HARVARD
School of Engineering
and Applied Sciences

Threads

CS61, Lecture 19

Prof. Stephen Chong

November 4, 2010

Announcements

- CS colloquium at 4pm, MD G-125
  - *Jumble and FastTrack: Efficient and Precise Dynamic Detection of Destructive Races*
    Steve Freund, Williams College.

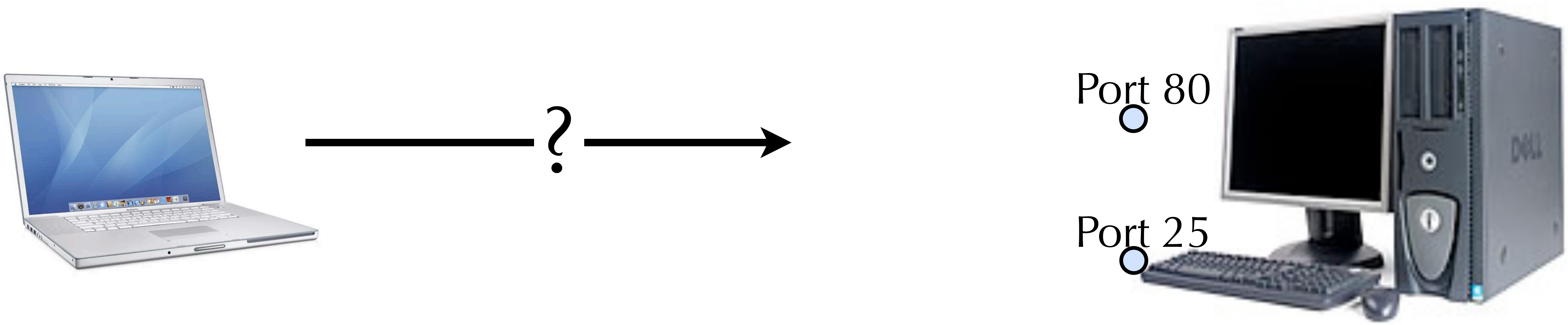
  - Very relevant to the next 3 or 4 lectures! I encourage you to attend

HARVARD
School of Engineering and Applied Sciences

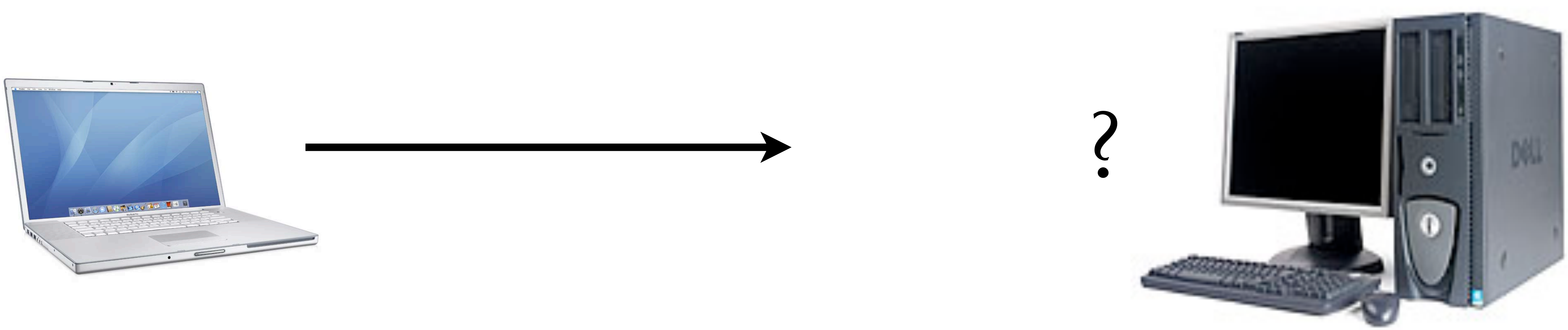
Network programming with sockets (ctd)

Programming with sockets

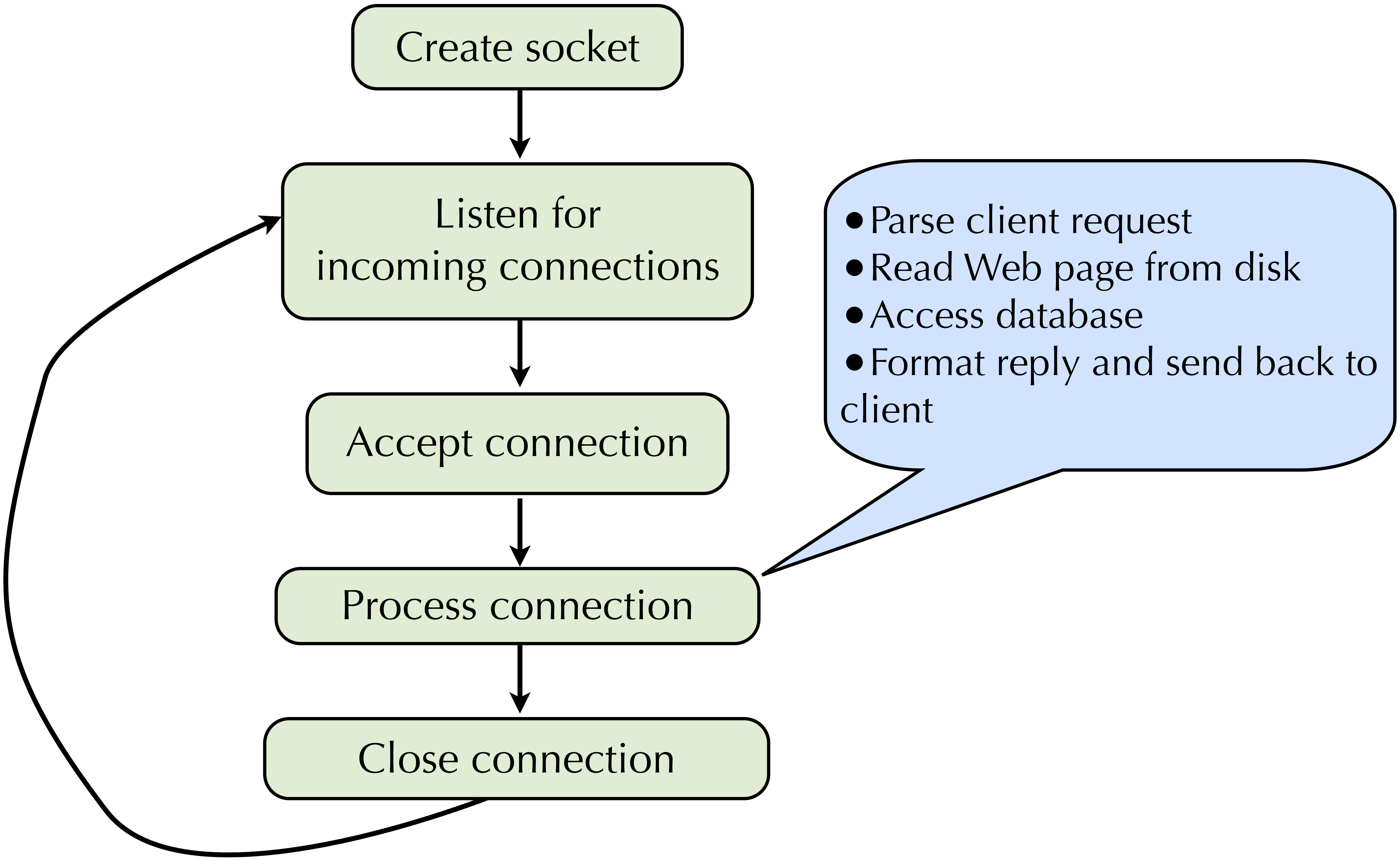
- Last lecture we saw how a client connects to a server



- How does a server listen for and accept connections from clients?



Lifecycle of a server



Step 1: Creating socket

```
int listenfd;    /* Socket to listen on */
int optval = 1; /* Used by setsockopt() below */

if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
  perror("Cannot create socket");
  exit(1);
}

/* Allow this socket to reuse a port number */
if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
     (const void*)&optval, sizeof(int)) < 0) {
  perror("Cannot set SO_REUSEADDR socket option");
  exit(1);
}
```

- Create a socket: `socket()` system call
- Use `setsockopt()` to permit socket to reuse the server port number
  - Otherwise if you restart the server, the kernel thinks the port number is already in use by the (now dead) server process.
  - Eventually the OS would allow the port to be reused, but only after a long timeout.
  - This has to do with details of the TCP protocol.

