CSMC 412

Operating Systems

Prof. Ashok K Agrawala

Set 6

Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Monitors
- Java Synchronization
- Solaris Synchronization
- Windows XP Synchronization
- Linux Synchronization
- Pthreads Synchronization
- Atomic Transactions
- Log-based Recovery
- Checkpoints
- Concurrent Transactions
- Serializability
- Locking Protocols

Concurrency and Synchronization

- Concurrency
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Critical Regions
- Monitors
- Synchronization in Solaris 2 & Windows 2000

Systems = Objects + Activities

- Safety is a property of objects, and groups of objects, that participate across multiple activities.
  - Can be a concern at many different levels: objects, composites, components, subsystems, hosts, …
- Liveness is a property of activities, and groups of activities, that span across multiple objects.
  - Levels: Messages, call chains, threads, sessions, scenarios, scripts workflows, use cases, transactions, data flows, mobile computations, …

Violating Safety

- Data can be shared by threads
  - Scheduler can interleave or overlap threads arbitrarily
  - Can lead to *interference*
    - Storage corruption (e.g. a *data race*/*race condition)*
    - Violation of representation invariant
    - Violation of a protocol (e.g. *A* occurs before *B*)

How does this apply to OSs?

- Any resource that is shared could be accessed inappropriately
  - Shared memory
    - Kernel threads
    - Processes (shared memory set up by kernel)
  - Shared resources
    - Printer, Video screen, Network card, …
- OS must protect shared resources
  - And provide processes a means to protect their own abstractions

Data Race Example

```
static int cnt = 0;
t1.run() {
  int y = cnt;
  cnt = y + 1;
}
t2.run() {
  int y = cnt;
  cnt = y + 1;
}
```

Shared state cnt = 0

Start: both threads ready to run. Each will increment the global count.

Data Race Example

```
static int cnt = 0;
t1.run() {
  int y = cnt;
  cnt = y + 1;
}
t2.run() {
  int y = cnt;
  cnt = y + 1;
}
```

Shared state cnt = 0

y = 0

T1 executes, grabbing the global counter value into y.

Data Race Example

```
static int cnt = 0;
t1.run() {
  int y = cnt;
  cnt = y + 1;
}
t2.run() {
  int y = cnt;
  cnt = y + 1;
}
```

Shared state cnt = 0

y = 0

y = 0

T1 is pre-empted. T2 executes, grabbing the global counter value into y.

Data Race Example

```
static int cnt = 0;
t1.run() {
  int y = cnt;
  cnt = y + 1;
}
t2.run() {
  int y = cnt;
  cnt = y + 1;
}
```

Shared state **cnt = 1**

y = 0

y = 0

T2 executes, storing the incremented cnt value.

Data Race Example

```
static int cnt = 0;
t1.run() {
  int y = cnt;
  cnt = y + 1;
}
t2.run() {
  int y = cnt;
  cnt = y + 1;
}
```

Shared state **cnt = 1**

y = 0

y = 0

T2 completes. T1 executes again, storing the old counter value (1) rather than the new one (2)!

But When I Run it Again?

Data Race Example

```
static int cnt = 0;
t1.run() {
  int y = cnt;
  cnt = y + 1;
}
t2.run() {
  int y = cnt;
  cnt = y + 1;
}
```

Shared state cnt = 0

Start: both threads ready to run. Each will increment the global count.

Data Race Example

```
static int cnt = 0;
t1.run() {
  int y = cnt;
  cnt = y + 1;
}
t2.run() {
  int y = cnt;
  cnt = y + 1;
}
```

Shared state cnt = 0

y = 0

T1 executes, grabbing the global counter value into y.

Data Race Example

```
static int cnt = 0;
t1.run() {
  int y = cnt;
  cnt = y + 1;
}
t2.run() {
  int y = cnt;
  cnt = y + 1;
}
```

Shared state **cnt = 1**

y = 0

T1 executes again, storing the counter value

Data Race Example

```
static int cnt = 0;
t1.run() {
  int y = cnt;
  cnt = y + 1;
}
t2.run() {
  int y = cnt;
  cnt = y + 1;
}
```

Shared state cnt = 1

y = 0

y = 1

T1 finishes. T2 executes, grabbing the global counter value into y.

Data Race Example

```
static int cnt = 0;
t1.run() {
  int y = cnt;
  cnt = y + 1;
}
t2.run() {
  int y = cnt;
  cnt = y + 1;
}
```

Shared state **cnt = 2**

y = 0

y = 1

T2 executes, storing the incremented cnt value.

What happened?

- In the first example, **t1** was preempted after it read the counter but before it stored the new value.
  - Depends on the idea of an *atomic action*
  - Violated an object invariant
- A particular way in which the execution of two threads is interleaved is called a *schedule*. We want to prevent this undesirable schedule.
- Undesirable schedules can be hard to reproduce, and so hard to debug.

