Synchronization

CS 416: Operating Systems Design, Spring 2011

Department of Computer Science
Rutgers University

Synchronization

Basic problem:

Threads are concurrently accessing shared variables

The access should be controlled for predictable result.

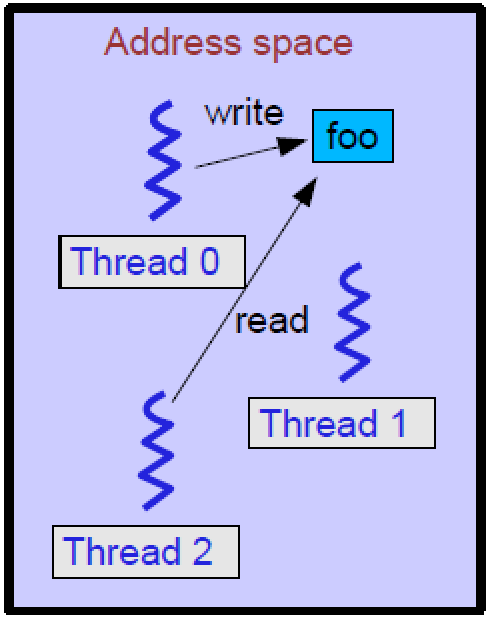
Solution: Need a mechanism to control the access

Low-Level Mechanism

- Locks

Higher Level Mechanism

- Mutexes
- Semaphores
- Condition Variables
- Monitors



Shared Variable Example

Suppose we want to withdraw money from a bank

```
int withdraw(account, amount) {
  balance = get_balance(account);
  balance = balance - amount;
  put_balance(account, balance);
  return balance;
}
```

Suppose you are sharing this account (with $1500) with your friend and you both withdraw $100 from this account at the same time. – What happens ?

Example (continued)

Suppose we represent this situation with a separate thread for each ATM user doing a withdrawal.

- Both threads run on the same bank server

Thread 1

```
int withdraw(account, amount) {
  balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

Thread 2

```
int withdraw(account, amount) {
  balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

What is the problem with the above approach ?

What are the possible values for balance ?

Interleaved Execution

The execution of the threads could be *interleaved*

- Assume preemptive schedule
- Each thread can context switch after every instruction

Execution Schedule as seen by CPU

balance = $1500

```
balance = get_balance(account);
balance -= amount;
```

balance = $1400

Context Switch

```
balance = get_balance(account);
```

balance = $1500

```
balance -= amount;
```

balance = $1400

```
put_balance(account, balance);
```

balance = $1400

Context Switch

```
put_balance(account, balance);
```

balance = $1400

Race Conditions

Two concurrent threads accessed a shared resource without any synchronization. This is called ***Race Condition.***

