CS414 Section 1
Project 1: Minithreads

Bernard Wong
bwong@cs.cornell.edu

Slides modified from Adrian Bozdog’s slides

What are minithreads?

- User-level thread package for Windows NT/2000/XP
  - Windows only comes with kernel-level threads, but user-level threads are better in some cases because of its low overhead
- Real motivation?
  - We want you to learn how threading and scheduling works

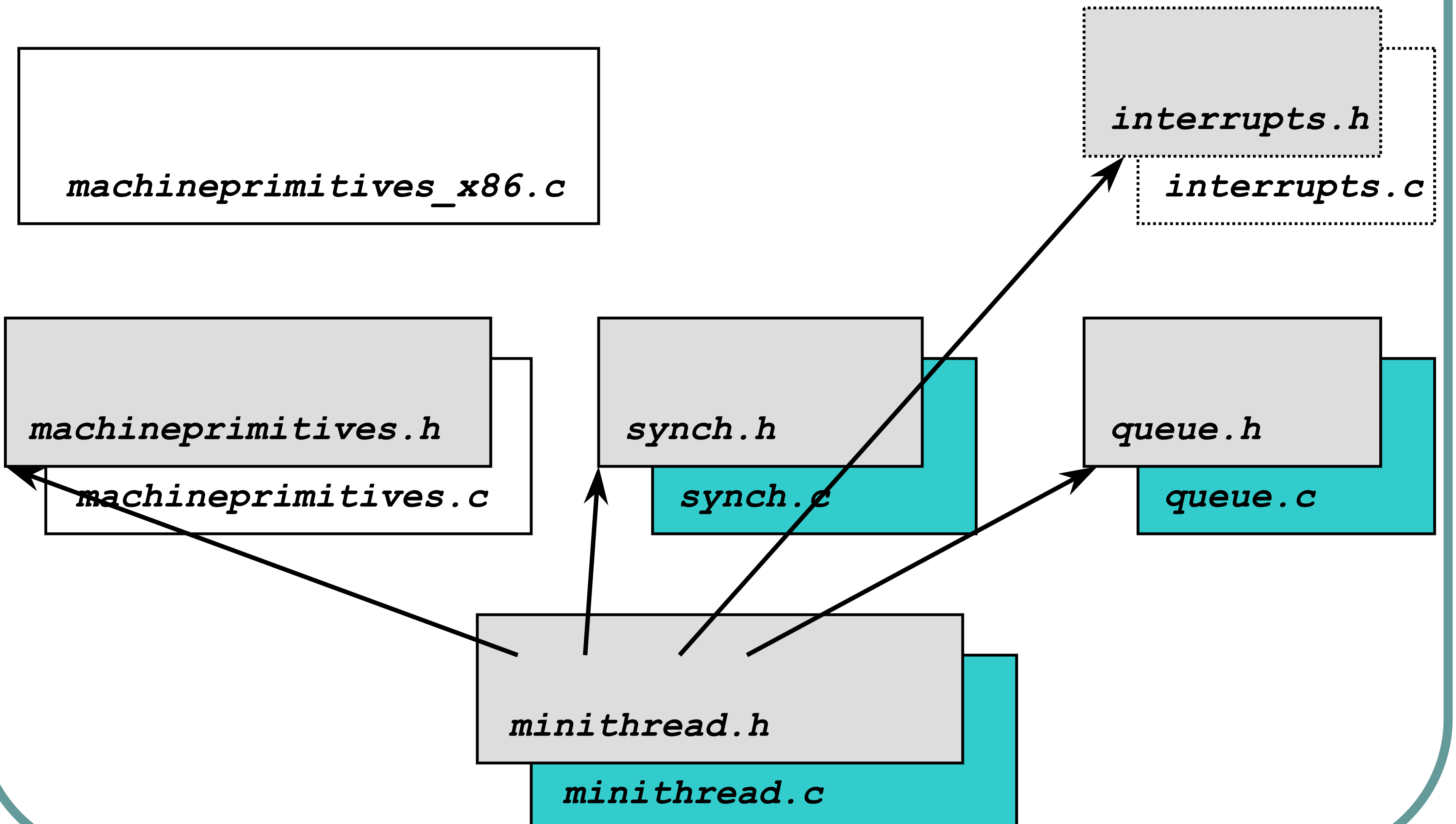
What do I have to do?

