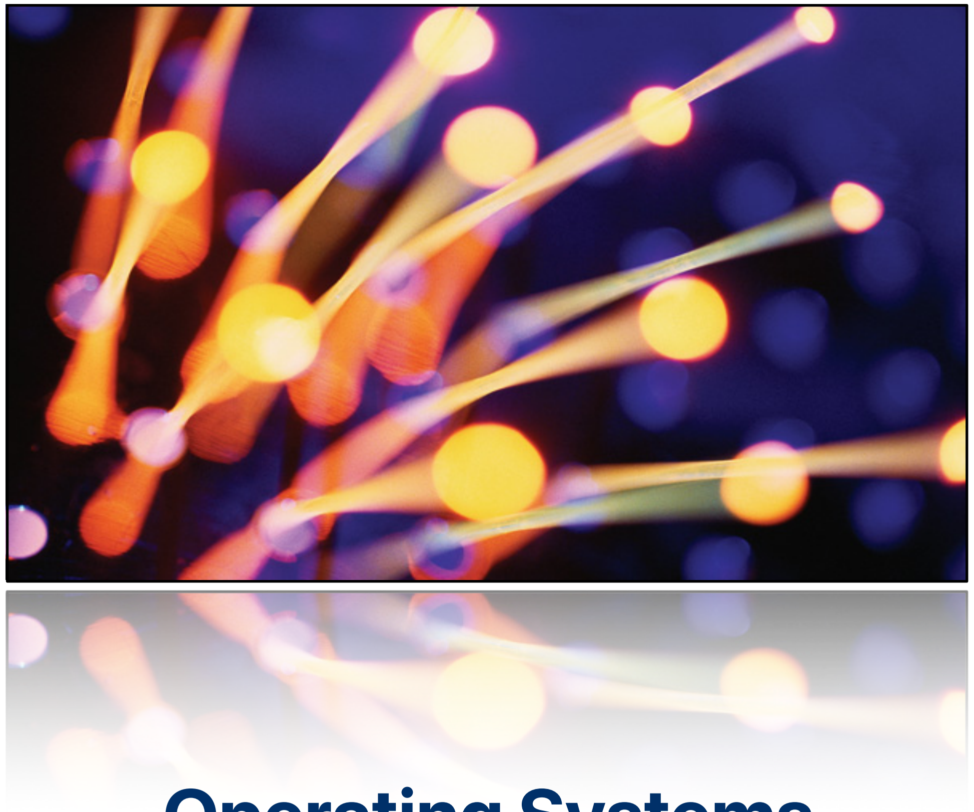
Context Switching: A Deep Dive



Operating Systems

Baochun Li

University of Toronto

Context switching: revisited

In our producer-consumer problem, we assumed a separate processor was available to run each thread

But there are usually not enough processors to go around

We need to share a limited number of processors among a large number of threads

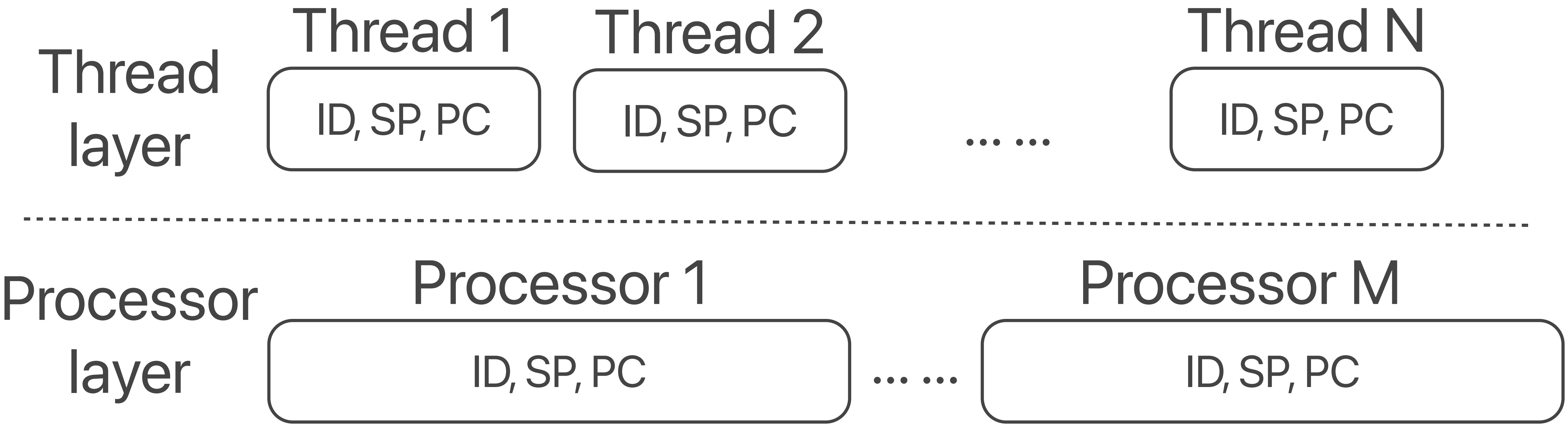
- To make things simple, we first assume that no thread hogs the processor, either accidentally or intentionally

