Lecture 7: Operating System

Process Management: Scheduling

Dr. Tushar, Mosaddek Hossain Kamal
Professor

Computer Science and Engineering, University of Dhaka,

Third Year Fall Semester (July–Dec), Academic Year: 2023

CSE3201: Operating System

November 14, 2024

Outline

Scheduling

What is Scheduling

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

The **Scheduler** decides who to run next

- The process of choosing is called scheduling

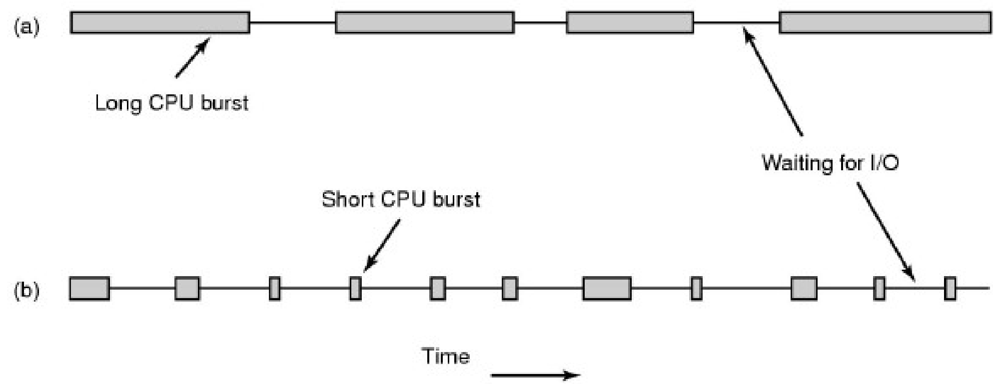
Is scheduling important?

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 or Smart Phone
    - Only ran a word processor, etc. . . .
    - Facebook or Google Map
  - Simple and Smart Embedded Systems
    - TV remote control, washing machine, etc. . . .
    - IoT Devices

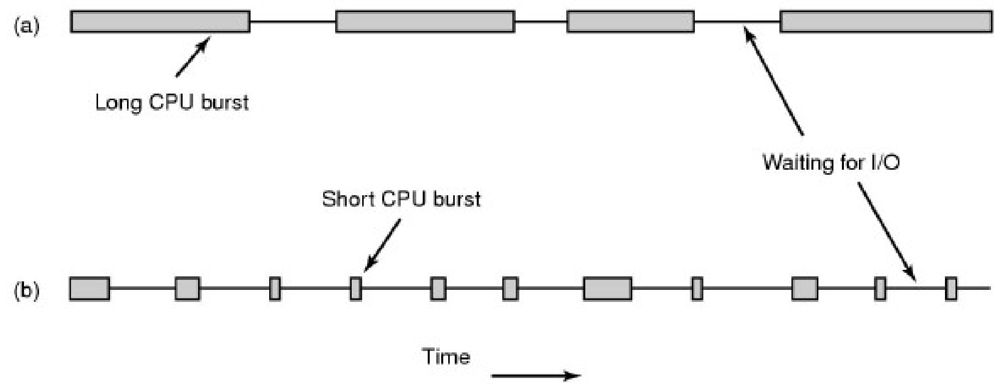
Application Behavior

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



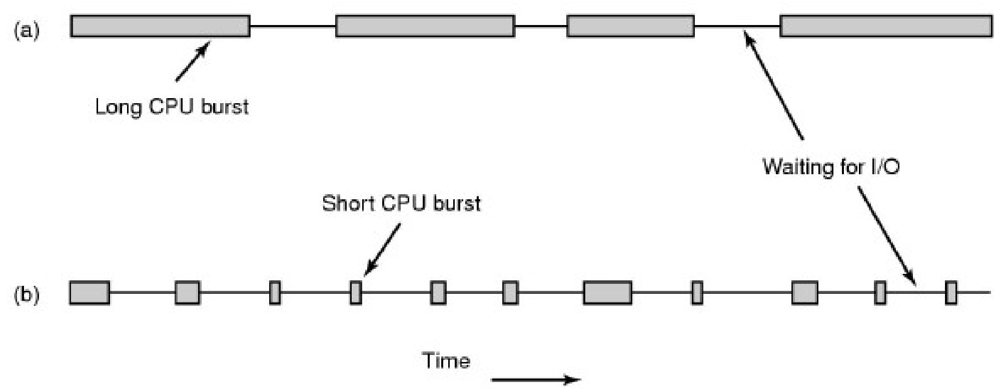
1. CPU-Bound process
   - Spends most of its computing
   - Time to completion largely determined by received CPU time
2. I/O-Bound process
   - Spend most of its time waiting for I/O to complete
     - Small bursts of CPU to process I/O and request next I/O
   - Time to completion largely determined 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
- 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?

When is scheduling performed?

- A new process is created – 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

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 monopolized the entire system

Preemptive

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 monopolize the system
  - Requires a timer interrupt

Categories of Scheduling

Categories of Scheduling

- 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

Goals

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

Batch Algorithms

- Maximise throughput
  - Throughput is measured in jobs per hour (or similar)
- Minimize 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

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

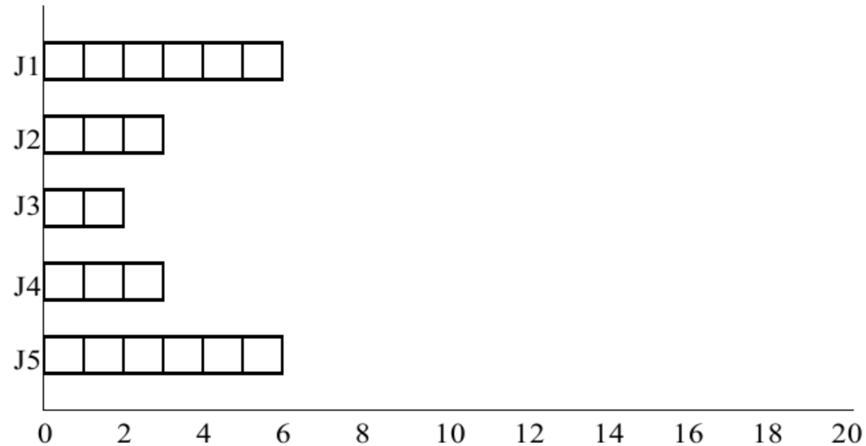


Figure 1

FCFS Schedule

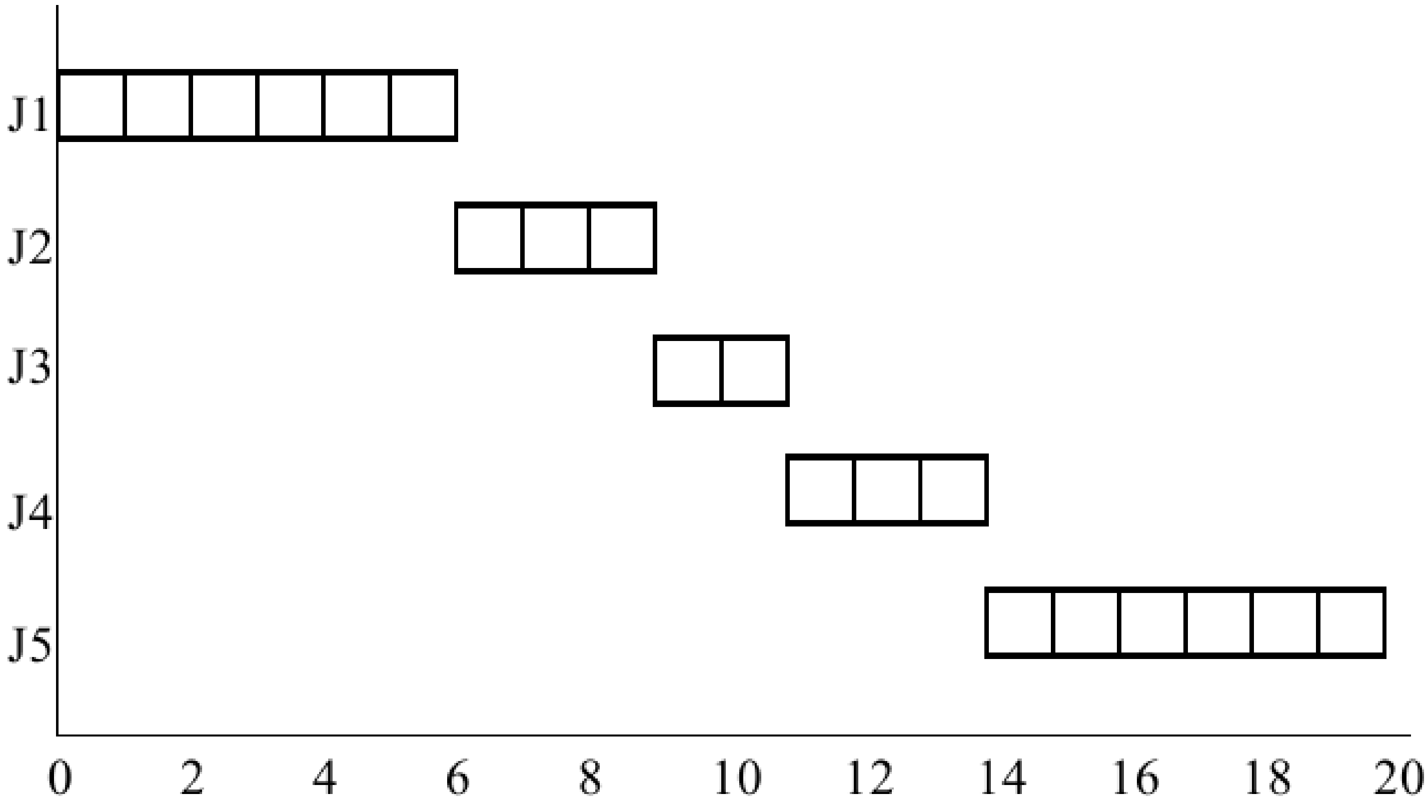


Figure 2

FCFS Scheduling

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

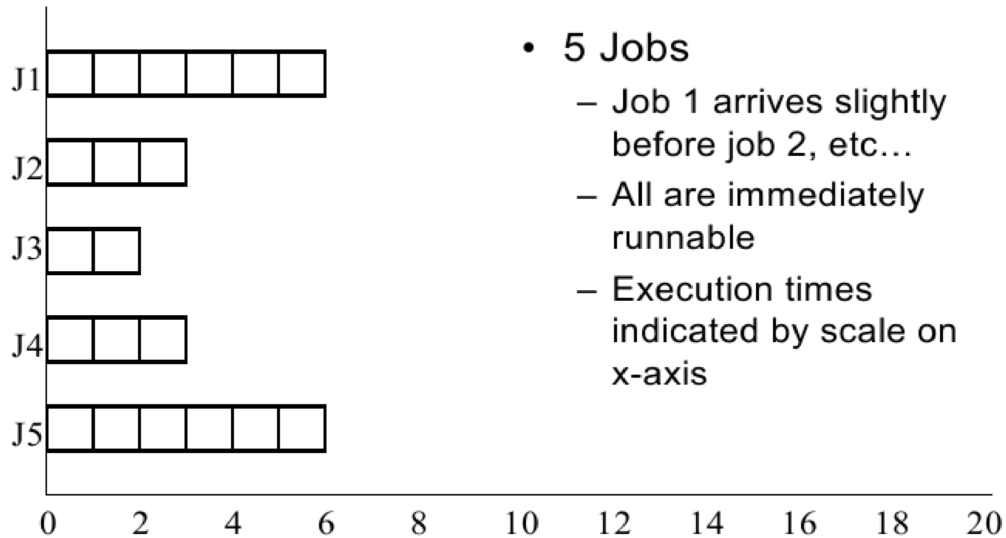


Figure 3: SJF – shortest job first

Shortest Job First

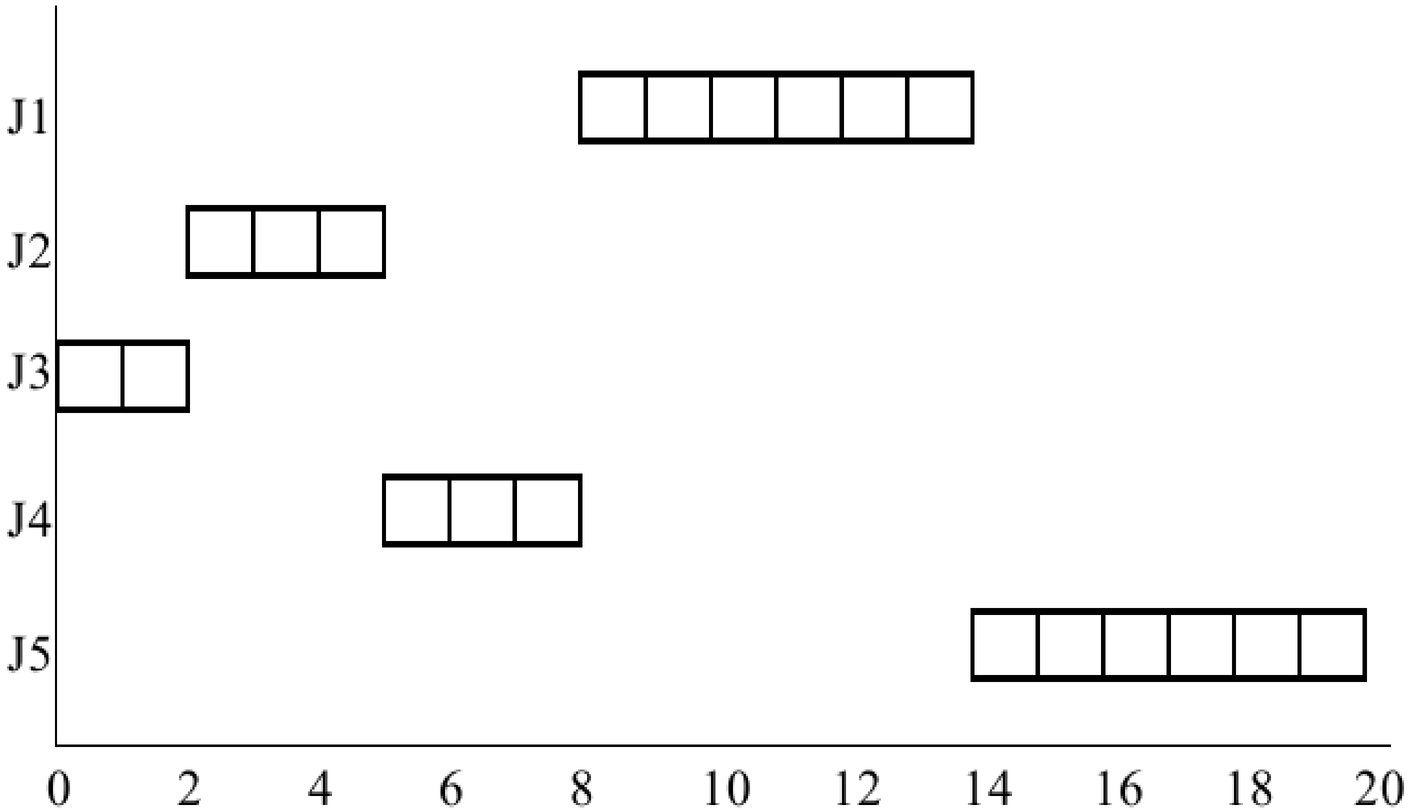


Figure 4

SJF

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 (4a + 3b + 2c + 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

