Chapter 6 in Old Ed:
Chapter 5 in 9th Ed:
Process Synchronization

Get processes (threads) to work together in a coordinated manner.

Thanks to the author of the textbook [**SGG**] for providing the base slides. I made several changes/additions.
These slides may incorporate materials kindly provided by Prof. Dakai Zhu.
So I would like to thank him, too.
Turgay Korkmaz

Process Synchronization

■ Background	**
● Problems with concurrent access to shared data (Race condition)	
■ The Critical-Section Problem	*****
■ Peterson’s Solution	*****
■ Synchronization mechanisms	
● Hardware support: e.g., *TestAndSet*	***
● Software solution: e.g., *Semaphore*	*****
■ Classic Problems of Synchronization	****
■ Monitors	***
■ Java Synchronization	***
■ Synchronization Examples	*
■ Atomic Transactions	(more later)

Objectives

- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To introduce the concept of an atomic transaction and describe mechanisms to ensure atomicity

Shared data

at the same logical address space √

at different address space through messages (later in DS)

BACKGROUND

Shared Data

- **Concurrent** access to **shared** data may result in data **inconsistency** (how/hwy)
- Maintaining data consistency requires mechanisms to ensure the **orderly** execution of **cooperating** processes/threads

Example1:
Concurrent Access to Shared Data

- Two processes/threads A and B have access to a shared global variable “Balance”

Thread A:
Balance = Balance + 100

```
A1. LOAD R1, BALANCE
A2. ADD R1, 100
A3. STORE BALANCE, R1
```

Thread B:
Balance = Balance - 200

```
B1. LOAD R2, BALANCE
B2. SUB R2, 200
B3. STORE BALANCE, R2
```

What is the problem then?

- Observe: In a *time-shared* system the *exact instruction execution order* cannot be predicted

- Scenario 1:

  A1. LOAD R1, BALANCE

  A2. ADD R1, 100

  A3. STORE BALANCE, R1

  Context Switch!

  B1. LOAD R2, BALANCE

  B2. SUB R2, 200

  B3. STORE BALANCE, R2

- *Sequential* correct execution
- Balance is effectively decreased by 100!

- Scenario 2:

  B1. LOAD R2, BALANCE

  B2. SUB R2, 200

  Context Switch!

  A1. LOAD R1, BALANCE

  A2. ADD R1, 100

  A3. STORE BALANCE, R1

  Context Switch!

  B3. STORE BALANCE, R2

- Mixed wrong execution
- Balance is effectively decreased by 200!

Example2: Consumer-Producer

- Provide a solution to the consumer-producer problem that fills all the buffers.
- Have an integer count that keeps track of the number of full buffers. Initially, count is set to 0.
- It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

```
// Producer
while (count == BSIZE)
   ; // do nothing

   // add an item to the buffer
buffer[in] = item;
in = (in + 1) % BSIZE;
++count;
```

```
// Consumer
while (count == 0)
   ; // do nothing

   // remove an item from the
buffer item = buffer[out];
out = (out + 1) % BSIZE;
--count;
```

What is the problem then?

- count++ could be implemented as
```
register1 = count
register1 = register1 + 1
count = register1
```
- count-- could be implemented as
```
register2 = count
register2 = register2 - 1
count = register2
```

How many different interleaving can we have?

- Consider this execution interleaving with “count = 5” initially:

```
T0: producer execute register1 = count                 {register1 = 5}
T1: producer execute register1 = register1 + 1         {register1 = 6}
T2: consumer execute register2 = count                 {register2 = 5}
T3: consumer execute register2 = register2 – 1         {register2 = 4}
T4: producer execute count = register1                 {count = 6 }
T5: consumer execute count = register2                 {count = 4}
```

Race Condition (RC)

Race Conditions

- **Race Condition (RC)**: the outcome of an execution depends on the particular order in which the accesses to shared data take place
- RC is a serious problem for concurrent systems using shared variables!
- To solve it, we need to make sure that only **one process** executes some *high-level code sections* `(e.g., count++)` known as **critical sections (CS)**
  - In other words, CS must be executed **atomically**
  - Atomic operation means that it completes in its entirety without worrying about interruption by any other potentially conflict-causing process

Multiple processes/threads compete to use some shared data

Solutions

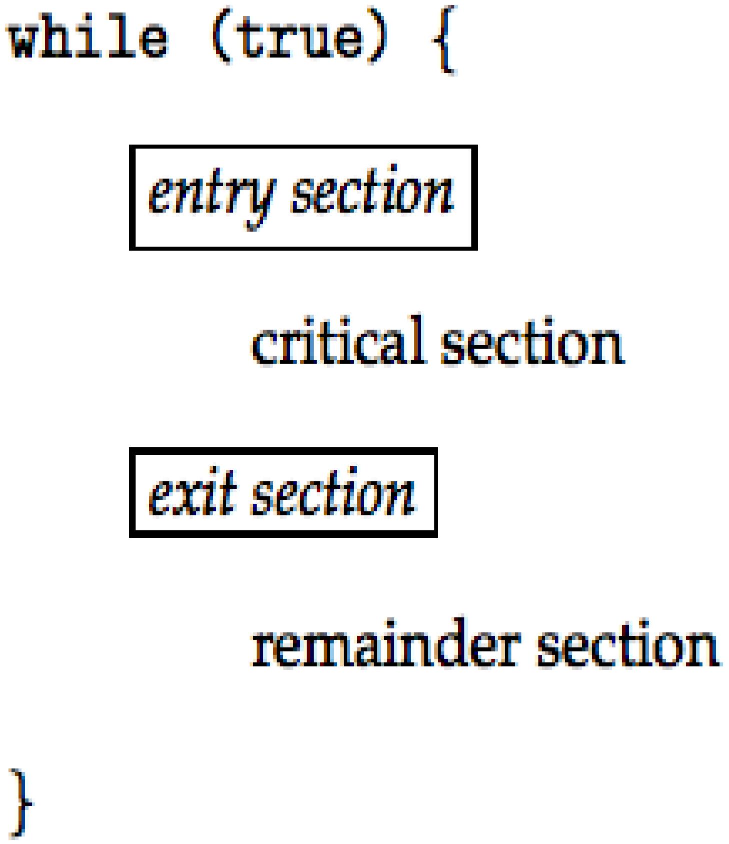
- SW based (e.g., Peterson’s solution, Semaphores, Monitors) and
- HW based (e.g., Locks, disable interrupts, atomic get-and-set instruction)

CRITICAL-SECTION (CS) PROBLEM

Critical-Section (CS) Problem

- Cooperating processes or threads compete to use some shared data in a code segment, called **critical section** (CS)
- **CS Problem** is to ensure that only **one** process is allowed to execute in its CS (for the same shared data) at any time
- Each process must request permission to **enter** its CS (allow or wait)
- CS is followed by **exit** and **remainder** sections.

```
// count++                // count- -;
reg1  = count             reg2  = count
reg1  = reg1 + 1          reg2  = reg2 – 1
count = reg1              count = reg2
```

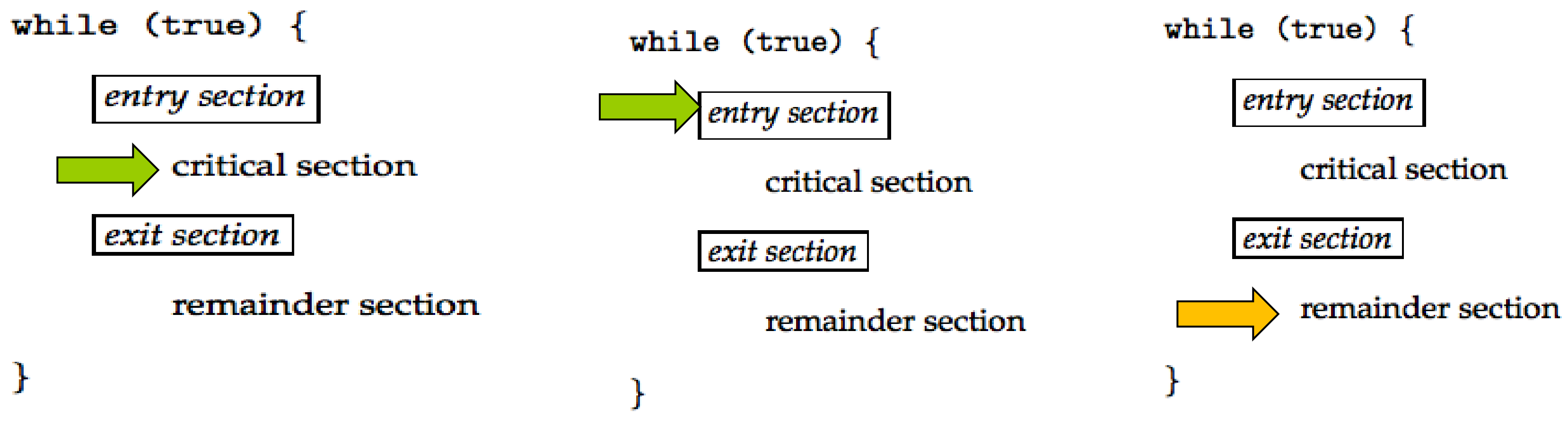


```
while (true) {

        entry section

            critical section

        exit section

            remainder section

}
```

Requirements for the Solutions 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. (At most one process is in its critical section at any time.)
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then only the ones that are not busy in their reminder sections must compete and one should be able to enter its CS (the selection cannot be postponed indefinitely to wait a process executing in its remainder section).
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

Mutual Exclusion

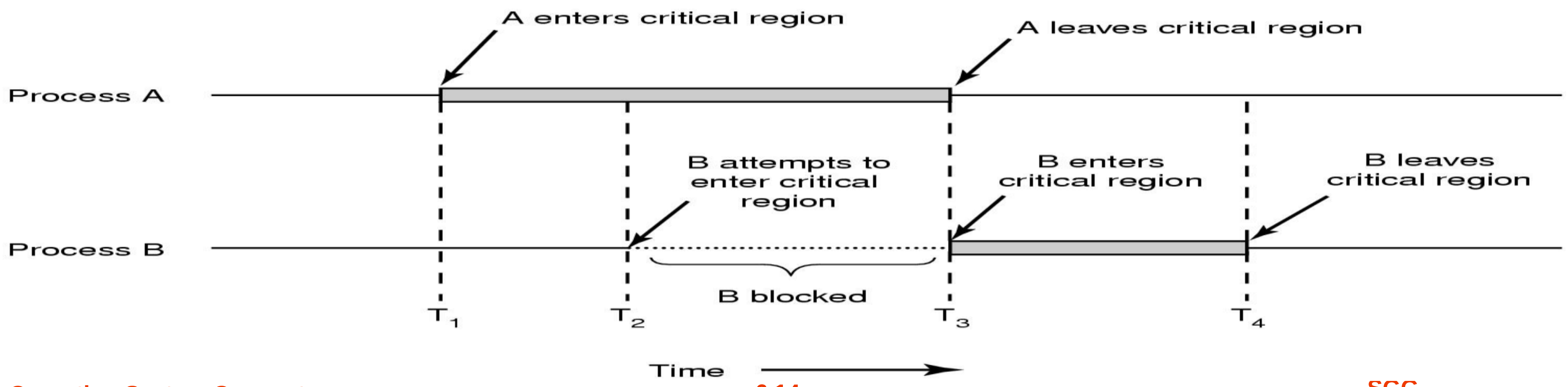


```
while (true) {
        entry section
    ➡       critical section
        exit section
            remainder section
}
```

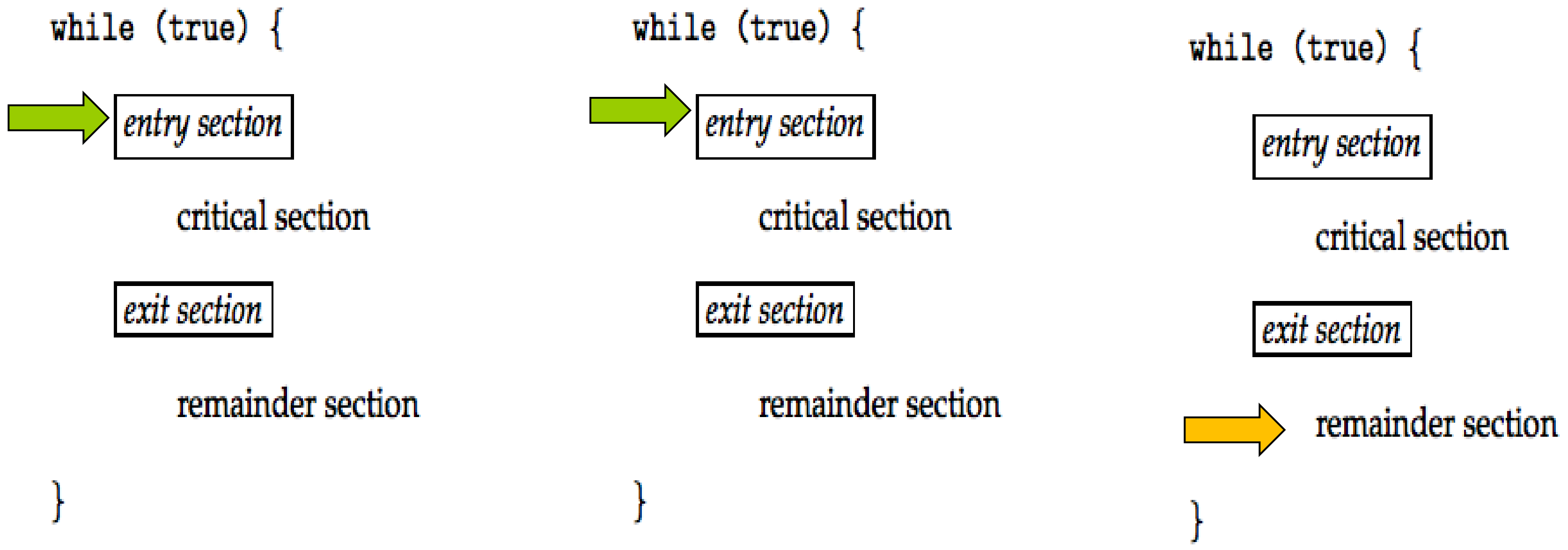
```
while (true) {
    ➡   entry section
            critical section
        exit section
            remainder section
}
```

```
while (true) {
        entry section
            critical section
        exit section
    ➡       remainder section
}
```

- If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
- At most one process is in its critical section at any time.



