

Based on slides by Harsha V. Madhyastha

# EECS 482 Introduction to Operating Systems

Spring/Summer 2020

Lecture 2: Threads

Nicole Hamilton

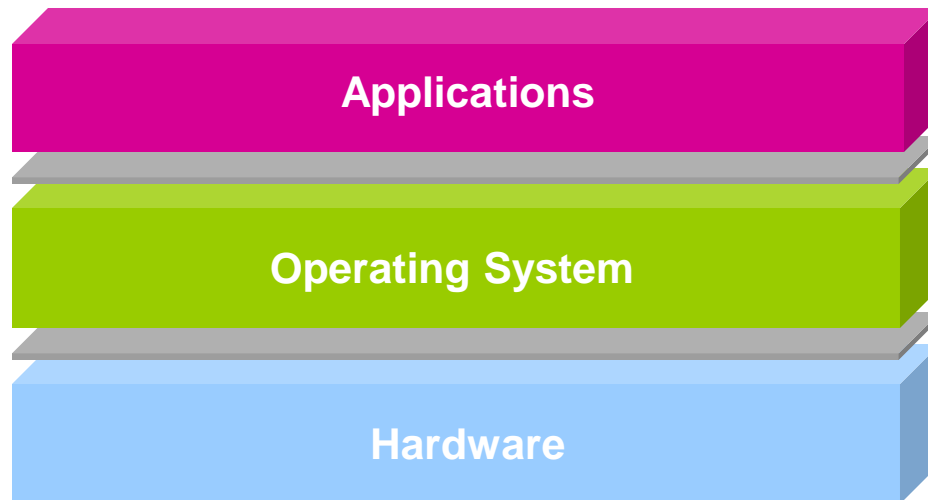
[https://web.eecs.umich.edu/~nham/  
nham@umich.edu](https://web.eecs.umich.edu/~nham/nham@umich.edu)

# Recall: What does an OS do?

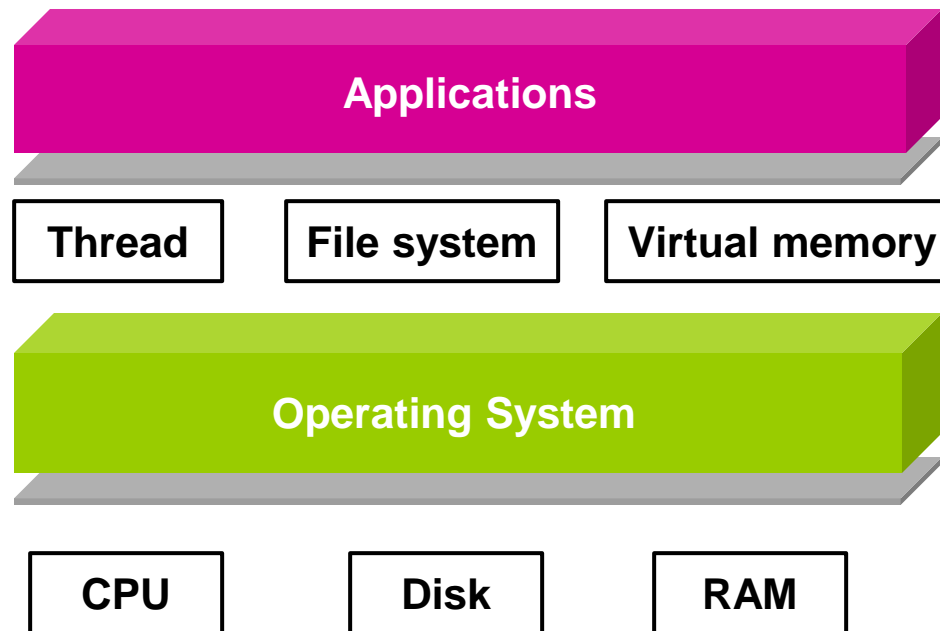
Enables portability

Creates **abstractions** to make hardware easier to use

Manages **shared** hardware resources



# OS Abstractions



# Upcoming Schedule

This lecture starts a class segment that covers processes, threads, and synchronization

Perhaps the most important in this class

**Basis for Projects 1 and 2**

# Managing Concurrency

## Source of OS complexity

Multiple users, programs, I/O devices, etc.

Originally for efficient use of H/W, but useful even now

```
main( )  
{  
  getInput();  
  computeResult();  
  printOutput();  
}
```

## How to manage this complexity?

Divide and conquer

Modularity and abstraction

# The Process

The process is the OS  
**abstraction for execution**

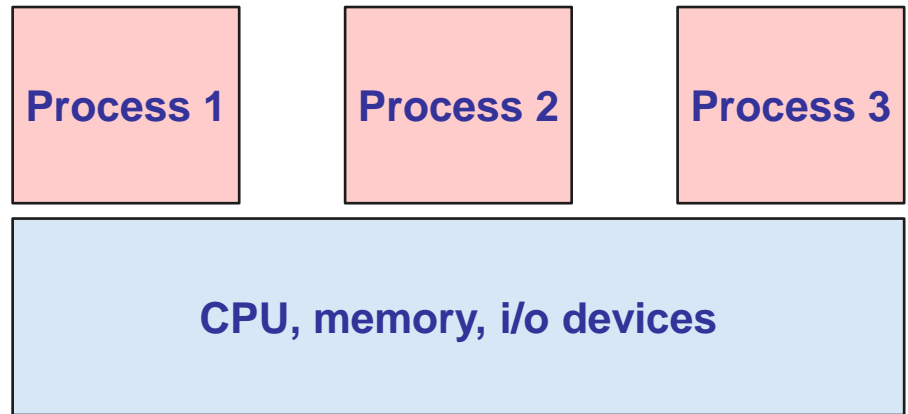
Also sometimes called a **job**  
or a **task**

For each area of OS, ask

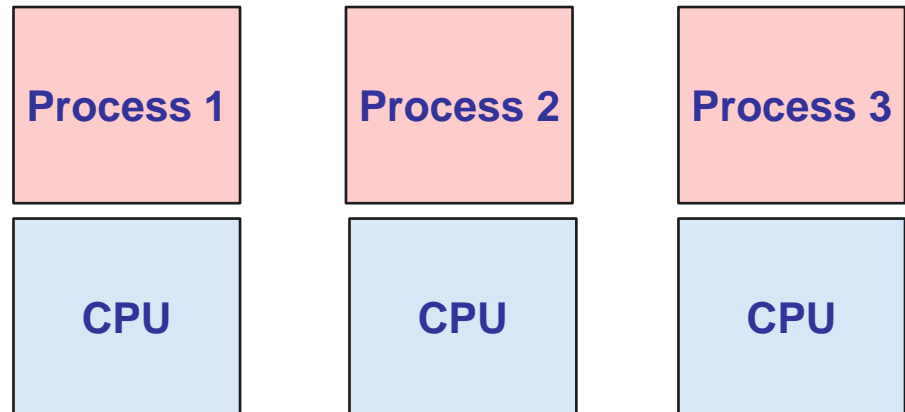
What interface does  
hardware provide?

