

# CS161: Operating Systems

Matt Welsh  
mdw@eecs.harvard.edu



Lecture 4: Threads  
February 13, 2007

# Concurrent Programming

Many programs want to do many things “at once”

Web browser:

- Download web pages, read cache files, accept user input, ...

Web server:

- Handle incoming connections from multiple clients at once

Scientific programs:

- Process different parts of a data set on different CPUs

In each case, would like to *share memory* across these activities

- Web browser: Share buffer for HTML page and inlined images
- Web server: Share memory cache of recently-accessed pages
- Scientific programs: Share memory of data set being processes

Can't we simply do this with multiple processes?

# Why processes are not always ideal...

## Processes are not very efficient

- Each process has its own PCB and OS resources
- Typically high overhead for each process: e.g., 1.7 KB per `task_struct` on Linux!
- Creating a new process is often very expensive

## Processes don't (directly) share memory

- Each process has its own address space
- Parallel and concurrent programs often want to directly manipulate the same memory
  - *e.g., When processing elements of a large array in parallel*

## Note: Many OS's provide some form of inter-process shared memory

- cf., UNIX `shmget()` and `shmat()` system calls
  - *Still, this requires more programmer work and does not address the efficiency issues.*

# Can we do better?

What can we share across all of these tasks?

- 
- 
- 

What is private to each task?

- 
- 
-

# Can we do better?

## What can we share across all of these tasks?

- Same code – generally running the same or similar programs
- Same data
- Same privileges
- Same OS resources (files, sockets, etc.)

## What is private to each task?

- Execution state: CPU registers, stack, and program counter

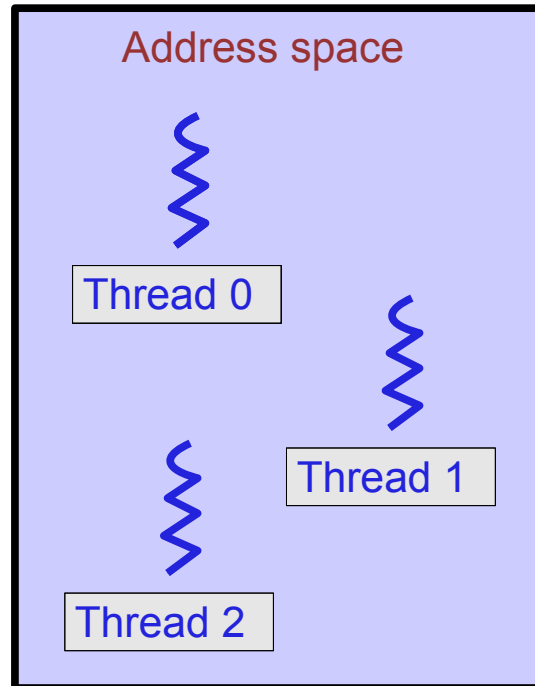
## Key idea of this lecture:

- Separate the concept of a **process** from a **thread of control**
- The **process** is the address space and OS resources
- Each **thread** has its own CPU execution state

