# CSE150
# Operating Systems
# Lecture 4

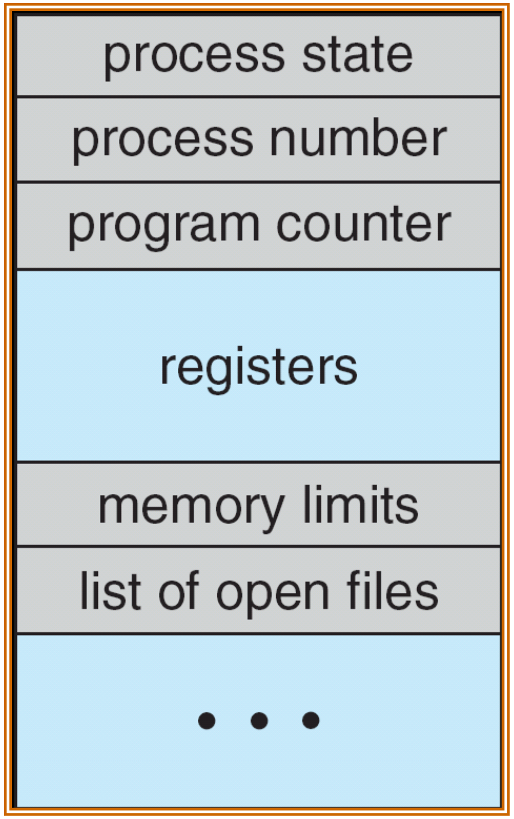
## Processes, Threads and Address Spaces (wrap up)
## Concurrency and Thread Dispatching

# Recall: Processes, Threads, Address Spaces and Dual-mode

- Processes have two parts
  - Threads (Concurrency)
  - Address Spaces (Protection)
- Concurrency accomplished by multiplexing CPU Time:
  - Unloading current thread (PC, registers)
  - Loading new thread (PC, registers)
  - Such context switching may be voluntary (`yield()`, I/O operations) or involuntary (timer, other interrupts)
- Protection accomplished restricting access:
  - Memory mapping isolates processes from each other
  - Dual-mode for isolating I/O, other resources

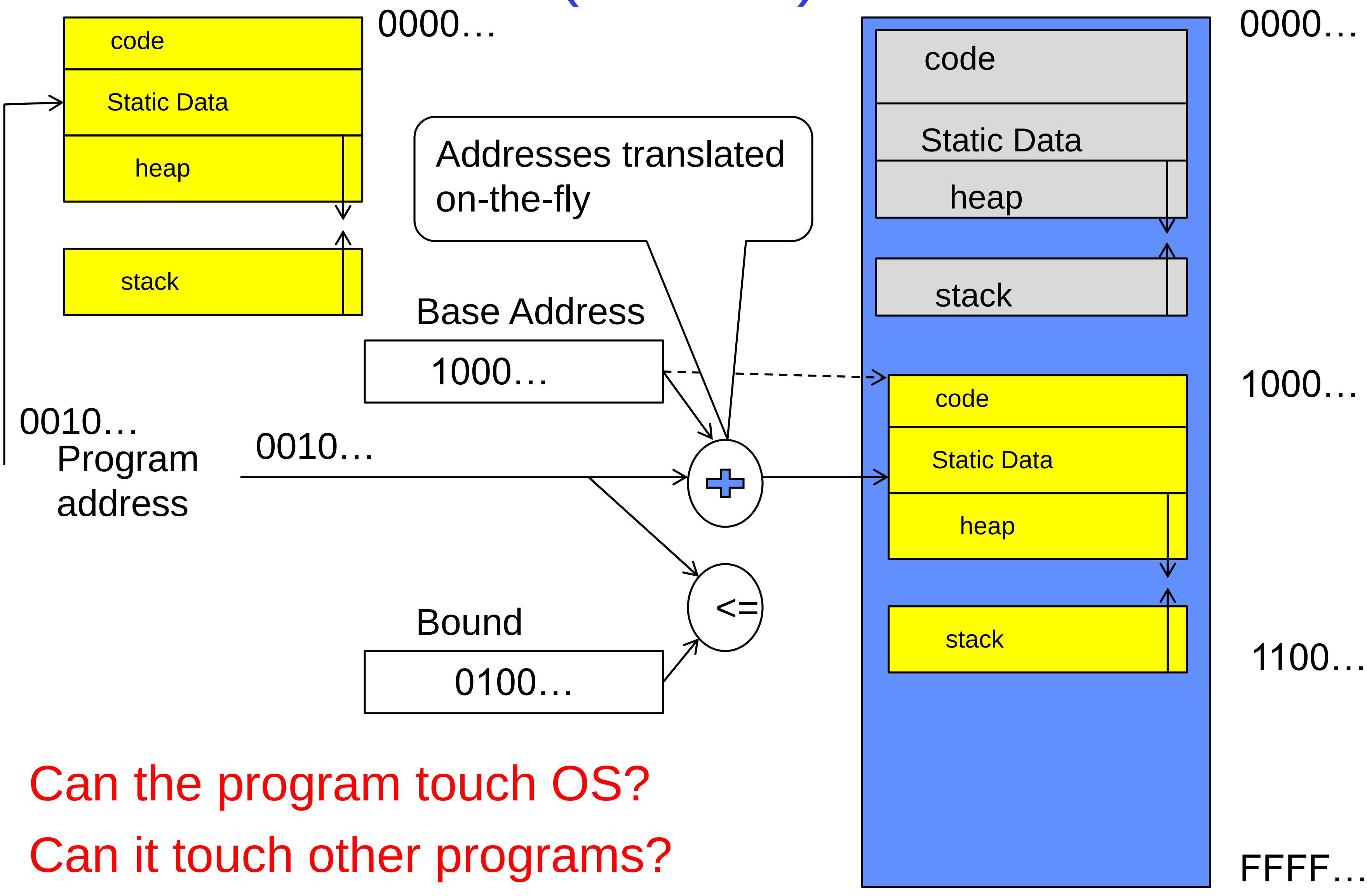
# How do we multiplex processes? (Review)

- The current state of process held in a process control block (PCB):
  - This is a “snapshot” of the execution and protection environment
  - Only one PCB active at a time
- Give out CPU time to different processes (Scheduling):
  - Only one process “running” at a time
  - Give more time to important processes
- Give pieces of resources to different processes (Protection):
  - Controlled access to non-CPU resources
  - Sample mechanisms:
    - » Memory Mapping: Give each process their own address space
    - » Kernel/User duality: Arbitrary multiplexing of I/O through system calls



