Programming in Parallel

COMP755

"All games have morals; and the game of Snakes and Ladders captures, as no other activity can hope to do, the eternal truth that for every ladder you hope to climb, a snake is waiting just around the corner, and for every snake a ladder will compensate"

Salman Rushdie

Midnight's Children

Goals

- Be able to write simple concurrent programs
- Learn to avoid mutual exclusion problems

Uneven Execution Speed

- Threads are frequently interrupted. This means their execution “speed” moves forward in an uneven manner.
- At any instance, a thread can temporarily stop executing while the CPU does something else.

Execution Assumptions

- Values to be manipulated are loaded into registers, changes, then stored back to memory.
- Each thread has its own set of registers
- Shared values exist once in RAM.
- Any intermediate results of a complex expression are stored in private memory (i.e. stack)

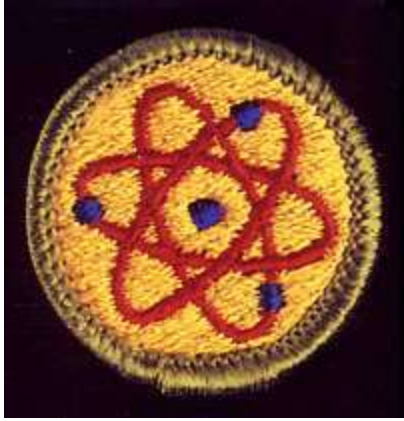
Nondeterministic Execution

- When multiple threads access shared data, the results may be nondeterministic.
- Consider two threads executing in parallel

Thread 1	**Thread 2**
x = 13;	**x = 47;**

- Depending upon which thread the OS executes first, the final value of x may be 13 or 47.

Atomic Actions



- Reads and writes to basic data types are atomic
- An atomic action is indivisible.
- You can see the state of an object before or after an atomic action, but you cannot see any intermediate states.
- If you atomically change a byte in memory, you will never catch it half way with only some of the bits changed.

Sequentially Equivalent

- We consider the results of the parallel execution of two threads to be ***Sequentially Equivalent*** if the results are equal to thread 1 running to completion and then thread 2 running **or** thread 2 running to completion and then thread 1.
- Results not equal to running the threads sequentially are not Sequentially Equivalent
- We often consider the results of parallel execution to be *"correct"* if it is sequentially equivalent.

Erroneous Results

- Sometimes when multiple threads access shared data, the results may not be sequentially equivalent
- Each thread has its own set of registers values. When the OS switches between threads, it saves and restores the registers
- Values in the register may not match RAM if another thread changed a variable

Think Machine Language

- Each thread has its own set of registers where arithmetic appears to be performed

counter in RAM	47	
Thread 1		Thread 2
mov eax, counter	47	
add eax,1	48	
		mov ebx, counter 47
		add ebx, 1 48
		mov counter, ebx 48
mov eax, counter	48	

Dual Core Race Condition

- With a dual core processor, two instructions can be executed at the same time
- Consider 2 threads doing counter++

counter in RAM 47

Thread 1	Thread 2
T0: mov eax, counter	mov ebx, counter
T1: add eax, 1	add ebx, 1
T2: mov counter, eax	mov counter, ebx

counter in RAM is now 48 not 49

Mutual Exclusion

- To avoid problems when sharing data between threads, each thread has to have exclusive access to the data.
- A segment of code that can only be executed by one thread at a time is called a ***critical section***.
- If a second thread attempts to execute the critical section while another thread is already executing there, the second thread will be suspended until the first thread exits the critical section.

Consistent Shared Data

- Whenever a thread **changes** a data value that could be read or changed by another thread, it must be done with mutual exclusion
- Whenever a thread **reads** a data value that could be changed by another thread, it must be done with mutual exclusion
- Multiple threads can safely read shared data at the same time

At-Most-Once property

- An expression ***e*** satisfies the at-most-once property if it refers to at most one simple variable ***y*** that might be changed by another process while ***e*** is being evaluated and it refers to ***y*** at most once.
- An assignment ***x = e*** satisfies the at-most-once property either if ***e*** satisfies the property and ***x*** is not read by another process or if ***e*** does not refer to any variable that might be changed.