Progress



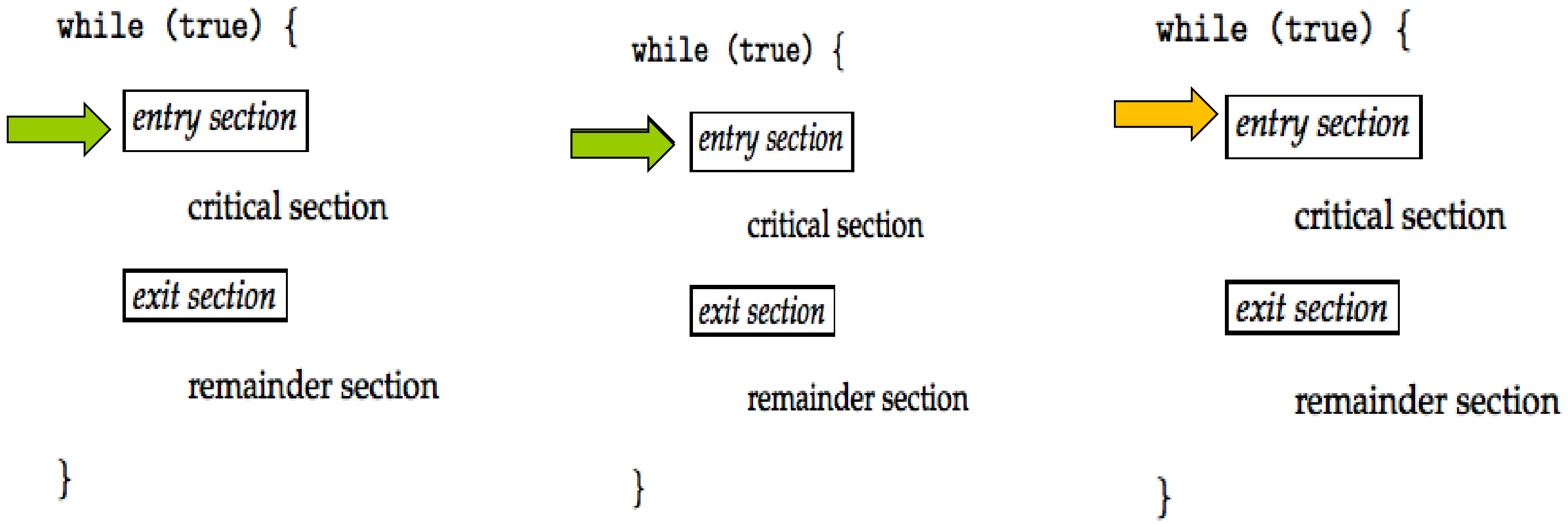
```
while (true) {
    entry section
        critical section
    exit section
        remainder section
}
```

```
while (true) {
    entry section
        critical section
    exit section
        remainder section
}
```

```
while (true) {
    entry section
        critical section
    exit section
        remainder section
}
```

- If no process is executing in its critical section and there exist some processes that wish to enter their critical section,
- then only the ones that are not busy in their reminder sections must compete and one should be able to enter its CS
- the selection cannot be postponed indefinitely to wait a process executing in its remainder section.

Bounded Waiting



```
while (true) {
    entry section
        critical section
    exit section
        remainder section
}
```

```
while (true) {
    entry section
        critical section
    exit section
        remainder section
}
```

```
while (true) {
    entry section
        critical section
    exit section
        remainder section
}
```

- 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

RC and CS in OS kernel code

- Suppose two processes try to open files at the same time, to allocate memory etc.
- Two general approach
  - Non-preemptive kernel *(free from RC, easy to design)*
  - Preemptive kernel *(suffer from RC, hard to design)*
- Why, then, would we want non-preemptive kernel
  - Real-time systems
  - Avoid arbitrarily long kernel processes
  - Increase responsiveness
- Later, we will study how various OSes manage preemption in the kernel

Suppose load and store are atomic!

PETERSON'S SOLUTION

Here is a software solution to CS

Suppose we have two processes/threads and a shared variable `turn`, which is initially set to 0 or 1;

```
Process 0:
----------
while(TRUE) {
  while (turn == 1) ;
     critical section
  turn = 1;
     remainder section
}
```

```
Process 1:
----------
while(TRUE) {
   while (turn == 0) ;
      critical section
   turn = 0;
      remainder section
}
```

Is this correct or incorrect? Why?
Think about mutual exclusion, progress, bounded waiting

Strict alternation

Here is a ***corrected*** solution to CS

Known as **Peterson's solution**

```
Process 0:                            Process 1:
----------                            ----------
while(TRUE) {                         while(TRUE) {
  flag[0] = 1;  // I am ready           flag[1] = 1;
  turn = 1;                             turn = 0;
  while (flag[1]==1 &&                  while (flag[0]==1 &&
         turn == 1) ;                          turn == 0) ;
     critical section                      critical section
  flag[0] = 0; // I am not ready        flag[1] = 0;
     remainder section                     remainder section
}                                     }
```

- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P_i is ready!
- If I am not ready, the other process can enter CS even if it is not its turn..
- So we avoid strict alternation…. How about bounded waiting?

Does Peterson's Solution Satisfy the Requirements

- Mutual Exclusion
  - P0 enters CS only if either flag[1] is false or turn=0
  - P1 enters CS only if either flag[0] is false or turn=1
  - If both are in CS then both flag should be true, but then turn can be either 0 or 1 but cannot be both
- Progress
- Bounded-waiting

```
Process 0:
---------
while(TRUE) {
  flag[0] = 1;
  turn = 1;
  while (flag[1]==1 &&
         turn == 1) ;
     critical section *
  flag[0] = 0;
     remainder section
}
```

```
Process 1:
---------
while(TRUE) {
  flag[1] = 1;
  turn = 0;
* while (flag[0]==1 &&
         turn == 0) ;
     critical section
  flag[1] = 0;
     remainder section
}
```

Peterson’s Solution +/-

- Software solution for two processes/threads (T0 and T1): alternate between CS and remainder codes
- Assume that the LOAD and STORE instructions are **atomic** (cannot be interrupted).
- Otherwise, and actually it cannot be guaranteed that this solution will work on modern architectures
- But still it is a good algorithmic solution to understand the synchronization issues such as *mutual exclusion, progress, bounded waiting*

```
// process i,   j=1-i
do {
   flag[i] = true;
   turn = j;
   while (flag[j] &&
          turn == j)   ;
      critical section
   flag[i] = false;
      remainder section
} while(1);
```

Many systems provide hardware support for critical section code

SYNC HARDWARE

Solution to Critical-Section Problem Using Locks

- SW-based solutions are not guaranteed to work on modern architectures, why?
- In general we need a LOCK mechanism which could be based on HW (easy and efficient) or SW (quite sophisticated) ….
- So we will now study HW based supports first.

```
while (true) {
        acquire lock
        critical section
        release lock
        remainder section
}
```

Synchronization Hardware

- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this are not broadly scalable
    - Clock updates!
- Modern machines provide special **atomic** (non-interruptible) hardware instructions
  - Test memory word and set value
  - Swap contents of two memory words
  - Lock the bus not the interrupt, not easy to implement on multiprocessor systems

```
do {
   ……
   DISABLE INTERRUPT
      critical section
   ENABLE INTERRUPT
      Remainder statements
} while (1);
```

```
do {
   ……
   acquire lock
      critical section
   release lock
      Remainder statements
} while (1);
```

Data Structure for Hardware Solutions

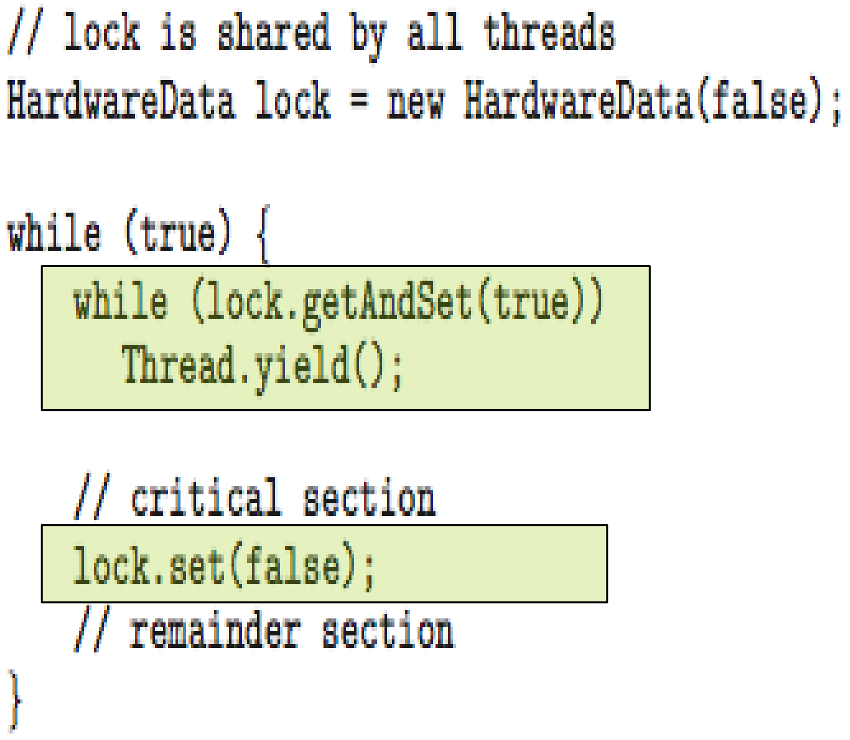
```
public class HardwareData
{
    private boolean value = false;

    public HardwareData(boolean value) {
       this.value = value;
    }

    public boolean get() {
       return value;
    }

    public void set(boolean newValue) {
       value = newValue;
    }

    public boolean getAndSet(boolean newValue) {
       boolean oldValue = this.get();
       this.set(newValue);

       return oldValue;
    }

   public void swap(HardwareData other) {
      boolean temp = this.get();

      this.set(other.get());
      other.set(temp);
    }
}
```

Suppose these methods functions are atomic

```
//int TestAndSet(int *target)
int GetAndSet(int *target)
{
    int m = *target;
    *target = TRUE;
    return m;
}
```

```
void Swap(int *a, int *b)
{
   int temp = *a;
    *a = *b;
    *b = temp:
 }
```

Solution using GetAndSet Instruction



```
// lock is shared by all threads
HardwareData lock = new HardwareData(false);

while (true) {
   while (lock.getAndSet(true))
      Thread.yield();

   // critical section
   lock.set(false);
   // remainder section
}
```

```
lock = FALSE;

while(1) {
     ......
     while (GetAndSet(&lock));

        critical section

     lock = FALSE; //free the lock

        remainder section

}
```

a) spin-locks → busy waiting;
b) Waiting processes loop continuously at the entry point; waste cpu cycles
c) User space threads might not give control to others !
d) Hardware dependent; and **NOT Bounded waiting!!**

Solution 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);

    // critical section
    lock.set(false);
    // remainder section
}
```

```
lock = FALSE
while(1){
    … …
    key = TRUE;
    while (key == TRUE)
        Swap(&lock, &key );

          Critical Section;

    lock = FALSE //release the lock

         remainder section

}
```

What is the problem with solutions so far!

- Mutual exclusion √
- Progress √
- Bounded waiting ?

Exercise

Write a general solution to synchronize *n* processes

```
// shared data structures
int waiting[n]={0}; // to enter CS
int lock=0;
// code for P_i
do {
    Entry section
        // CS
    Exit Section
        // Remainder
        // Section
} while(1);
```

```
waiting[i] = 1;
key = 1;
while(waiting[i] && key)
   key = GetAndSet(&lock);
waiting[i] = 0;
```

```
j = (i+1) % n;
while((j!=i) && !waiting[j])
     j = (j+1) % n;
if (j == i)
   lock=0;
else
   waiting[j] = 0
```

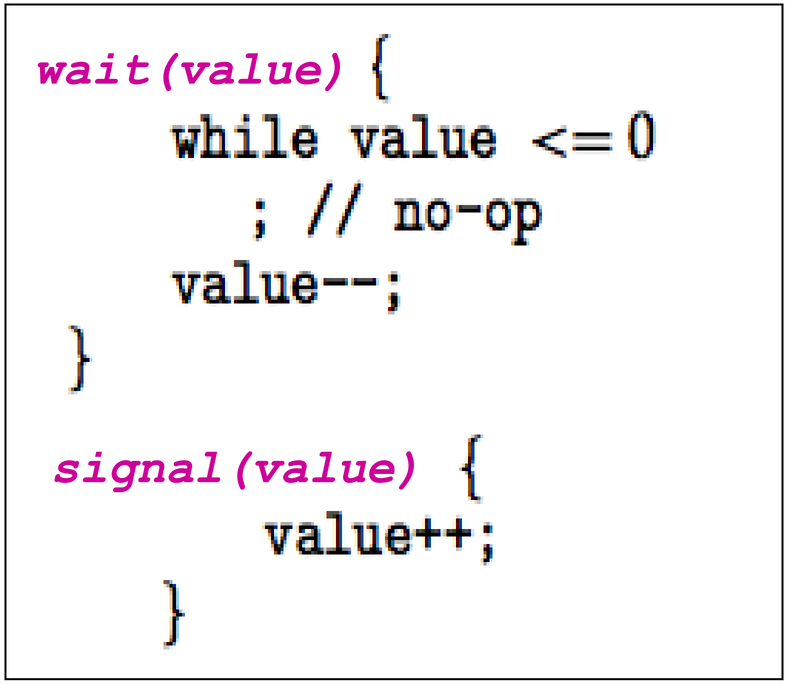
Hardware instructions are complicated for programmers and spin-locks waste CPU time…

Software-based synchronization support to deal with these two problems

SEMAPHORES

Semaphore

- Semaphore *S* – integer variable
- Can only be accessed via two indivisible (**atomic**) operations
  - **acquire()** and **release()**
  - Originally called **P()** and **V(),**
  - Also called: **wait()** and **signal()**; or **down()** and **up()**



```
wait(value) {
      while value <= 0
         ; // no-op
      value--;
}

signal(value) {
        value++;
   }
```

- How can we make wait() and signal() atomic? (We will discuss later)
- First let us focus on their usage
  - Easy to generalize, and less complicated for application programmers
  - Busy waiting (spinlock) can be avoided by blocking a process execution until some condition is satisfied

Semaphore Usage

- Counting semaphore – integer value can range over an unrestricted domain
  - Can be used to control access to a given resources consisting of finite number of instances
- Binary semaphore – integer value can range only between 0 and 1; Also known as mutex locks