Question

- If you run a program with a race condition, will you always get an unexpected result?
  - No! It depends on the scheduler
  - ...and on the other threads/processes/etc that are running on the same CPU
- Race conditions are hard to find

Synchronization

```
static int cnt = 0;
struct Mutex lock;
Mutex_Init(&lock);
void run() {
  Mutex_Lock (&lock);
  int y = cnt;
  cnt = y + 1;
  Mutex_Unlock (&lock);
}
```

Lock*, for protecting*
The shared state

Acquires *the lock;*
Only succeeds if not held by another thread

Releases *the lock*

Java-style `synchronized` block

```
static int cnt = 0;
struct Mutex lock;
Mutex_Init(&lock);
void run() {
  synchronized (lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
```

Lock, *for protecting The shared state*

Acquires *the lock; Only succeeds if not held by another thread*

Releases *the lock*

Applying synchronization

```
int cnt = 0;
t1.run() {
  synchronized(lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
t2.run() {
  synchronized(lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
```

Shared state cnt = 0

T1 acquires the lock

Applying synchronization

```
int cnt = 0;
t1.run() {
  synchronized(lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
t2.run() {
  synchronized(lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
```

Shared state cnt = 0

y = 0

T1 reads cnt into y

Applying synchronization

```
int cnt = 0;
t1.run() {
  synchronized(lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
t2.run() {
  synchronized(lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
```

Shared state cnt = 0

y = 0

T1 is pre-empted.
T2 attempts to
acquire the lock but fails
because it's held by
T1, so it blocks

Applying synchronization

```
int cnt = 0;
t1.run() {
  synchronized(lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
t2.run() {
  synchronized(lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
```

Shared state **cnt = 1**

y = 0

T1 runs, assigning to cnt

Applying synchronization

```
int cnt = 0;
t1.run() {
  synchronized(lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
t2.run() {
  synchronized(lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
```

Shared state cnt = 1

y = 0

T1 releases the lock and terminates

Applying synchronization

```
int cnt = 0;
t1.run() {
  synchronized(lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
t2.run() {
  synchronized(lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
```

Shared state cnt = 1

y = 0

T2 now can acquire the lock.

Applying synchronization

```
int cnt = 0;
t1.run() {
  synchronized(lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
t2.run() {
  synchronized(lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
```

Shared state cnt = 1

y = 0

T2 reads cnt into y.

y = 1

Applying synchronization

```
int cnt = 0;
t1.run() {
  synchronized(lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
t2.run() {
  synchronized(lock) {
    int y = cnt;
    cnt = y + 1;
  }
}
```

Shared state **cnt = 2**

y = 0

T2 assigns cnt,
then releases the lock

y = 1

Mutexes (locks)

- Only one thread can "acquire" a mutex
  - Other threads block until they can acquire it
  - Used for implementing critical sections
- A critical section is a piece of code that should not be interleaved with code from another thread
  - Executed atomically
- We'll look at other ways to implement critical sections later …

Mutex Policies

- What if a thread already holds the mutex it's trying to acquire?
  - Re-entrant mutexes: The thread can reacquire the same lock many times. Lock is released when object unlocked the corresponding number of times
    - This is the case for Java
  - Non-reentrant: Deadlock! (defined soon.)
    - This is the case in GeekOS
- What happens if a thread is killed while holding a mutex? Or if it just forgets to release it
  - Could lead to deadlock

Java Synchronized statement

- **synchronized (obj) {** *statements* **}**
- Obtains the lock on **obj** before executing statements in block
  - **obj** can be any Object
- Releases the lock when the statement block completes
  - Either normally, or due to a return, break, or exception being thrown in the block
- Can't forget to release the lock!

Synchronization not a Panacea

- Two threads can block on locks held by the other; this is called *deadlock*

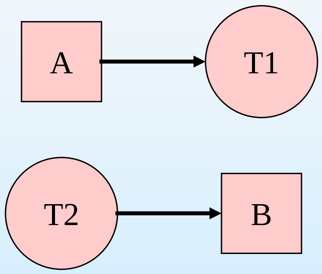
```
Object A = new Object();
Object B = new Object();
T1.run() {
  synchronized (A) {
    synchronized (B) {
      …
    }
  }
}
```

```
T2.run() {
  synchronized (B) {
    synchronized (A) {
      …
    }
  }
}
```


Deadlock

- Quite possible to create code that deadlocks
  - Thread 1 holds lock on **A**
  - Thread 2 holds lock on **B**
  - Thread 1 is trying to acquire a lock on **B**
  - Thread 2 is trying to acquire a lock on **A**
  - Deadlock!
- Not easy to detect when deadlock has occurred
  - other than by the fact that nothing is happening