# Processes and Threads

Each process has one or more threads “within” it

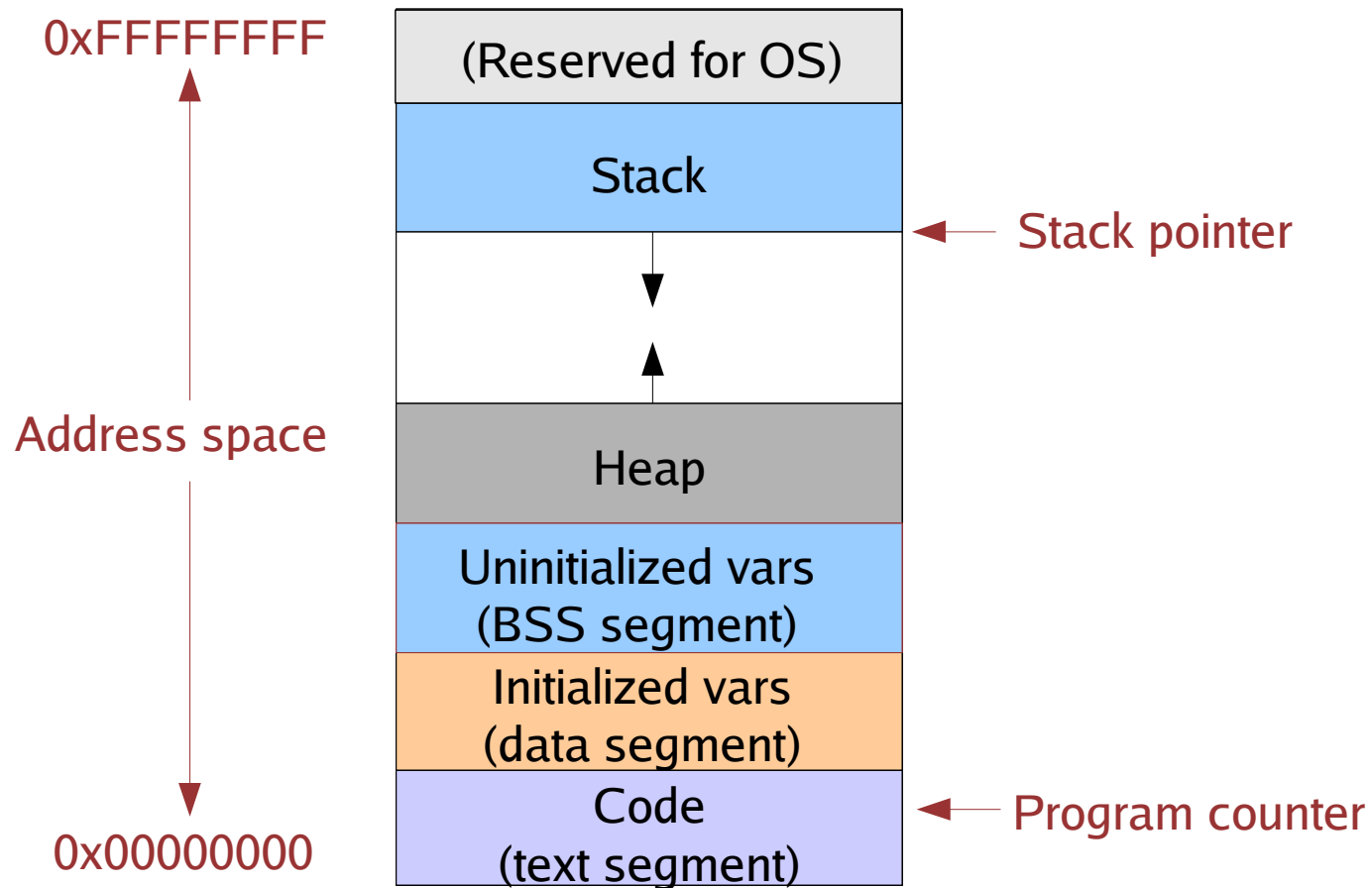
- Each thread has its own stack, CPU registers, etc.
- All threads within a process share the same address space and OS resources
  - *Threads share memory, so they can communicate directly!*



