SUPSI

Scheduling

Operating Systems

Amos Brocco, Lecturer & Researcher

22 settembre 2020

Objectives

- Understand different scheduling policies
- Understand how schedulers in current operating systems work

Browsing
- Get a rapid overview.

Reading
- Read it and try to understand the concepts.

Studying
- Read in depth, understand the concepts as well as the principles behind the concepts.

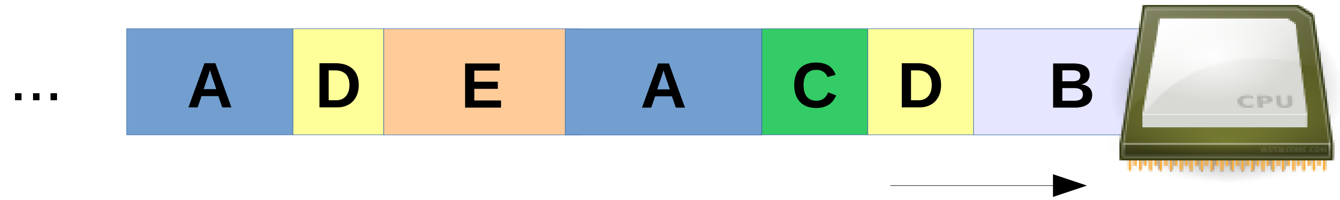
You are also encouraged to try out (compile and run) code examples!

Question

What is a scheduler?

Scheduling

- In a multiprogrammed system the **scheduler** determines the order in which processes/threads can obtain system resources (typically the CPU) depending on a **scheduling policy/algorithm**



Answer

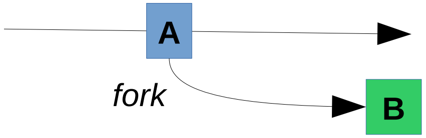
Ensuring that the scheduling policy is respected

- Consider a computer with just one CPU: if the operating system starts executing a process then it looses control... to achieve multitasking
  - ... either each process *cooperates* with others and voluntarily gives back the CPU (after some time) (**cooperative multitasking**)
    - ... what if the executing process never gives up control back to the OS?
  - ... or the operating system must employ *some technique* to take back the CPU (after some time) from the executing process (**preemptive multitasking**)

Explaination

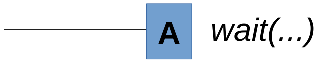
Other situations where the scheduler is called

- When a new process/thread is created



- When a process/thread terminates

- When a process/thread is waiting for a resource

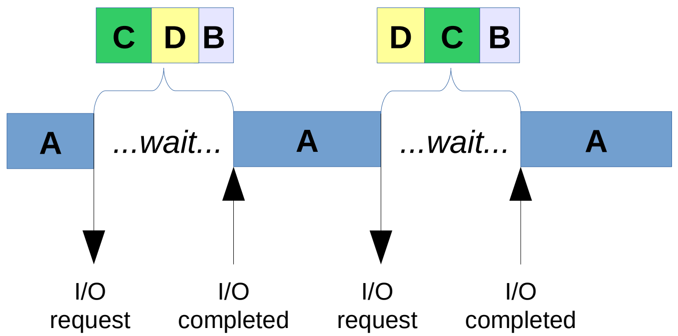


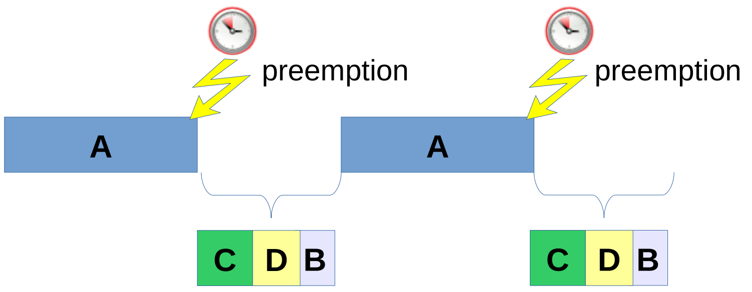
- When a resource becomes available

Explaination

I/O Bound – CPU Bound

- Process behavior influences scheduling:
  - **I/O Bound processes**
    - Frequent I/O waits (*I/O burst*) → frequent yield
  - **CPU bound**
    - Long computations (*CPU burst*) → require preemption





Explaination

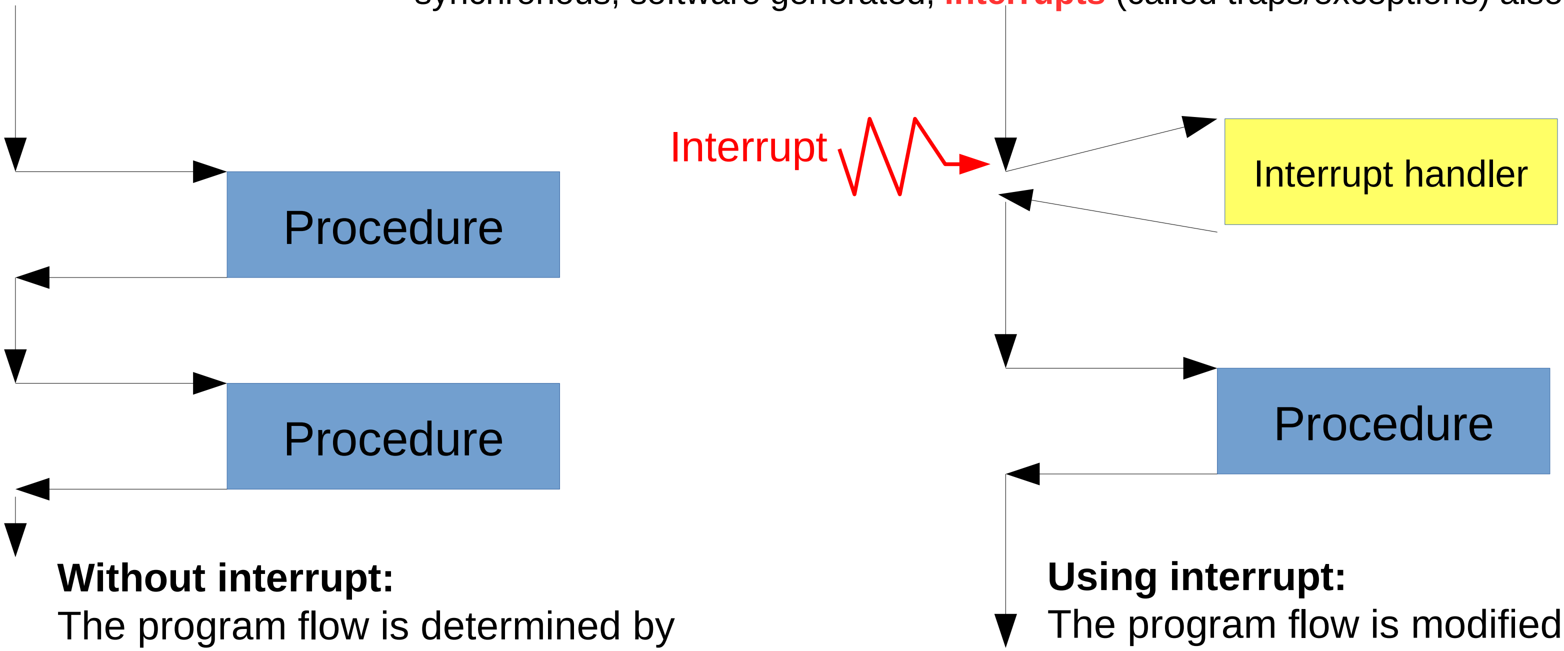
Question

How to pre-empt a process?

Hardware can lend a hand

- The hardware implements a signaling mechanism that enables devices to **interrupt** the CPU (and thus the program flow) in an **asynchronous** * way.

* synchronous, software generated, **interrupts** (called traps/exceptions) also exist



Without interrupt:
The program flow is determined by the sequence of instructions

Using interrupt:
The program flow is modified by an external event (the interrupt)

Answer

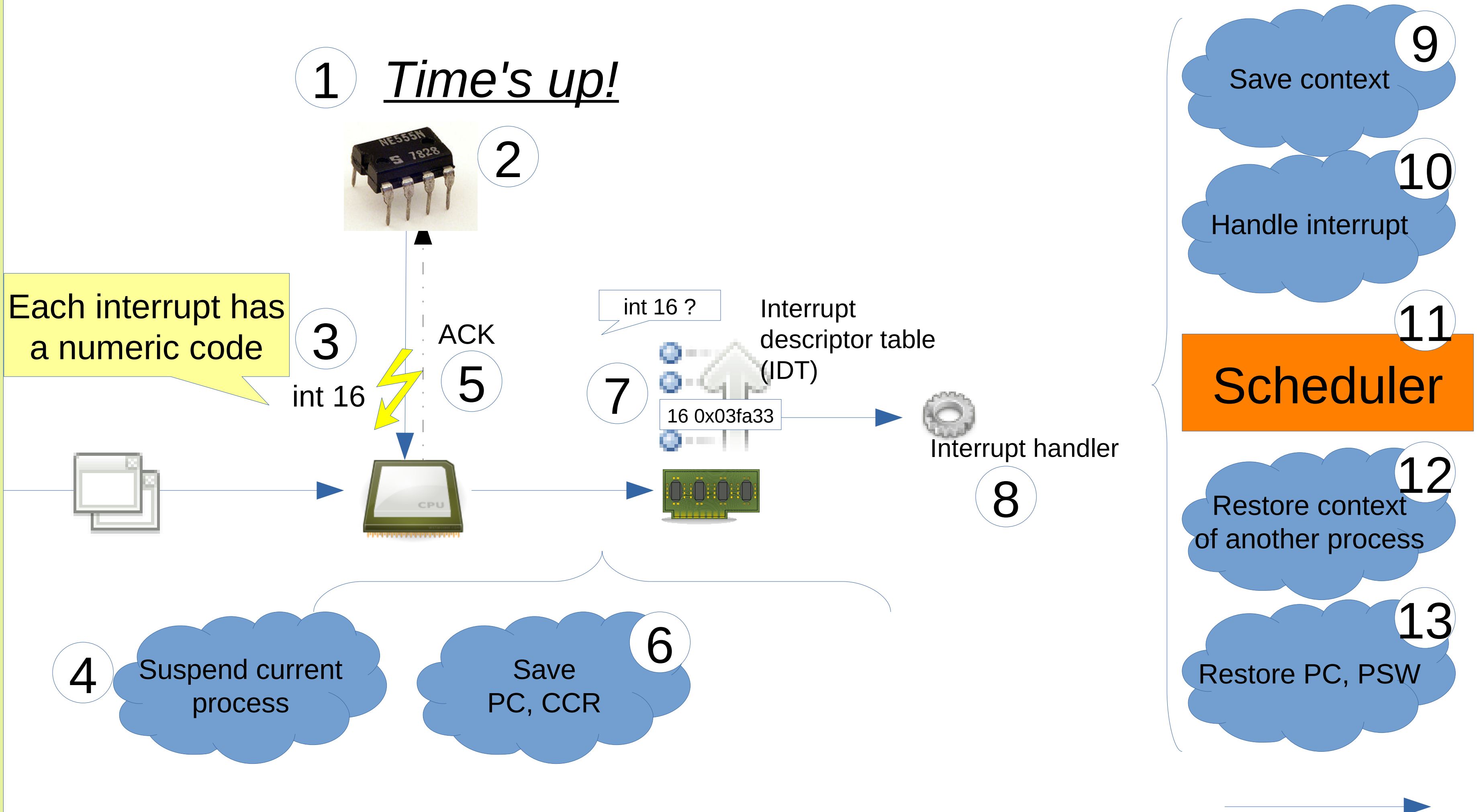
Interrupt handlers

- Each interrupt is associated with a specific routine implemented by the operating system called **interrupt handler** (or **Interrupt Service Routine**)

- Because many different interrupts exist this association is done using an in-memory array called **interrupt vector table** (or **interrupt descriptor table**)
  - The CPU knows the address of this table (either because it is found at a fixed memory address, or through a base register)
  - The contents of this table (address of the handlers) are set up by the operating system
  - Interrupt handlers run within the kernel

Explaination

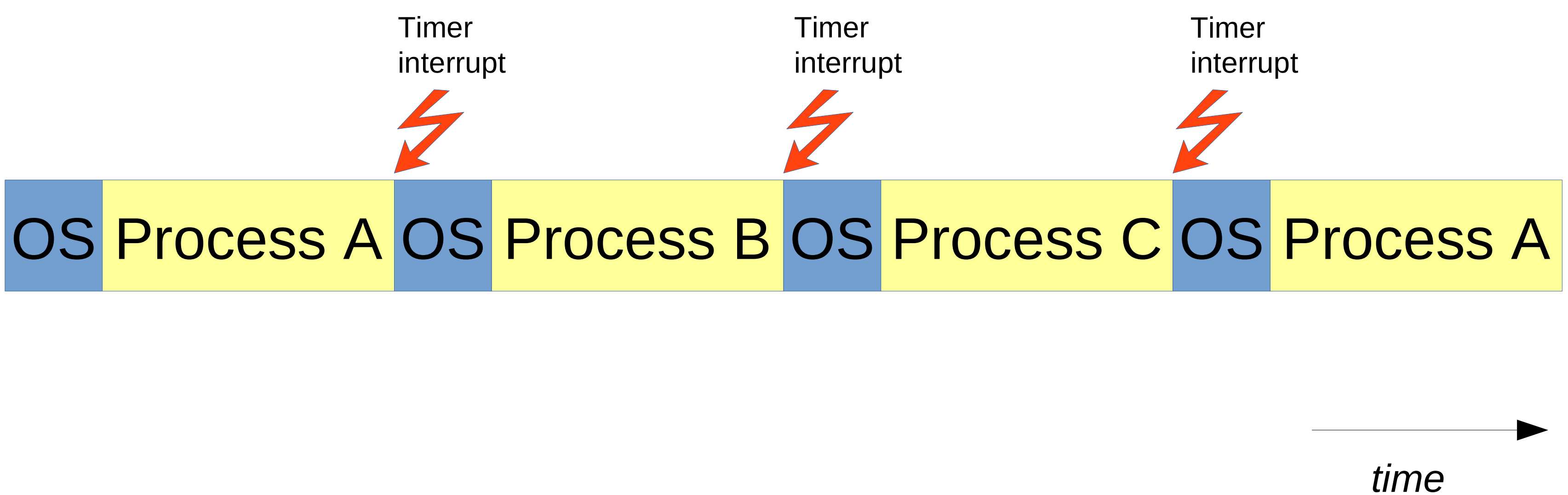
Preemption using a timer



Explaination

Ensuring that the scheduling policy is respected

- Using an interrupt the operating system gets the CPU back at predefined intervals



Explaination

How to keep track of the process's state?