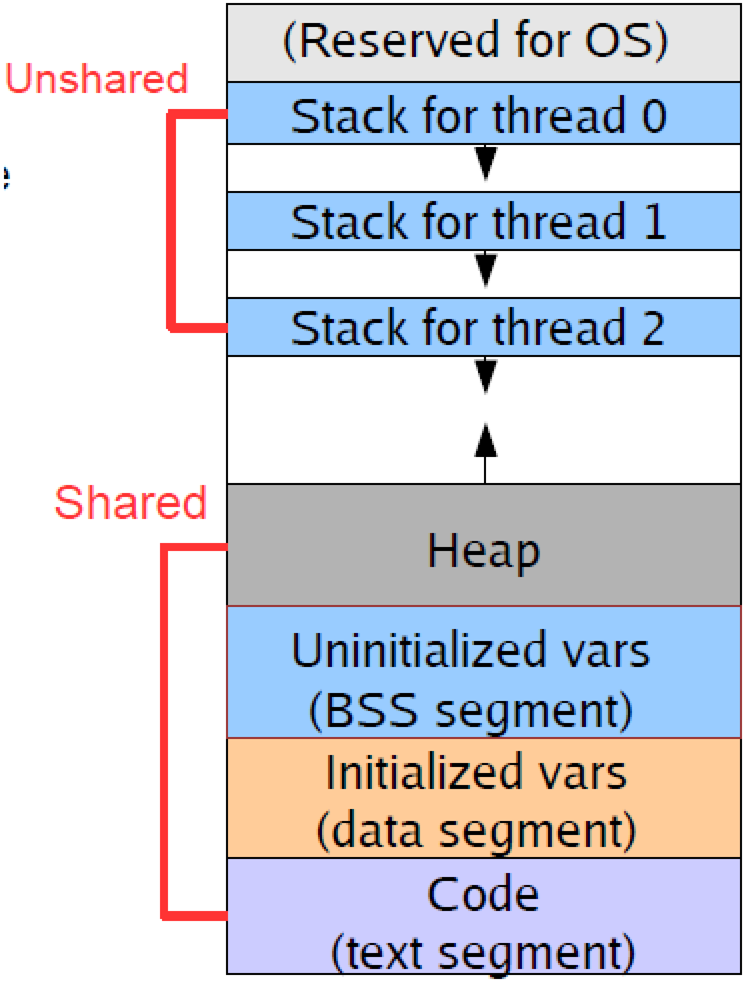
*The result of race condition is **non-deterministic***

By introducing synchronization, we bring in ***determinism***

Synchronization is necessary for any shared data structure

Queue, buffers, lists, hash-tables

Which Resources are shared ?



Local Variables – ***Not Shared***

Allocated in stack - private to every thread

Global Variables – ***Shared***

Allocated in Global segment

Dynamically Allocated Variables - ***Shared***

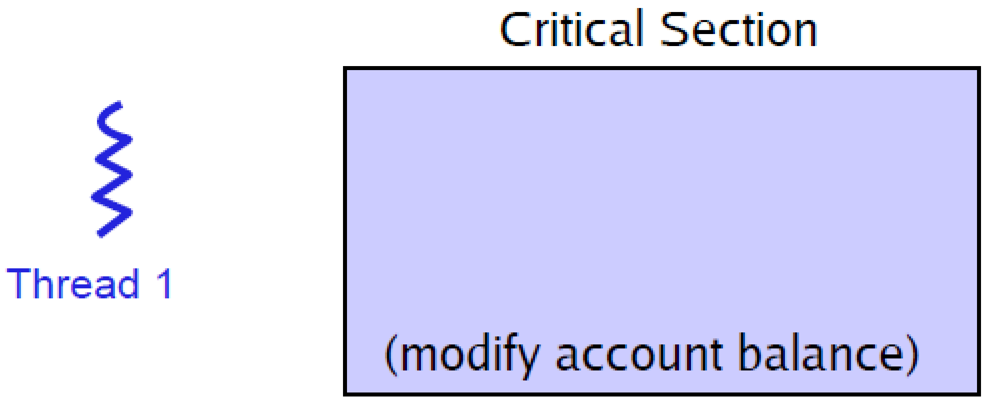
Allocated in Heap

Mutual Exclusion

We want to use *mutual exclusion* to synchronize access to shared resources

Critical Section:

- **Code that uses mutual exclusion to synchronize its execution**
- **Only one thread can execute in Critical Section at any time**
- **All other threads are forced to wait on entry**



Critical Section Requirements

Mutual Exclusion

Only one thread is executing in the critical section

Progress

If Thread-1 is outside the critical section, Thread-1 cannot prevent Thread-2 from entering the critical section

Bounded Waiting

If Thread-1 is waiting outside the critical section, Thread-1 should ultimately be able to enter the critical section

Performance

The overhead of entering and exiting the critical section should be small compared to the work being done within the critical section

Different Solutions

Software solutions to Mutual Exclusion (Peterson's Solution)

- Hard to get it right in modern architecture
- Wastes CPU cycles

Hardware Supported solutions

Low-Level Constructs

Locks

Higher-Level Constructs

Mutexes

Semaphores

Condition Variables

Monitors

Software Solution to Mutual Exclusion

```
Int turn
Boolean flag[2]
```

← Variables shared between 2 processes i,j

```
do {
    flag[i] = TRUE;
    turn = j;
    while(flag[j] && turn ==j);

        Critical Section

    flag[i] = FALSE;

        remainder section

} while(TRUE)
```

Busy Waiting (points to the while loop)

Are CS conditions met?

