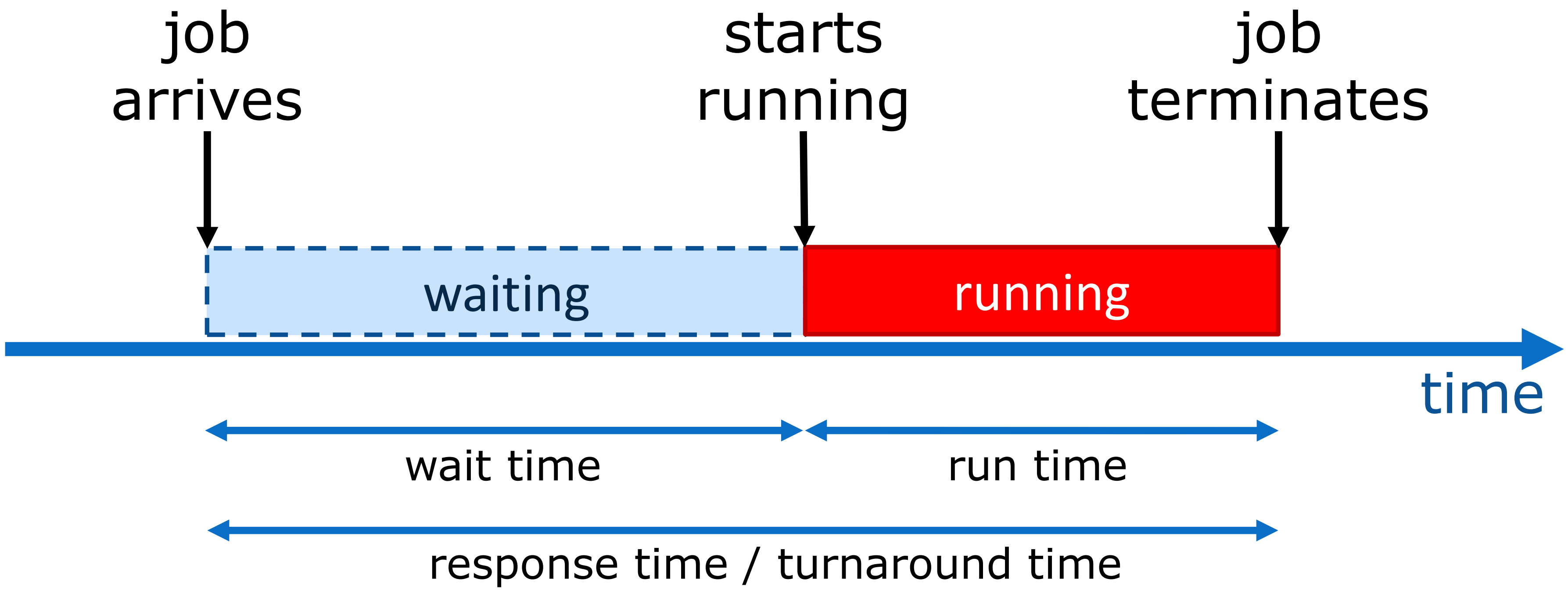
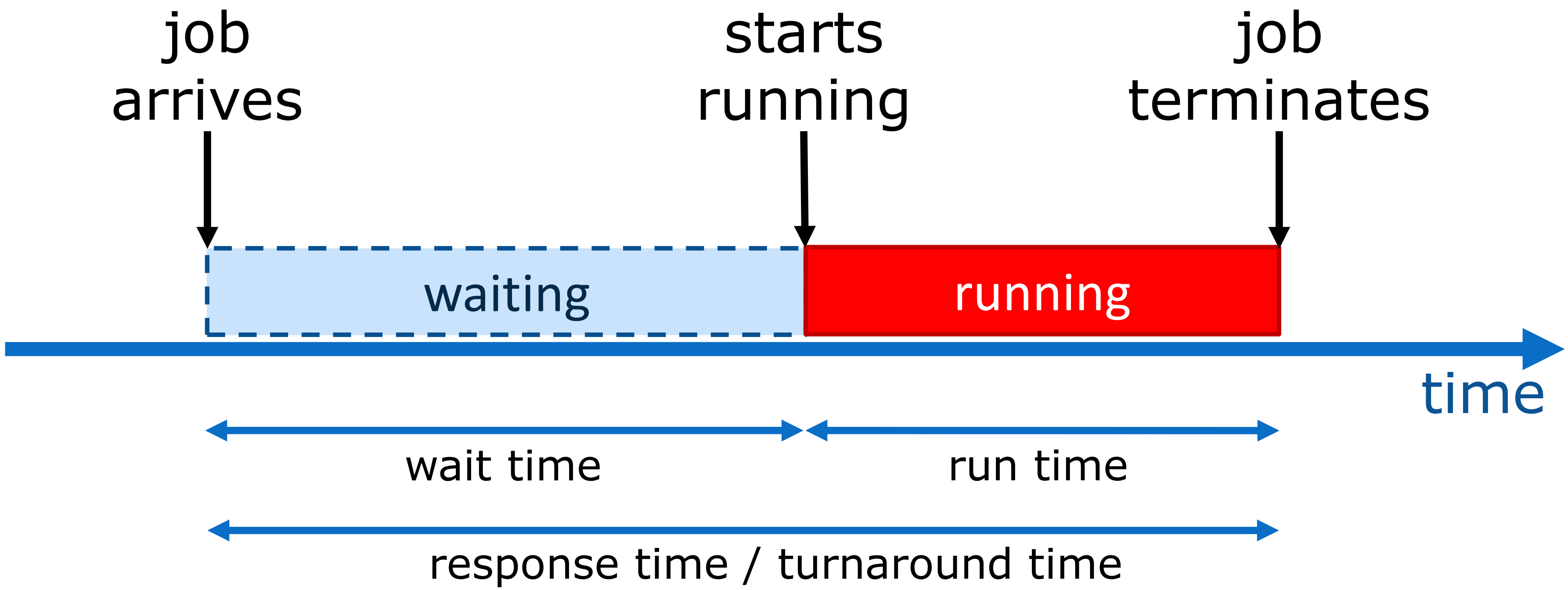
Operating Systems
Process/Thread Scheduling II

David Hay
Dror Feitelson

Job Timeline



Job Timeline



- Interactive (reactive) programs have multiple CPU bursts — one after each event
- Or perform I/O operations separated by CPU bursts
- We (so far) treat these bursts as independent jobs

Offline Schedulers

Jobs:

Job	Runtime
P1	17
P2	2
P3	3

FCFS:

Suffers from "convoy effect"

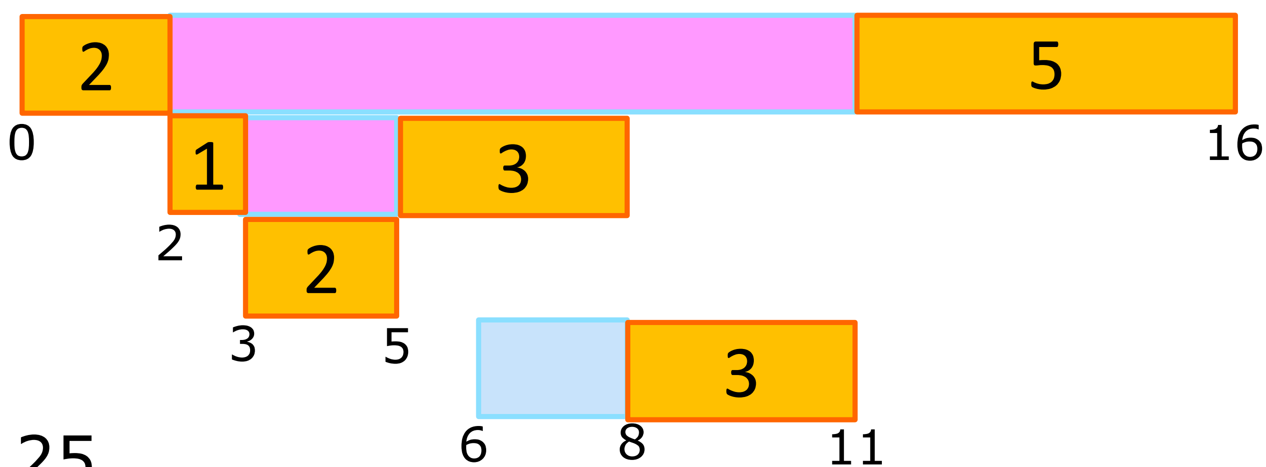
SJF:

Optimal (minimal) average wait time

Shortest Remaining Processing Time (SRPT)

Schedule:

Job	Arrival	Runtime	Wait	Start	End	Response
P1	0	7	(9)	0	16	16
P2	2	4	(2)	2	8	6
P3	3	2	0	3	5	2
P4	6	3	2	8	11	5

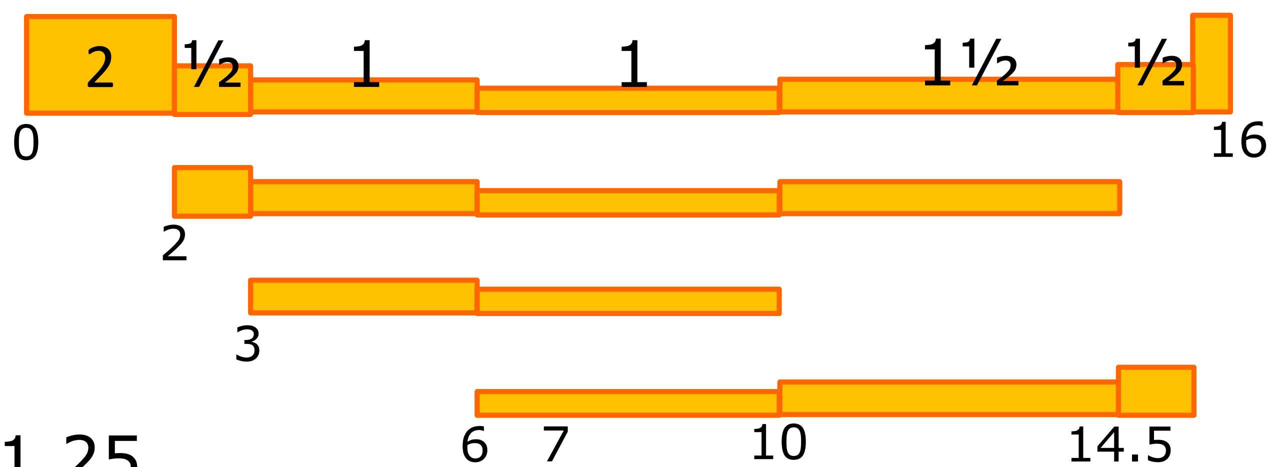


Avg. wait: 3.25
Avg. response: 7.25
Throughput: 0.25

Processor Sharing (PS)

Schedule:

Job	Arrival	Runtime	Wait	Start	End	Response
P1	0	7	0	0	16	16
P2	2	4	0	2	14.5	12.5
P3	3	2	0	3	10	7
P4	6	3	0	6	15.5	9.5

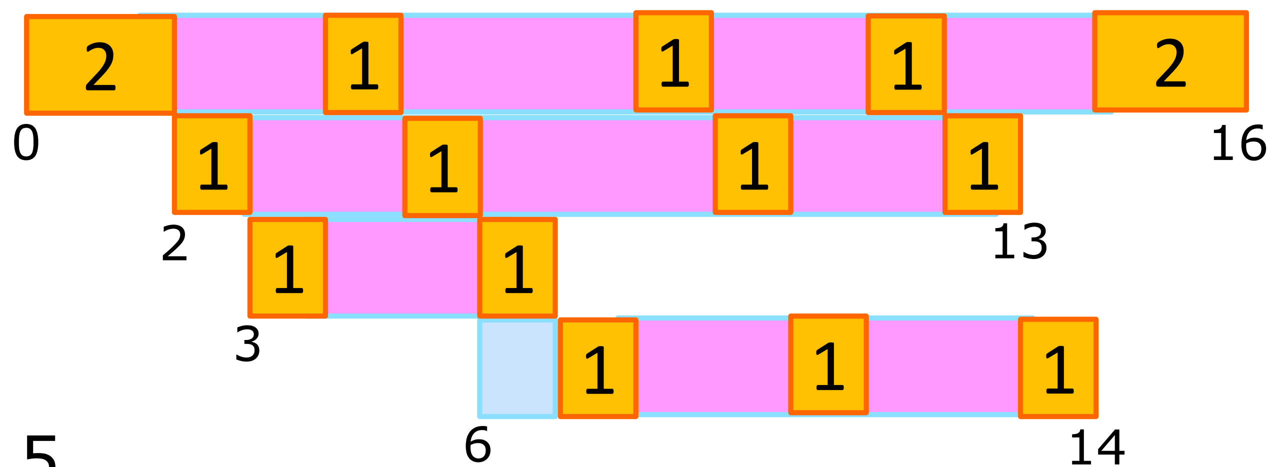


Avg. wait: undef
Avg. response: 11.25
Throughput: 0.25

Round-Robin (RR) Scheduling

Schedule:

Job	Arrival	Runtime	Wait	Start	End	Response
P1	0	7	(9)	0	16	16
P2	2	4	(7)	2	13	11
P3	3	2	(2)	6	7	4
P4	6	3	(5)	7	14	7



Avg. wait: 5.75
Avg. response: 9.5
Throughput: 0.25

RR Notes

- RR works in an **online** setting
- RR uses preemption to cope with **lack of knowledge**
  - Will additional jobs arrive?
  - How long will jobs run?
- RR gives **uniform treatment** to all jobs

- **Can we do better?**

End of previous lecture

USING ACCOUNTING DATA

Learning About Jobs

- When a job is preempted because it completed its time quantum we know something about it

Learning About Jobs

- When a job is preempted because it completed its time quantum we know something about it

It Is Not Short

(at least not shorter than a quantum)

Learning About Jobs

- When a job is preempted because it completed its time quantum we know something about it

It Is Not Short

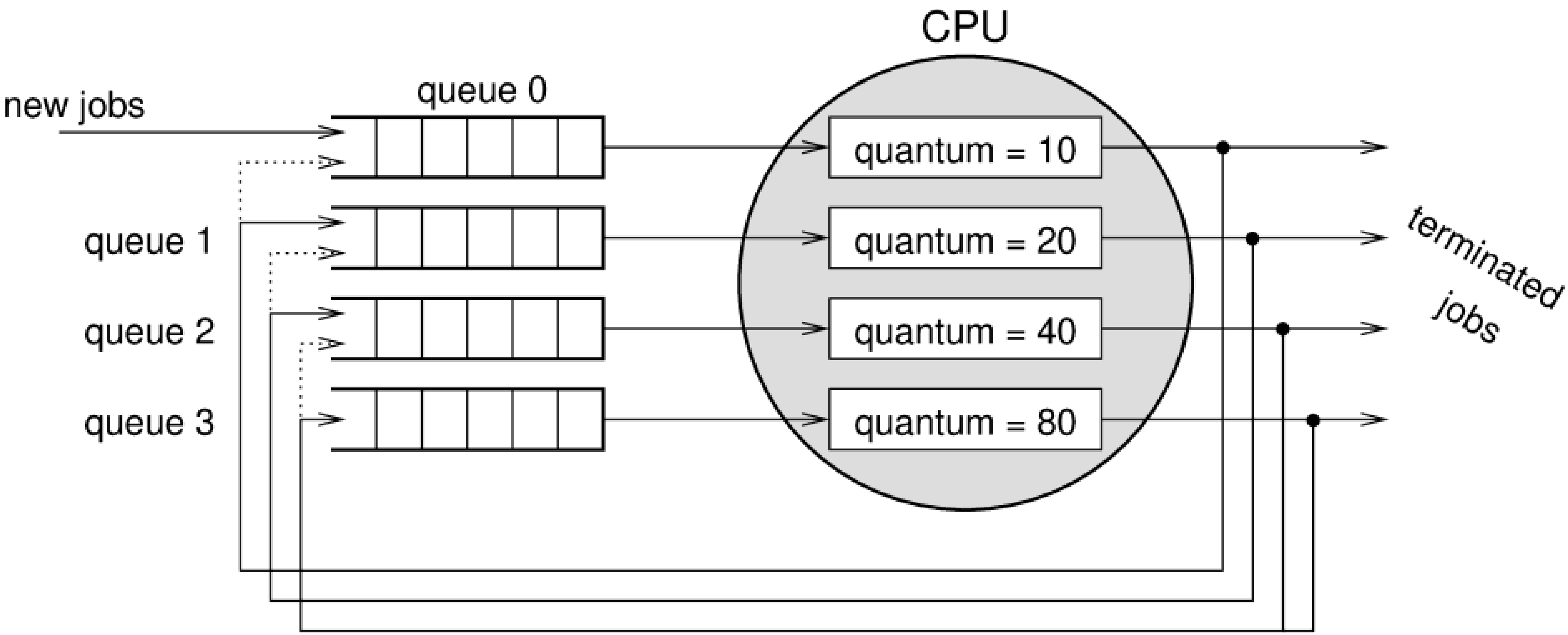
(at least not shorter than a quantum)