Deadlock: Wait graphs

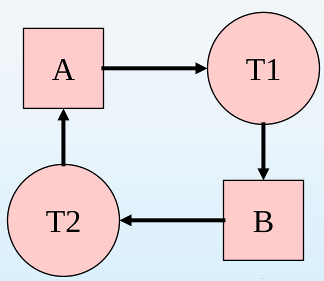


Thread T1 holds lock A

Thread T2 attempting to acquire lock B

Deadlock occurs when there is a cycle in the graph

Wait graph example



T1 holds lock on **A**
T2 holds lock on **B**
T1 is trying to acquire a lock on **B**
T2 is trying to acquire a lock on **A**

Key Ideas

- Multiple threads can run simultaneously
  - Either truly in parallel on a multiprocessor
  - Or can be scheduled on a single processor
    - A running thread can be pre-empted at any time

- Threads can share data
  - Need to prevent interference
    - Synchronization is one way, but not the only way
  - Overuse use of synchronization can create deadlock
    - Violation of liveness

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Shared-memory solution to bounded-butter problem (Chapter 4) has a race condition on the class data **count**.

Race Condition

The Producer calls

```
while (1) {
        while (count == BUFFER_SIZE)
                ; // do nothing
        // produce an item and put in nextProduced
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```


Race Condition

The Consumer calls

```
while (1) {
        while (count == 0)
                ; // do nothing
        nextConsumed =  buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        // consume the item in nextConsumed
}
```

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:

  S0: producer execute register1 = count {register1 = 5}
  S1: producer execute register1 = register1 + 1 {register1 = 6}
  S2: consumer execute register2 = count {register2 = 5}
  S3: consumer execute register2 = register2 - 1 {register2 = 4}
  S4: producer execute count = register1 {count = 6 }
  S5: consumer execute count = register2 {count = 4}

The Critical-Section Problem

- *n* processes all competing to use some shared data
- Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

Solution to Critical-Section Problem

1. Mutual Exclusion - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the N processes

Two-task Solution

- Two tasks, T_0 and T_1 (T_i and T_j)
- Three solutions presented. All implement this MutualExclusion interface:

```
public interface MutualExclusion
{
        public static final int TURN 0 = 0;
        public static final int TURN 1 = 1;

        public abstract void enteringCriticalSection(int turn);
        public asbtract void leavingCriticalSection(int turn);
}
```

Algorithm Factory class

Used to create two threads and to test each algorithm

```
public class AlgorithmFactory
{
    public static void main(String args[]) {
      MutualExclusion alg = new Algorithm 1();
      Thread first = new Thread( new Worker("Worker 0", 0, alg));
      Thread second = new Thread(new Worker("Worker 1", 1,
    alg));

      first.start();
      second.start();
    }
}
```


Worker Thread

```
public class Worker implements Runnable
{
    private String name;
    private int id;
    private MutualExclusion mutex;

    public Worker(String name, int id, MutualExclusion mutex) {
        this.name = name;
        this.id = id;
        this.mutex = mutex;
    }
    public void run() {
        while (true) {
            mutex.enteringCriticalSection(id);
            MutualExclusionUtilities.criticalSection(name);
            mutex.leavingCriticalSection(id);
            MutualExclusionUtilities.nonCriticalSection(name);
        }
    }
}
```

Algorithm 1

- Threads share a common integer variable turn
- If turn==i, thread i is allowed to execute
- Does not satisfy progress requirement
  - Why?

Algorithm 1

```
public class Algorithm_1 implements MutualExclusion
{
    private volatile int turn;

    public Algorithm 1() {
        turn = TURN 0;
    }
    public void enteringCriticalSection(int t) {
        while (turn != t)
            Thread.yield();
    }
    public void leavingCriticalSection(int t) {
        turn = 1 - t;
    }
}
```

Algorithm 1

- Satisfies mutual exclusion but not progress.
  - Processes are forced to enter their critical sections alternately.
  - One process not in its critical section thus prevents the other from entering its critical section.

Algorithm 2

- Add more state information
  - Boolean flags to indicate thread's interest in entering critical section
- Progress requirement still not met
  - Why?

Algorithm 2

```
public class Algorithm_2 implements MutualExclusion
{
      private volatile boolean flag0, flag1;
      public Algorithm 2() {
                flag0 = false; flag1 = false;
      }
      public void enteringCriticalSection(int t) {
                 if (t == 0) {
                              flag0 = true;
                              while(flag1 == true)
                                              Thread.yield();
                }
                else {
                               flag1 = true;
                              while (flag0 == true)
                                              Thread.yield();
                }
      }
      // Continued On Next Slide
```


Algorithm 2 - cont

