CS202 – Advanced Operating Systems

Scheduling

January 27, 2025

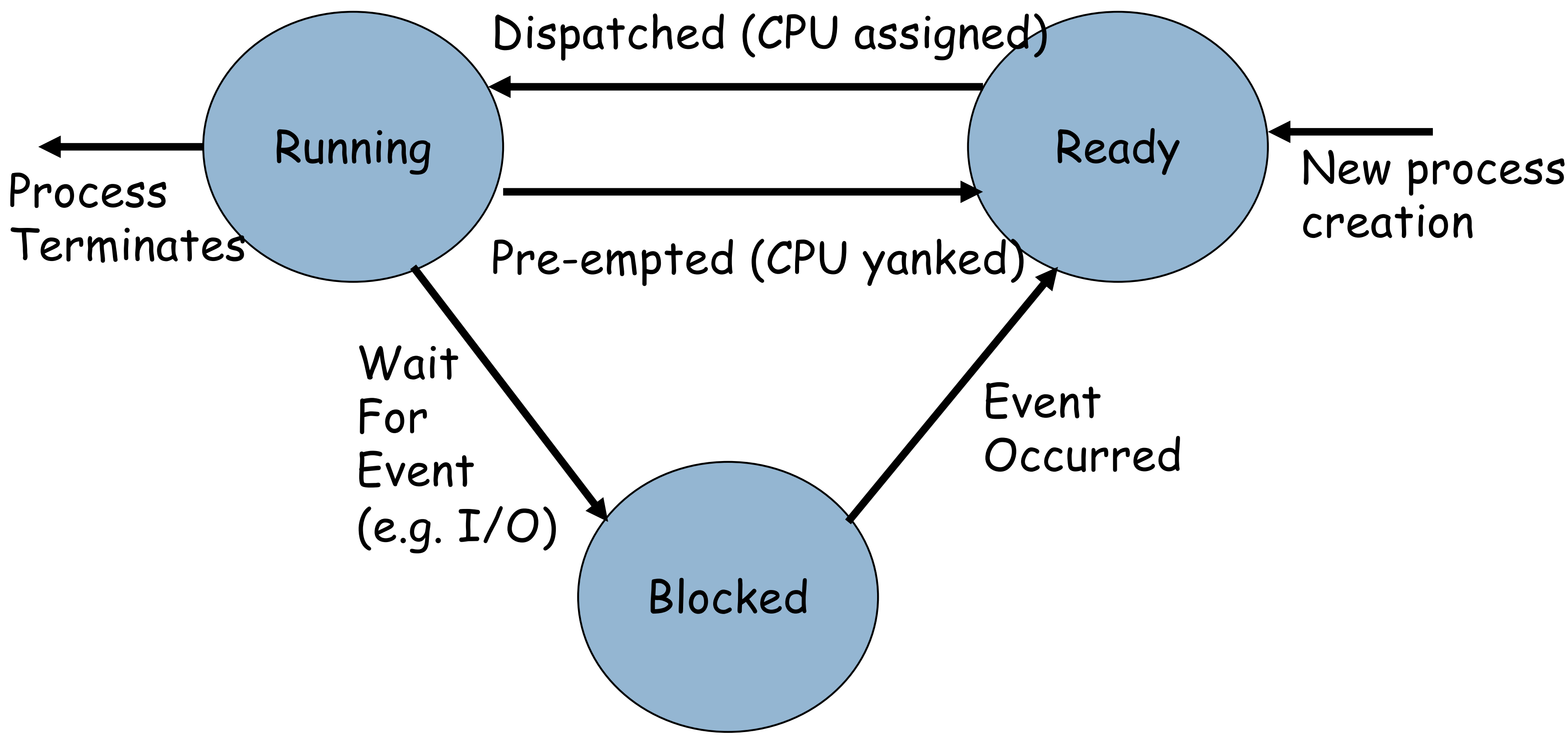
Scheduling

- We now move on to

```
next = schedule();
```

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

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.

