# Scheduling



# What is Scheduling?

- On a multi-programmed system
  - We may have more than one *Ready* process
- On a batch system
  - We may have many jobs waiting to be run
- On a multi-user system
  - We may have many users concurrently using the system

- The ***scheduler*** decides who to run next.
  - The process of choosing is called *scheduling*.



# Is scheduling important?

- It is not in certain scenarios
  - If you have no choice
    - Early systems
      - Usually batching
      - Scheduling algorithm simple
        - » Run next on tape or next on punch tape
  - Only one thing to run
    - Simple PCs
      - Only ran a word processor, etc….
    - Simple Embedded Systems
      - TV remote control, washing machine, etc….

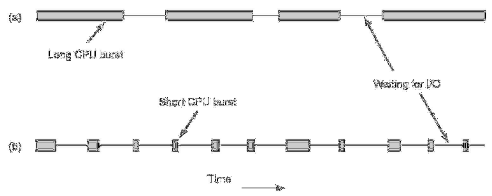


# Is scheduling important?

- It is in most realistic scenarios
  - Multitasking/Multi-user System
    - Example
      - Email daemon takes 2 seconds to process an email
      - User clicks button on application.
    - Scenario 1
      - Run daemon, then application
        - » System appears really sluggish to the user
    - Scenario 2
      - Run application, then daemon
        - » Application appears really responsive, small email delay is unnoticed
- Scheduling decisions can have a dramatic effect on the perceived performance of the system
  - Can also affect correctness of a system with deadlines



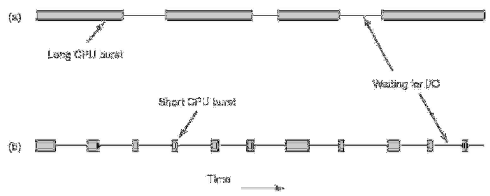
# Application Behaviour




- Bursts of CPU usage alternate with periods of I/O wait

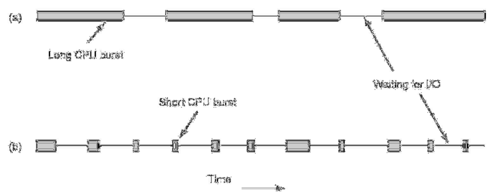


# Application Behaviour




a) CPU-Bound process
   - Spends most of its computing
   - Time to completion largely determined by received CPU time

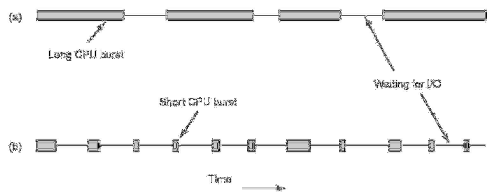
# Application Behaviour




b) I/O-Bound process
- Spend most of its time waiting for I/O to complete
  - Small bursts of CPU to process I/O and request next I/O
- Time to completion largely determined by I/O request time



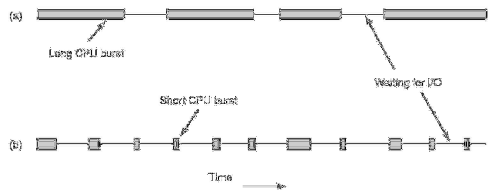
# Observations




- Generally, technology is increasing CPU speed much faster than I/O speed
  - ⇒ CPU bursts becoming shorter, I/O waiting is relatively constant
  - ⇒ Processes are becoming more I/O bound



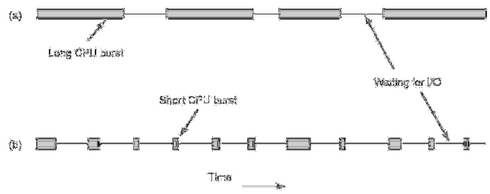
# Observations




- We need a mix of CPU-bound and I/O-bound processes to keep both CPU and I/O systems busy
- Process can go from CPU- to I/O-bound (or vice versa) in different phases of execution



# Observations




- Choosing to run an I/O-bound process delays a CPU-bound process by very little
- Choosing to run a CPU-bound process prior to an I/O-bound process delays the next I/O request significantly
  - No overlap of I/O waiting with computation
  - Results in device (disk) not as busy as possible

⇒ Generally, favour I/O-bound processes over CPU-bound processes

# When is scheduling performed?

- A new process
  - Run the parent or the child?
- A process exits
  - Who runs next?
- A process waits for I/O
  - Who runs next?
- A process blocks on a lock
  - Who runs next? The lock holder?
- An I/O interrupt occurs
  - Who do we resume, the interrupted process or the process that was waiting?
- On a timer interrupt? (See next slide)

- Generally, a scheduling decision is required when a process (or thread) can no longer continue, or when an activity results in more than one ready process.



