

Operating Systems (Honor Track)

Abstractions 1: Threads and Processes A quick, programmer's viewpoint

Xin Jin

Spring 2022

Recall: Four fundamental OS concepts

- **Thread**
 - Single unique execution context
 - Program Counter, Registers, Execution Flags, Stack
- **Address Space** w/ translation
 - Programs execute in an *address space* that is distinct from the memory space of the physical machine
- **Process**
 - An instance of an executing program is *a process consisting of an address space and one or more threads of control*
- **Dual Mode** operation/protection
 - Only the “system” has the ability to access certain resources
 - The OS and the hardware are protected from user programs and user programs are isolated from one another by *controlling the translation* from program virtual addresses to machine physical addresses

Motivation for Threads

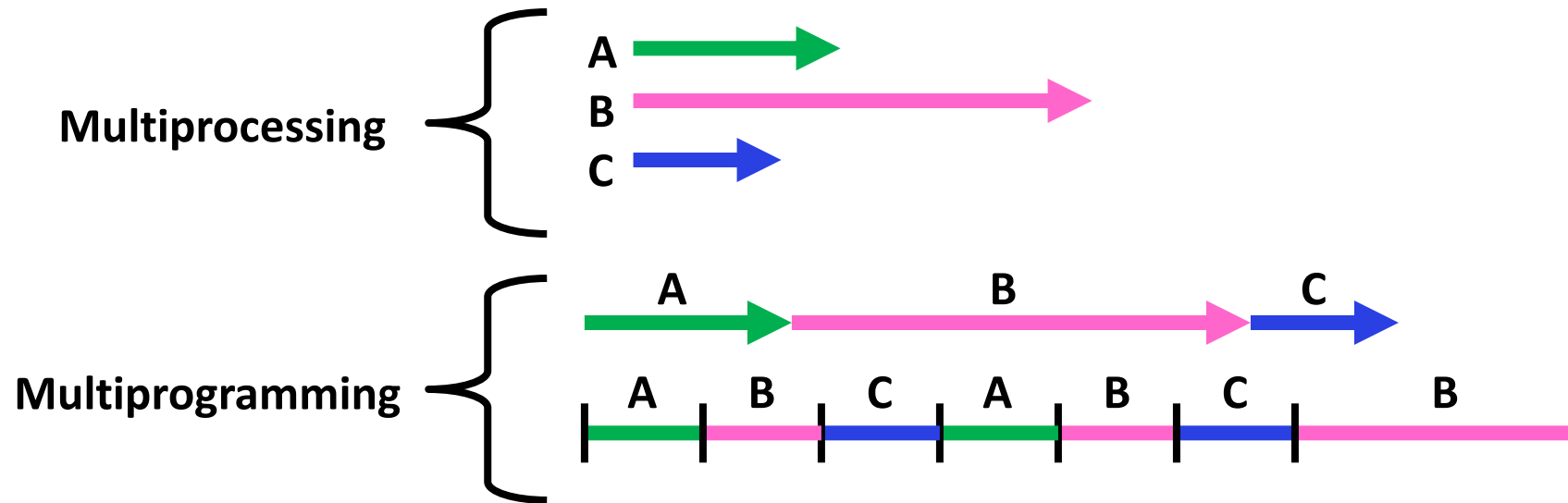
- Operating systems must handle multiple things at once (MTAO)
 - Processes, interrupts, background system maintenance
- Networked servers must handle MTAO
 - Multiple connections handled simultaneously
- Parallel programs must handle MTAO
 - To achieve better performance
- Programs with user interface often must handle MTAO
 - To achieve user responsiveness while doing computation
- Network and disk bound programs must handle MTAO
 - To hide network/disk latency
 - Sequence steps in access or communication

Threads Allow Handling MTAO

- Threads are a unit of *concurrency* provided by the OS
- Each thread can represent one thing or one task

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 (MTAO)
- 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

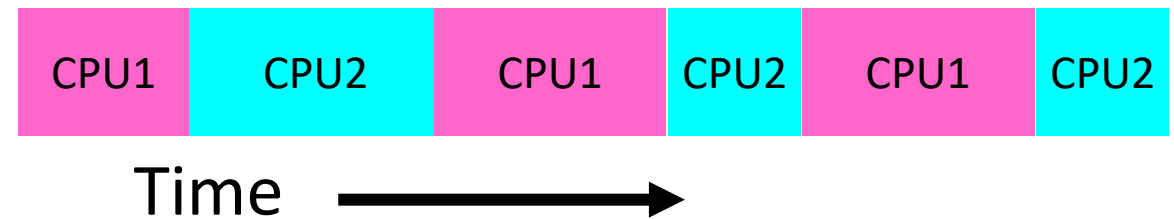
Adding Threads

- Version of program with threads (loose syntax):

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

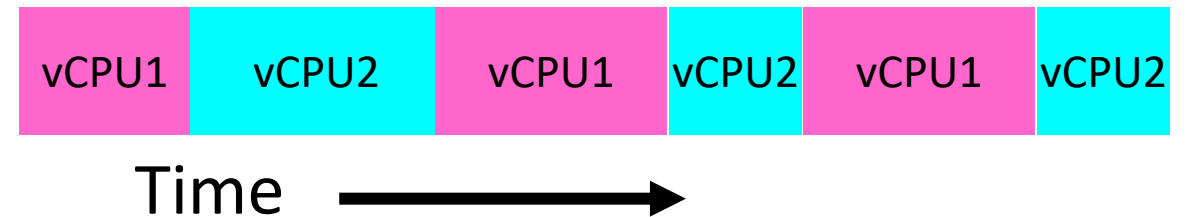
- `create_thread`: Spawns a new thread running the given procedure
 - *Should* behave as if another CPU is running the given procedure

- Now, you would actually see the class list



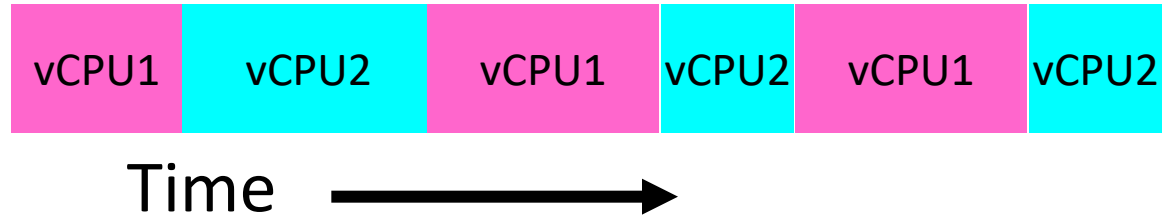
Threads Mask I/O Latency

- 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

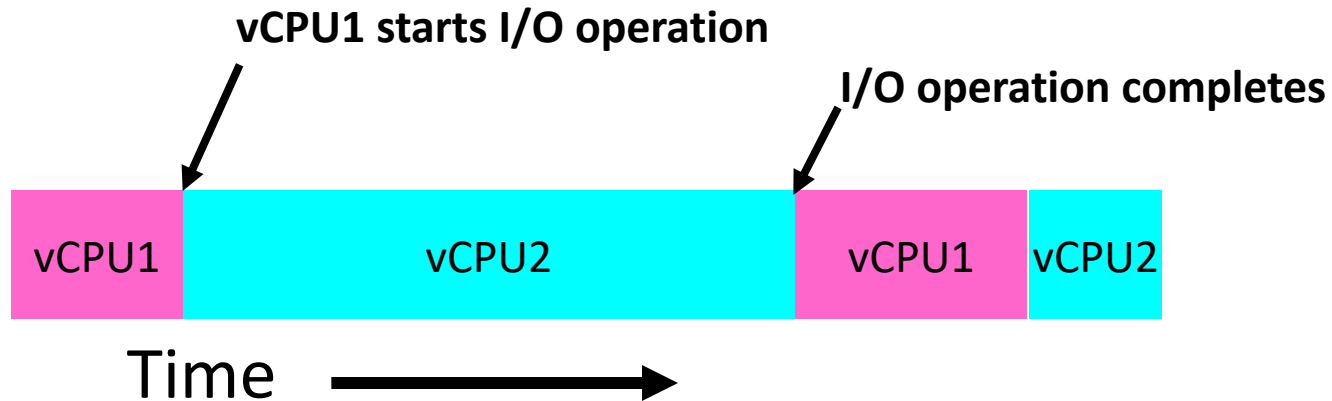


Threads Mask I/O Latency

- If no thread performs I/O:



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



A Better Example for Threads

- Version of program with threads (loose syntax):

```
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

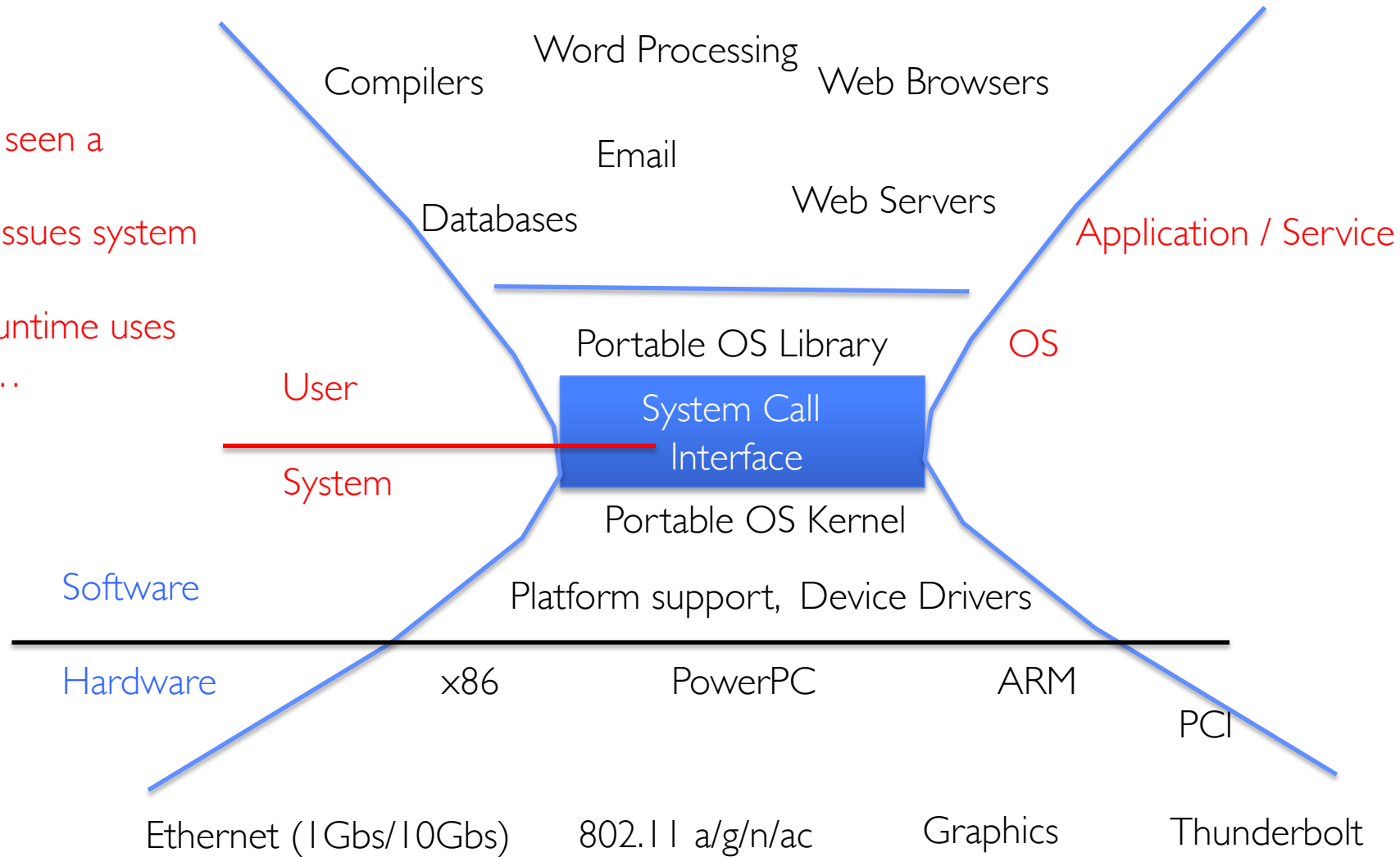
Multithreaded Programs

- 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

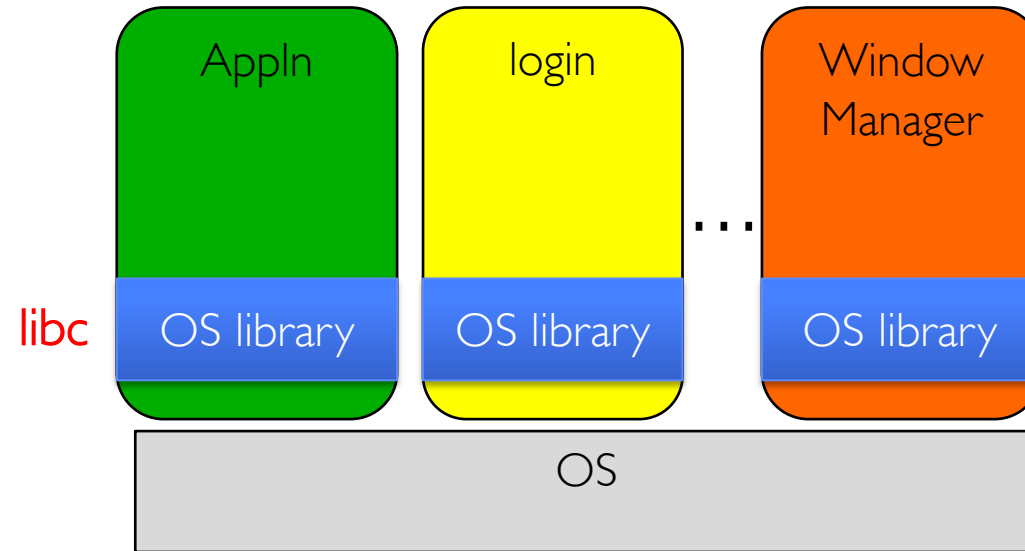
System Calls (“Syscalls”)

“But, I’ve never seen a syscall!”

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



OS Library Issues Syscalls



OS Library API for Threads: *pthread*

```
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`

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

prompt% man pthread

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

Peeking Ahead: System Call Example

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

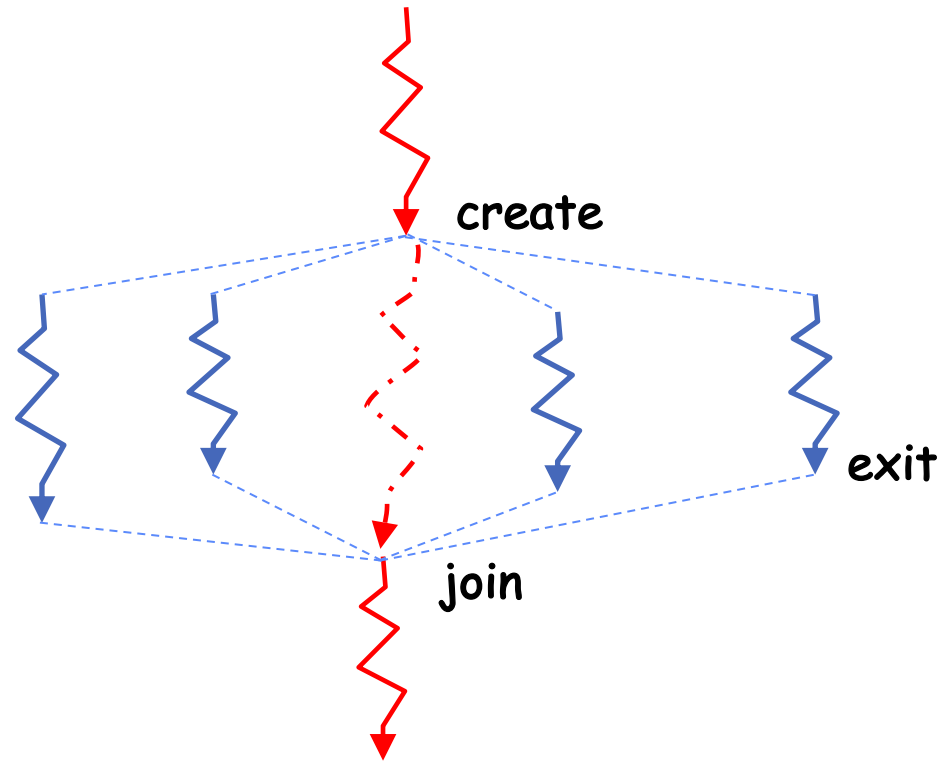
Library:

```
int pthread_create(...) {  
    Do some work like a normal fn..  
  
    asm code ... syscall # into %eax  
    put args into registers %ebx, ...  
    special trap instruction  
  
    get return values from regs  
    Do some more work like a normal fn..  
};
```

Kernel:

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


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.