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

- **Utilization/efficiency**: keep the CPU busy 100% of the time with useful work
- **Throughput**: maximize the number of jobs processed per hour.
- **Turnaround time**: from the time of submission to the time of completion.
- **Waiting time**: Sum of times spent (in Ready queue) waiting to be scheduled on the CPU.
- **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
  4. Terminates
- Scheduling for events 1 and 4 do not preempt a process
  - Process volunteers to give up the CPU

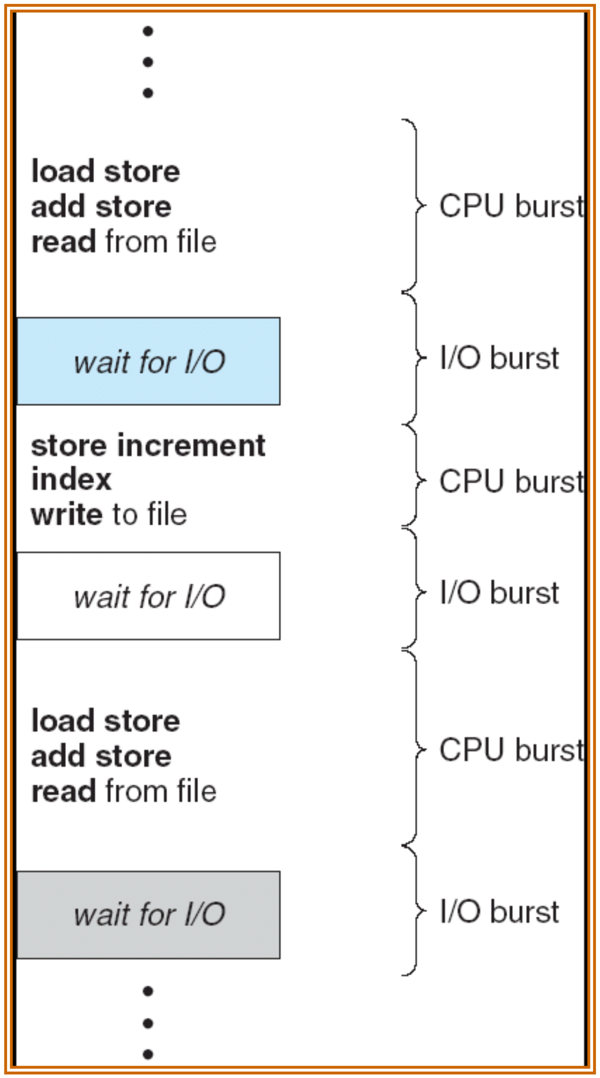
Preemptive vs Non-preemptive

- 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?