```
Semaphore sem = new Semaphore(1);
sem.acquire();
   // critical section
sem.release();
   // remainder section
```

```
S = number of resources
while(1){
   ……
   wait(S);
      use one of S resource
   signal(S);
      remainder section
}
```

```
mutex = 1
while(1){
    ……
    wait(mutex);

         Critical Section

    signal(mutex);

         remainder section
}
```

Semaphore Usage (cont'd)

- Also used for synchronization

Suppose we require S2 to be executed after S1 is completed

```
// Proc1
…
S1;
…
```

```
// Proc2
…
S2;
…
```

- How can we synchronize these two processes?
- Declare a **`sync`** semaphore and initially set it to 0

```
// Proc1
…
S1;
Signal(sync);
…
```

```
// Proc2
…
wait(sync);
S2;
…
```

Java Example Using Semaphores

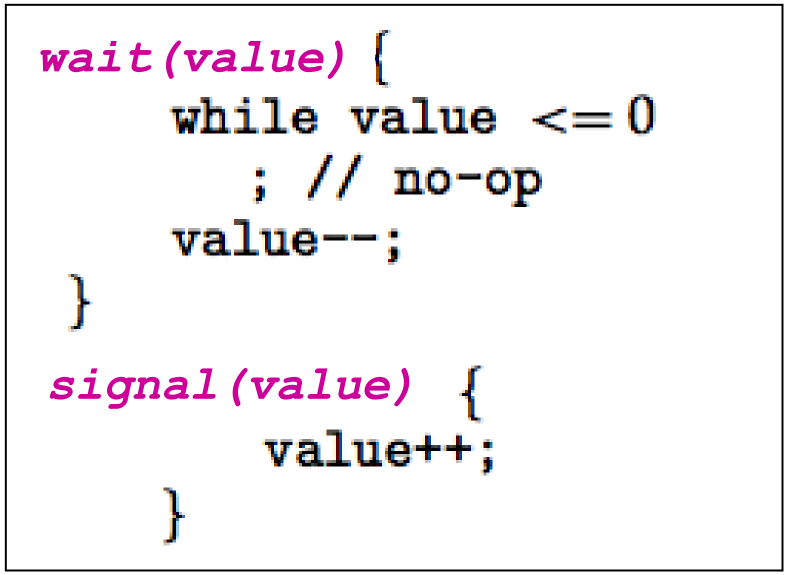
```
public class Worker implements Runnable
{
   private Semaphore sem;

   public Worker(Semaphore sem) {
      this.sem = sem;
   }

   public void run() {
      while (true) {
         sem.acquire();
         criticalSection();
         sem.release();
         remainderSection();
      }
   }
}
```

```
criticalSection() {
  Balance = Balance – 100;
}
```

```
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));
      for (int i = 0; i < 5; i++)
         bees[i].start();
   }
}
```

Semaphore Implementation

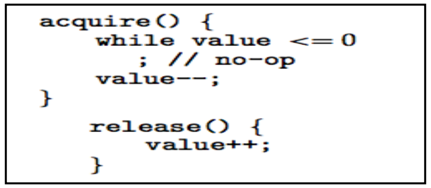
- Main disadvantage of this implementation is that it requires **busy waiting** because of **spin lock**
  - Waste CPU time
  - Processes might do context SW but threads would be in loop for ever



```
wait(value) {
        while value <=0
            ; // no-op
        value--;
 }
signal(value)  {
            value++;
      }
```

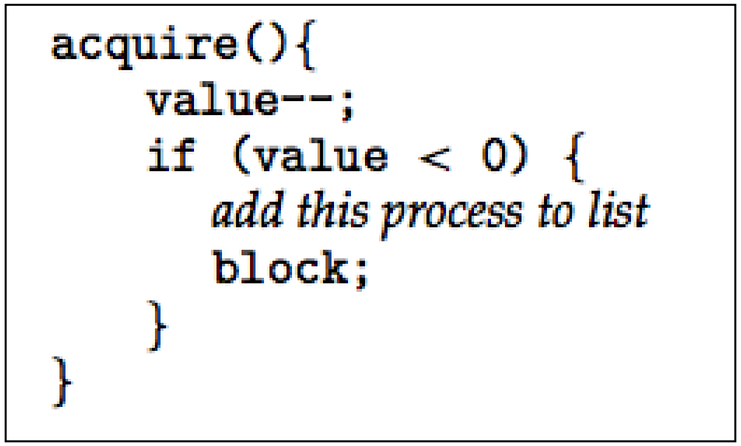
- To overcome busy waiting, we can replace spinlock with **block process**,
  - Wait/acquire blocks the process (e.g., places the process in a queue) if the semaphore value is not positive. And give CPU to another process
  - Signal /release will remove a process from queue and wake it up
  - If CS is short, spinlock might be better than this option as it avoids context SW

Semaphore Implementation with no Busy waiting

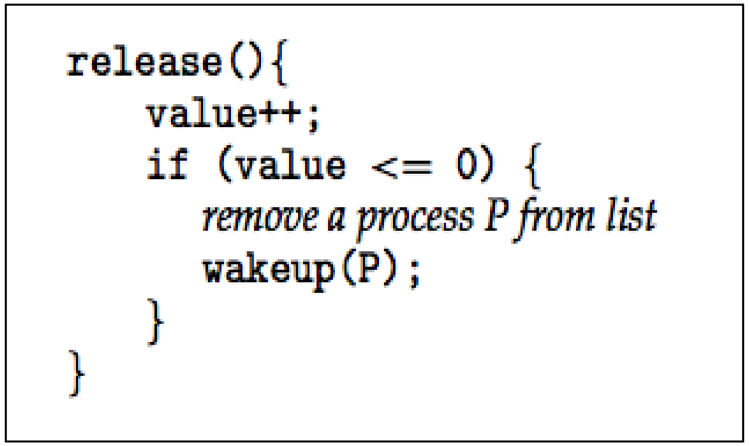


```
acquire() {
     while value <= 0
        ; // no-op
     value--;
}
   release() {
      value++;
   }
```

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - block – place the process invoking the operation on the appropriate waiting queue.
  - wakeup – remove one of processes in the waiting queue and place it in the ready queue.
- Negative semaphore values. (what does that mean?)



```
acquire(){
     value--;
     if (value < 0) {
        add this process to list
        block;
     }
}
```



```
release(){
     value++;
     if (value <= 0) {
        remove a process P from list
        wakeup(P);
     }
}
```

Semaphore Implementation

```
acquire() {
    while value <= 0
       ; // no-op
    value--;
}

   release() {
       value++;
   }
```

```
acquire(){
    value--;
    if (value < 0) {
       add this process to list
       block;
    }
}
```

```
release(){
    value++;
    if (value <= 0) {
       remove a process P from list
       wakeup(P);
    }
}
```

- The second major issue is how to implement acquire()/wait() and release()/signal() in an **atomic** manner
- Must guarantee that no two processes can execute acquire/wait and release/signal on the same semaphore at the same time
- In other words, their implementation becomes the critical section problem where the **acquire/wait** and **release/signal** codes are placed in the critical section.
  - Could now have busy waiting in critical section implementation because these implementation codes are short
  - We moved busy waiting from application to here; but, note that applications may spend lots of time in critical sections and therefore busy waiting is not a good solution there while OK here.

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

```
     P0                  P1

S.acquire();        Q.acquire();
Q.acquire();        S.acquire();
     .                   .
     .                   .
     .                   .
S.release();        Q.release();
Q.release();        S.release();
```

Application programmer must be careful!

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

Priority Inversion

- L < M < H
- L has resource x
- H wants resource x, but it will wait for this resource
- Now M becomes runnable and preempts L
- M will have higher priority than H !!!!

(Mars path finder had that problem)

Solutions

- Use only two priorities, but this not flexible
- Priority-inheritance
  - L will inherit Highest priority while using resource x. When it is finished, its priority will be reverted to the original one

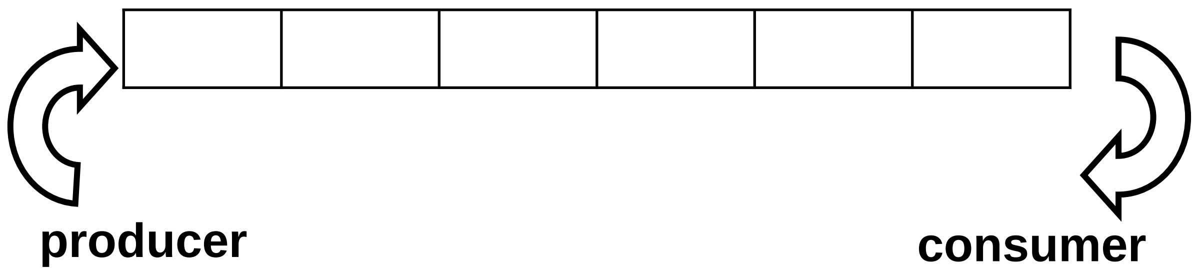
Bounded-Buffer (Consumer-Producer) Problem

Readers and Writers Problem

Dining-Philosophers Problem

CLASSICAL PROBLEMS OF SYNCHRONIZATION

Bounded-Buffer Problem for consumer-producer application



- Need to make sure that
  - The producer and the consumer do not access the buffer area and related variables at the same time
  - No item is made available to the consumer if all the buffer slots are empty.
  - No slot in the buffer is made available to the producer if all the buffer slots are full

What Semaphores are needed?

- `semaphore mutex, full, empty;`

What are the initial values?

Initially:

mutex = 1 // controlling mutual access to the buffer

full = 0 // The number of full buffer slots

empty = n // The number of empty buffer slots

Bounded buffer Codes

Insert an item

```
wait(empty); // init. n
wait(mutex);

add item to buffer

signal(mutex);
signal(full);
```

Remove an item

```
wait(full); // 0
wait(mutex);

remove an item from buffer

signal(mutex);
signal(empty);

return the item
```

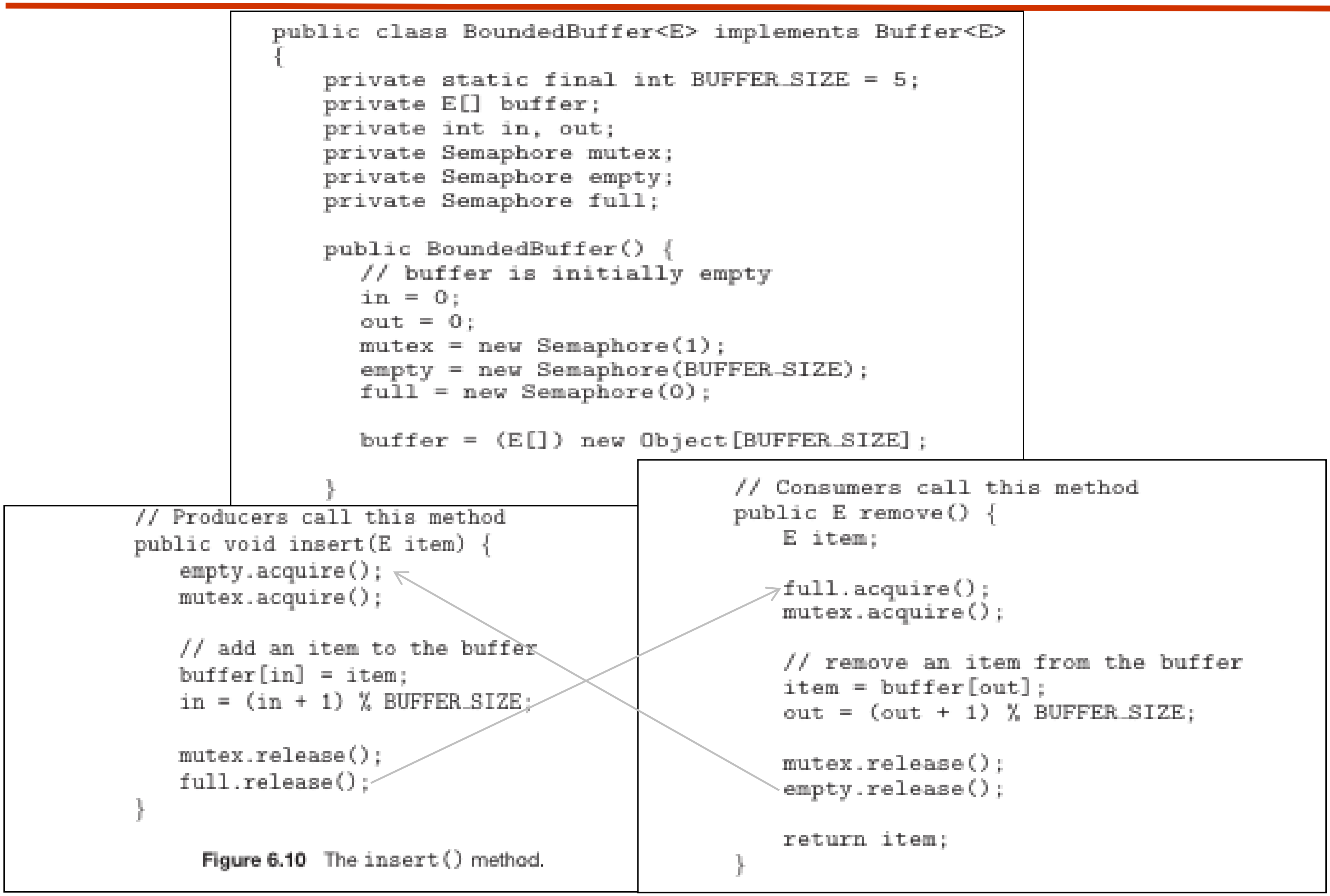
What will happen if we change the order?

What will happen if we change the order?

Be careful of the **sequence** of semaphore operations; → deadlock could happen as shown before;

The general rule: get the easy resource first, and then difficult; release in reverse order;

Bounded-Buffer Problem with Java



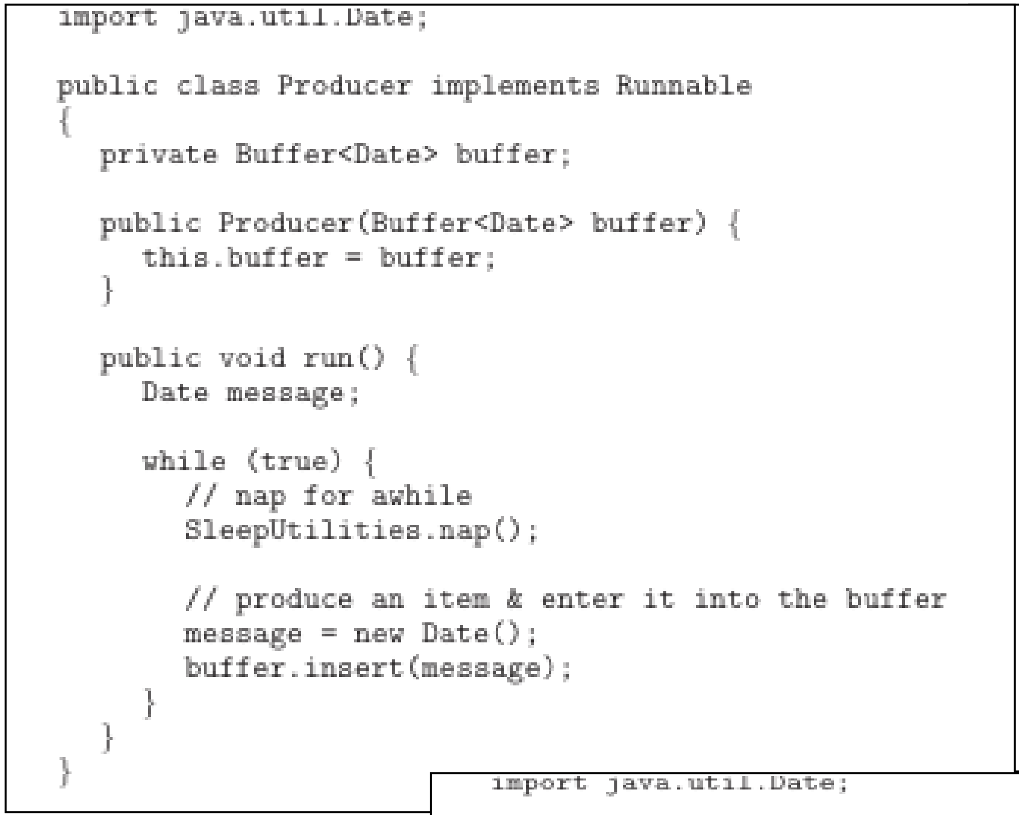
```
public class BoundedBuffer<E> implements Buffer<E>
{
    private static final int BUFFER_SIZE = 5;
    private E[] buffer;
    private int in, out;
    private Semaphore mutex;
    private Semaphore empty;
    private Semaphore full;

