CS330: Concurrency and thread

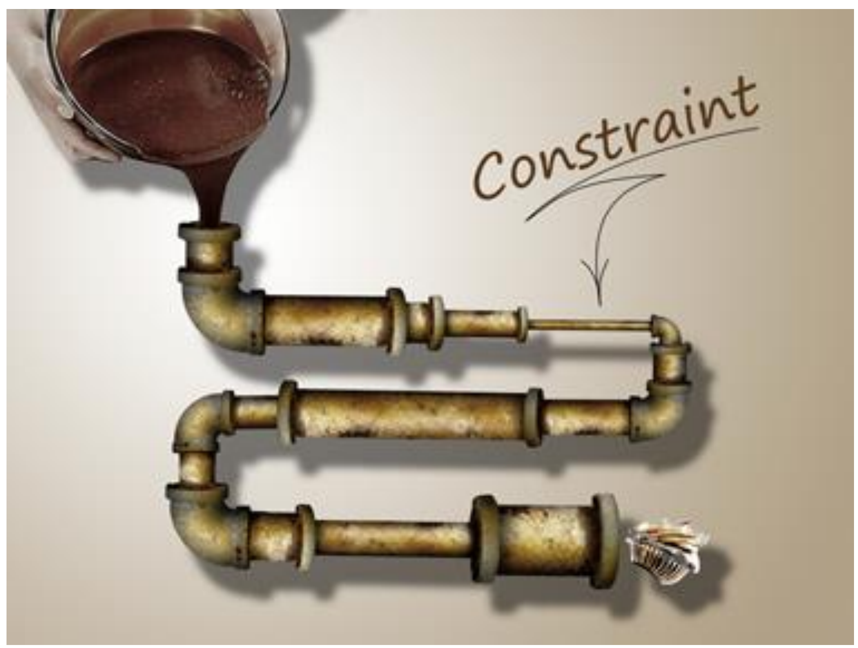
Instructor: Youngjin Kwon

Design: Motivation

- Operating systems (and application programs) often need to be able to handle multiple things happening at the same time
  - Process execution, interrupts, background tasks, system maintenance

Bottleneck

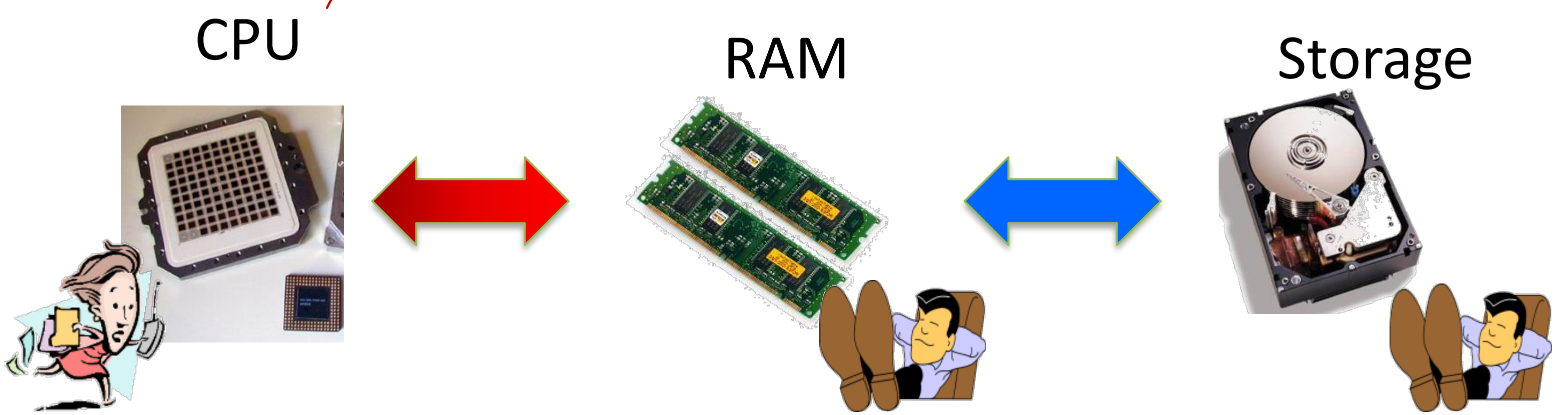
- A phenomenon where the performance of an entire system is limited by one or more components/resources
- System designers will try to avoid bottlenecks
  - try to locate and tune existing bottlenecks



The I/O Bottleneck

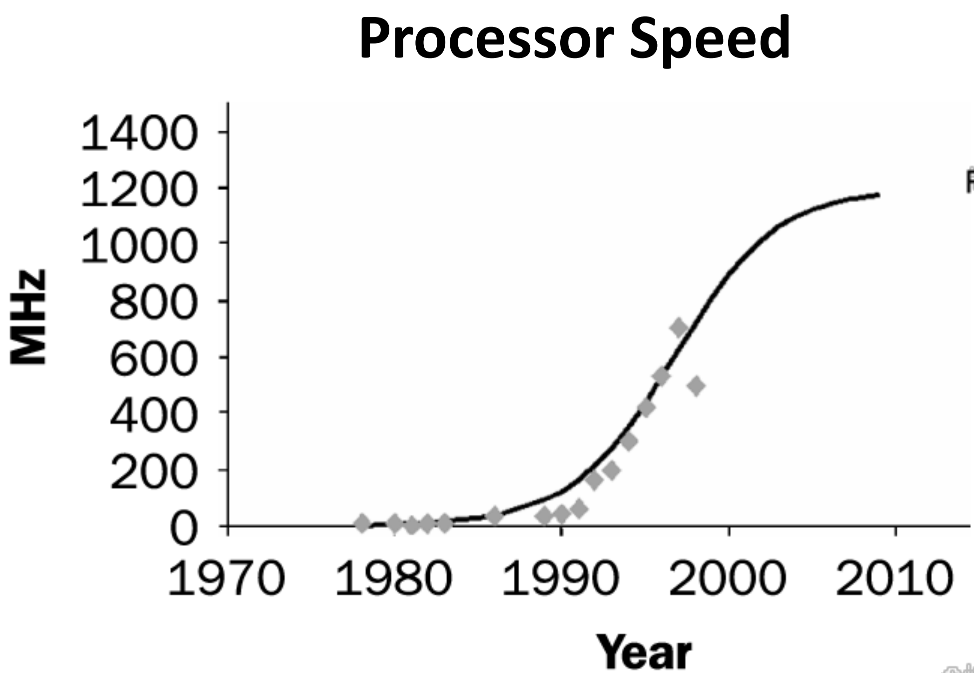
- In old days, memory and storage were faster than the processor
  - they were waiting for CPU to finish computation to feed data

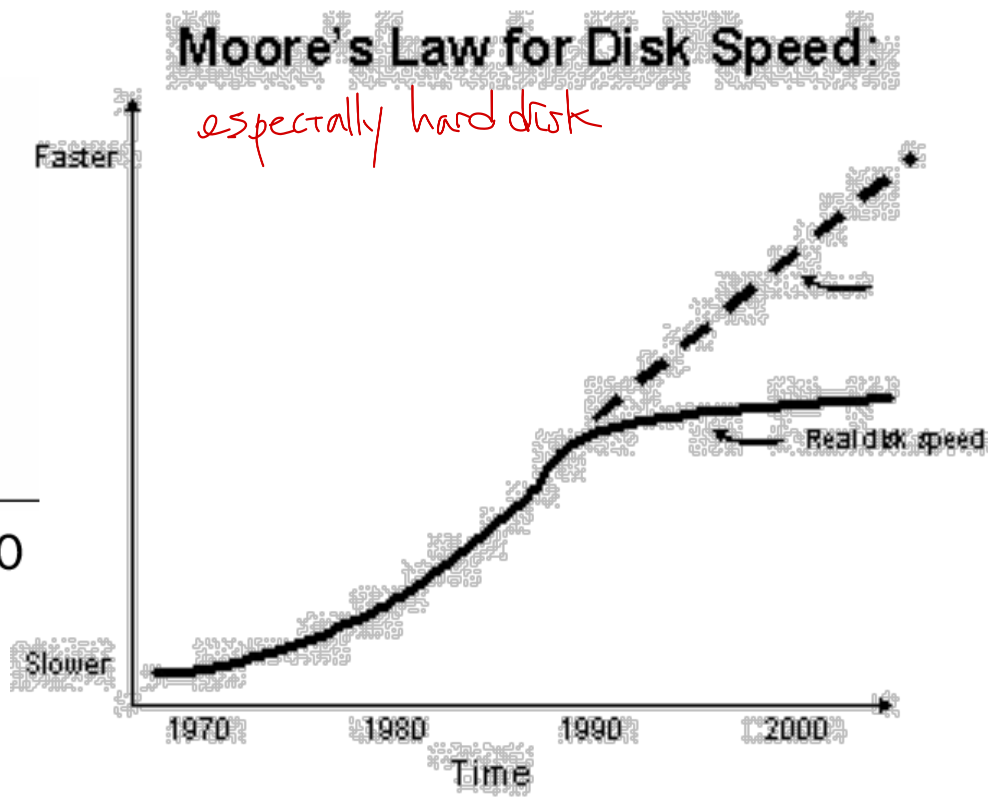
bottleneck old days



The I/O Bottleneck

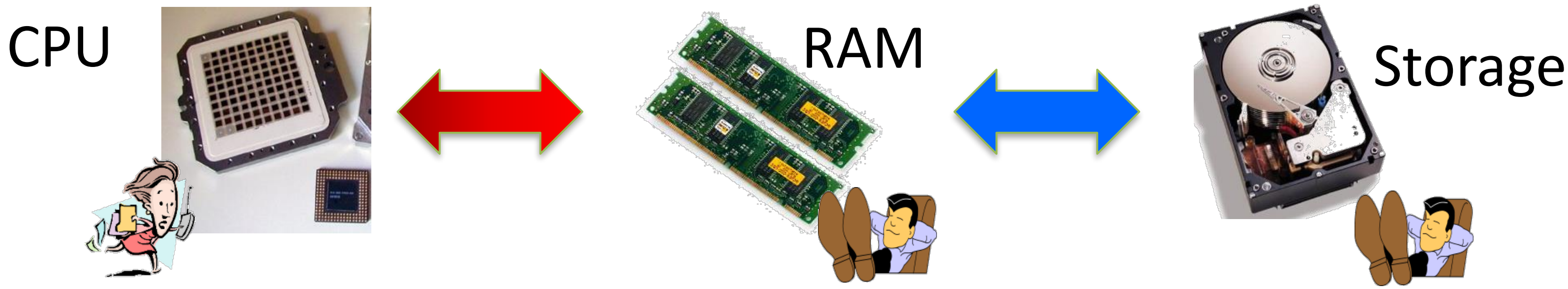
- Over the years, processor speed was improved more than that of memory and storage



