CH 8: Process Control Notes
Advanced Programming in the Unix Environment 3e
*CIT-5950

Tyler Ryan

February 2, 2023

Contents

*Fall 2023

1 Chapter Summary

A thorough understanding of the UNIX System's process control is essential for advanced programming. There are only a few functions to master: ***fork()***, the ***exec()*** family, ***_exit***, ***wait()***, ***waitpid()***.
These primatives are used in many applications. The ***fork()*** function also gave us an opportunity to look at **race conditions**.

Our examination of the ***system()*** function and **process accounting** gave us another look at all these process control functions. We also looked at another variation of the ***exec()*** functions: **interpreter files** and how they operate.

An understanding of the various User IDs and Group IDs that are provided– Real, Effective, and Saved–is critial to writing safe set-userID programs.

2 Process Identifiers

2.1 The PID

- Process ID (PID).
- Every process has one.
- Always unique.
- All PIDs are non-negative numbers.
- Are reused after processes are terminated.
  - However, using algorithms to delay this reuse prevents new PIDs from being mistaken for previous PIDs with the same number.
- Because the PID is the only well known identifier of a process that is always unique, it is often used as a piece of other identifiers to gurantee uniqueness.

2.2 Scheduler and Init Processes

PID 0 is usually the *scheduler process* and is often known as the *swapper*.
PID 1 is usually the *init process* and is invoked by the kernel at the end of the bootstrap procedure.

- Responsible for bringin up the UNIX System after the kernel has been bootstrapped.
- Reads the system-dependent inizialization files –*/etc/rc** or */etc/inittab* and */etc/init.d/*– and brings the system to a certain state, such as multiuser.
- **Never Dies**.
- It is a normal user process, not a system process within the kernel, like the swapper.
- Does have Superuser Privileges.
- Becomes the parent process of any orphaned child process.
- New program file is *sbin/init*.
- Old program file was */etc/init*.

2.3 Additional Process Identifiers

In addition to the PID, there are other identifiers for every process (though not unique!). Below are the functions that return those identifiers.

Listing 1: Additional Process Identifier Functions.

```
#include <unistd.h>

pid_t getpid(void);      # Returns: PID of calling process.
pid_t getppid(void);     # Returns: Parent Process ID (PPID) of calling process.
uid_t getuid(void);      # Returns: Real User ID (UID) of calling process.
uid_t geteuid(void);     # Returns: Effective User ID (EUID) of calling process.
gid_t getgid(void);      # Returns: Real Group ID (GID) of calling process.
gid_t getegid(void);     # Returns: Effective Group ID (EGID) of calling process.
```

3 The *fork()* Function

Listing 2: fork() Prototype

```
#include <unistd.h>

pid_t fork(void);        # Returns: 0 in child, PPID, or -1 if error.
```

3.1 About *fork()*

- The new process created by fork() is called the **Child Process**.
- fork() is called once, but returned twice.

- A Parent can have more than one child, so to uniquely identify it's children, the children return their PIDs.
  - There is no function that tells the parent the PIDs of it's children.
- fork() returns 0 to a child, because a child can only have one parent.
  - Additionally, a child can always call *getppid()* to obtain the PID of the parent.
- The child process **copies** all data from the parent process.
  - This includes variables from your source code!
  - The child copies these variables, but does not alter the original values of the parent.

The two uses for fork are:

1. **When a process wants to duplicate itself so that the parent and child can each execute different sections of code at the same time** *This is common for network servers*–the parent waits for a serice request from the client. When the request arrives, the parent calls *fork()* and lets the child handle the request. The parent goes back to waiting for the next service request to arrive.
2. **When a process wants to execute a different program.** *Common for shells.* In this case, the child does an *exec()* right after it returns from *fork()*.

The two main reasons for *fork()* to fail are

1. If too may processes are already in the system. This usually means that something else is wrong.
2. If the total number of processes for the real user ID (UID) exceeds the system's limit. **CHILD_MAX** specifies this maximum number.

