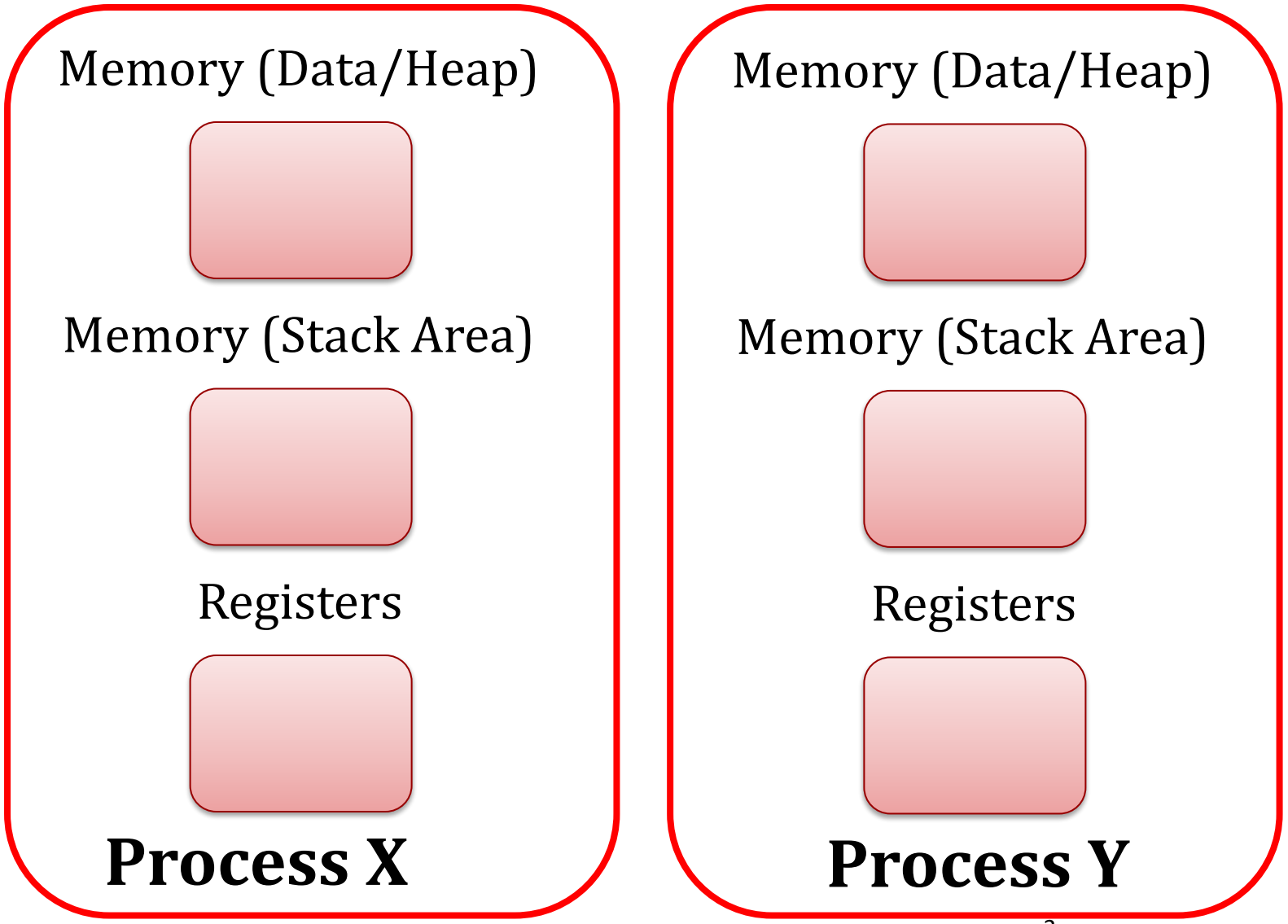
EC 440 – Introduction to Operating Systems

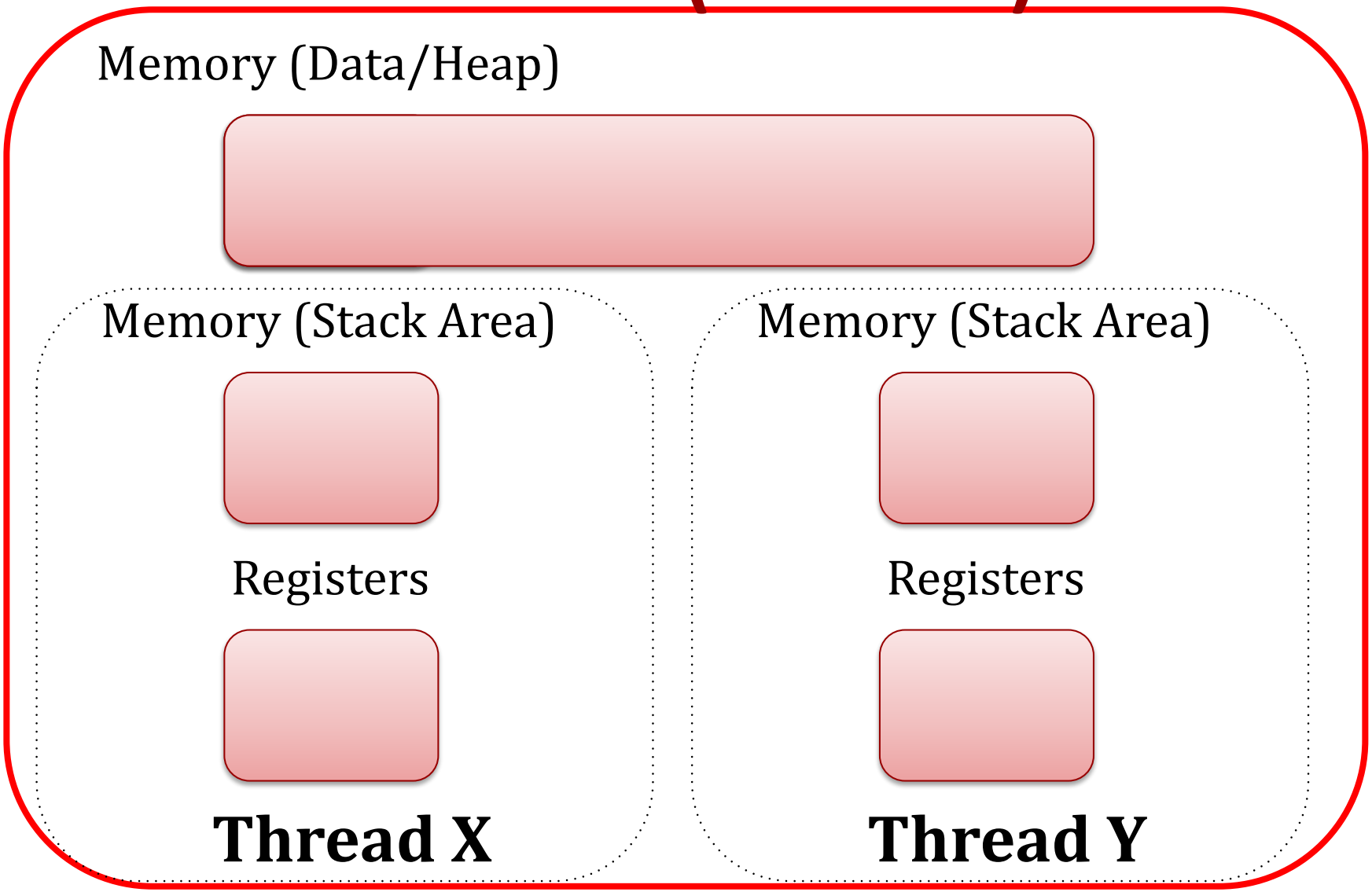
Orran Krieger (BU)
Larry Woodman (Red Hat)

Administration and Review

Processes (Context)



Threads (Context)



Threads

- More efficient than using seperate processes
- Adds complexity, need to write re-entrant code (more later)
- Threading can be done in either user level or kernel level, or via a combination of both.

Scheduling

Scheduling

Problem: Many processes to execute, but many more processes than 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*

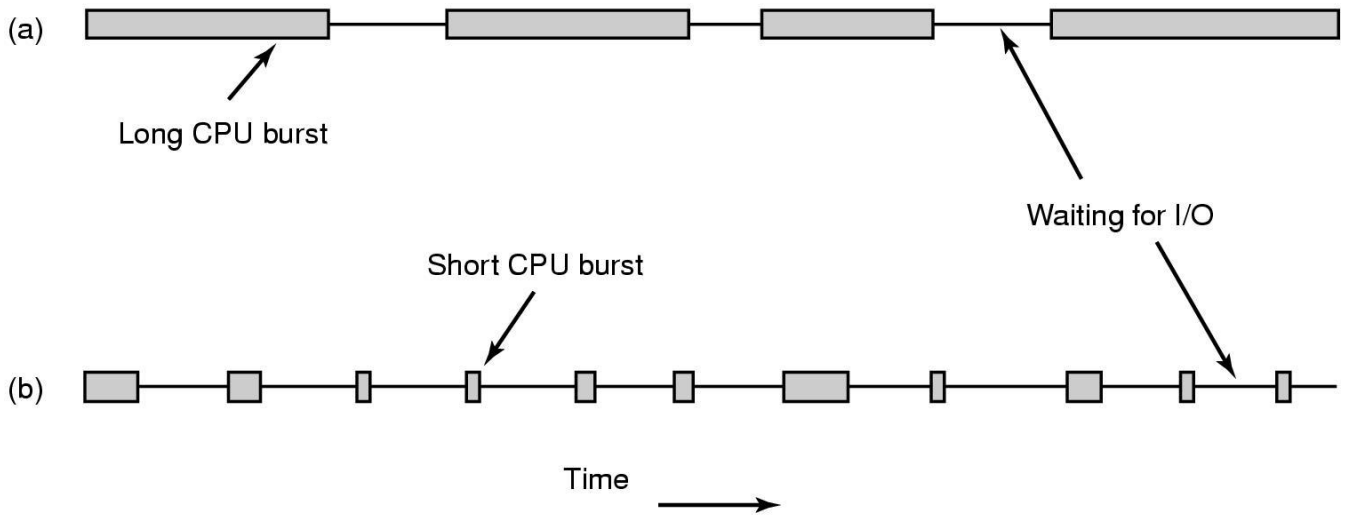
Switching is expensive

- 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
- Swithc back to user mode in the new process

CPU-bound and I/O-bound Processes

Bursts of CPU usage typically 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
- current process has been running for a long time

