# Project 1

## Non-Preemptive Multitasking with minithreads

Zhiyuan Teo

Cornell CS 4411, September 9, 2011

# Administrative Information

- Project 1 has been published, due 11.59pm September 18.
- Please make sure your partner is still in the course!
- All unpartnered students will be purged from CMS this week.
- No formal lecture next week, instead it will be an FAQ session with some tips.

# Goals of this project

- Learn how threading and scheduling work.
- Actually implement said processes.*

* "In theory, there is no difference between theory and practice. But, in practice, there is." Jan L. A. van de Snepscheut

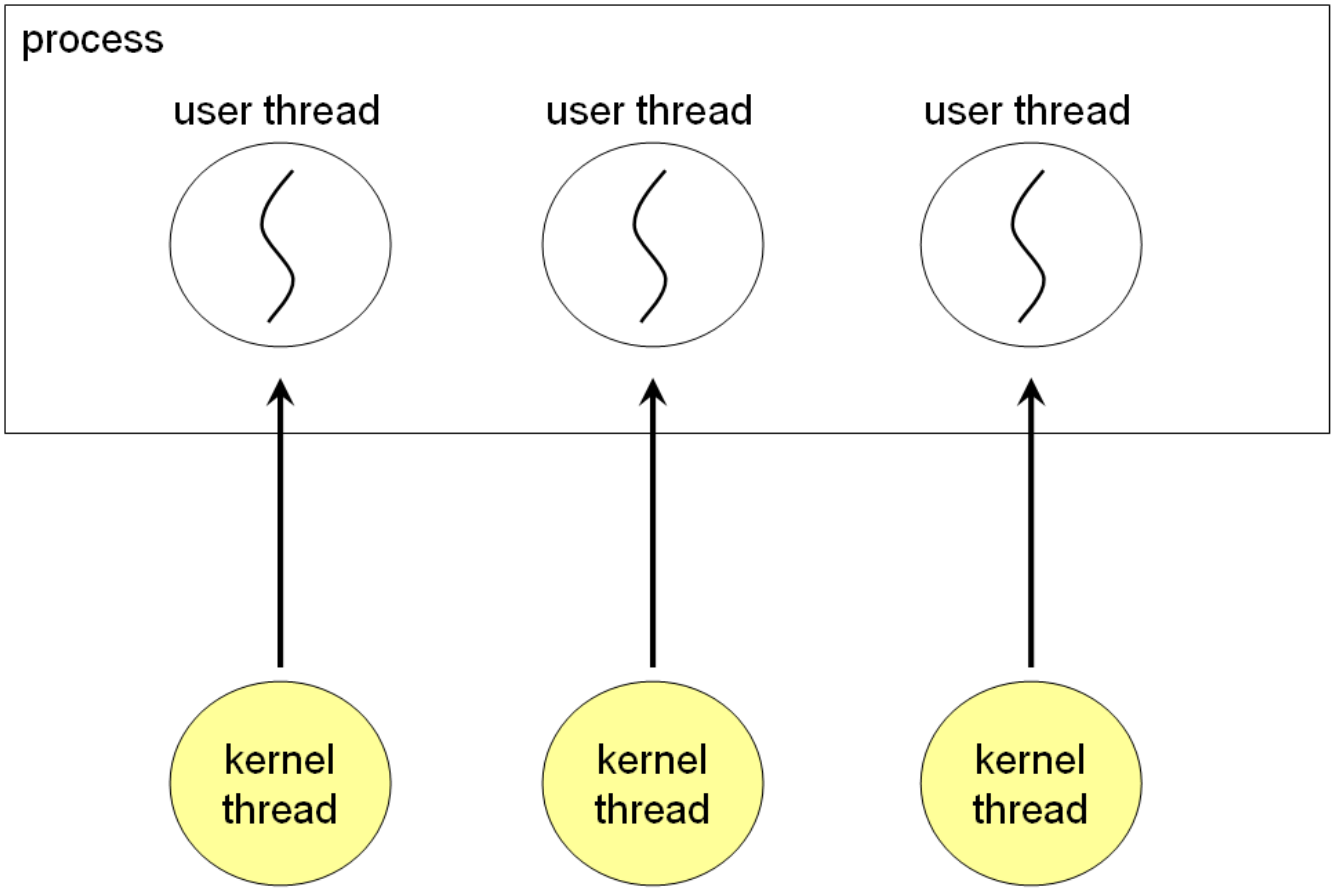
# Deliverables

- A working implementation of minithreads.
- Required pieces (we recommend this order for implementation)
  - FIFO Queue with O(1) append/prepend/dequeue.
  - Non-preemptive threads and FCFS scheduling.
  - Semaphore implementation.
  - A simple "barber shop" application.
- Optional (for those itching to start on Project 2):
  - Add preemption.
  - Optional material is not graded for this project; focus on getting Part 1 right.
  - E-mail or meet the course staff before beginning on the optional component.

