Operating Systems
[7. Synchronization Examples]

Chung-Wei Lin

cwlin@csie.ntu.edu.tw

CSIE Department

National Taiwan University

From Operating System Concepts (10th Edition)

Objectives

- Explain the bounded-buffer, readers-writers, and dining-philosophers synchronization problems
- Describe specific tools used by Linux and Windows to solve process synchronization problems
- Illustrate how POSIX and Java can be used to solve process synchronization problems

Outline

- **Classic Problems of Synchronization**
  - **Bounded-Buffer Problem**
  - Readers-Writers Problem
  - Dining-Philosophers Problem
- Synchronization within the Kernel
- POSIX Synchronization
- Synchronization in Java
- Alternative Approaches

Bounded-Buffer Problem

- n buffers
  - Each can hold one item
- Shared data
  - Semaphore **`mutex`** initialized to the value 1 (binary)
  - Semaphore **`full`** initialized to the value 0
  - Semaphore **`empty`** initialized to the value n

Bounded-Buffer Problem: Producer

- The structure of the producer process

```
while (true) {
    ...
    /* produce an item in next_produced */
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add next_produced to the buffer */
    ...
    signal(mutex);
    signal(full);
}
```

Bounded-Buffer Problem: Consumer

- The structure of the consumer process

```
while (true) {
    wait(full);
    wait(mutex);
    ...
    /* remove an item from buffer to next_consumed */
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume the item in next_consumed */
    ...
}
```

Outline

- **Classic Problems of Synchronization**
  - Bounded-Buffer Problem
  - **Readers-Writers Problem**
  - Dining-Philosophers Problem
- Synchronization within the Kernel
- POSIX Synchronization
- Synchronization in Java
- Alternative Approaches

Readers-Writers Problem (1/2)

- A data set is shared among a number of concurrent processes
  - **Readers**
    - They only read the data set
    - They do **not** perform any update
  - **Writers**
    - They can both read and write
- Problem
  - Multiple readers can read at the same time
  - Only one single writer can access the shared data at the same time

Readers-Writers Problem (2/2)

- Several variations of how readers and writers are considered
  - All involve some form of priorities
  - First readers-writers problem
    - No reader be kept waiting unless a writer has already obtained permission to use the shared object
  - Second readers-writers problem
    - Once a writer is ready, that writer perform its write as soon as possible
- Shared data (for the first readers-writers problem)
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1 (binary)
  - Semaphore **`mutex`** initialized to 1 (binary)
  - Integer **`read_count`** initialized to 0

Readers-Writers Problem: Writer

- The structure of a writer process

```
while (true) {
    wait(rw_mutex);
    ...
    /* writing is performed */
    ...
    signal(rw_mutex);
}
```

Readers-Writers Problem: Reader

- The structure of a reader process

```
while (true){
    wait(mutex);
    read_count++;
    if (read_count == 1) /* first reader */
        wait(rw_mutex);
    signal(mutex);
    ...
    /* reading is performed */
    ...
    wait(mutex);
    read_count--;
    if (read_count == 0) /* last reader */
        signal(rw_mutex);
    signal(mutex);
}
```

Reader-Writer Locks

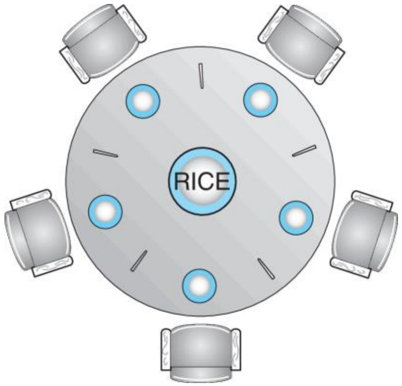
- The readers-writers problem and its solutions have been generalized to provide reader-writer locks on some systems
  - Acquiring a reader-writer lock requires specifying the mode of the lock
    - A process wishing only to read shared data requests the lock in "read" mode
    - A process wishing to modify shared data requests the lock in "write" mode
    - Multiple processes are permitted to concurrently acquire the lock in read mode
    - Only one process may acquire the lock in write mode
- Reader-writer locks are most useful
  - In applications where it is easy to identify which processes only read shared data and which processes only write shared data
  - In applications that have more readers than writers

Outline

- **Classic Problems of Synchronization**
  - Bounded-Buffer Problem
  - Readers-Writers Problem
  - **Dining-Philosophers Problem**
- Synchronization within the Kernel
- POSIX Synchronization
- Synchronization in Java
- Alternative Approaches

Dining-Philosophers Problem

- N philosophers sit at a round table with a bowel of rice in the middle
- They spend their lives alternating thinking and eating
- They occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- Shared data (in the case of 5 philosophers)
  - Bowl of rice (data set)
  - Semaphore **`chopstick[5]`** initialized to 1 (binary)



Semaphore Solution

- The structure of philosopher i

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

- What is the problem with this algorithm?

