香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

Operating Systems
Lecture 3

Threads and processes:
a quick, programmer's viewpoint

Prof. Yunming XIAO

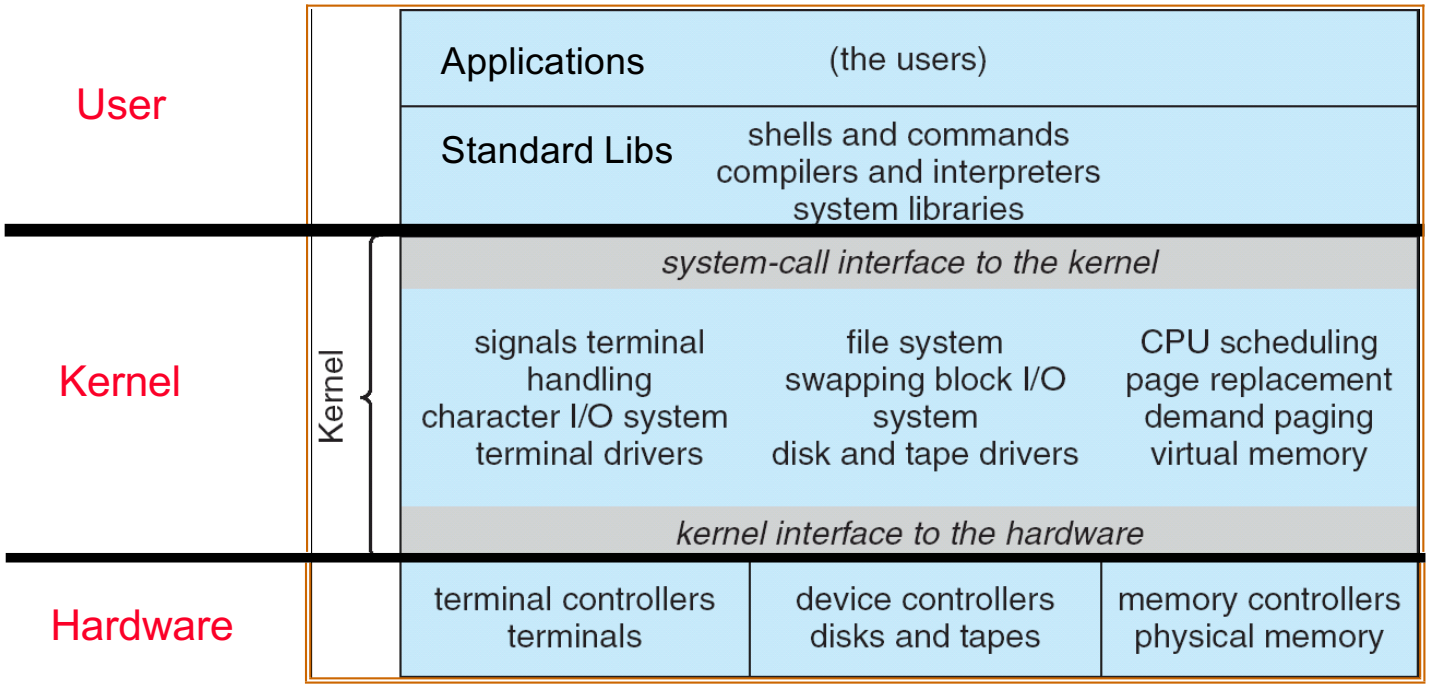
School of Data Science

Acknowledgments: Ion Stoica and Matei Zaharia, Berkeley CS 162

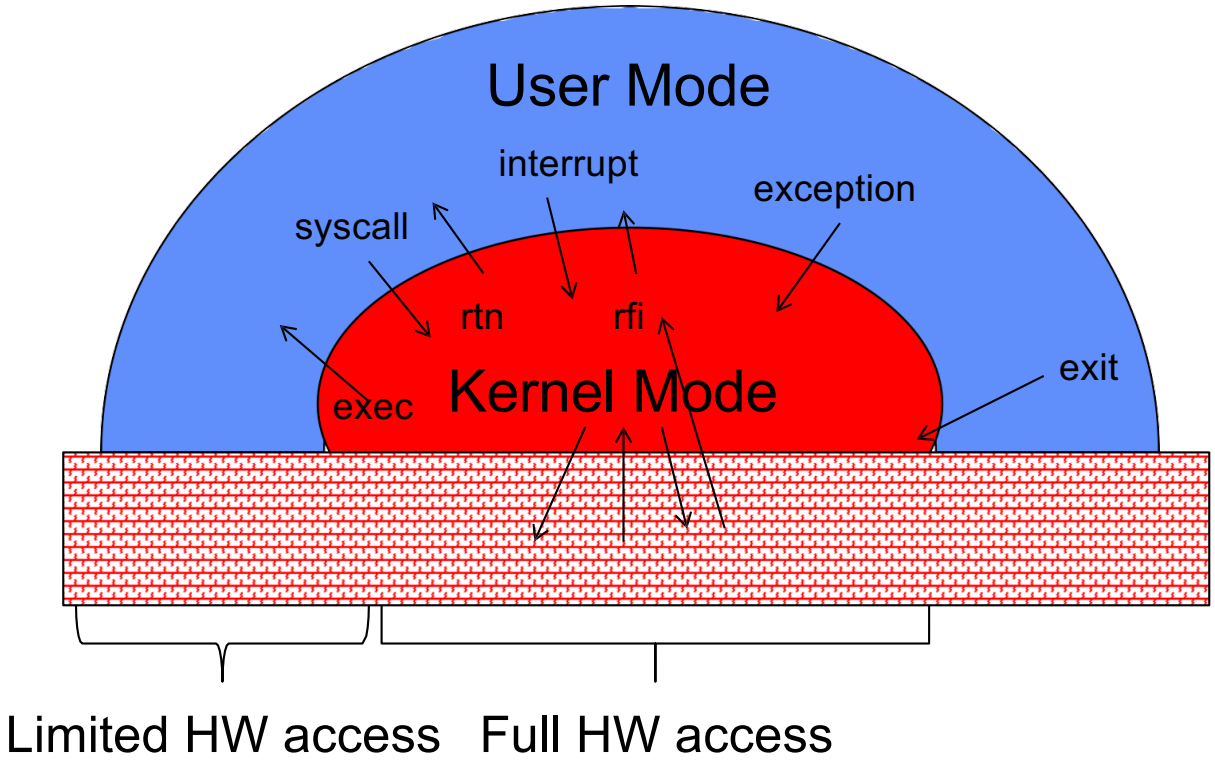
Conclusion: four fundamental OS concepts

- **Process: an instance of a running program**
  - Address Space + one or more Threads
- **Thread: execution context**
  - Fully describes program state
  - Program Counter, Registers, Stack, Execution Flags,
- **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)
- **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
  - Key to virtualizing hardware to multiple processes (apps)

