EEL-4736/5737
Principles of Computer System Design

Lecture Slides 12

Textbook Chapter 5

Virtual Processors Using Threads

Introduction

- Assumption so far
  - Each thread has own separate processor
- We have seen the importance of
  - privilege separation
  - the role of the kernel as a manager of shared resources
- We will now focus on how to share a processor by multiple threads

Basic thread management

- Allocating and terminating threads
- Thread_id <- ALLOCATE_THREAD (starting_procedure, address_space_id)
  - Allocate a new thread in given address space, start execution at given address
  - Return identifier that names this thread
- Thread manager:
  - Allocate a stack for thread, set SP
  - Assign a processor for execution, set its PC to the starting address

Virtualizing the processor

- Time-sharing/multiplexing/multi-tasking
- Many threads are long-running
- At many instances, threads are waiting for events
  - Spin loop in SEND/RECEIVE
  - Waiting for a slow device to reply
- Inefficient to hog the processor up
- A primitive to *yield* the processor to another thread allows for time-sharing

Yielding

- Allow threads to call a procedure YIELD
  - An entry to the thread manager
  - Allows the thread manager to give the processor to a different thread
- Manager keeps thread state
  - Running
  - Runnable
- YIELD
  - Save yielding thread's state
  - Decide which other thread will run on the processor
  - Dispatch processor to chosen thread

Spin loop with yield

procedure SEND (*buffer* **reference** *p*, *message* **instance** *msg*)

ACQUIRE (*p.buffer_lock*)

while *p.in* – *p.out* = N **do**

RELEASE (*p.buffer_lock*)

YIELD ()

ACQUIRE (*p.buffer_lock*)

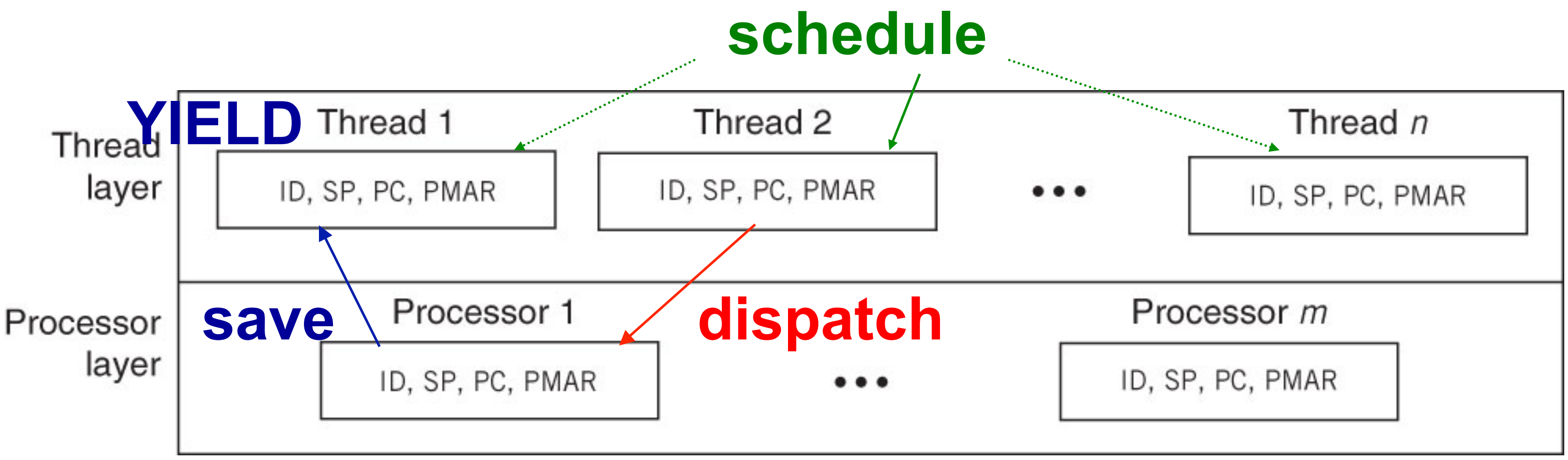
p.message[*p.in* **modulo** N] <- *msg*

p.in <= *p.in* + 1

RELEASE (*p.buffer_lock*)

YIELD

- ID (identifier), SP (stack pointer), PC (program counter), PMAR (page map address register)



YIELD implementation

- Requires processor support to implement the saving/restoring of state
  - In addition to the core registers from previous slide
  - Different instruction set architectures, different implementations - register by register, or all at once
- Implementation of YIELD belongs in the kernel, exposed through supervisor calls

Interrupts and exceptions

- Interrupts occur when an event that is unrelated to the running thread occurs
  – E.g. an I/O device interrupt
  – Note, the thread may eventually use information related to interrupt (e.g. a keyboard character) but it does not know how to handle the interrupt directly
- Exceptions are events that pertain specifically to the running thread
  – E.g. divide by zero, access out of bounds
- "Fault", "trap", "signal" are terms also used

YIELD version 1.0

- Assumptions
- Fixed number of threads (larger than number of processors)
- Assume they all run in the same address space
- Assume all threads are already running

Shared data structures

shared structure processor_table[2]

