CS49K Programming Languages

Chapter 13: Concurrency

LECTURE 16: CONCURRENT PROGRAMMING MECHANISM

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- Monitor
- Semaphore
- Condition Variable
- Conditional Critical Region
- Transaction Memory
- Other Topics

Language-level Mechanisms I

Monitor Definition

SECTION 1

Semaphores’ Weakness

Complex patterns of resource usage (Semaphore has limited seats, not a expandable queue.)

- Cannot capture relationship with only semaphores
- Need extra State variables to record information (not a class)
- Use semaphores such that
  - One is for mutual exclusion around state variables
  - One for each class waiting (hard to debug)

Monitors

Semaphore is only a synchronization scheme, while monitor is a shared data class with operations and synchronization scheme

- A Programming language construct that supports controlled access to shared data
  - Synchronization code added by compiler, enforced at runtime.
  - Why does this help?
- Monitor is a software module that encapsulates: (A shared-data class with synchronization)
  - **Shared data** structures
  - **Procedures** that operate on shared data
  - **Synchronization** between concurrent processes that invoke those procedures
- Monitor protects the data from unstructured access
  - Guarantees only access data through procedures, hence in legitimate ways

Example: Java Monitor

Synchronized Data Class (Can also be implemented by Condition Variable)

Threads Competing resources

Shared Data, **Operations** and **wait()**, **notify()** are defined inside anObject

Task 1

```
synchronized (anObject) {
  try {
    // Wait for the condition to become true
    while (!condition)
      anObject.wait();

    // Do something when condition is true
  }
  catch (InterruptedException ex) {
    ex.printStackTrace();
  }
}
```

resume

Task 2

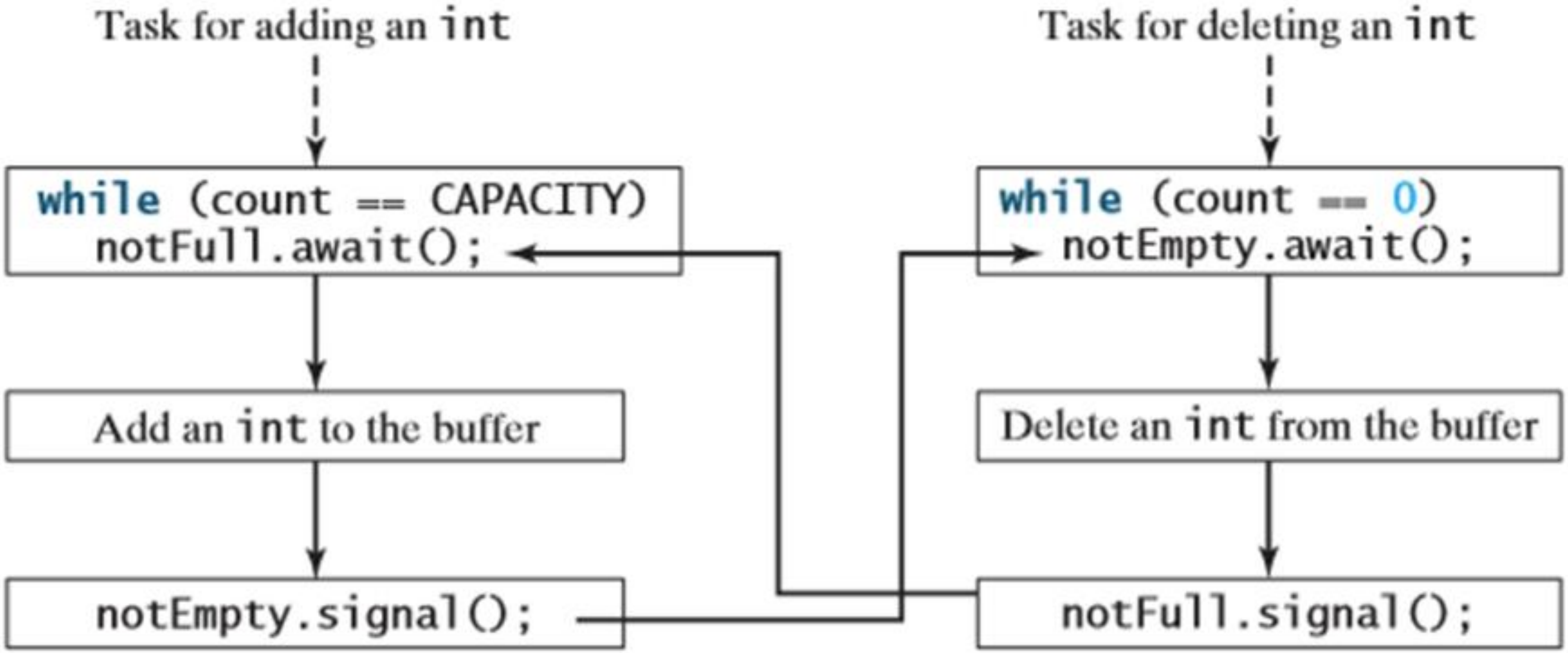
```
synchronized (anObject) {
  // When condition becomes true
  anObject.notify(); or anObject.notifyAll();
  ...
}
```

Java Provides synchronized class and synchronized methods for mutual exclusive use.
Here is an example of synchronized class.

wait() and signal():
notify() are notify(), notifyAll() are used in Java. It can be P(), V() or wait(), signal() in other languages.

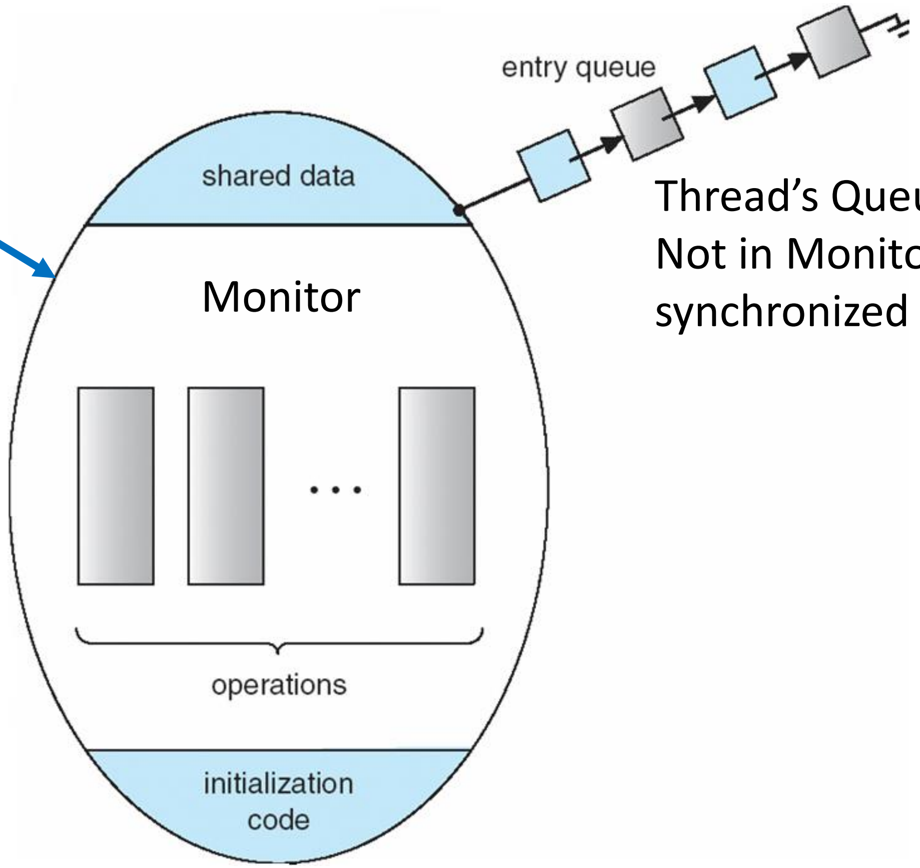
Example for Using Monitors

Consumer/Producer Model



Schematic View for Monitor

Shared Data Class and Operations



Thread's Queue in O.S. (or Java Virtual Machine) Not in Monitor (Created by the keyword synchronized in Java's case.

Monitor is a object with added mutual exclusive access feature.

Language-Level Mechanisms

Monitors' History

- They were suggested by Edsger W. Dijkstra, developed more thoroughly by Brinch **Hansen**, and formalized nicely by Tony **Hoare** (a real cooperative effort!) in the early 1970s
- Several parallel programming languages have incorporated monitors as their fundamental synchronization mechanism
  - none incorporate the precise semantics of Hoare's formalization

Language-Level Mechanisms

Monitors

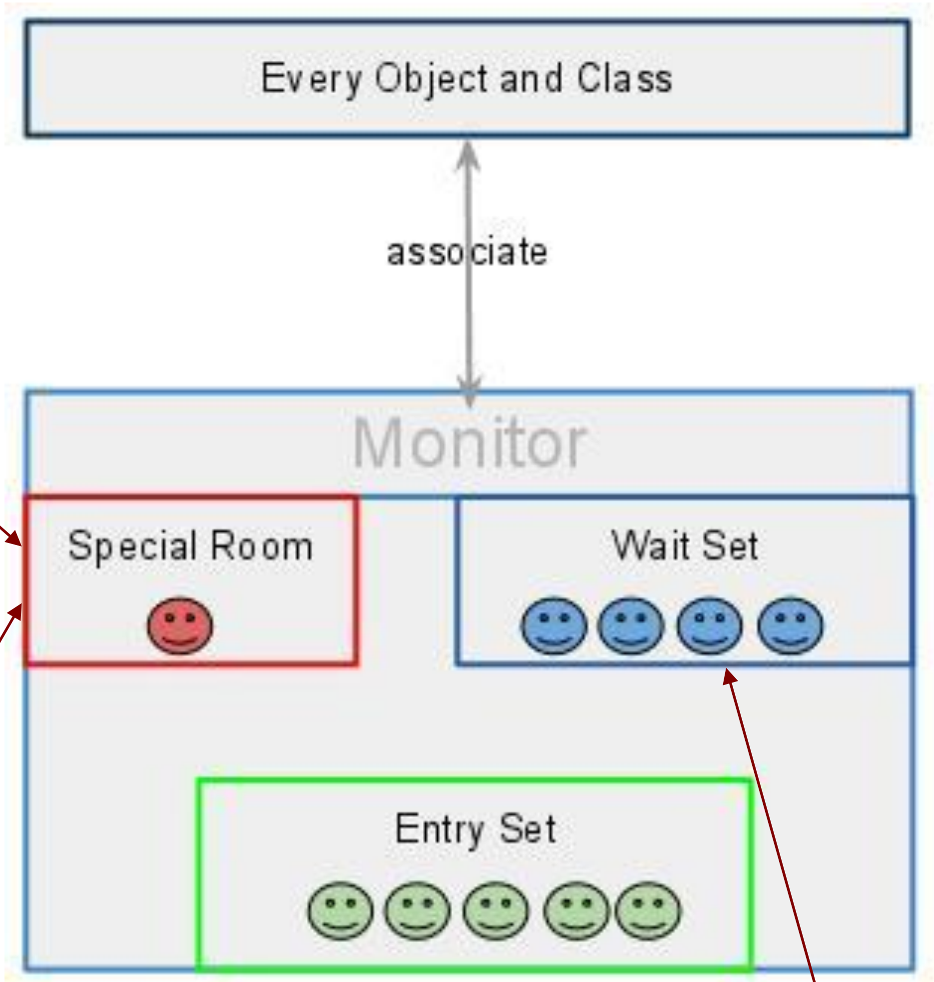
- A **monitor** is a shared object with operations, internal state, and a number of condition queues. Only one operation of a given monitor may be active at a given point in time
- A process that calls a busy monitor is delayed until the monitor is free
  - On behalf of its calling process, any operation may suspend itself by waiting on a condition
  - An operation may also signal a condition, in which case one of the waiting processes is resumed, usually the one that waited first

