

CS414
Minithreads project
overview

Ben Atkin

`batkin@cs.cornell.edu`

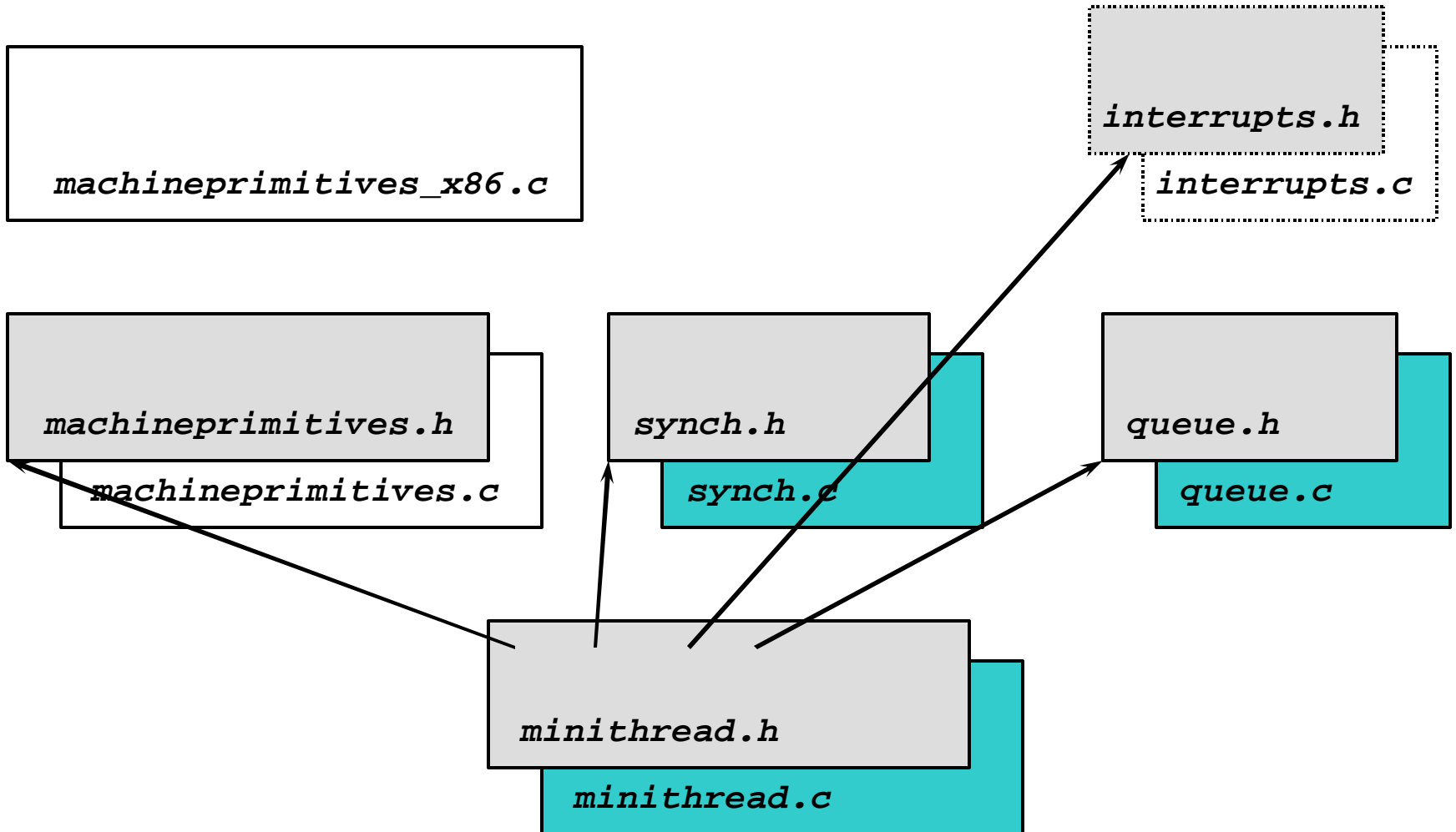
What you have to do

- Implement Minithreads, a user-level threads package on Windows NT
- Includes semaphores and queues
- Non-preemptive
- The hardest parts (context switching, stack initialisation) are done for you
- For the adventurous: add preemption

What we'll cover

- What order to do things in
- How context switching works
- How yielding between threads works
- Minithread implementation hints

