Operating Systems: Scheduling

Review on processes

- Process image
    - Collection of programs, data, stack and attributes that form the process
    - User data
        - Modifiable part of the user space
        - Program data, user stack area, and modifiable code
    - User program
        - Executable code

Kernel execution

- Nonprocess kernel
    - Kernel executes outside of any user process
    - Common on older systems
    - Kernel takes over upon interrupt or system call
    - Has its own memory, own stack
    - Concept of process only applies to user programs and not OS

Kernel execution

- Execution with user processes
    - OS executes in context of user process
    - Kernel is collection of functions called by user process to provide a service
    - Each user process must have memory for prog, data, stack areas of kernel routines
    - Separate kernel stack used to manage code execution in kernel mode

Kernel execution

- Execution with user processes
    - OS code and data are in shared address space and shared by all processes
    - Interrupt, trap, or system call execute in user address space
        - but in kernel mode
    - Termination of kernel's job allows process to run with just a mode switch

Kernel execution

- Process-based kernel
  - Kernel is a collection of system processes (microkernels)
  - Allows modular OS design
    - Different system processes can call each other

Unix kernel

- Controls execution of processes by allowing their creation, termination, suspension, communication
- Schedules processes *fairly* for execution on CPU
  - CPU executes a process
  - Kernel suspends process when time quantum elapses
  - Kernel schedules another process to execute
  - Kernel later reschedules the suspended process

Unix kernel

- Allocates main memory for an executing process
- Allocates secondary memory for efficient storage and retrieval of user data
- Allows controlled peripheral device access to processes

Unix kernel

- Allocates main memory for an executing process
- Allocates secondary memory for efficient storage and retrieval of user data
- Allows controlled peripheral device access to processes

Highest level of user process

- The *shell* in Unix
  - Created for each user (login request)
  - Initiates, monitors, and controls the progress for user
  - Maintains global accounting and resource data structures for user
  - Keeps static information about user
    - Identification, time requirements, I/O requirements, priority, types of processes, resource needs
  - May create child processes (progenies)

Sync and critical regions

- Needed to avoid race conditions amongst parts of kernel code
- Protect the modification and observation of resource counts
- Synchronization achieved by:
    - Disable kernel preemption
        - A process executing in kernel mode may be made nonpreemptive
        - If a process in kernel mode voluntarily gives up CPU, it must make sure that all data structures are left in consistent state
        - On resumption, must recheck values of any previously accessed data structures
        - Works for uniprocessors but not with multiple CPUs

Sync and critical regions

- Disable interrupts
  - If critical section is large, interrupts remain disabled for long periods
    - All hardware activity freezes
  - Does not work on multiprocessor systems

Sync and critical regions

- Spin locks
  - Busy/wait in some systems, including linux
  - Convenient in kernel code because kernel resources are locked just for a fraction of a second
    - More time consuming to release CPU and reacquire it later
    - If time required to update data is short, semaphore is inefficient
      - process checks sem, suspends itself (expensive), meanwhile other process just released it

Sync and critical regions

- Spin locks cont.
  - Data structures need to be protected from being concurrently accessed by kernel control paths that run on different CPUs
  - When process finds the lock closed by another process, spin until lock is open
  - Problem is it is useless in uniprocessor environment

Sync and critical regions

- Semaphores
  - Implement a locking primitive that allows waiting process to sleep until desired resource is available
  - Two types in linux: kernel and IPC semaphores
- Kernel semaphores
  - Similar to spin locks
  - When a kernel control path tries to acquire a busy resource, it is suspended
    - Becomes runnable again when resource is released
  - Kernel semaphores can be acquired only by functions that are allowed to sleep (no interrupt handlers or deferrable functions).

kernel semaphore

```
struct semaphore
{
   atomic_t count;   // Number of available resource
        // if < 0, no resource available and at
        // least one process waiting for resource

   queue_t wait;     // Sleeping processes that are
        // waiting for resource

   int sleepers;     // Processes sleeping on semaphore?
};
```

Sync and critical regions

- Avoiding deadlocks
  - Big issue when number of kernel locks used is high
  - Difficult to ensure that no deadlock state will ever be reached for all possible ways to interleave kernel control paths
  - Linux avoids this by requesting locks in a predefined order
    - For example:

```
Suppose we have three resources: cat, dog, rat
Always acquire cat before dog, good idea to document this
/*
*   cat_lock - always obtain before dog lock
*/
```

Data structures for Processes and Resources

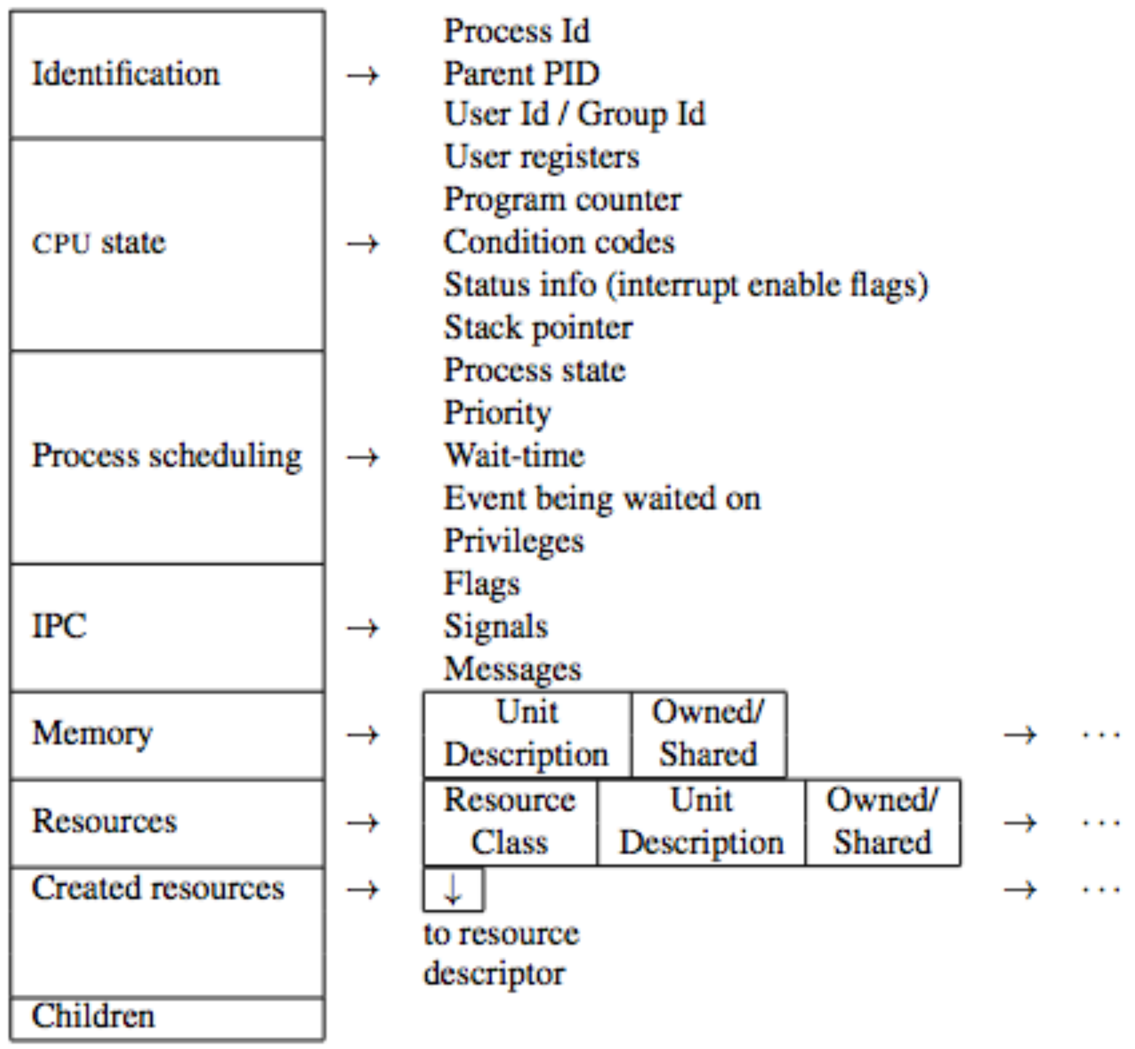
- Used by the OS to keep track of each process and resource
- Cross-referenced or linked in main memory for proper coordination
- Memory tables
  - Used to keep track of allocated and requested main/sec memory
  - Protection attributes of blocks of main and secondary memory
  - Information to map main memory to secondary memory
- I/O tables
  - Used to manage I/O devices and channels
  - State of I/O operation and locations in main memory as source/ destination of operations

Sync and critical regions

- File tables
    - Information on file existence, location in secondary memory, current status and other attributes
- Process Control Block
    - Most important data structure in an OS
    - Set of all process control blocks describes *state* of the OS
    - Read and modified by almost every subsystem of OS
        - Scheduler, resource allocator, performance monitor
    - Constructed at process creation time
        - Physical manifestation of that process
        - Set of data locations for local, global and constant variables

Process Control Block

- Contains specific information associated with a specific process
    - Information can be broadly classified as process identification, processor state information and process control information



Process Control Block

- Identification
  - Always unique integer in Unix, providing index into primary process table
  - All subsystems in the OS use it to determine information about process
- CPU state
  - Provides snapshot of the process
  - The program counter register is initialized to entry of program (`main()`)
  - While process is running, registers contain a certain value that is to be saved if processor is interrupted
  - Typically described by the registers that form the processor status word (PSW)
  - EFLAGS register on Pentium