- Implement minithreads of course!
- Requires the following parts:
  - FIFO Queue
    - O(1) enqueue and dequeue
  - Non-preemptive threads and FCFS scheduler
  - Semaphore
    - Threads not very useful if they can't work together
  - Simple application – "Food services" problem
- Optional:
  - Add preemption, not covered today
  - Optional material not graded

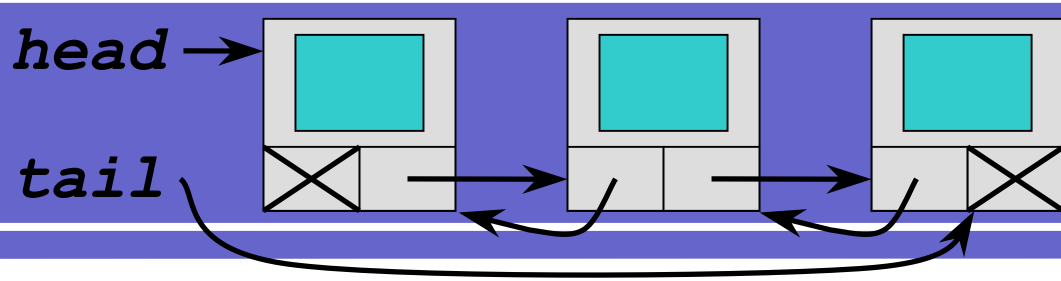
What do we give you?

- Interfaces for the queue, minithread, and semaphore
- Machine specific parts
  - i.e. context switching, stack initialization
- Simple test applications
  - Not exhaustive tests!
  - Write you own test programs to verify the correctness of your code.

Minithreads structure



Queues



- Singly or doubly link list are both fine and can satisfy the O(1) requirements
- Queue must be able to hold arbitrary data
  - Take any_t as queue_append and queue_prepend argument
  - any_t really just a void*
- Note that queue_dequeue takes any_t* as its second argument
  - Why? Remember that C is call by value
    - If you want the any_t variable in your calling function to point to the where the item you just dequeued points to, you must pass the address of your any_t pointer to the queue_dequeue function.
  - Your queue_dequeue function must dereference the any_t* argument before assigning it the value it just dequeued.

Example of using queue_dequeue

- In the calling function:

```
any_t datum = NULL;
queue_dequeue(run_queue, &datum);
/* You should check the return value in your code */
```

- In queue_dequeue function:

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

Minithread structure

- Need to create a Thread Control Block (TCB) for each thread
- Things that must be in a TCB:
  - Stack top pointer
  - Stack base pointer
    - i.e. where the stack start in memory
  - Thread identifier
  - Anything else you think might be useful

Minithread operations to implement

```
minithread_t minithread_fork(proc, arg)
```
create thread and make it runnable

```
minithread_t minithread_create(proc, arg)
```
create a thread but don’t make it runnable

```
void minithread_yield()
```
Let another thread in the ready queue run
(make the scheduling decisions here)

```
void minithread_start(minithread_t t)
void minithread_stop()
```
start another thread, stop yourself

