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

CMSC 433, Fall 2002 - Michael Hicks 9

CMCS 433, Fall 2002 - Michael Hicks 10
Scheduling Example (2)

CPU 1
- p1
- p2

CPU 2
- p1
- p2

Threads shared between CPU’s

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;
}
```

*Shared state*  
\(cnt = 0\)

*Start: both threads ready to run. Each will increment the global count.*
# Data Race Example

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

*Shared state*  
\[ \text{cnt} = 0 \]

*Thread1 executes, grabbing the global counter value into y.*  
\[ \text{cnt} = 0 \]

*Thread1 is pre-empted. Thread2 executes, grabbing the global counter value into y.*  
\[ \text{cnt} = 0 \]
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 \( cnt = 1 \)

Thread2 executes, storing the incremented \( cnt \) value.

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

```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`

**Thread1 acquires lock l**

**Thread1 reads cnt into y**

```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;
    }
}
```
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`

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

**Shared state**  
`cnt = 1`

**Thread1 runs, assigning to** cnt
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 = 1

Thread1 releases the lock and terminates

Thread2 now can acquire lock l.

CMCS 433, Fall 2002 - Michael Hicks
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**

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**

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
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