Scheduling

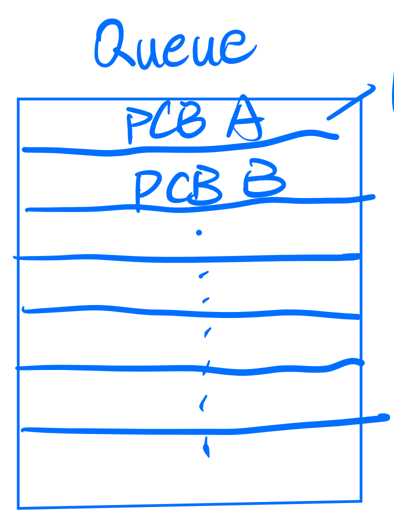
- We now move on to
  next = schedule();

- i.e. we need to find the next process to schedule.

PENNSTATE

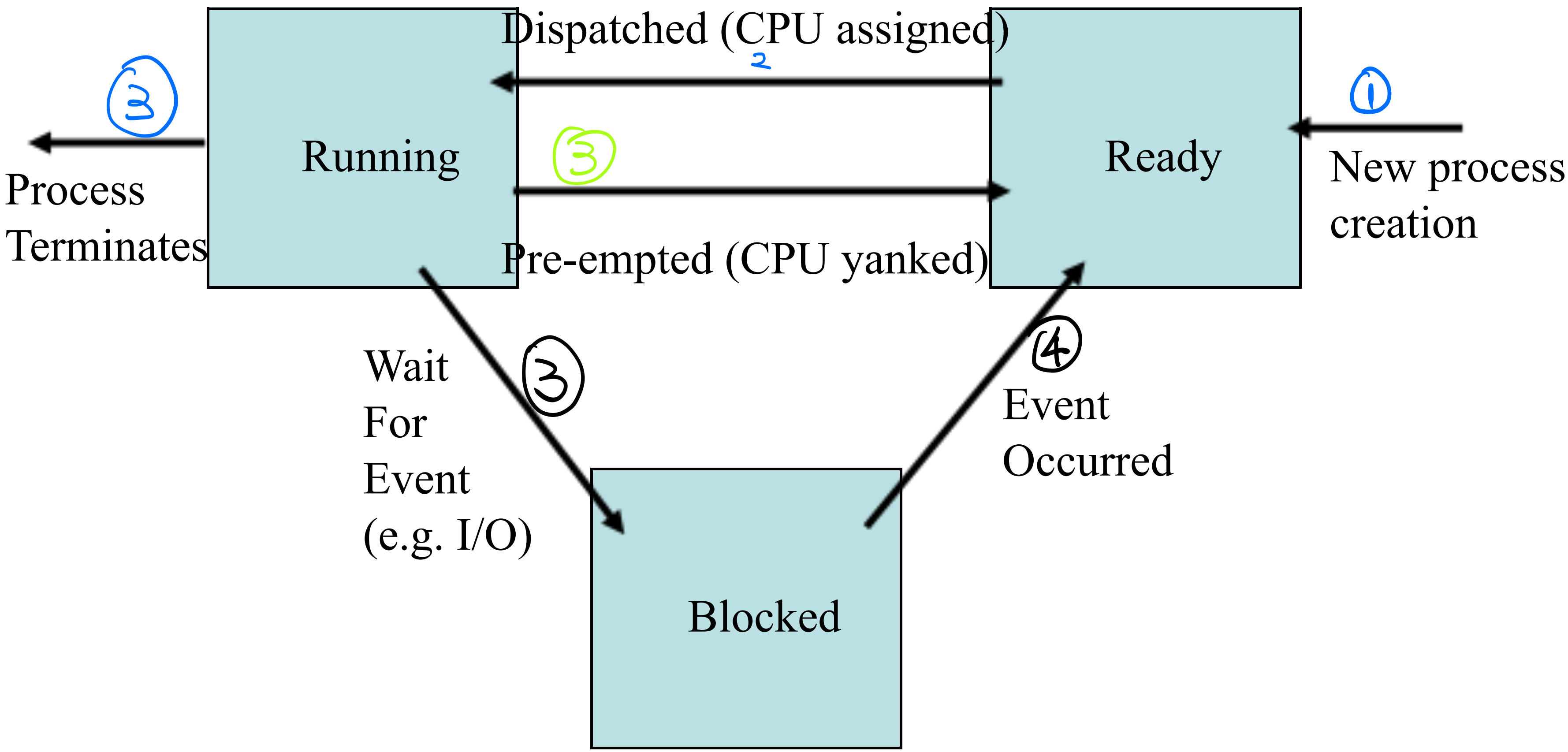
Process States

- A process in the system could be either:
  - Running (on the CPU)
  - Ready (is ready to run if it is given the CPU)
  - Blocked (even if it is given the CPU it cannot proceed since it is waiting for an event to occur, e.g. I/O)
- The OS maintains a queue for each of these states, and the PCB of the process is put in the appropriate queue.



pointers stored

Process State Transition Diagram



NOTE: Each of these states is a queue of PCBs, and the PCB of the concerned process is being moved from one queue to another.

Process Terminate

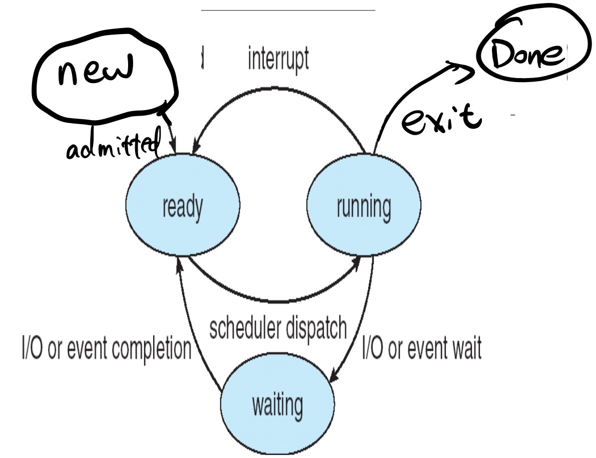
$\leftarrow$ Running $\underset{\longrightarrow}{\xleftarrow{\text{dispatched}}}$ ready $\leftarrow$