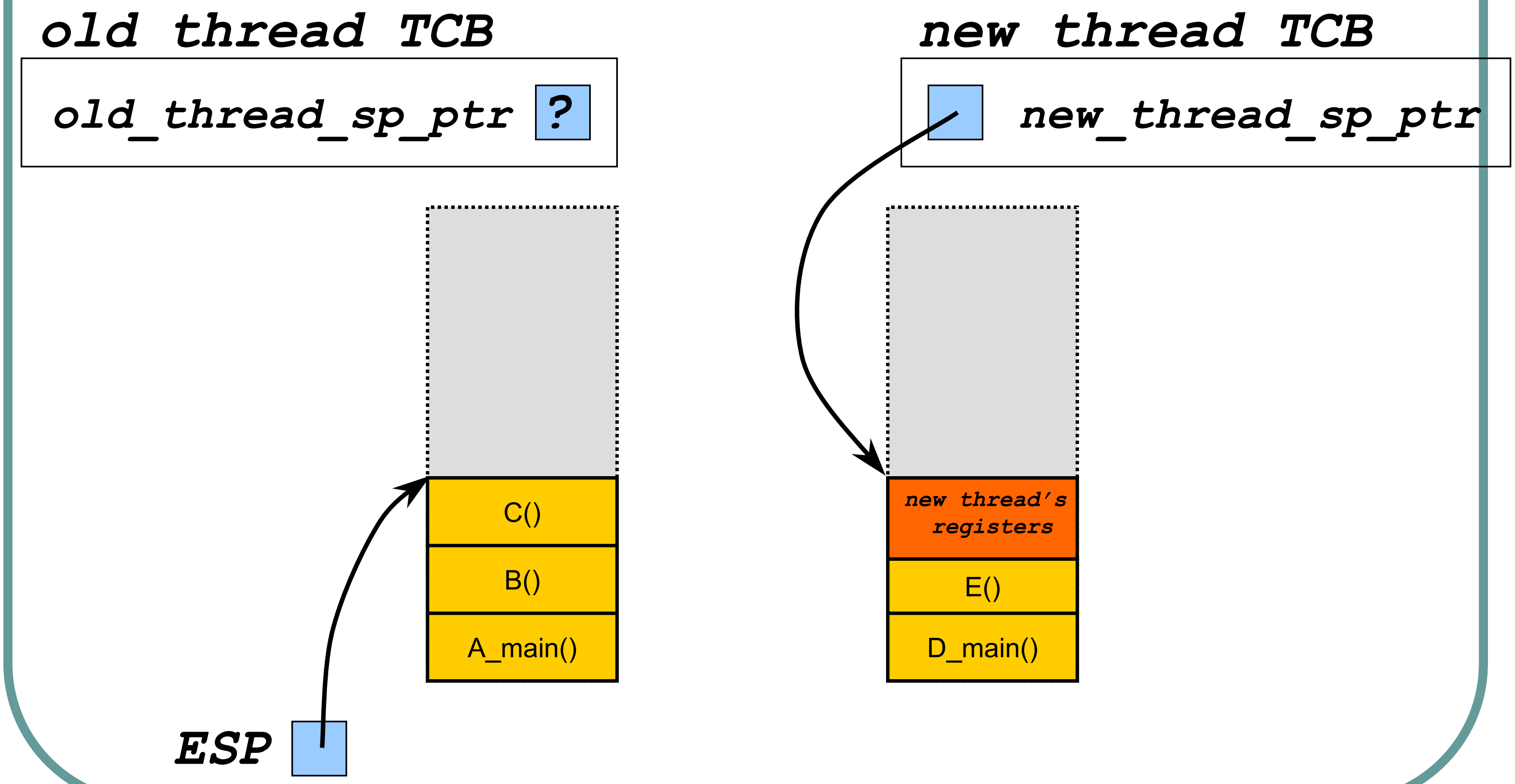
Context switching

- Swap current execution contexts with a thread from the ready queue (a queue that holds all your ready to run processes)
  - Registers
  - Program counter
  - Stack pointer

