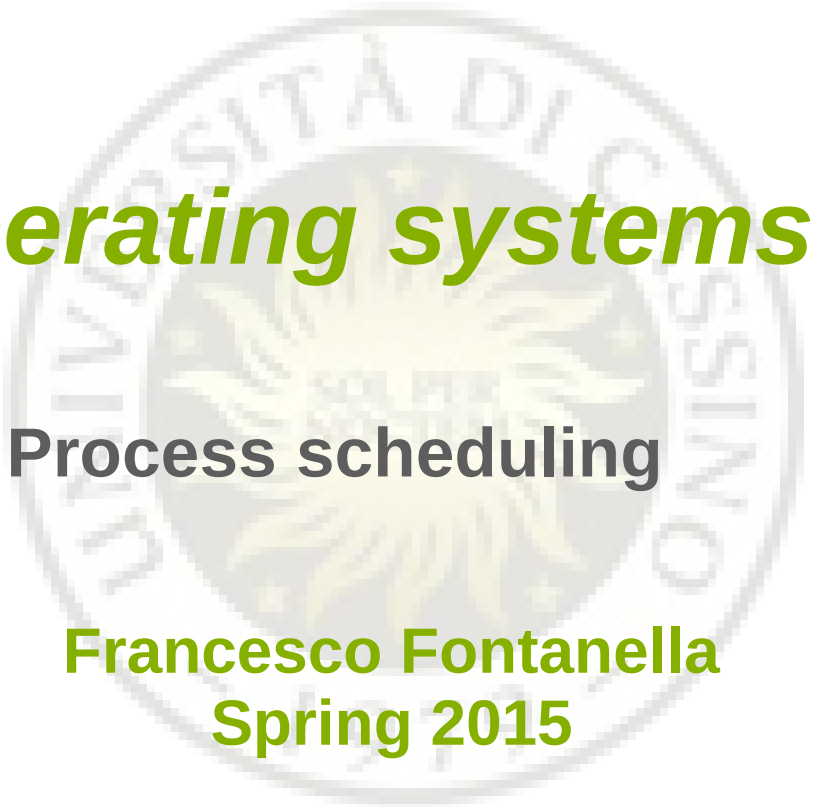


Operating systems

Process scheduling

Francesco Fontanella

Spring 2015

Dispatcher vs. scheduler

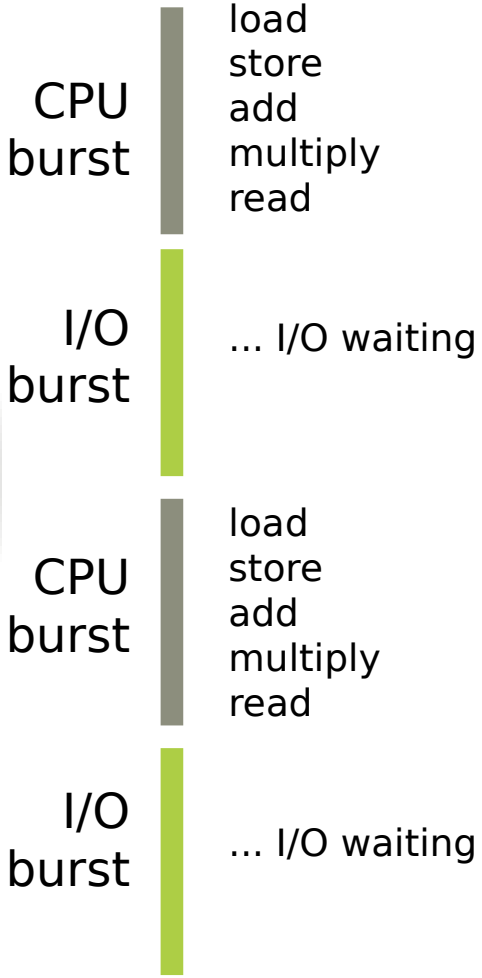
- **Dispatcher**
  - Low-level mechanism
  - Responsibility:context switch
- **Scheduler**
  - High-level policy
  - Responsibility:deciding which process to run

When to schedule

- When does the process makes decisions?
  When a process:
  - switches from running to waiting state
  - switches from running to ready state
  - switches from waiting to ready
  - terminates

Process classification

- Process execution can be essentially divided in to categories:
  - CPU activities (**CPU burst**)
  - I/O activities (**I/O burst**)



- Processes characterized by long CPU bursts are called ***CPU bound***
- Processes caracterized by long I/O burst are called ***I/O bound***
- Other classification exist. Processes may also be:
  - Interactive
  - Batch
  - Real time

Interactive processes

- Continuous user interaction input
- Spend most of their time waiting for I/O events (mouse and keyboard)
- Must be quickly resumed when an input is received
- Accettable average delays are in the range 50-150 ms with low variance
- Examples:
  - Shell
  - Text editor

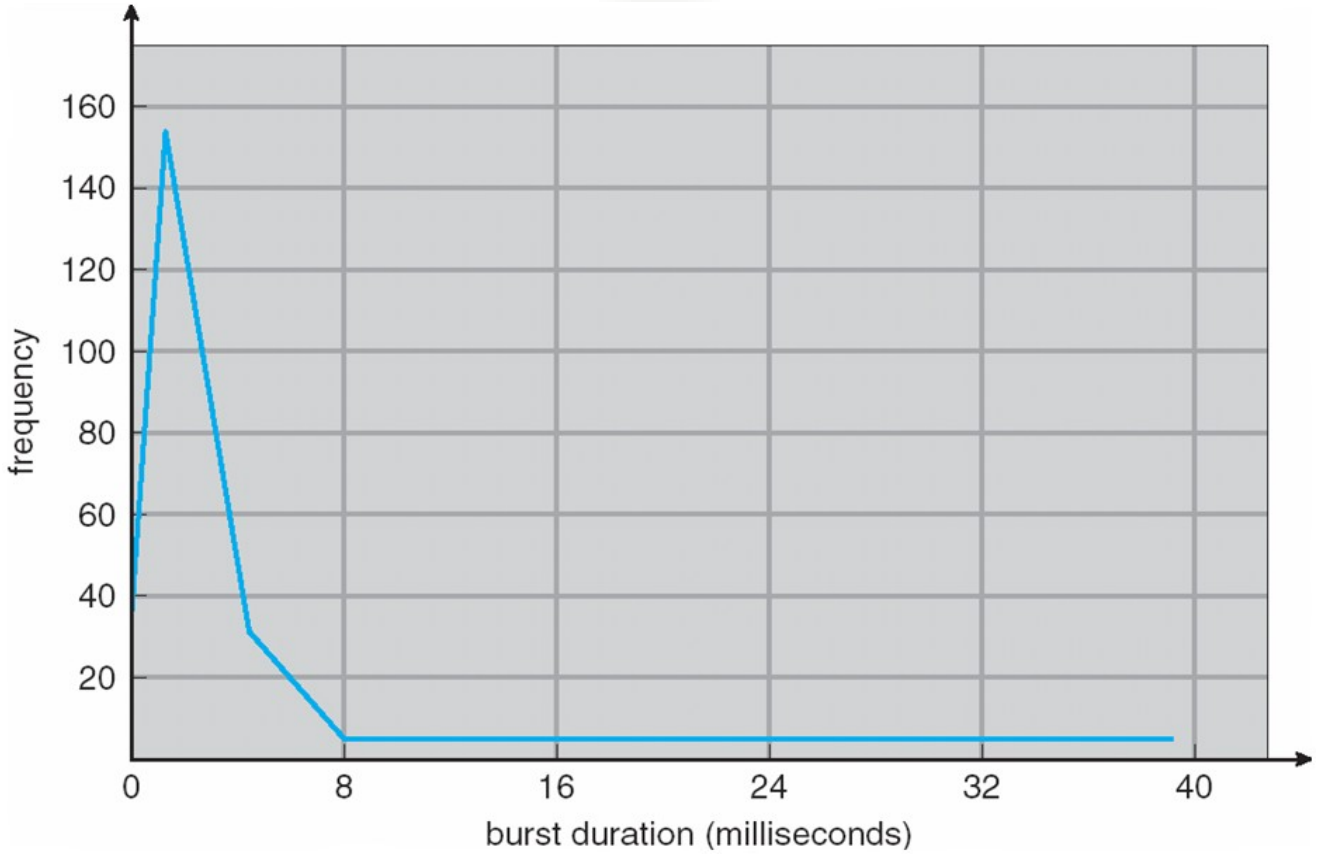
Batch processes

- No user interaction
- Most executed in background mode
- No short time responses
- Examples:
  - Compilers
  - Database search engines,
  - Scientific calculus
  - Payroll processing

