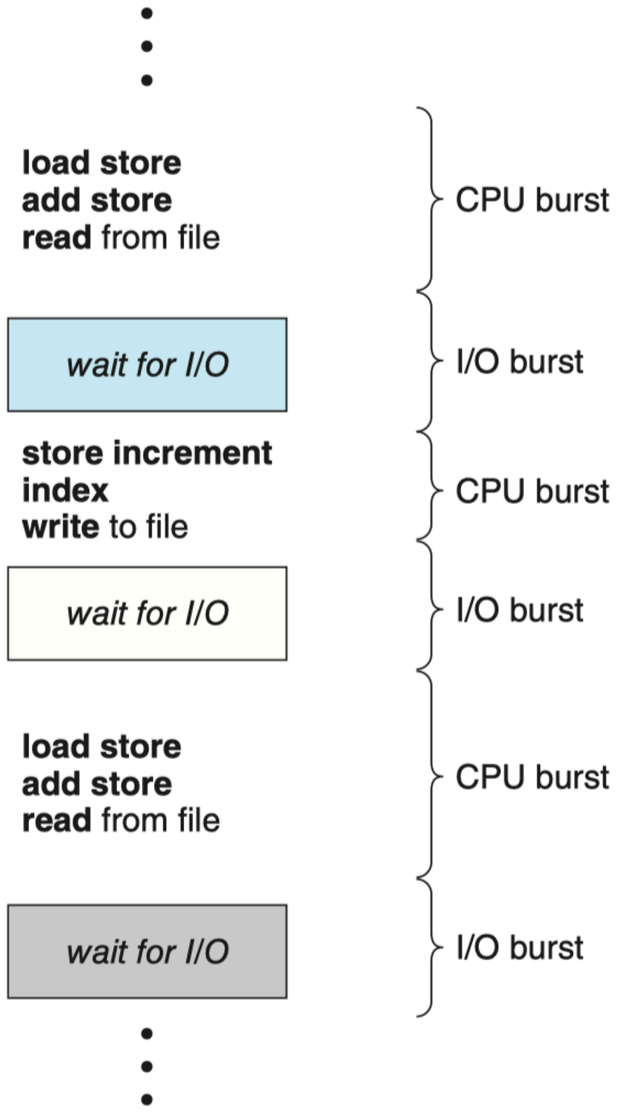
Scheduling

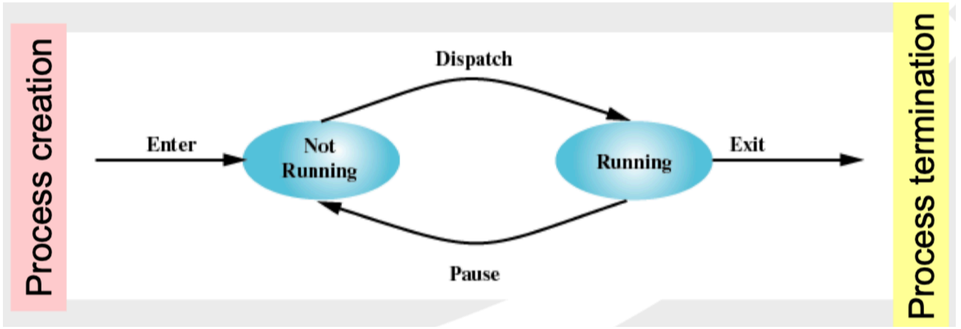
Objective of CPU Scheduling

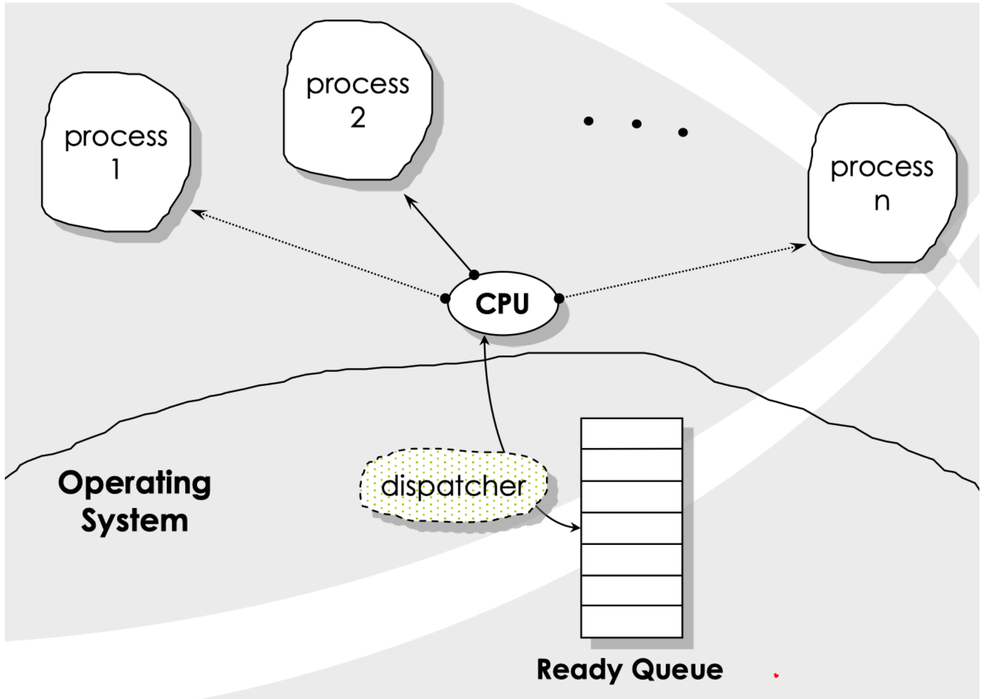
- **<u>Informally</u>: Use the CPU resources efficiently to perform all the user tasks in time**
- CPU waiting and the jobs waiting is a bad schedule
- Good schedule might still have one of them waiting - why?
- Jobs waiting because CPU is busy
- CPU waiting because there are no jobs (tasks launched by users or even system)