Minithreads structure

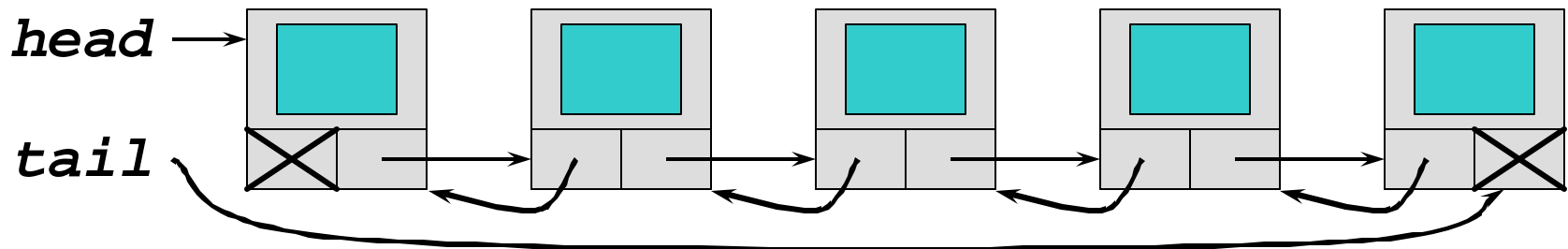


Minithreads, step-by-step

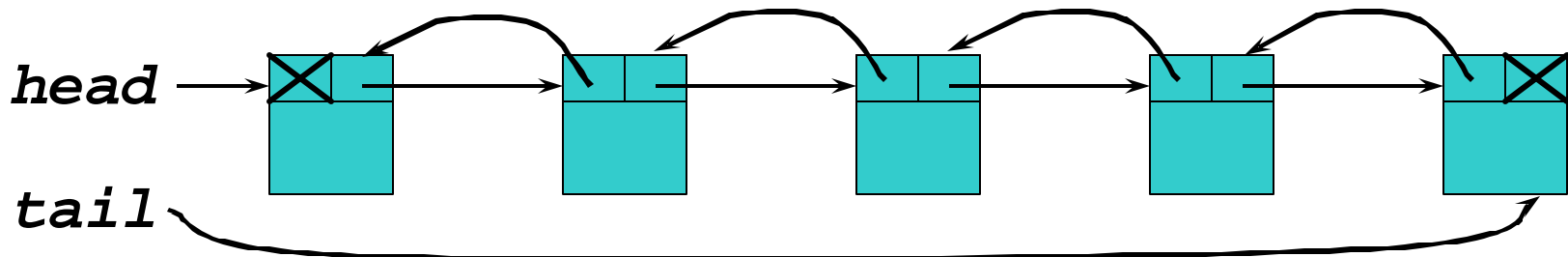
- Implement queues
- Define **struct minithread**
- Implement minithreads operations
 - fork and yield
 - system initialisation
 - termination and cleanup
 - start, stop
- Implement semaphores

Queue alternatives

- Implement by using enclosing structs:



- Or implement by using pointers in element types:



Defining a minithread

- What's in a **struct minithread** (thread control block)?
 - stack top pointer
 - stack base pointer
 - numerical identifier (**int**)
 - thread status
 - anything else you think necessary

Minithread operations

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()

stop this thread and run a new one from the run queue
(make the scheduling decisions here)

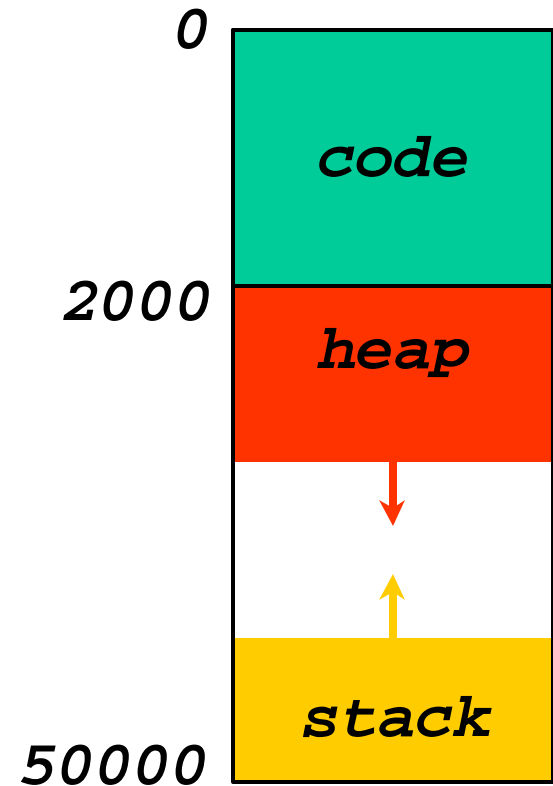
void minithread_start(minithread_t t)

void minithread_stop()

start another thread, stop yourself

Threads and their stacks

- NT gives you an initial stack
- Subsequent minithread stacks are allocated on the process's heap using **malloc**



Context switching

- `minithread_switch(old_thread_sp_ptr, new_thread_sp_ptr)` is provided
- Swap execution contexts with a thread from the run queue
 - registers
 - program counter
 - stack pointer

