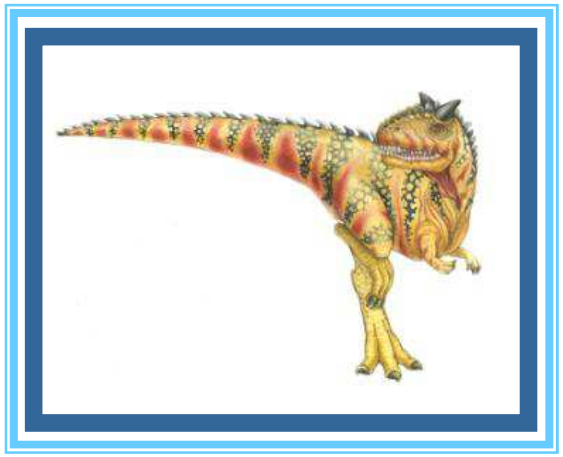
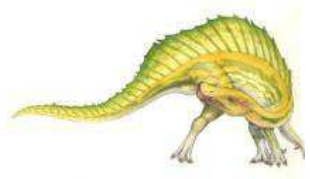
Topic 2.2: Processes Scheduling



- Chapter 5: Process Scheduling

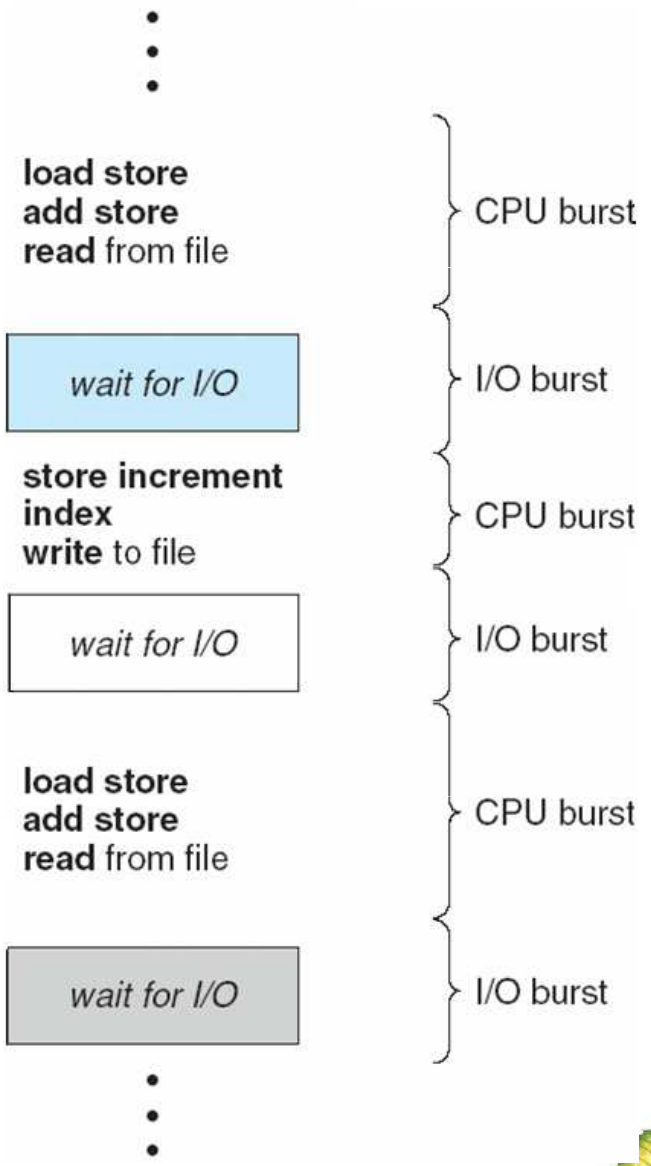
Contents

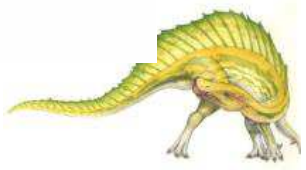
- **Processes**
  - Process concept
  - Process states
  - Process Control Block (PCB)
  - Operations on Processes
- **Threads**
  - Thread concept
  - Libraries to create threads
- **Operating System Examples**
- **CPU scheduling**
  - **Basic concepts**
  - Scheduling Algorithms
  - Examples



Assumption: CPU Bursts

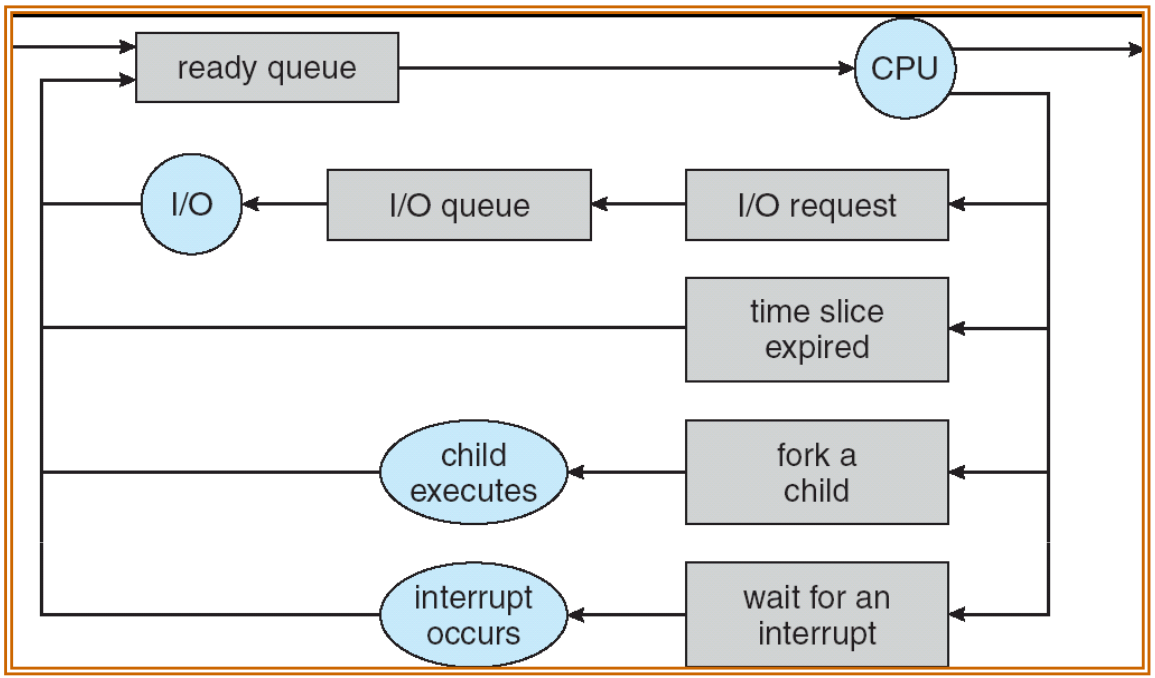
- Execution model: programs alternate between bursts of CPU and I/O
  - ***I/O-bound process*** – spends more time doing I/O than computations, many short CPU bursts.
  - ***CPU-bound process*** – spends more time doing computations; few very long CPU bursts.



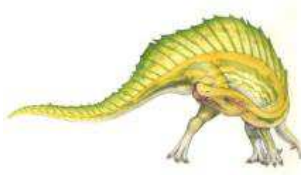


Scheduling issues

- Each scheduling decision is about which job to give to the CPU for use by its next CPU burst
- With timeslicing, process/thread may be forced to give up CPU before finishing current CPU burst

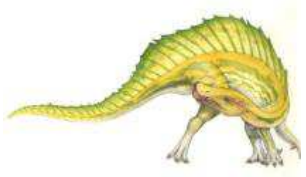


- Scheduling is central to OS design: CPU and almost all computer resources are scheduled before use



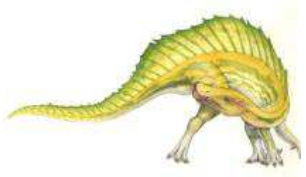
Schedulers

- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU
- **Medium-term scheduler** – selects which processes should be swapped from memory
- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue



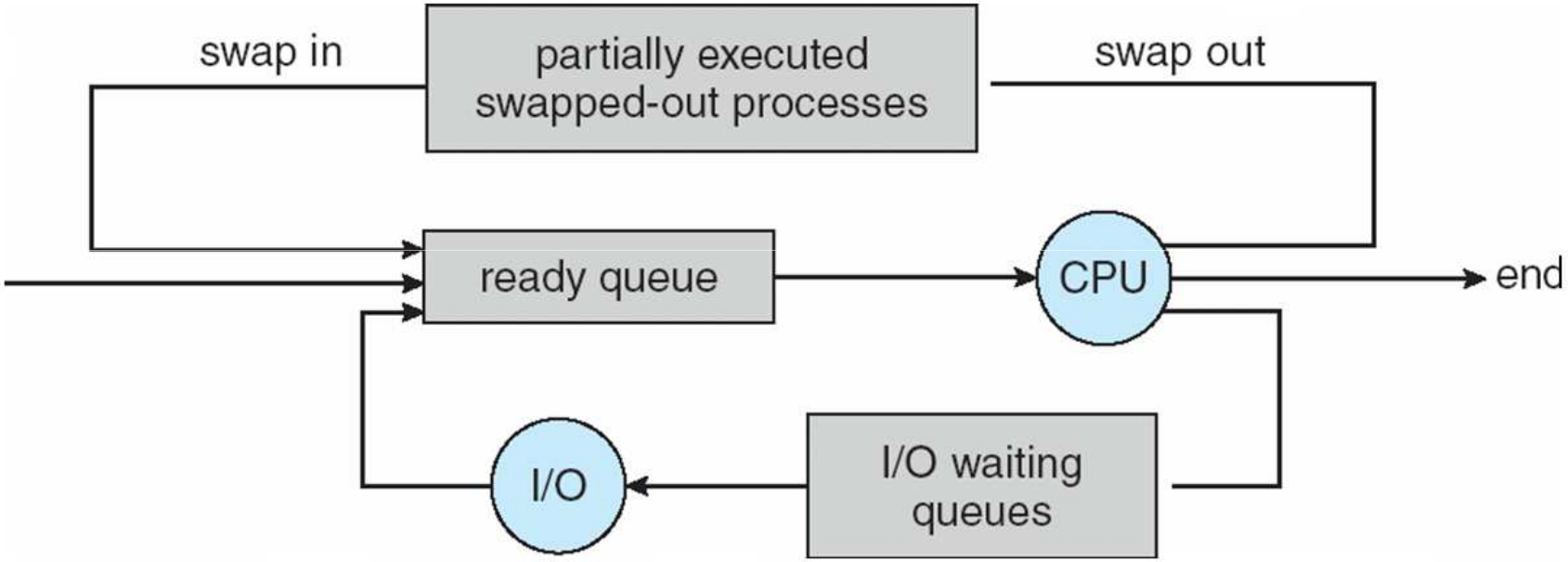
Schedulers (Cont)

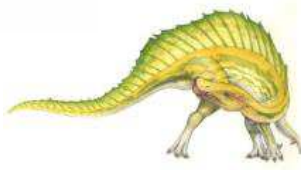
- Short-term scheduler is invoked very frequently (once every 100 milliseconds) ⇒ must be fast or else waste too much time scheduling and not executing
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (may be slow).
  - The long-term scheduler controls the ***degree of multiprogramming - the number of processes in memory.***
- Long-term scheduler should select a good process mix of processes to better the system performance



Addition of Medium Term Scheduling

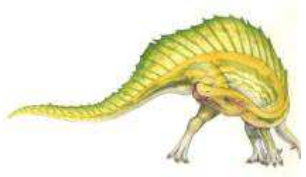
- Time sharing OSs may introduce a medium term scheduler
- Removes processes from memory (and thus CPU contention) to reduce the degree of multiprogramming – ***swapping***
- Swapping may be needed to improve the process mix or to free up memory if it has become overcommitted





Scheduling Criteria

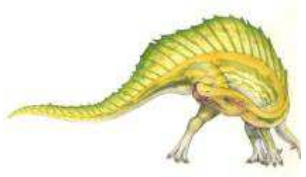
- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time (= Completion Time) (Tr)** – amount of time to execute a particular process
  - waiting to get into memory + waiting in the ready queue + executing on the CPU + I/O
- **Waiting time (Te)** – amount of time a process has been waiting in the ready queue
- **Response time (Ta)**– amount of time it takes from when a request was submitted until the first response is produced, **not** output (for time-sharing environment)