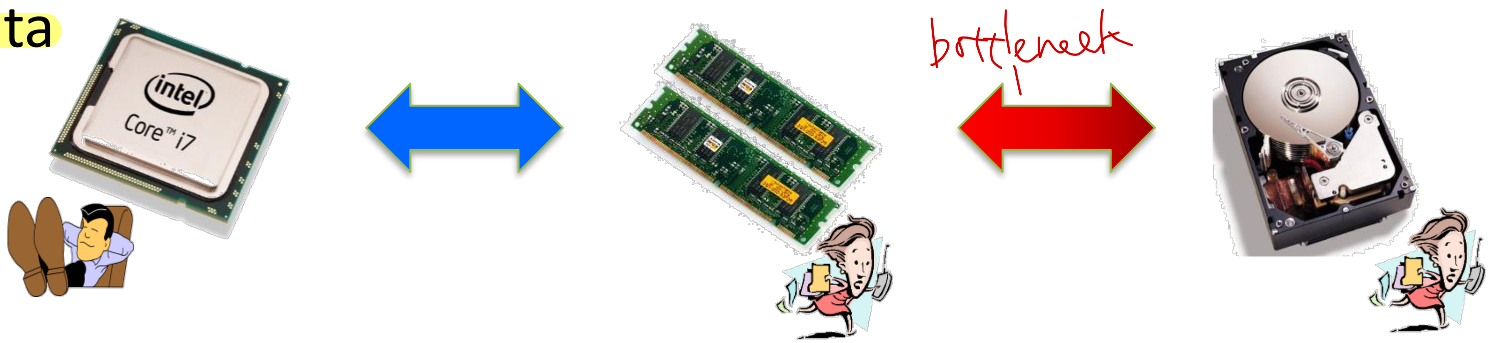


The I/O Bottleneck

- In old days, memory and storage were faster than the processor
  - they were waiting for CPU to finish computation to feed data

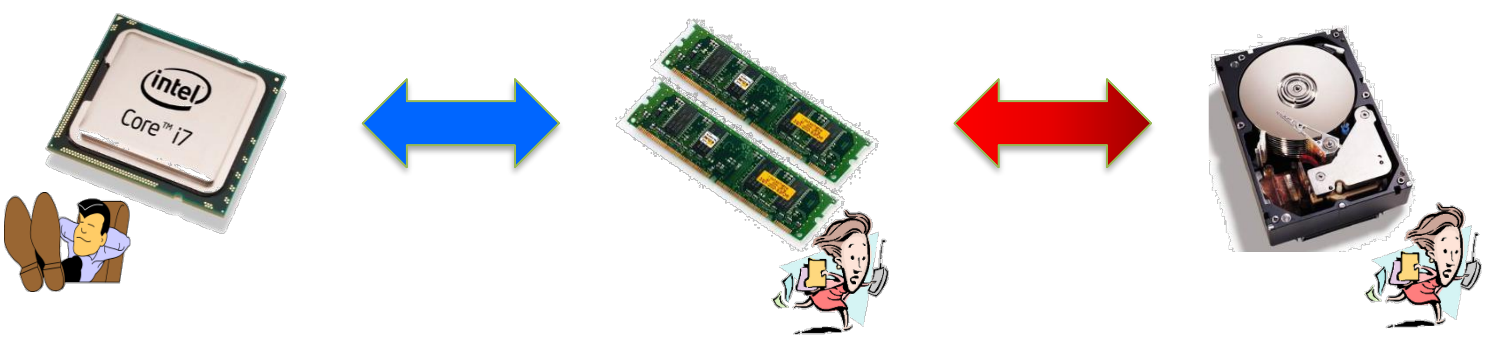


- Over the years, processor speed was improved more than that of memory and storage
- Nowadays, **the processor is waiting** for memory and storage drive to feed data



How to Design OS ?

- How to design OS to overcome the I/O bottleneck?
  - Multi-programming
    - Multi process
    - Multi ?
  - Bigger cache & better cache management algorithms



Uniprogramming vs. Multiprogramming

- Uniprogramming: *one program (application) at a time*
  - MS/DOS, early Macintosh, batch processing
  - Easier for operating system builder
  - Get rid of concurrency (only one program accessing resources!)
  - Does this make sense for personal computers?

- Multiprogramming: *more than one programs at a time*
  - Multics, UNIX/Linux, OS/2, Windows NT – 7, Mac OS X

Solution Design

Goals:

Addressing the I/O bottleneck

Solution:

- Multiprogramming: Run multiple applications concurrently

How to design?

Design: Need a new abstraction!

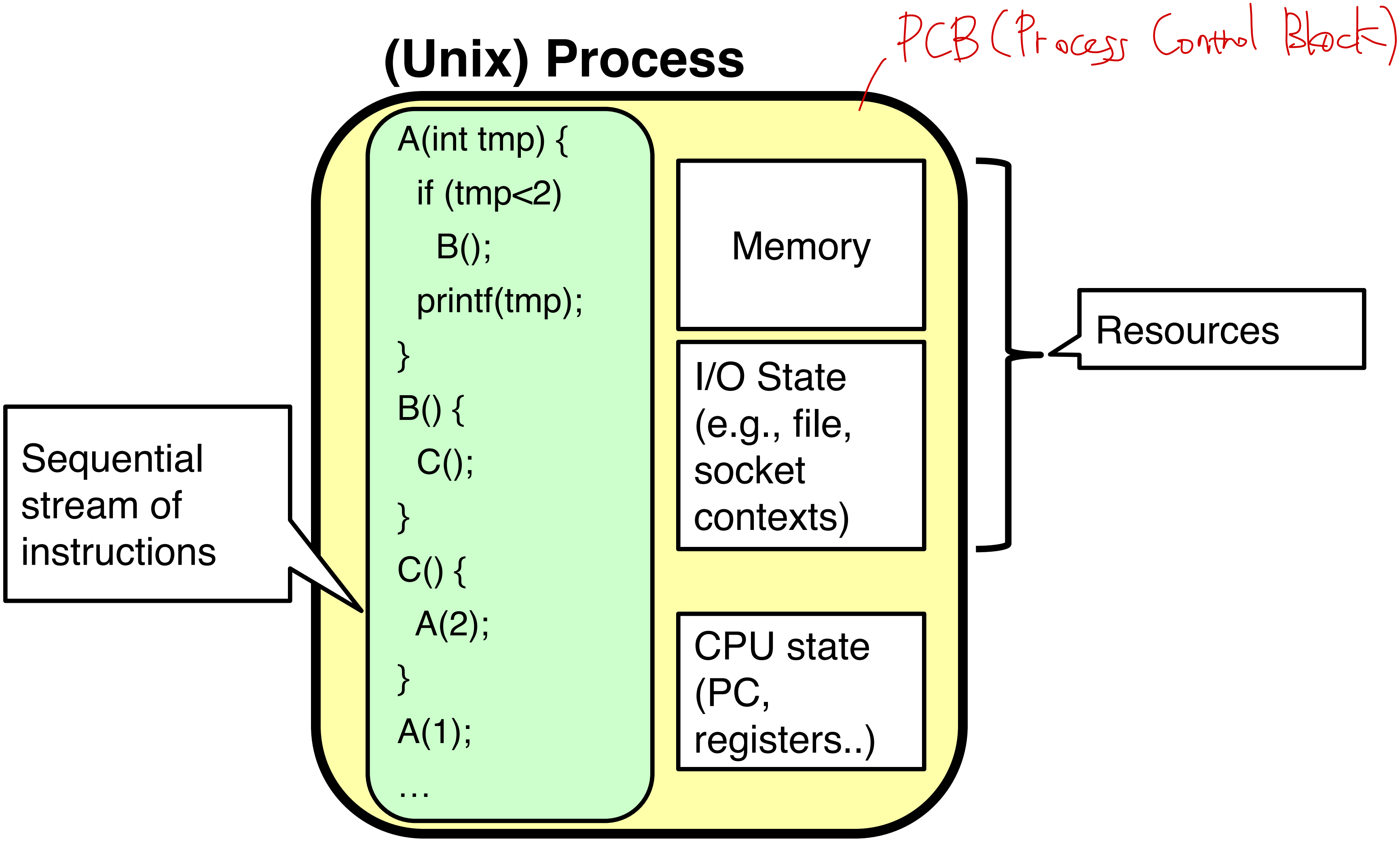
a schedulable task of execution stream

- A thread is a single execution sequence that represents a separately schedulable task
  - Single execution sequence: familiar programming model
  - Separately schedulable: OS can run or suspend a thread at any time
- Protection is an orthogonal concept
  - Can have one or many threads per protection domain

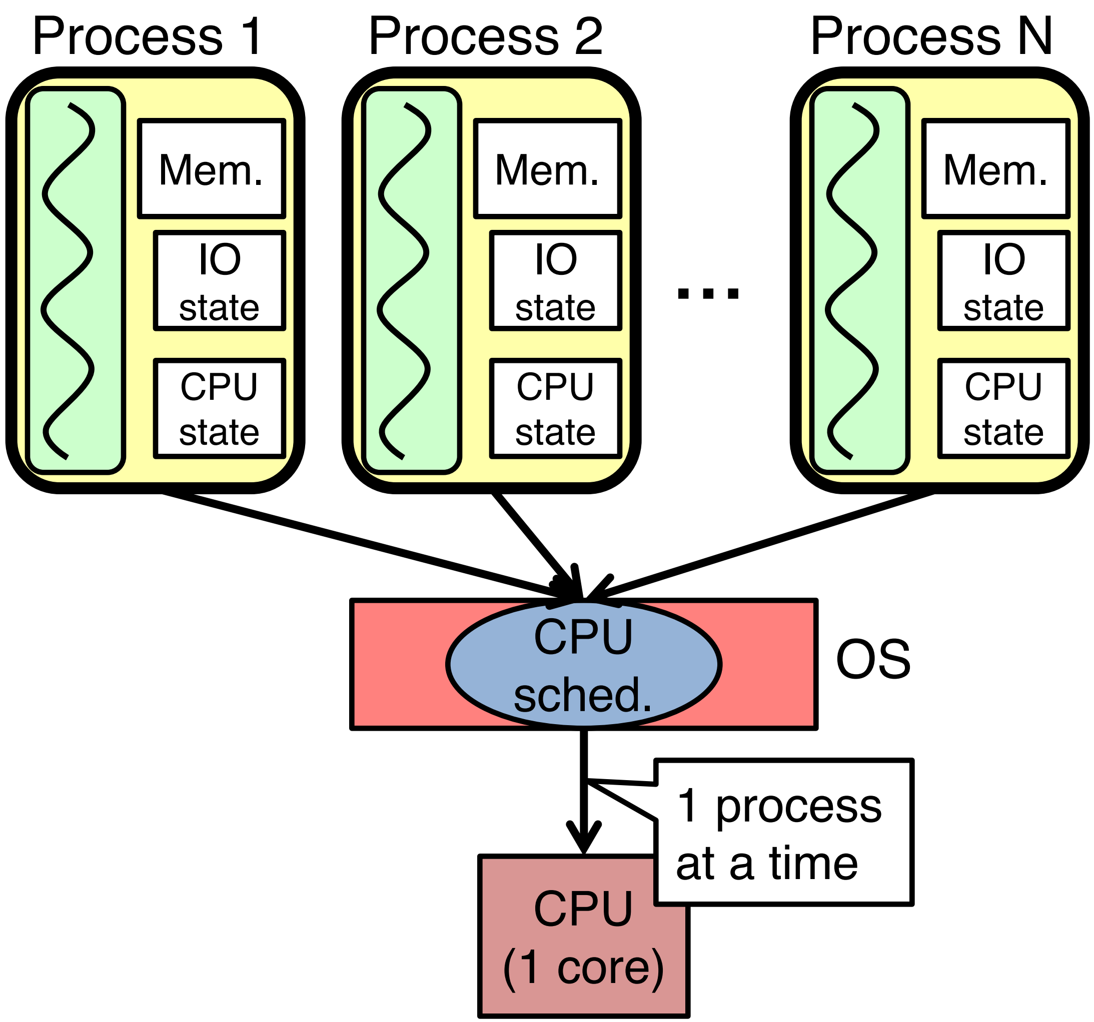
- Process
  - **Protection unit**
  - Abstraction of (Machine)