Step 2: Binding and listening

```
struct sockaddr_in server_addr;

/* Zero out the server address */
bzero(&server_addr, sizeof(server_addr));
server_addr.sin_family = AF_INET;
/* Listen for connections from any client on the Internet. */
server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
/* Listen for connections on port 80 */
server_addr.sin_port = htons(80);
```

- First, prepare the server address
  - Generally set address to INADDR_ANY to indicate you will accept connections from any IP address on the Internet.

Step 2: Binding and listening

```
/* Bind socket to address */
if (bind(listenfd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
  perror("Cannot bind socket");
  exit(1);
}

/* Listen for incoming connections. Use listen queue length of 10. */
if (listen(listenfd, 10) < 0) {
  perror("Cannot listen");
  exit(1);
}
```

- Then `bind()` the socket to the port you want to listen on.
- Then `listen()` for incoming connections.

Step 3: Accept connection

```
int clifd;
struct sockaddr_in client_addr;
int client_addr_len = sizeof(client_addr);

while (1) {
  clifd = accept(listenfd,
                 (struct sockaddr *)&client_addr,
                 (socklen_t*)&client_addr_len);
  /* Process connection here */
  do_something(clifd);

  close(clifd);
}
```

- The server spins in a loop doing the following:
  - Call `accept()` to accept an incoming connection from the Internet
    - This blocks until a connection is received!
  - Process the connection
  - `close()` the client socket
- `accept()` returns new file descriptor of socket for the new connection.

Find out the client's hostname

- `accept()` returns the client's IP address and port number.
- Can use `gethostbyaddr()` to look up the hostname.
  - Note that not all IP addresses have a corresponding hostname.

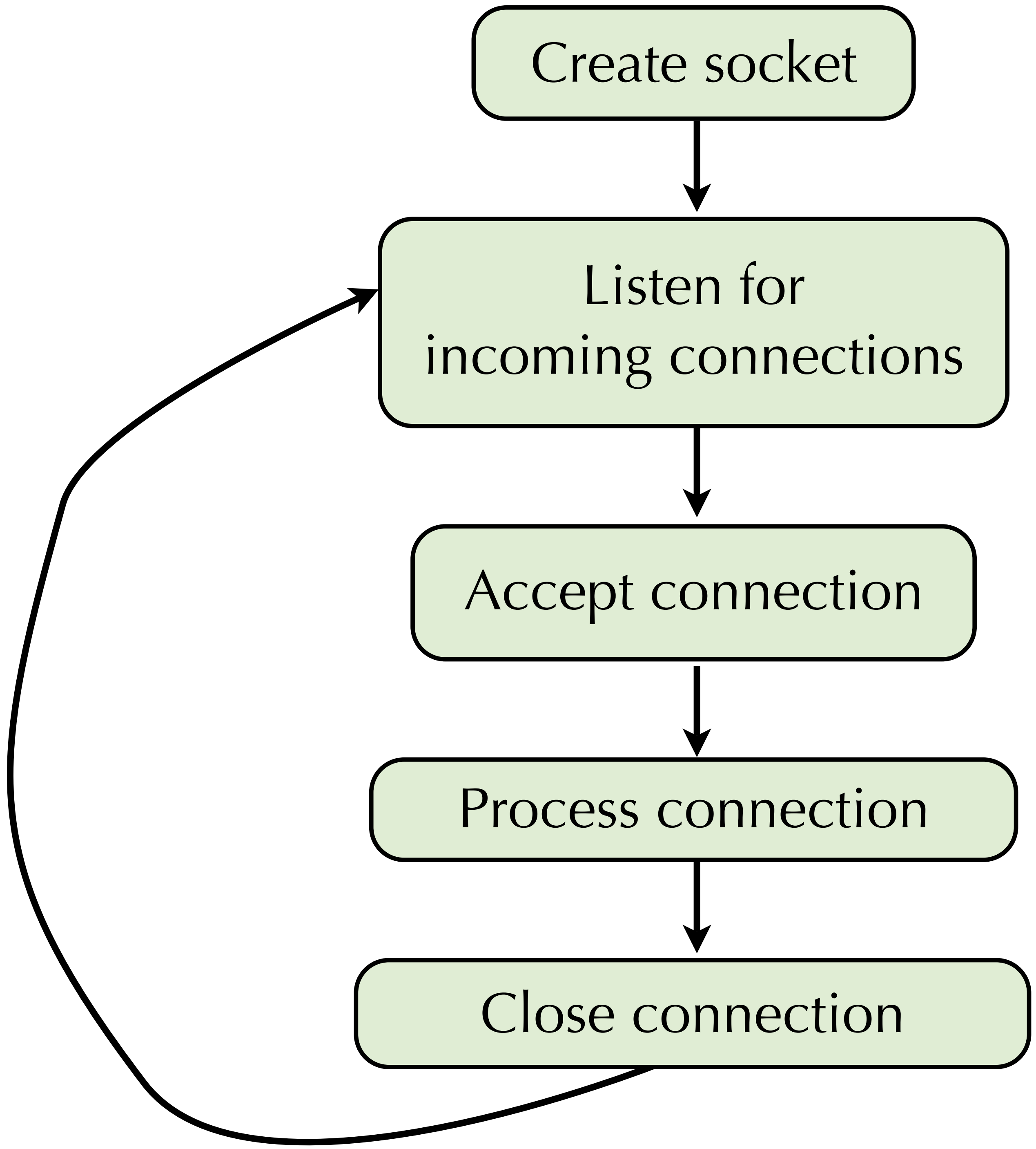
```
struct hostent *hp;
char *hostname, *hostip;

while (1) {
  clifd = accept(listenfd,
                 (struct sockaddr *)&client_addr,
                 (socklen_t*)&client_addr_len);

  hp = gethostbyaddr((const char *)&client_addr.sin_addr.s_addr,
                     sizeof(client_addr.sin_addr.s_addr), AF_INET);
  if (hp != NULL) {
    hostname = hp->h_name;
  } else {
    hostname = "unknown hostname";
  }
  hostip = inet_ntoa(client_addr.sin_addr);
  fprintf(stderr, "Got connection from %s (%s)\n", hostname, hostip);

  /* Do stuff... */
```

What's wrong with this design?