Below is an example of the fork function. As a result of the sleep() function, the child should execute first. However, it **is normally never known whether the Parent Process will execute before the Child Process or vice versa.**

3.2 *fork()* Example Code

Listing 3: Example of the fork() function.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <wait.h>
#include <sys/types.h>


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

int main()
{
    int var; // Automatic variable on the stack.
    pid_t pid;

    var = 88;

    // Not flushing STDOUT
    int result = write(STDOUT_FILENO, buf, sizeof(buf) -1);
    if (result != sizeof(buf) -1)
    {
        perror("error");
    }

    printf("Before Fork\n");

    pid = fork();

    if (pid < 0)
    {
        perror("Error");
        exit(EXIT_FAILURE);
    }

    else if (pid == 0) // Child Process
    {
        // Modifies these variables.
        glob++;
        var++;
    }
    else  // Parent Process
    {
        // Making parent sleep for 2 seconds
        // This will increase the likelyhood of the Child
        // executing first.
        sleep(2);
    }

    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(EXIT_SUCCESS);
}
```

If you run: **gcc main.c && ./a.out**, then the output will be:

Listing 4: fork() function output to terminal.

```
a write to stdout
Before Fork
pid = 145800, glob = 7, var = 89
pid = 145794, glob = 6, var = 88
```

If you run: **gcc main.c && ./a.out > temp.txt && cat temp.txt** or run it in Vim, the output will be:

Listing 5: Fork function output to file.

```
a write to stdout
Before Fork
pid = 146403, glob = 7, var = 89
Before Fork
pid = 146402, glob = 6, var = 88
```

Don't think too hard about the second "Before Fork". It has to do with the way C flushes it's print statements. Just be aware of it.

3.3 File Sharing

One characteristic of fork is that all file descriptors that are open in the parent are duplicated in the child. There are two normal cases for handling the file descriptors after a fork.

1. **The parent waits for the child to complete.**
   - Parent does not need to do anything with descriptors.
   - When the child terminates, any of the shared descriptors tat the child read from or wrote to will ahve their file offsets updated accordingly.
2. **Both the parent and the child go their own ways.**
   - After the *fork()*, the parent closes the file descriptors that it doesn't need, and the child does the same things.
   - Neither interferes with the other's open descriptors. This scenario is often the case with network servers.

4 Exit Functions

4.1 Normal Terminations

A process can terminate normally in the following 5 ways.

- ***return:*** from a main function is equivalent to ***exit***.
- ***exit():*** Defined by ISO C and includes the calling of all exit handlers that have been registerd by the calling ***atexit***.
- ***_exit*** and ***_Exit:***
  - _Exit and _exit are synonomous and do not flush standar I/O streams.
  - ISO C defines _Exit to provide a way for a process to terminate without running exit handlers or signal handlers.
  - The _exit function is called by ***exit()*** and handles the UNIX system-specific details.
- ***pthread_exit():*** calling this function from the last thread in the process.

4.2 Abnormal Terminations

The 3 forms of abnormal terminations are:

- ***abort():*** A special case of the ntext item, as it generates the **SIGABRT** signal.
- By receiving certain **signals**.
- The last thread reponds to a cencellation request.
  - By default, cancellation occurs in a deferred manner: one thread requests that another be canceled, and somtime later, the target thread terminates.

Additional Info Regarding Exit Functions

- Regardless of how a process terminates, the same code in the kernel is eventually executed.
- In the case of an abnormal termination, the kernel generates the termination status to indicate the reason for the abnormal termination, not the process itself.
- Regardless of how a process was terminated, the parent of the process can obtain the termination status from either the ***wait()*** or the ***waitpid()*** function.
- If the parent terminates before the child, the **init process** becomes the parent process of said child process.
  - This way, we're guaranteed that every process has a parent.
- A process that has terminated, but whose parent has not yet *waited* for it is called a **Zombie Process**.
  - In other words, if a child terminates before the parent, but the parent does not call a ***wait()*** function, then it is a Zombie. This is because the ***wait()*** function literally removes zombie processes.
  - Ths ***ps*** command prints the state of a zombie process as **Z**.

5 *wait()* and *waitpid()* Functions.

5.1 Function Prototypes

Listing 6: Wait Function Prototypes

