Chapter 8: Process Control

Figure 8.27  Process structure for accounting example
Process Identifiers

- Guaranteed to be unique for each currently executing process on a single computer
- Usually sequentially allocated
- Some system services have PIDs as well
  - 0: scheduler/swapper
  - 1: init
  - 2: pagedaemon
getpid(), getppid(), getuid(), geteuid()

- getpid() – get process ID
  - pid_t getpid(void);

- getppid() – get parent process ID
  - pid_t getppid(void);

- getuid() – get the real user ID of the calling process
  - uid_t getuid(void);

- uid_t geteuid(void) – get the effective user ID of the calling process
getgid(), getegid()

- **getgid()** – get the real group ID of the calling process
  - `gid_t getgid(void);`

- **getegid()** – get the effective group ID of the calling process
  - `gid_t getegid(void)`
fork() – creates a new process

- `pid_t fork(void);`

The parent and children are determined by the return of `fork()`

- returns 0 for the parent process
- returns PID of new process for child
Fork details

- The child is a clone of the parent, it has copies of the parents:
  - Address space (heap, stack, variables, stdio bufs)
  - File descriptors
  - Code (may be shared since it’s read only)

- After the fork() call, each process executes as though it was the one that called fork()
  - Usually the child and parent preform different tasks, the parent/child sections are determined by checking the return of fork()
File Sharing Between Parent and Child

- Each process has its own file descriptors

- The underlying kernel structures for managing the files are shared

- Specifically, the offsets are shared

- This means that shared output to the same file will work correctly

- Important if stdout has been redirected to a file before fork()
```c
#include "apue.h"

int globvar = 6; /* external variable in initialized data */
char buf[] = "a write to stdout\n";

int
main(void)
{
    int var; /* automatic variable on the stack */
    pid_t pid;

    var = 88;
    if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
        err_sys("write error");
    printf("before fork\n"); /* we don’t flush stdout */

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) { /* child */
        globvar++;
        /* modify variables */
        var++;
    } else {
        sleep(2); /* parent */
    }

    printf("pid = %ld, glob = %d, var = %d\n", (long)getpid(), globvar, var);
    exit(0);
}
```

**Figure 8.1** Example of fork function
Normal cases

- Input and output isn’t redirected, so it doesn’t matter
- Parent waits for child to finish
  - Parent gets updated file pointers when it resumes executing
- Child redirects its input/output so no shared file pointers
Differences between parent and child

- The return value from fork()
- The process ID’s
- The process ID of the parent
- The accumulated CPU time
- File locks
- Pending alarms
- Pending signals
Figure 8.2  Sharing of open files between parent and child after fork
vfork()

- vfork() - create a child process and block parent
  - pid_t vfork(void);

- Usually used with exec()
```c
#include "apue.h"

int globvar = 6;    /* external variable in initialized data */
int main(void)
{
    int var;          /* automatic variable on the stack */
    pid_t pid;

    var = 88;
    printf("before vfork\n");    /* we don't flush stdout */
    if ( ((pid = vfork()) < 0) ) {
        err_sys("vfork error");
    } else if (pid == 0) {  /* child */
        globvar ++;          /* modify parent's variables */
        var ++;
        _exit(0);            /* child terminates */
    }

    /* parent continues here */
    printf("pid = %ld, glob = %d, var = %d\n", (long)getpid(), globvar, var);
    exit(0);
}
```

**Figure 8.3** Example of vfork function
Three ways to terminate normally and two ways to terminate abnormally

**Normal Termination**
- Return from main()
- Call exit()
  - ...which calls _exit()s

**Abnormal Termination**
- Receive certain signals from parent or kernel
- Call abort (sends SIGABRT to self)

**Termination status:** exit parameter or other status from kernel
Process Termination Details

- When a process terminates the parent receives SIGCHLD

- `wait()` allows a parent process to wait for a child process to terminate.

