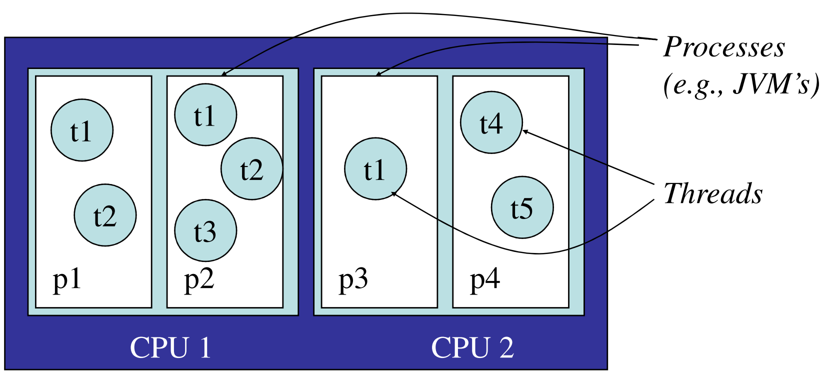
CMSC 330: Organization of Programming Languages

Threads

Multiprocessing

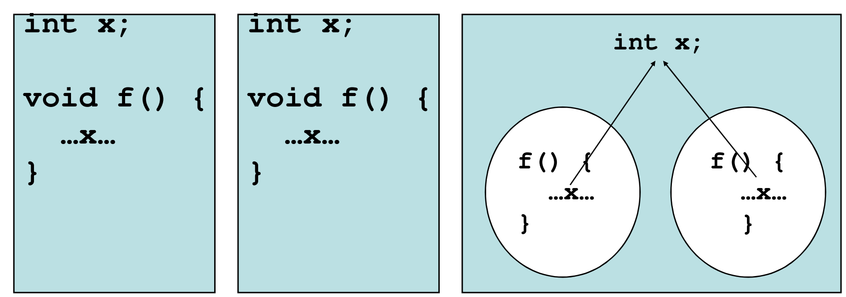
- Description
  - Multiple processing units (or multiple cores)
  - From single microprocessor to large computer clusters
  - Can perform multiple tasks in parallel simultaneously

Computation Abstractions



A computer

Processes vs. Threads



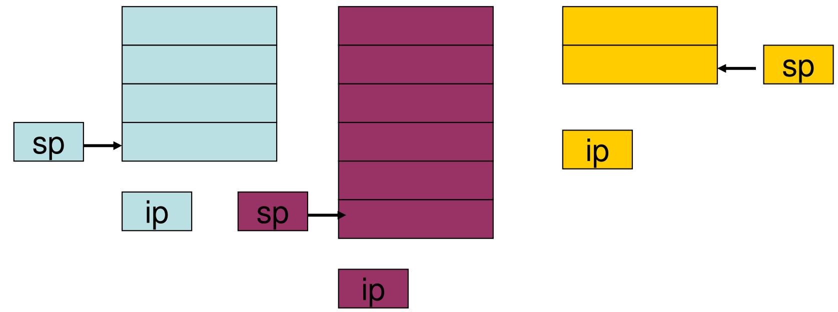
Processes do not share data

Threads share data within a process

So, What Is a Thread?

- **Conceptually**: it is a parallel computation occurring within a process
- **Implementation view**: it's a program counter and a stack. The heap and static area are shared among all threads
- All programs have at least one thread (main)

Implementation View



- Per-thread stack pointer (sp) and instruction pointer (ip)
  - Saved in memory when thread suspended
  - Copied to hardware sp/ip when thread resumes

Tradeoffs

- Threads can increase performance
  - Parallelism on multiprocessors
  - Concurrency of computation and I/O
- Natural fit for some programming patterns
  - Event processing
  - Simulations
- But increased complexity
  - Need to worry about safety, liveness, composition
- And higher resource usage

