Definition and Goals

Assign processes to be executed by the processor(s)

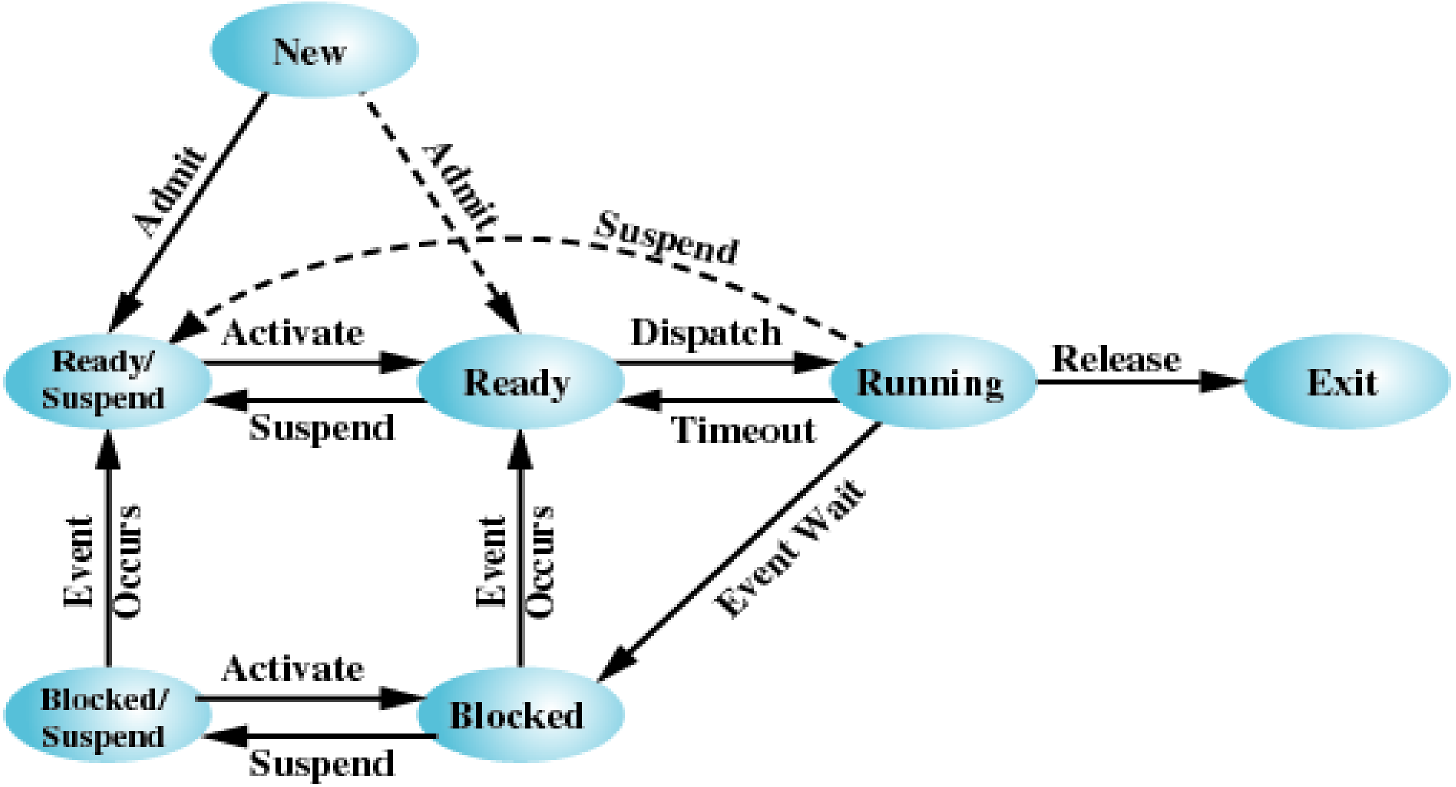
More general: Assign consumers to resources
- Examples: I/O requests ➔ Device-specific queues
- Memory pages ➔ Primary/secondary memory

Goals:
- Throughput, i.e., effectively use processing time
- Response time / fairness, i.e., interactivity of individual processes
- Processor efficiency, i.e., optimal utilization of CPU (as resource)

Conflicting goals: Maximal throughput means unpredictable response time (and vice versa)

Process States and Scheduling

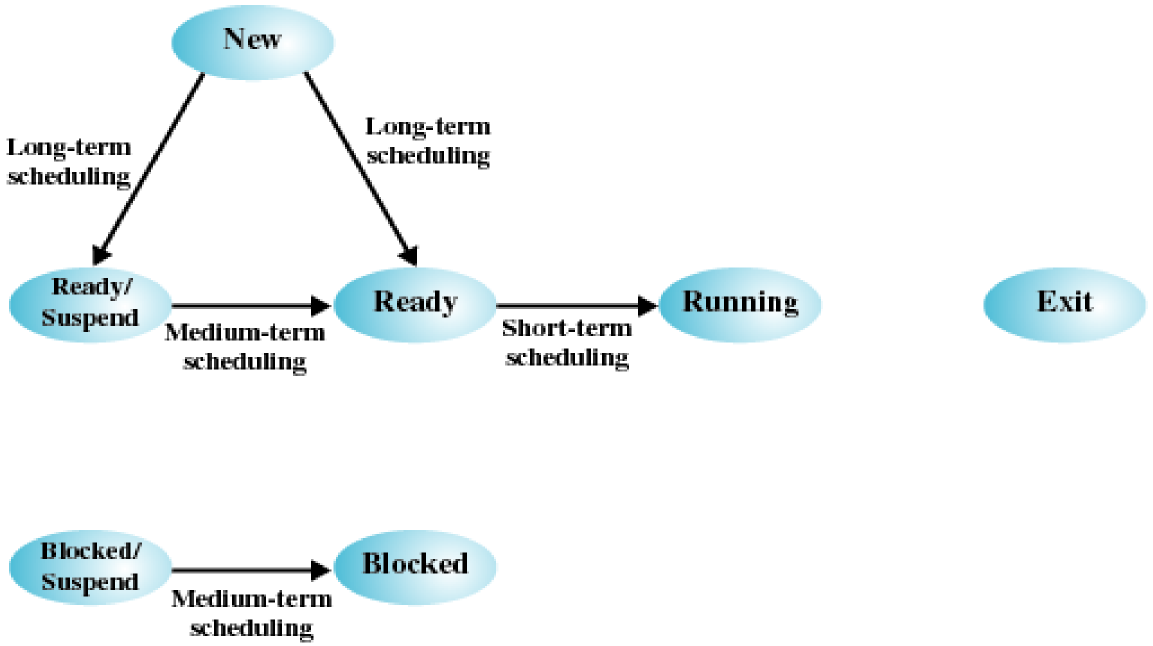
Scheduling decisions correspond to state transitions in process state graph

