Lecture #6 – Process Control (Chapter 8 continued)

- Race conditions

A **race condition** occurs when multiple processes are trying to do something with a shared resource (data) and the final outcome depends on the order in which the processes run.

These conditions are very difficult to identify and debug, and serve as a roadblock for concurrent programming.

These conditions are resolved generally with signaling and synchronization mechanisms to be discussed in later chapters.

Example (program with a race condition):

```
static void pr_char(char *str)
{
        char    *ptr;
        int     c;

        setbuf(stdout, NULL);                   /* unbuffered */

        for (ptr = str; c = *ptr++; )
                putc(c, stdout);
}

int main(void)
{
        pid_t   pid;

        if ((pid = fork()) < 0)
        {
                perror("fork failed");
                exit(1);
        }
        else if (pid == 0)
                pr_char("output from child\n");
        else
                pr_char("output from parent\n");

        exit(0);
}
```

Output:

```
$ race_prog
output from child
output from parent

$ race_prog
ooutput from parent
utput from child
```

- Exec functions

  Exec is 6 function calls which allow a process to begin running a new program.

  When a process calls one of the exec functions, that process is completely replaced by the new program, and that program begins executing at its main functions.

  Note: The PID does not change (no new process created).

```
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[]);
```

  Note: all six return -1 on error, and no return for success

Function	Pathname	Filename	Arg list	Argv[]	Environ	Envp[]
Execl	*		*		*	
Execlp		*	*		*	
Execle	*		*			*
Execv	*			*	*	
Execvp		*		*	*	
Execve	*			*		*
(letter)		p	L	v		e

  Note: The versions that end in "p" will search the path if filename doesn't start with "/".
  Note: The versions that end in "e" allow you to specify an environment pointer.
  Note: The versions that have "l" after exec, list command args individually.
  Note: The versions that have "v" after exec, have command args in an array.

Properties inherited after exec:

PID and PPID, Process group ID, and session ID
RUID, RGID, and supplementary group ID's
Controlling terminal
Time left until alarm
Current working directory
Root directory
File mode creation mask
File locks
Process signal mask and pending signals
Resource limits
tms_utime, tms_stime, tms_cutime, tms_ustime values

Example (echoall.c):

```
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);

        return(0);
}
```

Example (exec_test.c):

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

int main(void)
{
        pid_t   pid;

        if ((pid = fork()) < 0)
        {
                perror("fork failed");
                exit(1);
        }
        else if (pid == 0)
        {
```

```
                if (execle("/home/richj/echoall", "echoall", "myarg1", "ARG2", (char *) 0,
                        env_init) < 0)
                {
                        perror("execle failed");
                        exit(1);
                }
        }

        if (waitpid(pid, NULL, 0) < 0)
        {
                perror("waitpid failed");
                exit(1);
        }

        if ((pid = fork()) < 0)
        {
                perror("fork failed");
                exit(1);
        }
        else if (pid == 0)
        {
                if (execlp("echoall", "echoall", "only 1 arg", (char *) 0) < 0)
                {
                        perror("execlp failed");
                        exit(1);
                }
        }

        exit(0);
}
```

- Changing UID and GID

  int setuid(uid_t uid);
  int setgid(gid_t gid);

  Rules for setuid:

  1. If EUID = 0, setuid sets RUID, EUID, and saved UID to "uid".
  2. If EUID != 0 but "uid" equals either RUID or saved UID, setuid sets EUID to "uid".
  3. Otherwise, errno is set to EPERM, and error is returned.

  Only a superuser process can change the RUID (normally set by login(1)).
  EUID is set by "exec" only if SUID is set for the program file.

Versions that change only effective ID's:

```
int     seteuid(uid_t uid);
int     setegid(gid_t gid);
```

- System function

The system fct allows an easy way to execute a command string from within a program.

Example: system("date > file");