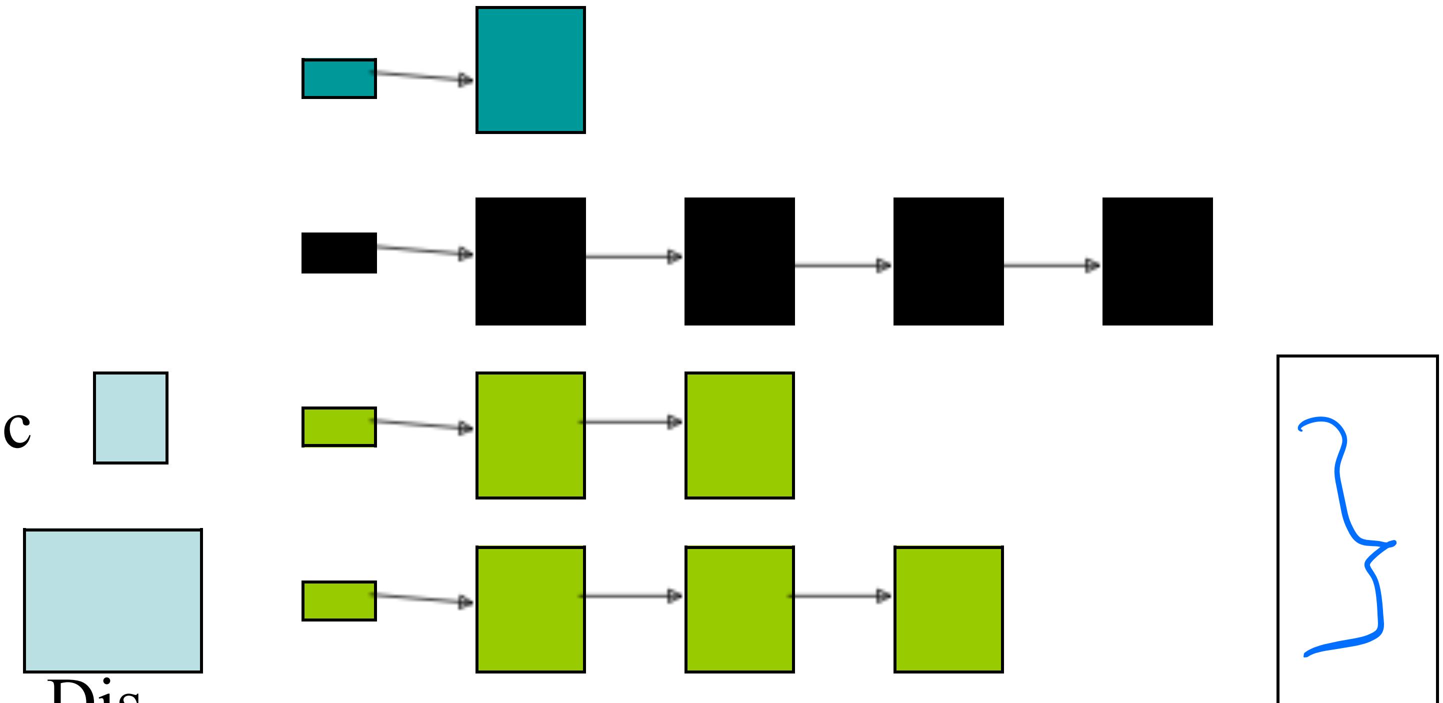
cpu yanks
wait for
event

blocked

Event
occur

All rectangles refer to PCB





Runnin
g

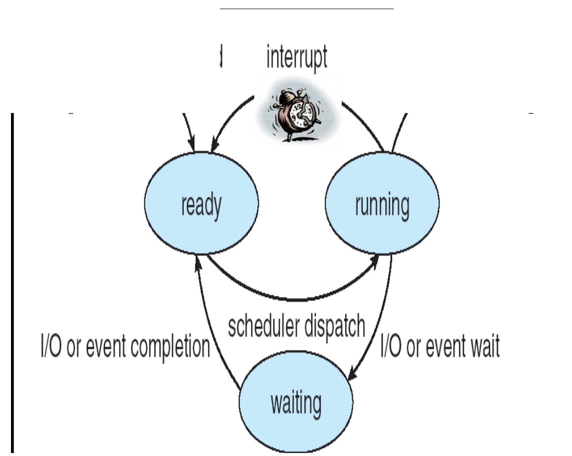
Read
y

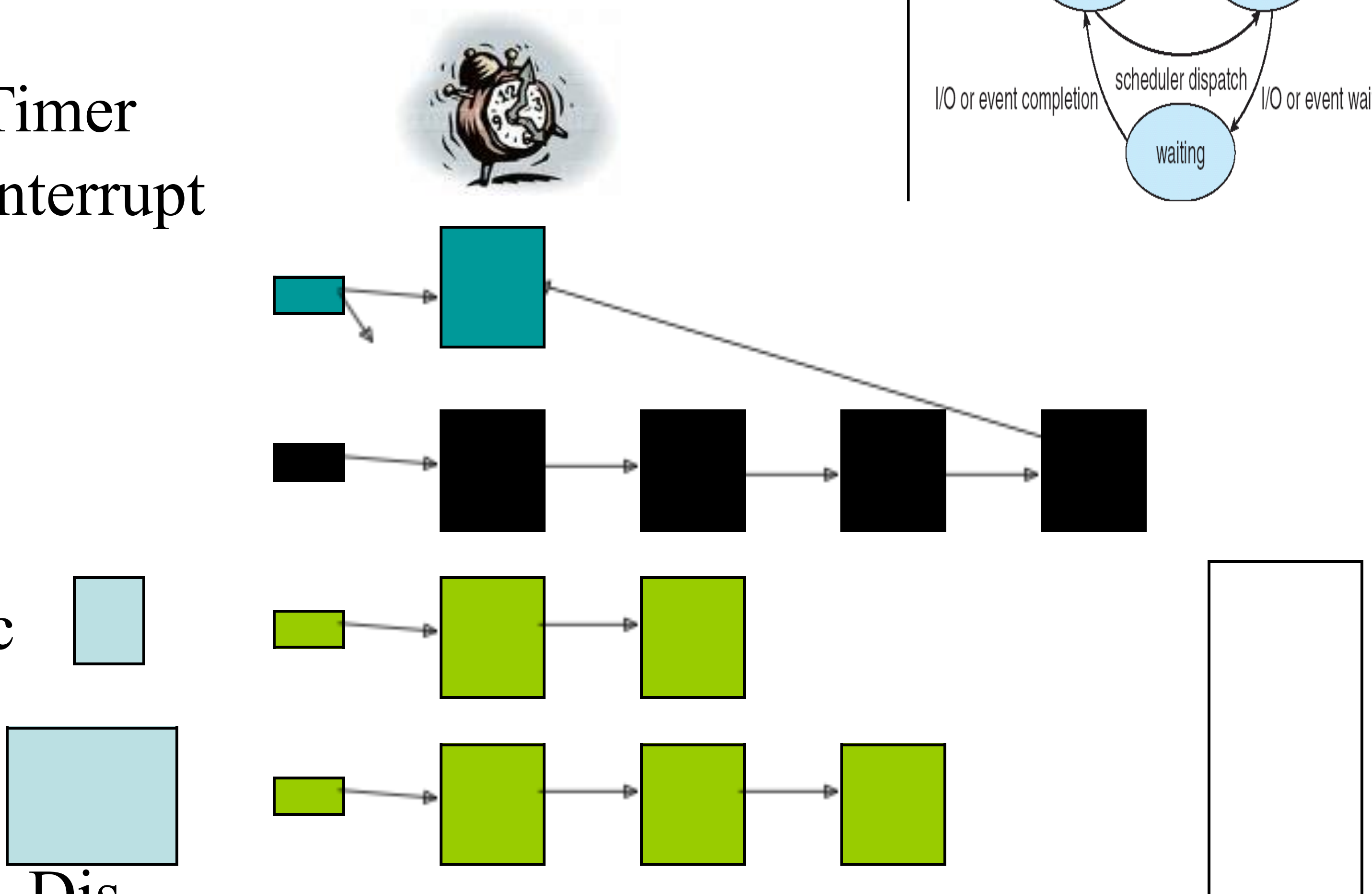
Waiting

Loc
k

Dis
k

Timer interrupt