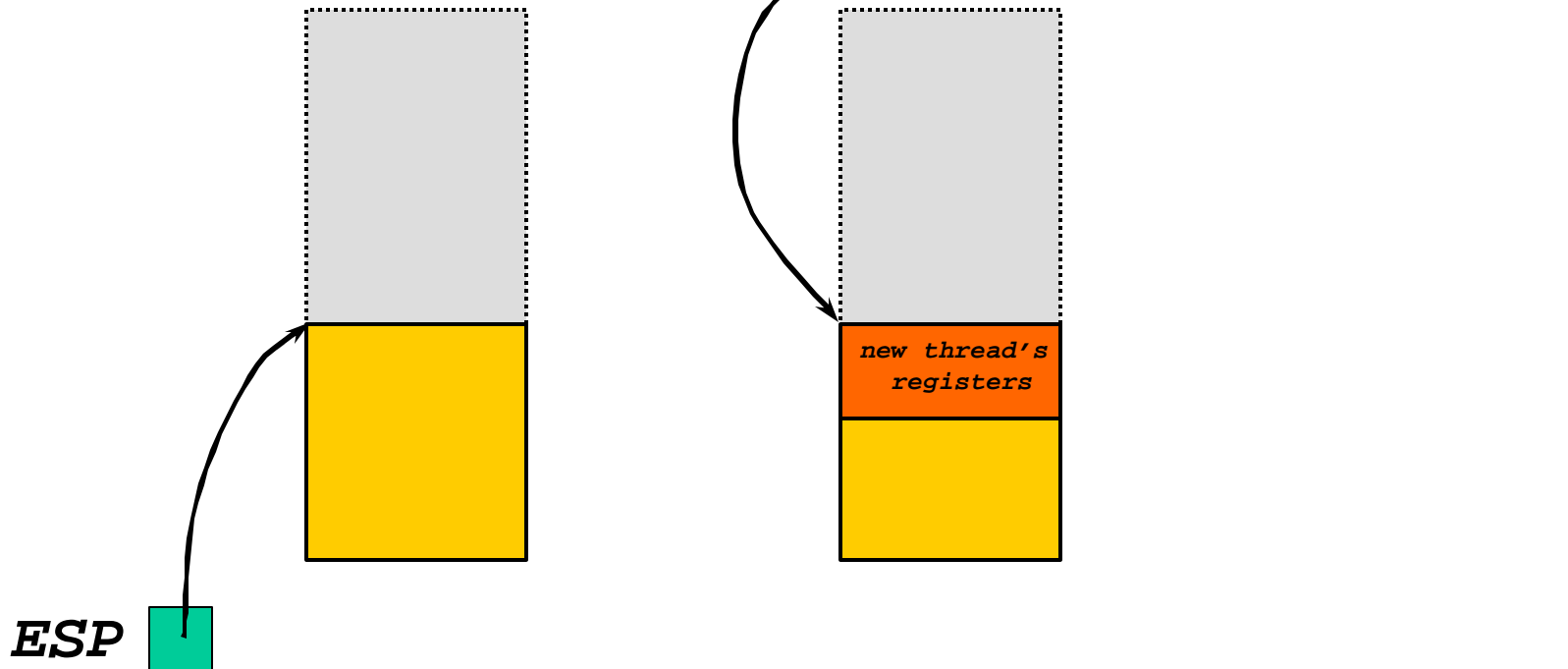
Context switching

old thread TCB

old_thread_sp_ptr 

new thread TCB

 *new_thread_sp_ptr*



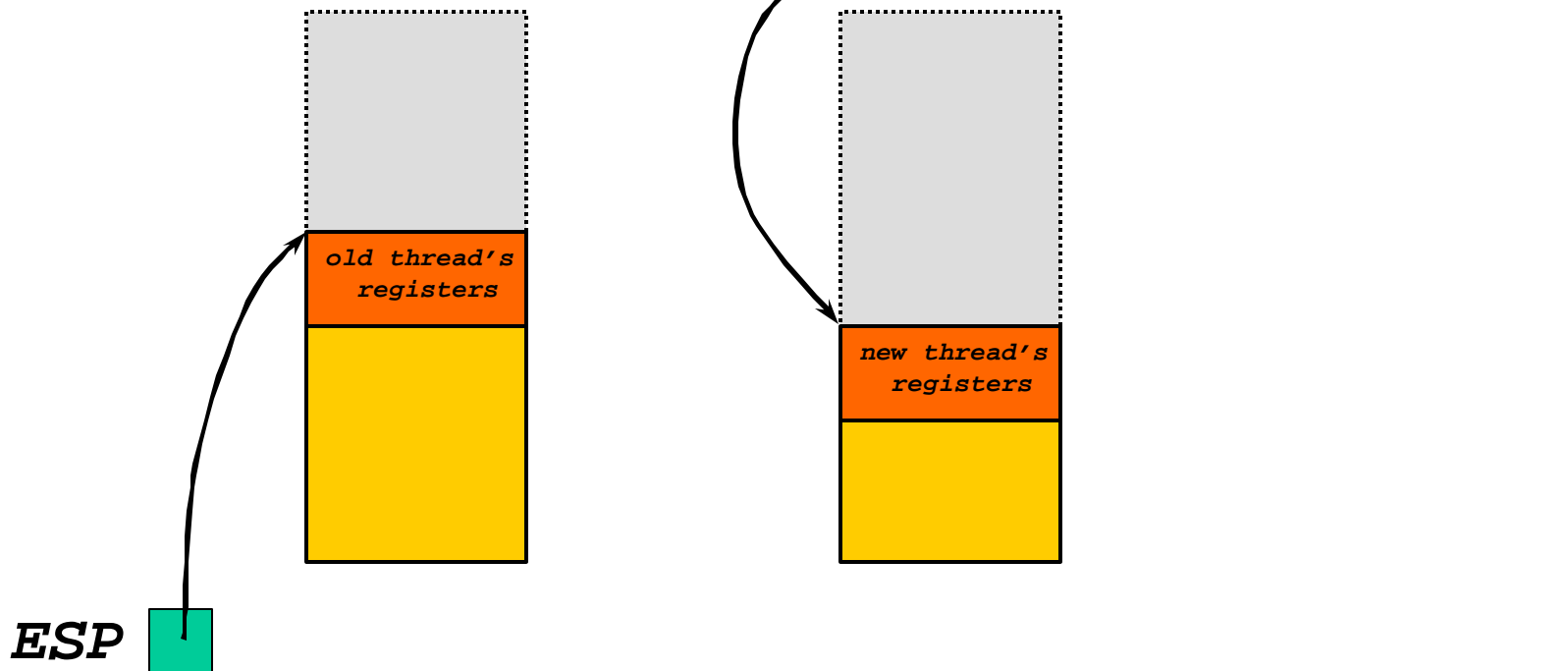