- When a process terminates, the kernel maintains a small amount of info until the parent calls `wait()`
  - Such a process is a zombie until the parent calls `wait()`

- If the parent terminates first, the child is inherited by init
The parent receives a SIGCHLD

Parent can
  - Ignore the signal (default)
  - Set up a signal handler that is called when the signal arrives

Use wait() to wait for the child to finish
  - Blocks parent
  - Returns when a child process terminates
    - If the child is a zombie it returns immediately
  - Returns child’s PID
wait() and waitpid()

- **wait()** – wait for any child process to terminate
  - `pid_t wait(int* statloc);`

- **waitpid()** – wait for a specific child to terminate
  - `pid_t waitpid(pid_t pid, int* statloc, int options);`

- Statloc contains the child’s termination status
  - the child’s parameter to `exit()`, possibly with extra info
```c
#include "apue.h"
#include <sys/wait.h>

int main(void)
{
    pid_t pid;
    int status;

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)              /* child */
        exit(7);

    if (wait(&status) != pid)       /* wait for child */
        err_sys("wait error");
    pr_exit(status); /* and print its status */

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)              /* child */
        abort(); /* generates SIGABRT */

    if (wait(&status) != pid)       /* wait for child */
        err_sys("wait error");
    pr_exit(status); /* and print its status */

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)              /* child */
        status /= 0; /* divide by 0 generates SIGFPE */

    if (wait(&status) != pid)       /* wait for child */
        err_sys("wait error");
    pr_exit(status); /* and print its status */

    exit(0);
}
```

Figure 8.6 Demonstrate various exit statuses
Race Conditions

- A race condition
  - Two or more processes access the same shared data
  - The outcome of the processing depends upon the order in which the processes execute

- Example: two processes do \( x = x + 1 \), where \( x \) is a shared variable

- Need some form of synchronization
  - Signals
  - File locks
  - Semaphores
  - ...


```c
#include "apue.h"

static void charatatime(char *);

int
main(void)
{
    pid_t pid;

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

**Figure 8.12** Program with a race condition
```c
#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);
}
```

**Figure 8.13** Modification of Figure 8.12 to avoid race condition
Running a different program

- fork() allows us to clone a process
  - We want parent and child to have different functionality
  - … but parent and child are clones?

- exec() allows us to run a new program
  - Often parent waits for exec() to finish

- fork() creates a new process and exec() makes it run a new program
  - Same PID, new text, data, BSS, stack, heap
exec*()  

- int execl(const char *path, const char *arg, ...);
- int execlp(const char *file, const char *arg, ...);
- int execlen(const char *path, const char *arg, ..., char * const envp[]);
- int execv(const char *path, char *const argv[]);
- int execve(const char *file, char *const argv[]);
- int execvpe(const char *file, char *const argv[], char *const envp[]);
<table>
<thead>
<tr>
<th>Function</th>
<th>pathname</th>
<th>filename</th>
<th>fd</th>
<th>Arg list</th>
<th>argv[]</th>
<th>environ</th>
<th>envp[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>exec1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>execlp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>execle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>execv</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>execvp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>execve</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fexecve</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(letter in name) p f l v e

**Figure 8.15** Relationship of the seven exec functions
```c
#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 (execl("/home/sar/bin/echoall", "echoall", "myarg1",
                  "MY ARG2", (char *)0, env_init) < 0)
            err_sys("execl 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 8.16** Example of exec functions
Variations of exec()

- ‘L’ versions use a list of parameters
- ‘V’ versions use an argv[] parameter
- ‘E’ versions include an environment parameter
- ‘P’ versions search PATH for executable
setuid(), setgid(),

- setuid() – set effective user ID of the calling process
  - int setuid(uid_t uid);

- setgid() – set effective group ID of the calling process
  - int setgid(gid_t gid);

- If the process has superuser privileges
  - setuid() sets the real user ID, effective user ID, and saved set-user-ID to uid

- If the process does not have superuser privileges, but uid is the real user ID or the save set-user-ID
  - setuid() sets the effective user ID to uid

