

Time Sharing Basics

P. J. Denning
For CS471/CS571

© 2001, P. J. Denning

- Time sharing depends on the multiplexing of CPU among many user jobs
- Multiplexing depends on clock interrupts that switch CPU at regular intervals
- Begin with review of how interrupts affect execution of a job

Interrupt Operation

- Event i triggers interrupt
- Interrupt hardware calls $IH[i]$
- $IH[i]$ gets its own stack frame -- on stack of currently running process
- $IH[i]$ executes, with lower-priority interrupts disabled
- $IH[i]$ returns, restoring control to the interrupted process

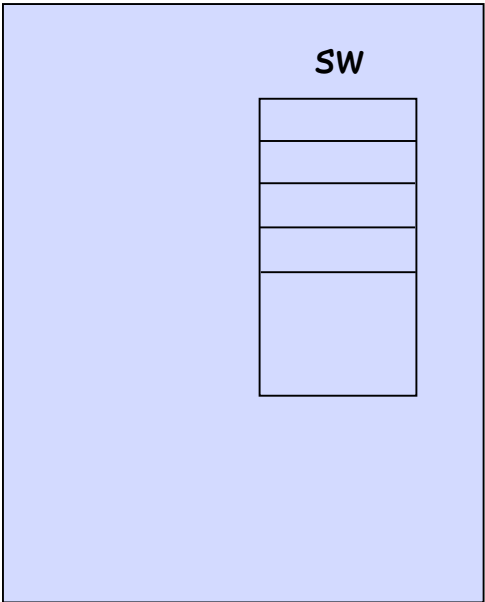
IMPORTANT

- Interrupt handler “borrows” stack of running process
- Looks like “unexpected procedure call”
- Much faster response than a context switch to a system process for dealing with condition
- Effect of handler execution:
 - if invoked by error in the running process: either correct error and let process retry, or abort process
 - if invoked by device signal: no effect on current process

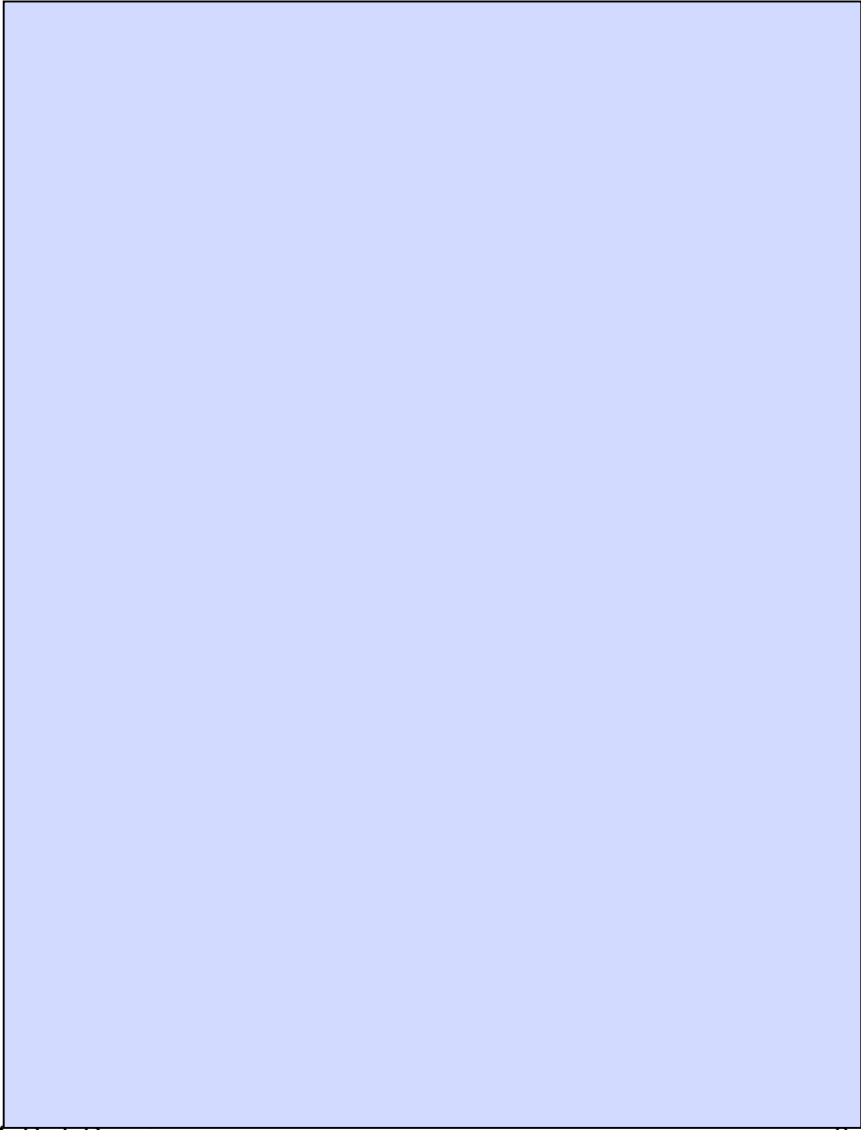
Time Sharing

- Create abstraction process (thread) -- sequence of statewords of a user's program in execution.
- Allow multiple processes with one CPU
- Processes run autonomously, unpredictable speeds
- All processes progress together; average speed less than the CPU speed
- Process synchronization explicit