    CPU + memory + I/O
- Thread
  - **Execution unit**
  - Abstraction of (CPU)

Process


PCB (Process Control Block)
(Unix) Process
A(int tmp) {
if (tmp<2)
B();
printf(tmp);
}
B() {
C();
}
C() {
A(2);
}
A(1);
…
Memory
I/O State (e.g., file, socket contexts)
CPU state (PC, registers..)
Resources
Sequential stream of instructions

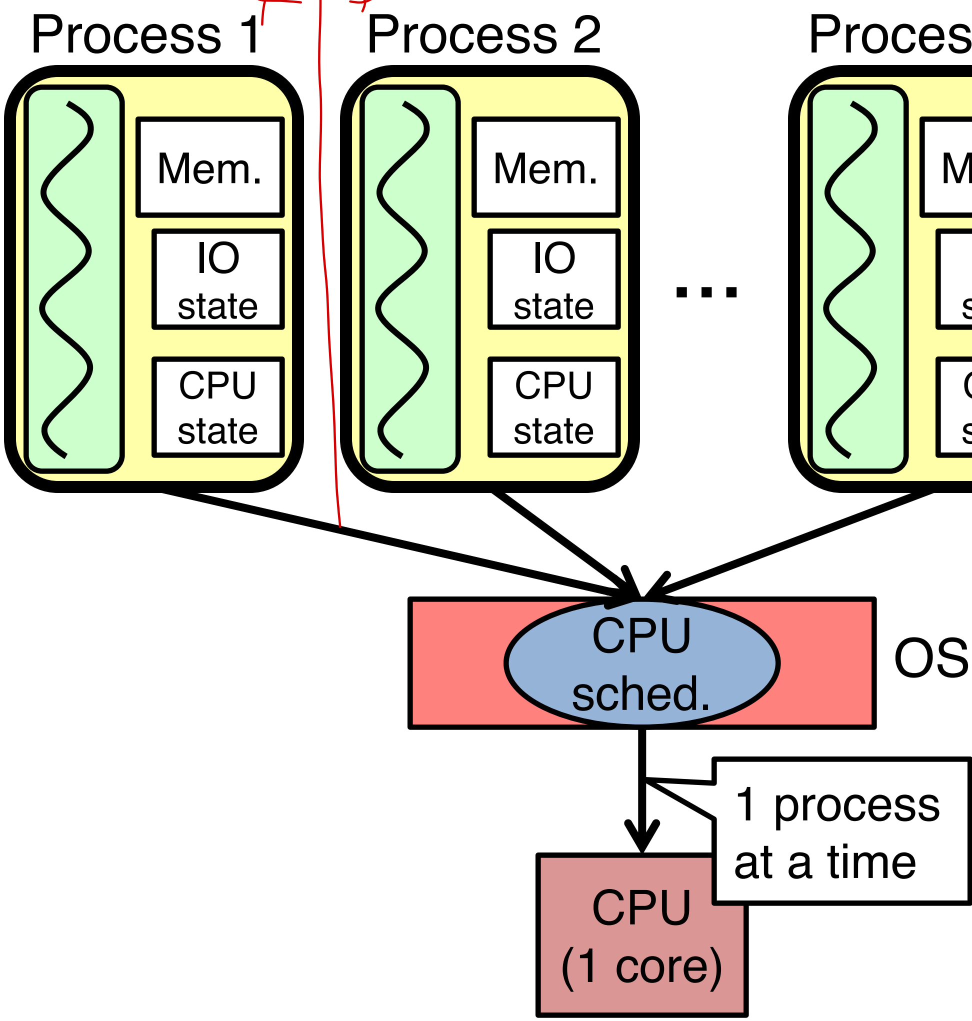
Processes



Process 1
Process 2
Process N
Mem.
IO state
CPU state
Mem.
IO state
CPU state
...
Mem.
IO state
CPU state
CPU sched.
OS
1 process at a time
CPU (1 core)

Putting it together: Processes

expensive. (Context switch(?))



Change Page Table

→ Costly

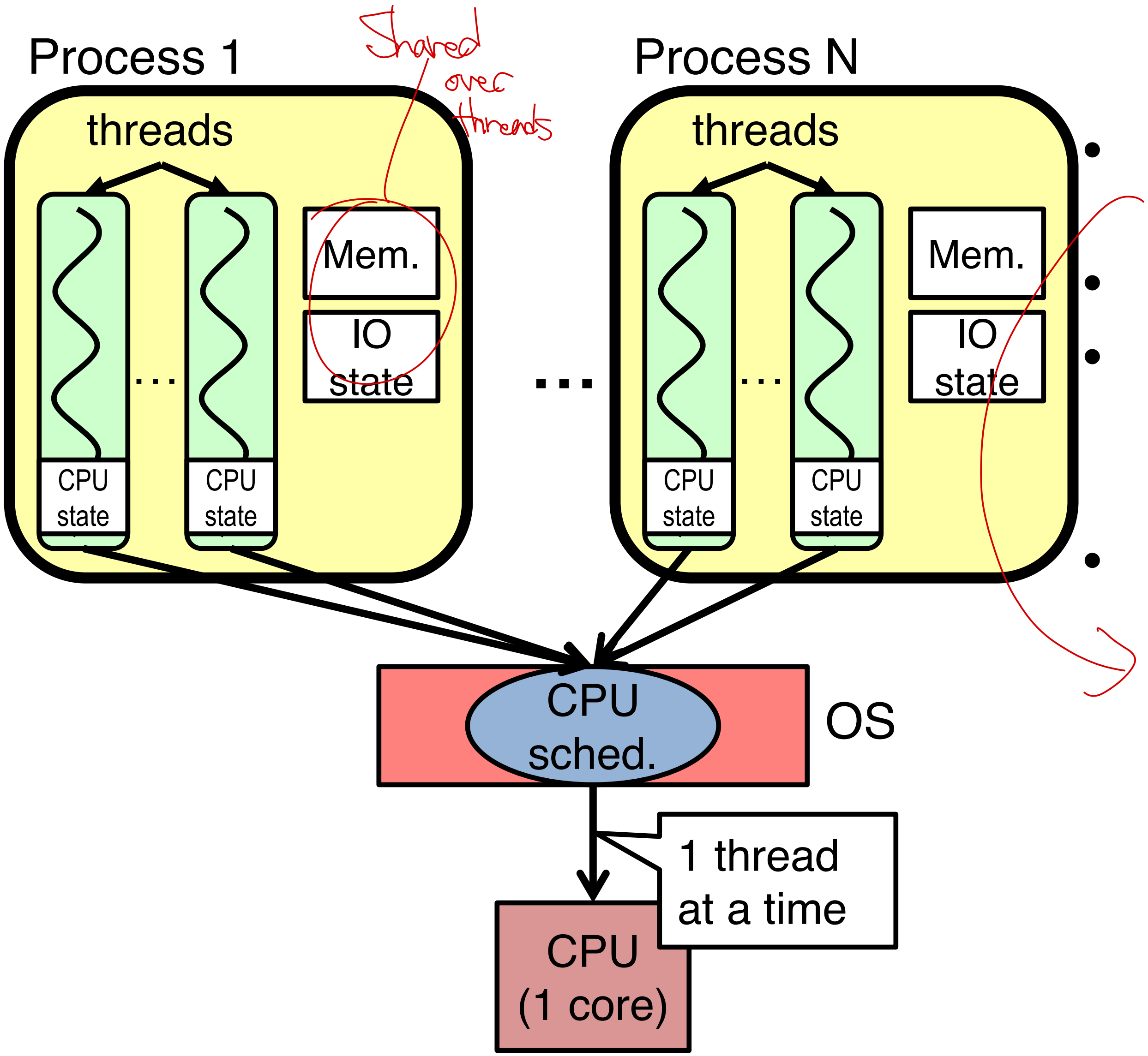
few registers

- Switch overhead: high
  - CPU state: low
  - Memory/IO state: high
- Process creation: high

Page table alloc

- 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)