- So it should be given lower priority than new jobs (which may indeed be short)

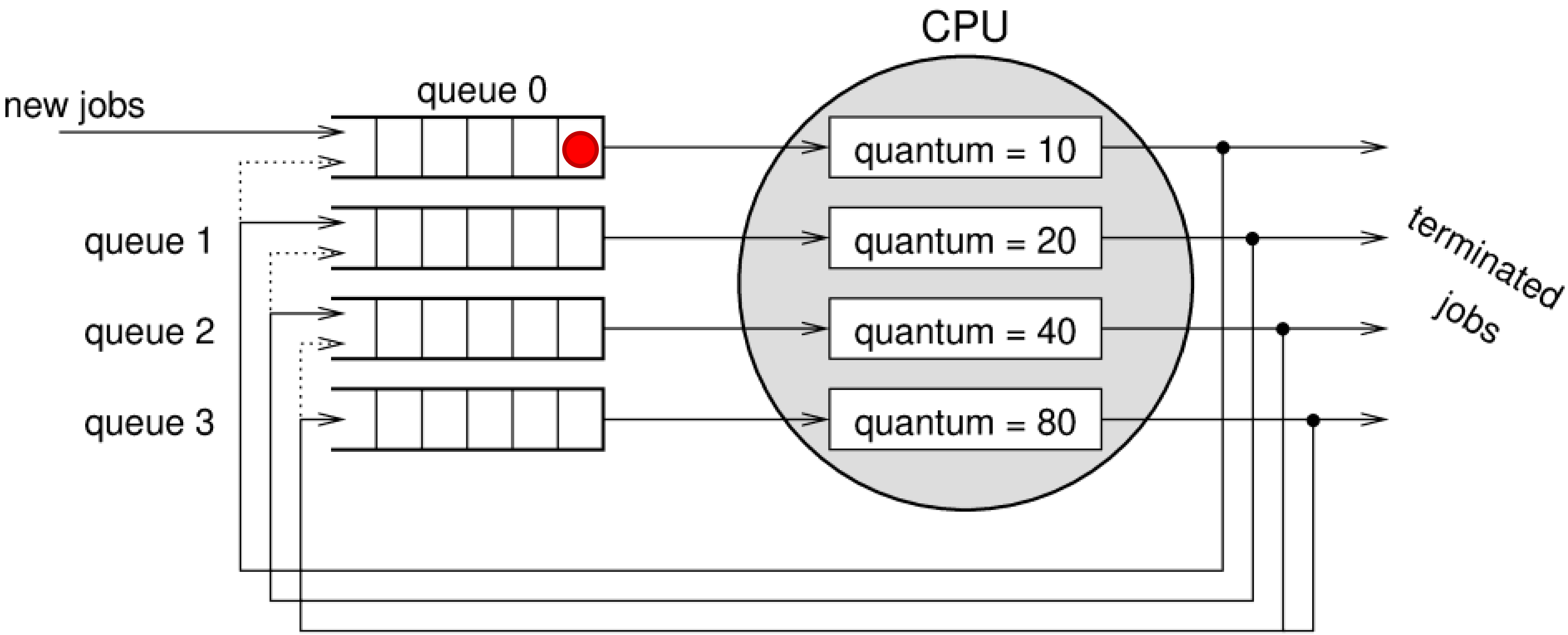
Multi-Level Feedback Queues

- When a job completes its quantum, do **not** return it to the run queue
- Instead, place it in a *separate* queue with other long jobs
- Serve them only if the original queue is empty
- Can have multiple such queues
  - The different queues can have different time quanta
  - And different scheduling disciplines (FCFS, RR)