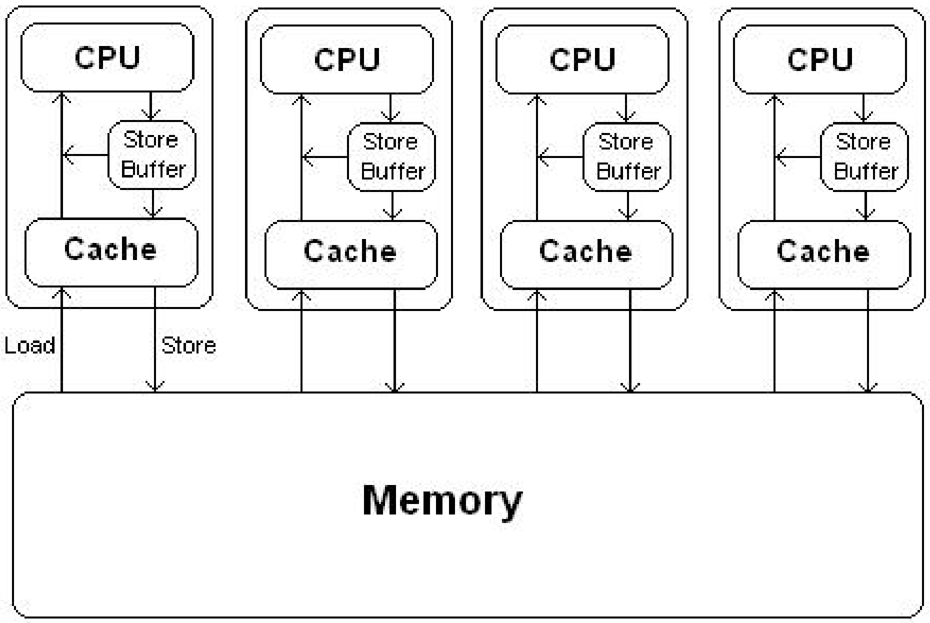
-Mutual Exclusion

-Progress

-Bounded Waiting

What is the problem with the software solutions?

Problem with software solution



The way load and store instructions work on modern multiprocessor systems, there is no guarantee that software solutions would work

When a process/thread executes a store instruction, the data is put into the ***store buffer***. The buffered data is sent to the cache sooner or later, but not necessarily right away.

Locks

A ***Lock*** is an Object(in memory) with the following 2 operations:

- **acquire(): A thread calls this before entering the CS**
- **release(): A thread calls this after leaving the CS**

A call to acquire() MUST have a corresponding call to release()

- Between acquire() and release(), the thread holds the lock
- acquire() does not return until the caller holds the lock
  - **At most one thread can hold a lock at any time**

What happens if acquire() and release() are not paired ?

Using locks

```
int withdraw(account, amount) {
  acquire(lock);
  balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  release(lock);
  return balance;
}
```

critical section

Why is the return statement outside the critical section ?

Execution with locks

```
acquire(lock);
balance = get_balance(account);
balance -= amount;
```

Thread 1 runs

```
acquire(lock);
```

Thread 2 waits on lock

```
put_balance(account, balance);
release(lock);
```

Thread 1 completes

```
balance = get_balance(account);
balance -= amount;
put_balance(account, balance);
release(lock);
```

Thread 2 resumes

Implementing Locks - **Spinlocks**

Spinlocks: Very simple way to implement locks

```
struct lock {
  int held = 0;
}
void acquire(lock) {
   while (lock->held);
   lock->held = 1;
}
void release(lock) {
  lock->held = 0;
}
```

← The caller *busy waits* for the lock to be released

Why doesn't this work ? Where is the race condition ?

