Agenda

- Assignment 2 is out, due October 20!
- Exam Schedule is out!
- Today's lecture
  - Critical section problem
  - Software and hardware solution
- Reading: Sections 3.4, 4.4-4.5, 6.1-6.2
- Next : More on Synchronization

Mutual Exclusion

- Consider a block of shared memory
- Process P wants to write data to the shared memory while Process R wants to read the data
- **Sequential execution**: a synchronous system call used to cause the R to wait until the P is complete
  - The approach works because the synchronous system calls provide mutual exclusion
- **Concurrent execution**: maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Mutual Exclusion

- **Destructive Interference**: when cooperation among concurrent processes results in incorrect behavior
- **Critical section** (region): section of code within a process that requires access to shared resources and must not be executed while another process is in a corresponding section of code
- **Race Condition**: A case where multiple threads read and write a shared data item and the final result depends on the relative timing of their execution
- When a resource (i.e., memory) can be accessed by only one process at a time, we say that the resource is ***<u>mutually exclusive</u>***
- To prevent race conditions, we use mutual exclusion and critical sections

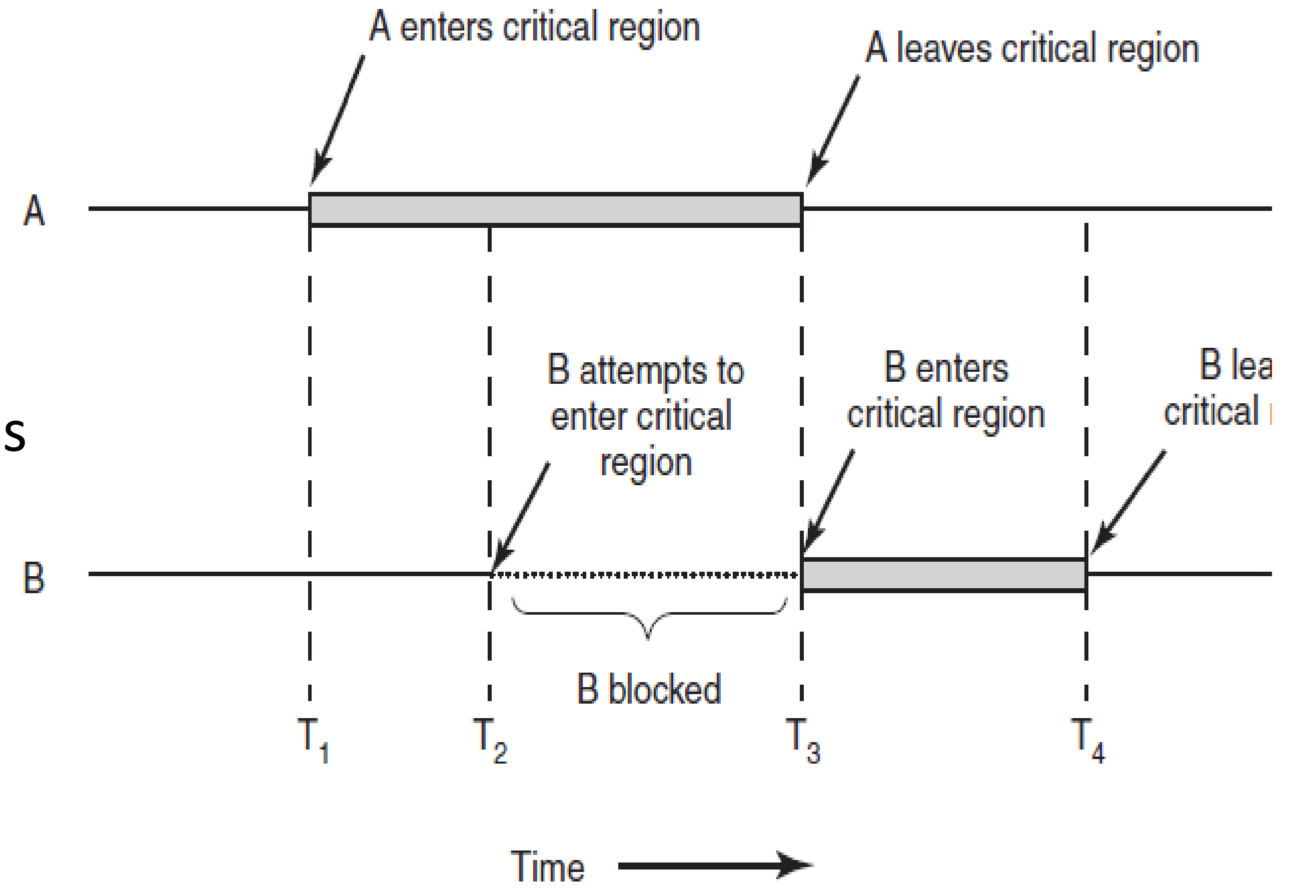
Critical Region Requirements

1. **Mutual exclusion**: No two processes may be simultaneously inside their critical regions.

2. **Scheduler independent**: No assumptions may be made about speeds or the number of CPUs.

3. **Allows progress**: No process running outside its critical region may block other processes.

4. **Starvation free**: No process should have to wait forever to enter its critical region



Typical Process Structure

```
while (true) {

      entry section

         Critical section

      exit section


      remainder section
}
```

Locks

- Locks can be used as mutual exclusion mechanisms
  - Prevent threads from entering a critical section if another thread is present
  - Allow threads to wait, and eventually enter their critical section
- Locks support two operations