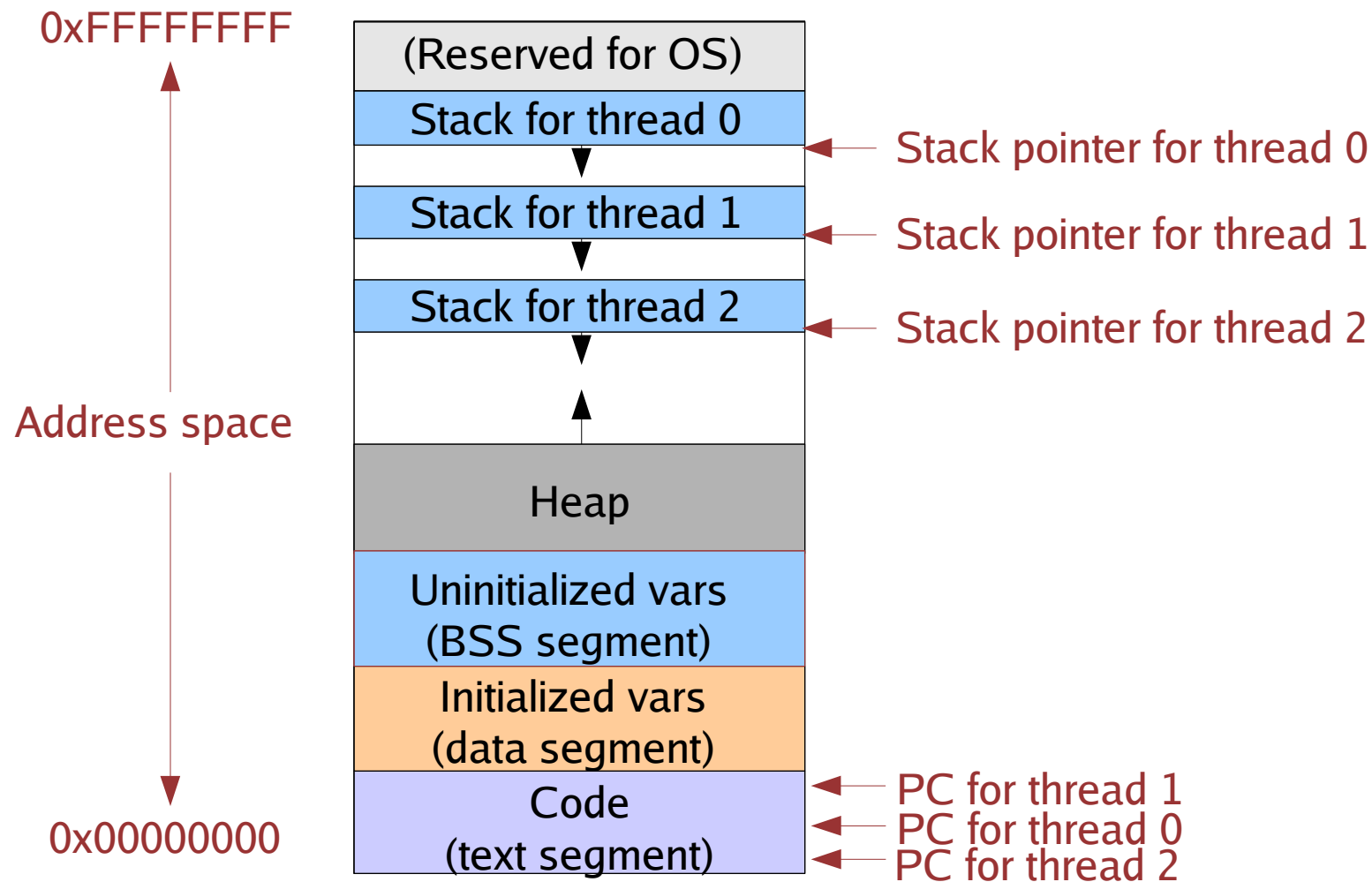
The thread is now the *unit of CPU scheduling*

- A process is just a “container” for its threads
- Each thread is bound to its containing process

# (Old) Process Address Space



# (New) Address Space with Threads



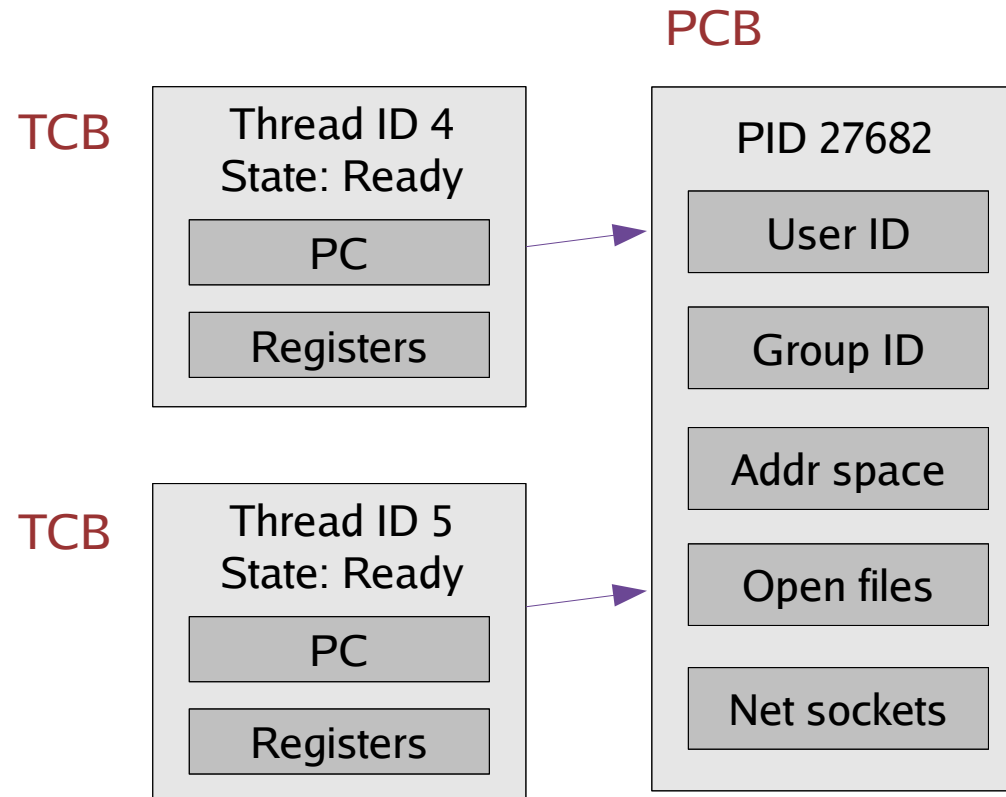
All threads in a single process share the same address space!