```
   public void leavingCriticalSection(int t) {
                   if (t == 0)
                                 flag0 = false;
           else
                                 flag1 = false;
   }
}
```

Algorithm 2

- Satisfies mutual exclusion, but not progress requirement.
  - Both processes can end up setting their flag[] variable to true, and thus neither process enters its critical section!

Algorithm 3

- Combine ideas from 1 and 2
- Does it meet critical section requirements?

Algorithm 3

```
public class Algorithm_3 implements MutualExclusion
{
    private volatile boolean flag0;
    private volatile boolean flag1;
    private volatile int turn;
    public Algorithm_3() {
            flag0 = false;
            flag1 = false;
            turn = TURN_0;
    }
    // Continued on Next Slide
```


Algorithm 3 - enteringCriticalSection

```
public void enteringCriticalSection(int t) {
        int other = 1 - t;
        turn = other;
        if (t == 0) {
                flag0 = true;
                while(flag1 == true && turn == other)
                        Thread.yield();
        }
        else {
                flag1 = true;
                while (flag0 == true && turn == other)
                        Thread.yield();
        }
}
// Continued on Next Slide
```

Algo. 3 – leavingingCriticalSection()

```
public void leavingCriticalSection(int t) {
        if (t == 0)
                flag0 = false;
    else
                flag1 = false;
 }
}
```


Algorithm 3

- Meets all three requirements; solves the critical-section problem for two processes.
  - One process is always guaranteed to get into its critical section.
  - Processes are forced to take turns when they both want to get in.

Bakery Algorithm

Critical section for n processes

- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.
- If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first.
- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...

Bakery Algorithm

- Notation <≡ lexicographical order (ticket #, process id #)
  - $(a,b) < c,d)$ if $a < c$ or if $a = c$ and $b < d$
  - max $(a_0,\ldots, a_{n-1})$ is a number, k, such that $k \geq a_i$ for i - 0, …, $n - 1$
- Shared data

  boolean choosing[n];

  int number[n];

Data structures are initialized to **false** and **0** respectively

Bakery Algorithm

```
do {
    choosing[i] = true;
    number[i] = max(number[0], number[1], …, number [n – 1])+1;
    choosing[i] = false;
    for (j = 0; j < n; j++) {
            while (choosing[j]) ;
            while ((number[j] != 0) && (number[j,j] < number[i,i])) ;
    }
      critical section
    number[i] = 0;
      remainder section
} while (1);
```


Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words

Disabling Interrupts

- Doesn’t work for multiprocessors
- Doesn’t permit different groups of critical sections

Synchronization Hardware

- Test and modify the content of a word atomically

.

```
boolean TestAndSet(boolean &target) {
    boolean rv = target;
    tqrget = true;

    return rv;
}
```

Data Structure for Hardware Solutions

```
public class HardwareData
{
    private boolean data;
    public HardwareData(boolean data) {
                this.data = data;
    }
    public boolean get() {
                return data;
    }
    public void set(boolean data) {
                this.data = data;
    }
    // Continued on Next Slide
```


Data Structure for Hardware Solutions - cont

```
public boolean getAndSet(boolean data) {
                boolean oldValue = this.get();
                this.set(data);
                return oldValue;
}
public void swap(HardwareData other) {
                boolean temp = this.get();
                this.set(other.get());
                other.set(temp);
}
}
```

Thread Using get-and-set Lock

```
// lock is shared by all threads
HardwareData lock = new HardwareData(false);
while (true) {
    while (lock.getAndSet(true))
                Thread.yield();
    criticalSection();
    lock.set(false);
    nonCriticalSection();
}
```


Thread Using swap Instruction

```
// lock is shared by all threads
HardwareData lock = new HardwareData(false);
// each thread has a local copy of key
HardwareData key = new HardwareData(true);

    while (true) {
                key.set(true);
                do {
                        lock.swap(key);
                }
                while (key.get() == true);
                criticalSection();
                lock.set(false);
                nonCriticalSection();
    }
```

Semaphore

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

```
acquire(S) {
   while S <= 0
                ; // no-op
   S--;
}
release(S) {
   S++;
}
```


Information Implications of Semaphore

- A process has synch points
  - To go past a synch point certain conditions must be true
    - Conditions depend not only on ME but other processes also
    - Have to confirm that the conditions are true before proceeding, else have to wait.
- P(S) – Wait (S)
  - If can complete this operation
    - Inform others through changed value of S
    - Proceed past the synch point
  - If can not complete
    - Wait for the event when S becomes >0
- V(S) – Signal (S)
  - Inform others that I have gone past a synch point.

Semaphore as General Synchronization Tool

- 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
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion

```
Semaphore S; // initialized to 1

acquire(S);
criticalSection();
release(S);
```


Synchronization using Semaphores Implementation - Worker

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

        MutualExclusionUtilities.nonCriticalSection(name);
            }
    }
}
```

Synchronization using Semaphores Implementation - SemaphoreFactory

```
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, "Worker " + (new Integer(i)).toString()
    ));
                for (int i = 0; i < 5; i++)
                        bees[i].start();
        }
}
```


Semaphore Implementation

```
acquire(S){
    value--;
    if (value < 0) {
                add this process to list
                block;
    }
}
release(S){
    value++;
    if (value <= 0) {
                remove a process P from list
                wakeup(P);
    }
}
```

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
  - Could now have busy waiting in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
  - Applications may spend lots of time in critical sections
    - Performance issues addressed throughout this lecture

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

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

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

Two Types of Semaphores

- *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.
- Can implement a counting semaphore *S* as a binary semaphore.

