Lonique Alexander

CS 340

Home work 3

Part A:

1. Use man to find out more about: fork( ), execve( ) commands in Unix. Use man to find more about built-in commands: exec( ), wait( ), kill?

It's in the C library

#include <unistd.h>

parameters is pid\_t fork(void);

The “ fork() creates a new process by duplicating the calling process. The new process, referred to as the child, is an exact duplicate of the calling process, referred to as the parent, except for the following points: “

“ \* The child has its own unique process ID, and this PID does not match the ID of any existing process group (setpgid(2)). ” etc...

The execve() come from the same library the fork. It's parameters are following:

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

\* “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: ”

\* “execve() does not return on success, and the text, data, bss, and stack of

the calling process are overwritten by that of the program loaded. ”

The exec() “The exec() family of functions replaces the current process image with a new process image. The functions described in this manual page are front-ends for execve(2). (See the manual page for execve(2) for further details about the replacement of the current process image.) ”

The wait() come from the C library.

#include <sys/types.h>

#include <sys/wait.h>

The parameters:

pid\_t wait(int \*status);

pid\_t waitpid(pid\_t pid, int \*status, int options);

int waitid(idtype\_t idtype, id\_t id, siginfo\_t \*infop, int options);

waitid(): \_SVID\_SOURCE || \_XOPEN\_SOURCE

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 per‐ formed, then the terminated child remains in a "zombie" state (see NOTES below). If a child has already changed state, then these calls return immediately. Otherwise they block until either a child changes state or a signal handler interrupts the call (assuming that system calls are not automatically restarted using the SA\_RESTART flag of sigaction(2)). In the remainder of this page, a child whose state has changed and which has not yet been waited upon by one of these system calls is termed waitable.”

Last is the kill and it send a signal to a process

SYNOPSIS :

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

kill [ -L | -V, --version ]

kill -l [ signal ]

DESCRIPTION :

“The default signal for kill is TERM. Use -l or -L to list available signals. Particularly useful signals include HUP, INT, KILL, STOP, CONT, and 0. Alternate signals may be specified in three ways: -9 -SIGKILL -KILL. Negative PID values may be used to choose whole process groups; see the

PGID column in ps command output. A PID of -1 is special; it indicates all processes except the kill process itself and init. ”

2. Use Internet sources and give an overview of the command that is

used in Windows for creating a process?

In window NT one can “use the posix fork/exec calls to create a new process”. In addition, “The **CreateProcess** function is used to run a new program. The [WinExec](http://www.cs.rpi.edu/academics/courses/fall01/os/prothred_36w3.htm) **a**nd [LoadModule](http://www.cs.rpi.edu/academics/courses/fall01/os/dll_6vad.htm) functions are still available, but they are implemented as calls to CreateProcess.”

**"**[ExitProcess](http://www.cs.rpi.edu/academics/courses/fall01/os/prothred_8dv7.htm)**,** [ExitThread](http://www.cs.rpi.edu/academics/courses/fall01/os/prothred_531g.htm),[**CreateThread**](http://www.cs.rpi.edu/academics/courses/fall01/os/prothred_4084.htm), [**CreateRemoteThread**](http://www.cs.rpi.edu/academics/courses/fall01/os/prothred_8b38.htm), and a process that is starting (as the result of a call by **CreateProcess**) are serialized between each other within a process. Only one of these events can happen in an address space at a time.”

3. In a Unix environment, execute parent.c, child.c and orphan.c?

souji@souji-laptop:~$ ./parent

Process[16041]: Parent in execution ...

I'm the original process with PID 16042 and PPID 16041.

I'm the parent process with PID 16042 and PPID 16041.

my child's PID 16043

PID 16042 terminates.

Process[16041]: Parent detects terminating child

Process[16041]: Parent terminating ...

souji@souji-laptop:~$ I'm the child process with PID 16043 and PPID 1.

PID 16043 terminates.

My answer. The parent is being executed and given two ID parent ID and PPID. The child given a unique parent ID. The parent process continue to execute concurrently with its children. The child is terminated since the parent has only one child therefore the PPID is terminated.

souji@souji-laptop:~$ ./orphan

I'm the original process with PID 16061 and PPID 12053.

I'm the parent process with PID 16061 and PPID 12053.

my child's PID 16062

PID 16061 terminates.

souji@souji-laptop:~$ I'm the child process with PID 16062 and PPID 1.

PID 16062 terminates.

My answer. The child cant have the original process because it is not parent. When the child is be created a original process is created with PPID. Child is given a child parent ID. Then parent is terminated and then pass to original process.

PART B:Synchronization

A. 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.

Second Attempt: global variable initialized to false.

flag is initialized to 'false'

P0 P1

while(true) { while(true){

while (flag[1]) do no-op; while (flag[0]) do no-op;

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

CS CS

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

remainder section; } remainder section; }

I assume both flag are false. Both denied to enter loop second while loop. P0 flag[0] is then set to true and enter the CS.

Then, P1 is busying waiting because only on process can be in the CS. Once it out the CS flag[0] set to false. P1 enter the remainder. However flag flag[1] is still true. Therefore, if P0 is finish or little more faster the P1 it can enter the CS again before P1 can get in the CS. Thus, we have Starvation.

B. 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.

Peterson's 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) do no-op; while (flag[0] and turn ==0) do no-op;

CS CS

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

remainder section; } remainder section;}

Mutual exclusion:

Proof by contradiction, Assume both process are in the CS. This mean both flag[0] is true and flag[1] is true. If p0 is in the CS that mean turn is 1. In addition, if P1 is in the CS that mean turn is 0. However, turn can be 1 and 0 at the same time therefore the assumption is wrong.

No starvation: prove it for process P0.

Assume P0 is going to CS and it was denied and left in the while loop. If P1 is fast and immediately after leaving it's CS and tries to enter again. It will get denied as well, since turn is 1 and flag[0] is true. The turn is 1 and p0 is in it loop, and p1 go through the CS, performing assignment to turn =0 go CS because flag[0] is true. Now the turn is 0 the other P1 can go to CS. Thus, no starvation.

No Deadlock:

A P0 leave CS flag[0]= false. So will it stay outside the flag remain false. If try to reenter it will fail and process P2 will enter CS. Getting it resoures.