COT 4600 Operating Systems Fall 2009

Dan C. Marinescu

Office: HEC 439 B

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

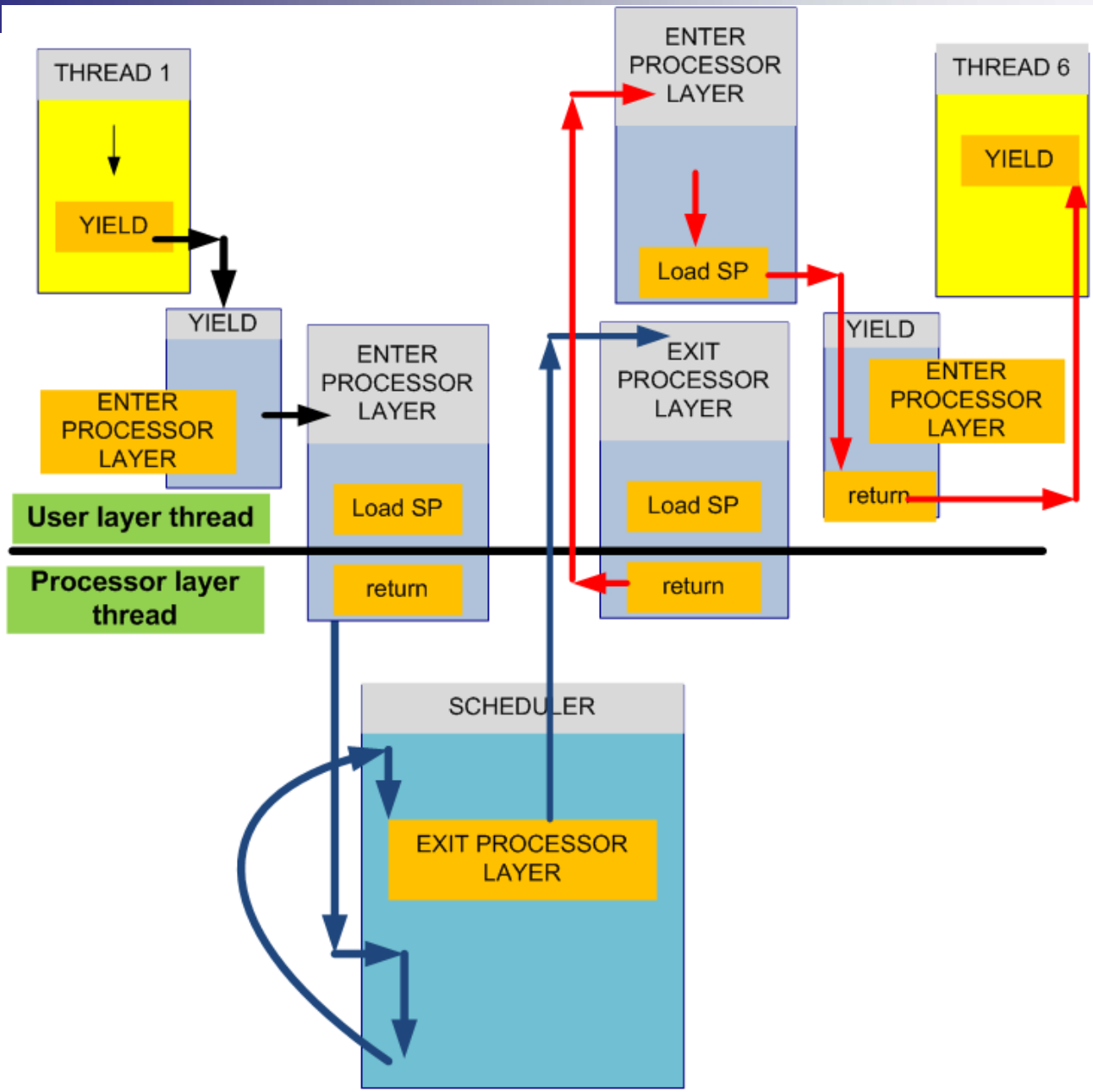
Lecture 21

- Last time:
  - Preemptive scheduling
  - Thread primitives for sequence coordination
- Today:
  - Implementation of AWAIT, ADVANCE, TICKET, and READ
  - Polling and interrupts
  - Evolution of the Intel x86 architecture
  - Virtual Machines
- Next Time:
  - Performance Metrics (Chapter 5)

