ENSC 351

Process Synchronization

Dr. Brian Fraser

Slides for course derived from
Dr. Mohamed Hefeeda's slides

Objectives

- Understand
  - The Critical-Section Problem
  - Practical solution to the critical section problem

- Definition:
  - Atomic instruction:
    ..
    (i.e., cannot be partially completed)

Consumer-Producer Problem

- Classic example of process coordination
- Two processes sharing a buffer
- Producer: places items into the buffer
  - Must wait if the buffer is full
- Consumer: takes items from the buffer
  - Must wait if buffer is empty
- Solution:
  - Keep a counter on number of items in the buffer

```
// Shared
int count = 0;
int buffer[BUFFER_SIZE];
```

Producer/Consumer Threads

Producer Thread

```
int in = 0;
while (true) {
    // Produce a new item
    item = …;

    // Store item into buffer
    while (count == BUFFER_SIZE)
        ;  // do nothing

    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    count++;
}
```

Consumer Thread

```
int out = 0;
while (true)  {
    // Get next item to process
    while (count == 0)
        ;    // do nothing

    nextItem =  buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;

    // Consume nextItem
    ...
}
```

What can go wrong with this solution?

Race Condition

- count++ could be implemented as
  - register1 = count
  - register1 = register1 + 1
  - count = register1
- count-- could be implemented as
  - register2 = count
  - register2 = register2 - 1
  - count = register2
- Consider this execution interleaving with “count = 5” initially:

Inst #	Producer	Consumer
S0	reg1 = count	
S1	reg1 = reg1 + 1	
S2		reg2 = count
S3		reg2 = reg2 – 1
S4	count = reg1	
S5		count = reg2

.. What other final values in count are possible depending on how these 6 instructions are ordered?

Race Condition

- Race Condition: when multiple processes..

  and the result depends on the..

  - Data inconsistency may arise
- Solution idea
  - Mark code segment that manipulates shared data as a..
  - If a process is executing its critical section, no other processes can execute their critical sections
- More formally, any method that solves the Critical-Section Problem must satisfy three requirements …<next>

Critical-Section (CS) Problem

1. **..**
   If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **..**
   If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then selection of the process that will enter the critical section next cannot be postponed indefinitely
3. **..**
   Must limit the number of times other processes can enter their critical sections after a process requested to enter its critical section and before that request is granted

- Assumptions
  - Each process executes at a nonzero speed
  - No restriction on the relative speed of the N processes

Solutions for CS Problem

- Disable interrupts during CS
  - Currently running code would

    ..
  - Only works on single CPU system:
    - Other processors executing code could enter a CS
    - Plus, every processor has its own interrupts: don't want to disable interrupts system wide.
  - Drawback of disabling interrupts (even on uniprocessor systems)
    - Disabled interrupts makes system unresponsive: .. for servicing interrupts.

- Hardware instructions provide low level test-and-set primitives to prevent mutual exclusion of a CS.

Recap

- Processor Synchronization
  - Techniques to coordinate access to shared data
- Race condition
  - Multiple processes manipulating shared data and result depends on execution order
- Critical section problem
  - Three requirements: mutual exclusion, progress, bounded waiting
- Up next:
  - Semaphores
  - Some classical synchronization problems

Higher Level Synchronization: Semaphores

Semaphores

- Usually provided by OS kernel
  - Much easier to use than hardware-based solutions
- Semaphore *S* holds..
- Two standard operations to modify *S:*
  - wait()
  - signal()
- Each operation is indivisible (**atomic**)

Semaphore Operations

- **If implemented in C, it would look like:**

```
int S = 1;        // Resource available
void wait (int S)  {
        // Busy wait for S>0, called a spinlock
        while (S <= 0) {
                // Do nothing; OK to interrupt here.
        }
        S--;
}

void signal(int S) {
        S++;
}
```

- **In reality a process may block when waiting instead of using a spinlock (busy-wait)**

Semaphore Types and Usage

- **Two types**
  - Counting semaphore: can be any integer value
  - Binary semaphore: can be 0 or 1
    Called a..
- **Usage examples**
  - Counting semaphore: Allow 4 concurrent radio transmissions
  - Mutex: Allow at most 1 concurrent access to thruster valve control
- **Mutual exclusion** (mutex)

```
Semaphore mutex;    // Initialized to 1
wait (mutex);
// Critical Section code here.
signal (mutex);
```

Semaphore Usage (cont'd)

- ..
  myBar() in P2 should execute after myFoo() in P1

```
Process P1:
    myFoo();
    signal (sem);

Process P2:
    wait (sem);
    myBar();
```

- **Control access to a resource with a ..**
  - e.g., producer-consumer problem with finite buffer

Semaphore Implementation

- Must guarantee that no two processes can execute wait and signal on same semaphore at same time
- Thus, code in wait and signal becomes .. and must be protected by:
  - Disabling interrupts (uniprocessor systems only)
  - Busy waiting or spinlocks (multiprocessor systems)
- **But why did this help?**
  - **Use semaphore to avoid spinlock, but need spinlock to implement semaphore!**
  - However, wait and signal are.. , so a spinlock is not so bad.
  - Whereas an application may spend long (and unknown) amount of time in its critical sections.

