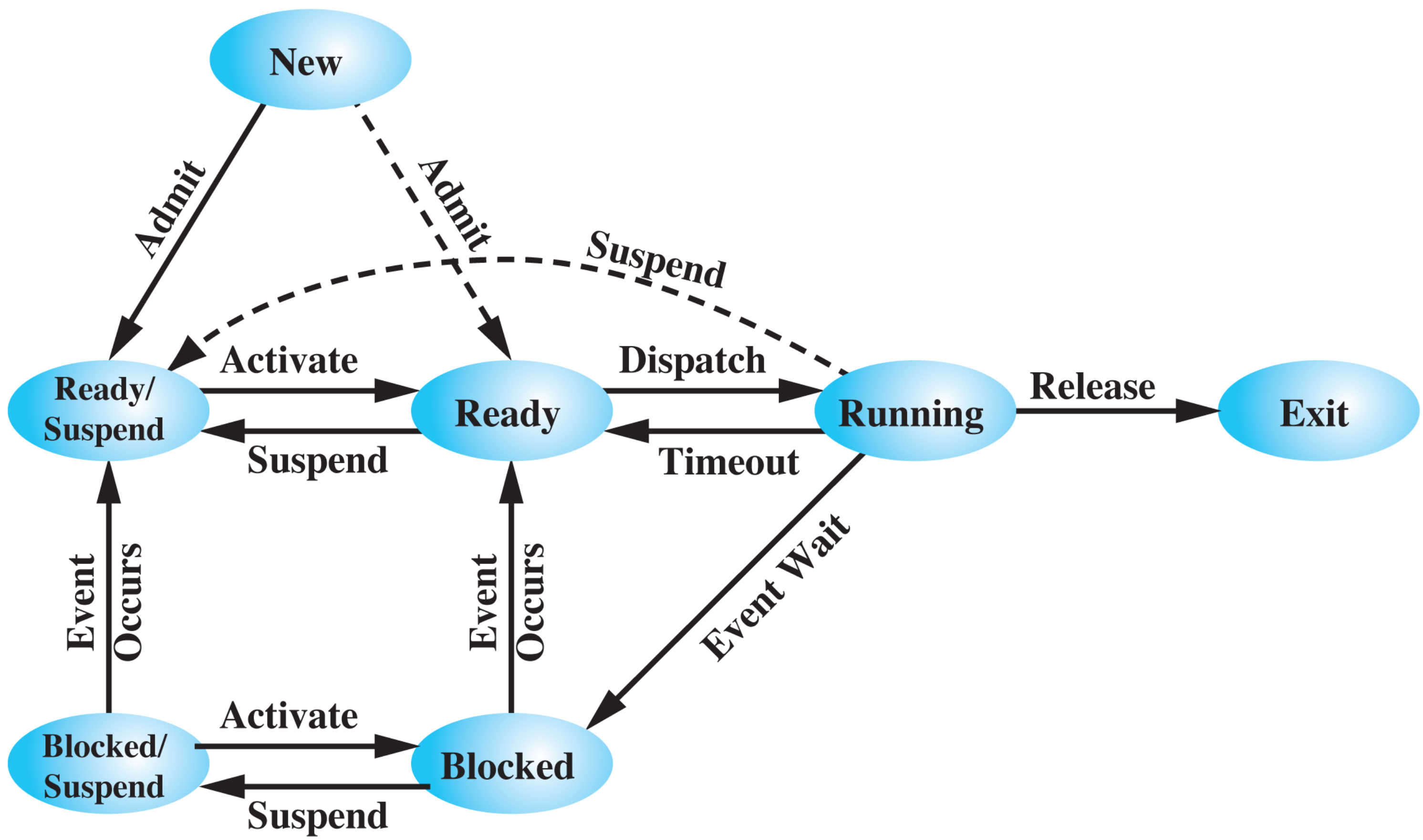
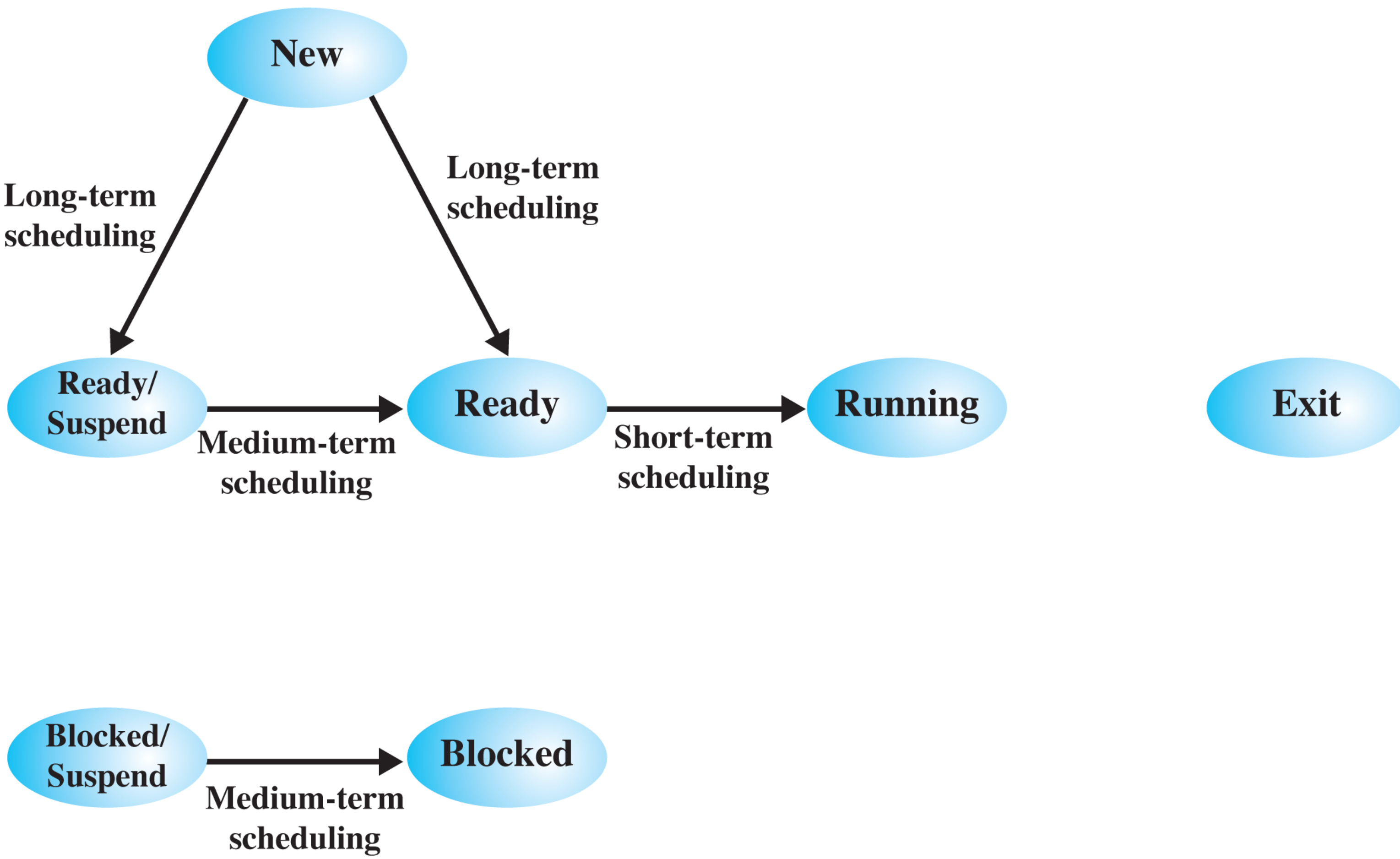
CPU SCHEDULING

