COT 4600 Operating Systems Fall 2009

Dan C. Marinescu

Office: HEC 439 B

Office hours: Tu-Th 3:00-4:00 PM

Lecture 19

- Last time:
  - Enforcing Modularity in Memory
- Today:
  - Sharing a processor among multiple threads
  - Implementation of the YIELD
  - Creating and terminating threads
  - Preemptive scheduling
- Next Time:
  - Thread primitives for sequence coordination

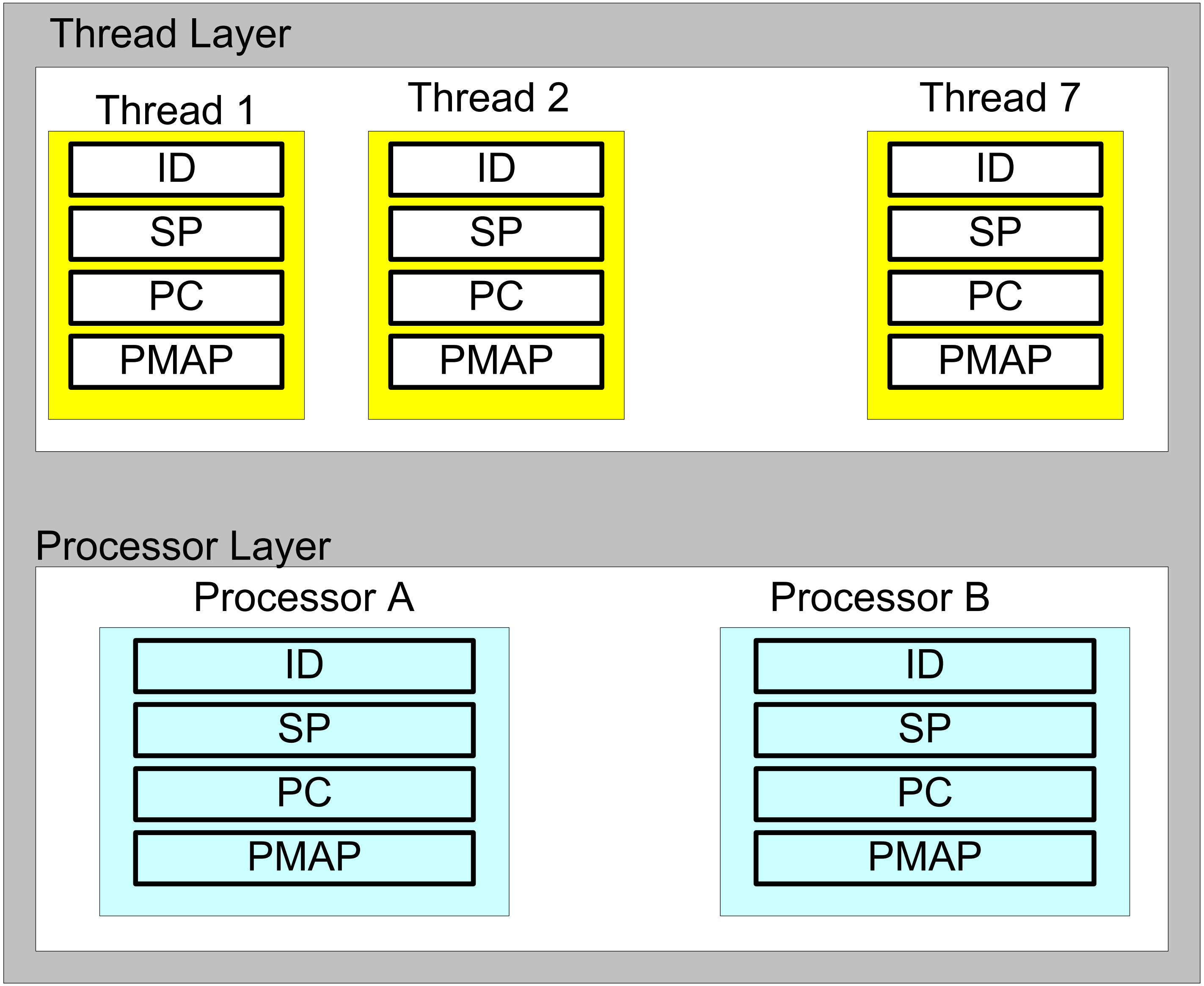
Virtualization of threads

- Implemented by the operating system for the three abstractions:

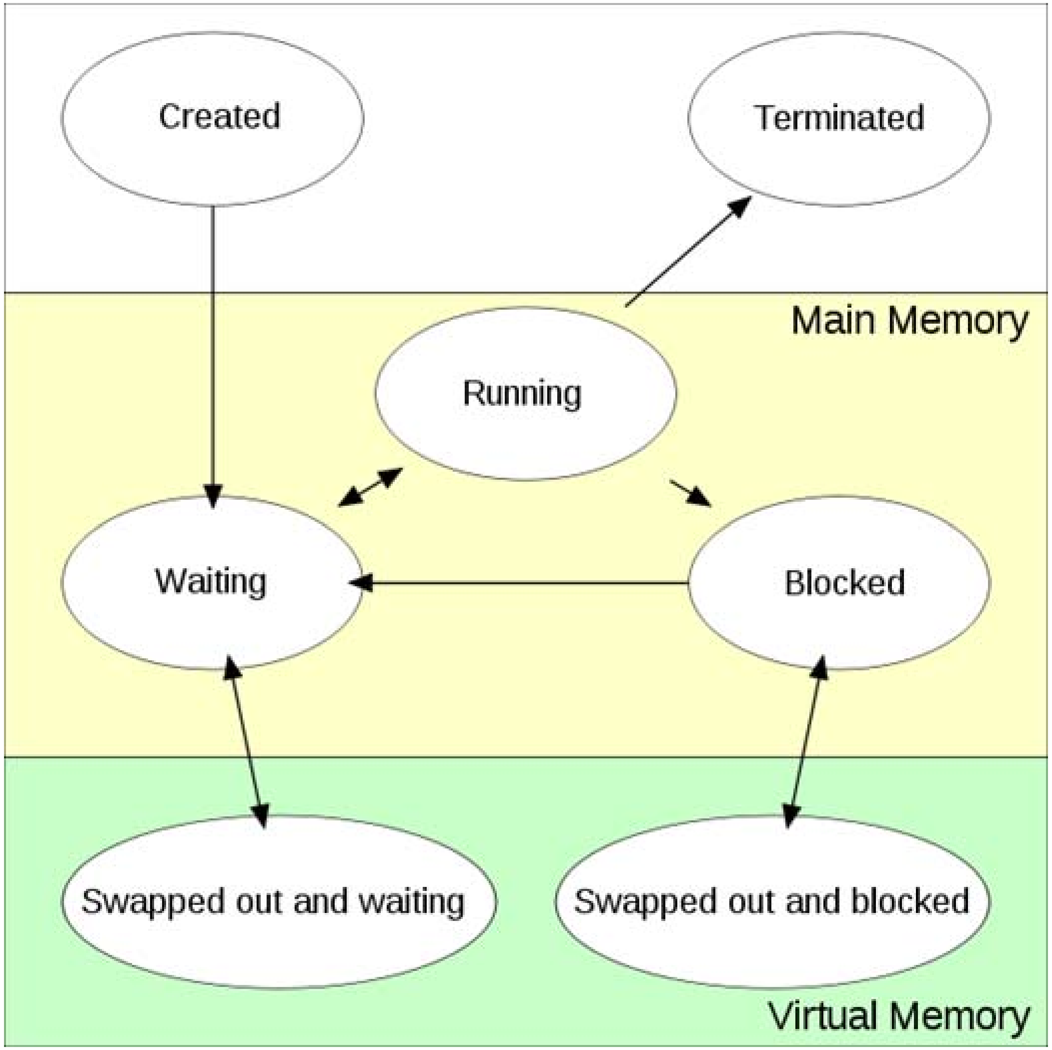
1. Threads → a thread is a virtual processor; a module in execution
   1. Multiplexes a physical processor
   2. The state of a thread: (1) the reference to the next computational step (the Pc register) + (2) the environment (registers, stack, heap, current objects).
   3. Sequence of operations:
      1. Load the module's text
      2. Create a thread and lunch the execution of the module in that thread.
   4. A module may have several threads.
   5. The thread manager implements the thread abstraction.
      1. Interrupts → processed by the interrupt handler which interacts with the thread manager
      2. Exception → interrupts caused by the running thread and processed by exception handlers
      3. Interrupt handlers run in the context of the OS while exception handlers run in the context of interrupted thread.

