TDDB68/TDDE47 Concurrent programming and operating systems

Lecture:
CPU Scheduling

Klas Arvidsson

Slides by Adrian Pop and Mikael Asplund

Reading

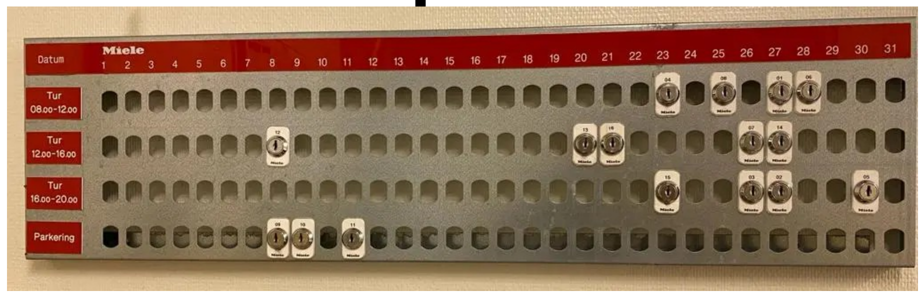
- Silberschatz et al. 9th and 10th editions
  - Chapter 5.1-5.5, 5.8

- Lab notice 2024:
  - The function “poweroff” mentioned in lab material has been renamed in the Pintos codebase.
  - The new name is “shutdown” and it resides in “devices/shutdown.h”

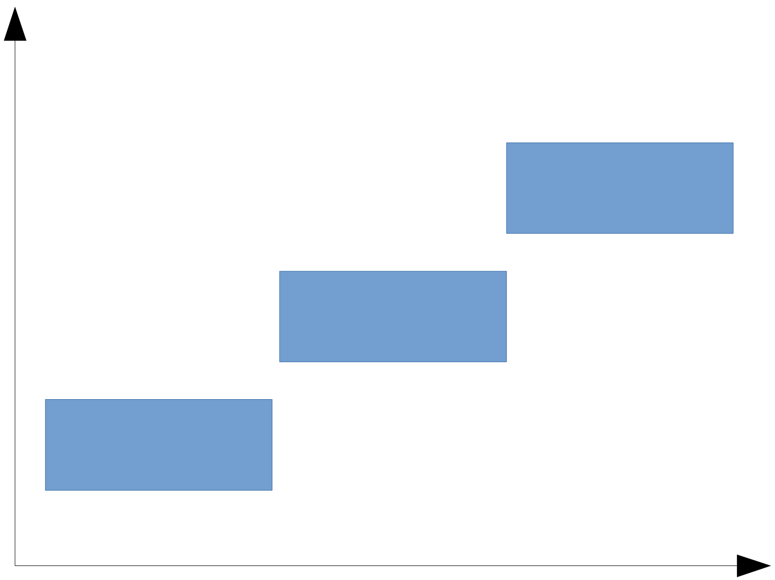
Scheduling

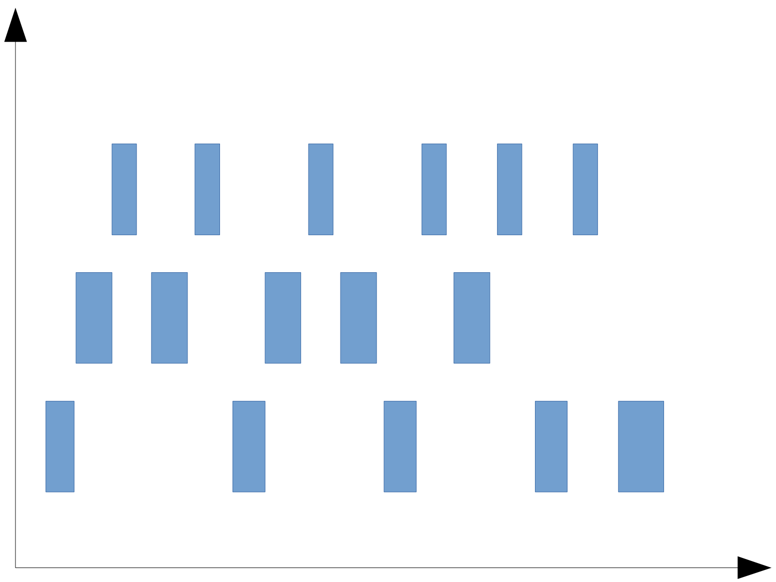
- A form of resource allocation
- Resources
  - CPU
  - Bus
  - Router
  - ...
- Demand exceeds resources

Related problems



Non-preemptive vs preemptive





Examples:
Network transmissions
I/O operations
Atomic operations

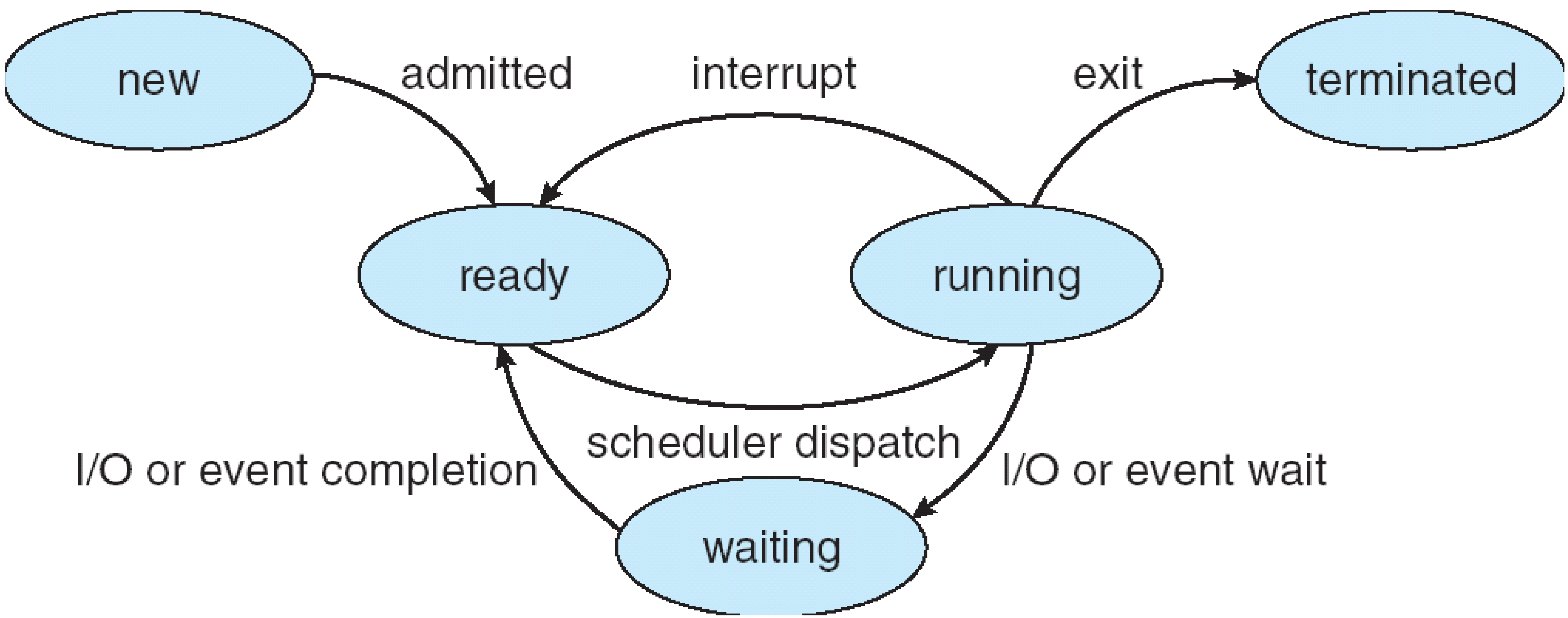
Examples:
Multitasking

Static vs dynamic scheduling

- Static (off-line)
  - complete a priori knowledge of the task set and its constraints is available
  - hard/safety-critical system

