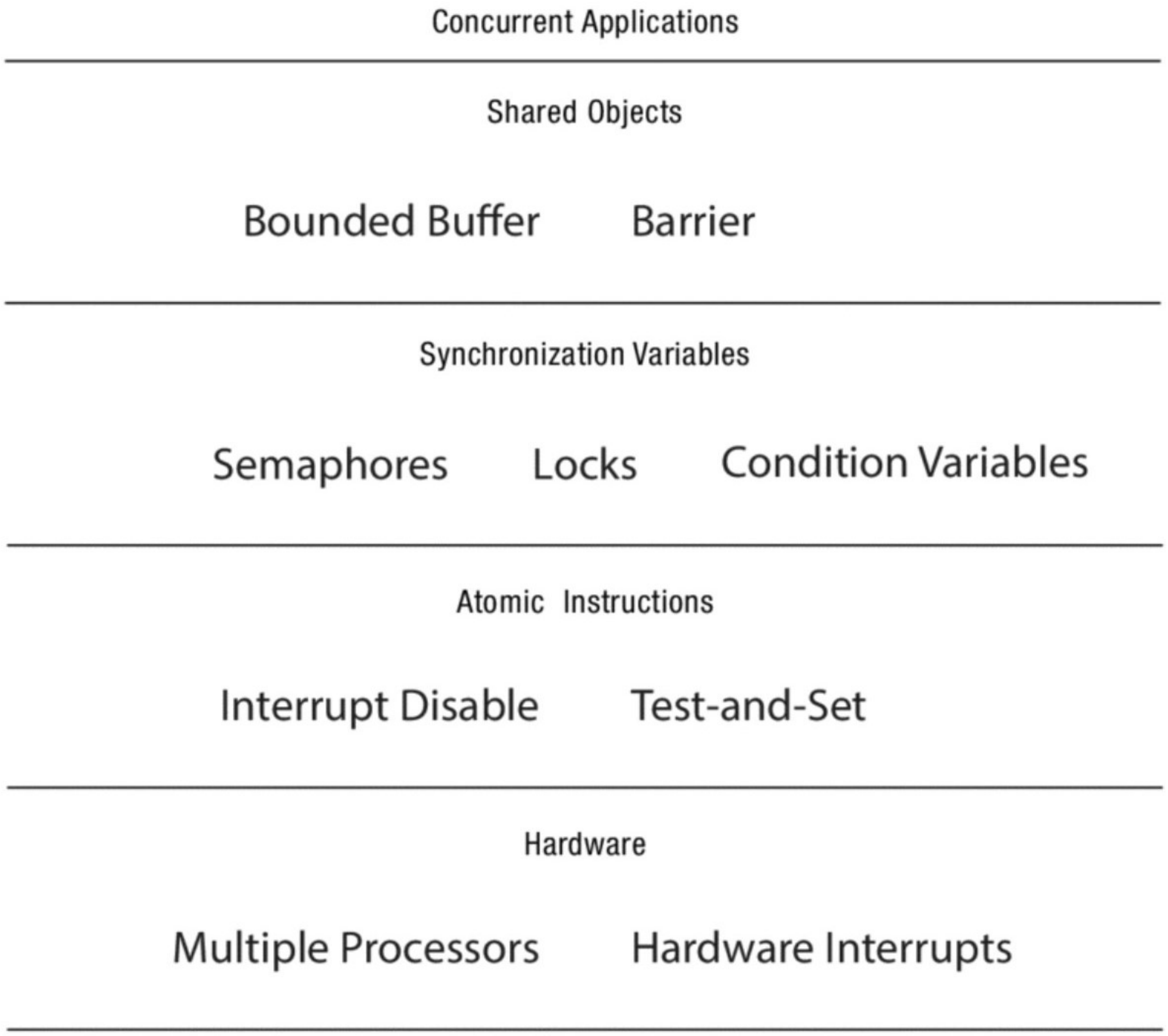
Monitors


Concurrent Applications
Shared Objects
Bounded Buffer
Barrier
Synchronization Variables
Semaphores
Locks
Condition Variables
Atomic Instructions
Interrupt Disable
Test-and-Set
Hardware
Multiple Processors
Hardware Interrupts