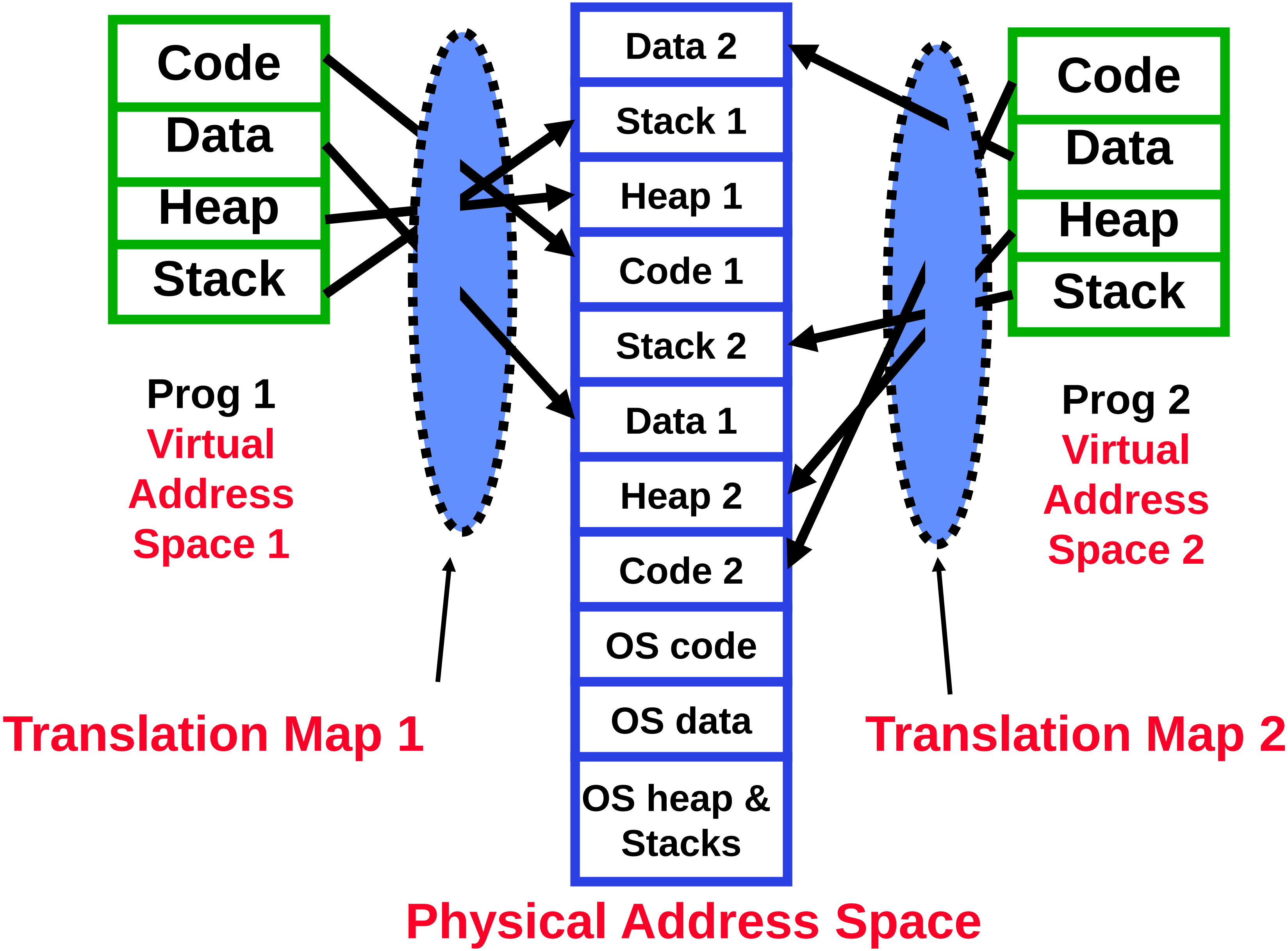

**Process Control Block**

# Address translation with Base and Bound (Review)

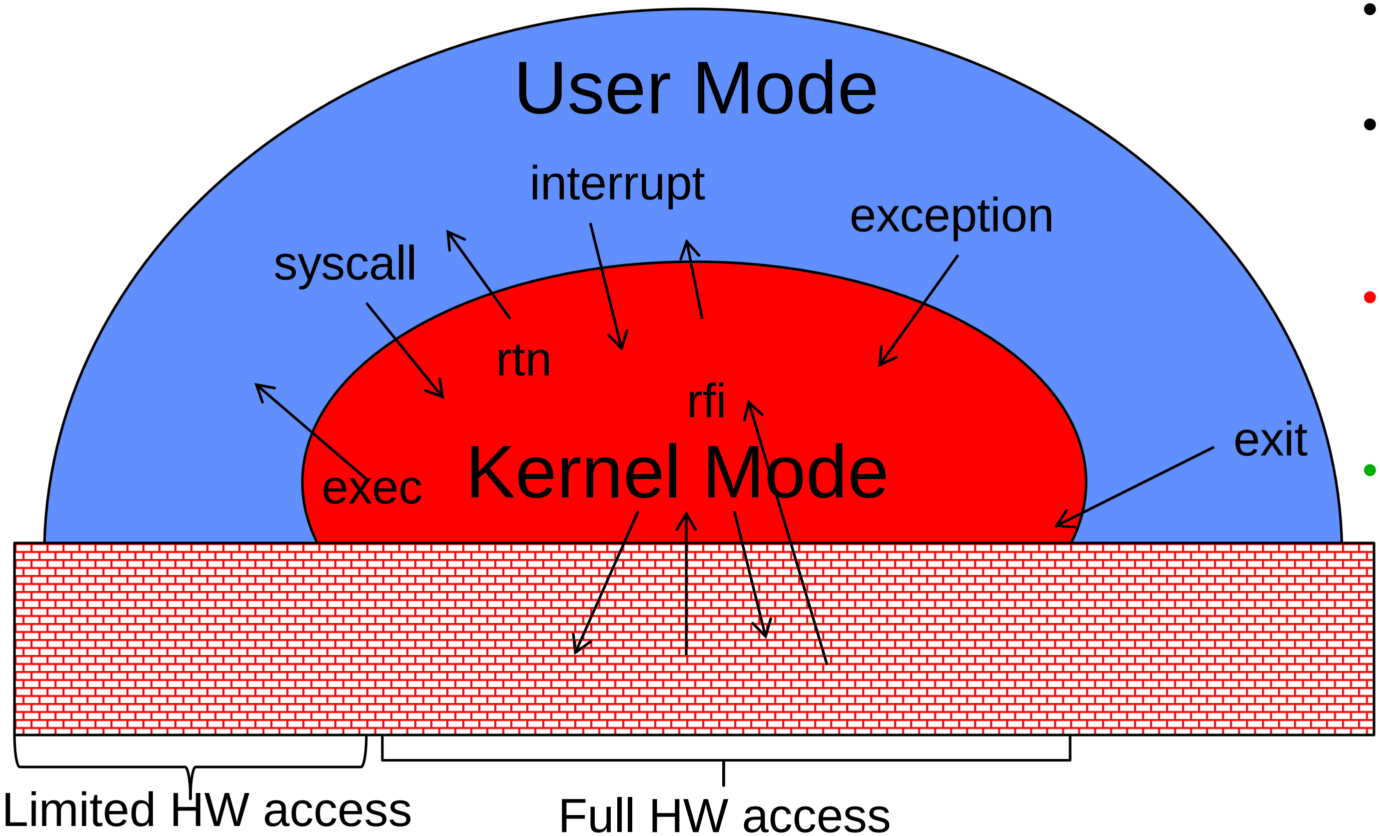



- Can the program touch OS?
- Can it touch other programs?

# Providing Illusion of Separate Address Space by Loading new Translation Map on Switch (Review)



# User/Kernel Mode (Review)



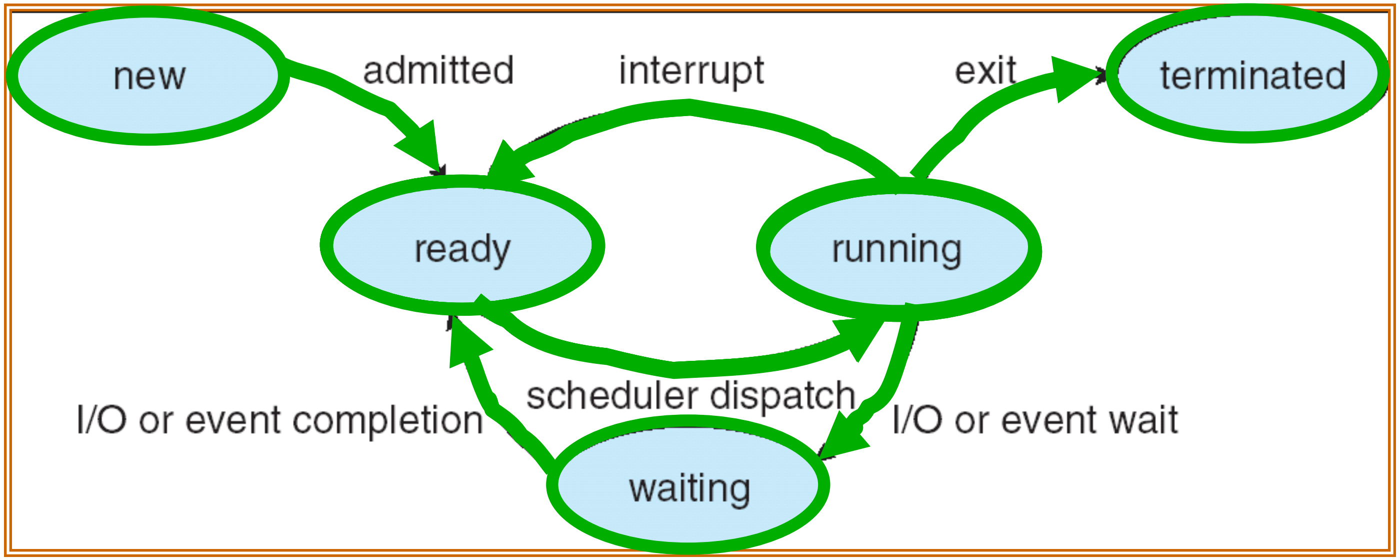

- A user/system mode bit
- Certain operations only permitted in system/kernel mode
- User→Kernel: *sets* system mode AND saves the user PC
- Kernel→User: clears system mode AND restores appropriate user PC

- 3 types of mode transfer
  - Syscall: Process requests a system service (e.g., exit)
  - Interrupt: External asynchronous event triggers context switch (e.g., Timer, I/O device)
  - Exception: Internal synchronous event in process triggers context switch, (e.g., segmentation fault, Divide by zero)

