Course Syllabus

1. **Introduction** - History; Views; Concepts; Structure
2. **Process Management -** Processes; State + Resources; Threads; Unix implementation of Processes
3. **Scheduling** – Paradigms; Unix; Modeling
4. **Synchronization -** Synchronization primitives and their equivalence; Deadlocks
5. **Memory Management -** Virtual memory; Page replacement algorithms; Segmentation
6. **File Systems -** Implementation; Directory and space management; Unix file system; Distributed file systems (NFS)
7. **Security –** General policies and mechanisms; protection models; authentication
8. **Multiprocessors**

Scheduling: high-level goals

- Interleave the execution of processes to maximize CPU utilization while providing reasonable response time

- The scheduler determines:
  - Who will run
  - When it will run
  - For how long

Scheduling algorithms: quality criteria

- **Fairness**: *Comparable processes should get comparable service*
- **Efficiency:** *keep CPU and I/O devices busy*
- **Response time**: *minimize, for interactive users*
- **Turnaround time:** *average time for a job to complete*
- **Waiting time:** *minimize the average over processes*
- **Throughput:** *number of completed jobs per time unit*

More on quality criteria

- *Throughput* – number of processes completed per unit time
- *Turnaround time* – interval of time from process submission to completion
- *Waiting time* – sum of time intervals the process spends in the ready queue
- *Response time* – the time between submitting a command and the generation of the first output
- *CPU utilization* – the percentage of time in which the CPU is not idle

Criteria by system type

- **All systems**
  - Fairness - giving each process a fair share of the CPU
  - Policy enforcement - seeing that stated policy is carried out
  - Balance - keeping all parts of the system busy

- **Batch systems**
  - Throughput - maximize jobs per hour
  - Turnaround time - minimize time between submission and termination
  - CPU utilization - keep the CPU busy all the time

- **Interactive systems**
  - Response time - respond to requests quickly
  - Proportionality - meet users' expectations

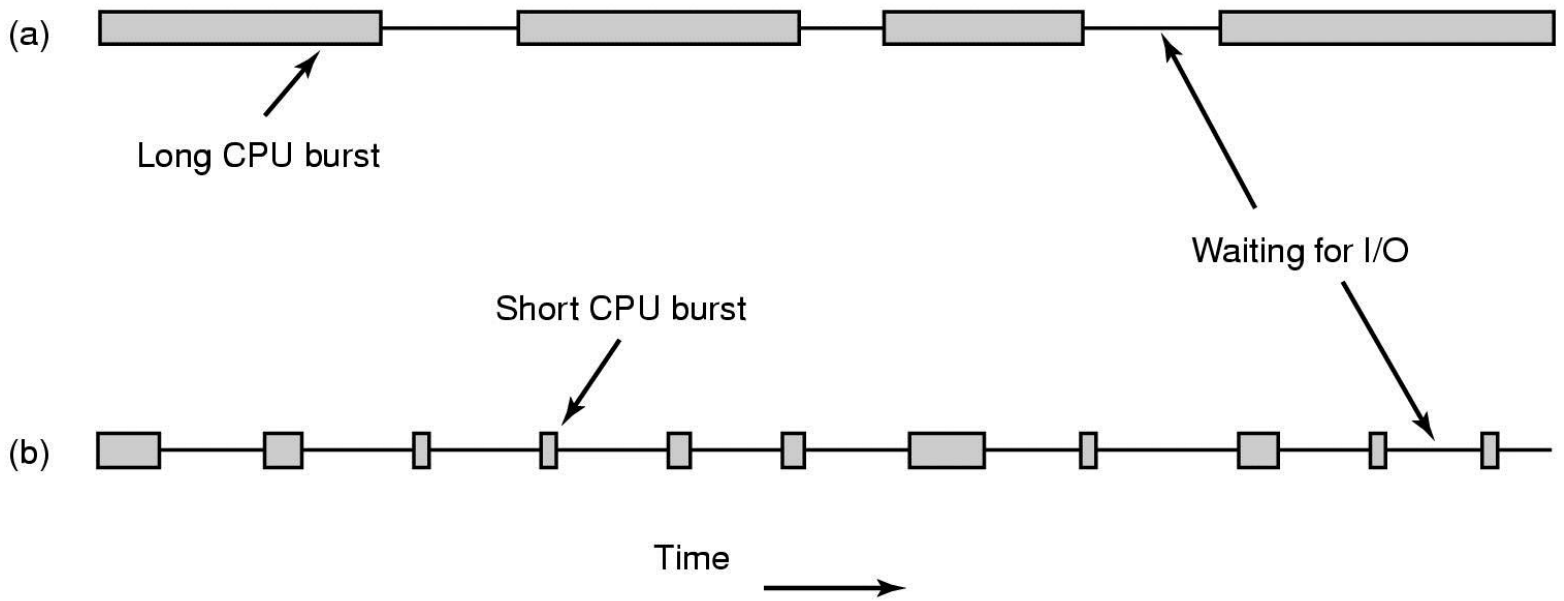
- **Real-time systems**
  - Meeting deadlines - avoid losing data
  - Predictability - avoid quality degradation in multimedia systems

Conflicting Goals

- Response Time vs. Turnaround time:
  A conflict between interactive and batch users

- Fairness vs. Throughput:
  How should we schedule very long jobs?

Scheduling – Types of behavior



- Bursts of CPU usage alternate with periods of I/O wait
  - CPU-bound processes
  - I/O bound processes

CPU Utilization vs. Turnaround time

We have 5 interactive jobs ***i1…i5*** and one batch job ***b***

Interactive jobs: 10% CPU; 20% disk I/O; 70% terminal I/O; total time for each job 10 sec.

Batch job: 90% CPU; 10% disk I/O; total time 50 sec.

Cannot run all in parallel !!

- ***i1..i5*** *in parallel - disk I/O is 100% utilized*
- ***b*** *and one* ***i*** *in parallel - CPU is 100% utilized*

CPU utilization vs. Turnaround time (II)

Two possible schedules:

1. First ***i1…i5***, then ***b***

   UT = (10x0.5 + 50x0.9)/60 = 83%

   TA = (10x5 + 60x1)/6 = 18.33sec.

2. ***b*** and each of the ***i***'s (in turn) in parallel:

   UT = (50x(0.9+0.1))/50 = 100%

   TA = (10+20+30+40+50+50)/6 = 33sec.

When are Scheduling Decisions made ?

1. Process switches from *Running* to *Waiting* (e.g. I/O)
2. New process created (e.g. fork)
3. Process switches from *Running* to *Ready* (clock..)
4. Process switches from *Waiting* to *Ready* (e.g. IO completion)
5. Process *terminates*

Preemptive scheduling:
process preepmtion may be initiated by scheduler

Which of above possible for non-preemptive scheduling? only 1 And 5.

Scheduling: outline

- Scheduling criteria
- Scheduling algorithms
- Unix scheduling
- Linux scheduling
- Win NT scheduling

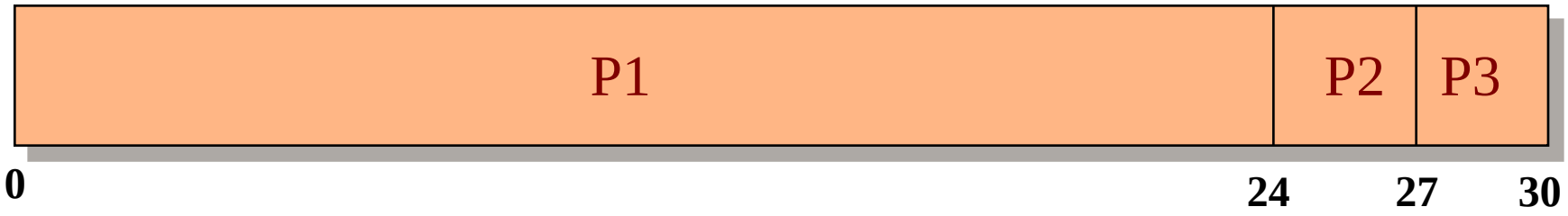
First-come-first-served (FCFS) scheduling

- Processes get the CPU in the order they request it and run until they release it
- Ready processes form a FIFO queue

FCFS may cause long waiting times

Process	Burst time (milli)
P1	24
P2	3
P3	3

If they arrive in the order P1, P2, P3, we get:



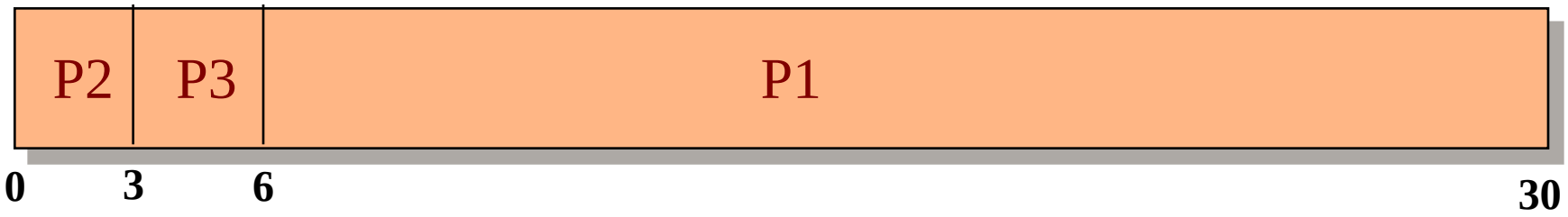
Average waiting time: **(0+24+27) / 3 = 17**

FCFS may cause long waiting times (cont'd)

Process	Burst time (milli)
P1	24
P2	3
P3	3

If they arrive in the order P1, P2, P3, average waiting time 17

What if they arrive in the order P2, P3, P1?



