

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

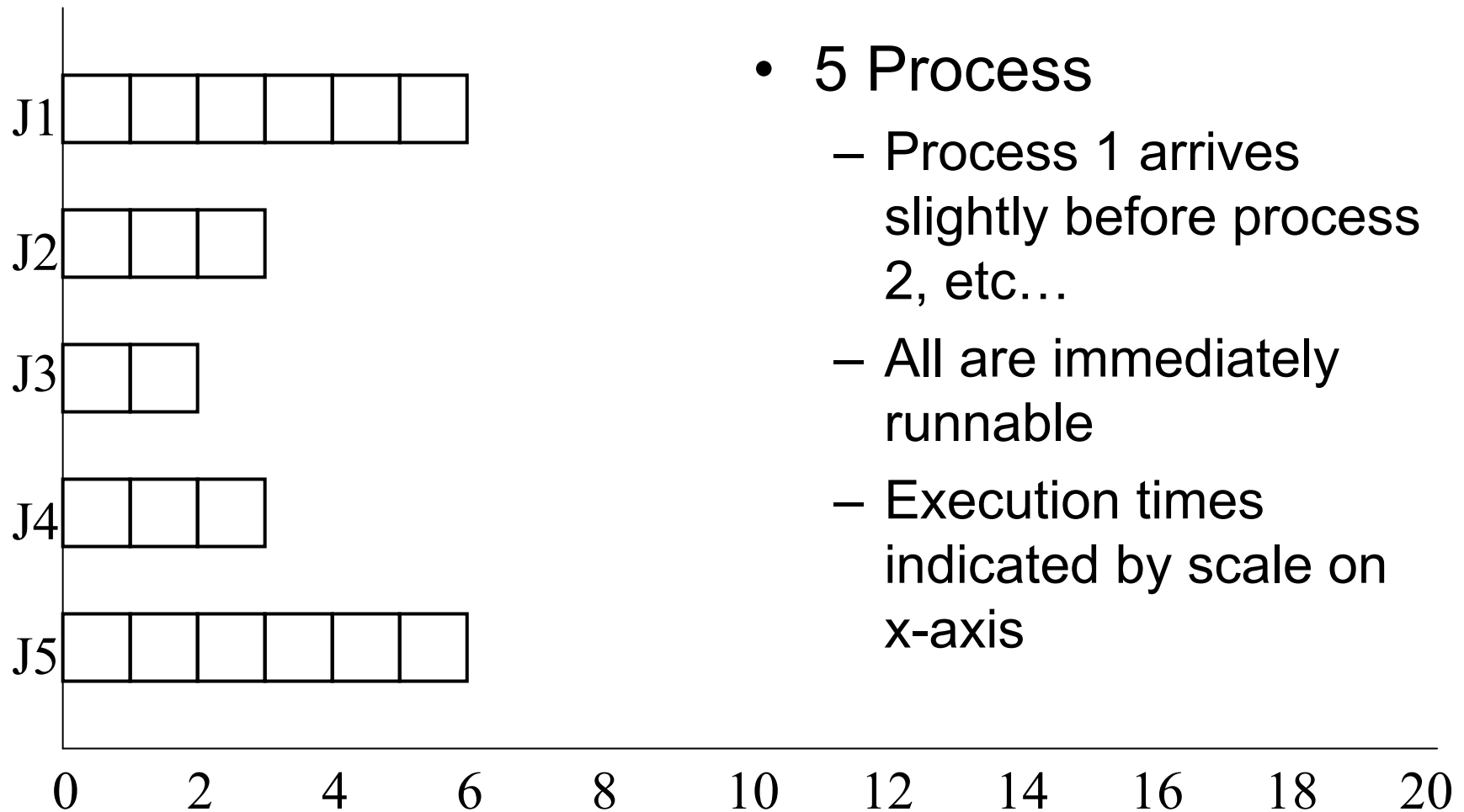


Round Robin Scheduling

- 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

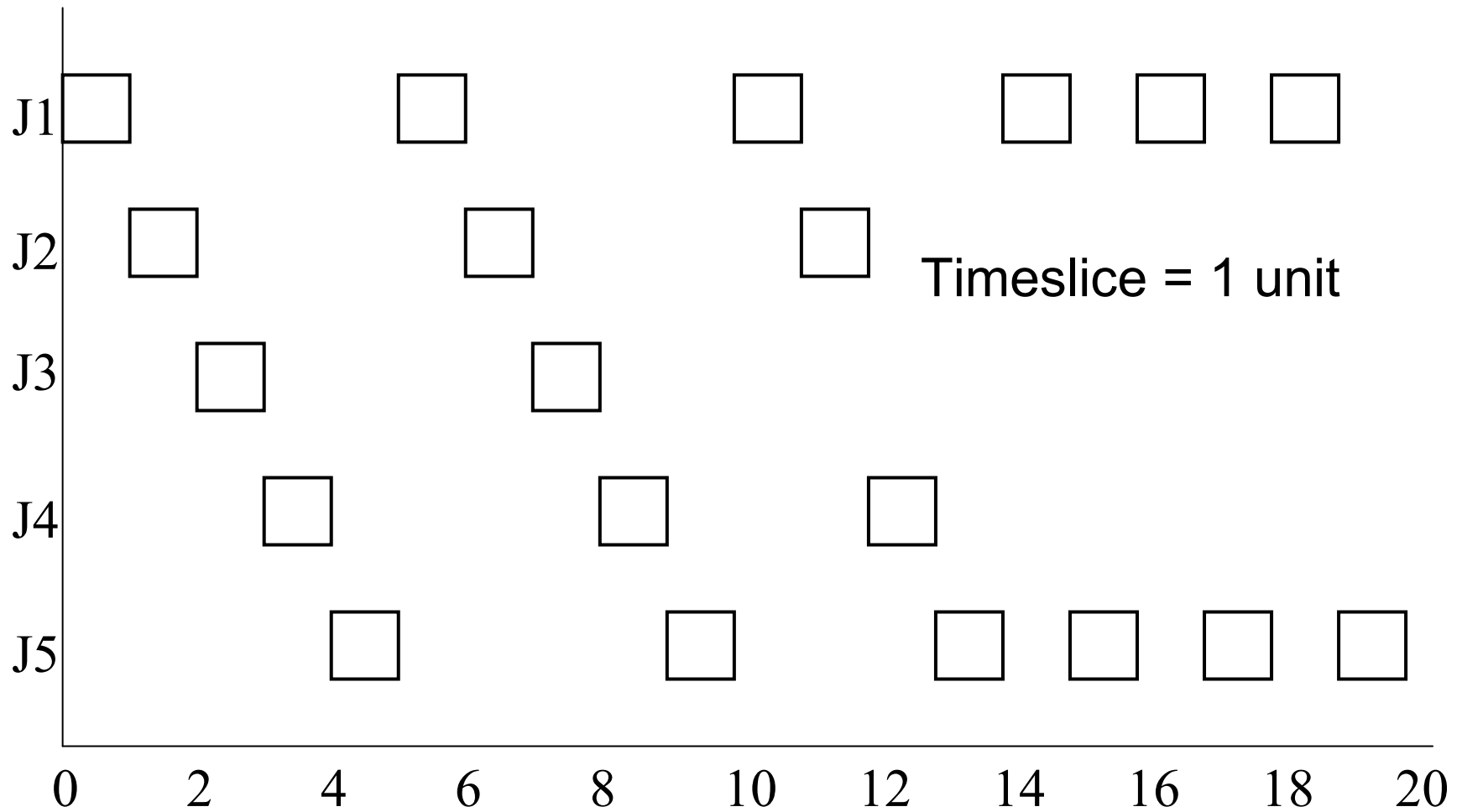


- 5 Process

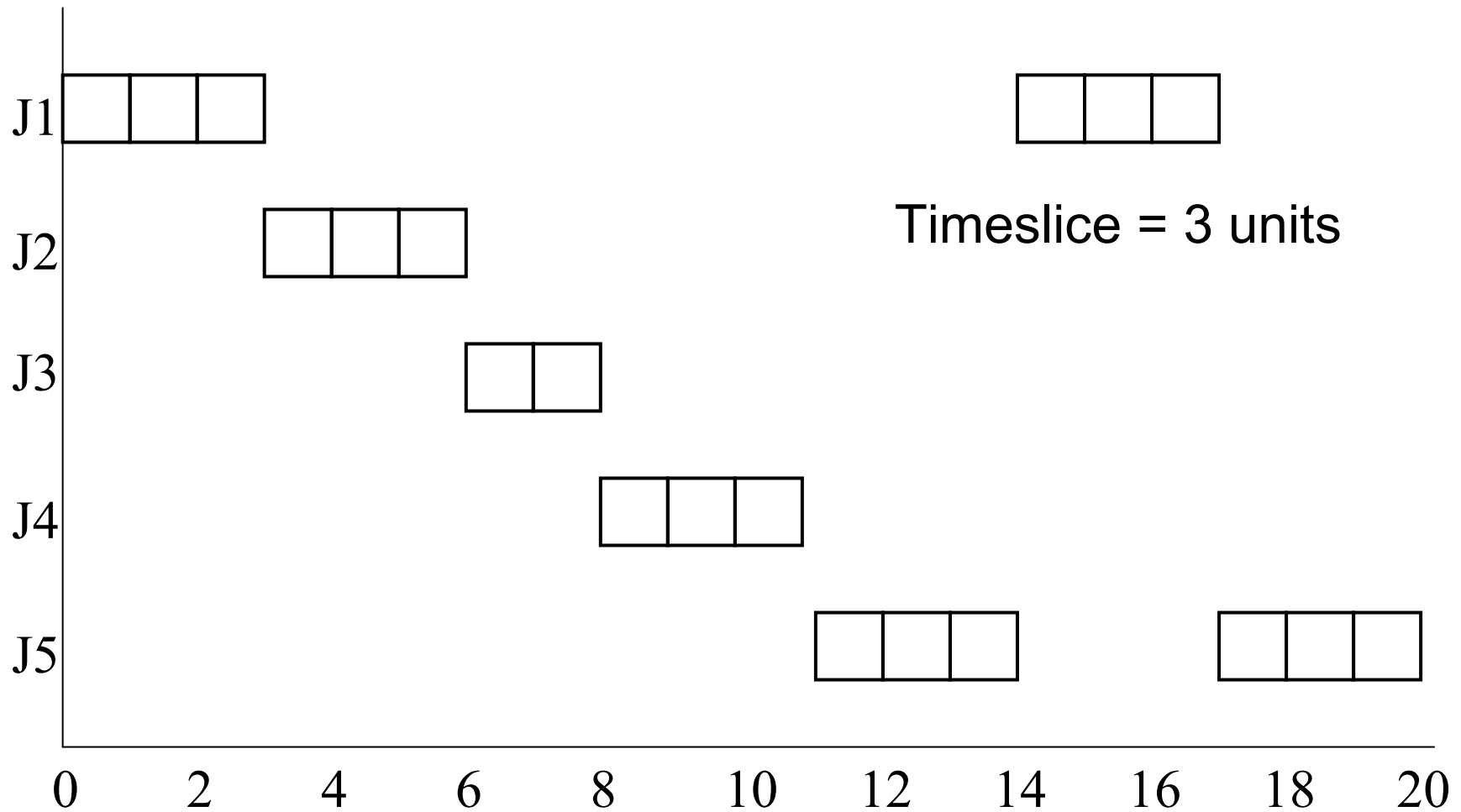
- Process 1 arrives slightly before process 2, etc...
- All are immediately runnable
- Execution times indicated by scale on x-axis



Round Robin Schedule



Round Robin Schedule



Round Robin

- 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 robin 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