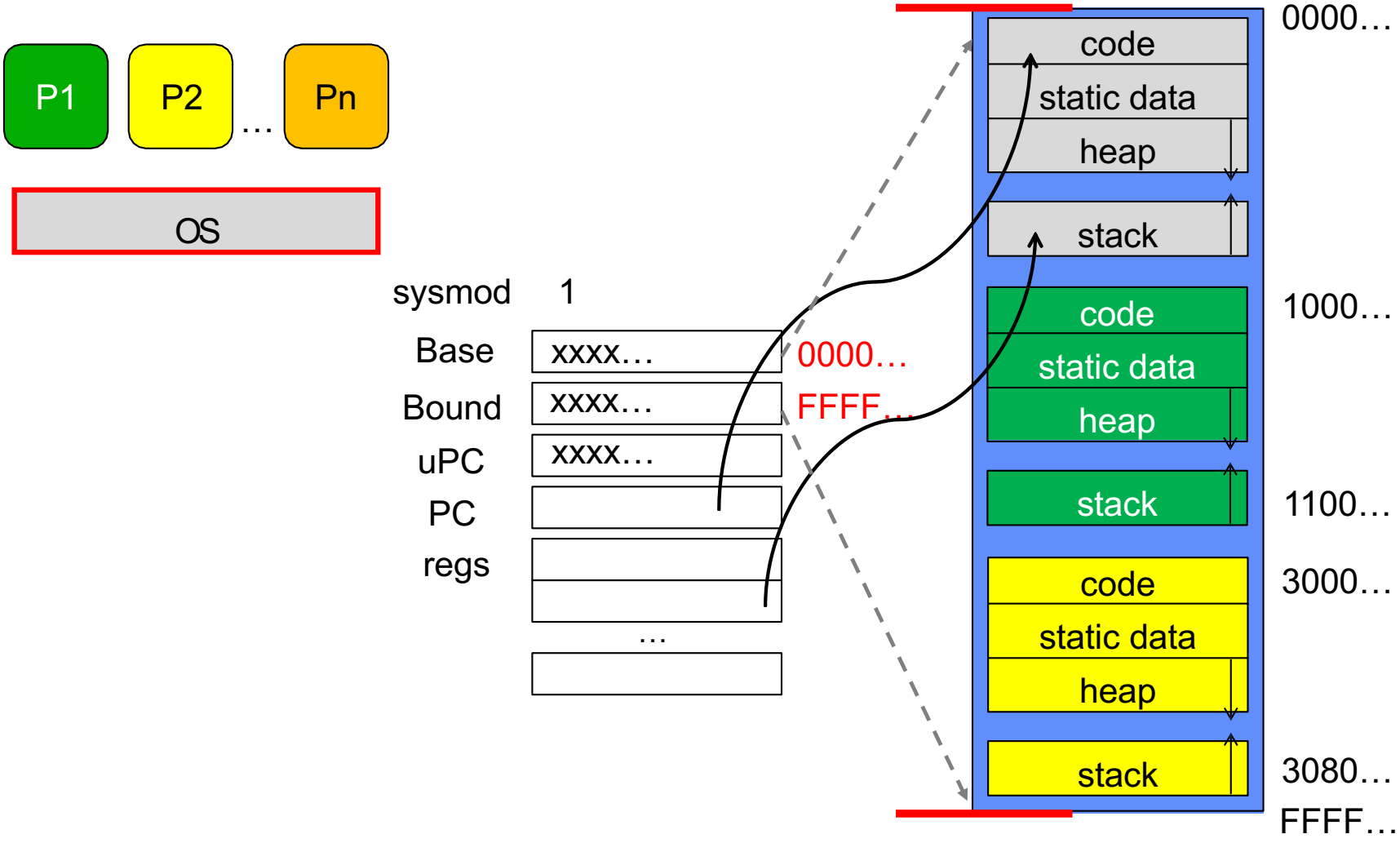
Example: UNIX system structure



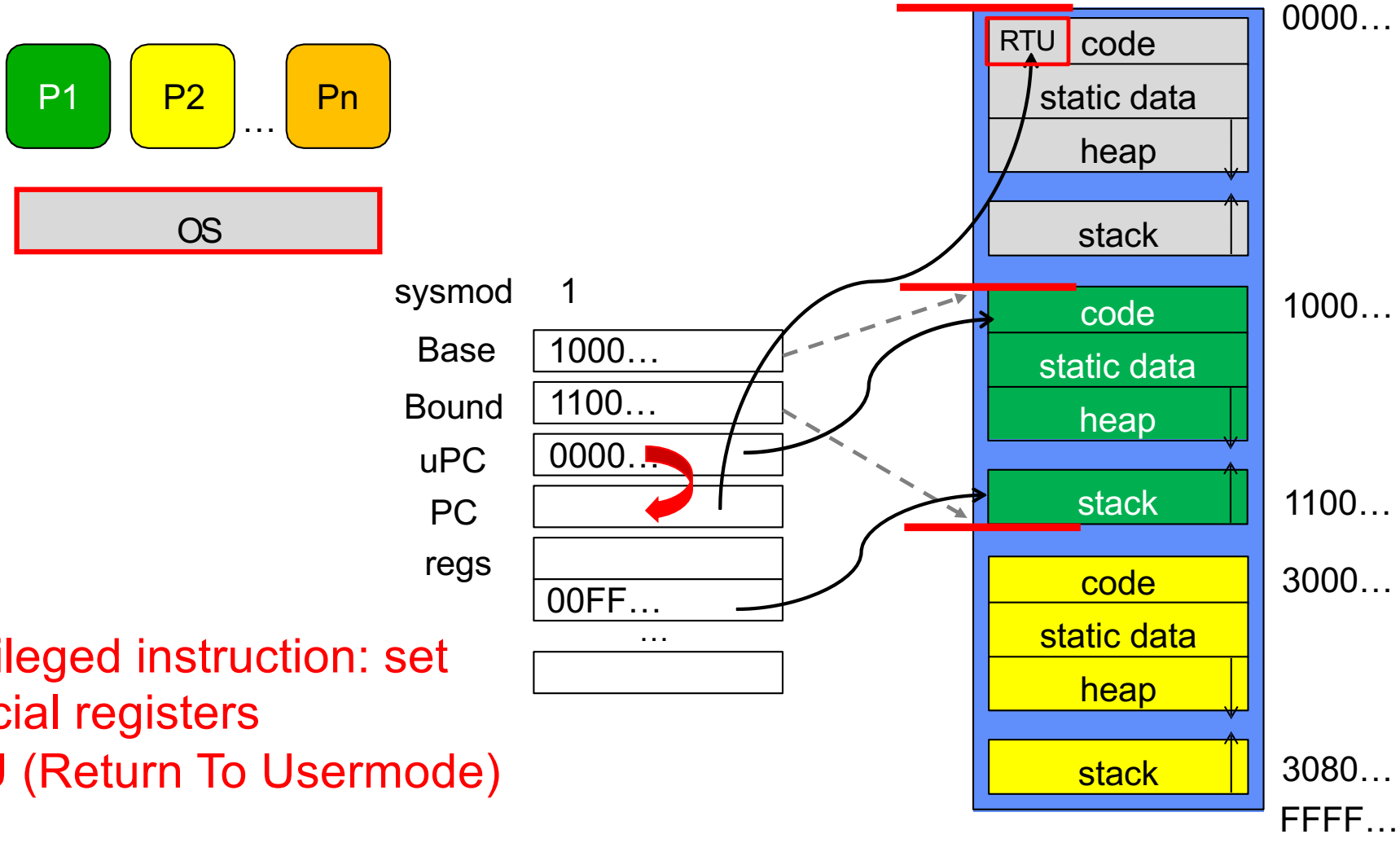
User/Kernel (privileged) mode



Trying it together: simple B&B: OS load process

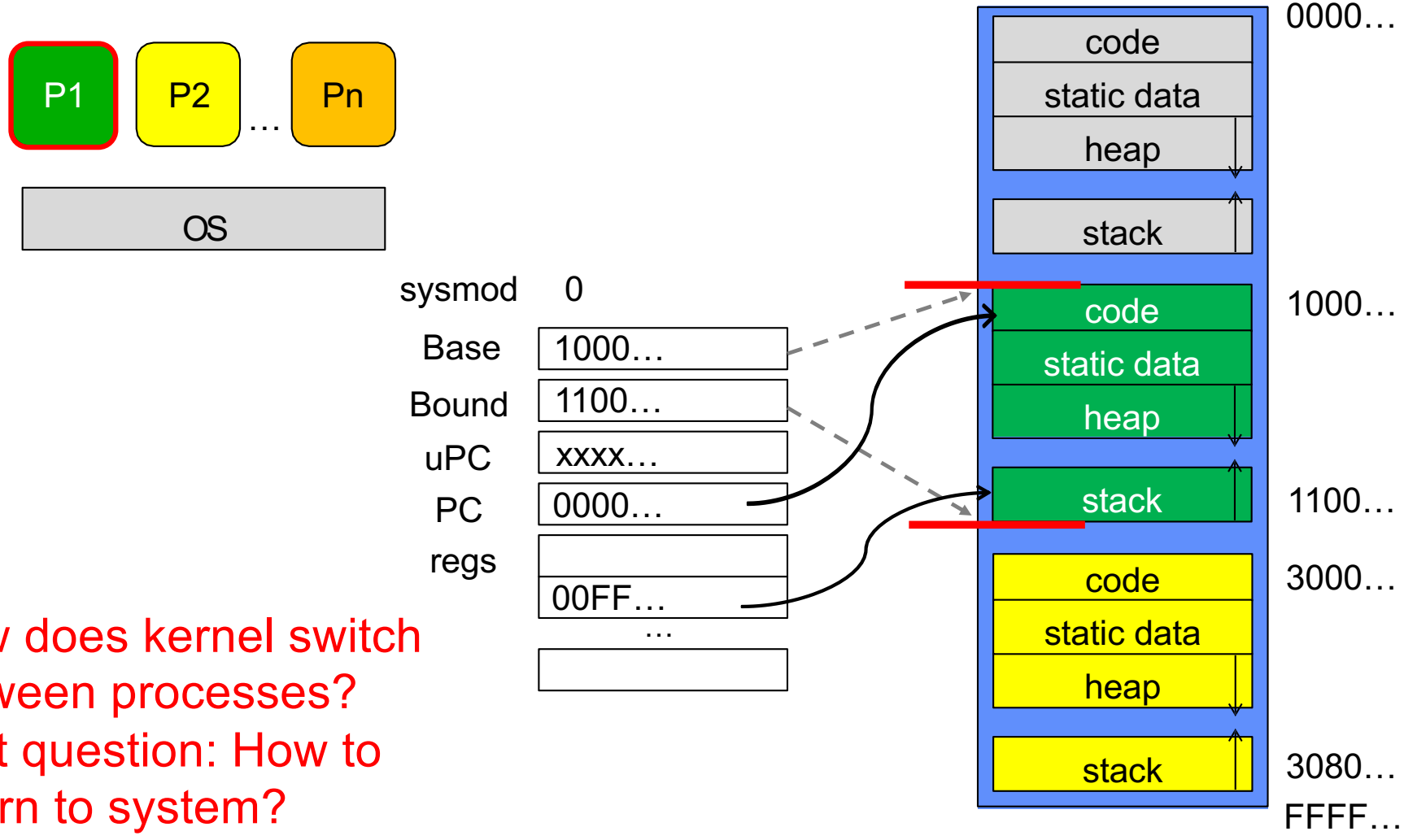


Simple B&B: OS gets ready to execute process



- Privileged instruction: set special registers
- RTU (Return To Usermode)

Simple B&B: user code running



- How does kernel switch between processes?
- First question: How to return to system?

