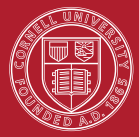


Cornell University

Project 1: Non-preemtive Multitasking

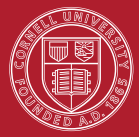
CS 4411: OS Practicum

Kai Mast

Department of Computer Science
Cornell University

September 4, 2014

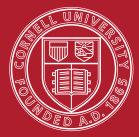
Announcements



Cornell University

- Project 1 is on online; due on September 16th
- Make sure you're on CMS and have a project partner
- Up to 2 "free" late days for this project
- Up to 4 late days the whole semester

Contents



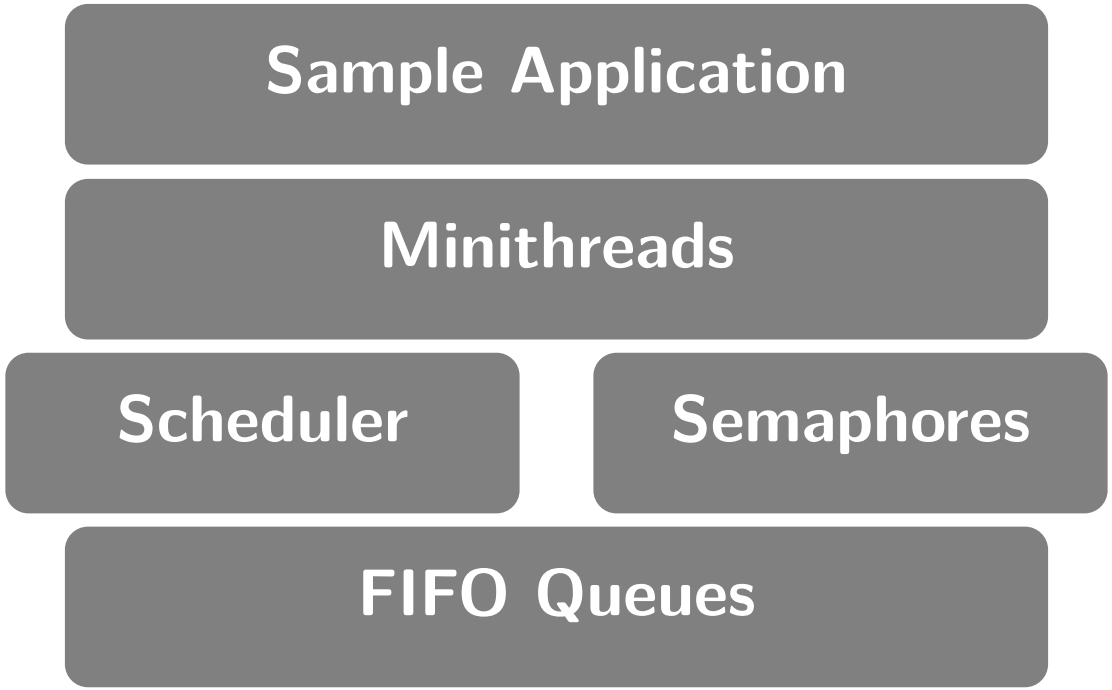
Cornell University

Goals of this Project

Cornell University

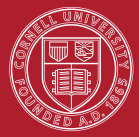
- Revive your C skills
- Learn how threading in scheduling work
- Implement basic synchronization primitives
- No practice without theory
  $\Rightarrow$ make sure you know how they work ⌣̈

Project Structure



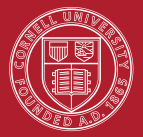
Not to be solved in this order.