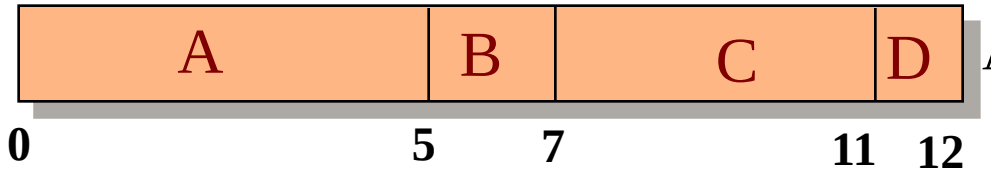
Average waiting time: **(0+3+6) / 3 = 3**

(Non preemptive) Shortest Job First (SJF) scheduling

- The CPU is assigned to the process that has the smallest next CPU burst
- In some cases, this quantity is known or can be approximated

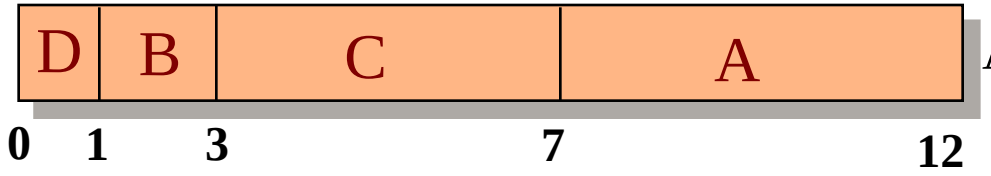
Process	Burst time (milli)
A	5
B	2
C	4
D	1

FCFS schedule



Average waiting time: **8.75**

SJF schedule



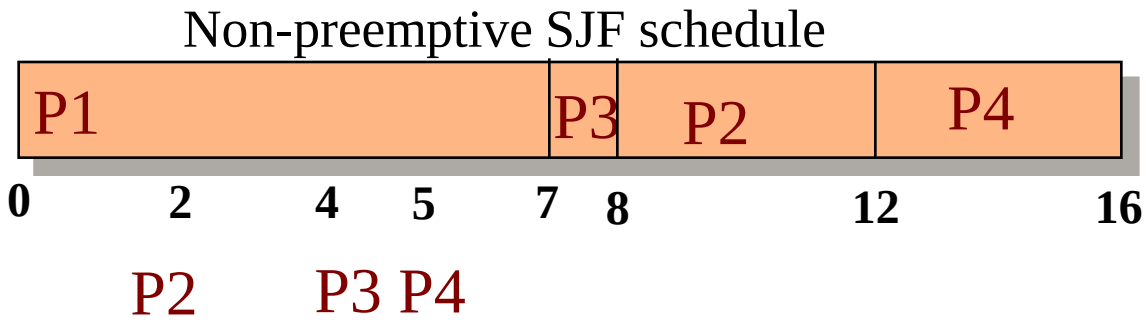
Average waiting time: **5.75**

Approximating next CPU-burst duration

- Can be done by using the length of previous CPU bursts
  - t_n = ***actual*** length of n^{th} CPU burst
  - T_{n+1} = ***predicted*** value for the next CPU burst
  - for $0 \leq \alpha \leq 1$ define the exponential average:
  - $$T_{n+1} = \alpha\, t_n + (1 - \alpha)\, T_n$$
  - α determines the relative weight of recent bursts

Non-preemptive SJF: example (varying arrival times)

Process	Arrival time	Burst time
P1	0	7
P2	2	4
P3	4	1
P4	5	5

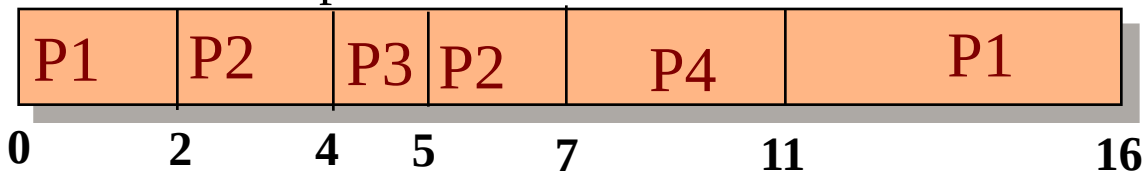


Average waiting time: **(0+6+3+7) / 4 = 4**

Preemptive SJF (Shortest Remaining Time First)

Process	Arrival time	Burst time
P1	0	7
P2	2	4
P3	4	1
P4	5	5

Preemptive SJF schedule



P1(5) P2(2) P4(4)

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

Round Robin - the oldest method

- Each process gets a small unit of CPU time (***time-quantum***), usually 10-100 milliseconds
- For ***n*** ready processes and time-quantum ***q***, no process waits more than **(*n* - 1)*q***
- Approaches FCFS when q grows
- Time-quantum ~ switching time (or smaller)

  relatively large waste of CPU time
- Time-quantum >> switching time

  long response (waiting) times, FCFS

Context switch overhead

- Context Switching is time consuming
- Choosing a long time quantum to avoid switching:
  - Deteriorates response time.
  - Increases CPU utilization
- Choosing a short time quantum:
  - Faster response
  - CPU wasted on switches

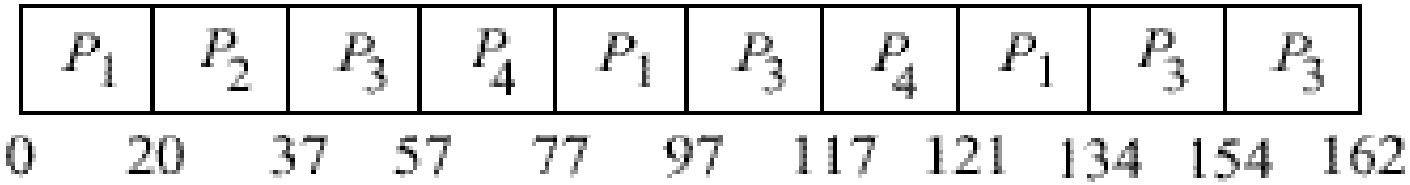
Context switch overhead - example

- Assume a context switch takes 5 milliseconds
- Switching every 20 ms. quantum would mean 20% of the CPU time is wasted on switching
- Switching after a 500 ms. quantum and having 10 concurrent users could possibly mean a 5 second response time
- possible solution: settle for 100 ms.

Example: RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:

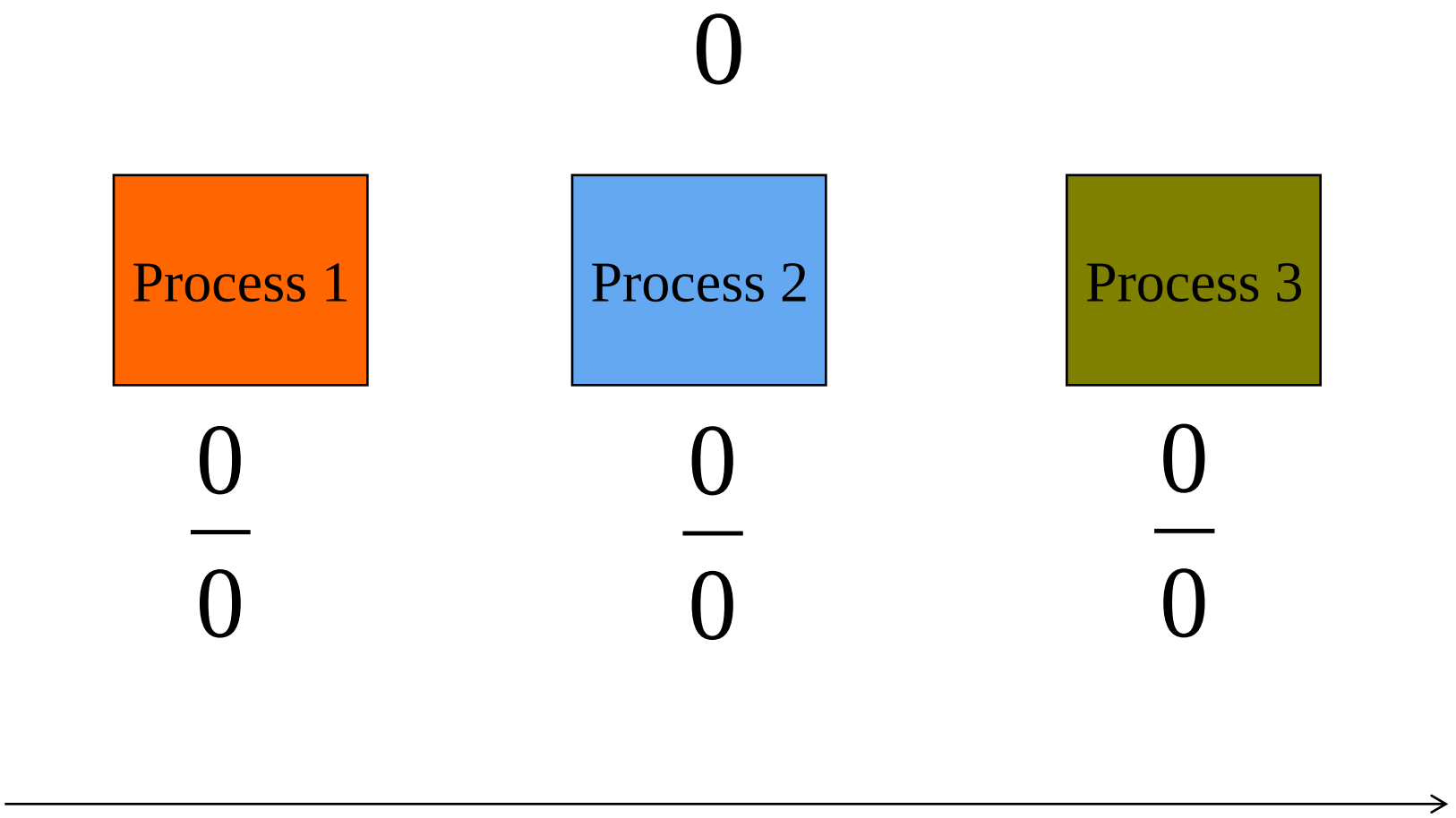


- Typically, higher average turnaround than SRTF, but better *response*.

