CMPS 405: OPERATING SYSTEMS

Synchronization

This material is based on the operating system books by Silberschatz and Stallings and other Linux and system programming books.

Objectives

- Motivations
  - Concurrent Programming
    - **Producer/Consumer Problem (PCP)**
  - Synchronization Terminology
  - Cooperative Processes Interaction Structure
  - The Concept of a Lock
  - Conditions for a Valid Solution

Objectives

- Implicit reentrant locks using **synchronized** keyword
  - Solutions of PCP
    - using **synchronized** with **yield** method of **Thread**
    - using **synchronized** with **wait/notify** and **wait/notifyAll** method s pairs of **Object**

- Explicit reentrant lock in **java.util.concurrent.locks**
  - Solution of PCP
    - using **ReentrantLock** with **Condition** classes of the package **java.util.concurrent.locks**

Objectives

- Synchronization Semaphores
  - The concept of semaphores
  - The **Semaphore** class in **java.util.concurrent.locks** package
    - Solutions to the Dining Philosophers Problem (DPP) using semaphores.
    - Solution of the Producer/Consumer Problem using semaphores.

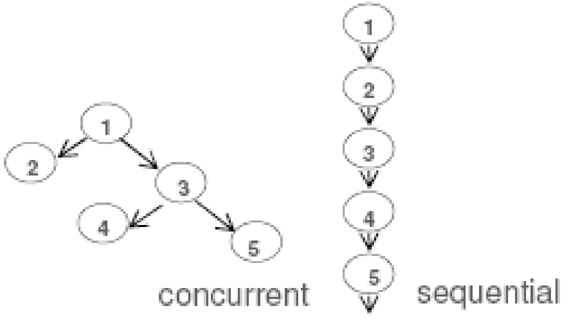
Concurrent Programming

- In sequential programming: all computational tasks are executed in sequence, one after the other.

- In concurrent programming: multiple computational tasks are executed simultaneously, at the same time.

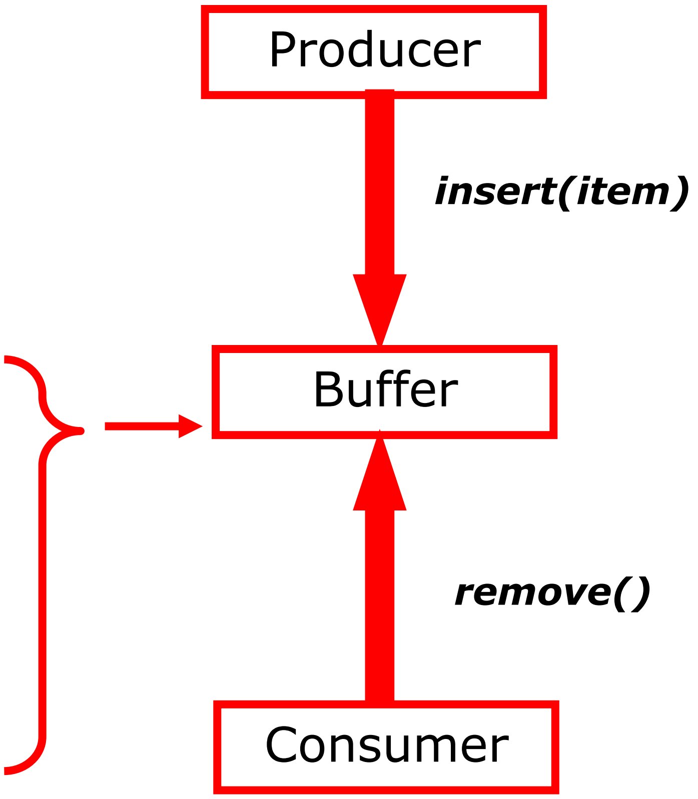
- Implementation of concurrent tasks:
  - as separate programs
  - as a set of processes or threads created by a single program

- Execution of concurrent tasks:
  - on a single processor
    - ***Multithreaded programming***
  - on several processors in close proximity
    - ***Parallel computing***
  - on several processors distributed across a network
    - ***Distributed computing***