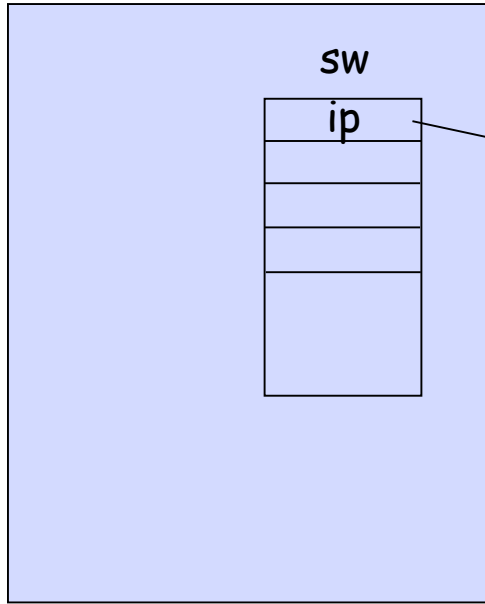
CPU



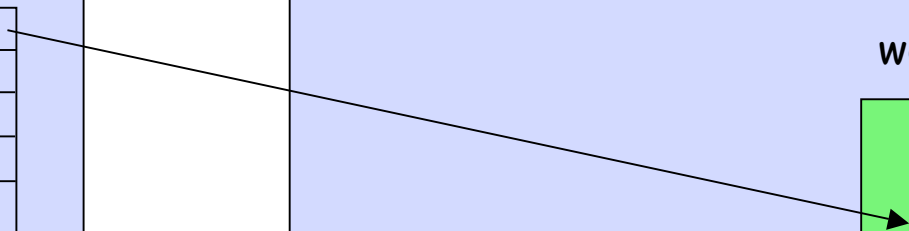
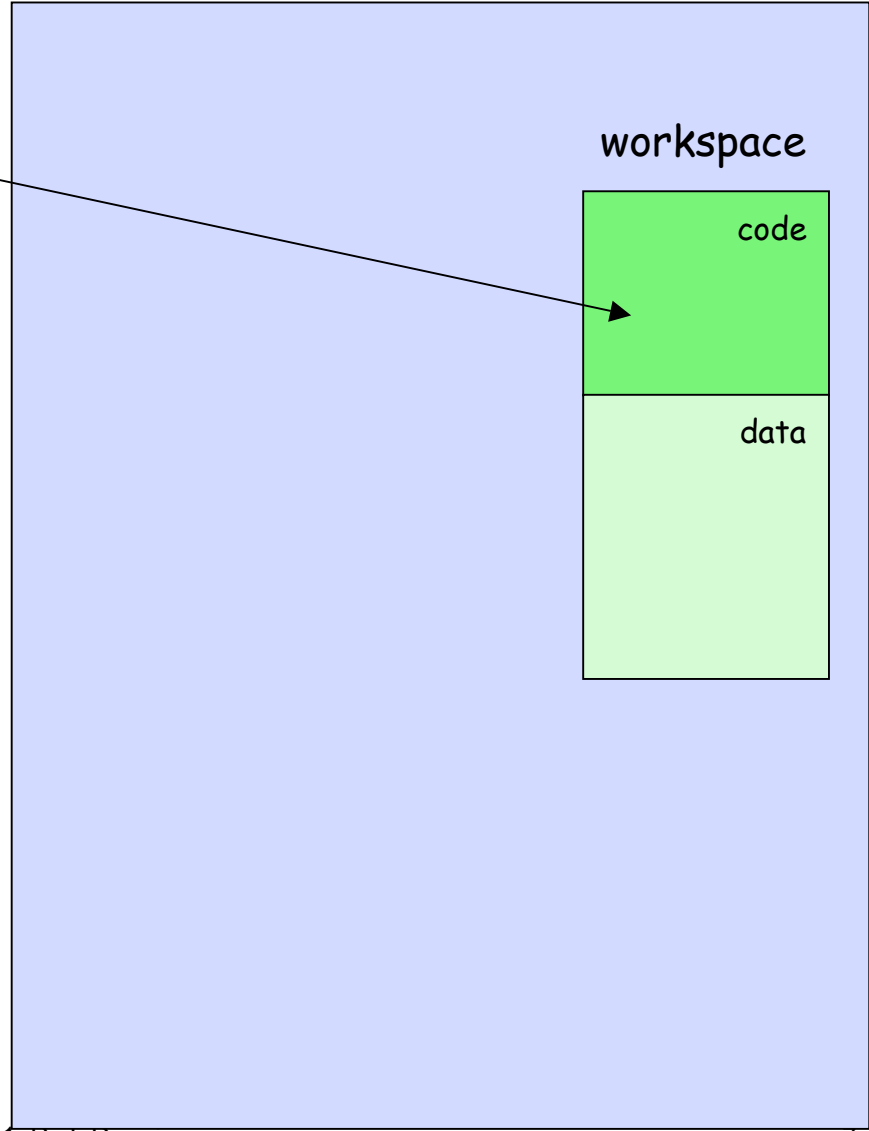
RAM