Basic primitives for processor virtualization

Memory	CREATE/DELETE_ADDRESS SPACE ALLOCATE/FREE_BLOCK MAP/UNMAP UNMAP	
Interpreter	ALLOCATE_THREAD EXIT_THREAD AWAIT TICKET ACQUIRE	DESTROY_THREAD YIELD ADVANCE RELEASE
Communication channel	ALLOCATE/DEALLOCATE_BOUNDED_BUFFER SEND/RECEIVE	


Thread Layer
Thread 1
ID
SP
PC
PMAP
Thread 2
ID
SP
PC
PMAP
Thread 7
ID
SP
PC
PMAP
Processor Layer
Processor A
ID
SP
PC
PMAP
Processor B
ID
SP
PC
PMAP

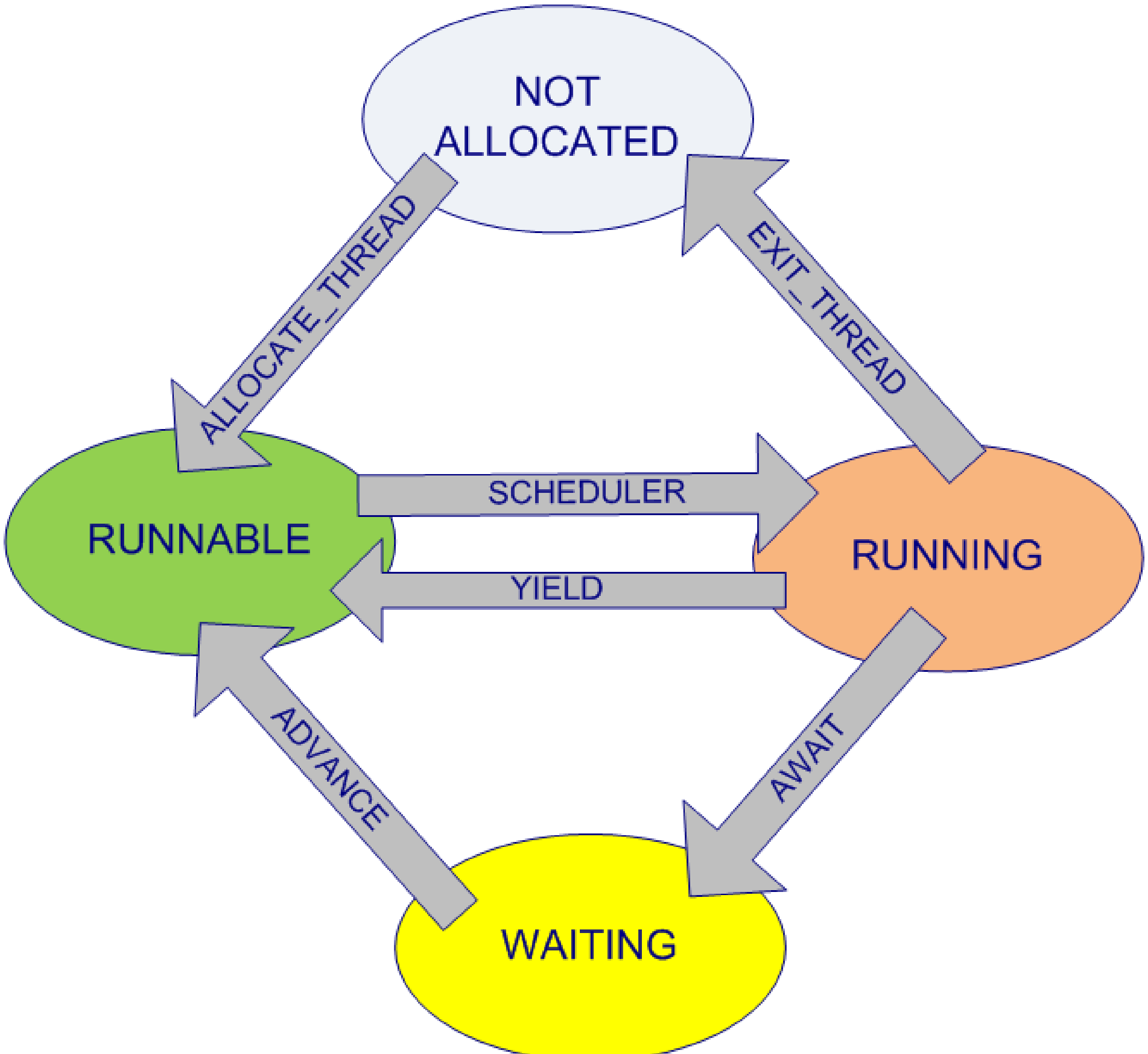
The state of a thread and its associated virtual address space



