System Programming

Prof. Chuan-Ju Wang
Dept. of Computer Science
University of Taipei

Race Conditions

- A race condition occurs when
  - multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run.
- A process that wants to wait for a child to terminate must call one of the **`wait`** functions.
- A process wants to wait for its parent to terminate, as in the program from Figure 8.8, a loop of the following form could be used.

```
while (getppid() != 1)
    sleep(1);
```

The problem with this type of loop, called polling.

Race Conditions

- To avoid race conditions and to avoid polling, some form of signaling is required between multiple processes.
  - E.g., the parent could update a record in a log file with the child's process ID, and the child might have to create a file for the parent. We require that each process tell the other when it has finished its initial set of operations, and that each wait for the other to complete, before heading off on its own.

```c
#include  "apue.h"

TELL_WAIT();    /* set things up for TELL_xxx & WAIT_xxx */

if ((pid = fork()) < 0) {
    err_sys("fork error");
} else if (pid == 0) {              /* child */

    /* child does whatever is necessary ... */

    TELL_PARENT(getppid());     /* tell parent we're done */
    WAIT_PARENT();              /* and wait for parent */

    /* and the child continues on its way ... */

    exit(0);
}

/* parent does whatever is necessary ... */

TELL_CHILD(pid);                /* tell child we're done */
WAIT_CHILD();                   /* and wait for child */

/* and the parent continues on its way ... */

exit(0);
```

Figure 8.12. Program with a race condition

```
#include "apue.h"

static void charatat

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()
        err_sys("fork
    } else if (pid ==
        charatatime("output from child\n");
    } else {
        charatatime("output from parent\n");
    }
    exit(0);
}

static void
charatatime(char *s
{
    char    *ptr;
    int     c;

    setbuf(stdout, NULL);           /* set unbuffered */
    for (ptr = str; (c = *ptr++) != 0; )
        putc(c, stdout);
}
```

```
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig8-12
output from parent
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] output from child

jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig8-12
output from parent
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] output from child

jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig8-12
pouuttppuutt  ffrroom m cphairledn
t
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig8-12
output from parent
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] output from child
```

```
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig8-12
output from parent
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] output from child

jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig8-12
output from paorent
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] utput from child
```

Figure 8.13. Modification of Figure 8.12 to avoid race condition

```
#include "apue.h"

static void charatatime(char *);

int
main(void)
{
    pid_t   pid;

+       TELL_WAIT();
+
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {
+           WAIT_PARENT();      /* parent goes first */
        charatatime("output from child\n");
    } else {
        charatatime("output from parent\n");
+           TELL_CHILD(pid);
    }
    exit(0);
}
static void
charatatime(char *str)
{
    char    *ptr;
    int     c;

    setbuf(stdout, NULL);           /* set unbuffered */
    for (ptr = str; (c = *ptr++) != 0; )
        putc(c, stdout);
}
```

```
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig8-13
output from parent
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] output from child

jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig8-13
output from parent
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] output from child
```

exec Functions

- There are six different exec functions.
- With fork, we can create new processes; and with the exec functions, we can initiate new programs.
- The exec family of functions are used to completely replace a running process with a new executable, and the new program starts executing at its **main** function.
- The process ID does not change across an exec because a new process is not created.
  - exec merely replaces the current process its text, data, heap, and stack segments with a brand new program from disk.

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0,
➥ ... /* (char *)0 */ );

int execv(const char *pathname, char *const argv []);

int execle(const char *pathname, const char *arg0, ...
            /* (char *)0,  char *const envp[] */ );

int execve(const char *pathname, char *const
➥  argv[], char *const envp []);

int execlp(const char *filename, const char *arg0,
➥  ... /* (char *)0 */ );

int execvp(const char *filename, char *const argv []);


                    All six return: 1 on error, no return on success
```

- If it has a **v** in its name, *argv* is a vector: **`char *const argv[]`**
- If it has an **l** in its name, *argv* is are a list: **`const char *arg0, ...s /* (char *) 0 */`**
- If it has an **e** in its name, it takes a **`char *const envp[]`** array of environment variables

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

- If it has a **p** in its name, it uses the `PATH` environment variable to search for the file

exec Functions

- Every system has a limit on the total size of the argument list and the environment list.
- On some systems, for example, the command

```
grep getrlimit /usr/share/man/*/*
```

can generate a shell error of the form.

```
Argument list too long
```

- **xargs(1)**

```
[root@www ~]# xargs [-0epn] command
選項與參數：
-0 ：如果輸入的 stdin 含有特殊字元，例如`,\, 空白鍵等等字元時，這個 -0 參數
     可以將他還原成一般字元。這個參數可以用於特殊狀態喔！
-e ：這個是 EOF (end of file) 的意思。後面可以接一個字串，當 xargs 分析到
     這個字串時，就會停止繼續工作！
-p ：在執行每個指令的 argument 時，都會詢問使用者的意思；
-n ：後面接次數，每次 command 指令執行時，要使用幾個參數的意思。看範例三。
當 xargs 後面沒有接任何的指令時，預設是以 echo 來進行輸出喔！

範例一：將 /etc/passwd 內的第一欄取出，僅取三行，使用 finger 這個指令將每個
     帳號內容秀出來
[root@www ~]# cut -d':' -f1 /etc/passwd |head -n 3| xargs finger
Login: root                             Name: root
Directory: /root                        Shell: /bin/bash
Never logged in.
No mail.
No Plan.
.....底下省略.....
# 由 finger account 可以取得該帳號的相關說明內容，例如上面的輸出就是 finger root
# 後的結果。在這個例子當中，我們利用 cut 取出帳號名稱，用 head 取出三個帳號，
# 最後則是由 xargs 將三個帳號的名稱變成 finger 後面需要的參數！
```

- **xargs(1)**

```
範例三：將所有的 /etc/passwd 內的帳號都以 finger 查閱，但一次僅查閱五個帳號
[root@www ~]# cut -d':' -f1 /etc/passwd | xargs -p -n 5 finger
finger root bin daemon adm lp ?...y
.....(中間省略)....
finger uucp operator games gopher ftp ?...y
.....(底下省略)....
# 在這裡鳥哥使用了 -p 這個參數來讓您對於 -n 更有概念。一般來說，某些指令後面
# 可以接的 arguments 是有限制的，不能無限制的累加，此時，我們可以利用 -n
# 來幫助我們將參數分成數個部分，每個部分分別再以指令來執行！這樣就 OK 啦！^_^

範例四：同上，但是當分析到 lp 就結束這串指令？
[root@www ~]# cut -d':' -f1 /etc/passwd | xargs -p -e'lp' finger
finger root bin daemon adm ?...
# 仔細與上面的案例做比較。也同時注意，那個 -e'lp' 是連在一起的，中間沒有空白鍵。
# 上個例子當中，第五個參數是 lp 啊，那麼我們下達 -e'lp' 後，則分析到 lp
# 這個字串時，後面的其他 stdin 的內容就會被 xargs 捨棄掉了！
```

```
find /usr/share/man -type f -print | xargs grep getrlimit
```

```
find /usr/share/man -type f -print | xargs bzgrep getrlimit
```

exec Functions

- Handling of open files
  - close-on-exec flag for each descriptor (`FD_CLOEXEC`)*
    - If this flag is set, the descriptor is closed across an `exec`.
    - Otherwise, the descriptor is left open across the `exec` (default).

* See the man page of **`fcntl(2)`**

exec Functions

- The real user ID and the real group ID remain the same across the exec.
- But the effective IDs can change.
  - set-user-ID/set- group-ID bits
  - If the set-user-ID bit is set for the new program, the effective user ID becomes the owner ID of the program file.
  - Otherwise, the effective user ID is not changed.
  - The group ID is handled in the same way.

exec Functions

Figure 8.15. Relationship of the six exec functions

[View full size image]

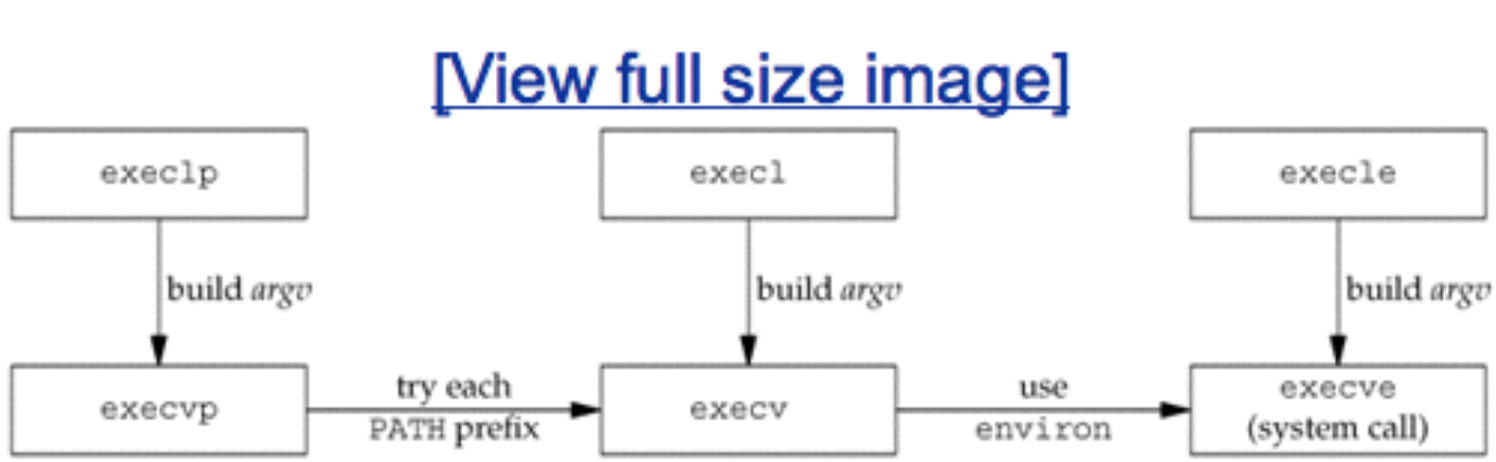


Figure 8.16. Example of exec functions

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

char    *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {  /* specify pathname, specify environment */
        if (execle("/home/sar/bin/echoall", "echoall", "myarg1",
                "MY ARG2", (char *)0, env_init) < 0)
            err_sys("execle error");
    }

    if (waitpid(pid, NULL, 0) < 0)
        err_sys("wait error");

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {  /* specify filename, inherit environment */
        if (execlp("echoall", "echoall", "only 1 arg", (char *)0) < 0)
            err_sys("execlp error");
    }

    exit(0);
}
```