Monitor Solution (1/3)

```
monitor DiningPhilosophers
{
    enum {THINKING, HUNGRY, EATING} state[5];
    condition self[5];
    void pickup(int i){
        state[i] = HUNGRY;
        test(i);
        if(state[i] != EATING) self[i].wait();
    }
    void putdown(int i){
        state[i] = THINKING;
        // test left and right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
    }
```

Monitor Solution (2/3)

```
    void test(int i){
        if((state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING)){
            state[i] = EATING;
            self[i].signal();
        }
    }
    initialization_code(){
        for(int i = 0; i < 5; i++)
            state[i] = THINKING;
    }
}
```

Monitor Solution (3/3)

- Each philosopher i invokes the operations **`pickup()`** and **`putdown()`** in the following sequence:

```
...
DiningPhilosophers.pickup(i);
/* eat */
DiningPhilosophers.putdown(i);
...
```

- No deadlock, but starvation is possible

Outline

- Classic Problems of Synchronization
- **Synchronization within the Kernel**
  - **Synchronization in Windows**
  - Synchronization in Linux
- POSIX Synchronization
- Synchronization in Java
- Alternative Approaches

Synchronization in Windows (1/3)

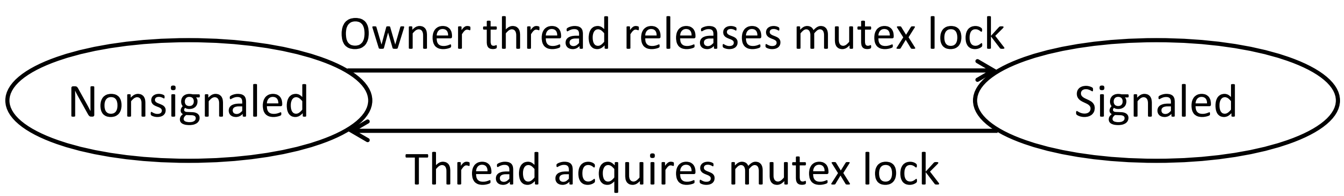
- When the kernel accesses a global resource on a single-processor system
  - It temporarily mask interrupts for all interrupt handlers that may also access the global resource
- On a multiprocessor system
  - It protects access to global resources using spinlocks
    - The kernel uses spinlocks only to protect short code segments
    - For reasons of efficiency, the kernel ensures that a thread will never be preempted while holding a spinlock

Synchronization in Windows (2/3)

- For thread synchronization outside the kernel, Windows provides **dispatcher objects**
  - Using a dispatcher object, threads synchronize according to several different mechanisms
    - Mutex locks, semaphores, events, and timers
  - The system protects shared data by requiring a thread to
    - Gain ownership of a mutex to access the data
    - Release ownership when it is finished
  - Events are similar to condition variables
    - They may notify a waiting thread when a desired condition occurs
  - Timers are used to notify one (or more than one) thread that a specified amount of time has expired

Synchronization in Windows (3/3)

- Dispatcher objects may be in either a **signaled** or **nonsignaled** state
  - An object in a signaled state is available
    - A thread will not block when acquiring the object
  - An object in a nonsignaled state is not available
    - A thread will block when attempting to acquire the object



Outline

- Classic Problems of Synchronization
- **Synchronization within the Kernel**
  - Synchronization in Windows
  - **Synchronization in Linux**
- POSIX Synchronization
- Synchronization in Java
- Alternative Approaches

Synchronization in Linux

- A nonpreemptive kernel prior to Version 2.6; preemptive now
- Atomic integers

  `atomic_t counter;` (`atomic_t` is the type for atomic integer)

```
int value;
atomic_set(&counter,5);  // counter = 5
atomic_add(10,&counter); // counter = counter + 10
atomic_sub(4,&counter);  // counter = counter - 4
atomic_inc(&counter);    // counter = counter + 1
value = atomic_read(&counter); // value = 12
```

- Single processor
  - Disable/enable kernel preemption
- Multiple processors
  - Acquire/release spinlock

Outline

- Bounded-Buffer Problem
- Readers-Writers Problem
- Dining-Philosophers Problem
- Synchronization in Windows and Linux
- **POSIX Synchronization**
  - **POSIX Mutex Locks**
  - POSIX Semaphores
  - POSIX Condition Variables
- Synchronization in Java
- Alternative Approaches