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

Goals that systems may aim for include

- 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

Scheduler Goals

Goals that systems may aim for include

- 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

Scheduler Goals

Goals that systems may aim for include

- 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

Scheduler Goals

Goals that systems may aim for include

- 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

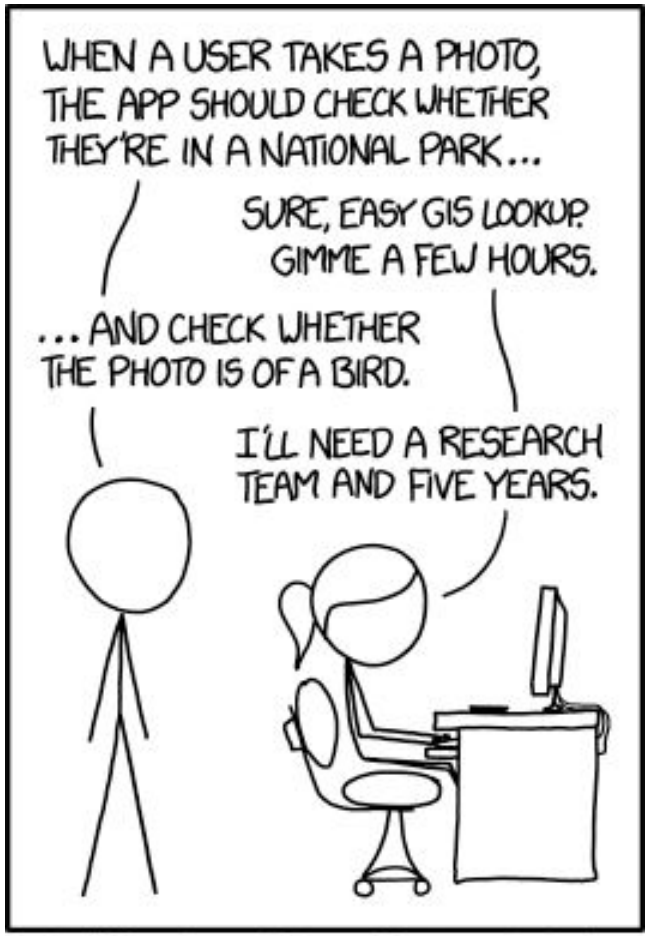
Interactive Systems

- 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

It gets really complicated...



Scheduling in Batch Systems

Goals are typically:

- Throughput:
  maximize jobs per hour
- Turnaround time:
  minimize average 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:

Analyzing First Come First Served

Turnaround time depends on order we pick jobs
Assuming jobs arrive at time 0:

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

Analyzing First Come First Served

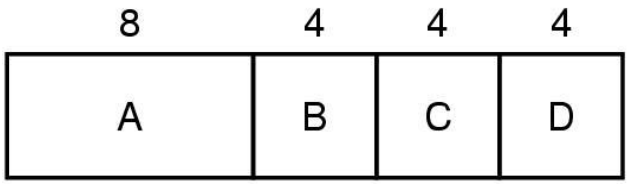
Turnaround time depends on order we pick jobs
Assuming jobs arrive at time 0:

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

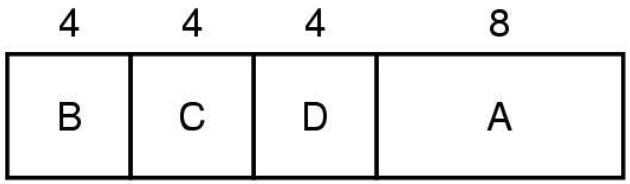
Turnaround time: (5 + 10 + 42) / 3 = 19 minutes

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)



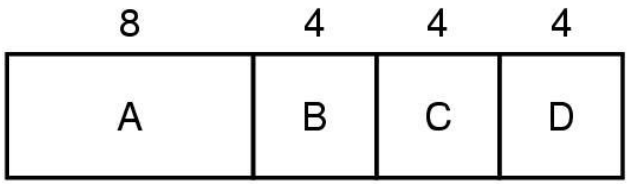
(a)



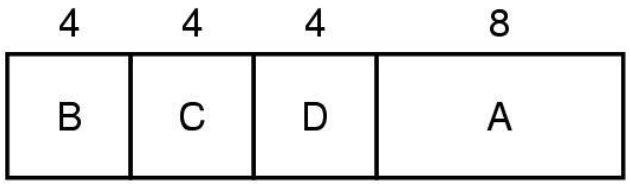
(b)

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)



(a)

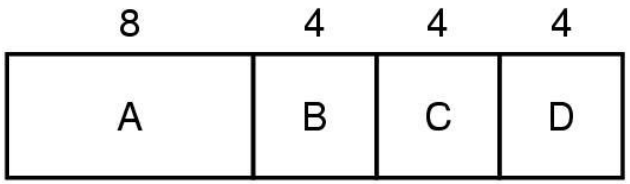


(b)

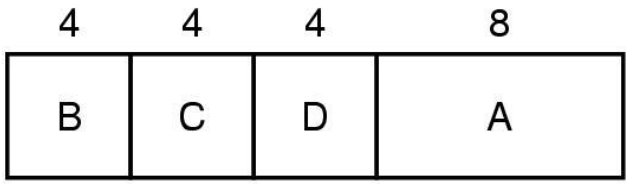
- When do we know run time ahead of time?

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)



(a)

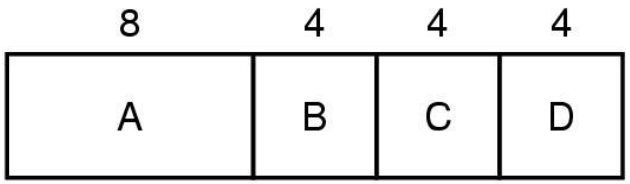


(b)

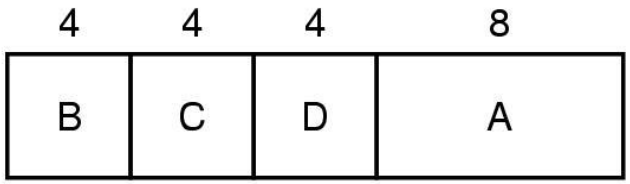
- When do we know run time ahead of time? HPC clusters, Real Time systems are examples, but normally we do not.

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)



(a)



(b)

- When do we know run time ahead of time? HPC clusters, Real Time systems are examples, but normally we do not.
- What if we don't know all the jobs ahead of time?

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)

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)

- Average turnaround time is
  (2 + 6 + (7 -3) + (8 -3) + (9-3))/5 = 4.6

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)

- Average turnaround time is
  (2 + 6 + (7 -3) + (8 -3) + (9-3))/5 = 4.6
- Can we find an order that performs better?

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)

- Average turnaround time is
  (2 + 6 + (7 -3) + (8 -3) + (9-3))/5 = 4.6
- 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

Shortest Remaining Time Next

Available at minute 0

Available at minute 3

A (2 mins) B (4 mins) C (1 mins) D (1 mins) E (1 mins)

- Recall, with shortest job first:
  (2 + 6 + (7 -3) + (8 -3) + (9-3))/5 = 4.6

Shortest Remaining Time Next

Available at minute 0

Available at minute 3

A (2 mins) B (4 mins) C (1 mins) D (1 mins) E (1 mins)

- Recall, with shortest job first:
  (2 + 6 + (7 -3) + (8 -3) + (9-3))/5 = 4.6

A B

- With shortest remaining time:
  2

Shortest Remaining Time Next

Available at minute 0

Available at minute 3

- With shortest job first:

  (2 + 6 + (7 -3) + (8 -3) + (9-3))/5 = 4.6

- With shortest remaining time:

  2 + (4-3) + (5-3) + (6-3)

Shortest Remaining Time Next

Available at minute 0

Available at minute 3

- With shortest job first:

  (2 + 6 + (7 -3) + (8 -3) + (9-3))/5 = 4.6

- With shortest remaining time:

  (2 + (4-3) + (5-3) + (6-3) + 9)/5 = 3.4

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

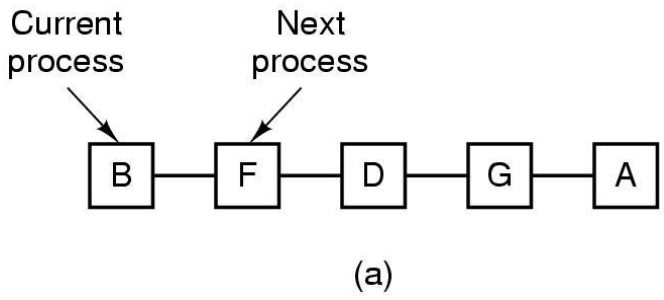
Round Robin Scheduling

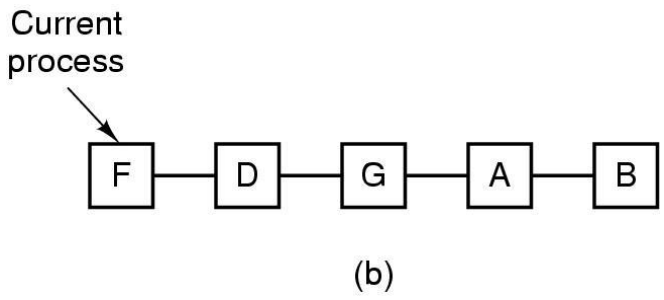
Each is process is assigned a *quantum*

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)