Contents



Cornell University

Starting Point



To implement

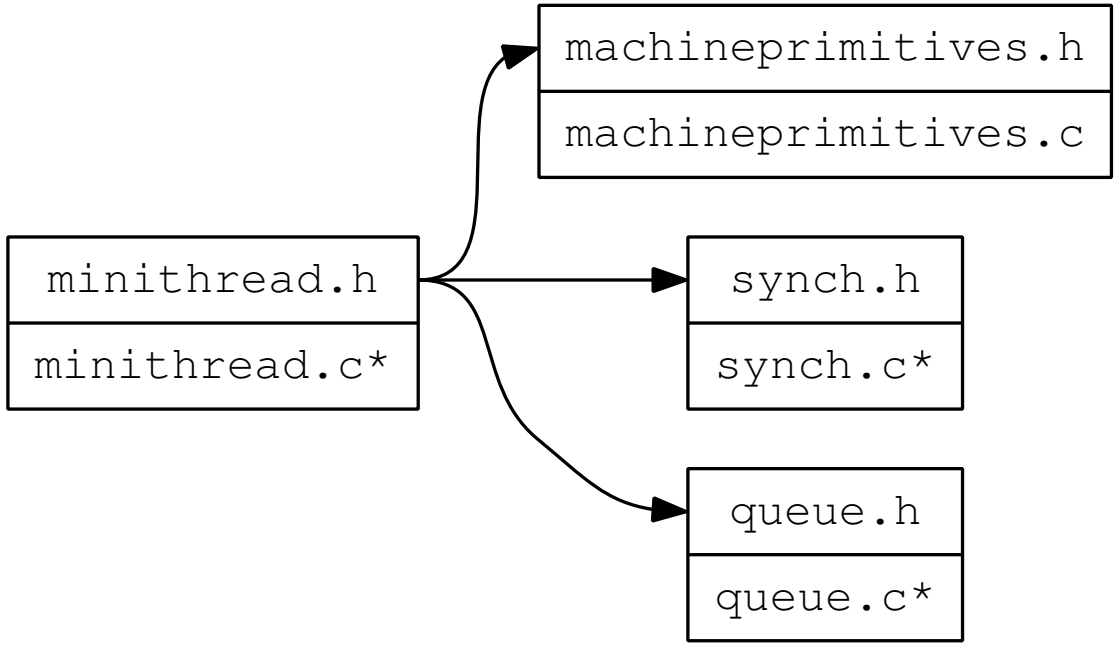
Interfaces for the queue (`queue.h`), minithreads (`minithread.h`), and semaphores (`synch.h`).

Helper functions

- Machine specific parts (`machineprimitives.h`).
  - Context switching, stack initialization, etc.
- No need to modify this code!
- No need to write your own context switching implementation!

Starting Point

File Structure



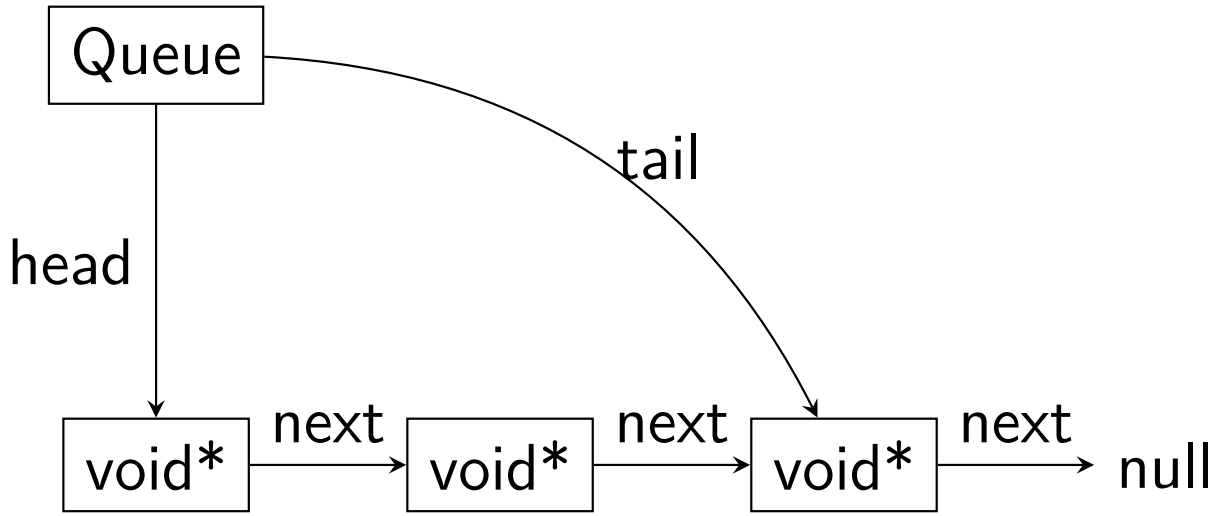
* need to be implemented

Queues
Motivation

Cornell University

- Every system needs efficient datastructures
- FIFO queues are useful in many parts of an OS
  - Think of how a scheduler could work
  - Or how processes could queue up for a resource

Queues
Linked Lists



Where is the data?

