EC 440 – Introduction to Operating Systems

Manuel Egele

Department of Electrical & Computer Engineering

Boston University

Scheduling

Problem: Many processes to execute, but only one CPU

OS time-multiplexes the CPU by *context switching*

- Between user processes
- Between user processes and the operating system

Operation carried out by the *scheduler* that implements a *scheduling algorithm*

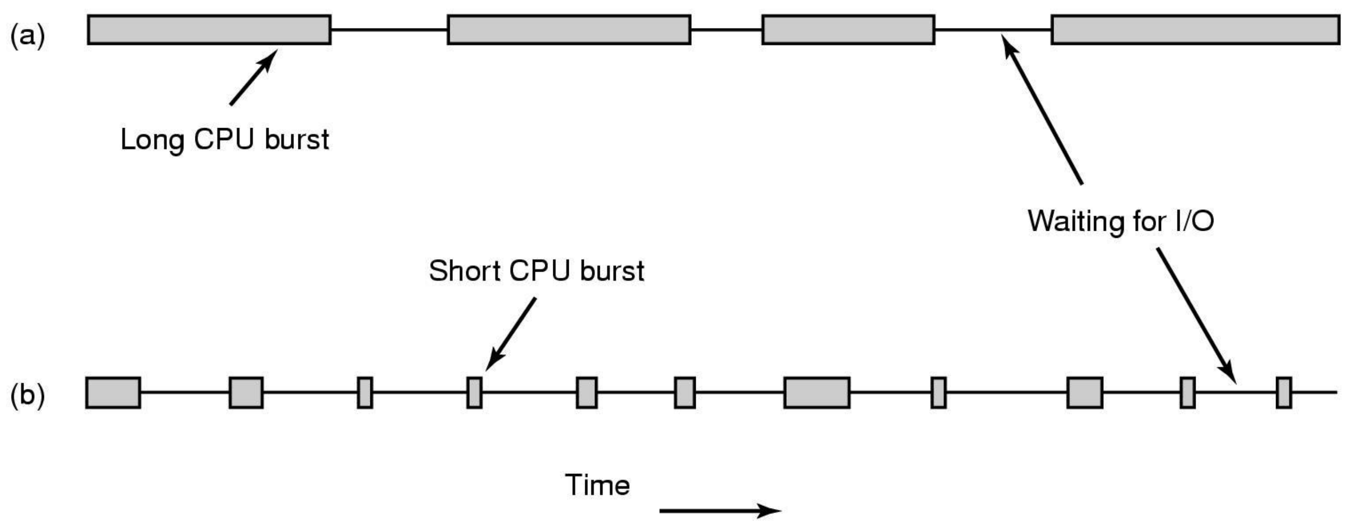
Switching is expensive

- Switch from user to kernel model
- Save the state of the current process (including memory map)
- Select a process for execution (scheduler)
- Restore the saved state of the new process

CPU-bound and I/O-bound Processes

Bursts of CPU usage alternate with periods of I/O wait

- a CPU-bound process (a)
- an I/O bound process (b)



When To Schedule

Must schedule

- a process exits
- a process blocks (I/O, semaphore, etc.)

May schedule

- new process is created (parent and child are both ready)
- I/O interrupt
- clock interrupt

Scheduling Algorithms

Non-preemptive

- CPU is switched when process
  - has finished
  - executes a yield()
  - blocks

Preemptive

- CPU is switched independently of the process behavior
  - A clock interrupt is required

Scheduling algorithms should enforce

- Fairness
- Policy
- Balance

Scheduler Goals

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

Proportionality

Source: xkcd

Scheduling in Batch Systems

Goals

- Throughput:
  maximize jobs per hour
- Turnaround time:
  minimize time between submission and termination
- CPU utilization:
  keep processor busy

Examples

- First-come first-served
- Shortest job first
- Shortest remaining time next

Tradeoffs

Improving on one metric can hurt another

For example:

- We want to improve *throughput,* so we decide to only schedule short jobs
- But now longer jobs never get run, so their *turnaround time* is effectively infinite

First-Come First-Served

- Processes are inserted in a queue
- The scheduler picks up the first process, executes it to termination or until it blocks, and then picks the next one

Advantage:

- Very simple

Disadvantage:

- I/O-bound processes could be slowed down by CPU-bound ones

Analyzing First Come First Served

Turnaround time depends on order we pick jobs

Assuming jobs arrive at time 0:

A (32 mins)	B (5 mins)	C (5 mins)

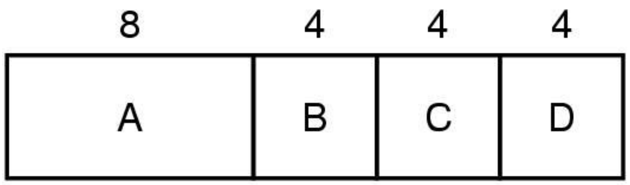
Turnaround time: (32 + 37 + 42) / 3 = 37 minutes

B (5 mins)	C (5 mins)	A (32 mins)

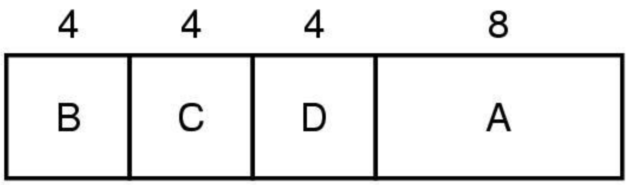
Turnaround time: (5 + 10 + 42) / 3 = 19 minutes

Shortest Job First

- This algorithm assumes that running time for all the processes to be run is known in advance
- Scheduler picks the shortest job first
- Optimizes turnaround time
  - a) Turnaround is A=8, B=12, C=16, D=20 (avg. 14)
  - b) Turnaround is B=4, C=8, D=12, A=20 (avg. 11)
- Problem: what if new jobs arrive?



(a)



(b)

Counterexample

- The optimality proof only applies when all jobs are available at time 0
- Suppose we have instead:

Available at minute 0		Available at minute 3		
A (2 mins)	B (4 mins)	C (1 mins)	D (1 mins)	E (1 mins)

- Then turnaround time is

  (2 + 6 + (7 -3) + (8 -3) + (9-3))/5 = 4.6
- But if we run them in order, B, C, D, E, A, turnaround time is:

  (4 + (5-3) + (6-3) + (7-3) + 9)/5 = 4.4

Shortest Remaining Time Next

- This algorithm also assumes that running time for all the processes to be run is known in advance
- The algorithm chooses the process whose remaining run time is the shortest
- When a new job arrives, its remaining run time is compared to the one of the currently running process
- If current process has more remaining time than the run time of new process, the current process is preempted and the new one is run

Scheduling in Interactive Systems

Goals

- Response time:
  minimize time needed to react to requests
- Proportionality:
  meet user expectations

Examples

- Round robin
- Priority scheduling
- Lottery scheduling

Scheduling in Interactive Systems

- In an interactive system, scheduling algorithms are generally *preemptive*

- Time is divided up into slices called *quanta*

- Each process runs for 1 *quantum* and then the scheduler runs again

Round Robin Scheduling

- Simple algorithm
- Run first process until its quantum is used up
- Move that process to the end and run the next process until its quantum is used up
- Simple, fair

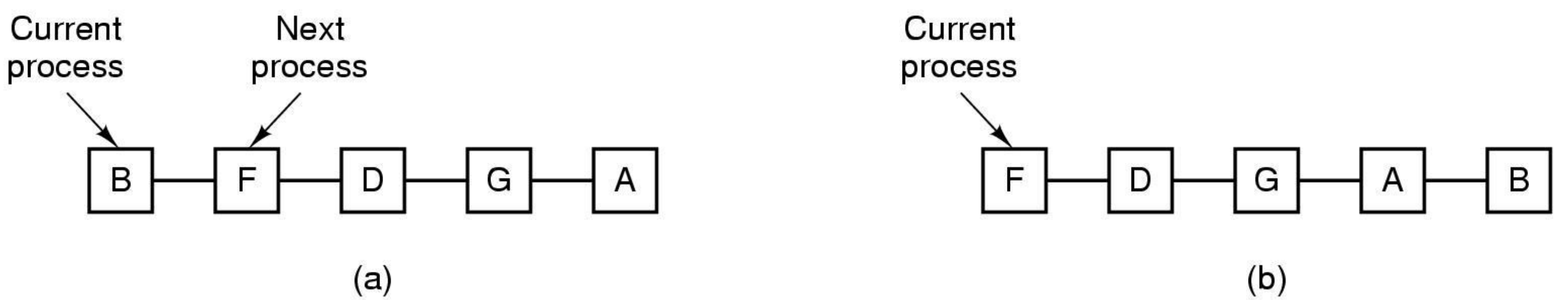
Round Robin Scheduling

Each is process is assigned a *quantum*

The process

- Suspends before the end of the quantum or
- Is preempted at the end of the quantum

Scheduler maintains a list of ready processes



Round Robin Scheduling

Parameters:

- Context switch (e.g., 1 msec)
- Quantum length (e.g., 25 msec)

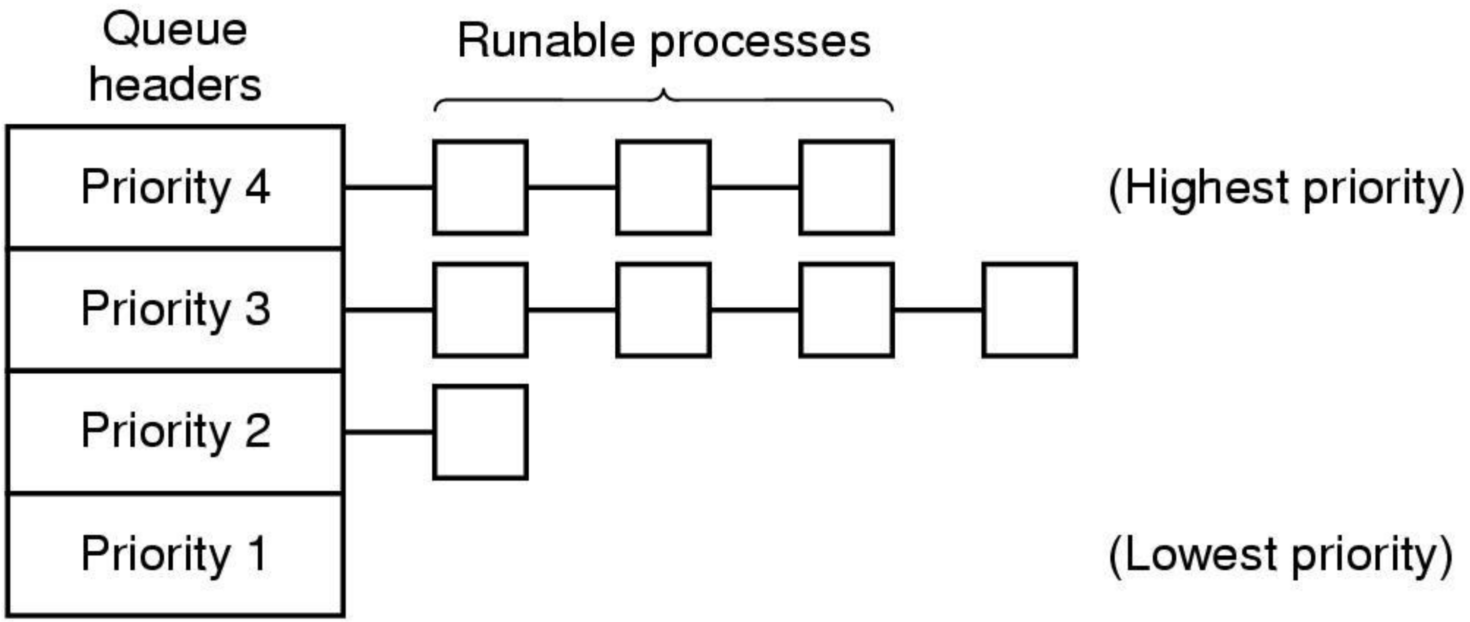
If the quantum is …

- too small, a notable percentage of the CPU time is spent in switching contexts
  (e.g., cs=1, ql=4, 20% of time just for switching)
- too big, response time can be very bad
  (e.g., cs=1, ql=100, up to 5 seconds wait for a ready process with 50 ready processes in the system)

Priority Scheduling

- Each process is assigned a priority
- The process with the highest priority is allowed to run
- I/O bound processes should be given higher priorities
- Problem: low priority processes may end up *starving*...
- First solution: As the process uses CPU, the corresponding priority is decreased
- Second solution: Set priority as the inverse of the fraction of quantum used
- Third solution: Use priority classes (starvation is still possible)

Priority Scheduling



Scheduling: Example

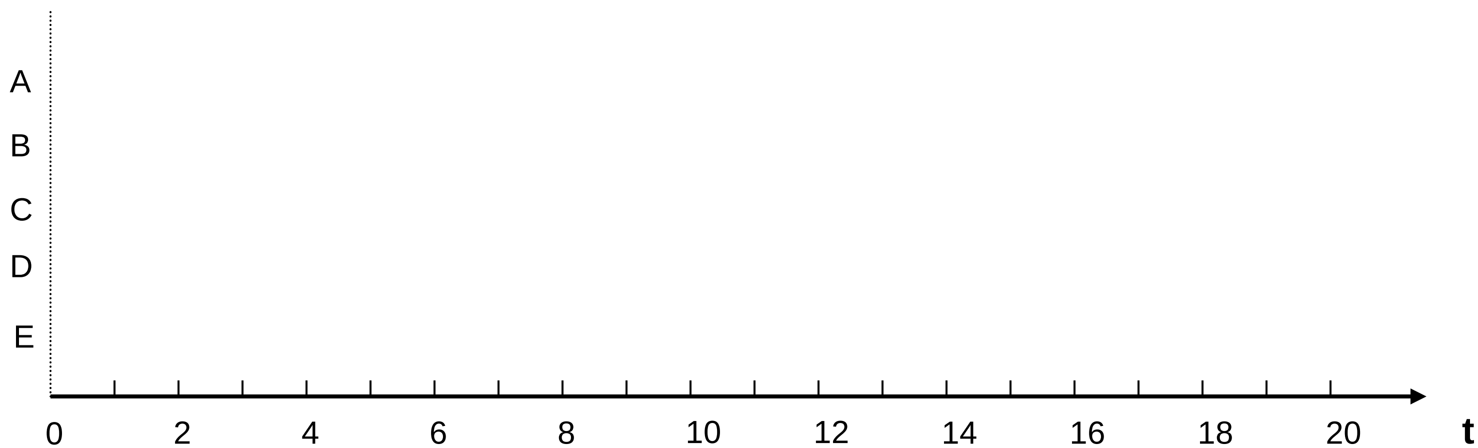
non-preemtive priority scheduling

Process	Start	Runtime	Priority
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3

Scheduling

non-preemtive priority scheduling

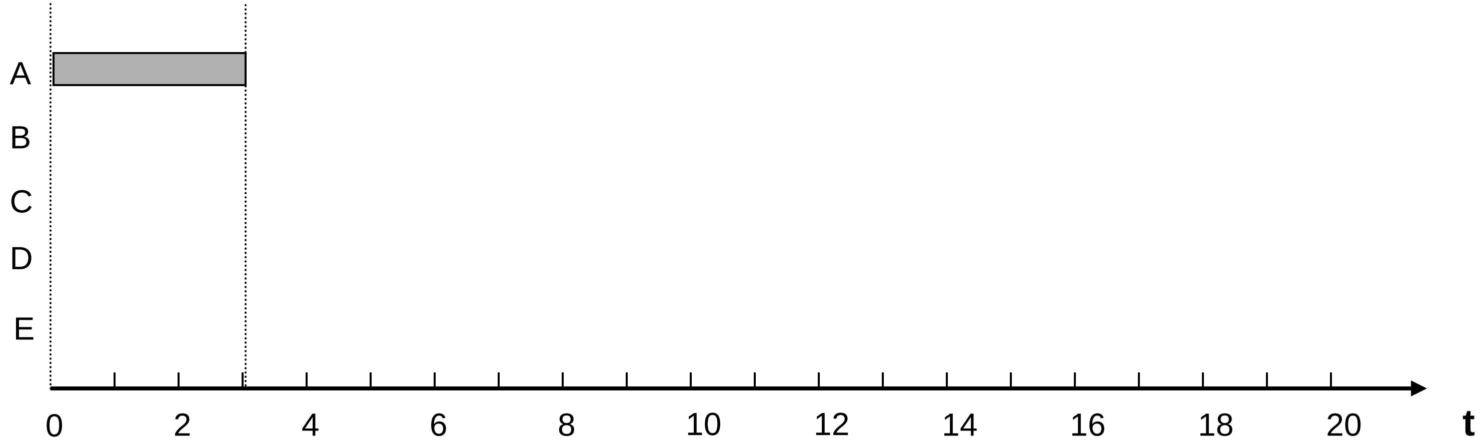
Process	Start	Runtime	Priority
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