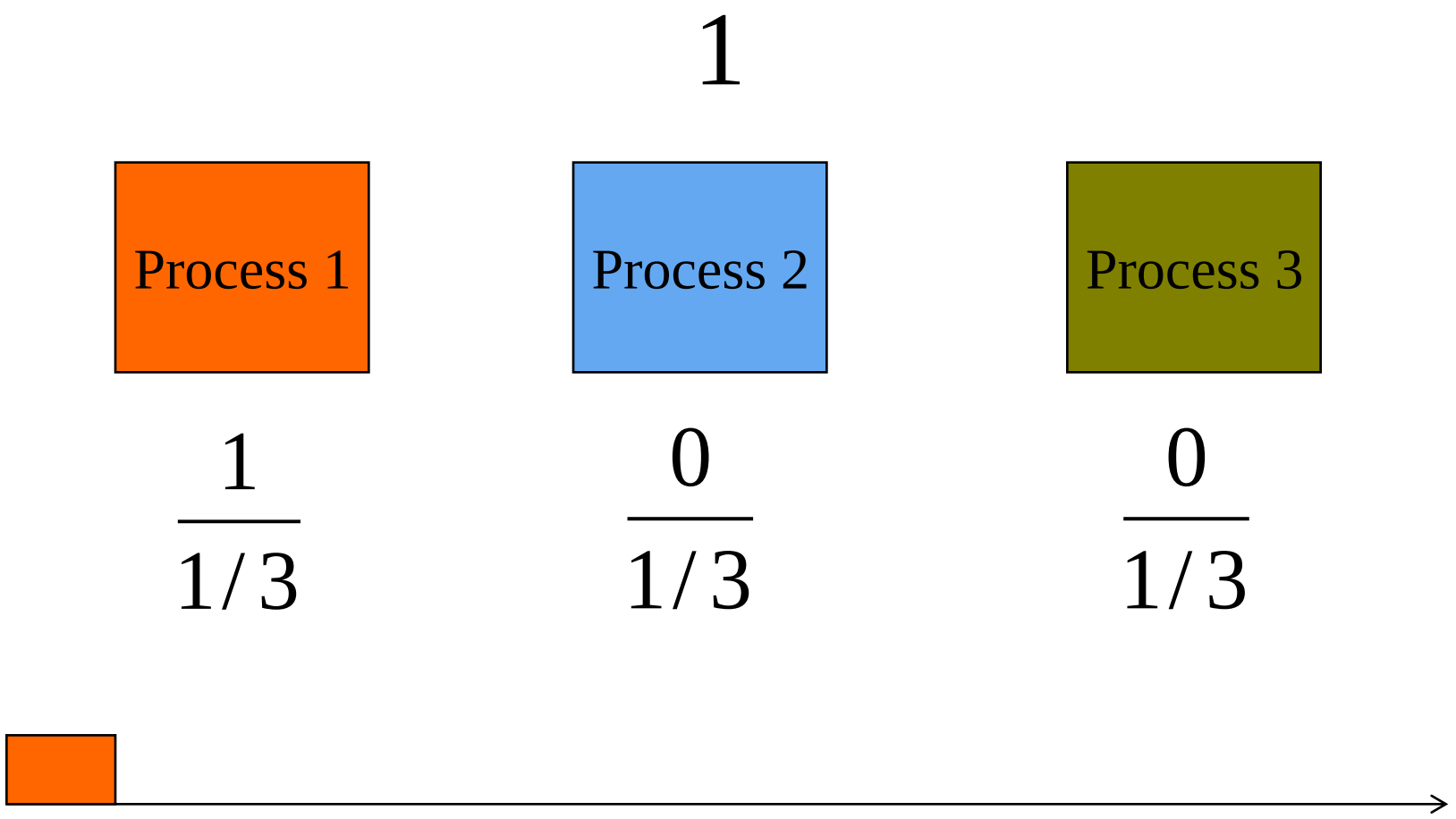
Guaranteed (Fair-share) Scheduling

- To achieve guaranteed 1/***n*** of cpu time (for ***n*** processes/users logged on):
- Monitor the total amount of CPU time per process and the total logged on time
- Calculate the ratio of allocated cpu time to the amount of CPU time each process is entitled to
- Run the process with the lowest ratio
- Switch to another process when the ratio of the running process has passed its *"goal ratio"*