Deadlock and Starvation

- Deadlock – two or more processes are waiting **indefinitely** for..

- Let S and Q be two semaphores initialized to 1

Process 0	Process 1

wait(S);
wait(Q);
... do some work...
signal(S);
signal(Q);
	wait(Q);
	wait(S);
	... do some work...
	signal(Q);
	signal(S);

- Starvation – ..
  - A process may never be removed from the semaphore queue in which it is suspended

Be Careful When Using Semaphores

- Some common programming problems:
  - signal (mutex) …. wait (mutex)
    - Multiple processes can access CS at the same time
  - wait (mutex) … wait (mutex)
    - Processes may block for ever
  - Forgetting wait (mutex) or signal (mutex)
    - Deadlocks or prevents mutual exclusion.
  - Not identifying a required critical section
    - No mutual exclusion: race cases.

Recap

- Semaphores
  - wait() - Acquire / consume the resource
  - signal() - Mark resource as available
- Mutex
  - Lock & Unlock
- Race case:
  - The behaviour depends on the order tasks execute and changes unexpectedly
- Critical section
  - Managing access to shared data to prevent race cases
- Up next:
  - Some classical synchronization problems

Classical Problems of Synchronization

- Dining-Philosophers Problem
- Readers-Writers Problem

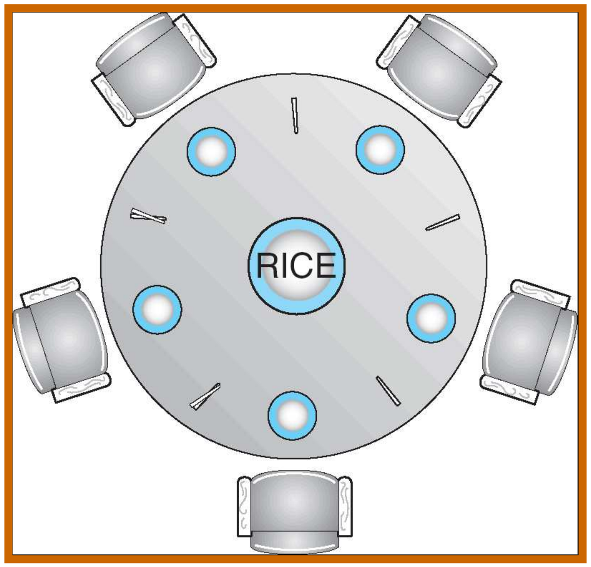
- These problems are
  - abstractions that can be used to model many other resource sharing problems
  - used to test newly proposed synchronization schemes

Dining-Philosophers Problem

- Philosophers alternate between eating and thinking
  - To eat, a philosopher needs..

    (at her left and right)

- Models:
  multiple processes (philosophers)
  sharing multiple resources (chopsticks)



- Write a program for each philosopher such that no starvation/deadlock occurs

Dining-Philosophers Problem: Philosopher *i*

First try:

```
while (true)  {
          wait ( chopstick[i] );
          wait ( chopstick[ (i + 1) % 5] );
          // now eat
          signal ( chopstick[ (i + 1) % 5] );
          signal ( chopstick[i] );
          // now think
}
```

- What can go wrong with this solution?
  - All philosophers..

- Solution
  - All philosophers pickup lower-numbered chopstick first

Readers-Writers Problem

- Data shared by many concurrent processes
  - – only read the data; no updates
  - – can read and write
- Problem
  - Allow.. to read at the same time.
  - But.. to write at a time (no readers).

Readers-Writers Problem (cont'd)

- Some systems implement reader-writer locks
  - E.g., Linux, Pthreads API
  - A process can ask for a reader-write lock either in read or write mode
- When would you use reader-writer locks?
  - Applications where it is easy to identify readers only and writers only processes
  - Applications with more readers than writers

pthreads Synchronization

- POSIX-Threads (pthreads) API is OS-independent
- It provides:
  - mutex locks
  - semaphores
  - read-write locks
  - spin locks

```
#include <pthread.h>
pthread_mutex_t mutex;

pthread_mutex_init(&mutex, null);
pthread_mutex_lock(&mutex);
pthread_mutex_unlock(&mutex);
```

```
#include <semaphore.h>
sem_t  sem;
sem_init(&sem, 0, 5);
sem_wait(&sem);
sem_post(&sem);
```

Summary

- Some classical synchronization problems
  - Dining-philosophers
  - Reader-writer
- pthreads