scheduling

THE CRUX: HOW TO DEVELOP SCHEDULING POLICY

How should we develop a basic framework for thinking about scheduling policies? What are the key assumptions? What metrics are important? What basic approaches have been used in the earliest of computer systems?


“Well, I found a solution to your problem with the chickens. But, the solution only works for spherical chickens in the vacuum and with a uniform mass distribution…”
https://nossotradeoff.wordpress.com/2013/02/14/spherical-chickens-in-the-vacum/

Initial (simplifying) assumptions

1. Each job runs for the same amount of time.
2. All jobs arrive at the same time.
3. Once started, each job runs to completion.
4. All jobs only use the CPU (i.e., they perform no I/O)
5. The run-time of each job is known.

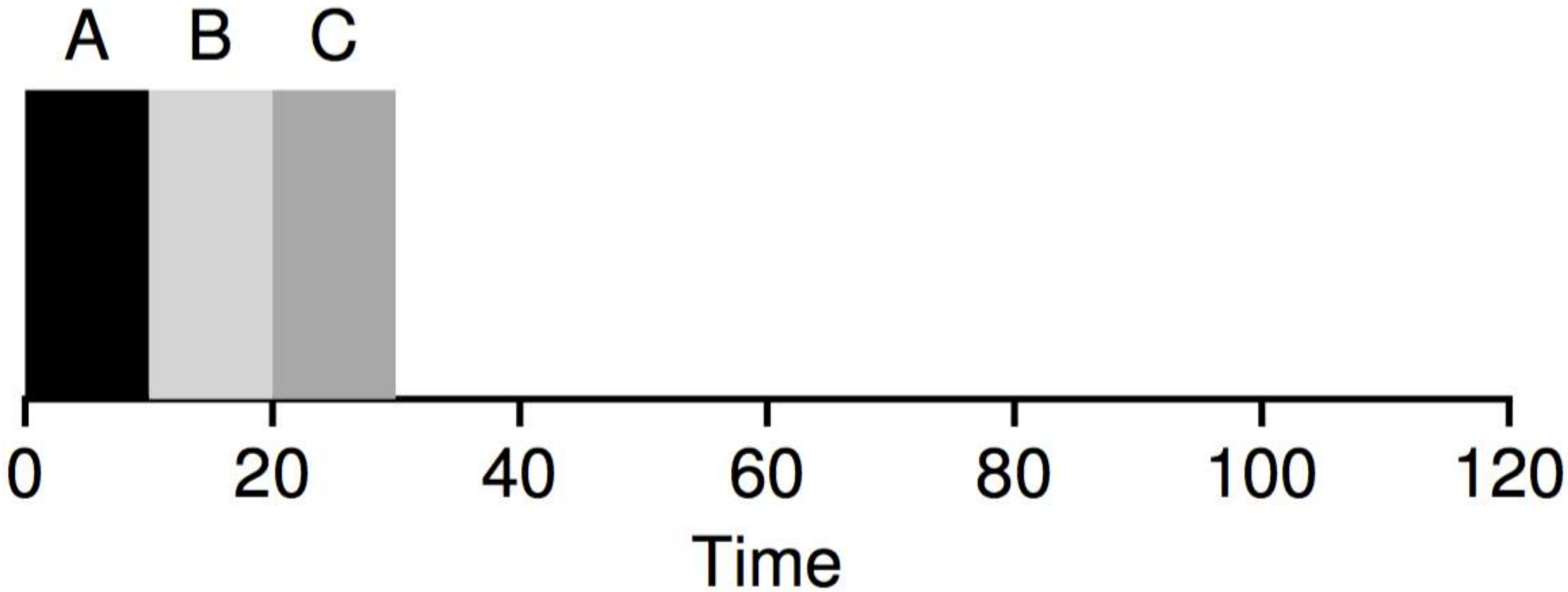


Figure 7.1: **FIFO Simple Example**

