THREADS AND THREAD SYNCHRONIZATION

Instructor: Prasun Dewan (FB 150, dewan@unc.edu)
**Distribution vs. Concurrency Program**

- **Process**
  - **Connection**
  - **Distribution, no fine-grained concurrency**

- **Thread**
  - **Process**
    - **Concurrency, not distribution**
  - **Thread**

- **Thread**
  - **Process**
    - **Connection**
    - **Distribution and fine-grained concurrency (typical)**
  - **Thread**
Demo: Halloween Simulation

Single thread implementation?
WAITING SINGLE-THREAD SOLUTION

loop
  wait for local user input
  process user input
  wait for remote user input
  process remote user input
end

Should not impose order on user input

Not a chess game!
Polling Single-Thread Solution

loop
  if (local user input received)
    process user input
  if (remote user input received)
    process remote user input
  sleep (interval)
end

Busy waiting
Wastes computer resources
Reduces response time based on sleep time
Multiple Threads

One reason threads were added to the OS Kernel

Much cleaner code

```
loop
    wait for local user input
    process user input
end
```

```
loop
    wait for remote user input
    process remote user input
end
```
THREADS AND SYNCHRONIZATION

Programming: Abstraction use

Systems: Abstraction design and implementation

Theory: Models and algorithms

Some repetition for those who have seen threads and synchronization before
Topics

- Threads
- Command objects
- Bounded Buffer
- Generics
- Interrupts
- Critical sections
- Busy waiting
- Semaphores
- Monitors
  - Entry procedures
  - Conditions
    - Hints and Absolutes
    - Java and General Hints
- Path Expressions
Program vs. Process vs. Thread

Program

Execution instance

Thread is an independent activity, within a process, associated with a process and a stack.

Process is execution instance of program, associated with program and memory.
THREAD AS AN ACTIVE AGENT WITH DATA STRUCTURES
Thread as a data structure to implementation

To the thread implementation thread is a data structure

Thread implementation can be operating system, programming language, library
Thread as an Independent Activity

- **Ready Threads**: Thread 1, Thread 3
- **Non Ready Threads**: Thread 4, Thread 2, Thread 6
- **Current Thread**: Thread 5
- **Processor**

Whether current, ready, non ready captured by thread status

What triggers context switching?
Rescheduling

With multiple CPUs, multiple current threads, and maybe even multiple ready queues

Context switching occurs as a result of time quantum expiring or some other higher priority thread becoming ready
Thread as a data structure

Where does a thread start executing
What Does a Thread Execute?

- Executes some identifiable portion of a program
- Executes a method call asynchronously
- Thread creator does not wait for method to terminate
PROCEDURE CALL VS. THREAD CREATION

buffer.put("hello");
System.out.println("Put complete");

Thread putThread = async buffer.put("hello");
System.out.println("Put started");

Not real Java code, Java syntax used to illustrate design choices

Why thread object?

What thread operations?
Thread as an Object

A thread is an object representing an independent execution of code, and can be started, suspended, resumed, interrupted while sleeping, given lower/higher priority...
PROCEDURE CALL VS. THREAD CREATION

Thread putThread = async buffer.put (“hello”);
System.out.println(“Put started”);
Thread putThread = (new Thread(buffer, put, “ca va“)).start();
System.out.println(“Put started”);

Object address

Method address

Method parameters

Thread constructor must take varying number of arguments

Varying number of parameters to constructor /method not allowed in type safe languages

Method parameters not allowed in most OO languages
**COMMAND OBJECT**

Command Object = Embedded Operation + Parameters

- **execute ()**
  - Provides an execute operation defined by a well-defined interface to execute some procedure
  - The execute operation takes no arguments

- **constructor (targetObject, param1, param2, ...);**
  - Constructor takes target object and parameters of operation as arguments
  - A command represents a procedure call
Runnable Implementation

package java.lang;
public interface Runnable {
    public void run();
}

Command object defined by the Runnable interface
Example Runnable Implementation

```java
public class AProducer implements Runnable {
    BoundedBuffer<String> boundedBuffer;
    String element;

    public AProducer(
        BoundedBuffer<String> aBoundedBuffer,
        String aString) {
        boundedBuffer = aBoundedBuffer;
        element = aString;
    }

    @Override
    public void run() {
        boundedBuffer.put(element);
    }
}
```

Type parameter assigned as in List

Constructor (targetObject, param1, param2, ...);

run()
public class ABoundedBufferMain {
    public static void main(String[] args) {
        BoundedBuffer<String> greetings =
        new ABoundedBuffer();
        Runnable producer1 =
        new AProducer<String>(greetings, "Hello");
        Runnable producer2 =
        new AProducer<String>(greetings, "Ca Va");
        (new Thread(producer1)).start();
        (new Thread(producer2)).start();
    }
}
Delegation vs. Inheritance Based Thread/Command Object

Thread

HAS-A

AProducer

implements

Runnable

Thread

IS-A

AProducer

implements

Runnable

A command object is not a thread!
public class ABoundedBufferMain { public static void main(String[] args) {
    BoundedBuffer<String> greetings =
        new ABoundedBuffer();
    Runnable producer1 =
        new AProducer<String>(greetings, "Hello");
    Runnable producer2 =
        new AProducer<String>(greetings, "Ca Va");
    (new Thread(producer1)).start();
    (new Thread(producer2)).start();
}
public interface BoundedBuffer<ElementType> {
    void put(ElementType element);
    ElementType get();
}
**Java Generic Types (Type Parameters)**

A scope name (class, interface, method) can be succeeded by a series of type parameters within angle brackets \(<A, B, C, ...>\) that have the same value in all places.

```java
public interface I<T> {
    public void addElement(T t);
    public T elementAt(int index);
    public int size();
}
```

A method, class, interface with type parameter is called a **generic**.

