HARVARD
School of Engineering
and Applied Sciences

# Threads and concurrency

*CS61, Lecture 17*

Prof. Stephen Chong

November 1, 2011

# Announcements

- Midterm summary
  - Mean 52.8, Std dev 11.7
  - Median 52, lower quartile 45, upper quartile 62
- Midterm pickup
  - From Maxwell Dworkin 143
- Midterm regrades
  - Give me your midterm if you would like it to be regraded
  - Q6: Partial credit for a functionally correct circuit; full credit for a functionally correct circuit that uses few gates and does not use a wire with constant value
- Assignment 5: Bank simulation concurrency lab
  - Will be released today
  - Go to website for instructions

# Topics for today

- Threads: Allowing a single program to do multiple things concurrently.
- Implementing
- Scheduling
- Programming with threads (pthreads library)
- Shared vs. private resources
- The need for synchronization

# Concurrent Programming

- Many programs want to do many things “at once”
  - Web browser:
    - Download web pages, read cache files, accept user input, ...
  - Servers:
    - Handle incoming requests from multiple clients at once
  - Scientific programs:
    - Process different parts of a data set on different CPUs
- We can do more than one thing at a time using processes!
  - Fork a new process to concurrently perform a task

# Why processes are not always ideal...

- Processes are not very efficient
  - Each process has its own page table, file table, open sockets, ...
  - 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
    - e.g., 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?

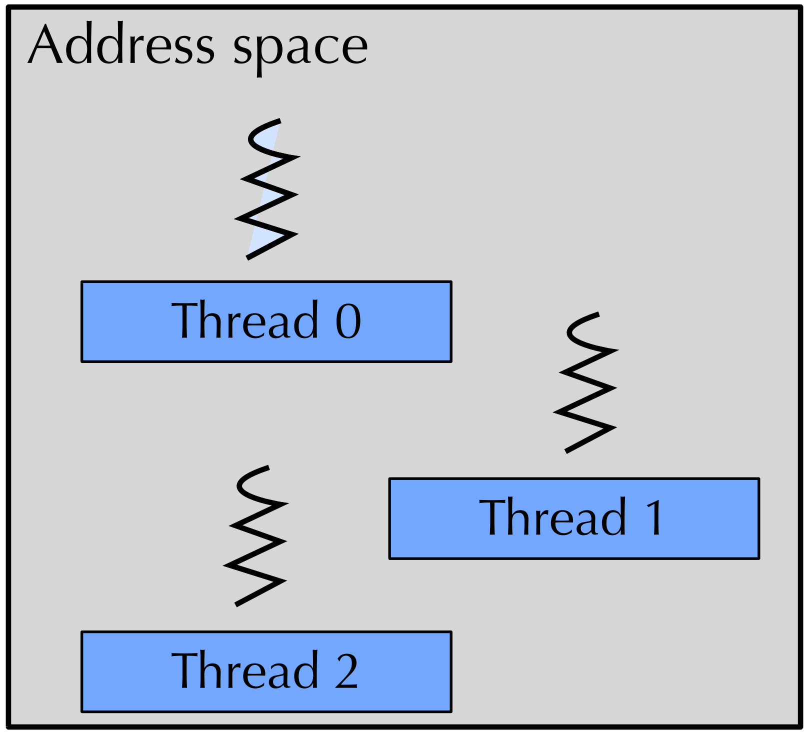
- What is private to each task?

# Can we do better?

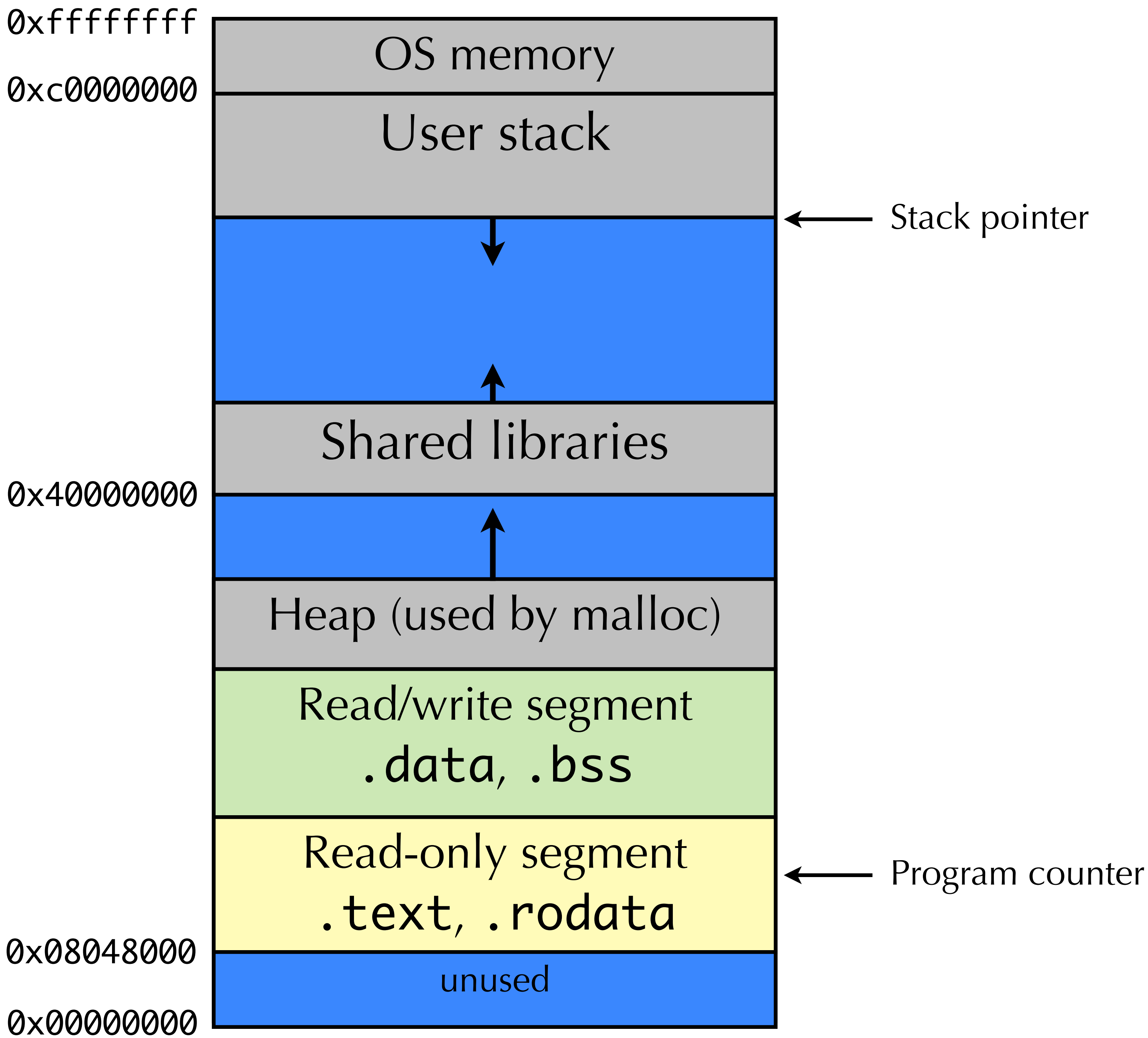
- What can we share across all of these tasks?
  - Same code – generally running the same or similar programs
  - Same data
  - Same privileges
  - Same OS resources (files, sockets, etc.)
- What is private to each task?
  - Execution state: CPU registers, stack, and program counter
- Key idea:
  - Separate 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