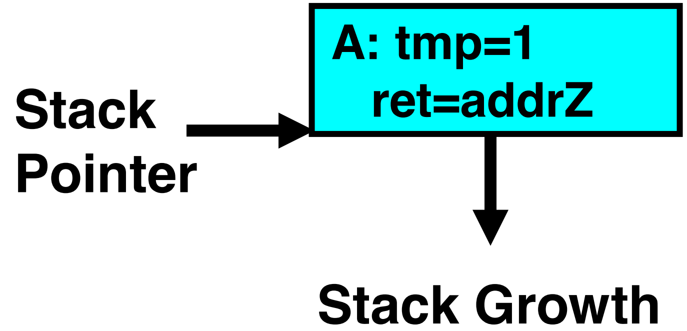
Review: Execution Stack Example

```
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 → Compiler
- Permits recursive execution
- Crucial to modern languages

Review: Execution Stack Example

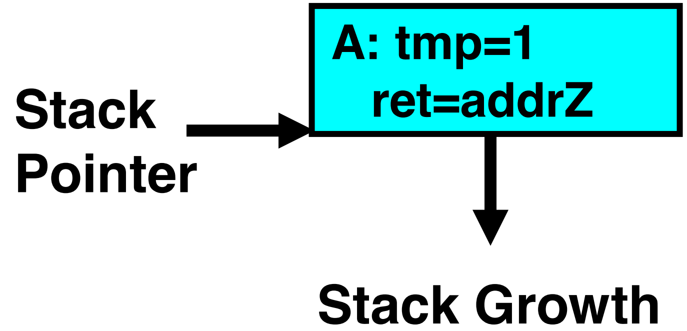
```
addrX:  A(int tmp) {        PC
  .       if (tmp<2)
  .         B();
  .
addrY:    printf(tmp);
  .     }
  .     B() {
  .       C();
addrU:  }
  .     C() {
  .       A(2);
  .
addrV:  }
  .     A(1);
  .
addrZ:  exit;
```



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

Review: Execution Stack Example

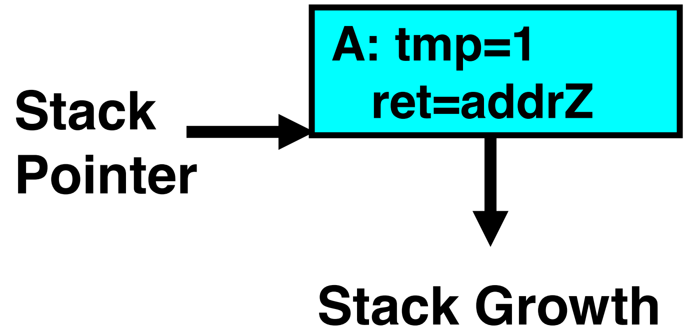
```
addrX:  A(int tmp) {
   .      if (tmp<2)          PC
   .        B();
   .
addrY:    printf(tmp);
   .    }
   .    B() {
   .      C();
addrU:  }
   .    C() {
   .      A(2);
   .
addrV:  }
   .    A(1);
   .
addrZ:  exit;
```



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

Review: Execution Stack Example

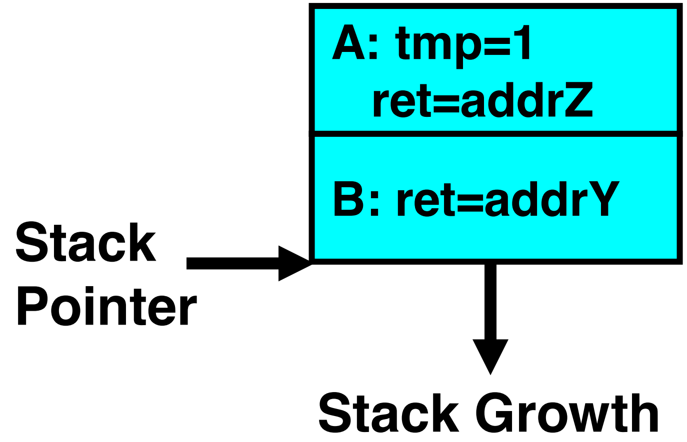
```
addrX:  A(int tmp) {
   .      if (tmp<2)
   .
   .        B();
addrY:    printf(tmp);
        }
   .
   .    B() {
   .      C();
addrU:  }
   .    C() {
   .      A(2);
   .
addrV:  }
   .    A(1);
   .
addrZ:  exit;
```



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

Review: Execution Stack Example

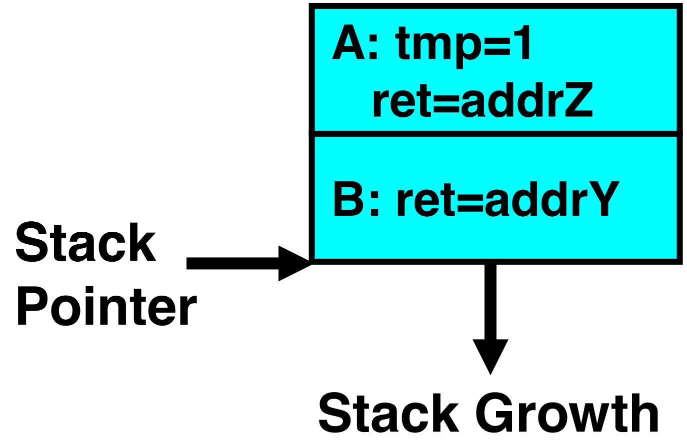
```
addrX:  A(int tmp) {
   .      if (tmp<2)
   .        B();
   .
addrY:    printf(tmp);
   .    }
   .    B() {
   .      C();
addrU:  }
   .    C() {
   .      A(2);
   .
addrV:  }
   .    A(1);
   .
addrZ:  exit;
```



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

Review: Execution Stack Example

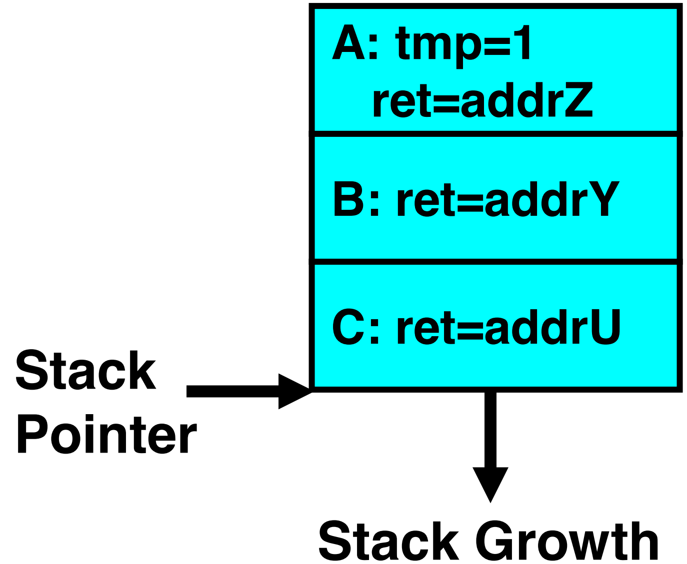
```
addrX:  A(int tmp) {
  .       if (tmp<2)
  .         B();
  .
addrY:    printf(tmp);
  .     }
  .     B() {
  .       C();
addrU:  }
  .     C() {
  .       A(2);
  .
addrV:  }
  .     A(1);
  .
addrZ:  exit;
```



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

Review: Execution Stack Example

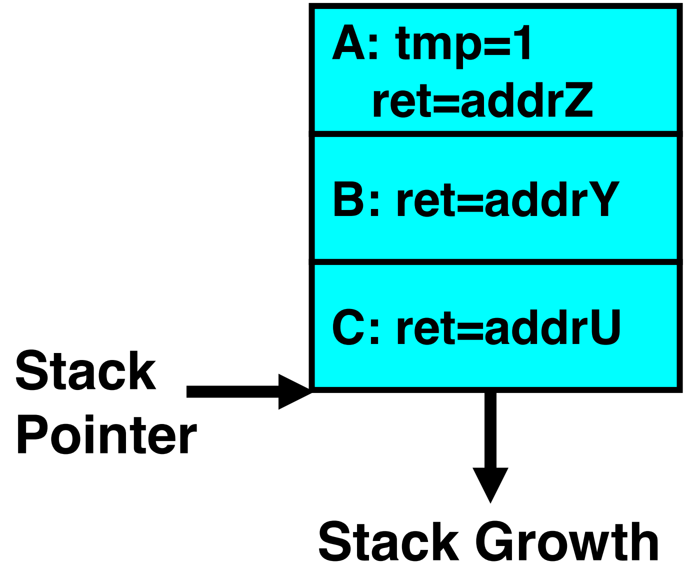
```
addrX:  A(int tmp) {
   .      if (tmp<2)
   .        B();
   .
addrY:    printf(tmp);
   .    }
   .    B() {
   .      C();
addrU:  }
   .    C() {
   .      A(2);
   .
addrV:  }
   .    A(1);
   .
addrZ:  exit;
```



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

Review: Execution Stack Example

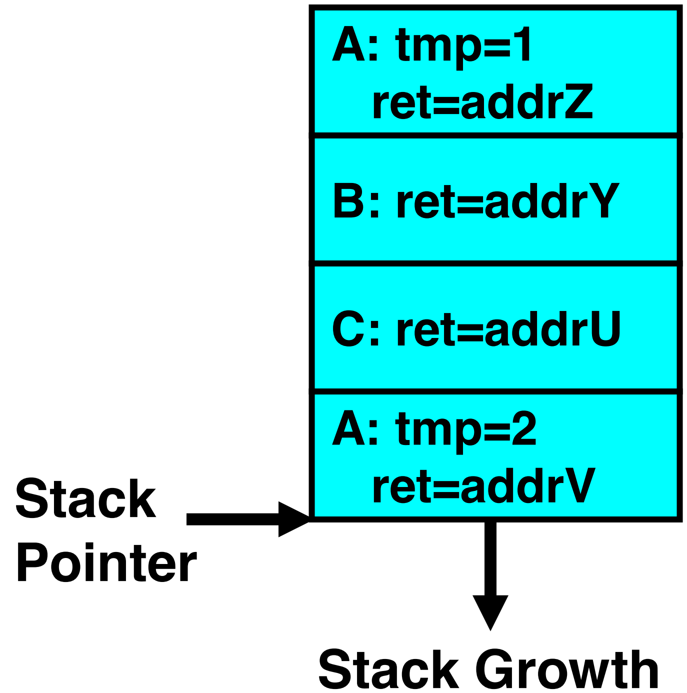
```
addrX:  A(int tmp) {
 .        if (tmp<2)
 .          B();
 .
addrY:    printf(tmp);
 .      }
 .      B() {
 .        C();
addrU:  }
 .      C() {
 .        A(2);
 .
addrV:  }
 .      A(1);
 .
addrZ:  exit;
```



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

Review: Execution Stack Example

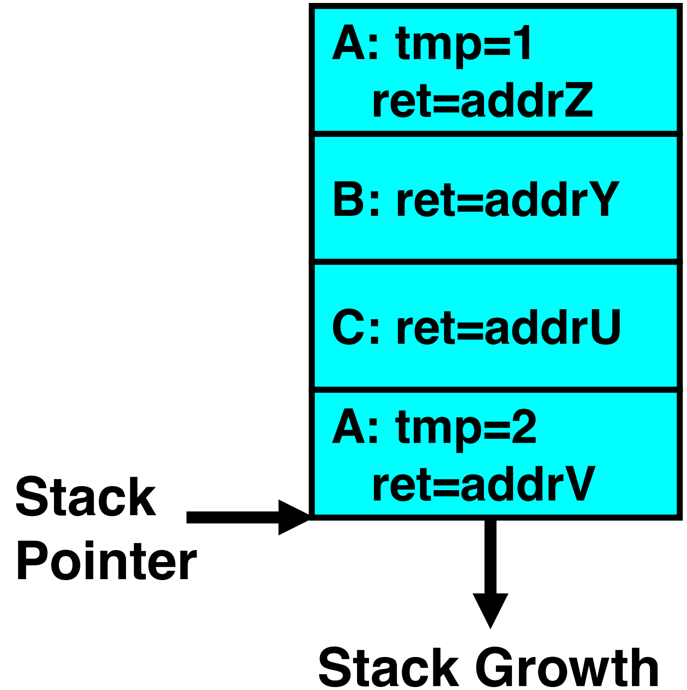
```
addrX:  A(int tmp) {
  .       if (tmp<2)
  .         B();
  .
addrY:    printf(tmp);
  .     }
  .     B() {
  .       C();
addrU:  }
  .     C() {
  .       A(2);
  .
addrV:  }
  .     A(1);
  .
addrZ:  exit;
```



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

Review: Execution Stack Example

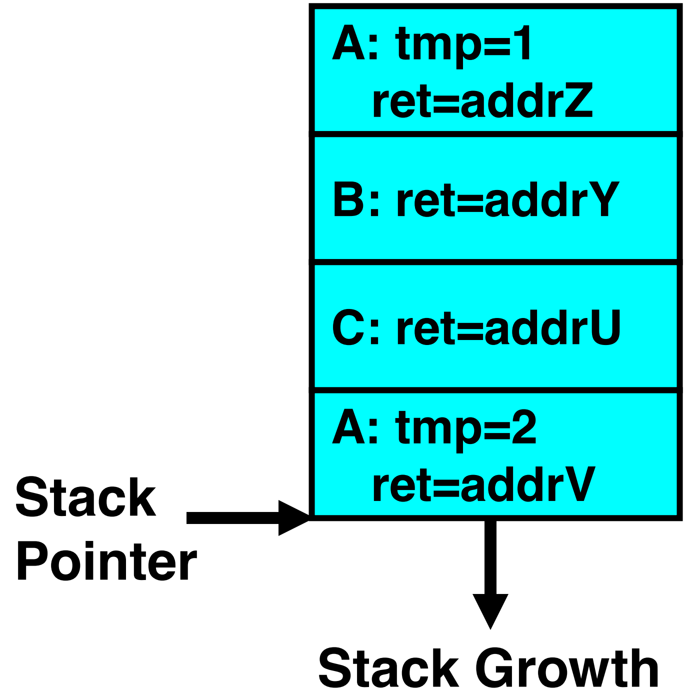
```
addrX:  A(int tmp) {
  .       if (tmp<2)
  .         B();
  .
addrY:    printf(tmp);
  .     }
  .     B() {
  .       C();
addrU:  }
  .     C() {
  .       A(2);
  .
addrV:  }
  .     A(1);
  .
addrZ:  exit;
```



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

Review: Execution Stack Example

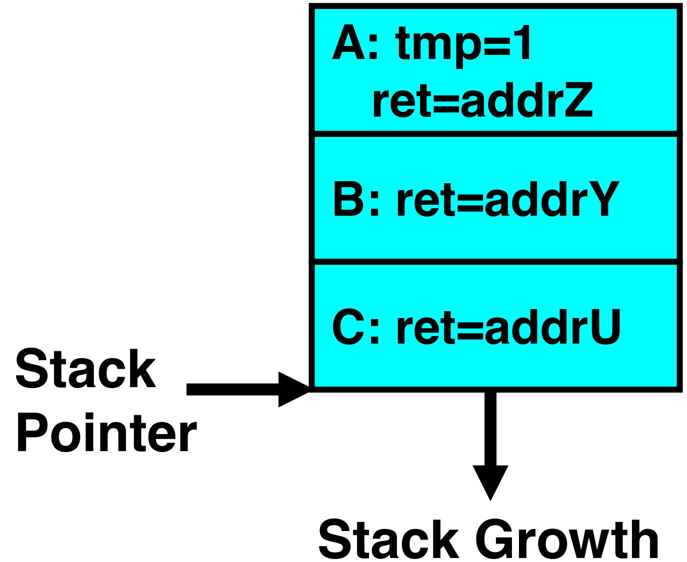
```
addrX:  A(int tmp) {
  .       if (tmp<2)
  .         B();
  .
addrY:    printf(tmp);
  .     }
  .     B() {
  .       C();
addrU:  }
  .     C() {
  .       A(2);
  .
addrV:  }
  .     A(1);
  .
addrZ:  exit;
```



Output:
2

Review: Execution Stack Example

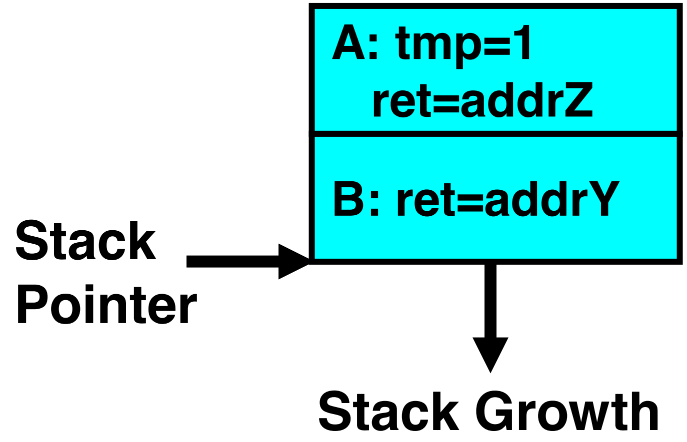
```
addrX:  A(int tmp) {
  .       if (tmp<2)
  .         B();
  .
addrY:    printf(tmp);
  .     }
  .     B() {
  .       C();
addrU:  }
  .     C() {
  .       A(2);
  .
addrV:  }
  .     A(1);
  .
addrZ:  exit;
```



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;
```