Implementing *S* as a Binary Semaphore

- Data structures:

  binary-semaphore S1, S2;
  int C:

- Initialization:

  S1 = 1
  S2 = 0
  C = initial value of semaphore **S**

Implementing *S*

- *wait* operation

```
wait(S1);
C--;
if (C < 0) {
        signal(S1);
        wait(S2);
}
signal(S1);
```

- *signal* operation

```
wait(S1);
C ++;
if (C <= 0)
    signal(S2);
else
    signal(S1);
```


Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

```
public class BoundedBuffer implements Buffer
{
    private static final int BUFFER SIZE = 5;
    private Object[] buffer;
    private int in, out;
    private Semaphore mutex;
    private Semaphore empty;
    private Semaphore full;

    // Continued on next Slide
```


Bounded Buffer Constructor

```
public BoundedBuffer() {
            // buffer is initially empty
            in = 0;
            out = 0;
            buffer = new Object[BUFFER SIZE];
            mutex = new Semaphore(1);
            empty = new Semaphore(BUFFER SIZE);
            full = new Semaphore(0);
}
public void insert(Object item) { /* next slides */ }

public Object remove() { /* next slides */ }
}
```

Bounded Buffer Problem: insert() Method

```
public void insert(Object item) {
    empty.acquire();
    mutex.acquire();
    // add an item to the buffer
    buffer[in] = item;
    in = (in + 1) % BUFFER SIZE;
    mutex.release();
    full.release();
}
```


Bounded Buffer Problem: remove() Method

```
public Object remove() {
    full.acquire();
    mutex.acquire();
    // remove an item from the buffer
    Object item = buffer[out];
    out = (out + 1) % BUFFER SIZE;
    mutex.release();
    empty.release();
    return item;
}
```

Bounded Buffer Problem: Producer

```
import java.util.Date;
public class Producer implements Runnable
{
    private Buffer buffer;
    public Producer(Buffer buffer) {
                    this.buffer = buffer;
    }
    public void run() {
                    Date message;
                    while (true) {
                                // nap for awhile
                                SleepUtilities.nap();
                                // produce an item & enter it into the buffer
                                message = new Date();
                                buffer.insert(message);
                    }
    }
}
```


Bounded Buffer Problem: Consumer

```
import java.util.Date;
public class Consumer implements Runnable
{
    private Buffer buffer;
    public Consumer(Buffer buffer) {
                    this.buffer = buffer;
    }
    public void run() {
                    Date message;
                    while (true) {
                                // nap for awhile
                                SleepUtilities.nap();
                                // consume an item from the buffer
                                message = (Date)buffer.remove();
                    }
    }
}
```

Bounded Buffer Problem: Factory

```
public class Factory
{
    public static void main(String args[]) {
        Buffer buffer = new BoundedBuffer();
        // now create the producer and consumer threads
        Thread producer = new Thread(new Producer(buffer));
        Thread consumer = new Thread(new Consumer(buffer));
        producer.start();
        consumer.start();
    }
}
```


Readers-Writers Problem: Reader

```
public class Reader implements Runnable
{
    private RWLock db;
    public Reader(RWLock db) {
        this.db = db;
    }
    public void run() {
        while (true) { // nap for awhile
            db.acquireReadLock();

            // you now have access to read from the database
            // read from the database

            db.releaseReadLock();
        }
    }
}
```

Readers-Writers Problem: Writer

```
public class Writer implements Runnable
{
    private RWLock db;
    public Writer(RWLock db) {
            this.db = db;
    }
    public void run() {
            while (true) {
                    db.acquireWriteLock();
                    // you have access to write to the database

                    // write to the database

                    db.releaseWriteLock();
            }
    }
}
```


Readers-Writers Problem: Interface

```
public interface RWLock
{
    public abstract void acquireReadLock();
    public abstract void acquireWriteLock();
    public abstract void releaseReadLock();
    public abstract void releaseWriteLock();
}
```