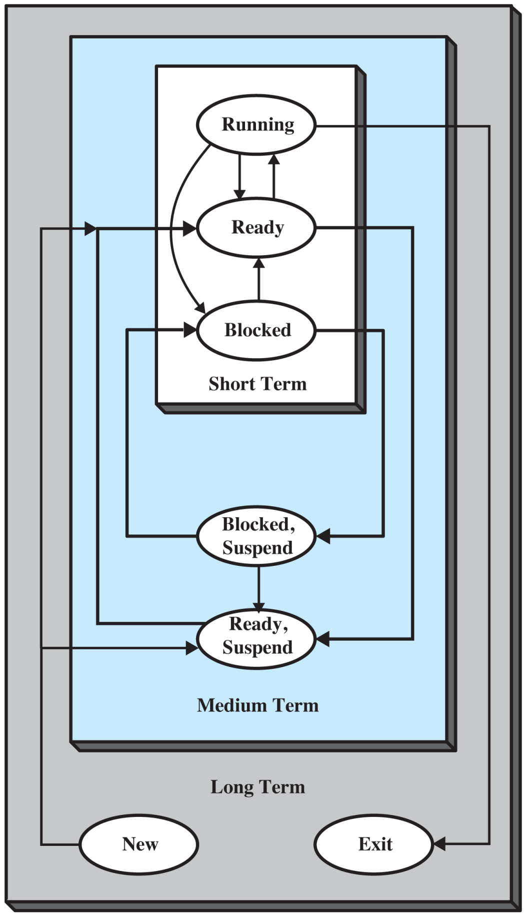
- Aims to assign processes to be executed by the CPU in a way that meets system objectives such as response time, throughput, and processor efficiency
- Broken down into three separate functions:
  - Long term scheduling = the decision to add to the pool of processes being executed
  - Medium term scheduling = the decision to add to the number of processes that are partially or fully into main memory
  - Short term scheduling = decides which available process will be executed by the CPU
  - I/O scheduling = decides which process' pending I/O request is handled by the available I/O devices

CPU Scheduling (cont'd)

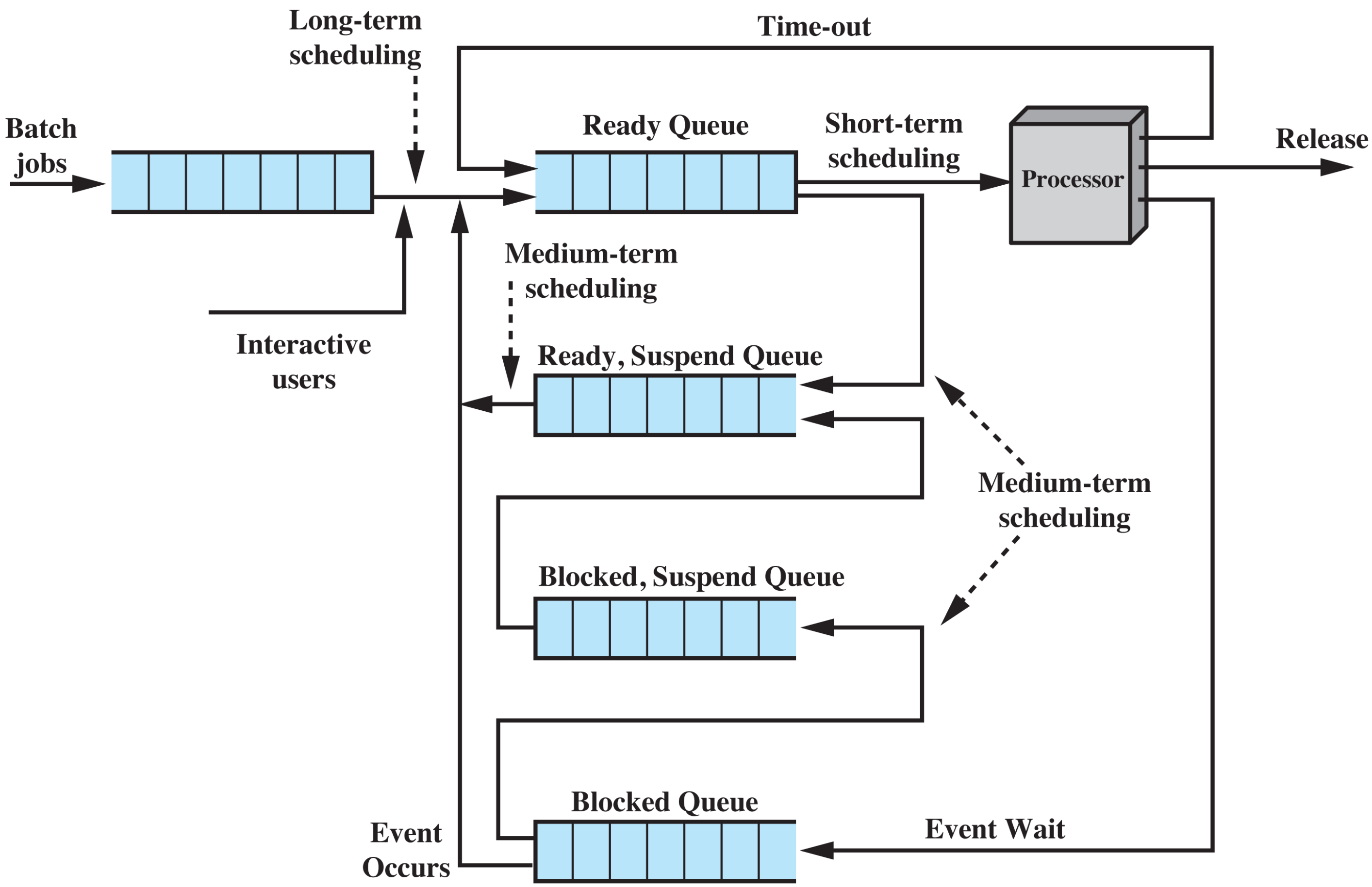



New
Long-term scheduling
Long-term scheduling
Ready/ Suspend
Medium-term scheduling
Ready
Short-term scheduling
Running
Exit
Blocked/ Suspend
Medium-term scheduling
Blocked