Scheduling

non-preemtive priority scheduling

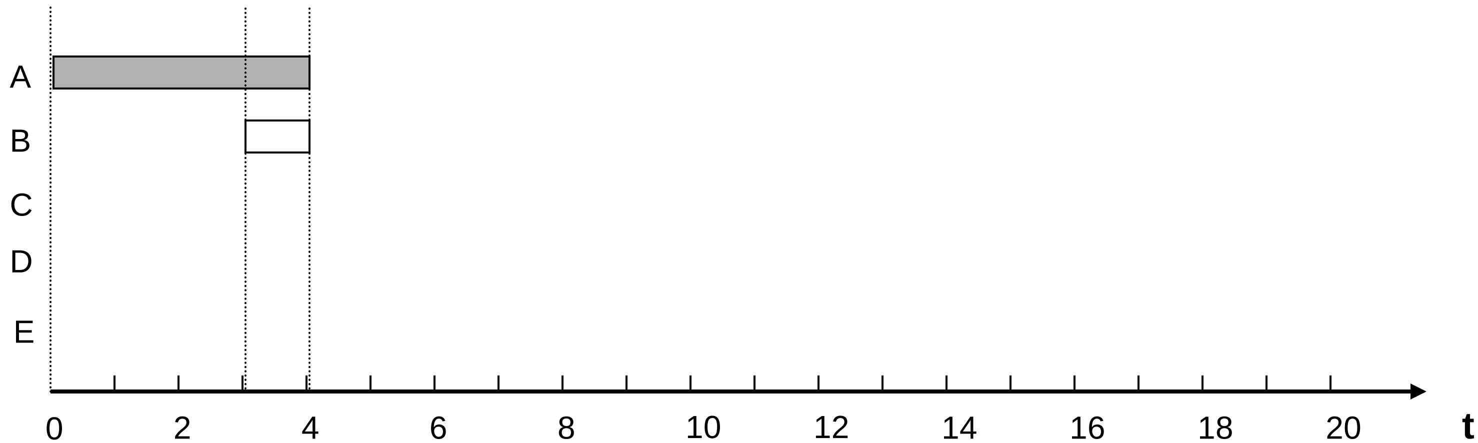
Process	Start	Runtime	Priority
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



Scheduling

non-preemtive priority scheduling

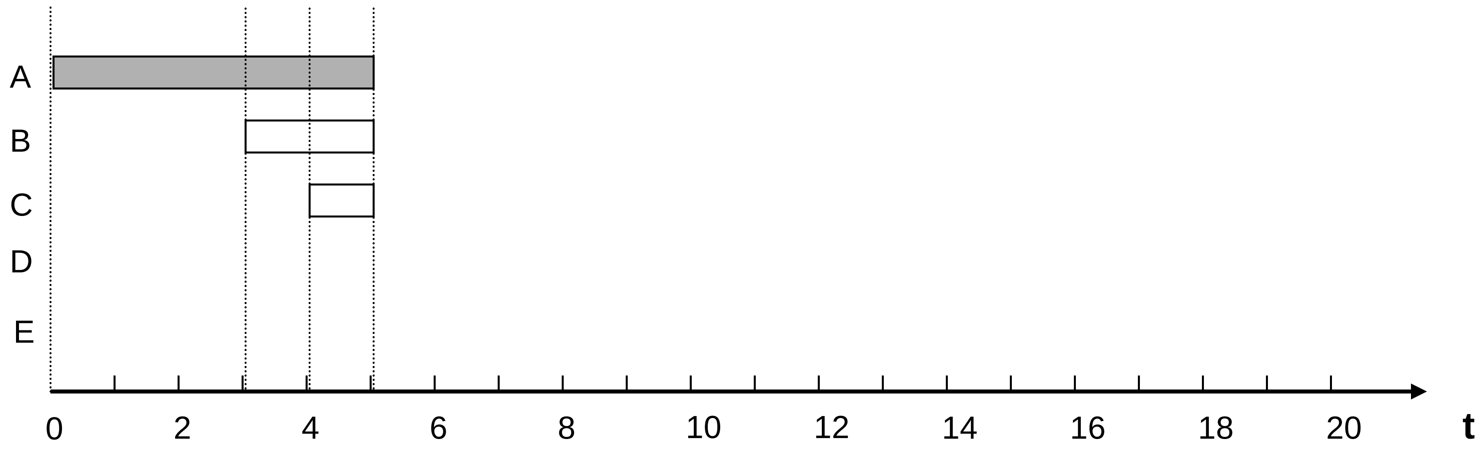
Process	Start	Runtime	Priority
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



Scheduling

non-preemtive priority scheduling

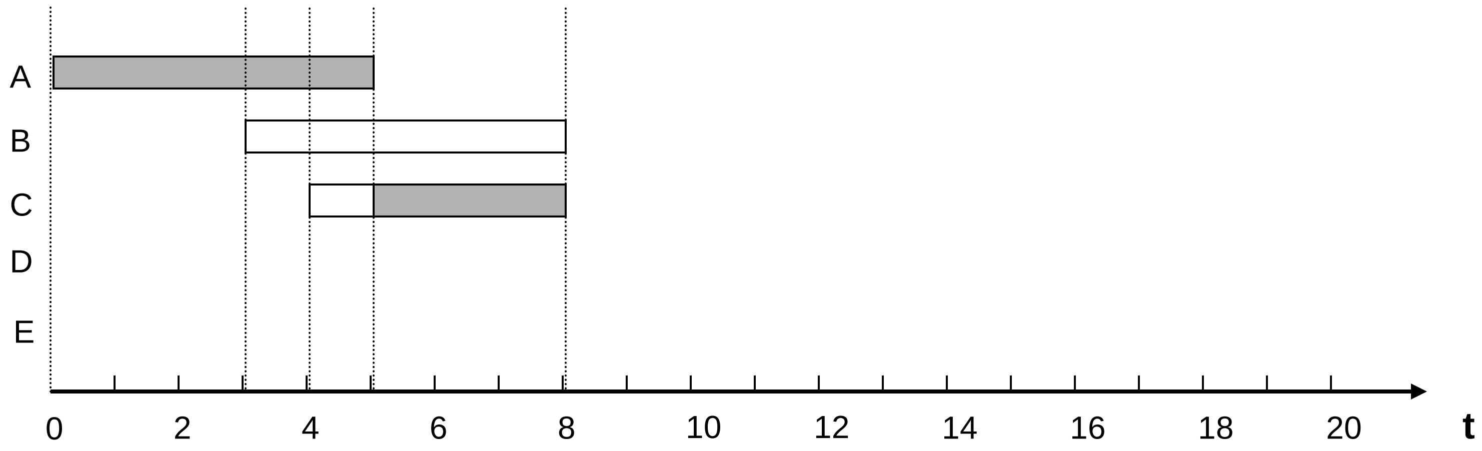
Process	Start	Runtime	Priority
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



Scheduling

non-preemtive priority scheduling

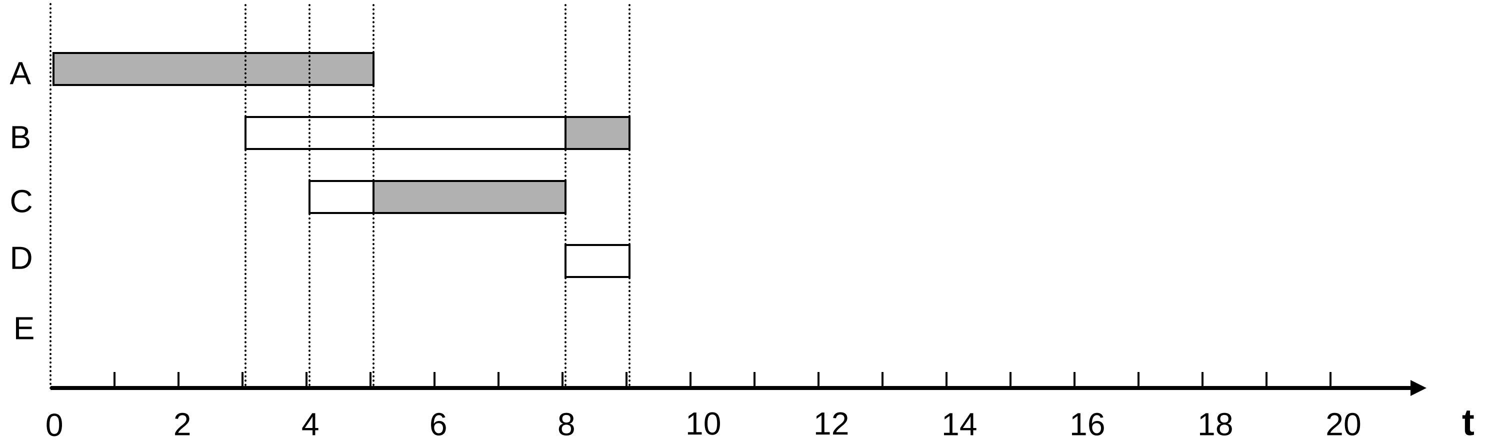
Process	Start	Runtime	Priority
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



