HW 8

Mike Hanling

19 MAR 18

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)

   A zombie is a process that has terminated but has not been wait() 'ed on by a parent, therefore its information still exists and stays that way "forever". They are bad because they eat up brains....I mean memory.

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

   An orphan is a process that is still running, but its parent has terminated. init adopts all orphan processes.

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

   A job is completed through however many processes it needs, all labeled undder the same process group id.

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

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

   This will run for 100 seconds because all of the sleep processes will run in parallel, so the one that takes the longest is the overall time: sleep 100 runs for 100 seconds.

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

   Sequential execution is when processes are doen one after another (waiting for the end of one before the beginning of another). Parallel execution is when multiple processes are executed at the same time (all starting at once).

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

   (a) `setpgrp()`

This will set the process group of the calling process to itself.

(b) `setpgid()`

This will set the process group of the first arguemnt to the second argument. Zero for the first argument means that the calling process should be used. Zero for the second argument means to set the process group to the pid of the first argument.

(c) `getpgrp()`

This will get the process group of the calling process.

(d) `getpgid()`

This will get the process group of the argument.

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)`

This will retrieve the process group of the calling process.

(b) `setpgid(0,0)`

This will set the process group of the calling process to the pid of the calling process.

(c) `setpgid(0,pgid)`

This will set the process group of the calling process to pgid.

(d) `setpgid(pid, 0)`

This will set the process group of pid to the pid of the calling process.

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);
}
```

P: NOT SAME PGID
C: SAME PGID
This is because the parent is executing before the child (at least in this run). Therefore, the parent will see the cpid not matching its group, but in the child it sets its group and then checks it.

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
}
```

Yes, this will terminate properly because all of the processes are still in the foreground process group.

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
}
```

The output is ”Beat Army!”. This is because The child reads the first 10 bytes (” Go Navy!”) and the parent is waiting. Once the child terminates, the parent the rest 10 bytes and prints it out.

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

The read end is closed.

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;
}
```

The output file has the text line separated. This is because stdin and stdout were duplicated into the new file descriptors for this problem.

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);
}
```

First slot: dup2(0, 1);
Second slot: dup2(1, 0);