POSIX Mutex Locks

- Create and initialize the lock

```
#include <pthread.h>
pthread_mutex_t mutex;
/* create and initialize the mutex lock */
pthread_mutex_init(&mutex, NULL);
```

- Acquire and release the lock

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);
/* critical section */
/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

Outline

- Bounded-Buffer Problem
- Readers-Writers Problem
- Dining-Philosophers Problem
- Synchronization in Windows and Linux
- **POSIX Synchronization**
  - POSIX Mutex Locks
  - **POSIX Semaphores**
  - POSIX Condition Variables
- Synchronization in Java
- Alternative Approaches

POSIX Semaphores

- Semaphores are not part of the POSIX standard and instead belong to the POSIX SEM extension
- **Named** semaphores
  - Can be used by unrelated processes
- **Unnamed** semaphores
  - Cannot be used by unrelated processes

POSIX Named Semaphores

- Create and initialize the semaphore

```
#include <semaphore.h>
sem_t *sem;
/* Create the semaphore & initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
```

- Another process can access the semaphore by referring to its name **SEM**
- Acquire and release the semaphore

```
/* acquire the semaphore */
sem_wait(sem);
/* critical section */
/* release the semaphore */
sem_post(sem);
```

POSIX Unnamed Semaphores

- Create and initialize the semaphore

```
#include <semaphore.h>
sem_t sem;
/* Create the semaphore & initialize it to 1 */
sem_init(&sem, 0, 1);
```

- Acquire and release the semaphore

```
/* acquire the semaphore */
sem_wait(&sem);
/* critical section */
/* release the semaphore */
sem_post(&sem);
```

Outline

- Bounded-Buffer Problem
- Readers-Writers Problem
- Dining-Philosophers Problem
- Synchronization in Windows and Linux
- **POSIX Synchronization**
  - POSIX Mutex Locks
  - POSIX Semaphores
  - **POSIX Condition Variables**
- Synchronization in Java
- Alternative Approaches

POSIX Condition Variables (1/2)

- POSIX condition variables are associated with a POSIX mutex lock to provide mutual exclusion
  - Since POSIX is typically used in C/C++ and these languages do not provide a monitor
- Create and initialize 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 (2/2)

- 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);
```

Outline

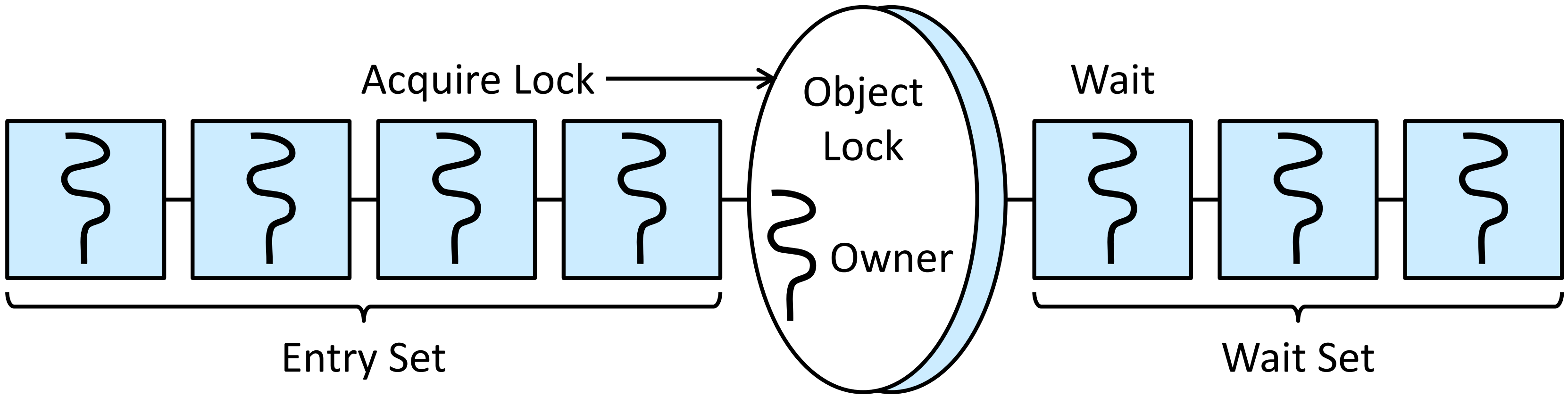
- Bounded-Buffer Problem
- Readers-Writers Problem
- Dining-Philosophers Problem
- Synchronization in Windows and Linux
- POSIX Synchronization
- **Synchronization in Java**
  - **Java Monitors**
  - Reentrant Locks
  - Semaphores
  - Condition Variables
- Alternative Approaches

Java Monitors (1/3)

- Every object in Java has associated with it a single lock
- When a method is declared to be **synchronized**, calling the method requires owning the lock for the object
  - Declare a synchronized method by placing the synchronized keyword in the method definition
  - Examples: the `insert()` and `remove()` methods in the `BoundedBuffer` class
- If the lock is already owned by another thread, the calling thread blocks and is placed in the **entry set** of the object
  - The entry set represents the set of threads waiting for the lock to become available
- The lock is released when the thread exits the method

Java Monitors (2/3)

- A thread that tries to acquire an unavailable lock is placed in the object's entry set
- Each object also has a **wait set**
  - When a thread calls **`wait()`**
    - The thread 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 Monitors (3/3)

- A thread typically calls **`wait()`** when it is waiting for a condition to become true
  - How does the thread get notified?
- When a thread calls **`notify()`**
  - Pick an arbitrary thread T from the list of threads in the wait set
  - Move T from the wait set to the entry set
  - Set the state of T from blocked to runnable
- T is now eligible to compete for the lock with the other threads

Java Monitors: Bounded-Buffer (1/3)