Scheduling

non-preemtive priority scheduling

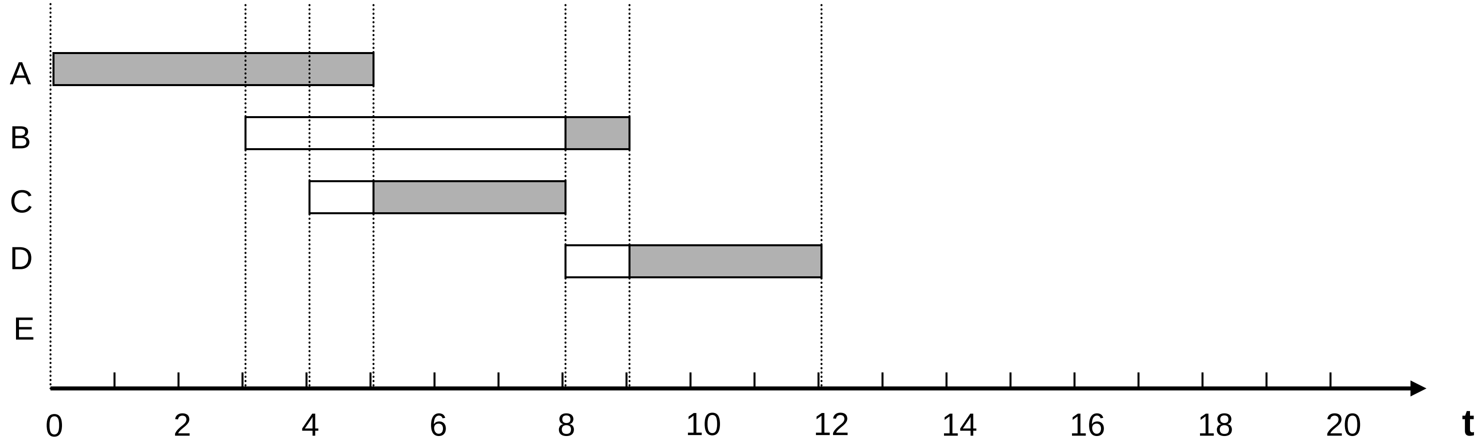
Process	Start	Runtime	Priority
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



Scheduling

non-preemtive priority scheduling

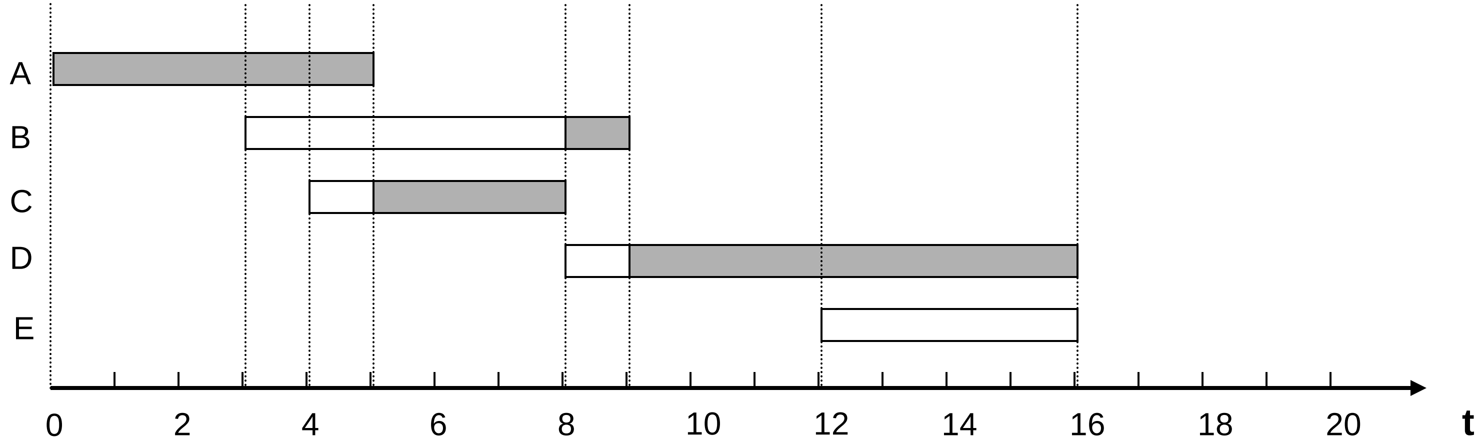
Process	Start	Runtime	Priority
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



Scheduling

non-preemtive priority scheduling

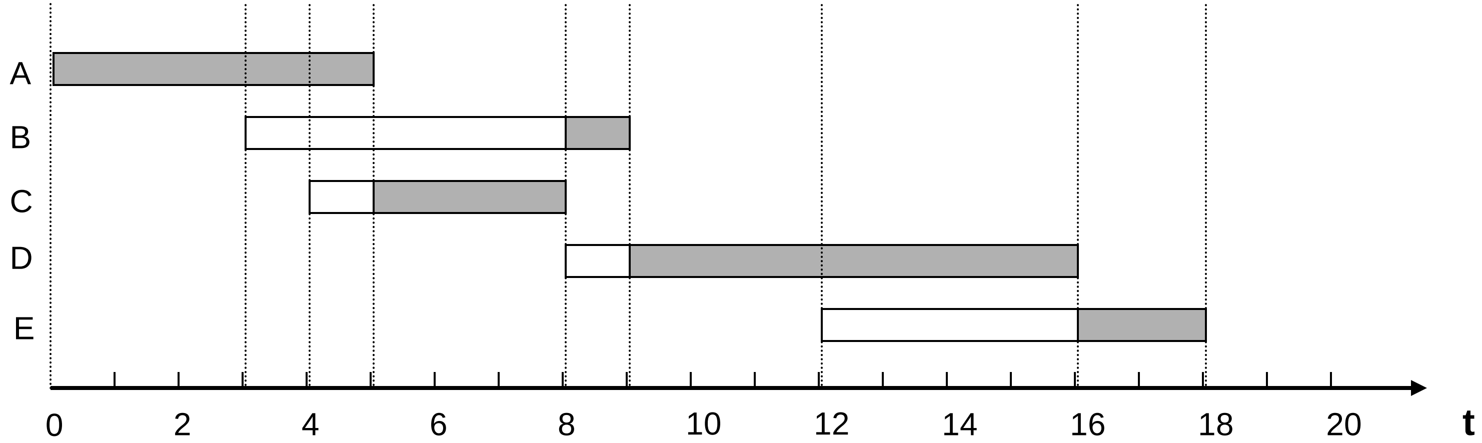
Process	Start	Runtime	Priority
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



Scheduling

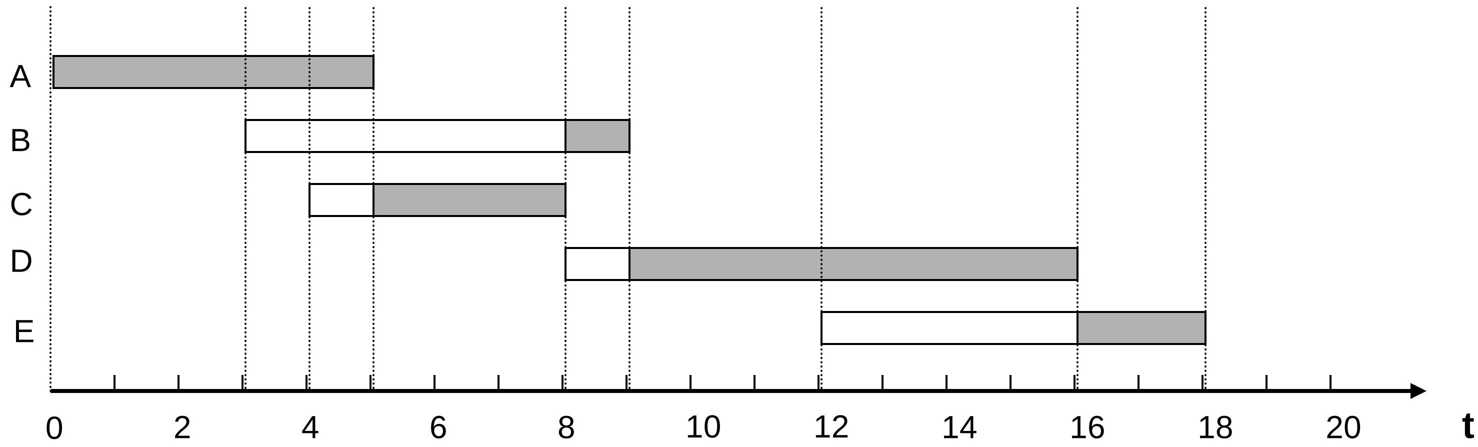
non-preemtive priority scheduling

Process	Start	Runtime	Priority
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



