Q.2 a. Explain salient features of UNIX operating system. (4)

Answer:

Unix (officially UNIX) is a registered trademark of The Open Group that refers to a family of computer operating systems and tools conforming to The Open Group Base Specification, Issue 7 (also known as POSIX.1-2008 or IEEE Std 1003.1 - 2008). To use the Unix trademark, an operating system vendor must pay a licensing fee and annual trademark royalties to The Open Group. Officially licensed Unix operating systems (and their vendors) include OS X (Apple), Solaris (Oracle), AIX (IBM), IRIX (SGI), and HP-UX (Hewlett-Packard).

Note: Operating systems that behave like Unix systems and provide similar utilities, but do not conform to Unix specification or are not licensed by The Open Group, are commonly known as Unix-like systems. These include a wide variety of Linux distributions (e.g., Red Hat Enterprise Linux, Ubuntu, and CentOS) and several descendents of the Berkeley Software Distribution operating system (e.g., FreeBSD, OpenBSD, and NetBSD).

Propietary Unix operating systems (and Unix-like variants) run on a wide variety of digital architectures, and are commonly used on web servers, mainframes, and supercomputers. In recent years, smartphones, tablets, and personal computers running versions or variants of Unix have become increasingly popular.

The original Unix operating system was developed at AT&T's Bell Labs research center in 1969. In the 1970s and 1980s, AT&T licensed Unix to third-party vendors, leading to the development of several Unix variants, including Berkeley Unix, HP-UX, AIX, and Microsoft's Xenix. In 1993, AT&T sold the rights to the Unix operating system to Novell, Inc., which a few years later sold the Unix trademark to the consortium that eventually became The Open Group.

Unix was developed using a high-level programming language (C) instead of platform-specific assembly language, enabling its portability across multiple computer platforms. Unix also was developed as a self-contained software system, comprising the operating system, development environment, utilities, documentation, and modifiable source code. These key factors led to widespread use and further development in commercial settings, and helped Unix and its variants become an important teaching and learning tool used in academic settings.

b. Compare internal and external commands of UNIX with suitable examples. Also explain the file attributes displayed by ls –l command. (8)

Answer:

UNIX commands are classified into two types

- Internal Commands - Ex: cd, source, fg
- External Commands - Ex: ls, cat

Let us look at these in detail

Internal Command:

Internal commands are something which is built into the shell. For the shell built in commands, the execution speed is really high. It is because no process needs to be spawned for executing it. For example, when using the "cd" command, no process is created. The current directory simply gets changed on executing it.

External Command:
External commands are not built into the shell. These are executables present in a separate file. When an external command has to be executed, a new process has to be spawned and the command gets executed. For example, when you execute the "cat" command, which usually is at /usr/bin, the executable /usr/bin/cat gets executed.

How to get the list of Internal commands?
You can get only if you are in bash shell. Bash shell has a command called "help" which will list out all the built-in shell commands.

```
$ help

 alias [-p] [name[=value] ... ]     bg [job_spec ...]

 bind [-lpvsPVS] [-m keymap] [-f fi break [n]

 builtin [shell-builtin [arg ...]]  caller [EXPR]

 case WORD in [PATTERN [| PATTERN]. cd [-L|-P] [dir]

 command [-pVv] command [arg ...]   compgen [-abcdefgjksuv] [-
o option

.....
```

How to find out whether a command is internal or external?
type command:

```
$ type cd

cd is a shell builtin

$ type cat

cat is /bin/cat
```

For the internal commands, the type command will clearly say its shell built-in, however for the external commands, it gives the path of the command from where it is executed.

Internal vs External?
The question whether should we use an internal command or an external command OR which is better always does not make sense. Because in most of the situations you will end

up using the command which does your job which could be either internal or external.

The big difference in internal vs external command is performance. Internal command are much much faster compared to external for the simple reason that no process needs to be spawned for an internal command since it is all built-into the shell. So, as the size of a script gets bigger, using external commands a lot does adds to its performance.

Not always we get a choice to choose an internal over an external command. However, a careful look at our scripting practices, we might find quite a few places where we can avoid external commands.

Example:
Say to add 2 numbers say x & y:

Not good:

```
z=`expr $x+$y`
```

Good:

```
let z=x+y
```

let is a shell built-in command, whereas expr is an external command. Using expr will be slower. This might be very negligible when you are using it at an one-off instance. Using it in a place say on every record of a file containing million records does give a different dimension to it.

c. Explain the following commands: **(2×2)**
i. creat function
ii. lseek function

Answer:

3.4 creat Function

A new file can also be created by calling the creat function.

```
#include <fcntl.h>

int creat(const char *pathname, mode_t mode);
```

Returns: file descriptor opened for write-only if OK, −1 on error

Note that this function is equivalent to

```
open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

Historically, in early versions of the UNIX System, the second argument to open could be only 0, 1, or 2. There was no way to open a file that didn't already exist. Therefore, a separate system call, creat, was needed to create new files. With the O_CREAT and O_TRUNC options now provided by open, a separate creat function is no longer needed.

We'll show how to specify *mode* in Section 4.5 when we describe a file's access permissions in detail.

One deficiency with creat is that the file is opened only for writing. Before the new version of open was provided, if we were creating a temporary file that we wanted to write and then read back, we had to call creat, close, and then open. A better way is to use the open function, as in

```
open(pathname, O_RDWR | O_CREAT | O_TRUNC, mode);
```

3.6 lseek Function

Every open file has an associated "current file offset," normally a non-negative integer that measures the number of bytes from the beginning of the file. (We describe some exceptions to the "non-negative" qualifier later in this section.) Read and write operations normally start at the current file offset and cause the offset to be incremented by the number of bytes read or written. By default, this offset is initialized to 0 when a file is opened, unless the O_APPEND option is specified.

An open file's offset can be set explicitly by calling lseek.

```
#include <unistd.h>
off_t lseek(int filedes, off_t offset, int whence);
                                  Returns: new file offset if OK, -1 on error
```

The interpretation of the *offset* depends on the value of the *whence* argument.

- If *whence* is SEEK_SET, the file's offset is set to *offset* bytes from the beginning of the file.
- If *whence* is SEEK_CUR, the file's offset is set to its current value plus the *offset*. The *offset* can be positive or negative.
- If *whence* is SEEK_END, the file's offset is set to the size of the file plus the *offset*. The *offset* can be positive or negative.

Because a successful call to lseek returns the new file offset, we can seek zero bytes from the current position to determine the current offset:

```
off_t    currpos;
currpos = lseek(fd, 0, SEEK_CUR);
```

This technique can also be used to determine if a file is capable of seeking. If the file descriptor refers to a pipe, FIFO, or socket, lseek sets errno to ESPIPE and returns -1.

Q.3 a. Explain the procedure of process creation in UNIX with the help of system calls. (4)

Answer:

Process creation in UNIX

The seven-state logical process model we considered in a previous lecture can accommodate the UNIX process model with some modifications, actually becoming a ten state model.

First, as we previously observed, UNIX executes most kernel services within a process's context, by implementing a mechanism which separates between the two possible modes of execution of a process. Hence our previously unique ``Running'' state must actually be split in a ``User Running'' state and a ``Kernel Running'' state. Moreover a process preemption mechanism is usually implemented in the UNIX scheduler to enforce priority. This allows a process returning from a system call (hence after having run in kernel mode) to be immediately blocked and put in the ready processes queue instead of returning to user mode running, leaving the CPU to another process. So it's worth considering a ``Preempted'' state as a special case of ``Blocked''. Moreover, among exited processes there's a distinction between those which have a parent process that waits for their completion (possibly to clean after them), and those which upon termination have an active parent that might decide to wait for them sometime in the future (and then be immediately notified of its children's termination)�. These last processes are called ``Zombie'', while the others are ``Exited''. The difference is that the system needs to maintain an entry in the process table for a zombie, since its parent might reference it in the future, while the entry for an exited (and waited for) process can be discarded without further fiddling. So the much talked about ``Zombie'' processes of UNIX are nothing but entries in a system table, the system having already disposed of all the rest of their image. This process model is depicted in fig. 5.

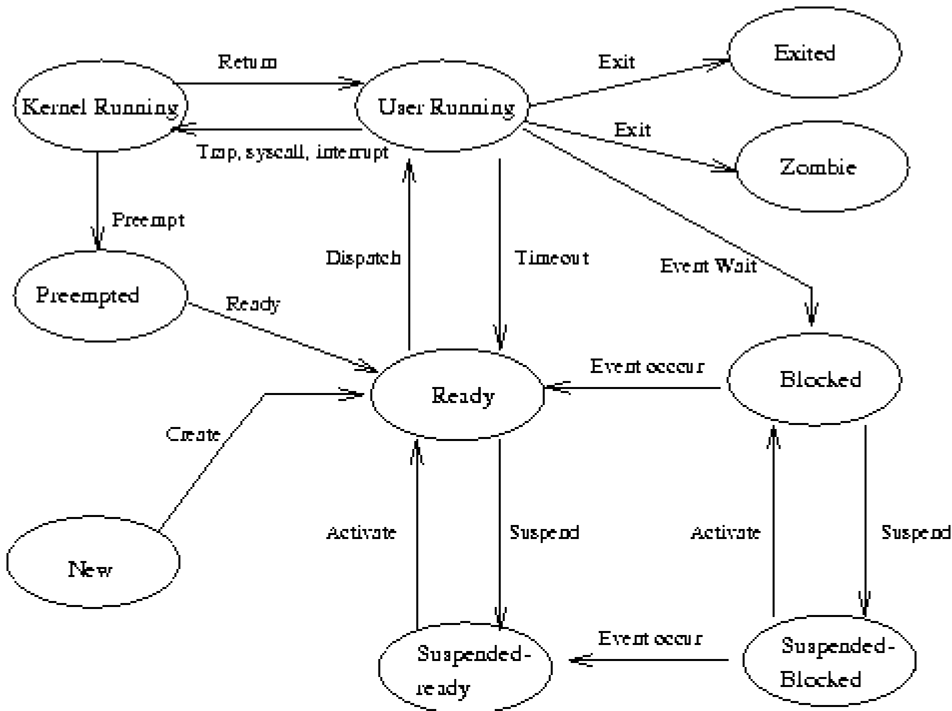


Figure 5: UNIX process state model

All processes in UNIX� are created using the `fork()` system call. System calls in UNIX can be best thought of as C functions provided with the standard C library. Even if their particular implementation is depends on the particular UNIX flavor (and on hardware, for many of them), a C API� is always provided, and is consistent among the different unices, at least in the fundamental traits.

UNIX implements through the `fork()` and `exec()` system calls an elegant two-step mechanism for process creation and execution. `fork()` is used to create the image of a process using the one of an existing one, and `exec` is used to execute a program by overwriting that image with the program's one. This separation allows to perform some interesting housekeeping actions in between, as we'll see in the following lectures.

A call to `fork()` of the form:

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

pid_t childpid;
...
childpid = fork(); /* child's pid in the parent, 0 in the child */
...
```

creates (if it succeeds) a new process, which a child of the caller's, and is an exact copy of of the (parent) caller itself. By exact copy we mean that it's image is a physical bitwise copy of the parent's (in principle, they *do not* share the image in memory: though there can be exceptions to this rule, we can always thing of the two

images as being stored in two separate and protected address spaces in memory, hence a manipulation of the parent's variables won't affect the child's copies, and vice versa). The only visible differences are in the PCB, and the most relevant (for now) of them are the following:

- The two processes obviously have two different process id.s. (pid). In a C program process id.s are conveniently represented by variables of `pid_t` type, the type being defined in the `sys/types.h` header.
- In UNIX the PCB of a process contains the id of the process's parent, hence the child's PCB will contain as parent id (ppid) the pid of the process that called`fork()`, while the caller will have as ppid the pid of the process that spawned it.
- The child process has its own copy of the parent's file descriptors. These descriptors reference the same under-lying objects, so that files are shared between the child and the parent. This makes sense, since other processes might access those files as well, and having them already open in the child is a time-saver.

The `fork()` call returns in both the parent and the child, and both resume their execution from the statement immediately following the call. One usually wants that parent and child behave differently, and the way to distinguish between them in the program's source code is to test the value returned by `fork()`. This value is 0 in the child, and the child's pid in the parent. Since `fork()` returns -1 in case the child spawning fails, a catch-all C code fragment to separate behaviours may look like the following:

```
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
...
pid_t childpid;
...
childpid=fork();
switch(childpid)
{
   case -1:
      fprintf(stderr,"ERROR: %s\n", sys_errlist[errno]);
      exit(1);
      break;
   case 0:
      /* Child's code goes here */
      break;
   default:
      /* Parent's code goes here */
      break;
}
```

The array of strings `char *sys_errlist[]` and the global integer variable `int errno` are defined in the `errno.h` header. The former contains a list of system error messages, and the latter is set to index the appropriate message whenever an error occurs. For each system call several possible error conditions are defined. Each of them is associated to an integer constant - defined via a `#define` directive in one system header file - whose value is exactly the one that `errno` takes when an error occurs. A sample definition (from the sys/errno.h header) is:

```
...
#define ENOMEM 12
...
```

which defines the error that might occur when a process creation fails because there's not enough memory available.