Queue Example

Usage:

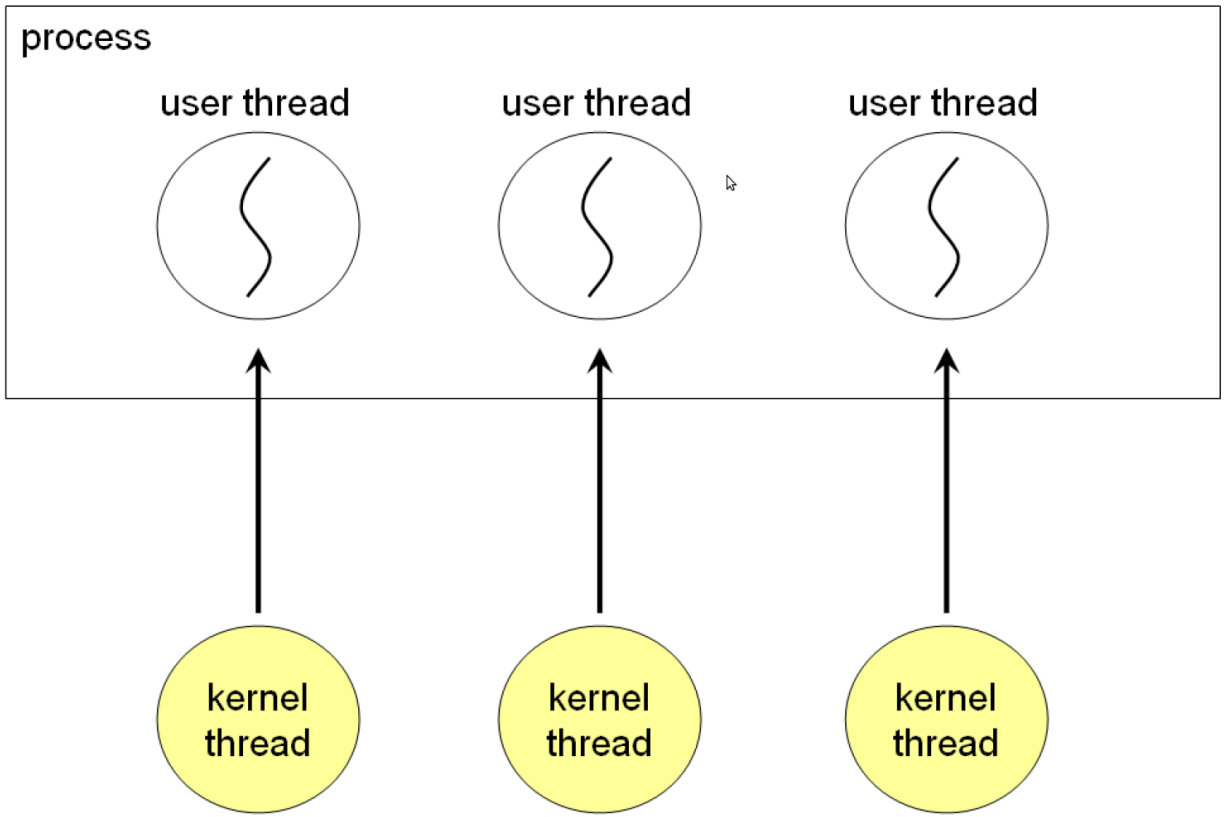
```
void* item = NULL;
int ret = queue_dequeue(run_queue, &item);
// item will point to something if ret == 0
```

Internals:

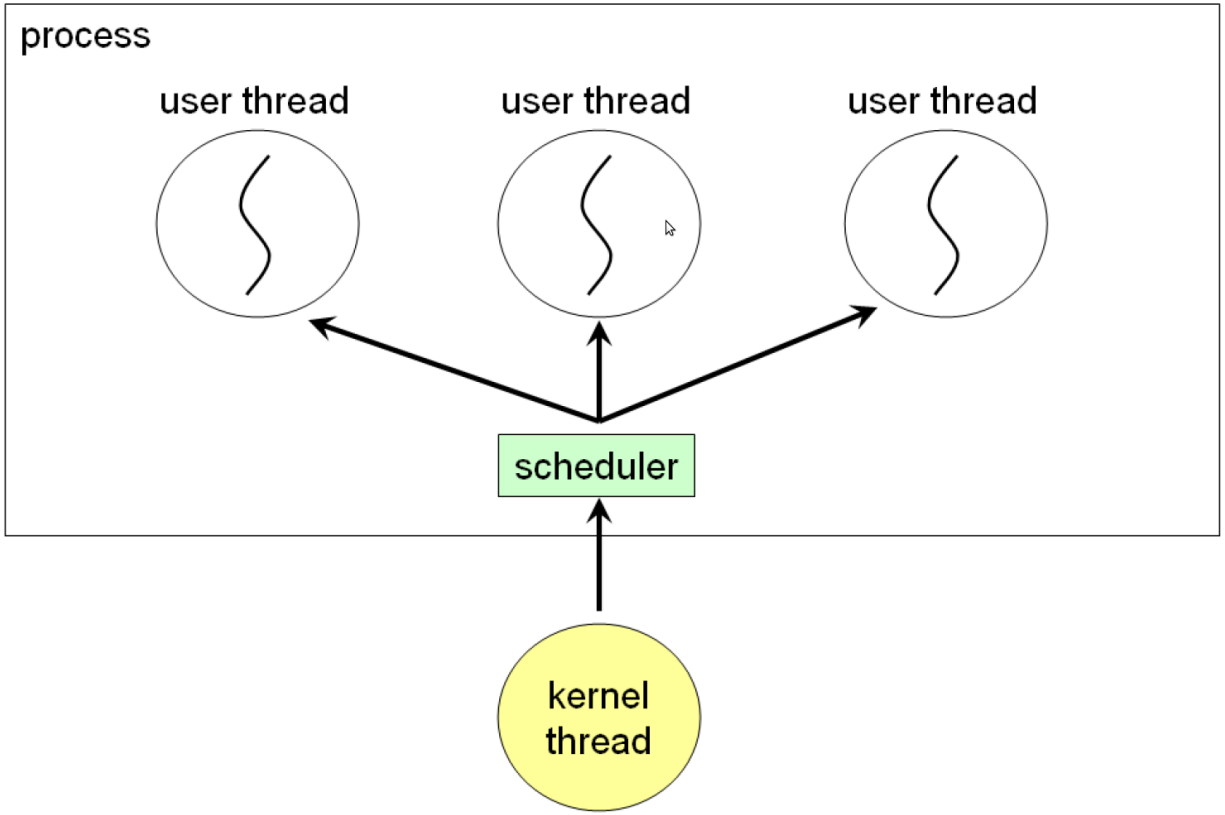
```
int queue_dequeue(queue_t *queue, void **item) {
  [...]
  *item = queue->head->datum;
  [...]
}
```

What if the queue is empty?

Kernel Threads



User Threads



This is what minithreads will do!

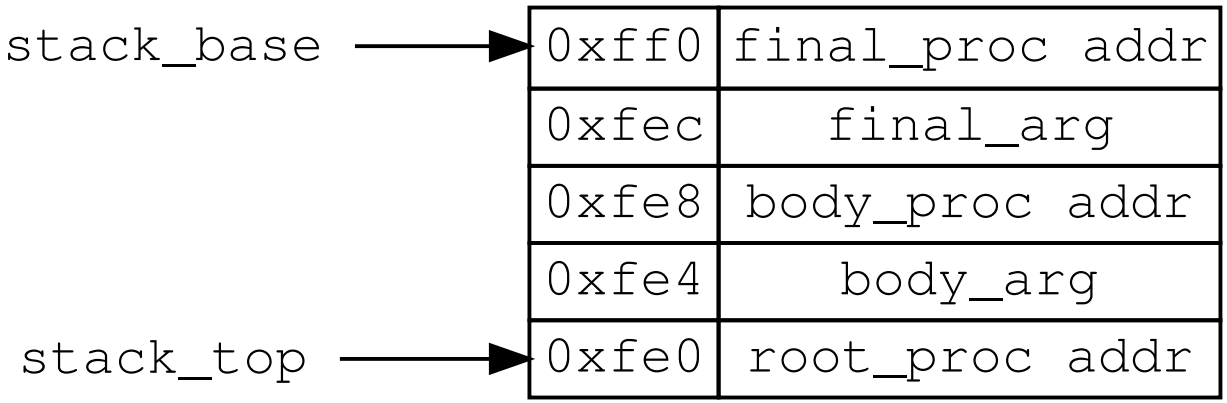
Minithread Structure

Cornell University

- Need to create a Thread Control Block (TCB) for each thread
- The TCB must have:
  - Stack top pointer (saved esp)
  - Stack base pointer (points to allocated stack)
  - Thread identifier
  - Anything else you find useful

Setting up the stack

- `minithread_allocate_stack` allocates memory
- `minithread_initialize_stack` set up stack content