  integer thread_id   // id of running thread

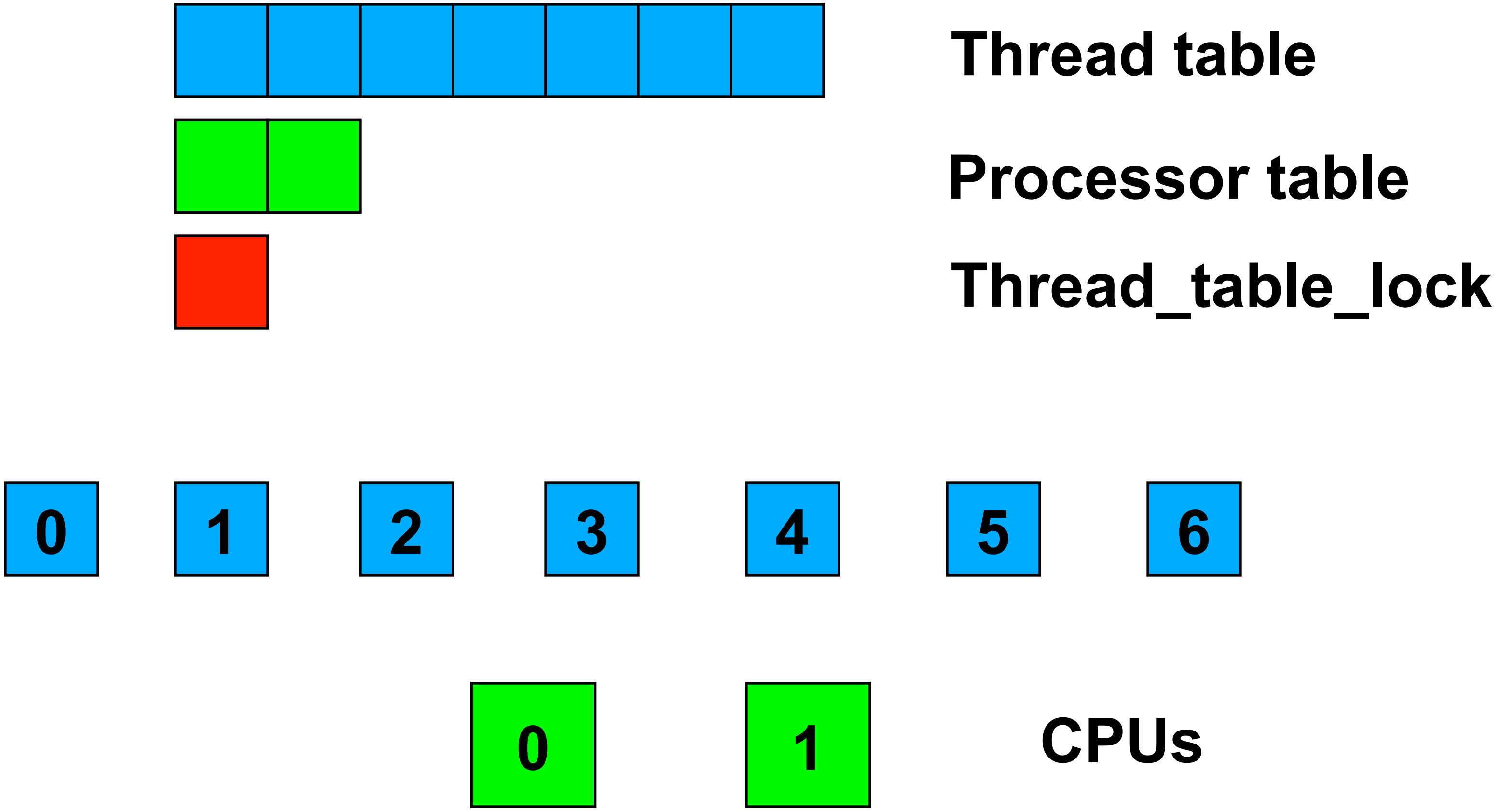
shared structure thread_table[7]

  integer topstack   // stack pointer

  integer state   // runnable/running

shared *lock* **instance** *thread_table_lock*

Example

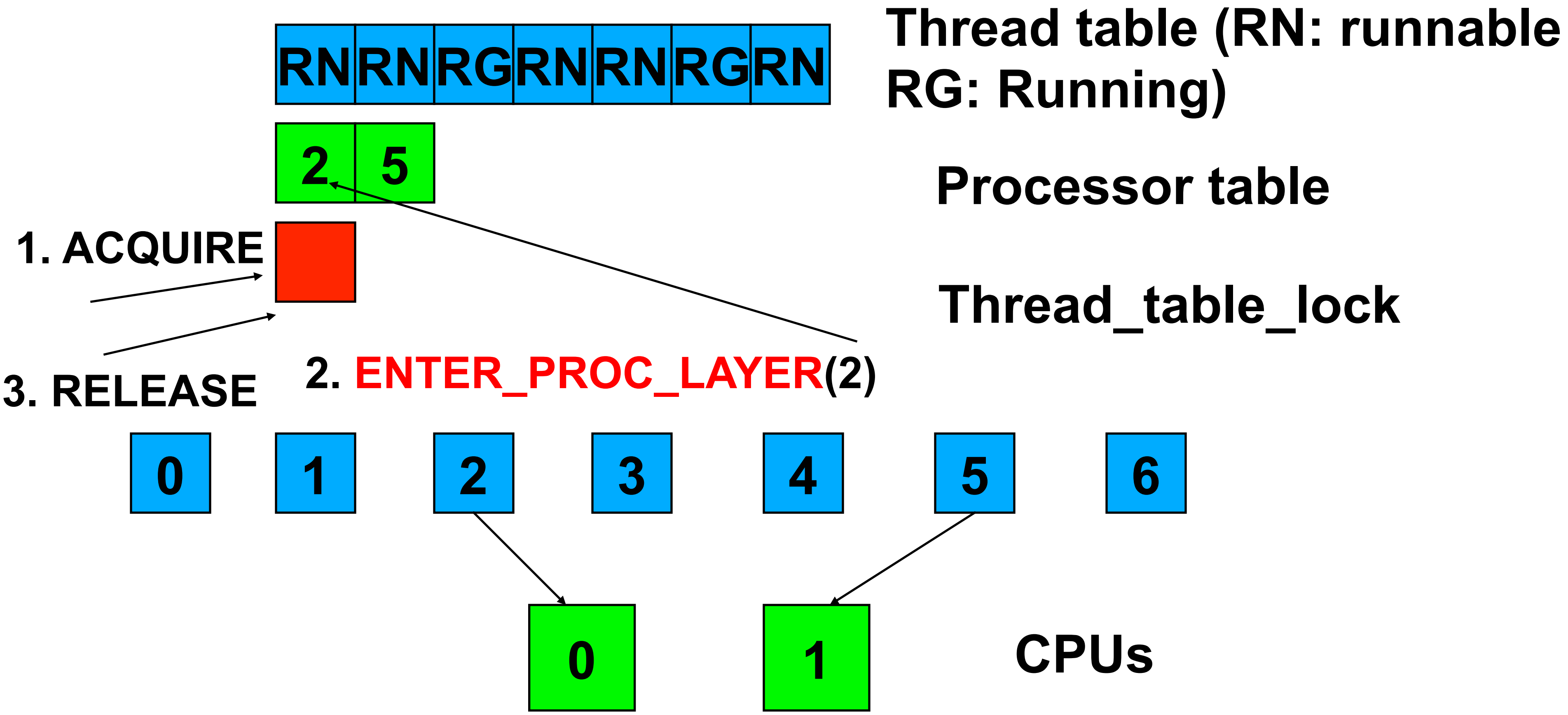

Thread table
Processor table
Thread_table_lock
0
1
2
3
4
5
6
0
1
CPUs

Main YIELD body

```
procedure GET_THREAD_ID() return
   processor_table[CPUID].thread_id
   // which thread is running on this CPU?
procedure YIELD()
   ACQUIRE (thread_table_lock)
   ENTER_PROCESSOR_LAYER
        (GET_THREAD_ID())
   RELEASE (thread_table_lock)
return
```