Scheduling

non-preemtive priority scheduling



Process	A	C	B	D	E
Time (RUNNING)	0	5	8	9	16

Scheduling: Example

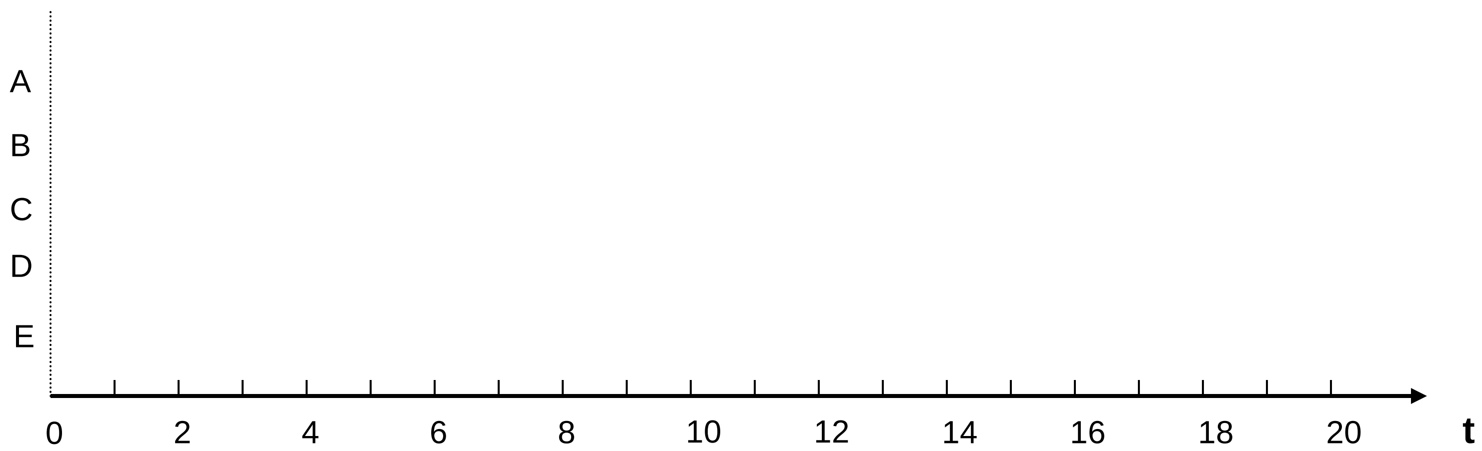
preemtive priority scheduling

Process	Start	Runtime	Priority
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3

Scheduling

preemtive priority scheduling

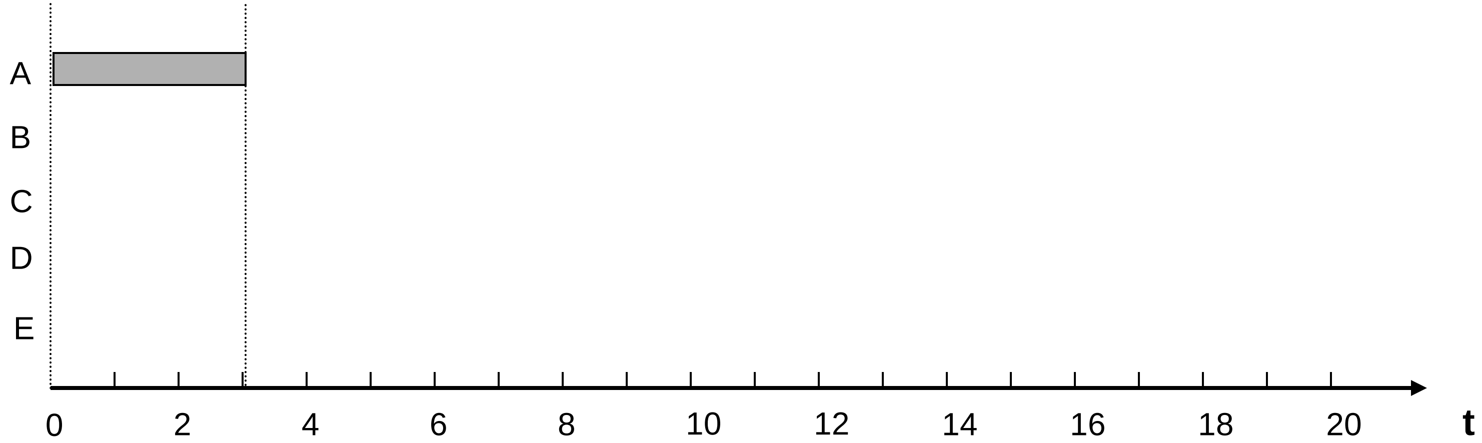
Process	Start	Runtime	Priority
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



Scheduling

preemtive priority scheduling

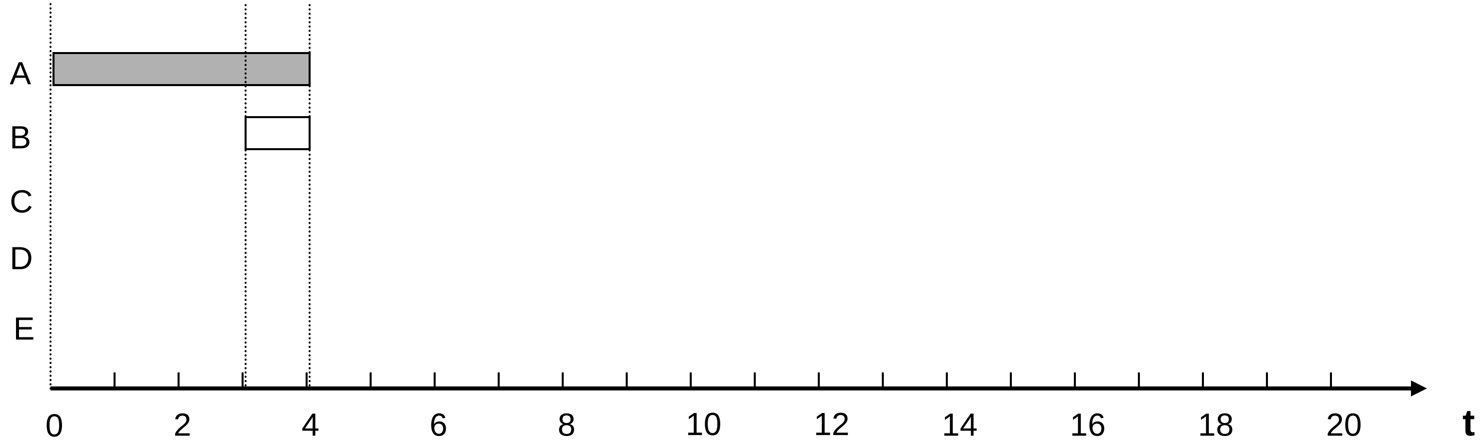
Process	Start	Runtime	Priority
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



Scheduling

preemtive priority scheduling

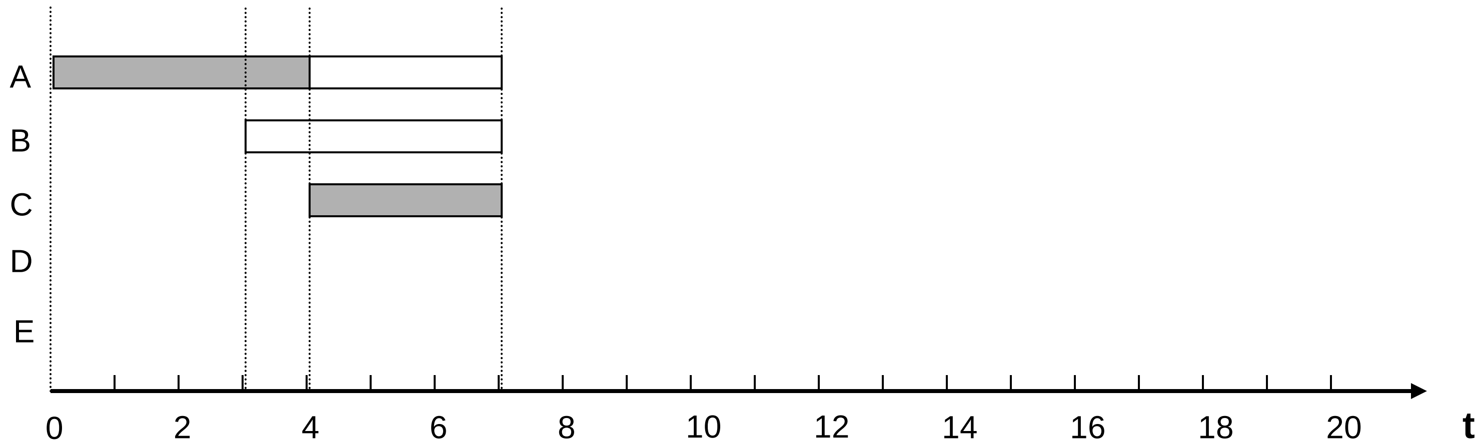
Process	Start	Runtime	Priority
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