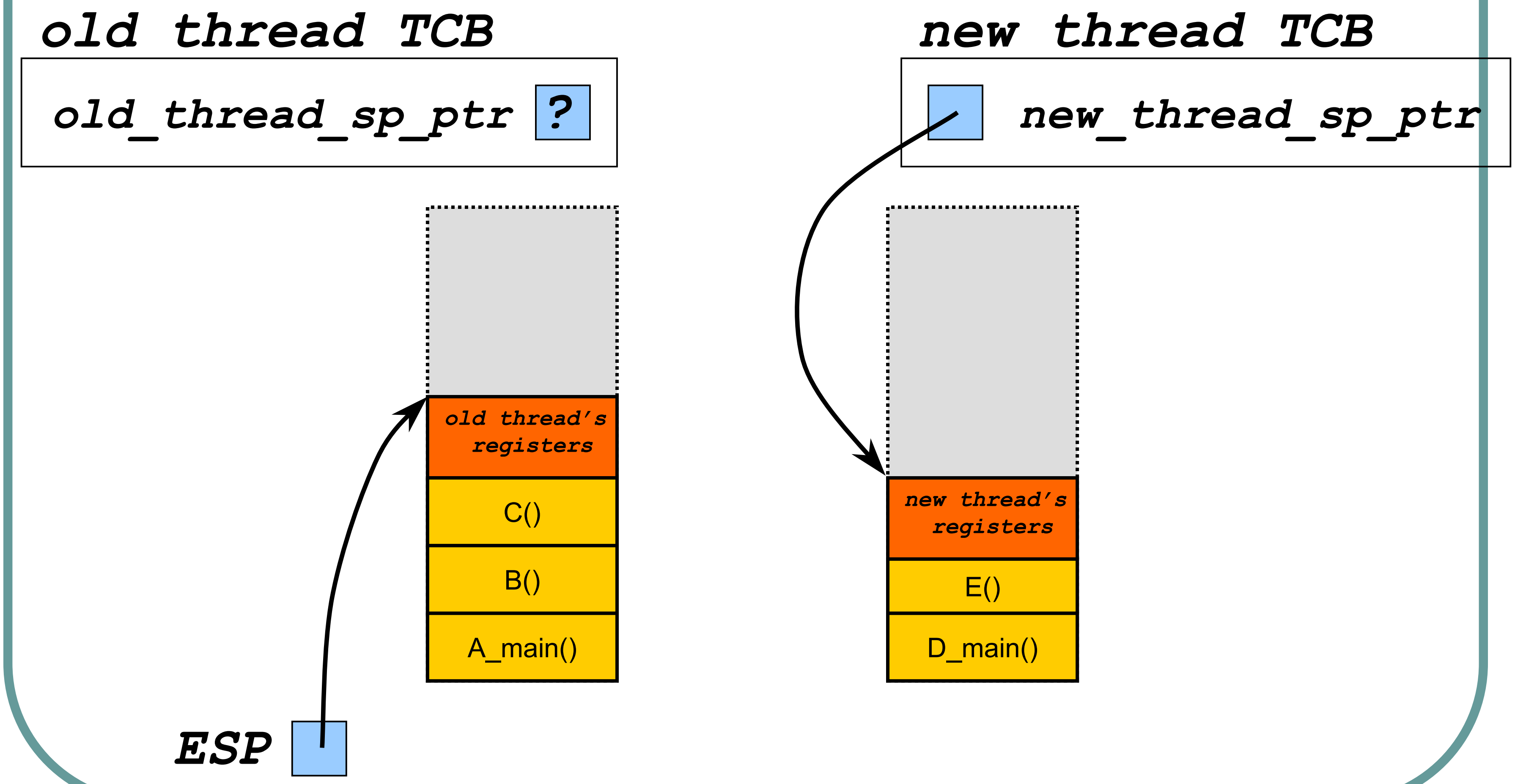
- **`minithread_switch(old_thread_sp_ptr, new_thread_sp_ptr)`** is provided

- So how does context switching work?