Monitors

- Hoare (1974) and Brinch Hansen (1973) proposed a high-level synchronization primitive to make it easier to write correct programs involving data sharing among processes
- **Monitor:** an Abstract Data Type (ADT) for handling/defining shared resources (aka critical sections)
- A Monitor Comprises:
  - Shared Private Data
    - The resource
    - Cannot be accessed from outside
  - Procedures that operate on the data
    - Gateway to the resource
    - Can only act on data local to the monitor
  - Synchronization primitives
    - Among processes/threads that access the procedures

Monitor Semantics

- Definition: A built-in (implicit) lock and zero or more condition variables for managing concurrent access to shared data
  - Use of Monitors is a programming paradigm
  - Some languages like Java/C# provide monitors in the language
- Monitors guarantee mutual exclusion
  - Only one thread can execute one of monitor procedures at any time
    - "*in the monitor*"
  - If second thread invokes monitor procedure at that time
    - It will block and wait for entry to the monitor
      - $\Rightarrow$ Need for a wait queue like in Semaphores
  - If thread within a monitor blocks, another can enter!
- Effect on parallelism?

Structure of a Monitor

```
Monitor monitor_name
{
    // shared variable declarations

    procedure P1(. . . .) {
        . . . .
    }

    procedure P2(. . . .) {
        . . . .
    }
    .
    .
    procedure PN(. . . .) {
        . . . .
    }

    initialization_code(. . . .) {
        . . . .
    }
}
```

For example:

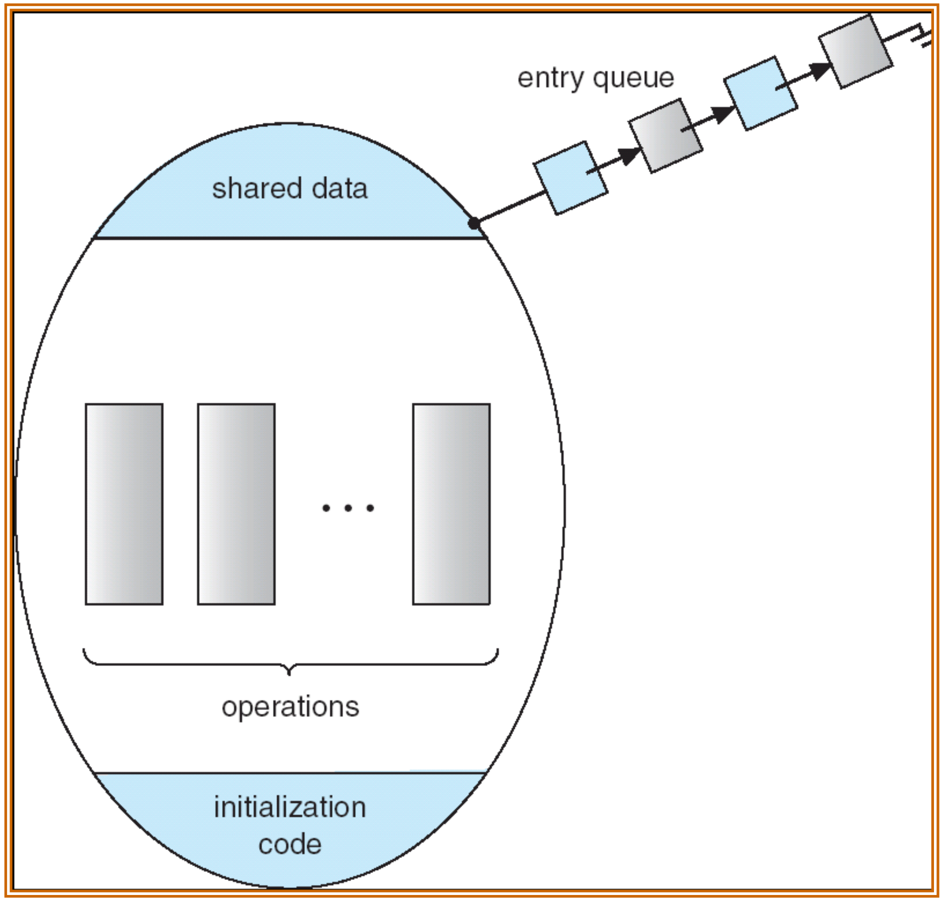
```
Monitor stack
{
    int top;
    void push(any_t *) {
        . . . .
    }

    any_t * pop() {
        . . . .
    }

    initialization_code() {
        . . . .
    }
}
```