What interface does OS  
provide?

The reality

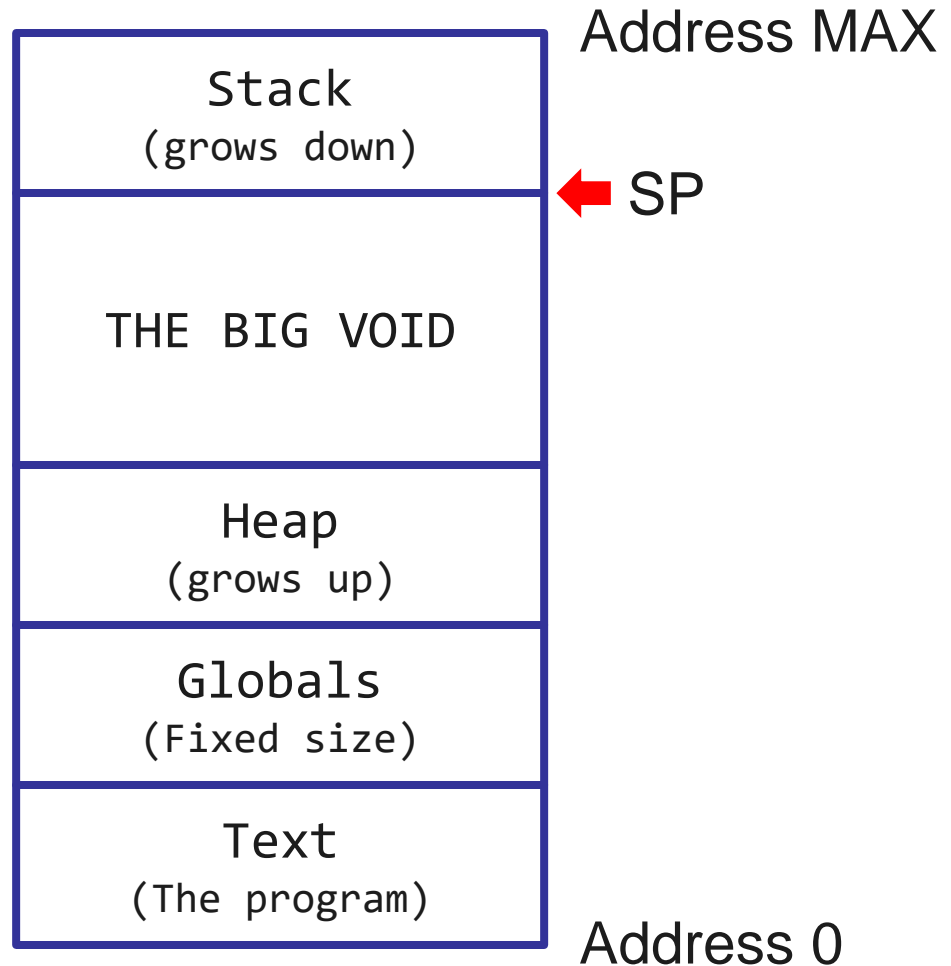


The abstraction



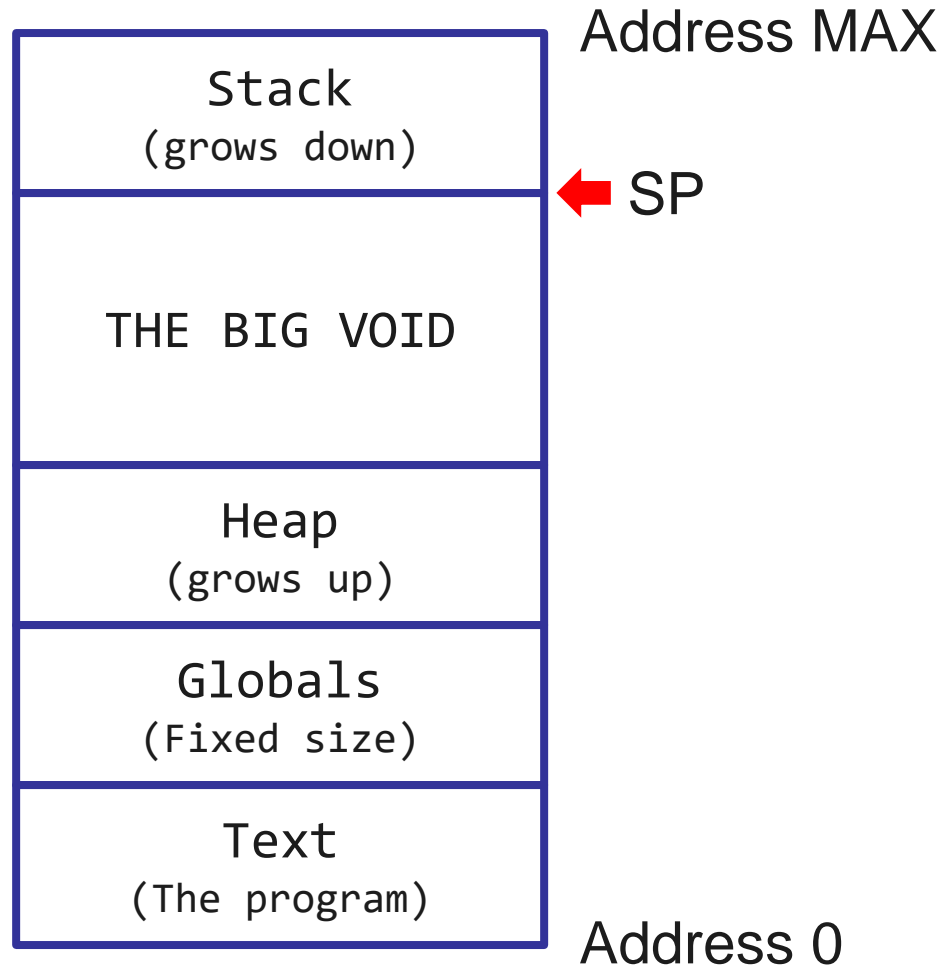
# Memory layout

Broken into 5 parts.



