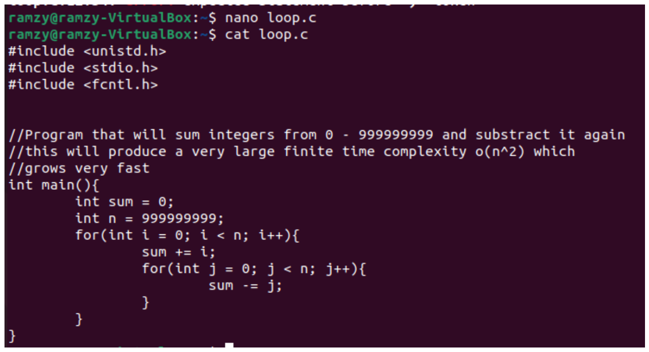
Name : Ramzy Izza Wardhana
NIM : 21/472698/PA/20322
Class : IUP CS B

Assignment 4 – Process and Process Management

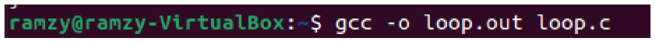
1. Create a program X that runs an arbitrarily large finite amount of loop. Run the program and print the process table using ps. What is the state of the process X during execution? Send a signal to stop and continue process X. How does the state of process X change? Explain with some screenshots of this process.

   Step 1: Create a C program that contains large finite amount of loop, in this case, we create a summation of natural non-negative numbers until 999999999 and subtract it again until the final value will equal to 0. This loop will grow very large with respect to time and space.

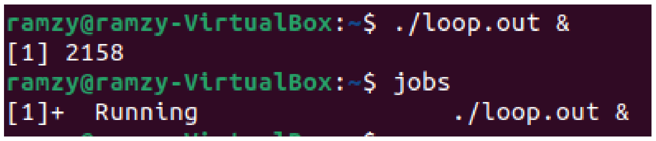


```
ramzy@ramzy-VirtualBox:~$ nano loop.c
ramzy@ramzy-VirtualBox:~$ cat loop.c
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>


//Program that will sum integers from 0 - 999999999 and substract it again
//this will produce a very large finite time complexity o(n^2) which
//grows very fast
int main(){
        int sum = 0;
        int n = 999999999;
        for(int i = 0; i < n; i++){
                sum += i;
                for(int j = 0; j < n; j++){
                        sum -= j;
                }
        }
}
```

   Step 2: Next, we will compile the program and run the program in the background by using ampersand (&) and check the state with jobs

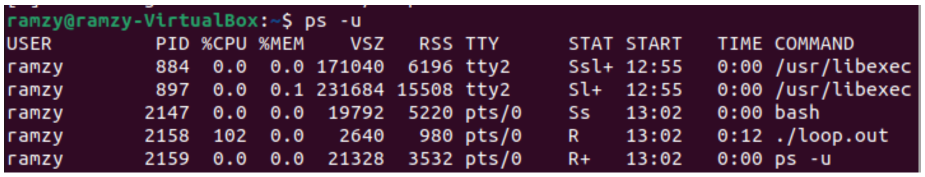


```
ramzy@ramzy-VirtualBox:~$ gcc -o loop.out loop.c
```



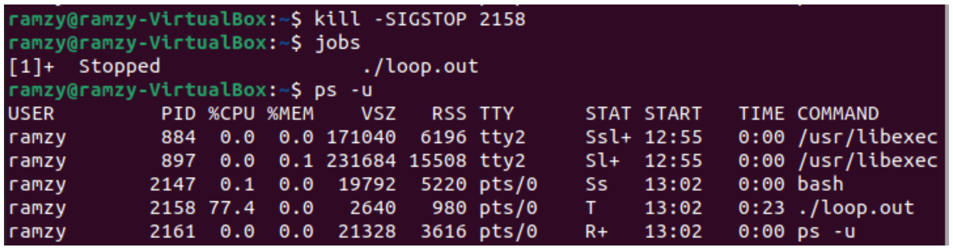
```
ramzy@ramzy-VirtualBox:~$ ./loop.out &
[1] 2158
ramzy@ramzy-VirtualBox:~$ jobs
[1]+  Running                 ./loop.out &
```

   Step 3: After that, let's check the process status of the program running in the background. As we may see, the loop program is denoted with the PID of 2158 and the status R (Running / Runnable State)



