Operating System Midterm exam

1. For each of the following process state transition, say whether the transition is legal and how the transition occurs or why it cannot.
   1. Change from thread state WAIT to thread state RUNNING

This is not legal because when a thread is in the WAIT state, it cannot transition straight to the RUNNING state. To go from WAIT to RUNNING, it has to first transition to the READY state.

* 1. Change from thread state RUNNING to thread state WAIT

This is legal because a thread can enter the WAIT state when it needs to wait for a resource. This would usually happen through system calls like “wait()”, where the thread goes to a blocked state until the resource or condition it is waiting for becomes available.

* 1. Change from thread state READY to thread state WAIT

This is legal because a thread in the READY state is ready to run, but it can be blocked if it needs to wait for some other resource. This may happen because the thread may not be able to continue its execution until the resource is available.

1. Write a program that opens a file (with the open () system call) and then calls fork () to create a new process. Can both the child and parent access the file descriptor returned by open ()? What happens when they are writing to the file concurrently, i.e., at the same time?

#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h> // For open()

#include <unistd.h> // For fork(), write(), close()

int main() {

// Open a file with read & write permissions, create if it doesn't exist

int fd = open("output.txt", O\_CREAT | O\_WRONLY | O\_TRUNC, 0644);

if (fd < 0) {

perror("Error opening file");

return 1;

}

// Create a new process

pid\_t pid = fork();

if (pid < 0) {

perror("Fork failed");

return 1;

}

if (pid == 0) {

// Child process

write(fd, "Hello from child\n", 17);

} else {

// Parent process

write(fd, "Hello from parent\n", 18);

}

close(fd);

return 0;

}

The parent process and the child process can access the file descriptor returned by open(). When fork() is called, the child process inherits a copy of the parent’s file descriptors. They still share the same file offset and file status flags.

When they are both writing concurrently, they may overlap in unpredictable ways. In this code, there is nothing that can ensure that the writing does not overlap. The output overall might get mixed up.

1. Write another program using a fork (). The child process should print “hello”; the parent process should print “goodbye”. You should try to ensure that the child process always prints first; can you do this without calling wait () in the parent?

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main() {

pid\_t pid = fork(); // Create a child process

if (pid < 0) {

// If fork() fails

perror("Fork failed");

return 1;

}

if (pid == 0) {

// Child process

printf("hello\n");

exit(0);

} else {

// Parent process

printf("goodbye\n");

}

return 0;

}

1. Write a program that creates a child process, and then in the child closes standard output (STDOUT FILENO). What happens if the child calls printf () to print some output after closing the descriptor?

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

int main() {

pid\_t pid = fork(); // Create a child process

if (pid < 0) {

// If fork() fails

perror("Fork failed");

return 1;

}

if (pid == 0) {

// Child process

close(STDOUT\_FILENO);

printf("This will not be printed by the child.\n");

// After closing stdout, this printf will have no output

} else {

// Parent process

printf("This will be printed by the parent.\n");

}

return 0;

}

If the child calls printf() to print some output after closing the descriptor, no output will be printed.

1. Consider the following piece of C code:

void main ( ) {

fork ( );

fork ( );

exit ( );

}

How many child processes are created upon execution of this program?

3 child processes are created upon execution of that program.

1. An interactive shell program such as bash shell (terminal in Linux and Mac) or PowerShell or CMD prompt in Windows takes command line input from the user and then execute the command/program specified by the user. In this exercise, you will implement closh (Clone Shell), a simple shell-like program designed to run multiple copies of a program at once.  
   Like any other shell, closh takes as input the name of the program to run (e.g.,  
   hello world). However, closh also takes two additional inputs:
2. The number of copies (processes) of the program to run. This is an  
   integer from 1 to 9.
3. Whether the processes should be executed concurrently or  
   sequentially, in sequential execution, the shell should wait for every  
   time a program is executed. In case concurrent execution, the shell  
   does not need to wait for the program to complete execution.

Closh executes the given program, the specified number of times, then  
returns to the prompt once all processes have either completed. Here is a  
simple example of using closh

(italic is user input and hello.exe is the hello world as in previous question):  
mint@mint:~$ ./closh  
closh> ./hello.exe  
count> 3  
[p]arallel or [s]equential> p  
hello world  
hello world  
hello world  
closh>

To write such a shell in C++, refer to the process lecture. As noted in the slide, you can use “execvp()” to create a new child process and have it execute a command. Use also the “waitpid” in the parent process to wait for the child process to finish in case  
sequential execution is selected.  
For simplicity, assume that the user specifies the full path name for any  
command/executable that they wish to execute. Thus, you do not need to deal  
with path name completion issues. You can test your shell on programs you  
write in C++ (after compiling to machine language).

#include <iostream>

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

#include <cstring>

using namespace std;

void run\_program(const char\* program, int count, char mode) {

pid\_t pid;

for (int i = 0; i < count; ++i) {

pid = fork();

if (pid < 0) {

cerr << "Fork failed!" << endl;

exit(1);

}

if (pid == 0) {

// Child process

char\* const args[] = { (char\*)program, nullptr }; // Program arguments

execvp(program, args); // Replace the child process with the program

cerr << "Exec failed!" << endl;

exit(1);

} else {

// Parent process

if (mode == 's') {

// If sequential, wait for the child to finish before forking the next

waitpid(pid, nullptr, 0);

}

}

}

if (mode == 'p') {

// If parallel, wait for all child processes to finish

for (int i = 0; i < count; ++i) {

waitpid(-1, nullptr, 0); // Wait for any child process to finish

}

}

}

int main() {

string program;

int count;

char mode;

while (true) {

// Prompt the user for input

cout << "closh> ";

getline(cin, program); // Get the program to run

if (program.empty()) break;

cout << "count> ";

cin >> count;

if (count < 1 || count > 9) {

cerr << "Count must be between 1 and 9!" << endl;

cin.ignore(numeric\_limits<streamsize>::max(), '\n'); // Clear input buffer

continue;

}

cout << "[p]arallel or [s]equential> ";

cin >> mode;

cin.ignore(); // Clear the newline character left by the previous input

if (mode != 'p' && mode != 's') {

cerr << "Invalid mode! Use 'p' for parallel or 's' for sequential." << endl;

continue;

}

// Run the program with the specified count and mode

run\_program(program.c\_str(), count, mode);

}

return 0;

}

Create a GitHub repository and upload this document with answers to question 1 and 6 with all your .cpp or .c programs. Copy the repository link and paste it in Canvas assignment.