Implementing Spinlocks

Problem is that the internals of the lock acquire/release have critical sections too !

- The acquire() and release() actions must be ***atomic***
- Atomic means that the code cannot be interrupted during execution

```
struct lock {
  int held = 0;
}
void acquire(lock) {
   while (lock->held);
   lock->held = 1;
}
void release(lock) {
  lock->held = 0;
}
```

What can happen if there is a context switch here?

Implementing Spinlocks

Problem is that the internals of the lock *acquire/release* have critical sections too.

Doing this requires help from the hardware:

- Atomic Instructions – CPU guarantees entire action will be atomic
  - Test and Set
  - Compare and Swap
- Disabling interrupts
  - Why does this guarantee atomicity ?

Spinlocks using Test and Set

CPU provides the following as an *atomic instruction*

```
bool test_and_set(bool *flag) {
  bool old = *flag;
  *flag = True;
  return old;
}
```

So, to fix our broken spinlock, we do this:

```
struct lock {
  int held = 0;
}
void acquire(lock) {
   while(test_and_set(&lock->held));
}
void release(lock) {
  lock->held = 0;
}
```

Atomic Operation:
test_and_set

What's the catch here ?

Spinlocks using Compare and Swap

```
void swap(bool *a, bool *b) {
        bool temp = *a;
        *a = *b;
        *b = temp;
}
```

```
struct lock {
        int held = 0;
}
void acquire(lock){
        key = TRUE;
        while(key == TRUE)
                swap(&lock->held,&key);
}
void release(lock) {
        lock ->held = 0;
}
```

Atomic Operation:
Compare and swap

Problems with Spinlocks

Horribly wasteful !

- Threads perform busy waiting to acquire locks
- Eats up a lot of CPU Cycles, slows down other threads
- What happens if you have a lot of threads trying to acquire locks

We only want spinlocks as ***primitives*** for building higher level synchronization constructs.

Alternatives to Spinlocks

Disabling Interrupts

```
struct lock {
  // Note - no state!
}
void acquire(lock) {
   cli();   // disable interrupts
}
void release(lock) {
   sti();   // reenable interupts
}
```

Can two threads disable or enable interrupts at the same time ?

What's wrong about this approach ?

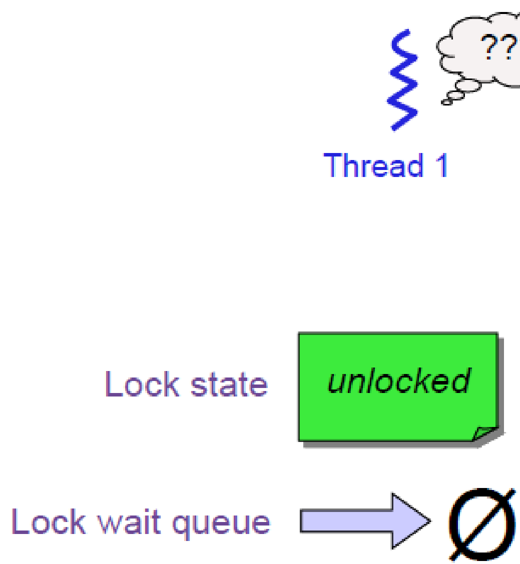
- Can only be implemented at the kernel level.(Why ?)
- Incorrect in multiprocessor system (Why ?)
- All locks in the system are mutually exclusive

Mutual Exclusion(Mutex) using Blocking Locks

Really want a thread ***waiting*** to enter the Critical Section to ***Block***

- Put the thread to sleep until it can enter the CS
- Free up the CPU for other threads to run

How to implement blocking Locks ? – **TCB Queues**



1. Check lock state

2. if(unlocked)