Scheduling ≈ changing process' state

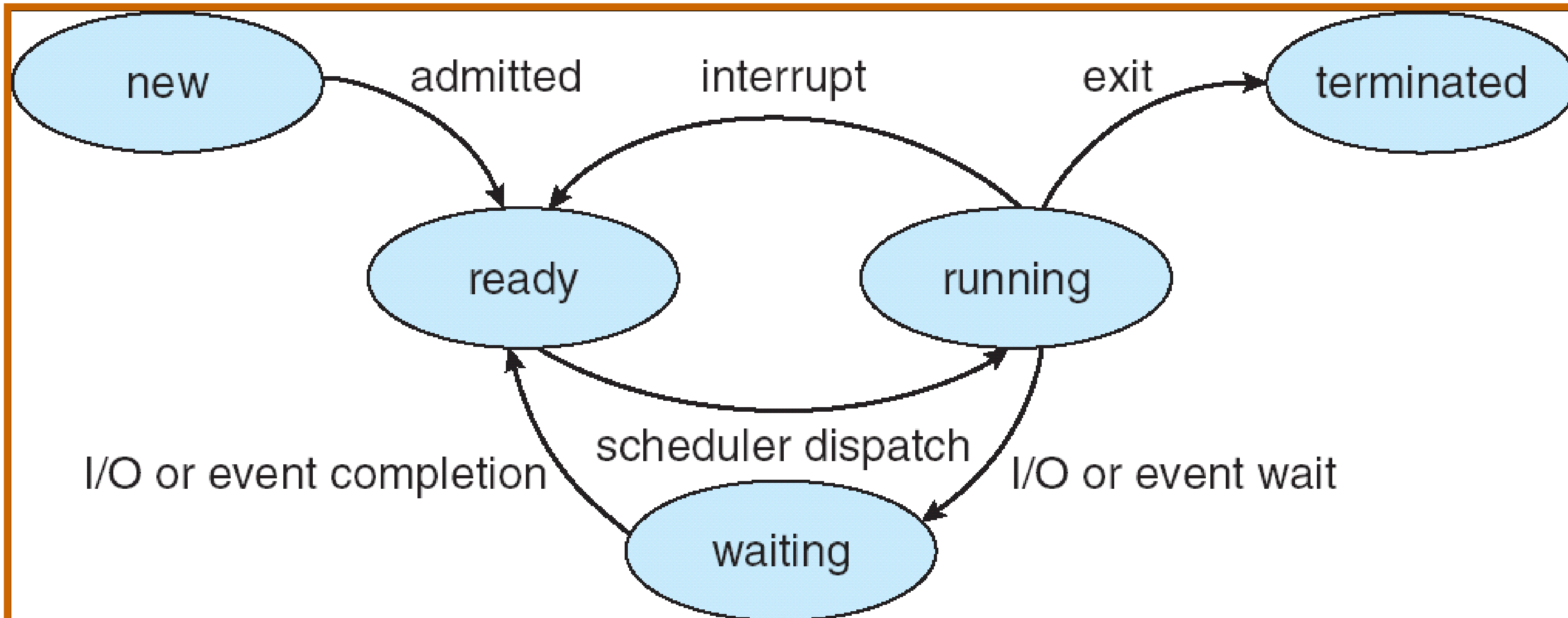


Image from Silberschatz et al. “Operating System Concepts – 7th Edition”

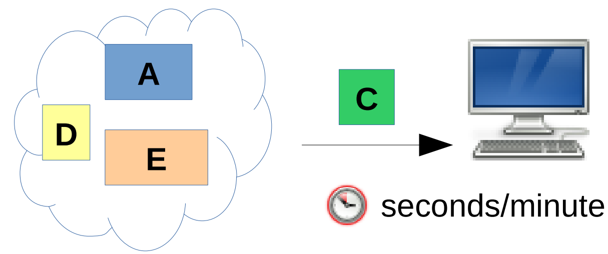
Answer

What kind of scheduling problems do we have?

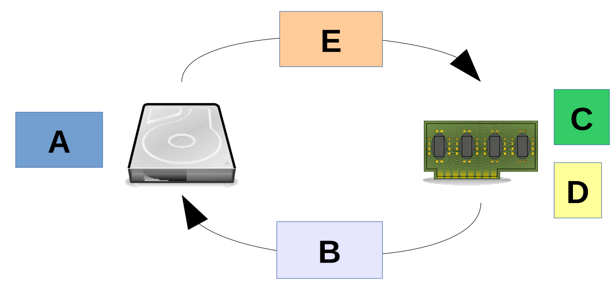
Question

Types of scheduling

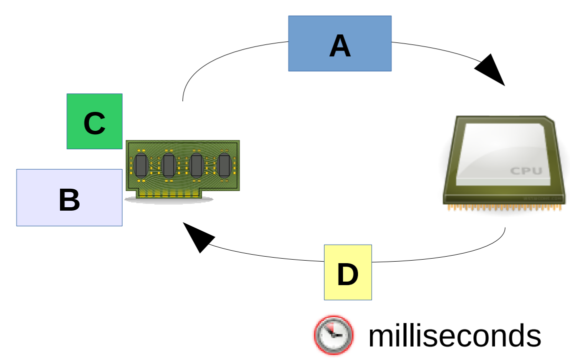
- **Long term scheduling**
  - admission scheduler (batch systems)
  - determines when to admit a process (depending on available resources)



- **Medium term scheduling**
  - emergency/memory scheduler
  - determines which process can be temporarily suspended (*swapped from memory to disk*) or restored (*swapped from disk to memory*)



- **Short term scheduling**
  - CPU scheduler (time sharing systems)
  - determines (very quickly) the next process to be given some CPU time



Answer

Different computing environments

- **Batch**
  - long computation
  - no interaction with the user
- **Interactive** (time sharing systems)
  - multiple user (local or remote) at the same time
- **Real time**

Answer

Question

How can we evaluate a scheduler?

Scheduling evaluation criteria

- **Throughput** (batch systems)
  - Number of jobs that the system can complete per time unit
- **Turnaround time** (batch systems)
  - Average time required to complete a job
    = *completion time – submission time*
- **Response time** (interactive systems)
  - Time between issuing a command and the result (for example, between a mouse click and the opening of a window)
- **Predictability and regularity** (realtime systems)
  - Response time must be guaranteed

Answer

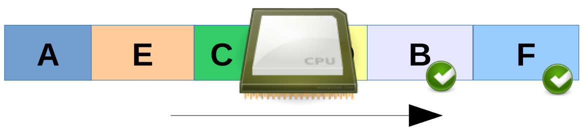
Scheduling: general objectives

- Different environment → different objective
- ...but there are general objectives:
  - **Fairness**
    - Each process should get a fair share of the CPU
  - **Policy enforcement**
    - Scheduling decisions must be enforced
  - **Balance**
    - Maximize resource usage

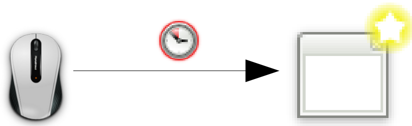
Answer

Scheduling: specific objectives

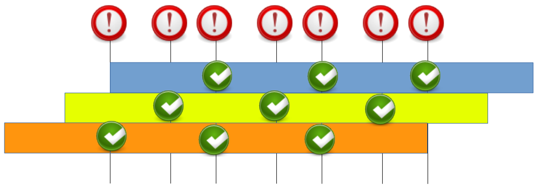
- **Batch systems**
  – Maximize throughput
  – Minimize turnaround time



- **Interactive systems**
  – Minimize response time



- **Realtime systems**
  – Fullfill all deadlines



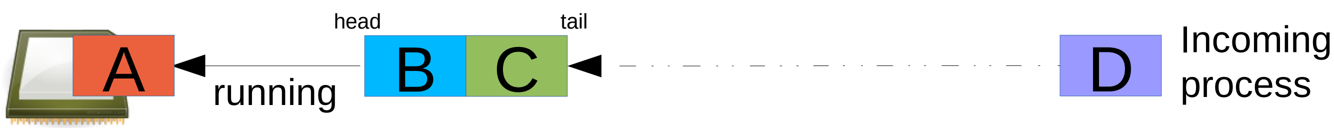
Explaination

What kind of scheduling algorithm do exist?

Question

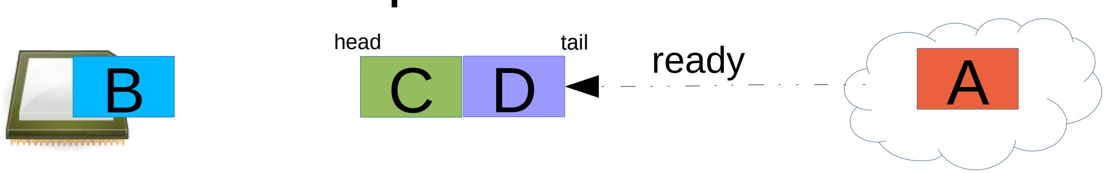
Batch scheduling: First-come First-served (FCFS)

- Processes are executed following their arrival order:
  - Each process runs as long as it wants
  - New jobs are put at the end of a queue



  - If a process blocks the next one is executed

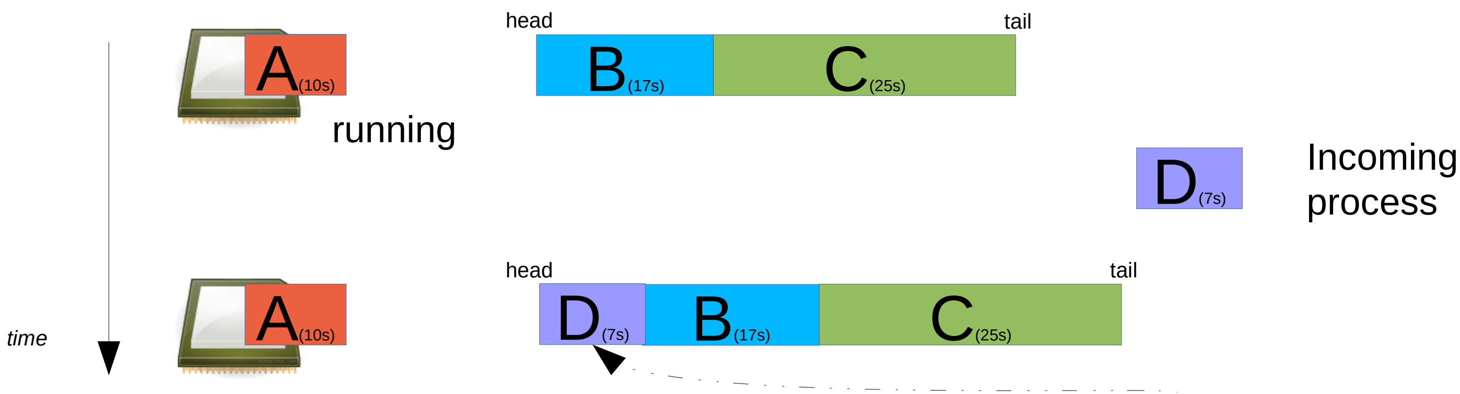
    - when a blocked process becomes ready it is put at the end of the queue



Explaination

Batch scheduling: Shortest Job First (SJF)

- The scheduler gives priority to the process with the shortest estimated run time