- Dynamic (on-line)
  - partial taskset knowledge, runtime predictions
  - firm/soft/best-effort systems, hybrid systems

Recall: Process states



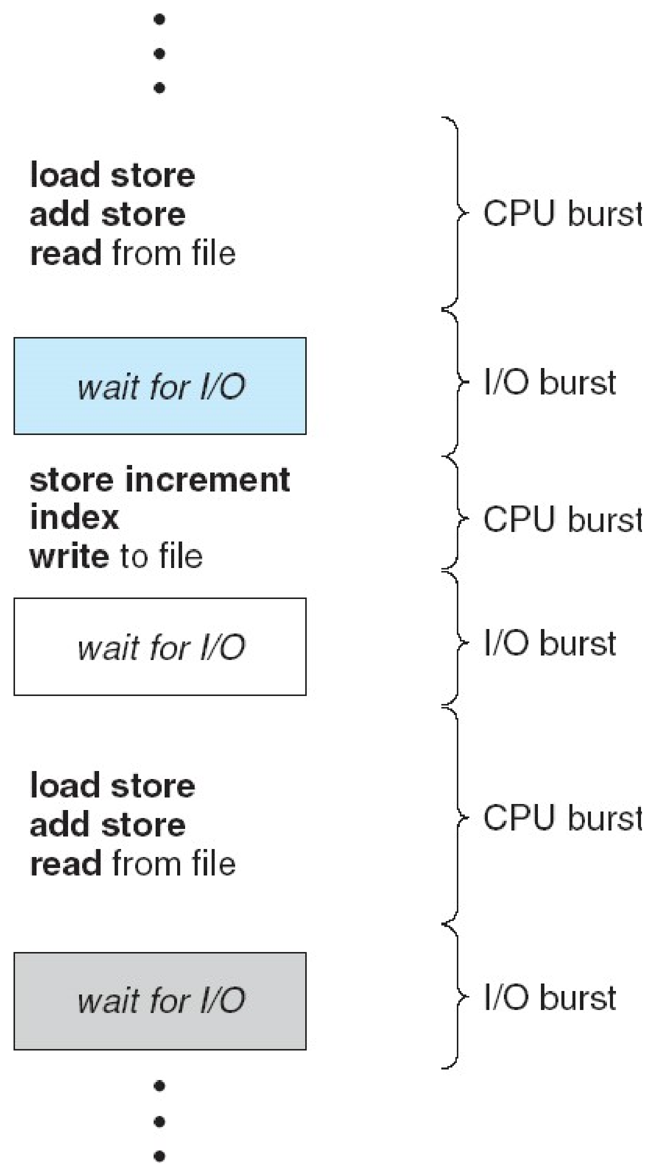
Scheduler

- Resides in the kernel
  - Operating system is responsible for managing processes

- Periodically called by a **timer interrupt**

- Length of timer determines how frequent context switching can occur

Burstiness


load store
add store
read from file
CPU burst
wait for I/O
I/O burst
store increment
index
write to file
CPU burst
wait for I/O
I/O burst
load store
add store
read from file
CPU burst
wait for I/O
I/O burst

Consequence of burstiness

- Processes can in some cases be treated as a set of bursts (jobs)
- Each job has a certain execution time (burst time)
- CPU-bound vs IO-bound

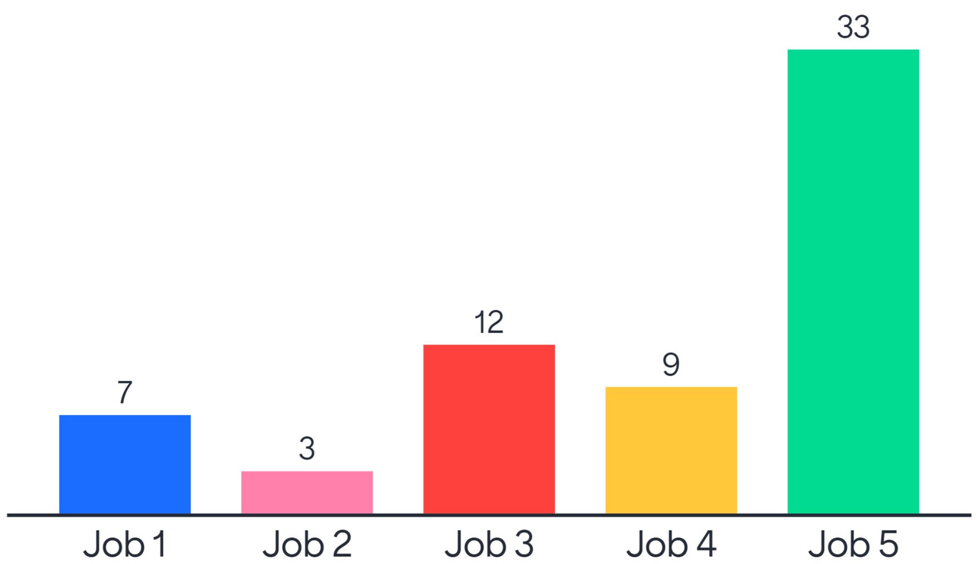
menti.com 3513 0716

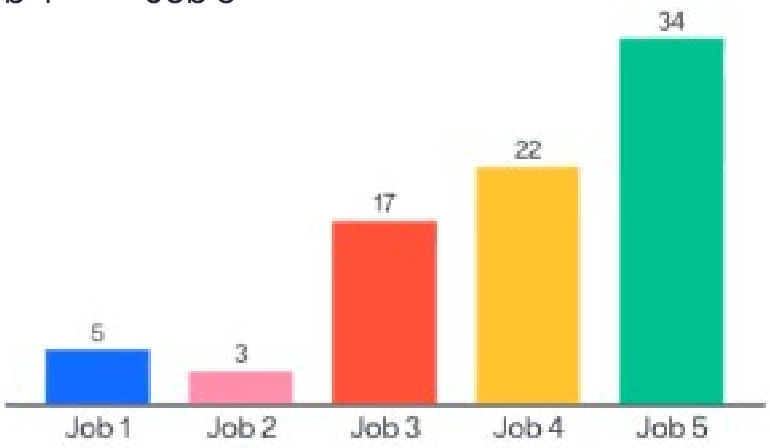
Which job shuld run first?