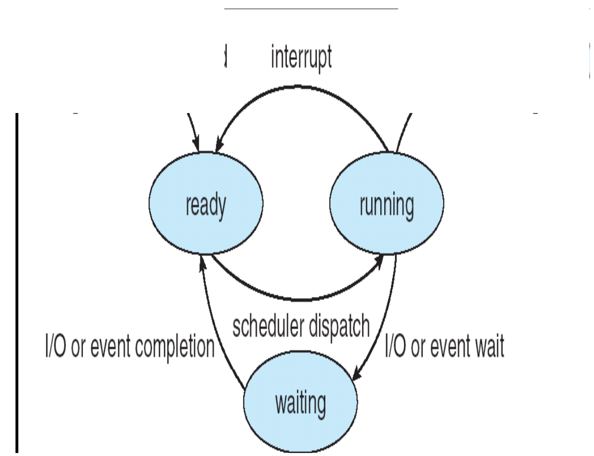
Running

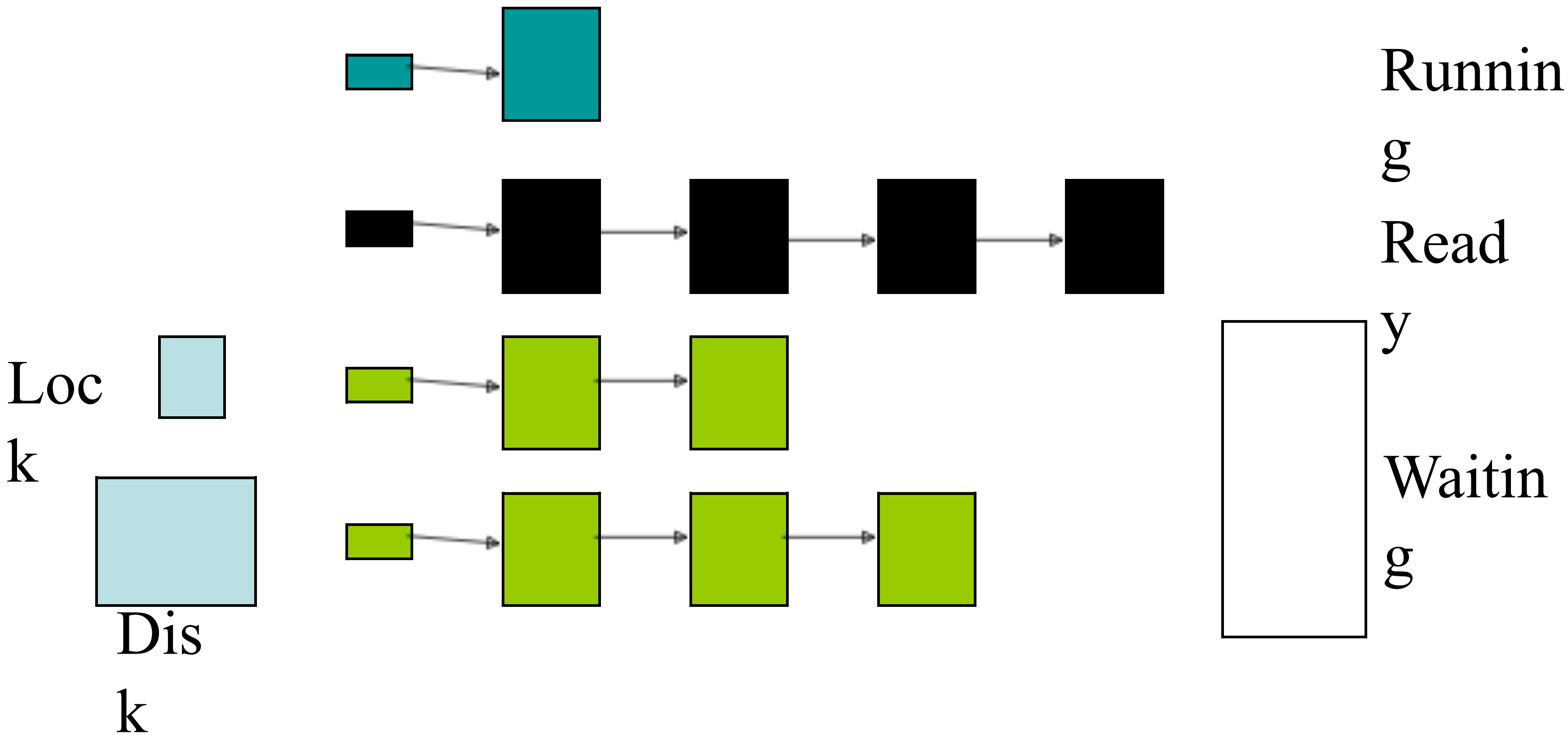
Ready

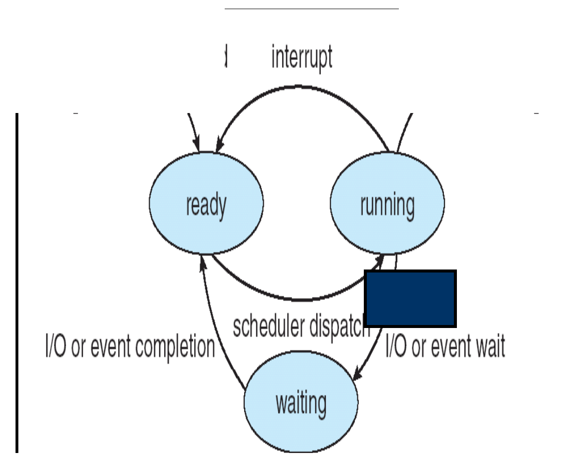
Loc
k

Dis
k

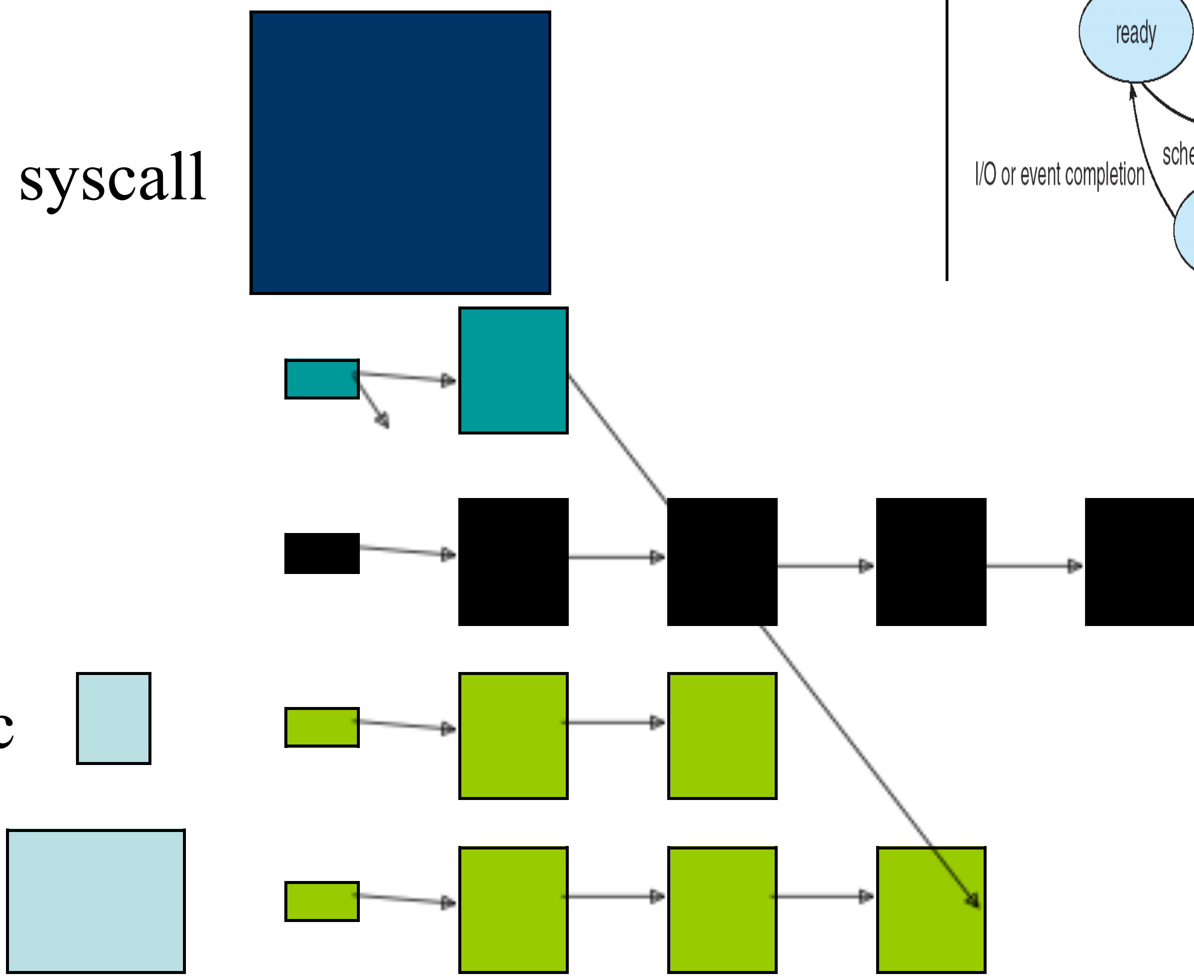
Waitin g


interrupt
ready
running
scheduler dispatch
I/O or event completion
I/O or event wait
waiting


