**Student: Yu Cheng (Jade)**

**ICS 412**

**Homework Solution #3**

**September 20, 2009**

## Homework #3

**Exercise #1:** Consider the following program:

```
#include<sys/types.h>
#include<unistd.h>

int main() {
        for (;;) {
                if (! fork()) {
                        exit(0);
                }
                sleep(1);
        }
}
```

What would be an eventual problem if you were to execute this program on your computer? Explain.

**Answer:** Children processes are generated unlimitedly in the infinite loop. The parents do not place any wait()/waitpid() function calls to retrieve the child's exit code. The zombies may eventually fill up the OS's "process table" and cause fork() fail. When fork() fails, it returns a negative value but the error condition is not handled, and the program will be stuck in an infinite loop.

**Exercise #2:** Consider the following program. Assume that the PID of the main process is 33. Each call to fork() in the code is commented to state the PID of the newly created process.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
        pid_t pid;

        // Main process's PID=33

        pid = fork(); // creates process with PID=13
        if (!pid) {
                pid_t pid2 = fork(); // creates process with PID=42
```

```
                if (!pid2) {
                        sleep(100);
                        exit(0);
                }
                if (waitpid(pid2,NULL,0) == -1)  {
                        perror("waitpid()");
                }
                exit(0);
        } else {
                pid_t pid3 = fork(); // creates process with PID=64
                if (!pid3) {
                        sleep(10);
                        fprintf(stderr,"one\n");
                        exit(0);
                }
                pid_t pid4 = fork(); // creates process with PID=89
                if (!pid4) {
                        sleep(30);
                        fprintf(stderr,"two\n");
                        exit(0);
                }
        }
        sleep(20);
        exit(0);
}
```

For both questions assume that all system calls are successful (for brevity, the program does not check error codes.)

**a.** For each running process at the time "one" is being printed out on the screen, other than process 33, give its PID, it's PPID, and the PIDs of its children, if any.

**Answer:**

| PID | PPID | PIDs of its children |
|---|---|---|
| 13 | 33 | 42 |
| 42 | 13 | none |
| 64 | 33 | none |
| 89 | 33 | none |

**b.** For each running process at the time "two" is being printed out on the screen, give its PID, it's PPID, and the PIDs of its children, if any.

**Answer:**

| PID | PPID | PIDs of its children |
|---|---|---|
| 13 | 33 | 42 |
| 42 | 13 | none |
| 89 | 33 | none |

**Exercise #3:** Consider the following (incomplete) program. Based on what we've said in the course, can you think of a reason why this code may be very inefficient? Can you suggest a solution to make it

better? This solution would be in the OS, i.e., the implementation of the fork system call. (Nothing can really be done about rewriting the user code.)

```
#include <sys/types.h>
#include <unistd.h>

int main() {

        int i,j;
        double matrix[1000][1000];
        double vector[1000];

        // Start CPU monitor
        pid_t pid = fork();
        if (pid == -1) {
                perror("fork()");
                exit(1);
        }
        if (!pid) {
                if (execl("/usr/bin/monitor_cpu_usage",
                          "monitor_cpu_usage",argv[0],"12") == -1) {
                        perror("execl()");
                        exit(1);
                }
        }

        // Initialize Matrix
        . . .

        // Do big computation
        . . .

        exit(0);
}
```

**Answer:** The call to fork() in this example is extremely inefficient because the 8+ MB of untouched stack space is copied to the stack of the child process. This is especially useless because the child process never reads from or writes to this stack space. If fork() could be modified to avoid copying uninitialized stack space, the code would become much more efficient.

(Also note that execl() arguments should be terminated with a NULL to indicate the last argument to the new process.)