CPR E 308 Lab 2

Akira DeMoss
01-31-19
Section M

3.1

1.)
• output

```
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  3367  3289  0  80   0 -  7454 wait   pts/0    00:00:00 bash
0 S  1000  3381  3367  0  80   0 -  1127 hrtime pts/000:00:00 part1
4 R  1000  3382  3367  0  80   0 -  9004 -      pts/0    00:00:00 ps
```

• process name
part1

• process state (decode the letter!)
Interruptible sleep (waiting for an event to complete)

• process ID (PID)
Process ID is: 3381

• parent process ID (PPID)
Parent process ID is: 3367

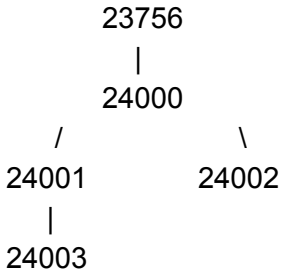
2.) What changes and doesn't change:
What changes is the process ID

3.) Find out the name of the process that started your programs. What is it, and what does it do?
The process that started the programs is Bash, which is a Unix shell. Bash is a command processor. It interprets the text entered either in a terminal or in a shell script and executes programs. E.g. ls, ps, mkdir are all bash executables in Linux which can be found in the /usr/bin path.

3.2

1.) Program output:
Process 24000's parent process ID is 23756
Process 24002's parent process ID is 24000
Process 24001's parent process ID is 24000
Process 24003's parent process ID is 24001

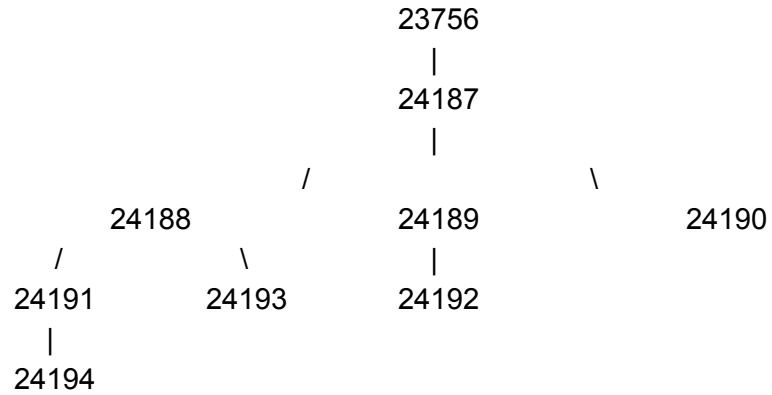
2.) Draw the process tree (label processes with PIDs)



3.) Explain how the tree was built in terms of program code

The first process represents the process from the main call in our code. Next the fork call creates a new process which becomes the child process of the caller, where process 24000 is the caller and process 24001 is the child. The second fork call spawns an additional process off of the caller aswell as one from the first child process. See additional experiment below with 3 fork calls.

Process 24187's parent process ID is 23756
Process 24189's parent process ID is 24187
Process 24188's parent process ID is 24187
Process 24191's parent process ID is 24188
Process 24194's parent process ID is 24191
Process 24190's parent process ID is 24187
Process 24193's parent process ID is 24188
Process 24192's parent process ID is 24189



4.) What happens when sleep is removed:

The parent will sometimes exit before getppid is called, thus the old ppid cannot be returned because it is no longer valid. Specifically, when the parent of a program exits, the **init** program (pid 1) becomes its parent. Thus, when the child prints out its parent after sleeping, it prints out pid 1.

3.3

1.) Completed program and its output

```
#include <stdio.h>
int main() {
int ret;
ret = fork();
if (ret == 0) {
/* this is the child process */
printf("The child process ID is %d\n", getpid());
printf("The child's parent process ID is %d\n", getppid());
} else {
/* this is the parent process */
printf("The parent process ID is %d\n", getpid());
printf("The parent's parent process ID is %d\n", getppid());
}
sleep(2);
return 0;
}
```

The parent process ID is 24257
The parent's parent process ID is 23756
The child process ID is 24258
The child's parent process ID is 24257

2.) Speculate why it might be useful to have fork return different values to the parent and child. What advantage does returning the child's PID have? Keep in mind that often the parent and child processes need to know each other's PID.
One reason would be that a developer may want to write code that kills a child process, but keep the parent process. Being able to distinguish between these two within program code would allow a developer that control. The advantage to the parent being able to return a child's PID is similar. Basically it gives the programmer / developer more control. This is especially useful considering that machines have limited resources and the fact that creating new processes is a heavyweight operation.

3.4

1.) Include small (but relevant) sections of the output

Child: 751
Child: 752
Child: 753
Child: 7Parent: 0
Parent: 1
Parent: 2
Parent: 3

Parent: 688
Parent: 689
Parent: 690
Parent: 6954
Child: 755
Child: 756
Child: 757

Child: 2820
Child: 2821
Child: 2822
Ch1
Parent: 692
Parent: 693
Parent: 694

Parent: 1342
Parent: 1343
Parent: 1344
Parent: 1ild: 2823
Child: 2824
Child: 2825
Child: 2826

2.) Make some observations about time slicing. Can you find any output that appears to have been cut off? Are there any missing parts? What's going on (mention the kernel scheduler)?

The output gets cut off as shown below

Child: 753
Child: 7Parent: 0
….
Parent: 6954
Child: 755

E.g. got cut off, then resumed later. The child counted from 753, then got cut off in the middle of 754.

With time slicing in respect to running this program it looks like the kernel scheduler splits up the time between the 2 processes as equally as possible which makes sense as their process states are both runnable and they are equivalent in priority level.

3.5

1.) Explain the major difference between this experiment and experiment 4. Be sure to look up what wait does (man 2 wait).

