

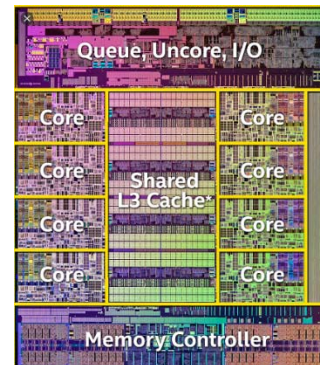
CSE216 Programming Abstractions

Concurrency

YoungMin Kwon

Concurrent Programming

- A program is concurrent
 - If it has more than one active execution context (thread of control)
- Why concurrent programming?
 - <https://www.youtube.com/watch?v=MNhubpzhs-Q>

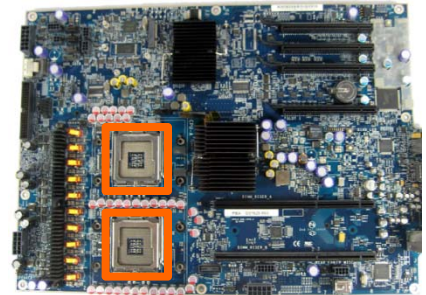


Concurrent Programming

- To **capture the logical structure** of a program
 - Many programs need to keep track of largely independent tasks at the same time
 - Represent each task with a separate thread
- Exploit **parallel hardware** for speed
- Physical **distribution**
 - Applications run across the Internet
 - Automobile: applications running on dozens of processors spread throughout the vehicle

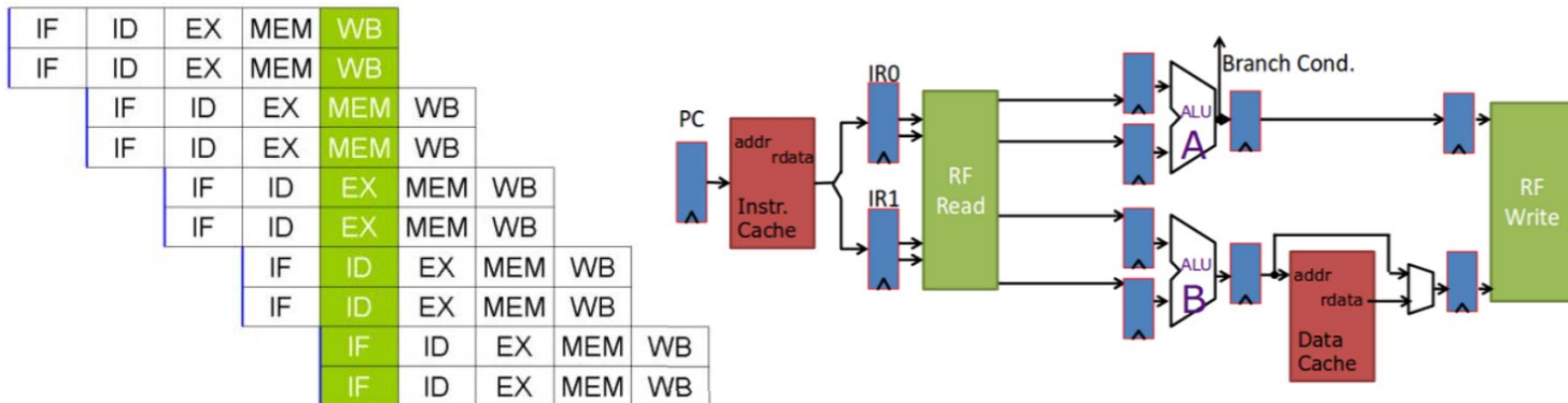
Concurrent Programming

- **Concurrent** system: two or more tasks may be underway at the same time
- **Parallel** system: a concurrent system with more than one task can be physically active at once
- **Distributed** system: a parallel system with physically separated devices



Levels of Parallelism

- Instruction Level Parallelism (ILP)
 - Superscalar pipelines
 - Aggressive speculation
 - However, a limit seems to be reached...