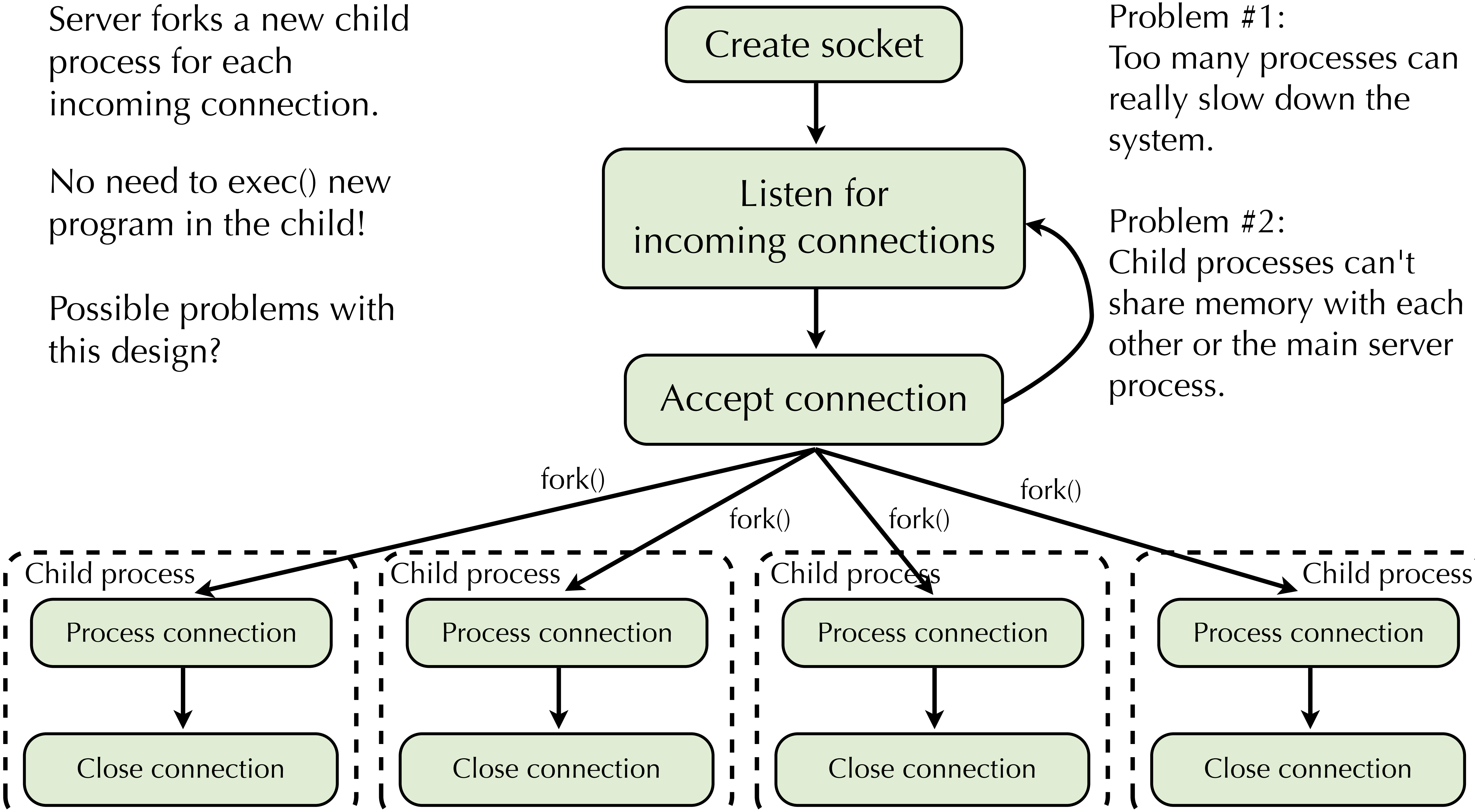


Multiprocess server

Server forks a new child process for each incoming connection.

No need to exec() new program in the child!

Possible problems with this design?



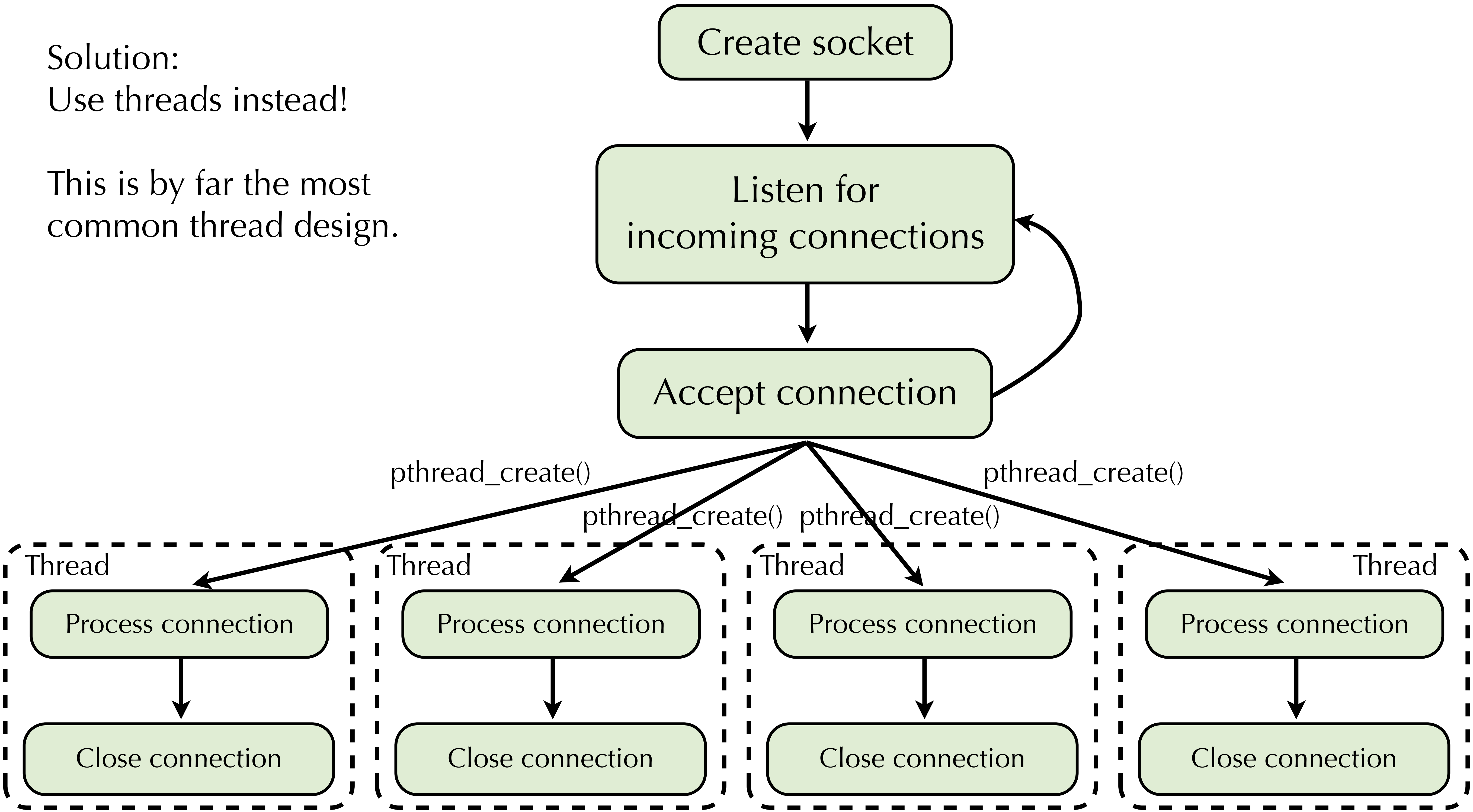
Problem #1:
Too many processes can really slow down the system.

Problem #2:
Child processes can't share memory with each other or the main server process.

Multithreaded server

Solution:
Use threads instead!

This is by far the most common thread design.



HARVARD
School of Engineering
and Applied Sciences

Threads

CS61, Lecture 19

Prof. Stephen Chong

November 4, 2010

Topics for today

- Threads: Allowing a single program to do multiple things concurrently.
- Implementing
- Scheduling
- Private vs. shared memory
- Programming with threads (pthreads library)

- Reading: 12.1, 12.3

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?

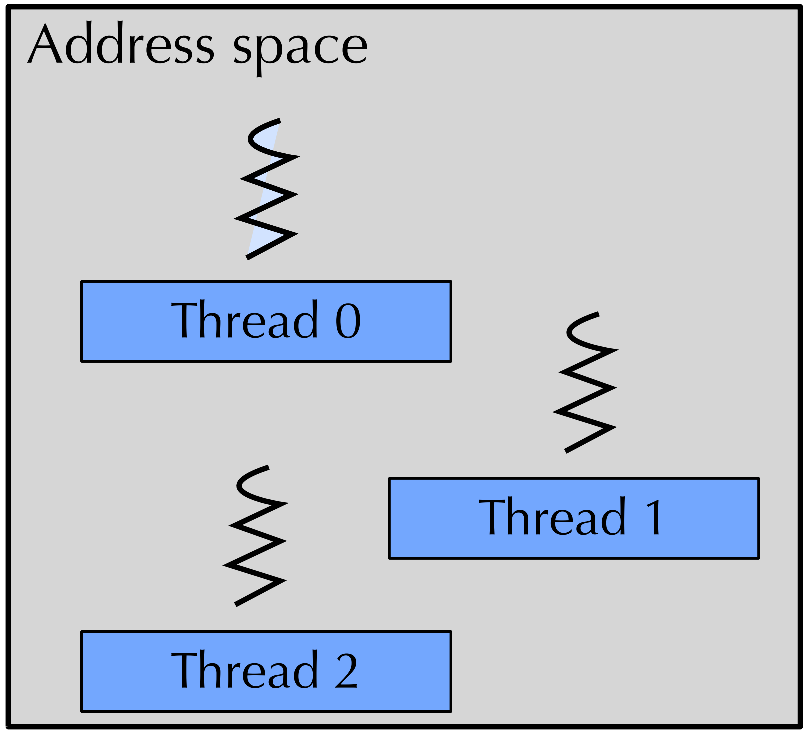
- What is private to each task?

Can we do better?

