

CS162
Operating Systems and
Systems Programming
Lecture 3

Abstractions I: Threads and Processes
A quick, programmer's viewpoint

October 5th, 2024

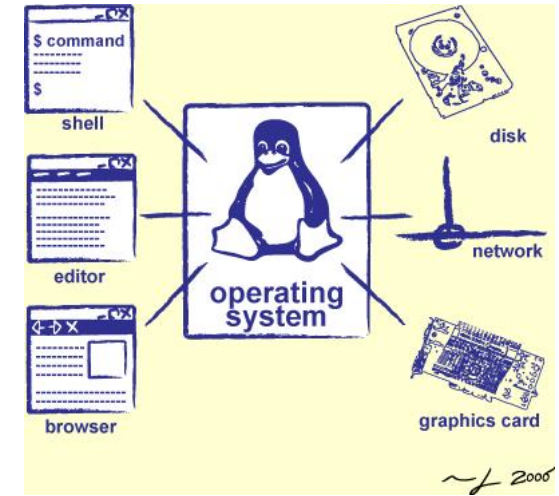
Prof. Ion Stoica

<http://cs162.eecs.Berkeley.edu>

Slides courtesy of David Culler, Natacha Crooks, Anthony D. Joseph, John Kubiawicz, Aj
Shankar, Alex Aiken, Eric Brewer, Ras Bodik, Doug Tygar, and David Wagner.

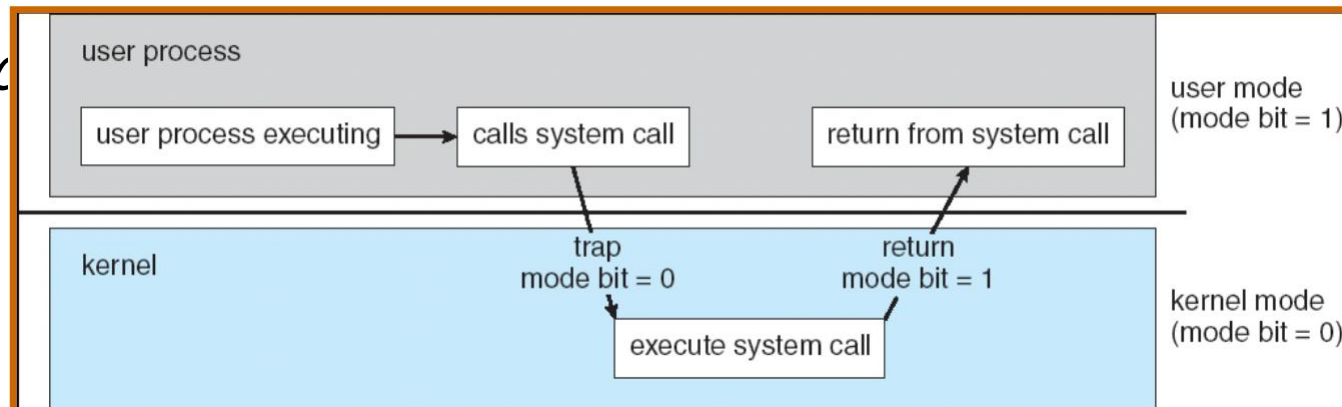
Goals for Today: The Thread Abstraction

- *What threads are*
 - And what they are not
- *Why threads are useful (motivation)*
- *How to write a program using threads*
- *Alternatives to using threads*

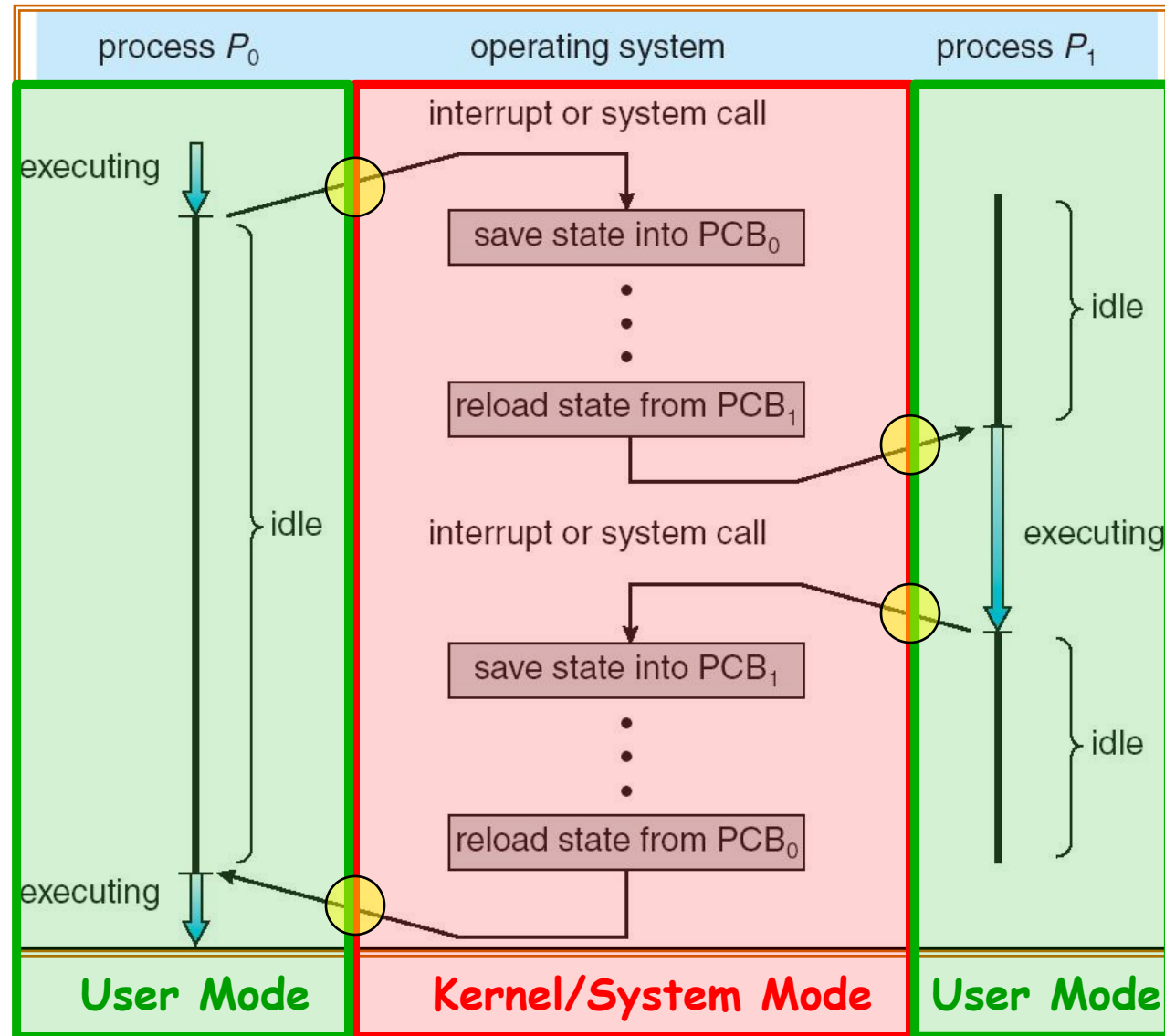


Recall: Dual Mode Operation

- Processes (i.e., programs you run) execute in **user mode**
 - To perform privileged actions, processes request services from the OS kernel
 - Carefully controlled transition from user to kernel mode
- Kernel executes in **kernel mode**
 - Performs privileged actions to support running processes
 - ... and configures hardware to properly protect them (e.g., address translation)
- Carefully controlled transitions between user mode and kernel mode
 - System call



Adding Protection: CPU Switch From Process A to Process B

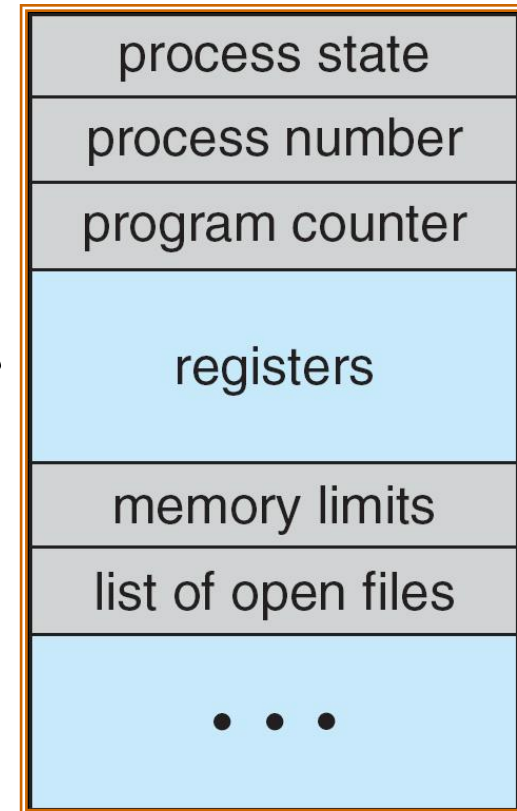


Running Many Programs

- We have the basic *mechanism* to:
 - switch between user processes and the kernel,
 - the kernel can switch among user processes,
 - protect OS from user processes and processes from each other
- Questions:
 - How do we represent user processes in the OS?
 - How do we decide which user process to run?
 - How do we pack up the process and set it aside?
 - How do we get a stack and heap for the kernel?
 - Aren't we wasting a lot of memory?