    public BoundedBuffer() {
       // buffer is initially empty
       in = 0;
       out = 0;
       mutex = new Semaphore(1);
       empty = new Semaphore(BUFFER_SIZE);
       full = new Semaphore(0);

       buffer = (E[]) new Object[BUFFER_SIZE];

    }
```

```
// Producers call this method
public void insert(E item) {
    empty.acquire();
    mutex.acquire();

    // add an item to the buffer
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;

    mutex.release();
    full.release();
}
```

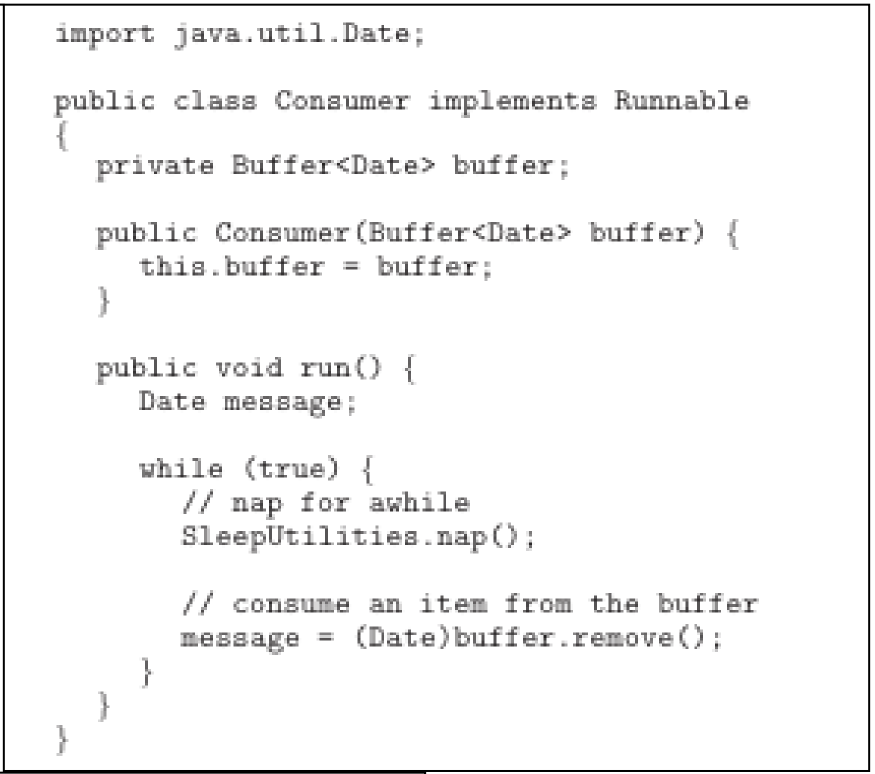
Figure 6.10 The insert() method.

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

    full.acquire();
    mutex.acquire();

    // remove an item from the buffer
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    mutex.release();
    empty.release();

    return item;
}
```

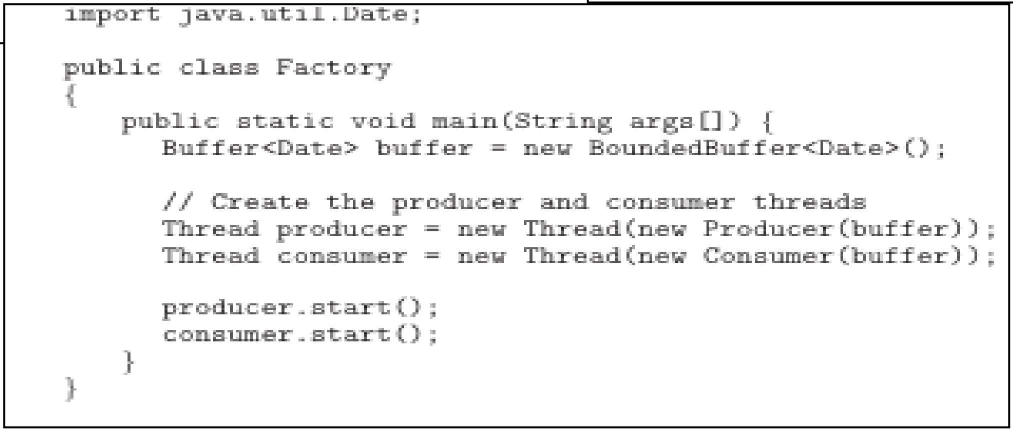
Producer and consumer application



```
import java.util.Date;

public class Producer implements Runnable
{
   private Buffer<Date> buffer;

   public Producer(Buffer<Date> 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);
      }
   }
}
```



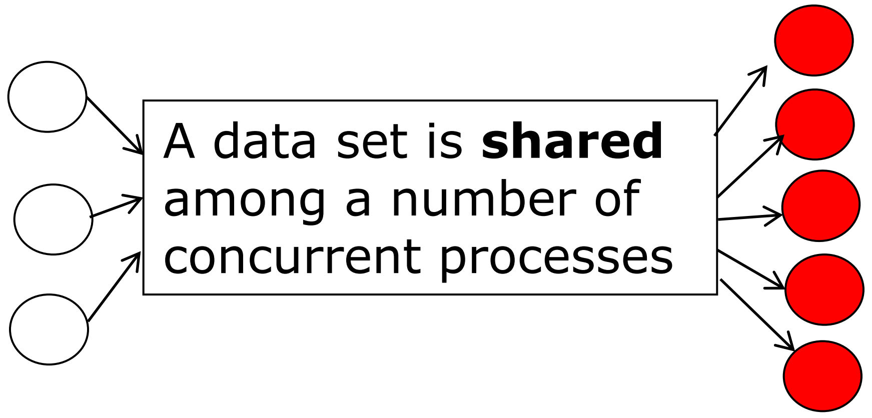
```
import java.util.Date;

public class Consumer implements Runnable
{
   private Buffer<Date> buffer;

   public Consumer(Buffer<Date> 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();
      }
   }
}
```



```
import java.util.Date;

public class Factory
{
   public static void main(String args[]) {
      Buffer<Date> buffer = new BoundedBuffer<Date>();

      // 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

Writers can both read and write, so they must have exclusive access



A data set is **shared** among a number of concurrent processes

Readers only read the data set; they do **not** perform any updates, so multiple readers may access the shared data simultaneously

- **Problem** – allow multiple readers to read at the same time; but only one single writer can access the shared data at the same time
- Shared Data
  - Data set
  - Integer readerCount initialized to 0
  - Semaphore mutex initialized to 1   //for readers to access readerCount
  - Semaphore db initialized to 1   //for writer/reader mutual exclusive

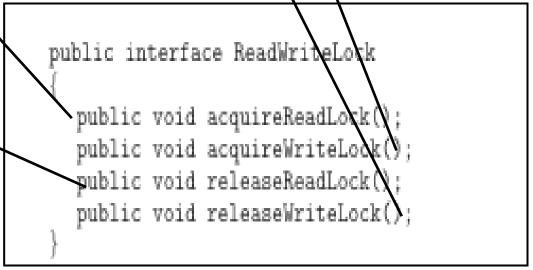
Reader   Writer

```
wait(mutex);
    readerCount++;
    if(readerCount == 1)
        wait(db);
signal(mutex);
 …
reading is performed
 …
wait(mutex);
    readerCount--;
    if(readcount == 0)
        signal(db);
signal(mutex);
```

```
wait(db);
…
writing is performed
…
signal(db);
```

Any problem with this solution?!

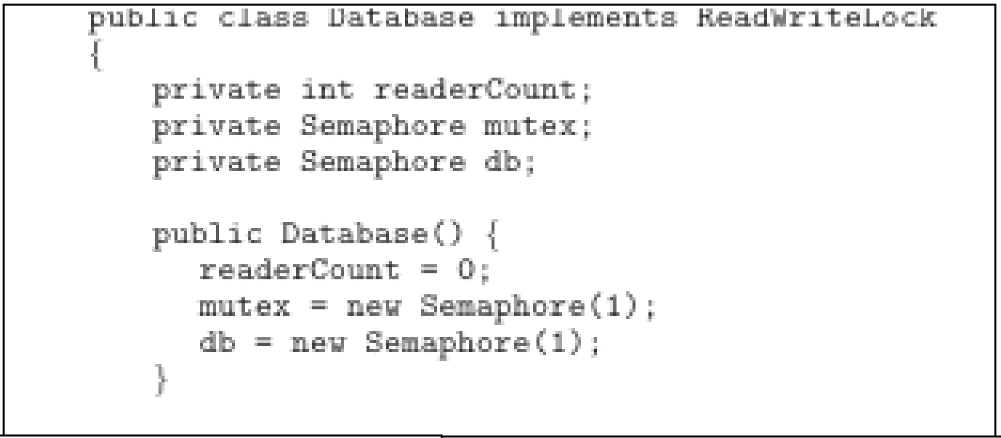
What happens if one reader gets in first?



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

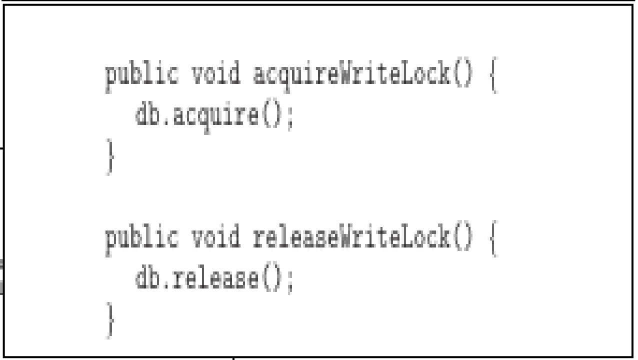
This solution is generalized as readers-writers lock… multiple processes acquire r lock but only one can acquire w lock

Readers-Writers Problem with Java



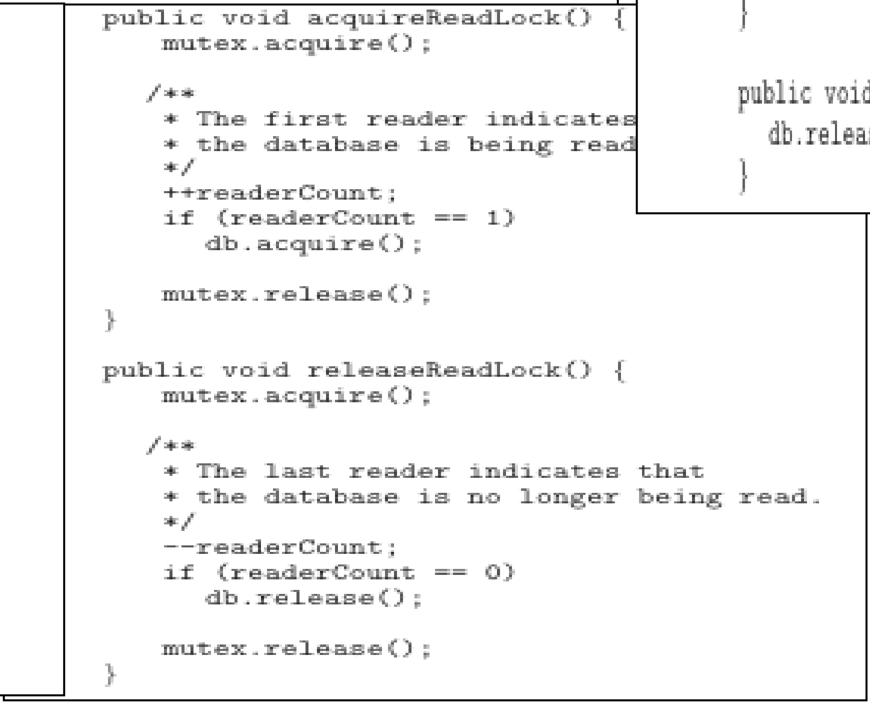
```
public class Database implements ReadWriteLock
{
   private int readerCount;
   private Semaphore mutex;
   private Semaphore db;

   public Database() {
      readerCount = 0;
      mutex = new Semaphore(1);
      db = new Semaphore(1);
   }
```