# Threads and Processes

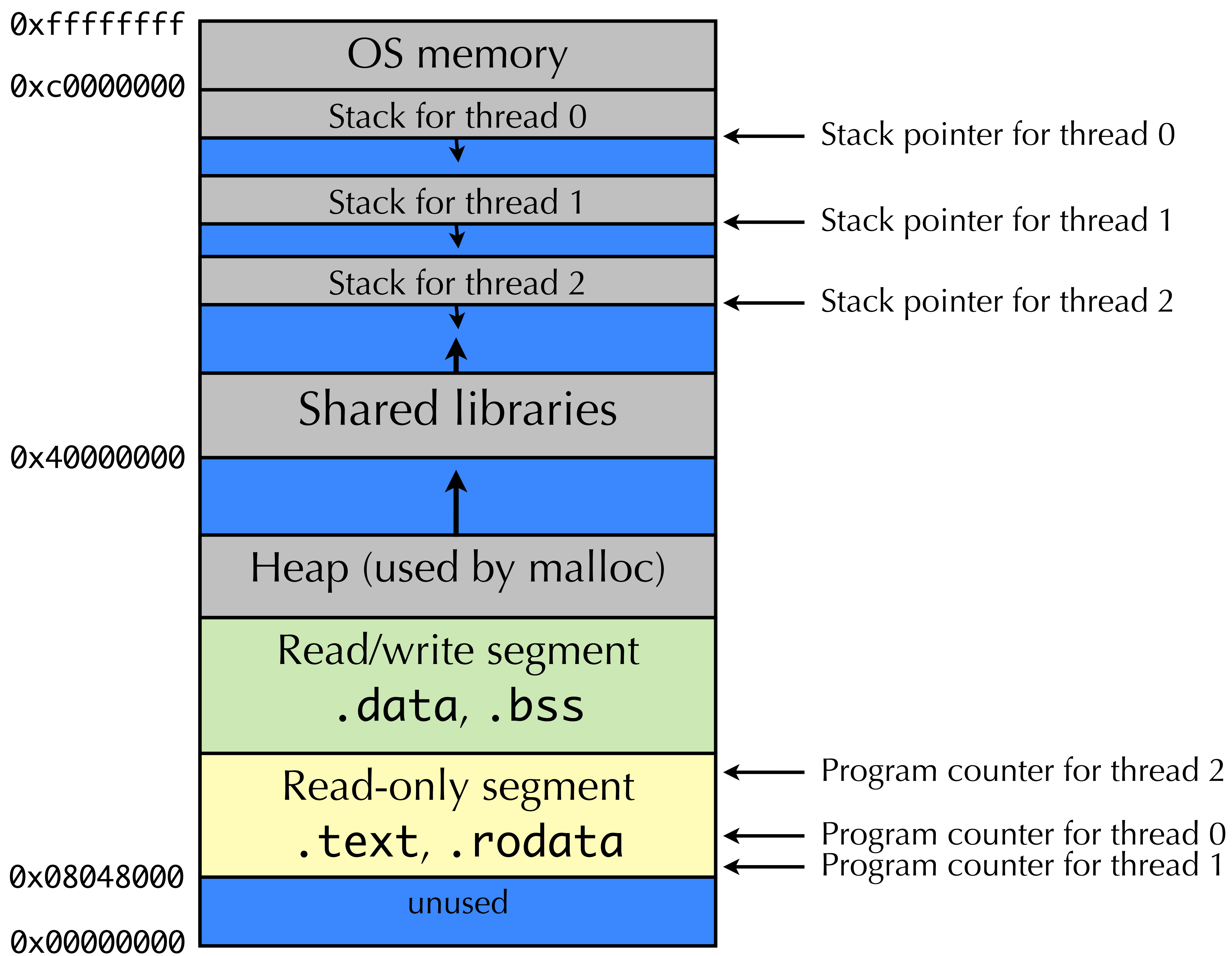
- A **thread** is a logical flow of control that runs in the context of a process
- Each process has one or more threads “within” it
  - Each process begins with a single **main** thread
  - Threads can create new threads, called **peer** threads
- Each thread has its own stack, stack pointer, program counter, and other CPU registers.
  - All threads within a process share the same address space and OS resources
    - Threads share memory, so they can communicate directly!



# (Old) Process Address Space



# (New) Address Space with Threads

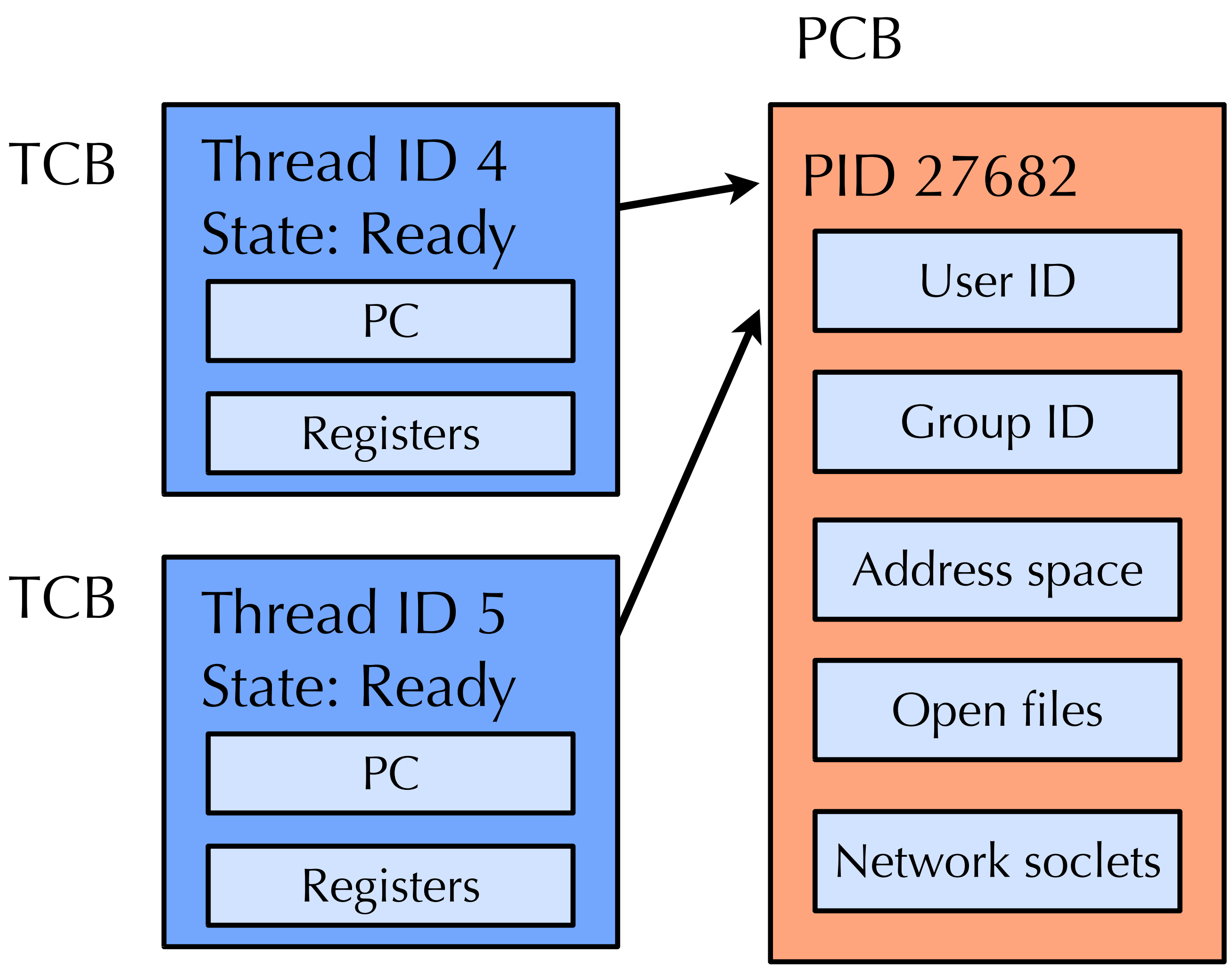


# Topics for today

- Threads: Allowing a single program to do multiple things concurrently.
- Implementing
- Scheduling
- Programming with threads (pthreads library)
- Shared vs. private resources
- The need for synchronization