only one instance of stack element can be modified at a time

Schematic view of a Monitor

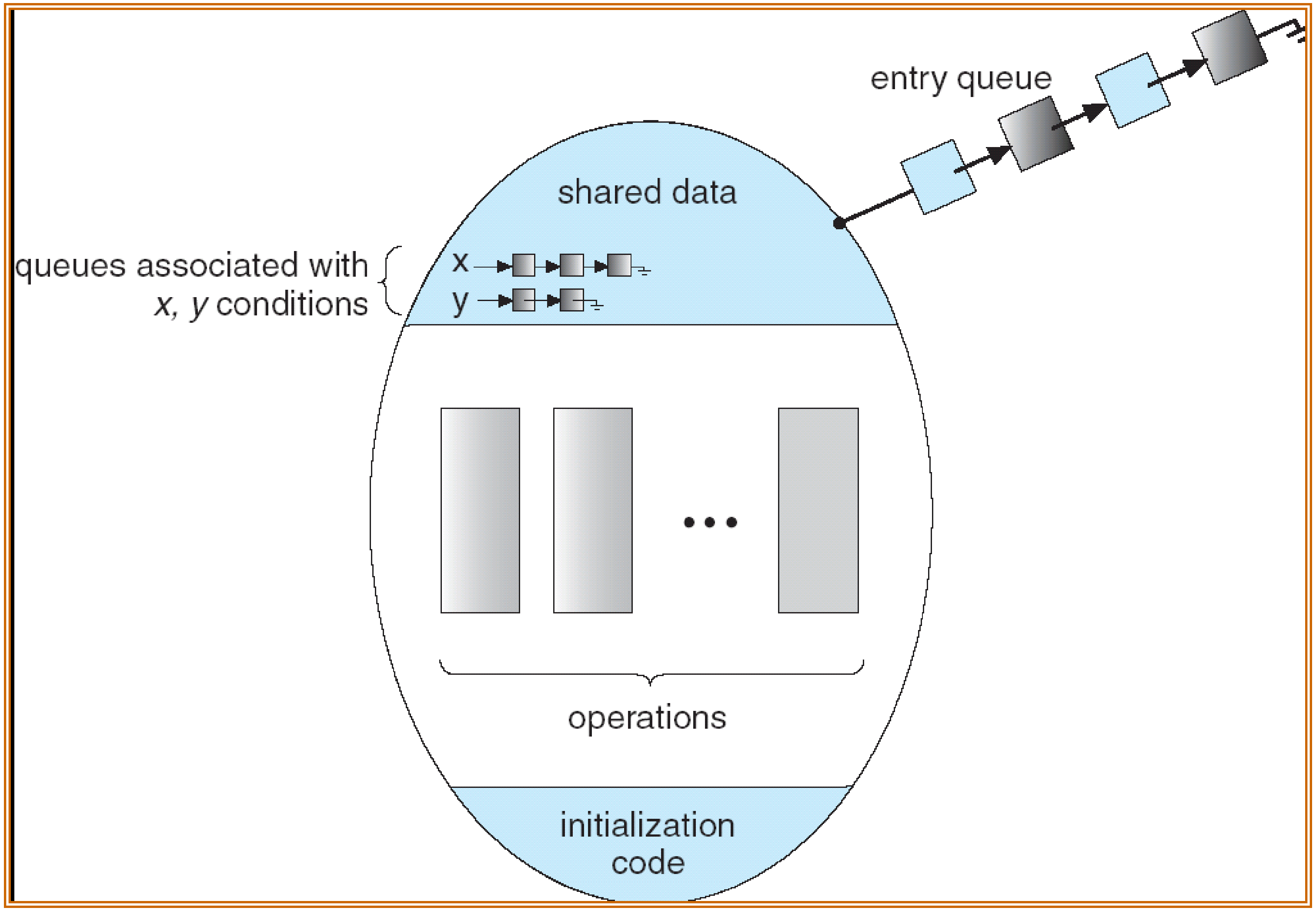


Synchronization Using Monitors

- Defines Condition Variables:
  - Provides a mechanism to wait for events within a monitor
    - Resources available, any writers
  - *condition* x, y; //Not variables at all. Event Queues inside Monitor
- 3 atomic operations on *Condition Variables*
  - x.wait(): releases monitor lock atomically, sleeps/blocks on x's waiting queue until woken up by x.signal()