```
wait(db);
…
writing isperformed
…
signal(db);
```



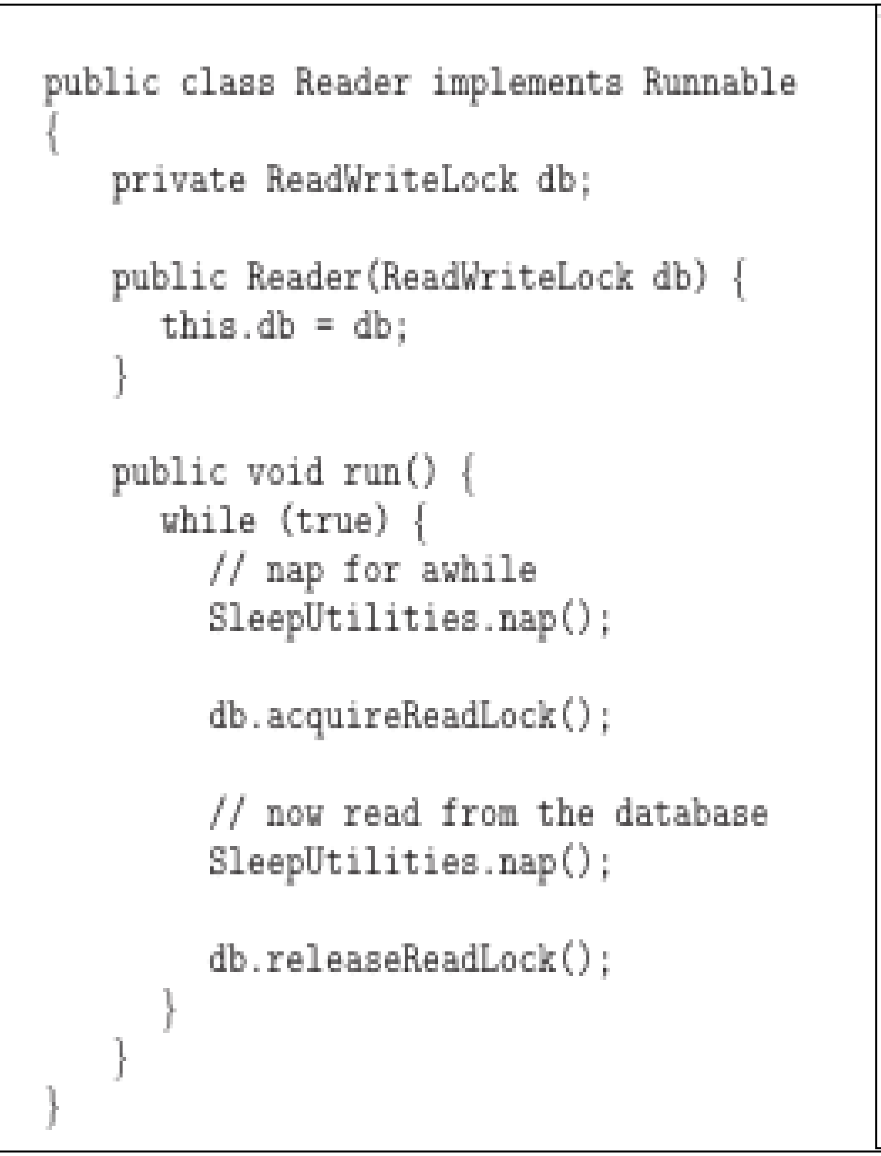
```
public void acquireWriteLock() {
   db.acquire();
}

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

```
wait(mutex);
   readerCount++;
   if(readerCount == 1)
        wait(db);
signal(mutex);
 …
reading is performed
 …
wait(mutex);
   readerCount--;
   if(readcount == 0)
        signal(db);
signal(mutex);
```

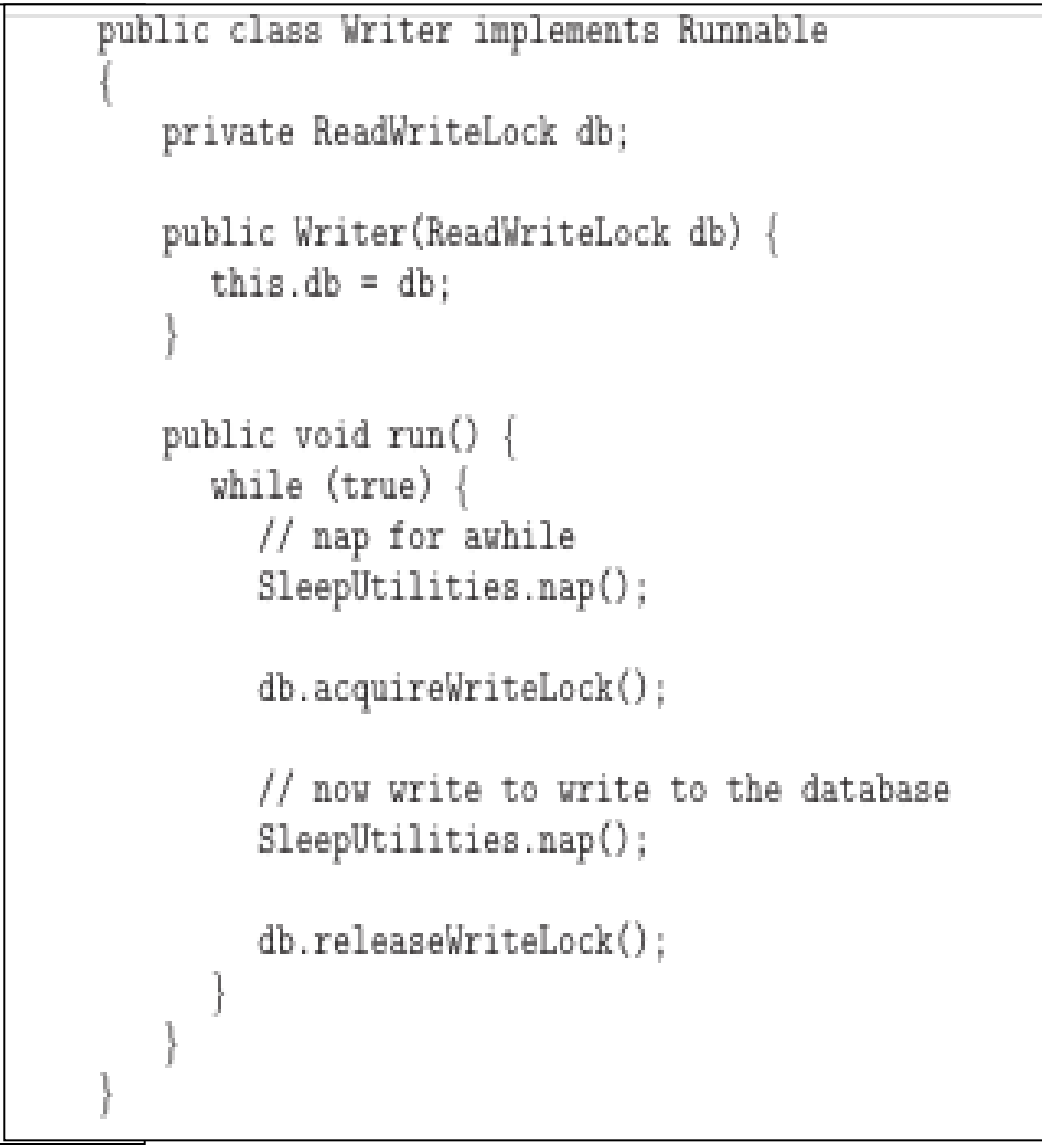


```
public void acquireReadLock() {
   mutex.acquire();

   /**
    * The first reader indicates
    * the database is being read
    */
   ++readerCount;
   if (readerCount == 1)
      db.acquire();

   mutex.release();
}

public void releaseReadLock() {
   mutex.acquire();

   /**
    * The last reader indicates that
    * the database is no longer being read.
    */
   --readerCount;
   if (readerCount == 0)
      db.release();

   mutex.release();
}
```

Readers-Writers Problem with Java (Cont.)



```
public class Reader implements Runnable
{
   private ReadWriteLock db;

   public Reader(ReadWriteLock db) {
      this.db = db;
   }

   public void run() {
      while (true) {
         // nap for awhile
         SleepUtilities.nap();

         db.acquireReadLock();

         // now read from the database
         SleepUtilities.nap();

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



```
public class Writer implements Runnable
{
   private ReadWriteLock db;

   public Writer(ReadWriteLock db) {
      this.db = db;
   }

   public void run() {
      while (true) {
         // nap for awhile
         SleepUtilities.nap();

         db.acquireWriteLock();

         // now write to write to the database
         SleepUtilities.nap();

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

*

Advanced Reader/Writer Problems

- Preferred Reader: original solution favors readers
- **Preferred Writer Problem**
  - If there is a writer in or **waiting**, no additional reader in
- Alternative Reader/Writer Problem
  - Reader/writer take turn to read/write

*

Preferred Writer: Variables & Semaphores

- Variables
  - readcount: number of readers current reading, init 0
  - writecount: number of writers current writing or waiting, init 0

- Semaphores
  - rmtx: reader mutex for updating readcount, init 1;
  - wmtx: writer mutex for updating writecount, init 1;

  - wsem: semaphore for exclusive writers, init 1;
  - rsem: semaphore for readers to wait for writers, init 1;
  - renter: semaphore for controlling readers getting in;

*

Preferred Writer: Solutions

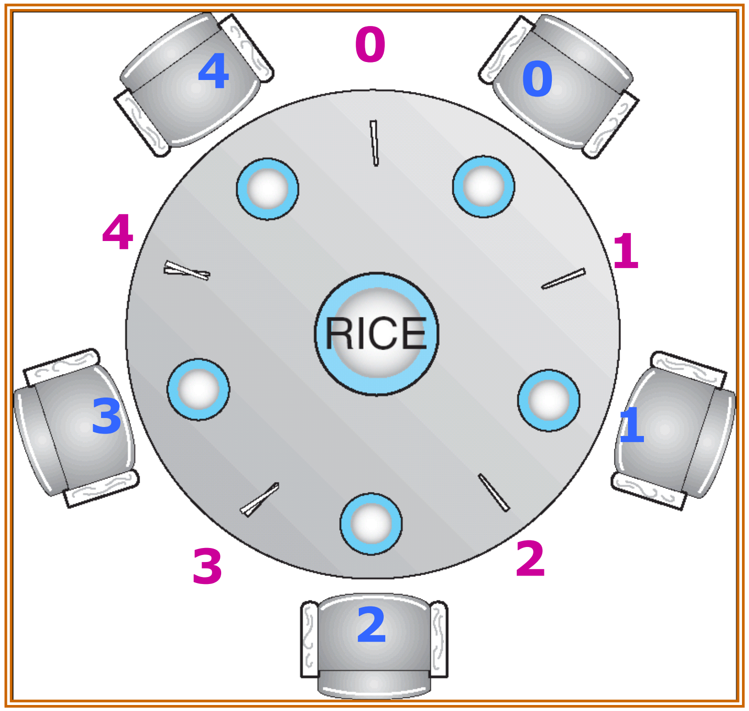
Reader

```
//wait(renter);
wait(rsem);
   wait(rmtx);
     readcount++;
     if (readcount = = 1)
       wait(wsem);
   signal(rmtx);
signal(rsem);
//signal(renter);
READING
wait(rmtx);
  readcount--;
  if (readcount == 0)
      signal(wsem);
signal(rmtx);
```

Writer

```
wait(wmtx);
  writecount++;
  if (writecount = = 1)
      wait(rsem);
signal(wmtx);

wait(wsem);
WRITING
signal(wsem);

wait(wmtx);
  writecount--;
  if (writecount == 0)
      signal(rsem);
signal(wmtx);
```

Dining-Philosophers Problem



Five philosophers share a common circular table. There are five chopsticks and a bowl of rice (in the middle).

When a philosopher gets **hungry**, he tries to pick up the closest chopsticks.

A philosopher may pick up only one chopstick at a time, and cannot pick up a chopstick already in use.

When done, he puts down both of his chopsticks, one after the other.

How to design a deadlock-free and starvation-free protocol….

- Shared data
  - Bowl of rice (data set)
  - *semaphore chopStick[5]; //* Initially all set to 1

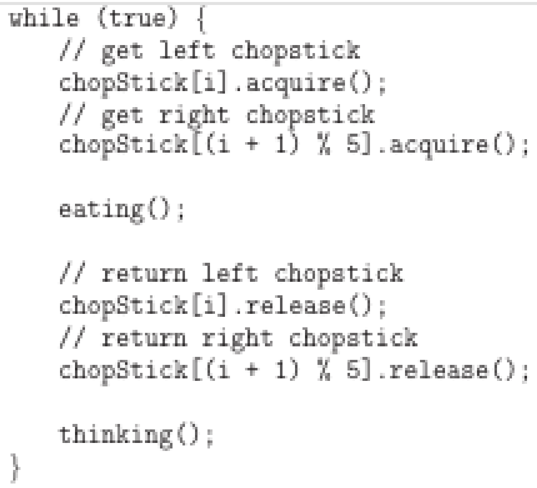
*

Dining-Philosophers Problem (cont.)

- **Solution** for the *i'th philosopher:*

```
while(1) {
   think; //and become hungry

   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
   eat
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
   …
}
```

Agreement:
- First, pick right chopstick
- Then, pick left chopstick



```
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();
}
```

What is the problem?!