Runnin
g
Read
y
Waitin
g
Loc
k
Dis
k



I/O syscall



Runnin
g

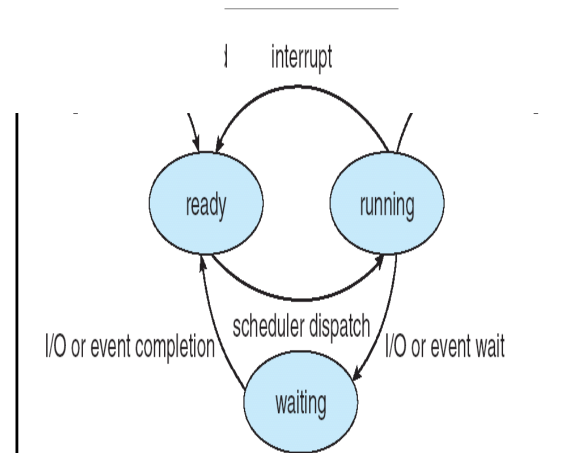
Read
y

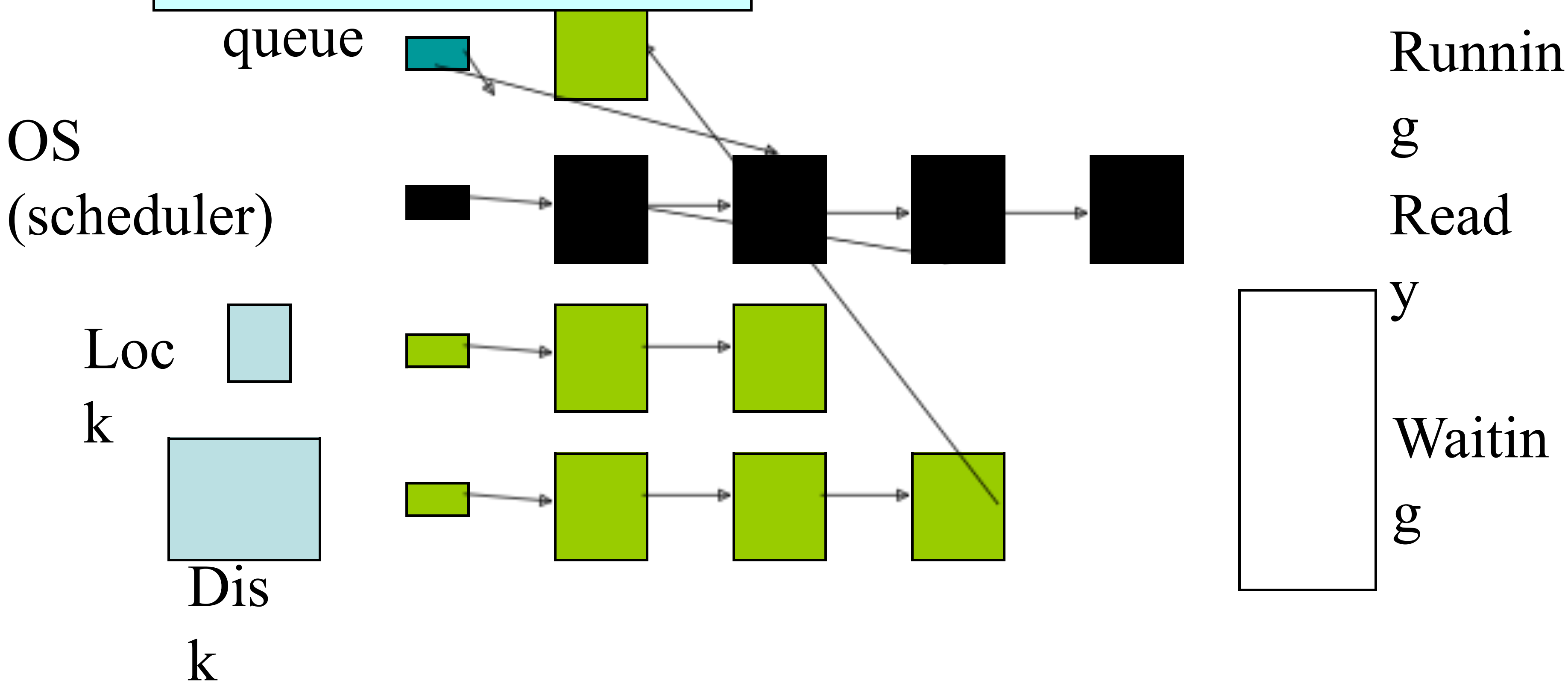
Waitin
g

Loc
k

Dis
k

Lets pick the second process in the ready queue





next = schedule()

- The goal is to pick a process from the Ready queue and give it the CPU.
- Note: there is no point giving the CPU to a process from Blocked queue.

Criteria 符合标准

cpu

- **Utilization/efficiency**: keep the CPU busy 100% of the time with useful work
- **Throughput**: maximize the number of jobs processed per hour.

jobs

- **Turnaround time**: from the time of submission to the time of completion.
- **Waiting time**: Sum of times spent (in Ready qqueue) waiting to be scheduled on the CPU.

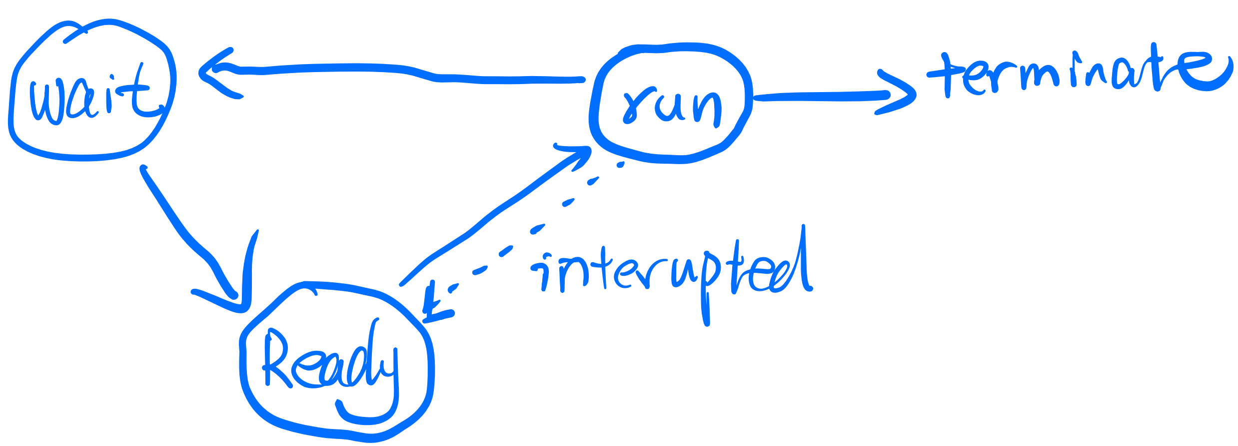
balance cpu and job requirement

- **Response Time**: time from submission till the first response is produced (mainly for interactive jobs)
- **Fairness**: make sure each process gets a fair share of the CPU

Scheduling Concepts

When Can Scheduling Occur?

- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  1. Terminates
- Scheduling for events 1 and 4 do not *preempt* a process
- Process volunteers to give up the CPU



Preemptive vs Non-preemptive

先发制人的

- Can we reschedule a process that is actively running?
  - If so, we have a *preemptive* scheduler
  - If not, we have a *non-preemptive* scheduler
- Suppose a process becomes ready
  - E.g., new process is created or it is no longer waiting
- It may be better to schedule this process
  - So, we preempt the running process
- *In what ways could the new process be better?*