Example

- Thread 2 YIELDs



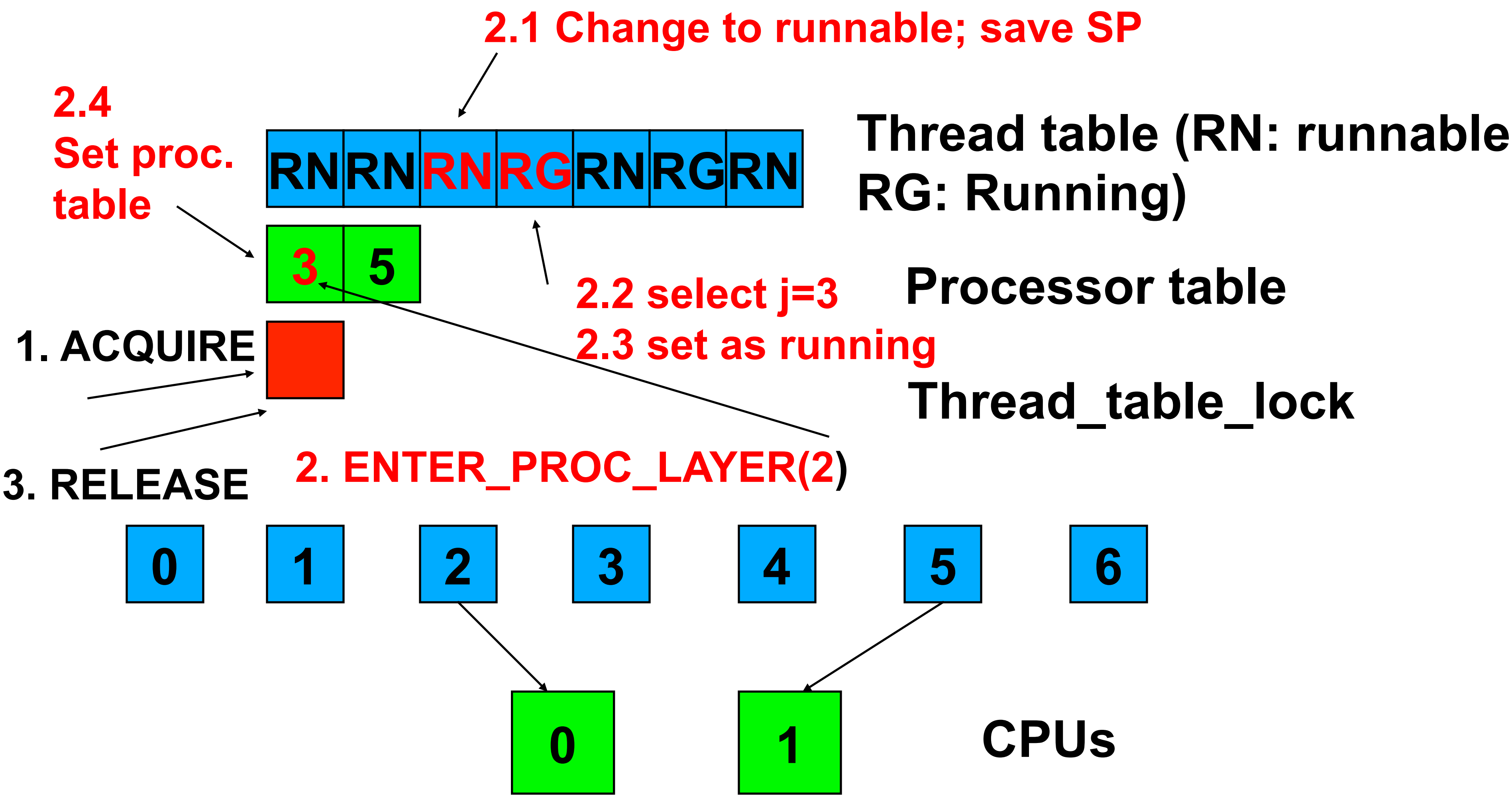
Unfolding ENTER_PROC_LAYER

procedure ENTER_PROC_LAYER(*this_thread*)
thread_table[*this_thread*].*state* <- RUNNABLE
thread_table[*this_thread*].*topstack* <- SP
SCHEDULER()
return

procedure SCHEDULER()
j = GET_THREAD_ID()
do j <- (j+1) **modulo** 7
while *thread_table*[*j*].*state* != RUNNABLE
thread_table[j].state <- RUNNING
processor_table[CPUID].*thread_id* <- j
EXIT_PROCESSOR_LAYER(j)
return

Example

- Thread 2 YIELDs



Exiting processor layer

procedure EXIT_PROCESSOR_LAYER(*new*)

SP <- *thread_table*[new].*topstack*

return

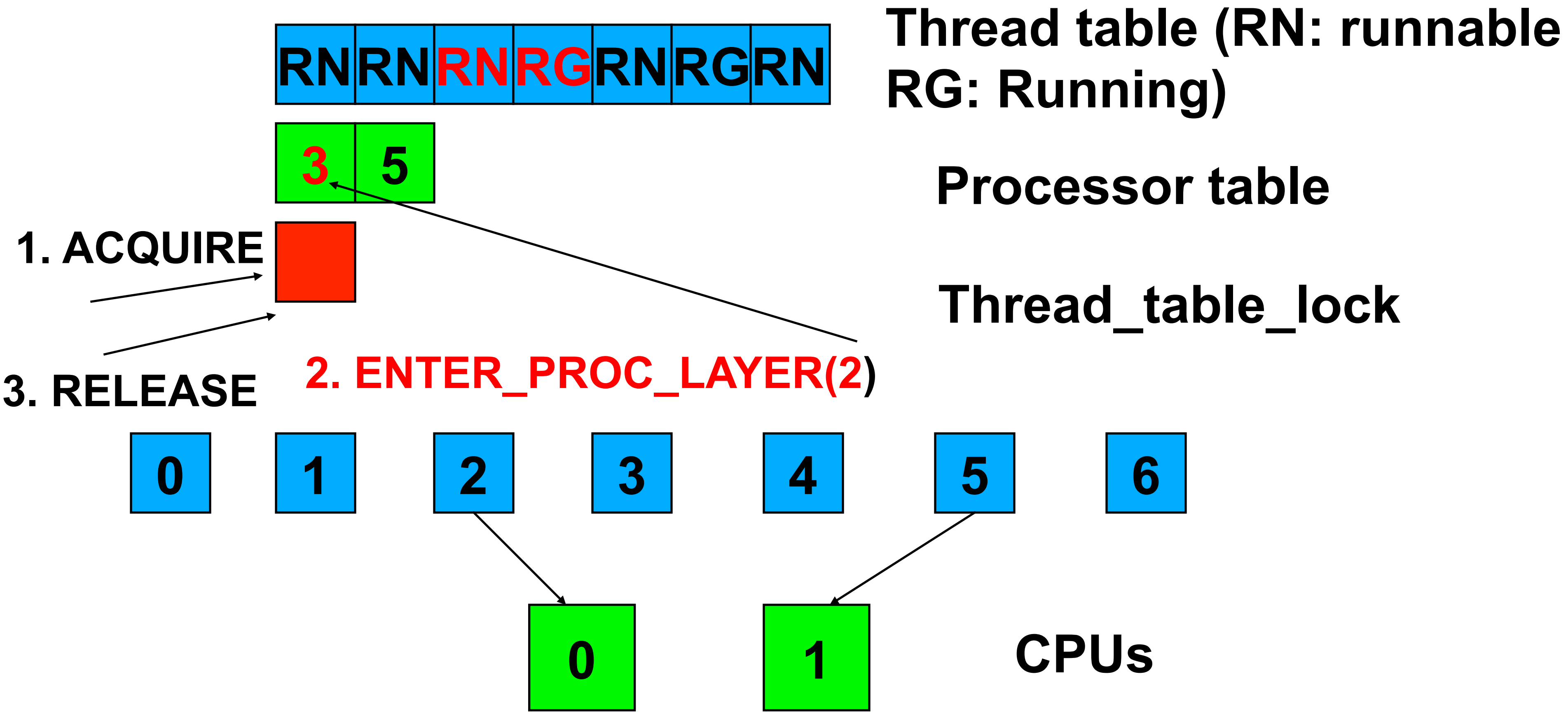
What happens when this return is called?