Note that a child (i.e. a process whatsoever, since they are all children of some other process, with the exception of processes 0, `swapper` and 1, `init`) cannot use the value returned by `fork()` to know its pid, since this is always 0 in the child. A system call named `getpid()` is provided for this purpose, and another one, named `getppid()` is used to ask the system about the parent's id. Both functions take no arguments and return the requested value in `pid_t` type, or -1 in case of failure.

In the above program fragment, a system call to `exit()` is made in case of failure, which causes the program to abort (you might want to deal with the errors in a smoother way, depending on your application, and perform some application-dependent error recovery action). We'll see later that the `exit()` call returns the lower 8 bits of its argument (1, in the above example) to a waiting parent process, which can use them to determine the child's exit status and behave accordingly. The usual convention is to exit with 0 on correct termination, and with a meaningful (for the parent) error code on abort.

It is often the case that a parent process must coordinate its actions with those of its children, maybe exchanging with them various kind of messages. UNIX defines several sophisticated inter-process communication (IPC) mechanisms, the simplest of which is a parent's ability to test the termination status of its children. A synchronization mechanism is provided via the `wait()` system call, that allows a parent to sleep until one of its children exits, and then get its exit status. This call actually comes in three flavors, one simply called `wait()` and common to all version of UNIX (that i know of), one called `waitpid()`, which is a POSIX extension, and one called `wait3()`, and it's a BSD extension.

b. With the help of suitable diagram, explain UNIX file system and also explain the various types of files supported in UNIX. (8)

Answer:

A file system is a logical collection of files on a partition or disk. A partition is a container for information and can span an entire hard drive if desired.

Your hard drive can have various partitions which usually contains only one file system, such as one file system housing the / file system or another containing the /home file system.

One file system per partition allows for the logical maintenance and management of differing file systems.

Everything in Unix is considered to be a file, including physical devices such as DVD-ROMs, USB devices, floppy drives, and so forth.

Directory Structure:

Unix uses a hierarchical file system structure, much like an upside-down tree, with root (/) at the base of the file system and all other directories spreading from there.

A UNIX filesystem is a collection of files and directories that has the following properties:

- It has a root directory (/) that contains other files and directories.
- Each file or directory is uniquely identified by its name, the directory in which it resides, and a unique identifier, typically called an inode.
- By convention, the root directory has an inode number of 2 and the lost+found directory has an inode number of 3. Inode numbers 0 and 1 are not used. File inode numbers can be seen by specifying the -i option to ls command.
- It is self contained. There are no dependencies between one filesystem and any other.

The directories have specific purposes and generally hold the same types of information for easily locating files. Following are the directories that exist on the major versions of Unix:

Directory	Description
/	This is the root directory which should contain only the directories needed at the top level of the file structure.
/bin	This is where the executable files are located. They are available to all user.
/dev	These are device drivers.
/etc	Supervisor directory commands, configuration files, disk configuration files, valid user lists, groups, ethernet, hosts, where to send critical messages.
/lib	Contains shared library files and sometimes other kernel-related files.
/boot	Contains files for booting the system.
/home	Contains the home directory for users and other accounts.
/mnt	Used to mount other temporary file systems, such as cdrom and floppy for the CD-ROM drive and floppy diskette drive, respectively
/proc	Contains all processes marked as a file by process number or other information that is dynamic to the system.
/tmp	Holds temporary files used between system boots

/usr	Used for miscellaneous purposes, or can be used by many users. Includes administrative commands, shared files, library files, and others
/var	Typically contains variable-length files such as log and print files and any other type of file that may contain a variable amount of data
/sbin	Contains binary (executable) files, usually for system administration. For example*fdisk* and *ifconfig* utlities.
/kernel	Contains kernel files

c. Explain the commands Umask and chown. Give syntax and examples. (4)

Answer:

4.8 umask Function

Now that we've described the nine permission bits associated with every file, we can describe the file mode creation mask that is associated with every process.

The umask function sets the file mode creation mask for the process and returns the previous value. (This is one of the few functions that doesn't have an error return.)

```
#include <sys/stat.h>

mode_t umask(mode_t cmask);
```

Returns: previous file mode creation mask

The *cmask* argument is formed as the bitwise OR of any of the nine constants from Figure 4.6: S_IRUSR, S_IWUSR, and so on.

The file mode creation mask is used whenever the process creates a new file or a new directory. (Recall from Sections 3.3 and 3.4 our description of the open and creat functions. Both accept a *mode* argument that specifies the new file's access permission bits.) We describe how to create a new directory in Section 4.20. Any bits that are *on* in the file mode creation mask are turned *off* in the file's *mode*.

Example

The program in Figure 4.9 creates two files, one with a umask of 0 and one with a umask that disables all the group and other permission bits.

```
#include "apue.h"
#include <fcntl.h>

#define RWRWRW (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)

int
main(void)
{
    umask(0);
    if (creat("foo", RWRWRW) < 0)
        err_sys("creat error for foo");
    umask(S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
    if (creat("bar", RWRWRW) < 0)
```

```
        err_sys("creat error for bar");
    exit(0);
}
```

Figure 4.9 Example of umask function

If we run this program, we can see how the permission bits have been set.

```
$ umask                          first print the current file mode creation mask
002
$ ./a.out
$ ls -l foo bar
-rw-------  1 sar                0 Dec  7 21:20 bar
-rw-rw-rw-  1 sar                0 Dec  7 21:20 foo
$ umask                          see if the file mode creation mask changed
002
```

Most users of UNIX systems never deal with their umask value. It is usually set once, on login, by the shell's start-up file, and never changed. Nevertheless, when writing programs that create new files, if we want to ensure that specific access permission bits are enabled, we must modify the umask value while the process is running. For example, if we want to ensure that anyone can read a file, we should set the umask to 0. Otherwise, the umask value that is in effect when our process is running can cause permission bits to be turned off.

In the preceding example, we use the shell's umask command to print the file mode creation mask before we run the program and after. This shows us that changing the file mode creation mask of a process doesn't affect the mask of its parent (often a shell). All of the shells have a built-in umask command that we can use to set or print the current file mode creation mask.

Users can set the umask value to control the default permissions on the files they create. The value is expressed in octal, with one bit representing one permission to be masked off, as shown in Figure 4.10. Permissions can be denied by setting the corresponding bits. Some common umask values are 002 to prevent others from writing your files, 022 to prevent group members and others from writing your files, and 027 to prevent group members from writing your files and others from reading, writing, or executing your files.

Mask bit	Meaning
0400	user-read
0200	user-write
0100	user-execute
0040	group-read
0020	group-write
0010	group-execute
0004	other-read
0002	other-write
0001	other-execute

Figure 4.10 The umask file access permission bits

The Single UNIX Specification requires that the shell support a symbolic form of the umask command. Unlike the octal format, the symbolic format specifies which permissions are to be allowed (i.e., clear in the file creation mask) instead of which ones are to be denied (i.e., set in the file creation mask). Compare both forms of the command, shown below.

```
$ umask                     first print the current file mode creation mask
002
$ umask -S                  print the symbolic form
u=rwx,g=rwx,o=rx
$ umask 027                 change the file mode creation mask
$ umask -S                  print the symbolic form
u=rwx,g=rx,o=
```

4.11 chown, fchown, and lchown Functions

The chown functions allow us to change the user ID of a file and the group ID of a file.

```
#include <unistd.h>

int chown(const char *pathname, uid_t owner, gid_t group);

int fchown(int filedes, uid_t owner, gid_t group);

int lchown(const char *pathname, uid_t owner, gid_t group);
```

All three return: 0 if OK, –1 on error

These three functions operate similarly unless the referenced file is a symbolic link. In that case, lchown changes the owners of the symbolic link itself, not the file pointed to by the symbolic link.

The lchown function is an XSI extension to the POSIX.1 functionality defined in the Single UNIX Specification. As such, all UNIX System implementations are expected to provide it.

If either of the arguments *owner* or *group* is –1, the corresponding ID is left unchanged.

Historically, BSD-based systems have enforced the restriction that only the superuser can change the ownership of a file. This is to prevent users from giving away

their files to others, thereby defeating any disk space quota restrictions. System V, however, has allowed any user to change the ownership of any files they own.

> POSIX.1 allows either form of operation, depending on the value of _POSIX_CHOWN_RESTRICTED.
>
> With Solaris 9, this functionality is a configuration option, whose default value is to enforce the restriction. FreeBSD 5.2.1, Linux 2.4.22, and Mac OS X 10.3 always enforce the chown restriction.

&call from Section 2.6 that the _POSIX_CHOWN_RESTRICTED constant a n optionally be defined in the header <unistd.h>, and can always be queried using either the pathconf function or the fpathconf function. Also recall that this option can depend on the referenced file; it can be enabled or disabled on a per file system basis. We'll use the phrase, if _POSIX_CHOWN_RESTRICTED is in effect, to mean if it applies to the particular file that we're talking about, regardless of whether this actual constant is defined in the header.

If _POSIX_CHOWN_RESTRICTED is in effect for the specified file, then

1. Only a superuser process can change the user ID of the file.
2. A nonsuperuser process can change the group ID of the file if the process owns the file (the effective user ID equals the user ID of the file), *owner* is specified as −1 or equals the user ID of the file, and *group* equals either the effective group ID of the process or one of the process's supplementary group IDs.

This means that when _POSIX_CHOWN_RESTRICTED is in effect, you can't change the user ID of other users' files. You can change the group ID of files that you own, but only to groups that you belong to.

If these functions are called by a process other than a superuser process, on successful return, both the set-user-ID and the set-group-ID bits are cleared.

Q.4 a. What do you mean by standard input, standard output and standard error? (8)

Answer:

5.3 Standard Input, Standard Output, and Standard Error

Three streams are predefined and automatically available to a process: standard input, standard output, and standard error. These streams refer to the same files as the file descriptors STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO, which we mentioned in Section 3.2.

These three standard I/O streams are referenced through the predefined file pointers stdin, stdout, and stderr. The file pointers are defined in the <stdio.h> header.

b. Write short note on:

(i) Password file (ii) Shadow passwords (8)

Answer:

6.2 Password File

The UNIX System's password file, called the user database by POSIX.1, contains the fields shown in Figure 6.1. These fields are contained in a passwd structure that is defined in <pwd.h>.

> Note that POSIX.1 specifies only five of the ten fields in the passwd structure. Most platforms support at least seven of the fields. The BSD-derived platforms support all ten.

Description	struct passwd member	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
user name	char *pw_name	•	•	•	•	•
encrypted password	char *pw_passwd		•	•	•	•
numerical user ID	uid_t pw_uid	•	•	•	•	•
numerical group ID	gid_t pw_gid	•	•	•	•	•
comment field	char *pw_gecos		•	•	•	•
initial working directory	char *pw_dir	•	•	•	•	•
initial shell (user program)	char *pw_shell	•	•	•	•	•
user access class	char *pw_class		•		•	
next time to change password	time_t pw_change		•		•	
account expiration time	time_t pw_expire		•		•	

Figure 6.1 Fields in `/etc/passwd` file

Historically, the password file has been stored in `/etc/passwd` and has been an ASCII file. Each line contains the fields described in Figure 6.1, separated by colons. For example, four lines from the `/etc/passwd` file on Linux could be

```
root:x:0:0:root:/root:/bin/bash
squid:x:23:23::/var/spool/squid:/dev/null
nobody:x:65534:65534:Nobody:/home:/bin/sh
sar:x:205:105:Stephen Rago:/home/sar:/bin/bash
```

Note the following points about these entries.

- There is usually an entry with the user name `root`. This entry has a user ID of 0 (the superuser).
- The encrypted password field contains a single character as a placeholder where older versions of the UNIX System used to store the encrypted password. Because it is a security hole to store the encrypted password in a file that is readable by everyone, encrypted passwords are now kept elsewhere. We'll cover this issue in more detail in the next section when we discuss passwords.
- Some fields in a password file entry can be empty. If the encrypted password field is empty, it usually means that the user does not have a password. (This is not recommended.) The entry for `squid` has one blank field: the comment field. An empty comment field has no effect.
- The shell field contains the name of the executable program to be used as the login shell for the user. The default value for an empty shell field is usually `/bin/sh`. Note, however, that the entry for `squid` has `/dev/null` as the login shell. Obviously, this is a device and cannot be executed, so its use here is to prevent anyone from logging in to our system as user `squid`.

  Many services have separate user IDs for the daemon processes (Chapter 13) that help implement the service. The `squid` entry is for the processes implementing the `squid` proxy cache service.

- There are several alternatives to using /dev/null to prevent a particular user from logging in to a system. It is common to see /bin/false used as the login shell. It simply exits with an unsuccessful (nonzero) status; the shell evaluates the exit status as false. It is also common to see /bin/true used to disable an account. All it does is exit with a successful (zero) status. Some systems provide the nologin command. It prints a customizable error message and exits with a nonzero exit status.
- The nobody user name can be used to allow people to log in to a system, but with a user ID (65534) and group ID (65534) that provide no privileges. The only files that this user ID and group ID can access are those that are readable or writable by the world. (This assumes that there are no files specifically owned by user ID 65534 or group ID 65534, which should be the case.)
- Some systems that provide the finger(1) command support additional information in the comment field. Each of these fields is separated by a comma: the user's name, office location, office phone number, and home phone number. Additionally, an ampersand in the comment field is replaced with the login name (capitalized) by some utilities. For example, we could have

```
sar:x:205:105:Steve Rago, SF 5-121, 555-1111, 555-2222:/home/sar:/bin/sh
```

Then we could use finger to print information about Steve Rago.

```
$ finger -p sar
Login: sar                              Name: Steve Rago
Directory: /home/sar                    Shell: /bin/sh
Office:  SF 5-121,  555-1111            Home Phone:  555-2222
On since Mon Jan 19 03:57 (EST) on ttyv0 (messages off)
No Mail.
```

Even if your system doesn't support the finger command, these fields can still go into the comment field, since that field is simply a comment and not interpreted by system utilities.