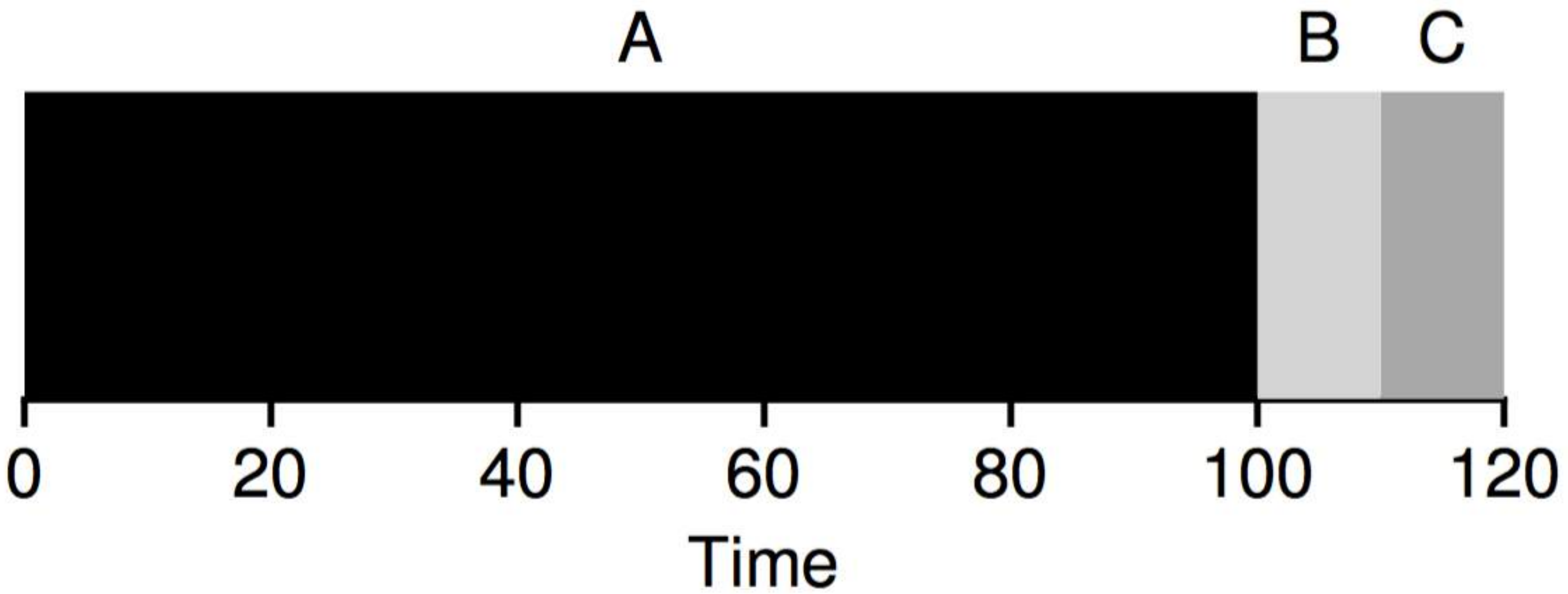


Figure 7.2: **Why FIFO Is Not That Great**

Shortest Job First

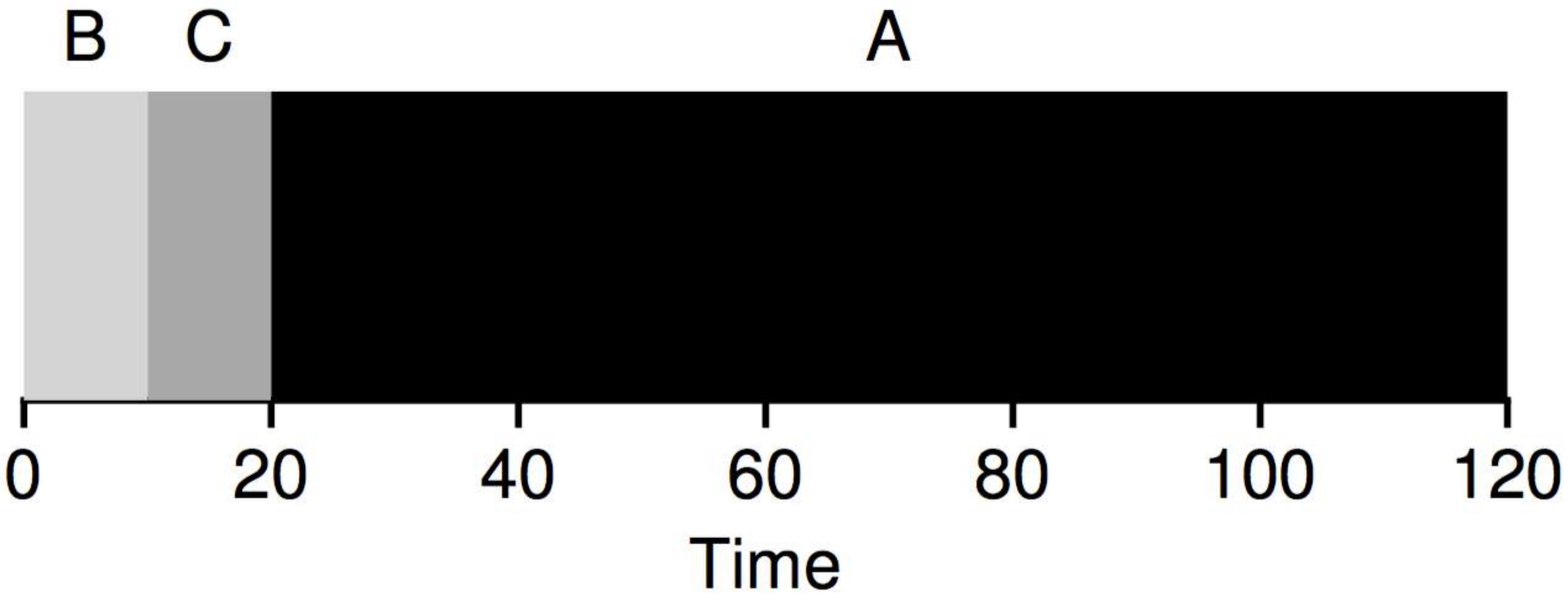


Figure 7.3: **SJF Simple Example**

Initial (simplifying) assumptions

1. Each job runs for the same amount of time.
2. ~~All jobs arrive at the same time.~~
3. Once started, each job runs to completion.
4. All jobs only use the CPU (i.e., they perform no I/O)
5. The run-time of each job is known.

Shortest Job First: different arrival times

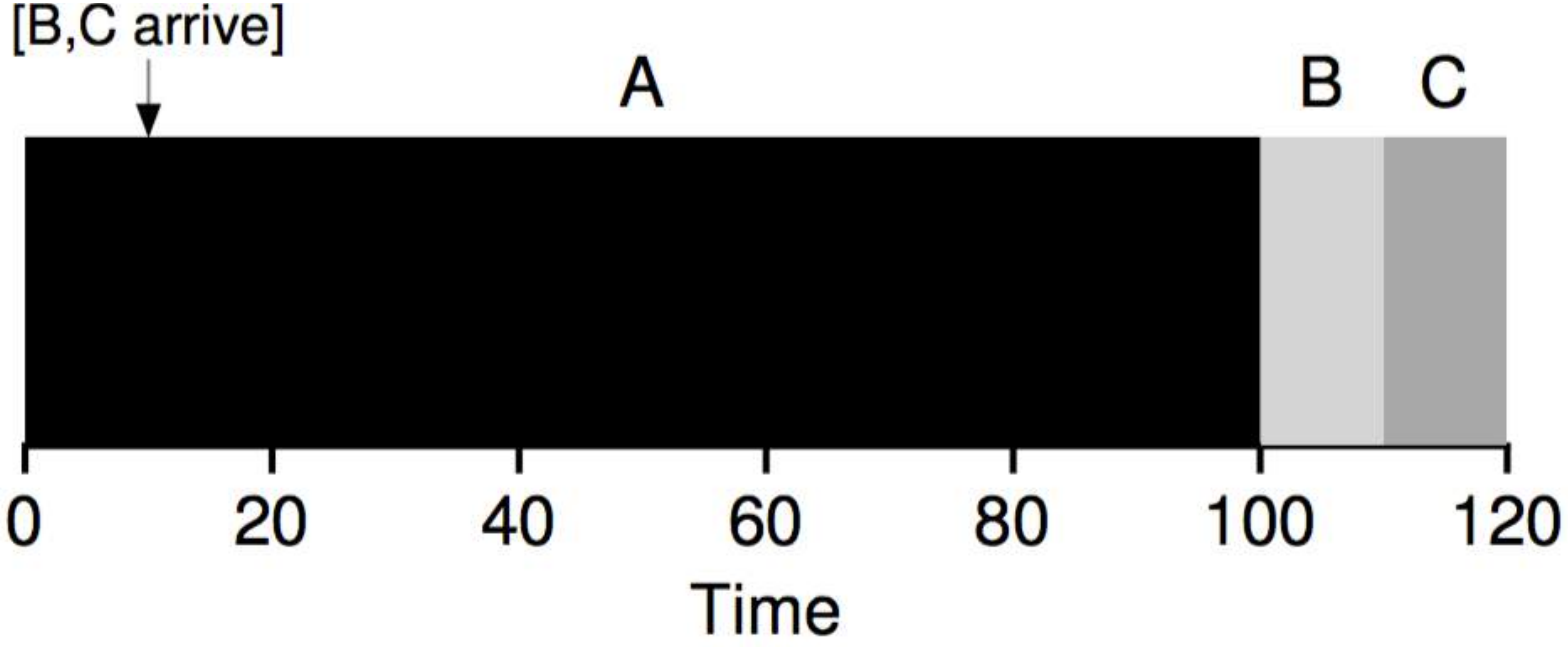


Figure 7.4: **SJF With Late Arrivals From B and C**

Initial (simplifying) assumptions

1. Each job runs for the same amount of time.
2. ~~All jobs arrive at the same time.~~
3. ~~Once started, each job runs to completion.~~
4. All jobs only use the CPU (i.e., they perform no I/O)
5. The run-time of each job is known.