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

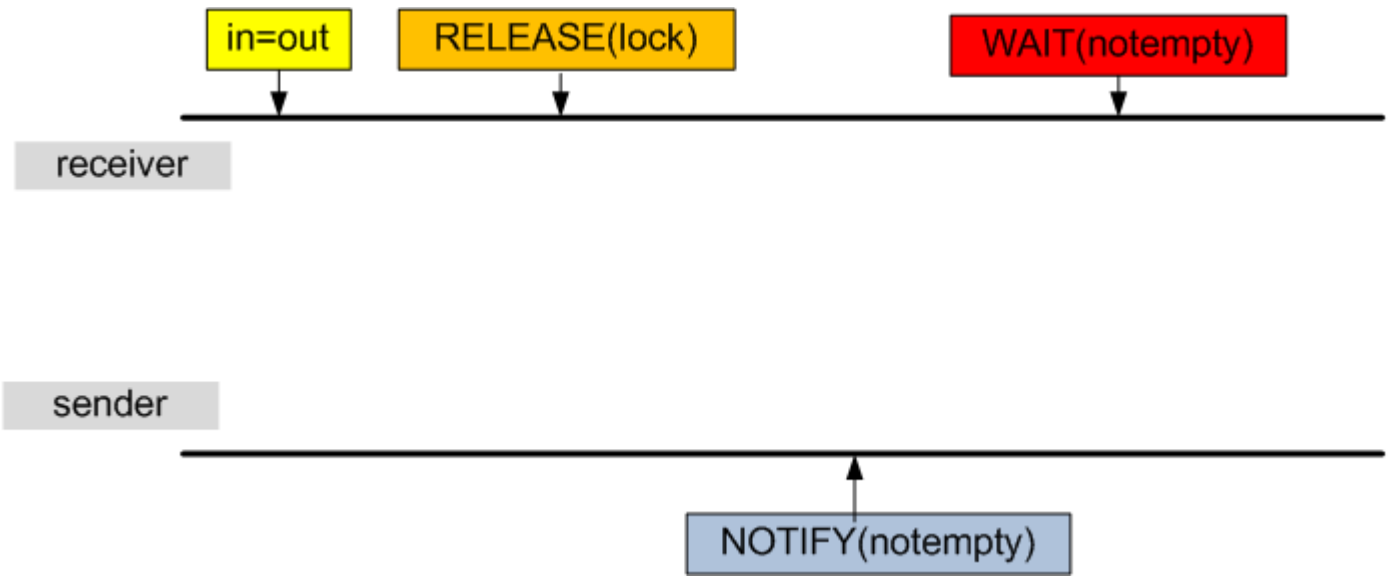
Primitives for thread sequence coordination