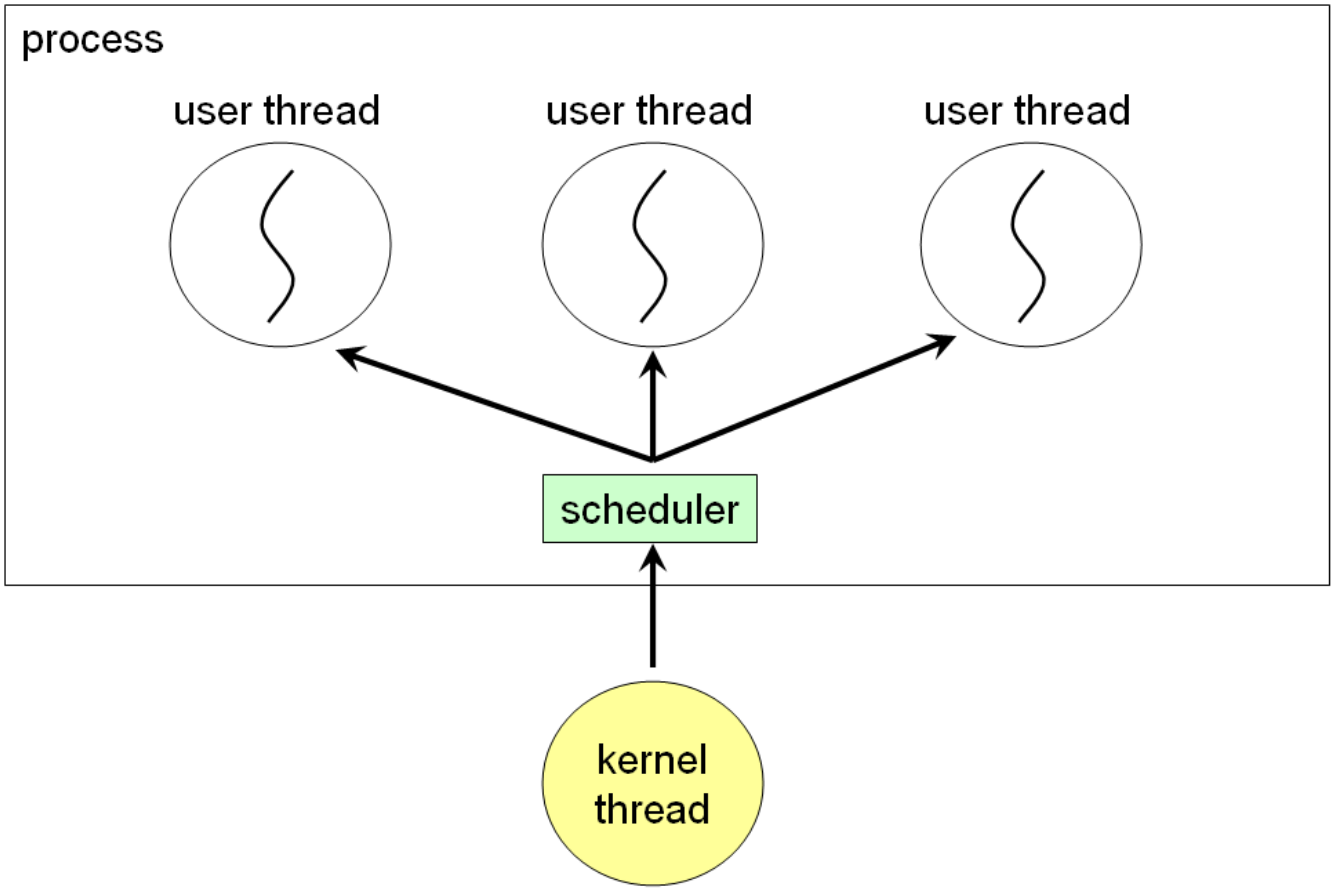
# What are minithreads?

- User-level threads for Windows NT/2000/XP/Vista/7
  - Windows only provides kernel-level threads; user-level threads can perform better in some cases.
- User-level threads can also be useful in OSes that do not support kernel threads.

