Unix Introduction

- Kernel
- Users and Groups
- FileSystem Structure
- FileSystem Structure Continued
- Directories
- Symbolic Links
- Error Handling
- Error Handling Example
- Input and Output
- Input and Output Continued
- Copy Input to Output Using Buffered I/O
- Copy Input to Output Using Unbuffered I/O
- Programs and Processes
- Creating Processes
- Executing Programs
- A Simple Shell
- Processes
- Processes Continued
- Memory Mappings
- Libraries
- Inter Process Communication
- Signals
- Handling Signals
- Signal Handling Functions
- Signal-Handling Shell
- Time Values
- Threads
- A Serial Alarm Clock
- Serial Alarm Execution
- A Forked Alarm Clock
- Forked Alarm Execution
- Threaded Alarm
- Threaded Alarm Continued
- Process Groups and Sessions
- References

Kernel

- *Multi-task* multiple processes, *preemptively* scheduling them.
- Manage all computer resources including CPUs (*scheduling*), memory, disks (*file systems*), devices.
- Kernel accessed via a system call interface or API.
- Kernel mode has access to entire computer, including special instructions.
- User mode has restricted access computer.
- Transition between user mode and kernel mode is expensive.

Users and Groups

- Associated with each user *login name*, there is a unique integer called a *user ID* or `UID`. Most programs refer to users using `UID`s.
- `/etc/passwd` contains line-oriented records specifying the login-name to UID mapping. Additionally, it also contains other information like *home directory*, *login shell* and *primary group*. It used to also contain password hashes, but for security reasons, these are now moved to privileged access *shadow password file*.
- Each user may be in one or more *groups*. Each group has a name and associated *group ID* or `GID`. Groups are usually maintained in the `/etc/group` file.
- Superuser (with UID 0) and normal login name *root* has all permissions.

FileSystem Structure

- Filesystem is a hierarchical arrangement of directories and files.
- The root of the entire hierarchy is denoted by `/`.
- A *filename* can contain any character except `/` and ASCII `NUL` (portable character set of alnum chars + `.`, `-` and _). Most modern systems have no practical limit (Linux is 255) on the length of a filename, though historical systems were limited to 14.

FileSystem Structure Continued

- A filesystem is not restricted to a single device but can span multiple devices from floppy disks to DVD drives. Each device is *mount*ed at some point in the filesystem.
- A *pathname* consists of a sequence of zero or more filenames separated by a `/`. If a pathname begins with a `/`, then it is a *absolute pathname*, otherwise it is a *relative pathname*.
- Relative pathnames are interpreted relative to a current *working directory*. Each process always has a current working directory.
- File types include regular files, devices, directories, pipes, sockets, symbolic links.

Directories

- Directories are containers containing files and other directories.
- Directories are implemented as special files giving mapping from file-names to corresponding files.
- Each mapping is referred to as a *hard link* or simply *link*.
- A file can have multiple mappings referencing it; i.e. it can have multiple names or *hard links*.
- Each directory always has two entries: `.` referencing back to itself, and `..` refering to its parent (`/..` `==` `/`).

Symbolic Links

A *symbolic link* (aka *soft link*) is a special file containg the name of another file.

- When a symbolic link is referenced, most kernel calls automatically recursively dereference the link.
- Limit on length of reference chains to avoid getting getting caught up within circular chains.
- File referenced by symbolic link may not exist: *dangling link*.

Error Handling

- `errno` is a global variable or macro set by system when a error occurs. No system routine resets `errno`; return value of system call indicates that an error occurred and `errno` indicates what the error was.
- `char *strerror(int)` used to get string form of error message.
- `void perror(const char *msg)` used to print out `msg` followed by `:`, error message and newline.