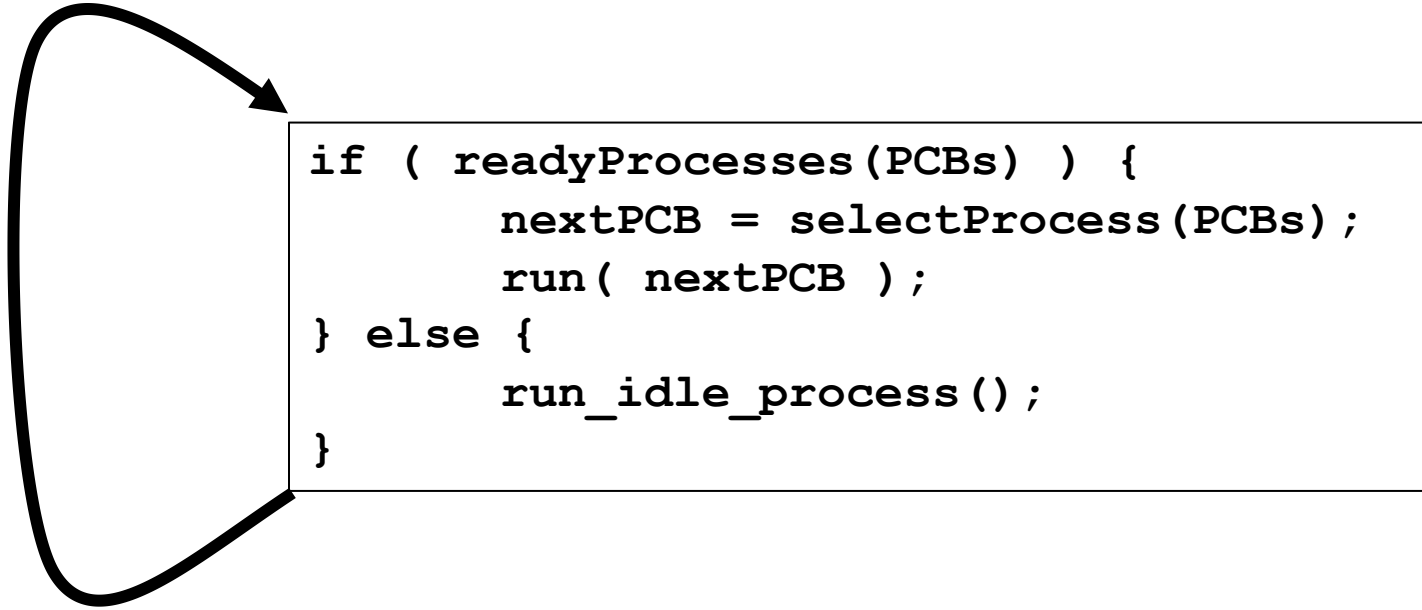
Multiplexing Processes: The Process Control Block

- Kernel represents each process with a process control block (PCB)
 - Status (running, ready, blocked, ...)
 - Register state (when not ready)
 - Process ID (PID), User, Executable, Priority, ...
 - Execution time, ...
 - Memory space, translation, ...
- Kernel Scheduler maintains a data structure containing the PCBs
 - Give out CPU to different processes
 - This is a Policy Decision
- Give out non-CPU resources
 - Memory/IO
 - Another policy decision



Process
Control
Block

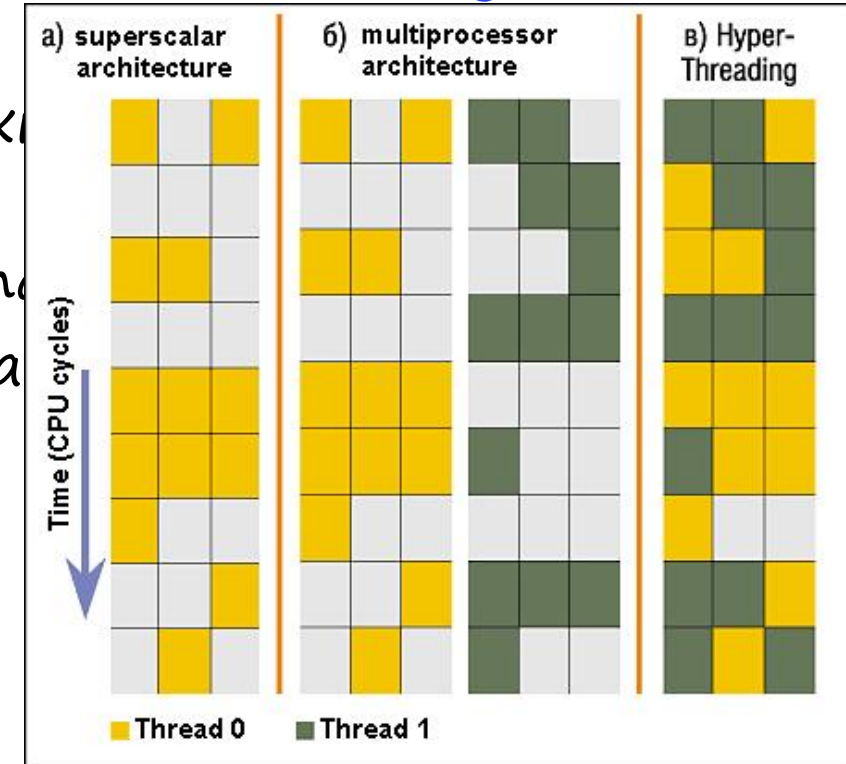
Scheduler



- Scheduling: Mechanism for deciding which processes/threads receive hardware CPU time, when, and for how long
- Lots of different scheduling policies provide ...
 - Fairness or
 - Realtime guarantees or
 - Latency optimization or ..

Simultaneous MultiThreading/Hyperthreading

- Hardware scheduling technique
 - Avoids software overhead of multiplexing
 - Superscalar processors can execute multiple instructions that are independent
 - Hyperthreading duplicates register state to make a second “thread,” allowing more instructions to run.
- Can schedule each thread as if were separate CPU
 - But, sub-linear speedup!



Colored blocks show instructions executed

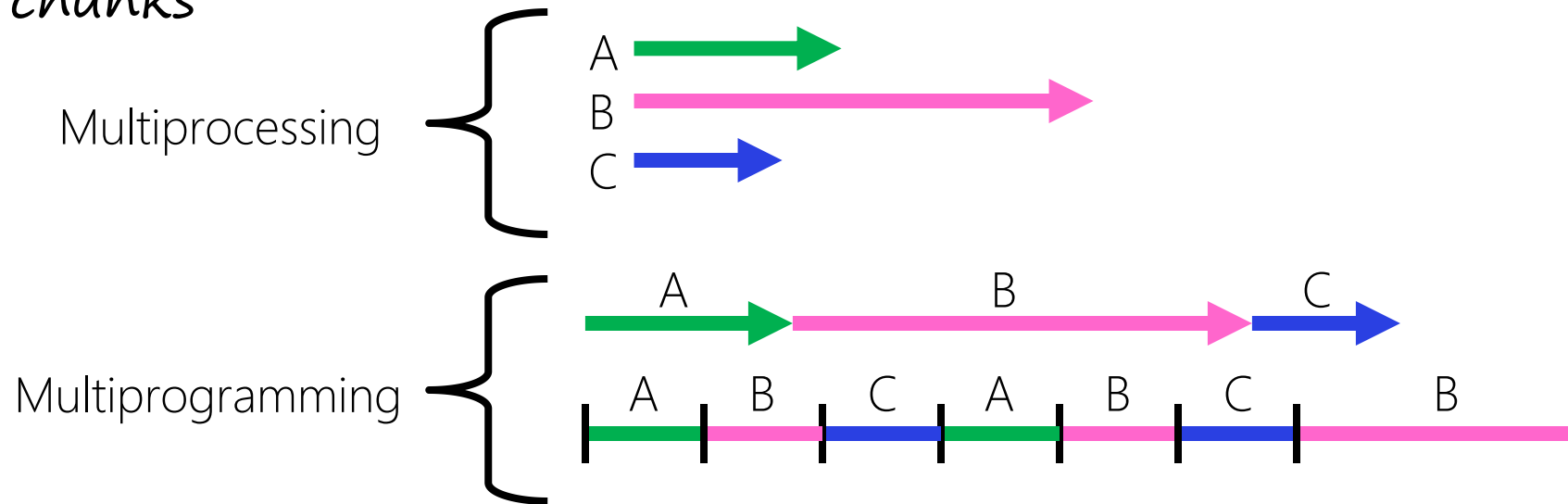
- Original technique called “Simultaneous Multithreading”
 - <http://www.cs.washington.edu/research/smt/index.html>
 - SPARC, Pentium 4/Xeon (“Hyperthreading”), Power 5

More About Threads: What are they?