Shortest Time to Completion First

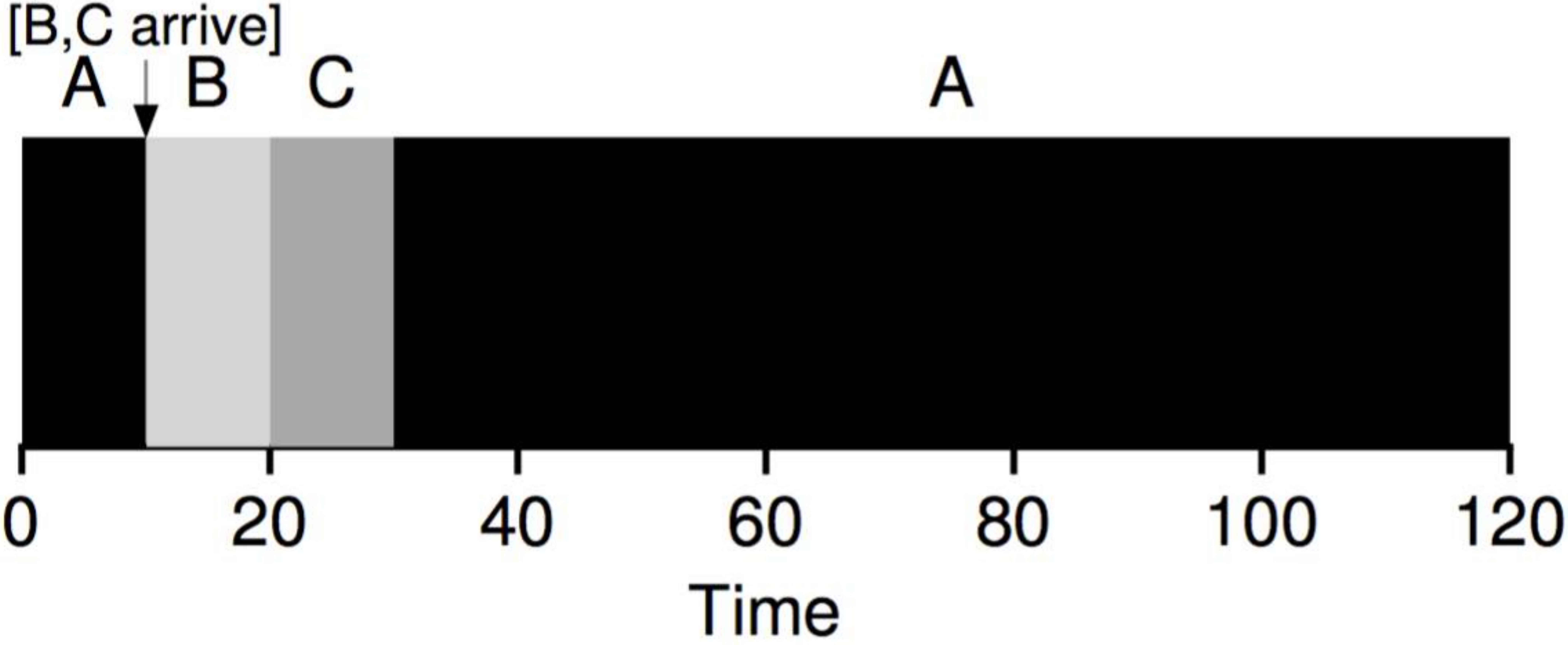
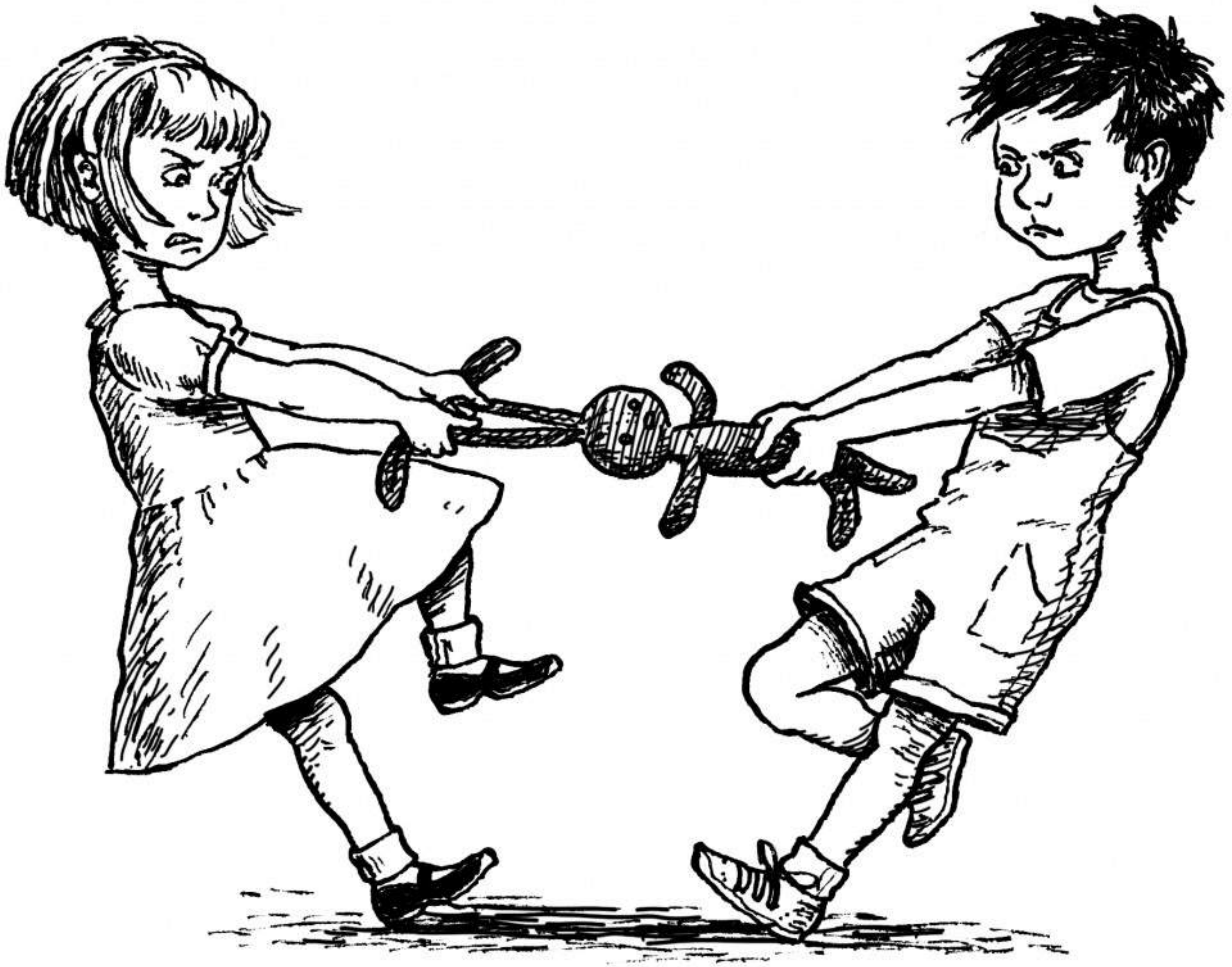