Stack Pointer

A: tmp=1
ret=addrZ

Stack Growth

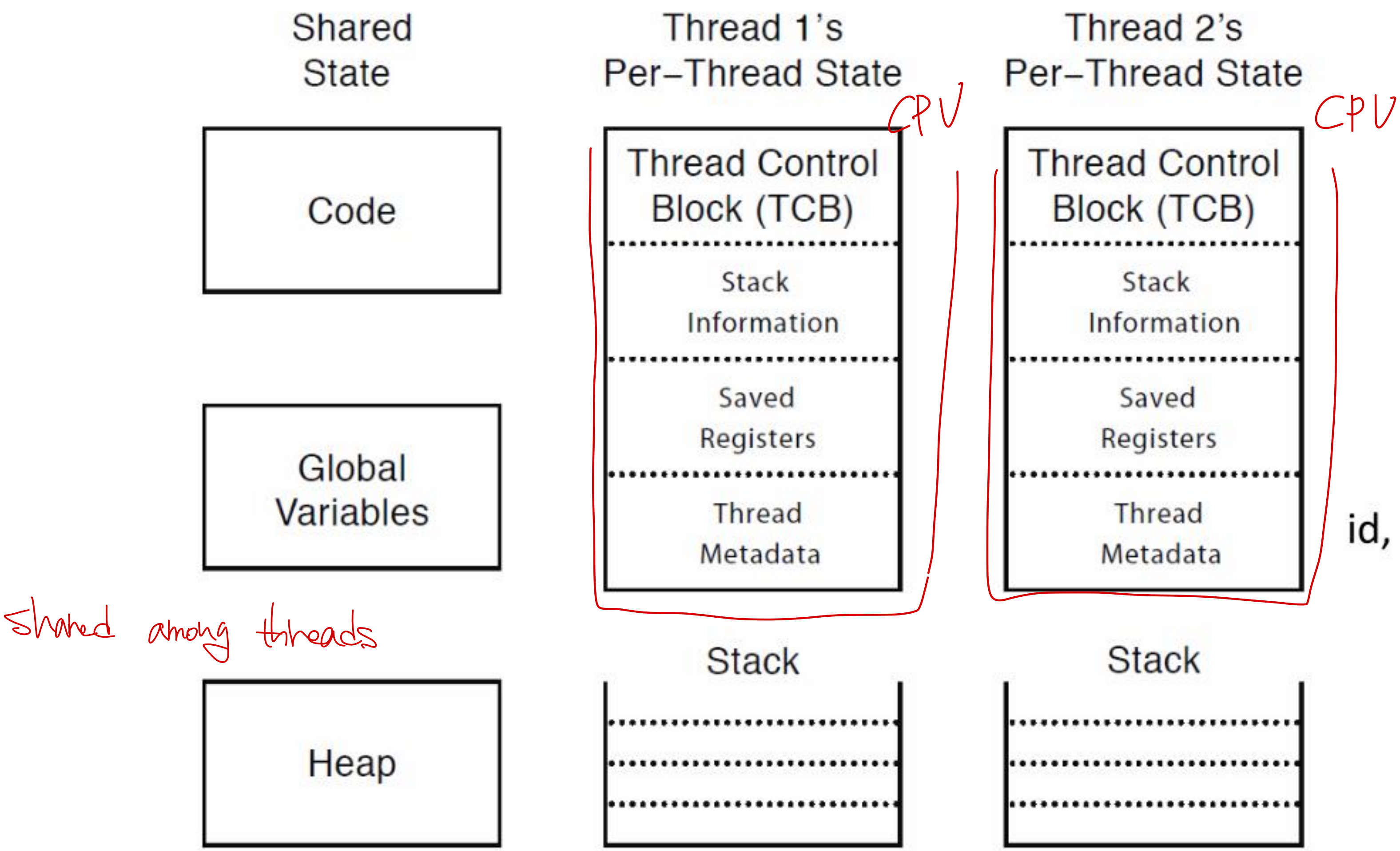
Output:
2
1

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

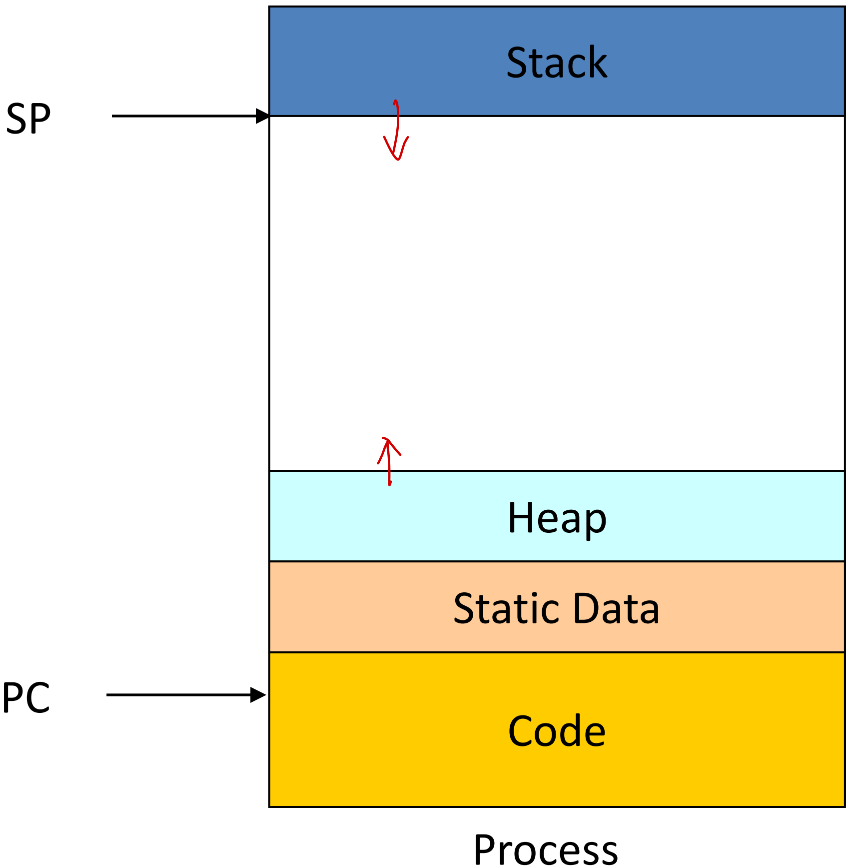
Thread Data Structures



id, status, ...