Error Handling Example

Error handling illustrated by following program:

```
int
main(int argc, char *argv[])
{
  fprintf(stderr, "EACCES: %s\n",
          strerror(EACCES));

  errno = ENOENT;
  perror(argv[0]);

  exit(0);
}
```

Produces output:

```
$ ./programs/errors
EACCES: Permission denied
./programs/errors: No such file or directory
$
```

Input and Output

- Can do I/O using C-library's standard I/O. Main advantage is *buffering*.
- Can also do I/O using lower-level system primitives. No buffering is provided, but there may be greater flexibility and efficiency.
- Programatically, a file is referred to using some kind of **opaque** *handle*. C-library I/O uses `FILE` *streams*, whereas system I/O uses integer *file descriptors*.

Input and Output Continued

- Every program has available to it 3 abstract I/O devices. These default to the terminal the program is run under, but can be *redirected*.

  Standard Input
  : Used for input. `stdin` stream for C-library I/O; file descriptor `STDIN_FILENO` (`0`) for system I/O.

  Standard Output
  : Used for output. `stdout` stream for C-library I/O; file descriptor `STDOUT_FILENO` (`1`) for system I/O.

  Standard Error
  : Used for error messages. `stderr` stream for C-library I/O; file descriptor `STDERR_FILENO` (`2`) for system I/O.

- The proper use of *buffering* is critical to the I/O performance of a program.

Copy Input to Output Using Buffered I/O

Using the routines in the standard C-library, we can copy the input of a program to its output using the following program:

```
int
main(void)
{
  int c;

  while ( (c = getc(stdin)) != EOF) {
    if (putc(c, stdout) == EOF) {
      fprintf(stderr, "output error: %s\n", strerror(errno));
      exit(1);
    }
  }

  if (ferror(stdin)) {
    fprintf(stderr, "input error: %s\n", strerror(errno));
    exit(1);
  }

  exit(0);
}
```

Copy Input to Output Using Unbuffered I/O

Using the routines in the Unix library, we can copy the input of a program to its output using the following program:

```
#define BUFFSIZE 8192

int
main(void)
{
  int n;
  char buf[BUFFSIZE];

  while ( (n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0) {
    int nW = 0;  /* # of chars written so far */
    while (nW < n) {
      int nW0 = write(STDOUT_FILENO, buf + nW, n - nW);
      if (nW0 < 0) {
        fprintf(stderr, "write error: %s\n", strerror(errno));
        exit(1);
      }
      nW += nW0;
    }
  }

  if (n < 0) {
    fprintf(stderr, "read error: %s", strerror(errno)); exit(1);
  }

  return 0;
}
```

Programs and Processes

- A *program* is a static entity, whereas a *process* is a dynamic entity.
- Processes are protected from each other. They cannot share resources (like memory) directly without using some form of *inter-process communication* or *IPC*.
- A process usually has only a single *thread* of control, though most modern OS's now allow multiple control threads.
- Every process has a process id or `PID` associated with it.
- On Unix, process control is achieved using `fork()`, `exec()` and `wait()` routines.

Creating Processes

- New processes created using the `fork()` system call.
- Calling `fork()` creates a child process, with both parent and child returning from `fork()` call.
- `fork()` returns 0 in child process.
- `fork()` returns child's PID in parent process.
- Address spaces of parent and child totally disjoint.
- Parent process can `wait()` for child `process(es)` to terminate.

Executing Programs

- An existing process can start executing a new program by using one of the `exec()` family of system calls.
- Current program image is completely replaced by image of new program.
- On success, `exec()` call will not return.

A Simple Shell

`fork()`/`exec()` illustrated by following simple shell program:

```
enum { MAXLINE = 120 };

int
main(void)
{
  char buf[MAXLINE];
  pid_t pid;
  int status;
  while (printf("%% ") && fgets(buf, MAXLINE, stdin) != NULL) {
    buf[strlen(buf) - 1] = '\0';  /* null out '\n' */
    if ( (pid = fork()) < 0) {
      fprintf(stderr, "fork error: %s", strerror(errno));
      exit(1);
    }
    else if (pid == 0) {        /* child */
      execlp(buf, buf, (char *) 0);
      fprintf(stderr, "couldn't execute: %s", buf);
    }
    else { /* parent */
      if ( (pid = waitpid(pid, &status, 0)) < 0) {
        fprintf(stderr, "waitpid error: %s\n", strerror(errno));
        exit(1);
      }
    }
  }
  exit(0);
}
```

Processes

- Process memory area divided into *text* (code), *data*, *heap* and *stack* segments.
- Each process uniquely identified by a integer process id (`PID`). Also possible to obtain its parent's PID (`PPID`).
- Process terminates by requesting its termination (`exit()`) or by being killed by a signal. Termination status available.
- Process credentials include real [ug]id (who is actually running process), effective [ug]id (governs process permissions), and supplementary gid's.