Figure 7.5: **STCF Simple Example**

Time sharing and interactive systems



http://www.heathershumaker.com/blog/wp-content/uploads/2013/09/rule10_final-Conflict-1024x810.jpg

Metric for Interactive systems:

$$T_{response} = T_{firstrun} - T_{arrival}$$

Interactive systems

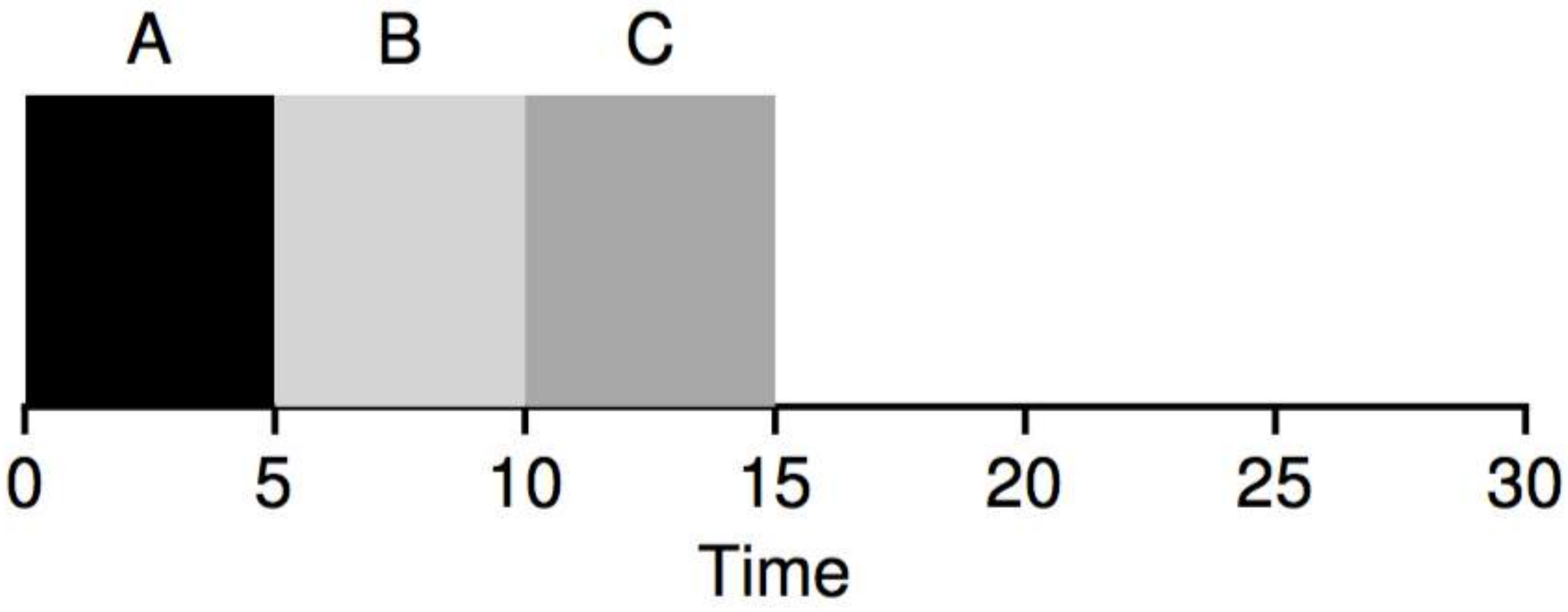


Figure 7.6: **SJF Again (Bad for Response Time)**

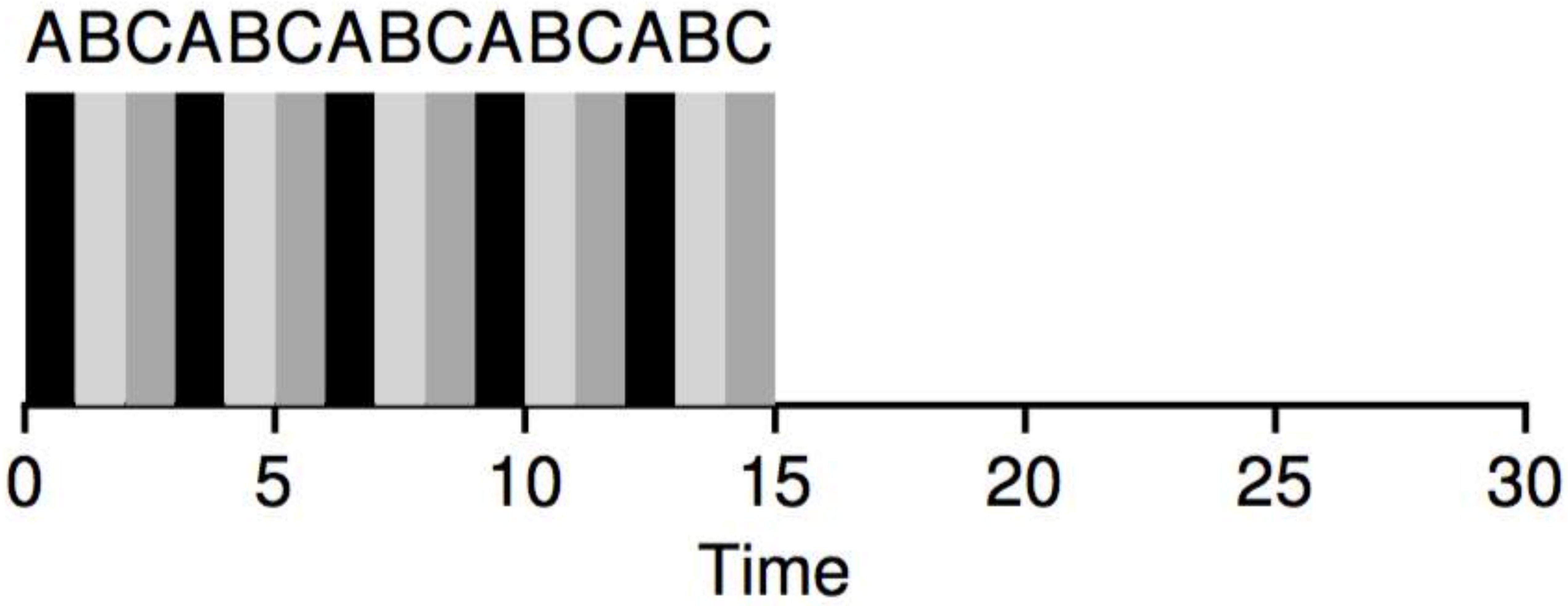


Figure 7.7: **Round Robin (Good for Response Time)**

Incorporating I/O

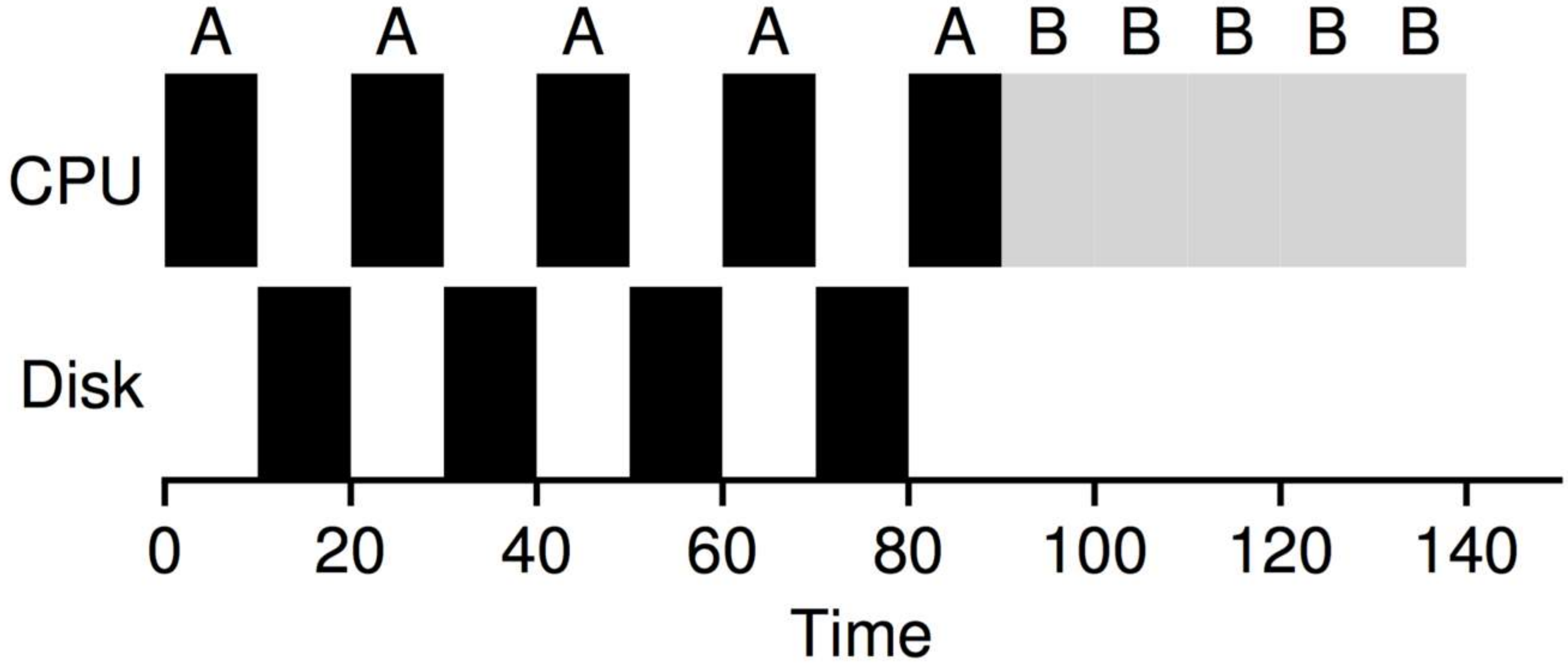


Figure 7.8: **Poor Use of Resources**

Overlap

Tip: Overlap Enables Higher Utilization

When possible, **overlap** operations to maximize the utilization of systems. Overlap is useful in many different domains, including when performing disk I/O or sending messages to remote machines; in either case, starting the operation and then switching to other work is a good idea, and improves the overall utilization and efficiency of the system.

Initial (simplifying) assumptions

1. Each job runs for the same amount of time.
2. ~~All jobs arrive at the same time.~~
3. ~~Once started, each job runs to completion.~~
4. ~~All jobs only use the CPU (i.e., they perform no I/O)~~
5. The run-time of each job is known.