Create an **elaboration** of generic by giving actual value to type parameter

```java
I<String> stringI;
I<Point> pointI;
```

A single implementation is shared by all of its elaborations

Assigning values to type parameters is a compile time activity for type checking only.
**Generic Bounded Buffer Class**

```java
public class ABoundedBuffer<ElementType>
    implements BoundedBuffer<ElementType>{
    // data structures
    ...
    void put(ElementType element) {
        ...
    }
    ElementType get() {
        ...
    }
}
```

- Multiple occurrences of a type variable assigned (unified to) the same value
- Why repeat type parameter in class and interface?
- If a class is elaborated with String then so is the interface

Can we give them different names?
public class ABoundedBuffer implements BoundedBuffer<String>{

    // data structures

    ...

    void put(String element) {
        ...
    }

    ElementType get() {
        ...
    }

}
Parameterized class may implement multiple parameterized interfaces
public interface BoundedBuffer&lt;ElementType&gt; {
    void put(ElementType element);
    ElementType get();
}
public class ABuffer<ElementType> implements BoundedBuffer<ElementType> {
    public static final int MAX_SIZE = 10;
    Object[] buffer = new Object[MAX_SIZE];
    int size = 0;
    int nextIn = 0;
    int nextOut = 0;
}

Convention: Class A<T> implements interface <T> containing public methods of A
Bounded Buffer (Methods)

```java
public void put(ElementType element) {
    if (size >= MAX_SIZE) {
        return;
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
}

public ElementType get() {
    if (size == 0) {
        return null;
    }
    ElementType retVal = (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    return retVal;
}
```

Requirements:
- Requires polling to wait for non-empty and empty slots
- Thread safe? Will it work if multiple threads access it
public class ABoundedBufferMain {
    public static void main(String[] args) {
        BoundedBuffer<String> greetings =
            new ABoundedBuffer();
        Runnable producer1 =
            new AProducer<String>(greetings, "Hello");
        Runnable producer2 =
            new AProducer<String>(greetings, "Ca Va");
        (new Thread(producer1)).start();
        (new Thread(producer2)).start();
    }
}
“Hello” Thread

“Hello” Thread Starts Executing `put()`
"Hello" Thread Proceeds

"Hello" Thread Adds Item

nextIn not incremented before rescheduling happens
“Ca Va” Thread
Context Switch to “Ca Va” Thread

Program behavior depends on when rescheduling occurs

Previous greeting overwritten

Non deterministic program
Critical Section

Code that can be executed only if certain conditions are met by other threads.

Other threads should not be executing the same critical section.

Other threads should not be executing other critical sections that access the same data.
Critical Section (Review)

Code that can be executed only if certain conditions are met by other threads

Other threads should not be executing the same critical section

Other threads should not be executing other critical sections that access the same data
Critical Section

```java
public interface Lock {
    void lock();
    void unlock();
}
```
Critical Section (Review)

```java
public interface Lock {
    void lock();
    void unlock();
}
```

```java
lock.lock();

Critical Section

lock.unlock();
```
public class ALockingBoundedBuffer<ElementType> extends ABoundedBuffer<ElementType>{
    Lock lock;
    public ALockingBoundedBuffer(Lock aLock) {
        lock = aLock;
    }
    public void put(ElementType element) {
        lock.lock();
        super.put(element);
        lock.unlock();
    }
    public ElementType get() {
        lock.lock();
        ElementType retVal = super.get();
        lock.unlock();
        return retVal;
    }
}
**DISABLE CONTEXT SWITCHING**

```
public class AContextSwitchingLock implements Lock {
    int originalInterrupts;
    public void lock() {
        originalInterrupts = System.disableInterrupts();
    }
    public void unlock() {
        System.restoreInterrupts(originalInterrupts);
    }
}
```

- Context switching occurs as a result of a blocking call or an interrupt.
- Timer interrupt can cause time quantum to expire.
- Input/disk/network interrupt can cause higher priority thread to become eligible.
- Disable interrupts and make no blocking call in critical section.
**Pros and Cons**

Coarse grained: stops all threads, even those that do not access the same or any critical section

Insecure: a thread can cheat and hog processor time

Simple

Needed to implement critical sections in operating system

Need critical sections to implement critical sections!
FINE-GRAINED CRITICAL SECTION

lock.lock()

Critical Section

lock.unlock()

Boolean status
public class ABooleanObject implements BooleanObject{
    boolean value;
    public boolean get() {
        return value;
    }
    public void set(boolean newVal) {
        value = newVal;
    }
}

boolean is not an object

Boolean is not an mutable
Busy Waiting Lock

```java
public class APollingLock implements Lock{
    BooleanObject booleanObject = new ABooleanObject();
    public void lock() {
        while (booleanObject.get()) { // do nothing
            
        }
        booleanObject.set(true);
    }
    public void unlock() {
        booleanObject.set(false);
    }
}
```

Test of lock

Test and set not done atomically

Context switch can occur between test and set
Thread 1 Gets Lock

Thread 1 gets lock and context switches
Thread 2 also gets lock
**Busy Waiting Lock**

```java
public class APollingLock implements Lock{
    BooleanObject booleanObject = new ABooleanObject();
    public void lock() {
        while (booleanObject.get()) {
            ; // busy waiting
        }
        booleanObject.set(true);
    }
    public void unlock() {
        booleanObject.set(false);
    }
}
```

- **Test of lock**
- **Set of lock**

Test and set not done atomically

Context switch can occur between test and set

Fix how?
**Test and Set Hardware Instruction**

```java
public class TestAndSetHardware {
    public static atomic boolean testAndSet(BooleanObject aBooleanObject) {
        boolean retVal = aBooleanObject.get(); // test
        aBooleanObject.set(true); // set
        return retVal; // return tested value
    }
}
```

Function with side effects!