Deadlock: Each one has one chopstick

Here are some options: allow at most 4 to sit, allow to pick up chopsticks if both are available, odd ones take left-then-right while even ones take right-then-left
Deadlock-free does not mean starvation-free (how about progress, and bounded waiting)

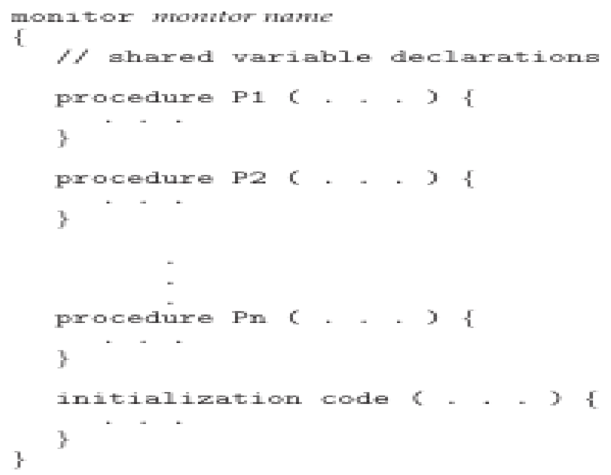
Programmers may mistake in the order of wait and signal and cause deadlock

A high-level abstraction that provides a convenient and effective mechanism for process synchronization

MONITORS

What are the problems with Sem?

- Wait/signal should be executed in the correct order; but, programmers may make mistake
- To deal with such mistakes, researchers developed **monitors** to provide a convenient and effective mechanism for process synchronization
- A monitor is a collection of procedures, variables, and data structures grouped together
- Only one process may be active within the monitor at a time
- A monitor is a language construct (e.g., synchronized methods in Java)
  - The compiler enforces mutual exclusion.
  - Semaphores are usually an OS construct



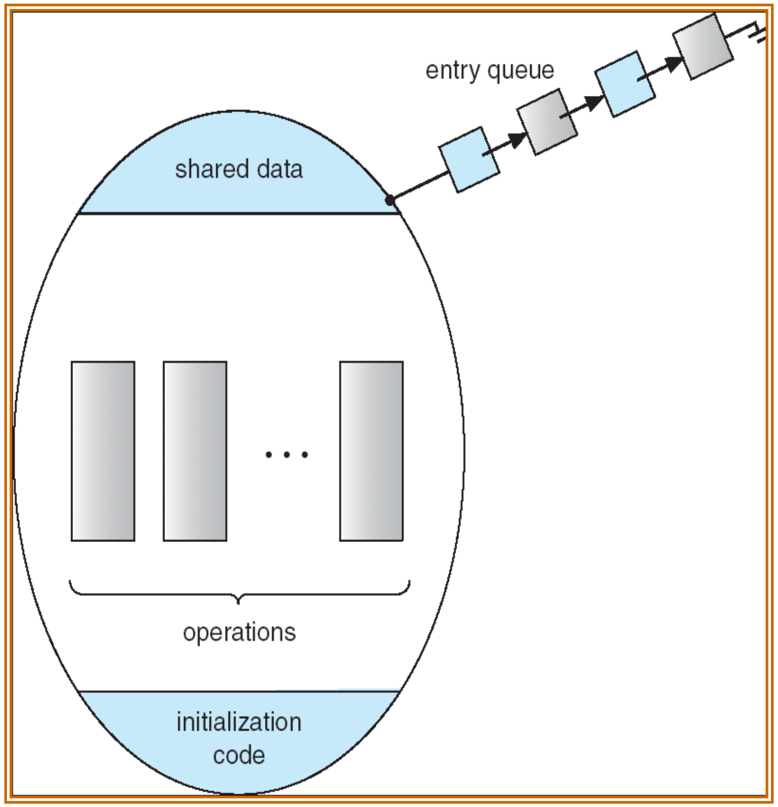
```
monitor monitor name
{
    // shared variable declarations
    procedure P1 ( . . . ) {
        . . .
    }
    procedure P2 ( . . . ) {
        . . .
    }
            .
            .
            .
    procedure Pn ( . . . ) {
        . . .
    }
    initialization code ( . . . ) {
        . . .
    }
}
```

```
public class SynchronizedCounter
{
  private int c = 0;
  public synchronized void inc(){
     c++;
  }
  public synchronized void dec(){
     c--;
  }
  public synchronized int value()
  {
     return c;
  }
}
```

*

Schematic View of a Monitor

- Monitor construct ensures at most one process/thread can be active within the monitor at a given time.
- Shared data (local variables) of the monitor can be accessed only by local procedures.
- So programmer does not need to code this synchronization constraints explicitly
- However, this is not sufficiently powerful , so for tailor-made *synchronization*, **condition variable** construct is provided

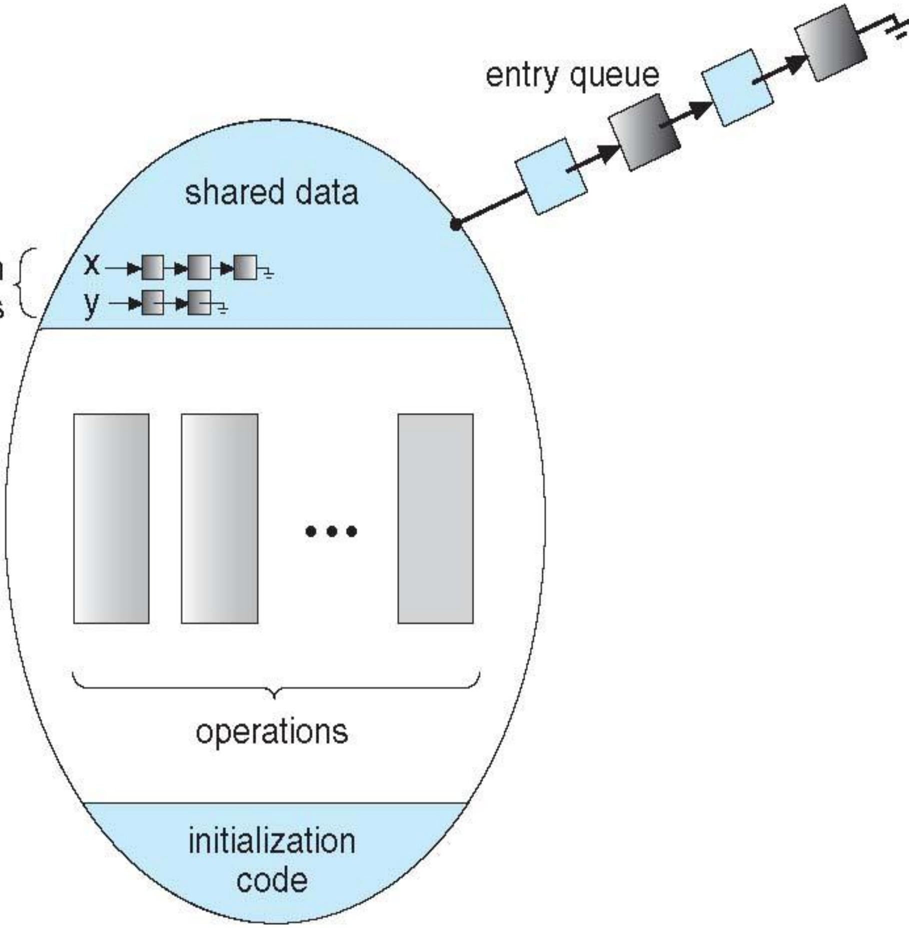


Condition Variables

- **Condition x, y**;

// a queue of blocked processes

- Two operations
  - **x.wait ()** means that the process invoking this operation is suspended until another process invokes x.signal(); *(similar to processes waiting for I/O to complete)*
  - **x.signal ()** resumes **exactly one** suspended process (if any) that invoked x.wait (); otherwise, no effect.
  - Monitor is not a counter



Signal and wait
Signal and continue

Many languages have some support,
We will see Java later

Solution to Dining Philosophers

- Each philosopher picks up chopsticks if both are available
- P_i can set state[i] to eating if her two neighbors are not eating
- We need condition self[i] so P_i can delay herself when she is hungry but cannot get chopsticks
- Each P_i invokes the operations **takeForks(i)** and **returnForks(i)** in the following sequence:

  dp.takeForks (i)

  EAT

  dp.returnForks (i)

- No deadlocks
- How about Starvation?
- http://vip.cs.utsa.edu/nsf/pubs/starving/starving.html

```
monitor DiningPhilosophers
{
   enum State {THINKING, HUNGRY, EATING};
   State[] states = new State[5];
   Condition[] self = new Condition[5];

   public DiningPhilosophers {
      for (int i = 0; i < 5; i++)
         state[i] = State.THINKING;
   }

   public void takeForks(int i) {
      state[i] = State.HUNGRY;
      test(i);
      if (state[i] != State.EATING)
         self[i].wait;
   }

   public void returnForks(int i) {
      state[i] = State.THINKING;
      // test left and right neighbors
      test((i + 4) % 5);
      test((i + 1) % 5);
   }

   private void test(int i) {
      if ( (state[(i + 4) % 5] != State.EATING) &&
         (state[i] == State.HUNGRY) &&
         (state[(i + 1) % 5] != State.EATING) ) {
            state[i] = State.EATING;
            self[i].signal;
      }
   }
}
```

.

Exercise

Producer-Consumer (PC) Monitor

```
void producer()   {
   //Producer process
   while (1) {
      item=Produce_Item();
      PC.insert(item);
   }
}
```

```
void consumer(){
   //Consumer process
   while (1) {
      item = PC.remove();
      consume_item(item);
    }
}
```

```
Monitor PC {
  condition full, empty;
  int count;

  void init() {
    count = 0;
  }

  void insert(int item){
   if (count==N) full.wait();
   insert_item(item);
   count++;
   if (count==1) empty.signal();
 }

 int remove(){
   int m;
   if (count==0)  empty.wait();
   m = remove_item();
   count--;
   if (count==N-1) full.signal();
   return m;
 }
```

compiler
will do this

Monitor Implementation

- Monitors are implemented by using queues to keep track of the processes attempting to become active in the monitor.
- To be active, a monitor must obtain a **lock** to allow it to execute.
- Processes that are blocked are put in a queue of processes waiting for an unblocking event to occur.
  - **The entry queue** contains processes attempting to call a monitor procedure from outside the monitor. Each monitor has one entry queue.
  - **The signaller queue** contains processes that have executed a notify operation. Each monitor has at most one signaller queue. In some implementations, a notify leaves the process active and no signaller queue is needed.
  - **The waiting queue** contains processes that have been awakened by a notify operation. Each monitor has one waiting queue.
  - **Condition variable queues** contain processes that have executed a condition variable wait operation. There is one such queue for each condition variable.
- The relative priorities of these queues determines the operation of the monitor implementation.

JAVA SYNCHRONIZATION

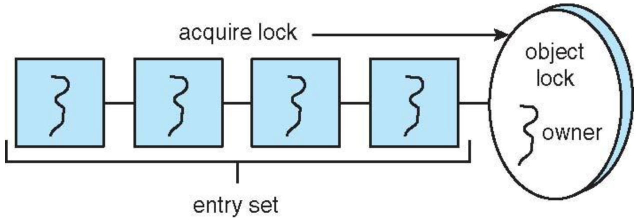
http://www.caveofprogramming.com/tag/multithreading/

Recall Bounded Buffer

- Busy waiting loops when buffer is full or empty
- Shared variable `count` may develop race condition
- We will see how to solve these problems using Java sync
- **Busy waiting** can be removed by blocking a process
  - `while(full/empty);` vs.
  - `while(full/empty) Thread.yield();`
  - Problem: livelock
    - (e.g., a process with high priority waits here while another low priority process tries to update full/empty)
    - We will see there is a better alternative than busy waiting or yielding
- Race condition
  - Can be solved using synchronized methods (see next page)

Java Synchronization

- Java provides synchronization at the language-level.
- Each Java object has an associated lock.
- This lock is acquired by invoking a **synchronized** method.
- This lock is released when exiting the **synchronized** method.
- Threads waiting to acquire the object lock are placed in the **entry set** for the object lock.

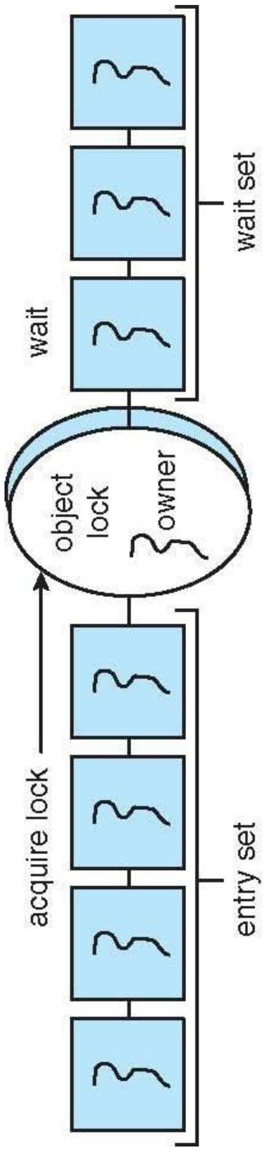


```
// Producers call this method
public synchronized void insert(E item) {
    while (count == BUFFER_SIZE)
      Thread.yield();

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


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

    while (count == 0)
      Thread.yield();

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

    return item;
}
```

Incorrect! Why?

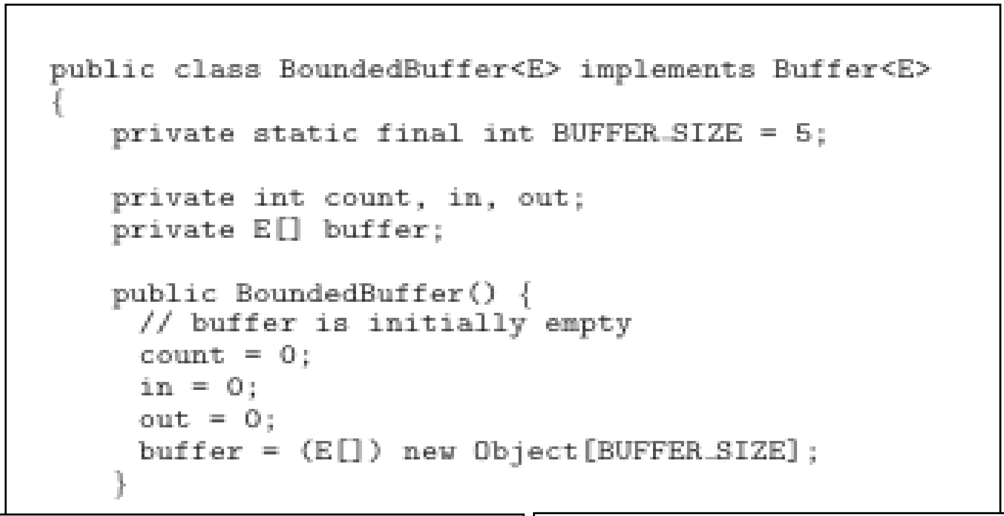
Java Synchronization

- Why the previous solution is incorrect?
  - Suppose buffer is full and consumer is sleeping.
  - Producer call `insert()`, gets the lock and then yields. But it still has the lock,
  - So when consumer calls `remove()`, it will block because the lock is owned by producer …
  - Deadlock
- We can solve this problem using two new Java methods
  - When a thread invokes **wait():**
    1. The thread releases the object lock;
    2. The state of the thread is set to Blocked;
    3. The thread is placed in the **wait set** for the object.
  - When a thread invokes **notify()**:
    1. An arbitrary thread T from the wait set is selected;
    2. T is moved from the wait to the entry set;
    3. The state of T is set to Runnable.