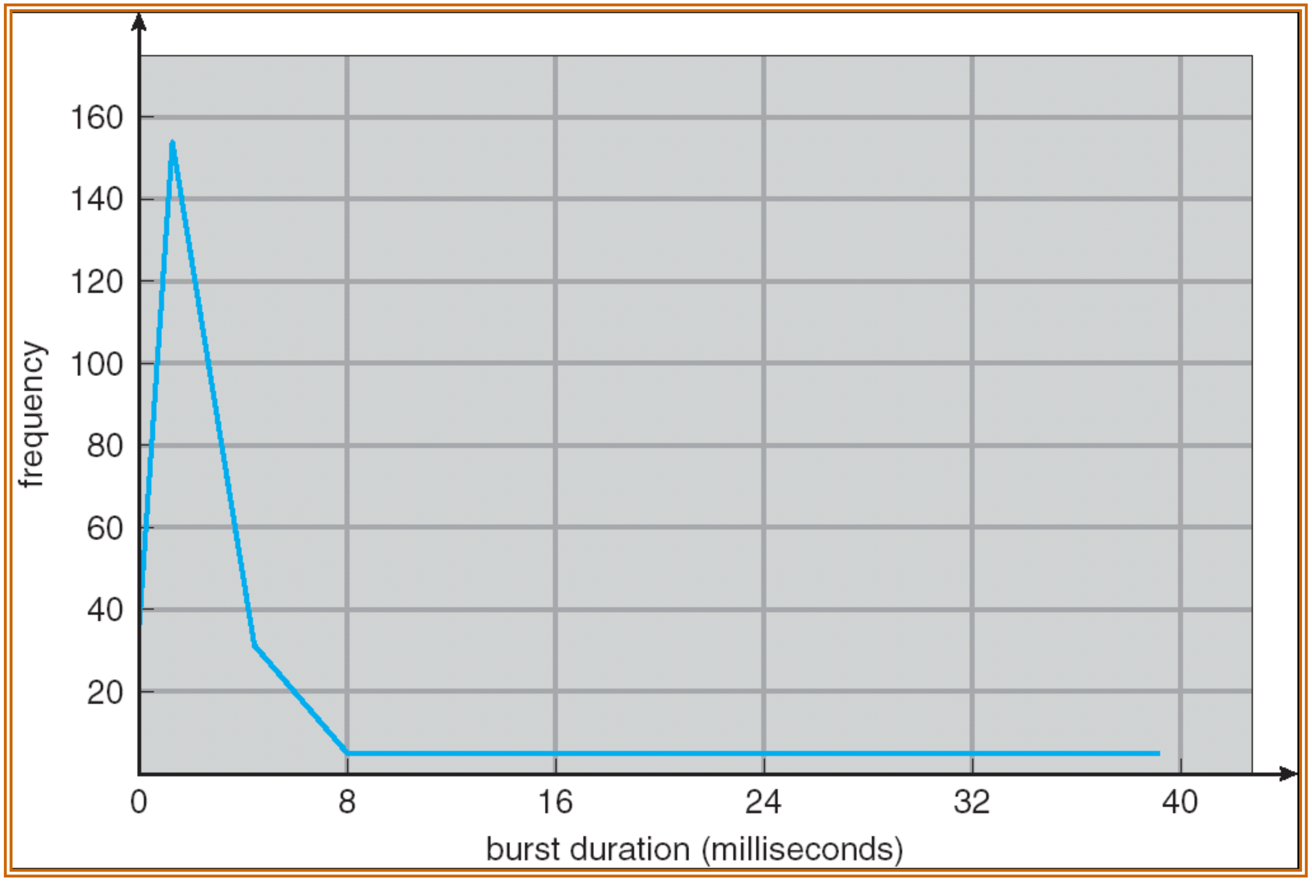
Bursts

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

Bursts


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

CPU Burst Duration



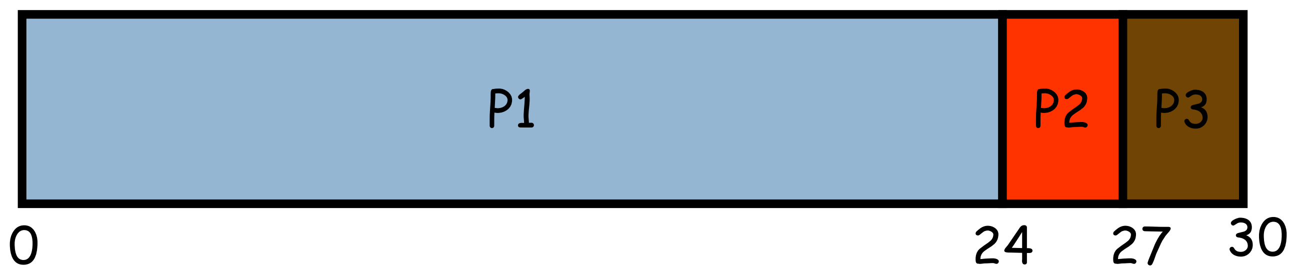
Scheduling Algorithms

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.
  - 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 + 15 + 13)/3 = 17 1/3

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

How do we estimate duration?