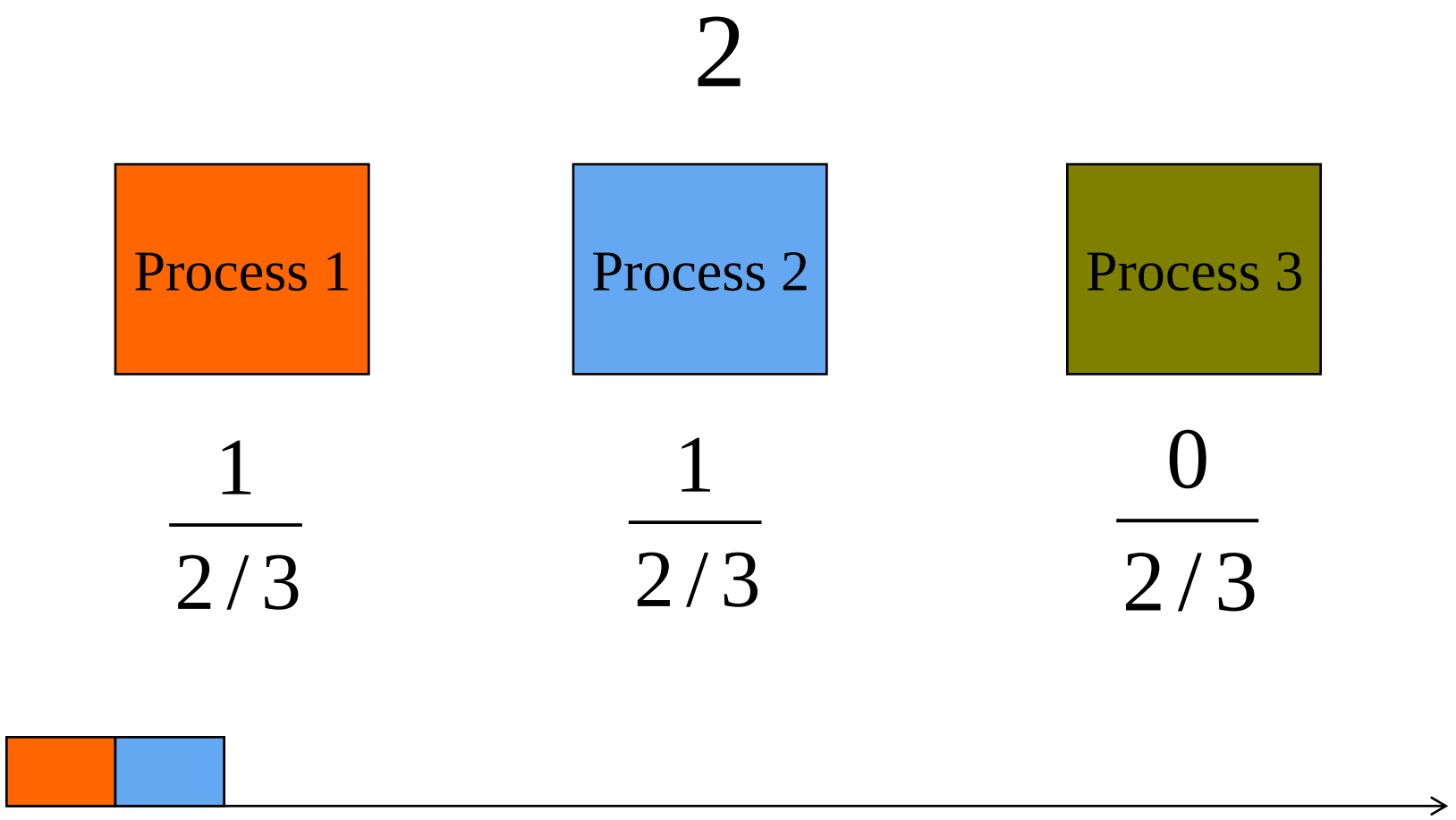
Guaranteed scheduling example



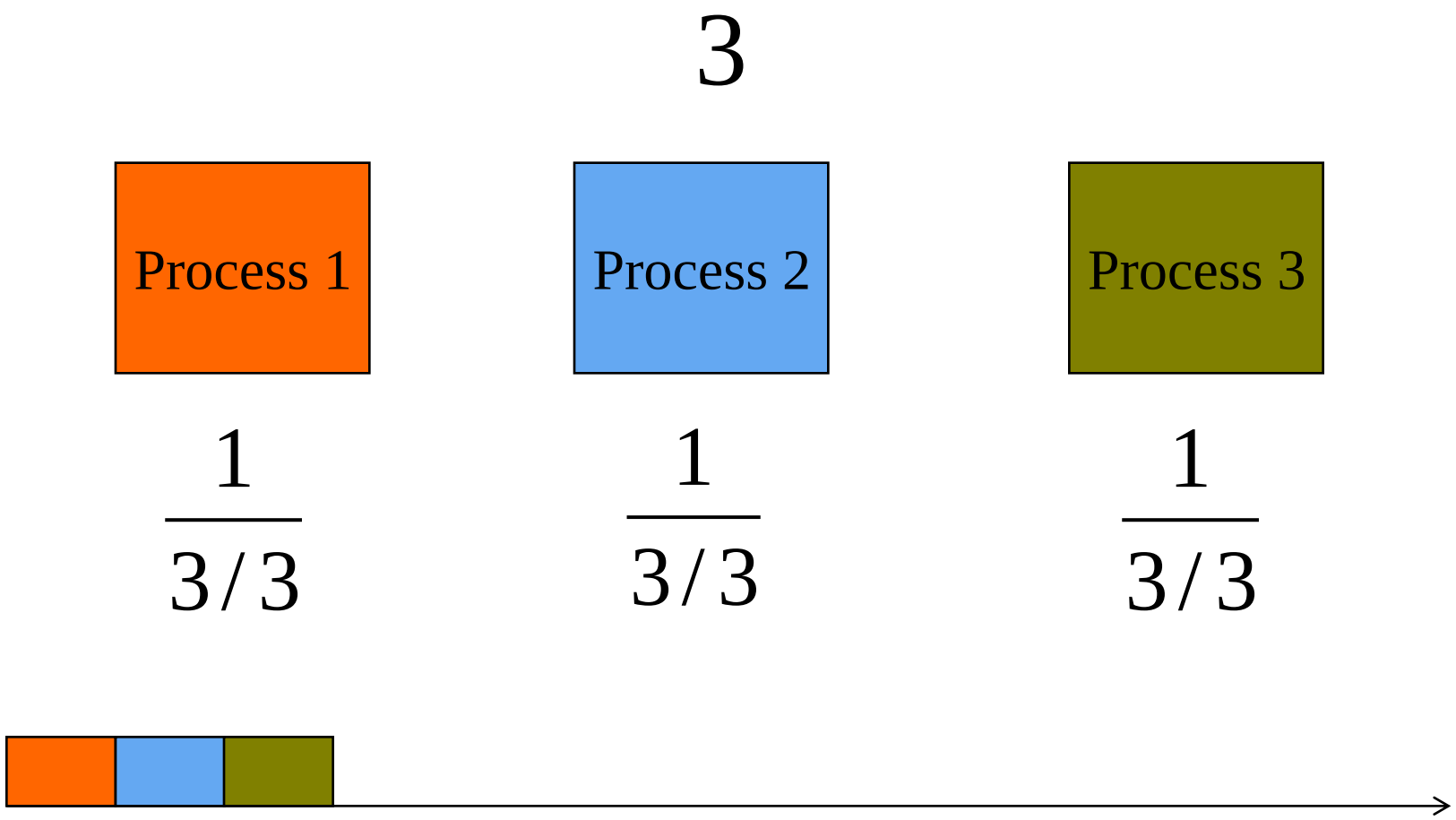
Guaranteed scheduling example



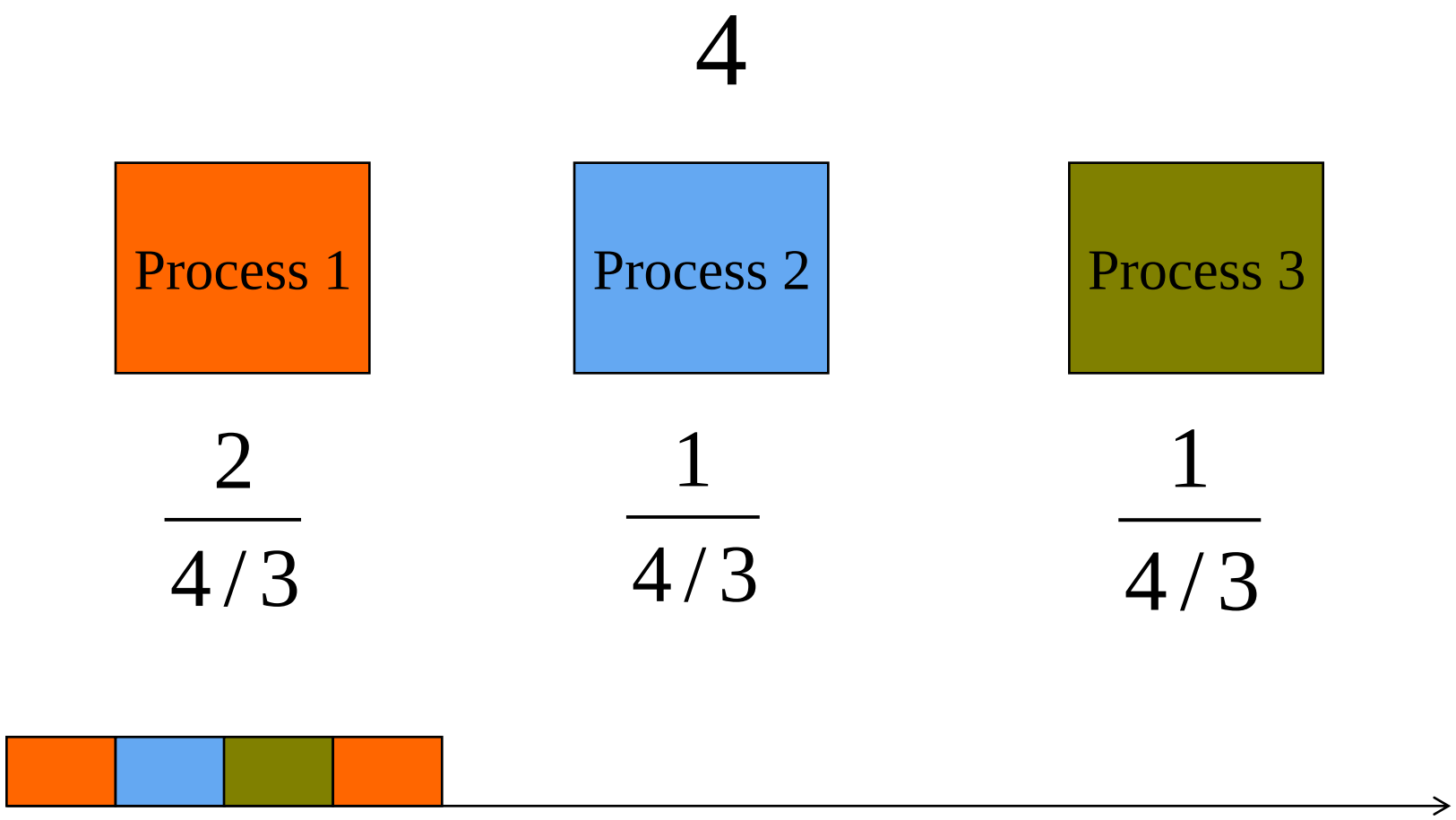
Guaranteed scheduling example



Guaranteed scheduling example



Guaranteed scheduling example



Guaranteed scheduling example

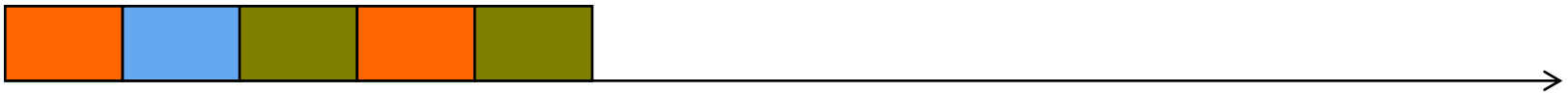
5

Process 1

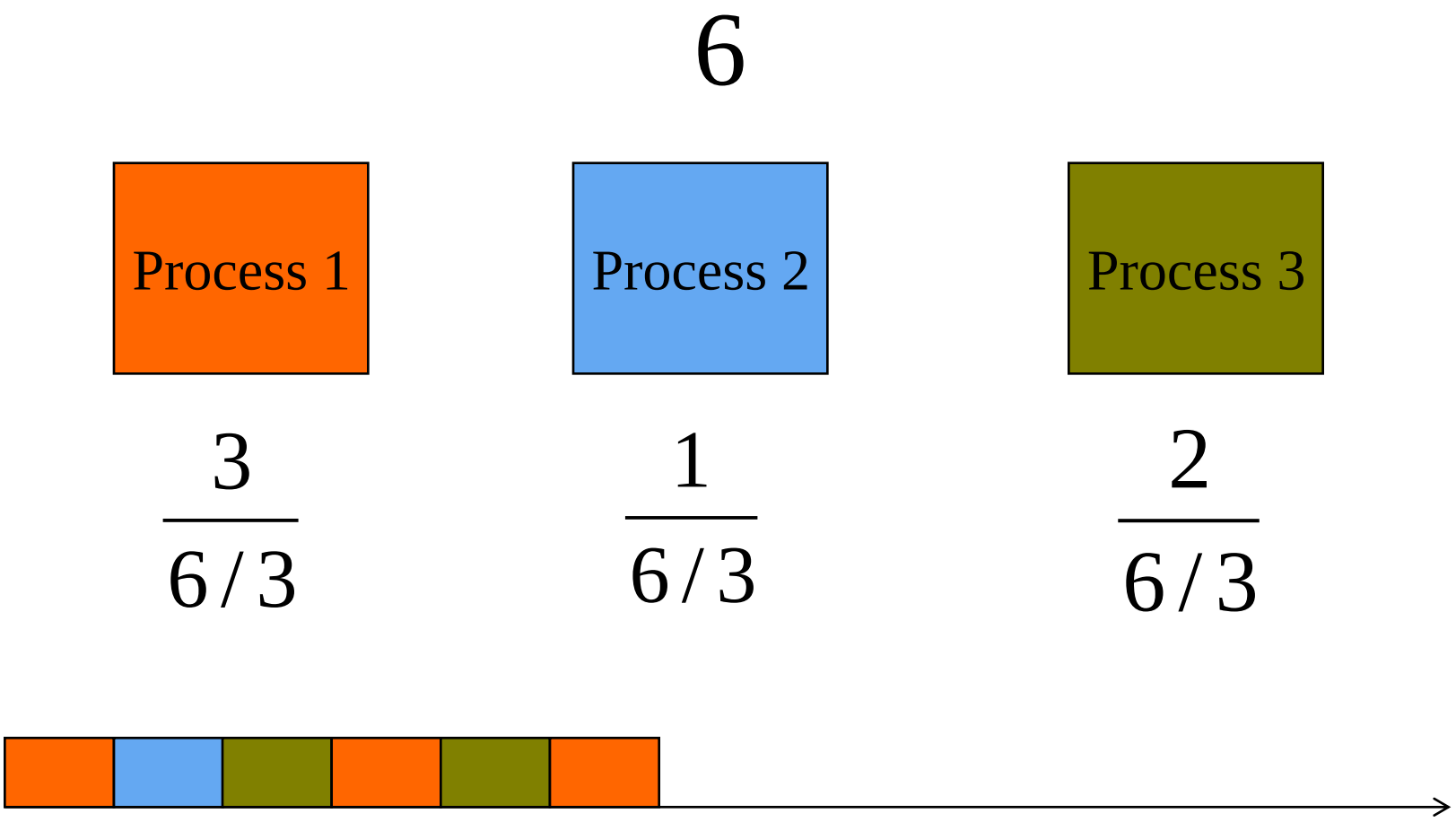
$\frac{2}{5/3}$

Process 3

$\frac{2}{5/3}$



Guaranteed scheduling example



Guaranteed scheduling example

7

Process 1

Process 2

Process 3

$\frac{3}{7/3}$

$\frac{2}{7/3}$

$\frac{2}{7/3}$

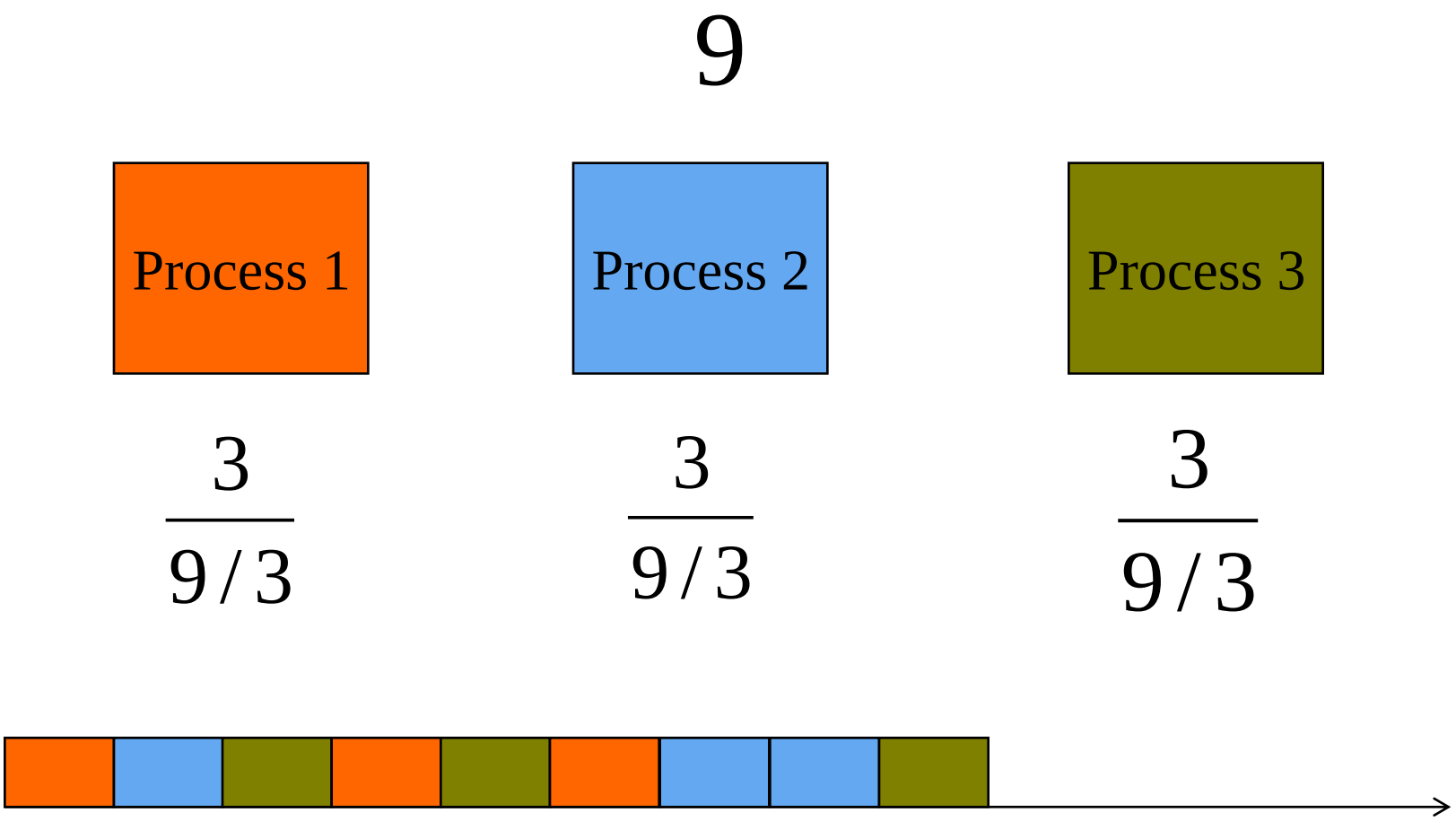
Guaranteed scheduling example

8

Process 1 | Process 2 | Process 3

$\frac{3}{8/3}$ $\frac{3}{8/3}$ $\frac{2}{8/3}$

Guaranteed scheduling example

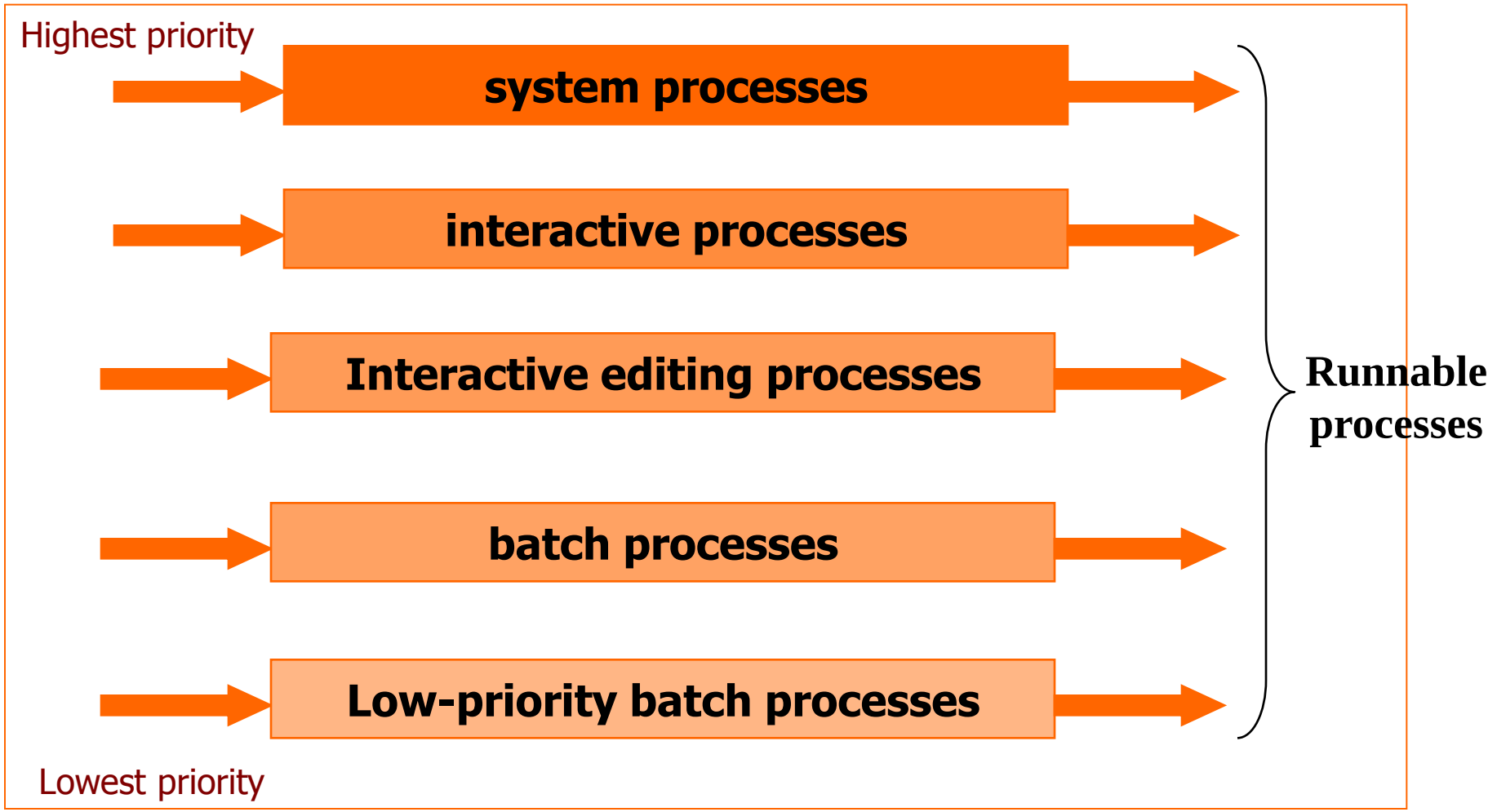


Priority Scheduling

- Select ***runnable*** process with highest priority

- When there are classes of processes with the same priority:
  - each priority group may use round robin scheduling
  - A process from a lower priority group runs only if there is no higher-priority process waiting

Multi-Level Queue Scheduling



Disadvantage?

Dynamic multi-level scheduling

The main drawback of priority scheduling: ***starvation***!

To avoid it:

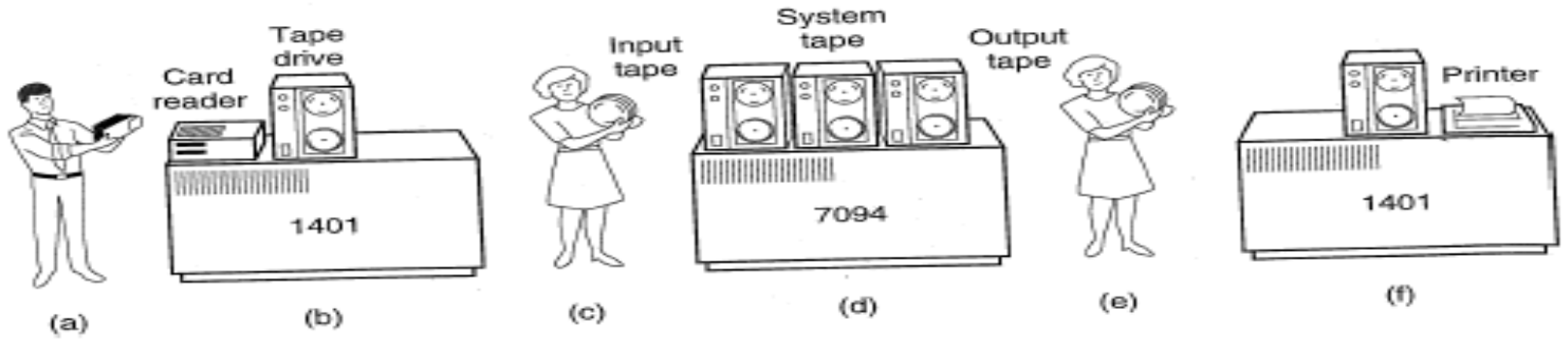