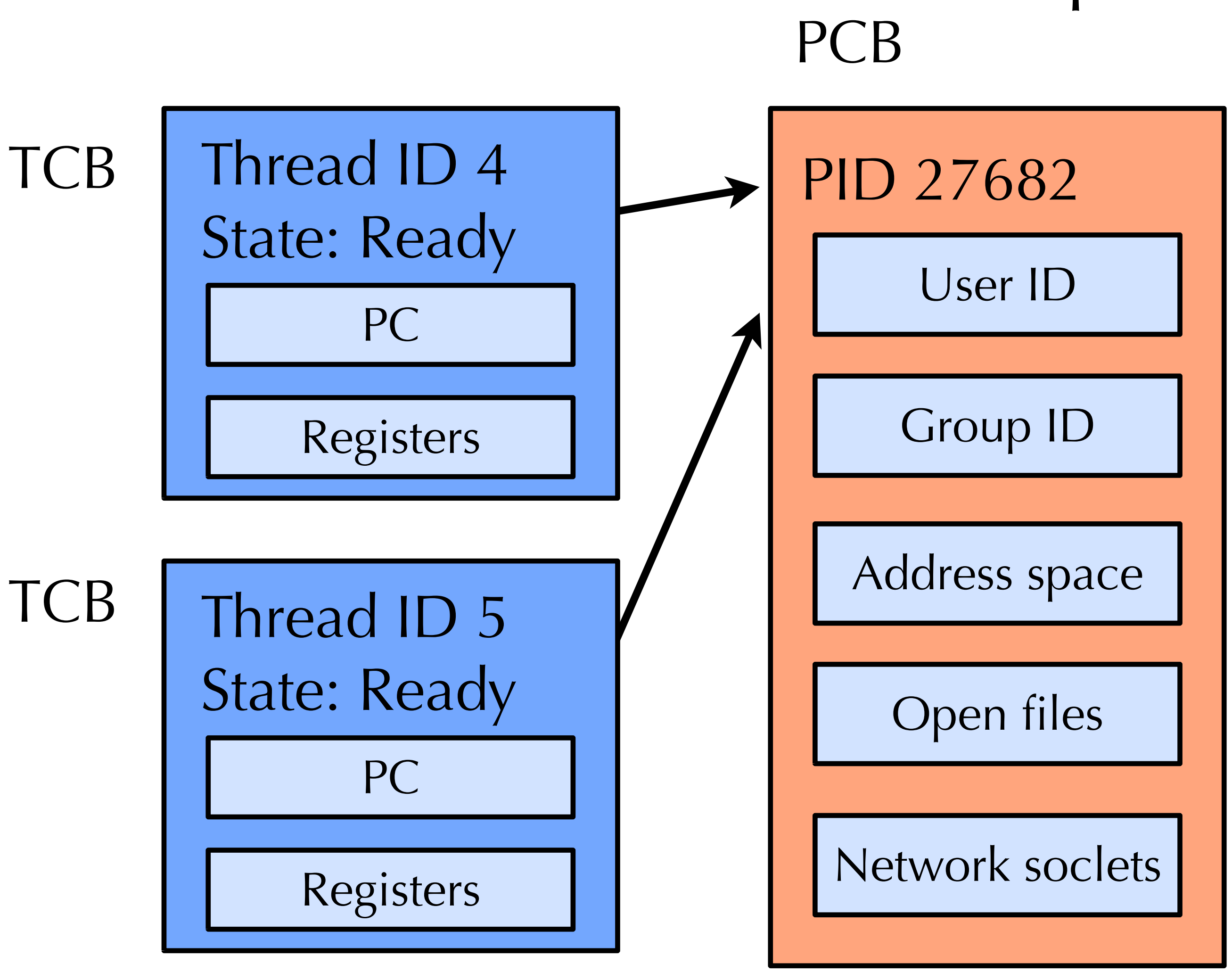
# Implementing Threads

- Operating system maintains two internal data structures:
  - **Thread control block** (**TCB**) – One for each thread
  - **Process control block** (**PCB**) – One for each process
- Each TCB points to its “container” PCB.



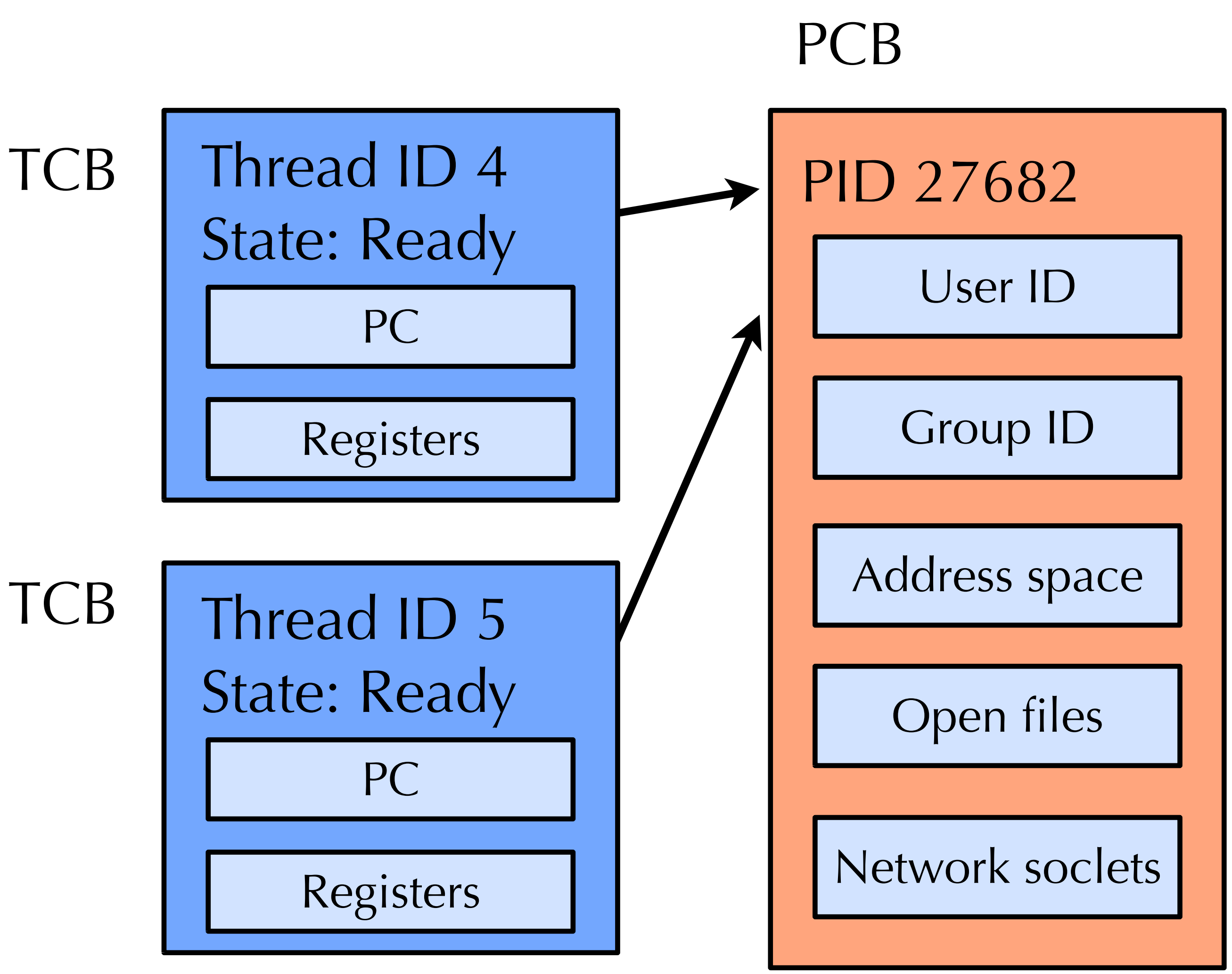
# Thread Control Block (TCB)