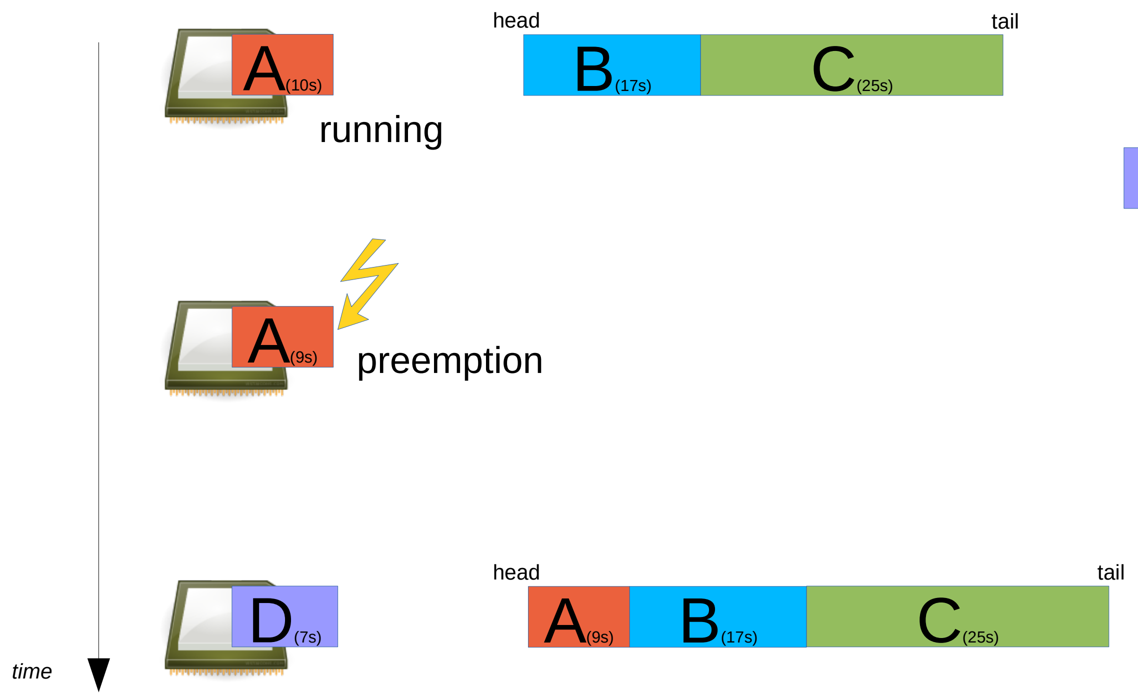


  - **Maximizes throughput** (many short tasks are quickly and frequently completed)
  - **Minimizes turnaround** time for short processes
  - **Requires an estimated run time**
  - Can lead to **starvation** of long processes (indefinitely postponed)

Explaination

Batch scheduling: Shortest Remaining Time Next (SRTN)

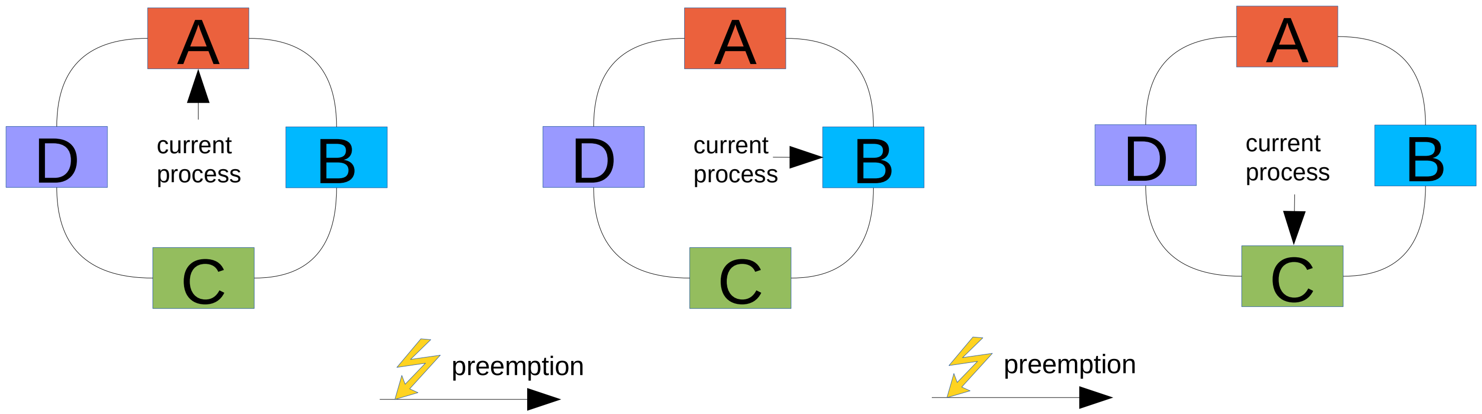
- Preemptive version of SJF, considers the remaining run time of the process



Explaination

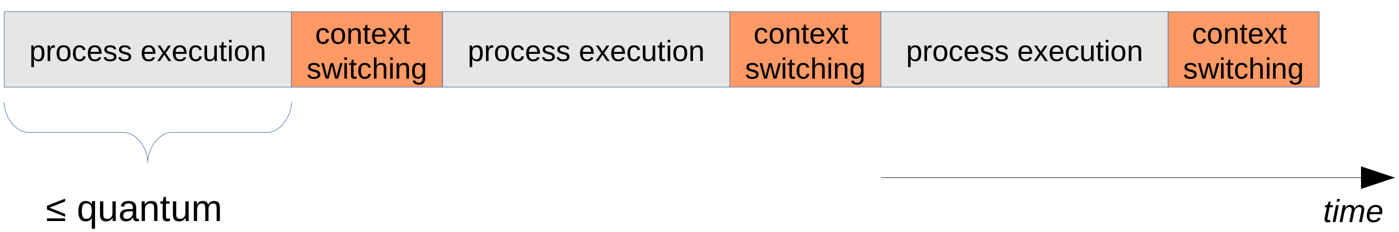
Interactive system scheduling: Round Robin

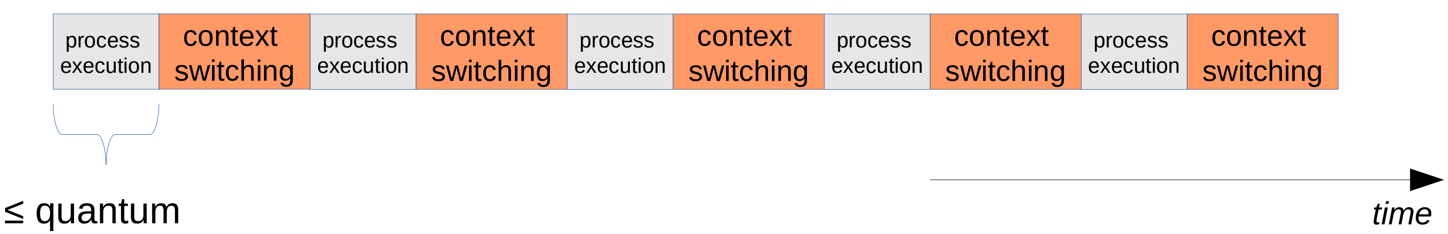
- Processes are organized in a (circular) list
- The scheduler gives to each process a time slot (**quantum**):
  - When the quantum expires the CPU is preempted and given to another (ready) process
  - If the process yields the CPU is given to another process



Interactive system scheduling: Round Robin

- Context switching takes some time
  - the length of a quantum must be carefully determined to avoid spending too much time “context switching”





Explaination

Interactive system scheduling: Priority scheduling

- Issue: Round-robin assigns the same quantum to each process
  - does not account for differences between processes (some processes might be more important than others)

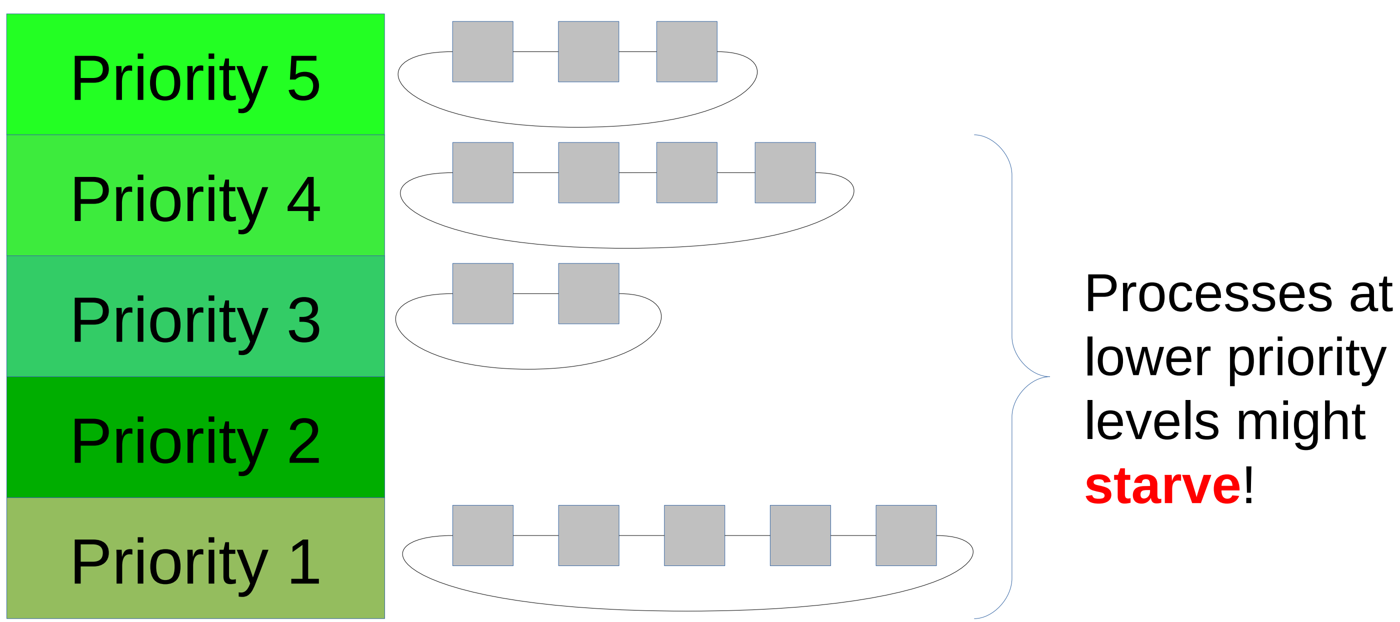
- Solution: introduce **priorities**
  - The scheduler chooses the highest priority ready process
    - high priority processes have more chances to run → should get more CPU time
    - high priority processes can preempt lower priority ones (if a higher-priority process becomes ready during the quantum, the lower-priority process is interrupted and the higher-priority thread is run)
  - To avoid starvation of low priority processes the priority might need to be adjusted dynamically

...quite difficult to get it right!

Explaination

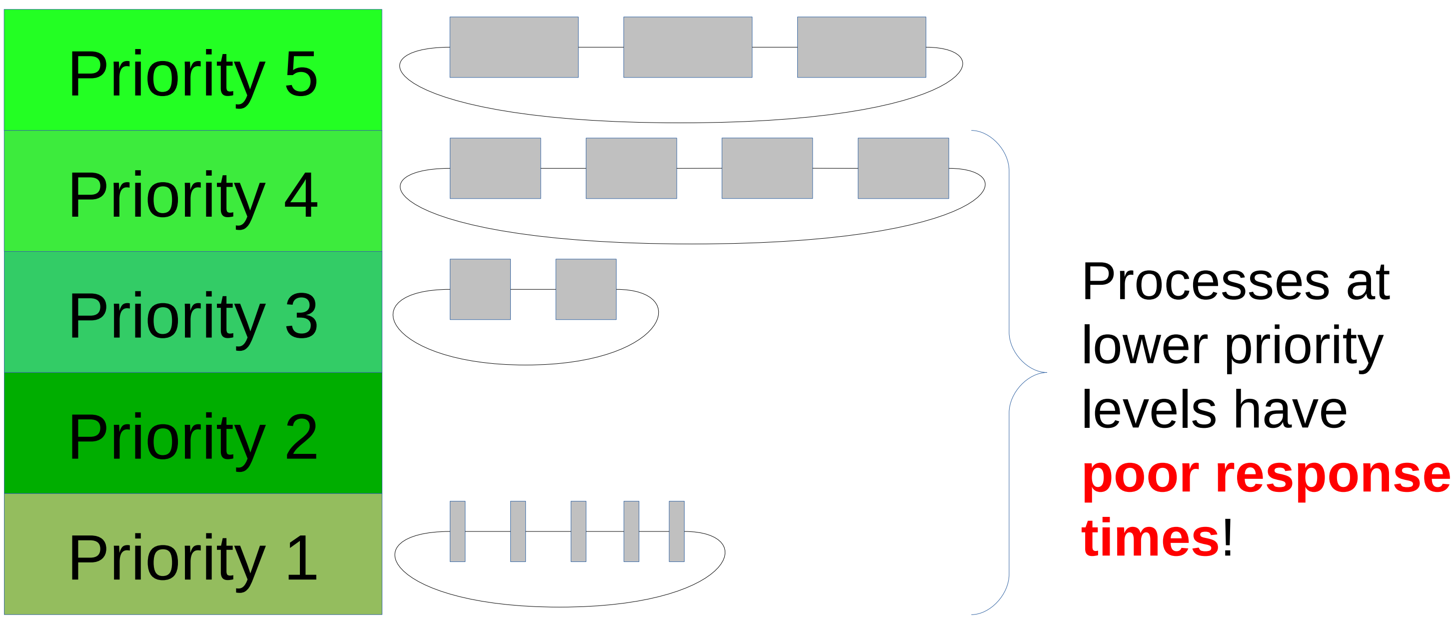
Priority scheduling *...quite difficult to get it right! (Take one)*