Group Discussion: pThreads Example

Discuss in groups of two to three students

- How many threads are in this program?
- Does the main thread join with the threads in the same order that they were created?
- Do the threads exit in the same order they were created?
- If we run the program again, would the result change?

```
[(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: 70000d8beef8 common: 10cf95048 (165)
Thread #0 stack: 70000d7b8ef8 common: 10cf95048 (163)
```

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

```
int common = 162;
```

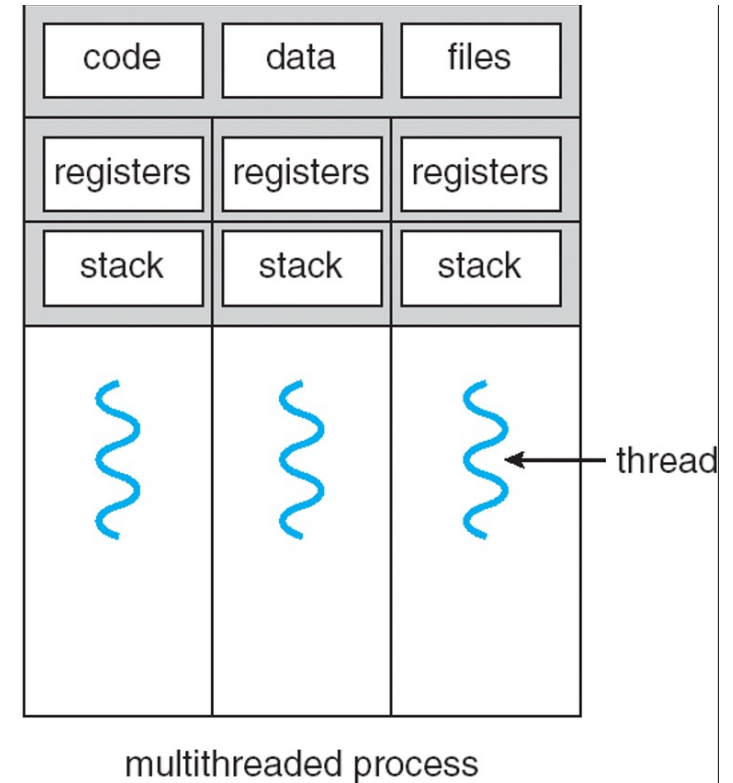
```
void *threadfun(void *threadid)
{
    long tid = (long)threadid;
    printf("Thread #%lx stack: %lx common: %lx (%d)\n", tid,
           (unsigned long) &tid, (unsigned long) &common, common++);
    pthread_exit(NULL);
}
```

```
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 */
```

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 – what is this?
- Execution Stack
 - Parameters, temporary variables
 - Return PCs are kept while called procedures are executing

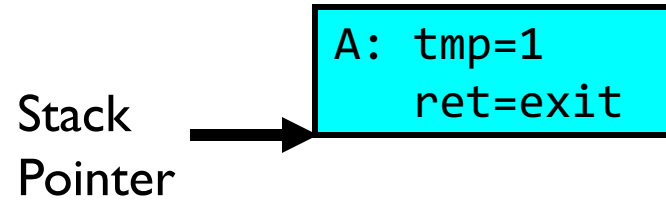
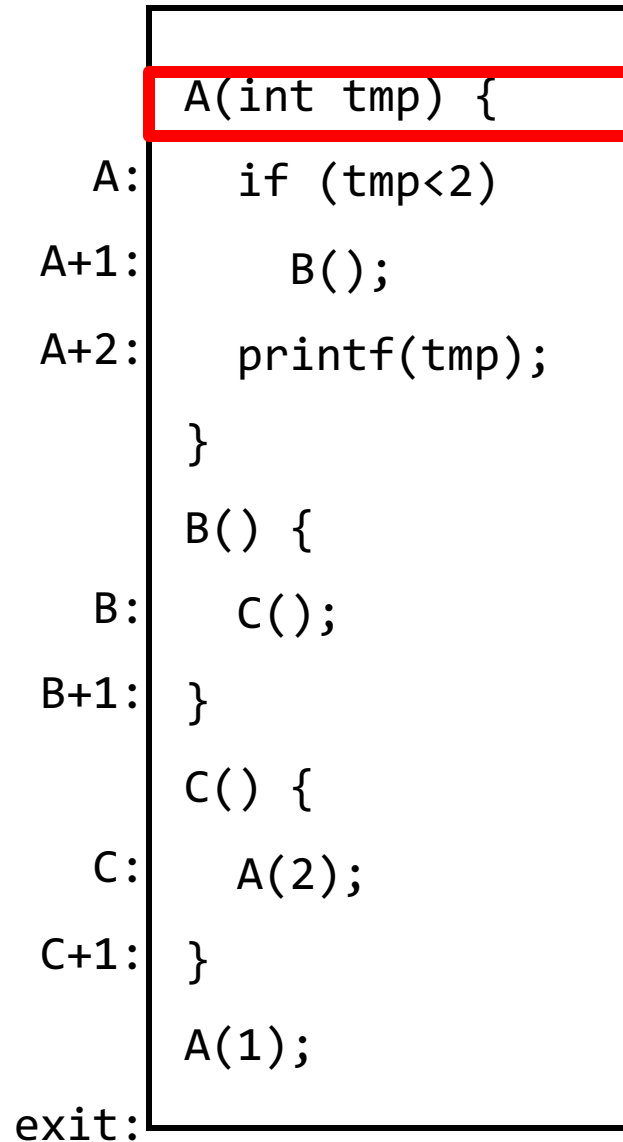


Execution Stack Example

```
    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:
```

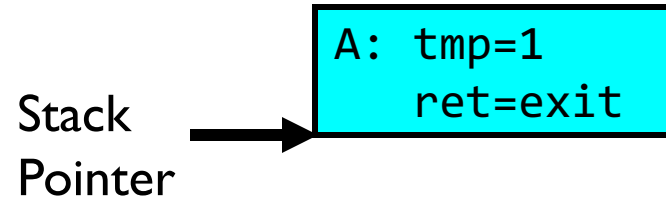
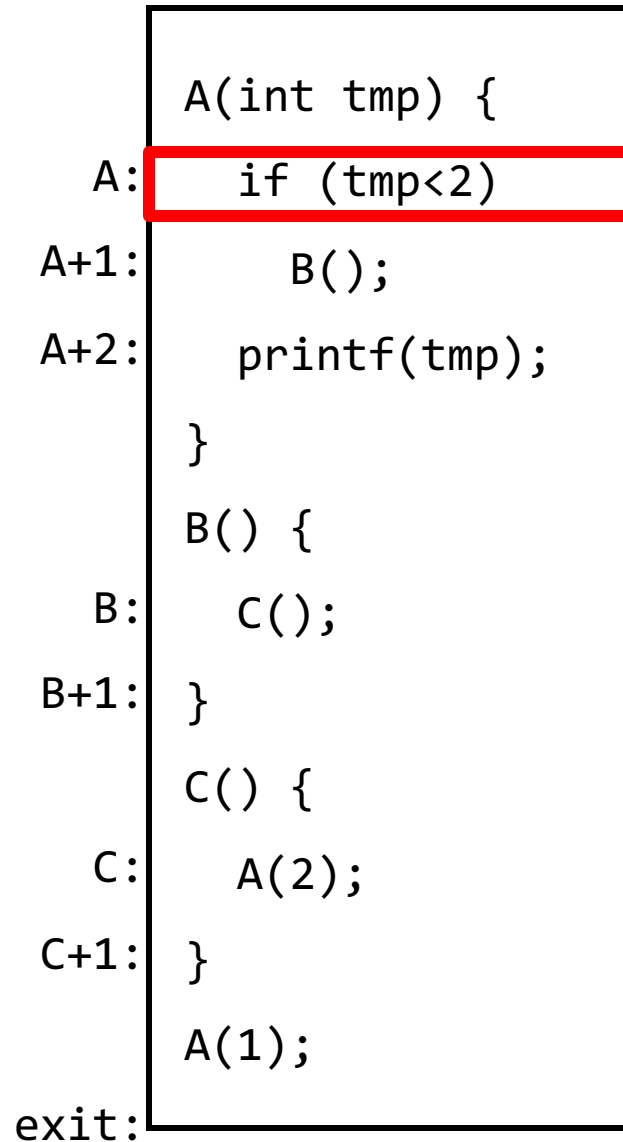
- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example



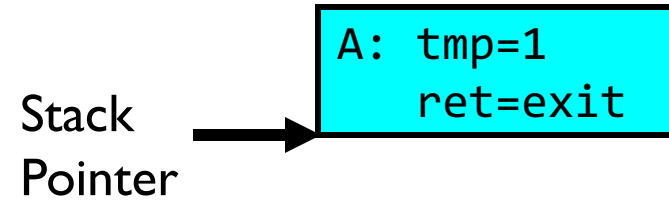
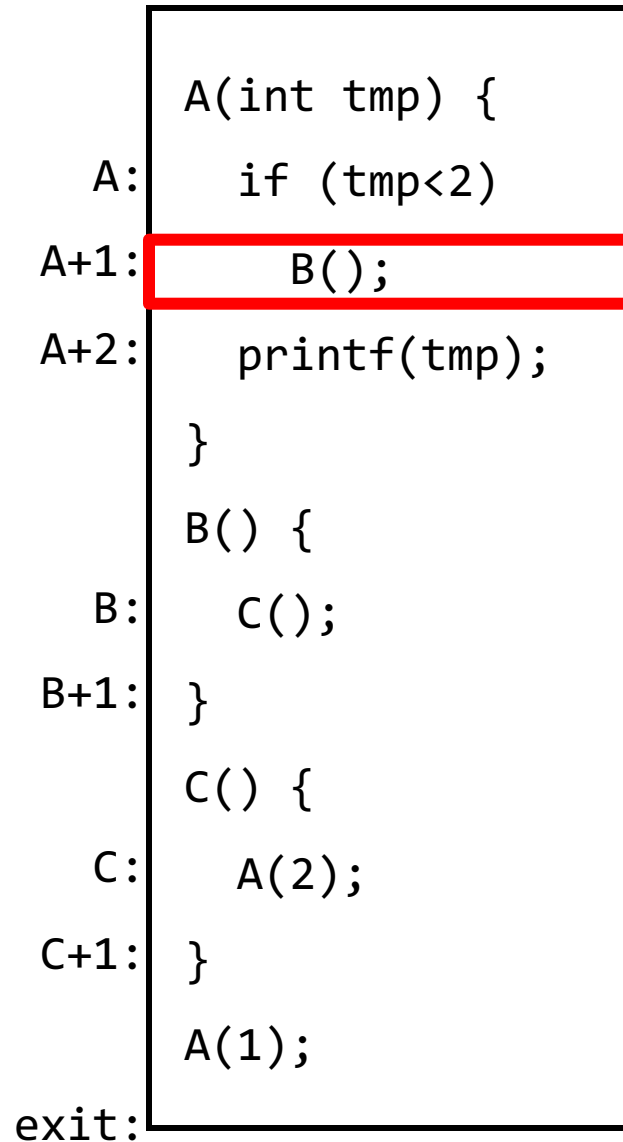
- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example