Processor sharing

- Possible because threads spend a significant percentage of their lifetime waiting for external events.
- Called:
  - Time-sharing
  - Processor multiplexing
  - Multiprogramming
  - Multitasking
- The kernel must support a number of functions:
  - Creation and destruction of threads
  - Allocation of the processor to a ready to run thread
  - Handling of interrupts
  - Scheduling – deciding which one of the ready to run threads should be allocated the processor

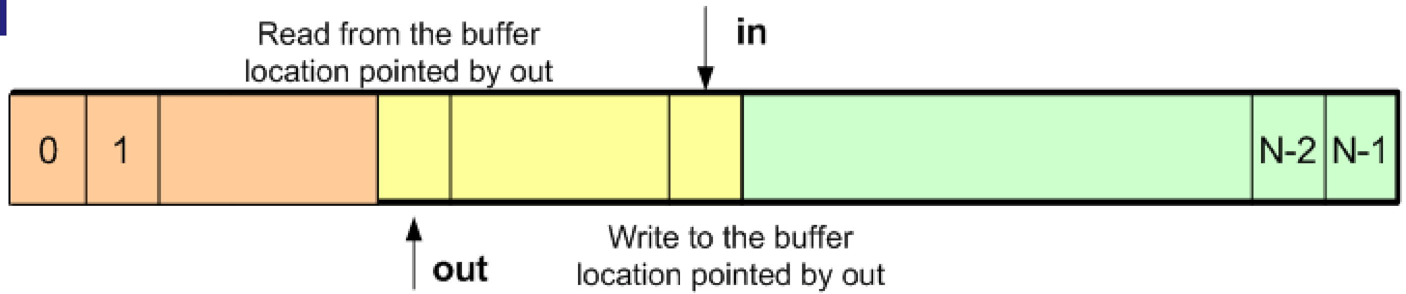
Thread states and state transitions



Switching the processor from one thread to another

- Thread creation:

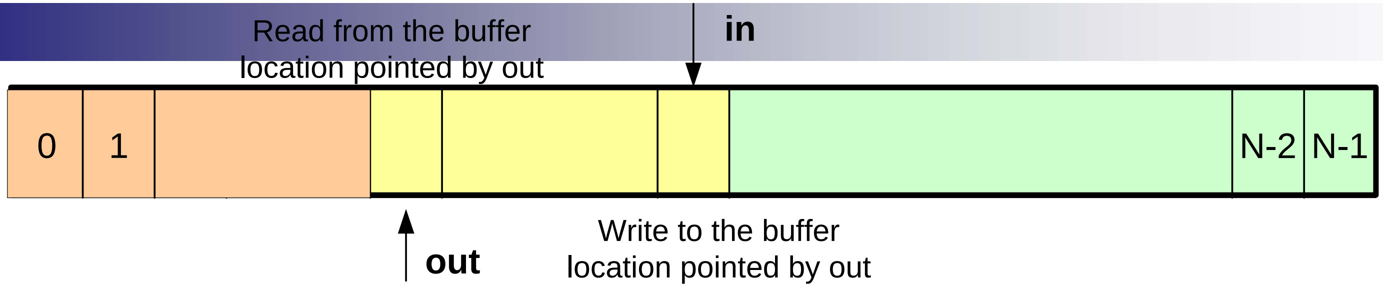
  thread_id ←ALLOCATE_THREAD(starting_address_of_procedure, address_space_id);
- YIELD → function implemented by the kernel to allow a thread to wait for an event.
  - Save the state of the current thread
  - Schedule another thread
  - Start running the new thread – dispatch the processor to the new thread
- YIELD
  - cannot be implemented in a high level language, must be implemented in the machine language.
  - can be called from the environment of the thread, e.g., C, C++, Java
  - allows several threads running on the same processor to wait for a lock. It replaces the busy wait we have used before.




```
shared structure buffer
    message instance message[N]
    integer in initially 0
    integer out initially 0
    lock instance buffer_lock initially UNLOCKED

procedure SEND (buffer reference p, message instance msg)
    ACQUIRE (p_buffer_lock)
    while p.in – p.out = N do                 /* if buffer full wait
        RELEASE (p_buffer_lock)
        ACQUIRE (p_buffer_lock)
    p.message [p.in modulo N] ←msg            /* insert message into buffer cell
    p.in  ← p.in + 1                          /* increment pointer to next free cell
    RELEASE (p_buffer_lock)

procedure RECEIVE (buffer reference p)
    ACQUIRE (p_buffer_lock)
    while p.in = p.out   do                   /* if buffer empty wait for message
        RELEASE (p_buffer_lock)
        ACQUIRE (p_buffer_lock)
    msg← p.message [p.in modulo N]            /* copy message from buffer cell
    p.out ← p.out + 1                         /* increment pointer to next message
    return msg
```




```
shared structure buffer
    message instance message[N]
    integer in initially 0
    integer out initially 0
    lock instance buffer_lock initially UNLOCKED

procedure SEND (buffer reference p, message instance msg)
    ACQUIRE (p_buffer_lock)
    while p.in – p.out = N do                 /* if buffer full wait
        RELEASE (p_buffer_lock)
        YIELD()
        ACQUIRE (p_buffer_lock)
    p.message [p.in modulo N] ←msg            /* insert message into buffer cell
    p.in  ← p.in + 1                          /* increment pointer to next free cell
    RELEASE (p_buffer_lock)

procedure RECEIVE (buffer reference p)
    ACQUIRE (p_buffer_lock)
    while p.in = p.out   do                   /* if buffer empty wait for message
        RELEASE (p_buffer_lock)
        YIELD()
        ACQUIRE (p_buffer_lock)
    msg← p.message [p.in modulo N]            /* copy message from buffer cell
    p.out ← p.out + 1                         /* increment pointer to next message
    return msg
```