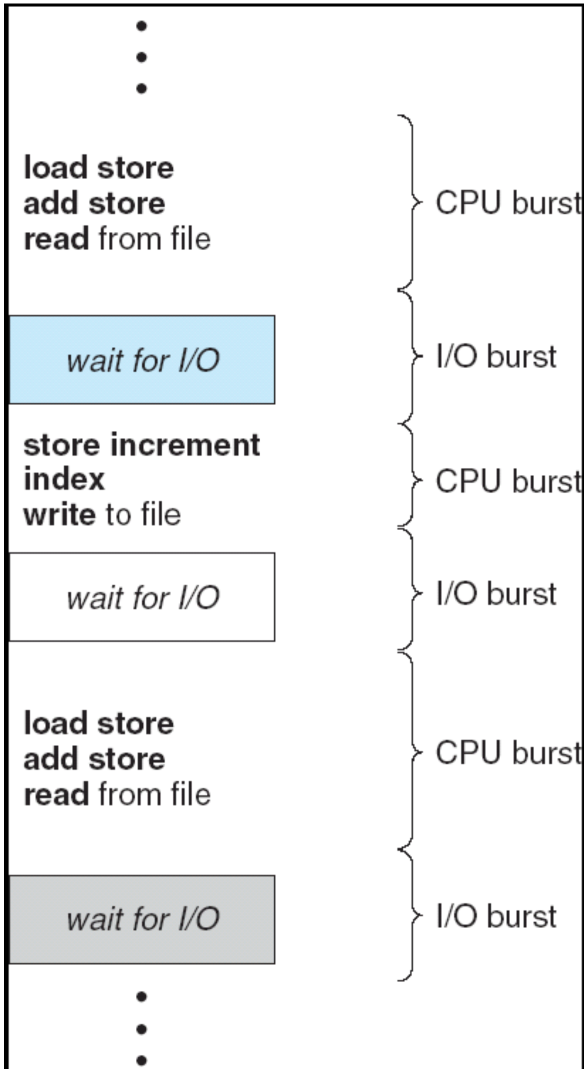
PENNSTATE

Bursts

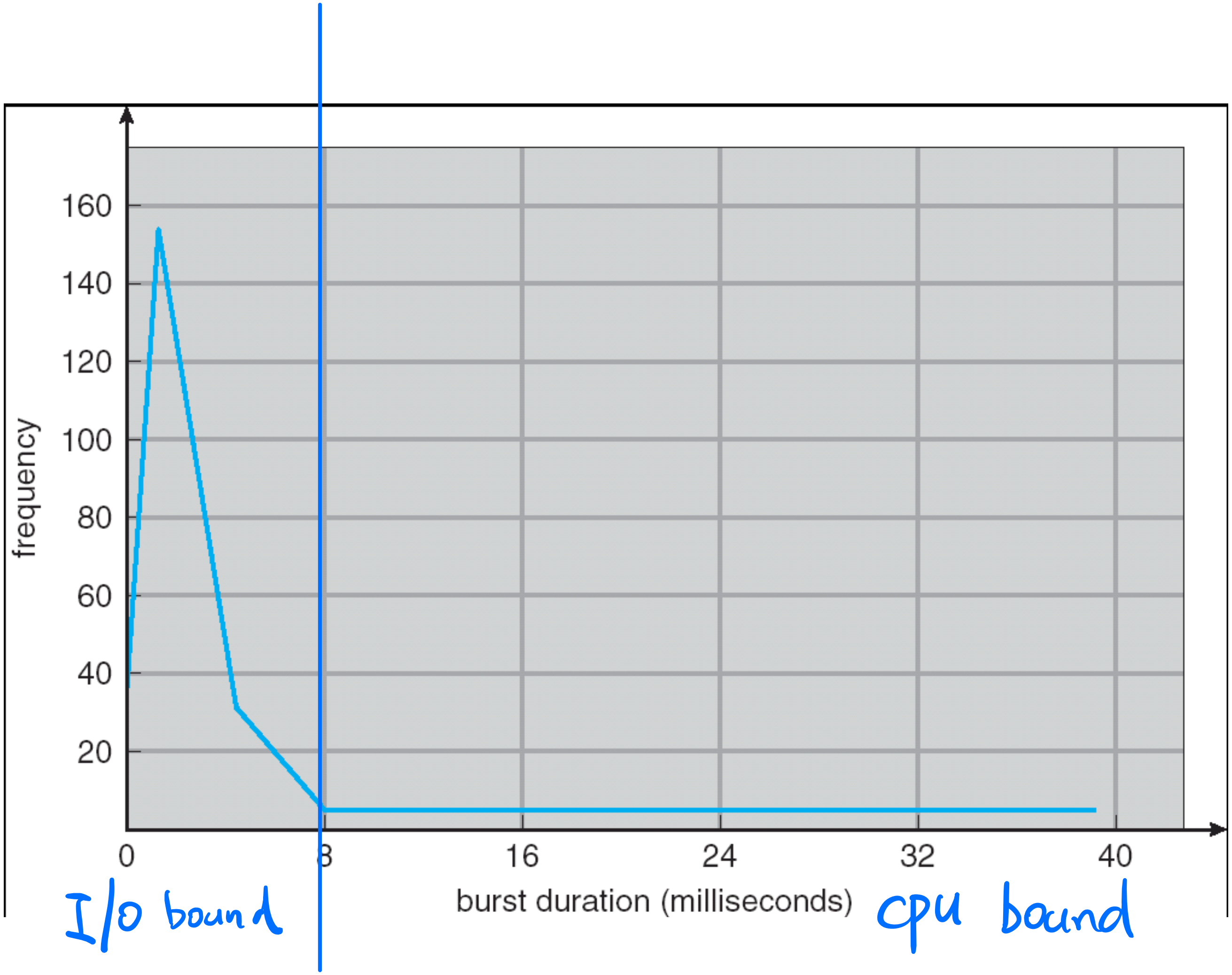
work

- A process runs in CPU and I/O Bursts
  - Run instructions (CPU Burst) → cpu takes long time for
  - Wait for I/O (I/O Burst) → process waiting for I/O response
- Scheduling is aided by knowing the length of these bursts
  - More later…

Bursts


load store
add store
read from file
CPU burst
wait for I/O
I/O burst
store increment
index
write to file
CPU burst
wait for I/O
I/O burst
load store
add store
read from file
CPU burst
wait for I/O
I/O burst

CPU Burst Duration



PENNSTATE

— 调度员

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - Switching context
  - Switching to user mode
  - Jumping to the proper location in the user program to restart that program
- *Dispatch latency* – time it takes for the dispatcher to stop one process and start another running

Scheduling Loop

- How a system runs
  - From a scheduling perspective
    - Don't care about what the process is actually doing…
- Sequence of:
  - Run
  - Scheduling event
  - Schedule
    - Latency
  - Dispatch (if necessary)
    - Latency
  - Rinse, repeat…

Scheduling Algorithms

Scheduling Algorithms

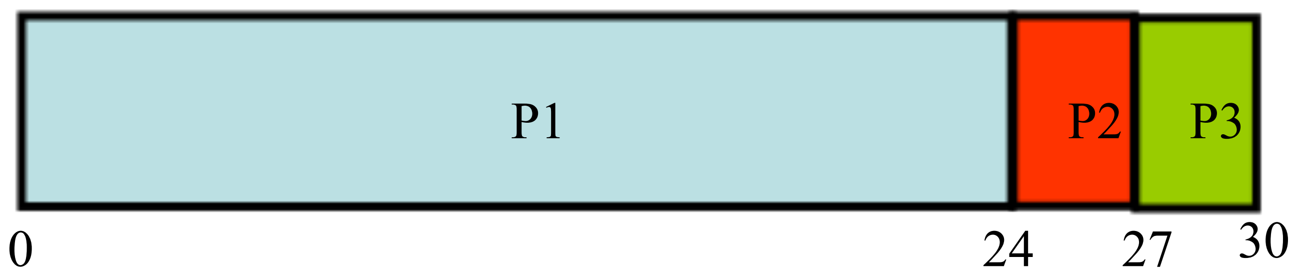
- First Come first Served (FCFS)
  - **Serve the jobs in the order they arrive.**
  - **Non-preemptive**
  - **Simple and easy to implement: When a process is ready, add it to tail of ready queue, and serve the ready queue in FCFS order.**

process does not get a long enough cpu time

  - **Very fair: No process is starved out, and the service order is immune to job size, etc.**

	Arrival Time (s)	Job length (s)
P1	0	24
P2	12	3
P3	17	3

Gantt Chart for FCFS

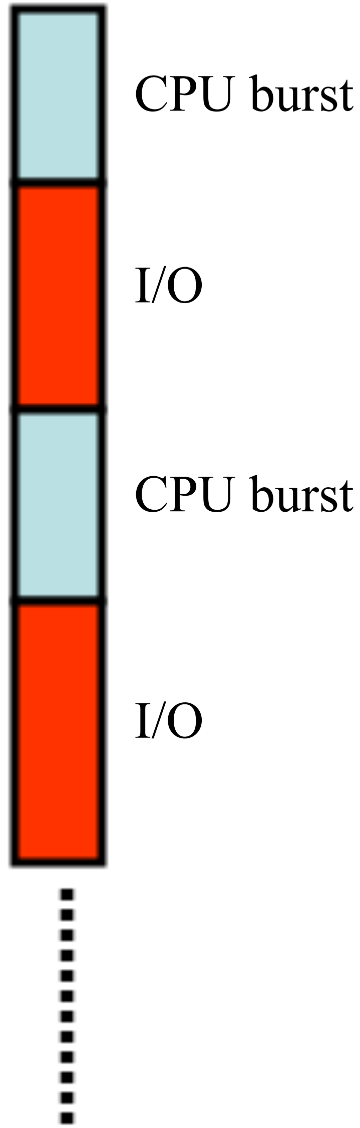


- **Turnaround time for P1 = 24**
- **Turnaround time for P2 = (24+3)-12 =15**
- **Turnaround time for P3 = (24+3+3) – 17=13**
- **Average turnaround time = [(24*3 + 3*2 + 3*1) – 0 – 12 -17]/3**
- **i.e. (n*a + (n-1)*b + …. - …)/n**
- **If you want to minimize this,**
  - **a, b, c, …. should be in increasing sorted order!**

Shortest Job First (SJF)

- Pick the job which is of the shortest duration after the current job is done (let us focus on a non-preemptive version).
- Has low turnaround time.
- Disadvantages:
  - **How do we know job duration?**
  - **Starvation/fairness**

Job Characteristics



For purposes of scheduling, each CPU burst can be viewed as separate job.

How do we estimate duration?

- Typically the same job keeps coming back (either after I/O or newly) requesting for CPU.
- So, we may be able to use prior history.
- Say T(n) is the actual time taken the last time for which we estimated E(n), then we use an exponential averaging technique as follows:

old history

$\alpha T(n) + E(n) \cdot (1-\alpha)$

– $E(n+1) = a \cdot T(n) + (1-a) \cdot E(n)$

recent history

- If a=0, no weightage to recent history
- If a=1, no weightage to old history
- Typically, choose a=1/2 which gives more weightage to newer information compared to older information.

Priority Scheduling

- Each process is given a certain priority "value".
- Always schedule the process with the highest priority.