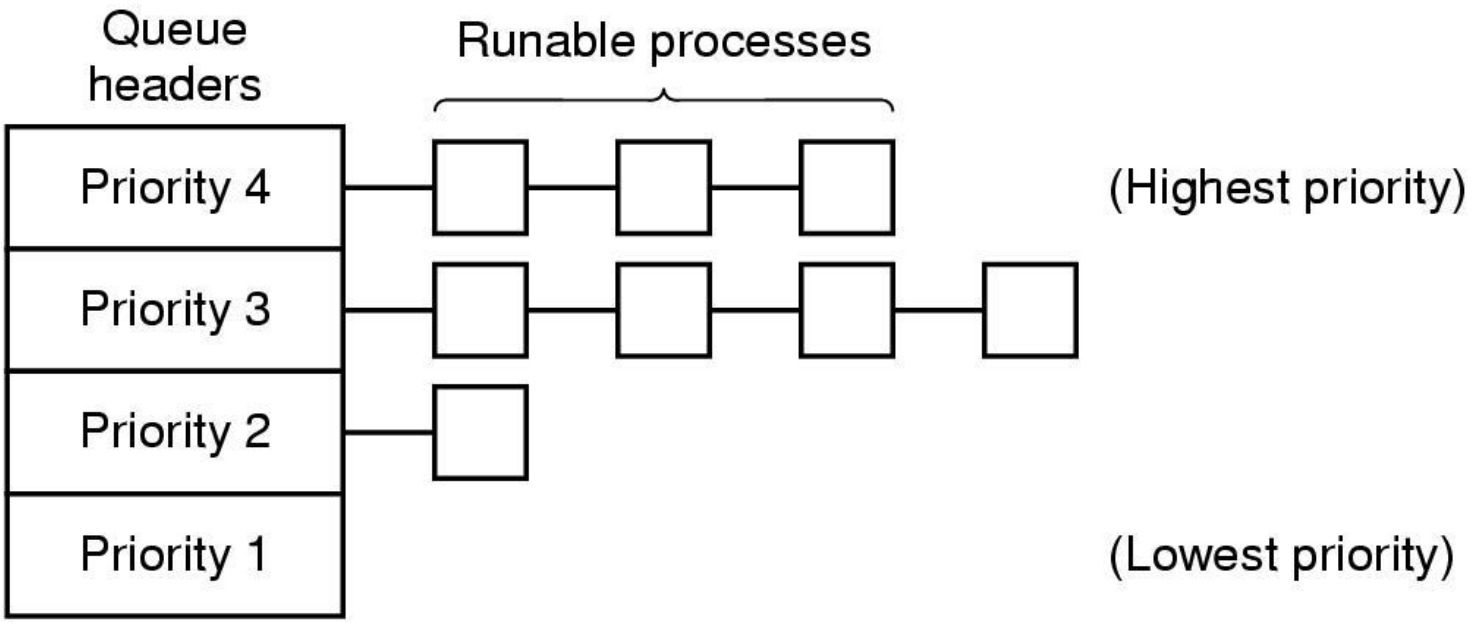
- too big, response time can be very bad