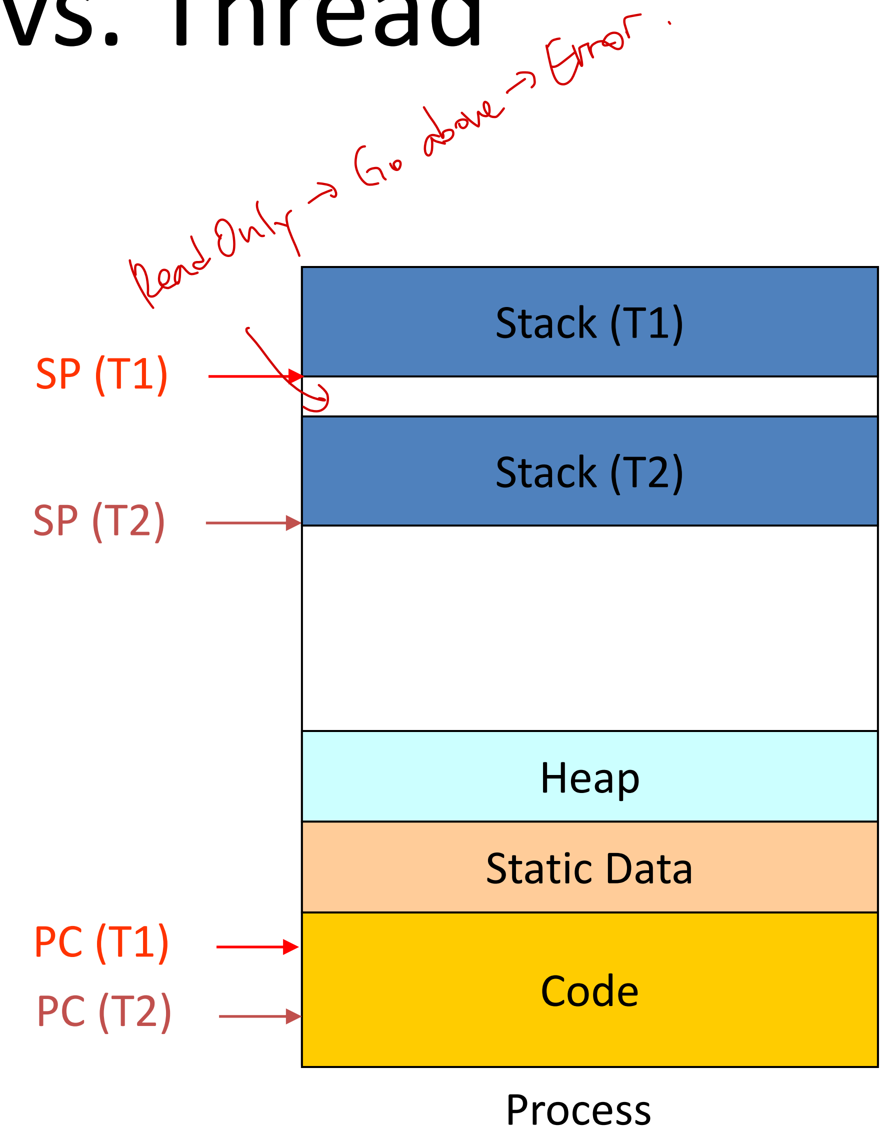
Process vs. Thread

- Execution context
  - Program counter (PC)
  - Stack pointer (SP)
  - Data registers
- Code
- Data
- Stack

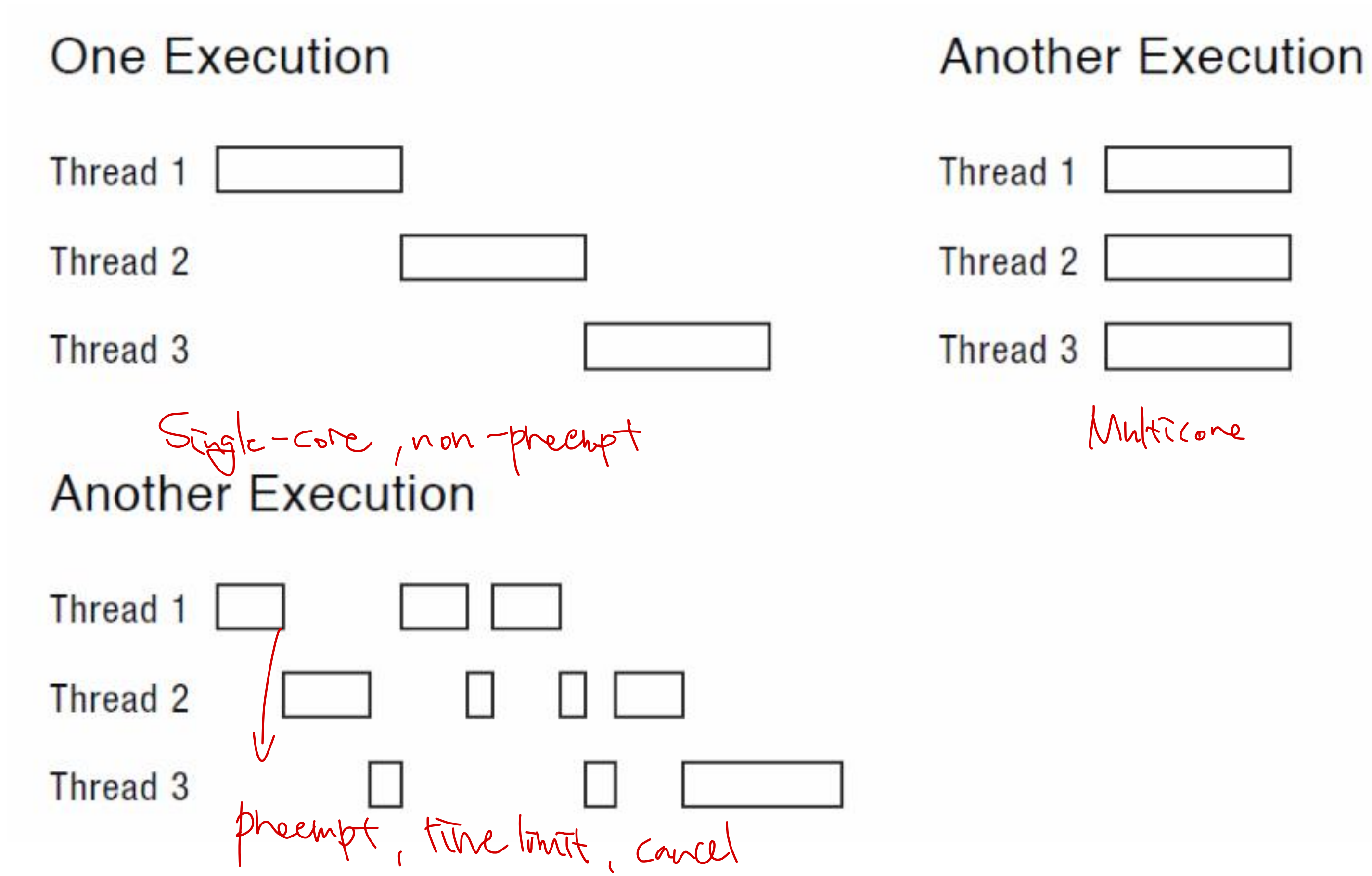


Process vs. Thread

- Execution context
  - Program counter (PC)
  - Stack pointer (SP)
  - Data registers
- Stack



Possible Executions



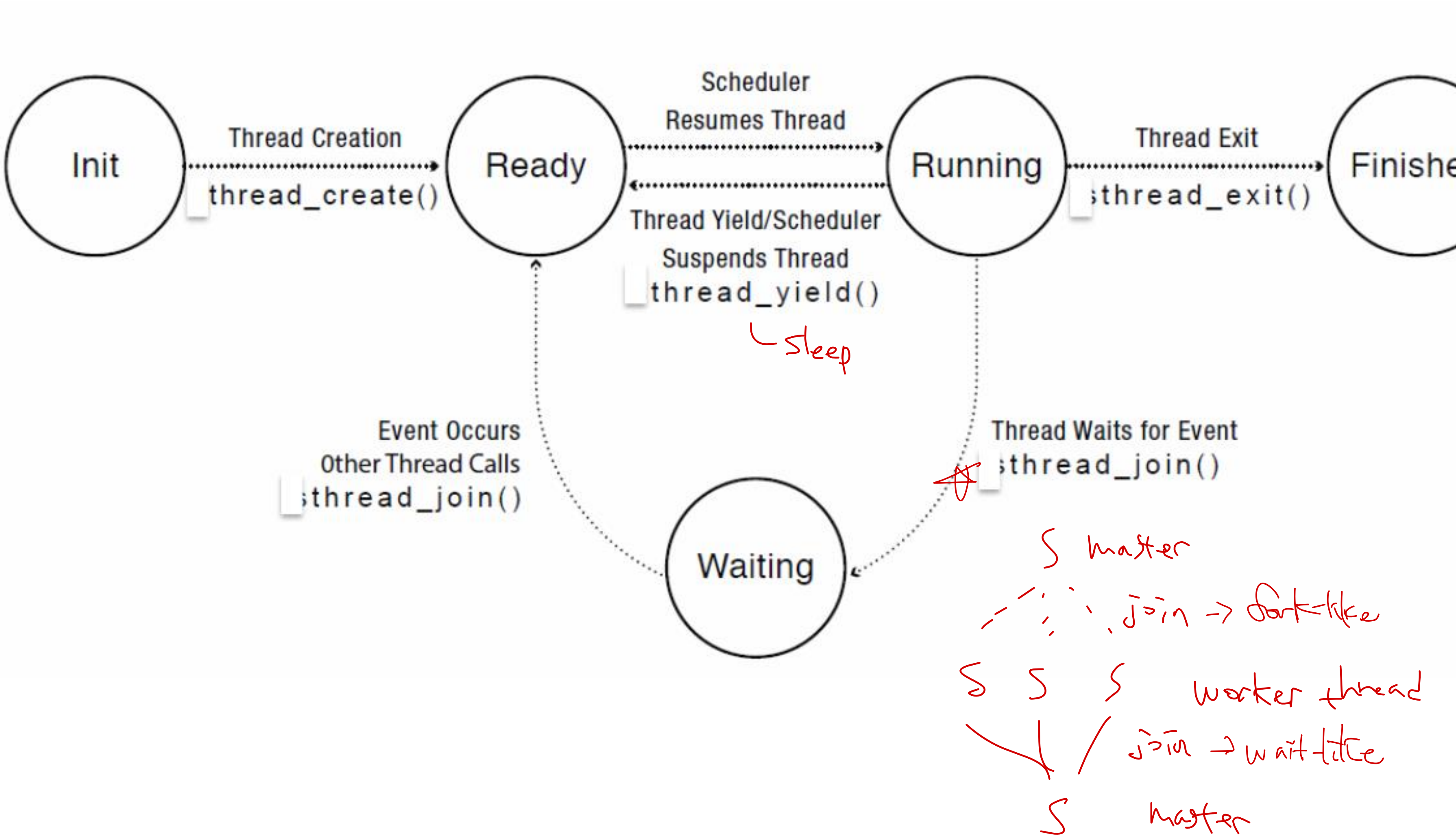
Thread performance is highly unpredictable!

Thread programming model

- Cannot assume execution time of other threads (which one is correct?)
  - X – Thread 1 uses sleep (3) to wait for thread 2 to finish its task → never expect timing of other threads
  - O – Thread 1 waits a signal from thread 2 to wait for thread 2 to finish its task → Semaphore, conditional variable

- Need synchronization when accessing shared data
  - Synchronization provides "expected execution orders"

Thread Lifecycle to Design APIs



Thread APIs

- thread_create(thread, func, args)
  - Create a new thread to run func(args)
- thread_yield()

  [illegible]
  - Relinquish processor voluntarily
- thread_join(thread)
  - In parent, wait for forked thread to exit, then return
- thread_exit
  - Quit thread and clean up, wake up joiner if any

Example: threadHello

```
#define NTHREADS 10
thread_t threads[NTHREADS];
main() {
   for (i = 0; i < NTHREADS; i++)  thread_create(&threads[i], &go, i);
   for (i = 0; i < NTHREADS; i++) {
      exitValue = thread_join(threads[i]);
      printf("Thread %d returned with %ld\n", i, exitValue);
   }
   printf("Main thread done.\n");
}
void go (int n) {
   printf("Hello from thread %d\n", n);
   thread_exit(100 + n);
   // REACHED?
}
```

Create 10 threads

→ All threads are done.

but exec order not guaranteed

Design: What is the next?

