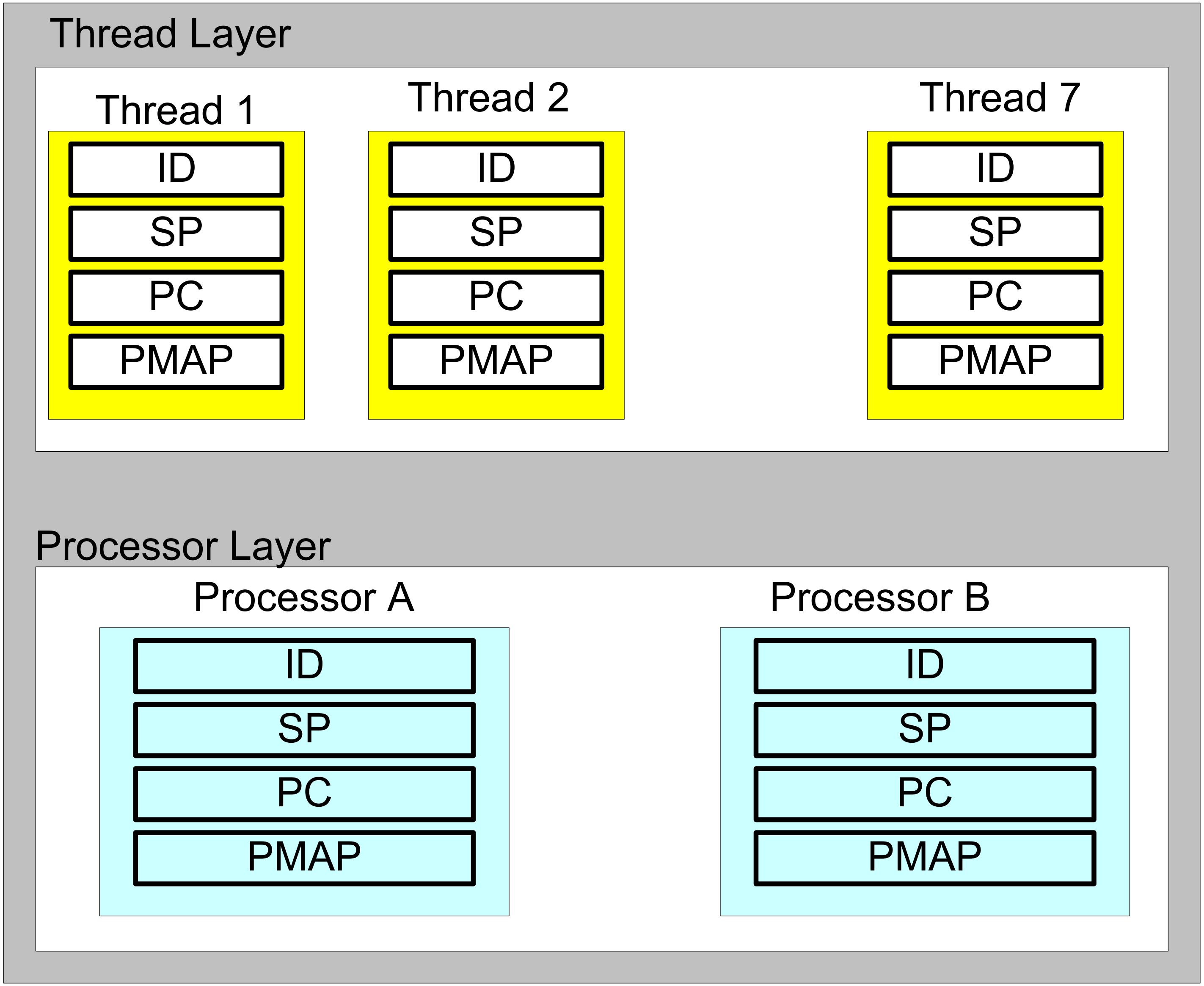
COT 4600 Operating Systems Fall 2009

Dan C. Marinescu
Office: HEC 439 B
Office hours: Tu-Th 3:00-4:00 PM

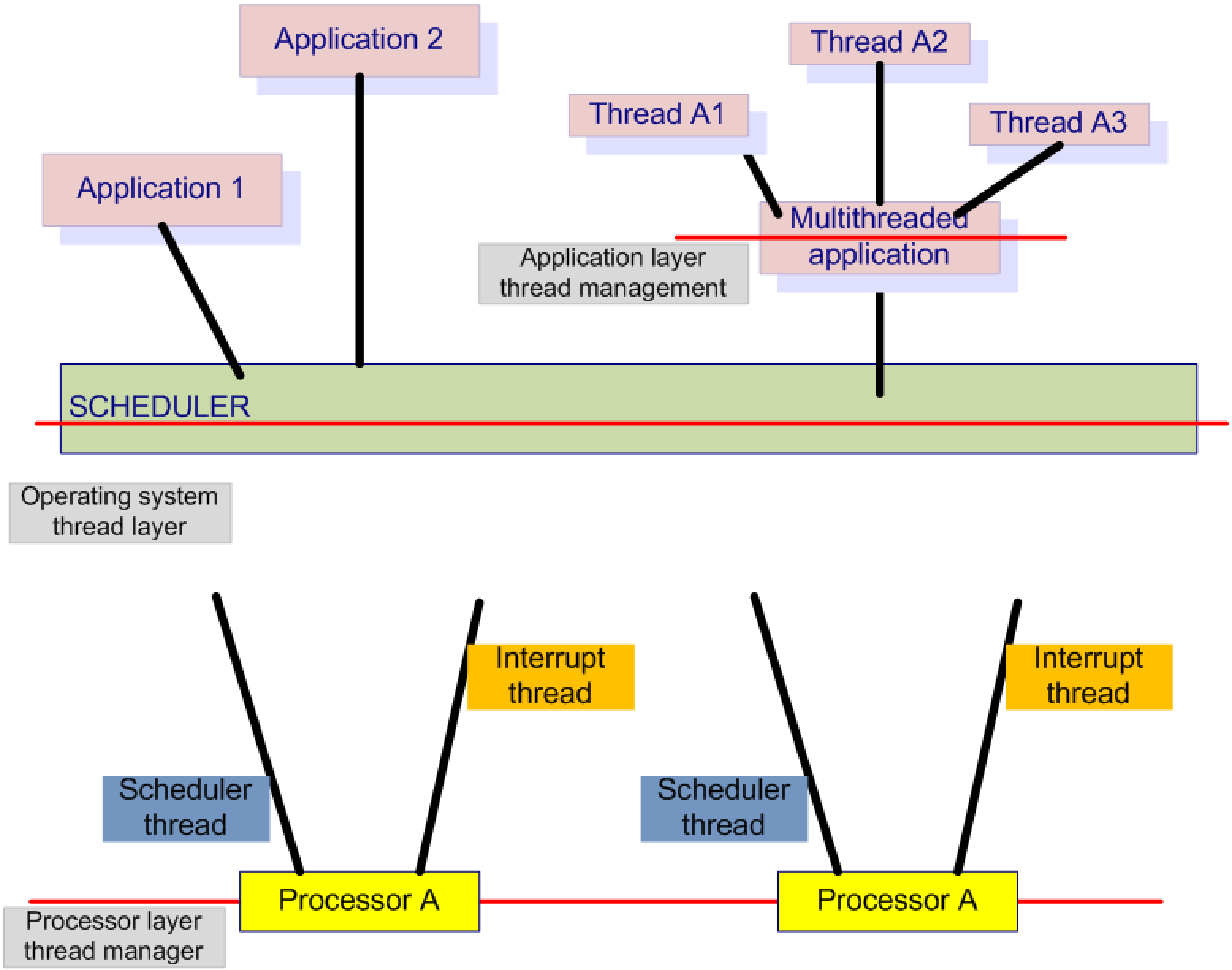
Lecture 20

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


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

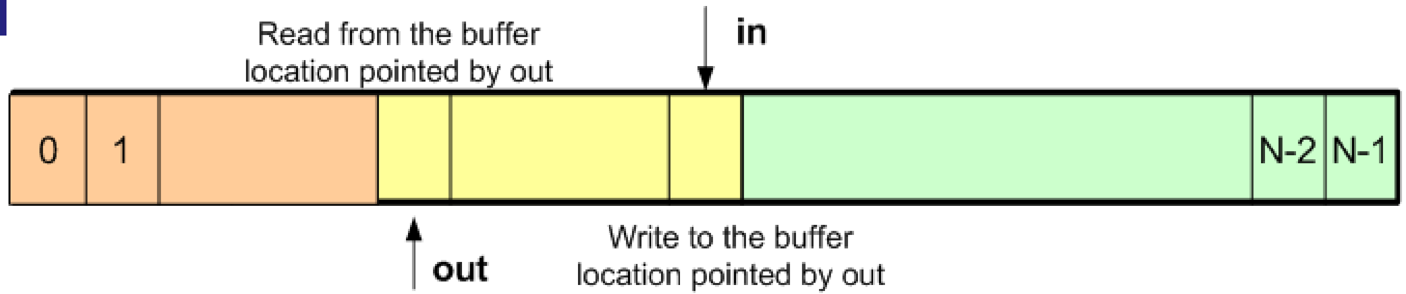
Thread scheduling policies

- Non-preemptive scheduling → a running thread releases the processor at its own will. Not very likely to work in a greedy environment.
- Cooperative scheduling → a thread calls YIEALD periodically
- Preemptive scheduling → a thread is allowed to run for a time slot. It is enforced by the thread manager working in concert with the interrupt handler.
  - The interrupt handler should invoke the thread exception handler.
  - What if the interrupt handler running at the processor layer invokes directly the thread? Imagine the following sequence:
    - Thread A acquires the *thread_table_lock*
    - An interrupt occurs
    - The YIELD call in the interrupt handler will attempt to acquire the *thread_table_lock*
- Solution: the processor is shared between two threads:
  - The processor thread
  - The interrupt handler thread
- Recall that threads have their individual address spaces so the scheduler when allocating the processor to thread must also load the page map table of the thread into the page map table register of the processor