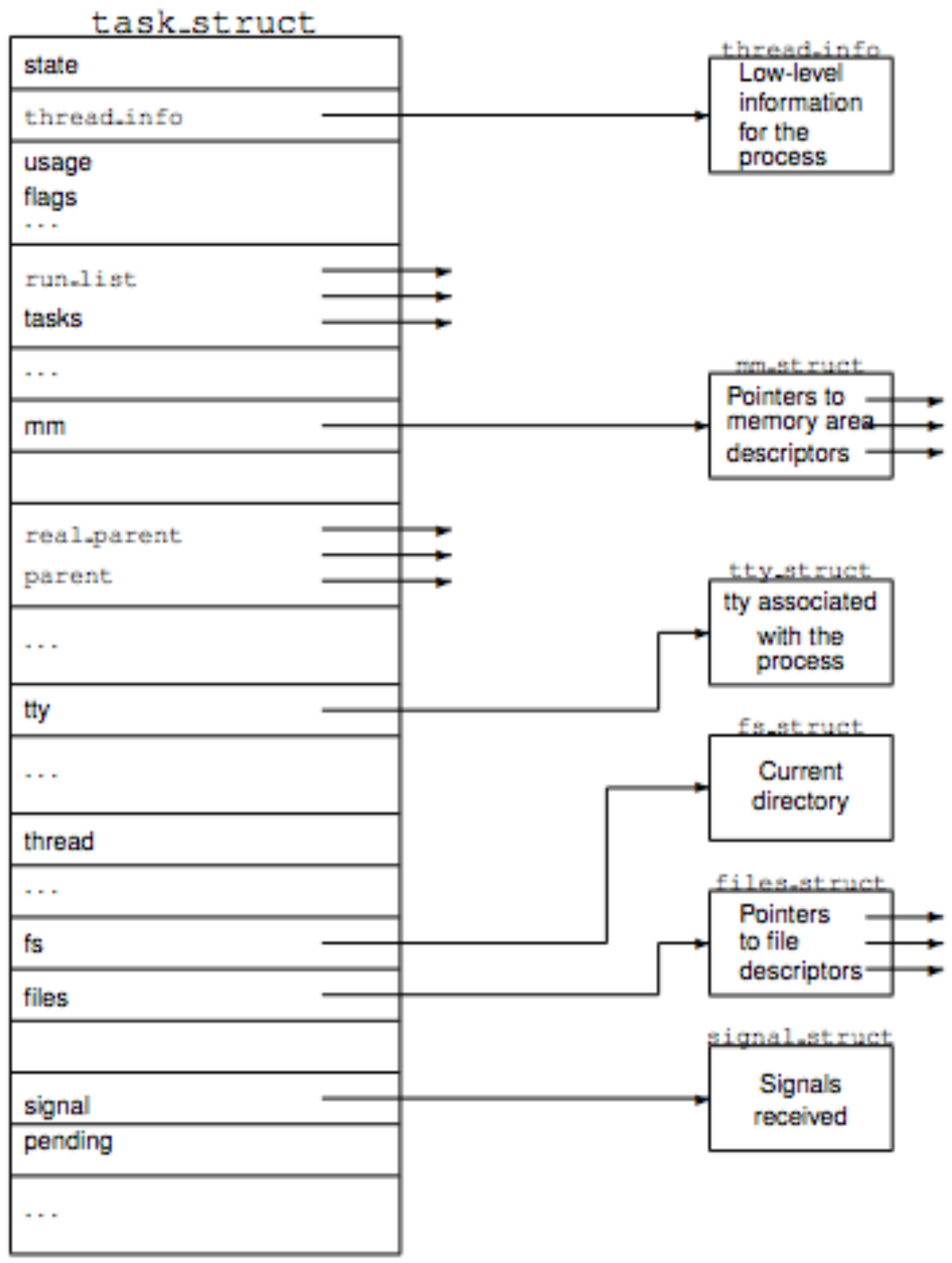
Process Control Block

- Process state
  - Current activity of the process
  - Running: Executing instructions
  - Ready: Waiting to be assigned to a processor, also initial state
  - Blocked: Waiting for some event to occur
- Allocated address space
  - Memory map

Process Control Block

- Resource list
  - Resources associated with the process
  - Open files, I/O devices, etc
- Other information
  - Progenies/offspring of the process
  - Priority of the process
  - Accounting information: CPU and real time used, limits, etc
  - I/O status information: Outstanding I/O requests, I/O devices allocated, list of open files

Linux task structure



Process Control Block

- Linux identifies process by 32-bit address of task_struct, called *process descriptor pointer*
    - Points to doubly linked list of `task_struct`
- Unix identifies process by PID stored in `pid` field of process descriptor
    - PIDs are numbered sequentially, assigned in increasing order, over long term are reused
    - PIDs are not reused immediately to prevent race conditions
        - Process could send signal to another process
        - Before signal is received, recipient terminated and PID reassigned
        - Signal gets sent to wrong process

Process Control Block

- UNIX pid
  - Maximum PID number is 32,767 (`PID_MAX_DEFAULT` - 1)
  - Default can be increased
    - By writing to file `/proc/sys/kernel/pid_max`
  - In 64-bit architectures, PID can be changed to 4,194,303
  - PIDs managed by bitmap `pidmap_array`
    - Determines which PIDs are currently assigned
    - Page frame contains 4kb, 32-bit arch allows storage in one page
  - 64bit archs, requires additional pages added to bitmap for PIDs too large for the current bitmap

Process Control Block

- POSIX standard requires threads in same group to have a common PID to send a signal to affect all threads in the group
  - PID of thread group leader (first lwp) is shared by all threads
  - Stored in `tgid` field of process descriptor
- Linux stores process information in process descriptor pointers linked by a doubly linked list
  - Head of the list is process 0 or `swapper`
    - Identified by `task_struct init_task`

Resource descriptors

- Resource
  - Reusable, relatively stable, and often scarce commodity
  - Successively requested, used, and released by processes
  - Hardware (physical) or software (logical) components
- Resource class
  - *Inventory*: Number and identification of available units
  - *Waiting list*: Blocked processes with unsatisfied requests
  - *Allocator*: Selection criterion for honoring the requests

Resource descriptors

- Contains specific information associated with a certain resource
  - p->status_data points to waiting list associated with resource
  - Dynamic and static resource descriptors, with static descriptors being more cmmon
  - Identification, type and origin
    - Resource class identified by a pointer to its descriptor
    - Descriptor pointer maintained by creator in pcb
    - External resource name goes with unique resource id

Resource descriptors

- Information about resources cont.
  - Inventory List
    - Associated with each resource class
    - Shows the availability of resources of that class
    - Description of the units in the class
  - Waiting Process List
    - List of processes blocked on the resource class
    - Details and size of the resource request
    - Details of resources allocated during the process

Resource descriptors

- Information about resources continued
  - Allocator
    - Matches available resources to the requests from blocked processes
    - Uses the inventory list and the waiting process list
  - Additional information is also required
    - Measurement of resource demand and allocation
      Total current allocation and availability of elements in resource class

Process Context

- Context of a process
  - Information to be saved that may be altered when a interrupt is serviced by the interrupt handler
  - PC and stack pointer registers
  - General purpose registers
  - Floating point registers
  - processor control registers (PSW) containing information on CPU state
  - Memory management registers used to keep track of RAM accessed by process

Process Context

- Context of a process cont.
  - Process switch/Task switch/Context switch
    - Suspend currently running process and resume execution of some process previously suspended
    - Hardware context:
      - All processes share CPU registers
      - Before resuming process, kernel must make sure that each register is loaded with value it had when the process was suspended

Process Context

- Context of a process cont.
  - Hardware context
    - Subset of process execution context
    - Part of hardware context is stored in process descriptor while remaining part is stored in kernel mode stack
  - Context switching vs interrupt handling
    - Code executed by interrupt handler does not constitute a process
    - Just a kernel control path that runs at expense of that process that was running when interrupt occurred

Process Group

- Process groups represent a job abstraction
- Example:
  - processes in a pipeline form a group
  - Shell acts on those as single entity
- Process descriptor contains a field called *process group* ID
  - PID of the *group leader*

Process Context

- Login session
  - All processes that are descendants of the process that started a working session on a specific terminal
  - All processes in a process group are in the same login session
  - A login session may have several process groups
  - One of the processes is always in the foreground
    - It has access to the terminal
  - When background process tries to access terminal, it receives a SIGTTIN or SIGTTOUT signal

Basic Operations on Processes

- Basic operations on processes are implemented by kernel primitives
- Maintain the state of the operating system
- These primitives are indivisible, protected by “busy-wait” type of locks

Basic Operations on Processes

- create - Establish a new process
  - Assign a new unique PID to the process
  - Allocate memory to the process for all elements of process image, including private user address space and stack; the values can possibly from from the parent process; set up any linkages, and then allocate space for PCB
  - Create a new process control block corresponding to PID and add it to process table, initialize different values in there such as parent PID, list of children (initialized to null), program counter (set to program entry point), system stack pointer (set to define the process stack boundaries)

Basic Operations on Processes

- create - Establish a new process
  - Initial CPU state, typically initialized as Ready or Ready, suspend
  - Add the process id of new process to list of children of the creating process
  - Assign initial priority
    - Initial priority of the process may be greater than the parents
  - Fill out accounting information and limits
  - Add the process to the ready list

Basic Operations on Processes

- suspend - Change process state to suspended
    - A process may suspend only its descendants
    - May include cascaded suspension
    - Stop the process if process is in running state and save the state of the processor in PCB
    - If process is already in blocked state, leave it blocked, else change to ready
    - If need be, call scheduler to schedule the processor to some other process

Basic Operations on Processes

- activate - Change process state to active
  - Change one of the descendant processes to ready state
  - Add the process to the ready list
- destroy - Remove one or more processes
  - Cascaded destruction possible
  - Only descendant processes may be destroyed
  - If the process to be killed is running, stop execution
  - Free all resources currently allocated to process
  - Remove PCB associated with killed process

Basic Operations on Processes

- change_priority - Set a new priority for the process
  - Change priority in the PCB
  - Move process to a different queue to reflect priority change

Resource primitives

- create_resource_class - Create descriptor for a new resource class
  - Dynamically establish descriptor for a new resource class
  - Initialize and define inventory and waiting lists
  - Criterion for allocation of the resources
  - Specification for insertion and removal of resources