```
#include <sys/wait.h>
// Also <wait.h> works on Fedora as of Feb. 2, 2023

pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
```

5.2 Wait Function Return Values

Return Value of waitpid()	Description
pid == -1	Waits for any cild process. In this respect, *waitpid()* is equivalent to *wait()*.
pid > 0	Waits for thechild whose PID equals *pid*.
pid == 0	Waits for any child whose PID equals that of the calling process.
pid <	Waits for any child whose PID equals the absolute value of *pid*.

Table 1: *waitpid()* return value descriptions.

NOTE: The ***waitpid()*** function returns the PID of the child that terminated and stores the child's termination status in the memory location pointed to by ***statloc***.

With ***wait()***, the only real error is if the calling process has no children. With ***waitpid()***, it is also possible to get an error if the specified process or process group does not exist or is not a child of the calling process.

5.3 Useful Macros

Remember: You can run ***man 2 wait*** and find a list of these macros.

Macro	Description
WIFEXITED(status)	True if status was returned for a child that terminated normally. ***WEXITSTATUS(status)*** can be called after WIFEXITED() to fetch the actual return status.
WIFSIGNALED(status)	True if status was returned for a child that terminated abnormally. ***WTERMSIG(status)*** can be called after WIFSIGNALED() to fetch the signal number that caused the termination. ***WCOREDUMP(status)*** returns true if a core file of the terminated process was generates.
WIFSTOPPED(status)	True if status was returned for a child that is currently stopped. ***WSTOPSIG(status)*** can be called to fetch the signal number that caused the child to stop.
WIFCONTINUE(status)	True if status was returned for a child that has been continued after a job control stop (XSI extention to POSIX.1; ***waitpid()*** only).

5.4 About *wait()* and *waitpid()*

When a process terminates, either normally or abnormally, the kernel notifies the parent by sending the ***SIGCHLD*** signal (asynchronously) to the parent.

Remember: If a child is terminated *before* the parent and the parent is not notified of it's exit status (***wait()*** or ***waitpid()*** was not called), then that child turns into a **Zombie**.

A process that calls ***wait()*** or ***waitpid()*** can:

- Block, if all tis children are still running.
- Return immediately with the termination status of a child, if a child has terminated and is waiting for its termination status to be feched.
- Return immediately with an error, if it doesn't have any child processes.

The differences between ***wait()*** and ***waitpid()*** are as follows:

- The ***wait()*** function can block the caller until a child process terminates, whereas ***waitpid()*** has an option that prevents it from blocking.
- The ***waitpid()*** function doesn't wait for the child that terminates first;
  - In other words, if there is more than one child, ***wait()*** returns on termination of *any* of the children, whereas ***waitpid()*** can choose which child to wait on.

  it has a number of options that control which process it waits for.

3 additional features provided by ***waitpid()***:

- Allows us to wait for one particular process, whereas the ***wait*** function returns the status of the first terminated child.
- Provides a *nonblocking* version of ***wait()***.
- Provies support for *job control* with the ***WUNTRACED*** and ***WCONTINUED*** options.

Additional Info About the Wait Functions

- If the process is calling ***wait()*** because it received the ***SIGCHLD*** signal, we expect ***wait()*** to return immediately. But if we call it at any random point in time, it can block.
- If a child has already terminated and is a zombie, ***wait()*** returns immediately with that child's status.
- If you don't care about the children's termination status, the ****statloc*** parameter can be ***NULL***.

5.5 Other Wait Functions

Listing 7: Other wait functions mentioned in the book.

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

// Only Solaris provides this I believe
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);

#include <sys/types.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>

// These are available in FreeBSD, Linux, OSX, Solaris
pid_t wait3(int *statloc, int options, struct rusage *rusage);
pid_t wait4(pid_t pid, int *statloc, int options, struct rusage *rusage);
```

6 Race Conditions

6.1 Basics

- 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.
- If a process wants to wait for a *child* to terminate, use one of the **wait** functions.
- If a process wants to wait for its *parent* to terminate, a loop of the following form could be used (AKA Polling)

Listing 8: Example of Polling

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

- The problem with **polling** is that it wastes CPU time.
- To avoid race conditoins and to avoid polling, some form of **signaling** is required between multiple processes.
  - Various forms of **interprocess communication (IPC)** can also be used.