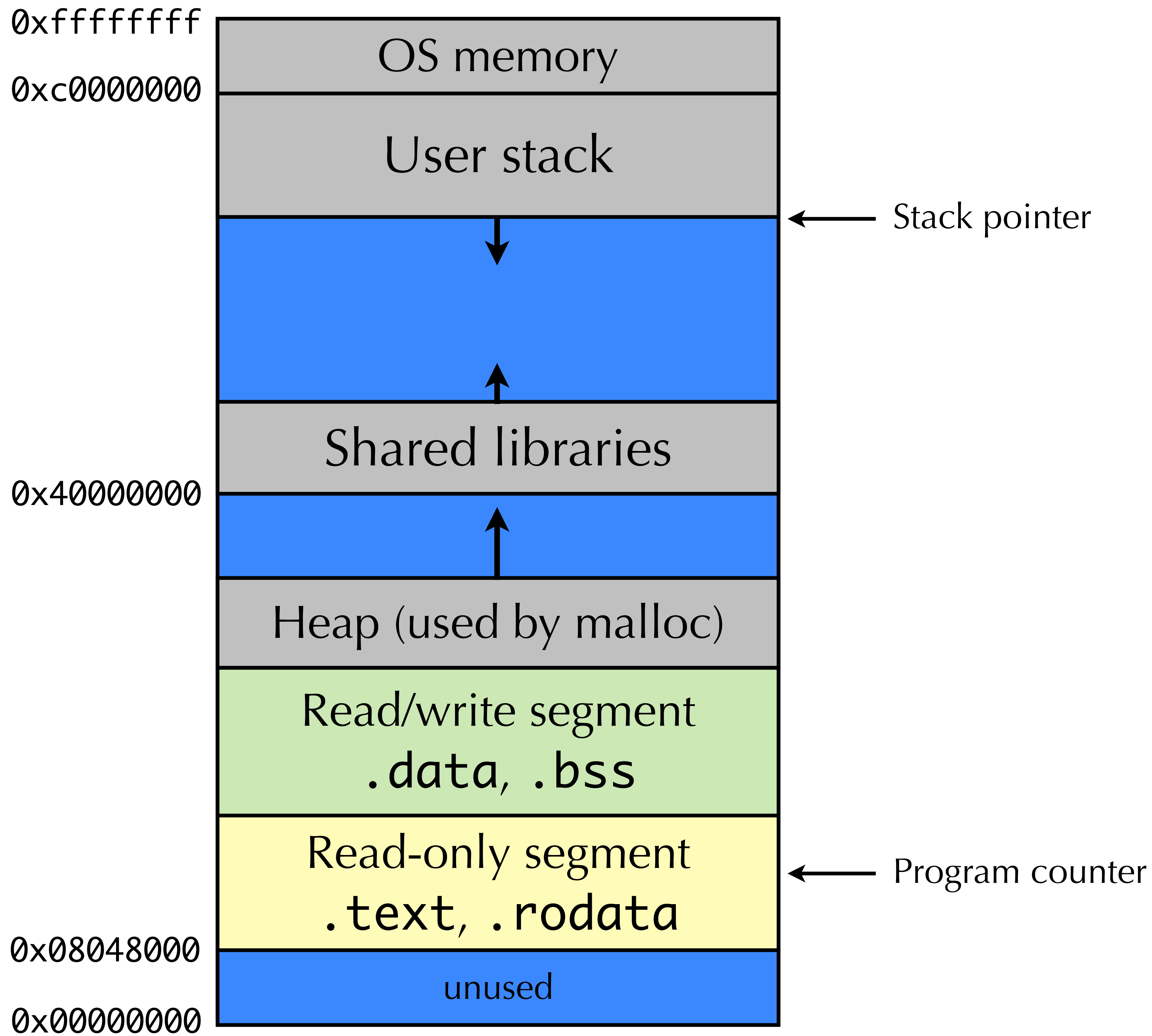
- What can we share across all of these tasks?
  - Same code – generally running the same or similar programs
  - Same data
  - Same privileges
  - Same OS resources (files, sockets, etc.)
- What is private to each task?
  - Execution state: CPU registers, stack, and program counter
- Key idea:
  - Separate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

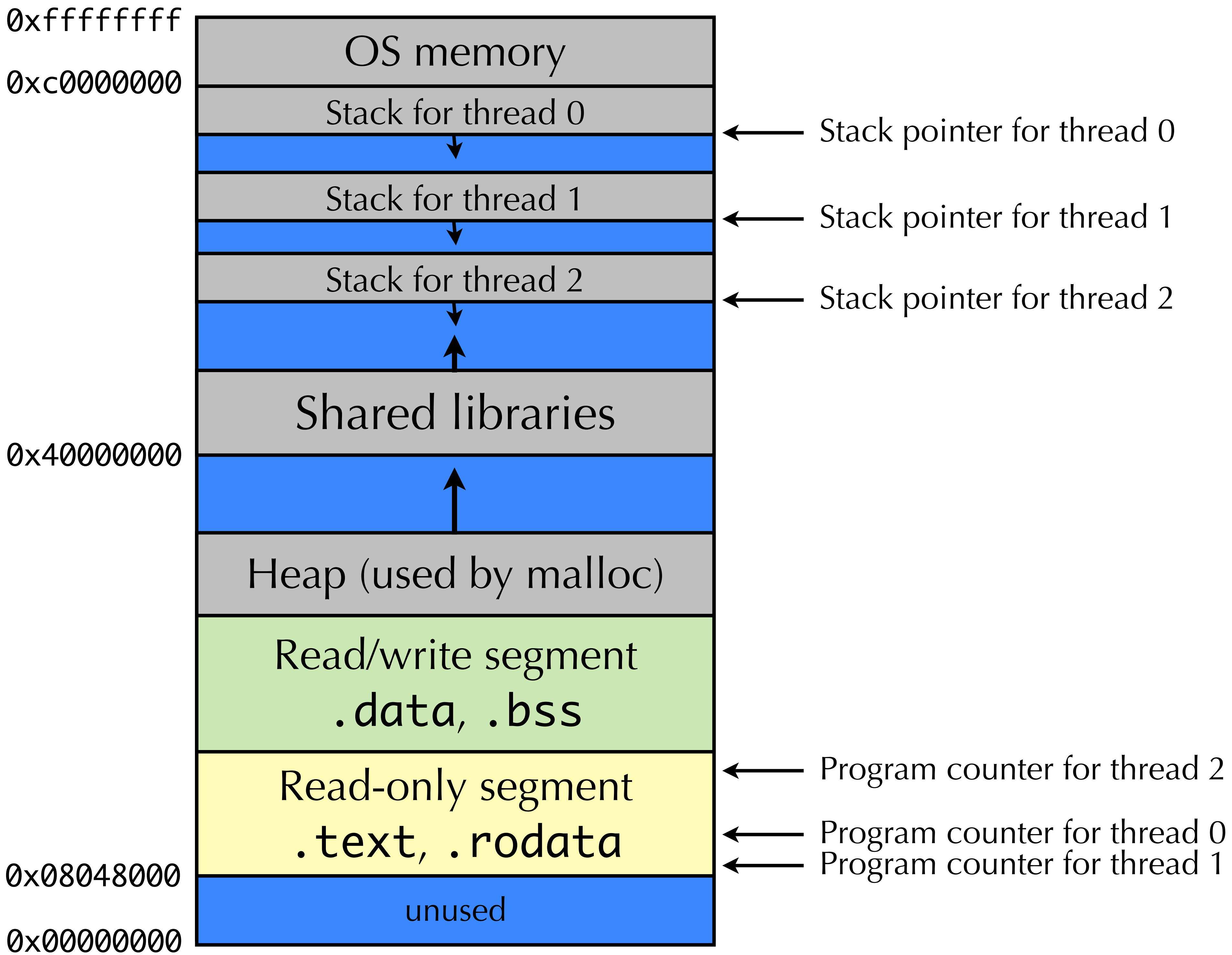
- A **thread** is a logical flow of control that runs in the context of a process
- Each process has one or more threads “within” it
  - Each process begins with a single **main** thread
  - Threads can create new threads, called **peer** threads
- Each thread has its own stack, stack pointer, program 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



