CS 214 Recitation(Sec. 6)

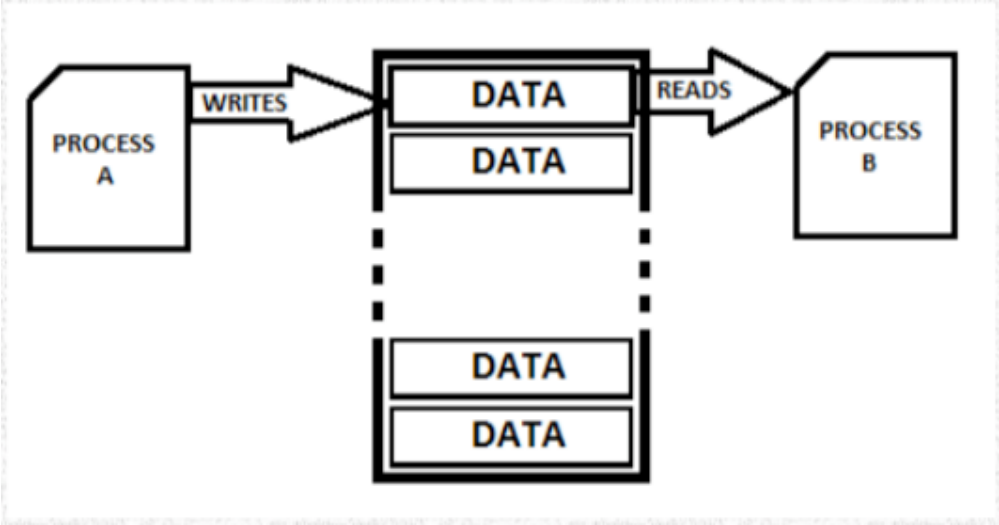
Zuohui Fu
Ph.D. Department of Computer Science
Office hour: Mon 2pm-3pm
Email: zf87 AT cs dot rutgers dot edu
11/08/2017

Topics

- Synchronization
- HW6

What is Synchronization

- Several Processes run in an Operating System while some of them share resources due to which problems like data inconsistency may arise
- Ex: One process changing the data in a memory location where another process is trying to read the data from the same memory location. It is possible that the data read by the second process will be erroneous



Race Condition

- A race condition is the behavior that the output is dependent on the sequence of other uncontrollable events. In case the events do not occur as the developer wanted, a fault happens.
- The term originates with the idea of two events racing each other to influence the output.
- How do we guarantee correct interaction between threads? Using Synchronization!

Race Condition

- Example in BOARD

Implement synchronization:

- Mutual exclusion (Mutex): used for exclusive access to a shared resource (critical section) . Also to avoid the simultaneous use of a common resource, such as a global variable.
- Semaphores: counting number of available “resources”, manipulated atomically through two operations.
- Conditional variables: wait for a specific event to happen, tied to a mutex for exclusive access operations: wait for event, signal occurrence of event

Semaphores

- wait(semaphore): decrement
- signal(semaphore): increment

```
void wait(sem S)
{
    S.count--;
    if (S.count < 0) {
        add the caller to the waiting list;
        block();
    }
}
```

```
void signal(sem S)
{
    S.count++;
    if (S.count <= 0) {
        remove a process P from the waiting list;
        resume(P);
    }
}
```

After decreasing the counter by 1, if the counter value becomes negative, then add the caller to the waiting list, and then block itself.

After increasing the counter by 1, if the new counter value is not positive, then vremove a process P from the waiting list, resume the execution of process P, and return

Blocking Semaphores

- wait() is called by a thread
  - if semaphore is “available”, thread continues
  - if semaphore is “unavailable”, thread blocks, waits on queue
- signal() opens the semaphore
  - if thread(s) are waiting on a queue, one thread is unblocked
  - if no threads are on the queue, the signal is remembered for next time a wait() is called

How to use

- Example in BOARD

Monitor

- Sometimes Semaphores can be hard to use
- Monitor is a software module that encapsulates:
  - shared data structures
  - procedures that operate on the shared data
  - synchronization between concurrent processes that invoke those procedures

Example

```
Struct counter{
        int count;
        pthread_mutex_t lock;
}

Void increment(struct counter *c)
pthread_mutex_lock(&c -> lock);
int n = c -> count = n + 1;
pthread_mutex_unlock(&c -> lock);
```

Additional Reading

- Deadlock
  - http://www2.latech.edu/~box/os/ch07.pdf
  - https://web.cs.wpi.edu/~cs3013/c07/lectures/Section07-Deadlocks.pdf

HW6 - Threads Synchronization

Write a function that uses threads to *synchronize printing* between them to print out a triangle. Make *two threads*, one to print out even rows, one to print out odd. The goal is to print out:

```
*
**
***
****
*****
******
```

where each line is printed by a different thread. Have **thread 0** print out the even lines (2 stars, 4 stars, etc) and **thread 1** print out the odd ones (1 star, 3 stars, etc).

HW6 - Threads Synchronization

At first, just run them with no synchronization. You'll get an interleaving of rows ... likely you'll get a batch of rows and then another batch of rows.

Next add a pair of mutexes to trade off control between the threads. They should trade off using the mutexes to synchronize between them.

Sometimes thread 0 may be scheduled before thread 1 and you get the rows printed in the wrong order. You need to make sure the first thread gets the mutex first - but you can't, really since you do not have control of the scheduler.

HW6 - Threads Synchronization

Change the mutex into a binary semaphore. This way if the wrong thread starts first, it will block until the other thread gets to run.

Thread 1 ought to notify/produce and thread 0 ought to wait/consume. This way you can trade control in an intentional manner.