# Kernel threads



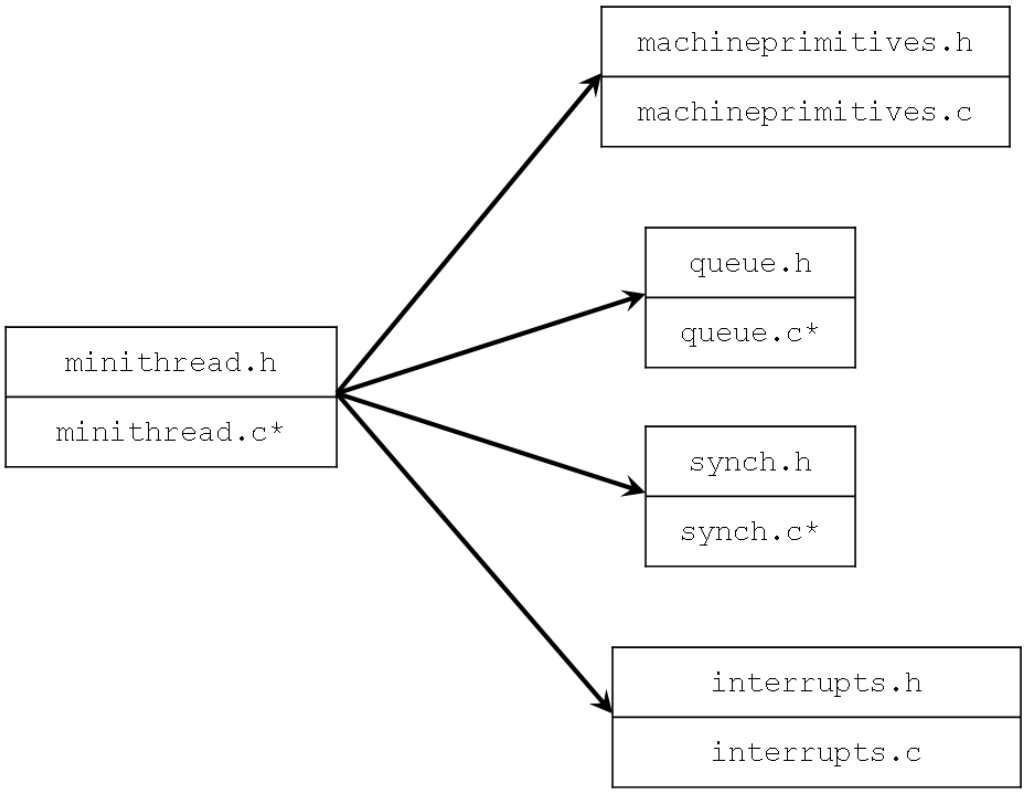
# User threads (minithreads)



# Starting point

- Interfaces for the queue (`queue.h`), minithreads (`minithread.h`), and semaphores (`synch.h`).
- Machine specific parts (`machineprimitives.h`).
  - Context switching, stack initialization, etc.
- Simple (non-exhaustive) test applications.
  - Statistically, there are a large number of untested potential bugs.
  - Write some tests of your own (be abusive to minithreads; it can take it).

# Minithreads structure




* files to finish implementing

# Noteworthy things while coding

- Read the comments in `machineprimitives.h`.
- You may (will) need to define your own structures in `queue.h` and `minithread.h`.
- Do not modify the given data types and function signatures!

# Why Queues?

- Queues are needed to hold thread control blocks (TCBs).
- Why a queue? Because of FCFS scheduling.
- Since each thread control block can only be on at most one queue, you don't have to worry about deep copying of queue nodes.
- Think about the queues you will need by considering the possible states of each thread.

# Implementation

- Refer to last lecture for implementation details.
- You can use `malloc()` and `free()` within your queue functions, but try to see if you can do without.
- You will need to define your own `struct queue` and interior list nodes (do it inside `queue.h`).
- Don't change the function signatures.

# Useful things to know

- We will not be testing your queue implementation directly, so you can make some assumptions.
- `any_t` can be a pointer to data or a list node (you get to choose).
- Within your queue functions, cast `any_t` into the type you are working with.

# `any_t` as a pointer to data

- List nodes are not visible to the caller.
- You will need to allocate/free list nodes within the queue functions.
- Enqueue a pointer to raw data; dequeue gives you back that pointer to the raw data.
- Catch: you will need to use `malloc()` and `free()` in your queue functions.

# `any_t` as a pointer to list node

- Caller has to allocate the list node and embed the user data before calling queue functions.
- No need to allocate/free list nodes in queue functions, just need to manipulate pointers.
- Enqueue a pointer to list node; dequeue gives you back a pointer to the list node.
- Avoids `malloc()` and `free()` in queue functions.
- Catch: need to set aside space for list node pointers in every queueable object.
- List node implementation is somewhat visible to user.

# Example of `queue_dequeue`, pointer to data

Usage:

```
any_t datum = NULL;
queue_dequeue(run_queue, &datum);
/* check return value */
```

Internals:

```
int queue_dequeue(queue_t queue,
                  any_t* item) {
    *item = queue->head->datum;
}
```

# Minithread structure

- Need to create a Thread Control Block (TCB) for each thread.
- The TCB must have:
  - Stack top pointer (saved `esp`).
  - Stack base pointer (given to us by `minithread_allocate_stack`).
  - Thread identifier (an `int`).
  - Anything else you find useful.

# Operations to implement (`minithread.c`)

`minithread_t minithread_create(proc, arg);`
: Create a thread and but don't make it runnable yet.

`minithread_t minithread_fork(proc, arg);`
: Create a thread and make it runnable (ie put it on the ready queue).

`void minithread_yield();` Voluntarily gives up the CPU and lets the next thread in the ready queue run.

# Operations to implement (`minithread.c`)

`void minithread_start(minithread_t t);`
: Makes a thread runnable by putting it onto the ready queue. Useful in semaphore operations, or to start a thread after it has been created through `minithread_create()`.

`void minithread_stop();`
: Stops running a thread immediately (ie blocks the thread); the next scheduled thread on the ready queue should run. Also useful in semaphore operations.