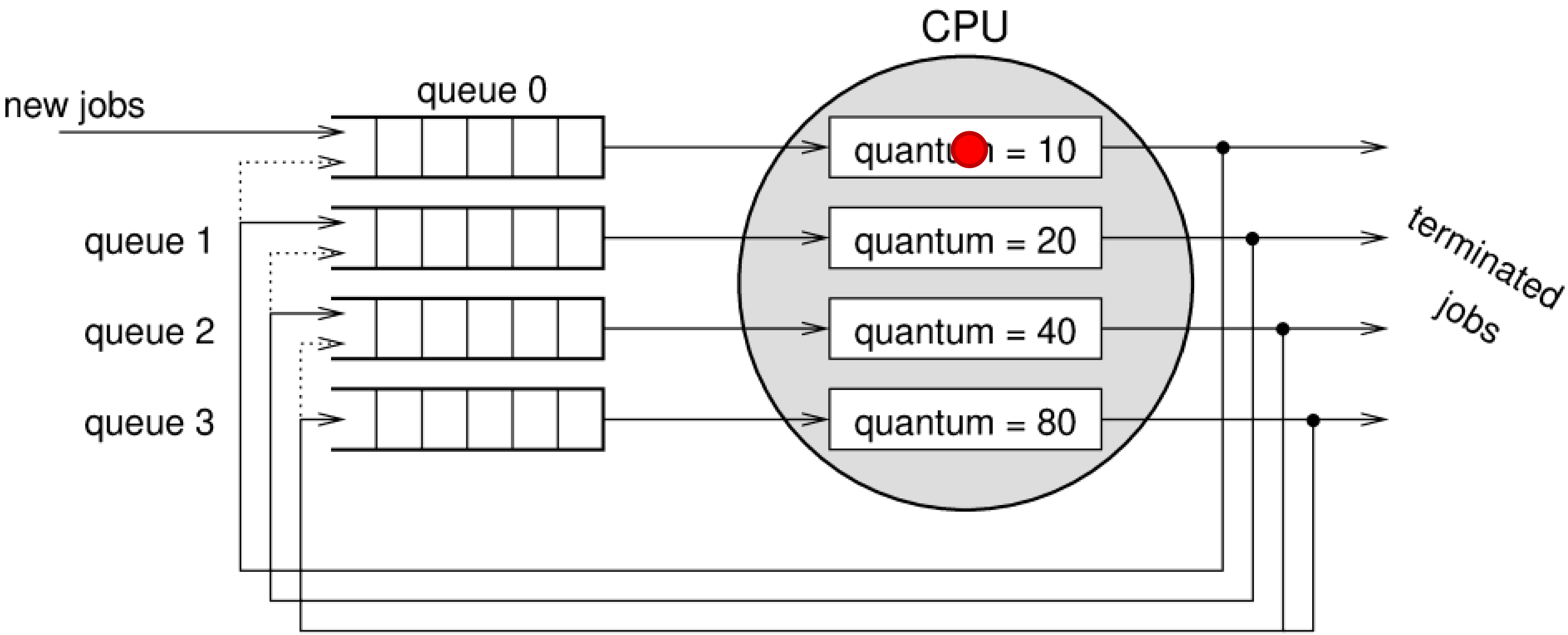
Multi-Level Feedback Queues



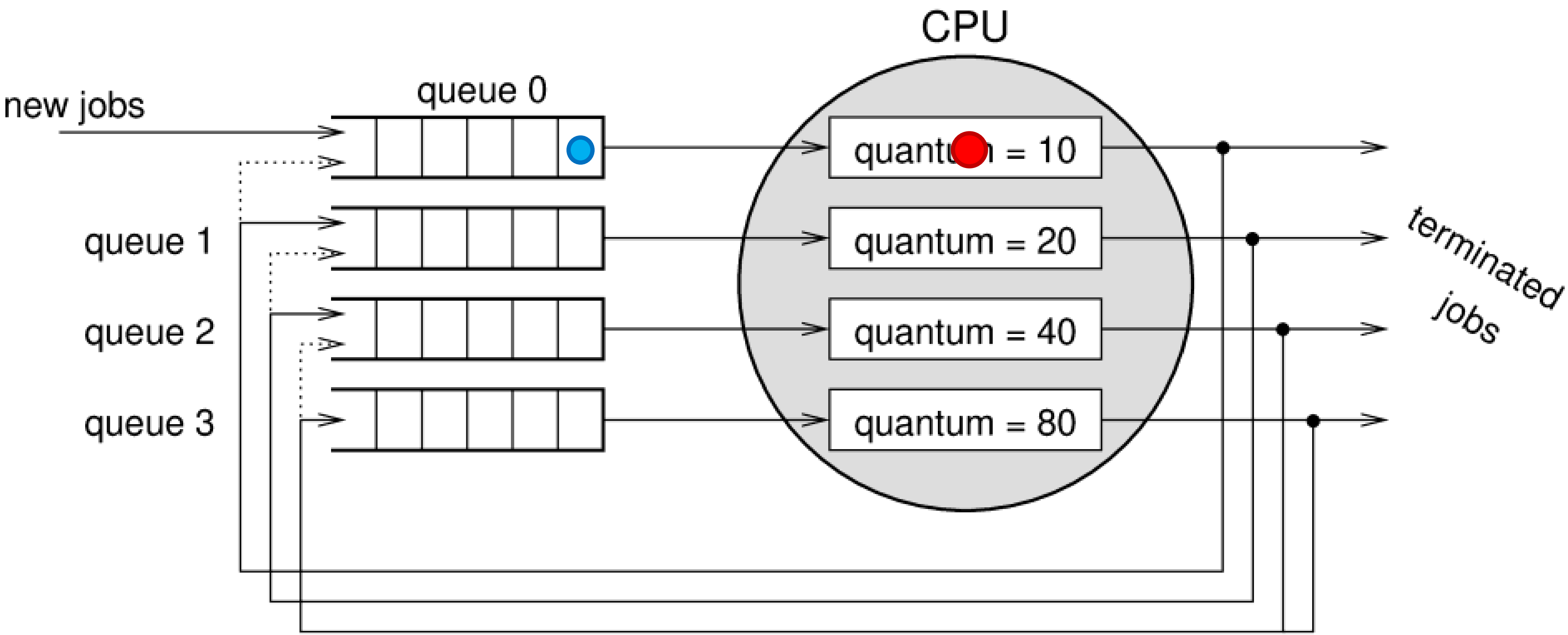
Multi-Level Feedback Queues



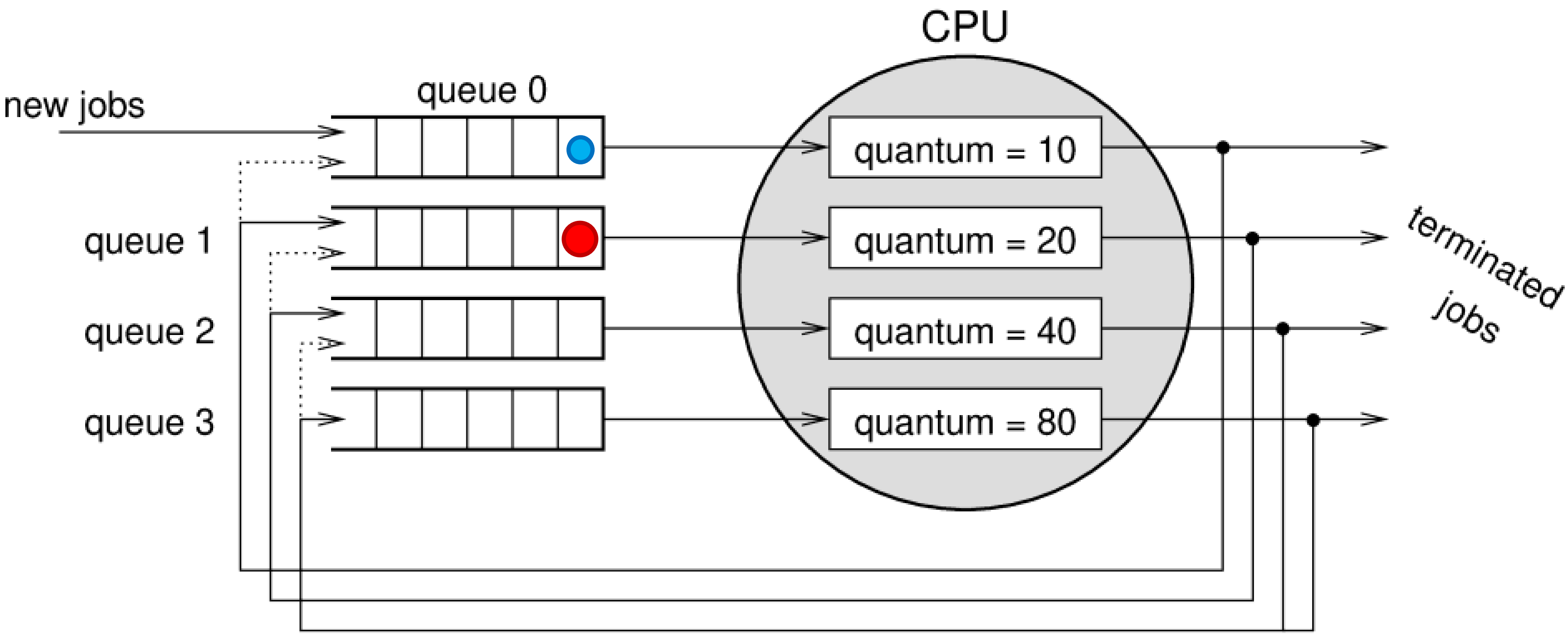
Multi-Level Feedback Queues



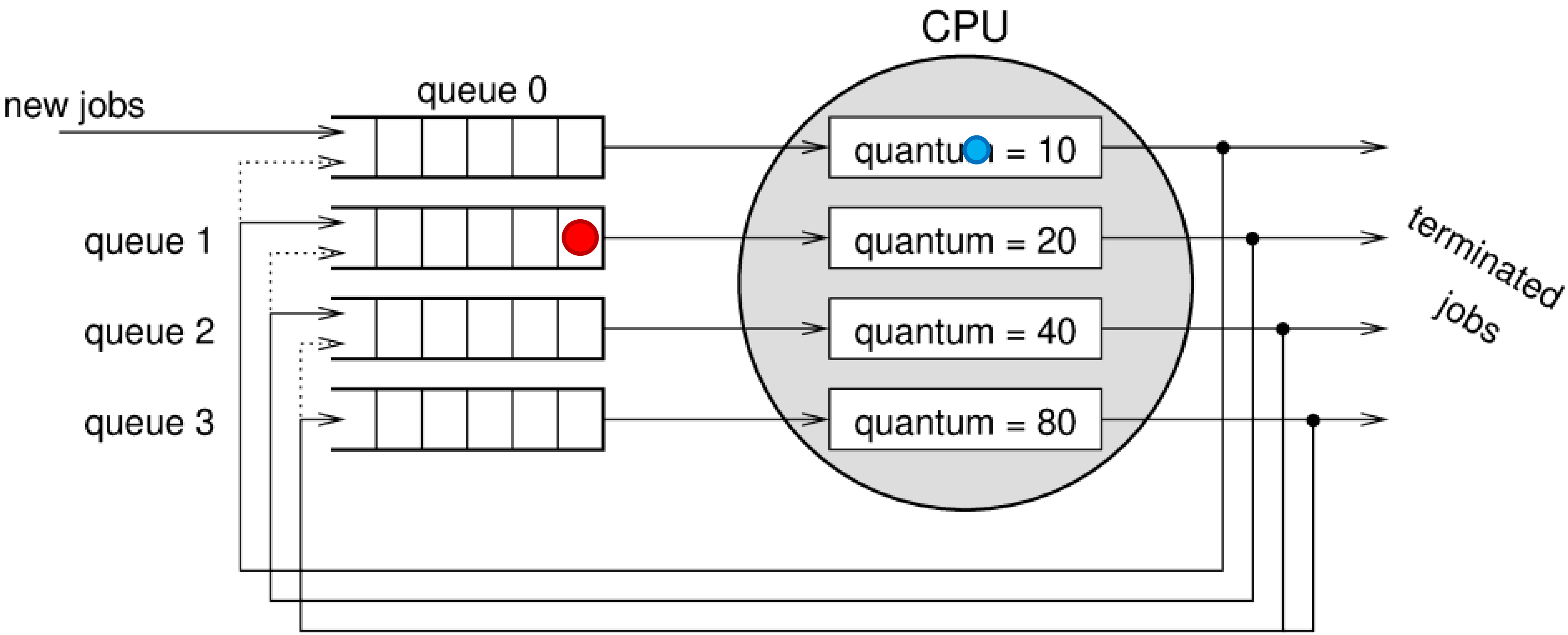
Multi-Level Feedback Queues



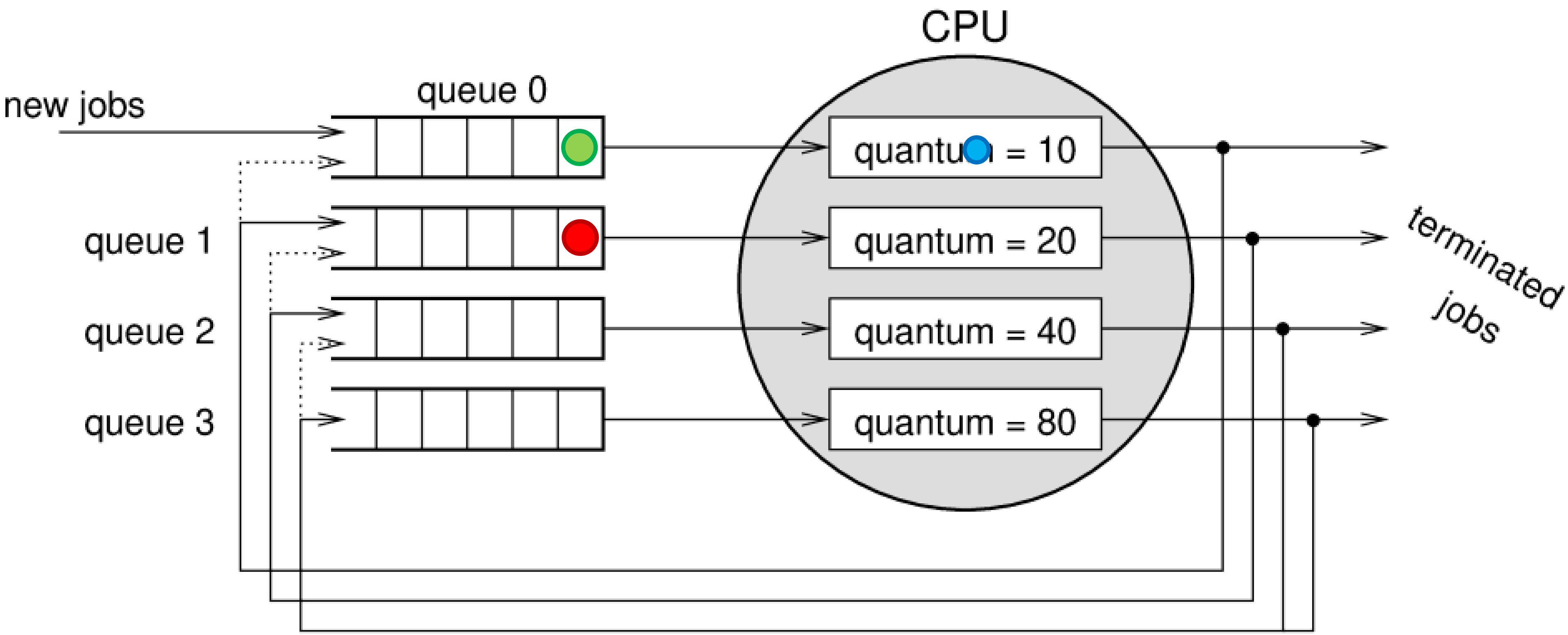
Multi-Level Feedback Queues



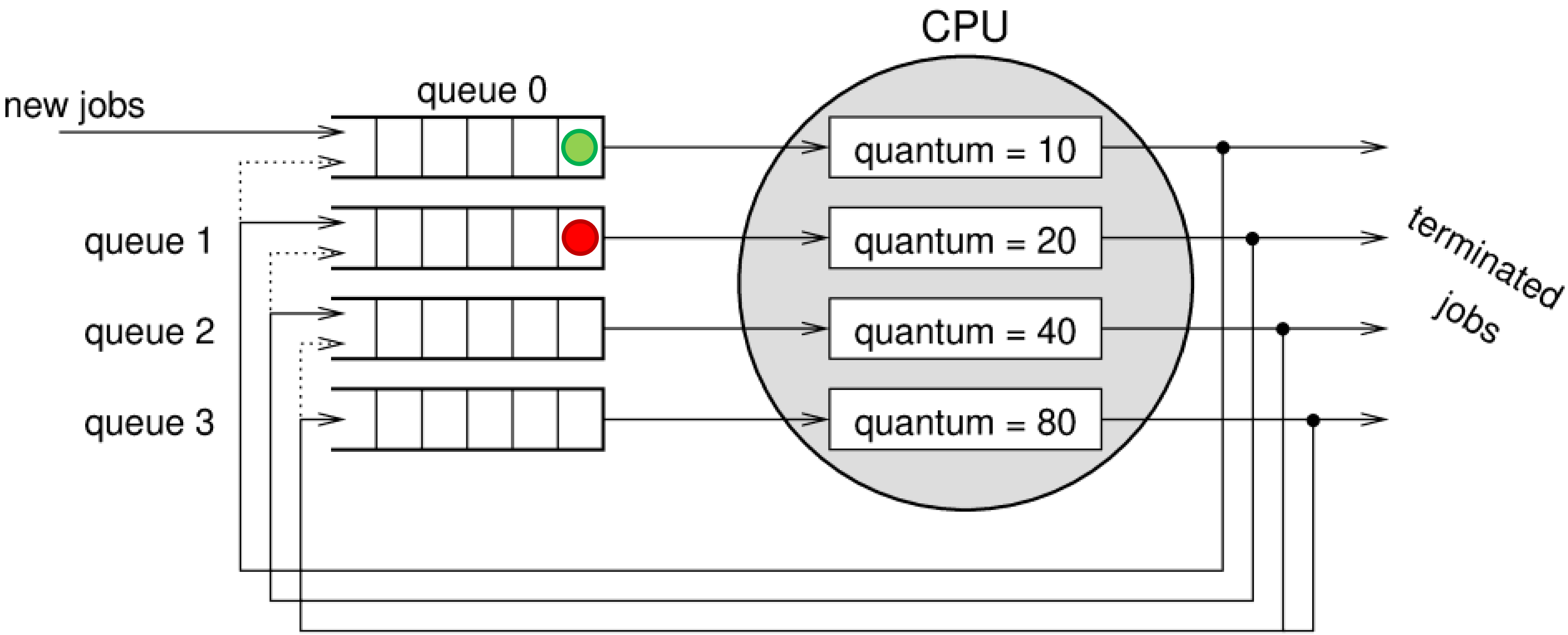
Multi-Level Feedback Queues



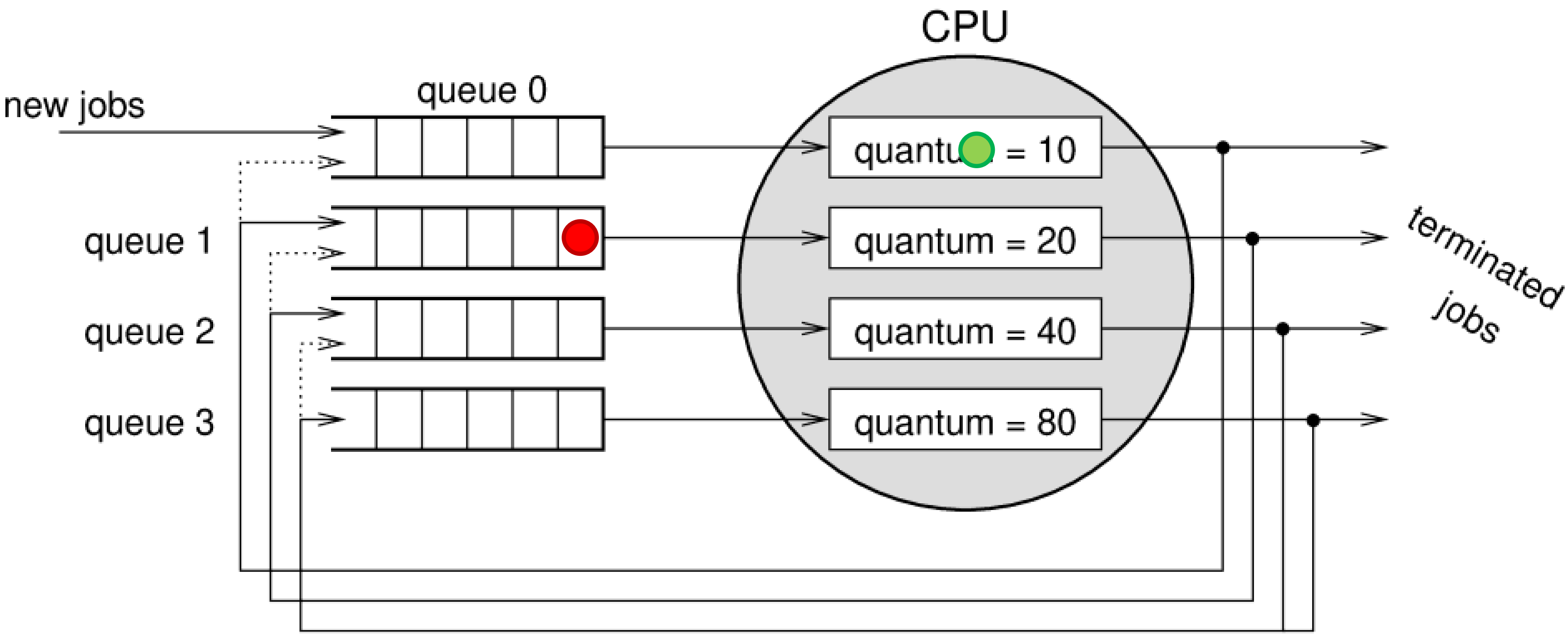
Multi-Level Feedback Queues



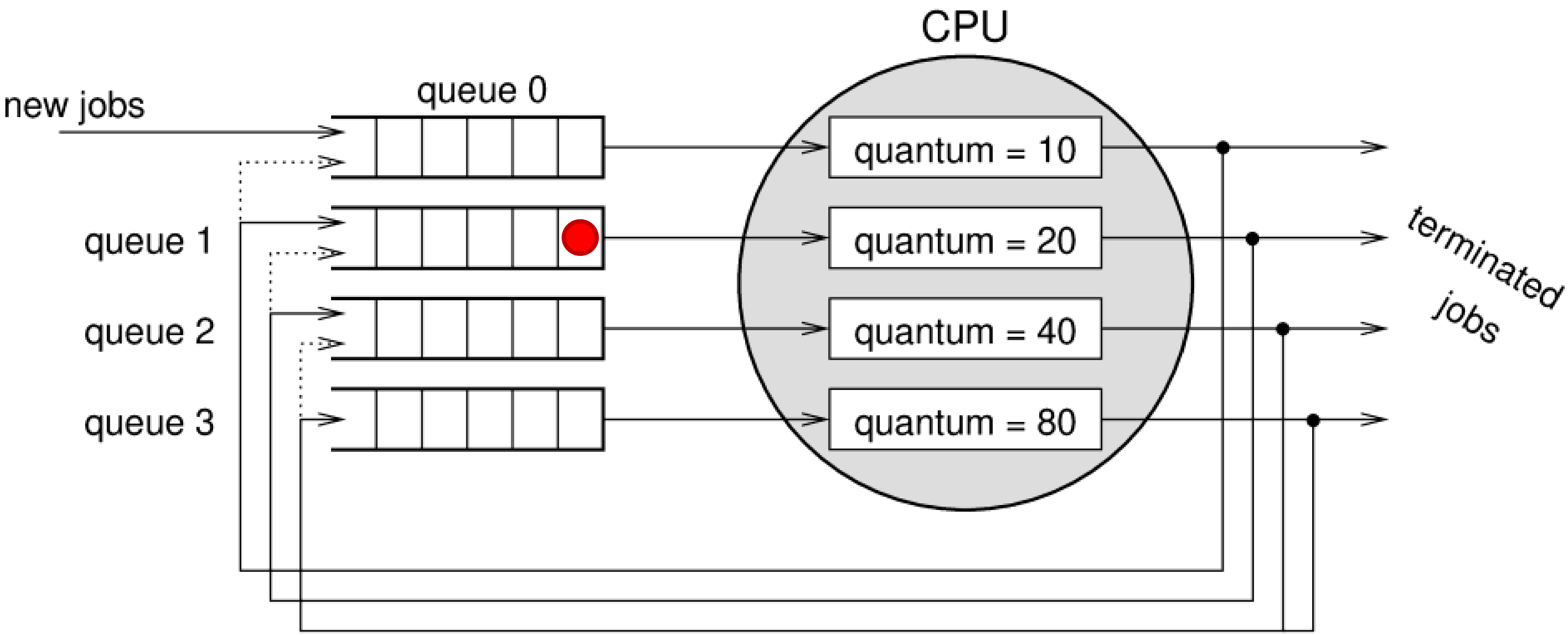
Multi-Level Feedback Queues



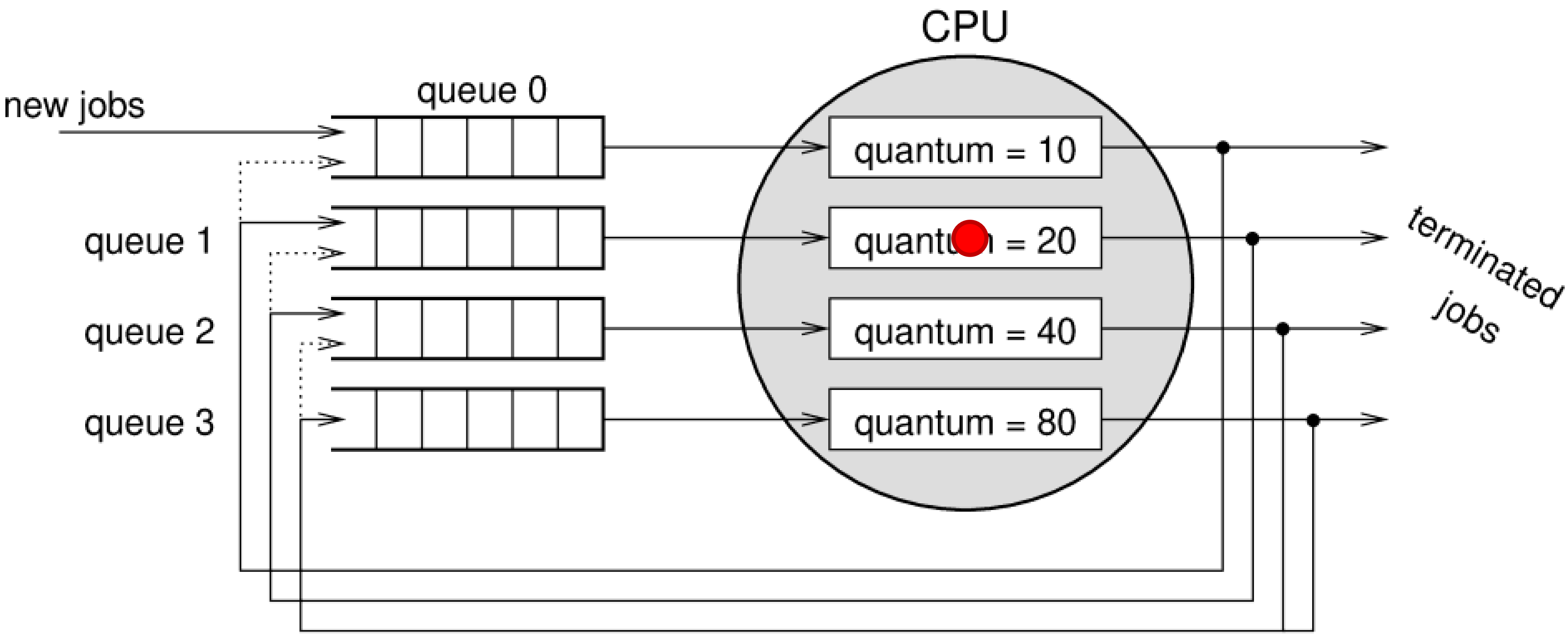
Multi-Level Feedback Queues



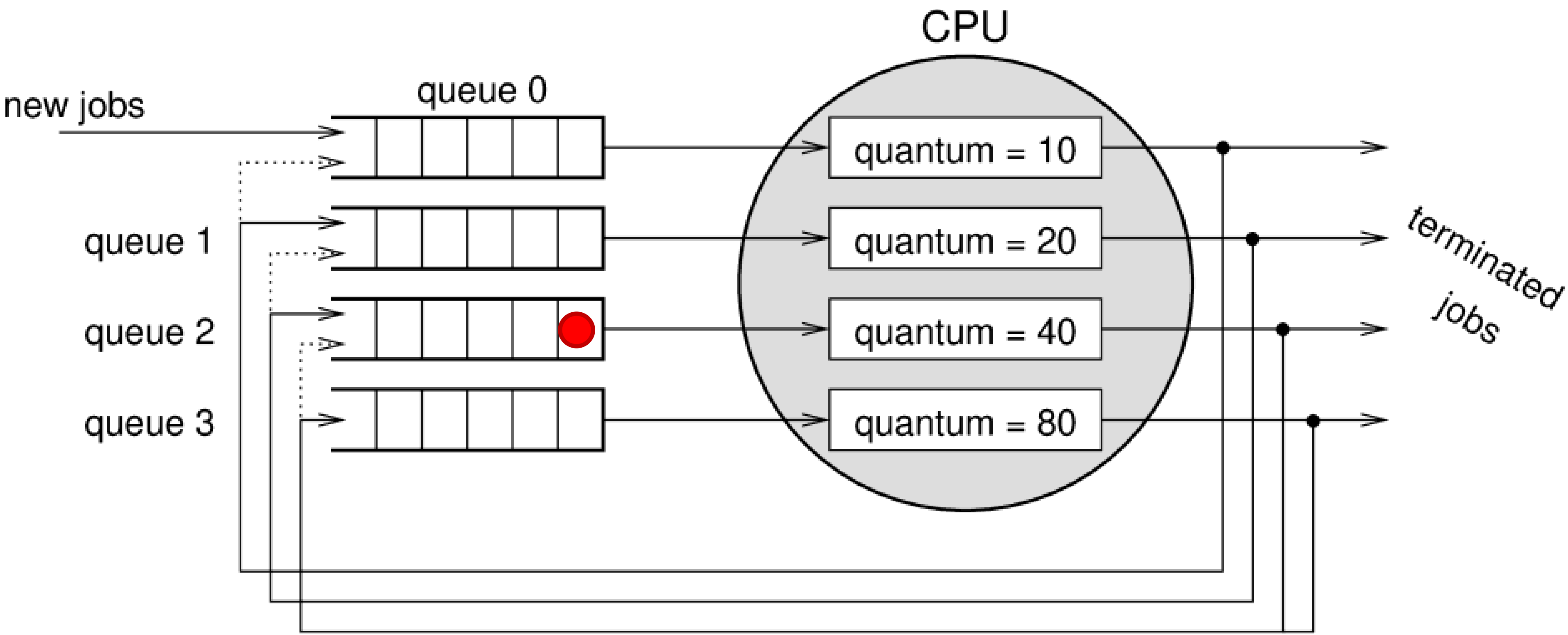
Multi-Level Feedback Queues



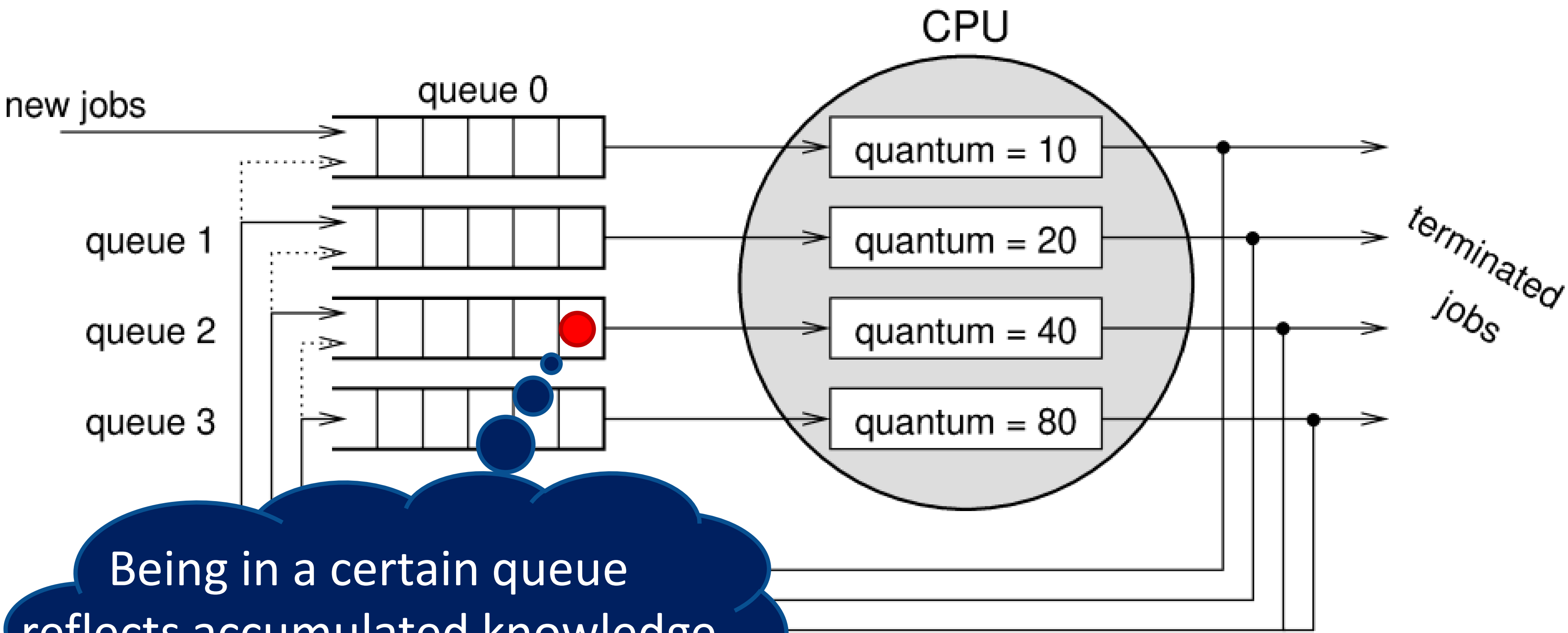
Multi-Level Feedback Queues



Multi-Level Feedback Queues



Multi-Level Feedback Queues



Being in a certain queue reflects accumulated knowledge about process length

The Rules

- New jobs enter the highest-priority queue
- Always schedule jobs from the highest priority non-empty queue
  - Jobs in lower priority queues don't run
- If a job completes its quantum it is demoted to a lower priority queue
- Jobs in the same queue are scheduled round-robin

Multi-Level Feedback Queues

To summarize, Multi-Level Feedback Queues:

- Manage to prioritize short jobs (like SJF)
- Without knowing in advance when additional jobs will arrive
- Or how long new jobs will run
- By using preemption
- And lightweight accumulated knowledge about jobs

Variants used by all major systems

Starvation

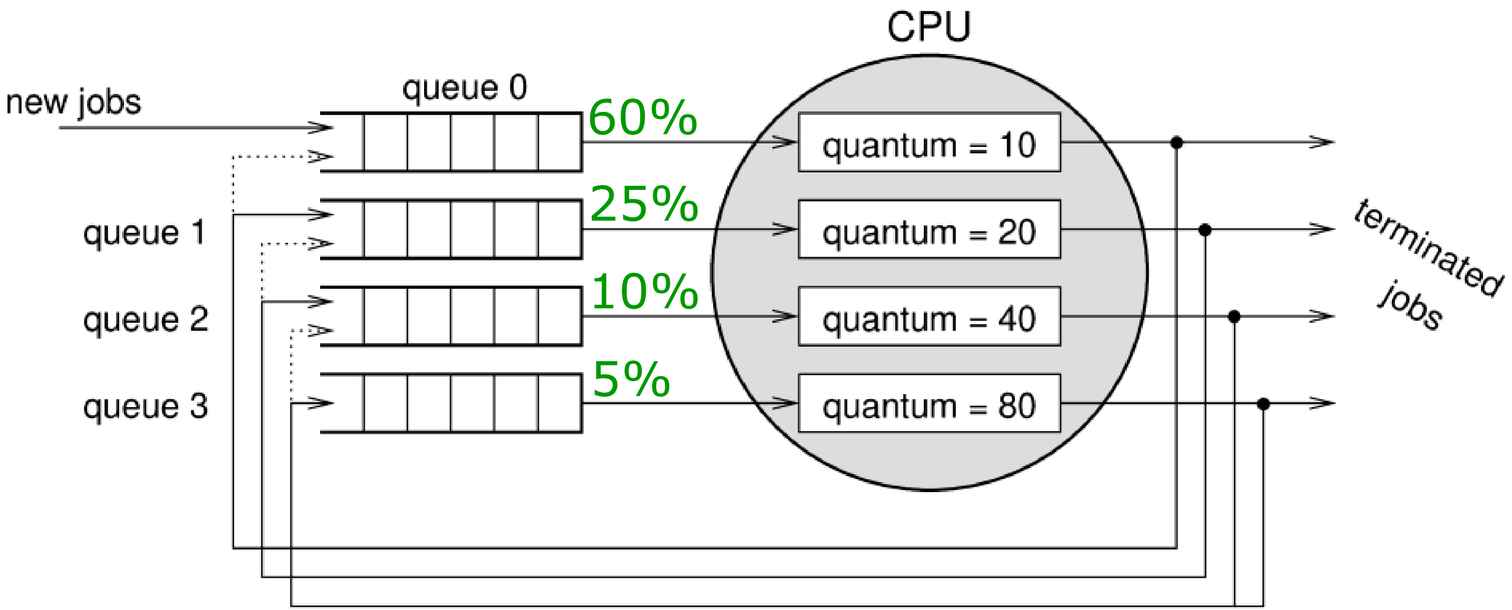
But what if new (short) jobs continue to arrive all the time?

The longer job will be starved

(it will never be scheduled to run)