Recall: the abstraction of threads

A thread is an abstraction that encapsulates the state of execution

The execution environment captures everything needed for a thread scheduler to stop a thread and then resume it later

The ability to stop a thread and then resume it later can be used to multiplex many threads over a limited number of physical processors



Thread layer

Thread 1
ID, SP, PC

Thread 2
ID, SP, PC

… …

Thread N
ID, SP, PC

Processor layer

Processor 1
ID, SP, PC

… …

Processor M
ID, SP, PC

Creating a new thread with the thread scheduler

```
thread_id = thread_allocate(starting_function,
                            address_space_id)
```

To implement this, the thread scheduler:

allocates a range of memory in address_space_id to be used as the stack for function calls

selects a processor, and sets the processor's PC to the address starting_function

sets the processor's SP to the bottom of the allocated stack

How does the thread scheduler share a limited number of processors among potentially many threads?

Revisiting the producer-consuming problem

```
message buffer[N]
int in = 0, out = 0
mutex buffer_lock = UNLOCKED
send(message msg)
   acquire(buffer_lock)
   while in - out == N do
      release(buffer_lock)
      acquire(buffer_lock)
   buffer[in modulo N] = msg
   in = in + 1
   release(buffer_lock)
message receive()
   acquire(buffer_lock)
   while in == out do
      release(buffer_lock)
      acquire(buffer_lock)
   msg = buffer[out modulo N]
   out = out + 1
   release(buffer_lock)
   return msg
```

Revisiting the producer-consumer problem

Previously, we assume having one processor per thread, so the spin-loop implementation of send() and receive() is appropriate at the time

but they are now inappropriate since we have fewer processors than threads

- If there is just one processor and if the receiver started before the sender, we have a major problem
- The receiver thread executes its spinning loop, and the sender never gets a chance to run (to add an item to the buffer)

Adding yield() to the implementation

```
message buffer[N]
int in = 0, out = 0
mutex buffer_lock = UNLOCKED
send(message msg)
   acquire(buffer_lock)
   while in - out == N do
      release(buffer_lock)
      yield()
      acquire(buffer_lock)
   buffer[in modulo N] = msg
   in = in + 1
   release(buffer_lock)
message receive()
   acquire(buffer_lock)
   while in == out do
      release(buffer_lock)
      yield()
      acquire(buffer_lock)
   msg = buffer[out modulo N]
   out = out + 1
   release(buffer_lock)
   return msg
```

The job of yield(): context switching

yield() switches a processor from one thread to another

- Save this thread's state so that it can be resumed later
- Schedule another thread to run on this processor
- Dispatch this processor to that thread

But there is a problem!

- A thread in the thread layer calls yield()
- The job of yield(), however, needs to be done by the thread scheduler
- The thread scheduler is in the processor layer
- This sounds simple, but it can be the most mysterious part in an OS kernel!

