CS2106 Operating Systems

Semester 1 2020/2021

Week of 1st March 2021

Tutorial 5

Synchronization

Note: Synchronization is important to both multithreaded and multi-process programs. Hence, we will use the term **task** in this tutorial, i.e. do not distinguish between process and thread.

1. [*Race Conditions*] Consider the following two tasks, *A* and *B*, to be run concurrently and use a shared variable *x*. Assume that:
   - load and store of *x* is atomic
   - *x* is initialized to 0
   - *x* must be loaded into a register before further computations can take place.

Task A	Task B
`x++;` `x++;`	`x = 2*x;`

   How many **relevant** interleaving scenarios are possible when the two threads are executed? What are all possible values of *x* after both tasks have terminated? Use a step-by-step execution sequence of the above tasks to show all possible results.

2. [Critical Section] Can disabling interrupts avoid race conditions? If yes, would disabling interrupts be a good way of avoiding race conditions? Explain.

3. [Semaphore] Consider three concurrently executing tasks using two semaphores S1 and S2 and a shared variable *x*. Assume S1 has been initialized to 1, while S2 has been initialized to 0. What are the possible values of the global variable *x*, initialized to 0, after all three tasks have terminated?

A	B	C

`P(S2);`
`P(S1);`
`x = x*2;`
`V(S1);`
	`P(S1);`
	`x = x * x;`
	`V(S1);`
		`P(S1);`
		`x = x + 3;`
		`V(S2);`
		`V(S1);`

*Note: P(), V() are a common alternative name for Wait() and Signal() respectively.

4. [Semaphore] In cooperating concurrent tasks, sometimes we need to ensure that all N tasks reached a certain point in code before proceeding. This specific synchronization mechanism is commonly known as a **barrier**. Example usage:

```
//some code

Barrier( N );  //The first N-1 tasks reaching this point
               // will be blocked.
               //The arrival of the Nth task will release
               //  all N tasks.

//Code here only get executed after all N processes
// reached the barrier above.
```

   Use semaphores to implement a **one-time use `Barrier()`** function **without using any form of loops.** Remember to indicate the variables declarations clearly.

For your own exploration

5. [Thread vs Process] For each of the following scenarios, discuss whether **multithread** or **multiprocess** 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.
   a. Implementing a command line shell.
   b. Implement the "tabbed browsing" in a web browser, i.e. each tab visit an independent webpage.
   c. Solving a challenging problem using concurrent / parallel method.
   d. Implement a web server. (A webserver handles request of webpage from clients. It retrieves the correct webpage and sends to the client).

6. [Using pthread] Let us revisit the **prime factorization** from tutorial 2. In this question, we will use **pthread** to write a multithreaded program as an alternative to multiprocess model.

   To help you along, two source codes are given in the `exploration/` folder:
   - **`tParameter.c`**: A simple ***pthread*** program to illustrate how to pass parameter and receive result from a thread starting function.
   - **`PF_Sequential.c`**: A sequential version of the prime factorization program.

a. (Let's thread!) Use the skeleton **`PF_Thread.c`** to write a multithreaded version of the prime factorization program. [Quick recap: ask user for number input, N ➔ read N integer input ➔ run prime factorization on each input.]

   Note that pthread does not support "waiting for any thread" functionality directly (i.e. there is no equivalent of ***`wait`***()). So, we will just wait for the spawned thread in order.

b. (Let's compare) Use the *nix command **`time`** to find out the time consumed by your program. e.g.

   "**`time ./PF_Thread < test2.in`**"

   will report the time used by the executable "**`PF_Thread`**" with the input redirection from "**`test2.in`**"

   Do the same experiment for the sequential version (PF_Sequential) and the multiprocess version (PF_Process) from tutorial 2. What do you think is the meaning of "real time, user time and system time" reported? Comment on the real time vs user time statistics collected from your system. [You can replace "`time`" with "`/usr/bin/time -v`" on Linux system for even more comprehensive info.]

7. [*General Semaphores*] We mentioned that general semaphore (S > 1) can be implemented by using **binary semaphore** (S == 0 or 1). Consider the following attempt:

```
int count =  <initially: any non-negative integer>;
Semaphore mutex = 1;   //binary semaphore
Semaphore queue = 0;     //binary semaphore, for blocking tasks
```

```
GeneralWait() {
    wait( mutex );
    count = count - 1;
    if (count < 0) {
        signal( mutex );
        wait( queue )
    } else {
        signal( mutex );
    }
}
```

```
GeneralSignal() {
    wait( mutex );
    count = count + 1;
    if (count <= 0) {
        signal( queue );
    }
    signal( mutex );
}
```

Note that for ease of discussion, we allow the count to go negative in order to keep track of the number of task blocked on queue.

a. The solution comes very close to a working solution, but unfortunately can still have **undefined behavior** in some execution scenarios. Give one such execution scenario to illustrate the issue. (hint: binary semaphore works only when its value S = 0 or S = 1).

b. [Challenge] Correct the attempt. Note that you only need very small changes to the two functions.

8. [Synchronization Problem - Challenging] We explore *the sleeping barber* problem. The sleeping barber has a barbershop with 10 chairs for customers who wait for a haircut, and the barber chair. If a customer enters, and all the waiting chairs are occupied, then the customer queues outside the shop. If the barber is busy, but a waiting chair is available, then the customer sits in one of the free chairs. When there are no customers, the barber sleeps. Customers sit in the barber chair during the haircut, allowing more customers to wait seated.

   a. Complete the code skeleton below to synchronize the barber and customer processes using semaphores.

```
void barber(){
   while(1)
    {
          …
        cutHair();
          …
     }
}

void customer()  {
     while(1)
     {
          …
      delay(rand()); // wait for a random amount of time
                        between barbershop visits
     }
}
```

   b. Modify your code to allow for customers to leave the shop if there are no free chairs, instead of queueing outside.

   c. Modify the solution for (a) to synchronize the barber and customer processes using indirect asynchronous message passing. Assume that messages are passed through message queues of finite capacity, specified at the creation time, with sending (receiving) processes blocking when and only when the queue is full (empty).

   d. Explain why this solution is not any more elegant compared to solution for (a). Why is it that in the case of producer-consumer the solution using message passing is much more elegant compared to the solution with semaphores? Why is it that this is not the case here?