At the end of instruction, boolean object (memory slot) is always true
returns false if object was false at start of instruction
public class ATestAndSetLock implements Lock{
    BooleanObject booleanObject;
    public void lock() {
        while (TestAndSetHardware.testAndSet(booleanObject)) {
            ;
        }
    }
    public void unlock() {
        booleanObject.set(false);
    }
}
Busy Waiting Pros and Cons

- Busy waiting: polling wastes computer resources
- Higher priority polling thread would not allow lower priority thread in critical section to release lock
- Needed to implement critical sections in multiprocessor operating system
- Disabling interrupts on one processor does not disable on others
- Fine-grained approach with out disadvantages?
public class ASemaphore implements Semaphore{
    Queue<Thread> queue = new AQueue<Thread>();
    int count;
    public ASemaphore(int initialCount) {
        count = initialCount;
    }
    public void semWait() {
        int originalInterrupts = System.disableInterrupts();
        count--;
        if (count < 0) {
            Thread currentThread = Thread.currentThread();
            currentThread.setStatus(Thread.WAIT);
            queue.add(currentThread);
        }
        System.restoreInterrupts(originalInterrupts);
    }
}
public void semSignal() {
    int originalInterrupts = System.disableInterrupts();
    count++;
    if (count <= 0) {
        Thread nextThread = queue.remove();
        nextThread.setStatus(Thread.READY);
        Thread.reschedAndRestoreInterrupts(originalInterrupts);
    } else {
        System.restoreInterrupts(originalInterrupts);
    }
}

If count is positive, indicates how many semWait() can be done without blocking

If count is negative, its absolute value indicates queue size
**Semaphore-Based Lock**

```java
public class ASemaphoreLock implements Lock {
    // let first thread go
    Semaphore semaphore = new ASemaphore(1);
    public void lock() {
        semaphore.semWait();
    }
    public void unlock() {
        semaphore.semSignal();
    }
}
```
Like other schemes discussed so far, requires explicit lock and unlock

Programmer overhead in creating and using lock

Programmer may forget to lock

Worse, may forget to unlock

Program reader must look at code to find critical sections
public class ALockingBoundedBuffer<ElementType> extends ABoundedBuffer<ElementType> {

    Lock lock;

    public ALockingBoundedBuffer(Lock aLock) {
        lock = aLock;
    }

    public void put(ElementType element) {
        lock.lock();
        super.put(element);
        lock.unlock();
    }

    public ElementType get() {
        lock.lock();
        ElementType retVal = super.get();
        lock.unlock();
        return retVal;
    }
}

Create declarative rather than procedural solution
Declarative vs. Procedural Primitive

```
public class ALockingBoundedBuffer<ElementType> extends ABoundedBuffer<ElementType>{
    Lock lock;
    public ALockingBoundedBuffer(Lock aLock) {
        lock = aLock;
    }
    public void put(ElementType element) {
        lock.lock();
        super.put(element);
        lock.unlock();
    }
    public ElementType get() {
        lock.lock();
        ElementType retVal = super.get();
        lock.unlock();
        return retVal;
    }
}
```

Declarative: Directive to system by giving declaration specifying what

Procedural: Directive to system by executing statements specifying how

A la Regular expressions vs. Finite State Automata
**HIGH-LEVEL SUPPORT**

- Critical Section

Other threads should not be executing other critical sections that access the same data.

Connect lock to shared data and methods that manipulate them.
When declaring a class, provide a label (e.g. synchronized) that says that on an instance a method cannot be executed if the same or another method is executing on that instance.

Such a class is a monitor class, and its instance a monitor.

Not all classes have critical sections, hence labeling can prevent unnecessary code and data structures in such non monitor classes.
Labeling a Class as a Monitor

```java
public synchronized ABoundedBuffer<ElementType>
    implements BoundedBuffer<ElementType>{
    public static final int MAX_SIZE = 10;
    Object[] buffer = new Object[MAX_SIZE];
    int size = 0;
    int nextIn = 0;
    int nextOut = 0;

    Some methods of the class may not be critical sections
```
When declaring a method, provide a label (e.g. synchronized). A single synchronized method can execute at one time on an instance.
**Labeling Methods as Atomic**

A lock associated with class instance is automatically obtained on entry to the marked procedure and released on its exit.

Such a method is classically called an **entry procedure**.

Java calls it a **synchronized method**, we will call it an **atomic method**.
public synchronized void put(
    ElementType element) {
    if (size >= MAX_SIZE) {
        return;
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.condSignal();
}

public synchronized ElementType get() {
    if (size == 0) {
        return null;
    }
    ElementType retVal =
        (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.condSignal();
    return retVal;
}
public synchronized void put(
    ElementType element) {
    if (size >= MAX_SIZE) {
        return;
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.condSignal();
}

public synchronized ElementType get() {
    if (size == 0) {
        return null;
    }

