

CISC 7310X
C09c Process
Synchronization: OS
Examples

Hui Chen

Department of Computer & Information Science

CUNY Brooklyn College

Acknowledgement

- These slides are a revision of the slides provided by the authors of the textbook via the publisher of the textbook

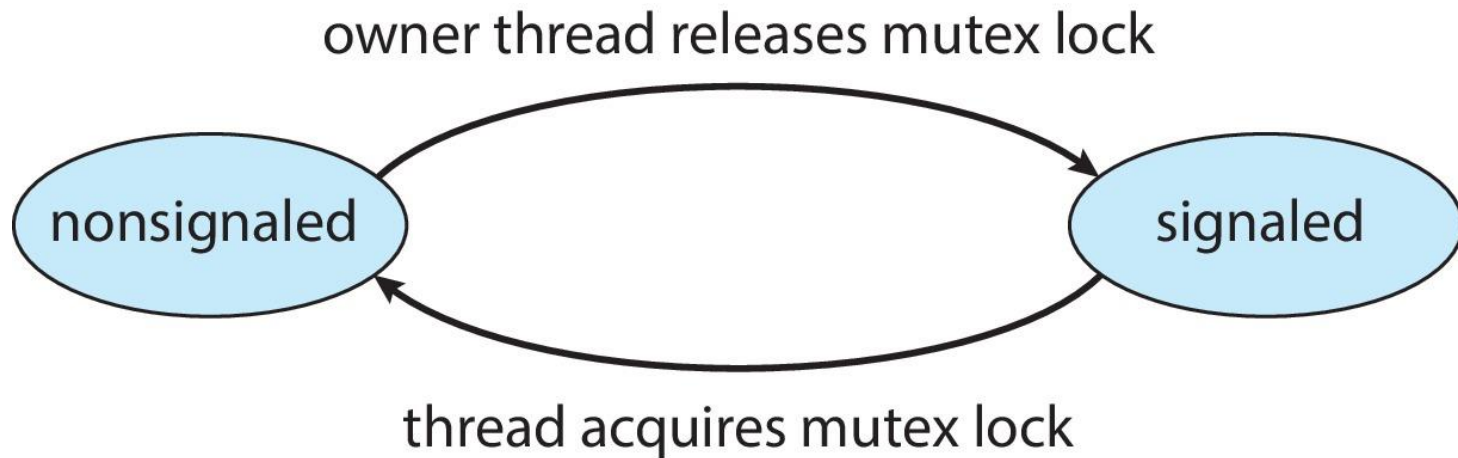
Outline

- Windows
- Linux
- POSIX
- Java
- Alternative Approaches

Kernel Synchronization - Windows

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
 - Spinlocking-thread will never be preempted
- Also provides **dispatcher objects** user-land which may act mutexes, semaphores, events, and timers
 - **Events**
 - An event acts much like a condition variable
 - Timers notify one or more thread when time expired
 - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)

- **Mutex dispatcher object**



Linux Synchronization

- Linux:
 - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
 - Version 2.6 and later, fully preemptive
- Linux provides:
 - Semaphores
 - atomic integers
 - spinlocks
 - reader-writer versions of both
- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

Atomic Operations

- Atomic variables

`atomic_t` is the type for atomic integer

- Consider the variables

- `atomic_t counter;`
`int value;`

<i>Atomic Operation</i>	<i>Effect</i>
<code>atomic_set(&counter,5);</code>	<code>counter = 5</code>
<code>atomic_add(10,&counter);</code>	<code>counter = counter + 10</code>
<code>atomic_sub(4,&counter);</code>	<code>counter = counter - 4</code>
<code>atomic_inc(&counter);</code>	<code>counter = counter + 1</code>
<code>value = atomic_read(&counter);</code>	<code>value = 12</code>

POSIX Synchronization

- POSIX API provides
 - mutex locks
 - semaphores
 - condition variable
- Widely used on UNIX, Linux, and macOS

POSIX Mutex Locks

- Creating and initializing the lock

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex, NULL);
```

- Acquiring and releasing the lock

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

POSIX Semaphores

- POSIX provides two versions - **named** and **unnamed**.
- Named semaphores can be used by unrelated processes, unnamed cannot.

POSIX Named Semaphores

- Creating and initializing the semaphore:

```
#include <semaphore.h>
sem_t *sem;

/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
```

Another process can access the semaphore by referring to its name **SEM**.

- Acquiring and releasing the semaphore:

```
/* acquire the semaphore */
sem_wait(sem);

/* critical section */

/* release the semaphore */
sem_post(sem);
```

POSIX Unnamed Semaphores

- Creating and initializing the semaphore:

```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
```

- Acquiring and releasing the semaphore:

```
/* acquire the semaphore */
sem_wait(&sem);

/* critical section */

/* release the semaphore */
sem_post(&sem);
```

POSIX Condition Variables

- Since POSIX is typically used in C/C++ and these languages do not provide a monitor, POSIX condition variables are associated with a POSIX mutex lock to provide mutual exclusion:
- Creating and initializing the condition variable:

```
pthread_mutex_t mutex;  
pthread_cond_t cond_var;
```

```
pthread_mutex_init(&mutex, NULL);  
pthread_cond_init(&cond_var, NULL);
```

POSIX Condition Variables

- Thread waiting for the condition `a == b` to become true:

```
pthread_mutex_lock(&mutex);  
while (a != b)  
    pthread_cond_wait(&cond_var, &mutex);  
  
pthread_mutex_unlock(&mutex);
```

- Thread signaling another thread waiting on the condition variable:

```
pthread_mutex_lock(&mutex);  
a = b;  
pthread_cond_signal(&cond_var);  
pthread_mutex_unlock(&mutex);
```

Java Synchronization

- Java provides rich set of synchronization features:
 - Java monitors
 - Reentrant locks
 - Semaphores
 - Condition variables

Java Monitors

- Every Java object has associated with it a single lock.
- If a method is declared as `synchronized`, a calling thread must own the lock for the object.
- If the lock is owned by another thread, the calling thread must wait for the lock until it is released.
- Locks are released when the owning thread exits the `synchronized` method.

Bounded Buffer - Java Synchronization

```

public class BoundedBuffer<E>
{
    private static final int BUFFER_SIZE = 5;

    private int count, in, out;
    private E[] buffer;

    public BoundedBuffer() {
        count = 0;
        in = 0;
        out = 0;
        buffer = (E[]) new Object[BUFFER_SIZE];
    }

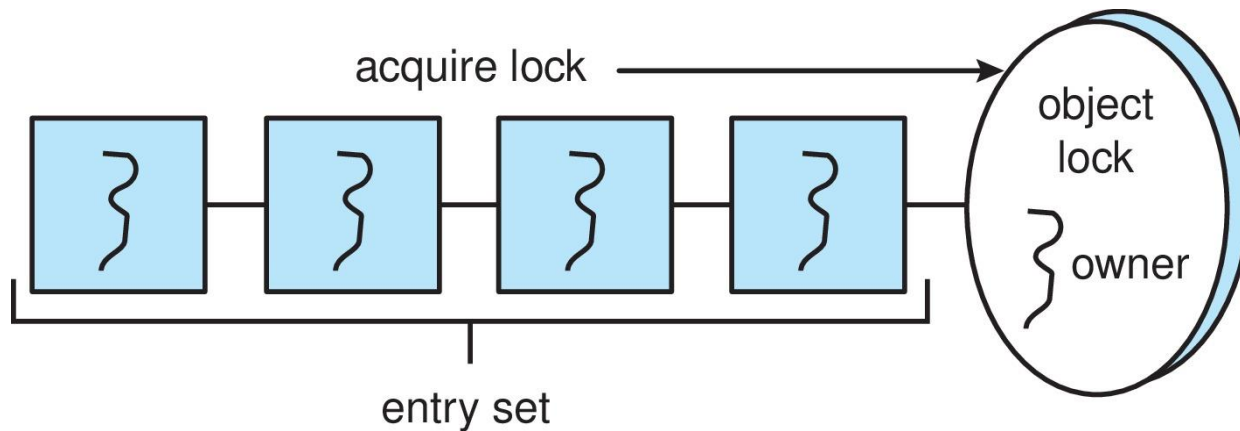
    /* Producers call this method */
    public synchronized void insert(E item) {
        /* See Figure 7.11 */
    }

