INF 212
ANALYSIS OF PROG. LANGS
CONCURRENCY

Instructors: Crista Lopes
Copyright © Instructors.
Basics
Concurrent Programming

- More than one thing at a time

Examples:
- Network server handling hundreds of clients
- Data processing spread across multiple CPUs
- App receiving input from several peripherals
Concurrency

1 CPU

```
Thing 1 → run → Scheduler → run → Scheduler → run → Scheduler → run → Scheduler

switch

Thing 2 → run → Scheduler → run → Scheduler → run → Scheduler → run → Scheduler
```
Concurrency

- Many CPUs
- When there are more things than CPUs, CPUs are shared like previous slide
Tasks alternate between processing and IO

IO must wait for data

Runtime system will resume task when data becomes available
CPU-bound tasks

- When it spends most of the time processing
  - CPU is busy

  ![Diagram showing CPU-bound task (Thing 1) running and then held up]

- Examples:
  - Math
  - Image processing
**IO-bound tasks**

- When it spends most of the time doing IO
  - CPU is idle

  ![Diagram](image)

  **Thing 1** → run →...

- Examples:
  - User input
  - Networking
  - Files
Shared memory

- Simultaneous access to memory space
Separate processes, shared machine

- No shared memory
  - Inter-Process Communication (IPC)

Diagram:
- Two things (Thing 1 and Thing 2) running on separate processes.
- Each process has its own memory (CPU 1 and CPU 2).
- Communication between processes is facilitated by Inter-Process Communication (IPC).
Different machines, distributed computing

- Network
  - Sockets

Diagram:

- **Thing 1**
  - Process
  - Memory
  - Run
  - Machine 1

- **Thing 2**
  - Process
  - Memory
  - Run
  - Messages
  - Machine 2
Shared Memory

Threads
Simplified Machine Model

- Registers
- Code
- Data

Program counter
Environment pointer

Stack
Heap

PROCESS
Threads

Single-threaded vs Multithreaded
Threads: benefits

- Responsiveness
- Resource sharing
- [Performance in multi-core machines]
Threads: pitfalls

- Race conditions
- **Heisenbugs**
- Easy to end up with a colossal code mess
- Heavy startup time for new threads
  - [Thread pools to the rescue]
Two Types of Threads

- User-level threads libraries:
  - POSIX Pthreads
  - Java Threads
  - .NET Threads
  - Python Threads
- ...  

- Kernel threads
Thread API Examples
class ThreadedObject implements Runnable {
    Thread t;
    ThreadedObject() {
        // Create a thread
        t = new Thread(this, "Demo Thread");
        System.out.println("Child thread: "+ t);
        t.start();  // Start the thread
    }

    // This is the entry point for the ThreadedObject thread.
    public void run() {
        for (int i = 5; i > 0; i--) {
            System.out.println("Child Thread: "+ i);
            // Let the thread sleep for a while.
            Thread.sleep(50);
        }
        System.out.println("Exiting child thread.");
    }
}
public class ThreadDemo {
    public static void main(String args[]) {
        new ThreadedObject();  // create a new thread
        for (int i = 5; i > 0; i--) {
            System.out.println("Main Thread: " + i);
            Thread.sleep(100);
        }
        System.out.println("Main thread exiting.");
    }
}

OOP Threads (Java/C#/PHP/Python) v.1—instantiate Thread
class ThreadedObject extends Thread {
    ThreadedObject() {
        super("Demo Thread");
        System.out.println("Child thread: " + this);
        start(); // Start the thread
    }

    public void run() {
        for (int i = 5; i > 0; i--) {
            System.out.println("Child Thread: " + i);
            // Let the thread sleep for a while.
            Thread.sleep(50);
        }
        System.out.println("Exiting child thread.");
    }
}
public class ThreadDemo {
    public static void main(String args[]) {
        new ThreadedObject(); // create a new thread
        for (int i = 5; i > 0; i--) {
            System.out.println("Main Thread: "+i);
            Thread.sleep(100);
        }
        System.out.println("Main thread exiting.");
    }
}
#lang racket

(thread (lambda ()
  (for ([i 10])
    (sleep 2)
    (printf "thread 1\n"))))

(thread (lambda ()
  (for ([i 20])
    (sleep 1)
    (printf "thread 2\n"))))
#include <string>
#include <iostream>
#include <thread>
using namespace std;

// The function we want to make the thread run.
void task1(string msg)
{
    cout << "task1 says: " << msg;
}

int main()
{
    // Constructs the new thread and runs it. Does not block
    thread t1(task1, "Hello");

