//CS340 - Yuqian Zhang Homework-3 CUNY ID: 23321201

PART A:

1.

(1) [zhyu1201@venus ~]$ man fork

Fork(): takes no argument, 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:

\* has its own unique process id

\* same as parents’ process id

\* does not inherit memory locks from parent

\* process resource utilizations and CPU time counters reset to zero

\* the set of pending signals is initially empty

\* semaphore adjustments not inherited

\* record locks not inherited

\* timers not inherited

\* outstanding asynchronous I/O operations not inherited

Return value:

On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and errno is set appropriately.

(2) [zhyu1201@venus ~]$ man execve

Execve(): always take arguments (name & location of executable file), executes the program pointed by filename. Filename must be either a binary executable, or an interpreter script.

Return value:

On success, execve() does not return, on error -1 is returned, and errno is set appropriately.

(3) [zhyu1201@venus ~]$ man wait

wait [n ...]

Wait for each specified process and return its termination status. Each n may be a process ID or a job specification; if a job spec is given, all processes in that job's 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.

Return value:

If wait() returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. Otherwise, a value of -1 is returned and errno is set to indicate the error.

2.

The CreateProcess function creates a new process, which runs independently of the creating process. However, for simplicity, the relationship is referred to as a parent-child relationship.

If CreateProcess succeeds, it returns a PROCESS\_INFORMATION structure containing handles and identifiers for the new process and its primary thread. The thread and process handles are created with full access rights, although access can be restricted if you specify security descriptors. When you no longer need these handles, close them by using the CloseHandle function.

You can also create a process using the CreateProcessAsUser or CreateProcessWithLogonW function. This allows you to specify the security context of the user account in which the process will execute.

3.

[zhyu1201@venus ~]$ gcc parent.c -o parent

[zhyu1201@venus ~]$ gcc child.c -o child

[zhyu1201@venus ~]$ ./parent

Process[29940]: Parent in execution ...

// parent called fork() to create a new child, which is successful and return value greater than 0, and print Process[29940]: Parent in execution ... by the command in parent source code: printf("Process[%d]: Parent in execution ...\n", getpid());

Process[29941]: child in execution ...

// parent enter sleep status for 5 seconds and call wait method to wait child process. by the command in parent source code: execve("child", NULL, NULL) Child got cpu resource and execute the child program and print Process[29941]: child in execution ...by the command in child source code: printf("Process[%d]: child in execution ...\n",getpid());

Then enter sleep status for 1 second

Process[29941]: child terminating ...

// after 1 second print Process[29941]: child terminating ... by the command in child source code: printf("Process[%d]: child terminating ...\n", getpid());

Process[29940]: Parent detects terminating child

//after 5 seconds sleep, parent got the signal from the wait of child, in which child terminates normally. And then print Process[29940]: Parent detects terminating child by the command in parent source code: printf("Process[%d]: Parent detects terminating child \n", getpid());

Process[29940]: Parent terminating ...

//print Process[29940]: Parent terminating ... by the command in parent source code: printf("Process[%d]: Parent terminating ...\n", getpid());

[zhyu1201@venus ~]$ gcc orphan.c -o orphan

[zhyu1201@venus ~]$ ./orphan

I'm the original process with PID 3589 and PPID 29401.

//print I'm the original process with PID 3589 and PPID 29401. by the command in orphan source code: printf ("I'm the original process with PID %d and PPID %d.\n", getpid(), getppid ());

I'm the parent process with PID 3589 and PPID 29401.

//fork() returned the value >0, return to the parent process. And print I'm the parent process with PID 3589 and PPID 29401. by the command in orphan source code: printf ("I'm the parent process with PID %d and PPID %d.\n",getpid(), getppid());

my child's PID 3590

// print my child's PID 3590 by the command orphan source code: printf ("my child's PID %d\n", pid);

PID 3589 terminates.

//print information by the command in orphan source code: printf ("PID %d terminates.\n", getpid());parent process terminates and executes the child process

[zhyu1201@venus ~]$ I'm the child process with PID 3590 and PPID 1.

//fork() returned the value =0, return to the child process. CPU sleeps 5 seconds in order to make sure parent terminates. When parent dominates, the child has a new parent with PID 1. And then print information by the command: printf ("I'm the child process with PID %d and PPID %d.\n",getpid(), getppid());

PID 3590 terminates.

//print by the command: printf ("PID %d terminates.\n", getpid());

Part B:

No. Not satisfied.

The following sequence will cause starvation: P0 Do CS over and over & P1 starving in the while loop

P0: while exit (flog[1]=F)

P0: set flag[0]=T

P1: while loop (flag[0]=T)

P0: CS

P1: while loop (flag[0]=T)

P0: set flag[0]=F

P0: while exit (flag[1]=F)

P0: set flag[0]=T

P1: while loop (flag[0]=T)

P0: CS

P1: while loop (flag[0]=T)

P0: set flag[0]=F

P0: while exit (flag[1]=F)

P0: set flag[0]=T

…

Part C:

Mutual Exclusive:

By contradiction. If both processes try to enter CS at same time, the turn value will block itself and let another one go first. That means only one process can enter CS because the other one let it enter first by turn’s value.

Progress Condition:

No delay - Assume the P0 is running at CS, P1 is in busy wait. After P0 complete CS, it will set flag[0] to false that let P1 exit busy wait to enter CS right away. Likewise, P1 is running at CS, P0 is in busy wait. After P1 complete CS, it will set flag[1] to false that let P0 exit busy wait to enter CS right away. Flag[0]=false in P0 remainder and flag[1]=false in P1 remainder let other process can use the CS as times it needs. So there is no process delay.

No deadlock –decision as to which process will use the CS next is taken in finite time. P0 run through and P1in busy wait or P1 run through and P0 in busy wait.

Starvation:

No process will wait forever. Either flag[] or turn setting condition in the entry section will prevents it from waiting forever. For example, let’s discuss just on turn. (flag[] is the same situation)

Assume P1 is in the busy wait because(turn=0).If I want to let P0 run all the time to keep P1 starving, I will make sure P0 keeps running the while true loop. But every time it starts, it will reset (turn=1), which breaks the busy wait condition in P1.So there is no possible situation for starvation.