Readers-Writers Problem: Database

```
public class Database implements RWLock
{
     private int readerCount;
     private Semaphore mutex;
     private Semaphore db;
     public Database() {
                    readerCount = 0;
                    mutex = new Semaphore(1);
                    db = new Semaphore(1);
     }
     public int acquireReadLock() { /* next slides */ }
     public int releaseReadLock() {/* next slides */ }
     public void acquireWriteLock() {/* next slides */ }
     public void releaseWriteLock() {/* next slides */ }
}
```


Readers-Writers Problem: Methods called by readers

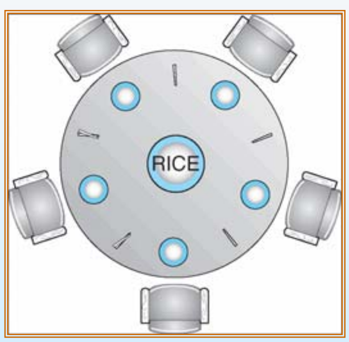
```
public void acquireReadLock() {
     mutex.acquire();
     ++readerCount;
     // if I am the first reader tell all others
     // that the database is being read
     if (readerCount == 1)
                    db.acquire();
     mutex.release();
}
public void releaseReadLock() {
     mutex.acquire();
     --readerCount;
     // if I am the last reader tell all others
     // that the database is no longer being read
     if (readerCount == 0)
                    db.release();
     mutex.release();
}
```

Readers-Writers Problem: Methods called by writers

```
public void acquireWriteLock() {
    db.acquire();
}

public void releaseWriteLock() {
    db.release();
}
```


Dining-Philosophers Problem



- Shared data

```
Semaphore chopStick[]  = new Semaphore[5];
```

Dining-Philosophers Problem (Cont.)

- Philosopher *i*:

```
while (true) {
    // get left chopstick
    chopStick[i].acquire();
    // get right chopstick
    chopStick[(i + 1) % 5].acquire();
    eating();
    // return left chopstick
    chopStick[i].release();
    // return right chopstick
    chopStick[(i + 1) % 5].release();
    thinking();
}
```


Monitors

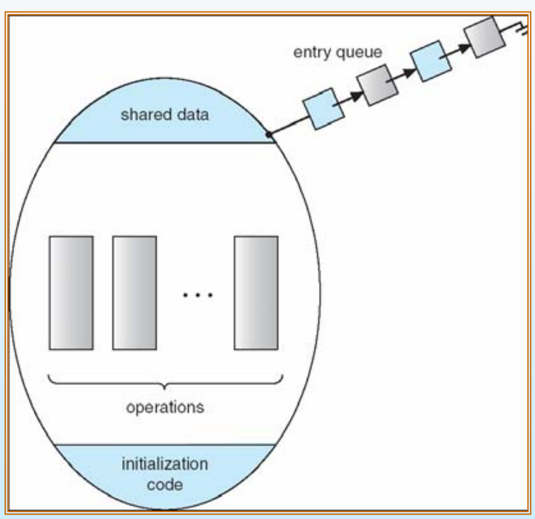
- A monitor is a high-level abstraction that provides thread safety
- Only one thread may be active within the monitor at a time

```
monitor monitor-name
{
   // variable declarations
   public entry p1(…) {
          …
   }
   public entry p2(…) {
          …
   }
}
```

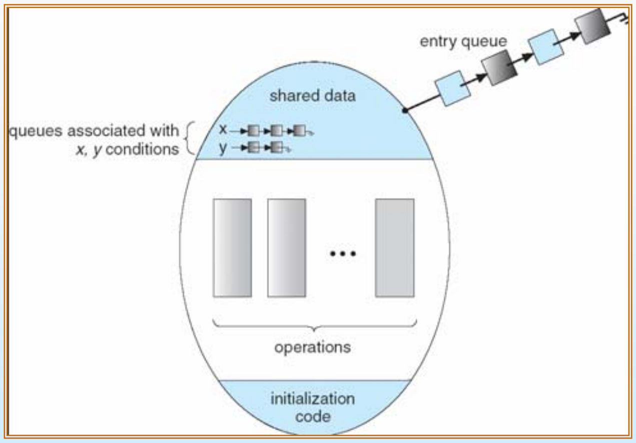
Condition Variables

- condition x, y;
- A thread that invokes x.wait is suspended until another thread invokes x.signal

Monitor with condition variables



Monitor with Condition Variables



Condition Variable Solution to Dining Philosophers

```
monitor DiningPhilosophers {
      int[] state = new int[5];
      static final int THINKING = 0;
      static final int HUNGRY = 1;
      static final int EATING = 2;
      condition[] self = new condition[5];
      public diningPhilosophers {
                          for (int i = 0; i < 5; i++)
                                   state[i] = THINKING;
            }
      public entry pickUp(int i) {
                          state[i] = HUNGRY;
                          test(i);
                          if (state[i] != EATING)
                                   self[i].wait;
      }
      // Continued on Next Slide
```