Some systems provide the vipw command to allow administrators to edit the password file. The vipw command serializes changes to the password file and makes sure that any additional files are consistent with the changes made. It is also common for systems to provide similar functionality through graphical user interfaces.

POSIX.1 defines only two functions to fetch entries from the password file. These functions allow us to look up an entry given a user's login name or numerical user ID

```
#include <pwd.h>

struct passwd *getpwuid(uid_t uid);

struct passwd *getpwnam(const char *name)
```

Both return: pointer if OK, NULL on error

The getpwuid function is used by the ls(1) program to map the numerical user ID contained in an i-node into a user's login name. The getpwnam function is used by the login(1) program when we enter our login name.

© IETE

Both functions return a pointer to a passwd structure that the functions fill in. This structure is usually a static variable within the function, so its contents are overwritten each time we call either of these functions.

These two POSIX.1 functions are fine if we want to look up either a login name or a user ID, but some programs need to go through the entire password file. The following three functions can be used for this.

```
#include <pwd.h>

struct passwd *getpwent(void);

                    Returns: pointer if OK, NULL on error or end of file

void setpwent(void);

void endpwent(void);
```

These three functions are not part of the base POSIX.1 standard. They are defined as XSI extensions in the Single UNIX Specification. As such, all UNIX systems are expected to provide them.

We call getpwent to return the next entry in the password file. As with the two POSIX.1 functions, getpwent returns a pointer to a structure that it has filled in. This structure is normally overwritten each time we call this function. If this is the first call to this function, it opens whatever files it uses. There is no order implied when we use this function; the entries can be in any order, because some systems use a hashed version of the file /etc/passwd.

The function setpwent rewinds whatever files it uses, and endpwent closes these files. When using getpwent, we must always be sure to close these files by calling endpwent when we're through. Although getpwent is smart enough to know when it has to open its files (the first time we call it), it never knows when we're through.

Example

Figure 6.2 shows an implementation of the function getpwnam.

```
#include <pwd.h>
#include <stddef.h>
#include <string.h>

struct passwd *
getpwnam(const char *name)
{
    struct passwd  *ptr;

    setpwent();
    while ((ptr = getpwent()) != NULL)
        if (strcmp(name, ptr->pw_name) == 0)
            break;      /* found a match */
    endpwent();
    return(ptr);    /* ptr is NULL if no match found */
}
```

Figure 6.2 The getpwnam function

The call to setpwent at the beginning is self-defense: we ensure that the files are rewound, in case the caller has already opened them by calling getpwent. The call to endpwent when we're done is because neither getpwnam nor getpwuid should leave any of the files open. □

6.3 Shadow Passwords

The encrypted password is a copy of the user's password that has been put through a one-way encryption algorithm. Because this algorithm is one-way, we can't guess the original password from the encrypted version.

Historically, the algorithm that was used (see Morris and Thompson [1979]) always generated 13 printable characters from the 64-character set [a-zA-Z0-9./]. Some newer systems use an MD5 algorithm to encrypt passwords, generating 31 characters per encrypted password. (The more characters used to store the encrypted password, the more combinations there are, and the harder it will be to guess the password by trying all possible variations.) When we place a single character in the encrypted password field, we ensure that an encrypted password will never match this value.

Given an encrypted password, we can't apply an algorithm that inverts it and returns the plaintext password. (The plaintext password is what we enter at the Password: prompt.) But we could *guess* a password, run it through the one-way algorithm, and compare the result to the encrypted password. If user passwords were randomly chosen, this brute-force approach wouldn't be too successful. Users, however, tend to choose nonrandom passwords, such as spouse's name, street names, or pet names. A common experiment is for someone to obtain a copy of the password file and try guessing the passwords. (Chapter 4 of Garfinkel et al. [2003] contains additional details and history on passwords and the password encryption scheme used on UNIX systems.)

To make it more difficult to obtain the raw materials (the encrypted passwords), systems now store the encrypted password in another file, often called the *shadow password file*. Minimally, this file has to contain the user name and the encrypted password. Other information relating to the password is also stored here (Figure 6.3).

Description	struct spwd member
user login name	char *sp_namp
encrypted password	char *sp_pwdp
days since Epoch of last password change	int sp_lstchg
days until change allowed	int sp_min
days before change required	int sp_max
days warning for expiration	int sp_warn
days before account inactive	int sp_inact
days since Epoch when account expires	int sp_expire
reserved	unsigned int sp_flag

Figure 6.3 Fields in /etc/shadow file

The only two mandatory fields are the user's login name and encrypted password. The other fields control how often the password is to change—known as "password aging"—and how long an account is allowed to remain active.

The shadow password file should not be readable by the world. Only a few programs need to access encrypted passwords—login(1) and passwd(1), for example—and these programs are often set-user-ID root. With shadow passwords, the regular password file, /etc/passwd, can be left readable by the world.

On Linux 2.4.22 and Solaris 9, a separate set of functions is available to access the shadow password file, similar to the set of functions used to access the password file.

```
#include <shadow.h>

struct spwd *getspnam(const char *name);

struct spwd *getspent(void);

                    Both return: pointer if OK, NULL on error

void setspent(void);
void endspent(void);
```

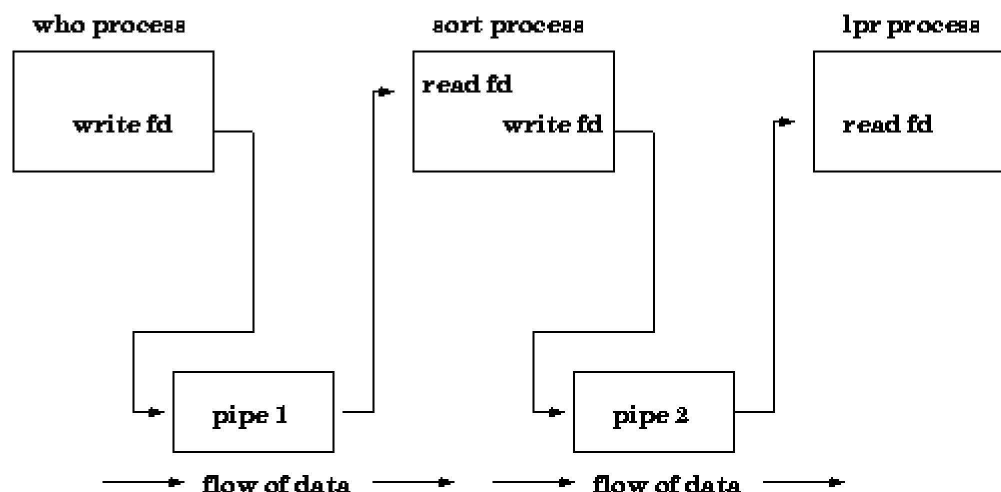
On FreeBSD 5.2.1 and Mac OS X 10.3, there is no shadow password structure. The additional account information is stored in the password file (refer back to Figure 6.1).

Q.5 a. Discuss the concept of pipes. Illustrate the syntax and working of DUP and open system calls. (8)

Answer:

- A Unix pipe provides a one-way flow of data.
- For example, if a Unix users issues the command
- `who | sort | lpr`

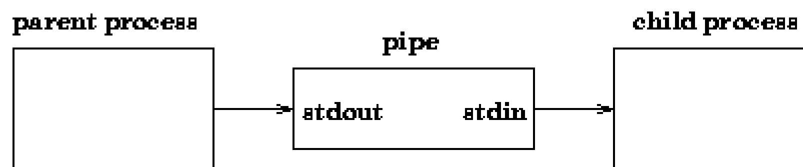
then the Unix shell would create three processes with two pipes between them:



- A pipe can be explicitly created in Unix using the *pipe* system call. Two file descriptors are returned--fildes[0] and fildes[1], and they are both open for reading and writing. A read from fildes[0] accesses the data written to fildes[1]

on a first-in-first-out (FIFO) basis and a read from fildes[1] accesses the data written to fildes[0] also on a FIFO basis.

- When a pipe is used in a Unix command line, the first process is assumed to be writing to stdout and the second is assumed to be reading from stdin. So, it is common practice to assign the pipe write device descriptor to stdout in the first process and assign the pipe read device descriptor to stdin in the second process. This is elaborated below in the discussion of multiple command pipelines.

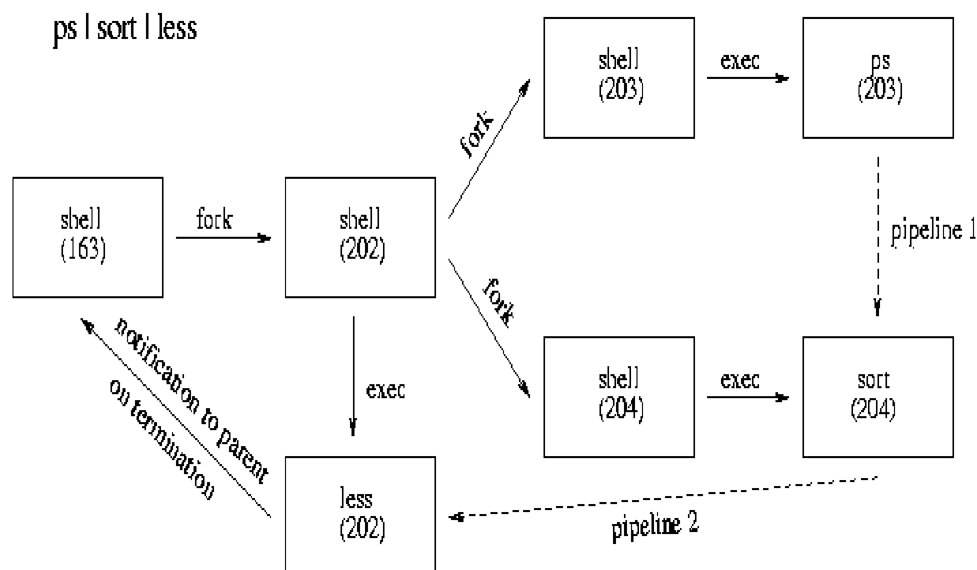


Multiple Command Pipelines: Architecture

Creating a pipeline between two processes is fairly simple, but building a multiple command pipeline is more complicated. The relationship between all of the processes in question is different than what one would expect when creating a simple pipeline between two processes. Normally a pipeline between two processes results in a fork() where child and parent are able to communicate.

In an extension of this model to n pipes, it is natural to assume a chain of processes in which each is the child of the previous one, until the n'th child is forked. But this model does not work because the parent shell must wait for the *last* command in the pipeline to complete, not the first, as would be the case with a chained pipeline.

A multiple process pipeline can be represented graphically as:



Dup:

```
#include <unistd.h>



int dup(int oldfd);

int dup2(int oldfd, int newfd);
```

DESCRIPTION

dup() and **dup2**() create a copy of the file descriptor *oldfd*.

After a successful return from **dup**() or **dup2**(),the old and new file descriptors may be used interchangeably. They refer to the same open file description (see **open**(2)) and thus share file offset and file status flags; for example, if the file offset is modified by using **lseek**(2) on one of the descriptors, the offset is also changed for the other.

The two descriptors do not share file descriptor flags (the close-on-exec flag). The close-on-exec flag (**FD_CLOEXEC**; see **fcntl**(2)) for the duplicate descriptor is off.

dup() uses the lowest-numbered unused descriptor for the new descriptor.

dup2() makes *newfd* be the copy of *oldfd*, closing *newfd* first if necessary.

RETURN VALUE
dup() and **dup2**() return the new descriptor, or -1 if an error occurred (in which case, *errno* is set appropriately).

b. What are standard input, output and errors? Explain in context of Unix. Give examples. (6)

Answer:
Standard input, standard output, and standard error files
When a command begins running, it usually expects that the following files are already open: standard input, standard output, and standard error (sometimes called error output or diagnostic output).

A number, called a file descriptor, is associated with each of these files, as follows:

File descriptor 0 Standard input
File descriptor 1 Standard output
File descriptor 2 Standard error (diagnostic) output
A child process normally inherits these files from its parent. All three files are initially assigned to the workstation (0 to the keyboard, 1 and 2 to the display). The shell permits them to be redirected elsewhere before control is passed to a command.

When you enter a command, if no file name is given, your keyboard is the standard input, sometimes denoted as stdin. When a command finishes, the results are displayed on your screen.

Your screen is the standard output, sometimes denoted as stdout. By default, commands take input from the standard input and send the results to standard output.

Error messages are directed to standard error, sometimes denoted as stderr. By default, this is your screen.

These default actions of input and output can be varied. You can use a file as input and write results of a command to a file. This is called input/output redirection.

The output from a command, which normally goes to the display device, can be redirected to a file instead. This is known as output redirection. This is useful when you have a lot of output that is difficult to read on the screen or when you want to put files together to create a larger file.