3 types of user to kernel mode transfer

- Syscall
  - Process requests a system service, e.g., exit
  - Like a function call, but “outside” the process
  - Does not have the address of the system function to call
  - Like a Remote Procedure Call (RPC) – for later
  - Marshall the syscall id and args in registers and exec syscall
- Interrupt
  - External asynchronous event triggers context switch
  - e. g., Timer, I/O device
  - Independent of user process
- Trap or Exception
  - Internal synchronous event in process triggers context switch
  - e.g., Protection violation (segmentation fault), Divide by zero, …

3 types of user to kernel mode transfer

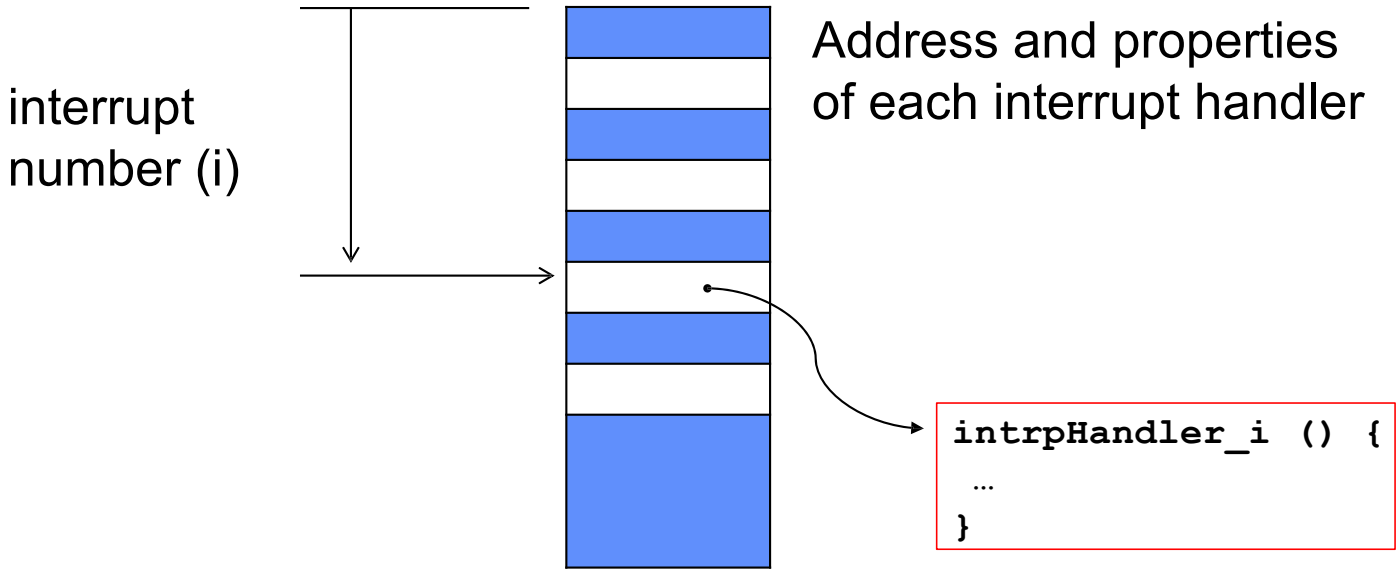
- Syscall
- Interrupt
- Trap or Exception

- All 3 are an UNPROGRAMMED* CONTROL TRANSFER
  - Where does it go?

*Switching to OS without a direct instruction from the running process

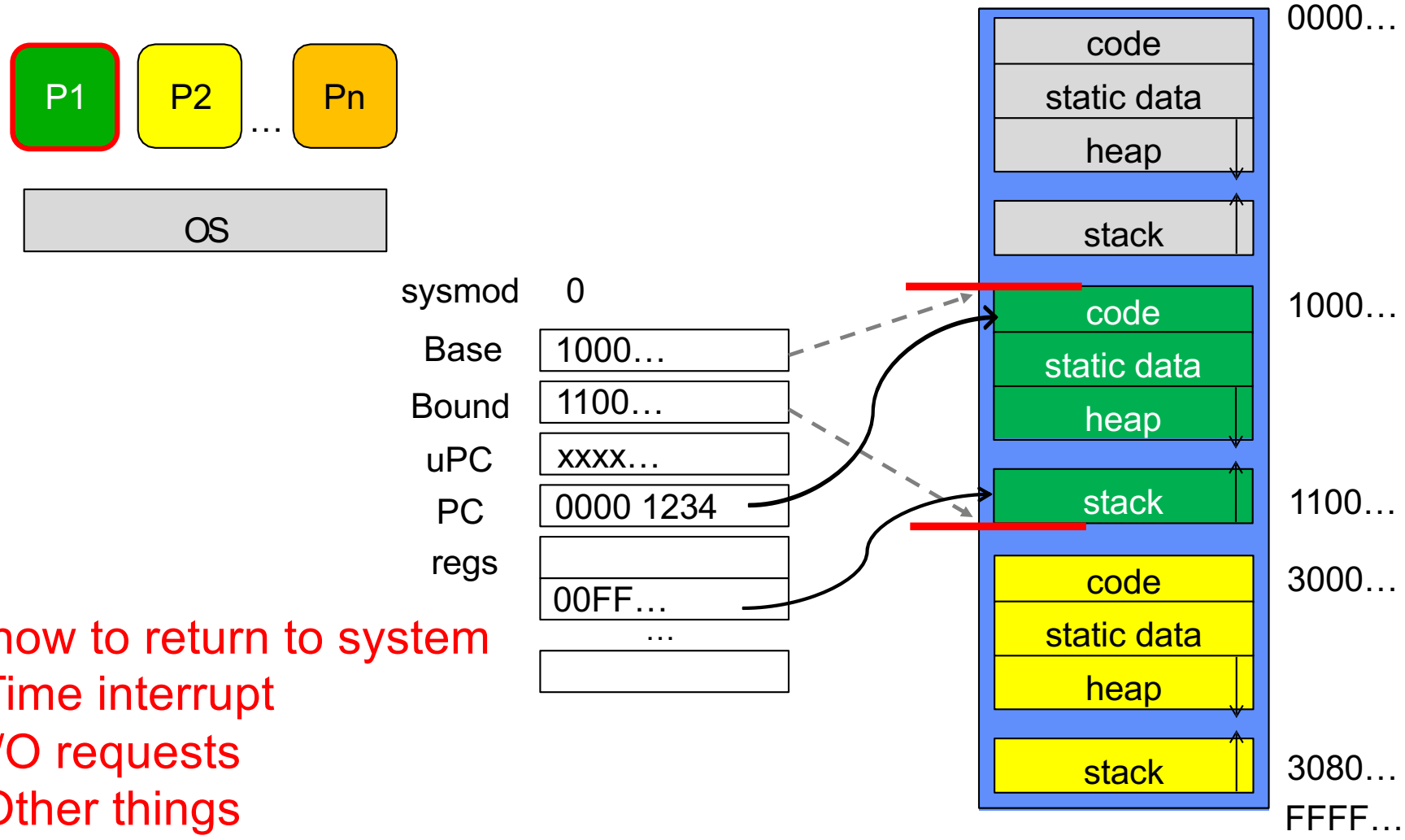
How do we get the system target address of the "unprogrammed control transfer"?

Interrupt vector



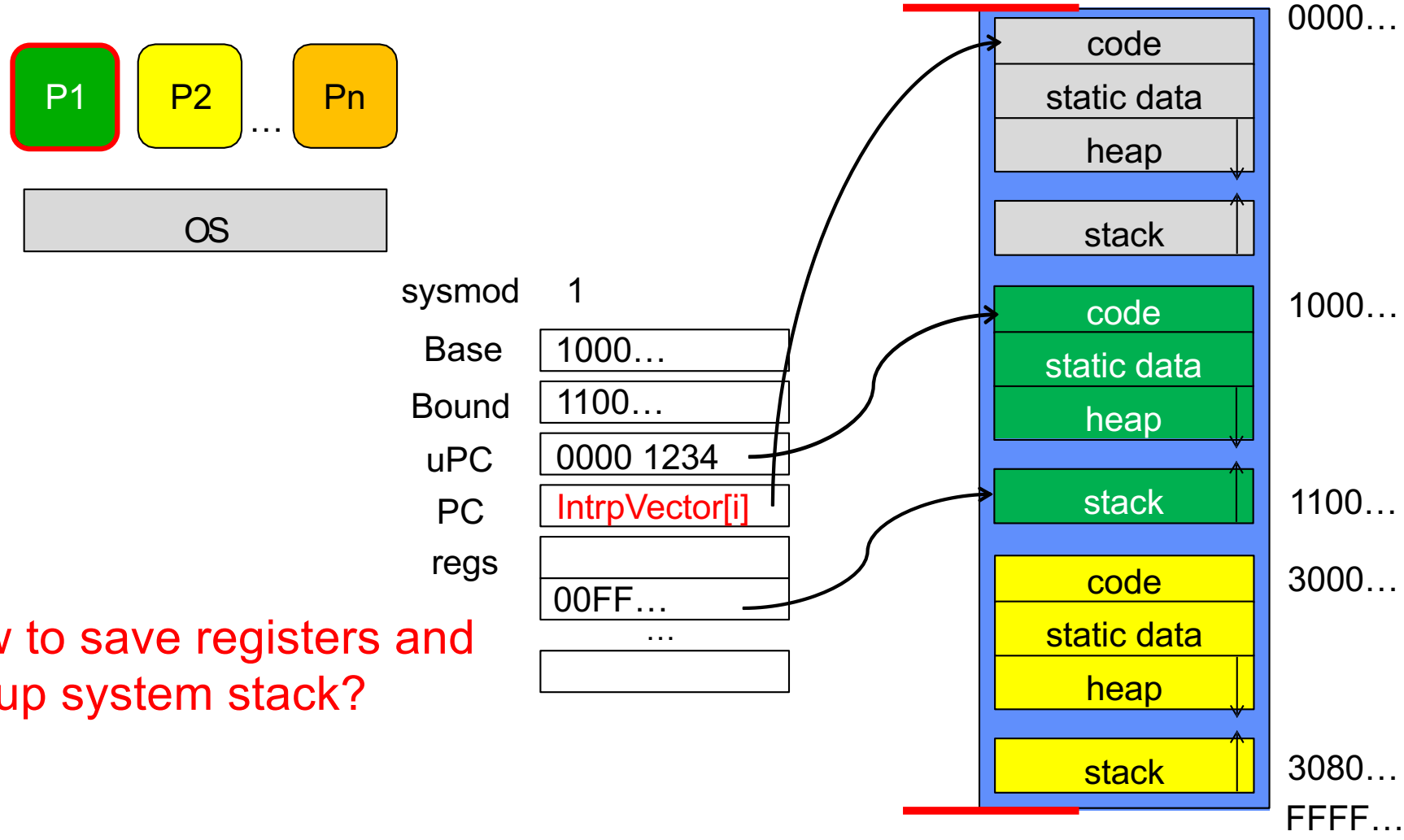
- Where else do you see this dispatch pattern?