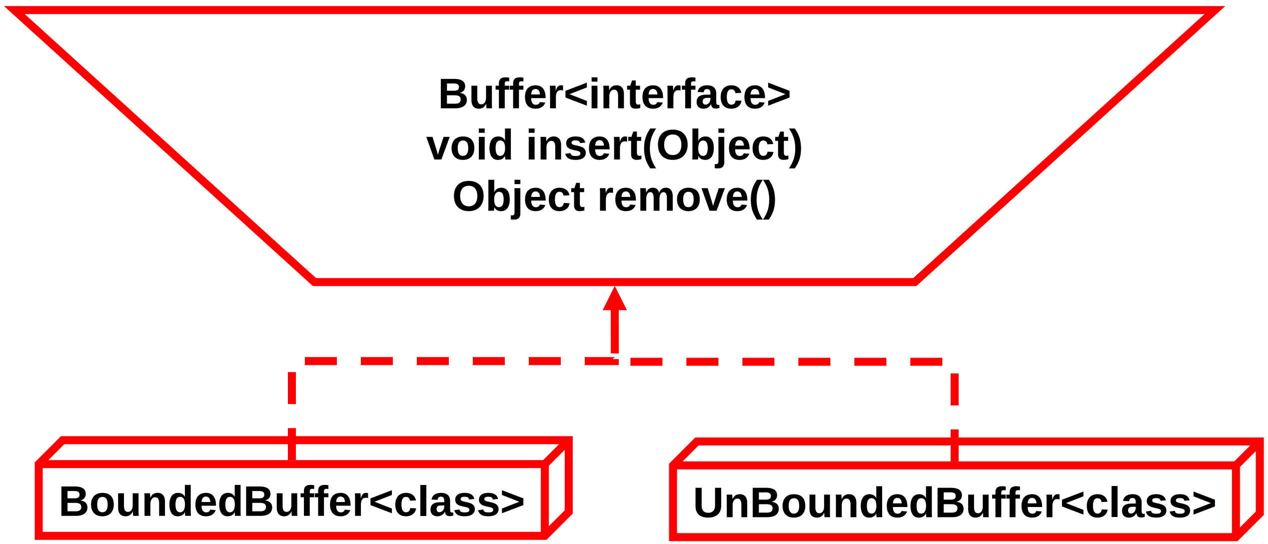


Producer/Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process.
  - *unbounded-buffer* places no practical limit on the size of the buffer.
  - *bounded-buffer* assumes that there is a fixed buffer size.



Modeling The Shared Buffer

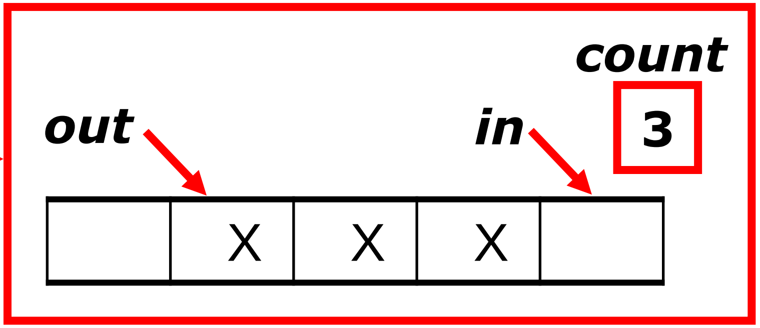



```
public interface Buffer
{
   // producers call this method
   public abstract void insert(Object item);

   // consumers call this method
   public abstract Object remove();
}
```

Bounded Buffer Implementation

```
public class BoundedBuffer implements Buffer {
        private static final int BUFFER_SIZE = 5;
        private int count;
        private int in;
        private int out;
        private Object[] buffer;
        public BoundedBuffer(){   }
        public void insert(Object item){  }
        public Object remove(){  }
}
```




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

```
public void insert(Object item) {
  while (count == BUFFER_SIZE);
  ++count;
  buffer[in] = item;
  in = (in + 1) % BUFFER_SIZE;
}
```

```
public Object remove() {
  Object item;
  while (count == 0);
  --count;
  item = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  return item;
}
```

The Producer

```
import java.util.*;
public class Producer implements Runnable {
 private Buffer buffer;

 public Producer(Buffer b) {
  buffer = b;
 }

 public void run() {
  Date message;
  while (true) {
    int sleeptime = (int) (5000 * Math.random());
   System.out.println("Producer napping "+sleeptime+" milliseconds");
    try {Thread.sleep(sleeptime); }
    catch (InterruptedException e){ }
    message = new Date();
    System.out.println("Producer is a wake and produced " + message);
    buffer.insert(message);
   }
 }

}
```