	Duration(s)	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

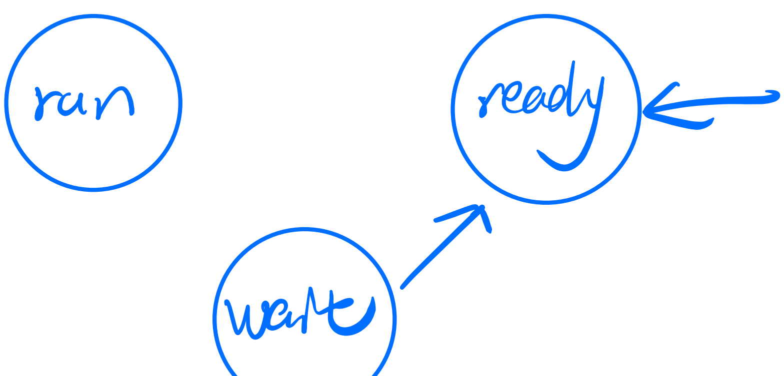
Gantt Chart for Priority Scheduling

- Note that FCFS and SJF are specialized versions of Priority Scheduling, i.e. there is a way of assigning priorities to the processes so that Priority Scheduling would result in FCFS/SJF.

Until now …

- **Non-preemptive**
  - **FCFS**
  - **SJF**
  - **Priority Scheduling**
- **Note that we can have preemptive versions**
  - **i.e. whenever the conditions change during the execution of current job, it can be rescheduled even if it has not finished.**
  - **SJF (Shortest Remaining-time First (SRTF))**
  - **Priority Scheduling**

	Arrial	CPU Burst
P1	0	24
P2	12	3
P3	17	3



PENNSTATE

Gantt chart

P1	P2	P1	P3	P1
0–12	12–15	15–17	17–20	20–30

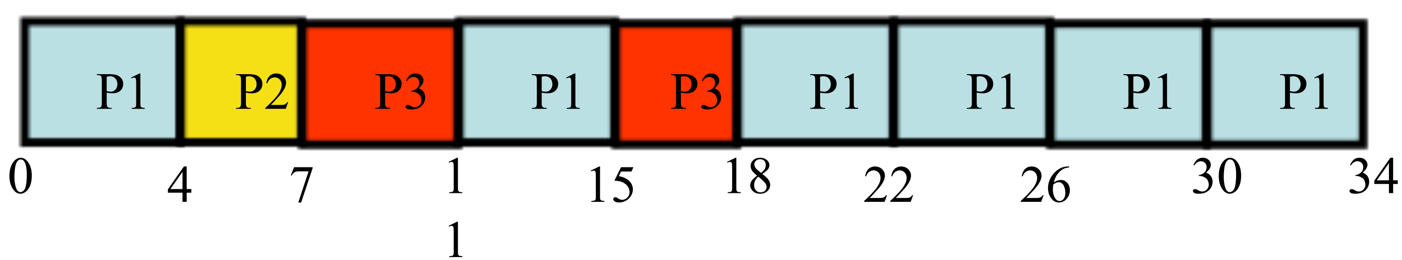
Round Robin (preemptive)

- **Periodically switch from 1 process to another.**
- **You time slice the CPU between processes in units called "time quanta".**
- **Implementation:**
  - **When a process arrives, add it to end of ready queue.**
  - **When current time quantum expires, preempt current process and put it at end of ready queue.**
  - **Give the CPU to the process at head of ready queue for the next time quantum.**
  - **If the process blocks (during the middle of its quantum), then put it in blocked queue, and give CPU to head of ready queue for another "time quantum" units.**

An example of Round Robin

	Arrival Time (s)	Job length (s)
P1	0	24
P2	2	3
P3	3	7

Time Quantum = 4 s



- **Round robin is virtually sharing the CPU between the processes giving each process the illusion that it is running in isolation (at 1/n-th the CPU speed).**
- **Smaller the time quantum, the more realistic the illusion (note that when** the order of **time quantum is of the order of job size, it degenerates to FCFS).**
- **But what is the drawback when time quantum gets smaller?**

- For the considered example, if time quantum size drops to 2s from 4s, the number of context switches increases to ???? 2
- But context switches are not free!
  - **Saving/restoring registers**
  - **Switching address spaces**
  - **Indirect costs (cache pollution)**

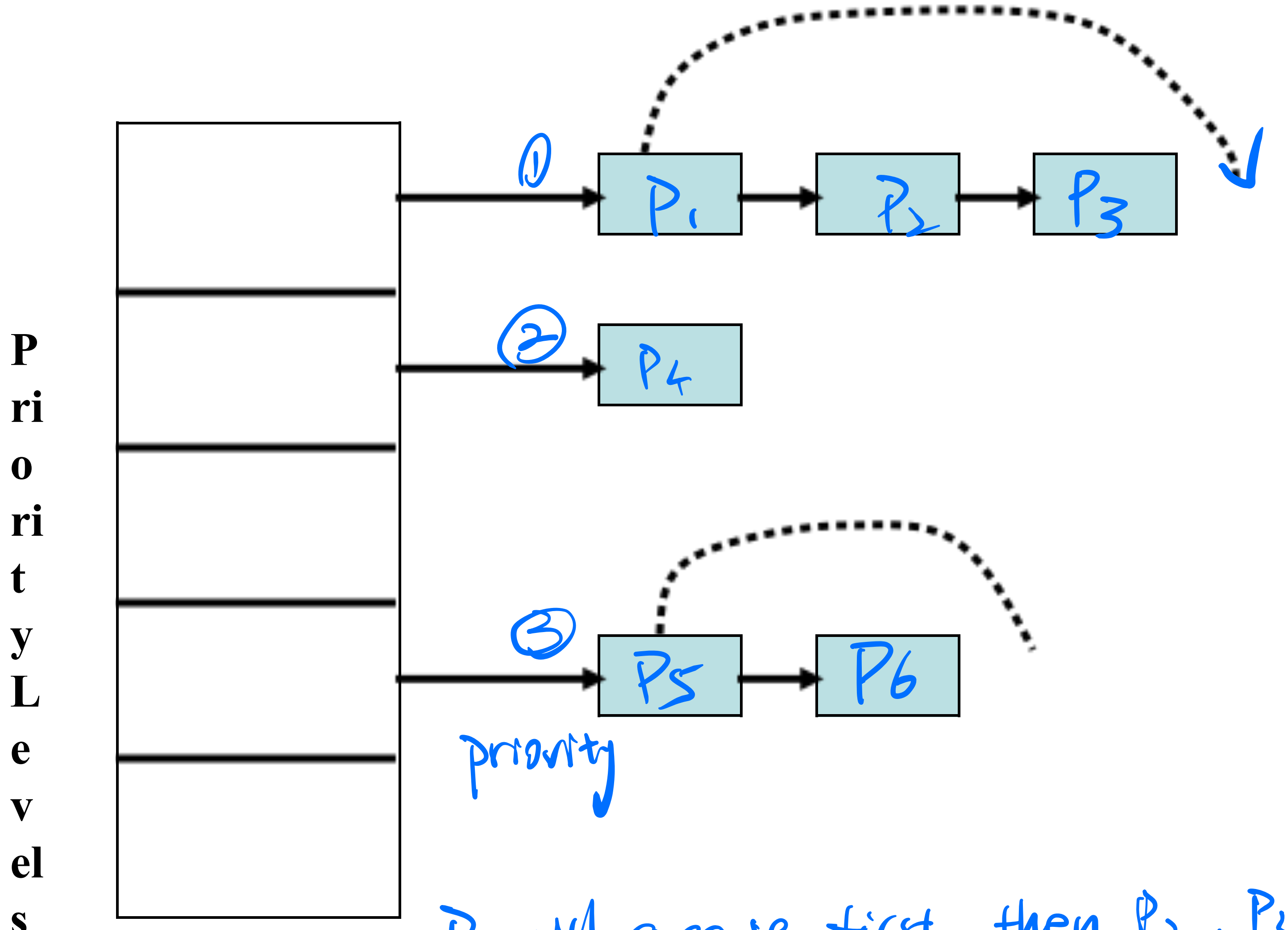
General rules of thumb

- Keep quanta large enough to accommodate most CPU bursts within 1 quantum
- Keep quanta large enough to keep context switch overheads relatively low.
- Typically context switch costs are in 10s of microseconds.
- Time quanta are in 10s/100s of milliseconds.

Round Robin with Priority

- Have a ready queue for each priority level.
- Always service the non-null queue at the highest priority level.
- Within each queue, you perform round-robin scheduling between those processes.

Round-robin with priority



P1 will execute first, then P2, P3

What is the problem?

- With fixed priorities, processes lower in the priority level can get starved out!

- In general, you employ a mechanism to "age" the priority of processes.

Desirables

- Round-robin scheduling is attractive from the point of processor sharing (virtualizing the CPU).
- Round-robin scheduling is attractive for interactive jobs since you may get to start running much earlier than other non-premptive strategies.