	Burst time	Interactive	Waited
1	5ms	Y	1ms
2	10ms	N	20ms
3	2ms	N	10ms
4	15ms	Y	15ms
5	10ms	N	40ms

2023 & 2022

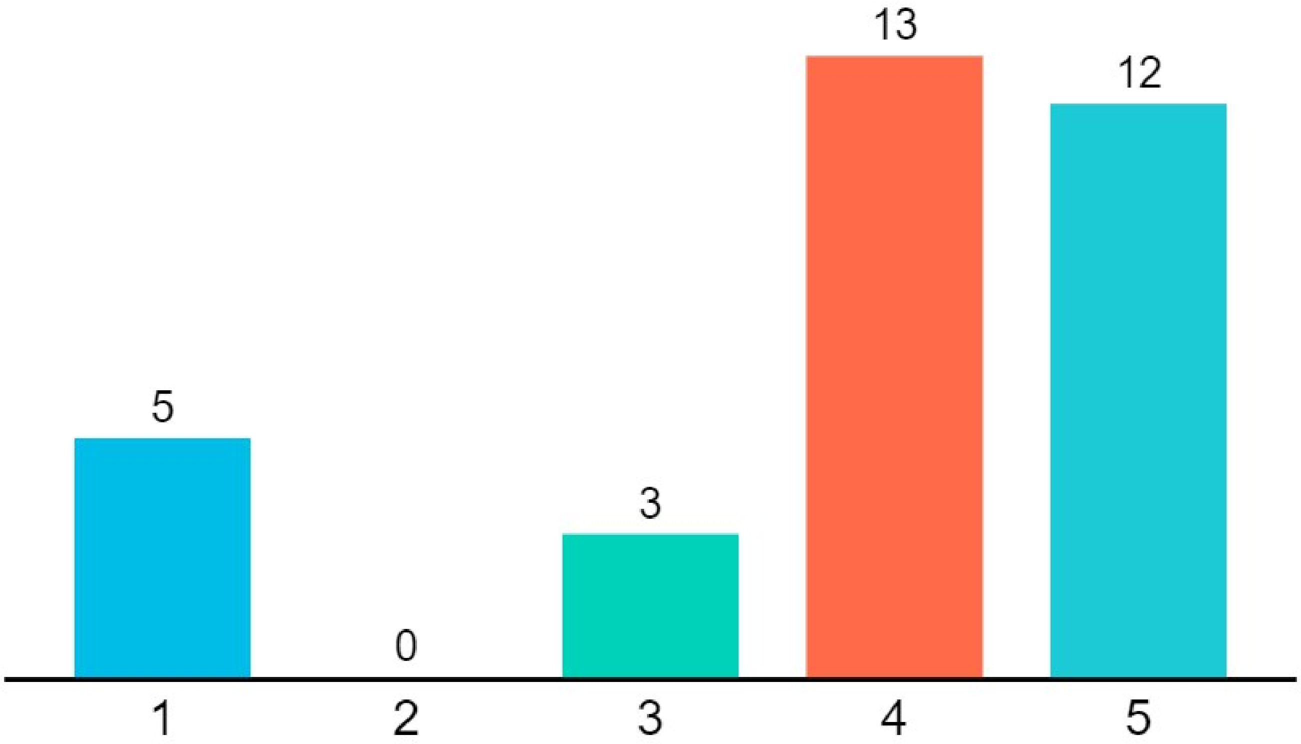
Which job should run?





2024

Which job would run?



What is a good scheduler?

General scheduling Criteria

- **CPU utilization**
  - keep the CPU as busy as possible
- **Throughput**
  - # of processes that complete their execution per time unit
- **Deadlines met?**
  - in real-time systems
- **Energy usage**
  - Mobile and cloud-based computing
  - In particular for multi-core

Time-based Scheduling Criteria

- **Turnaround time**

  time to execute a particular job

- **Waiting time**

  the time a process has been waiting in the ready queue

- **Response time**

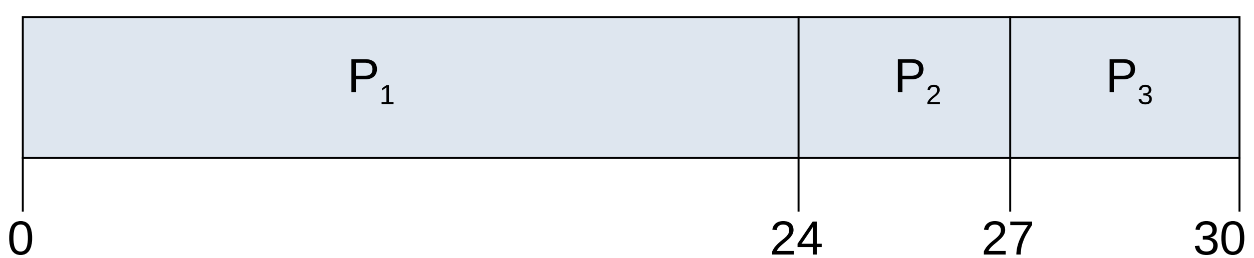
  time it takes from when a request was submitted until the first response is produced (until it start)

First-Come, First-Served (FCFS) Scheduling

<u>Process</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3

The **Gantt Chart** for the schedule is:



FCFS Performance

Waiting time P_i = start time P_i – time of arrival for P_i

FCFS Performance

Waiting time P_i = start time P_i – time of arrival for P_i

- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27) / 3 = 17
- Average turnaround time: (24 + 27 + 30) / 3= 27

FCFS normally used for non-preemptive batch scheduling, e.g. printer queues

(i.e., burst time = job size)

Can we do better?

Yes!

Suppose that the processes arrive in the order P_2 , P_3 , P_1

- The Gantt chart for the schedule is:

- Waiting time for $P_1 = 6$; $P_2 = 0$, $P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$ - much better!
- Average turnaround time: $(3 + 6 + 30) / 3) = 13$

Convoy effect

- Short process behind long process
- IO-bound process delayed by CPU-bound process limits use of IO-devices

- Idea: shortest job first?

Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst.

- Use these lengths to schedule the shortest ready process

- SJF is **optimal**
  - gives minimum average waiting time for a given set of processes

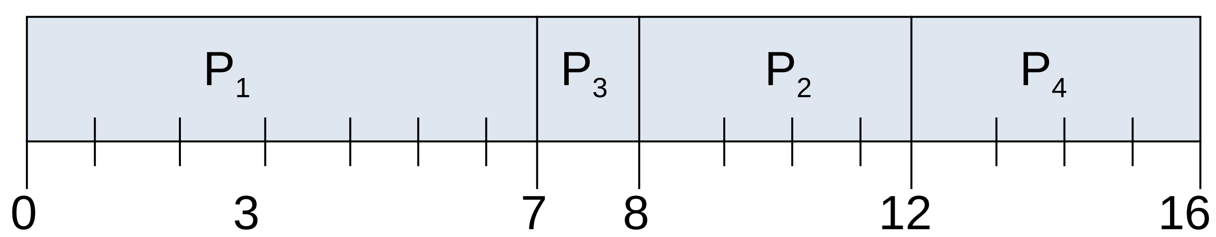
Two variants of SJF

- ***nonpreemptive SJF*** – once CPU given to the process, it cannot be preempted until it completes its CPU burst
- ***preemptive SJF*** – preempt if a new process arrives with CPU burst length less than remaining time of current executing process.
  - Also known as Shortest-Remaining-Time-First (SRTF)

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- with non-preemptive SJF:

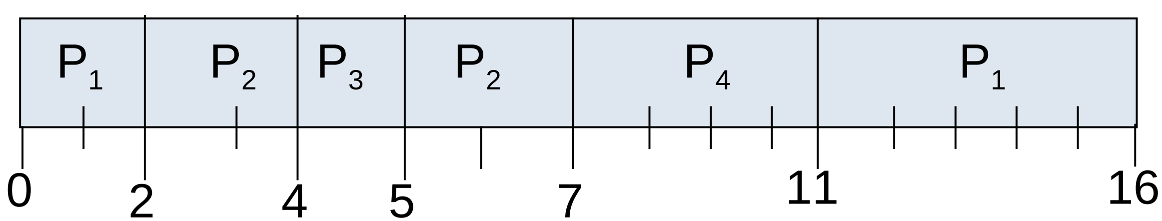


- Average waiting time = (0 + 6 + 3 + 7) / 4 = 4
- Average turnaround time = (7 + 10 + 4 + 11) /4 = 8