# Creating minithreads

- Two methods
  - `minithread_t minithread_create(proc, arg);`
  - `minithread_t minithread_fork(proc, arg);`
- `proc` is a `proc_t` (a function pointer)

```
/* the definition of arg_t */
typedef int* arg_t;
/* the definition of proc_t */
typedef int (*proc_t) (arg_t);
/* how you declare a proc_t */
int run_this_proc (arg_t arg);
```

# Preparing for a new thread

- Create a stack for the new thread.
- Need to set up an initial execution context:
  - Thread entrypoint.
  - Parameters to pass along to the thread when it starts executing.
  - Thread cleanup handler.
  - Parameters to pass along to the thread cleanup handler when the thread exits.
- Once the execution context has been set up, we can run it! (by calling a special function)

# Allocating a stack

We give you a function to allocate the stack.

```
void minithread_allocate_stack
     (stack_pointer_t *stackbase,
      stack_pointer_t *stacktop);
```

Values are returned through function parameters.

# Setting up the initial execution context

The thread hasn't run yet, so there is no context to switch to. Setting up the initial context allows us to context switch to the entry point of the thread.

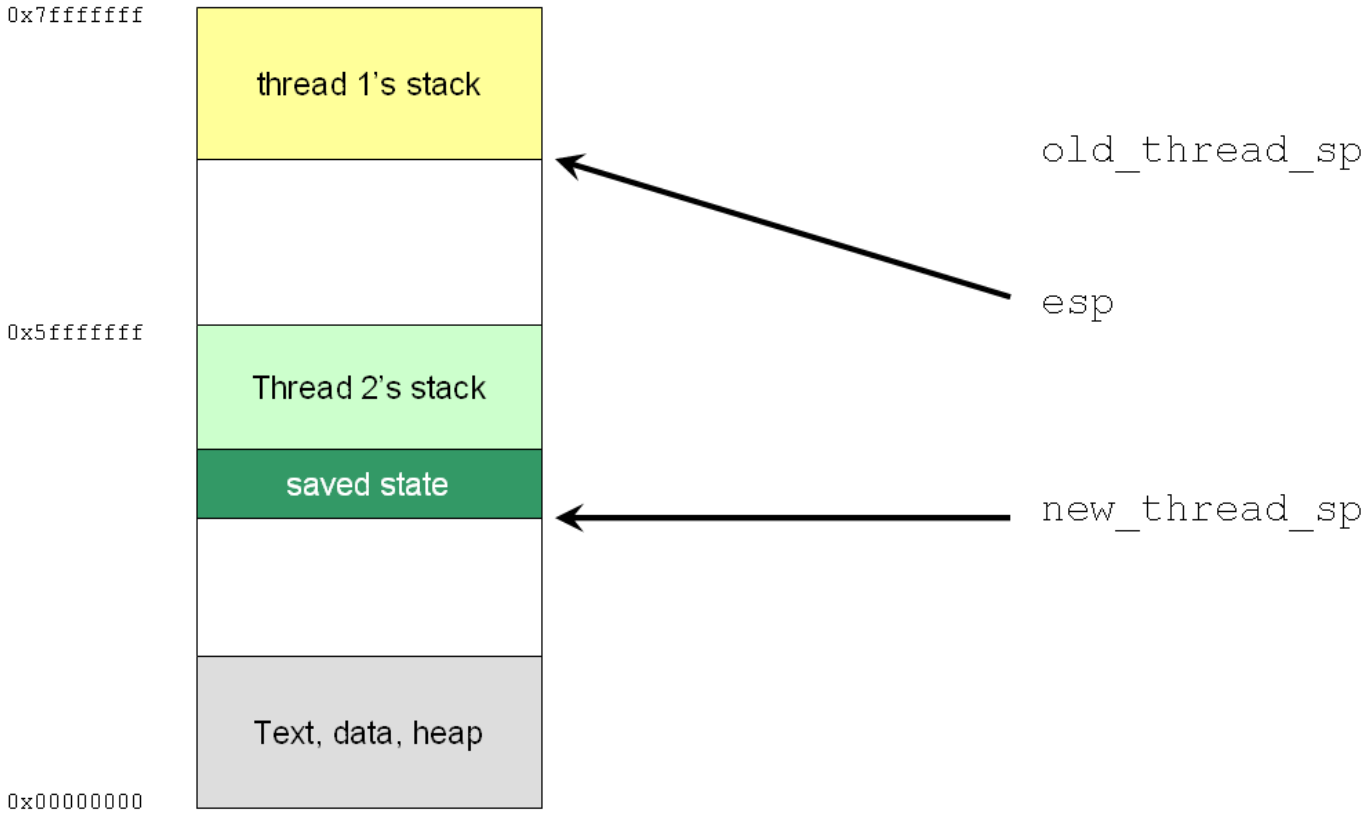
```
void minithread_initialize_stack
      (stack_pointer_t *stacktop,
       proc_t body_proc,
       arg_t body_arg,
       proc_t final_proc,
       arg_t final_arg);
```