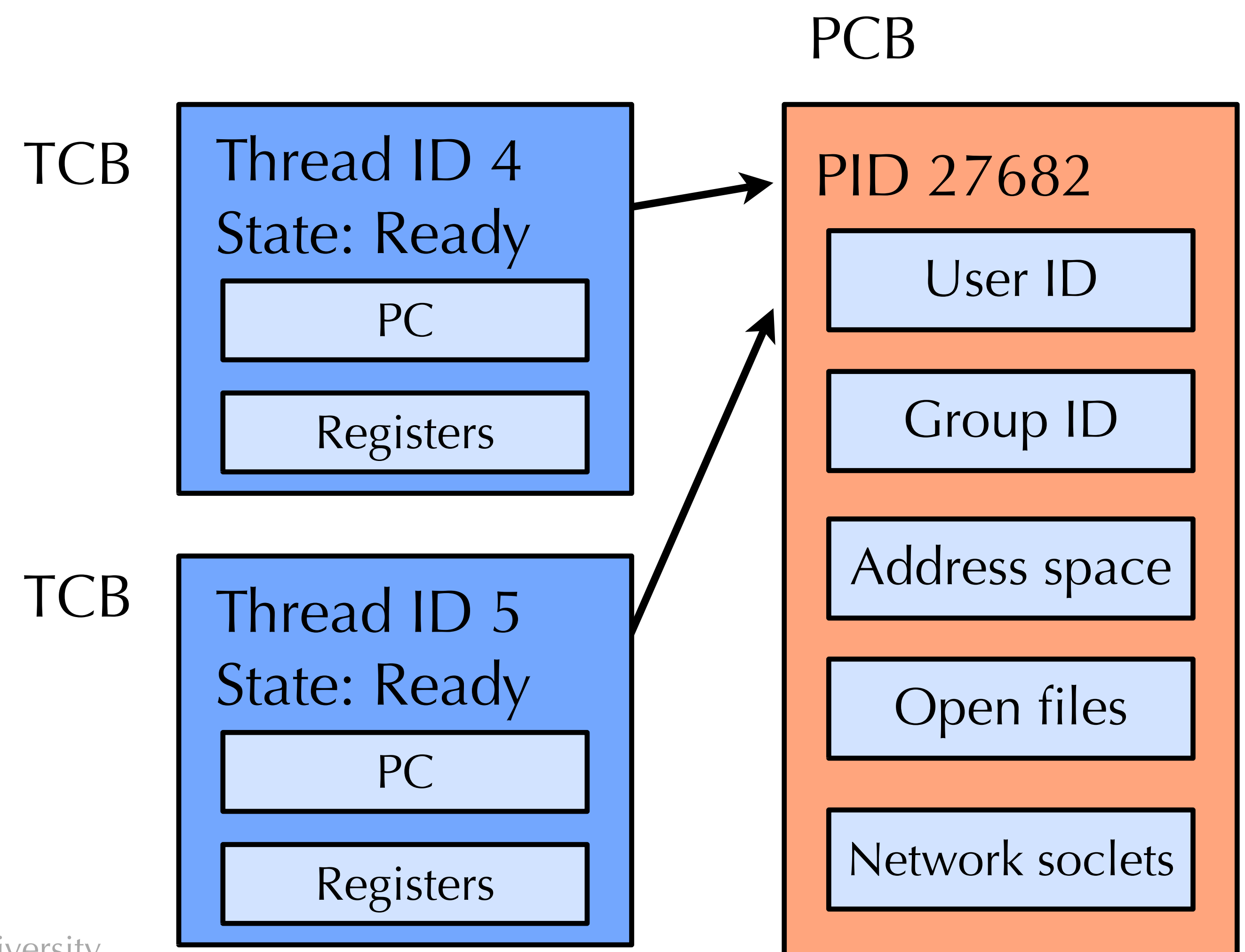
Topics for today

- Threads: Allowing a single program to do multiple things concurrently.
- Implementing
- Scheduling
- Private vs. shared memory
- Programming with threads (pthreads library)

- Reading: 12.1, 12.3

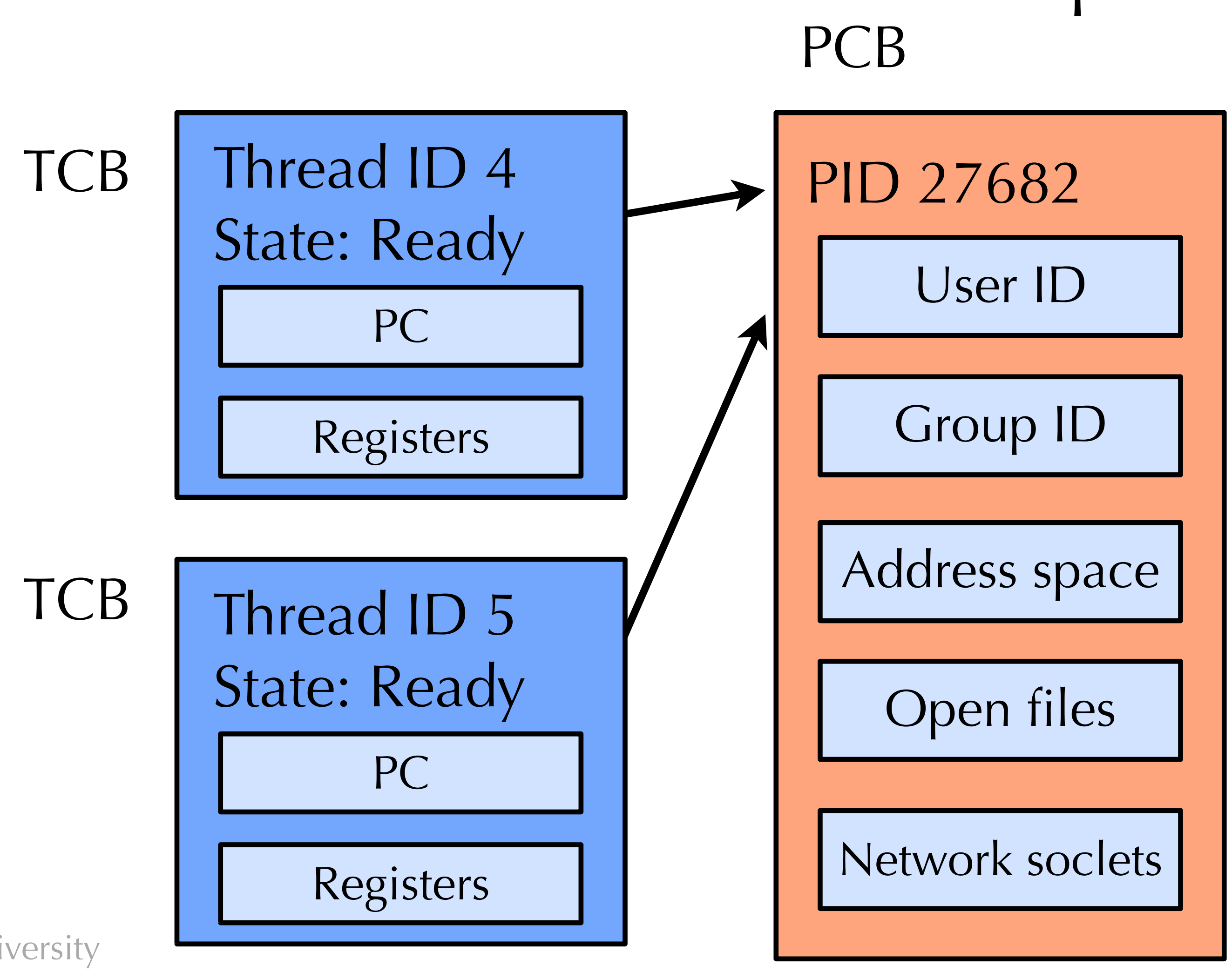
Implementing Threads

- 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.



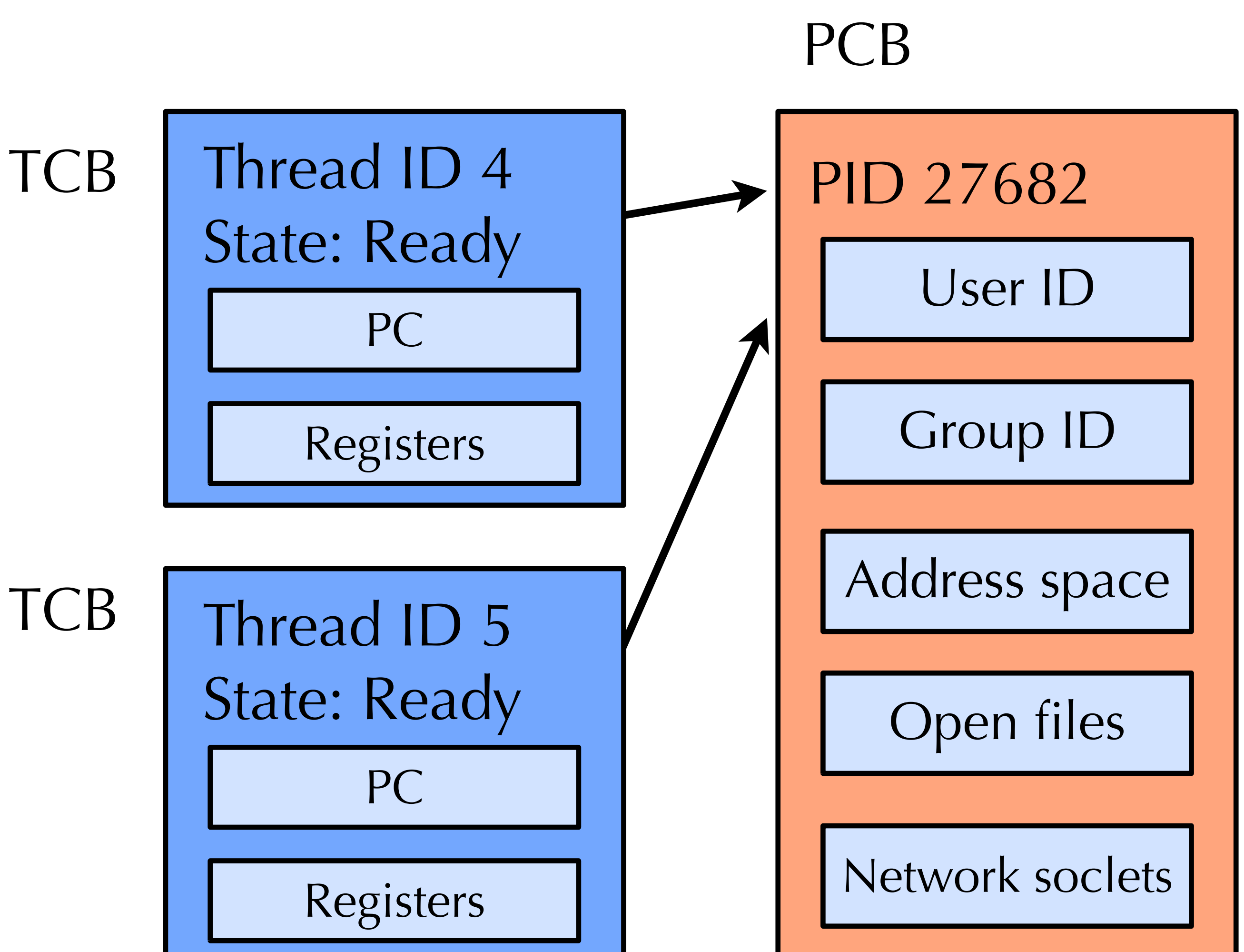
Thread Control Block (TCB)