# Preemptive versus Non-preemptive Scheduling

- Non-preemptive
  - Once a thread is in the *running* state, it continues until it completes, blocks on I/O, or voluntarily yields the CPU
  - A single process can monopolised the entire system
- Preemptive Scheduling
  - Current thread can be interrupted by OS and moved to *ready* state.
  - Usually after a timer interrupt and process has exceeded its maximum run time
    - Can also be as a result of higher priority process that has become *ready* (after I/O interrupt).
  - Ensures fairer service as single thread can't monopolise the system
    - Requires a timer interrupt

## Categories of Scheduling Algorithms

- The choice of scheduling algorithm depends on the goals of the application (or the operating system)
  - No one algorithm suits all environments
- We can roughly categorise scheduling algorithms as follows
  - Batch Systems
    - No users directly waiting, can optimise for overall machine performance
  - Interactive Systems
    - Users directly waiting for their results, can optimise for users perceived performance
  - Realtime Systems
    - Jobs have deadlines, must schedule such that all jobs (mostly) meet their deadlines.



## Goals of Scheduling Algorithms

- All Algorithms
  - Fairness
    - Give each process a *fair* share of the CPU
  - Policy Enforcement
    - What ever policy chosen, the scheduler should ensure it is carried out
  - Balance/Efficiency
    - Try to keep all parts of the system busy



## Goals of Scheduling Algorithms

- Batch Algorithms
  - Maximise *throughput*
    - Throughput is measured in jobs per hour (or similar)
  - Minimise *turn-around time*
    - Turn-around time ($T_r$)
      - difference between time of completion and time of submission
      - Or waiting time ($T_w$) + execution time ($T_e$)
  - Maximise *CPU utilisation*
    - Keep the CPU busy
    - Not as good a metric as overall throughput



## Goals of Scheduling Algorithms

- Interactive Algorithms
  - Minimise *response time*
    - Response time is the time difference between issuing a command and getting the result
      - E.g selecting a menu, and getting the result of that selection
    - Response time is important to the user's perception of the performance of the system.
  - Provide *Proportionality*
    - Proportionality is the user expectation that short jobs will have a short response time, and long jobs can have a long response time.
    - Generally, favour short jobs



## Goals of Scheduling Algorithms

- Real-time Algorithms
  - Must meet deadlines
    - Each job/task has a deadline.
    - A missed deadline can result in data loss or catastrophic failure
      - Aircraft control system missed deadline to apply brakes
  - Provide Predictability
    - For some apps, an occasional missed deadline is okay
      - E.g. DVD decoder
    - Predictable behaviour allows smooth DVD decoding with only rare skips



## Scheduling Algorithms

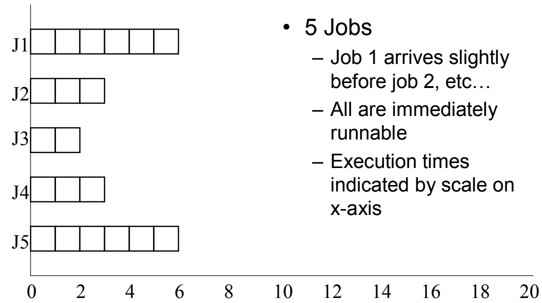
Batch Systems

## First-Come First-Served (FCFS)

- Algorithm
  - Each job is placed in single queue, the first job in the queue is selected, and allowed to run as long as it wants.
  - If the job blocks, the next job in the queue is selected to run
  - When a blocked jobs becomes ready, it is placed at the end of the queue



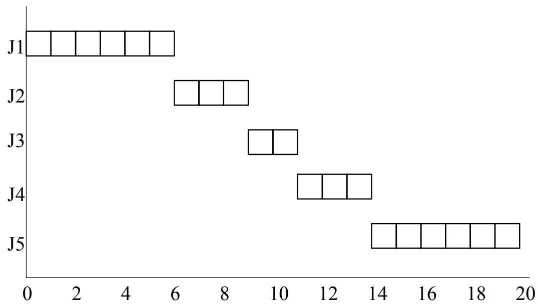
## Example




- 5 Jobs
  - Job 1 arrives slightly before job 2, etc…
  - All are immediately runnable
  - Execution times indicated by scale on x-axis



## FCFS Schedule






## FCFS

- Pros
  - Simple and easy to implement
- Cons
  - I/O-bound jobs wait for CPU-bound jobs
  - ⇒Favours CPU-bound processes
    - Example:
      - Assume 1 CPU-bound process that computes for 1 second and blocks on a disk request. It arrives first.
      - Assume an I/O bound process that simply issues a 1000 blocking disk requests (very little CPU time)
      - FCFS, the I/O bound process can only issue a disk request per second
        - » the I/O bound process take 1000 seconds to finish
      - Another scheme, that preempts the CPU-bound process when I/O-bound process are ready, could allow I/O-bound process to finish in 1000* average disk access time.