    ⇒ condition variables also have waiting queues too!
  - x.signal(): wakes one process waiting on condition (if there is one)
    - Signalled process is kept in ready queue of Scheduler and so it may not run immediately
    - If no process is sleeping, then the **signal** operation has no effect
    - No history associated with signal unlike Semaphores V()
  - x.broadcast(): wakes all processes waiting/sleeping on condition
    - Useful for resource manager
- Condition variables are memoryless
  - If(x) then { } does not make sense

Monitor with Condition Variables



Need not to be FIFO queues

Simple Producer Consumer using Monitors

- **Here is an (infinite) synchronized queue**

```
Shared resource: Queue queue;

Producer(item) {
   queue.enqueue(item);   // Add item
}

Consumer() {
   item = queue.dequeue();// Get next item or null
   return(item);          // Might return null
}
```

- **Not very interesting use of “Monitor”**
  - **It only uses a built-in lock with no condition variables**
  - **Cannot put consumer to sleep if no work!**

Better Solution?

- **How do we change the consumer() routine to wait until something is on the queue?**
  - **Could do this by keeping a count of the number of things on the queue (with semaphores), but error prone**

- **Condition Variable: a queue of threads waiting for something *inside* a critical section**
  - **Key idea: allow sleeping inside critical section by atomically releasing lock at time we go to sleep**
  - **Contrast to semaphores which can't wait inside critical section**

Consumer

```
char get() {

    P(full);
    P(mutex);

    // remove ch from
buffer
    ch = buf[tail%N];
    tail++;

    V(mutex);
    V(empty);

     return ch;
}
```

Better Solution

- **Here is an (infinite) synchronized queue**

```
Condition dataready; //cond variable
Queue queue; //shared resource

producer(item) {
   queue.enqueue(item);      // Add item
   dataready.signal();       // Signal any waiters
}

consumer() {
   while (queue.isEmpty()) {
   dataready.wait(); //If nothing, sleep in monitor
   }
   item = queue.dequeue();  // Get next item
   return(item);  // no chance of returning null
}
```

Mesa vs Hoare monitors

- **Need to be careful about precise definition of signal and wait. Consider a piece of our dequeue code:**

```
while (queue.isEmpty()) {
  dataready.wait(); // If nothing, sleep
}
item = queue.dequeue();// Get next item
```