- Jobs have CPU bursts and IO bursts
- Running multiple jobs allow us to overlap the executions to minimize CPU wait times



CPU Scheduler

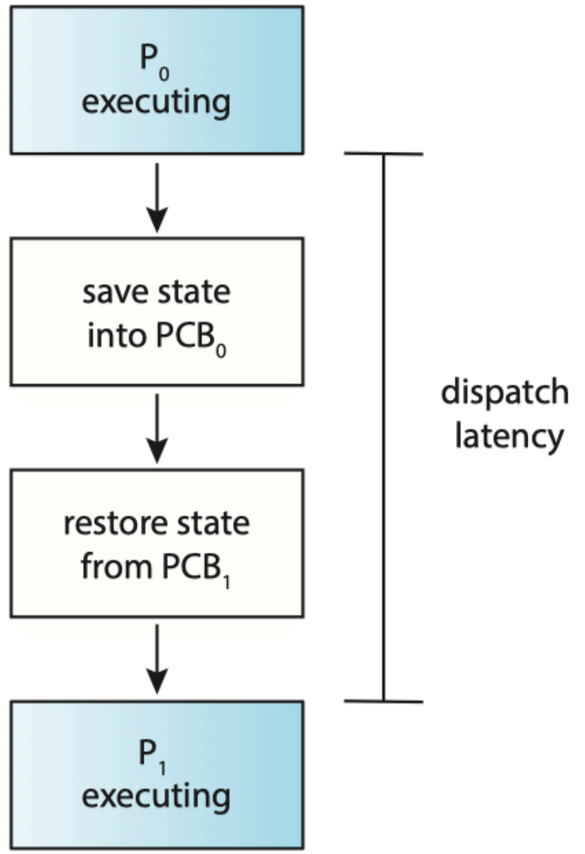
- When CPU becomes idle (done with the previous job), OS must select next job
- CPU scheduler is the one that selects the next job
- It picks the next job from the ready queue and makes it run in the CPU - called as dispatching
- **Dispatcher** is the component that actually swaps the processes