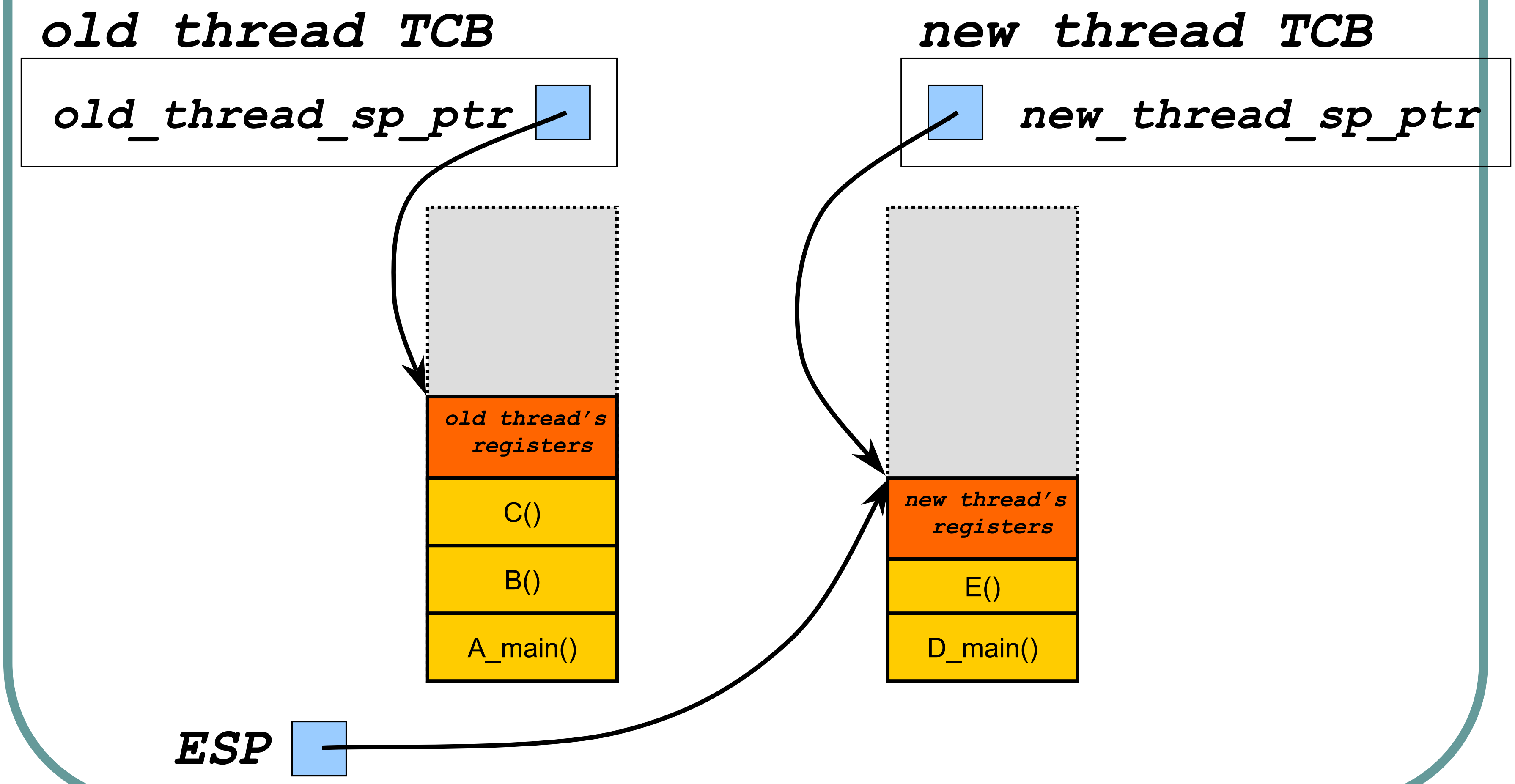
Before context switch starts



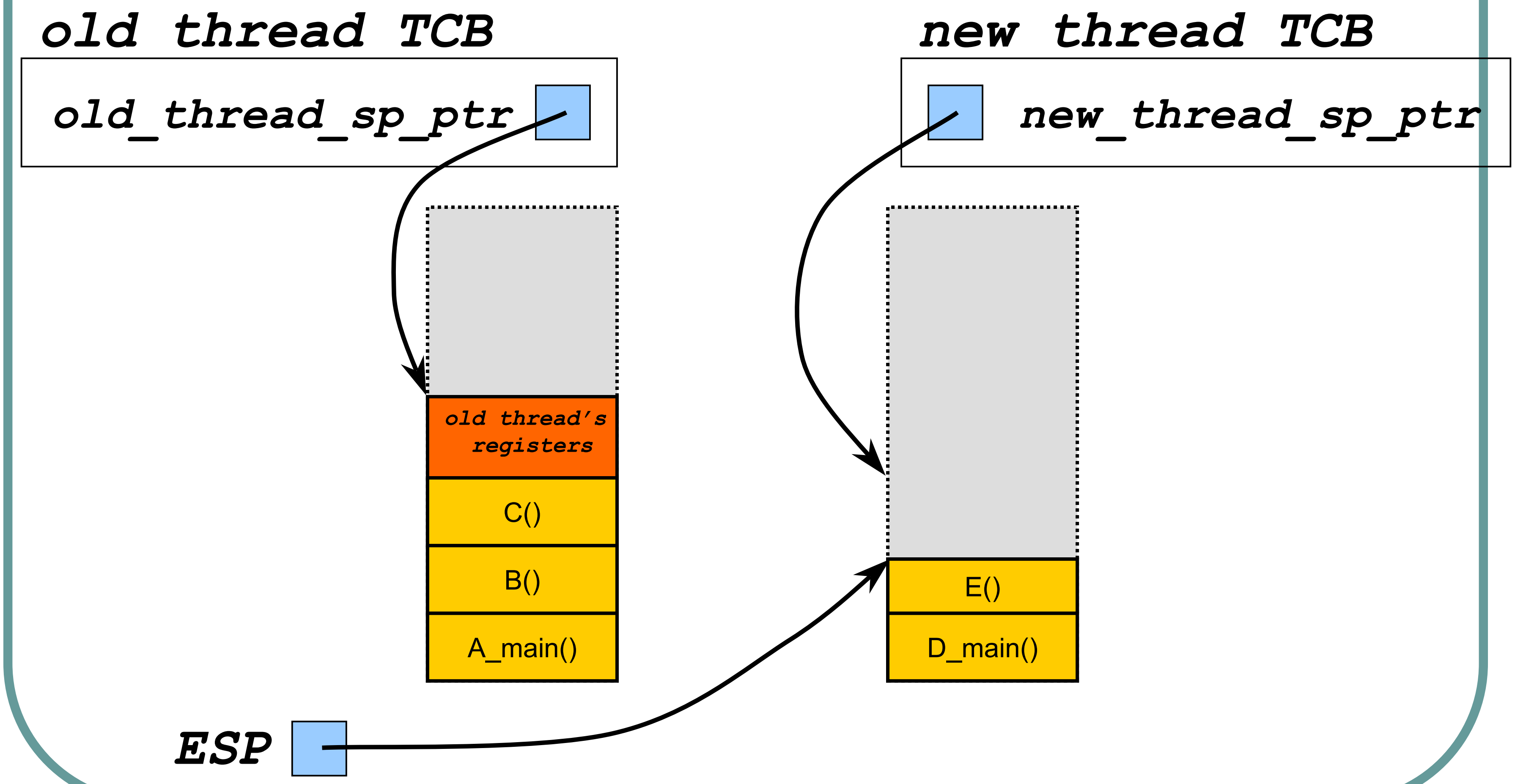
Push on old context



Change stack pointers



Pop off new context



Minithread Creation

- Two methods to choose from

  ```
  minithread_create(proc, arg)
  minithread_fork(proc, arg)
  ```

- **`proc`** is a **`proc_t`** (a function pointer)
  - `typedef int (*proc_t)(arg_t)`
  - e.g. `int run_this_proc(int* x)`

- **`arg_t`** is actually an **`int*`**, but you can cast any pointer to it.