Towards an implementation of yield()

Step 1. A simple implementation of the thread scheduler

Step 2. Extending **Step 1** to support creating and terminating threads

Step 3. Relax the assumption that threads cannot hog the processor

Towards an implementation of yield()

Step 1. A simple implementation of the thread scheduler

Step 2. Extending Step 1 to support creation and termination of threads

Step 3. Relax the assumption that threads cannot hog the processor

Simple implementation of yield(): assumptions

To simplify, we assume the following:

There are a fix number of threads, N, and there are fewer than N processors

All threads run in the same address space

- so that we do not need to worry about switching to a different address space, a topic in memory management

All threads are already runnable (in either RUNNING or READY state)

Two tables

processor_table: an array that records information for each processor, such as the ID of the thread that the processor is currently running

thread_table: each entry holds the stack pointer, and the state of the thread (RUNNING or READY)

- in a system with *M* processors, *M* threads can be in the RUNNING state at the same time
- For simplicity, we do not show additional code to save (or restore) the registers or other states in an entry of thread_table, and assume that they will be saved (or restored) when the stack pointer is saved (or restored)

Simple implementation of yield()

```
struct processor_table[M]
    int thread_id
struct thread_table[N]
    int top_of_stack, state                         // thread states: RUNNING or READY


yield()

    enter_processor_layer(processor_table[CPUID].thread_id)

    return
enter_processor_layer(int this_thread)
    thread_table[this_thread].state = READY         // switch state to READY
    thread_table[this_thread].top_of_stack = SP     // store yielding thread’s SP
    scheduler()
    return
scheduler()
    j = processor_table[CPUID].thread_id
    do j = (j + 1) modulo N while thread_table[j].state != READY   // schedule a READY j
    thread_table[j].state = RUNNING                 // set state to RUNNING
    processor_table[CPUID].thread_id = j            // this processor now runs j
    exit_processor_layer(j)                         // dispatch this processor to j
    return
exit_processor_layer(int new_thread)
    SP = thread_table[new_thread].top_of_stack      // load SP of new thread
    return
```

Problem: Race condition

When we have more than one processor, different threads running on separate processors may try to invoke yield() at the same time!

As usual, we solve the problem using mutex locks

Simple implementation of yield()

```
struct processor_table[M]
    int thread_id
struct thread_table[N]
    int top_of_stack, state                       // thread states: RUNNING or READY
mutex thread_table_lock

yield()
    acquire(thread_table_lock)
    enter_processor_layer(processor_table[CPUID].thread_id)
    release(thread_table_lock)
    return

enter_processor_layer(int this_thread)
    thread_table[this_thread].state = READY       // switch state to READY
    thread_table[this_thread].top_of_stack = SP   // store yielding thread's SP
    scheduler()
    return

scheduler()
    j = processor_table[CPUID].thread_id
    do j = (j + 1) modulo N while thread_table[j].state != READY   // schedule a READY j
    thread_table[j].state = RUNNING               // set state to RUNNING
    processor_table[CPUID].thread_id = j          // record that processor runs j
    exit_processor_layer(j)                       // dispatch this processor to j
    return

exit_processor_layer(int new_thread)
    SP = thread_table[new_thread].top_of_stack    // load SP of new thread
    return
```

Important observations

The thread scheduler selects the next thread in a round-robin fashion

The thread that releases the lock is most likely a different thread from the one that acquired the lock!

- The scheduler is likely to choose a different thread to run

It is unnecessary to save and restore the program counter — why?

Our simple implementation: an in-depth look

The return statement: pops the return address off the top of the stack, and move that address to the PC

If we are switching from thread 1 to thread 2 on processor A —

- Thread 1 calls yield()
- yield() acquires thread_table_lock, calls enter_processor_layer()
- enter_processor_layer() saves states, calls scheduler(), still in thread 1
- scheduler() calls exit_processor_layer()
- exit_processor_layer() changes SP to the top_of_stack in thread 2
- The return statement in exit_processor_layer() pops the return address off the top of the stack in thread 2
- **Where does it return to?**

Where does return in exit_processor_layer()

It depends on what is on the top of the stack in thread 2!

