Chapter 6: Process Synchronization
Module 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
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

```c
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

```c
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}
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, T₀ and T₁ (Tᵢ and Tⱼ)
- Three solutions presented. All implement this `MutualExclusion` interface:

```java
public interface MutualExclusion {
    public static final int TURN_0 = 0;
    public static final int TURN_1 = 1;
    public abstract void enteringCriticalSection(int turn);
    public abstract void leavingCriticalSection(int turn);
}
```
Algorithm Factory class

Used to create two threads and to test each algorithm

```java
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();
    }
}
```
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

```java
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 2

- Add more state information
  - Boolean flags to indicate thread’s interest in entering critical section
- Progress requirement still not met
  - Why?
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
public void leavingCriticalSection(int t) {
    if (t == 0)
        flag0 = false;
    else
        flag1 = false;
}
Algorithm 3

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

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

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

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

```c
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

\[
P_0 \quad P_1
\]
\[
\hspace{1cm} \text{acquire}(S); \quad \text{acquire}(Q);
\]
\[
\hspace{1cm} \text{acquire}(Q); \quad \text{acquire}(S);
\]
\[
\hspace{1cm} . \quad .
\]
\[
\hspace{1cm} . \quad .
\]
\[
\hspace{1cm} \text{release}(S); \quad \text{release}(Q);
\]
\[
\hspace{1cm} \text{release}(Q); \quad \text{release}(S);
\]

**Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.
Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers 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
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 */ }
}
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

```java
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;
}
```
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);
        }
    }
}
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();
        }
    }
}
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();
    }
}

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();
        }
    }
}
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();
}
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 */ }
}
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();
}
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 $i$:
  ```java
  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

```java
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 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
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;
    }
Java Synchronization

- Bounded Buffer solution using \texttt{synchronized, wait()}, \texttt{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

acquire lock

object lock

owner

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;
}
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

Entry set

Object lock

Owner

Wait set

Acquire lock

Wait
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
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();
}
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;
}
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
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;
}
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();
}
public synchronized void acquireWriteLock() {
    while (readerCount > 0 || dbWriting == true) {
        try {
            wait();
        } catch(InterruptedException e) {
        }
        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
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;
    }
}
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
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
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 a 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 a 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
<table>
<thead>
<tr>
<th></th>
<th>single processor</th>
<th>multiple processors</th>
</tr>
</thead>
</table>
### Single Processor vs. Multiple Processors

<table>
<thead>
<tr>
<th>Single Processor</th>
<th>Multiple Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disable kernel preemption.</td>
<td>Acquire spin lock.</td>
</tr>
<tr>
<td>Enable kernel preemption.</td>
<td>Release spin lock.</td>
</tr>
</tbody>
</table>
6.18

entry queue

shared data

queues associated with $x, y$ conditions

$x \rightarrow \square \rightarrow \square \rightarrow \downarrow$

$y \rightarrow \square \rightarrow \square \rightarrow \downarrow$

operations

initialization code
owner thread releases mutex lock

nonsignaled  →  signaled

thread acquires mutex lock
End of Chapter 6