CASE STUDY 1: UNIX AND LINUX

10.1 HISTORY OF UNIX
10.2 OVERVIEW OF UNIX
10.3 PROCESSES IN UNIX
10.4 MEMORY MANAGEMENT IN UNIX
10.5 INPUT/OUTPUT IN UNIX
10.6 THE UNIX FILE SYSTEM
10.7 SECURITY IN UNIX
10.8 SUMMARY
Fig. 10-1. The layers in a UNIX system.
<table>
<thead>
<tr>
<th>Program</th>
<th>Typical use</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>Concatenate multiple files to standard output</td>
</tr>
<tr>
<td>chmod</td>
<td>Change file protection mode</td>
</tr>
<tr>
<td>cp</td>
<td>Copy one or more files</td>
</tr>
<tr>
<td>cut</td>
<td>Cut columns of text from a file</td>
</tr>
<tr>
<td>grep</td>
<td>Search a file for some pattern</td>
</tr>
<tr>
<td>head</td>
<td>Extract the first lines of a file</td>
</tr>
<tr>
<td>ls</td>
<td>List directory</td>
</tr>
<tr>
<td>make</td>
<td>Compile files to build a binary</td>
</tr>
<tr>
<td>mkdir</td>
<td>Make a directory</td>
</tr>
<tr>
<td>od</td>
<td>Octal dump a file</td>
</tr>
<tr>
<td>paste</td>
<td>Paste columns of text into a file</td>
</tr>
<tr>
<td>pr</td>
<td>Format a file for printing</td>
</tr>
<tr>
<td>rm</td>
<td>Remove one or more files</td>
</tr>
<tr>
<td>rmdir</td>
<td>Remove a directory</td>
</tr>
<tr>
<td>sort</td>
<td>Sort a file of lines alphabetically</td>
</tr>
<tr>
<td>tail</td>
<td>Extract the last lines of a file</td>
</tr>
<tr>
<td>tr</td>
<td>Translate between character sets</td>
</tr>
</tbody>
</table>

Fig. 10-2. A few of the common UNIX utility programs required by POSIX.
<table>
<thead>
<tr>
<th>System calls</th>
<th>Interrupts and traps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Terminal handling</td>
<td></td>
</tr>
<tr>
<td>Sockets</td>
<td>File naming</td>
</tr>
<tr>
<td>File mapping</td>
<td>Page faults</td>
</tr>
<tr>
<td>Mapping</td>
<td>Signal handling</td>
</tr>
<tr>
<td>Page faults</td>
<td>Process creation</td>
</tr>
<tr>
<td>File systems</td>
<td>and termination</td>
</tr>
<tr>
<td>Virtual memory</td>
<td></td>
</tr>
<tr>
<td>Buffer cache</td>
<td></td>
</tr>
<tr>
<td>Page cache</td>
<td></td>
</tr>
<tr>
<td>Process</td>
<td></td>
</tr>
<tr>
<td>creation</td>
<td></td>
</tr>
<tr>
<td>and termination</td>
<td></td>
</tr>
<tr>
<td>Hardware</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 10-3. Structure of the 4.4BSD kernel.
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. */
}

Fig. 10-4. Process creation in UNIX.
<table>
<thead>
<tr>
<th>Signal</th>
<th>Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGABRT</td>
<td>Sent to abort a process and force a core dump</td>
</tr>
<tr>
<td>SIGALRM</td>
<td>The alarm clock has gone off</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>A floating-point error has occurred (e.g., division by 0)</td>
</tr>
<tr>
<td>SIGHUP</td>
<td>The phone line the process was using has been hung up</td>
</tr>
<tr>
<td>SIGILL</td>
<td>The user has hit the DEL key to interrupt the process</td>
</tr>
<tr>
<td>SIGQUIT</td>
<td>The user has hit the key requesting a core dump</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>Sent to kill a process (cannot be caught or ignored)</td>
</tr>
<tr>
<td>SIGPIPE</td>
<td>The process has written to a pipe which has no readers</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>The process has referenced an invalid memory address</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>Used to request that a process terminate gracefully</td>
</tr>
<tr>
<td>SIGUSR1</td>
<td>Available for application-defined purposes</td>
</tr>
<tr>
<td>SIGUSR2</td>
<td>Available for application-defined purposes</td>
</tr>
</tbody>
</table>