Context switching

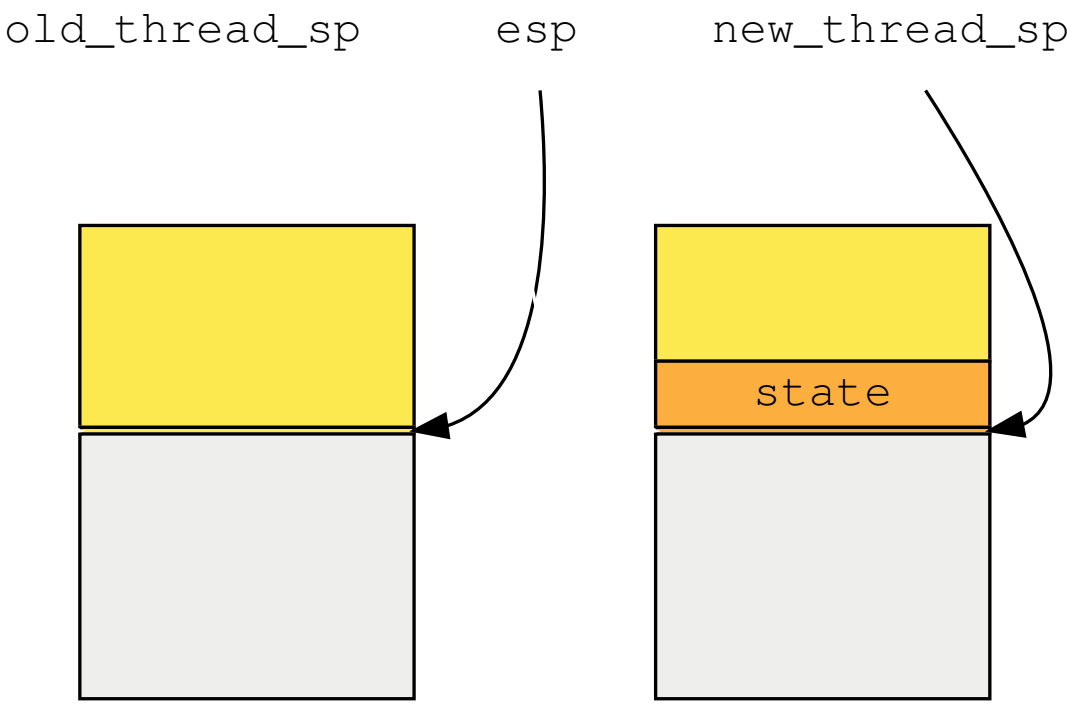
- Swap the currently executing thread with one from the run queue.
- State to save:
  - Registers
  - Program counter
  - Stack pointer

We give you a function for this:

```
void minithread_switch(stack_pointer_t *
    old_thread_sp, stack_pointer_t *new_thread_sp);
```