# Today

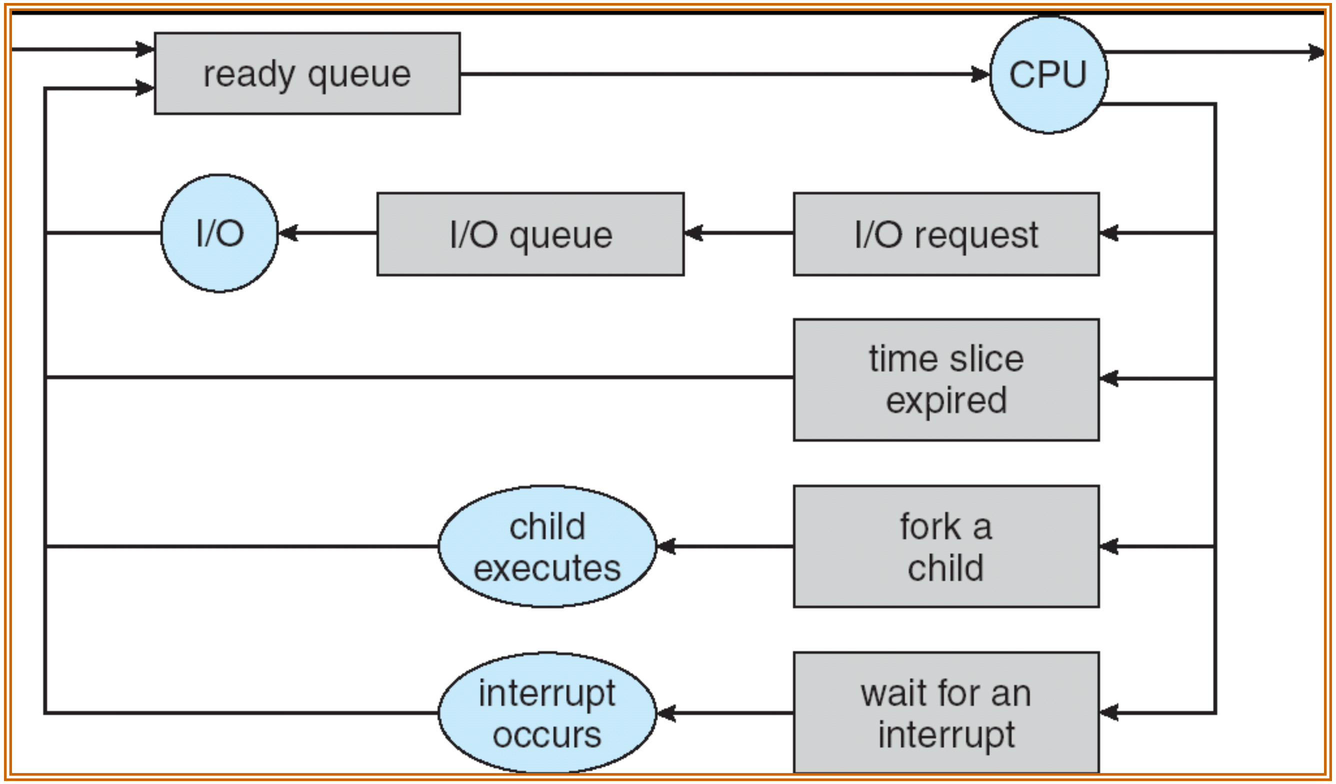
- Wrap up Basic OS Concepts
- Further Understanding Threads
  - Dispatching
  - Beginnings of Thread Scheduling

# Lifecycle of a Process




- As a process executes, it changes state:
  - new: The process is being created
  - ready: The process is waiting to run
  - running: Instructions are being executed
  - waiting: Process waiting for some event to occur
  - terminated: The process has finished execution

# Process Scheduling

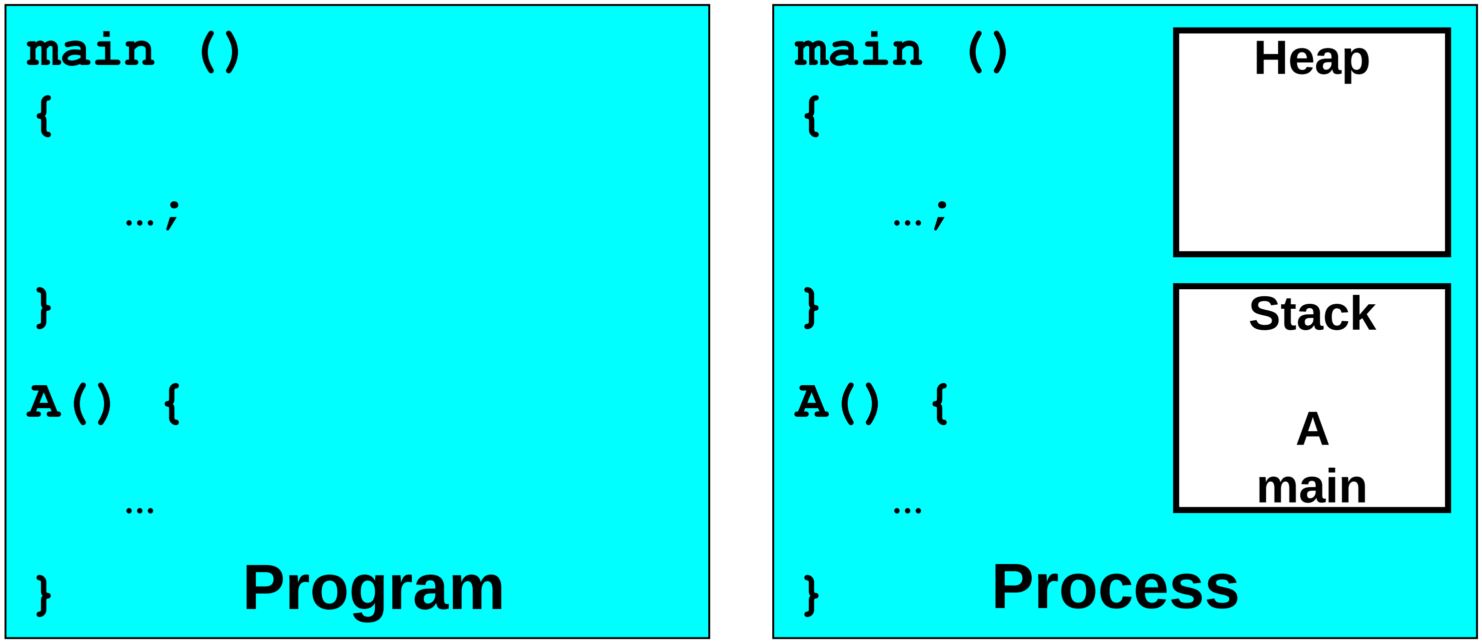



- PCBs move from queue to queue as they change state
  - Decisions about which order to remove from queues are Scheduling decisions
  - Many algorithms possible (few weeks from now)

# What does it take to create a process?

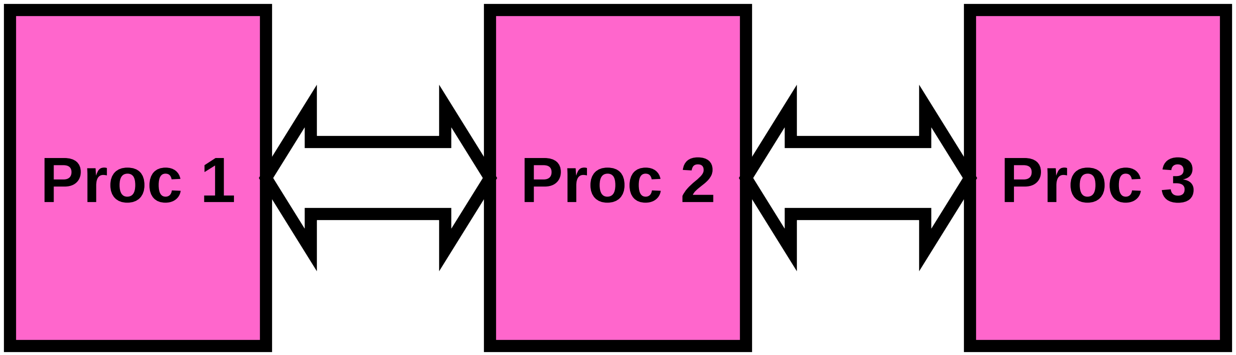
- Must construct new PCB
  - Inexpensive
- Must set up new page tables for address space
  - More expensive
- Copy data from parent process? (Unix `fork()` )
  - Semantics of Unix `fork()` are that the child process gets a complete copy of the parent memory and I/O state
  - Originally *very* expensive
  - Much less expensive with "copy on write"
- Copy I/O state (file handles, etc)
  - Medium expense