Scheduling Algorithm Optimization Criteria

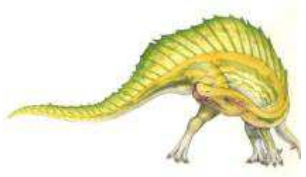
- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

- In most cases we optimize the average measure
- In some circumstances we want to optimize the min or max values rather than the average – e.g., minimize the max response time so all users get good service



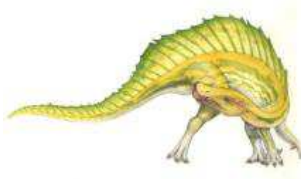
Different environments = different goals

- Scheduling policy
  - Strategies and decisions used to design the scheduling in order to get the goals
- Batch systems
  - Capacity of processing/performance
  - Turnaround time
  - CPU use
- Time-Sharing Systems
  - Response times (variance)
  - Complexity/response time proportion
- Real time Systems
  - Meet deadline service
  - Predictability or consistency



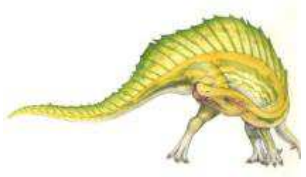
Contents

- **Processes**
  - Process concept
  - Process states
  - Process Control Block (PCB)
  - Operations on Processes
- **Threads**
  - Thread concept
  - Libraries to create threads
- **Operating System Examples**
- **CPU scheduling**
  - Basic concepts
  - **Scheduling Algorithms**
  - Examples



CPU Scheduler

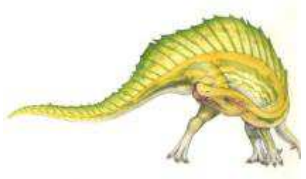
- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive:**
  - Once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or switching to a waiting state
- All other scheduling is **preemptive**
  - Running process is preempted from CPU
    - Another process (with higher priority) arrives to the ready queue
    - Every so often interrupt happens to launch the scheduler



Scheduling Algorithms

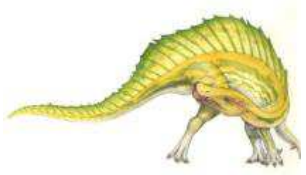
Sched. \ Preem.	Non Preempt.	Preempt.
Arrival Time	**FCFS**	**Round Robin**
ExecutionTime	**SJF**	**SRTF**
Priority	**Priority**	**Preemption Priority**

- Multilevel feedback scheduling



First-Come, First-Served (FCFS) Scheduling

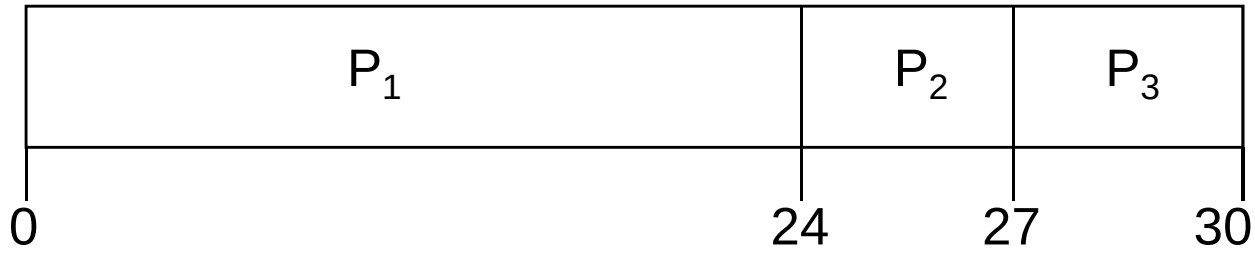
- **FCFS** is non-preemptive
  - Process leaves CPU when finishes or switches to waiting state
  - **FCFS is simple (+)**
  - Lowers CPU and device utilization
  - High Te and Tr
- Not good for time sharing systems where each user needs to get a share of the CPU at regular intervals
- ***Convoy effect*** short process (I/O bound) wait for one long CPU-bound process to complete a CPU burst before they get a turn



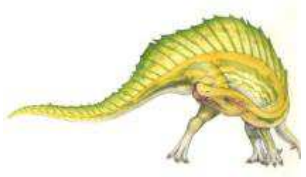
FCFS Scheduling (Cont)

Process	Burst Time
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:



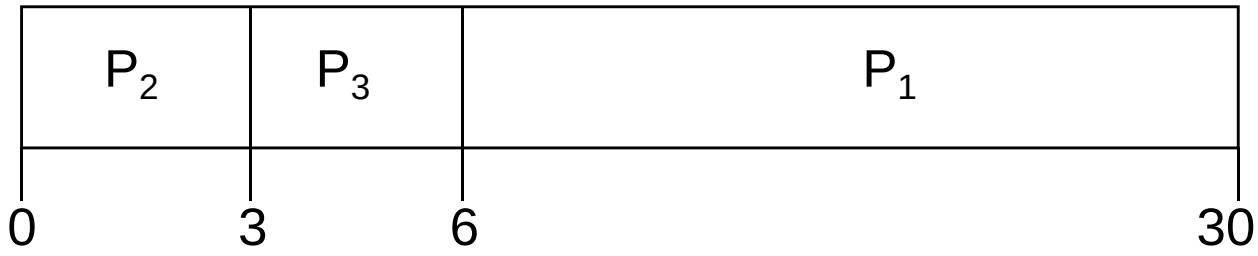
  - Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
  - Average waiting time (Te): (0 + 24 + 27)/3 = 17
  - Average completion time (Tr): (24 + 27 + 30)/3 = 27
- *Convoy effect:* short process behind long process



FCFS Scheduling (Cont)

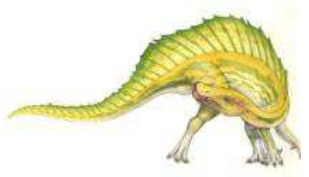
Suppose that the processes arrive in the order: P_2 , P_3 , P_1

- The Gantt chart for the schedule is:



  - Waiting time for $P_1 = 6; P_2 = 0; P_3 = 3$
  - Average waiting time (Te): (6 + 0 + 3)/3 = 3
  - Average completion time (Tr): (3 + 6 + 30)/3 = 13

- In second case:
  - **Average waiting time is much better** (before it was 17)
  - **Average completion time is better** (before it was 27)



Scheduling Example

Process	Arrival time	Burst time	Priority
T0	0	50	4
T1	10	50	3
T2	20	20	1
T3	30	10	2

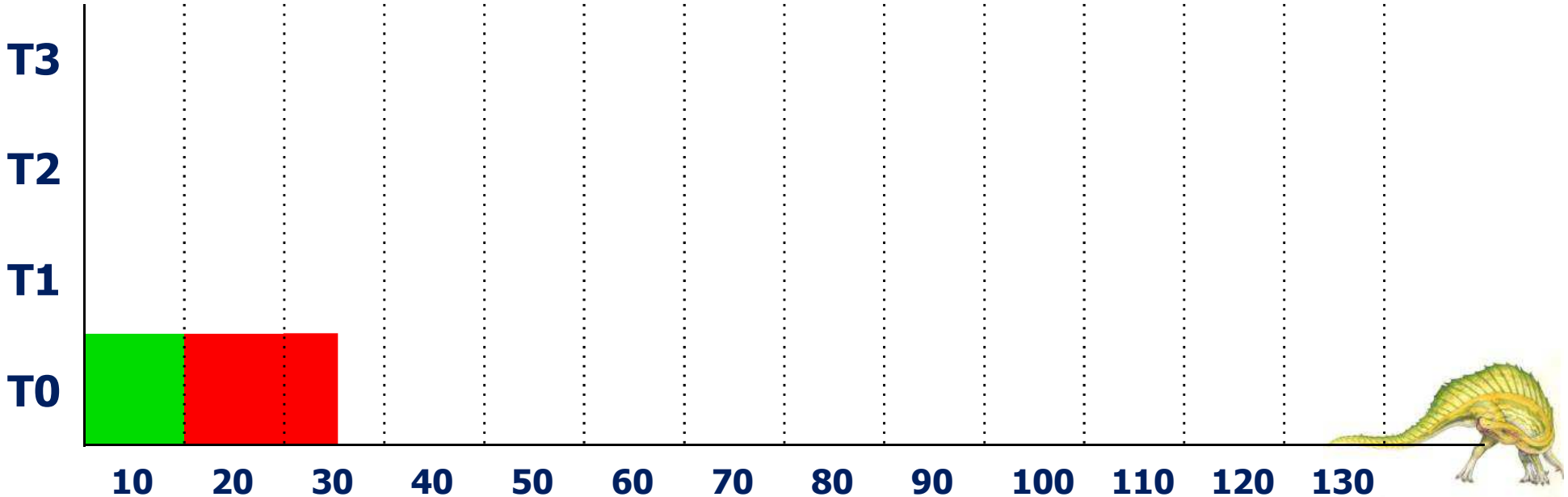
- Process T0 blocked at time 10 and during 15 time units

	Non preempt.	**Preemption**
Arrival time	**FCFS**	**Round Robin (RR) Q=10**
Execution time	**SJF**	**SRTF**
Priority	**Priorities**	**Preemtion priorities**

FCFS Example

Tiempo	Ejecucion	Listos	Bloqueado	
0	T0			LL T0
....				

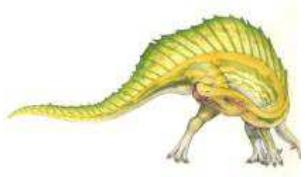
Tr = **Te =**



Shortest Job First (SJF)

- **Shortest Job First (SJF)**:
  - Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time (on a tie use FCFS)
  - ***nonpreemptive*** – once CPU given to the process it cannot be preempted until completes its CPU burst
  - **Good for timesharing systems (low Te)**
  - Bad for CPU-bound process

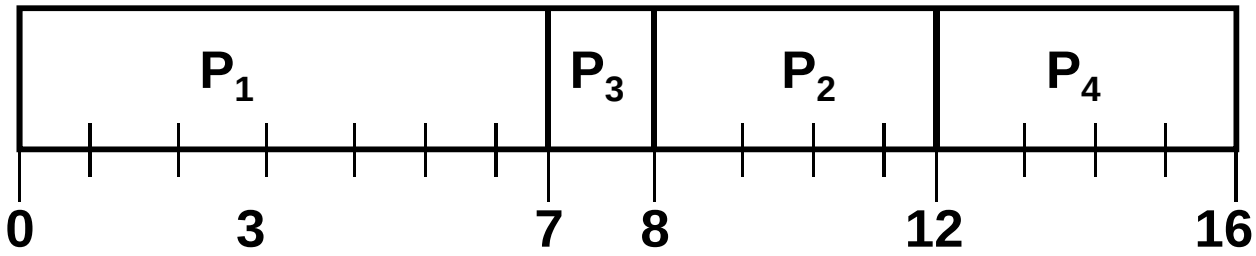
- **Starvation**
  - SJF can lead to starvation if many small jobs!
  - Large jobs never get to run



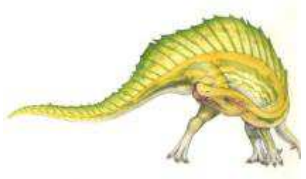
Example of SJF

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

- SJF scheduling chart



- Average waiting time (Te) : (0 + 6 + 3 + 7) / 4 = 4
- Average completion time (Tr): (7+10+4+11)/4 = 8



Scheduling Example

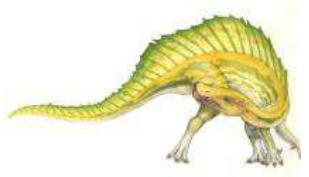
Process	Arrival time	Burst time	Priority
T0	0	50	4
T1	10	50	3
T2	20	20	1
T3	30	10	2

- Process T0 blocked at time 10 and during 15 time units

	Non preempt.	**Preemption**
Arrival time	**FCFS**	**Round Robin (RR) Q=10**
Execution time	**SJF**	**SRTF**
Priority	**Priorities**	**Preemtion priorities**