- 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

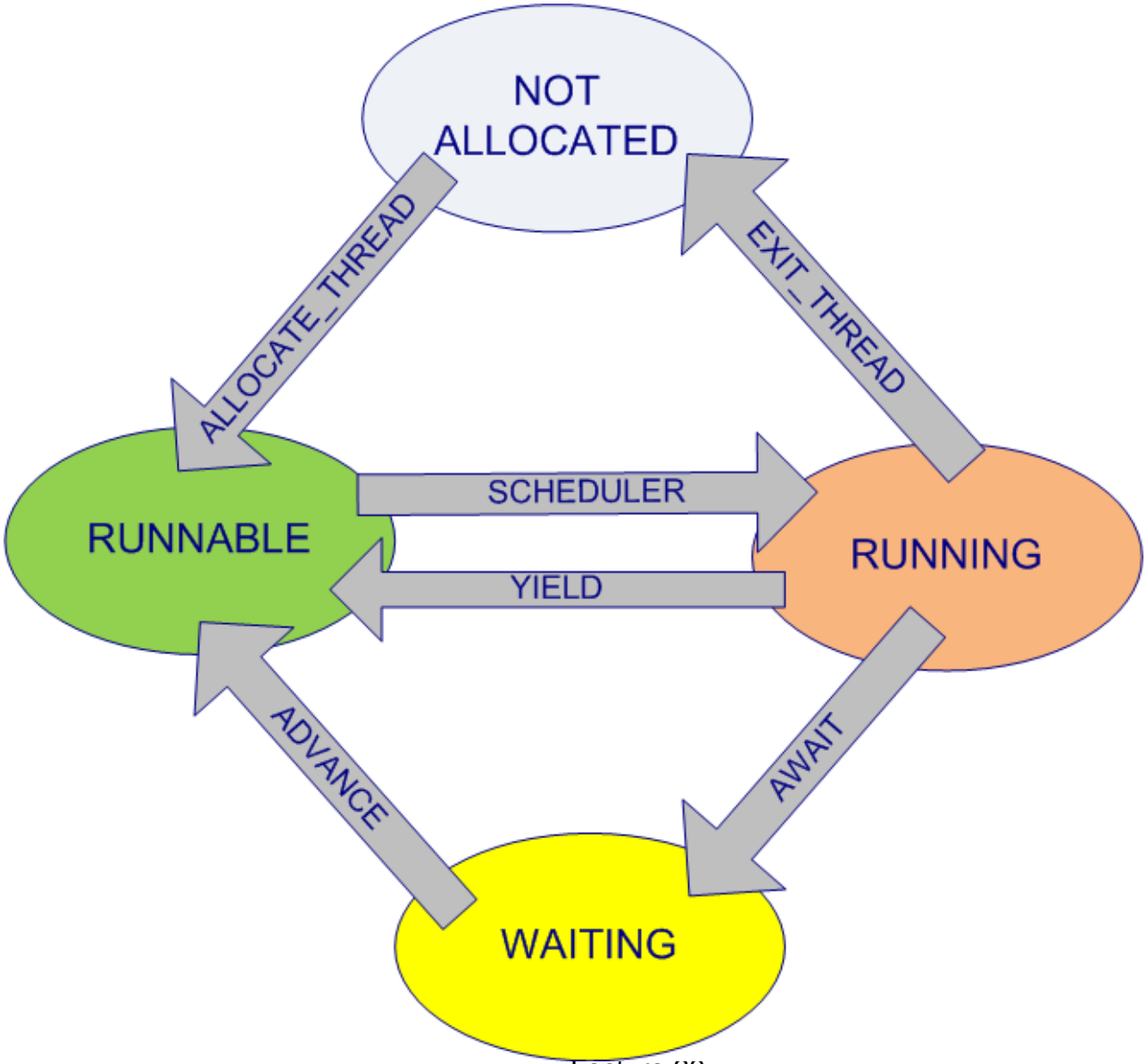
- The NOTIFY should always be sent after the WAIT. If the **sender** and the **receiver** run on two different processor there could be a race condition for the *notempty* event. The NOTIFY could be sent before the WAIT.
- Tension between modularity and locks
- Several possible solutions: AWAIT/ADVANCE,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

```
shared structure thread_table[7]
        integer topstack
        integer state
        eventcount reference event
        long integer value

shared lock instance thread_table_lock

structure everntcount
          long integer count


procedure AWAIT(eventcount reference event, value)
    ACQUIRE (thread_table_lock)
    id ← GET_THREAD_ID()
    thread_table[id].event ← event
    thread_table[id].value ← value
    if event_count <= value then thread_table[id].state ← WAITING
    ENTER_PROCESSOR_LAYER(id,CPUID)
    RELEASE(thread_table_lock)

procedure ADVANCE(eventcount reference event)
     ACQUIRE (thread_table_lock)
     eventcount ← eventcount +1
     for i from 0 until 7 do
          if thread_table[i].state =WAITING  and thread_table[i].event =event and
               event.count > thread_table[i].value then thread_table[i].state ← RUNNABLE
       RELEASE(thread_table_lock)
```