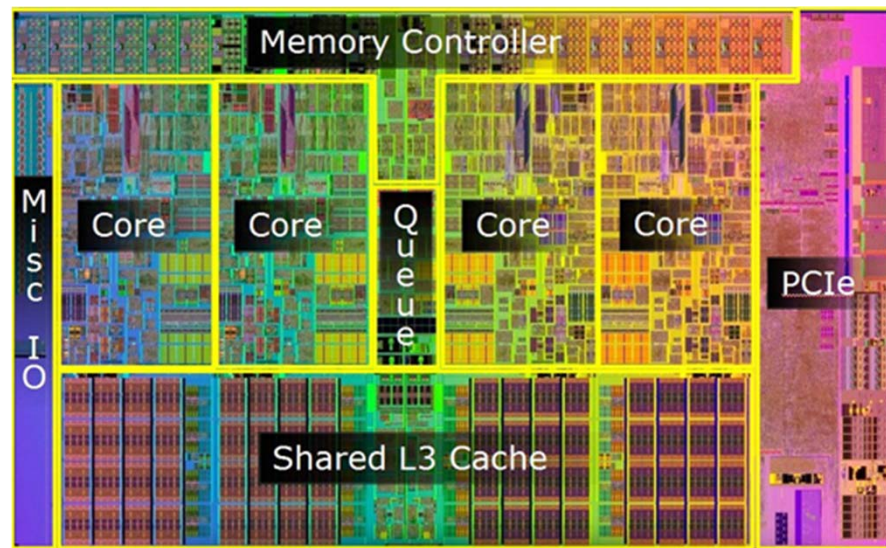
Levels of Parallelism

- Vector Parallelism
 - Perform operations repeatedly on every element of a very large data set
 - Supercomputers
 - GPU programming



Levels of Parallelism

- General purpose computing
 - Multicore processors
 - Coarser-grain thread-level parallelism



Levels of Abstraction

- **Black box** parallel libraries
 - Parallel algorithms: sorting routine, linear algebra package, ...
 - Caller does not know the implementation
- **Mutually independent** tasks (less abstract)
 - In C# task parallel library

```
Parallel.For(0, 100, i => { A[i] = foo(A[i]) });
```

Levels of Abstraction

- Tasks are not independent
 - Explicitly **synchronize** their interactions to eliminate races
- Example: synchronization error

Thread 1	Thread 2
<code>a := count;</code>	<code>...</code>
<code>a := a + 1;</code>	<code>a := count;</code>
<code>count := a;</code>	<code>a := a + 1;</code>
<code>...</code>	<code>count := a;</code>

`count` is a shared var, `a` is a local var

Levels of Abstraction

- Race condition
 - Two or more threads are racing to touch a common object
 - System behavior depends on which one gets there first
- Critical section
 - A section of a code that that should be accessed **mutually exclusively**

Process vs Thread vs Task

- Processes

- A virtualized computer
- Has its own address space, open files, process state, ...

- Threads

- A virtualized CPU
- Has its own set of registers and stack
- Shares the other resources with other threads

- Tasks

- Well-defined unit of work that must be performed by some threads

Threads: Two Issues (1/2)

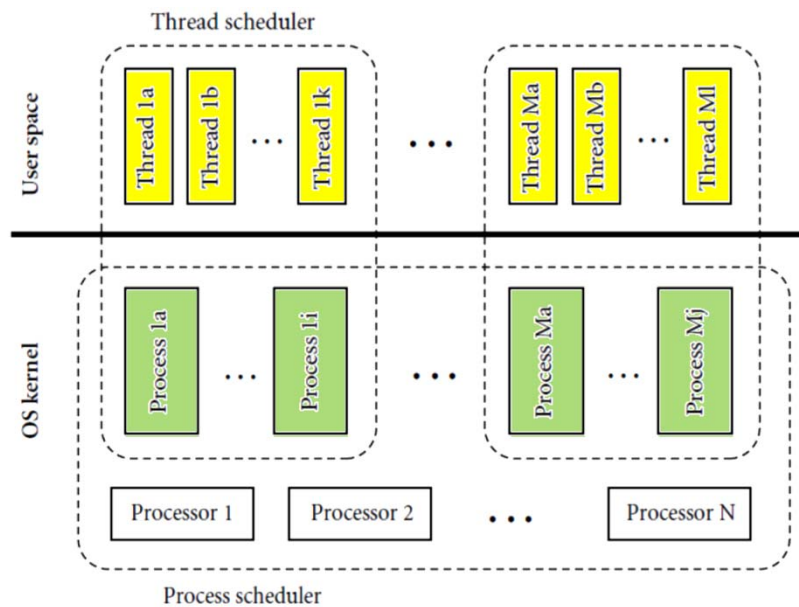
- Communication
 - A mechanism that allows one thread to obtain information produced by another
 - Shared memory
 - Message passing

Threads: Two Issues (2/2)