Push on old context

old thread TCB

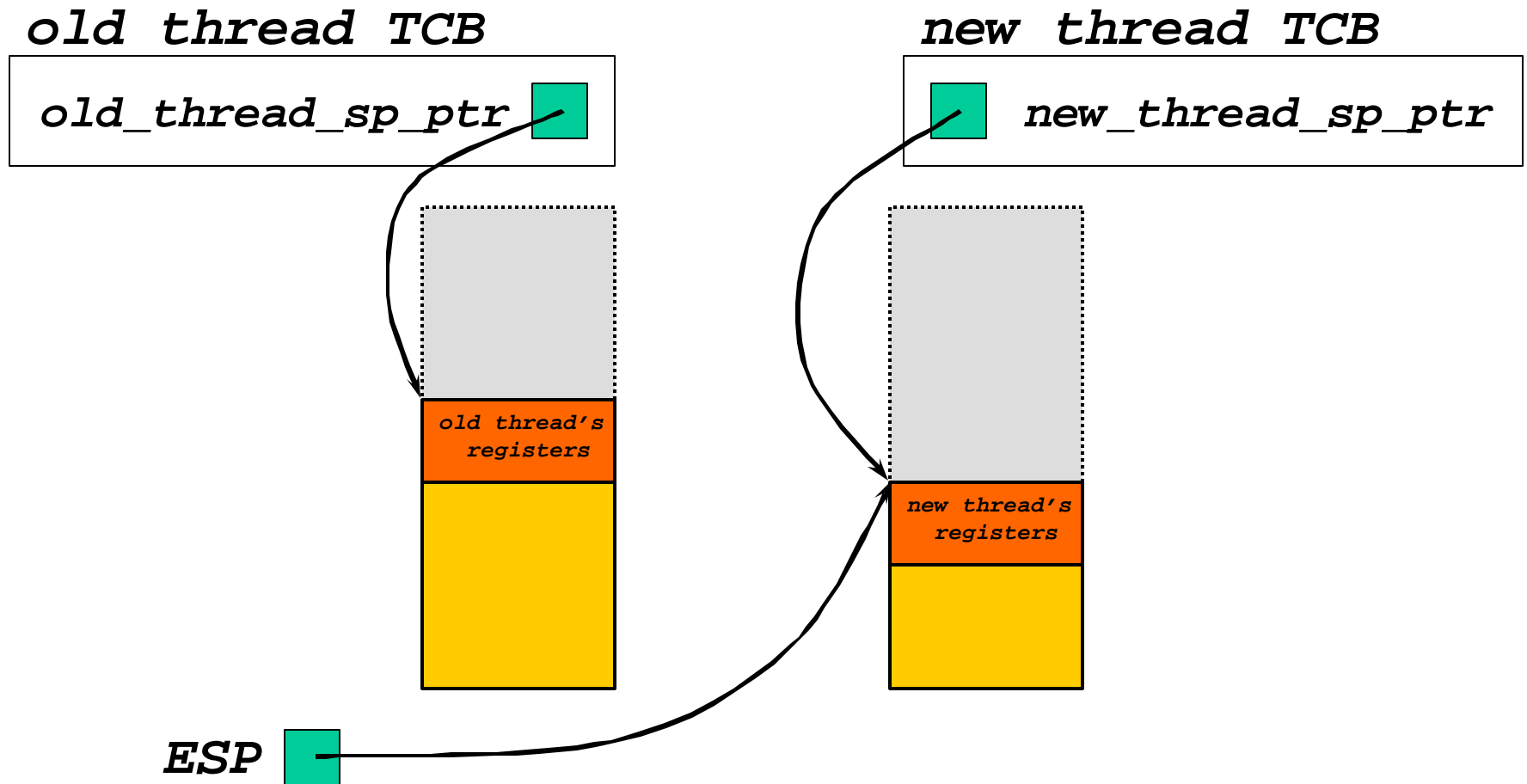
old_thread_sp_ptr 

new thread TCB

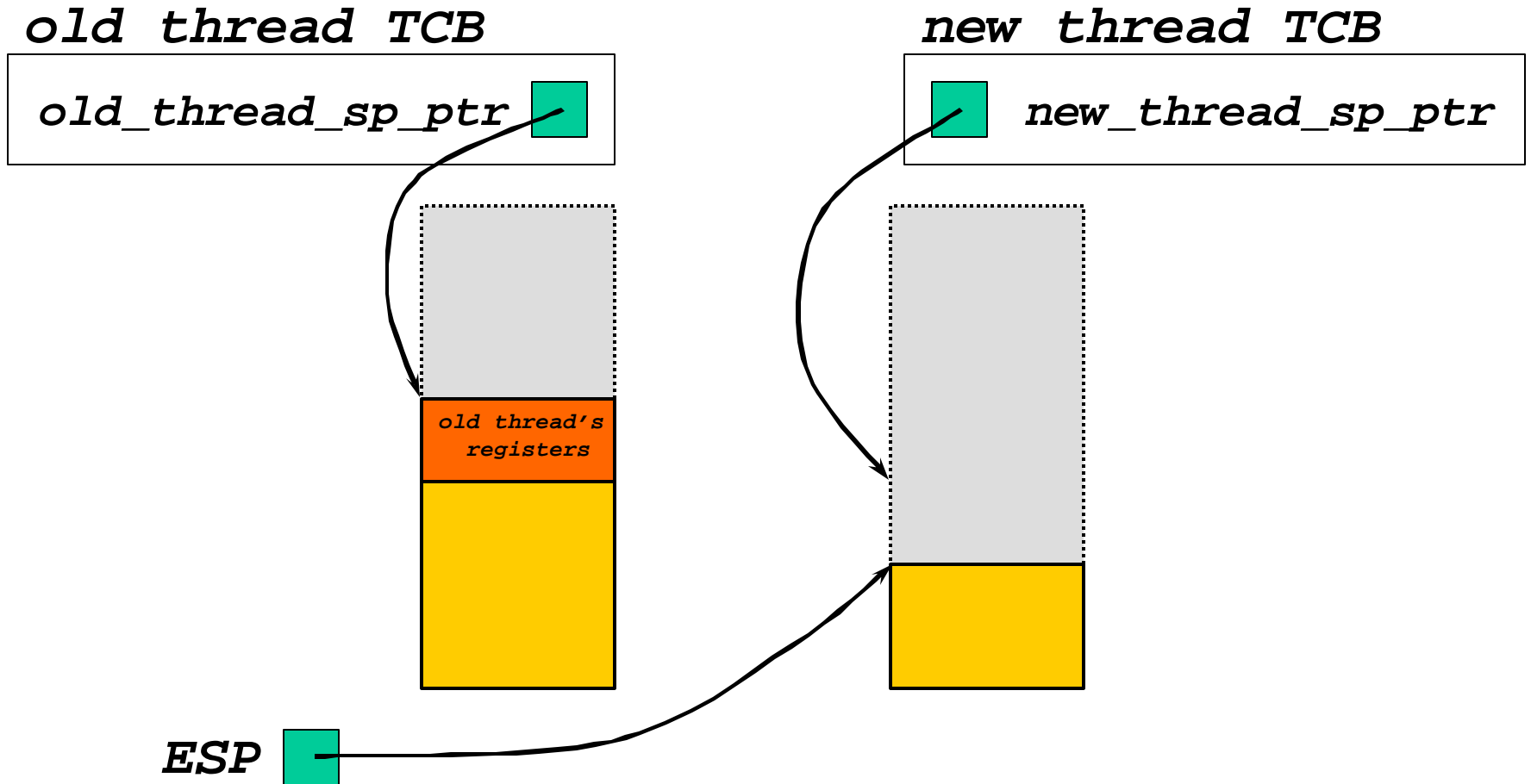