The Consumer

```
import java.util.*;
public class Consumer implements Runnable {
 private Buffer buffer;

 public Consumer(Buffer b) {
   buffer = b;
 }

 public void run() {
     Date message;
     while (true) {
         int sleeptime = (int) (5000 * Math.random());
         System.out.println("Consumer napping "+sleeptime+" milliseconds");
         try {sleep(sleeptime);}
         catch (InterruptedException e){}
         System.out.println("Consumer is awake and wants to consume.");
         message = (Date) buffer.remove();
     }
 }

}
```

The Factory Class

```
public class Factory{

  public Factory(){
      Buffer buffer = new BoundedBuffer();
      Thread producerThread = new Thread(new Producer(buffer));
      Thread consumerThread = new Thread(new Consumer(buffer));

      producerThread.start();
      consumerThread.start();
  }

  public static void main(String args[]) {
      new Factory();
  }

}
```

The Problem

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having 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.

Threads accessing shared data

```
while (count == BUFFER_SIZE)
   ; // do nothing

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

Body of the insert method invoked by the Producer

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

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

Body of the remove method invoked by the Consumer

A scenario demonstrating the problem

- 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 with “count = 5” initially:

  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}

Synchronization Terminology

- Concurrent accesses to same shared variable, where at least one access is a write:
  - Order of accesses may change result of program
  - May cause irregular errors, very hard to debug

- *Definition of* **Race Condition**: the situation when there is concurrent access to shared data and the final outcome depends upon order of execution.

- *Definition of* **Critical Section:** Section of code where shared data is accessed.

- *Definition of* **Synchronization:** Mechanism that allows the programmer to control the relative order in which operations occur in different threads or processes. (Coordinate Access of shared data)

Cooperative Processes Interaction Structure

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

. . . .

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

Process/thread 1 . . . . ***Process/thread n***

- Entry Section - Code that requests permission to enter its critical section.
- Exit Section - Code that runs after exiting the critical section

The Concept of a Lock

- ***Definition* of a Lock:** LockEntity can be held by only one thread at a time granting a type of permission to do something. LockEntity has an operation to **acquire/hold/lock** and another operation to **release/give-away/unlock**.

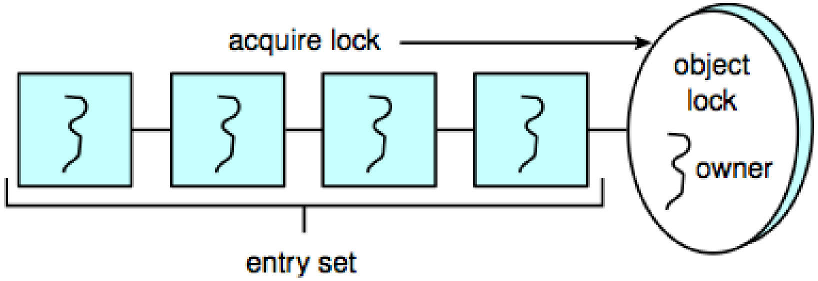
- ***Properties*:**
  - A type of synchronization
  - Used to enforce mutual exclusion
  - Thread can acquire / release locks
  - Thread will wait to acquire lock (stop execution)
    - If lock held by another thread

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.



Each object has an associated **entry set**.

Synchronized Methods In Java

```
Public synchronized void insert(Object item){

            …… // body of the method goes here

}
```

Short hand notation for

```
Public void insert(Object item){
            synchronized(this){

            …… // body of the method goes here

            }
}
```

Locks in Java

- **Properties:**
  - No other thread can get lock in synchronized block
  - Locked block of code Is the critical section

- **Lock is released when block terminates:**
  - End of block reached
  - Exit block due to ***return***, ***continue***, ***break***
  - Exception thrown

Example: Producer/Consumer Problem

Synchronized insert() and remove() methods