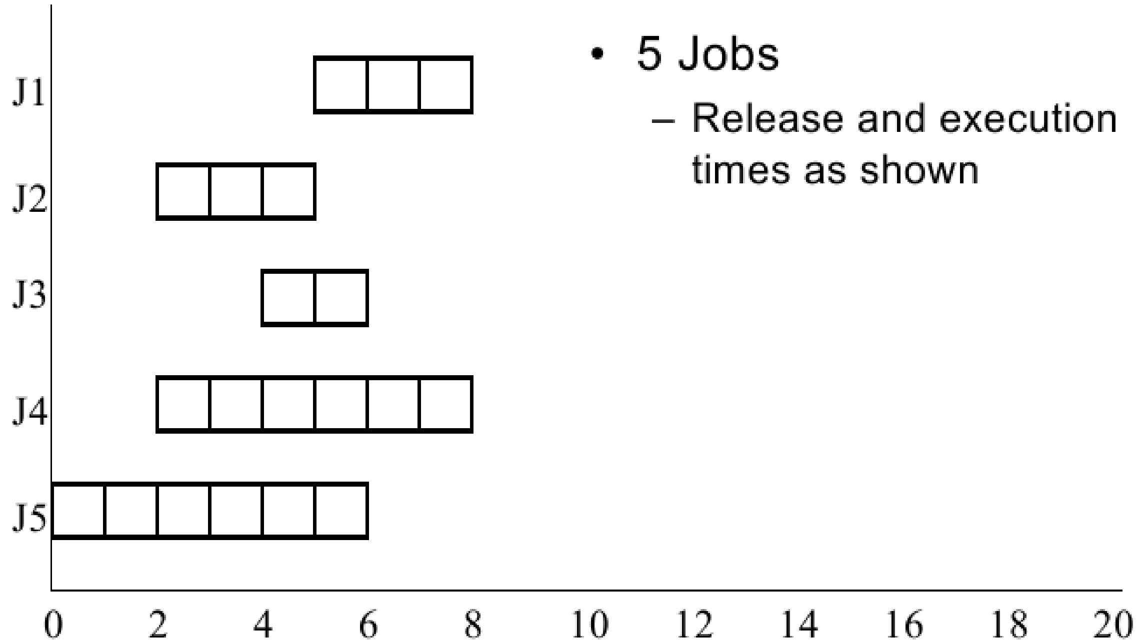


Figure 5

Example

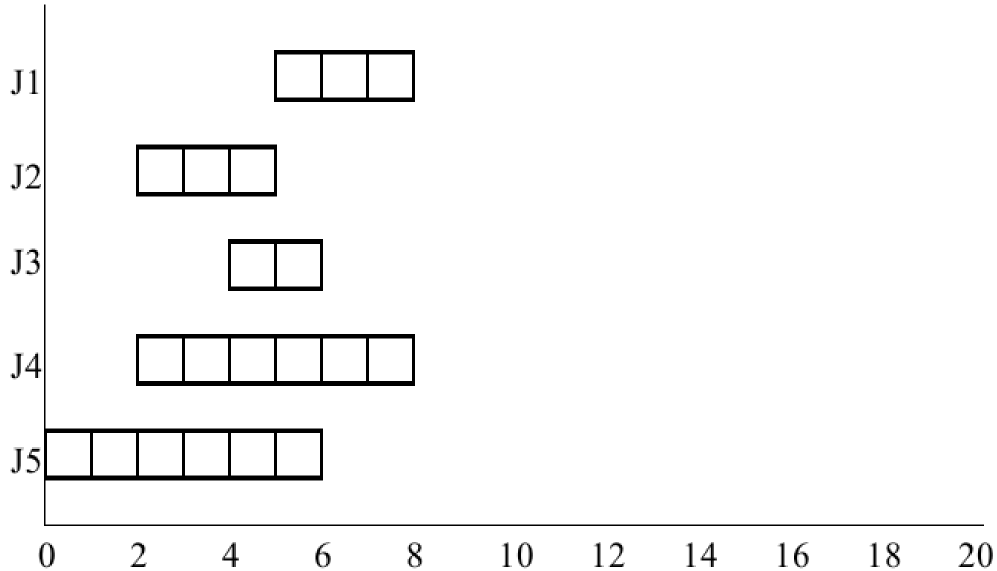


Figure 6

Example

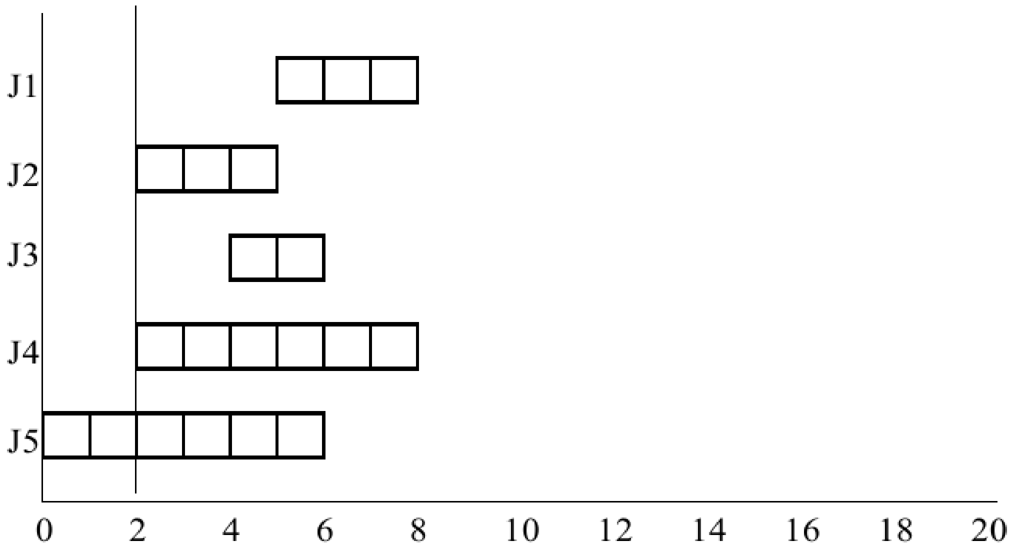


Figure 7

Example

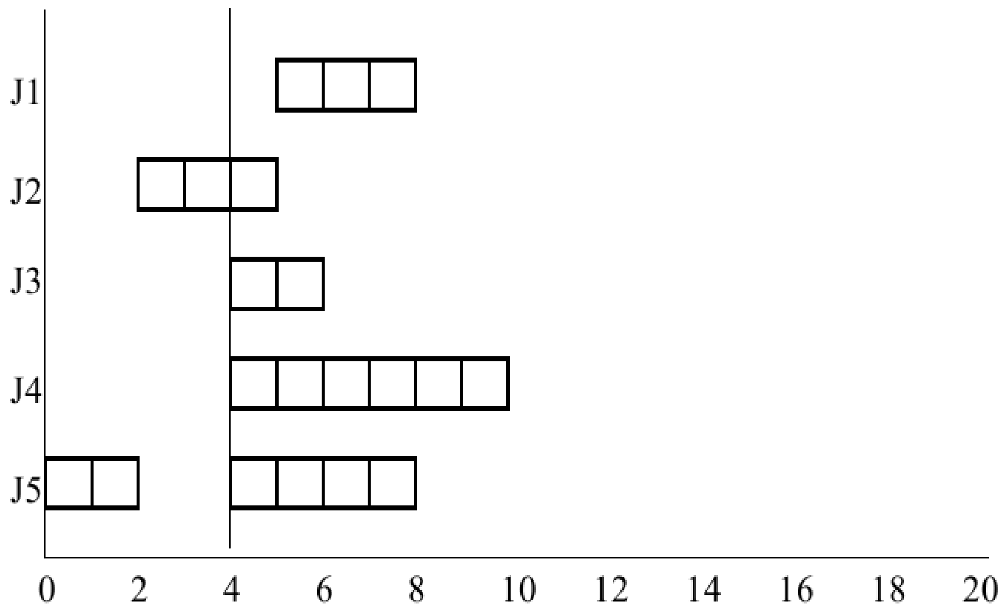


Figure 8

Shortest Remaining Time First

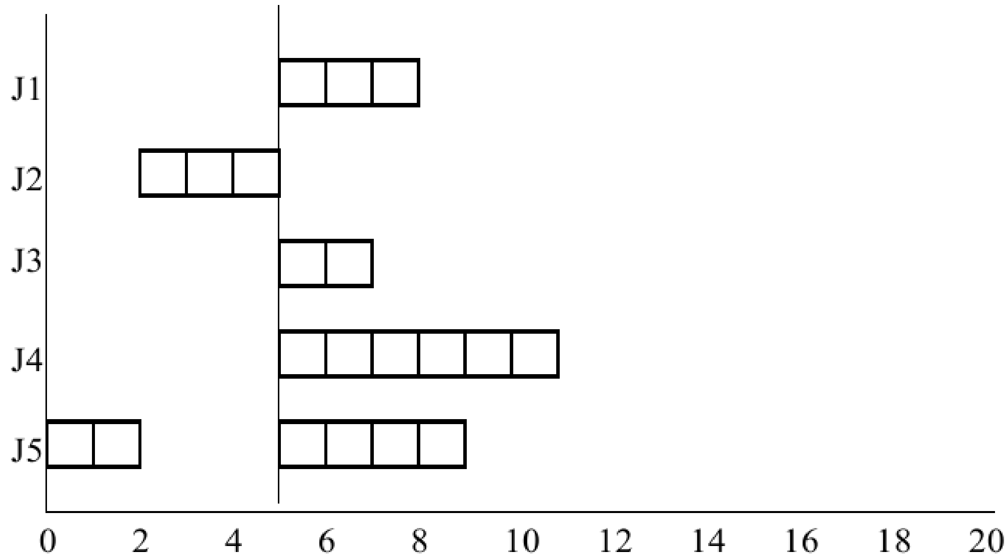


Figure 9

Shortest Remaining Time First

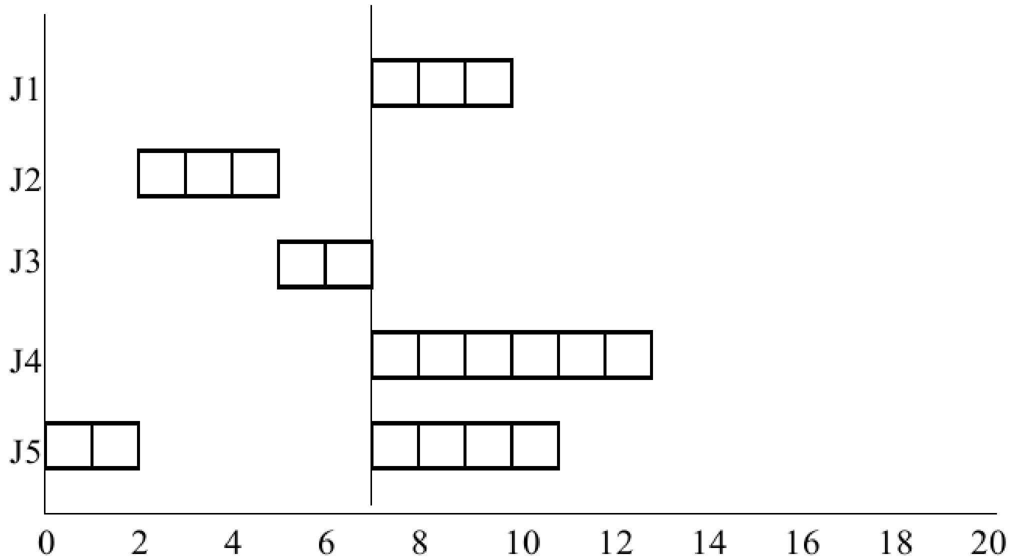


Figure 10

Shortest Remaining Time First

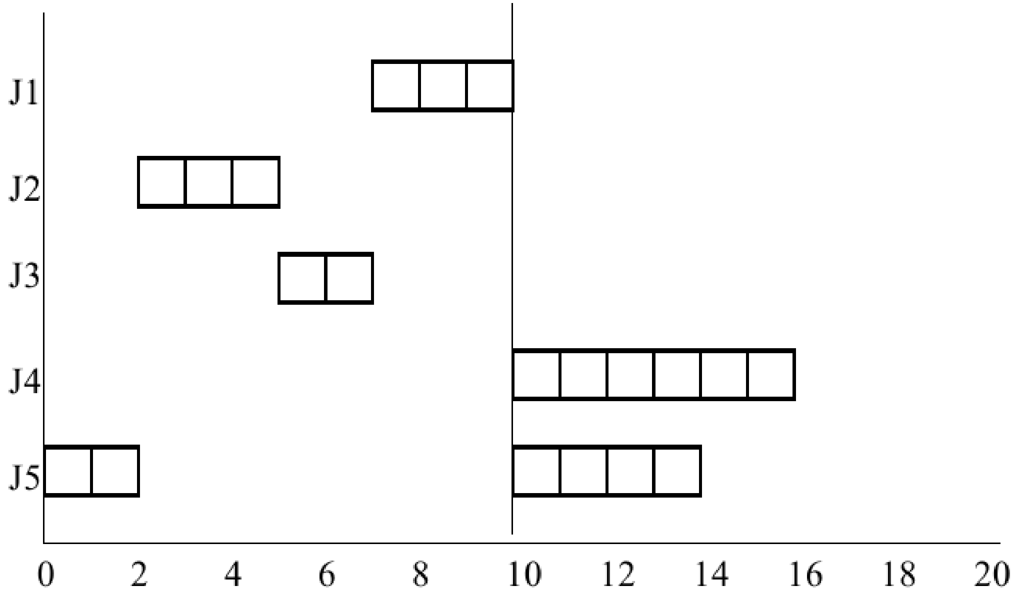


Figure 11

Shortest Remaining Time First

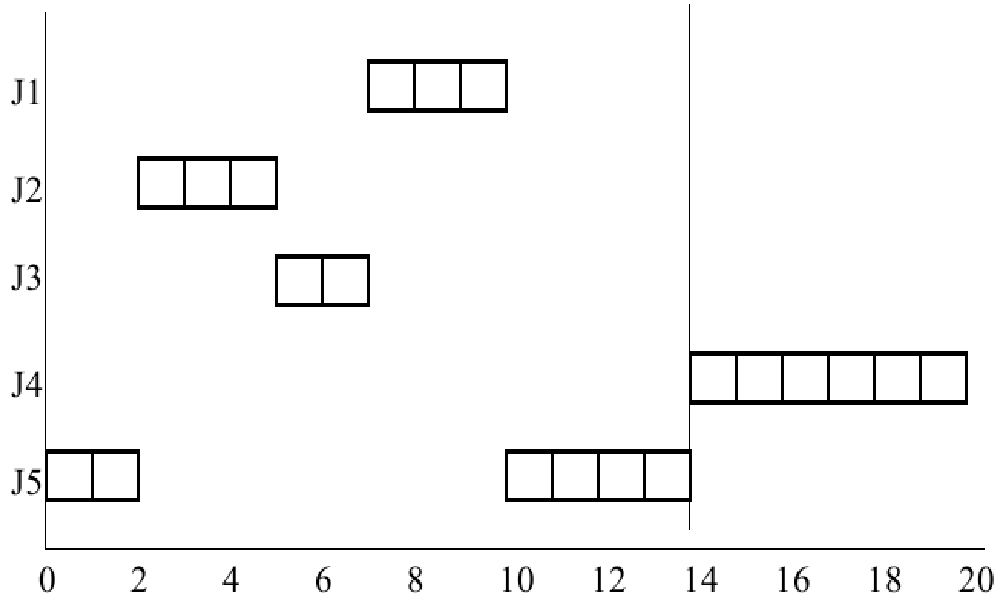


Figure 12

Scheduling in Batch Systems

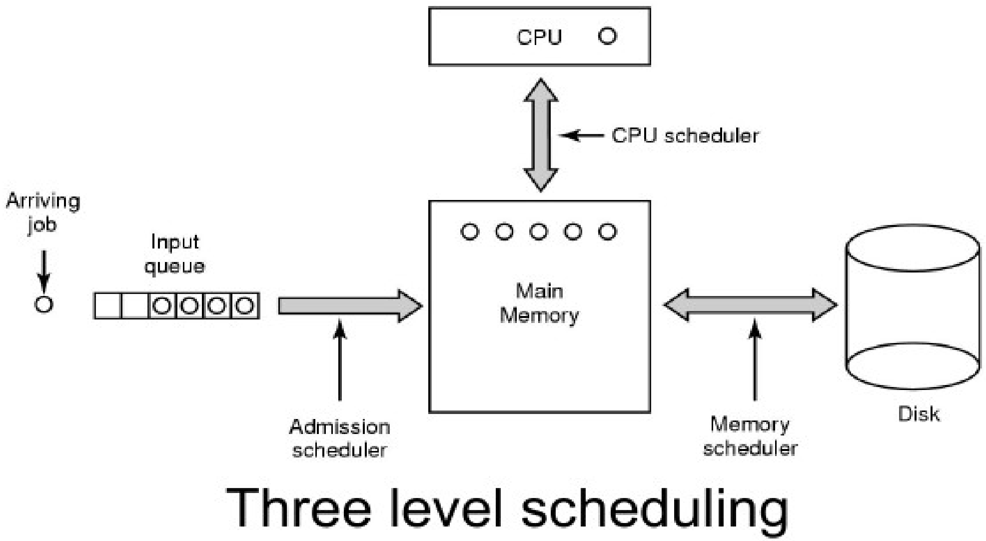


Figure 13

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 $\Rightarrow$ less CPU available per process

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

Interactive Scheduling

Two Level Scheduling

- Interactive systems commonly employ two-level scheduling
  - CPU scheduler and Memory Scheduler
    - Memory scheduler was covered in VM
  - We will focus on CPU scheduling

Round Robin Scheduling

- Each process is given a timeslice to run in
- When the timeslice expires, the next process preempts the current process, and runs for its timeslice, and so on
    - The preempted process is placed at the end of the queue
- Implemented with
    - A ready queue
    - A regular timer interrupt