The top of the stack in thread 2 is saved in enter_processor_layer() before it calls scheduler()

When exit_processor_layer() returns, it is as if enter_processor_layer() returns

The return statement will take PC back to yield()

Thread 2 will now release the thread_table_lock, and return from yield()

Towards an implementation of yield()

Step 1. A simple implementation in the thread scheduler

Step 2. Extending Step 1 to support creating and terminating threads

Step 3. Relax the assumption that threads cannot hog the processor

Relaxing previous assumptions

To progress to a more complete thread scheduler, we no longer assume that —

- There exists a fixed number of threads
- There are more threads than physical processors

This implies that we need two more functions in addition to thread_allocate()

- thread_exit(): destroy and clean up the calling thread. When a thread is done with its work, it invokes this function to release its state
- thread_destroy(*id*): destroy the thread identified by *id*. In some cases, one thread may need to terminate another thread (e.g., one in an endless loop)

Subtle issues that need to be solved

If a thread terminates itself with thread_exit(), how can it deallocate its own stack?

- One possible idea: Let the next thread being scheduled to deallocate the stack of a terminated thread?
- What if a processor sits idle, with no thread scheduled next? (Remember we no longer assume more threads than processors)

Even if no processor sits idle, how can a target thread running on one of the processors be destroyed by another (calling) thread?

- The calling thread cannot just deallocate the target thread’s stack!
- The processor running the target thread must do that

Solution: Add a processor-layer thread for each processor!

yield() using processor-layer threads

```
struct processor_table[M]
   int top_of_stack
   int reference stack        // pre-allocated stack for this processor th