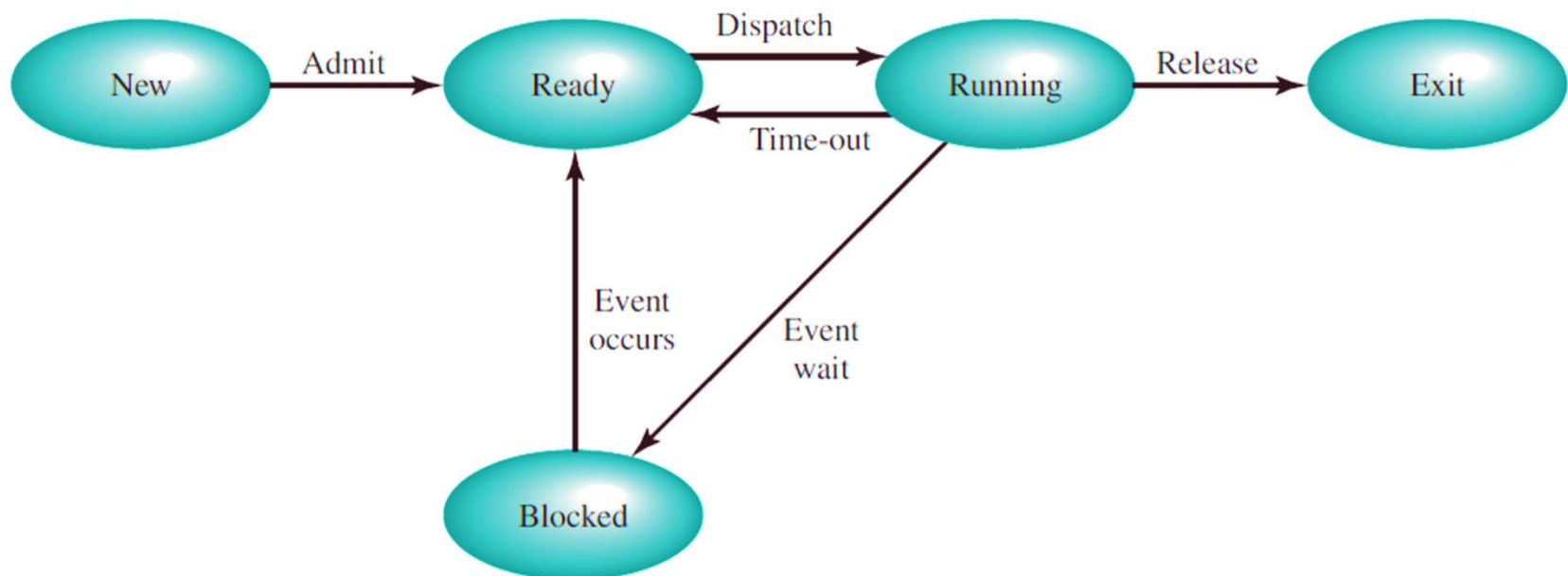
- Synchronization
 - A mechanism that allows a programmer to control the order of operations in different threads
 - Spinning (busy waiting)
 - Blocking (scheduler based)

Thread Implementation

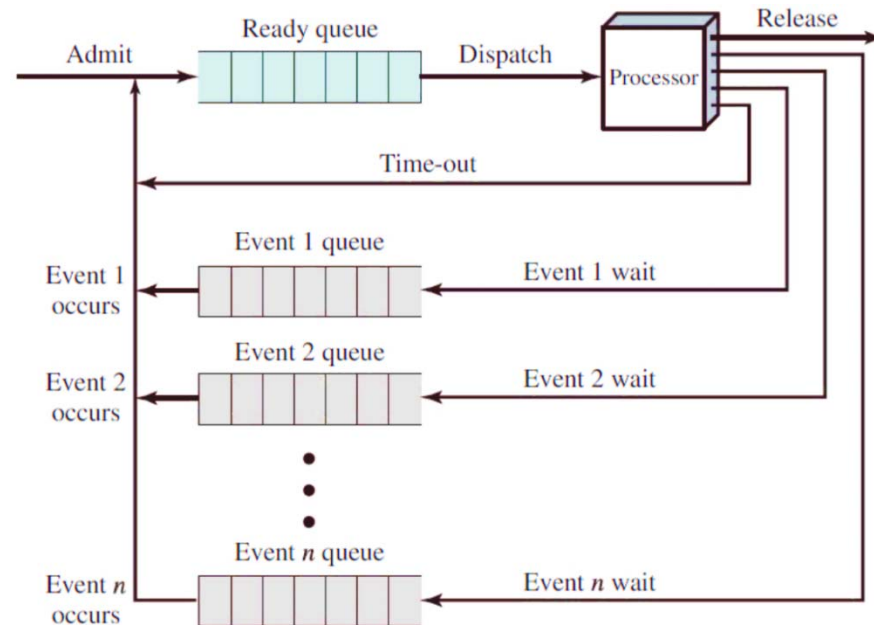
- Thread implementation
 - One extreme: separate process for each thread
 - Another extreme: put all threads of a program in a single process
 - Intermediate approaches



A State Model of a Process



A State Model of a Process



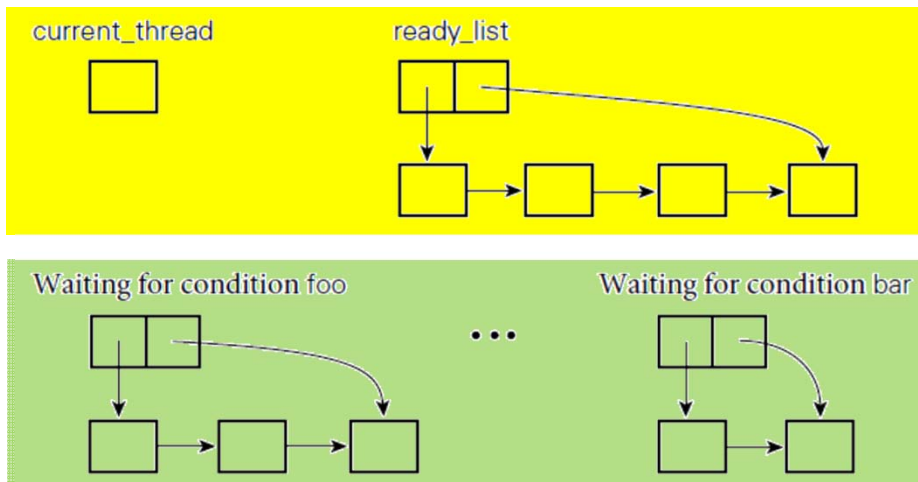
- Multiple blocked queues
 - Single blocked queue: OS has to scan the blocked queue for every event
 - The processes in a certain event queue are moved to the Ready queue