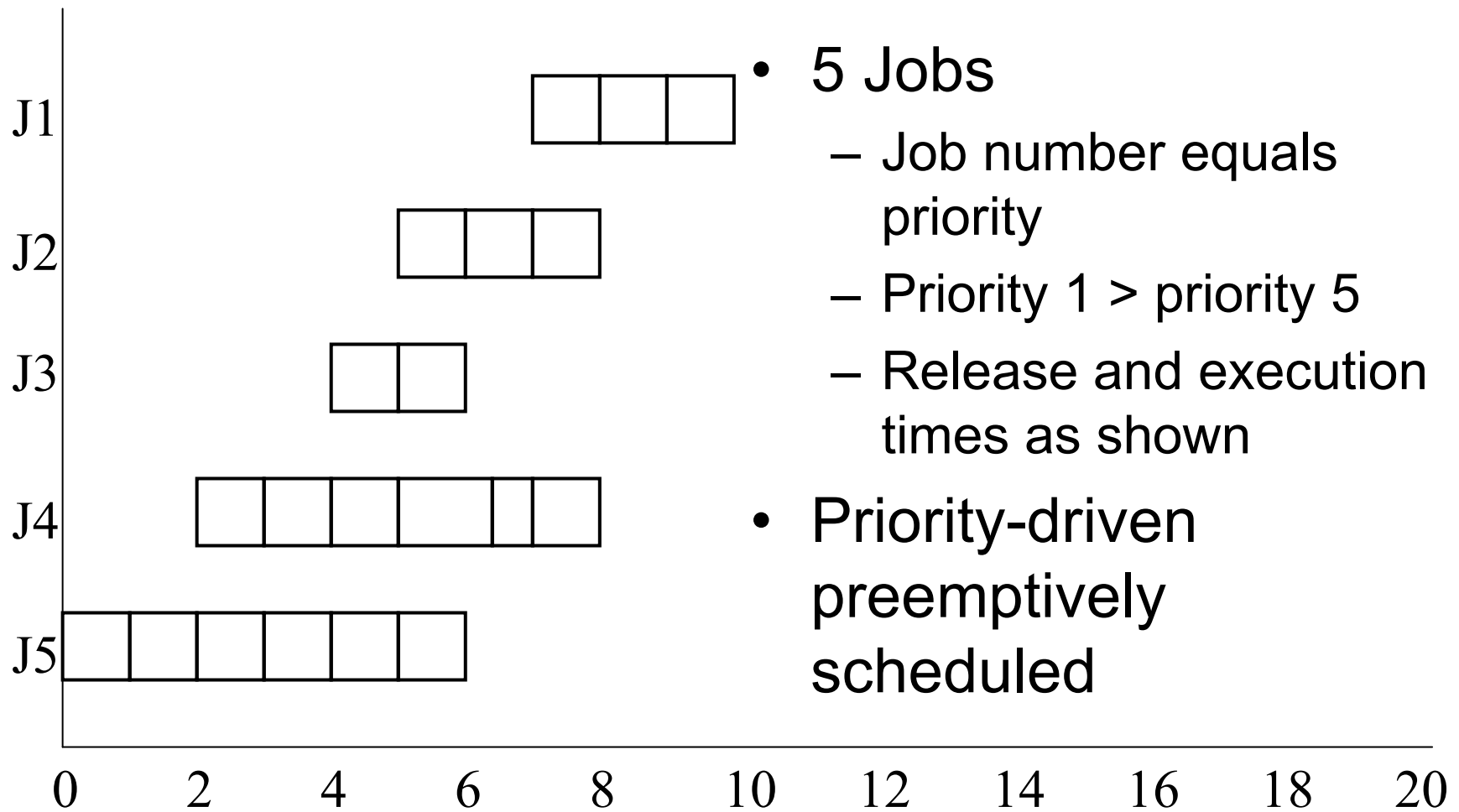


Priorities

- 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



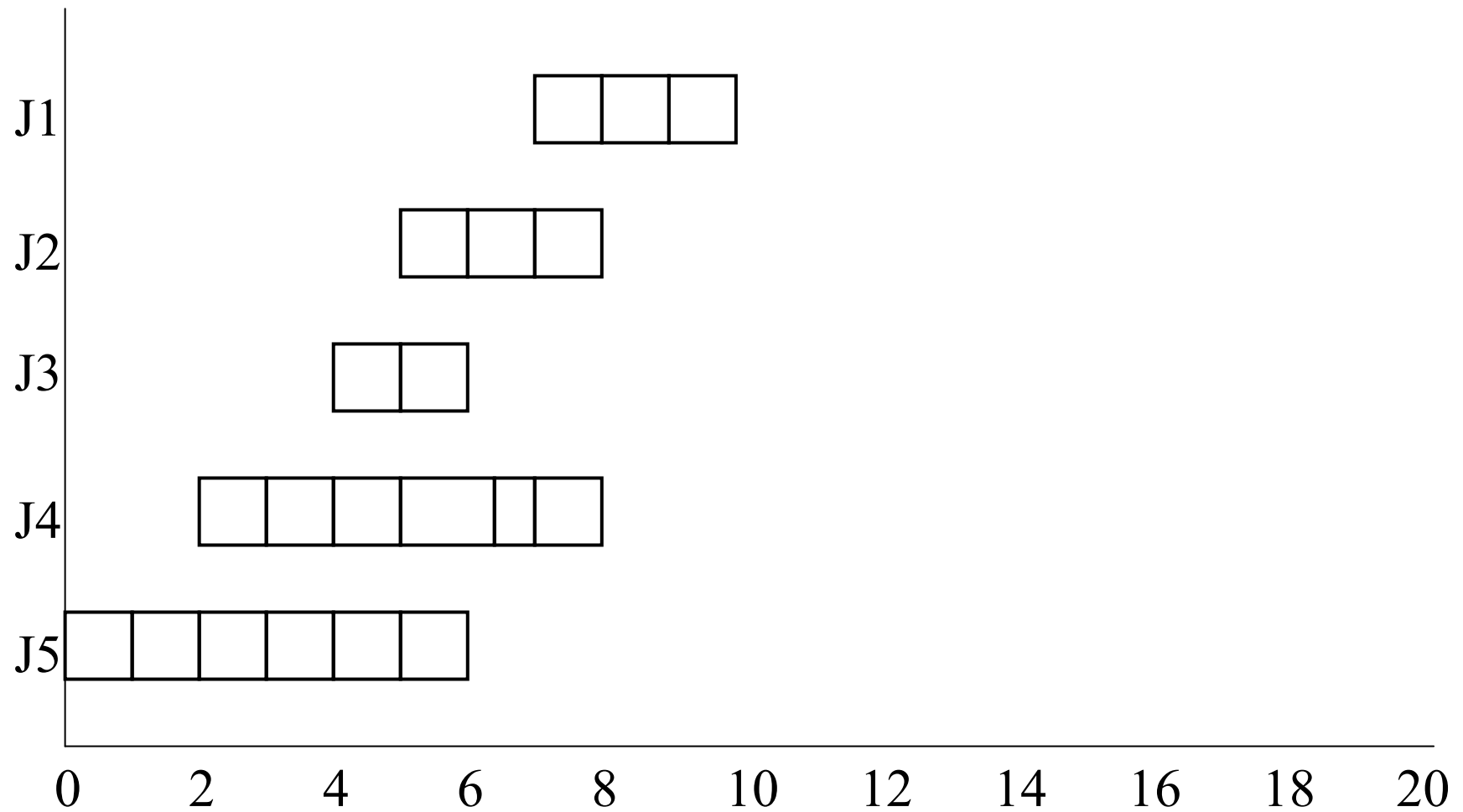
Example



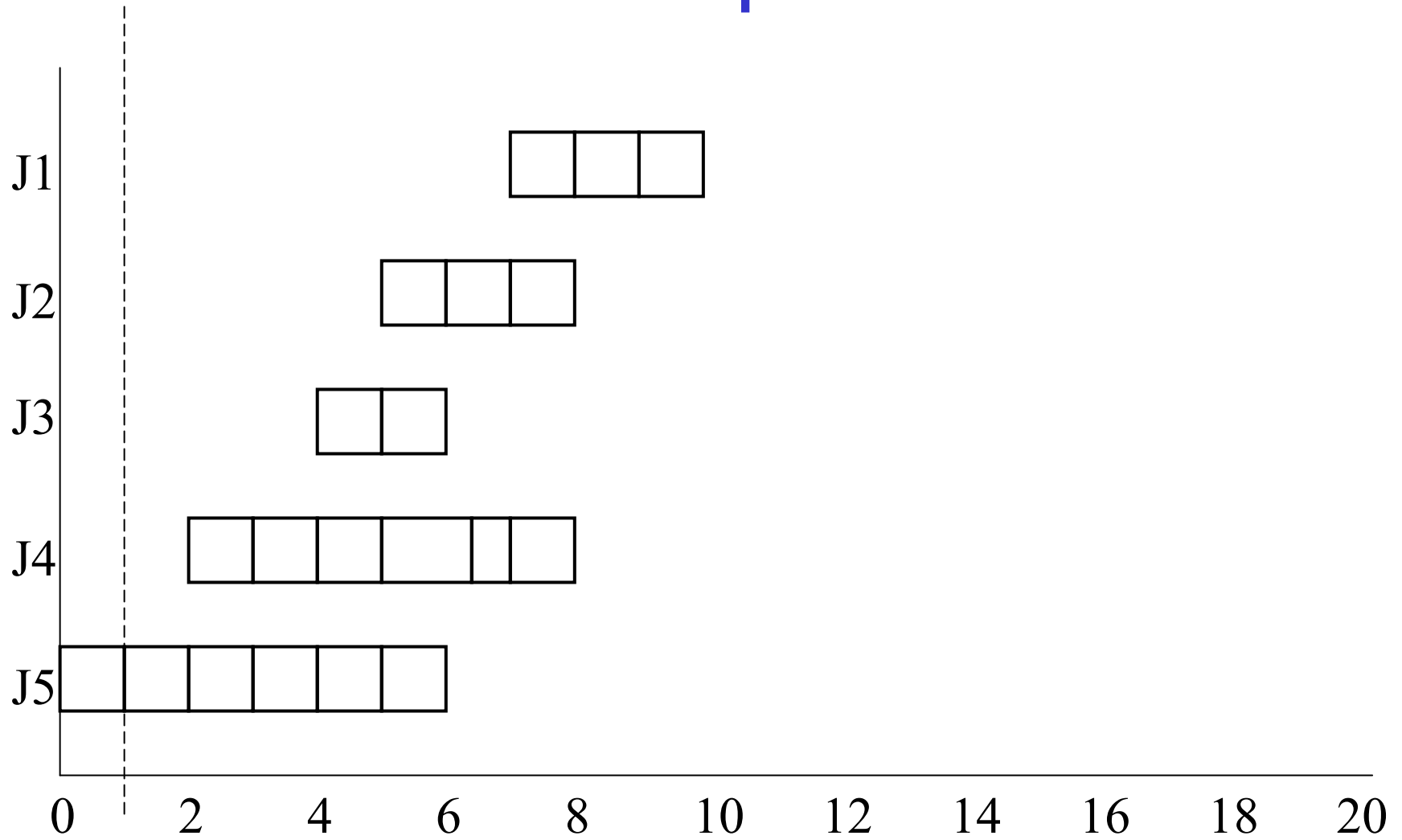
- 5 Jobs
 - Job number equals priority
 - Priority 1 > priority 5
 - Release and execution times as shown
- Priority-driven preemptively scheduled



