Stamatios Morellas
COM S 352 - Homework 2
Due 1/31/20 @ 11:59pm

1. (10 points) Textbook Exercise 3.8: **Describe the actions taken by a kernel to context-switch between processes.**

In order for a kernel to context-switch between different processes, control is first transferred to the clock interrupt handler, which then saves and records the stack pointer for the current process, the registers, and other machine states. Once this is complete, the OS must determine the next task to execute, which is then restored to the registers.

2. (10 points) Textbook Exercise 3.11. **You should draw a process tree to show the relationship between the processes: Including the initial parent process, how many processes are created by the program shown in Figure 3.32? Draw a process tree to show the relationship between the processes.**

The result of the following code:

```
for(i = 0; i < 4; i++)
    fork();
```

Is equivalent to:

```
fork();
fork();
fork();
fork();
```

Calling `fork()` four times will result in 15 processes being created by the program. When run on my personal machine using the command `pstree`, I received this as my output which represents the process tree:

```
› pstree 13257
-+= 13257 stamos ./p2
 |-+- 13258 stamos ./p2
 | |-+- 13261 stamos ./p2
 | | |-+- 13267 stamos ./p2
 | | | \--- 13272 stamos ./p2
```

```
 | | \--- 13271 stamos ./p2
 | |-+- 13264 stamos ./p2
 | | \--- 13270 stamos ./p2
 | \--- 13268 stamos ./p2
 |-+- 13259 stamos ./p2
 | |-+- 13263 stamos ./p2
 | | \--- 13269 stamos ./p2
 | \--- 13266 stamos ./p2
 |-+- 13260 stamos ./p2
 | \--- 13265 stamos ./p2
 \--- 13262 stamos ./p2
```

3. (10 points) Textbook Exercise 3.12. **Explain the circumstances when the line of code marked printf("LINE J") in Figure 3.33 is reached.**

```
1 #include <sys/types.h>
2 #include <stdio.h>
3 #include <unistd.h>
4
5 int main() {
6   pid_t pid;
7
8   /* fork a child process */
9   pid = fork();
10
11  if (pid < 0) { /* error occurred */
12    fprintf(stderr, "Fork Failed");
13    return 1;
14  }
15  else if (pid == 0) { /* child process */
16    execlp("/bin/ls", "ls", NULL);
17    printf("LINE J");
18  }
19  else { /* parent process */
20     /* parent will wait for the child to complete */
21     wait(NULL);
22     printf("Child Complete");
23  }
24
25   return 0;
26 }
```

The code at line 17 will execute when the child process is done executing and right before the parent process is ready to begin.

4. (10 points) Textbook Exercise 3.13. **Using the program in Figure 3.34, identify the values of pid at lines A, B, C, and D. (Assume that the actual pids of the parent and child are 2600 and 2603, respectively.)**

	Value of pid	Value of pid1
Line A	0	
Line B		2603
Line C	2603	
Line D		2600

5. (10 points) Textbook Exercise 3.16. **Using the program shown in Figure 3.35, explain what the output will be at lines X and Y.**

The output at line X is shown below:

```
CHILD: 0 CHILD: -1 CHILD: -4 CHILD: -9 CHILD: -16
```

The output of this program at line Y is shown below:

```
PARENT: 0 PARENT: 1 PARENT: 2 PARENT: 3 PARENT: 4
```

6. (10 points) In a multitasking system, the context-switch time is 1ms and the time slice is 10ms. Will the CPU efficiency increase or decrease when we increase the time slice to 15ms? Explain why.

Since increasing the time slice will result in longer CPU usage for processes, one could say that efficiency would decrease since the tasks take-up more of the processor's time. However, in some cases, efficiency can either increase **or** decrease based on the time taken on certain processes. Another way to allow for better performance is to use context-switches as rarely as possible; Depending on how often they occur, they can take up processor time to the point where other applications would not be allowed to run.

7. (10 points) Give an example event that will cause the following process state transition:

(a) from running to waiting - Happens when an interrupt occurs.

(b) from running to ready - Happens when a process buffers; a process cannot fully execute so it gets queued.

(c) from waiting to ready - Happens when a process is available to execute before the running process is finished executing; The waiting process will be placed into the ‘ready’ queue.