# Process =? Program



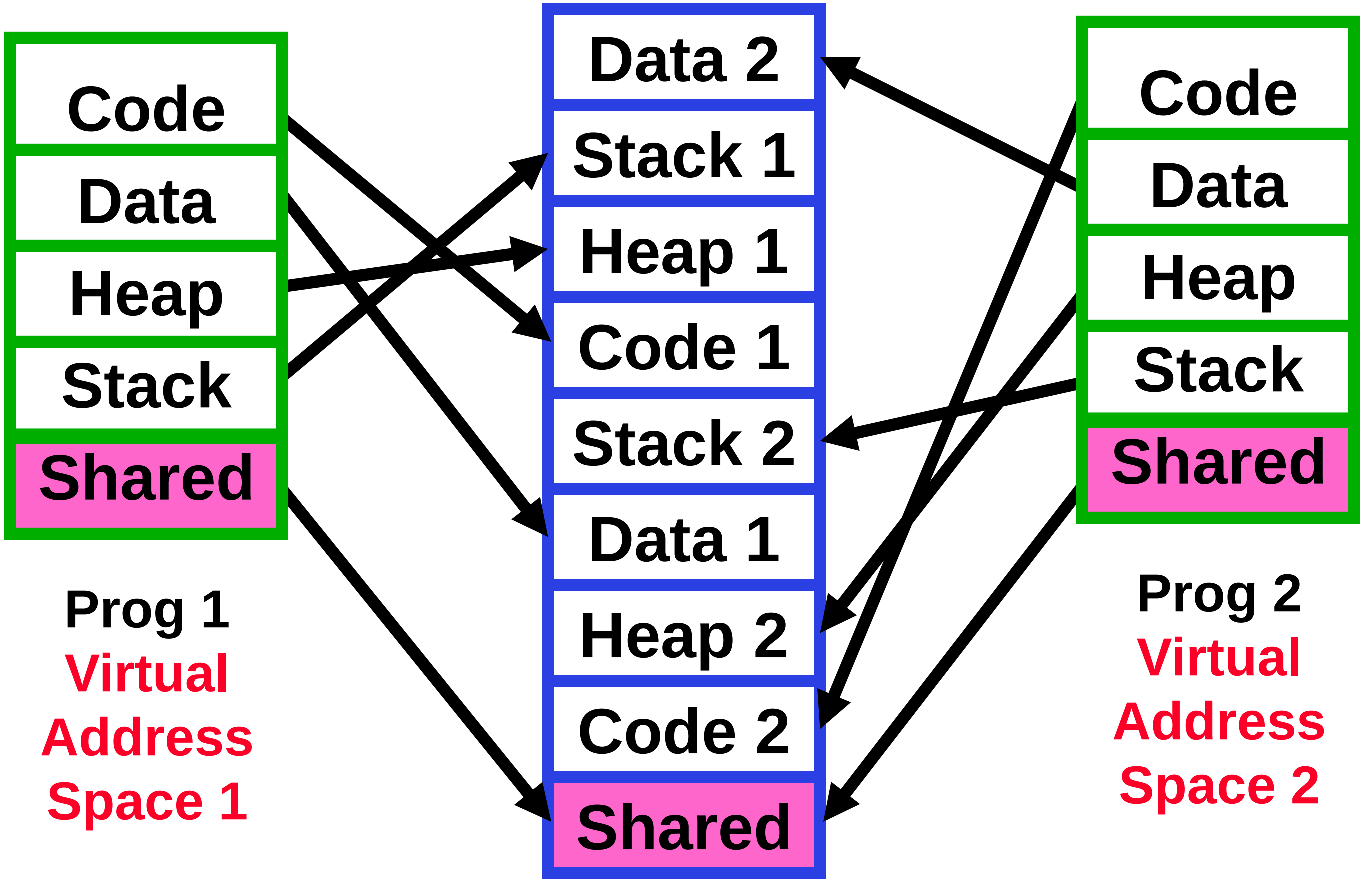

- Less to a process than a program:
  - A program can invoke more than one process
- More to a process than just a program:
  - Program is just part of the process state
  - I run Sublime on lectures.txt, you run it on homework.java – Same program, different processes

# Multiple Processes Collaborate on a Task




- Need communication mechanism:
  - Why? Separate Address Spaces Isolates Processes
  - Shared-Memory Mapping
    - » Accomplished by mapping addresses to common DRAM
    - » Read and Write through memory
  - Message Passing
    - » `send()` and `receive()` messages
    - » Works across network

# Shared Memory Communication




- Communication occurs by “simply” reading/writing to shared address page
  - Really low overhead communication
  - Introduces complex synchronization problems

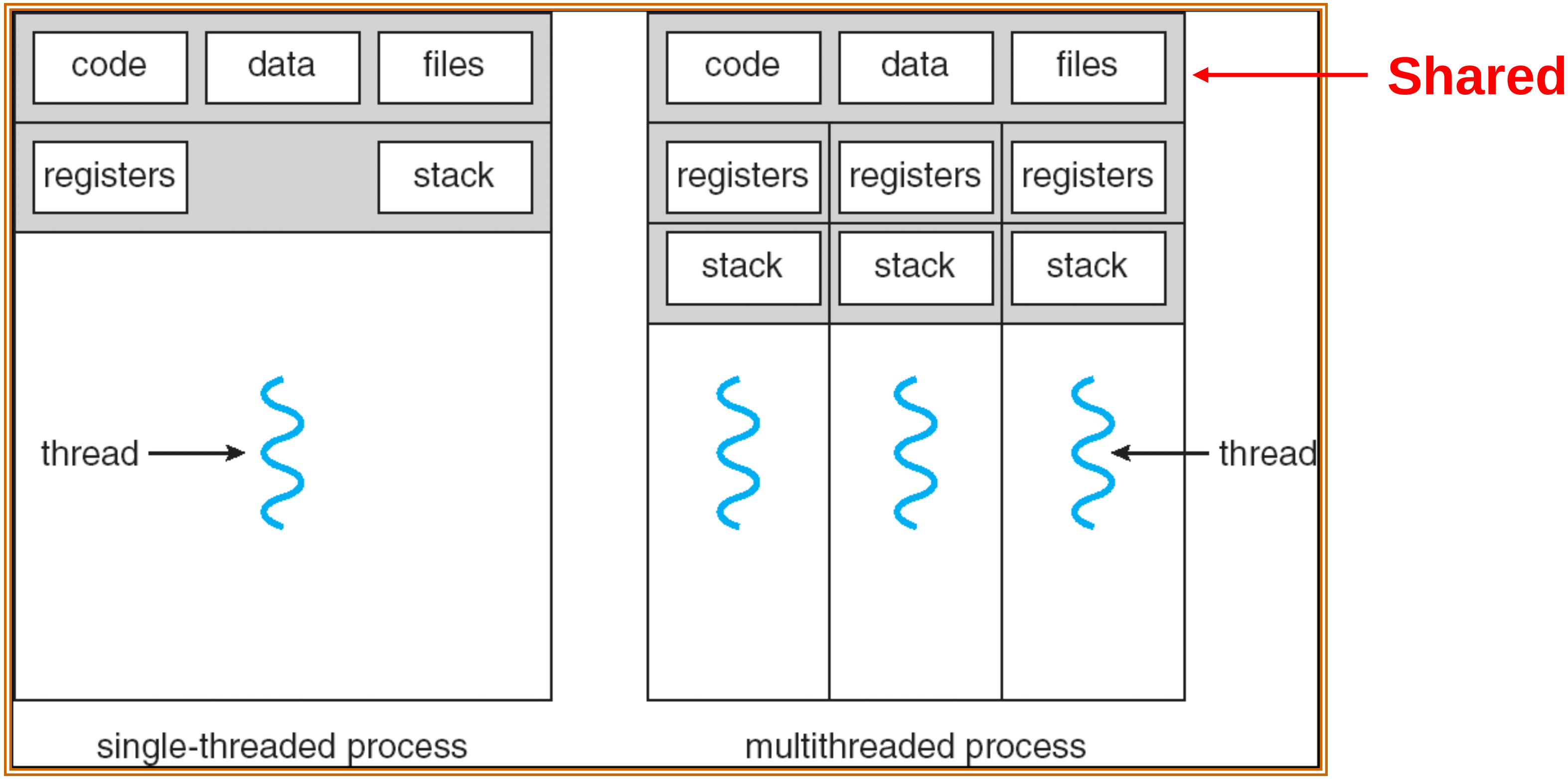
# Inter-process Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - `send(`*`message`*`)` – message size fixed or variable
  - `receive(`*`message`*`)`
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation of communication link involves
  - physical aspects (e.g., shared memory, hardware bus, systcall/trap in Chapter 17)
  - logical properties (direct/indirect, symmetric/asymmetric, etc.)

# Modern "Lightweight" Process with Threads

- Thread: *a sequential execution stream within process* (Sometimes called a "Lightweight process")
  - Process still contains a single Address Space
  - No protection between threads
- Multithreading: *a single program made up of a number of different concurrent activities*
  - Sometimes called multitasking, as in Ada…
- Why separate the concept of a thread from that of a process?
  - Discuss the "thread" part of a process (concurrency)
  - Separate from the "address space" (Protection)
  - Heavyweight Process $\equiv$ Process with one thread

# Single and Multithreaded Processes




- Threads encapsulate concurrency: "Active" component
- Address spaces encapsulate protection: "Passive" part
  – Keeps buggy program from trashing the system
- Why have multiple threads per address space?

# Examples of multithreaded programs

- Embedded systems
  - Elevators, Planes, Medical systems, Wristwatches
  - Single Program, concurrent operations
- Most modern OS kernels
  - Internally concurrent because have to deal with concurrent requests by multiple users
  - But no protection needed within kernel
- Database Servers
  - Access to shared data by many concurrent users
  - Also background utility processing must be done

# Examples of multithreaded programs (con't)

- Network Servers
  - Concurrent requests from network
  - Again, single program, multiple concurrent operations
  - File server, Web server, and airline reservation systems
- Parallel Programming (More than one physical CPU)
  - Split program into multiple threads for parallelism
  - This is called Multiprocessing

# Why Processes & Threads?

**Goals:**

- **Multiprogramming:** Run multiple applications concurrently
- **Protection:** Don't want a bad application to crash system!