    /* Consumers call this method */
    public synchronized E remove() {
        /* See Figure 7.11 */
    }
}

```

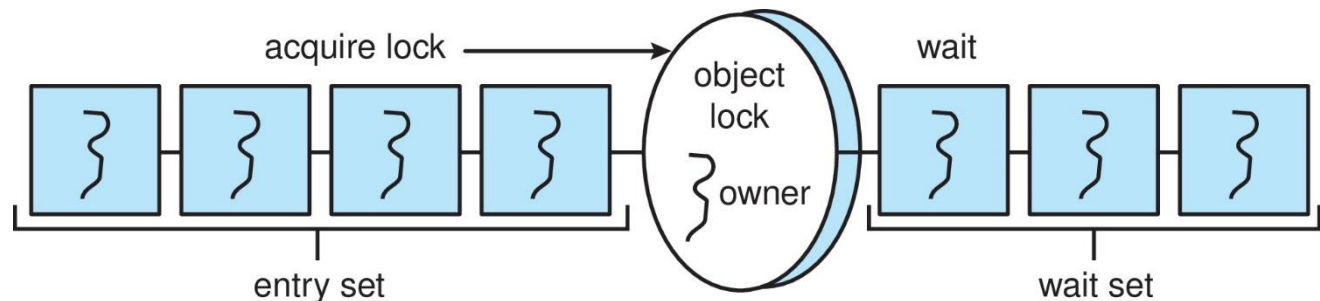
Java Synchronization

- A thread that tries to acquire an unavailable lock is placed in the object's **entry set**:



Java Synchronization

- Similarly, each object also has a wait set.
- When a thread calls `wait()`:
- It releases the lock for the object
- The state of the thread is set to blocked
- The thread is placed in the wait set for the object



Java Synchronization

- A thread typically calls `wait()` when it is waiting for a condition to become true.
 - How does a thread get notified?
 - When a thread calls `notify()`:
- An arbitrary thread T is selected from the wait set
 1. T is moved from the wait set to the entry set
 2. Set the state of T from blocked to runnable.
 3. T can now compete for the lock to check if the condition it was waiting for is now true.

Bounded Buffer - Java Synchronization

```
/* Producers call this method */
public synchronized void insert(E item) {
    while (count == BUFFER_SIZE) {
        try {
            wait();
        }
        catch (InterruptedException ie) { }
    }

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

    notify();
}
```

Bounded Buffer - Java Synchronization

```
/* Consumers call this method */
public synchronized E remove() {
    E item;

    while (count == 0) {
        try {
            wait();
        }
        catch (InterruptedException ie) { }
    }

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

    notify();

    return item;
}
```


Java Reentrant Locks

- Similar to mutex locks
- The `finally` clause ensures the lock will be released in case an exception occurs in the `try` block.

```
Lock key = new ReentrantLock();

key.lock();
try {
    /* critical section */
}
finally {
    key.unlock();
}
```

Java Semaphores

- Constructor:

```
Semaphore(int value);
```

- Usage:

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

try {
    sem.acquire();
    /* critical section */
}
catch (InterruptedException ie) { }
finally {
    sem.release();
}
```

Java Condition Variables

- Condition variables are associated with an `ReentrantLock`.
- Creating a condition variable using `newCondition()` method of `ReentrantLock`:

```
Lock key = new ReentrantLock();  
Condition condVar = key.newCondition();
```

- A thread waits by calling the `await()` method, and signals by calling the `signal()` method.

Java Condition Variables

- Example:
- Five threads numbered 0 .. 4
- Shared variable `turn` indicating which thread's turn it is.
- Thread calls `doWork()` when it wishes to do some work. (But it may only do work if it is their turn.
- If not their turn, wait
- If their turn, do some work for awhile
- When completed, notify the thread whose turn is next.
- Necessary data structures:

```
Lock lock = new ReentrantLock();
Condition[] condVars = new Condition[5];

for (int i = 0; i < 5; i++)
    condVars[i] = lock.newCondition();
```

```

/* threadNumber is the thread that wishes to do some work */
public void doWork(int threadNumber)
{
    lock.lock();

    try {
        /**
         * If it's not my turn, then wait
         * until I'm signaled.
         */
        if (threadNumber != turn)
            condVars[threadNumber].await();

        /**
         * Do some work for awhile ...
         */

        /**
         * Now signal to the next thread.
         */
        turn = (turn + 1) % 5;
        condVars[turn].signal();
    }
    catch (InterruptedException ie) { }
    finally {
        lock.unlock();
    }
}

```

Alternative Approaches

- Transactional Memory
- OpenMP
- Functional Programming Languages

Transactional Memory

- Consider a function `update()` that must be called atomically. One option is to use mutex locks:

```
void update ()
{
    acquire();

    /* modify shared data */

    release();
}
```

- A memory transaction is a sequence of read-write operations to memory that are performed atomically. A transaction can be completed by adding `atomic{S}` which ensure statements in `S` are executed atomically:

```
void update ()
{
    atomic {
        /* modify shared data */
    }
}
```

OpenMP

- OpenMP is a set of compiler directives and API that support parallel programming.

```
void update(int value)
{
    #pragma omp critical
    {
        count += value
    }
}
```

- The code contained within the `#pragma omp critical` directive is treated as a critical section and performed atomically.

Functional Programming Languages

- Functional programming languages offer a different paradigm than procedural languages in that they do not maintain state.
- Variables are treated as immutable and cannot change state once they have been assigned a value.
- There is increasing interest in functional languages such as Erlang and Scala for their approach in handling data races.

Questions?