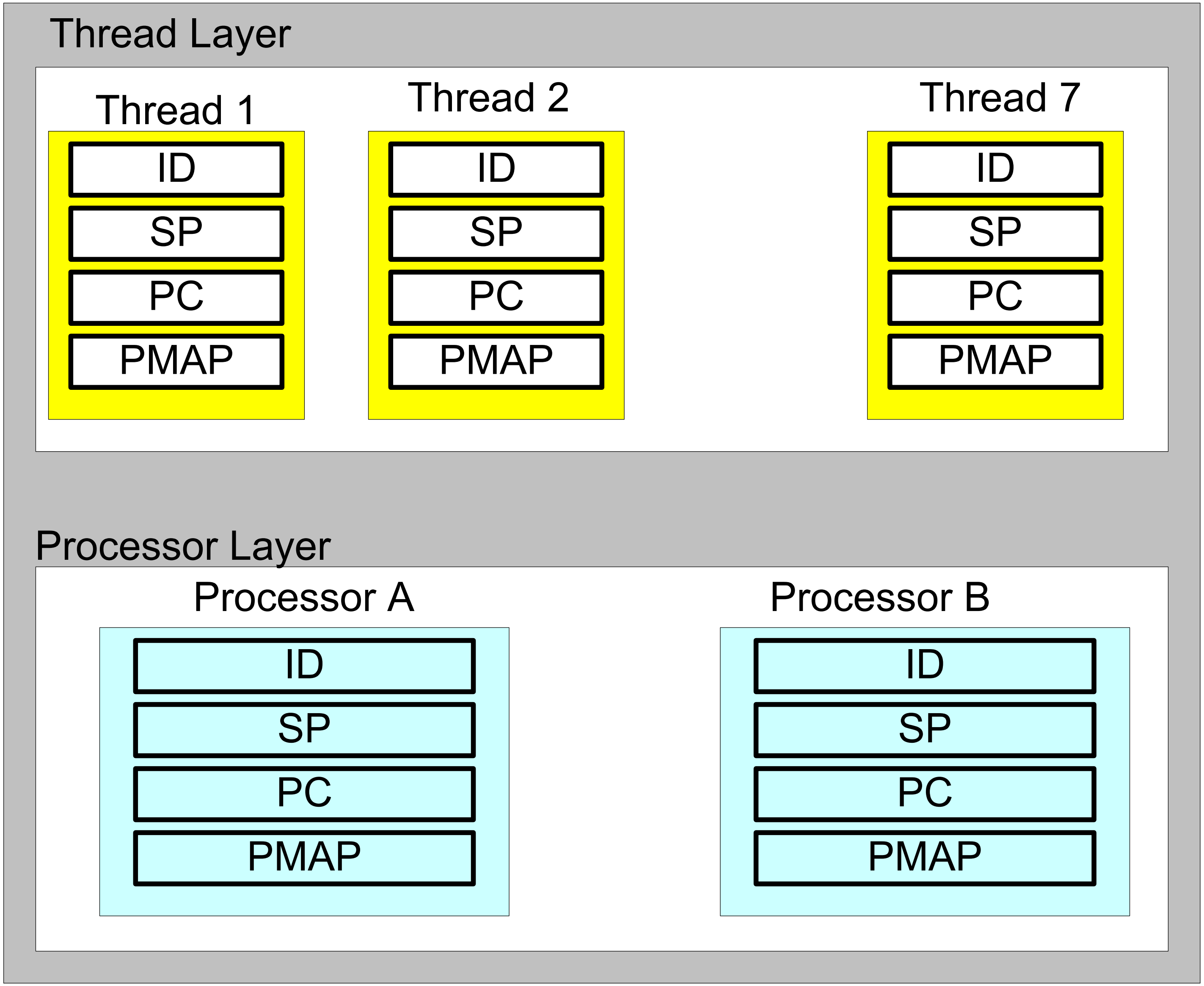

Application 2
Thread A2
Thread A1
Thread A3
Application 1
Multithreaded application
Application layer thread management
SCHEDULER
Operating system thread layer
Interrupt thread
Interrupt thread
Scheduler thread
Scheduler thread
Processor A
Processor A
Processor layer thread manager

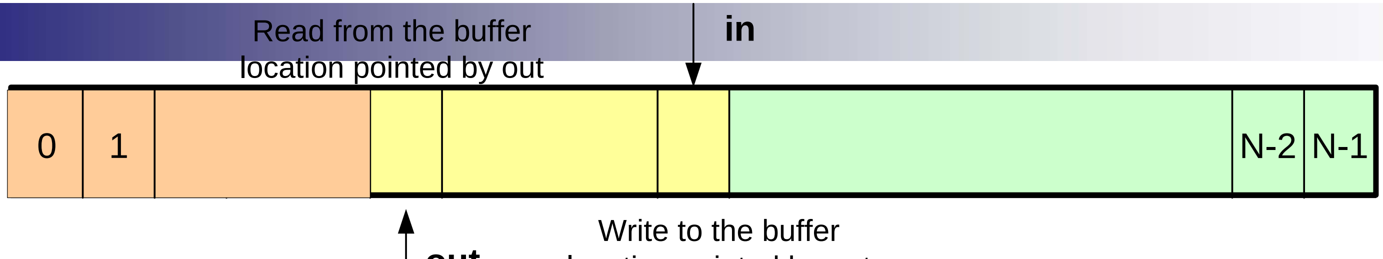
Evolution of ideas regarding communication among threads using a bounded buffer

1. Use locks → did not address the busy waiting problem
2. YIELD → based on voluntary release of the processor by individual threads
3. Use WAIT (for an event) and NOTIFY (when the event occurs) primitives .
4. Use AWAIT (for an event) and ADVANCE (when the event occurs)



