Reid Bixler
Homework 1
CS4414
Operating Systems
Spring, 2016

On my honor as a student I have neither given nor received aid on this assignment.

(1)

<u>parent process</u>

```
 i=0
int main(){
  int i;
  for(i=0; i<4; i++)        2 processes
      fork(); ——→ child 1
  return 0;
}
 i=1
int main(){
  int i;
  for(i=0; i<4; i++)  4 processes
      fork(); ——→ child 2, child 3
  return 0;
}
 i=2
int main(){
  int i;
  for (i=0; i<4; i++) 8 processes
        fork(); → child 4, child 5, child 6, child 7
  return 0;
}
```

i=3

```
int main(){
  int i;
  for (i=0; i<4; i++) 16 processes
        fork(); → child 8, child 9 , child 10, child 11,
  return 0;
}
                    child 12, child13 , child 14, child15
```

i=4

end loop;

16 processes created (including parent process)

---

(2) Explain the circumstances under which the line of code marked printf("LINE J") in the following program will be reached.

```
int main(){
     /* fork a child process */
     pid = fork();
     if (pid < 0) { /* error occurred */
```

```
            fprintf(stderr, "Fork Failed");
            return 1;
        }
        else if (pid==0){ /* child process */
            execlp("/bin/ls", "ls", NULL);
            printf("LINE J");
        }
        else { /* parent process */
            /* parent will wait for the child to complete */
            wait(NULL);
            printf("Child Complete");
        }
        return 0;
    }
```

```
execlp("/bin/ls", "ls", NULL);
        ↓          ↓
       path     command
```

The parent process will first create a child process using the **fork( )** command. The parent's process id will always be non-zero whereas the child's process id will always be 0. Because of this, the parent process will skip the *if* as long as there has been no errors, since the parent's process id should also be positive, and then skip the *else if* because the parent has a non-zero process id. Finally the parent will default into the *else* clause and execute the **wait( )** command, in order to wait for the child to complete its own process.

Now the child has forked off of the parent and has process id of 0, it has the same exact contents of the parent so it too will go through the first *if* statement. Assuming that the child process doesn't have an error by having a negative process id, the child will skip this statement. At the *else if* statement the child's process id of 0 will have it execute the **execlp("/bin/ls", "ls", NULL)** command. The **execlp( )** command "duplicate[s] the actions of the shell in searching for an executable file if the specified file-name does not contain a slash (/) character).

Since the first argument in this case does contain a slash (/) character, the child process will execute the second argument as a command and replace the contents of the child process with this command. As long as the **execlp( )** command executes successfully, then the child process will actually finish there by calling **exit( )**, and no longer finish the rest of what was after the command. This is because **exec** functions do not return due to their creation of a new process image which replaces the old process. In order for the **printf("LINE J")** command to execute, The **execlp( )** function must have had an error, and thus will return a value of **-1**, allowing for the **printf( )** command to execute. Otherwise, the **printf( )** command will never be executed.

(3) Using the program shown below, 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)

```
int main() {
	/* fork a child process */
	pid = fork();     pid_c = 2603

	if (pid < 0) { /* error occurred */
		fprintf(stderr, "Fork Failed");
		return 1;
	}
	else if (pid = 0) { /* child process */
		pid1 = getpid(); → get child's pid = 2603
		printf("child: pid = %d", pid); /* A */ → 0
		printf("child: pid1 = %d", pid); /* B */ → 2603 (Child's pid)
	}
	else { /* parent process */
		pid1 = getpid(); → get parent's pid = 2600
		printf("parent: pid = %d", pid); /* C */ → 2603 (child's pid)
		printf("parent: pid1 = %d", pid); /* D */ → 2600 (parent's pid)
		wait(NULL);
	}
	return 0;
```

A: 0, B: 2603, C: 2603, D: 2600

(4). Using the program shown below, explain what the output will be at lines X and Y. (You will get no point if you just write down the result without explanation)

```
#define SIZE 5

int nums[SIZE] = {0, 1, 2, 3, 4};

int main() {
	int i;
	pid_t pid;

	pid = fork();
		for (i = 0; i < SIZE; i++) {
			nums[i] *= -i;
			printf("CHILD: %d ", nums[i]); /* LINE X */
		}
	}
	else if (pid > 0}
        wait(NULL)'
        for(i = 0; i < SIZE; i++)
            printf("PARENT: %d ", nums[i]; /* LINE Y */
        }
    }
    return 0;
}
```

```
parentNums = {0, 1, 2, 3, 4}
childNums = {0, 1, 2, 3, 4}
int i = 0
    child[0] = 0
    parent = WAIT
> CHILD: 0

parentNums = {0, 1, 2, 3, 4}
childNums = {0, 1, 2, 3, 4}
int i = 1
    child[1] = -1
    parent = WAIT
> CHILD: -1

parentNums = {0, 1, 2, 3, 4}
childNums = {0, -1, 2, 3, 4}
int i = 2
    child[2] = -4
    parent = WAIT
> CHILD: -4

parentNums = {0, 1, 2, 3, 4}
childNums = {0, -1, -4, 3, 4}
int i = 3
    child[3] = -9
    parent = WAIT
> CHILD: -9

parentNums = {0, 1, 2, 3, 4}
childNums = {0, -1, -4, -9, 4}
int i = 4
    child[4] = -16
    parent = WAIT
> CHILD: -16

parentNums = {0, 1, 2, 3, 4}
childNums = {0, -1, -4, -9, -16}
int i = 0
    child = QUIT
    parent[0] = 0
> PARENT: 0

parentNums = {0, 1, 2, 3, 4}
childNums = {0, -1, -4, -9, -16}
int i = 1
    child = QUIT
    parent[1] = 1
```

```
> PARENT: 1

parentNums = {0, 1, 2, 3, 4}
childNums = {0, -1, -4, -9, -16}
int i = 2
        child = QUIT
        parent[2] = 2
> PARENT: 2

parentNums = {0, 1, 2, 3, 4}
childNums = {0, -1, -4, -9, -16}
int i = 3
        child = QUIT
        parent[3] = 3
> PARENT: 3


parentNums = {0, 1, 2, 3, 4}
childNums = {0, -1, -4, -9, -16}
int i = 4
        child = QUIT
        parent[4] = 4
> PARENT: 4
```

The reason as to why the values for the parent process do not change is because of the fact that all variables initialized before **fork( )** are independent of any other process. However, the child process also starts off with the same values as the parent process because it takes on the same values in the same address spaces. The important thing to note is that the child process' changing of **nums[ ]** does not affect the values of the parent process. It would be a problem in most cases if the child or parent was able to change the values of each other's variables, which is why **fork( )** prevents that from happening.