Our Earlier Example

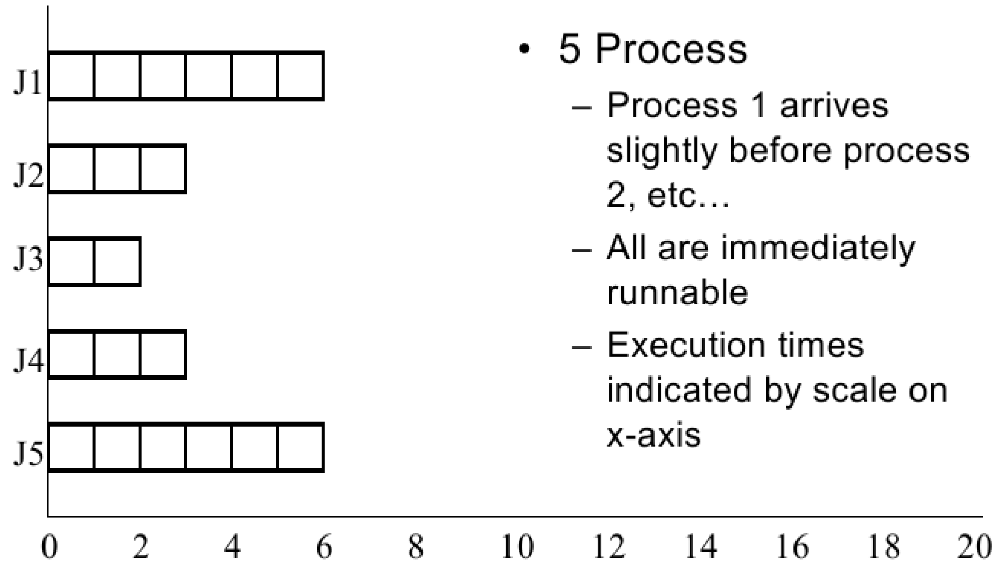


Figure 14

Round Robin Schedule

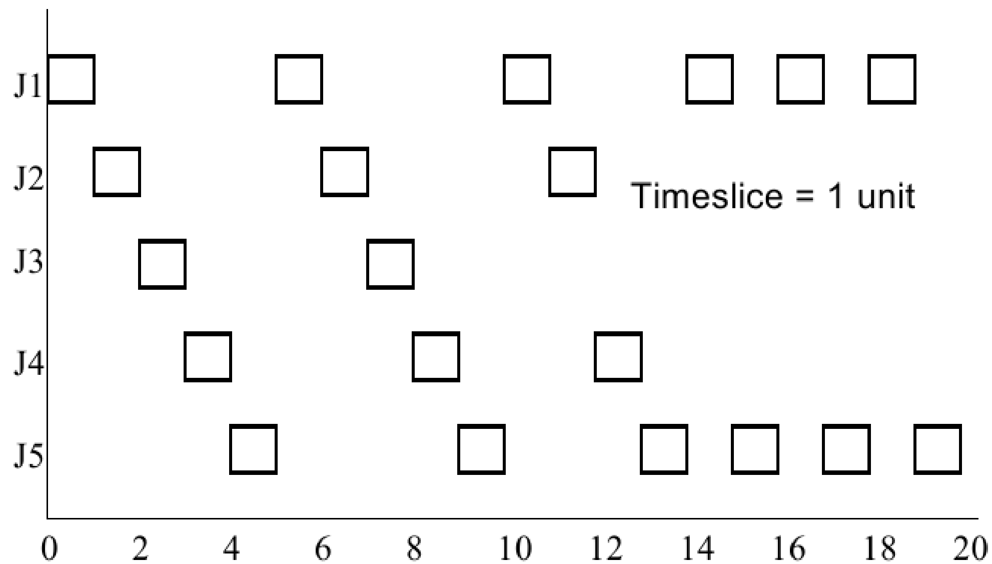


Figure 15

Round Robin Schedule

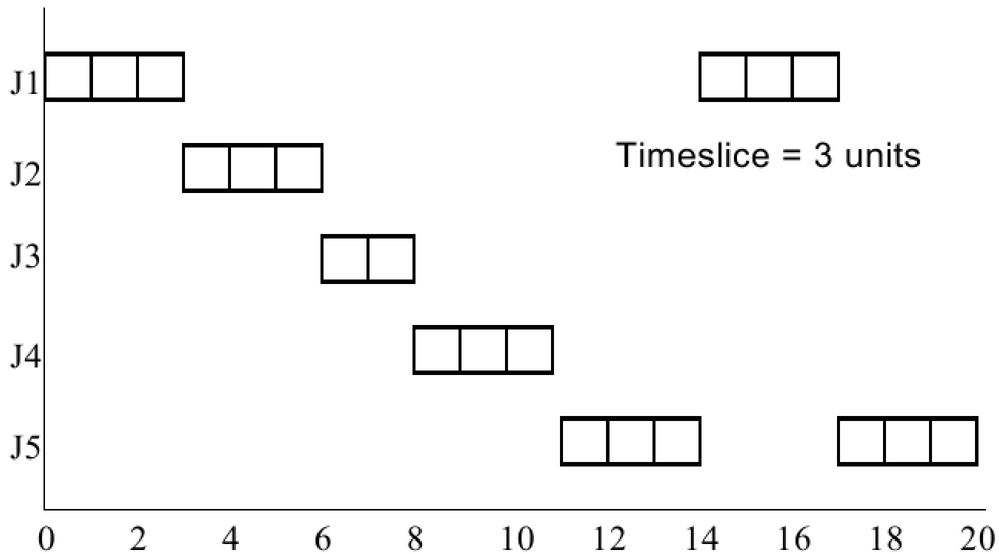


Figure 16

Round Robin Scheduling

Pros

- Fair, easy to implement

Con

- Assumes everybody is equal

Issue: What should the timeslice be?

- Too short
  - Waste a lot of time switching between processes
  - Example: timeslice of 4ms with 1 ms context switch = 20% round overhead
- Too long
  - System is not responsive
  - Example: timeslice of 100ms
    - If 10 people hit "enter" key simultaneously, the last guy to run will only see progress after 1 second.
  - Degenerates into FCFS if timeslice longer than burst length

Priorities

- Each Process (or thread) is associated with a priority
- Provides basic mechanism to influence a scheduler decision:
  - Scheduler will always chooses a thread of higher priority over lower priority
- Priorities can be defined internally or externally
  - Internal: e.g. I/O bound or CPU bound
  - External: e.g. based on importance to the user