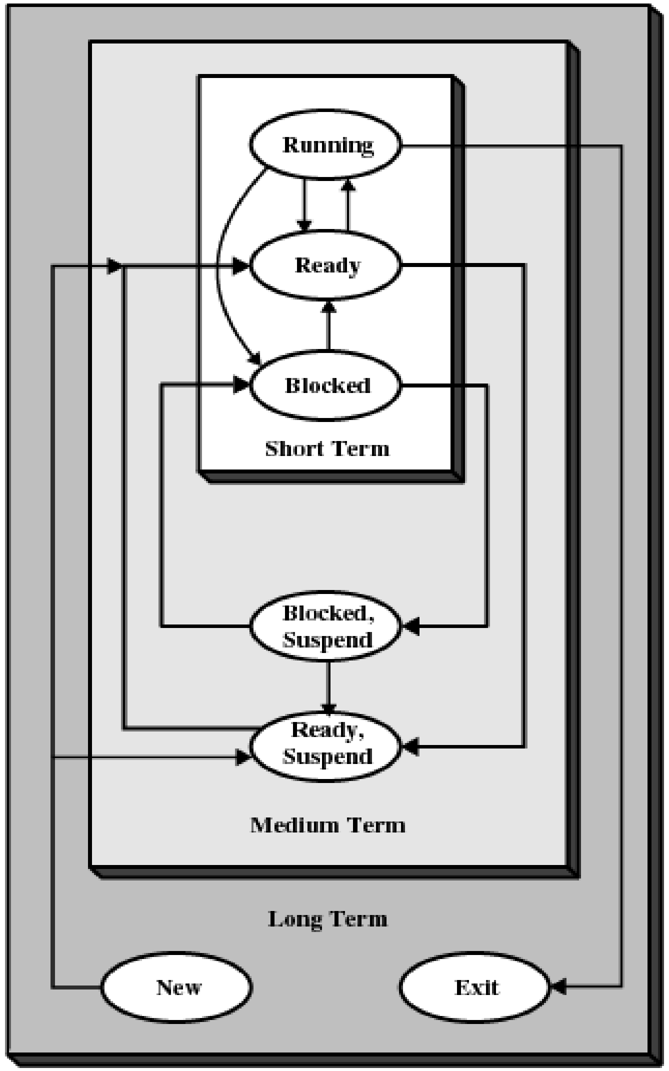


Process States and Scheduling

Scheduling decisions correspond to state transitions in process state graph

- States form hierarchy depending on transition frequency





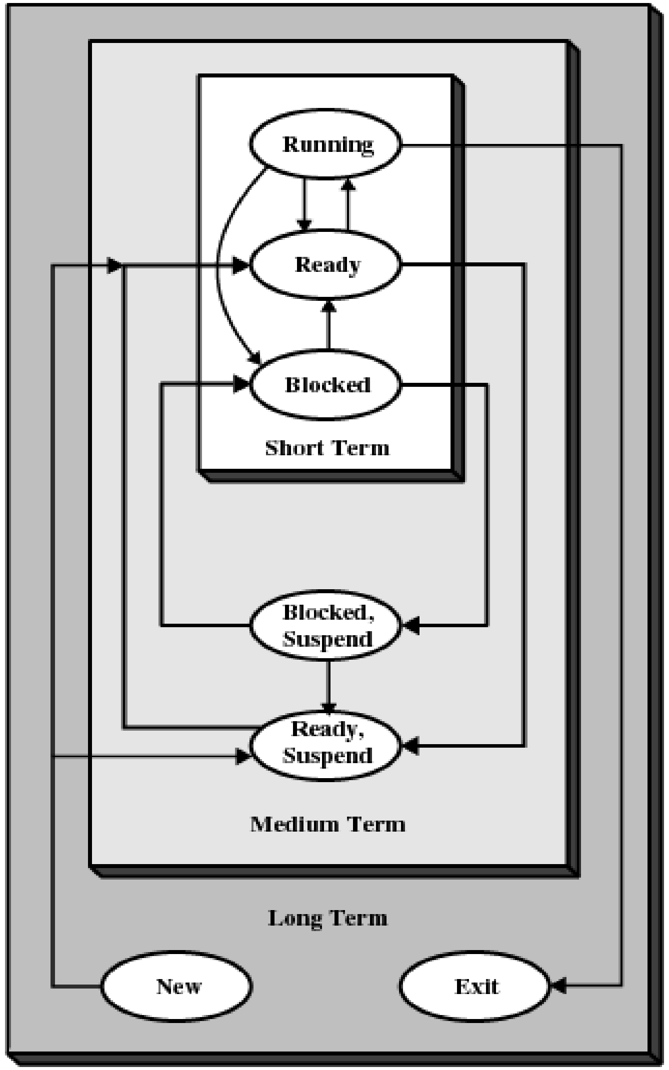
Types of Scheduling

Long-term scheduling

Whether to add process to running queue and execute it

- Determines which programs are admitted to system for processing, e.g., based on user
- Specifies degree of multiprogramming, i.e., maximal number of processes
- The more processes, the smaller percentage of time each process is executed

➢ How many processes should be allowed?



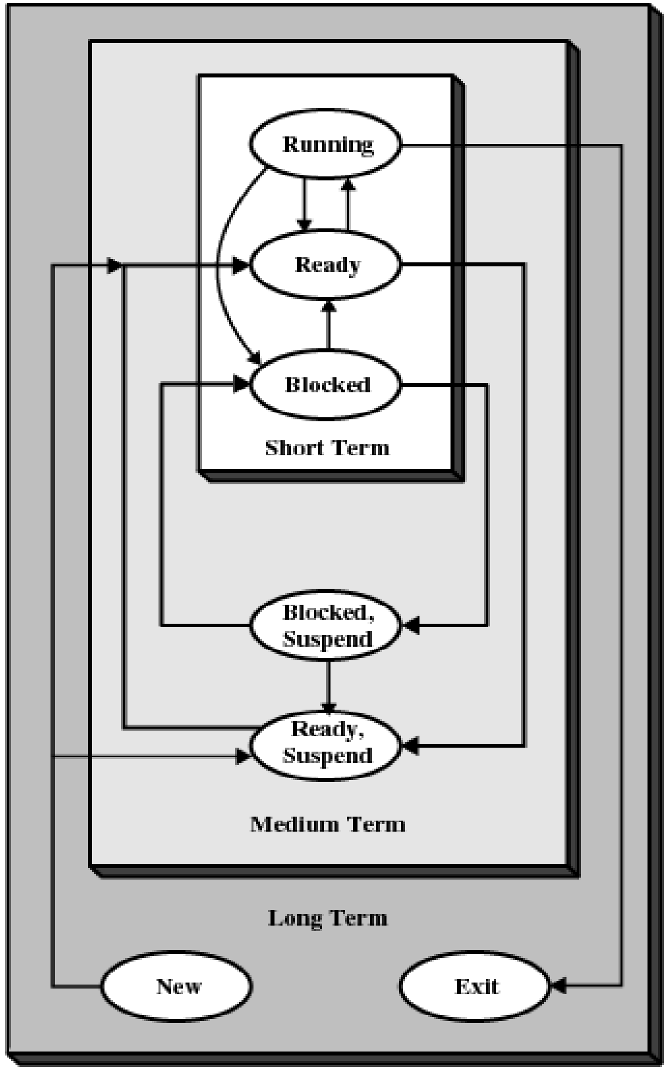
Types of Scheduling

Medium-term scheduling

Whether to add/remove existing process (that is only partially in primary memory)

- Part of swapping function
- Based on need to dynamically manage degree of multiprogramming (considering available resources)

➢ Should processes be swapped in or out? If so, which ones?



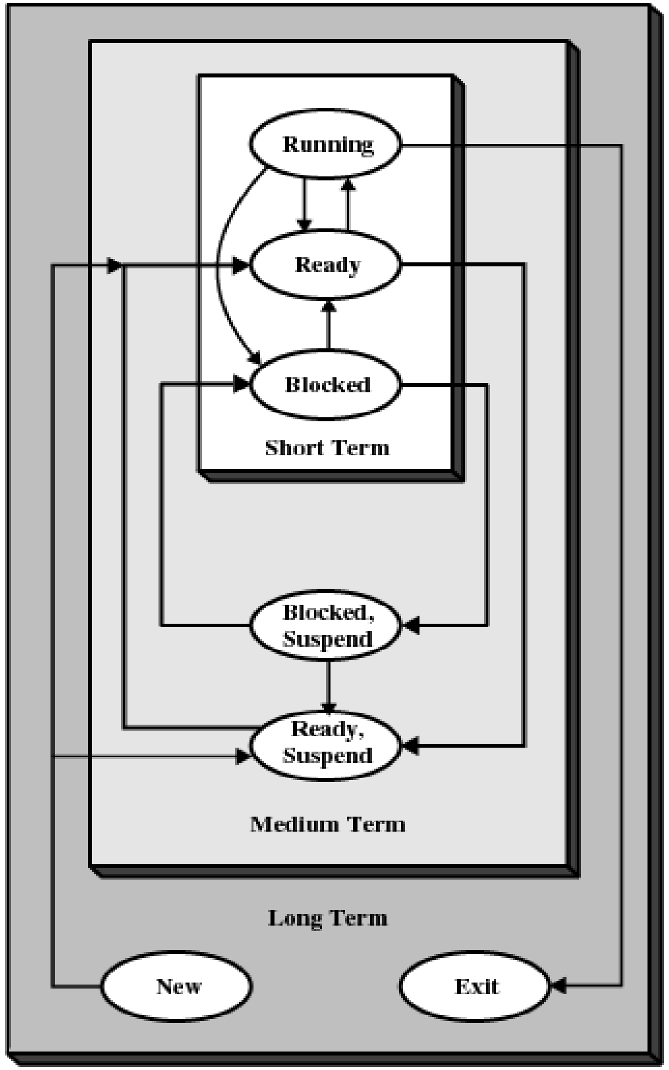
Types of Scheduling

Short-term scheduling

Which one of fully available processes to run

- Known as “dispatcher”
- Executes most frequently
  - Overhead / algorithmic complexity matters
- Invoked when event occurs (clock interrupts, I/O interrupts, operating system calls, signals)

➢ Whose turn is it?