Real time processes

- Strict time contraints:
  - A maximum response time must be guaranteed (very low variance)
- Examples:
  - Audio and video applications
  - Robot controllers
  - Sensor measurements (sampling)

CPU burst duration



Scheduler types

- *Non-preemptive* scheduler:
  - A process is executed until
    - terminates
    - Makes a syscall
  - Processes directly manage their scheduling (Windows 3.1, Mac OS 1,2, … 7)
- *preemptive* scheduler:
  - A context switch can always occur
  - The OS manages process scheduling (all modern schedulers)

Non-preemptive vs preemptive

- non-preemptive:
  - Scheduler does not require addional hardware mechanisms, like, for example, programmable timers (time sharing)
- preemptive:
  - (much) better CPU utilization

Scheduling performance criteria

- **CPU utilization:**
  - Percentage of time that CPU is busy
  - To maximize
- **throughput**
  - Number of processes that complete their execution per time unit (it depends on the duration of the processes involved)
  - To maximize
- **turnaround**
  - amount of time the OS takes to execute a process
  - To minimize

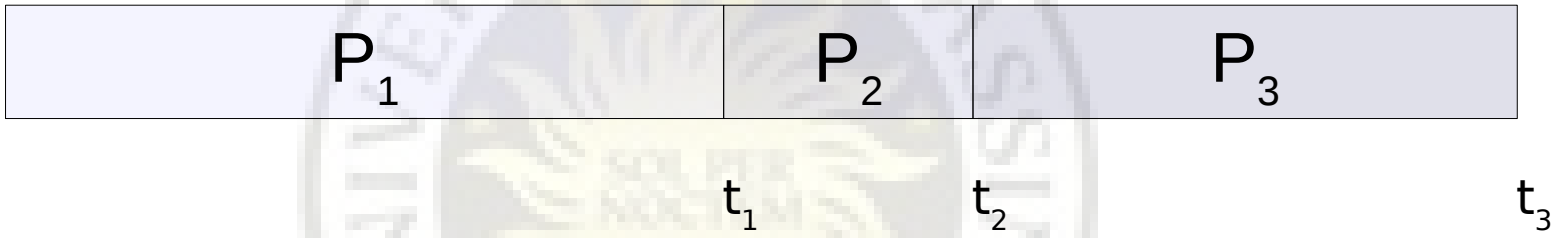
- **waiting time**
  - amount of time the process is waiting in ready queue
  - To mimimize
- **response time**
  - amount of time that the system takes to produce the first response
  - Crucial for interactive processes
  - To minimize
- **Fairness**
  - give each process (or user) same percentage of CPU

Scheduling algorithms

- First Come, First Served
- Shortest-Job First
  - Shortest-Next-CPU-Burst First
  - Shortest-Remaining-Time-First
- Round-Robin

Gantt charts

- Gantt charts are used to represent process schedule:



- In this example the resource (e.g. CPU) is used by process P_1 from (time) 0 to t_1, then it is assigned to P_2 until t_2 and then to P_3 until t_3

First Come, First Served (FCFS)

- The simplest CPU scheduling algorithm:
  - First job that requests the CPU gets the CPU
- Non-preemptive
- Implemented by means of a FIFO queue (list of ready processes)

Advantages & disavantages

- **Advantages**:
  - fair
  - Simple

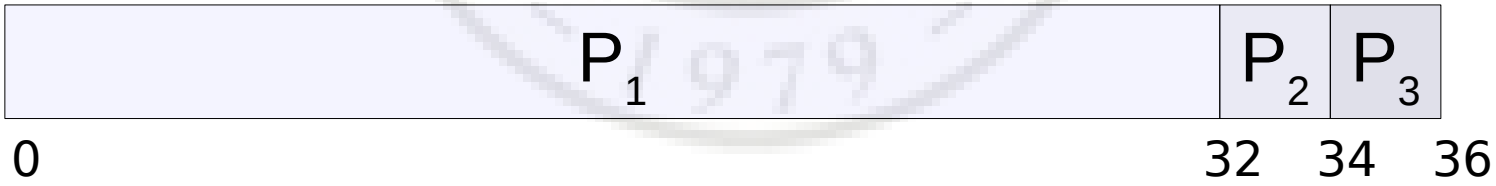
- **disavantages**
  - waiting time depends on arrival order
  - Convoy effect: short processes (I/O bounded) stuck waiting for long process (CPU bounded)

Example

- Arrival order: P_1, P_2, P_3
  - CPU-burst time length (ms): 32, 2, 2
  - average turnaround time:

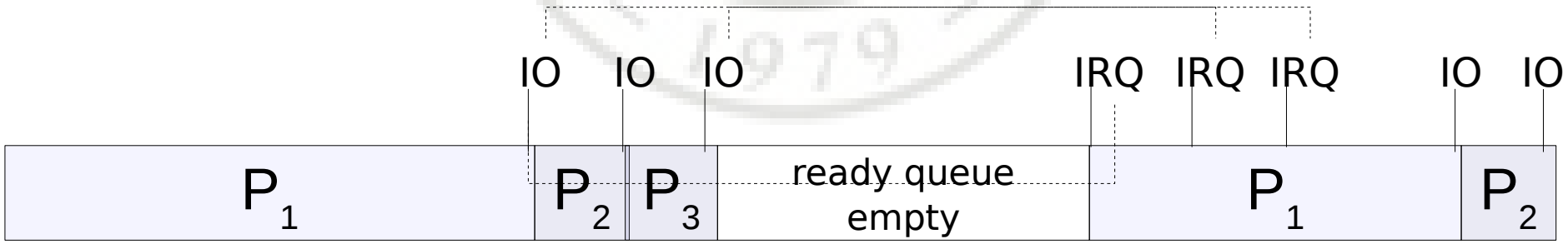
    (32+34+36)/3 = 34 ms
  - Average wating time:

    (0+32+34)/3 = 22 ms



Convoy effect

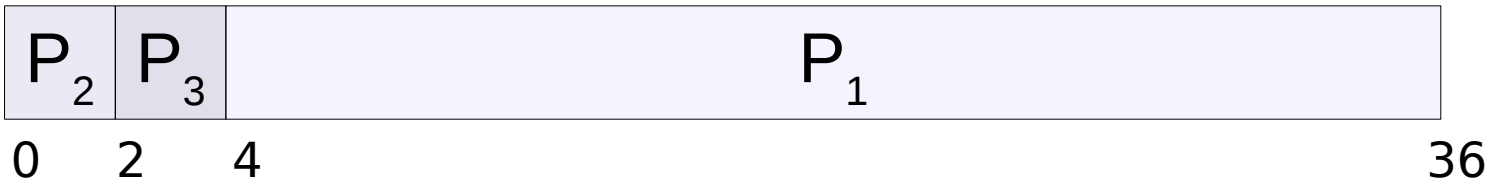
- Suppose this scenario:
  - a CPU bound process
  - some I/O bound processes
  - I/O bound processes “are queued” back to the CPU bound process
  - Sometimes the ready queue may be empty: *Convoy effect*



Shortest Job First (SJF)

- Assigns the CPU to the process with the shortest burst time (Shortest Next CPU Burst First).
- FCFS if processes have same/similar burst times
- Nonpreemptive

- **Example**
  - Average turnaround time: (0+2+4+36)/3 = 7 ms
  - Average waiting time: (0+2+4)/3 = 2 ms