Scheduling

preemtive priority scheduling

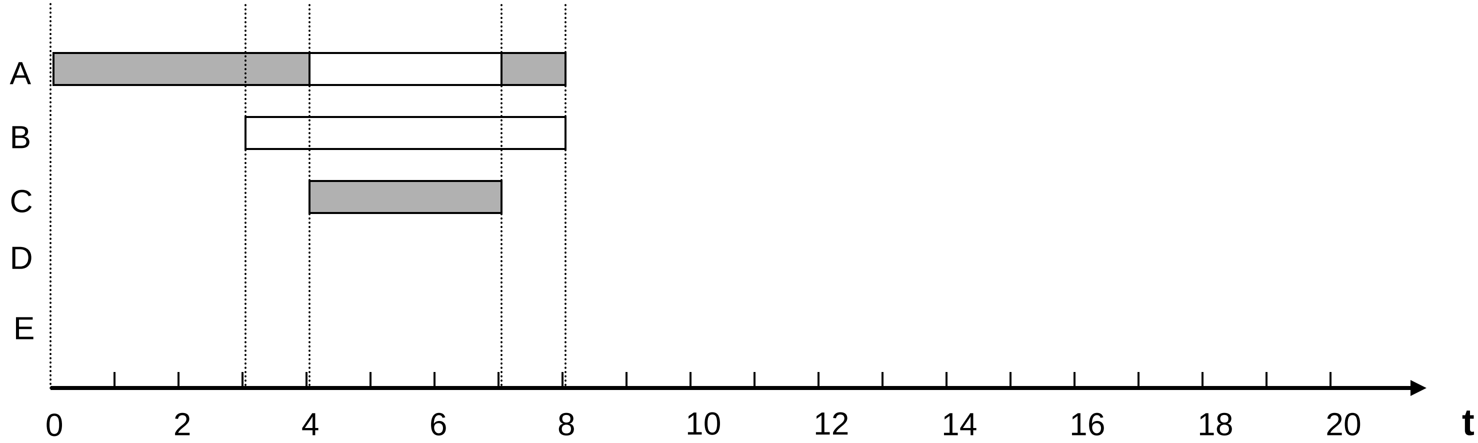
Process	Start	Runtime	Priority
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



Scheduling

preemtive priority scheduling

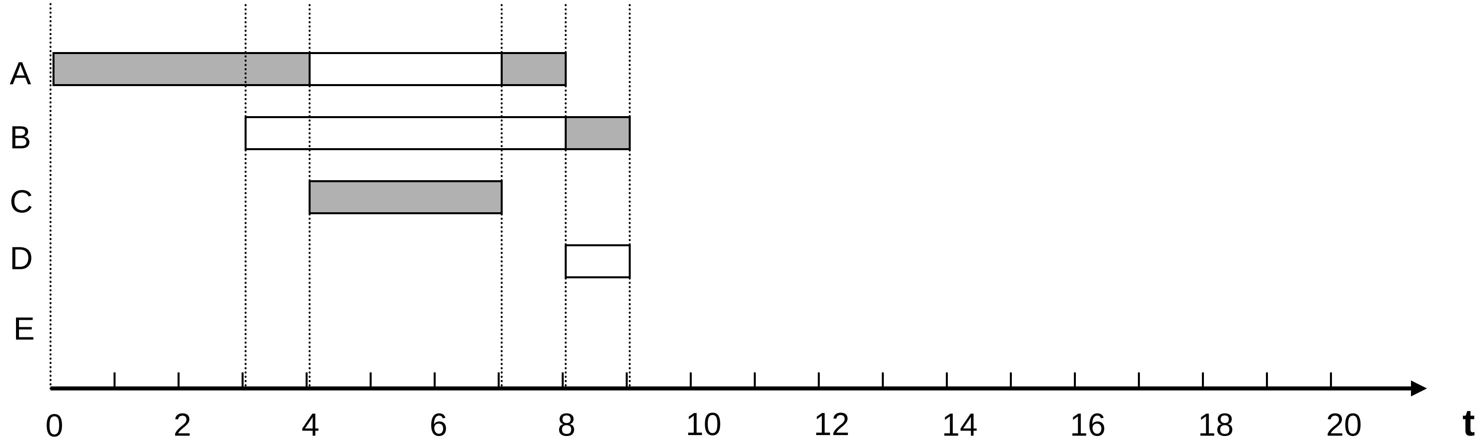
Process	Start	Runtime	Priority
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



Scheduling

preemtive priority scheduling

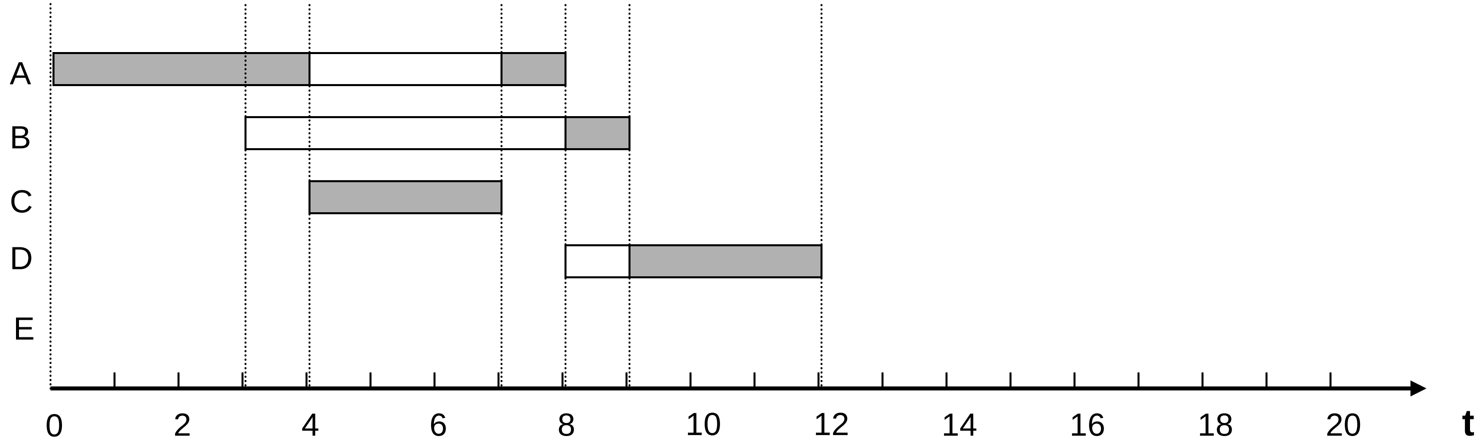
Process	Start	Runtime	Priority
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



Scheduling

preemtive priority scheduling

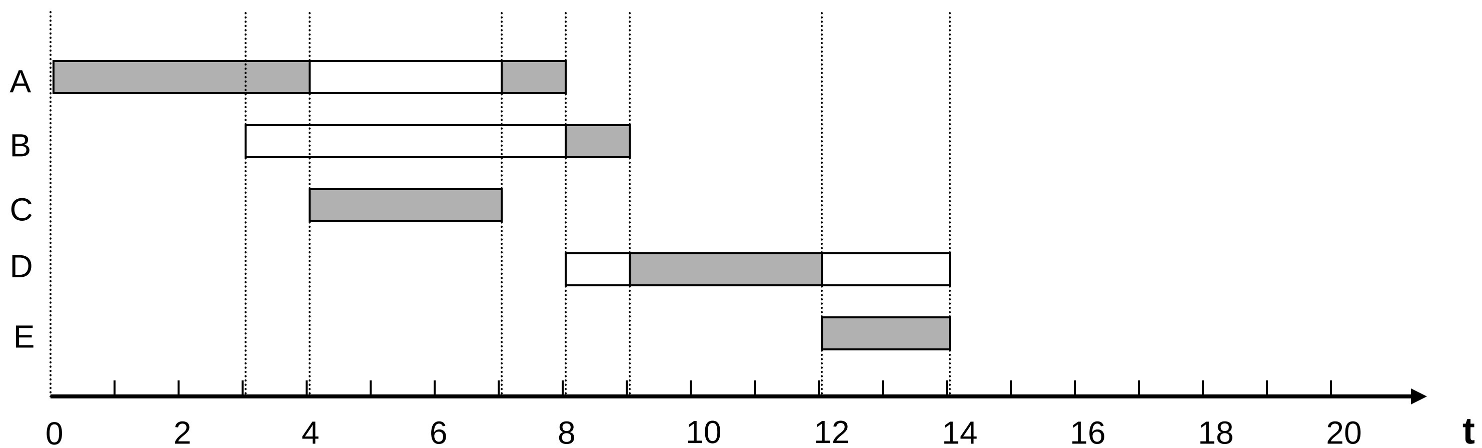
Process	Start	Runtime	Priority
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