Though not used as much as output redirection, the input for a command, which normally comes from the keyboard, can also be redirected from a file. This is known as input redirection. Redirection of input lets you prepare a file in advance and then have the command read the file.

c. Differentiate between a wildcard and a regular expression. (2)

Answer:

In the **Formal** definition the symbols of regular expressions operators are

. : which is concatenation like a.b.c would match a text having abc . Some times to indicate concatenation simply two symbols are used back to back.

* : match 0 more more times the last symbol, (abc)* would match a null string, abc, abcabc, abcabcabc, but not abcaabc. Known as the Kleen's star.

+ : would match either the left hand side or the right hand side . (abc + def) would match abc or def. Also the union operator or the | operator is used.

These are applied on a set of symbols *sigma*, which includes the symbols in your language within other special sumbols are the *epsilon* which denotes the empty string, and the *null* means no symbols at all. For details see 3

These are the **formal** definitions.

When you use applications accepting the POSIX regular expression syntax the meaning of the different operators are like this:

Q.6 a. Write a AWK program to find the square root of all the numbers from 1-10. (8)

Answer:

```
In this
example, the value of `sqrt(ARGUMENT)' is the square root of ARGUMENT.
The following program reads numbers, one number per line, and prints the
square root of each one:

     $ awk '{ print "The square root of", $1, "is", sqrt($1) }'
     1
     -| The square root of 1 is 1
     3
```

```
-| The square root of 3 is 1.73205
5
-| The square root of 5 is 2.23607
Ctrl-d
```

b. Discuss the following:
(i) setjmp and longjmp functions
(ii) getrlimt and setrlimit functions **(8)**

Answer:

setjmp and longjmp Functions

In C, we can't goto a label that's in another function. Instead, we must use the setjmp and longjmp functions to perform this type of branching. As we'll see, these two functions are useful for handling error conditions that occur in a deeply nested function call.

Consider the skeleton in Figure 7.9. It consists of a main loop that reads lines from standard input and calls the function do_line to process each line. This function then calls get_token to fetch the next token from the input line. The first token of a line is assumed to be a command of some form, and a switch statement selects each command. For the single command shown, the function cmd_add is called.

The skeleton in Figure 7.9 is typical for programs that read commands, determine the command type, and then call functions to process each command. Figure 7.10 shows what the stack could look like after cmd_add has been called.

```c
#include "apue.h"

#define TOK_ADD    5

void    do_line(char *);
void    cmd_add(void);
int     get_token(void);

int
main(void)
{
    char    line[MAXLINE];

    while (fgets(line, MAXLINE, stdin) != NULL)
        do_line(line);
    exit(0);
}

char    *tok_ptr;       /* global pointer for get_token() */

void
do_line(char *ptr)       /* process one line of input */
{
    int     cmd;

    tok_ptr = ptr;
    while ((cmd = get_token()) > 0) {
        switch (cmd) {  /* one case for each command */
        case TOK_ADD:
                cmd_add();
                break;
        }
    }
}

void
cmd_add(void)
{
    int     token;

    token = get_token();
    /* rest of processing for this command */
}

int
get_token(void)
{
    /* fetch next token from line pointed to by tok_ptr */
}
```

Figure 7.9 Typical program skeleton for command processing

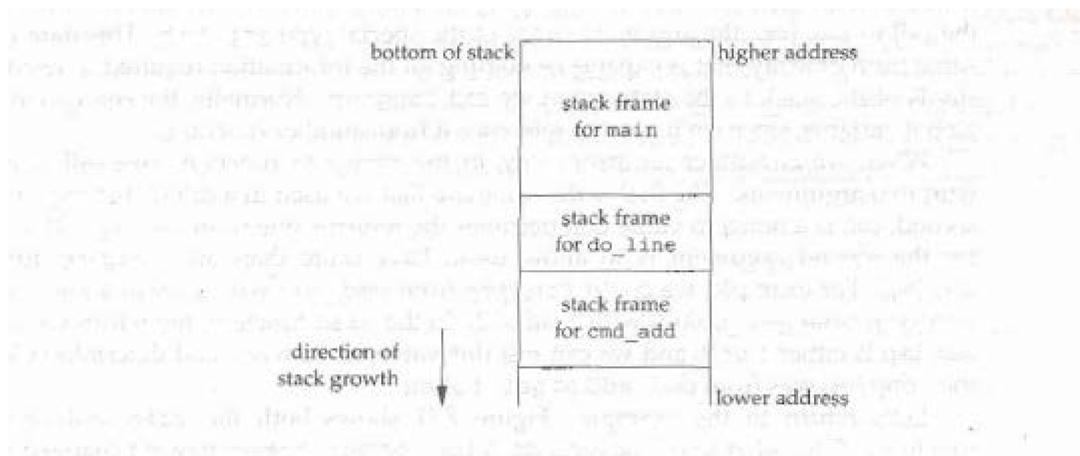


Figure 7.10 Stack frames after cmd_add has been called

Storage for the automatic variables is within the stack frame for each function. The array `line` is in the stack frame for `main`, the integer `cmd` is in the stack frame for `do_line`, and the integer `token` is in the stack frame for `cmd_add`.

As we've said, this type of arrangement of the stack is typical, but not required. Stacks do not have to grow toward lower memory addresses. On systems that don't have built-in hardware support for stacks, a C implementation might use a linked list for its stack frames.

The coding problem that's often encountered with programs like the one shown in Figure 7.9 is how to handle nonfatal errors. For example, if the `cmd_add` function encounters an error—say, an invalid number—it might want to print an error, ignore the rest of the input line, and return to the `main` function to read the next input line. But when we're deeply nested numerous levels down from the `main` function, this is difficult to do in C. (In this example, in the `cmd_add` function, we're only two levels down from `main`, but it's not uncommon to be five or more levels down from where we want to return to.) It becomes messy if we have to code each function with a special return value that tells it to return one level.

The solution to this problem is to use a nonlocal `goto`: the `setjmp` and `longjmp` functions. The adjective nonlocal is because we're not doing a normal C `goto` statement within a function; instead, we're branching back through the call frames to a function that is in the call path of the current function.

```
#include <setjmp.h>

int setjmp(jmp_buf env);

                    Returns: 0 if called directly, nonzero if returning from a call to longjmp

void longjmp(jmp_buf env, int val);
```

We call setjmp from the location that we want to return to, which in this example is in the main function. In this case, setjmp returns 0 because we called it directly. In the call to setjmp, the argument *env* is of the special type jmp_buf. This data type is some form of array that is capable of holding all the information required to restore the status of the stack to the state when we call longjmp. Normally, the *env* variable is a global variable, since we'll need to reference it from another function.

When we encounter an error—say, in the cmd_add function—we call longjmp with two arguments. The first is the same *env* that we used in a call to setjmp, and the second, *val*, is a nonzero value that becomes the return value from setjmp. The reason for the second argument is to allow us to have more than one longjmp for each setjmp. For example, we could longjmp from cmd_add with a *val* of 1 and also call longjmp from get_token with a *val* of 2. In the main function, the return value from setjmp is either 1 or 2, and we can test this value, if we want, and determine whether the longjmp was from cmd_add or get_token.

Let's return to the example. Figure 7.11 shows both the main and cmd_add functions. (The other two functions, do_line and get_token, haven't changed.)

```
#include "apue.h"
#include <setjmp.h>

#define TOK_ADD    5

jmp_buf jmpbuffer;

int
main(void)
{
    char    line[MAXLINE];

    if (setjmp(jmpbuffer) != 0)
        printf("error");
    while (fgets(line, MAXLINE, stdin) != NULL)
        do_line(line);
    exit(0);
}


void
cmd_add(void)
{
    int     token;

    token = get_token();
    if (token < 0)      /* an error has occurred */
        longjmp(jmpbuffer, 1);
    /* rest of processing for this command */
}
```

Figure 7.11 Example of setjmp and longjmp

When main is executed, we call setjmp, which records whatever information it needs to in the variable jmpbuffer and returns 0. We then call do_line, which calls cmd_add, and assume that an error of some form is detected. Before the call to longjmp in cmd_add, the stack looks like that in Figure 7.10. But longjmp causes the stack to be "unwound" back to the main function, throwing away the stack frames for cmd_add and do_line (Figure 7.12). Calling longjmp causes the setjmp in main to return, but this time it returns with a value of 1 (the second argument for longjmp).

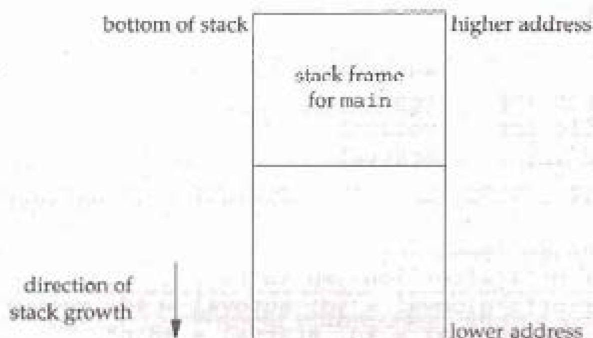


Figure 7.12 Stack frame after longjmp has been called

Automatic, Register, and Volatile Variables

We've seen what the stack looks like after calling longjmp. The next question is, "what are the states of the automatic variables and register variables in the main function?" When main is returned to by the longjmp, do these variables have values corresponding to when the setjmp was previously called (i.e., are their values rolled back), or are their values left alone so that their values are whatever they were when do_line was called (which caused cmd_add to be called, which caused longjmp to be called)? Unfortunately, the answer is "it depends." Most implementations do not try to roll back these automatic variables and register variables, but the standards say only that their values are indeterminate. If you have an automatic variable that you don't want rolled back, define it with the volatile attribute. Variables that are declared global or static are left alone when longjmp is executed.

Example

The program in Figure 7.13 demonstrates the different behavior that can be seen with automatic, global, register, static, and volatile variables after calling longjmp.

```c
#include "apue.h"
#include <setjmp.h>

static void f1(int, int, int, int);
static void f2(void);

static jmp_buf  jmpbuffer;
static int      globval;

int
main(void)
{
    int           autoval;
    register int  regival;
    volatile int  volaval;
    static int    statval;

    globval = 1; autoval = 2; regival = 3; volaval = 4; statval = 5;

    if (setjmp(jmpbuffer) != 0) {
        printf("after longjmp:\n");
        printf("globval = %d, autoval = %d, regival = %d,"
            " volaval = %d, statval = %d\n",
            globval, autoval, regival, volaval, statval);
        exit(0);
    }

    /*
     * Change variables after setjmp, but before longjmp.
     */
    globval = 95; autoval = 96; regival = 97; volaval = 98;
    statval = 99;

    f1(autoval, regival, volaval, statval); /* never returns */
    exit(0);
}

static void
f1(int i, int j, int k, int l)
{
    printf("in f1():\n");
    printf("globval = %d, autoval = %d, regival = %d,"
        " volaval = %d, statval = %d\n", globval, i, j, k, l);
    f2();
}

static void
f2(void)
{
    longjmp(jmpbuffer, 1);
}
```

Figure 7.13 Effect of longjmp on various types of variables

If we compile and test the program in Figure 7.13, with and without compiler optimizations, the results are different:

```
$ cc testjmp.c                    compile without any optimization
$ ./a.out
in f1():
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
after longjmp:
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
$ cc -O testjmp.c                 compile with full optimization
$ ./a.out
in f1():
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
after longjmp:
globval = 95, autoval = 2, regival = 3, volaval = 98, statval = 99
```

Note that the optimizations don't affect the global, static, and volatile variables; their values after the longjmp are the last values that they assumed. The setjmp(3) manual page on one system states that variables stored in memory will have values as of the time of the longjmp, whereas variables in the CPU and floating-point registers are restored to their values when setjmp was called. This is indeed what we see when we run the program in Figure 7.13. Without optimization, all five variables are stored in memory (the register hint is ignored for regival). When we enable optimization, both autoval and regival go into registers, even though the former wasn't declared register, and the volatile variable stays in memory. The thing to realize with this example is that you must use the volatile attribute if you're writing portable code that uses nonlocal jumps. Anything else can change from one system to the next.

Some printf format strings in Figure 7.13 are longer than will fit comfortably for display in a programming text. Instead of making multiple calls to printf, we rely on ISO C's string concatenation feature, where the sequence

```
"string1" "string2"
```

is equivalent to

```
"string1string2"
```

□

We'll return to these two functions, setjmp and longjmp, in Chapter 10 when we discuss signal handlers and their signal versions: sigsetjmp and siglongjmp.

Potential Problem with Automatic Variables

Having looked at the way stack frames are usually handled, it is worth looking at a potential error in dealing with automatic variables. The basic rule is that an automatic variable can never be referenced after the function that declared it returns. There are numerous warnings about this throughout the UNIX System manuals.

Figure 7.14 shows a function called open_data that opens a standard I/O stream and sets the buffering for the stream.

```
#include     <stdio.h>

#define DATAFILE    "datafile"

FILE *
open_data(void)
{
    FILE    *fp;
    char    databuf[BUFSIZ];  /* setvbuf makes this the stdio buffer */

    if ((fp = fopen(DATAFILE, "r")) == NULL)
        return(NULL);
    if (setvbuf(fp, databuf, _IOLBF, BUFSIZ) != 0)
        return(NULL);
    return(fp);        /* error */
}
```

Figure 7.14 Incorrect usage of an automatic variable

The problem is that when `open_data` returns, the space it used on the stack will be used by the stack frame for the next function that is called. But the standard I/O library will still be using that portion of memory for its stream buffer. Chaos is sure to result. To correct this problem, the array `databuf` needs to be allocated from global memory, either statically (`static` or `extern`) or dynamically (one of the `alloc` functions).

