Programs and Processes

COMP 321

Dave Johnson

1

The State of a Running Process

User-visible state

- The process's address space (it's memory)
- Current CPU register values: PC, SP, R0, R1, R2, R3, R4, R5, R6, R7, R8, …
  - (Intel CPUs: rax, rbx, rcx, rdx, rbp, rsp, rsi, rdi, r8, r9, r10, r11, …, r15)

State inside the kernel (traditionally, the Process Control Block, or PCB)

- One per process, collects together all OS state for that single process
  - The process's process ID
  - Saved CPU register values (when not currently running on the CPU)
  - Table of the process's open file descriptors
  - Lots of other bookkeeping info about the process

2

Processes and Threads

Classically, a process has a single thread of execution

- One point of execution progress, one set of register values
- Example:

```
main( … )
{
        …
        return 0
}
```

But a process may be "multithreaded"

- Multiple "***threads***" sharing the same address space
- All running concurrently, all "at once," cooperating
- Threads are also called ***lightweight processes***

Why Multiple Threads Sharing an Address Space?

Easy cooperation between these threads since they share all data, such as

- ***A windowing GUI system***
  - All threads share the same data structures of what's on the screen
  - One thread tracking the mouse on the screen
  - One thread for each open window
- ***Microsoft Word***
  - On thread managing the user's keyboard
  - One thread doing line breaks, one doing paragraph breaks, one page breaks
  - One thread doing spell checking, one for grammar checking, etc.

For now, we will limit ourselves to a ***single*** thread of control: A "classical" process

The main() Procedure of Any Program

Every program has a main() procedure, with these arguments

```
int  main(int argc,  char *argv[])
```

Suppose the program was run as

```
./myprog Hello world
```

- argc = 3
- argv[0] = “./myprog”
  argv[1] = “Hello”
  argv[2] = “world”
  argv[3] = NULL

By convention, argv[0] always equal to the name the program was run as

And argv[argc] will be NULL

Example main()

```
#include <stdio.h>
int main(int argc,  char *argv[])
{
        printf("program executed as %s, argc = %d\n", argv[0], argc);
        for (int i = 1; i < argc; i++)
                printf("arg % d = %s\n", i, argv[i]);
        return 0;
}
```

```
$ cc -o foo foo.c
$ ./foo aaa bb ccccc
```

output

```
program executed as ./foo, argc = 4
arg  1 = aaa
arg  2 = bb
arg  3 = cccc
```

But main() is Really Just a Regular Procedure

The <u>name</u> "main" is special (expected), but it works like any other procedure

- It gets called called like a normal procedure (because it is a normal procedure)
- It returns like a normal procedure (because it is a normal procedure)

Some mysteries

- Who actually does the procedure call to main(), passing it those arguments (argc and argv) with the right values?
- When main() executes any "return" statement, where does that return go back to?
- And what happens if the execution of main() just "falls of the bottom" of that procedure (with no explicit "return" statement or "return" value)?

Terminating a Process: Returning from main()

```
return value;       /* from inside main() */
return;             /* from inside main(), or "falling off the bottom" of main() */
```

Doing "return value" from main(), with some explicit return value

- Then "value" is the return value from that function

Falling off the bottom of a function is the same as "return" from that function

- Whatever value happens to be in some specific CPU register (e.g., %rax on x86_64) becomes the function's return value
- (So there ***always*** is ***<u>some</u>*** return value from ***any*** function)

The "exit status" of the process

- The least significant byte of the main() return value (i.e., value & 0xff)

Terminating a Process: Calling exit() vs. _exit()

```
[[noreturn]]  void  _exit(int status);
[[noreturn]]  void  exit(int status);
```

_exit(status) is a <u>kernel</u> call

- Just causes the process to immediately terminate

exit(status) is a <u>library</u> call

- Executes various, e.g., "cleanup"-type functions, including flushing and closing all open stdio streams
- And finally calls the kernel call _exit(status)

The "exit status" of the process

- The least significant byte of status (i.e., status & 0xff)

The C Runtime "Wrapper" Code for main()

A small piece of assembly language code (traditionally called crt0)

- The ***real*** entry point for any program (the first code to execute)
- The kernel arranges to initialize the PC register when this program first begins execution to be equal to the address of this C runtime code
- Different systems are a bit different, but in general this code does
  - Packages command line arguments in argv[] format
  - Calls status = main(argc, argv);
  - Calls exit(status); /* library exit(), which ultimately calls kernel _exit() */
- This is how the return value from main() turns into the exit status for the process

What Should the Exit Status of a Process Be?