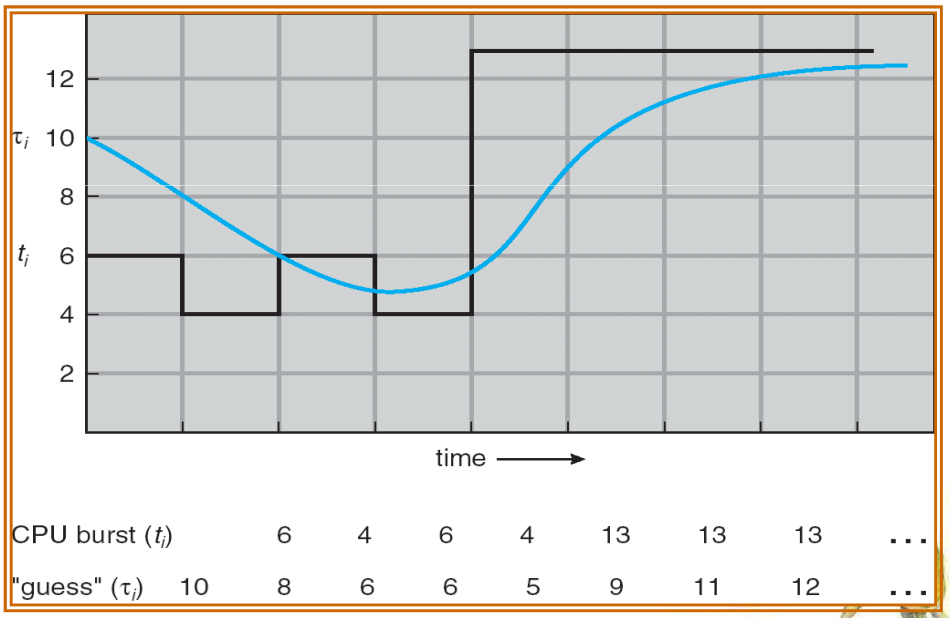
What if we Knew the Future?

- These can be applied either to a whole program or the current CPU burst of each program
  - Idea is to get short jobs out of the system
  - Big effect on short jobs, only small effect on long ones
  - Result is better average completion time
- The difficulty is knowing the length of the next CPU request
- **Adaptive**: Changing policy based on past behavior
  - Works because programs have predictable behavior
    - If program was I/O bound in past, likely in future
    - If computer behavior were random, wouldn't help



What if we Knew the Future?

- Use an estimator function on previous bursts:
  - Let t_{n-1}, t_{n-2}, t_{n-3}, etc. be previous CPU burst lengths.
    Estimate next burst $\tau_n = f(t_{n-1}, t_{n-2}, t_{n-3}, \ldots)$
  - **Exponential averaging: $\tau_n = \alpha t_{n-1} + (1-\alpha)\tau_{n-1}$ with $(0<\alpha\leq 1)$**
  - Second term shows a historical
  - The older the burst, less influence in the equation
  - If $\alpha=0$, last burst time not taken into account
  - If $\alpha=1$, historical not taken into
  - account
  - Usually, $\alpha=1/2$



α= 1/2

SJF

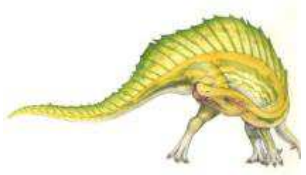
Estim. CPU burst n+1 | CPU burst n | Estim. CPU burst n

$$\tau_{n+1} = \alpha \cdot t_n + (1-\alpha) \cdot \tau_n \qquad 0 \le \alpha \le 1$$

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

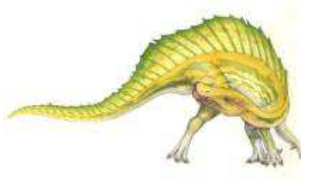
Process	CPU	CPU	CPU	CPU	CPU
T0	5	4	3	2	1
T1	1	2	3	4	5
T2	1	1	1	1	1

- I/O Burst always 1
- α = 0.5



Round Robin (RR)

- **Round Robin** Scheme
  - 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 (preemptive version of FCFS)
  - If there are *n* processes in the ready queue and the time quantum is *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.
- Performance
  - *q* large $\Rightarrow$ FCFS
  - *q* small $\Rightarrow$ Interleaved (*q* must be large with respect to context switch, otherwise overhead is too high)

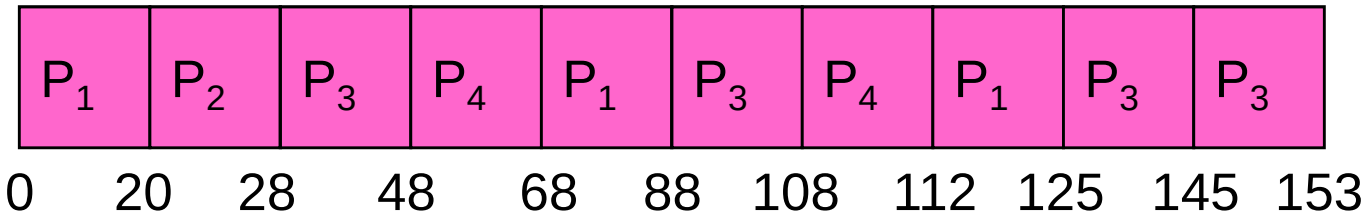


Example of RR with Time Quantum = 20

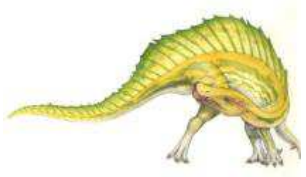
- Example:

Process	Burst Time
P_1	53
P_2	8
P_3	68
P_4	24

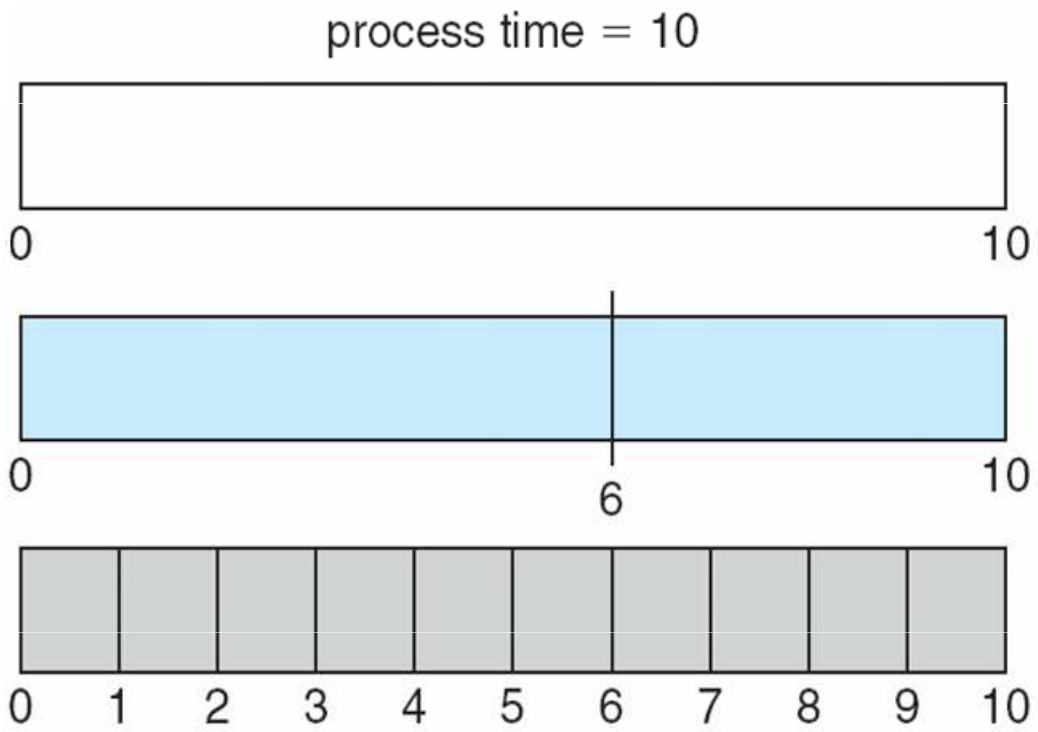
  - The Gantt chart is:



  - Waiting time for P_1=(68-20)+(112-88)=72
    P_2=(20-0)=20
    P_3=(28-0)+(88-48)+(125-108)=85
    P_4=(48-0)+(108-68)=88
  - Average waiting time = (72+20+85+88)/4=66¼
  - Average completion time = (125+28+153+112)/4 = 104½
- Thus, Round-Robin Pros and Cons:
  - **Better for short jobs, Fair (+)**
  - **Context-switching time adds up for long jobs (-)**

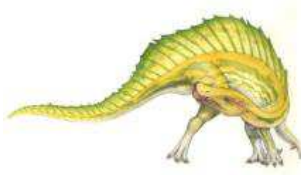


Time Quantum and Context Switch Time



quantum	context switches
12	0
6	1
1	9

A smaller time quantum increases context switches



Scheduling Example

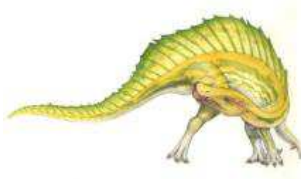
Process	Arrival time	Burst time	Priority
T0	0	50	4
T1	10	50	3
T2	20	20	1
T3	30	10	2

- Process T0 blocked at time 10 and during 15 time units

	Non preempt.	**Preemption**
Arrival time	**FCFS**	**Round Robin (RR) Q=10**
Execution time	**SJF**	**SRTF**
Priority	**Priorities**	**Preemtion priorities**

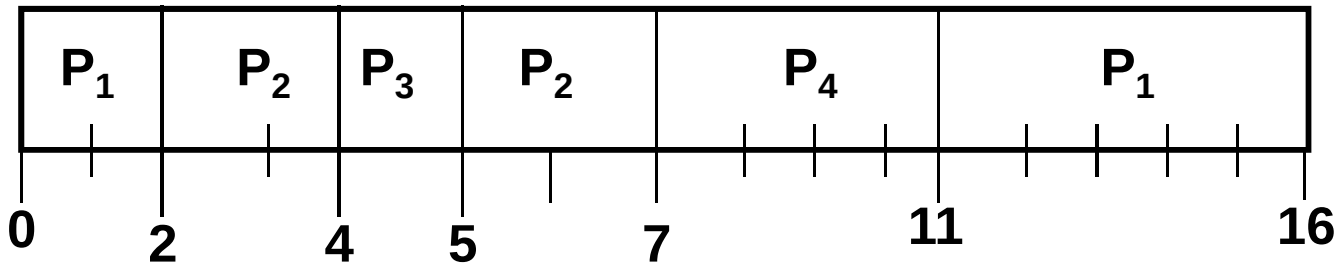
Shortest Remaining Time First (SRTF)

- **Shortest Remaining Time First (SRTF)/ Shortest Remaining Time Next (SRTN):**
  - Preemptive version of SJF: if a new process arrives with CPU burst length less than remaining time of current executing process, preempt
- **Starvation**
  - SRTF can lead to starvation if many small jobs!
  - Large jobs never get to run
- SRTF Pros & Cons
  - **Optimal (average response time) (+)**
  - **Hard to predict future (-)**
  - **Unfair (-)**

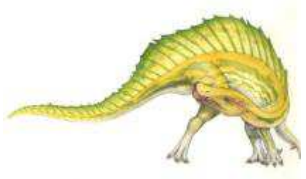


Example of SRTF

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



- Average waiting time = (9 + 1 + 0 +2)/4 = 3



Scheduling Example

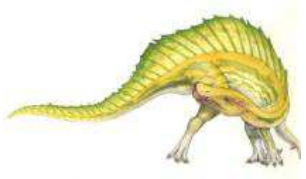
Process	Arrival time	Burst time	Priority
T0	0	50	4
T1	10	50	3
T2	20	20	1
T3	30	10	2

- Process T0 blocked at time 10 and during 15 time units

	Non preempt.	**Preemption**
Arrival time	**FCFS**	**Round Robin (RR) Q=10**
Execution time	**SJF**	**SRTF**
Priority	**Priorities**	**Preemtion priorities**