Figure 8.17. Echo all command-line arguments and all environment strings

```
#include "apue.h"

int
main(int argc, char *argv[])
{
    int         i;
    char        **ptr;
    extern char **environ;

    for (i = 0; i < argc; i++)      /* echo all command-line args */
        printf("argv[%d]: %s\n", i, argv[i]);

    for (ptr = environ; *ptr != 0; ptr++)   /* and all env strings */
        printf("%s\n", *ptr);

    exit(0);
}
```

```
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig8-16
argv[0]: echoall
argv[1]: myarg1
argv[2]: MY ARG2
USER=unknown
PATH=/tmp;/home/jere/SystemProgramming/apue.2e/test/bin
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] argv[0]: echoall
argv[1]: only 1 arg
SSH_AGENT_PID=1457
GPG_AGENT_INFO=/tmp/keyring-pPNXAg/gpg:0:1
TERM=xterm
SHELL=/bin/bash
```

- How to make Figs. 8.16 and 8.17 run?
  - Left to be your homework.

Changing User IDs and Group IDs

- In the UNIX System, privileges are based on user and group IDs.
  - Use the least-privilege model when we design our applications.
    - Reduce the likelihood that security can be compromised by a malicious user.

Changing User IDs and Group IDs

```
#include <unistd.h>

int setuid(uid_t uid);

int setgid(gid_t gid);
```

Both return: 0 if OK, 1 on error

- Rules for who can change the IDs:
  1. If the process has superuser privileges, the `setuid` function sets the real user ID, effective user ID, and saved set-user-ID to *`uid`*.
  2. If the process does not have superuser privileges, but *`uid`* equals either the real user ID or the saved set-user-ID, `setuid` sets only the effective user ID to *`uid`*. The real user ID and the saved set-user-ID are not changed.
  3. If neither of these two conditions is true, `errno` is set to `EPERM`, and 1 is returned.

Changing User IDs and Group IDs

- Three user IDs that the kernel maintains:
  - Only a superuser process can change the real user ID.
    - Normally, the real user ID is set by the `login(1)` program and never changes. Because `login` is a superuser process, it sets all three user IDs when it calls `setuid`.
  - The effective user ID is set by the `exec` functions only if the set-user-ID bit is set.
  - The saved set-user-ID is copied from the effective user ID by `exec`.
    - If the file's set-user-ID bit is set, this copy is saved after `exec` stores the effective user ID from the file's user ID.

Changing User IDs and Group IDs

Figure 8.18. Ways to change the three user IDs

ID	exec		setuid(*uid*)	
	set-user-ID bit off	set-user-ID bit on	superuser	unprivileged user
real user ID	unchanged	unchanged	set to *uid*	unchanged
effective user ID	unchanged	set from user ID of program file	set to *uid*	set to *uid*
saved set-user ID	copied from effective user ID	copied from effective user ID	set to *uid*	unchanged

Utility of the Saved Set-User-ID

- **`man(1)`**
  - Might have to execute several other commands to process the files containing the manual page to be displayed.
  - To prevent being tricked into running the wrong commands or overwriting the wrong files, the `man` command has to switch between two sets of privileges:
    1. those of the user running the **`man`** command
    2. those of the user that owns the **`man`** executable file

Utility of the Saved Set-User-ID

1. Assume that the **`man`** is owned by the user name **`man`** and has its set-user-ID bit set. When we **`exec`** it,

```
real user ID = our user ID
effective user ID = man
saved set-user-ID = man
```

2. The **`man`** program accesses the required configuration files and manual pages.

3. Before **`man`** runs any command on our behalf, it calls `setuid(getuid())`.

```
real user ID = our user ID (unchanged)
effective user ID = our user ID
saved set-user-ID = man (unchanged)
```

This means that we can access only the files to which we have normal access.

Utility of the Saved Set-User-ID

- When the filter is done, **`man`** calls `setuid(`*`euid`*`)`, where *`euid`* is the user ID for the user name **`man`**.
  - This call is allowed because the argument to `setuid` equals the saved set-user-ID.

```
      real user ID = our user ID (unchanged)
 effective user ID = man
saved set-user-ID = man (unchanged)
```

- The **`man`** program can now operate on its files again.

Other Functions

- Swapping of the real user ID and the effective user ID

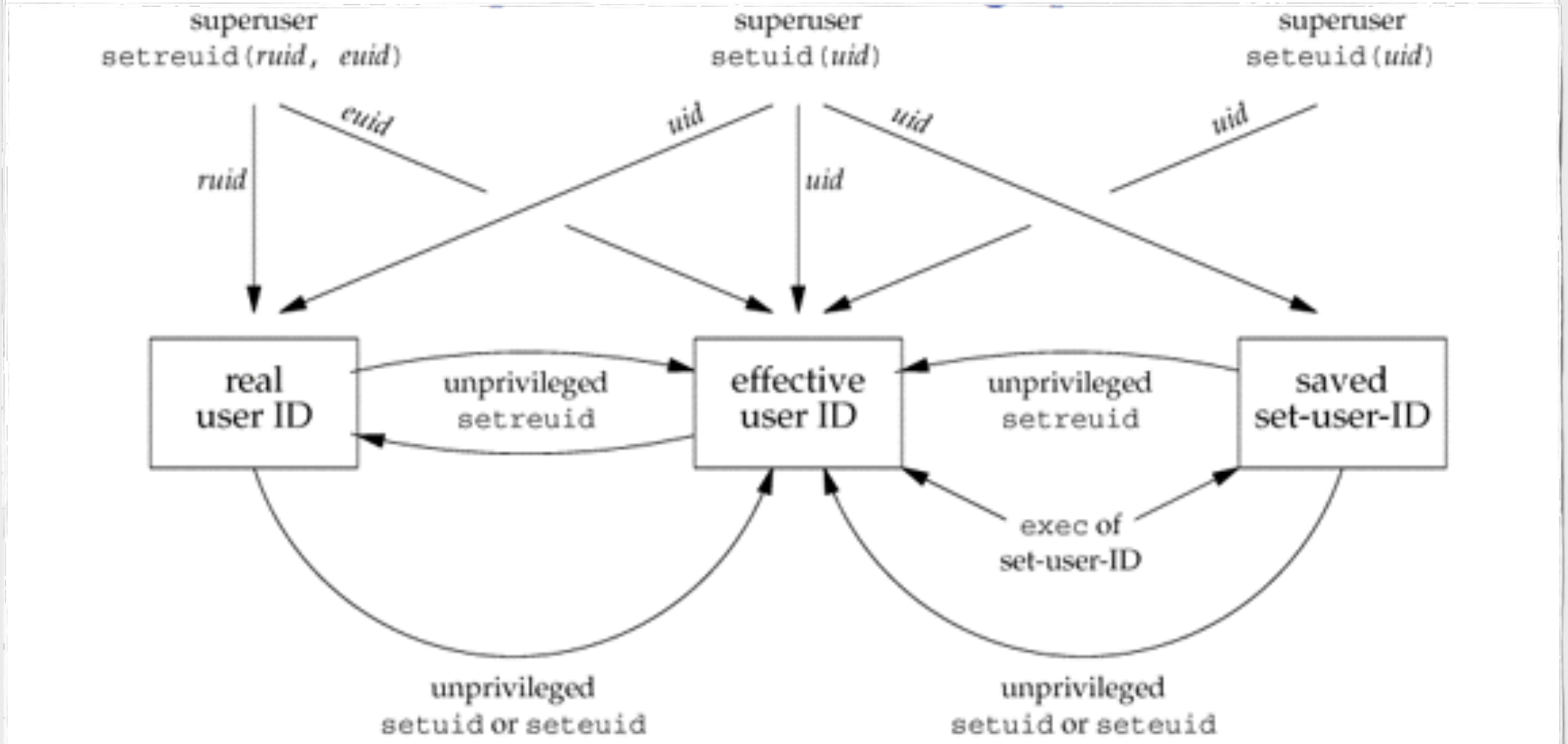
```
#include <unistd.h>

int setreuid(uid_t ruid, uid_t euid);

int setregid(gid_t rgid, gid_t egid);


                    Both return: 0 if OK, 1 on error
```