Language-Level Mechanisms

Monitors

- The precise semantics of mutual exclusion in monitors are the subject of considerable dispute. Hoare's original proposal remains the clearest and most carefully described
  - It specifies two bookkeeping queues for each monitor: an entry queue, and an urgent queue
  - When a process executes a signal operation from within a monitor, it waits in the monitor's urgent queue and the first process on the appropriate condition queue obtains control of the monitor
  - When a process leaves a monitor it unblocks the first process on the urgent queue or, if the urgent queue is empty, it unblocks the first process on the entry queue instead

Like doctor’s treatment room



In Critical Section , Access controlled by mutex

Like the lobby in doctor’s clinic

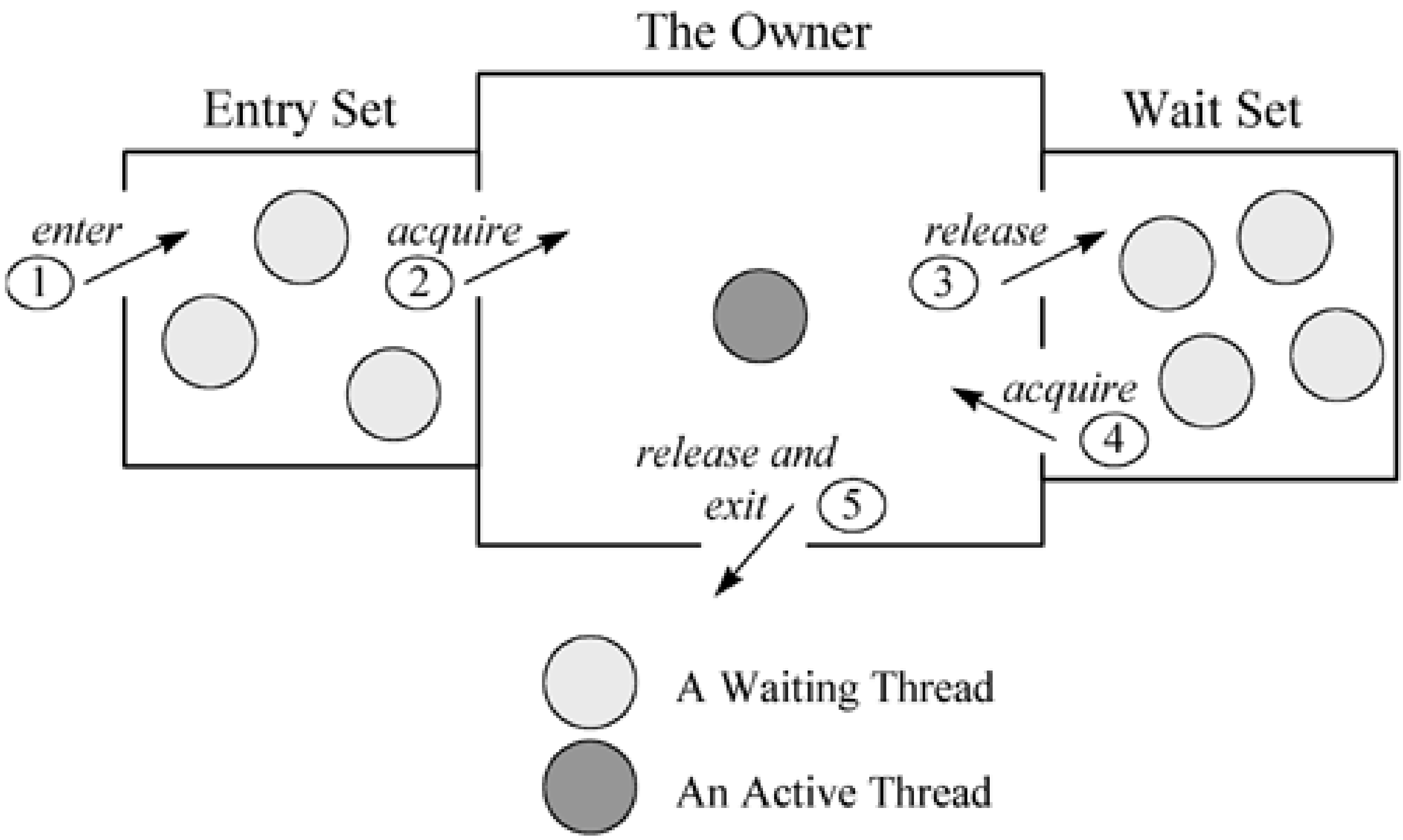
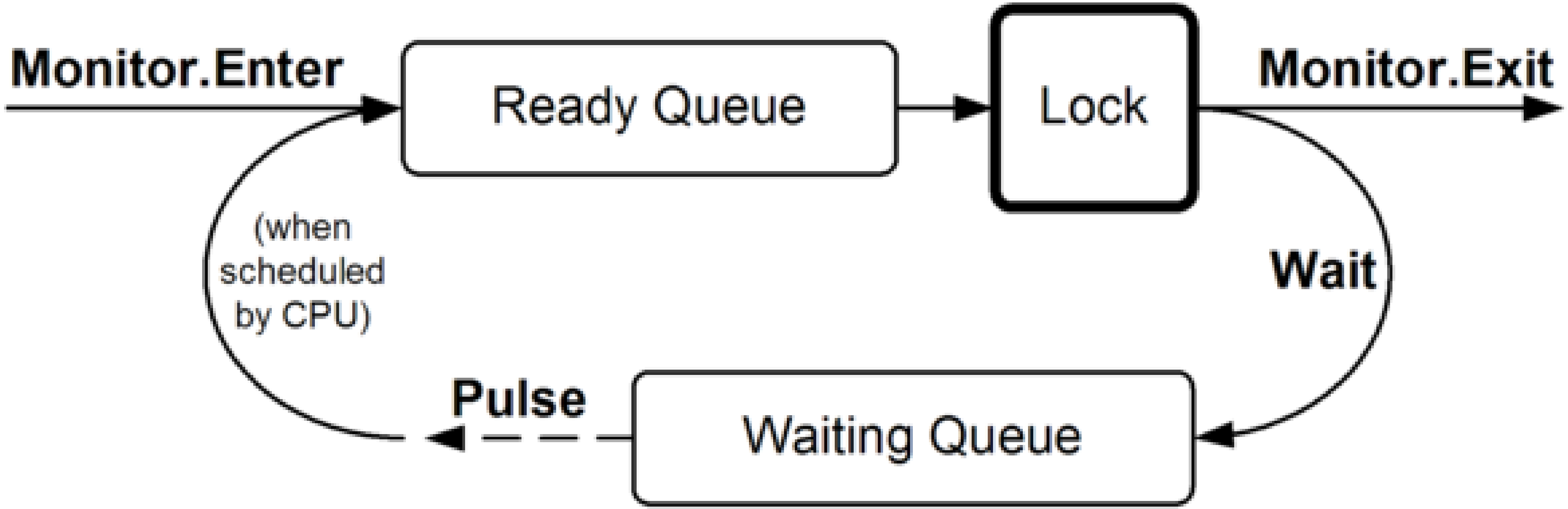


Figure 20-1. A Java monitor.



State Transition Diagram

Language-Level Mechanisms

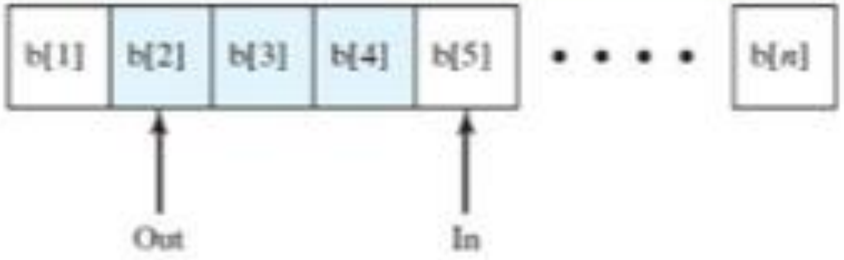
Monitors

- Building a correct monitor requires that one think about the "**monitor invariant**“. The monitor invariant is a predicate that captures the notion "the state of the monitor is consistent."
  - It needs to be true initially, and at monitor exit **(0<= # of items <= Buffer Size)**
  - It also needs to be true at every wait statement
  - In Hoare's formulation, needs to be true at every signal operation as well, since some other process may immediately run
- Hoare's definition of monitors in terms of semaphores makes clear that semaphores can do anything monitors can
- The inverse is also true; it is trivial to build a **semaphores** from **monitors**

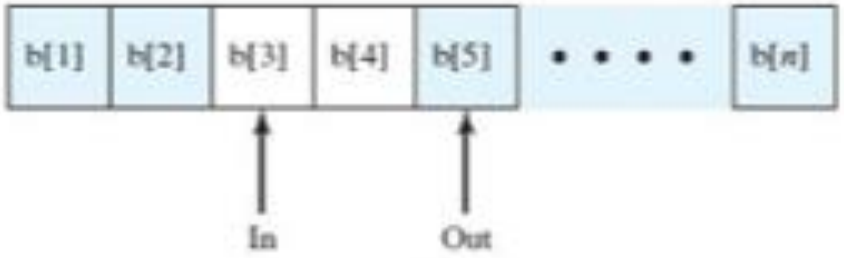
This Example is using Condition Variable