# Implementing Threads

Given what we know about processes, implementing threads is “easy”

Idea: Break the PCB into two pieces:

- Thread-specific stuff: Processor state
- Process-specific stuff: Address space and OS resources (open files, etc.)



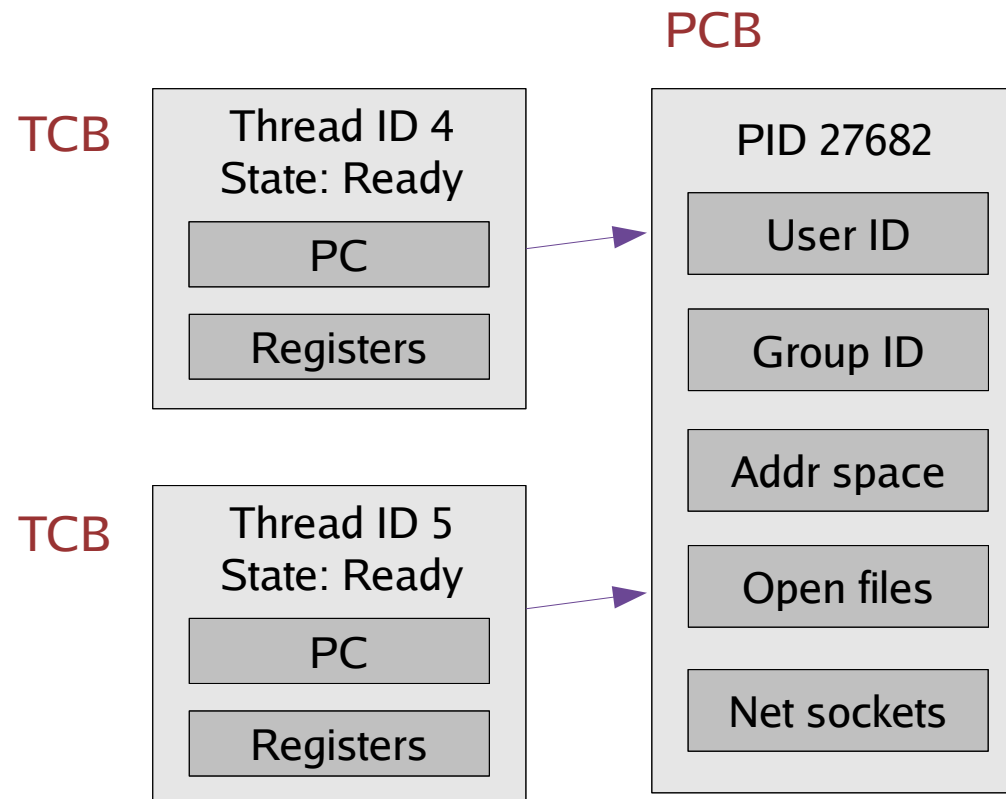
# Thread Control Block (TCB)