  - Acquire: locks the critical section so that it is safe to enter
  - Release: unlock the critical section on exit
- Both hardware and software solutions are possible

```
while (true) {

      acquire lock

            critical section

      Release lock
            remainder section
}
```

Software Solution 1: Lock Variable

```
// thread
while (true) {
        entry section
        while (busy);
        busy = true
        critical section
        busy = false
}
```

Busy waiting

busy is a shared variable

```
// thread
while (true) {
        entry section
        while (busy);
        busy = true
        critical section
        busy = false
}
```

1. **Mutual exclusion:** ☹
2. **Scheduler independent:** ☺
3. **Allows progress:** ☺
4. **Starvation free:** ☹

Problems

- Requires busy waiting
- Prone to starvation

Software Solution 2: Strict Alternation

```
// thread
while (true) {
        entry section
        while (turn == 1);
        critical section

        Turn = 1;
        remainder section
```

```
// thread
while (true) {
        entry section
        while (turn == 0);
        critical section

        Turn = 0;
        remainder section
```

Busy waiting

Turn is a shared variable

1. **Mutual exclusion:** ☺
2. **Scheduler independent:**☺
3. **Allows progress:** ☹
4. **Starvation free:** ☺

Problems

- Works for 2 processes!
- Requires busy waiting
- One thread may block the other!

Software Solution 3: Peterson's Solution

- Based on Dekker's solution
- Processes share two variables:
  - int turn;
  - boolean flag[2]
- **Strict alternation**: the variable turn indicates whose turn it is to enter the critical section.
- **Lock:** the flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P_i is ready!

Software Solution 3: Peterson's Solution

```
//Producer thread
while (true) {
      entry section
      flag[i] = true;
      turn = j;
      While (flag[j] && turn == j);
      critical section
      flag[i] = false;
```

```
//Consumer thread
while (true) {
      entry section
      flag[j] = true;
      turn = i;
      While (flag[i] && turn == i);
      critical section
      flag[j] = false;
```

Entering CS

Let other thread go first.