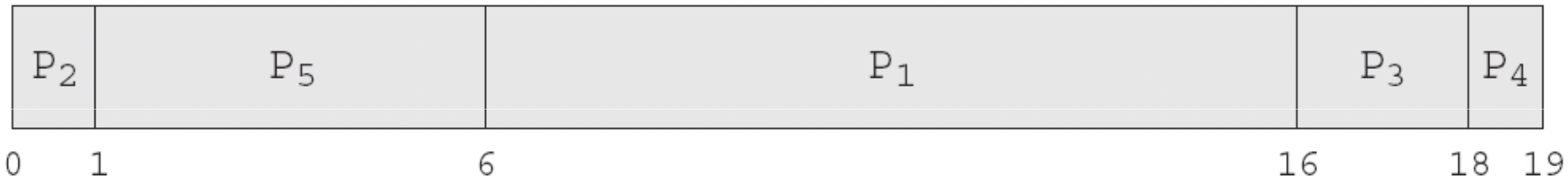
Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive (compares priority of process that has arrived at the ready queue with priority of currently running process)
  - Nonpreemptive (put at the head of the ready queue)
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

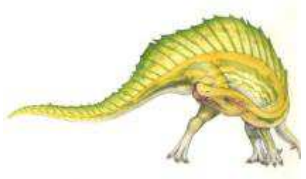


Priority Scheduling

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2



Average waiting time: 8.2



Scheduling Example

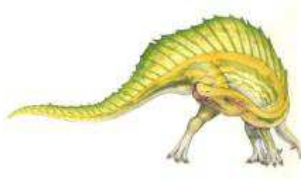
Process	Arrival time	Burst time	Priority
T0	0	50	4
T1	10	50	3
T2	20	20	1
T3	30	10	2

- Process T0 blocked at time 10 and during 15 time units

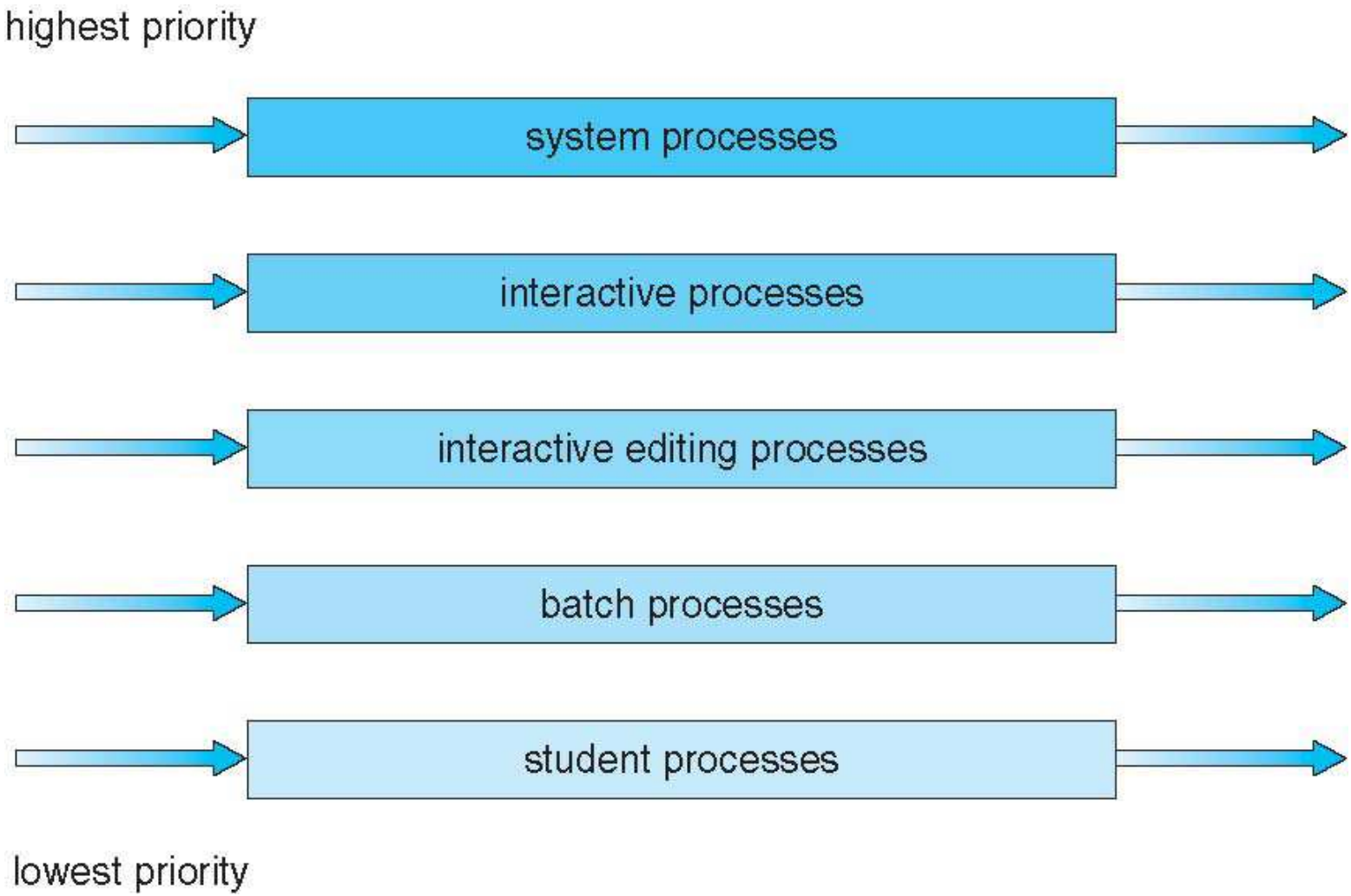
	Non preempt.	**Preemption**
Arrival time	**FCFS**	**Round Robin (RR) Q=10**
Execution time	**SJF**	**SRTF**
Priority	**Priorities**	**Preemtion priorities**

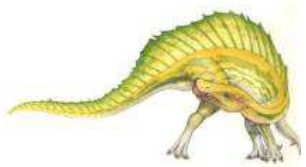
Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - **foreground** (interactive)
  - **background** (batch)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR, 20% to background in FCFS



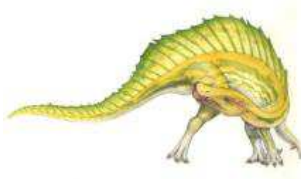
Multilevel Queue Scheduling



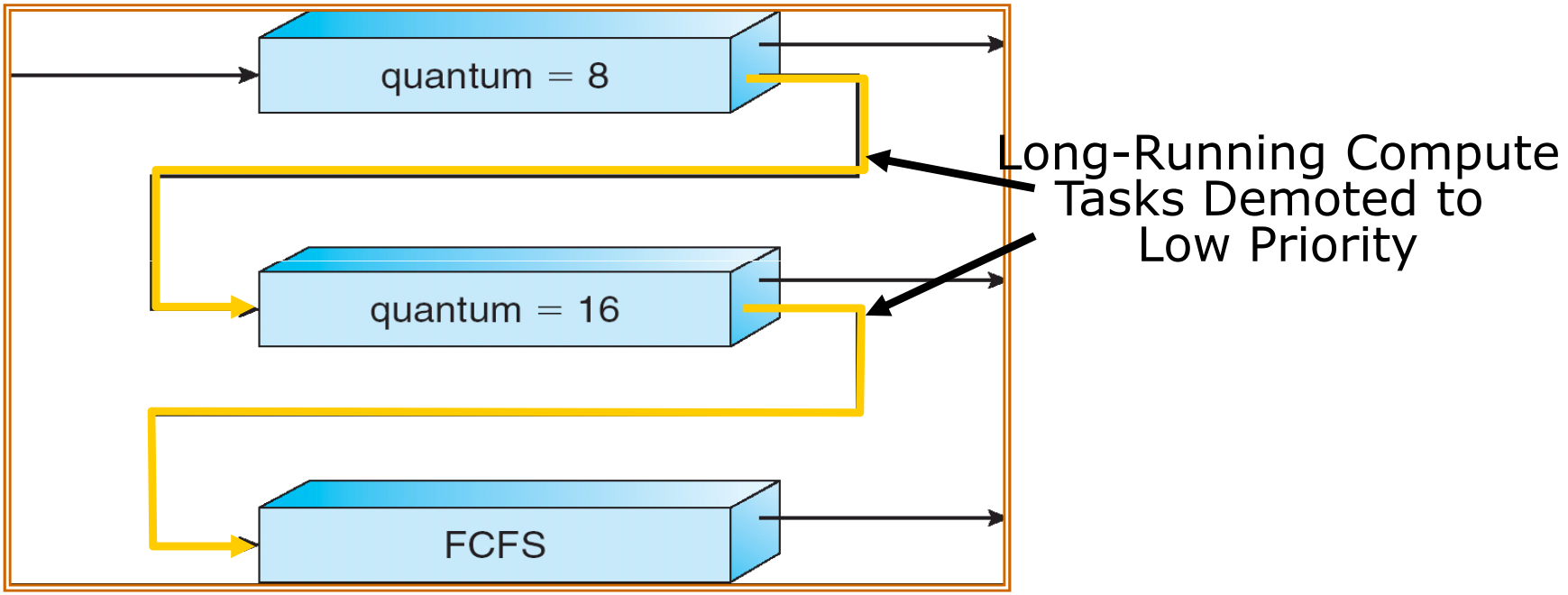


Multilevel Feedback Queue

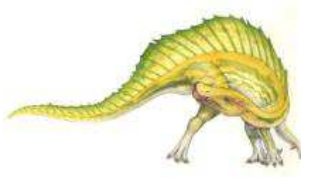
- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service



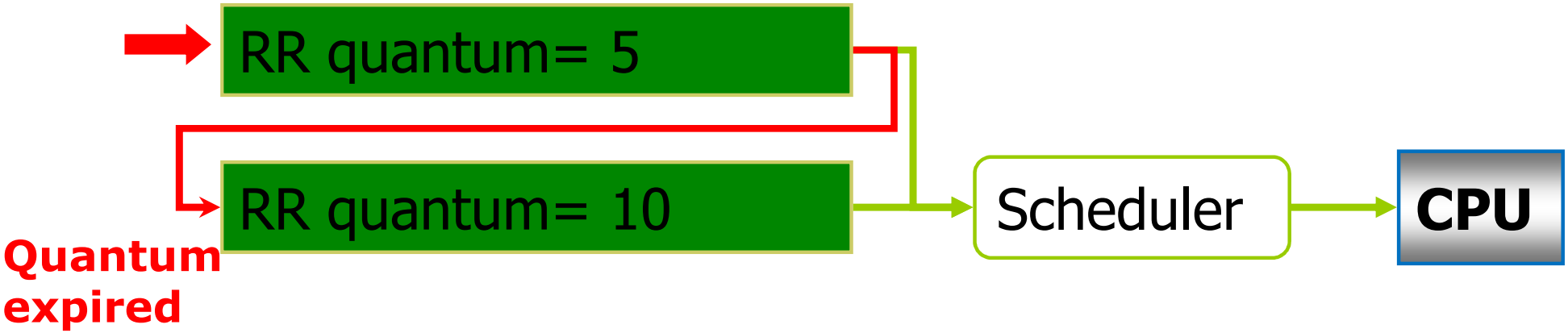
Multi-Level Feedback Scheduling



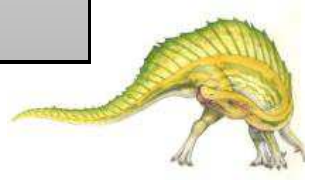
- Adjust each job's priority as follows (details vary)
  - Job starts in highest priority queue
  - If timeout expires, drop one level
  - If timeout doesn't expire, push up one level (or to top)



Multilevel Feedback Ex.