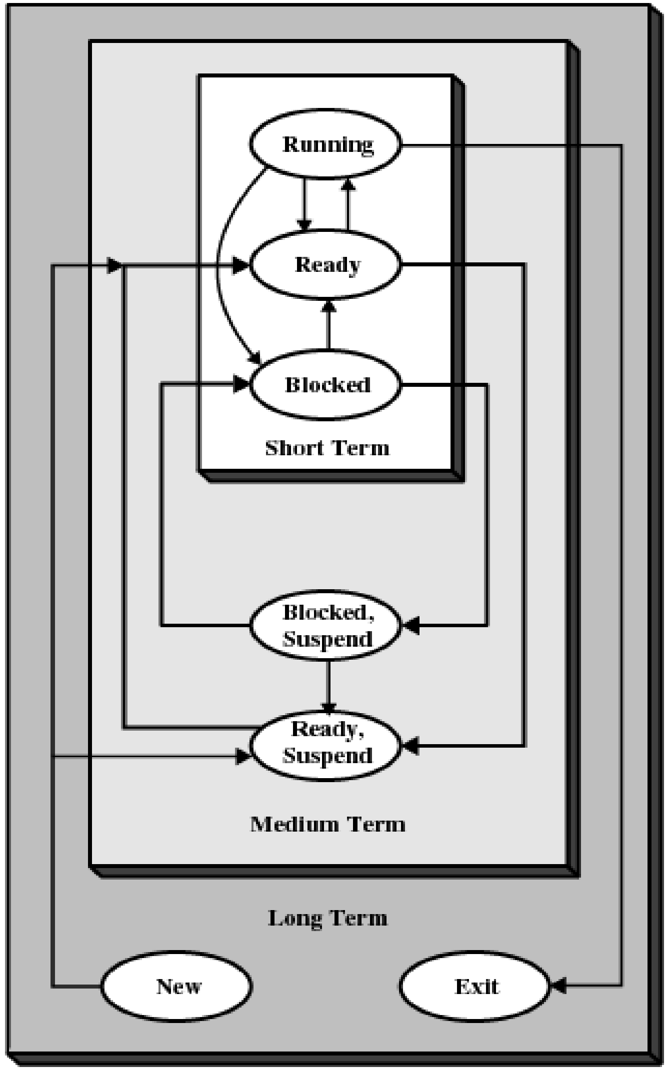


Types of Scheduling

I/O scheduling

Which I/O request (of which process) to dispatch to I/O device for handling

➢ Consider state of external device



Short-Term Scheduling Criteria

User-oriented:

- Response time: elapsed time between submission of a request until there is output
- ➢ Interactivity: user *perceives* system as “responsive”

System-oriented (hardware and resources):

- Effective and efficient utilization of processor

Performance-related:

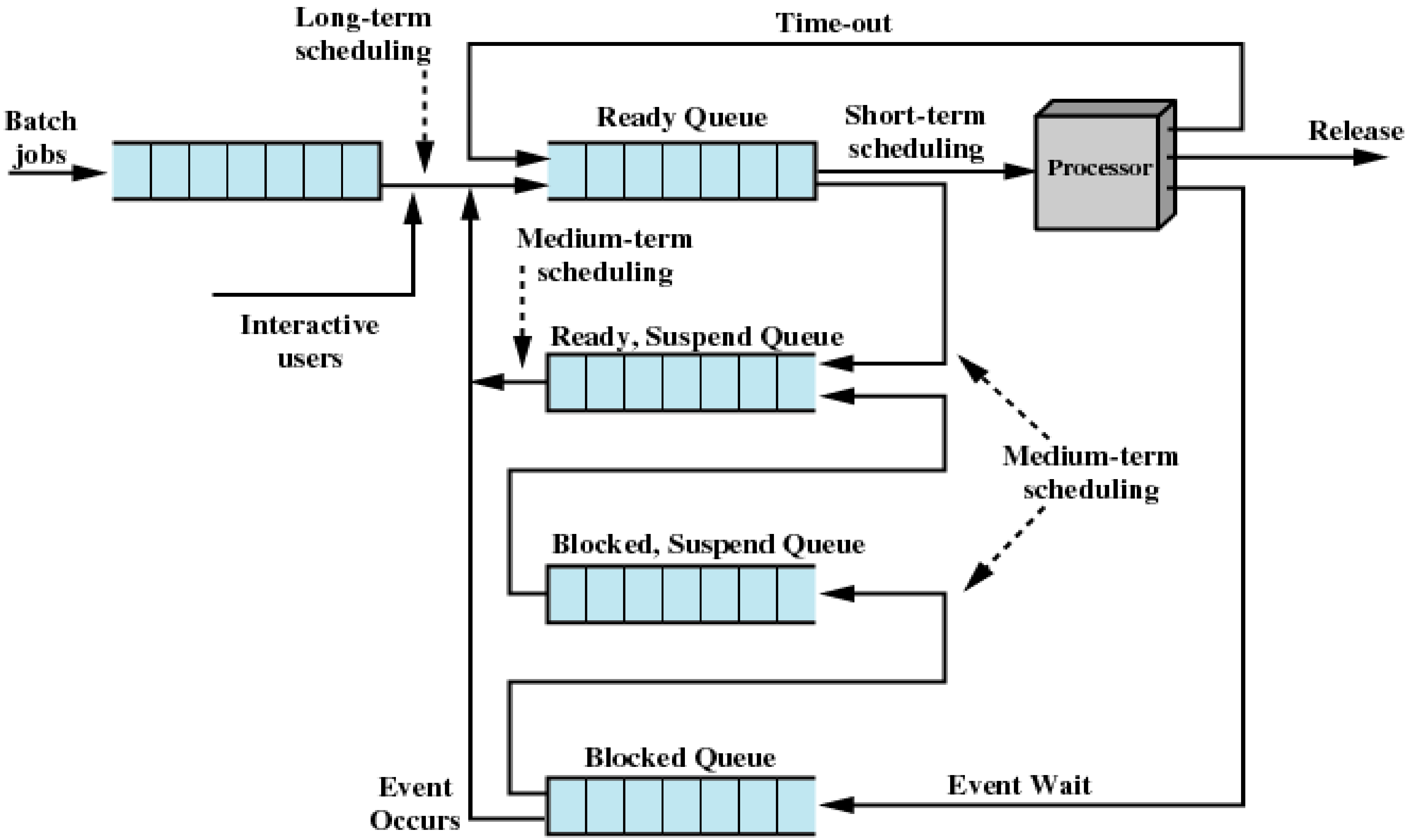
- Quantitative / measurable properties
- Examples: response time, throughput

Non-functional:

- Algorithmic properties
- Examples: predictability, fairness

	Performance-related	Non-functional
User-oriented	• Turnaround time • Response time • Deadlines	• Predictability
System-oriented	• Throughput • Processor utilization	• Fairness • Enforcing priorities • Balancing resources

