# Operating Systems

Complutense University of Madrid

2015-2016

## Unit 3.2: OS Scheduling

# Contents

# Contents

# Scheduling

## Objectives

- Optimize CPU usage
- Reduce waiting times (latency)
- Guarantee an even distribution of CPU cycles among processes (*fairness*)
- Support user-defined priorities

## Types of scheduling algorithms

- **Non Preemptive**: The process stays on the CPU until (1) blocking, (2) yielding the CPU deliberately or (3) completion
- **Preemptive**: The OS can evict (*preempt*) the process from the CPU
  - require a system timer that interrupts periodically

# Data structures in the scheduler

## Data structures

- The scheduler maintains processes in a **run queue** of processes/threads
  - Typically implemented as a doubly-linked list of PCBs
- The run queue consists of PCBs of processes in the "Ready" state
  - Usually, the PCB of a process running on the CPU is not in the run queue
  - The scheduler does not really care about threads in the "Waiting" state
- Some scheduler implementations maintain several run queues
  - By priority, by type, ...

# Activation of the OS scheduler

## Scheduler activation points

- Periodically (interrupt raised by the system timer on a CPU)
- As a result of an interrupt raised by an I/O device
- The current process running on a CPU causes an exception that blocks it (e.g., page fault) or forces its termination (e.g., segmentation fault)
- A process running on a CPU terminates or requests a blocking operation (e.g., blocking system call)
- The current process relinquishes the CPU
  - `sched_yield()`
- A process with a higher priority than the one currently running enters the "Ready" state
  - User preemption

# Scheduler-related metrics

## Entity-specific metrics (per-process/per-thread)

- Completion time, Real time or Turnaround time
  - $CompletionTime = T_{end} - T_{start}$
- Waiting time: Total time the process/thread spends sitting on a run queue throughout the execution
- Response time:
  - $ResponseTime = T_{mappedToCPU} - T_{ready}$

## System-wide metrics

- Percentage of CPU usage
- Throughput: number of jobs completed per unit time

# Contents

# Non-preemptive algorithms

- The scheduler does not evict a process running on the CPU
  - To release the CPU, the process must terminate, block (e.g., I/O operation) or relinquish the CPU

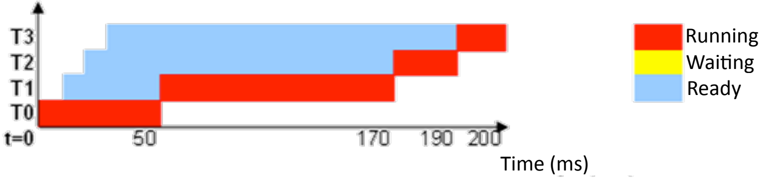
## Algorithms

- **FCFS**: *First-Come First-Served*
- **SJF**: *Shortest Job First*
  - Also known as SPN (*Shortest Process Next*)
- **Priority based**

# First-Come First-Served (FCFS)

- Run queue managed as a regular FIFO queue
- Very simple algorithm that optimizes CPU usage

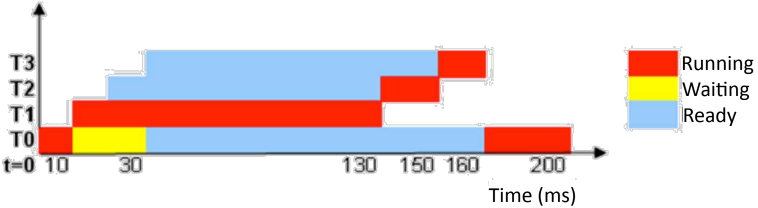
| Process or Thread | Arrival time (ms) | CPU time (ms) |
|---|---|---|
| T0 | 0 | 50 |
| T1 | 10 | 120 |
| T2 | 20 | 20 |
| T3 | 30 | 10 |



# First-Come First-Served (FCFS)

- Processes performing I/O operations are pushed at the end of the queue
- Long-running processes lead to increased waiting times

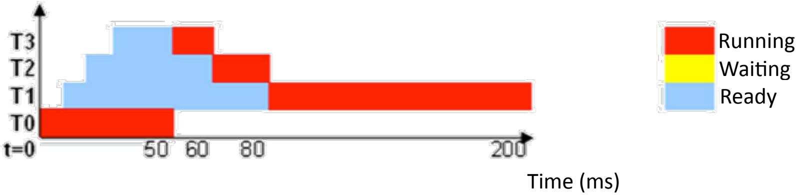
| Process or Thread | Arrival time (ms) | CPU time (ms) |
|---|---|---|
| T0 | 0 | 50 |
| T1 | 10 | 120 |
| T2 | 20 | 20 |
| T3 | 30 | 10 |