Advantages & disavantages

- **Adavantages**
  - **Optimal** average waiting time
- **disavantages**
  - Not implementable: you can't exactly predict next burst times!
  - Possible starvation for long processes
- An approximation is possible:
  - Past burst time lengths are known

Shortest remaining time first

- If a new process arrives with a shorter CPU burst than the remaining for the current running process, schedule the new process
- SJF with preemption
- Advantage: reduces average waiting time

Priority scheduling

- A priority is associated with each process
- The scheduler chooses the highest priority ready process
- Priorities can be
  - Computed by the OS, using some (measurable) quantities
    - Example: SJF
  - Set by users

- **Static priority**
  - Fixed when the process starts. Does not change
  - Problem: (possible) starvation of low priority processes
- **Dynamic priority**
  - Priority may vary during the process life
  - Dinamic priorities avoid starvation

Aging priority

- The priority of the waiting process in the ready queue is gradually incremented
- No starvation:
  - No process waits indefinitely because, sooner or later, it gets the maximum priority

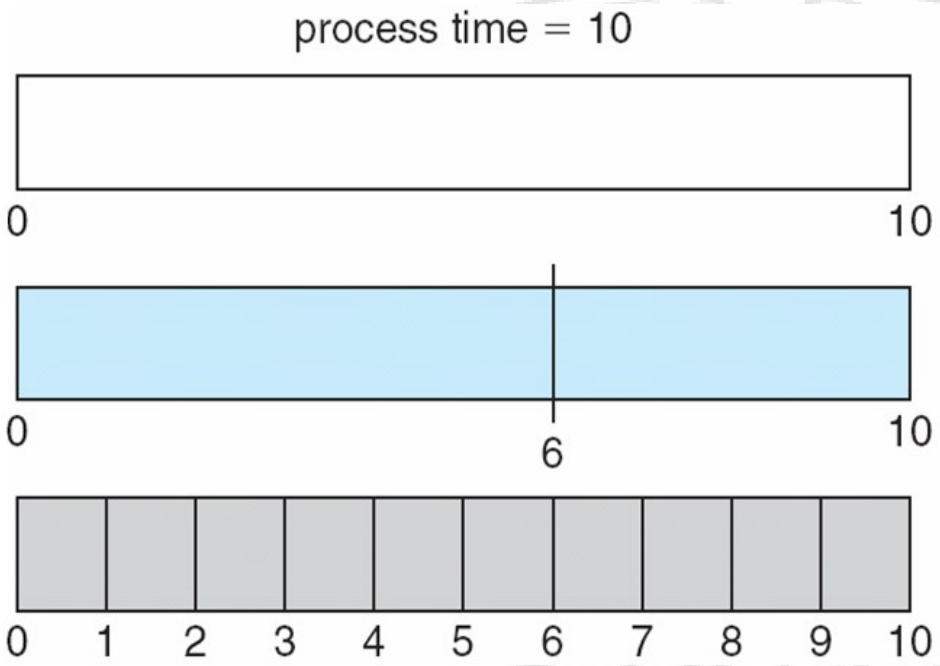
Round-Robin scheduling

- Each process runs for a time slice (or quantum)
- At the end of the time slice, if still running, the process is preempted and moved to the tail of the ready queue (FIFO)
- **Preemptive**
- With N ready processes and the time slice is q, the maximum waiting time is (N-1)q

Time slice

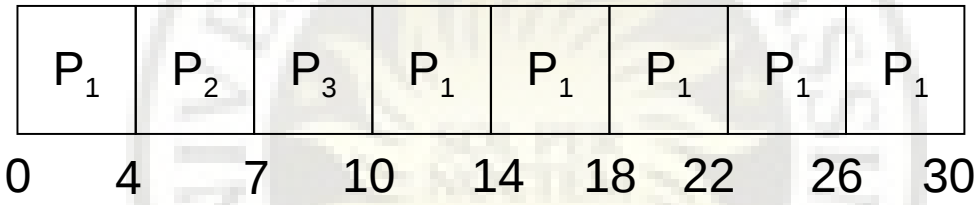
- How to determine the time slice?
- Short time slice:
  - High overhead, because of too frequent context switches
- Long time slice
  - Tends to become FCFS
  - As N increases, too high average wating times
- solution...

empirical!



quantum	context switches
12	0
6	1
1	9

Process	Burst Time
P_1	24
P_2	3
P_3	3



On average, the turnaround time of RR is greater than that of SJF, but the response time is better (lower)

Advantages and disadvantages

- **Advantages**
  - Low response time, good interactivity
  - Fair allocation of CPU across processes
  - Low average waiting time when process lengths vary widely

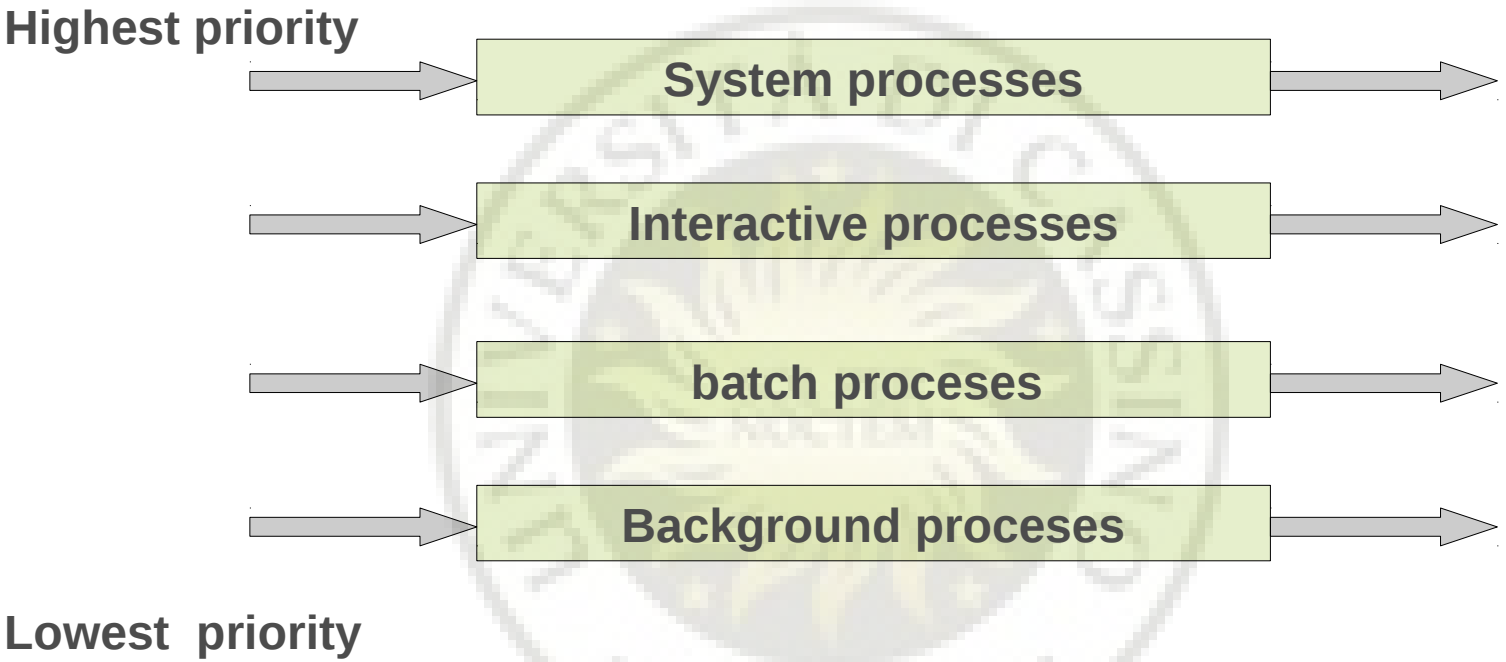
- **Disadvantages**
  - Poor average waiting time when processes have similar lengths (even worse than FCFS)
  - Performance depends on **length of time slice:**
    - Too high: degenerate to FCFS
    - Too low too many context switches, costly