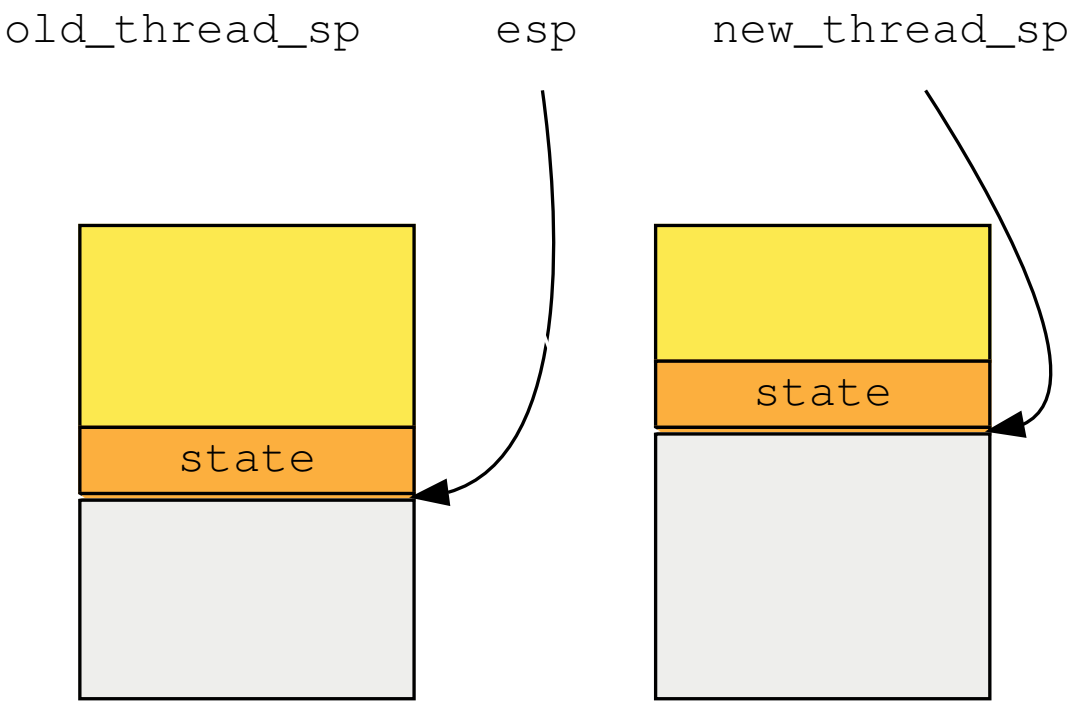
Context Switching

Initial state



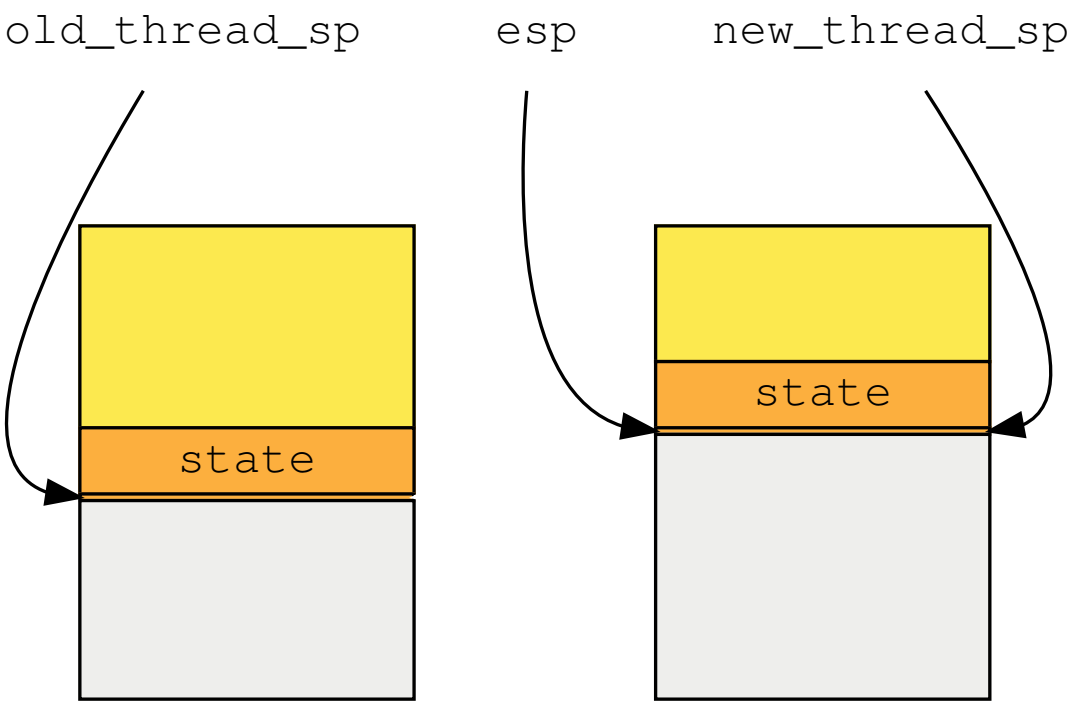
Context Switching

Push old context



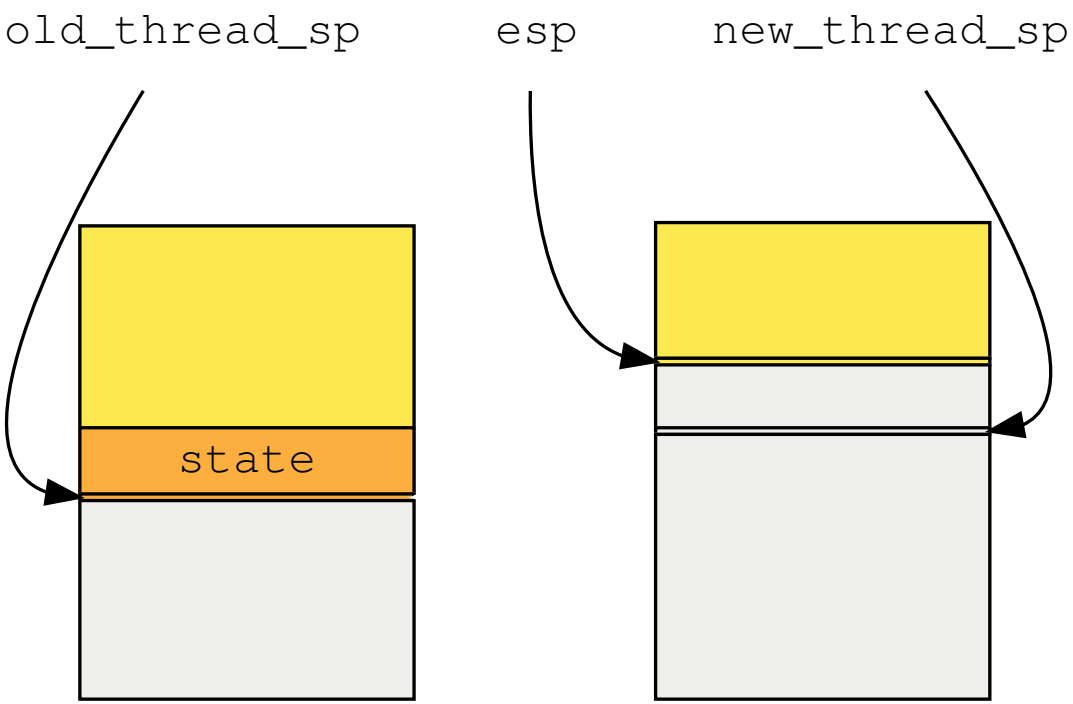
Context Switching

Change Stack Pointers

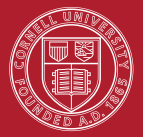


Context Switching

Pop off new context



Bootstrapping



Cornell University

```
void minithread_system_initialize(proc_t mainproc,
    arg_t mainarg)
```

- Starts up the system, and initializes global datastructures