# Memory layout

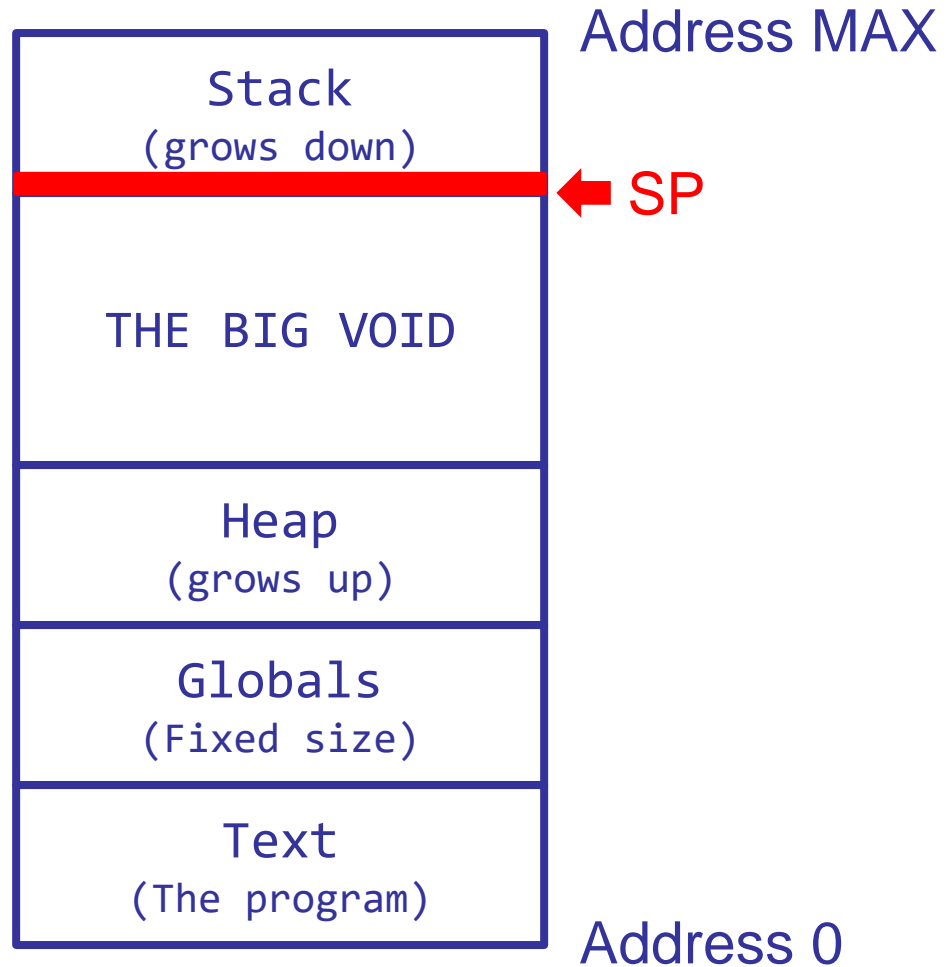
1. C++ programs use a *stack pointer*, usually a hardware register in the CPU, which points to the bottom of the stack.
2. The stack grows *downward* to lower addresses so that objects on the stack can be referenced in machine instructions as  $SP + offset$ .
3. When a function is called, *memory is allocated on the stack* by adjusting the stack pointer.





# Memory layout

1. When a procedure is called, the space allocated on the stack is called a *stack frame* or an *activation record*.
2. The stack frame holds *parameters* being passed, the *return address* and any *local variables* the function needs.




# Call stack example


```
int PlusOne( int x)
{
    return x + 1;
}

int PlusTwo( int x )
{
    return 1 + PlusOne( x );
}

int main( )
{
    int result = 0;
    result = PlusOne( 0 );
    result = PlusTwo( result );
    cout << result; // prints 3
}
```

## Memory image

result  main

SP 

# Call stack example

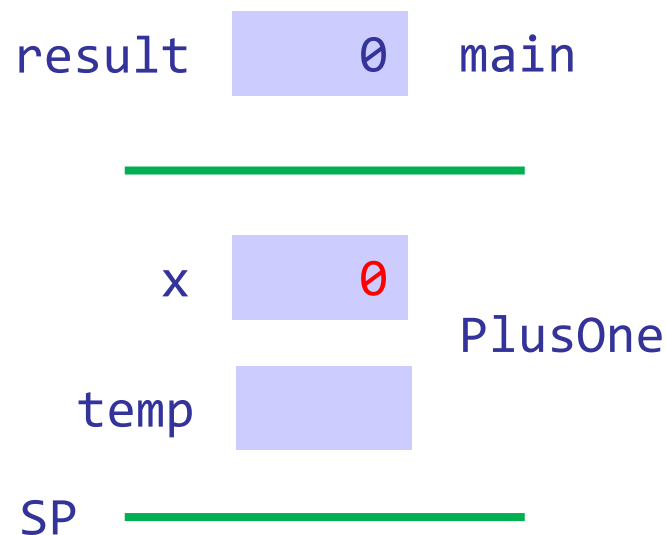
```
int PlusOne( int x)
{
    return x + 1;
}

int PlusTwo( int x )
{
    return 1 + PlusOne( x );
}

int main( )
{
    int result = 0;
    result = PlusOne( 0 );
    result = PlusTwo( result );
    cout << result; // prints 3
}
```