Any 8-bit value you want to exit with

- But there is a very old, long-standing convention
  - exit 0 for ***successful*** completion
  - exit any nonzero value (e.g., 1) for any ***error/failure*** completion
- Example: Using the bash shell

```
$ true
$ echo $?
0

$ false
$ echo $?
1
```

```
$ if true; then echo yes; else echo no; fi
yes

$ if false; then echo yes; else echo no; fi
no
```

- Recommendation: #include <stdlib.h>
- Use exit(EXIT_SUCCESS); or exit(EXIT_FAILURE);

The Termination of a Process

Regardless of how a process terminates

- The kernel frees the process's entire address space (all its memory)
- Closes all of the process's open files
- Frees all other resources held by the process, except …

The process becomes a "zombie" process

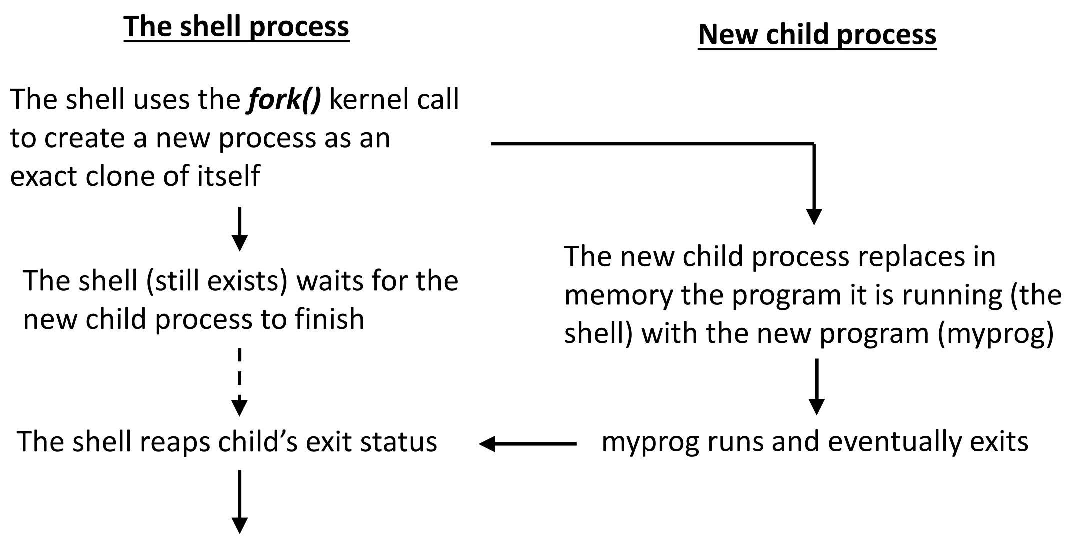
- Meaning that the kernel retains just enough of the process's state to be able to report the process's termination to the process's parent process
- The process in this "zombie" state remains until it is "reaped" (i.e., collected) by the parent process
- Once reaped, the child is then completely gone (and thus not reported again)

How the Shell Runs a Program

Consider the shell running a program such as ./myprog

- The program myprog has to get loaded into memory and executed
- The shell must still be there, ready for the next command
  - Can't just throw away the shell's process state and address space
  - And can't allow myprog to possibly mess up the shell
- Means myprog must run as a separate process, with its own address space
- Normally, the shell waits for myprog process to finish
- But if the command includes "&" (as in, e.g., "./myprog &"), the shell process and the new myprog process actively run concurrently

How the Shell Runs a Program



<u>The shell process</u>

<u>New child process</u>

The shell uses the ***fork()*** kernel call to create a new process as an exact clone of itself

The shell (still exists) waits for the new child process to finish

The new child process replaces in memory the program it is running (the shell) with the new program (myprog)

The shell reaps child's exit status

myprog runs and eventually exits

A Simple Example of the Shell Running “./myprog”

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

int main()
{
        pid_t pid = fork();

        if (pid == 0) {
                execl(“./myprog", "myprog", NULL);
        } else {
                wait(NULL);
        }

        return 0;
}
```

fork() returns ***twice***: once in the parent and once in the child

The only difference: fork() returns 0 in the child and nonzero in the parent

This simple example does not do any error checking, but you should!

Creating a New Process

```
pid_t  fork(void);
```

Creates a new process as an identical “clone” of the calling process

- Kernel creates a new PCB for the new process, substantially as a copy of the calling process’s existing PCB
- Kernel assigns new process a new pid, remembered in the kernel in child’s PCB
  - pids assigned in ascending order, wrapping around, skipping those in use
- Child address space is created as a ***copy*** of the calling process’s address space
  - Child thus ***appears*** to have called fork(), since the parent did call fork()
  - So fork() returns **twice**
    - Once (as normal) in the ***parent***: returns the new child’s pid
    - And once (appearing to be normal) in the ***child*** process: returns 0

Running a New Program in the Current Process

