Overview

- What are threads?
- Thread scheduling, data races, and synchronization
- Thread mechanisms in Java
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

Why multiple threads?

- Performance:
  - Parallelism on multiprocessors
  - Concurrency of computation and I/O
- Can easily express some programming paradigms
  - Event processing
  - Simulations
- Keep computations separate, as in an OS
  - Java OS
Programming Threads

• Most languages have threads
  – C, C++, Objective Caml, Java, SmallTalk …
• The thread API differs with each, but most have the basic features we will now present

Thread Applications

• Web browsers
  – one thread for I/O
  – one thread for each file being downloaded
  – one thread to render web page
• Graphical User Interfaces (GUIs)
  – Have one thread waiting for each important event, like key press, button press, etc.
Thread Scheduling

- OS schedules a single-threaded process on a single processor
- Multithreaded process scheduling:
  - One thread per processor
    - Effectively splits a process across CPU’s
    - Exploits hardware-level concurrency
  - Many threads per processor
    - Need to share CPU in slices of time

Scheduling Example (1)

One process per CPU

CPU 1
- p1
- p2

CPU 2
- p1
- p2

p2 threads:  p1 threads:
Scheduling Example (2)

<table>
<thead>
<tr>
<th>CPU 1</th>
<th>p1</th>
<th>p2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CPU 2</th>
<th>p1</th>
<th>p2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Threads shared between CPU’s*

p2 threads: __________ p1 threads: __________

Scheduling Consequences

- **Concurrency**
  - Different threads from the same application can be running *at the same time* on different processors
- **Interleaving**
  - Threads can be “pre-empted” *at any time* in order to schedule other threads
Data Races

- Data shared between threads can become corrupted due to “inopportune” scheduling of the sharing threads. These are called “data races.”
- Therefore need to selectively control the scheduler to avoid such data accesses. This is usually done via synchronization.

Data Race Example

```c
int cnt = 0;
void thread1() {
    int y = cnt;
    cnt = y + 1;
}
void thread2() {
    int y = cnt;
    cnt = y + 1;
}
```

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

Shared state \( \text{cnt} = 0 \)
Data Race Example

```c
int cnt = 0;
void thread1() {
    int y = cnt;    // y = 0
    cnt = y + 1;
}
void thread2() {
    int y = cnt;    // y = 0
    cnt = y + 1;
}
```

**Shared state**  
`cnt = 0`

**Thread 1 executes, grabbing the global counter value into `y`.**

Thread 1 is pre-empted. Thread 2 executes, grabbing the global counter value into `y`. 
Data Race Example

```c
int cnt = 0;
void thread1() {
    int y = cnt;    // y = 0
    cnt = y + 1;
}
void thread2() {
    int y = cnt;    // y = 0
    cnt = y + 1;
}
```

**Shared state**

```
cnt = 1
```

Thread2 executes, storing the incremented cnt value.

---

**Data Race Example**

```c
int cnt = 0;
void thread1() {
    int y = cnt;    // y = 0
    cnt = y + 1;
}
void thread2() {
    int y = cnt;    // y = 0
    cnt = y + 1;
}
```

**Shared state**

```
cnt = 1
```

Thread2 completes. Thread1 executes again, storing the old counter value (1) rather than the new one (2)!
What happened?

- Thread1 was preempted after it read the counter but before it stored the new value.
- A particular way in which the execution of two threads is interleaved is called a schedule. We want to prevent this undesirable schedule.
- Undesirable schedules can be hard to reproduce, and so hard to debug.

Applying synchronization

```java
int cnt = 0;
lock l;
void thread1() {
    synchronized(l) {
        int y = cnt;
        cnt = y + 1;
    }
}
void thread2() {
    synchronized(l) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

- **Shared state** `cnt = 0`
- **Lock**, for protecting the shared state
- **Acquires** the lock; only succeeds if not held by another thread
- **Releases** the lock
Applying synchronization

int cnt = 0;
lock l;
void thread1() {
    synchronized(l) {
        int y = cnt;
        cnt = y + 1;
    }
}
void thread2() {
    synchronized(l) {
        int y = cnt;
        cnt = y + 1;
    }
}

Shared state  cnt = 0

Thread1 acquires lock l

Thread1 reads cnt into y
Applying synchronization

```java
typen cnt = 0;
lock l;
void thread1() {
    synchronized(l) {
        int y = cnt;
        y = 0
        cnt = y + 1;
    }
}
void thread2() {
    synchronized(l) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