## Memory image



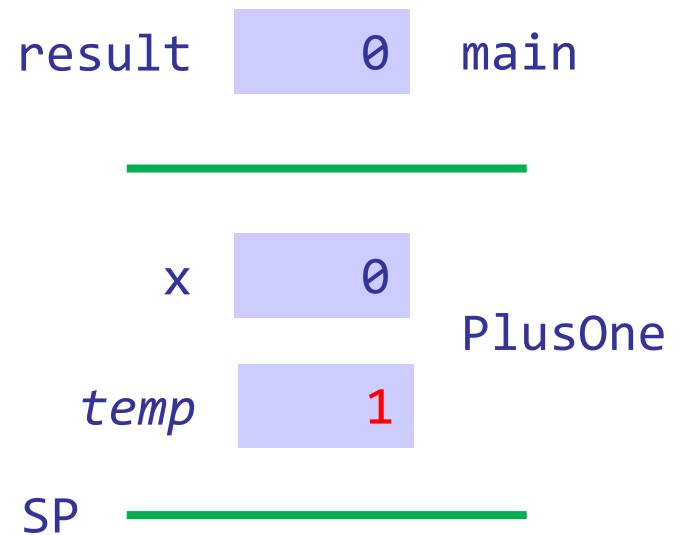
# Call stack example

```
int PlusOne( int x)
{
  return x + 1;
}

int PlusTwo( int x )
{
  return 1 + PlusOne( x );
}

int main( )
{
  int result = 0;
  result = PlusOne( 0 );
  result = PlusTwo( result );
  cout << result; // prints 3
}
```

## Memory image



# Call stack example

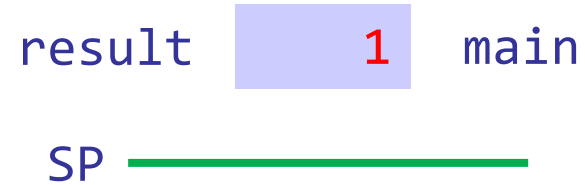
```
int PlusOne( int x)
{
    return x + 1;
}
```

```
int PlusTwo( int x )
{
    return 1 + PlusOne( x );
}
```

```
int main( )
{
    int result = 0;
    result = PlusOne( 0 );
    result = PlusTwo( result );
    cout << result; // prints 3
}
```



## Memory image



# Call stack example

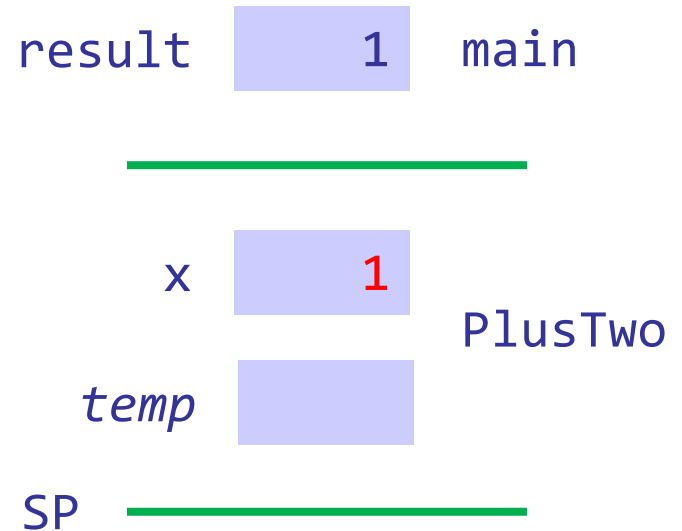
```
int PlusOne( int x)
{
    return x + 1;
}
```

```
int PlusTwo( int x )
{
    return 1 + PlusOne( x );
}
```

```
int main( )
{
    int result = 0;
    result = PlusOne( 0 );
    result = PlusTwo( result );
    cout << result; // prints 3
}
```



## Memory image



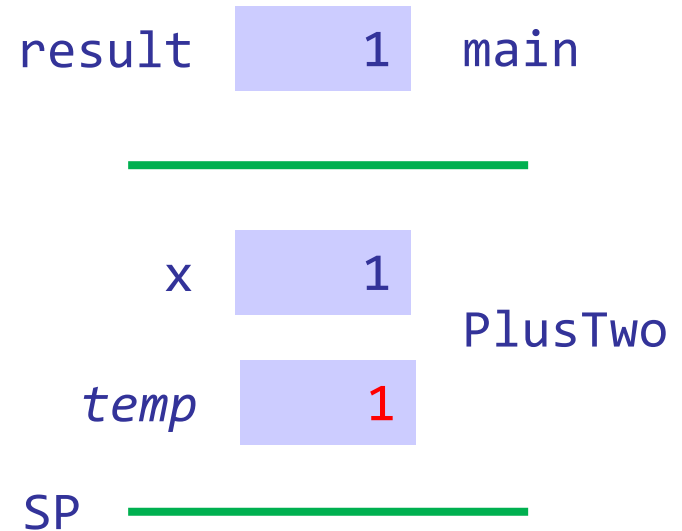
# Call stack example

```
int PlusOne( int x)
{
    return x + 1;
}
```

```
int PlusTwo( int x )
{
    return 1 + PlusOne( x );
}
```

```
int main( )
{
    int result = 0;
    result = PlusOne( 0 );
    result = PlusTwo( result );
    cout << result; // prints 3
}
```

## Memory image



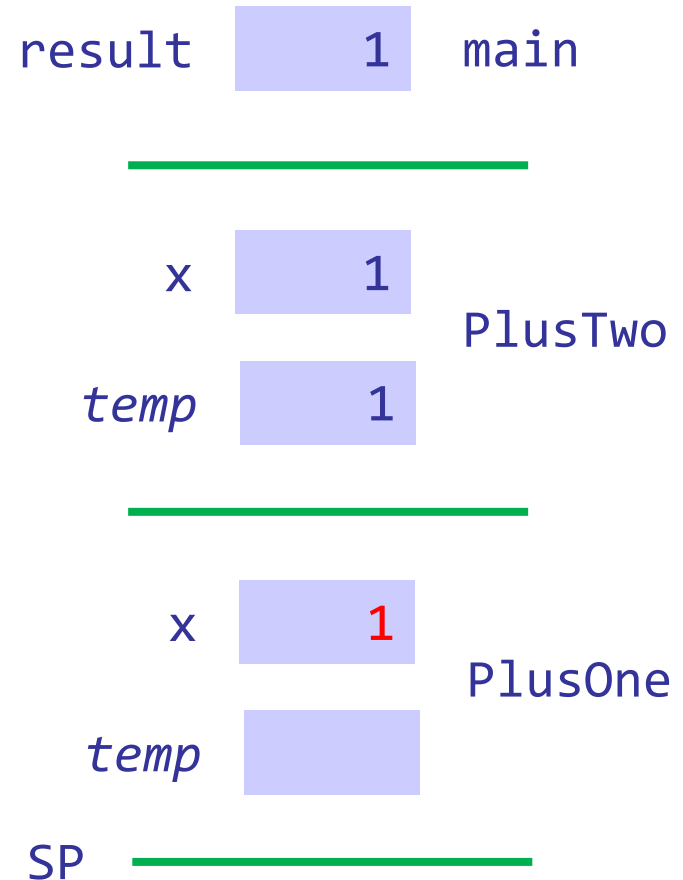
# Call stack example

```
int PlusOne( int x)
{
  return x + 1;
}
```

```
int PlusTwo( int x )
{
  return 1 + PlusOne( x );
}
```

```
int main( )
{
  int result = 0;
  result = PlusOne( 0 );
  result = PlusTwo( result );
  cout << result; // prints 3
}
```