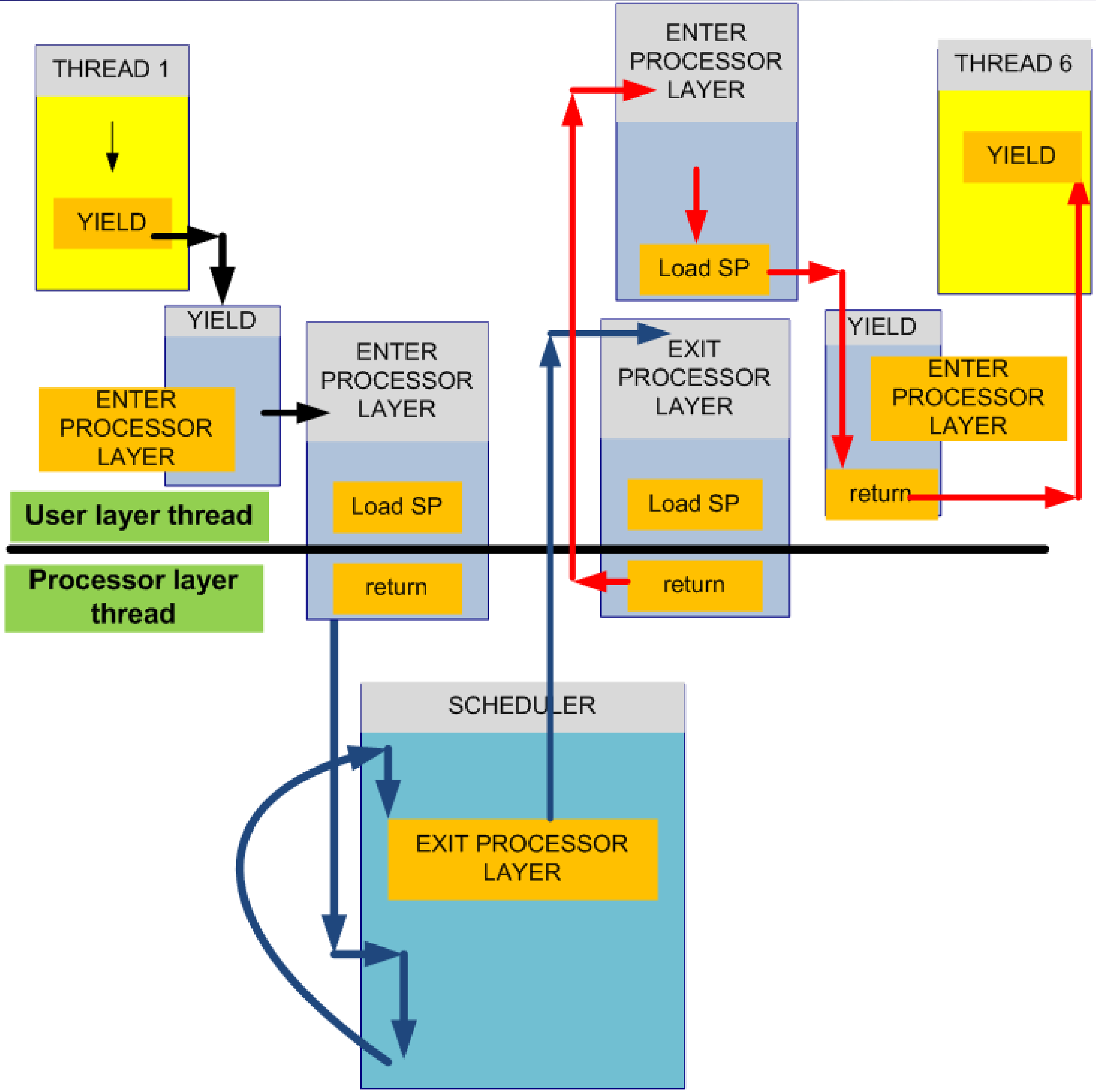

```
shared structure buffer
    message instance message[N]
    integer in initially 0
    integer out initially 0
    lock instance buffer_lock initially UNLOCKED

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
```