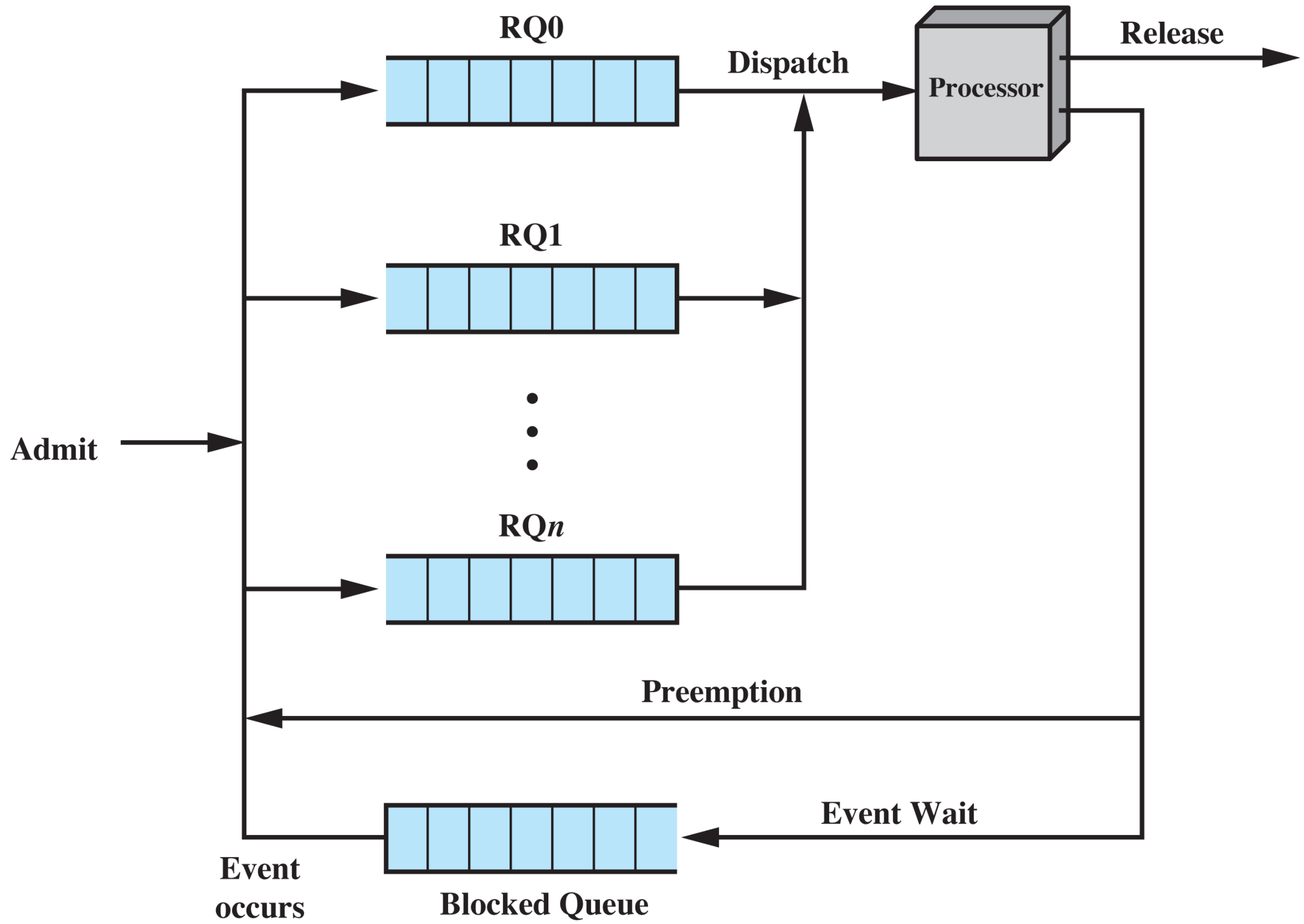
Nested Scheduling Functions



Queuing Diagram



SHORT-TERM PRIORITY SCHEDULING



Long- and Medium-Term Scheduler

- Long-term scheduler controls the degree of multiprogramming
  - May need to limit this degree to provide satisfactory service to the current set of processes
  - Must decide when the operating system can take on one or more additional processes
  - Must decide which jobs to accept and turn into processes
    - First come, first served
    - Priority
    - Execution times, I/O requirements, etc.
- Medium-term scheduler is part of the swapping function
  - Swapping-in decisions also based on the need to manage the degree of multi-programming
  - Also considers the memory requirements of the swapped-out processes

Short-Term Scheduling (Dispatcher)

- Executes most frequently, makes fine-grained decisions of which process to execute next
- Invoked for every occurrence of an event that may lead to the blocking of the current process
  - E.g, clock interrupt, I/O interrupt, OS call, signal, semaphore
- Attempts to optimize certain aspect of the system behaviour = needs a set of criteria to evaluate its policy
  - User-oriented criteria (such as response time) relates the behaviour of the system as perceived by the user
  - System-oriented criteria focus on efficient utilization of the CPU (or the rate at which processes are completed)
  - Performance-related criteria (e.g., response time): quantitative, easy to measure
  - Non-performance-related criteria (e.g., predictability): qualitative, not os easy to measure

Scheduling Criteria

- User Oriented, Performance Related
  - Turnaround time: execution + waiting time between the submission of a process and its completion; appropriate for batch jobs
  - Response time: time from the submission of a request until the response begins to be received (particularly meaningful for interactive jobs)
  - Deadlines: when deadlines exist (real time) they take precedence
- User Oriented, Other
  - Predictability: a job should run in about the same amount of time and at about the same cost regardless of the load (minimize surprise)
- System Oriented, Performance Related
  - Throughput: maximize the number of processes completed per unit of time
  - Processor utilization: the percentage of time that the processor is busy (efficiency measure, significant for expensive, shared systems)
- System Oriented, Other
  - Fairness: processes should be treated the same; no one should suffer starvation
  - Priority enforcement: favor higher-priority processes if applicable
  - Balancing resources: keep the resources of the system busy, favour processes that will underutilize stressed resources (also long- and medium-term scheduling criterion)

Characteristics of Scheduling Algorithms

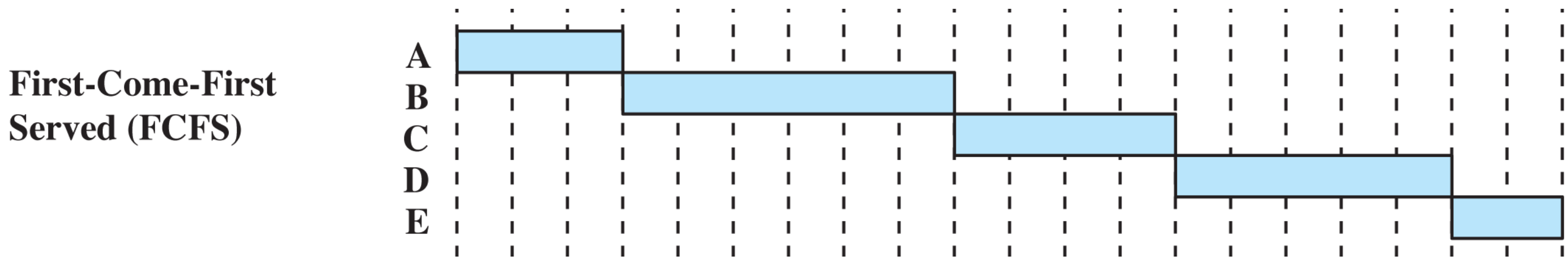
- Selection Function determines which ready process is selected next for execution
  - May be based on priority, resource requirements, or the execution characteristics
  - Significant characteristics:

    w = time spent in system so far, waiting
    e = time spent in execution so far
    s = total service time required by the process (supplied or estimated)
- Decision mode determines when is the selection function exercised
  - Non-preemptive – process continues to be in the running state until it terminates or blocks itself on I/O
  - Preemptive – processes may be moved from Running to Ready by the OS

First-Come-First-Served (FCFS)

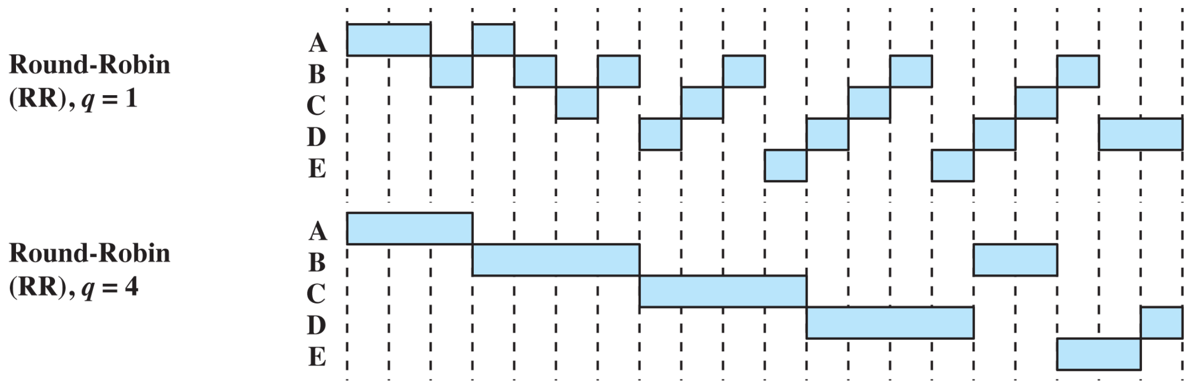
Process	Arrival time	Service time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

- Strict queuing scheme, simplest policy
- Performs better for long processes
- Favours processor-bound processes over I/O-bound ones

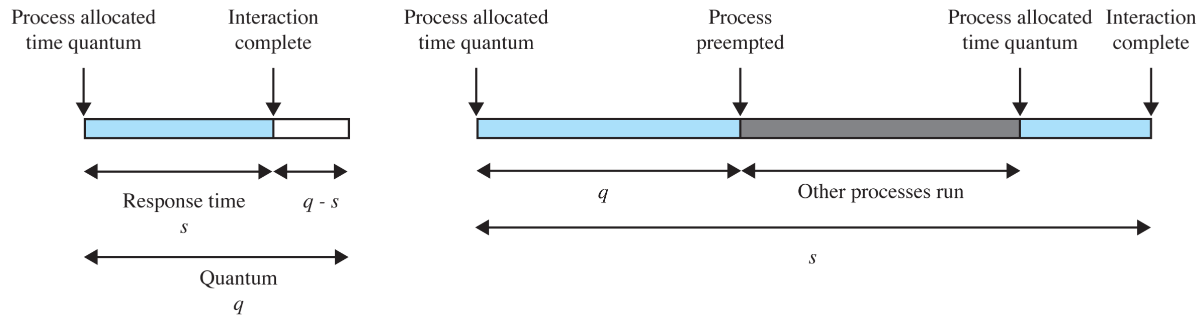


Round Robin (RR)