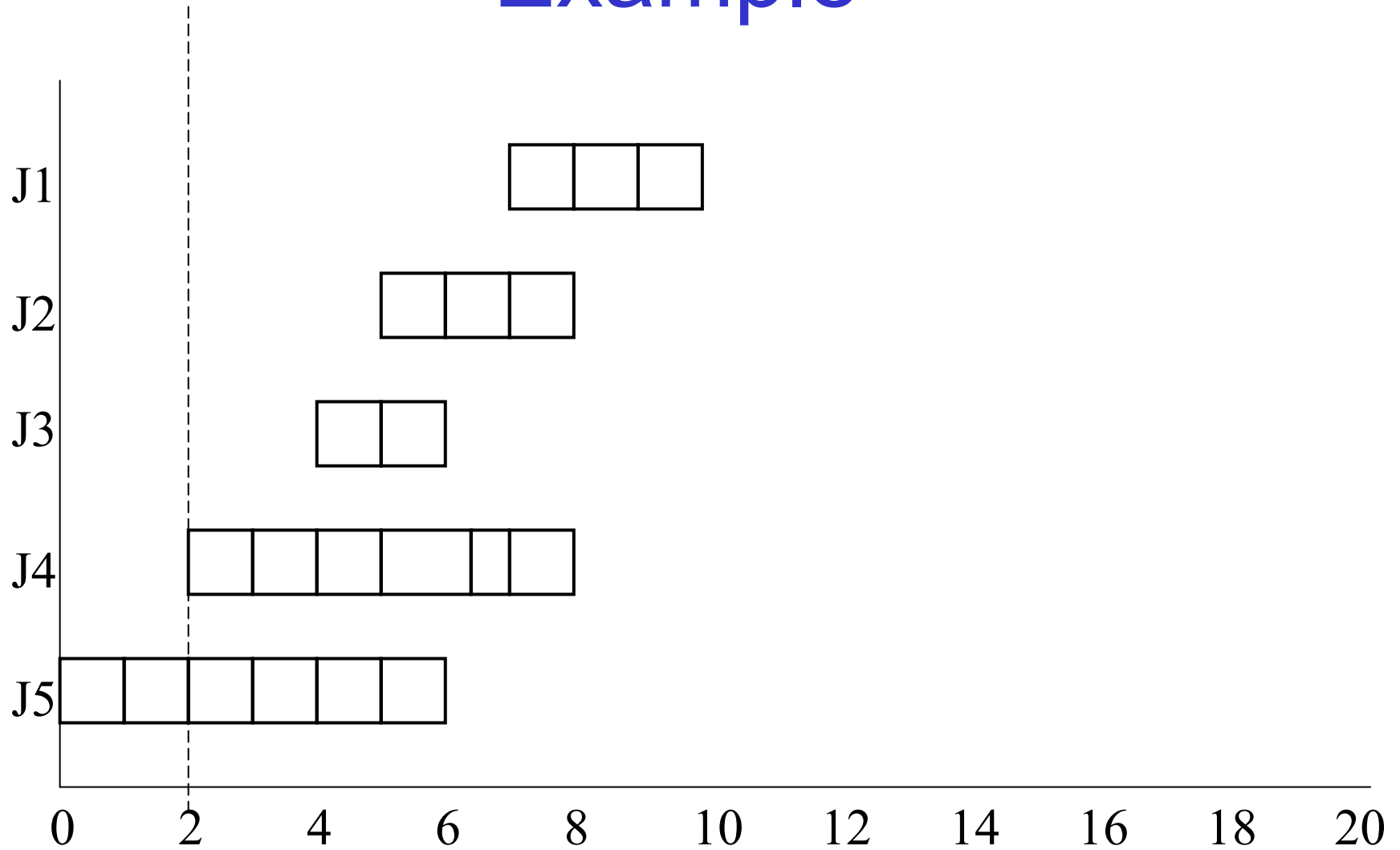
Example



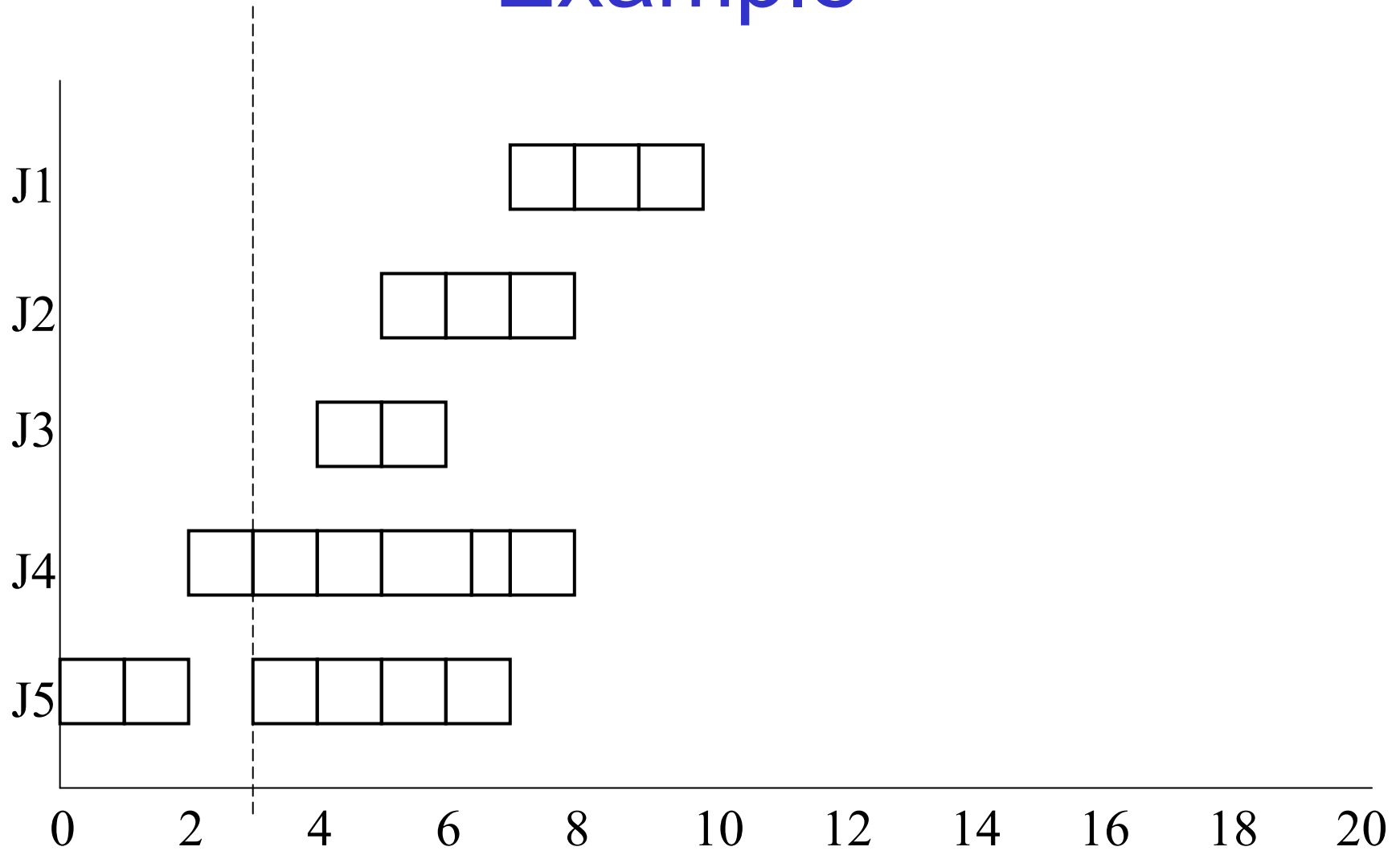
Example



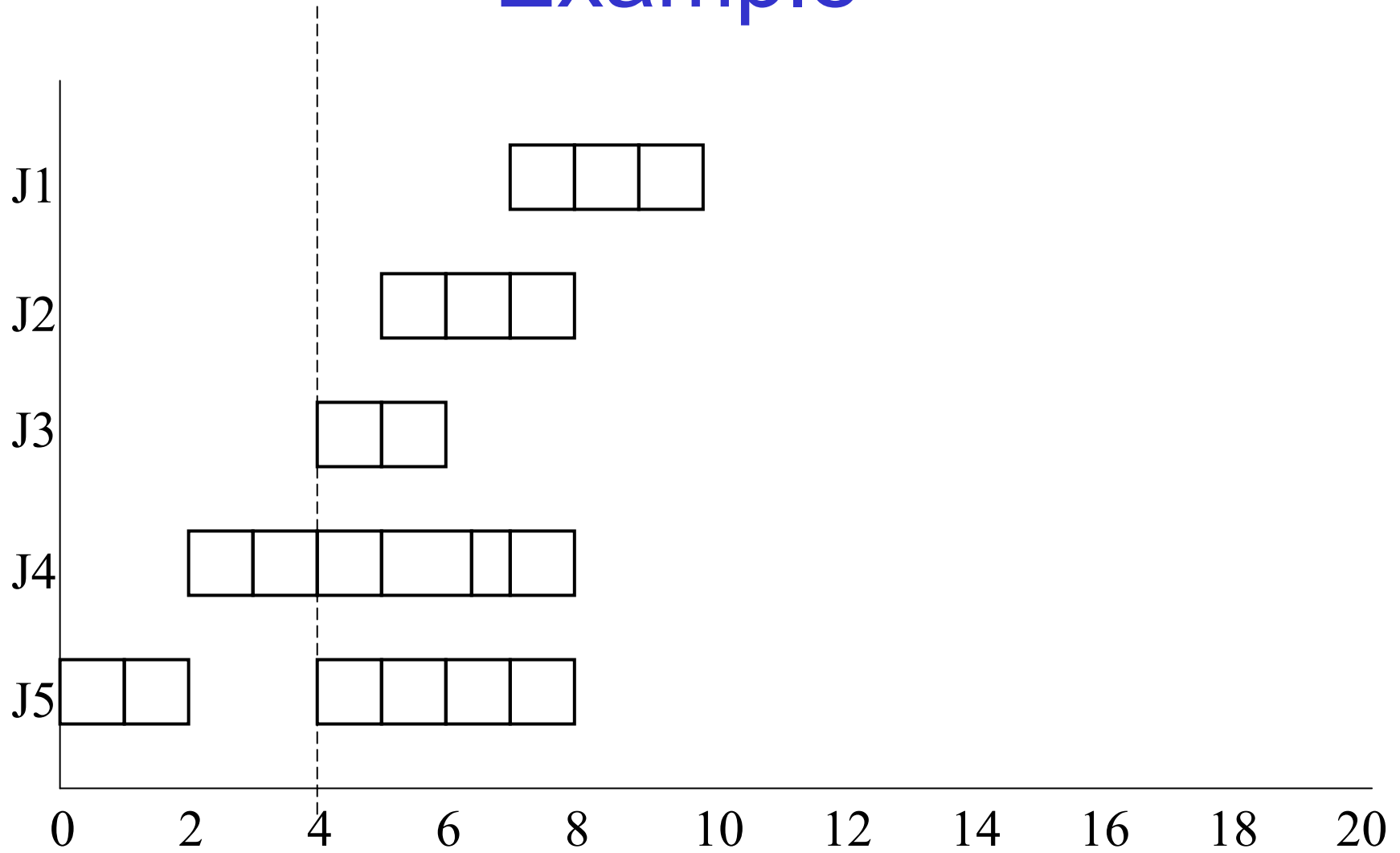
Example



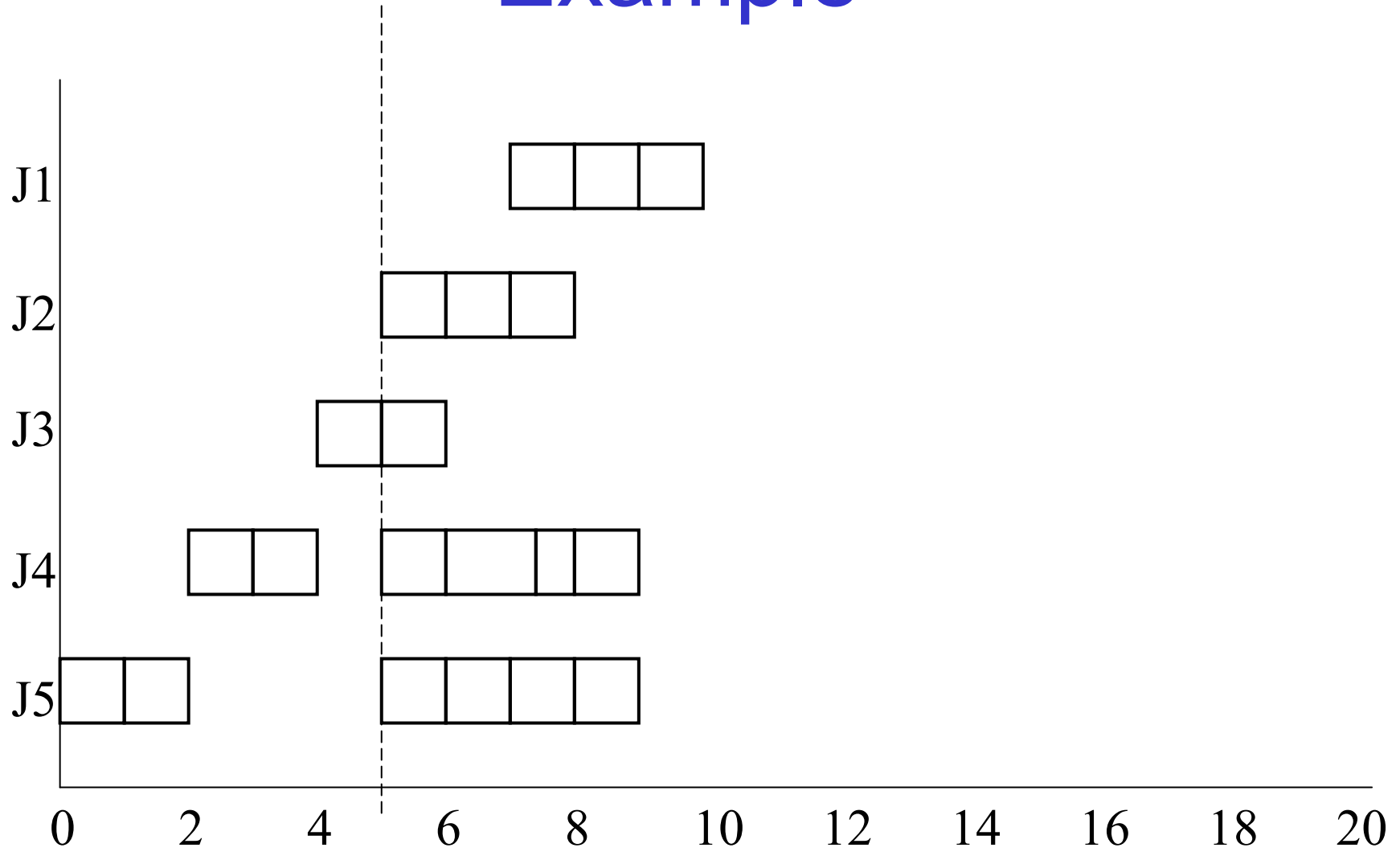
Example



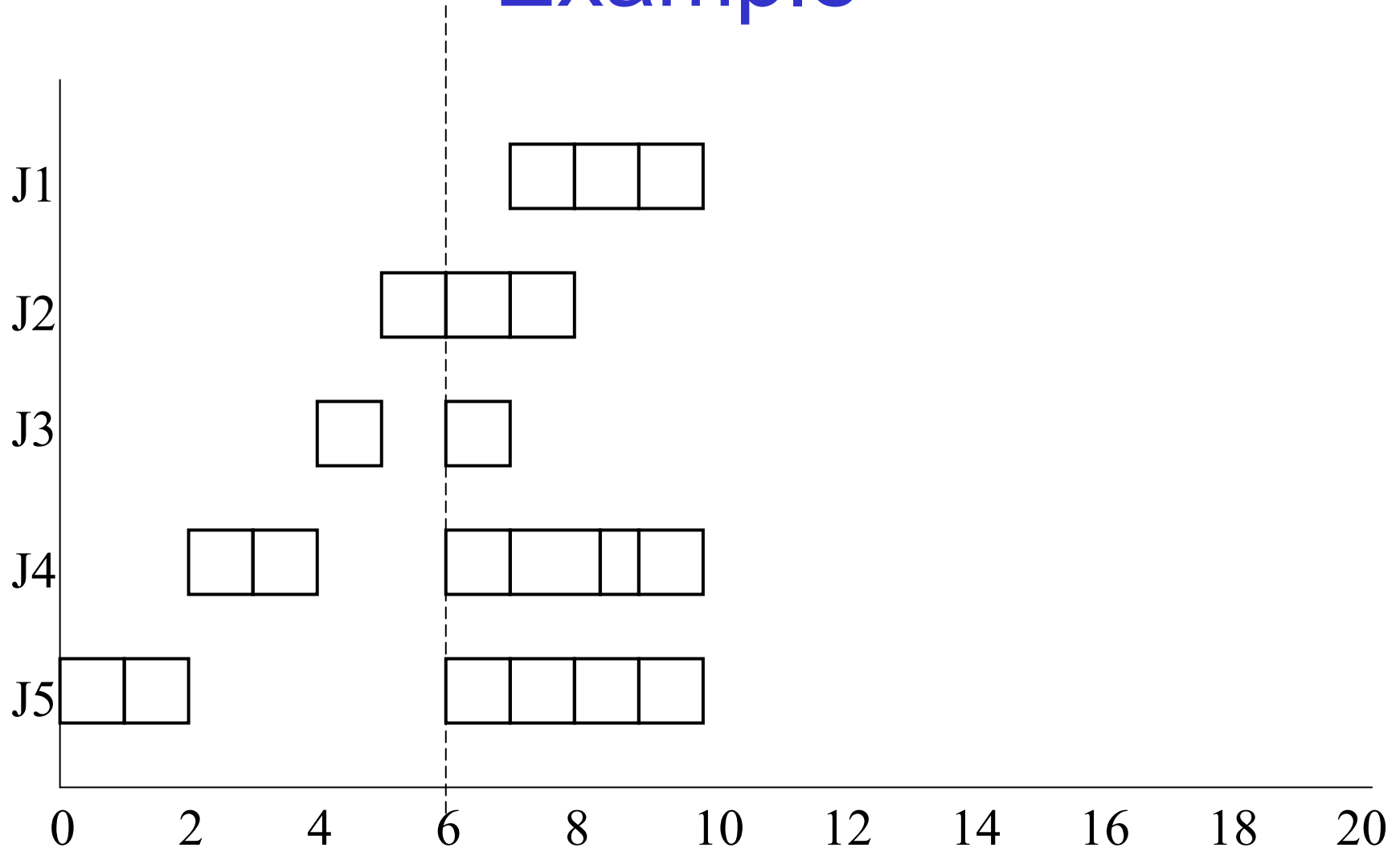
Example



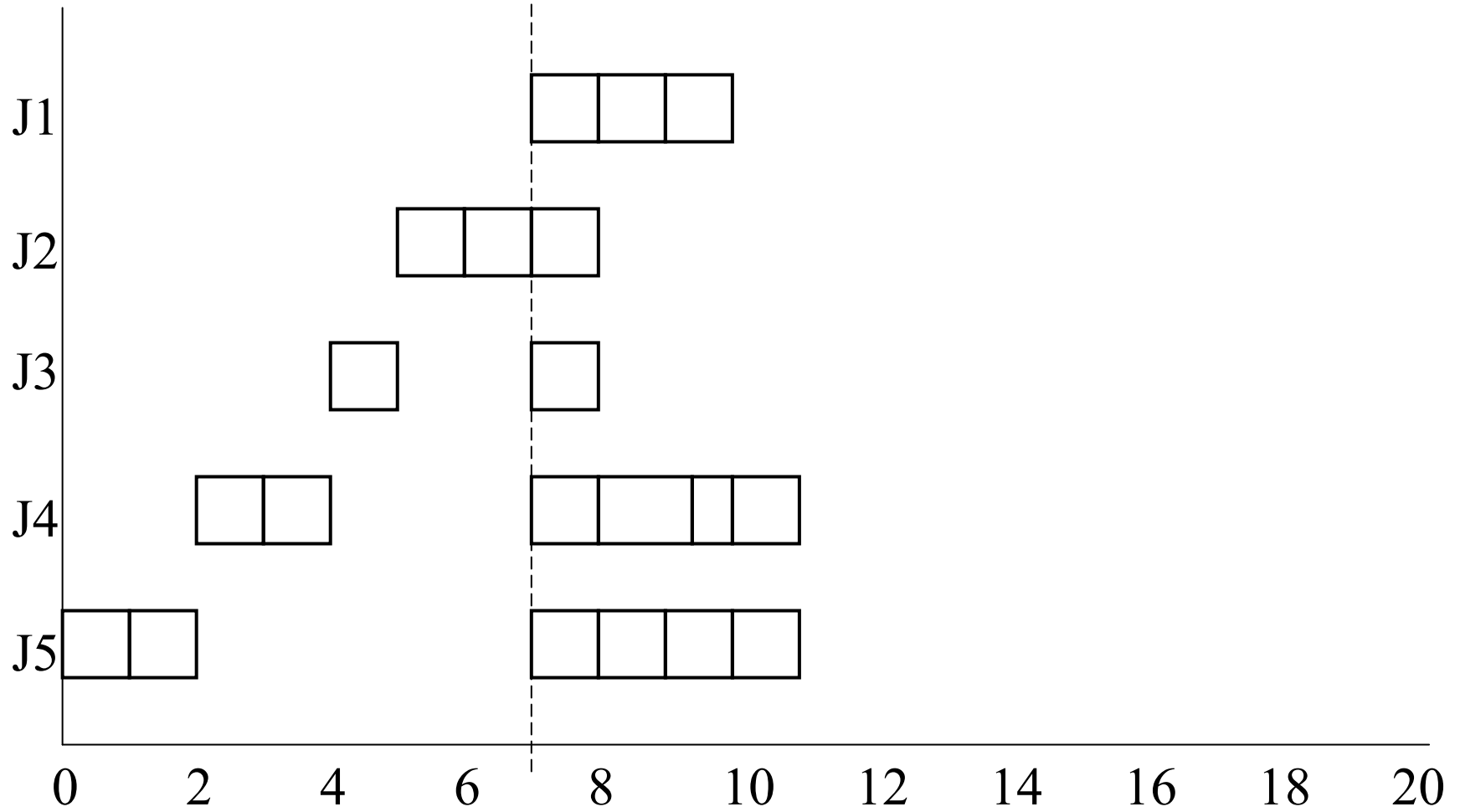
Example



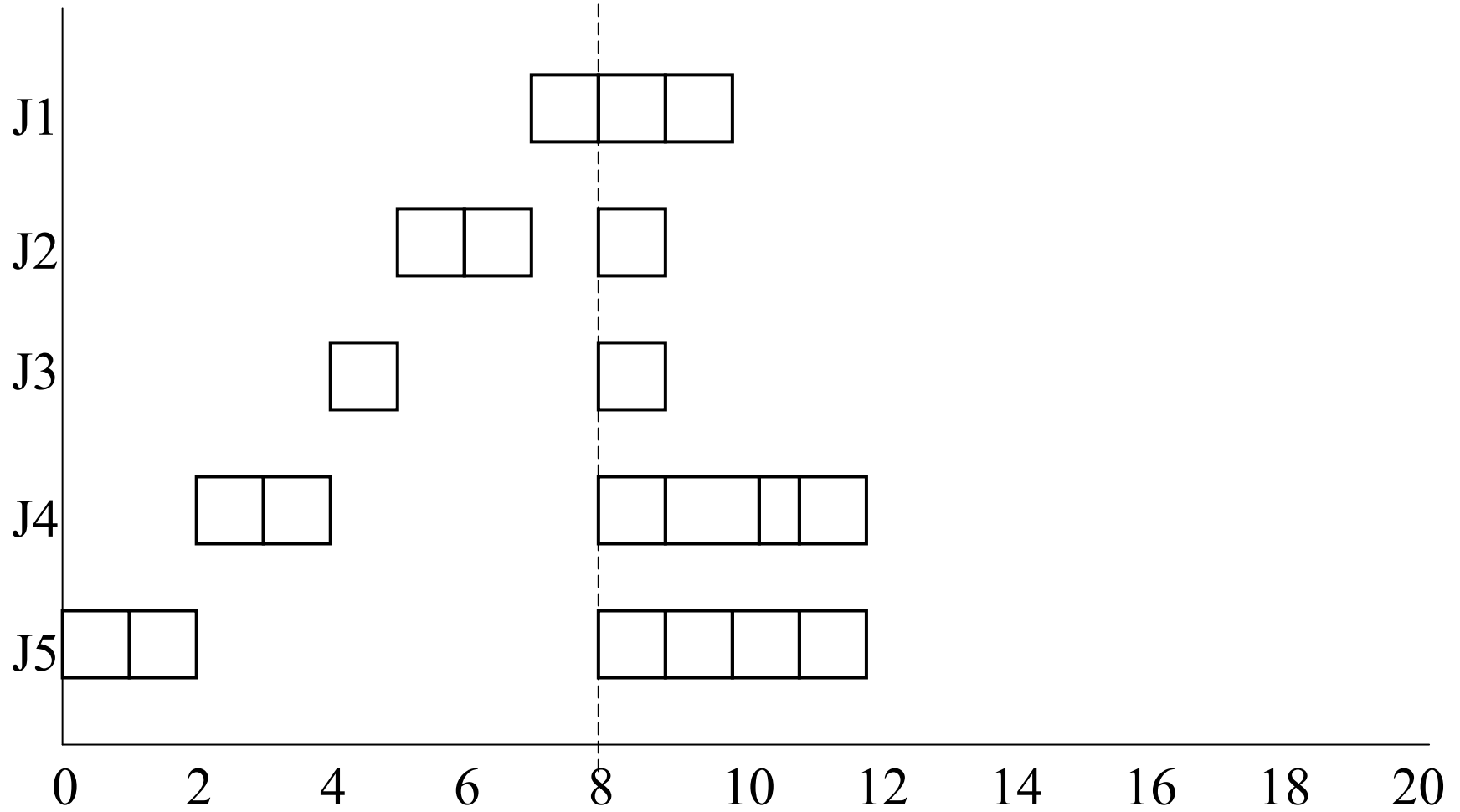
Example



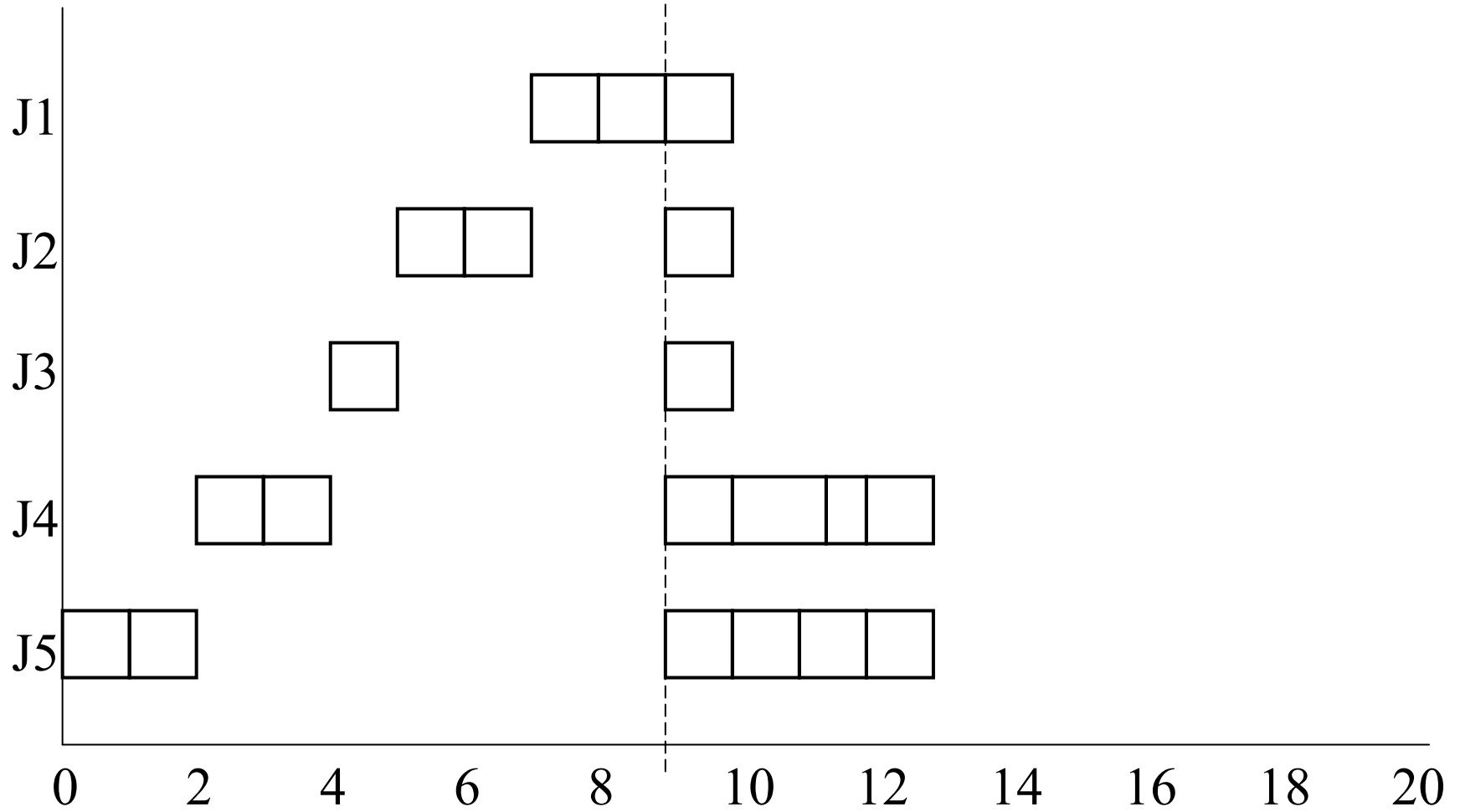