Resource primitives

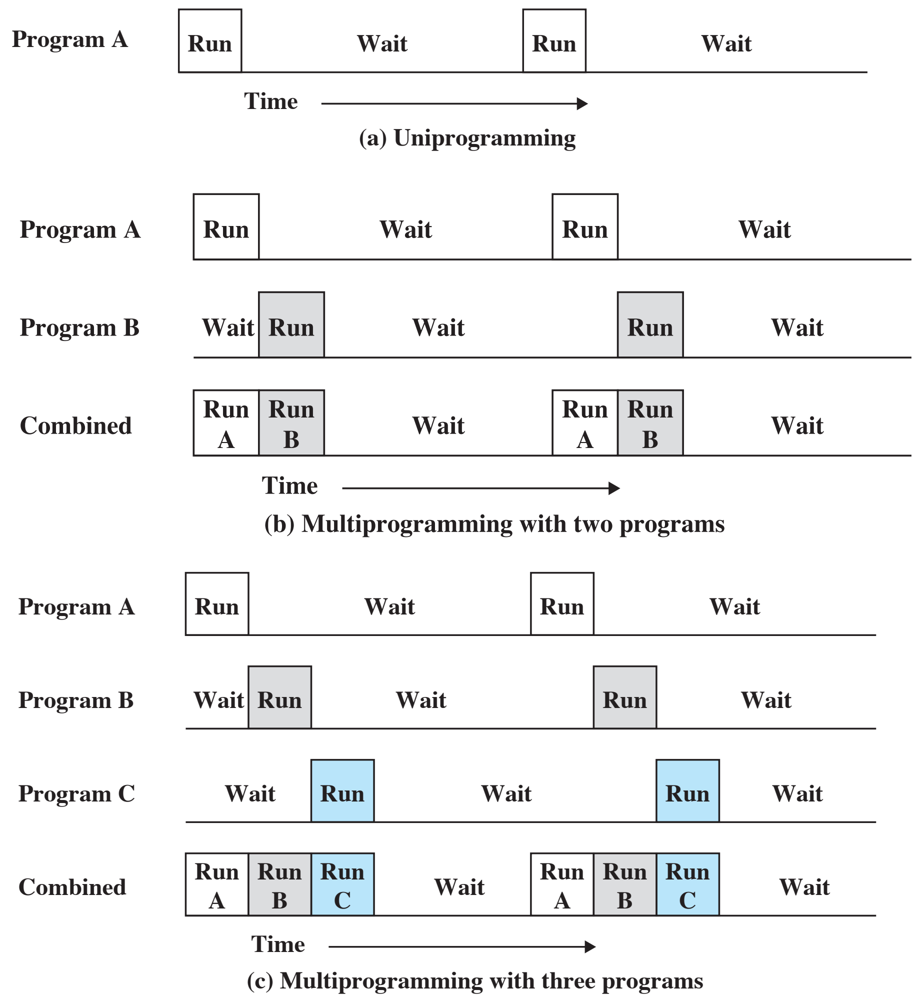
- destroy_resource_class - Destroy descriptor for a resource class
    - Dynamically remove descriptor for an existing resource class
    - Resource class can only be destroyed by its creator or an ancestor of creator
    - If any processes are waiting for the resource, their state is changed to ready
- release - release some units of a resource class
    - Return unwanted and serially reusable resources to the resource inventory
    - Inform the allocator about the return

Resource primitives

- request - Request some units of a resource class
    - Includes the details of request - number of resources, absolute minimum required, urgency of request
    - Request details and calling PIDs are added to the waiting queue
    - Allocation details are returned to the calling process
    - If request cannot be immediately satisfied, process is blocked
    - Allocator gives resources to waiting processes and modifies allocation details for the process and its inventory
    - Allocator also modifies the resource ownership in the PCB of the process

Organization of Schedulers

- What is the objective of multitasking?
  - Maximize CPU utilization and increase throughput



Organization of Schedulers

- Each entering process goes into job queue. Processes in job queue:
    - Reside in mass storage
    - Await allocation of main memory
    - If residing in main memory and awaiting CPU time
        - Kept in *ready queue*
    - If waiting for allocation of a certain I/O device
        - Reside in *device queue*

Organization of Schedulers

- So what is a scheduler?
  - Concerned with deciding a policy about which process to be dispatched
  - After selection, loads process state or dispatches
  - Process selection based on a scheduling algorithm

Organization of Schedulers

- Autonomous vs shared scheduling
  - Shared scheduling:
    - Scheduler is invoked by a function call as a side effect of kernel operation
    - Kernel and scheduler are potentially contained in the address space of all processes and execute as part of the process

Organization of Schedulers

- Autonomous scheduling
  - Scheduler (and possibly kernel) are centralized
  - Scheduler is considered a separate process running autonomously
  - Continuously polls system for work or can be driven by wakeup signals
  - Which is preferred?

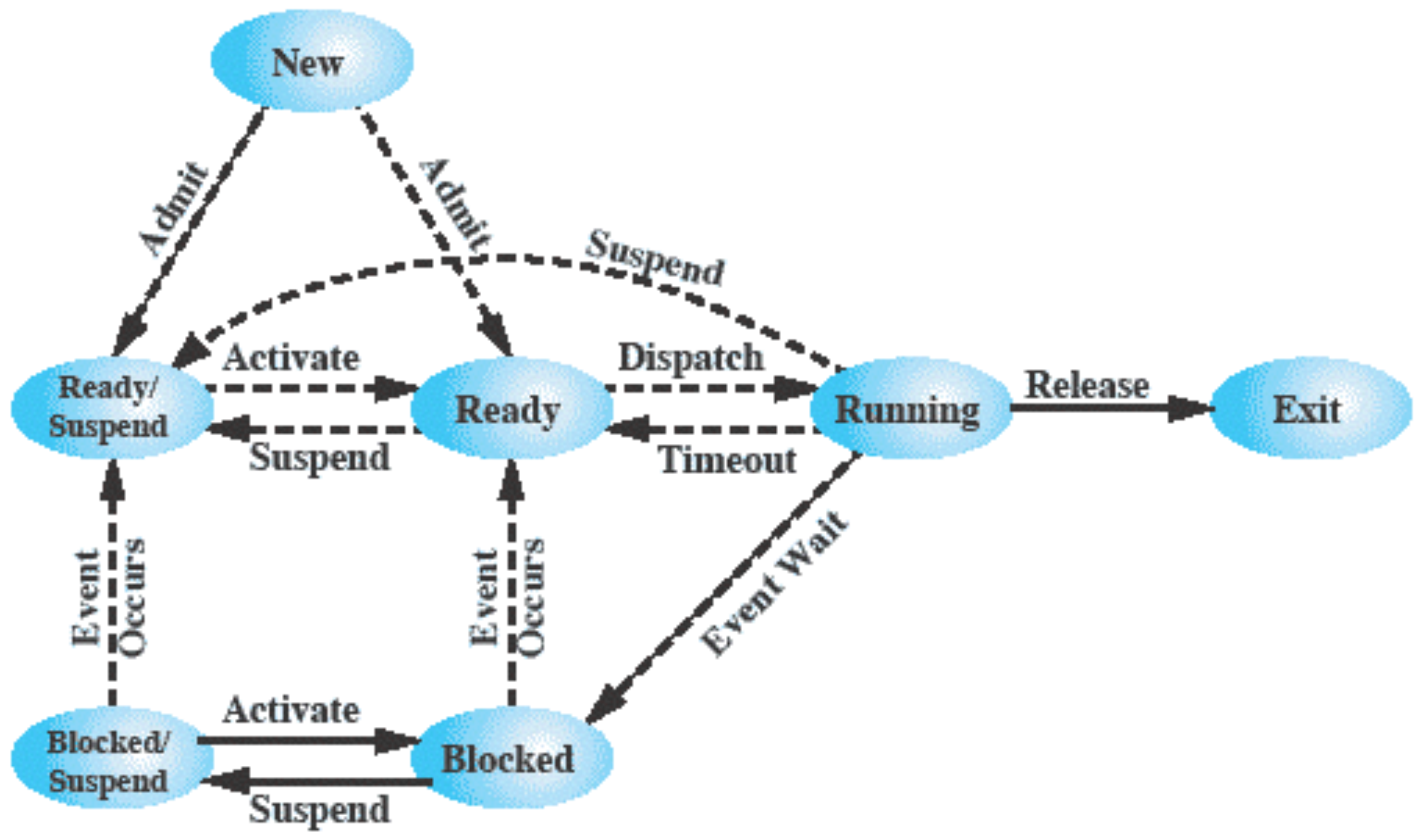
Organization of Schedulers

- Autonomous scheduling:
  - Preferable in multiprocessor systems as master/slave configuration
    - One cpu can be permanently dedicated to scheduling, kernel and other supervisory activities such as I/O and program loading
    - OS is clearly separated from user processes
  - How does scheduler get dispatched?
    - Solved by transferring control to the scheduler whenever a process is blocked or awakened

Organization of Schedulers

- Unix scheduler
  - Autonomous scheduler
  - Runs between any two other processes
  - Serves as a dummy process that runs when no other process is ready

Process states



(b) With Two Suspend States

Schedulers

- Long-term scheduler
    - Selects processes from job queue
    - Loads the selected processes into memory for execution
    - Updates teh ready queue
    - Controls the degree of multiprogramming (number of procs in memory)
    - Not executed as frequently as short-term scheduler
    - Should generate a good mix of CPU-bound and I/O bound processes
    - May not be present in some systems (like time-sharing systems)

Schedulers

- Short-term scheduler
    - Selects processes from ready queue
    - Allocates CPU to the selected process
    - Dispatches the process
    - Executed frequently (very few milliseconds, like 10 msec)
    - Must make a decision quickly, so be very fast