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- with preemptive SJF:

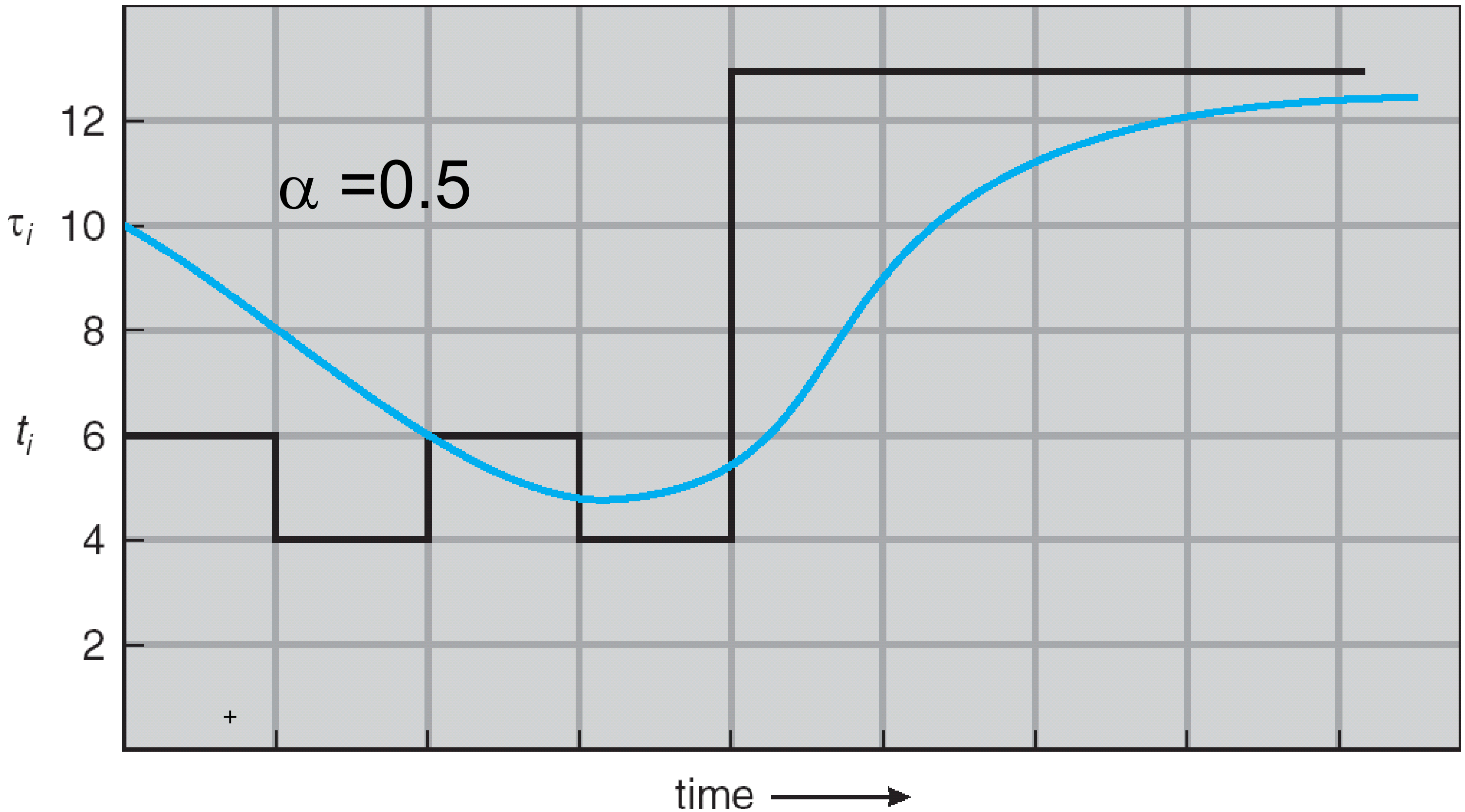


- Average waiting time = (9 + 1 + 0 +2) / 4 = 3
- Average turnaround time = (16 + 5 + 1 + 6) /4 = 7

Predicting Length of Next Burst

- Need to estimate!
- Based on length of previous CPU bursts, using **exponential averaging**:

1. $t_n =$ actual length of n^{th} CPU burst
2. $\tau_{n+1} =$ predicted value for the next CPU burst
3. α, $0 \leq \alpha \leq 1$
4. Define: $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$.



CPU burst (t_i)		6	4	6	4	13	13	13	…
"guess" (τ_i)	10	8	6	6	5	9	11	12	…

Extreme cases

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n.$$

- α =0
  - $\tau_{n+1} = \tau_n$
  - New data does not count

- α =1
  - $\tau_{n+1} = \alpha\, t_n$
  - Only the latest CPU burst counts

Exponential Averaging
All other cases

- Expand the formula:

$$\tau_{n+1} = \alpha\ t_n + (1 - \alpha)\alpha\ t_{n-1} + \ldots$$
$$+(1 - \alpha)^j \alpha\ t_{n-j} + \ldots$$
$$+(1 - \alpha)^{n+1} \tau_0$$

- Since both α and $(1 - \alpha)$ are less than 1, each successive term has less weight than its predecessor

SJF is a special case of **priority scheduling**

Priority Scheduling

- A priority value (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (often smallest integer $\equiv$ highest priority)
  - preemptive
  - nonpreemptive
- Allows giving high priority to important jobs
  - What are important jobs?

Challenge for Priority Scheduling

- **Problems:**
  - Starvation – low-priority processes may never execute
  - Long jobs, even if delayed will monopolize the CPU
- **Solution:**
  - Aging – as time progresses increase the priority of waiting(ready) processes
- How to balance age and priority?

What if we make aging the main scheduling factor?

Round Robin (RR)

- Each process gets a small unit of CPU time:
  - ***time quantum***, usually 10-100 milliseconds.

- After this time has elapsed, the process is preempted and added to the end of the ready queue.

Round Robin performance

- Assume n processes in the ready queue and time quantum q
- Each process gets $1/n$ of the CPU time in chunks of at most q time units at once.
- No process waits more than $(n-1)q$ time units.

Example: RR with Time Quantum q = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3