7.11 getrlimit and setrlimit Functions

Every process has a set of resource limits, some of which can be queried and changed by the `getrlimit` and `setrlimit` functions.

```
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlptr);

int setrlimit(int resource, const struct rlimit *rlptr);
```

Both return: 0 if OK, nonzero on error

These two functions are defined as XSI extensions in the Single UNIX Specification. The resource limits for a process are normally established by process 0 when the system is initialized and then inherited by each successive process. Each implementation has its own way of tuning the various limits.

Each call to these two functions specifies a single *resource* and a pointer to the following structure:

```
struct rlimit {
  rlim_t  rlim_cur;  /* soft limit: current limit */
  rlim_t  rlim_max;  /* hard limit: maximum value for rlim_cur */
};
```

Three rules govern the changing of the resource limits.

1. A process can change its soft limit to a value less than or equal to its hard limit.
2. A process can lower its hard limit to a value greater than or equal to its soft limit. This lowering of the hard limit is irreversible for normal users.
3. Only a superuser process can raise a hard limit.

An infinite limit is specified by the constant `RLIM_INFINITY`.

The *resource* argument takes on one of the following values. Figure 7.15 shows which limits are defined by the Single UNIX Specification and supported by each implementation.

`RLIMIT_AS`	The maximum size in bytes of a process's total available memory. This affects the `sbrk` function (Section 1.11) and the `mmap` function (Section 14.9).
`RLIMIT_CORE`	The maximum *size* in bytes of a core file. A limit of 0 prevents the creation of a core file.
`RLIMIT_CPU`	The maximum amount of CPU time in seconds. When the soft limit is exceeded, the `SIGXCPU` signal is sent to the process.
`RLIMIT_DATA`	The maximum size in bytes of the data segment: the sum of the initialized data, uninitialized data, and heap from Figure 7.6
`RLIMIT_FSIZE`	The maximum size in bytes of a file that may be created. When the soft limit is exceeded, the process is sent the `SIGXFSZ` signal.
`RLIMIT_LOCKS`	The maximum number of file locks a process can hold. (This number also includes file leases, a Linux-specific feature. See the Linux `fcntl`(2) manual page for more information.)
`RLIMIT_MEMLOCK`	The maximum amount of memory in bytes that a process can lock into memory using `mlock`(2).
`RLIMIT_NOFILE`	The maximum number of open files per process. Changing this limit affects the value returned by the `sysconf` function for its `_SC_OPEN_MAX` argument (Section 2.5.4). See Figure 2.16 also.
`RLIMIT_NPROC`	The maximum number of child processes per real user ID. Changing this limit affects the value returned for `_SC_CHILD_MAX` by the `sysconf` function (Section 2.5.4).
`RLIMIT_RSS`	Maximum resident set size (RSS) in bytes. If available physical memory is low, the kernel takes memory from processes that exceed their RSS.
`RLIMIT_SBSIZE`	The maximum size in bytes of socket buffers that a user can consume at any given time.
`RLIMIT_STACK`	The maximum size in bytes of the stack. See Figure 7.6.
`RLIMIT_VMEM`	This is a synonym for `RLIMIT_AS`.

Limit	XSI	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
RLIMIT_AS	•		•		•
RLIMIT_CORE	•	•	•	•	•
RLIMIT_CPU	•	•	•	•	•
RLIMIT_DATA	•	•	•	•	•
RLIMIT_FSIZE	•	•	•	•	•
RLIMIT_LOCKS			•		
RLIMIT_MEMLOCK		•	•	•	
RLIMIT_NOFILE	•	•	•	•	•
RLIMIT_NPROC		•	•	•	
RLIMIT_RSS		•	•	•	
RLIMIT_SBSIZE		•			
RLIMIT_STACK	•	•	•	•	•
RLIMIT_VMEM		•			•

Figure 7.15 Support for resource limits

The resource limits affect the calling process and are inherited by any of its children. This means that the setting of resource limits needs to be built into the shells to affect all our future processes. Indeed, the Bourne shell, the GNU Bourne-again shell, and the Korn shell have the built-in `ulimit` command, and the C shell has the built-in `limit` command. (The `umask` and `chdir` functions also have to be handled as shell built-ins.)

Example

The program in Figure 7.16 prints out the current soft limit and hard limit for all the resource limits supported on the system. To compile this program on all the various implementations, we have conditionally included the resource names that differ. Note also that we must use a different `printf` format on platforms that define `rlim_t` to be an `unsigned long long` instead of an `unsigned long`.

```
#include "apue.h"
#if defined(BSD) || defined(MACOS)
#include <sys/time.h>
#define FMT "%10lld  "
#else
#define FMT "%10ld  "
#endif
#include <sys/resource.h>

#define doit(name)  pr_limits(#name, name)

static void pr_limits(char *, int);

int
main(void)
{
```

```
#ifdef  RLIMIT_AS
    doit(RLIMIT_AS);
#endif
    doit(RLIMIT_CORE);
    doit(RLIMIT_CPU);
    doit(RLIMIT_DATA);
    doit(RLIMIT_FSIZE);
#ifdef  RLIMIT_LOCKS
    doit(RLIMIT_LOCKS);
#endif
#ifdef  RLIMIT_MEMLOCK
    doit(RLIMIT_MEMLOCK);
#endif
    doit(RLIMIT_NOFILE);
#ifdef  RLIMIT_NPROC
    doit(RLIMIT_NPROC);
#endif
#ifdef  RLIMIT_RSS
    doit(RLIMIT_RSS);
#endif
#ifdef  RLIMIT_SBSIZE
    doit(RLIMIT_SBSIZE);
#endif
    doit(RLIMIT_STACK);
#ifdef  RLIMIT_VMEM
    doit(RLIMIT_VMEM);
#endif
    exit(0);
}

static void
pr_limits(char *name, int resource)
{
    struct rlimit   limit;

    if (getrlimit(resource, &limit) < 0)
        err_sys("getrlimit error for %s", name);
    printf("%-14s  ", name);
    if (limit.rlim_cur == RLIM_INFINITY)
        printf("(infinite)  ");
    else
        printf(FMT, limit.rlim_cur);
    if (limit.rlim_max == RLIM_INFINITY)
        printf("(infinite)");
    else
        printf(FMT, limit.rlim_max);
    putchar((int)'\n');
}
```

Figure 7.16 Print the current resource limits

Note that we've used the ISO C string-creation operator (#) in the `doit` macro, to generate the string value for each resource name. When we say

```
doit(RLIMIT_CORE);
```

the C preprocessor expands this into

```
pr_limits("RLIMIT_CORE", RLIMIT_CORE);
```

Running this program under FreeBSD gives us the following:

```
$ ./a.out
RLIMIT_CORE      (infinite)  (infinite)
RLIMIT_CPU       (infinite)  (infinite)
RLIMIT_DATA       536870912   536870912
RLIMIT_FSIZE     (infinite)  (infinite)
RLIMIT_MEMLOCK   (infinite)  (infinite)
RLIMIT_NOFILE          1735        1735
RLIMIT_NPROC            867         867
RLIMIT_RSS       (infinite)  (infinite)
RLIMIT_SBSIZE    (infinite)  (infinite)
RLIMIT_STACK       67108864    67108864
RLIMIT_VMEM      (infinite)  (infinite)
```

Solaris gives us the following results:

```
$ ./a.out
RLIMIT_AS        (infinite)  (infinite)
RLIMIT_CORE      (infinite)  (infinite)
RLIMIT_CPU       (infinite)  (infinite)
RLIMIT_DATA      (infinite)  (infinite)
RLIMIT_FSIZE     (infinite)  (infinite)
RLIMIT_NOFILE           256       65536
RLIMIT_STACK        8388608  (infinite)
RLIMIT_VMEM      (infinite)  (infinite)
```

Exercise 10.11 continues the discussion of resource limits, after we've covered signals.

Q.7 a. Differentiate between the following: (4x2)

(i) kill and raise functions

(ii) alarm and pause functions

Answer:

10.9 kill and raise Functions

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.

> `raise` was originally defined by ISO C. POSIX.1 includes it to align itself with the ISO C standard, but POSIX.1 extends the specification of `raise` to deal with threads (we discuss how threads interact with signals in Section 12.8). Since ISO C does not deal with multiple processes, it could not define a function, such as `kill`, that requires a process ID argument.

```
#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 > 0 The signal is sent to the process whose process ID is *pid*.

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. Note that the term *all processes* excludes an implementation-defined set of system processes. For most UNIX systems, this set of system processes includes the kernel processes and `init` (pid 1).

pid < 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. Again, the set of all processes excludes certain system processes, as described earlier.

pid == –1 The signal is sent to all processes on the system for which the sender has permission to send the signal. As before, the set of processes excludes certain system processes.

As we've mentioned, a process needs permission to send a signal to another process. The superuser can send a signal to any process. For other users, the basic rule is that the real or effective user ID of the sender has to equal the real or effective user ID of the receiver. If the implementation supports `_POSIX_SAVED_IDS` (as POSIX.1 now requires), the saved set-user-ID of the receiver is checked instead of its effective user ID. There is also one special case for the permission testing: if the signal being sent is `SIGCONT`, a process can send it to any other process in the same session.

POSIX.1 defines signal number 0 as the null signal. If the *signo* argument is 0, then the normal error checking is performed by `kill`, but no signal is sent. This is often used to determine if a specific process still exists. If we send the process the null signal and it doesn't exist, `kill` returns –1 and `errno` is set to `ESRCH`. Be aware, however, that UNIX systems recycle process IDs after some amount of time, so the existence of a process with a given process ID does not mean that it's the process that you think it is.

Also understand that the test for process existence is not atomic. By the time that `kill` returns the answer to the caller, the process in question might have exited, so the answer is of limited value.

© IETE

If the call to kill causes the signal to be generated for the calling process and if the signal is not blacked, either s i p or some other pending, unblocked signal is delivered to the process before kill returns. (Additional conditions occur with threads; see Section 12.8 fox more information.)

10.10 alarm and pause Functions

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.

Earlier UNIX System implementations warned that the signal could also be sent up to 1 second early. POSIX.1 does not allow this.

There is only one of these alarm clocks per process. 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.

Example

Using alarm and pause, we can put a process to sleep for a specified amount of time. The sleep1 function in Figure 10.7 appears to do this (but it has problems, as we shall see shortly).

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

static void
sig_alrm(int signo)
{
    /* nothing to do, just return to wake up the pause */
}

unsigned int
sleep1(unsigned int nsecs)
{
    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        return(nsecs);
    alarm(nsecs);           /* start the timer */
    pause();                /* next caught signal wakes us up */
    return(alarm(0));       /* turn off timer, return unslept time */
}
```

Figure 10.7 Simple, incomplete implementation of sleep

This function looks like the sleep function, which we describe in Section 10.19, but this simple implementation has three problems.

1. If the caller already has an alarm set, that alarm is erased by the first call to alarm. We can correct this by looking at the return value from the first call to alarm. If the number of seconds until some previously set alarm is less than the argument, then we should wait only until the previously set alarm expires. If the previously set alarm will go off after ours, then before returning we should reset this alarm to occur at its designated time in the future.
2. We have modified the disposition for SIGALRM. If we're writing a function for others to call, we should save the disposition when we're called and restore it when we're done. We can correct this by saving the return value from signal and resetting the disposition before we return.
3. There is a race condition between the first call to alarm and the call to pause. On a busy system, it's possible for the alarm to go off and the signal handler to be called before we call pause. If that happens, the caller is suspended forever in the call to pause (assuming that some other signal isn't caught).

Earlier implementations of sleep looked like our program, with problems 1 and 2 corrected as described. There are two ways to correct problem 3. The first uses setjmp, which we show in the next example. The other uses sigprocmask and sigsuspend, and we describe it in Section 10.19. □

Example

The SVR2 implementation of `sleep` used `setjmp` and `longjmp` (Section 7.10) to avoid the race condition described in problem 3 of the previous example. A simple version of this function, called `sleep2`, is shown in Figure 10.8. (To reduce the size of this example, we don't handle problems 1 and 2 described earlier.)