- 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



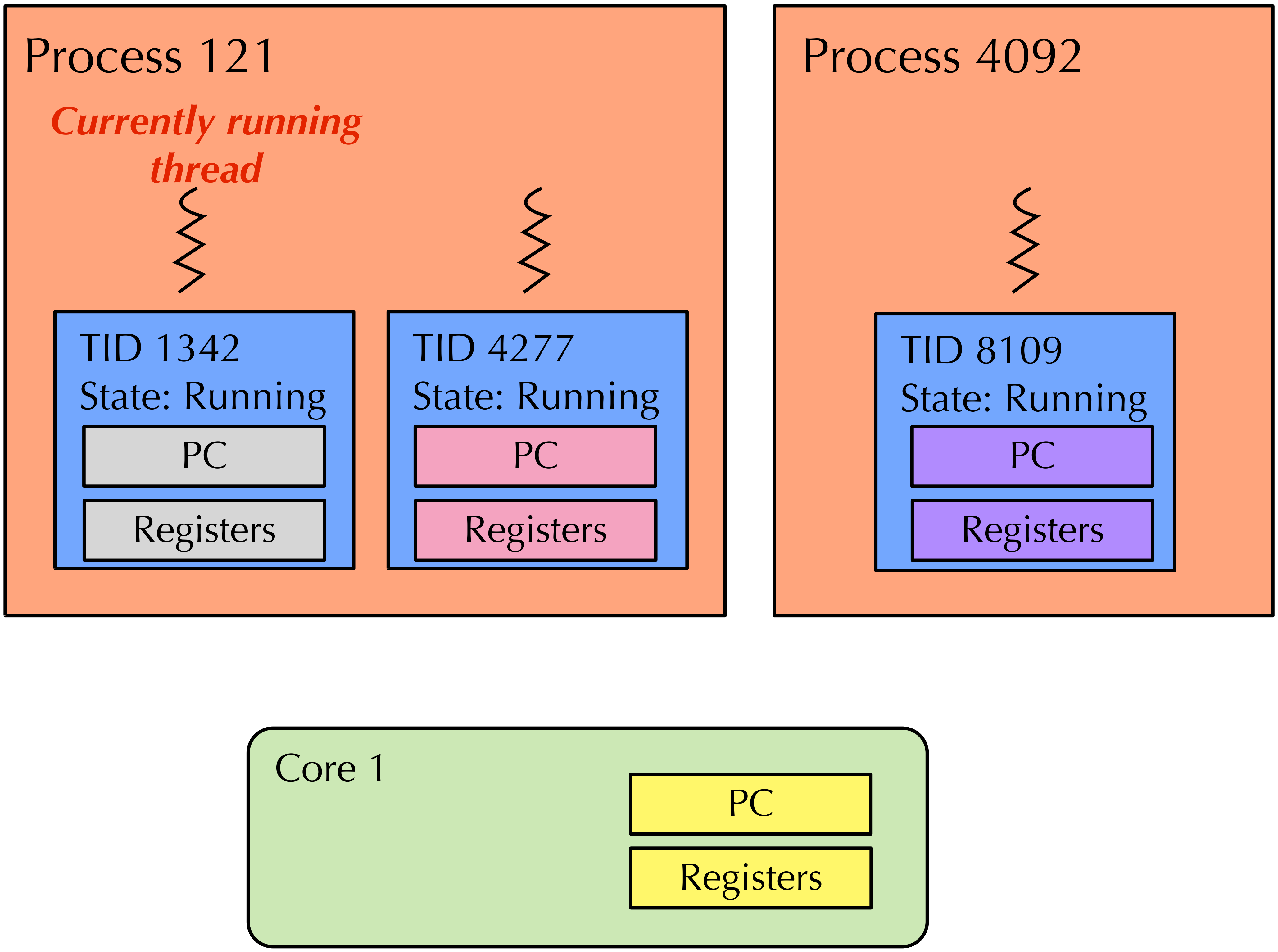
# Topics for today

- Threads: Allowing a single program to do multiple things concurrently.
- Implementing
- Scheduling
- Programming with threads (pthreads library)
- Shared vs. private resources
- The need for synchronization

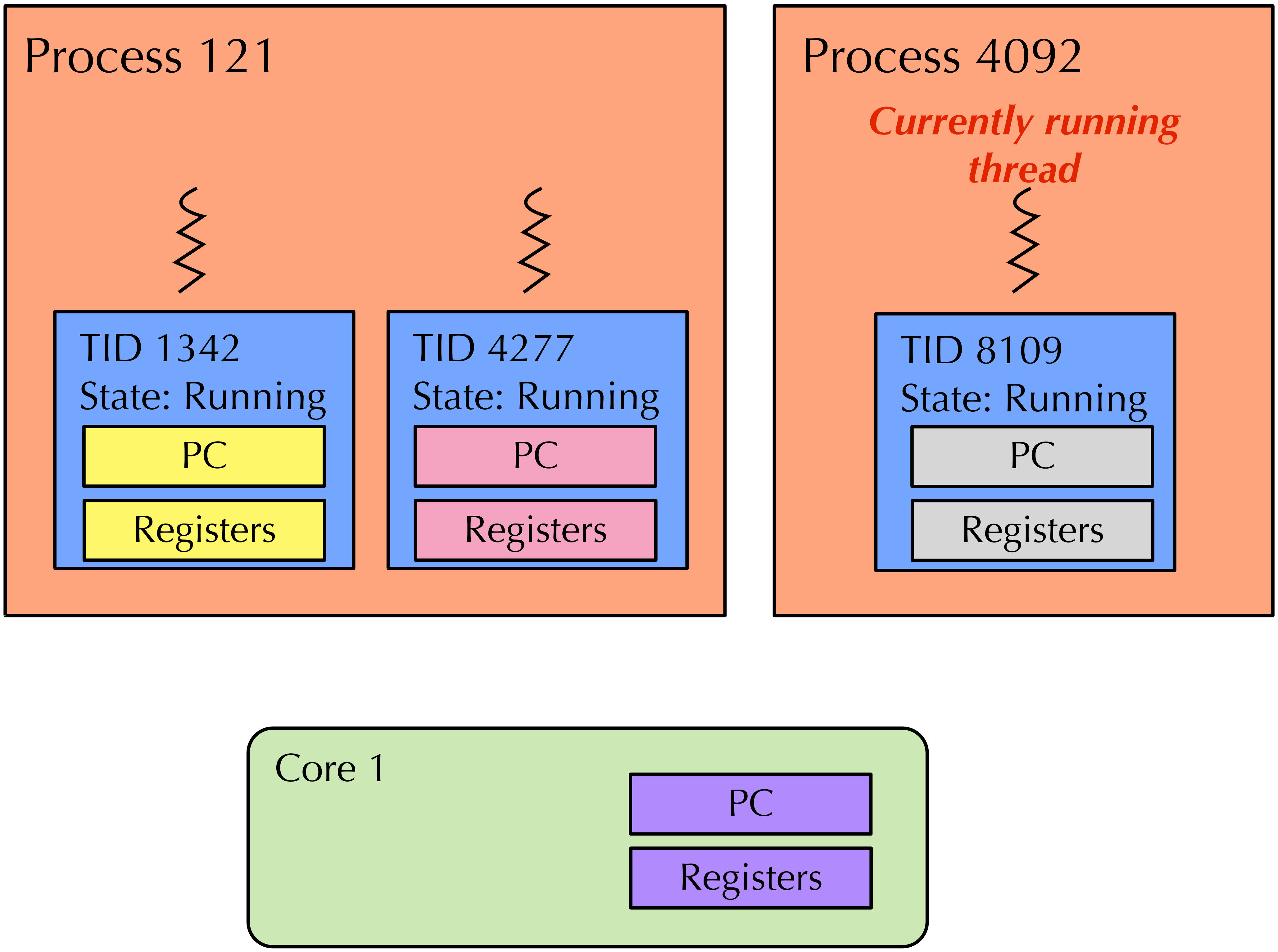
# Threads, scheduling, and cores

- The thread is now the unit of CPU scheduling
  - A process is just a “container” for its threads
- Can timeshare the CPU, execute threads concurrently
- Most modern CPUs have multiple cores.
  - Different threads can run on different cores
- Single cores can run multiple threads from same process!
  - Called **hyperthreading** by Intel
  - Example: During a cache miss, can run another part of the program on otherwise “idle” portions of the core.

# Context Switching



# Context Switching



# Inter- vs. intra-process context switching

- OS can switch between two threads in the same process, or two threads in different processes.
  - Which is faster?
- Switching across processes:
  - The new thread is in a different address space!
  - So, need to update the MMU state to use the new page tables
  - Also need to flush the TLB .... why?
  - When the new thread starts running, it will suffer both TLB and cache misses.
- Switching within the same process:
  - Only need to save CPU registers and PC into the TCB, and restore them.
  - This can be pretty fast ... tens of instructions.