# Context switching

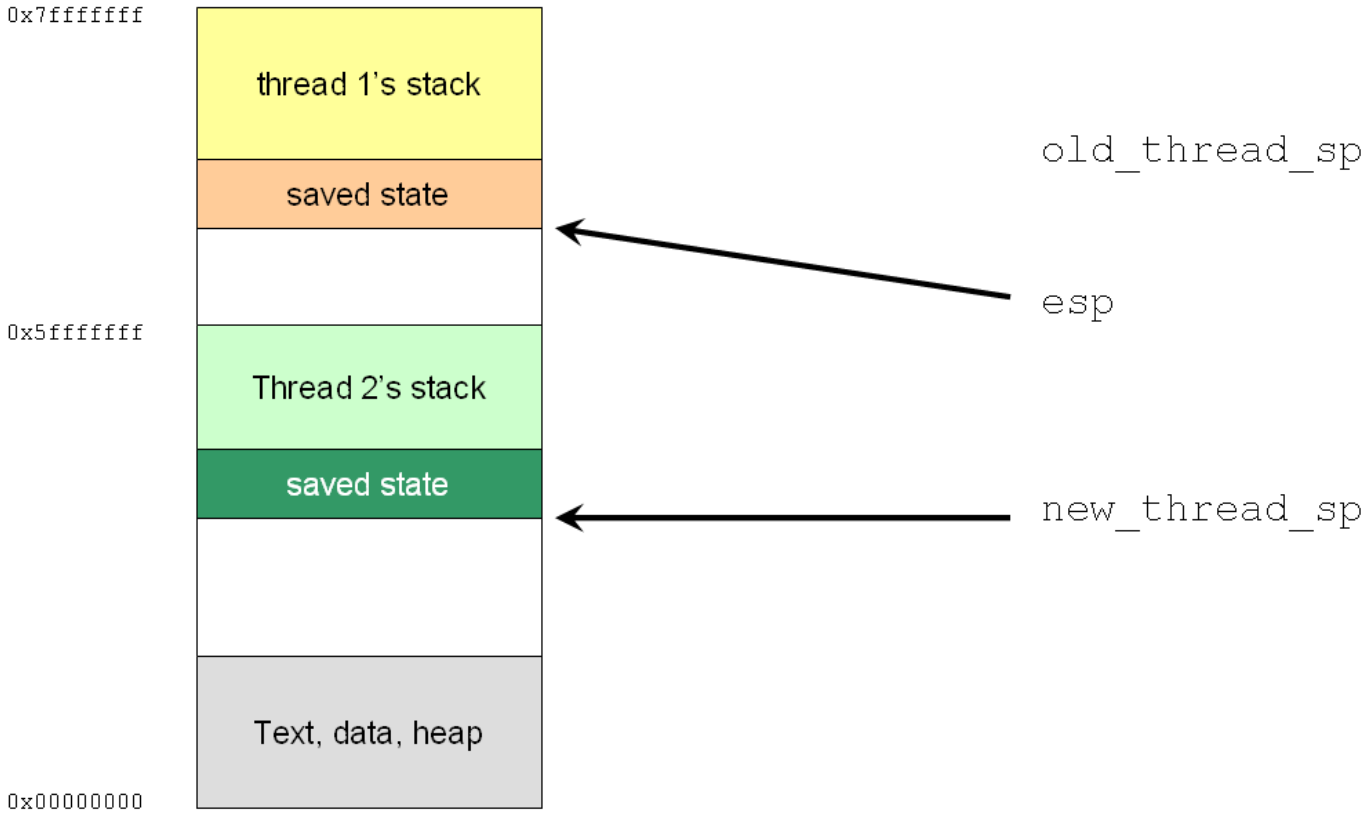
- Swap the currently executing thread with one from the ready queue.
  - Save the state of the current thread onto the stack.
  - Switch from current thread's stack to new thread's stack.
  - Restore the state of the new thread by popping context off the stack.
- We supply a function to do all this for you:

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

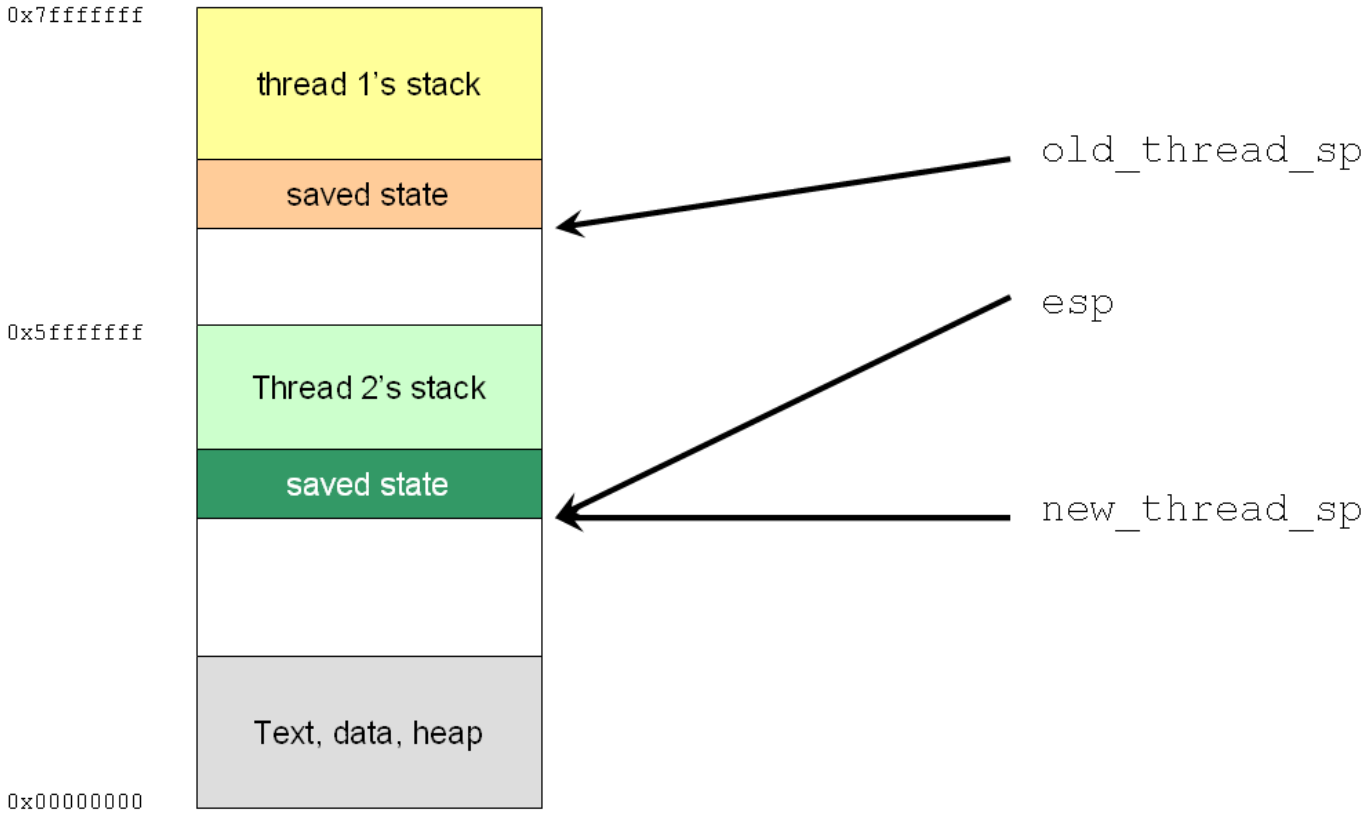
# Before starting a context switch



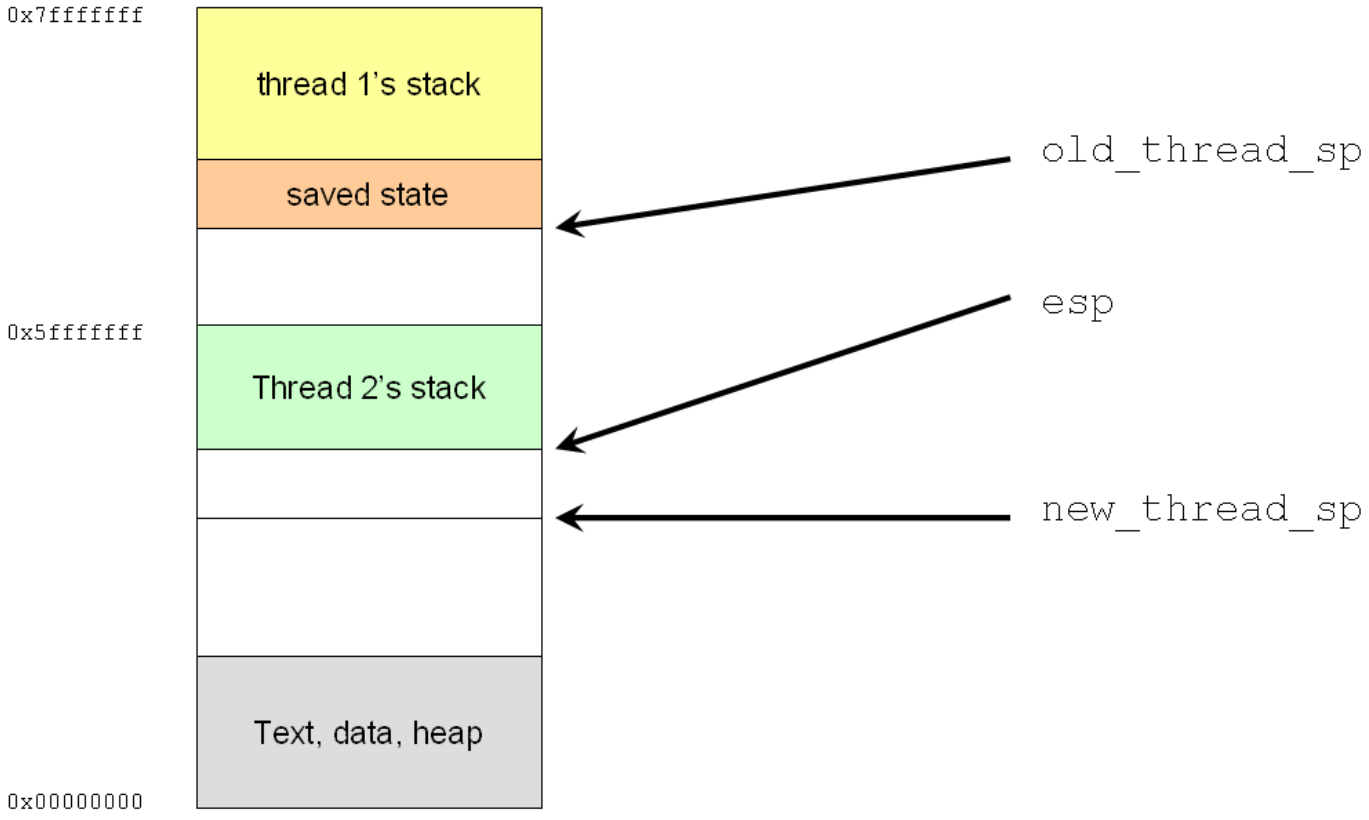
# Push old context



# Change stack pointers



# Pop off new context



# Yielding a thread

- We haven't specified any preemption. We need a way to voluntarily switch between threads.

```
void minithread_yield();
```