Example

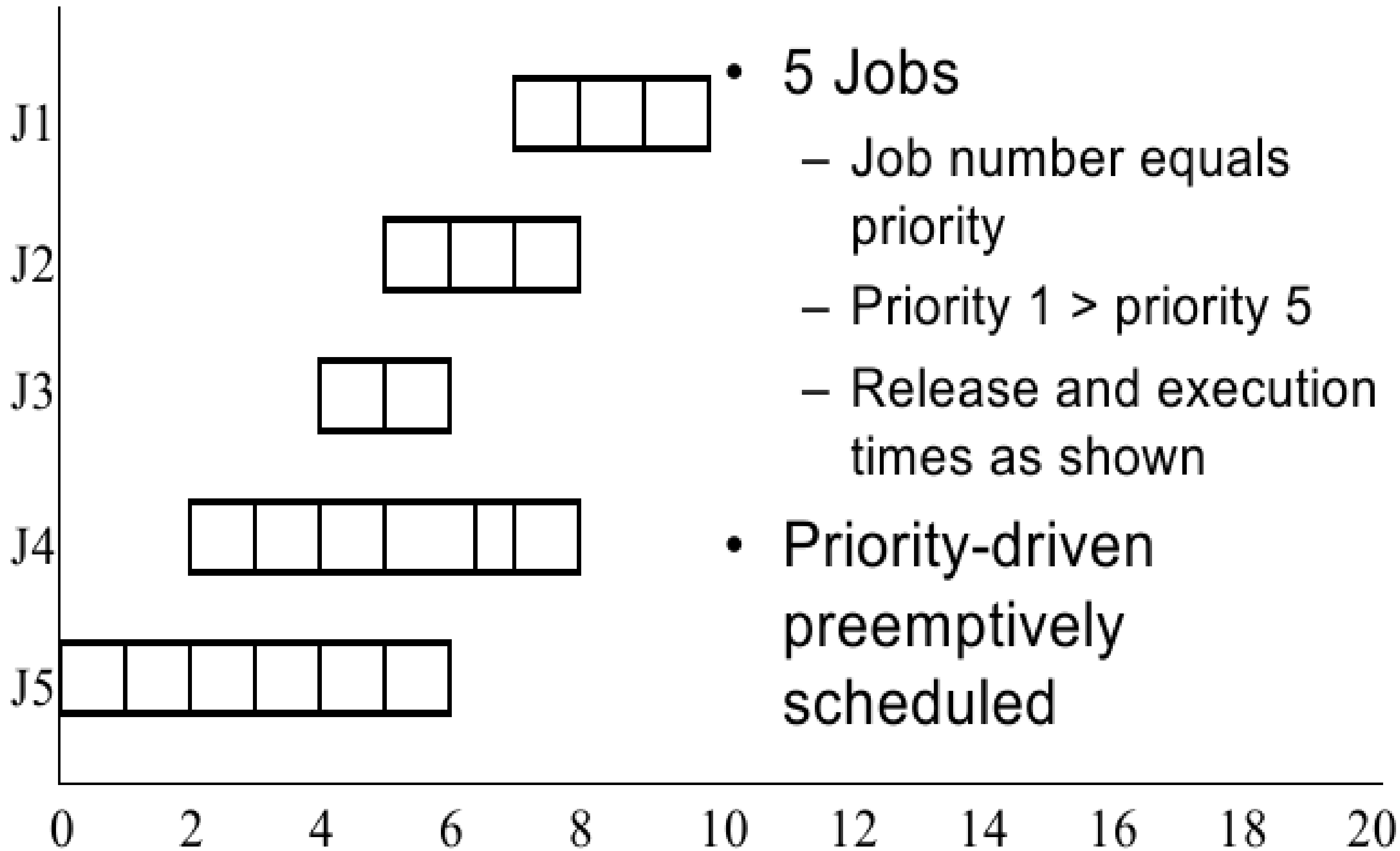


Figure 17

Example

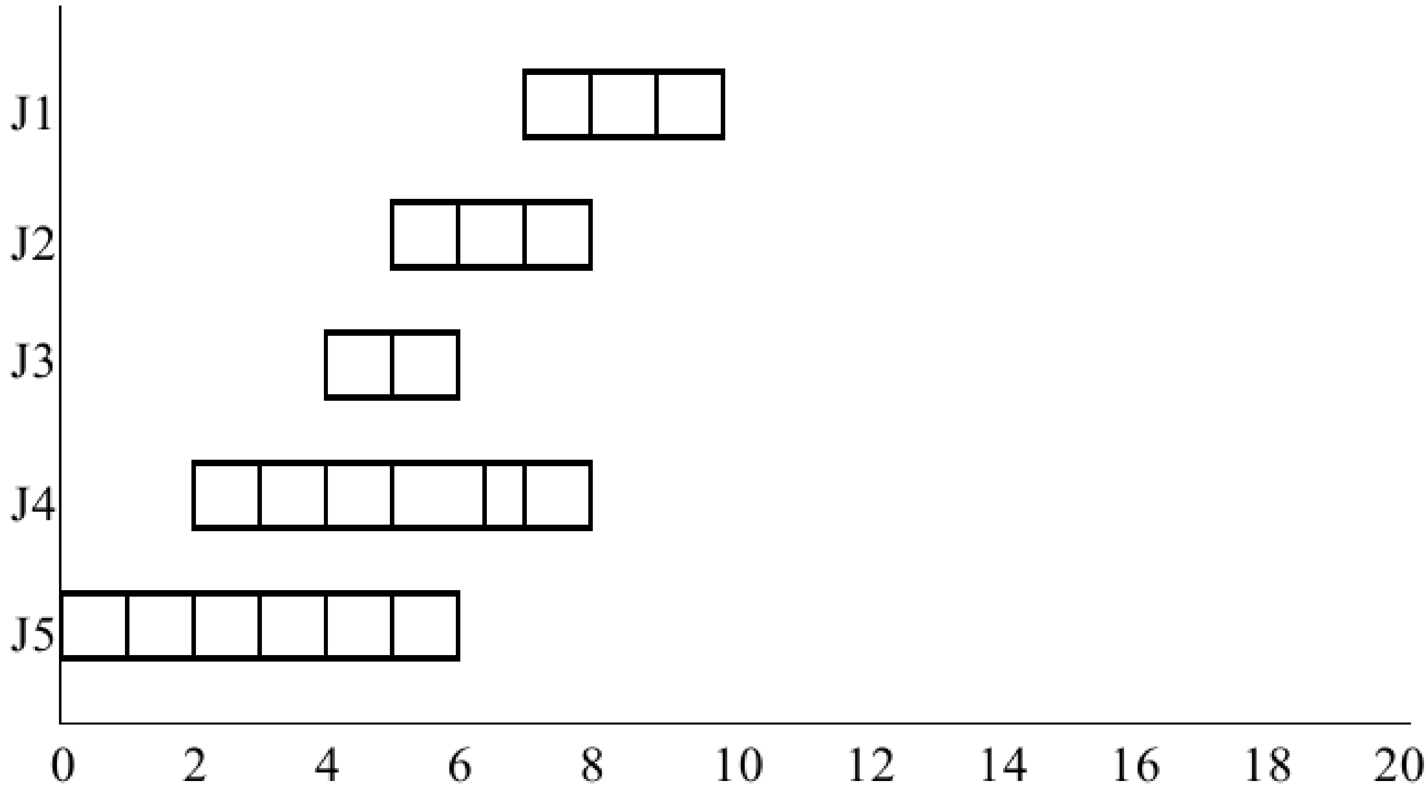


Figure 18

Example

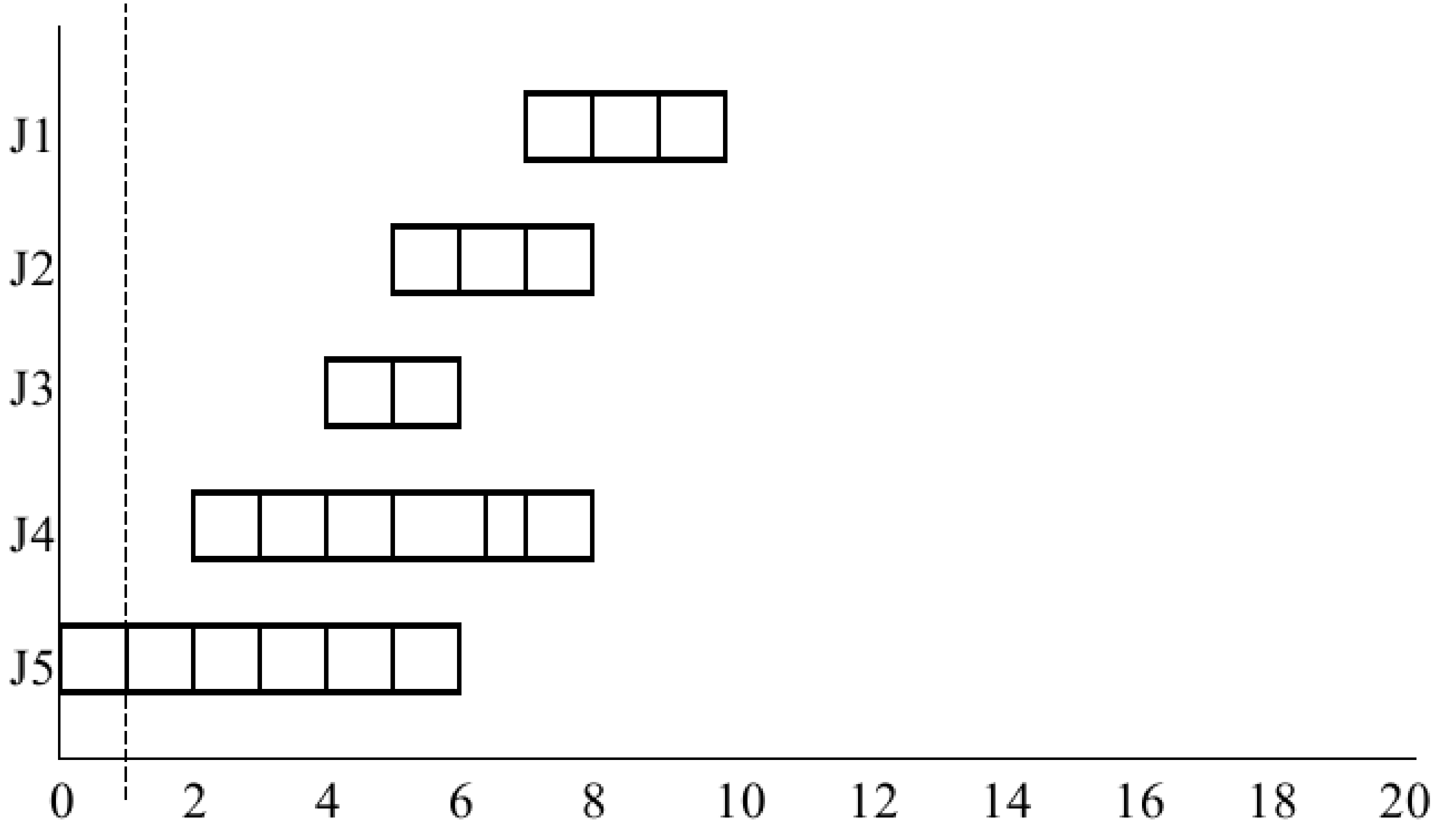


Figure 19

Example

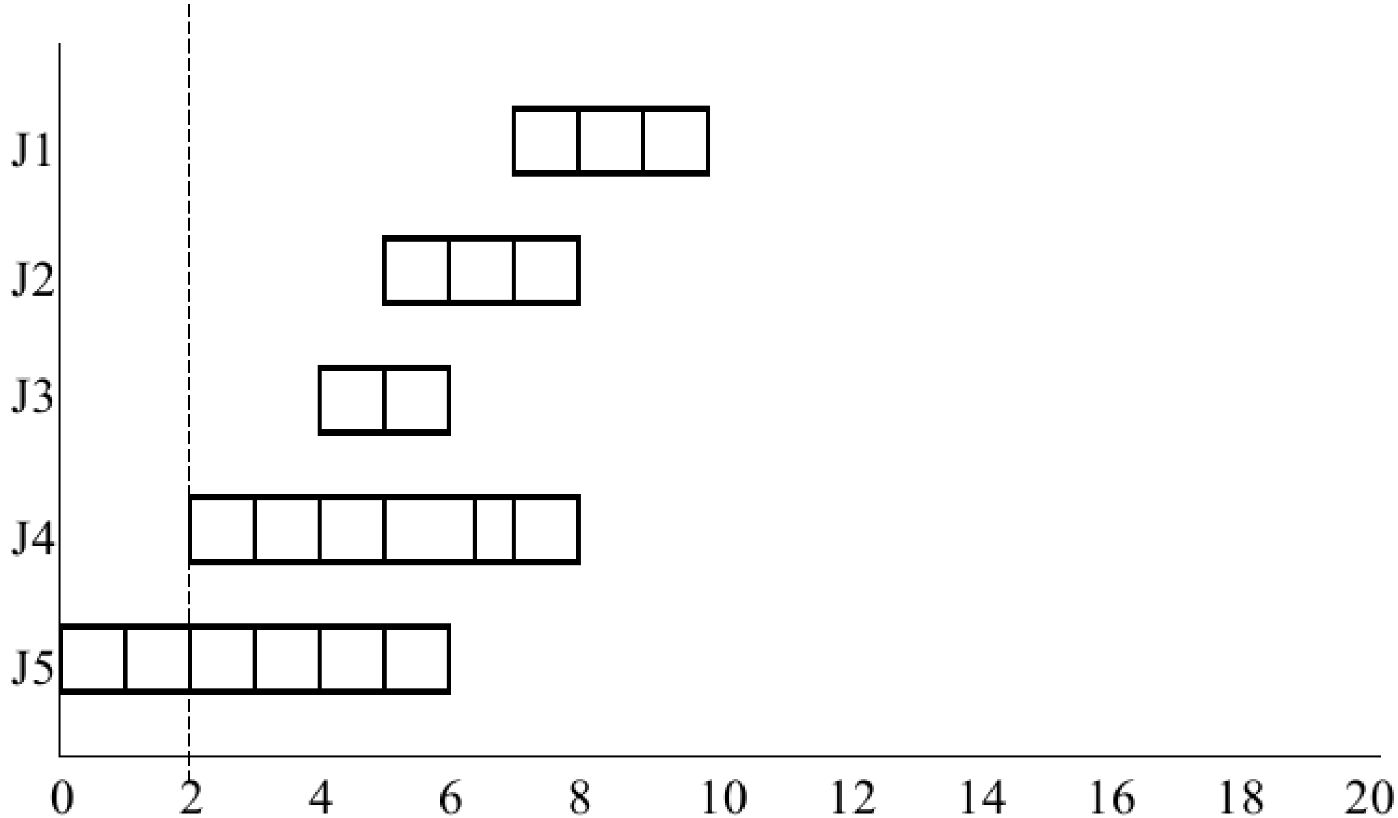


Figure 20

Example

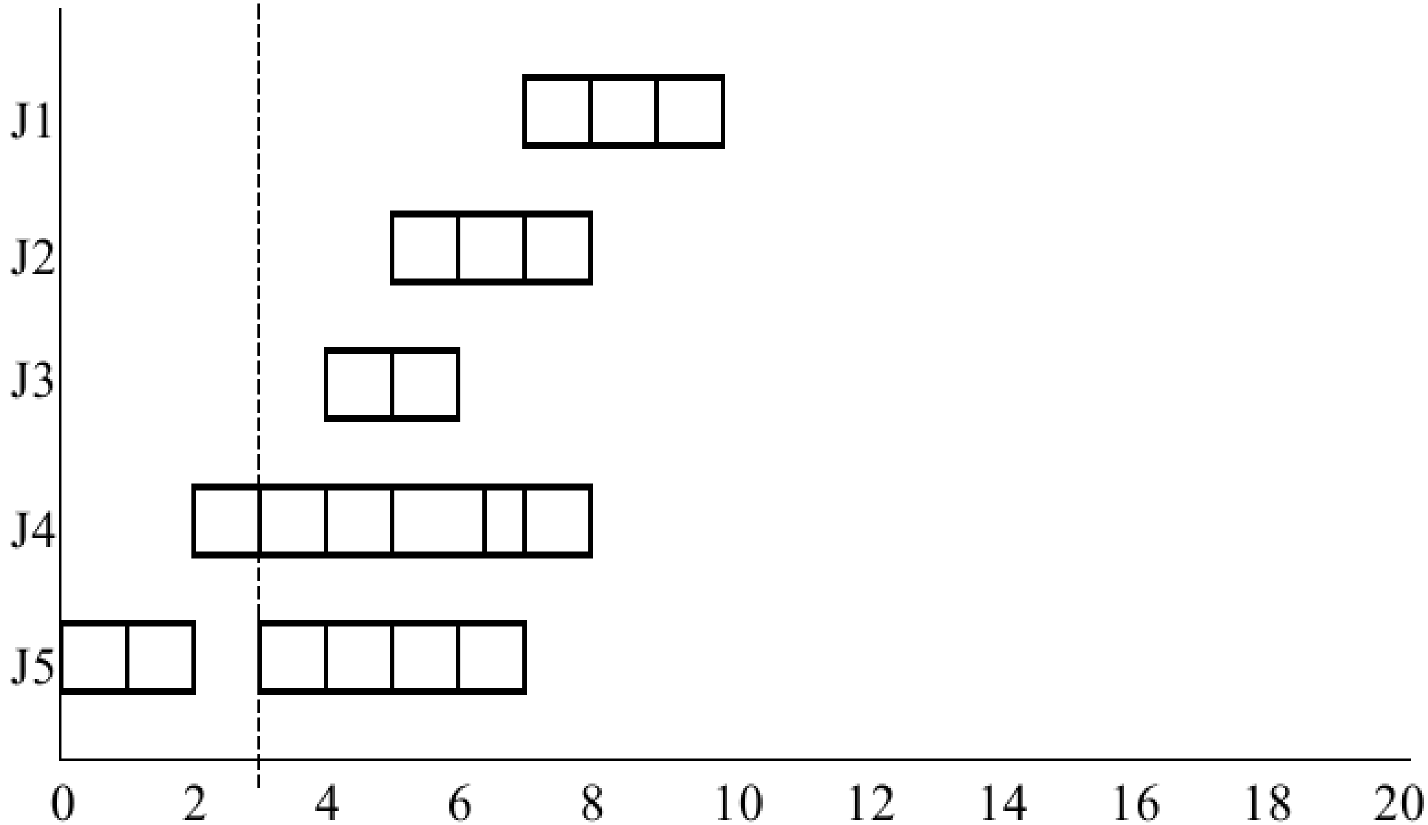


Figure 21

Example

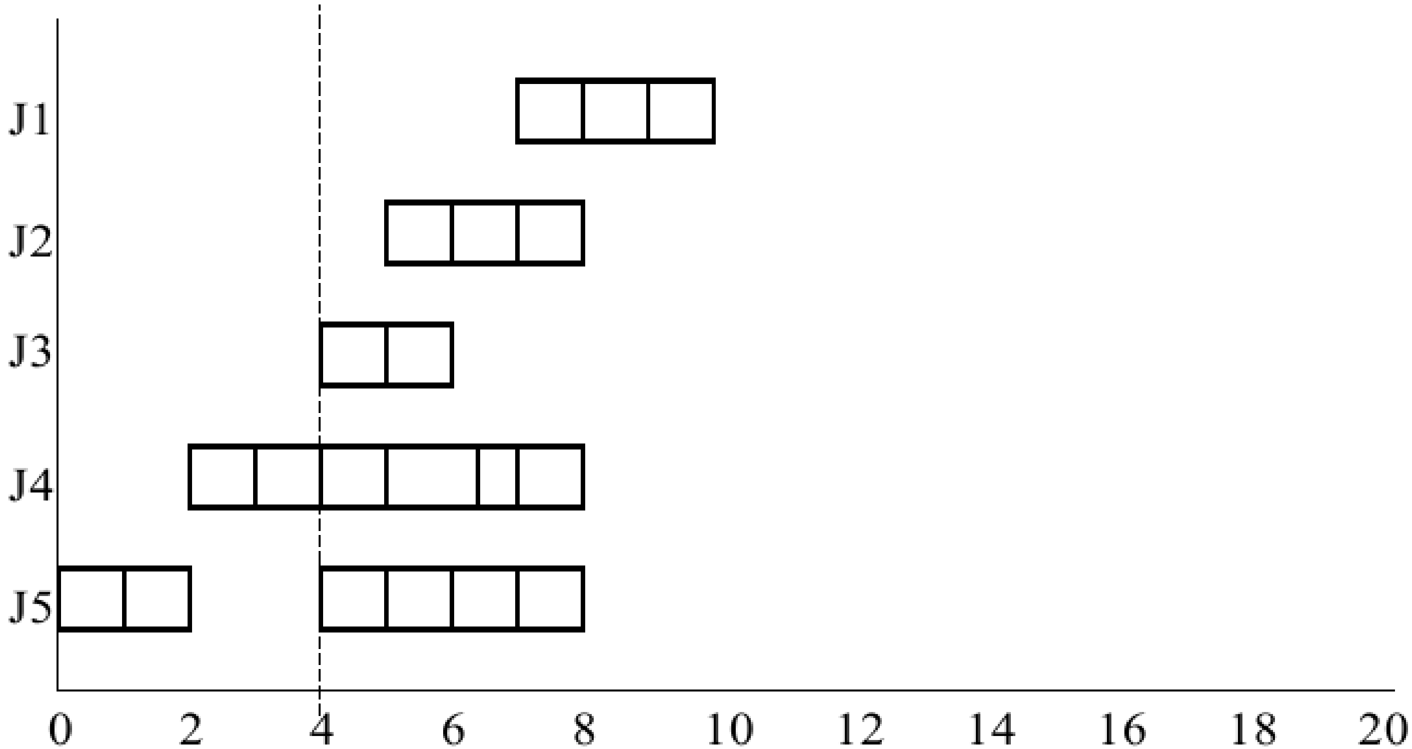


Figure 22

Example

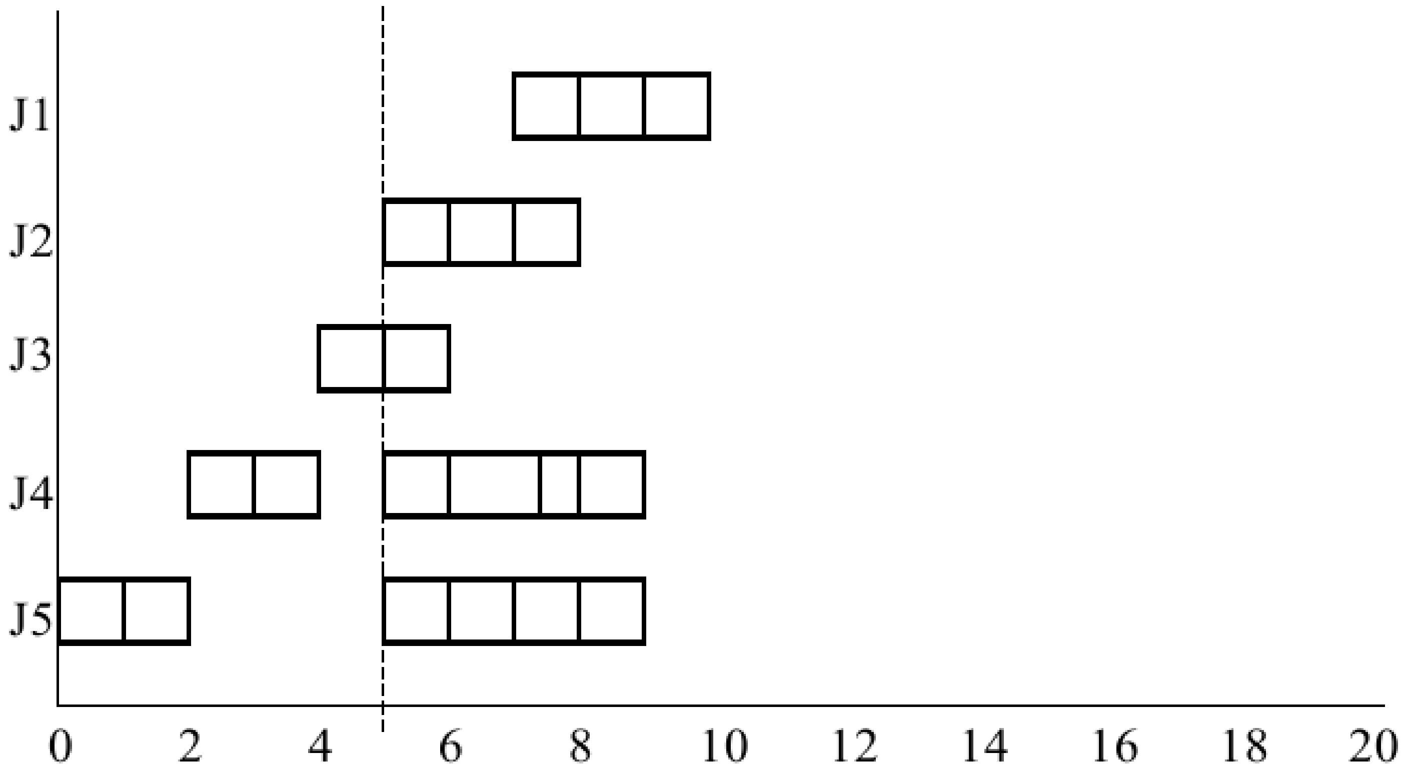


Figure 23

Example

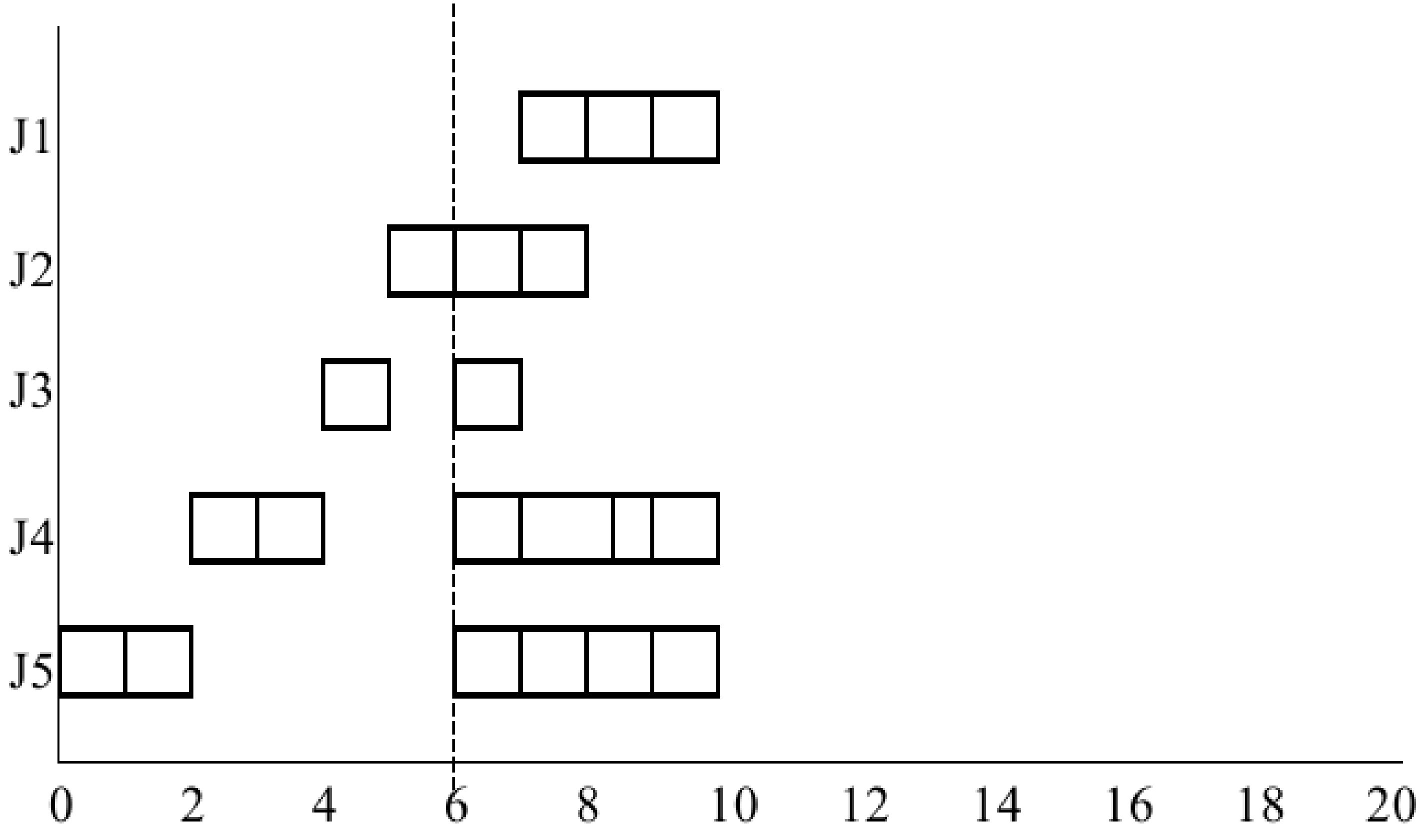


Figure 24

Example

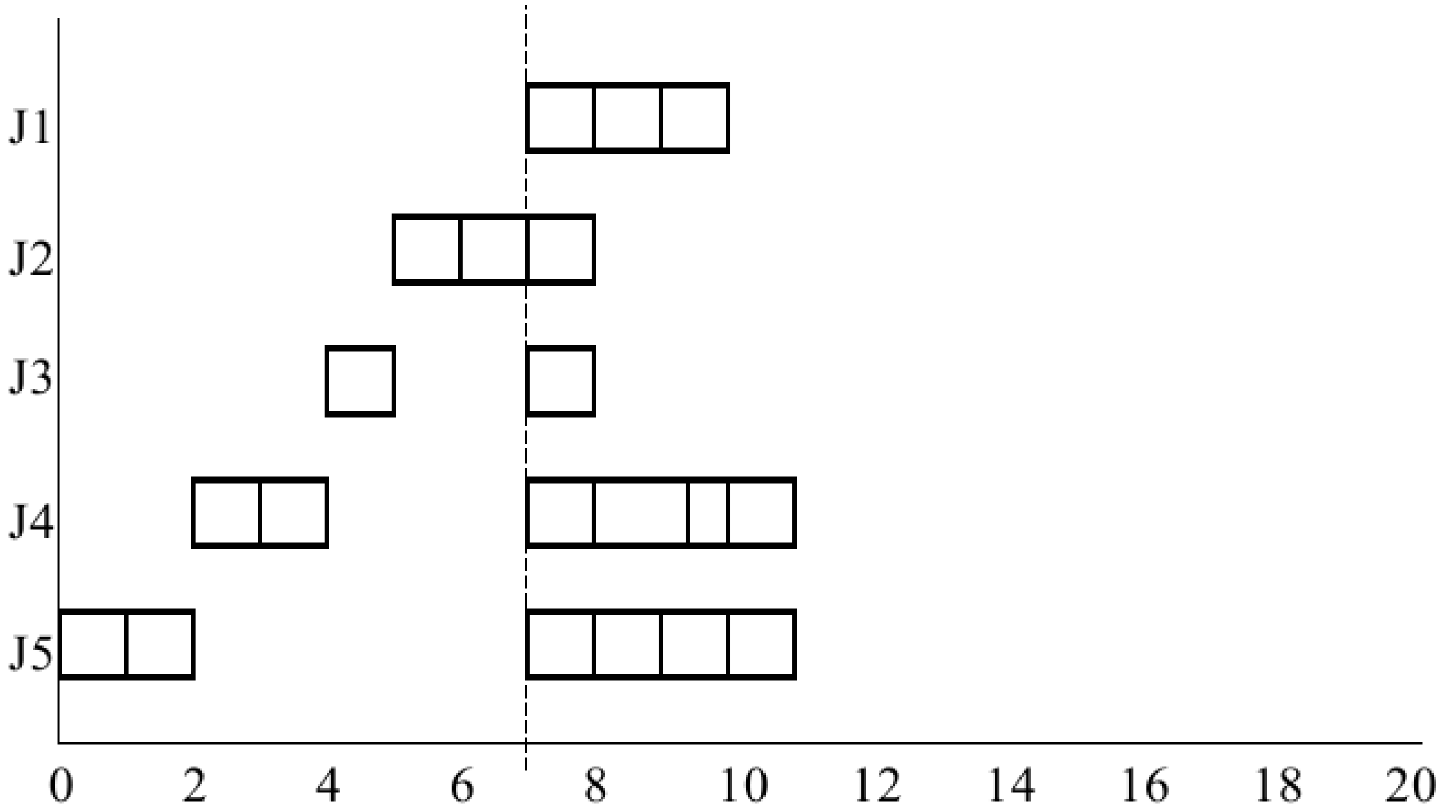


Figure 25

Example

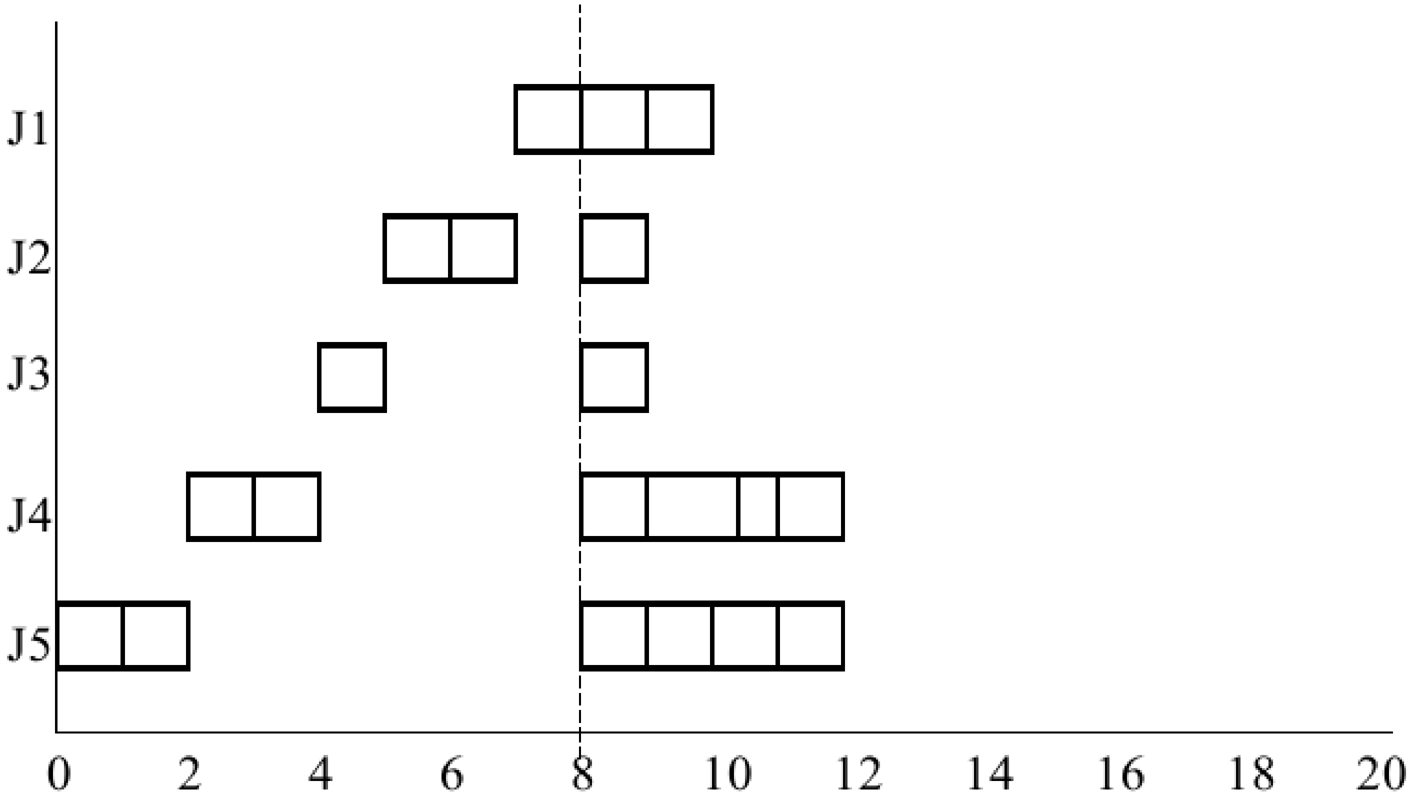


Figure 26

Example

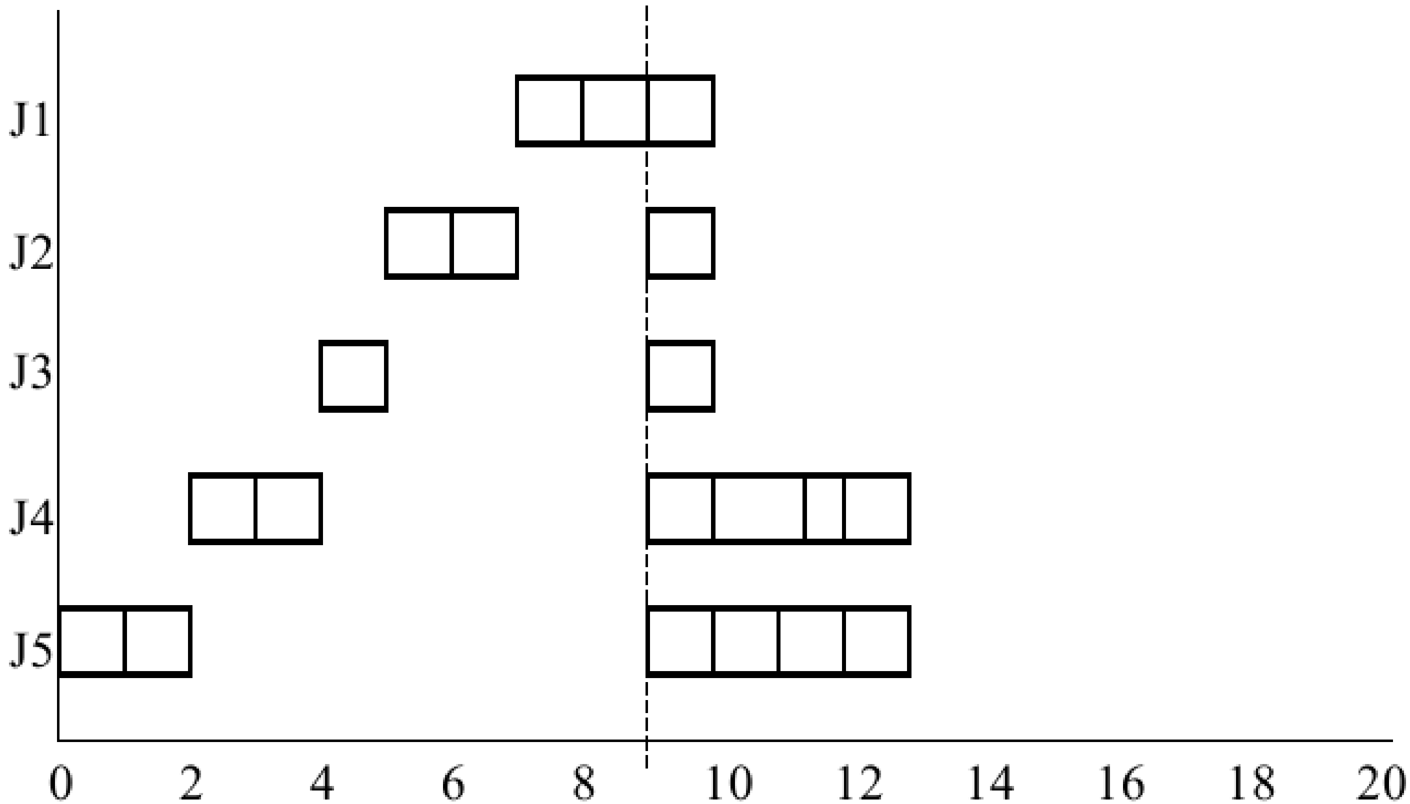


Figure 27

Example

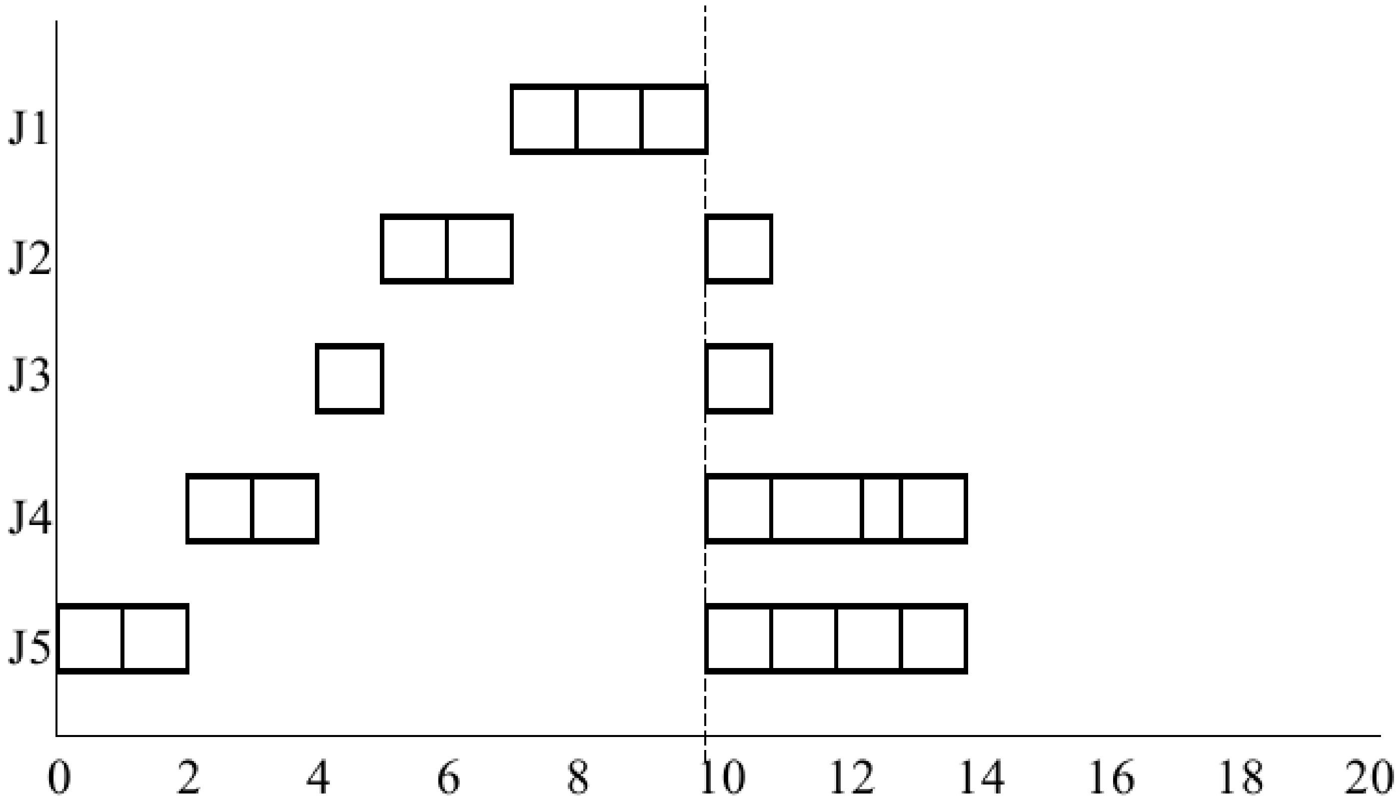


Figure 28

Example

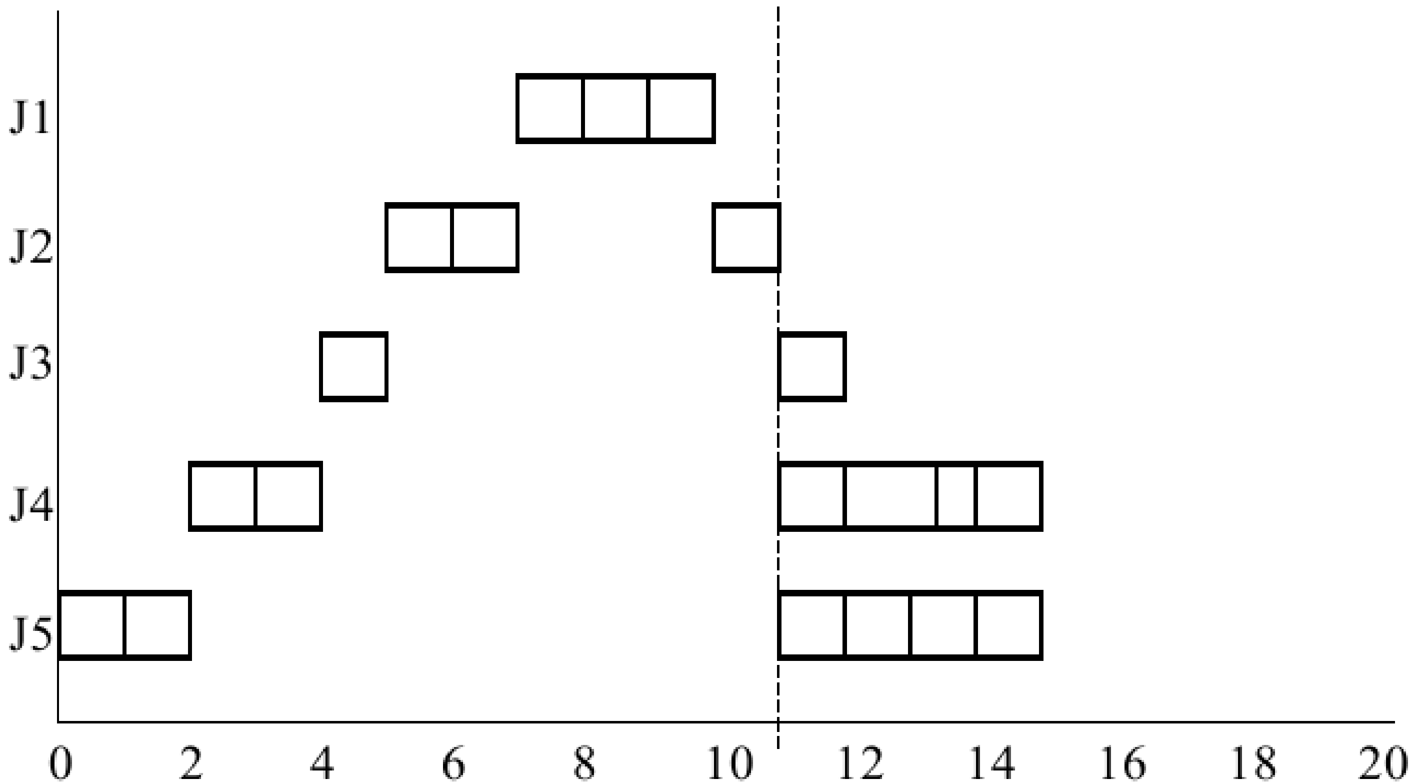


Figure 29

Example

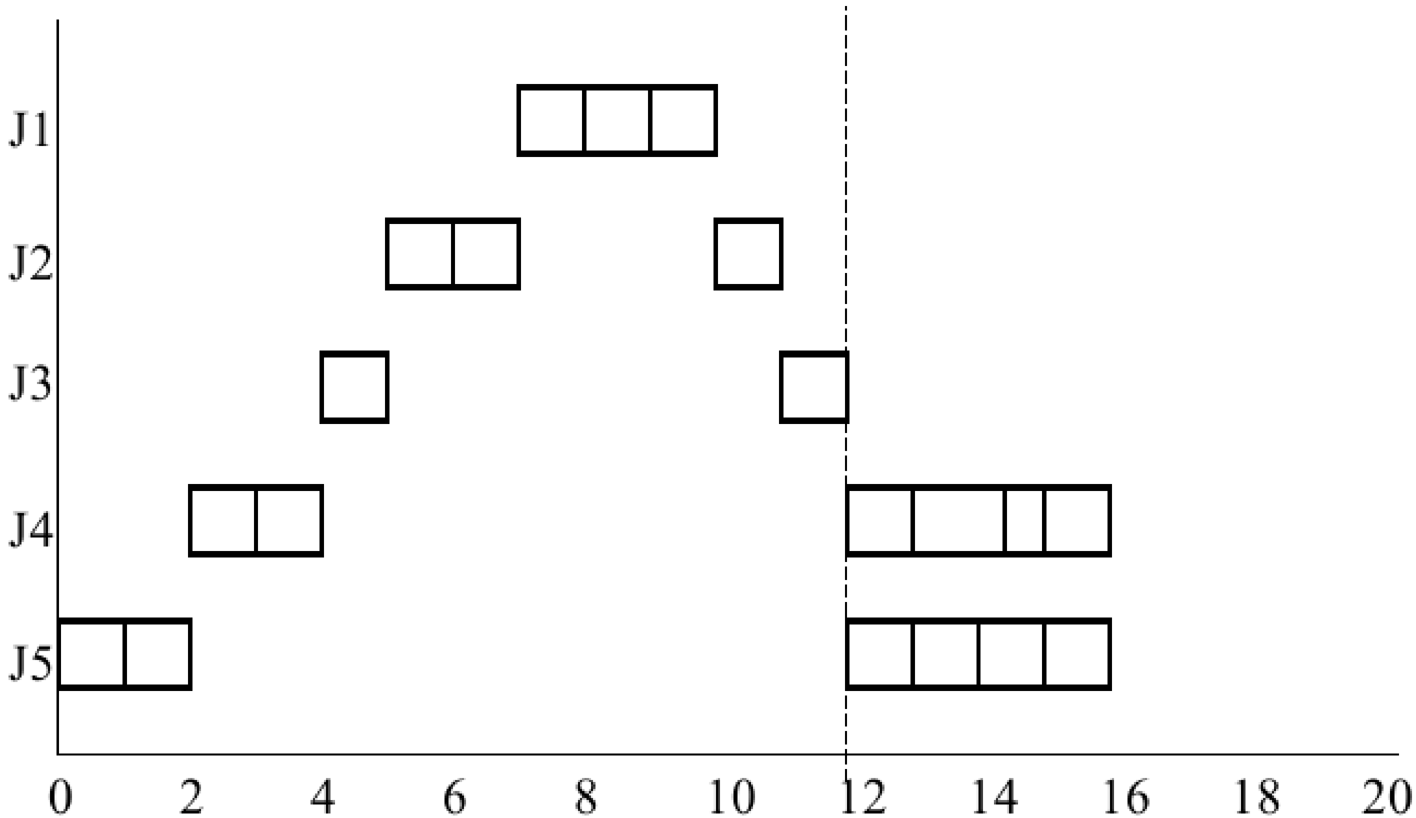


Figure 30

Example

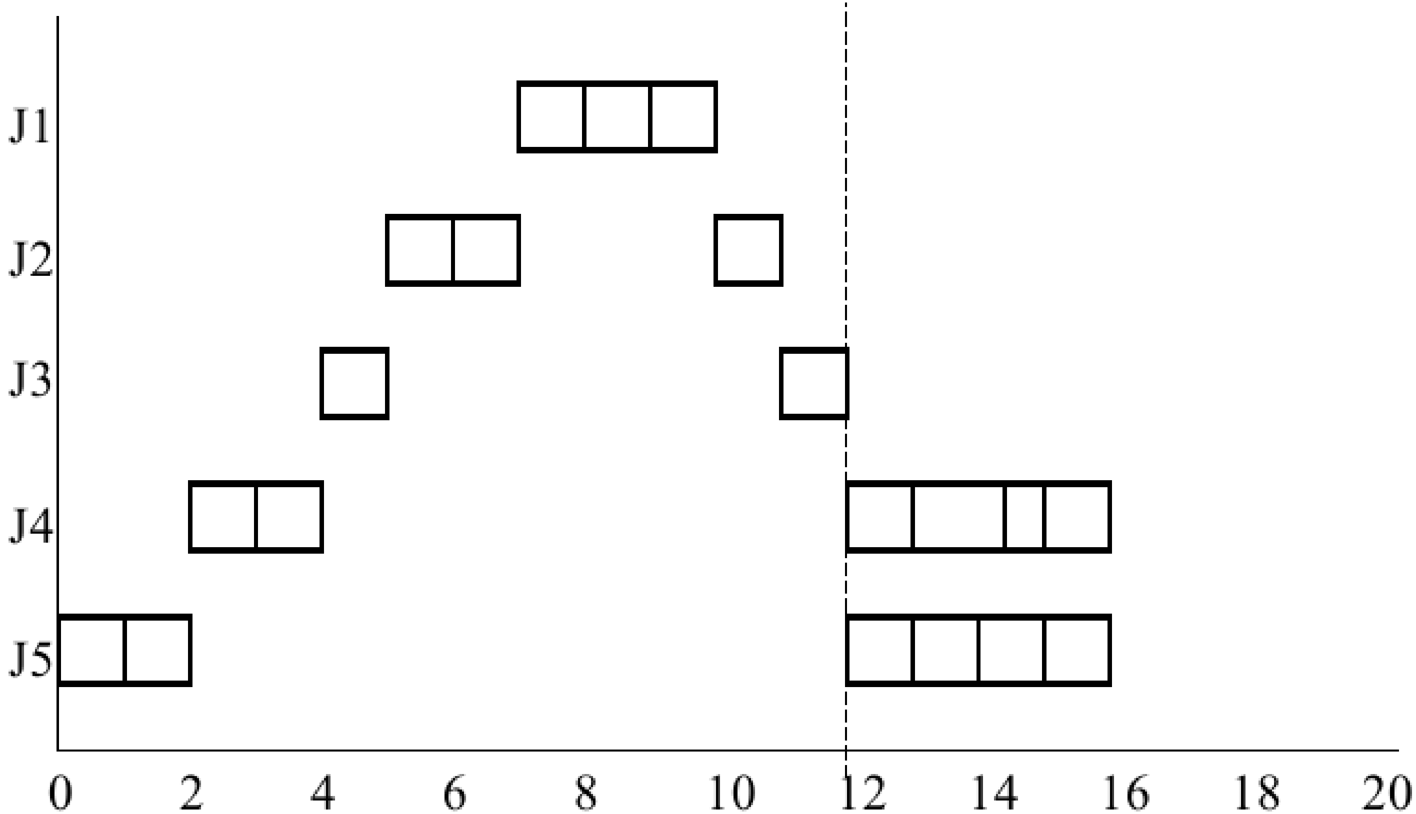


Figure 31

Example

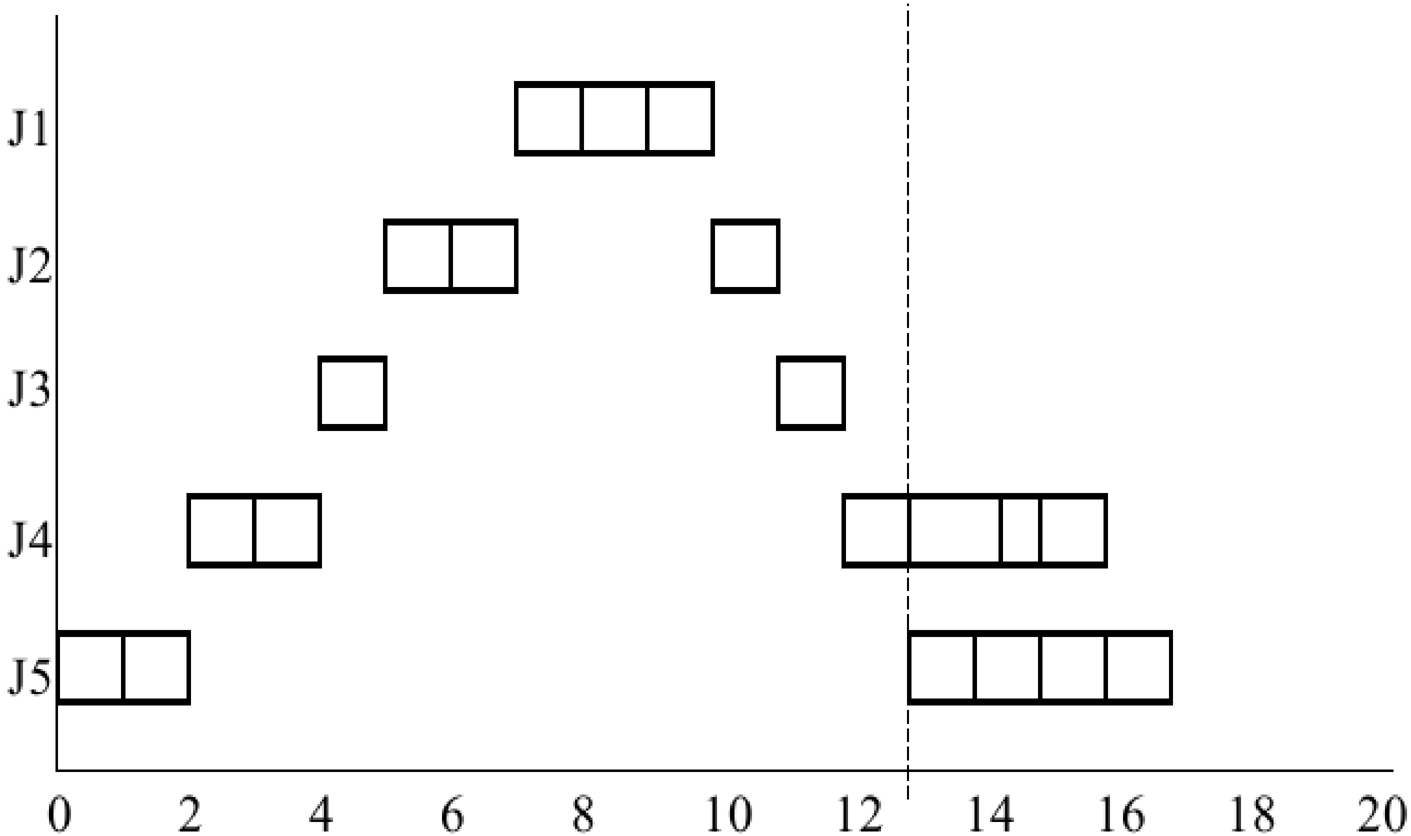


Figure 32

Example