Fig. 10-5. The signals required by POSIX.
<table>
<thead>
<tr>
<th>System call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid = fork( )</td>
<td>Create a child process identical to the parent</td>
</tr>
<tr>
<td>pid = waitpid(pid, &amp;statloc, opts)</td>
<td>Wait for a child to terminate</td>
</tr>
<tr>
<td>s = execve(name, argv, envp)</td>
<td>Replace a process' core image</td>
</tr>
<tr>
<td>exit(status)</td>
<td>Terminate process execution and return status</td>
</tr>
<tr>
<td>s = sigaction(sig, &amp;act, &amp;oldact)</td>
<td>Define action to take on signals</td>
</tr>
<tr>
<td>s = sigreturn(&amp;context)</td>
<td>Return from a signal</td>
</tr>
<tr>
<td>s = sigprocmask(how, &amp;set, &amp;old)</td>
<td>Examine or change the signal mask</td>
</tr>
<tr>
<td>s = sigpending(set)</td>
<td>Get the set of blocked signals</td>
</tr>
<tr>
<td>s = sigsuspend(sigmask)</td>
<td>Replace the signal mask and suspend the process</td>
</tr>
<tr>
<td>residual = alarm(seconds)</td>
<td>Set the alarm clock</td>
</tr>
<tr>
<td>s = pause( )</td>
<td>Suspend the caller until the next signal</td>
</tr>
</tbody>
</table>

Fig. 10-6. Some system calls relating to processes. The return code $s$ is $-1$ if an error has occurred, $pid$ is a process ID, and residual is the remaining time in the previous alarm. The parameters are what the name suggests.
while (TRUE) {
    /* repeat forever */
    type_prompt(); /* display prompt on the screen */
    read_command(command, params); /* read input line from keyboard */

    pid = fork(); /* fork off a child process */
    if (pid < 0) {
        printf("Unable to fork\0"); /* error condition */
        continue; /* repeat the loop */
    }

    if (pid != 0) {
        waitpid(−1, &status, 0); /* parent waits for child */
    } else {
        execve(command, params, 0); /* child does the work */
    }
}

Fig. 10-7. A highly simplified shell.
<table>
<thead>
<tr>
<th>Thread call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_create</td>
<td>Create a new thread in the caller's address space</td>
</tr>
<tr>
<td>pthread_exit</td>
<td>Terminate the calling thread</td>
</tr>
<tr>
<td>pthread_join</td>
<td>Wait for a thread to terminate</td>
</tr>
<tr>
<td>pthread_mutex_init</td>
<td>Create a new mutex</td>
</tr>
<tr>
<td>pthread_mutex_destroy</td>
<td>Destroy a mutex</td>
</tr>
<tr>
<td>pthread_mutex_lock</td>
<td>Lock a mutex</td>
</tr>
<tr>
<td>pthread_mutex_unlock</td>
<td>Unlock a mutex</td>
</tr>
<tr>
<td>pthread_cond_init</td>
<td>Create a condition variable</td>
</tr>
<tr>
<td>pthread_cond_destroy</td>
<td>Destroy a condition variable</td>
</tr>
<tr>
<td>pthread_cond_wait</td>
<td>Wait on a condition variable</td>
</tr>
<tr>
<td>pthread_cond_signal</td>
<td>Release one thread waiting on a condition variable</td>
</tr>
</tbody>
</table>