Overlap

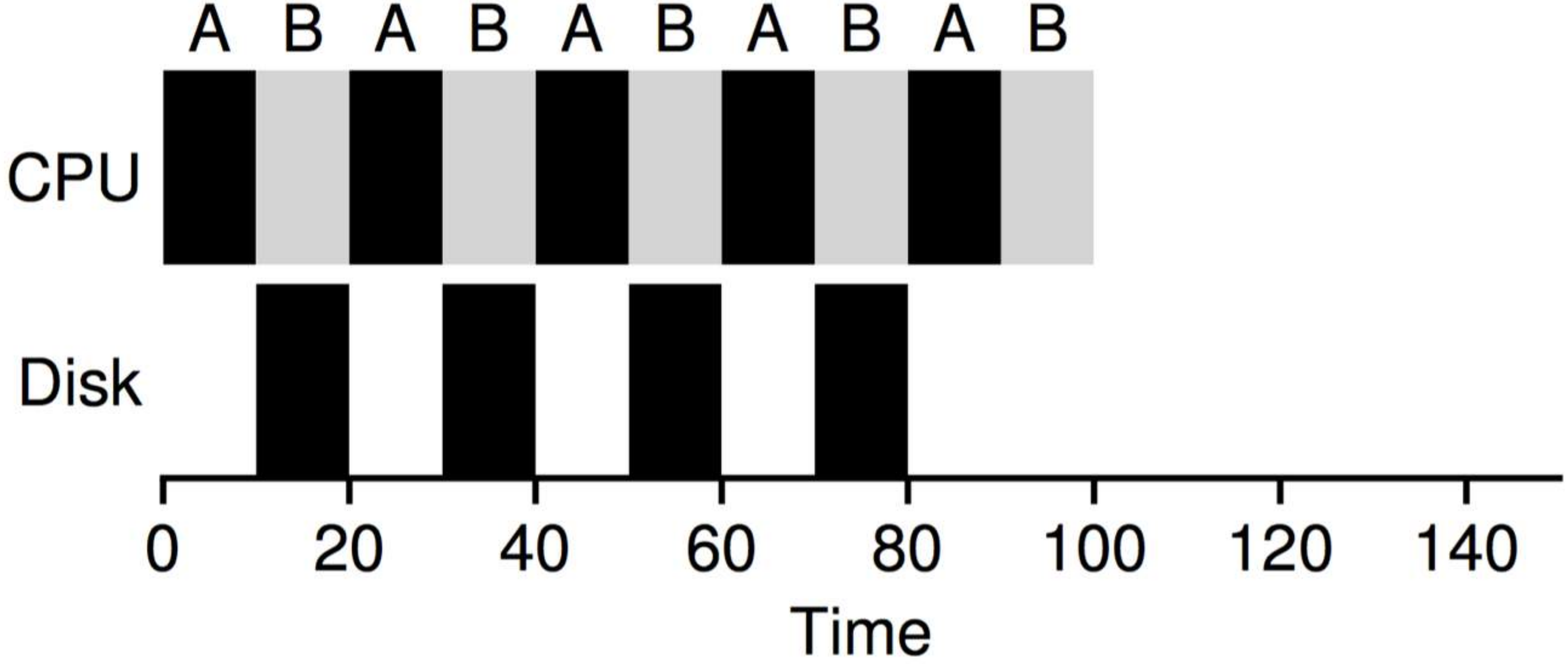
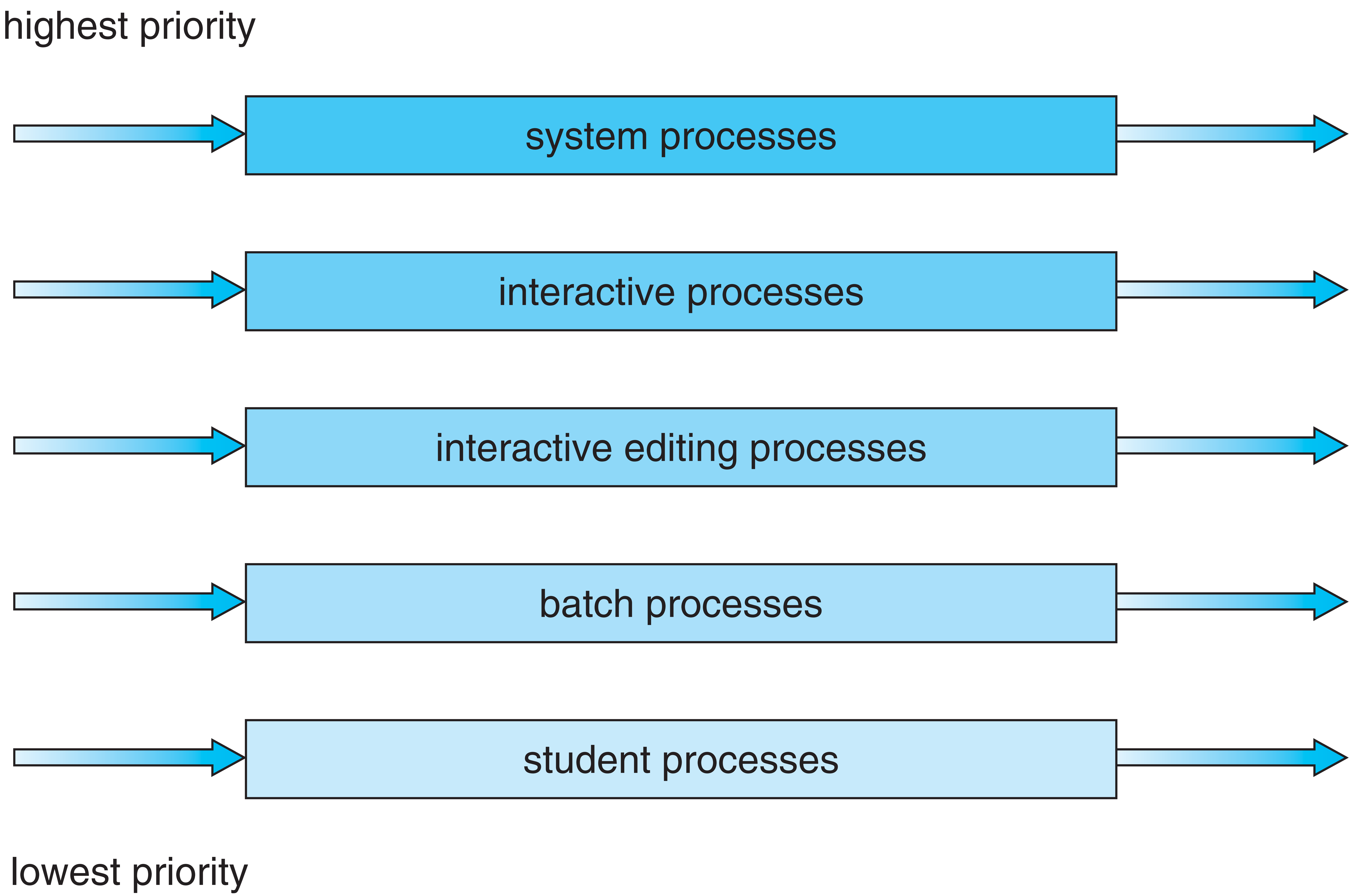