Example



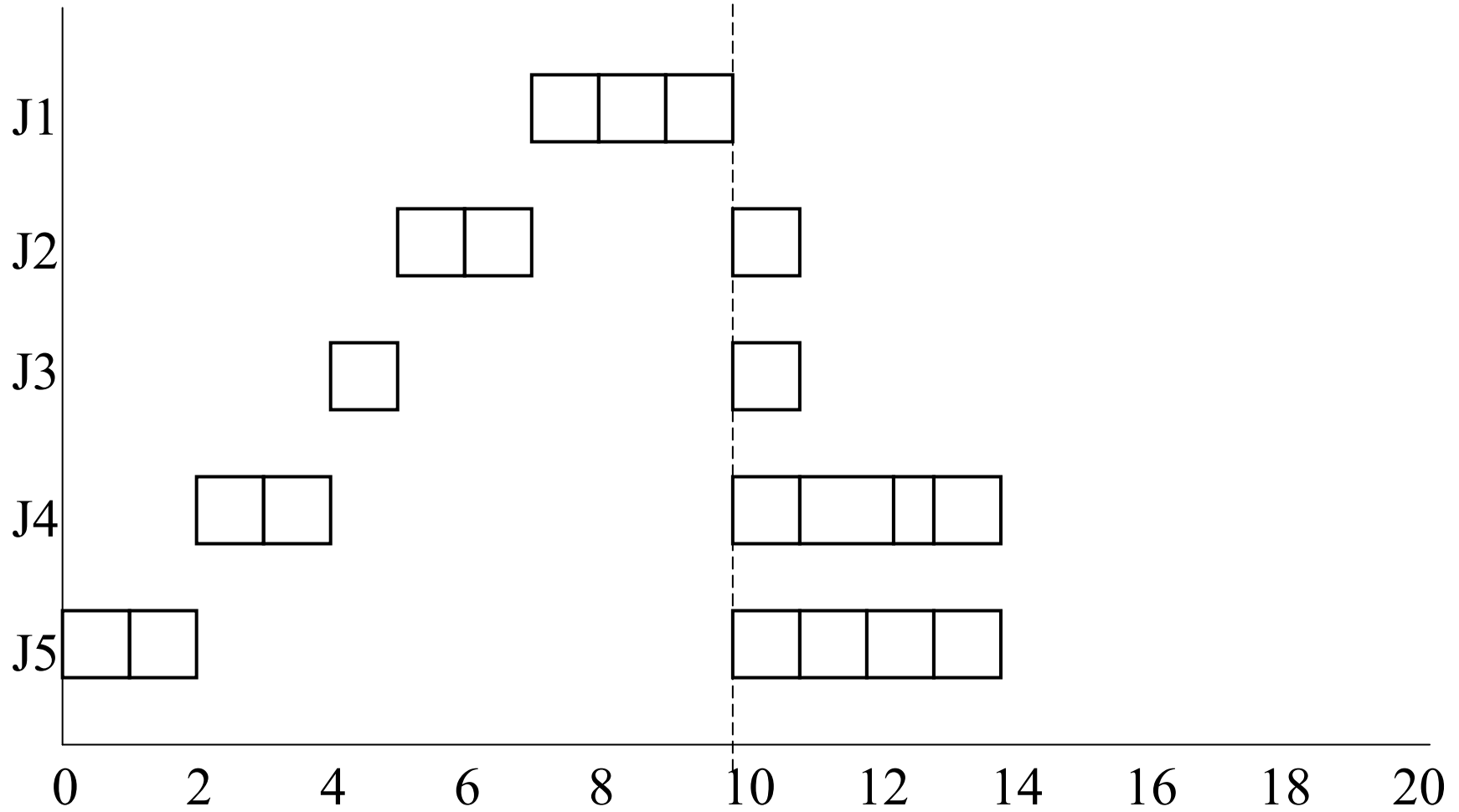
Example



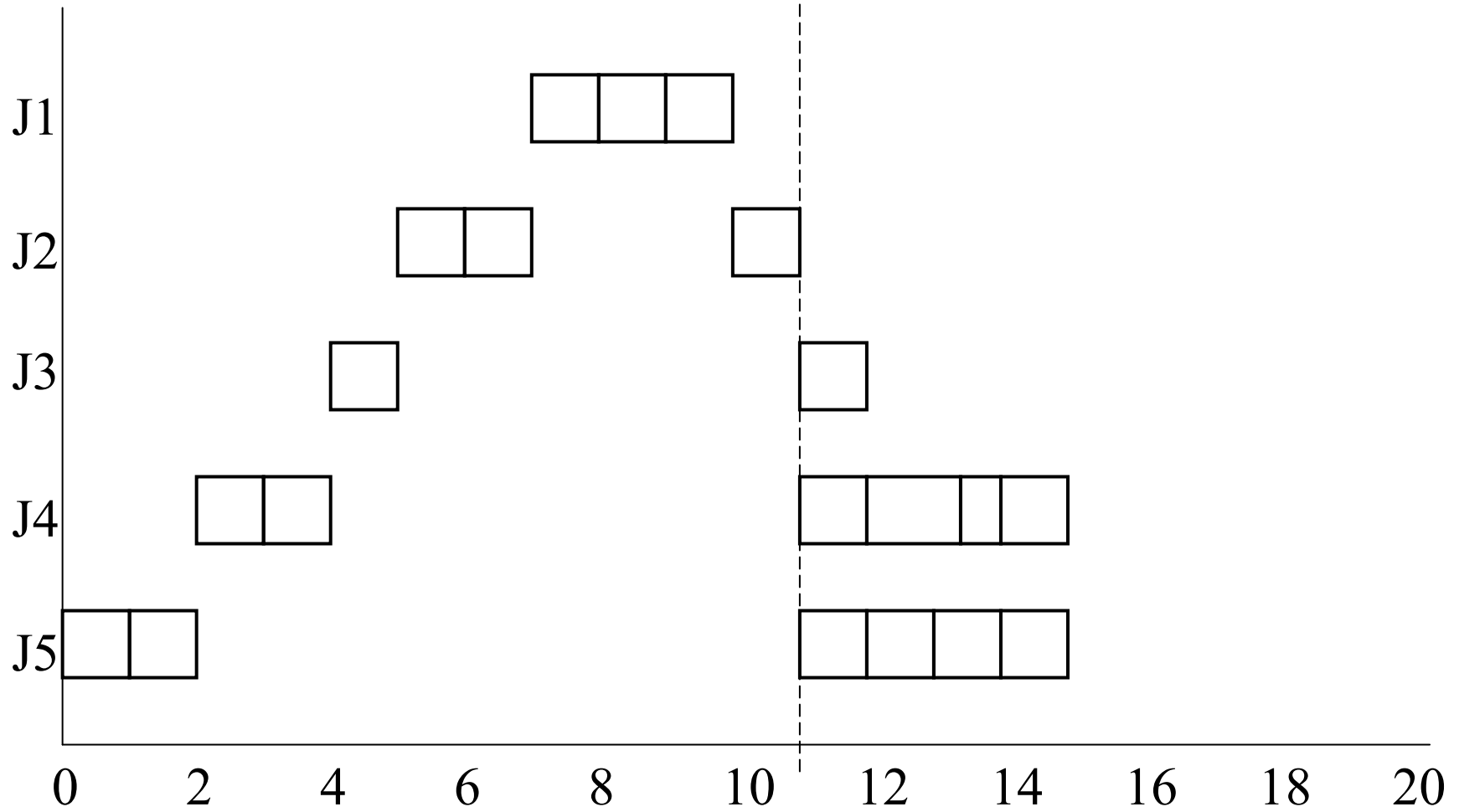
Example



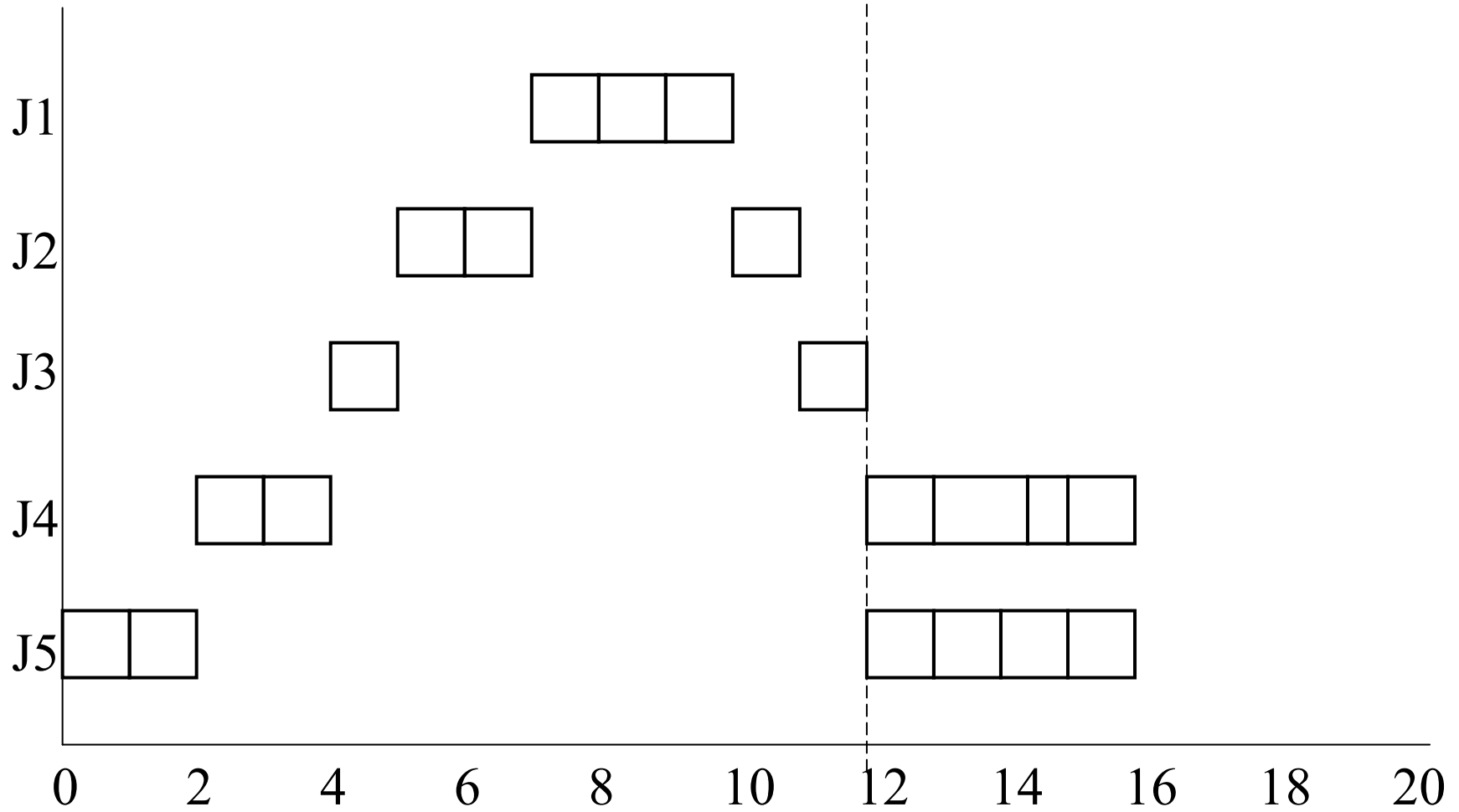
Example



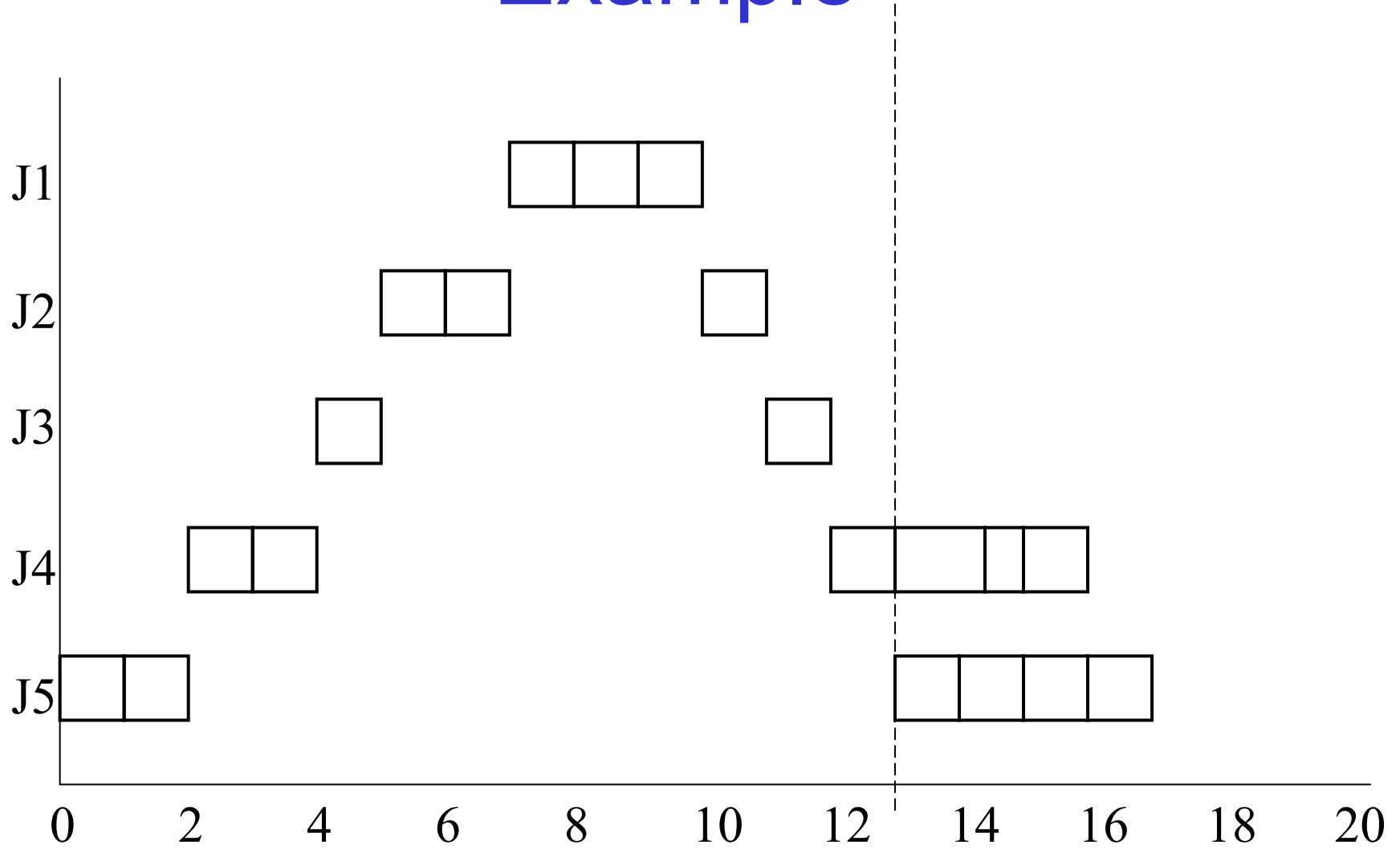
Example



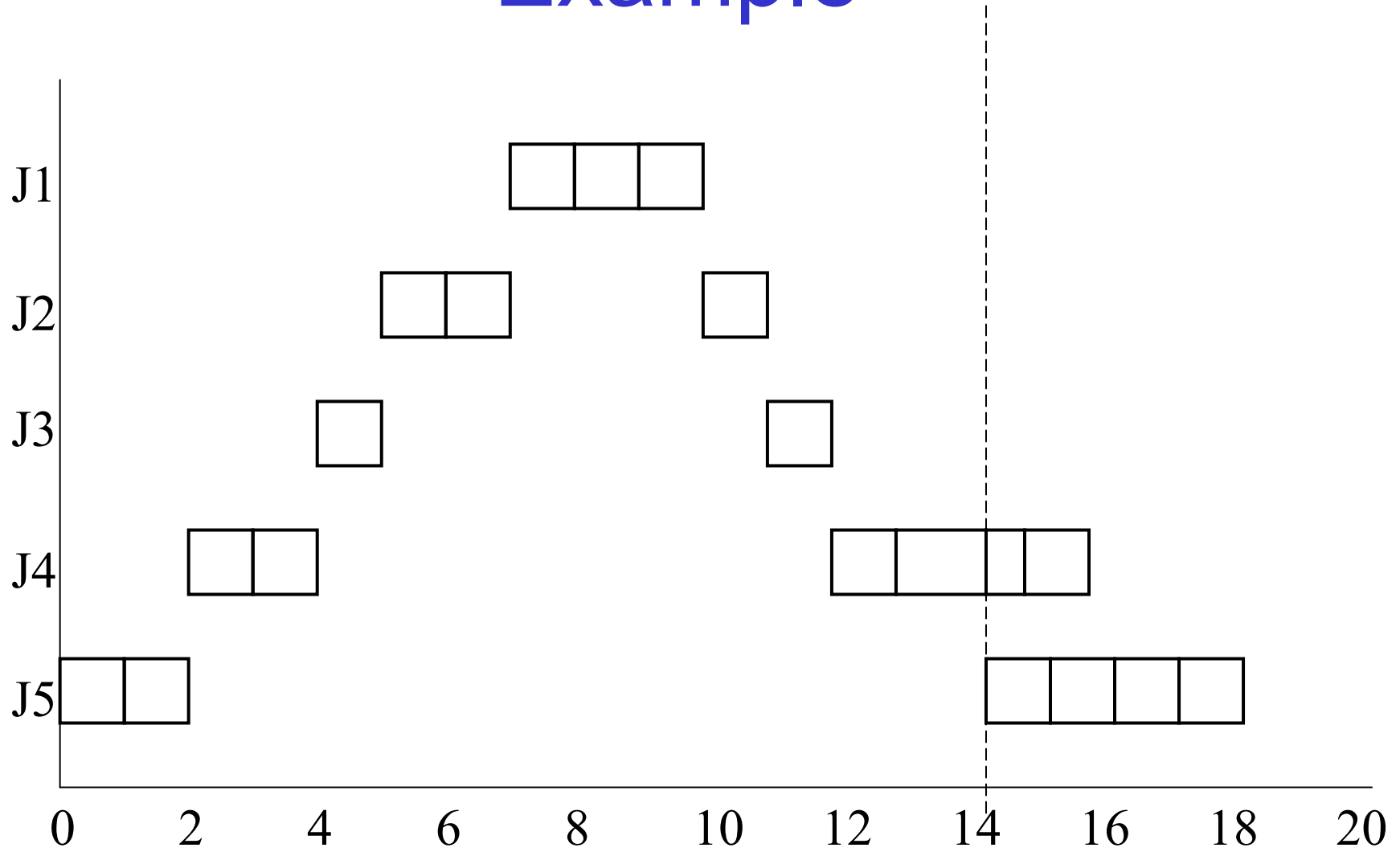
Example



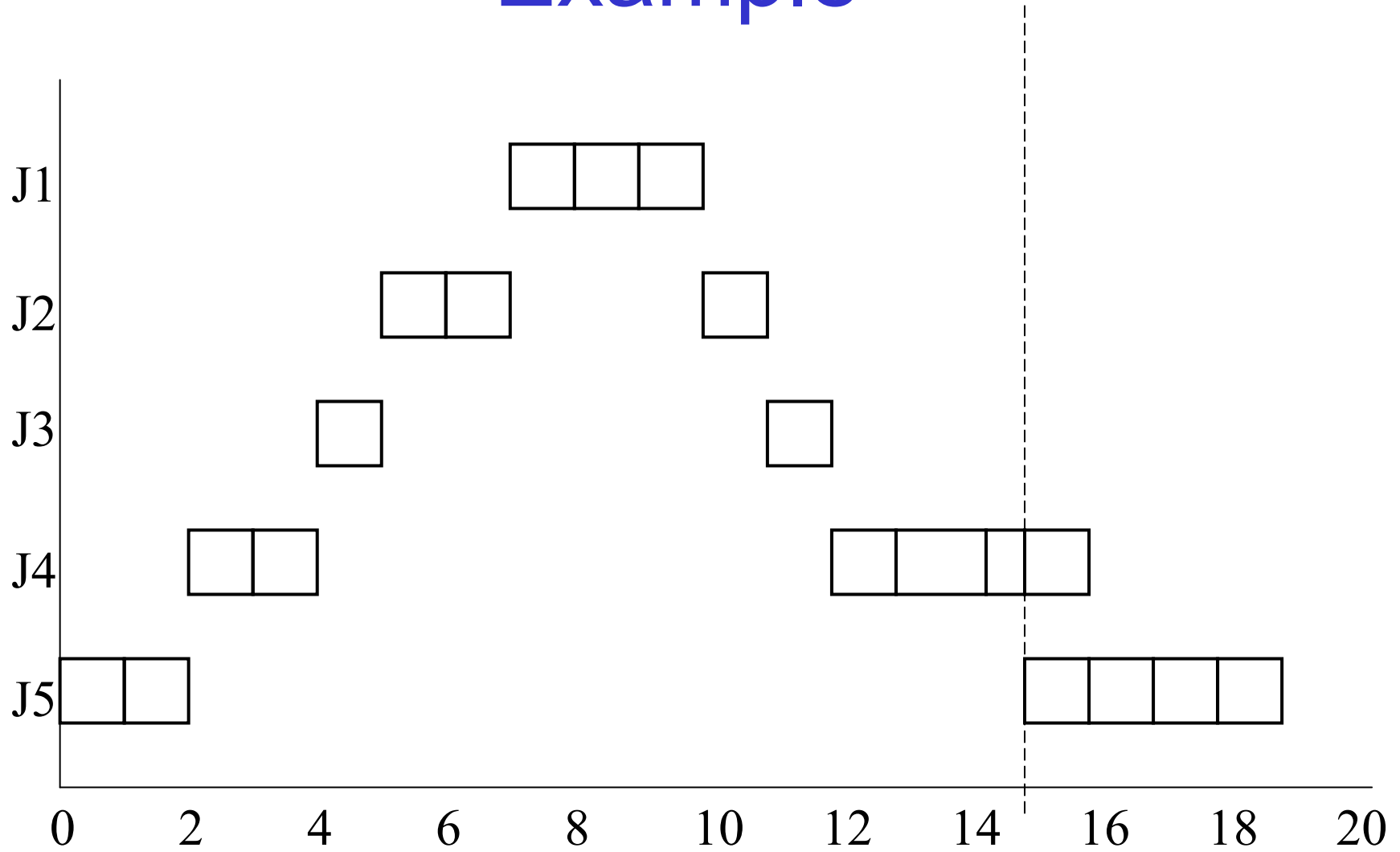
Example



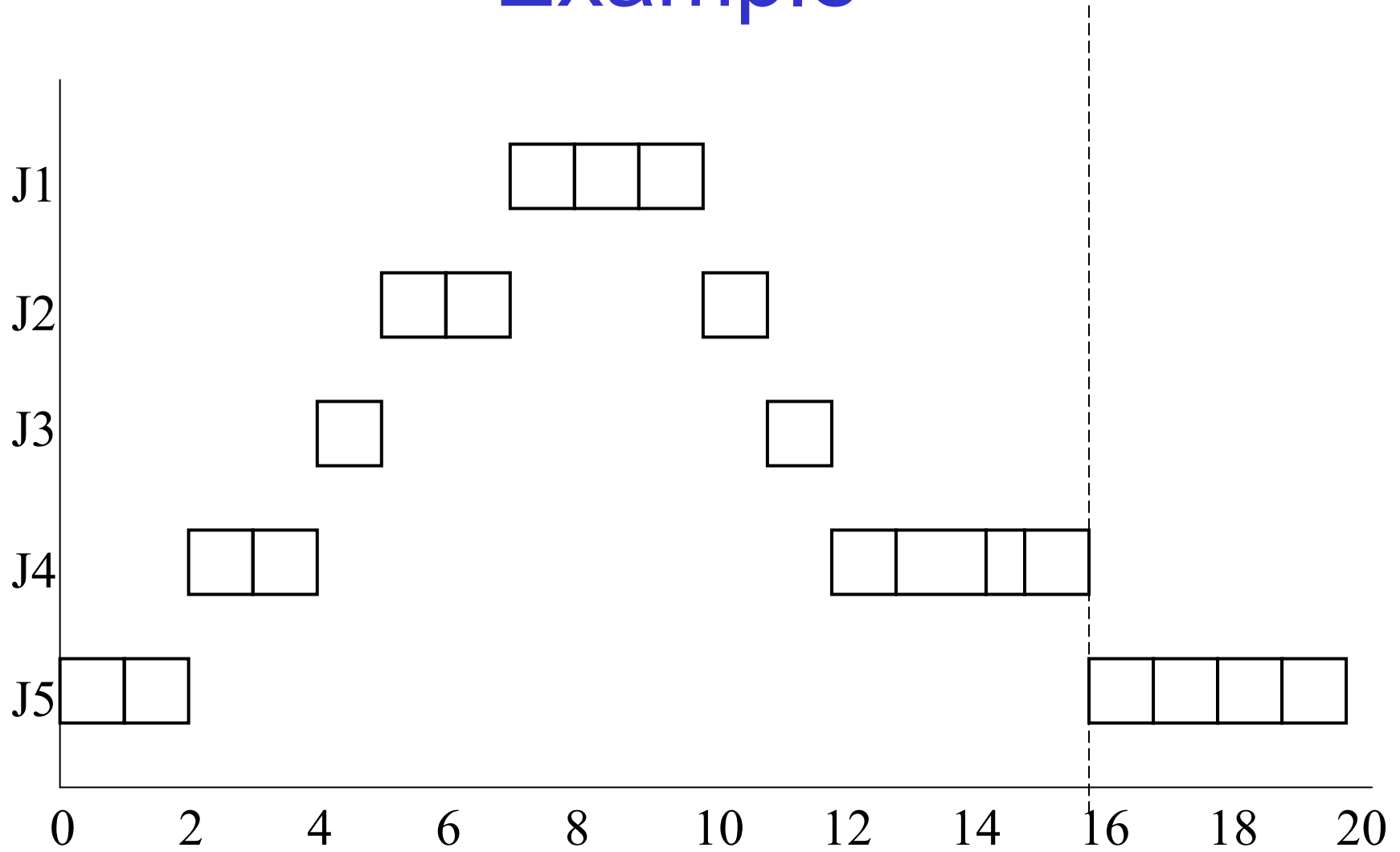
Example



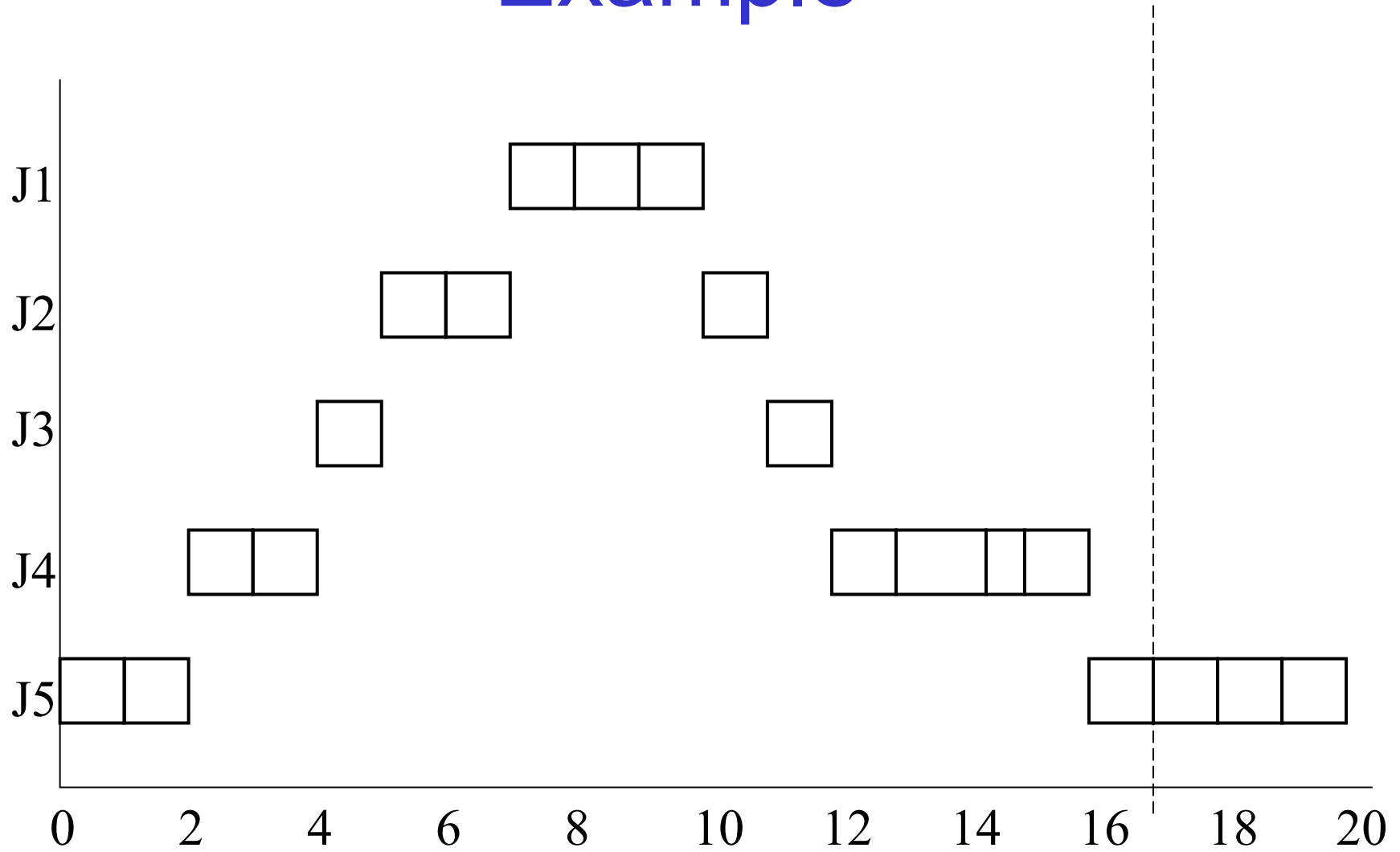