```
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 following slides */
    }
    /* Consumers call this method */
    public synchronized E remove() {
        /* See following slides */
    }
}
```

Java Monitors: Bounded-Buffer (2/3)

```
/* 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();
}
```

Java Monitors: Bounded-Buffer (3/3)

```
/* 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;
}
```

Outline

- Bounded-Buffer Problem
- Readers-Writers Problem
- Dining-Philosophers Problem
- Synchronization in Windows and Linux
- POSIX Synchronization
- **Synchronization in Java**
  - Java Monitors
  - **Reentrant Locks**
  - Semaphores
  - Condition Variables
- Alternative Approaches

Java Reentrant Locks

- **`lock()`** assigns the invoking thread lock ownership
  - If the lock is available, or
  - If the thread already owns it, which is why it is termed reentrant
- The **`finally`** clause ensures that 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();
}
```

Outline

- Bounded-Buffer Problem
- Readers-Writers Problem
- Dining-Philosophers Problem
- Synchronization in Windows and Linux
- POSIX Synchronization
- **Synchronization in Java**
  - Java Monitors
  - Reentrant Locks
  - **Semaphores**
  - Condition Variables
- Alternative Approaches

Java Semaphores

- Constructor

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

- Usage

```
Semaphore sem = new Semaphore(1);
try {
   sem.acquire();
   /* critical section */
}
catch (InterruptedException ie) {}
finally {
   sem.release();
}
```

Outline

- Bounded-Buffer Problem
- Readers-Writers Problem
- Dining-Philosophers Problem
- Synchronization in Windows and Linux
- POSIX Synchronization
- **Synchronization in Java**
  - Java Monitors
  - Reentrant Locks
  - Semaphores
  - **Condition Variables**
- Alternative Approaches

Java Condition Variables (1/3)

- Create a condition variable by
  - Creating a **`ReentrantLock`**
  - Invoking its **`newCondition()`** method
    - It returns a **`Condition`** object representing the condition variable for the associated **`ReentrantLock`**
- Usage

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

- A thread
  - Wait by calling the **`await()`** method
  - Signal by calling the **`signal()`** method

Java Condition Variables (2/3)

- Example
  - Five threads numbered from 0 to 4
    - Threads share variable **`turn`** indicating which thread's turn it is
  - A thread calls **`doWork()`** when it wishes to do some work
    - If it is not its turn, wait
    - If it is its turn, do some work for awhile
    - When it completes, it notifies 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();
```

Java Condition Variables (3/3)

```
/* threadNumber is the thread wishing to do some work */
public void doWork(int threadNumber) {
    lock.lock();
    try {
        /* If not my turn, then wait until 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();
    }
}
```

Outline

- Bounded-Buffer Problem
- Readers-Writers Problem
- Dining-Philosophers Problem
- Synchronization in Windows and Linux
- POSIX Synchronization
- Synchronization in Java
- **Alternative Approaches**
  - Transactional Memory
  - OpenMP
  - Functional Programming Languages

Transactional Memory

- 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
  - 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

Objectives

- Explain the bounded-buffer, readers-writers, and dining-philosophers synchronization problems
- Describe specific tools used by Linux and Windows to solve process synchronization problems
- Illustrate how POSIX and Java can be used to solve process synchronization problems

Q&A