- Definition from before: *A single unique execution context*
 - Describes its representation
- It provides the abstraction of: *A single execution sequence that represents a separately schedulable task*
 - Also a valid definition!
- Threads are a mechanism for concurrency (overlapping execution)
 - However, they can also run in parallel (simultaneous execution)
- Protection is an orthogonal concept
 - A protection domain can contain one thread or many

Multiprocessing vs. Multiprogramming

- Some Definitions:
 - Multiprocessing: Multiple CPUs(cores)
 - Multiprogramming: Multiple jobs/processes
 - Multithreading: Multiple threads/processes
- What does it mean to run two threads concurrently?
 - Scheduler is free to run threads in any order and interleaving
 - Thread may run to completion or time-slice in big chunks or small chunks



Concurrency is not Parallelism

- *Concurrency is about handling multiple things at once*
- *Parallelism is about doing multiple things simultaneously*
- *Example: Two threads on a single-core system...*
 - *... execute concurrently ...*
 - *... but not in parallel*
- *Each thread handles or manages a separate thing or task...*
- *But those tasks are not necessarily executing simultaneously!*

Silly Example for Threads

- *Imagine the following program:*

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

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

Adding Threads

- Version of program with threads (loose syntax):

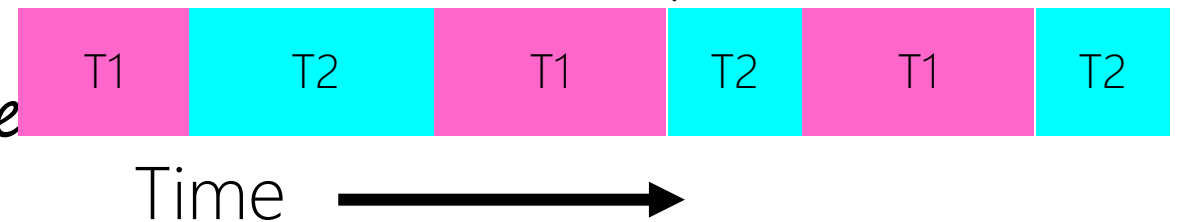
```
main() {  
    create_thread(ComputePI, "pi.txt");  
    create_thread(PrintClassList, "classlist.txt");  
}
```

→ T1*

→ T2

- create_thread: Spawns a new thread running the given procedure
 - Should behave as if another CPU is running the given procedure

- Now, you would actually see the class list



*we use thread (T) and vCPU interchangeable. vCPU suggests that the OS provides virtual CPU abstraction to the thread.

More Practical Motivation: Compute/I/O overlap

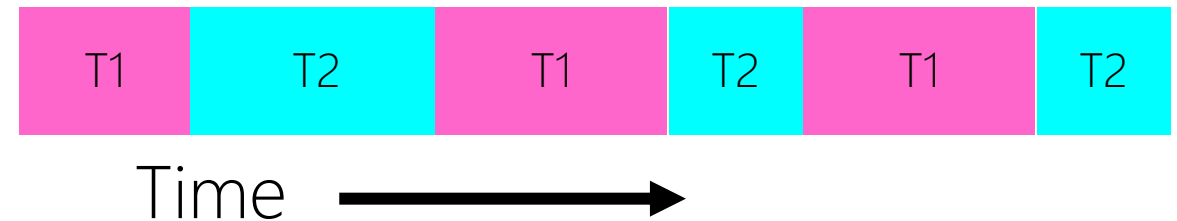
Back to Jeff Dean's
“Numbers
Everyone Should
Know”

*Handle I/O in
separate thread,
avoid blocking
other progress*

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

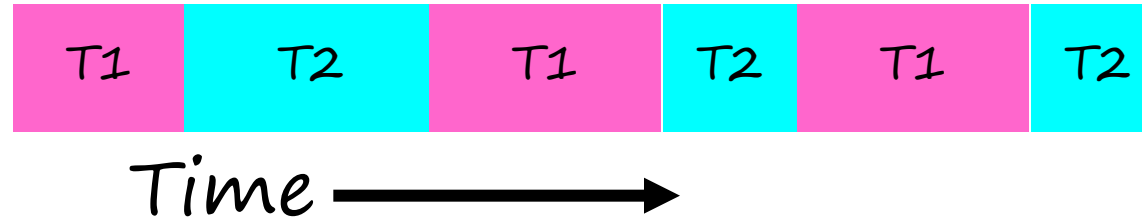
Threads Mask I/O Latency

- A thread is in one of the following three states:
 - RUNNING* – running
 - READY* – eligible to run, but not currently running
 - BLOCKED* – ineligible to run
- If a thread is waiting for an I/O to finish, the OS marks it as *BLOCKED*
- Once the I/O finally finishes, the OS marks it as *READY*

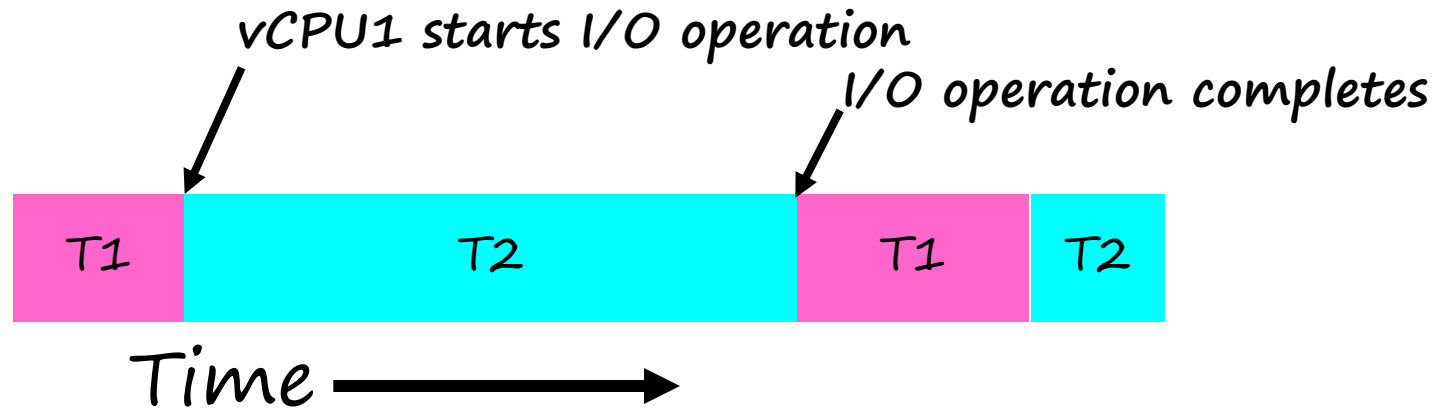


Threads Mask I/O Latency

- If no thread performs I/O:



- If thread 1 performs a blocking I/O operation:



A Better Example for Threads

```
main() {  
    create_thread(ReadLargeFile, "pi.txt");  
    create_thread(RenderUserInterface);  
}
```

- *What is the behavior here?*
 - *Still respond to user input*
 - *While reading file in the background*