- Each priority level is linked to a round-robin queue:
  - As long as there are ready processes in the highest priority queue execute them, otherwise switch to a lower priority class queue



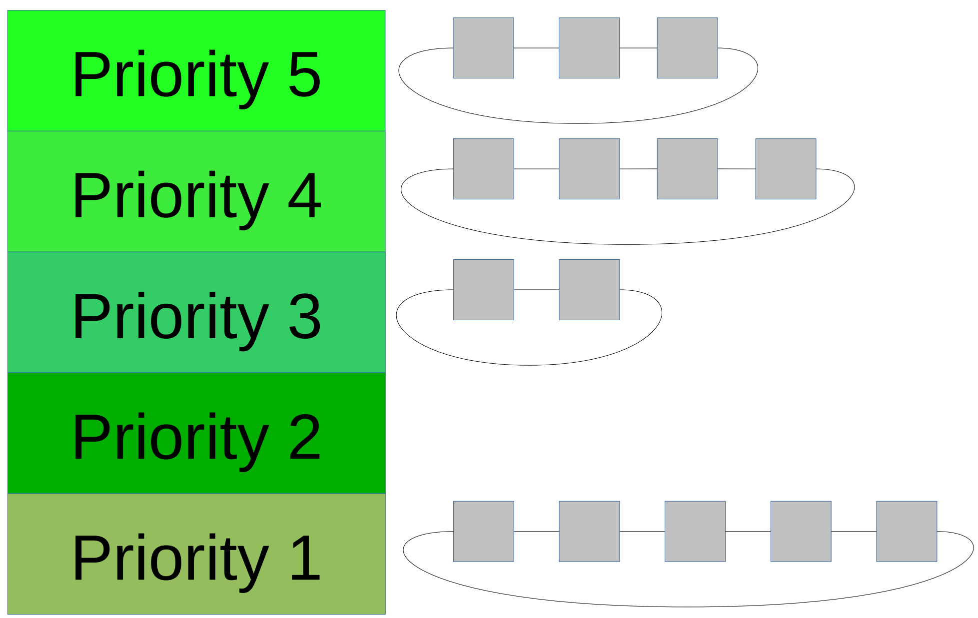
Priority scheduling ...*quite difficult to get it right! (Take two)*

- Each priority level is linked to a round-robin queue:
  - Processes with higher priority receive a longer quantum



Priority scheduling: Multilevel Feedback Queue (MLFQ) *

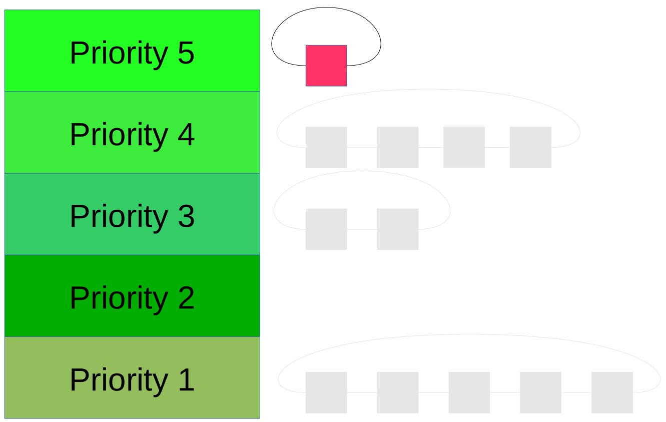
- Each priority level is linked to a round-robin queue:
  - As long as there are ready processes in the highest priority queue execute them, otherwise switch to a lower priority class queue
- **Depending on their behavior, move processes between queues**



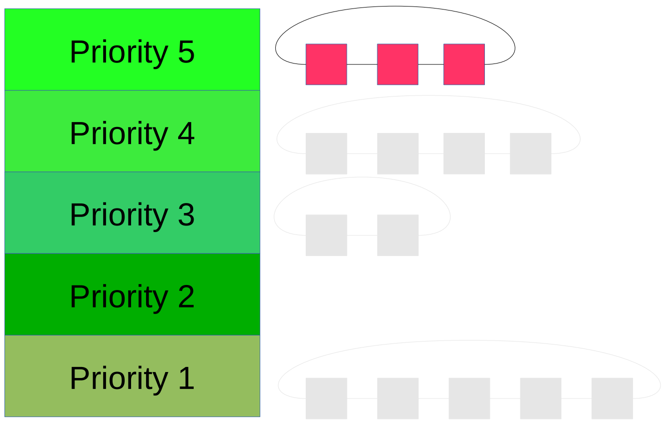
* As presented in Arpaci-Dusseau, Remzi H.; Arpaci-Dusseau, Andrea C. (2014). Operating Systems: Three Easy Pieces [Chapter Multi-level Feedback Queue].

Explaination

Priority scheduling: Multilevel Feedback Queue (MLFQ)

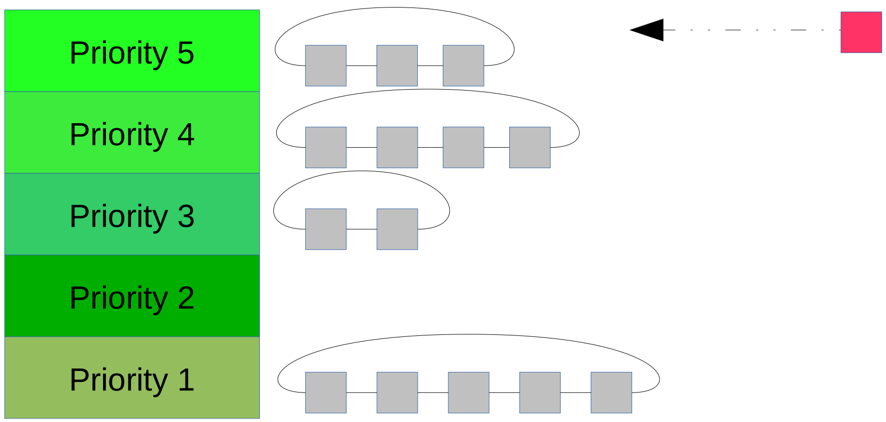


The scheduler always chooses to executed the highest priority ready process

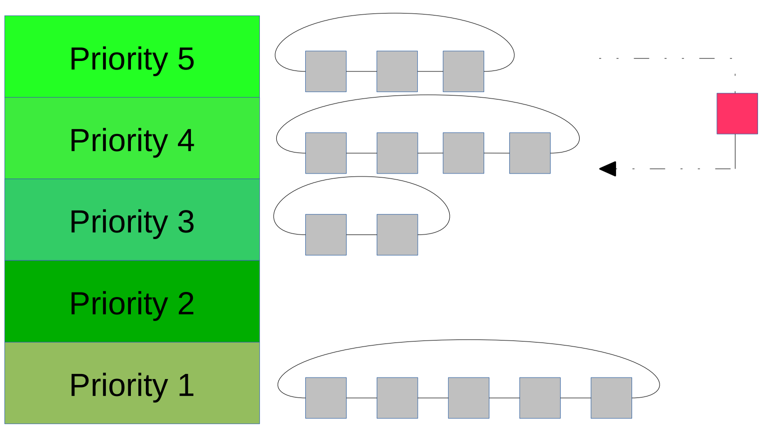


If there are more processes ready with the same priority level, they are executed with round-robin scheduling

Priority scheduling: Multilevel Feedback Queue (MLFQ)



When a job enters the system, it is given the highest assigned priority (it is inserted in the topmost queue according to its base priority)

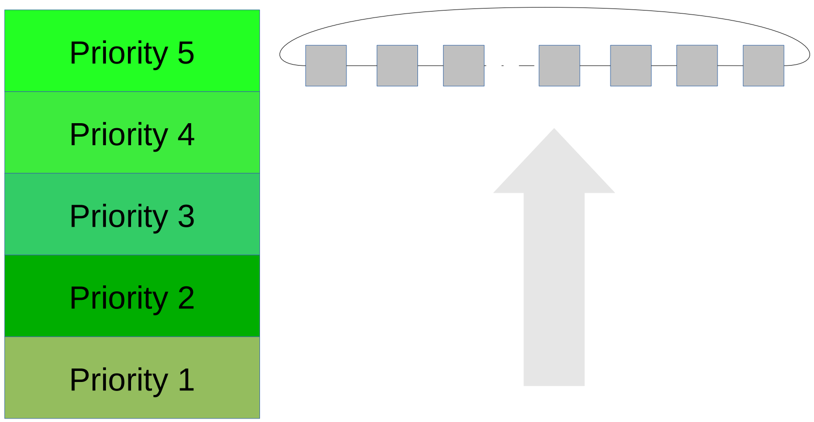


If a job uses up its given time (regardless of how many times it has yielded *), its priority is reduced (i.e., it moves down one queue) → ***aging***

** to avoid “cheaters” which yield at 99.9999% of their quantum*

Explaination

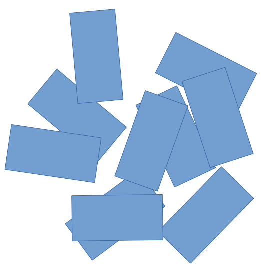
Priority scheduling: Multilevel Feedback Queue (MLFQ)



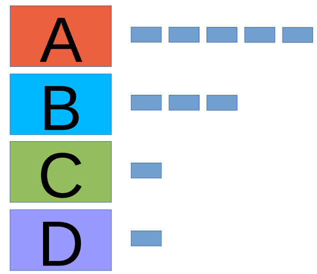
To prevent starvation of low priority processes, periodically, all processes are moved to the topmost queue corresponding to its base priority level → ***boost***

Explaination

Lottery scheduling



10 tickets



priority = 5, 5 tickets → 50% chance of being executed

priority = 3, 3 tickets → 30% chance of being executed

priority = 1, 1 ticket → 10% chance of being executed

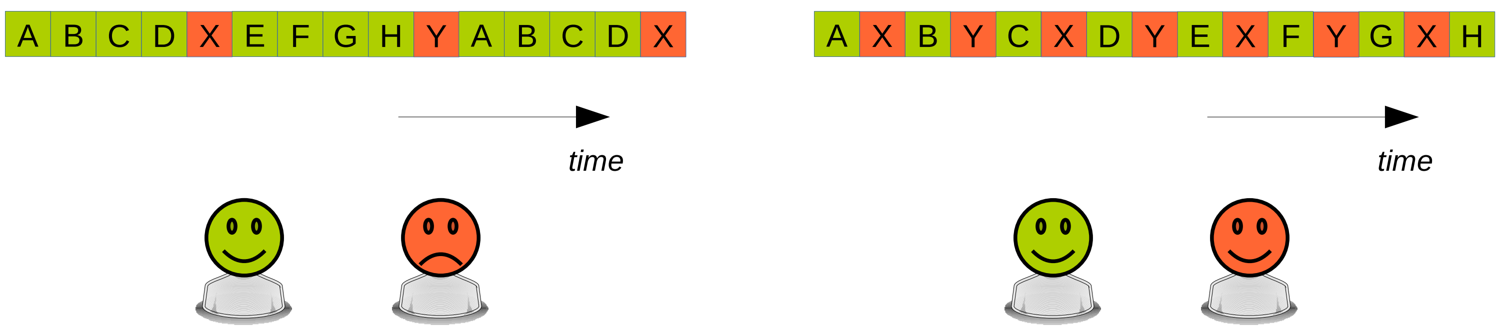
priority = 1, 1 ticket → 10% chance of being executed

- We assign to each process N (> 1) lottery tickets (proportional to their priority)
- Periodically (for example, each 20ms) we draw lots:
  - The process which has the winning ticket can execute for the next time slot
- Advantages:
  - Dynamic priorities are easy to implement
  - Processes can give away tickets to other processes
    - for example, a client process can give his tickets to a server process to speed up servicing during requests

Explaination

Fair-share scheduling

- The aforementioned scheduling policies are fair to processes but not to users:
  - For example, with 10 processes executed with round robin, a user with 8 processes will get 80% of the CPU, a user with 2 process will only get the remaining 20%.



- We could implement policies which divide CPU time among users, or assign priorities to users, independently from the number of their processes.

Explaination

What about multicore, multiprocessor scheduling?

Multicore / Multiprocessor scheduling

- Shared-memory multiprocessors provide more resources for executing threads and processes but they...
- ...introduce **data locality** problems
- ...transform scheduling into a **multi-dimensional problem**:
  - Uniprocessor scheduling: “which process/thread to run next?”
  - Multiprocessor scheduling: “which process/thread to run next, on which CPU?”

Question

What is locality?

Locality

- **Temporal locality**
  - when a piece of data is accessed, it is likely to be accessed again in the near future
- **Spatial locality**
  - when a piece of data at address X is accessed, it is likely that also nearby data is accessed

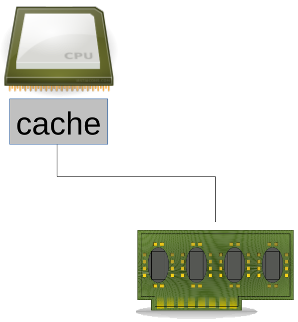
Answer

Why is locality so important when scheduling?