## Memory image





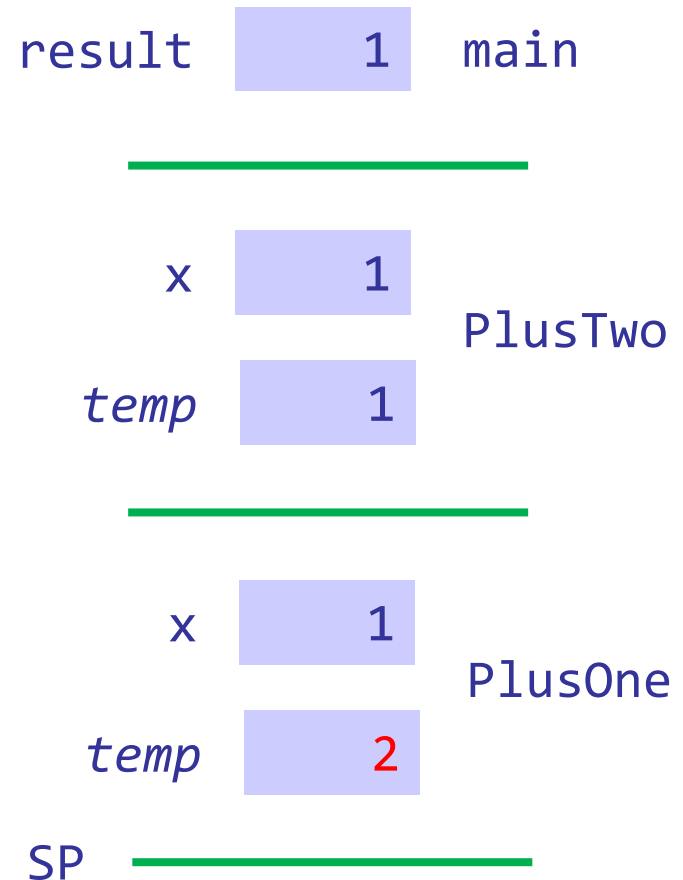
# Call stack example

```
int PlusOne( int x)
{
  return x + 1;
}

int PlusTwo( int x )
{
  return 1 + PlusOne( x );
}

int main( )
{
  int result = 0;
  result = PlusOne( 0 );
  result = PlusTwo( result );
  cout << result; // prints 3
}
```

## Memory image



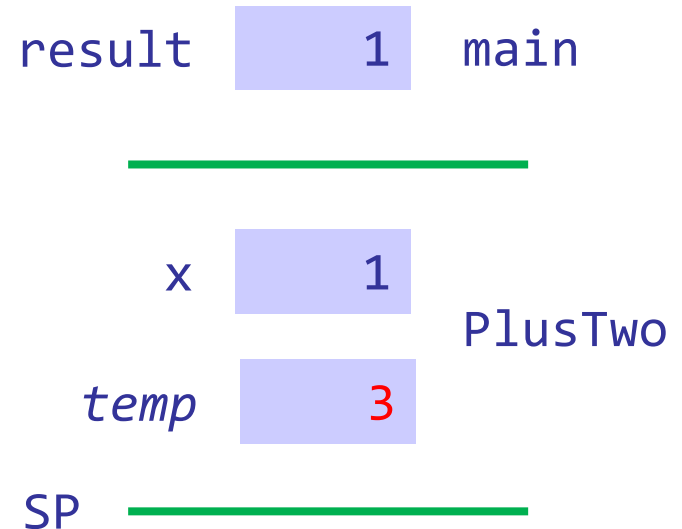
# Call stack example

```
int PlusOne( int x)
{
    return x + 1;
}
```

```
int PlusTwo( int x )
{
    return 1 + PlusOne( x );
}
```

```
int main( )
{
    int result = 0;
    result = PlusOne( 0 );
    result = PlusTwo( result );
    cout << result; // prints 3
}
```

## Memory image



# Call stack example

```
int PlusOne( int x)
{
    return x + 1;
}

int PlusTwo( int x )
{
    return 1 + PlusOne( x );
}

int main( )
{
    int result = 0;
    result = PlusOne( 0 );
    result = PlusTwo( result );
    cout << result; // prints 3
}
```

## Memory image

result 3 main

SP



# What if you could have lots of stacks?

You could have lots of procedures in your code running *concurrently*, switching rapidly between them.

If your PC has multiple cores (multiple CPUs) they could all be running *simultaneously*.

That's called a thread.

# What if you could have lots of stacks?

To create a thread, you give the OS a pointer to the procedure it should run and a void \* to an argument.