Example



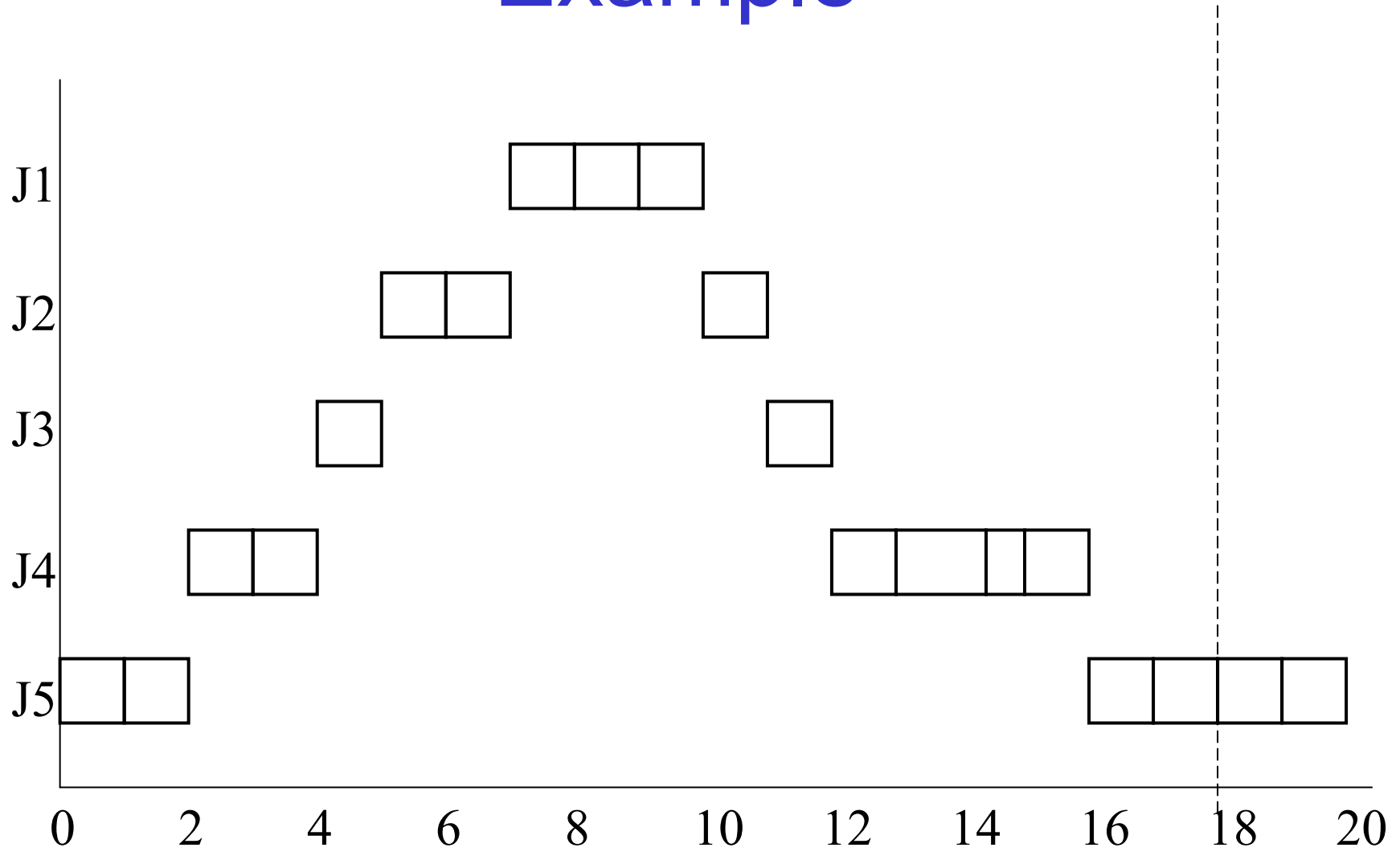
Example



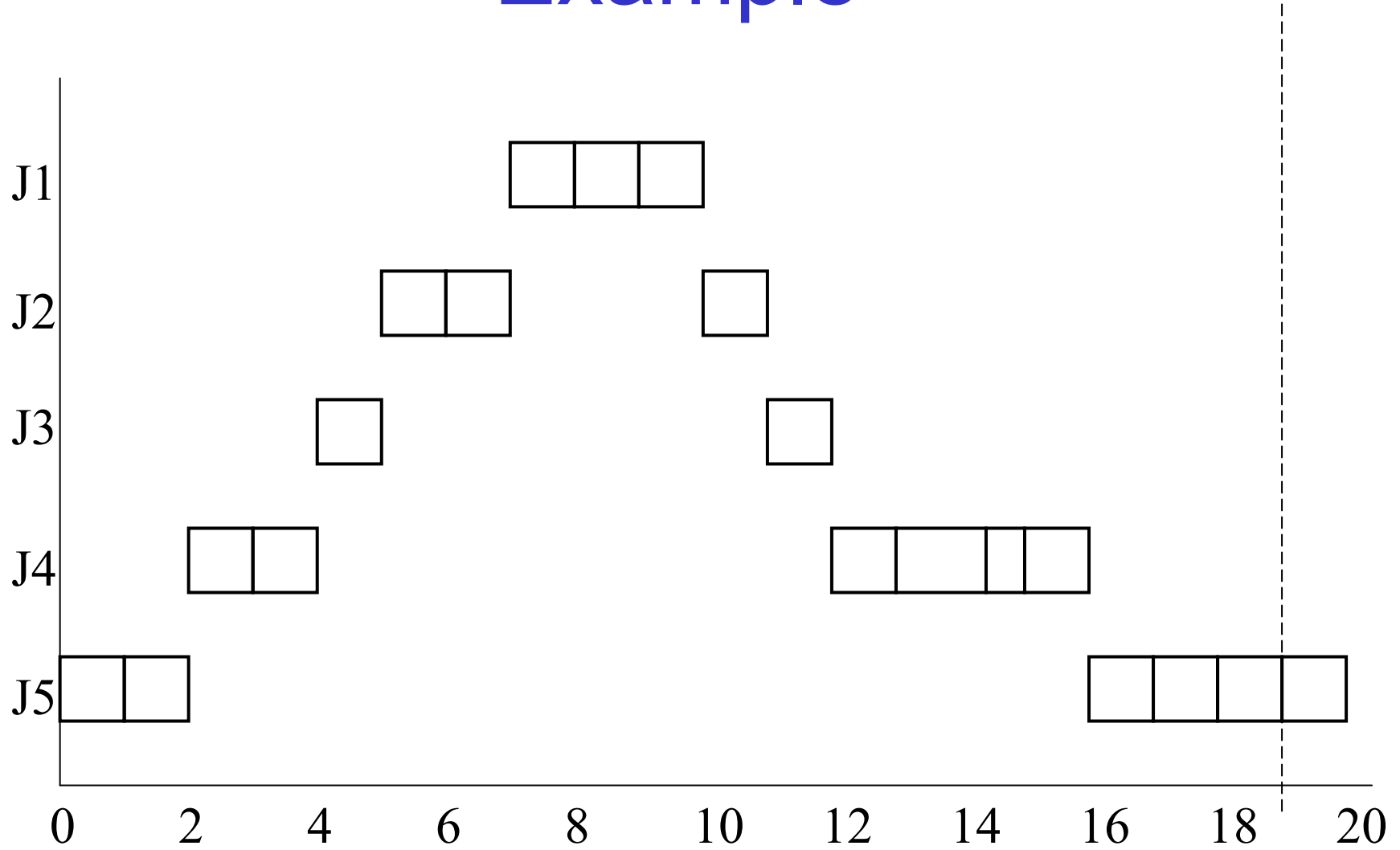
Example



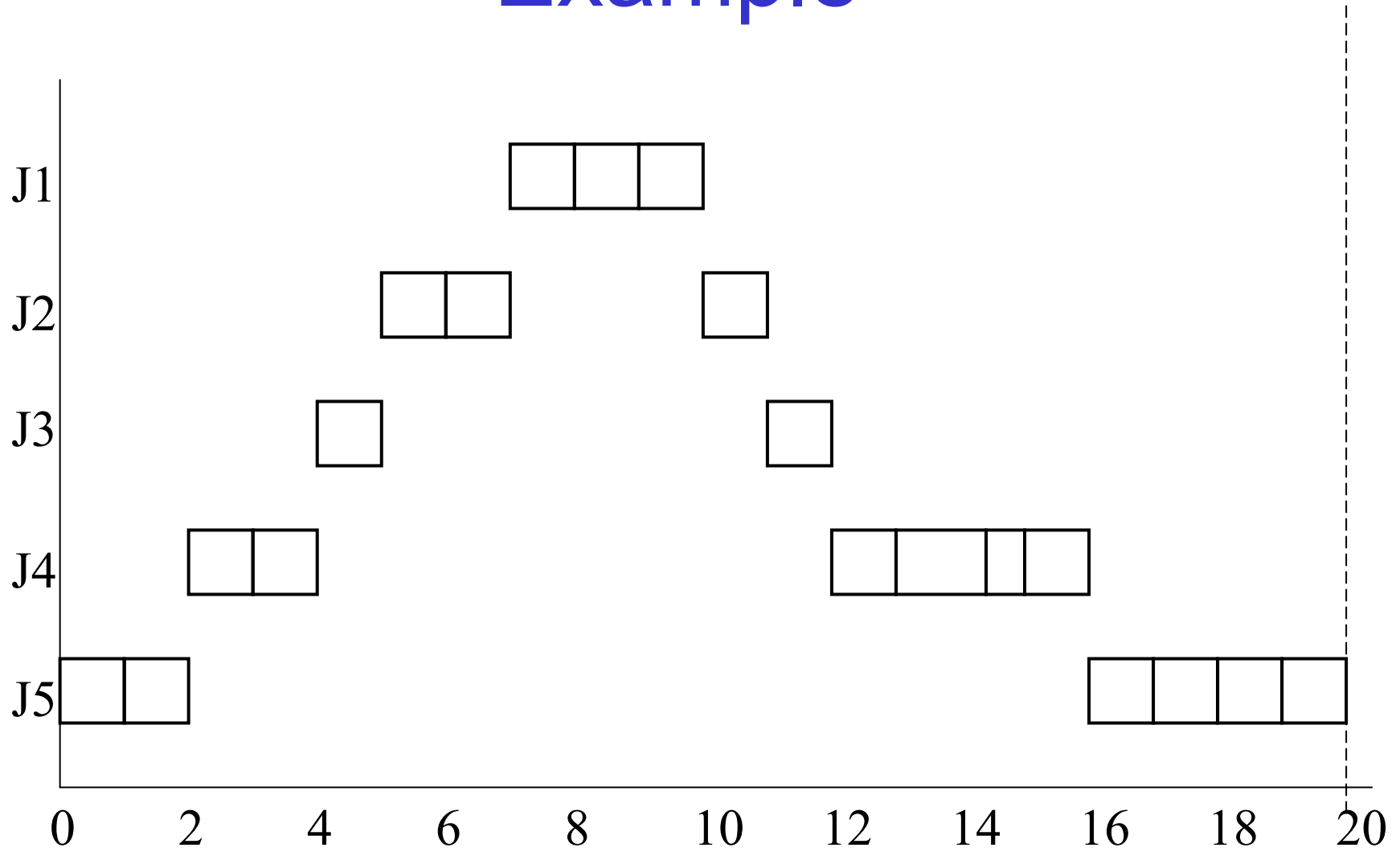
Example



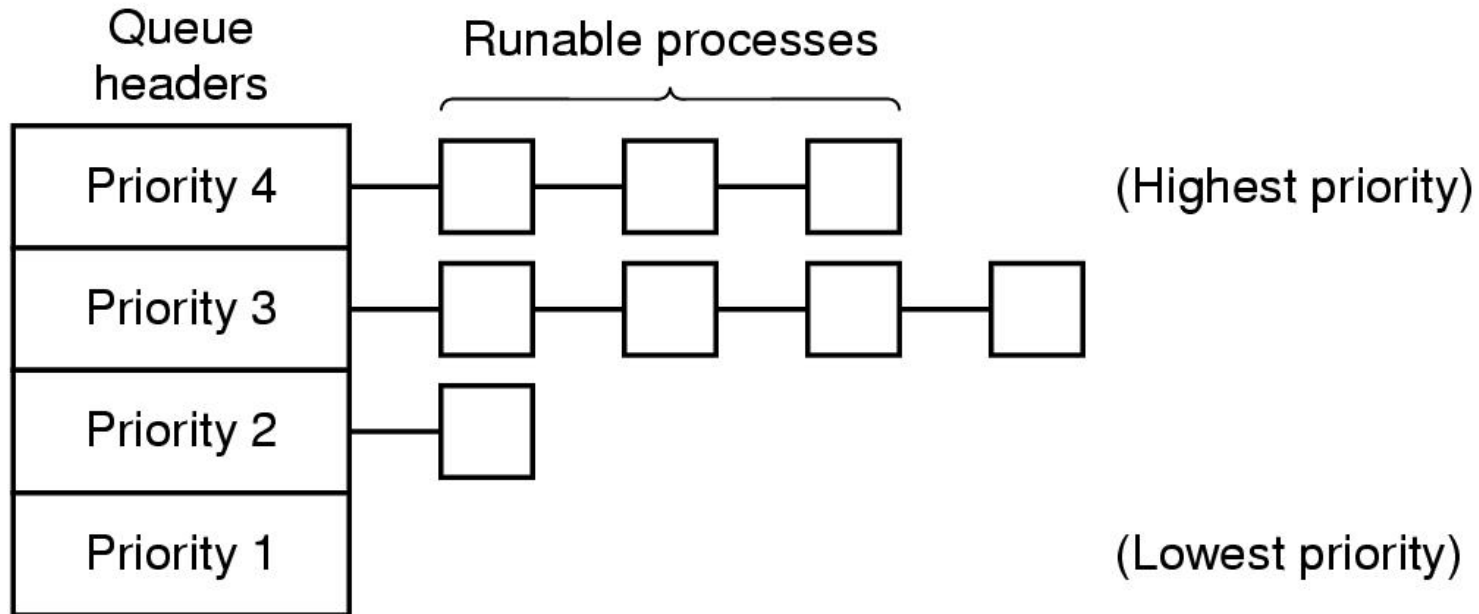
Example



Example



Priorities

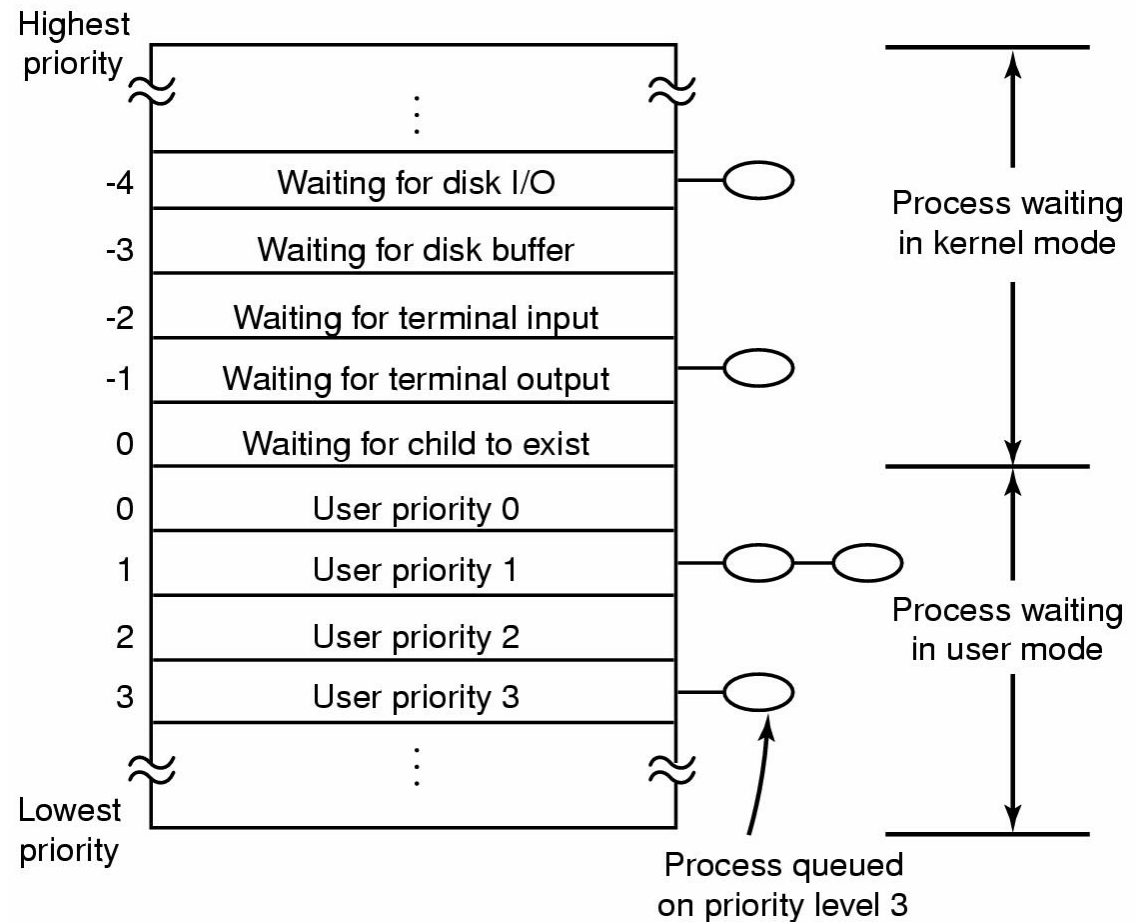


- Usually implemented by multiple priority queues, with round robin on each queue
- Con
 - Low priorities can starve