**Solution:**

**Process**: unit of execution and allocation

- Virtual Machine abstraction: give process illusion it owns machine (i.e., CPU, Memory, and IO device multiplexing)

**Challenge:**

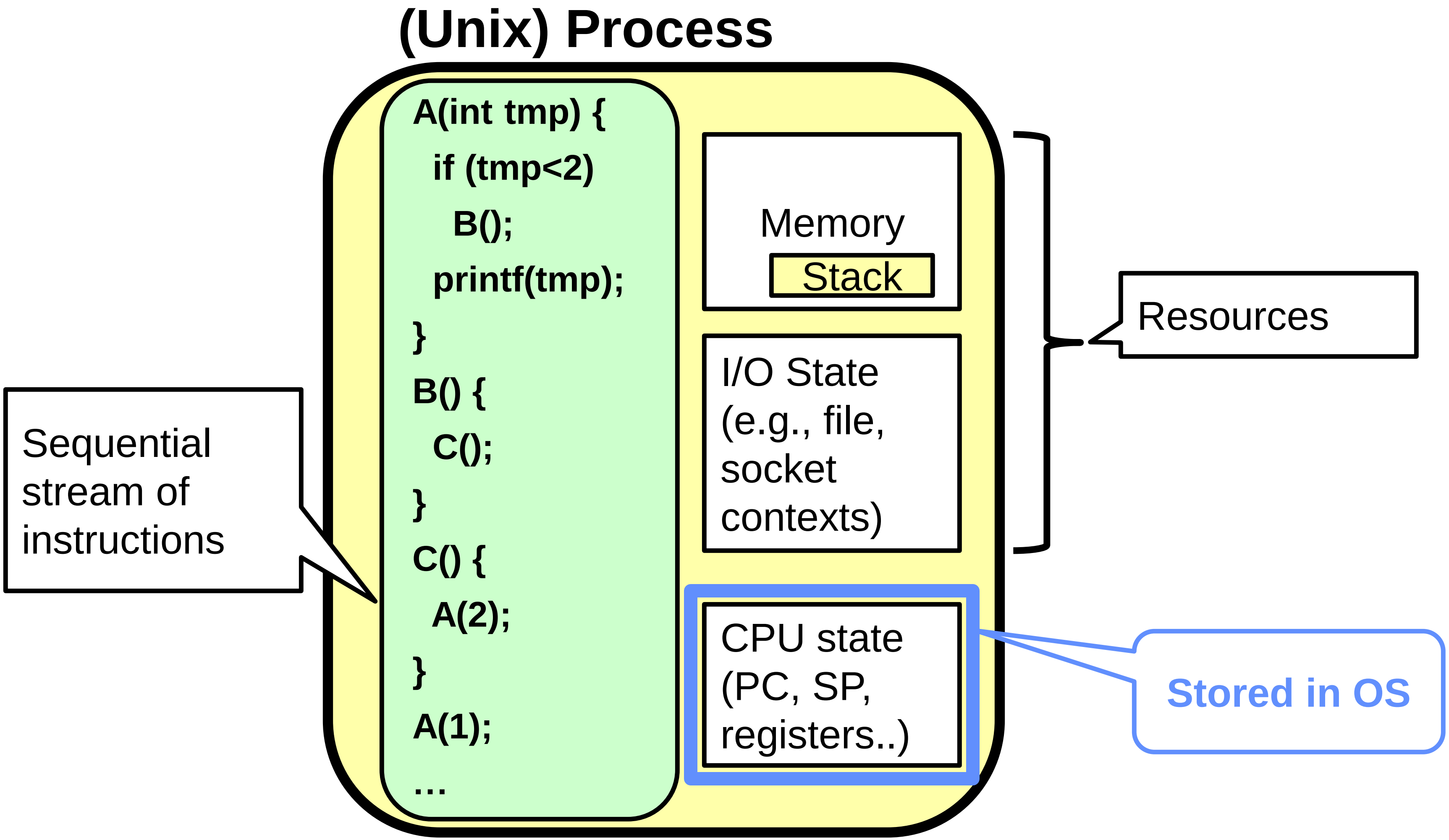
- Process creation & switching expensive
- Need concurrency within same app (e.g., web server)

**Solution:**

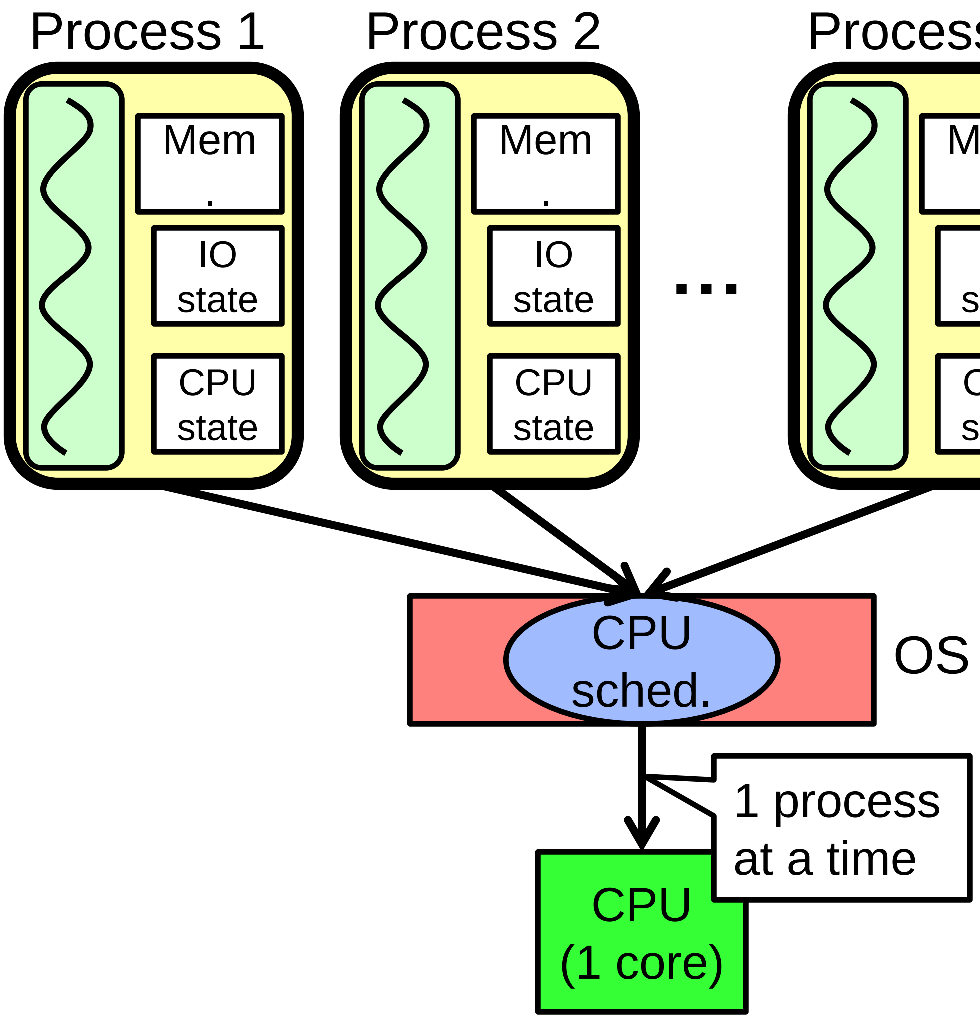
**Thread:** Decouple allocation and execution