TCB contains info on a single thread

- Just processor state and pointer to corresponding PCB

PCB contains information on the containing process

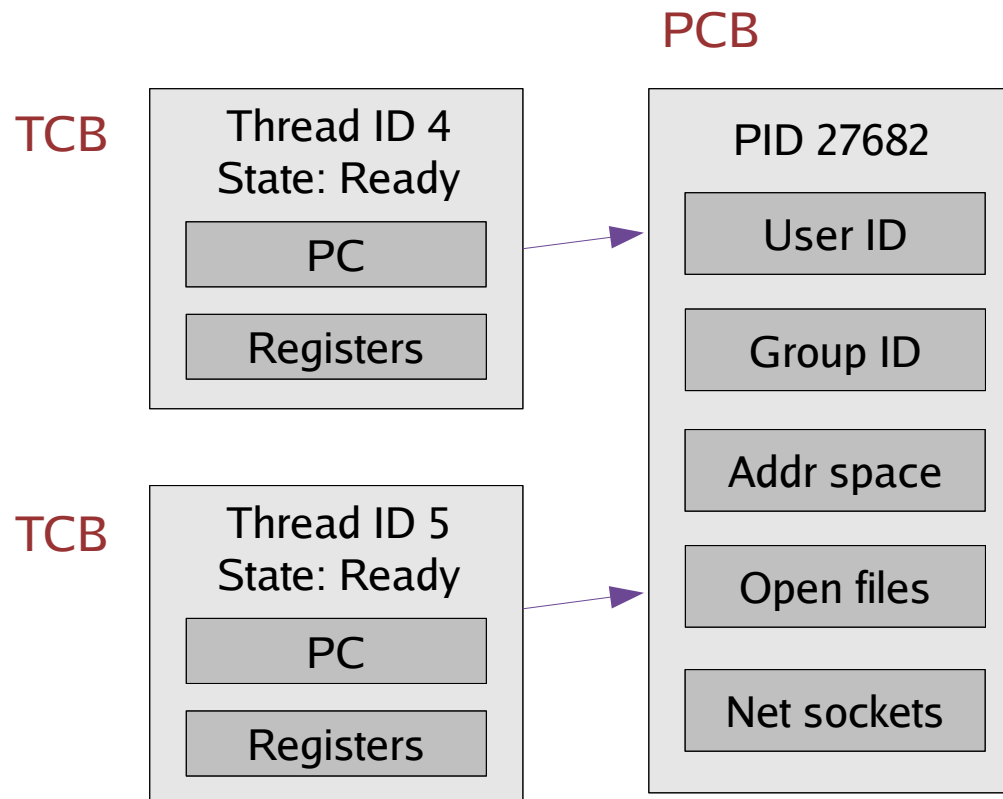
- Address space and OS resources ... but NO processor state!



# Thread Control Block (TCB)

TCB's are smaller and cheaper than processes

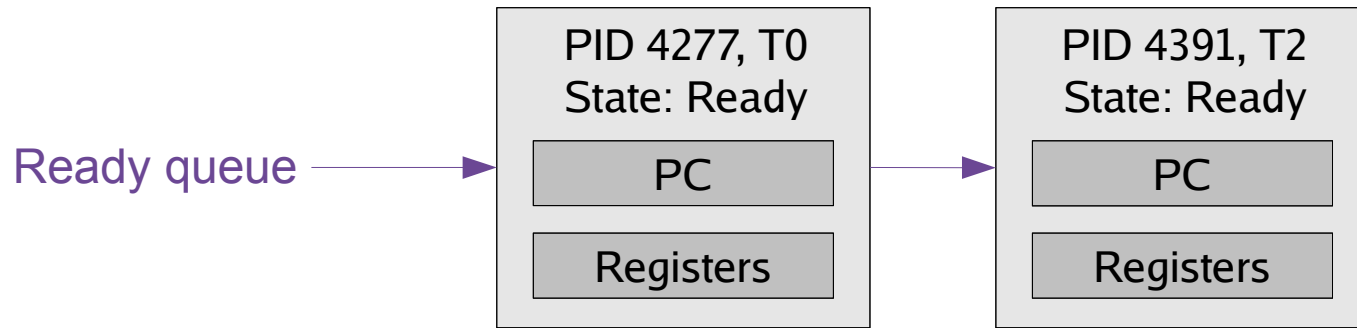
- Linux TCB (thread\_struct) has 24 fields
- Linux PCB (task\_struct) has 106 fields



# Context Switching

TCB is now the unit of a context switch

- Ready queue, wait queues, etc. now contain pointers to TCB's
- Context switch causes CPU state to be copied to/from the TCB



Context switch between two threads in the *same* process:

- No need to change address space

Context switch between two threads in *different* processes:

- Must change address space, sometimes invalidating cache
- This will become relevant when we talk about virtual memory.

# User-Level Threads

Early UNIX designs did not support threads at the kernel level

- OS only knew about processes with separate address spaces

However, can still implement threads as a **user-level library**

- OS does not need to know anything about multiple threads in a process!

How is this possible?

- Recall: All threads in a process share the same address space.
- So, managing multiple threads only requires *switching the CPU state* (PC, registers, etc.)
- **And this can be done directly by a user program without OS help!**