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

static jmp_buf  env_alrm;

static void
sig_alrm(int signo)
{
    longjmp(env_alrm, 1);
}

unsigned int
sleep2(unsigned int nsecs)
{
        (signal(SIGALRM, sig_alrm) == SIG_ERR)
        return(nsecs);
    if (setjmp(env_alrm) == 0) {
        alarm(nsecs);           /* start the timer */
        pause();                /* next caught signal wakes us up */
    }
    return(alarm(0));           /* turn off timer, return unslept time */
}
```

Figure 10.8 Another (imperfect) implementation of `sleep`

The `sleep2` function avoids the race condition from Figure 10.7. Even if the `pause` is never executed, the `sleep2` function returns when the `SIGALRM` occurs.

There is, however, another subtle problem with the `sleep2` function that involves its interaction with other signals. If the `SIGALRM` interrupts some other signal handler, when we call `longjmp`, we abort the other signal handler. Figure 10.9 shows this scenario. The loop in the `SIGINT` handler was written so that it executes for longer than 5 seconds on one of the systems used by the author. We simply want it to execute longer than the argument to `sleep2`. The integer `k` is declared `volatile` to prevent an optimizing compiler from discarding the loop. Executing the program shown in Figure 10.9 gives us

```
$ ./a.out
^?                          we type the interrupt character
sig_int starting
sleep2 returned: 0
```

We can see that the `longjmp` from the `sleep2` function aborted the other signal handler, `sig_int`, even though it wasn't finished. This is what you'll encounter if you mix the SVR2 `sleep` function with other signal handling. See Exercise 10.3. □

```
#include "apue.h"

unsigned int    sleep2(unsigned int);
static void     sig_int(int);

int
main(void)
{
    unsigned int    unslept;

    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal(SIGINT) error");
    unslept = sleep2(5);
    printf("sleep2 returned: %u\n", unslept);
    exit(0);
}

static void
sig_int(int signo)
{
    int             i, j;
    volatile int    k;

    /*
     * Tune these loops to run for more than 5 seconds
     * on whatever system this test program is run.
     */
    printf("\nsig_int starting\n");
    for (i = 0; i < 300000; i++)
        for (j = 0; j < 4000; j++)
            k += i * j;
    printf("sig_int finished\n");
}
```

Figure 10.9 Calling `sleep2` from a program that catches other signals

The purpose of these two examples, the `sleep1` and `sleep2` functions, is to show the pitfalls in dealing naively with signals. The following sections will show ways around all these problems, so we can handle signals reliably, without interfering with other pieces of code.

Example

A common use for `alarm`, in addition to implementing the `sleep` function, is to put an upper time limit on operations that can block. For example, if we have a `read` operation on a device that can block (a "slow" device, as described in Section 10.5), we might want the `read` to time out after some amount of time. The program in Figure 10.10 does this, reading one line from standard input and writing it to standard output.

```
#include "apue.h"

static void sig_alrm(int);

int
main(void)
{
    int     n;
    char    line[MAXLINE];

    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        err_sys("signal(SIGALRM) error");

    alarm(10);
    if ((n = read(STDIN_FILENO, line, MAXLINE)) < 0)
        err_sys("read error");
    alarm(0);

    write(STDOUT_FILENO, line, n);
    exit(0);
}

static void
sig_alrm(int signo)
{
    /* nothing to do, just return to interrupt the read */
}
```

Figure 10.10 Calling read with a timeout

This sequence of code is common in UNIX applications, but this program has two problems.

1. The program in Figure 10.10 has one of the same flaws that we described in Figure 10.7: a race condition between the first call to `alarm` and the call to `read`. If the kernel blocks the process between these two function calls for longer than the alarm period, the `read` could block forever. Most operations of this type use a long alarm period, such as a minute or more, making this unlikely; nevertheless, it is a race condition.
2. If system calls are automatically restarted, the `read` is not interrupted when the `SIGALRM` signal handler returns. In this case, the timeout does nothing.

Here, we specifically want a slow system call to be interrupted. POSIX.1 does not give us a portable way to do this; however, the XSI extension in the Single UNIX Specification does. We'll discuss this more in Section 10.14. □

Example

Let's redo the preceding example using `longjmp`. This way, we don't need to worry about whether a slow system call is interrupted.

```
#include "apue.h"
#include <setjmp.h>

static void     sig_alrm(int);
static jmp_buf  env_alrm;

int
main(void)
{
    int     n;
    char    line[MAXLINE];

    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        err_sys("signal(SIGALRM) error");
    if (setjmp(env_alrm) != 0)
        err_quit("read timeout");

    alarm(10);
    if ((n = read(STDIN_FILENO, line, MAXLINE)) < 0)
        err_sys("read error");
    alarm(0);

    write(STDOUT_FILENO, line, n);
    exit(0);
}

static void
sig_alrm(int signo)
{
    longjmp(env_alrm, 1);
}
```

Figure 10.11 Calling `read` with a timeout, using `longjmp`

This version works as expected, regardless of whether the system restarts interrupted system calls. Realize, however, that we still have the problem of interactions with other signal handlers, as in Figure 10.8. □

If we want to set a time limit on an I/O operation, we need to use `longjmp`, as shown previously, realizing its possible interaction with other signal handlers. Another option is to use the `select` or `poll` functions, described in Sections 14.5.1 and 14.5.2.

b. Create a script file called file properties that reads a file name entered and output its properties. (8)

Answer:

```
echo "enter filename"
read file
c=1
if [ -e $file ]            #checks the existence of the file
then
     for i in `ls -l $file | tr -s " "`
     # 'tr -s " "' treats 2 or more spaces as a single space
     do
```

```
        case "$c" in              #case condition starts
         1)  echo "file permission=" $i ;;
         2)  echo "link =" $i;;
         3)  echo "file owner =" $i;;
         4)  echo "file group="$i ;;
         5)  echo "file size=" $i ;;
         6)  echo "file created month=" $i ;;
         7)  echo "file created date=" $i ;;
         8)  echo "last modified time=" $i ;;
         9)  echo "file name=" $i ;;
        esac                      #end of case condition
        c=`expr $c + 1`
    done
else
    echo "file does not exist"
fi
```

Output

```
$sh lab4a.sh
       enter filename
        lab8a.sh
       file permission=-rw-r- -r- -
       link=1
       file owner=hegde
       file group=hegde
       file size =339
       file created month=april
       file created date=7
       last modified time=05:19
       file name=lab8a.sh
```

Q.8 a. Your printer has stopped working, and you want to be sure whether the printing daemon is working. How will you ensure that? (4)

Answer:

Printing can be enabled in `lpc` using its `start` command and disabled using its `stop` command. Jobs held in a print queue when a printer is stopped will remain there until printing is restarted. The `stop` command functions by setting a lock file in the printer spool directory and killing the print daemon for that queue, but it allows the currently printing job to complete. The `abort` command works like `stop`, but halts

any printing job immediately, too. (Since the job did not complete, `lpr`retains it and starts over again when the queue is restarted.)

The `down` command functions as though both a `disable` and a `stop` command were issued, and the `up` command does the reverse, issuing `enable` and `start` commands.

You could also limit the display to one printer:

```
$ lpc status crow
crow:
        queuing is enabled
        printing is enabled
        1 entry in spool area
        crow is ready and printing
```

The status-reporting feature is useful for anyone, and `lpc` allows all users to use it. The real work for `lpc` usually involves solving a printing crisis. Sometimes a print daemon dies, and printing jobs back up. Sometimes a printer runs out of ink or paper, or even fails. Jobs in the print spools have to be suspended or moved to another spool where they can be printed. Someone may simply have an urgent printing task that needs to be moved to the top of the queue.

The `lpc` command is a classic Unix command: tight-lipped and forbidding. When you simply enter the `lpc` command, all you get back is a prompt:

```
lpc>
```

The command is interactive and waiting for your instructions. You can get help by entering `help` or a question mark at the`lpc` prompt. `lpc` responds and gives you a new prompt. For example, entering a question mark displays:

```
#  lpc
lpc> ?
Commands may be abbreviated.  Commands are:
abort   enable  disable help    restart status  topq    ?
clean   exit    down    quit    start   stop    up
lpc>
```

You can get additional help by asking for help about a specific command. For example, to learn more about restarting a stalled print queue, type:

```
lpc> help restart
restart         kill (if possible) and restart a spooling daemon
lpc>
```

The lpc help message does not offer online help about the secondary arguments you can specify in some places. The manual page will offer you some guidance. Most of the commands accept all or a print spool name as a secondary argument.

The lpc topq command recognizes a print spool name as the first argument and printer job numbers or user IDs as following arguments. The arguments are used to reorder the print queue. For example, to move job 237 to the top of the ada print queue, followed by all jobs owned by bckeller in the queue, enter:

```
lpc> topq ada 237 bckeller
```

The lpd daemon will start job 237 as soon as the current job is finished and will put any files in the queue owned by bckeller before the rest of the print spool. If you were very impatient, you could use the abort and clean commands to kill and purge the currently printing job, then use topq to put the job you want at the top of the queue, before usingrestart to create a new lpd and restart the queue.

When you use the stop command to stop a print spool (or all print spools) you can broadcast a message to all system users at the same time. For example:

```
lpc> stop ada "Printer Ada taken down to replace toner cartridge."
```

b. Write a shell script to display the period for which a given user has been working in the system. (8)

Answer:

```
/* In order to get the valid user names use the "who" command */

t1=`who | grep  "$1" | tr -s " " | cut -d " " -f 5 | cut -d ":"
-f 1 `

t2=`who | grep "$1" | tr -s " " | cut -d " " -f 5 | cut -d ":" -
f 2 `

t1=`expr $t1 \* 60 `

min1=`expr $t1 + $t2`

d1=`date +%H`

d2=`date +%M`

d1=`expr $d1 \* 60`

min2=`expr $d1 + $d2`
```

```
sub=`expr $min2 - $min1`

p=`expr $min2 - $min1`

p=`expr $p / 60`

p1=`expr $min2 - $min1`

p1=`expr $p1 % 60`

echo " The user  $1  has been working since : $pr Hrs $pr1
minutes "
```

Output

```
$sh 10a.sh mca30

        The user mca30 has been working since : 2 Hrs 30 minutes
```

c. Explain the command gzip with their syntax also. (4)

Answer:

gzip reduces the size of the named files using Lempel-Ziv coding (LZ77). Whenever possible, each file is replaced by one with the extension .gz, while keeping the same ownership modes, access, and modification times. (The default extension is -gz for VMS, z for MSDOS, OS/2 FAT, Windows NT FAT and Atari.) If no files are specified, or if a file name is "-", the standard input is compressed to the standard output. gzip will only attempt to compress regular files. In particular, it will ignore symbolic links.

If the compressed file name is too long for its file system, gzip truncates it. gzip attempts to truncate only the parts of the file name longer than 3 characters. (A part is delimited by dots.) If the name consists of small parts only, the longest parts are truncated. For example, if file names are limited to 14 characters, gzip.msdos.exe is compressed to gzi.msd.exe.gz. Names are not truncated on systems which do not have a limit on file name length.

By default, gzip keeps the original file name and timestamp in the compressed file. These are used when decompressing the file with the -N option. This is useful when the compressed file name was truncated or when the time stamp was not preserved after a file transfer.

Q.9 a. Differentiate between popen and pclose functions. (8)

Answer:

15.3 popen and pclose Functions

Since a common operation is to create a pipe to another process, to either read its output or send it input, the standard I/O library has historically provided the popen and pclose functions. These two functions handle all the dirty work that we've been doing ourselves: creating a pipe, forking a child, closing the unused ends of the pipe, executing a shell to run the command, and waiting for the command to terminate.

```
#include <stdio.h>

FILE *popen(const char *cmdstring, const char *type);

                              Returns: file pointer if OK, NULL on error

int pclose(FILE *fp);

              Returns: termination status of cmdstring, or -1 on error
```

The function popen does a fork and exec to execute the *cmdstring*, and returns a standard I/O file pointer. If *type* is "r", the file pointer is connected to the standard output of *cmdstring* (Figure 15.9).

Figure 15.9 Result of fp = popen(*cmdstring*, "r")

If *type* is "w", the file pointer is connected to the standard input of *cmdstring*, as shown in Figure 15.10.

Figure 15.10 Result of `fp = popen(cmdstring, "w")`

One way to remember the final argument to popen is to remember that, like fopen, the returned file pointer is readable if *type* is `"r"` or writable if *type* is `"w"`.

The `pclose` function closes the standard I/O stream, waits for the command to terminate, and returns the termination status of the shell. (We described the termination status in Section 8.6. The `system` function, described in Section 8.13, also returns the termination status.) If the shell cannot be executed, the termination status returned by `pclose` is as if the shell had executed `exit(127)`.

The *cmdstring* is executed by the Bourne shell, as in

sh -c *cmdstring*

This means that the shell expands any of its special characters in *cmdstring*. This allows us to say, for example,

```
fp = popen("ls *.c", "r");
```

or

```
fp = popen("cmd 2>&1", "r");
```

Example

Let's redo the program from Figure 15.6, using popen. This is shown in Figure 15.11.