Question

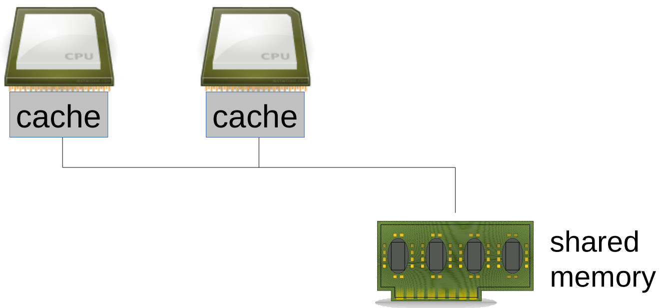
Locality in uniprocessor systems

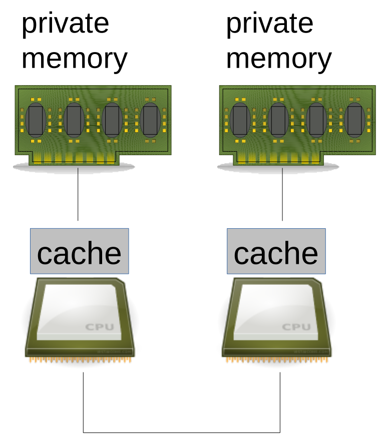
- When a thread has run for some time on a CPU the corresponding cache will contain most of the thread's data
- The **cache** works on the principle of locality to:
  - reduce the possibility of stalling the CPU while waiting for memory requests to be fullfilled (→ **hide memory latency**)
  - reduce the amount of data that needs to be transferred from the main memory (→ **decrease memory bandwidth**)



Answer

Locality in multi-processor systems





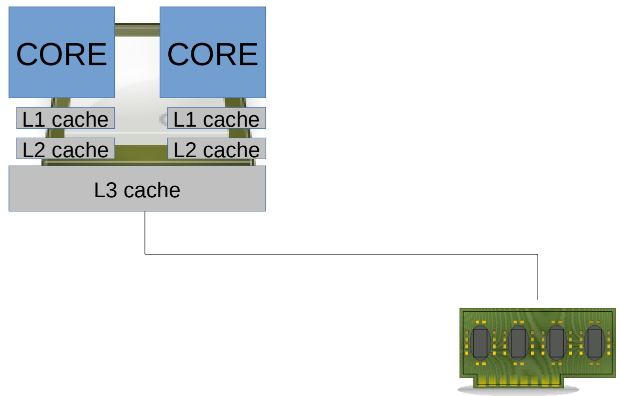
Uniform Memory Access (UMA) architecture
(Access to main-memory occurs at the same speed for all processors)

Nonuniform Memory Access (NUMA) architecture
(Access to some areas of the main-memory is faster for some processors than other parts of memory)

With UMA accessing data which is not in the local cache is slower; the same with NUMA when accessing data which is not in the *private* memory (one CPU has to ask another CPU)

Situation

Locality in multi-core systems



Multi-core

Uniform Memory Access (UMA) architecture

In multi-core systems accessing data which is not in the faster L1/L2 caches of a core results in slower performance

Situation

What can a scheduler do concerning locality?

Locality-aware scheduling

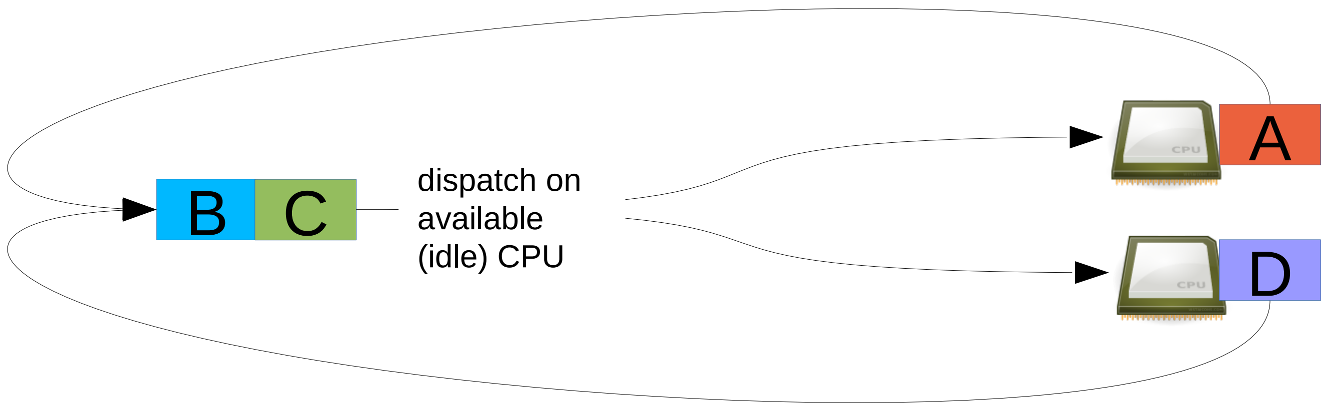
- Schedulers can implement locality-aware policies:
  - **Soft affinity:**
    - The scheduler *tries to* always assign a thread to the same CPU/core or to a specific set of CPUs/cores
  - **Hard affinity:**
    - The scheduler *is forced to* assign a thread to a specific CPU/core or to a specific set of CPUs/cores

Answer

Answer

Single queue scheduling

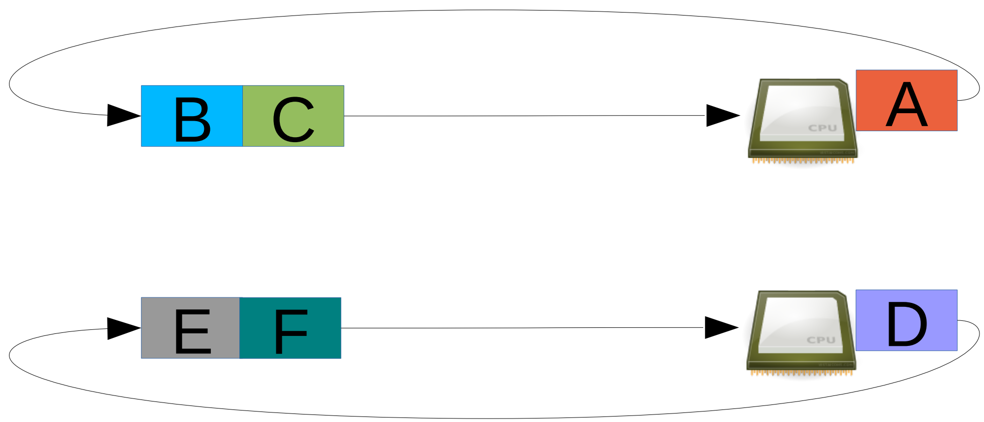
- All processes in a single queue



- All CPUs can be fully exploited (**load balancing**)
- Simple to implement but **not scalable**:
  - synchronization mechanisms are required to ensure safe access to the shared queue
- Introduce **locality issues**:
  - processes/threads move between CPUs/cores
  - cache affinity is compromised

Multiple queues scheduling

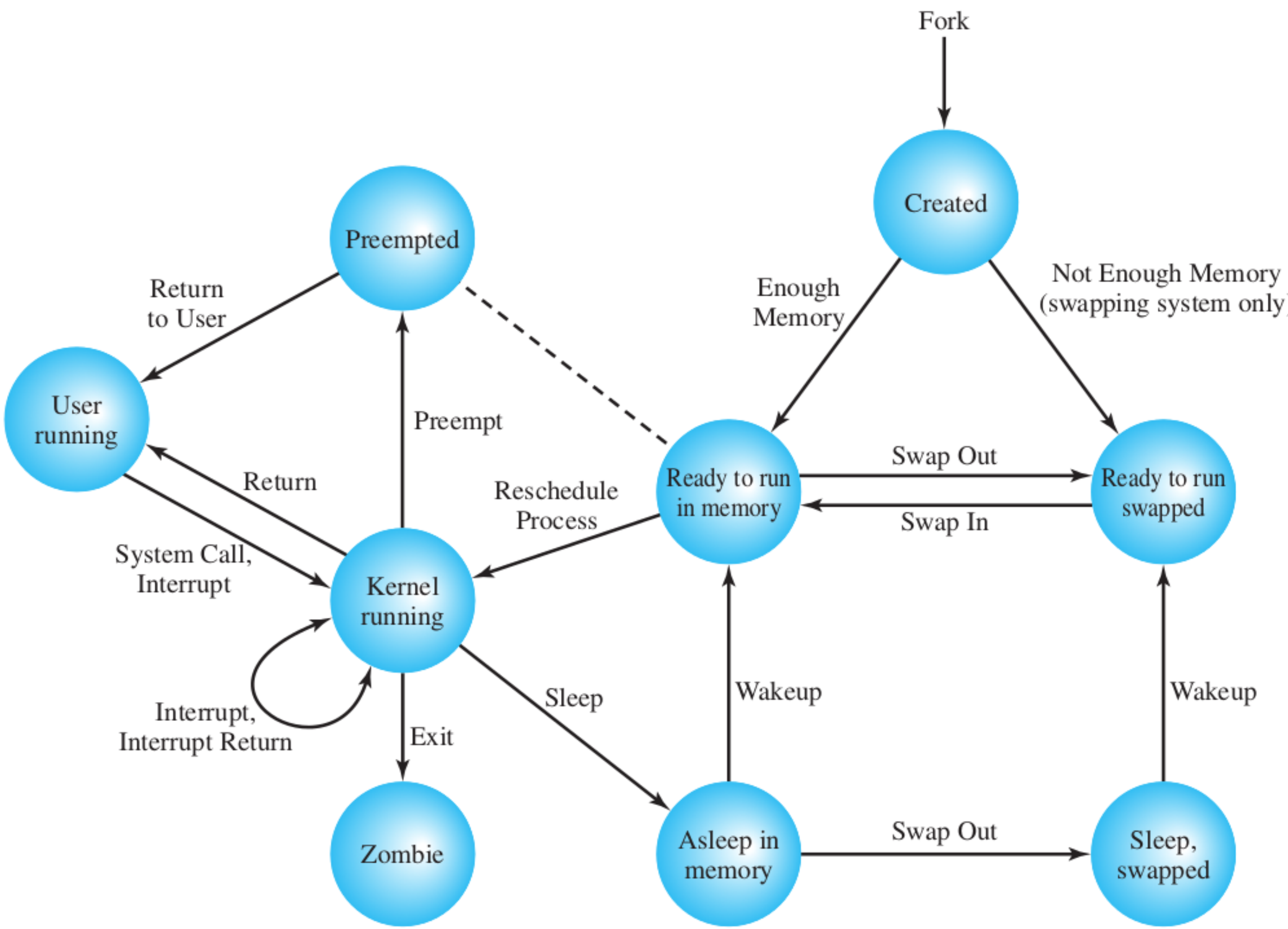
- Each processor/core has its own queue



- **Scalable**
- Can ensure **cache affility**
- Can lead to **load imbalance**, solved either with
  - **migration** of processes to less loaded CPUs
  - **work stealing** mechanism (CPU can take tasks from other queues when idle)

Answer

Unix scheduling: process states



From “Operating Systems: Internals and Design Principles”, 6/E, William Stallings, Prentice Hall, 2008

How does scheduling in real systems work?