Scheduler Implementation: Queuing



Scheduling Decision Modes

Non-preemptive

- Current process explicitly yields CPU
  - ➢ Cooperative multitasking, e.g., Windows (<95), Mac OS (<X)
- Once a process is in running state, it will continue until it terminates or blocks itself for I/O

Preemptive

- OS may interrupt current process
  - Transparent to process
  - ➢ Preemptive multitasking, e.g., Windows (≥95), Mac OS X, Unix
- Allows for better scheduling since no process can monopolize CPU

Priorities

Some processes are more *important* than other processes, i.e., should get more CPU cycles than others

- Similar for other resources

Scheduling is controlled by per-process priorities

- OS internal vs. user-visible priorities

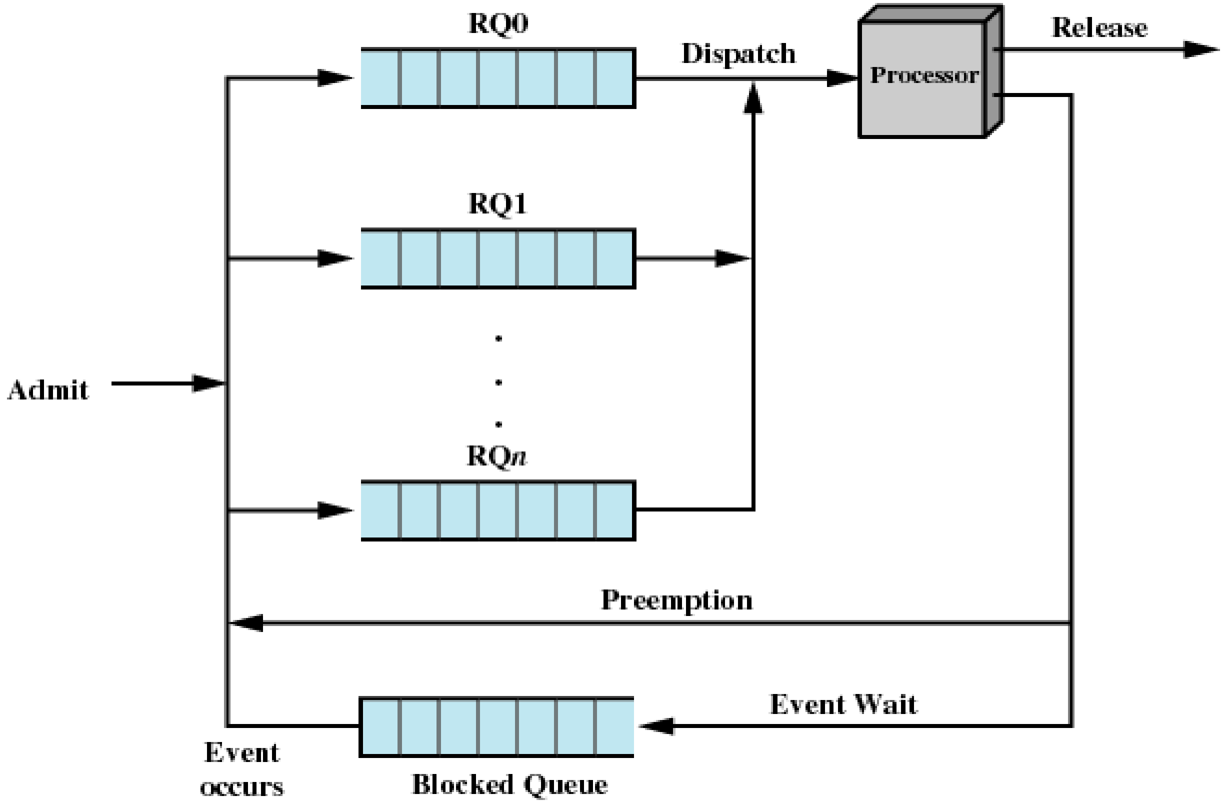
Scheduler will always choose a process of higher priority over one of lower priority

➢ Lower-priority processes may suffer starvation, i.e. are never scheduled and do not make *any* progress

Priority Implementation: Queuing

Have multiple ready queues to represent each level of priority

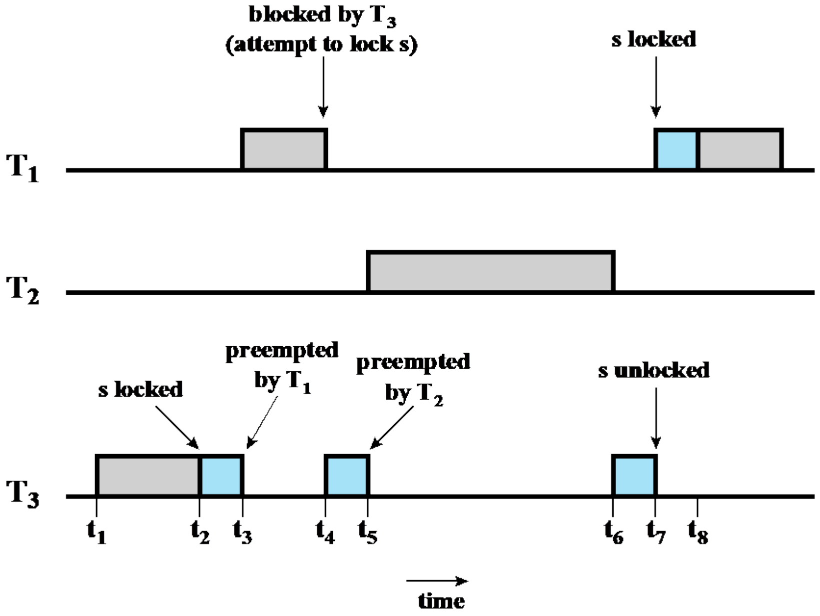
Move process data between queues according to scheduling algorithm



Priority Inversion and Inheritance

Problem: Priority Inversion

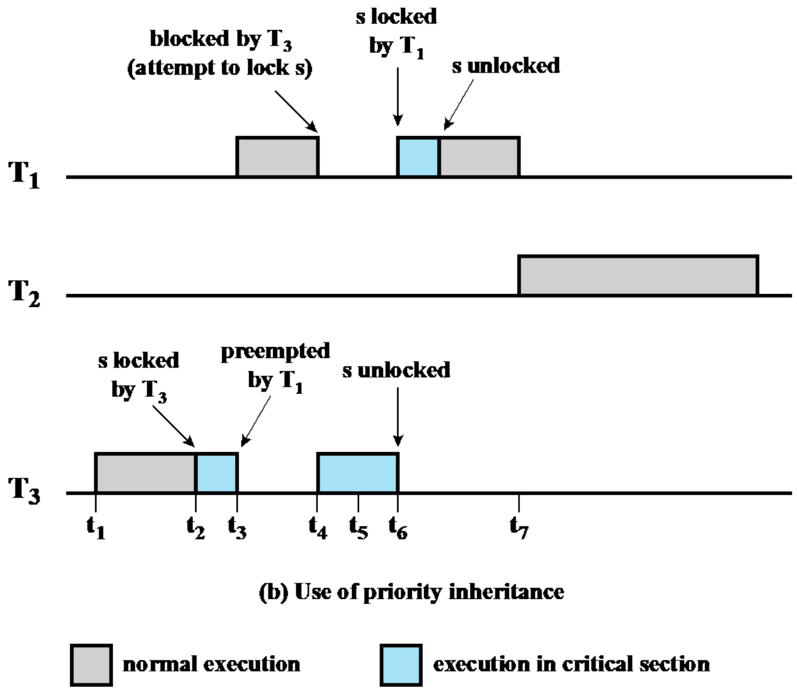
Occurs when circumstances within the system force a higher priority task to wait for a lower priority task



(a) Unbounded priority inversion

Solution: Priority Inheritance

Lower-priority task inherits priority of any higher priority task pending on a resource they share



(b) Use of priority inheritance

Scheduling Algorithm Classes

Non-preemptive

- First-Come-First-Served (FCFS)
- Shortest Process Next (SPN)
- Highest Response Ratio Next (HRRN)