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




```
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
          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
     RELEASE (p_buffer_lock)
     return msg
```

```
shared structure processor_table(2)
        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)
        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
```



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

```
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
```

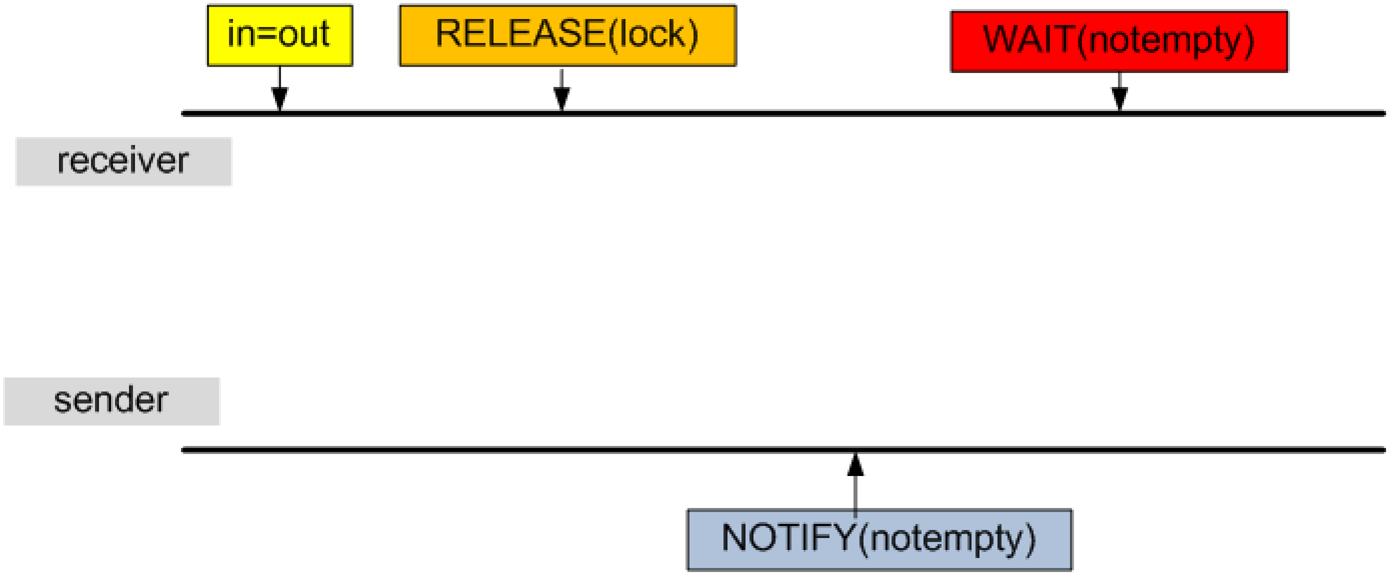
Primitives for thread sequence coordination

- YIELD requires the thread to periodically check if a condition has occurred.
- Basic idea → use events and construct two before-or-after actions
  - **WAIT***(event_name)* → issued by the thread which can continue only after the occurrence of the event *event_name*.
  - **NOTIFY***(event_name)* → search the *thread_table* to find a thread waiting for the occurrence of the event *event_name*.