Thread Implementation

- Separate process for each thread
 - Too expensive
 - Performing thread-related operations requires a **system call**
- Single process to host all threads
 - Precludes **parallel execution** on a multicore or multiprocessor machine
 - If a thread is making a **blocking system** call, all threads in the process are blocked

Simple Thread Scheduler

- Data structure
 - At any time a thread is either **blocked** or **runnable**



procedure reschedule

```
t : thread := dequeue(ready_list)
transfer(t)
```

procedure yield

```
enqueue(ready_list, current_thread)
reschedule
```

procedure sleep_on(ref Q : queue of thread)

```
enqueue(Q, current_thread)
reschedule
```

Java Threads

- Two options to create threads
 - Extend **Thread** class
 - Implement **Runnable** interface
 - Note: Thread class implements Runnable

```
class PrimeThread extends Thread {  
    long minPrime;  
    PrimeThread(long minPrime) {  
        this.minPrime = minPrime;  
    }  
  
    public void run() {  
        // compute primes larger than minPrime  
        . . .  
    }  
}
```

```
PrimeThread p = new PrimeThread(143);  
p.start();
```

```
class PrimeRun implements Runnable {  
    long minPrime;  
    PrimeRun(long minPrime) {  
        this.minPrime = minPrime;  
    }  
  
    public void run() {  
        // compute primes larger than minPrime  
        . . .  
    }  
}
```

```
PrimeRun p = new PrimeRun(143);  
new Thread(p).start();
```

Java Threads

```
public static class ImplAsThread extends Thread {  
    private String msg;  
    public ImplAsThread(String msg) {  
        this.msg = msg;  
    }  
    public void run() {  
        try {  
            for(int i = 0; i < 10; i++) {  
                System.out.print(msg + " ");  
                Thread.sleep(10);  
            }  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

extend Thread class

override run method
run() will run
a new thread

...

...

```
public static void test() {  
    try {  
        System.out.println("Testing ImplAsThread...");  
  
        ImplAsThread a = new ImplAsThread("A");  
        ImplAsThread b = new ImplAsThread("B");  
  
        a.start();  
        b.start();  
  
        a.join();  
        b.join();  
  
        System.out.println("\nDone");  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```

a.run and b.run will run
on their own threads

main thread will wait
until a and b terminate

Testing ImplAsThread...

A B B A A B B A B A A B B A A B B A

Done

Java Thread

```
public static class ImplAsRunnable {  
    public static void foo(String msg) {  
        try {  
            for(int i = 0; i < 10; i++) {  
                System.out.print(msg + " ");  
                Thread.sleep(10);  
            }  
        } catch(Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
public static class Foo implements Runnable {  
    public void run() {  
        foo("A");  
    }  
}
```

Implement run
method

Foo implements
Runnable interface

```

public static void test() {
    try {
        System.out.println("Testing ImplAsRunnable...");
        Thread a = new Thread(new Foo());
        Thread b = new Thread(() -> foo("B"));

        a.start();
        b.start();

        a.join();
        b.join();

        System.out.println("\nDone");
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

a.run and b.run will run
on their own threads

a.start();
b.start();

Runnable using
lambda

main thread will wait
until a and b terminate

a.join();
b.join();

```

Testing ImplAsRunnable...
A B B A A B B A A B B A A B B A A B A B
Done

```

Synchronization

- **Busy-wait** synchronization
 - A thread reads a variable **X** in a loop, waiting for it to be a value **Y**
 - Spin lock: provides a **mutual exclusion**
 - Only the thread holding a lock can enter a **critical section**
 - Barriers: no thread continues past a given point until **all threads have reached** that point

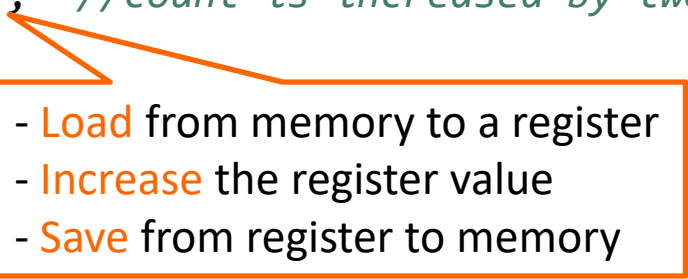
Java Synchronization (Erroneous)

- Counter example

- Two threads increase a **shared** variable *count*
- **Race** condition exists between two threads