  - **Why didn't we do this?**

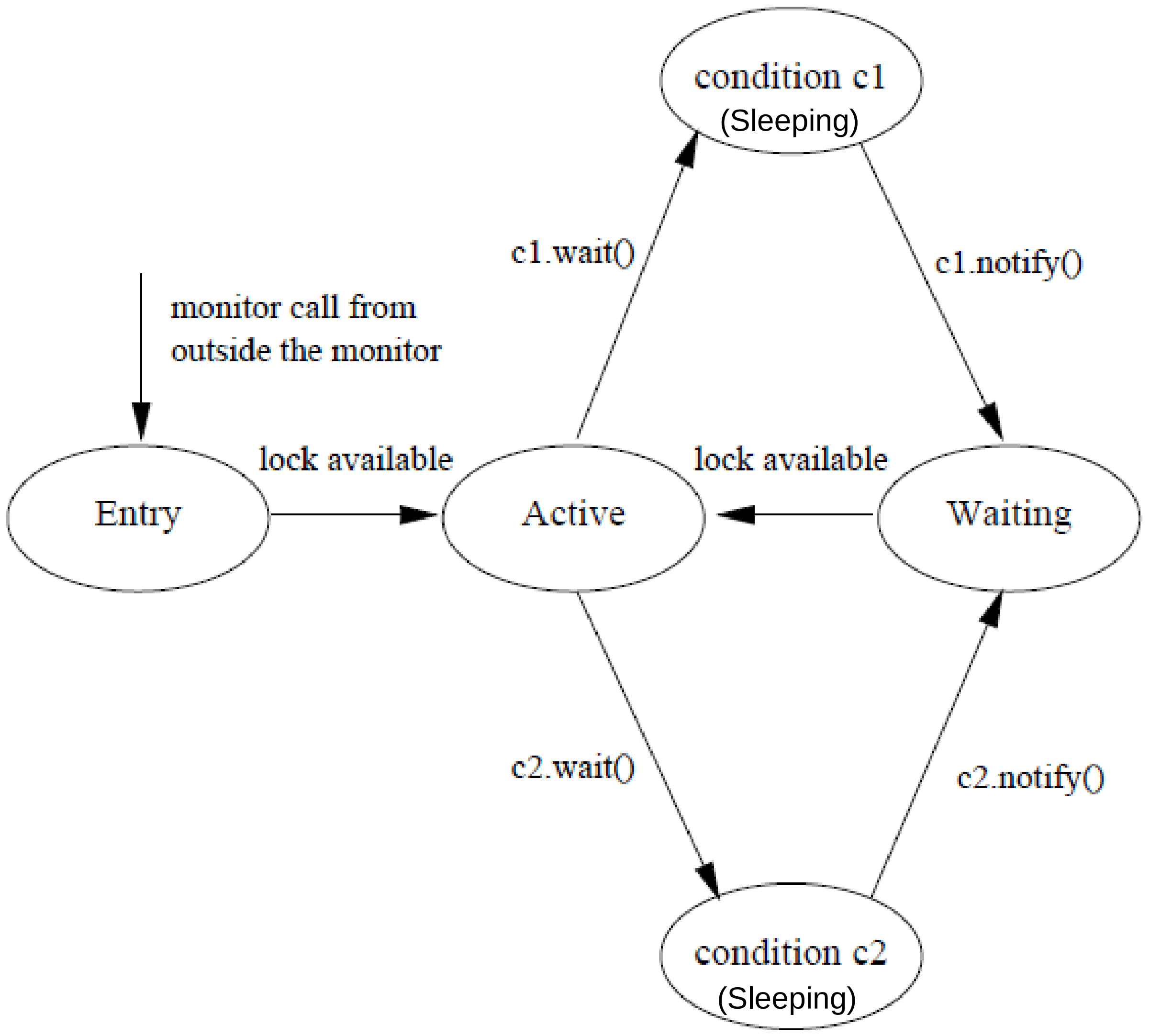
```
if (queue.isEmpty()) {
  dataready.wait(); // If nothing, sleep
}
item = queue.dequeue();// Get next item
```

- **Answer: depends on the type of scheduling**
  - **Hoare-style (most textbooks): (Signal-and-Wait)**
    - » **Signaler gives lock, CPU to waiter; the waiter runs immediately**
    - » **Waiter gives up lock, CPU back to signaler when waiter exits critical section or if it waits again**
  - **Mesa-style (most real OSs):Signal-and-Continue**
    - » **Signaler keeps lock and CPU**
    - » **Waiter placed on ready queue of OS Scheduler with no special priority**
    - » **Practically, need to check condition again after wait. So, Wait must always be called from within a loop.**