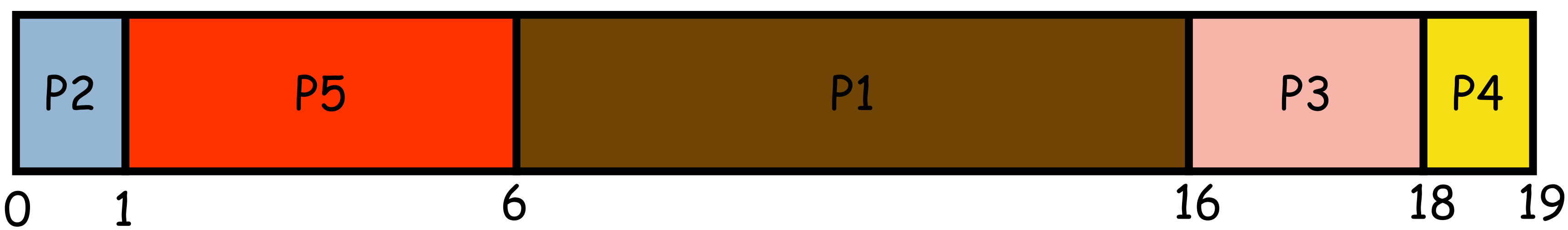
- Typically, the same job keeps coming back (either after I/O or newly) requesting for CPU.
- So, we may be able to use prior history.
- Say T(n) is the actual time taken the last time for which we estimated E(n), then we use an exponential averaging technique as follows:
  - $E(n+1) = a.T(n) + (1-a).E(n)$
- 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

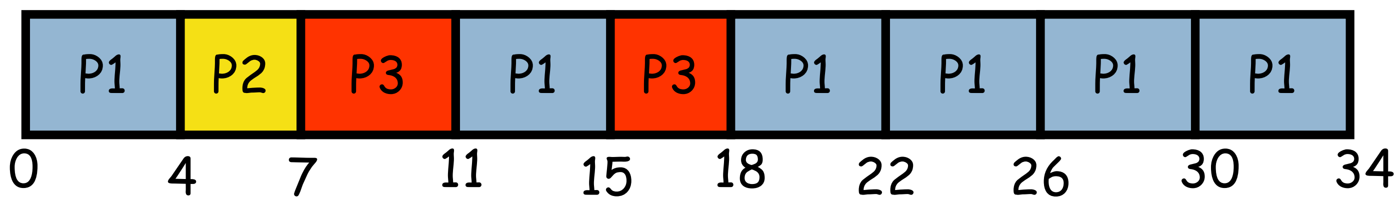
Round Robin (preemptive)

- Periodically switch from 1 process to another.
- You time slice the CPU among 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 = 4s



- Round robin is virtually sharing the CPU among processes giving each process the illusion that it is running in isolation (at 1/n-th CPU speed).
- Smaller the time quantum, the more realistic the illusion (note that when 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 ????
- But context switches are not free!
  - Saving/restoring registers
  - Switching address spaces
  - Indirect costs (cache pollution)