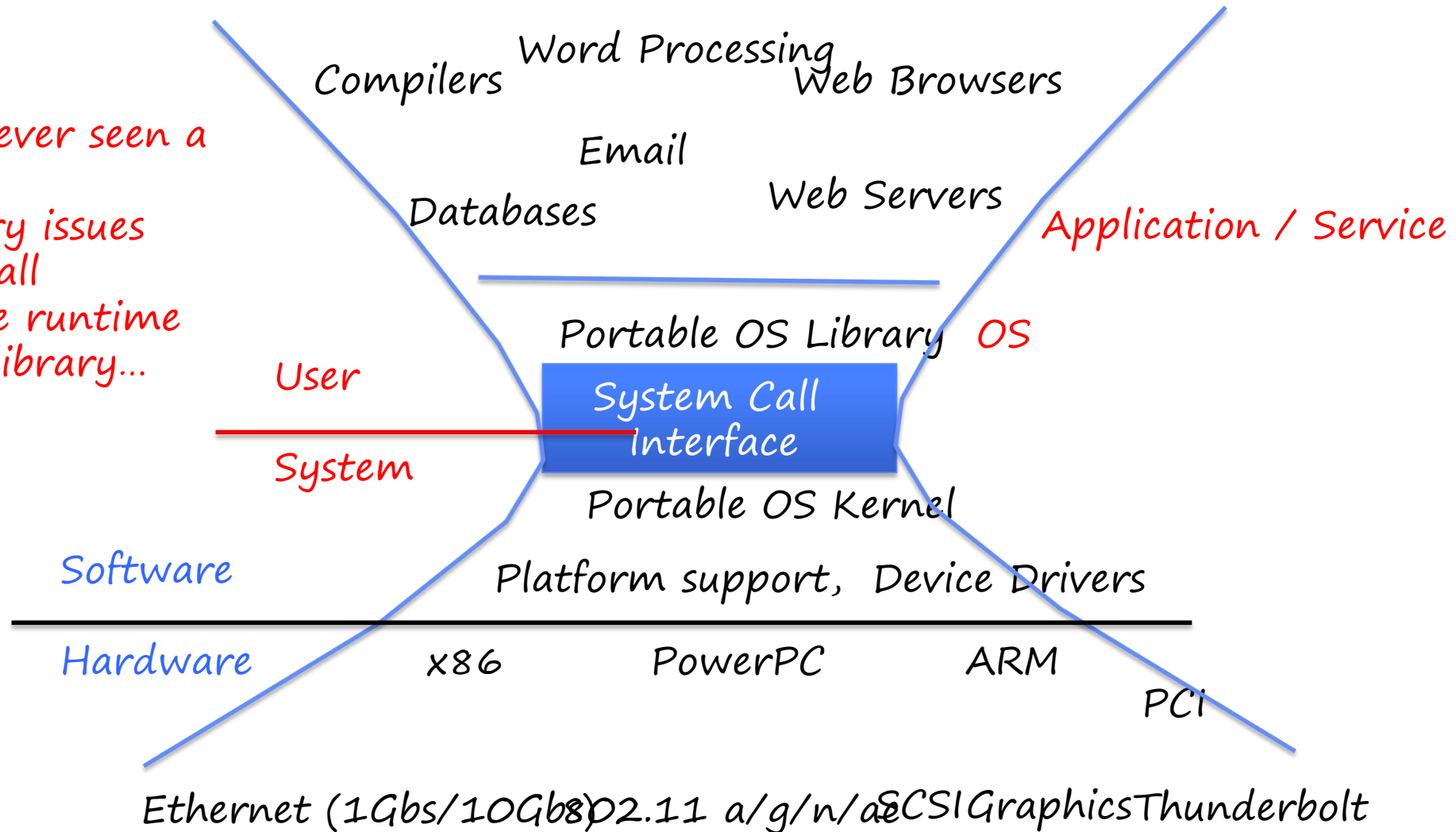
Multithreaded Programs

- You know how to compile a *C* program and run the executable
 - This creates a process that is executing that program
- Initially, this new process has *one thread* in its own address space
 - With code, global variables, etc. as specified in the executable
- Q: How can we make a multithreaded process?
- A: Once the process starts, it issues *system calls* to create new threads
 - These new threads are part of the process: they share its address space

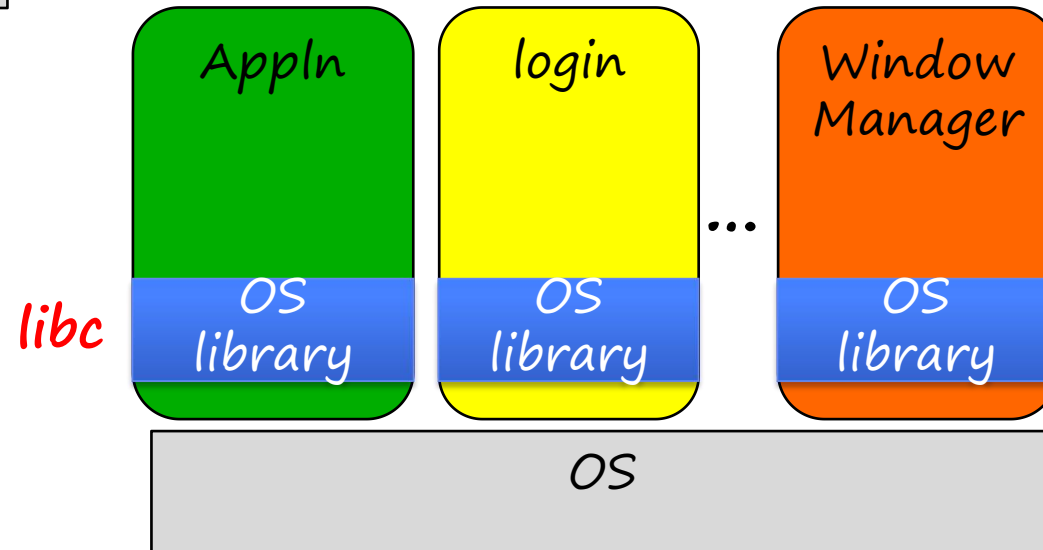
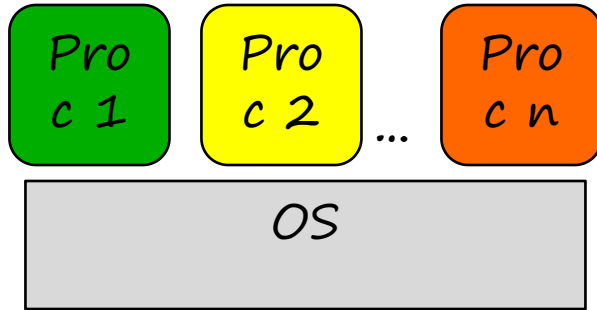
System Calls (“Syscalls”)

“But, I’ve never seen a syscall!”

- OS library issues system call
- Language runtime uses OS library...



OS Library Issues Syscalls



Administrivia: Getting started!

- Sky Lab Office Hours:
 - Tuesday 11–12am, in 465 Soda Hall
- Homework 0: **Due Tomorrow!**
 - Get familiar with the cs162 tools
 - configure your VM, submit via git
 - Practice finding out information:
 - » How to use GDB? How to understand output of unix tools?
 - » We don't assume that you already know everything!
 - » Learn to use “man” (command line), “help” (in gdb, etc), google
- Project 0: **Started two days ago!**
 - Learn about Pintos and how to modify and debug kernel
 - Important for getting started on projects!
- Should be going to sections now – Important information there
 - Any section will do until groups assigned

Administrivia (Con't)

- **THIS Friday is Drop Deadline! HARD TO DROP LATER!**
 - If you know you are going to drop, do so now to leave room for others on waitlist!
 - Why do we do this? So that groups aren't left without members!
- Group sign up via autograder form next week
 - Get finding groups of 4 people ASAP
 - Priority for same section; if cannot make this work, keep same TA
 - Remember: Your TA needs to see you in section!
- Midterm 1: 10/3
 - 7-9PM in person
 - We will say more about material when we get closer...