Traditional Unix scheduling

- Multi-level feedback scheduler, round-robin at each priority level
- Preemption if process does not yield before 1s

$$CPU_j(i) = \frac{CPU_j(i-1)}{2}$$

$$P_j(i) = Base_j + \frac{CPU_j(i)}{2} + nice_j$$

Recompute once per second to determine rescheduling decision

where

$CPU_j(i)$ = measure of processor utilization by process j through interval i

$P_j(i)$ = priority of process j at beginning of interval i; lower values equal higher priorities

$Base_j$ = base priority of process j

$nice_j$ = user-controllable adjustment factor

From “Operating Systems: Internals and Design Principles”, 6/E, William Stallings, Prentice Hall, 2008

Explaination

Traditional Unix scheduling: example

Time	Process A		Process B		Process C	
	Priority	CPU count	Priority	CPU count	Priority	CPU count
0	60	0 1 2 • • 60	60	0	60	0
1	75	30	60	0 1 2 • • 60	60	0
2	67	15	75	30	60	0 1 2 • • 60
3						

60 + (60/2)/2

From "Operating Systems: Internals and Design Principles", 6/E, William Stallings, Prentice Hall, 2008

Explaination

Traditional Unix scheduling: example

Previous CPU count

Time	Process A Priority	Process A CPU count	Process B Priority	Process B CPU count	Process C Priority	Process C CPU count
3	63	7 8 9 • • 67	67	15	75	30
4	76	33	63	7 8 9 • • 67	67	15
5	68	16	76	33	63	7

60 + (67/2)/2

Explaination

From “Operating Systems: Internals and Design Principles”, 6/E, William Stallings, Prentice Hall, 2008

UNIX SVR4 scheduling

- 160 priority levels, divided into 3 priority classes (Round-Robin)
  - **Real time**: 159 – 100
    - Fixed priority and fixed quantum
  - **Kernel**: 99 – 60
  - **Time-shared**: 59 – 0
    - Dynamic priority (depends on quantum utilization)
    - Dynamic quantum (depends on priority)

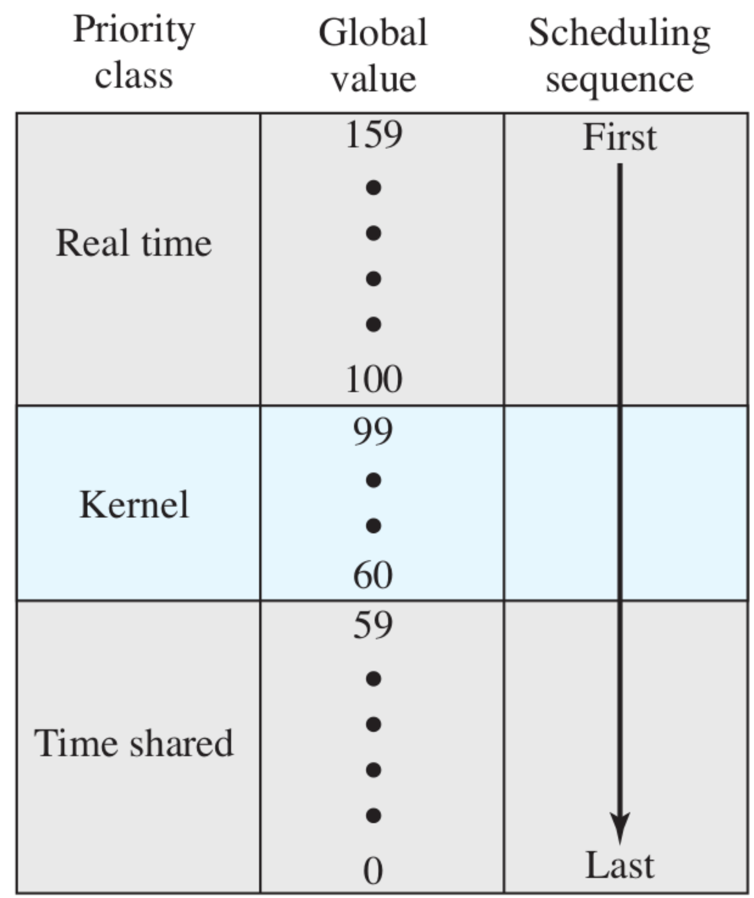


Figure 10.13 **SVR4 Priority Classses**

From "Operating Systems: Internals and Design Principles", 6/E, William Stallings, Prentice Hall, 2008

Explaination

SVR4 dispatch queues

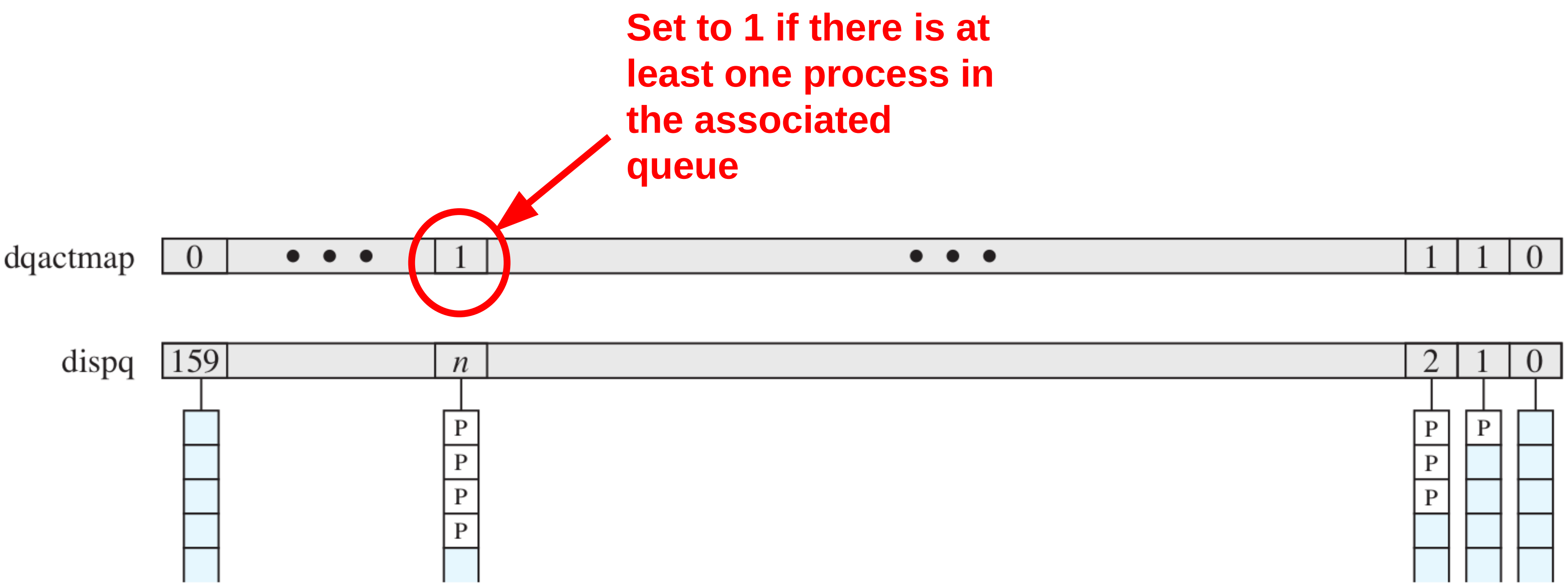
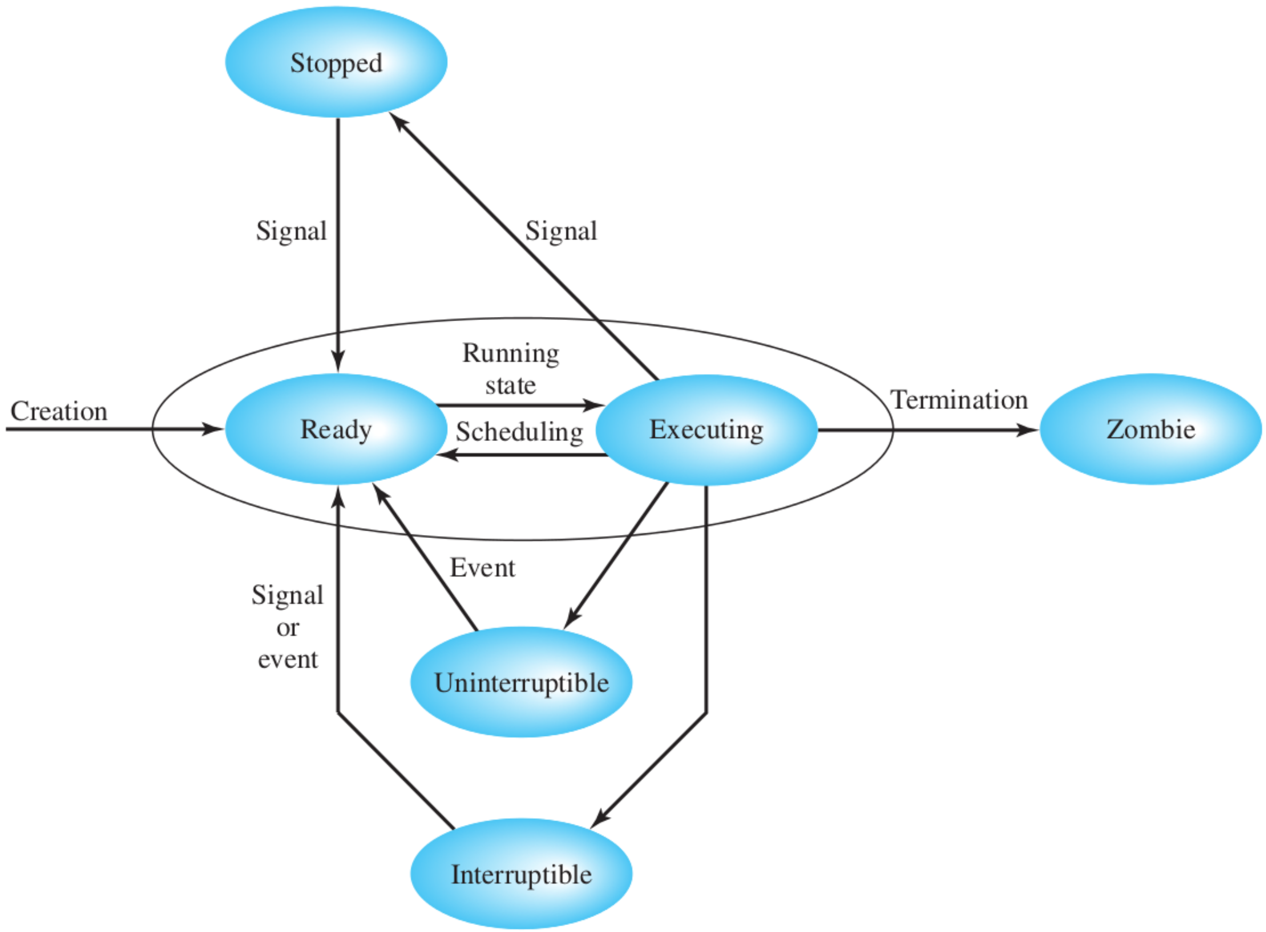


Figure 10.14 **SVR4 Dispatch Queues**

Explaination

Linux scheduling: thread states



From “Operating Systems: Internals and Design Principles”, 6/E, William Stallings, Prentice Hall, 2008

Linux schedulers

- Linux 2.4
  - Similar to traditional UNIX scheduler
- Linux 2.6
  - O(1) scheduler
- Since Linux 2.6.23
  - Completely Fair Scheduler (CFS)

Explaination

Linux scheduling policies*

- Normal policies (i.e., non-real-time):
  - **SCHED_OTHER** the standard round-robin time-sharing policy;
  - **SCHED_BATCH** for "batch" style execution of processes;
  - **SCHED_IDLE** for running very low priority background jobs.

- Real-time policies:
  - **SCHED_FIFO** a first-in, first-out policy;
  - **SCHED_RR** a round-robin policy.
  - **SCHED_DEADLINE** Earliest Deadline First policy (since Linux 3.14)

see http://man7.org/linux/man-pages/man7/sched.7.html

Explaination

SCHED_FIFO

1. The system will not interrupt an executing FIFO thread except in the following cases:
   a. Another FIFO thread of higher priority becomes ready.
   b. The executing FIFO thread becomes blocked waiting for an event, such as I/O.
   c. The executing FIFO thread voluntarily gives up the processor following a call to the primitive `sched_yield`.
2. When an executing FIFO thread is interrupted, it is placed in the queue associated with its priority.
3. When a FIFO thread becomes ready and if that thread has a higher priority than the currently executing thread, then the currently executing thread is preempted and the highest-priority ready FIFO thread is executed. If more than one thread has that highest priority, the thread that has been waiting the longest is chosen.

Explaination

From "Operating Systems: Internals and Design Principles", 6/E, William Stallings, Prentice Hall, 2008

SCHED_RR vs SCHED_FIFO

A	Minimum
B	Middle
C	Middle
D	Maximum

(a) Relative thread priorities

D ⟶ B ⟶ C ⟶ A ⟶

(b) Flow with FIFO scheduling

D ⟶ B ⟶ C ⟶ B ⟶ C ⟶ A ⟶

(c) Flow with RR scheduling

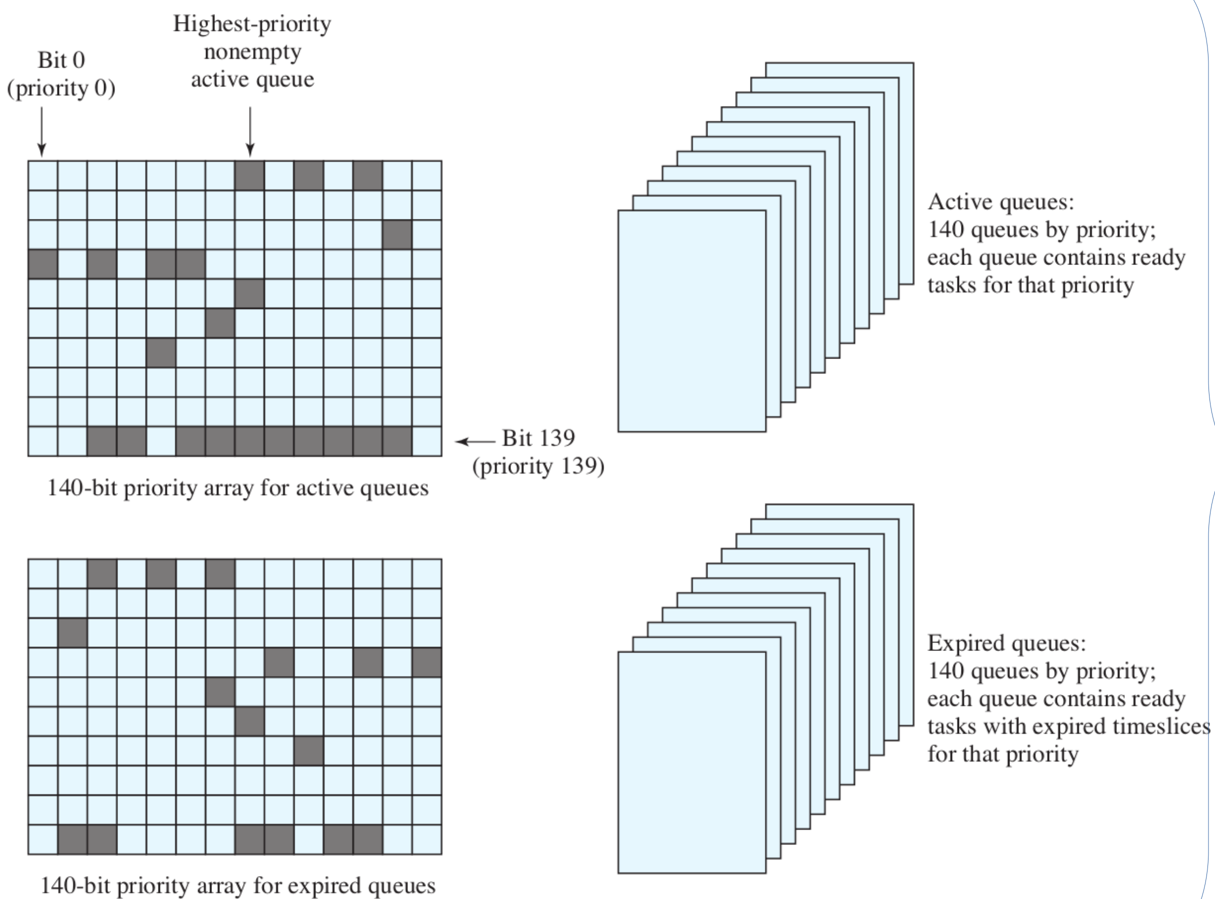
Figure 10.11 Example of Linux Real-Time Scheduling