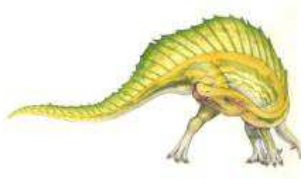


	Arriving time	CPU	I/O	CPU	Tin	Tout	Te	Tr
P1	0	8	X	X				
P2	2	4	6	2				
P3	4	6	X	X				
P4	7	3	8	3				
P5	9	2	X	X				



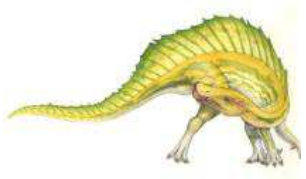
Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling:
  - all processes in common ready queue, or
  - each has its own private queue of ready processes
- **SMP** currently most common



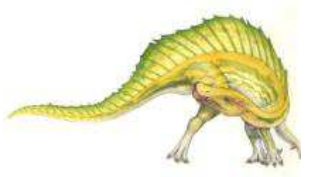
Multiple-Processor Scheduling

- **Processor affinity** – process has affinity for processor on which it is currently running
  - **soft affinity**
  - **hard affinity**
  - Variations including **processor sets**



Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pulls waiting task from busy processor



Example: Sept. 2006 exam

8.- En un sistema con planificación de colas de realimentación multinivel de tres niveles (expropiativo) se ejecutan tres procesos.

Los tiempos de computación pura de los procesos (total CPU) son los siguientes: P1=60 ciclos, P2=130 ciclos y P3=180 ciclos. Además el proceso P1 realiza una operación de E/S cuando han pasado 20 ciclos de su tiempo de computación. El proceso P2 realiza una operación de E/S cuando han pasado 80 ciclos de su tiempo de computación. El proceso P3 no realiza operaciones de E/S. Cada operación de E/S dura 20 ciclos.

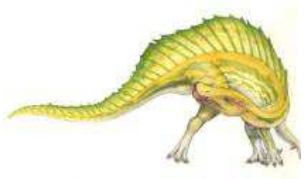
Los cuantos de tiempo asociados a cada cola, a medida que desciende la prioridad de las colas son Cola1: 50, Cola2: 100 y Cola3: 120 ciclos (la Cola1 es la de mayor prioridad).

Suponiendo que, inicialmente, todos los procesos llegan la mismo tiempo al sistema de planificación y que el orden inicial en la cola es P1(primero), P2(segundo) y P3 (tercero), completar el diario de ejecución en la tabla siguiente anotando el tiempo en el cual ocurre cada nuevo evento, dónde se encuentra cada proceso en ese momento y la descripción del evento.

Tiempo	Ejecución	Cola1	Cola2	Cola3	Bloqueados	Descripción evento
0	P1	P2,P3				Llegada de los procesos P1,P2, P3 Despacho de P1

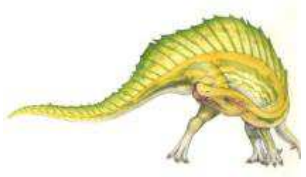
Contents

- **Processes**
  - Process concept
  - Process states
  - Process Control Block (PCB)
  - Operations on Processes
- **Threads**
  - Thread concept
  - Libraries to create threads
- **Operating System Examples**
- **CPU scheduling**
  - Basic concepts
  - Scheduling Algorithms
  - **Examples**



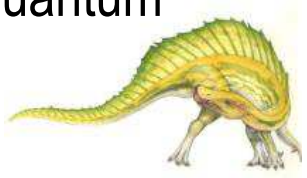
Examples: Windows

Windows Scheduling



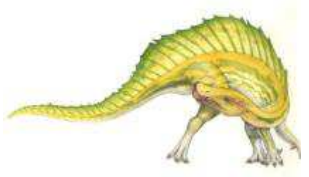
Examples: Windows

- Windows: scheduling rules
  - 7/10 is a preemptive OS with the **thread** as the unit to be scheduled
  - Preemptive scheduling based on priorities:
    - Static for real time [16..31], only the administrator (static class)
    - Variable for user [1..15], typical applications (variable class)
  - Priority 0 is assigned to thread zero (idle)
  - Real time processes have the highest priority
  - Promotes interactive threads increasing priority
  - Timesharing via Round Robin:
    - Quantum durations can vary between clients and servers:
    - longer on a server to maximize throughput
    - shorter quantums overall client computers, but provides a longer quantum to the thread associated with the current foreground window



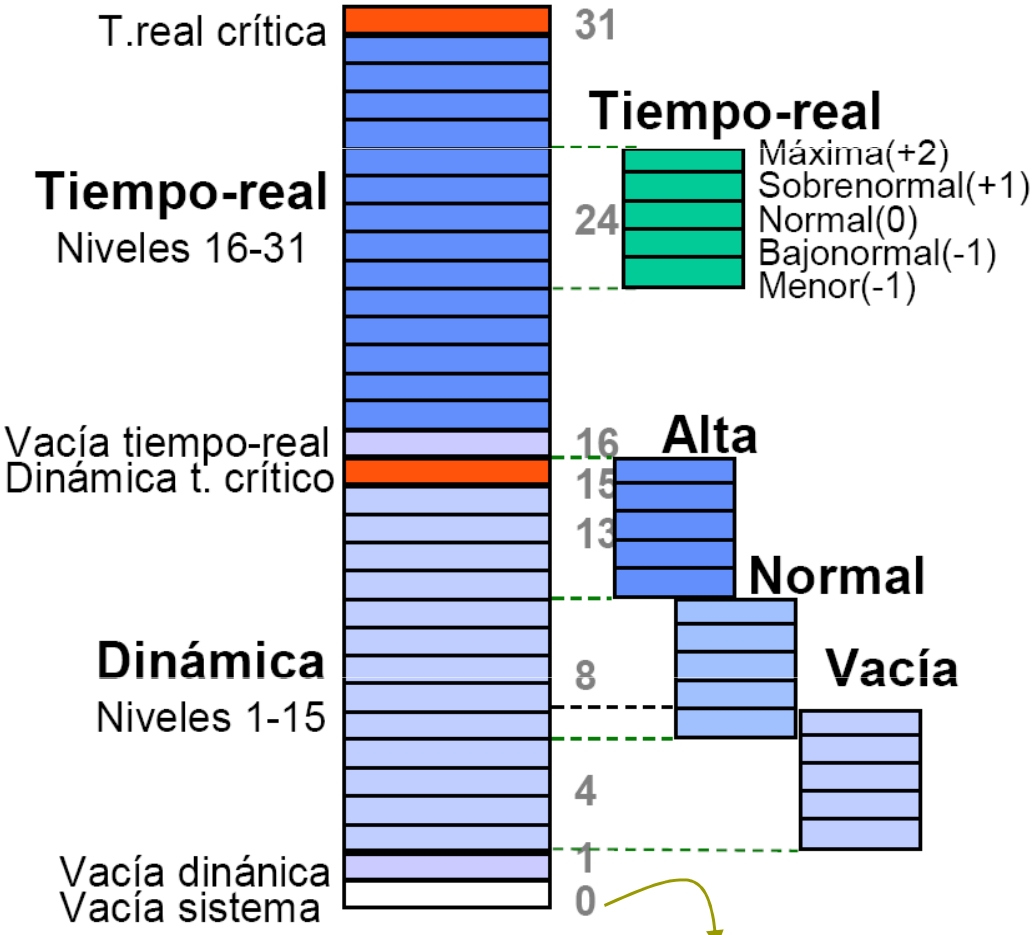
Examples: Windows

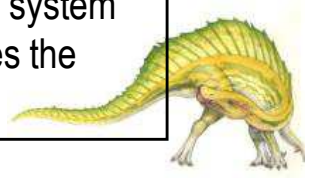
- Assign priority: two stage
  - assign priority class (of the process)
  - assign relative priority (for each thread)
- Priority class (priority base)
  - Realtime (24)
  - High (13)
  - Above Normal (10)
  - Normal (8)
  - Below Normal (6)
  - Idle (4)



Examples: Windows

- Add to the base priority of a thread its relative priority
- Move ±2:
  - Highest (+2)
  - Above normal (+1)
  - Normal (+0)
  - Below Normal (-1)
  - Lowest (-2)
  - Time_Critical (15 for all classes apart from real-time 31)
  - Idle (1 for all classes apart from real-time 16)

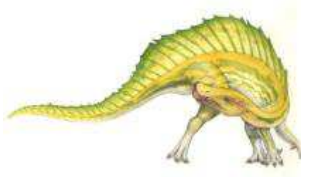




Examples: Windows

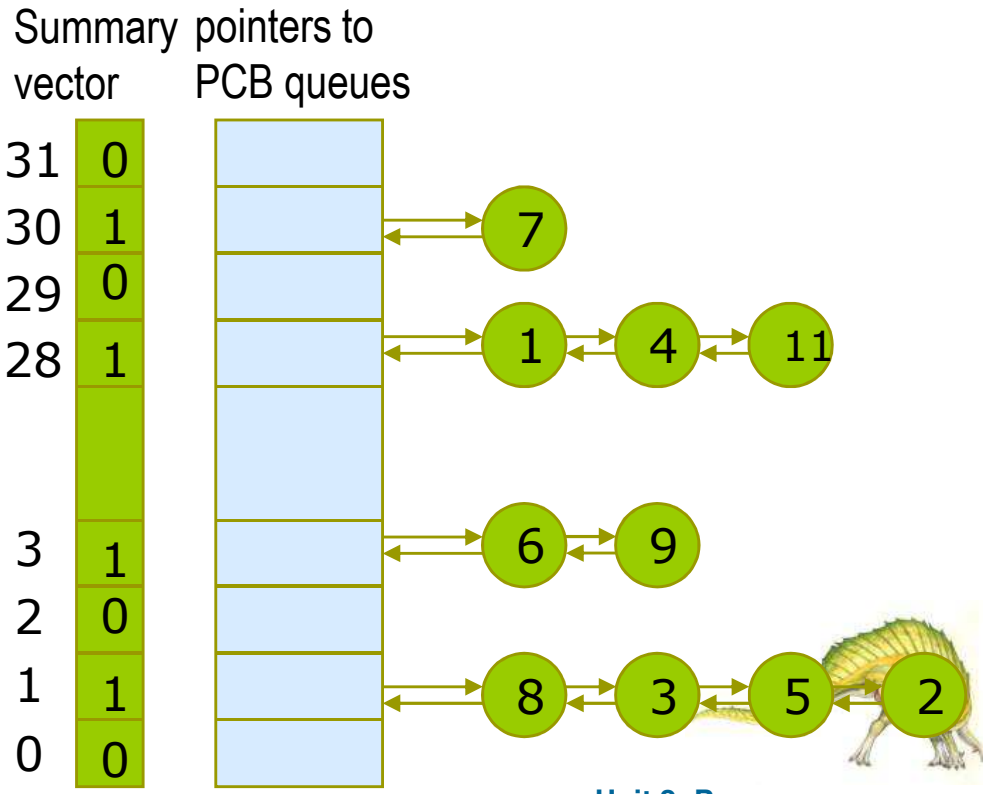
Priority table

		Win32 process class priorities					
		Realtime	High	Above Normal	Normal	Below Normal	Idle
Win32 thread priorities	Time critical	31	15	15	15	15	15
	Highest	26	15	12	10	8	6
	Above normal	25	14	11	9	7	5
	Normal	24	13	10	8	6	4
	Below normal	23	12	9	7	5	3
	Lowest	22	11	8	6	4	2
	Idle	16	1	1	1	1	1



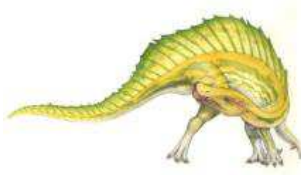
Examples: Windows

- Scheduler has 32 queues for ready threads
  - One queue per priority
- It chooses the first ready thread from the highest priority queue not empty
- Round-roubin in each queue
- The preempted thread
  - **Finished quantum** put at the tail of its priority queue (RR)
  - **Preempted by higher priority thread** put at the head of its priority queue



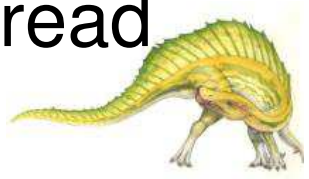
Examples: Windows

- Automatic priority update
  - User threads change in multilevel feedback queue
    - for priorities 1 to 15
    - increase (boosts) of priority may happen
    - and decays of priority
  - Promotes I/O intensive processes and avoids starvation
  - No update for real-time threads (priority > 15)
    - The scheduling is predictable: keeps the original priority for each thread. However, NO guarantee on response times. Only appropriate in the context of soft real-time