- Use `minithread_switch` within `minithread_yield` to do context switch.
- Yielding thread goes to the back of the ready queue.

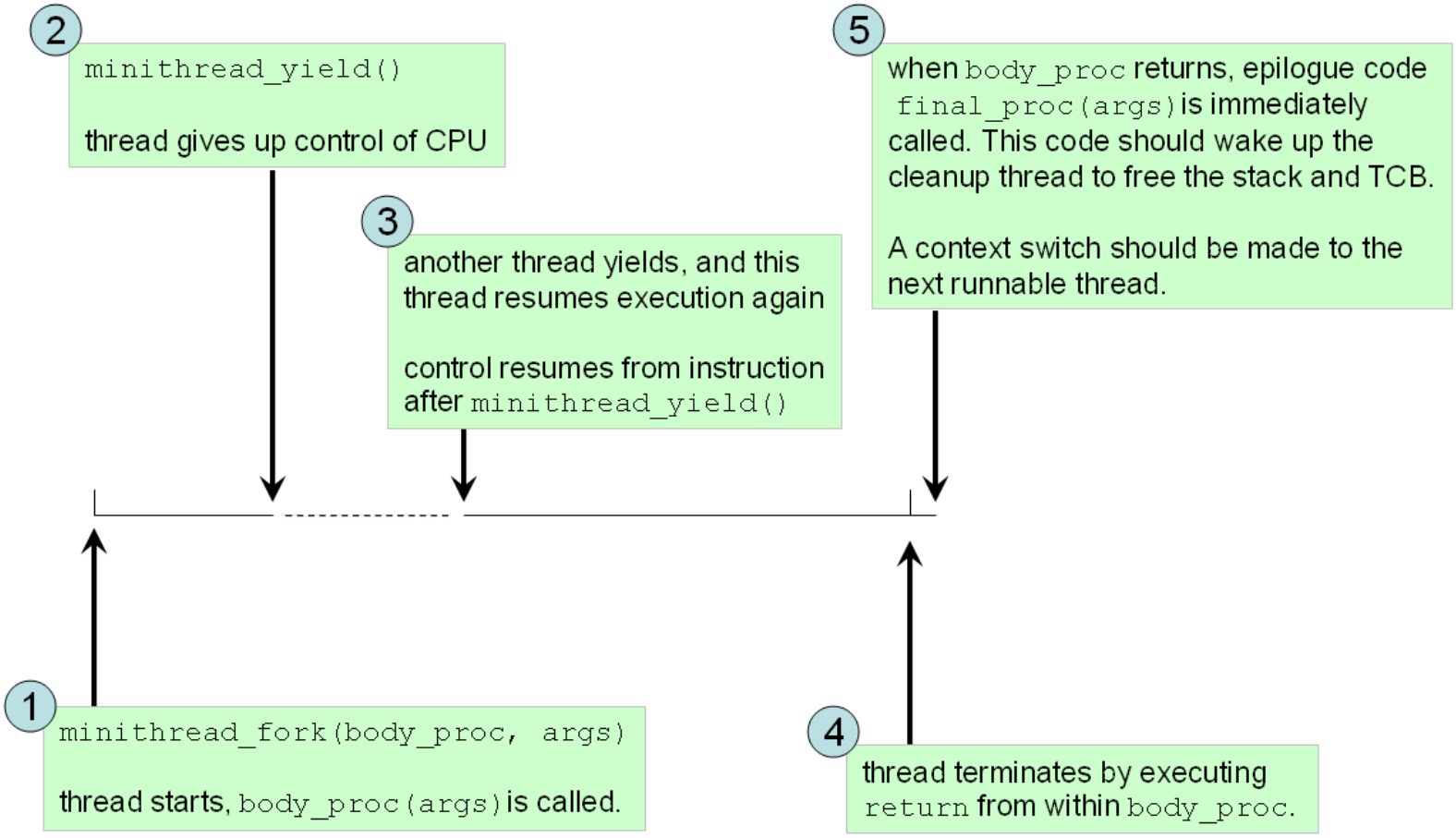
# `final_proc`

- `final_proc` is called immediately after a thread terminates (ie. by returning from its procedure).
- It is unsafe and unwise for a thread to free its own stack or TCB.
- Solution: dedicated cleanup thread.
  - It should **wait** for threads to be ready for cleanup; otherwise it should be blocked.
  - Use a semaphore to signal the cleanup thread that this TCB and stack is ready to be purged.