Fig. 10-8. The principal POSIX thread calls.
Fig. 10-9. The steps in executing the command `ls` typed to the shell.
<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning when set</th>
<th>Meaning when cleared</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLONE_VM</td>
<td>Create a new thread</td>
<td>Create a new process</td>
</tr>
<tr>
<td>CLONE_FS</td>
<td>Share umask, root, and working dirs</td>
<td>Do not share them</td>
</tr>
<tr>
<td>CLONE.FILES</td>
<td>Share the file descriptors</td>
<td>Copy the file descriptors</td>
</tr>
<tr>
<td>CLONE_SIGHAND</td>
<td>Share the signal handler table</td>
<td>Copy the table</td>
</tr>
<tr>
<td>CLONE_PID</td>
<td>New thread gets old PID</td>
<td>New thread gets own PID</td>
</tr>
</tbody>
</table>

Fig. 10-10. Bits in the *sharing* *flags* bitmap.
Fig. 10-11. The UNIX scheduler is based on a multilevel queue structure.
Fig. 10-12. The sequence of processes used to boot some UNIX systems.
Fig. 10-13. (a) Process A’s virtual address space. (b) Physical memory. (c) Process B’s virtual address space.
Fig. 10-14. Two processes can share a mapped file.
<table>
<thead>
<tr>
<th>System call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s = brk(addr)</td>
<td>Change data segment size</td>
</tr>
<tr>
<td>a = mmap(addr, len, prot, flags, fd, offset)</td>
<td>Map a file in</td>
</tr>
<tr>
<td>s = unmap(addr, len)</td>
<td>Unmap a file</td>
</tr>
</tbody>
</table>

Fig. 10-15. Some system calls relating to memory management. The return code $s$ is $-1$ if an error has occurred; $a$ and $addr$ are memory addresses, $len$ is a length, $prot$ controls protection, $flags$ are miscellaneous bits, $fd$ is a file descriptor, and $offset$ is a file offset.
Fig. 10-16. The core map in 4BSD.
Fig. 10-17. Linux uses three-level page tables.
Fig. 10-18. Operation of the buddy algorithm.
Fig. 10-19. The uses of sockets for networking.
<table>
<thead>
<tr>
<th>Function call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s = cfsetospeed(&amp;termios, speed)</code></td>
<td>Set the output speed</td>
</tr>
<tr>
<td><code>s = cfsetispeed(&amp;termios, speed)</code></td>
<td>Set the input speed</td>
</tr>
<tr>
<td><code>s = cfgetospeed(&amp;termios, speed)</code></td>
<td>Get the output speed</td>
</tr>
<tr>
<td><code>s = cfgetispeed(&amp;termios, speed)</code></td>
<td>Get the input speed</td>
</tr>
<tr>
<td><code>s = tcsetattr(fd, opt, &amp;termios)</code></td>
<td>Set the attributes</td>
</tr>
<tr>
<td><code>s = tcgetattr(fd, &amp;termios)</code></td>
<td>Get the attributes</td>
</tr>
</tbody>
</table>

Fig. 10-20. The main POSIX calls for managing the terminal.
<table>
<thead>
<tr>
<th>Device</th>
<th>Open</th>
<th>Close</th>
<th>Read</th>
<th>Write</th>
<th>Ioctl</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>...</td>
</tr>
<tr>
<td>Memory</td>
<td>null</td>
<td>null</td>
<td>mem_read</td>
<td>mem_write</td>
<td>null</td>
<td>...</td>
</tr>
<tr>
<td>Keyboard</td>
<td>k_open</td>
<td>k_close</td>
<td>k_read</td>
<td>error</td>
<td>k_ioctl</td>
<td>...</td>
</tr>
<tr>
<td>Tty</td>
<td>tty_open</td>
<td>tty_close</td>
<td>tty_read</td>
<td>tty_write</td>
<td>tty_ioctl</td>
<td>...</td>
</tr>
<tr>
<td>Printer</td>
<td>lp_open</td>
<td>lp_close</td>
<td>error</td>
<td>lp_write</td>
<td>lp_ioctl</td>
<td>...</td>
</tr>
</tbody>
</table>