# Implementing User-Level Threads

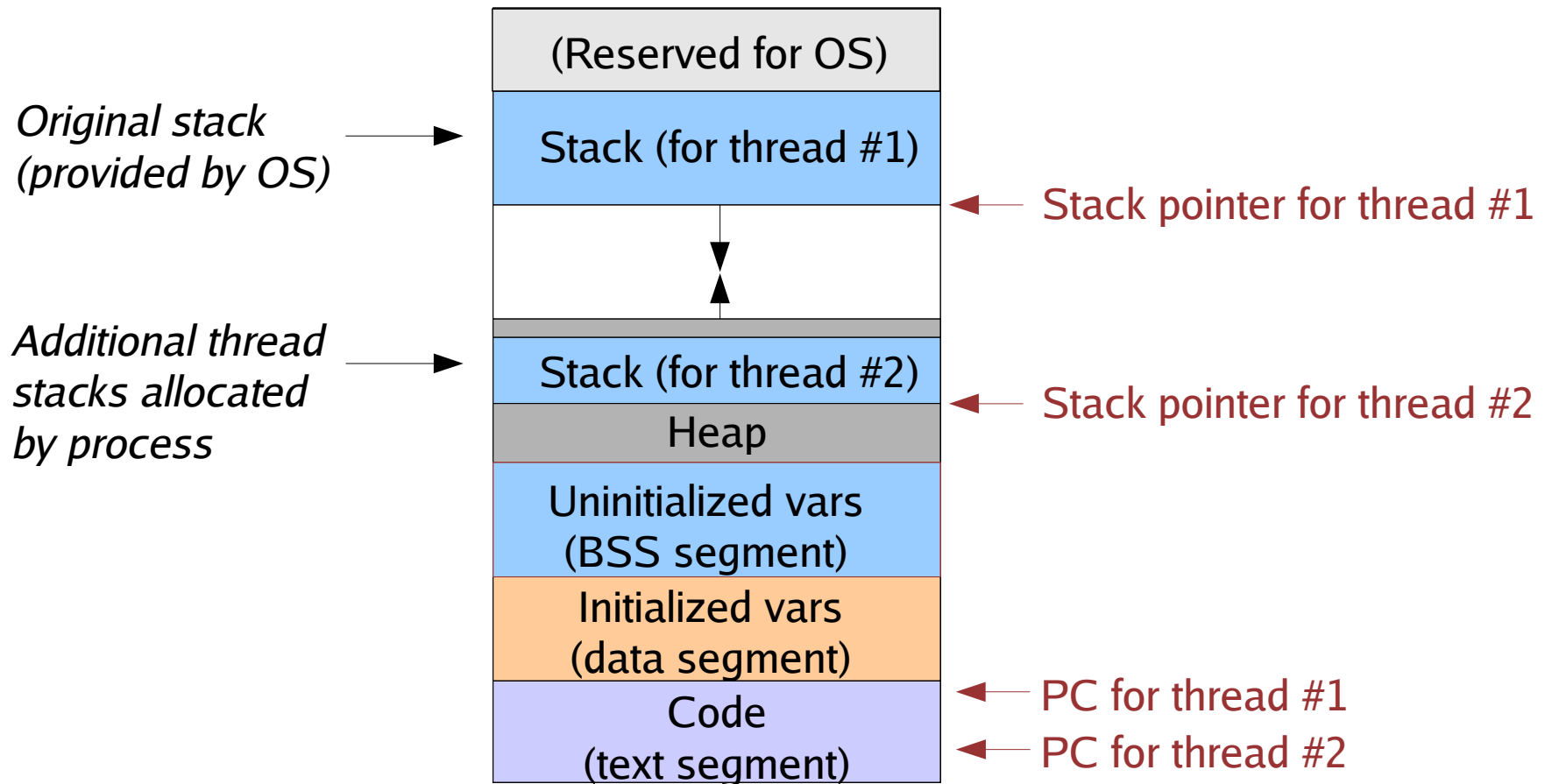
## Alternative to kernel-level threads:

- Implement all thread functions as a user-level library
  - *e.g., libpthread.a*
- OS thinks the process has a single thread
  - *Use the same PCB structure as in the last lecture*
- OS need not know anything about multiple threads in a process!

## How to create a user-level thread?

- **Thread library** maintains a TCB for each thread in the application
  - *Just a linked list or some other data structure*
- Allocate a separate stack for each thread (usually with malloc)

# User-level thread address space



Stacks must be allocated carefully and managed by the thread library.

# User-level Context Switching

How to switch between user-level threads?

Need some way to swap CPU state.

Fortunately, this does not require any privileged instructions!

- So, the threads library can use the same instructions as the OS to save or load the CPU state into the TCB.

Why is it safe to let the user switch the CPU state?

# setjmp() and longjmp()

C standard library routines for saving and restoring processor state.

```
int setjmp(jmp_buf env);
```

- Save current CPU state in the “jmp\_buf” structure

```
void longjmp(jmp_buf env, int returnval);
```

- Restore CPU state from “jmp\_buf” structure, causing corresponding setjmp() call to return with return value “returnval”

```
struct jmp_buf { ... }
```

- Contains CPU-specific fields for saving registers, program counter, etc.