- 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)

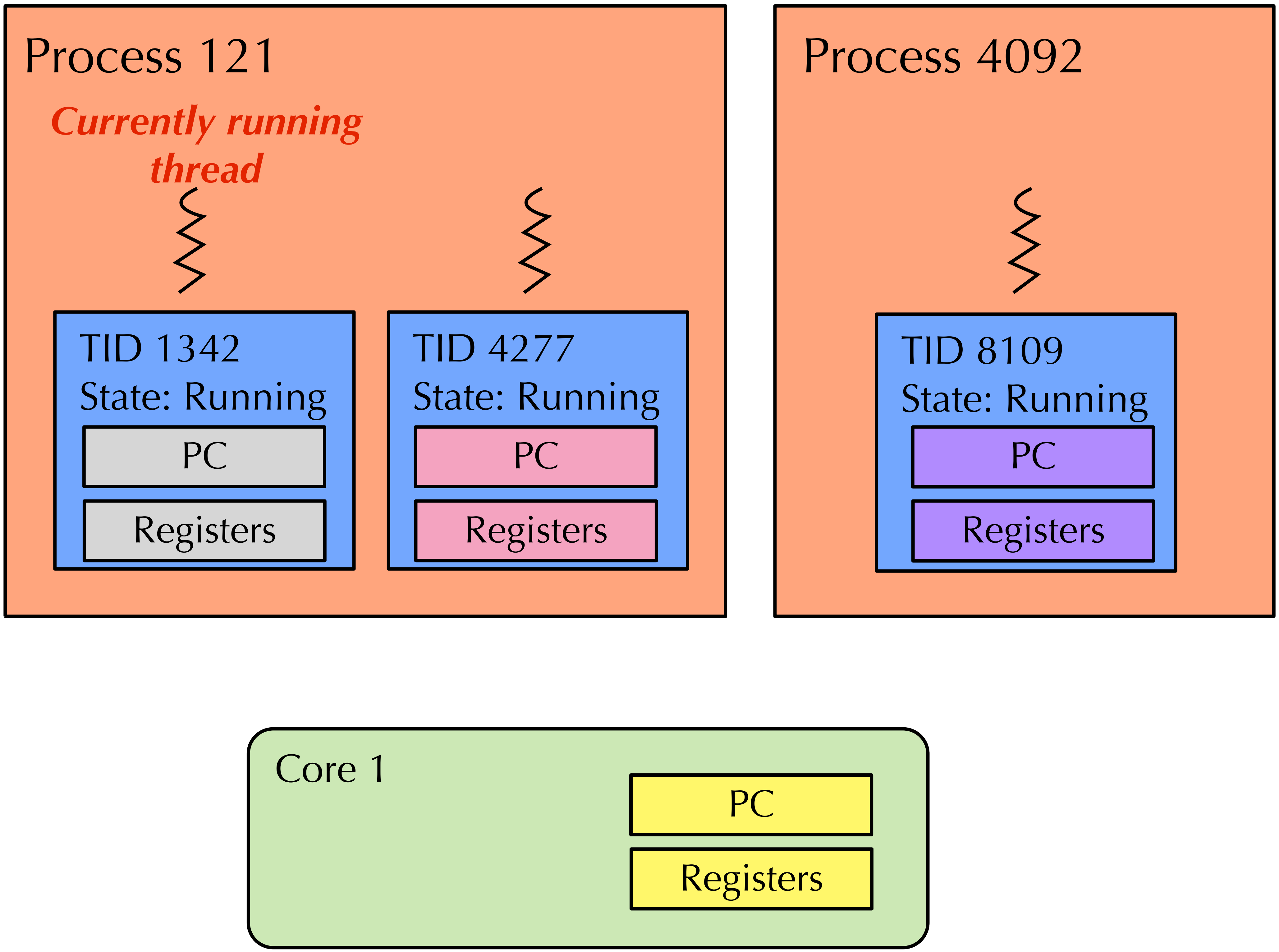
- TCB's are smaller and cheaper than processes
  - Linux TCB (`thread_struct`) has 24 fields
  - Linux PCB (`task_struct`) has 106 fields



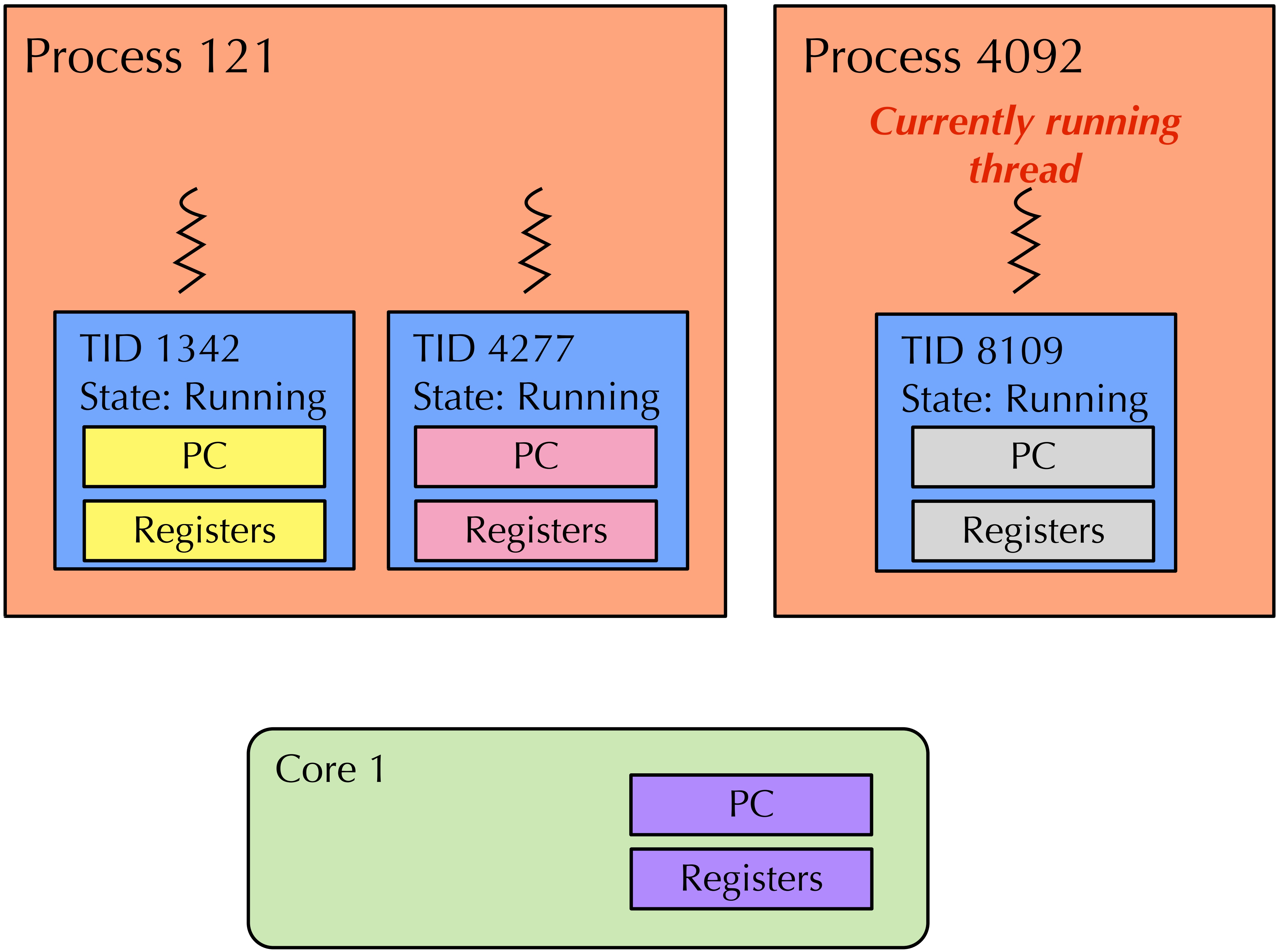
Threads, scheduling, and cores

- 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.