OS Library API for Threads: *pthread*

Here: the “p” is for “POSIX” which is a part of a standardized API

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,  
                  void *(*start_routine)(void*), void *arg);
```

- thread is created executing *start_routine* with *arg* as its sole argument.
- return is implicit call to *pthread_exit*
- (attr contains info like stack size, scheduling policy, etc)

```
void pthread_exit(void *value_ptr);
```

- terminates the thread and makes *value_ptr* available to any successful join

```
int pthread_join(pthread_t thread, void **value_ptr);
```

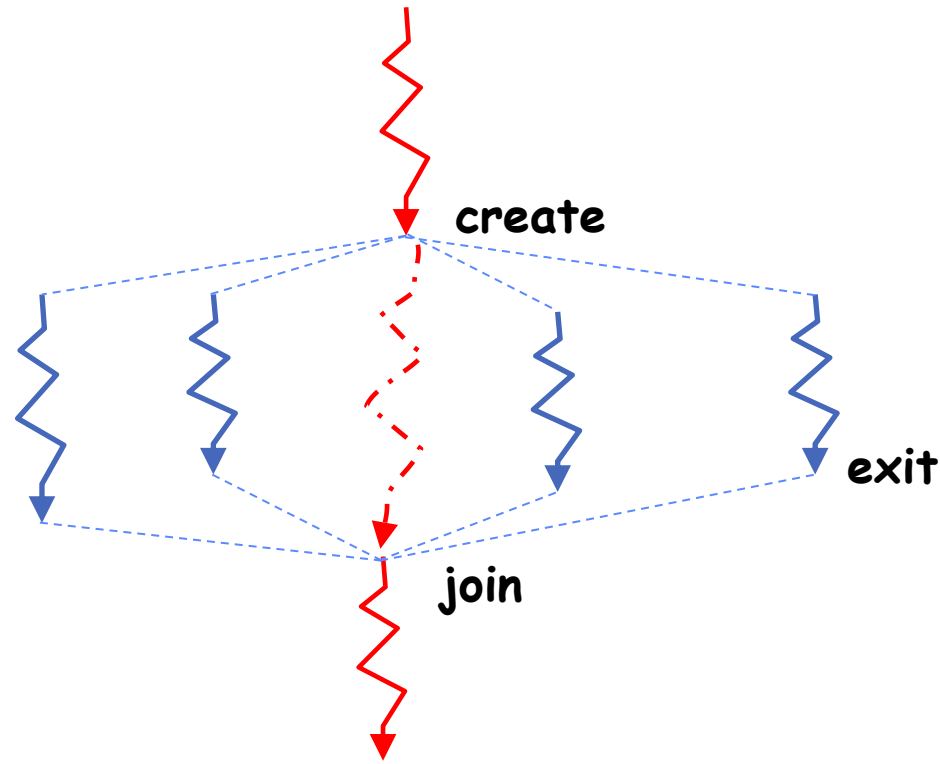
- suspends execution of the calling thread until the target thread terminates.

prompt% man pthread

<https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html>

- On return with a non-NULL *value_ptr* the value passed to *pthread_exit()* by the terminating thread is made available in

New Idea: Fork-Join Pattern



- *Main thread creates (forks) collection of sub-threads passing them args to work on...*
- *... and then joins with them, collecting results.*

pThreads Example

- How many threads are in this program?
- What function does each thread run?

```
((base) CullerMac19:code04 culler$ ./pthread 4
Main stack: 7ffee2c6b6b8, common: 10cf95048 (162)
Thread #1 stack: 70000d83bef8 common: 10cf95048 (162)
Thread #3 stack: 70000d941ef8 common: 10cf95048 (164)
Thread #2 stack: 70000d8b8ef8 common: 10cf95048 (165)
Thread #0 stack: 70000d7b8ef8 common: 10cf95048 (163)
```

- Does the main thread join with the threads in the same order that they were created?
 - Yes: Loop calls Join in thread order
- Do the threads exit in the same order they were created?
 - No: Depends on scheduling order!
- Would the result change if run again?
 - Yes: Depends on scheduling order!
- Is this code safe/correct???
- No – threads share a variable that is used without locking and there is

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
```

```
int common = 162;
```

```
void *threadfun(void *threadid)
{
    long tid = (long)threadid;
    printf("Thread #%lx stack: %lx common: %lx (%d)\n", tid,
           (unsigned long) &tid, (unsigned long) &common, common++);
    pthread_exit(NULL);
}
```

```
int main (int argc, char *argv[])
{
    long t;
    int nthreads = 2;
    if (argc > 1) {
        nthreads = atoi(argv[1]);
    }
    pthread_t *threads = malloc(nthreads*sizeof(pthread_t));
    printf("Main stack: %lx, common: %lx (%d)\n",
           (unsigned long) &t, (unsigned long) &common, common);
    for(t=0; t<nthreads; t++){
        int rc = pthread_create(&threads[t], NULL, threadfun, (void *)t);
        if (rc){
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }

    for(t=0; t<nthreads; t++){
        pthread_join(threads[t], NULL);
    }
    pthread_exit(NULL);
} /* last thing in the main thread */
```

Peeking Ahead: System Call Example

- *What happens when pthread_create(...) is called in a process?*

Library:

```
int pthread_create(...) {  
    Do some work like a normal fn...  
  
    asm code ... syscall # into %eax  
    put args into registers %ebx, ...  
    special trap instruction
```

Kernel:

```
    get args from regs  
    dispatch to system func  
    Do the work to spawn the new thread  
    Store return value in %eax
```

```
    get return values from regs  
    Do some more work like a normal fn...  
};
```

Thread State

- State shared by all threads in process/address space
 - Content of memory (global variables, heap)
 - I/O state (file descriptors, network connections, etc)
- State “private” to each thread
 - Kept in **TCB** \equiv **Thread Control Block**
 - CPU registers (including, program counter)
 - Execution stack
- Execution Stack
 - Parameters, temporary variables
 - Return PCs are kept while called procedures are executing

Shared vs. Per-Thread State

Shared State

Heap

Global Variables

Code

Per-Thread State

Thread Control Block (TCB)

Stack Information

Saved Registers

Thread Metadata

Stack

Per-Thread State

Thread Control Block (TCB)

Stack Information

Saved Registers

Thread Metadata

Stack

Execution Stack Example

```
    A(int tmp) {  
A:    if (tmp<2)  
A+1:    B();  
A+2:    printf(tmp);  
    }  
    B() {  
B:    C();  
B+1: }  
    C() {  
C:    A(2);  
C+1: }  
    A(1);  
exit:
```

- *Stack holds temporary results*
- *Permits recursive execution*
- *Crucial to modern languages*

Execution Stack Example

```

A(int tmp) {
A:   if (tmp<2)
A+1:   B();
A+2:   printf(tmp);
      }
      B() {
B:     C();
B+1:   }
      C() {
C:     A(2);
C+1:   }
      A(1);
exit:

```

Stack
Pointer

A: tmp=1
ret=exit

- *Stack holds temporary results*
- *Permits recursive execution*
- *Crucial to modern languages*

Execution Stack Example

```

A(int tmp) {
A:  if (tmp<2)
A+1:    B();
A+2:    printf(tmp);
      }
      B() {
B:    C();
B+1:  }
      C() {
C:    A(2);
C+1:  }
      A(1);
exit:

```

Stack
Pointer

A: tmp=1
ret=exit

- *Stack holds temporary results*
- *Permits recursive execution*
- *Crucial to modern languages*

Execution Stack Example

```
    A(int tmp) {  
A:    if (tmp<2)  
A+1:    B();  
A+2:    printf(tmp);  
    }  
    B() {  
B:    C();  
B+1: }  
    C() {  
C:    A(2);  
C+1: }  
    A(1);  
exit:
```

Stack
Pointer

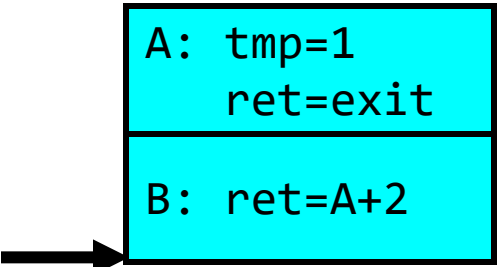
A: tmp=1
ret=exit

- *Stack holds temporary results*
- *Permits recursive execution*
- *Crucial to modern languages*

Execution Stack Example

```
A(int tmp) {  
A:   if (tmp<2)  
A+1:   B();  
A+2:   printf(tmp);  
      }  
      B() {  
B:     C();  
B+1:   }  
      C() {  
C:     A(2);  
C+1:   }  
      A(1);  
exit:
```

Stack
Pointer



A: tmp=1
ret=exit
B: ret=A+2

- *Stack holds temporary results*
- *Permits recursive execution*
- *Crucial to modern languages*

Execution Stack Example

```

A(int tmp) {
A:   if (tmp<2)
A+1:   B();
A+2:   printf(tmp);
      }
      B() {
B:   C();
B+1: }
      C() {
C:   A(2);
C+1: }
      A(1);
exit:

```

Stack
Pointer

A: tmp=1 ret=exit
B: ret=A+2

- *Stack holds temporary results*
- *Permits recursive execution*
- *Crucial to modern languages*