```
int     system(const char *cmdstring);
```

Return value:

-1	if fork fails or waitpid returns error other than EINTR
127	if exec fails (i.e. shell can't be executed)
exit status	if all functions succeed, the value is the termination status of shell

Example (system function simplified):

```
int system(const char *cmdstring)
{
        pid_t   pid;
        int     status;

        if (cmdstring == NULL)
                return(1);

        if ((pid = fork()) < 0)
                status = -1;
        else if (pid == 0)
        {
                execl("/bin/sh", "sh", "-c", cmdstring, (char *) 0);
                _exit(127);
        }
        else
        {
                while (waitpid(pid, &status, 0) < 0)
                        if (errno != EINTR)
                        {
                                status = -1;
                                break;
                        }
        }
```

```
        return(status);
}
```

- Process Accounting

  Most UNIX systems provide an option to do process accounting.
  When enabled, the kernel writes an accounting record each time a process terminates.

  These records are usually placed in /var/adm/pacct, and store information like UID's for the process and accumulated resource usage.

- User Identification

  ```
  char    *getlogin(void);
  ```

  Returns: pointer to string giving login name, NULL on error

- Process Times

  ```
  clock_t times(struct tms *buf);
  ```

  Returns: elapsed wall clock time in clock ticks, -1 on error

  ```
  struct tms {
          clock_t         tms_utime;          /* user CPU */
          clock_t         tms_stime;          /* system CPU */
          clock_t         tms_cutime;         /* user CPU, terminated children */
          clock_t         tms_cstime;         /* system CPU, terminated children */
  };
  ```

  Note: Number of clock ticks per second can be returned by sysconf(1).

  Note: See code on page 258 for times example (times_test.c)

Process Relationships (Chapter 9)

- Terminal Logins

In early UNIX systems, users logged into dumb terminals connected via RS-232.

4.3+BSD Terminal Logins:

The sysadmin would create a text file called /etc/ttys (one line per terminal).
Each line would specify the device name, and other parms passed to getty.

On system startup, init reads /etc/ttys, and fork/exec a getty process for each.

Getty calls open for the terminal device (read and write).
Getty then opens descriptors 0, 1, and 2 to the device, and outputs "login:".
Getty waits for someone to enter a user name.

After username is entered, getty invokes the login process:
Example: execle("/usr/bin/login", "login", "-p", username, (char *) 0, envp);

Login then calls getpwnam to fetch our password file entry.
It then calls getpass(3) to display "Password:" and read passwd without echo.
It then calls crypt(3) to encrypt the password and compare to passwd file entry.

If login attempt fails due to incorrect password, login exits with "1".
Init then respawns getty to start over.

If we log in correctly, login performs the following:

Change to our home directory (chdir)
Change ownership of terminal device (chown)
Group ID's are set (setgid, and initgroups)
Environment is initialized (HOME, SHELL, USER, LOGNAME, PATH)
User ID is set (setuid)
Login shell is invoked [execl("/bin/sh", "-sh", (char *) 0);

SVR4 Terminal Logins:

SVR4 supports 2 forms of terminal login a) getty b) ttymon.
ttymon is part of a larger facility called SAF (Service Access Facility).
Init is parent of sac (service access controller) which fork/exec ttymon.
ttymon forks when we've entered our username.
This child of ttymon does an exec of login to get password.

Note: our login shell is now parented by ttymon not init.

- Network Logins

  With network logins, instead of having a process for each terminal device, we have a process that waits for network connections.

  Starting with 4.3+ BSD, there is one process (inetd) that listens for most network connections. Inetd is sometimes called the Internet superserver.

  Inetd waits for TCP/IP connection requests to arrive at the host, and then to fork a process and exec the appropriate program.

  For example, if you type "telnet myhost" at the prompt, you are contacting the inetd daemon on "myhost", which accepts the connection, forks, and execs the telnet program.

  Once telnetd has started, it opens a pseudo-terminal device and splits in two by using fork. The parent process handles the communication across the network, and the child process does an exec of "login".

- Process Groups

  A process group is a collection of one or more processes.
  Each process group has a unique process group ID.
  Process group ID's are similar to PID's - they are positive integers (pid_t datatype).

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

  Each process group can have a process group leader (i.e. process group ID = PID)

  A process can set the PGID of itself or one of its children.
  If setting on a child process, that process can not have called exec.
  If pid is 0, then process ID of the caller is used.
  If pgid is 0, the process ID specified in pid is used for pgid.

  Note: Process groups become especially important relative to signals.

- Sessions

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

  See example on page 270 created by:

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