```
public class Counter {  
    public static class Unprotected {  
        static int count; //shared variable  
  
        public static void inc() {  
            for(int i = 0; i < 1000000; i++)  
                count++; //count is increased by two threads  
        }  
    }  
}
```

...

- 
- **Load** from memory to a register
 - **Increase** the register value
 - **Save** from register to memory

```

public static void test() {
    try {
        System.out.println("Testing Unprotected...");

        count = 0;

        Thread a = new Thread(() -> inc());
        Thread b = new Thread(() -> inc());

        a.start();
        b.start();
        a.join();
        b.join();

        System.out.println("count: " + count);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

Testing Unprotected...

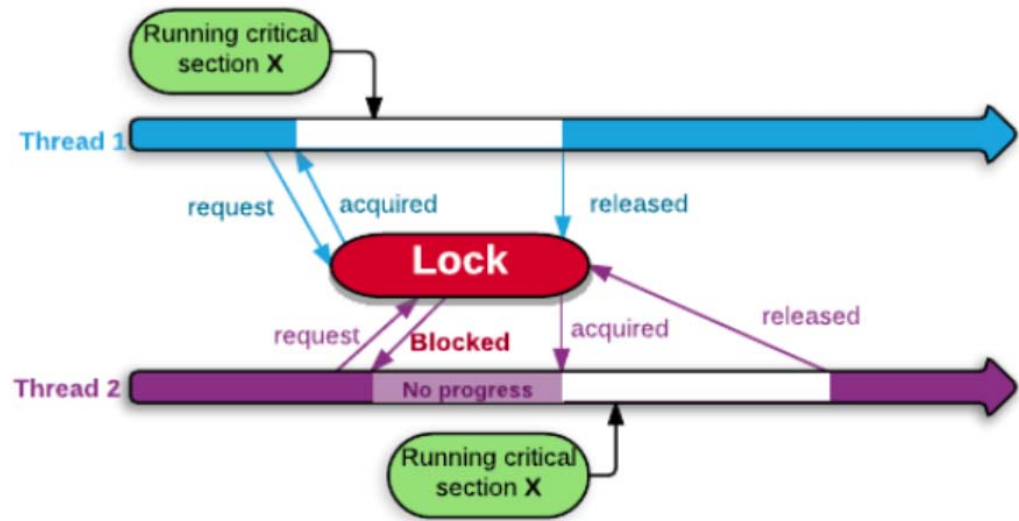
count: 1887043

Mutual Exclusion

- Mutual exclusion by lock

Thread 1	Thread 2
<pre>acquire(L); a := count; a := a + 1; count := a; release(L); ...</pre>	<pre>... acquire(L); a := count; a := a + 1; count := a; release(L); ...</pre>

`count` is a shared var, `a` is a local var



Synchronization: Spin Lock

```
type lock = Boolean := false;
```

```
procedure acquire_lock(ref L : lock)
```

```
  while not test_and_set(L)
```

```
    while L
```

```
      -- nothing -- spin
```

```
procedure release_lock(ref L : lock)
```