Example

Thread ID 2 called YIELD

Threads assumed to have been running, so thread ID 3 had called YIELD previously

We want thread ID 3 to resume executing right after its ENTER



Exiting processor layer

procedure EXIT_PROCESSOR_LAYER(*new*)

SP <- *thread_table*[new].*topstack*

return

This return pops a return address RA from thread ID new (new=3 in our example)

RA is the return address that was pushed into the stack when thread ID 3 called ENTER_PROCESSOR_LAYER (3)

Revisiting ENTER_PROC_LAYER

procedure YIELD()
  ACQUIRE (*thread_table_lock*)
  ENTER_PROCESSOR_LAYER (3)
  RELEASE (*thread_table_lock*)
return

procedure ENTER_PROC_LAYER(*this_thread*)
  thread_table[*this_thread*].*state* <- RUNNABLE
  thread_table[*this_thread*].*topstack* <- SP
  SCHEDULER()
  return

When thread ID 3 called ENTER_PROC_LAYER, its SP got saved with this return address RA

Control flow in our example

procedure YIELD()
    ACQUIRE (*thread_table_lock*)
    ENTER_PROCESSOR_LAYER (2)
    RELEASE (*thread_table_lock*)
return

Thread 2
Acquires lock

procedure EXIT_PROCESSOR_LAYER(*new*)
    SP <- *thread_table*[new].*topstack*
    return

RA from
3's stack

loads
thread 3's saved
stack into SP

Lock
is passed
between
threads

procedure YIELD()
    ACQUIRE (*thread_table_lock*)
    ENTER_PROCESSOR_LAYER (3)
    RELEASE (*thread_table_lock*)
return

Thread 3
Releases lock

Thread creation/termination

- id <- ALLOCATE_THREAD
- EXIT_THREAD ()
- DESTROY_THREAD (id)
- Possible to have fewer threads than processors
- Create a “processor-layer thread” for each processor, which runs the SCHEDULER procedure
  - Multiple “Thread-layer” threads

Processor-layer thread

- Gets created at each processor during kernel initialization
- Switching from a thread-layer thread to another thread-layer thread
  – Requires going through the processor-layer thread

Initializing processor thread

procedure RUN_PROCESSORS()

for each processor **do**

(allocate stack and set up processor thread)

shutdown <- FALSE

SCHEDULER ()

Will only return from scheduler when shutdown=TRUE

(deallocate processor thread stack)

(halt processor)

Extended data structures

shared structure processor_table[2]

integer topstack // stack pointer

byte reference stack // allocated stack (boot time)

integer thread_id // id of running thread

shared structure thread_table[7]

integer topstack // stack pointer

integer state // runnable/running/free

boolean kill_or_continue // initialized as “continue”

byte reference stack // allocated stack (create time)

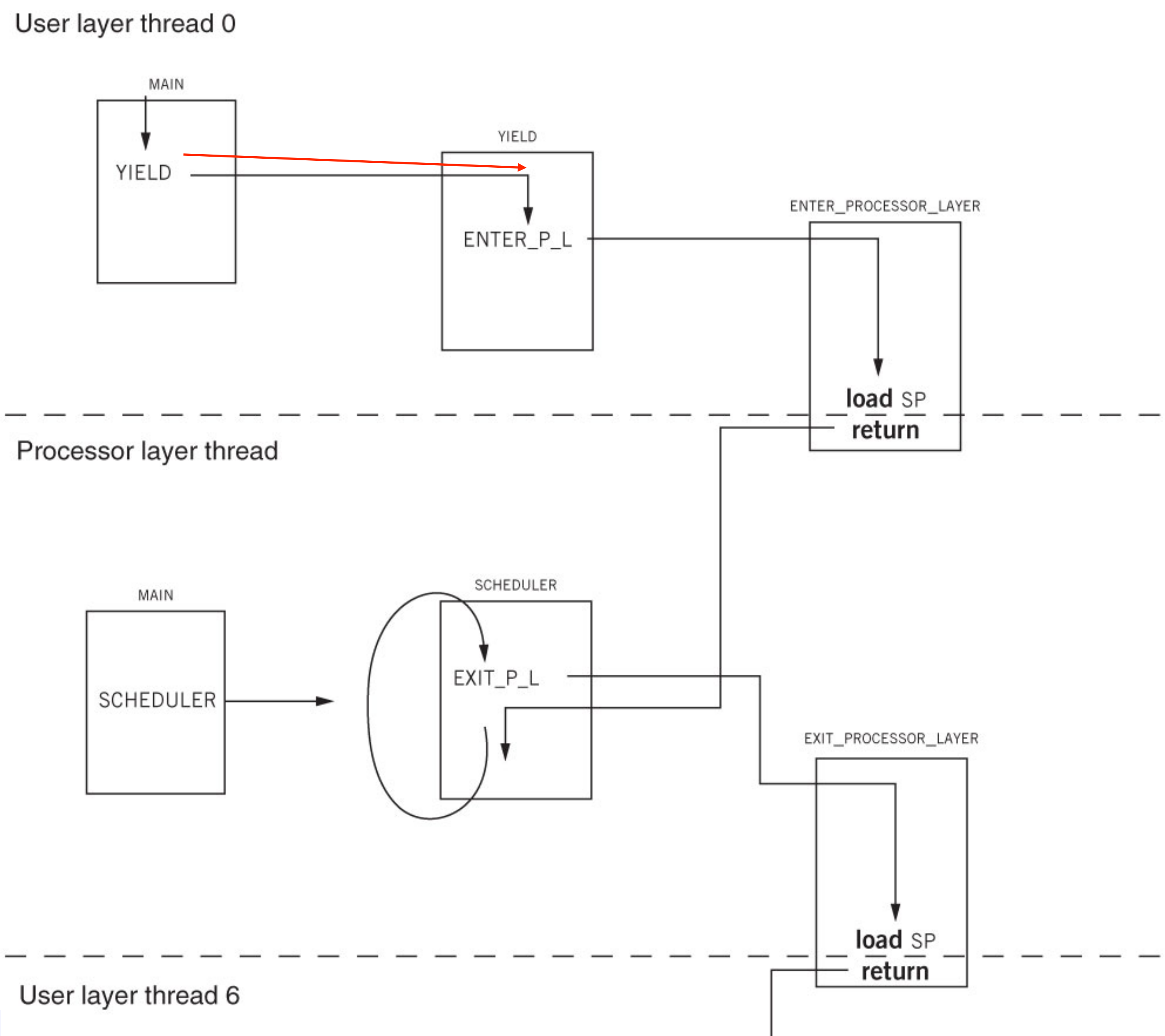
shared *lock* **instance** *thread_table_lock*