CPU Dispatcher

- Dispatcher gives control of the CPU to the process selected by the scheduler
- Dispatcher involves the following functions
  - Switching context from one process to another
  - Switching to user mode
  - Jumping to the proper location in the user program to resume that program
- Dispatcher must be fast - since it is invoked at every context switch
- Dispatch latency - time taken to stop one process and start another process



Preemptive and Nonpreemptive Scheduling

- CPU scheduling taken under the following four conditions
  - (1) When a process switches from running state to waiting state — calling waitpid()
  - (2) When a process switches from running state to ready state — timer interrupt
  - (3) When a process switches from waiting to ready state — completion of I/O
  - (4) When a process terminates
- (1) and (4) nonpreemptive or cooperative scheduling
- (2) and (3) are preemptive scheduling

Preemption Challenges

- Preemption of processes create challenges for OS kernel designers
- Should the kernel preempt the process' execution at the preemption point?
- A process might be doing a system call - kernel working on it - and preemption occurs. What should the kernel do? Defer the preemption until the system call processing can be suspend or preempt right away?
- OS kernel needs to be designed for preemption - suspending the system call execution and then resuming it later - makes OS kernel design very complicated
- Preemptive kernel is something that supports system call execution: suspend and resume

Scheduling Criteria

- To select a scheduling algorithm we need to have a way of comparing the performance delivered by the CPU scheduling algorithms
- **CPU utilization**: Want to keep the utilization as high possible (assuming there is work). CPU utilization is a percentage 0 to 100.

CPU utilization = capacity used/total capacity

- **Efficiency**: Don't do unnecessary work. Do work that advances the application requirements.

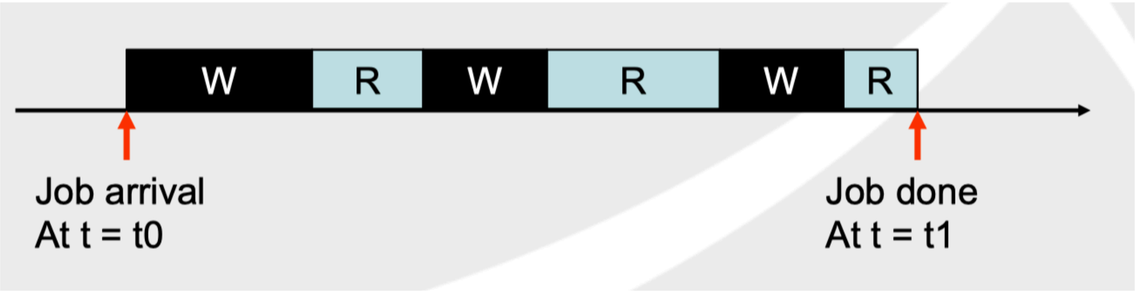
Efficiency = useful work/total work

More Scheduling Criteria

- **Throughput**: CPU is busy executing processes -> work done.

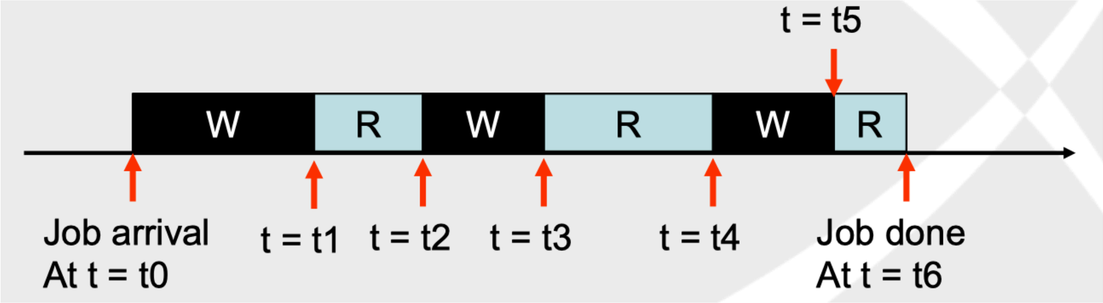
Throughput = number of processes completed per unit time

- **Turnaround time**: Measured from a single process' point-of-view. Interval from submission to completion is the turnaround time. Measured in wall clock time.