- Prevent high priority processes from running indefinitely by ***changing priorities dynamically***:
  - Each process has a *base* priority
  - increase priorities of waiting processes at each clock tick
- Approximation SJF scheduling
  - increase priorities of I/O bound processes (decrease those of CPU bound processes)

    priority = 1/f *(**f** is the fraction of time-quantum used)*

Parameters defining multi-level scheduling

- Number of queues and scheduling policy in each queue (FCFS, RR,…)
- When to demote a process to a lower queue
- Which queue to insert a process to according to:
  - Elapsed time since process received CPU (aging)
  - Expected CPU burst
  - Process priority

Dynamic multi-level scheduling example: Compatible Time Sharing System (CTSS)



IBM 7094 could hold a SINGLE process in memory

Process switch was VERY expensive…

Compatible Time Sharing System (CTSS)

- Assign time quanta of different lengths to different priority classes
- Highest priority class: 1 quantum, 2nd class: 2 quanta, 3rd class: 4 quanta, etc.
- Move processes that used their quanta to a lower class
- Net result - longer runs for *CPU-bound* processes; higher priority for *I/O-bound* processes
- for a CPU burst of 100 time quanta - 7 switches instead of 100 (in RR)

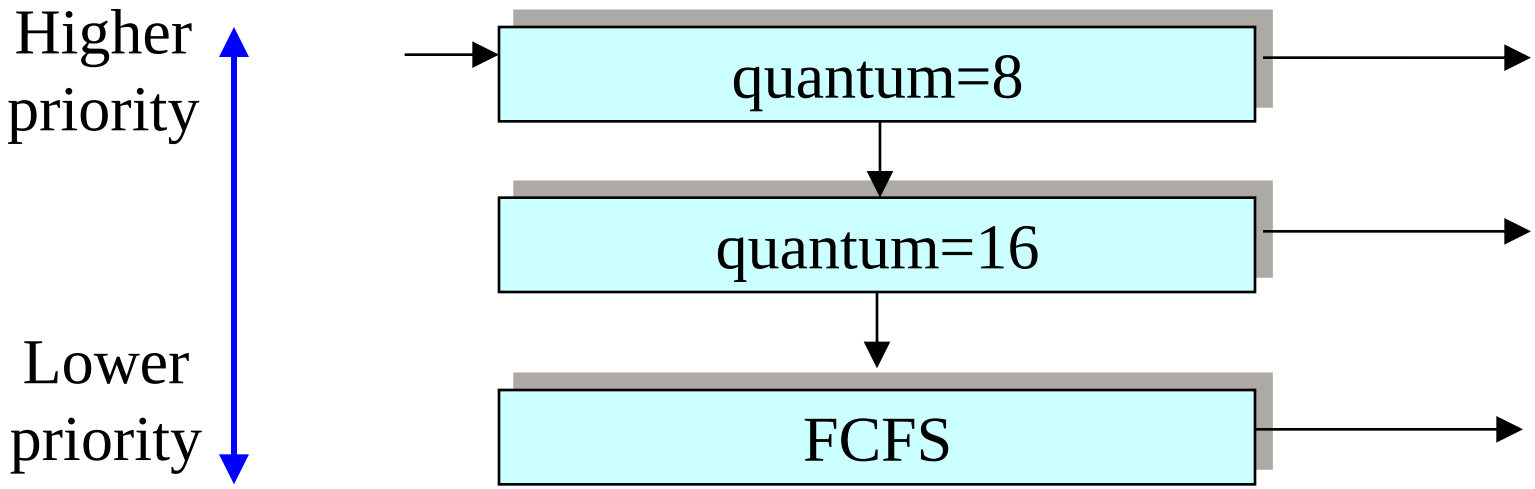
Highest Response Ratio Next (HRRN)