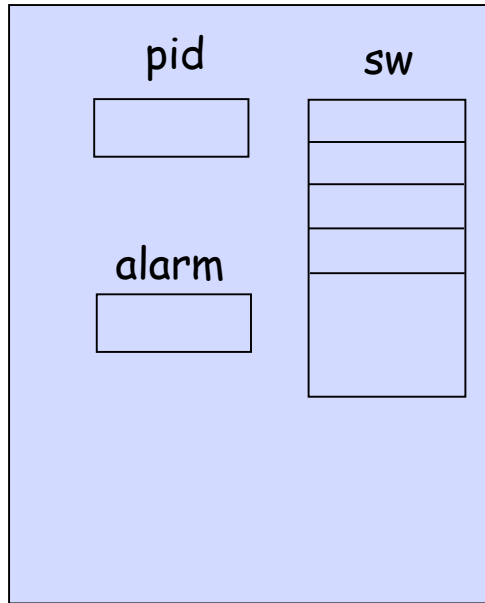
CPU



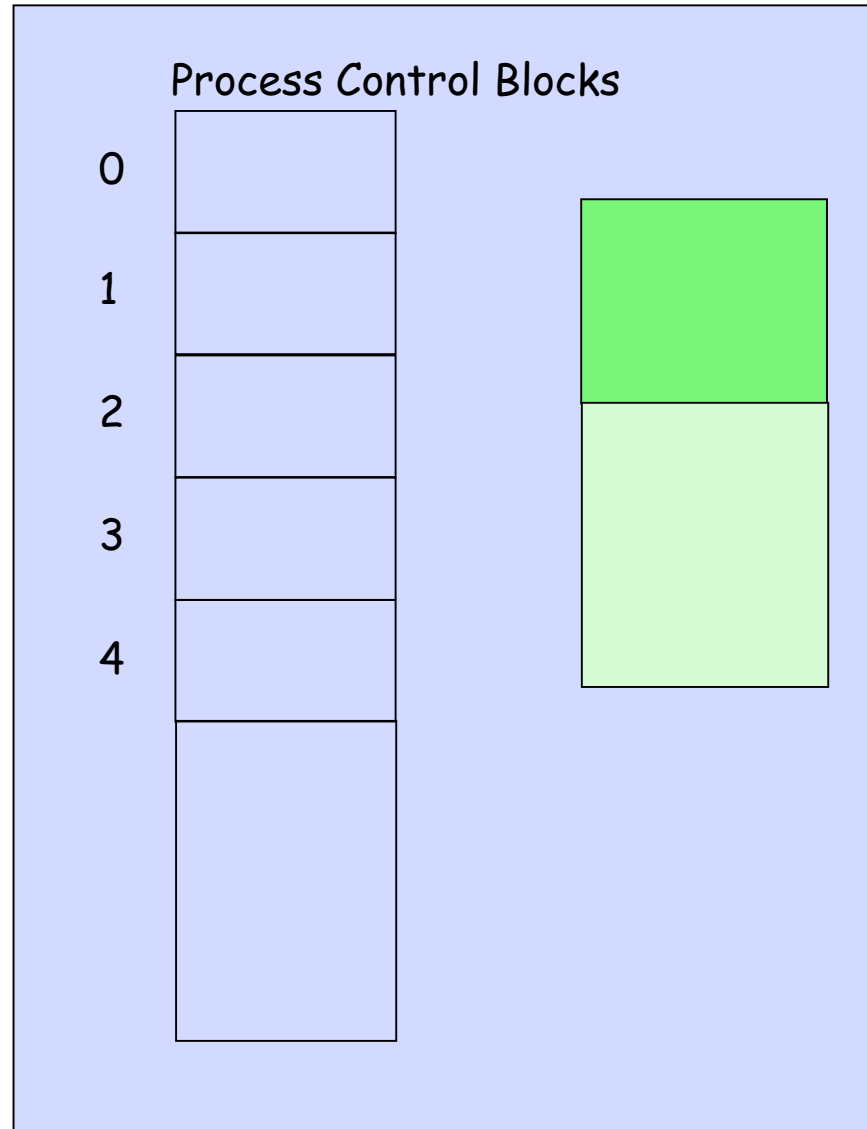
RAM



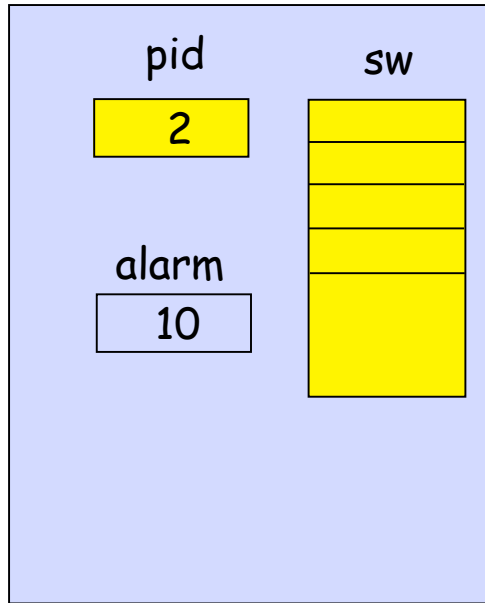
CPU



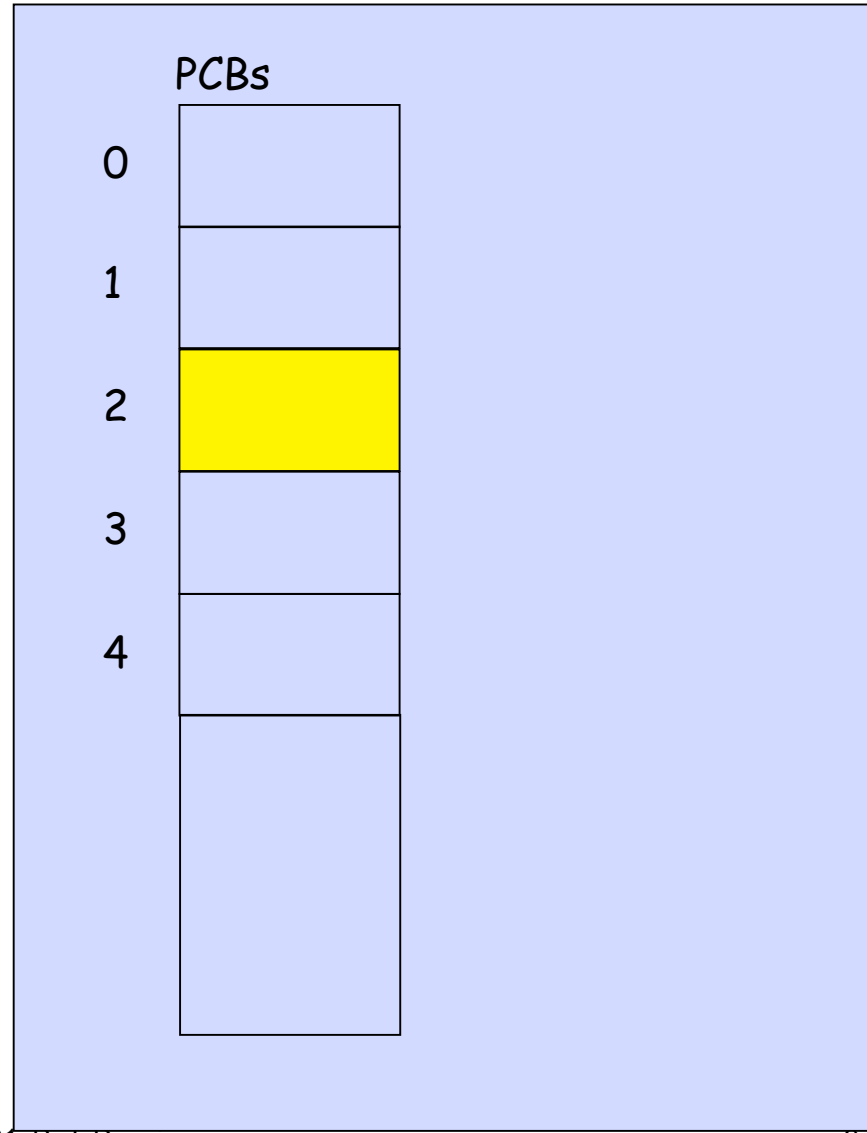