- Similar to `setuid` and `setgid`, but only the effective user ID or effective group ID is changed.

```
#include <unistd.h>

int seteuid(uid_t uid);

int setegid(gid_t gid);


                    Both return: 0 if OK, 1 on error
```

All the Functions

Figure 8.19. Summary of all the functions that set the various user IDs



Interpreter Files

- All contemporary UNIX systems support interpreter files.
  - Begin with a line of the form

```
#!  pathname [ optional-argument ]
```

- The most common one is

```
#!/bin/sh
```

The pathname is normally an absolute pathname

Interpreter Files

Figure 8.20. A program that `exec`s an interpreter file

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

int
main(void)
{
    pid_t   pid;
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {          /* child */
        if (execl("/home/sar/bin/testinterp",
                  "testinterp", "myarg1", "MY ARG2", (char *)0) < 0)
            err_sys("execl error");
    }
    if (waitpid(pid, NULL, 0) < 0) /* parent */
        err_sys("waitpid error");
    exit(0);
}
```

```
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig8-20
argv[0]: /home/jere/SystemProgramming/apue.2e/test/bin/echoarg
argv[1]: foo
argv[2]: /home/jere/SystemProgramming/apue.2e/test/testinterp
argv[3]: myarg1
argv[4]: MY ARG2
```

- How to make Fig. 8.20 run?
  - Left to be your homework.

`system` Function

- It is convenient to execute a command string within a program.
  - If we want to put a time-and-date stamp into a certain file, use functions `time`, `localtime`, `strftime`...
  - Much easier way:

```
system("date > file");
```

```
#include <stdlib.h>

int system(const char *cmdstring);

                                Returns: (see below)
```

`system` Function

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

  1. If either the `fork` fails or `waitpid` returns an error other than `EINTR`, system returns `1` with `errno` set to indicate the error.

  2. If the `exec` fails, implying that the shell can't be executed, the return value is as if the shell had executed `exit(127)`.

  3. Otherwise, all three functions `fork`, `exec`, and `waitpid` succeed, and the return value from system is the termination status of the shell.

Figure 8.22. The `system` function, without signal handling

```
#include    <sys/wait.h>
#include    <errno.h>
#include    <unistd.h>

int
system(const char *cmdstring)    /* version without signal handling */
{
    pid_t   pid;
    int     status;

    if (cmdstring == NULL)
        return(1);      /* always a command processor with UNIX */

    if ((pid = fork()) < 0) {
        status = -1;    /* probably out of processes */
    } else if (pid == 0) {              /* child */
        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127);     /* execl error */
    } else {                                /* parent */
        while (waitpid(pid, &status, 0) < 0) {
            if (errno != EINTR) {
                status = -1; /* error other than EINTR from waitpid() */
                break;
            }
        }
    }

    return(status);
}
```

Figure 8.23. Calling the `system` function

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

int
main(void)
{
    int      status;

    if ((status = system("date")) < 0)
        err_sys("system() error");
    pr_exit(status);

    if ((status = system("nosuchcommand")) < 0)
        err_sys("system() error");
    pr_exit(status);

    if ((status = system("who; exit 44")) < 0)
        err_sys("system() error");
    pr_exit(status);

    exit(0);
}
```

```
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig8-23
Sat Mar 17 21:10:09 CST 2012
normal termination, exit status = 0
sh: nosuchcommand: not found
normal termination, exit status = 127
jere     pts/0        2012-03-12 22:08 (:0)
normal termination, exit status = 44
```

Figure 8.24. Execute the command-line argument using `system`

```
#include "apue.h"

int
main(int argc, char *argv[])
{
    int     status;

    if (argc < 2)
        err_quit("command-line argument required");

    if ((status = system(argv[1])) < 0)
        err_sys("system() error");
    pr_exit(status);

    exit(0);
}
```

Figure 8.25. Print real and effective user IDs

```
#include "apue.h"

int
main(void)
{
    printf("real uid = %d, effective uid = %d\n", getuid(), geteuid());
    exit(0);
}
```

```
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] make fig8-24
gcc -DLINUX -ansi -I/home/jere/SystemProgramming/apue.2e/include -Wall -D_GNU_S
 -lapue  -o fig8-24
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] make fig8-25
gcc -DLINUX -ansi -I/home/jere/SystemProgramming/apue.2e/include -Wall -D_GNU_S
 -lapue  -o fig8-25
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] mv fig8-24 tsys
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] mv fig8-25 printuids
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./tsys printuids
sh: printuids: not found
normal termination, exit status = 127
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./tsys ./printuids
real uid = 1000, effective uid = 1000
normal termination, exit status = 0
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] sudo chown root tsys
[sudo] password for jere:
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] sudo chmod u+s tsys
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ls -l tsys
-rwsrwxr-x 1 root jere 7879 2012-03-17 21:14 tsys
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./tsys ./printuids
real uid = 1000, effective uid = 0
normal termination, exit status = 0
```

Assignment

- Reading (do it at home):
  - Read and try: http://linux.vbird.org/linux_basic/0440processcontrol.php
  - Manual pages for the functions covered
  - Stevens Chap. 8

Assignment

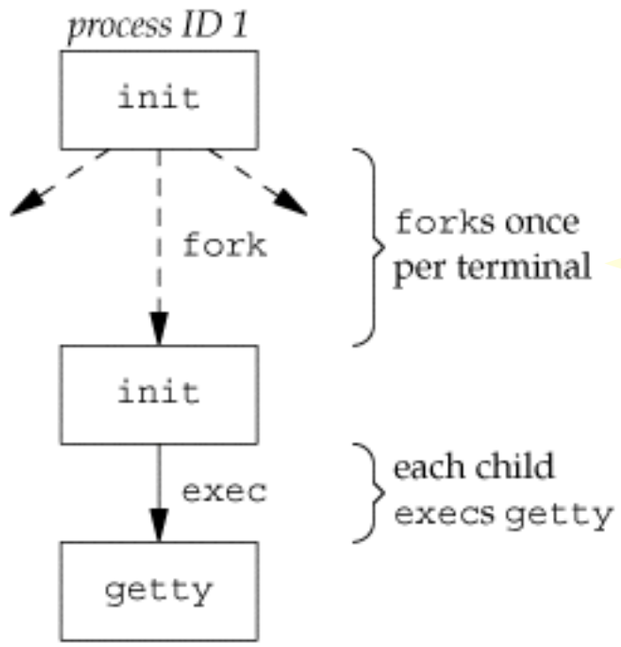
- Assignment 7
  1. How can we kill a zombie process?
     - ➡ A report should be handed in.
  2. Write a report to explain how you make Fig. 8.16(8.17) and Fig. 8.20 run?
     - ➡ A report with detailed steps you did should be handed in.
  3. Coding:
     - Write a program that creates a zombie, and then call `system` to execute the `ps(1)` command to verify that the process is a zombie.
     - ➡ A program with brief report should be handed in.
       - There are many ways to create zombie programs. Please describe at least one additional way instead of that written in your program in the report.
       - Also, please discuss plausible ways to avoid creating zombie programs in your report.

Process Relationships

Terminal Logins

- BSD Terminal Logins

Figure 9.1. Processes invoked by `init` to allow terminal logins



- All the processes have a real user ID of 0 and an effective user ID of 0 (i.e., they all have superuser privileges).

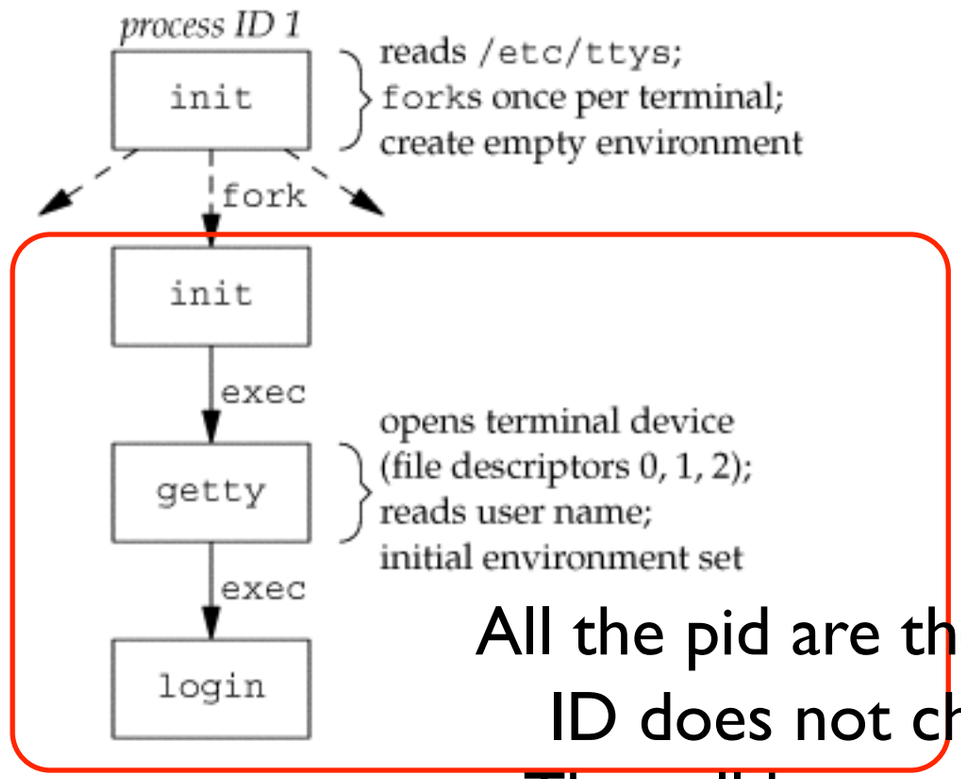
Terminal Logins

- It is `getty` that calls open for the terminal device.
- Once the device is open, file descriptors 0, 1, and 2 are set to the device.
- Then `getty` outputs something like login:

```
execle("/bin/login", "login", "-p", username, (char *)0, envp);
```

Terminal Logins

Figure 9.2. State of processes after `login` has been invoked



All the pid are the same since the process ID does not change across an exec. They all have a parent process ID of 1.

