Scheduling

Scheduler tasks and goals

Process states

Pre-emption

Scheduling strategies

- First come first served, Round robin (time sliced),
  Shortest job first, Priority based,
  Multi level feedback queues, Lottery

Thread scheduling

Scheduler Tasks

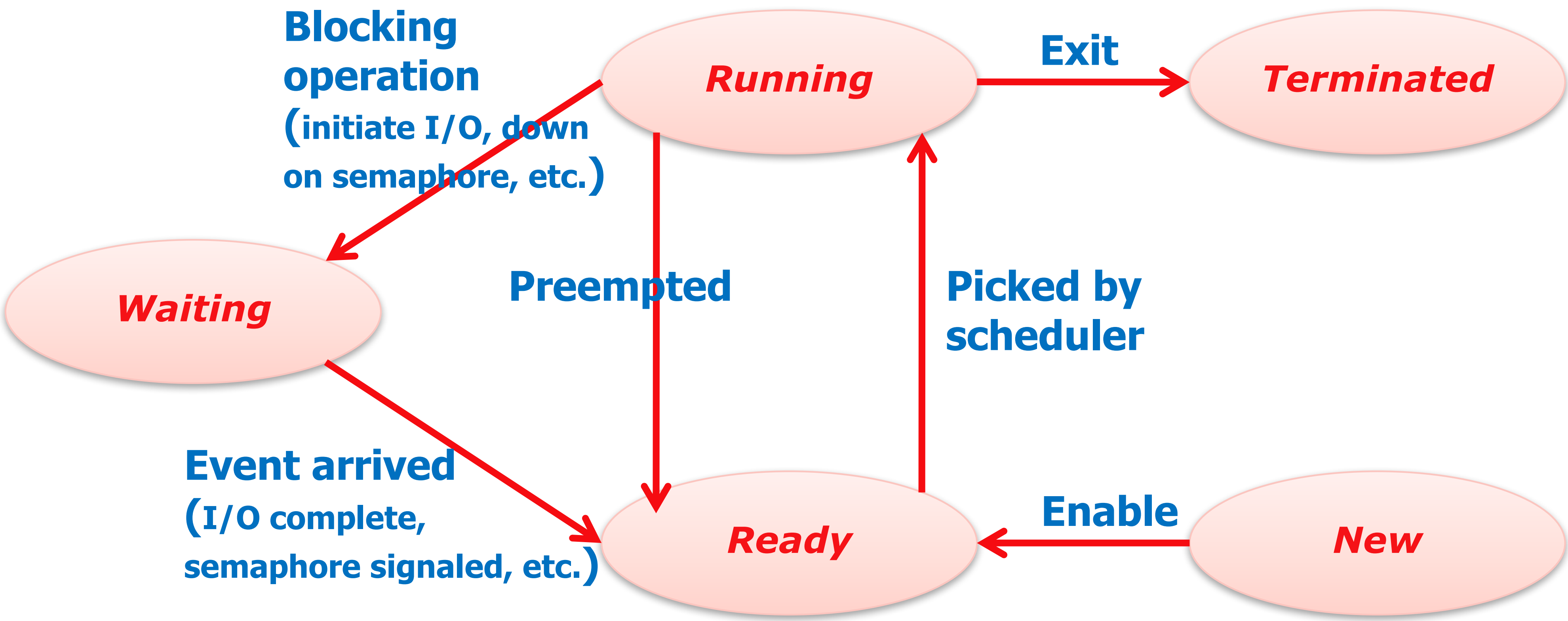
The scheduler

- Allocates processes to processors.
- Selects highest priority ready process (from head of Ready Queue) and moves it to the running state, i.e., allows it to start executing on the processor.
- Gets invoked after every entry to the kernel.

Current process continues unless:

- Kernel call moved it into waiting state (e.g. waiting on I/O).
- Error trap occurred (e.g. memory protection violation).
- Time slice expired.
- A higher priority process is made ready.

Process States



- **New**: the process is being created
- **Ready**: runnable and waiting for processor
- **Running**: executing on a processor
- **Waiting/Blocked**: waiting for an event
- **Terminated**: process is being deleted

If multiple processes are ready, which one should be run?

Menti.com Q1 42 63 05

Goals of Scheduling Algorithms

Ensure fairness
  – Comparable processes should get comparable services

Avoid indefinite postponement
  – No process should starve

Enforce policy
  – E.g., priorities

Maximize resource utilization
  – CPU, I/O devices

Minimize overhead
  – From context switches, scheduling decisions

Goals of Scheduling Algorithms

Batch systems:

- Throughput → maximize jobs per unit of time
- Turnaround time → minimize time between job submission and termination
- Maximize CPU utilization

Interactive systems:

- Response time crucial → quick response to requests
- Meet users expectations → predictability

Real-time systems:

- Meeting deadlines
  - Soft deadlines: e.g., leads to degraded video quality
  - Hard deadline: e.g., leads to plane crash
- Predictability

Preemptive vs. Non-Preemptive Scheduling

Non-preemptive

- Let process run until it blocks or voluntarily releases the CPU

Preemptive:

- Let process run for a maximum amount of fixed time
  - Requires clock interrupt
- External event results in higher priority process being run

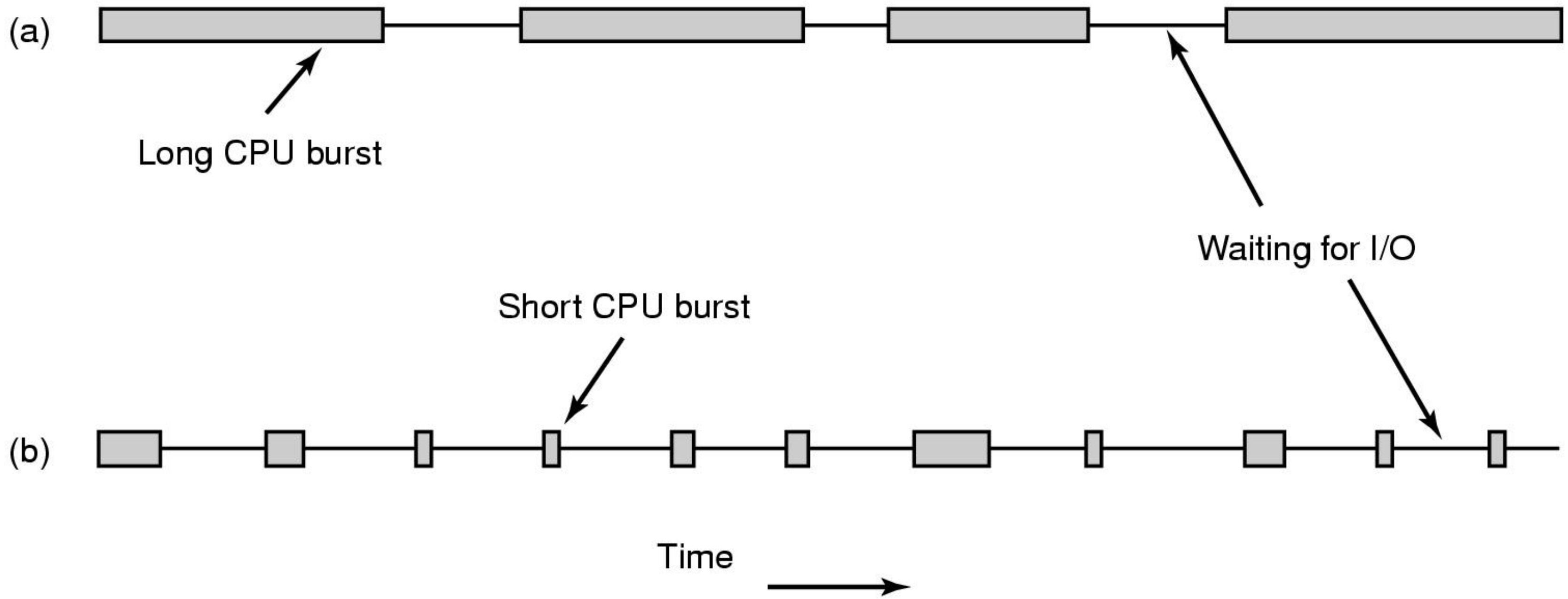
CPU-bound vs. I/O-bound Processes

CPU-bound processes

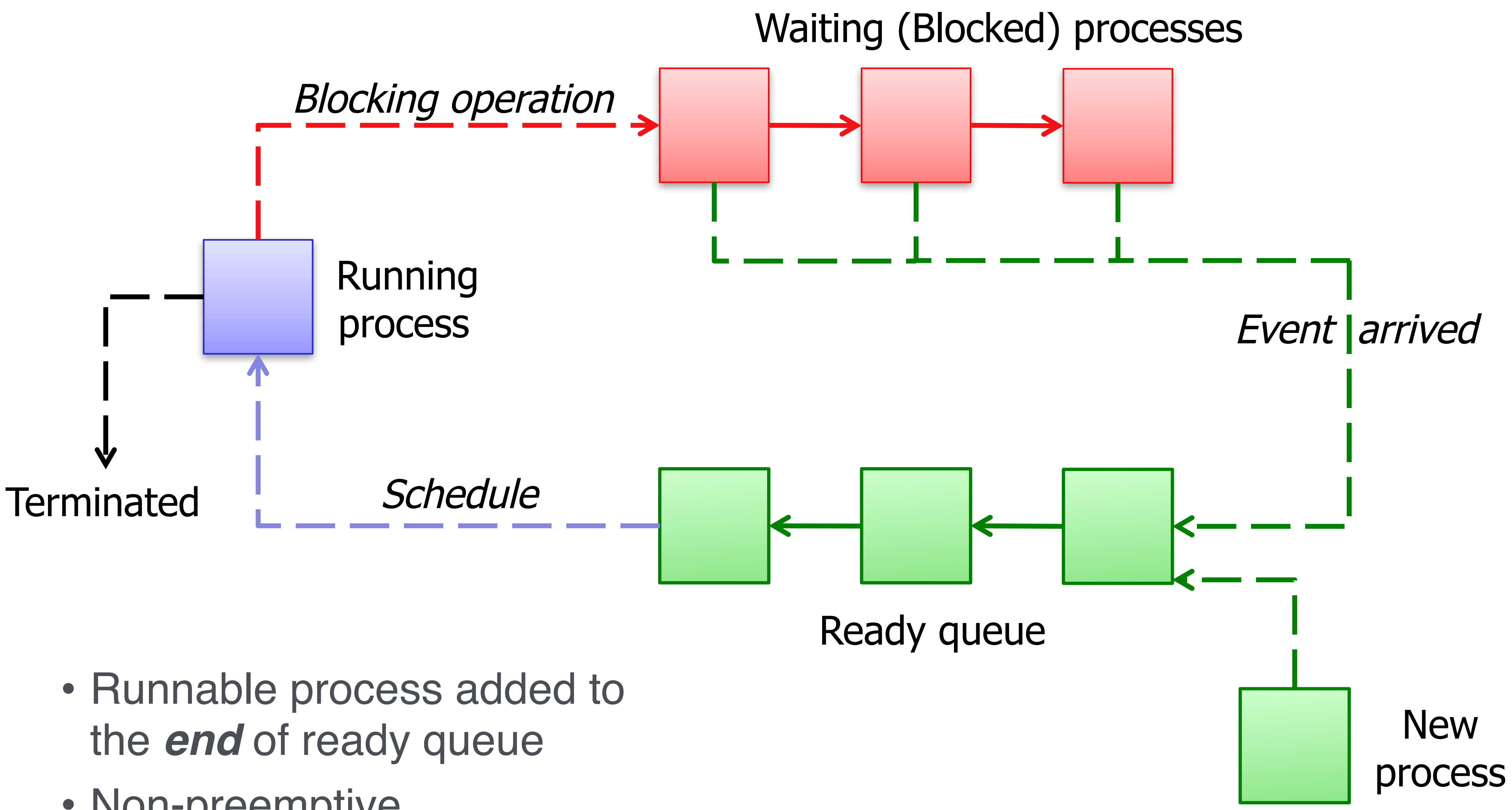
- Spend most of their time using the CPU

I/O-bound processes

- Spend most of their time waiting for I/O
- Tend to only use CPU briefly before issuing I/O request



First-Come First-Served (FCFS) (non-preemptive)



- Runnable process added to the ***end*** of ready queue
- Non-preemptive

FCFS Advantages

- No indefinite postponement
  - All processes are eventually scheduled
- Really easy to implement

FCFS Disadvantages

What happens if a long job is followed by many short jobs?

- E.g., 1h, 1s, 1s, 1s, with jobs 2-4 submitted just after job 1

3600s	*1s*	*1s*	*1s*

- Throughput?
- Average turnaround time?

1s	*1s*	*1s*	*3600s*

- Throughput?
- Average turnaround time?

Menti.com Q2,3 42 63 05

FCFS Disadvantages

What about I/O bound processes?