Topics for today

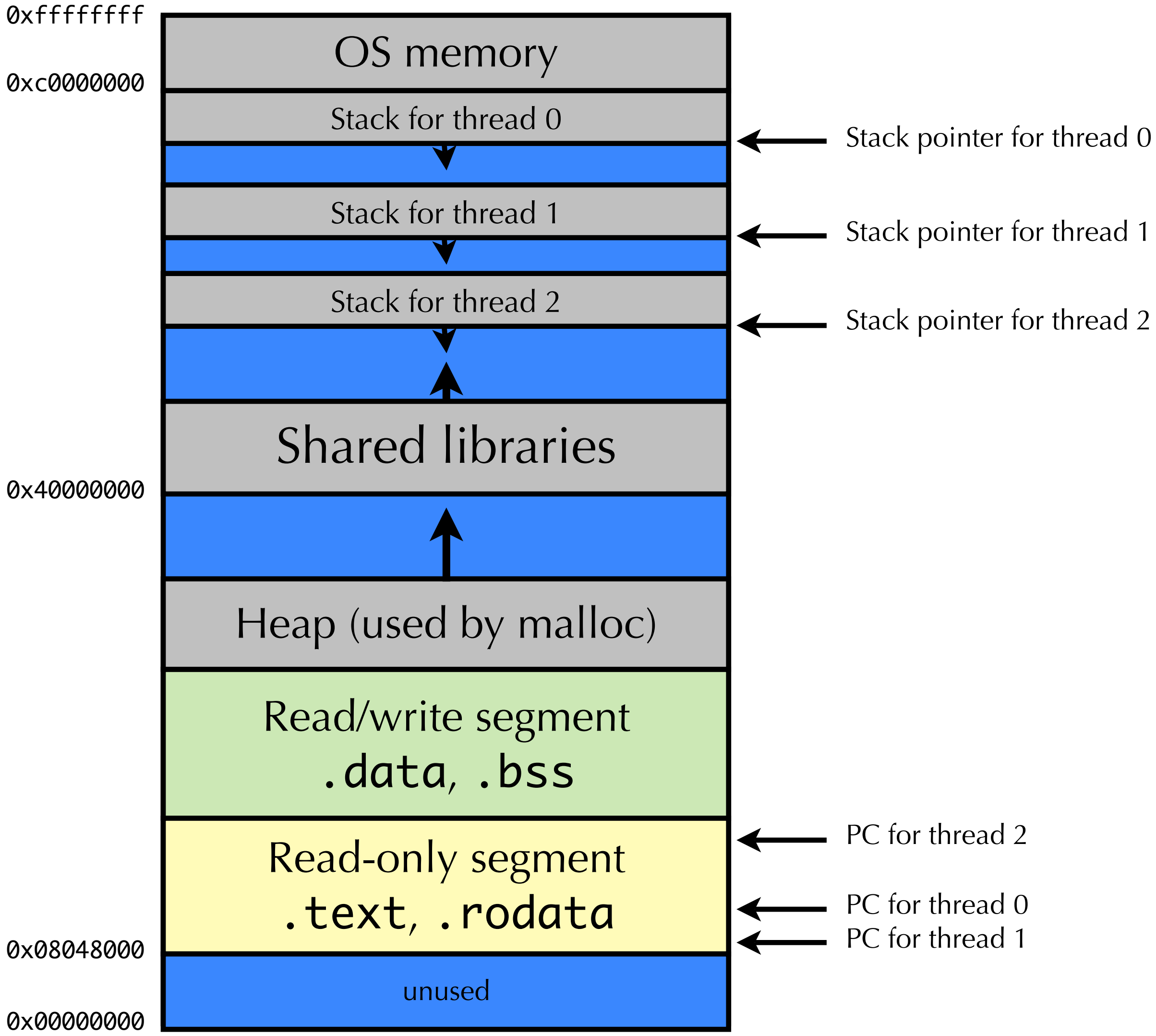
- Threads: Allowing a single program to do multiple things concurrently.
- Implementing
- Scheduling
- Private vs. shared memory
- Programming with threads (pthreads library)