Programming Threads

- Thread creation is inexpensive
- Threads reside on same physical processor
- Threads share memory, resources
  - Except for local thread variables
- Shared-memory programming paradigm
  - Threads communicate via shared data
  - Synchronization used to avoid race conditions
- Limited scalability (tens of threads)

Programming Processes

- Process creation is expensive
  - Request to operating system
- Processes may reside on separate processors
- Processes do not share memory
- Message-passing programming paradigm
  - Messages using I/O streams, sockets, network, files
- Processes must cooperate to communicate
  - Actions performed to send and receive data
- Highly scalable (thousands of processors)

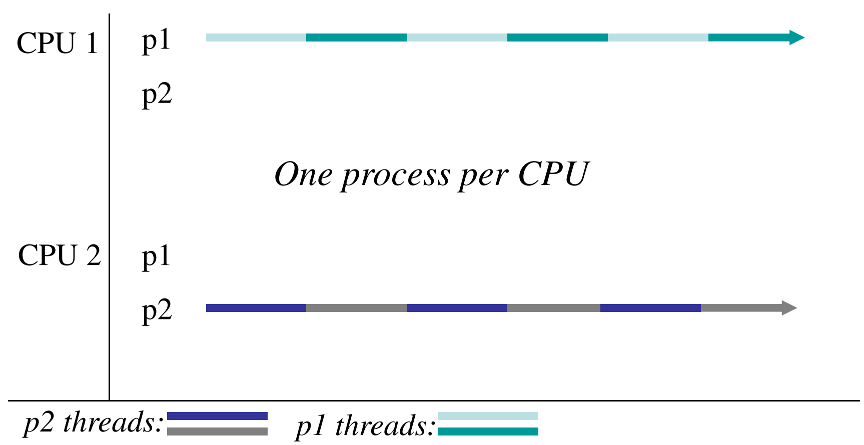
Threads in Programming Languages

- Threads are available in many languages
  - C, C++, Java, Ruby, OCaml,…
- In older languages (e.g., C and C++), threads are a platform-specific add-on
  - Not part of the language specification
  - Implemented as code libraries (e.g., pthreads)
- In newer languages (e.g., Java, Ruby), threads are part of the language specification
  - Not dependent on operating system
  - Can utilize special keywords, syntax

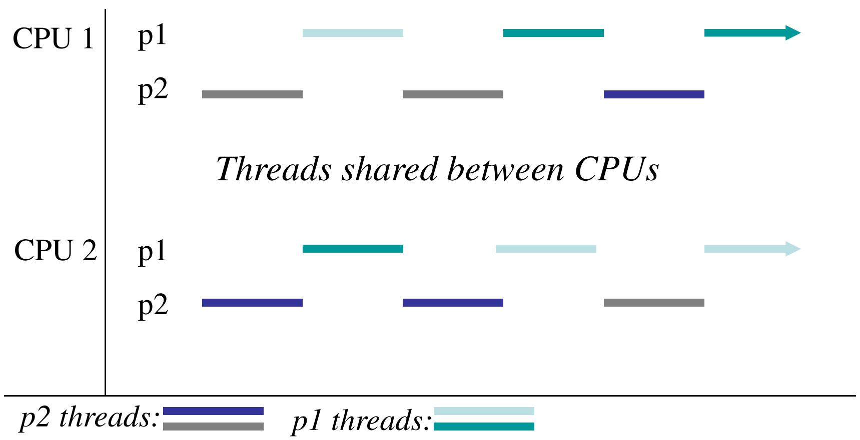
Concurrency

- A *concurrent* program is one that has multiple threads that may be active at the same time
  - Might run on one CPU
    - The CPU alternates between running different threads
    - The *scheduler* takes care of the details
      - Switching between threads might happen *at any time*
  - Might run *in parallel* on a *multiprocessor* machine
    - One with more than one CPU
    - May have multiple threads per CPU