Execution Stack Example

```

A(int tmp) {
A:   if (tmp<2)
A+1:   B();
A+2:   printf(tmp);
      }

      B() {
B:     C();
B+1:   }

      C() {
C:     A(2);
C+1:   }

      A(1);
exit:

```

Stack
Pointer

A: tmp=1
ret=exit

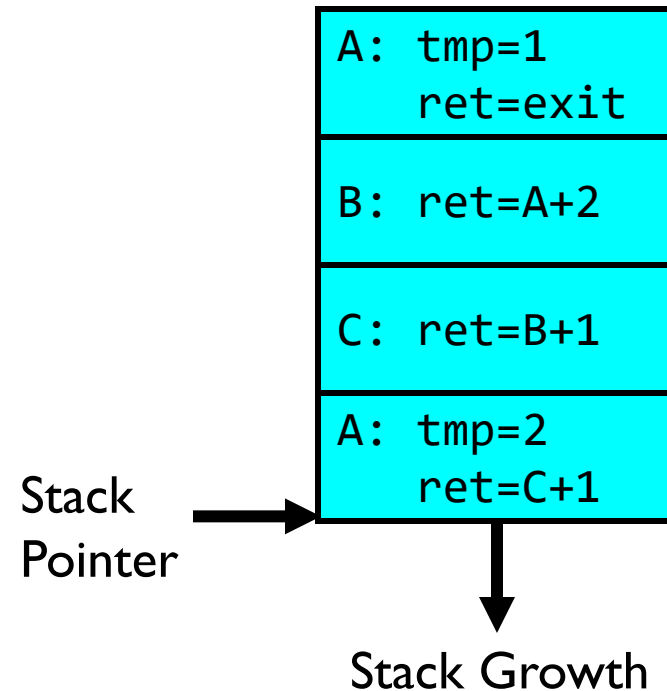
B: ret=A+2

C: ret=B+1

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

```
A(int tmp) {  
A:  if (tmp<2)  
A+1:    B();  
A+2:    printf(tmp);  
    }  
    B() {  
B:    C();  
B+1: }  
    C() {  
C:    A(2);  
C+1: }  
    A(1);  
exit:
```



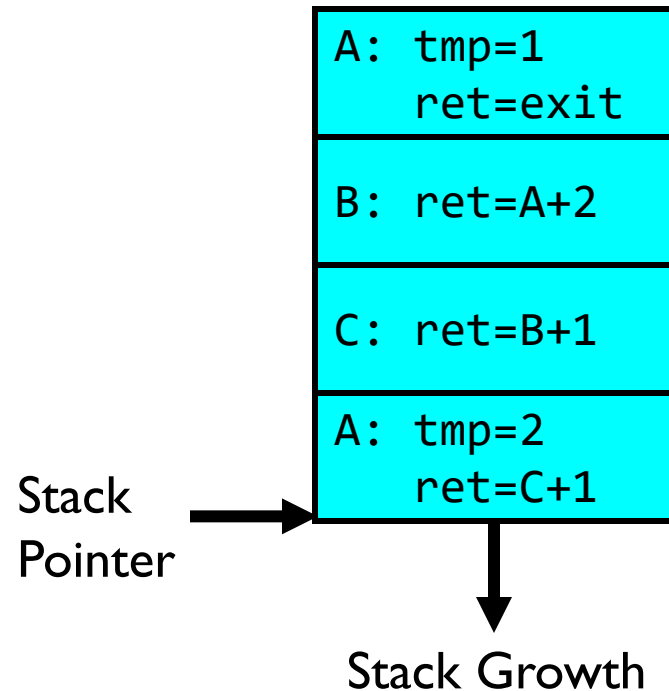
- *Stack holds temporary results*
- *Permits recursive execution*
- *Crucial to modern languages*

Execution Stack Example

```

A(int tmp) {
A:   if (tmp<2)
A+1:   B();
A+2:   printf(tmp);
      }
      B() {
B:     C();
B+1:   }
      C() {
C:     A(2);
C+1:   }
      A(1);
exit:

```



Output: **>2**

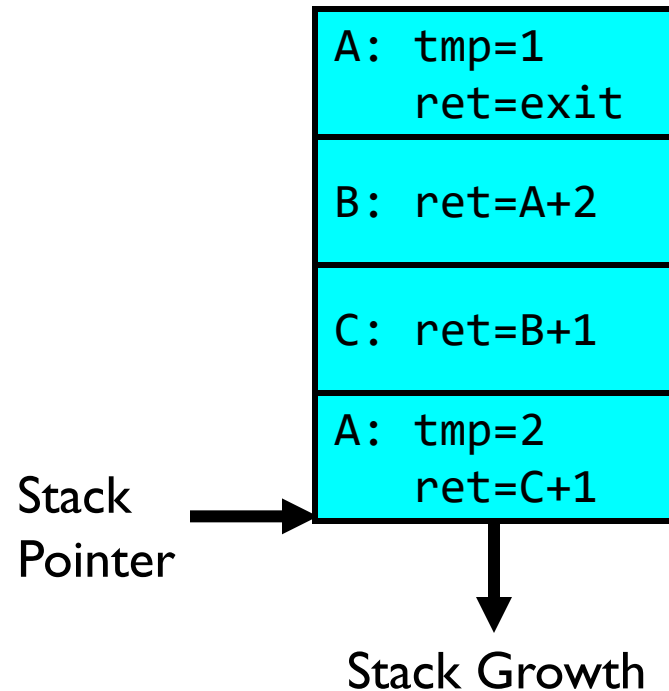
- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

```

A(int tmp) {
A:   if (tmp<2)
A+1:   B();
A+2:   printf(tmp);
      }
      B() {
B:     C();
B+1:   }
      C() {
C:     A(2);
C+1:   }
      A(1);
exit:

```



Output: **>2**

- *Stack holds temporary results*
- *Permits recursive execution*
- *Crucial to modern languages*

Execution Stack Example

```

A(int tmp) {
A:   if (tmp<2)
A+1:   B();
A+2:   printf(tmp);
      }

      B() {
B:     C();
B+1:   }

      C() {
C:     A(2);
C+1:   }
      A(1);
exit:

```

Stack
Pointer

A: tmp=1 ret=exit
B: ret=A+2
C: ret=B+1

Output: **>2**

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

```

A(int tmp) {
A:   if (tmp<2)
A+1:   B();
A+2:   printf(tmp);
      }
      B() {
B:     C();
B+1:   }
      C() {
C:     A(2);
C+1:   }
      A(1);
exit:

```

Stack
Pointer

A: tmp=1
ret=exit
B: ret=A+2

Output: **>2**

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

```

A(int tmp) {
A:   if (tmp<2)
A+1:   B();
A+2:   printf(tmp);
      }
      B() {
B:     C();
B+1:   }
      C() {
C:     A(2);
C+1:   }
      A(1);
exit:

```

Stack
Pointer

A: tmp=1
ret=exit

Output: **>2 1**

- *Stack holds temporary results*
- *Permits recursive execution*
- *Crucial to modern languages*

Execution Stack Example

```

A(int tmp) {
A:   if (tmp<2)
A+1:   B();
A+2:   printf(tmp);
      }
      B() {
B:     C();
B+1:   }
      C() {
C:     A(2);
C+1:   }
      A(1);
exit:

```

Stack
Pointer

A: tmp=1
ret=exit

Output: **>2 1**

- *Stack holds temporary results*
- *Permits recursive execution*
- *Crucial to modern languages*

Execution Stack Example

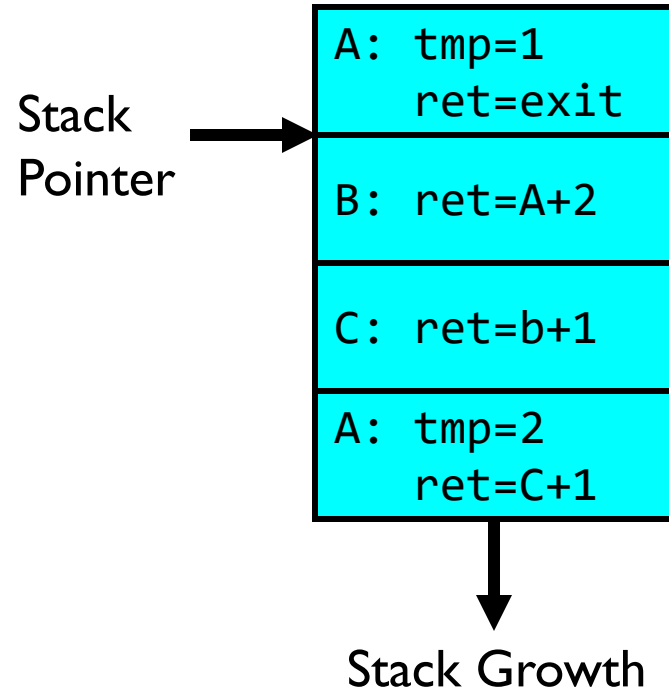
```
A(int tmp) {  
    if (tmp<2)  
        B();  
    printf(tmp);  
}  
  
B() {  
    C();  
}  
  
C() {  
    A(2);  
}  
A(1);
```