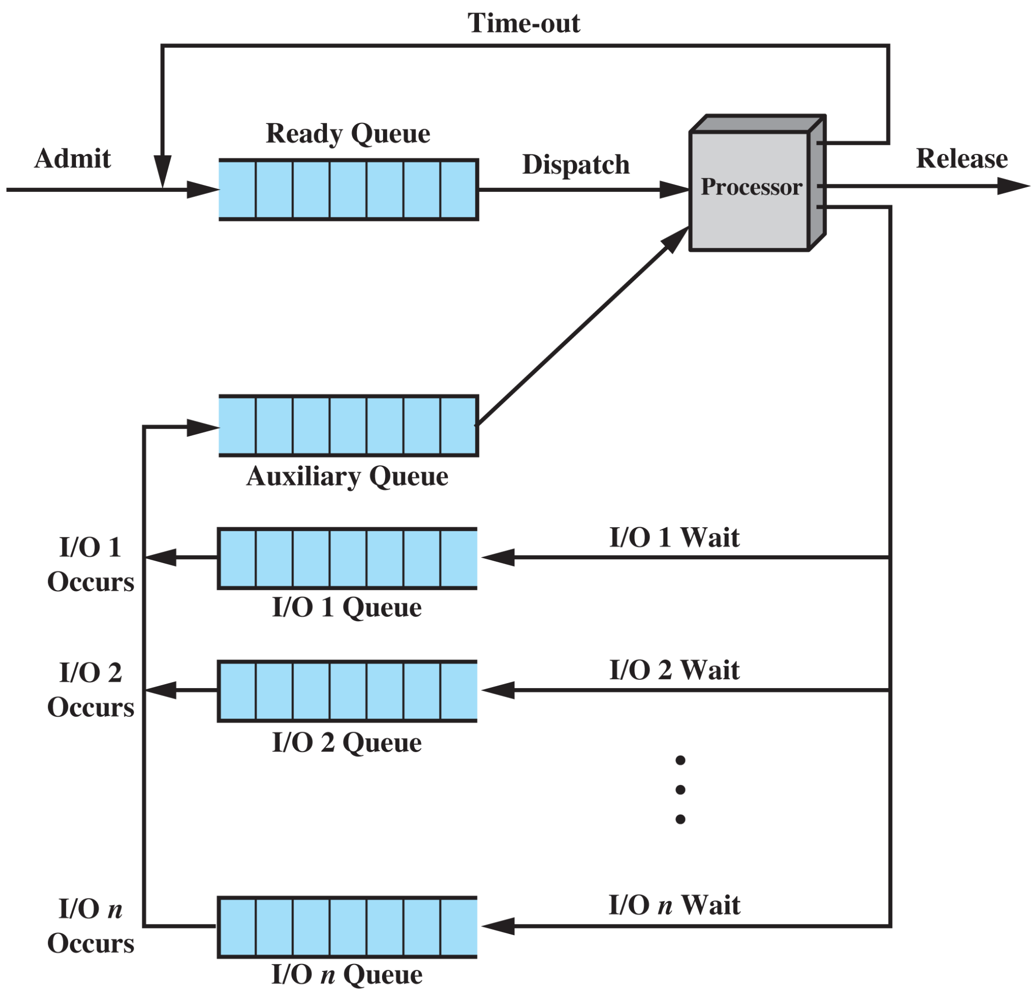
- Preemption based on a clock, also known as time slicing
- Effective in general-purpose, time-sharing systems; favours CPU-bound processes



- Main design choice: the size of the time slice (or time quantum) – affects response time as well as total service time

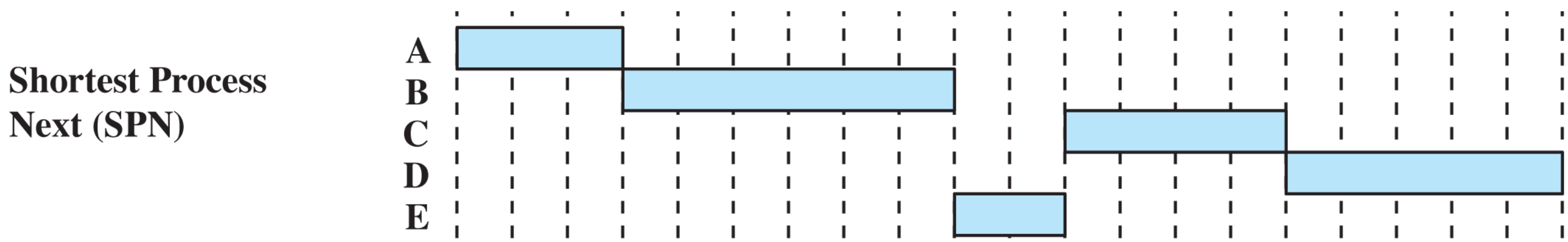


Virtual Round Robin (VRR)



Shortest Process Next (SPN)

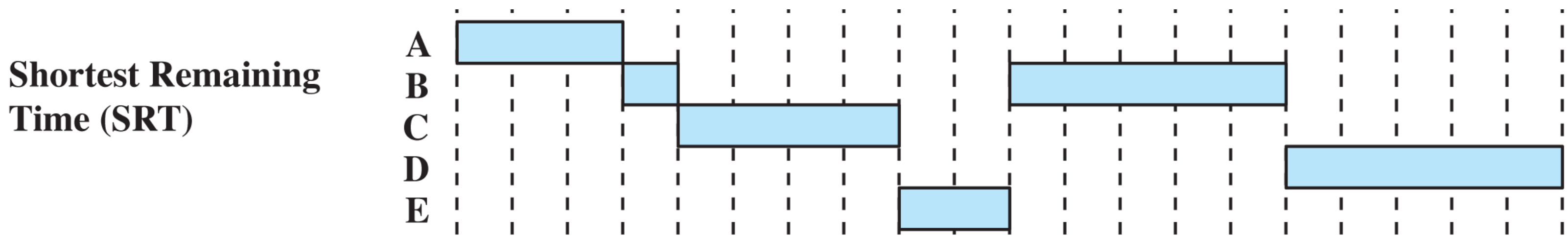
- Non-preemptive, selects the process with the shortest expecting processing time
- Short processes jump the queue, longer processes may starve



- Main difficulty: obtain an (estimate of) the running time
  - If estimate way off (shorter) the system may abort the job

Shortest Remaining Time (SRT)

- Preemptive variant of SPN
- Scheduler always chooses the process that has the shortest expected remaining processing time

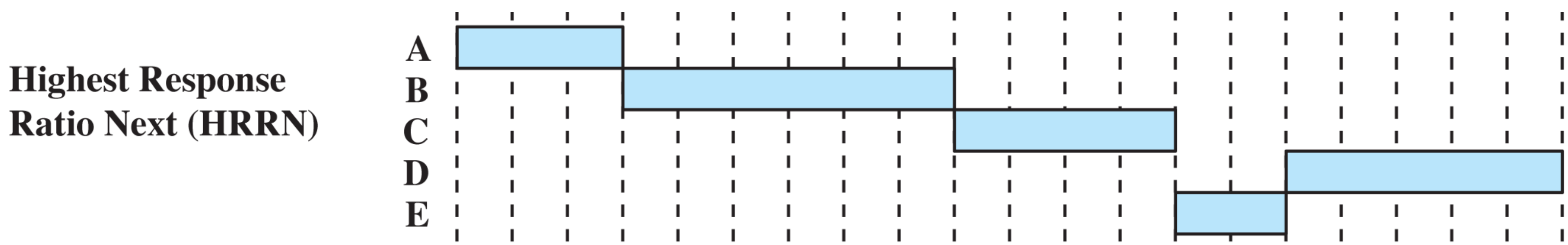


- Increased risk of starvation for longer processes
- But turnaround performance superior to SPN since a short job is given immediate preference

Highest Response Ratio Next (HRRN)