```
#include "apue.h"
#include <sys/wait.h>

#define PAGER   "${PAGER:-more}" /* environment variable, or default */

int
main(int argc, char *argv[])
{
    char    line[MAXLINE];
    FILE    *fpin, *fpout;

    if (argc != 2)
        err_quit("usage: a.out <pathname>");
    if ((fpin = fopen(argv[1], "r")) == NULL)
        err_sys("can't open %s", argv[1]);

    if ((fpout = popen(PAGER, "w")) == NULL)
        err_sys("popen error");

    /* copy argv[1] to pager */
```

```
    while (fgets(line, MAXLINE, fpin) != NULL) {
        if (fputs(line, fpout) == EOF)
            err-aye("fputs error to pipe");
    }
    if (ferror(fpin))
        err_sys("fgets error");
    if (pclose(fpout) == -1)
        err_sys("pclose error");

    exit(0);
}
```

Figure 15.11 Copy file to pager program using popen

Using popen reduces the amount of code we have to write.

The shell command ${PAGER:-more} says to use the value of the shell variable PAGER if it is defined and non-null; otherwise, use the string more. □

Example—popen and pclose Functions

Figure 15.12 shows our version of popen and pclose.

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>
#include <sys/wait.h>

/*
 * Pointer to array allocated at run-time.
 */
static pid_t    *childpid = NULL;

/*
 * From our open_max(), Figure 2.16
 */
static int      maxfd;

FILE -
popen(const char *cmdstring, const char *type)
{
    int     i;
    int     pfd[2];
    pid_t   pid;
    FILE    *fp;

    /* only allow "r" or "w" */
    if ((type[0] != 'r' && type[0] != 'w') || type[1] != 0) {
        errno = EINVAL;     /* required by POSIX */
        return(NULL);
    }
```

```
    if (childpid == NULL) {     /* first time through */
        /* allocate zeroed out array for child pids */
        maxfd = open_max();
        if ((childpid = calloc(maxfd, sizeof(pid_t))) == NULL)
            return(NULL);
    }

    if (pipe(pfd) < 0)
        return(NULL);   /* errno set by pipe() */

    If ((pid = fork()) < 0) {
        return(NULL);   /* errno set by fork() */
    } else if (pid == 0) {                          /* child */
        If (*type == 'r') {
            close(pfd[0]);
            if (pfd[1] != STDOUT_FILENO) {
                dup2(pfd[1], STDOUT_FILENO);
                close(pfd[1]);
            }
        } else {
            close(pfd[1]);
            if (pfd[0] != STDIN_FILENO) {
                dup2(pfd[0], STDIN_FILENO);
                close(pfd[0]);
            }
        }

        /* close all descriptors in childpid[] */
        for (i = 0; i < maxfd; i++)
            if (childpid[i] > 0)
                close(i);

        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127);
    }

    /* parent continues... */
    if (*type == 'r') {
        close(pfd[1]);
        if ((fp = fdopen(pfd[0], type)) == NULL)
            return(NULL);
    } else {
        close(pfd[0]);
        if ((fp = fdopen(pfd[1], type)) == NULL)
            return(NULL);
    }

    childpid[fileno(fp)] = pid; /* remember child pid for this fd */
    return(fp);
}
```

```
int
pclose(FILE *fp)
{
    int     fd, stat;
    pid_t   pid;

    if (childpid == NULL) {
        errno = EINVAL;
        return(-1);      /* popen() has never been called */
    }

    fd = fileno(fp);
    if ((pid = childpid[fd]) == 0) {
        errno = EINVAL;
        return(-1);      /* fp wasn't opened by popen() */
    }
    childpid[fd] = 0;
    if (fclose(fp) == EOF)
        return(-1);

    while (waitpid(pid, &stat, 0) < 0)
        if (errno != EINTR)
            return(-1); /* error other than EINTR from waitpid() */

    return(stat);   /* return child's termination status */
}
```

Figure 15.12 The popen and pclose functions

Although the core of popen is similar to the code we've used earlier in this chapter, there are many details that we need to take care of. First, each time popen is called, we have to remember the process ID of the child that we create and either its file descriptor or FILE pointer. We choose to save the child's process ID in the array childpid, which we index by the file descriptor. This way, when pclose is called with the FILE pointer as its argument, we call the standard I/O function fileno to get the file descriptor, and then have the child process ID for the call to waitpid. Since it's possible for a given process to call popen more than once, we dynamically allocate the childpid array (the first time popen is called), with room for as many children as there are file descriptors.

Calling pipe and fork and then duplicating the appropriate descriptors for each process is similar to what we did earlier in this chapter.

POSIX.1 requires that popen close any streams that are still open in the child from previous calls to popen. To do this, we go through the childpid array in the child, closing any descriptors that are still open.

What happens if the caller of pclose has established a signal handler for SIGCHLD? The call to waitpid from pclose would return an error of EINTR. Since the caller is allowed to catch this signal (or any other signal that might interrupt the call to waitpid), we simply call waitpid again if it is interrupted by a caught signal.

Note that if the application calls `waitpid` and obtains the exit status of the child created by `popen`, we will call `waitpid` when the application calls `pclose`, find that the child no longer exists, and return −1 with `errno` set to `ECHILD`. This is the behavior required by POSIX.1 in this situation.

> Some early versions of `pclose` returned an error of `EINTR` if a signal interrupted the wait. Also, some early versions of `pclose` blocked or ignored the signals `SIGINT`, `SIGQUIT`, and `SIGHUP` during the `wait`. This is not allowed by POSIX.1.

Note that popen should never be called by a set-user-ID or set-group-ID program. When it executes the command, popen does the equivalent of

```
execl("/bin/sh", "sh", "-c", command, NULL);
```

which executes the shell and *command* with the environment inherited by the caller. A malicious user can manipulate the environment so that the shell executes commands other than those intended, with the elevated permissions granted by the set-ID file mode.

One thing that `popen` is especially well suited for is executing simple filters to transform the input or output of the running command. Such is the case when a command wants to build its own pipeline.

Example

Consider an application that writes a prompt to standard output and reads a line from standard input. With popen, we can interpose a program between the application and its input to transform the input. Figure 15.13 shows the arrangement of processes.

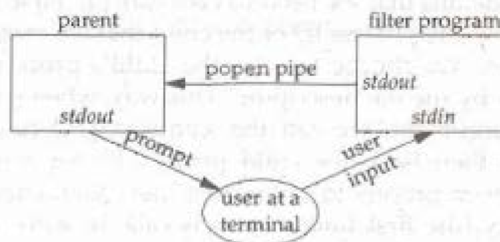


Figure 15.13 Transforming input using popen

The transformation could be pathname expansion, for example, or providing a history mechanism (remembering previously entered commands).

Figure 15.14 shows a simple filter to demonstrate this operation. The filter copies standard input to standard output, converting any uppercase character to lowercase. The reason we're careful to `fflush` standard output after writing a newline is discussed in the next section when we talk about coprocesses.

```
#include "apue.h"
#include <ctype.h>

int
main(void)
{
    int     c;

    while ((c = getchar()) != EOF) {
        if (isupper(c))
            c = tolower(c);
        if (putchar(c) == EOF)
            err-ays("output error");
        if (c == '\n')
            fflush(stdout);
    }
    exit(0);
}
```

Figure 15.14 Filter to convert uppercase characters to lowercase

We compile this filter into the executable file myuclc, which we then invoke from the program in Figure 15.15 using popen.

```
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    char    line[MAXLINE];
    FILE    *fpin;

    if ((fpin = popen("myuclc", "r")) == NULL)
        err_sys("popen error" ;
    for ( ; ; ) {
        fputs("prompt> ", stdout);
        fflush( k i t);
        if (fgets(line, MAXLINE, fpin) == NULL) /* read from pipe */
            break;
        if (fputs{line, stdout) == EOF)
            err_sys("fputs error to pipe");
    }
    if (pclose(fpin) == -1)
        err_sys("pclose error");
    putchar('\n');
    exit(0);
}
```

Figure 15.15 Invoke uppercase/lowercase filter to read commands

We need to call fflush after writing the prompt, because the standard output is normally line buffered, and the prompt does not contain a newline.

b. Write short notes on: **(4×2)**

i. shared memory

ii. client-server properties

Answer:

15.9 Shared Memory

Shared memory allows two or more processes to share a given region of memory. This is the fastest form of IPC, because the data does not need to be copied between the client and the server. The only trick in using shared memory is synchronizing access to a given region among multiple processes. If the server is placing data into a shared memory region, the client shouldn't try to access the data until the server is done. Often, semaphores are used to synchronize shared memory access. (But as we saw at the end of the previous section, record locking can also be used.)

The Single UNIX Specification includes an alternate set of interfaces to access shared memory in the shared memory objects option of its real-time extensions. We do not cover the real-time extensions in this text.

The kernel maintains a structure with at least the following members for each shared memory segment:

```
struct shmid_ds {
    struct ipc_perm  shm_perm;    /* see Section 15.6.2 */
    size_t           shm_segsz;   /* size of segment in bytes */
    pid_t            shm_lpid;    /* pid of last shmop() */
    pid_t            shm_cpid;    /* pid of creator */
    shmatt_t         shm_nattch;  /* number of current attaches */
    time_t           shm_atime;   /* last-attach time */
    time_t           shm_dtime;   /* last-detach time */
    time_t           shm_ctime;   /* last-change time */
    :
    :
};
```

(Each implementation adds other structure members as needed to support shared memory segments.)

The type `shmatt_t` is defined to be an unsigned integer at least as large as an `unsigned short`. Figure 15.30 lists the system limits (Section 15.6.3) that affect shared memory.

Description	Typical values			
	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
The maximum size in bytes of a shared memory segment	33,554,432	33,554,432	4,194,304	8,388,608
The minimum size in bytes of a shared memory segment	1	1	1	1
The maximum number of shared memory segments, systemwide	192	4,096	32	100
The maximum number of shared memory segments, per process	128	4,096	8	6

Figure 15.30 System limits that affect shared memory

The first function called is usually `shmget`, to obtain a shared memory identifier.

```
#include <sys/shm.h>

int shmget(key_t key, size_t size, int flag);
```

Returns: shared memory ID if OK, –1 on error

In Section 15.6.1, we described the rules for converting the *key* into an identifier and whether a new segment is created or an existing segment is referenced. When a new segment is created, the following members of the `shmid_ds` structure are initialized.

- The `ipc_perm` structure is initialized as described in Section 15.6.2. The `mode` member of this structure is set to the corresponding permission bits of *flag*. These permissions are specified with the values from Figure 15.24.
- `shm_lpid`, `shm_nattach`, `shm_atime`, and `shm_dtime` are all set to 0.
- `shm_ctime` is set to the current time.
- `shm_segsz` is set to the *size* requested.

© IETE

The *size* parameter is the size of the shared memory segment in bytes. Implementations will usually round up the size to a multiple of the system's page size, but if an application specifies *size* as a value other than an integral multiple of the system's page size, the remainder of the last page will be unavailable for use. If a new segment is being created (typically in the server), we must specify its *size*. If we are referencing an existing segment (a client), we can specify *size* as 0. When a new segment is created, the contents of the segment are initialized with zeros.

The shmctl function is the catchall for various shared memory operations.

```
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);

                                        Returns: 0 if OK, -1 on error
```

The cmd argument specifies one of the following five commands to be performed, on the segment specified by *shmid*.

IPC_STAT Fetch the shmid_ds structure for this segment, storing it in the structure pointed to by *buf*.

IPC_SET Set the following three fields from the structure pointed to by *buf* in the shmid_ds structure associated with this shared memory segment: shm_perm.uid, shm_perm.gid, and shm_perm.mode. This command can be executed only by a process whose effective user ID equals shm_perm.cuid or shm_perm.uid *or* by a process with superuser privilege.:.

IPC_RMID Remove the shared memory segment set from the system. Since an attachment count is maintained for shared memory segments (the shm_nattch field in the shmid_ds structure), the segment is not removed until the last process usrng the segment terminates or detaches it. Regardless of whether the segment is still in use, the segment's identifier is immediately removed so that shmat can no longer attach the segment. This command can be executed only by a process whose effective user ID equals shm_perm.cuid or shm_perm.uid or by a process with superuser privileges.

Two additional commands are provided by Linux and Solaris, but are not part of the Single UNIX Specificatinn.

SHM_LOCK Lock the shared memory segment in memory. This command can be executed only by the superuser.

SHM_UNLOCK Unlock the shared memory segment. This command can be executed only by the superuser.

Once a shared memory segment has been created, a process attaches it to its address space by calling shmat.

```
#include <sys/shm.h>

void *shmat(int shmid, const void *addr, int flag);

                 Returns: pointer to shared memory segment if OK, -1 on error
```

The address in the calling process at which the segment is attached depends on the *addr* argument and whether the `SHM_RND` bit is specified in *flag*.

- If *addr* is 0, the segment is attached at the first available address selected by the kernel. This is the recommended technique.
- If *addr* is nonzero and `SHM_RND` is not specified, the segment is attached at the address given by *addr*.
- If *addr* is nonzero and `SHM_RND` is specified, the segment is attached at the address given by (*addr* − (*addr* modulus `SHMLBA`)). The `SHM_RND` command stands for "round." `SHMLBA` stands for "low boundary address multiple" and is always a power of 2. What the arithmetic does is round the address down to the next multiple of `SHMLBA`.

Unless we plan to run the application on only a single type of hardware (which is highly unlikely today), we should not specify the address where the segment is to be attached. Instead, we should specify an *addr* of 0 and let the system choose the address.

If the `SHM_RDONLY` bit is specified in *flag*, the segment is attached read-only. Otherwise, the segment is attached read–write.

The value returned by `shmat` is the address at which the segment is attached, or -1 if an error occurred. If `shmat` succeeds, the kernel will increment the `shm_nattch` counter in the `shmid_ds` structure associated with the shared memory segment.

When we're done with a shared memory segment, we call `shmdt` to detach it. Note that this does not remove the identifier and its associated data structure from the system. The identifier remains in existence until some process (often a server) specifically removes it by calling `shmctl` with a command of `IPC_RMID`.

```
#include <sys/shm.h>

int shmdt(void *addr);

                                          Returns: 0 if OK, -1 on error
```

The *addr* argument is the value that was returned by a previous call to `shmat`. If successful, `shmdt` will decrement the `shm_nattch` counter in the associated `shmid_ds` structure.