Simple B&B: user to kernel



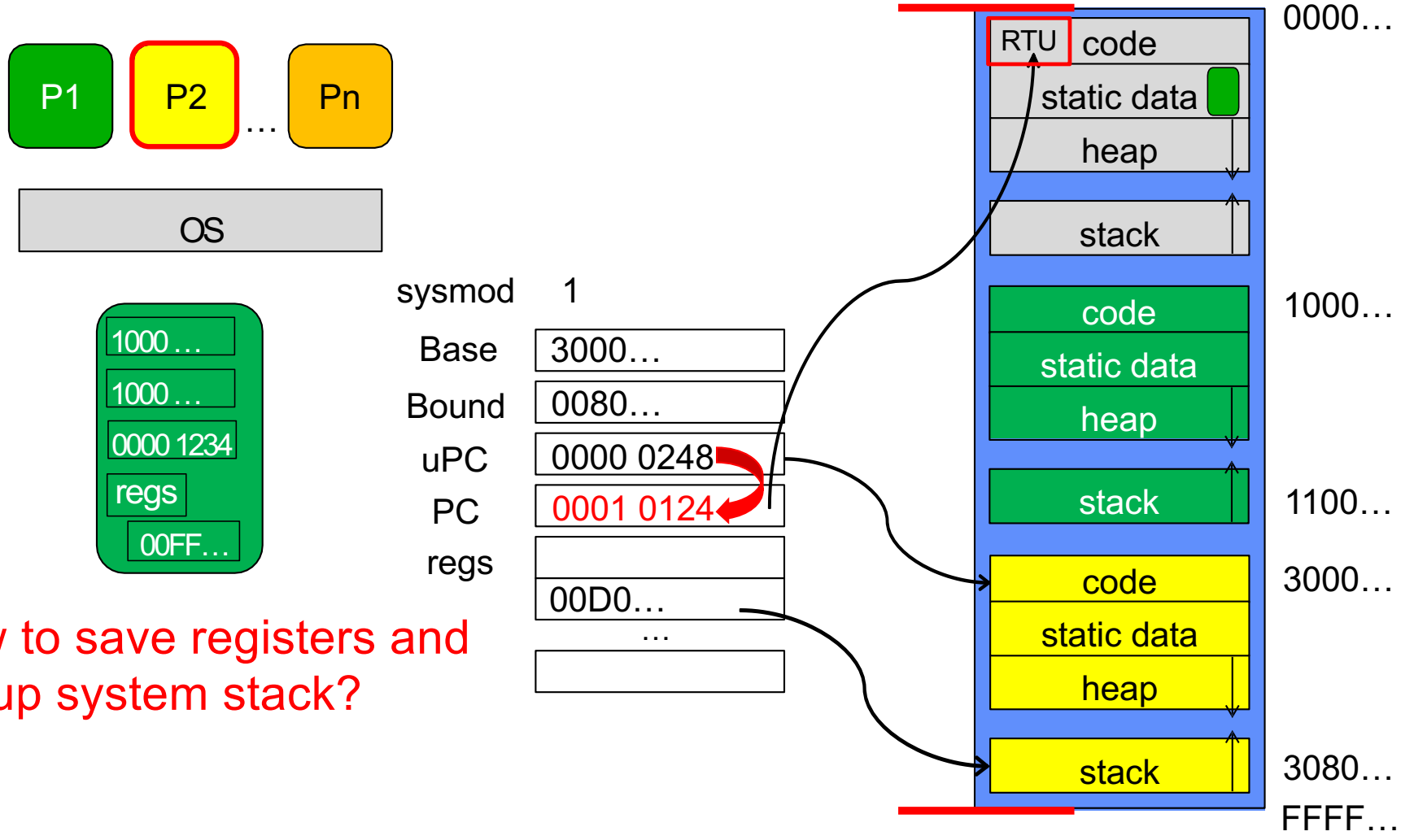
- So how to return to system
  - Time interrupt
  - I/O requests
  - Other things

Simple B&B: interrupt



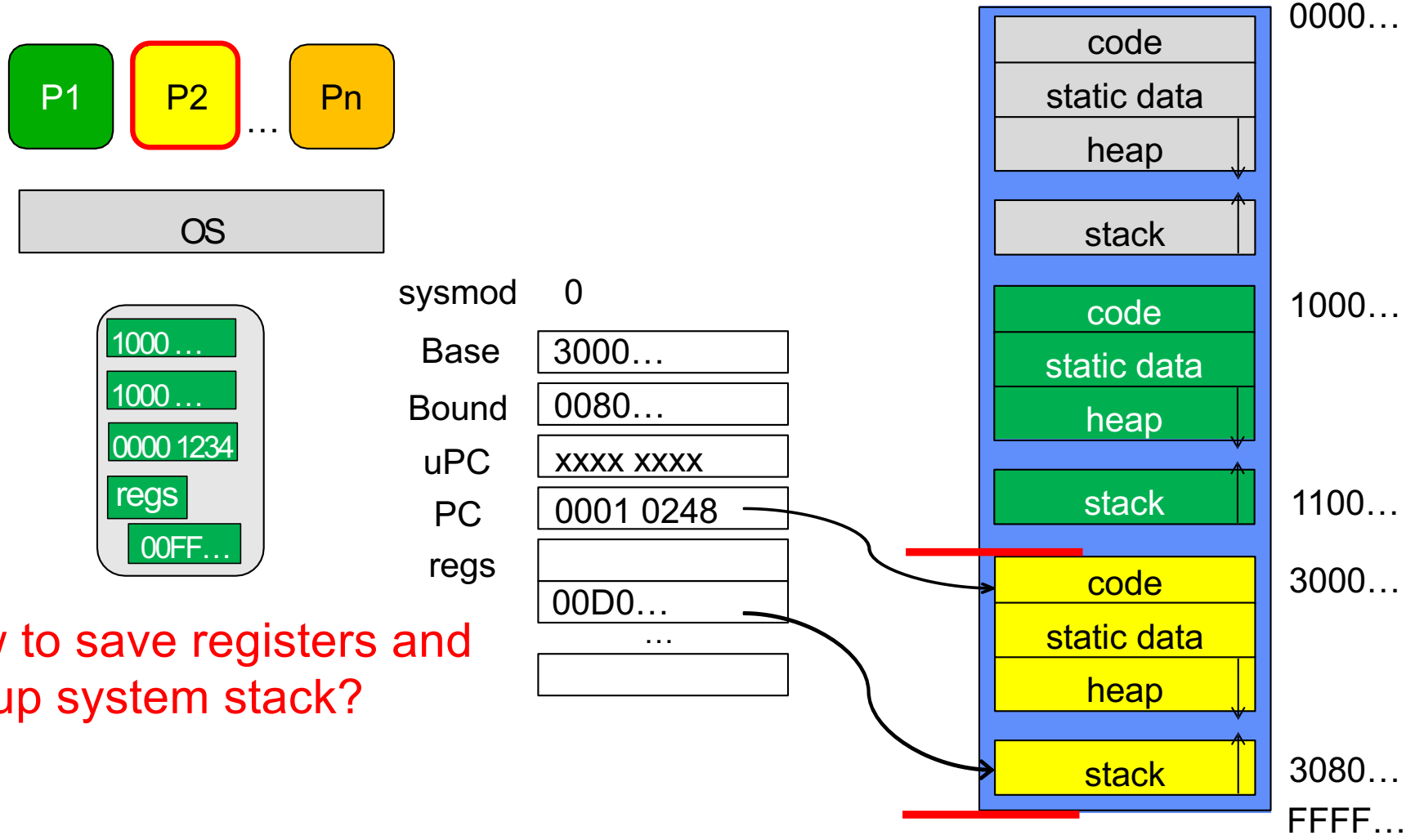
- How to save registers and set up system stack?