The procedure runs in the new thread and when it returns, the thread exits.

```
// Linux multi-threaded hello world program.
```

```
#include <stdlib.h>
#include <pthread.h>
#include <iostream>
using namespace std;

void *Hello( void *p )
{
    cout << "Hello from the child!" <<
        endl;
}

int main( int argc, char **argv )
{
    cout << "Starting child" << endl;
    pthread_t child;

    pthread_create( &child, nullptr, Hello, nullptr );

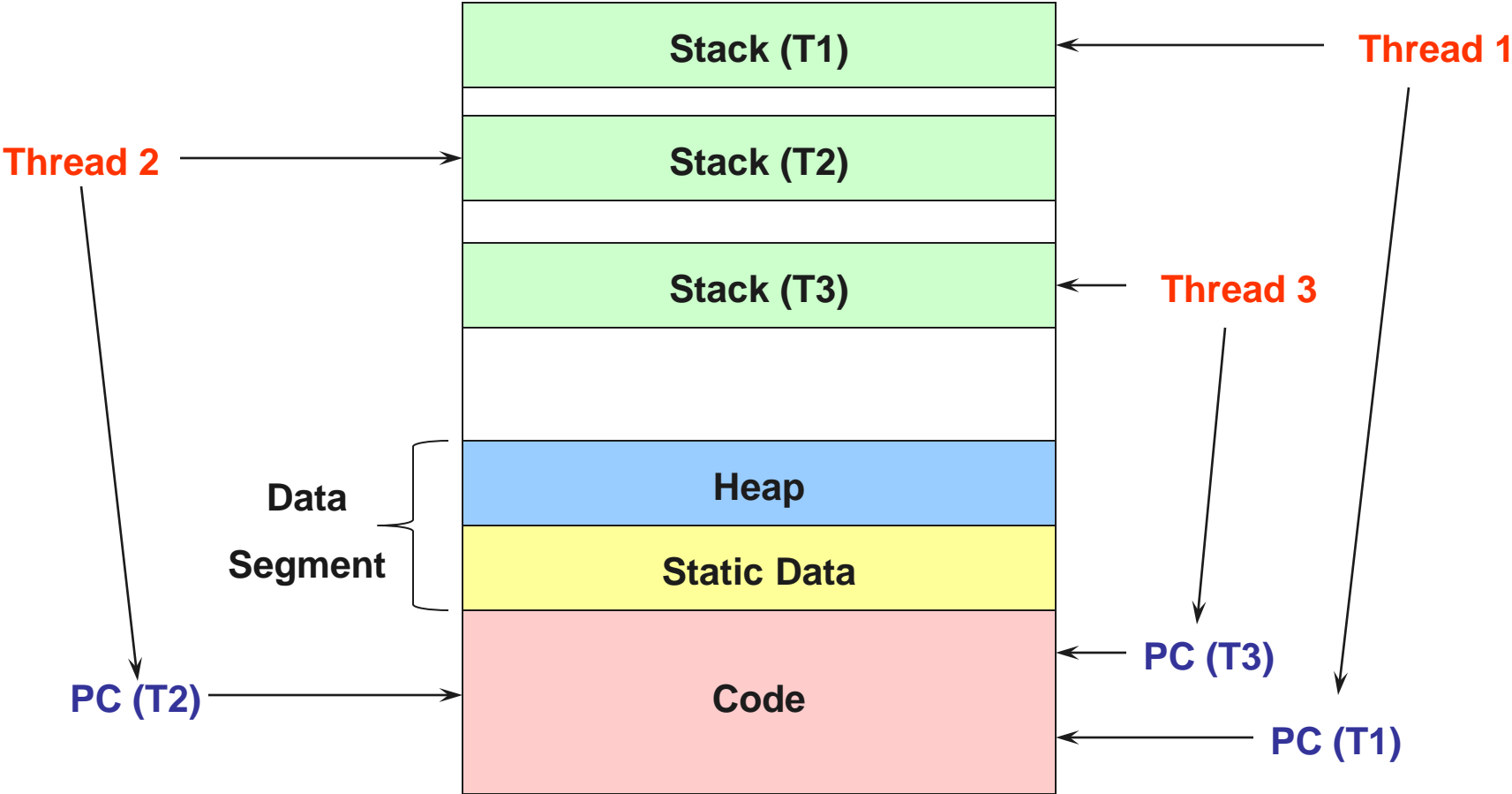
    cout << "Waiting for child" << endl;
    pthread_join( child, NULL );

    cout << "Child has exited" << endl;
}
```

```
80 % g++ LinuxHelloMT.cpp -o LinuxHelloMT
81 % !$
LinuxHelloMT
Starting child
Waiting for child
Hello from the child!
Child has exited
82 %
```

Could I have gotten other outputs?

# Process Address Space



# The Process Model

Windows, Linux and other operating systems have similar models.

Each process is protected from other processes.

It owns resources:

1. Memory image containing instructions and data.
2. Open handles to files and other system objects.

It also has some “state” information, including:

1. Current directory.
2. Environment variables passed as `envp` to `main()`.
3. One or more *threads* of execution.



# Threads

A thread is an execution path through your code that starts with a call to a procedure that runs independently of any others.

This is how the C++ runtime starts and then calls `main()`.

A thread's "state" consists of:

1. An instruction pointer aka a program counter.
2. A stack and a stack pointer.
3. A general register set.
4. Its scheduling priority and other attributes.
5. Any locks it might hold. (More about this soon.)

Each thread shares the rest of the process state, the memory, open file handles, etc., with every other thread.

If one thread changes a variable in memory, it affects all of them.

# Threads and processes

## Per thread

1. An instruction pointer
2. A stack
3. A register set
4. Its scheduling priority
5. Any locks it might hold

## Shared process state

1. Memory
2. Open file handles
3. Current directory
4. Environment variables

# Threads

Within a running process, there can be lots of running threads.

The operating system switches between them really quickly, letting one thread run for a while or until it has to wait for i/o, then switches to another to let it run.

Each thread shares the rest of the process state, the memory, open file handles, etc., with every other thread.

If one thread changes a variable in memory or opens or closes a file, it affects all of them.