Example

Where a kernel places shared memory segments that are attached with an address of 0 is highly system dependent. Figure 15.31 shows a program that prints some information on where one particular system places various types of data.

```
#include "apue.h"
#include <sys/shm.h>

#define ARRAY-SIZE  40000
#define MALLOC_SIZE 100000
#define SHM_SIZE    100000
#define SHM_MODE    0600    /* user read/write */

char    array[ARRAY_SIZE];  /* uninitialized data = bss */

int
main(void)
{
    int     shmid;
    char    *ptr, *shmptr;

    printf("array[] from %lx to %lx\n", (unsigned long)&array[0],
      (unsigned long)&array[ARRAY_SIZE]);
    printf("stack around %lx\n", (unsigned long)&shmid);

    if ((ptr = malloc(MALLOC_SIZE)) == NULL)
        err_sys("malloc error");
    printf("malloced from %lx to %lx\n", (unsigned long)ptr,
      (unsigned long)ptr+MALLOC_SIZE);

    if ((shmid = shmget(IPC_PRIVATE, SHM_SIZE, SHM_MODE)) < 0)
        err_sys("shmget error");
    if ((shmptr = shmat(shmid, 0, 0)) == (void *)-1)
        err_sys("shmat error");
    printf("shared memory attached from %lx to %lx\n",
      (unsigned long)shmptr, (unsigned long)shmptr+SHM_SIZE);

    if (shmctl(shmid, IPC_RMID, 0) < 0)
        err_sys("shmctl error");

    exit(0);
}
```

Figure 15.31 Print where various types of data are stored

Running this program on an Intel-based Linux system gives us the following output:

```
$ ./a.out
array[] from 804a080 to 8053cc0
stack arwnd bffff9e4
malloced from 8053cc8 to 806c368
shared memory attached from 40162000 to 4017a6a0
```

Figure 15.32 shows a picture of this, similar to what we said-was a typical memory layout in Figure 7.6. Note that the shared memory segment is placed well below the stack. □

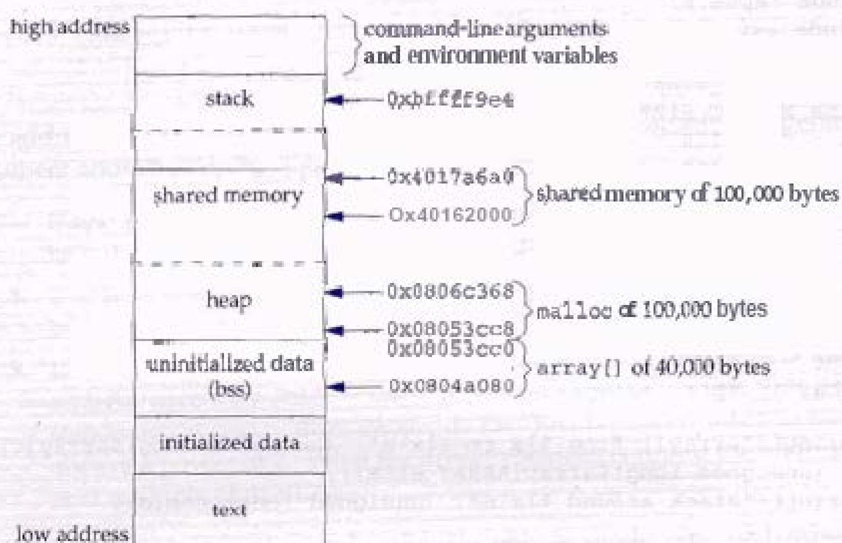


Figure 15.32 Memory layout on an Intel-based Linux system

Recall that the mmap function (Section 14.9) can be used to map portions of a file into the address space of a process. This is conceptually similar to attaching a shared memory segment using the shmat XSI IPC function. The main difference is that the memory segment mapped with mmap is backed by a file, whereas no file is associated with an XSI shared memory segment.

Example—Memory Mapping of /dev/zero

Shared memory can be used between unrelated processes. But if the processes are related, some implementations provide a different technique.

> The following technique works on FreeBSD 5.2.1, Linux 2.4.22, and Solaris 9. Mac OS X 10.3 currently doesn't support the mapping of character devices into the address space of a process.

The device /dev/zero is an infinite source of 0 bytes when read. This device also accepts any data that is written to it, ignoring the data. Our interest in this device for IPC arises from its special properties when it is memory mapped.

- An unnamed memory region is created whose size is the second argument to mmap, rounded up to the nearest page size on the system.
- The memory region is initialized to 0.
- Multiple processes can share this region if a common ancestor specifies the MAP_SHARED flag to mmap.

The program in Figure 15.33 is an example that uses this special device.

```
#include "apue.h"
#include <fcntl.h>
#include <sys/mman.h>

#define NLOOPS      1000
#define SIZE        sizeof(long)    /* size of shared memory area */

static int
update(long *ptr)
{
    return((*ptr)++);    /* return value before increment */
}

int
main(void)
{
    int     fd, i, counter;
    pid_t   pid;
    void    *area;

    if ((fd = open("/dev/zero", O_RDWR)) < 0)
        err_sys("open error");
    if ((area = mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED,
      fd, 0)) == MAP_FAILED)
        err_sys("mmap error");
    close(fd);      /* can close /dev/zero now that it's mapped */

    TELL_WAIT();

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {           /* parent */
        for (i = 0; i < NLOOPS; i += 2) {
            if ((counter = update((long *)area)) != i)
                err_quit("parent: expected %d, got %d", i, counter);

            TELL_CHILD(pid);
            WAIT_CHILD();
        }
    } else {                        /* child */
        for (i = 1; i < NLOOPS + 1; i += 2) {
            WALT-PARENT();

            if ((counter = update((long *)area)) != i)
                err_quit("child: expected %d, got %d", i, counter);

            TELL_PARENT(getppid());
        }
    }

    exit(0);
}
```

Figure 15.33 IPC between parent and child using memory mapped I/O of /dev/zero

The program opens the /dev/zero device and calls mmap, specifying a size of a long integer. Note that once the region is mapped, we can close the device. The process then creates a child. Since MAP_SHARED was specified in the call to mmap, writes to the memory-mapped region by one process are seen by the other process. (If we had specified MAP_PRIVATE instead, this example wouldn't work.)

The parent and the child then alternate running, incrementing a long integer in the shared memory-mapped region, using the synchronization functions from Section 8.9. The memory-mapped region is initialized to 0 by mmap. The parent increments it to 1, then the child increments it to 2, then the parent increments it to 3, and so on. Note that we have to use parentheses when we increment the value of the long integer in the update function, since we are incrementing the value and not the pointer.

The advantage of using /dev/zero in the manner that we've shown is that an actual file need not exist before we call mmap to create the mapped region. Mapping /dev/zero automatically creates a mapped region of the specified size. The disadvantage of this technique is that it works only between related processes. With related processes, however, it is probably simpler and more efficient to use threads (Chapters 11 and 12). Note that regardless of which technique is used, we still need to synchronize access to the shared data. □

Example —Anonymous Memory Mapping

Many implementations provide anonymous memory mapping, a facility similar to the /dev/zero feature. To use this facility, we specify the MAP_ANON flag to mmap and specify the file descriptor as −1. The resulting region is anonymous (since it's not associated with a pathname through a file descriptor) and creates a memory region that can be shared with descendant processes.

> The anonymous memory-mapping facility is supported by all four platforms discussed in this text. Note, however, that Linux defines the MAP_ANONYMOUS flag for this facility, but defines the MAP_ANON flag to be the same value for improved application portability.

To modify the program in Figure 15.33 to use this facility, we make three changes: (a) remove the open of /dev/zero, (b) remove the close of fd, and (c) change the call to mmap to the following:

```
if ((area = mmap(0, SIZE, PROT_READ | PROT_WRITE,
                  MAP_ANON | MAP_SHARED, -1, 0)) == MAP_FAILED)
```

In this call, we specify the MAP_ANON flag and set the file descriptor to −1. The rest of the program from Figure 15.33 is unchanged. □

The last two examples illustrate sharing memory among multiple *related* processes. If shared memory is required between unrelated processes, there are two alternatives. Applications can use the XSI shared memory functions, or they can use mmap to map the same file into their address spaces using the MAP_SHARED flag.

15.10 Client–Server Properties

Let's detail some of the properties of clients and servers that are affected by the various types of IPC used between them. The simplest type of relationship is to have the client `fork` and `exec` the desired server. Two half-duplex pipes can be created before the `fork` to allow data to be transferred in both directions. Figure 15.16 is an example of this. The server that is executed can be a set-user-ID program, giving it special privileges. Also, the server can determine the real identity of the client by looking at its real user ID. (Recall from Section 8.10 that the real user ID and real group ID don't change across an `exec`.)

With this arrangement, we can build an *open server*. (We show an implementation of this client–server in Section 17.5.) It opens files for the client instead of the client calling the `open` function. This way, additional permission checking can be added, above and beyond the normal UNIX system user/group/other permissions. We assume that the server is a set-user-ID program, giving it additional permissions (root permission, perhaps). The server uses the real user ID of the client to determine whether to give it access to the requested file. This way, we can build a server that allows certain users permissions that they don't normally have.

In this example, since the server is a child of the parent, all the server can do is pass back the contents of the file to the parent. Although this works fine for regular files, it can't be used for special device files, for example. We would like to be able to have the server open the requested file and pass back the file descriptor. Whereas a parent can pass a child an open descriptor, a child cannot pass a descriptor back to the parent (unless special programming techniques are used, which we cover in Chapter 17).

We showed the next type of server in Figure 15.23. The server is a daemon process that is contacted using some form of IPC by all clients. We can't use pipes for this type of client–server. A form of named IPC is required, such as FIFOs or message queues. With FIFOs, we saw that an individual per client FIFO is also required if the server is to send data back to the client. If the client–server application sends data only from the client to the server, a single well-known FIFO suffices. (The System V line printer spooler used this form of client–server arrangement. The client was the `lp(1)` command, and the server was the `lpsched` daemon process. A single FIFO was used, since the flow of data was only from the client to the server. Nothing was sent back to the client.)

Multiple possibilities exist with message queues.

1. A single queue can be used between the server and all the clients, using the type field of each message to indicate the message recipient. For example, the clients can send their requests with a type field of 1. Included in the request must be the client's process ID. The server then sends the response with the type field set to the client's process ID. The server receives only the messages with a type field of 1 (the fourth argument for `msgrcv`), and the clients receive only the messages with a type field equal to their process IDs.
2. Alternatively, an individual message queue can be used for each client. Before sending the first request to a server, each client creates its own message queue

with a kev of IPC PRIVATE. The server also has its own queue, with a key or identifier known to all clients. The client sends its first request to the server's well-known queue, and this request must contain the message queue ID of the client's queue. The server sends its first response to the client's queue, and all future requests and responses are exchanged on this queue.

One problem with this technique is that each client-specific queue usually has only a single message on it: a request for the server or a response for a client. This seems wasteful of a limited systemwide resource (a message queue), and a FIFO can be used instead. Another problem is that the server has to read messages from multiple queues. Neither select nor poll works with message queues.

Either of these two techniques using message queues can be implemented using shared memory segments and a synchronization method (a semaphore or record locking).

The problem with this type of client-server relationship (the client and the server being unrelated processes) is for the server to identify the client accurately. Unless the server is performing a nonprivileged operation, it is essential that the server know who the client is. This is required, for example, if the server is a set-user-ID program. Although all these forms of IPC go through the kernel, there is no facility provided by them to have the kernel identify the sender.

With message queues, if a single queue is used between the client and the server (so that only a single message is on the queue at a time, for example), the msg_lspid of the queue contains the process ID of the other process. But when writing the server, we want the effective user ID of the client, nut its process ID. There is no portable way to obtain the effective user ID, given the process ID. (Naturally, the kernel maintains both values in the process table entry, but other than rummaging around through the kernel's memory, we can't obtain one, given the other.)

We'll use the following technique in Section 17.3 to allow the server to identify the client. The same technique can be used with FIFOs, message queues, semaphores, or shared memory. For the following description, assume that FIFOs are being used, as in Figure 15.23. The client must create its own FIFO and set the file access permissions of the FIFO so that only user-read and user-write are on. We assume that the server has superuser privileges (or else it probably wouldn't care about the client's true identity), so the server can still read and write to this FIFO. When the server receives the client's first request on the server's well-known FIFO (which must contain the identity of the client-specific FIFO), the server calls either stat or fstat on the client-specific FIFO. The server assumes that the effective user ID of the client is the owner of the FIFO (the st_uid field of the stat structure). The server verifies that only the user-read and user-write permissions are enabled. As another check, the server should also look at the three times associated with the FIFO (the st_atime, st_mtime, and st_ctime fields of the stat structure) to verify that they are recent (no older than 15 or 30 seconds, for example). It a malicious client can create a FIFO with someone else as the owner and set the file's permission bits to user-read and user-write only, then the system has other fundamental security problems.

To use this technique with XSI IPC, recall that the ipc_perm structure associated with each message queue, semaphore, and shared memory segment identifies the creator of the IPC structure (the cuid and cgid fields). As with the example using FIFOs, the server should require the client to create the IPC structure and have the client set the access permissions to user-read and user-write only. The times associated with the IPC structure should also be verified by the server to be recent (since these IPC structures hang around until explicitly deleted).

We'll see in Section 17.2.2 that a far better way of doing this authentication is for the kernel to provide the effective user ID and effective group ID of the client. This is done by the STREAMS subsystem when file descriptors are passed between processes.

TEXT BOOK

I. Advanced Programming in the UNIX Environment, W. Richards Stevens, Pearson Education, 2004