Extended YIELD

```
procedure YIELD()
   ACQUIRE (thread_table_lock)
   ENTER_PROCESSOR_LAYER
       (GET_THREAD_ID(), CPUID)
   RELEASE (thread_table_lock)
return
```

Set up to switch Into processor-layer thread

Example – 0 yields to 6



Entering processor layer

procedure ENTER_PROC_LAYER(*tid,procid*)

thread_table[*tid*].*state* <- RUNNABLE

thread_table[*tid*].*topstack* <- SP

SP <- *processor_table*[*procid*].*topstack*

~~SCHEDULER()~~

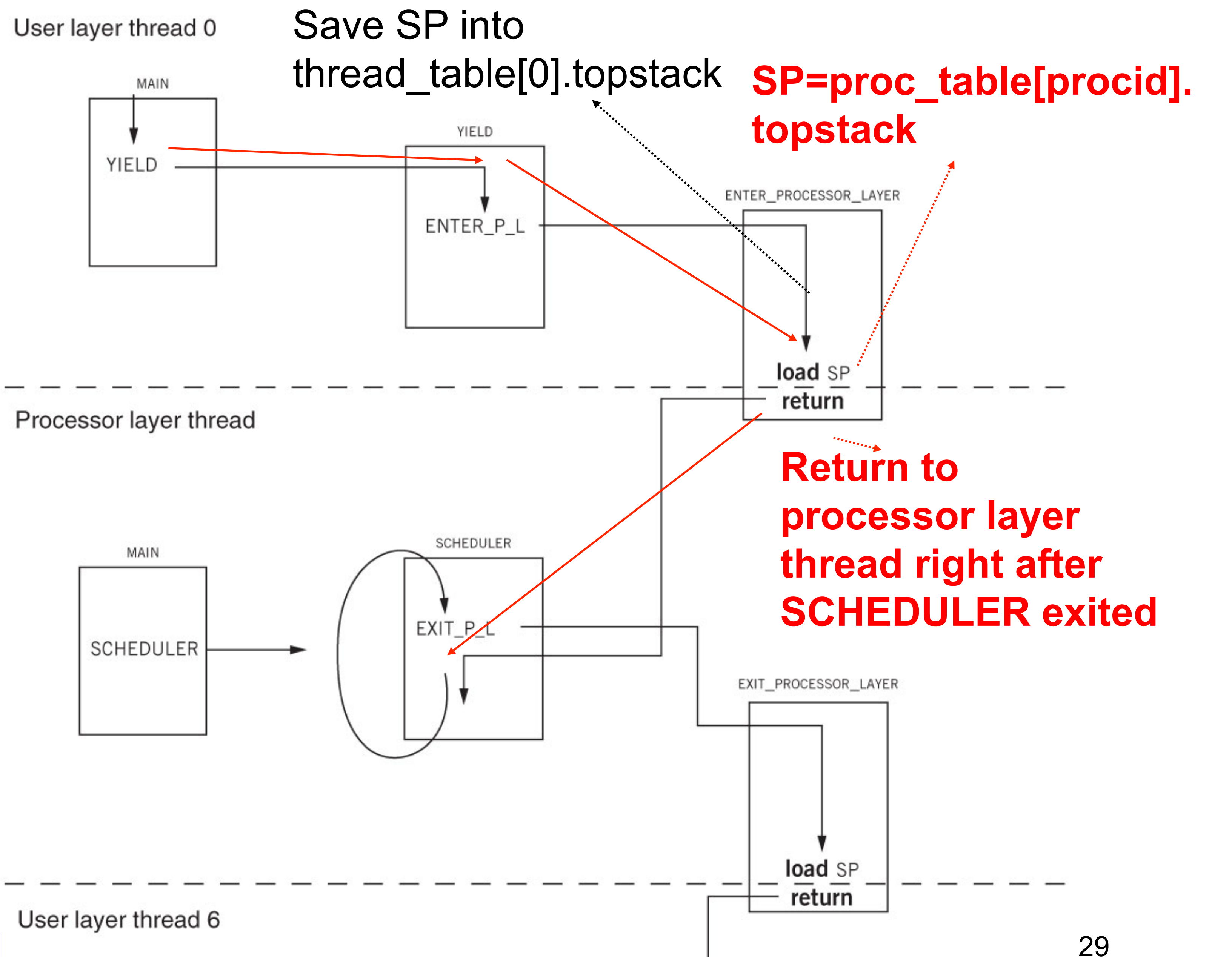
return

Enter processor layer

No need for this call anymore

Set up stack so Return will take It to processor thread (i.e. SCHEDULER)

Switching threads



Processor thread - SCHEDULER

procedure SCHEDULER()
while shutdown = FALSE **do**
ACQUIRE (*thread_table_lock*)
for i from 0 until 7 do
if *thread_table*[*i*].*state* = RUNNABLE **then**
thread_table[i].state <- RUNNING
processor_table[CPUID].*thread_id* <- i
EXIT_PROCESSOR_LAYER(CPUID, i)
if thread_table[i].kill_or_continue = KILL
thread_table[i].state <- FREE
DEALLOCATE(thread_table[i].stack)
thread_table[i].kill_or_continue = CONT
RELEASE (thread_table_lock)
return // shutdown this processor

exit to thread layer

This is where control flow returns to when a thread ENTERs

procedure
EXIT_PROCESSOR_LAYER(*procid*, *tid*)