Bounded Buffer

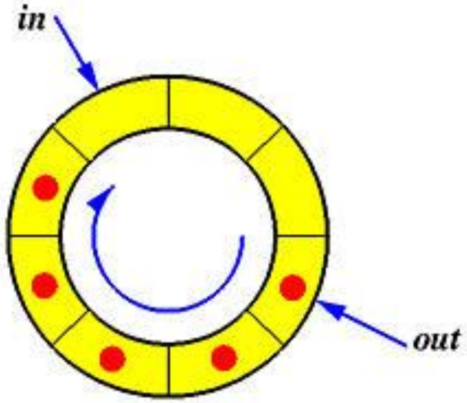
Block on:	Unblock on:
Producer: insert in full buffer	Consumer: item inserted
Consumer: remove from empty buffer	Producer: item removed



(a)

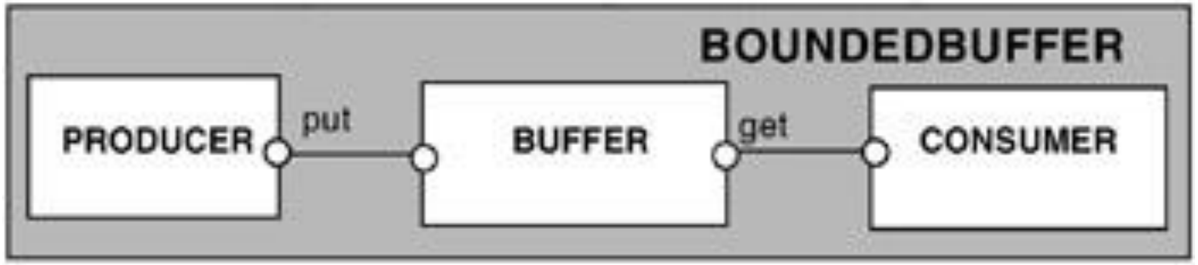


(b)



```
monitor bounded_buf
imports bdata, SIZE
exports insert, remove

    buf : array [1..SIZE] of bdata
    next_full, next_empty : integer := 1, 1
    full_slots : integer := 0
    full_slot, empty_slot : condition

    entry insert(d : bdata)
        if full_slots = SIZE
            wait(empty_slot)
        buf[next_empty] := d
        next_empty := next_empty mod SIZE + 1
        full_slots +:= 1
        signal(full_slot)

    entry remove : bdata
        if full_slots = 0
            wait(full_slot)
        d : bdata := buf[next_full]
        next_full := next_full mod SIZE + 1
        full_slots −:= 1
        signal(empty_slot)
        return d
```

Figure 13.17: **Monitor-based code for a bounded buffer.** Insert and remove are *entry* subroutines: they require exclusive access to the monitor's data. Because conditions are memory-less, both insert and remove can safely end their operation with a signal.



Language-level Mechanisms II

Monitor Implementation

SECTION 2

Semaphore

IMPLEMENTATION

A Simple Semaphore Design

Usage of Semaphore

```
Semaphore mutex =  new Semaphore();

do {
  wait(mutex);
  // critical section
  signal(mutex);
  // remaining parts
} (true);
```

Semaphore Definition

Wait:

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

Signal:

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

A Simple Version of Semaphore

- S (count) keep track of how many remaining seats allowed to enter critical section.
- Initial Condition of S keep track of the maximum number of seats in critical section.
- Wait and Signal are not always paired. (Maybe more Signal() executed than Wait(). Then, the maximum number of seats will be increased.

Monitor

IMPLEMENTATION

Implementing a Monitor Using Semaphores

Global variables

- **mutex**: Semaphore initialized to 1; It is used to control the number of processes allowed in the monitor.(critical section)
- **next**: Semaphore initialized to 0; it is used as a waiting queue by processes that are in the monitor after being released from a condition's queue by a signal operation. (waiting queue)
- **next_count**: integer variable initialized to 0; It counts the number of processes sleeping in the next semaphore. It is always equal to the number of processes executing monitor operations, minus 1.

[This is a complex way of saying that only one process at one time can be actively executing within a monitor operation, this process is called the *active process*. The processes waiting on next are the *inactive processes*]

Implementing a Monitor Using Semaphores

- Entrance Code executed when starting execution of a monitor's operation:

```
P(mutex);
```

- Exit Code executed when ending execution of a monitor's operation:

```
if next_count > 0
   then V(next)
   else V(mutex);
```

P(x): wait(x), **V(x):** signal(x): where x is a semaphore

Implementing a Monitor Using Semaphores

synchronized method(function)

If we originally has a method f() to be synchronized,

```
synchronized void f(){
  /* f method's body*/
}
```

Be compiled to

added to system

```
void f(){
  wait(mutex);                                P

  /* f method's body (mutual exclusive) */

  if (next_count>0)
        signal(next);
     else
        signal(mutex);                        V
} /* added code in blue */
```

Monitor Implementation Using Semaphores Variables:

```
semaphore mutex; // (initially = 1)
semaphore next;  // (initially = 0)
int next_count = 0;
/* In system or JVM, not seen by programmer */
```

All instance method in a synchronized class should be added with this feature to become monitor.

Monitor Buffer Class

- Using synchronized method in a class.
- Mutually exclusive access in methods.
- wait() and notify() are available in a synchronized method

Instance Synchronized Method

```
void myMethod() {
  synchronized(this) {
    //code
  }
}
```

equivalent to

```
synchronized void myMethod() {
  //code
}
```

```
class Buffer {
      private char [] buffer;
      private int count = 0, in = 0, out = 0;

      Buffer(int size)
      {
            buffer = new char[size];
      }

      public synchronized void Put(char c) {
            while(count == buffer.length)
            {
                  try { wait(); }
                  catch (InterruptedException e) { }
                  finally { }
            }
            System.out.println("Producing " + c + " ...");
            buffer[in] = c;
            in = (in + 1) % buffer.length;
            count++;
            notify();
      }

      public synchronized char Get() {
            while (count == 0)
            {
                  try { wait(); }
                  catch (InterruptedException e) { }
                  finally { }
            }
            char c = buffer[out];
            out = (out + 1) % buffer.length;
            count--;
            System.out.println("Consuming " + c + " ...");
            notify();
            return c;
      }
}
```

Synchronized Class Method

```
static void myMethod() {
  synchronized(MyClass.class) {
    //code
  }
}
```

equivalent to

```
static synchronized void myMethod() {
  //code
}
```

Monitor Features

- Java has built-in Monitor feature.
- C/C++ don’t. Need to use pthreads library.

Condition Variable

IMPLEMENTATION

Condition Variable

(Improved from Semaphore with Threads in Queue)

- Consider an operating system. An I/O request from a user program via a system call may not be completed immediately (e.g., waiting for an I/O operation to complete). Should this happen, the operating system may postpone this I/O service and serve another user until the I/O operation completes. Then, the operating system resumes the I/O service. In this case, the user "feels" that he is blocked within the operating system.

- For a similar reason, a thread in a monitor may have to block itself because of its request may not complete immediately. **This waiting for an event (e.g., I/O completion) to occur is realized by condition variables.**

Condition Variable

(Improved from Semaphore with Threads in Queue)

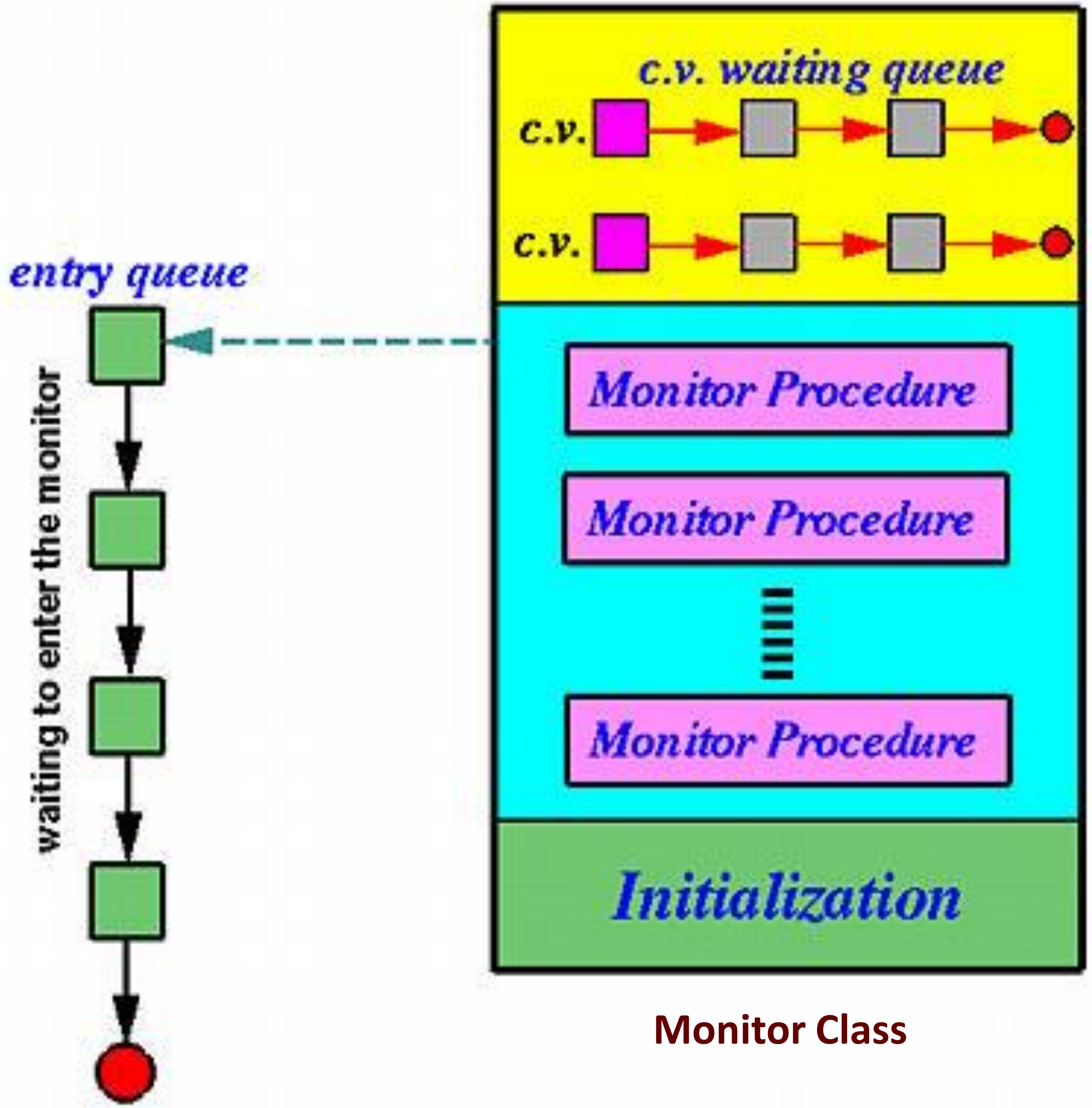
- **A condition variable** indicates an event and **has no value**. More precisely, one cannot store a value into nor retrieve a value from a condition variable. **If a thread must wait for an event to occur, that thread waits on the corresponding condition variable.** If another thread causes an event to occur, that thread simply signals the corresponding condition variable.
- Thus, **a condition variable has a queue for those threads** that are waiting the corresponding event to occur to wait on, and, as a result, the original monitor is extended to the following. The private data section now can have a number of condition variables, each of which has a queue for threads to wait on.