Simple B&B: switch user process



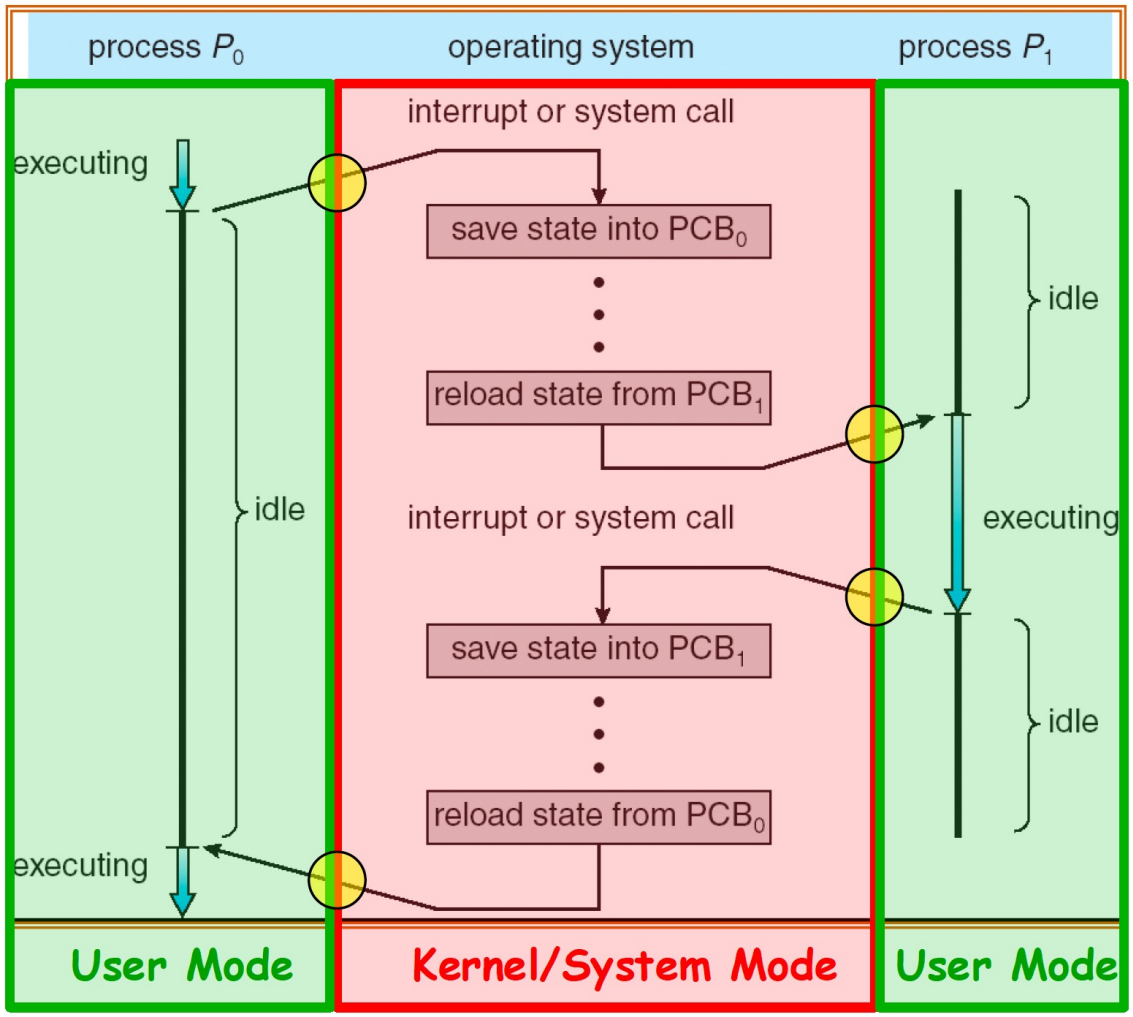
- How to save registers and set up system stack?

Simple B&B: "resume"



- How to save registers and set up system stack?

Another view: CPU switch from process P_0 to process P_1



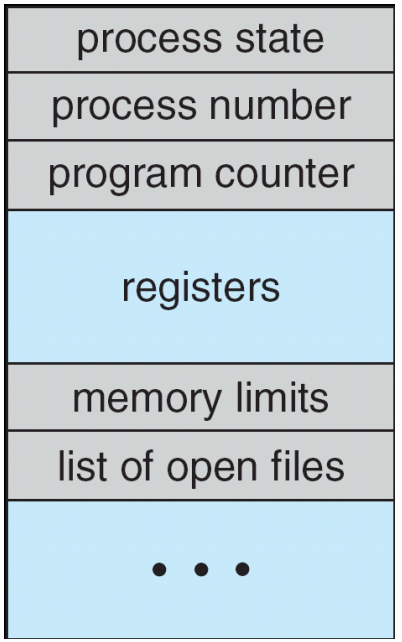
Running many programs?

- 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 decide which user process to run?
  - How do we represent user processes in the OS?
  - 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?
  - …

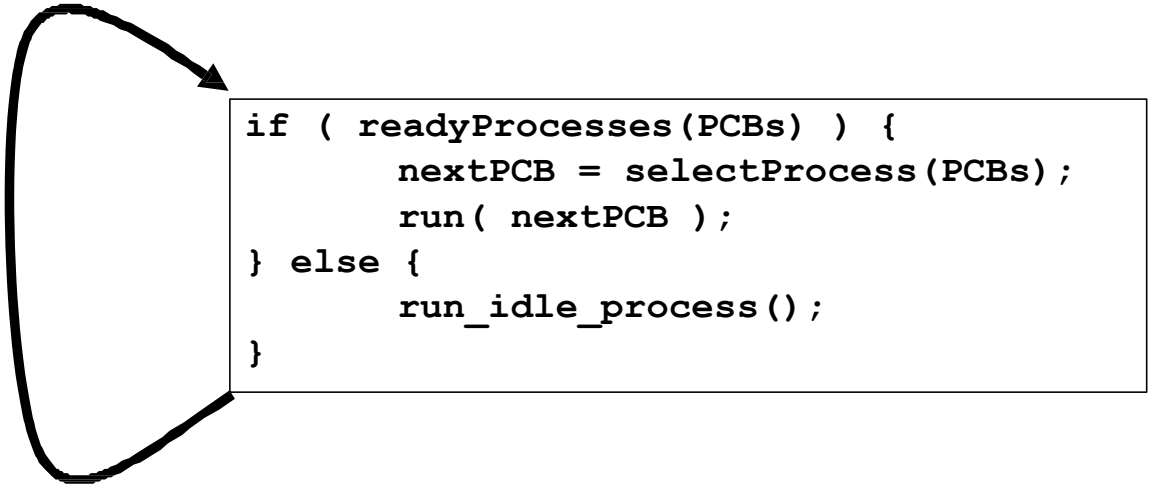
Multiplexing processes: the Process Control Block

- 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



```
if ( readyProcesses(PCBs) ) {
        nextPCB = selectProcess(PCBs);
        run( nextPCB );
} else {
        run_idle_process();
}
```

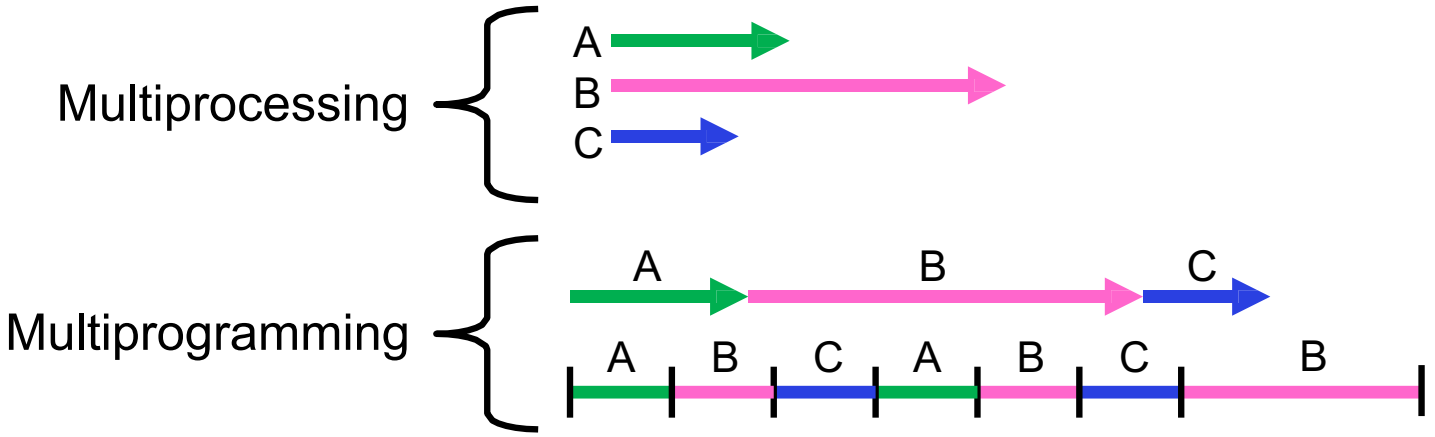
- Scheduling: mechanism for deciding which processes/threads receive hardware CPU time, when, and for how long
- Lots of different scheduling policies provide …
  - Fairness
  - Realtime guarantees
  - Latency optimization
  - …

Going back to threads

- 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 (i.e., process) 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 smal 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):

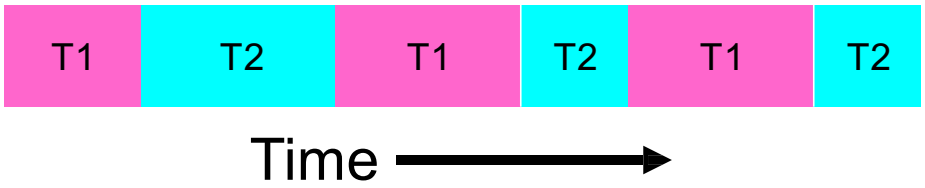
```
main() {
    create_thread(ComputePI, "pi.txt");
    create_thread(PrintClassList, "classlist.txt");
}
```

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



More practical motivation: compute/IO overlap

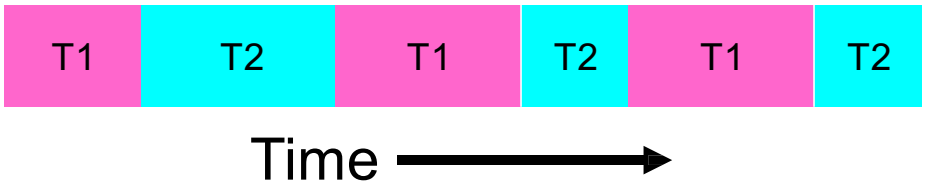
- Back to Jeff Dean's "Numbers Everyone Should Know"