# When to switch between threads?

- Preemptive scheduling
  - Scheduler decides when it is time to switch
  - Forcibly removes thread from the CPU, and switches to another
- Cooperative scheduling
  - Aka non-preemptive scheduling
  - Threads must explicitly **yield** control
    - By calling a special function
  - +ve: makes it easier to reason about concurrent code
    - (At least, in single threaded machines)
  - -ve: one thread may hog resources

- Modern OSes are typically preemptive

# Topics for today

- Threads: Allowing a single program to do multiple things concurrently.
- Implementing
- Scheduling
- Programming with threads (pthreads library)
- Shared vs. private resources
- The need for synchronization

# Programming with threads

- Standard API called POSIX threads
  - AKA Pthreads
  - POSIX (=Portable Operating System Interface for uniX) is a suite of IEEE standards
  - Large library: ~100 functions for:
    - creating, killing, reaping threads,
    - synchronizing threads
    - safely communicating between threads

# Programming with threads

- ```
  int pthread_create(pthread_t  *tid,
                     pthread_attr_t *attr,
                     void *(*start_routine)(void *),
                     void *arg);
  ```
  - `tid`: Returns thread ID of newly created thread
  - `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()`
- `pthread_t pthread_self(void);`
  - Returns thread ID of current thread
- Thread IDs (values of type `pthread_t`) are unique within a process
  ```

# Terminating threads

- Threads terminate **implicitly** when top-level thread routine terminates
  - i.e., `main` routine for the main thread, or the start routine for a peer thread
- `void pthread_exit(void *retval);`
  - Explicitly terminates current thread, with thread return value of `retval`
  - If current thread is main thread, will wait for all other threads to terminate, and exit process with return value of `retval`
- `int pthread_cancel(pthread_t tid);`
  - Terminate thread `tid`
- If `exit` function is called, process terminates as do all threads in process

# pthreads example

```
#include <pthread.h>

volatile int myvar = 0;

void *run_thread1(void *arg) {
  while (1) {
    myvar++;
  }
}

void *run_thread2(void *arg) {
  while (1) {
    printf("Hello from thread2, myvar is %d.\n", myvar);
    sleep(1);
  }
}
```

myvar is global!

```
Hello from thread2, myvar is 10280.
Hello from thread2, myvar is 303978686.
Hello from thread2, myvar is 609594391.
Hello from thread2, myvar is 913397409.
Hello from thread2, myvar is 1220379635.
Hello from thread2, myvar is 1527953404.
...
```

```
int main(int argc, char **argv) {
  pthread_t thread1, thread2;
  pthread_create(&thread1, NULL, run_thread1, NULL);
  pthread_create(&thread2, NULL, run_thread2, NULL);
  pthread_exit(NULL);
}
```

# pthreads example

What happens if we get rid of volatile here?

```
#include <pthread.h>

volatile int myvar = 0;

void *run_thread1(void *arg) {
  while (1) {
    myvar++;
  }
}

void *run_thread2(void *arg) {
  while (1) {
    printf("Hello from thread2, myvar is %d.\n", myvar);
    sleep(1);
  }
}
```

```
int main(int argc, char **argv) {
  pthread_t thread1, thread2;
  pthread_create(&thread1, NULL, run_thread1, NULL);
  pthread_create(&thread2, NULL, run_thread2, NULL);
  pthread_exit(NULL);
}
```

# pthreads example

What happens if we get rid of volatile here?

```
#include <pthread.h>

int myvar = 0;

void *run_thread1(void *arg) {
  while (1) {
    myvar++;
  }
}

void *run_thread2(void *arg) {
  while (1) {
    printf("Hello from thread2, myvar is %d.\n", myvar);
    sleep(1);
  }
}
```

```
Hello from thread2, myvar is 0.
Hello from thread2, myvar is 0.
Hello from thread2, myvar is 0.
Hello from thread2, myvar is 0.
Hello from thread2, myvar is 0.
Hello from thread2, myvar is 0.
...
```

```
int main(int argc, char **argv) {
  pthread_t thread1, thread2;
  pthread_create(&thread1, NULL, run_thread1, NULL);
  pthread_create(&thread2, NULL, run_thread2, NULL);
  pthread_exit(NULL);
}
```

# pthreads example

- What's going on here?
- `volatile` keyword tells the compiler that `myvar` might change in between two subsequent reads of the variable.
  - For example, because another thread modified it!
- In general, should declare shared variables `volatile` if you want to ensure the compiler won't optimize away memory reads and writes.

# pthreads example

```
#include <pthread.h>

volatile int myvar = 0;

void *run_thread1(void *arg) {
  while (1) {
    myvar++;
    printf("Hello from thread1, myvar is %d.\n", myvar);
    sleep(1);
  }
}

void *run_thread2(void *arg) {
  while (1) {
    myvar *= 2;
    printf("Hello from thread2, myvar is %d.\n", myvar);
    sleep(1);
  }
}
```

Both threads are now writing to `myvar`

```
int main(int argc, char **argv) {
  pthread_t thread1, thread2;
  pthread_create(&thread1, NULL, run_thread1, NULL);
  pthread_create(&thread2, NULL, run_thread2, NULL);
  pthread_exit(NULL);
}
```

# pthreads example

```
#include <pthread.h>

volatile int myvar = 0;

void *run_thread1(void *arg) {
  while (1) {
    myvar++;
    printf("Hello from thread1, myvar is %d.\
    sleep(1);
  }
}

void *run_thread2(void *arg) {
  while (1) {
    myvar *= 2;
    printf("Hello from thread2, myvar is %d.\n", m
    sleep(1);
  }
}
```

```
int main(int argc, char **argv) {
  pthread_t thread1, thread2;
  pthread_create(&thread1, NULL, run_thread1, NULL);
  pthread_create(&thread2, NULL, run_thread2, NULL);
  pthread_exit(NULL);
}
```

No guarantee of the order in which threads run.

```
...
Hello from thread2, myvar is 94.
Hello from thread1, myvar is 95.
Hello from thread2, myvar is 190.
Hello from thread2, myvar is 380.
Hello from thread1, myvar is 381.
Hello from thread1, myvar is 763.
Hello from thread2, myvar is 762.
Hello from thread2, my    is 1526.

...
```

Why is this out of order?

# pthreads example

No guarantee of the order in which threads run.

```
#include <pthread.h>

volatile int myvar = 0;

void *run_thread1(void *arg) {
  while (1) {
    myvar++;
    printf("Hello from thread1, myvar is %d.\
    sleep(1);
  }
}

void *run_thread2(void *arg) {
  while (1) {
    myvar *= 2;
    printf("Hello from thread2, myvar is %d.\n", myvar);
    sleep(1);
  }
}
```

```
int main(int argc, char **argv) {
  pthread_t thread1, thread2;
  pthread_create(&thread1, NULL, run_thread1, NULL
  pthread_create(&thread2, NULL, run_thread2, NULL
  pthread_exit(NULL);
}
```

```
...
Hello from thread2, myvar is 94.
Hello from thread1, myvar is 95.
Hello from thread2, myvar is 190.
Hello from thread2, myvar is 380.
Hello from thread1, myvar is 381.
Hello from thread1, myvar is 763.
Hello from thread2, myvar is 762.
Hello from thread2, myvar is 1526.