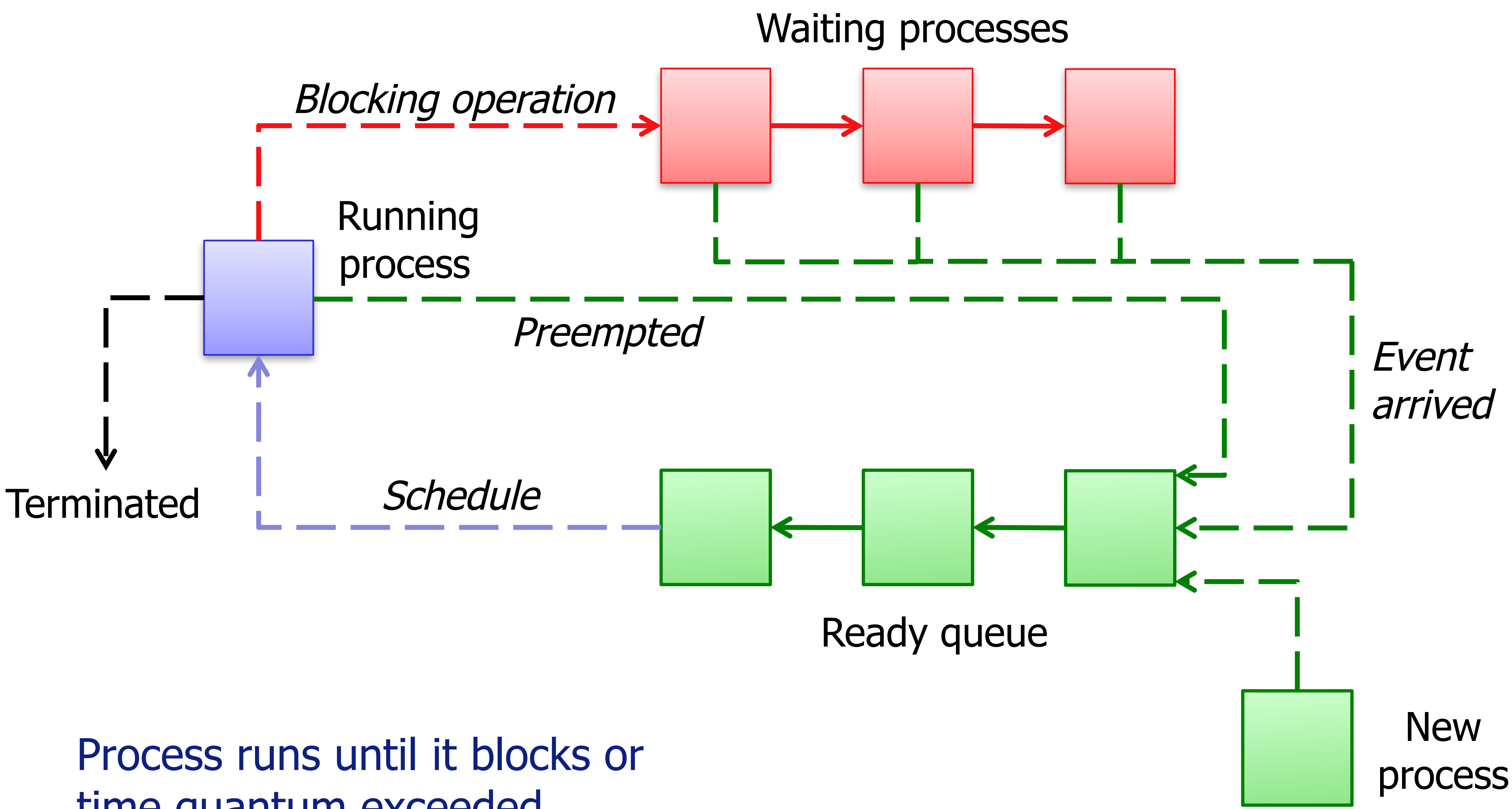
E.g., one CPU-bound process runs for 1s each time, before doing I/O to release processor. Needs > 1000s CPU time

Many I/O-bound process needs to perform 1000 disk reads to complete, with minimal processing between I/O

Compute process runs for 1 sec then starts read so I/O process can then run and start their read, but only do 1 read per sec.

- I/O bound processes initiates 1 request at a time?
  - 1000s to complete
- Preempting CPU-bound process every 10ms?
  - 10s to complete

Round-Robin Scheduling (RR)



Process runs until it blocks or time quantum exceeded

Round-Robin

Fairness

- Ready jobs get equal share of the CPU

Response time

- Good for small number of jobs

Average turnaround time:

- Low when run-times differ
- Poor for similar run-times

```
Quantum = 100ms,
Context switch time = negligible
```

```
A: 200ms, B: 10s
Turnaround time:
  FCFS:   A = 200ms, B = 10,200ms
          Avg = 5200ms
  RR:     A ≈ 300ms, B ≈ 10,200ms
          Avg ≈ 5250m → 1.01x
```

```
A: 10s, B: 10s
Turnaround time:
  FCFS:   A = 10s, B = 20s
          Avg = 15s
  RR:     A ≈ 20s, B ≈ 20s
          Avg ≈ 20s → 1.33x
```

RR Quantum (Time Slice)

RR Overhead:

- 4ms quantum, 1ms context switch time:
  20% of time → overhead high
- 1s quantum, 1ms context switch time:
  only 0.1% of time → overhead low

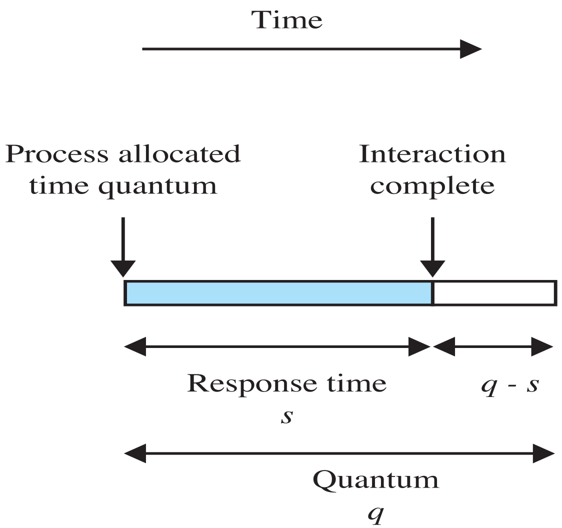
Large quantum:

- Smaller overhead
- Worse response time
  - Quantum = ∞ → FCFS

Small quantum:

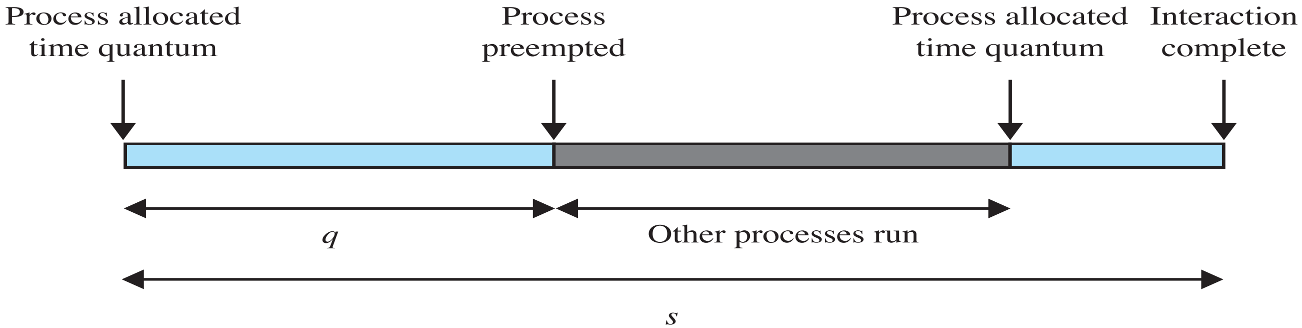
- Larger overhead
- Better response time
- Ideal quantum ≈ ave. CPU time between I/O

Time quantum vs I/O times



(a) Time quantum greater than typical interaction

If time quantum is too small, most processes will require > 1 quantum to complete, which increases response time.



(b) Time quantum less than typical interaction

RR Quantum

Choosing a quantum value:

– Should be much larger than context switch cost
– But provide decent response time

Typical values: 10-200ms

Some example values for standard processes (values vary depending on process type and behaviour, priority, etc.):

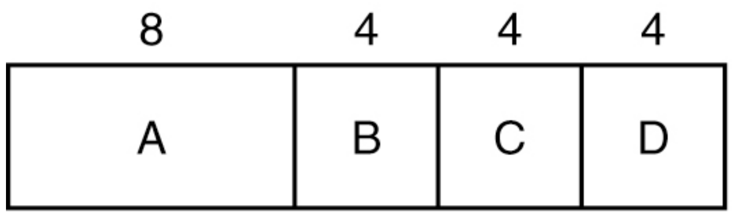
– Linux: 100ms
– Windows client: 20ms
– Windows server: 180ms

Shortest Job First (SJF)

Non-preemptive scheduling with run-times known in advance

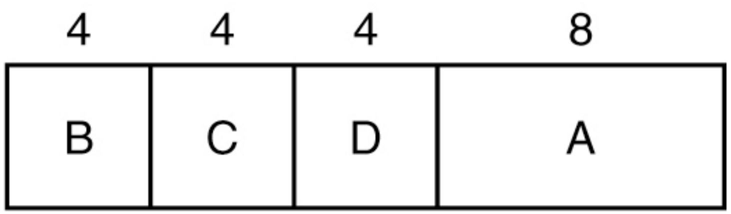
Pick the shortest job first

- Process with shortest CPU burst



Turnaround time:

```
  A:  8s
  B: 12s
  C: 16s
  D: 20s
Avg: 56/4 = 14s
```



Turnaround time:

```
  B:  4s
  C:  8s
  D: 12s
  A: 20s
Avg: 44/4 = 11s
```

- Provably optimal when all jobs are available simultaneously

Shortest Remaining Time (SRT)

Preemptive version of shortest job first

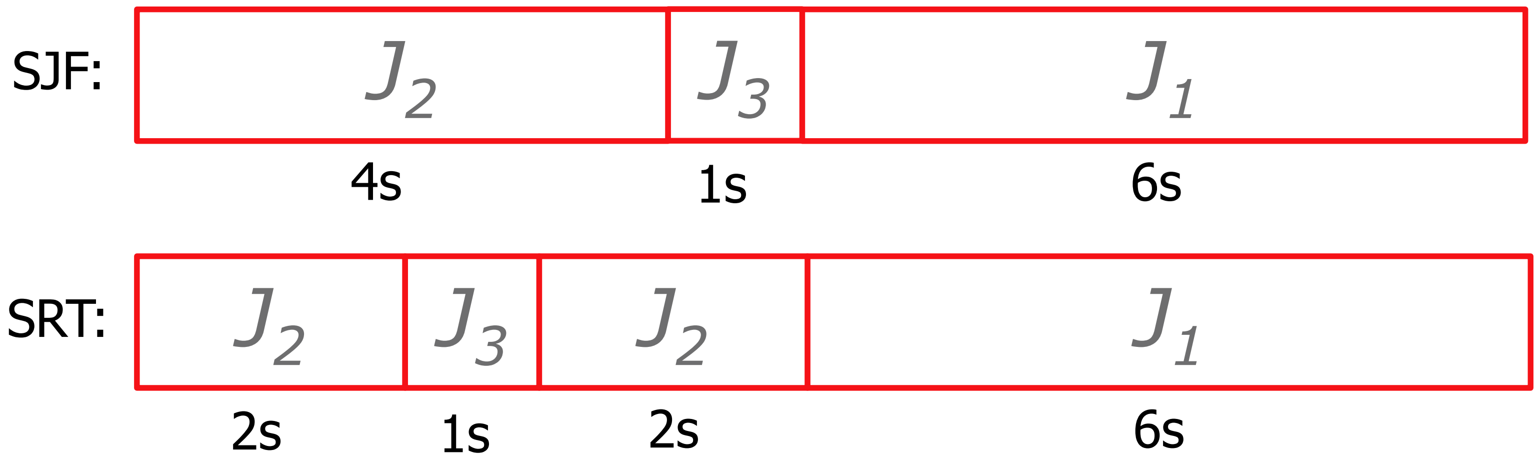
- Again, runtimes have to be known in advance