From “Operating Systems: Internals and Design Principles”, 6/E, William Stallings, Prentice Hall, 2008

Linux O(1) scheduler goals

- Background ideas
  - Separate process queue for each processor
  - Periodical rebalancing of queues
  - Adjust priority depending on process behavior (run time vs wait time)
  - Quantum related to priority: higher priority threads receive larger quantums
  - Process has **affinity mask**
  - **Constant** rescheduling time (O(1))

Linux O(1) scheduler



Data structures for each processor

From "Operating Systems: Internals and Design Principles", 6/E, William Stallings, Prentice Hall, 2008

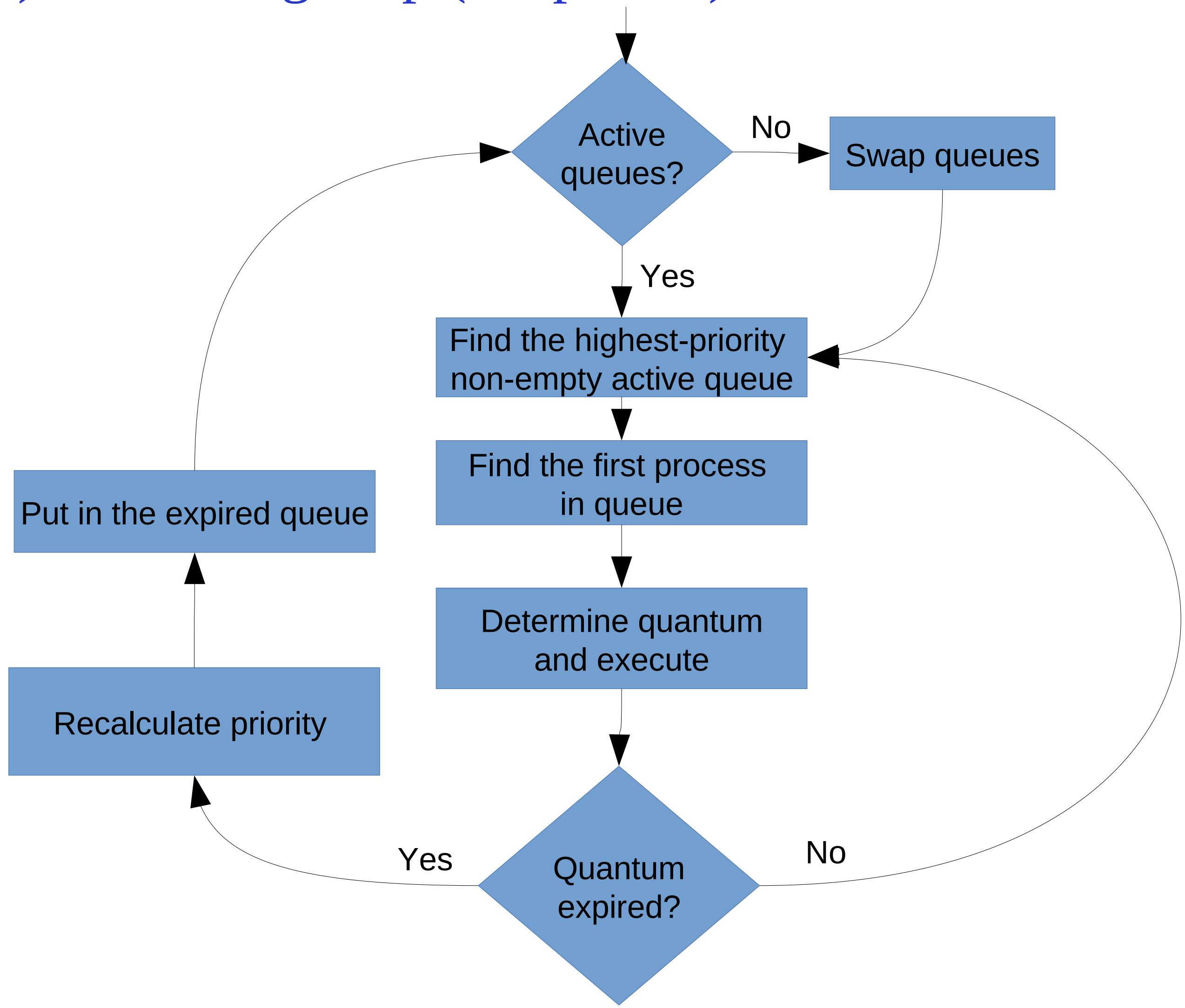
Explaination

Priorities

- **Non-realtime tasks**
  - Initial priority between 100 and 139 (default 120), can be adjusted with **`nice`**
  - Dynamic priority depending on I/O activities: I/O bound processes are given higher priority
  - Quantums vary between 10ms and 200ms (higher priority tasks receive larger quantums)
  - When a task consumes all its quantum it is put in the expired queue
- **Real time tasks**
  - Static priority (between 0 and 99)
  - Execute until termination (even SCHED_RR): when quantum expires the value is reset and the task is kept in the active queue.

Explaination

Linux O(1) scheduling loop (simplified)



Linux O(1) scheduler advantages

- Priority is recalculated <u>only when the quantum expires</u> and <u>only for the current process</u>
- Finding the highest-priority non-empty active queue is fast
  - scan the bit array for active queues looking for the first 1
    - → 140 bits are ≈ 5 integers 32-bit
    - → BSR instruction on x86
- Queues are modified only when quantum expires
- Swap queues as simple as swapping pointers

Explaination

Linux CFS (since 2.6.23) *

- **C**ompletely **F**air **S**cheduler

- Implements three non-realtime scheduling policies:
  - SCHED_NORMAL (a.k.a SCHED_OTHER)
  - SCHED_BATCH (for long running tasks)
  - SCHED_IDLE (very low priority tasks)

- Tries to ensure **fairness** by giving each non-realtime process a fair amount of CPU-time

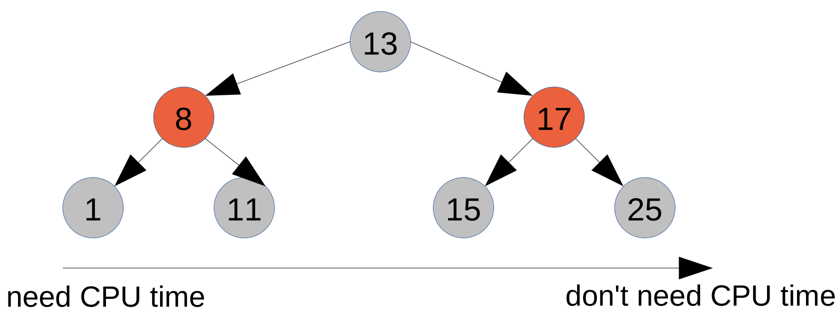
- SCHED_FIFO, SCHED_RR similar to O(1), but simplified to 100 active queues, no expired queue

* Detailed information: https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt

Explaination

Linux CFS (since 2.6.23)

- Instead of a queue it uses a per-CPU time-ordered **red-black tree** (thus self-balancing) :
  - each node corresponds to a task, its weight being the **virtual runtime** (run time in relation to the number of running processes) adjusted depending on the nice value of the task
  - search, insertion and deletion time complexity is **O(log n)**



- **Idea:** When scheduling, remove nodes from the left-most part ot the tree, execute, recompute virtual time and insert back in the tree

Explaination

Windows scheduling: thread states

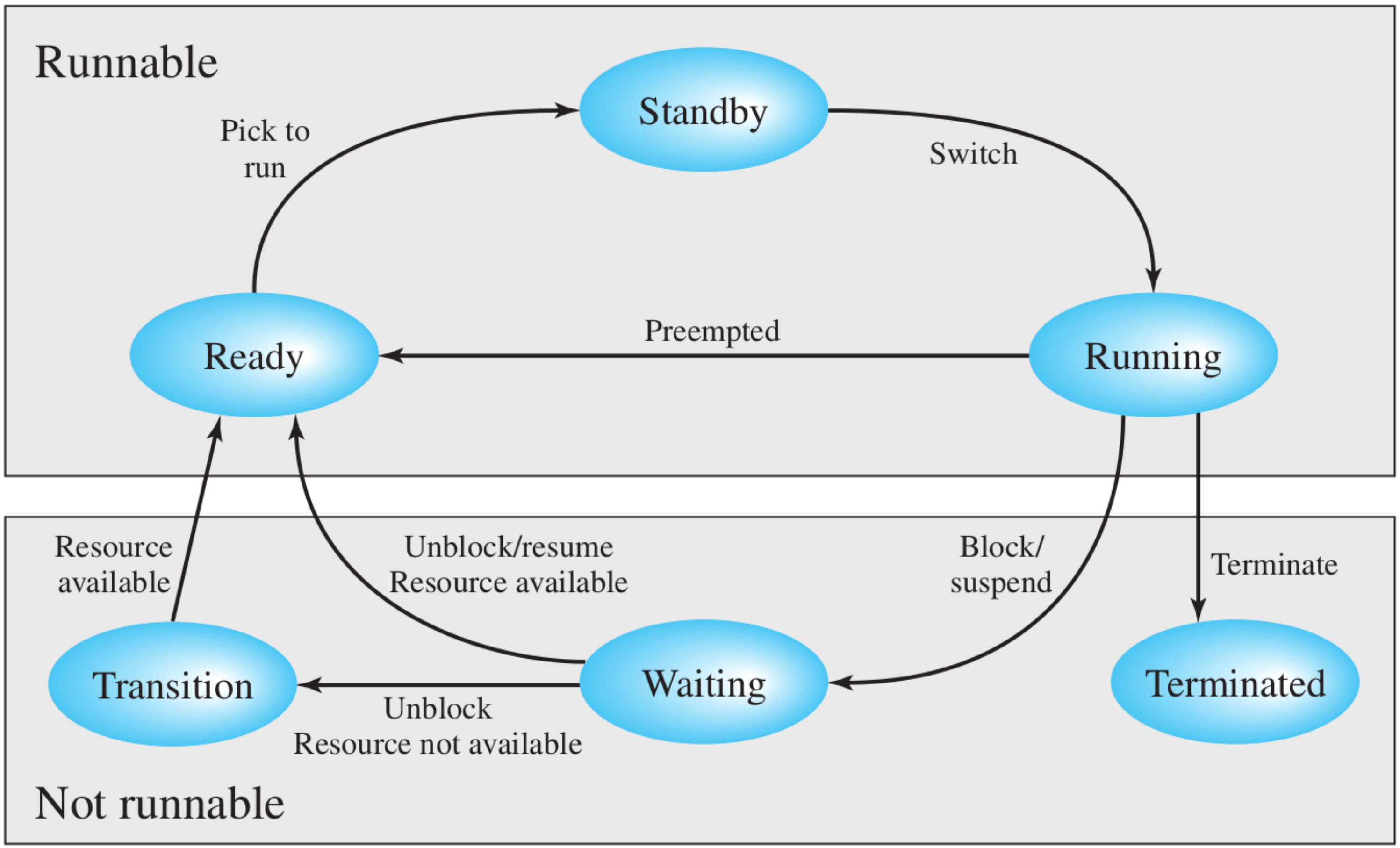


Figure 4.14 Windows Thread States

From “Operating Systems: Internals and Design Principles”, 6/E, William Stallings, Prentice Hall, 2008