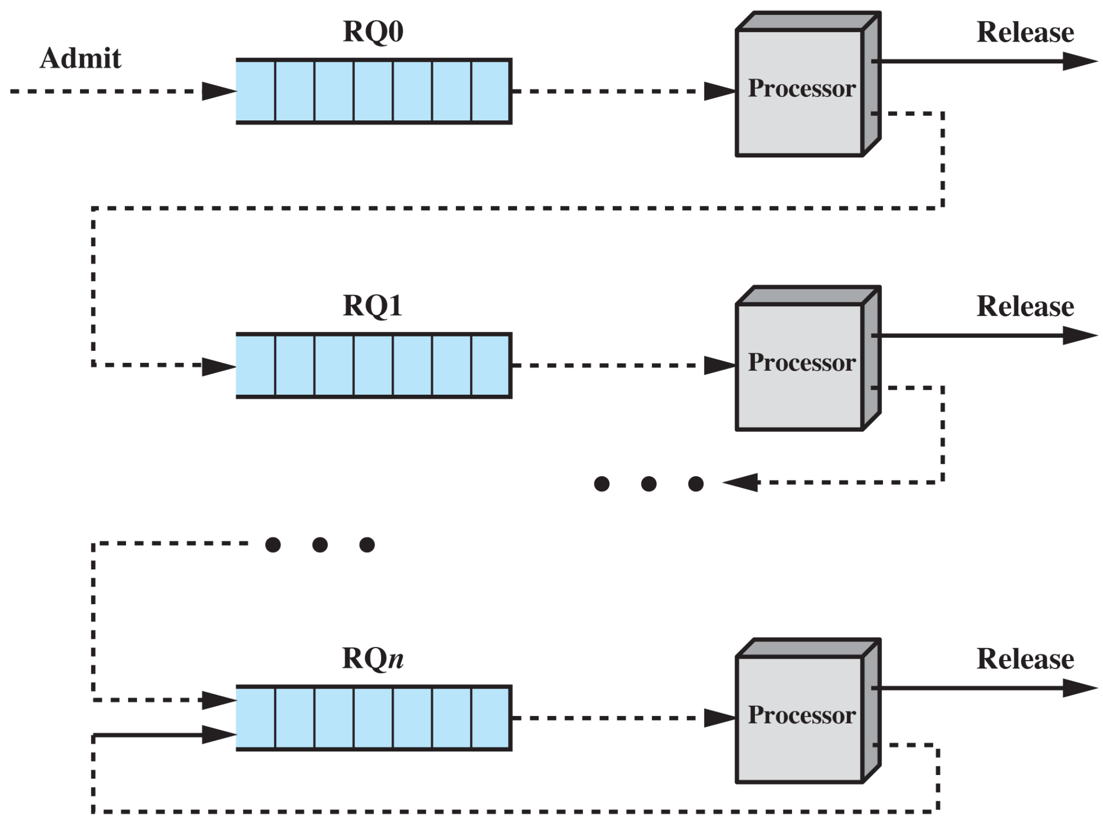
- Chooses next process with the greatest ratio

$$\text{Ratio} = \frac{\text{time spent waiting} + \text{expected service time}}{\text{expected service time}}$$



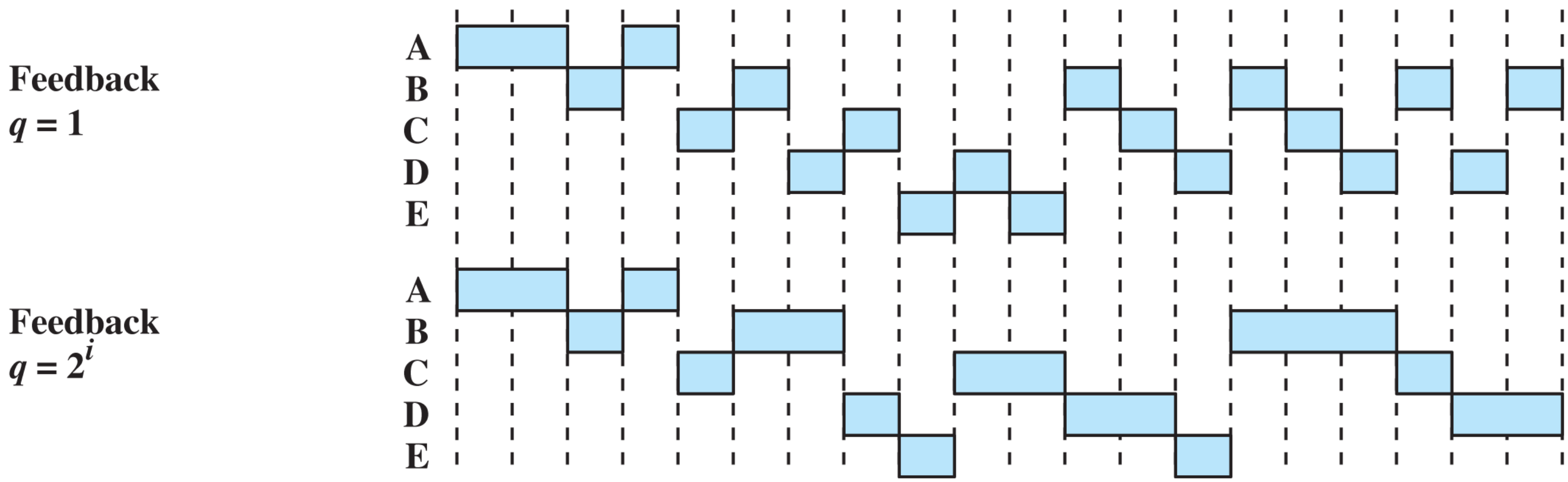
- Attractive because it accounts for the age of the process
- Shorter processes are favoured, but longer processes have a chance
  - The longer a process waits, the greater its ratio

Feedback Scheduling



FEEDBACK PERFORMANCE SCHEDULING

- Good when no estimate running time is available – will penalize jobs that have been running the longest instead
- Preemptive, dynamic priority
- Each time a process is preempted, it is also demoted to a lower-level queue
- Time quanta may be different in different queues



Comparison of Scheduling Algorithms

	FCFS	RR	SPN	SRT	HRRN	Feedback
Selection function	max[w]	constant	min[s]	min[s - e]	max$\left(\frac{w+s}{s}\right)$	
Decision mode	Non-preemptive	Preemptive (at time quantum)	Non-preemptive	Preemptive (at arrival)	Non-preemptive	Preemptive (at time quantum)
Throughput	Not emphasized	Low if quantum is too small	High	High	High	Not emphasized
Response time	May be high	Good for short processes	Good for short processes	Good	Good	Not emphasized
Overhead	Minimum	Minimum	Can be high	Can be high	Can be high	Can be high
Effect on processes	Penalizes short & I/O bound processes	Fair treatment	Penalizes long processes	Penalizes long processes	Good balance	May favor I/O bound processes
Starvation	No	No	Possible	Possible	No	Possible

Comparison of Scheduling Algorithms (cont'd)

	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/Ts	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

Traditional Unix Scheduling

- Used in both SV R3 and 4.3 BSD UNIX - time-sharing, interactive systems
- Provides good response time for interactive users while ensuring that low-priority background jobs do not starve
- Uses multilevel feedback using round robin within each of the priority queues
- Makes use of one-second preemption
- Priority is based on process type and execution history

$$
\begin{aligned}
CPU_j(i) &= \frac{CPU_i(i-1)}{2} \\
P_j(i) &= Base_j + \frac{CPU_j(i)}{2} + nice_j
\end{aligned}
$$

  - $CPU_j(i)$ = processor utilization by process j through interval i
  - $P_j(i)$ = priority of process j at the beginning of interval i (lower is higher)
  - $Base_j$ = base priority of process j
  - $nice_j$ = user-defined adjustment factor

Multiprocessor Scheduling

- Granularity of synchronization:
  - Independent – multiple, unrelated processes; typical for time-sharing systems
    - Multiprocessor systems will do the same thing, only faster
  - Coarse (200–1M instructions) – concurrent processes in a multiprogramming environment
    - No significant change for multiprocessor systems
  - Medium (20–200 instructions) – parallel processing in a single application
    - Explicit parallelism (multiple threads)
    - Frequent interaction affects scheduling considerably
  - Fine ($<$ 20 instructions) – parallelism inherent in a single instruction stream; complex interaction
    - No good, general solution
- Design issues: dispatching, use of multiprogramming on every individual processor, assignment of processes to processor

Assigning Processes to Processors

- Treat processors as a pool of resources and assign on demand
  - Assumes symmetric multiprocessing (SMP)
- Assign processes to specific processors – group or gang scheduling
  - Less overhead in the scheduling function
  - Different processors can have different utilizations
- Both these methods need some way to decide which process goes on which processor
  - Master/slave: kernel always run on a particular (master) processor
    - Master responsible for scheduling, slaves send requests to the master
    - Conflict resolution is simplified (one processor controls everything)
    - But the master can become a bottleneck
  - Peer: kernel can run on any processor
    - Each processor self-schedules from a pool of available processes
    - Complicates the OS design

Load Sharing Scheduling

- No particular assignment to any processor; load distributed evenly across processors
- No centralized scheduler, single queue system – can be organized as seen earlier (FCFS, RR, etc.)
- Disadvantages:
  - Central queue system must be accessed under mutual exclusion (bottleneck)
  - Preempted threads are unlikely to execute on the same processor, so caching is less efficient
  - All threads treated the same, so context switching is most of the time between processes (expensive)

Gang Scheduling

- Simultaneous scheduling of threads that make up a single process
  - Cheaper context switching
  - Less scheduling overhead
- Particularly useful for medium- to fine-grained parallel applications (performance degrades when part of the application is blocked while other parts run)