RAM



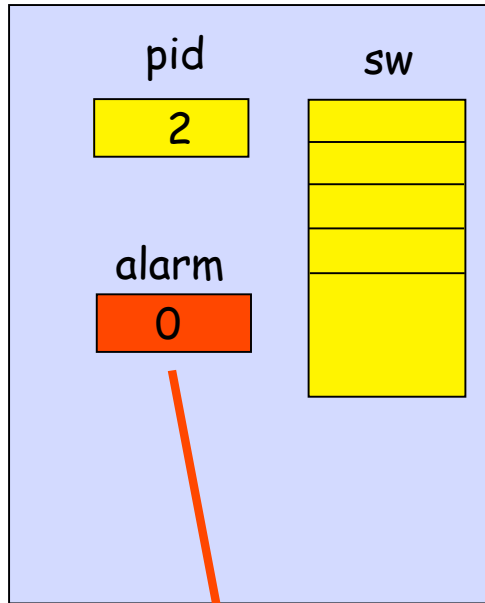
CPU



RAM

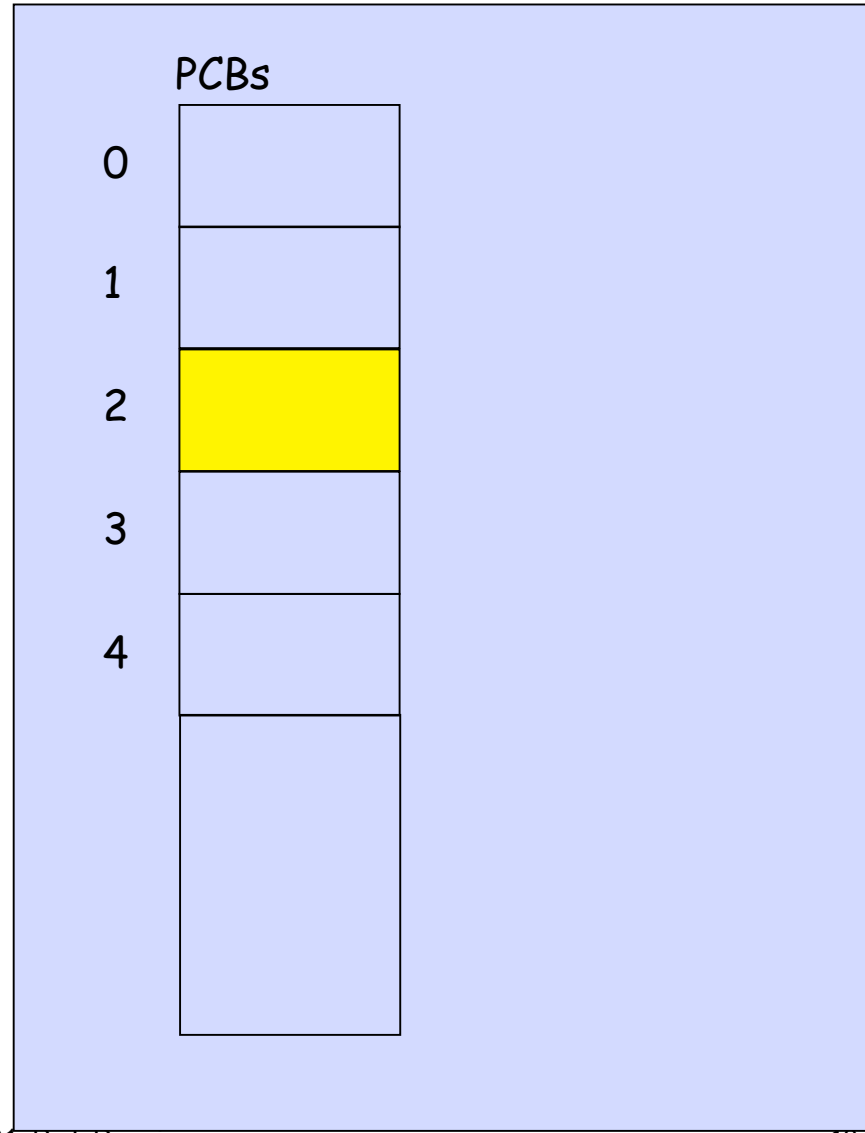


CPU

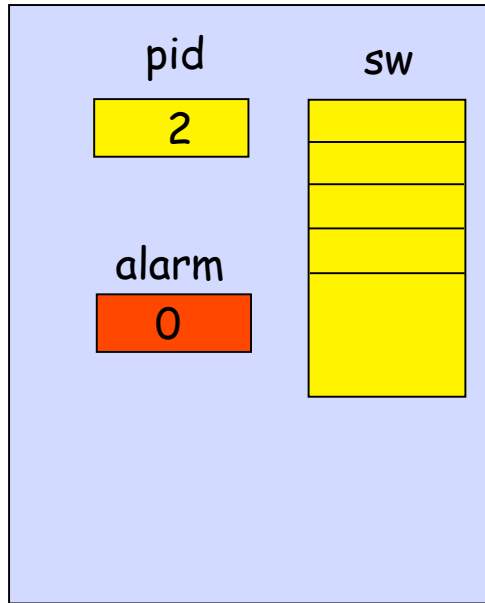


Initiate
Context
Switch