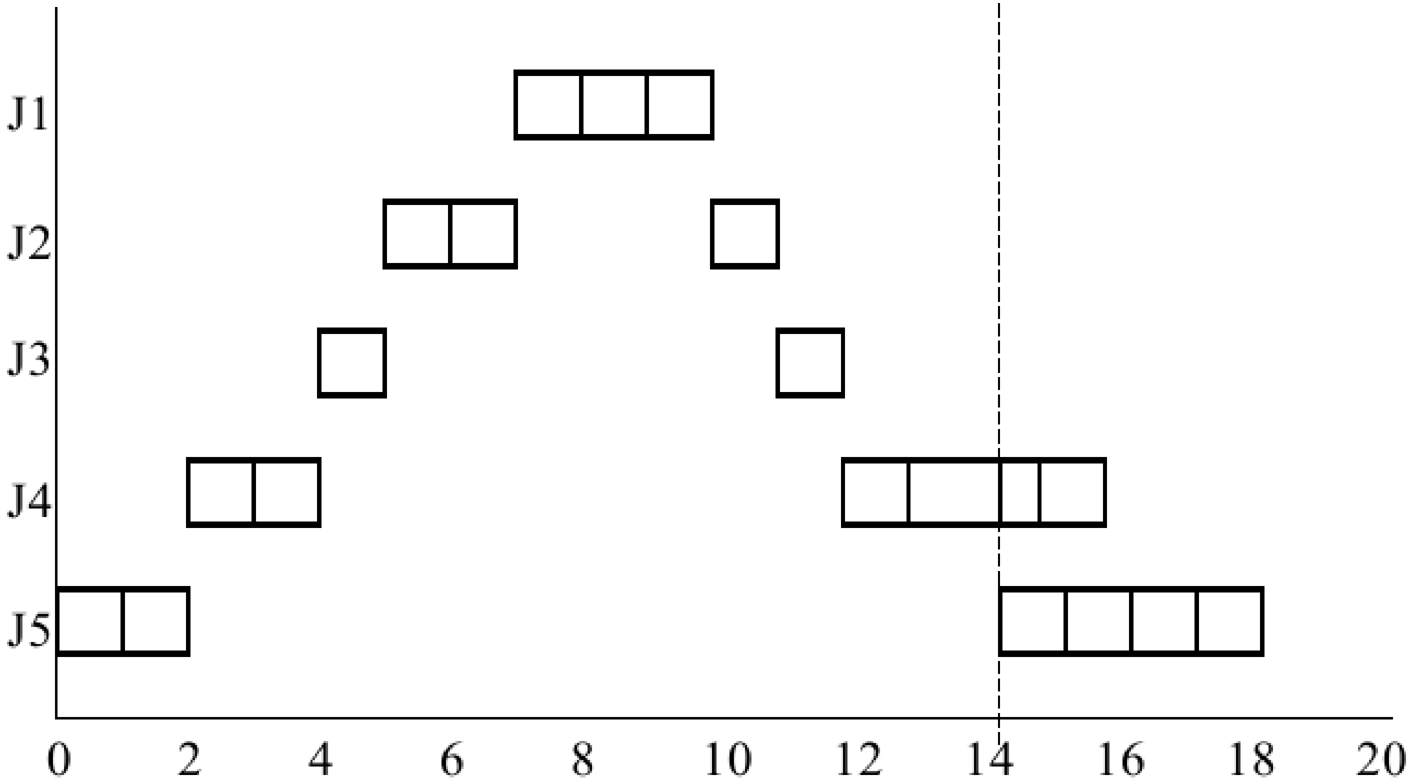


Figure 33

Example

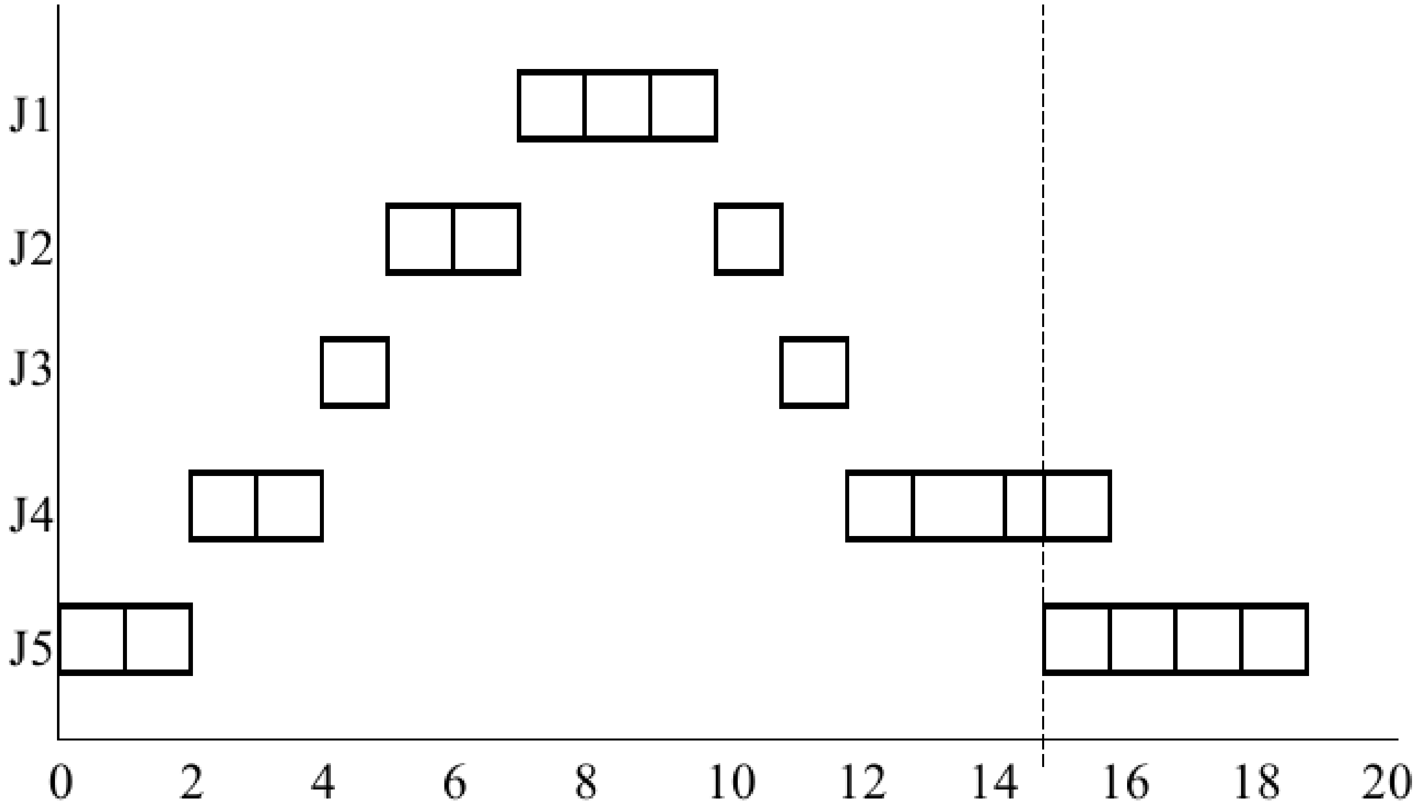


Figure 34

Example

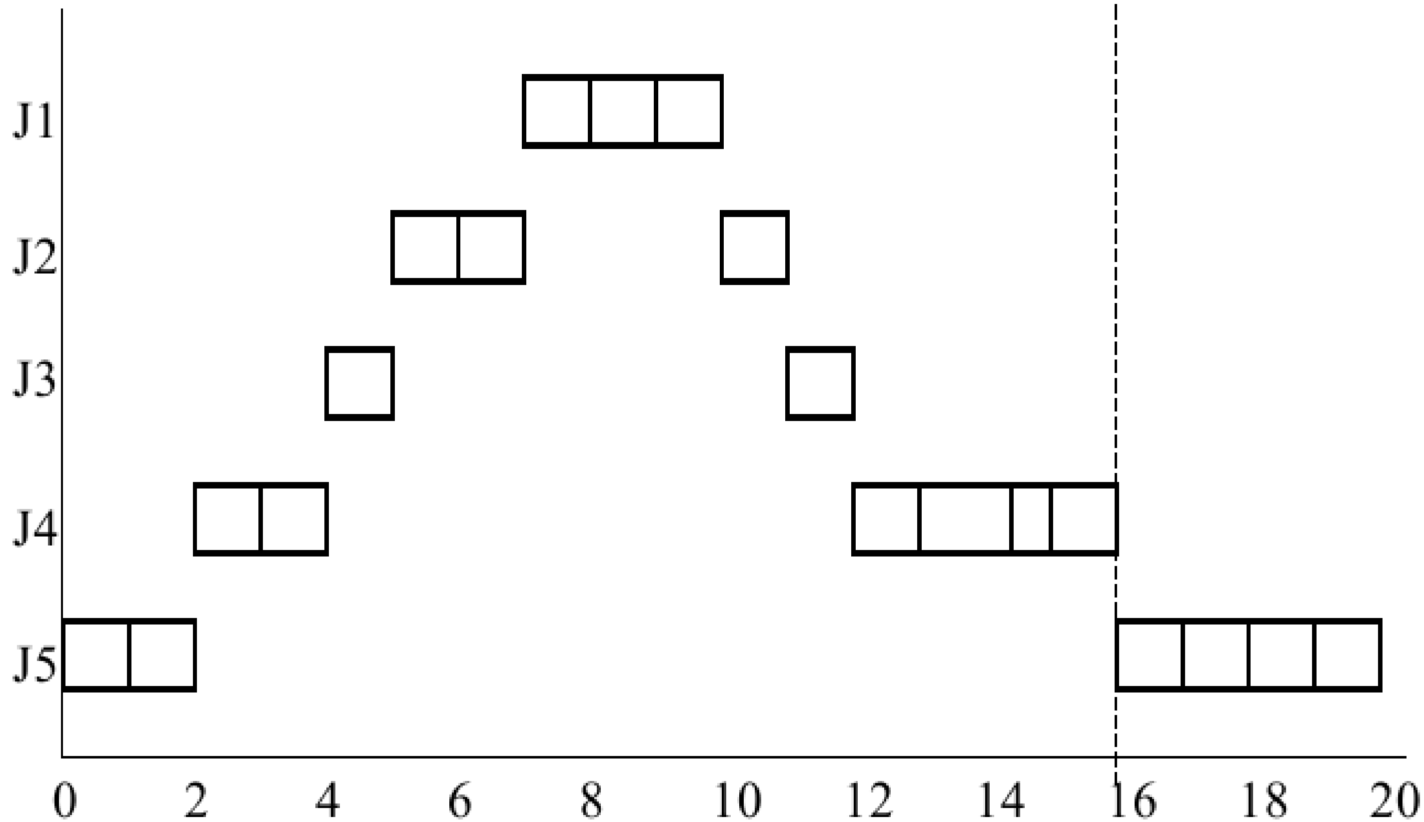


Figure 35

Example

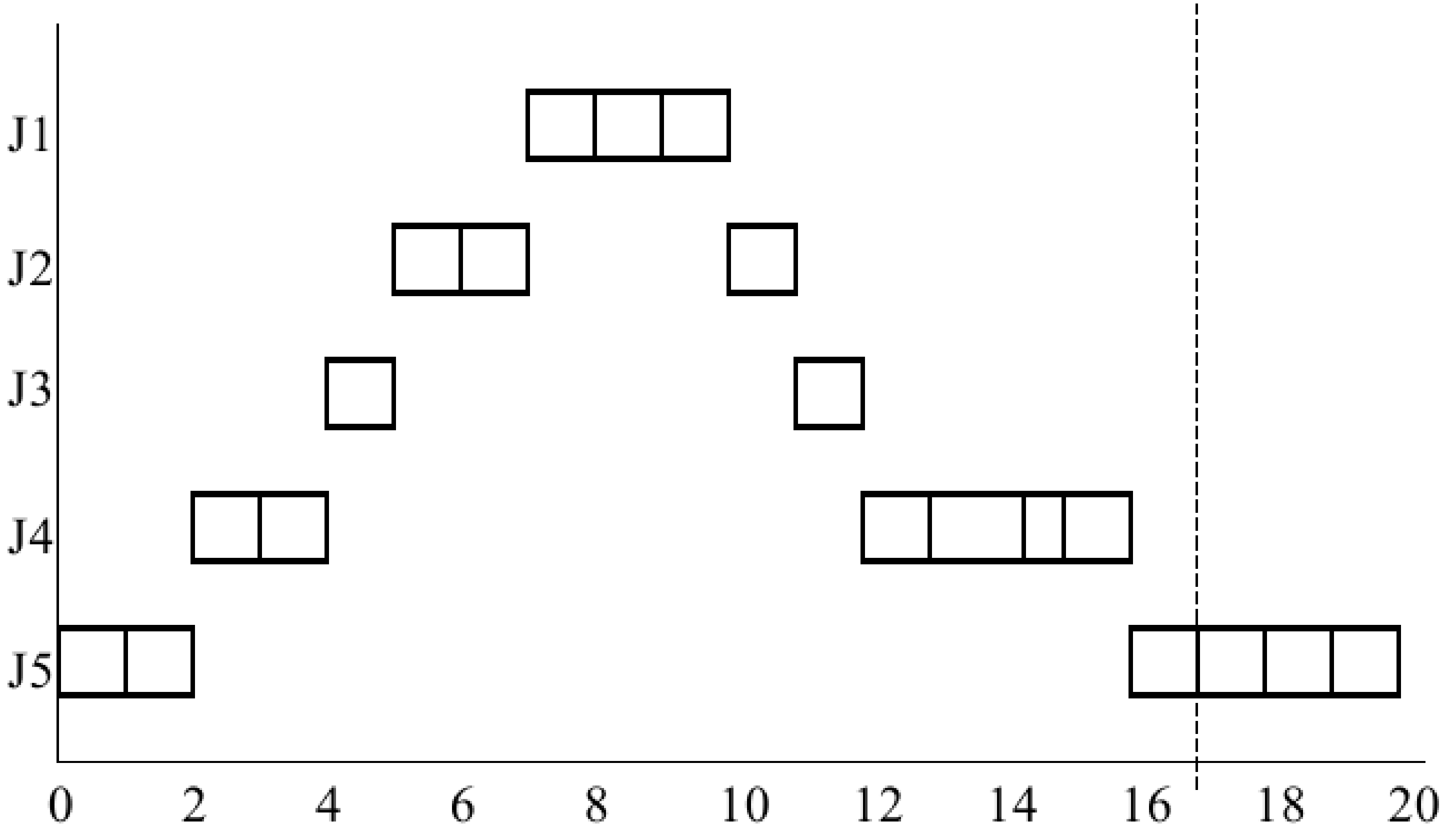


Figure 36

Example

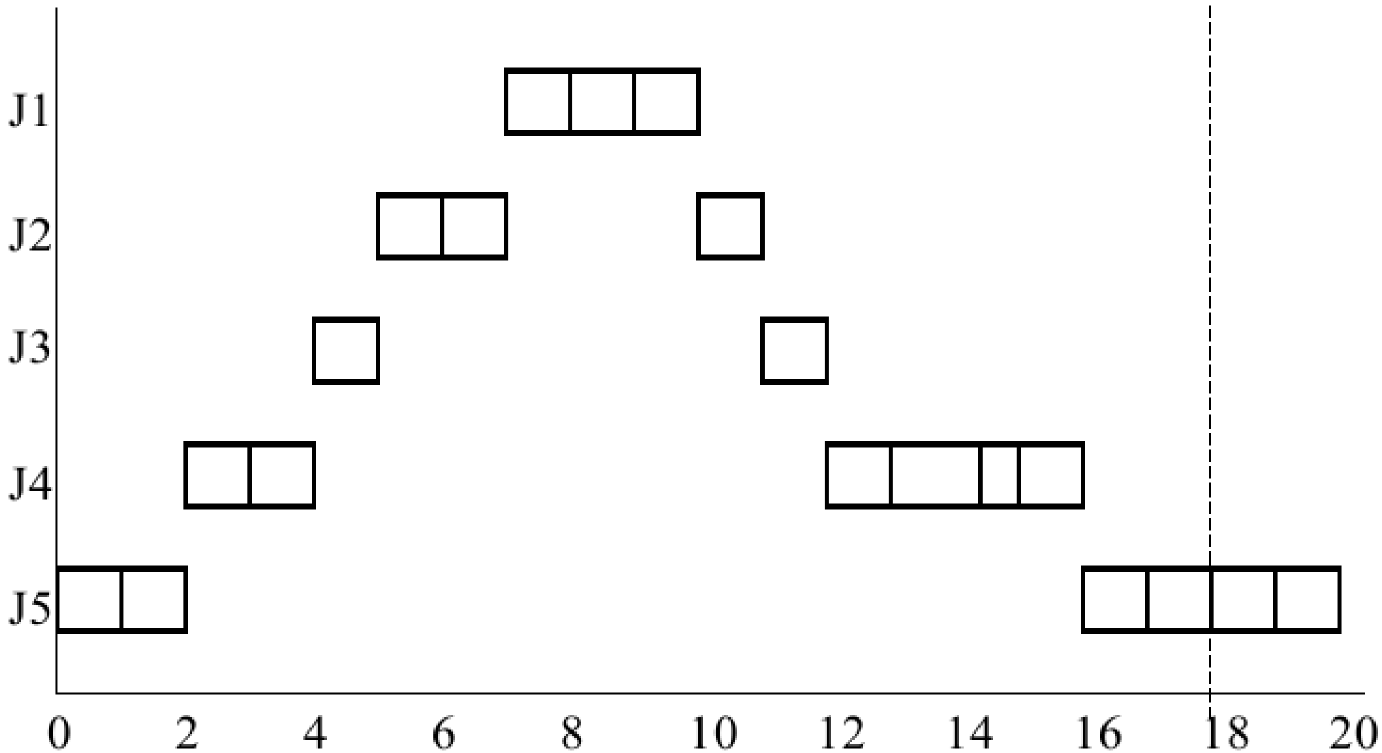


Figure 37

Example

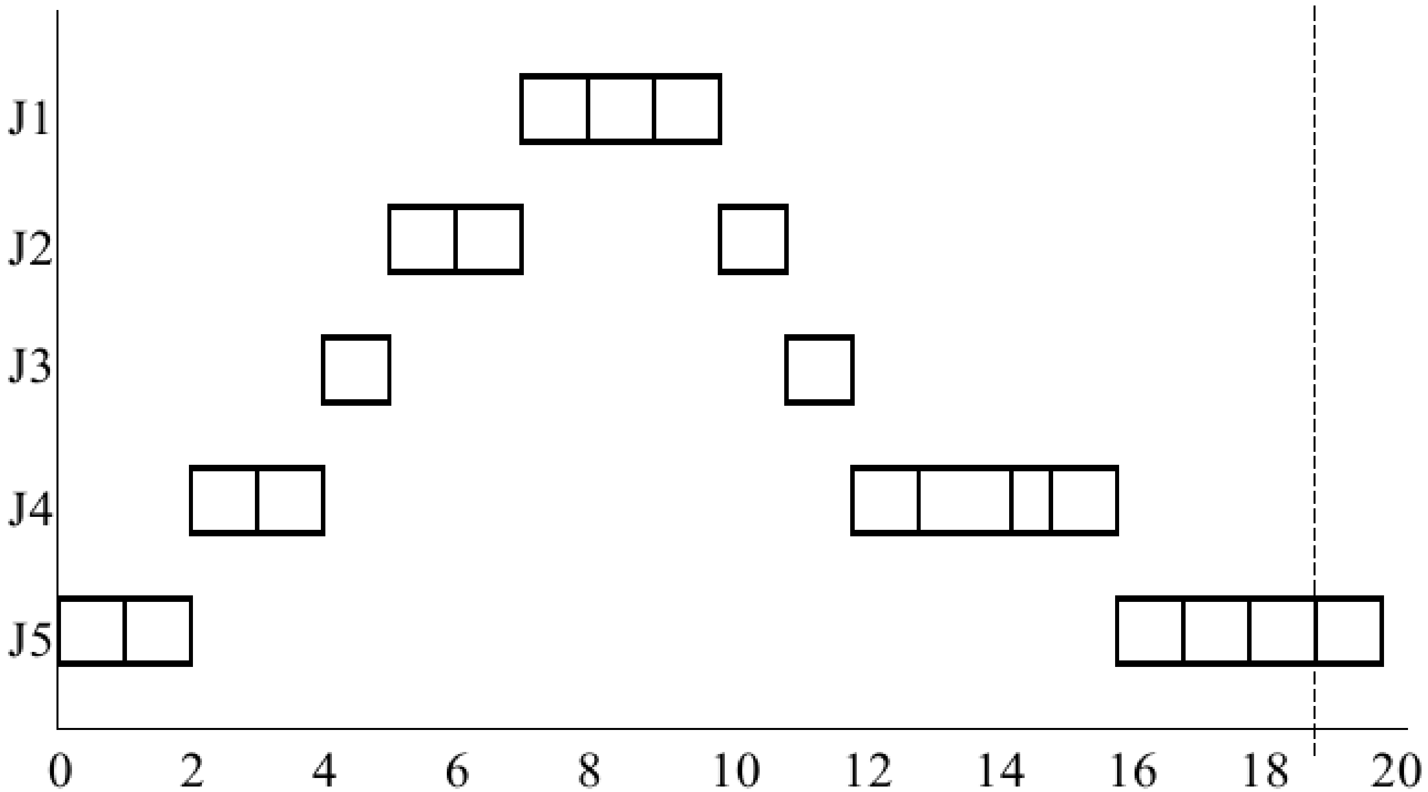


Figure 38

Example

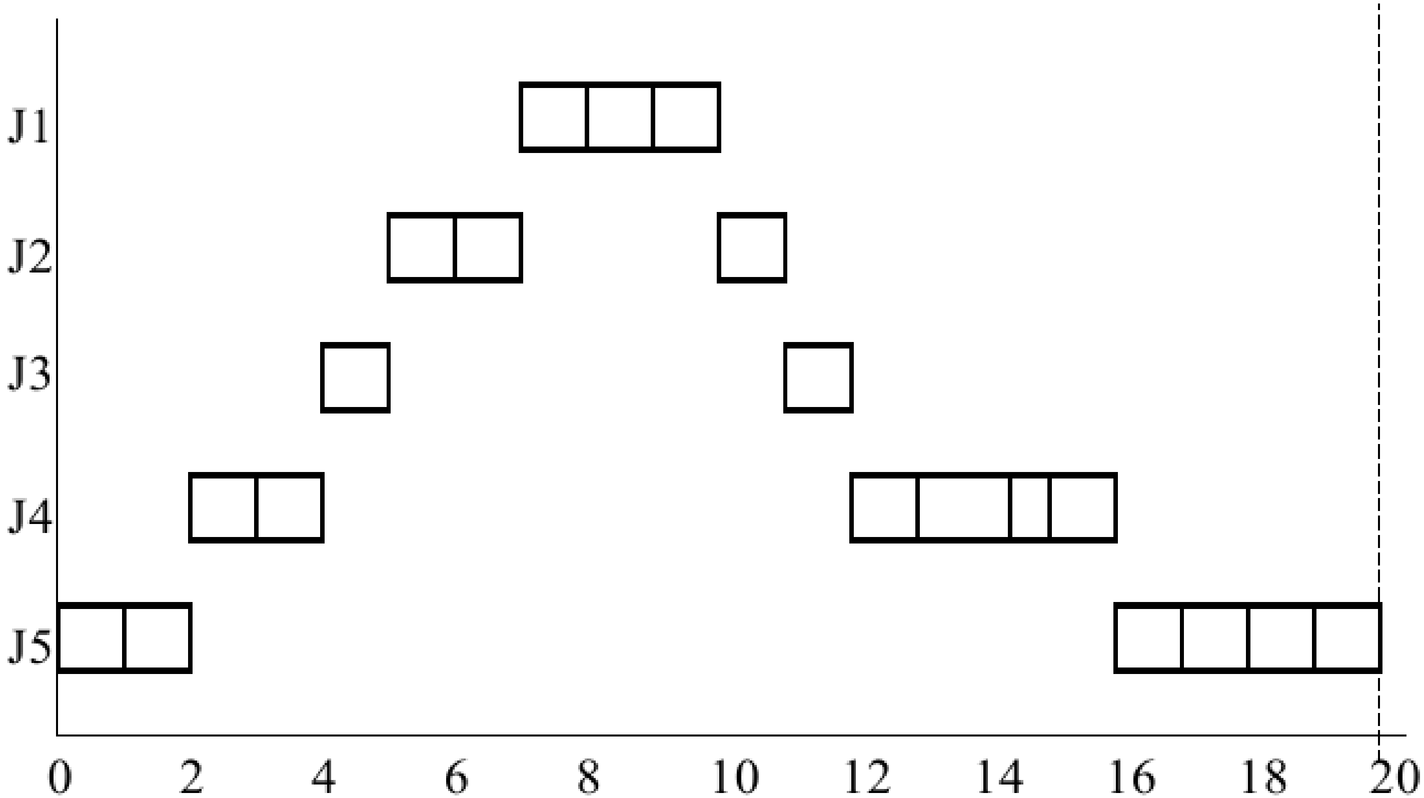
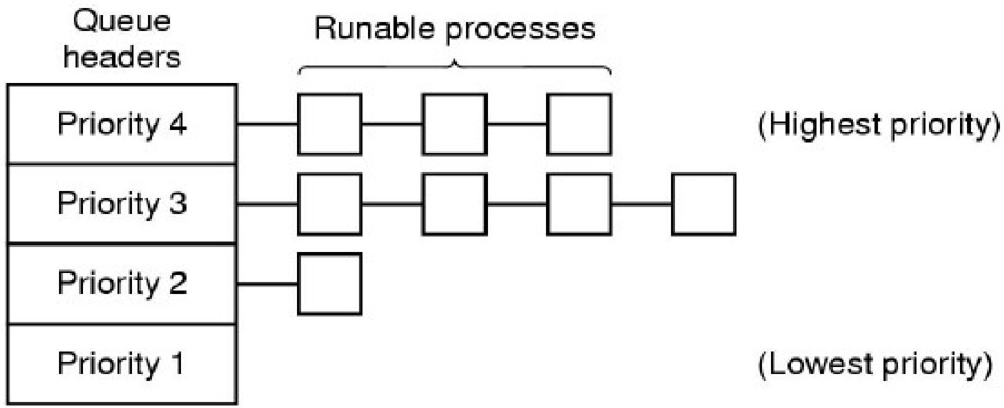


Figure 39

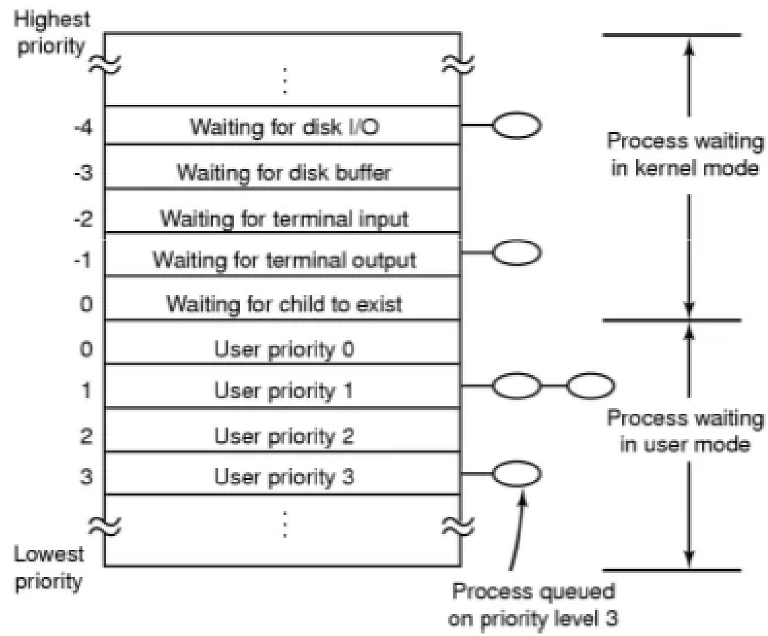
Priorities



- Usually implemented by multiple priority queues, with round robin on each queue
- Con
  - Low priorities can starve
    - Need to adapt priorities periodically
    - Based on ageing or execution history

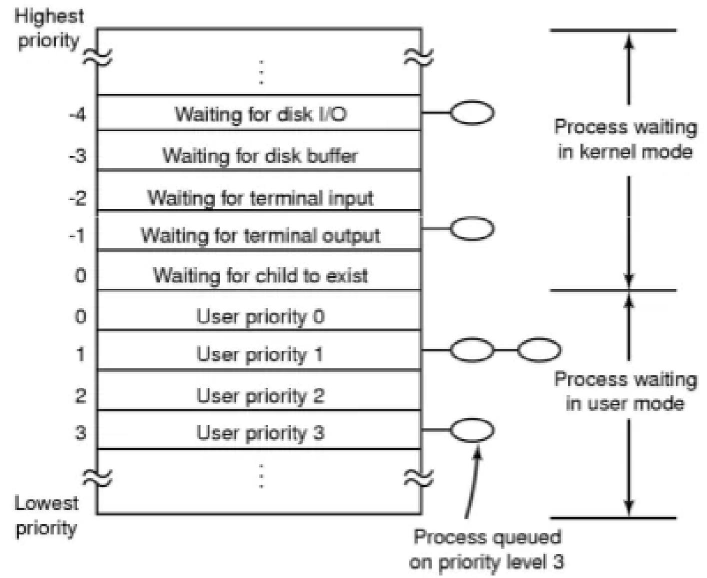
Traditional UNIX Scheduler

- Two-level Scheduler
  - High-level scheduler schedules processes between memory and disk
  - Low-level scheduler is CPU scheduler
    - Based on a multi-level queue structure with round robin at each level