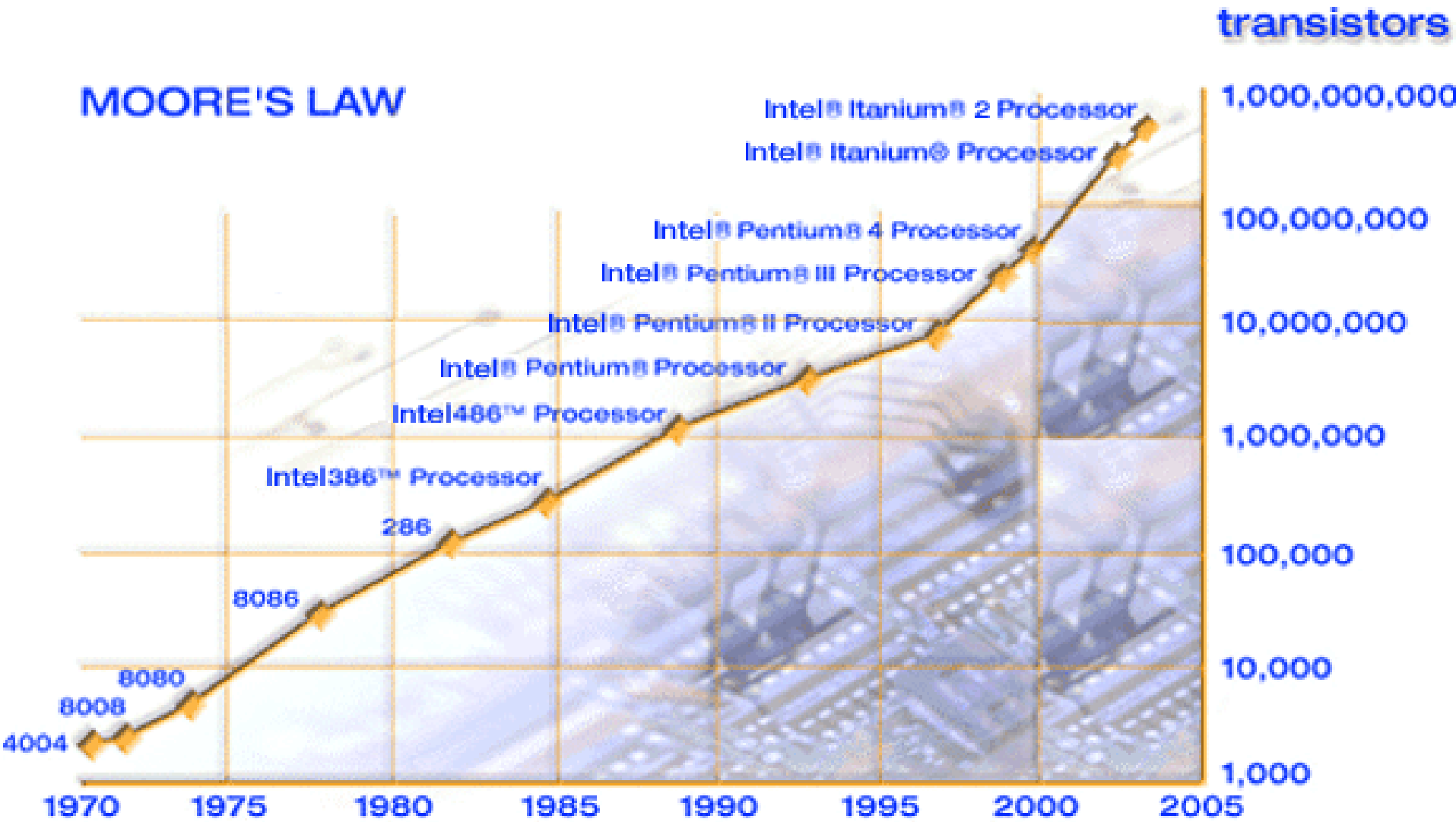
```
structure sequencer
        long integer ticket


procedure TICKET(sequence reference s)
   ACQUIRE (thread_table_lock)
   t ← s.ticket
   s.ticket ← s.ticket + 1
   RELEASE(thread_table_lock)
return t

procedure READ(eventcount reference event)
    ACQUIRE (thread_table_lock)
    e ← event.count
    RELEASE(thread_table_lock)
return
```