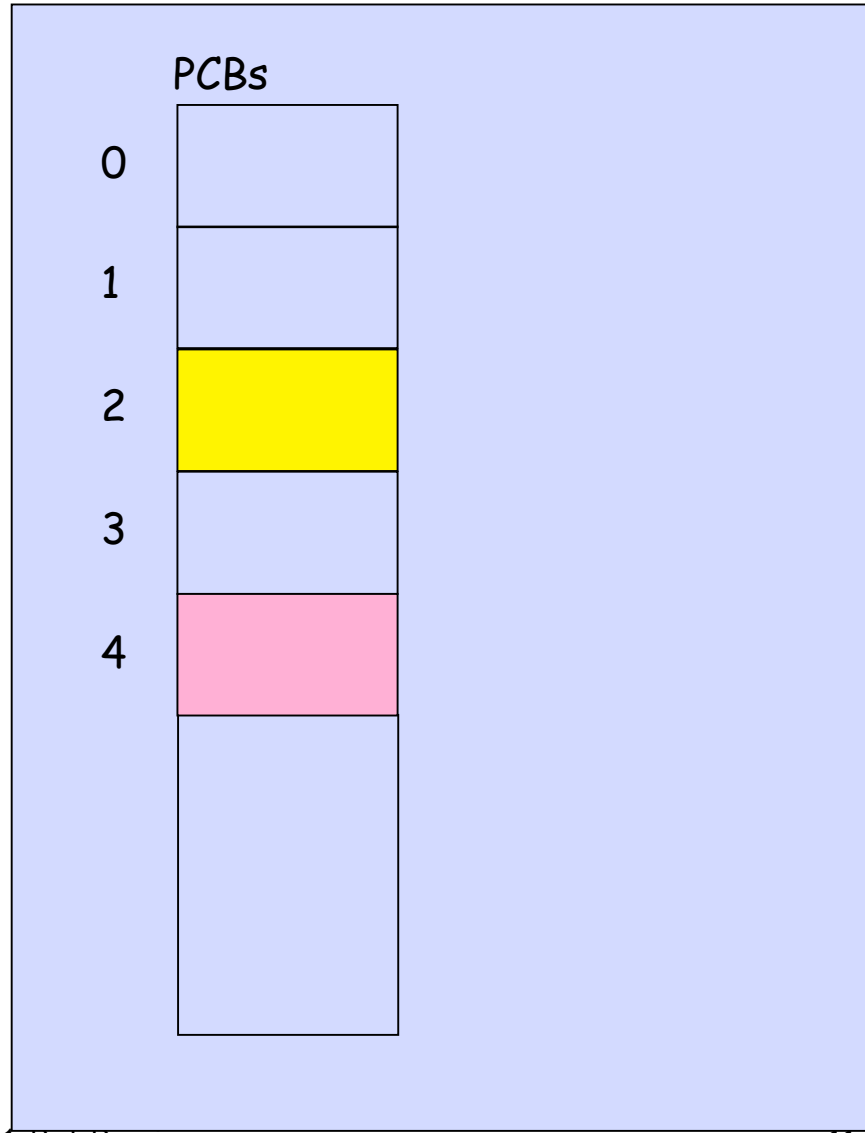
RAM



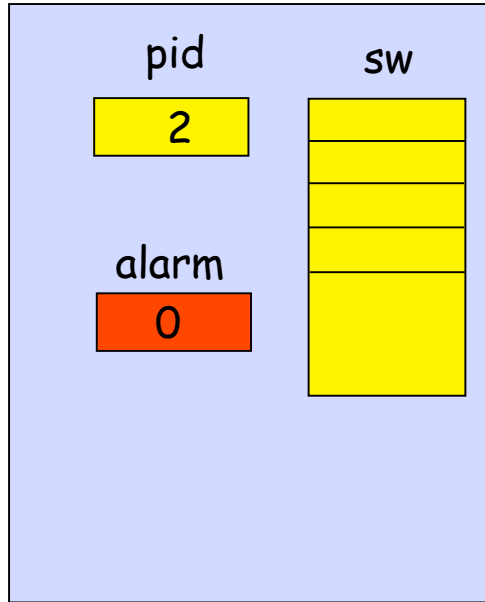
CPU



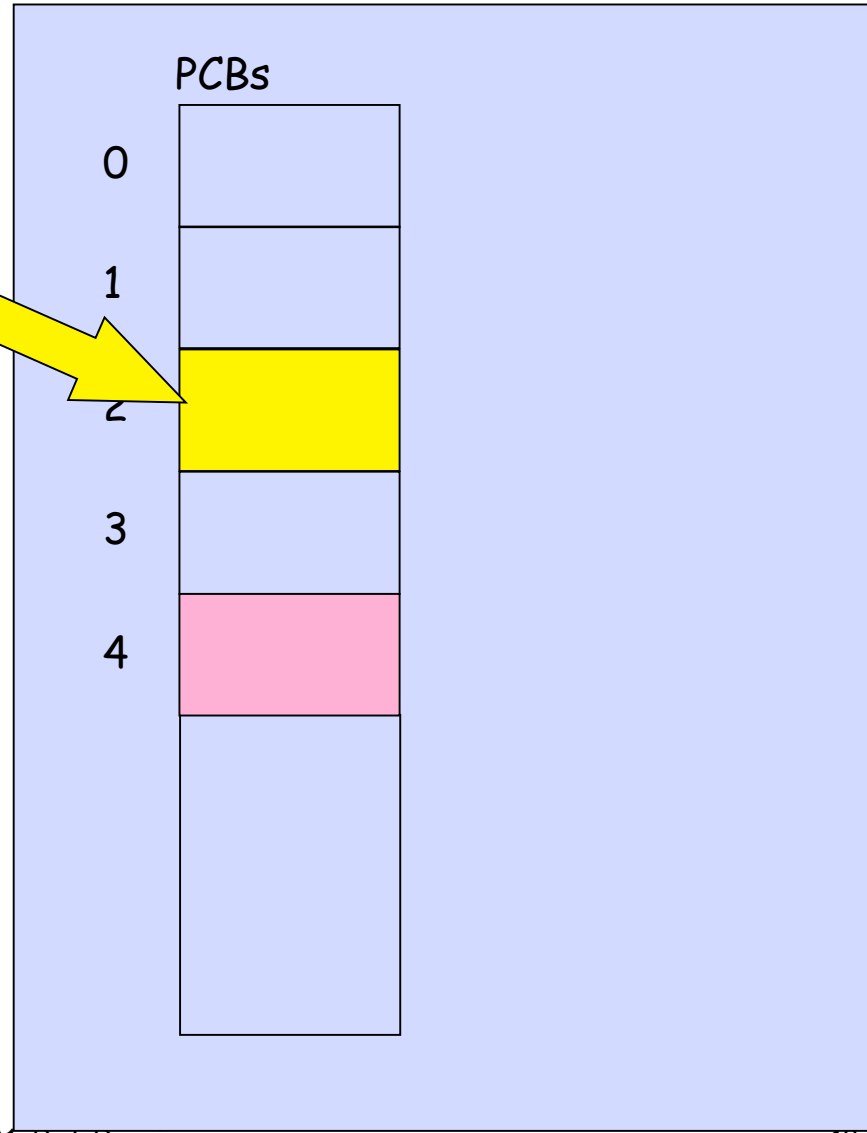
RAM



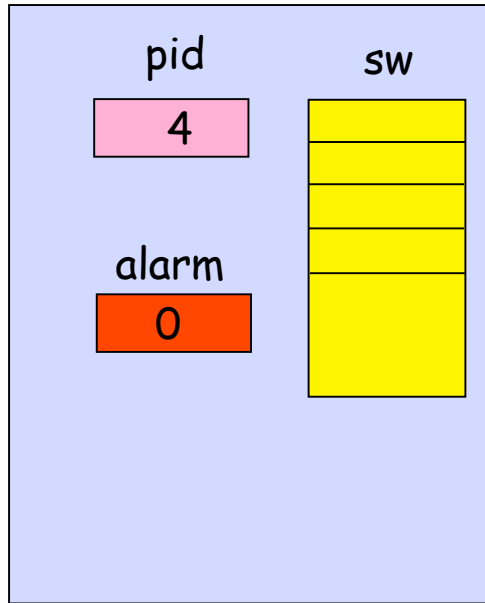
CPU