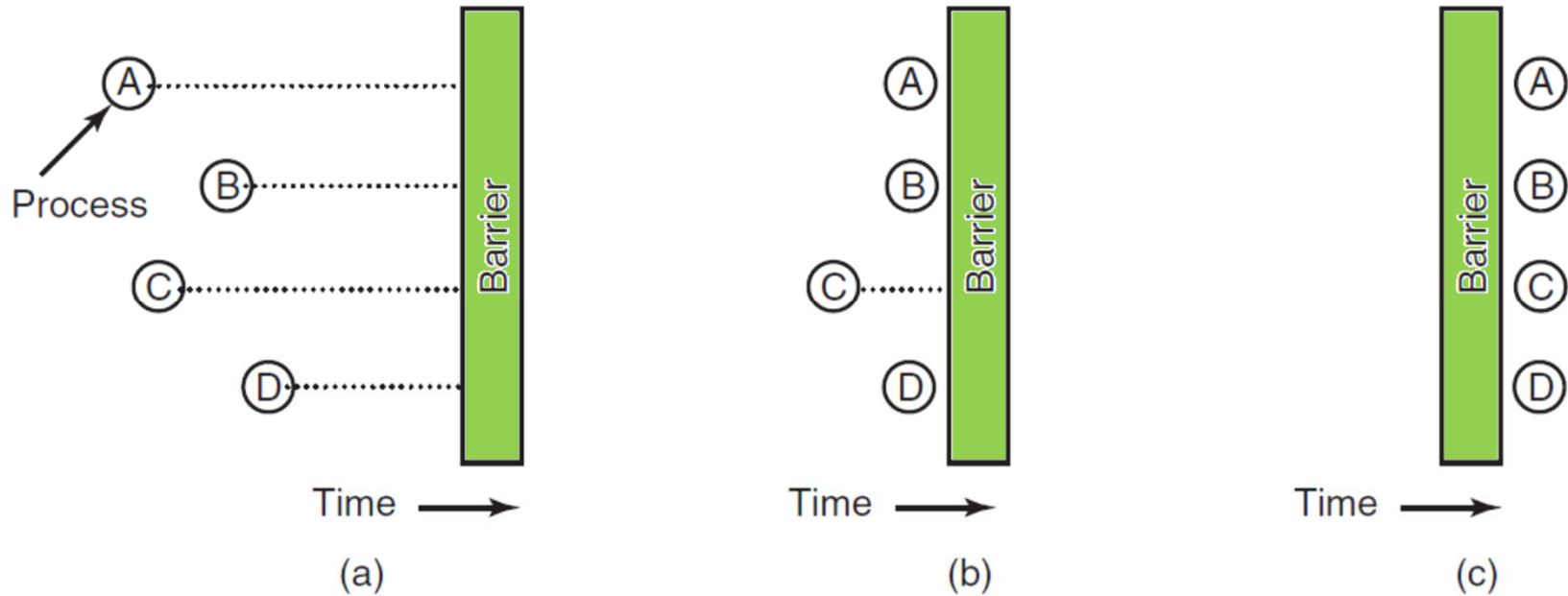
```
  L := false
```

In a multi core system,
test_and_set(L) is expensive
compared to reading L

■ Atomic operations

- Operations without context switches in the middle
- **test_and_set**: sets a Boolean variable true and returns whether it was previously false
- **compare_and_swap**: if a variable has an expected value, change it to the new value

Barriers



- a) Threads approaching a barrier
- b) All threads but one blocked at the barrier
- c) When the last thread arrives the barrier, all of them are let through

Synchronization: Barrier

- Every thread should complete their previous step before any moves to the next
 - Used in data-parallel algorithms, structured as phases

```
shared count : integer := n
shared sense : Boolean := true
per-thread private local_sense : Boolean := true
procedure central_barrier
  local_sense := not local_sense
  -- each thread toggles its own sense
  if fetch_and_decrement(count) = 1
    -- last arriving thread
    count := n                -- reinitialize for next iteration
    sense := local_sense      -- allow other threads to proceed
  else
    repeat
      -- spin
    until sense = local_sense
```

Synchronization: Atomic Operation

- Non-blocking algorithms

`x := foo(x);`

x is a shared var;
need mutual-exclusion

`acquire(L)`

`r1 := x`

`r2 := foo(r1)`

`x := r2`

`release(L)`

Mutual-exclusion using
a lock

-- probably a multi-instruction sequence

`start:`

`r1 := x`

`r2 := foo(r1)`

`r2 := CAS(x, r1, r2)`

`if !r2 goto start`

Mutual-exclusion
without using a lock

-- probably a multi-instruction sequence

-- replace x if it hasn't changed

Java Synchronization (atomic opr)

- Counter example using atomic operations
 - Issue
 - Context switch occurred before a thread writes count
 - The other thread increases the count many times
 - When switched back, the first thread finishes the write
 - Fix
 - The issue can be fixed by increasing count **atomically**

```

public static class Atomic {
    static AtomicInteger count;
    public static void inc() {
        for(int i = 0; i < 1000000; i++)
            count.incrementAndGet();
    }

    public static void test() {
        try {
            System.out.println("Testing Atomic...");
            count = new AtomicInteger(0);
            Thread a = new Thread(() -> inc());
            Thread b = new Thread(() -> inc());
            a.start();
            b.start();
            a.join();
            b.join();
            System.out.println("count: " + count.get());
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}

```

Testing Atomic...
count: 2000000

Synchronization: Scheduler-based

- Scheduler-based synchronization
 - Switch to other thread

```
type lock = Boolean := false;

procedure acquire_lock(ref L : lock)
  while not test_and_set(L)
    count := TIMEOUT
    while L
      count -= 1
      if count = 0
        OS_yield      -- relinquish processor and drop priority
        count := TIMEOUT

procedure release_lock(ref L : lock)
  L := false
```

Synchronization: Semaphores

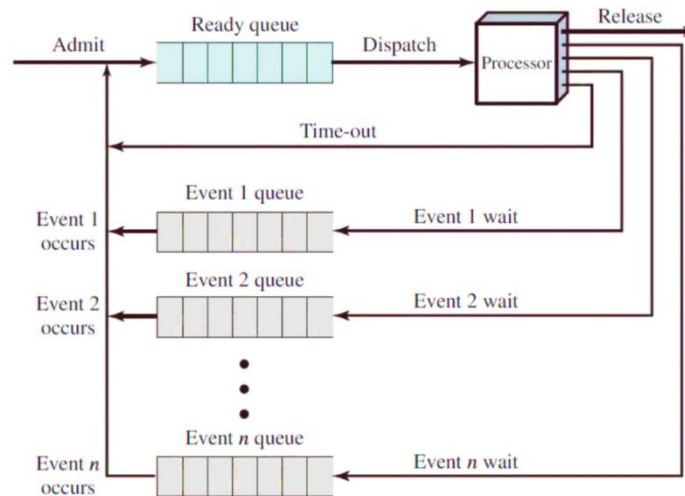
- A *semaphore* is a counter with two operations
 - P: (acquire) waits until the counter becomes positive and decrements it
 - V: (release) atomically increments the counter