...
```

thread2 called printf("762") but OS context switched to thread1 before it got a chance to write to the screen!

# Reaping threads

- `int pthread_join(pthread_t tid,`
  `                  void **thread_return);`
  - Waits for thread `tid` to exit, places return value of the thread in location pointed to by `thread_return`
  - Reaps any memory resources held by terminated thread
  - Can only wait for a specific thread

# Joinable and detached threads

- Threads are either **joinable** or **detached**

- A **joinable thread** can be killed and reaped by other threads
  - Memory resources not recovered until it is reaped by another thread
- **Detached thread** cannot be killed or reaped by another thread
  - Memory resources recovered when detached thread terminates

- By default, threads are joinable
- `int pthread_detach(pthread_t tid);`
  - Detaches thread tid `tid`
- Why have detached threads?

# Topics for today

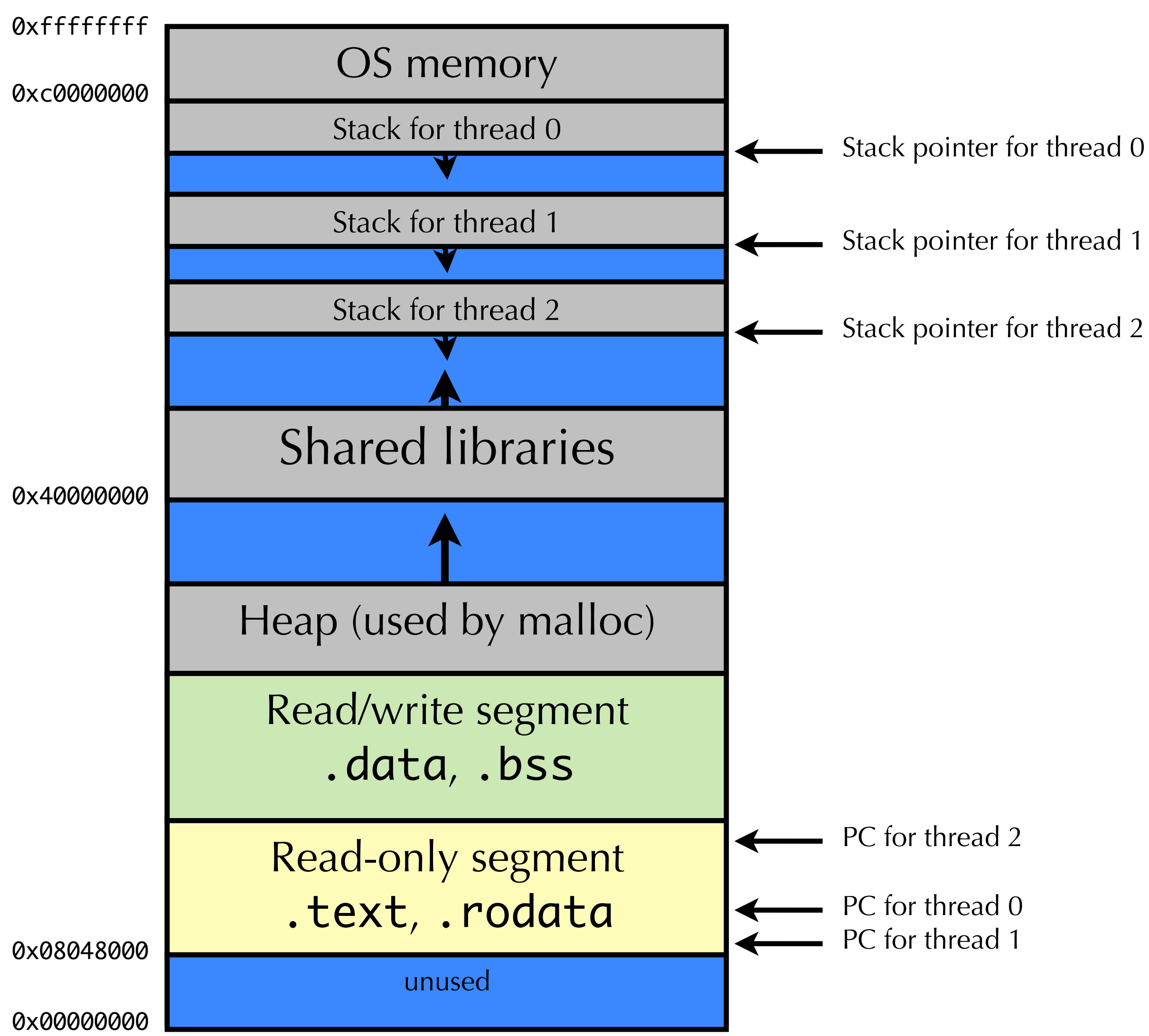
- Threads: Allowing a single program to do multiple things concurrently.
- Implementing
- Scheduling
- Programming with threads (pthreads library)
- Shared vs. private resources
- The need for synchronization

# Local and global variables

- Threads in same process share address space
- So which locations are shared between threads?
- Suppose thread1 and thread2 both run `foo` at the same time.

```
void foo() {
  int i=0;
  i++;
  sleep(1);
  printf("i is %d.\n", i);
}
```

  - Both output 1
- Local variables are not (usually) shared
  - Either local variable is stored in register
    - Part of context of thread
  - Or stored on stack
    - Each thread has its own stack

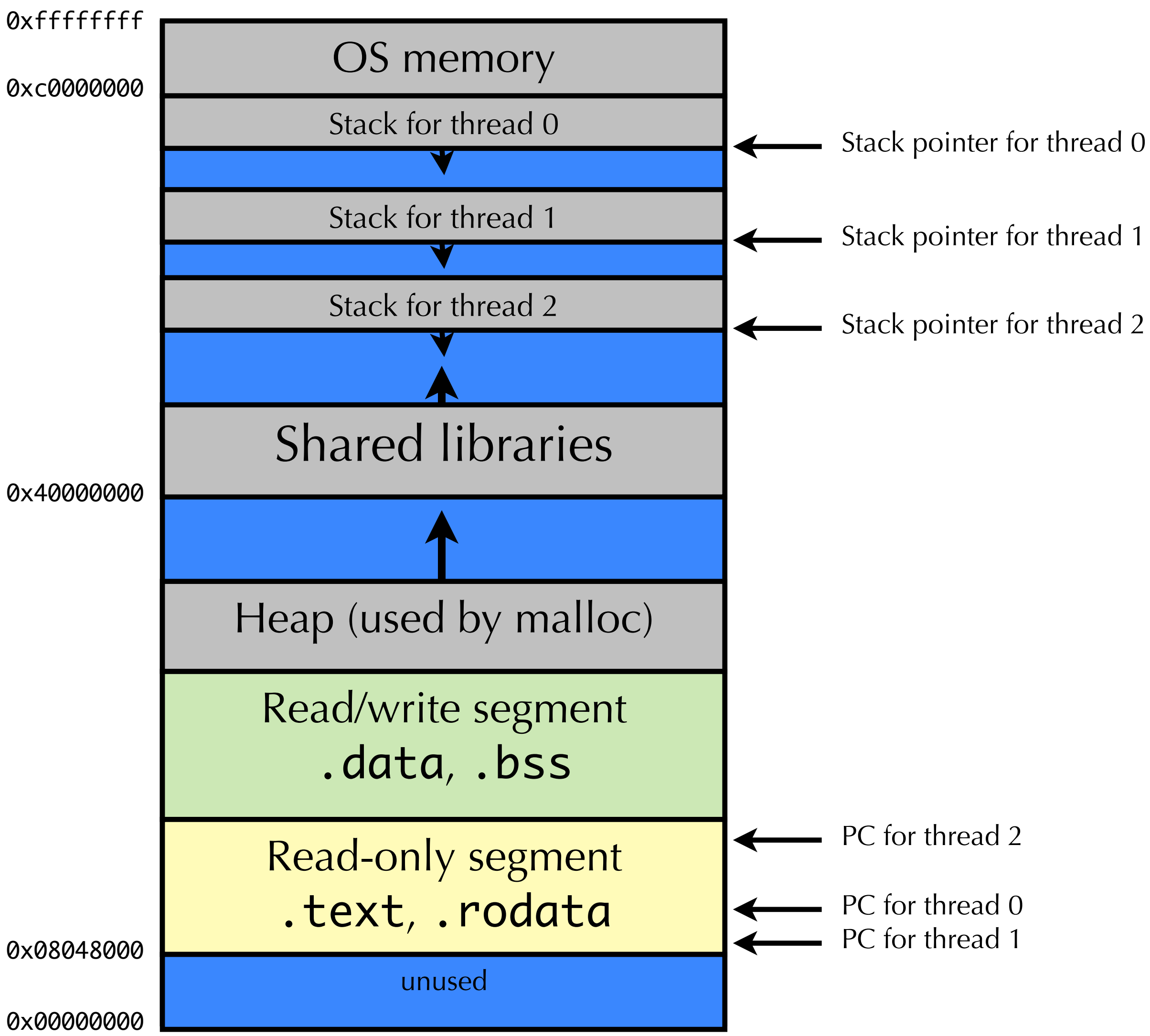


# Local and global variables

- Threads in same process share address space
- So which locations are shared between threads?
- Suppose thread1 and thread2 both run `bar` at the same time.