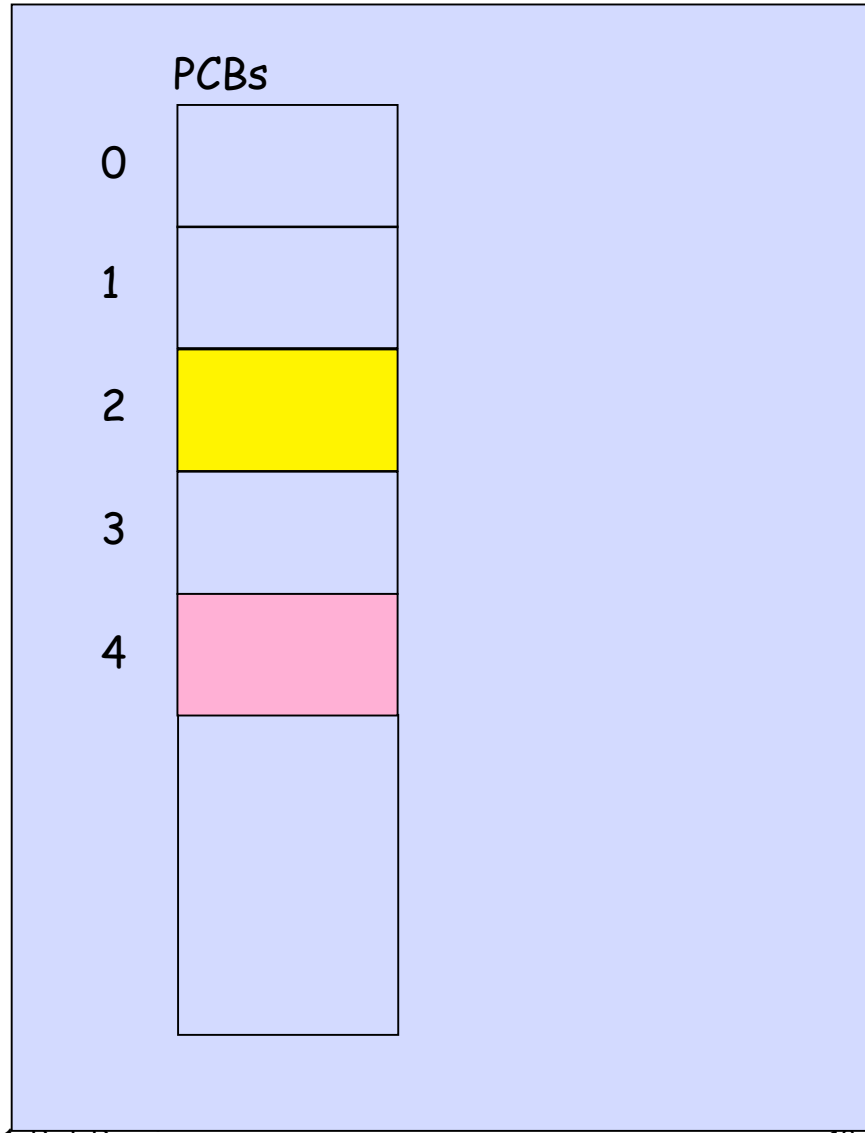
RAM



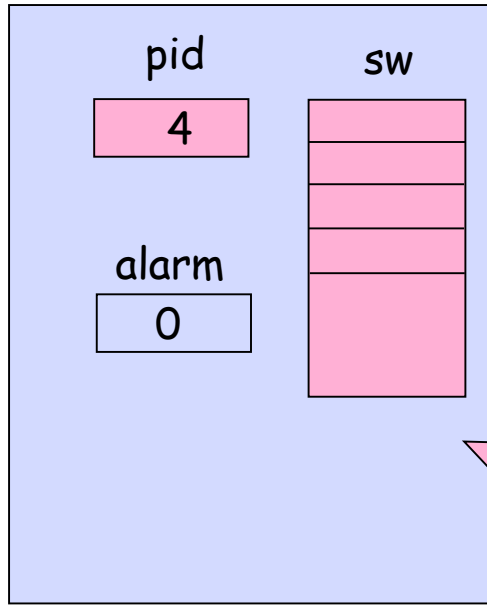
CPU



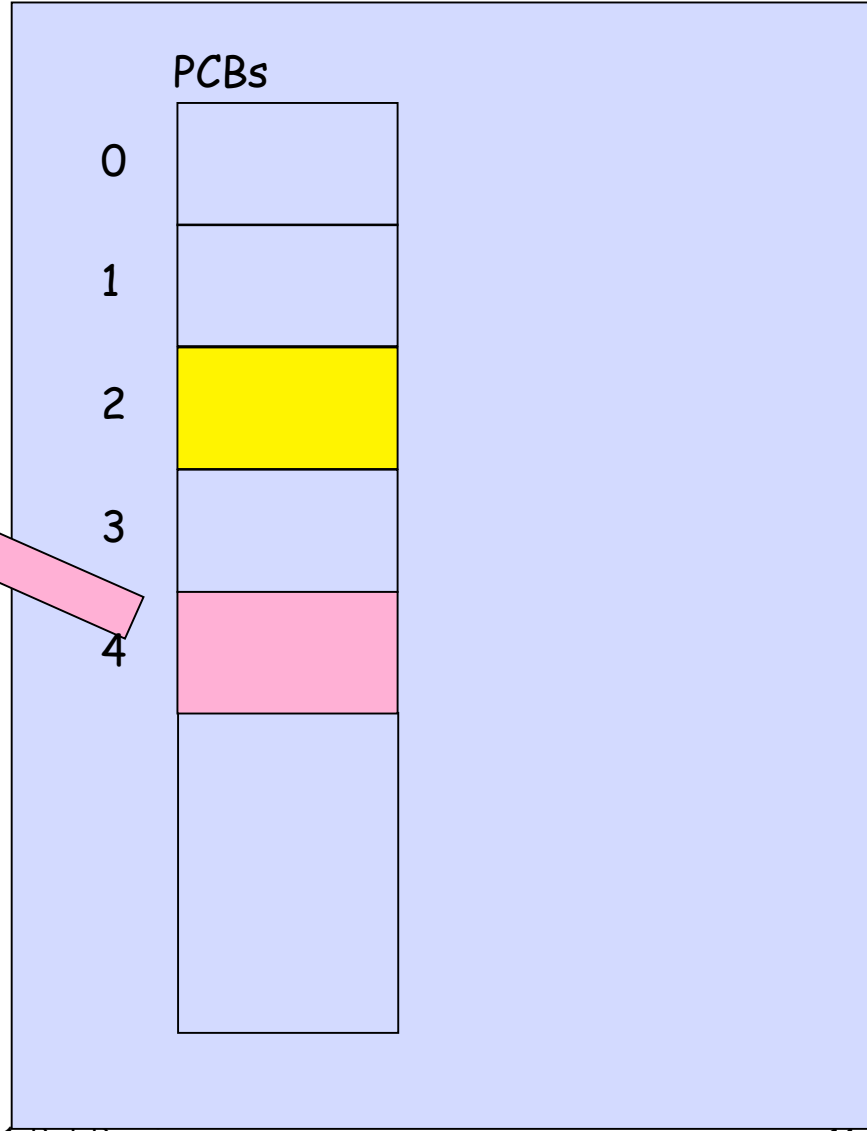
RAM



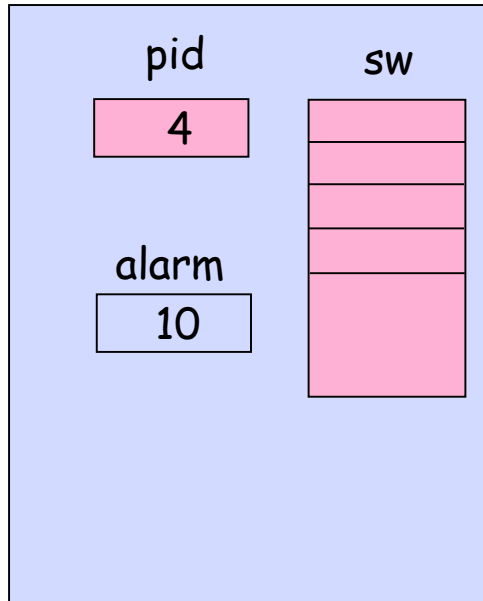