Schedulers

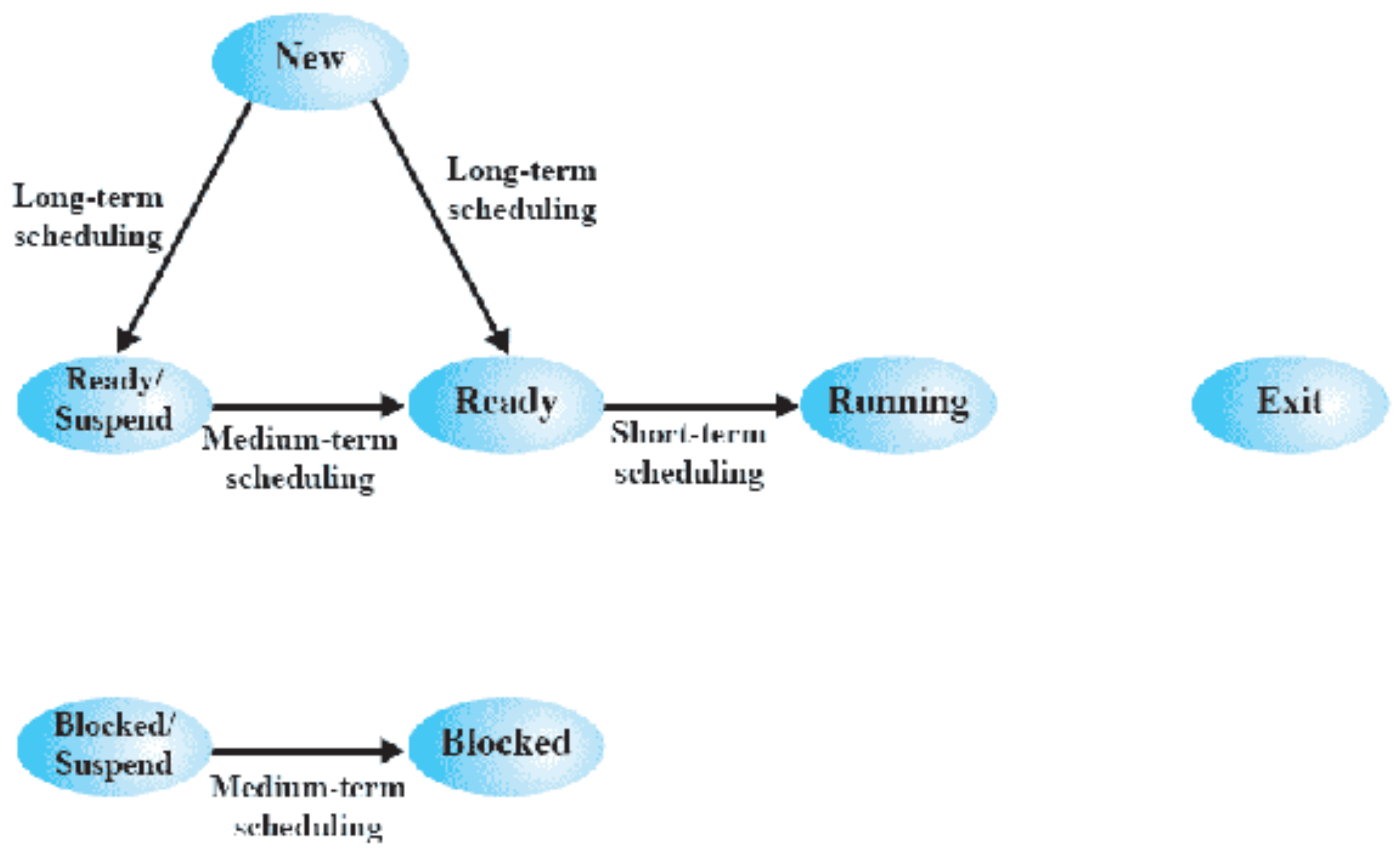
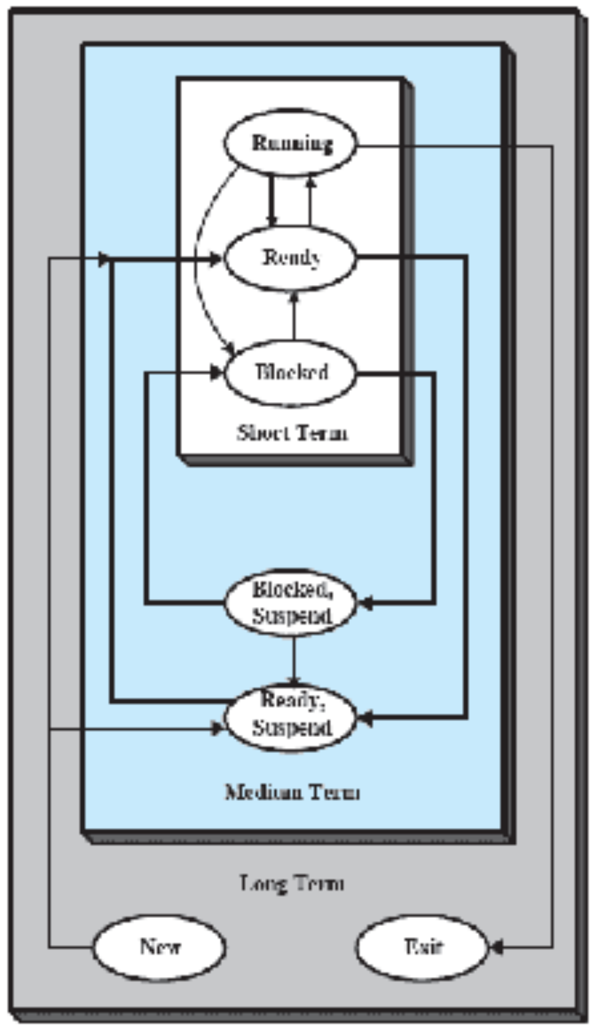


Figure 9.1 Scheduling and Process State Transitions

Schedulers


Running
Ready
Blocked
Short Term
Blocked, Suspend
Ready, Suspend
Medium Term
Long Term
New
Exit

Schedulers

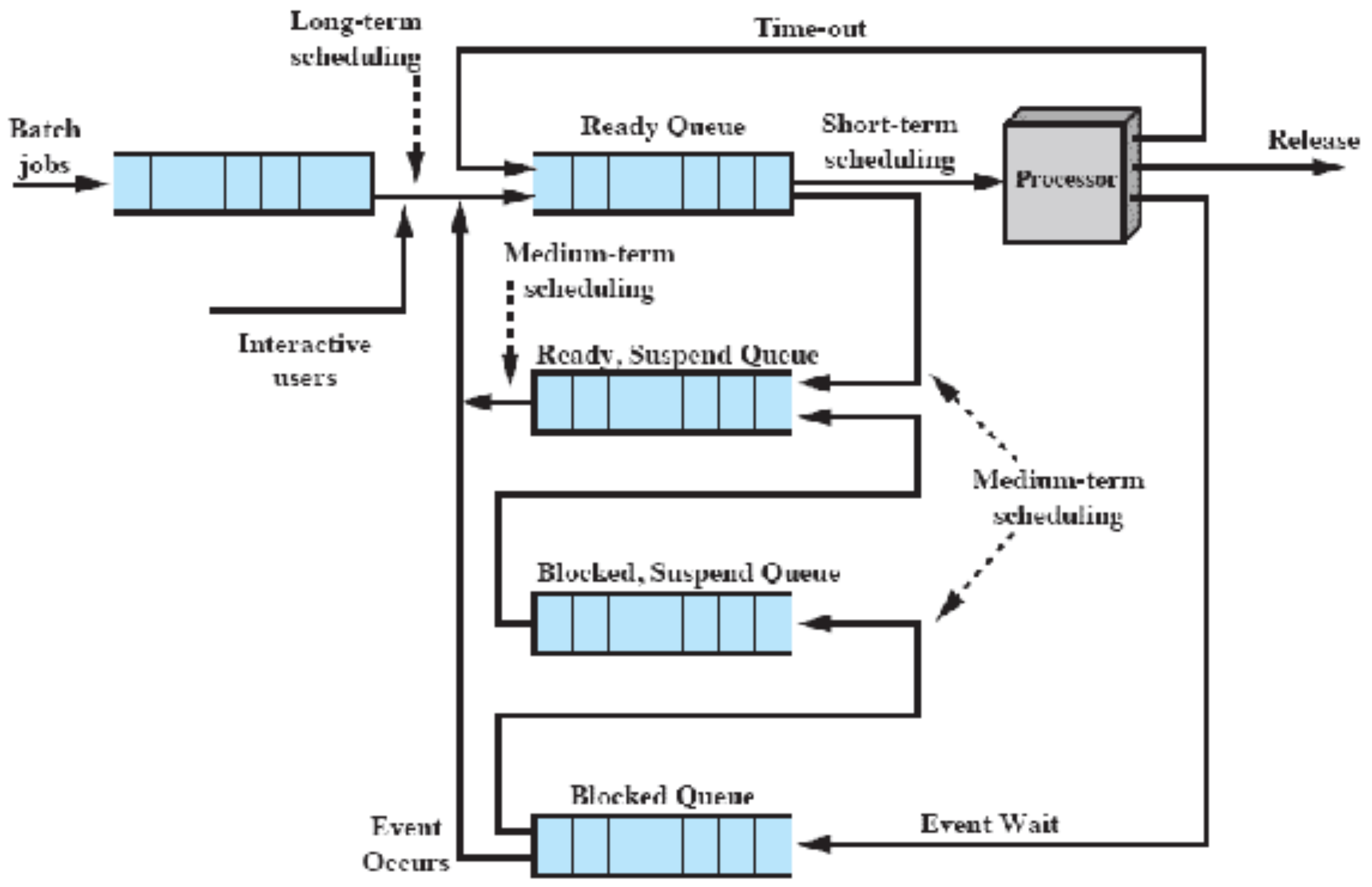


Figure 9.3 Queuing Diagram for Scheduling

Schedulers

- Major task of any operating system
  - Allocate ready processes to available processors
  - Scheduler decides process to run first based on scheduling algorithm
  - Two components of a scheduler
    - Process scheduling
      - Decision making policies to determine order in which active processes compute for use of CPu
    - Process dispatch
      - Actual binding of selected process to the CPU

Process Scheduling

- Desirable features of a scheduling algorithm
  - Fairness: Make sure each process gets its fair share of CPU and that no process starves
  - Efficiency: Keep the CPU busy 100% of the time
  - Response time: Minimize response time for interactive users
  - Turnaround: Minimize the time batch users must wait for output
  - Throughput: Maximize the number of jobs processed per hour