 *new_thread_sp_ptr*



Change stack pointers



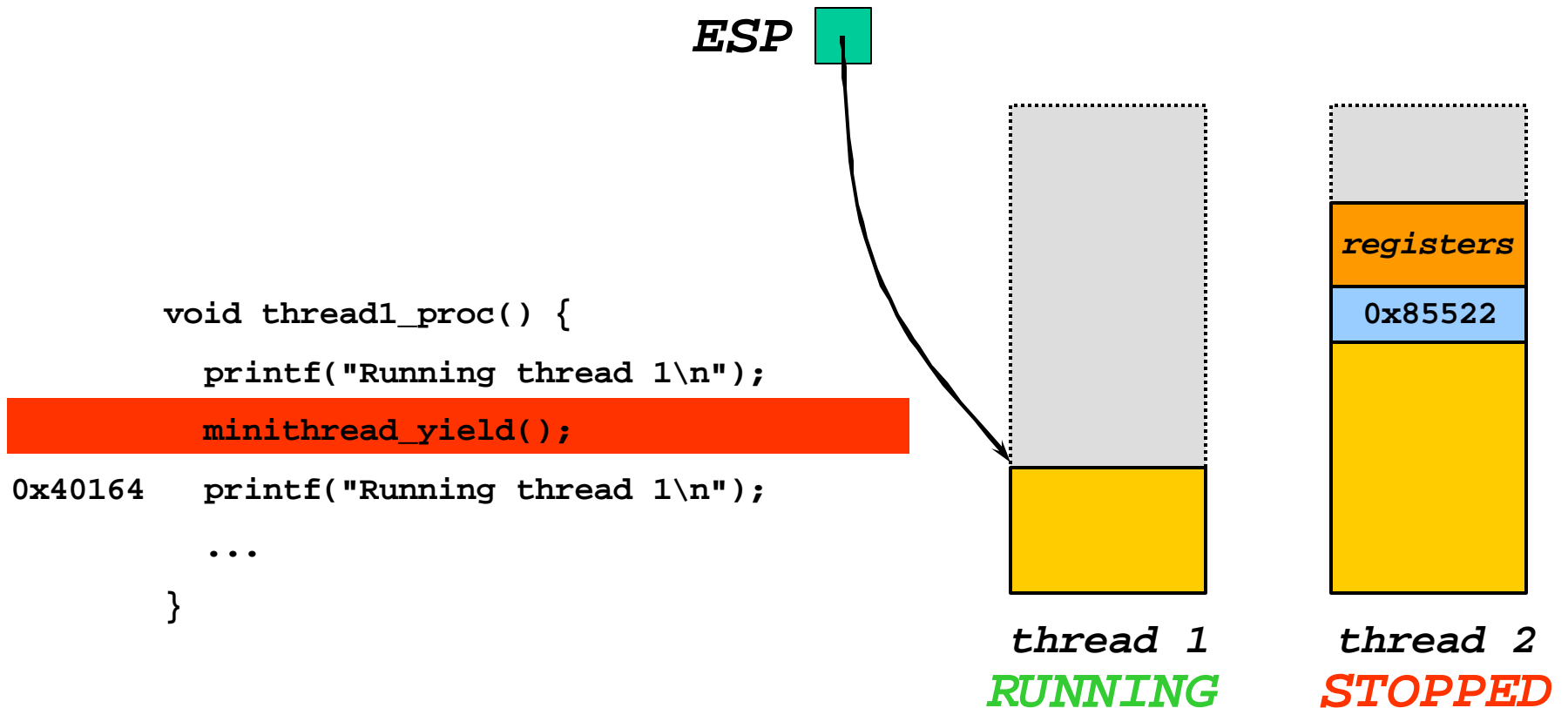
Pop off new context



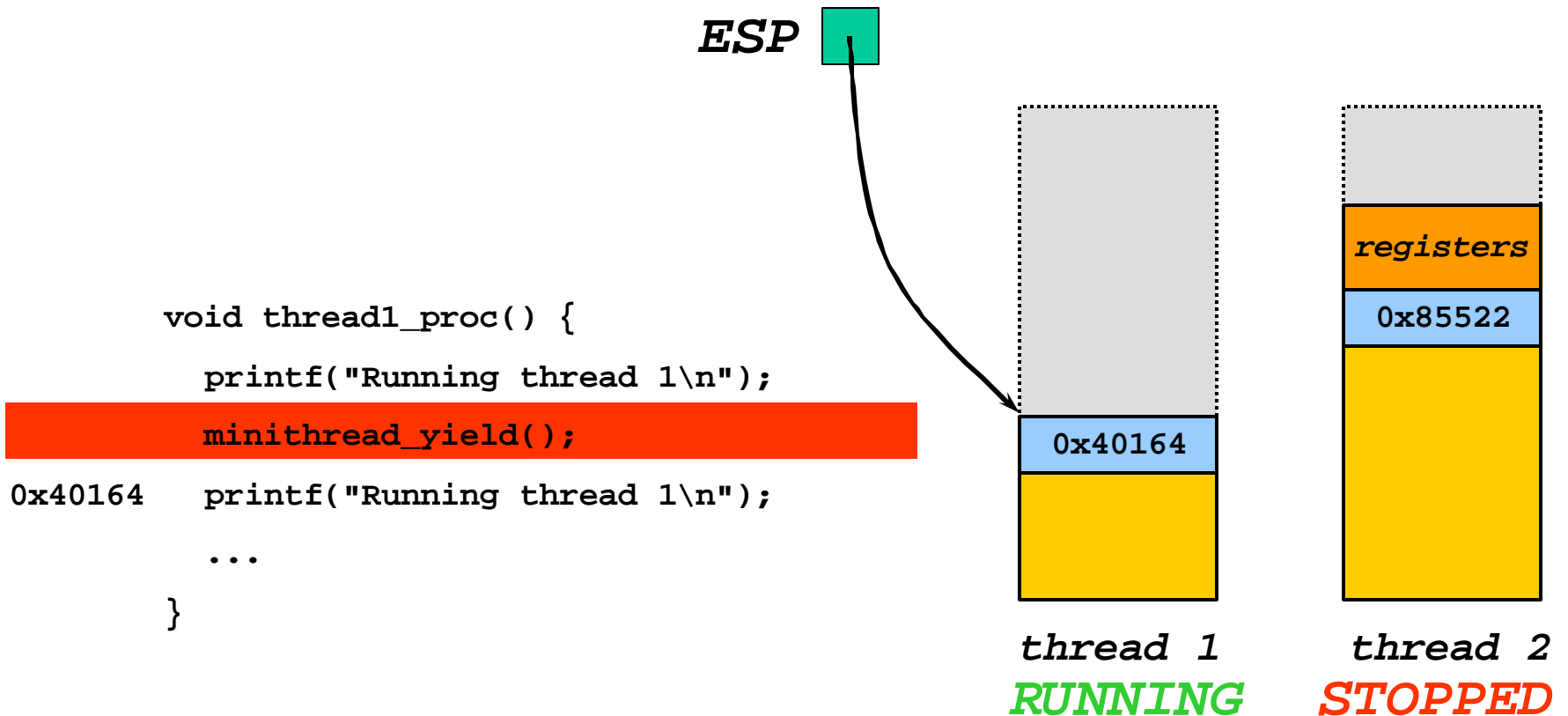
Thread yield

- Use `minithread_switch` to implement `minithread_yield`
- What does a yield do?
- Where does a yielding thread return to when it's rescheduled?