CPU



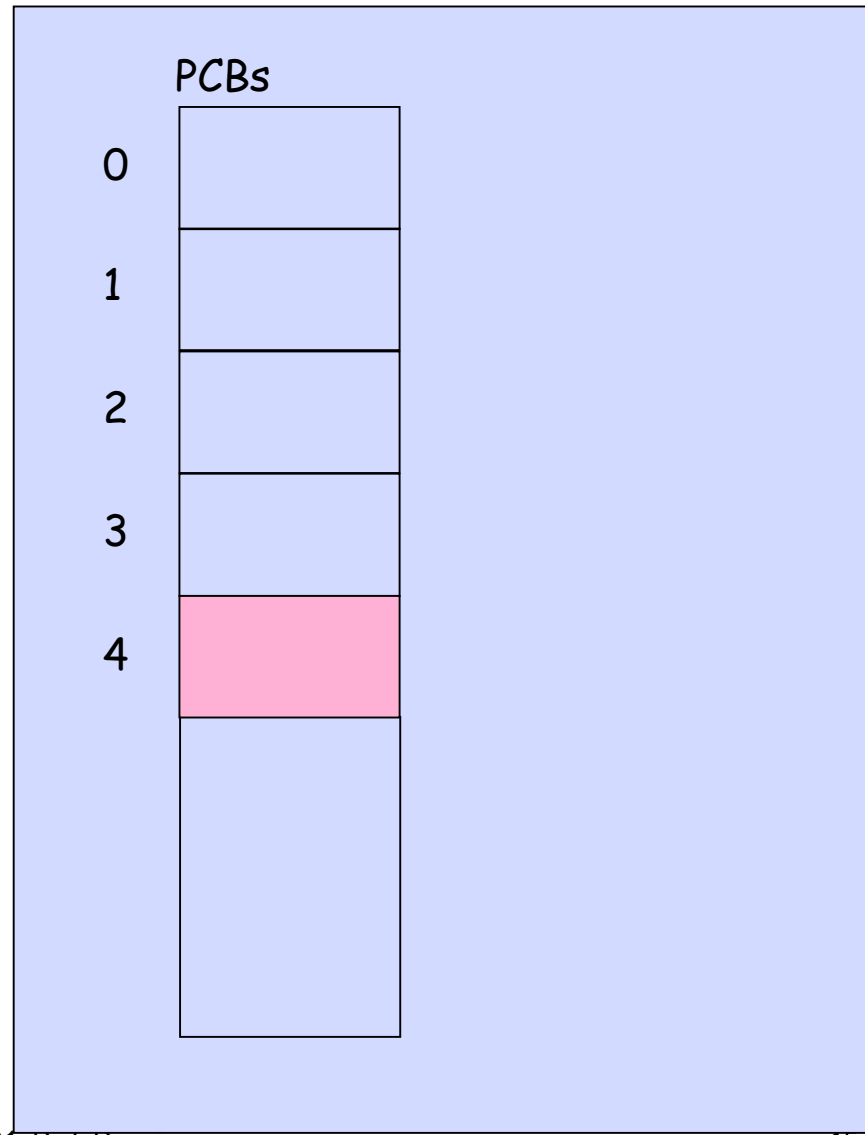
RAM



CPU

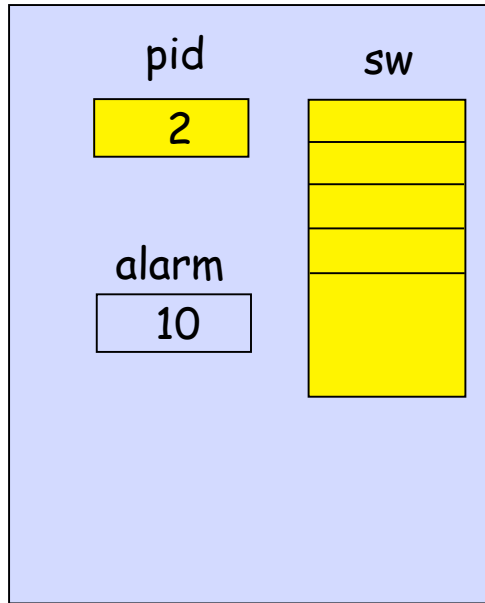


RAM

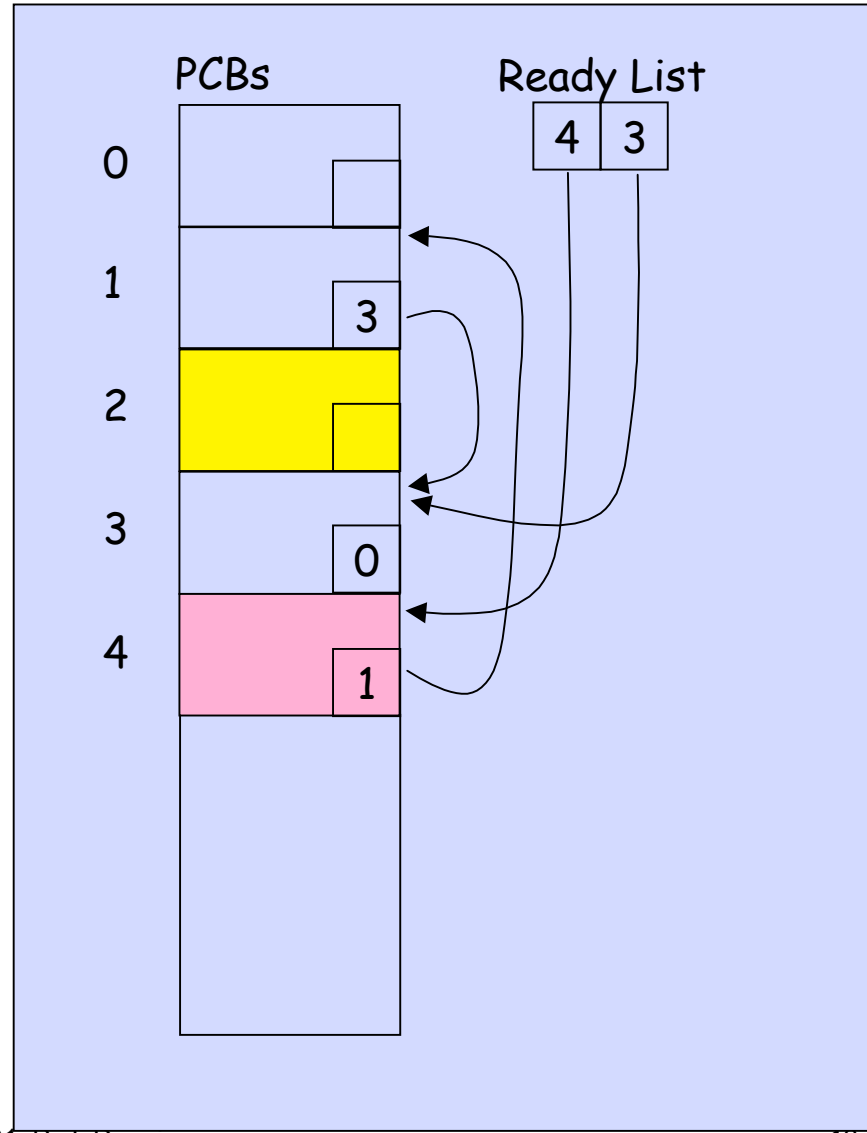


How did 4
come next
after 2?