Process Scheduling

- Desirable features of a scheduling algorithm
  - Predictability: A job should run in same time and have same cost
  - Balance resources: Keep resources of the system busy
  - Enforcing priorities: Higher priority processes should be favored

Process Scheduling

- Types of scheduling
  - Preemptive
    - Temporarily suspend the logically runnable processes
    - More expensive in terms of CPU time
    - Can be caused by:
      - Interrupt
      - Trap
      - Supervisor call
    - Nonpreemptive
      - Run the process to completion

Process Scheduling

- Universal Scheduler: specified in terms of the following concepts
  - *Decision mode*
    - Select the process to be assigned to the CPU
  - *Priority function*
    - Applied to all processes in the ready queue to determine current priority
  - *Arbitration rule*
    - Applied to select a process in case two processes are found with the same current priority

Process Scheduling

- Decision mode
  - When (time) to select a process for execution
  - Preemptive and nonpreemptive decisions
  - Selection of a process occurs when:
    - A new process arrives
    - An existing process terminates
    - A waiting process changes state to ready
    - A running process changes state to waiting (I/O) or ready (INT)
    - Every *q* seconds (quantum)
    - Priority of a ready process exceeds priority of running process

Process Scheduling

- Decision mode:
  - Selective preemption
    - Uses a bit pair (u_p,v_p)
      - u_p set if p may preempt another process
      - v_p set if p may be preempted by another process

Process Scheduling

- Priority function
  - Defines priority of a ready process using parameters associated with process
  - Memory requirements are important due to swapping overhead
    - Smaller memory means less swapping overhead
    - Smaller memory also means can service more processes
  - Attained service time
    - Total time when the process is in the running state
  - Real time in system
    - Total actual time the process spends in the system since arrival

Process Scheduling

- Priority function
    - Total service time
        - Total CPU time consumed by process during lifetime
        - Equals attained service time when process terminates
        - Higher priority for shorter processes
        - Preferential treatment of shorter processes reduces average time a process spends in system

Process Scheduling

- Priority function
    - External priorities
        - Differentiate between classes of user and system processes
            - Interactive processes should have higher priority
            - Batch processes should have lower priority
            - Account for resource utilization

Process Scheduling

- Priority function
  - Timeliness
    - Dependent upon the urgency of a task and deadlines
  - System load
    - Maintain good response time during heavy load
    - Reduce swapping overhead by larger quanta of time

Process Scheduling

- Arbitration rule
    - Random choice
    - Round robin (cyclic ordering)
    - Chronological ordering (FIFO)
- Time-Based Scheduling Algorithms
    - May be independent of required service time

Priority Queuing

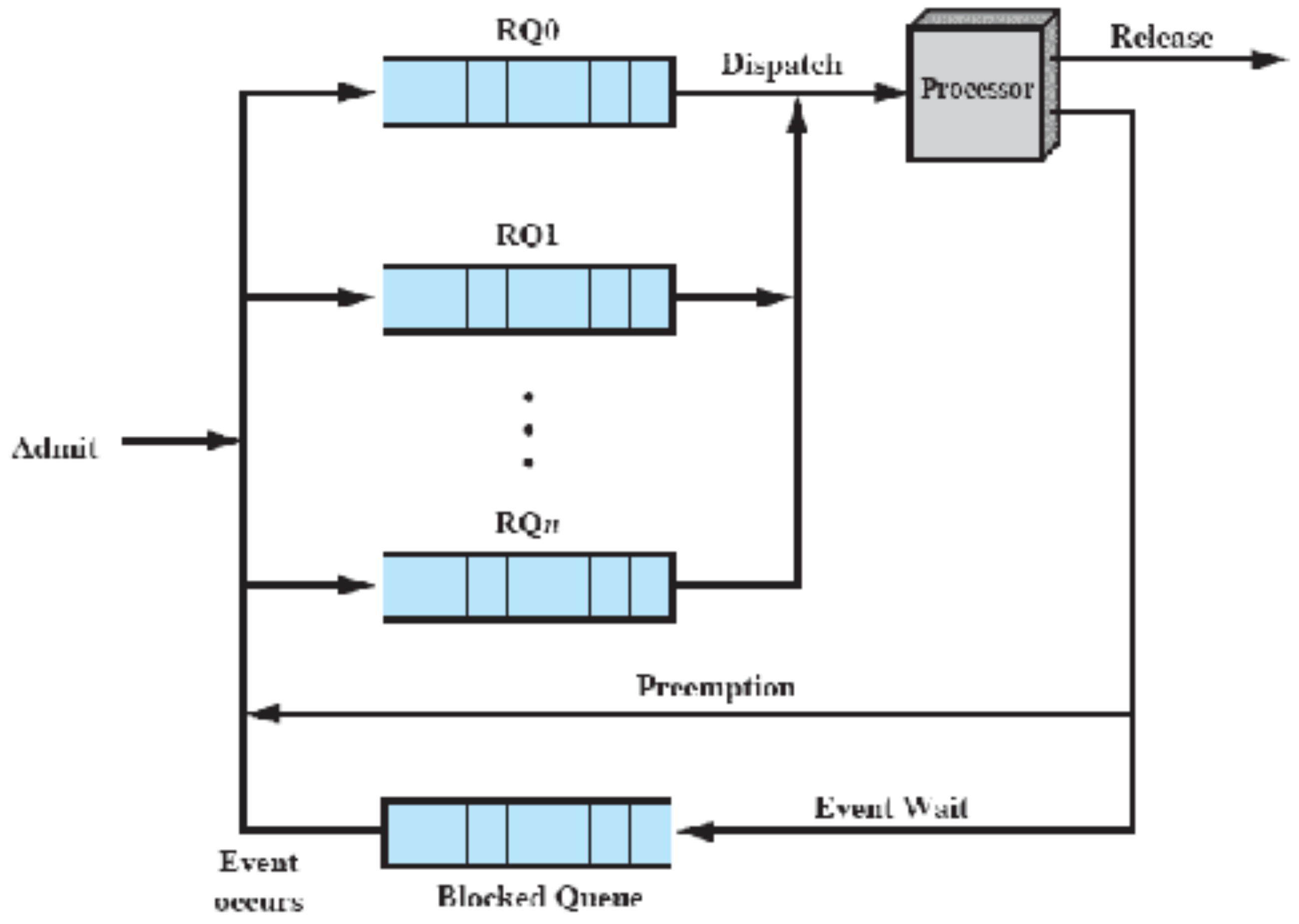
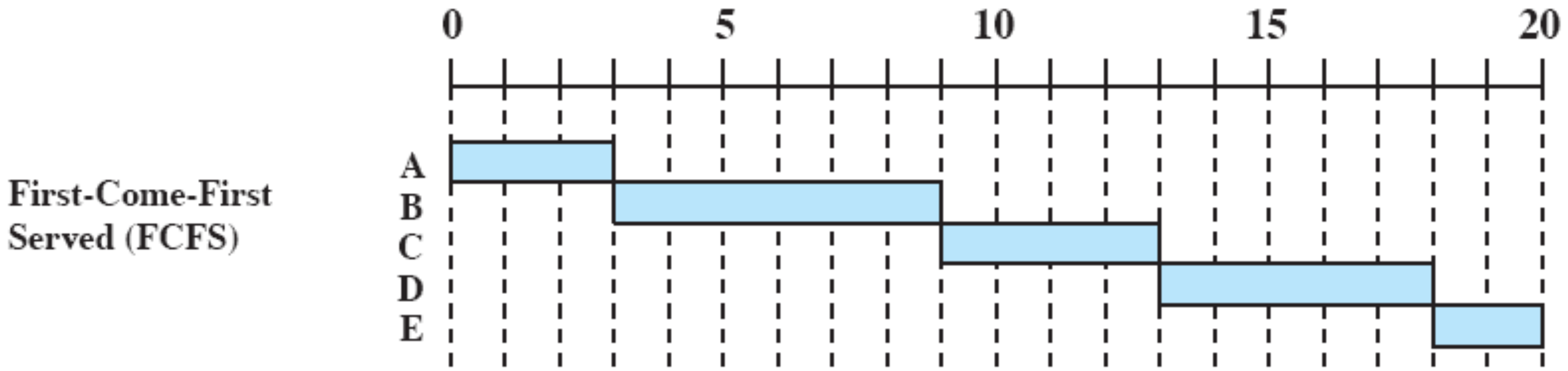


Figure 9.4 Priority Queuing

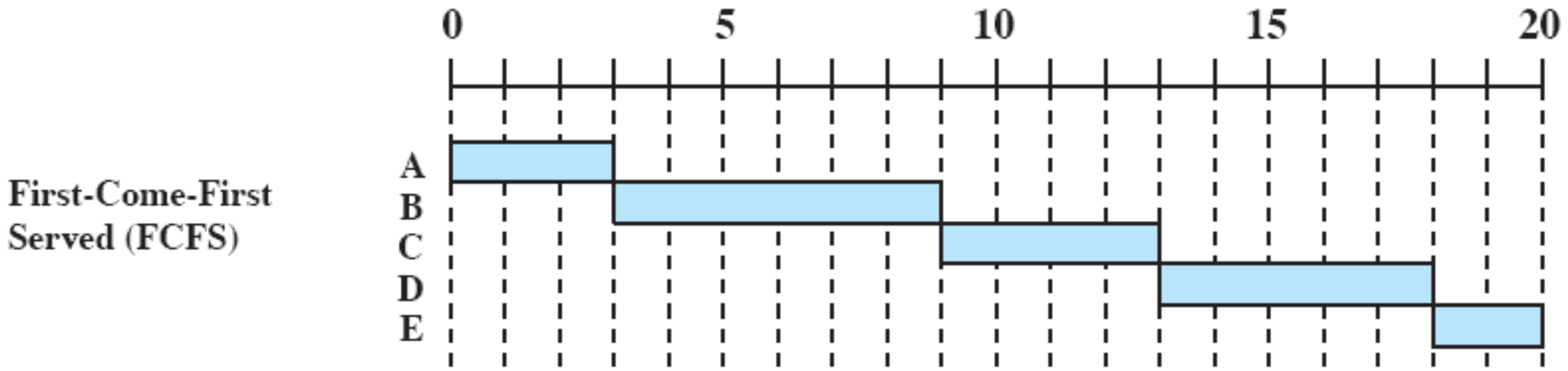
First-Come-First-Served

- Also called FIFO or strict queueing
- Nonpreemptive decision mode
- Upon process creation, link its PCB to rear of FIFO queue
- Scheduler allocates CPU to the process at the front of FIFO queue

