Adam Cohen

CS 340

Professor Fluture

Homework #3

**Input**: Man fork

**Output**:

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.

**Input**: Man exec

**Output**:

Name

execve - execute program

Synopsis

#include <unistd.h>

int execve(const char \*filename, char \*const argv[],

char \*const envp[]);

Description

execve() executes the program pointed to by filename. filename must be either a binary executable, or a script starting with a line of the form:

#! interpreter [optional-arg]

**Input**: Man wait

**Output**:

Name

wait, waitpid, waitid - wait for process to change state

Synopsis

pid\_t wait(int \**status*);

Description

All of these system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal. In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state.

2. 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.

3.

Process[22131]: Parent in execution ... 🡨 Parent process begins

Process[22132]: child in execution ... 🡨 Child process begins

Process[22132]: child terminating ... 🡨 Child completes process (kill)

Process[22131]: Parent detects terminating child 🡨 Parent sees child terminated

Process[22131]: Parent terminating ... 🡨 Parent finishes process (kill)

I'm the original process with PID 22144 and PPID 24261. 🡨 Orphan process starts

I'm the parent process with PID 22144 and PPID 24261. 🡨It’s own parent process

my child's PID 22145 🡨 Child Process ID is displayed

PID 22144 terminates. 🡨Parent Process is killed

I'm the child process with PID 22145 and PPID 1. 🡨 Child’s process is the orphan

PID 22145 terminates. 🡨 Child Process finishes

A.

The 2nd attempt does not satisfy no starvation. The following is an execution sequence that 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.

B.

**Mutual Exclusion:** We assume P0 and P1 are in the CS.

If P0 is in the CS: flag[1]=false and turn!=1

If P1 is in the CS: flag[0]=false and turn!=0

The turn is a bit that can be 0 or 1 but not both. The above example shows that for P0 to be in the CS the turn cannot be 1 and for P1 to be in the CS the turn cannot be 0 thus there is a contradiction and they cannot both be in the CS simultaneously.

**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 and flag[1]=false

If P1 is in the busy wait state it’s because it’s waiting for the turn to be 1 and 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.