Relaxed Requirements

- A block of code can safely read or write a shared data value without mutual exclusion if it meets the “At Most Once” property
- In machine language, a register load or store of a single word is atomic
- If you are going to reference only a single word of shared memory, the hardware can access it atomically

Which statements meet the At-Most-Once property?

shared variables S1, S2;
local variables L1, L2;

a) S2 = S1;
b) L1 = L2 + S1;
c) S1 = L2 + S2;

1. a
2. b
3. c
4. all of the above
5. none of the above

Using At-Most-Once

```
int flag = true;
//  in code used by multiple threads

if (flag) { /*do something locally*/ }
…
flag = false;
```

Mutual Exclusion Format

- The general format for creating a mutually exclusive critical section is:

…

```
Critical Section Entry code
        Critical Section
Critical Section Entry code
```

…

- Some languages do this automatically

Mutual Exclusion Techniques

- Busy Waiting
- Disabling interrupts
- Synchronized methods or blocks
- Semaphores
- Test and Set instructions

Busy Waiting

- With busy waiting, a program executes in a tight loop waiting for the critical section to become available.

```
while (thing != 0) {}
```

- The Dekker and Peterson algorithms provide mutual exclusion through busy waiting.
- Busy waiting uses lots of CPU time that would be better spent running other threads.
- **Programmers should avoid busy waiting**

Disabling Interrupts

- When interrupts are disabled, including the timer interrupt, nothing will stop the execution of the CPU
- This only works if there is just one processor
- By disabling interrupts, a thread can be assured that it will be able to execute a critical section without other threads running
- Only the OS can disable interrupts
- Disabling interrupts for a long time will cause a loss of I/O interrupts

Synchronized Methods

- Java and C# support synchronized methods.

  public synchronized int myfunc(char x){ … }

- Only one thread at a time can execute in any synchronized method of an object.
- If another thread calls that or any other synchronized method in the object, its execution will be suspended until the first thread returns from the method.

Synchronized Method Example

```
public class COMP755 {
  private int numThings = 0;
  public synchronized void add(int x) {
     numThings += x;
  }
  public synchronized void sub(int x) {
     numThings -= x;
  }
}
```

`wait()` Method

- **`wait()`** is a method of Object and thus a method of every object.
- When a thread executes the wait method, it is suspended until another thread executes the notify or notifyAll method on this object.
- **`wait()`** must be executed inside a synchronized method or block.

`wait( )` Method

- When a thread calls **`wait()`** inside a synchronized method, the method then becomes available to other threads.
- If you call **`wait()`** without specifying an object, it refers to **`this`** object.
- Calls to **`wait()`** should be in a while statement that checks on the condition allowing the thread to proceed.

`notify()` Method

- **`notify()`** will activate **one** thread that has been suspended by executing the **`wait()`** function on the same object.
- If no thread is suspended on that object, calling **`notify()`** has no effect. Calling **`notify()`** before the other thread calls **`wait()`** will not unlock the waiting thread.

`notifyAll()` method

- **`notifyAll()`** will activate **all** threads that have been suspended by executing the **`wait()`** function on the same object.
- If no thread is suspended on that object, calling **`notifyAll()`** has no effect.

Java Synchronized Method

```
public class prodcon {
  queue q;
   public synchronized void put (Thing x) {
     q.add(x);
     notify();
   }
   public synchronized Thing get( ) {
     while (q.empty()) wait();
     Thing y = q.remove();
     return y;
   }
}
```

Synchronized Blocks

- In addition to synchronized methods, you can restrict a block of code to only one thread.

```
synchronized (object) { … }
```

- Only one thread can execute inside the block.

Locking Objects

- A synchronized method or block locks the object involved. The object is unlocked at the end of the synchronized method or block.
- Locking an object does not prevent other threads from accessing non-private fields of the object.
- Locking is only done through synchronized methods or blocks.
- Java does not prevent or detect deadlock.