   int thread_id              // id of thread running on this processor
struct thread_table[N]
   int top_of_stack, state    // thread states: RUNNING, READY,
                           // UNUSED, EXITED or DESTROYED
   int reference stack        // stack for this thread
mutex thread_table_lock

yield()
   acquire(thread_table_lock)
   enter_processor_layer(processor_table[CPUID].thread_id, CPUID)
   release(thread_table_lock)
   return

run_processors()
   for each processor do
      allocate stack and set up a processor thread
      shutdown = FALSE
      scheduler()
      deallocate stack
      halt processor
```

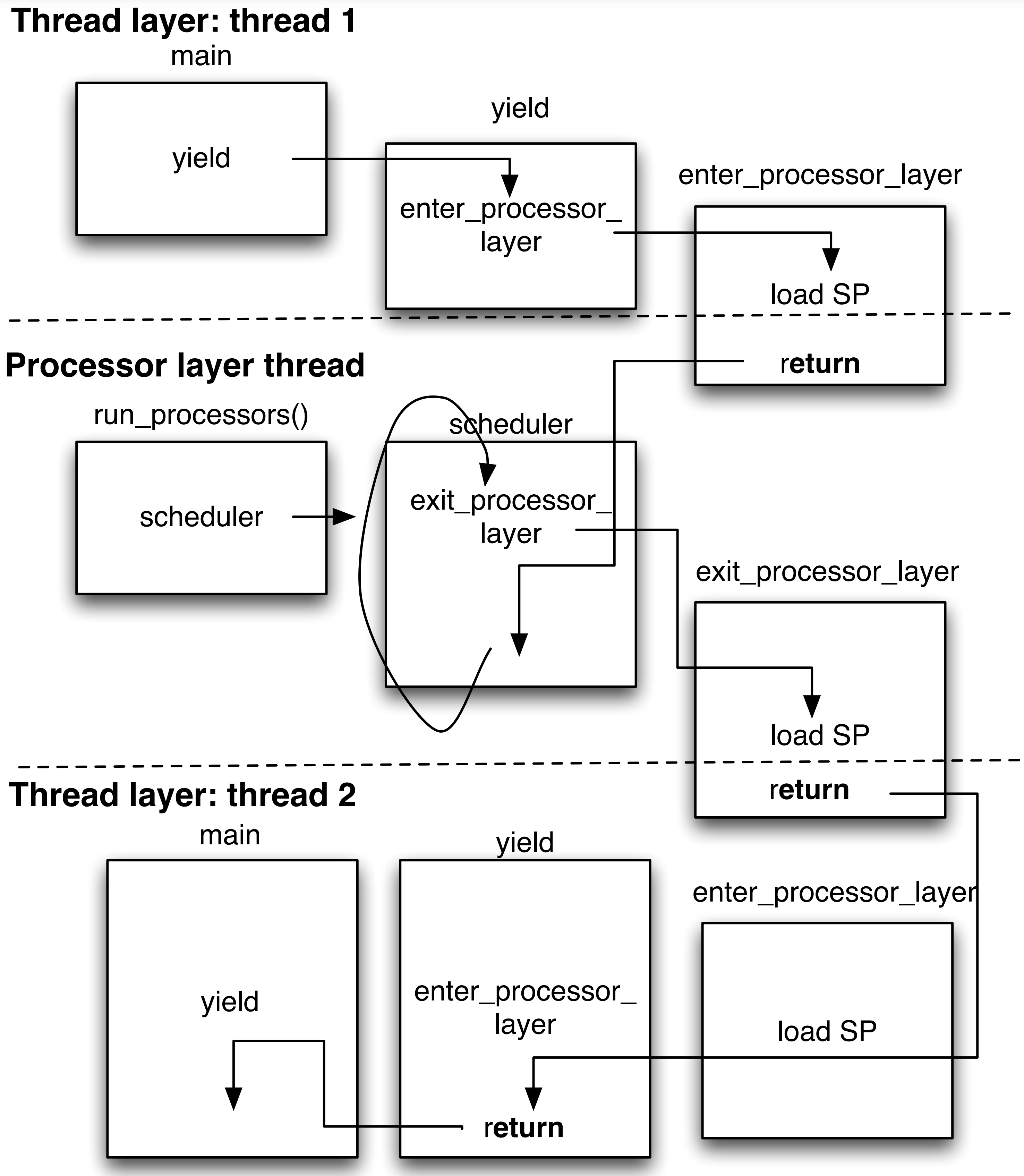
yield() using processor-layer threads

```
enter_processor_layer(int this_thread, int processor_id)
    if thread_table[this_thread].state == RUNNING     // if not yet destroyed
        thread_table[this_thread].state = READY       // switch state to READY
    thread_table[this_thread].top_of_stack = SP       // store yielding thread's SP
    SP = processor_table[processor_id].top_of_stack // dispatch: load SP of
                                                    // processor thread
    return

scheduler()
    while shutdown == FALSE do
        acquire(thread_table_lock)
        for i = 0 to N - 1 do
            if thread_table[i].state == READY then
                thread_table[i].state = RUNNING
                processor_table[CPUID].thread_id = i
                exit_processor_layer(i, CPUID)
                if thread_table[i].state == EXITED or DESTROYED then
                    thread_table[i].state = UNUSED
                    deallocate(thread_table[i].stack)
        release(thread_table_lock)
    return                                          // go shut down this processor

exit_processor_layer(int new_thread, int processor_id)
    processor_table[processor_id].top_of_stack = SP // store SP of processor thread
    SP = thread_table[new_thread].top_of_stack  // dispatch: load SP of new thread
    return
```