    ElementType retVal =
        (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.condSignal();
    return retVal;
}
public synchronized void put(ElementType element) {
    if (size >= MAX_SIZE) {
        return;
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.condSignal();
}

public synchronized ElementType get() {
    if (size == 0) {
        return null;
    }
    ElementType retVal = (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.condSignal();
    return retVal;
}
public synchronized void put(ElementType element) {
    if (size >= MAX_SIZE) {
        return;
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.condSignal();
}

public synchronized ElementType get() {
    if (size == 0) {
        return null;
    }
    ElementType retVal =
        (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.condSignal();
    return retVal;
}
public synchronized void put(ElementType element) {
    if (size >= MAX_SIZE) {
        return;
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.condSignal();
}

public synchronized ElementType get() {
    if (size == 0) {
        return null;
    }
    ElementType retVal =
        (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.condSignal();
    return retVal;
}
```java
public synchronized void put(ElementType element) {
    if (size >= MAX_SIZE) {
        return;
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.condSignal();
}

public synchronized ElementType get() {
    if (size == 0) {
        return null;
    }
    ElementType retVal = (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.condSignal();
    return retVal;
}
```
public synchronized void put(ElementType element) {
    if (size >= MAX_SIZE) {
        return;
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.condSignal();
}

public synchronized ElementType get() {
    if (size == 0) {
        return null;
    }
    ElementType retval =
        (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.condSignal();
    return retval;
}
public synchronized void put(ElementType element) {
    if (size >= MAX_SIZE) {
        return;
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.condSignal();
}

public synchronized ElementType get() {
    if (size == 0) {
        return null;
    }
    ElementType retVal =
        (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.condSignal();
    return retVal;
}
public synchronized void put(
    ElementType element) {
    if (size >= MAX_SIZE) {
        return;
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;  
    nonEmpty.condSignal();
}

public synchronized ElementType get() {
    if (size == 0) {
        return null;
    }
    ElementType retVal =
        (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;  
    nonFull.condSignal()
    return retVal;
}
<table>
<thead>
<tr>
<th>Monitor Pros and Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Higher-level declarative solution</td>
</tr>
<tr>
<td>No programmer overhead in creating and using lock</td>
</tr>
<tr>
<td>No danger of forgetting to unlock</td>
</tr>
<tr>
<td>Can simply look at method headers to determine critical sections</td>
</tr>
<tr>
<td>Can forget to label method</td>
</tr>
<tr>
<td>Does not provide (general) thread synchronization</td>
</tr>
</tbody>
</table>
**Synchronization vs. Mutual Exclusion**

- Thread general synchronization (Thread coordination): a thread waiting for another thread to do something

- Thread mutual exclusion: a thread waiting for another thread to leave a critical section

- Threads are “competing” rather than “cooperating”

- Thread synchronization: general synchronization other than mutual exclusion

- Threads are “cooperating” rather than “competing”
public synchronized void put(ElementType element) {
    if (size >= MAX_SIZE) {
        return;
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
}

public synchronized ElementType get() {
    if (size == 0) {
        return null;
    }
    ElementType retVal = (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    return retVal;
}

Should wait for non full buffer
Should wait for non empty buffer
Semaphores? How many?
Monitors do not support general thread synchronization
public synchronized void put(ElementType element) {
    nonFull.semWait();
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.semSignal();
}

public synchronized ElementType get() {
    nonEmpty.semWait();
    ElementType retVal = (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.semSignal();
    return retVal;
}
**Bounded Buffer (Data Structures)**

```java
public class ABoundedBuffer<ElementType>
    implements BoundedBuffer<ElementType>
{
    public static final int MAX_SIZE = 10;
    Object[] buffer = new Object[MAX_SIZE];
    int size = 0;
    int nextIn = 0;
    int nextOut = 0;
    Semaphore nonFull = new ASemaphore(MAX_SIZE);
    Semaphore nonEmpty = new ASemaphore(0);
}
```

General synchronization solution: Monitors + Semaphores?

should be able to deposit MAX_SIZE buffers without waiting
**Seminaphores too Heavyweight**

```java
public synchronized void put(ElementType element) {
    nonFull.semWait();
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.semSignal();
}

public synchronized ElementType get() {
    nonEmpty.semWait();
    ElementType retVal = (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.semSignal();
    return retVal;
}
```
**Condition (Synchronizer)**

```java
public class AConditionSynchronizer implements ConditionSynchronizer {
    Queue<Thread> queue = new AQueue<Thread>();

    public void condWait() {
        Thread currentThread = Thread.currentThread();
        currentThread.setStatus(Thread.WAIT);
        queue.add(currentThread);
    }

    public void condSignal() {
        Thread nextThread = queue.remove();
        nextThread.setStatus(Thread.READY);
        Thread.reschedAndRestoreInterrupts(originalInterrupts);
    }
}
```

- Non atomic operations – no disabling of context switching
- Int count is only one kind of condition for synchronization
- No integer count
- Synchronization condition checked by caller
public synchronized void put(ElementType element) {
    nonFull.semWait();
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.semSignal();
}

public synchronized ElementType get() {
    nonEmpty.semWait();
    ElementType retVal = (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.semSignal();
    return retVal;
}
P O L I C Y  A N D  M E C H A N I S M  O R T H O G O N A L

```java
public synchronized void put(ElementType element) {
    if (size >= MAX_SIZE) {
        nonFull.condWait();
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.condSignal();
}

public synchronized ElementType get() {
    if (size == 0) {
        return nonEmpty.condWait();
    }
    ElementType retVal = (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.condSignal();
    return retVal;
}
```

Must check programmer-defined boolean expression before waiting

Policy for synchronizing and mechanism for blocking are orthogonal

Could we put check outside semaphore also
**Lightweight Semaphore**

```java
public class ASemaphore implements Semaphore {
    Queue<Thread> queue = new AQueue<Thread>();
    public void semWait() {
        int originalInterrupts = System.disableInterrupts();
        Thread currentThread = Thread.currentThread();
        currentThread.setStatus(Thread.WAIT);
        queue.add(currentThread);
        System.restoreInterrupts(originalInterrupts);
    }
    public void semSignal() {
        int originalInterrupts = System.disableInterrupts();
        Thread nextThread = queue.remove();
        nextThread.setStatus(Thread.READY);
        Thread.reschedAndRestoreInterrupts(originalInterrupts);
    }
}
```

