CS2106 Operating Systems
25/26 Semester 1
Tutorial 2
Process Abstraction in Unix

1. [Process Creation Recap - Taken from AY18/19 S1 Midterms] Each of the following cases insert zero or more lines at the Point α and β. Evaluate whether the described behaviour is correct or incorrect. (Note that **`wait()`** does not block when a process has no children.)

C code:

```
int main( ) {
    //This is process P
    if ( fork() == 0 ) {
        //This is process Q
        if ( fork() == 0 ) {
            //This is process R
            ......
            return 0;
        }
        <Point α>
    }
    <Point β>
    return 0;
}
```

Point α	Point β	Behavior
Nothing	**`wait(NULL);`**	Process Q ***always*** terminate before P. Process R can terminate at any time w.r.t. P and Q.
`wait(NULL);`	*nothing*	Process Q ***always*** terminate before P. Process R can terminate at any time w.r.t. P and Q.
`execl(valid executable....);`	**`wait(NULL);`**	Process Q ***always*** terminate 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`**] The C program below attempts to highlight the behavior of the **`fork()`** system call:

C code:

```
int dataX = 100;

int main( ) {
    pid_t childPID;

    int dataY = 200;
    int* dataZptr = (int*) malloc(sizeof(int));

    *dataZptr = 300;

    //First Phase
    printf("PID[%d] | X = %d | Y = %d | Z = %d |\n",
getpid(), dataX, dataY, *dataZptr);


    //Second Phase
    childPID = fork();
    printf("*PID[%d] | X = %d | Y = %d | Z = %d |\n",
getpid(), dataX, dataY, *dataZptr);

    dataX += 1;
    dataY += 2;
    (*dataZptr) += 3;
    printf("#PID[%d] | X = %d | Y = %d | Z = %d |\n",
getpid(), dataX, dataY, *dataZptr);

    //Insertion Point

    //Third Phase
    childPID = fork();
    printf("**PID[%d] | X = %d | Y = %d | Z = %d |\n",
getpid(), dataX, dataY, *dataZptr);

    dataX += 1;
    dataY += 2;
    (*dataZptr) += 3;
    printf("##PID[%d] | X = %d | Y = %d | Z = %d |\n",
getpid(), dataX, dataY, *dataZptr);

    return 0;
}
```

The code above can also be found in the given program "**`ForkTest.c`**" . Please run it on your system before answering the questions below.

a. What is the difference between the 3 variables: **`dataX`**, **`dataY`** and the memory location pointed by **`dataZptr`**?
b. 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?
c. 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.
d. Do you think it is possible to get different ordering between the output messages, why?
e. Can you point how which pair(s) of messages can never swap places? i.e. their relative order is always the same?
f. 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 assumption, if any.

g. Instead of the code in (f), 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 the 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 "**`Parallel.c`**" and "**`PrimeFactors.c`**". The "**`PrimeFactors.c`**" is a simple prime factorization program. "**`Parallel.c`**" use the "**`fork`**()" and "**`execl`**()" combination to spawn off a new process to run the prime factorization.

Let's setup the programs as follows:

1. Compiles "**`PrimeFactors.c`**" to get a executable with 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.

<u>**Modify only `Parallel.c`**</u> such that we can now initiate prime factorization on [1-9] user inputs <u>simultaneously</u>. More importantly, we want to report result as soon as they are ready regardless of the user input order. Sample session below:

```
$ ./a.out
5                    // 5 user inputs
44721359
99999989
9
111113111
118689518
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. If you are using a rather powerful machine (e.g. the SoC Compute Cluster), you can use the **`test3.in`** to provide a bit more grind.

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?**

For your own exploration:

4. (Process Creation) Given the following full program, give and explain the execution output. The source code **`FF.c`** is also given for your own test.

C code:

```
int factorial(int n) {
    if (n == 0) {
        fork();        // NOTE the change
        return 1;
    }
    return factorial(n-1) * n;
}

int main() {
    printf("fac(2) = %d\n", factorial(2));

    return 0;
}
```

5. (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.

6. (Parent-Child Synchronization) Consider the following sequence of instructions in a C program:

C code:

```
int i;
pid_t cPidArray[3];  // an array of 3 child pids

for (i = 0; i < 3; i++) {
    cPidArray[i] = fork();
    if (cPidArray[i] == 0) {
        // do something
        printf("Child [%d] is done!\n", getpid());
        return 0; // exit
    }
}

// Code insert point here
printf("Parent [%d] is done!\n", getpid());
```

Similar to Q1, let's assume the first process has pid of 100, and that the processes created (in order) are `101`,`102` and so on.

Suppose we insert the following code fragments at the end of the program above, describe the effects on the synchronization / timing property of the program. If it helps, you can give a sample output to aid your explanation.

a) wait()

```
for (i = 0; i < 3; i++) {
    wait(NULL);
    printf("Parent: one child exited\n");
}
```

b) waitpid()

```
for (i = 0; i < 3; i++) {
    waitpid(cPidArray[i], NULL, 0);
    printf("Parent: one child exited\n");
}
```