Priority classes

- Processes can be distinguished according to their scheduling needs
- **Example**:
  - foreground and background
  - Two ready queues:
    - Foreground: RR
    - Background: FCFS

- A scheduling policy among the queues must be adopted
- There are many choises:
  - Fixed priority: higher priority queue first (starvation)
  - Round robin:
    - Example 80% foreground queue and 20% background queue

Example

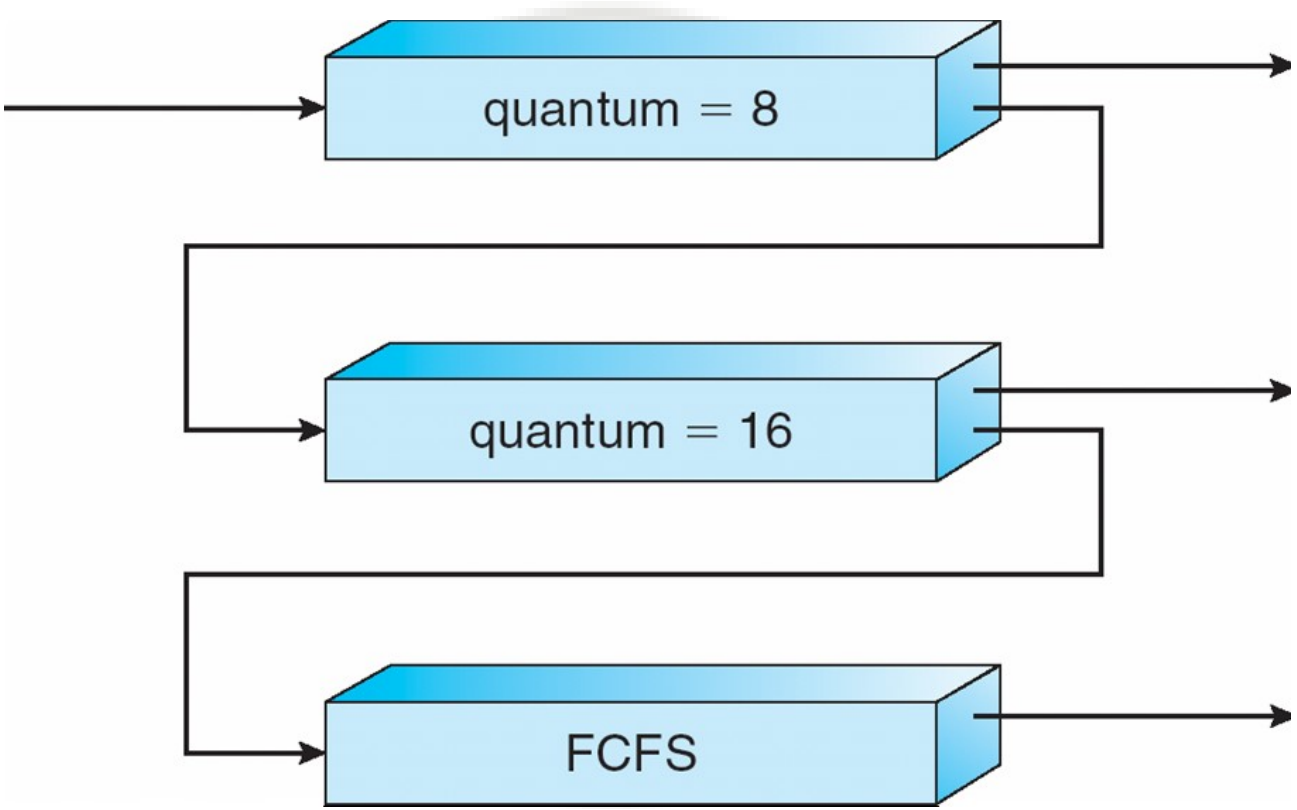


NOTE

Suitable when processes can be easily classified

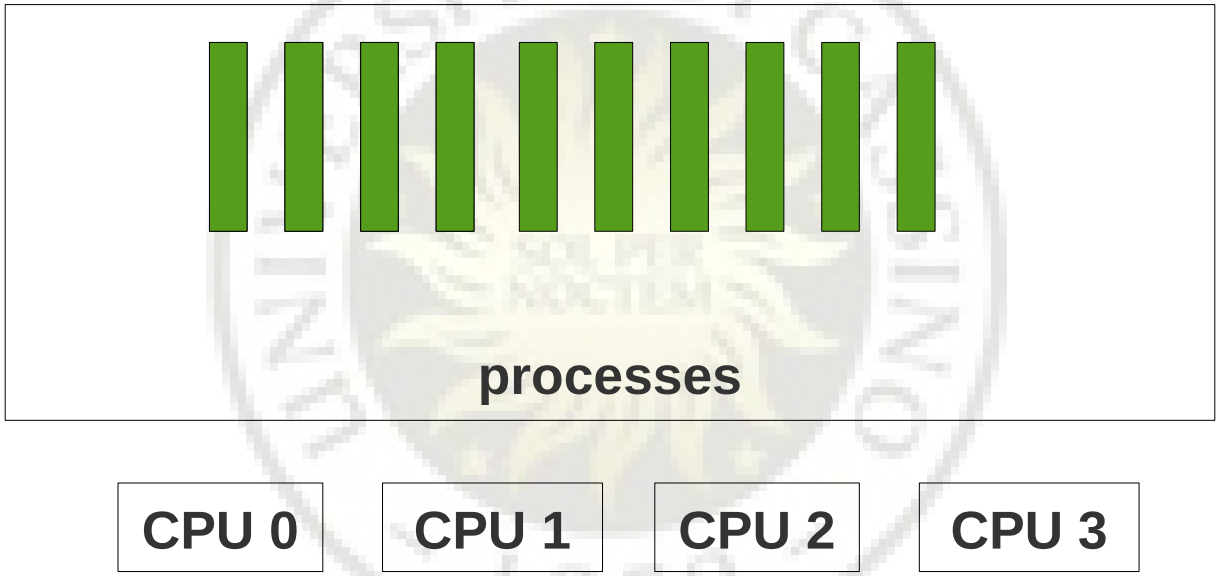
Multilevel feedback scheduling

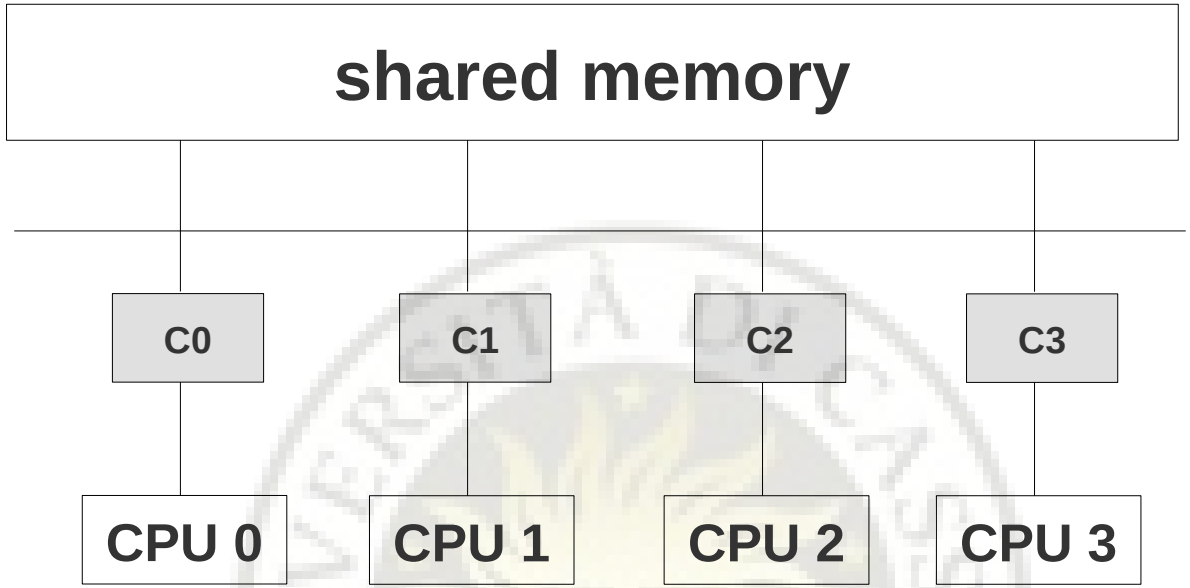
- Processes can move among the various queues
- Several parameters:
  - Number of queues
  - Algorithm for each queue
  - How to move processes among the queues?
  - which queue for new processes?