FIFO scheduling



FIFO scheduling



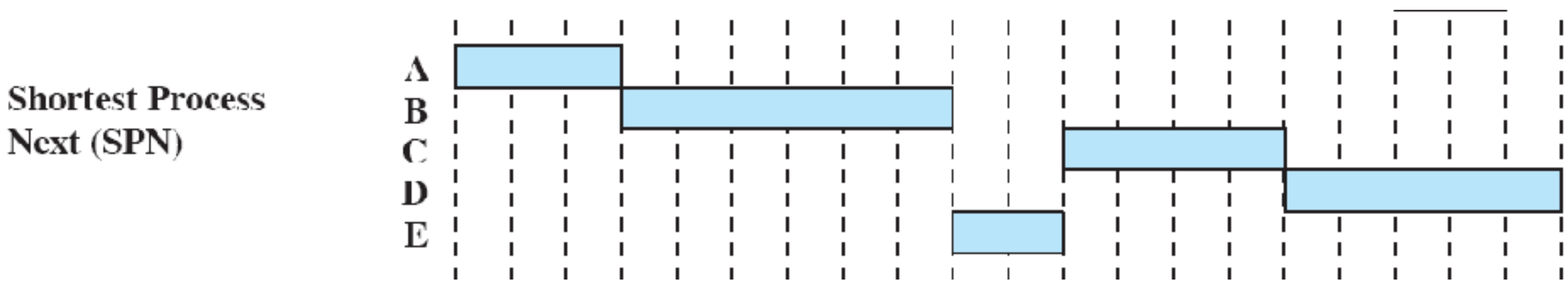
FIFO Scheduling

- Tends to fair CPU-bound processes compared to I/O bound processes
  - Due to non-preemptive nature
  - If CPU bound process is in wait state when I/O bound process initiates I/O, CPU goes idle
    - Inefficient use of both CPU and I/O devices

Shortest Job Next Scheduling

- Associate length of next CPU burst with each process
- Assign process with shortest CPU burst requirement to the CPU
- Nonpreemptive scheduling
- Specially suitable for batch processing or longterm scheduling
- Ties broken by FIFI scheduling

Shortest Job Next Scheduling



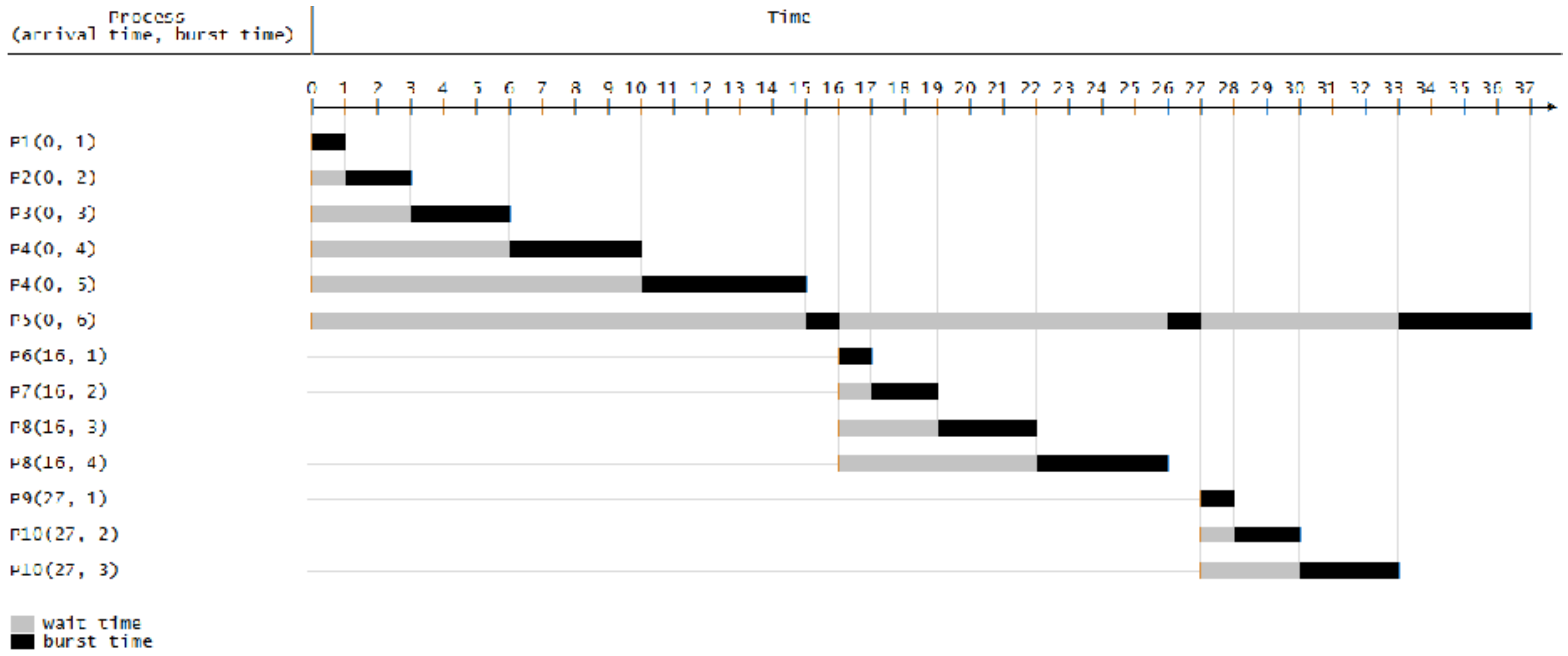
Shortest Job Next Scheduling

- Priority function $P(t) = 1/t$
- Provably optimal scheduling
  - Least average waiting time
    - Moving a short job before a long one decreases the waiting time for short job more than it increases waiting time for longer proc
  - Difficult in practice to determine length of CPU burst

Shortest Remaining Time First

- Preemptive version of shortest job next scheduling
- Preemptive in nature (only at arrival time)
- Highest priority to process that needs least time to complete
- Priority function $P(t) = a - t$, where a is arrival time

Shortest Job Next Scheduling



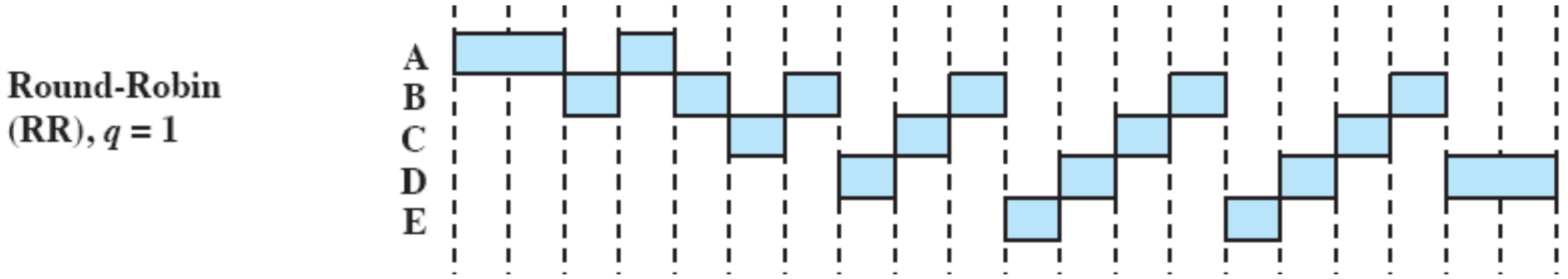
Round-Robin Scheduling

- Preemptive in nature, based on fixed time quanta q
    - Reduces penalty that short jobs suffer with FIFO
- Generate clock interrupts at periodic intervals
    - Time quantum between 10 and 100 milliseconds
    - Each process gets a slice of time before preemption
- All user processes treated to be at same priority

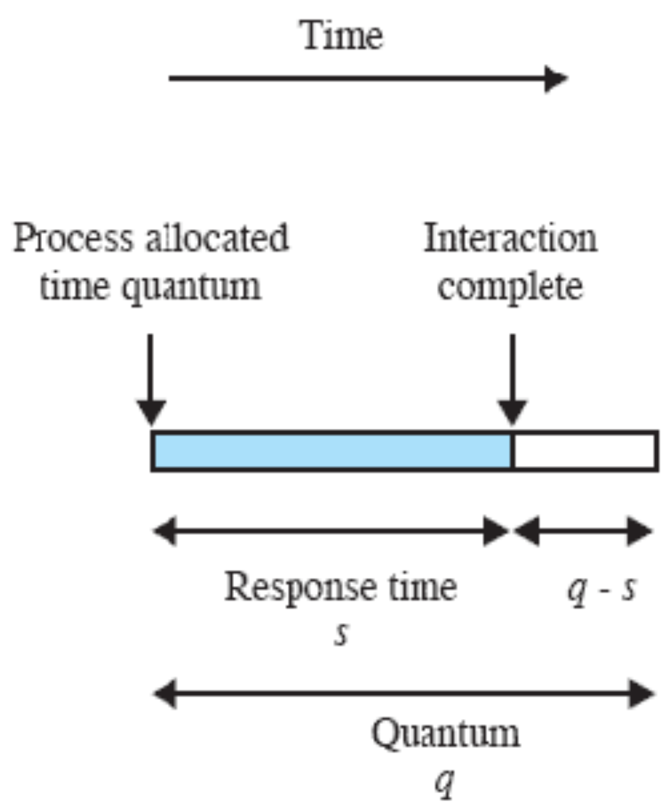
Round-Robin Scheduling

- Ready queue treated as circular
  - New processes added to the rear of ready queue
  - Preempted processes added to the rear of the ready queue
  - Scheduler pricks up a process form head of queue and dispatches it with a timer interrupt set after the time quantum
- CPU burst < *q* means process releases CPU voluntarily
- Timer interrupt results in context switch and process put at rear of queue
- No process is allocated CPU time for more than 1 quantum in a row