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


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)
        WAIT (p.room)
        ACQUIRE (p_buffer_lock)
    p.message [p.in modulo N] ←msg             /* insert message into buffer cell
    if p.in= p.out  then NOTIFY(p.notempty)
    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)
        WAIT (p.notempty)
        ACQUIRE (p_buffer_lock)
    msg← p.message [p.in modulo N]             /* copy message from buffer cell
    if (p.in- p.out=N)  then NOTIFY(p.room)
    p.out ← p.out + 1                          /* increment pointer to next message
    RELEASE (p_buffer_lock)
    return msg
```

This solution does not work

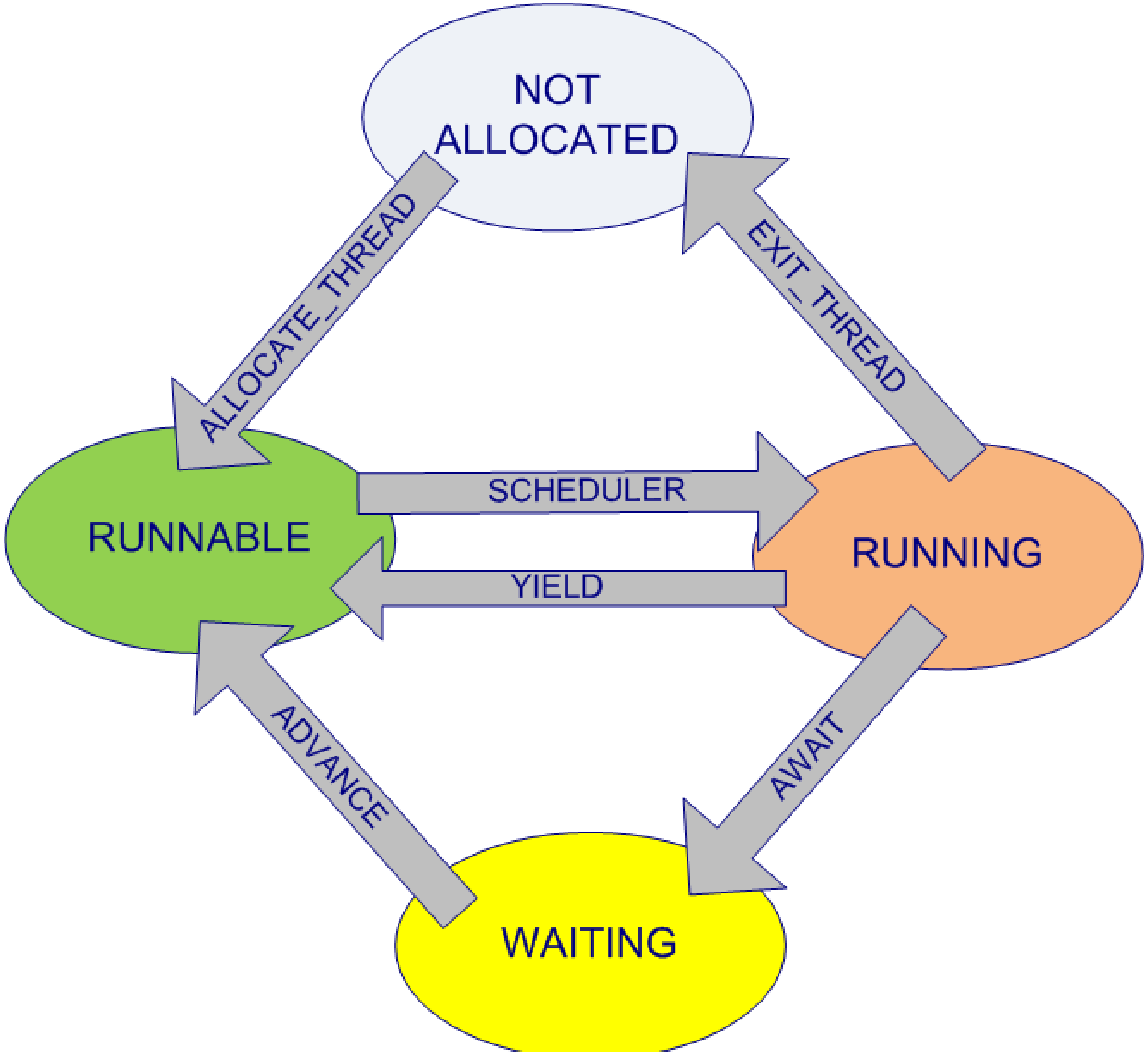
- The NOTIFY should always be sent after the WAIT. If the **sender** and the **receiver** run on two different processor there could be a race condition for the *notempty* event. The NOTIFY could be sent before the WAIT.
- Tension between modularity and locks
- Several possible solutions: AWAIT/ADVANCE, semaphores, etc



AWAIT - ADVANCE solution

- A new state, WAITING and two before-or-after actions that take a RUNNING thread into the WAITING state and back to RUNNABLE state.
- eventcount → variables with an integer value shared between threads and the thread manager; they are like events but have a value.
- A thread in the WAITING state waits for a particular value of the *eventcount*
- AWAIT(*eventcount,value*)
  - If *eventcount >value* ➔ the control is returned to the thread calling AWAIT and this thread will continue execution
  - If *eventcount ≤value* ➔ the state of the thread calling AWAIT is changed to WAITING and the thread is suspended.
- ADVANCE(*eventcount*)
  - increments the *eventcount* by one then
  - searches the *thread_table* for threads waiting for this *eventcount*
  - if it finds a thread and the eventcount exceeds the value the thread is waiting for then the state of the thread is changed to RUNNABLE

Thread states and state transitions



Solution for a single sender and multiple receivers