Synchronization: Semaphores

```
type semaphore = record  
  N : integer  
  Q : queue of threads
```

```
procedure P(ref S : semaphore)  
  disable_signals  
  acquire_lock(scheduler_lock)  
  S.N -= 1  
  if S.N < 0  
    sleep_on(S.Q)  
  release_lock(scheduler_lock)  
  reenale_signals
```

```
procedure V(ref S : semaphore)  
  disable_signals  
  acquire_lock(scheduler_lock)  
  S.N += 1  
  if N ≤ 0  
    -- at least one thread is waiting  
    enqueue(ready_list, dequeue(S.Q))  
  release_lock(scheduler_lock)  
  reenale_signals
```



Java Synchronization (Semaphore)

- Counter example using semaphore
 - count is a shared variable
 - Accessing count is a critical section
 - Ensure the **mutual exclusion** when accessing the critical section
- Mutex
 - Mutex: a semaphore **initialized to 1**
 - Call **acquire** before entering the critical section
 - Call **release** after leaving the critical section

```
public static class Mutex {  
    static int count;  
    static Semaphore mutex;  
  
    public static void inc() {  
        try {  
            for(int i = 0; i < 1000000; i++) {  
                mutex.acquire(); //acquire mutex before CS  
                count++;  
                mutex.release(); //release mutex after CS  
            }  
        } catch(Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```

public static void test() {
    try {
        System.out.println("Testing Mutex...");
        count = 0;
        mutex = new Semaphore(1);
        Thread a = new Thread(() -> inc());
        Thread b = new Thread(() -> inc());
        a.start();
        b.start();
        a.join();
        b.join();
        System.out.println("count: " + count);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

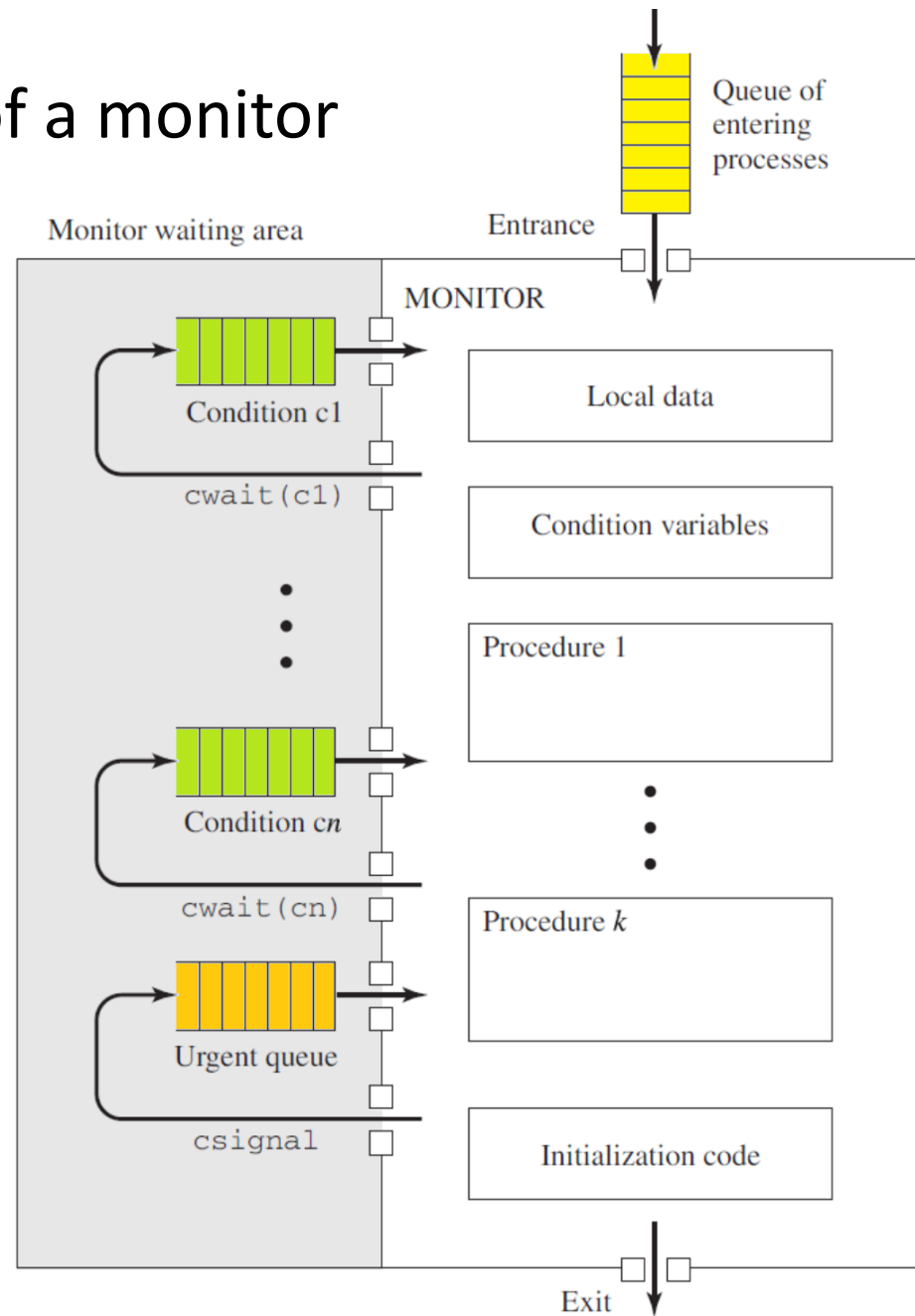
```