Preemptive

- Shortest Remaining Time (SRT)
- Round-Robin
- Feedback

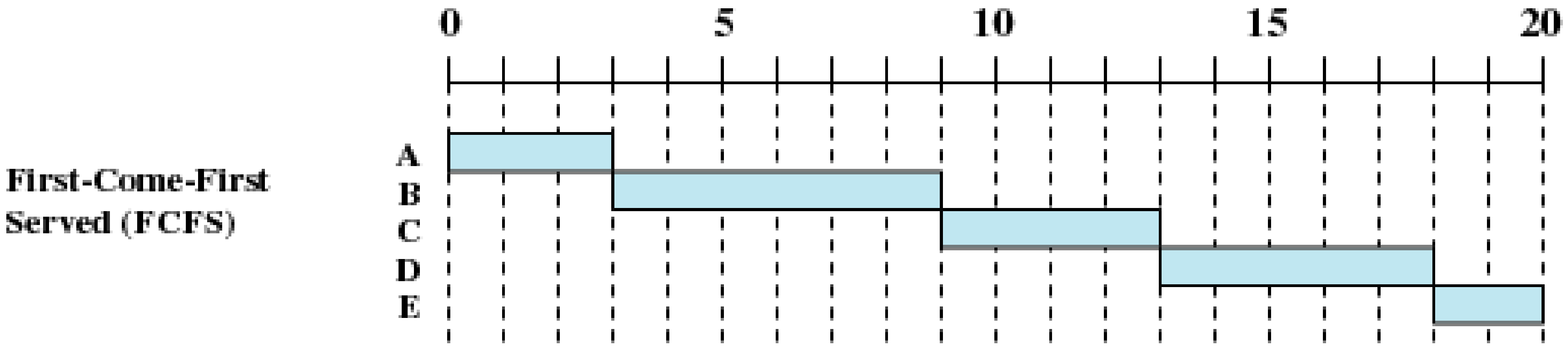
Example workload

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

First-Come-First-Served (FCFS)

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

New process placed at end of Ready queue
When current process ceases to execute, oldest process in the Ready queue is selected



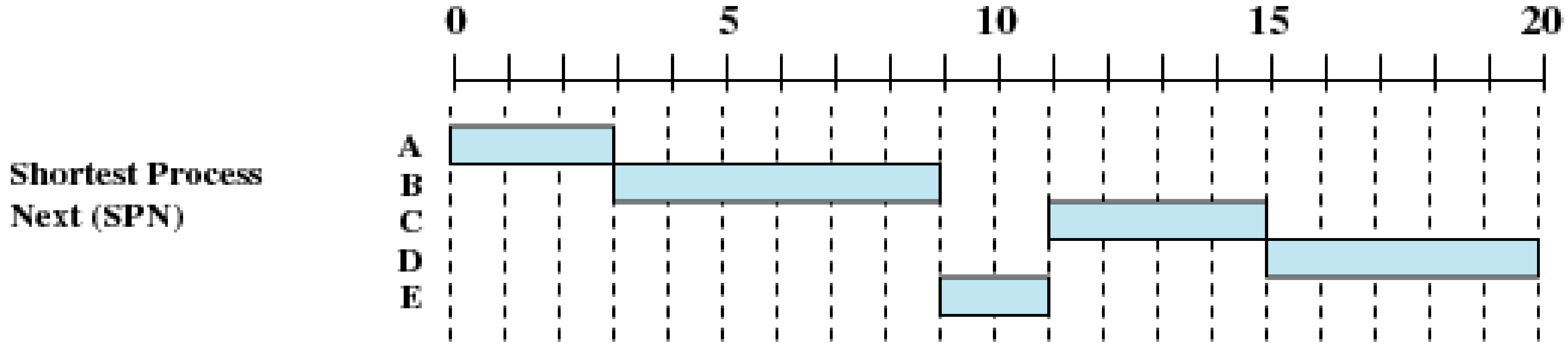
- Short process may have to wait a very long time before it can execute
- Poor response time / interactivity
- Favors CPU-bound processes
  - I/O processes have to wait until CPU-bound process completes, since I/O processes frequently call into OS

Shortest Process Next (SPN)

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

Process with shortest expected processing time is selected
- OS may abort processes with incorrect time estimates

Short processes jump ahead of longer processes

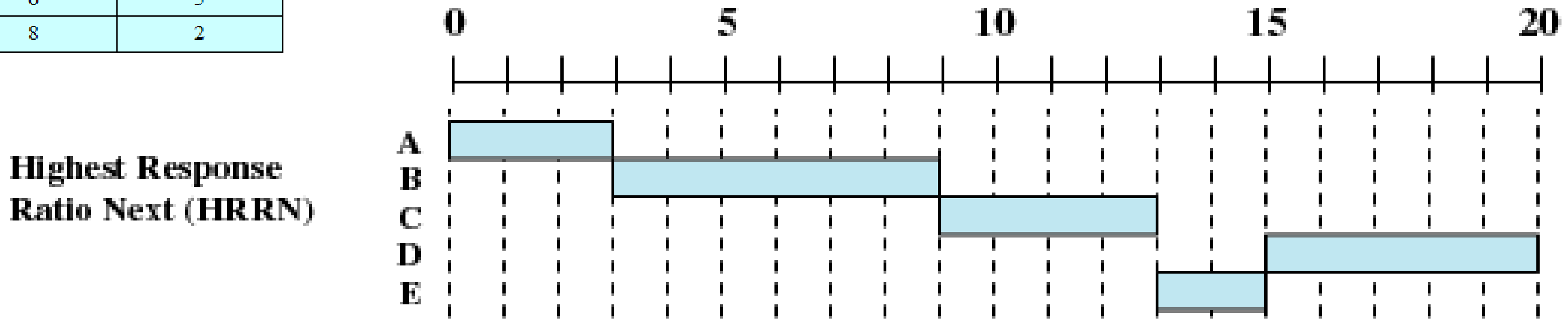


- ➢ Improves interactivity (based on assumption that short processes are due to user interaction)
- ➢ Predictability of longer processes is reduced
- ➢ Possibility of starvation for longer processes

Highest Response Ratio Next (HRRN)

Choose next process with the highest ratio $\dfrac{\textit{time spent waiting} + \textit{expected service time}}{\textit{expected service time}}$

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2



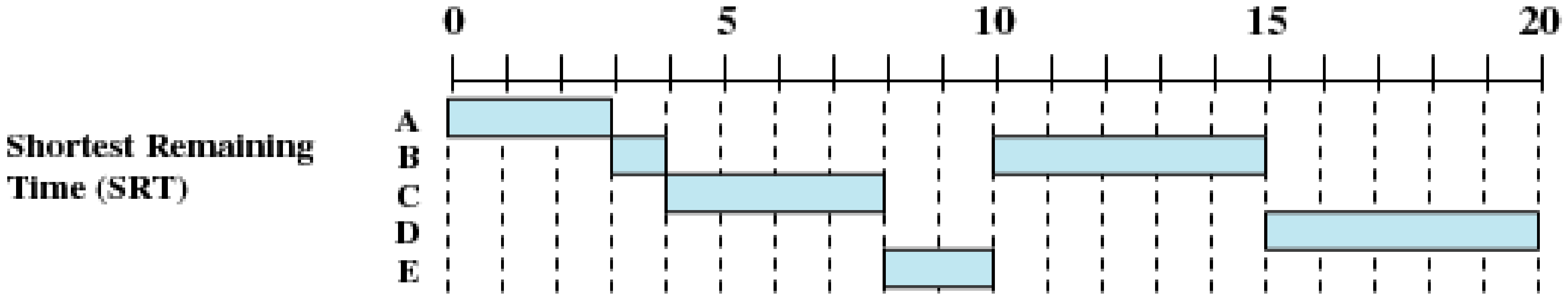
- ➢ Even long process will run eventually
- ➢ Generally, predictable response times not feasible without preemption

Shortest Remaining Time (SRT)

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

Ready queue is sorted by remaining processing time
- Requires estimate of remaining processing time

New processes may preempt current process upon arrival
- Preemptive version of shortest process next policy



- ➢ Improved response time of short processes by using preemption
  - Limited additional overhead due to process switches upon process creation
- ➢ But what happens to interactive requests that don't spawn a new process?