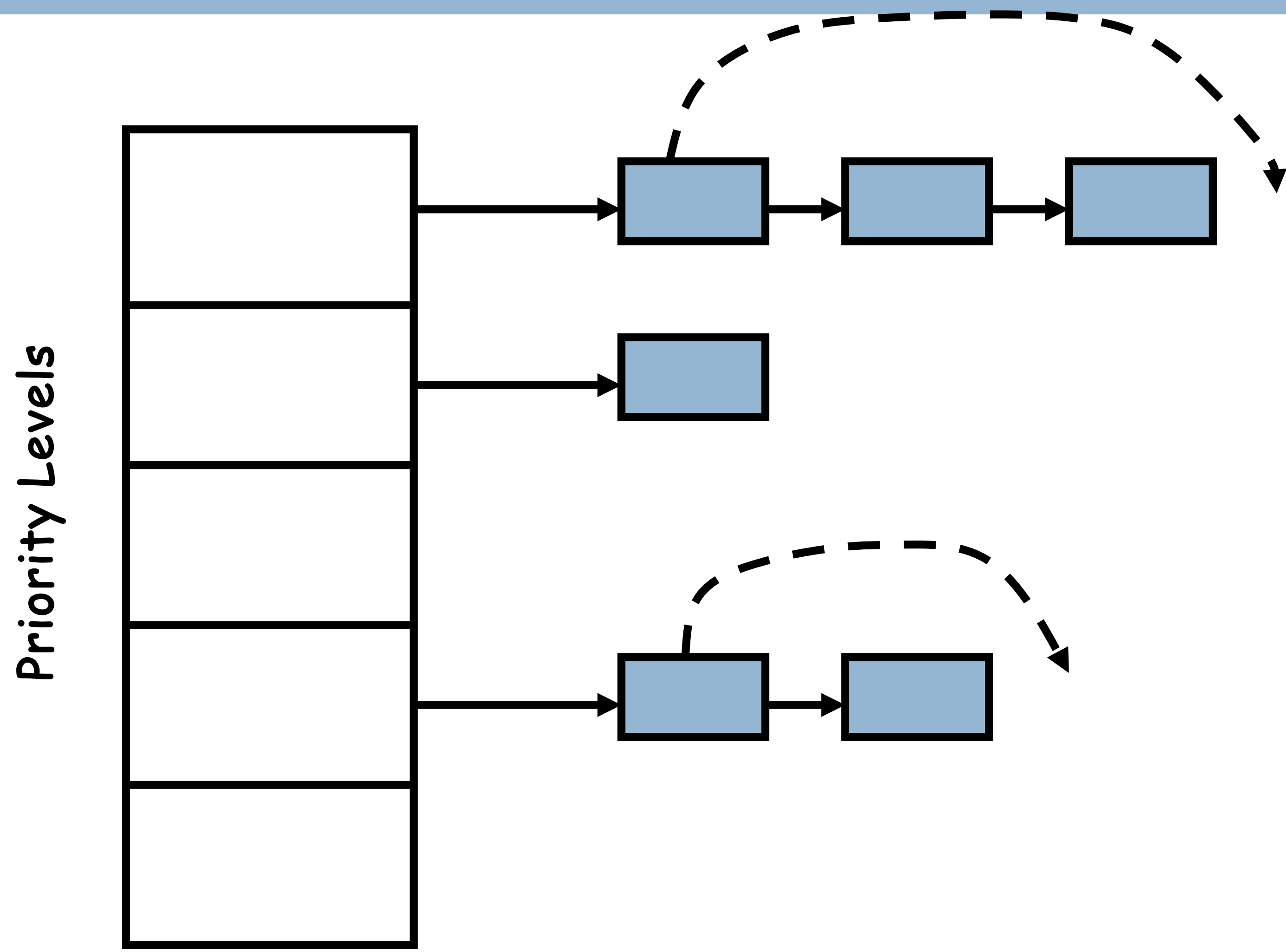
General rules of thumb

- 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 among those processes.

Round-robin with priority



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 non-preemptive strategies.
- But you need to “age” the priorities to avoid starvation.