*Shared state*  
*cnt = 0*

*Thread1 is pre-empted.*  
*Thread2 attempts to acquire lock l but fails because it’s held by Thread1, so it blocks*
Applying synchronization

```java
int cnt = 0;
lock l;

void thread1() {
    synchronized(l) {
        int y = cnt; y = 0
        cnt = y + 1;
    }
}

void thread2() {
    synchronized(l) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

Shared state  cnt = 1

Thread1 releases the lock and terminates

```
int cnt = 1
```

Applying synchronization

```
int cnt = 0;
lock l;

void thread1() {
    synchronized(l) {
        int y = cnt; y = 0
        cnt = y + 1;
    }
}

void thread2() {
    synchronized(l) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

Shared state  cnt = 1

Thread2 now can acquire lock l.

```
int cnt = 1
```
Applying synchronization

```java
int cnt = 0;
lock l;
void thread1() {
    synchronized(l) {
        int y = cnt;  // y = 0
        cnt = y + 1;
    }
}
void thread2() {
    synchronized(l) {
        int y = cnt;  // y = 1
        cnt = y + 1;
    }
}
```

**Shared state**  \( cnt = 1 \)

**Thread2 reads\( cnt \) into \( y \).**

---

Applying synchronization

```java
int cnt = 0;
lock l;
void thread1() {
    synchronized(l) {
        int y = cnt;  // y = 0
        cnt = y + 1;
    }
}
void thread2() {
    synchronized(l) {
        int y = cnt;  // y = 1
        cnt = y + 1;
    }
}
```

**Shared state**  \( cnt = 2 \)

**Thread2 assigns\( cnt \), then releases the lock**
Synchronization not a Panacea

- Can be expensive to acquire a lock
- Two threads can block on locks held by the other; this is called *deadlock*

```java
class Example {
    lock l;
    lock m;
    void thread1() {
        synchronized (l) {
            synchronized (m) {
                ...
            }
        }
    }
    void thread2() {
        synchronized (m) {
            synchronized (l) {
                ...
            }
        }
    }
}
```

Other Thread Operations

- *Condition variables*: **wait** and **notify**
  - Alternative synchronization mechanism
- **Yield**
  - Voluntarily give up the CPU
- **Sleep**
  - Wait for a certain length of time
Thread Lifecycle

- While a thread executes, it goes through a number of different phases
  - **New**: created but not yet started
  - **Runnable**: either running, or able to run on a free CPU
  - **Blocked**: waiting for I/O or on a lock
  - **Sleeping**: paused for a user-specified specified interval

Java Threads

- The class `java.lang.Thread`
  - Implements the basic threading abstraction
  - Can extend this class to create your own threads
- The interface `java.lang.Runnable`
  - Can create a thread by passing it a class that implements this interface
  - Favors composition over inheritance; more flexible
Extending class Thread

- Can build a thread class by extending `java.lang.Thread`
- Must supply a public void `run()` method
- Start a thread by invoking the `start()` method
- When a thread starts, executes `run()`
- When `run()` returns, thread is finished/dead

Example: Synchronous alarms

```java
while (true) {
    System.out.print("Alarm> ");

    // read user input
    String line = b.readLine();
    parseInput(line);

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

while (true) {
    System.out.print("Alarm> ");

    // read user input
    String line = b.readLine();
    Thread t = parseInput(line);

    // wait (in secs) asynchronously
    if (t != null)
        t.start();
}
Runnable interface

- Extending Thread means can’t extend any other class
- Instead implement Runnable
  - declares that the class has a void run() method
- Can construct a new Thread
  - and give it an object of type Runnable as an argument to the constructor
  - Thread(Runnable target)
  - Thread(Runnable target, String name)

Thread example revisited

```java
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);
    }
}
```
Change is in object creation

- Old parseInput does
  - return new AlarmThread(m,t);
- New parseInput does
  - return new Thread(new AlarmRunnable(m,t));
- Code in while loop doesn’t change

Another example