Processes Continued

- The `init` process (`PID == 1`) is ancestor of all regular processes.
- Daemon processes are long-lived processes without a controlling terminal.
- Processes have environment list inherited from parent. `exec()` can change environment.
- Possible to limit resources which can be accessed by a process; *soft* and *hard* limits.

Memory Mappings

- `mmap()` used to change memory mapped into process' vm.
- Mapping can correspond to a file or be anonymous.
- Mappings can be private or shared.
- Used for process initialization, memory allocation, inter-process communication.

Libraries

Libraries are collections of object files. 2 types:

Static

Library code linked into executable. Disadvantage that executable and runtime is larger, library change requires relinking. Advantage that executable is self-contained.

Shared

Executable only contains references to library code which is loaded as needed. Advantage of smaller executables, smaller runtime (library code can be shared among multiple processes), library change automatically propagated to all programs. Disadvantge of requiring a particualr environment (DLL hell).

Inter Process Communication

- Pipes and FIFO’s.
- Message queues.
- Shared memory.
- Semaphores for synchronization.
- Sockets for network communication.

Signals

Signals are used to alert a process of an asynchronous event. They can also be used to provide a simple form of IPC. Signal sources can vary:

- Errors detected by the hardware, like a *divide-by-0* error, or *invalid-memory-access* error.
- Signals generated by the user pressing `Ctl-C` or equivalent.
- Signals generated by other processes.

Handling Signals

A process which receives a signal has the following options:

1. Ignore the signal. This is usually a bad idea.
2. Let the default action occur. The default action often terminates the process.
3. Register a function as a *handler* for the signal. This function will be called asynchronously when a signal is received by the process.

Signal Handling Functions

Signal handling was classically implemented by the following poorly named functions:

`signal()`
: (you don’t want to see the prototype!) is used for registering a handler. It receives as arguments the signal number and a pointer to the handler. If successful, it returns a pointer to the previous handler.

`kill(pid_t pid, int sig)`
: sends a signal `sig` to the process with PID `pid`.

Newer API.

Signal-Handling Shell

Modify simple-shell to get following program which catches `SIGINT`.

```
enum { MAXLINE = 120 };

/* Signal handler */
void
sig_int(int signo)
{
  printf("interrupt\n%% ");
  fflush(stdout);
}

int
main(void)
{
  char buf[MAXLINE];
  pid_t pid;
  int status;

  if (signal(SIGINT, sig_int) == SIG_ERR) {
    fprintf(stderr, "cannot set SIGINT handler: %s\n", strerror(errno));
    exit(1);
  }

  while (printf("%% ") && fgets(buf, MAXLINE, stdin) != NULL) {
    ...
  }
  exit(0);
}
```

Time Values

Calendar Time

The number of seconds since the *Epoch* which is defined as 00:00:00 Jan 1, 1970 UTC. `time_t` is `typedef`'d to hold these values.

Process Time

Measures CPU time in clock ticks. `clock_t` holds these values. `CLK_TCK` or the Ansi-C `CLOCKS_PER_SEC` used for converting the tick count to a time in seconds. The process time is maintained in 3 components:

Clock time

Total elapsed process time. Also referred to as *wall time*.

System time

Time spent executing system/kernel code in kernel-space.

User time

Time spent executing user code in user-space.

Threads

- Processes are isolated from each other. This is an advantage.
- Processes are isolated from each other. This is a disadvantage.
- Creating processes is expensive.
- All threads within a process share the same address space. Hence they are not isolated, but it is also easier to have multiple threads communicate.