The way that control flows across threads



Thread layer: thread 1

main

yield

yield

enter_processor_layer

enter_processor_layer

load SP

return

Processor layer thread

run_processors()

scheduler

scheduler

exit_processor_layer

exit_processor_layer

load SP

return

Thread layer: thread 2

main

yield

yield

enter_processor_layer

enter_processor_layer

load SP

return

Implementing thread_allocate()

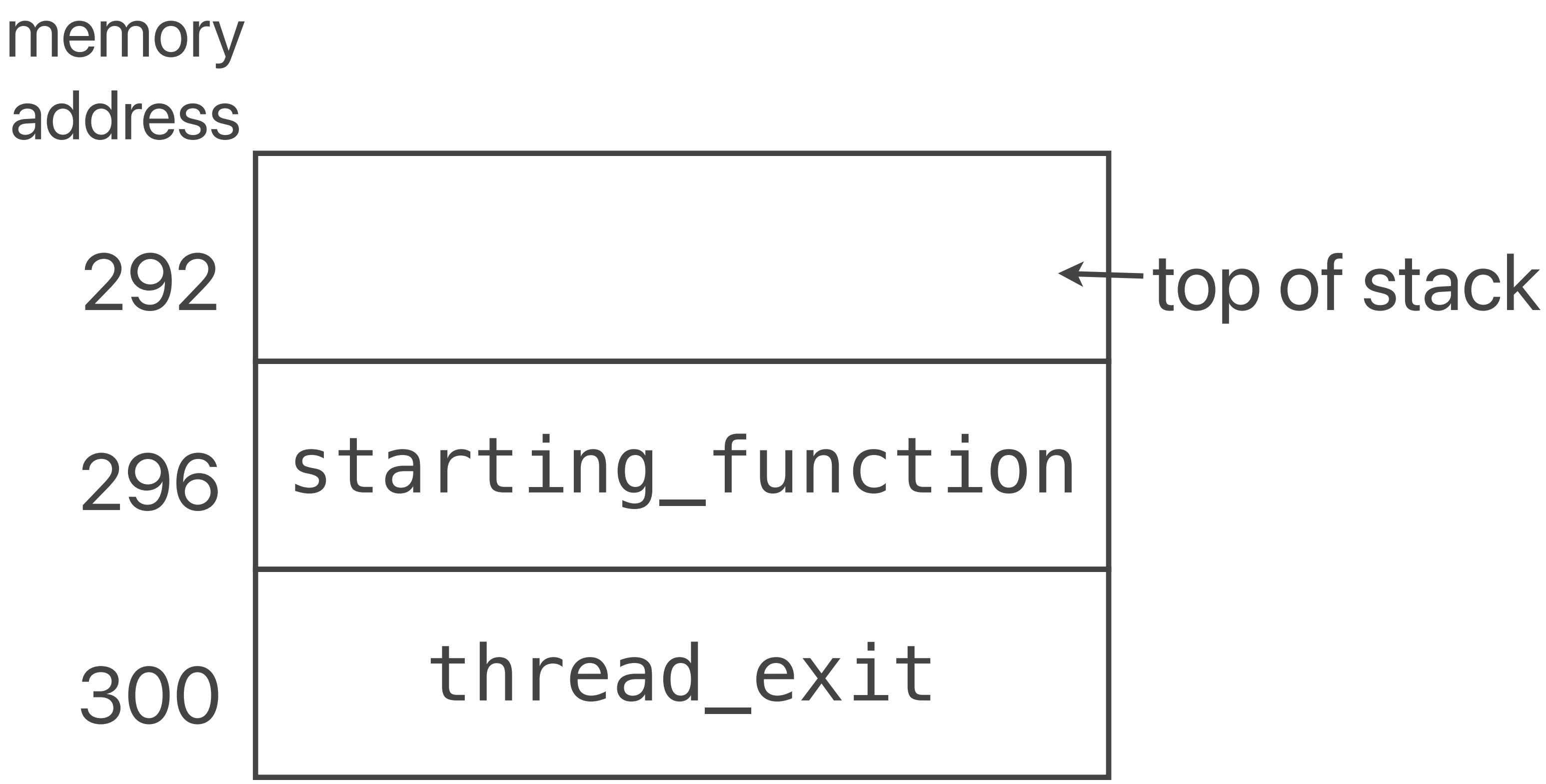
```
thread_allocate()
    allocate memory for a new stack

    push an empty frame onto the new stack, with just a return address
    initialize that return address with the address of thread_exit()

    push a second empty frame, with just a return address
    initialize that return address with the address of starting_function()

    find an entry in the thread table that is UNUSED
    store the top of the stack in that entry
    set the state of the new thread in that entry to READY
```

Implementing thread_allocate()



With the initial setup of the new stack, it appears that thread_exit() called starting_function(), and the thread is about to return to this function

When scheduler() selects this thread, its return will go to the function starting_function

- starting_function will release thread_table_lock, and the new thread is running

Implementing thread_exit()

When a thread finishes with **starting_function**, it returns using the standard procedure return convention

Since **thread_allocate** has pushed the address of **thread_exit** on the stack, this **return** transfers control to **thread_exit**

```
thread_exit()
   acquire(thread_table_lock)
   thread_table[get_thread_id()].state = EXITED
   enter_processor_layer(get_thread_id(), CPUID)


get_thread_id()
   return processor_table[CPUID].thread_id
```

Implementing thread_destroy()

Recall that the calling thread cannot just deallocate the target thread's stack

- The processor running the target thread must do that

Instead, thread_destroy() simply sets the state of the thread to DESTROYED, and returns

When the target thread invokes yield(), the processor-layer thread's scheduler() will check the state and release the thread's resources

But how do we ensure that each thread running on a processor will call yield() occasionally?