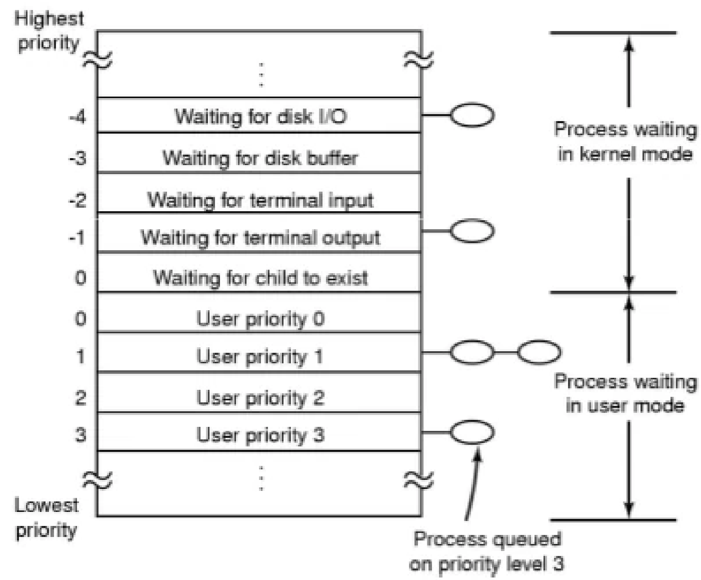
Traditional UNIX Scheduler

- The highest priority (lower number) is scheduled
  - Priorities are re-calculated once per second, and re-inserted in appropriate queue
    - Avoid starvation of low priority threads
    - Penalize CPU-bound threads



Traditional UNIX Scheduler

- $Priority = CPU_usage + nice + base$
  - CPU_usage = number of clock ticks
    - Decays over time to avoid permanently penalising the process
  - *Nice* is a value given to the process by user to permanently boost of reduce its priority
    - Reduce priority of background jobs
  - *Base* is a set of hardwired, negative values used to boost priority of I/O bound system activities



Some Issues with Priorities

- Require adaption over time to avoid starvation (not considering hard real-time which relies on strict priorities)
- Adaption is:
  - usually ad-hoc,
    - hence behaviour not thoroughly understood, and unpredictable
  - Gradual, hence unresponsive
- Difficult to guarantee a desired share of the CPU
