CS2106 Operating Systems

Semester 1 2021/2022

Week 4 (30 August - 03 September 2021)

Tutorial 2: **Process Abstraction in Unix**

1. (Process Creation) [Taken from AY18/19 S1 Midterms]

	C code:
00	`int main( ) {`
01	`    //This is process P`
02	`    if ( fork() == 0 ){`
03	`        //This is process Q`
04	`        if ( fork() == 0 ) {`
05	`            //This is process R`
06	`            ......`
07	`            return 0;`
08	`        }`
09	`        <Point α>`
10	`    }`
11	`    <Point β>`
12	
13	`    return 0;`
14	`}`

Each of the following cases insert zero or more lines at Point α and β. **Evaluate whether each described behaviour is correct or incorrect.** (Note that `wait()` does not block when a process has no children.)

Point α	**Point β**	**Behaviour**
nothing	`wait(NULL);`	Process Q ***always*** terminates before P. Process R can terminate at any time w.r.t. P and Q.
`wait(NULL);`	*nothing*	Process Q ***always*** terminates before P. Process R can terminate at any time w.r.t. P and Q.
`execl(valid executable....);`	`wait(NULL);`	Process Q ***always*** terminates before P. Process R can terminate at any time w.r.t. P and Q.
`wait(NULL);`	`wait(NULL);`	Process P **never terminates**.

2. (Behavior of **`fork()`** system call) The C program below attempts to highlight the behavior of the **`fork()`** system call:

	C code:
01	`int dataX = 100;`
02	`int main( )`
03	`{`
04	`    pid_t childPID;`
05	
06	`    int dataY = 200;`
07	`    int* dataZptr = (int*) malloc(sizeof(int));`
08	
09	`    *dataZptr = 300;`
10	
11	`    //First Phase`
12	`    printf("PID[%d] | X = %d | Y = %d | Z = %d |\n",`
13	`            getpid(),  dataX, dataY, *dataZptr);`
14	
15	`    //Second Phase`
16	`    childPID = fork();`
17	`    printf("*PID[%d] | X = %d | Y = %d | Z = %d |\n",`
18	`            getpid(),  dataX, dataY, *dataZptr);`
19	
20	`    dataX += 1;`
21	`    dataY += 2;`
22	`    (*dataZptr) += 3;`
23	`    printf("#PID[%d] | X = %d | Y = %d | Z = %d |\n",`
24	`            getpid(),  dataX, dataY, *dataZptr);`
25	
26	`    //Insertion Point (for parts (g), (h))`
27	
28	`    //Third Phase`
29	`    childPID = fork();`
30	`    printf("**PID[%d] | X = %d | Y = %d | Z = %d |\n",`
31	`            getpid(),  dataX, dataY, *dataZptr);`
32	
33	`    dataX += 1;`
34	`    dataY += 2;`
35	`    (*dataZptr) += 3;`
36	`    printf("##PID[%d] | X = %d | Y = %d | Z = %d |\n",`
37	`            getpid(),  dataX, dataY, *dataZptr);`
38	
39	`    return 0;`
40	`}`
41	
42	

Please run the given program "**`ForkTest.c`**" on your system before answering the questions below.

a. What is the difference in how the 3 variables: **`dataX, dataY, dataZptr,`** and the memory location pointed to by **`dataZptr,`** are stored?

b. Explain the **values** that are printed by the program.

c. Focusing on the messages generated by second phase (they are prefixed with either "**`*`**" and "**`#`**"), what can you say about the behavior of the **`fork()`** system call?

d. Using the messages seen on your system, draw a **process tree** to represent the processes generated. Use the process tree to explain the values printed by the child processes.

e. Do you think it is possible to get different orderings between the output messages? Why?

f. Can you point out which pair(s) of messages can never swap places? i.e., their relative order is always the same?

g. If we insert the following code at the insertion point:

Sleep Code

```
if (childPID == 0){
    sleep(5);  //sleep for 5 seconds
}
```

How does this change the ordering of the output messages? State your assumptions, if any.

h. Instead of the code in (g), we insert the following code at the insertion point:

Wait Code

```
if (childPID != 0){
    wait(NULL);  //NULL means we don't care
                 //   about the return result
}
```

How does this change the ordering of the output messages? State your assumption, if any.

3. (Parallel computation) Even with this crude synchronization mechanism, we can solve programming problems in new (and exciting) ways. We will attempt to utilize multiple processes to work on a problem simultaneously in this question.

   You are given two C source code files: "**`Parallel.c`**" and "**`PrimeFactors.c`**". "**`PrimeFactors.c`**" is a simple prime factorization program. "**`Parallel.c`**" uses the "**`fork`**()" and "**`execl`**()" combination to spawn off a new process to run "**`PrimeFactors.c`**"

   Let's setup the programs as follows:
   1. Compile "**`PrimeFactors.c`**" to get an executable with the name "**`PF`**":
      `gcc PrimeFactors.c –o PF`

   2. Compiles "**`Parallel.c`**": **`gcc Parallel.c`**

   Run the **`a.out`** generated from step (2). Below is a sample session:

```
$> a.out
1024
1024 has 10 prime factors //note: not unique prime factors
```

   If you try large prime numbers, e.g. 111113111, the program may take a while.

   Modify only `Parallel.c` such that we can now initiate prime factorization on [1-9] user inputs simultaneously. More importantly, we want to report results as soon as they are ready regardless of the user input order.

   Sample session below:

```
$> a.out < test2.in
9 has 2 prime factors          //Results
118689518 has 3 prime factors
44721359 has 1 prime factors
99999989 has 1 prime factors
111113111 has 1 prime factors
```

   Note the order of the result may differ on your system. Most of time, they should follow roughly the computation time needed (composite number < prime number and small number < large number). Two simple test cases are given "**`test1.in`**" and "**`test2.in`**" to aid your testing.

   Most of what you need is already demonstrated in the original **`Parallel.c`** (so that this is more of a mechanism question rather than a coding question). You only need "**`fork`**()", "**`execl`**()" and "**`wait`**()" for your solution.

   After you have solved the problem, find a way to change your **`wait()`** to **`waitpid()`**, **what do you think is the effect of this change?**

Additional Questions (For exploration only, not for tutorial discussion)

1. (Process Creation) Consider the following sequence of instructions in a C program:

C code:

```
int x = 10;
int y = 123;

y = fork();
if (y == 0)
    x--;

y = fork();
if (y == 0)
    x--;

printf("[PID %d]: x=%d, y=%d\n",getpid(),x ,y);
```

You can assume that the first process has process number **100** (and so **getpid()** returns the value **100** for this process), and that the processes created (in order) are **101**,**102** and so on.

Give:

a) A possible final set of printed messages.
b) An impossible final set of printed messages.