- Now, you built the thread abstraction
  - How the thread look like
  - Execution model of the thread
  - Define necessary data structures for thread
- Then?

Schedule

Thread scheduling

- Concept of scheduler
  - Map execution unit to processor
  - How to (policies)?
    - FCFS, SJF, RR, priority scheduling, and MLFQ
    - We already covered the scheduling policies

Summary: Process vs. Thread

- A thread is bounded to a single process
- Processes are now containers in which threads execute
- A process can have multiple threads
- Sharing data between threads is cheap: all see the same address space
- Threads become the unit of scheduling

Process Thread

Protection

Weight

mid-way? Light weight context

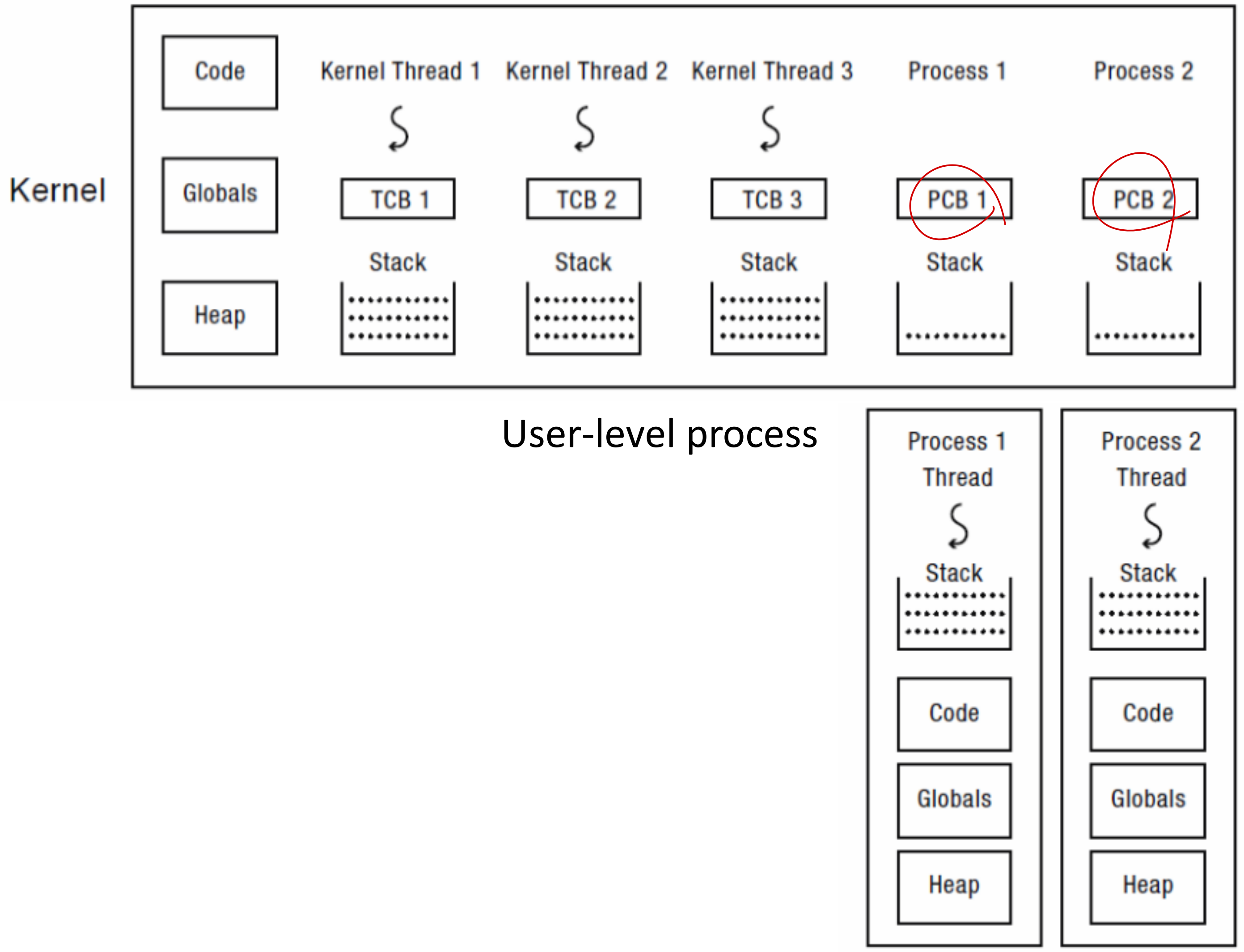
Implementation question

- **Who takes care of thread management?**
  - OS (kernel threads)
    - System calls for thread creation and management
  - User-level process (user-level threads)
    - a library linked into the program takes care of it

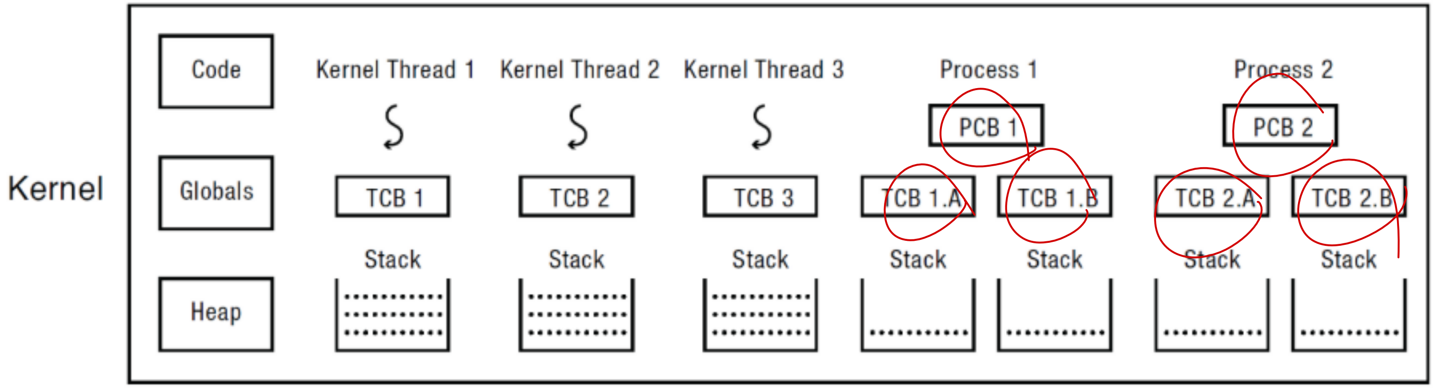
Implementing Threads: Roadmap

- Kernel threads
  - Thread abstraction only available to kernel
  - To the kernel, a kernel thread and a single threaded user process look quite similar
- Multithreaded processes using kernel threads (Linux, MacOS)
  - Kernel thread operations available via syscall
- User-level threads
  - Thread operations without system calls

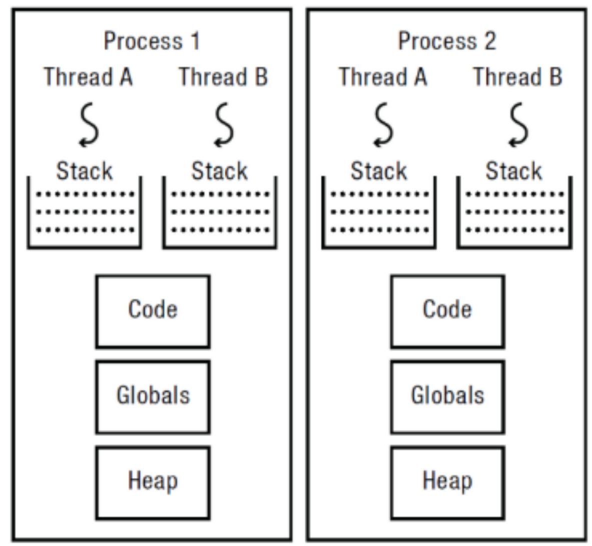
Multithreaded OS Kernel single-threaded process



Multithreaded OS Kernel
Multi-threaded process



User-level process



pthread

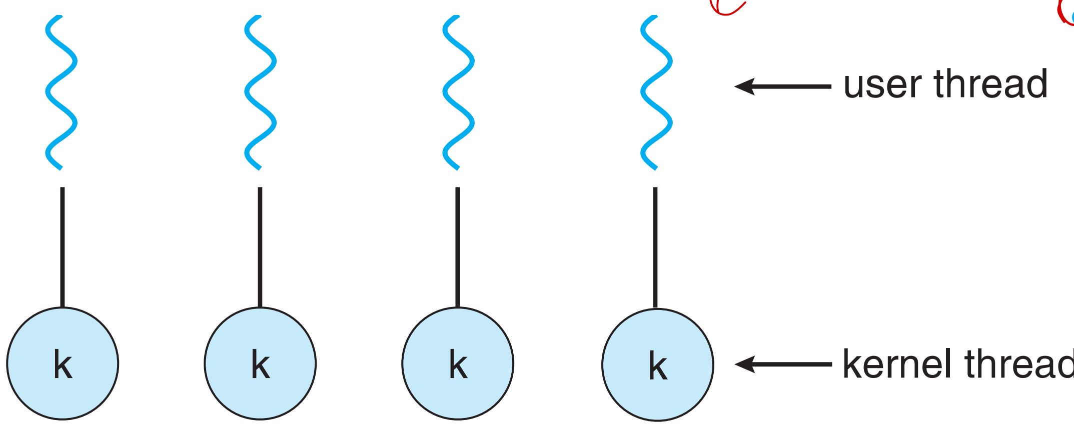
→ standard thread API

→ implement : kernel thread

Threading model

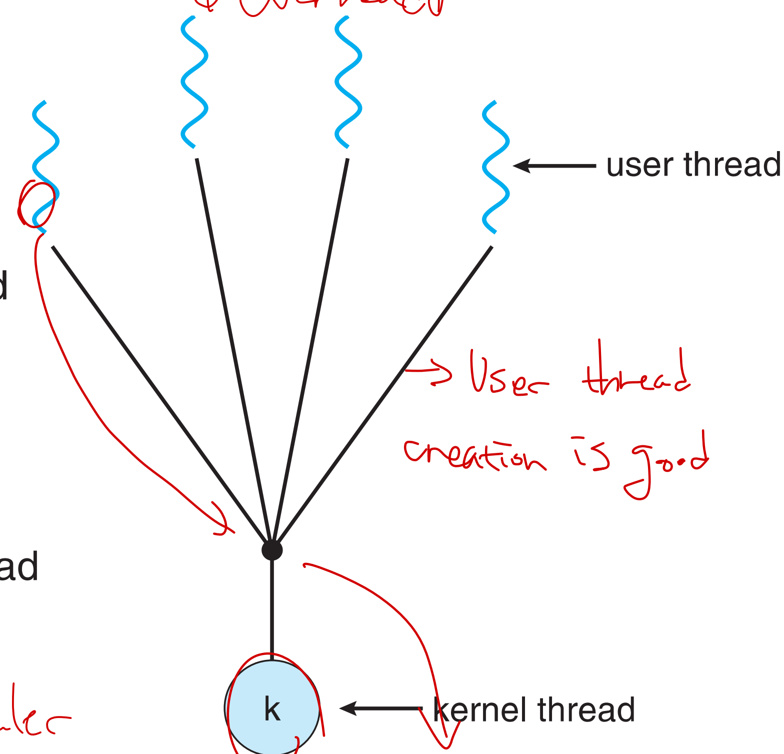
https://www.crocus.co.kr/1404

- Kernel thread
- User-level thread



thread lib scheduler

OS scheduler
(Need kernel mode)