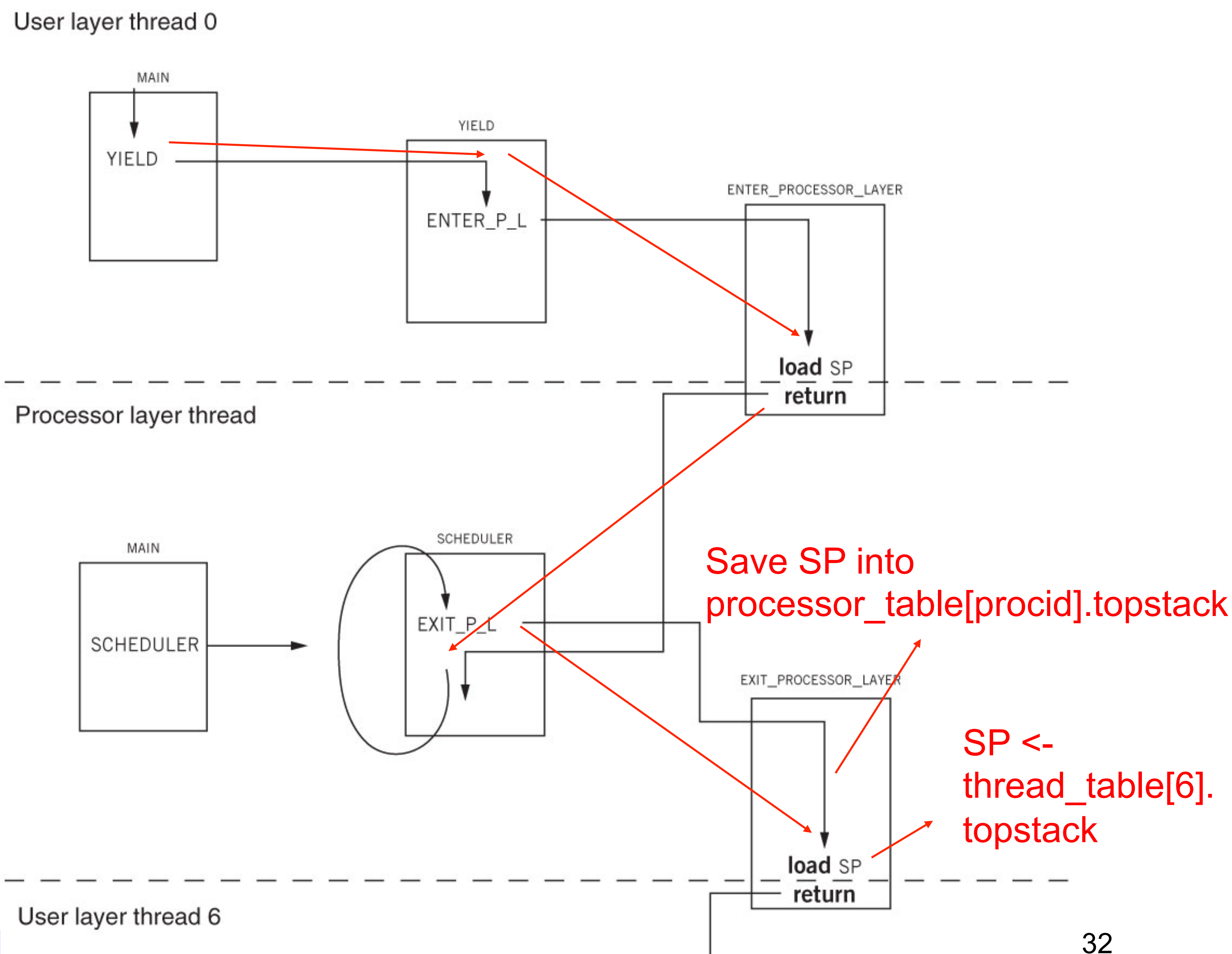
processor_table[procid].topstack <- SP

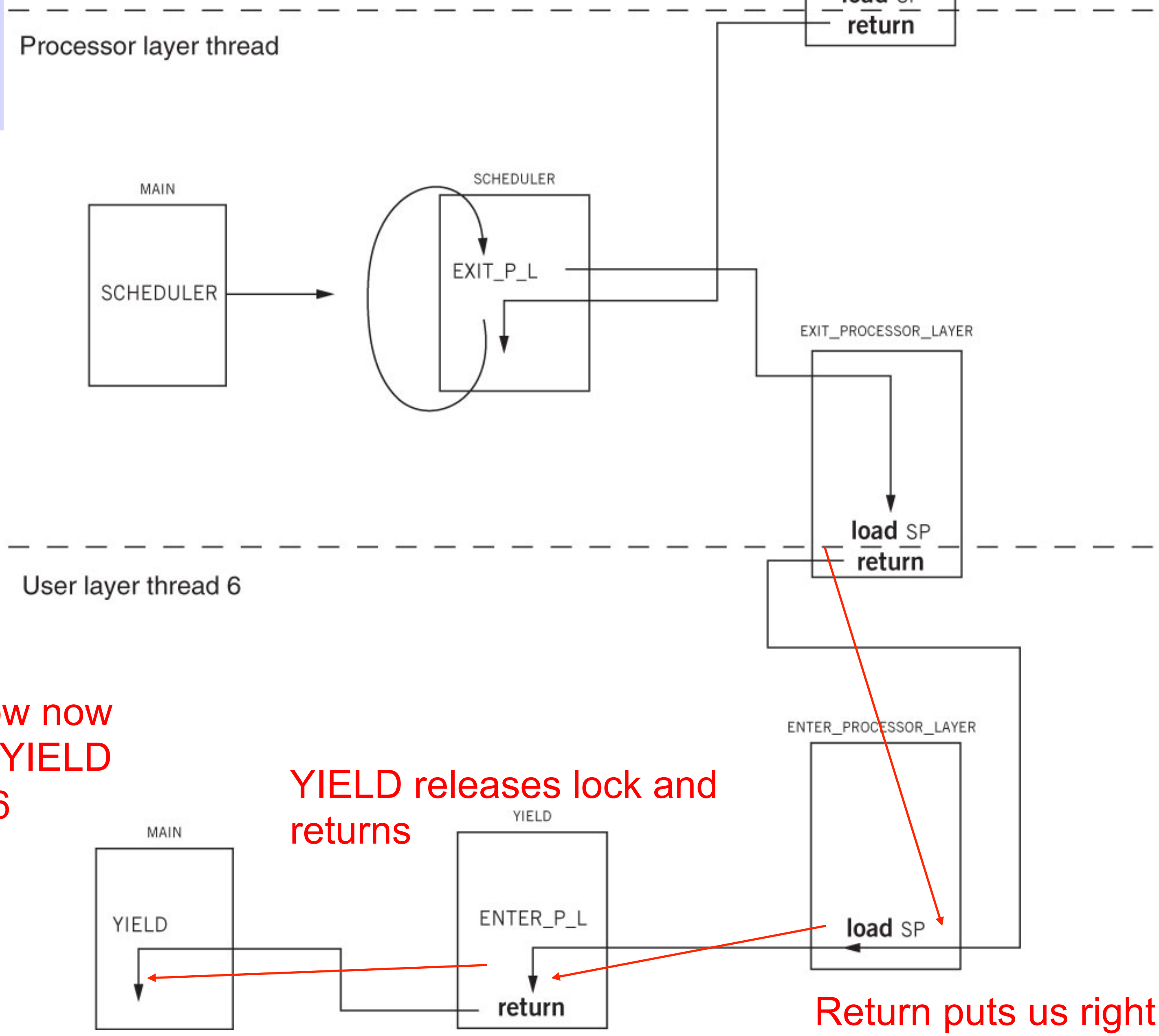
SP <- *thread_table*[tid]*.topstack*

return

Save stack pointer to processor layer thread's topstack

Switching threads





Control flow now right after YIELD of thread 6

YIELD releases lock and returns

Return puts us right after the ENTER_PROC_LAYER call of thread 6

Initializing processor thread

procedure RUN_PROCESSORS()

for each processor **do**

(allocate stack and set up processor thread)

shutdown <- FALSE

SCHEDULER ()

Will only return from scheduler when shutdown=TRUE

(deallocate processor thread stack)

(halt processor)

Processor thread - SCHEDULER

```
procedure SCHEDULER()
    while shutdown = FALSE do
        ACQUIRE (thread_table_lock)
        for i from 0 until 7 do
                if thread_table[i].state = RUNNABLE then
                  thread_table[i].state <- RUNNING