```
int i = 0; // global variable
void bar() {
  i++;
  sleep(1);
  printf("i is %d.\n", i);
}
```

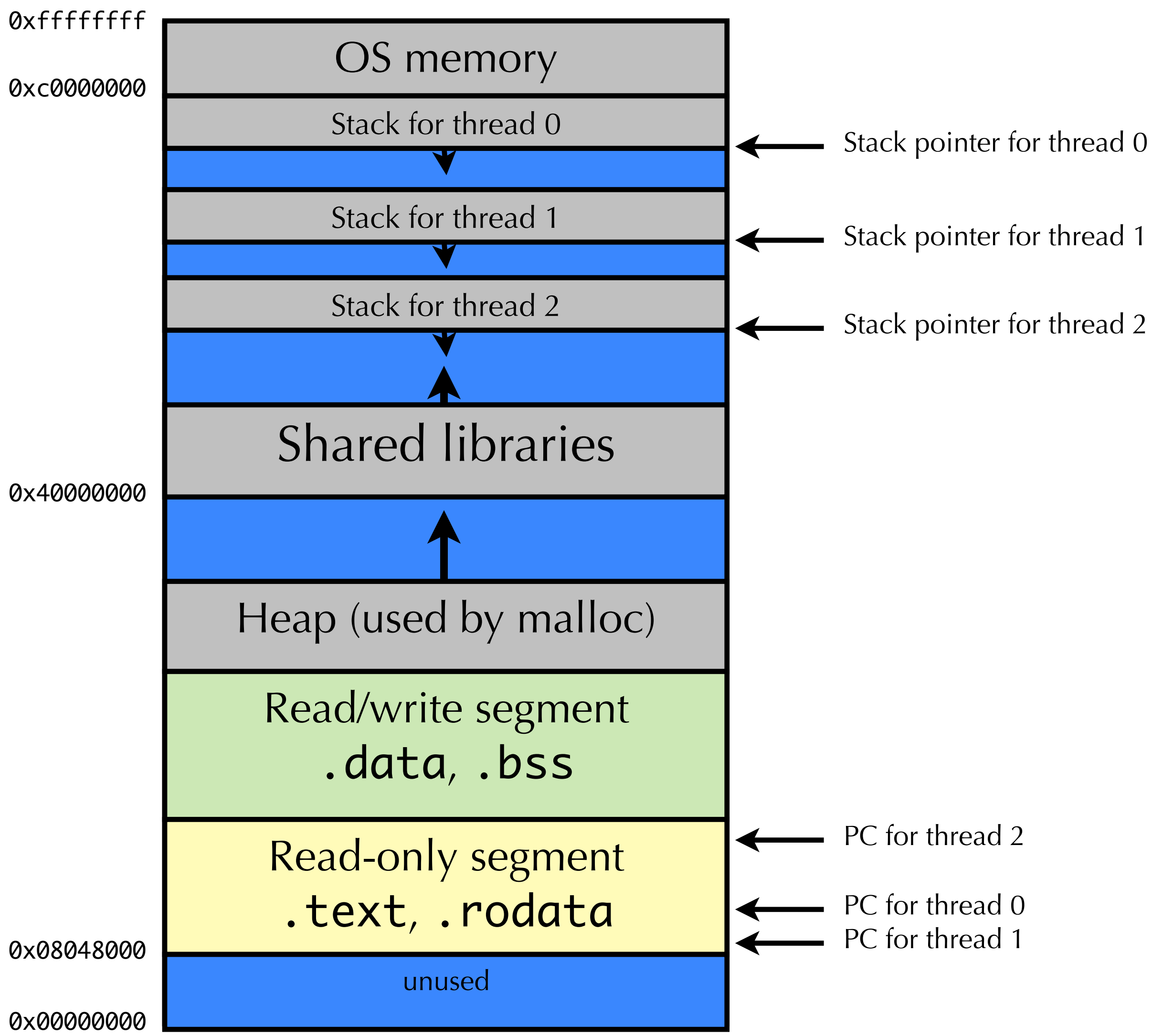
  - Both output 2
- Global variables are shared



# Local and global variables

- Threads in same process share address space
- So which locations are shared between threads?
- Suppose thread1 and thread2 both run **baz** at the same time.

```
void baz() {
  static int i = 0;
  i++;
  sleep(1);
  printf("i is %d.\n", i);
}
```




  - Both output 2
- Local static variables are shared

# Topics for today

- Threads: Allowing a single program to do multiple things concurrently.
- Implementing
- Scheduling
- Programming with threads (pthreads library)
- Shared vs. private resources
- The need for synchronization

# Synchronization

- Threads cooperate in multithreaded programs in several ways:
  - Access to shared variables and other memory
    - e.g., multiple threads accessing a memory cache in a Web server
  - To coordinate their execution
    - e.g., Pressing stop button on browser cancels download of current page
    - "stop button thread" has to signal the "download thread"
- For correctness, we have to control this cooperation
- We must assume that **threads can interleave executions arbitrarily** and **run at different rates**
  - In some sense this is the "worst case" scenario.
  - Our goal: to control thread cooperation using synchronization
    - enables us to restrict the interleaving of executions

# Shared Resources

- We'll focus on coordinating access to shared resources
- Basic problem:
  - Two concurrent threads are accessing a shared variable
  - If the variable is read/modified/written by both threads, then access to the variable must be controlled
  - **Otherwise, unexpected results may occur**
- Tools for solutions
  - Mechanisms to control access to shared resources
    - Low-level mechanisms: locks
    - Higher level mechanisms: mutexes, semaphores, monitors, and condition variables
  - Patterns for coordinating access to shared resources
    - bounded buffer, producer-consumer, …
- This stuff is complicated and rife with pitfalls

# Shared Variable Example

- Suppose we implement a function to withdraw money from a bank account:

```
int withdraw(account, amount) {
  balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

- Now suppose that you and your roommate share a bank account with a balance of $1500.00 (not necessarily a good idea...)
  - What happens if you both go to separate ATM machines, and simultaneously withdraw $100.00 from the account?

# Example continued

- We represent the situation by creating a separate thread for each ATM user doing a withdrawal
  - Both threads run on the same bank server system

```
int withdraw(account, amount) {
  balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

```
int withdraw(account, amount) {
  balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

- What are the possible balance values after each thread runs?

# Interleaved Execution

- The execution of the two threads can be **interleaved**
  - Assume **preemptive scheduling**
    - i.e., Thread may be context switched arbitrarily, without cooperation from the thread
  - Each thread may context switch after **each** assembly instruction (or, in some cases, part of an assembly instruction!)
  - We need to worry about the worst-case scenario!

*Execution sequence as seen by CPU*

```
balance = get_balance(account);
balance -= amount;
```

*context switch*

```
balance = get_balance(account);
balance -= amount;
put_balance(account, balance);
```

*context switch*

```
put_balance(account, balance);
```

- What's the account balance after this sequence?
  - And who's happier, the bank or you???

# Interleaved Execution

- The execution of the two threads can be **interleaved**
  - Assume **preemptive scheduling**
    - i.e., Thread may be context switched arbitrarily, without cooperation from the thread
  - Each thread may context switch after **each** assembly instruction (or, in some cases, part of an assembly instruction!)
  - We need to worry about the worst-case scenario!

*Execution sequence as seen by CPU*

*account.bal = $1500*

```
balance = get_balance(account);
balance -= amount;
```
*local balance = $1400*

```
balance = get_balance(account);
balance -= amount;
```
*local balance = $1400*

```
put_balance(account, balance);
```
*account.bal = $1400*

```
put_balance(account, balance);
```
*account.bal = $1400*

- What's the account balance after this sequence?
  - And who's happier, the bank or you???

# Little white lie...

- Sleeping does not help!
- Earlier I showed some examples to highlight which locations were shared between threads

```
int i = 0; // global variable
void bar() {
  i++;
  sleep(1);
  printf("i is %d.\n", i);
}
```

```
int i = 0; // global variable
void bar() {
  i++;
  sleep(1);
  printf("i is %d.\n", i);
}
```

- Possible outputs: 1 2, 1 2, 2 2, 2 2
- All are possible, not all equally likely.

# It's gets worse...

- Most programmers assume that memory is **sequentially consistent**
  - state of memory is due to some interleaving of threads, with instructions in each thread executed in order
  - E.g., Given A B C and D E, memory is result of some ordering such as

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| A | A | A | A | A | D | D | D | D |
| B | B | B | D | D | A | A | E | A |
| C | D | D | B | B | B | B | A | E |
| D | C | E | C | E | C | E | B | B |
| E | E | C | E | C | E | C | C | C |

… etc.