- All the processes have superuser privileges, since the original `init` process has superuser privileges.

Terminal Logins

- The login program does many things.
  - Call `getpwnam(3)` to fetch our password file entry
  - Call `getpass(3)` to display the prompt Password: and read our password
  - Call `crypt(3)` to encrypt the password that we entered and compares the encrypted result to the `pw_passwd` field from our shadow password file entry.
  - ...

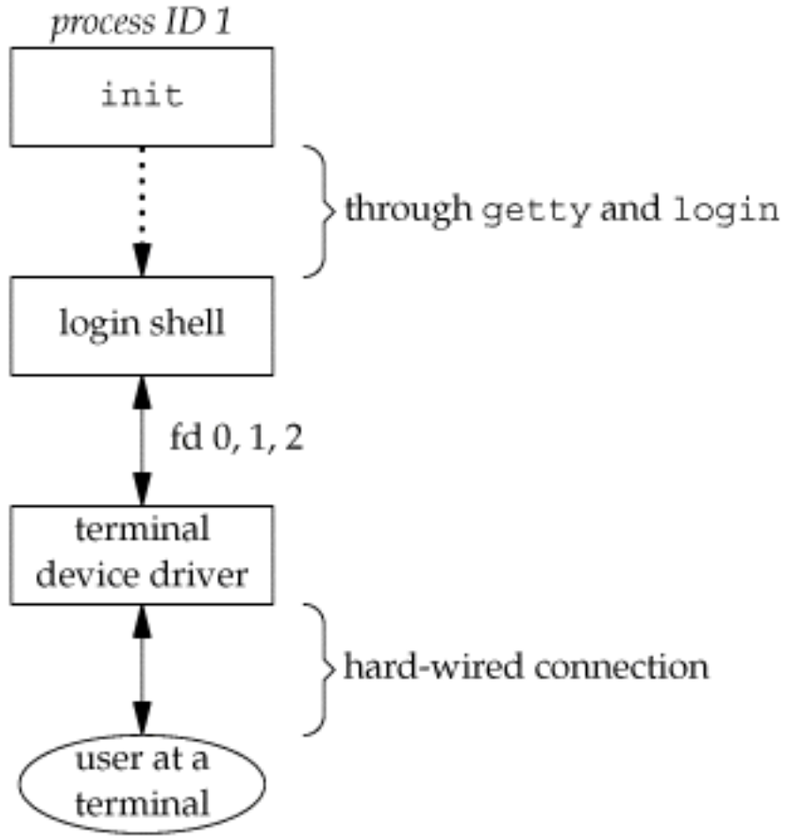
Terminal Logins

- If we log in correctly, login will
  - Change to our home directory (`chdir`)
  - Change the ownership of our terminal device (`chown`) so we own it
  - Change the access permissions for our terminal device so we have permission to read from and write to it
  - Set our group IDs by calling `setgid` and `initgroups`
  - Initialize the environment with all the information that login has: our home directory (`HOME`), shell (`SHELL`), user name (`USER` and `LOGNAME`), and a default path (`PATH`)
  - Change to our user ID (`setuid`) and invoke our login shell, as in

```
execl("/bin/sh", "-sh", (char *)0);
```

Terminal Logins

Figure 9.3. Arrangement of processes after everything is set for a terminal login



Terminal Logins

- Mac OS X Terminal Logins
  - The terminal login process follows the same steps as in the BSD login process.
  - A graphical-based login screen from the start instead
- Linux Terminal Logins
  - Very similar to the BSD procedure
  - The main difference between the BSD login procedure and the Linux login procedure is in the way the terminal configuration is specified.
    - `/etc/inittab`

Network Logins

- The main (physical) difference between logging in to a system through a serial terminal and a system through a network is that the connection between the terminal and the computer isn't point-to-point.
- In this case, login is simply a service available, just like any other network service, such as FTP or SMTP.

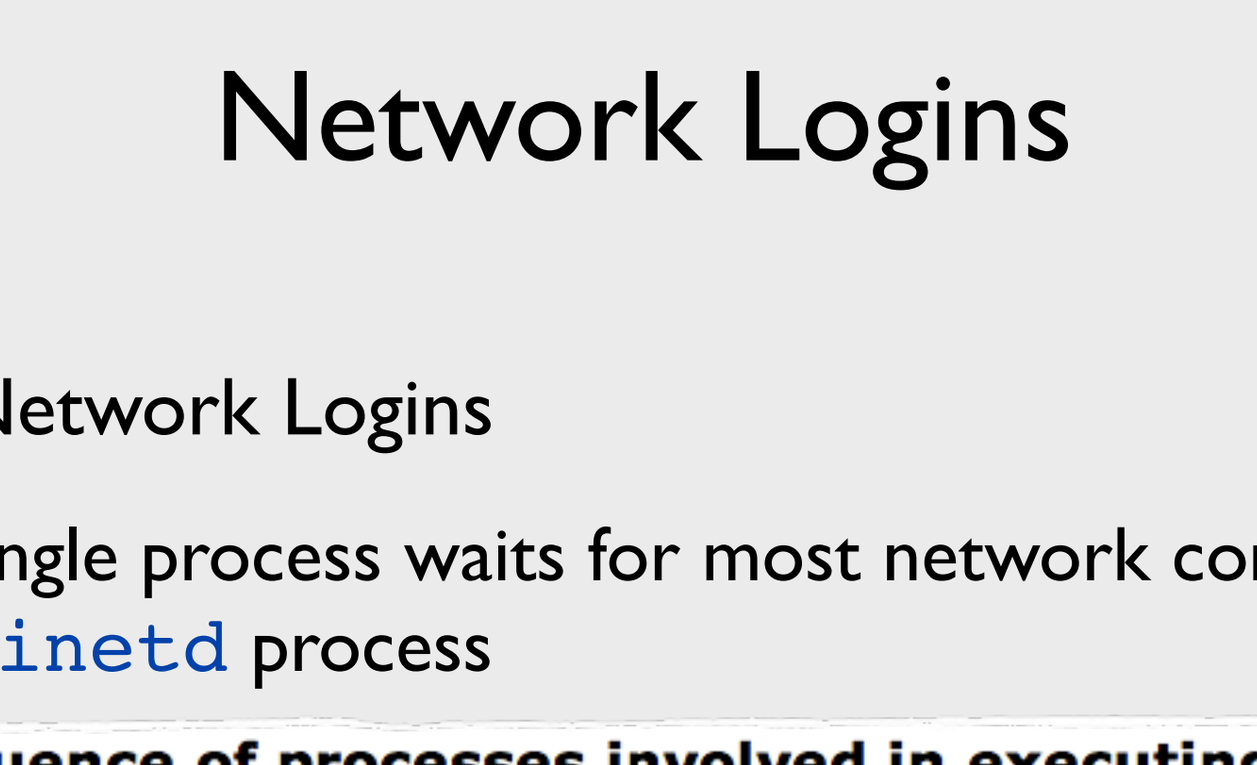
Network Logins

- With the terminal logins, `init` knows which terminal devices are enabled for logins.
- In the case of network logins, however, we don't know ahead of time how many of these will occur.
    - Instead of having a process waiting for each possible login, we now have to wait for a network connection request to arrive.
- A software driver called a *pseudo terminal* is used to emulate the behavior of a serial terminal.
    - To allow the same software to process logins over both terminal logins and network logins

Network Logins

- BSD Network Logins
  - A single process waits for most network connections: the `inetd` process

