**Gufran Azam (ID-23044536)**

Operating Systems CS 340 (M, W 3:10-4:25 PM)

Dr. **Simina Fluture**

**Homework 3:**

**1A) Use Internet sources and give an overview of the command that is used in Windows for creating a process.**

BOOL WINAPI CreateProcess(

\_In\_opt\_    LPCTSTR               lpApplicationName,

\_Inout\_opt\_ LPTSTR                lpCommandLine,

\_In\_opt\_    LPSECURITY\_ATTRIBUTES lpProcessAttributes,

\_In\_opt\_    LPSECURITY\_ATTRIBUTES lpThreadAttributes,

\_In\_        BOOL                  bInheritHandles,

\_In\_        DWORD                 dwCreationFlags,

\_In\_opt\_    LPVOID                lpEnvironment,

\_In\_opt\_    LPCTSTR               lpCurrentDirectory,

\_In\_        LPSTARTUPINFO         lpStartupInfo,

\_Out\_       LPPROCESS\_INFORMATION lpProcessInformation

);

The command that is used for process creation in Windows is called the CreateProcess function. This windows function creates a process with a single thread. The calling process acts as the security context for the new process. Thus when the calling process is impersonating another user the token used for the new process is the token process and not the impersonating process. To run the new process in the security context of the user represented by the impersonation token, use the CreateProcessAsUser or CreateProcessWithLogonW function.

**1B) In a Unix environment, execute parent.c, child.c and orphan.c as follows: Note: first you need to upload the 3 files in your venus home directory.**

[azgu4536@venus ~/cs340]$ gcc parent.c -o parent

**parent.c:** In function ‘**main**’:

**parent.c:4:2:** **warning:** incompatible implicit declaration of built-in function ‘**printf**’ [enabled by default]

printf("Process[%d]: child in execution ...\n",getpid());

**^**

[azgu4536@venus ~/cs340]$ gcc child.c -o child

**child.c:** In function ‘**main**’:

**child.c:4:2:** **warning:** incompatible implicit declaration of built-in function ‘**printf**’ [enabled by default]

printf("Process[%d]: child in execution ...\n",getpid());

**^**

[azgu4536@venus ~/cs340]$ ./parent

Process[24454]: child in execution ...

Process[24454]: child terminating ...

[azgu4536@venus ~/cs340]$ gcc orphan.c -o orphan

[azgu4536@venus ~/cs340]$ pico orphan

[azgu4536@venus ~/cs340]$ ls

child child.c orphan.c parent parent.c

[azgu4536@venus ~/cs340]$ rm orphan

rm: cannot remove ‘orphan’: No such file or directory

[azgu4536@venus ~/cs340]$ rm orphan.c

[azgu4536@venus ~/cs340]$ ls

child child.c parent parent.c

[azgu4536@venus ~/cs340]$ pico

[azgu4536@venus ~/cs340]$ gcc orphan.c -o orphan

[azgu4536@venus ~/cs340]$ ./child

Process[28518]: child in execution ... //the child process starts

Process[28518]: child terminating ...

[azgu4536@venus ~/cs340]$ ./orphan

I'm the original process with PID 28591 and PPID 6501.

I'm the parent process with PID 28591 and PPID 6501.

my child's PID 28592

PID 28591 terminates.

[azgu4536@venus ~/cs340]$ I'm the child process with PID 28592 and PPID 1.

PID 28592 terminates.

// Child terminates because there is no parent, also the process number is 1 more than the last one.

**1C. Write a very simple program that will show the possibility of having zombie processes.**

#include<stdio.h>

main()

{

int id;

id=fork();

if(id>0)

{

printf(“I am the child with pid ….. and my parent has ppid ….”);

sleep(1);

}

if(id==0)

printf(I am the parent and my id is…

sleep(30);

}

**2A. Consider the 2nd attempt (from the lecture notes). Is the “No Starvation” condition satisfied? Hint: in your proof you might want to check if there is a particular execution sequence by which a process might be able to use the CS over and over, while the other process is starving in the while loop.**

The 2nd attempt does not satisfy no starvation. Once a process sets its flag to be true, it is able to keep iterating while the other process is busy waiting. The execution sequence proves this:

flag[0]=false; flag [1]=false;

P0: while(flag[1]){ }; 🡨 skipped

P0: flag[0]=true;

P0: CS

P1: while(flag[0]) {};  <- since flag[0]=true P1 busy waits

P0: flag[0]=false;

P0: remainder section

P0: while(flag[1]){ }; 🡨 skipped

P0: flag[0]=true;

P0: CS

P1: while(flag[0]) {};  <- since flag[0]=true P1 busy waits

P0: flag[0]=false;

P0: remainder section

.

.

.

**This process repeats endlessly and P1 will just continue to busy wait and never get into the Critical Section so it will just starve in the while loop.**

**2B. Prove that the Peterson Solution is correct by showing that all 3 conditions for a correct solution to the Critical Section Problem are respected. Hint: you can use the textbook comments but your proof should be clearer and more detailed.**

**Mutual Exclusion: Proof by Contradiction:**

Assume that both processes can be in the critical section at the same time, and that P0 is in the critical section and P1 is trying to enter the critical section. In order to enter the CS, P1 must first test its wait condition. If P0 is in the CS, then flag[0] = true, and turn had to have been set such that turn = 0 prior to entering the CS. When P1 tries to enter, flag[0] will be true, and turn = 0, which meets the wait condition, therefore blocking P1 from the CS. As such, only one process may inhabit the critical section.

**No Deadlock:** We assume P0 and P1 are stuck in the busy wait state.

If P0 is in the busy wait state it’s because it’s waiting for the turn to be 0 or flag[1]=false

If P1 is in the busy wait state it’s because it’s waiting for the turn to be 1 or flag[0]=false

Thus since turn must be 1 or 0 deadlock cannot occur.

**No Delay:** We assume P0 is in the critical section and P1 is busy waiting and thus flag[0]=true and turn=0.

At the moment P0 exits the critical section it sets the flag[0] to false allowing P1 to exit the busy wait and enter it’s critical section without delay while P0 can go to it’s remainder section.