quantum = 8
quantum = 16
FCFS

Multiprocessor scheduling

- Shared-memory Multiprocessor

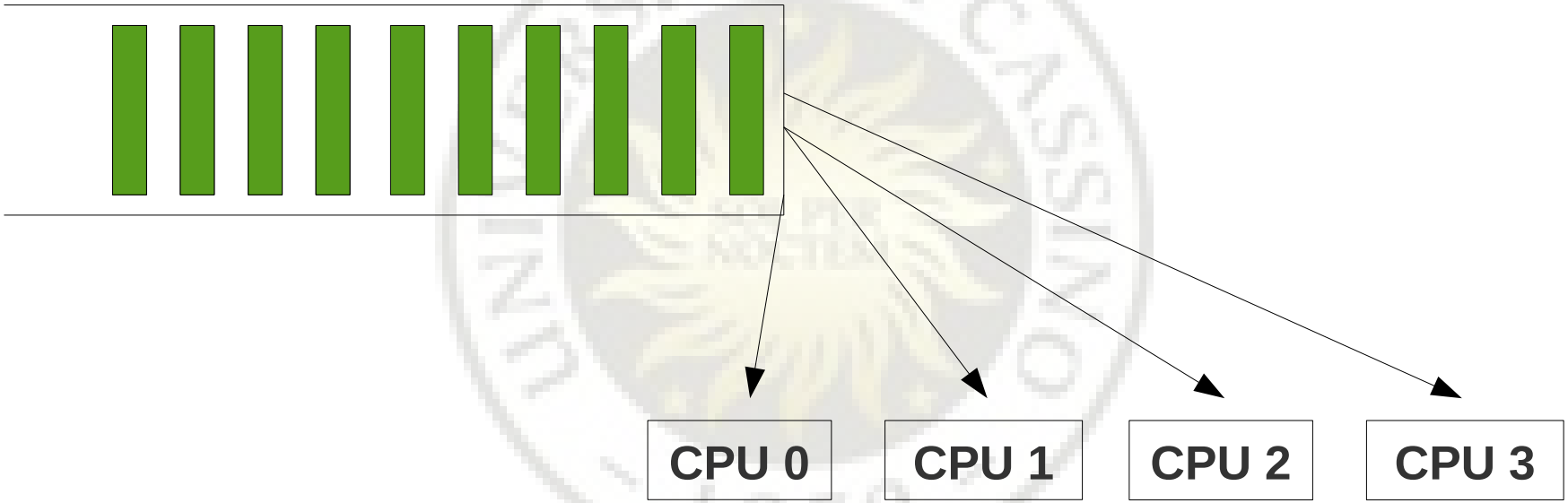




- Architecture:
  - Small number of CPUs
  - Same access time to main memory
  - Private caches (C0, C1, C3, C4)

Global queue

- One ready queue shared across all CPUs

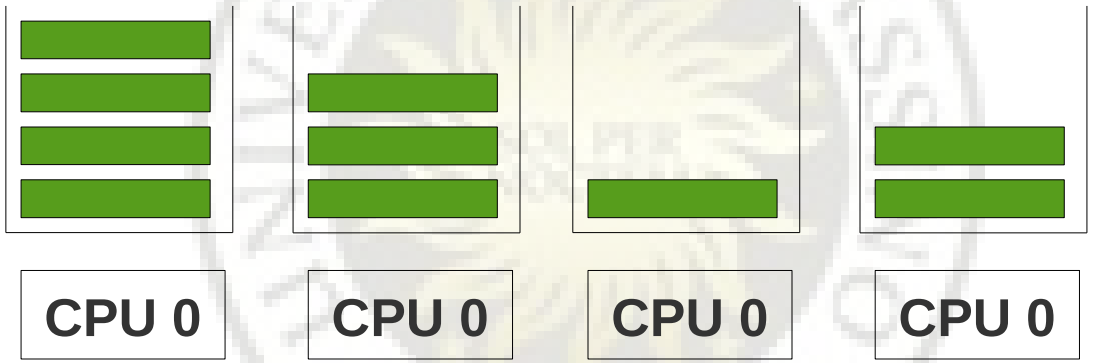


Advantages and disavantages

- Advantages
  - Good CPU utilization
  - Fair to all processes
- Disavantages
  - Not scalable (contention for the lock of the global queue)
  - Poor cache locality
- Linux 2.4

per-CPU queue

- Static partition of processes to CPUs

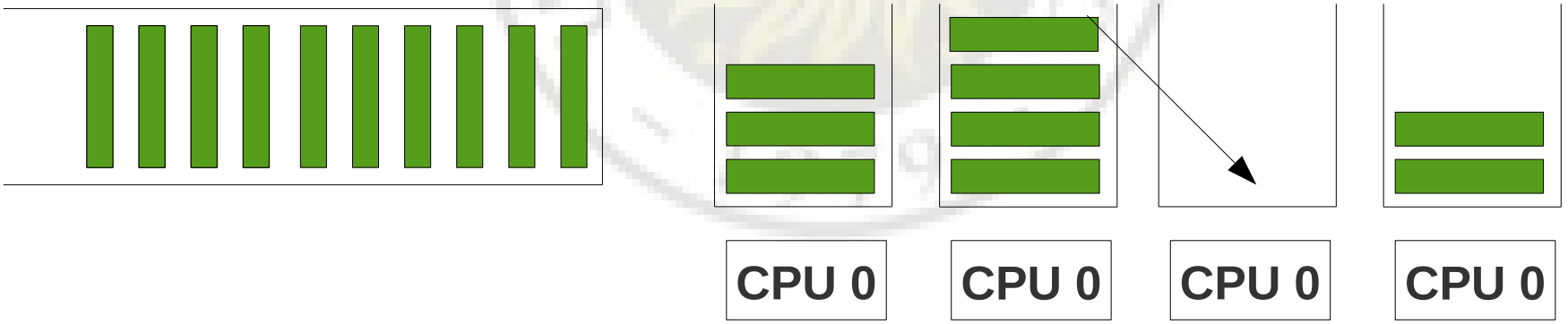


Advantages and disavantages

- **Advantages**
  - Easy to implement
  - Scalable (no contention on ready queue)
  - Better cache locality
- **Disadvantages**
  - Load-imbalance (some CPUs have more processes)
  - Unfair to processes and lower CPU utilization

Hybrid approach

- Use both global and per-CPU queues
- Balance processes across queues
- all “modern” OS