# setjmp/longjmp example

```
int main(int argc, void *argv) {
    int i, restored = 0;
    jmp_buf saved;

    for (i = 0; i < 10; i++) {
        printf("Value of i is now %d\n", i);
        if (i == 5) {
            printf("OK, saving state...\n");
            if (setjmp(saved) == 0) {
                printf("Saved CPU state and breaking from loop.\n");
                break;
            } else {
                printf("Restored CPU state, continuing where we saved\n");
                restored = 1;
            }
        }
    }
    if (!restored) longjmp(saved, 1);
}
```

# setjmp/longjmp example

```
Value of i is now 0
Value of i is now 1
Value of i is now 2
Value of i is now 3
Value of i is now 4
Value of i is now 5
OK, saving state...
Saved CPU state and breaking from loop.
Restored CPU state, continuing where we saved
Value of i is now 6
Value of i is now 7
Value of i is now 8
Value of i is now 9
```

# Preemptive vs. nonpreemptive threads

How to prevent a single user-level thread from hogging the CPU?

Strategy 1: Require threads to cooperate

- Called *non-preemptive threads*
- Each thread must call back into the thread library periodically
  - *This gives the thread library control over the thread's execution*
- *yield()* operation:
  - *Thread voluntarily “gives up” the CPU*

**Pop quiz: What happens when a thread calls yield() ??**

# Preemptive vs. nonpreemptive threads

How to prevent a single user-level thread from hogging the CPU?

Strategy 1: Require threads to cooperate

- Called *non-preemptive threads*
- Each thread must call back into the thread library periodically
  - *This gives the thread library control over the thread's execution*
- *yield()* operation:
  - *Thread voluntarily “gives up” the CPU*

**Pop quiz: What happens when a thread calls yield() ??**

Strategy 2: Use *preemption*

- Thread library tells OS to send it a *signal* periodically
- A signal is like a hardware interrupt
  - *Causes the process to jump into a signal handler*
- The signal handler gives control back to the thread library
  - *Thread library then context switches to a new thread*

# Which approach is better?

## Kernel-level threads:

- Pros:

- 
- 

- Cons:

- 
- 

## User-level threads:

- Pros:

- 
- 

- Cons:

- 
-

# Kernel-level threads

Pro: OS knows about all the threads in a process

- Can assign different scheduling priorities to each one
- Kernel can context switch between multiple threads in one process

Con: Thread operations require calling the kernel

- Creating, destroying, or context switching require system calls

# User-level threads

Pro: Thread operations are very fast

- Typically 10-100x faster than going through the kernel

Pro: Thread state is very small

- Just CPU state and stack, no additional overhead

Con: If one thread *blocks*, it stalls the entire process

- e.g., If one thread waits for file I/O, all threads in process have to wait

Con: Can't use multiple CPUs!

- Kernel only knows about one CPU context

Con: OS may not make good decisions

- Could schedule a process with only idle threads
- Could deschedule a process with a thread holding a lock

# Threads programming interface

Standard API called *POSIX threads*

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

- `thread`: Returns a pointer to the new TCB
- `attr`: Set of attributes for the new thread
  - *Scheduling policy, etc.*
- `start_routine`: Function pointer to “main function” for new thread
- `arg`: Argument to `start_routine()`

```
void pthread_exit(void *retval);
```