Wait while busy

Leaving CS

flag and turn are shared

1. **Mutual exclusion:** ☺
2. **Scheduler independent:**☺
3. **Allows progress:** ☺
4. **Starvation free:** ☺

Problems

- Works for 2 processes! ☹
- Requires busy waiting ☹
- Assumes writes and reads are atomic! ☹

Hardware Solution: Interrupt Disabling

- Many systems provide hardware support for critical section code
- **Uniprocessors – could disable interrupts**
- In a uniprocessor system, concurrent processes cannot have overlapped execution; they can only be interleaved
- A process will continue to run until it invokes an OS service or until it is interrupted
- Therefore, to guarantee mutual exclusion, it is sufficient to prevent a process from being interrupted
- This capability can be provided in the form of primitives defined by the OS kernel for disabling and enabling interrupts

Hardware Solution: Interrupt Disabling

Disadvantages

- The efficiency of execution could be noticeably degraded because the processor is limited in its ability to interleave processes
- This approach will not work in a multiprocessor architecture

- Modern machines provide special *atomic* hardware instructions

Atomic Instruction

- Atomic Instruction: a function or action implemented as a sequence of one or more instructions that appears to be indivisible.
  - No other process can see an intermediate state or interrupt the instruction
  - The sequence of instruction is guaranteed to execute as a group, or not execute at all
  - Atomicity guarantees isolation from concurrent processes

Hardware Solution: Special Machine Instruction

Test&Set Instruction (TSL instruction)

- Takes one operand, a Lock variable, to coordinate access to shared memory
- A **test** is made between memory value and a test value
- If the values are the same, a **set** value occurs

```
enter_region:
        TSL REGISTER,LOCK        | copy lock to register and set lock to 1
        CMP REGISTER,#0          | was lock zero?
        JNE enter_region         | if it was nonzero, lock was set, so loop
        RET                      | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0             | store a 0 in lock
        RET                      | return to caller
```

Hardware Support: Using Test and Set

```
// thread
while (true) {
      entry section
      while (TestSet(busy));
      critical section
      busy = false
}
```

1. **Mutual exclusion: ☺**
2. **Scheduler independent:☺**
3. **Allows progress: ☺**
4. **Starvation free: ☹**

Problems

- Requires busy waiting ☹
- Starvation is possible

Hardware Solution: Special Machine Instruction

Compare&Swap Instruction (XCHG instruction)

- Also called a “compare and exchange instruction”
- A **compare** is made between a memory value and a test value
- If the values are the same a **swap** occurs
- Carried out atomically (not subject to interruption)

```
enter_region:
        MOVE REGISTER,#1             | put a 1 in the register
        XCHG REGISTER,LOCK           | swap the contents of the register and lock variable
        CMP REGISTER,#0              | was lock zero?
        JNE enter_region             | if it was non zero, lock was set, so loop
        RET                          | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0                 | store a 0 in lock
        RET                          | return to caller
```

Hardware Solution: Special Machine Instruction

Advantages

- It is applicable to any number of processes on either a single processor or multiple processors sharing main memory
- It is simple and therefore easy to verify
- It can be used to support multiple critical sections; each critical section can be defined by its own variable

Hardware Solution: Special Machine Instruction

Disadvantages

- **Busy waiting is employed**
  - Thus, while a process is waiting for access to a critical section, it continues to consume processor time
- **Starvation is possible**
  - When a process leaves a critical section and more than one process is waiting, the selection of a waiting process is arbitrary. Thus, some process could indefinitely be denied access
- **Deadlock is possible**
- **Difficult to implement**

Sleep and Wakeup

- Previous solutions require busy waiting!
  - Wasted CPU time
  - Unexpected effects (e.g., *priority inversion problem*)
- ***Sleep*** causes the caller to block, until another process wakes it up
- ***Wakeup*** causes a thread to resume work after it was put to sleep
- How is this helpful?