Polling and interrupts

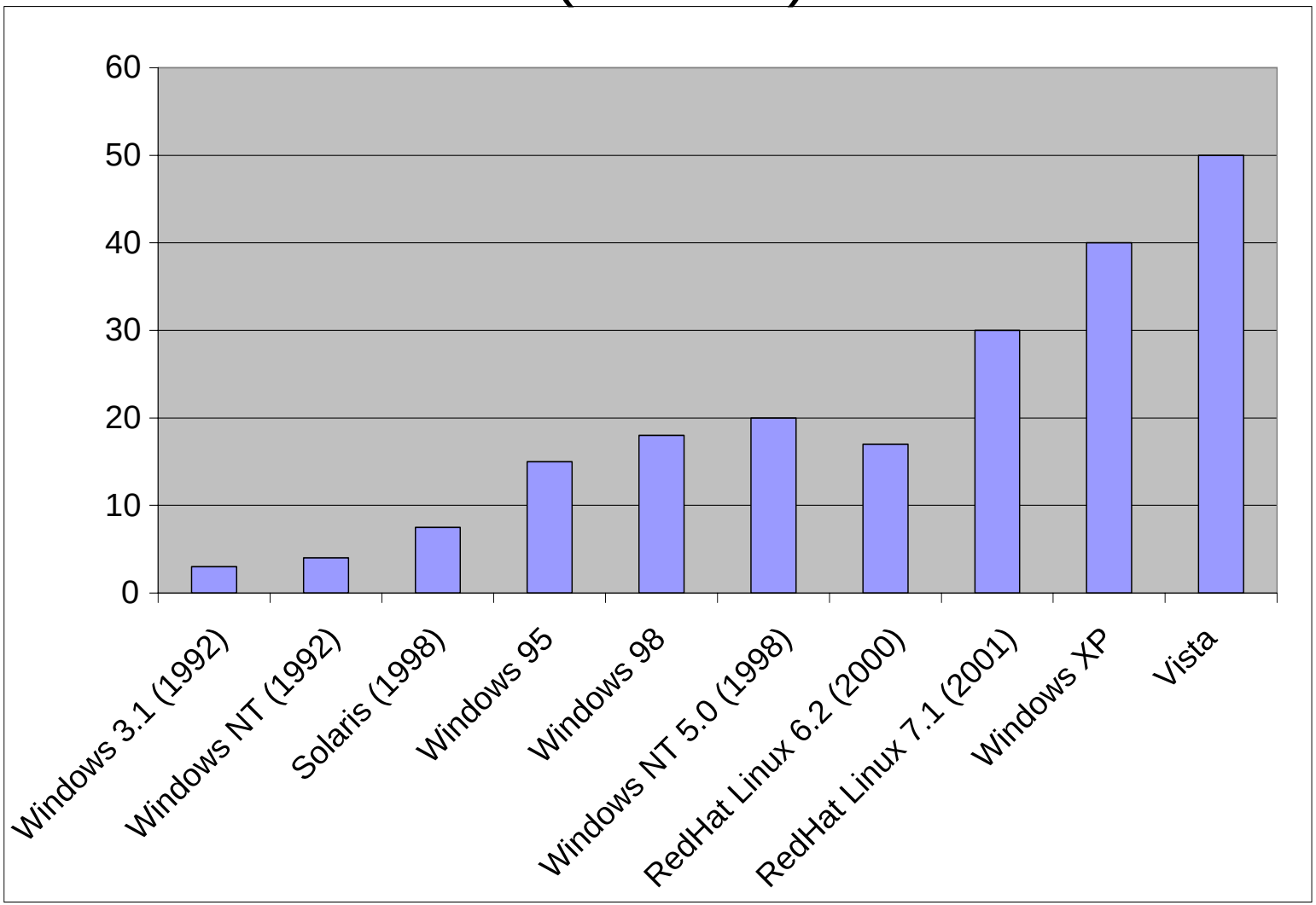
- Polling → periodically checking the status of a subsystem.
  - How often should the polling be done?
    - Too frequently → large overhead
    - After a large time interval → the system will appear non-responsive
- Interrupts
  - could be implemented in hardware as polling → before executing the next instruction the processor checks an “interrupt” bit implemented as a flip-flop
    - If the bit is ON invoke the interrupt handler instead of executing the next instruction
    - Multiple types of interrupts → multiple “interrupts” bits checked based upon the priority of the interrupt.
  - Some architectures allow the interrupts to occur durin the execution of an instruction
- The interrupt handler should be short and very carefully written. Interrupts of lower priority could be masked.

Evolution of modularity for the Intel architecture x86

- The address space size determined by the number of address bits:
  - 24 for 80286 a 16 bit processor → modularity enforced through segmentation
  - 32 for 80386 a 32 bit processor →
    - each segment could have up to 2^{32} bytes
    - within each segment support for virtual memory
- Backward compatibility