- Creates a thread to run `mainproc(mainarg)`
- This should be where all queues, global semaphores, etc. are initialized.

Bootstrapping

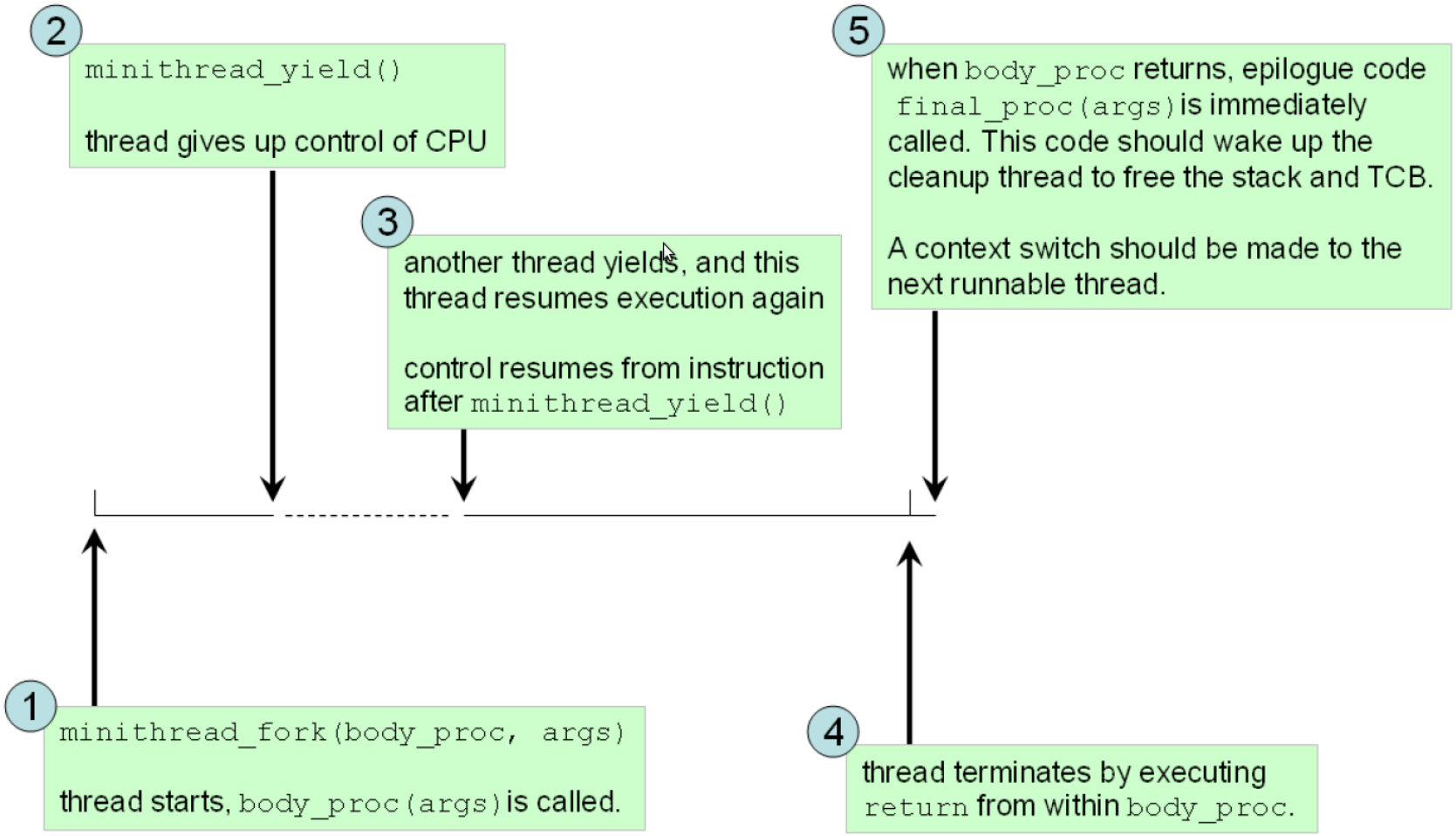
- How do we get from a full-blow unix process to a minithread?
- Host thread can be reused as idle thread
  - No need to allocate stack here
  - TCB's stacktop and stackbase should be `NULL`
  - Don't try to clean up this stack!
- The program should never really exit, so it is a good idea to use the host thread (which never should be terminated) as the idle thread