# Shortest Job First (SJF)

- Suitable for interactive processes
- Requires to know tasks' execution profiles beforehand
- Subject to starvation issues

| Process or Thread | Arrival time (ms) | CPU time (ms) |
|---|---|---|
| T0 | 0 | 50 |
| T1 | 10 | 120 |
| T2 | 20 | 20 |
| T3 | 30 | 10 |



# Priority-based policy

- The user can tell the OS which processes are the most important in the workload
- Subject to starvation issues
  - Workaround: Increase priority with age or waiting time

| Process or Thread | Arrival time (ms) | CPU time (ms) | Priority |
|---|---|---|---|
| T0 | 0 | 50 | 4 |
| T1 | 10 | 120 | 3 |
| T2 | 20 | 20 | 1 |
| T3 | 30 | 10 | 2 |

# Preemptive algorithms

- Non-preemptive algorithms are not suitable for general-purpose OSes
  - Both compute-bound processes and interactive processes may be included in the same workload
- A preemptive scheduler is activated periodically
  - The *system timer* is configured to raise periodic interrupts on a per-CPU basis (~ms)
    - Each interrupt is referred to as a *tick*
    - Default setting on Linux/x86: 250 *ticks* per second (4ms)

**Algorithms**

- **RR**: Round Robin
- **SRTF**: Shortest Remaining Time First
- **Preemptive Priority-based scheduling**
- **Multi-level queue**

# Round Robin - RR (I)

- Scheduling is done by dividing CPU time into equal-size intervals referred to as quantum or *time slice* (expressed in ticks)
- RR: $\rightarrow$ FCFS $+$ time slice
  - The scheduler preempts the current process in the event it consumes the allowed time slice
  - When a process is preempted, the scheduler pushes it back at the end of the queue
  - Implementation: each process is assigned a *tick* counter
    - The counter is initialized with the time slice value and decremented on each tick consumed by the process on the CPU
    - Preemption $\Longleftrightarrow$ tick counter $= 0$

Running Process

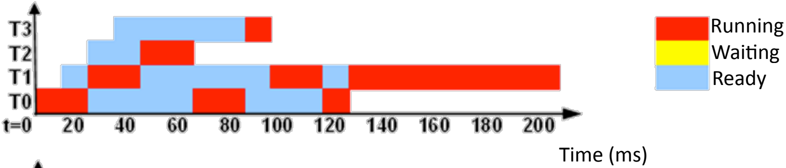
| 5 | 8 | 13 | 3 | 37 | 72 | 2 |
|---|---|---|---|---|---|---|

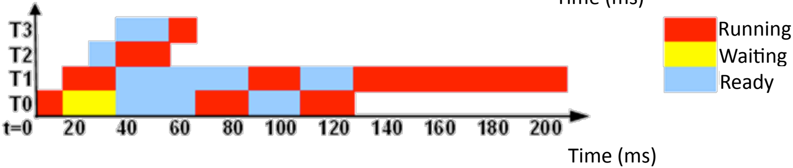
Running Process

| 8 | 13 | 3 | 37 | 72 | 2 | 5 |
|---|---|---|---|---|---|---|

# Round Robin - RR (I)

| Process or Thread | Arrival time (ms) | CPU time (ms) |
|---|---|---|
| T0 | 0 | 50 |
| T1 | 10 | 120 |
| T2 | 20 | 20 |
| T3 | 30 | 10 |

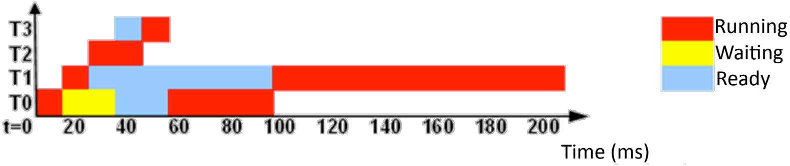


# Shortest Remaining Time First (SRTF)

- SRTF: SJF + preemption
  - Suitable for interactive processes
  - Requires to know tasks' execution profiles beforehand
  - Subject to starvation issues