Testing Mutex...
count: 2000000

Synchronization: Monitors

- Monitor
 - A **module** or an object with
 - operations, internal state, and a number of condition variables
 - **Only one operation** of a monitor is active at a given time
 - **Calling an operation** on a busy monitor will be delayed until the monitor is free
 - A thread will be suspended by **waiting** on a condition variable
 - A thread may be resumed by **signaling** a condition variable

Structure of a monitor



Monitor-based Bounded Buffer

```
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
```

Java Synchronization (synchronized)

- Counter example using synchronized
 - Only one thread can be in a synchronized block
 - Entering a monitor
 - Mutual exclusion can be easily achieved
 - Within a synchronized block
 - wait: blocks on an event
 - notify: resume a blocked thread
 - notifyAll: resume all blocked threads

```

public static class Synchronized {
    static int count;
    static Object shared;
    public static void inc() {
        for(int i = 0; i < 1000000; i++) {
            synchronized(shared) {
                count++;
            }
        }
    }
    public static void test() {
        try {
            System.out.println("Testing Synchronized...");
            count = 0;
            shared = new Object();
            Thread a = new Thread(() -> inc());
            Thread b = new Thread(() -> inc());
            a.start();
            b.start();
            a.join();
            b.join();
            System.out.println("count: " + count);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}

```

Producer-Consumer Example

- Bounded buffer problem using Semaphores
 - Producer thread should be blocked if the buffer is full
 - Consumer thread should be blocked if the buffer is empty
- Semaphores to solve the problem
 - **slots**: represents the number of empty slots in the buffer (initialized to the size of buffer)
 - **items**: represents the number of items in the buffer (initialized to 0)

Producer-Consumer Example

```
import java.util.concurrent.Semaphore;

public class ProducerConsumer {
    public static class Queue {
        int[] data;           //bounded buffer
        int f, size;

        Semaphore mutex;      //lock to access queue
        Semaphore slots, items; //semaphores for the buffer

        public Queue(int n) {
            data = new int[n]; //bounded buffer (circular queue)
            f = size = 0;

            mutex = new Semaphore(1); //access lock
            slots = new Semaphore(n); //to block producer
            items = new Semaphore(0); //to block consumer
        }
    }
}
```

```

public void add(int x) {
    try {
        //try switch the order of slots and mutex
        //mutex.acquire();

        //block if no empty slot exists
        slots.acquire();

        //mutual exclusion to access queue (shared data)
        mutex.acquire();

        int i = (f+size) % data.length;
        data[i] = x;
        size++;

        //release the access lock
        mutex.release();

        //wake up any blocked consumer
        items.release();
    } catch(Exception e) {
        e.printStackTrace();
    }
}

```

```

public int remove() {
    int x = 0;
    try {
        //try switch the order of items and mutex
        //mutex.acquire();

        //block if no item exists
        items.acquire();

        //mutual exclusion to access queue (shared data)
        mutex.acquire();

        x = data[f];
        f = (f+1) % data.length;
        size--;

        //release the access lock
        mutex.release();

        //wake up any blocked consumer
        slots.release();
    } catch(Exception e) {
        e.printStackTrace();
    }
    return x;
}
}

```

```

public static void test() {
    try {
        System.out.println("Testing Producer Consumer...");

        Queue queue = new Queue(5);

        Thread producer = new Thread(() -> {
            for(int i = 0; i < 100000; i++)
                queue.add(i);
        });

        Thread consumer = new Thread(() -> {
            for(int i = 0; i < 100000; i++)
                onFalseThrow(i == queue.remove());
        });

        producer.start();
        consumer.start();
        producer.join();
        consumer.join();
        System.out.println("Done");
    } catch(Exception e) {
        e.printStackTrace();
    }
}

```

Assignment 9

- In this assignment, you are requested to implement the object based circuit simulator again.
- This time, each gate is running on its own thread while synchronized with others through Semaphores.
 - Download hw9.zip
 - Implement all TODOs.
- Due date 6/11/2020