# Cleanup thread

- Should be blocked until there is a thread to clean up.
- Free up stack using: `minithread_free_stack(stack_pointer_t stack_base)`
  - Note that this takes the stack base, not the top of the stack.
- Don't forget to free up the TCB too!

# Summary of fork, run, yield, terminate and cleanup



# Initializing minithreads

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

- Starts up the system, and initializes global data structures.
- This should be where all queues, global semaphores, etc. are initialized.
- Creates a thread to run `mainproc(mainarg)`
- Creates the cleanup thread.
- You should not return from this function!

# What to do with the kernel thread given by Windows?

- You could create your threads and then context switch to them without ever coming back to the stack that was used for `minithread_system_initialize`, but this is wasteful.
- Suggestion: re-use this stack instead for the idle thread.
- What code should an idle thread run?

# How to re-use the original stack for the idle thread

- When `minithread_system_initialize` is called, create a TCB for the idle thread.
- In the TCB, set `stacktop` and `stackbase` to `NULL`.
  - Don't need stacktop because the stack is already initialized.
  - Don't need stackbase because the stack will never be freed (idle thread never terminates).
- You can now context switch into/out of your idle thread.
- Make sure the first context switch you make from `minithread_system_initialize` originates from the idle thread.

# A quick primer on concurrency bugs

- Data race/race condition: result of computation depends on relative running speeds of the threads.
  - Multiple concurrent threads reading/writing into the same memory location.
  - Example: two threads manipulating a linked list.
- Atomic operation: either the operation goes to completion, or fails to run altogether.
- Examples of concurrency bugs: Heisenbugs, crashes, mysterious behavior.

# Solution: synchronization

- Forces all concurrent threads to be well-behaved during important operations with shared data structures (critical section).
  - Example: disallow concurrent reads and writes.
- We want critical sections to be run without other threads interfering.
- Synchronization can be used to make critical section code appear atomic to other threads.
- Essentially enforces serial behavior in critical sections (ie, take turns to manipulate shared data).
- Caveat: synchronization can introduce other problems such as deadlock and starvation (will be discussed in 4410).

# Semaphores

- A synchronization primitive that is used to limit the number of threads accessing a shared resource.
- When initializing the semaphore, you decide how many threads can concurrently hold the semaphore.
- Semaphore value is manipulated atomically.
  - `semaphore_P` procures the semaphore and decreases the value by 1.
  - `semaphore_V` vacates the semaphore and increases the value by 1.
- Special case: a binary semaphore is also known as a mutex (mutual exclusion).

# Semaphore "P" behavior: procure

- When a thread requests to procure a semaphore and its current value is:
  - $>$ 0: value is decreased by 1, thread proceeds.
  - $\leq$ 0: value is decreased by 1, thread is blocked and waits in the semaphore queue.

# Semaphore "V" behavior: vacate

- When a thread vacates a semaphore and its current value is:
  - $\geq 0$: value is increased by 1 (note: this can go over the initial limit!)
  - $< 0$: value is increased by 1, first waiting thread in semaphore queue is unblocked (and is now runnable).
- Vacating a semaphore never blocks.

# Semaphore operations

`semaphore_t semaphore_create();` Create a semaphore (and allocate its resources).

`void semaphore_destroy(semaphore_t);` Destroy a semaphore (and `free` its resources).

`void semaphore_initialize(semaphore_t, int);` Set the initial value of a semaphore (how many `semaphore_P` functions may be called without blocking).

`void semaphore_P(semaphore_t);` Decrements a semaphore; (block if *value* $\leq 0$ before decrementing).

`void semaphore_V(semaphore_t);` Increments a semaphore, unblocking a thread that is blocked on it.

# Semaphore implementation hints

- You need:
  - A variable to keep track of the value.
  - A queue to keep track of threads blocked on the semaphore.
- When a procure operation causes a thread to block, move that thread to the semaphore queue and context switch to another thread.
- When a vacate operation unblocks a thread, the unblocked thread should be enqueued to the ready queue.†

† why should it not run immediately?

# Submitting your work

- Include a `README` file with your names and net IDs.
- Write **short** notes about anything you think we should know (e.g. broken code).
- This `README` should be nearly empty as all of your code should work and be well-tested.
- A clean compile (no errors) will be worth 10%.

# Mistakes to avoid

- Scheduler:
  - Don't schedule the idle thread unless there is really nothing else to do.
  - Idle thread is not supposed to return.
  - Don't call cleanup thread unnecessarily.
  - Cleanup must free both stack and TCB.
  - Don't exit or return from `minithread_system_initialize`.

# Mistakes to avoid

- Semaphores:
  - Don’t use a global queue to track all blocked threads.
  - Don’t copy Wikipedia’s implementation!
  - Don’t rely on counting the blocked queue length to deduce the semaphore value.

# Concluding Advice

- Manage your memory and pointers, for they are the key to bug-free code.
- Write clean and understandable code.
  - Variables should have proper names (e.g. `stack_pointer` not `lol`)
  - Provide meaningful comments (but do not comment in excess).
- Do not terminate when program threads are done.
  - Idle threads never terminate.
- E-mail course staff if you have any questions.
- Good luck!