Implementation of YIELD

Thread table

Thread Id	Saved SP	State
0	X'2000'	RUNNABLE
1	X'10000'	RUNNING
4	X'10000'	RUNNING
6	X'4000'	RUNNABLE

Processor Table

Processor Id	Thread ID
Processor A	1
Processor B	4

```
shared structure processor_table(7)
        integer thread_id
shared structure thread_table(7)
        integer topstack
        integer state
shared lock instance thread_table_lock

procedure GET_THREAD_ID() return processor_table(CPUID).thread_id

procedure YIELD()
    ACQUIRE (thread_table_lock)
    ENTER_PROCESSOR_LAYER(GET_THREAD_ID())
    RELEASE(thread_table_lock)
return

procedure ENTER_PROCESSOR_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 (mod 7)
    RELEASE(thread_table_lock)
     while thread_table(j).state¬= RUNNABLE
           thread_table(j).state ← RUNNING
           processor_table(CPUID).thread_id ← j
           EXIT_PROCESSOR_LAYER(j)
          return

procedure EXIT_PROCESSOR_LAYER(new)
  SP,-- thread_table(new).topstack
 return
```

More about thread creation and termination

- What if want to create/terminate threads dynamically → we have to:
  - Allow a tread to self-destroy and clean-up -> EXIT_THREAD
  - Allow a thread to terminate another thread of the same application →DESTRY_THREAD
- What if no thread is able to run →
  - create a dummy thread for each processor called a **processor_thread** which is scheduled to run when no other thread is available
  - the **processor_thread** runs in the thread layer
  - the SCHEDULER runs in the processor layer
  - The procedure followed when a kernel starts

Procedure RUN_PROCESSORS()

for each processor **do**

allocate stack and setup processor thread /*allocation of the stack done at processor layer

shutdown ← FALSE

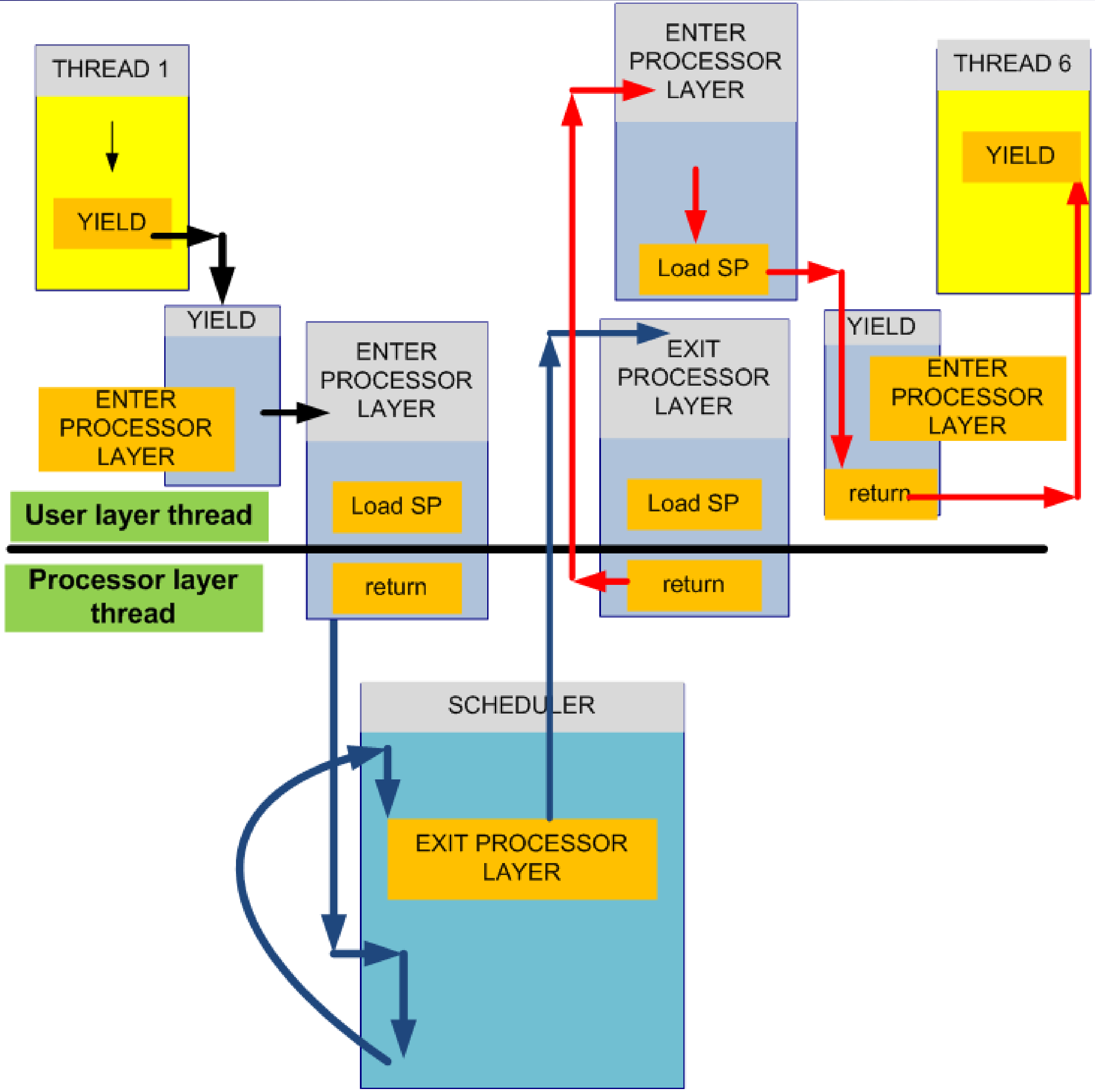
SCHEDULER()