- If neither is true, errno is set to EPERM and an error is returned
```c
#include "apue.h"

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

Figure 8.25 Print real and effective user IDs
Three user IDs

- Only a super-user process can change the real user ID
- The effective user ID is set by the exec functions, only if the setuid bit is set for the program file
  - Can call setuid any time to set the effective user ID to the real user ID or the saved set-user-ID
- The saved set-user-ID is copied from the effective user ID by exec

<table>
<thead>
<tr>
<th>ID</th>
<th>exec set-user-ID bit off</th>
<th>exec set-user-ID bit on</th>
<th>setuid(uid)</th>
</tr>
</thead>
<tbody>
<tr>
<td>real user ID</td>
<td>unchanged</td>
<td>unchanged</td>
<td>superuser</td>
</tr>
<tr>
<td>effective user ID</td>
<td>unchanged</td>
<td>unchanged</td>
<td>set to uid</td>
</tr>
<tr>
<td>saved set-user ID</td>
<td>copied from effective user ID</td>
<td>copied from effective user ID</td>
<td>set to uid</td>
</tr>
</tbody>
</table>

Figure 8.18  Ways to change the three user IDs
setreuid(), setregid(), seteuid() and setegid()

- setreuid() – swap real and effective user ID
  - int setreuid(uid_t ruid, uid_t euid);

- setregid() – swap real and effective group ID
  - int setregid(gid_t rgid, gid_t egid);

- seteuid() – set effective user ID
  - int seteuid(uid_t uid);

- setegid() – set effective group ID
  - int setegid(gid_t gid);
Figure 8.19  Summary of all the functions that set the various user IDs
Interpreter Files

- Text files starting with: “#!” pathname [args] on the first line
- Recognized by the kernel
- Kernel starts the interpreter specified by the pathname
  - Redirects the rest of the file to interpreter’s stdin
#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);
}

Figure 8.20 A program that execs an interpreter file
#!/usr/bin/awk -f
# Note: on Solaris, use nawk instead
BEGIN {
    for (i = 0; i < ARGC; i++)
        printf "ARGV[%d] = %s\n", i, ARGV[i]
    exit
}

Figure 8.21  An awk program as an interpreter file
```c
#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 8.22** The `system` function, without signal handling
system()

- system() – does fork(), exec(), and waitpid() to execute cmdstring
  - int system(char* cmdstring);

- Waits for any child to finish

- Don’t call system in a set-user-ID program
#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 8.24** Execute the command-line argument using `system`
Process Times: times()

- times() – Fills in the tms struct and returns the current clock time (in seconds)
  - clock_t times(struct tms* buf);

```c
struct tms {
    clock_t tms_utime; /* user time */
    clock_t tms_stime; /* system time */
    clock_t tms_cutime; /* user time of children */
    clock_t tms_cstime; /* system time of children */
};
```
Process accounting

- Record accounting info each time a process terminates
  - CPU time used,
  - user/group ID
  - starting time
  - ... the rest is in struct acct

- Two issues
  - We don’t get info from process if it doesn’t terminate
  - We don’t know exact start time due to granularity issues because it uses calendar time (seconds)
User Identification: getlogin()

- getlogin() – gets pointer to user name
  - char* getlogin(void);
  - Returns NULL on error
Process Scheduling

- **nice()** - run a program with modified scheduling priority
  - int nice(int incr);
  - return new nice value or -1 on error

- **getpriority()** - get program scheduling priority
  - int getpriority(int which, int who);

- **setpriority()** - set program scheduling priority
  - int setpriority(int which, int who, int prio);
/proc

- Provides access to the state of each process and light-weight process in the system

- The name of the entry for a process is /proc/[pid] where pid is the PID of the process

- Actual process state is contained in files in the directory

- The owner of the files is determined by the user ID of the process it describes
Accessing /proc

- Standard system calls are used to access /proc:
  - open(), close(), read(), and write()
- Most files can only be opened for reading
- ctl and lwpctl (control) files can only be opened for writing
- Address Space (as) files contain the image of the running process and can be opened for reading and writing
  - Data can be transferred to and from the address space using read and write
- Files can be opened exclusively with O_EXCL
Information and Control Operations

- `#include <procfs.h>`
  - Contains definitions of data structures and message formats used with these files

- Every process contains at least one LWP
  - Each LWP represents a flow of execution that is independently scheduled by the OS
  - All LWPs in a process share its address space and many other attributes
/proc Directory Structure

- as (r/w): address space image, can seek
- ctl (w): messages can be written to control process state or behavior
- status (r): state information
- psinfo (r): misc info
- cred (r): description of credentials
- sigact (r): array of sigaction structs
- map (r): virtual address map
- fd (r): directory containing references to open files
- usage (r): usage info (times, faults, blocks, msgs, sigs, syscalls, context switches)