Solution to Dining Philosophers (cont)

```
public entry putDown(int i) {
        state[i] = THINKING;
        // test left and right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
}
private test(int i) {
        if ( (state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING) ) {
                state[i] = EATING;
                self[i].signal;
        }
}
```


Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next-count = 0;
```

- Each external procedure ***F*** will be replaced by

```
wait(mutex);
   …
 body of F;
   …
if (next-count > 0)
  signal(next)
else
  signal(mutex);
```

- Mutual exclusion within a monitor is ensured.

Monitor Implementation

- For each condition variable ***x***, we have:

```
semaphore x-sem; // (initially  = 0)
int x-count = 0;
```

- The operation **x.wait** can be implemented as:

```
x-count++;
if (next-count > 0)
      signal(next);
else
      signal(mutex);
wait(x-sem);
x-count--;
```


Monitor Implementation

- The operation **x.signal** can be implemented as:

```
if (x-count > 0) {
     next-count++;
     signal(x-sem);
     wait(next);
     next-count--;
}
```

Monitor Implementation

- *Conditional-wait* construct: **x.wait(c);**
  - **c** – integer expression evaluated when the **wait** operation is executed.
  - value of **c** (a *priority number*) stored with the name of the process that is suspended.
  - when **x.signal** is executed, process with smallest associated priority number is resumed next.
- Check two conditions to establish correctness of system:
  - User processes must always make their calls on the monitor in a correct sequence.
  - Must ensure that an uncooperative process does not ignore the mutual-exclusion gateway provided by the monitor, and try to access the shared resource directly, without using the access protocols.

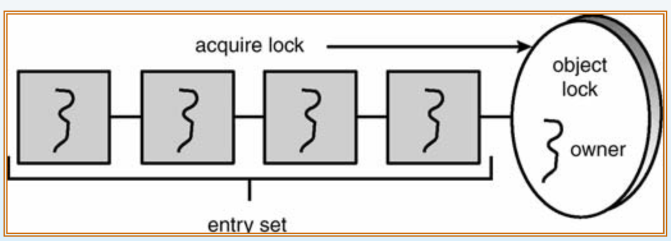
Java Synchronization

- Bounded Buffer solution using synchronized, wait(), notify() statements
- Multiple Notifications
- Block Synchronization
- Java Semaphores
- Java Monitors

Synchronized Statement

- Every object has a lock associated with it
- Calling a synchronized method requires "owning" the lock
- If a calling thread does not own the lock (another thread already owns it), the calling thread is placed in the wait set for the object's lock
- The lock is released when a thread exits the synchronized method

Entry Set



Synchronized Insert() Method

```
public synchronized void insert(Object item) {
    while (count == BUFFER SIZE)
            Thread.yield();
    ++count;
    buffer[in] = item;
    in = (in + 1) % BUFFER SIZE;
}
```

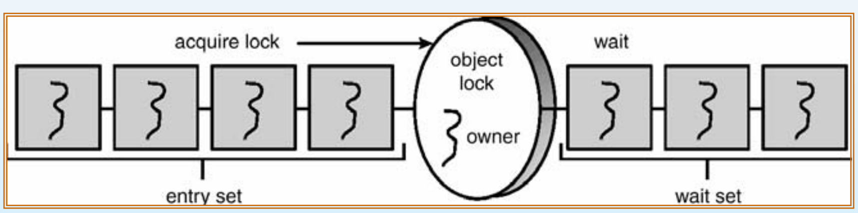

synchronized remove() Method

```
public synchronized Object remove() {
    Object item;
    while (count == 0)
            Thread.yield();
    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER SIZE;
    return item;
}
```

The wait() Method

- When a thread calls wait(), the following occurs:
  1. the thread releases the object lock
  2. thread state is set to blocked
  3. thread is placed in the wait set

Entry and Wait Sets



The notify() Method

When a thread calls notify(), the following occurs:

1. selects an arbitrary thread *T* from the wait set
2. moves *T* to the entry set
3. sets *T* to Runnable

T can now compete for the object's lock again

insert() with wait/notify Methods

```
public synchronized void insert(Object item) {
    while (count == BUFFER SIZE) {
                    try {
                              wait();
                    }
                    catch (InterruptedException e) { }
    }
    ++count;
    buffer[in] = item;
    in = (in + 1) % BUFFER SIZE;
    notify();
}
```

remove() with wait/notify Methods

```
public synchronized Object remove() {
    Object item;
    while (count == 0) {
                    try {
                        wait();
                    }
                    catch (InterruptedException e) { }
    }
    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER SIZE;
    notify();
    return item;
}
```


Complete Bounded Buffer using Java Synchronization