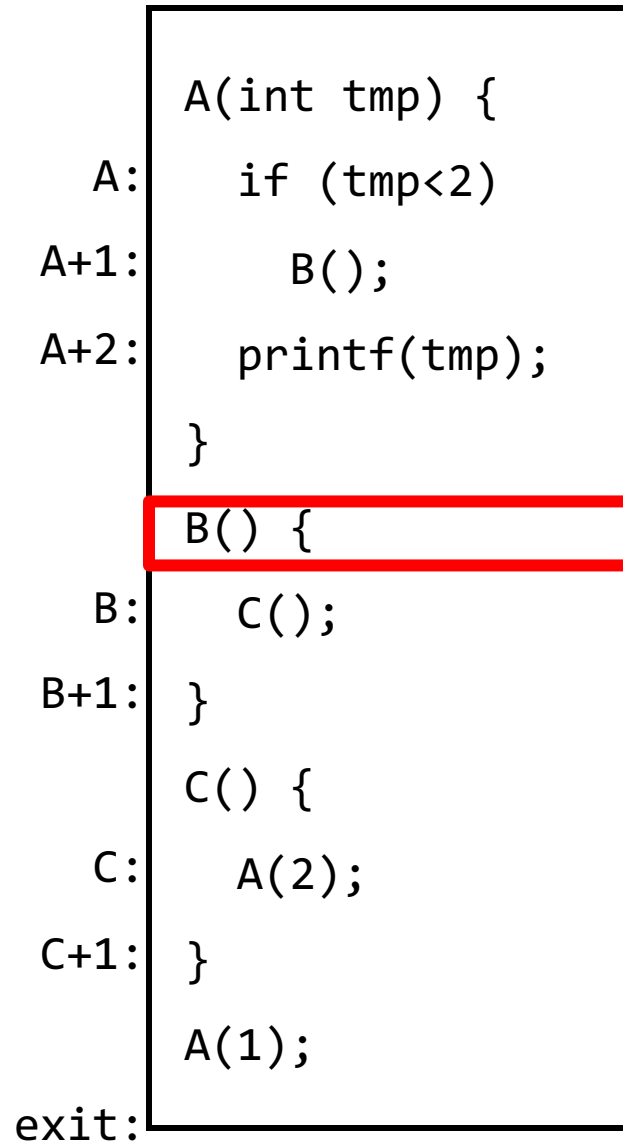
- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

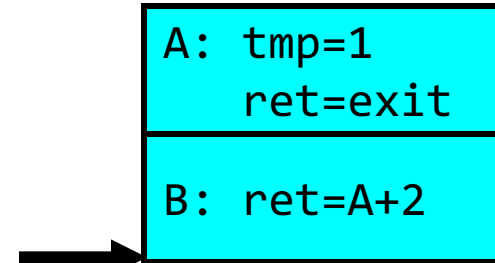


- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example



Stack
Pointer



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

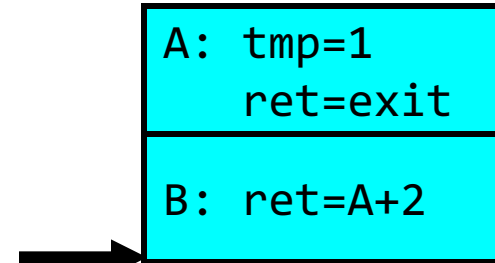
Execution Stack Example

```

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:

```

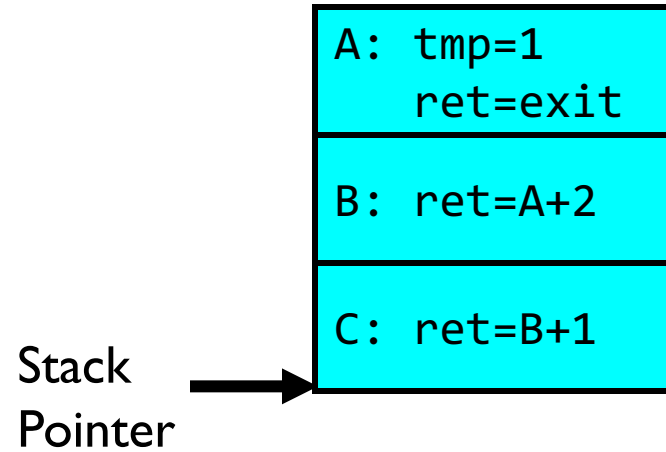
Stack
Pointer



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

Execution Stack Example

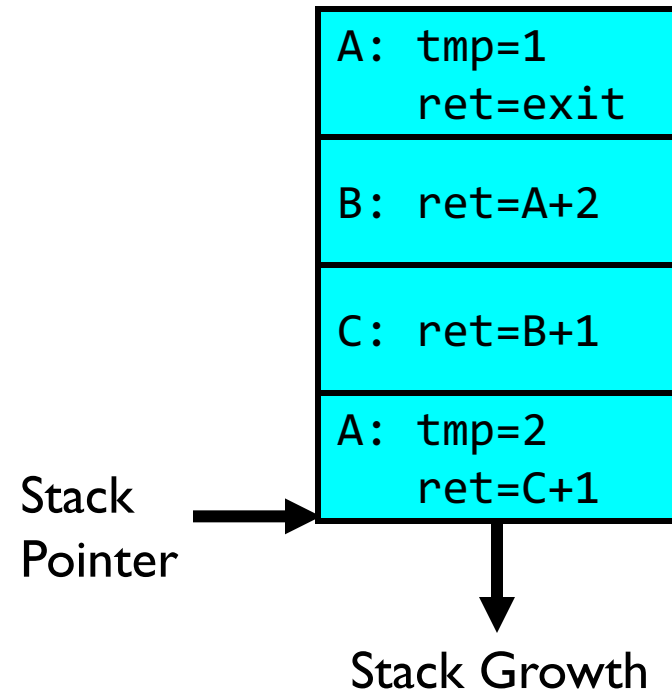
```
A(int tmp) {  
A:   if (tmp<2)  
A+1:   B();  
A+2:   printf(tmp);  
      }  
      B() {  
B:     C();  
B+1:  }  
      C() {  
C:     A(2);  
C+1:  }  
      A(1);  
exit:
```



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

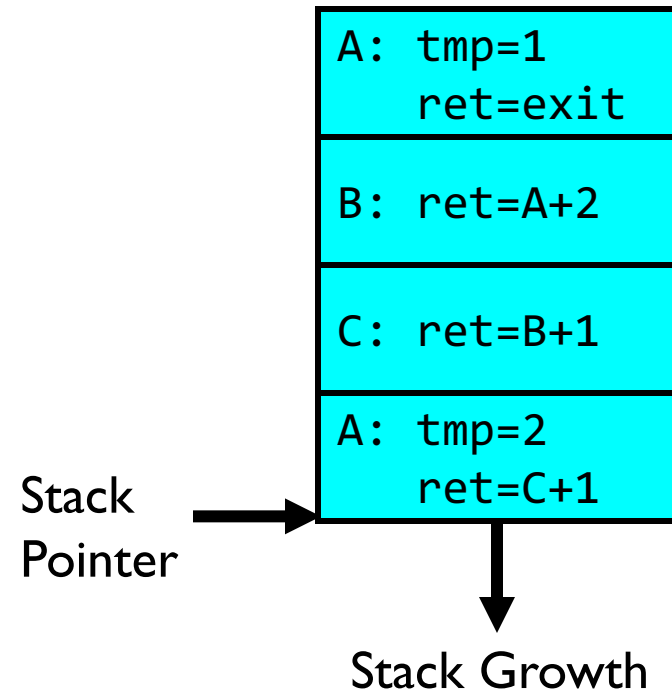
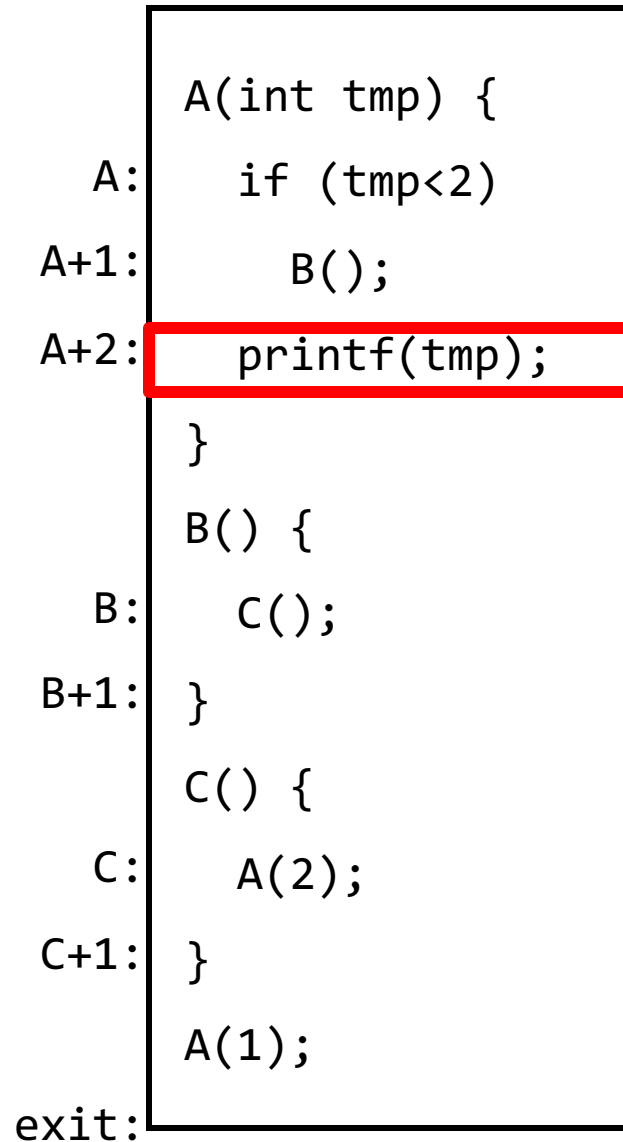
Execution Stack Example

```
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:
```