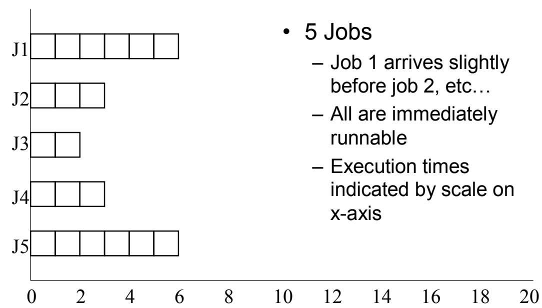


## Shortest Job First

- If we know (or can estimate) the execution time *a priori,* we choose the shortest job first.
- Another non-preemptive policy

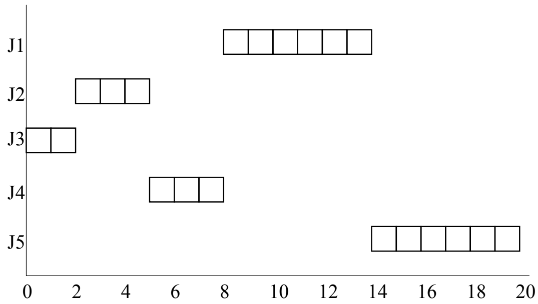


## Our Previous Example




- 5 Jobs
  - Job 1 arrives slightly before job 2, etc…
  - All are immediately runnable
  - Execution times indicated by scale on x-axis

## Shortest Job First






## Shortest Job First

- Con
  - May starve long jobs
  - Needs to predict job length
- Pro
  - Minimises average turnaround time (if, and only if, all jobs are available at the beginning)
  - Example: Assume for processes with execution times of *a, b, c, d.*
    - *a* finishes at time *a*, *b* finishes at *a + b*, *c* at *a + b + c*, and so on
    - Average turn-around time is (4*a* + 3*b* + 2*c* + *d*)/4
    - Since *a* contributes most to average turn-around time, it should be the shortest job.

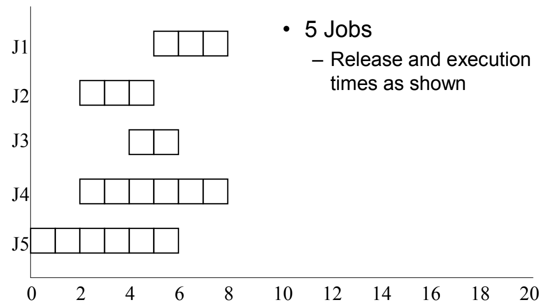


## Shortest Remaining Time First

- A preemptive version of shortest job first
- When ever a new jobs arrive, choose the one with the shortest remaining time first
  - New short jobs get good service