Types of wait queues

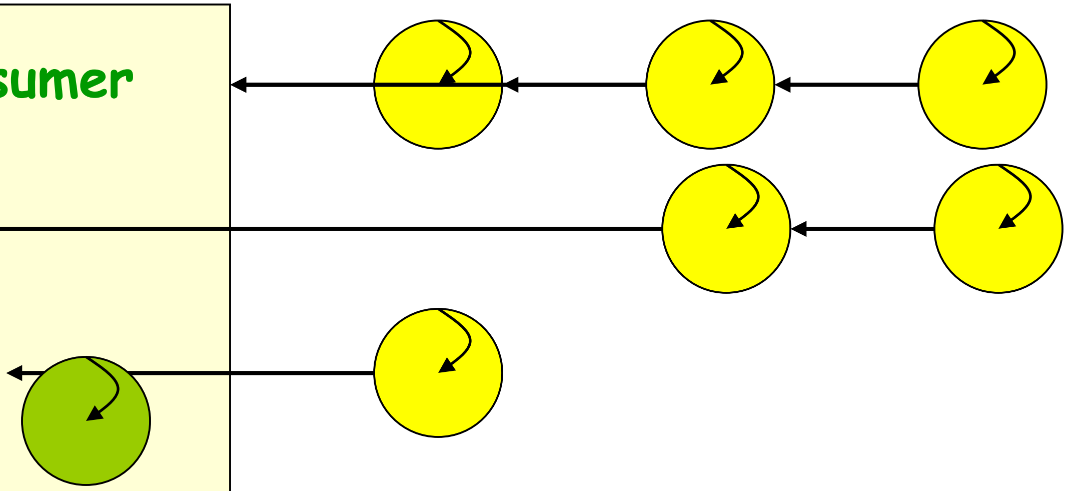
- Monitors have several kinds of “wait” queues!
  - Condition variable: has a queue of threads waiting on the associated condition
    - Thread is added to the end of the queue
    - One queue per conditional variable
  - Entry to the monitor: has a queue of threads waiting to obtain mutual exclusion so they can enter
    - One queue per Monitor
    - Again, a new arrival goes to the end of the queue
  - So-called “urgent waiting” queue: threads that were just woken up using signal().
    - One per Monitor
    - Newly signaled thread normally goes to the *front* of this queue
  - Signaller queue: threads that were just performed signal/notify are kept in this queue
    - One per Monitor

State Diagram



Producer Consumer using Monitors

```
Monitor Producer_Consumer
{
    condition full;
    /* other vars */
    condition empty;
    void put(char ch) {
    if all_full full.wait();
            . . .
  if one_full empty.signal();
    }
    char get()  {
            . . .
    }
}
```



Case1: Buffer is full

Case2: Buffer is empty & Consumer wants to get

Case3: Produer puts and signals waiting consumer

Mesa-style monitor subtleties

```
char buf[N];
int n = 0, tail = 0, head = 0;
condition empty, full;
void producer(char ch)
        if(n == N)
                full.wait();
        buf[head%N] = ch;
        head++;
        n++;
     if(n==1) empty.signal();

char consumer()
        if(n == 0)
          empty.wait();
        ch = buf[tail%N];
        tail++;
        n--;
        if(n==N-1) full.signal();
        return ch;
```

```
// Bounded producer/consumer with monitors
```

Consider the following time line:

0. initial condition: n = 0
1. c0 tries to take char, blocks on empty (releasing monitor lock)
2. p0 puts a char (n = 1), signals C0 waiting on empty
3. c0 is put on run queue
4. Before c0 runs, another consumer thread c1 enters and takes character (n = 0)
5. c0 runs.

Possible fixes?

Mesa-style monitor subtleties

```
char buf[N];                          // Bounded producer/consumer with monitors
int n = 0, tail = 0, head = 0;
condition empty, full;
void producer(char ch)
        if(n == N)
                full.wait();
        buf[head%N] = ch;
        head++;
        n++;
     if(n==1) empty.signal();

char consumer()
        while(n == 0)
          empty.wait();
        ch = buf[tail%N];
        tail++;
        n--;
        if(n==N-1) full.signal();
        return ch;
```

When can we replace “while” with “if” in Consumer?

Do we need to do same even for Producer?

Dining Philosophers Solution using Monitors?

Dining Philosophers Sol using Monitors

```
monitor dp
{
  enum {thinking, hungry, eating} state[5];
  condition self[5];
  void pickup(int i)              // following slides
  void putdown(int i)             // following slides
  void test(int i)                // following slides
  void init() {
       for (int i = 0; i < 5; i++)
                 state[i] = thinking;
  }
}
```

Dining Philosophers Example

```
void pickup(int i) {
        state[i] = hungry;
        test[i];
        if (state[i] != eating)
                self[i].wait();
}

void putdown(int i) {
        state[i] = thinking;
        // test left and right neighbors
        test((i+4) % 5);
        test((i+1) % 5);
}
```

Dining Philosophers Example

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

- Each philosopher *i* invokes the operations **`pickup()`** and **`putdown()`** in the following sequence:

```
        DiningPhilosophers.pickup(i);
              EAT
        DiningPhilosophers.putdown(i);
```