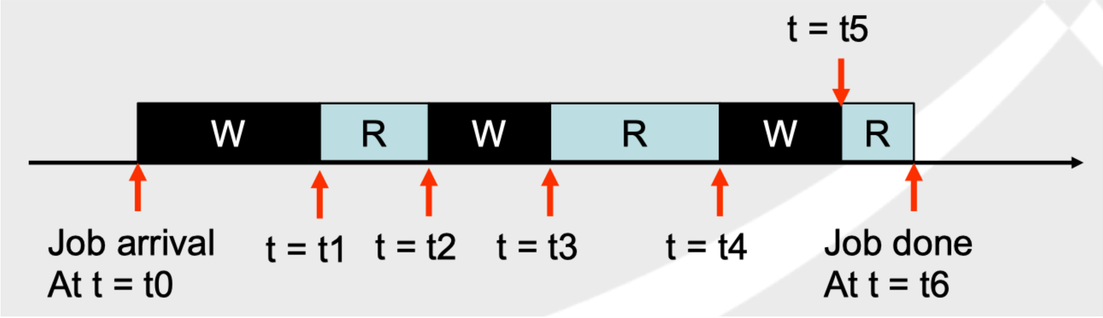


More Scheduling Criteria

- **Waiting time**: Sum of the periods spent waiting in the ready queue. CPU-scheduling algorithm does not affect the time a process executes or does I/O



- Response time: Time from submission to the time the request got the first response - start of the response is taken.



Scheduler Design

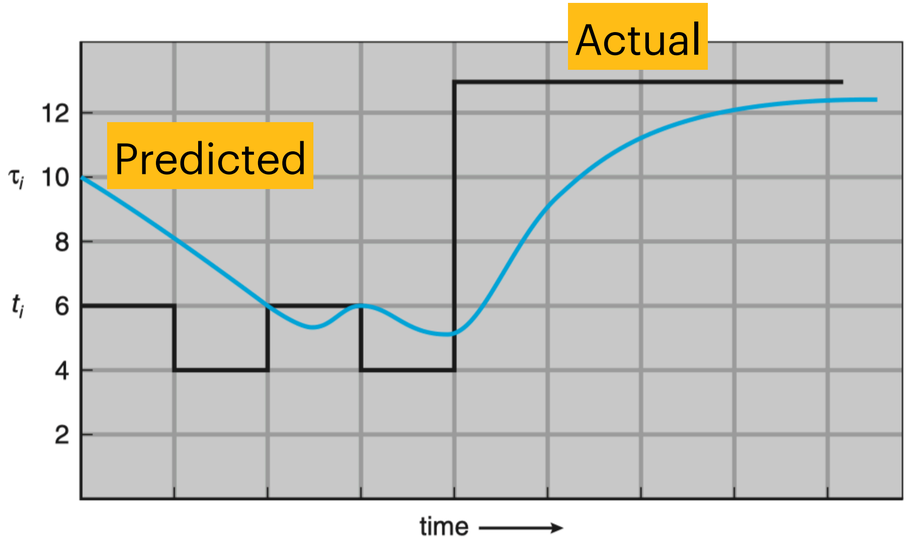
- Scheduler is designed for a purpose
  - OS for robotic vehicles need real-time computing support
  - OS for data centre needs to maximize number of jobs completed
  - OS for implanted medical devices need to be super reliable and energy efficient
  - OS for spacecraft need to be support redundant operation, reliable, real-time, etc
- How do we design a scheduler?
  - Select one or more primary performance criteria (previous list not complete)
  - Rank them in order of importance
  - Performance criteria might be independent or correlated
  - Design the scheduler by balancing the different objectives that could be conflicting

Scheduling Algorithms

- What do we need to schedule?
  - Number and types of CPUs - heterogeneous cores?
  - Number of jobs and access to the ready jobs
  - Arrival order of the jobs to be fair in selecting them
  - Job execution requirements (time remaining for execution, other resources that are needed - like memory, IO, etc)
  - Job priorities (which job is more important than others)
  - Job dependency (what other jobs need execution at the same time?)
  - Job conflict (reverse of dependency - don’t load them together - too much resource usage or other problems)