Figure 7.9: **Overlap Allows Better Use of Resources**

Multi-level queue scheduling



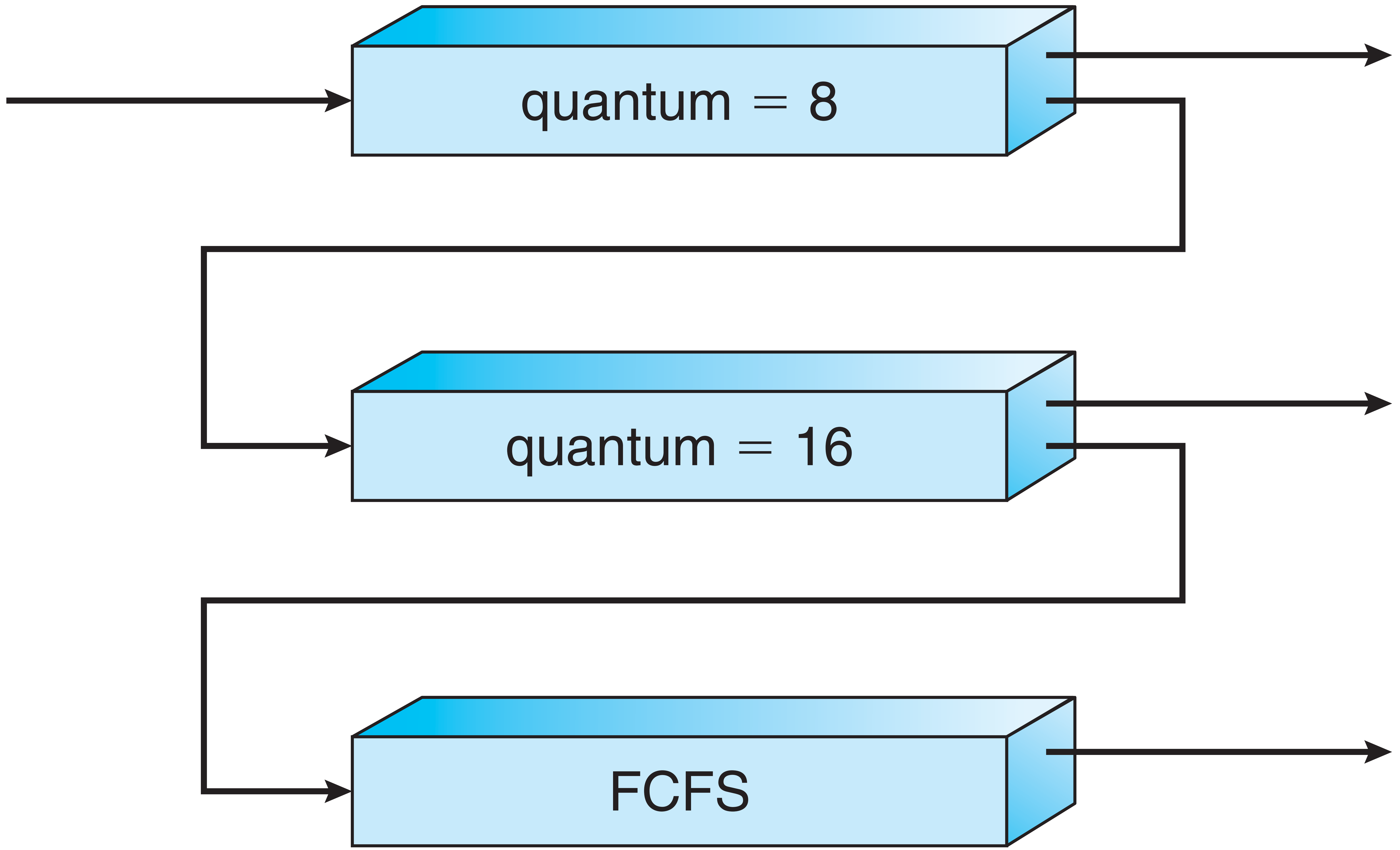
Initial (simplifying) assumptions

1. ~~Each job runs for the same amount of time.~~
2. ~~All jobs arrive at the same time.~~
3. ~~Once started, each job runs to completion.~~
4. ~~All jobs only use the CPU (i.e., they perform no I/O)~~
5. The run-time of each job is known.