- No way for applications to trade CPU time

Lottery Scheduling

- Each process is issued with “lottery tickets” which represent the right to use/consume a resource
  - Example: CPU time
- Access to a resource is via “drawing” a lottery winner
  - The more tickets a process possesses, the higher chance the process has of winning.

Lottery Scheduling

- Advantages
  - Simple to implement
  - Highly responsive
    - can reallocate tickets held for immediate effect
  - Tickets can be traded to implement individual scheduling policy between co-operating threads
  - Stavation free
    - A process holding a ticket will eventually be scheduled.

Example Lottery Scheduling

- Four process running concurrently
  - Process A: 15% CPU
  - Process B: 25% CPU
  - Process C: 5% CPU
  - Process D: 55% CPU
- How many tickets should be issued to each?

Lottery Scheduling Performance

- Observed performance of two processes with varying ratios of tickets

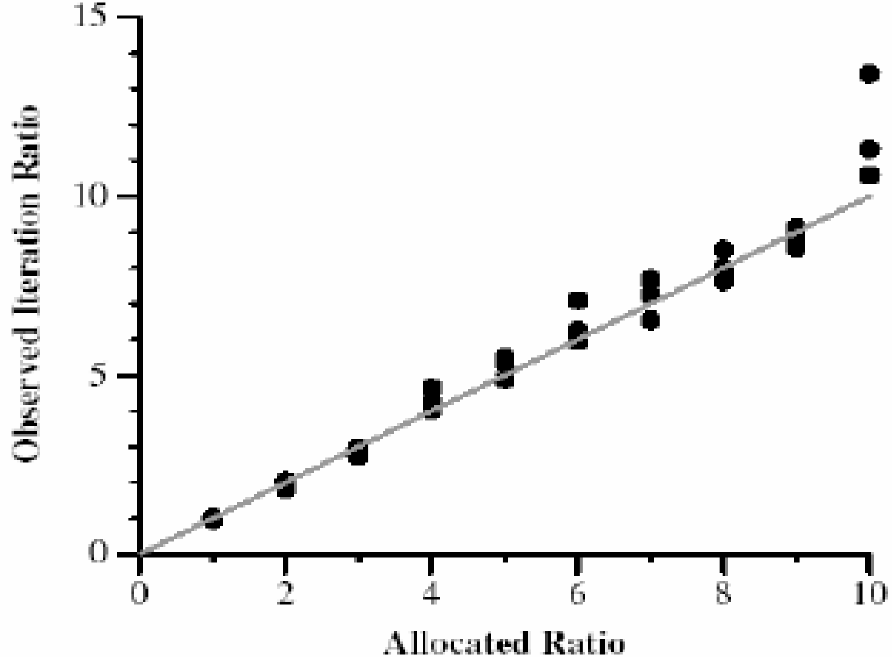