                  processor_table[CPUID].thread_id <- i
                  EXIT_PROCESSOR_LAYER(CPUID, i)
                  if thread_table[i].kill_or_continue = KILL
                    thread_table[i].state <- FREE
                    DEALLOCATE(thread_table[i].stack)
                    thread_table[i].kill_or_continue = CONT
        RELEASE (thread_table_lock)
    return  // return to RUN_PROCESSORS
            //  and shutdown this processor
```

Allocating threads

- Allocate space in memory for new stack
- Initialize this stack with return addresses of EXIT_THREAD and starting_procedure
- Find an entry in thread table that is FREE and initialize for the new thread by storing the top of the new stack
- Set state of newly created thread as RUNNABLE

Example

- id <- ALLOCATE_THREAD (starting_procedure)

Thread_table

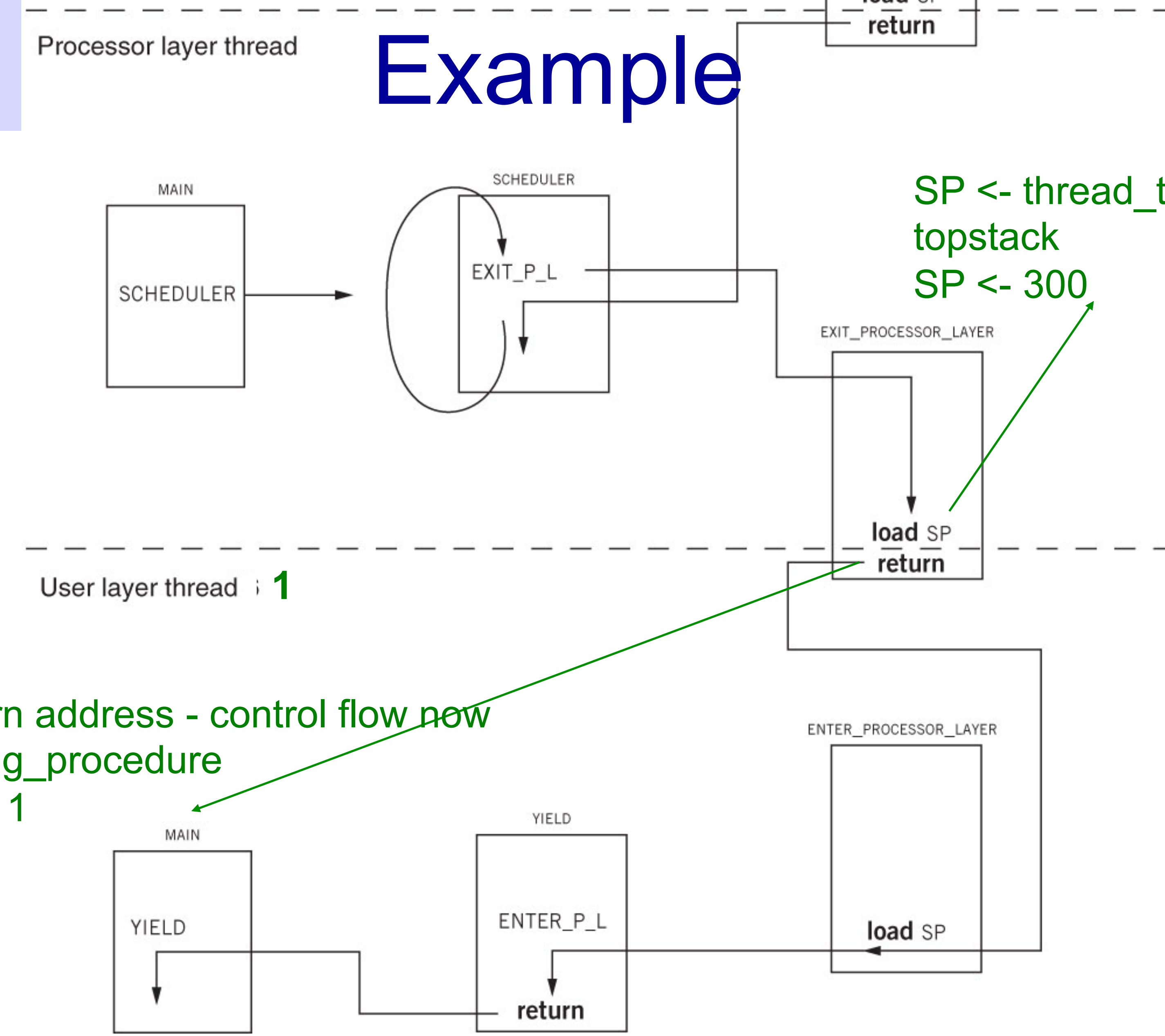
	topstack	state
0	100	RUNNABLE
1	300	RUNNABLE
⋮		
6	200	RUNNING