Possible Solution: Allocations

- Give each queue a relative allocation of CPU time
- Higher allocations to higher priority queues (which contain shorter jobs)
- Non-zero allocation to lower priority queues (longer jobs)



Possible Solution: Aging

Negative feedback principle:

- Running reduces your priority to run more
  - ➢Move to lower queue
- Waiting increases your priority to run
  - ➢Move to higher queue

Classic Unix Scheduler

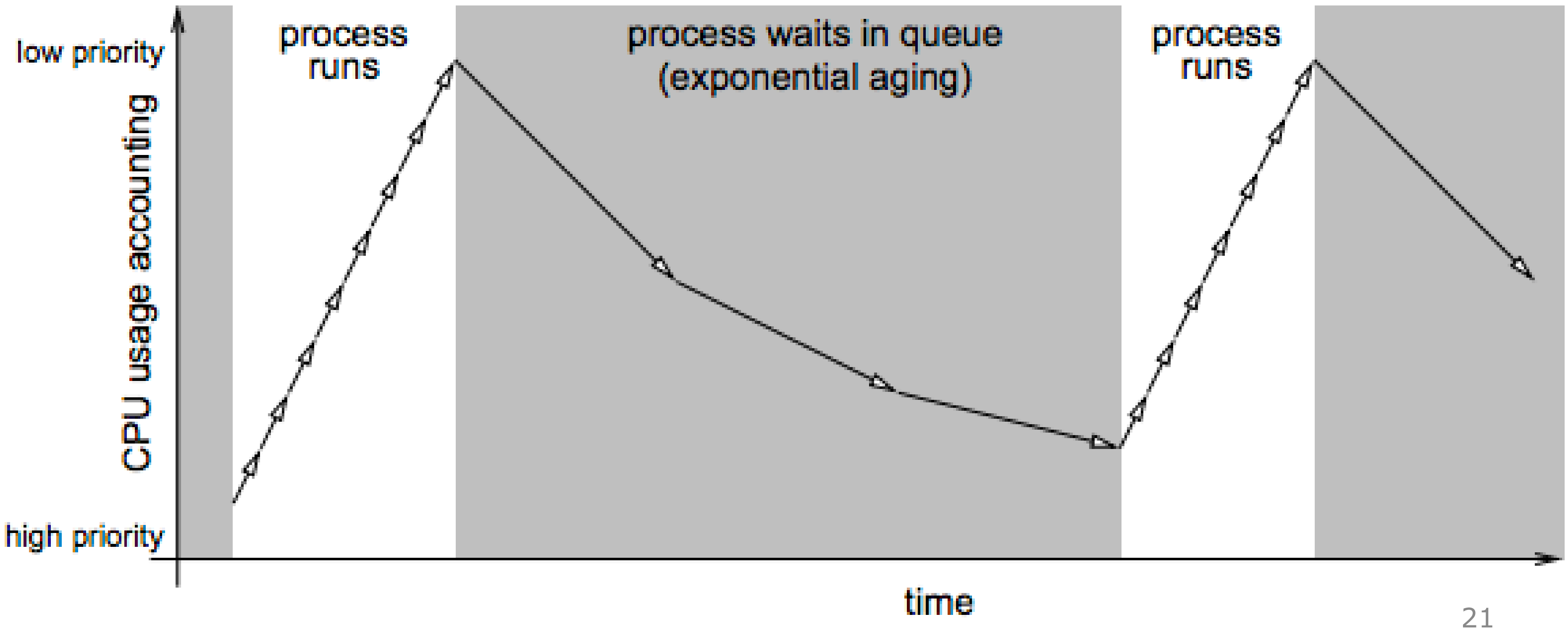
- 128 queues for 128 priority levels
  - 0-49 for kernel
  - 50-127 for user mode processes
- Always schedule highest priority process (=lowest priority score)
- Kernel: priority based on reason for sleep
  - Disk I/O = 20
  - Terminal I/O = 28

Classic Unix Scheduler (simplified)

- **User priority = cpu_usage + base** (=50)
- On each clock tick (1/100th of second):
  - Add 1 to *running process* cpu_usage (accounting info in PCB; reduces priority)
- At end of burst (quantum=10 ticks or block):
  - If a higher priority process exists, switch to it
  - Switch to next process at same priority (RR)
- Every second (100 clock ticks):
  - Divide cpu_usage of *all processes* by 2 (increase priority)

Aging in Classic Unix Scheduler

- **priority = cpu_usage + base**
- cpu_usage incremented when process runs (priority is reduced)
- cpu_usage is divided by 2 every second the process does not run

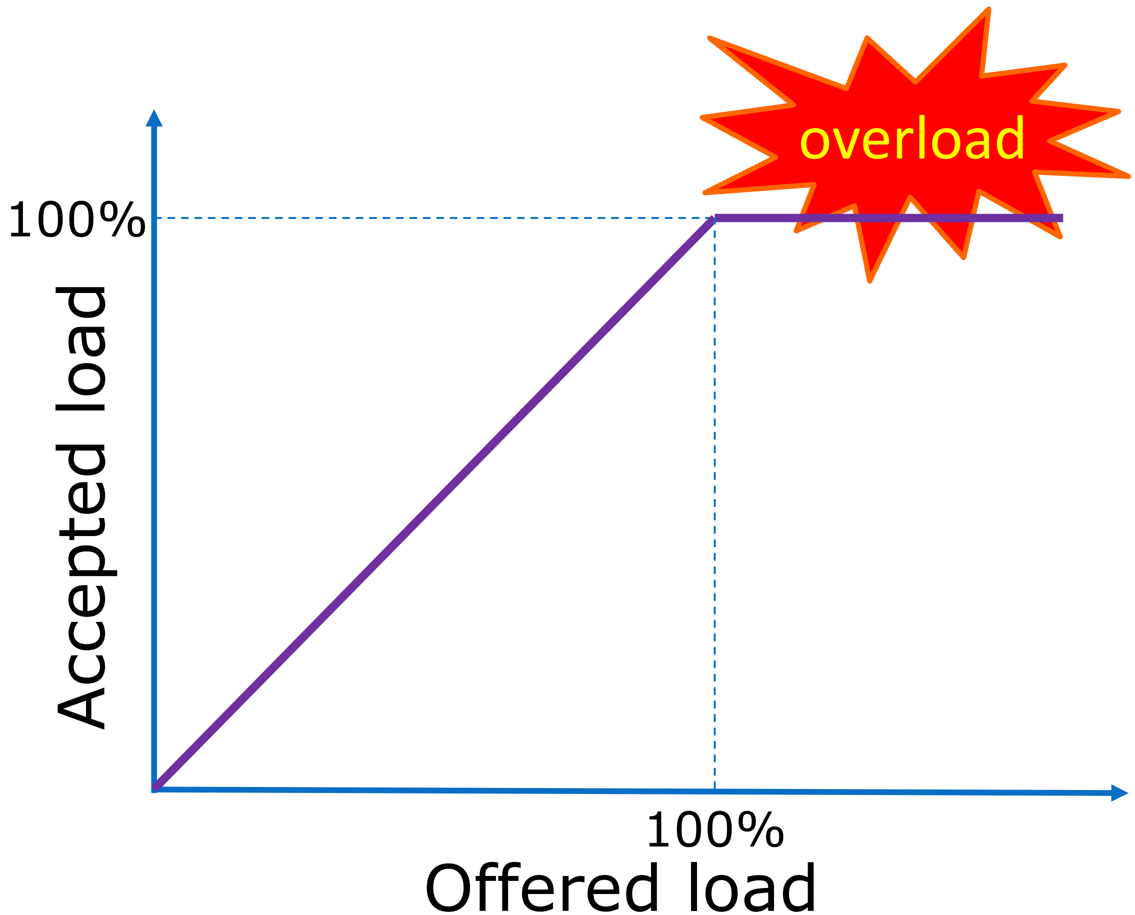


Alternative View: Not a Bug - A Feature!

- "Short jobs arrive all the time" means the system is **overloaded**
- It is **impossible** to run everything

Overload

- "Load" is the fraction of system capacity the users want
- If they want more than 100% they won't get it…



Alternative View: Not a Bug – A Feature!