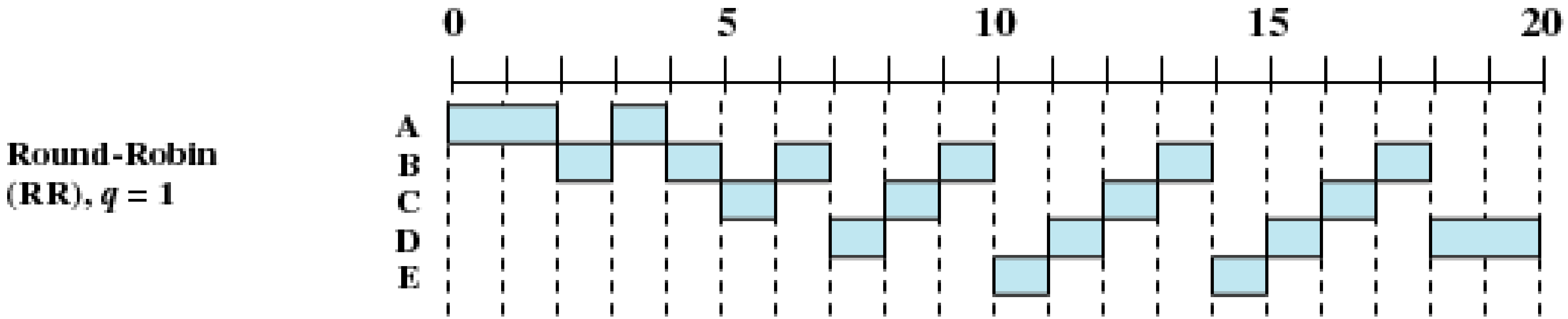
Round-Robin

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

Each process may use CPU for given amount of time

- Process preemption based on clock interrupt generated at periodic intervals, i.e., time slicing
- Time quantum *q* as tunable parameter

When interrupt occurs, currently running process is placed in Ready queue, next ready job is selected



- ➢Initial support for interactivity
- ➢Scheduling overhead (scheduling decision, process switch)
- ➢ Tradeoff between interactivity and efficiency, directly tunable by *q*
- ➢Problematic for I/O processes that hardly ever use full quantum

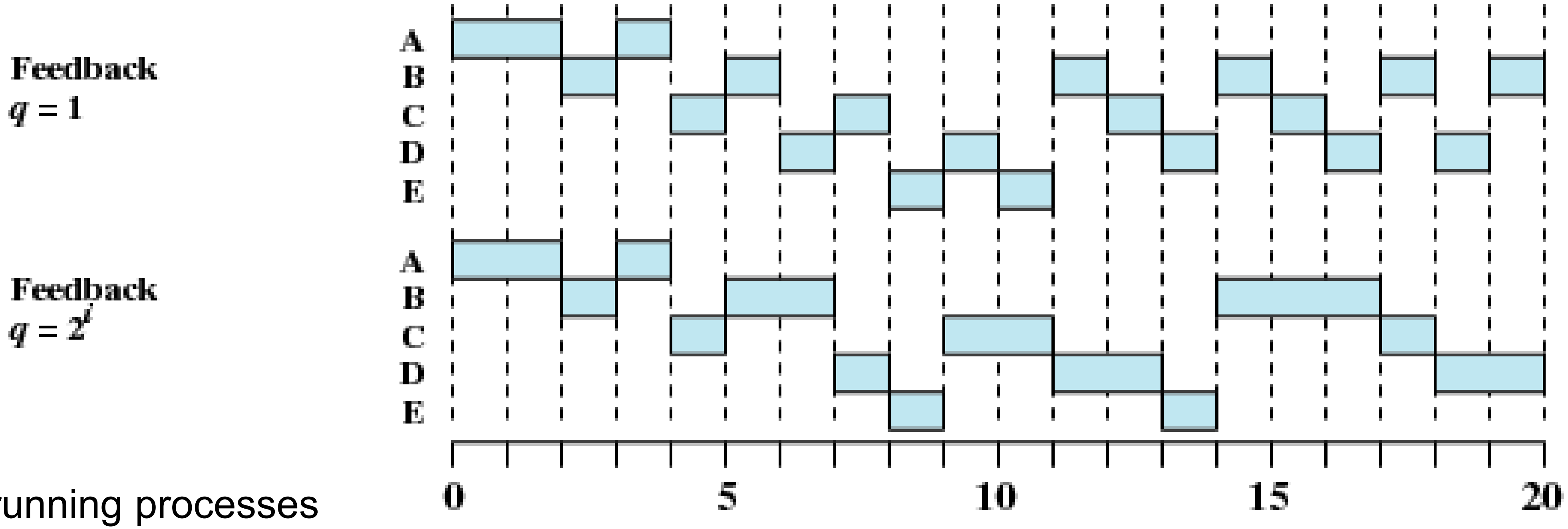
Feedback

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

Processes start in the queue with highest priority RQ_0 and move to queues with lower priority after each time slice

- Multiple queues with different priorities

For fairness, allow longer time slices q for queues RQ_i



- ➢ Penalize long running processes
- ➢ No need to know remaining execution time of process

Qualitative Comparison of Policies

	Selection Function	Decision Mode	Throughput	Response Time	Overhead	Effect on Processes	Starvation
FCFS	$\max[w]$	Nonpreemptive	Not emphasized	May be high, especially if there is a large variance in process execution times	Minimum	Penalizes short processes; penalizes I/O bound processes	No
Round Robin	constant	Preemptive (at time quantum)	May be low if quantum is too small	Provides good response time for short processes	Minimum	Fair treatment	No
SPN	$\min[s]$	Nonpreemptive	High	Provides good response time for short processes	Can be high	Penalizes long processes	Possible
SRT	$\min[s - e]$	Preemptive (at arrival)	High	Provides good response time	Can be high	Penalizes long processes	Possible
HRRN	$\max\left(\frac{w+s}{s}\right)$	Nonpreemptive	High	Provides good response time	Can be high	Good balance	No
Feedback	(see text)	Preemptive (at time quantum)	Not emphasized	Not emphasized	Can be high	May favor I/O bound processes	Possible

w = time spent waiting, e = time spent in execution so far, s = total service time required by process, including e

Quantitative Comparison of Policies

	Process	A	B	C	D	E	
	Arrival Time	0	2	4	6	8	
	Service Time (T_s)	3	6	4	5	2	Mean
FCFS	Finish Time	3	9	13	18	20	
	Turnaround Time (T_r)	3	7	9	12	12	8.60
	T_r/T_s	1.00	1.17	2.25	2.40	6.00	2.56
RR $q = 1$	Finish Time	4	18	17	20	15	
	Turnaround Time (T_r)	4	16	13	14	7	10.80
	T_r/T_s	1.33	2.67	3.25	2.80	3.50	2.71
RR $q = 4$	Finish Time	3	17	11	20	19	
	Turnaround Time (T_r)	3	15	7	14	11	10.00
	T_r/T_s	1.00	2.5	1.75	2.80	5.50	2.71
SPN	Finish Time	3	9	15	20	11	
	Turnaround Time (T_r)	3	7	11	14	3	7.60
	T_r/T_s	1.00	1.17	2.75	2.80	1.50	1.84
SRT	Finish Time	3	15	8	20	10	
	Turnaround Time (T_r)	3	13	4	14	2	7.20
	T_r/T_s	1.00	2.17	1.00	2.80	1.00	1.59
HRRN	Finish Time	3	9	13	20	15	
	Turnaround Time (T_r)	3	7	9	14	7	8.00
	T_r/T_s	1.00	1.17	2.25	2.80	3.5	2.14
FB $q = 1$	Finish Time	4	20	16	19	11	
	Turnaround Time (T_r)	4	18	12	13	3	10.00
	T_r/T_s	1.33	3.00	3.00	2.60	1.5	2.29
FB $q = 2^i$	Finish Time	4	17	18	20	14	
	Turnaround Time (T_r)	4	15	14	14	6	10.60
	T_r/T_s	1.33	2.50	3.50	2.80	3.00	2.63

Multiprocessor Scheduling

Assignment of processes to processors

- Permanently assign process to a processor
- Treat processors as a pooled resource and assign process to processors on demand
  - Possibly move running process between processors (expensive!)