  ```
  pid_t setsid(void);
  ```

If the calling process is not a process group leader:

1. The process becomes the session leader of the new session.
2. The process becomes the process group leader of a new process group.
3. The process has no controlling terminal.

Note: Error is returned is the caller is already a process group leader.

Note: To ensure we are not a process group leader, fork and have parent exit, but child continue (i.e. child has PGID of parent but different PID).

- Controlling Terminal

  A session can have a single controlling terminal (usually the terminal we logged in on).

  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.

  If a session has a controlling terminal, then it has a single foreground process, and all other process groups in the session are background process groups.

  Whenever we type interrupt (ctrl-C), this causes the interrupt signal to be sent to all processes in the foreground process group.

  If a modem disconnect is detected by the terminal interface, the hang-up signal is sent to the controlling process (session leader).

  Note: If we wish to talk to the controlling terminal regardless of where stdin and stdout are pointed, then we can access **/dev/tty**.

- Tcgetpgrp and tcsetpgrp

  ```
  pid_t   tcgetpgrp(int filedes);
  int     tcsetpgrp(int filedes, pid_t pgrpid);
  ```

  tcgetpgrp returns the PGID of the foreground process group associated with the terminal open on filedes.

  If the process has a controlling terminal, the process can call tcsetpgrp to set the foreground process group to pgrpid.

- Job Control

  Job control requires the following:

  A shell that supports job control (i.e. C shell, Korn, Bash, Tcsh, etc.).
  The terminal driver must support job control.
  Support for certain job control signals must be provided.

  Special characters:

  Interrupt character (often ctrl-c) generates a SIGINT
  Quit character (often ctrl-\) generates a SIGQUIT
  Suspend character (often ctrl-z) generates a SIGTSTP

  If background job attempts to read from terminal, SIGTTIN is sent to process. This stops the process, and we are notified at the shell so we can bring the process to foreground to give the program input.

  Bringing a job to the foreground (using fg command), sends the process a SIGCONT.

  Normally output from background processes is allowed to the terminal, but if you issue "stty tostop", then the process gets a SIGTTOU and is stopped until we bring it to foreground.

- Shell execution of programs

  Single command (i.e. cat file):

  If we type in a normal command in a shell, then the shell:

  Calls fork to make a copy of itself
  Calls exec to run the program (i.e. cat)
  The parent shell waits for the child process to complete.

  If we had chosen to put this process in background (i.e. cat file &), then the shell would not have waited to the child process to complete before continuing.

  Pipeline (i.e. ps | cat1):

  The shell forks a copy of itself.
  The copy forks again, and the parent execs "cat1" while the child execs "ps".
  The parent shell waits for both processes to complete.

Another pipeline (i.e. ps | cat1 | cat2):

Assume the original shell is PID 949.
The shell forks a copy of itself (PID 1988).
PID 1988 forks twice (PID's 1989, 1990), and then execs "cat2"
PID 1989 execs "ps"
PID 1990 execs "cat1"

Note: See picture page 281.

- Orphaned Process Groups

When the parent terminates, the child process is orphaned, so the child's parent process ID becomes 1 (init).

At this point, the child is now a member of an orphaned process group. Every process in the newly orphaned process group is sent a SIGHUP followed by a SIGCONT.