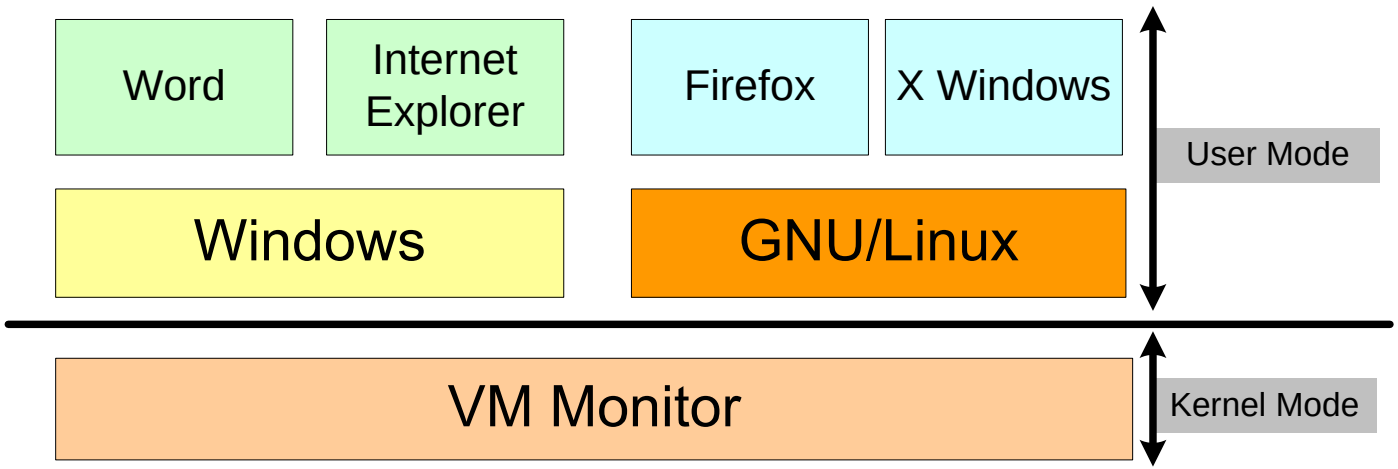

MOORE'S LAW
transistors
Intel® Itanium® 2 Processor
Intel® Itanium® Processor
Intel® Pentium® 4 Processor
Intel® Pentium® III Processor
Intel® Pentium® II Processor
Intel® Pentium® Processor
Intel486™ Processor
Intel386™ Processor
286
8086
8080
8008
4004
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
1,000
1970
1975
1980
1985
1990
1995
2000
2005

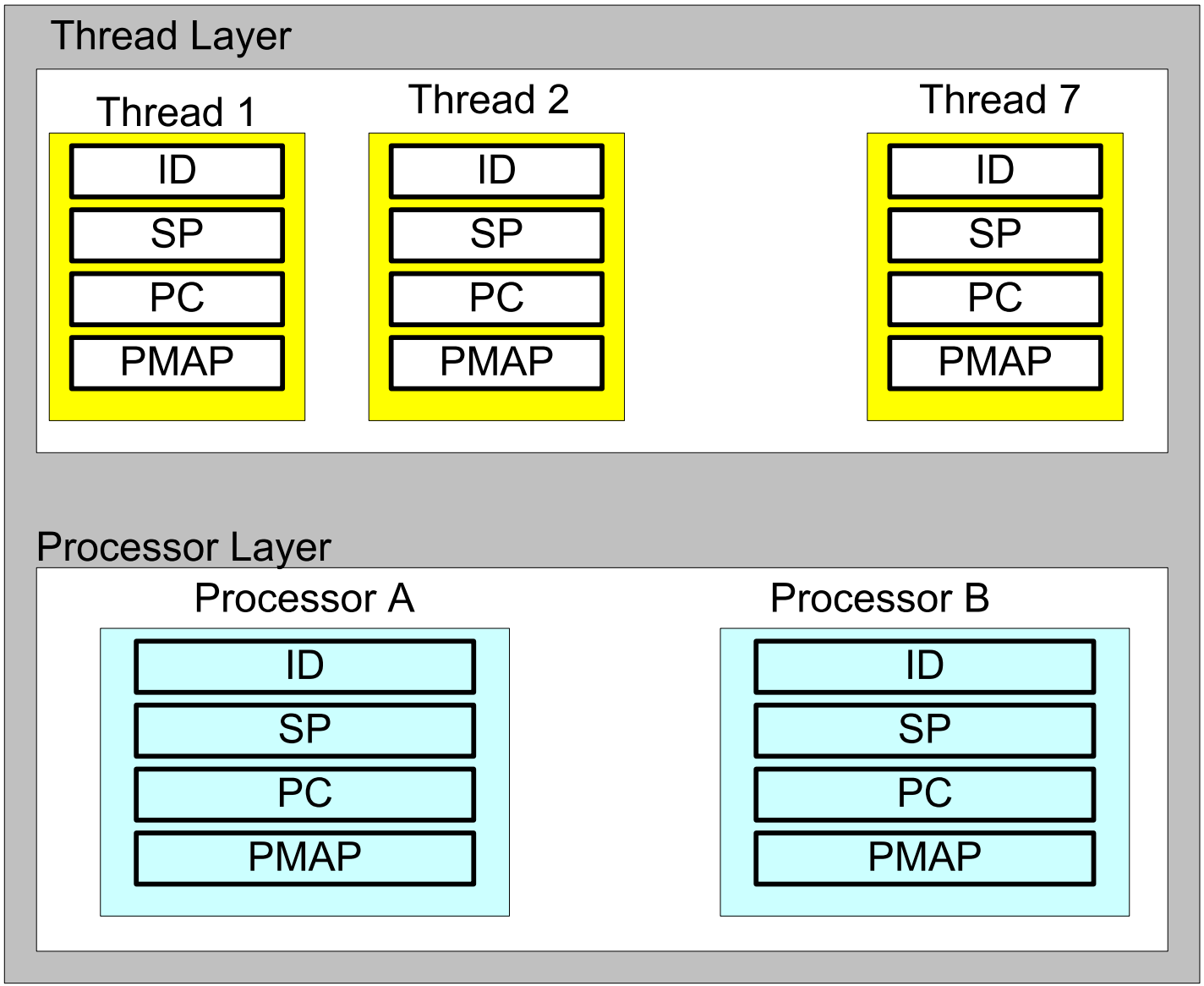
The increase in the number of lines of operating systems source code (millions)