Thread yield



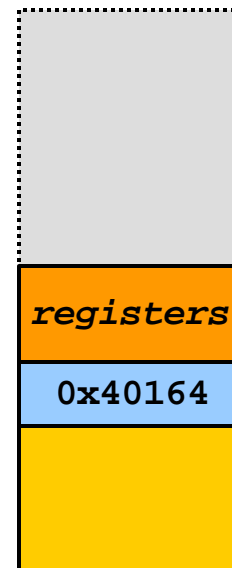
Push return address and call yield



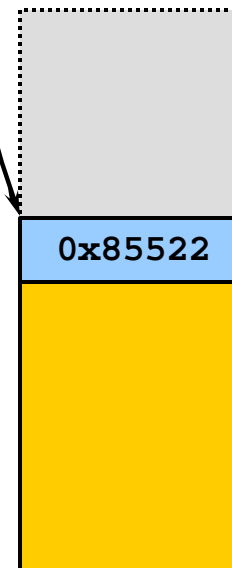
Switch to new thread

```
void thread2_proc() {  
    int x;  
    for (;;) {  
        minithread_yield();  
        printf("x is now %d.\n", x++);  
    }  
}
```

ESP



thread 1
STOPPED



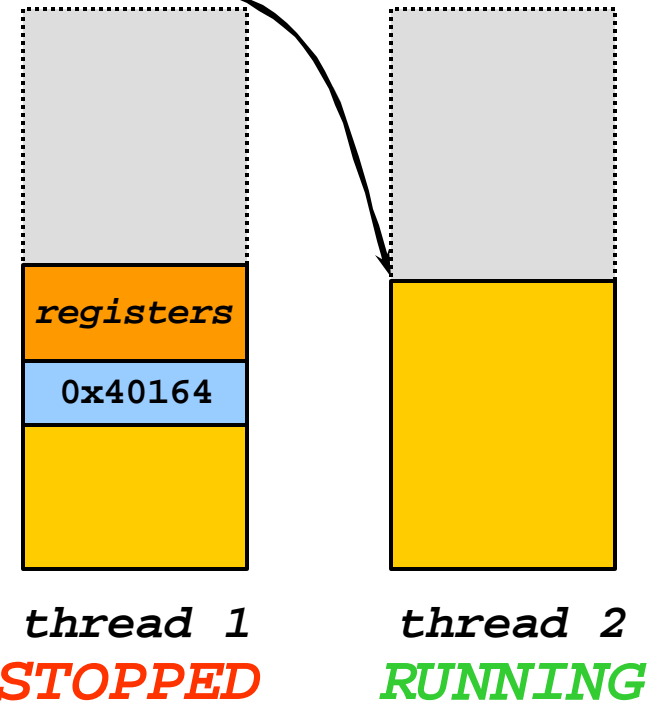
thread 2
RUNNING

Return from yield into new context

ESP

```
void thread2_proc() {  
    int x;  
    for (;;) {  
        minithread_yield();  
        printf("x is now %d.\n", x++);  
    }  
}
```

0x85522



Implementation details

- How do we switch to a newly-created thread?
- Where do the stacks come from?
- How do we create a thread?
- How do we initialise the system?

Minithread creation

- Two methods to choose from
 - `minithread_create(proc, arg)`
 - `minithread_fork(proc, arg)`
- `proc` is a `proc_t`
 - `typedef int (*proc_t)(arg_t)`
 - e.g. `int run_this_proc(int* x)`
 - could cast any pointer to `(int *)`