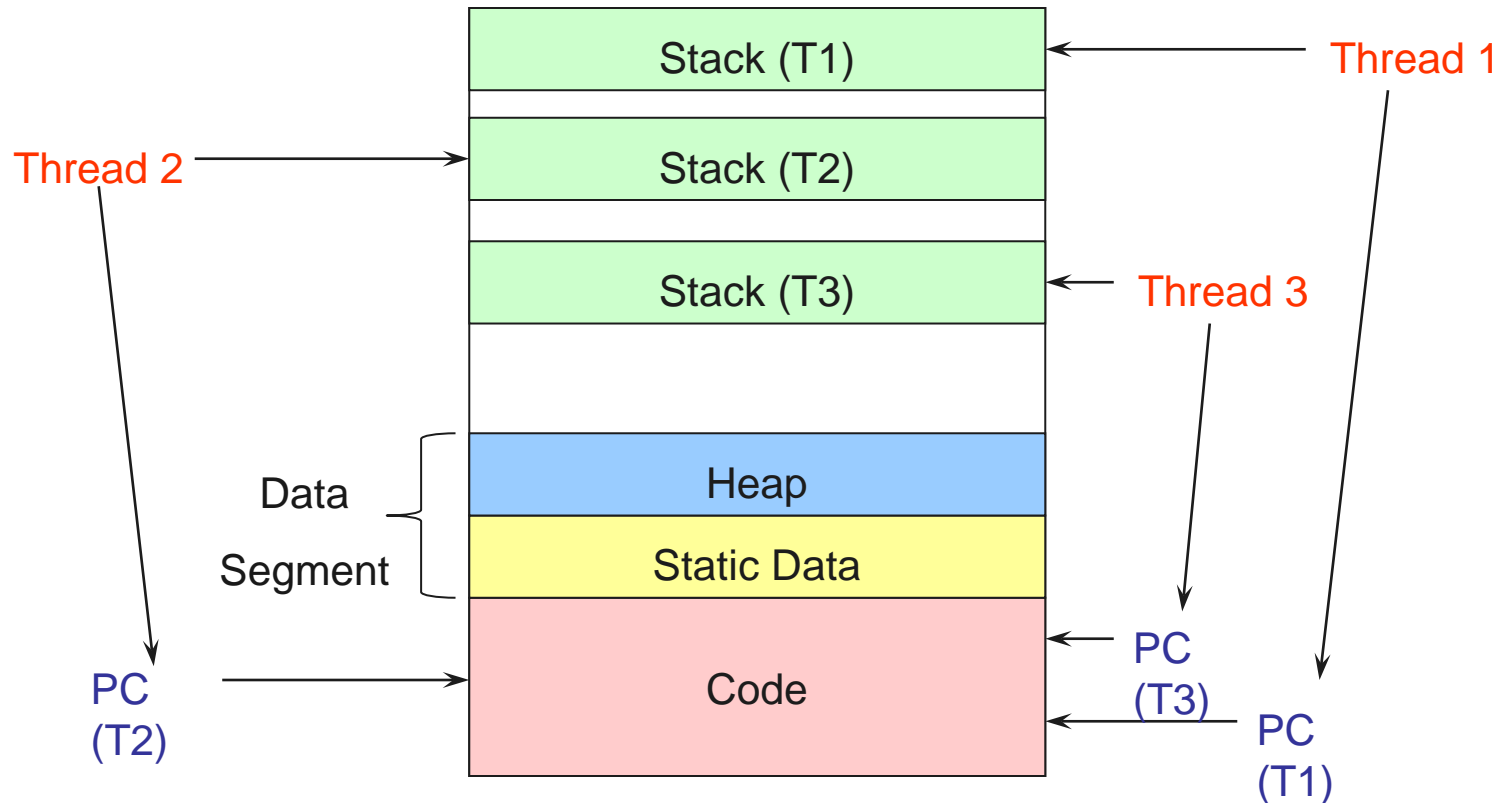
What problems might this cause?

# Multiple Threads

Can have lots of threads in a single address space changing things.

Sometimes they have interact and cooperate. (When?)

Sometimes they can work independently. (When?)



# Major themes over the next month

**Threads** that can be scheduled concurrently.

How do multiple threads cooperate to accomplish a task?

How do multiple threads share a limited number of CPUs?

Does having multiple CPUs add new problems?

**Address spaces** for managing state information.

How do address spaces share single physical memory?

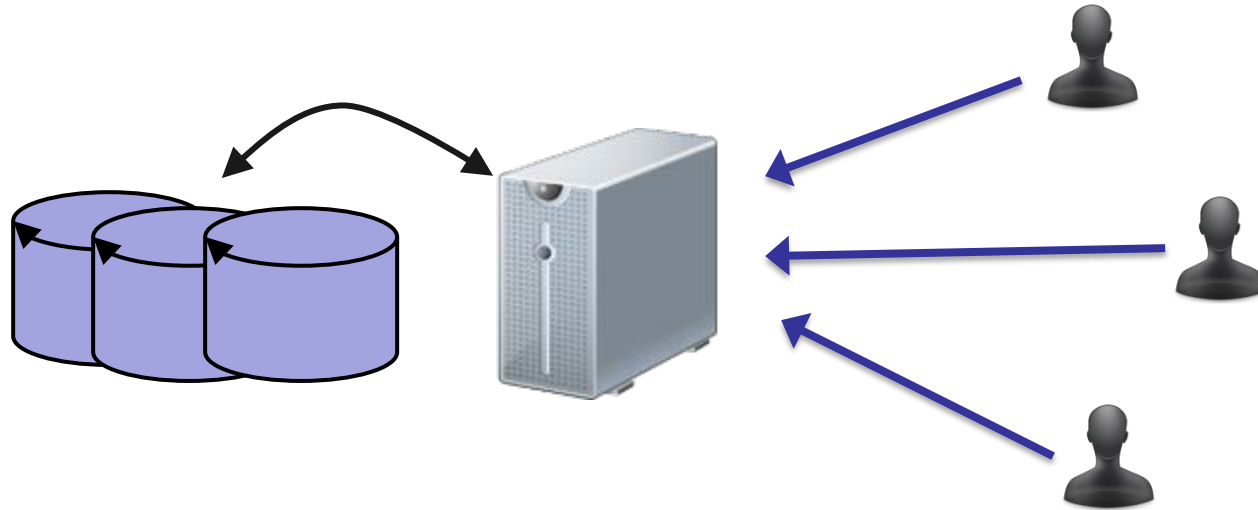
How is it done efficiently, flexibly and safely?

# Why can threads do for us?

## Example: Web server

Receives multiple simultaneous requests

Reads web pages from disk to satisfy each request



# Option 1: Handle one request at a time

Request 1 arrives

Server reads in request 1

Server starts disk I/O for request 1

Request 2 arrives

Disk I/O for request 1 finishes

Server responds to request 1

Server reads in request 2



Pros and cons?

Easy to program, but slow

Can't overlap disk requests with computation

Can't overlap either with network sends and receives

## Option 2: Overlap the i/o

Many early OS's supported "overlapped i/o". You only got single thread but you could start an i/o without having to wait for it to complete.

Request 1 arrives

Server reads in request 1

Server starts disk I/O for request 1

Request 2 arrives

Server reads in request 2

Server starts disk I/O for request 2

Disk I/O for request 1 completes

Server responds to request 1



Fast, but **hard to program**

Why?