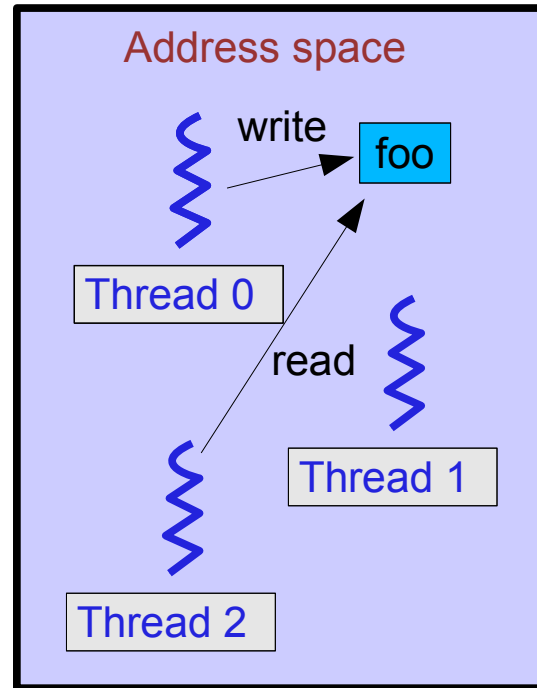
- Exit with the given return value

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

- Waits for “thread” to exit, returns return val of the thread

# Thread Issues

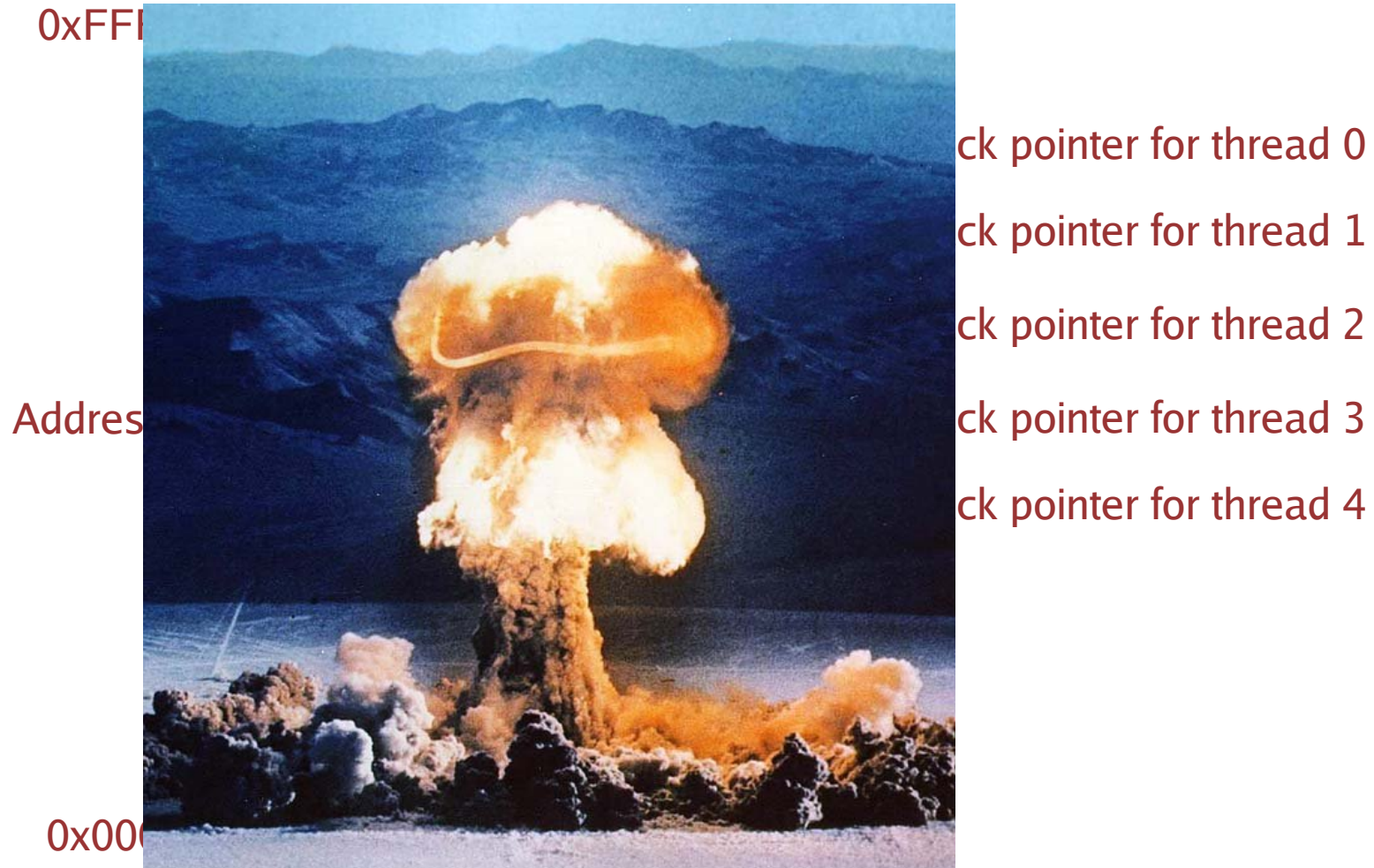
All threads in a process share memory:



- What happens when two threads access the same variable?
- Which value does Thread 2 see when it reads “foo” ?
- What does it depend on?

# Thread Issues Continued

How many threads can the system support?



# Administrative Stuff

## Next Lecture: Synchronization

- How do we prevent multiple threads from stomping on each other's memory?
- How do we get threads to coordinate their activity?
  - *This will be one of the most important lectures in the course...*

Read Tanenbaum 2.3-2.4