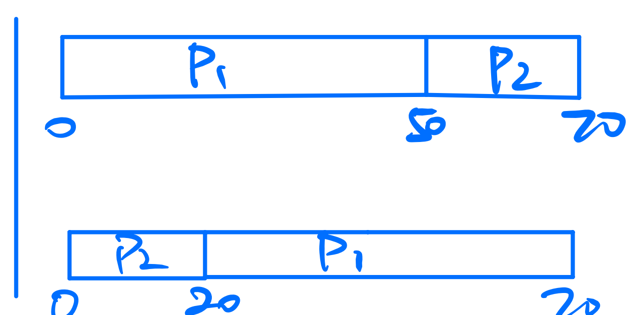
starts when you sign on to the system and ends when you sign off the system. During the job, we will interact with system.

you request and system act.

- But you need to "age" the priorities to avoid starvation.

Desirables (contd.)

- Consider 2 processes (P1,P2), where P1 has 50 ms CPU burst, while P2 has 20 ms CPU burst followed by 30 ms of I/O.
  - **Which would you schedule first?**
    - **P2 (i.e. in general give I/O bound processes higher priority).**
  - **But how do we classify/separate a process to be CPU/IO bound?**
  - **P1 prefers a time quantum of 50 while a time quantum of 20 suffices for P2**



PENNSTATE

Desirables (contd.)

Round-robin with priorities

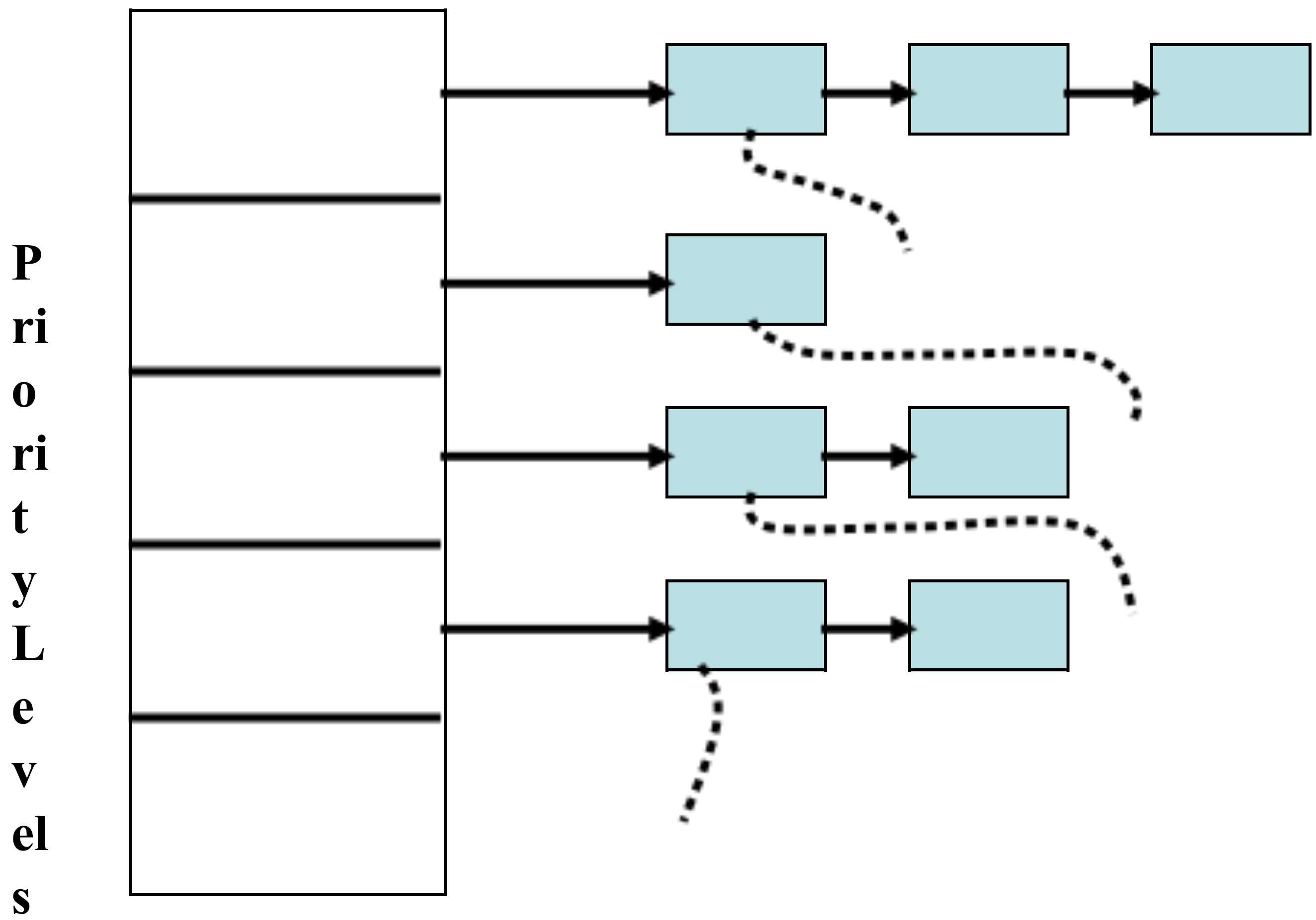
Accommodate Aging

Automatically classify processes to be CPU and I/O bound

Give higher priorities to I/O bound processes.

Give larger time quanta for CPU bound processes compared to I/O bound processes.

A Solution: Multi-level Feedback Queues



Multi-level feedback queues

- Pick the process at the head of the highest priority non-null queue.
- Give it the allotted time quantum.
- If its CPU burst is not yet done, move it to the tail of the queue of a lower priority level.

- Eventually you will find processes with large CPU bursts at much lower priority queues and processes with frequent I/O remaining at higher priority levels.
- Typically, the time quanta for higher priority levels are kept smaller than those for lower priority.

PENNSTATE

比例分配

Proportional-Share Schedulers

A type of scheduling that preallocate a certain amount of CPU Time to each of process.

- A generalization of round robin
- Process Pi given a CPU weight wi > 0
- The scheduler needs to ensure the following
  - *forall* i, j, |Ti(t1, t2)/Tj(t1,t2) - wi/wj| ≤ e
  - Given Pi and Pj were backlogged during [t1,t2]

P2

$$\frac{0.75s}{0.25s} = \frac{T(T_2 - T_1)}{T(T_2 - T_1)} = \frac{w_{p_2}}{w_{p_1}} = \frac{3}{1}$$

P1

— 抽奖调度

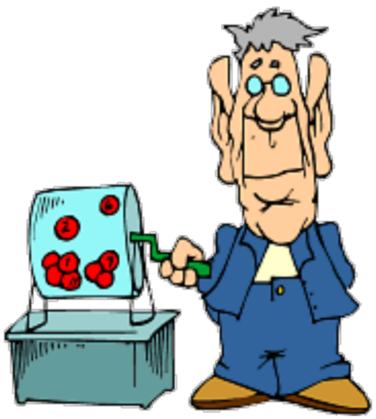
Lottery Scheduling

- Perhaps the simplest proportional-share scheduler
- Create lottery tickets equal to the sum of the weights of all processes
- Draw a lottery ticket and schedule the process that owns that ticket

Lottery Scheduling Example

P1= 6

P2= 9



1	4	7	10	13
2	5	8	11	14
3	6	9	12	15

9

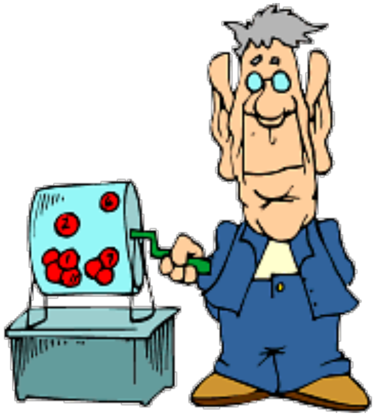
Schedule P2

Lottery Scheduling Example

P1= 6

P2= 9

1	4	7	10	13
2	5	8	11	14
3	6	9	12	15



3

Schedule P1

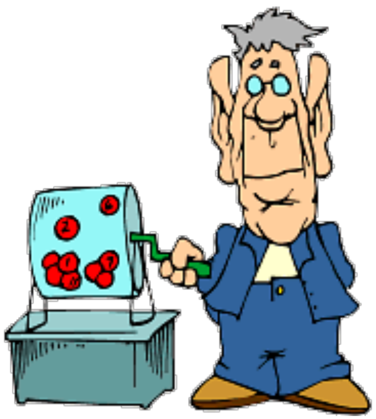
Lottery Scheduling Example

P1= 6

1	4
2	5
3	6

P2= 9

7	10	13
8	11	14
9	12	15



11

- As t $\longrightarrow \infty$, processes will get their share (unless they were blocked a lot)
- Problem with Lottery scheduling: Only probabilistic guarantee
- What does the scheduler have to do
  - When a new process arrives?
  - When a process terminates?