Output: **>2 1**

- *Stack holds temporary results*
- *Permits recursive execution*
- *Crucial to modern languages*

Execution Stack Example

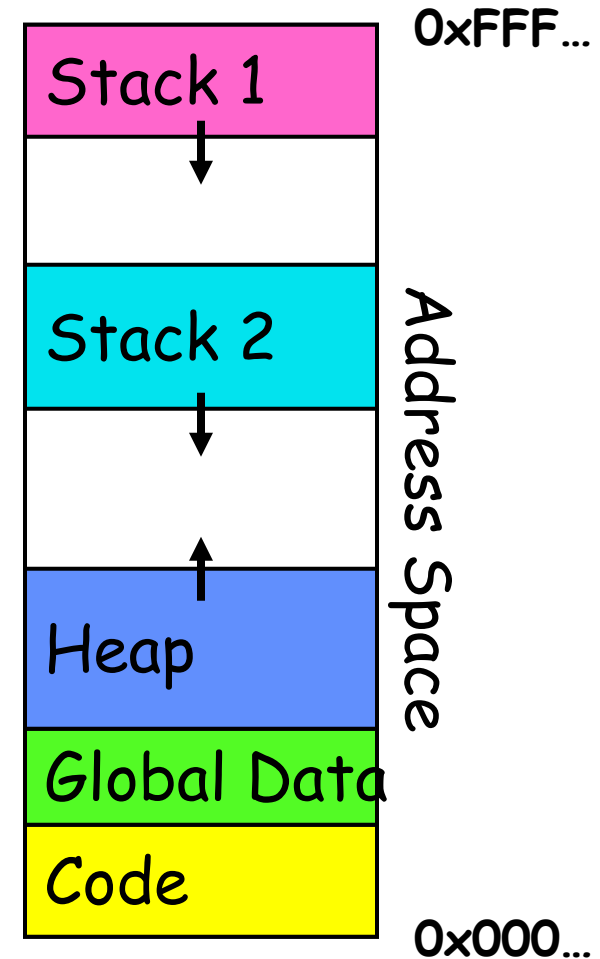
```
A(int tmp) {  
    if (tmp<2)  
        B();  
    printf(tmp);  
}  
  
B() {  
    C();  
}  
  
C() {  
    A(2);  
}  
  
A(1);
```



- *Stack holds temporary results*
- *Permits recursive execution*
- *Crucial to modern languages*

Memory Layout with Two Threads

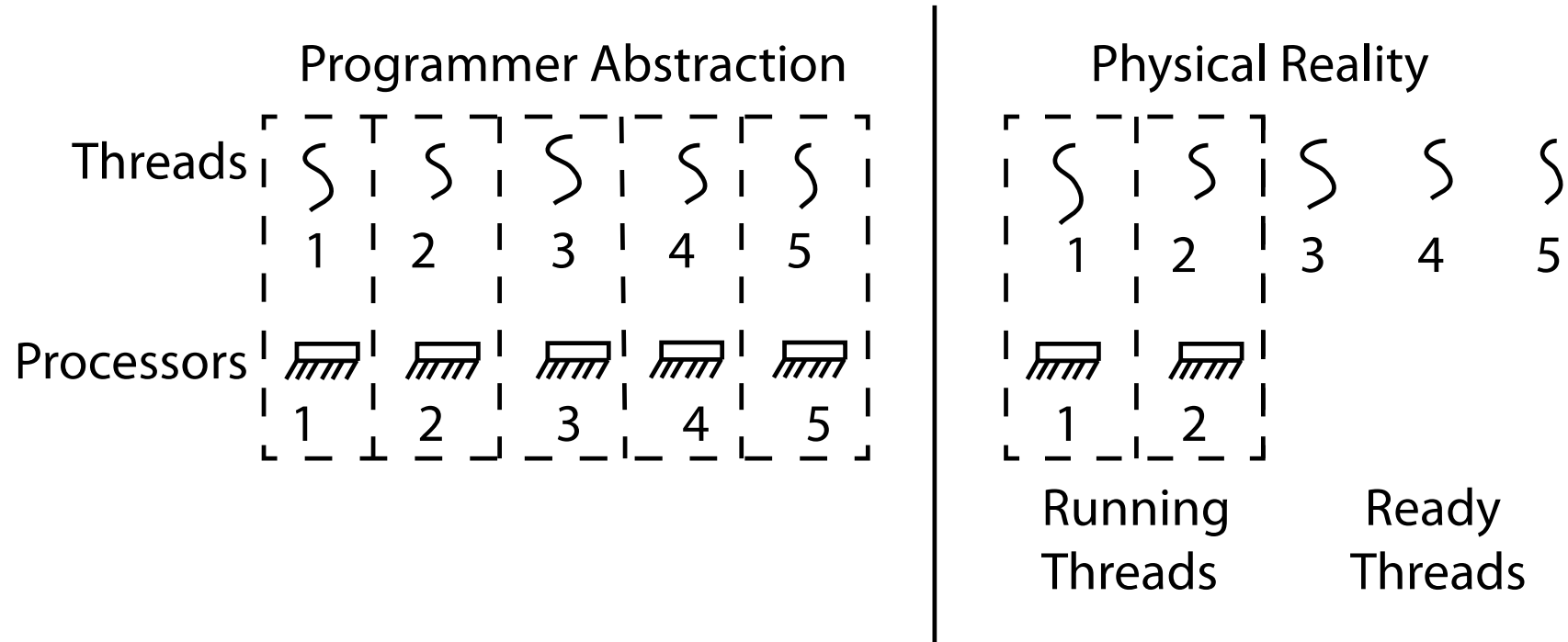
- Two sets of CPU registers
- Two sets of Stacks
- Issues:
 - How do we position stacks relative to each other?
 - What maximum size should we choose for the stacks?
 - What happens if threads violate this?
 - How might you catch violations?



INTERLEAVING AND NONDETERMINISM

(The beginning of a long discussion!)

Thread Abstraction



- *Illusion: Infinite number of processors*
- *Reality: Threads execute with variable “speed”*
 - *Programs must be designed to work with any schedule*

Programmer vs. Processor View

Programmer's View

.
. .
.
 $x = x + 1;$
 $y = y + x;$
 $z = x + 5y;$
. .
. .
. .

Possible Execution #1

.
. .
.
 $x = x + 1;$
 $y = y + x;$
 $z = x + 5y;$
. .
. .
. .

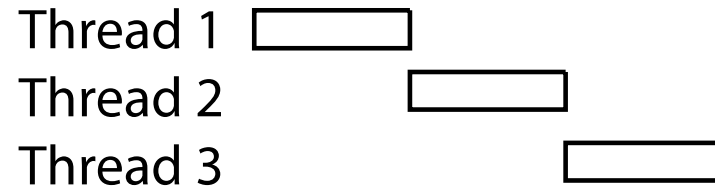
Possible Execution #2

.
. .
.
 $x = x + 1$
.....
thread is suspended
other thread(s) run
thread is resumed
.....
 $y = y + x$
 $z = x + 5y$

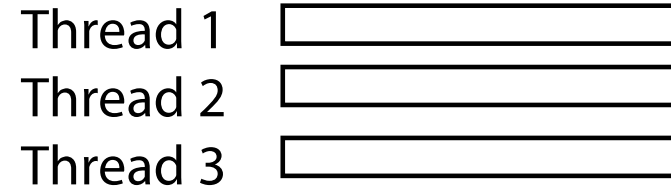
Possible Execution #3

.
. .
.
 $x = x + 1$
 $y = y + x$
.....
thread is suspended
other thread(s) run
thread is resumed
.....
 $z = x + 5y$