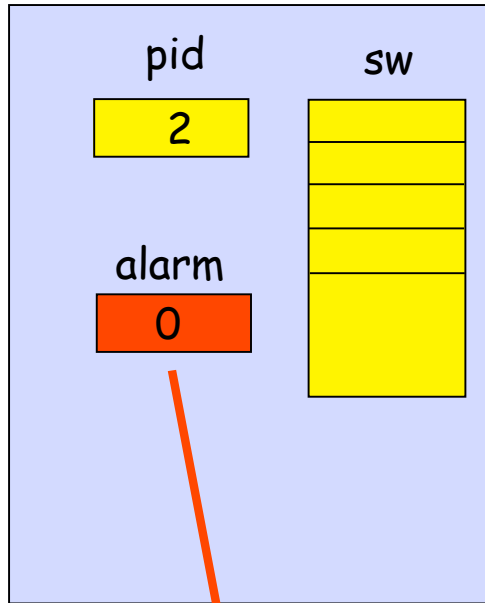
CPU



RAM

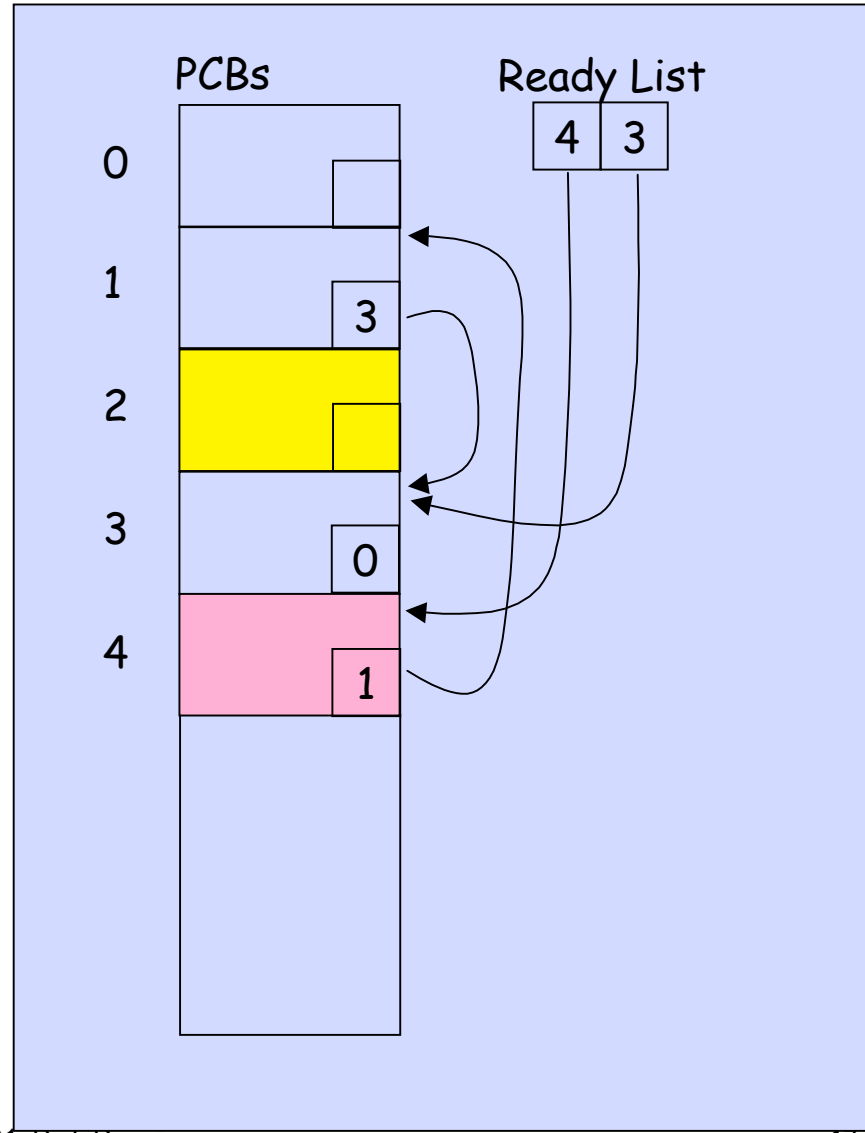


CPU

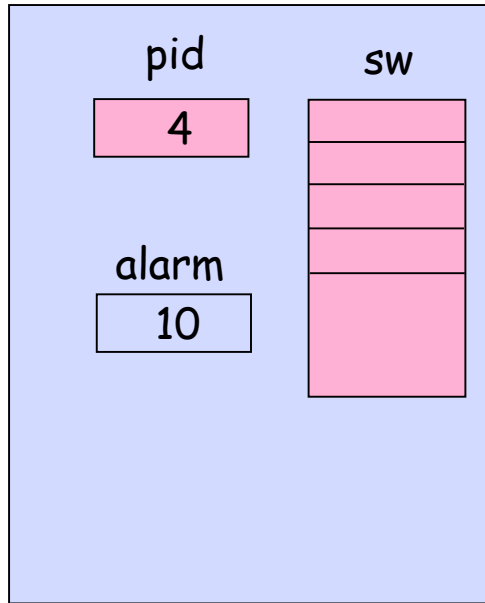


Initiate
Context
Switch

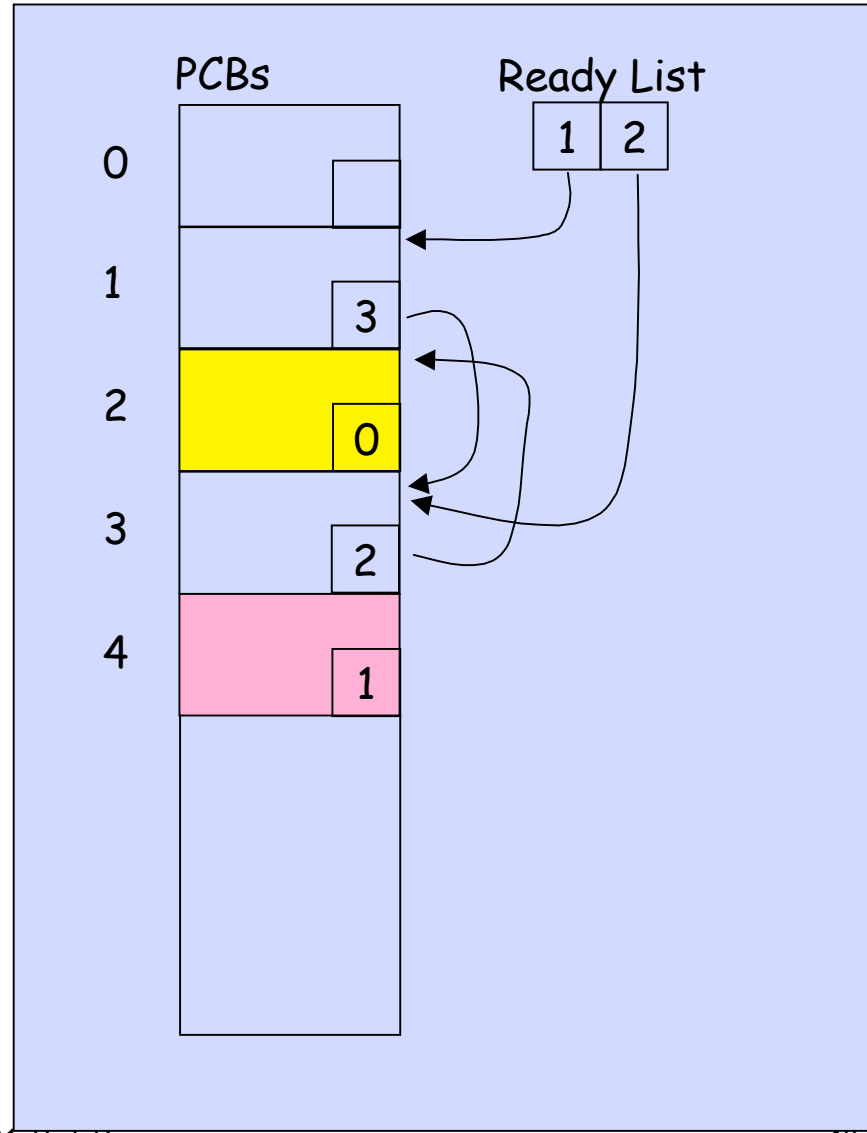
RAM



CPU



RAM



Context Switching

- SAVESW: copy the sw from CPU to PCB[pid].
 - Actually use SAVESW(L), meaning copy the sw to PCB[pid] with IP=L.
- LOADSW: transfer RL.h to pid register in CPU (and make successor of pid the new RL.h); then copy PCB[pid].sw to CPU.

Simple Scheduling (Round Robin)

- Clock interrupt switches CPU to next ready process
- Q = time quantum = max time for CPU to run in one process
- Clock interrupt handler:

```
disable
SAVESW
link pid to RL tail
set alarm clock = Q
LOADSW
enable
return
```

Process 0

- Use the process index 0 to denote the end of the RL -- i.e., $PCB[RL.t].LINK = 0$
- If the RL ever becomes empty, LOADSW will always load process 0 next.
- Process 0 is an IDLE PROCESS that runs a continuous no-op loop, screen saver, etc.
- Operations that make a process stop waiting (discussed later) will displace process 0 from CPU in favor the new ready process.

Creating and Deleting Processes

- Link all unused PCB's together on free list, with descriptor $FL = (h,t)$ just like RL.
- $h = \text{create_process}(\text{init sw})$ -- return first free PCB index and initialize sw in $PCB[h]$ (return 0 means all PCB's taken)
- $\text{delete_process}(h)$ -- put $PCB[h]$ on tail of FL, clear sw in $PCB[h]$

Process Parentage

- Process A creates B -- A is “parent” of B.
- Keep track of parents and children by additional links in the PCB.
- Certain operations, such as `delete_process`, `make_ready`, and `make_waiting`, can only be performed by a parent on its children.

Making Processes Wait 1.0

- Define a third process state, WAITING (along with READY and RUNNING), and a descriptor WL = (h,t) linking all the waiting processes together.
- Make_wait(h) -- unlinks PCB[h] from RL or CPU and adds to WL
- Make_ready(h) -- unlinks PCB[h] from WL and adds to tail of RL

- This waiting mechanism is clumsy -- it does not keep track of the reason that a process is waiting.
- Can define all wait's to be relative to conditions---e.g.,
 - Wait for page of memory
 - Wait for buffer
 - Wait for signal from process 17
- Define SEMAPHORE = object denoting waiting for a specific condition

- Semaphore = (c, h, t) -- a count, and a queue represented by (h,t) descriptor
- Count can be positive or negative
 - $c > 0$: no one is waiting, and the next c processes that ask for condition can proceed without waiting
 - $c \leq 0$: $|c|$ processes are waiting
- Queue is all processes waiting for the condition

- Allow semaphores $1, 2, \dots, M$
- Semaphore list = series of semaphore control blocks, each containing (c, h, t) descriptor of a semaphore and the pid of its creator process.
- The queue of SCB[j] is linked through the PCB link fields as with RL and FL lists.
- $j = \text{create_semaphore}(\text{init } c \geq 0)$ -- get free SCB from semaphore free list, set its count to c , return index j
- $\text{delete_semaphore}(j)$ -- return semaphore block SCB[j] to semaphore free list (allowed only of process that created the semaphore)

- Wait(j) -- subtract 1 from c (of semaphore j).
 - If result is less than 0, add pid to tail of SCB[j] queue and switch to next ready process.
 - If result is 0 or larger, return immediately to caller
- Signal(j) -- add 1 to c (of semaphore j).
 - If result is 0 or less, transfer head process of queue to tail of ready list
 - If results is larger than 0, no action.
 - Always return immediately to caller.


```
wait(j):
  disable
  with SL[j] do {
    c--
    if c < 0 then {
      SAVESW(L)
      "link pid to tail of SL[j]"
      LOADSW }
    }
  L: enable
  return
```

```
signal(j):
  disable
  with SL[j] do {
    c++
    if c ≤ 0 then {
      "move SL[j].h to RL.t" }
    }
  enable
  return
```

Making Processes Wait 2.0

- Use semaphores for waiting.
- Powerful programming aid
 - Process ordering
 - Mutual exclusion
 - Pool control
 - Producer-consumer
 - etc. (see book and Synchronization Slides)

```
P2sem: init c 0
P1: actions
    signal(p2sem)
P2: wait(p2sem)
    actions
```



```
mutex: init c 1
P1: wait(mutex)
    critical section
    signal(mutex)
P2: wait(mutex)
    critical section
    signal(mutex)
```

