Student Name

CS 340 – Operating Systems

Dr. Simina Fluture

Homework #3

***Part A - Unix***

1.) Prompt:

[ENTERYOURVENUSNAMEHERE@venus ~]$ man fork

Displayed:

NAME

fork - create a new process

SYNOPSIS

#include <unistd.h>

pid\_t fork(void);

DESCRIPTION

The fork() function shall create a new process. The new process (child process) shall be an exact copy of the calling process (parent pro- cess) except as detailed below:

\* The child process shall have a unique process ID.

\* The child process ID also shall not match any active process group ID.

…

Prompt:

[ENTERYOURVENUSNAMEHERE@venus ~]$ man exec

Displayed:

exec [-cl] [-a name] [command [arguments]]

If command is specified, it replaces the shell. No new process is created. The arguments become the arguments to command. If the -l option is supplied, the shell places a dash at the beginning of the zeroth arg passed to command. This is what login(1) does. The -c option causes command to be executed with an empty environment. If -a is supplied, the shell passes name as the zeroth argument to the executed command. If command cannot be executed for some reason, a non-interactive shell exits, unless the shell option execfail is enabled, in which case it returns failure. An interactive shell returns failure if the file cannot be executed. If command is not specified, any redirections take effect in the current shell, and the return status is 0. If there is a redirection error, the return status is 1.

Prompt:

[ENTERYOURVENUSNAMEHERE@venus ~]$ man wait

Displayed:

wait [n]

Wait for the specified process and return its termination status. n may be a process D or a job specification; if a job spec is given, all processes in that jobs pipeline are waited for. If n is not given, all currently active child processes are waited for, and the return status is zero. If n specifies a non-existent process or job, the return status is 127. Otherwise, the return status is the exit status of the last process or job waited for.

Prompt:

[ENTERYOURVENUSNAMEHERE@venus ~]$ man kill

Displayed:

NAME

kill - terminate a process

SYNOPSIS

kill [ -s signal | -p ] [ -a ] [ -- ] pid ...

kill -l [ signal ]

DESCRIPTION

The command kill sends the specified signal to the specified process or process group. If no signal is specified, the TERM signal is sent. The TERM signal will kill processes which do not catch this signal. For other processes, it may be necessary to use the KILL (9) signal, since this signal cannot be caught.

2.) The command that is used to create a process in windows is the:

CreateProcess Function which creates a new process and its primary thread. The new process runs in the security context of the calling process. If the calling process is impersonating another user, the new process uses the token for the calling process, not the impersonation token. To run the new process in the security context of the user represented by the impersonation token, use the CreateProcessAsUser or CreateProcessWithLogonW function.

Excerpt from: <http://msdn.microsoft.com/en-us/library/ms682425%28VS.85%29.aspx>

3.) Prompt/Execution:

[ENTERYOURVENUSNAMEHERE@venus CS340]$ gcc -o parent parent.c <- Compiles the parent

[ENTERYOURVENUSNAMEHERE@venus CS340]$ gcc -o child child.c <- Compiles the child

[ENTERYOURVENUSNAMEHERE@venus CS340]$ ./parent <- Starts parent

Process[26146]: Parent in execution ... <- Parent process starts

Process[26147]: child in execution ... <- Child process starts

Process[26147]: child terminating ... <- Child finishes process (kill)

Process[26146]: Parent detects terminating child <- Parent sees that child is done

Process[26146]: Parent terminating ... <- Parent finishes process (kill)

[ENTERYOURVENUSNAMEHERE@venus CS340]$ gcc -o orphan orphan.c <- Compiles the orphan

[ENTERYOURVENUSNAMEHERE@venus CS340]$ ./orphan <- Starts the orphan

I'm the original process with PID 26159 and PPID 24241. <- Parent process starts and

I'm the parent process with PID 26159 and PPID 24241. displays its Process ID

my child's PID 26160 <- Child Process ID is displayed

PID 26159 terminates. <- Parent Process is killed

[ENTERYOURVENUSNAMEHERE@venus CS340]$ I'm the child process with PID 26160 and PPID 1. <- Child’s process = orphan

PID 26160 terminates. <- Child Process finishes

***Part B – Synchronization***

1.) *2nd Attempt:*

P0          P1

while (true) {      while(true) {

   while(flag[1]){ };    while(flag[0]){ };

   flag[0] =true;           flag[1]=true;

   CS            CS

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

remainder section; remainder section;

}          } 

This attempt does not satisfy no starvation. Here is a particular execution sequence that proves this:

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

Start in P0: while(flag[1]){ }; <- it skips this because flag[1]=false;

Then in P0: flag[0]=true; <- gets set to true

Then P0 goes into: CS

Switch over to P1: while(flag[0]) {bw};  <- since flag[0]=true it busy waits

Switch back to P0: flag[0]=false; <- then it sets flag[0]=false

Then P0 sits in the: remainder section

P0 then begins again: while(flag[1]){ } ; <- it skips again because flag[1]=false

Then in P0: flag[0]=true;

Then P0 goes into: CS

Switch over to P1: while(flag[0]) {bw}; <- at this point flag[0]=true so it can't access the CS

Switch back to P0: flag[0]=false;

Then P0 sits in the: remainder section

P0 begins again: while(Flag[1]){ } ; <- skips again because flag[1]=false still!

This process continues and P1 will just continue to busy wait and never get into the CS so it will just starve while P0 keeps going.

2.) *Peterson Solution:*

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

P0          P1

while (true) {         while(true) {

flag[0]=true; flag[1]=true;

turn=1; turn=0;

   while (flag[1] and turn==1){ };             while(flag[0] and turn==0){ };

   CS            CS

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

remainder section; remainder section;

}         }

Satisfy Mutual Exclusion: This solution starts out with all flags equaling false and turn equaling 0. When it runs through the first thing it does is it sets flag[0]=true and turn=1 and then checks if flag[1]==true and if turn==1. Since flag[1]=false here it skips the loop and goes straight into the CS. P1 will then busy wait because flag[0] is currently true and it has to wait for flag[0]=false. This means that P1 can never enter the CS at the same time that P0 does because one of the process will always busy wait for the other.

Satisfy No Deadlock: This solution also shows that before either process is in the CS it sets the flag to true and the turn to either 1 or 0. This allows for either process to busy wait for the other until it is done in the CS. After it is done in the CS it sets flag back to false.

Satisfy No Delay: This solution also shows that after a process executes the other process gets to be the next process on the "queue". This is because when it exits the CS the flag of the process is set to false which allows the other process to get out of the busy wait stage and go right into the CS.