Processes (Practice)

ICS332
Operating Systems

Henri Casanova (henric@hawaii.edu)

(q1) What does this print?

```
int x = 12;
int pid = fork();
x += (pid == 0);
printf("%d\n",x);
```

- Note that the order in which things are printed will not be deterministic

(q1) Answer

```
int x = 12;
int pid = fork();
x += (pid == 0);
printf("%d\n",x);
```

12
13

or

13
12

(q2) What does this print?

```
int x = 12;
int pid = fork();
x += (pid == 0);
printf("%d\n",x);
if (fork()) {
  x++;
}
printf("%d\n",x);
```

- Note that the order in which things are printed will not be deterministic

(q2) Answer

```
int x = 12;
int pid = fork();
x += (pid == 0);
printf("%d\n",x);
if (fork()) {
  x++;
}
printf("%d\n",x);
```

```
12
13
12
13
13
14
```

(q2) Count Processes

```
int i = 3;
int pid;

while(i > 0) {
  pid = fork();
  if (pid > 0) {
    exit(0);
  } else {
    i--;
  }
}
```

- How many new processes does this program create (not counting the initial process)?

(q2) Answer

```
int i = 3;
int pid;

while(i > 0) {
  pid = fork();
  if (pid > 0) {
    exit(0);
  } else {
    i--;
  }
}
```

- 3 calls to fork(), so the answer is 3
  - The parent process exits after calling fork(), and so there is no proliferation of processes

(q3) Count Processes

```
int i = 3;
int pid;

while(i > 0) {
  pid = fork();
  if (pid > 0) {
    i--;
    exit(0);
  } else {
    // nothing
  }
}
```

- How many new processes does this program create (not counting the initial process)?

(q3) Answer

```
int i = 3;
int pid;

while(i > 0) {
  pid = fork();
  if (pid > 0) {
    i--;
    exit(0);
  } else {
    // nothing
  }
}
```

- Infinite
  - The loop index is never decremented in the child process each time, and only the child process loops back
- This is NOT a “fork bomb” because at any given time there is a single process running
- If you run this it will be hard to kill manually
  - By the time you look at the PID, it’s already gone and there is a new PID to kill
- The **`killall`** command saves the day!

(q4) Count Processes

```
for (int i=0;i<=3;i++) {
  fork();
}
```

- How many new processes does this loop create (not counting the initial process)?

(q4) Answer

```
for (int i=0;i<=3;i++){
  fork();
}
```

- Iteration i=0: the main process creates a child process, and now **2** processes loop back and do iteration i=1
- Iteration i=1: 2 processes each create one child process, so now **4** processes loop back and do iteration i=2
- Iteration i=2: 4 processes each create one child process, so now **8** processes loop back to do iteration i=3
- Iteration i=3: 8 processes each create one child process, for a total of **16** running processes

- Overall, this program creates 15 processes (not counting the main process)

(q5) Problem?

```
int main() {
  for (;;) {
    pid = fork();
    if (pid == 0) {
      exit(0);
    } else {
      sleep(1);
    }
  }
}
```

- What would be an eventual problem if you were to execute this program on your computer? Explain.

(q5) Answer

```
int main() {
  for (;;) {
    pid_t pid = fork();
    if (pid == 0) {
      exit(0);
    } else {
      sleep(1);
    }
  }
}
```

At each iteration the child exists, but the parent never acknowledges its death

All created children linger as zombies, eventually filling up the process table

We can run this (because we have a **`sleep(1)`** in there!) and observe….

(q6) No Zombies

```
for (;;) {
  pid_t pid = fork();
  if (pid == 0) {
     fork();
     exit(0);
  } else {
     // nothing
  }
}
```

- This program creates an increasing number of zombies
- How can we add one line of code to make sure no zombies are generated?