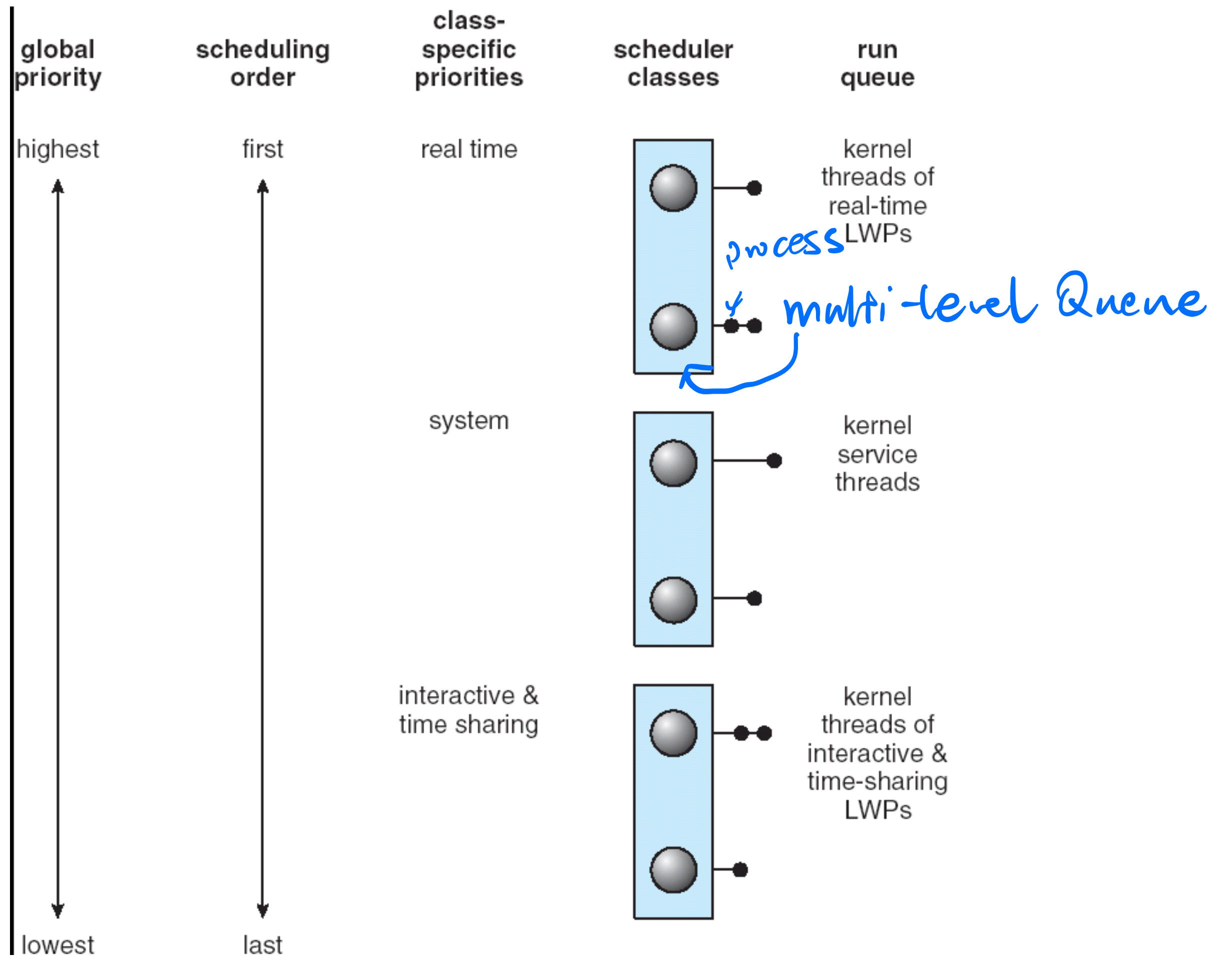
Schedule P2

Lottery Scheduling

- Exercise: Calculate the time complexity of the operations Lottery scheduling will involve

Scheduling in Systems

Solaris Scheduling



PENNSTATE

Solaris Dispatch Table

larger number higher priority

aged .

when the same process coming back to ready queue

prompt

CPU bound

IO bound

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

after 20ms, OS determine set priority 59 → 49, since it want this process take more cpu time

PENNSTATE

using higer number to present lower priority number.

Linux 2.5 Scheduling

- Two algorithms: time-sharing and real-time
- Time-sharing (still abstracted)
  - Two queues: *active* and *expired*
  - In active, until you use your entire time slice (quantum), then expired
    - Once in expired, Wait for all others to finish (fairness)
  - Priority recalculation -- based on waiting vs. running time
    - From 0-10 milliseconds
    - Add waiting time to value, subtract running time
    - Adjust the static priority
- Real-time
  - Soft real-time
  - Posix.1b compliant – two classes
    - FCFS and RR
    - Highest priority process always runs first

In Linus, how long the process going to run and still be consider to be run

↳ quantum

110 □ → P1 100ms

130 □ → P2 50ms

P1	P2	P1	P2	

0 10 40 50 70

Say P1 is I/O bound, P2 is cpu bound.

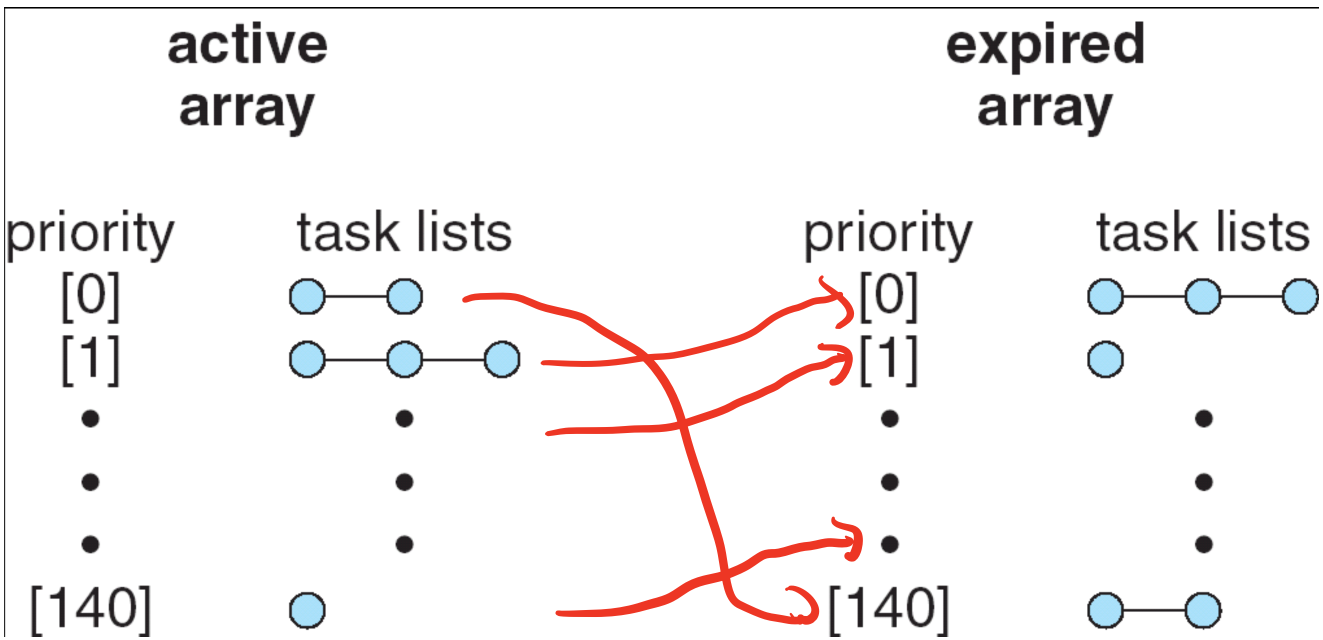
① P1 runs 10ms, then P2 runs 30ms, then P1 comes in again, then, Even though P2 need to run over

50 ms, but OS only gives P2 50ms. so P2 will only runs 20 ms. and then expired. then need to wait all others to expire.

The Relationship Between Priorities and Time-Slice length

numeric priority	relative priority		time quantum
0	highest	real-time tasks	200 ms
•			
•			
•			
99			
100		other tasks	
•			
•			
•			
140	lowest		10 ms

List of Tasks Indexed According to Priorities



Q: | us | x | us. |

cpu bound process, it will be lower its priority again and again until it is evicted.

how do we ensure to run right befor X and after X ?

PENNSTATE

Questions?