  (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



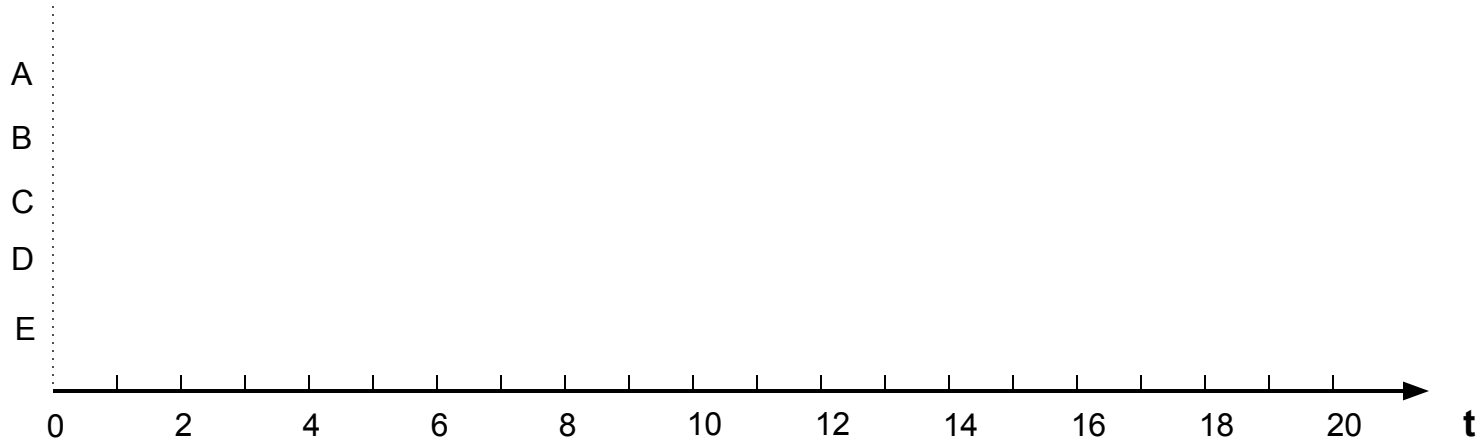
Scheduling: Example

non-preemptive 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

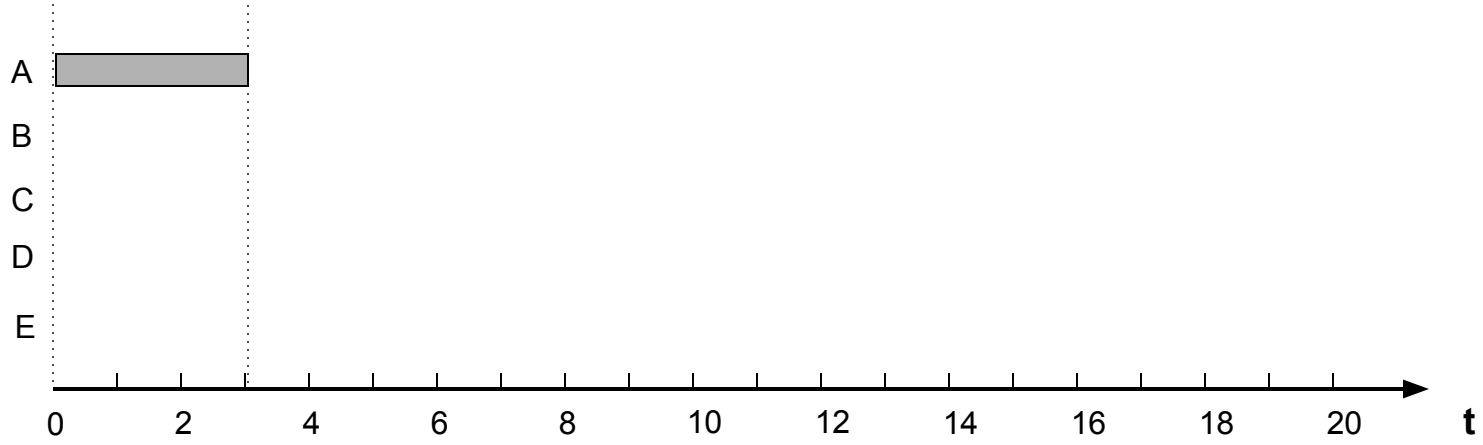
Scheduling

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



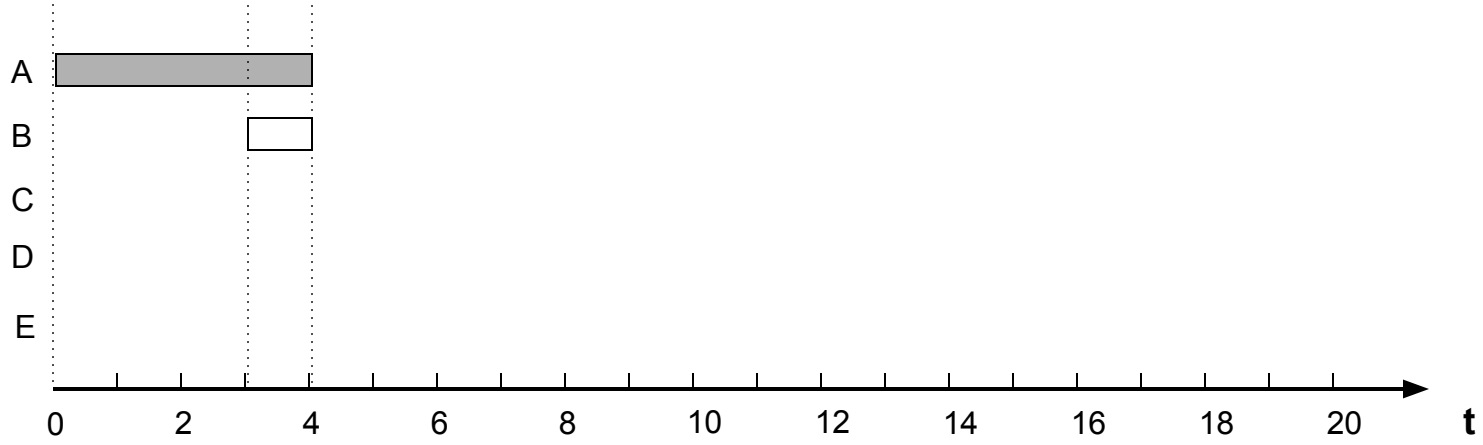
Scheduling

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



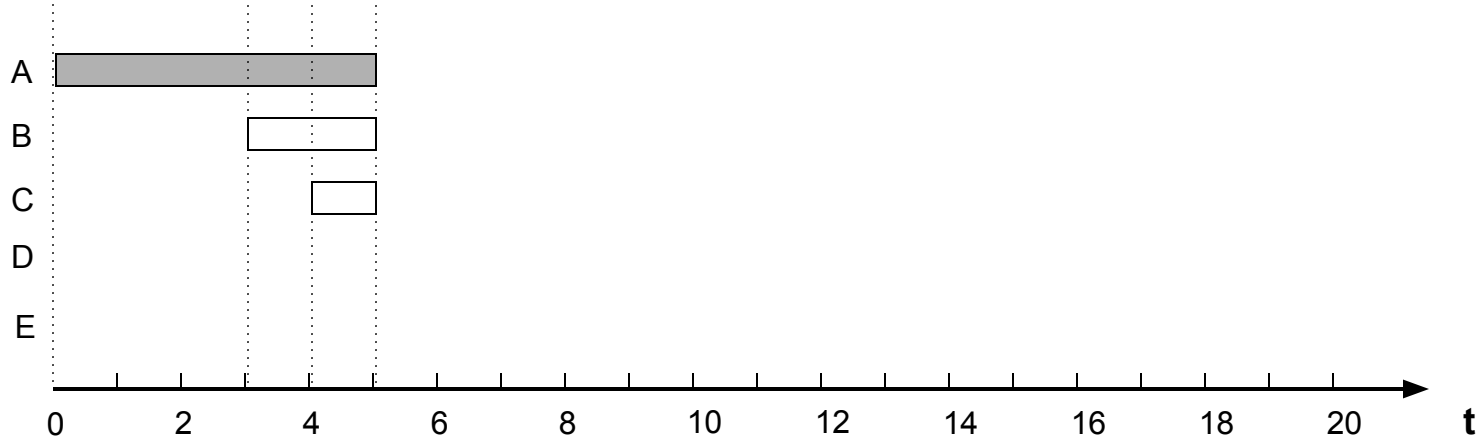
Scheduling

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



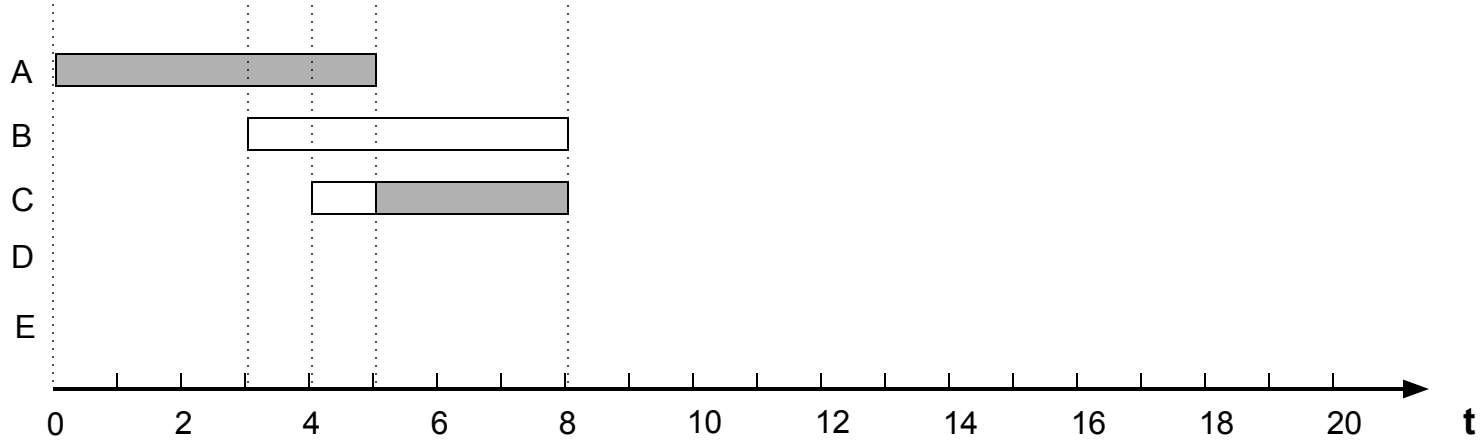
Scheduling

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



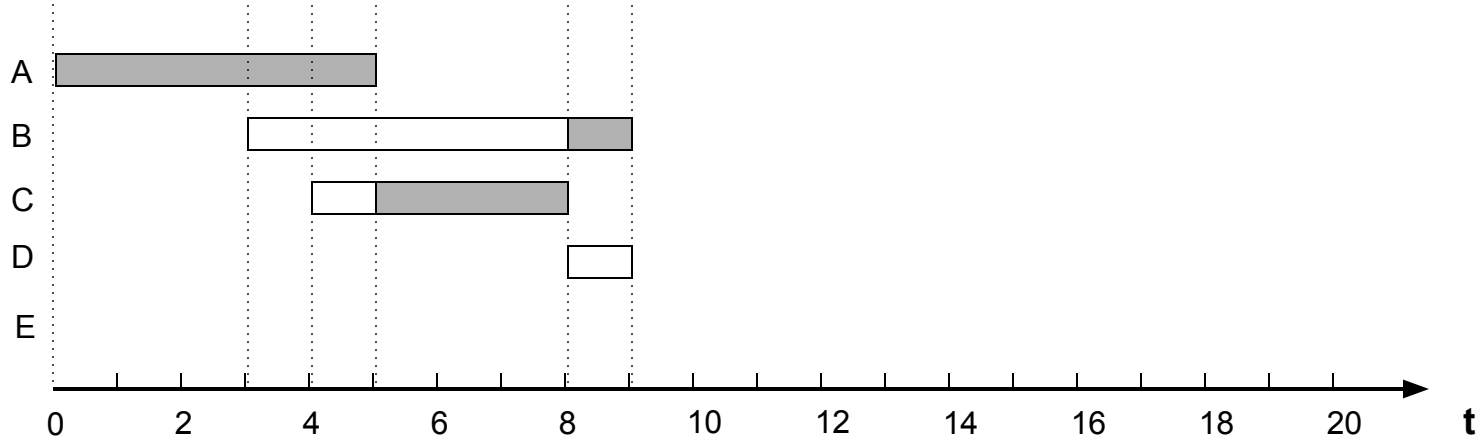
Scheduling

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



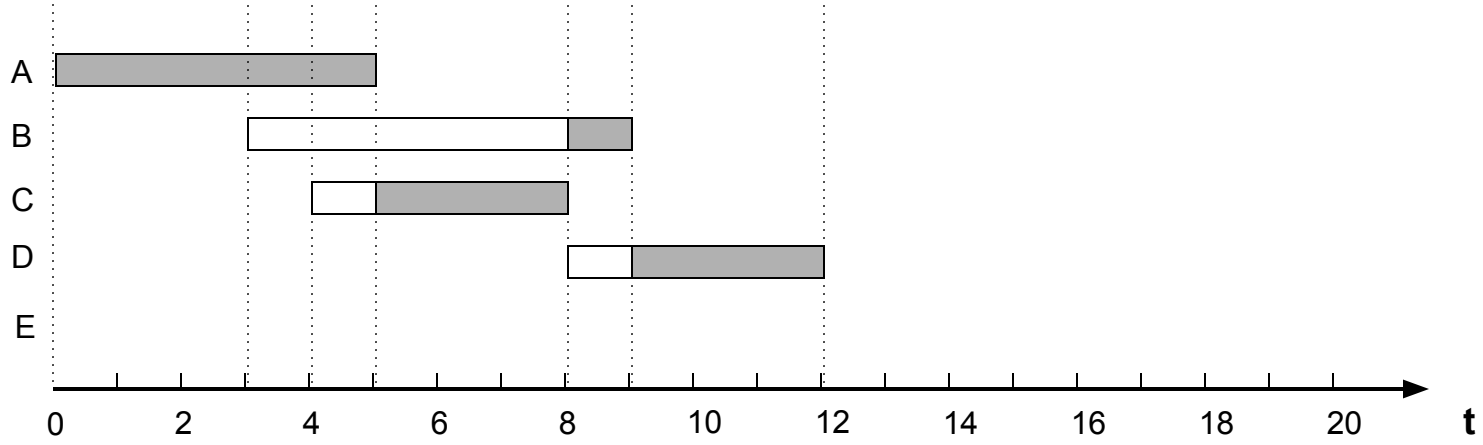
Scheduling

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



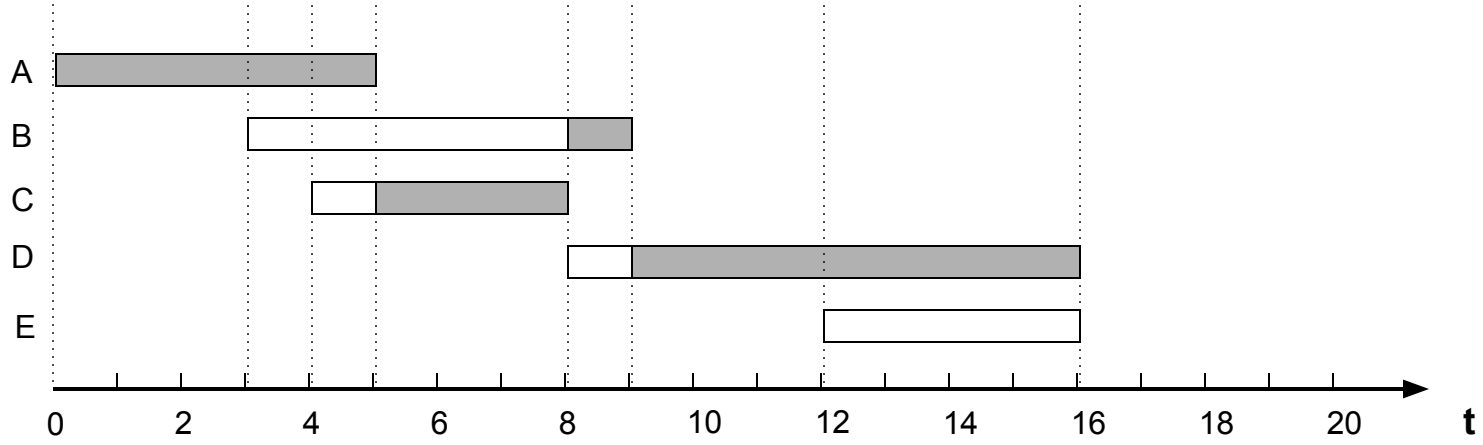
Scheduling

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



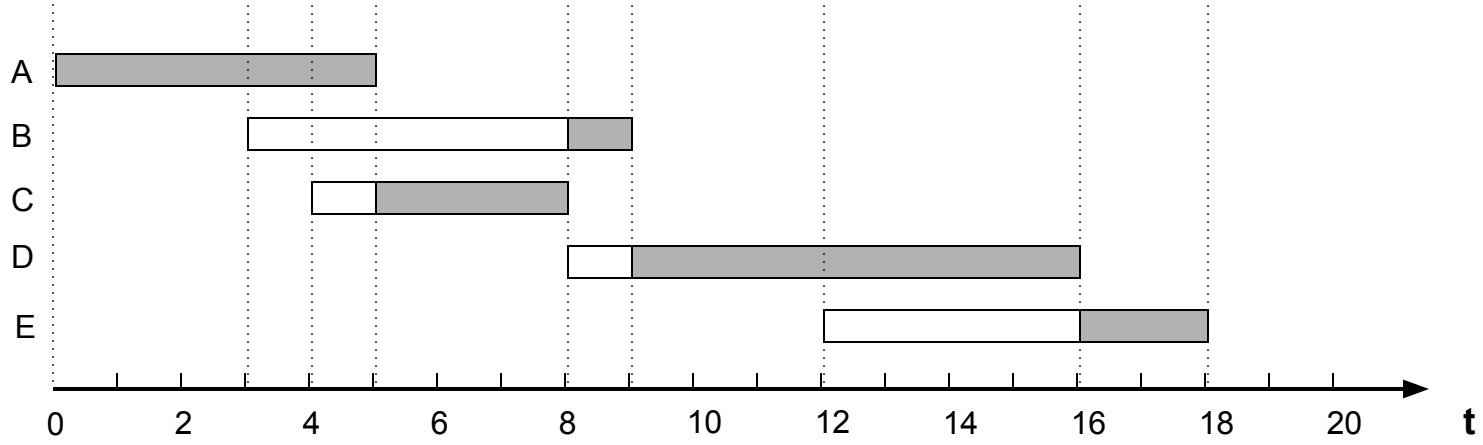
Scheduling

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

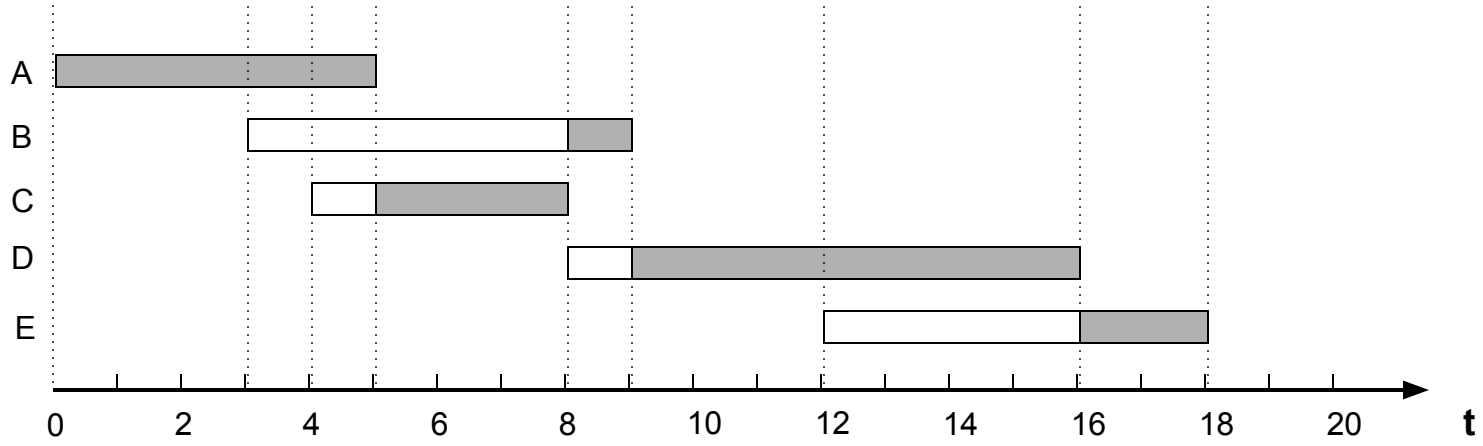


Scheduling

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



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

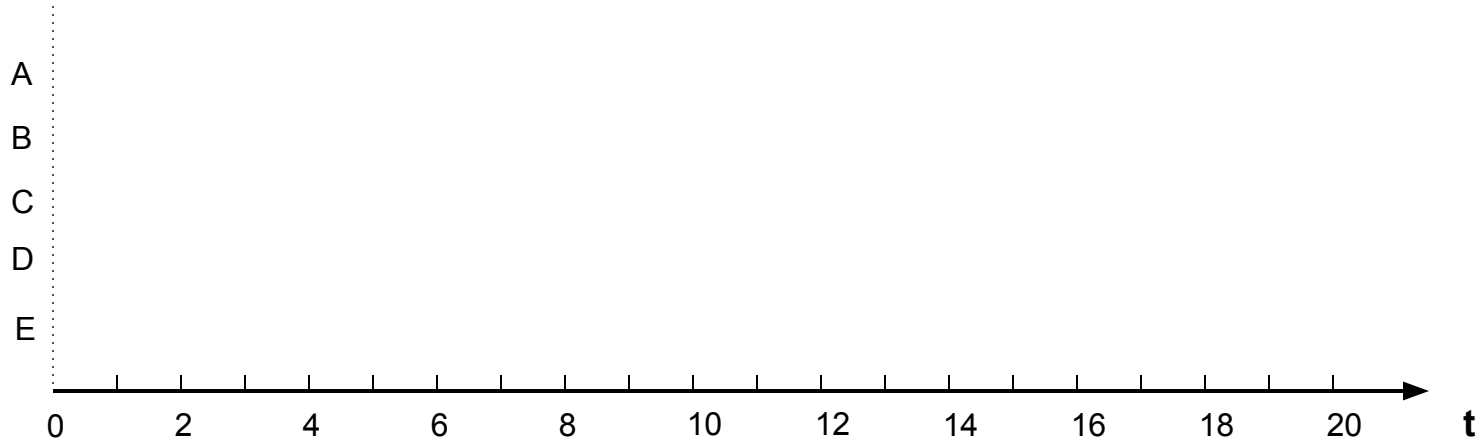
Scheduling: Example

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

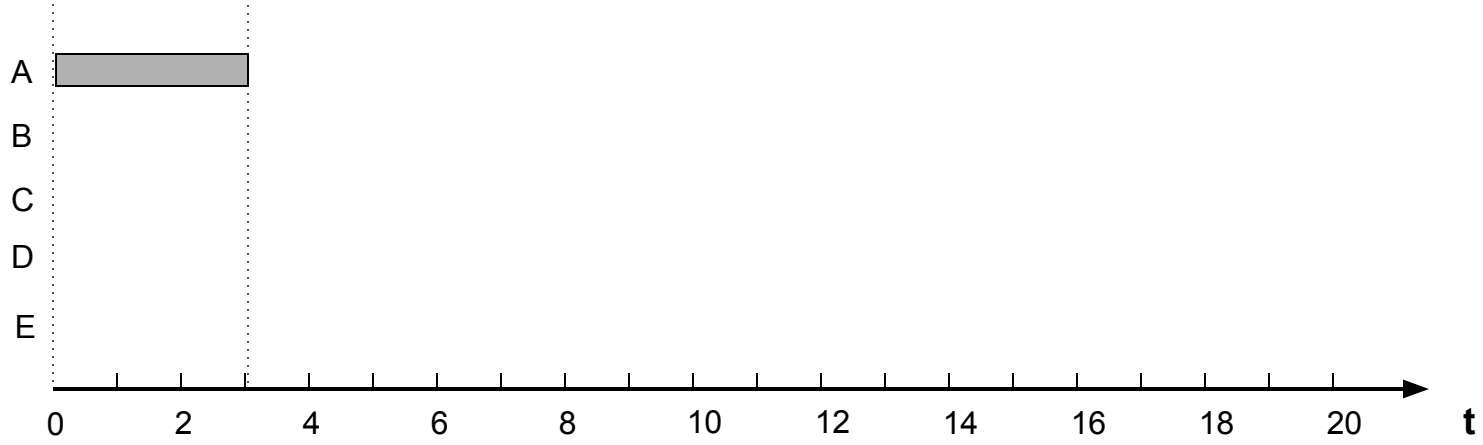
Scheduling

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



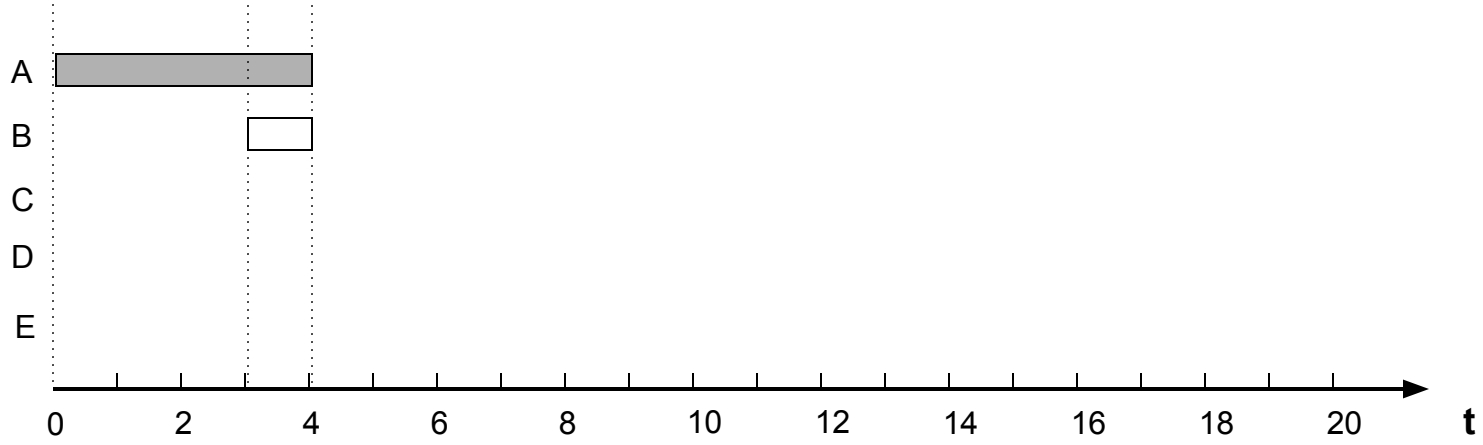
Scheduling

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



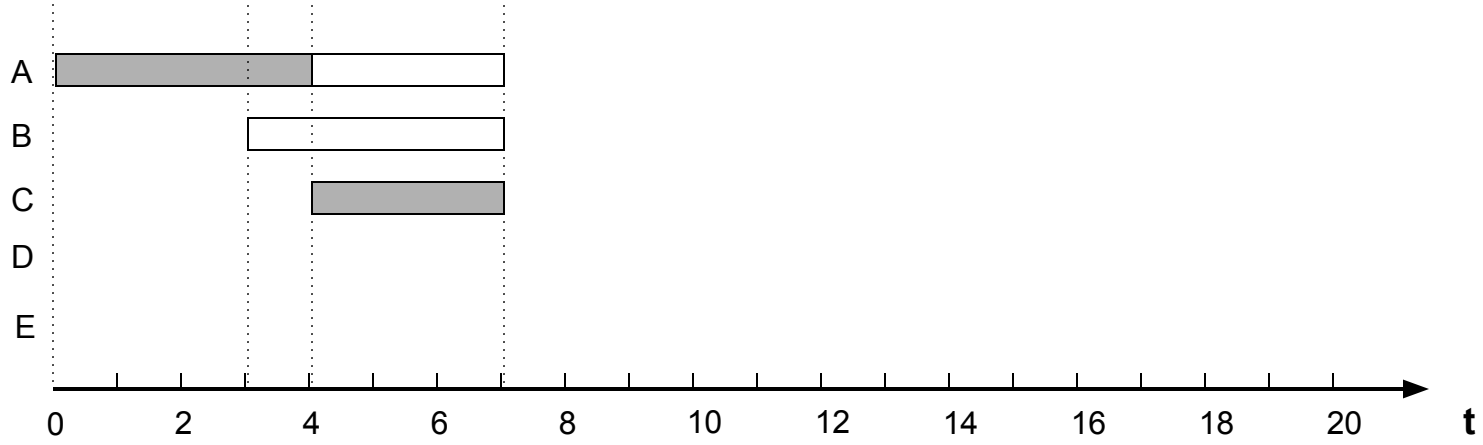
Scheduling

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



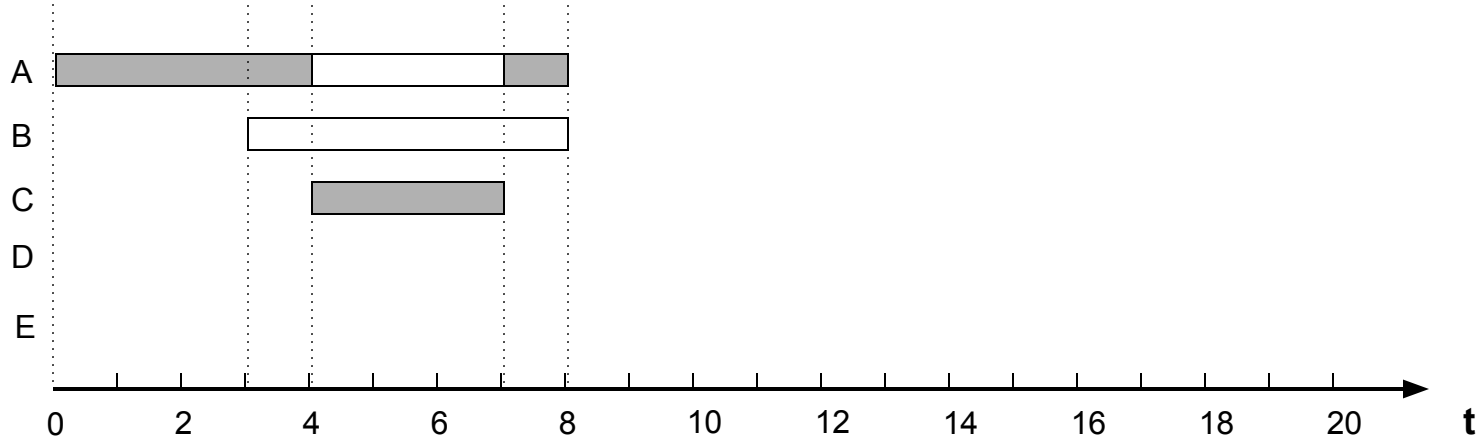
Scheduling

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



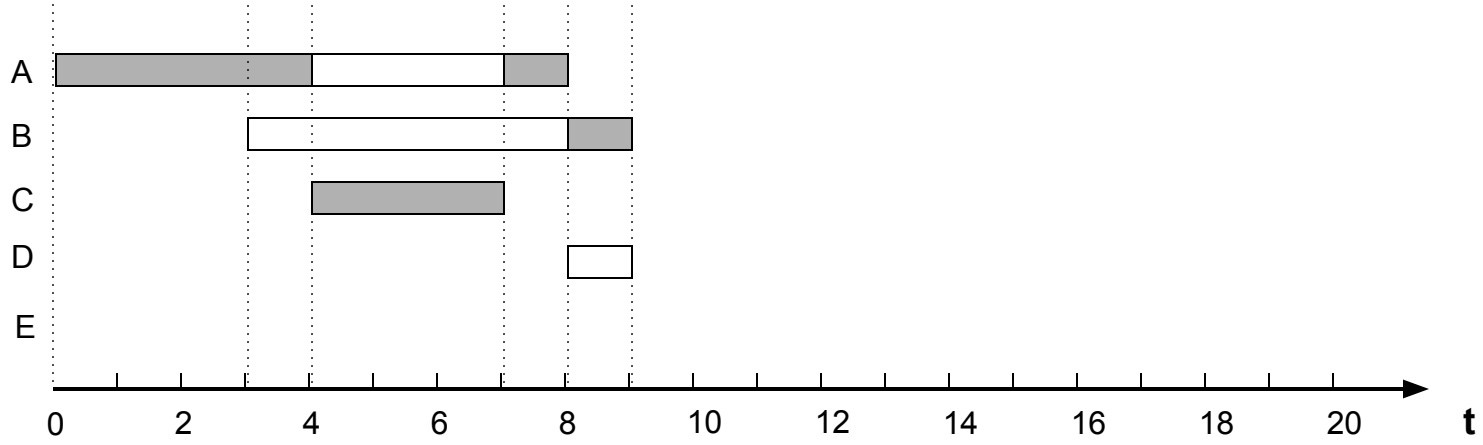
Scheduling

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



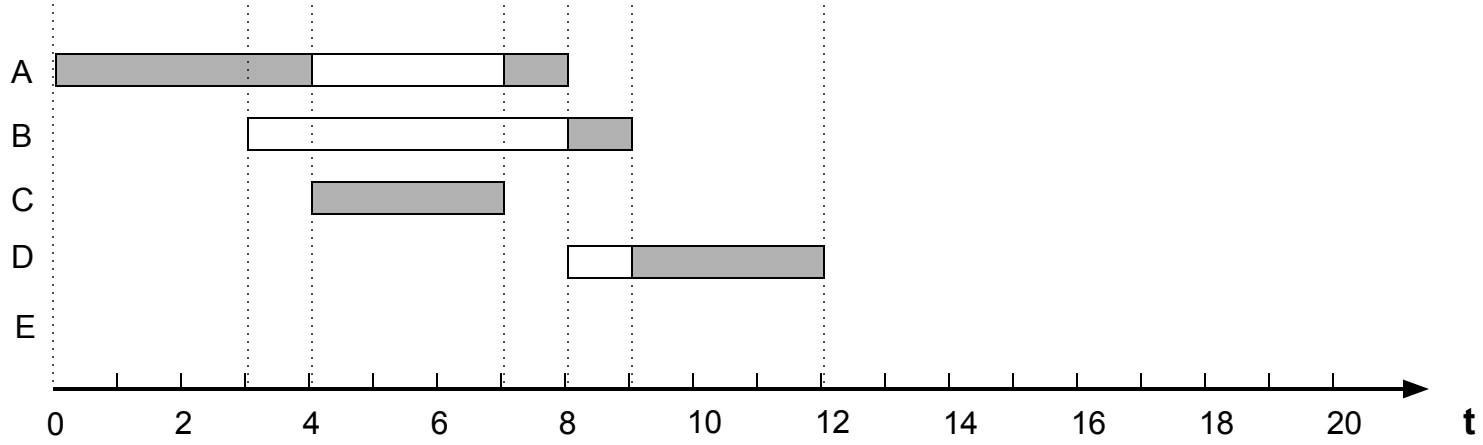
Scheduling