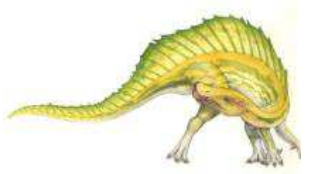
Examples: Windows

- Increase in thread priority
  - When process blocks, usually due to I/O request
    - Slow devices give large increases
      - For instance keyboard and mouse: + 6 (interactive processes)
    - Fast devices give small increases (p.e. HD + 1)
  - Increase with respect to the base priority
    - Can never go over 15 (maximum for user processes)
  - Good response time for interactive, keep I/O devices busy
- Decrease in thread priority
  - each time a quantum is used, decrease priority by 1 up to reaching the base priority of the thread



Examples: Windows

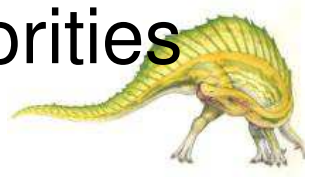
- Avoid starvation
  - The “Balance Set Manager” is a thread with priority (system) 16 waking up each second
  - Looks for user threads that have been ready without running for more than 3 seconds
  - Increases their priority and quantum
    - To priority 15 (maximum for user process)
    - Double time quantum
  - Temporal increase: only up to using its quantum or blocking. After returns to its normal priority and quantum



Examples: Windows

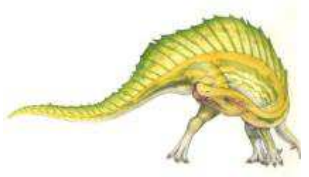
Properties of the scheduler

- Automatic priority increase a bit arbitrary
  - Promotes threads that use I/O devices
    - Increases are fixed
    - Sound card has a maximum increase of +8
  - Being fast can be playing a lot of music
- Starvation is implicitly not avoided
  - No aging, one should keep track of indefinite discrimination
- Soft real-time support
  - No guaranteed response times
  - Scheduling is predictable due to static priorities



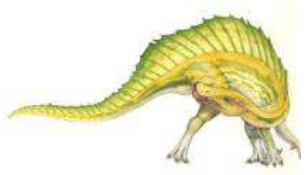
Examples: Linux

Linux Scheduling



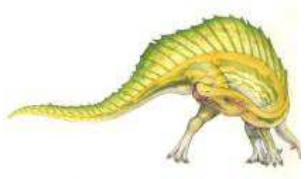
Examples: Linux

- Objectives Unix (conflicting?)
  - Low response time, high productivity (throughput) for background, avoid starvation, balance needs of high and low priority processes...
- Scheduling algorithm
  - Schedule processes and threads (tasks)
  - Preemptive scheduling based on **priorities** (multilevel queues)
  - Timesharing via Round Robin
  - Promotes interactive processes increasing their priority



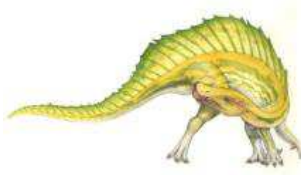
Linux real-time Scheduling

- Real-time is scheduled before normal processes
- Scheduling by preemptive priority
  - Always select real-time process with the highest priority; if several, select first in the queue
  - Static priorities 1..99
- Two classes for real-time scheduling with equal priority
  - FIFO – in order of arrival (SCHED_FIFO)
    - run up to finish, no quantum is set
    - only preempted by a real-time process of higher priority
  - Round Robin –circular turn (SCHED_RR)
    - removed when Q used up and returns to the tail of the ready list with the same priority
    - time is shared by real-time processes of the same priority
  - If preempted, moved to the head of the queue

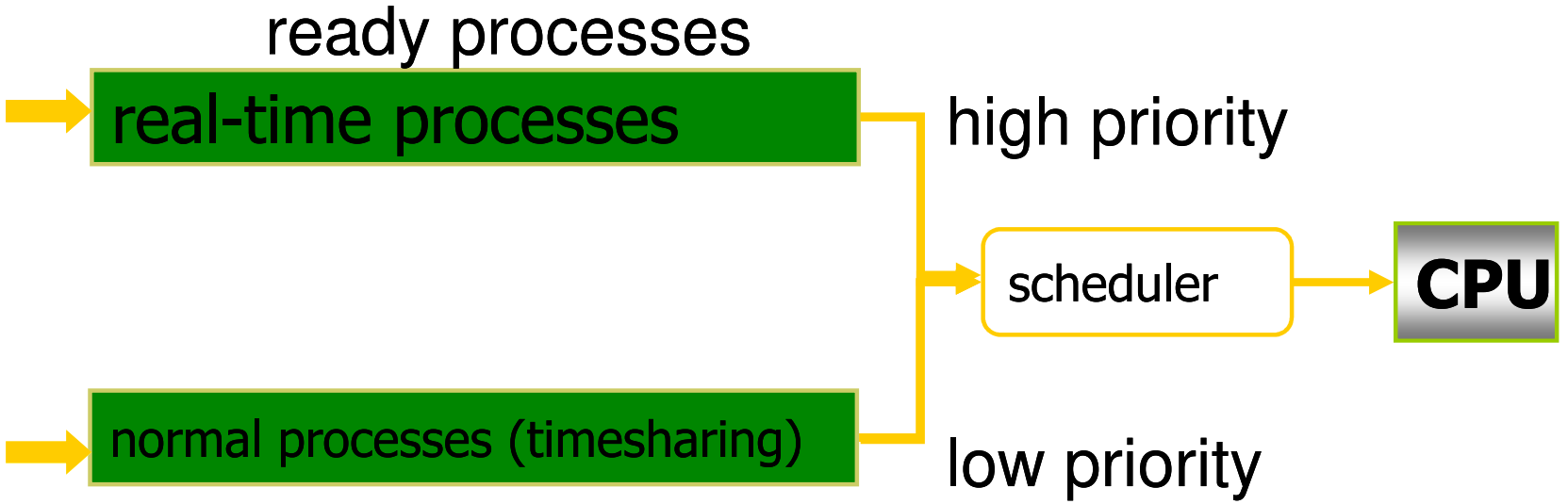


Linux Scheduling: O(1)

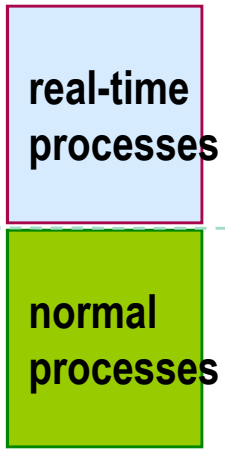
- From kernel 2.6: scheduling in constant time
- Multilevel queues
  - Real-time processes (no feedback)
  - Normal processes (feedback)
- Two process classes – two strategies
  - Real-time processes
    - **Always dispatch before** normal processes
    - Static priorities
    - Not fair, most important: absolute priority
  - Normal processes (timesharing)
    - Dynamic priorities
    - Try being fair
    - Low priority processes may reach the CPU
    - Promote interactive processes

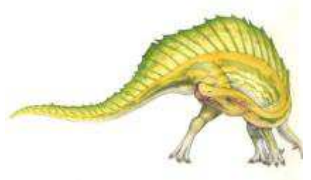


Linux Scheduling: O(1)



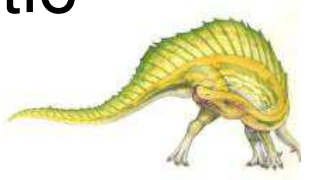
numeric priority	relative priority
1	highest
•	
•	
•	
99	
100 (-20)	
•	
•	
•	
139 (19)	lowest





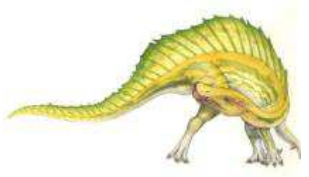
Linux Time-Sharing Scheduling: O(1)

- Preemptive priority scheduling
  - always select the process of highest priority
- Dynamic priority
  - determine starting with base priority or static priority (100..139) (nice -20...19). Typically 120
  - processes inherit priority from parent
  - users may **lower** the base priority (calls nice() and setpriority()) of their processes
- Round Robin scheduling for processes of the same priority
  - fixed quantum depends directly on the static priority of the process



Linux Time-Sharing Scheduling: O(1)

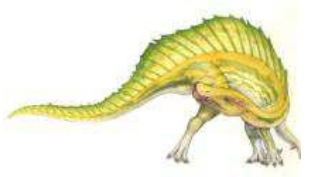
- Determine dynamic priority
  - each process has a base priority (nice) or static inherited from the parent process
    - by default this priority is 120
  - the dynamic priority is determined starting from the base by promoting interactive processes
    - dynp = max (100, min(statp – bonus +5, 139))
    - bonus = 0..10
      - reward (to -5) I/O intensive processes
      - punish (to +5) CPU intensive processes
  - the reward is determined based on the average sleeping time (TMD in milliseconds)
    - When process “awakes” →TMD+= sleep time
    - When process leaves CPU → TMD-= execution time
    - 0 ≤ TMD ≤ 1000 ms
    - bonus = floor(TMD/ 100)



Linux Time-Sharing Scheduling: O(1)

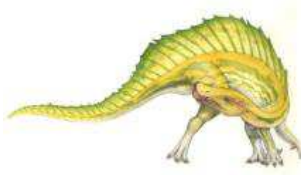
- Determine the dynamic priority
  - dynamic interval (±5)
    - sufficient to promote interactive processes
    - not to distort completely the base priority
      - priority 112 process never more urgent than priority 101
  - recalculate priority if the process has used its quantum, Q
  - fixed Q depending on the static priority of the process
    - $Q = \begin{cases} (140 - \text{static priority}) \times 20 & \text{if static priority} < 120 \\ (140 - \text{static priority}) \times 5 & \text{if static priority} \geq 120 \end{cases}$

	Static priority	nice value	Q
Maximum	100	-20	800 ms
High	110	-10	600 ms
Default	120	0	100 ms
Low	130	10	50 ms
Minimum	139	19	5 ms



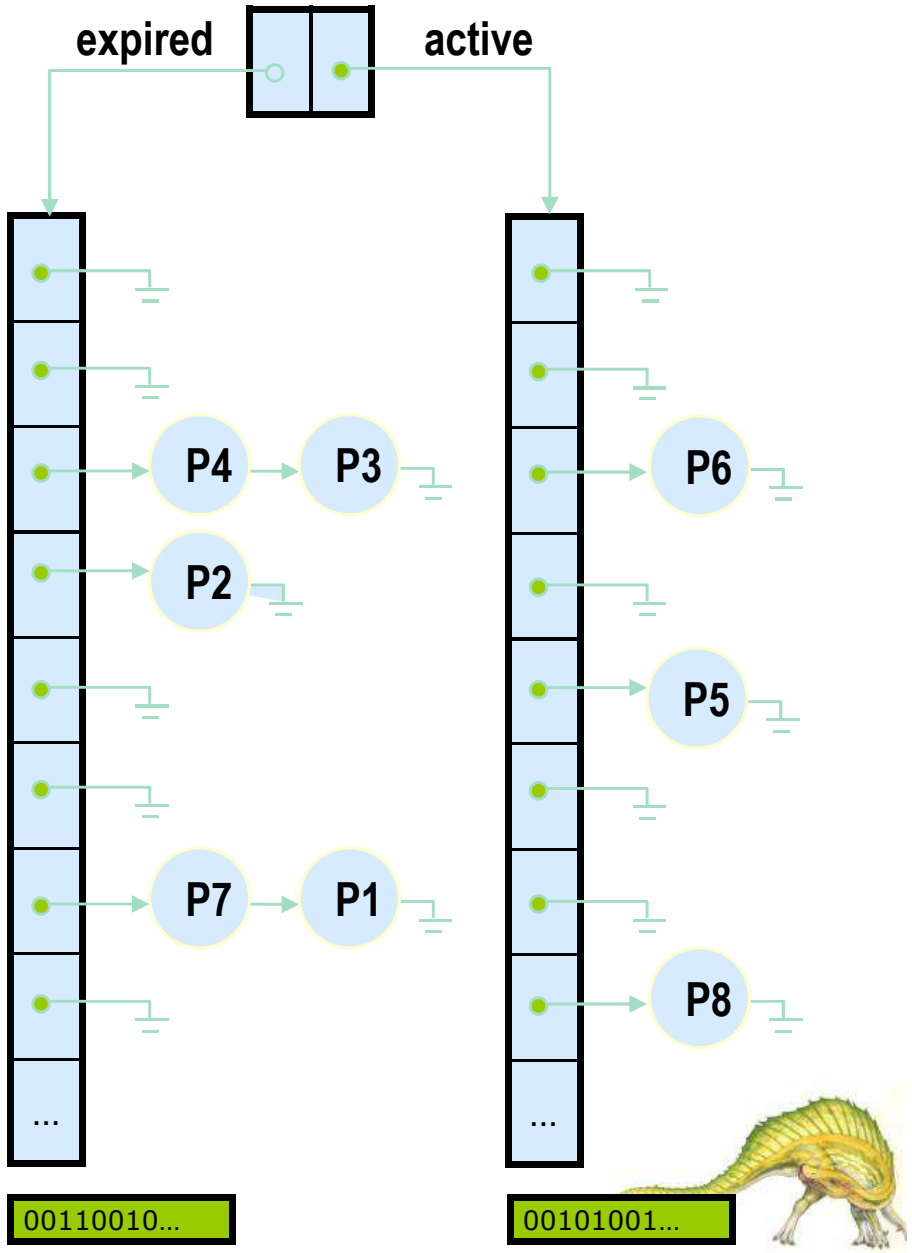
Linux Time-Sharing Scheduling: O(1)