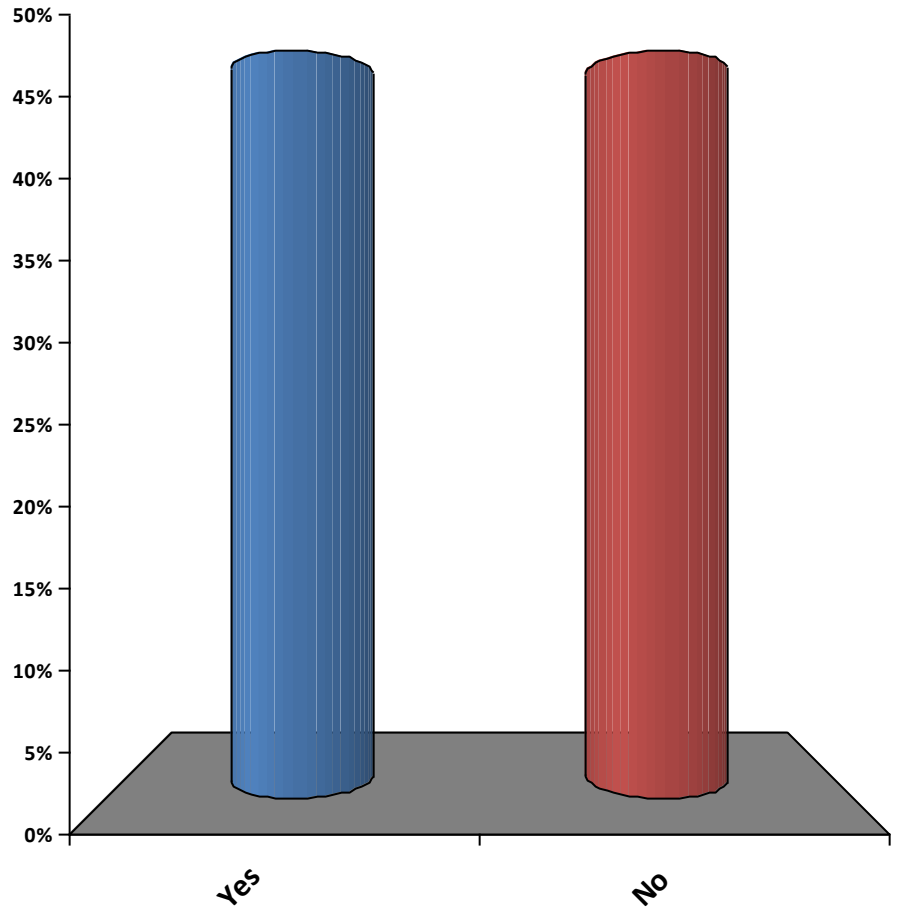
Multiple locks

- A single thread can lock the same object multiple times.

```
public static void main(String[] args) {
        Test t = new Test();
        synchronized(t) {
                synchronized(t) {
                        System.out.println("made it!");
        }
}
```

Can you have recursive synchronized methods?

1. Yes
2. No



Semaphores

- Semaphores were originally described by E. Dykstra.
- A semaphore has an integer counter and a queue of threads suspended on the semaphore.
- Semaphores can only be modified by two methods P *(or wait)* and V *(or signal)*.

P or Wait Method

```
void P() {
   while (this.counter == 0) {/* nothing */}
   this.counter--;
}
```

- If the semaphore counter is zero, the thread waits until it is nonzero.
- The counter is decremented.

V or Signal Method

```
void V() {
   this.counter++;
}
```

- Increments semaphore counter.
- If another thread was waiting in the P method, it will be allowed to continue.