- No deadlock, but starvation is possible
- It shows starvation-free solution and Java simulator: http://dx.doi.org/10.1145/366413.364612

Condition Variables & Semaphores

- Condition Variables != semaphores
- Access to monitor is controlled by an implicit lock
  - Wait: blocks on thread and gives up the lock
    - To call wait, thread has to be in monitor, hence the lock
    - Semaphore P() blocks thread only if its S.Val <=0
  - Signal: causes waiting thread to wake up
    - If there is no waiting thread, the signal is lost
    - V() increments S.Val, so future threads need not wait on P()
    - Condition variables have no history
- However they can be used to implement each other

Hoare Monitors using Semaphores

```
semaphore mutex;   Set to 1

semaphore next; Set to 0

int next_count = 0;
```

Each procedure F is replaced by:

```
P(mutex);

/* body of F */

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

```
semaphore x_sem;(set to 0)
int x_count = 0;
```

Condition Var Wait: x.wait:

```
x_count++;
if(next_count > 0)
    V(next);
else
    V(mutex);
P(x_sem);
x.count--;
```

Condition Var Notify: x.notify:

```
If(x_count > 0) {
    next_count++;
    V(x_sem);
    P(next);
    next_count--;
}
```

Language Support

- Can be embedded in programming language:
  - Synchronization code added by compiler, enforced at runtime
  - Mesa/Cedar from Xerox PARC
  - Java: **synchronized, wait, notify, notifyall**
  - C++: **std::mutex, std::atomic types, std::condition_variable, Tasks (vs Threads)**
  - C#: **lock, wait (with timeouts), pulse, pulseall**
- Monitors easier and safer than semaphores
  - Compiler can check, lock implicit (cannot be forgotten)
- But not available in all languages
  - C does not have monitors/semaphores
    - glibc provides semaphores with underlying OS support
- Monitors and Semaphores require to have shared resource and then offer mutual exclusion
  - Distributed systems with each CPU with private memory?
    - Message passing

Java Language Support for Synchronization

- Java has explicit support for threads and thread synchronization
- It offer two basic synchronization idioms: *synchronized methods* and *synchronized statements*.

  - To make a method synchronized, simply add the synchronized keyword to its declaration.
  - Two or more invocations of synchronized methods on to the same object do not interleave
  - Do not synchronize constructors!

```
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

Java Language Support for Synchronization

- Every object has an associated intrinsic lock (aka monitor lock) which gets automatically acquired and released on entry and exit from a *synchronized* method.
- When a thread invokes a synchronized method, it automatically acquires the intrinsic lock for that method's object and releases it when the method returns.
- The lock release occurs even if the return was caused by an uncaught exception.
- Static synchronized methods get lock of the Class object associated with the class
- Bank Account example:

```
class Account {
   private int balance;
   // object constructor
   public Account (int initialBalance) {
      balance = initialBalance;
   }
   public synchronized int getBalance() {
      return balance;
   }
   public synchronized void deposit(int amount) {
      balance += amount;
   }
}
```

Java Language Support

- Java also has *synchronized* statements:

```
synchronized (object) {
     S1;
     S2;
     …
}
```

  - Since every Java object has an associated implicit lock, this type of statement acquires and releases the object’s lock on entry and exit of the statement block
  - Works properly even with exceptions:

```
synchronized (this) {
   …
   DoFoo();
   …
}
void DoFoo() {
   throw errException;
}
```

Example1 with Sync Methods

```
public class MyClass {
    private long c1 = 0;
    private long c2 = 0;

    public synchronized void inc1() {
            c1++;
    }

    public synchronized void inc2() {
            c2++;
    }
}
```

Updates to C1 and C2 can’t be interleaved.