```
public class BoundedBuffer implements Buffer
{
    private static final int BUFFER SIZE = 5;
    private int count, in, out;
    private Object[] buffer;
    public BoundedBuffer() { // buffer is initially empty
                    count = 0;
                    in = 0;
                    out = 0;
                    buffer = new Object[BUFFER SIZE];
    }
    public synchronized void insert(Object item) { // See previous slides
    }
    public synchronized Object remove() { // See previous slides
    }
}
```

Multiple Notifications

- notify() selects an arbitrary thread from the wait set.
  *This may not be the thread that you want to be selected.
- Java does not allow you to specify the thread to be selected
- notifyAll() removes ALL threads from the wait set and places them in the entry set. This allows the threads to decide among themselves who should proceed next.
- notifyAll() is a conservative strategy that works best when multiple threads may be in the wait set

Reader Methods with Java Synchronization

```
public class Database implements RWLock {
    private int readerCount;
    private boolean dbWriting;
    public Database() {
                readerCount = 0;
                dbWriting = false;
    }
    public synchronized void acquireReadLock() { // see next slides
    }
    public synchronized void releaseReadLock() { // see next slides
    }
    public synchronized void acquireWriteLock() { // see next slides
    }
    public synchronized void releaseWriteLock() { // see next slides
    }
}
```

acquireReadLock() Method

```
public synchronized void acquireReadLock() {
    while (dbWriting == true) {
            try {
                wait();
            }
            catch(InterruptedException e) { }
    }
    ++readerCount;
}
```


releaseReadLock() Method

```
public synchronized void releaseReadLock() {
    --readerCount;
    // if I am the last reader tell writers
    // that the database is no longer being read
    if (readerCount == 0)
            notify();
}
```

Writer Methods

```
public synchronized void acquireWriteLock() {
    while (readerCount > 0 || dbWriting == true) {
            try {
                wait();
            }
            catch(InterruptedException e) { }
    }
    // once there are either no readers or writers
    // indicate that the database is being written
    dbWriting = true;
}
public synchronized void releaseWriteLock() {
    dbWriting = false;
    notifyAll();
}
```


Block Synchronization

- Scope of lock is time between lock acquire and release
- Blocks of code – rather than entire methods – may be declared as **synchronized**
- This yields a lock scope that is typically smaller than a synchronized method

Block Synchronization (cont)

```
Object mutexLock = new Object();
. . .
public void someMethod() {
    nonCriticalSection();
    synchronized(mutexLock) {
                    criticalSection();
    }
    nonCriticalSection();
}
```


Java Semaphores

- Java does not provide a semaphore, but a basic semaphore can be constructed using Java synchronization mechanism

Semaphore Class

```
public class Semaphore
{
    private int value;
    public Semaphore() {
            value = 0;
    }
    public Semaphore(int value) {
            this.value = value;
    }
```


Semaphore Class (cont)

```
public synchronized void acquire() {
            while (value == 0)
                try {
                    wait();
                } catch (InterruptedException ie) { }
            value--;
        }
    public synchronized void release() {
            ++value;
            notify();
        }
}
```

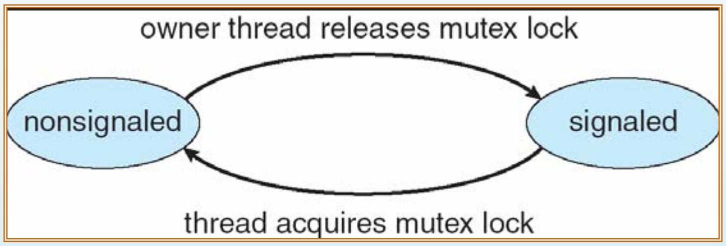
Syncronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads

Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses adaptive mutexes for efficiency when protecting data from short code segments
- Uses condition variables and readers-writers locks when longer sections of code need access to data
- Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock

Mutex Dispatcher Object



Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses spinlocks on multiprocessor systems
- Also provides dispatcher objects which may act as either mutexes and semaphores
- Dispatcher objects may also provide events
  - An event acts much like a condition variable

Linux Synchronization

- Linux:
  - disables interrupts to implement short critical sections
- Linux provides:
  - semaphores
  - spin locks

Synchronization in Linux

single processor	multiple processors
Disable kernel preemption.	Acquire spin lock.
Enable kernel preemption.	Release spin lock.

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
- Non-portable extensions include:
  - read-write locks
  - spin locks

Atomic Transactions

- Assures that operations happen as single logical unit of work, in its entirety, or not at all
- Related to field of database systems
- Challenge is assuring atomicity despite computer system failures
- Transaction – collection of instructions or operations that performs single logical function
  - Here we are concerned with changes to stable storage – disk
  - Transaction is series of read and write operations
  - Terminated by commit (transaction successful) or abort (Transaction failed operation
  - Aborted transaction must be rolled back to undo any changes it performed.