- Run multiple threads within same process

# Putting it together: Process

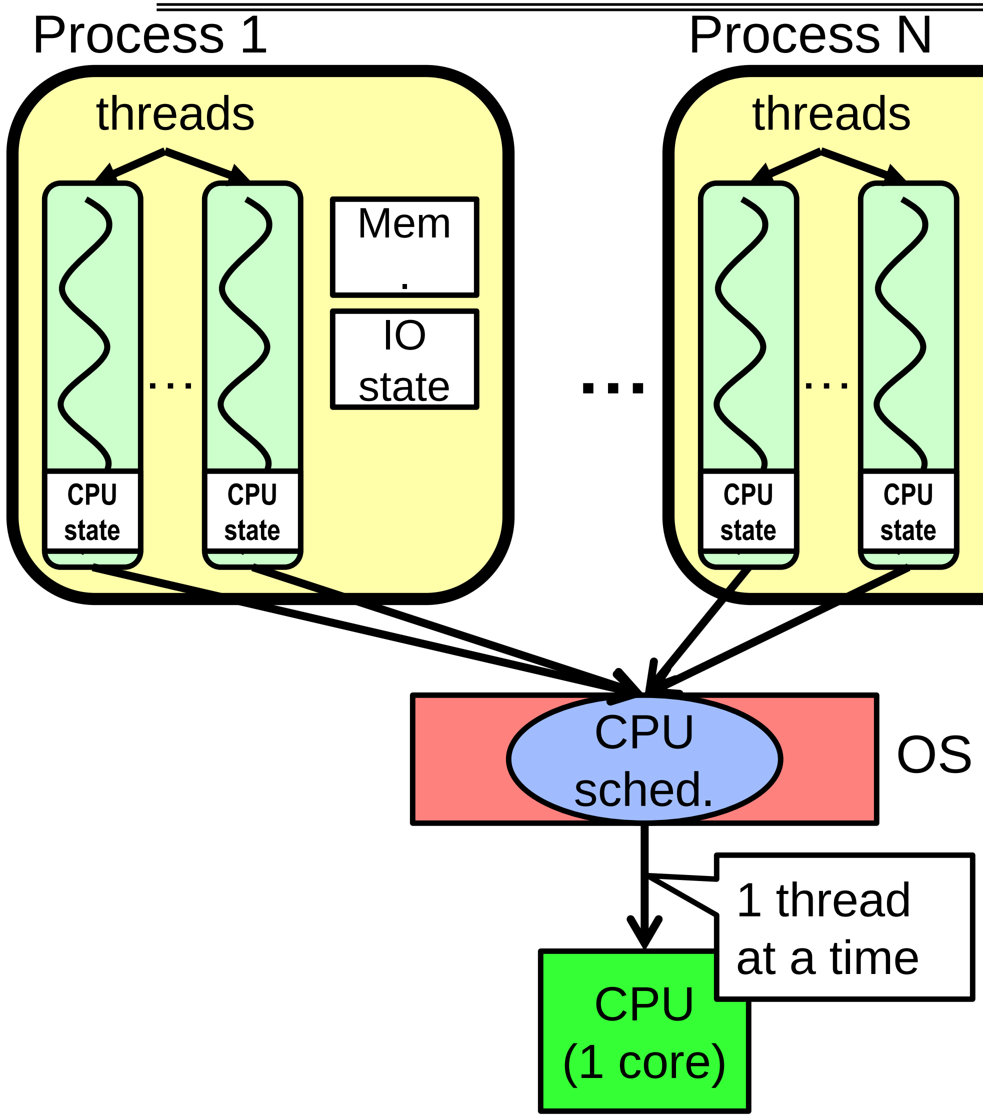


# Putting it together: Processes




- Switch overhead: high
  - CPU state: **low**
  - Memory/IO state: high
- Process creation: high
- Protection
  - CPU: **yes**
  - Memory/IO: **yes**
- Sharing overhead: high (involves at least a context switch)

# Putting it together: Threads




- Switch overhead: **low** (only CPU state)
- Thread creation: **low**
- Protection
  - CPU: **yes**
  - Memory/IO: No
- Sharing overhead: **low** (thread switch overhead low)

# Concurrency and Thread Dispatching

# Thread State

- State shared by all threads in process/address space
  - Content of memory (global variables, heap)
  - I/O state (file system, network connections, etc)

- State "private" to each thread
  - Kept in TCB ≡ Thread Control Block
  - CPU registers (including, program counter)
  - Execution stack – what is this?

- Execution Stack
  - Parameters, temporary variables
  - Return PCs are kept while called procedures are executing

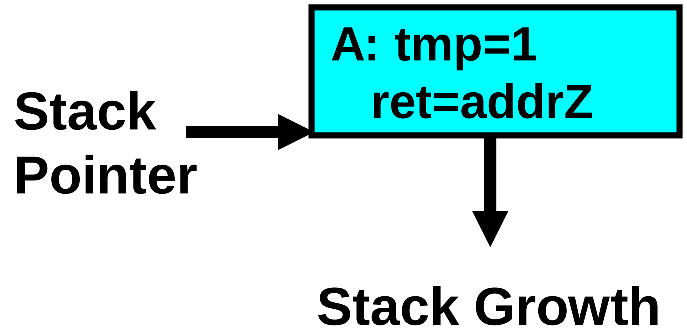
# Review: Execution Stack Example

```
addrX:  A(int tmp) {
   .      if (tmp<2)
   .        B();
   .
addrY:    printf(tmp);
   .    }
   .    B() {
   .      C();
addrU:  }
   .    C() {
   .      A(2);
   .
addrV:  }
   .    A(1);
   .
addrZ:  exit;
```

- Stack holds function arguments, return address
- Permits recursive execution
- Crucial to modern languages

# Review: Execution Stack Example

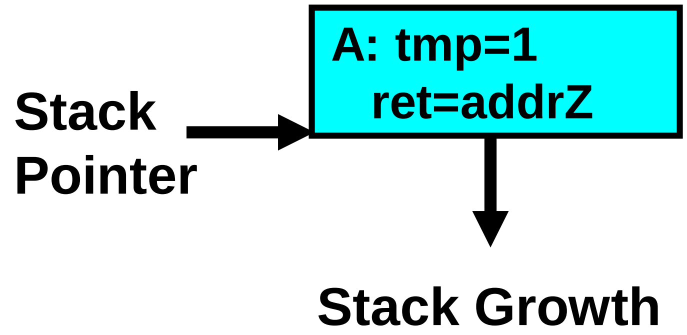
```
addrX:  A(int tmp) {
  .       if (tmp<2)
  .         B();
  .
addrY:    printf(tmp);
  .     }
  .     B() {
  .       C();
addrU:  }
  .     C() {
  .       A(2);
  .
addrV:  }
  .     A(1);
  .
addrZ:  exit;
```




- Stack holds function arguments, return address
- Permits recursive execution
- Crucial to modern languages

# Review: Execution Stack Example

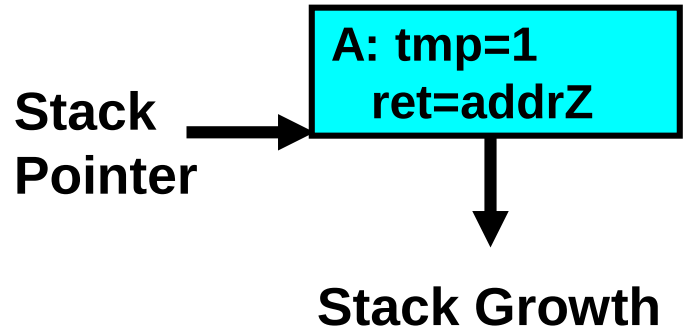
```
addrX:  A(int tmp) {
  .       if (tmp<2)
  .         B();
  .
addrY:    printf(tmp);
  .     }
  .     B() {
  .       C();
addrU:  }
  .     C() {
  .       A(2);
  .
addrV:  }
  .     A(1);
  .
addrZ:  exit;
```




- Stack holds function arguments, return address
- Permits recursive execution
- Crucial to modern languages

# Review: Execution Stack Example

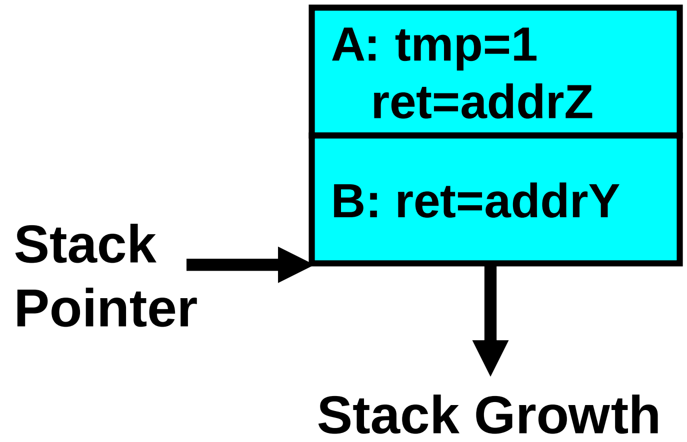
```
addrX:  A(int tmp) {
  .       if (tmp<2)
  .
  .         B();
addrY:    printf(tmp);
  .     }
  .     B() {
  .       C();
addrU:  }
  .     C() {
  .       A(2);
  .
addrV:  }
  .     A(1);
  .
addrZ:  exit;
```




- Stack holds function arguments, return address
- Permits recursive execution
- Crucial to modern languages

# Review: Execution Stack Example

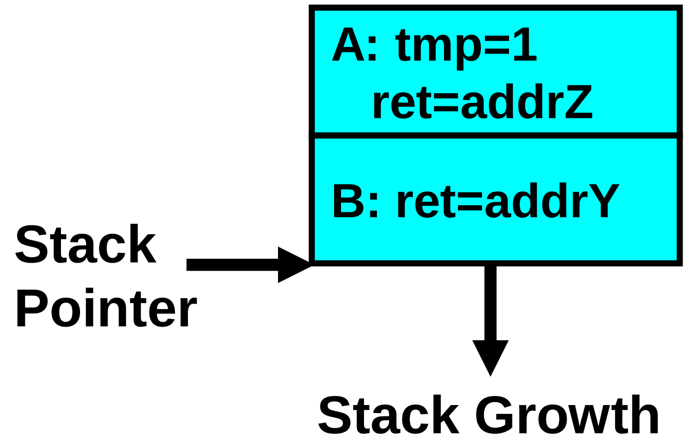
```
addrX:  A(int tmp) {
  .       if (tmp<2)
  .         B();
  .
addrY:    printf(tmp);
  .     }
  .     B() {
  .       C();
addrU:  }
  .     C() {
  .       A(2);
  .
addrV:  }
  .     A(1);
  .
addrZ:  exit;
```