OS Semaphore Implementation

- The previous definitions of the P and V methods used busy waiting
- A better implementation asks the OS to suspend the thread

Better P and V

```
void P() {
   if (this.counter == 0) {put thread on s.queue}
   this.counter--;
}
void V() {
   this.counter++;
   if (this.queue != NULL) {activate one thread}
}
```

Critical Section with Semaphores

```
semaphore s = new semaphore(1);

s.p();          // enter critical section

// only one thread at a time executing here

s.v();          // exit critical section
```

Special Critical Section with Semaphores

- To allow at most 2 threads in the critical section

```
semaphore s = new semaphore(2);

s.p();          // enter critical section

// up to two threads at a time executing here

s.v();          // exit critical section
```

Pthreads Locks

- Pthreads uses mutex objects as locks which are binary semaphores
- A muxex object can be created by:

pthread_mutex_t *mutexVariable*;

pthread_mutex_init (*mutexVariable*, NULL);

Pthreads Locking and Unlocking

- You can lock a mutex object with

pthread_mutex_lock (& *mutexVariable*);

- and unlock it with

pthread_mutex_unlock (& *mutexVariable*);

- Once locked, any thread that attempts to lock the mutex again will be blocked until the mutex is unlocked

Techniques that do NOT work

- Try to access a value. If it is unavailable, wait for a time period and try again
  - Waits should end when an event triggers a waiting condition
  - How long should you wait?
- Having one locking object for different logical conditions
  - A signal will awaken any waiting thread
  - Separate logical reasons for waiting requires separate objects

Rules for Parallelism

- Access to shared variables must be protected (unless the at-most-once principle holds).
- Think of semaphores as counters. Know what they are counting.
- Apply the patterns whenever possible

- Keep it simple

A Good Program *(and one that gets a good grade)*

- uses multiple processes or threads
- produces the correct result
- will not deadlock under any situation
- does not use busy waiting
- has maximum concurrency
- utilizes an efficient algorithm
- To aid in understanding the operation of the algorithm, the program should print useful information whenever an important event occurs
- It is helpful to slow the program's execution by inserting random sleeps in appropriate places

Producer Consumer with Semaphores

```
void producer (Thing x) {
p(mutex);
put thing on queue
v(mutex);
v(qsize);
}
```

```
Thing consumer () {
p(qsize);
p(mutex);
get thing from queue
v(mutex);
return thing;
}
```

Will this work?

```
void producer (Thing x) {
p(mutex);
put thing on queue
v(qsize);
v(mutex);
}
```

1. True
2. False

Will this work?

```
Thing consumer () {
p(mutex);
p(qsize);
get thing from queue
v(mutex);
return thing;
}
```

1. True
2. False

Concurrency Patterns

- **Mutual Exclusion** – Used to ensure that only one task executes a particular segment of code. Can be used for resource allocation.
- **Synchronization** – Coordinates the action of different tasks.
- **Barrier Synchronization** – Waits until all tasks are complete.

Mutual Exclusion with semaphores

```
semaphore s = 1;


p(s);
  // Critical section;
v(s);
```

Mutual Exclusion with synchronized methods

```
public class Example {
  public synchronized object getThing() {
     Critical section; // nothing to it
  }
}
```

Synchronization

- Multiple threads often need to interact. When one thread has completed something, it may need to inform another thread.
- In the producer/consumer problem, the producers need to be informed when the consumers have made a new item.