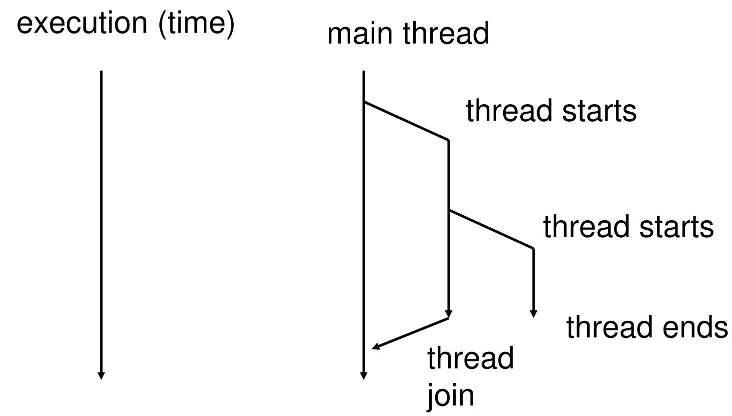
Scheduling Example (1)



Scheduling Example (2)



Thread Creation



Thread Creation in Java

- To explicitly create a thread:
  - Instantiate a Thread object (an object of class Thread *or* a subclass of Thread)
  - Override its run() method
  - Invoke the object's start() method
    - This will start executing the Thread's run() method concurrently with the current thread
  - A thread terminates when its run() method returns

Example: Alarms

- Goal: let's set alarms that will be triggered in the future
  - Input: time t (seconds) and message m
  - Result: we'll see m printed after t seconds

Example: Synchronous alarms

```
while (there is more input to read) {
  System.out.print("Alarm> ");

  // read user input into timeout and msg

  // wait (in seconds)
  try {
    Thread.sleep(timeout * 1000);
  } catch (InterruptedException e) { }
  System.out.println("(" + timeout + ") " + msg);
}
```


Making It Threaded (1)

```
public class AlarmThread extends Thread {

  private String msg = null;
  private int timeout = 0;

  public AlarmThread(String msg, int time) {
    this.msg = msg;
    this.timeout = time;
  }

  public void run() {
    try {
      Thread.sleep(timeout * 1000);
    } catch (InterruptedException e) { }
    System.out.println("(" + timeout + ") " + msg);
  }

}
```


Making It Threaded (2)

```
while (there is more input to read) {
  System.out.print("Alarm> ");

  // read user input into m and tm

  if (m != null) {
    // start alarm thread
    Thread t = new AlarmThread(m, tm);
    t.start();
  }
}
```


Alternative: The Runnable Interface

- Extending Thread prohibits a different superclass
- Instead implement Runnable
  - Declares that the class has a void run() method
- Construct a Thread from the Runnable
  - Constructor Thread(Runnable target)
  - Constructor Thread(Runnable target, String name)
- This approach is preferred

Thread Example Revisited

```
public class AlarmRunnable implements Runnable {

  private String msg = null;
  private int timeout = 0;

  public AlarmRunnable(String msg, int time) {
    this.msg = msg;
    this.timeout = time;
  }

  public void run() {
    try {
      Thread.sleep(timeout * 1000);
    } catch (InterruptedException e) { }
    System.out.println("(" + timeout + ") " + msg);
  }
}
```


Thread Example Revisited (2)

```
while (there is more input to read) {
  System.out.print("Alarm> ");

  // read user input into m and tm

  if (m != null) {
    // start alarm thread
    Thread t = new Thread(new
                    AlarmRunnable(m, tm));
    t.start();
  }
}
```


Notes: Passing Parameters

- run() doesn't take parameters
- We "pass parameters" to the new thread by storing them as private fields
  - In the extended class
  - Or the Runnable object
  - Example: the time to wait and the message to print in the AlarmThread class

Concurrency and Shared Data

- Concurrency is easy if threads don't interact
  - Each thread does its own thing, ignoring other threads
  - Typically, however, threads need to communicate with each other
- Communication is done by *sharing* data
  - In Java, different threads may access the heap simultaneously
  - But the scheduler might interleave threads arbitrarily
  - Problems can occur if we're not careful.