- Modified Round Robin (RR)
  - two ready queues
    - queue “active list” of processes that did not use up their quantum yet
    - queue “expired list” of processes that have already used their quantum
  - next process to run is selected only from the “active”
  - if blocked, returning to active list when waking up
  - if using its time slice, is placed in expired list



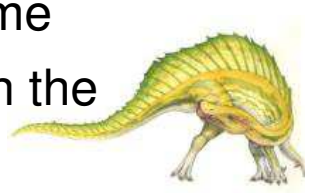
Linux Time-Sharing Scheduling: O(1)

- Modified Round Robin
  - avoid starvation:
    - a lower priority process is run despite higher priority ready processes exists. The latter have used their quantum.
  - "running periods" appear
    - all processes are run during an period
    - a period finishes if all processes have been run. No ready active exist.
  - period switch
    - simply interchange the expired queue with the active one



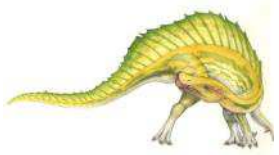
Linux Time-Sharing Scheduling: O(1)

- Modified Round Robin
  - Real implementation is a bit more sophisticated
  - Classify processes into Batch and Interactive
  - Interactive if:
    - dynamic priority ≤ 3 x static priority / 4 + 28
    - depends on bonus (TMD) of the process
  - Batch processes
    - if Q is used up, moved to “expired”
  - Interactive processes
    - usually stay in “active” when Q used up
    - moved to expired if:
      - the oldest expired process has been waiting for a long time
      - exists an expired process with a higher static priority than the interactive one



Linux Scheduling: CFS

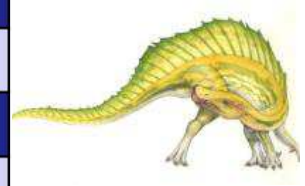
- From kernel 2.6.23:
  - Completely Fair Scheduler
  - $O(\log_2(n))$
- Two process classes – two strategies
  - Real-time processes
    - **Always dispatch before** normal processes
    - Static priorities
    - Not fair, most important: absolute priority
  - Normal processes (timesharing)
    - **Dynamic** priorities
    - Try being fair
    - Low priority processes may reach the CPU
    - Promote interactive processes



Linux Scheduling: CFS

- Idea:
  - To share CPU time proportionally to each running process NICE value
  - No quantum assigned to the process, just a portion of CPU time
  - NICE value acts as a weight for this portion
- NICE:
  - Value in [-20, 19] (lower value, higher weight)
  - NICE weights in CFS:

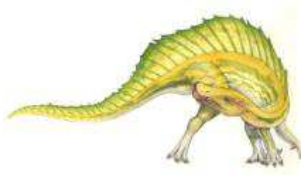
Nice	-20	-19	-18	-17	-16	-15	-14	-13	-12	-11
Weight	88761	71755	56483	46273	36291	29154	23254	18705	14949	11916
Nice	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
Weight	9548	7620	6100	4904	3906	3121	2501	1991	1586	1277
Nice	0	1	2	3	4	5	6	7	8	9
Weight	1024	820	655	526	423	335	272	215	172	137
Nice	10	11	12	13	14	15	16	17	18	19
Weight	110	87	70	56	45	36	29	23	18	15



Linux Scheduling: CFS

- Target latency (sched_latency)
  - Time all process in system **have to** enter to be executed
  - Used values: 20 ms (desktop), 100 ms (server)
  - Distribution proportional to NICE value
- Time slice (time_slice)
  - Time a process is allowed to executed without preemption
  - Proportional to its NICE value

$$time\_slice(P_i) = \frac{weight_{P_i}}{\sum weight_{P_j}} \times sched\_latency$$

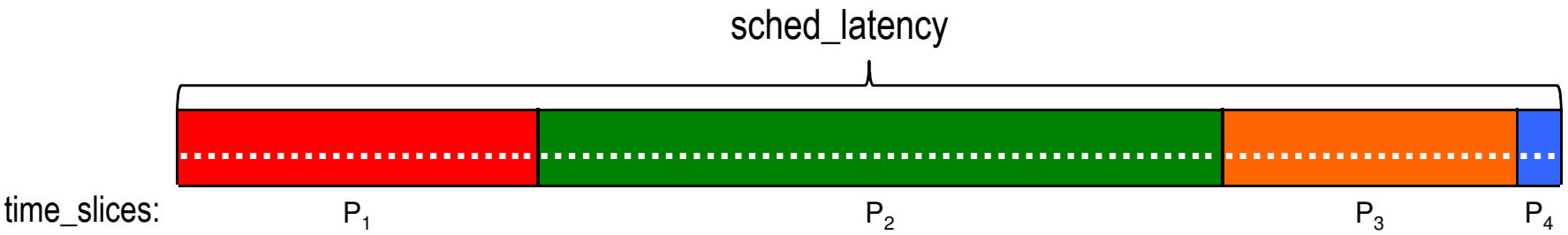


Linux Scheduling: CFS

$$time\_slice(P_i) = \frac{weight_{P_i}}{\sum weight_{P_j}} \times sched\_latency$$

- Example with 4 process
- $\Sigma weight_P$= 1024+1991+820+110 = 3945

	P_1	P_2	P_3	P_4
Nice	0	-3	2	10
Weight	1024	1991	820	110
Weight/Σw	0,26	0,50	0,21	0,03



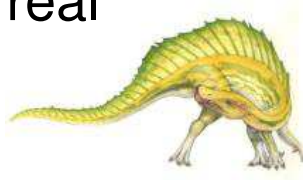
- time_slice(P) always ≥ min_granularity
  - If inequality not meet then sched_latency is increased

Linux Scheduling: CFS

- Virtual runtime
  - No “quantum” assigned, but used CPU percentage
  - Time consumed by each process has to be count
  - Virtual_runtime: accumulated execution time by a process inversely weighed by its weight

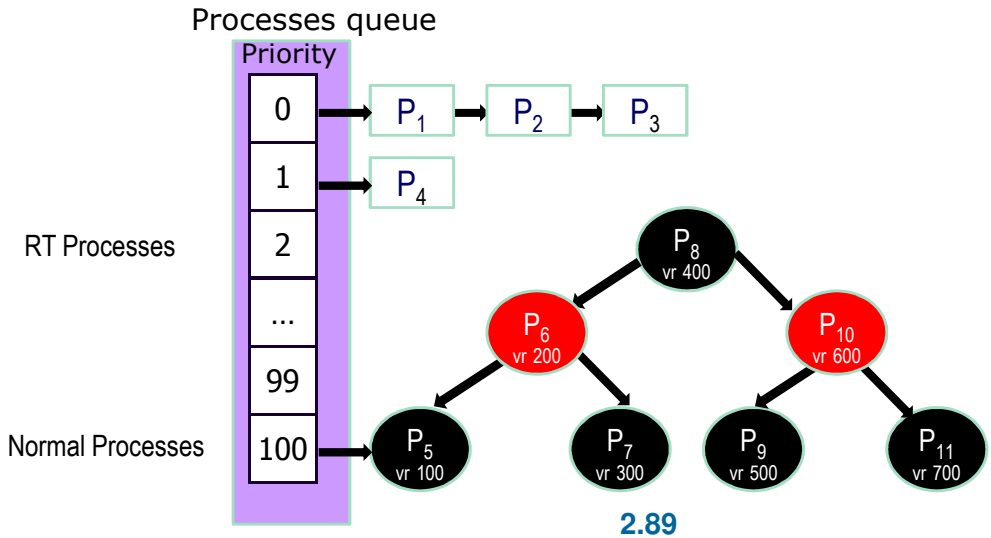
$$virtual\_runtime(P_i, t) += \frac{weight_0}{weight_{P_i}} \times tiempo\_ejec(P_i, t)$$

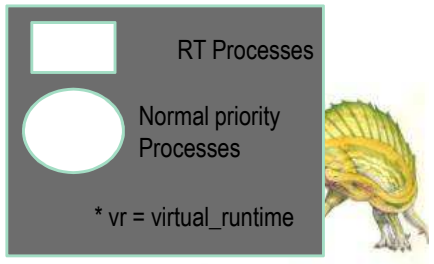
  - $Weight_0$= weight of nice 0
  - Processes with nice = 0 counted **exactly** their execution time
  - Processes with nice < 0 counted **less** time than their real execution time
  - Processes with nice > 0 counted **more** time than their real execution time



Linux Scheduling: CFS

- Processes queue
  - Only one READY queue for all processes (normal priority)
  - Sorted by increasing values of virtual_runtime
  - Implemented by a red-black tree
    - Insertions and deletions in $O(\log_2(n))$
    - Process to be executed: the leftmost of the tree





Linux Scheduling: CFS

- Scheduling algorithm CFS
  - In every scheduling “tick”
    - Substract to time_slice of current process the period of a tick
    - If time_slice becomes 0, NEED_RESCHED = true
  - Update current process virtual_runtime

$$virtual\_runtime(P_i, t) += \frac{weight_0}{weight_{P_i}} \times tiempo\_ejec(P_i, t)$$

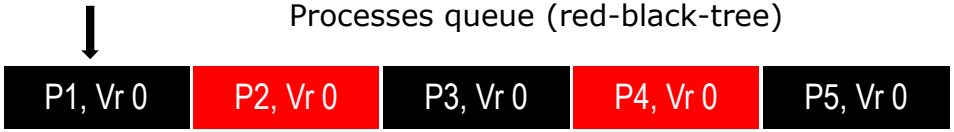
  - If NEED_RESCHED = true
    - Schedule task with the smallest virtual_runtime of the tree (the leftmost)

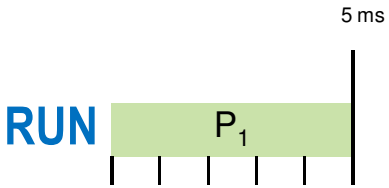
NEED_RESCHED can be true when a process is inserted in the tree (new, awakes) and gets in the first (leftmost with the smallest vr)

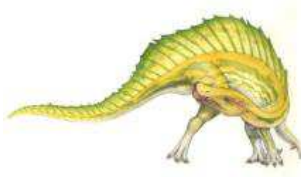
Linux Scheduling: CFS

- Example: 5 processes, sched_latency=6, tick=1 ms

	nice	$weight_i$	$weight_i/weight_\Sigma$	time_slice
P_1	-10	9548	0,6753	4,0518
P_2	-5	3121	0,2208	1,3248
P_3	0	1024	0,0724	0,4344
P_4	5	335	0,0237	0,1422
P_5	10	110	0,0078	0,0468
Total		14138	1,0000	6