Monitor and *c.v.*

Each control variable is associated with a resource.

A monitor class is a resource.

A **Condition Variable** works like semaphore.



Monitor Class

Condition Variables by Semaphores

Support conditional synchronization

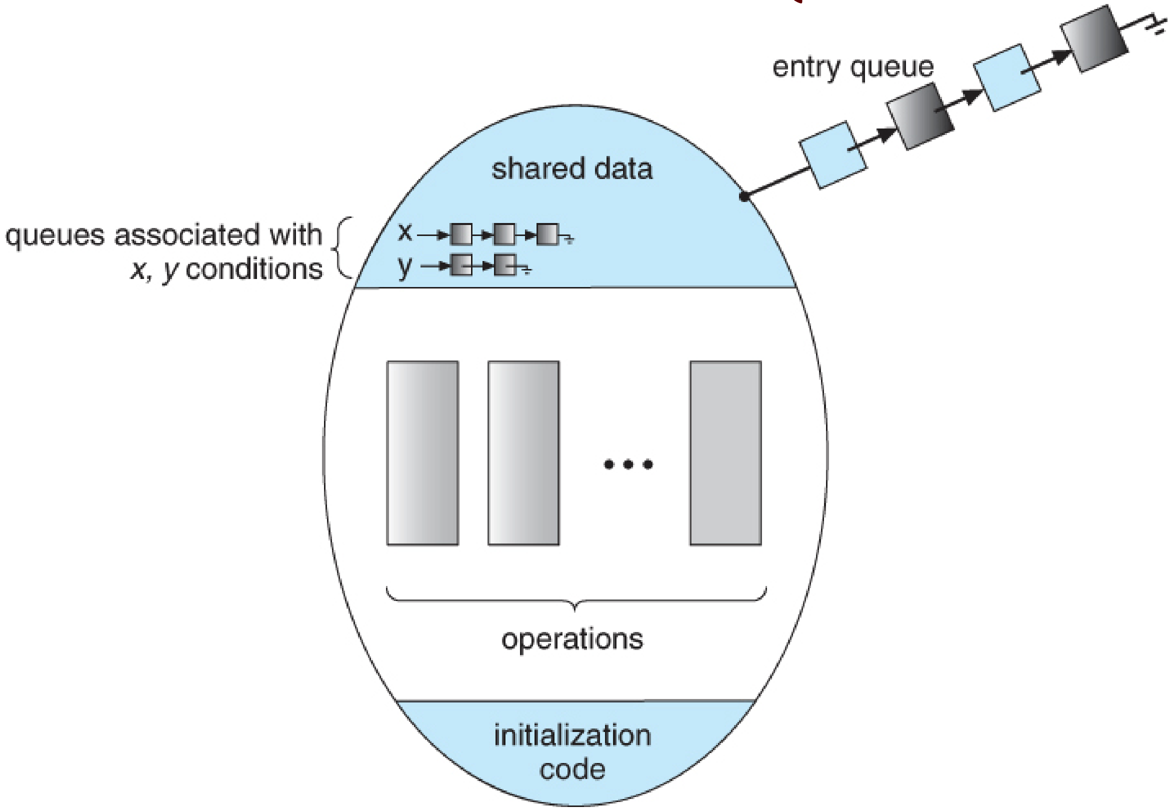
`condition x, y;`

Two operations on a condition variable:

`x.wait()` – a process that invokes the operation is suspended.

`x.signal()` – resumes of processes (if any) that invokded by `x.wait()`

Thread or Monitor Event Queue



Implementation of Condition Variables

```
condition x:
    semaphore sem; // initially = 0;
    int count=0;

 wait() {
        count++;
        if (next_count>0)
               signal(next);
        else
               signal(mutex);
        wait(x.sem);
        count--;
 } //P(x)

signal() {
    if (count >0){
           next_count++;
           signal(sem);
           wait(next);
           next_count--;
    }
} // V(x)
```

Monitor Implementation Using Condition Variables:

```
semaphore mutex; // (initially = 1)
semaphore next;     // (initially = 0)
int next_count = 0;
/* In system or JVM, not seen by programmer */
```

- Implement Monitor using Condition Variable is just like using semaphore except that it has a queue.

Language-level Mechanisms III

Conditional Critical Region

SECTION 3

Conditional Critical Region

- Conditional critical regions (**CCR**s) are another alternative to semaphores.
- A critical region is a syntactically **delimited critical section**.
- A conditional critical region also specifies a Boolean condition, which must be true before control will enter the region:

region *protected_variable*, when *Boolean_condition* do
    . . .
end region

- No thread can access a protected variable except within a region statement for that variable.
- Regions can nest, though as with nested monitor calls, the programmer needs to worry about deadlock.

Conditional Critical Regions for a Bounded Buffer

- Boolean conditions on the region statements eliminate the need for explicit condition variables.
- Condition variable has no value. Boolean condition in CCR can have variable and expression.

```
buffer : record
    buf : array [1..SIZE] of bdata
    next_full, next_empty : integer := 1, 1
    full_slots : integer := 0

procedure insert(d : bdata)
    region buffer when full_slots < SIZE
        buf[next_empty] := d
        next_empty := next_empty mod SIZE + 1
        full_slots −:= 1

function remove() : bdata
    region buffer when full_slots > 0
        d : bdata := buf[next_full]
        next_full := next_full mod SIZE + 1
        full_slots +:= 1
    return d
```

Protected Variable

Access Condition

CCR – Conditional Critical Region

```
atomic (condition) {
    statements;
}
```

Means

1. Wait until Condition is satisfied
2. Execute statements atomically

Pattern

```
public int get() {
  atomic (items != 0) {
    items --;
    return buffer[items];
  }
}
```

Example Use

CCR Implementation

- Built on top of Software Transactional Memory
- Uses all traditional **STM** commands and **STMWait**
- Like implementation of Monitor using Semaphore. Implementation of CCR can use Software Transactional Memory (**STM**)

```
atomic (condition) {
    statements;
}
```

Compiled to

```
boolean done = false;
while (!done) {
    STMStart ();
    try {
        if (condition) {
            statements;
            done = STMCommit ();
        } else {
            STMWait();
        }
    } catch (Throwable t) {
        done = STMCommit ();
        if (done) {
            throw t;
        }
    }
}
```

Language-level Mechanisms IV

Synchronization in Java

SECTION 4

Language-level Mechanisms III (Synchronization in Java)

CHAPTER 13, LECTURE M11

Complete Video Tutorials

http://ec.teachable.com/p/java-concurrent-programming-multithreading-and-multicore

- Video tutorial created by Dr. Eric Chou
- Including Current Programming Topics (This Chapter) and all **Java Multi-threading Programming** issues.
- Lectures
- Example programs
- Quizzes
- Reference document download and web-links

Synchronization in Java

- Thread, Future, and Callable classes and related pools.
- Semaphore, newCondition classes
  - Lock, ReentrantLock classes or mutual exclusive access (locks)
  - Semaphore using Semaphore class
  - Monitor design using **synchronized** keyword for lock variables.
  - Condition variables using newCondition class.

Code Examples

Critical Section:

```
synchronized (my_shared_obj) {
    ...     // critical section
}
```

Lock variable:

```
Lock l = new ReentrantLock();
l.lock();
try {
    ...     // critical section
} finally {
    l.unlock();
}
```

Semaphore:

```
private static class Account {
  // Create a semaphore
  private static Semaphore semaphore = new Semaphore(1);
  private int balance = 0;

  public int getBalance() {
    return balance;
  }

  public void deposit(int amount) {
    try {
      semaphore.acquire(); // Acquire a permit
      int newBalance = balance + amount;
      balance = newBalance;
    }
    catch (InterruptedException ex) {
    }
    finally {
      semaphore.release(); // Release a permit
    }
  }
}
```

Code Examples

Condition Variable:

```
Condition c1 = l.newCondition();
Condition c2 = l.newCondition();
...
c1.await();
...
c2.signal();
```

Monitor:

```
public class SimpleMonitor {
    private final Lock lock = new ReentrantLock();

    public void testA() {
        lock.lock();

        try {
            //Some code
        } finally {
            lock.unlock();
        }
    }

    public int testB() {
        lock.lock();

        try {
            return 1;
        } finally {
            lock.unlock();
        }
    }
}
```

//Lock can be replaced by **Semaphore**, **Condition Variable**

The Java Memory Model

- **The Java Memory Model** specifies exactly which operations are guaranteed to be ordered across threads.
- It also specifies, for every pair of reads and writes in a program execution, whether the read is permitted to return the value written by the write.
- A Java thread is allowed to buffer or reorder its writes until the point at which it writes a **`volatile`** variable or leaves a monitor (releases a lock, leaves a synchronized block, or waits).
- At that point all its previous writes must be visible to other threads. Similarly, a thread is allowed to keep cached copies of values written by other threads until it reads a volatile variable or enters a monitor (acquires a lock, enters a **`synchronized`** block, or wakes up from a wait). At that point any subsequent reads must obtain new copies of anything that has been written by other threads.

The Java Memory Model

volatile is related to write-back scheme

- The compiler is free to reorder ordinary reads ad write in the absence of intra-thread data dependencies.
- If the compiler can prove that a **volatile** variable or monitor isn't used by more than one thread during a given interval of time, it can reorder its operations like ordinary accesses.

Language-level Mechanisms V

Transactional Memory

SECTION 5

What is STM?

Software Transaction Memory

Algorithms for Database-like Data Access

STM usually is a language extension or API framework. It is jut like another currency control mechanism like locks, actors, semaphore, condition variable or CCR.

ACID Not for database. It’s for memory transactions.