Choose next process with the highest ratio of:

$$\frac{\textbf{time spent waiting + expected service time}}{\textbf{expected service time}}$$

Similar to non-preemptive SJF but avoids starvation

Feedback Scheduling: example

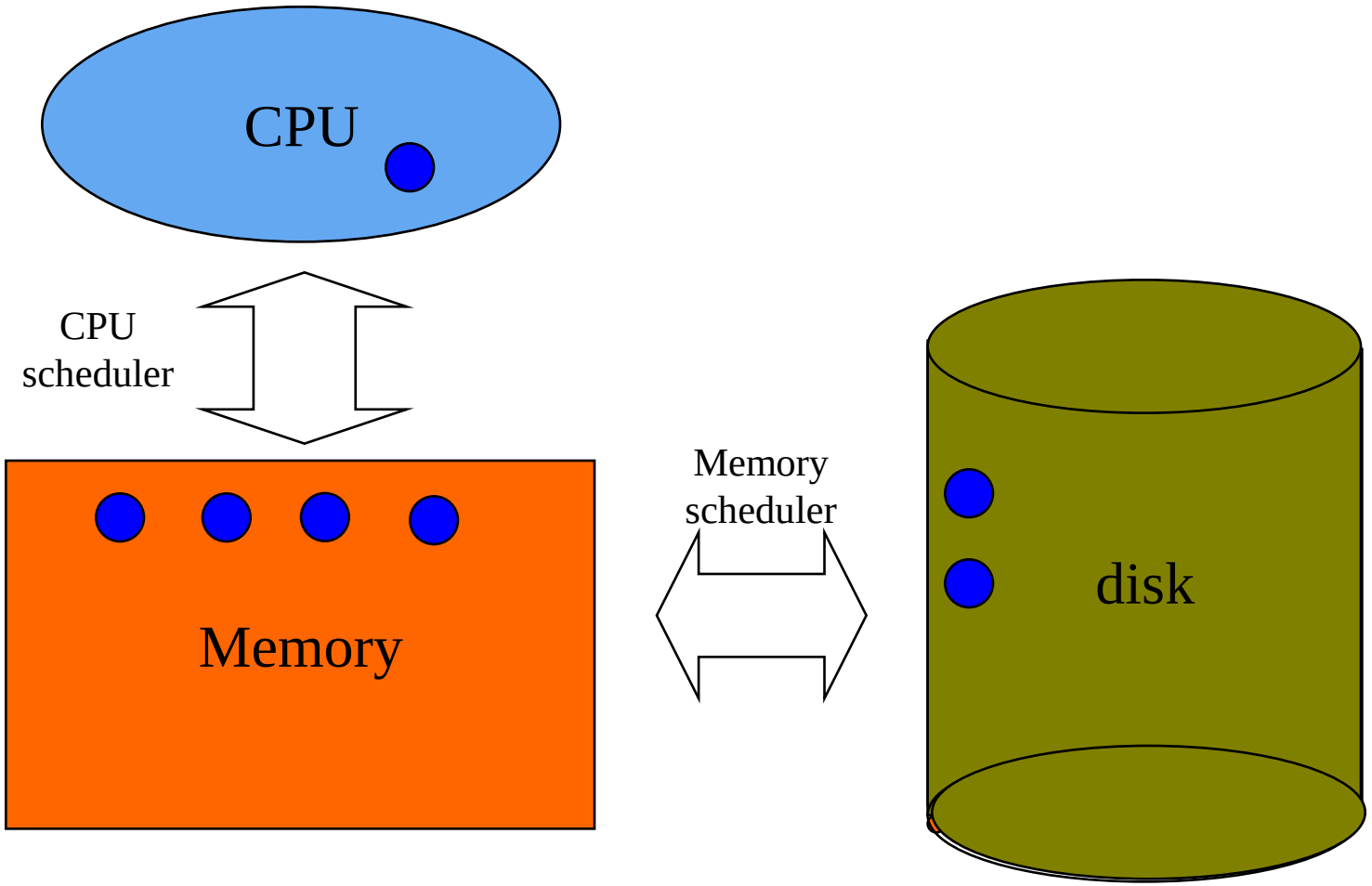


- Demote jobs that have been running longer
- Need not know remaining process execution time
- Always perform a higher-priority job (preemptive)
- Aging may be used to avoid starvation

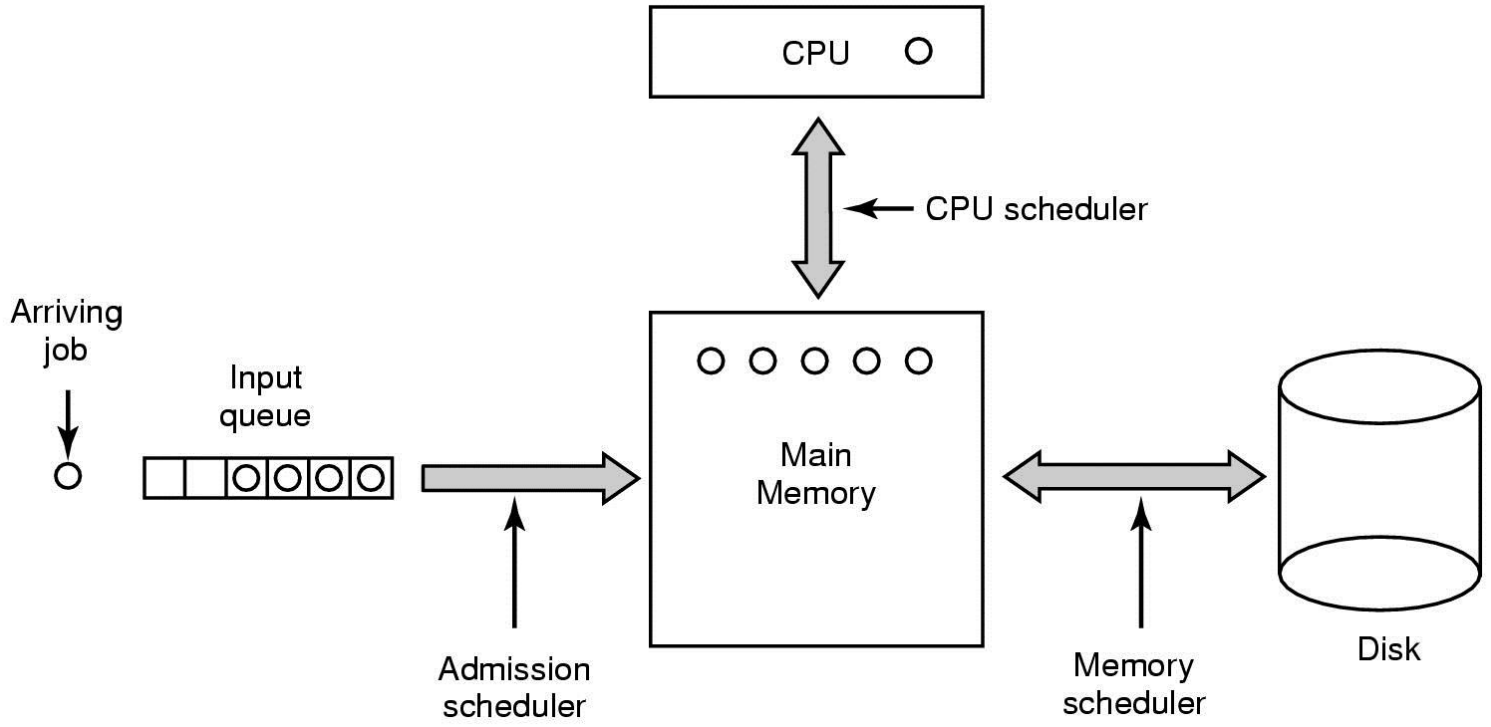
Two-Level Scheduling

- Recall that processes can be either in memory or swapped out (to disk)
- Schedule memory-resident processes by a *CPU (low-level) scheduler* as before
- Swap in/out by a *Memory (high-level) scheduler* using:
  - Elapsed time since process swapped in/out
  - CPU time allocated to process
  - Process memory size
  - Process priority

Two level scheduling



Scheduling in Batch Systems (2)



Three level scheduling

Scheduling in Real-Time Systems

- Must respond to external events within fixed time (CD player, autopilot, robot control,…)

- Schedulable real-time system

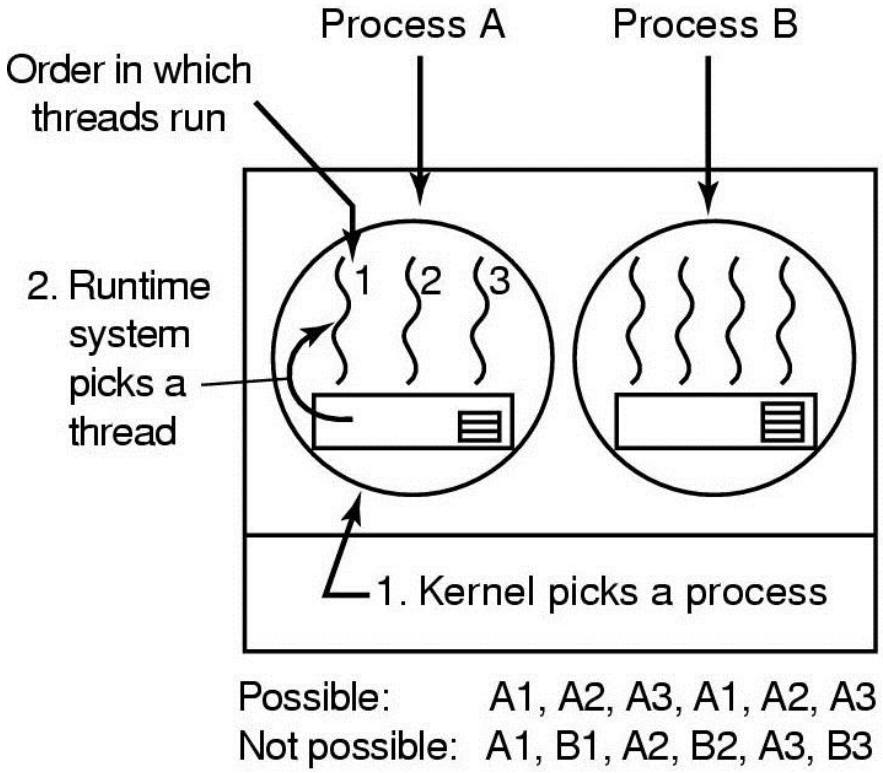
  Given

  - *m* periodic events
  - event *i* occurs with period P_i and requires C_i seconds