Towards an implementation of yield()

Step 1. A simple implementation in the thread scheduler

Step 2. Extending Step 1 to support creating and terminating threads

Step 3. Relax the assumption that threads cannot hog the processor

Preemptive scheduling

Cooperative scheduling is not good enough as a programmer may forget to include a yield() call

- If there is only one processor, it may appear to freeze, as no other thread has an opportunity to make progress (example: Windows 3.1)

Preemptive scheduling: the thread scheduler forces a thread to give up the processor after some time (say, 100 milliseconds)

- by using timer interrupts

The timer interrupt is handled in the processor layer

The timer interrupt handler can then invoke yield() in the thread layer

- Any problems here?

Potential deadlock

Potential deadlock

The interrupt handler calls yield()

By chance, the interrupt happens right after the thread on that processor has acquired thread_table_lock in yield()

Deadlock!

- The yield call in the handler will try to acquire thread_table_lock too
- but it already has been acquired by the interrupted thread
- That thread cannot continue and release the lock, because it has been interrupted by the timer interrupt handler!

The problem

The problem: we have concurrent activity within the processor layer: the thread scheduler (i.e., yield) and the interrupt handler

The concurrent execution within the thread layer is coordinated with locks

But the processor needs its own mechanism

- The processor may stop processing instructions of a thread at any time and switch to processing interrupt instructions
- We lack a mechanism to turn processor instructions and interrupt instructions into separate before-or-after atomic actions!

The solution: disabling interrupts

Before a thread acquires the thread_table_lock, it also disables interrupts on its processor

Now the processor will not switch to an interrupt handler when an interrupt arrives

- Interrupts are delayed until they are enabled again

After the thread has released the thread_table_lock, it is safe to reenable interrupts

Summary: Two alternatives

There are two alternatives to implement the thread scheduler

- in the current thread, appropriate for a user thread scheduler
- in a separate thread, one for each physical processor

Need to disable and reenable interrupts to avoid deadlocks caused by concurrency with timer interrupt handlers

We made implicit assumptions to skip some details —

- For kernel threads, we need to use system calls to use the thread scheduler
- The system calls will need to trap into the kernel, and switch to kernel stacks when running in kernel mode
- We did not include the **BLOCKED** state of threads

Revisiting Semaphores

Semaphores: maintaining a “table count”

Defining semaphores: the first alternative

A semaphore is a non-negative integer that remembers past wakeups

down(semaphore): if semaphore > 0, decrement semaphore. Otherwise, wait until another thread increments semaphore, then try to decrement again

up(semaphore): increment semaphore, and wake up all threads waiting on semaphore

Implementing semaphores

```
struct semaphore
  int count
up(semaphore reference sem)
  acquire(thread_table_lock)
  sem.count = sem.count + 1
  for i = 0 to N - 1 do          // wake up all threads waiting
    if thread_table[i].state == BLOCKED
    and thread_table[i].sem == sem then
      thread_table[i].state = READY
  release(thread_table_lock)
down(semaphore reference sem)
  acquire(thread_table_lock)
  tid = processor_table[CPUID].thread_id
  thread_table[tid].sem = sem    // record the semaphore reference
  while sem.count < 1 do         // give up the processor when sem<1
    thread_table[tid].state = BLOCKED
    enter_processor_layer(tid, CPUID)
  sem.count = sem.count - 1
  release(thread_table_lock)
```

Can we change the while loop to if statement?

In the implementation of down(), we used a while loop to keep checking the condition (sem.count < 1) after exiting from the processor layer

Can we change it to an if statement?

```
if sem.count < 1 then // give up the processor when sem < 1
  thread_table[i].state == BLOCKED
  enter_processor_layer(tid, CPUID)
sem.count = sem.count - 1
```

Can we change the while loop to if statement?

Can we change it to an if statement?

```
if sem.count < 1 then  // give up the processor when sem < 1
  thread_table[i].state == BLOCKED
  enter_processor_layer(tid, CPUID)
sem.count = sem.count - 1
```

Not really!

More than one thread may wake up in an **up()** call

These threads will all decrement the semaphore in **down()** if we do not check the condition (sem.count < 1) again

Only one of these threads should be allowed to proceed with down() — just like in a restaurant!

What we’ve covered so far

Principles of Computer Systems Design, An Introduction

Section 5.5.1 — 5.5.6