Architectures

- Global queue: schedule to any available processor
- Master/slave: Key kernel functions always run on particular processor, master is responsible for scheduling
- Peer: Operating system can execute on any processor, each processor does self-scheduling

Use of multiprogramming on individual processors

Actual dispatching of processes

Real-Time Scheduling

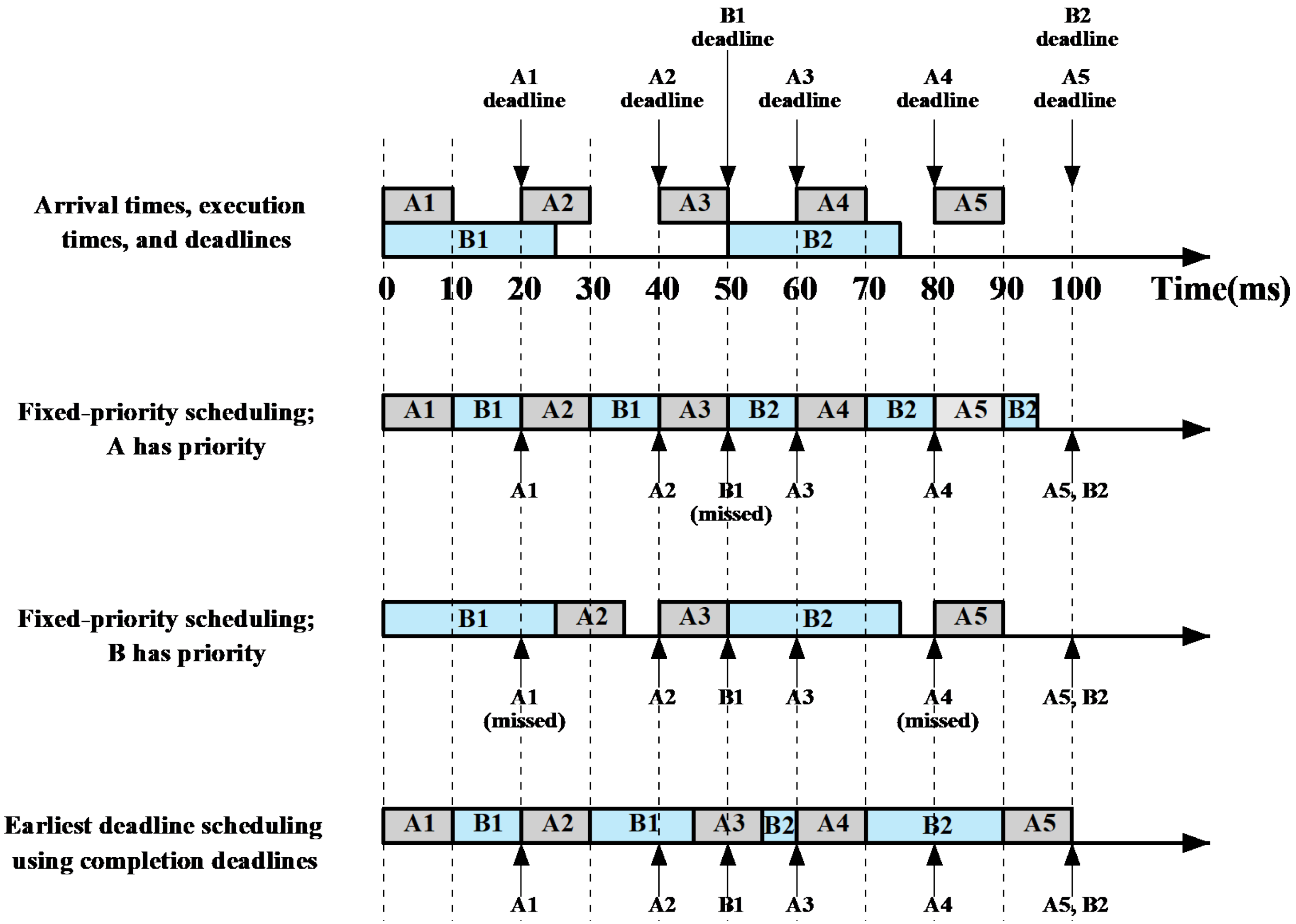
Correctness of system depends
- on logical result of the computation
- **AND** on time at which the results are produced

Tasks or processes attempt to control or react to events that take place in outside world

Examples:
- Control of laboratory experiments
- Process control in industrial plants
- Robotics
- Air traffic control
- Telecommunications
- Military command and control systems

Real-time applications are *not concerned with speed* but with *completing tasks*

Real-Time Scheduling: Examples



Examples: Traditional UNIX Scheduling

Multilevel feedback using round robin within each priority queue

If running process does not block or complete within one second, it is preempted

Priorities are recomputed once per second

Base priority (set upon process creation) divides all processes into fixed bands of priority levels

Examples: UNIX SVR4 Scheduling

Preemptable static priority scheduler

Introduces set of 160 priority levels divided into three priority classes

- Highest preference to real-time processes
- Next-highest to kernel-mode processes
- Lowest preference to other user-mode processes

In-kernel preemption points, i.e. long running kernel operations may be preempted

SVR4 Priority Classes:

- Real time (159 – 100)
- Kernel (99 – 60)
- Time-shared (59-0)

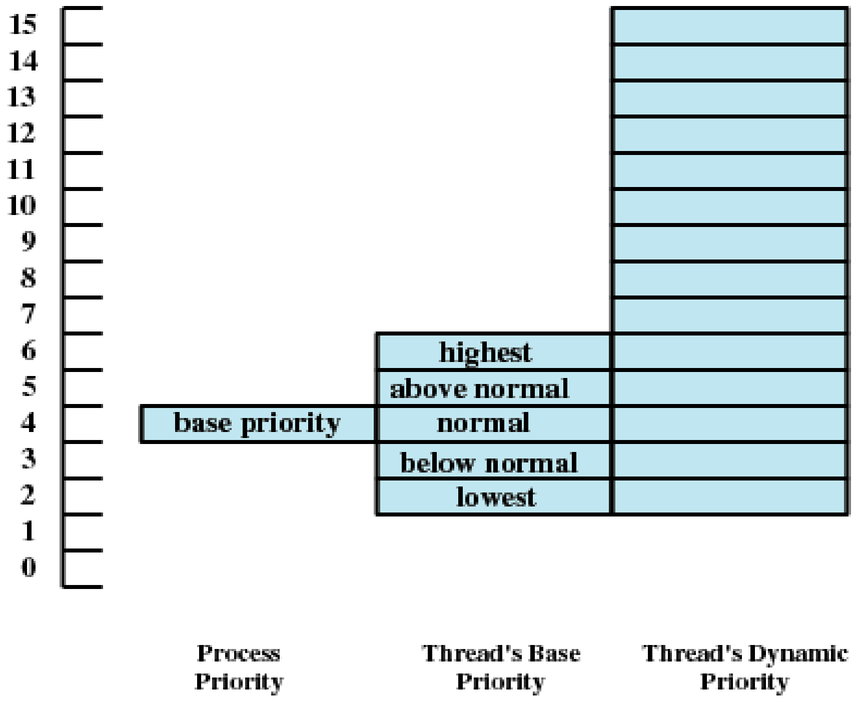
Priority Class	Global Value	Scheduling Sequence
Real-time	159 100	first
Kernel	99 . . 60	
Time-shared	59 0	last