    // Makes the main thread wait for the new thread to finish
    t1.join();
}
Common operations on threads

- Create
- Start
- Sleep
- Suspend
- Resume
- Yield
- Stop

(you’ll find most of these in most thread APIs)
Concurrency Control
Threads – shared resources

- Single-threaded
- Multithreaded

- Code
- Heap
- Files
- Registers
- Stack

DANGER
class ThreadedObject extends Thread {
    Counter c;
    public void run() {
        for (int i = 500; i > 0; i--) {
            System.out.println("Child Thread: " + i);
            if (i % 2 == 0) c.increment();
            else c.decrement();
            Thread.sleep(50);
        }
        System.out.println("Exiting child thread.");
    }
}

class Counter {
    private int c = 0;

    public void increment() {
        c = c + 1;
    }

    public void decrement() {
        c = c - 1;
    }
}

Shared data problems
public class ThreadDemo {
    public static void main(String args[]) {
        Counter c = new Counter();
        ThreadedObject[] ao = new ThreadedObject[5];
        // Create 5 threads
        for (int i = 5; i > 0; i--)
            ao[i] = new ThreadedObject(c); // create a new thread

        // Wait for the threads to finish
        for (ThreadedObject a : ao)
            a.join();
        System.out.println("Main thread exiting.");
    }
}
Shared data problems

Threaded Object

Threaded Object

Threaded Object

Threaded Object

Threaded Object

Counter c

increment
decrement
Things that can go wrong

```java
class Counter {
    private int c = 0;

    public void increment() {
        c = c + 1;
    }

    public void decrement() {
        c = c - 1;
    }
}
```

One action lost:
- $T_1$ enters the method
- $T_1$ reads value of $c$ (e.g. 3)
- $T_1$ is interrupted by scheduler
- $T_2$ enters the method
- $T_2$ reads value of $c$ (3)
- $T_2$ adds 1 to $c$ (4)
- $T_2$ assigns $c$ to the new value (4)
- $T_2$ returns
- $T_1$ is resumed
- $T_1$ adds 1 to [old] value of $c$ (4)
- $T_1$ assigns $c$ to that value (4)

Race condition
Things that can go wrong

```java
class Counter {
    private int c = 0;

    public void increment() {
        c = c + 1;
    }

    public void decrement() {
        c = c - 1;
    }
}
```

**One action lost:**
- T1 enters increment
- T2 enters decrement
- T1 reads value of c (e.g. 3)
- T2 reads value of c (3)
- T1 adds 1 to its value of c (4)
- T2 subtracts 1 to its value of c (2)
- T1 assigns new value to c (4)
- T2 assigns c (2)

Race condition
Race conditions

- Corruption of shared data due to thread scheduling
- Very hard to deal with
  - Non-deterministic
  - Hard to reproduce
  - Program may be ok for long time until it hits a race condition
Concurrent control

- Primitives for controlling the execution of concurrent threads over the same code
Concurency control (Java)

class Counter {
    private int c = 0;

    public synchronized void increment() {
        c = c + 1;
    }

    public synchronized void decrement() {
        c = c - 1;
    }
}
Concurrent control (others)

class Counter {
    private int c = 0;
    Object o = new Object();

    public void increment() {
        lock (o) {
            c = c + 1;
        }
    }

    public void decrement() {
        lock (o) {
            c = c - 1;
        }
    }
}
Concurrent control (others’)

```java
class Counter {
    private int c = 0;
    Lock o = new Lock();

    public void increment() {
        o.acquire();
        c = c + 1;
        o.release();
    }

    public void decrement() {
        o.acquire();
        c = c - 1;
        o.release();
    }
}
```

Only one thread at a time can enter the locked blocks
Locks

- They seem easy
- They are very hard to manage
Lock management

- Acquired locks must always be released
  - Including when exceptions occur
  - A lock that is not released will prevent any other thread from entering the block forever
class Counter {
    private int c = 0;
    Object o1 = new Object();
    Object o2 = new Object();
    public void increment() {
        lock (o1) {
            lock (o2) {
                c = c + 1;
            }
        }
    }

    public void decrement() {
        lock (o2) {
            lock (o1) {
                c = c - 1;
            }
        }
    }
}
Threads Summary

- Nice, but...
  - Synchronization primitives are hard to get right
    - Deadlocks
    - Race conditions
  - Many corner cases
  - Bugs hard to reproduce
  - Potentially bad performance (locking is heavy)

- Many reasons to avoid threads!
  - ... or at least to use them under strict constraints