This program has a wait command. The wait command waits for the running process to complete. When comparing "ps -l" output of both of the programs, we can see that in the program from part4 the process corresponding to the parent's iteration is in a runnable state, while in part5 the process state is in interruptible sleep and wchan is set to do_wait. The effect of this is that the first iteration will complete entirely printing 0 through 499,999 for Child, then it will do with same printing 0 through 499,999 for the Parent process.

```
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S 315789 18788 18404  0 80   0 - 32492 sigsus pts/1    00:00:00 tcsh
0 R 315789 24269 18788  0 80   0 -  1053 -      pts/1    00:00:00 part4
0 R 315789 24270 18788  0 80   0 - 38301 -      pts/1    00:00:00 ps
1 S 315789 24271 24269  0 80   0 -  1053 -      pts/1    00:00:00 part4
```

```
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S 315789 18788 18404  0 80   0 - 32492 sigsus pts/1    00:00:00 tcsh
0 S 315789 24067 18788  0 80   0 -  1052 do_wai pts/1    00:00:00 part5
0 R 315789 24068 18788  0 80   0 - 38301 -      pts/1    00:00:00 ps
1 S 315789 24069 24067  0 80   0 -  1053 -      pts/1    00:00:00 part5
```

3.6

1.) The program appears to have an infinite loop. Why does it stop
The program stops because the line of code which contains kill(child, SIGTERM); kills the child process.

2.) From the definitions of sleep and usleep, what do you expect the child's count to be just before it ends?
I would expect the count to be 10 seconds / .01 seconds to be 1,000.

3.) Why doesn't the child reach this count before it terminates?
Because the 2 processes are running in parallel and the the parent process executes faster, the child process is delayed by the i++ and print statements during each iteration which add to the program's execution time while the child process is killed immediately after 10 seconds is counted.

3.7

1.) Read the man page for execl and relatives (man 3 exec). Under what conditions is the printf statement executed? Why isn't it always executed? (consider what would happen if you changed the program to execute something other than /bin/ls.)
The execl command creates replaces the current process image with a new process image, where the initial argument is the name of the file that is to be executed. With that being said, the printf is executed when the initial argument in the execl command does not contain a path to an executable file. If the execl command is written properly the printf command won't execute because the executable file's process takes the place of the current process.

3.8

1.) What is the range of values returned from the child process?
The range of values returned from the child process is determined by the "rand()" function, thus the values range from 0 to 2^15

2.) What is the range of values sent by child process and captured by the parent process?
If rand() is less than RAND_MAX/4, the child is killed
There are 2 possible options dependent on the value of rand()

1. When rand() >= RAND_MAX/4: WEXITSTATUS(status) - related to WEXITED macro which returns a nonzero value if the child process terminated normally with exit or _exit. This occurs when rand() >= RAND_MAX/4. This returns the low-order 8 bits of the exit status value from the child process. Thus the range is 0 - 255
2. When rand() < RAND_MAX/4: WTERMSIG(status) - related to WIFSIGNALED macro which returns nonzero if child process terminated because it received a signal that was

not handled. This macro returns the signal number of the signal that terminated the child process. Because SIGKILL corresponds to signal number 15, this number is returned.

3.) When do you think the return value would be useful? Hint: look at the commands true and False.

It is useful in determining in what way a child process was terminated. E.g. kill -l will display numbers corresponding to different signals [terminal output included below)

```
 1) SIGHUP     2) SIGINT     3) SIGQUIT     4) SIGILL     5) SIGTRAP
 6) SIGABRT     7) SIGBUS     8) SIGFPE     9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT     17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG     24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM     27) SIGPROF     28) SIGWINCH     29) SIGIO     30) SIGPWR
31) SIGSYS     34) SIGRTMIN     35) SIGRTMIN+1     36) SIGRTMIN+2     37) SIGRTMIN+3
38) SIGRTMIN+4     39) SIGRTMIN+5     40) SIGRTMIN+6     41) SIGRTMIN+7     42)
SIGRTMIN+8
43) SIGRTMIN+9     44) SIGRTMIN+10     45) SIGRTMIN+11     46) SIGRTMIN+12     47)
SIGRTMIN+13
48) SIGRTMIN+14     49) SIGRTMIN+15     50) SIGRTMAX-14     51) SIGRTMAX-13     52)
SIGRTMAX-12
53) SIGRTMAX-11     54) SIGRTMAX-10     55) SIGRTMAX-9     56) SIGRTMAX-8     57)
SIGRTMAX-7
58) SIGRTMAX-6     59) SIGRTMAX-5     60) SIGRTMAX-4     61) SIGRTMAX-3     62)
SIGRTMAX-2
63) SIGRTMAX-1     64) SIGRTMAX
```

3.9

1.) What is a zombie process?

Basically it is a process that has been completed and in the terminated state that still appears in the process table so that the parent may read the child's exit status. Processes that stay zombies for a long period of time are generally an error and can cause resource leaks.

2.) Write a program that creates a zombie process and describe how to verify that the zombie process has been created.

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

```
int main (){
    pid_t child;
    child = fork ();
    if (child > 0) {
            sleep (60);
    }
    else {
            exit (0);
    }
    return 0;
}
```

We can verify the zombie process by telling the program to sleep and using ps -l. Hence our <defunct> is our zombie process.

```
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  3797  3717  0  80   0 -  7583 wait   pts/0    00:00:00 bash
0 S  1000  9565  3797  0  80   0 -  1094 hrtime pts/000:00:00 zombie
1 Z  1000  9566  9565  0  80   0 -     0 -      pts/0    00:00:00 zomb <defunct>
4 R  1000  9567  3797  0  80   0 -  9004 -      pts/0    00:00:00 ps
```