HW 8

Joshua Thompson

March 17, 2018

Questions

1. (3 points) What is a zombie and how are they created? AND, why are zombie process a bad thing? (process zombies not human zombies)

   when a child process is not waited on by the parent. It still shows on the process table. They are bad because enough of them can clog up the process table.

2. (3 points) What is an orphan process? How are they created and who "adopts" all orphans?

   An orphan process is a process in which the parent dies before the child does. The init process inherits the child and waits until the child process is complete.

3. (5 points) How are process groups and jobs related in the shell?

   Because some jobs require more than one process to complete, the multiple processes are grouped together into a process group.

4. (4 points) How long with the following shell command run for and why?

   ```
   sleep 10 | sleep 20 | sleep 100 | sleep 30 | sleep 1
   ```

   The command will run for 100 seconds then stop. This is because all the commands are run in parallel. Since they don't rely on any information from the previous command, they all just run at the same time.

5. (5 points) Explain the difference between sequential and parallel execution of a command line?

   The difference is that in sequential execution, the process will always wait for the previous one to finish before it begins executing. In the above question that would mean that the time to complete the command would be all the sleep commands added up together. For parallel execution, all the processes are run at once, so instead of waiting for the last command to finish, it begins completing the command while the other is also running.

6. (8 points) For each of the system calls associated with process groupings, provide a brief explanation of each.

   (a) `setpgrp()`

Sets the process group to the same as itself. (pid)

(b) `setpgid()`

Sets the identified process in the first argument to itself. If first arg is 0 then sets the process group to pgid.

(c) `getpgrp()`

Gets the process group id of calling process

(d) `getpgid()`

Gets the process group id for process identified in first arg

7. (10 points) For each of the system calls with arguments, briefly describe the resulting action with respect to the calling process or target process.

(a) `getpgid(0)`

Would get the process group id for stdin

(b) `setpgid(0,0)`

sets the current processes pgid to its own pid

(c) `setpgid(0,pgid)`

sets the current processes pgid to the one specified in the second argument

(d) `setpgid(pid, 0)`

Sets the pgid of the process specified in the first argument to the caller's pgid

8. (10 points) Consider the following code snippet, what is the output and why? (*Hint: why not run it?*)

```
int main(){
  pid_t cpid;

  cpid = fork();
  if(cpid == 0){

    setpgrp();
    if( getpid() == getpgrp()){
      printf("C: SAME PGID\n");
    }
    exit(0);

  }else if(cpid > 0){

    if(getpgid(cpid) == cpid){
      printf("P: SAME PGID\n");
```

```
    }else{
      printf("P: NOT SAME PGID\n");
    }

    wait();
    exit(0);
  }

  exit(1);
}
```

output: P: NOT SAME PGID C: SAME PGID

9. (12 point) Consider the following code snippet. If we were to run this program in a terminal, will it be properly terminated by Ctrl-c? If so, why? If not, why not?

```
int main(){
  pid_t cpid;

  cpid = fork();
  if( cpid == 0 ){
    setpgrp();
    while(1);
  }else if( cpid > 0 ){
    wait(NULL);
    exit(0);
  }
  exit(1); //fork failed
}
```

ctrl+c does not terminate the process because the process group id of the child is changed from the parent group id.

10. (12 point) Consider the following code snippet with the open file `fight.txt` containing the text " Go Navy! Beat Army!" (yes, there are spaces in there). What is the output of this program, and why?

```
int main(){
  pid_t cpid;

  int fd = open("fight.txt",O_RDONLY);
  char buf[1024];

  cpid = fork();
  if( cpid == 0 ){
    read(fd, buf, 10);
    exit(0);
  }else if( cpid > 0 ){
    wait(NULL); /* wait for child*/

    read(fd,buf, 10);
```

```
    write(1, buf, 10);
    exit(0);
  }
  exit(1); //fork failed
}
```

Output is "Beat Army!". Child reads 10 bytes from the file, then parent reads next 10 and writes to stdout

11. (4 points) What does it mean to "widow" a pipe?

closes the end of the pipe

12. (12 points) Consider the following code snippet with the open file `fight.txt` containing the text " Go Navy! Beat Army!" (yes, there are spaces in there). What is the output of this program, and why?

```
int main(){
  int fd_in = open("fight.txt",O_RDONLY);

  int fd_out = open("output.txt",O_WRONLY | O_TRUNC | O_CREAT,0755);
  char buf[1024];

  close(0);
  dup2(fd_in,0);

  close(1);
  dup2(fd_out,1);

  while(scanf("%s",buf) != EOF){
    printf("%s\n",buf);
  }

  return 0;
}
```

When the close(1) happens, an eof is hit and the while loop is never begun.

13. (12 points) What is the missing code in the program below such that the child's write to `stdout` will be ready by the parent through its `stdin`?

```
int main(){
  pid_t cpid;
  int pfd[2], n;
  char gonavy[] = "Go Navy!";
  char buffer[1024];

  pipe(pfd);

  cpid = fork();
  if( cpid == 0 ){
```

```
    /* What goes here? */

    write(1, gonavy,strlen(gonavy));
  }else if( cpid > 0 ){

    /* What goes here? */


    n = read(0, buffer, 1024);
    write(1,buffer,n);
  }

  exit(1);
}
```

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

  int main(){
    pid_t cpid;
    int pfd[2], n;
    char gonavy[] = "Go Navy!";
    char buffer[1024];

    pipe(pfd);

    cpid = fork();
    if( cpid == 0 )
    {
      close(1);
      dup2(pfd[1], 1);
      close(pfd[0]);

      write(1, gonavy,strlen(gonavy));
    }
    else if( cpid > 0 )
    {
      close(0);
      dup2(pfd[0], 0);
      close(pfd[1]);


      n = read(0, buffer, 1024);
      write(1,buffer,n);
    }

    exit(1);
  }
```