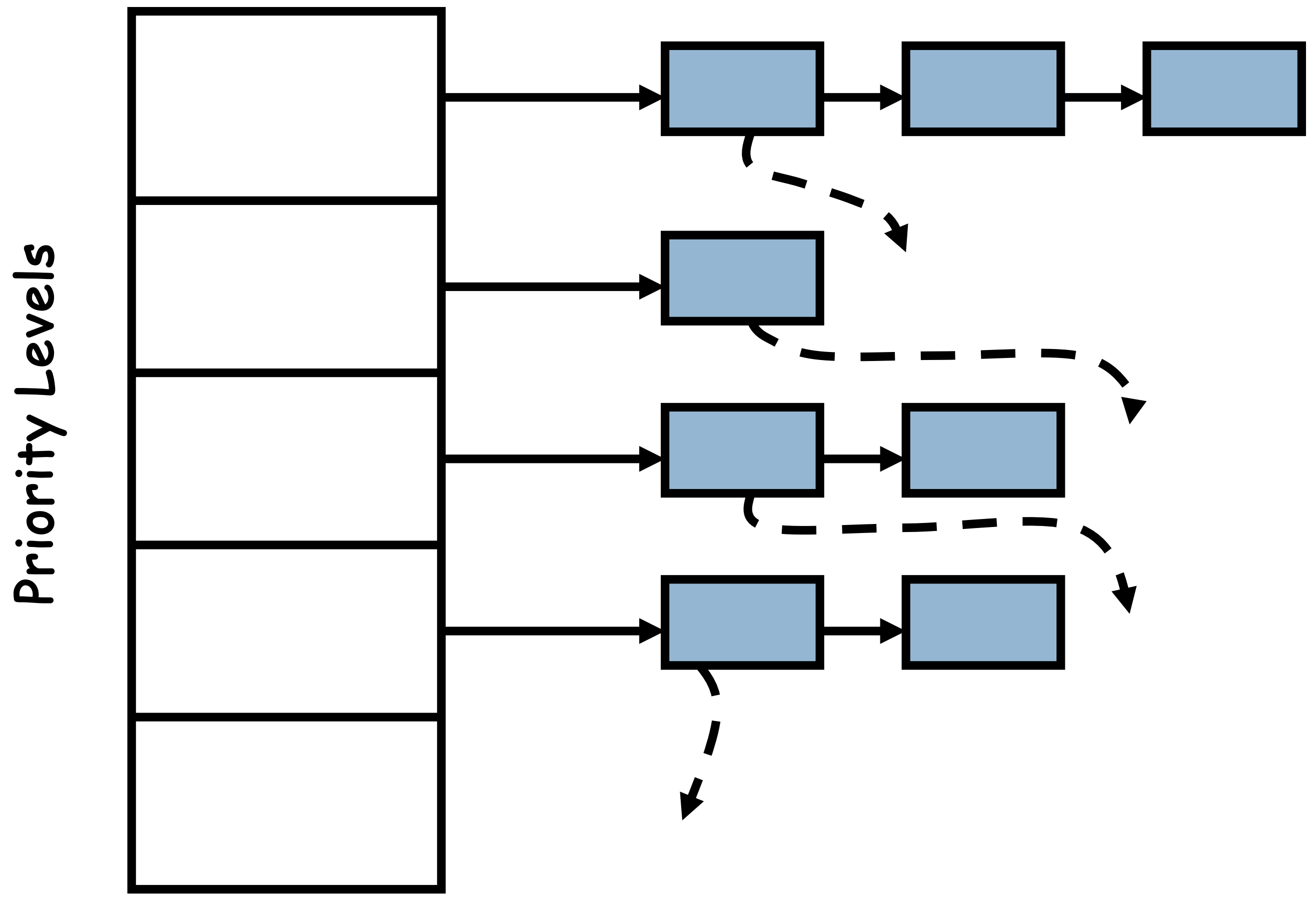
Desirables (contd.)

- Consider 2 processes (P1,P2), where P1 has 50ms CPU burst, while P2 has 20ms CPU burst followed by 30ms 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

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.
- Rules for demotion (what queue to assign when a process uses its allotted quantum) and promotion (what queue to assign when a process does not use its time quantum)

- 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.