```
int  execve(const char *pathname,  char *const _Nullable argv[],
          char *const _Nullable envp[]);
int execl(const char *pathname,  const char *arg, ...
          /*, (char *) NULL */);
int  execv(const char *pathname,  char *const argv[]);
```

Replaces entire calling address space with program specified by pathname

- Many variants of "exec": execve() is a ***kernel*** call, others are ***library*** calls
- argv (or args ...) is a vector of the individual char * command line arguments
  - argv[0] should be the program name
- On success, does ***not return to caller*** – begins at entry point of new program (i.e., the "wrapper" code that calls its main() and then exits)

Waiting for a Child Process to Finish

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

wait() waits for any child to exit, waitpid() can wait for a specific child

- The parent "reaps" (i.e., collects) the exit status (and pid) of its child
- Calling wait() is equivalent to calling waitpid() with pid = -1, options = 0
  - can also give other reasons you want child status report (not relevant here)
- Both wait() and waitpid() return the pid of the child
- If wstatus != NULL, points to int into which to store the exit status of that child
- Returns -1 if no remaining children (none still running and none unreported), with errno = ECHILD

The “Tree” of All Processes

The parent/child relationship created by fork() makes all processes form a tree

- The initial process created at “boot time” is called “init” (pid = 1)
  - init forks one child for log in on each hardware terminal
  - init forks one child for each of several “daemons” (services) such as “sshd”
- Each login process eventually exec’s your login shell
  - When you log out, your shell exits
- init loops, calling wait() to reap each of its children
  - When init sees a log in process exit, it forks to create a new child for log in
- If any process terminates while some of its children are still running
  - They are inherited by (“reparented to”) init, so will be reaped when needed

A wait() Example

```
pid_t  wpid;
Int  wstatus, i;

for (i = 0; i < N; i++)
        if (fork() == 0)
                exit(100 + i);          /* exit a child process */

for (i = 0; i < N; i++) {
        wpid = wait (&wstatus);
        if (WIFEXITED(wstatus))
                printf("Child %d terminated with exit status %d\n",
                        wpid, WEXITSTATUS(wstatus));
        else
                printf("Child %d terminated abnormally\n", wpid);
}
```

WIFEXITED and WEXITSTATUS are defined by #include <sys/wait.h>

Sees processes in the arbitrary order they exit

A waitpid() Example

```
pid_t  pid[N], wpid;
Int  wstatus, i;

for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
                exit(100 + i);            /* exit a child process */

for (i = 0; i < N; i++) {
        wpid = waitpid(pid[i], &wstatus, 0);
        if (WIFEXITED(wstatus))
                printf("Child %d terminated with exit status %d\n",
                        wpid, WEXITSTATUS(wstatus));
        else
                printf("Child %d terminated abnormally\n", wpid);
}
```

WIFEXITED and WEXITSTATUS are defined by #include <sys/wait.h>

Sees processes in the order created, given their order in pid[] array

Getting Process IDs

```
pid_t  getpid(void);
pid_t  getppid(void);
```

fork() tells your parent the process ID of the new child process

- For the child process (or any process) to gets its own process ID, getpid()
  - Always succeeds, perhaps the simplest possible kernel call
- For the child process (or any process) to get the process ID of its own parent, get**p**pid()
  - This will generally be the process that did the fork() to create you
  - But if your parent already terminated earlier, you will have been inherited by (reparented to) init = process id 1

Possible Alternatives to Fork() in Other OSs

Example: Digital Equipment Corporation VMS Operating System

status = sys$creprc (12 arguments);

Example: Microsoft Windows Operating System

status = CreateProcess (10 arguments);

In both cases, many arguments are complex structs or arrays of structs

These operations are basically the combination of fork plus exec

- Creates a new process ***and*** starts that process running some specified program
- In Unix/Linux, fork and exec are two separate operations
  - And you can do ***anything*** you want to in the new process (the child) after the ***fork*** and before you make the child call ***exec*** to actually run the new program **. . .**

Examples: Between the fork and exec in the Child

- Change what file is open as standard output (stdout) in the child process
  - Example: ./foo > output_file
- Change while file is open as standard input (stdin) in the child process
  - Example: ./foo < input_file
- Change the child process's user id (change who the child is running as)
- Define resource limits for the child (e.g., how much memory can be used)

If done by the child after fork returns and before calling exec, these changes affect the child's execution but do not disturb the parent at all

Things like sys$creprc or CreateProcess must encode these kinds of changes in their complicated many arguments, instead of the **0** arguments for Unix fork