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

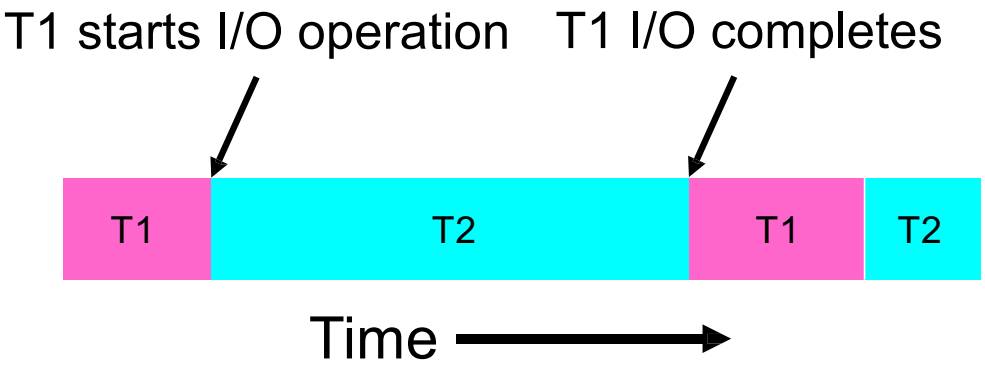
- Handle I/O in a separate thread, avoiding blocking other progress

Threads mask I/O latency

- If no thread performs I/O:



- If thread 1 performs blocking I/O operation:



A better example of 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

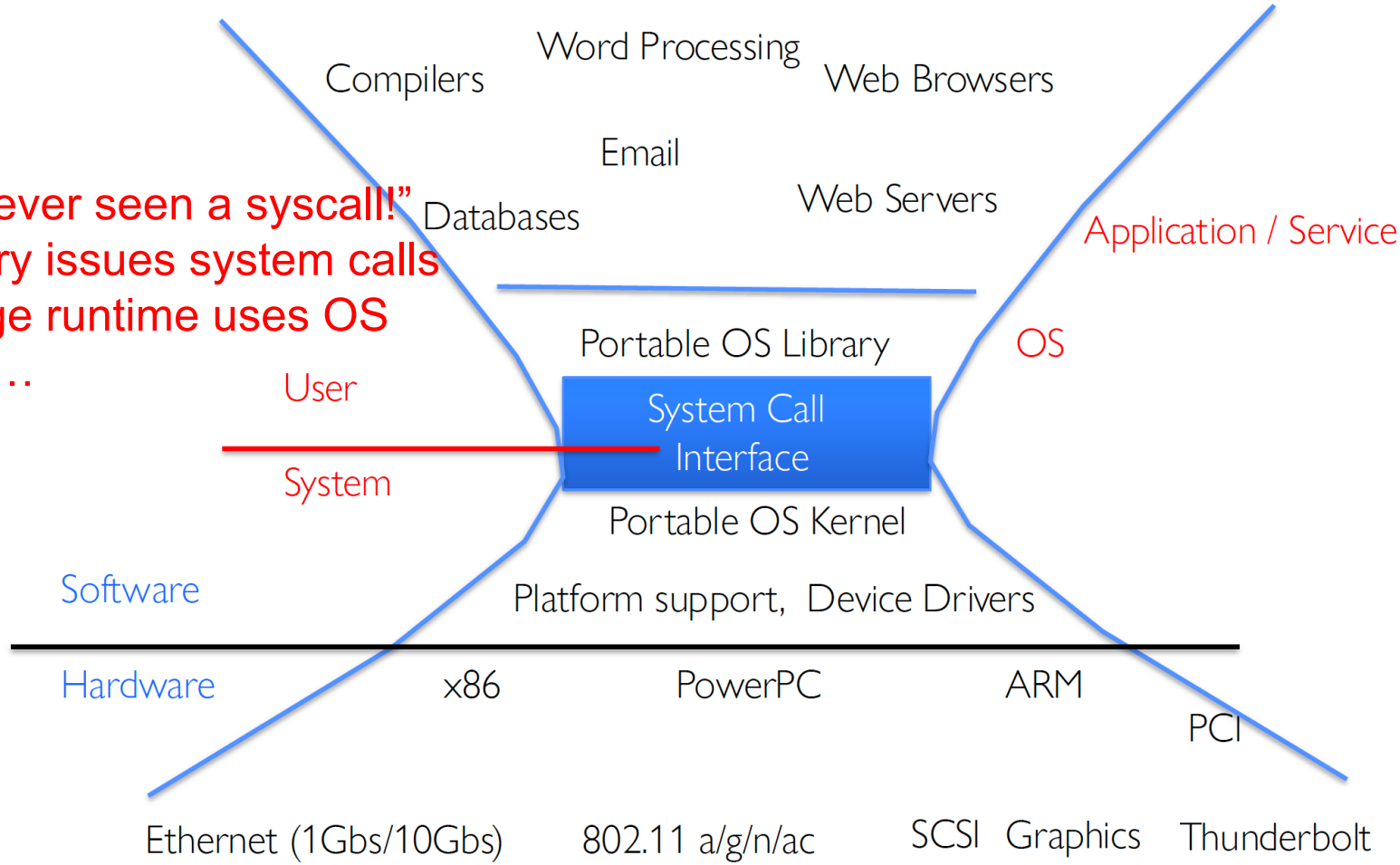
Multithreaded programs

- 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

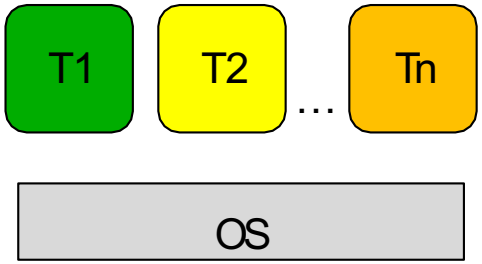
System calls ("syscalls")

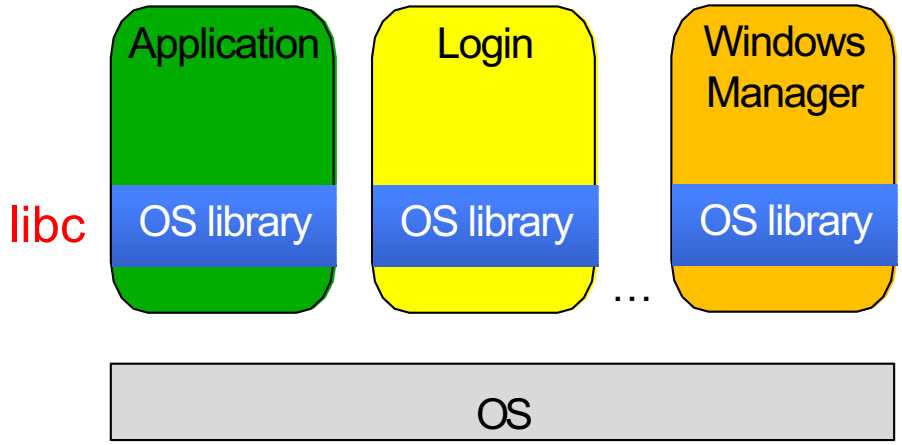
"But I've never seen a syscall!"

- OS library issues system calls
- Language runtime uses OS libraries…



OS library issues syscalls





OS library API for threads: *pthread*

(Here, the "p" is for "POSIX" which is a part of 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)

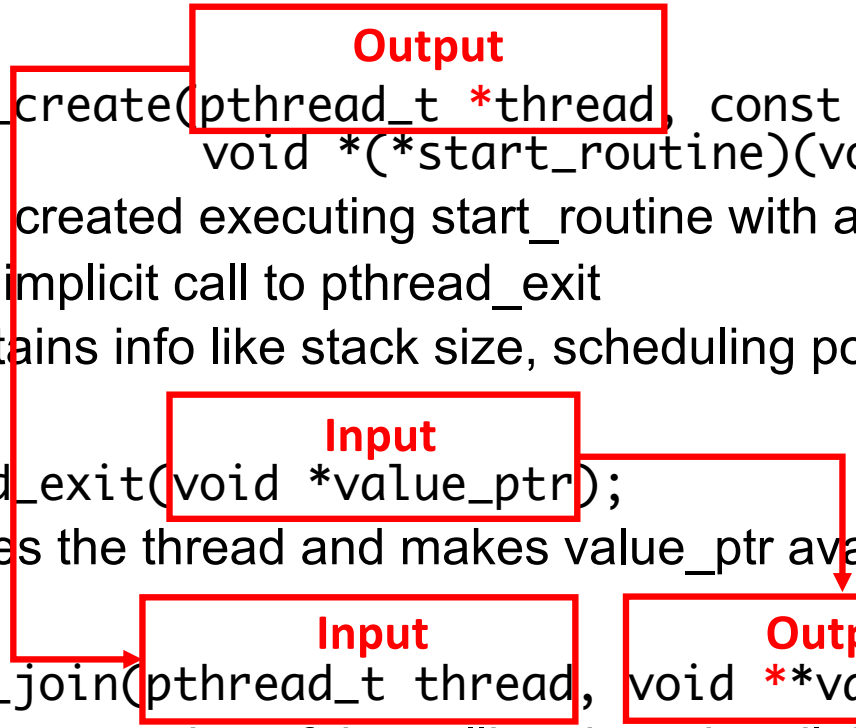
```
void pthread_exit(void *value_ptr);
```

- 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.
- 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.