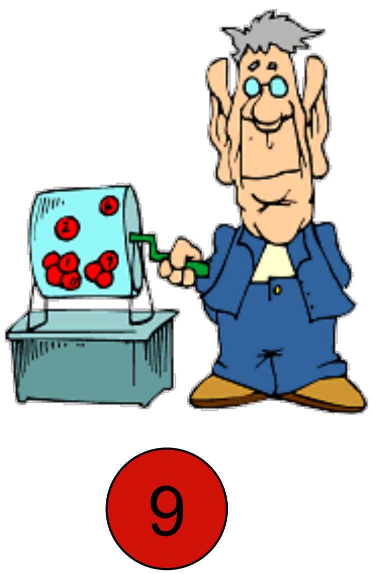
Proportional-Share Schedulers

- A generalization of round robin
- Process P_i given a CPU weight $w_i > 0$
- The scheduler needs to ensure the following
  - *forall* i, j, $|T_i(t_1, t_2)/T_j(t_1,t_2) - w_i/w_j| \le e$
  - I.e., ratio of time scheduled is essential same as ratio of weights

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

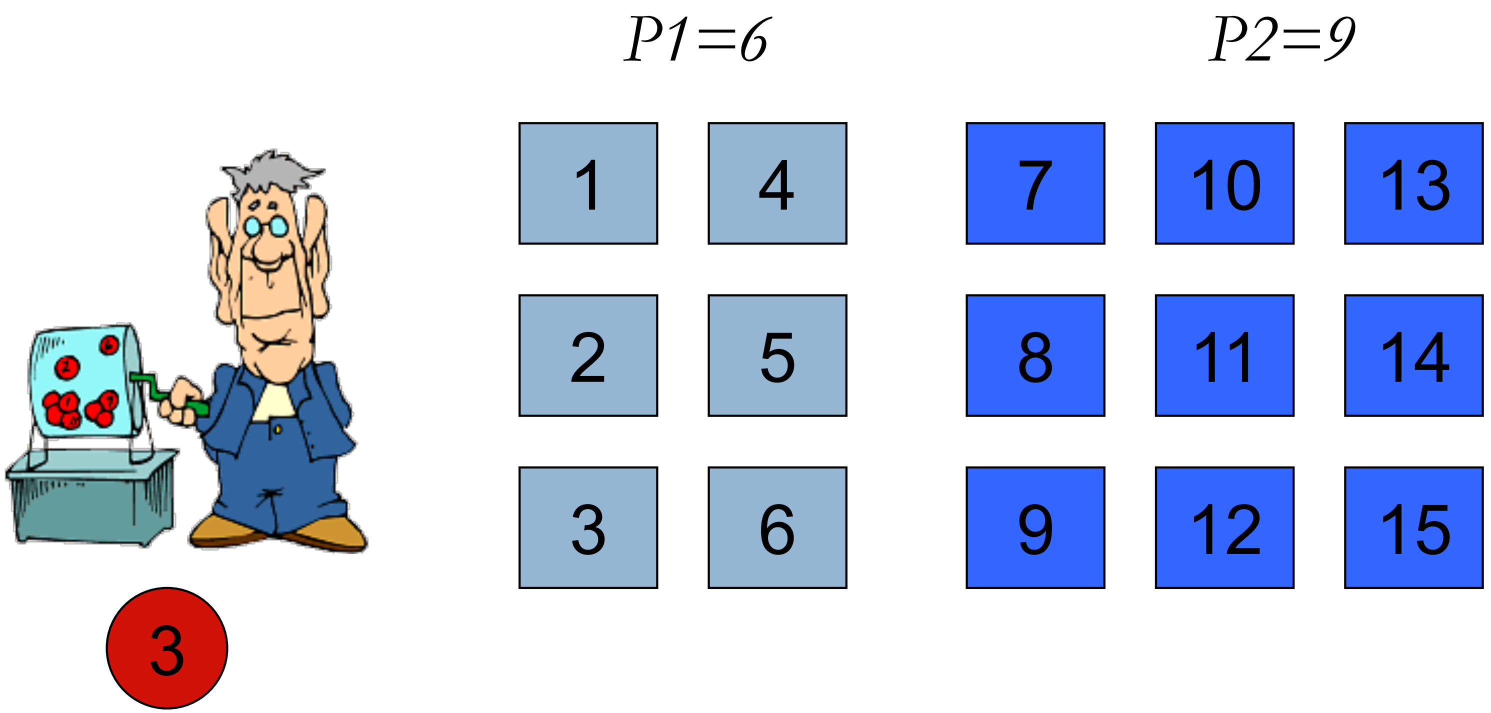
1	4
2	5
3	6

P2=9

7	10	13
8	11	14
9	12	15

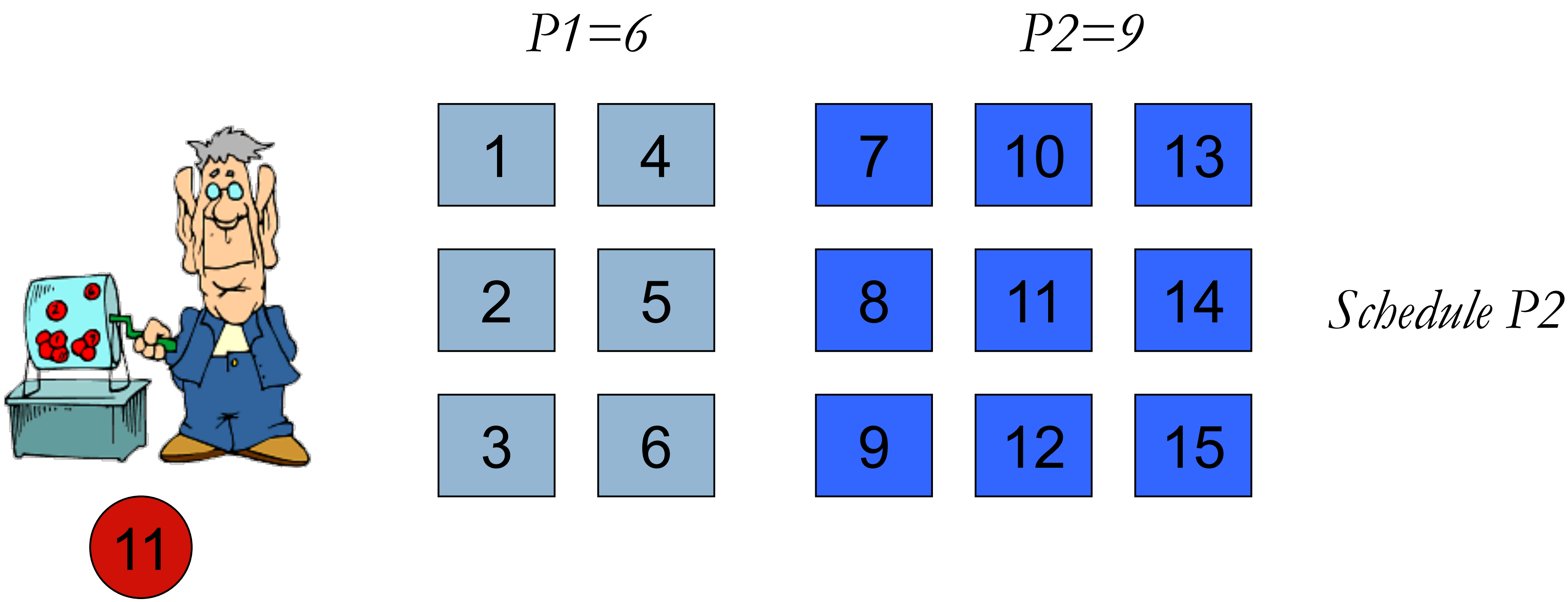
Schedule P2

Lottery Scheduling Example



Schedule P1

Lottery Scheduling Example



- As t →∞, 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?

Conclusions

- Today was a review of CPU scheduling
- Choosing the process to run to use the CPU optimally is a computational complex task
  - Made more difficult because we cannot predict how processes will execute in the future
- CPU scheduling provides a lot of knobs for control
  - CPU Bursts, preemption, priorities, aging, etc.
- CPU scheduling algorithms use this information to make heuristic decisions about scheduling
  - Need to know how these algorithms work

Questions