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



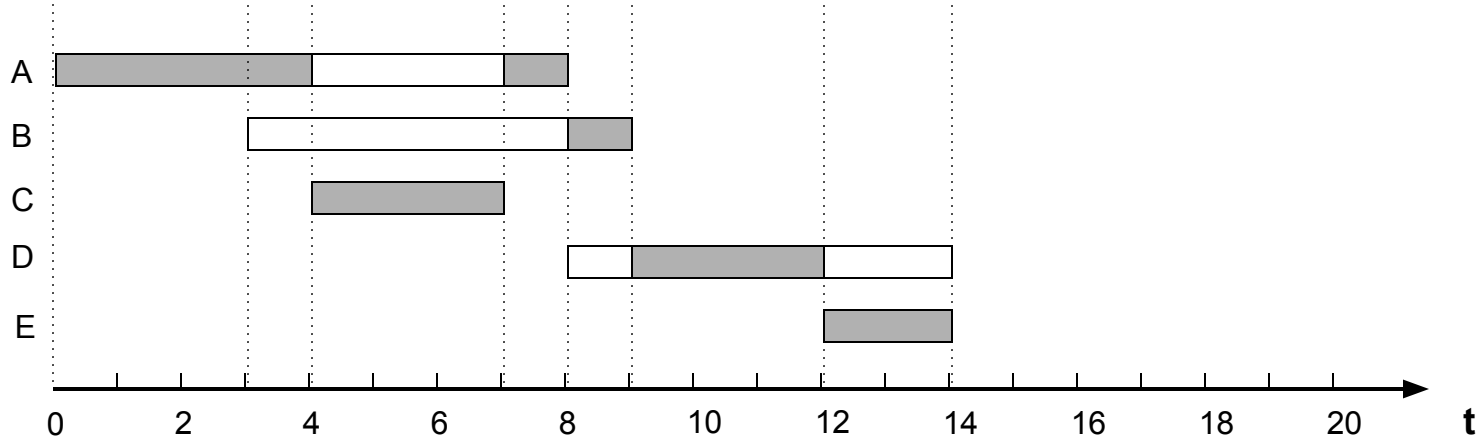
Scheduling

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



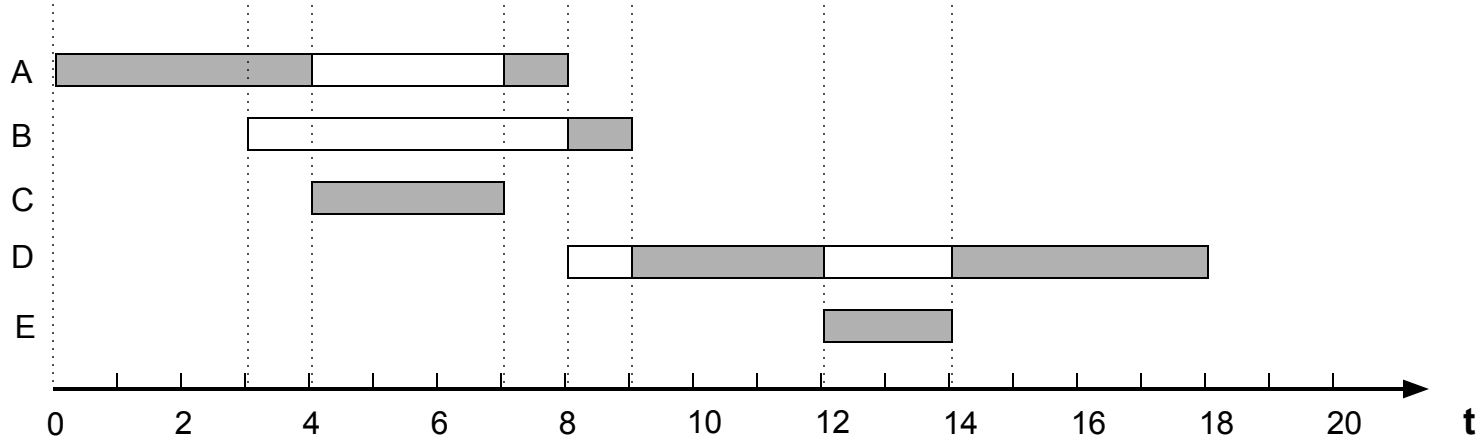
Scheduling

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

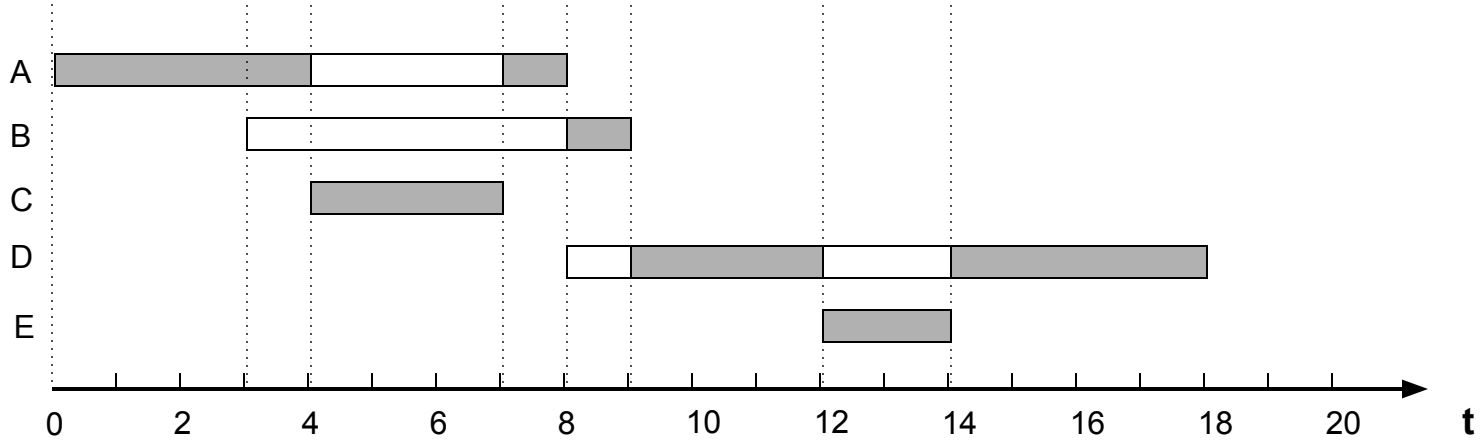


Scheduling

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



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

Lottery Scheduling

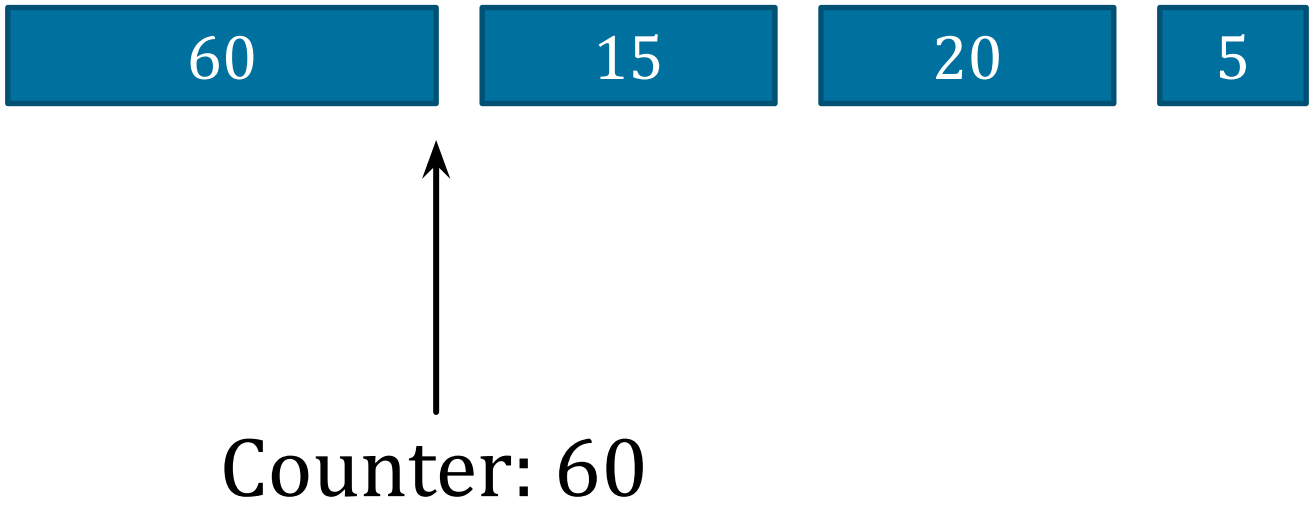
- Winner: 83

Lottery Scheduling

- Winner: 83

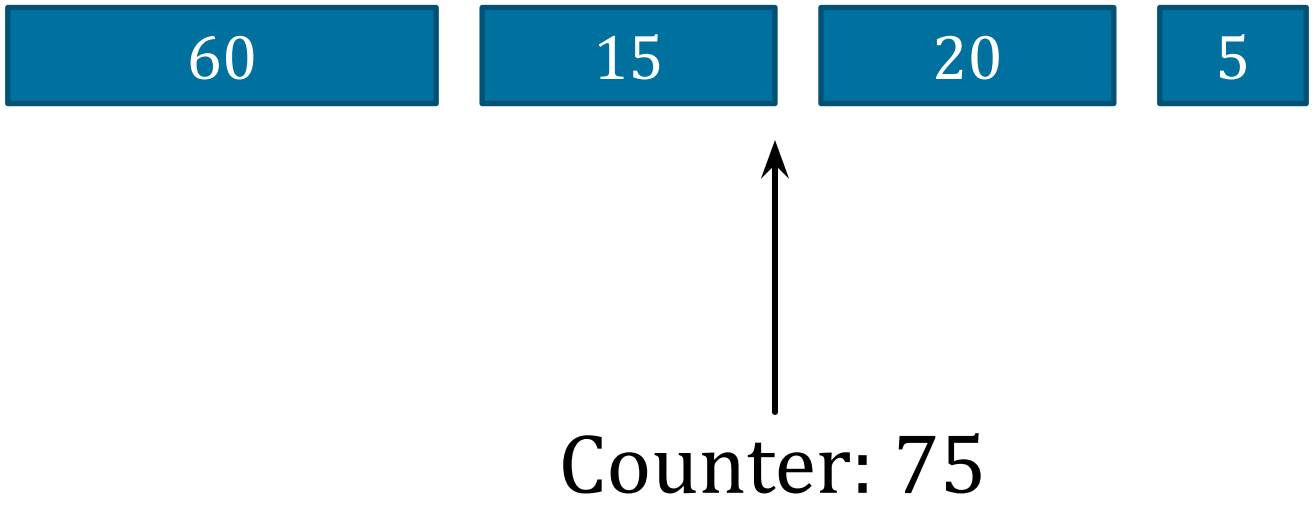
Lottery Scheduling

- Winner: 83



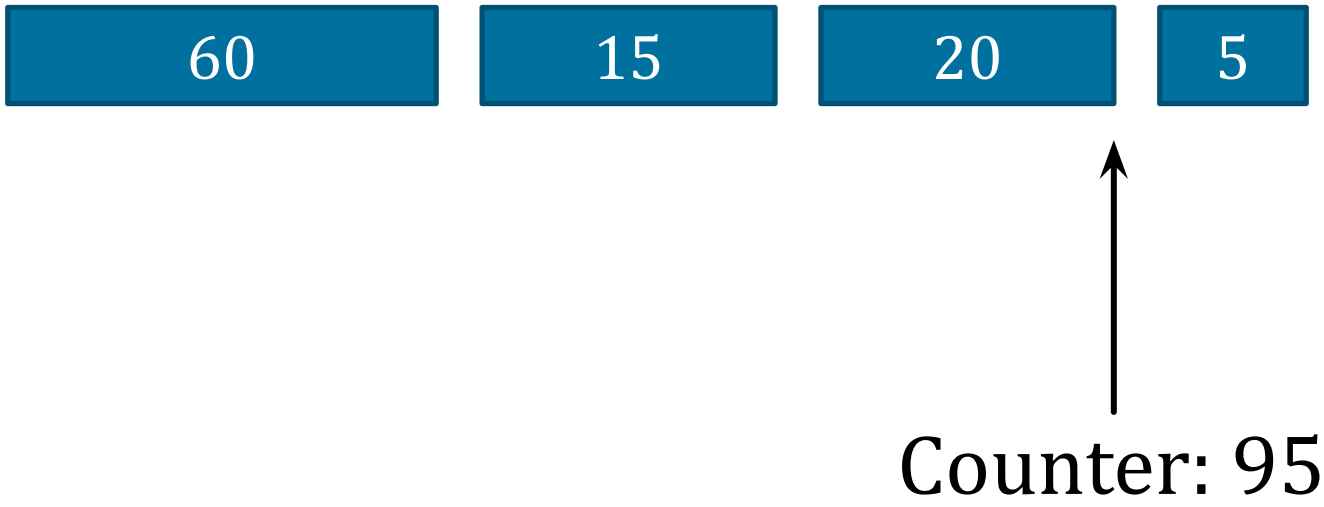
Lottery Scheduling

- Winner: 83



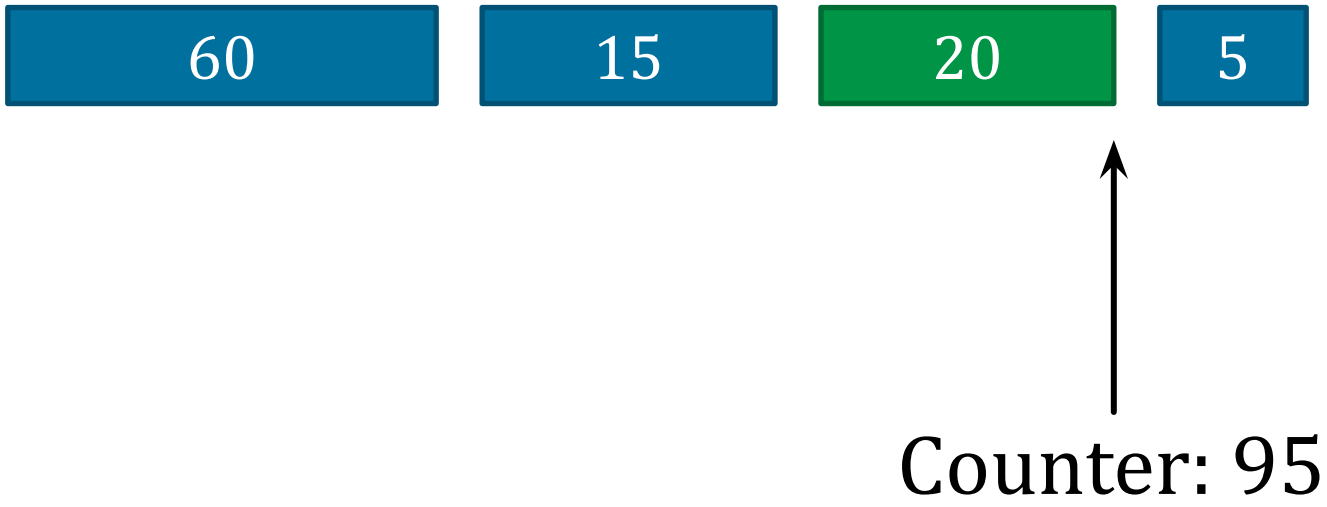
Lottery Scheduling

- Winner: 83



Lottery Scheduling

- Winner: 83

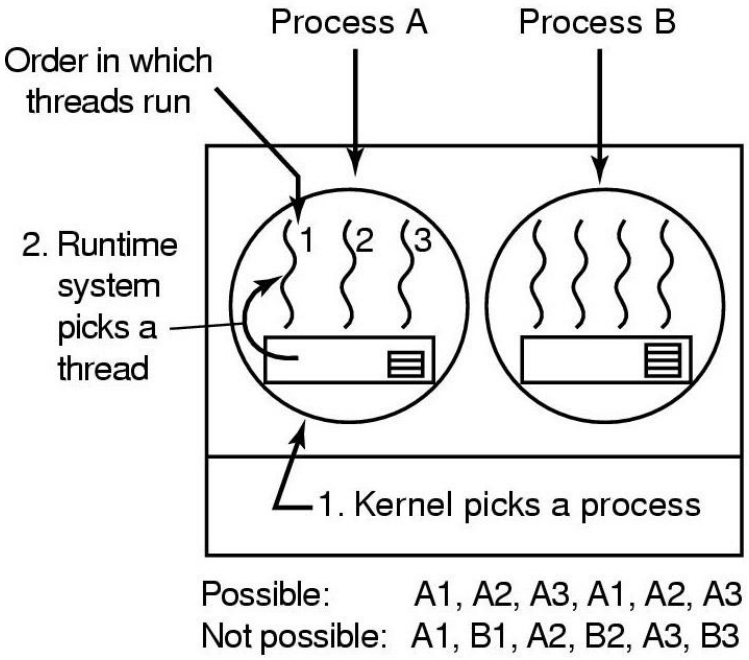


Thread Scheduling

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

Possible scheduling of user-level threads

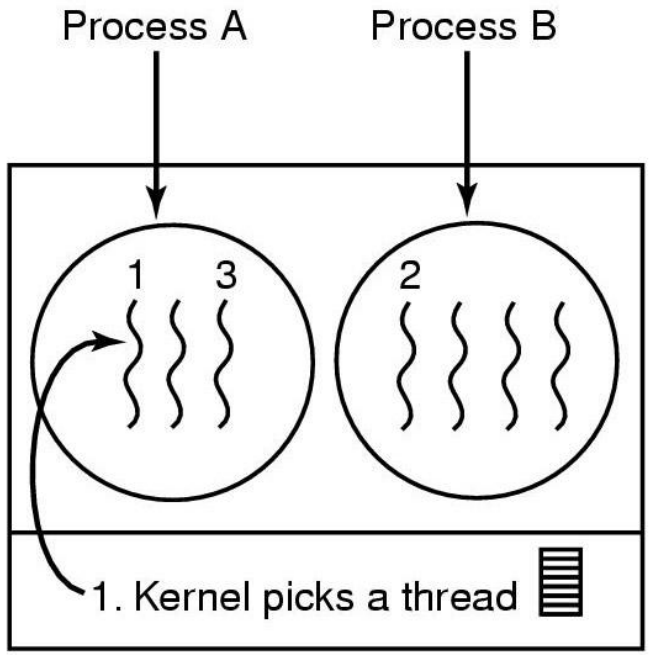
- 48-msec process quantum
- Threads run 8 msec/CPU burst



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 allowed to be done with how 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

Lets go deep on a real system

- Linux allows you to specify a scheduling policy for a subset of processes
- These processes can be isolated to different processors, enabling fine grained control

Linux scheduler

- **Priorities**

  0 - 139, lower number is higher priority, default = 20

  - **nice()**
    - syscall/command to increase/decrease scheduling priority
  - **top**
    - utility that displays scheduling priority & nice values

- **policies**

  SCHED_FIFO, SCHED_RR - non-preemptive/preemptive RT
  SCHED_BATCH, SCHED_IDLE - batch mode/lowest priority
  SCHED_OTHER - default is CFS - Completely Fair Scheduler

  - **chrt/sched_setscheduler()**
    - command/syscall to set scheduling policy for a process

Linux scheduler implementation summary (man sched)