- **This is not true in most systems!**

# Example

- Suppose we have two threads
  - (`x` and `y` are global, `a` and `b` are thread-local, all variables initially 0)

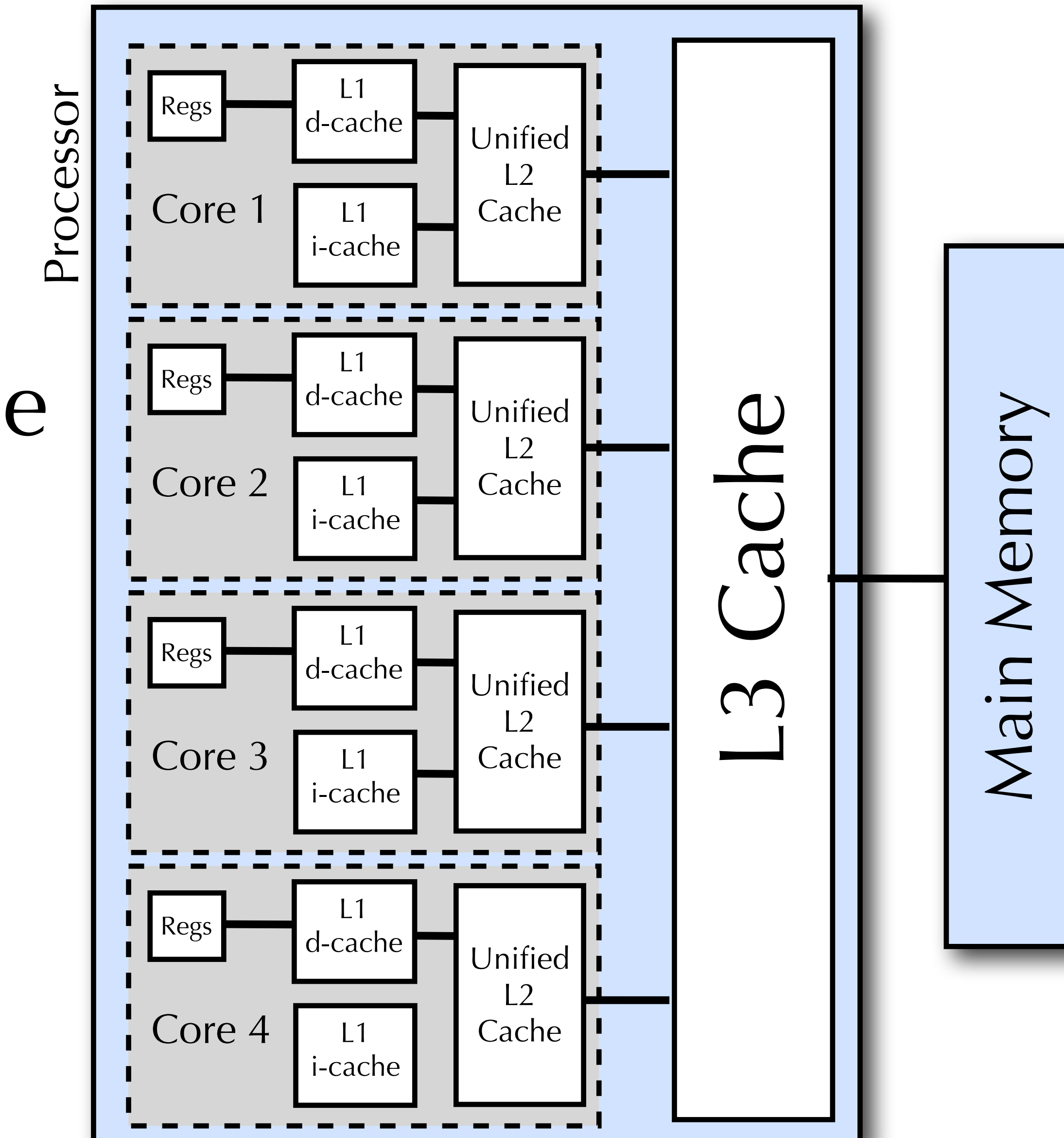
```
x=1;
y=2;
```

```
a = y;
b = x;
printf("%d", a+b);
```

  - What are the possible outputs?
    - 0 `a=y` `b=x` `x=1` `y=2`
    - 1 `a=y` `x=1` `b=x` `y=2` and others
    - 3 `x=1` `y=2` `a=y` `b=x` and others
    - 2 Requires `a=2` and `b=0`. Is possible, but no such order!

# What the ...?

- What's going on?
- Several things, including:
  - With multiple processors, multiple caches
    - A cache may not write values from cache to memory in same order as updates
    - Processor may have cache hits for some locations and not others
  - Compiler optimizations
    - Compiler may change order of instructions



# Relaxed memory models

- A **model** of how memory behaves provides
  - 1) programmers with a way to think about memory
  - 2) compiler writers with limits on what optimizations they can do
  - 3) hardware designers with limits on what optimizations they can do
- Relaxed memory models provide a weaker model than sequential consistency
  - Can be complicated!

# Race Conditions

- The problem: concurrent threads accessing a shared resource without any synchronization
  - This is called a **race condition**
  - The result of the concurrent access is non-deterministic, depends on
    - Timing
    - When context switches occurred
    - Which thread ran at which context switch
    - What the threads were doing
- A solution: mechanisms for controlling concurrent access to shared resources
  - Allows us to reason about the operation of programs
  - We want to **re-introduce some determinism** into the execution of multiple threads

# Race conditions in real life

- Race conditions are bugs, and difficult to detect
- Northeast Blackout of 2003
  - About 55 million people in North America affected
  - Race condition in monitoring code in part responsible: alarm system failed
  - Code had been running since 1990, over 3 million hours of operation, without manifesting bug


Vincent Laforet

# Race conditions in real life

- Race conditions are bugs, and difficult to detect
- Therac-25 radiation therapy machine
  - Designed to give non-lethal doses of radiation to cancer patients
  - Race conditions contributed to incorrect lethal doses
  - Several fatalities in mid-80s.

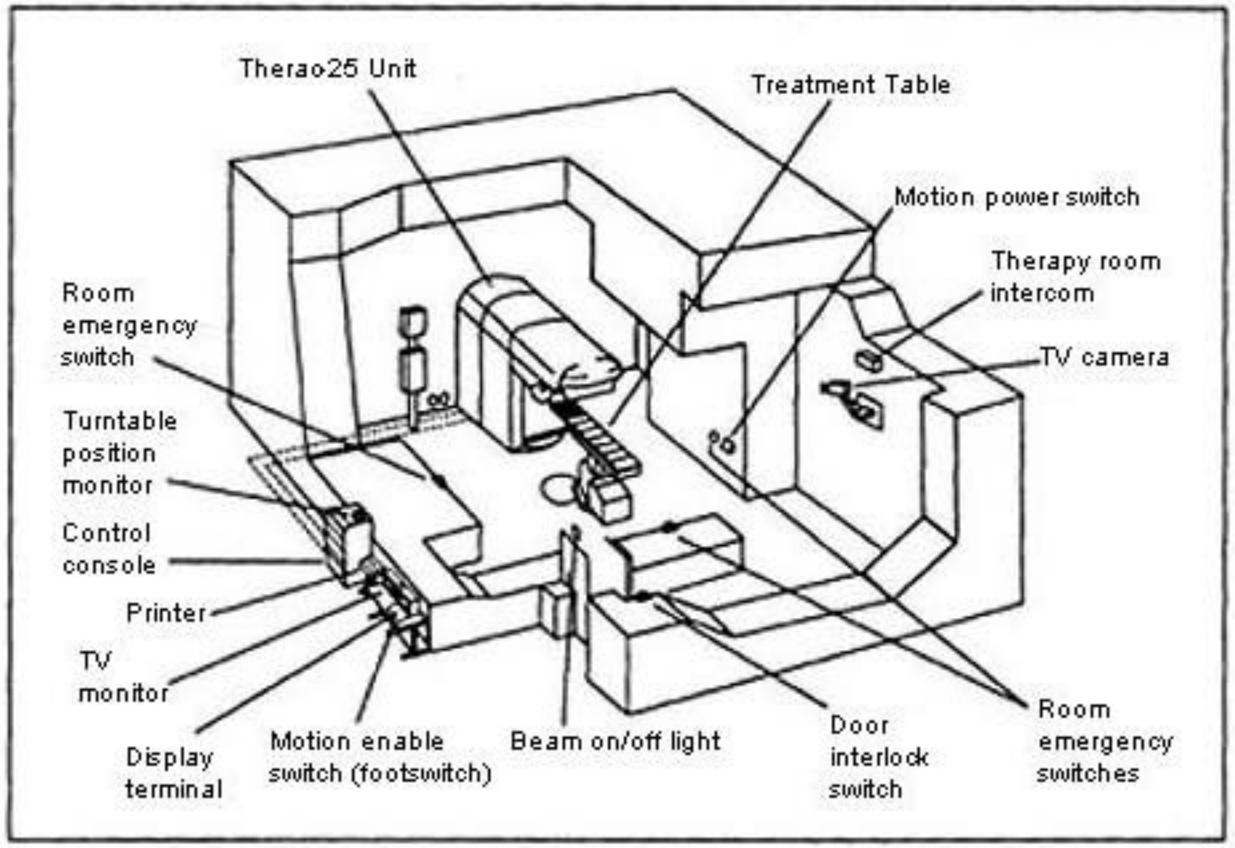



Figure 1. Typical Therac-25 facility

# Next Lecture

- 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?