- "Short jobs arrive all the time" means the system is **overloaded**
- It is **impossible** to run everything
- So need to **decide** what not to run
- Sacrificing long jobs makes sense
  - Not interactive: nobody is waiting for them
  - One long job = many short jobs
- Alternatively, they will run once the load abates (if it does, e.g. at night)

Back to Multi-Level Feedback Qs

- How does it treat computational vs. interactive processes?

Back to Multi-Level Feedback Qs

- How does it treat computational vs. interactive processes?
  - Computational get lower priority

Back to Multi-Level Feedback Qs

- How does it treat computational vs. interactive processes?
  - Computational get lower priority
  - Simple interactive jobs (e.g. editor) get higher priority: will run immediately when ready!

Back to Multi-Level Feedback Qs

- How does it treat computational vs. interactive processes?
  - Computational get lower priority
  - Simple interactive jobs (e.g. editor) get higher priority: will run immediately when ready!
  - Complex interactive jobs (e.g. 3D game, lots of CPU to render) may get low priority!
  - Modern schedulers try to prioritize, e.g. based on active window

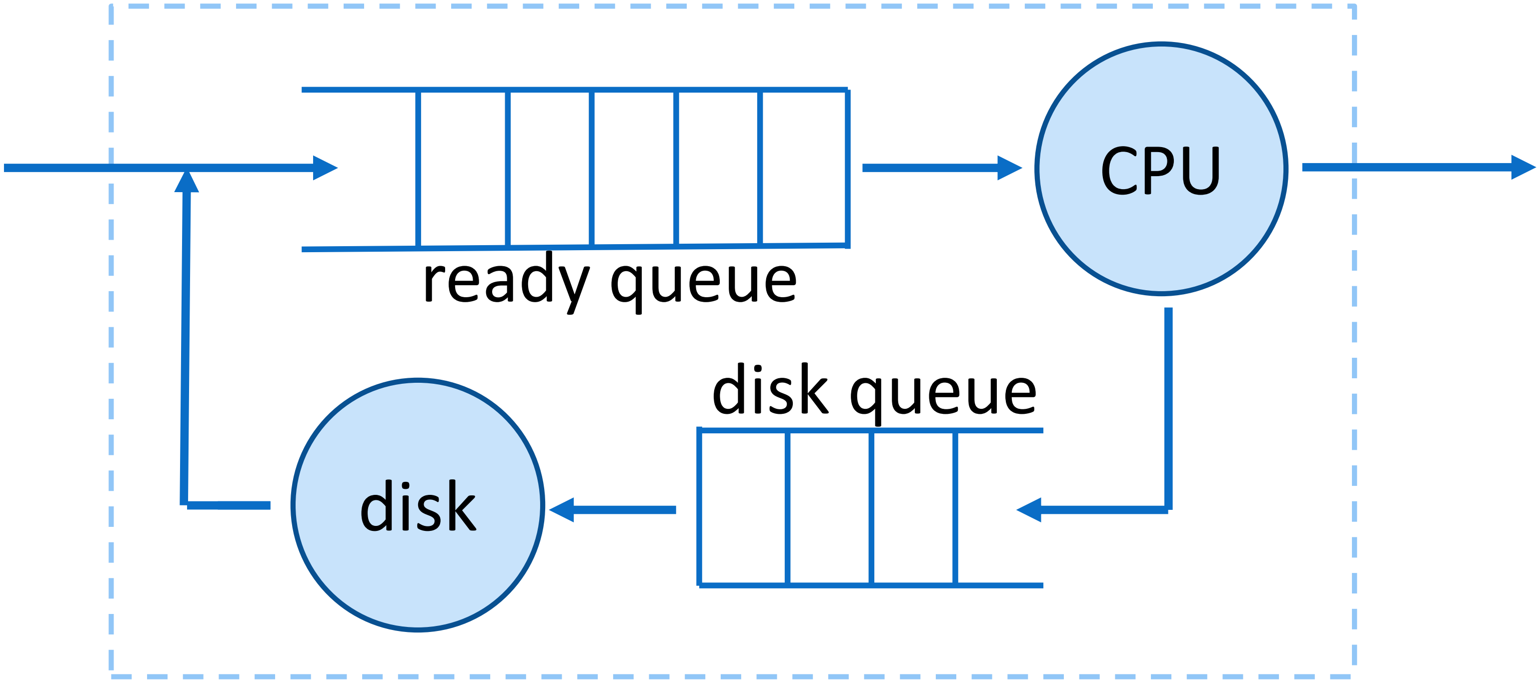
PERFORMANCE EVALUATION

How to Evaluate a Scheduler?

- Queuing models
  - Use queuing theory to solve a stochastic model and derive average performance metrics
  - Based on simplifying mathematical assumptions
- Simulation
  - Use a program that implements a model of a computer system
  - Model is a simplification of reality
- Implementation
  - Put the actual algorithm in a real system for evaluation under real operating conditions

Queueing Analysis

- System model: servers with queues
  - Queueing network: multiple interconnected servers



Queueing Analysis

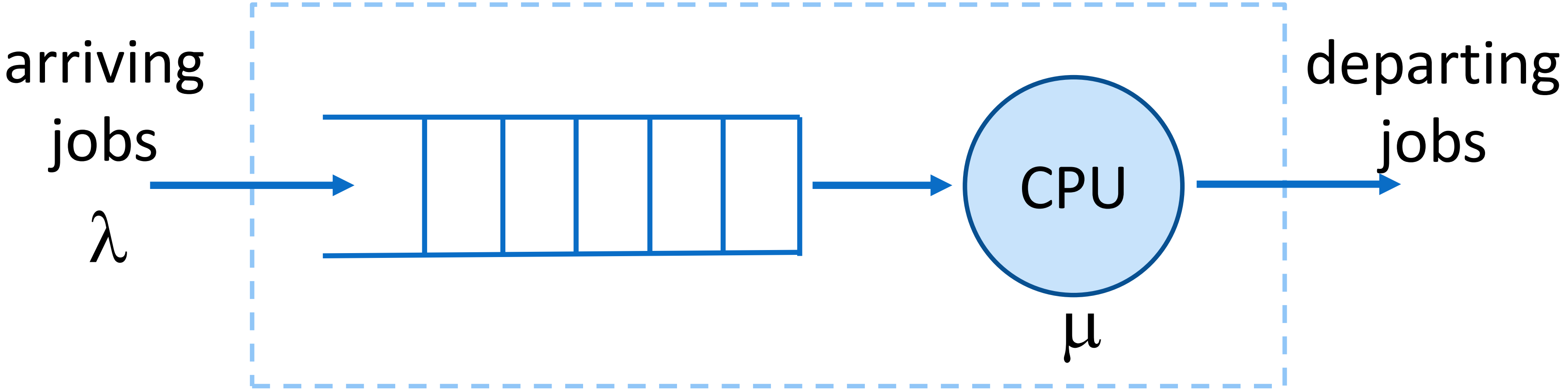
- System model: servers with queues
  - Queueing network: multiple interconnected servers
- Arrival process
  - Average jobs per unit time
- Service discipline: FCFS
- Service times
  - Exponentially distributed

parameters

- Result: average response time

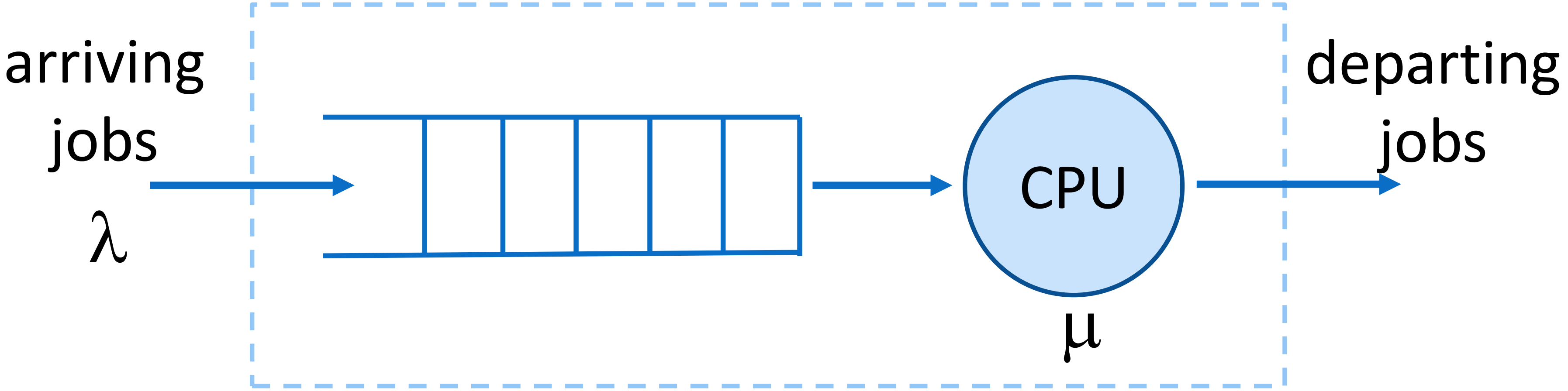
M/M/1 Queue

- A single server
- Poisson arrivals at rate λ
- Exponential service at rate μ



M/M/1 Queue

- A single server
- Poisson arrivals at rate λ
- Exponential service at rate μ
- Stability constraint: $\lambda \leq \mu$



The Question:

Given λ and μ,

What will the average response time be?

A Simplistic Answer

- Assume jobs arrive at precise intervals
- And run for exactly equal times

A Simplistic Answer

- Assume jobs arrive at precise intervals
- And run for exactly equal times

Inter-arrival time	Runtime	Response time
10	1	

A Simplistic Answer

- Assume jobs arrive at precise intervals
- And run for exactly equal times

Inter-arrival time	Runtime	Response time
10	1	1

A Simplistic Answer

- Assume jobs arrive at precise intervals
- And run for exactly equal times

Inter-arrival time	Runtime	Response time
10	1	1
5	1	

A Simplistic Answer

- Assume jobs arrive at precise intervals
- And run for exactly equal times

Inter-arrival time	Runtime	Response time
10	1	1
5	1	1

A Simplistic Answer

- Assume jobs arrive at precise intervals
- And run for exactly equal times

Inter-arrival time	Runtime	Response time
10	1	1
5	1	1
3	1	

A Simplistic Answer

- Assume jobs arrive at precise intervals
- And run for exactly equal times

Inter-arrival time	Runtime	Response time
10	1	1
5	1	1
3	1	1

A Simplistic Answer

- Assume jobs arrive at precise intervals
- And run for exactly equal times

Inter-arrival time	Runtime	Response time
10	1	1
5	1	1
3	1	1
2	1	1

A Simplistic Answer

- Assume jobs arrive at precise intervals
- And run for exactly equal times

Inter-arrival time	Runtime	Response time
10	1	1
5	1	1
3	1	1
2	1	1
1	1	

A Simplistic Answer

- Assume jobs arrive at precise intervals
- And run for exactly equal times

Inter-arrival time	Runtime	Response time
10	1	1
5	1	1
3	1	1
2	1	1
1	1	1

A Simplistic Answer

- Assume jobs arrive at precise intervals
- And run for exactly equal times

Inter-arrival time	Runtime	Response time	System load
10	1	1	10%
5	1	1	20%
3	1	1	33%
2	1	1	50%
1	1	1	100%

A More Realistic Setting

- What if jobs arrive at different intervals
- And run for different times

A More Realistic Setting

- What if jobs arrive at different intervals
- And run for different times

Average inter-arrival time	Average runtime	System load	Average response time
10	1	10%	?
5	1	20%	?
3	1	33%	?
2	1	50%	?
1	1	100%	?

Little's Law

- We know λ and μ (arrival rate and processing rate)
- We want to find $\bar{r}$ (average response time)
- We can use Little's Law:

$$\bar{n} = \lambda \cdot \bar{r}$$

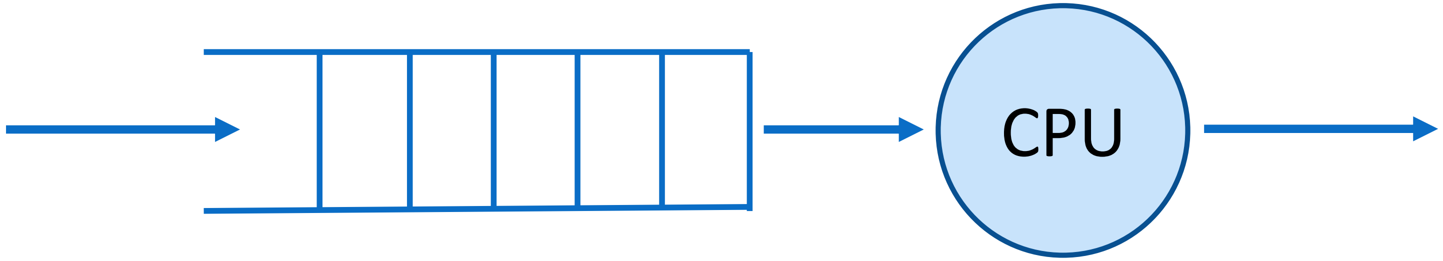
($\bar{n}$ = number of processes in the system)

- But need to find $\bar{n}$ as a function of λ and μ…

M/M/1 States

The system can be in many different states:

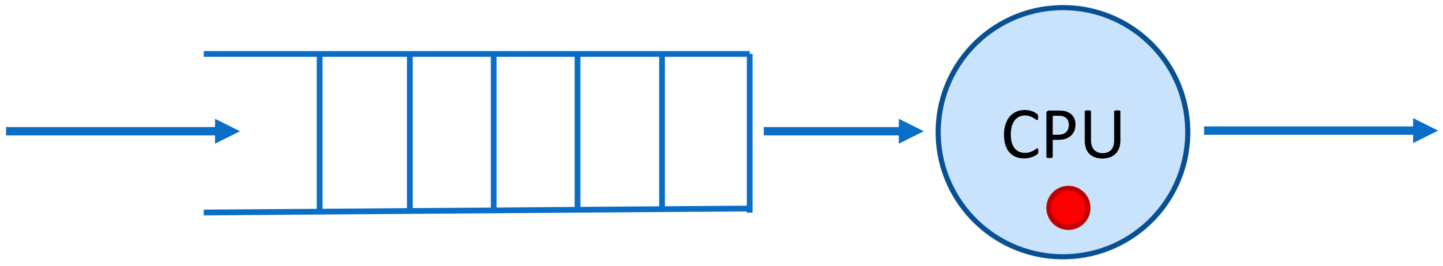
- There may be no process at all



M/M/1 States

The system can be in many different states:

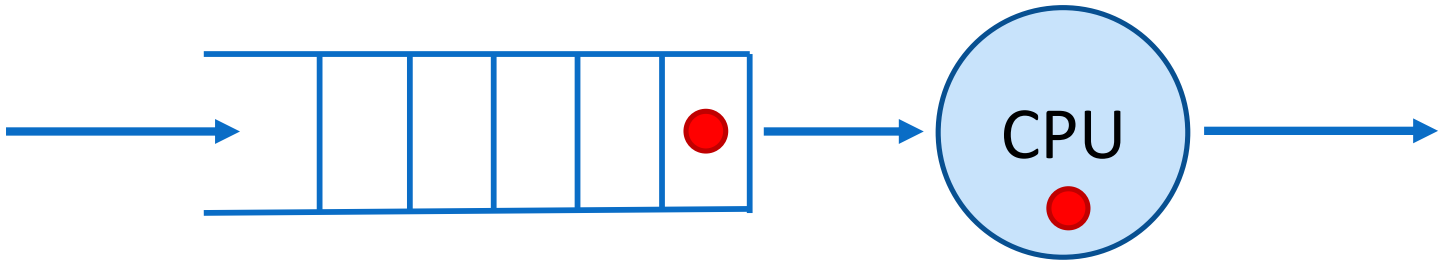
- There may be no process at all
- There can be one process, running



M/M/1 States

The system can be in many different states:

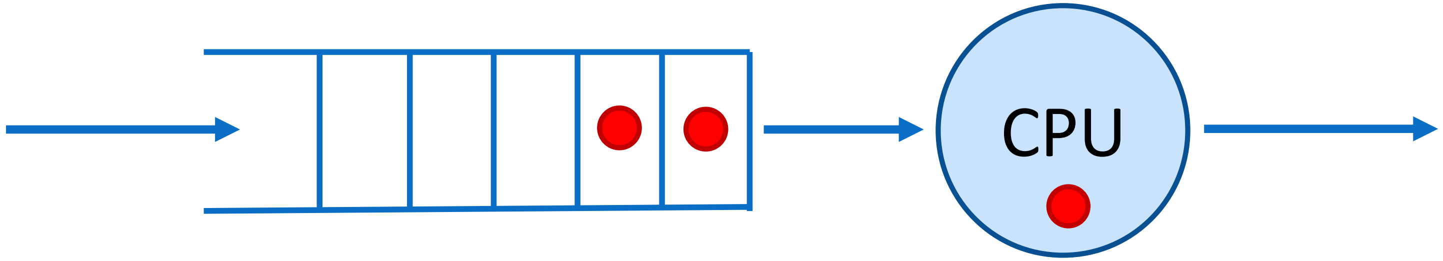
- There may be no process at all
- There can be one process, running
- There can be two processes, one running and one in the queue