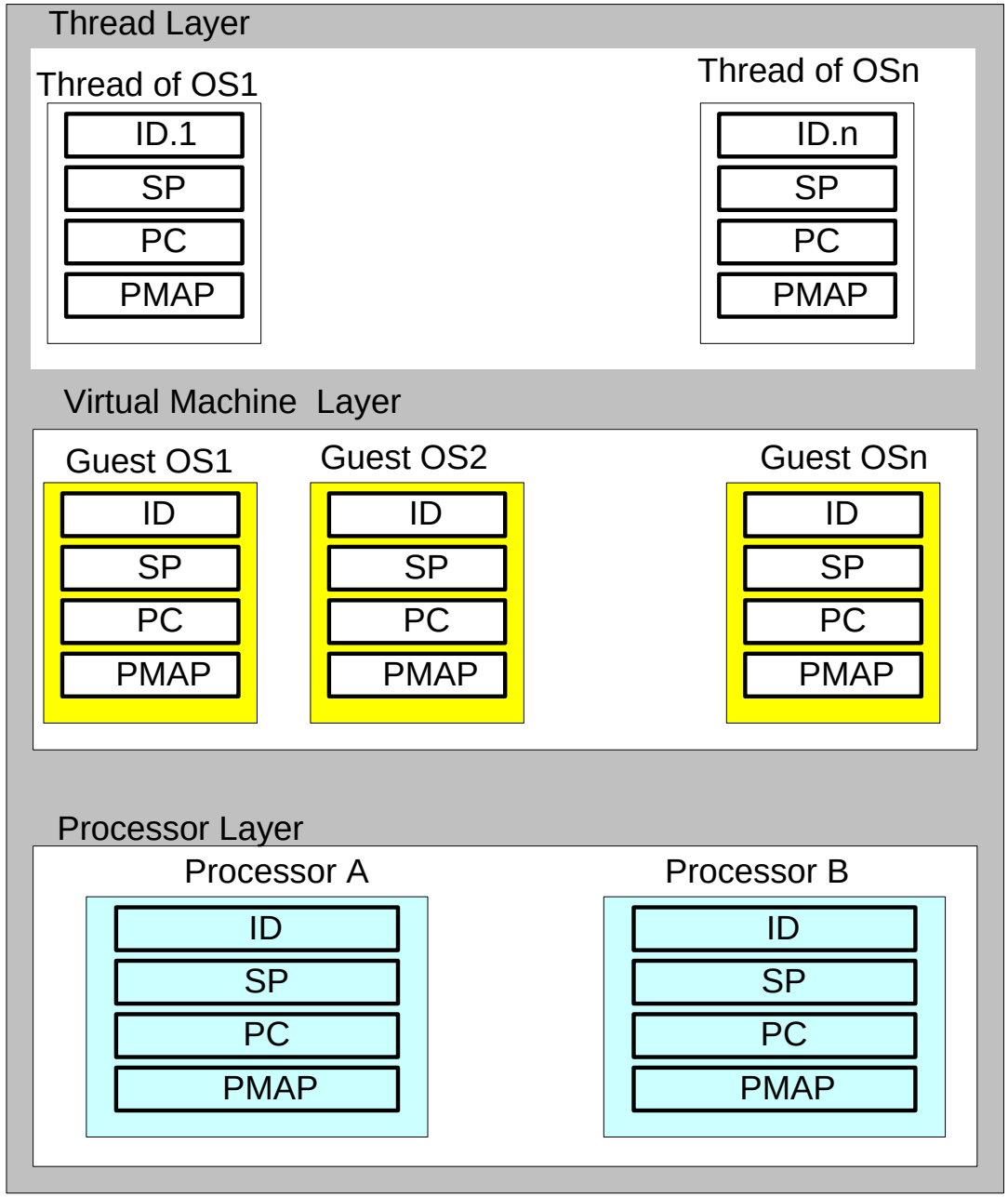


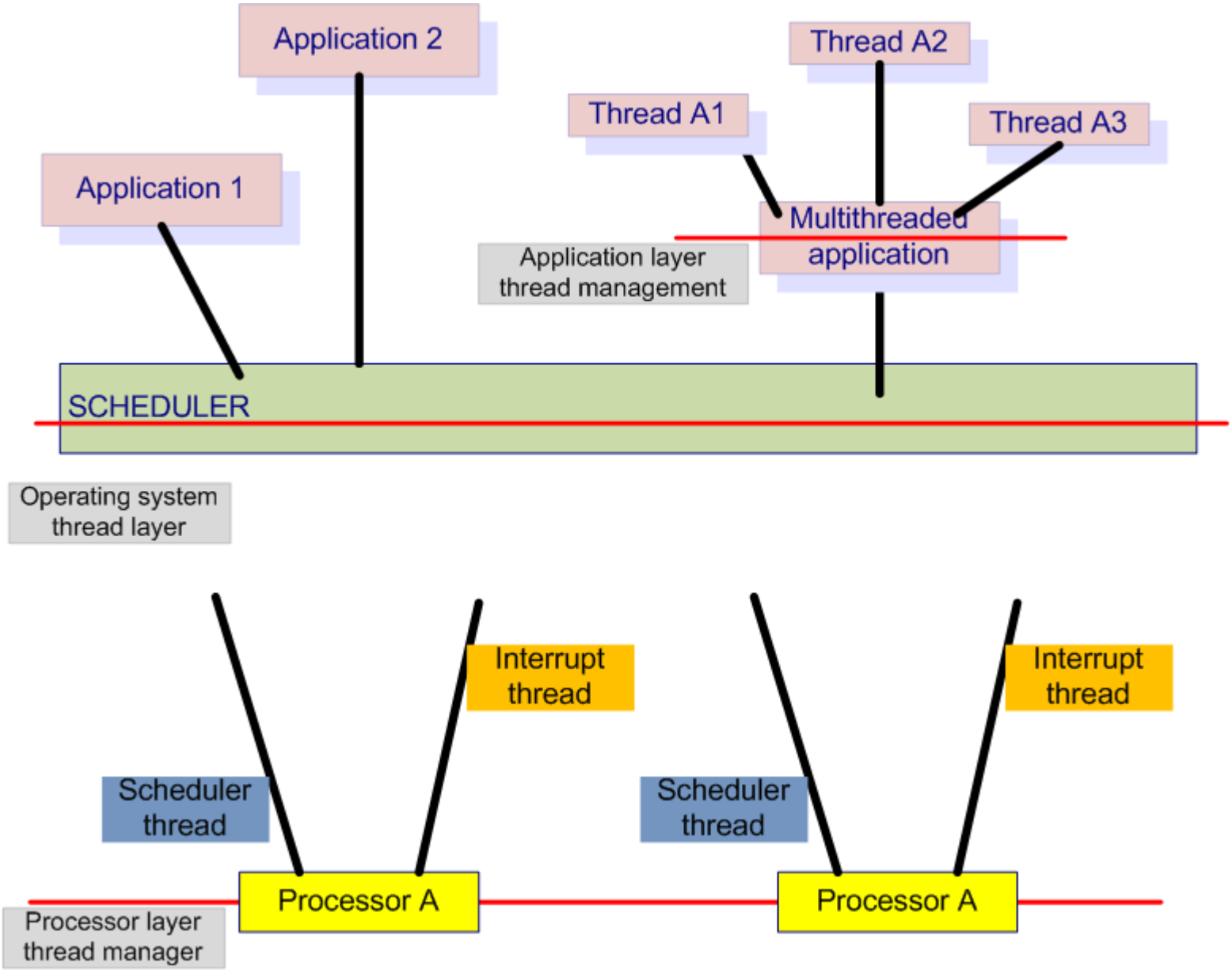
Virtual machines

- First commercial product→ IBM VM 370 originally developed as CP-67
- Advantages:
  - One could run multiple guest operating systems on the same machine
  - An error in one guest operating system does not bring the machine down
  - An ideal environment for developing operating systems




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 Layer
Thread of OS1
ID.1
SP
PC
PMAP
Thread of OSn
ID.n
SP
PC
PMAP
Virtual Machine Layer
Guest OS1
ID
SP
PC
PMAP
Guest OS2
ID
SP
PC
PMAP
Guest OSn
ID
SP
PC
PMAP
Processor Layer
Processor A
ID
SP
PC
PMAP
Processor B
ID
SP
PC
PMAP


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