Like regular semaphore, except no count
Like condition, except there is atomicity implementation
public void put(ElementType element) {
    if (size >= MAX_SIZE) {
        nonFull.semWait();
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.semSignal();
}

public ElementType get() {
    if (size == 0) {
        return nonEmpty.semWait();
    }
    ElementType retVal = (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.semSignal();
    return retVal;
}
SYNCHRONIZER OPERATIONS IN ATOMIC METHODS

```java
public synchronized void put(ElementType element) {
    if (size >= MAX_SIZE) {
        nonFullCondWait();
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmptyCondSignal();
}

public synchronized ElementType get() {
    if (size == 0) {
        return nonEmptyCondWait();
    }
    ElementType retVal = (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFullCondSignal();
    return retVal;
}
```

Blocking/unblocking operations should be done atomically

Conditions for blocking/unblocking should also be checked atomically with blocking/unblocking operations

Force blocking/unblocking operations to be done only in atomic methods, which can then check and set arbitrary conditions atomically
public class ABoundedBuffer<ElementType> implements BoundedBuffer<ElementType> {
    public static final int MAX_SIZE = 10;
    Object[] buffer = new Object[MAX_SIZE];
    int size = 0;
    int nextIn = 0;
    int nextOut = 0;

    ConditionSynchronizer nonFull =
        new AConditionSynchronizer();
    ConditionSynchronizer nonEmpty =
        new AConditionSynchronizer();
**Queues with Conditions**

```java
public synchronized void put(
    ElementType element) {
    if (size >= MAX_SIZE) {
        nonFull.condWait();
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.condSignal();
}

public synchronized ElementType get() {
    if (size == 0) {
        nonEmpty.condWait();
    }
    ElementType retVal =
        (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.condSignal();
    return retVal;
}
```
public synchronized void put(
    ElementType element) {
    if (size >= MAX_SIZE) {
        nonFull.condWait();
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.condSignal();
}

cpyublic synchronized ElementType get() {
    if (size == 0) {
        nonEmpty.condWait();
    }
    ElementType retVal =
        (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.condSignal();
    return retVal;
}
public synchronized void put(ElementType element) {
    if (size >= MAX_SIZE) {
        nonFull.condWait();
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.condSignal();
}
public synchronized ElementType get() {
    if (size == 0) {
        nonEmpty.condWait();
    }
    ElementType retVal =
        (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.condSignal()
    return retVal;
}
public synchronized void put(ElementType element) {
    if (size >= MAX_SIZE) {
        nonFull.condWait();
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.condSignal();
}

public synchronized ElementType get() {
    if (size == 0) {
        nonEmpty.condWait();
    }
    ElementType retVal =
        (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.condSignal();
    return retVal;
}
public synchronized void put(ElementType element) {
    if (size >= MAX_SIZE) {
        nonFull.condWait();
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.condSignal();
}

public synchronized ElementType get() {
    if (size == 0) {
        nonEmpty.condWait();
    }
    ElementType retVal =
        (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.condSignal()
    return retVal;
}
public synchronized void put(ElementType element) {
    if (size >= MAX_SIZE) {
        nonFull.condWait();
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.condSignal();
}

public synchronized ElementType get() {
    if (size == 0) {
        nonEmpty.condWait();
    }
    ElementType retVal =
    (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.condSignal();
    return retVal;
}
CONSUMER UNLOCKS MONITOR

```java
public synchronized void put(
    ElementType element) {
    if (size >= MAX_SIZE) {
        nonFull.condWait();
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.condSignal();
}

public synchronized ElementType get() {
    if (size == 0) {
        nonEmpty.condWait();
    }
    return buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.condSignal();
}
```

Before waiting, class invariant should be restored

Class invariant is a condition that is true before and after each public method in the class is called

E.g.: size should indicate the number of filled slots
```java
public synchronized void put(
    ElementType element) {
    if (size >= MAX_SIZE) {
        nonFull.condWait();
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.condSignal();
}

public synchronized ElementType get() {
    if (size == 0) {
        nonEmpty.condWait();
    }
    ElementType retVal =
        (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.condSignal();
    return retVal;
```
**Producer Signals, Consumer Dequeued**

```java
public synchronized void put(ElementType element) {
    if (size >= MAX_SIZE) {
        nonFull.condWait();
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.condSignal();
}

public synchronized ElementType get() {
    if (size == 0) {
        nonEmpty.condWait();
    }
    ElementType retVal = (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.condSignal();
    return retVal;
}
```
**Signaled Thread in Monitor?**

```java
public synchronized void put(ElementType element) {
    if (size >= MAX_SIZE) {
        nonFull.condWait();
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.condSignal();
}

public synchronized ElementType get() {
    if (size == 0) {
        nonEmpty.condWait();
    }
    ElementType retval =
    (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.condSignal();
    return retval;
}
```

- **Entry Queue**
- **Cons. 1**
- **Cons. 2**
- **nonEmpty Condition Queue**
- **nonFull Condition Queue**

**Prod. 1**

 Violates Monitor Definition
**Signaled Thread in Entry Queue**

**Entry Queue**

- **Cons. 1**
- **Cons. 2**

**nonEmpty Condition Queue**

**nonFull Condition Queue**

```java
public synchronized void put(ElementType element) {
    if (size >= MAX_SIZE) {
        nonFull.condWait();
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.condSignal();
}

public synchronized ElementType get() {
    if (size == 0) {
        nonEmpty.condWait();
    }
    ElementType retVal =
        (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.condSignal();
    return retVal;
}
```