(q6) Answer

```
for (;;) {
  pid_t pid = fork();
  if (pid == 0) {
     fork();
     exit(0);
  } else {
     wait(NULL);
  }
}
```

- This is the classical “bad grandpa” example from the lecture notes
- At each iteration, the parent process creates a child and a grandchild, both of which exit right away
- The grandchild will be immediately adopted because it becomes an orphan
- But the child lingers
- The fix: have the parent acknowledge the child’s death

(q7) Orphans?

```
for (int i=0; i < 1000; i++) {
  pid_t pid = fork();
  if (pid == 0) {
     if (!fork()) {
       sleep(10);
     }
     exit(0);
  } else {
    wait(NULL);
  }
  sleep(100);
}
```

- After this program terminates, how many orphans where adopted?
- And how many of these orphans are still running?

(q7) Answer

```
for (int i=0; i < 1000; i++) {
  pid_t pid = fork();
  if (pid == 0) {
     if (!fork()) {
       sleep(10);
     }
     exit(0);
  } else {
    wait(NULL);
  }
  sleep(100);
}
```

- 1000 orphans have been adopted
  - All the grand children
- 0 are running
  - They all exit after the sleep 10

(q8) Quick Orphan

- Write the shortest possible program that creates an orphan that survives it forever…

(q8) Answer

- Write the shortest possible program that creates an orphan that survives it forever…

```
int main() {
  if (!fork()) {
     for (;;);
  }
}
```

(q9) What does this print?

```
int x = 6;
int main() {
  int pid = fork();
  if (pid != 0) {
    x += 4;
    printf("%d\n",x);
    waitpid(pid, NULL, 0);
    printf("%d\n",x);
  } else {
    sleep(2);
    fork();
    printf("%d\n", x);
  }
}
```

(q9) What does this print?

```
int x = 6;
int main() {
  int pid = fork();
  if (pid != 0) {
    x += 4;
    printf("%d\n",x);
    waitpid(pid, NULL, 0);
    printf("%d\n",x);
  } else {
    sleep(2);
    fork();
    printf("%d\n", x);
  }
}
```

```
10
6
6
10
```

or

```
10
6
10
6
```

- The **`waitpid()`** call waits only for the child, not the grandchild, so the second option above is also possible

(q10) What does this print?

```
int main(int argc, char **argv) {
  FILE *output = fopen("sort","w");
  pid_t pid = fork();
  if (pid) {
    int exit_code;
    waitpid(pid, &exit_code, 0);
    fclose(output);
  } else {
    dup(fileno(output));
    execl("/bin/ls","/bin/ls",
          (char *)NULL);
  }
  exit(0);
}
```

- An ICS 332 student decided to write a program that calls **`ls`** and pipes its output to the **`sort`** command, so that the program should print the sorted files in the current directory
- This program has two main mistakes
- Question 1: What does this program actually do and print?
- Question 2: How to fix it?

(q10) Answer

```
int main(int argc, char **argv) {
  FILE *output = fopen("sort","w");
  pid_t pid = fork();
  if (pid) {
    int exit_code;
    waitpid(pid, &exit_code, 0);
    fclose(output);
  } else {
    dup(fileno(output));
    execl("/bin/ls","/bin/ls",
          (char *)NULL);
  }
  exit(0);
}
```

- Question 1:
  - The program will just print the output of the **`ls`** command to the terminal
  - And it will create an empty file called **`sort`**

(q10) Answer

```
int main(int argc, char **argv) {
  FILE *output = popen("sort","w");
  pid_t pid = fork();
  if (pid) {
    int exit_code;
    waitpid(pid, &exit_code, 0);
    pclose(output);
  } else {
    close(1);
    dup(fileno(output));
    execl("/bin/ls","/bin/ls",
          (char *)NULL);
  }
  exit(0);
}
```

- Question 2:
  - Use **popen/pclose** as opposed to **fopen/fclose**
  - Note that **fopen/fclose** would be useful if we wanted the output to go to a file!
- Don't forget to close file descriptor 1 (**stdout**) so that the output of the **ls** command is redirected