CS2106 Operating Systems

Semester 1 2021/2022

Tutorial 4

IPC & Threads

1) (Shared Memory) In this question, we are going to analyze the pitfalls of having multiple processes accessing and modifying data at the same time. Compile and run the code in **shm.c**. When running the executable, you can specify two command line arguments, **n** and **nChild**; if no command line arguments are given, the default values will be used: n is initialized to 100 and nChild is initialized to 1.

   The code in shm.c does a simple job: the parent creates nChild processes and then each process, the parent included, increases a shared memory location by n times. After all processes have finished incrementing the shared value, the parent prints the shared result and exits. Because the value of the shared memory is initialized to 0, we expect the printed value to be equal to (1 + nChild) * n at the end. Is this always the case?

   Try running the program multiple times with increasingly higher values for n and nChild (n = 10000 and nChild = 100 should do the trick). Explain the results.

2) (Message Passing) In the lecture we talked about message passing between processes. Here we are implementing **Direct Communication** beween master and worker processes.

Pseudo message passing API
`Send( Pid , Msg );` Send ***Msg*** to the process whose process id is ***Pid***.
`Receive( Pid , Msg );` Receive ***Msg*** from the process whose process id is ***Pid***.

   As a continuation of Q3 from Tutorial 2, let us investigate how to implement the same prime factorization program using message passing. Fill in the pseudo code for the master and worker processes below. Your pseudocode must ensure that the master process receives results from workers in the same order as the user input is sent to workers by the master.

   For each of the message passing operations, indicate whether it is blocking or non-blocking.

   For simplicity, you can assume the following: We have 1 master process and N worker processes. Master process is responsible for distributing workload among worker processes. Worker processes have been already spawned via a fork call by master process.

Master Process:

```
workerPid[] = PID array for 1..N worker processes
N = number of inputs
inputs[] = N user inputs

//Give user input to the corresponding worker to work on


//Wait for each worker to finish and get back the result
```

Each Worker Process:

```
//Get the user input to work on from master


result = PrimeFactorization(_____________);


//Send back the result to master
```

3) (Thread vs Process)
   a) For each of the following scenarios, discuss whether **multithreading** or **multiprocessing** is the best implementation model. If you think that both models have merits for a particular scenario, briefly describe additional criteria you would use to choose between the models.
      i) Implementing a command line shell.
      ii) Implementing the "tabbed browsing" in a web browser, i.e., each tab visits an independent webpage.
      iii) Implementing a complex multi-player game with dynamic environments and sophisticated.
   b) Suppose task A is CPU-intensive and task B is I/O intensive (reading from a file), which of the following statement(s) is/are TRUE if the two implementations are executed on a single CPU (single core)?
      i) If the threads are implemented as user threads, then multithreaded implementation will finish execution in shorter amount of time compared to the sequential implementation.
      ii) If the threads are implemented as kernel threads, then multithreaded implementation will finish execution in shorter amount of time compared to the sequential implementation.

4) (Threads)

a) Suppose we have the following multi-threaded processes on a **hybrid thread** model:

Process ID	Number of User Threads	Number of Kernel Threads
P1	U1	K1
P2	U2	K2
P3	U3	K3

i) If all threads are CPU intensive and run forever, what is the approximate proportion of CPU time utilized by the process P3 after running for a long time? You can assume a pre-emptive and fair process scheduler is used by the OS. You can express your answer in terms of U1-U3 and K1-K3.

ii) When [U1 = 4, K1 = 3] [U2 = 3, K2 = 1] [U3 = 5, K3 = 2] what is the CPU utilization by one user thread of process P1? Does the number of user threads of other process affect this value? State your assumptions.

b) Evaluate each of the following statements regarding Thread and POSIX Thread (pthread) as True or False.

i) All pthreads must execute the same function when they start.

ii) Multi-threaded program using pure user threads can never exploit multi-core processors.

iii) If we use a 1-to-1 binding in a hybrid thread model, the end result is the same as pure-user thread model.

iv) On a single-core processor that supports simultaneous multithreading, we can execute more than one thread at the **same time**.

Questions for your own exploration

5) (Protecting the Shared Memory) Allowing processes to access and modify shared data whenever they please can be problematic! Therefore, we would like to modify the code in **shm.c** such that the output is deterministic regardless of how large n and nChild are. More precisely, we want the processes to take turns when modifying the result value that resides in the shared memory. To this end, we will add another field in the shared memory, called the order value, that specifies which process' turn it is to increment the shared result. If the order value is 0, then the parent should increase the shared result, if the order value is 1, then the first child should increase it, if the order value is 2, then the second child and so on. Each process has an associated pOrder and checks whether the value order is equal to its pOrder; if it is, then it proceeds to increment the shared result. Otherwise, it waits until the order value is equal to its pOrder.

Your task is to modify the code in **shm_protected.c** to achieve the desired result. You will have to:

a) Create a shared memory region with two locations, one for the shared result, and the other one for the order value;

b) Write the logic that allows a process to modify the shared result only if the order value is equal to its pOrder (the skeleton already takes care of assigning the right pOrder to each process);
c) Print the result value and cleanup the shared memory.

Why is **shm** faster than **shm_protected**? Why is running **shm_protected** with large values for nChild particularly slow? (Hint: You may want to take a look at the output of *htop*)