CS 4348/5348 Operating Systems -- Project 2

The goal of this project is to let you get familiar with the use of socket. Also, you will practice the use of a few more unix system calls, including settimer and signal (the interrupt handler). The project has to be programmed in C++ and executed under Unix (or Linux). We divide the project into two different phases.

1 Phase 1

Modify your Project 1 code (from Phase 2). Convert the adder and factorial code to separate programs. After the main process created two child processes, it should use "exec" to execute the adder and factorial programs. At the same time, build sockets between the programs for communication (instead of the pipes used in the original program). Use a configuration file "config.dat" to specify the IP addresses and port numbers of the adder and factorial programs. The format of the configuration file is as follows:

```
mainp <main process's IP> < main process's port number>
adder <adder's IP> <adder's port number>
facto <factorial's IP> <factorial's port number>
N M
```

N and M will be explained later. All three processes should read this file in the beginning to establish sockets (their IPs will be the same, but we still specify it to make it more convenient). If multiple students use the same port number, there will be conflict, causing socket creation failure. If that happens, please change your port number.

The main process needs to make sure that the sockets of the child processes are ready before reading and sending instructions. You can let the child processes send ready messages to the main process after their sockets are created in order to synchronize.

2 Phase 2

Unlike the first project, we use a different method to achieve synchronization. First, you can do the same as the first project, try to let the main process continuously sending new instructions to the child processes and see whether the socket will have data loss or the sender will be blocked. Then, you can implement the following synchronization mechanism. Same as the first project, you need to make sure that the number of pending instructions (buffered by the socket) in each child process is within a bound N. But in this project, the child processes keep the count.

The main process, after sending an instruction to the child process, sends a signal SIGUSR1 to the child process using "kill" system call. The child process should handle this interrupt signal. The handler increases a counter (which keeps track of the number of pending instructions). If the counter reaches N, then the handler signals the main process using SIGUSR1, to inform the main process to stop sending any more instructions. At the same time, the child process should decrement the counter after finishing each instruction. When the number of instructions is below M, and if the main process has been informed to stop sending the instructions, the child process should send a message to the main process to reactive the main process to send instructions.

The main process should have a signal handler for SIGUSR1, which sets a flag to stop the main process from reading and sending instructions. Same as project 1, the main process will stop sending instructions to either child after it receives SIGUSR1 interrupt, even though the number of pending instructions of the other child process may not have exceeded its threshold. The main process can block itself on reading the socket for acknowledgements from the child processes. You should only use one socket for reading acknowledges (best is to use the same as the one in Phase 1 which reads the ready message from the child processes) and you need to differentiate whose acknowledgements they are in order to achieve proper control.

Since both child processes may send SIGUSR1 to the main process, the main process should keep counts and reads acknowledgements correspondingly in order to ensure correct synchronization. Also, in each child process, the counter may be manipulated by both the signaler and the normal process, there is potential of interference. To ensure safe programming of signal handler, it would be better to keep two counters, one to count the number of instructions receive and the other to keep track of the number of instructions completed.

3 Phase 3 (Do not submit this version, just for you to investigate)

Instead of using socket to read acknowledgements for synchronization, the child process can use signal SIGUSR2 to inform the main process to continue sending instructions. You can change your code to rid the socket based acknowledgement mechanism and use the signaling mechanism instead. In this case, the main process needs to know who has sent each signal (for both SIGUSR1 and SIGUSR2) in order to achieve proper synchronization. You can use the signalfd system call to obtain the information.

When the main process sends signals to the child process after sending each instruction, the consecutive signals may be sent quickly and cause previous signal handler(s) being interrupted by new signals. In a severe situation, the new signals may come so fast that the child process will never get a chance to finish any signal handler logic and would fail to inform the main process to stop sending new instructions. It is also possible that the update to the counter by the signal handlers would conflict and resulting in incorrect updates. To solve this problem, signal handler can block other signals while processing a signal. You can add this into your program to achieve correct synchronization (the solution in Phase 2 could encounter problems sometimes).