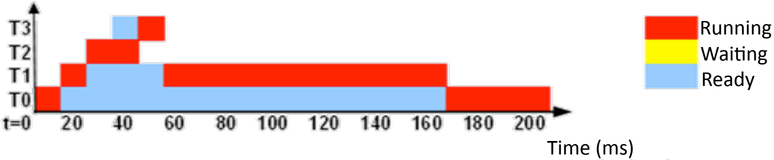
| Process or Thread | Arrival time (ms) | CPU time (ms) |
|---|---|---|
| T0 | 0 | 50 |
| T1 | 10 | 120 |
| T2 | 20 | 20 |
| T3 | 30 | 10 |



# Preemptive priority-based policy

- The user can tell the OS which processes are the most important in the workload
- Subject to starvation issues
  - Workaround: Increase priority with age or waiting time

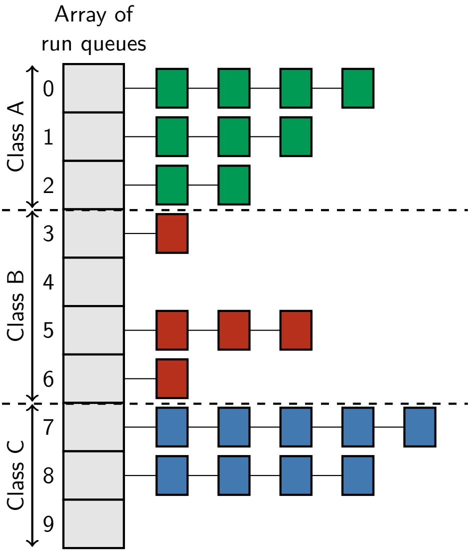
| Process or Thread | Arrival time (ms) | CPU time (ms) | Priority |
|---|---|---|---|
| T0 | 0 | 50 | 4 |
| T1 | 10 | 120 | 3 |
| T2 | 20 | 20 | 1 |
| T3 | 30 | 10 | 2 |



# Multilevel queue scheduling (I)

- Goal: provide support for different types of processes
- The scheduler supports *k* priority levels
  - A separate run queue is maintained for each level (array of queues)
  - A different *time slice* may be defined for each priority level
- The various priority levels are divided into ranges associated with different application types
  - Real time
  - System
  - Interactive
  - Batch
  - ...

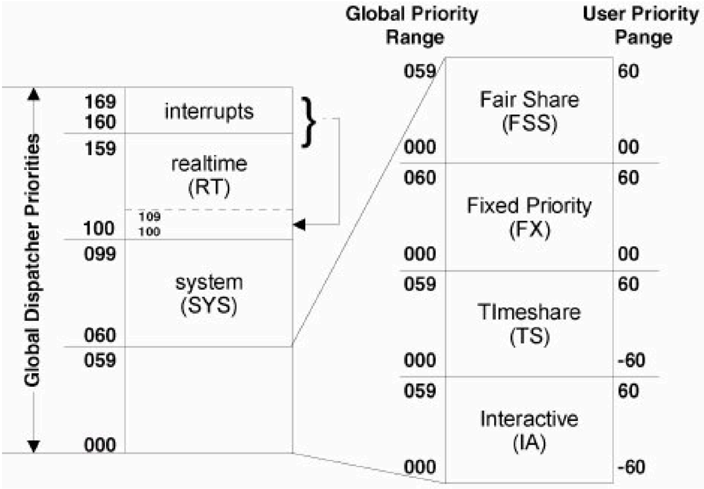
# Multilevel queue scheduling (II)



# Multilevel queue scheduling (III)

- Scheduling is performed at two levels:
  - Global (*dispatcher*): Selects which process will run next and performs context switches
    - The dispatcher always selects the highest priority runnable process in the system
  - Local (*scheduling class*): Manages the run queues associated with a given priority range (specific type of processes)
    - Time slice management and tick processing
    - Invokes the dispatcher to trigger user preemptions

# Multilevel queue scheduling: Solaris OS



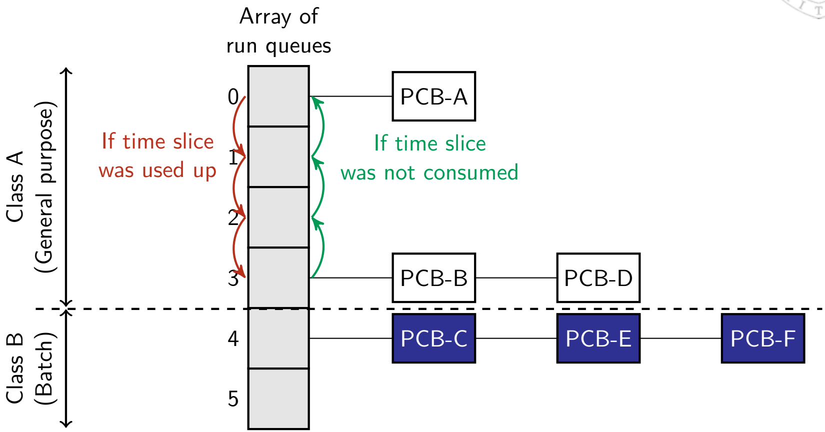
# Multilevel queue scheduling (IV)