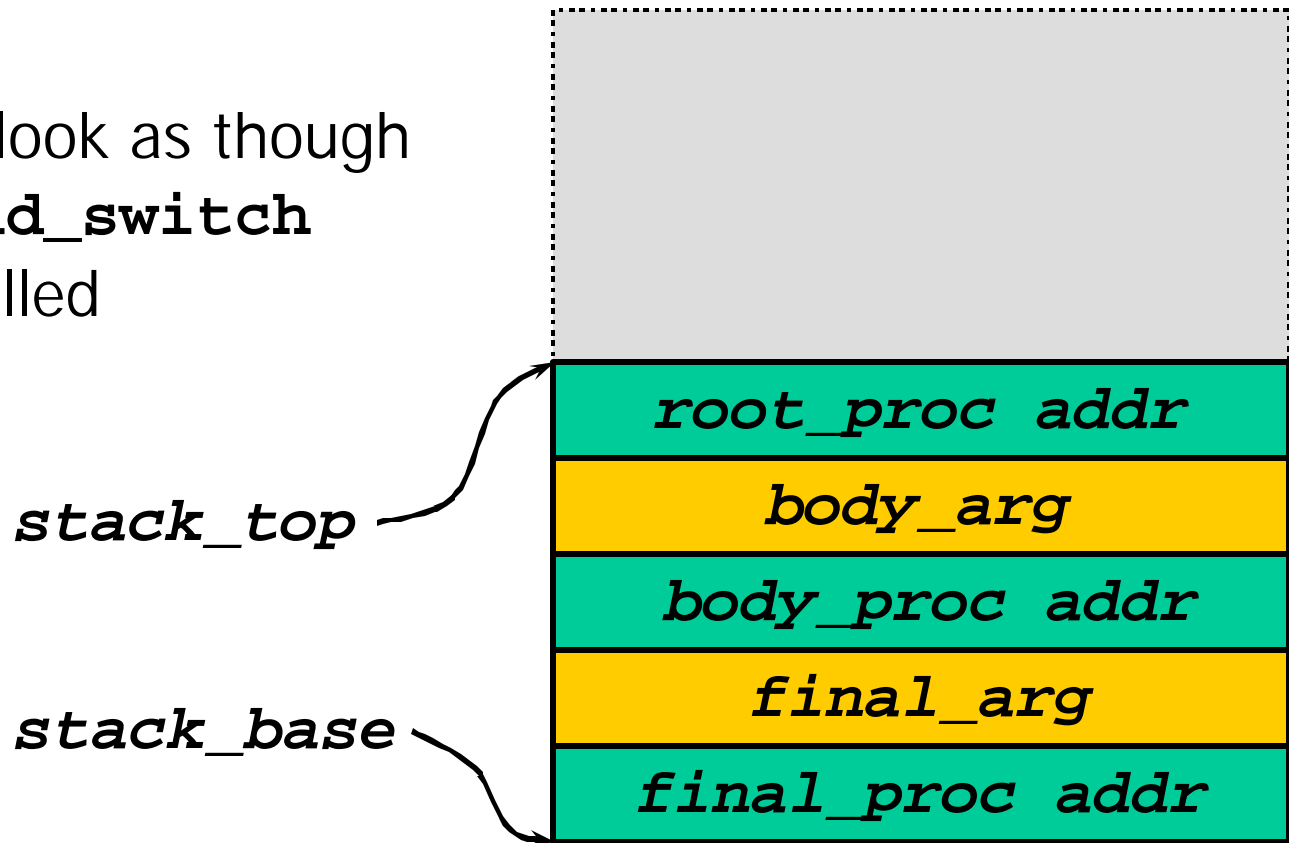
Minithread creation

- Allocate a **struct minithread** (TCB)
- Allocate and initialise a new stack

```
minithread_allocate_stack(stackbase,  
stacktop)  
  
minithread_initialize_stack(stacktop,  
body_proc, body_arg,  
final_proc, final_arg)
```
- Set the initial thread status
- Whatever else is appropriate

An initialised stack

Stack must look as though `minithread_switch` has been called

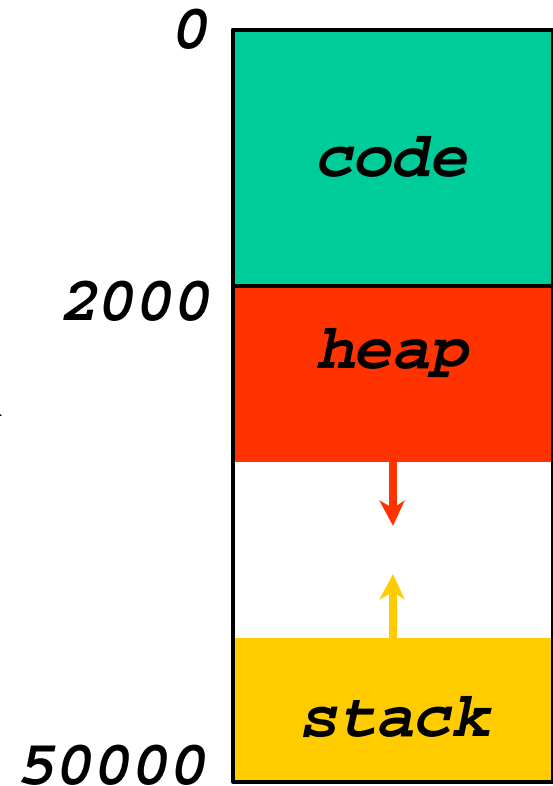


How a new thread starts

- **root_proc** is popped off the stack after “return” from **minithread_switch**
- It runs
 - body_proc(body_arg)**
 - final_proc(final_arg)**
- To execute the user-provided function and allow thread cleanup
- **root_proc** doesn't return

When your program starts

- NT has kernel threads
- When your program starts:
 - one kernel thread of control
 - NT-provided execution stack



Code example

```
int proc(int* arg) {  
    printf("Hello, world!\n");  
  
    return 0;  
}  
  
main() {  
    minithread_system_initialize(proc, NULL);  
}
```

Initialising the system

- `minithreads_system_initialize`
`(proc_t mainproc, arg_t mainarg)`
- Starts up the system
- First user thread runs
`mainproc(mainarg)`
- Should the initial minithread be the same as the kernel thread?

Initialising the system

- If **main()** returns, it will terminate the entire process!
- Make **minithread_system_init()** not return
- It should create the first user thread, which runs **mainproc(mainarg)**
- Your kernel thread needn't terminate

Cleaning up threads

- A minithread terminates when it reaches the end of its **body_proc**
- A good solution will clean up its stack:
`minithread_free_stack(stackbase)`
- But a thread can't destroy its stack itself: **minithread_switch** won't work!

Minithread destruction

- A terminating thread T
 - runs its **final_proc**
 - “notifies the system that it wants to finish”
 - relinquishes the processor
- Some other thread
 - sees T needs to be cleaned up
 - frees T 's stack, etc

A word of warning

- The NT kernel thread has a stack
- You can make a minithread out of it
- But you can't free the NT stack!

Summary

- Implement queues
- Fork, initialisation, yield for 1 thread
- Yielding between multiple threads
- Thread cleanup
- Semaphores
- Implement “food services” problem concurrently if you like, or later