Scheduling

Cornell University

- We haven't specified any preemption. We need a way to voluntarily switch between threads.
  `void minithread_yield()`
- Use `minithread_switch` to implement `minithread_yield`
- What happens to the yielding thread?
- Is this a fair execution model?

Minithread Lifecycle



Concurrency 101

- Race condition: result of computation depends on the relative running speed of threads.
  - Multiple concurrent threads reading from/writing to the same memory location.
  - E.g. two threads manipulating a linked list.
- Atomic operation: either the operation goes to completion, or fails altogether
- Deadlock: Two operations/threads wait on each other
- Starvation: An operation never gets to run and thus never completes

Synchronization

We want critical section of code to run without other threads interfering.

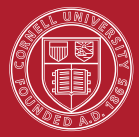
```
// Shared between threads
queue process_queue;
lock process_queue_lock;

void manipulate_queue {
  lock_acquire (process_queue_lock);
  // critical section begins
  queue_dequeue (process_queue);
  queue_append (minithread_self);
  // critical section ends
  lock_release (process_queue_lock);
}
```

Semaphores

One (of many) synchronization primitives

- You decide how many threads can concurrently hold the semaphore when initializing it.
- Semaphore value is manipulated atomically
  - `semaphore_P` decrements the value by 1
  - `semaphore_V` increments the value by 1
- Threads wait on a semaphore
  - if count is 0, `semaphore_P` blocks
  - if count is 0, `semaphore_V` wakes up waiting threads
- Special case: binary semaphore is a lock

Contents



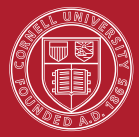
Cornell University

Test your code!

- We supply some basic tests.
  - Read them to understand how minithreads work
- Statistically, there are a large number of untested potential bugs.
- Write some (or many?) tests of your own (be abusive to minithreads; it can take it).

Coding Style

Avoid unnecessary polling




```
while(some_var != 42) {
    minithread_yield();
}
```

Unnecessary context switches are expensive and should be avoided![1]

[1] Remember this when implementing your semaphores and scheduler.

Coding Style
Comments

Comments make it easier for us to give you a good grade.

```
int x = 0; //set the variable x to 0
assert(x == 0); //make sure that this x is 0
```

...but shouldn't be too verbose.[2]

[2] That example is too verbose

Coding Style

Variable names

Cornell University

Whats better?

```
int y_1 = 42;
```

or

```
int thread_identifier = 42;
```

Coding Style

Use const if possible

This will compile

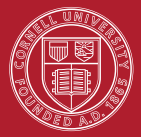
```
int some_constant = 1;

//somebody wrote = instead of ==
if(some_constant = 0)
  do_something();
```

This won't

```
const int some_constant = 1;

if(some_constant = 0) //cannot assign to a constant
  do_something();
```

Common Errors

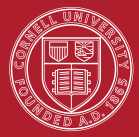
Weak type system




```
int *my_int_pointer = malloc(sizeof(int));
my_int_pointer = 4;
```

This is probably not what you wanted to do.

Thanks



Cornell University

Have fun solving the project!

Slides inspired by previous TA's: Sean Odgen, Robert Escriva, Z. Teo, Ayush Dubey, et al.