Estimating Runtimes of Jobs

- How can we estimate the runtime?
- Use prior historical data and some simple filtering algorithm - exponential averaging
- If $\alpha = 0$, then $\tau_{n+1} = \tau_n$ and recent history has no effect
- If $\alpha = 1$, then $\tau_{n+1} = t_n$, i.e., only the CPU burst matters
- Normally, $\alpha = 1/2$



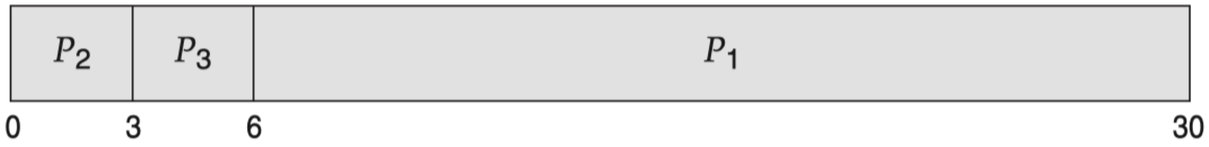
CPU burst (t_i)	6	4	6	4	13	13	13	…
"guess" (τ_i)	10	8	6	6	5	9	11	12	…

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n.$$

First-Come, First-Served Scheduling

- Job that requests the CPU first is allocated it first
- Implemented using the FIFO queue
- Average waiting time is often long - short jobs can be waiting on long ones
- If the arrival order is P_1, P_2, P_3, then average waiting time is 17 ms
- If the arrival order is P_2, P_3, P_1, AWT is 3ms

Process	Burst Time
P_1	24
P_2	3
P_3	3



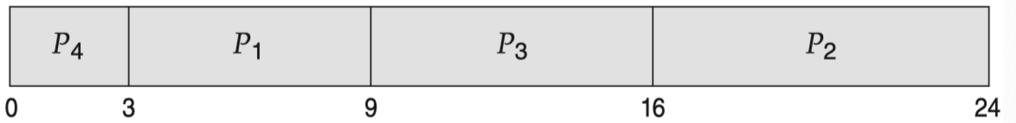
FCFS Problems

- We have CPU bound and IO bound processes - CPU bound processes like to run in the CPU for long times!
- IO bound processes wait on the IO device(s)
- Consider a situation where all the IO bound processes are waiting on IO
- CPU bound process gets on the CPU - going to hold the CPU for a long time
- All the IO bound processes complete IO and get back on the ready queue -need to wait there for a while — **convoy effect**
- This cycle repeats

Shortest-Job-First Scheduling

- Each job has a burst size (amount of CPU it would consume next)
- We assume a set of jobs have already arrived and waiting to be selected in the ready queue
- Select the job with the shortest burst size for running
- SJF is optimal in the sense it provides the minimal average waiting time
- It is not possible to implement SJF directly for CPU scheduling - burst size unknown for incoming jobs
- Approximate SJF - use predictions for burst sizes

	Process	Burst Time
First arrival	P_1	6
	P_2	8
	P_3	7
	P_4	3



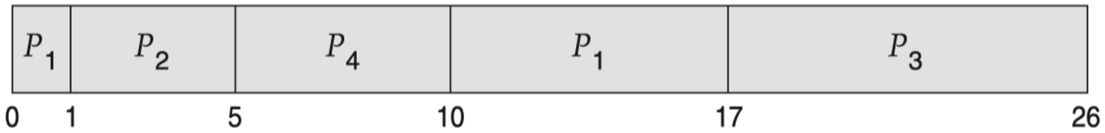
Average waiting time
SJF = 7 ms
FCFS = 10.25 ms

Shortest Remaining Time Next

Preemptive SJF

- We consider SJF as nonpreemptive - selected job keeps running no matter
- What if another shorter job arrives? SJF does not preempt the current job
- SRTN does preempt the current job and select the smallest among all the jobs again using remaining burst sizes