 - Need to adapt priorities periodically
 - Based on ageing or execution history



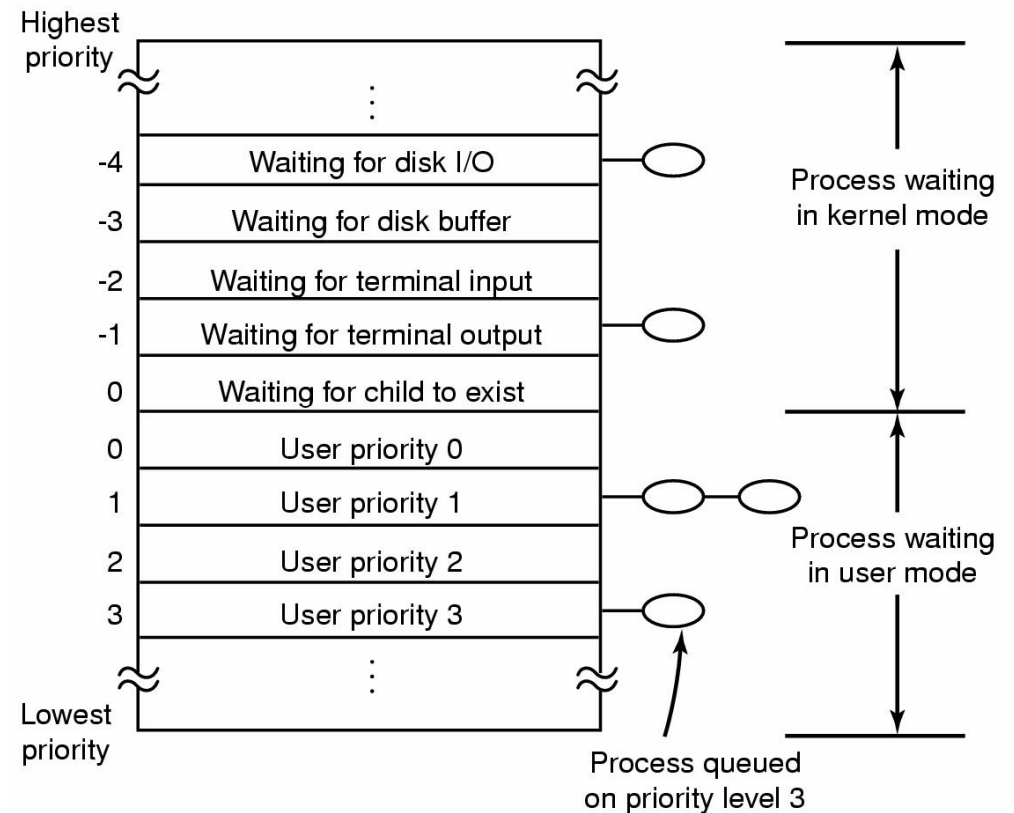
Traditional UNIX Scheduler

- 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



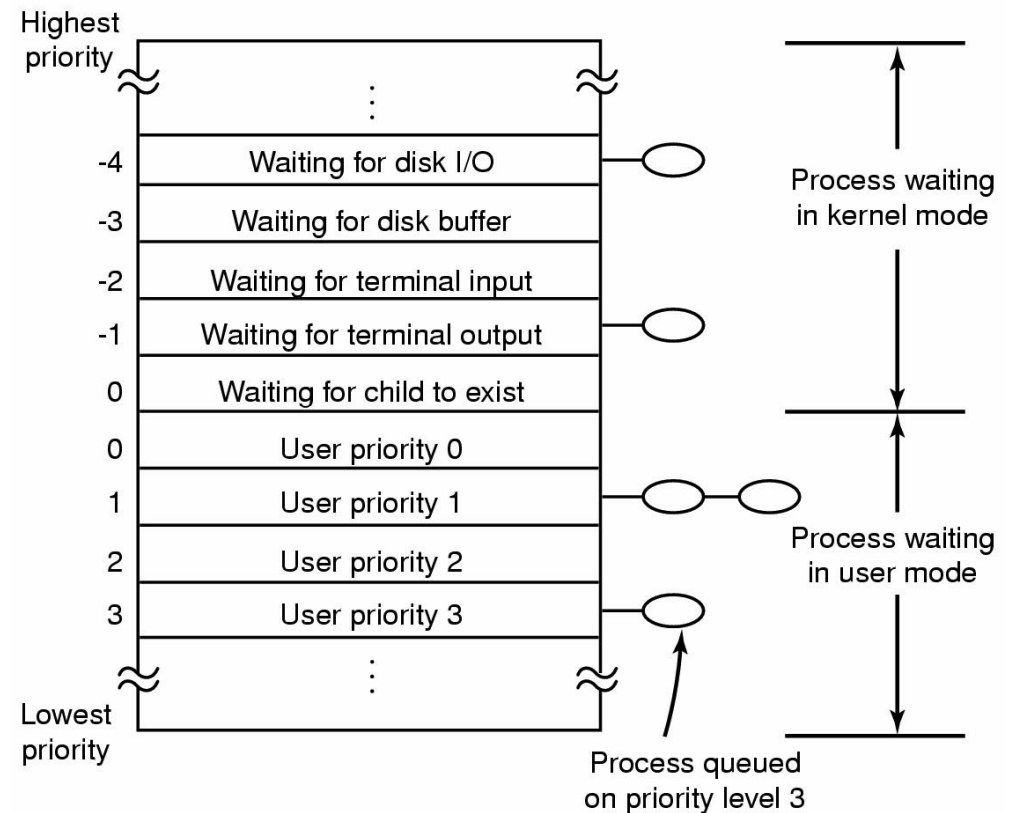
Traditional UNIX Scheduler

- 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
 - Penalise 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 a user