```java
public class ThreadDemo implements Runnable {
    public void run() {
        for (int i = 5; i > 0; i--)
            try { Thread.sleep(1000); }
            catch (InterruptedException e) { }
        System.out.println("exiting " + Thread.currentThread());
    }

    public static void main(String[] args) {
        Thread t = new Thread(new ThreadDemo(), "Demo Thread");
        System.out.println("main thread: " + Thread.currentThread());
        System.out.println("Thread created: " + t);
        t.start();
        try { Thread.sleep(3000); }
        catch (InterruptedException e) {}
        System.out.println("exiting "+Thread.currentThread());
    }
}
```
InterruptedException

- A number of thread methods throw it
  - really means: “wake-up call!”
- `interrupt()` tries to wake up a thread
- Won’t disturb the thread if it is working
- Will wake up the thread if it is sleeping, or otherwise waiting (or will do so when such a state is entered)
  - Thrown by `sleep()`, `join()`, `wait()`

Thread scheduling

- When multiple threads share a CPU, must decide:
  - When the current thread should stop running
  - What thread to run next
- A thread can voluntarily `yield()` the CPU
- `Preemptive schedulers` can de-schedule the current thread at any time
  - But not all JVM implementations use preemptive scheduling; so a thread stuck in a loop may never yield by itself. Therefore, put `yield()` into loops
- Threads are descheduled whenever they block (e.g. on a lock or on I/O) or go to sleep
Which thread to run next?

- The scheduler looks at all of the runnable threads; these will include threads that were unblocked because
  - A lock was released
  - I/O became available
  - They finished sleeping, etc.
- Of these threads, it considers the thread’s priority. This can be set with `setPriority()`. Higher priority threads get preference.
  - Oftentimes, threads waiting for I/O are also preferred.

Simple thread methods

- `void start()`
- `boolean isAlive()`
- `void setPriority(int newPriority)`
  - thread scheduler might respect priority
- `void join()` throws `InterruptedException`
  - waits for a thread to die/finish
Example: threaded, sync alarm

```java
while (true) {
    System.out.print("Alarm> ");

    // read user input
    String line = b.readLine();
    Thread t = parseInput(line);

    // wait (in secs) asynchronously
    if (t != null)
        t.start();
    // wait for the thread to complete
    t.join();
}
```

Simple static thread methods

- void yield()
  - Give up the CPU
- void sleep(long milliseconds)
  - Sleep for the given period
- Thread currentThread()
  - Thread object for currently executing thread
- All apply to thread invoking the method
Daemon threads

- void setDaemon(boolean on)
  - Marks thread as a daemon thread
- By default, thread acquires status of thread that spawned it
- Program execution terminates when no threads running except daemons

Example - why synchronization?

class UnSyncTest extends Thread {
    String msg;
    public UnSyncTest(String s) {
        msg = s;  start();  }
    public void run() {
        System.out.println("[");
        try { Thread.sleep(1000); } catch(InterruptedException e) {} System.out.println("]"); }
    public static void main(String [] args) {
        new UnSyncTest("Hello");
        new UnSyncTest("UnSynchronized");
        new UnSyncTest("World");
    }
}
Synchronization Topics

• Locks
• `synchronized` statements and methods
• `wait` and `notify`
• Deadlock

Locks

• *Any* Object subclass can act as a lock
• Only one thread can hold the lock on an object
  – other threads block until they can acquire it
• If your thread already holds the lock on an object
  – can lock many times
  – Lock is released when object unlocked the corresponding number of times
• No way to only attempt to acquire a lock
  – Either succeeds, or blocks the thread
Synchronized methods

- A method can be synchronized
  - add `synchronized` modifier before return type
- Obtains the lock on object referenced by `this`, before executing method
  - releases lock when method completes
- For a `static synchronized` method
  - locks the class object

Synchronized statement

- `synchronized (obj) { statements }`
- Obtains the lock on `obj` before executing statements in block
- Releases the lock when the statements block completes
- Finer-grained than synchronized method
Synchronization example

class SyncTest extends Thread {
    String msg;
    public SyncTest(String s) {
        msg = s;
        start();
    }
    public void run() {
        synchronized (SyncTest.class) {
            System.out.print("[" + msg);
            try { Thread.sleep(1000); }
            catch (InterruptedException e) {};
            System.out.println("]");
        }
    }
    public static void main(String [] args) {
        new SyncTest("Hello");
        new SyncTest("Synchronized");
        new SyncTest("World");
    }
}

Wait and Notify

• Must be called inside synchronized method or block of statements
• a.wait()
  – releases the lock on a
  – adds the thread to the wait set for a
  – blocks the thread
• a.wait(int m)
  – limits wait time to m milliseconds (but see below)
Wait and Notify (cont.)

- `a.notify()` resumes one thread from `a`'s wait list
  - and removes it from wait set
  - no control over which thread
- `a.notifyAll()` resumes all threads on `a`'s wait list
- resumed thread(s) must reacquire lock before continuing
- `wait` doesn’t give up locks on any other objects
  - e.g., acquired by methods that called this one

Producer/Consumer Example