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

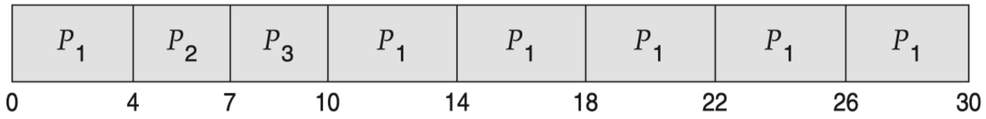


Average waiting time = 6.5 ms

Round-Robin Scheduling

- Round-robin is similar to FCFS, but preemption is added to switch between processes - so long processes do not hog the CPU
- Quantum (time slice) defines the length at which running process is preempted
- Ready queue is a circular queue - scheduler goes around the ready queue allocating CPU
- An executing process might have more time left - goes back to ready queue
- An executing process might be done within a quantum - need to select another job

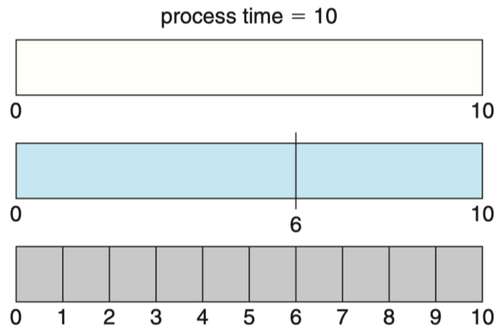
Process	Burst Time
P_1	24
P_2	3
P_3	3



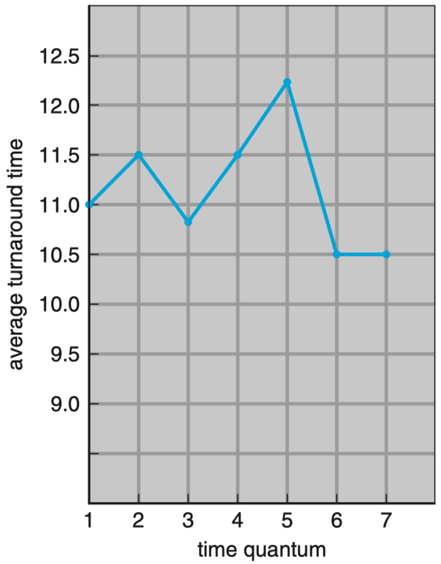
Average waiting time = 5.66 ms

Round-Robin Scheduling Issues

- A job gets $1/n$ of the CPU every q time units (quantum)
- A job needs to wait at most $(n-1) \times q$ times units before running again
  - More jobs in the queue is going to be a job feel sluggish
  - Larger quantum size is also going to make a job feel slow
- Small quantum size would mean many context switches - wastes work - overall efficiency decreases
- Turnaround time is affected by quantum size - quantum size equal to the average job size gives good performance



quantum	context switches
12	0
6	1
1	9

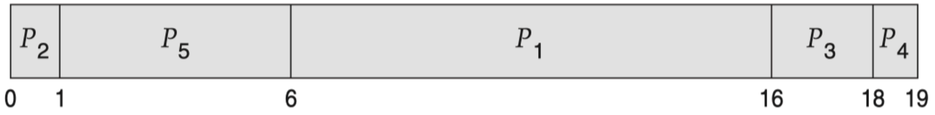


process	time
P_1	6
P_2	3
P_3	1
P_4	7

Priority Scheduling

- Priority scheduling - CPU is given to the process with the highest priority
- Equal priority processes are handled using a FCFS algorithm
- Low priority numbers mean high priority (not always, but common convention)
- SJF is special case of priority scheduling
- Priority scheduling can be preemptive or non preemptive
- Preemptive priority scheduler will evict a running process that has a lower priority than the arriving process
- Non preemptive priority scheduler will keep the ready queue sorted by the priority - priority queue!

	Process	Burst Time	Priority
First arrival	P_1	10	3
	P_2	1	1
	P_3	2	4
	P_4	1	5
	P_5	5	2



Average waiting time is 8.2 ms

Priority Queue Scheduling Issues

- Priority scheduling can have indefinite blocking - **starvation**
- Happens when higher priority jobs keep arriving - low priority jobs are in the queue forever
- Solution is aging - gradually increasing the priority of jobs that are not selected for execution
- We can also combine RR and Priority
- Jobs with equal priority are executed using RR

Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3