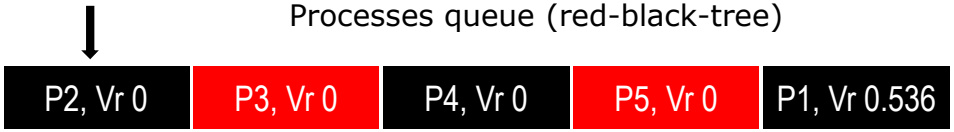




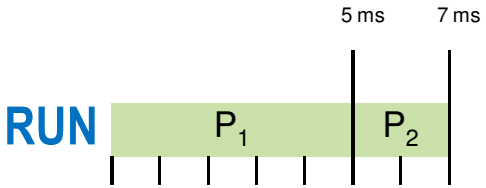
Linux Scheduling: CFS

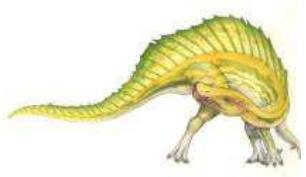
- Example: 5 processes, sched_latency=6, tick=1 ms

	nice	$weight_i$	$weight_i/weight_\Sigma$	time_slice
P_1	-10	9548	0,6753	4,0518
P_2	-5	3121	0,2208	1,3248
P_3	0	1024	0,0724	0,4344
P_4	5	335	0,0237	0,1422
P_5	10	110	0,0078	0,0468
Total		14138	1,0000	6



$Vr(P_1)$ = 5 ms * 1024/9548 = 0.536 ms

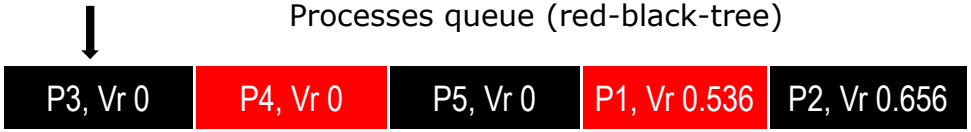




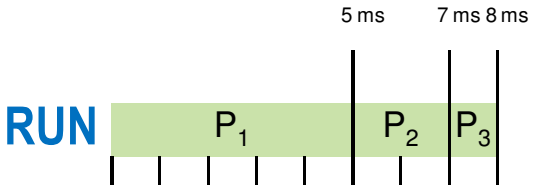
Linux Scheduling: CFS

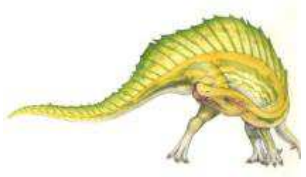
- Example: 5 processes, sched_latency=6, tick=1 ms

	nice	$weight_i$	$weight_i/weight_\Sigma$	time_slice
P_1	-10	9548	0,6753	4,0518
P_2	-5	3121	0,2208	1,3248
P_3	0	1024	0,0724	0,4344
P_4	5	335	0,0237	0,1422
P_5	10	110	0,0078	0,0468
Total		14138	1,0000	6



$Vr(P_2)$ = 2 ms * 1024/3121 = 0.656 ms

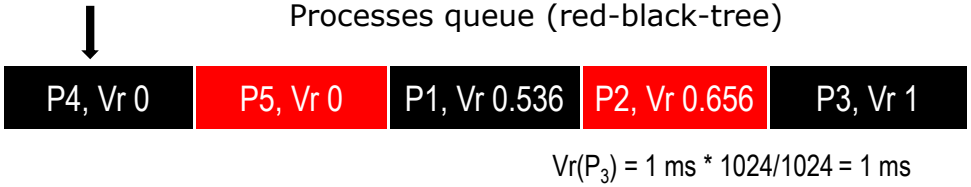




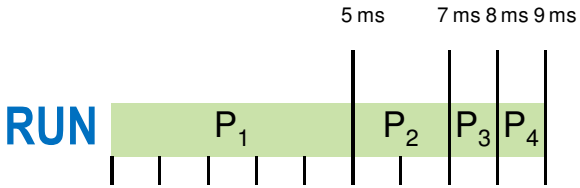
Linux Scheduling: CFS

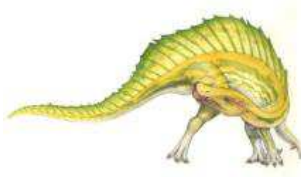
- Example: 5 processes, sched_latency=6, tick=1 ms

	nice	$weight_i$	$weight_i/weight_\Sigma$	time_slice
P_1	-10	9548	0,6753	4,0518
P_2	-5	3121	0,2208	1,3248
P_3	0	1024	0,0724	0,4344
P_4	5	335	0,0237	0,1422
P_5	10	110	0,0078	0,0468
Total		14138	1,0000	6



$Vr(P_3)$ = 1 ms * 1024/1024 = 1 ms

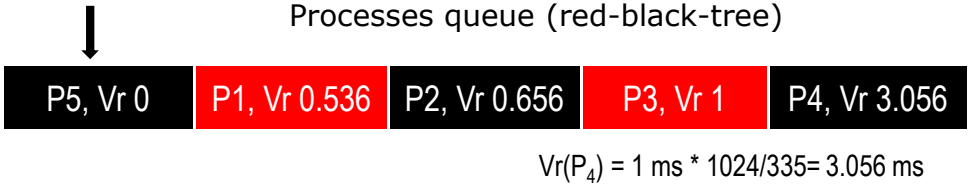




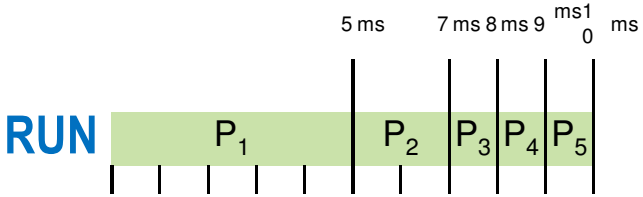
Linux Scheduling: CFS

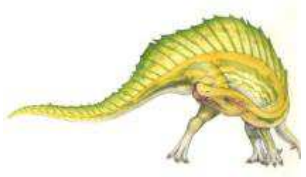
- Example: 5 processes, sched_latency=6, tick=1 ms

	nice	$weight_i$	$weight_i/weight_\Sigma$	time_slice
P_1	-10	9548	0,6753	4,0518
P_2	-5	3121	0,2208	1,3248
P_3	0	1024	0,0724	0,4344
P_4	5	335	0,0237	0,1422
P_5	10	110	0,0078	0,0468
Total		14138	1,0000	6



$Vr(P_4)$ = 1 ms * 1024/335= 3.056 ms

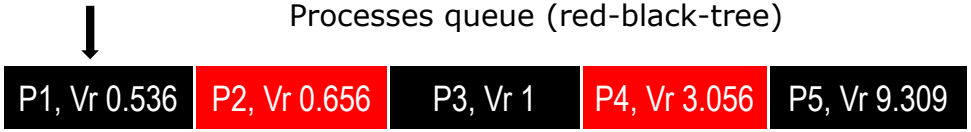




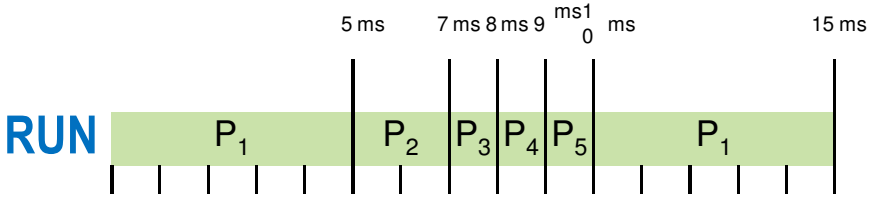
Linux Scheduling: CFS

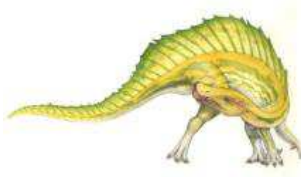
- Example: 5 processes, sched_latency=6, tick=1 ms

	nice	$weight_i$	$weight_i/weight_\Sigma$	time_slice
P_1	-10	9548	0,6753	4,0518
P_2	-5	3121	0,2208	1,3248
P_3	0	1024	0,0724	0,4344
P_4	5	335	0,0237	0,1422
P_5	10	110	0,0078	0,0468
Total		14138	1,0000	6



$Vr(P_5)$ = 1 ms * 1024/110 = 9.309 ms

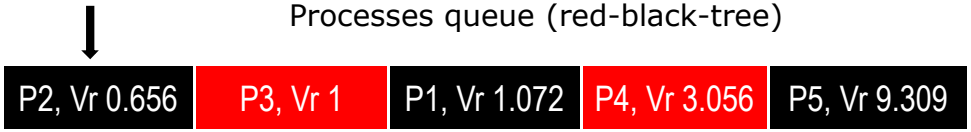




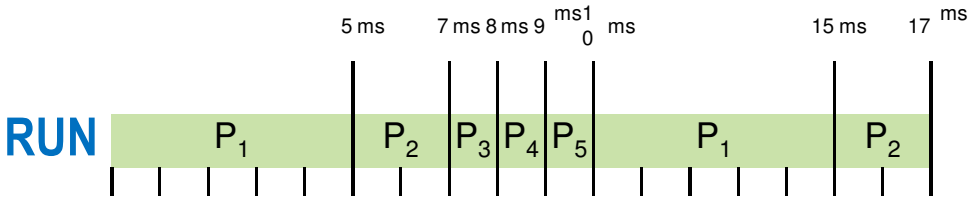
Linux Scheduling: CFS

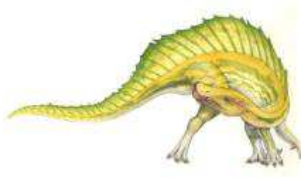
- Example: 5 processes, sched_latency=6, tick=1 ms

	nice	$weight_i$	$weight_i/weight_\Sigma$	time_slice
P_1	-10	9548	0,6753	4,0518
P_2	-5	3121	0,2208	1,3248
P_3	0	1024	0,0724	0,4344
P_4	5	335	0,0237	0,1422
P_5	10	110	0,0078	0,0468
Total		14138	1,0000	6



Vr(P1) = 0.536 + 5 ms * 1024/9548 = 1.072 ms

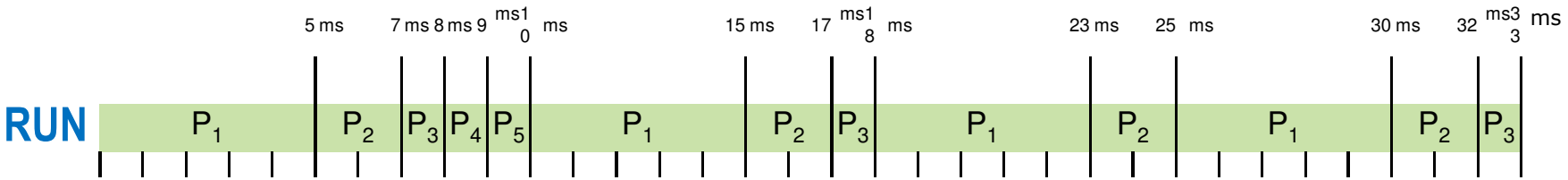


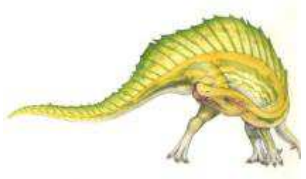


Linux Scheduling: CFS

- Example: 5 processes, sched_latency=6, tick=1 ms

	nice	$weight_i$	$weight_i/weight_\Sigma$	time_slice	Total time	%CPU
P_1	-10	9548	0,6753	4,0518	20 ms	60,6
P_2	-5	3121	0,2208	1,3248	8 ms	24,2
P_3	0	1024	0,0724	0,4344	3 ms	9
P_4	5	335	0,0237	0,1422	1 ms	3
P_5	10	110	0,0078	0,0468	1 ms	3
Total		14138	1,0000	6	33 ms	1





Bibliography

- A. SILBERSCHATZ, P. GALVIN, G. GAGNE, **Operating System Concepts.** 9th Edition, Wiley, 2014.
  - **Chapter 5.**
- J. CARRETERO, F. GARCÍA, P. DE MIGUEL, F. PÉREZ, **Sistemas Operativos. Una visión aplicada.** 2ª Edición, Mc Graw-Hill, 2007.
  - **Chapter 3.**
- W. STALLINGS, **Operating Systems. Internals and Design Principles.** 8th Edition, Pearson, 2014.
  - **Chapter 9 and 10.**