```java
public class ProducerConsumer {
    private boolean full = false;
    private Object obj;

    public ProducerConsumer() { }
    public ProducerConsumer(Object o) {
        obj = o;  f = true;  }

    synchronized void produce(Object o) {
        while (full) wait();
        obj = o;  full = true;
        notifyAll();
    }

    synchronized Object consume() {
        while (!full) wait();
        full = false;
        notifyAll();
        return obj;
    }
}
```

Do we need to wake up everybody?
Changed example –
Attempt to refine synch.

synchronized void produce(Object o) {
    while (full) {
        wait();
        if (full) notify();
    }
    obj = o; full = true;
    notify();
}

Still might not work –
no guarantee about
who gets woken up

A Better Solution?

synchronized void produce(Object o) {
    synchronized (empty) {
        try { e.wait(); }
        catch (InterruptedException ex) {} }
    }
    obj = o; full = true;
    synchronized (f) {
        f.notify();
    }
}

Use two objects,
empty and f, to allow
two different wait sets

synchronized Object consume() {
    while (!full) {
        wait();
        if (!full) notify();
    }
    full = false;
    notify();
    return obj;
}

But does this work?

Object o = obj; full = false;
synchronized(e){e.notify();
return obj;
A Better Solution

```java
void produce(Object o) {
    while (full) {
        synchronized (empty) {
            try {
                empty.wait();
            } catch (InterruptedException e) {
            }
        }
        synchronized (this) {
            obj = o; full = true;
        }
        synchronized (full) {
            full.notify();
        }
    }
}

Object consume() {
    Obj o;
    while (!full) {
        synchronized (full) {
            try { full.wait(); } catch (InterruptedException e) {
            }
        }
        synchronized (this) {
            o = obj;
            full = false;
        }
        synchronized (empty) {
            empty.notify();
        }
    }
    return o;
}
```

Synchronizing on two locks at the same time can be dangerous

notify() vs. notifyAll()

- Very tricky to use notify() correctly
  - notifyAll() generally much safer
- To use correctly, should have:
  - all waiters be equal
  - each notify only needs to wake up 1 thread
  - handle `InterruptedException` correctly
### Interrupted Exception Example

- Threads t1 and t2 are waiting
- Thread t3 performs a notify
  - thread t1 is selected
- Before t1 can acquire lock, t1 is interrupted
- t1’s call to wait throws InterruptedException
  - t1 doesn’t process notification
  - t2 doesn’t wake up

### Handling InterruptedException

```java
synchronized (this) {
    while (!ready) {
        try { wait(); }
        catch (InterruptedException e) {
            notify();
            throw e;  }
        // do whatever
    }
}
```
Deadlock

- Quite possible to create code that deadlocks
  - Thread 1 holds lock on A
  - Thread 2 holds lock on B
  - Thread 1 is trying to acquire a lock on B
  - Thread 2 is trying to acquire a lock on A
  - Deadlock!
- Not easy to detect when deadlock has occurred
  - other than by the fact that nothing is happening
  - Use logging server!

A common multi-threading bug

- Threads might cache values
- Obtaining a lock forces the thread to get fresh values
- Releasing a lock forces the thread to flush out all pending writes
- volatile variables are never cached
- sleep(...) doesn’t force fresh values
- Many compilers don’t perform these optimizations
- Problem might also occur with multiple CPUs
Guidelines to simple/safe multi-threaded programming

- Synchronize access to shared data
- Don’t hold a lock on more than one object at a time
  - could cause deadlock
- Hold a lock for as little time as possible
  - reduces blocking waiting for locks
- While holding a lock, don’t call a method you don’t understand
  - e.g., a method provided by someone else, especially if you can’t be sure what it locks

Guidelines (cont.)

- Have to go beyond these guidelines for more complex situations
  - but need to understand threading and synchronization well
- We’ll discuss threads more from the textbook *Concurrent Programming in Java*
  and from a talk at a Java conference by Bill Pugh and Doug Lea