OS library API for threads: *pthread*

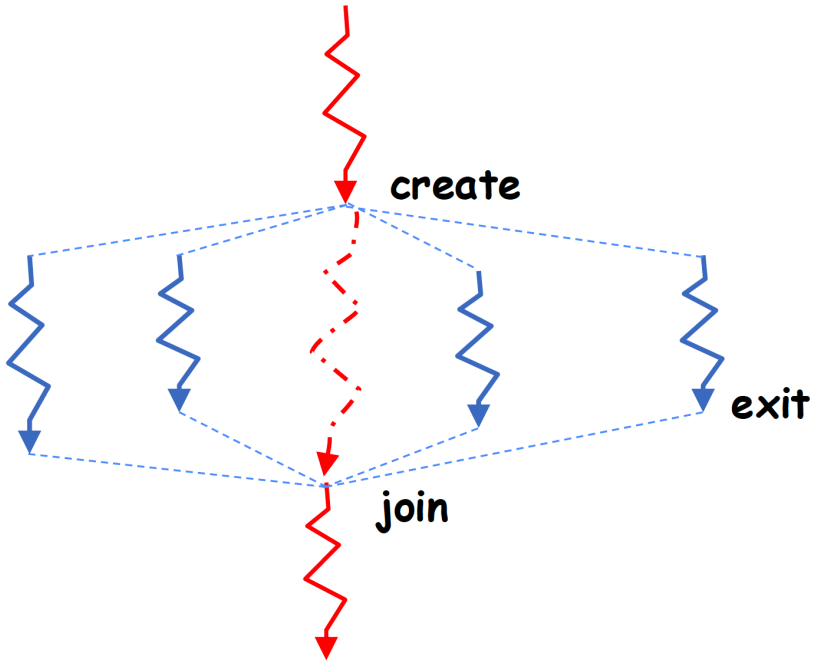


- thread is created executing start_routine with arg as its sole argument.
- return is implicit call to pthread_exit
- (attr contains info like stack size, scheduling policy, etc)

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

- 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.

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.

Pthread examples

- How many threads are in this program?
- What function does each thread run?
- One possible result:

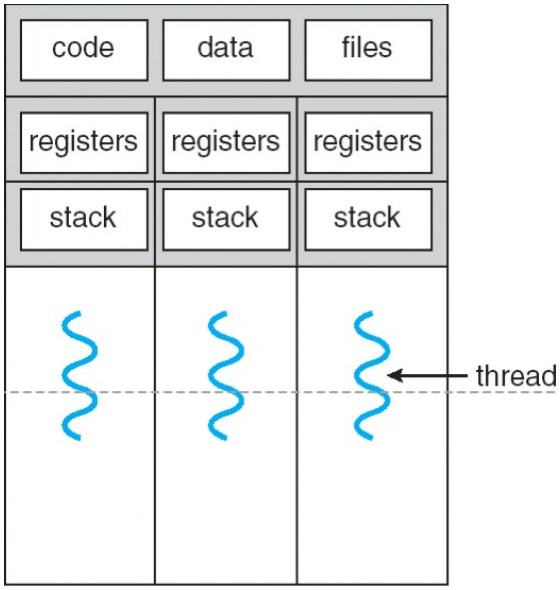
```
(base) CullerMac19:code04 culler$ ./pthread 4
Main stack: 7ffee2c6b6b8, common: 10cf95048 (162)
Thread #1 stack: 70000d83bef8 common: 10cf95048 (162)
Thread #3 stack: 70000d941ef8 common: 10cf95048 (164)
Thread #2 stack: 70000d8beef8 common: 10cf95048 (165)
Thread #0 stack: 70000d7b8ef8 common: 10cf95048 (163)
```

- 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 a race condition!

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

int common = 3150;

void *thread_fun(void *thread_id) {
    long tid = (long) thread_id;
    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, thread_fun, (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);
}
```

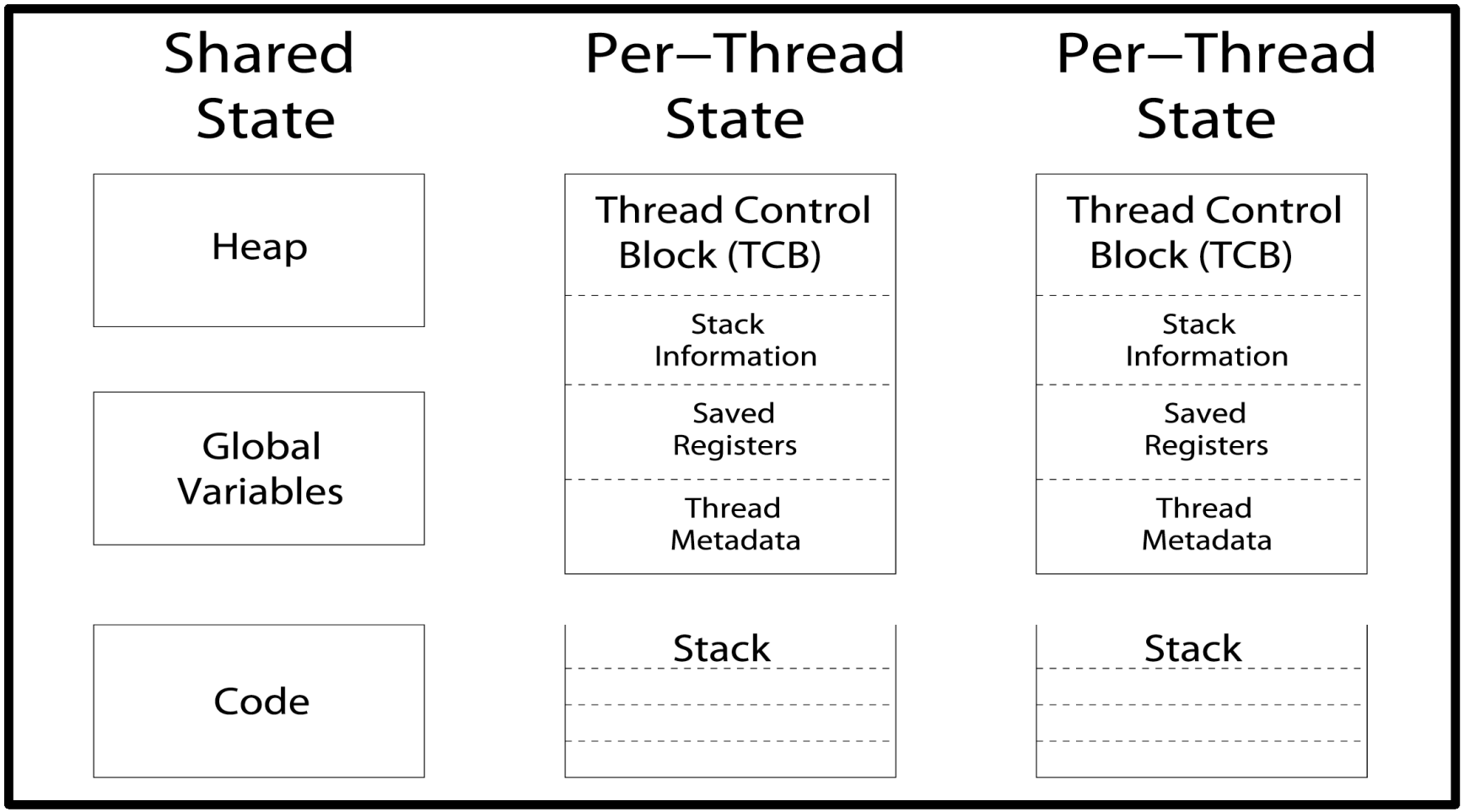
Thread state

- 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 (Thread Control Block)
  - CPU registers (including, program counter)
  - Execution stack
- Execution Stack
  - Parameters, temporary variables
  - Return PCs are kept while called procedures are executing

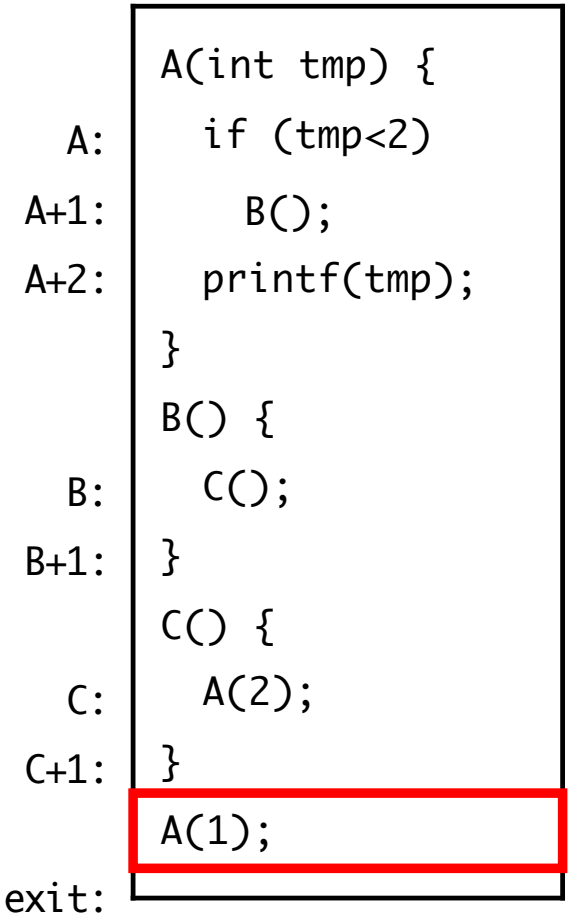


multithreaded process

Shared vs. pre-thread state



Execution stack example



- 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:
```

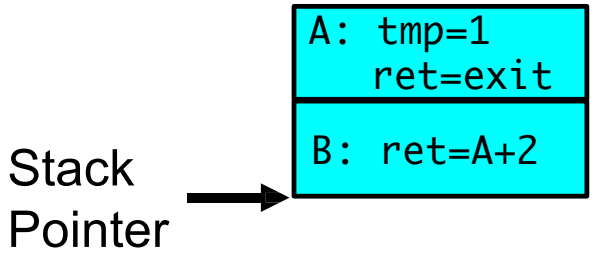
Stack Pointer →

```
A: tmp=1
   ret=exit
```