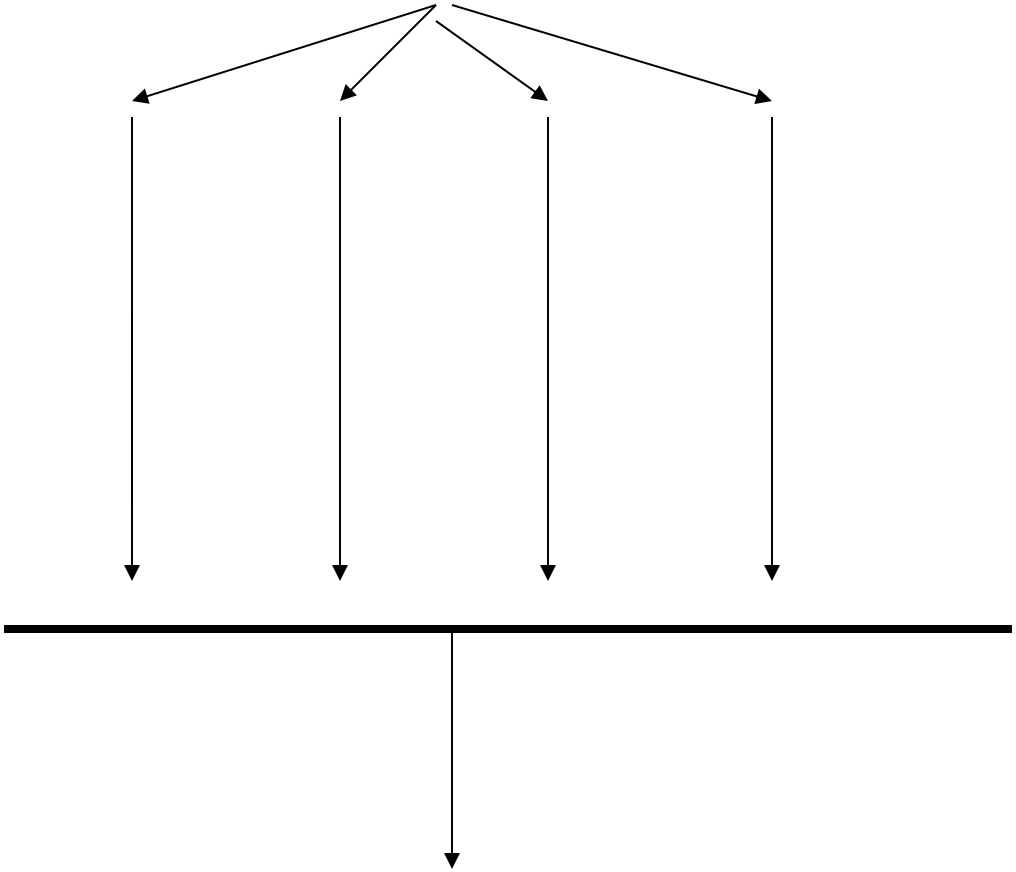
Synchronization with semaphores

```
semaphore numReady = 0;
void makeIt() {
  v(numReady);
}
void useIt() {
  p(numReady);
}
```

Synchronization with Synchronized Methods

```
public class numReady {
int available = 0;
public synchronized void makeIt(){
  available++;
  notify();
}
public synchronized void useIt() {
  while (available == 0) wait();
  available--;
}
```

Barrier Synchronization



One task continues after all tasks complete.

Barrier Synchronization with Semaphores

```
semaphore mutex = 1;
int numRunning = N;
// start threads and run the parallel stuff here
p(mutex);
numRunning--;
If (numRunning != 1) {
    v(mutex);
    exit();  // all but last task terminates
}
v(mutex);
```

Barrier Synchronization with Synchronized Methods

```
int numRunning = N;
public void run() {
  // run the parallel stuff here
  synchronized (this) {
     numRunning—;
     if(numRunning==1)morestuff();
  }
}
```

Java Barrier Synchronization with Join

```
SubThread[] t = SubThread[N];
// Start the parallel threads
for (i = 0; i < N; i++) {
  t[i] = new SubThread();
  t.start();
}
// Wait for thread to terminate
for (i = 0; i < N; i++)
     t[i].join();
```

Phaser class

```
void runTasks(List<Runnable> tasks) {
  final Phaser phaser = new Phaser(1); // "1" to register self
  // create and start threads
  for (final Runnable task : tasks) {
    phaser.register();
    task.start();
  }
  // allow threads to start and deregister self
  phaser.arriveAndDeregister();
 // all other threads have terminated by now
}

public class Task Thread() {
      public void run() {
         // do something useful here
        phaser.arrive();                    // signal thread is done
      }
    }
```

Semaphore Use

- If you have shared data, you will need a semaphore to provide mutual exclusion.
- Do a P() before you access the shared data and a V() afterwards.
- If a thread is going to have to wait for something, it will have to do a P() on a semaphore.
- There has to be one V() for every P().