Dedicated Processor Assignment

- Each thread of an application is assigned to one processor and will remain so until the end of the program
- But if a thread is blocked, then that processor is idle (decreased utilization)
  - However, in a highly parallel system with tens or hundreds of processors, processor utilization is no longer so important as a metric for effectiveness or performance
  - The total avoidance of process switching during the lifetime of a program should result in a substantial speedup of that program

Dynamic Scheduling

- Provide language and system tools that permit the number of threads in the process to be altered dynamically
  - This allows the operating system to adjust the load to improve utilization
- Both the operating system and the application are involved in making scheduling decisions
- The scheduling responsibility of the operating system is primarily limited to processor allocation
- This approach is superior to gang scheduling or dedicated processor assignment for applications that can take advantage of it

POSIX THREAD SCHEDULING

- Process-contention scope (PCS) with scheduling competition within the process
- System-contention scope (SCS) with scheduling competition among all threads in system
- Pthreads API allows specifying either PCS or SCS during thread creation

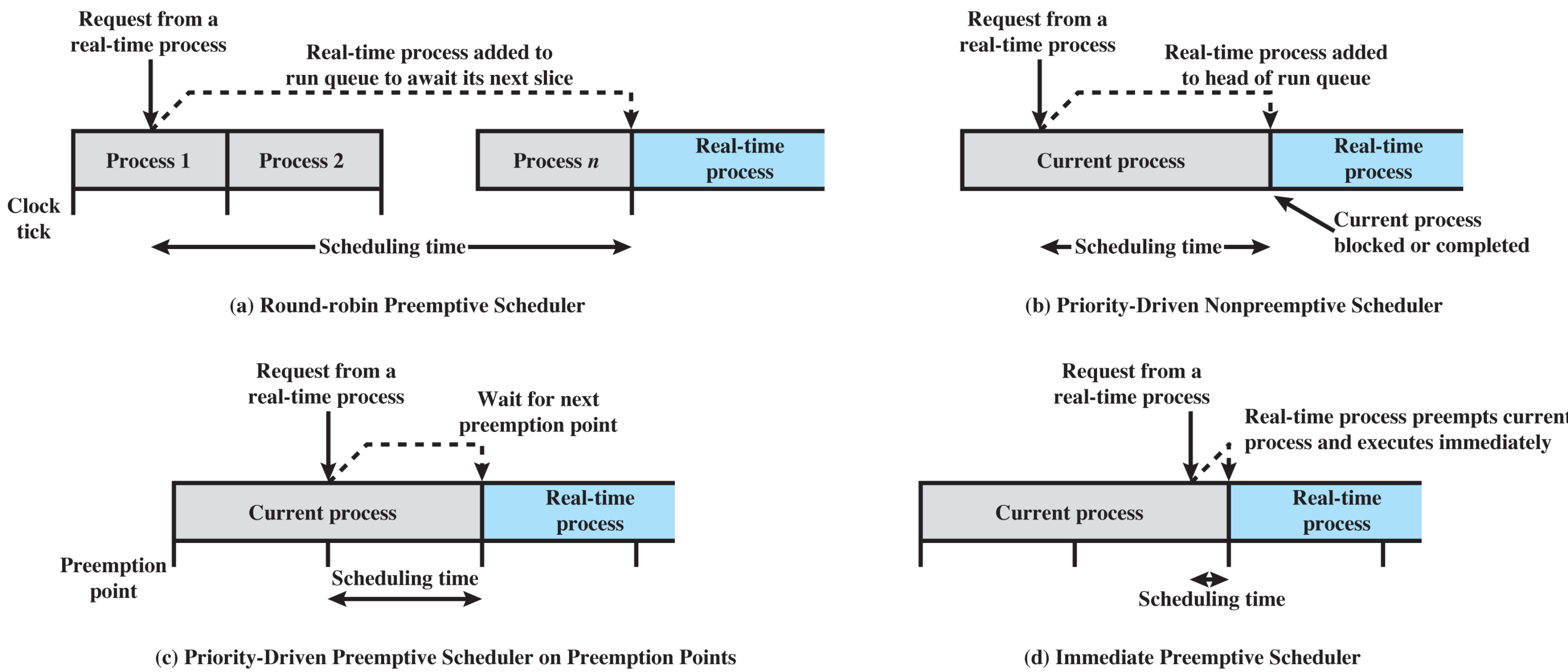
  `PTHREAD_SCOPE_PROCESS` schedules threads using PCS scheduling
  `PTHREAD_SCOPE_SYSTEM` schedules threads using SCS scheduling

```
int i;    pthread t tid[NUM_THREADS];    pthread attr t attr;
pthread attr init(&attr); /* get the default attributes */
/* set the scheduling algorithm to PROCESS or SYSTEM */
pthread attr setscope(&attr, PTHREAD_SCOPE_SYSTEM);
/* set the scheduling policy - FIFO, RT, or OTHER */
pthread attr setschedpolicy(&attr, SCHED_OTHER);
for (i = 0; i < NUM THREADS; i++) /* create the threads */
    pthread create(&tid[i],&attr,runner,NULL);
for (i = 0; i < NUM THREADS; i++) /* join on each thread */
    pthread join(tid[i], NULL);
```

REAL-TIME SYSTEMS

- Real time system: The correctness of the system depends not only on the logical result of the computations but also on the time at which those results are produced
  - Most often time constraints are stated as deadlines
  - Tasks or processes attempt to control or react to events that take place in the outside world
  - These events occur in real time and tasks must be able to keep up with them
  - The scheduler is the most important component of these systems
- Hard real time: Timing violations will cause unacceptable damage or a fatal error to the system
- Soft real time: Deadlines are desirable but not mandatory, so that it makes sense to schedule and execute a job even if its deadline has passed
- Further characteristics: determinism, responsiveness, reliability, fail-soft operation
- Real-time tasks can be
  - Periodic, with requirements stated as "once per period T" or "every T time units"
  - Aperiodic, which may have constraints on both start and end times