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

Execution stack example

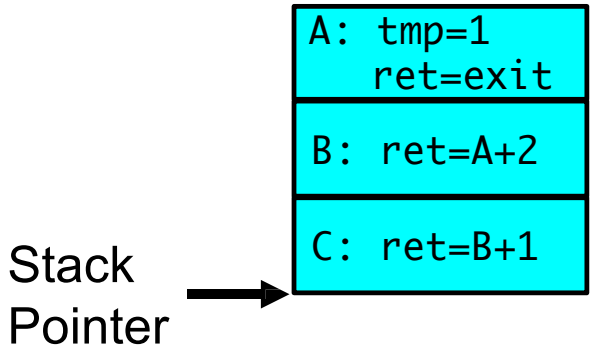
```
        A(int tmp) {
A:        if (tmp<2)
A+1:        B();
A+2:      printf(tmp);
        }
        B() {
B:        C();
B+1:    }
        C() {
C:        A(2);
C+1:    }
        A(1);
exit:
```



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

Execution stack example

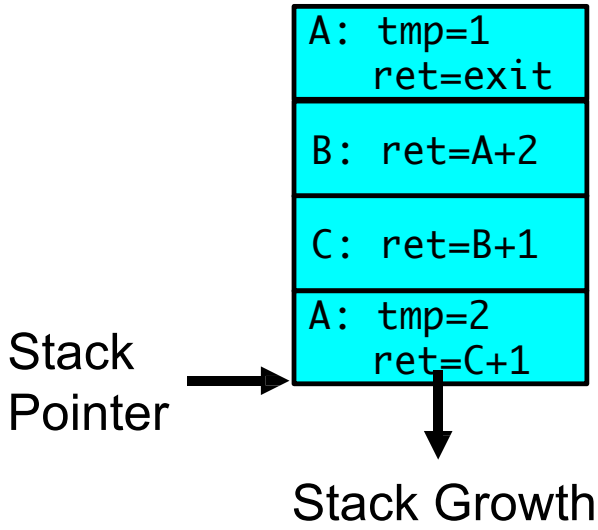
```
        A(int tmp) {
A:        if (tmp<2)
A+1:        B();
A+2:      printf(tmp);
        }
        B() {
B:        C();
B+1:    }
        C() {
C:        A(2);
C+1:    }
        A(1);
exit:
```



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

Execution stack example

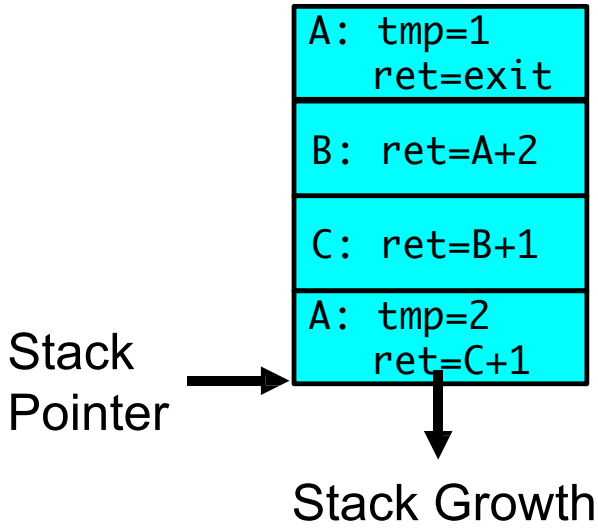
```
        A(int tmp) {
A:        if (tmp<2)
A+1:        B();
A+2:      printf(tmp);
        }
        B() {
B:        C();
B+1:    }
        C() {
C:        A(2);
C+1:    }
        A(1);
exit:
```



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

Execution stack example

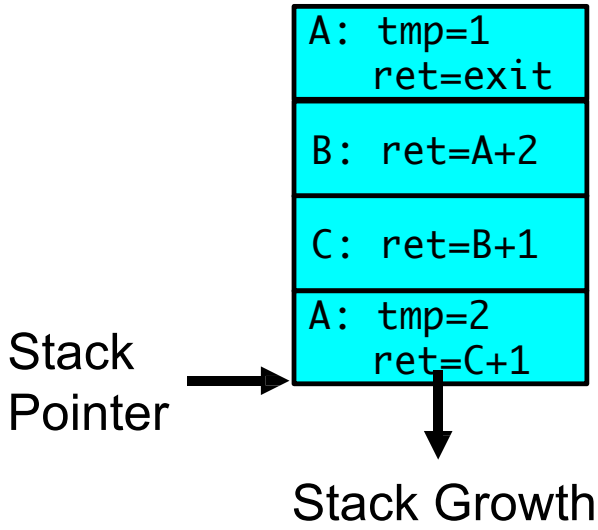
```
        A(int tmp) {
A:        if (tmp<2)
A+1:        B();
A+2:      printf(tmp);
        }
        B() {
B:        C();
B+1:    }
        C() {
C:        A(2);
C+1:    }
        A(1);
exit:
```



- 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:
```

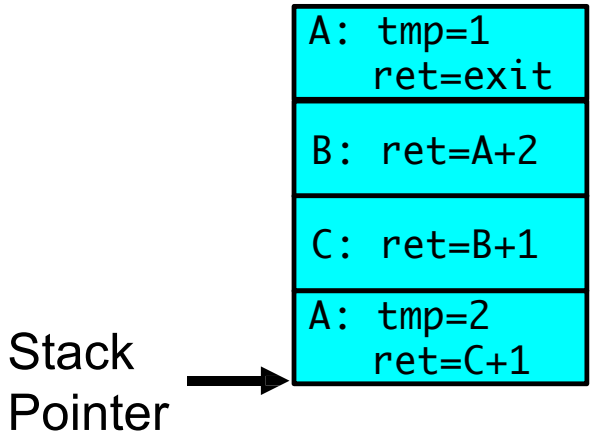


Output: >2

- 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:
```

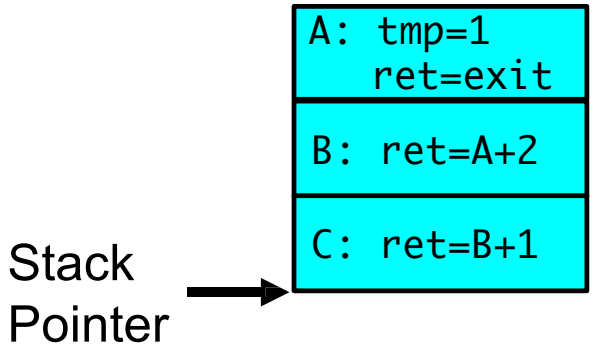


Output: >2

- 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:
```

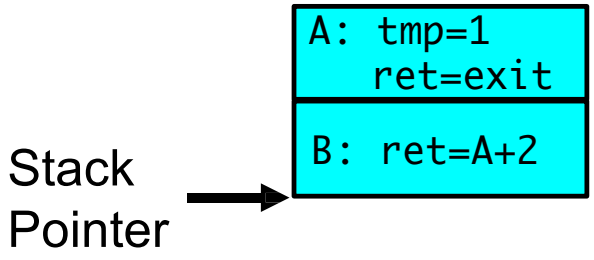


Output: >2

- 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
```

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:
```

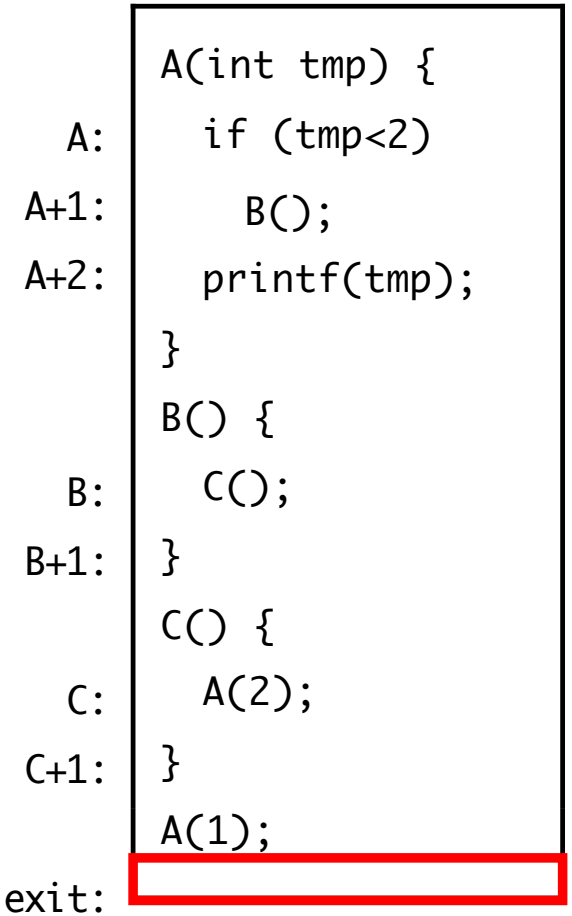
Stack Pointer →

```
A: tmp=1
   ret=exit
```

Output: >2 1

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

Execution stack example

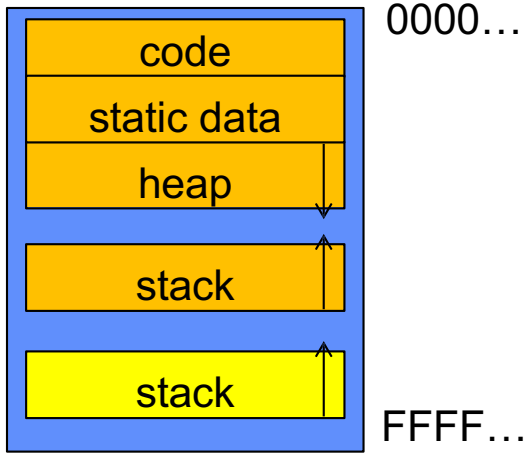


Output: >2 1

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

Memory layout with two threads

- Two sets of CPU registers (where is it?)
- 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?



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 processes
- We saw the role of the OS library
  - Provide API to programs
  - Interface with the OS to request services

Thanks