**Cons. 1 will get the buffer first**

**Starvation possible**
**Signaled Thread in Urgent Queue**

```
public synchronized void put(ElementType element) {
    if (size >= MAX_SIZE) {
        nonFull.condWait();
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.condSignal();
}

public synchronized ElementType get() {
    if (size == 0) {
        nonEmpty.condWait();
    }
    ElementType retVal = (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.condSignal();
    return retVal;
}
```

- Signaled thread in urgent Q
- Urgent Q serviced before Entry Q
- Condition for Entering May Not Exist Later
- Signaler may violate condition or previously signaled thread may violate it
Signaled Thread in Monitor and Signaling Thread Urgent Queue

```java
public synchronized void put(ElementType element) {
    if (size >= MAX_SIZE) {
        nonFull.condWait();
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.condSignal();
}

public synchronized ElementType get() {
    if (size == 0) {
        nonEmpty.condWait();
    }
    ElementType retVal = (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.condSignal();
    return retVal;
}
```

Signaling thread must restore monitor invariant before signaling.
public synchronized void put(ElementType element) {
    if (size >= MAX_SIZE) {
        nonFull.condWait();
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.condSignal();
}

public synchronized ElementType get() {
    if (size == 0) {
        nonEmpty.condWait();
    }
    ElementType retVal =
        (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.condSignal();
    return retVal;
}
public synchronized void put(ElementType element) {
    if (size >= MAX_SIZE) {
        nonFull.condWait();
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.condSignal();
}

public synchronized ElementType get() {
    if (size == 0) {
        nonEmpty.condWait();
    }
    ElementType retVal = (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.condSignal();
    return retVal;
}
**Make Signal a Return, Let Signaled Thread Enter**

```java
public synchronized void put(ElementType element) {
    if (size >= MAX_SIZE) {
        nonFull.condWait();
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.condSignal();
}

public synchronized ElementType get() {
    if (size == 0) {
        nonEmpty.condWait();
    }
    ElementType retVal = (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.condSignal();
    return retVal;
}
```

- **Entry Queue**
- **Prod. 1**
- **Cons. 2**
- **nonEmpty Condition Queue**
- **nonFull Condition Queue**

A single signal allowed

No statement allowed after signal

Make signal a hint
**Hint vs. Absolute**

**Absolute:** Application is provided information with absolute guarantee

A la non probabilistic algorithm

The server is on machine M a la the person is at address A

The condition for which you were waiting is true

**Hint:** Application is provided information that is likely to be true but can be checked

A la probabilistic algorithm

The server is probably on machine M, and if the message bounces, check with some central authority at some cost

The condition for which you are waiting is probably true, but must check
**HINT VS. ABSOLUTE**

- Programmer must have code to check truth
- If check fails, operations are more expensive
- Usual case is efficient, unusual case is possible but not efficient
- Servers do not move usually, condition does not usually get violated
**Signal is a Hint**

Entry Queue

Cons. 2

nonEmpty Condition Queue

nonFull Condition Queue

Urgent Queue

Cons. 1

Make usual case efficient and other cases possible

```
public synchronized void put(ElementType element) {
    if (size >= MAX_SIZE) {
        nonFull.condWait();
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.condSignal();
}
```

```
public synchronized ElementType get() {
    if (size == 0) {
        nonEmpty.condWait();
    }
    ElementType retVal = (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.condSignal();
    return retVal;
}
```

Signaled thread in urgent Q, not guaranteed state at signal time

Signaled thread must recheck condition

while (size >= MAX_SIZE) {
    nonFull.condWait();
}

while (size == 0) {
    nonEmpty.condWait();
}
if (!okToProceed) {
    condSynchronizer.condWait();
}

while (!okToProceed) {
    condSynchronizer.condWait();
}

Polling?

- Not polling as each iteration has a wait
- Allows broadcast: all waiting threads are unblocked
public synchronized void put(ElementType element) {
    while (size >= MAX_SIZE) {
        nonFull.condWait();
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.condBroadcastSignal();
}

public synchronized ElementType get() {
    while (size == 0) {
        nonEmpty.condWait();
    }
    ElementType retVal =
        (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.condSignal();
    return retVal;
}
public synchronized void put(ElementType element) {
    while (size >= MAX_SIZE) {
        nonFull.condWait();
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.condBroadcastSignal();
}

public synchronized ElementType get() {
    while (size == 0) {
        nonEmpty.condWait();
    }
    ElementType retVal =
    (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.condSignal();
    return retVal;
}
```java
public synchronized void put(ElementType element) {
    while (size >= MAX_SIZE) {
        nonFull.condWait();
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.condBroadcastSignal();
}

public synchronized ElementType get() {
    while (size == 0) {
        nonEmpty.condWait();
    }
    ElementType retVal =
        (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.condSignal()
    return retVal;
}
```
```java
public synchronized void put(ElementType element) {
    while (size >= MAX_SIZE) {
        nonFull.condWait();
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.condBroadcastSignal();
}

public synchronized ElementType get() {
    while (size == 0) {
        nonEmpty.condWait();
    }
    ElementType retVal =
        (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.condSignal();
    return retVal;
}
```
**SECOND SIGNALED THREAD ENTERS**

```
public synchronized void put(ElementType element) {
    while (size >= MAX_SIZE) {
        nonFull.condWait();
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.condBroadcastSignal();
}

public synchronized ElementType get() {
    while (size == 0) {
        nonEmpty.condWait();
    }
    ElementType retVal =
        (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.condSignal()
    return retVal;
}
```
public synchronized void put(ElementType element) {
    while (size >= MAX_SIZE) {
        nonFull.condWait();
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    nonEmpty.condBroadcastSignal();
}
public synchronized ElementType get() {
    while (size == 0) {
        nonEmpty.condWait();
    }
    ElementType retVal =
        (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    nonFull.condSignal()
    return retVal;
}
Extra check for each signaled thread