- Then the load can only be handled if

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

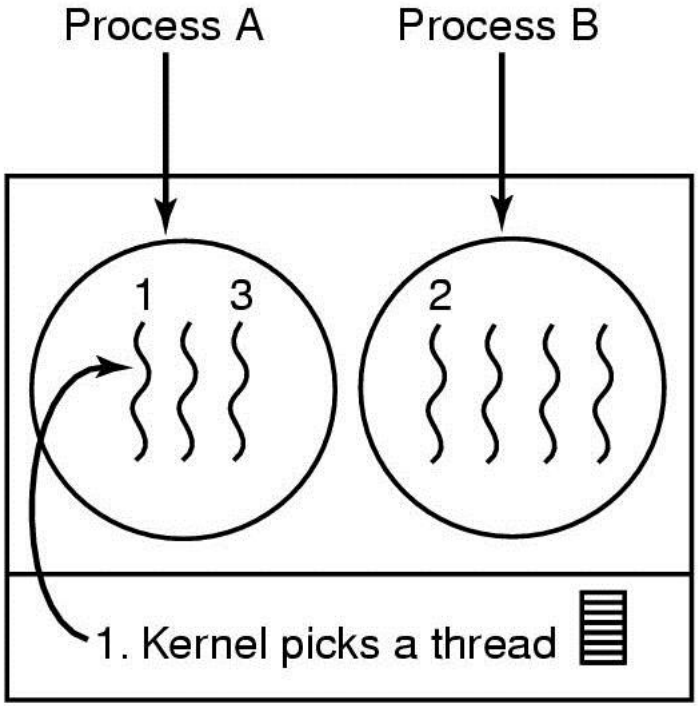
User-level Thread Scheduling



Scheduling within a 60-msec quantum, when threads have 10-msec CPU bursts

- Different scheduling algorithms for threads/processes possible
- No way to preempt user threads
- Thread context switch much faster for user threads
- Application-specific scheduling possible

Kernel-level Thread Scheduling



Possible: A1, A2, A3, A1, A2, A3
Also possible: A1, B1, A2, B2, A3, B3

Scheduling within a 60-msec quantum, when threads have 10-msec CPU bursts

- Scheduler may prefer switches within same process
- Context switch more expensive
- A blocking thread does not block all process threads

Scheduling: outline

- Scheduling criteria
- Scheduling algorithms
- Unix scheduling
- Linux Scheduling
- Win NT scheduling

Scheduling in Unix

- Two-level scheduling
- Low level (CPU) scheduler uses multiple queues to select the next process, ***out of the processes in memory***, to get a time quantum.
- High level (memory) scheduler moves processes from memory to disk and back, to enable all processes their share of CPU time
- Low-level scheduler keeps queues for each priority
- Processes in user mode have positive priorities
- Processes in kernel mode have negative priorities (lower is higher)

Unix priority queues

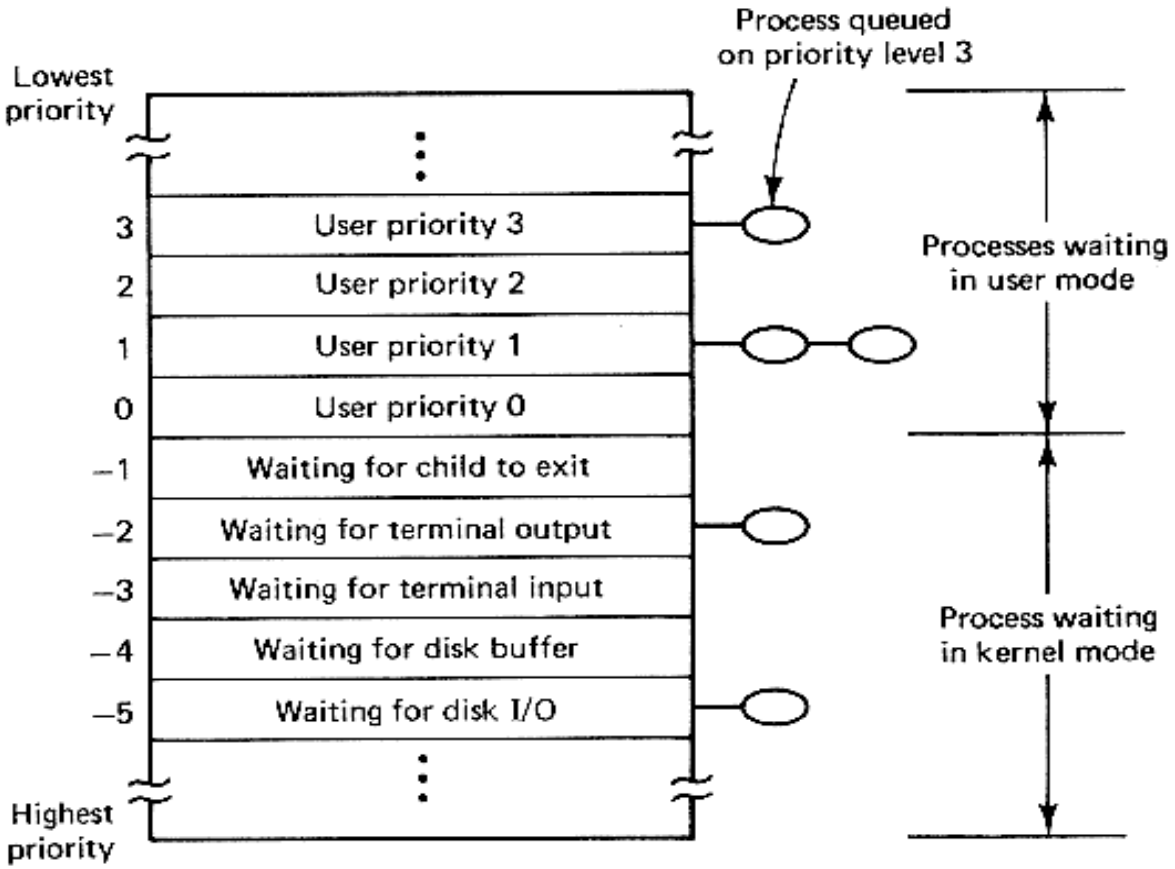


Fig. 7-16. The UNIX scheduler is based on a multilevel queue structure.

Unix low-level Scheduling Algorithm

- Pick process from highest (non-empty) priority queue
- Run for 1 quantum (usually 100 ms.), or until it blocks
- Increment CPU usage count every clock tick
- Every second, ***recalculate*** priorities:
  - Divide cpu usage by 2
  - New priority = base + cpu_usage + nice
  - Base is negative if the process is released from waiting in kernel mode
- Use round robin for each queue (separately)

Unix low-level scheduling Algorithm - I/O

- Blocked processes are removed from queue, but when the blocking event occurs, are placed in a high priority queue
- The negative priorities are meant to release processes quickly from the kernel
- Negative priorities are hardwired in the system, for example, -5 for Disk I/O is meant to give high priority to a process released from disk I/O
- Interactive processes get good service, CPU bound processes get whatever service is left...

Priority Calculation in Unix

$$P_j(i) = Base_j + \frac{CPU_j(i-1)}{2} + \frac{GCPU_k(i-1)}{4 \times W_k}$$

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

$$GCPU_k(i-1) = \frac{GU_k(i-1)}{2} + \frac{GCPU_k(i-1)}{2}$$

$P_j(i)$ = Priority of process *j* at beginning of interval *i*

$Base_j$ = Base priority of process *j*

$U_j(i)$ = Processor utilization of process *j* in interval *i*

$Gu_k(i)$ = Total processor utilization of all processes in group *k* during interval *i*

$CPU_j(i)$ = Exponentially weighted average processor utilization by process *j* through interval *i*

GCPUk(I)= Exponentially weighted average total processor utilization of group *k* through interval *i*

W_k = Weighting assigned to group *k*, with the constraint that $0 \leq W_k \leq 1$ and $\sum_k w_k = 1$