M/M/1 States

The system can be in many different states:

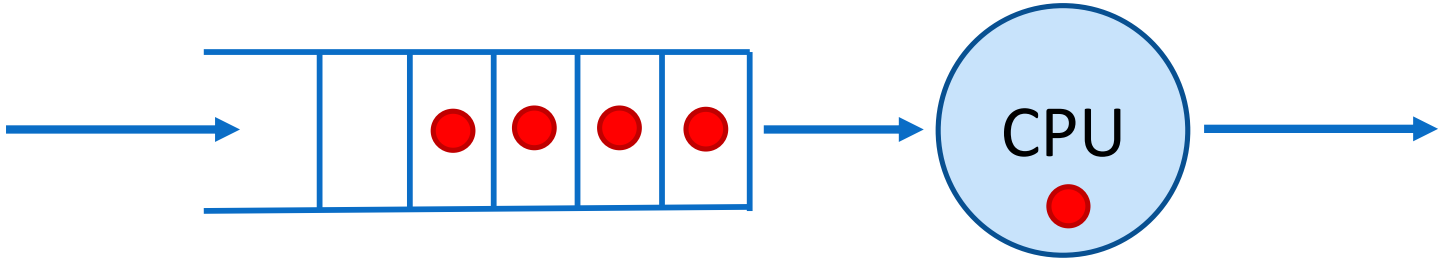
- There may be no process at all
- There can be one process, running
- There can be two processes, one running and one in the queue
- There can be three processes, one running and two in the queue



M/M/1 States

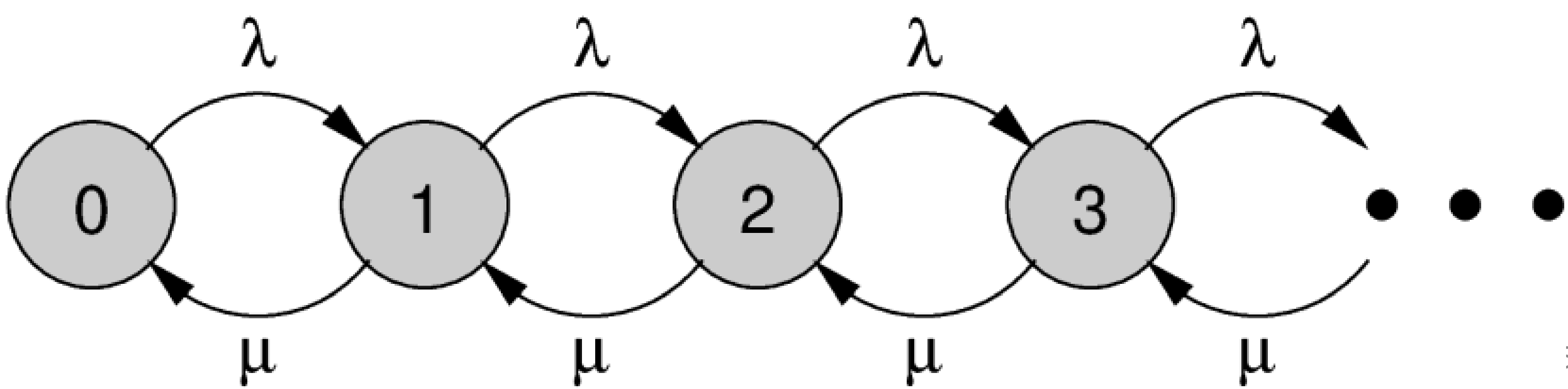
The system can be in many different states:

- There may be no process at all
- There can be one process, running
- There can be two processes, one running and one in the queue
- There can be three processes, one running and two in the queue
- Etc.



M/M/1 States

- Define states by the total number of processes in the system (running and waiting)
- The system moves from state *i* to state *i*+1 at rate λ
- The system moves from state *i*+1 to state *i* at rate μ
- this is a Markov chain



M/M/1 States Markov Chain

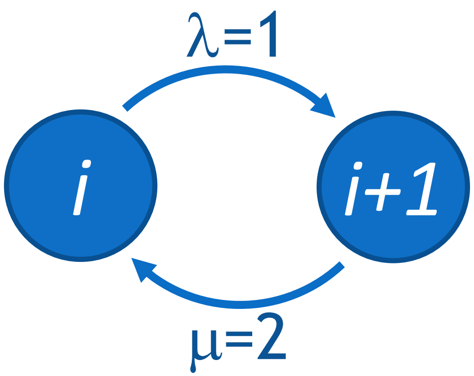
- The states of Markov chains have limiting probabilities (if it is ergodic = connected and no periodic cycles)
  - The probability to be in a given state
  - Denote the probability to be in state *i* by π_i
- Can be calculated using balanced flow (if reversible)
  - The flow from state i to i+1 is equal to the flow in the opposite direction

$$\pi_0 \cdot \lambda = \pi_1 \cdot \mu \qquad \pi_1 = \frac{\lambda}{\mu}\pi_0$$

$$\pi_1 \cdot \lambda = \pi_2 \cdot \mu \qquad \pi_2 = \frac{\lambda}{\mu}\pi_1 = \left(\frac{\lambda}{\mu}\right)^2 \pi_0$$

Example

- Assume $\lambda=1$ (one job arrives each minute)
- And $\mu=2$ (the CPU can process 2 jobs per minute)

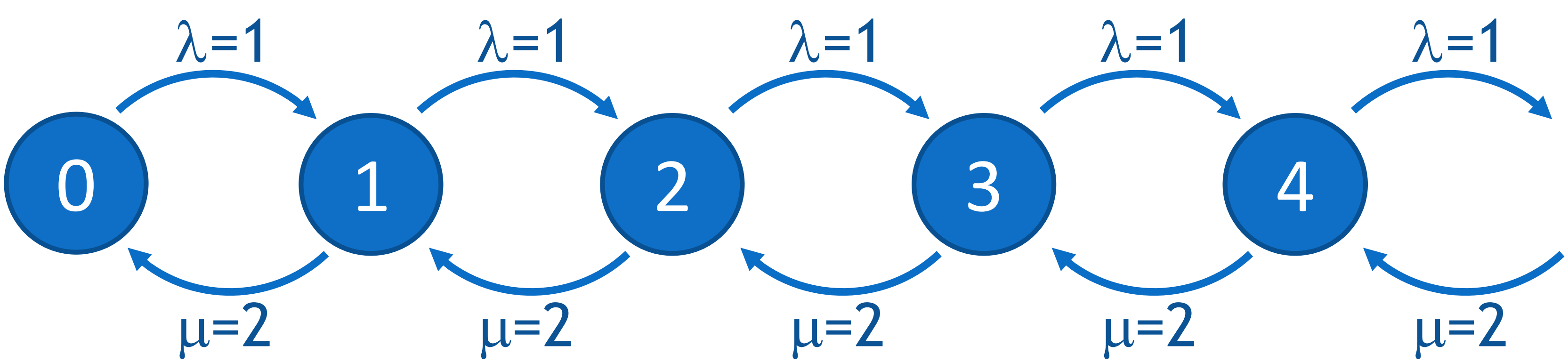


$$\pi_i \lambda = \pi_{i+1} \mu$$

$$\pi_i = 2\,\pi_{i+1}$$

Example

- Assume λ=1 (one job arrives each minute)
- And μ=2 (the CPU can process 2 jobs per minute)



$$\pi_0 = \frac{1}{2} \quad \pi_1 = \frac{1}{4} \quad \pi_2 = \frac{1}{8} \quad \pi_3 = \frac{1}{16} \quad \pi_4 = \frac{1}{32}$$

M/M/1 States Markov Chain

- In general,

$$\pi_i = \left(\frac{\lambda}{\mu}\right)^i \pi_0 = \rho^i \, \pi_0$$

- The sum total is

$$1 = \sum_{i=0}^{\infty} \pi_i = \pi_0 \sum_{i=0}^{\infty} \rho^i = \frac{\pi_0}{1-\rho}$$

- So

$$\pi_i = \rho^i \, (1-\rho) \qquad \rho = \frac{\lambda}{\mu}$$

M/M/1 Analysis

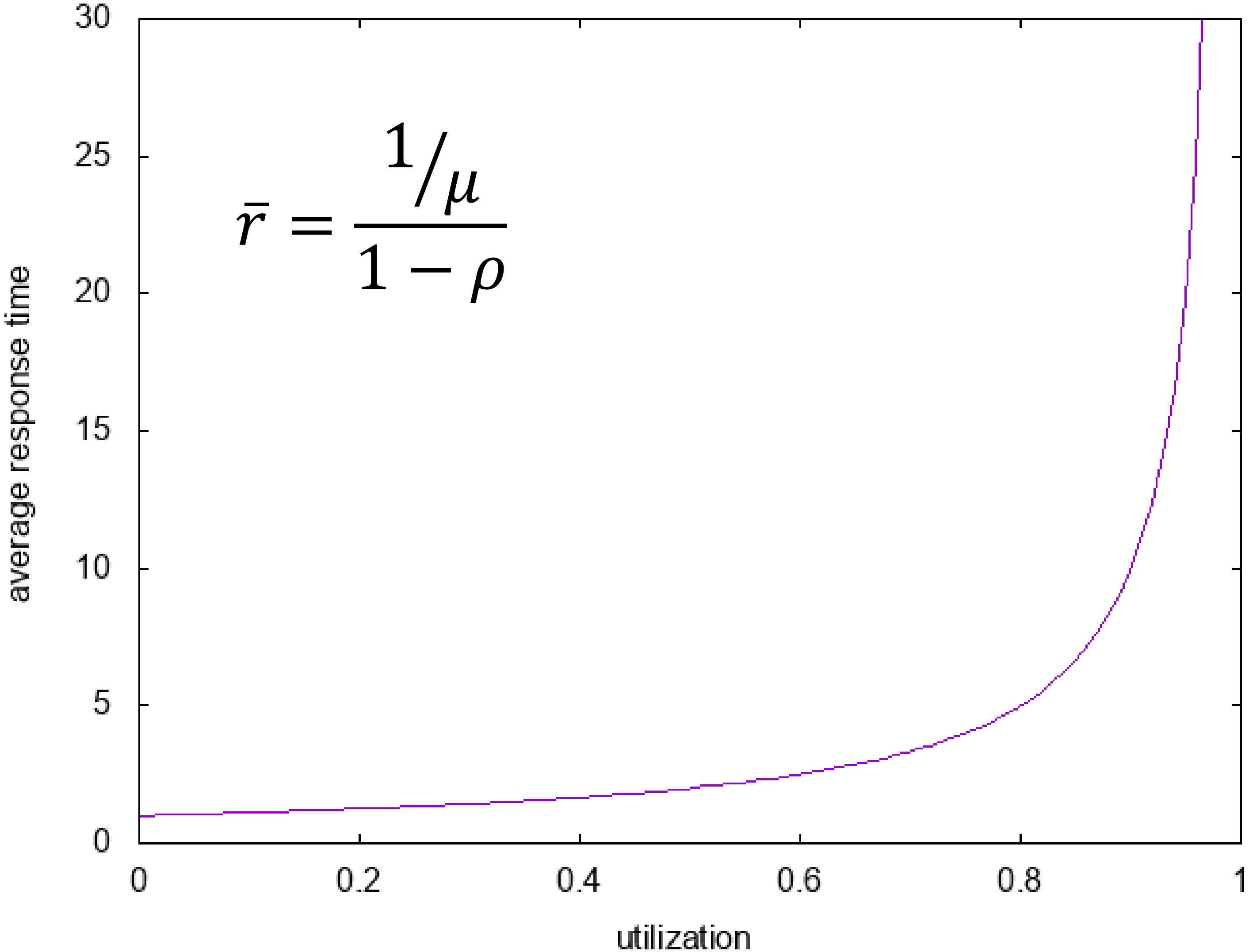
- Use the probabilities of the states to find number of jobs:

$$\bar{n} = \sum_{i=0}^{\infty} i \cdot \pi_i = \sum_{i=0}^{\infty} i(1-\rho)\rho^i = \frac{\rho}{1-\rho}$$

- Finally use Little's law to find the average response time:

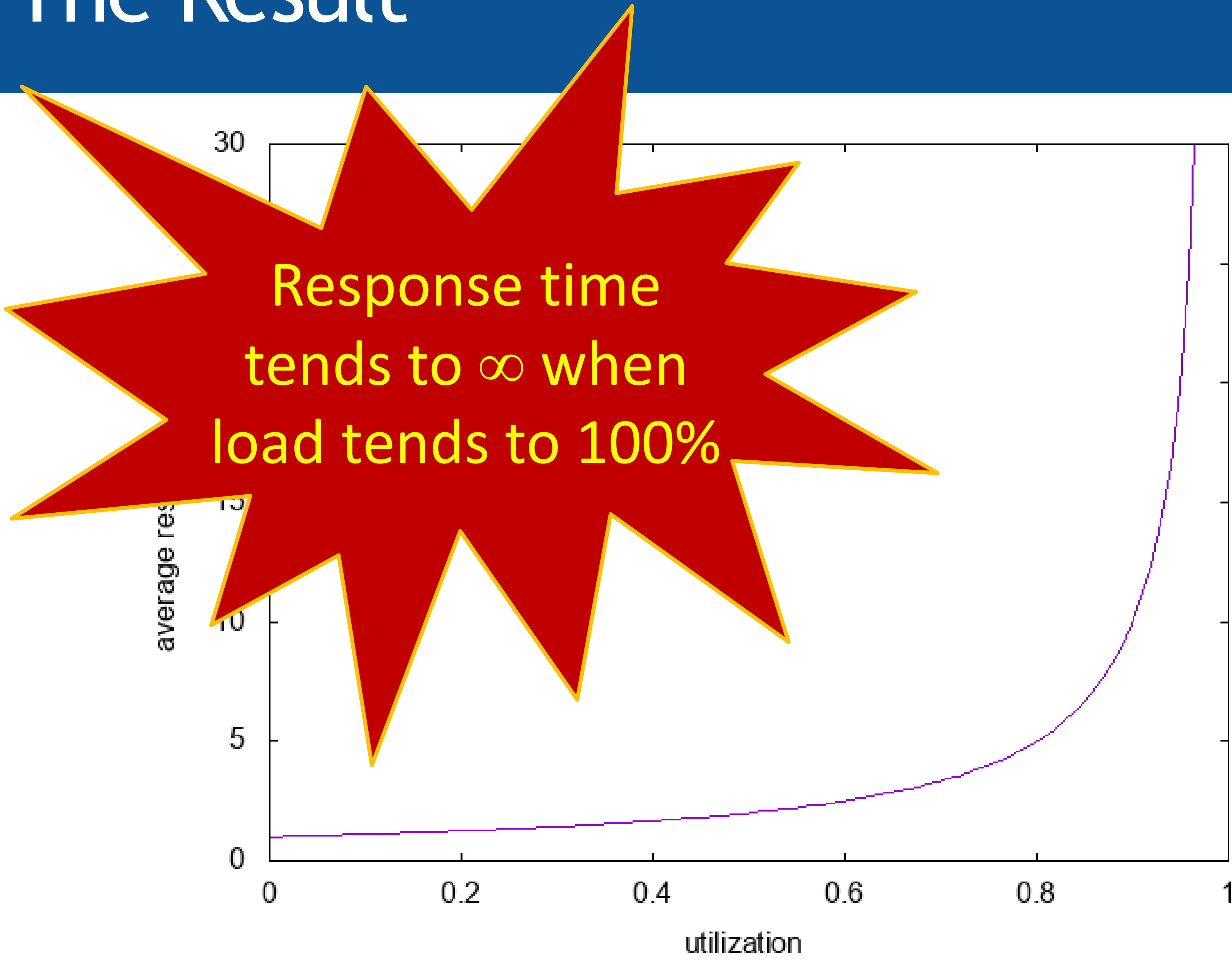
$$\bar{r} = \frac{\bar{n}}{\lambda} = \frac{\rho}{\lambda(1-\rho)} = \frac{1/\mu}{1-\rho}$$

The Result



$$\bar{r} = \frac{1/\mu}{1-\rho}$$

The Result



Response time tends to ∞ when load tends to 100%
30
15
10
5
0
average res
0
0.2
0.4
0.6
0.8
1
utilization