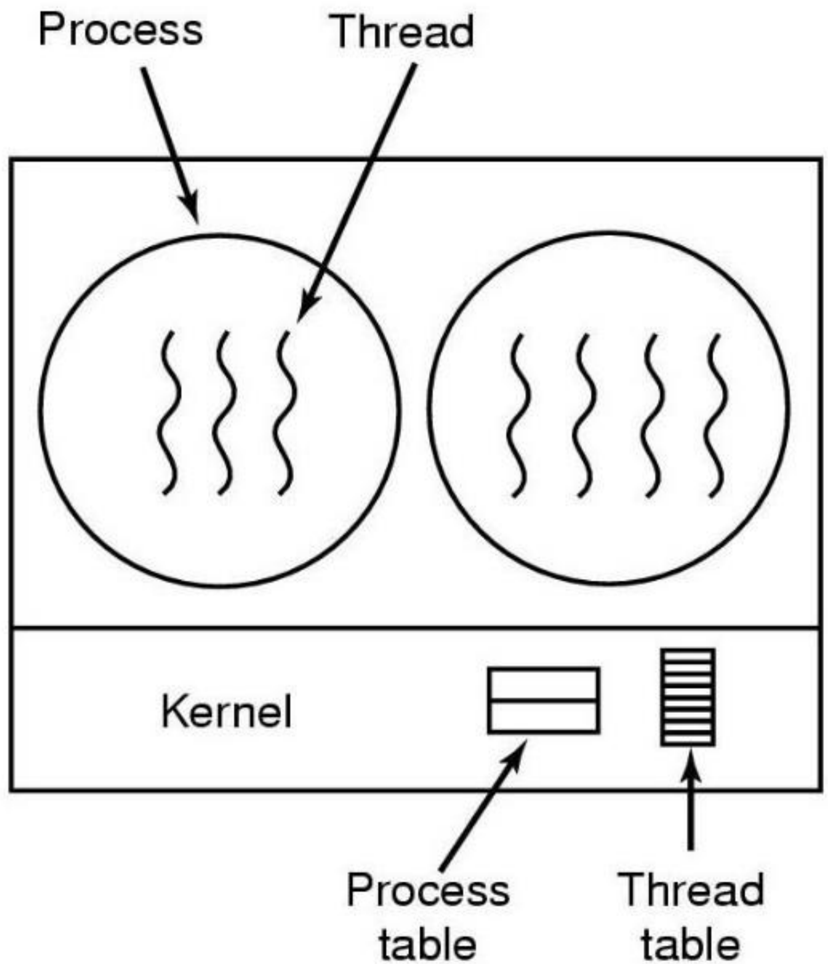
⊕ Overhead↓

→ User thread creation is good

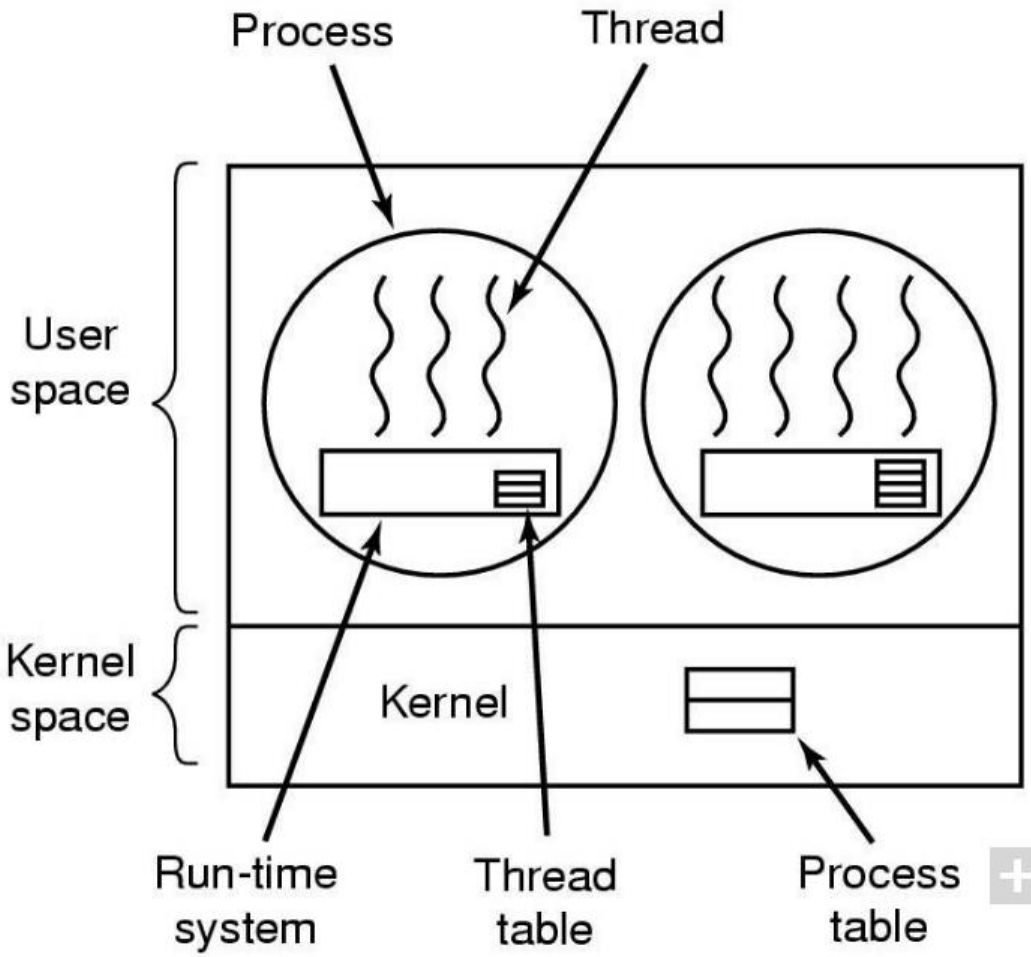
I/O

every thread will be blocked

Implementing threads in Kernel



Implementing threads in user space



Where is thread scheduler?

User thread library

What is performance benefit?

Overhead ↓ (No need syscall)

What is a main problem?

- All will Blocked when syscall, I/O
- 한 프로세스 in 여러 쓰레드 동시에 못돌림.
- User level에서 쓰레드 간 프로텍션 필요

Kernel thread limitations

- Every thread operation must go through kernel
  - create, exit, join, synchronize, or switch for any reason
  - On my laptop: syscall takes 100 cycles, fn call 5 cycles
  - Result: threads 10x-30x slower when implemented in kernel

→ Thread context switch overhead↑ (TCB, PCB control)

- One-size fits all thread implementation
  - Kernel threads must please all people
  - Maybe pay for fancy features (priority, etc.) you don't need

- General heavy-weight memory requirements
  - e.g., requires a fixed-size stack within kernel
  - other data structures designed for heavier-weight processes

TCB, PCB 모두 OS가 관리

User-level thread limitations

- Can't take advantage of multiple CPUs or cores
- User-level threads are invisible to the OS
  - They are not well integrated with the OS
- As a result, the OS can make poor decisions
  - Scheduling a process with idle threads
  - A blocking system call blocks all threads
    - Can replace read to handle network connections, but usually OSes don't let you do this for disk
  - Unscheduling a process with a thread holding a lock
- How to solve this?
  - Communication between the kernel and the user-level thread manager (Windows 8) [Scheduler Activation]

Summary

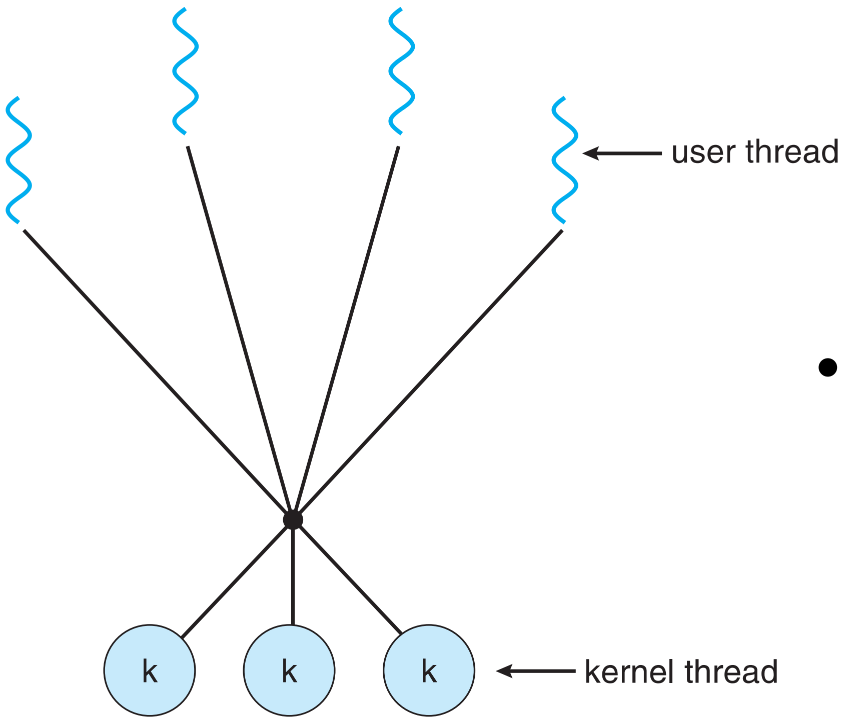
- Kernel-level threads
  - Integrated with OS (informed scheduling)
  - Slower to create, manipulate, synchronize
- User-level threads
  - Faster to create, manipulate, synchronize
  - Not integrated with OS (uninformed scheduling)
- Understanding their differences is important
  - Correctness, performance

Kernel and User threads

- Or use both kernel and user-level threads
  - Can associate a user-level thread with a kernel-level thread
  - Or, multiplex user-level threads on top of kernel-level threads

- Java Virtual Machine (JVM) (also C#, others)
  - Java threads are user-level threads
  - On older Unix, only one “kernel thread” per process
    - Multiplex all Java threads on this one kernel thread
  - On modern OSes
    - Can multiplex Java threads on multiple kernel threads
    - Can have more Java threads than kernel threads

User threads on Kernel threads



- User threads implemented on kernel threads
  – Multiple kernel-level threads per process
- Sometimes called n : m threading
  – Have n user threads per m kernel threads (user-level threads are n : 1, kernel threads 1 : 1)