```
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;
}
```

Java Synchronization wait/notify()

❖ When a thread invokes *wait()*:

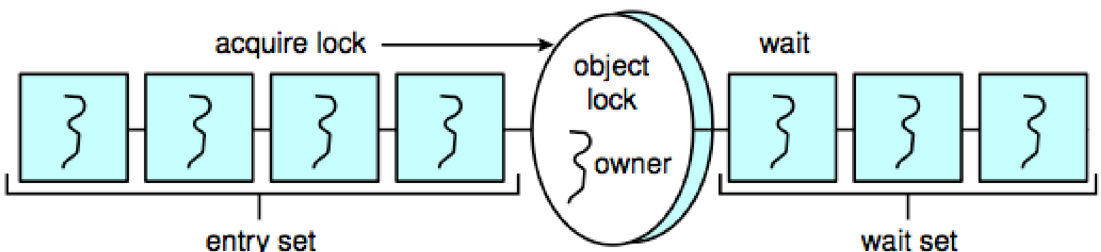
1. The thread releases the object lock;
2. The state of the thread is set to Blocked;
3. The thread is placed in the **wait set** for the object.

❖ 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.

❖ When a thread invokes notifyAll():

- selects all threads in the wait set and moves them to the entry set.



Entry and wait sets

Java Synchronization wait/notify()

- Synchronization rules in Java:
  - A thread that owns the lock for an object can enter another synchronized method (or block) for the same block. This is known recursive or reentrant lock.
  - A thread can nest synchronized method invocations for different objects. Thus, a thread can simultaneously own the lock of several different objects.
  - If a method is not declared synchronized, then it can be invoked regardless of lock ownership, even when another synchronized method for the same object is executing.
  - If the wait set of an object is empty, then the call for notify() or notifyall() has no effect.
  - wait(), notify(), and notifyall() may only be invoked from synchronized methods or blocks; otherwise, an IllegalMonitorStateException is thrown.

Producer/Consumer Problem

```
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;
}
```

Java Synchronization explicit lock

- Using
  - **ReentrantLock** class of **java.util.concurrent.locks** which is an implementation of the interface **java.util.concurrent.locks.Lock** Common methods are:
    - **lock(), unlock(), newCondition()**

  &

  - **Condition** class of **java.util.concurrent.locks** Common methods are:
    - **await(), signal(), signalAll()**

- In this technique a thread can wait using **await()** for a specific condition (which releases the lock) and can get to wake up when it is signaled for that condition using **signal()/signalAll()** if it was waiting for that condition.

Reentrant Locks

- **import**
  - **java.util.concurrent.locks.ReentrantLock;**
  - **Java.util.concurrent.locks.Lock;**
  - **Java.util.concurrent.locks.Condition;**
- Safe usage

```
Lock myLock = new ReentrantLock();
Condition cond1 = myLock.newCondition();
…
Condition cond2 = myLock.newCondition();
…
myLock.lock();
try{
      //critical section
      //cond1.signal();
      //cond2.signal();
      //cond1.signalAll();
      //cond2.signalAll();
}
finally{
      myLock.unlock();
}
```

The Concept of a Semaphore

- A Semaphore is a synchronization tool that does not require busy waiting
- 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

In Dutch *proberen* (test)

In Dutch *verhogen* (increment)

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

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

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

Usage of Semaphores

```
Semaphore S = new Semaphore();

S.acquire();

   // critical section

S.release();