```
SCHED(7)                                                                Linux Programmer's Manual

NAME
       sched - overview of CPU scheduling

DESCRIPTION
       Since Linux 2.6.23, the default scheduler is CFS, the "Completely Fair Scheduler".  The CFS scheduler replaced the earlier "O(1)" scheduler.

   API summary
       Linux provides the following system calls for controlling the CPU scheduling behavior, policy, and priority of processes (or, more precisely, threads).

       nice(2)
              Set a new nice value for the calling thread, and return the new nice value.

       getpriority(2)
              Return the nice value of a thread, a process group, or the set of threads owned by a specified user.

       setpriority(2)
              Set the nice value of a thread, a process group, or the set of threads owned by a specified user.

       sched_setscheduler(2)
              Set the scheduling policy and parameters of a specified thread.

       sched_getscheduler(2)
              Return the scheduling policy of a specified thread.

       sched_setparam(2)
              Set the scheduling parameters of a specified thread.

       sched_getparam(2)
              Fetch the scheduling parameters of a specified thread.

       sched_get_priority_max(2)
              Return the maximum priority available in a specified scheduling policy.

       sched_get_priority_min(2)
              Return the minimum priority available in a specified scheduling policy.

       sched_rr_get_interval(2)
              Fetch the quantum used for threads that are scheduled under the "round-robin" scheduling policy.

       sched_yield(2)
              Cause the caller to relinquish the CPU, so that some other thread be executed.

       sched_setaffinity(2)
              (Linux-specific) Set the CPU affinity of a specified thread.

       sched_getaffinity(2)
              (Linux-specific) Get the CPU affinity of a specified thread.