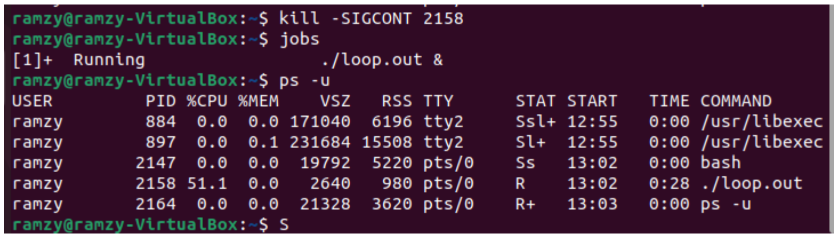
```
ramzy@ramzy-VirtualBox:~$ ps -u
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
ramzy        884  0.0  0.0 171040  6196 tty2     Ssl+ 12:55   0:00 /usr/libexec
ramzy        897  0.0  0.1 231684 15508 tty2     Sl+  12:55   0:00 /usr/libexec
ramzy       2147  0.0  0.0  19792  5220 pts/0    Ss   13:02   0:00 bash
ramzy       2158  102  0.0   2640   980 pts/0    R    13:02   0:12 ./loop.out
ramzy       2159  0.0  0.0  21328  3532 pts/0    R+   13:02   0:00 ps -u
```

   Step 4: Then, we can send a STOP SIGNAL to the loop process by using kill -SIGSTOP (PID) and check the process status table. We can see that the loop process has been stopped denoted with status T (Stopped state)



```
ramzy@ramzy-VirtualBox:~$ kill -SIGSTOP 2158
ramzy@ramzy-VirtualBox:~$ jobs
[1]+  Stopped                 ./loop.out
ramzy@ramzy-VirtualBox:~$ ps -u
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
ramzy        884  0.0  0.0 171040  6196 tty2     Ssl+ 12:55   0:00 /usr/libexec
ramzy        897  0.0  0.1 231684 15508 tty2     Sl+  12:55   0:00 /usr/libexec
ramzy       2147  0.1  0.0  19792  5220 pts/0    Ss   13:02   0:00 bash
ramzy       2158 77.4  0.0   2640   980 pts/0    T    13:02   0:23 ./loop.out
ramzy       2161  0.0  0.0  21328  3616 pts/0    R+   13:02   0:00 ps -u
```

Step 5: Finally, we can continue the process by sending a CONTINUE SIGNAL to the loop process targeted to the PID (2158)



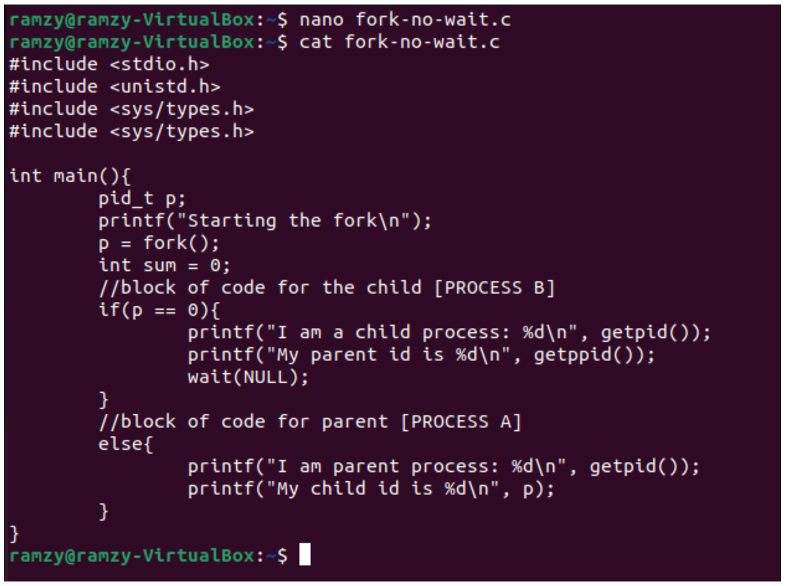
```
ramzy@ramzy-VirtualBox:~$ kill -SIGCONT 2158
ramzy@ramzy-VirtualBox:~$ jobs
[1]+  Running                 ./loop.out &
ramzy@ramzy-VirtualBox:~$ ps -u
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
ramzy        884  0.0  0.0 171040  6196 tty2     Ssl+ 12:55   0:00 /usr/libexec
ramzy        897  0.0  0.1 231684 15508 tty2     Sl+  12:55   0:00 /usr/libexec
ramzy       2147  0.0  0.0  19792  5220 pts/0    Ss   13:02   0:00 bash
ramzy       2158 51.1  0.0   2640   980 pts/0    R    13:02   0:28 ./loop.out
ramzy       2164  0.0  0.0  21328  3620 pts/0    R+   13:03   0:00 ps -u
ramzy@ramzy-VirtualBox:~$ S
```