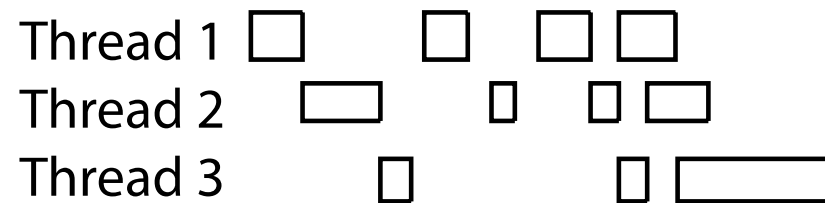
Possible Executions



a) One execution



b) Another execution



c) Another execution

Correctness with Concurrent Threads

- *Non-determinism:*
 - Scheduler can run threads in any order
 - Scheduler can switch threads at any time
 - This can make testing very difficult
- *Independent Threads*
 - No state shared with other threads
 - Deterministic, reproducible conditions
- *Cooperating Threads*
 - Shared state between multiple threads
- *Goal: Correctness by Design*

Race Conditions: Example I

- Initially $x == 0$ and $y == 0$

Thread A

$x = 1;$

Thread B

$y = 2;$

- What are the possible values of x below after all threads finish?
- Must be **1**. Thread B does not interfere

Race Conditions: Example 2

- Initially $x == 0$ and $y == 0$

Thread A

$x = y + 1;$

Thread B

$y = 2;$

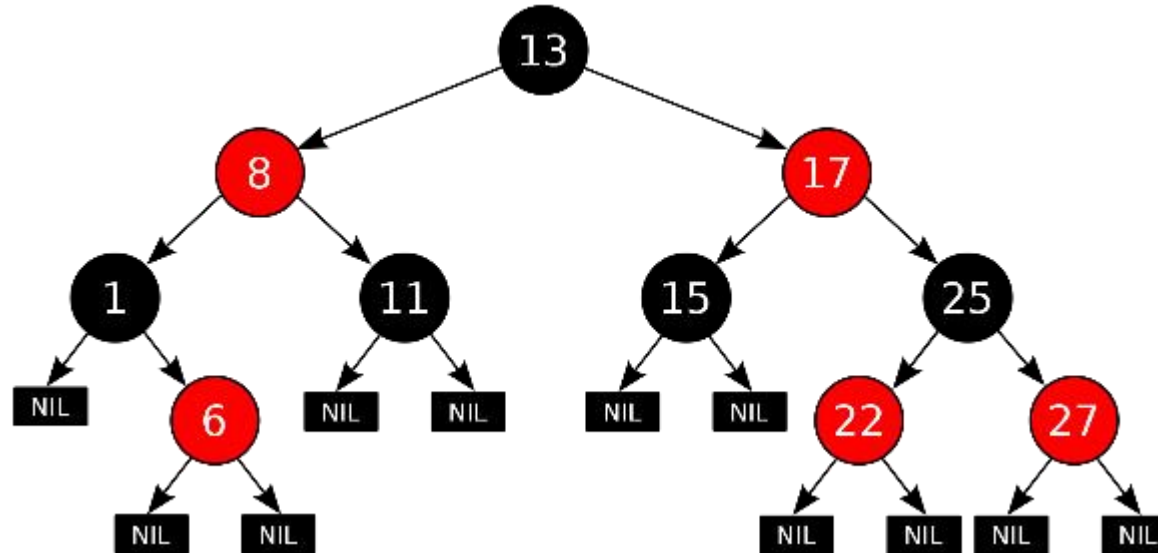
$y = y * 2;$

- What are the possible values of x below?
- 1 or 3 or 5 (non-deterministically)
- Race Condition: Thread A races against Thread B!*

Example: Shared Data Structure

Thread A

Insert(3)



Thread B

Insert(4)

Get(6)

Tree-Based Set Data Structure

How do we make sure this executes correctly?

Relevant Definitions

- **Synchronization**: Coordination among threads, usually regarding shared data
- **Mutual Exclusion**: Ensuring only one thread does a particular thing at a time (one thread *excludes* the others)
 - Type of synchronization
- **Critical Section**: Code exactly one thread can execute at once
 - Result of mutual exclusion
- **Lock**: An object only one thread can hold at a time
 - Provides mutual exclusion

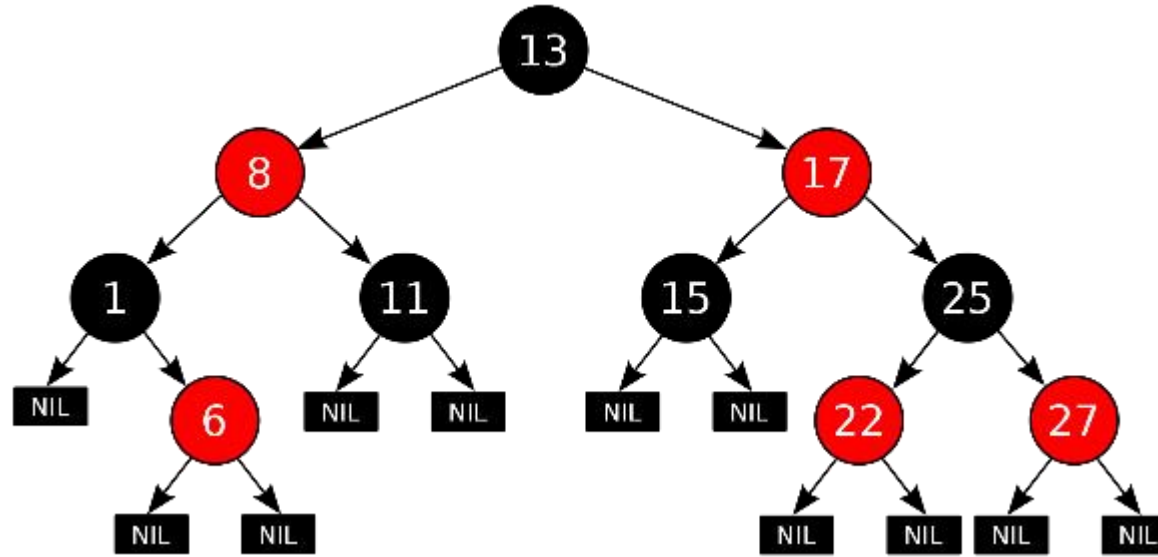
Locks

- Locks provide two *atomic* operations:
 - *Lock.acquire()* – wait until lock is free; then mark it as busy
 - » After this returns, we say the calling thread *holds* the lock
 - *Lock.release()* – mark lock as free
 - » Should only be called by a thread that currently holds the lock
 - » After this returns, the calling thread no longer holds the lock
- For now, don't worry about how to implement locks!
 - We'll cover that in substantial depth later on in the class

Thread A

Insert(3)

- Lock.acquire()
- Insert 3 into the data structure
- Lock.release()



Thread B

Insert(4)

- Lock.acquire()
- Insert 4 into the data structure
- Lock.release()

Get(6)

- Lock.acquire()
- Check for membership
- Lock.release()

Tree-Based Set Data Structure

OS Library Locks: *pthread*s

```
int pthread_mutex_init(pthread_mutex_t *mutex,  
                        const pthread_mutexattr_t *attr)
```

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

You'll get a chance to use these in Homework 1

Our Example

Critical section

```
int common = 162;
pthread_mutex_t common_lock = PTHREAD_MUTEX_INITIALIZER;

void *threadfun(void *threadid)
{
    long tid = (long)threadid;
    pthread_mutex_lock(&common_lock);
    int my_common = common++;
    pthread_mutex_unlock(&common_lock);

    printf("Thread #%lx stack: %lx common: %lx (%d)\n", tid,
           (unsigned long) &tid,
           (unsigned long) &common, my_common);
    pthread_exit(NULL);
}
```

Conclusion

- *Threads are the OS unit of concurrency*
 - *Abstraction of a virtual CPU core*
 - *Can use pthread_create, etc., to manage threads within a process*
 - They share data → need synchronization to avoid data races
- Processes consist of one or more threads in an address space
 - Abstraction of the machine: execution environment for a program
 - Can use fork, exec, etc. to manage threads within a process
- We saw the role of the OS library
 - Provide API to programs
 - Interface with the OS to request services

Recall: Four Fundamental OS Concepts

- **Thread: Execution Context**
 - Fully describes program state
 - Program Counter, Registers, Execution Flags, Stack
- **Address space (with or w/o translation)**
 - Set of memory addresses accessible to program (for read or write)
 - May be distinct from memory space of the physical machine (in which case programs operate in a virtual address space)
- **Process: an instance of a running program**
 - Protected Address Space + One or more Threads
- **Dual mode operation / Protection**
 - Only the “system” has the ability to access certain resources
 - Combined with translation, isolates programs from each other and the OS from programs