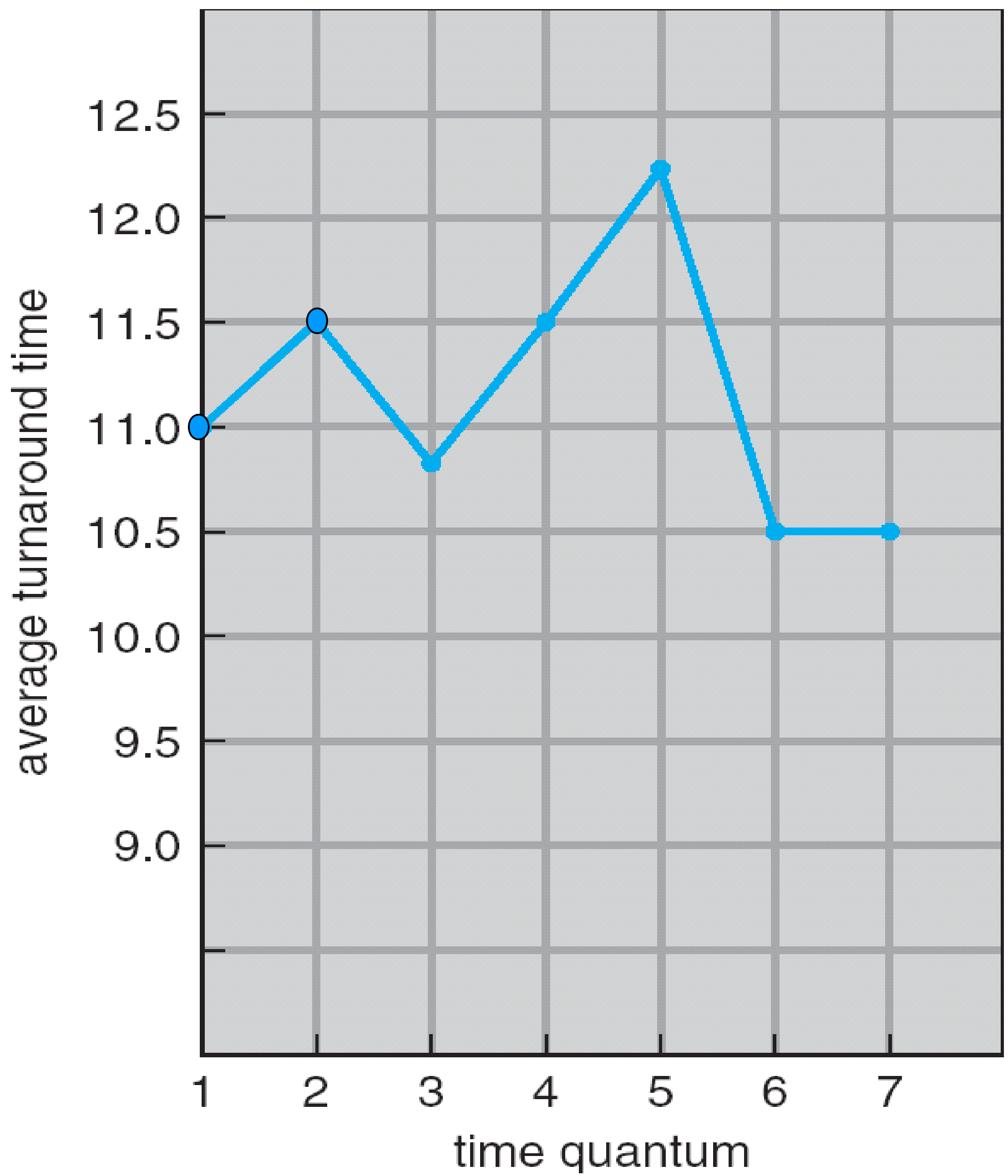
0 20 37 57 77 97 117 121 134 154 162

- Typically, higher average turnaround than SJF, but better *response*

Choice of time quantum (q)

- *q* very large $\Rightarrow$ FCFS
- *q* very small $\Rightarrow$ many context switches
- *q* must be large w.r.t. context switch time, otherwise too high overhead

RR: Turnaround Time Varies With Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Problems with RR and Priority Schedulers

- Priority based scheduling may cause *starvation* for some processes.

- Round robin based schedulers are maybe *too* ”fair”... we sometimes want to prioritize some processes.

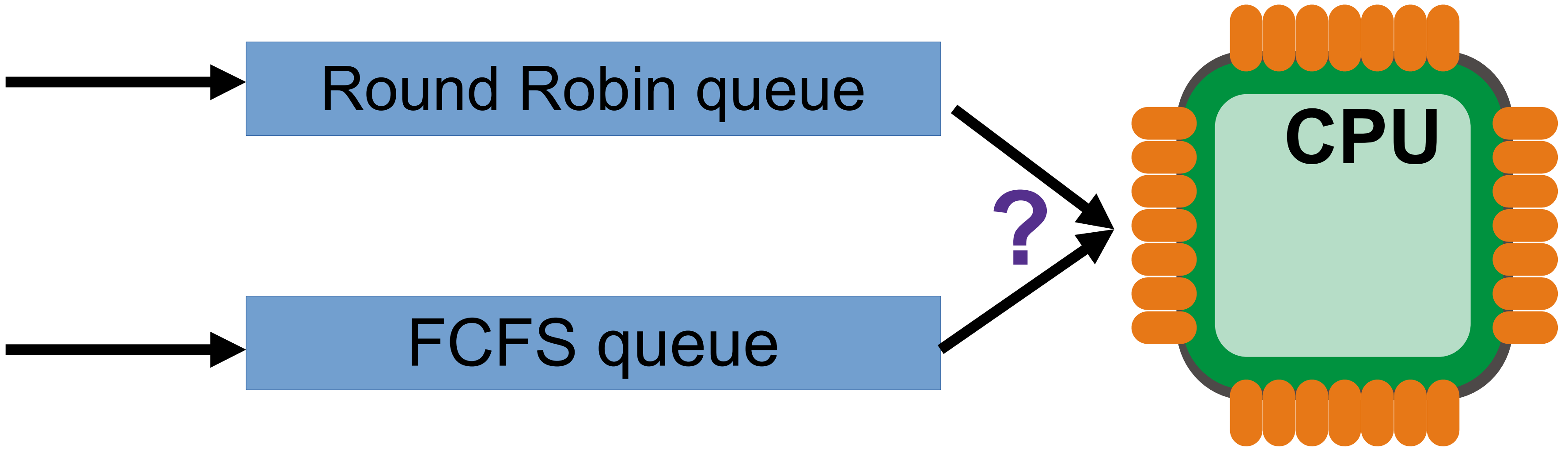
- Solution: Multilevel queue scheduling ...?

Multilevel Queue

- Ready queue is partitioned into separate queues, e.g.:
  - foreground (interactive)
  - background (batch)

- Each queue can have its own scheduling algorithm
  - foreground – RR
  - background – FCFS

Inter-queue scheduling



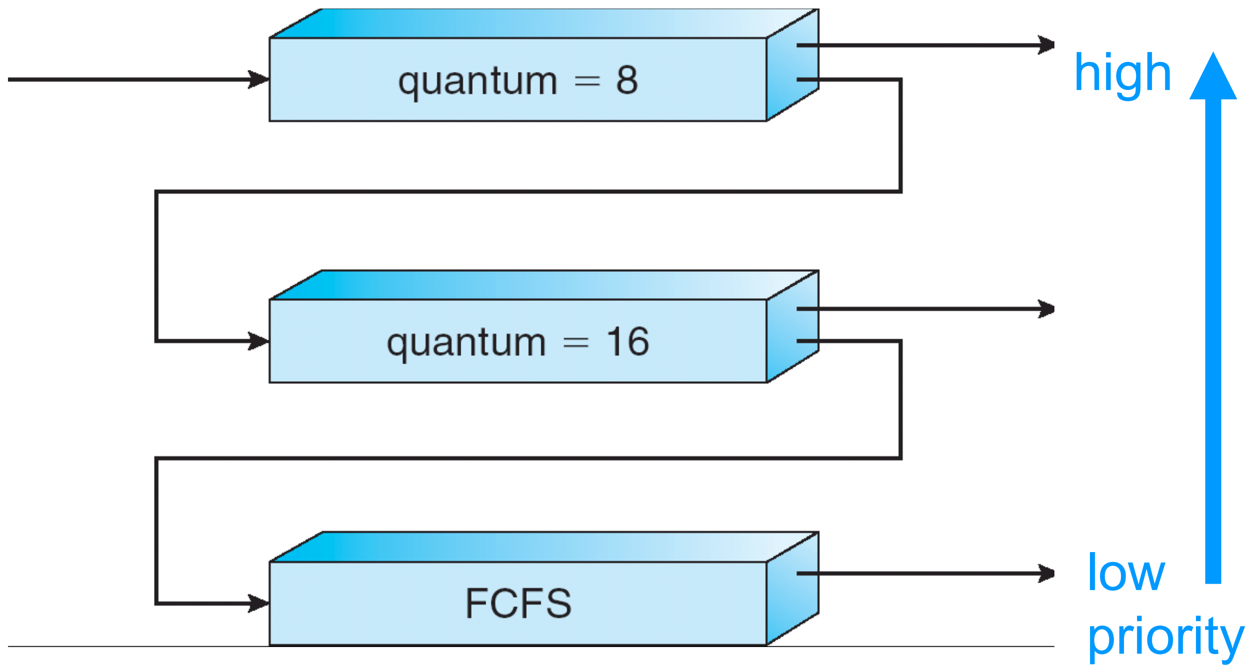
Inter-queue scheduling

- Fixed priority scheduling
  - Serve all from foreground queue, then from background queue.
  - Possibility of starvation.

