interface that is used to access files. For example, /dev/dsp is the name of

the sound card. /dev/dsp can be opened and read (recording sound samples from line-in) and written (playing sound samples to line-out) in the same way as any ordinary file.

Sometimes the file abstraction breaks down, however. For example, it is meaningless to lseek on /dev/dsp since the sound card can't go forward or backward in time. Naming can also break down. For example, pipes are anonymous, being implicitly named by the processes at their endpoints. Another example is sockets, the abstraction for using the network. We cannot very well name a remote socket within the local file system.

Files Try To Be Streams of Bytes

Unix treats all files as streams of bytes. It has no clue about what they mean. For example, it is a convention that ASCII text files use the linefeed to denote the end of a line, but Unix does not treat the linefeed character in any special way. There are only a few exceptions to this rule, mostly involving terminals, a topic that won't be discussed here. ("Terminals", despite the archaic name, are important, however, because all keyboard sessions go through the terminal system. The xterms that you have probably used enumerable times are based on "pseudoterminals.")

Errors in Unix

Generally, Unix interfaces have integer return values. A negative return value indicates an exceptional condition. The global integer variable errno provides more information on the error and the perror function can be used to print a meaningful description of the error. For example,

```
int fd =open("foo",O_RDWR);
if (fd<0) {
      perror("Can't open foo");exit(-1);
}
```

attempts to open the file "foo" for both reading and writing. If the open fails, open will return a negative number and set errno to the appropriate error code. The perror will then print "Can't open foo: [description of error code in errno]".

File I/O in Unix versus Standard I/O in C versus …

This document will next discuss how to do I/O with plain files with the raw Unix interfaces. It important to note that the run-time libraries of most programming languages add a layer above this to simplify application-level programming or to increase performance by sophisticated buffering. As you will see, the raw Unix interface is quite primitive.

Probably the most familiar run-time library is C's Standard I/O library, which provides buffered I/O, simple parsing tools, and other tools. For example, consider how much work you might have to go through to write the following with the raw Unix interface we'll describe below.

```
double x,y;
FILE *in=fopen("input.txt","r"), *out=fopen("output.txt", "w+");
fscanf(in,"%lf %lf", &x, &y) ;
fprintf(out,"%lf", x+y);
fclose(in); fclose(out);
```

C++ I/O streams provide even a higher level of abstraction, as well as static type checking to reduce the chance of errors.

File I/O

In Unix, the basic calls for file I/O are open(), read(), write(), lseek(), and close(). Before a file can be accessed, it must be opened. For example,

```
int fd=open("foo",O_RDWR | O_CREAT);
```

opens the file foo for reading and writing. If foo doesn't exist, it is first created and then opened. Note that "foo" could name a plain file, a fifo, a device, or many other kinds of objects. The same open() call (and the subsequent calls we discuss in this section) can be used. Open() returns a file descriptor, which identifies the open file to the kernel. Objects that do not have names in the file system cannot be opened using open. However, whatever call is made to open them, the result is a file descriptor which can be used with the remainder of the calls described in this section.

An open file has associated with it a file pointer, which denotes the next byte of the file that will be read or written. Generally, after an open, this pointer points to the first byte of the file (offset zero). The read call is used to read bytes starting at the current file pointer. For example, the following attempts to read 10 bytes from the file into a buffer:

```
char buf[10];
int num_read=read(fd,buf,10);
```

There are no guarantees that the read() will succeed, or whether it will actually read 10 bytes. If read() returns a negative value, an error has occurred and more detailed error information is available in errno. If read() returns zero, the end of the file has been reached. If read() returns a positive number less than the requested size of the read (10, here), then it is up to the program to call read() again to read the remaining bytes. The file pointer moves forward by the number of bytes that were actually read.

The write() call operates analogously to the read() call.

By default, both read() and write() are blocking—they will not return until either an error condition occurs, the end of the file is reached, or some number of bytes are read. It is also possible to set a file descriptor to be non-blocking, in which case if a call is about to block, it immediately returns a negative number and sets errno to EWOULDBLOCK.

A call may be interrupted by a signal (see below) in which case a negative number is returned and errno is set to EINTR. Strictly speaking, this is not really an error, but it is the programmer's responsibility to repeat the call.

While the file pointer moves implicitly during read() and write() calls, it is also possible to move it explicitly using the lseek() call. For example, suppose we want to back up to the beginning of the file. We would do this by executing:

```
lseek(fd,0,SEEK_SET);
```

When we are finished with the file descriptor, we must close() it:

```
close(fd);
```

More On File Descriptors

File descriptors are a central concept in Unix. Although some objects may not map into the file system namespace and therefore need to be opened or created using other interfaces, once they have been opened, they are accessed through a file descriptor. Such file descriptors can generally be passed to the same interfaces as file descriptors for actual files. Note that these statements are hedged because there are exceptions—Unix is not entirely orthogonal. There are special calls for certain kinds of file descriptors. For example, UDP sockets are usually used with the sendto() and recvfrom() calls instead of write() and read(). Also, it meaningless to execute lseek() on a socket or a sequential file (a tape, for example).