Example2 with Sync Stmts

Updates to C1 and C2 can be interleaved. Use this idiom with extreme care. You must be absolutely sure that it really is safe to interleave access of the affected fields.

```
public class MyClass {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1() {
        synchronized(lock1) {
            c1++;
        }
    }

    public void inc2() {
        synchronized(lock2) {
            c2++;
        }
    }
}
```

Java Language Support

- In addition to an implicit lock, every object has an implicit single condition variable associated with it with operations
  - wait(): thread releases the lock and suspends execution till some other thread calls notify() or notifyall()
    - To acquire implicit lock on object, you need to call wait() inside its synchronized method or synchronized statement
    - Notification does not necessarily mean this thread can continue, so re-check the condition on which wait() was invoked (i.e., while loop like in Signal-and-Continue)
  - notify(): wakes up the oldest waiter
  - notifyall(): like broadcast, wakes up everyone
- How to wait inside a synchronization method or block:
    - `void wait(long timeout); // Wait for timeout`
    - `void wait(long timeout, int nanoseconds); //variant`
    - `void wait();`
- How to signal in a synchronized method or block:
    - `void notify();    // wakes up oldest waiter`
    - `void notifyAll(); // like broadcast, wakes everyone`

Java Language Support

- Condition variables can wait for a bounded length of time. This is useful for handling exception cases:

```
t1 = time.now();
while (!ATMRequest()) {
   wait (CHECKPERIOD);
   t2 = time.new();
   if (t2 – t1 > LONG_TIME) checkMachine();
}
```

- Not all Java VMs equivalent!
    - Different scheduling policies, not necessarily preemptive!

Producer-Consumer Problem Solution using Monitors in Java?

Sol. to Producer-Consumer Problem in Java

```
public class Drop {
    // Message sent from P to C
    private String message;
    // True if consumer should wait
    // for producer to send message,
    // false if producer should wait for
    // consumer to retrieve message.
    private boolean empty = true;

    public synchronized String take() {
        // Wait until message is
        // available.
        while (empty) {
            try {
                wait();
     } catch (InterruptedException e) {}
        }
        // Toggle status.
        empty = true;
        // Notify producer that
        // status has changed.
        notifyAll();
        return message;
    }
```

```
public synchronized void put(String
message) {
        // Wait until message has
        // been retrieved.
        while (!empty) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        // Toggle status.
        empty = false;
        // Store message.
        this.message = message;
        // Notify consumer that status
        // has changed.
        notifyAll();
    }
}
```

Refer this link for complete code: http://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html

High Level Concurrency in Java

Useful for massively concurrent applications that fully exploit today's multiprocessor and multi-core systems.

•Lock objects support locking idioms that simplify many concurrent applications.
•Executors define a high-level API for launching and managing threads.
•Concurrent collections make it easier to manage large collections of data, and can greatly reduce the need for synchronization.
•Atomic variables have features that minimize synchronization and help avoid memory consistency errors.
•ThreadLocalRandom (in JDK 7) provides efficient generation of pseudorandom numbers fro multiple threads.

http://docs.oracle.com/javase/tutorial/essential/concurrency/highlevel.html

Structured Synchronization

- Identify objects or data structures that can be accessed by multiple threads concurrently
  - In OS kernel, everything!
- Add locks to object/module
  - Grab lock on start to every method/procedure
  - Release lock on finish
- If need to wait
  - while(needToWait()) { condition.Wait(lock); }
  - Do not assume when you wake up, signaller just ran
- If do something that might wake someone up
  - Signal or Broadcast
- Always leave shared state variables in a consistent state
  - When lock is released, or when waiting

Remember these rules

1. Use consistent structure
2. Always use locks and condition variables
3. Always acquire lock at beginning of procedure/method, release at end
4. Always hold lock when using a condition variable
5. Always wait in while loop!
6. Never spin in sleep()

Conclusions…

- Race conditions are a pain!
- We studied five ways to handle them
  - Each has its own pros and cons
- Support in Java, C++/C# has simplified writing multithreaded applications
- Some new program analysis tools automate checking to make sure your code is using synchronization correctly
  - The hard part for these is to figure out what “correct” means!

Reading Assignment

- Chapter 5 from OSC by Galvin et al
- Chapter 2 from MOS by Tanenbaum et al
- **Starving philosophers: expt with monitor synchronization:** http://dx.doi.org/10.1145/366413.364612
- **Java tutorial on synch:**

http://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html

- **C++ tutorial on synch:**
  - https://www.modernescpp.com/index.php/c-core-guidelines-rules-for-concurrency-and-parallelism
  - https://www.codeproject.com/Articles/1278737/Programming-Concurrency-in-Cplusplus-Part-2