- Time slice
  - Each queue gets a certain share of CPU time
    which it can schedule amongst its processes
  - Example: 80% to foreground in RR, 20% to background in FCFS

Multilevel Feedback Queue (MFQ)

- Example with three queues:
  - Q_0 – RR with q = 8 ms
  - Q_1 – RR with q = 16 ms
  - Q_2 – FCFS

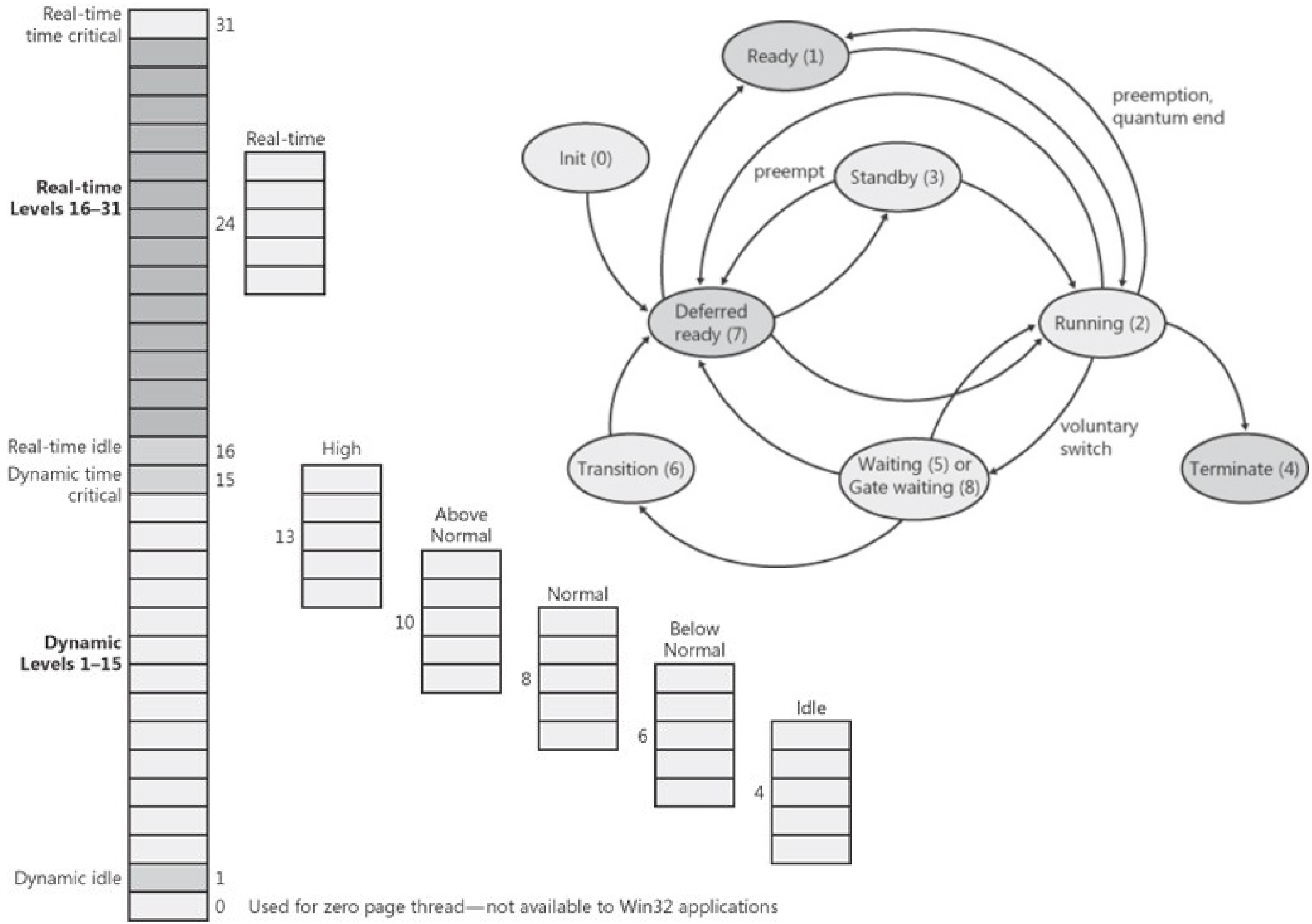


Multilevel Feedback Queue

- A process can move between the various queues
  - aging by moving to higher prio queue
  - Higher prio to IO-bound
  - Lower prio to CPU-bound

- Time-sharing among the queues in priority order
  - Processes in lower queues get CPU only if higher queues are empty

Windows Scheduling



https://www.microsoftpressstore.com/articles/article.aspx?p=2233328

A more general concept

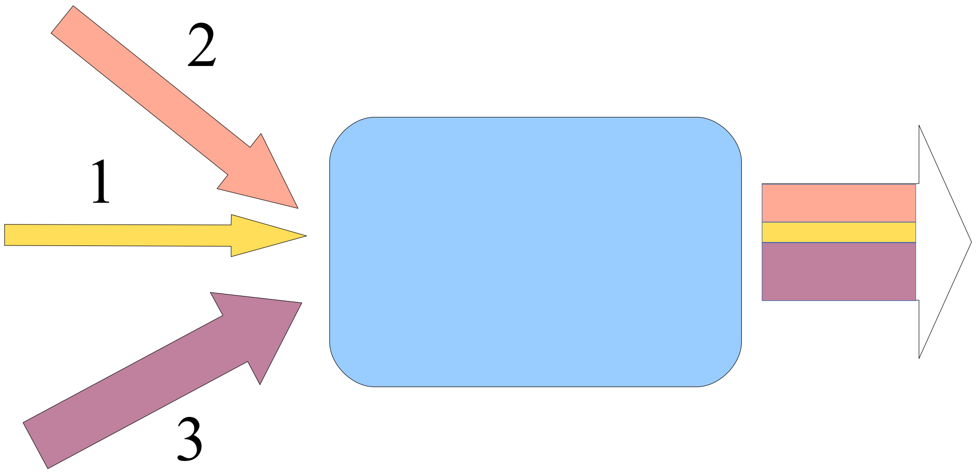
Proportional fairness

- Assume *n* long-running processes
- Give each process *i* a weight w_i
- During some time interval *T*, each process *i* is given the following access time

  $w_i * T / (w_1 + w_2 + \ldots + w_n)$

Generalized Processor Sharing

- Work conserving (CPU not idle when there is work to do)
- Guarantees *proportional fairness* (i.e., no starvation)
- Works as follows:
  - Assign a logical queue for each process / process group
  - Serve an infinitesimal amount from each queue