Processor affinity

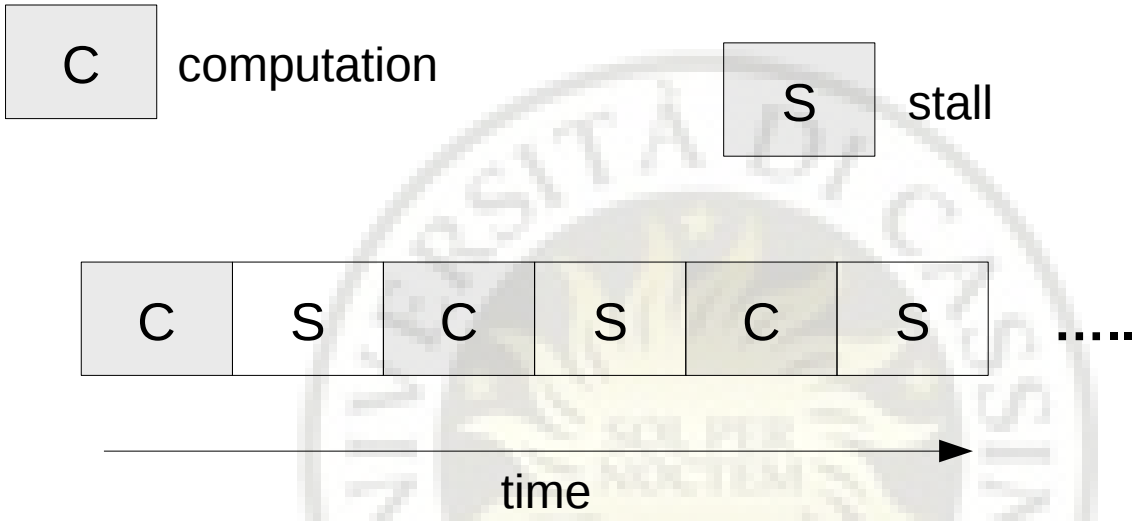
- Cach memory may strongly affect the performance
- Processor affinity:
  - Add process to a CPU's queue if recently run on the CPU
    - Cache state may still present
- Processor affinity can be:
  - Weak: not guaranteed (Solaris)
  - Strong (Solaris, linux 2.6)

Load balancing

- Two approaches:
  - *push migration*: an OS process periodically checks and manage load balancing
  - Spontaneous migration: inactive CPUs execute ready processes of other CPUs
- Linux:
  - every 200 ms, an OS process checks load balancing
  - Every time a queue is empty

NOTE
Load balancing may conflict with processor affinity

CPU Stall

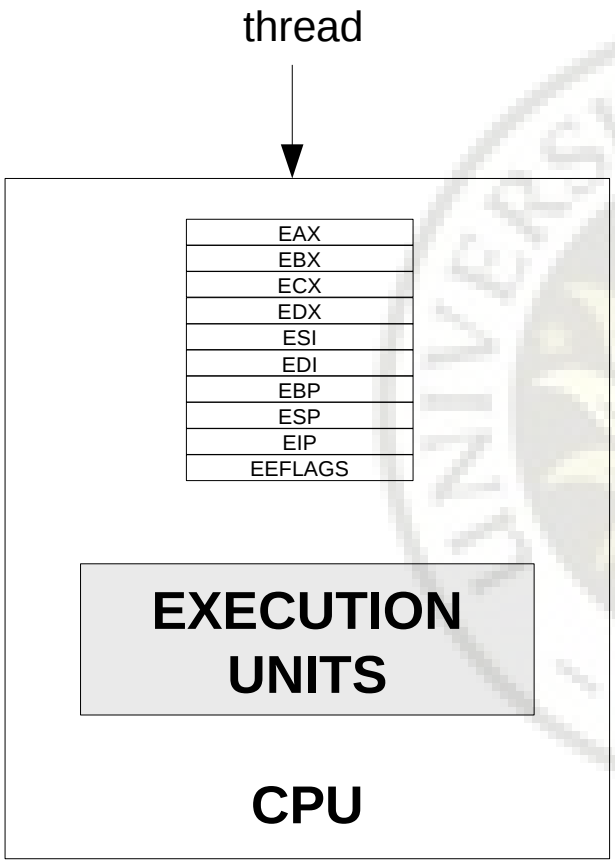


- Causes of stall:
  - Miss cache
  - branch misprediction (pipeline)
  - data dependency

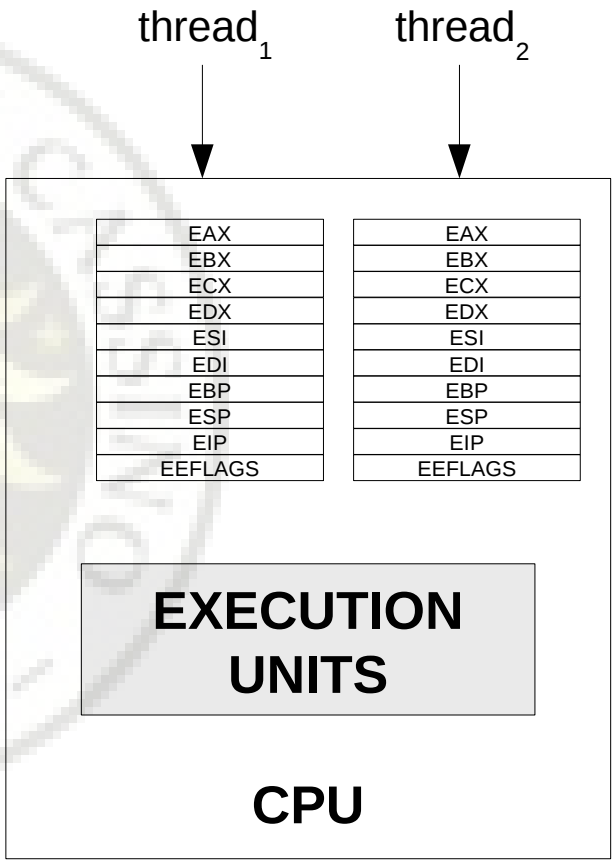
HyperThreading (INTEL)

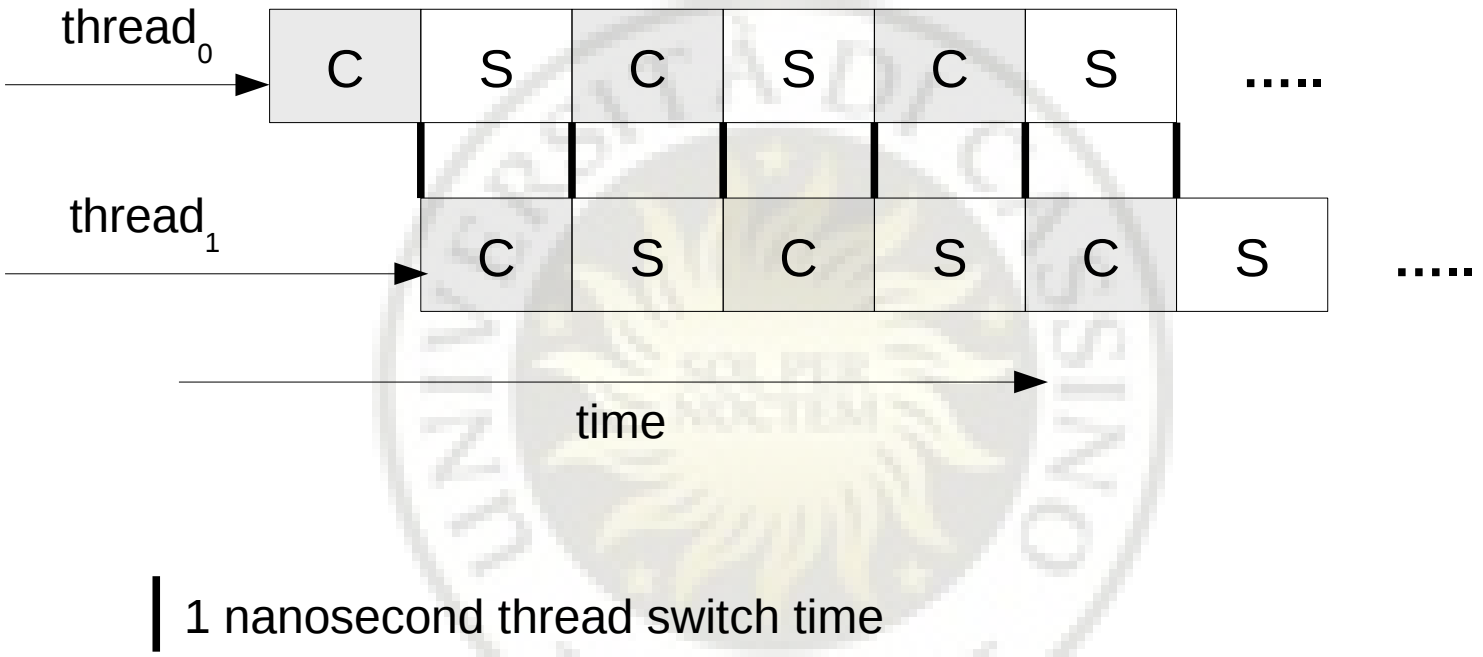
- Duplicated registers (Architectural state), but not the computing resources (ALU, FPU).
- A single core can stores the states of two trheads
- Thread-switching time in one nanosecond
- Not true parallelism:
  - each thread can actually execute only when the other one is stalled, i.e. waiting for many clock data and/or instructions
- thread execution units appear to the OS as distinct cores

singlethreaded CPU



multithreaded CPU




thread0
C
S
C
S
C
S
.....
thread1
C
S
C
S
C
S
.....
time
1 nanosecond thread switch time