Choose process whose remaining time is shortest

- When new process arrives with execution time less than the remaining time for the running process, run it

Allows new short jobs to get good service

Example: 3 jobs: J_1 = 6s, J_2 = 4s, J_3 = 1s, & arrives after 2s



Shortest Remaining Time (SRT)

What if a running process is almost complete and a shorter job arrives?

- Might want to disallow preemption when remaining run-time reaches a low threshold to avoid indefinite postponement
- What if context switch overhead is greater than the difference in remaining run-times for the two jobs?

Knowing Run-times in Advance

Run-times are usually not available in advance

Compute CPU burst estimates based on various heuristics?

- E.g., based on previous history
- Not always applicable

User-supplied estimates?

- Need to counteract cheating to get higher priority
- E.g., terminate or penalize processes after they exceed their estimated run-time

Minimise Turnaround Time

Five jobs are waiting to be run. Their expected run times are 9, 6, 3, 5, and X. In what order should they be run to minimize average turnaround time? (Your answer will depend on X.)

Fair-Share Scheduling

- Users are assigned some fraction of the CPU
  - Scheduler takes into account who owns a process before scheduling it
- E.g., two users each with 50% CPU share
  - User 1 has 4 processes: A, B, C, D
  - User 2 has 2 processes: E, F
- What does a fair-share RR scheduler do?
  - A, E, B, F, C, E, D, F, A, E, B, F…

Priority Scheduling

- Jobs are run based on their priority
  - Always run the job with the highest priority
- Priorities can be externally defined (e.g., by user) or based on some process-specific metrics (e.g., their expected CPU burst)
- Priorities can be static (i.e. they don't change) or dynamic (they may change during execution)

- Example: consider three processes arriving at essentially the same time with externally defined static priorities
  A = 4, B = 7, C = 1, where a higher value means higher priority.
  - Processes are run to completion in the order B, A, C.

General-Purpose Scheduling

Favor short and I/O-bound jobs

- Get good resource utilization
- And short response times

Quickly determine the nature of the job and adapt to changes

- Processes have periods when they are I/O-bound and periods when they are CPU-bound

Multilevel Feedback Queues 1

A form of priority scheduling

- Shortest remaining time also a form of priority scheduling!

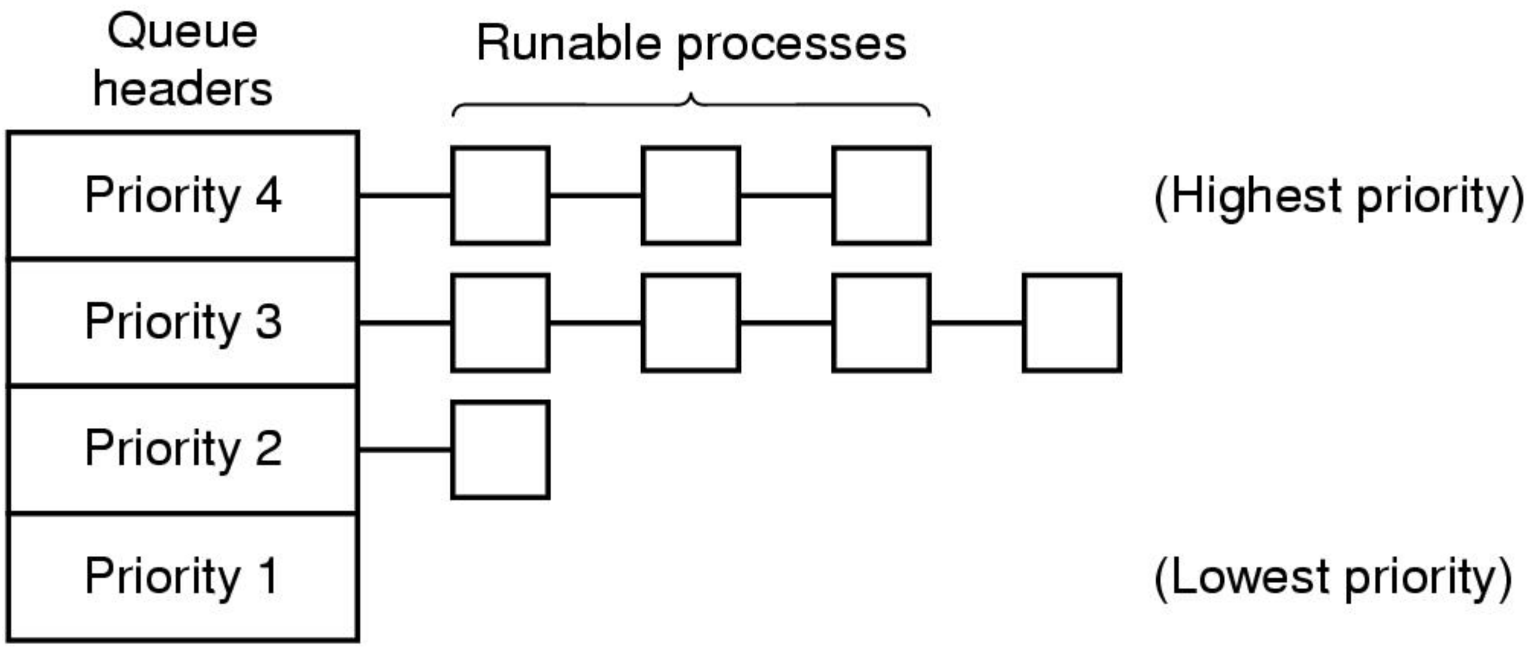
Implemented by many OSs:

- Windows Vista, Windows 7
- Mac OS X
- Linux 2.6 - 2.6.23

Multilevel Feedback Queues 2

One queue for each priority level

- Run job on highest non-empty priority queue
- Each queue can use different scheduling algorithm
  - Usually round-robin
  - Could be different quantum eg highest priority is I/O bound with short quantum. Exceed quantum, then move down level but get bigger quantum.



Multilevel Feedback Queues 3

Need to determine current nature of job

– I/O-bound? CPU-bound?

Need to worry about starvation of lower-priority jobs

Feedback mechanism:

– Job priorities recomputed periodically, e.g., based on how much CPU they have recently used
  - Exponentially-weighted moving average
– *Aging*: increase job's priority as it waits

Multilevel Feedback Queues 4

- Not very flexible
  - – Applications basically have no control
  - – Priorities make no guarantees
    - What does priority 15 mean?
- Does not react quickly to changes
  - – Often needs *warm-up* period
    - Running system for a while to get better results
  - – Problem for real-time systems, multimedia apps
- Cheating is a concern
  - – Add meaningless I/O to boost priority?
- Cannot donate priority

Menti.com Q4 42 63 05

Single vs Multithreaded File server

Assume single processer & takes 15 ms to get a request for work, dispatch it, and do the rest of the necessary processing, assuming that the data needed are in the block cache.

If a disk operation is needed, as is the case *one-third* of the time, an additional 75 ms is required, during which time the thread sleeps.

How many requests/sec can the server handle if it is single-threaded? If it is multithreaded?

(a) non-preemptive scheduler,

(b) a preemptive round robin scheduler with a small quantum > 25ms.

Lottery Scheduling [Waldspurger and Weihl 1994]

Jobs receive lottery tickets for various resources

- E.g., CPU time

At each scheduling decision, one ticket is chosen at random and the job holding that ticket wins

Example: 100 lottery tickets for CPU time, P1 has 20 tickets

- Chance of P1 running during the next CPU quantum: 20%
- In the long run, P1 gets 20% of the CPU time

Lottery Scheduling

- Number of lottery tickets meaningful
  - Job holding p% of tickets, gets p% of resource
  - Unlike priorities
- Highly responsive:
  - New job given p% of tickets has the p% chance to get the resource at the ***next*** scheduling decision
- No starvation
- Jobs can exchange tickets
  - Allows for priority donation
  - Allows cooperating jobs to achieve certain goals
- Adding/removing jobs affect remaining jobs proportionally
- Unpredictable response time
  - What if interactive process is unlucky for a few lotteries?

Policy versus Mechanism

Separate what is allowed to be done from how it is done

– a process knows which of its children threads are important and need priority

Scheduling algorithm parameterized

– mechanism in the kernel

Parameters filled in by user processes

– policy set by user process

Scheduling Questions

Five batch jobs, A through E, arrive at a computer centre at essentially the same time. Their estimated running time are as follows: A=15min, B=9min, C=3min, D=6min and E=12min.

Their (externally defined) priorities are:

A = 6, B = 3, C = 7, D = 9 and E = 4, with a *lower* value corresponding to a *higher* priority.

For each of the following scheduling algorithms, determine the turnaround time for each job, and the average turnaround time for all jobs.

Ignore process switching overhead and assume all jobs are completely CPU bound.