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

Execution Stack Example



Output: **>2**

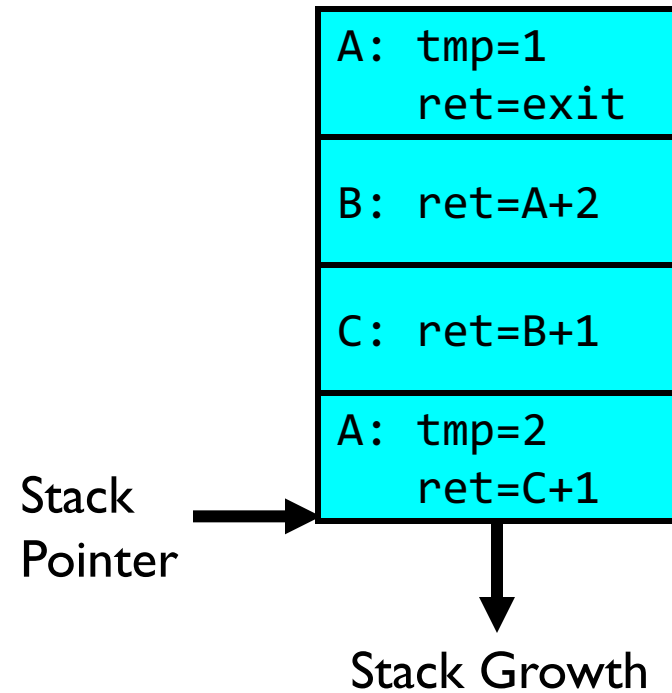
- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

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**

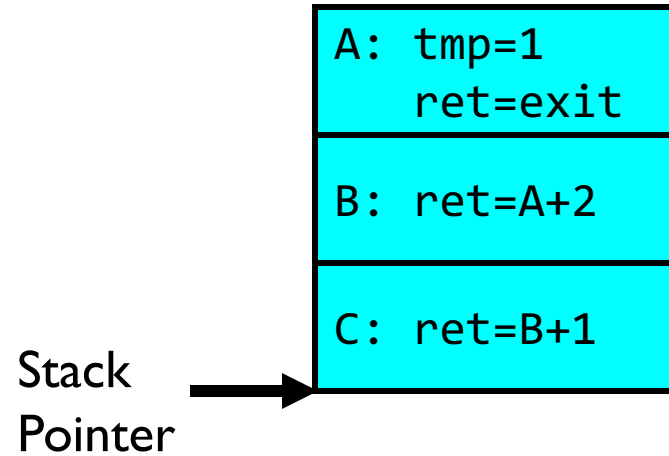
- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

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:

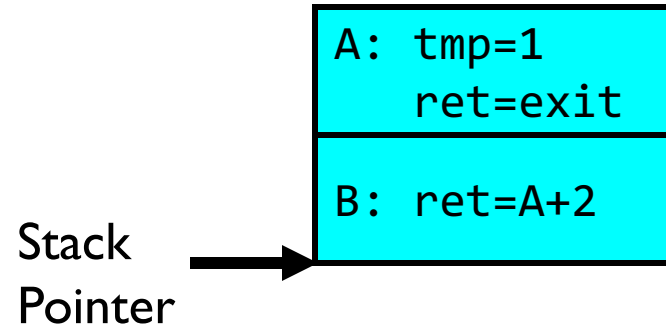
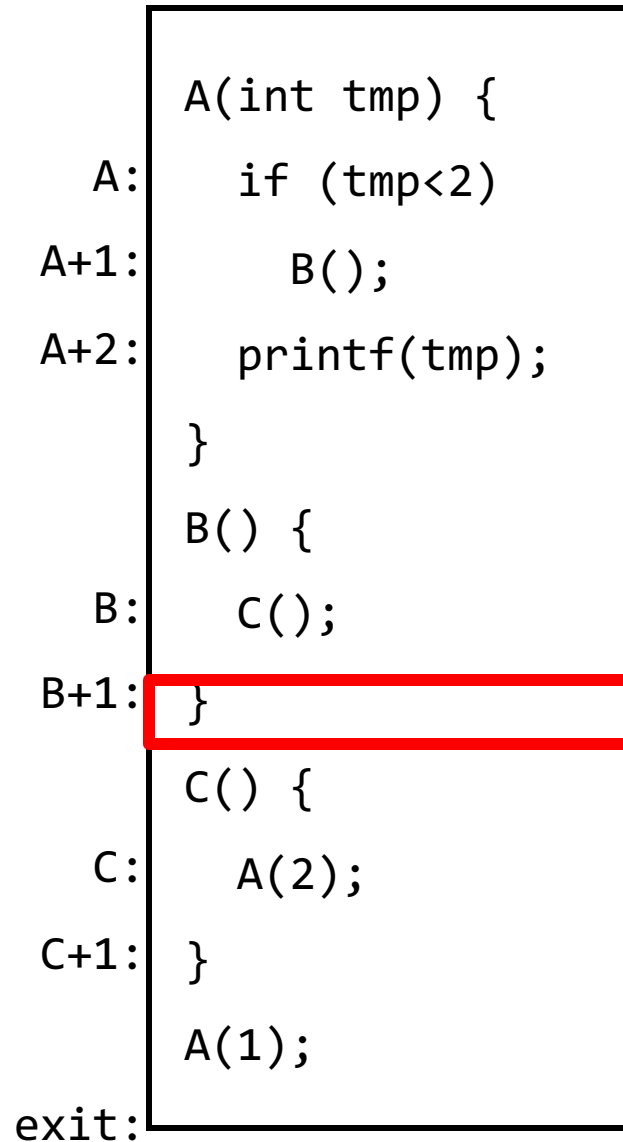
```



Output: **>2**

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

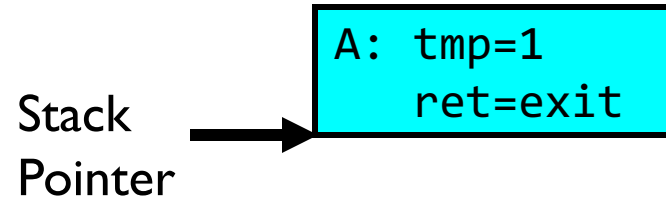
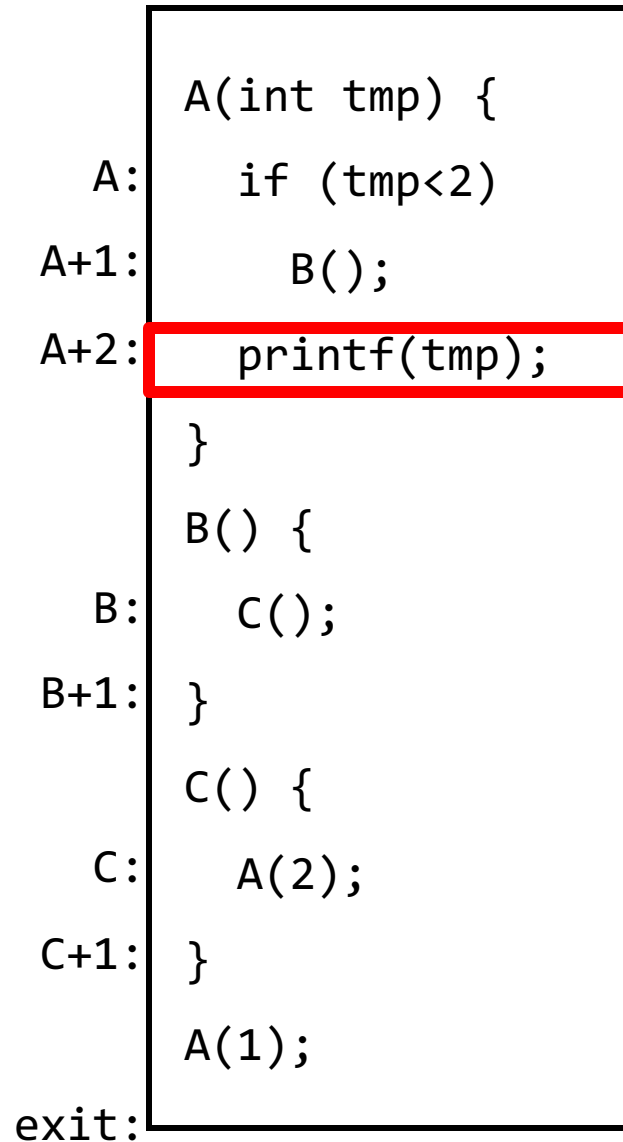
Execution Stack Example



Output: **>2**

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

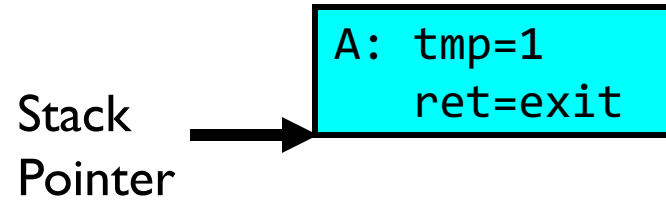
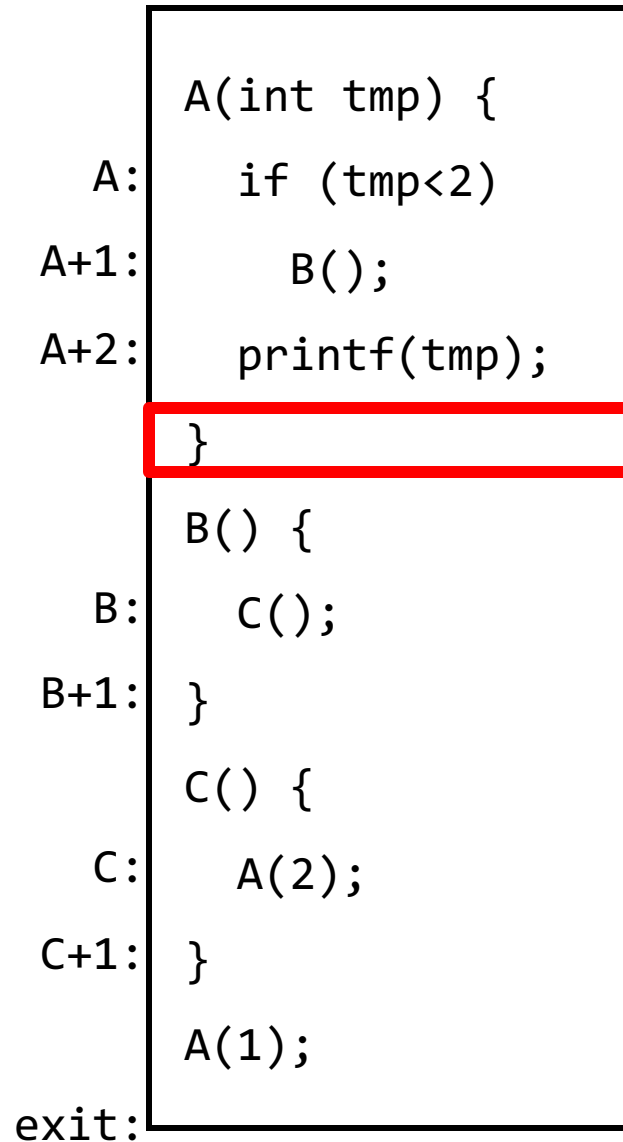
Execution Stack Example



Output: `>2 1`

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

Execution Stack Example



Output: **>2 1**

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

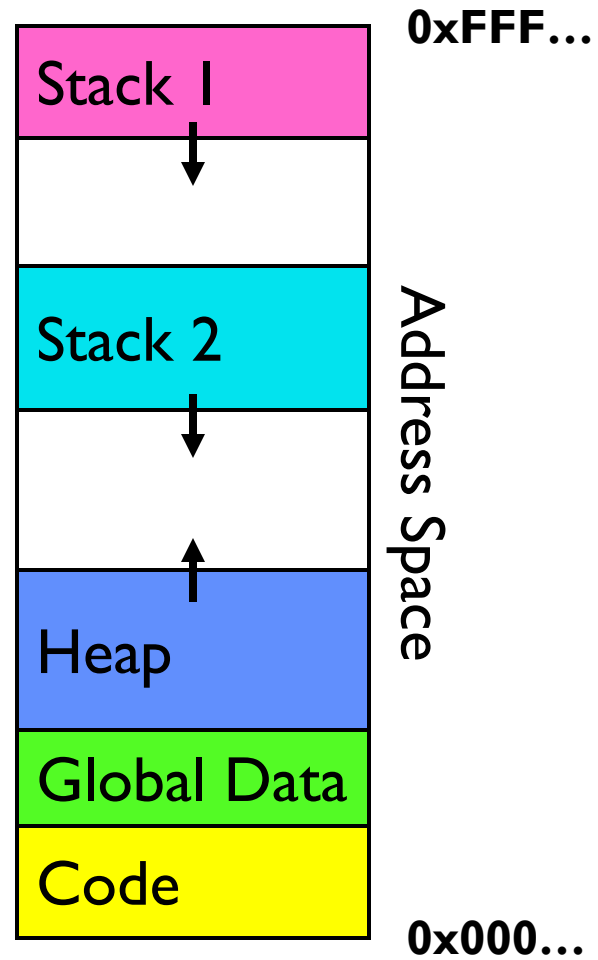
Execution Stack Example

```
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

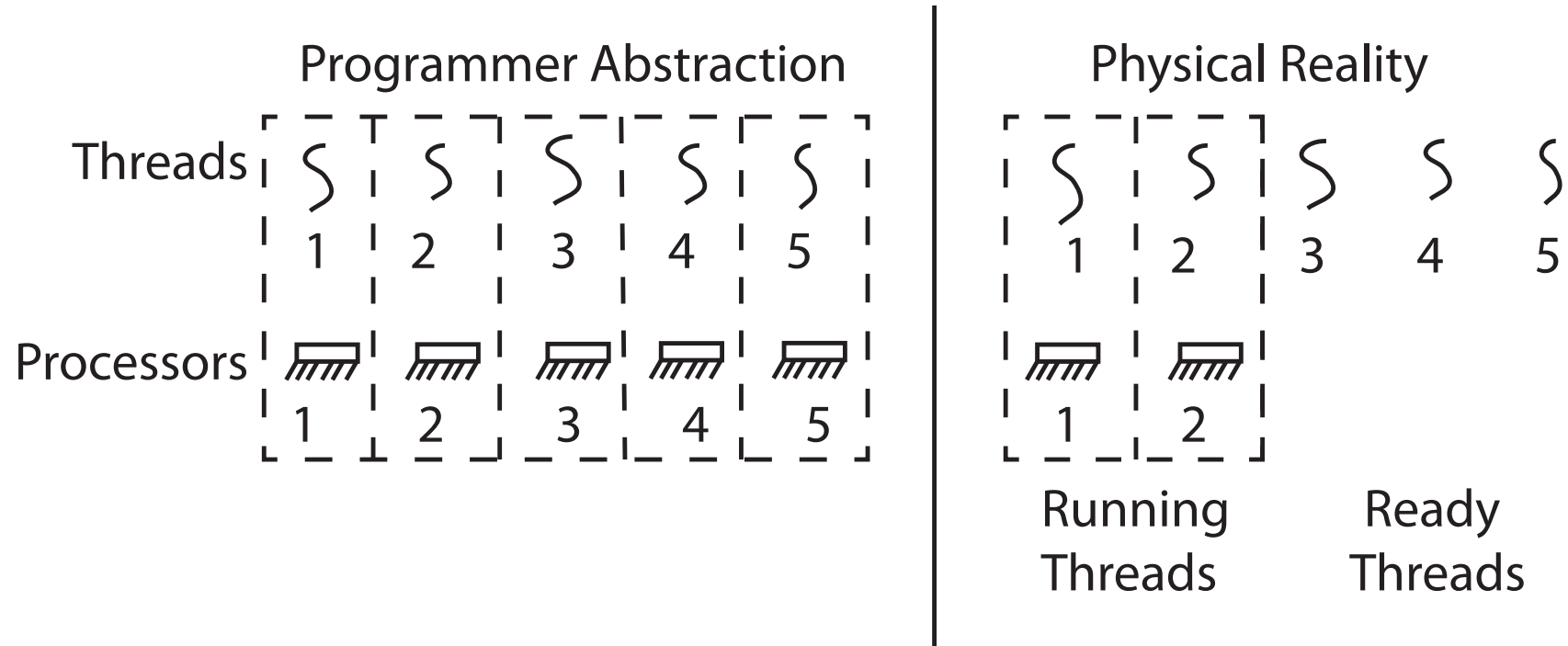
Memory Layout with Two Threads



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;
. .
. .
.