   // remainder section
```

Semaphore Implementation

- Must guarantee that no two processes can execute acquire () and release () on the same semaphore at the same time.

Busy wait: while a process is in its critical section, any other process trying to enter its critical section must loop continuously in the entry code.

Spin Lock: is a semaphore with busy wait, process spins while Waiting for the lock.

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

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

In a single CPU multiprogramming System where locks are expected to be held for long time – critical section is long -, busy wait wastes CPU cycles that some other process might be able to use productively.

In a multiprocessor System where locks are expected to be held for short time – critical section is short -, busy wait is useful requiring no context switching.

But in most applications, locks are expected to be held for long time – critical section is long – So, this implementation is not good.

Semaphore Implementation with no Busy waiting

- 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.

Semaphore Implementation with no Busy waiting (Cont.)

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

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

$$meaning(Value) = \begin{cases} \text{Number of available resources,} & value \geq 0 \\ |value| \text{ is the number of waiting processes,} & value \leq 0 \end{cases}$$

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.

Waits for P1 to release Q.

P_0

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

P_1

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

Waits for P0 to release S.

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

Semaphores

- To create a semaphore call the constructor
  - **Semaphore(int value)**, where value is allowed to be negative.

- To acquire a permit from this semaphore, blocking until one is available, or the thread is interrupted
  - **void acquire()**

- To acquire the given number of permits from this semaphore, blocking until all are available, or the thread is interrupted
  - **void acquire(int permits)**

- To return the current number of permits available in this semaphore
  - **int availablePermits()**

- To release a permit, returning it to the semaphore
  - **void release()**

- To release the given number of permits, returning them to the semaphore
  - **void release(int permits)**

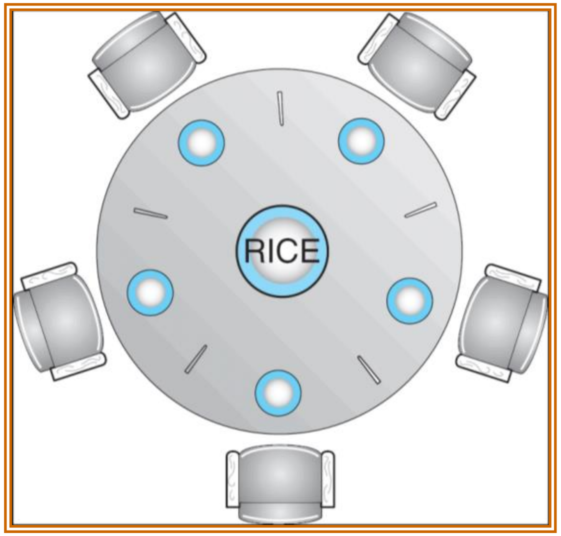
Semaphores

- How to?
  - import java.util.concurrent.Semaphore;
  - Save usage:

```
Semaphore sem = new Semaphore(1);
//……….
try{
    sem.acquire();
    //critical section
}
    finally{
    sem.release();
}
```

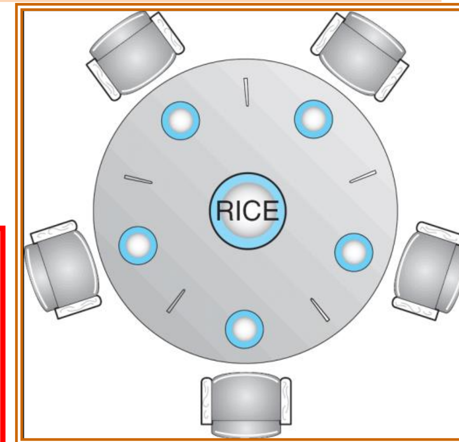
Dining Philosophers Problem

- 5 philosophers with 5 chopsticks placed between them to eat bowl of rice (data set).
- To eat requires two chopsticks.
- Philosophers alternate between thinking and eating.
- OS examples: simultaneous use of multiple resources.



The Dining Philosophers Problem

- Shared data
  - Bowl of rice (data set)
  - Semaphore chopStick [5] initialized to 1




```
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();
}
```

Structure of Philosopher *i*

The Producer/Consumer Problem

- Semaphore mutex
  initialized to the value *1*
- Semaphore full
  initialized to the value *0*.
- Semaphore empty
  initialized to the value *N*.

```
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;

    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) {
       // Figure 6.9
    }

    public Object remove() {
       // Figure 6.10
    }
}
```

The Producer/Consumer Problem

```
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();
}
```

```
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;
}
```

Atomic variables

- An AtomicBoolean is a boolean value that may be updated atomically.
- It is used in applications such as atomically updated flags, and cannot be used as a replacement for a Boolean.
- How to?
  - **import java.util.concurrent.atomic.AtomicBoolean;**
  - To create a new AtomicBoolean with a given initial value:
    - **AtomicBoolean(boolean initialValue)**
  - To atomically set the value to the given updated value if the current value equals the expected value:
    - **boolean compareAndSet(boolean expect, boolean update)**
  - To set to the given value and returns the previous value:
    - **boolean getAndSet(boolean newValue)**
  - To return the current value:
    - **boolean get()**
  - To unconditionally set to the given value:
    - **set(boolean newValue)**

Atomic variables

- Other atomic variables
  - **java.util.concurrent.atomic.AtomicInteger**
  - **java.util.concurrent.atomic.AtomicLong**
  - **java.util.concurrent.atomic.AtomicIntegerArray**
  - **java.util.concurrent.atomic.AtomicLongArray**