Real-Time Scheduling



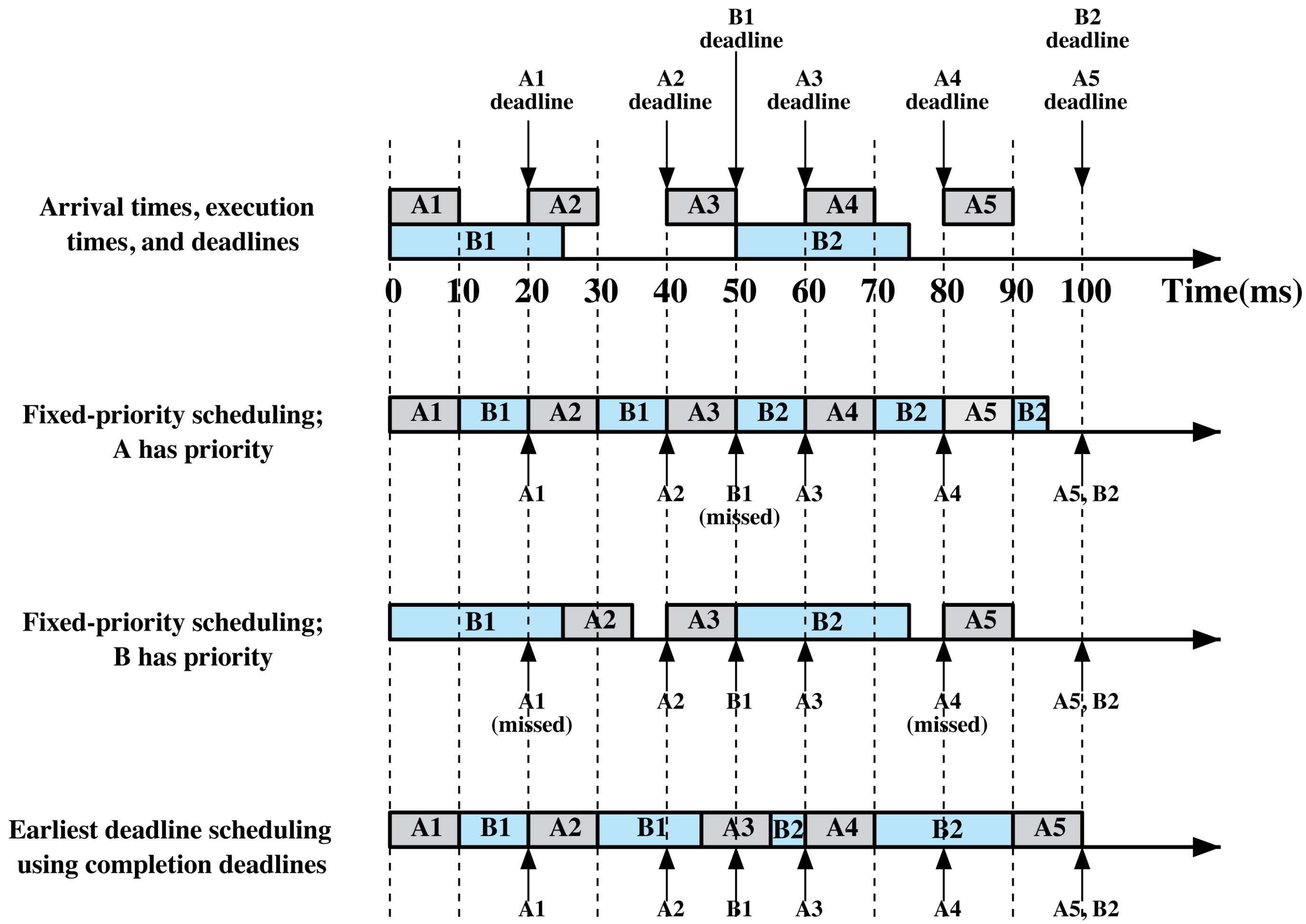
Classes of Real-Time Scheduling

- Static table-driven approaches
  - Performs a static analysis of feasible schedules of dispatching
  - Result is a schedule that determines at run time when a task must start
- Static priority-driven preemptive approaches
  - A static analysis is performed but no schedule is drawn up
  - Analysis is used to assign priorities to tasks so that a traditional priority-driven preemptive scheduler can be used
- Dynamic planning-based approaches
  - Feasibility is determined at run time rather than offline
  - One result of the analysis is a schedule or plan that is used to decide when to dispatch the task at hand
- Dynamic best effort approaches
  - No feasibility analysis is performed
  - System tries to meet deadlines, aborts any started process with missed deadline

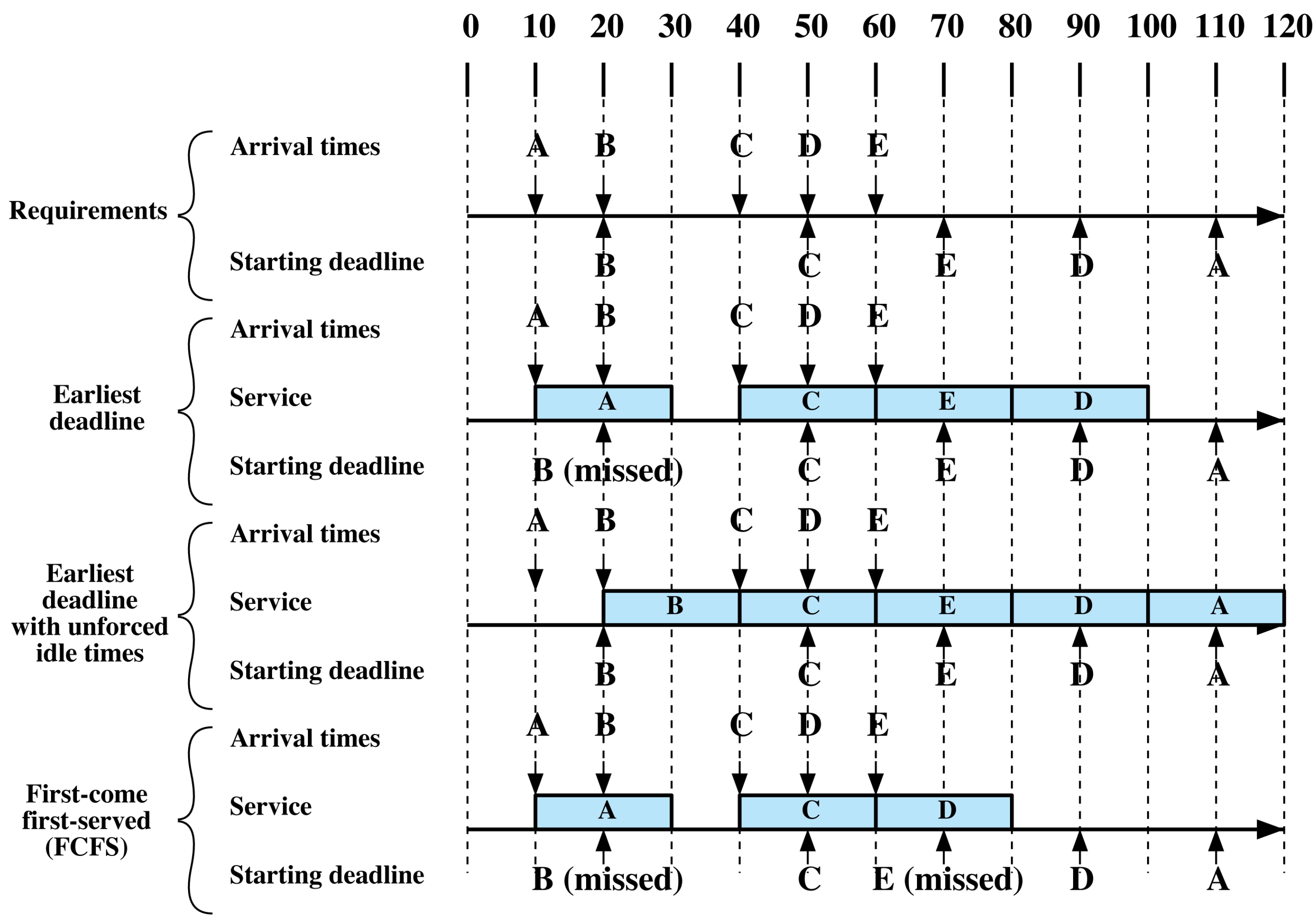
DEADLINE SCHEDULING

- Real-time operating systems will start real-time tasks as rapidly as possible and emphasize rapid interrupt handling and task dispatching
- Real-time applications are generally not concerned with sheer speed but rather with completing (or starting) tasks at the most valuable times
- Priorities provide a crude tool and do not capture the requirement of completion (or initiation) at the most valuable time
- Information used for deadline scheduling:
  - Ready time
  - Starting deadline
  - Completion deadline
  - Processing time
  - Resource requirements
  - Priority
  - Subtask scheduler (task may be split into a mandatory and an optional subtask)

Periodic Real-Time Scheduling with Completion Deadlines



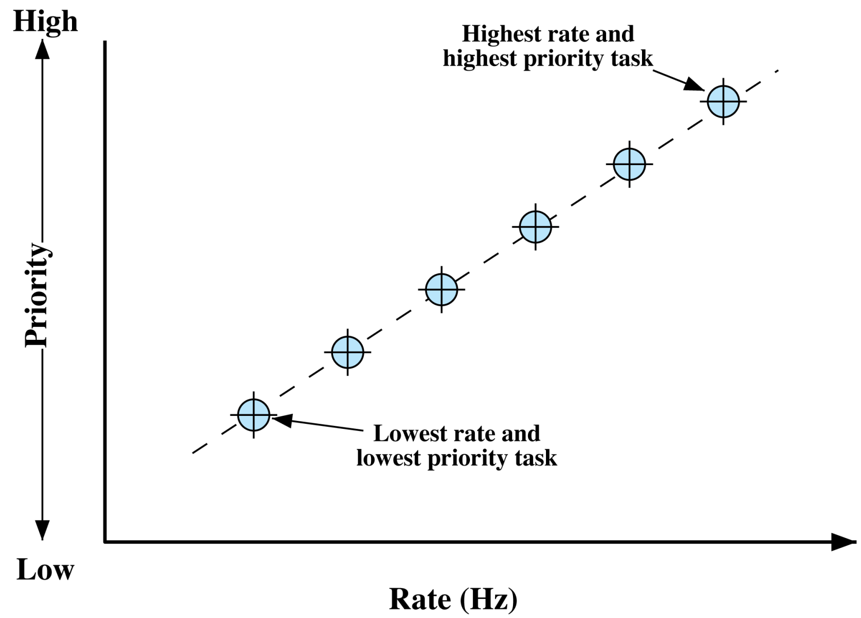
Aeriodic Real-Time Scheduling with Starting Deadlines