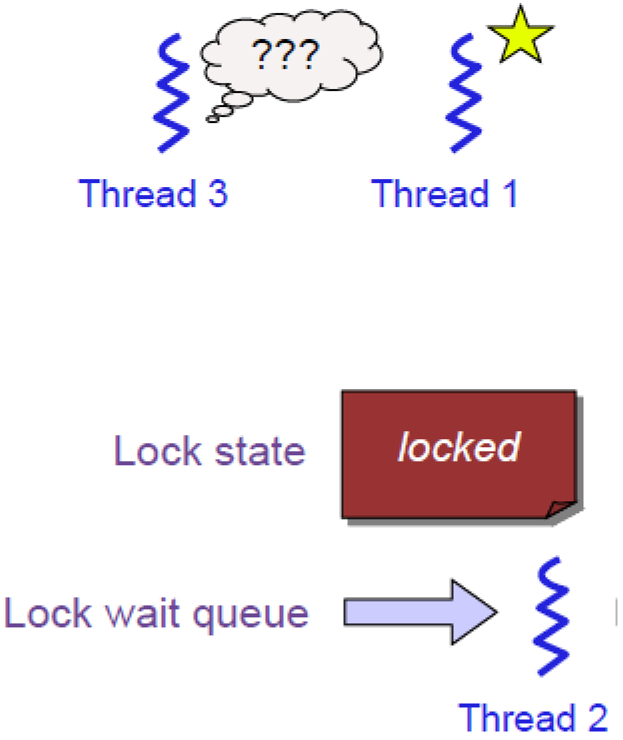
Set lock state to locked

Enter the CS Section

Mutex - Blocking Locks

1. Check lock state
2. If(locked)

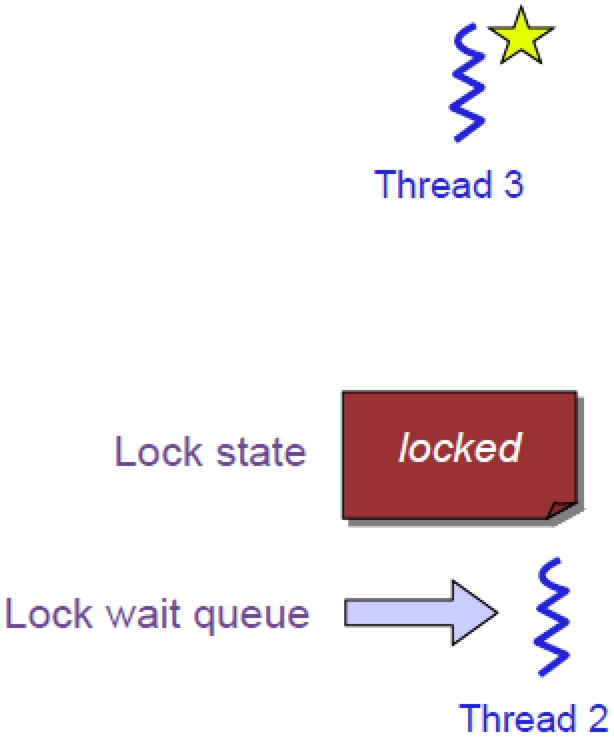
   Add self to wait queue (sleep)



Mutex – Blocking Locks

When a thread finished executing CS

1. Reset the lock to unlocked
2. Wake one thread from the wait-queue
3. Schedule it for execution of the CS



Limitations of Locks

Locks are simple. What can they NOT easily accomplish?

- atomicity without disabling interrupts or CPU support

What if there is a Data structure where its OK for many threads to read the data, but only one thread to write the data?

- Example: Bank Account
- Locks only let one thread access the data structure at a time

What if you want to protect access to two (or more) data structures at a time

e.g, Transferring money from one bank account to another?

Simple Approach: Use two separate locks for each account

What happens if you have to transfer from account A->B at the same time as transfer from account B->A ?

- We may end up in a DEADLOCK!!

Deadlock illustration