Figure 9.4. Sequence of processes involved in executing TELNET server

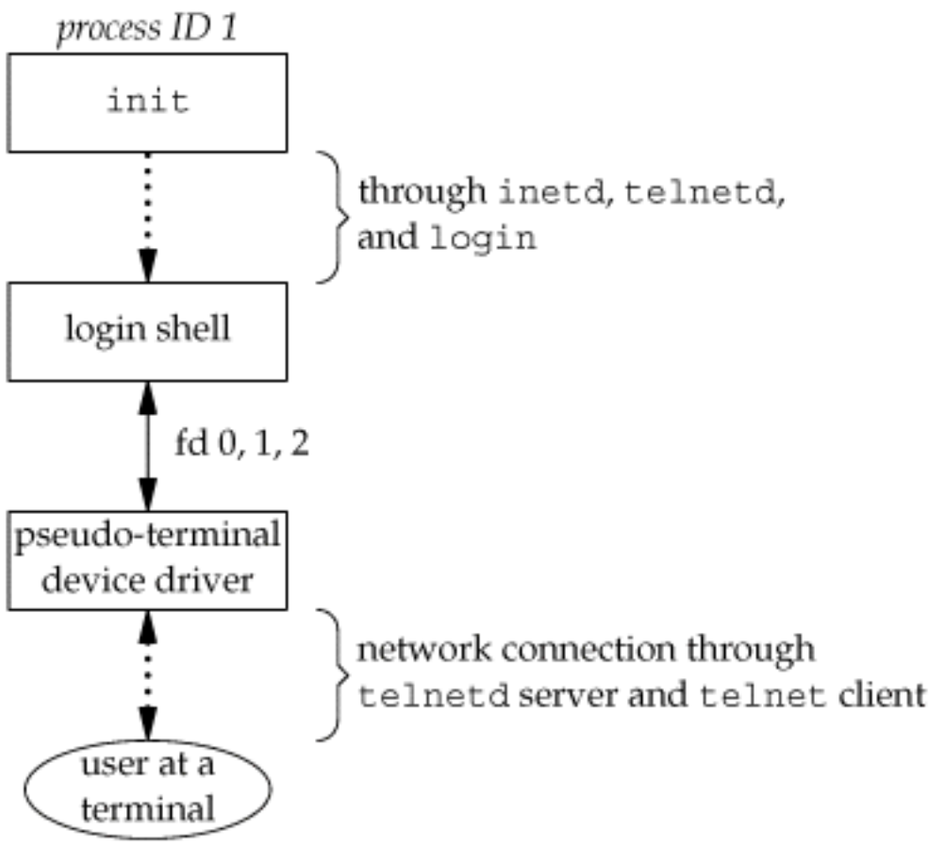


Network Logins

- The `telnetd` process then opens a pseudo-terminal device and splits into two processes using `fork`.
  - The parent handles the communication across the network connection.
  - The child does an `exec` of the login program.
  - The parent and the child are connected through the pseudo terminal.
  - Before doing the `exec`, the child sets up file descriptors 0, 1, and 2 to the pseudo terminal.

Network Logins

Figure 9.5. Arrangement of processes after everything is set for a network login



Process Groups

- Each process also belongs to a process group.
  - A process group is a collection of one or more processes, usually associated with the same job
  - Each process group has a unique process group ID.
  - Process group IDs (like PIDs) are positive integers and can be stored in a `pid_t` data type.

```
#include <unistd.h>

pid_t getpgrp(void);


  Returns: process group ID of calling process
```

Process Groups

- Each process group can have a process group leader.
  - Leader identified by its process group ID == PID.
  - Leader can create a new process group, create processes in the group.

Process Groups

- A process joins an existing process group or creates a new process group by calling `setpgid`.

```
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
```

Returns: 0 if OK, 1 on error

- Sets the process group ID to *`pgid`* in the process whose process ID equals *`pid`*.
- If the two arguments are equal, the process specified by *`pid`* becomes a process group leader.
- If *`pid`* is 0, the process ID of the caller is used.
- If *`pgid`* is 0, the process ID specified by *`pid`* is used as the process group ID.

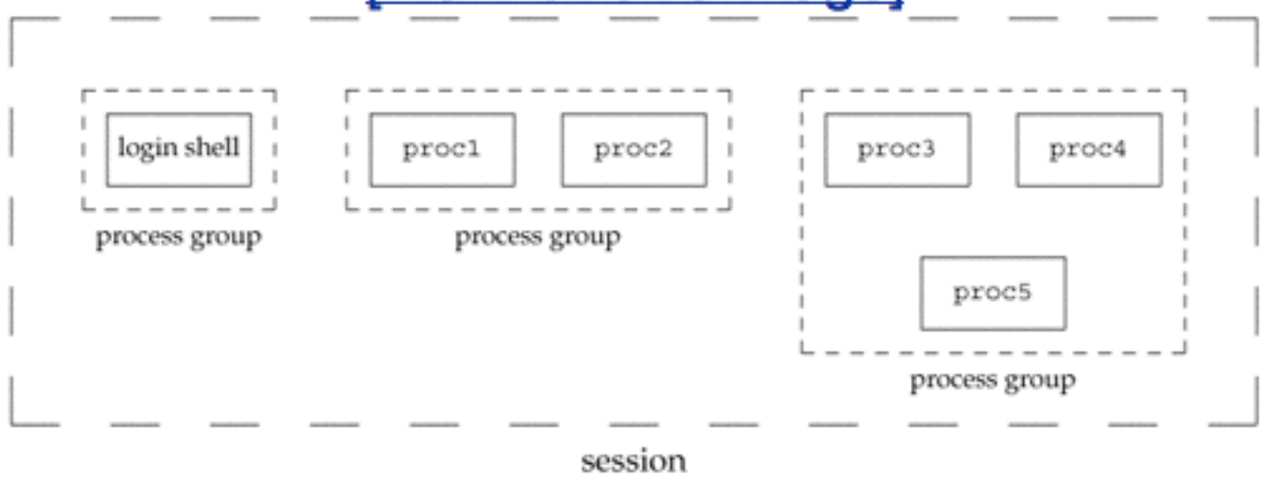
Sessions

- A session is a collection of one or more process groups.

Figure 9.6. Arrangement of processes into process groups and sessions

```
proc1 | proc2 &
proc3 | proc4 | proc5
```

[View full size image]



Sessions

- A process establishes a new session by calling the `setsid(2)` function.

```
#include <unistd.h>

pid_t setsid(void);


   Returns: process group ID if OK, 1 on error
```

- If the calling process is not a process group leader, this function creates a new session. Three things happen:
  - The process becomes the session leader of this new session.
  - The process becomes the process group leader of a new process group.
  - The process has no controlling terminal.

Controlling Terminal

- Sessions and process groups have a few other characteristics.
  - A session can have a single controlling terminal.
  - The session leader that establishes the connection to the controlling terminal is called the controlling process.
  - The process groups within a session can be divided into a single foreground process group and one or more background process groups.

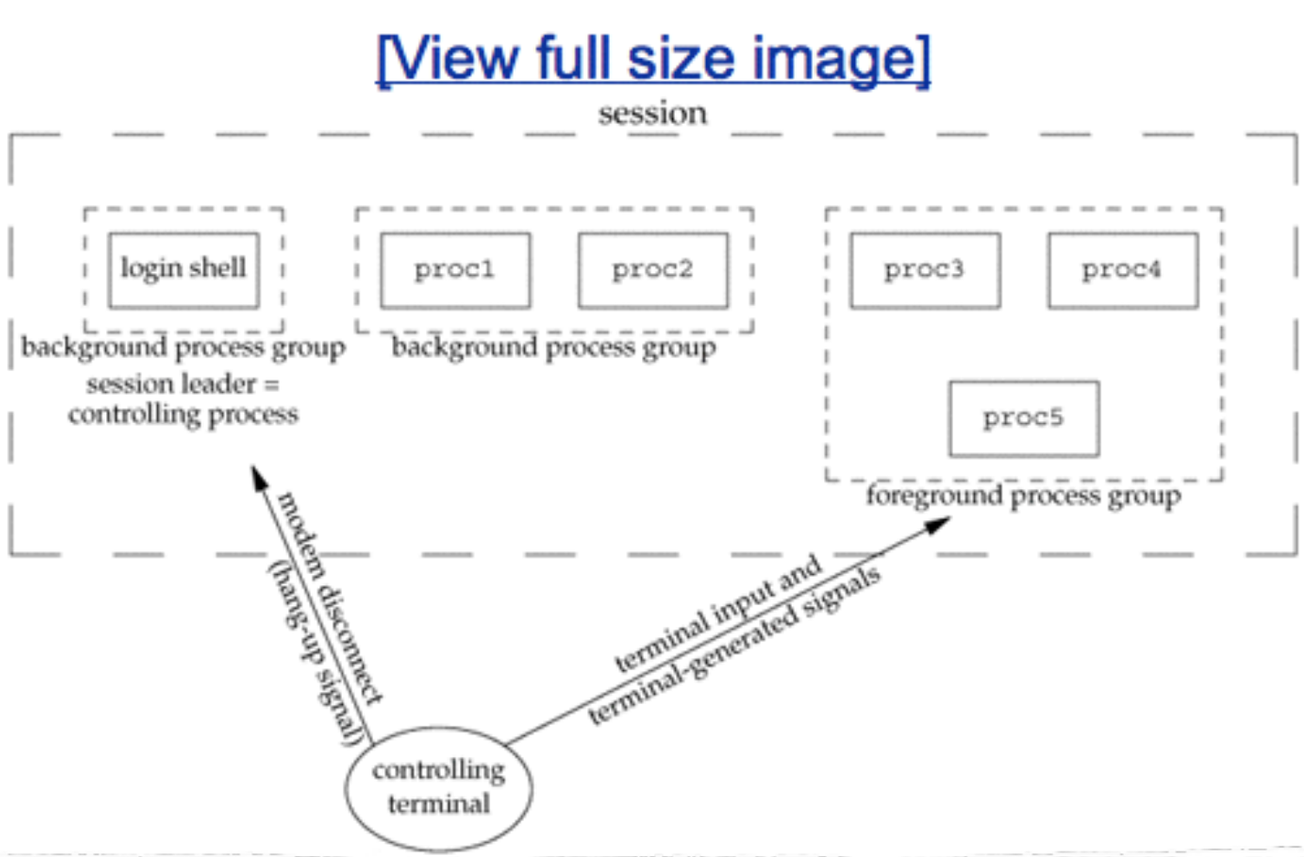
Controlling Terminal

- Sessions and process groups have a few other characteristics.
  - Whenever we type the terminal's interrupt key (often DELETE or Control-C), this causes the interrupt signal be sent to all processes in the foreground process group.
  - Whenever we type the terminal's quit key (often Control-backslash), this causes the quit signal to be sent to all processes in the foreground process group.
  - If a modem (or network) disconnect is detected by the terminal interface, the hang-up signal is sent to the controlling process (the session leader).