Implementing GPS

- Perfect implementation impossible due to
  - Non-preemption
  - Non-zero time quanta
  - Not knowing when the next (high-priority) job arrives

Problematic case, q = 5

Job	Arrival time	Length	Prio
1	0	4	1 (low)
2	0	9	2 (medium)
3	5	9	3 (high)

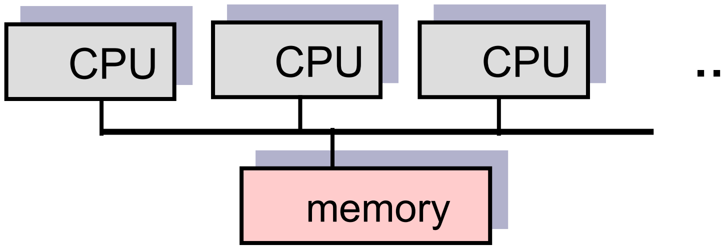
Expected finish time at t=0: Job 1 at t=12, Job 2 at t=13
Expected finish time at t=4: Job 2 at t=13
Expected finish time at t=9: Job 2 at t=19, Job 3 at t=24

Approximations

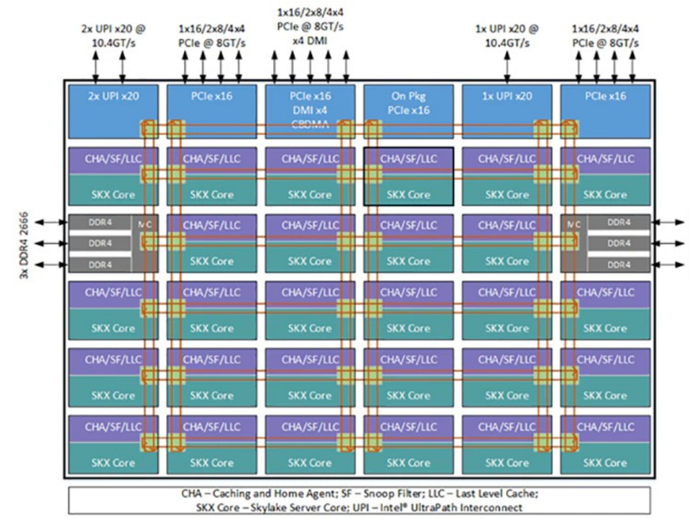
- Networking:
  - Weighted Fair Queuing (WFQ)
- CPU Scheduling in Linux
  - Completely Fair Scheduler
- Basic idea: Schedule packets/jobs as if GPS was running (don’t care about the future)

Multiprocessor Scheduling

Multiprocessor variants



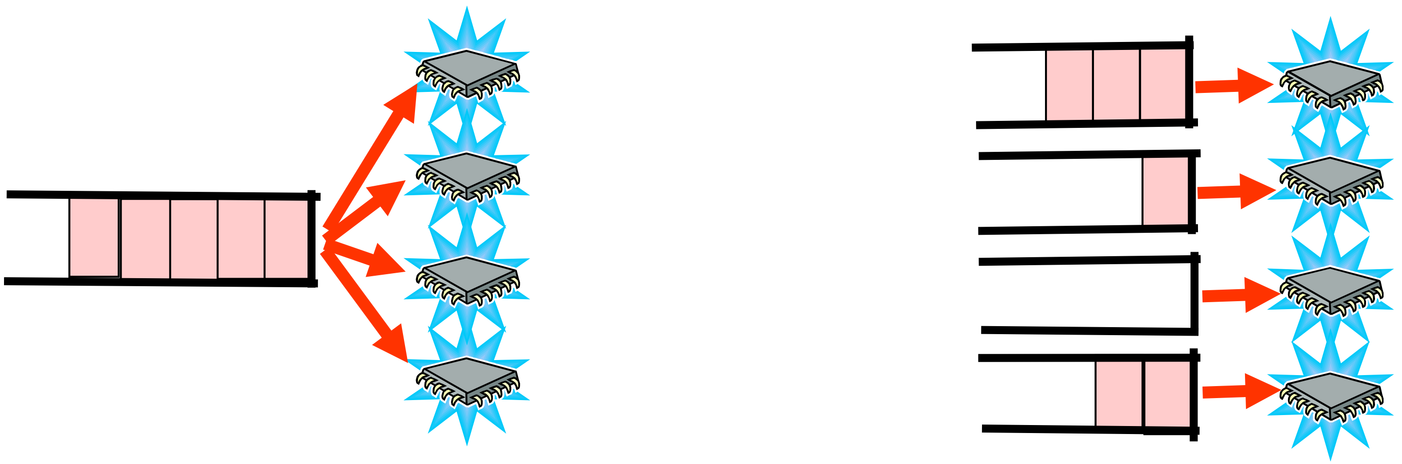
- Multiprocessor (SMP)
  - homogeneous processors, shared memory



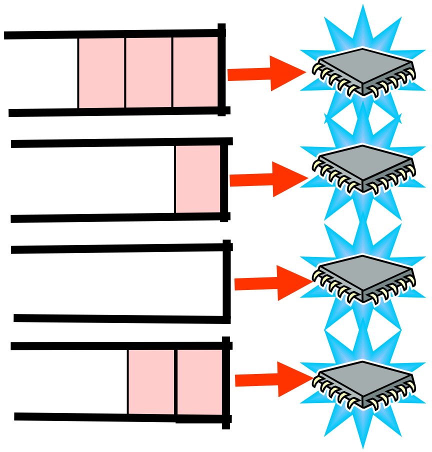
- (homogeneous) Multi-core processors
  - cores share L2 cache and memory

- Simultaneous multithreading
  - Take advantage of instruction-level parallelism

Common vs local queue



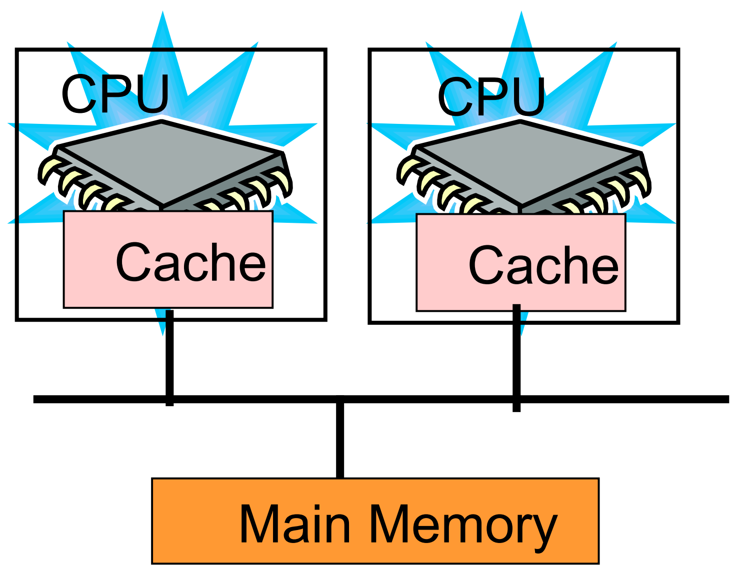
Processor-local ready queues



- Load balancing by **task migration**
- Push migration vs. pull migration (*work stealing*)
  - Linux: Push-load-balancing every 200 ms, pull-load-balancing whenever local task queue is empty

Affinity-based Scheduling

- Migration should be avoided due to the cache



Why does the cache influence scheduling?