Signaled thread may in unusual cases wait again

Signaling thread does not have to restore invariant

Usual case avoids context switch (faster than extra check) or restricted signal (on return)

Allows broadcast semantics
```java
public synchronized void put(
        ElementType element) {
    while (size >= MAX_SIZE) {
        try {
            this.wait();
        } catch (Exception e) {}  // This line is highlighted
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    this.notify();
}

public synchronized ElementType get() {
    while (size == 0) {
        try {
            this.wait();
        } catch (Exception e) {}  // This line is highlighted
    }
    ElementType retVal =
        (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    this.notify();
    return retVal;
}
```

Both producer and consumer waiting on same condition synchronizer but different conditions
public synchronized void put(ElementType element) {
    while (size >= MAX_SIZE) {
        try {
            this.wait();
        } catch (Exception e) {} 
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    this.notify();
}

public synchronized ElementType get() {
    while (size == 0) {
        try {
            this.wait();
        } catch (Exception e) {} 
    }
    ElementType retVal = (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    this.notify();
    return retVal;
}

notifyAll for Bounded Buffer?

Buffer cannot be full and empty at the same time

Bounded Buffer is driving problem

Multiple conditions share a synchronizer, hence modified name

Entry Queue

Single Condition Queue

Prod. 1 Cons. 2

Urgent Queue

JAVA HINT SEMANTICS FOR BOUNDED BUFFER
**JAVA vs. GENERAL HINT SEMANTICS**

- Every object has an entry and condition queue
- Probably why primitive values are not objects as in Smalltalk
- May need broadcast semantics even when one thread eligible
- Difficult to imagine cases when that happens
- No need for special class labeling as monitor
- No need to declare condition synchronizer
Always NonEmpty, NonFull Buffer

```java
public void put(
    ElementType element) {
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
}

public ElementType get() {
    ElementType retVal =
        (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    return retVal;
}
```

- Puts cannot be executed concurrently
- Gets cannot be executed concurrently
- Puts and gets can be executed concurrently
ATOMIC METHODS?

```java
public synchronized void put(
    ElementType element) {
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
}

public synchronized ElementType get() {
    ElementType retVal =
        (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    return retVal;
}
```

Puts and gets can be executed concurrently.
Condition synchronizer operations and associated checks should occur in atomic methods.
**Synchronized Statement Block**

```java
public void put(
    ElementType element) {
    synchronized (putLock) {
        buffer[nextIn] = element;
        nextIn = (nextIn + 1) % MAX_SIZE;
    }
}

public ElementType get() {
    ElementType retVal =
        (ElementType) buffer[nextOut];
    synchronized (getLock) {
        getLocked = true;
        nextOut = (nextOut + 1) % MAX_SIZE;
    }
    return retVal;
}
```

- **putLock** and **getLock** are monitors with no atomic methods associated with statement block.

- `synchronize(object) { <statement list> }` gets monitor lock at start of statement list and releases it on end, without entering monitor.
public class ABoundedBuffer<ElementType>
  implements BoundedBuffer<ElementType>{
    public static final int MAX_SIZE = 10;
    Object[] buffer = new Object[MAX_SIZE];
    int nextIn = 0;
    int nextOut = 0;
    Object putLock = new Object();
    Object getLock = new Object();

    putLock and getLock are monitors with no atomic methods: Object instances
Once a thread has the monitor lock, it can execute any number of atomic (and non atomic) methods before returning from initial method.
**Synchronized Calling Another Synchronized Method in Same Monitor**

When synchronized method in one monitor calls synchronized method in another monitor, it does not release lock of first monitor.

Does not have to re-get lock and restore monitor invariant before calling external method, can execute both atomically.

Can lead to deadlocks.
Some designs release lock when calling external method
Waiting in Synchronized Method

Monitor 1

method1()

this.wait();

Lock released when wait called on this in a synchronized method
WAITING IN EXTERNAL MONITOR IN SYNCHRONIZED METHOD

In Java: No-op (not illegal) when wait called on another monitor in a synchronized method

Monitor 1

method1()

putLock.wait();
WAITING IN EXTERNAL MONITOR IN SYNCHRONIZED METHOD

```java
synchronized(putLock) {
    try {
        putLock.wait();
    } catch (Exception e) {}
    ...
}
```

In Java: Synchronized statement block can wait on monitor associated with block

Can wait on or notify condition queue of monitor iff you have that monitor lock
In Java: all waiting calls keep the lock

Even though waiting method can do nothing and Java knows it is waiting

Better semantics: require invariant to be restore when lock is gotten
**Higher-Level Solution?**

Entry Queue

Single Condition Queue

Cons. 2

Urgent Queue

---

```java
public synchronized void put(
        ElementType element) {
    while (size >= MAX_SIZE)
        try{
            this.wait();
        } catch(Exception e) {}
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    this.notify();
}
```

Declarative for mutual exclusion

Procedural for (other) synchronization

Declarative Synchronization?

Hint: FSA vs. Regular Expressions?