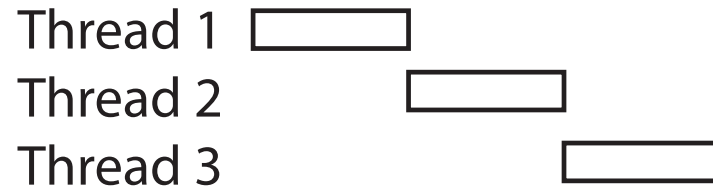
Possible
Execution
#2

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

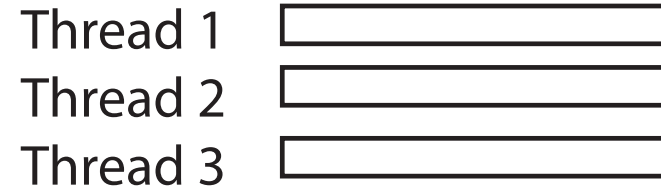
Possible
Execution
#3

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

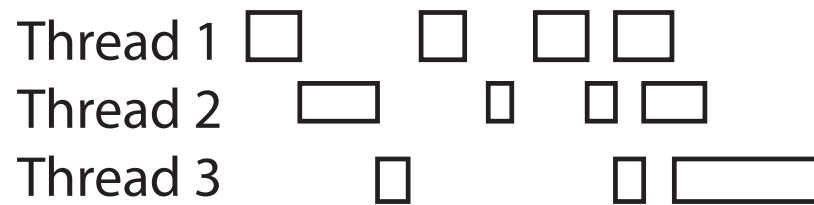
Possible Executions



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

- 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

- 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

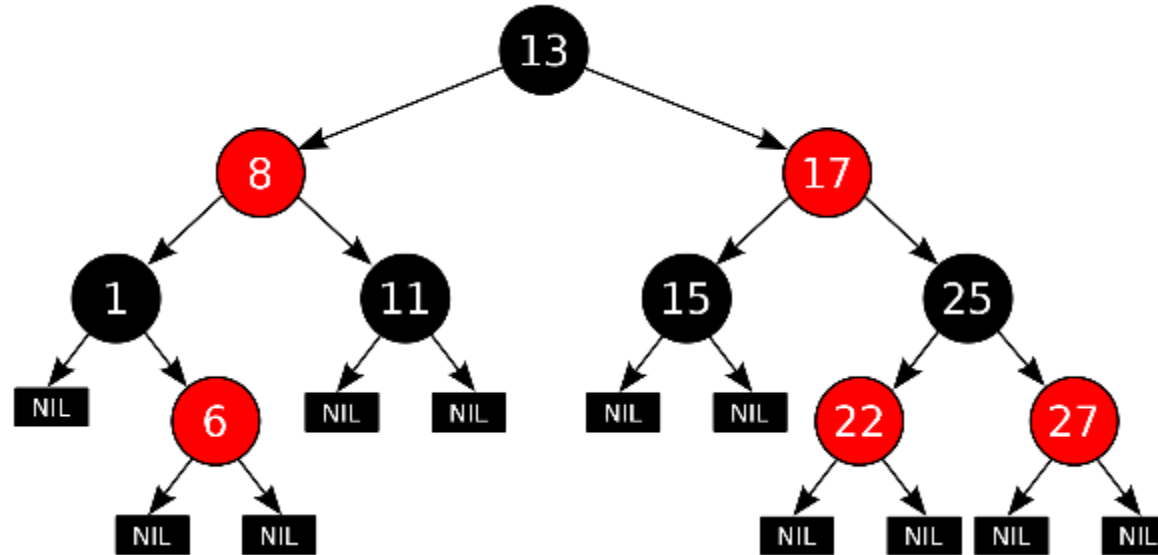
Thread A

Insert(3)

Thread B

Insert(4)

Get(6)



Tree-Based Set Data Structure

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

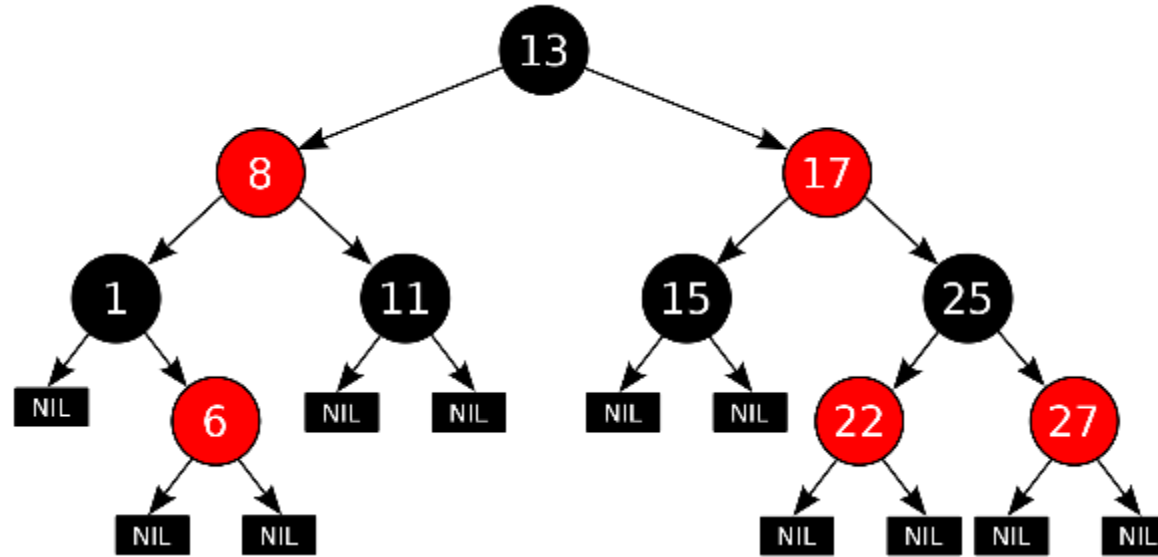
Locks

- 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()



Tree-Based Set Data Structure

Thread B

Insert(4)

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

Get(6)

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

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);
```

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);
}
```


Semaphores: A quick look

- Semaphores are a kind of *generalized lock*
 - First defined by Dijkstra in late 60s
 - Main synchronization primitive used in original UNIX (& Pintos)
- Definition: a Semaphore has a non-negative integer value and supports the following two operations:
 - **P()** or **down()**: atomic operation that waits for semaphore to become positive, then decrements it by 1
 - **V()** or **up()**: an atomic operation that increments the semaphore by 1, waking up a waiting P, if any

P() stands for “*proberen*” (to test) and **V()** stands for “*verhogen*” (to increment) in Dutch

Two Semaphore Patterns

- **Mutual Exclusion:** (like lock)

- Called a "binary semaphore" or "mutex"

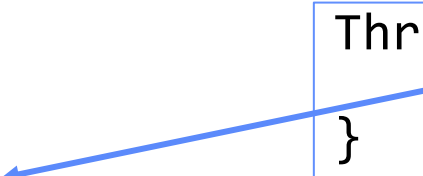
```
initial value of semaphore = 1;
semaphore.down();
    // Critical section goes here
semaphore.up();
```

- **Signaling** other threads, e.g. **ThreadJoin**

Initial value of semaphore = 0

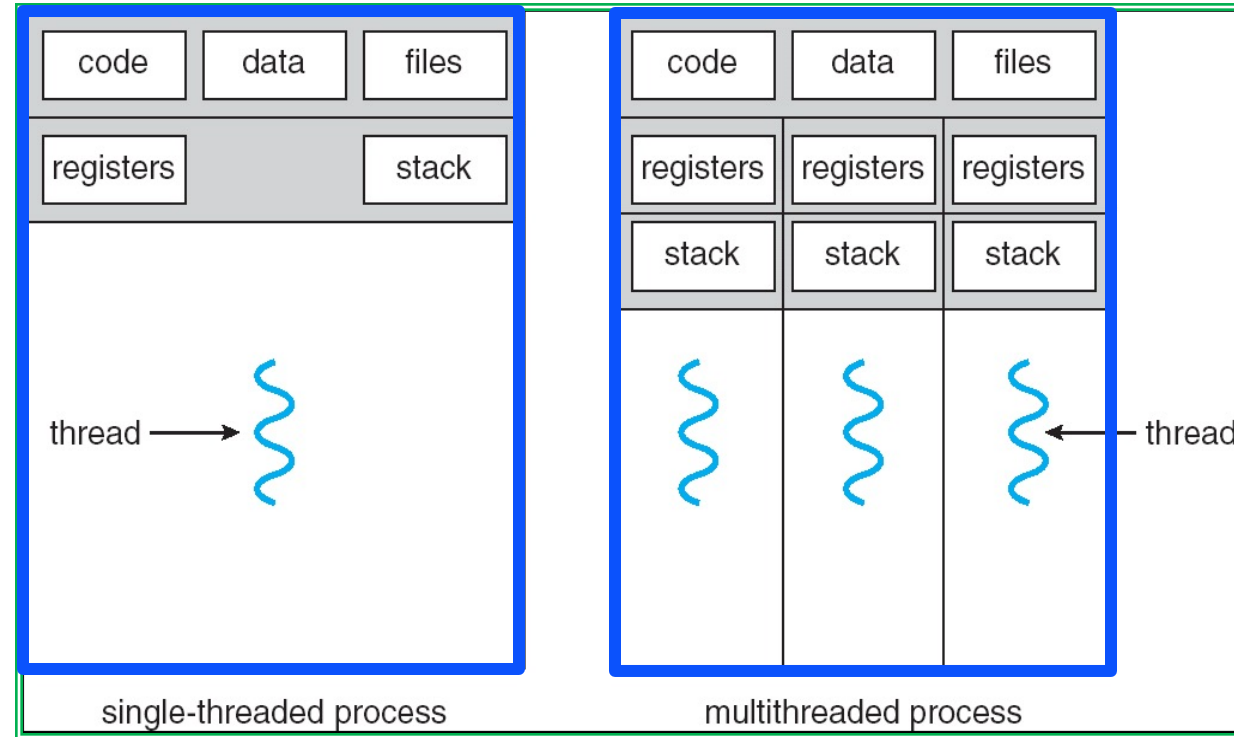
```
ThreadJoin {
    semaphore.down();
}
```

```
ThreadFinish {
    semaphore.up();
}
```