Scheduling

preemtive priority scheduling

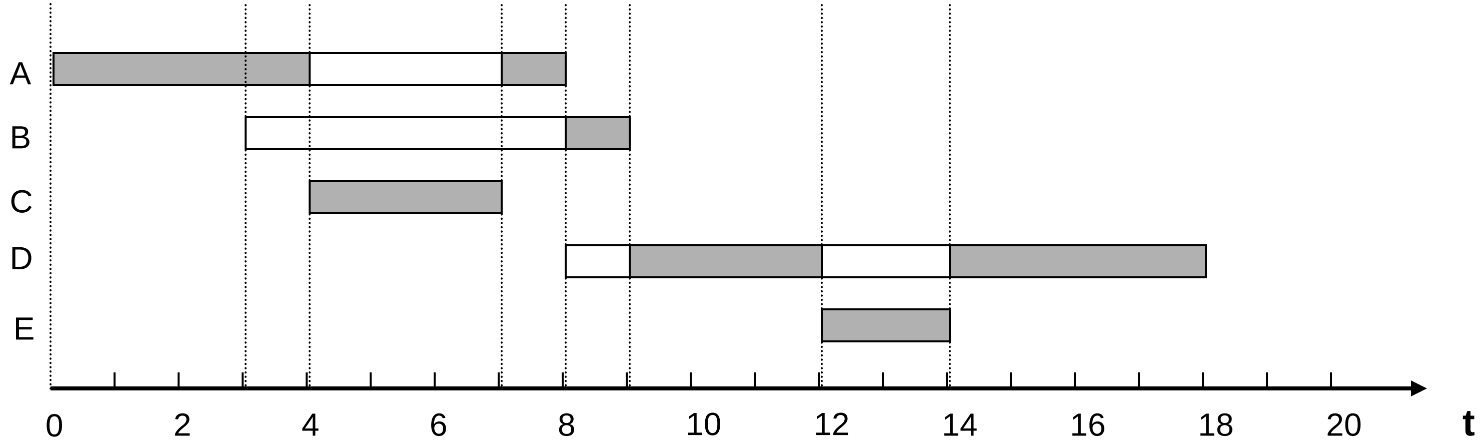
Process	Start	Runtime	Priority
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



Scheduling

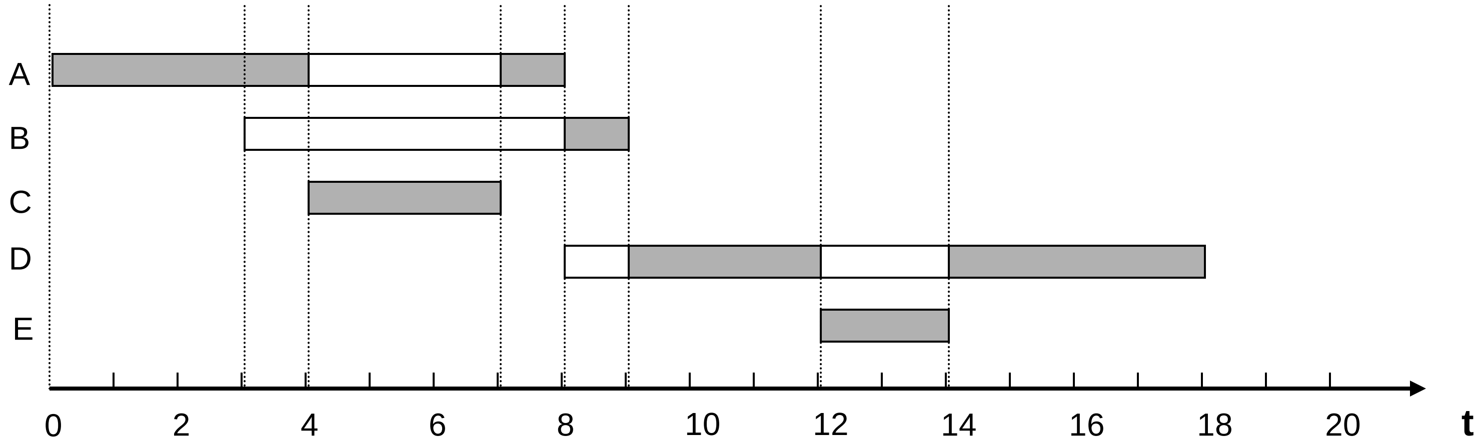
preemtive priority scheduling

Process	Start	Runtime	Priority
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



Scheduling

preemtive priority scheduling



Process	A	C	A	B	D	E	D
Time (RUNNING)	0	4	7	8	9	12	14

Lottery Scheduling

- OS gives “lottery tickets” to processes
- Scheduler picks a ticket randomly and gives CPU to the winner
- Higher-priority processes get more tickets

- Advantage:
  - processes may exchange tickets
  - it is possible to fine tune the share of CPU that a process receives
  - easy to implement

Implementing Lottery Scheduling

- Implementation is very simple
- Add a `num_tickets` field to PCB
- At scheduling time:
  - Generate a random ticket number *winner*
  - Loop over processes, keep a *counter*
  - If *counter* > *winner* then pick that process
  - Otherwise, add the process' tickets to *counter* and continue