Solaris scheduler

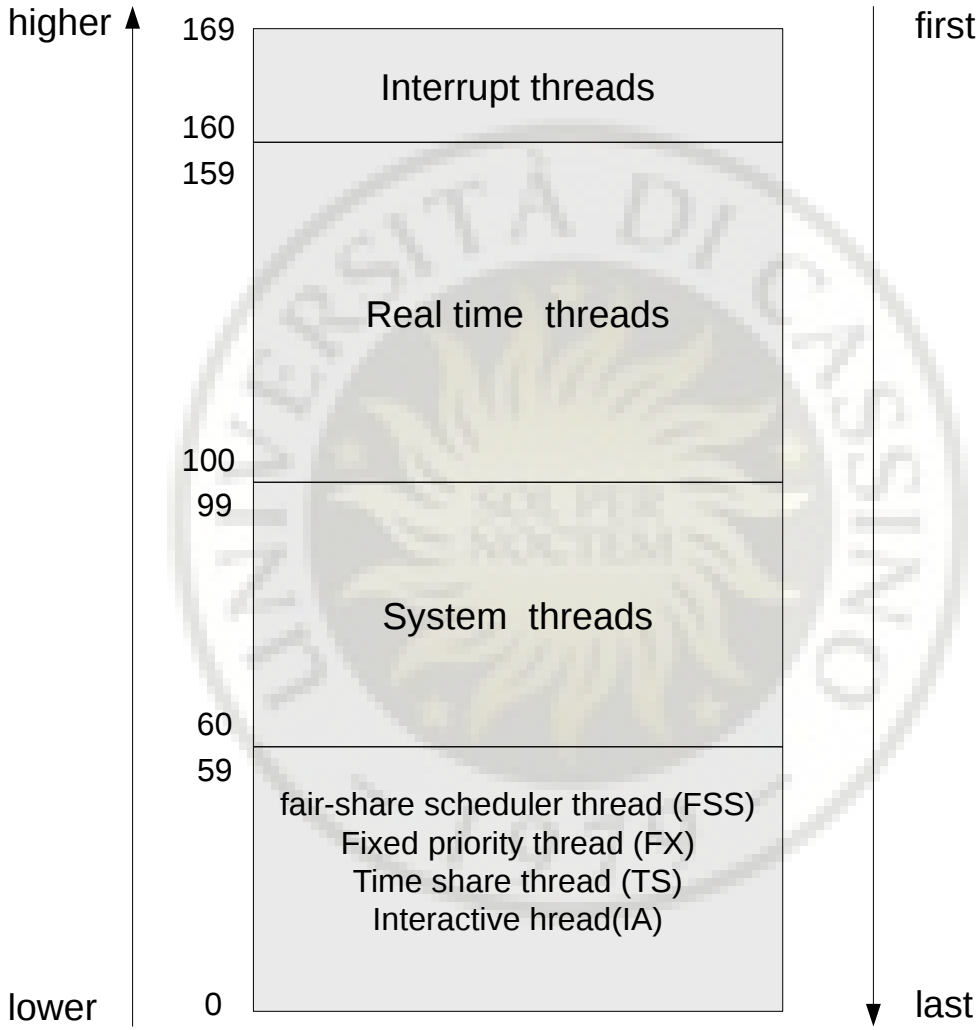
- Multilevel feedback scheduling
- Each class may use a different algorithm
- More processes in the same queue (i.e. same priority): Round-Robin (RR)

- In Solaris a process is executed until:
  - Makes a (blocking) syscall
  - Expires its quantum
  - Is preempted by a process with higher priority

Scheduling classes

- **Time share (TS)**: default class, dynamic priorities
- **Interactive (IA)**: dynamic priorities, improved version of the TS, for the windows manager.
- **Fixed-priority (FX)**: fixed priority threads
- **fair-share scheduler (FSS)**: fixed quotas instead of priorities
- **System (SYS)**: kernel threads, execute until terminate or block
- **Real Time (RT)**: fixed priority with fixed slice time

Priority table



Dispatcher table

- For the TS and IA classes the priorities are computed according this table:

priority	Time quantum	Time quantum expired	return from sleep
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Dispatcher table

- **Time quantum expired**: new priority assigned to the processes which consumed the assigned quantum time. New priority is lower (greater quantum)

- **Return from sleeping**: new priority assigned to the processes which blocked before consuming the assigned quantum time. The new priority is higher (smaller quantum)

Windows Scheduler

- Preemptive
- Feedback priorities
- An user process executes until:
  - a) terminates;
  - b) switches to wait state
  - c) uses up the time quantum
  - d) it is preempted

- 32 priority levels:
  - Real-time processes: from 16 to 31
  - Other processes: from 1 to 15 (0 is reserved to the idle process)
- New processes is assigned priority 1.

- The CPU is assigned to the highest priority process
- If more processes have the same priority: RR.
- If the running process switches to the wait state before that its quantum is expired, its priority is incremented (max: 15)
- The increment depends on the event the process is wating for:
  - Keyboard events (interactive processes) cause a greater increment than disk events

Interactive processes

- Processes which expire their quantum get a priority decrement (never < 1).
- Interacive processes (mouse and keyboard) are favoured: they require very low response time
- The scheduler distinguishes among background processes and that in foreground (active window).
- When a process switches to foreground, its quantum is multiplied by 3,

Linux scheduling

- Three scheduling classes
  - ***Real-time FIFO*** (**SCHED_FIFO**)
    - High priorities
    - preemptable only by threads with higher priority
  - ***Real-time Round-Robin*** (**SCHED_RR**)
    - Medium priority
    - preemptable by higher priority threads or when the time quantum expires
  - ***Timesharing*** (**SCHED_OTHER**)
    - Low priority
    - normal processes

- **dynamic vs static processes**
  - Real time processes have static priority
  - timesharing processes have dynamic priority
- **Full-preemption: scheduler gets CPU control when:**
  - A process expires its quantum time
  - a new process is added to the ready queue, and its priority is higher than that of the current running process

Linux Scheduling 2.6

- Complexity *O(1)*
- The higher the priority, the longer the quantum time
- A queue for each CPU (or core)
- Queue loads are periodically checked, and eventually balanced
- Processor affinity: PCB contains a bitmask of the CPUs that can be used by the process

Processes are grouped in two separated queues :

- **Active queue**: processes that do not expire their quantum time (preempted or waiting for something)
- **Expired queue**: processes which expired their quantum time


active
array
priority
task lists
[0]
[1]
[140]
expired
array
priority
task lists
[0]
[1]
[140]

Queue state bitmap

- A bitmap is used to represent queue states:
  1 → the queue contains at least one process
- Active process search:
  - 140 queue → 5 integers
  - At most 5 checks to find the first not empty queue
  - Hardware instruction to find the first 1-bit
    - **`bsfl`** on Intel
  - CONSTANT complexity with respect to #processes (N): it only depends on the number of priority levels

- The sheduler chooses the highest priority process in the active queue,
- The assigned quantum time is:
  - A whole quantum time (depends on its priority), it the process expired its previous assigned quantum
  - The remaining of the quantum time previously assigned to it, before it went in waiting state
- Once a process expired its quantum, it is moved to the expired queue