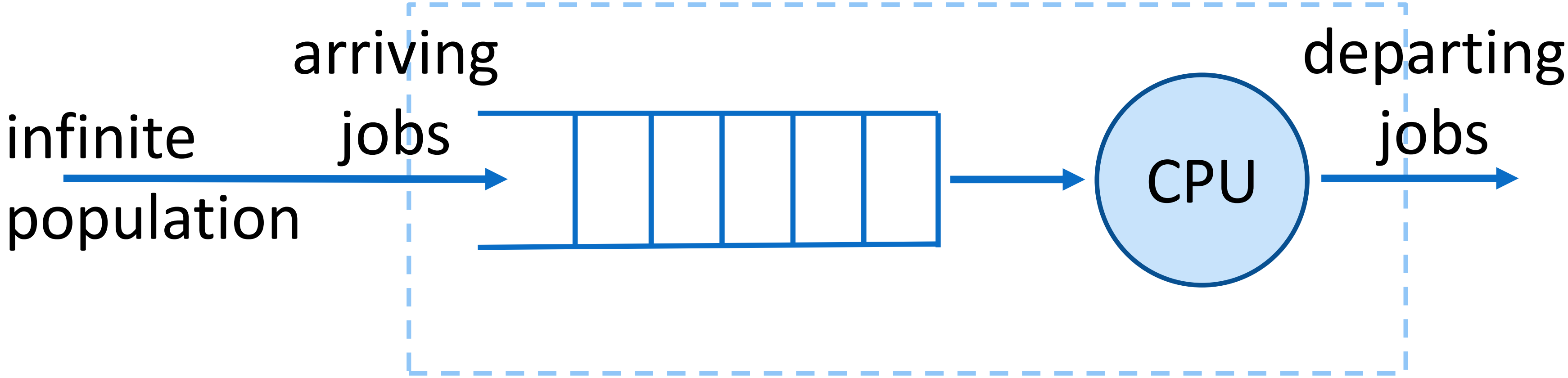
The Implications

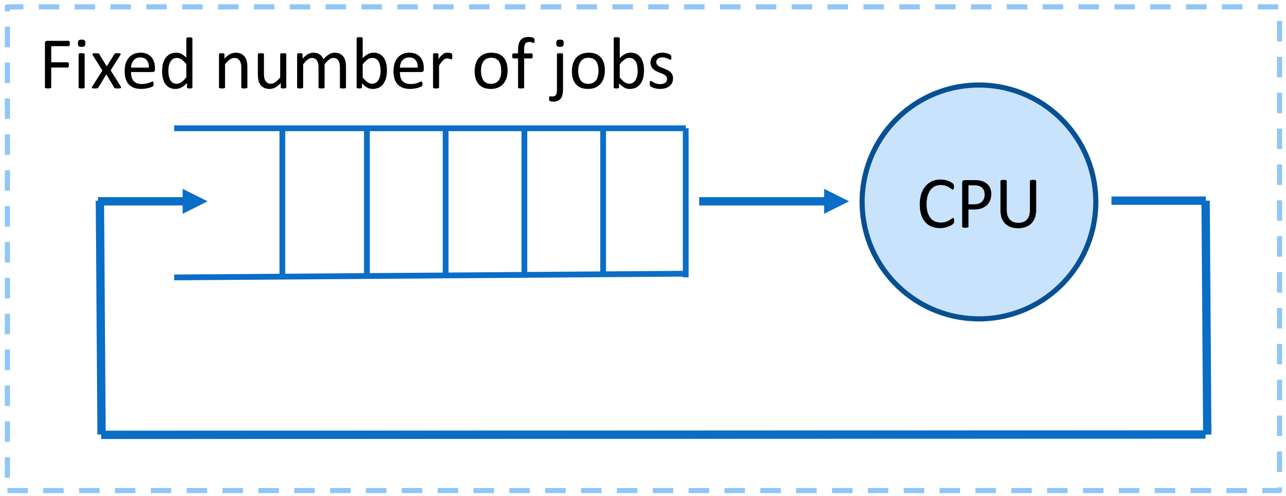
Starvation is not binary

- ➢ With higher load
- ➢ And more skewed the distributions

- There is a higher danger to wait a long time
- Need to leave buffer and not reach 100% utilization

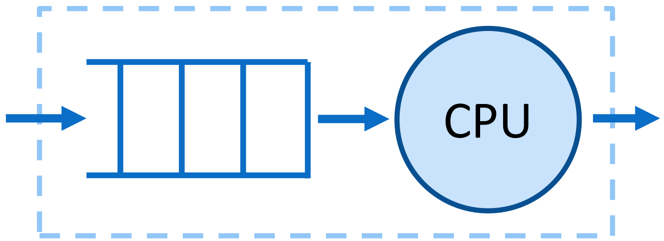
Open vs. Closed System Model





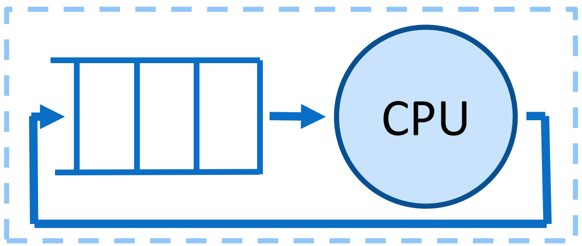
Open vs. Closed System Model

Open system



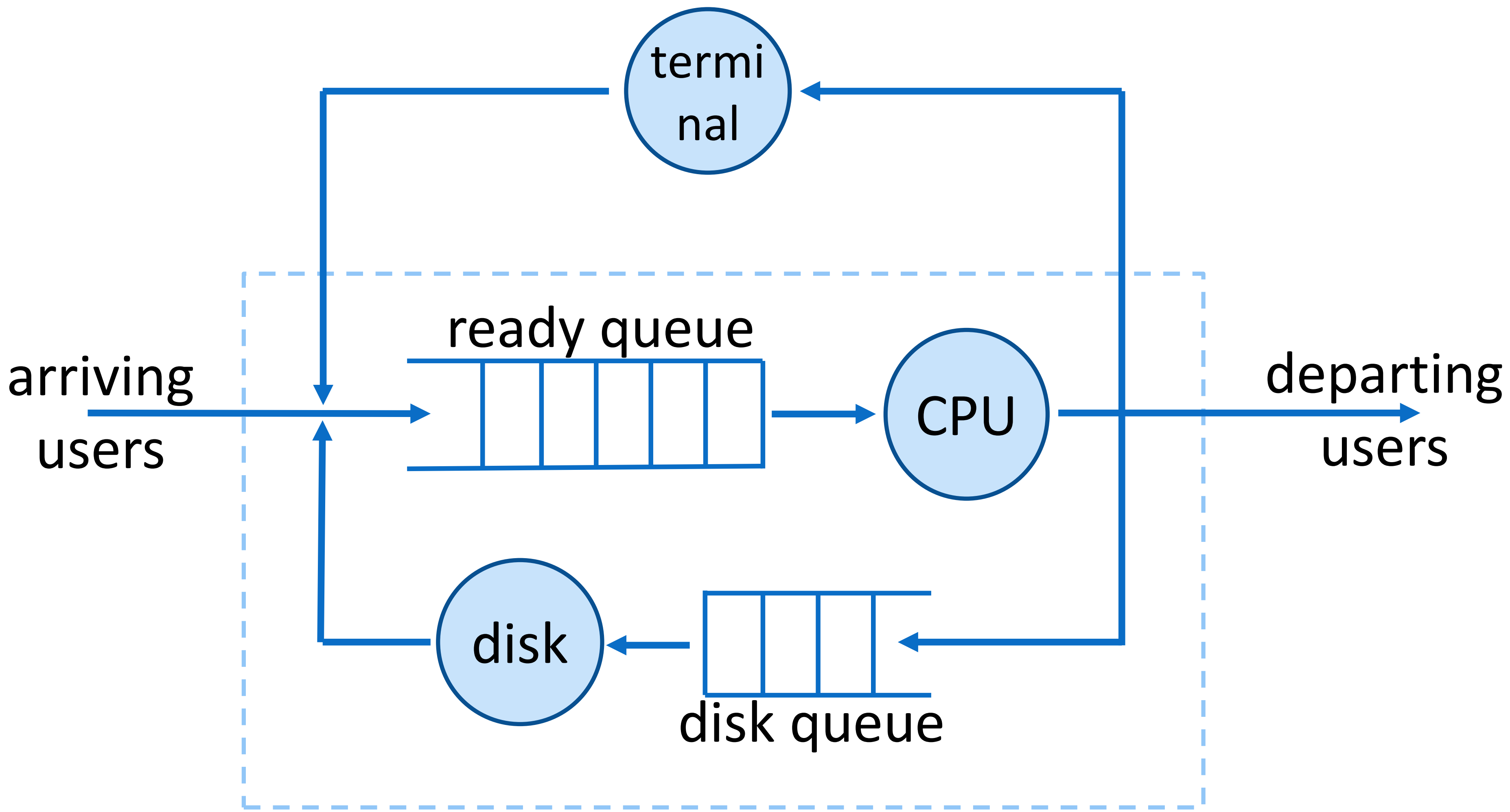
- No feedback from performance to jobs
- Fluctuating number of jobs
- Load < 100%
- Response time metric
- Example: web server

Closed system



- Feedback from performance to jobs
- Fixed number of jobs
- Load = 100%
- Throughput metric
- Example: controller

Combined Model



Bottlenecks

- In a queueing network performance is dictated by the bottleneck device
- If jobs are compute bound, the CPU is the bottleneck
  - And the disk is mostly idle
- If jobs are I/O bound, the disk is the bottleneck
  - And the CPU is mostly idle
- Only Scheduling of the bottleneck device is important

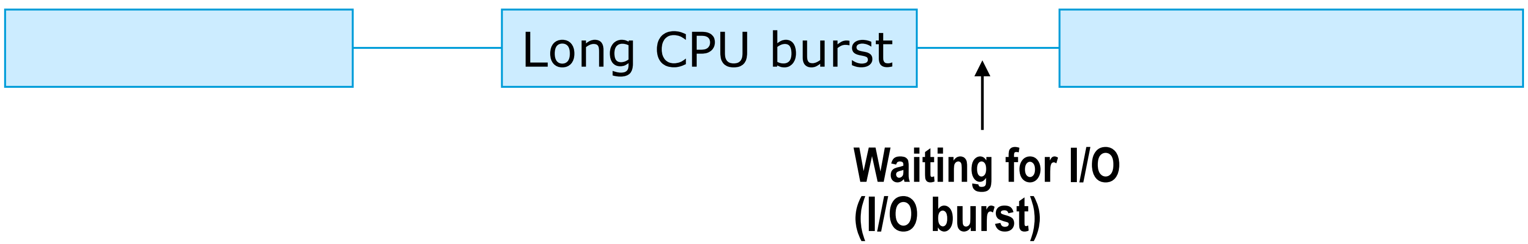
OTHER SCHEDULING CONTEXTS

Long-Term Scheduling

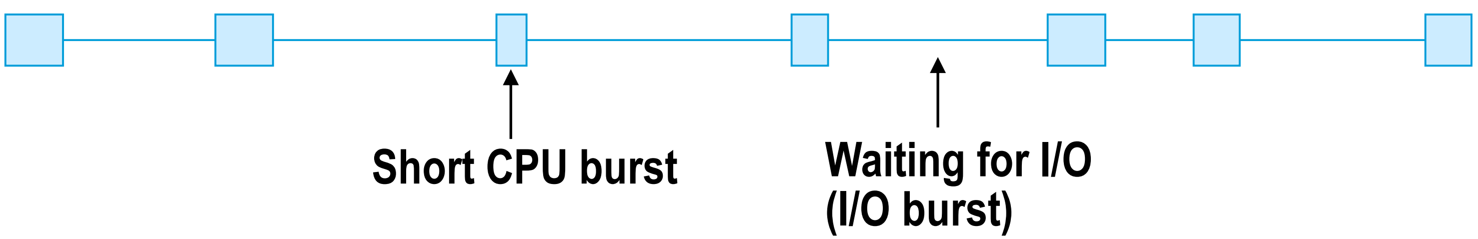
- Processes need memory space to run
- What if there is not enough for all of them?
- Need to **swap out** some of them
  – Store temporarily on disk and return later
- Considerations:
  – Keep interactive jobs
  – Create good job mix (complementary use of resources)

Job/Process Characterization

- CPU Bound: spends most of the time doing computations; few very long CPU bursts.



- I/O Bound: spends more time doing I/O than computations, many short CPU bursts.



A Good Job Mix

- The system has multiple resources
- Any one of them can become a **bottleneck**
  - The CPU if all jobs are compute-bound
  - The disk is all jobs are I/O bound
- When jobs wait for the bottleneck, other resources are left under-utilized
- A good mix uses the system more effectively
  - CPU bound jobs utilize the CPU
  - I/O bound jobs utilize the disk at small cost of CPU

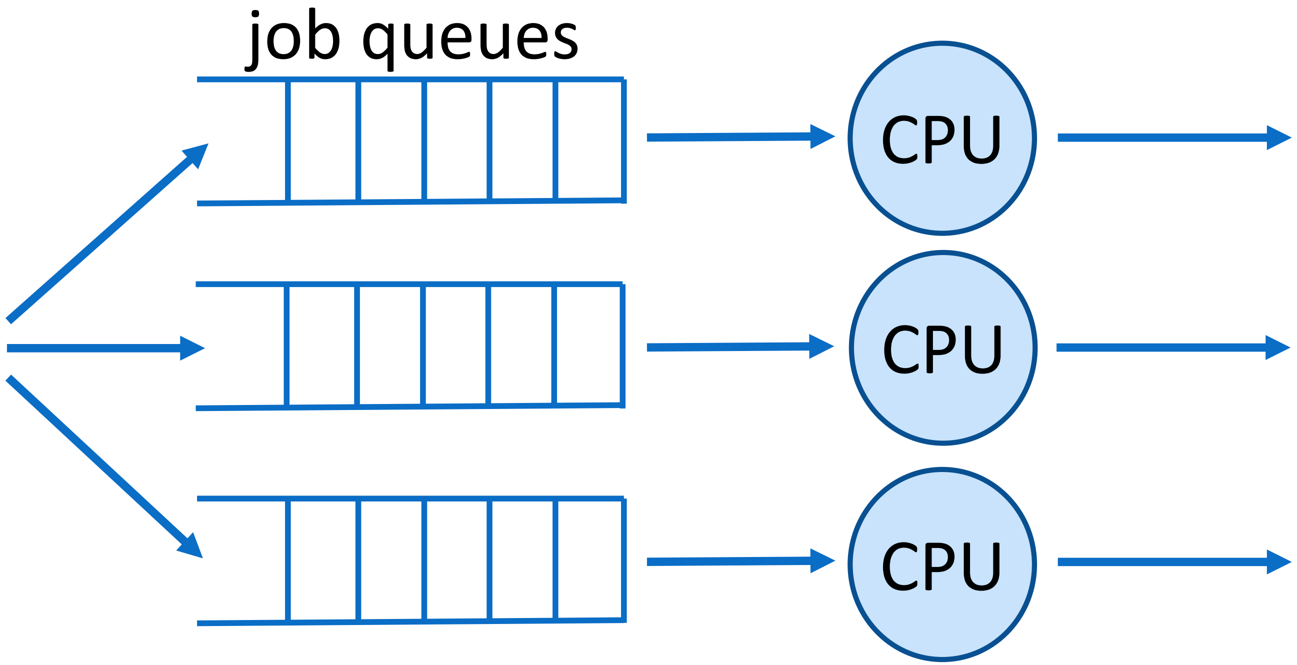
Fair Share Scheduling

- "Fair" does not necessarily mean "equal"
  - Fair = according to allocation
- Virtual time scheduling
  - for each process compute the ratio of actual CPU time consumed to CPU time entitled
  - run the process with the lowest ratio
- Lottery scheduling
  - give processes lottery tickets for various system resources, such as CPU time
  - Number of tickets reflects allocation
  - A lottery ticket is chosen at random, and the process holding the ticket gets the resource