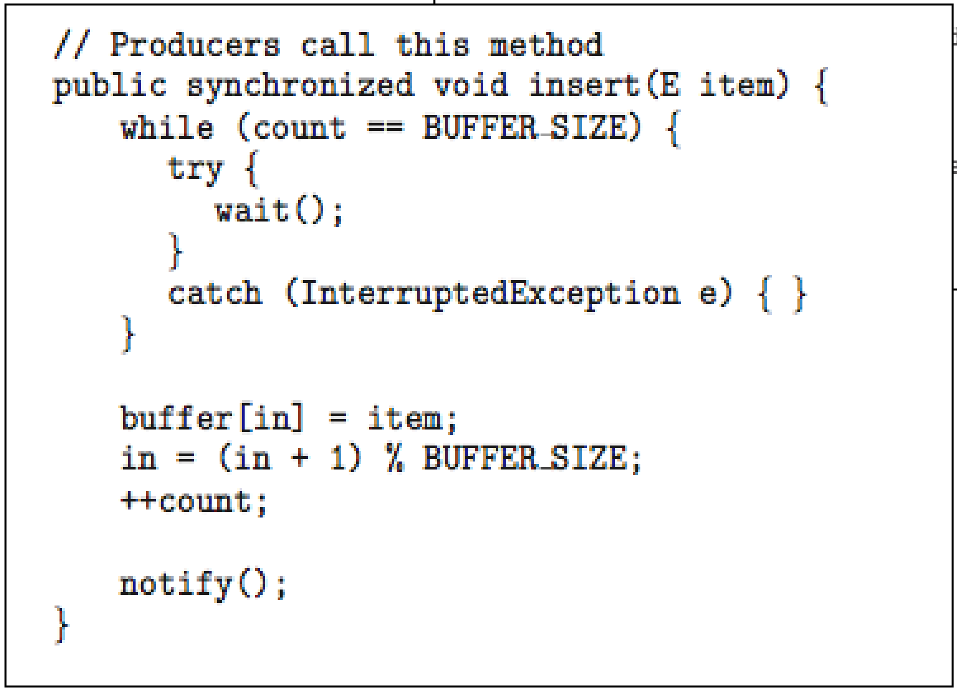
Java Synchronization - Bounded Buffer

Correct! Why?

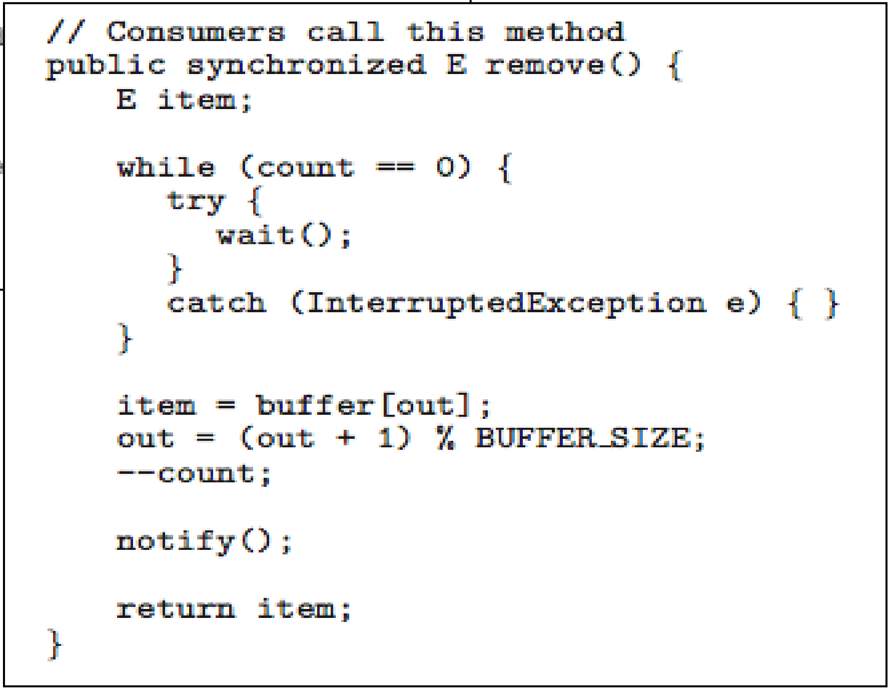


```
public class BoundedBuffer<E> implements Buffer<E>
{
    private static final int BUFFER_SIZE = 5;

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

    public BoundedBuffer() {
      // buffer is initially empty
      count = 0;
      in = 0;
      out = 0;
      buffer = (E[]) new Object[BUFFER_SIZE];
    }
```

Can be used instead of the one in slide 45 (see next for its copy)

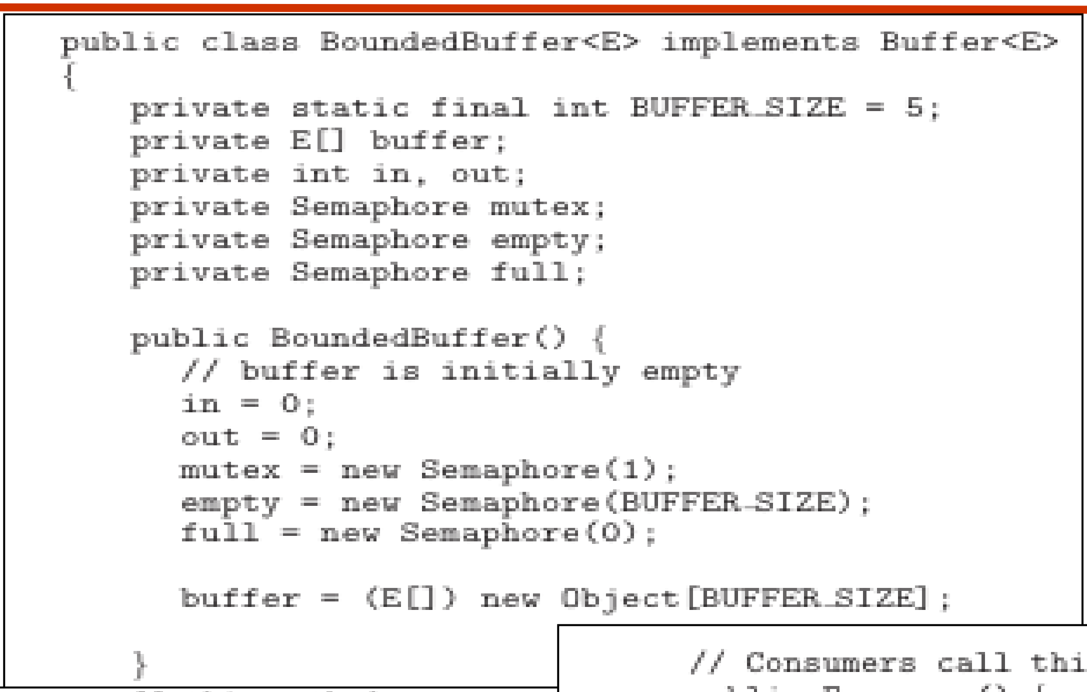


```
// Producers call this method
public synchronized void insert(E item) {
    while (count == BUFFER_SIZE) {
       try {
          wait();
       }
       catch (InterruptedException e) { }
    }

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

    notify();
}
```



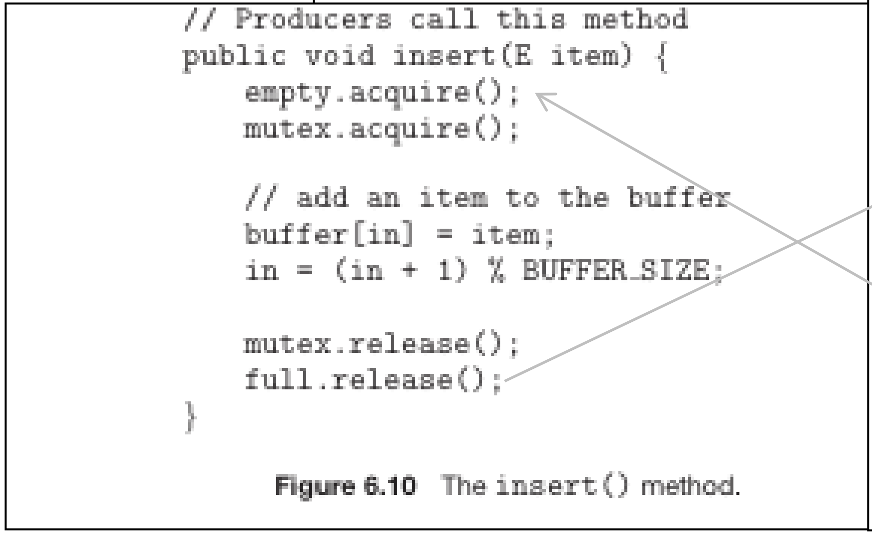
```
// Consumers call this method
public synchronized E remove() {
    E item;

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

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

    notify();

    return item;
}
```

Bounded-Buffer Problem with Java



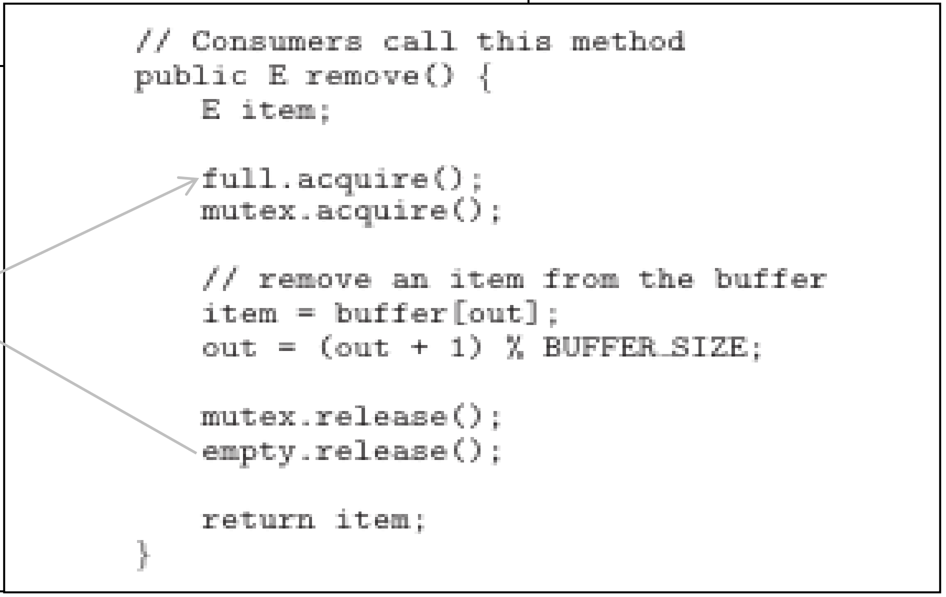
```
public class BoundedBuffer<E> implements Buffer<E>
{
    private static final int BUFFER_SIZE = 5;
    private E[] buffer;
    private int in, out;
    private Semaphore mutex;
    private Semaphore empty;
    private Semaphore full;

    public BoundedBuffer() {
       // buffer is initially empty
       in = 0;
       out = 0;
       mutex = new Semaphore(1);
       empty = new Semaphore(BUFFER_SIZE);
       full = new Semaphore(0);

       buffer = (E[]) new Object[BUFFER_SIZE];

    }
```

Covered before in slide 45



```
// Producers call this method
public void insert(E item) {
    empty.acquire();
    mutex.acquire();

    // add an item to the buffer
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;

    mutex.release();
    full.release();
}
```

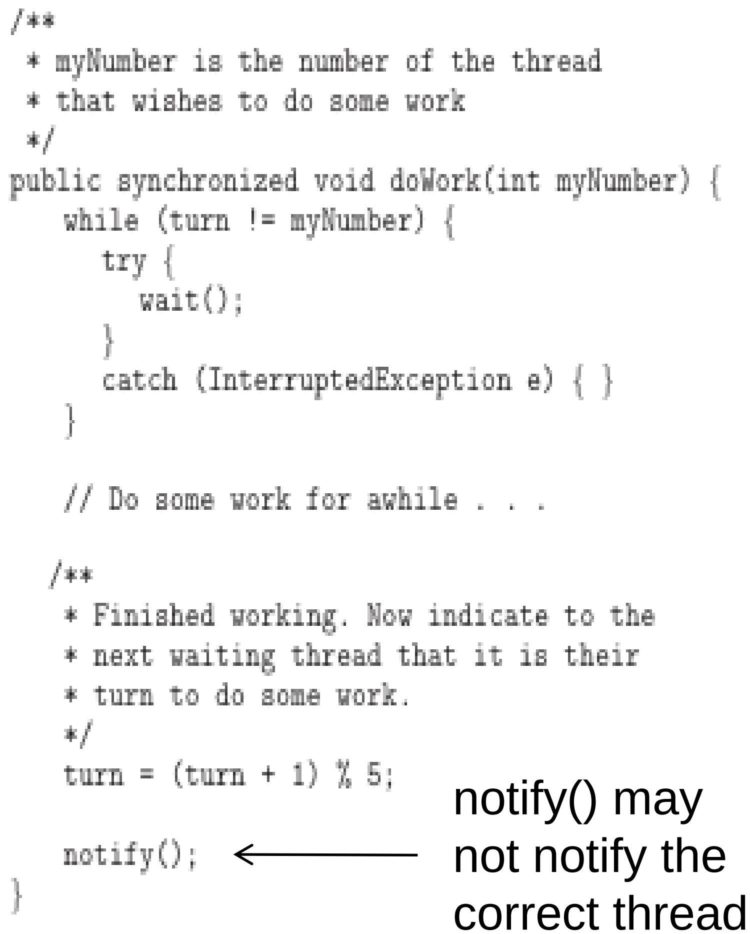
Figure 6.10 The `insert()` method.