Lottery Scheduling

60	15	20	5

Lottery Scheduling

- Winner: 83

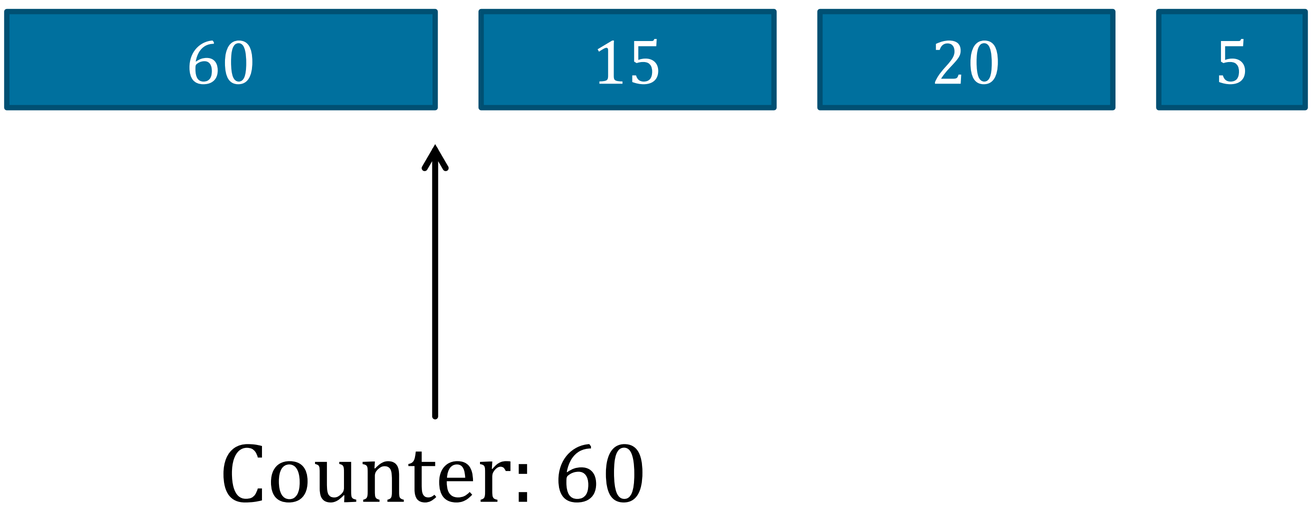
Lottery Scheduling

- Winner: 83

Counter: 0

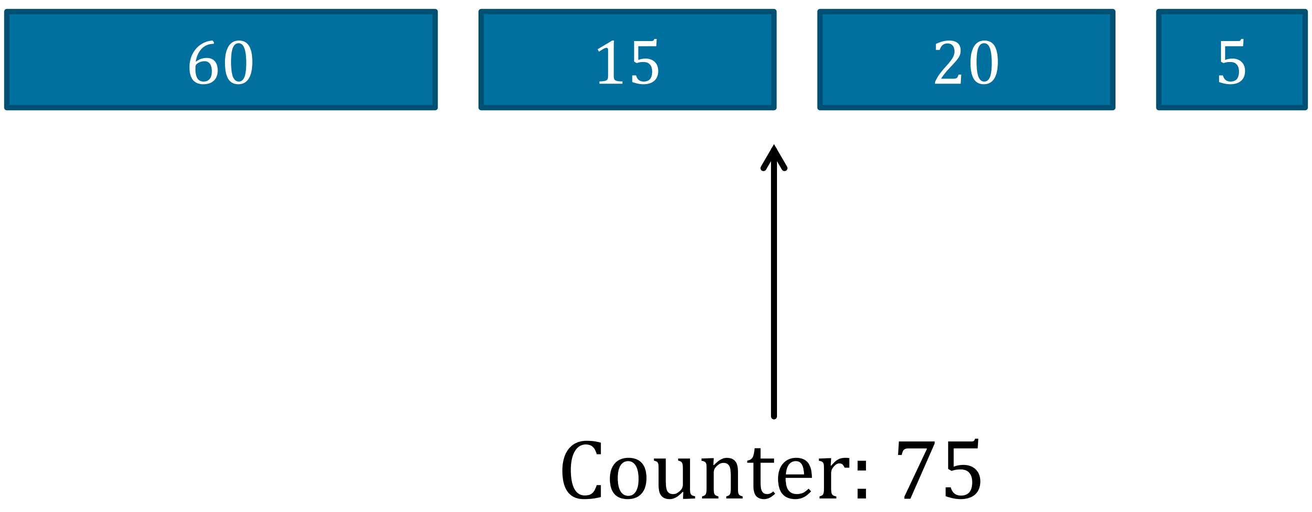
Lottery Scheduling

- Winner: 83



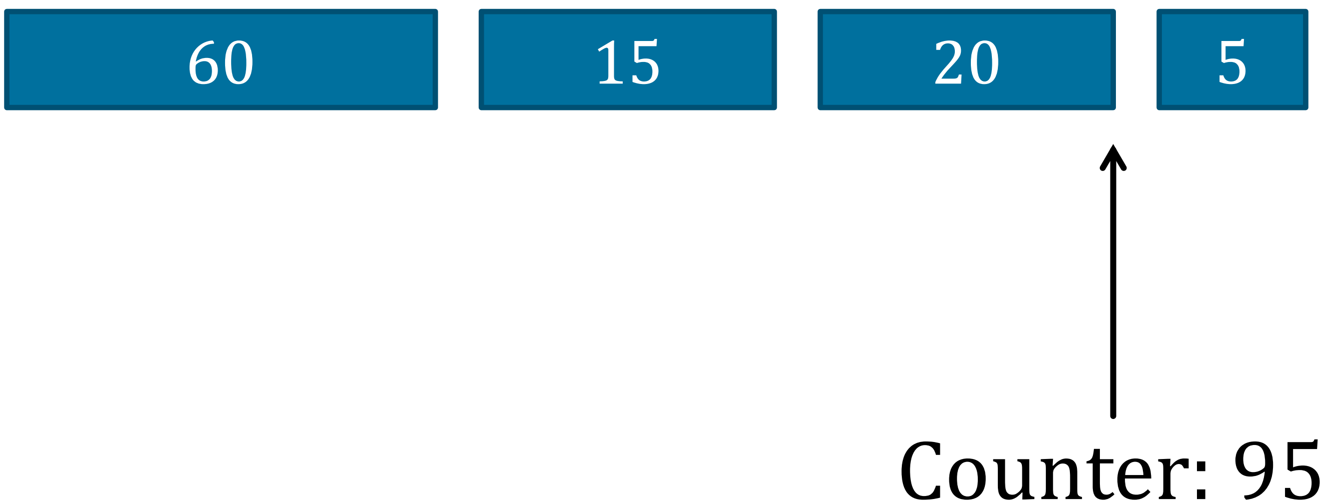
Lottery Scheduling

- Winner: 83



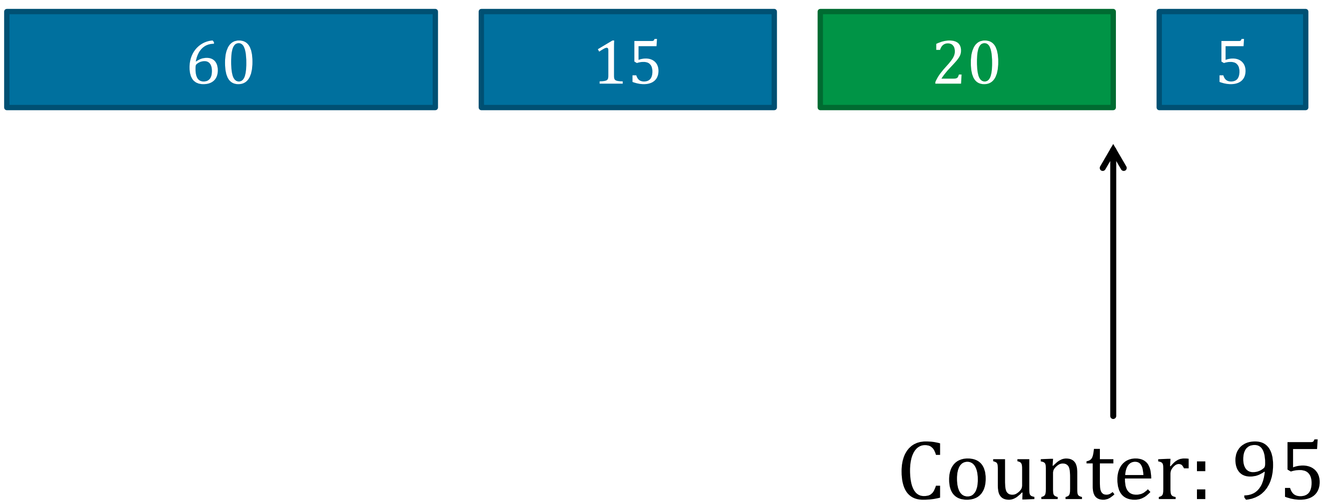
Lottery Scheduling

- Winner: 83



Lottery Scheduling

- Winner: 83

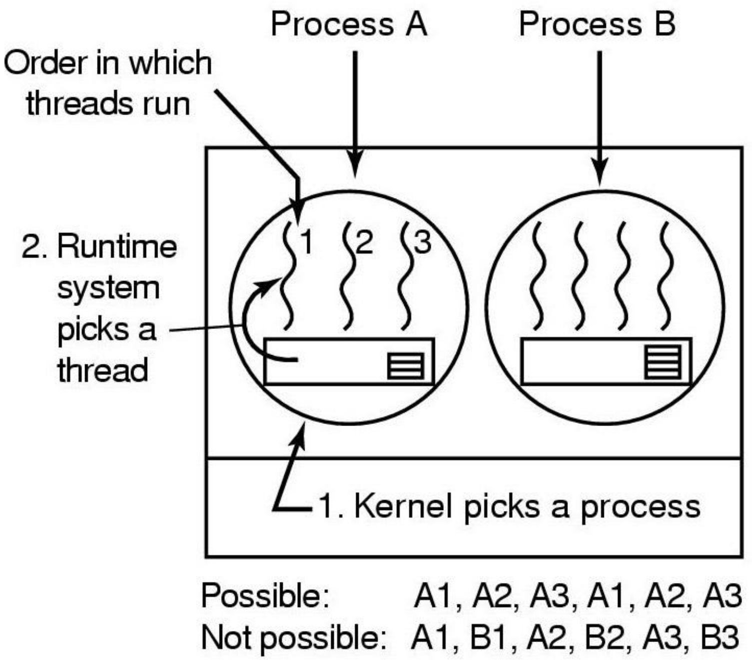


Thread Scheduling

If threads are implemented in user space, only one process' threads are run inside a quantum

Possible scheduling of user-level threads

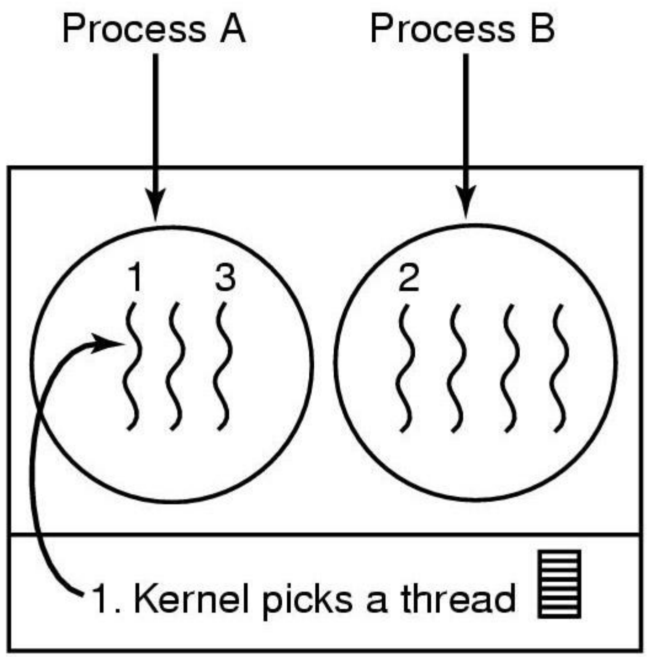
- 48-msec process quantum
- Threads run 8 msec/CPU burst



Thread Scheduling

If threads are implemented in the kernel, threads can be interleaved

Kernel may decide to switch to a thread belonging to the same process for efficiency reasons (memory map does not change)



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

Policy versus Mechanism

Sometimes an application may want to influence the scheduling of cooperating processes (same user, or children processes) to achieve better overall performance

Separate what is <u>allowed</u> to be done with <u>how</u> it is done

- process knows which of its children threads are important and need priority

Scheduling algorithm parameterized

- Mechanism in the kernel

Parameters filled in by user processes

- Policy set by user process

Linux - CFS

Completely fair scheduler (CFS)

Ingo Molnar:

80% of CFS's design can be summed up in a single sentence: CFS basically models an "ideal, precise multi-tasking CPU" on real hardware.

On real hardware, we can run only a single task at once, so while that one task runs, the other tasks that are waiting for the CPU are at a disadvantage - the current task gets an unfair amount of CPU time. In CFS this fairness imbalance is expressed and tracked via the per-task p->wait_runtime (nanosec-unit) value. "wait_runtime" is the amount of time the task should now run on the CPU for it to become completely fair and balanced.