7 *exec()* Functions

7.1 Function Prototypes

Listing 9: List of exec functions

```
#include <unistd.h>

// 'e' as in 'environment'
// 'v' as in 'vector'

/* 'p' for 'filename'. E.g. 'bash'
 * It has a 'p' because it looks up a filename (bash)
 * in your PATH. This way you don't have to specify
 * the WHOLE path. E.g. '/usr/bin/bash'
 */

// 'pathname' == 'whole path to file'. E.g. '/usr/bin/bash'
int execl(const char *pathname, const char *arg0, ..., (char *) NULL);

int execle(const char *pathname     ,
           const char *arg0         ,
           ...                      ,
           (char *) NULL            ,
           char * const env[]);

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

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

int execlp(const char *filename, const char *arg0, ..., (char *) NULL);

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

int execl(const char *pathname, const char *arg0, ..., (char *) NULL);

                                        All 6 return -1 on error, no return on success
```

7.2 About Exec Functions

- When a process calls one of the ***exec()*** functions, that process is completely replaced by the new program, and the new program starts executing at its **main** function.
- The PID does not change across an ***exec()***, because a new process is not created;
  - ***exec()*** just replaces the current process–its data, heap, and stack segments–with a brand new program from disk.

7.3 Inherited Properties Accross Execs

The PID does not change after an ***exec()***, but the new program inherits additional properties from the calling process.

- PID and PPID
- UID and GID
- Supplementary GIDs
- PGID
- SID
- Controlling Terminal
- Time left until alarm clock
- CWD
- Root Directory
- File mode creation mask
- File locks
- Process signal mask
- Pending signals
- Resource limits
- Values for ***tms_utime***, ***tms_stime***, ***tms_cutime***, ***tms_cstime***

7.4 Other Characteristics Accross Execs

- The default is to leave descriptors open accross the ***exec()*** unless we specifically set the close-on-exec flag using ***fcntl***.
- POSIX.1 specifically equires that open directory streams (***opendir()***) be closed accros an ***exec()***.
- Real User ID (UID) an Real Group ID (GID) remain the same accros the ***exec()***, but the Effective IDs (EID) can change, depending on the status of the Saved Set-User-ID.

8 Changing User IDS (UIDs) and Group IDs (GIDs)

8.1 Function Prototypes

Listing 10: User/Group ID Function Prototypes

```
#include <unistd.h>

// UID == Real User ID
int setuid(uid_t uid);
int setgid(gid_t gid);

// EUID == Effective User ID
int seteuid(uid_t uid);
int setegid(gid_t gid);
                                        All return: 0 if OK, -1 on error
```

In the UNIX System, privileges, such as being able to change the system's notion of the current date, and access control, such as being able to read or write a particular file, are based on user and group IDs.

When our programs need additional privileges or need to gain access to resources that they currently aren't allowed to access, they need to change their User or Group ID to an ID that has the appropriate privileges or access.

Similarly, when our programs need to lower their privileges or prevent acces to certain resources, they do so by changing either their User ID or Group ID to an ID without the privilege or ability access to the resource.

9 Interpreter Files: Nothing Here

This is pretty easy, just google it real quick.

10 *system()* Function

10.1 Function Prototype

Listing 11: system() prototype

```
#include <stdlib.h>

/* If either fork() fails or waitpid() returns an error other than
 * EINTR, system() returns -1 with errno set to indicate the error.
 *
 * If exec() fails, implying that the shell can't be executed, the return
 * value is as if the shell had executed exit(127).
 *
 * Otherwise, all 3 functions—fork(), exec(), and waitpid()--succeeded, and
 * the return value from system() is the termination status of the shell, in
 * the format specified for waitpid().
 */

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

Returns

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

10.2 About system()

The ***system()*** function combines the***fork()***, ***exec()***, and ***waitpid()*** functions into one.

10.3 Basic Example

Listing 12: A basic use case for system()

```
#include <stdlib.h>

int main()
{
    system("echo 'hello there' > hello.txt && cat hello.txt");
}

                                                    Output:
                                                    hello there
```

10.4 Recreate A Basic system() function

Listing 13: Simple system() recreation

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

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

    if (command == NULL)
        return(EXIT_FAILURE);

    pid = fork();

    if (pid < 0)
        status = -1;
    // Child Process
    else if (pid == 0)
    {
        execl("/bin/bash", "random string", "-c", command, (char *) NULL);
        _exit(127);
    }

    /* pid > 0
     *
     * Waits for any child whose process group ID (PGID)
     * equals the calling process ID.
     */
    else
    {
        while (waitpid(pid, &status, 0) < 0)
        {
            if (errno != EINTR)
            {
                status -1;
                break;
            }
        }
    }
    return status;
}
```

Explanation

- Example command: ***system("ls /usr/bin | grep ^b > out.txt")***
- The shell's ***-c*** option allows us to pass in a command as if we had the terminal open.
- The advantage in using ***system()*** instead of ***fork()*** and ***exec()*** directly, is that ***system()*** **does not require error handling and all the required signal handling.**

11 Set-User-ID Programs

11.1 Running Set-User-ID Programs As Superuser

It is a security hold to call ***system()*** from a set-user-ID program and should never be done.