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

    full.acquire();
    mutex.acquire();

    // remove an item from the buffer
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    mutex.release();
    empty.release();

    return item;
}
```

Java Synchronization

- The call to **notify()** selects an arbitrary thread from the wait set.
  - It is possible the selected thread is in fact not waiting upon the condition for which it was notified.
  - Consider doWork():
    - turn is 3, T1, T2, T4 are in wait set, and T3 is in doWork()
    - What happens when T3 is done?
- The call **notifyAll()** selects all threads in the wait set and moves them to the entry set.
- In general, **notifyAll()** is a more conservative strategy than **notify()**.



```
/**
 * myNumber is the number of the thread
 * that wishes to do some work
 */
public synchronized void doWork(int myNumber) {
   while (turn != myNumber) {
      try {
         wait();
      }
      catch (InterruptedException e) { }
   }

   // Do some work for awhile . . .

   /**
    * Finished working. Now indicate to the
    * next waiting thread that it is their
    * turn to do some work.
    */
   turn = (turn + 1) % 5;

   notify();
}
```

notify() may not notify the correct thread!

Java Synchronization - Readers-Writers

using both notify() and notifyAll()

```
public class Database implements ReadWriteLock
{
   private int readerCount;
   private boolean dbWriting;

   public Database() {
      readerCount = 0;
      dbWriting = false;
   }
```

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

   ++readerCount;
}

public synchronized void releaseReadLock() {
   --readerCount;

  /**
   * The last reader indicates 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) { }
   }

  /**
   * Once there are no readers or a writer,
   * indicate that the database is being written.
   */
   dbWriting = true;
}

public synchronized void releaseWriteLock() {
   dbWriting = false;

   notifyAll();
}
```

Java Synchronization

Block synchronization

- Rather than synchronizing an entire method, **Block synchronization** allows blocks of code to be declared as synchronized
- This will be also necessary if you need more than one locks to share different resources

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

    synchronized(mutexLock) {
       criticalSection();
    }

    remainderSection();
}
```

Java Synchronization

- Block synchronization using wait()/notify()

```
Object mutexLock = new Object();
. . .
synchronized(mutexLock) {
    try {
       mutexLock.wait();
    }
    catch (InterruptedException ie) { }
}

synchronized(mutexLock) {
    mutexLock.notify();
}
```

Concurrency Features in Java 5

- Prior to Java 5, the only concurrency features in Java were Using synchronized/wait/notify.
- Beginning with Java 5, new features were added to the API:
  - Reentrant Locks
  - Semaphores
  - Condition Variables

Concurrency Features in Java 5

- Reentrant Locks

```
Lock key = new ReentrantLock();

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

Concurrency Features in Java 5

- Semaphores

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

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

Concurrency Features in Java 5

- A condition variable is created by first creating a **ReentrantLock** and invoking its **newCondition()** method:

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

- Once this is done, it is possible to invoke the **await()** and **signal()** methods

Concurrency Features in Java 5

- doWork() method with condition variables

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

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

    // Do some work for awhile . . .

    /**
     * Finished working. Now indicate to the
     * next waiting thread that it is their
     * turn to do some work.
     */

    turn = (turn + 1) % 5;
    condVars[turn].signal();
  }
  catch (InterruptedException ie) { }
  finally {
    lock.unlock();
  }
}
```

EXTRAS

Pthreads

Pthread Library
is for reference and self-study

Solaris

Windows XP

Linux

SYNCHRONIZATION EXAMPLES

Pthreads Synchronization

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

*

Synchronization in Pthread Library

- Mutex variables
  - *pthread_mutex_t*
- Conditional variables
  - *pthread_cond_t*

- All POSIX thread functions have the form:

  pthread[_object] _operation
- Most of the POSIX thread library functions return 0 in case of success and some non-zero error-number in case of a failure

*

Mutex Variables: Mutual Exclusion

- A mutex variable can be either ***locked*** or ***unlocked***
  - *pthread_mutex_t lock;* // lock is a mutex variable

- Initialization of a mutex variable by default attributes
  - *pthread_mutex_init(&lock, NULL);*

- Lock operation
  - *pthread_mutex_lock(&lock) ;*

- Unlock operation
  - *pthread_mutex_unlock(&lock)*

*

Condition Variables

- In a critical section, a thread can suspend itself on a *condition variable* if the state of the computation is not right for it to proceed.
  - It will suspend by *waiting* on a condition variable.
  - It will, however, release the critical section lock .
  - When that condition variable is *signaled,* it will become ready again; it will attempt to reacquire that critical section lock and only then will be able proceed.

- **With POSIX threads, a condition variable can be associated with only one mutex variable**

*

Condition Variables (cont.)

- *pthread_cond_t   SpaceAvailable;*
- *pthread_cond_init (&SpaceAvailable, NULL);*

- *pthread_cond_wait*
- *pthread_cond_signal*
  - unblock one waiting thread on that condition variable

- *pthread_cond_broadcast*
  - unblock all waiting threads on that condition variable

*

Example: Producer-Consumer Problem

- Producer will produce a sequence of integers, and deposit each integer in a bounded buffer

  (implemented as an array).
- All integers are positive, 0..999.
- Producer will deposit -1 when finished, and then

  terminate.
- Buffer is of finite size: 5 in this example.
- Consumer will remove integers, one at a time, and print them.
- It will terminate when it receives -1.

*

Example: Definitions and Global

```
#include<pthread.h>
#include<stdio.h>
#include<string.h>
const int N = 5;
int Buffer[5];
int in = 0;
int out = 0;
int count = 0;
pthread_mutex_t lock;
pthread_cond_t SpaceAvailable, ItemAvailable;
```

*

Example: Producer Thread

```
void * producer (void *arg){
    int i;
    for ( i = 0; i< 1000; i++) {
      pthread_mutex_lock ( &lock); /* Enter critical section */
      while ( count == N ) /* Make sure that buffer is NOT full */
        pthread_cond_wait ( &SpaceAvailable, &lock) ;
         /* Sleep using a condition variable */
         /* now  count must be less than N */
      Buffer[in] = i; /* Put item in the buffer using "in" */
      in = (in + 1) % N;
      count++; /* Increment the count of items in the buffer */
```

*

Example: Producer Thread (Cont.)

```
        pthread_mutex_unlock ( &lock);
        pthread_cond_signal( &ItemAvailable );
        /* Wakeup consumer, if waiting */
    }  /* End of For loop */
    /* Put -1 in the buffer to indicate completion to the consumer */
    pthread_mutex_lock ( &lock);
    while ( count == N )
        pthread_cond_wait( &SpaceAvailable, &lock) ;
    Buffer[in] = -1; in = (in + 1) % N; count++;
    pthread_mutex_unlock ( &lock );
    pthread_cond_signal( &ItemAvailable );
    /* Wakeup consumer, if waiting */
} // End of producer
```

*

Example: Consumer Thread

```
void * consumer (void *arg){
    int i = 0;
    do {
        pthread_mutex_lock ( &lock); /* Enter critical section */
        while ( count == 0 )
         /* Make sure that buffer is NOT empty */
                pthread_cond_wait( &ItemAvailable, &lock) ;
                /* Sleep using a condition variable */
        /* count must be > 0 */
        i = Buffer[out] ; /* Remove item from buffer using "out" */
        out = (out + 1) % N;
        count--; /* Decrement the count of items in the buffer */
```

*

Example: Consumer Thread (Cont.)

```
        printf( "Removed %d \n", i);
        pthread_mutex_unlock ( &lock); /* exit critical 
    section */
        pthread_cond_signal( &SpaceAvailable);
        /* Wakeup producer, if waiting */
    } while ( i != -1 );  /* End of Do loop */
} // End of consumer
```

*

Example: Main program

```
main( ) {
    pthread_t    prod, cons; /* thread variables */
    int   n;
    pthread_mutex_init( &lock, NULL);
    pthread_cond_init (&SpaceAvailable, NULL);
    pthread_cond_init (&ItemAvailable, NULL);
    /* Create producer thread */
    if ( n = pthread_create(&prod, NULL, producer ,NULL)) {
            fprintf(stderr,"pthread_create :%s\n",strerror(n));
            exit(1);
    }
```

*

Example: Main Program (Cont.)

```
    /* Create consumer thread */
    if ( n = pthread_create(&cons, NULL, consumer, NULL) ) {
            fprintf(stderr,"pthread_create :%s\n",strerror(n));
            exit(1);
    }
    /* Wait for the consumer thread to finish. */
    if ( n = pthread_join(cons, NULL) ) {
            fprintf(stderr,"pthread_join:%s\n",strerror(n));
            exit(1);
    }
    printf("Finished execution \n" );
} // End of main
```

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:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - semaphores
  - spin locks

More later in DS

System Model

Log-based Recovery

Checkpoints

Concurrent Atomic Transactions

ATOMIC TRANSACTIONS

Transactional Memory

- **Memory transaction** is a series of read-write operations that are atomic.
- We replace

```
update () {
   acquire();
   /* modify shared data */
   release();
}
```

- With

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

- The **atomic{S}** statement ensures the statements in **S** execute as a transaction.

System Model

- 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 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

Types of Storage Media

- Volatile storage – information stored here does not survive system crashes
  - Example: main memory, cache
- Nonvolatile storage – Information usually survives crashes
  - Example: disk and tape
- Stable storage – Information never lost
  - Not actually possible, so approximated via replication or RAID to devices with independent failure modes

Goal is to assure transaction atomicity where failures cause loss of information on volatile storage

Log-Based Recovery

- Record to stable storage information about all modifications by a transaction
- Most common is **write-ahead logging**
  - Log on stable storage, each log record describes single transaction write operation, including
    - Transaction name
    - Data item name
    - Old value
    - New value
  - <T_i starts> written to log when transaction T_i starts
  - <T_i commits> written when T_i commits
- Log entry must reach stable storage before operation on data occurs

Log-Based Recovery Algorithm

- Using the log, system can handle any volatile memory errors
  - **Undo(T_i)** restores value of all data updated by T_i
  - **Redo(T_i)** sets values of all data in transaction T_i to new values
- Undo(T_i) and redo(T_i) must be **idempotent**
  - Multiple executions must have the same result as one execution
- If system fails, restore state of all updated data via log
  - If log contains <T_i starts> without <T_i commits>, **undo(T_i)**
  - If log contains <T_i starts> and <T_i commits>, **redo(T_i)**

Checkpoints

- Log could become long, and recovery could take long
- Checkpoints shorten log and recovery time.
- Checkpoint scheme:
  1. Output all log records currently in volatile storage to stable storage
  2. Output all modified data from volatile to stable storage
  3. Output a log record <checkpoint> to the log on stable storage
- Now recovery only includes Ti, such that Ti started executing before the most recent checkpoint, and all transactions after Ti All other transactions already on stable storage

Concurrent Transactions

- Must be equivalent to serial execution – **serializability**
- Could perform all transactions in critical section
  - Inefficient, too restrictive
- **Concurrency-control algorithms** provide serializability

Serializability

- Consider two data items A and B
- Consider Transactions T_0 and T_1
- Execute T_0, T_1 atomically
- Execution sequence called schedule
- Atomically executed transaction order called serial schedule
- For N transactions, there are N! valid serial schedules

Schedule 1: T_0 then T_1

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Nonserial Schedule

- **Nonserial schedule** allows overlapped execute
  - Resulting execution not necessarily incorrect
- Consider schedule S, operations O_i, O_j
  - **Conflict** if access same data item, with at least one write
- If O_i, O_j consecutive and operations of different transactions & O_i and O_j don't conflict
  - Then S' with swapped order O_j O_i equivalent to S
- If S can become S' via swapping nonconflicting operations
  - S is **conflict serializable**

Schedule 2: Concurrent Serializable Schedule

T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Locking Protocol

- Ensure serializability by associating lock with each data item
  - Follow locking protocol for access control
- Locks
  - **Shared** – T_i has shared-mode lock (S) on item Q, T_i can read Q but not write Q
  - **Exclusive** – Ti has exclusive-mode lock (X) on Q, T_i can read and write Q
- Require every transaction on item Q acquire appropriate lock
- If lock already held, new request may have to wait
  - Similar to readers-writers algorithm

Two-phase Locking Protocol

- Generally ensures conflict serializability
- Each transaction issues lock and unlock requests in two phases
  - Growing – obtaining locks
  - Shrinking – releasing locks
- Does not prevent deadlock

Timestamp-based Protocols

- Select order among transactions in advance – **timestamp-ordering**
- Transaction T_i associated with timestamp $TS(T_i)$ before T_i starts
  - $TS(T_i) < TS(T_j)$ if Ti entered system before T_j
  - TS can be generated from system clock or as logical counter incremented at each entry of transaction
- Timestamps determine serializability order
  - If $TS(T_i) < TS(T_j)$, system must ensure produced schedule equivalent to serial schedule where T_i appears before T_j

Timestamp-based Protocol Implementation

- Data item Q gets two timestamps
  - W-timestamp(Q) – largest timestamp of any transaction that executed write(Q) successfully
  - R-timestamp(Q) – largest timestamp of successful read(Q)
  - Updated whenever read(Q) or write(Q) executed
- **Timestamp-ordering protocol** assures any conflicting read and write executed in timestamp order
- Suppose Ti executes read(Q)
  - If $TS(T_i)$ < W-timestamp(Q), Ti needs to read value of Q that was already overwritten
    - read operation rejected and T_i rolled back
  - If $TS(T_i) \geq$ W-timestamp(Q)
    - read executed, R-timestamp(Q) set to max(R-timestamp(Q), $TS(T_i)$)

Timestamp-ordering Protocol

- Suppose Ti executes write(Q)
  - If TS(T_i) < R-timestamp(Q), value Q produced by T_i was needed previously and T_i assumed it would never be produced
    - Write operation rejected, T_i rolled back
  - If TS(T_i) < W-tiimestamp(Q), T_i attempting to write obsolete value of Q
    - Write operation rejected and T_i rolled back
  - Otherwise, write executed
- Any rolled back transaction T_i is assigned new timestamp and restarted
- Algorithm ensures conflict serializability and freedom from deadlock

Schedule Possible Under Timestamp Protocol

T_2	T_3
read(B)	
	read(B)
	write(B)
read(A)	
	read(A)
	write(A)

End of Chapter 6