Processes

- Definition: execution environment with restricted rights
 - One or more threads executing in a single address space
 - Owns file descriptors, network connections
- Instance of a running program
 - When you run an executable, it runs in its own process
 - Application: one or more processes working together



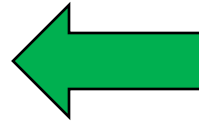
- Protected from each other; OS protected from them
- **In modern OSes, anything that runs outside of the kernel runs in a process**

Creating Processes

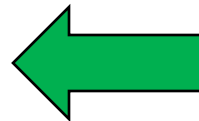
- `pid_t fork()` – copy the current process
 - New process has different pid
 - New process contains a single thread
- Return value from `fork()`: pid (like an integer)
 - When > 0 :
 - » Running in (original) **Parent** process
 - » return value is **pid** of new child
 - When $= 0$:
 - » Running in new **Child** process
 - When < 0 :
 - » Error! Must handle somehow
 - » Running in original process
- State of original process duplicated in *both* Parent and Child!
 - Address Space (Memory), File Descriptors (covered later), etc...

fork_race.c

```
int i;
pid_t cpid = fork();
if (cpid > 0) {
    for (i = 0; i < 10; i++) {
        printf("Parent: %d\n", i);
        // sleep(1);
    }
} else if (cpid == 0) {
    for (i = 0; i > -10; i--) {
        printf("Child: %d\n", i);
        // sleep(1);
    }
} else { /* ERROR! */ }
```



Parent Process
Runs HERE!



Child Process
Runs HERE!

- Group discussion
 - What does this print?
 - Would adding the calls to `sleep()` matter?

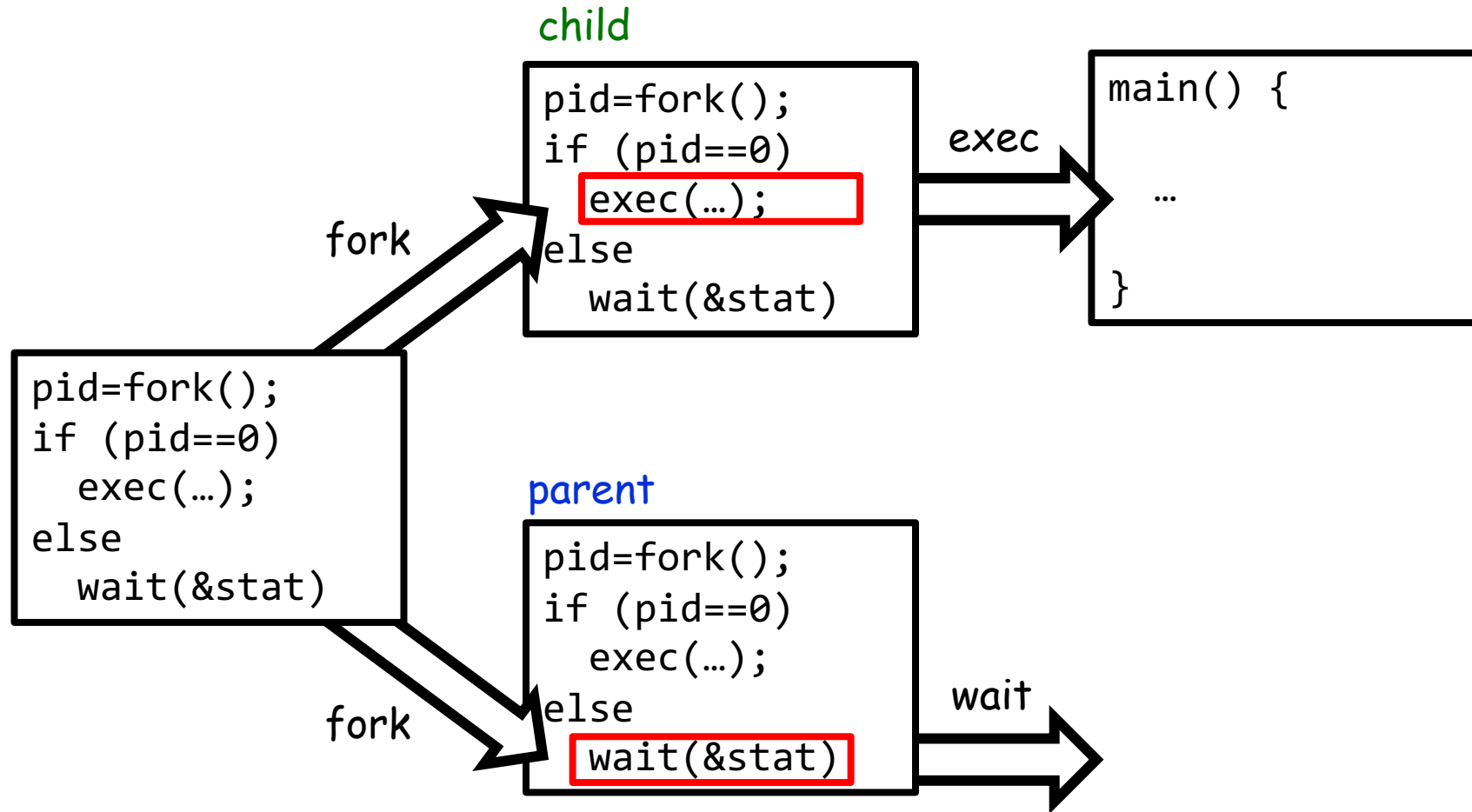
Start new Program with exec

```
...
cpid = fork();
if (cpid > 0) {                               /* Parent Process */
    tcpid = wait(&status);
} else if (cpid == 0) {                       /* Child Process */
    char *args[] = {"ls", "-l", NULL};
    execv("/bin/ls", args);

    /* execv doesn't return when it works.
       So, if we got here, it failed! */

    perror("execv");
    exit(1);
}
...
```

Starting New Program (for instance in Shell)



Finishing up: Process Management API

- `exit` – terminate a process
- `fork` – copy the current process
- `exec` – change the *program* being run by the current process
- `wait` – wait for a process to finish
- `kill` – send a *signal* (interrupt-like notification) to another process
- `sigaction` – set handlers for signals

fork2.c – parent waits for child to finish

```
int status;
pid_t tcpid;
...
cpid = fork();
if (cpid > 0) {                               /* Parent Process */
    mypid = getpid();
    printf("[%d] parent of [%d]\n", mypid, cpid);
    tcpid = wait(&status);
    printf("[%d] bye %d(%d)\n", mypid, tcpid, status);
} else if (cpid == 0) {                       /* Child Process */
    mypid = getpid();
    printf("[%d] child\n", mypid);
    exit(42);
}
...
```

Finishing up: Process Management API

- `exit` – terminate a process
- `fork` – copy the current process
- `exec` – change the *program* being run by the current process
- `wait` – wait for a process to finish
- `kill` – send a *signal* (interrupt-like notification) to another process
- `sigaction` – set handlers for signals

inf_loop.c

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>

void signal_callback_handler(int signum) {
    printf("Caught signal!\n");
    exit(1);
}

int main() {
    struct sigaction sa;
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);
    sa.sa_handler = signal_callback_handler;
    sigaction(SIGINT, &sa, NULL);
    while (1) {}
}
```

Process vs. Thread APIs

- Why have `fork()` and `exec()` system calls for processes, but just a `pthread_create()` function for threads?
 - Convenient to fork without `exec`: put code for parent and child in one executable instead of multiple
 - It will allow us to programmatically control child process' state
 - » By executing code before calling `exec()` in the child
 - We'll see this in the case of File I/O later
- Windows uses `CreateProcess()` instead of `fork()`
 - Also works, but a more complicated interface

Group Discussion

- Topic: Threads vs. Processes
 - If we have two tasks to run concurrently, do we run them in separate threads, or do we run them in separate processes?
 - What are the pros and cons?
- Discuss in groups of two to three students
 - Each group chooses a leader to summarize the discussion
 - In your group discussion, please do not dominate the discussion, and give everyone a chance to speak

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