       sched_setattr(2)
              Set the scheduling policy and parameters of a specified thread.  This (Linux-specific) system call provides a superset of the functionality of sched_setscheduler(2) and sched_setparam(2).
 Manual page sched(7) line 1 (press h for help or q to quit)
```

nice & top

[woodman@woodman]$ nice -n 15 top

```
top - 08:22:05 up 18 days, 21:39,  1 user,  load average: 0.18, 0.24, 0.26
Tasks: 320 total,   1 running, 319 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.3 us,  0.4 sy,  0.0 ni, 99.1 id,  0.0 wa,  0.2 hi,  0.0 si,  0.0 st
MiB Mem :  31862.5 total,  15595.8 free,   6886.2 used,   9380.5 buff/cache
MiB Swap:   8192.0 total,   8192.0 free,      0.0 used.  22994.1 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1970	lwoodman	20	0	3367416	19376	15840	S	4.0	0.1	573:10.38	pulseaudio
44864	lwoodman	20	0	4534052	1.0g	226856	S	1.3	3.2	432:57.26	Web Content
14687	lwoodman	20	0	6043348	1.0g	320900	S	1.0	3.2	1447:34	firefox
45344	lwoodman	20	0	4160224	735616	205712	S	0.7	2.3	227:56.58	Web Content
524733	lwoodman	20	0	693920	52736	36604	S	0.7	0.2	0:13.94	gnome-terminal-
1967	lwoodman	20	0	5277128	409792	104352	S	0.3	1.3	384:32.19	gnome-shell
2415	root	20	0	480780	16736	14228	S	0.3	0.1	5:17.78	abrt-dbus
123533	lwoodman	20	0	3627584	462804	162888	S	0.3	1.4	137:50.90	Web Content
545749	**lwoodman**	**35**	**15**	**235120**	**5236**	**4256**	**R**	**0.3**	**0.0**	**0:00.04**	**top**
1	root	20	0	175180	17544	11368	S	0.0	0.1	0:13.65	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:01.23	kthreadd
3	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	rcu_gp
4	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	rcu_par_gp
6	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	kworker/0:0H-events_highpri
9	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	mm_percpu_wq

chrt/sched_setscheduler

- **chrt**
  - Command to set scheduler policy(man chrt)

```
CHRT(1)                                                                 User Commands

