CS2106 Operating Systems
Semester 2 2020/2021

Week of 1st February 2021
Tutorial 2 Suggested Solutions
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:
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	`}`

Point α	**Point β**	**Behaviour**
Nothing	`wait(NULL);`	Process Q ***always*** terminate before P. Process R can terminate at any time w.r.t. P and Q. [False: Q waits for R]
`wait(NULL);`	*nothing*	Process Q ***always*** terminate before P. Process R can terminate at any time w.r.t. P and Q. [False: Q waits for R and P don't wait]
`execl(valid executable....);`	`wait(NULL);`	Process Q ***always*** terminate before P. Process R can terminate at any time w.r.t. P and Q. [True: P wait for Q even though Q is now a "new" executable]
`wait(NULL);`	`wait(NULL);`	Process P **never terminates**. [False: Although Q has an additioinal

		wait, the wait will return immediately as there is no child.]

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.

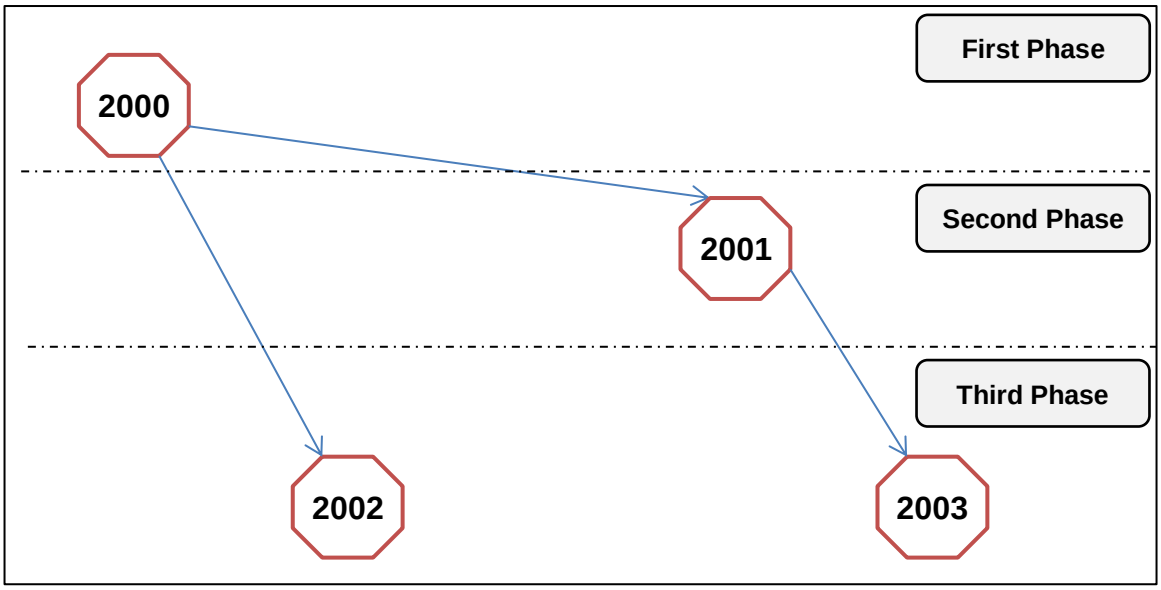
ANS:

a. They occupy different memory regions: **dataX** is in **data segment, dataY is in stack segment, the memory location pointed by dataZptr** is in heap segment. It is important to note that **dataZptr** (the pointer itself) is just a local variable in function, i.e. in stack segment. However, the memory location it points to is dynamically allocated (via **malloc()**), i.e. in heap segment.

b. Looking at the value at the "*" messages, we can see that all 3 data items are duplicated. Both the parent and child process has the same value after fork().

The "#" messages show the data items after change, we can see that the processes have independent memory space, i.e. updates do not impact each other's memory space.

c. For simplicity, we assume the process ids are 2000, 2001, 2002, etc in our drawing. You should focus on the **relationship** between processes instead of just the pids.



Note that all four processes are alive in the third phase. Also, note that we may not be able to deduce the relationship between (2001→2003 and 2000→2002) from the printed messages as they can be scheduled independently after spawning. If you want to know the exact pairing, using `getppid()` (get parent's PID) will give a definite answer.

d. Yes. Once the processes are created, they can be independent be chosen by the OS to run. Depending on the existence of other processes at that time, it is possible that OS choose differently between runs of the program.

e. The discussion for this question is based on the process tree from (c). Some possible answers:

- "*" and "#" messages from the same process can never change place as sequential ordering is still preserved in the same process.
- Likewise, messages from the same process will always follow the phases, i.e. "*", "#" before "**" and "##".
- Message from the first phase (only one) must precede all other messages. This is obviously correct as there is only one process executing at that time.

Some **wrong answers** worth noting:

- The messages from child process always precede (or always come after) the direct parent's message. **[Wrong because** Parent and child are scheduled independently, i.e. there is no fixed execution order between them**]**.
- The messages from the same phase always precede messages from the next phase. [**Wrong, as a counter-example:** The parent process can executes to the end, i.e. printing messages from all 3 phases before any of the forked processes has a chance to execute.

f. The inserted code "pause" the first child process (i.e. 2001) for 5 seconds. So, if we assume process 2002 takes less than 5 seconds to create and run, then it is likely that both process 2000 and 2002 will finished execution before 2001 and 2003.

g. The inserted code "pause" the first process (i.e. 2000) **until process 2001 is finished**. So, all messages from 2001 must be printed before process 2000 and subsequently 2003 can be printed. The difference to (f) is that this is a fixed behavior **regardless** of how long process 2001 takes to finish its execution.

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.

   Modify only `Parallel.c` such that we can now initiate prime factorization on [1-9] user inputs simultaneously. 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?**

ANS

See Parallel_Solved.c. Question can be discussed in terms of mechanisms, instead of pure coding. Points out that we are "paying" process spawning overhead to "earn" (real) parallel execution. If there is only a single processor, or the overhead > earning from parallel execution, the solution will NOT show any improvement.

The change of wait() to waitpid() forces the main process to wait for the child process in certain order, e.g. the creation order of the child processes. Using the same execution example given in the question, the messages we see is now:

```
$> a.out
5                          //5 user inputs
44721359
99999989
9
111113111
118689518
44721359 has 1 prime factors    //Results
99999989 has 1 prime factors
9 has 2 prime factors
111113111 has 1 prime factors
118689518 has 3 prime factors
```