deallocate processor_thread stack /*deallocation of the stack done at processor layer

halt processor

Switching threads with dynamic thread creation

- Switching from one user thread to another requires two steps
  - Switch from the thread releasing the processor to the processor thread
  - Switch from the processor thread to the new thread which is going to have the control of the processor
  - The last step requires the SCHEDULER to circle through the **thread_table** until a thread ready to run is found
- The boundary between user layer threads and processor layer thread is crossed twice
- Example: switch from thread 1 to thread 6 using
  - YIELD
  - ENTER_PROCESSOR_LAYER
  - EXIT_PROCESSOR_LAYER


THREAD 1
YIELD
YIELD
ENTER PROCESSOR LAYER
ENTER PROCESSOR LAYER
Load SP
return
User layer thread
Processor layer thread
ENTER PROCESSOR LAYER
Load SP
EXIT PROCESSOR LAYER
Load SP
return
THREAD 6
YIELD
YIELD
ENTER PROCESSOR LAYER
return
SCHEDULER
EXIT PROCESSOR LAYER

```
shared structure processor_table(7)
     integer topstack
     byte reference stack
     integer thread_id
shared structure thread_table(7)
     integer topstack
     integer state
     boolean kill_pr_continue
     byte reference stack


shared lock instance thread_table_lock

procedure GET_THREAD_ID() return processor_table(CPUID).thread_id
```

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

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) then
                        thread_table(j).state¬= RUNNABLE
                         thread_table(j).state ← FREE
                        DEALLOCATE(thread_table(i).stack)
                        thread_table(i).kill_or_continue ← CONTINUE
       RELEASE(thread_table_lock)
return


procedure ENTER_PROCESSOR_LAYER(thread_id, processor)
     thread_table(thread_id).state ← RUNNABLE
     thread_table(thread_id).topstack ← SP
     SCHEDULER()
return


procedure EXIT_PROCESSOR_LAYER(processor,thread_id)
  processor_table(processor).topstack ←SP
  SP,-- thread_table(thread_id).topstack
 return
```