```
shared structure buffer
    message instance message[N]
    eventcount instance in initially 0
    eventcount instance out initially 0


procedure SEND (buffer reference p, message instance msg)
   AWAIT (p.out,p.in-N )
   p.message [p.in modulo N] ←msg        /* insert message into buffer cell
   ADVANCE (p.in)

procedure RECEIVE (buffer reference p)
      AWAIT (p.in,p.out)
      msg← p.message [p.in modulo N]     /* copy message from buffer cell
      ADVANCE (p.out)
      return msg
```

Supporting multiple senders: the sequencer

- Sequencer→ shared variable supporting thread sequence coordination -it allows threads to be ordered and is manipulated using two before-or-after actions.
- TICKET(*sequencer*) ➔ returns a negative value which increases by one at each call. Two concurrent threads calling TICKET on the same sequencer will receive different values based upon the timing of the call, the one calling first will receive a smaller value.
- READ(*sequencer*) ➔ returns the current value of the *sequencer*

Multiple sender solution; only the SEND must be modified

```
shared structure buffer
    message instance message[N]
    eventcount instance in initially 0
    eventcount instance out initially 0
    sequencer instance sender


procedure SEND (buffer reference p, message instance msg)
   t ← TICKET(p.sender)
   AWAIT (p.in,t)
   AWAIT (p.out,READ(p.in) -N )
   p.message [p.in modulo N] ←msg        /* insert message into buffer cell
   ADVANCE (p.in)
```

Semaphores

- Introduced by Dijkstra in 1965
- Does not require busy waiting
- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations
  - ```
    wait (S) {
            while S <= 0
                ; // no-op
              S--;
      }
    ```
  - ```
    signal (S) {
        S++;
     }
    ```

Semaphore as General Synchronization Tool

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore mutex locks– integer value can range only between 0 and 1; simpler to implement.
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion
  - Semaphore S;   // initialized to 1
  - wait (S);

            Critical Section

     signal (S);

Semaphore Implementation

- Must guarantee that no two threads can execute wait () and signal () on the same semaphore at the same time
- Implementation becomes the critical section problem where the wait and signal code are placed in the critical section.
  - Could now have busy waiting in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Applications may spend lots of time in critical sections and therefore this is not a good solution.

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- The two operations on semphore S, Wait(S) and Signal(S) are implemented using:
  - block – place the thread invoking the operation on the appropriate waiting queue.
  - wakeup – remove one of thread in the waiting queue and place it in the ready queue.

Semaphore Implementation with no Busy waiting

- Implementation of wait:

```
wait (S){
        value--;
        if (value < 0) {
                add this thread to waiting queue
                 block();  }
 }
```

- Implementation of signal:

```
Signal (S){
          value++;
           if (value <= 0) {
             remove a thread P from the waiting queue
                 wakeup(P);  }
}
```