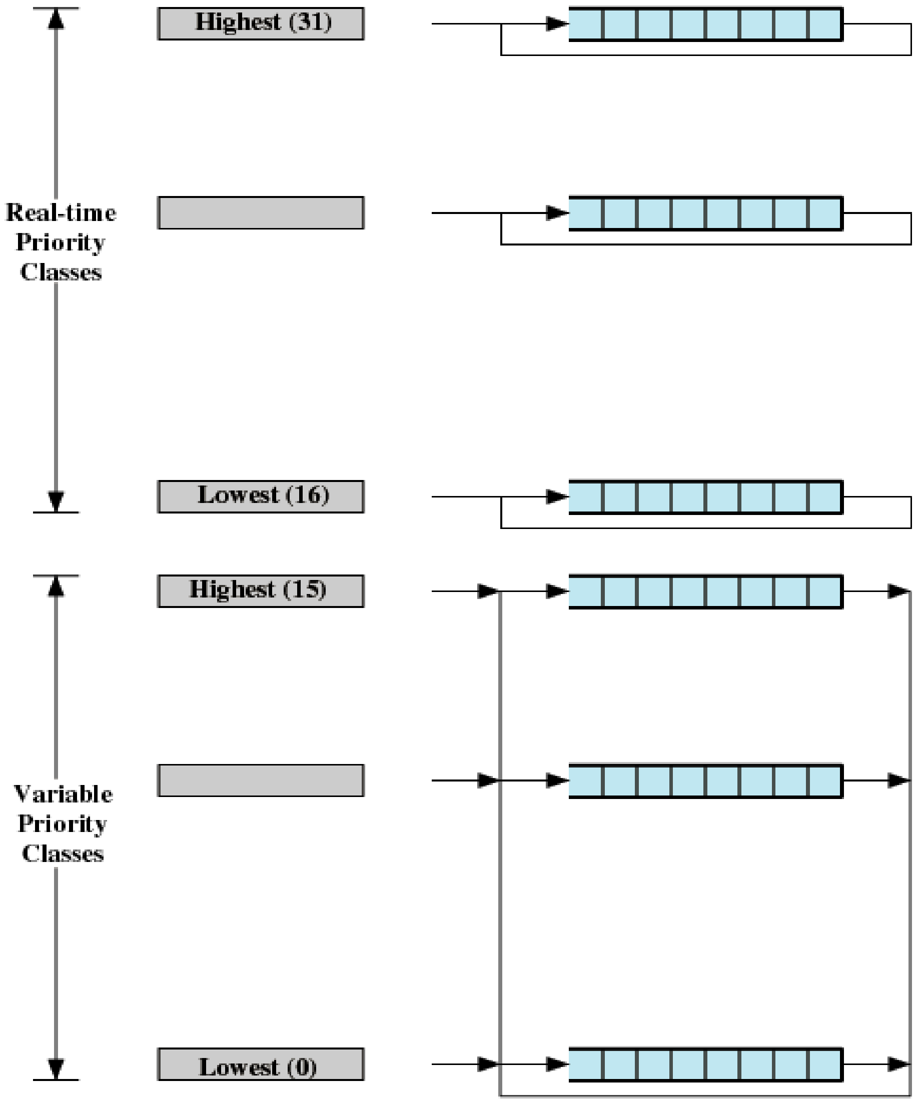
Examples: Windows Scheduling

Priorities organized into two bands or classes

- Real time
- Variable

Priority-driven preemptive scheduler within each class

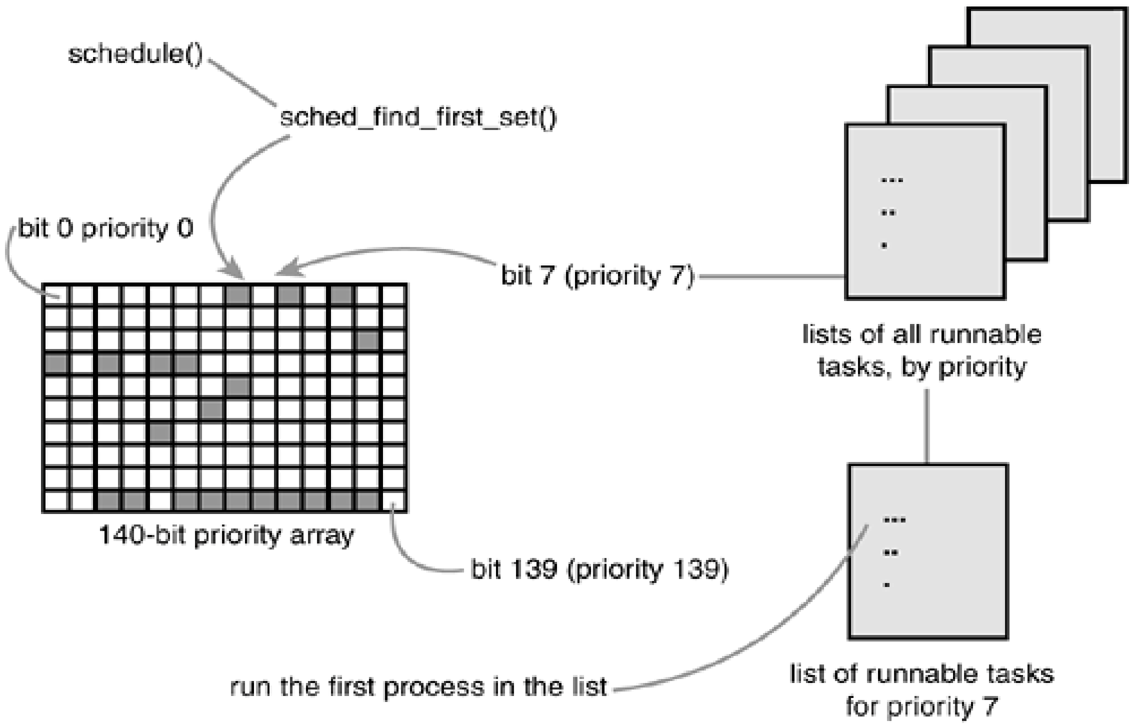




Example: Linux O(1) Scheduling

Scheduling algorithm needs to scale with number of processes

- Variable overhead unacceptable for real-time systems



➢ Linux O(1) scheduler

- Active/expired bit arrays for priorities; one list per priority
- Priority assigned based on
  - Static (process) priority
  - Heuristics to determine interactivity requirements, e.g. CPU- vs. I/O-bound
- Process timeslice (i.e. runtime in relation to other processes) calculated when process moves from active to expired state
- Switch from active to expired bit array when all processes have used their timeslice

➢ Scheduling decision in constant time

Related System Calls (Linux)

`int sched_yield(void)`

- Voluntarily yield processor, e.g. when waiting for input

`int getpriority(int which, int who)`
`int setpriority(int which, int who, int prio)`

- Get/set priority of user, group or process (**`which`**) with ID **`who`**
- Library interface: **`int nice(int inc)`**
  - Increment how nice you are; only root is allow not to be nice

`int sched_get_priority_max(int policy)`
`int sched_get_priority_min(int policy)`

- Returns max/min priority values for given scheduling **`policy`**

Related System Calls (Linux, cont.)

```
int sched_setscheduler(pid_t pid, int policy,
 conststruct sched_param *param)
int sched_getscheduler(pid_t pid)
```

- Controls which scheduling policy to use for a process
- Policies are `SCHED_BATCH`, `SCHED_FIFO`, `SCHED_RR` and `SCHED_OTHER`

```
int sched_setparam(pid_t pid, const struct sched_param
 *param)
int sched_getparam(pid_t pid, struct sched_param *param)
```

- Get/set policy specific scheduling parameters

```
int sched_setaffinity(pid_t pid, unsigned int
 cpusetsize, cpu_set_t *mask)
int sched_getaffinity(pid_t pid, unsigned int
 cpusetsize, cpu_set_t *mask)
```

- Controls on which CPU in multi-processor system a process can/should run