Taking closer look at the table above, the loop process (2158) has put into R (Running / Runnable State) again since we called the SIGCONT signal

Based on what we have been observed so far, the program will immediately run right after we call *./loop.out* (compiled program). Then, after sending the -SIGSTOP signal targeted to our PID, the loop process will be put into STOPPED STATE (T). Then after sending the -SIGCONT signal, the process will continue and put into RUNNING / RUNNABLE STATE (R).

2. Create a program that creates a parent process A and its child process B. In this program, make process B run longer than process A and process A does not have a wait() system call. Print the process id of A and B on each code block of processes A and B. Run the program and take some screenshots. What is the process id of the parent process on process B's code block? Why?

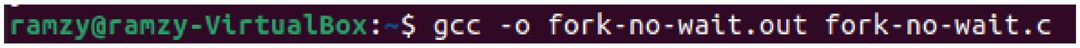
Step 1: Create the program using C languages with no wait() system call on process A.



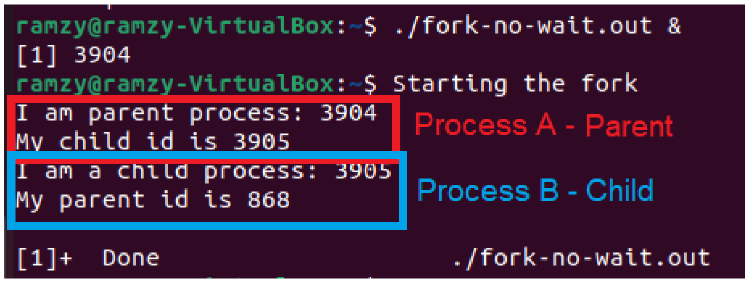
```
ramzy@ramzy-VirtualBox:~$ nano fork-no-wait.c
ramzy@ramzy-VirtualBox:~$ cat fork-no-wait.c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/types.h>

int main(){
        pid_t p;
        printf("Starting the fork\n");
        p = fork();
        int sum = 0;
        //block of code for the child [PROCESS B]
        if(p == 0){
                printf("I am a child process: %d\n", getpid());
                printf("My parent id is %d\n", getppid());
                wait(NULL);
        }
        //block of code for parent [PROCESS A]
        else{
                printf("I am parent process: %d\n", getpid());
                printf("My child id is %d\n", p);
        }
}
ramzy@ramzy-VirtualBox:~$ █
```

Observe that we remove the wait() system call on parent process (A) and replace it on child process (B) which will make the process B runs longer than it supposed to be.

Step 2: Compile the program and run in the background using ampersand(&). The process is denoted with the PID 3904



```
ramzy@ramzy-VirtualBox:~$ gcc -o fork-no-wait.out fork-no-wait.c
```



```
ramzy@ramzy-VirtualBox:~$ ./fork-no-wait.out &
[1] 3904
ramzy@ramzy-VirtualBox:~$ Starting the fork
I am parent process: 3904
My child id is 3905
I am a child process: 3905
My parent id is 868

[1]+  Done                    ./fork-no-wait.out
```

Let's try to analyze what is currently happening in the output here. Taking a closer look at the parent's PID on process A (first line output) it has the value 3904, which corresponds to the assigned PID [1] 3904 right after we run the program.

In contrast, the parent's PID on the second process B (line 4) has the value of 868, which surprisingly doesn't expect our expectation where the parent's PID supposed to be consist of one PID only but has multiple children (one parent with multiple children relation).

Although as we may see, the child's PID has the same value [3905] as the parent's not, it should be noted that this phenomenon could happen due to the presence of a time-dependent process created by Process B that we created before.

The underlying reason behind this is on process B (child) we have made the process run longer which will make process A (parent) complete before the child process. Once the parent process ends or finished before the child process, the process of parent will initialize a new implementation-defined process which in this case has the of PID 868, and this new process will become the new parent of the child.

Hence, it can be summarized that the changes in the parent's PID are caused by the time-dependent process, specifically, when the child process wants to fetch the parent's PID, the parent process has already been completed whereas the child process has not (due to longer processing time). As a consequence, the new parent then will be initiated the so-called implementation-defined process.