Fig. 10-21. Some of the fields of a typical `cdevsw` table.
Fig. 10-22. The UNIX I/O system in BSD.
Fig. 10-23. An example of streams in System V.
<table>
<thead>
<tr>
<th>Directory</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>bin</td>
<td>Binary (executable) programs</td>
</tr>
<tr>
<td>dev</td>
<td>Special files for I/O devices</td>
</tr>
<tr>
<td>etc</td>
<td>Miscellaneous system files</td>
</tr>
<tr>
<td>lib</td>
<td>Libraries</td>
</tr>
<tr>
<td>usr</td>
<td>User directories</td>
</tr>
</tbody>
</table>

Fig. 10-24. Some important directories found in most UNIX systems.
Fig. 10-25. (a) Before linking. (b) After linking.
Fig. 10-26. (a) Separate file systems. (b) After mounting.
Fig. 10-27. (a) A file with one lock. (b) Addition of a second lock. (c) A third lock.
<table>
<thead>
<tr>
<th>System call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fd = creat(name, mode)</code></td>
<td>One way to create a new file</td>
</tr>
<tr>
<td><code>fd = open(file, how, ...)</code></td>
<td>Open a file for reading, writing or both</td>
</tr>
<tr>
<td><code>s = close(fd)</code></td>
<td>Close an open file</td>
</tr>
<tr>
<td><code>n = read(fd, buffer, nbytes)</code></td>
<td>Read data from a file into a buffer</td>
</tr>
<tr>
<td><code>n = write(fd, buffer, nbytes)</code></td>
<td>Write data from a buffer into a file</td>
</tr>
<tr>
<td><code>position = lseek(fd, offset, whence)</code></td>
<td>Move the file pointer</td>
</tr>
<tr>
<td><code>s = stat(name, &amp;buf)</code></td>
<td>Get a file’s status information</td>
</tr>
<tr>
<td><code>s = fstat(fd, &amp;buf)</code></td>
<td>Get a file’s status information</td>
</tr>
<tr>
<td><code>s = pipe(&amp;fd[0])</code></td>
<td>Create a pipe</td>
</tr>
<tr>
<td><code>s = fcntl(fd, cmd, ...)</code></td>
<td>File locking and other operations</td>
</tr>
</tbody>
</table>

Fig. 10-28. Some system calls relating to files. The return code $s$ is $−1$ if an error has occurred; $fd$ is a file descriptor, and $position$ is a file offset. The parameters should be self explanatory.
<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device the file is on</td>
</tr>
<tr>
<td>I-node number (which file on the device)</td>
</tr>
<tr>
<td>File mode (includes protection information)</td>
</tr>
<tr>
<td>Number of links to the file</td>
</tr>
<tr>
<td>Identity of the file’s owner</td>
</tr>
<tr>
<td>Group the file belongs to</td>
</tr>
<tr>
<td>File size (in bytes)</td>
</tr>
<tr>
<td>Creation time</td>
</tr>
<tr>
<td>Time of last access</td>
</tr>
<tr>
<td>Time of last modification</td>
</tr>
</tbody>
</table>

Fig. 10-29. The fields returned by the stat system call.
<table>
<thead>
<tr>
<th>System call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s = mkdir(path, mode)</td>
<td>Create a new directory</td>
</tr>
<tr>
<td>s = rmdir(path)</td>
<td>Remove a directory</td>
</tr>
<tr>
<td>s = link(oldpath, newpath)</td>
<td>Create a link to an existing file</td>
</tr>
<tr>
<td>s = unlink(path)</td>
<td>Unlink a file</td>
</tr>
<tr>
<td>s = chdir(path)</td>
<td>Change the working directory</td>
</tr>
<tr>
<td>dir = opendir(path)</td>
<td>Open a directory for reading</td>
</tr>
<tr>
<td>s = closedir(dir)</td>
<td>Close a directory</td>
</tr>
<tr>
<td>dirent = readdir(dir)</td>
<td>Read one directory entry</td>
</tr>
<tr>
<td>rewinddir(dir)</td>
<td>Rewind a directory so it can be reread</td>
</tr>
</tbody>
</table>