to permanently boost or 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
 - Swapper, disk I/O, Character I/O



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
 - Starvation 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

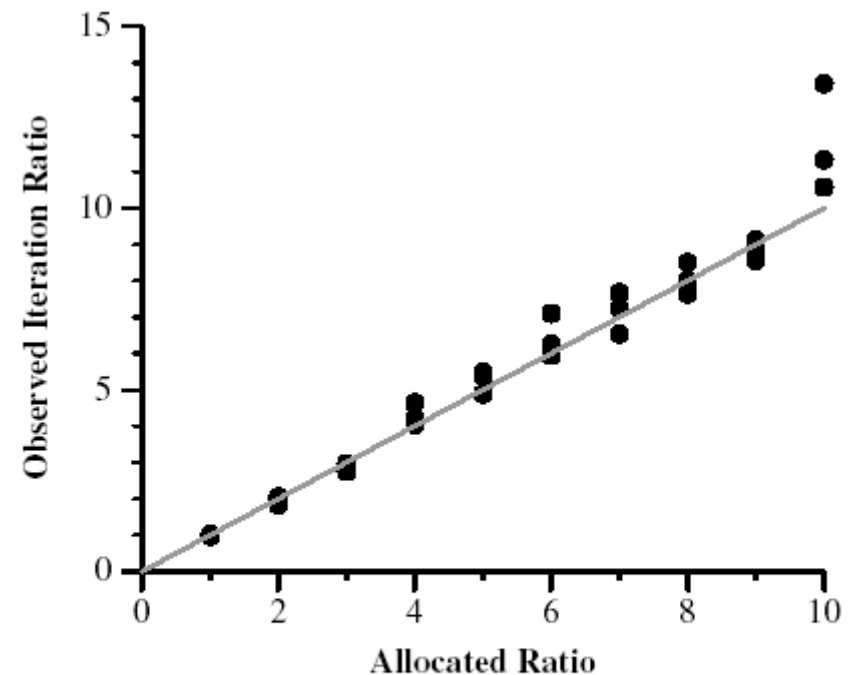


Figure 4: **Relative Rate Accuracy.** For each allocated ratio, the observed ratio is plotted for each of three 60 second