Round-Robin Scheduling

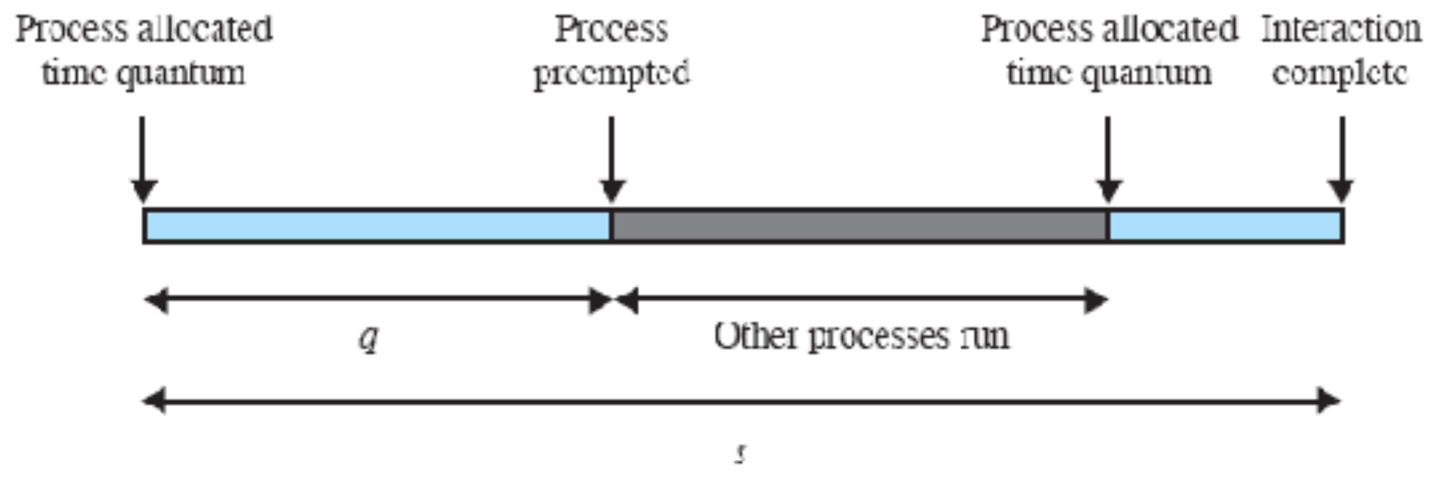


Round-Robin Scheduling



(a) Time quantum greater than typical interaction

Round-Robin Scheduling


Process allocated time quantum
Process preempted
Process allocated time quantum
Interaction complete
q
Other processes run
s

(b) Time quantum less than typical interaction

Figure 9.6 Effect of Size of Preemption Time Quantum

Round-Robin Scheduling

- One drawback is generally I/O bound processes have shorter process burst
    - Than a CPU bound process
    - Why?
- So processor-bound processes tend to receive unfair amount of processor time
- Can add an auxiliary queue of processes that were blocked on I/O
    - Take from that in preference to normal queue

Round-Robin Scheduling

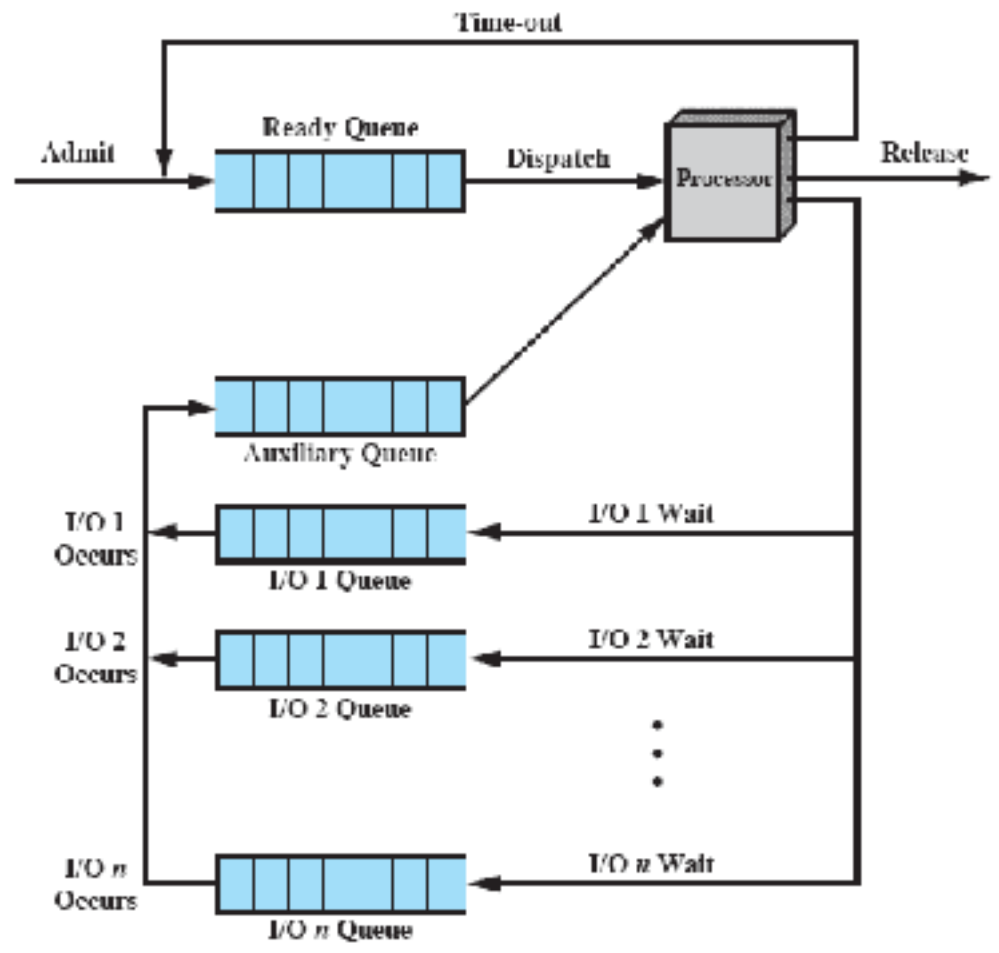


Figure 9.7 Queuing Diagram for Virtual Round-Robin Scheduler

Round-Robin Scheduling

- If *n* processes in ready queue, and *q* is time quantum, each gets 1/*n* of CPU time in chunks of at most *q* time units
- Each process waits no longer than (*n*-1)*q* time units for next quantum
- Performance depends heavily on size of time quantum
  - Large time quantum means essentially FIFO
  - Small time quantum means large context switching overhead
- Rule of thumb is 80% of CPU bursts should be shorter than time *q*

Multilevel Feedback Queue

- Most general CPU scheduling algorithm
- Prefers shorter jobs by penalizing jobs that use too much CPU time
  - Using dynamically computed priority
- Background
  - Make a distinction between foreground (interactive) and batch processes
  - Separate queues for different types of processes, with process priority being defined by what queue it is in
- Separates processes with different CPU burst requirements

Multilevel Feedback Queue

- Too much CPU time means lower priority
- I/O bound and interactive processes means higher priority
- *n* different priority levels: $\prod_1$ ... $\prod_n$
- Each process may not receive more than $T_\prod$ time units at priority $\prod$
- Let $T_\prod = 2^{n-\prod} T_n$, where T_n is the maximum time on highest priority queue at level n
  - Each process will spend T_n at priority n and time $2^i T_n$ at priority *n-i*, $1 <= i <= n$
  - Process may remain at lowest priority level for infinite time

Multilevel Feedback Queue

- CPU always serves highest priority queue
  - That has processes in it ready for execution
- Can result in starvation. How?
- Variation:
  - *Aging* to prevent starvation