Web server must remember

What requests are being served, and what stage they're in

What disk I/Os are outstanding (and which requests they belong to)

**Lots of extra state!**



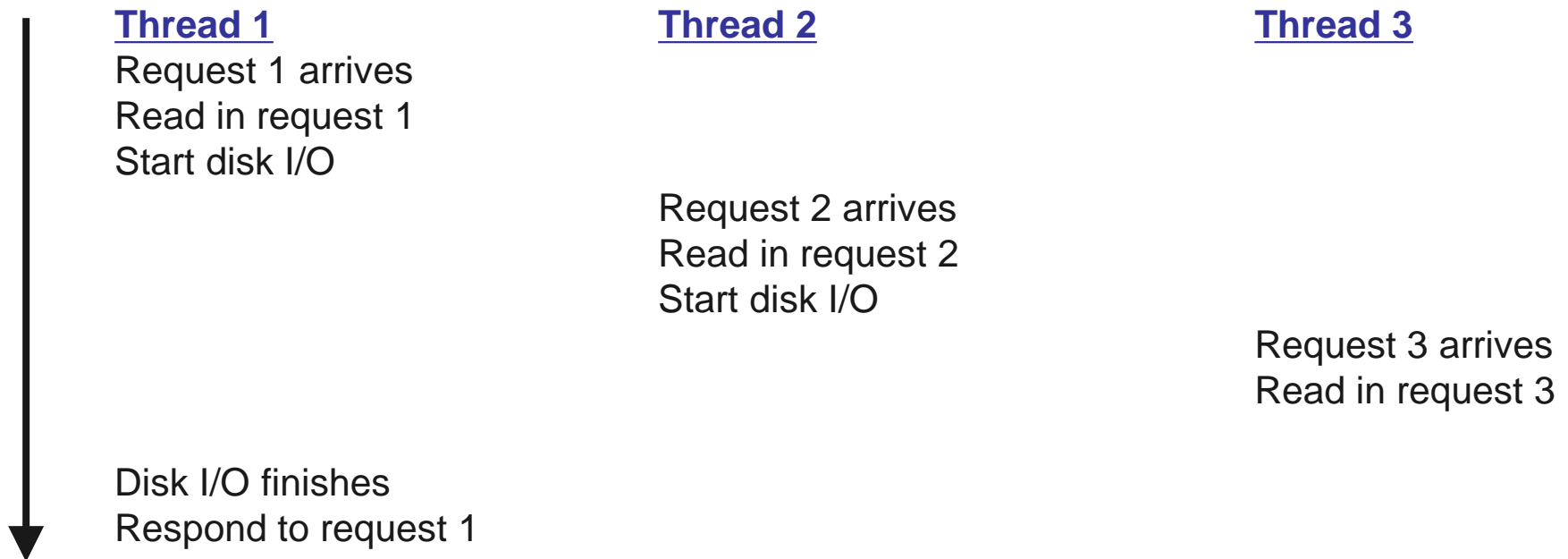
# Option 3: Multiple threads

Create a new thread for each request

Thread issues disk (or n/w) I/O, then waits for it to finish

Though thread is **blocked on I/O**, **other threads can run**

**Where is the state of each request stored?**



# Benefits of Threads

## Thread manager takes care of CPU sharing

When one thread blocks on an i/o, other threads can progress.

Each thread has its own stack and can do its own thing.

## Applications get a simpler programming model

The illusion of a dedicated CPU per thread.

## Downsides compared to event-driven model?

Overhead involved in scheduling and context-switching.

Sharing of data becomes more difficult.

Race conditions and programming hazards.

# Reminders

Sign up for GitHub and Piazza

Upload photo

Started putting together project group?

Group declaration due May 22.

Speak up when something is unclear

# When are threads useful?

Multiple things happening at once

Usually some slow resource

Network, disk, user, ...

Examples:

Controlling a physical system (e.g., airplane controller)

Bank ATM server

Window system

Parallel programming

# Ideal Scenario

Split computation into threads

Threads run **independent** of each other

Divide and conquer works best if divided parts are independent

**How practical is thread independence?**

# Dependence between threads

## Example 1: Microsoft Word

One thread formats document

Another thread spell checks document

## Example 2: Desktop computer

One thread is running Chrome

Another thread is compiling your EECS 482 project

Two types of sharing: **app resource** or **H/W**

**Example of non-interacting threads?**

# Cooperating threads

How can multiple threads cooperate on a single task?

Example: Ticketmaster's webserver

Assume each thread has a dedicated processor

Problem:

Ordering of events across threads is **non-deterministic**

Speed of each processor is unpredictable



**Consequences:**

Many possible global ordering of events

Some may produce incorrect results

# Non-deterministic ordering → Non-deterministic results

Printing example

Thread 1  
Print ABC

Thread 2  
Print 123

Possible outputs?

20 outputs: ABC123, AB1C23, AB12C3, AB123C, A1BC23, A12BC3, A123BC, 1ABC23, 1A2BC3, ...

Impossible outputs?

ABC321

Ordering within a thread is sequential.

Many ways to merge per-thread order into a global order

What's being shared between these threads?



# Non-deterministic ordering → Non-deterministic results

Arithmetic example (y is initially 10)

What's being shared between these threads?

Possible results?

If A runs first:  $x = 11$  and  $y = 20$

If B runs first:  $x = 21$  and  $y = 20$

Thread A

$x = y + 1$

Thread B

$y = y * 2$

# Non-deterministic ordering → Non-deterministic results

Another example

**Possible results?**

$x = 1$  or  $-1$

**Impossible results?**

$x = 0$

Thread A

$x = 0$

$x = 1$

Thread B

$x = 0$

$x = -1$

# Non-deterministic ordering → Non-deterministic results

A final example

**Possible results?**

$x = 0, 1$  or  $-1$

**Impossible results?**

$x = 2, -2$

Thread A

$x = 0$

$x++$

Thread B

$x = 0$

$x--$

# Atomic operations

Before we think about how make threads cooperate, we need for some operations to be **atomic**.

Indivisible, i.e., it either happens in its entirety or not at all.

No events from other threads can occur in between.

Print example:

What if each print statement were atomic?

What if printing a single character were not atomic?

Most computers

Memory load and store are atomic.

Many other instructions are not atomic.

Example: double-precision floating point.

Need an atomic operation to build a bigger atomic operation.

# Next Time ...

## Synchronization