- Cache contains copies of data recently accessed by CPU
- If a process is rescheduled to a different CPU (+cache):
  - Old cache contents invalidated by new accesses
  - Many cache misses when restarting on new CPU

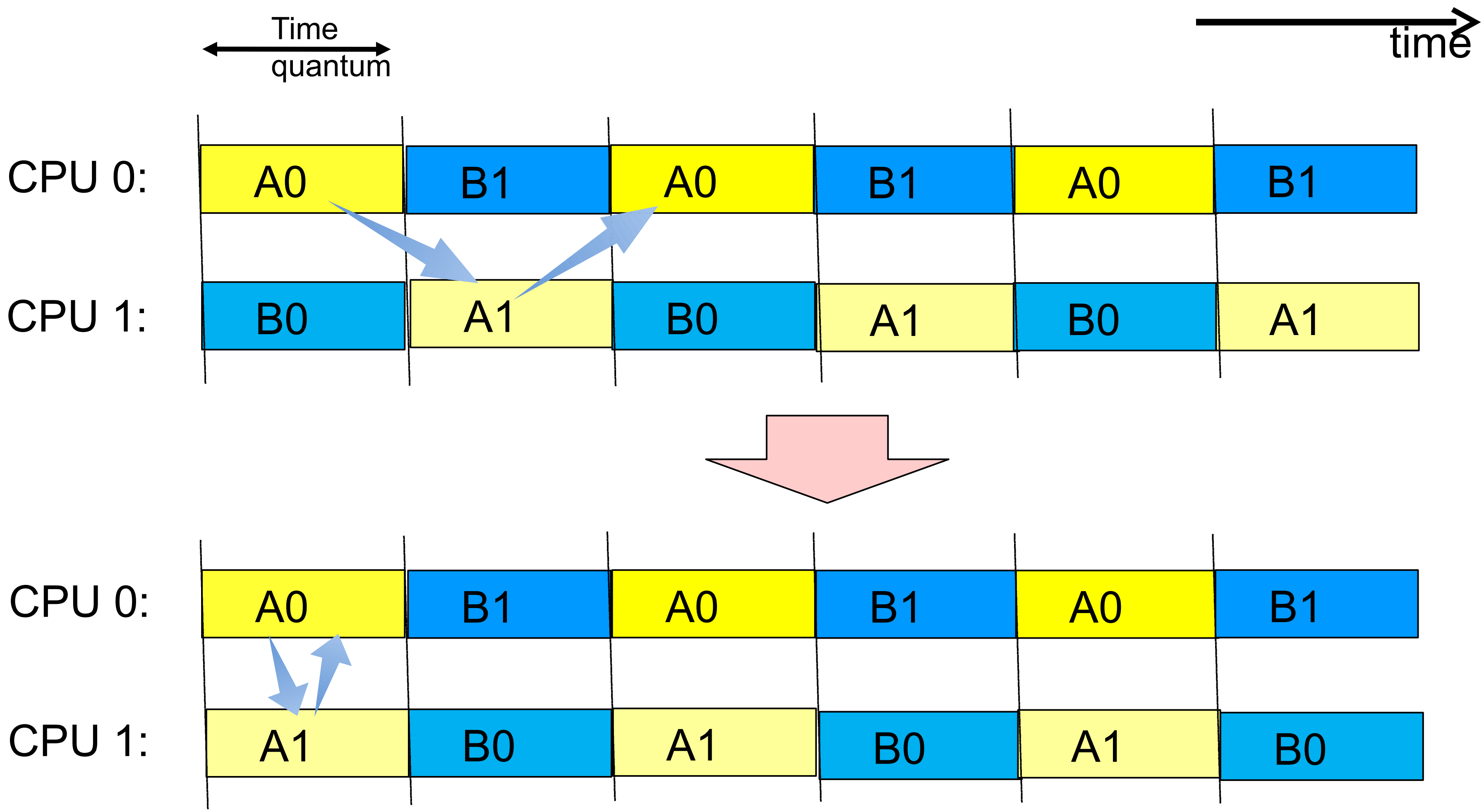
→ much bus traffic and many slow main memory accesses

Affinity-based scheduling

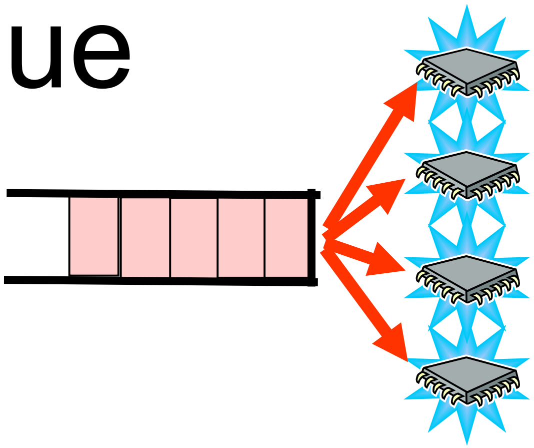
- Policy: Try to <u>avoid migration</u> to other CPU if possible.
- A process has **affinity** for the processor on which it is currently running
  - **Hard affinity** (e.g. Linux):
    Migration to other CPU is forbidden
  - **Soft affinity** (e.g. Solaris):
    Migration is possible but undesirable

Scheduling Communicating Threads

- Frequently communicating threads / processes (e.g., in a *parallel* program) should be scheduled simultaneously on different processors to avoid idle times



Common ready queue



Supported by Linux, Solaris, Windows XP, Mac OS X

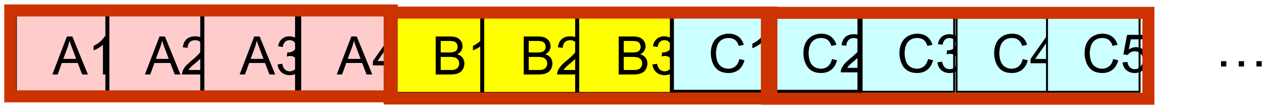
- **Variants**
  - **Job-blind scheduling** (FCFS, SJF, RR – as above)
    - schedule and dispatch one by one as any CPU gets available
  - **Affinity based scheduling**
    - guided by data locality (cache contents, loaded pages)
  - **Co-Scheduling / Gang scheduling** for *parallel* jobs

Energy-aware scheduling

- Power consumption grows quadratically with CPU “speed” $\alpha \cdot C \cdot V^2 \cdot f$
  - Reduce frequency and *voltage* in cases of low load
- Turning off a core or a CPU allows even more power saving
  - Requires that the remaining cores run at a higher speed
- Rotating which core to run on to reduce cooling problems

Co-Scheduling / Gang Scheduling

- Tasks can be parallel (have >1 process/thread)
- Global, shared RR ready queue
- Execute processes/threads from the same job simultaneously rather than maximizing processor affinity
- Example: **Undivided Co-scheduling algorithm**
  - Place threads from same task in adjacent entries in the global queue



  - Window on ready queue of size #processors
  - All threads within same window execute in parallel for at most 1 time quantum

☺ RR → fair (no indefinite postponement)

☺ Programs designed to run in parallel profit from multiprocessor env.

☹ May reduce processor affinity

Summary: CPU Scheduling

- **Goals:**
  - Enable multiprogramming
  - CPU utilization, throughput, ...
- **Scheduling Algorithms**
  - Preemptive vs Non-preemptive scheduling
  - RR, FCFS, SJF
  - Priority scheduling
  - Multilevel queue and Multilevel feedback queue
- **Multiprocessor Scheduling**

Next time

- Synchronization Ch. 6.1-6.7 + 7.1-7.3