A → Atomicity

C → Consistent

I → Isolation

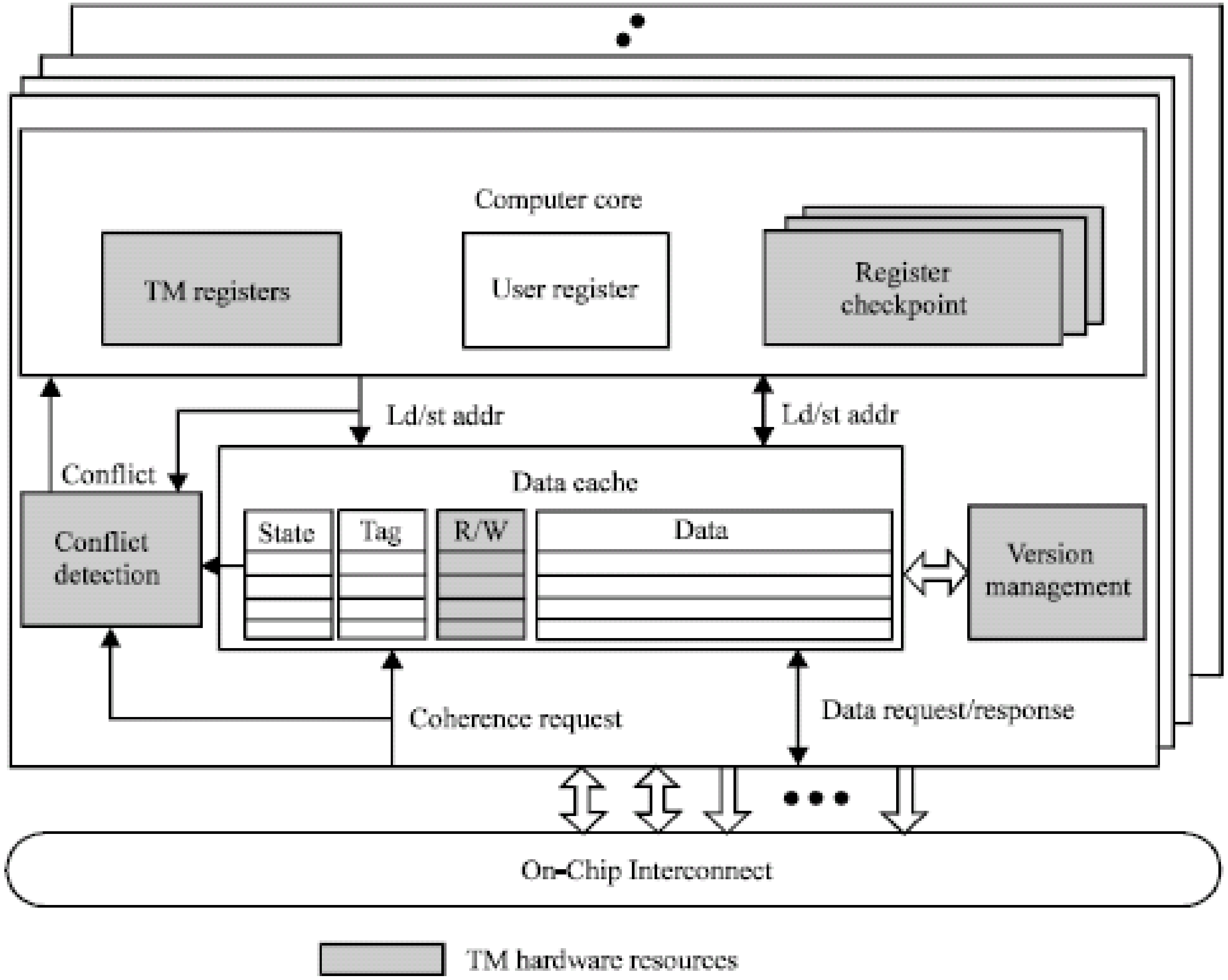
~~D~~ → ~~Durability~~

Transactional Memory

Used to Implement Conditional Critical Region

- **Transactional memory** attempts to simplify concurrent programming by allowing a group of load and store instructions to execute in an **atomic** way.
- It is a concurrency control mechanism analogous to database transactions for controlling access to shared memory in concurrent computing.
- Transactional memory systems provide high level abstraction as an alternative to low level thread synchronization.
- This abstraction allows for coordination between **concurrent reads and writes of shared data in parallel systems**.

Hardware Transactional Memory



STM – HTM Parallels: 3 Tier Implementation

HTM	STM
Transaction Runtime	Transaction Descriptors
Snoopy Cache	Ownership Records
Shared Memory	Application Heap

Software Transaction Management

- **STMStart():(start-memory-cycle)** begins a new transaction within the executing thread.
- **STMAbort**():**(write-miss)** aborts the transaction in progress by the executing thread.
- **STMread**(): get the right value and version numbers.
- **STMWrite**(): create value and version numbers.
- **STMCommit**():**(write-hit)** attempts to commit the transaction in progress by the executing thread, returning true if this succeeds and false if it fails.

Software Transaction Management

- **STMValidate**(): indicates whether the current transaction would be able to commit: that is, whether the values read represent a current and mutually consistent snapshot and whether any locations updated have been subject to conflicting updates by another transaction. It is an error to invoke **STMStart** if the current thread is already running a transaction. Similarly, it is an error to invoke the other operations unless there is a current transaction.
- **STMWait**():**(sleep and wait on signal)** is one that we introduce for allowing threads to block on entry to a CCR. It ultimately has the effect of aborting the current transaction. However, before doing so, it can delay the caller until it may be worthwhile attempting the transaction again. In a simplistic implementation **STMWait** would be equivalent to **STMAbort**, leading to callers spin-waiting. In our implementation, STMWait blocks the caller until an update may have been committed to one of the locations that the transaction has accessed.

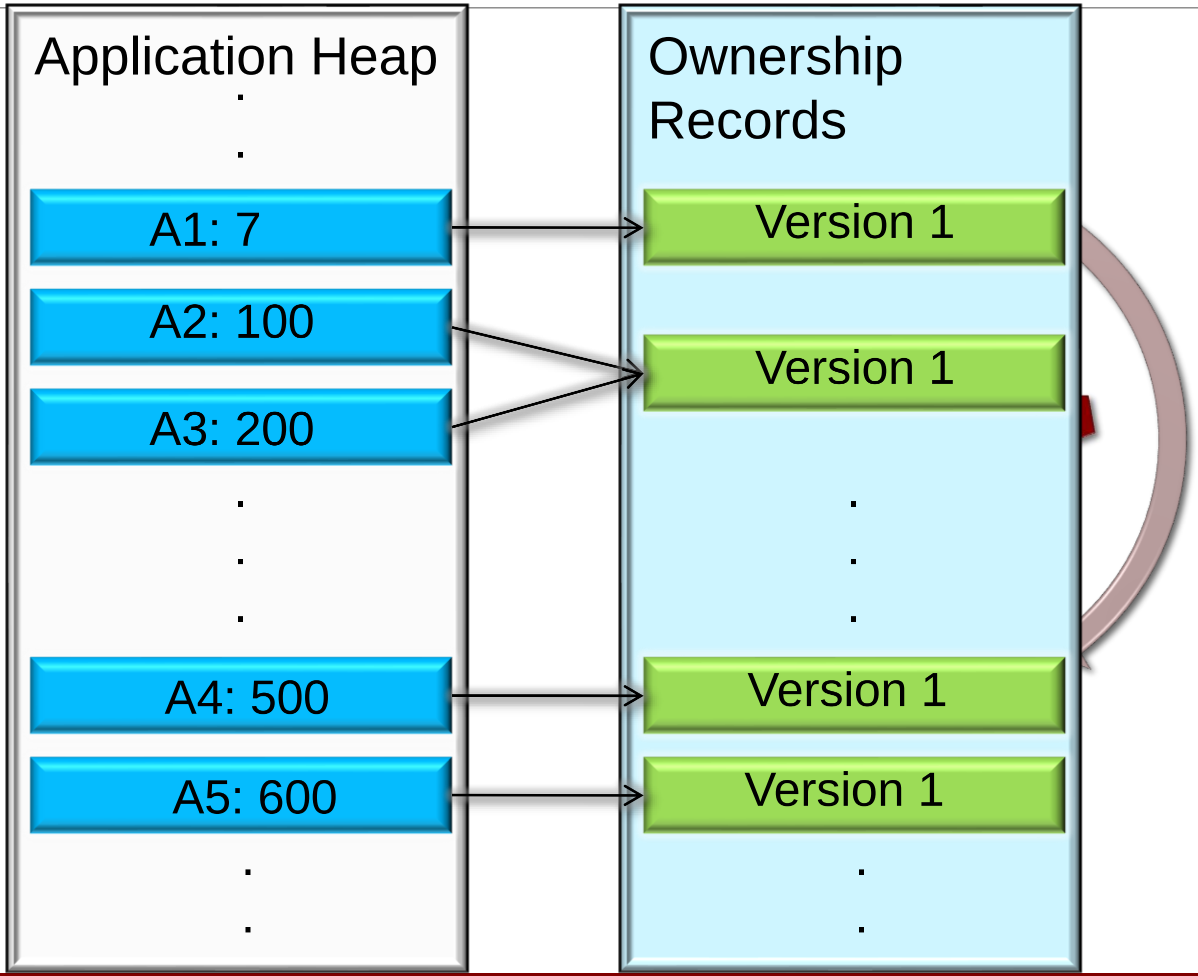
```
struct orec
    owned : Boolean
    val : union (time, transaction_id)

function read(x : address) : value
    if x ∈ write_map.domain then return write_map[x]
    loop
        repeat
            o : orec := orecs[hash(x)]
        until not o.owned
        t : time := o.val     –– when last modified
        if t > valid_time
            –– may be inconsistent with previous reads
            validate()        –– attempt to extend valid_time
        v : value := *x
        if o = orecs[hash(x)]
            read_set +:= {x}
            return v

procedure validate()
    t : time := clock
    for x : address ∈ read_set
        o : orec := orecs[hash(x)]
        if (not o.owned and o.val > valid_time)
                or (o.owned and o.val ≠ me)
            throw abort
    valid_time := t

procedure write(x : address, v : value)
    write_map[x] := v

procedure commit()
    try
        lock_map : map address → orec := ∅
        done : Boolean := false
        for x : address ∈ write_map.domain
            o : orec := orecs[hash(x)]
            if o ≠ ⟨true, me⟩
                if o.owned then throw abort
                if not CAS(&orecs[hash(x)], o, ⟨true, me⟩)
                    throw abort
                lock_map[x] := o
        n : time := 1 + fetch_and_increment(&clock)
        validate()
        done := true
        for ⟨x, v⟩ : ⟨address, value⟩ ∈ write_map
            *x = v                              –– write back
    finally
        –– do this however control leaves the try block
        for ⟨x, o⟩ : ⟨address, orec⟩ ∈ lock_map
            orecs[hash(x)] := if done
                then ⟨false, n⟩           –– update
                else o                    –– restore
```

Figure 13.19 Possible pseudocode for a **STM** system. The read and write routines are used to replace ordinary loads and stores within the body of the transaction. The validate routine is called from both read and commit. It attempts to verify that no previously read value has since been overwritten and, if successful, updates valid time. Various fence instructions (not shown) may be needed if the underlying hardware is not sequentially consistent.