Vocabulary is methods
public synchronized void put(
    ElementType element) {
    while (size >= MAX_SIZE) {
        this.wait();
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
    this.notify();
}

public synchronized ElementType get() {
    while (size == 0) {
        this.wait();
    }
    ElementType retVal =
        (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    size--;
    this.notify();
    return retVal;
}
public class ABoundedBuffer<ElementType>
    implements BoundedBuffer<ElementType>{{
    public static final int MAX_SIZE = 10;
    Object[] buffer = new Object[MAX_SIZE];
    int size = 0;
    int nextIn = 0;
    int nextOut = 0;
}

No size field
**Regular vs. Path Expressions**

**Regular**
- Vocabulary: $\{a_1, \ldots, a_N\}$
- Usually characters
- Defines legal sequences of vocabulary elements
- $a \mid d$
- Either $a$ or $d$ is acceptable

**Path**
- Vocabulary: $\{m_1, \ldots, m_N\}$
- Method calls
- Defines legal sequences of method calls in a module
- Get, put
- Either get or put can be called
### Regular vs. Path Expressions

<table>
<thead>
<tr>
<th>Regular</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R = a_i$</td>
<td>$P = m_i$</td>
</tr>
<tr>
<td>$R = R_1 \ R_2$</td>
<td>$P = P_1; P_2$</td>
</tr>
<tr>
<td>$R$ must be followed by $R_2$</td>
<td>$P_1$ must be called after $P_2$</td>
</tr>
<tr>
<td>$R = R_1 \mid R_2$</td>
<td>$P = P_1, P_2$</td>
</tr>
<tr>
<td>$R_1$ or $R_2$ is legal</td>
<td>Call to $P_1$ or $P_2$ can be made</td>
</tr>
<tr>
<td>$R^*$</td>
<td>$P = [P]$</td>
</tr>
<tr>
<td>Zero or more occurrences of $R$ can be matched next</td>
<td>Zero or more occurrences of call to $P$ can be executed</td>
</tr>
<tr>
<td></td>
<td>$N: (P)$</td>
</tr>
<tr>
<td></td>
<td>Up to $N$ concurrent executions of $P$ can be executed</td>
</tr>
</tbody>
</table>
Regular vs. Path Expressions (Review)

**Regular**
- Vocabulary = \{a_1, .. a_N\}
- Usually characters
- Defines legal sequences of vocabulary elements
- a | d
- either a or d is acceptable

**Path**
- Vocabulary = \{m_1, .. m_N\}
- Method calls
- Defines legal sequences of method calls in a module
- get, put
- either get or put can be called
### Regular vs. Path Expressions (Review)

<table>
<thead>
<tr>
<th>Regular</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R = a_i )</td>
<td>( P = m_i )</td>
</tr>
<tr>
<td>( R = R_1 \ R_2)</td>
<td>( P = P_1;P_2 )</td>
</tr>
<tr>
<td>( R_1 ) must be followed by ( R_2 )</td>
<td>( P_1 ) must be called after ( P_2 )</td>
</tr>
<tr>
<td>( R = R_1 \mid R_2 )</td>
<td>( P = P_1,P_2 )</td>
</tr>
<tr>
<td>( R_1 ) or ( R_2 ) is legal</td>
<td>Call to ( P_1 ) or ( P_2 ) can be made</td>
</tr>
<tr>
<td>( R^* )</td>
<td>( P = [P] )</td>
</tr>
<tr>
<td>Zero or more occurrences of ( R ) can be matched next</td>
<td>Zero or more occurrences of call to ( P ) can be executed</td>
</tr>
<tr>
<td></td>
<td>( N: (P) )</td>
</tr>
<tr>
<td></td>
<td>Up to ( N ) concurrent executions of ( P ) can be executed</td>
</tr>
</tbody>
</table>
**SPECIFYING MUTUAL EXCLUSION**

```java
public void put(
    ElementType element) {
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
}

public ElementType get() {
    ElementType retVal =
        (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    return retVal;
}
```

- **put and get can be executed concurrently, as no shared size variable**

- **An arbitrary number of activations of get or put and get can be executed concurrently, [] implicit on outermost expression**

- **Only one execution of get or put can be active at any one time**
public void put(
    ElementType element) {
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
}
public ElementType get() {
    ElementType retVal =
        (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    return retVal;
}
### Specifying Synchronization

```java
public void put(
    ElementType element) {
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
}

public ElementType get() {
    ElementType retVal =
        (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;
    return retVal;
}
```

- **Put; Get**
  - Each activation of get preceded by put, an infinite number of concurrent put; get sequences
  - A single put; get sequence can be active at any one time
  - The two methods alternate, BB of size 1
  - Upto N activations of (put; get) can be active at any one time

- **Puts can execute concurrently with another put**
- **No mutual exclusion!**
SPECIFYING MUTUAL EXCLUSION AND SYNCHRONIZATION

```java
class Buffer {
    public void put(ElementType element) {
        buffer[nextIn] = element;
        nextIn = (nextIn + 1) % MAX_SIZE;
    }

    public ElementType get() {
        ElementType retVal = (ElementType) buffer[nextOut];
        nextOut = (nextOut + 1) % MAX_SIZE;
        return retVal;
    }
}
```

Mutual Exclusion

1: (get), 1: (put)

Only one execution of get or put can active

Synchronization

N: (put; get)

N activations of (put; get) can be active at any one time

Both

N: (1:(put); 1:(get))

N activations of (put; get) can be active but only one activation of put and one activation of get

N: (path expression) gives global restriction, not within subexpression
There can be an arbitrary number of readers or a single writer active at any one time

1: ([read], write)

Fair readers writers: writer does not wait for a reader who comes after it

Path expression cannot handle all synchronization schemes

Regular expression cannot handle all languages
**Path Expressions**

- Method calls are matched according to specified path expressions.
- Method calls that do not match are blocked.
- After top-level path expression is matched, it can be matched again.
- Implementation unspecified: when is unblocking checking checked and which waiting method unblocked.
- Not practical but helps understand synchronization better.

Example: (get, put)