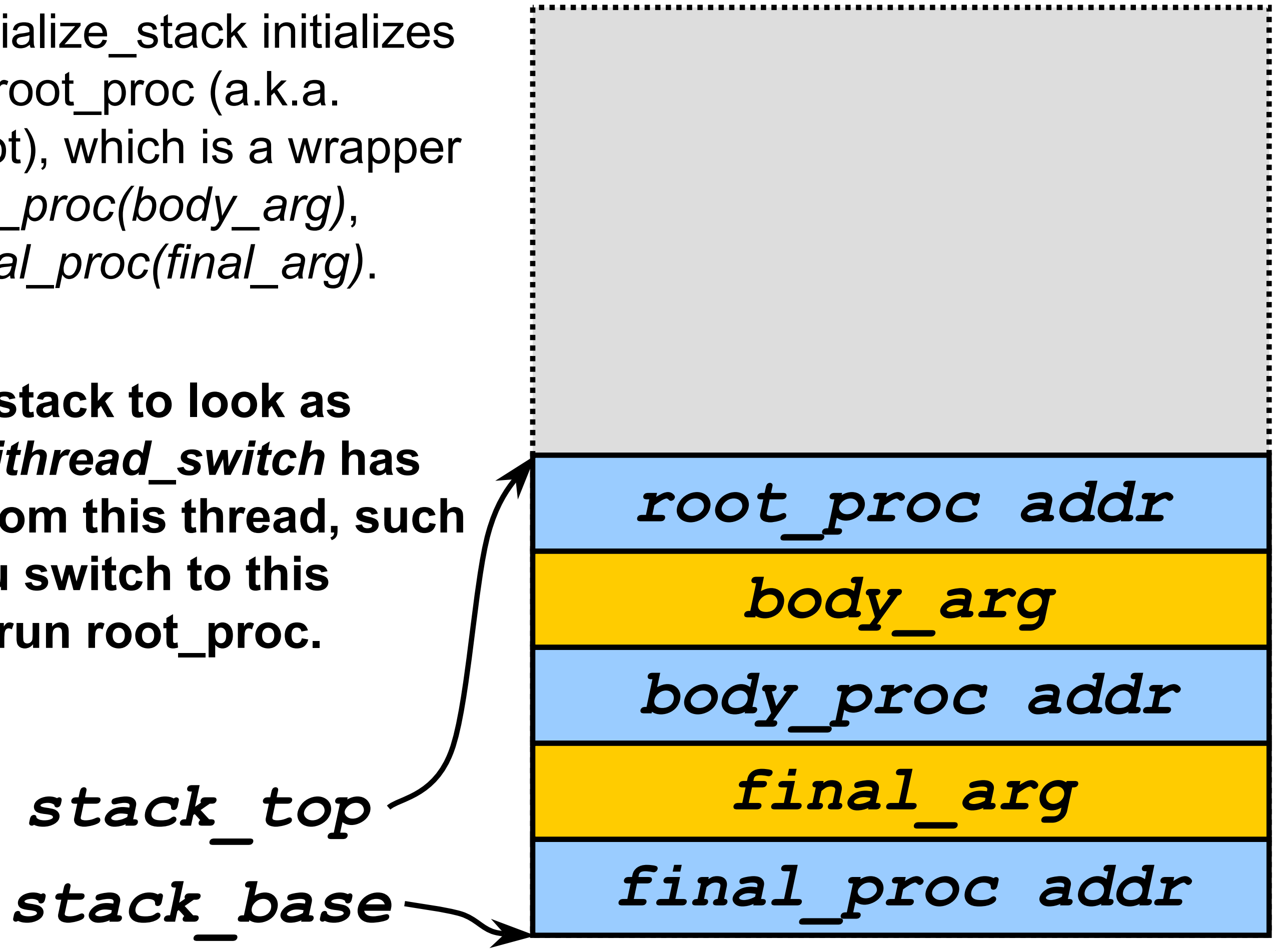
Minithread Creation

- For each thread, you must allocate a stack for it and initialize the stack
  - **`minithread_allocate_stack(stackbase, stacktop)`**
  - **`minithread_initialize_stack(stacktop, body_proc, body_arg, final_proc, final_arg)`**
- The implementation of allocate and initialize stack are given to you.