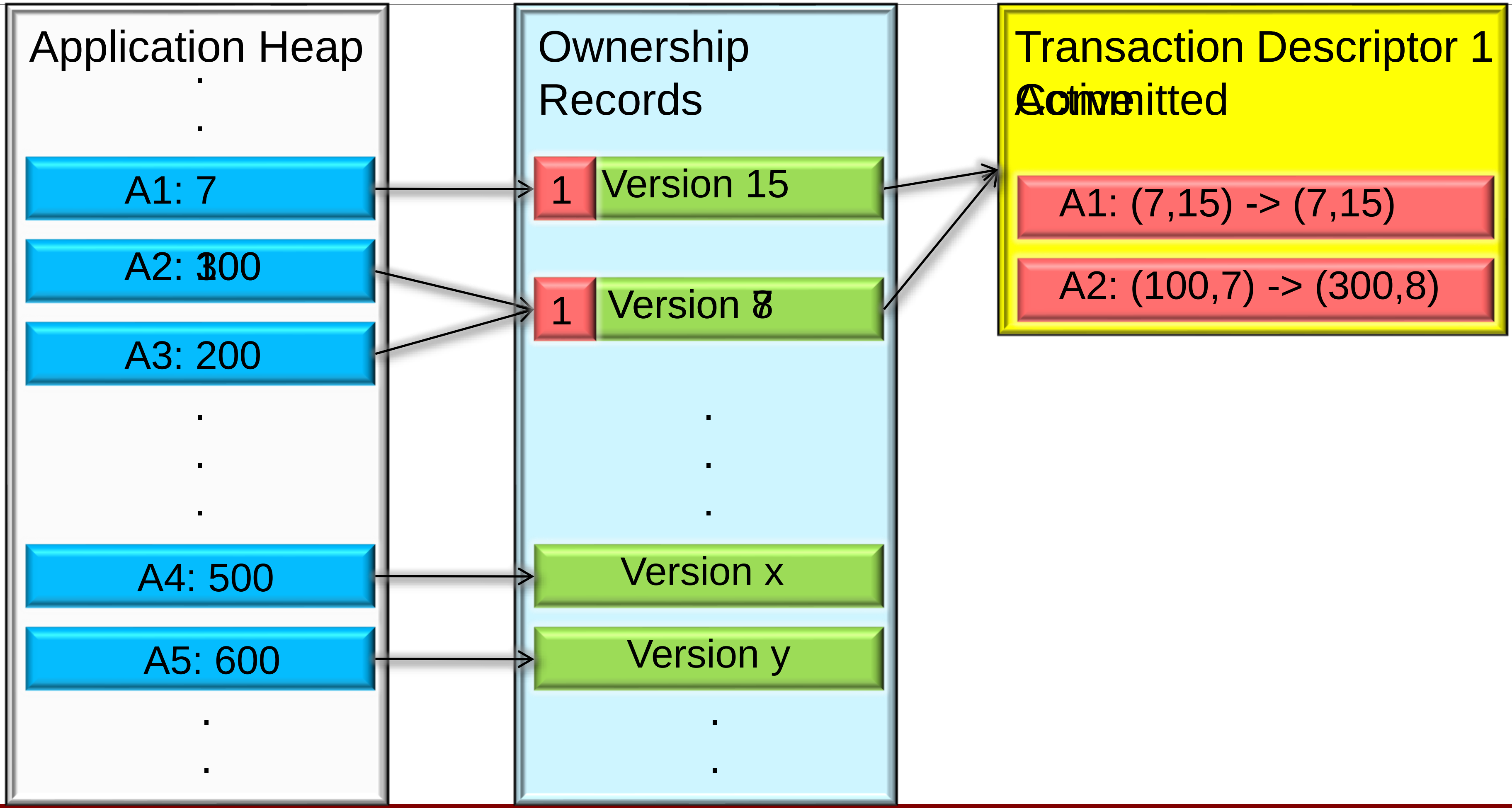
STM - Simple Transaction

```
boolean done = false;
while (true) {
   STMStart();
   readvalues;
   if(STMValidate()){
       statements;
       done = STMCommit();
       if(done) {
          break;
       }
   }
}
```

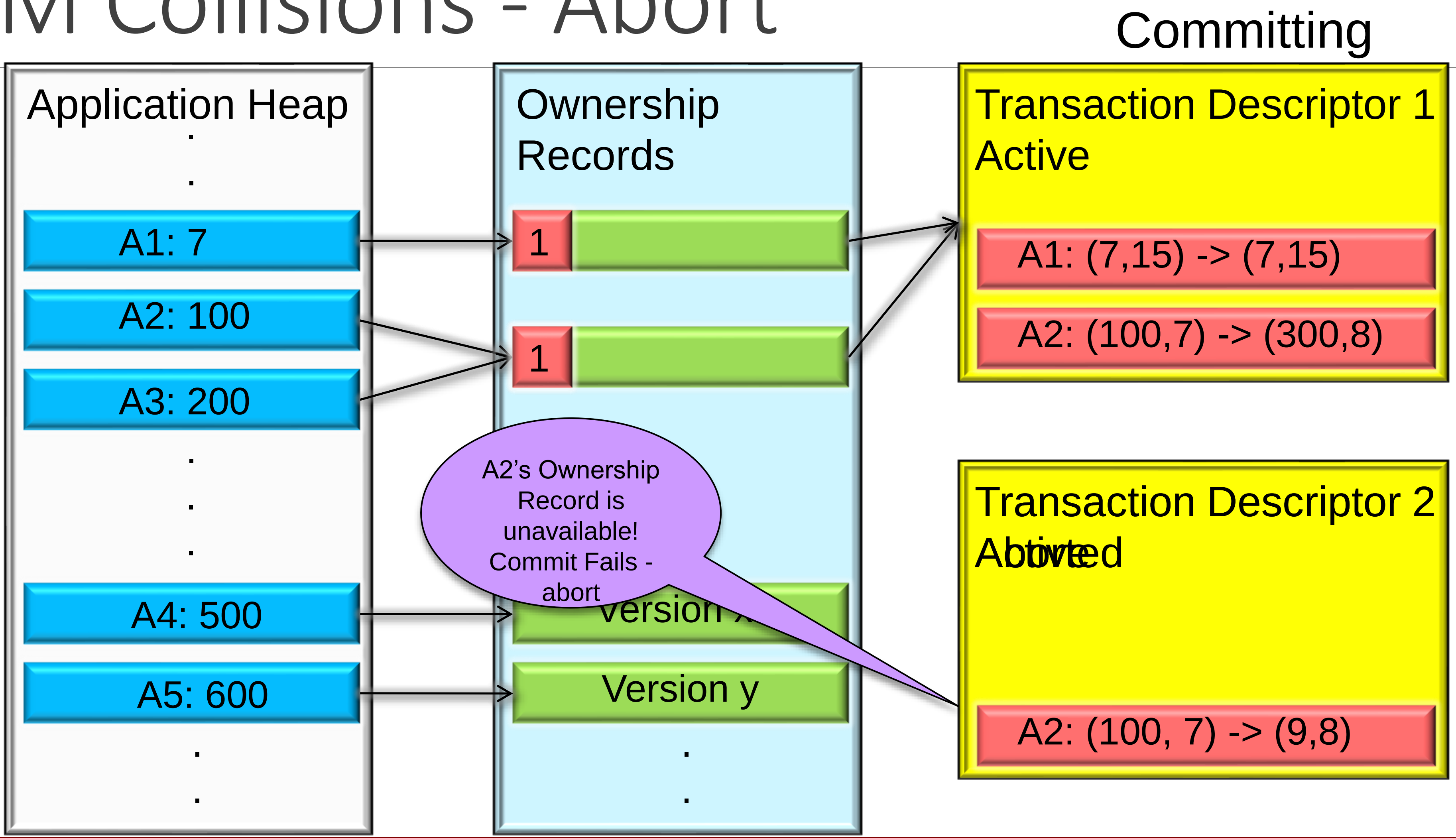
STM Heap Structure



STM - Simple Transaction



STM Collisions - Abort



STM Collisions - Sleep

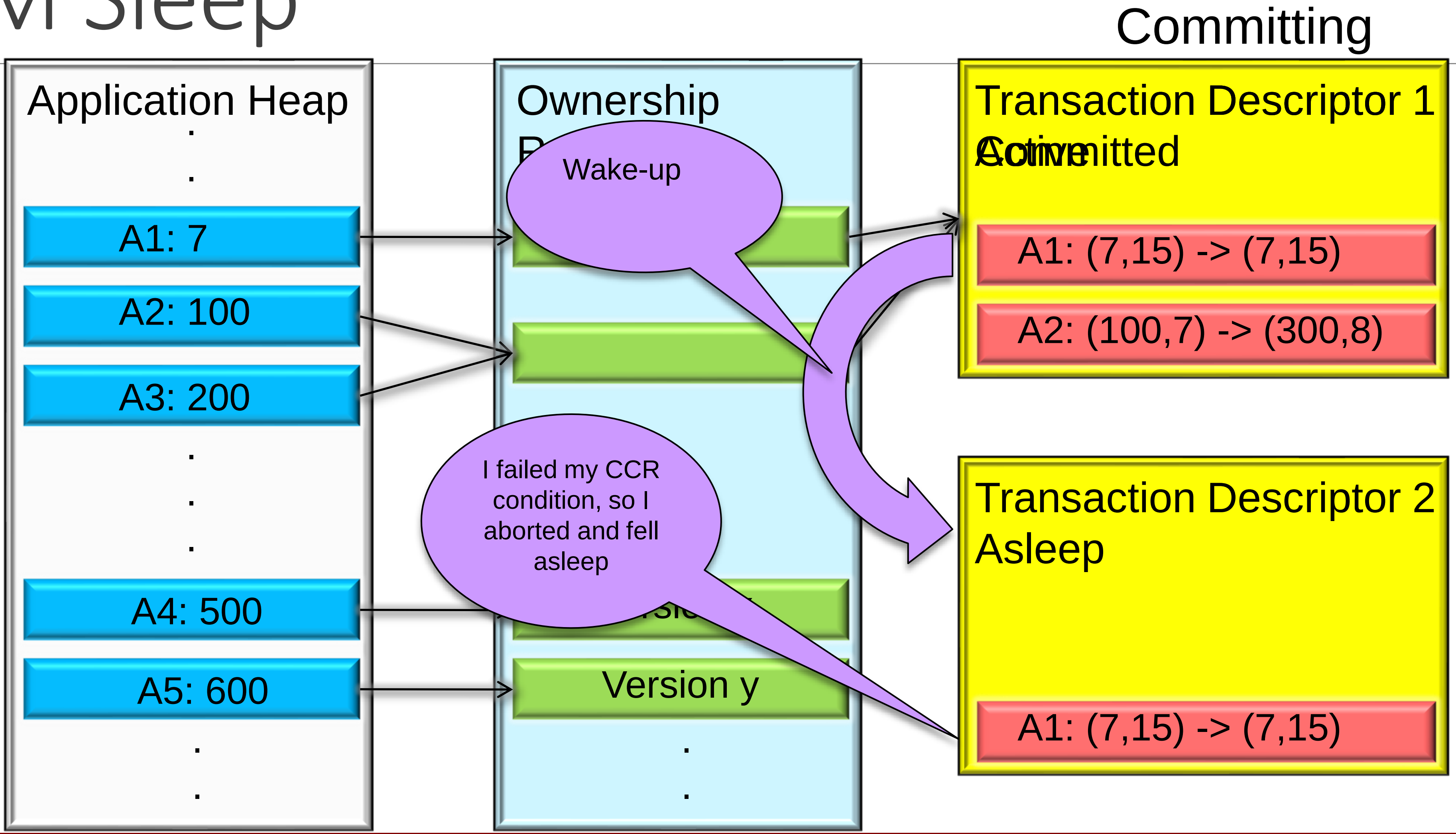
atomic (condition) {
    statements;
}

```
boolean done = false;
while (!done) {
   STMStart ();
   try {
      if (condition) {
      statements;
      done = STMCommit ();
      } else {
        STMWait();
      }
   } catch (Throwable t) {
      done = STMCommit ();
      if (done) {
         throw t;
      }
   }
}
```