- At the beginning, the expired queue is empty: no process expired its quantum
- If the active queue is empty: all ready processes received and used up the assigned quantum time (it is a sort of Round Robin)
- At this point the two queues are switched the active queue becomes the expired one, and viceversa
- Then scheduler restart assigning a quantum time to the highest priority process in the active queue

Normal process scheduling

$$\text{Quantum} = \begin{cases} (140 - \text{SP}) \times 20 & \text{if SP} < 120 \\ (140 - \text{SP}) \times 5 & \text{if SP} \geq 120 \end{cases}$$

NOTE
SP is the static priority

dynamic priorities

- Dinamic priorities are computed considering two quantities:
  - static priority
  - Average sleep time (AST): the average waiting time
- AST value:
  - It is incremented (up to a maximum) if the process is in the wait state
  - It is decremented at every time tick when the process is running

- A bonus (range [0,10]) proportional to the AST is assigned at every process
  - 5 is a neutral value
  - 10 causes an priority increment of 5
  - 0 causes a priority decrement of 5

  DP = *max*(100, *min*(SP- bonus+5, 139))

Interactive processes

- Once a expired its quantum, an interactive process (mouse or keyboard) is put at the end of its current queue:
  - It that queue does not contain more processes it gets the CPU back again
  -
- NOTE
  - More time the process executes, lower its AST value

Scheduler 2.6: problems

- switching rate depends on process priorities:
  - two processes with a priority of 120 switch every 100 ms
  - two processes with a priority of 139: switch every 5 ms
- Quantum time depends on priority, but:
  - Two processes with priority 120 e 119 have 95 and 100 ms quanta, respectively
  - Two processes with priority 139 e 138 have 10 and 15 ms, respectively!

Completely Fair Scheduler

- since 2007 (version 2.6.23) Linux kernel adopts, a new scheduler algorithm

  Completely Fair Scheduler (CFS)

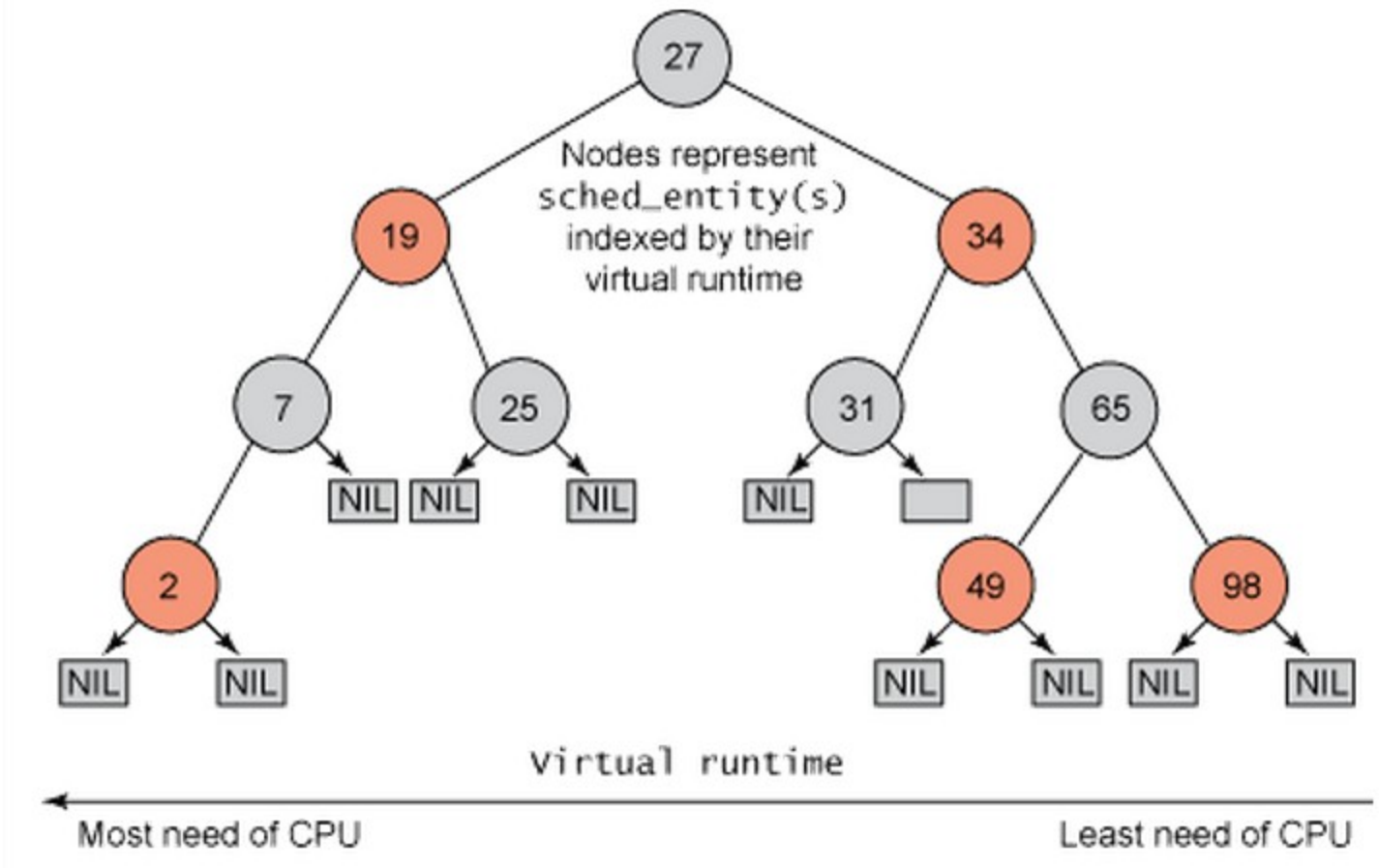
- Its main objective is fairness: it tries to equally distribute the CPU resource among all ready processes

Completely Fair Scheduler

- CFS Policy:
  - If there are N ready processes, then each process should have 1/N CPU time
- It stores the time assigned to each process
- It also considers I/O waiting times

Implementation

- The queue of the ready process is implemented through a binary red-black tree
- All tree operations (search, node delete, node insertion) have complexity *O(log(N))*
- Every tree node contains a value proportional to the assigned CPU time (virtual runtime)


27
Nodes represent
sched_entity(s)
indexed by their
virtual runtime
19
34
7
25
31
65
NIL
NIL
NIL
NIL
NIL
2
49
98
NIL
NIL
NIL
NIL
NIL
NIL
Virtual runtime
Most need of CPU
Least need of CPU

CFS and priority

- CFS does not use priorities directly
- Priorities are used to compute the virtual runtime (different form the actual CPU runtime)
- Priorities are represented by a decay factor, multiplied by the CPU runtime
  - The higher the priority, the lower the decay factor

Linux scheduling system calls

- `nice()`
  - Change the static priority of a conventional process
- `getpriority()`
  - Get the maximum static priority of a group of conventional processes
- `setpriority()`
  - Set the static priority of a group of conventional processes
- `sched_getscheduler()`
  - Get the scheduling policy of a process

- `sched_setscheduler()`
  - Sets the scheduling policy and the real-time priority of a process
- `sched_getparam()`
  - Gets the real-time priority of a process
- `sched_setparam()`
  - Set the real-time priority of a process
- `sched_yield()`
  - Relinquishes the processor voluntarily without blocking
- `sched_get_ priority_min()`
  - Gets the minimum real-time priority value for a policy

- `sched_get_ priority_max()`
  - Gets the maximum real-time priority value for a policy
- `sched_rr_get_interal()`
  - Gets the time quantum value for the Round Robin policy
- `sched_setaffinity()`
  - Sets the CPU affinity mask of a process
- `sched_getaffinity()`
  - Get the CPU affinity mask of a process

NOTE
For more detailed information, use the command man:
`$ man function_name`

Final remarks

- Scheduling is an interesting multi-objective problem
- Most of the modern OS are multi-platforms:
  - Desktop systems
  - Servers
  - embedded systems (many are real time)
  - Supercomputers
- They have very different work loads
- Quite different objectives:
  - Fairness among processes
  - Interactivity
  - Maximum throughput