Every program starts with three open file descriptors, stdin (0) , stdout (1), and stderr (2). File descriptors are inherited by child processes. This forms the basis for IPC using anonymous pipes, as we shall see later. The dup() and dup2() calls are useful for cloning and duplicating file descriptors within a process. The following example makes writes to stdout appear the in file "stdout.log".

```
int fd = open("stdout.log",O_WRONLY);
dup2(fd,fileno(stdout))
```

Unix shells use this kind of idiom to implement redirection.

The fcntl() is used to set attributes of file descriptors. Fcntl() is a good example of how Unix sweeps complexity under the rug. It is really a number of interfaces masquerading as one. The second argument is basically the name of the interface you desired. By making the interface an argument, fnctl is extensible. The Linux version has been extended to 12 calls. Sadly, however, this extensibility also leads to non-standard calls. Here is an example of how to set a file descriptor for non-blocking I/O:

```
fnctl(fd,F_SET_FL,fcntl(fd,F_GETFL) | O_NONBLOCK);
```

File descriptors may refer to objects that have special properties or functions that are specific to the kind of device that they are or on which they reside. For example, a sound card probably allows its sample rate and resolution to be adjusted. The ioctl() call provides a mechanism to make such device-specific calls. In effect, ioctl() allows the device driver and other optional components of the kernel to expose functionality to the application. The acceptable arguments to ioctl() depend on the version of Unix and what device drivers, etc, you have loaded.

Processes

Unix processes are fairly heavyweight entities. Except for the very first process, which is created by the kernel, each new process is be created by cloning an existing process using the fork() call. The forker and forkee have a parent/child relationship. These relationships imply the existence of a tree of processes rooted at the first process, which is normally called init. Each process has a unique id, which can be recovered with the getpid() call, and a unique parent, whose id can be found using the getppid() call.

The fork() call often paired with a version of the exec() call, which replaces the current process image with a new one loaded from disk, and with a version of the wait() call, which waits for a child process to terminate. This common idiom looks like this:

```
int rc=fork();
if (rc<0) {
      perror("fork failed");
} elseif (rc==0) {
      // child process
      execlp("ls","ls",0);
      // only get here if execlp failed
      perror("can't exec ls");
} else {
      // parent process
      waitpid(rc,0,0);
      // child finished, continue
}
```

Threads

Some Unices do not support threads, some support only user-level threads, and some support kernel threads that are comparable to those in Win32. Generally, the pthread interface is used to access thread functionality on Unix. For more information try "man pthread_intro" or "man pthread_create".

Interprocess Communication

Unix has a number of mechanisms for IPC. All Unices support (anonymous) pipes. Most support named pipes. Most that support Berkeley sockets (described in another handout) support Unix domain sockets. Most support shared memory segments.

Anonymous pipes are the most primitive IPC mechanism. A pipe supports a unidirectional flow of data between two file descriptors. Generally, pipes are used for communication between parent and child processes. A process creates a pipe and then forks. This clones the file descriptors that point to the pipe's endpoints—both the parent and the child have a handle to both ends of the pipe. The parent process then closes one end of the pipe while the child closes the other end. Writes on one end can then be read at the other end. Here is an example in which the parent uses a pipe to receive a message from its child (note lack of error checking).

```
int thepipe[2];
pipe(thepipe); // this creates the pipe
if (fork()) {
      // parent process sends data
      close(thepipe[1]); // close writing end of pipe
      read(thepipe[0],buf,len);  // read message from child
} else {
      // child process
      close(thepipe[0]); // close reading end of pipe
      write(thepipe[1],buf,len); // write message to parent
}
```

Named pipes extend the pipe abstraction to allow processes that do not share a parent/child relationship to communicate. Named pipes are created on the file system using mkfifo:

```
$ mkfifo thefifo
```

This command creates a special file "thefifo" which processes can open() for either reading or writing. In either case, the open() call blocks until the corresponding call is made on the other end. This allows two process to rendezvous. For example suppose process one executes:

```
int fd = open("thefifo",O_RDONLY);
```

This open will block until someone opens thefifo for writing. Some time later, process two might come along and execute:

```
int fd = open("thefifo",O_WRONLY);
```

At this point, both process one's and process two's opens will finish. Process two can then send messages to process one by writing to its file descriptor, while process one can receive these messages by reading from its file descriptor. Notice how strongly the file abstraction holds here.

Both named and anonymous pipes are half-duplex. Data can flow only one way through them. For two way communication it is necessary to use a pair of pipes or Unix domain sockets. Unix domain sockets are similar to named pipes in that the channel is named in the file system. However, open is not used. Instead, the Berkeley sockets interface, which is described in a separate handout, is used to acquire a file descriptor. Once the file descriptor has been acquired, it can be handled according to the file abstraction.