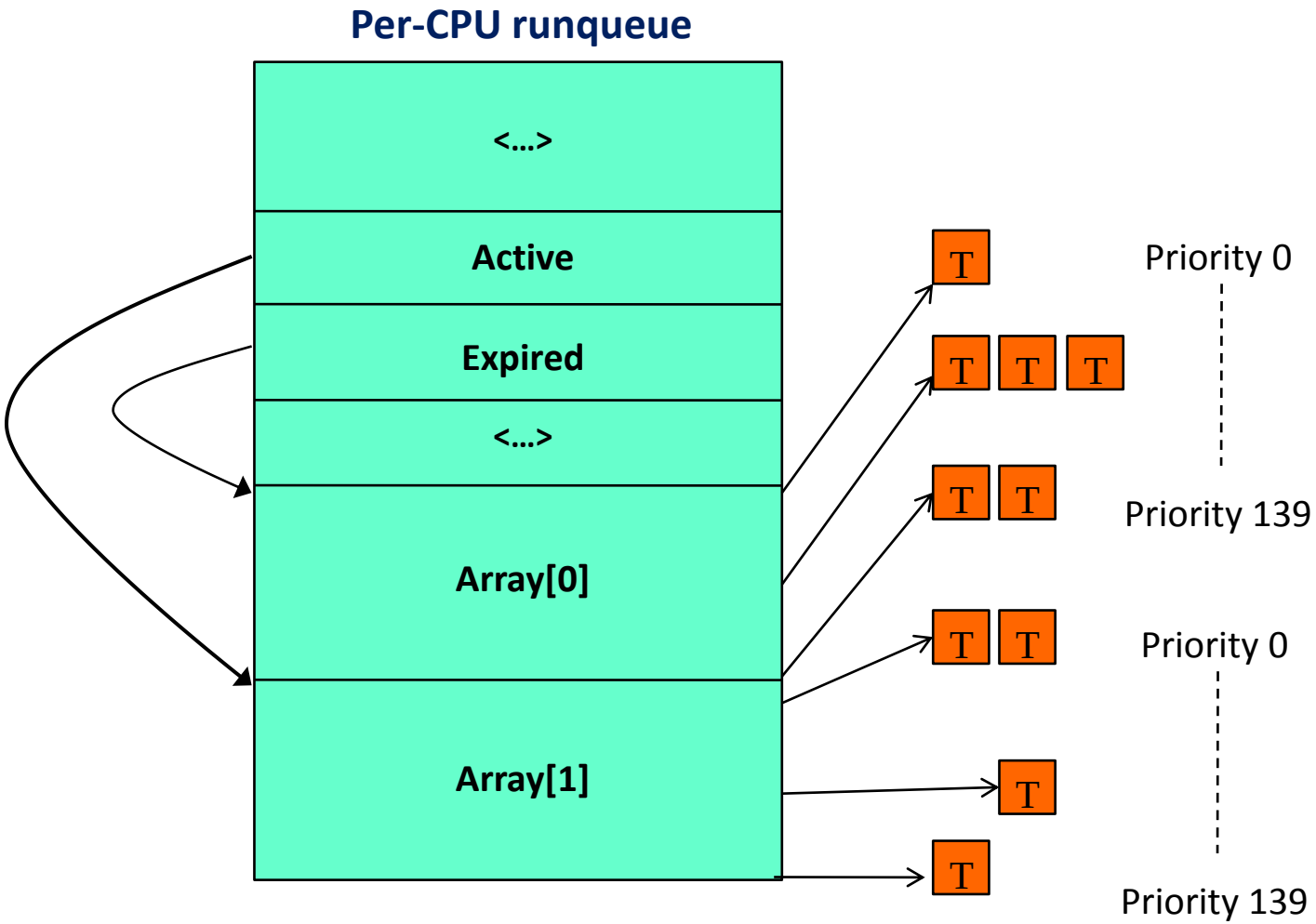
Scheduling: outline

- Scheduling criteria
- Scheduling algorithms
- Unix scheduling
- Linux Scheduling
- Win NT scheduling

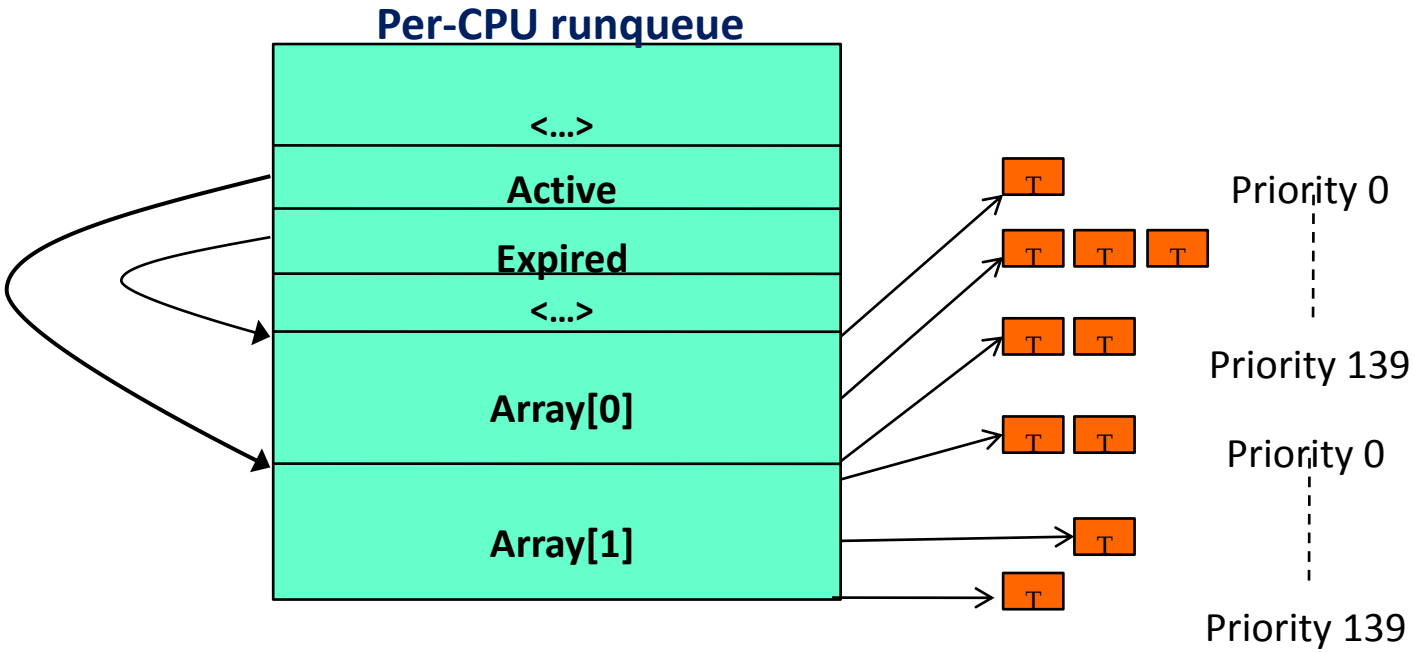
Three classes of threads

- Real-time FIFO
  - Have highest priority, can only be preempted by a higher-priority real-time FIFO thread
- Real-time round robin
  - Assigned time quantum
- Timesharing
  - Priorities 100-139
- Priority determines the number of clock ticks (**jiffy**s) assigned to a round-robin or timesharing thread

Linux runqueues



Linux runqueues (cont'd)



- Scheduler selects tasks from highest-priority active queue
  - If time-slice expires – moved to an expired list
  - If blocked then, when event arrives, returned to active queue (with smaller time-slice)
- When no more active tasks – swap active and expired pointers

Multiprocessor support

- Runqueue per processor
- Affinity scheduling
- Perform periodic load-balancing between runqueues
  - But in most cases, a process will run on the same process it ran before.

Scheduling: outline

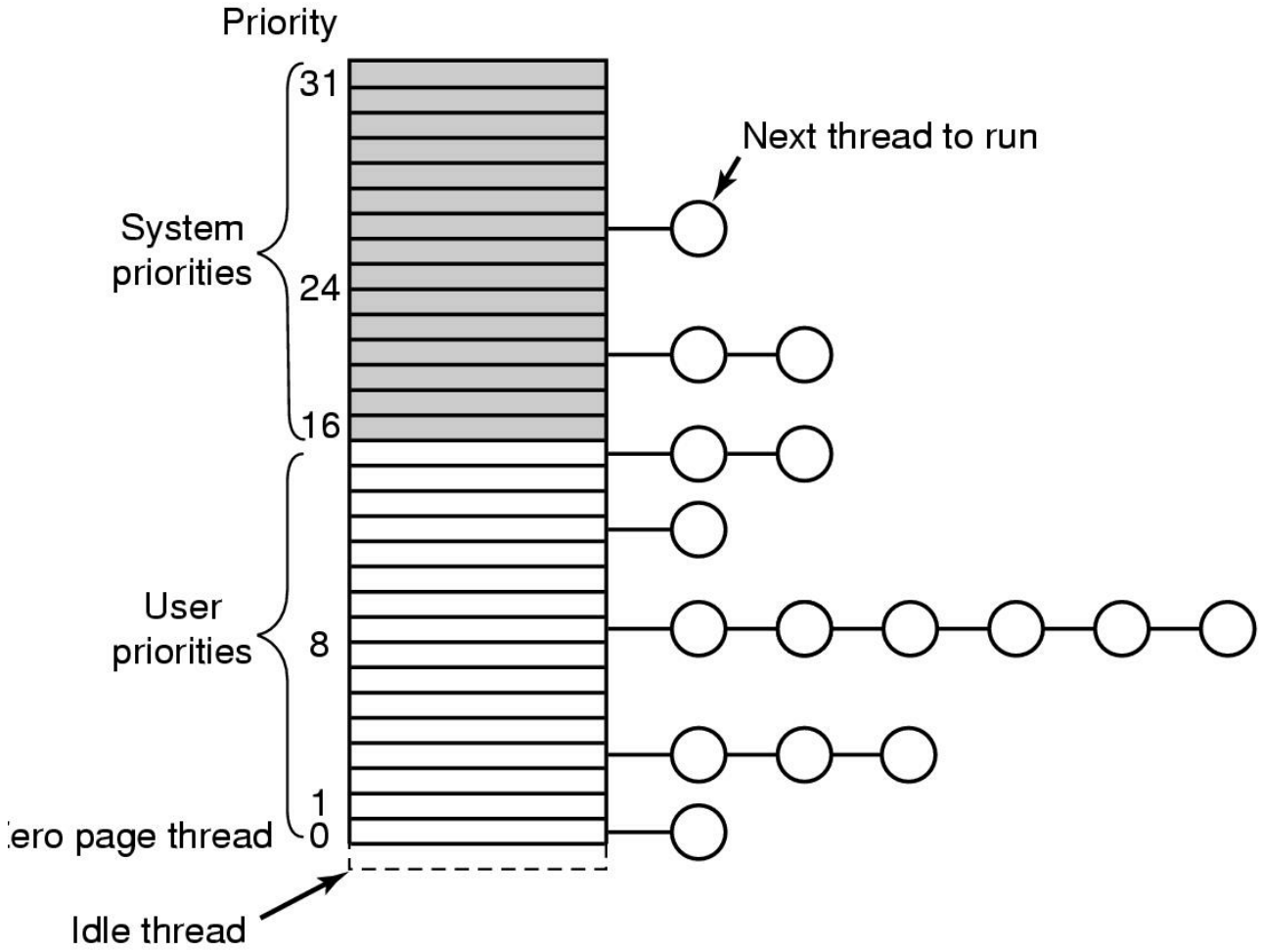
- Scheduling criteria
- Scheduling algorithms
- Unix scheduling
- Linux Scheduling
- Win NT scheduling

WINDOWS NT Scheduling

		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

Mapping of Win32 priorities to Windows 2000 priorities

WINDOWS NT Scheduling (II)



Windows 2000 supports 32 priorities for threads

Windows NT scheduling: dynamic changes

- Priority is raised:
  - Upon I/O completion (disk: 1, serial line: 2, keyboard: 6, sound card: 8)
  - Upon event reception (semaphore, mutex…): by 2 if in the foreground process, by 1 otherwise
  - If a thread waits `too much' – it is moved to priority 15 for two quanta (for solving priority inversion – see next slide)
- Priority is decreased:
  - By 1 if quantum is fully used
  - Never bellow base value (normal thread priority)

An example of priority inversion