NAME
       chrt - manipulate the real-time attributes of a process

SYNOPSIS
       chrt [options] priority command [argument...]
       chrt [options] -p [priority] pid

DESCRIPTION
       chrt sets or retrieves the real-time scheduling attributes of an existing pid, or runs command with the given attributes.

POLICIES
       -o, --other
              Set scheduling policy to SCHED_OTHER.  This is the default Linux scheduling policy.

       -f, --fifo
              Set scheduling policy to SCHED_FIFO.

       -r, --rr
              Set scheduling policy to SCHED_RR.  When no policy is defined, the SCHED_RR is used as the default.

       -b, --batch
              Set scheduling policy to SCHED_BATCH (Linux-specific, supported since 2.6.16).  The priority argument has to be set to zero.

       -i, --idle
              Set scheduling policy to SCHED_IDLE (Linux-specific, supported since 2.6.23).  The priority argument has to be set to zero.
```

- **sched_setscheduler()**
  - syscall to set scheduler policy(man sched_setscheduler)

```
SCHED_SETSCHEDULER(2)          Linux Programmer's Manual          SCHED_SETSCHEDULER(2)

NAME
       sched_setscheduler, sched_getscheduler - set and get scheduling policy/parameters

SYNOPSIS
       #include <sched.h>

       int sched_setscheduler(pid_t pid, int policy,
                              const struct sched_param *param);

       int sched_getscheduler(pid_t pid);

DESCRIPTION
       The  sched_setscheduler()  system call sets both the scheduling policy and parame-
       ters for the thread whose ID is specified in pid.  If pid equals zero, the  sched-
       uling policy and parameters of the calling thread will be set.
```

Linux - Completely Fair Scheduler (CFS)

- preemptive
- dynamic priority adjusting
  - CPU hog processes get priority penalty
  - long wait processes get priority boost
- Track 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