Below is an example of such a program. Imagine running it as a super user. How could this be dangerous?

11.2 Security Hole Example

Listing 14: Dangerous program!

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

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

    if (argc < 2)
    {
        printf("Error: command line argument required.");
        exit(EXIT_FAILURE);
    }

    if ((status = system(argv[1])) < 0)
    {
        perror("ERROR");
        exit(EXIT_FAILURE);
    }

    printf("Status: %d\n", status);
    exit(EXIT_SUCCESS);
}
```

Perhaps this will help. What if we were to run the following commands?

```
$ gcc main.c
$ su - root
# ./a.out "rm -rf /"
```

Running this command would remove everything on this person's computer!

To be safe, if a program is running with special permissions–either set-user-ID or set-group-ID–and wants to spawn another process, a process should use ***fork()*** and ***exec()*** **directly**. This will ensure that permissions change back to normal after the ***fork()***, before calling ***exec()***.

Never run *system()* from a set-user-ID or a set-group-ID program.

12 Process Accounting: Nothing Here

Boring. Look in the book.

13 User Identification

13.1 *getlogin()* Prototype

Listing 15: Get User Login Name

```
// Equivalent to Bash's "$ echo $USER" or "$ echo $LOGNAME"
#include <unistd.h>

char * getlogin(void);

                                Returns: char * of login name if OK, NULL on error.
```

14 Process Times

14.1 Function Prototype

Listing 16: times() prototype

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

clock_t times(struct tms * buf);
            Returns: elapsed wall clock time in clock ticks if OK, -1 on Error
```

14.2 tms Struct

Listing 17: tms (Times) Struct

```
/* Note: This struct does not contain wall clock time
 * This is because its return value IS the wall clock time.
 *
 * Also, these times are reported in CLOCK TICKS specified
 * by sysconf(_SC_CLK_TCK);
 *
 * To get this time into seconds, you must do the following.
 * long clock_ticks = sysconf(_SC_CLK_TCK);
 * long seconds = tms_utime / (double) clock_ticks;
 */

struct tms
{
    clock_t tms_utime;  // user CPU time
    clock_t tms_stime;  // system CPU time
    clock_t tms_cutime; // user CPU time of children
    clock_t tms_cstime; // system CPU time of children
}
```

14.3 Explanation

3 Types of Process Times:

- **Wall Clock Time**
- **User CPU Time:** Time spent running your application's functions.
- **System CPU Time:** Time spent running OS (Kernel) functions.

14.4 *times()* Function And *tms* Struct Example Code

Listing 18: Timing Program Execution

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/times.h>
#include <time.h>

/* A program that takes in a list of commands and
 * prints how long they took in various ways.
 */

static void
track_time(clock_t real_time, struct tms * tms_start, struct tms *tms_end);

int
main(int argc, char * argv[])
{
    struct      tms tms_start, tms_end;
    clock_t     start, end;
    clock_t     real_time;
    int         status;
    char * cmds[] = {"sleep 5", "date"};

    for (int i = 0; i < 2; i++)
    {
        start     = times(&tms_start);
        system(cmds[i]);
        end       = times(&tms_end);
        real_time = end - start;
        track_time(real_time, &tms_start, &tms_end);
    }

    exit(EXIT_SUCCESS);
}

static void
track_time(clock_t real_time, struct tms * tms_start, struct tms *tms_end)
{
    static long clktck = 0;
    clktck = sysconf(_SC_CLK_TCK);

    printf(" real: %.2f seconds\n",  real_time / (double) clktck);
    printf(" user: %.2f seconds\n",
           tms_end->tms_utime - tms_start->tms_utime / (double) clktck);
    printf(" sys:  %.2f seconds\n",
           tms_end->tms_stime - tms_start->tms_stime / (double) clktck);
    printf(" child user: %.2f seconds\n",
           tms_end->tms_cutime - tms_start->tms_cutime / (double) clktck);
    printf(" child sys:  %.2f seconds\n",
           tms_end->tms_cstime  - tms_start->tms_cstime / (double) clktck);
}
```