Figure 4: **Relative Rate Accuracy.** For each allocated ratio, the observed ratio is plotted for each of three 60 second runs. The gray line indicates the ideal where the two ratios are identical.

Fairness

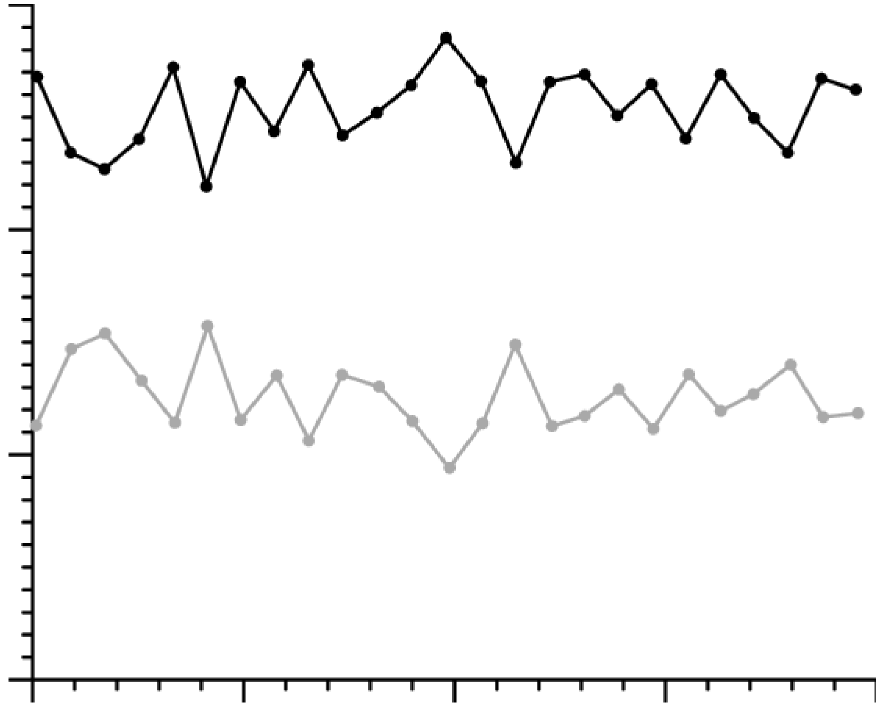


Figure 40: Fairness Over Time. Two tasks executing the Dhrystone benchmark with a 2 : 1 ticket allocation. Averaged over the entire run, the two tasks executed 25378 and 12619 iterations/sec., for an actual ratio of 2.01 : 1.

Fair-Share Scheduling

- So far we have treated processes as individuals
- Assume two users
  - One user has 1 process
  - Second user has 9 processes
- The second user gets 90% of the CPU
- Some schedulers consider the owner of the process in determining which process to schedule
  - e.g., for the above example we could schedule the first user's process 9 times more often than the second user's processes
- Many possibilities exist to determine a fair schedule
  - E.g. Appropriate allocation of tickets in lottery scheduler