Race Condition Example

```
public class Example extends Thread {

  private static int count = 0;  // shared state

  public void run() {
    int y = count;
    count = y + 1;
  }

  public static void main(String args[]) {
    Thread t1 = new Example();
    Thread t2 = new Example();
    t1.start();
    t2.start();
  }

}
```


Race Condition Example

```
static int count = 0;

t1.run() {
  int y = count;
  count = y + 1;
}

t2.run() {
  int y = count;
  count = y + 1;
}
```

Shared state count = 0

Start: both threads ready to run. Each will increment the global count.

Race Condition Example

```
static int count = 0;

t1.run() {
  int y = count;
  count = y + 1;
}

t2.run() {
  int y = count;
  count = y + 1;
}
```

Shared state count = 0

y = 0

T1 executes, grabbing the global counter value into its own y.

Race Condition Example

```
static int count = 0;

t1.run() {
  int y = count;
  count = y + 1;
}

t2.run() {
  int y = count;
  count = y + 1;
}
```

Shared state **count = 1**

y = 0

T1 executes again, storing its value of y + 1 into the counter.

Race Condition Example

```
static int count = 0;

t1.run() {
  int y = count;
  count = y + 1;
}

t2.run() {
  int y = count;
  count = y + 1;
}
```

Shared state count = 1

y = 0

y = 1

T1 finishes. T2 executes, grabbing the global counter value into its own y.

Race Condition Example

```
static int count = 0;

t1.run() {
  int y = count;
  count = y + 1;
}

t2.run() {
  int y = count;
  count = y + 1;
}
```

Shared state **count = 2**

y = 0

y = 1

T2 executes, storing its incremented count value into the global counter.

But When it's Run Again?

- **count** could be 1 (make sure you see why)

Why?

- Different schedules can lead to different outcomes
  - This is a *race condition* or *data race*
- A thread was preempted in the middle of an operation
  - Reading and writing count was supposed to be *atomic*- to happen with no interference from other threads
  - But the schedule (interleaving of threads) that was chosen allowed atomicity to be violated
  - These bugs can be extremely hard to reproduce, and so hard to debug
    - Depends on what scheduler chose to do, which is hard to predict

Question

- If instead of

```
int y = count;
count = y + 1;
```

- We had written
  - `count++;`
- Would the result be any different?
- Answer: NO!
  - Don't depend on your intuition about atomicity

Question

- If you run a program with a race condition, will you always get an unexpected result?
  - No! It depends on the scheduler, and on the other threads/processes/etc, that are running on the same CPU
- Race conditions are hard to find

Synchronization

- Refers to mechanisms allowing a programmer to control the execution order of some operations across different threads in a concurrent program.
- Different languages have adopted different mechanisms to allow the programmer to synchronize threads.
- Java has several mechanisms

Aspects of Synchronization

- Atomicity
  - Locking to obtain mutual exclusion
  - What we most often think about
- Visibility
  - Ensuring that changes to object fields made in one thread are seen in other threads
- Ordering
  - Ensuring that you aren't surprised by the order in which statements are executed

Java's "synchronized" keyword

- Every object has an implicit associated lock
- synchronized allows you to acquire an object's lock
  - **`synchronized (obj) { body }`**
  - Obtains the lock associated with **`obj`**
  - Executes ***`body`***
  - Releases the lock when scope is exited, even in cases of exception or method return

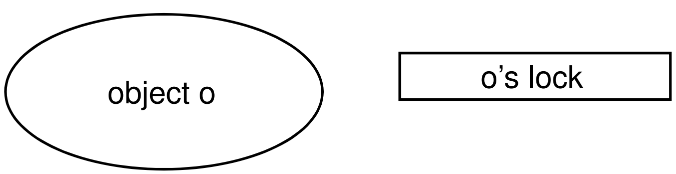
Example

```
static Object o = new Object();

void f() throws Exception {
  synchronized (o) {
    FileInputStream f = new FileInputStream("file.txt");
    // do something with f
    f.close();
  }
}
```

- Lock associated with o acquired before body executed, released even if exception is thrown