Two processes, even if they are parent and child, implicitly share no memory. Shared memory segments can, however, be explicitly created and used using the shmem interface. More information on how to use shared memory segments can be found via "man shmget" and friends.

Synchronization and Signals

The pthread interface supports many forms of thread synchronization primitives, such as semaphores and condition variables. More details on these tools can be found in any operating system textbook and in the pthread man pages.

Synchronization between processes uses different tools. We have already seen how a parent process can wait for a child to terminate. Processes can also synchronize by sending signals. In addition, the kernel synchronizes and communicates with processes by sending signals. To send a signal, the kill() system call is used. For example,

```
kill(getppid(),SIGUSR1)
```

sends the parent process a "SIGUSR1" signal, which is the first "user defined" signal. There are other, pre-defined, signals which are most often sent by the kernel. For example, the kernel sends a SIGSEGV when a process incurs a segfault. In response to a signal, the parent process immediately executes a signal handler. There are default signal handlers for all signals. Generally, these cause the process to terminate. That is precisely what would happen in our example. However, the process can install its own signal handlers, however, using the signal() and sigaction() calls. For example, suppose our process had executed the following before receiving the SIGUSR1.

```
void SignalHandler(int signum) {
      fprinf(stderr,"Hi!  Caught signal %d\n", signum);
}
signal(SIGUSR1,&SignalHandler);
```

In that case, SignalHandler() would have been called instead of the default handler and, instead of terminating, the program would have printed "Hi! Caught signal 10".

A common misconception about signals is that they somehow "queue", so that n calls to kill(getppid(),SIGUSR1) will result in n innovations of SignalHandler, for example. In fact, signals are either on or off. The kill call turns the signal on, while the invocation turns the signal off. If the "first" invocation of SignalHandler is delayed until all n kills have executed, then it will be executed only once. The implication is that the signal handler must be prepared to handle all that may have transpired before it was invoked. This is vitally important in asynchronous I/O, which we will not discuss further here.

Sometimes a process may want to temporarily ignore signals that arrive. For example, it may want to execute some code as a critical section. The sigprocmask() call can be used to "mask" signals. It is important to note that some signals cannot be masked. SIGKILL is one example. It is never possible to prevent your process from being forcibly killed.

Select and Friends

A common problem in Unix systems programming is to wait for one of several events to happen. Events in Unix include: a file descriptor becoming available for reading, writing, or entering an exceptional condition, a certain amount of time passing, and a signal having been handled. Consider file descriptors. Recall that, by default, calls such as read and write are blocking. If you try to read a file descriptor and nothing is available, the read will not return until data is available or an error occurs. Now suppose you want to read from whichever of two file descriptors has data available. Obviously, reading first from one and then from the other will not work—you may block on one file descriptor while there is a data available on the other. One option is to set the file descriptors to non-blocking and then repeatedly attempt the reads until one does not return EWOULDBLOCK. However, this will burn up CPU like crazy—CPU that another process could use. Another option is to use asynchronous I/O, in which you ask Unix to send you a SIGIO signal whenever some file descriptor changes state. This can be painful to code.

A simple alternative is to use the select() call. Select says "wait efficiently until at least one of this set of file descriptors is available for reading, or at least one of this other set of file descriptors is available for writing, or at least one of this still other set of file descriptors has an exceptional condition, or a signal is handled, or a certain amount of time has passed". We could code our example in the following way:

```
fd_set read_fds;
FD_ZERO(&read_fds)
FD_SET(fd1,&read_fds);
FD_SET(fd2,&read_fds);
int rc = select(MAX(fd1,fd2)+1,&read_fds,0,0,0);
if (rc<0) {
      // error happened
```

```
            if (errno==EINTR) {
                  // signal was handled, select interrupted
            }
      } elseif (rc==0) {
            // time out happened
      } else {
            // rc is the number of fds that are avaible
            if (FD_ISSET(fd1,&read_fds) {
                  // there is at least one byte to be read on fd1
            }
            if (FD_ISSET(fd2,&read_fds) {
                  // there is at least one byte to be read on fd2
            }
      }
```

Here read_fds is the set of file descriptors that we want to read from. In this example, the next three arguments to select (the set of file descriptors we want to write to, the set of file descriptors we want to check for exceptional conditions, and the amount of time before a timeout) are set to zero because we are not interested in any of these events and we are willing to wait indefinitely.

There are several other functions that are similar to select, but sometimes easier to use. These include poll(), sleep(), and usleep().

Other Topics Of Interest

This document has avoided talking about the following topics because they are not necessary in this course. They are presented here along with suggestions for man pages to look at to learn more. Stevens's book can also be quite helpful here.

- File system management (creat(), mkdir(), unlink(), stat(), lstat(), …)
- Security (chmod(), chown(), chgrp(), flock())
- Terminals, sessions, and process groups (tty, getty, termios)
- Xwindow system (X)
- Message queues (msgget)
- Semaphores (semget)
- Sockets (socket) (also see socket introduction)