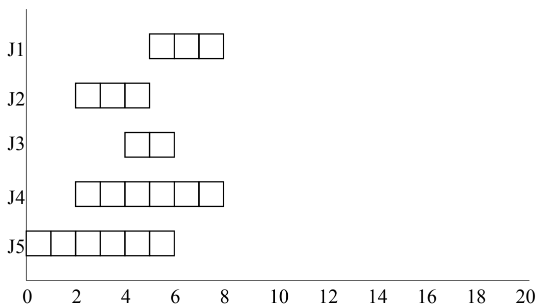
## Example




- 5 Jobs
  - Release and execution times as shown

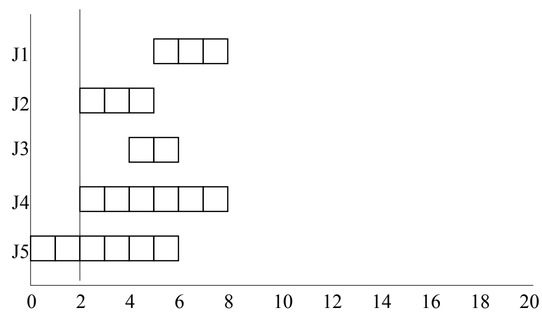


## Shortest Remaining Time First

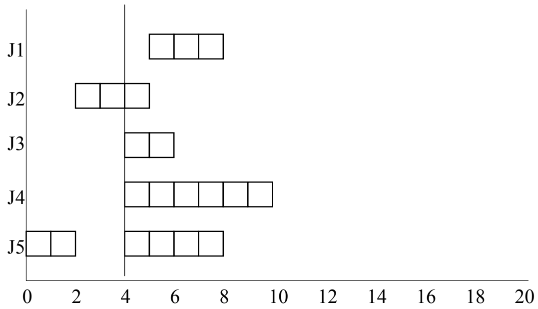





## Shortest Remaining Time First

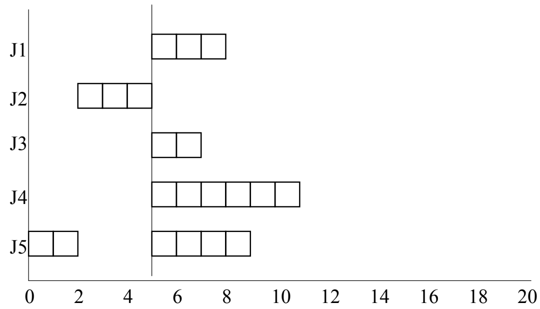


## Shortest Remaining Time First



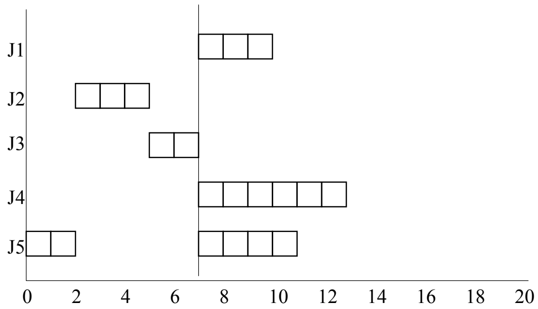



## Shortest Remaining Time First



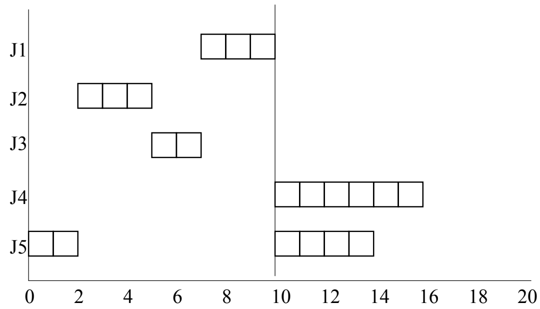



## Shortest Remaining Time First



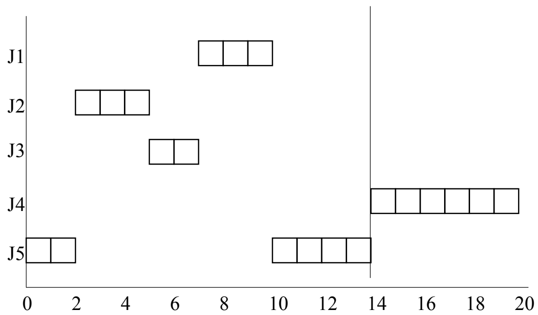



## Shortest Remaining Time First



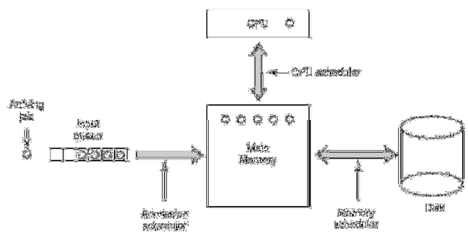



## Shortest Remaining Time First






## Scheduling in Batch Systems



Three level scheduling

## Three Level Scheduling

- Admission Scheduler
  - Also called *long-term* scheduler
  - Determines when jobs are *admitted* into the system for processing
  - Controls degree of multiprogramming
  - More processes ⇒ less CPU available per process



## Three Level Scheduling

- CPU scheduler
  - Also called *short-term* scheduler
  - Invoked when ever a process blocks or is released, clock interrupts (if preemptive scheduling), I/O interrupts.
  - Usually, this scheduler is what we are referring to if we talk about a *scheduler*.



## Three Level Scheduling

- Memory Scheduler
  - Also called *medium-term* scheduler
  - Adjusts the degree of multiprogramming via suspending processes and swapping them out