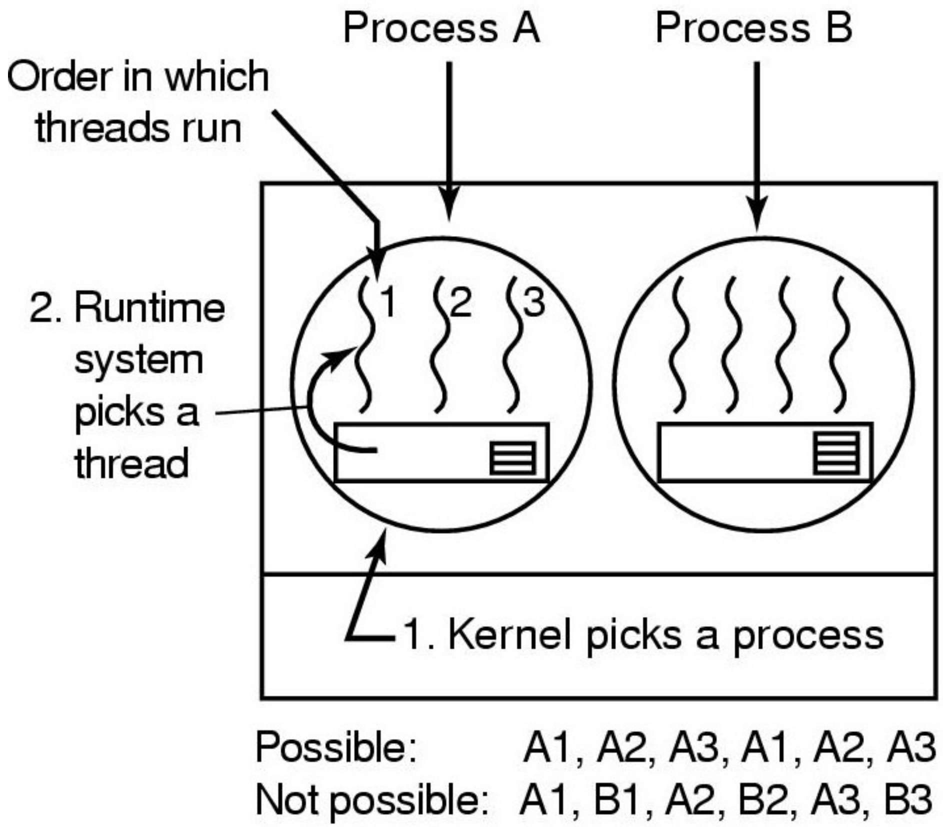
a) Non-preemptive priority scheduling.

b) FCFS (run in order A,B,C,D)

c) Shortest job first (SJF)

d) Round robin with a time quantum of 1 minute

User Thread Scheduling

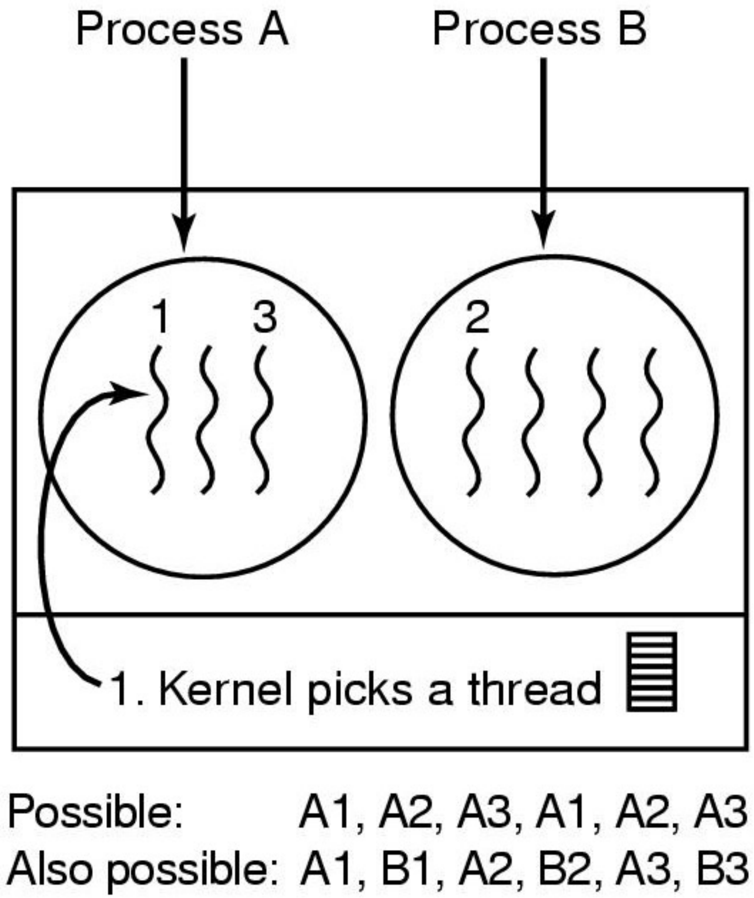


Possible scheduling of user-level threads

50-msec process quantum

Threads run 5 msec/CPU burst

Kernel Thread Scheduling



Possible scheduling of kernel-level threads
50-msec process quantum
Threads run 5 msec/CPU burst

Summary

- Scheduling algorithms often need to balance conflicting goals
  - E.g., ensure fairness, enforce policy, maximize resource utilization
- Different scheduling algorithms appropriate in different contexts
  - E.g., batch systems vs interactive systems vs real-time systems
- Well-studied scheduling algorithms include
  - First-Come First-Served FCFS, Round Robin, Shortest Job First (SJF), Shortest Remaining Time (SRT), Multilevel Feedback Queues and Lottery Scheduling

Menti.com Q5 42 63 05