A Serial Alarm Clock

Alarms processed serially by following program:

```
int main(int argc, char *argv[]) {
  int alarmN;
  for (alarmN = 0; alarmN >= 0; alarmN++) {
    int t;
    printf("Enter alarm period in seconds: ");
    fflush(stdout);
    if (scanf("%d", &t) != 1) {
      fprintf(stderr, "Integer expected\n");
      continue;
    }
    sleep(t);
    printf("Alarm %d gone off\n", alarmN);
  }
  return 0;
}
```

Serial Alarm Execution

```
$ ./serial_alarm
Enter alarm period in seconds: 10
Alarm 0 gone off
Enter alarm period in seconds: 4
Alarm 1 gone off
Enter alarm period in seconds: 2
Alarm 2 gone off
Enter alarm period in seconds:
$
```

A Forked Alarm Clock

Alarms processed concurrently by following program:

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

/* Buggy program; leaves zombies */
int main(int argc, char *argv[]) {
  int alarmN;
  for (alarmN = 0; alarmN >= 0; alarmN++) {
    int t;
    printf("Enter alarm period in seconds: ");
    fflush(stdout);
    if (scanf("%d", &t) != 1) {
      fprintf(stderr, "Integer expected\n");
      continue;
    }
    { int pid = fork();
      if (pid < 0) {
        fprintf(stderr, "fork error\n"); exit(1);
      }
      else if (pid == 0) { /* child */
        sleep(t);
        printf("Alarm %d gone off\n", alarmN);
        exit(0);        /* child becomes a zombie */
      }
      /* parent continues */
    }
  }
  return 0;
}
```

Forked Alarm Execution

```
$ ./fork_alarm
Enter alarm period in seconds: 10
Enter alarm period in seconds: 4
Enter alarm period in seconds: 2
Enter alarm period in seconds: Alarm 2 gone off
Alarm 1 gone off
Alarm 0 gone off
5
Enter alarm period in seconds: 2
Enter alarm period in seconds: 1
Enter alarm period in seconds: Alarm 5 gone off
Alarm 4 gone off
Alarm 3 gone off

$
```

Threaded Alarm

Alarms processed concurrently by following program:

```
typedef struct {
  int alarmN;      /* alarm # */
  int alarmPeriod; /* alarm period (secs) */
} AlarmInfo;

void *doAlarm(void *argP) {
  const AlarmInfo *aP = (AlarmInfo *)argP;
  sleep(aP->alarmPeriod);
  printf("Alarm %d gone off\n", aP->alarmN);
  free(argP);
  return NULL;
}
```

Threaded Alarm Continued

```
int main(int argc, char *argv[]) {
  int alarmN;
  for (alarmN = 0; alarmN >= 0; alarmN++) {
    int t;
    AlarmInfo *alarmInfoP = malloc(sizeof(AlarmInfo));
    if (!alarmInfoP) {
      perror("allocation error"); exit(1);
    }
    printf("Enter alarm period in seconds: ");
    fflush(stdout);
    if (scanf("%d", &t) != 1) {
      fprintf(stderr, "Integer expected\n");
      continue;
    }
    alarmInfoP->alarmN = alarmN; alarmInfoP->alarmPeriod = t;
    { pthread_t thread;
      int status =
        pthread_create(&thread, NULL, doAlarm, alarmInfoP);
      if (status != 0) {
        fprintf(stderr, "%s\n", strerror(status));
      }
      status = pthread_detach(thread);
      if (status != 0) {
        fprintf(stderr, "%s\n", strerror(status));
      }
    }
  }
  return 0;
}
```

Process Groups and Sessions

- Pipeline processes run in a process group.
- Session is a collection of process group.
- A session has upto one controlling terminal.
- Used by job control shells (all modern shells) to allow easy control of multiple jobs.

References

Text, Ch 2, 3.

Text, Ch 29 (will be revisited in detail later).

David R. Butenhof, *Programming with POSIX Threads*, Addison-Wesley, 1997.