The OS can’t see into future…

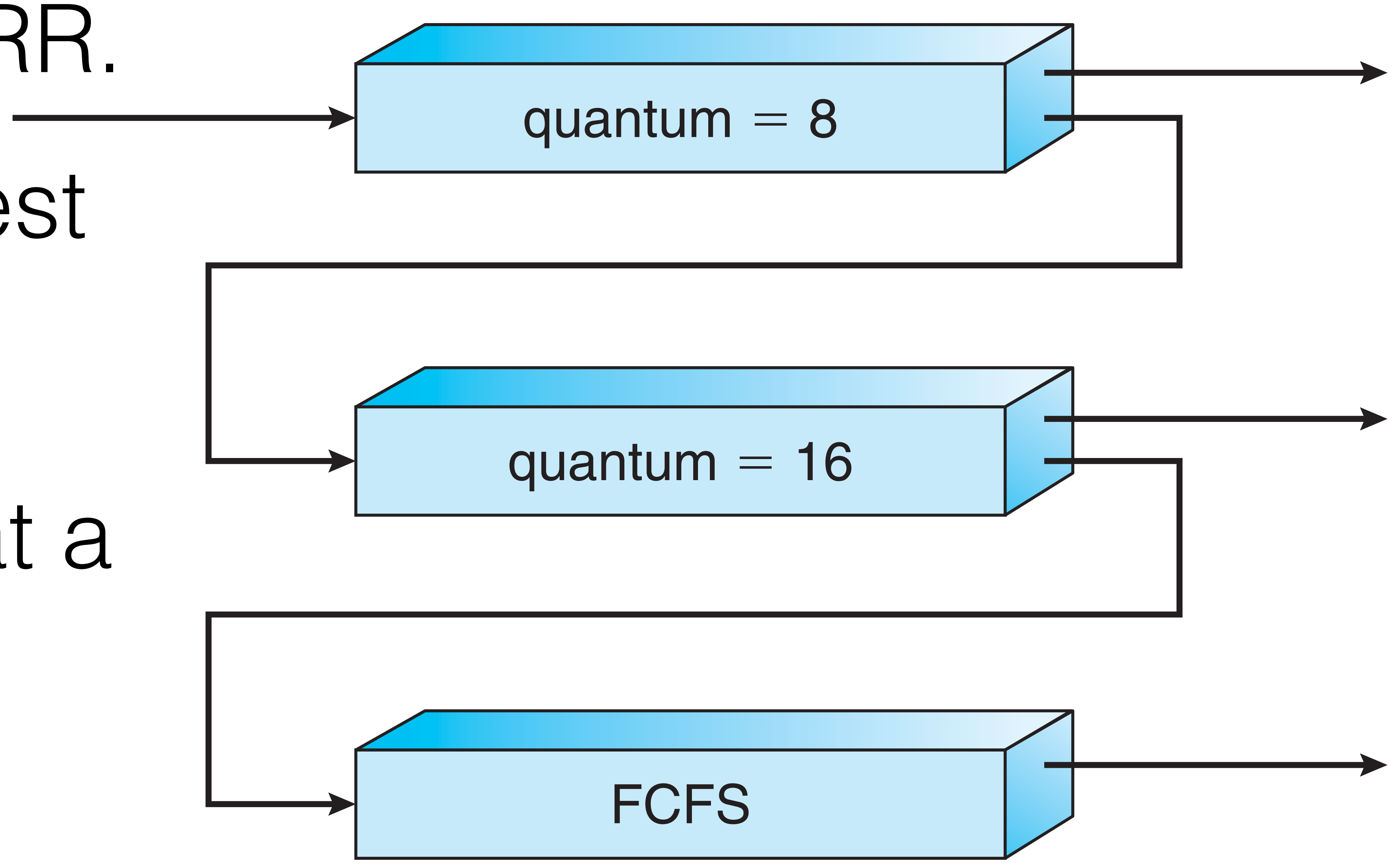
Multi-level feedback queue

Multi-level feedback queue scheduling



Multi-level feedback queue scheduling

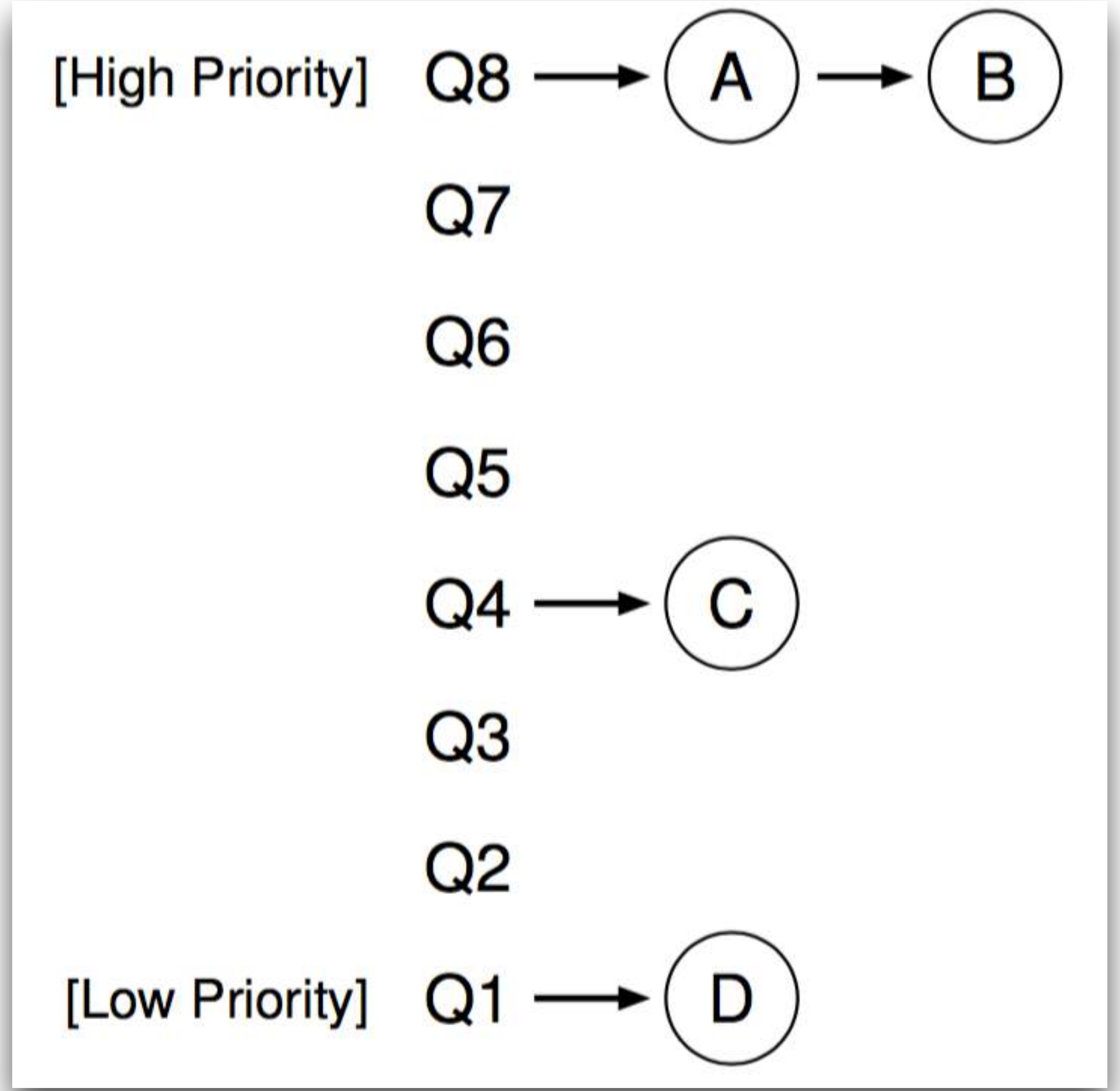
- **Rule 1**: If Priority(A) > Priority(B), A runs (B doesn't).
- **Rule 2**: If Priority(A) = Priority(B), A & B run in RR.
- **Rule 3**: All jobs enter the system with the highest priority (the topmost queue).
- **Rule 4**: Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
- **Rule 5**: After some time period S, move all the jobs in the system to the topmost queue.



http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-mlfq.pdf

Multi-level feedback queue scheduling

Example



Linux scheduling

Prior to Version 2.5:

- Linux used a variation of the traditional UNIX scheduling algorithm.
- Did not have good support for multiple processors.
- Had poor performance for systems with many runnable processes.

Linux scheduling

Version 2.5:

- Presented a new scheduling algorithm: O(1)
- O(1) ran in constant time regardless the number of runnable processes.
- Provided support for SMP systems including load balancing and processor affinity.
- It worked great for SMP systems. But, it wasn’t very good for interactive systems (e.g., Desktop systems) because of slow response times.

Linux scheduling

Version 2.6:

- Scheduler was revised again: *Completely Fair Scheduler* (CFS).
- CFS became the default linux scheduler.

Completely Fair Scheduler (CFS)

- Scheduling based on scheduling classes.
  - Each class has a priority.
  - Different classes allow for different scheduling algorithms depending on the system needs.
    - Example: Scheduling criteria for server systems can be different from criteria for mobile devices.

Completely Fair Scheduler (CFS): red-black tree

- A task is added to the tree when it becomes runnable.
- A task is removed from the tree when it is not runnable.
- Tasks that are given less processing time are on the left. Tasks that are given more time are on the right.