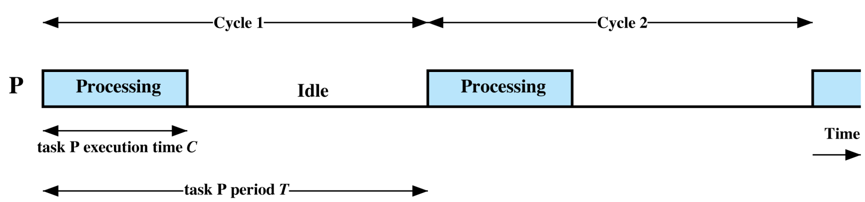


- Static-priority scheduling, priorities assigned on the basis of the cycle duration of the job: the shorter the cycle, the higher is the job's priority
- Rate monotonic analysis used to provide scheduling guarantees for a particular application: A feasible schedule always exists as long as the CPU utilization is below a specific bound

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$$

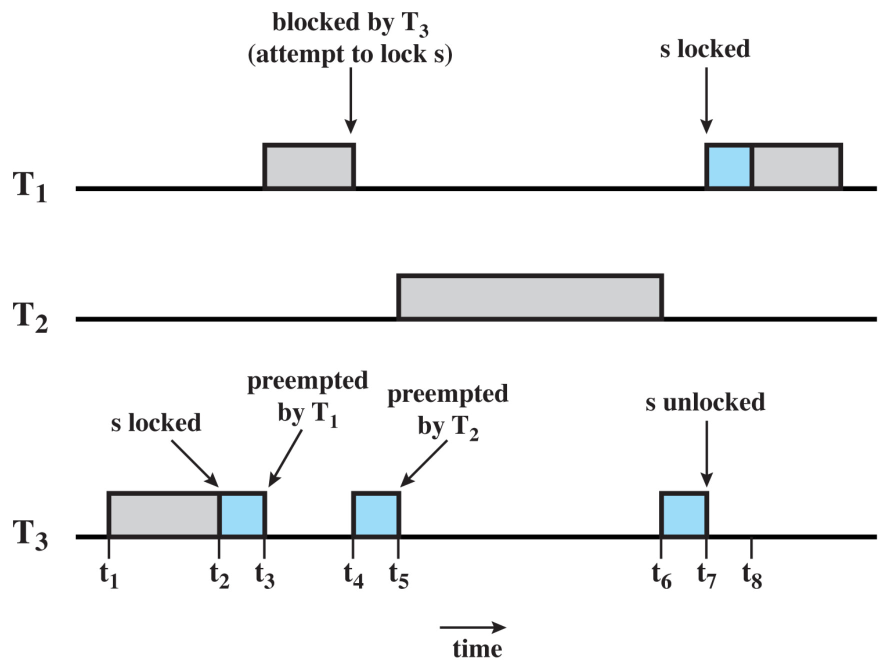
- $\lim_{n\to\infty}(n(2^{1/n} - 1)) = \ln 2$ so all deadlines can be met as long as the CPU load is less than 69.3%
- The rest 30.7% load usable for non-real-time tasks



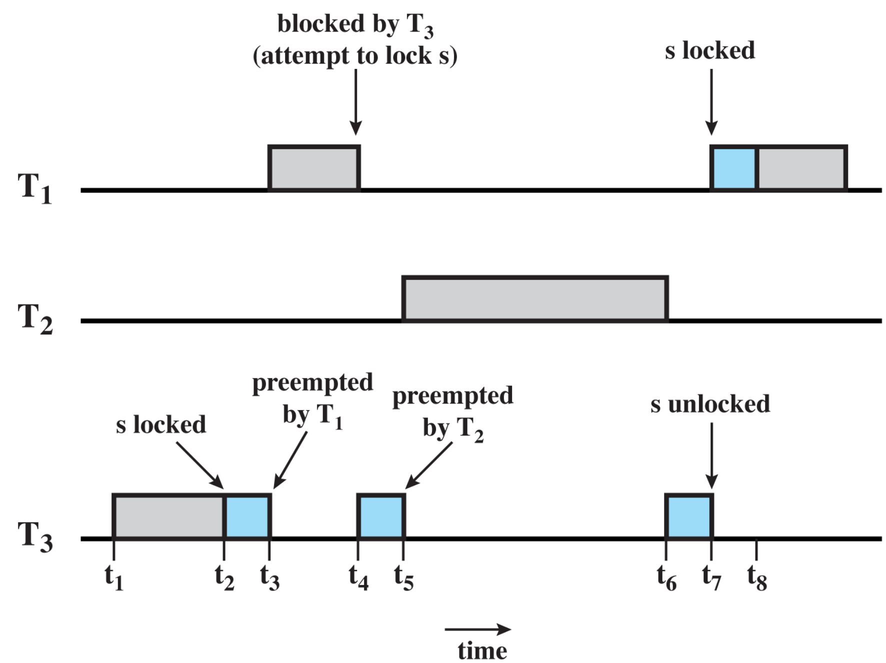


- Can occur in any priority-based preemptive scheduling scheme
- Particularly relevant in the context of real-time scheduling
- Occurs when circumstances within the system force a higher priority task to wait for a lower priority task

  – Unbounded Priority Inversion: the duration of a priority inversion depends not only on the time required to handle a shared resource, but also on the unpredictable actions of other unrelated tasks



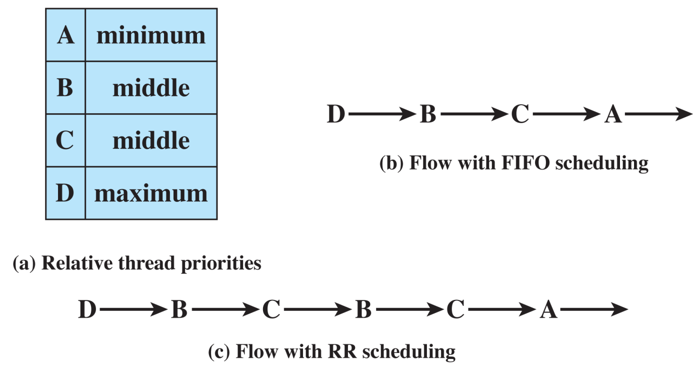
- Fixes the priority inversion problem
- Increase the priority of a process to the maximum priority of any process waiting for any resource on which the process has a resource lock
  - When a job blocks one or more high priority jobs, it ignores its original priority assignment and executes its critical section at the highest priority level of all the jobs it blocks
  - After executing its critical section, the job returns to its original priority level



- Three classes of processes

  `SCHED_FIFO`: FIFO, real-time threads
  `SCHED_RR`: Round-robin, real-time threads
  `SCHED_OTHER`: Non-real-time threads

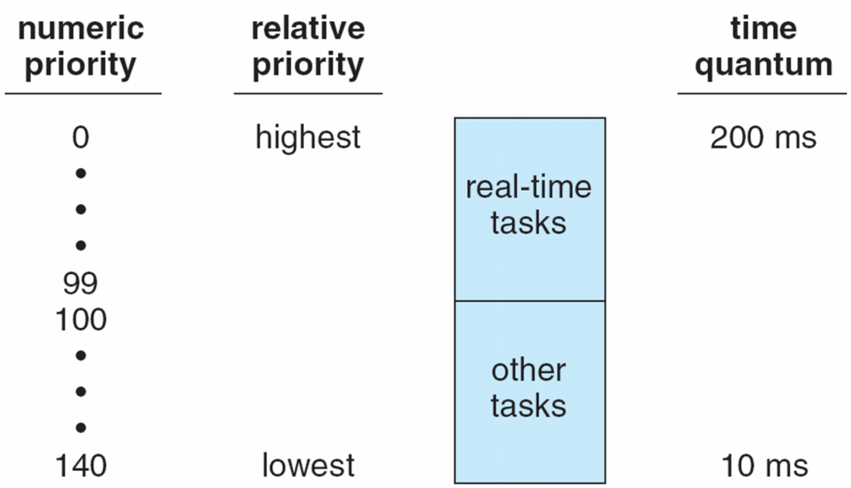
  – Multiple priorities within each class



(a) Relative thread priorities
(b) Flow with FIFO scheduling
(c) Flow with RR scheduling

- `SCHED_OTHER` oses an $O(1)$ scheduler
  – Two priority ranges: time-sharing and real-time
  – Real-time range from 0 to 99 and nice value from 100 to 139
  – Different time quanta assigned for each class
  – Kernel maintains two scheduling data structures for each processor in the system

LINUX SCHEDULING (CONT'D)



- Active queues: 140 queues by priority each containing ready tasks for that priority
- Expires queues: 140 queues containing ready tasks with expired time quanta