Controlling Terminal

Figure 9.7. Process groups and sessions showing controlling terminal

[View full size image]



Job Control

- Job control is a feature added to BSD around 1980.
  - Allow us to start multiple jobs (groups of processes) from a single terminal
  - Allow us to control which jobs can access the terminal and which jobs are to run in the background

Job Control

- Job control requires three forms of support:
  - A shell that supports job control
  - The terminal driver in the kernel must support job control
  - The kernel must support certain job-control signals

Job Control

- A job is simply a collection of processes, often a pipeline of processes.
  - The command "`vi main.c`" starts a job consisting of one process in the foreground.
  - The commands

```
pr *.c | lpr &
make all &
```

start two jobs in the background.

Job Control

- Some examples:

```
$ make all > Make.out &
[1]     1475
$ pr *.c | lpr &
[2]     1490
$                                          just press RETURN
[2] +  Done                    pr *.c | lpr &
[1] +  Done                    make all > Make.out &
```

```
$ cat > temp.foo &              start in background, but it'll read from standard input
[1]     1681
$                               we press RETURN
[1] + Stopped (SIGTTIN)         cat > temp.foo &
$ fg %1                          bring job number 1 into the foreground
cat > temp.foo                  the shell tells us which job is now in the foreground

hello, world                     enter one line

^D                              type the end-of-file character
$ cat temp.foo                   check that the one line was put into the file
hello, world
```

Job Control

- Some examples:

```
$ cat temp.foo &                    execute in background
[1]     1719
$ hello, world                      the output from the background job appears after the prompt
                                    we press RETURN
[1] + Done                      cat temp.foo &
$ stty tostop                         disable ability of background jobs to output to
controlling terminal
$ cat temp.foo &                     try it again in the background
[1]     1721
$                                   we press RETURN and find the job is stopped
[1] + Stopped(SIGTTOU)                      cat temp.foo &
$ fg %1                             resume stopped job in the foreground
cat temp.foo                        the shell tells us which job is now in the foreground
hello, world                        and here is its output
```

Job Control

- What happens in this case if a background process tries to read from its controlling terminal? For example,

```
cat > temp.foo &
```

- With job control

```
$ cat > temp.foo &                  start in background, but it'll read from standard input
[1]     1681
$                                   we press RETURN
[1] + Stopped (SIGTTIN)             cat > temp.foo &
$ fg %1                             bring job number 1 into the foreground
cat > temp.foo                      the shell tells us which job is now in the foreground

hello, world                        enter one line

^D                                  type the end-of-file character
$ cat temp.foo                      check that the one line was put into the file
hello, world
```

Job Control

- What happens in this case if a background process tries to read from its controlling terminal? For example,

```
cat > temp.foo &
```

  - Without job control: The shell automatically redirects the standard input of a background process to `/dev/null`, if the process doesn't redirect standard input itself.
    - A read from `/dev/null` generates an end of file.
    - The background `cat` process immediately reads an end of file and terminates normally.

Shell Execution of Programs

- A shell that doesn't support job control

If we execute

```
ps -o pid,ppid,pgid,sid,comm
```

the output is

```
  PID  PPID  PGID   SID  COMMAND
  949   947   949   949  sh
 1774   949   949   949  ps
```

Shell Execution of Programs

- A shell that doesn't support job control

If we execute the command in the background,

```
ps -o pid,ppid,pgid,sid,comm &
```

the only value that changes is the process ID of the command:

```
  PID  PPID  PGID   SID COMMAND
  949   947   949   949 sh
 1812   949   949   949 ps
```

This shell doesn't know about job control!!

Shell Execution of Programs

- A shell that doesn't support job control

Let's now look at how the Bourne shell handles a pipeline. When we execute

```
ps -o pid,ppid,pgid,sid,comm | cat1
```

the output is

```
  PID  PPID  PGID   SID COMMAND
  949   947   949   949 sh
 1823   949   949   949 cat1
 1824  1823   949   949 ps
```

Shell Execution of Programs

- If we execute three processes in the pipeline, we can examine the process control used by this shell:

```
ps -o pid,ppid,pgid,sid,comm | cat1 | cat2
```

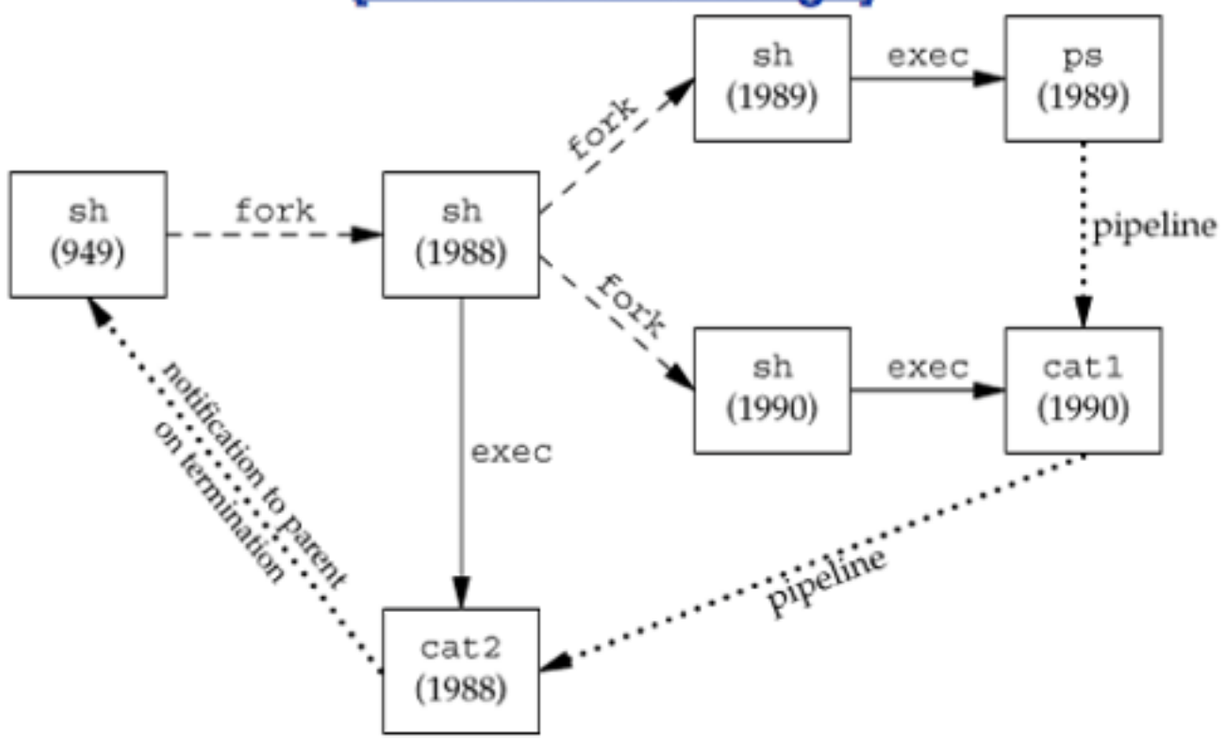
generates the following output

```
  PID  PPID  PGID   SID COMMAND
  949   947   949   949 sh
 1988   949   949   949 cat2
 1989  1988   949   949 ps
 1990  1988   949   949 cat1
```

Shell Execution of Programs

Figure 9.9. Processes in the pipeline `ps | cat1 | cat2` when invoked by Bourne shell

[View full size image]



Shell Execution of Programs

- A shell that supports job control

```
ps -o pid,ppid,pgrp,session,tpgid,comm
```

gives us

```
  PID  PPID  PGRP  SESS  TPGID COMMAND
 2837  2818  2837  2837   5796 bash
 5796  2837  5796  2837   5796 ps
```

Executing this process in the background,

```
ps -o pid,ppid,pgrp,session,tpgid,comm &
```

gives us

```
  PID  PPID  PGRP  SESS  TPGID COMMAND
 2837  2818  2837  2837   2837 bash
 5797  2837  5797  2837   2837 ps
```

Shell Execution of Programs

- The `TPGID` of 2837 indicates that the foreground process group is our login shell.