Stack initialized for thread id=1

300		stack top
296	starting_procedure	
292	EXIT_THREAD	

Upon first time it is scheduled, this address is popped from stack

Example



SP <- thread_table[1].
topstack
SP <- 300

Pop return address - control flow now
on starting_procedure
of thread 1

Exiting a thread

- Initialized thread stack
  - Constructed to appear as if EXIT_THREAD called starting_procedure
  - When thread finishes its execution and returns from starting_procedure, control flow goes to EXIT_THREAD

procedure EXIT_THREAD()

ACQUIRE (thread_table_lock)

thread_table[tid].kill_or_continue <- KILL

ENTER_PROCESSOR_LAYER(tid, CPUID)

Processor thread - SCHEDULER

```
procedure SCHEDULER()
    while shutdown = FALSE do
          ACQUIRE (thread_table_lock)
          for i from 0 until 7 do
                    if thread_table[i].state = RUNNABLE then
                      thread_table[i].state <- RUNNING
                      processor_table[CPUID].thread_id <- i
                      EXIT_PROCESSOR_LAYER(CPUID, i)
                      if thread_table[i].kill_or_continue = KILL
                        thread_table[i].state <- FREE
                        DEALLOCATE(thread_table[i].stack)
                        thread_table[i].kill_or_continue = CONT
          RELEASE (thread_table_lock)
    return  // return to RUN_PROCESSORS
            //  and shutdown this processor
```

Shortcomings

- YIELD – expects threads to be cooperative in sharing the processor
  - Non-preemptive scheduling
- Issues
  - Length of time a thread holds processor is up to the thread itself to decide
  - Errors can propagate from one module to another
    - E.g. infinite loop, fate sharing

Enforcing Modularity

- In order to enforce modularity, provide all threads with access to the processor
  - Pre-emptive scheduling
- Thread manager invoked not only when a thread YIELDs, but also re-scheduled on a periodic basis
  - Forces a thread to yield
- Core requirements:
  - A timer device that generates interrupts
  - Separation of privilege: Kernel vs. user

Pre-emptive scheduling

- Time interval – raise interrupt
- Interrupt handler
  - Runs in processor layer, in privileged kernel mode
  - Invokes exception handler in thread layer, which forces thread to YIELD
- Need to deal with concurrency
  - Interrupt handler thread and processor layer thread

Pre-emptive scheduling

- Example:
  - Thread 0 calls YIELD and acquires thread_table_lock
  - Hardware timer interrupt arrives and interrupt handler also calls YIELD
    - Another attempt to acquire thread_table_lock – deadlock
- An often-used solution relies on processor support to disable/re-enable interrupts
  - Disable interrupts before acquiring lock in YIELD
  - Re-enable after releasing the lock
  - Interrupt handling is disabled in between

Modularity in memory

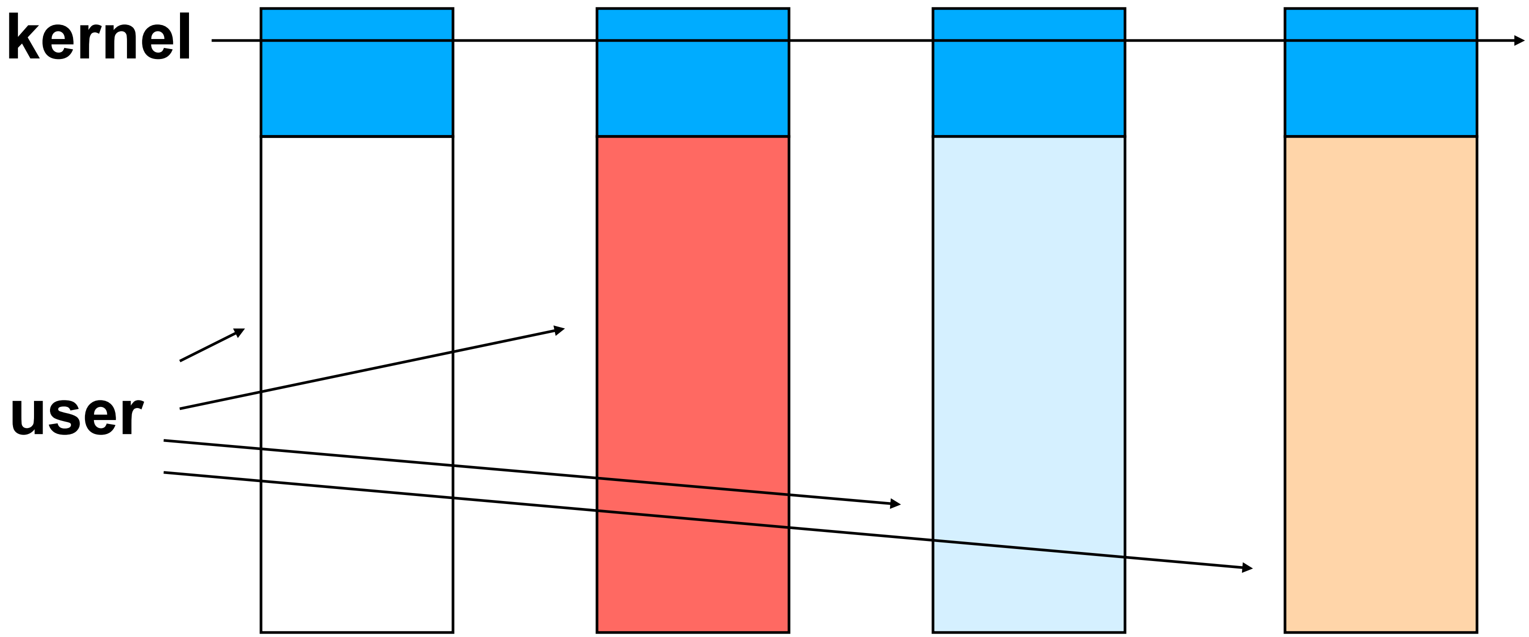
- Pre-emptive scheduling needs to be coupled with an approach to enforce modularity in memory
- Make thread manager aware of virtual address spaces of threads
  - When switching to another thread, also switch the virtual address space
    - Thread state also holds PMAR
    - ENTER_PROCESSOR_LAYER: save PMAR into thread state (thread_table[] extended to store PMAR)
    - EXIT_PROCESSOR_LAYER: load PMAR from thread state

Address space switch

- Once the PMAR register is switched, all further memory references will be translated with the new PMAR
- Additional complexity to deal with
  – Map thread manager text/data address space in a subset of every thread’s address space (with KERNEL_ONLY)
  – If hardware supports loading of PMAR, SP, PC in a single atomic action, may switch the entire context in one shot
    - Add processor-layer PMAR to processor_state

Address space mappings

- Reduces availability of virtual addresses to user-level threads
- Avoids the need to invalidate (flush) hardware-based TLB when entering the kernel
  - Flush still needed if another thread is scheduled



Reading

- Sections 6.1 and 6.3