## Alternatives to manage run queues within a priority range

1. Without feedback (fixed priority scheme)
   - Each process stays in the same run queue for its entire life-cycle
2. With feedback (dynamic priority scheme)
   - Processes can be assigned a different priority (level) over time
     - Priority values always fall within the range of priorities associated with the scheduling class
   - Changes in priority driven by a given policy
     - Example: policy to favor interactive processes over CPU-intensive processes

```
If process consumes time slice -> prio_level--
If process blocks before consuming time slice -> prio_level++
```

# Multilevel queue scheduling: Example

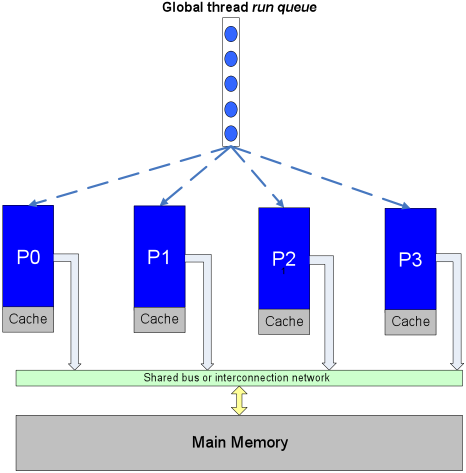


# Contents

# SMP scheduling

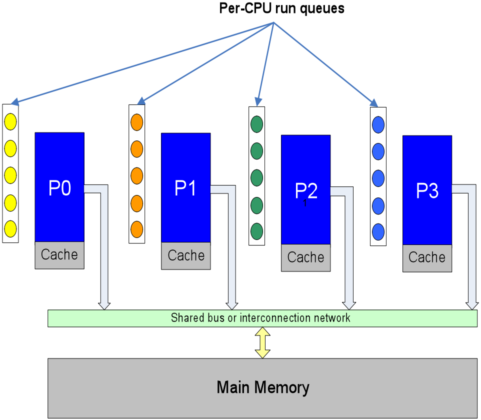
- Enforce load balance across CPUs
  - Avoid having idle CPUs while other CPUs have a high load
- Take processor affinities into consideration
  - Especially important when selecting the CPU where a process runs
  - Avoid thread migrations when possible
- Factor in data locality on NUMA systems (different memory nodes)
  - The scheduler should map a thread in a CPUs close to the memory node where most of the application data has been allocated

# SMP scheduling (Linux v2.4.x)




- Single run queue for the entire system
- Good for load balance
  - Each CPU potentially has the same amount of work
- Affinity-unaware scheme
  - Process A runs in CPU1 and then is moved onto CPU1 (*migration*)
  - Migrations may require rebuilding cache state
- Scalability issues

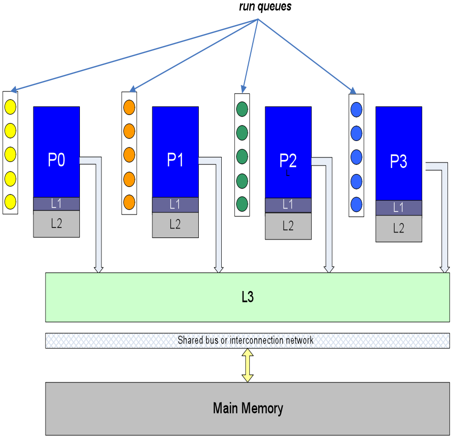
# SMP scheduling (Linux v2.6.x+)




- A run queue is maintained for each CPU
- Better scalability
- Necessary to enforce load balance across run queues
  - A load balancer gets invoked periodically or on demand
  - Takes affinity into account

*Most modern general-purpose OSes (Linux, Solaris, FreeBSD, MS Windows,...) rely on this model to schedule threads on SMP environments*