Abort and wait

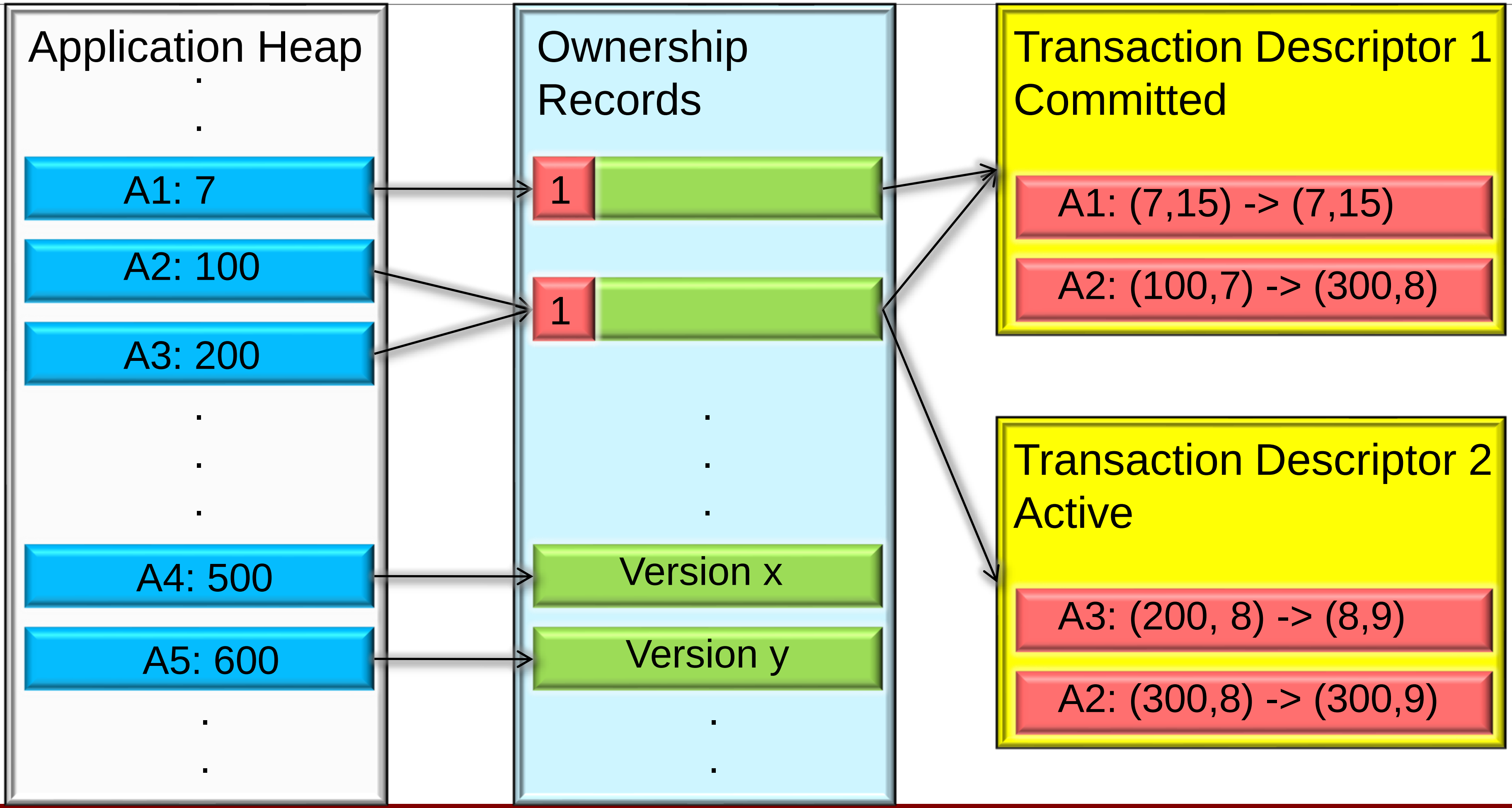
But when do I wake up?

STM Sleep

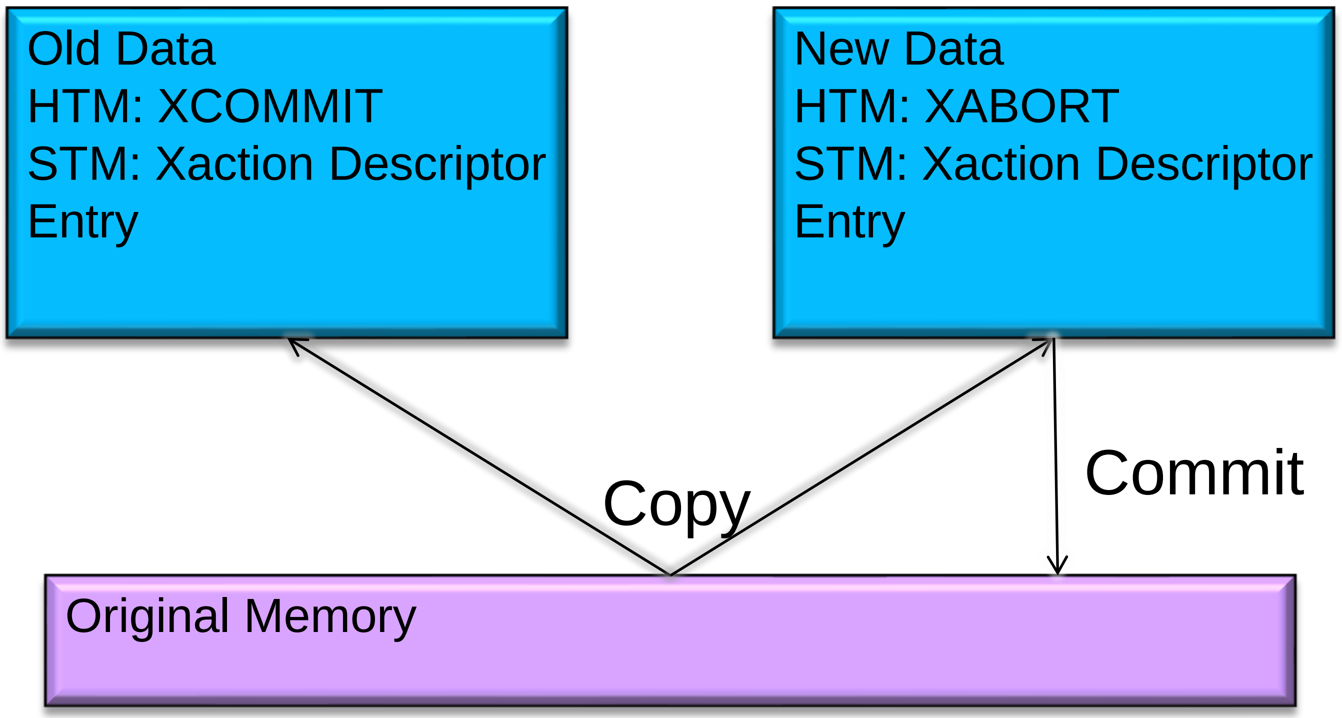


Committing
Application Heap
A1: 7
A2: 100
A3: 200
A4: 500
A5: 600
Ownership
Wake-up
I failed my CCR condition, so I aborted and fell asleep
Version y
Transaction Descriptor 1
Active
Committed
A1: (7,15) -> (7,15)
A2: (100,7) -> (300,8)
Transaction Descriptor 2
Asleep
A1: (7,15) -> (7,15)

STM Stealing - Optimization



STM – HTM Parallels: Data Copies



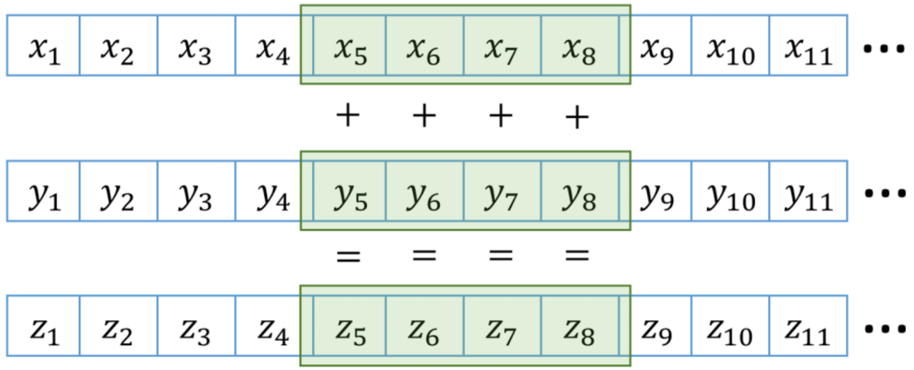
Language-level Mechanisms VI

Other Topics

SECTION 6

Topic: Implicit Synchronization

- In several shared-memory languages, the operations that threads can perform on shared data are implicit, rather than explicit operations. For example, the **forall** loop of HPF and Fortran 95.
- **forall loop**: each iteration reads all data used in its instance of the first assignment statement before any iteration updates its instance of the left-hand side. The left-hand side updates occur before any iteration reads the data used in its instance of the second assignment statement.
- Exploit the maximum Vector parallelism in data set, processing unit, and compiler technology. Dependence analysis plays a crucial role in other languages as well.