Discussion



- An object and its associated lock are different!
  - Holding the lock on an object does not affect what you can do with that object in any way
  - Ex:

```
synchronized (o) {      // acquires lock named o
  o.f();                // you can call o’s methods
  o.x = 3;              // you can read and write o’s fields
}
```


Example: Synchronizing on this

```
class C {
  int count;

  void inc() {
    synchronized (this) {
      count++;
    }
  }
}
```

```
C c = new C();
```

```
Thread 1
c.inc();
```

```
Thread 2
c.inc();
```

- Does this program have a race condition?
  - No, both threads acquire lock on the same object before they access shared data

Example: Synchronizing on this (cont'd)

```
class C {
  int count;

  void inc() {
    synchronized (this) {
      count++;
    }
  }

  void dec() {
    synchronized (this) {
      count--;
    }
  }
}
```

```
C c = new C();
```

```
Thread 1
c.inc();
```

```
Thread 2
c.dec();
```

- Race condition?
  - No, threads acquire lock on the same object before they access shared data

Example: Synchronizing on this (cont'd)

```
class C {
  int count;

  void inc() {
    synchronized (this) {
      count++;
    }
  }
}
```

```
C c1 = new C();
C c2 = new C();
```

```
Thread 1
c1.inc();
```

```
Thread 2
c2.inc();
```

- Does this program have a race condition?
  - No, threads acquire different locks, but they write to different objects, so that's ok

Synchronized Methods

- Marking a method as synchronized is the same as synchronizing on this in body of the method
  - The following two programs are the same

```
class C {

  int count;

  void inc() {
    synchronized (this) {
      count++;
    }
  }

}
```

```
class C {

 int count;

 synchronized void inc(){
   count++;
 }

}
```


Synchronized Methods (cont'd)

```
class C {

  int count;

  void inc() {
    synchronized (this) {
      count++;
    }
  }

  synchronized void dec() {
    count--;
  }

}
```

```
C c = new C();
```

```
Thread 1
c.inc();
```

```
Thread 2
c.dec();
```

- Race condition?
  - No, both acquire same lock

Locks (Java 1.5)

```
interface Lock {
  void lock();
  void unlock();
  ... /* some more stuff also */
}

class ReentrantLock implements Lock { ... }
```

- Explicit Lock objects are the same as the implicit lock used by synchronized keyword
- Only one thread can hold a lock at once
  - Other threads that try to acquire it *block* (or become suspended) until the lock becomes available
  - unlock() allows lock to be acquired by different thread

Avoiding Interference: Synchronization

```
public class Example extends Thread {

  private static int count = 0;
  static Lock lock = new ReentrantLock();

  public void run() {
    lock.lock();
    int y = count;
    count = y + 1;
    lock.unlock();
  }

  ...

}
```

***Lock**, for protecting the shared state*

***Acquires** the lock; Only succeeds if not held by another thread*

***Releases** the lock*

Applying Synchronization

```
int count = 0;

t1.run() {
  lock.lock();
  int y = count;
  count = y + 1;
  lock.unlock();
}

t2.run() {
  lock.lock();
  int y = count;
  count = y + 1;
  lock.unlock();
}
```

- Trace this and be sure you understand why it fixes the race condition

Synchronization Example (Java 1.4)

```
public class Example extends Thread {

  private static int count = 0;
  static Object value = new Object();

  public void run() {
    synchronized (value) {
      int y = count;
      count = y + 1;
    }
  }

  ...

}
```

***Value**, any Java object*

***Acquires** the lock associated w/ value; only succeeds if not held by another thread, otherwise blocks*

***Releases** the lock*

Different Locks Don't Interact

```
static int count = 0;
static Lock l =
    new ReentrantLock();
static Lock m =
    new ReentrantLock();

void inc() {
  l.lock();
  count++;
  l.unlock();
}
```

```
void inc() {
  m.lock();
  count++;
  m.unlock();
}
```

- This program has a race condition
  - Threads only block if they try to acquire a lock held by another thread