- Stack holds function arguments, return address
- Permits recursive execution
- Crucial to modern languages

# Review: Execution Stack Example

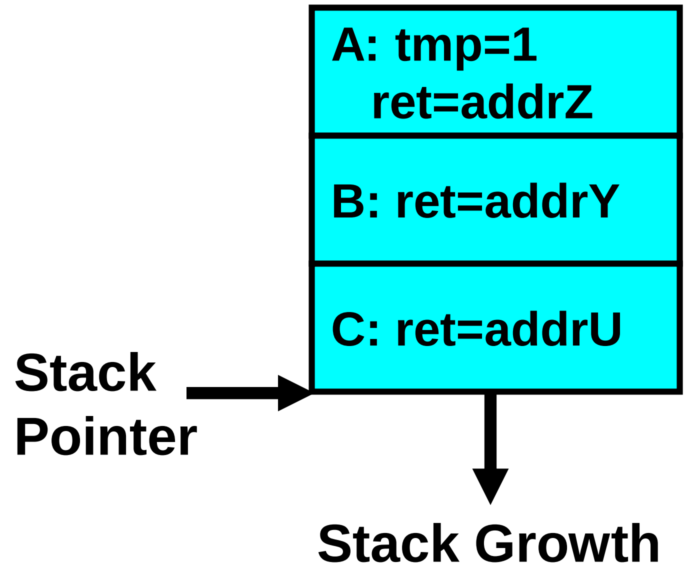
```
addrX: A(int tmp) {
         if (tmp<2)
           B();
addrY:   printf(tmp);
       }
       B() {
         C();
addrU: }
       C() {
         A(2);
addrV: }
       A(1);
addrZ: exit;
```




- Stack holds function arguments, return address
- Permits recursive execution
- Crucial to modern languages

# Review: Execution Stack Example

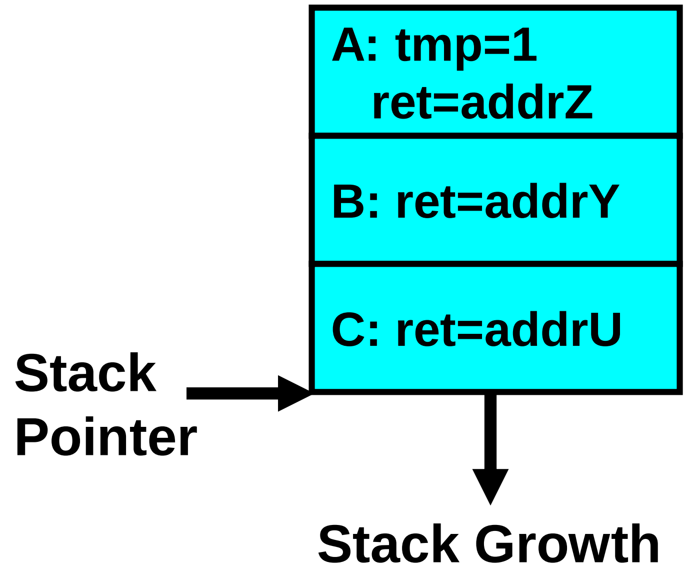
```
addrX:  A(int tmp) {
  .       if (tmp<2)
  .         B();
  .
addrY:    printf(tmp);
  .     }
  .     B() {
  .       C();
addrU:  }
  .     C() {
  .       A(2);
  .
addrV:  }
  .     A(1);
  .
addrZ:  exit;
```




- Stack holds function arguments, return address
- Permits recursive execution
- Crucial to modern languages

# Review: Execution Stack Example

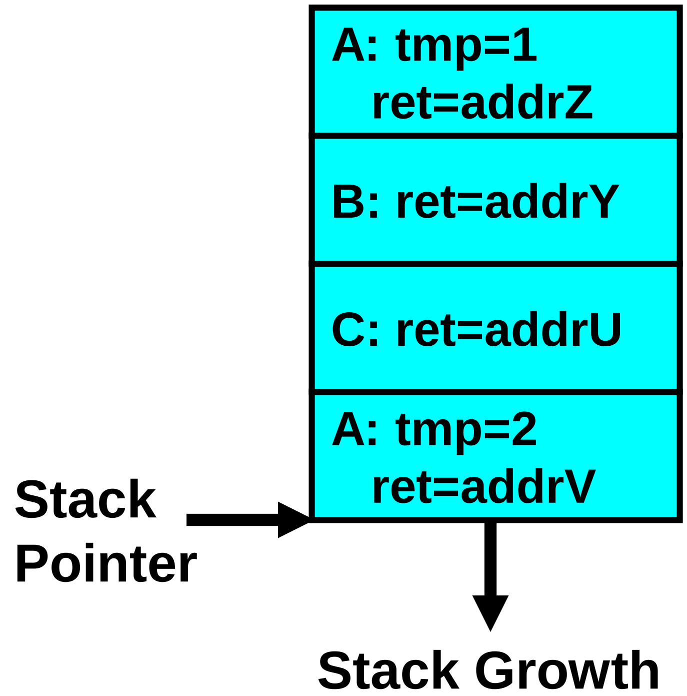
```
addrX:  A(int tmp) {
  .       if (tmp<2)
  .         B();
  .
addrY:    printf(tmp);
  .     }
  .     B() {
  .       C();
addrU:  }
  .     C() {
  .       A(2);
  .
addrV:  }
  .     A(1);
  .
addrZ:  exit;
```




- Stack holds function arguments, return address
- Permits recursive execution
- Crucial to modern languages

# Review: Execution Stack Example

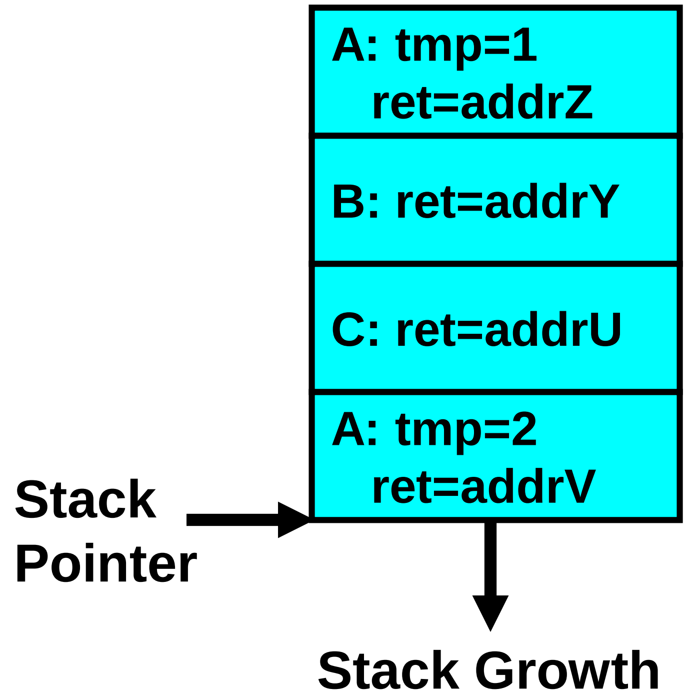
```
addrX:  A(int tmp) {
   .      if (tmp<2)
   .        B();
   .
addrY:    printf(tmp);
   .    }
   .    B() {
   .      C();
addrU:  }
   .    C() {
   .      A(2);
   .
addrV:  }
   .    A(1);
   .
addrZ:  exit;
```




- Stack holds function arguments, return address
- Permits recursive execution
- Crucial to modern languages

# Review: Execution Stack Example

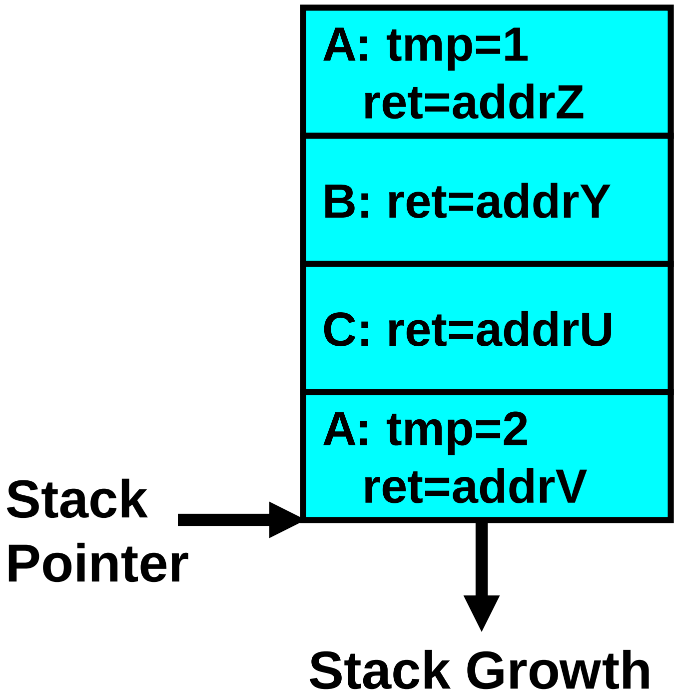
```
addrX:  A(int tmp) {
  .       if (tmp<2)
  .         B();
  .
addrY:    printf(tmp);
  .     }
  .     B() {
  .       C();
addrU:  }
  .     C() {
  .       A(2);
  .
addrV:  }
  .     A(1);
  .
addrZ:  exit;
```