runs. The gray line indicates the ideal where the two ratios are identical.



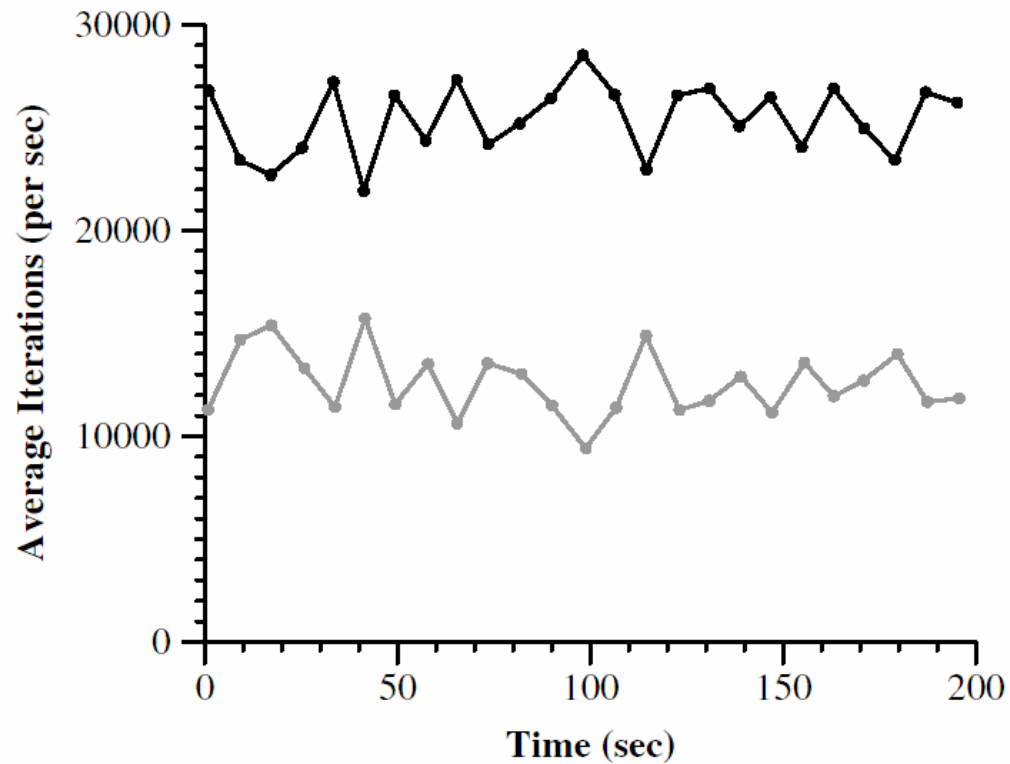


Figure 5: **Fairness Over Time.** Two tasks executing the Dhrystone benchmark with a 2 : 1 ticket allocation. Averaged over the entire run, the two tasks executed 25378 and 12619 iterations/sec., for an actual ratio of 2.01 : 1.



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