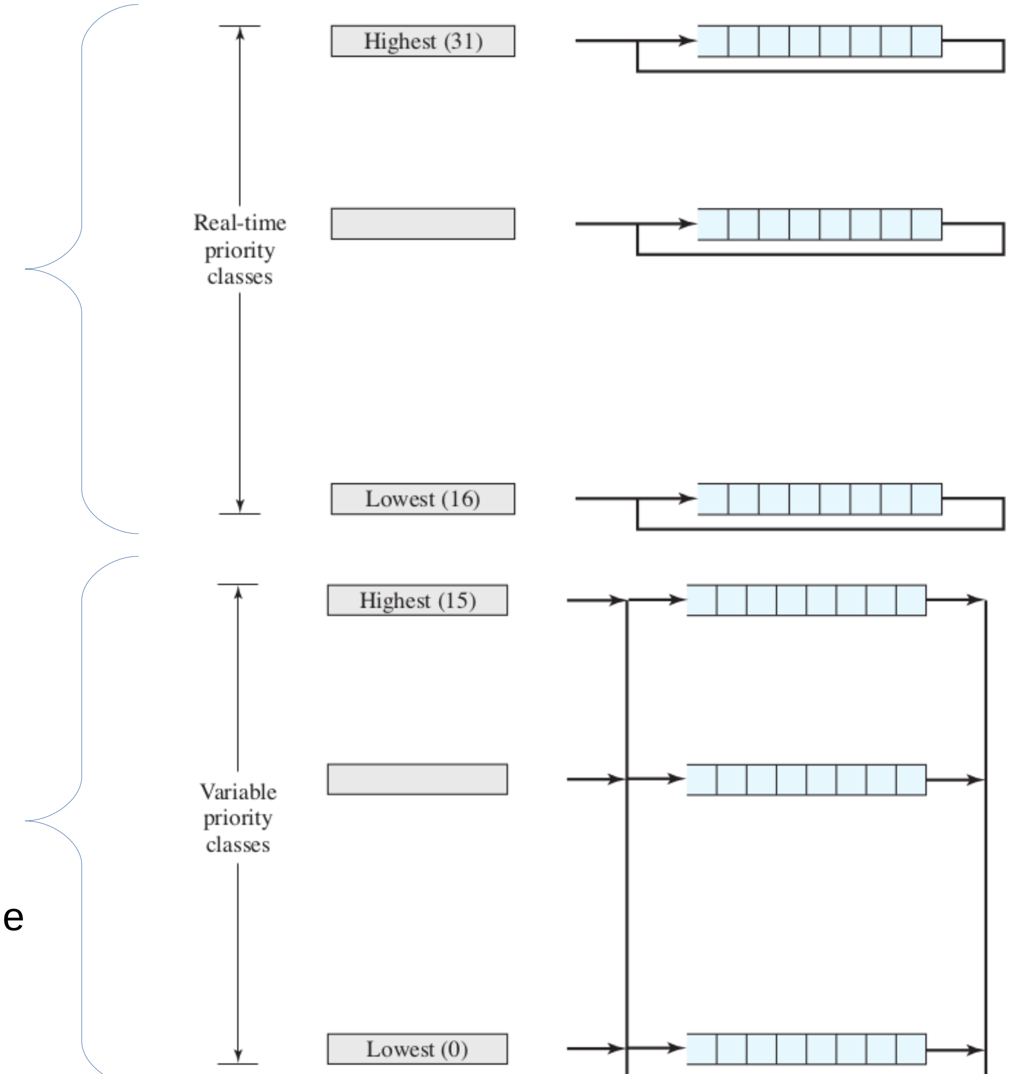
Priority classes

Real-time threads

- Fixed priority (16-31)
- Round-robin queues

Variable priority threads

- Variable priority (0-15)
- The priority class and priority level are combined to form the base priority of a thread
- Priority boost
- Round-robin queues

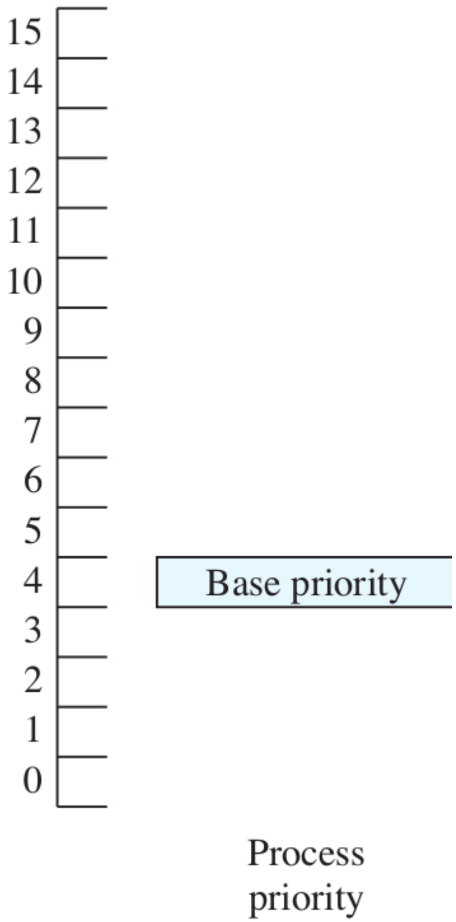


From "Operating Systems: Internals and Design Principles", 6/E, William Stallings, Prentice Hall, 2008

Explaination

Process priority classes

Image from "Operating Systems: Internals and Design Principles", 6/E, William Stallings, Prentice Hall, 2008



- Each process belongs to a **process priority** class:
  - IDLE_PRIORITY_CLASS
  - BELOW_NORMAL_PRIORITY_CLASS
  - NORMAL_PRIORITY_CLASS (default)
  - ABOVE_NORMAL_PRIORITY_CLASS
  - HIGH_PRIORITY_CLASS
  - REALTIME_PRIORITY_CLASS

details at http://msdn.microsoft.com/en-us/library/windows/desktop/ms685100%28v=vs.85%29.aspx

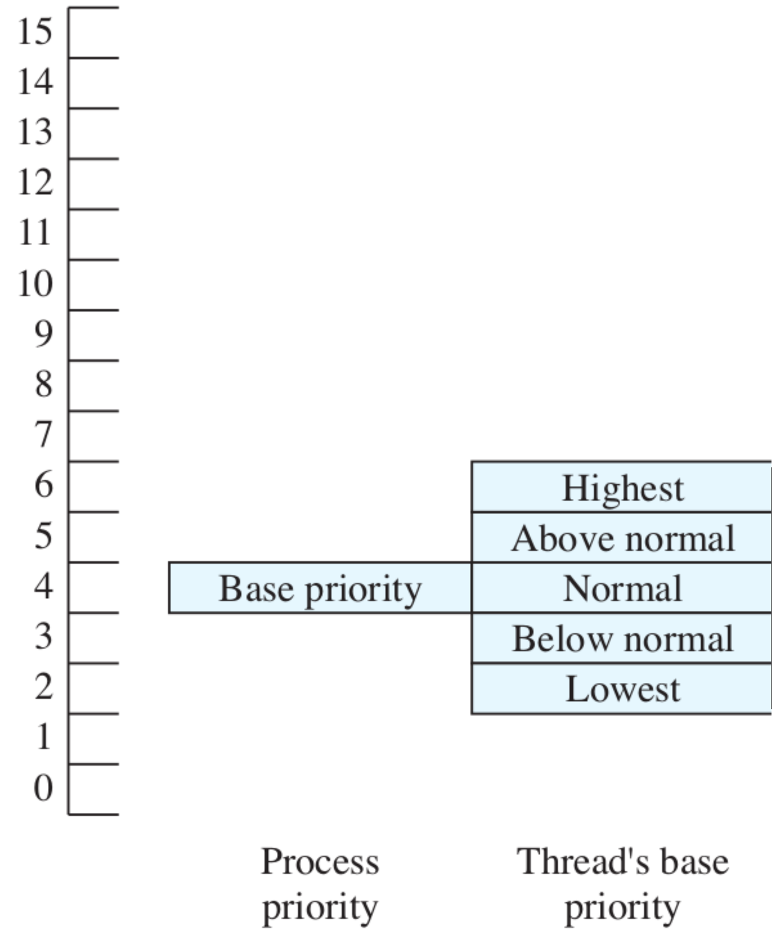
Explaination

Thread priority levels

- Threads within a process are assigned with a **thread priority level**:
  - THREAD_PRIORITY_IDLE (always 1 for non-RT, 16 for RT)
  - THREAD_PRIORITY_LOWEST
  - THREAD_PRIORITY_BELOW_NORMAL
  - THREAD_PRIORITY_NORMAL (default)
  - THREAD_PRIORITY_ABOVE_NORMAL
  - THREAD_PRIORITY_HIGHEST
  - THREAD_PRIORITY_TIME_CRITICAL (always 15 for non-RT, 31 for RT)

- Process priority class
  + thread priority level
  = **Thread's base priority**

Image from "Operating Systems: Internals and Design Principles", 6/E, William Stallings, Prentice Hall, 2008



details at http://msdn.microsoft.com/en-us/library/windows/desktop/ms685100%28v=vs.85%29.aspx

Explaination

Improving responsiveness

- To improve responsiveness some threads might receive a temporary **priority boost*** :
  - When a process that uses **NORMAL_PRIORITY_CLASS** is brought to the foreground (so that dynamic priority ≥ priority of any background process)
  - When a window receives input
  - When a thread blocked on I/O becomes ready

- Thread's base priority + Boost
  = Thread's dynamic priority (used by the scheduler)

* After a boost, the dynamic priority decays each time the thread completes a time slice, until it returns to its base priority.

Explaination

Thread's base priority

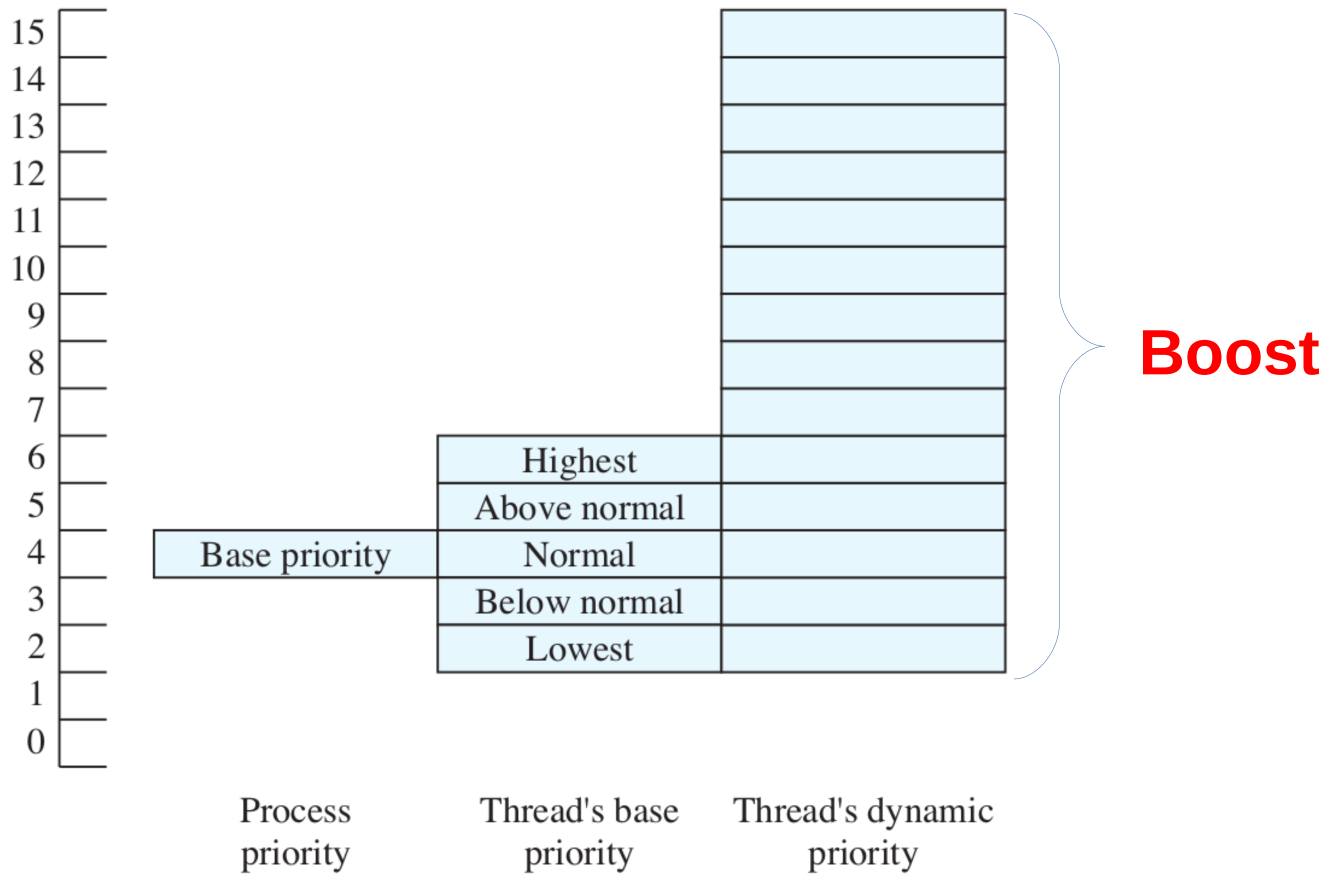


Figure 10.16 **Example of Windows Priority Relationship**

From “Operating Systems: Internals and Design Principles”, 6/E, William Stallings, Prentice Hall, 2008

Wrap Up