Executing two processes in a pipeline, as in

```
ps -o pid,ppid,pgrp,session,tpgid,comm | cat1
```

gives us

```
  PID  PPID  PGRP  SESS  TPGID COMMAND
 2837  2818  2837  2837   5799 bash
 5799  2837  5799  2837   5799 ps
 5800  2837  5799  2837   5799 cat1
```

Shell Execution of Programs

- If we execute this pipeline in the background,

```
ps -o pid,ppid,pgrp,session,tpgid,comm | cat1 &
```

the results are similar, but now `ps` and `cat1` are placed in the same background process group:

```
  PID  PPID  PGRP  SESS  TPGID COMMAND
 2837  2818  2837  2837   2837 bash
 5801  2837  5801  2837   2837 ps
 5802  2837  5801  2837   2837 cat1
```

Assignment

- Reading (do it at home):
  - Manual pages for the functions covered
  - Stevens Chap. 9
- Assignment 8
  - Try the commands in Chapter 9.9 to see if the shell you use supports the job control
    - ➡ A report with detailed steps you did should be handed in.

Signal

Signal Concepts

- Signals are a way for a process to be notified of asynchronous events.
- Every signal has a name all beginning with `SIG`.
- All defined by positive integer constants (the signal number) in the header `<signal.h>`.

signum.h = (/usr/include/i386-linux-gnu/bits) - VIM

signum.h

```
34 #define SIGINT      2   /* Interrupt (ANSI).  */
35 #define SIGQUIT     3   /* Quit (POSIX).  */
36 #define SIGILL      4   /* Illegal instruction (ANSI).  */
37 #define SIGTRAP     5   /* Trace trap (POSIX).  */
38 #define SIGABRT     6   /* Abort (ANSI).  */
39 #define SIGIOT      6   /* IOT trap (4.2 BSD).  */
40 #define SIGBUS      7   /* BUS error (4.2 BSD).  */
41 #define SIGFPE      8   /* Floating-point exception (ANSI).  */
```

Signal Concepts

- Some examples for asynchronous events:
  - a timer you set has gone off (`SIGALRM`)
  - some I/O you requested has occurred (`SIGIO`)

Signal Concepts

- Besides the asynchronous events listed previously, there are many ways to generate a signal:
  - terminal generated signals (user presses a key combination which causes the terminal driver to generate a signal)
  - hardware exceptions (divide by 0, invalid memory references, etc)
  - `kill(1)` allows a user to send any signal to any process (if the user is the owner or superuser)
  - `kill(2)` (a system call) performs the same task
  - software conditions (other side of a pipe no longer exists, urgent data has arrived on a network file descriptor, etc.)

Signal Concepts

- Once we get a signal, we can do one of several things:

1) Ignore it. (note: there are some signals which we CANNOT or SHOULD NOT ignore)
   - Two signals can never be ignored: `SIGKILL` and SIGSTOP.
2) Catch it. That is, have the kernel call a function which we define whenever the signal occurs.
3) Accept the default. Have the kernel do whatever is defined as the default action for this signal.

Figure 10.1. UNIX System signals

Name	Description	ISO C	SUS	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	Default action
SIGABRT	abnormal termination (abort)	•	•	•	•	•	•	terminate+core
SIGALRM	timer expired (alarm)		•	•	•	•	•	terminate
SIGBUS	hardware fault		•	•	•	•	•	terminate+core
SIGCANCEL	threads library internal use						•	ignore
SIGCHLD	change in status of child		•	•	•	•	•	ignore
SIGCONT	continue stopped process		•	•	•	•	•	continue/ignore
SIGEMT	hardware fault			•	•	•	•	terminate+core
SIGFPE	arithmetic exception	•	•	•	•	•	•	terminate+core
SIGFREEZE	checkpoint freeze						•	ignore

- **terminate+core**: A memory image of the process is left in the file named core of the current working directory of the process.
  - This file can be used with most UNIX System debuggers to examine the state of the process at the time it terminated.

Signal Concepts

- `SIGABRT`
  - Generated by calling the `abort` function (Section 10.17).
  - The process terminates abnormally.
- `SIGALRM`
  - Generated when a timer set with the `alarm` function expires (Section 10.10).
- `SIGSEGV`
  - This signal indicates that the process has made an invalid memory reference.

Signal Concepts

- `SIGINT`
  - Generated by the terminal driver when we type the interrupt key (often DELETE or Control-C).
  - This signal is sent to all processes in the foreground process group.
  - This signal is often used to terminate a runaway program, especially when it's generating a lot of unwanted output on the screen.

Signal Concepts

- `SIGCHLD`
  - Whenever a process terminates or stops, the `SIGCHLD` signal is sent to the parent.
  - By default, this signal is ignored, so the parent must catch this signal if it wants to be notified whenever a child's status changes.
  - The normal action in the signal-catching function is to call one of the wait functions to fetch the child's process ID and termination status.

Signal Concepts

- `SIGKILL`
  - Can't be caught or ignored
  - Provide the system administrator with a sure way to kill any process
- `SIGSTOP`
  - Can't be caught or ignored
  - Stop a process

Signal(3)

- The simplest interface to the signal features of the UNIX System

```
#include <signal.h>

void (*signal(int signo, void (*func)(int)))(int);

Returns: previous disposition of signal (see following) if
                                        OK, SIG_ERR on error
```

- The *signo* argument is just the name of the signal from Figure 10.1.
- *func* can be:
  - SIG_IGN which requests that we ignore the signal *signo*
  - SIG_DFL which requests that we accept the default action for signal *signo*
  - the address of a function which should catch or handle a signal

Signal(3)

- The prototype for the `signal` function states that the function
  - requires two arguments and returns a pointer to a function that returns nothing (void).
    - The first argument, *`signo`*, is an integer.
    - The second argument is a pointer to a function that takes a single integer argument and returns nothing.
    - The function whose address is returned as the value of signal takes a single integer argument (the final (int)).

Signal(3)

- The perplexing `signal` function prototype can be made much simpler through the use of the following `typedef`

```
typedef void Sigfunc(int);
```

```
Sigfunc *signal(int, Sigfunc *);
```

- Includ this typedef in `apue.h`

Signal(3)

- If we examine the system's header <signal.h>, we probably find declarations of the form

```
#define SIG_ERR (void (*)())-1
#define SIG_DFL (void (*)())0
#define SIG_IGN (void (*)())1
```

signum.h = (/usr/include/i386-linux-gnu/bits) - VIM

signum.h

```
19
20 #ifdef  _SIGNAL_H
21
22 /* Fake signal functions.  */
23 #define SIG_ERR ((__sighandler_t) -1)       /* Error return.  */
24 #define SIG_DFL ((__sighandler_t) 0)        /* Default action.  */
25 #define SIG_IGN ((__sighandler_t) 1)        /* Ignore signal.  */
```

Figure 10.2. Simple program to catch SIGUSR1 and SIGUSR2

```
#include "apue.h"

static void sig_usr(int);   /* one handler for both signals */

int
main(void)
{
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        err_sys("can't catch SIGUSR1");
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        err_sys("can't catch SIGUSR2");
    for ( ; ; )
        pause();
}

static void
sig_usr(int signo)      /* argument is signal number */
{
    if (signo == SIGUSR1)
        printf("received SIGUSR1\n");
    else if (signo == SIGUSR2)
        printf("received SIGUSR2\n");
    else
        err_dump("received signal %d\n", signo);
}
```

```
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig10-2 &
[1] 30797
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] kill -USR1 30797
received SIGUSR1
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] kill -USR2 30797
received SIGUSR2
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] kill 30797
[1]+  Terminated              ./fig10-2
```

`Signal(3)`

- When a program is `exec`ed, the status of all signals is either default or ignore.
- When a process `fork(2)`s, the child inherits the parent's signal dispositions.
- A limitation of the `signal(3)` function is that we can only determine the current disposition of a signal by changing the disposition.

Unreliable Signals

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

Unreliable Signals

- One problem is that the action for a signal was reset to its default each time the signal occurred.

```
int     sig_int();          /* my signal handling function */

...
signal(SIGINT, sig_int);  /* establish handler */
...

sig_int()
{
    signal(SIGINT, sig_int);  /* reestablish handler for next time */
    ...                       /* process the signal ... */
}
```

- There is a window of time after the signal has occurred, but before the call to signal in the signal handler when the interrupt signal could occur another time.
- This second signal would cause the default action to occur (terminates the process).

- Another problem is that the process was unable to turn a signal off when it didn't want the signal to occur.

```
int     sig_int_flag;           /* set nonzero when signal occurs */

main()
{
    int     sig_int();          /* my signal handling function */
    ...
    signal(SIGINT, sig_int);    /* establish handler */
    ...
    while (sig_int_flag == 0)
        pause();                /* go to sleep, waiting for signal */
    ...
}

sig_int()
{
    signal(SIGINT, sig_int);    /* reestablish handler for next time */
    sig_int_flag = 1;           /* set flag for main loop to examine */
}
```

- There is a window of time when things can go wrong.
- If the signal occurs after the test of `sig_int_flag`, but before the call to pause, the process could go to sleep forever (assuming that the signal is never generated again).

Interrupted System Calls

- Some system calls can block for long periods of time (or forever). These include things like:
  - `read(2)`s from files that can block (pipes, networks, terminals)
  - `write(2)` to the same sort of files
  - `open(2)` of a device that waits until a condition occurs (for example, a modem)
  - `pause(3),` which purposefully puts a process to sleep until a signal occurs
- The exception to these slow system calls is anything related to disk I/O.
  - Although a read or a write of a disk file can block the caller temporarily, unless a hardware error occurs, the I/O operation always returns and unblocks the caller quickly.