Minithread Creation

minithread_initialize_stack initializes the stack with root_proc (a.k.a. minithread_root), which is a wrapper that calls *body_proc(body_arg)*, followed by *final_proc(final_arg)*.

Sets up your stack to look as though a *minithread_switch* has been called from this thread, such that when you switch to this thread, it will run root_proc.



Minithread Creation

- What's final_proc for?
  - Thread cleanup
    - You will want to free up resources such as TCB and stack allocation after your thread terminates (or else your program will run out of memory like certain OS-es….)
- But can a thread cleanup after itself?
  - No, not directly, not safe for a thread to free it's own stack.
- Solution?
  - Dedicated cleanup thread
    - Should only run if there are threads to clean up though, otherwise, otherwise it should be blocked.

Yielding a thread

- Because our threads are non-preemptive, we need a user level way of initiating a switch between threads
  - Thus: minithread_yield
- Use **`minithread_switch`** to implement **`minithread_yield`**
- Where does a yielding thread go?
  - Into the ready queue, so it can be re-scheduled later

Initializing the system

- **`minithreads_system_initialize (proc_t mainproc,arg_t mainarg)`**
- Starts up the system
- First user thread runs

  `mainproc(mainarg)`
- Should probably create any additional threads (idle, cleanup, etc.), queues, semaphores, and any other global structures at this point.

What about the Windows thread?

- Windows gives me an initial (kernel) thread and stack to work with, can I re-use that for one of my threads?
  - Yes, and you should as you don't really want to throw away memory for no reason.
  - But be careful, make sure this thread never exits or gets cleaned up.
- Remember, your threaded program never really exits, as the idle thread will always keep running.
  - May want to re-use the initial Windows thread as the idle thread because of this property.

Semaphores

- semaphore_t semaphore_create();
  - Creates a semaphore (allocating resources for it)
- void semaphore_destroy(semaphore_t sem);
  - destroys a semaphore (freeing resources for it)
- void semaphore_initialize(semaphore_t sem, int cnt);
  - Initializes semaphore to an initial value
  - i.e. Determines how many more semaphore_P functions can be called than semaphore_V before a semaphore_P will block
- void semaphore_P(semaphore_t sem);
  - Decrements on semaphore, must block if semaphore value less than or equal to 0.
- void semaphore_V(semaphore_t sem);
  - Increments on semaphore, must unblock a thread that's blocked on it.

Properties of Semaphores

- Value of semaphore manipulated atomically through V and P
- Without preemption, trivial to implement
  - i.e. Just don’t have a minithread_yield in semaphore_P and semaphore_V
- With preemption, requires mutual exclusion around instructions that change the variable value
  - i.e. test_and_set on a lock variable
  - We’ll covered this in the next section

Properties of Semaphores

- Thread waiting to get a semaphore (i.e. after calling a semaphore_P with the semaphore value less than or equal to 0) must block on the semaphore
  - Each semaphore should therefore have a blocked thread queue
- After calling a semaphore_V, a thread waiting on that semaphore must unblock and be made runnable.

Concluding remarks

- Watch out for memory leaks
- Write a clean and understandable code
  - Variables should have proper names
  - Provide meaningful but not excessive comments
  - Don't make us guess at what you wrote, the project is simple enough that we should be able to understand what you are doing at a glance
  - Do not terminate when your user program threads are done
    - Remember that the idle thread should never terminate
- Due Date : Monday, February 13