# Scheduling in multicore systems




- The OS sees each core as an independent processor...
  - but cores share resources such as cache levels
- Potential performance degradation due to contention on shared resources
- Fairness-related issues

*Scheduling in multicore systems: active research area*

# Contents

# Scheduling on Linux (v3.14)

## 140 priority levels (0 → highest priority)

- 100 levels for real-time processes (deadline, RR and FIFO)
- 40 levels for regular user processes (CFS)
  - Priority can be changed with the `nice` command → [-20,19]

## Objectives *Completely Fair Scheduler* (CFS)

- Strives to ensure an even distribution of the CPU time by factoring in application priorities
  - CFS does not rely on time slices
- Enforce good response times
  - Well-suited to interactive environments (e.g.,GUIs)

# Completely Fair Scheduler (CFS)

## Overall idea

- Distribute CPU time among processes/threads in a similar way as if they could all run simultaneously
- If 4 equal-priority threads run in the system for 10 ms, each thread should run for 2.5 ms to ensure an even distribution (*fairness*)
  - What if threads have different priorities?

# CFS: CPU-time distribution

- Execution time is divided into variable-size intervals referred to as *scheduler periods*
  - Within a sched period, every runnable process must get a chance to run on the CPU for some time
- In a sched period, for each process P:
  - $T_{CPU}(P) = sched\_period\_ms \cdot \frac{weight(P)}{\sum_{i=1}^{n} weight(i)}$

**Example**

- 3 runnable processes (A,B and C) with weights 2,2 and 1, respectively
- sched_period_ms=20ms
- $T_{CPU}(A) = T_{CPU}(B) = 8ms$ and $T_{CPU}(C) = 4ms$

# CFS: CPU-time distribution

- If sched periods were assigned a fixed length (e.g., 20ms) and many runnable process exist on the system $T_{CPU}(P_i) \approx 0$
  - Very frequent context switches
  - Note that the scheduler reacts at the *tick* granularity (e.g., 4ms)

**Approximating completely fair schedule**

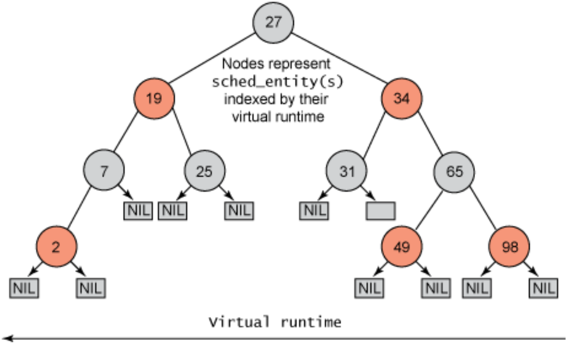
- The length of a sched period is established taking into account the number of runnable processes and other parameters
- Each process runs for a certain amount of time (`min_granularity`) without being preempted
  - After that time, the scheduler checks whether this process should be preempted or not
- A process can leave the CPU sooner than expected due to other reasons (I/O, relinquish the CPU voluntarily,...)

# CFS: Selecting the next process to run

- CFS keeps track of the amount of virtual CPU time (*vruntime*) that a process has received
  - A process's *vruntime* gets incremented every time that it consumes one *tick* or a fraction of a *tick* (e.g., when the process blocks)
  - The *vruntime* increases at a faster rate for low-priority processes and at a slower rate for high-priority processes
    - $\text{Virtual\_time\_unit}(P) = \text{Real\_time\_unit} \cdot \dfrac{Weight_{nice=0}}{Weight_P}$
    - $Weight_{nice=0}$ : Weight of a process with the default priority
- The scheduler tries to even out vruntimes across threads
  - The process with the smallest vruntime is selected to run next

# CFS: Selecting the next process to run

- CFS maintains processes in the run queue sorted in ascending order by vruntime
  - There is a run queue for each CPU, including all runnable processes assigned to that CPU (possibly with different priorities)
- Due to efficiency issues a redblack tree is used to implement the run queue → Balanced tree: operations $O(logN)$



# CFS: Putting all together

## Algorithm outline

- As a process runs its *vruntime* increases in accordance with its priority
  - The vruntime remains the same while a process sits on the run queue
- When a process $P$ has been running longer than a `min_granularity` interval without being preempted, the scheduler periodically checks whether this process *deserves* to continue running or not
  - If $vruntime(P) > min\_vruntime\_in\_run\_queue \rightarrow$ preemption
- When a process is preempted, CFS selects the process with smallest vruntime in the run queue to run next