Interrupted System Calls

- Catching a signal during execution of one of these calls traditionally led to the process being aborted with an `errno` return of `EINTR`.

```
again:
    if ((n = read(fd, buf, BUFFSIZE)) < 0) {
        if (errno == EINTR)
            goto again;     /* just an interrupted system call */
        /* handle other errors */
}
```

  - 4.2BSD: automatic restarting of certain interrupted system calls.
  - The system calls that were automatically restarted are `ioctl`, `read`, `readv`, `write`, `writev`, `wait`, and `waitpid`.
    - The first five of these functions are interrupted by a signal only if they are operating on a slow device.
    - `wait` and `waitpid` are always interrupted when a signal is caught.

Reentrant Functions

- If your process is currently handling a signal, what functions are you allowed to use?
- Consider that whenever a signal occurs, control is immediately transferred from whatever current instruction is being executed to the start of your signal handler.

Reentrant Functions

- Suppose your program was in the middle of a `malloc` system call when a signal occurs.
- Can you call `malloc` in your signal handler?
  - Most certainly not.
- Functions which can have several instances in progress simultaneously are said to be reentrant functions.
  - POSIX.1 guarantees that certain functions are reentrant.

Reentrant Functions

Figure 10.4. Reentrant functions that may be called from a signal handler

accept	fchmod	lseek	sendto	stat
access	fchown	lstat	setgid	symlink
aio_error	fcntl	mkdir	setpgid	sysconf

- Most functions are not in Figure 10.4 because
  a) they are known to use static data structures
  b) they call `malloc` or `free`
  c) they are part of the standard I/O library.
- Note that even though we call printf from signal handlers in some of our examples, it is not guaranteed to produce the expected results, since the signal hander can interrupt a call to printf from our main program.

Reliable-Signal Terminology and Semantics

- Define some of the terms used throughout our discussion of signals
  - A signal is *generated* for a process (or sent to a process) when the event that causes the signal occurs.
  - A signal is *delivered* to a process when the action for a signal is taken.
    - During the time between the generation of a signal and its delivery, the signal is said to be *pending*.

Reliable-Signal Terminology and Semantics

- Define some of the terms used throughout our discussion of signals
  - A process has the option of *blocking* the delivery of a signal.
  - If the action for that signal is either the default action or to catch the signal, then the signal remains pending for the process until the process either (a) unblocks the signal or (b) changes the action to ignore the signal.

Reliable-Signal Terminology and Semantics

- Define some of the terms used throughout our discussion of signals
  - A blocked signal is generated more than once before the process unblocks the signal.
    - POSIX.1 allows the system to deliver the signal either once or more than once.
    - The signals are *queued*.

Reliable-Signal Terminology and Semantics

- Define some of the terms used throughout our discussion of signals
  - What happens if more than one signal is ready to be delivered to a process?
    - POSIX.1 does not specify the order.
    - The Rationale for POSIX.1 does suggest, however, that signals related to the current state of the process be delivered before other signals. (`SIGSEGV`)

Reliable-Signal Terminology and Semantics

- Define some of the terms used throughout our discussion of signals
  - Each process has a signal mask that defines the set of signals currently blocked.
    - A process can examine and change its current signal mask by calling `sigprocmask` (Section 10.12).

Reliable-Signal Terminology and Semantics

- Define some of the terms used throughout our discussion of signals
  - POSIX.1 defines a data type, `sigset_t`, that holds a signal set.
    - E.g., the signal mask is stored in one of these signal sets (Section 10.11).

`kill(2)` and `raise(3)`

- `kill(2)` sends a signal to a process or a group of processes.
- `raise(3)` allows a process to send a signal to itself.

```
#include <signal.h>

int kill(pid_t pid, int signo);

int raise(int signo);
```

Both return: 0 if OK, 1 on error

kill(2) and raise(3)

```
#include <signal.h>

int kill(pid_t pid, int signo);

int raise(int signo);
```

Both return: 0 if OK, 1 on error

- *pid>0* – signal is sent to the process whose PID is *pid*
- *pid==0* – signal is sent to all processes whose process group ID equals the process group ID of the sender (also needs permissions)
- *pid<0* – signal is sent to all processes whose process group ID equals the absolute value of *pid* (also needs permissions)
- *pid==1* – signal is sent to all processes on the system for which the sender has permission to send the signal

`kill(2)` and `raise(3)`

- 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.

`kill(2)` and `raise(3)`

- 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.
  - 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`.

`alarm(2)` and `pause(2)`

- `alarm(2)` allows us to set a timer
  - 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

`alarm(2)` and `pause(2)`

- 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.

`alarm(2)` and `pause(2)`

- Most processes that use an alarm clock catch `SIGALRM`.
  - 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 (default action)!

alarm(2) and pause(2)

- pause(2) 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.

Figure 10.7. Simple, incomplete implementation of `sleep`

```
#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 */
}
```

- 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).

Figure 10.8. Another (imperfect) implementation of `sleep`

```
#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)
{
    if (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 10.9. Calling `sleep2` from a program that catches other signals

```
#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");
}
```

- A common use for alarm is to put an upper time limit on operations that can block.

Figure 10.10. Calling `read` with a timeout

```
#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 */
}
```

- Race condition still exits!
  - Between the first call to `alarm` and the call to `read`.

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

```
#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);
}
```

Signal Sets

- Signal Set: a data type to represent multiple signals
- `sigprocmask`: Tell the kernel not to allow any of the signals in the set to occur
- POSIX.1 defines the data type `sigset_t` to contain a signal set.
  - Why do we use an integer to store it?

Signal Sets

```
#include <signal.h>

int sigemptyset(sigset_t *set);

int sigfillset(sigset_t *set);

int sigaddset(sigset_t *set, int signo);

int sigdelset(sigset_t *set, int signo);
```

All four return: 0 if OK, 1 on error

```
int sigismember(const sigset_t *set, int signo);
```

Returns: 1 if true, 0 if false, 1 on error

Signal Sets

- If the implementation has fewer signals than bits in an integer, a signal set can be implemented using one bit per signal.

```
#define sigemptyset(ptr)    (*(ptr) = 0)
#define sigfillset(ptr)     (*(ptr) = ~(sigset_t)0, 0)
```

Figure 10.12. An implementation of `sigaddset`, `sigdelset`, and `sigismember`

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

/* <signal.h> usually defines NSIG to include signal number 0 */
#define SIGBAD(signo)   ((signo) <= 0 || (signo) >= NSIG)

int
sigaddset(sigset_t *set, int signo)
{
    if (SIGBAD(signo)) { errno = EINVAL; return(-1); }

    *set |= 1 << (signo - 1);       /* turn bit on */
    return(0);
}

int
sigdelset(sigset_t *set, int signo)
{
    if (SIGBAD(signo)) { errno = EINVAL; return(-1); }

    *set &= ~(1 << (signo - 1));    /* turn bit off */
    return(0);
}

int
sigismember(const sigset_t *set, int signo)
{
     if (SIGBAD(signo)) { errno = EINVAL; return(-1); }

     return((*set & (1 << (signo - 1))) != 0);
}
```

`sigprocmask(2)` Function

- Examine and change blocked signals

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *restrict set,
                sigset_t *restrict oset);
```

Returns: 0 if OK, 1 on error

- If *oset* is a non-null pointer, the current signal mask for the process is returned through *oset*.
- If *set* is a non-null pointer, the *how* argument indicates how the current signal mask is modified.
- If *set* is a null pointer, the signal mask of the process is not changed, and *how* is ignored.

Figure 10.13. Ways to change current signal mask using `sigprocmask`

how	Description
SIG_BLOCK	The new signal mask for the process is the union of its current signal mask and the signal set pointed to by *set*. That is, *set* contains the additional signals that we want to block.
SIG_UNBLOCK	The new signal mask for the process is the intersection of its current signal mask and the complement of the signal set pointed to by *set*. That is, *set* contains the signals that we want to unblock.
SIG_SETMASK	The new signal mask for the process is replaced by the value of the signal set pointed to by *set*.

Figure 10.14. Print the signal mask for the process

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

void
pr_mask(const char *str)
{
    sigset_t    sigset;
    int         errno_save;

    errno_save = errno;     /* we can be called by signal handlers */
    if (sigprocmask(0, NULL, &sigset) < 0)
        err_sys("sigprocmask error");

    printf("%s", str);
    if (sigismember(&sigset, SIGINT))   printf("SIGINT ");
    if (sigismember(&sigset, SIGQUIT))  printf("SIGQUIT ");
    if (sigismember(&sigset, SIGUSR1))  printf("SIGUSR1 ");
    if (sigismember(&sigset, SIGALRM))  printf("SIGALRM ");

    /* remaining signals can go here */

    printf("\n");
    errno = errno_save;
}
```

`abort(3)` Function

- The `abort(3)` function causes abnormal program termination.

```
#include <stdlib.h>

void abort(void);


                    This function never returns
```

- This function sends the `SIGABRT` signal to the caller.

Assignment

- Reading (do it at home):
  - Manual pages for the functions covered
  - Stevens Chap. 10