Vector Processing

Topic: Future

Future construct in Multilisp

- Implicit synchronization can also be achieved without compiler analysis.
- **`future`** construct in a dialect of Scheme: `(future (my-function my-args))`
- **`future`** is semantically **neutral**: assuming all evaluations terminate, program behavior will be exactly the same as if (my-function my-args) had appeared without the surrounding call.
- In the implementation, **`future`** arranges for the embedded function to be evaluated by a separate thread of control.

```
(parent (future (child1 args1)) (future (child2 args2)))
```

- There were no additional synchronization mechanisms in **Multilisp**: **`future`** itself was the language's only addition to Scheme.

Topic: Future

future construct in C#

- n C# 3.0 with Parallel FX

```
var description = Future.Create(() => GetDescription());
var numberInStock = Future.Create(() => GetInventory());
...
Console.WriteLine("We have " + numberInStock.Value
    + " copies of " + description.Value + " in stock.");
```

- Static class Future is a factory; its Create method supports generic type inference, allowing us to pass a delegate compatible with Func<T> (function returning T), for any T. We've specified the delegates here as lambda expressions.
- If GetDescription returns a String, description will be of type Future<String>; if GetInventory returns an int, numberInStock will be of type Future<int>.

Topic: Future

Future Interface in Java

- Callable – Runnable on steroids

```
Callable<V> {
  V call() throws Exception;
}
```

- Future – result of an asynchronous computation

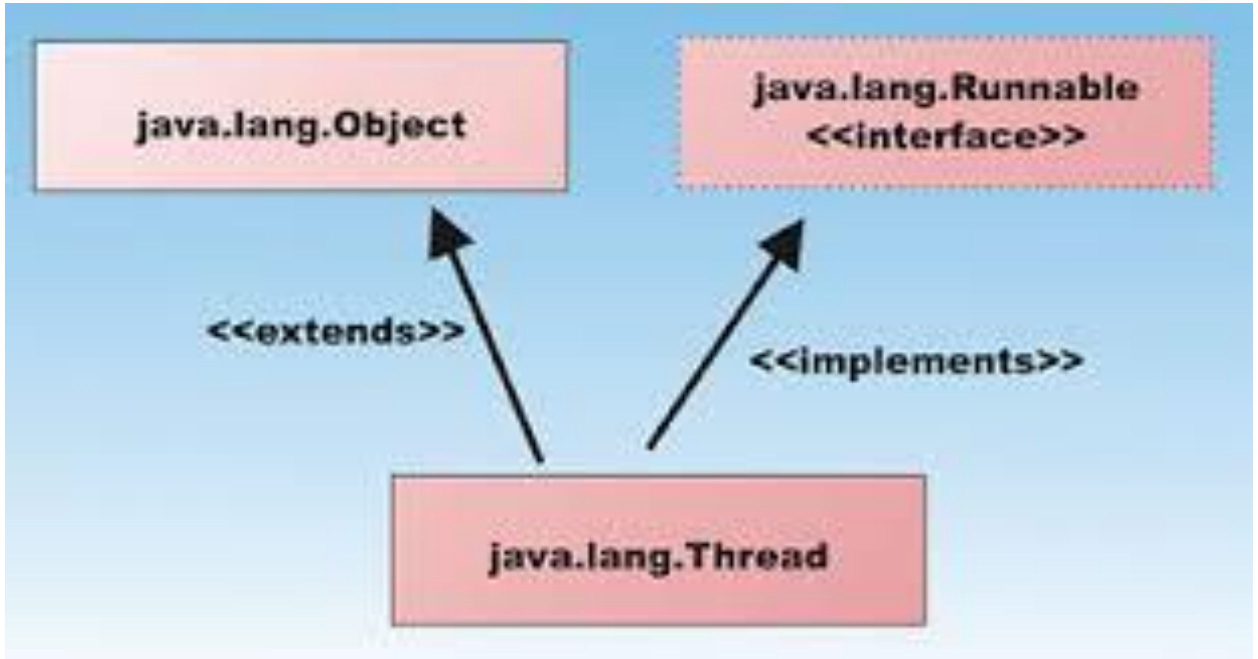
```
Future<V> {
  V get();
  boolean cancel();
  boolean isCancelled();
  boolean isDone();
}
```

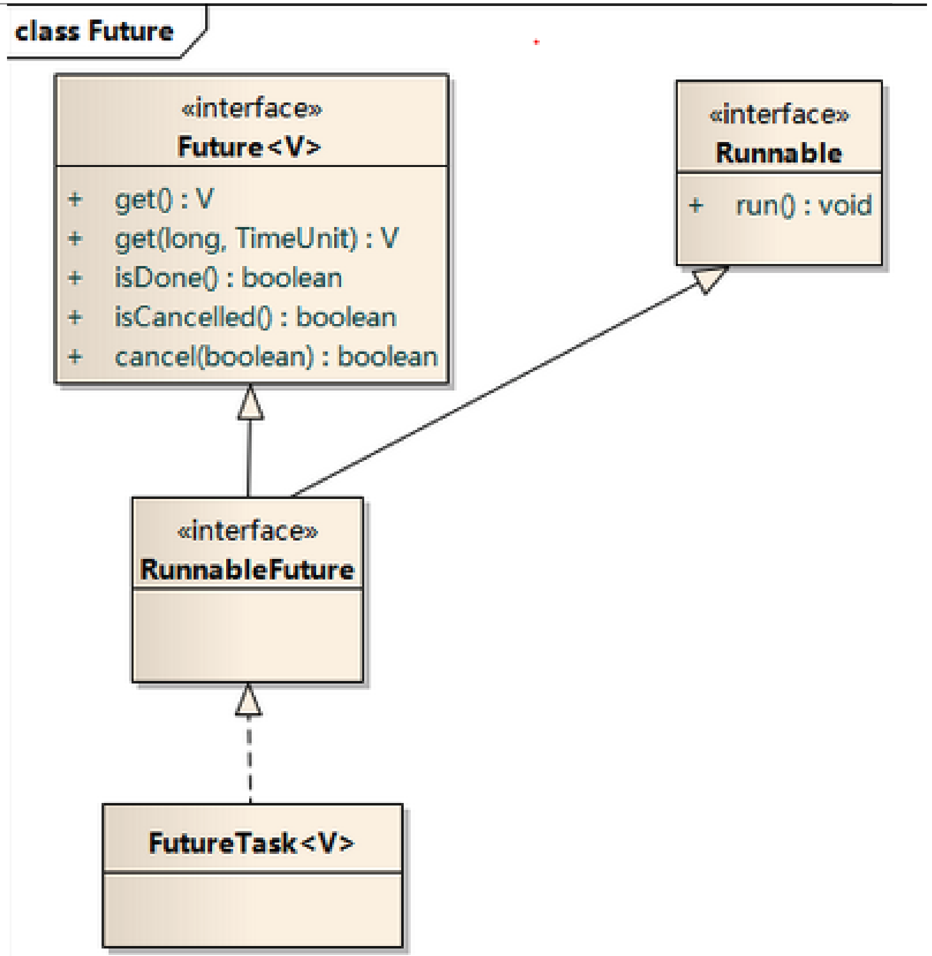
- Callable has 1 method
  - V call() throws Exception
  - Can return a value
  - Can throw exceptions
- Can be executed using an Executor object

- Runnable has 1 method
  - void run()
  - No return value
  - No exceptions
- Can be executed using an Executor object

Topic: Future

Runnable-Thread, Future-FutureTask





Topic: Future

Important points Future and FutureTask in Java

1. **Future** is a base interface and defines abstraction of an object which promises result to be available in future while **FutureTask** is an implementation of the Future interface.

2. **Future** is a parametric interface and type-safe written as **Future<V>**, where **V** denotes value.

3. Future provides **get()** method to get result, which is blocking method and blocks until result is available to Future.

4. Future interface also defines **cancel()** method to cancel task.

5. **isDone()** and **isCancelled()** method is used to query Future task states. **isDone()** returns true if task is completed and result is available to Future. If you call get() method, after **isDone()** returned true then it should return immediately. On the other hand, **isCancelled()** method returns true, if this task is cancelled before its completion.

Topic: Future

Important points Future and FutureTask in Java

6. **Future** has four sub interfaces, each with additional functionality e.g. **Response**, **RunnableFuture**, **RunnableScheduledFuture** and **ScheduledFuture**. **RunnableFuture** also implements **Runnable** and successful finish of **run()** method cause completion of this Future.

7. **FutureTask** and **SwingWorker** are two well known implementation of Future interface. **FutureTask** also implements **RunnableFuture** interface, which means this can be used as **Runnable** and can be submitted to **ExecutorService** for execution.

8. Though most of the time **ExecutorService** creates **FutureTask** for you, i.e. when you **submit() Callable** or **Runnable** object. You can also created it manually.

9. **FutureTask** is normally used to wrap **Runnable** or **Callable** object and submit them to **ExecutorService** for asynchronous execution.

Topic: Parallel Logic Programming

- There are two strategies for the backtracking search of logic languages such as Prolog is also amenable to parallelization:

  (1) AND parallelism: The fact that variables in logic, once initialized, are never subsequently modified ensures that parallel branches of an AND cannot interfere with one another.

  (2) OR parallelism: it pursues alternative resolutions in parallel. Because they will generally employ different unifications, branches of an OR must use separate copies of their variables.

Topic: Parallel Logic Programming

- AND parallelism and OR parallelism create new threads at alternating levels. OR parallelism is speculative: since success is required on only one branch, work performed on other branches is in some sense wasted. OR parallelism works well, however, when a goal cannot be satisfied (in which case the entire tree must be searched), or when there is high variance in the amount of execution time required to satisfy a goal in different ways (in which case exploring several branches at once reduces the expected time to find the first solution).
- Both AND and OR parallelism are problematic in Prolog, because they fail to adhere to the deterministic search order required by language semantics. Parlog [Che92], which supports both AND and OR parallelism, is the best known of the parallel Prolog dialects.

Topic: Message Passing

- Shared-memory concurrency as become ubiquitous on multicore processors and multiprocessor servers. Message passing still dominates both distributed and high-end computing.
- Supercomputers and large-scale clusters are programmed primarily in **Fortran** or **C/C++** with the **MPI** library package. Distributed computing increasingly relies on client-server abstractions layered on top of libraries that implement the **TCP/IP** Internet standard.
- As in shared-memory computing, scores of message-passing languages have also been developed for particular application domains, or for research or pedagogical purposes.