Fig. 10-30. Some system calls relating to directories. The return code $s$ is $-1$ if an error has occurred; $dir$ identifies a directory stream and $dirent$ is a directory entry. The parameters should be self explanatory.
Fig. 10-31. Disk layout in classical UNIX systems.
<table>
<thead>
<tr>
<th>Field</th>
<th>Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode</td>
<td>2</td>
<td>File type, protection bits, setuid, setgid bits</td>
</tr>
<tr>
<td>Nlinks</td>
<td>2</td>
<td>Number of directory entries pointing to this i-node</td>
</tr>
<tr>
<td>Uid</td>
<td>2</td>
<td>UID of the file owner</td>
</tr>
<tr>
<td>Gid</td>
<td>2</td>
<td>GID of the file owner</td>
</tr>
<tr>
<td>Size</td>
<td>4</td>
<td>File size in bytes</td>
</tr>
<tr>
<td>Addr</td>
<td>39</td>
<td>Address of first 10 disk blocks, then 3 indirect blocks</td>
</tr>
<tr>
<td>Gen</td>
<td>1</td>
<td>Generation number (incremented every time i-node is reused)</td>
</tr>
<tr>
<td>Atime</td>
<td>4</td>
<td>Time the file was last accessed</td>
</tr>
<tr>
<td>Mtime</td>
<td>4</td>
<td>Time the file was last modified</td>
</tr>
<tr>
<td>Ctime</td>
<td>4</td>
<td>Time the i-node was last changed (except the other times)</td>
</tr>
</tbody>
</table>

Fig. 10-32. Structure of the i-node in System V.
Fig. 10-33. The relation between the file descriptor table, the open file description table, and the i-node table.
Fig. 10-34. (a) A BSD directory with three files. (b) The same directory after the file *voluminous* has been removed.
Fig. 10-35. Layout of the Linux Ext2 file system.
Fig. 10-36. Examples of remote mounted file systems. Directories are shown as squares and files are shown as circles.
Fig. 10-37. The NFS layer structure.
<table>
<thead>
<tr>
<th>Binary</th>
<th>Symbolic</th>
<th>Allowed file accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>111000000</td>
<td>rwx------</td>
<td>Owner can read, write, and execute</td>
</tr>
<tr>
<td>111110000</td>
<td>rwrx-----</td>
<td>Owner and group can read, write, and execute</td>
</tr>
<tr>
<td>110100000</td>
<td>rw-r-----</td>
<td>Owner can read and write; group can read</td>
</tr>
<tr>
<td>110100100</td>
<td>rw-r-----r-</td>
<td>Owner can read and write; all others can read</td>
</tr>
<tr>
<td>111101101</td>
<td>rwrx-xr-x</td>
<td>Owner can do everything, rest can read and execute</td>
</tr>
<tr>
<td>000000000</td>
<td>---------x</td>
<td>Nobody has any access</td>
</tr>
<tr>
<td>000000111</td>
<td>---------rw</td>
<td>Only outsiders have access (strange, but legal)</td>
</tr>
</tbody>
</table>

Fig. 10-38. Some example file protection modes.
<table>
<thead>
<tr>
<th>System call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s = chmod(path, mode)</td>
<td>Change a file’s protection mode</td>
</tr>
<tr>
<td>s = access(path, mode)</td>
<td>Check access using the real UID and GID</td>
</tr>
<tr>
<td>uid = getuid( )</td>
<td>Get the real UID</td>
</tr>
<tr>
<td>uid = geteuid( )</td>
<td>Get the effective UID</td>
</tr>
<tr>
<td>gid = getgid( )</td>
<td>Get the real GID</td>
</tr>
<tr>
<td>gid = getegid( )</td>
<td>Get the effective GID</td>
</tr>
<tr>
<td>s = chown(path, owner, group)</td>
<td>Change owner and group</td>
</tr>
<tr>
<td>s = setuid(uid)</td>
<td>Set the UID</td>
</tr>
<tr>
<td>s = setgid(gid)</td>
<td>Set the GID</td>
</tr>
</tbody>
</table>

Fig. 10-39. Some system calls relating to security. The return code $s$ is $−1$ if an error has occurred; $uid$ and $gid$ are the UID and GID, respectively. The parameters should be self explanatory.