- Reading: 12.1, 12.3

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);
}
```



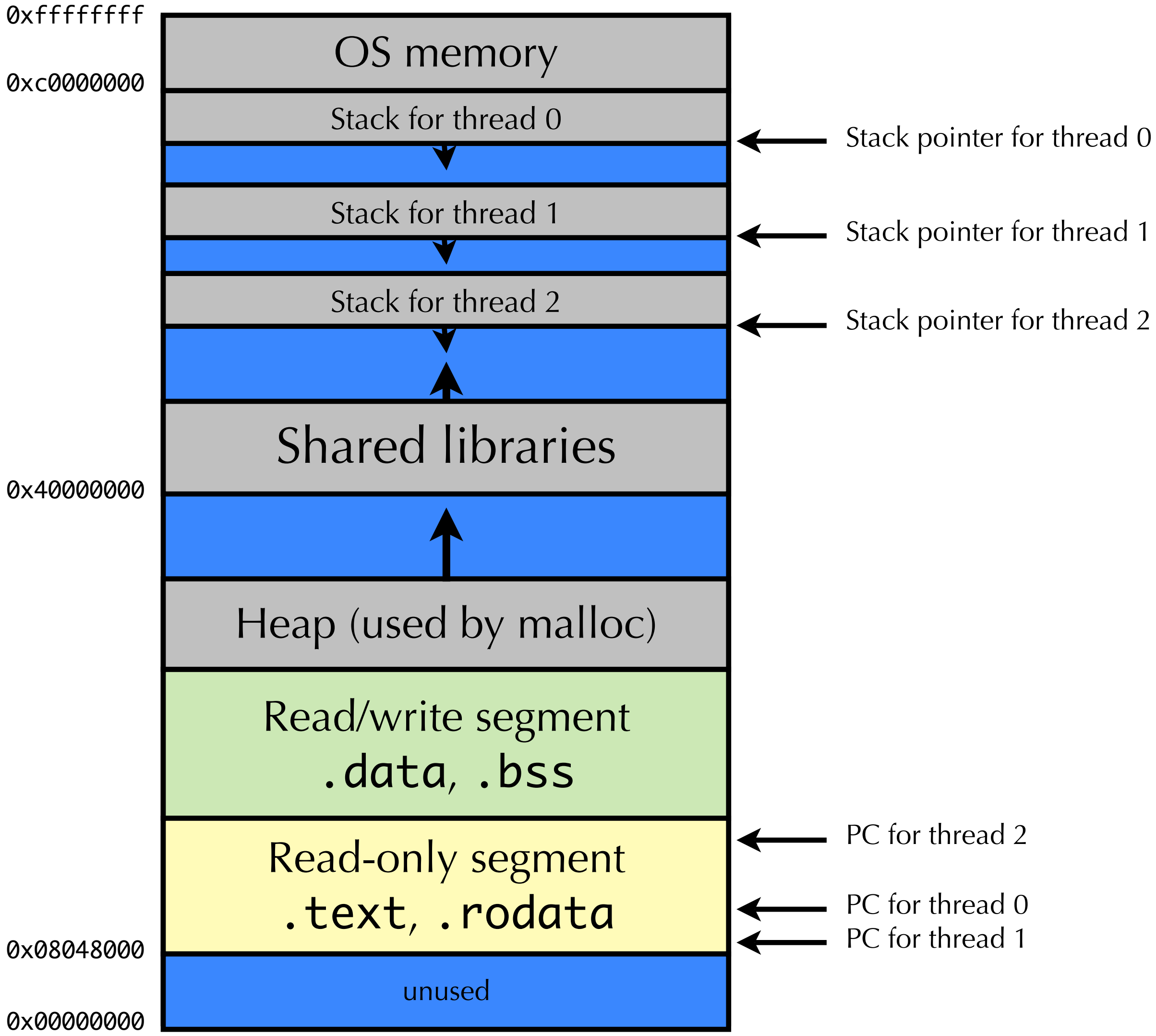
  - Both output 1
- Local variables are not shared
  - Each thread has its own stack

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

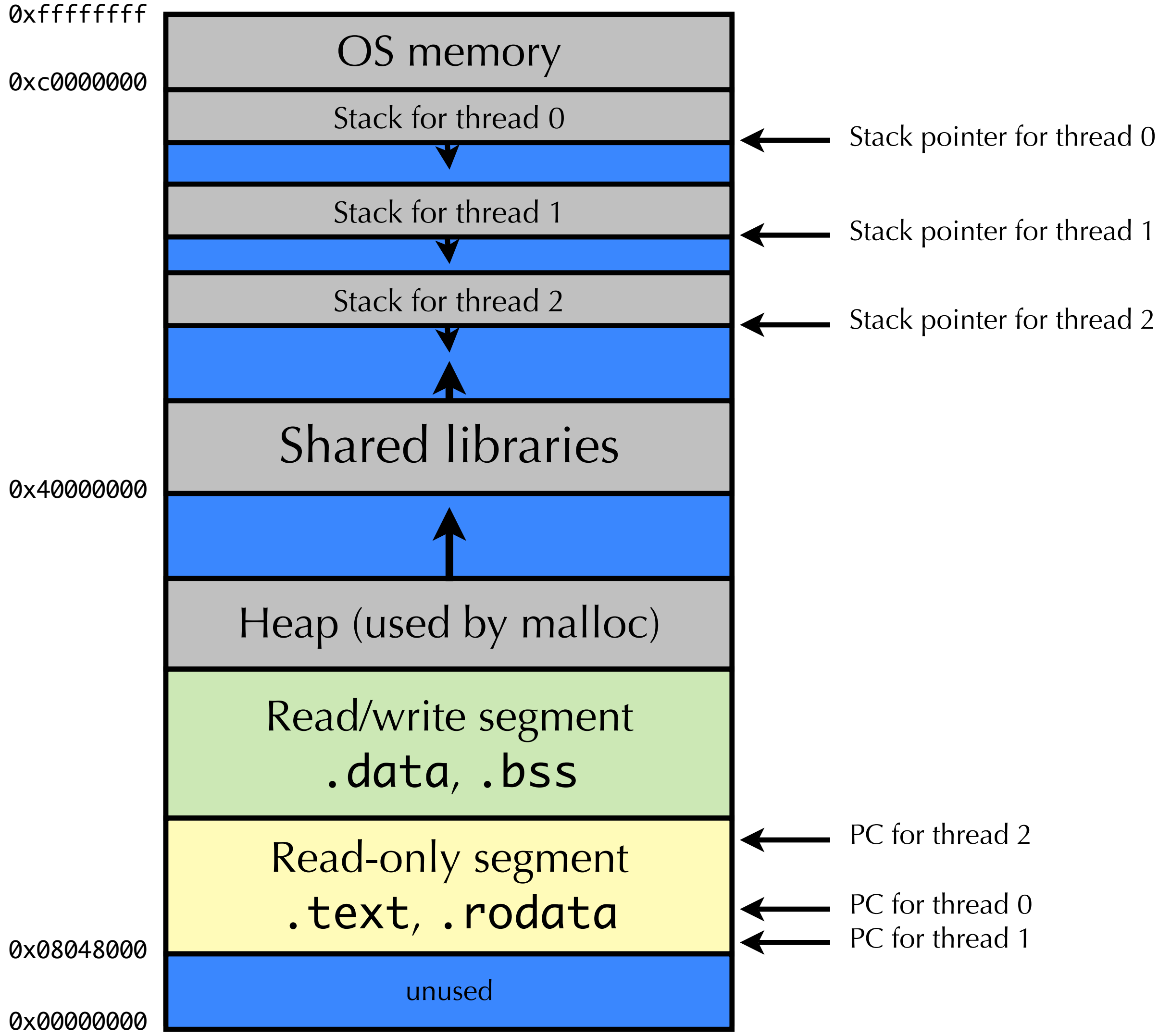


Local and global variables

- 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
- Private vs. shared memory
- Programming with threads (pthreads library)

- Reading: 12.1, 12.3

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);`

```
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

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", 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);
}
```

```
...
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!

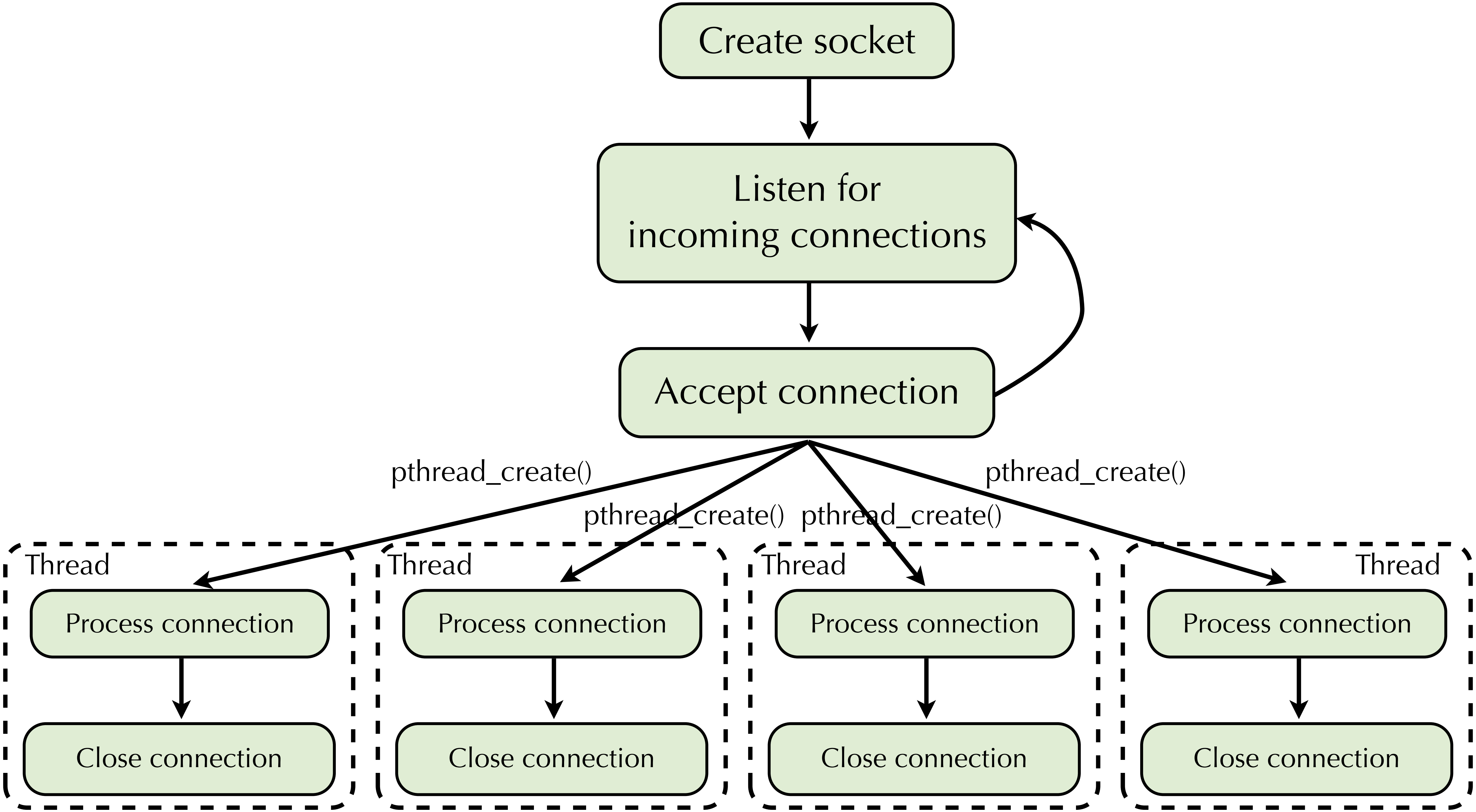
Reaping threads

- `int pthread_join(pthread_t tid, void **thread_return);`
  - Waits for thread `tid` to exit, returns return value of the thread
  - Reaps any memory resources held by terminated thread
  - Can only wait for a specific thread
    - Different from processes!

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?

Multithreaded server



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?