- Stack holds function arguments, return address
- Permits recursive execution
- Crucial to modern languages

# Review: Execution Stack Example

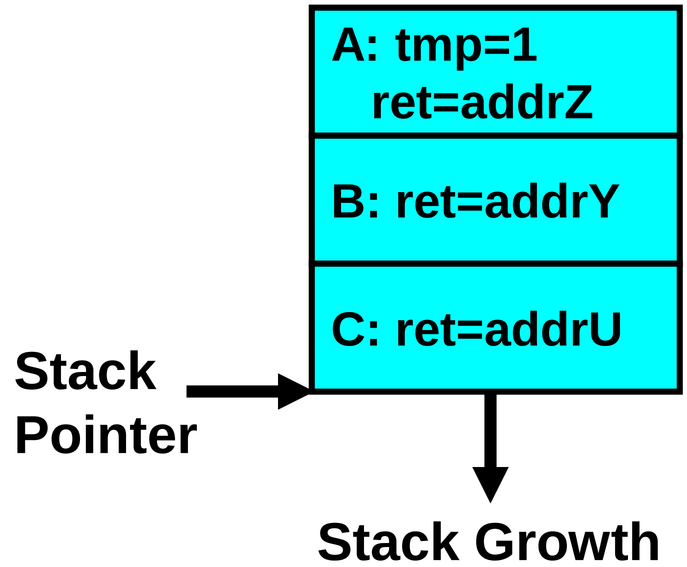
```
addrX:  A(int tmp) {
   .      if (tmp<2)
   .        B();
   .
addrY:    printf(tmp);
   .    }
   .    B() {
   .      C();
addrU:  }
   .    C() {
   .      A(2);
   .
addrV:  }
   .    A(1);
   .
addrZ:  exit;
```




**Output:**
2

# Review: Execution Stack Example

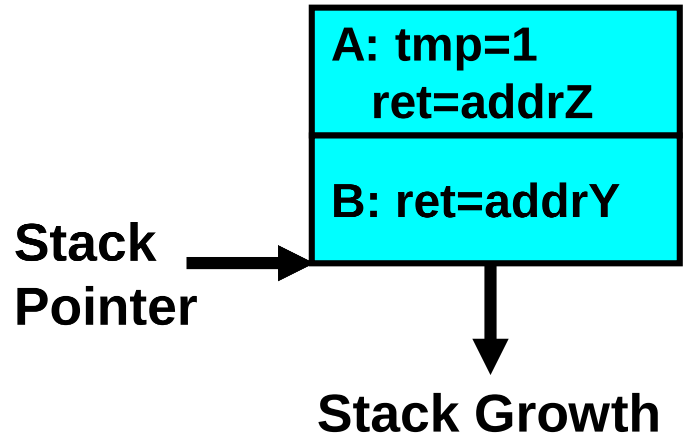
```
addrX:  A(int tmp) {
  .       if (tmp<2)
  .         B();
  .
addrY:    printf(tmp);
  .     }
  .     B() {
  .       C();
addrU:  }
  .     C() {
  .       A(2);
  .
addrV:  }
  .     A(1);
  .
addrZ:  exit;
```




**Output:**
2

# Review: Execution Stack Example

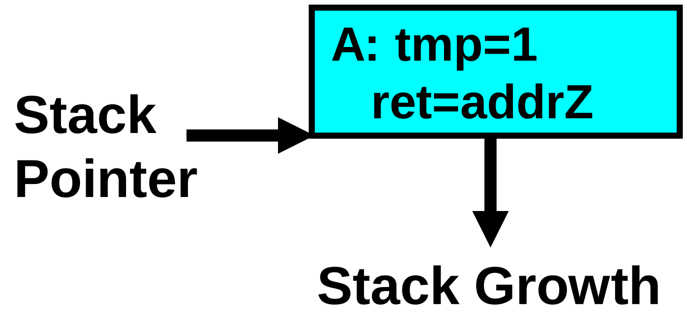
```
addrX:  A(int tmp) {
  .       if (tmp<2)
  .         B();
  .
addrY:    printf(tmp);
  .     }
  .     B() {
  .       C();
addrU:  }
  .     C() {
  .       A(2);
  .
addrV:  }
  .     A(1);
  .
addrZ:  exit;
```




**Output:**
2

# Review: Execution Stack Example

```
addrX:  A(int tmp) {
   .      if (tmp<2)
   .        B();
   .
addrY:    printf(tmp);
   .    }
   .    B() {
   .      C();
addrU:  }
   .    C() {
   .      A(2);
   .
addrV:  }
   .    A(1);
   .
addrZ:  exit;
```




**Output:**
2
1

# Review: Execution Stack Example

```
addrX:  A(int tmp) {
   .      if (tmp<2)
   .        B();
   .
addrY:    printf(tmp);
   .    }
   .    B() {
   .      C();
addrU:  }
   .    C() {
   .      A(2);
   .
addrV:  }
   .    A(1);
   .
addrZ:  exit;
```

**Output:**
**2**
**1**

# Single-Threaded Example

- Imagine the following C program:

```
main() {
    ComputePI("pi.txt");
    PrintClassList("clist.text");
}
```

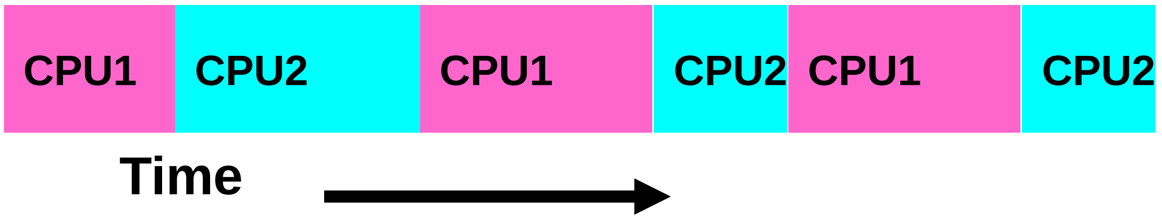
- What is the behavior here?
  - Program would never print out class list
  - Why? ComputePI would never finish

# Use of Threads

- Version of program with Threads:

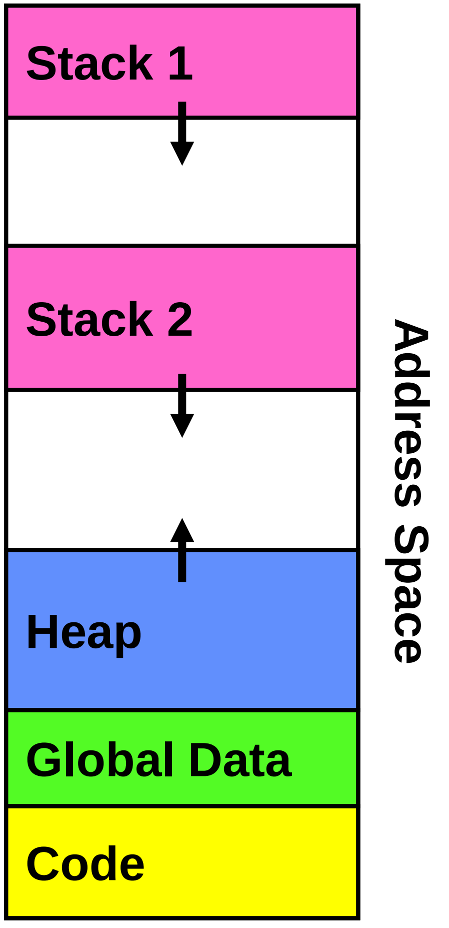
```
main() {
    CreateThread(ComputePI("pi.txt"));
    CreateThread(PrintClassList("clist.text"));
}
```

- What does `CreateThread` do?
  - Start independent thread running given procedure
- What is the behavior here?
  - This *should* behave as if there are two separate CPUs



# Memory Footprint of Two-Thread Example

- If we stopped this program and examined it with a debugger, we would see
  - Two sets of CPU registers
  - Two sets of Stacks
- Questions:
  - How do we position stacks relative to each other?
  - What maximum size should we choose for the stacks?
  - What happens if threads violate this?
  - How might you catch violations?



# Summary

- Systems typically run multiple processes simultaneously
  - Each process can be at one of various states
  - Can be scheduled in many ways
  - Can be expensive to create and switch between processes
- Multithreading
  - Multiple threads of execution within the same process
  - Useful when concurrency within an application is desired
- Thread State
  - Each thread maintains its own CPU registers (including, program counter) and execution stack in TCB
  - Shares memory and I/O with other threads of the same process

# Next

- Reading: Chapter 5.