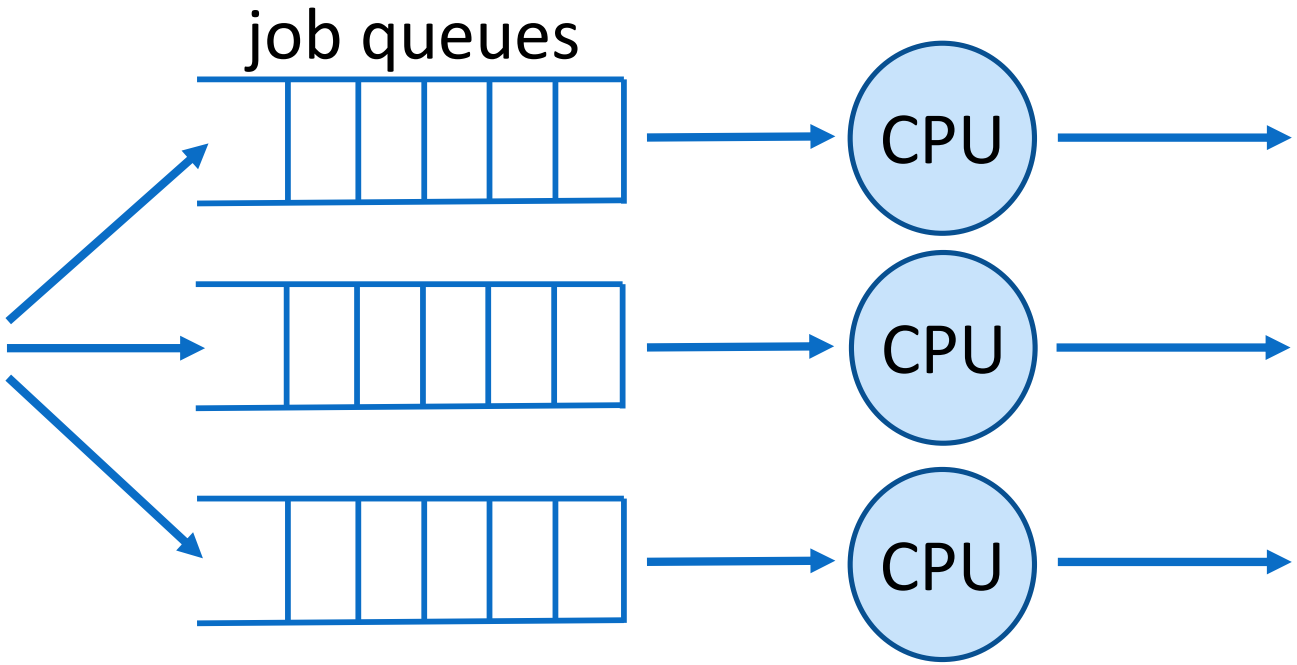
Multiple-Processor Scheduling

- CPU scheduling is more complex when multiple CPUs are available
- System model:
  - Homogeneous processors within a multiprocessor
  - All the memory is accessible from all processors
  - Any processor can run any job
- Two options:
  - Separate queue per CPU (supermarket)
  - Shared queue (bank / post office)

Separate Queues

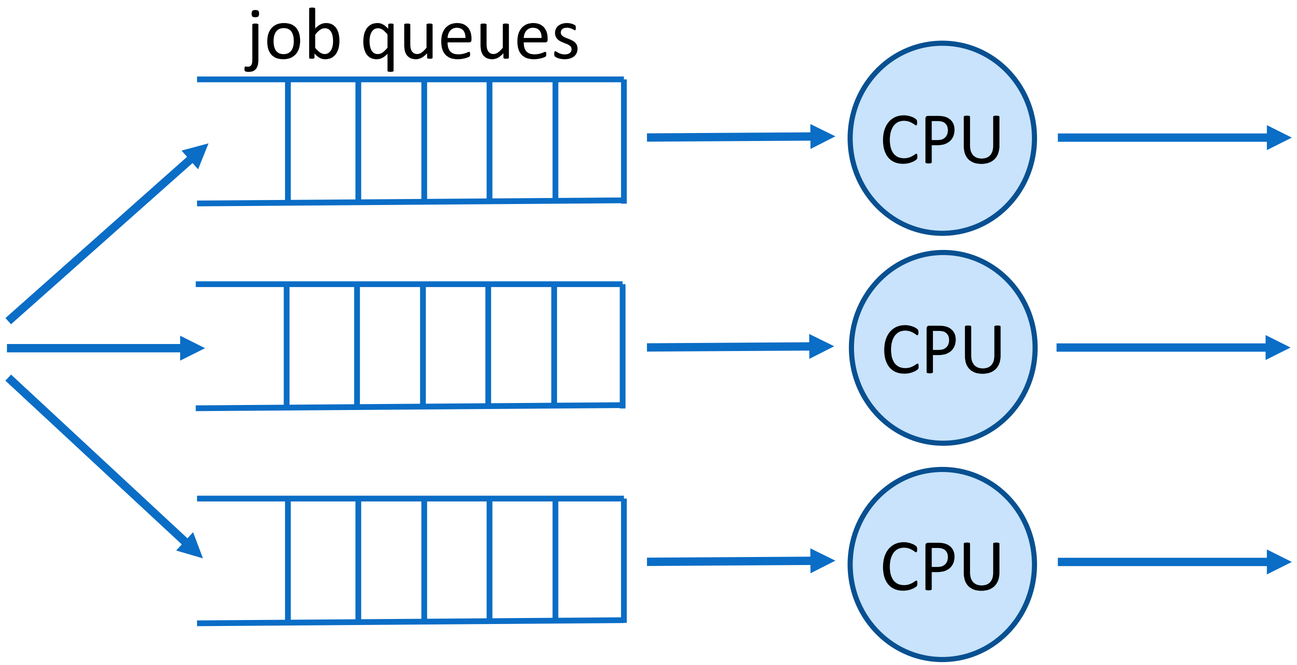


Separate Queues



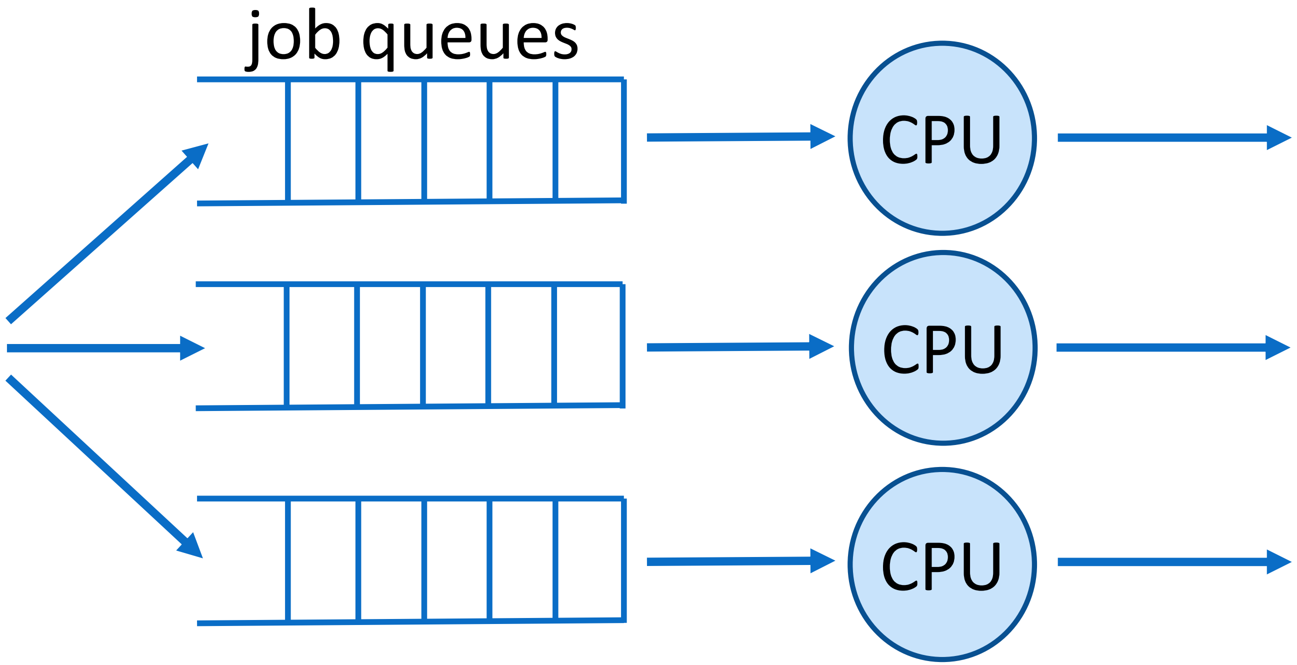
- Load balancing
  - E.g. “join shortest queue” discipline

Separate Queues



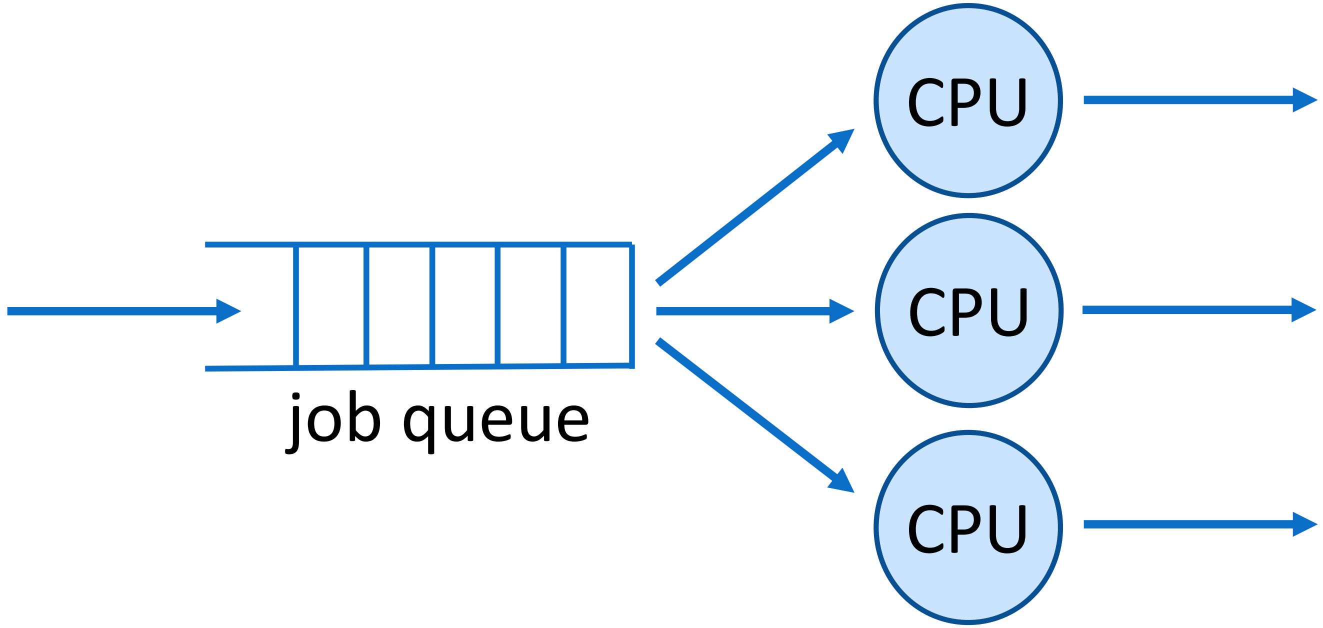
- Load balancing
  - E.g. “join shortest queue” discipline
- But loads may change
  - Could have jobs in one queue while another CPU is idle

Separate Queues

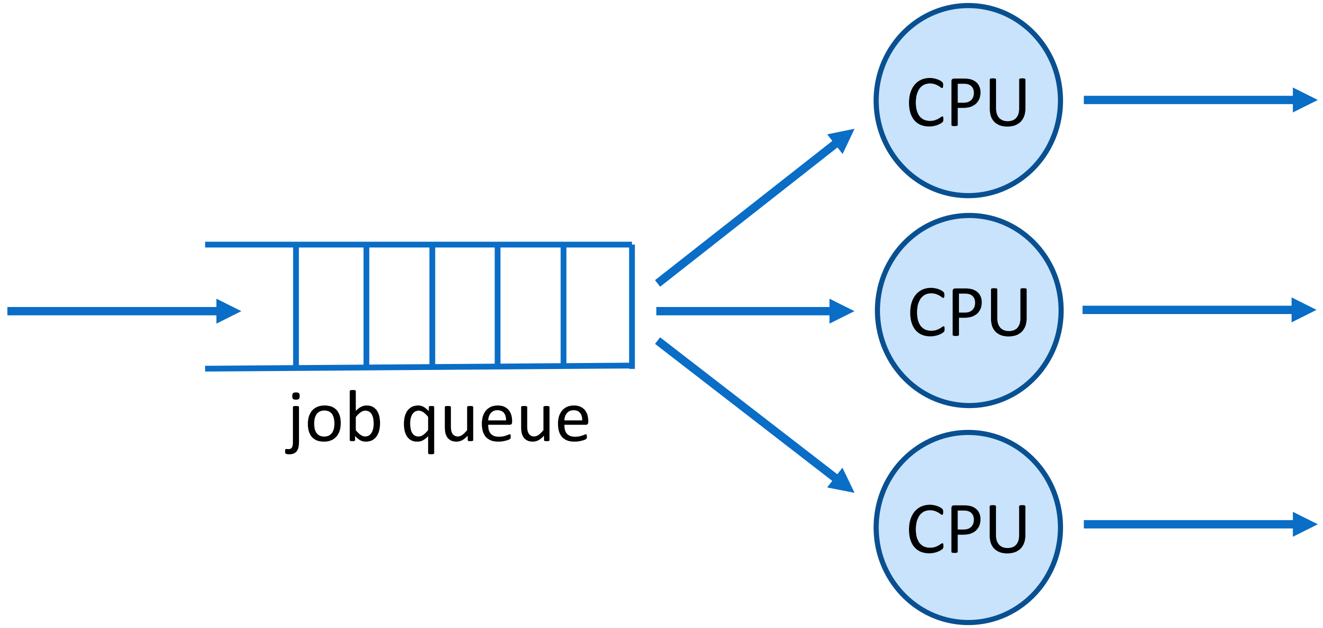


- Load balancing
  - E.g. “join shortest queue” discipline
- But loads may change
  - Could have jobs in one queue while another CPU is idle
  - Optional special queue for short jobs

Shared Queue

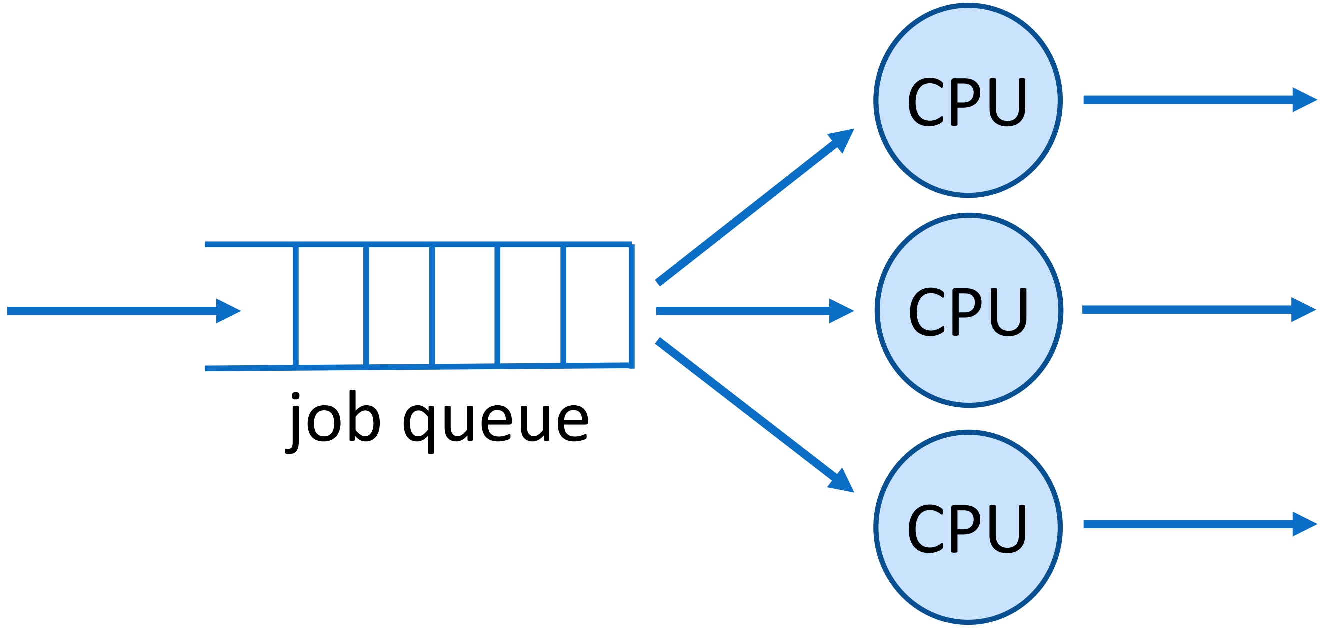


Shared Queue



- Load sharing
  - Dispatch jobs to first available processor

Shared Queue



- Load sharing
  - Dispatch jobs to first available processor
- Affinity scheduling
  - Prefer previously used processor which may have relevant cache state

Real-Time Scheduling

- Hard real-time systems:
  required to complete a critical task within a guaranteed amount of time
  – E.g. critical control tasks like flying a plane
- Soft real-time computing:
  requires that critical processes receive priority over less fortunate ones
  – E.g. display a video of a cat
- Tasks are often periodic

Earliest Deadline First (EDF)

- Given: n tasks with interarrival periods T_i and worst-case computation time c_i
- Schedulability test:

$$Utilization = \sum_{i=1}^{n} \frac{c_i}{T_i} \leq 1$$

- At each event scan the queue and schedule the task with the earliest deadline
- EDF guarantees that all tasks are scheduled and all deadlines met