Sleep and Wakeup

- To enter critical section (lock)
  - If thread can enter critical section
    - Enter critical section (acquire lock)
  - else
    - Add thread to a queue
    - Put thread to sleep until critical section is available
- When leaving critical section (unlock)
  - If another thread is waiting to enter critical section
    - Remove thread from queue
    - Wake thread
  - Leave critical section
- When a thread is woken (resumed)
  - Thread enters critical section (locks it)

```
while (true) {

    acquire lock

            critical section

    Release lock
            remainder section
}
```

Implementation Issues

- How do we lock the queue in order to add/remove threads?
  - Back to the original problem
- What happens if we resume a thread before it suspends itself?
  - In some systems it does not matter (In Java it does)
  - Thread can make sure that the resume “worked” (tricky)

Alternative

Let the System be responsible for entering/leaving critical sections

- Advantages:
  - Since System is in charge, no race conditions in entering/leaving critical sections
  - Programmers do not have to write/debug critical section entry/leave routines
  - Code becomes simpler
- Disadvantages:
  - Code becomes system
  - These routines are more expensive, since they invoke the system
  - May do more than necessary
- In general a “System Solution” is a good thing
- Question: What kind of interface should the System provide?

Abstractions for Mutual Exclusion

- We have several options:
  - Locks (mutex locks)
    - Usually not a good idea
    - Many implementations do not allow a thread to acquire its own lock
    - A spin lock is a lock mechanism that requires the thread to spin in a loop testing a condition
  - Semaphores
  - Monitors

Semaphore

- Synchronization tool that does not require busy waiting
- Abstraction: Semaphore *S* – integer variable
- Can only be accessed via two indivisible (atomic) operations:
  - **acquire()** and **release()**
  - Originally called **P()** and **V()**
- Less complicated

```
acquire() {
    while value <= 0
       ; // no-op
    value--;
}

   release() {
       value++;
   }
```

Semaphore Usage

- Associate a semaphore **sem** with a critical section
- To enter a critical section, thread acquires **sem**
- To leave CS, thread releases **sem**
- **Two types:**
  - **Counting** semaphore – integer value can range over an unrestricted domain
  - **Binary** semaphore – integer value can range only between 0 and 1; can be simpler to implement
    - Also known as **mutex locks**
- How do we implement a Semaphore?

```
Semaphore sem = new Semaphore(1);

sem.acquire();

   // critical section

sem.release();

   // remainder section
```

Java Example Using Semaphores

```java
public class Worker implements Runnable {
private Semaphore sem;
public Worker(Semaphore sem) {
              this.sem = sem;
}
public void run() {
       while (true) {
              sem.acquire();
              criticalSection();
              sem.release();
              nonCriticalSection();
       }
}
}
```

```java
public class SemaphoreFactory{
public static void main(String args[]) {
   Semaphore sem = new Semaphore(1)

   Thread[] bees = new Thread[5];
   for (int i = 0; i < 5; i++)
     bees[i] = new Thread(new Worker(sem));

   for (int i = 0; i < 5; i++)
     bees[i].start();
   }
}
```

Semaphore Implementation – No Busy waiting

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - block – place the process invoking the operation on the appropriate waiting queue.
  - wakeup – remove one of processes in the waiting queue and place it in the ready queue.

Semaphore Implementation

```
public class Semaphore{
  private int value;
  public Semaphore(int value) {
    this.value = value;
  }
....
 }
```

```
  public synchronized void release() {
    ++value;
    notify();
  }
```

- Wake up a waiting process

```
public synchronized void acquire() {
    while (value <= 0) {
      try {
        wait();
      }
      catch (InterruptedException e) { }
    }
  value--;
 }
```

- Avoid busy waiting by putting the thread to sleep
- System can use a queue and block all threads that are waiting

Semaphore Implementation

- Must guarantee that no two processes can execute **acquire ()** and **release ()** on the same semaphore at the same time

- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.
  - Could now have busy waiting in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied

- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Deadlock and Starvation

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

- Let S and Q be two semaphores initialized to 1
    - When p0 executes q.aquire
    - it must wait until p1 executes q.release
    - Similarlty, when p1 executes S.aquire
    - It must wait until p0 executes S.release

P_0	P_1
`S.acquire();`	`Q.acquire();`
`Q.acquire();`	`S.acquire();`
.	.
.	.
.	.
`S.release();`	`Q.release();`
`Q.release();`	`S.release();`

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