Examples

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

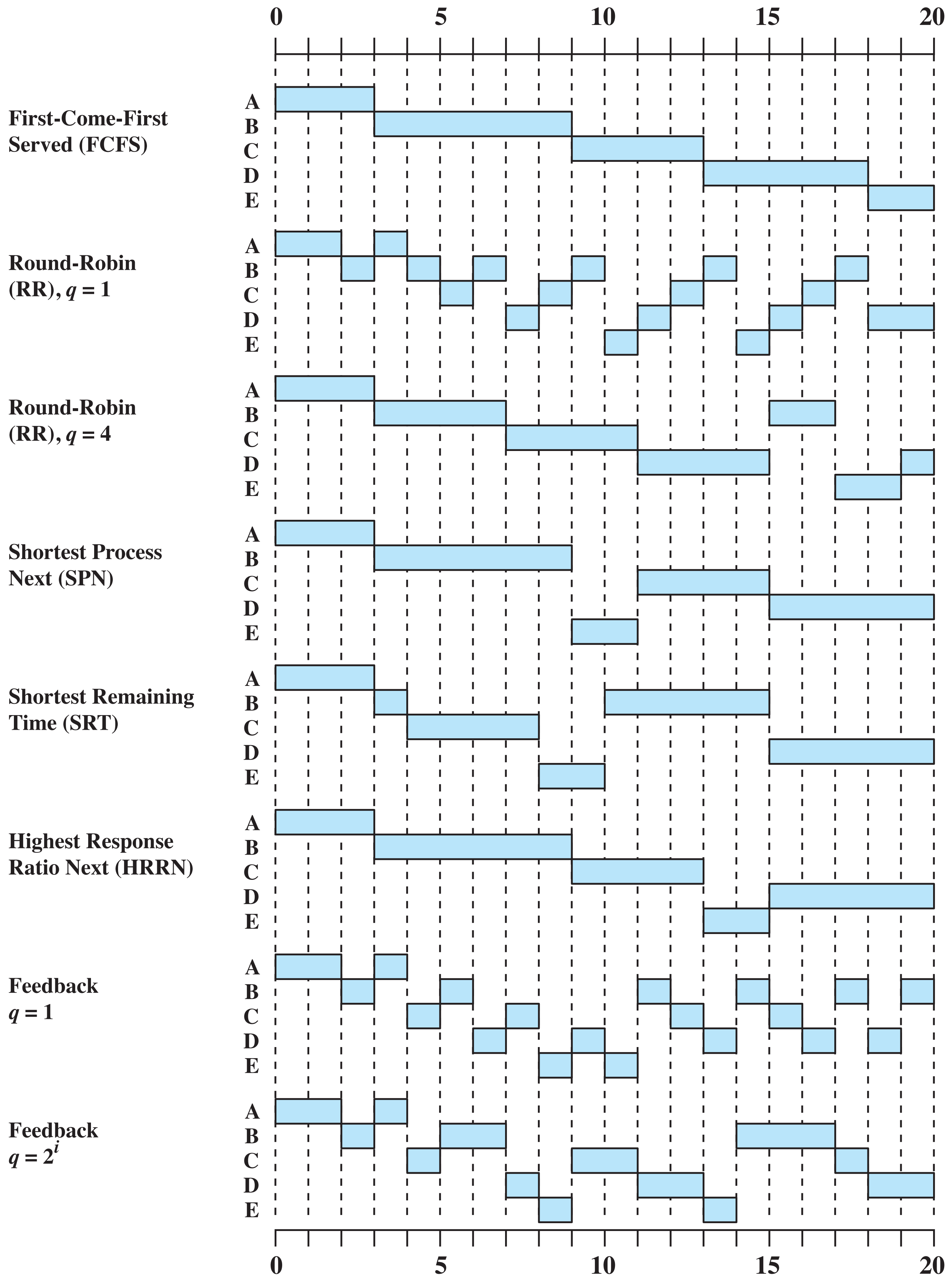


Figure 9.5 A Comparison of Scheduling Policies

Scheduling policy in Linux

- Linux objectives include:
    - Fast process response time
    - Good throughput for background jobs
    - Avoidance of process starvation
    - Reconcile the needs of low and high priority processes
- Based on time sharing
    - Several processes run in time-multiplexing mode, where users take turns

Multilevel Feedback Queue

- Process priority is dynamic
  - Scheduler keeps track of what the processes are doing and adjust their priorities
  - Processes that do not get CPU for a long time will have priority boosted by dynamically increasing their priority
  - Processes that run a long time are penalized by decreasing priority

Linux Scheduling

- Process classification
    - Interactive process
        - Interact constantly with users
        - Waiting for keypresses or mouse operations
        - When user input, need to respond quickly
        - Typical delay between 50-150ms
        - Variance of delay should be bounded or will seem erratic

Linux Scheduling

- Batch processes
    - No user interaction
    - Run in the background
    - Need not be very responsive and hence, are penalized by scheduler
    - May be I/O-bound or CPU-bound
- Real-time processes
    - Stringent scheduling requirements
    - Should never be blocked by lower priority processes
    - Short guaranteed response time with minimum variance

Linux Scheduling

- Linux scheduling pre-2.6
  - At every process switch:
    - Kernel scanned list of ready processes
    - Computes their priorities
    - Selects the one with highest priority
  - Expensive algorithm if list of ready processes is large

Linux Scheduling

- Linux Scheduling Post-2.6
  - Designed to scale well with number of ready processes
  - Selects process to run in constant time
    - Independent of the number of processes in ready queue
  - Designed to scale well with number of processors
    - Each processor has its own `runqueue`
  - Does a better job distinguishing between interactive and batch process

Linux Scheduling

- Scheduler always finds a process to be executed
  - There is always at least one runnable process
    - Always `swapper` with PID 0
    - Executes only when CPU cannot execute any other process
    - Every CPU in multiprocessor system has its own `swapper`

Linux Processes

- Linux scheduler
  - Assigns priority in range of 0 to 139
  - 140 different lists in `runqueue` structure
  - Process descriptor linked into list of runnable processes with same priority
  - On multiprocessor systems, each CPU has its own `runqueue`

Linux Scheduling

- Every Linux process scheduled according to its *scheduling class*
- `SCHED_FIFO`: A FIFO real-time process
    - When scheduler assigns CPU to the process, it leaves the process descriptor in its current position in `runqueue`
    - If no other higher priority runnable real-time process, current process uses CPU as long as it needs even if there are other runnable real-time processes at same priority
- `SCHED_RR`:A round robin real-time process
    - When a scheduler assigns CPU to the process, it leaves the process descriptor at the tail of `runqueue`
    - Ensures a faire assignment of CPU time to all `SCHED_RR` real-time processes with same priority

Linux Scheduling

- SCHED_NORMAL: Conventional time-shared process
  - Scheduling of a conventional process
    - Each conventional process has its own static priority
    - Static priority used by scheduler to rate process with respect to other conventional processes in system
    - Static priority in range 100(highest) to 139(lowest)
    - New process inherits static priority of its parent
      - Can change with using nice(2) or setpriority(3c)

Linux Scheduling

- Have an adjustable time quanta based on CPU use
  - Q_b time quantum in ms
  - Based on its static priority *P_s*

$$Q_b = \begin{cases} (140 \quad P_s) \times 20 & \text{if } P_s < 120 \\ (140 - P_s) \times 5 & \text{if } P_s \geq 120 \end{cases}$$

  - Higher the static priority (lower its numerical value), longer the base quantum

Linux Scheduling

- Dynamic priority P_d and average sleep time
  - Ranges from 100 (highest) to 139 (lowest)
  - Actual priority used by scheduler to select a new process to run

$$P_d = \max(100, \min(P_s - b + 5, 139))$$

  - b is a bonus value ranging from 0 to 10
  - b < 5 is a penalty to lower P_d
  - b > 5 raises P_d
  - b depends on past history of the process
    - Related to average sleep time

Linux Scheduling

- Average sleep time
  - Given by average number of nanoseconds spent by process in sleep state
  - Sleeping in `TASK_INTERRUPTIBLE` state contributes to average sleep time in a different way from sleeping in `TASK_UNINTERRUPTIBLE`
  - Average sleep time decreases while process is running
  - Can never become larger than 1s

Linux Scheduling

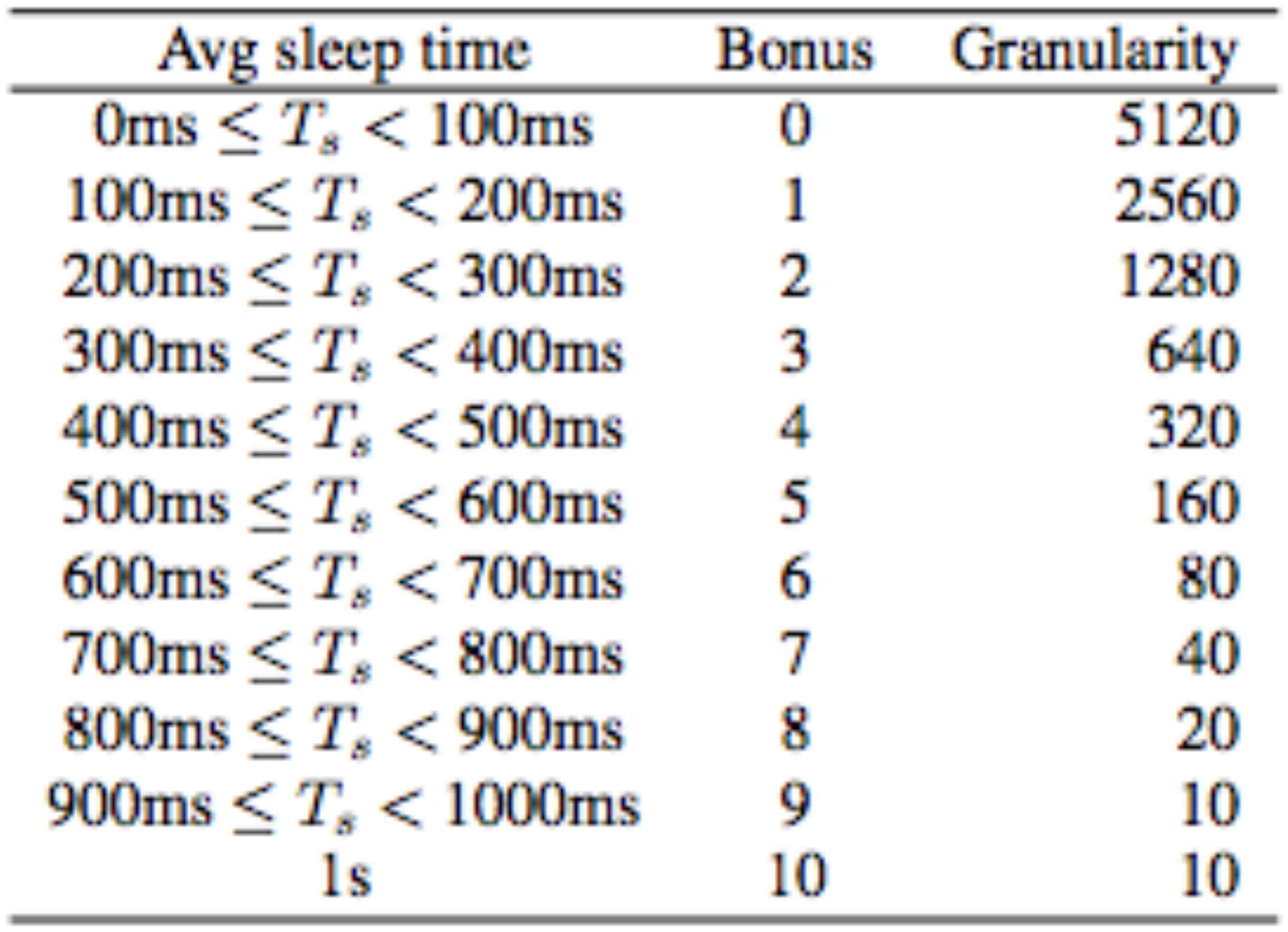
- Average sleeptime is also used to differentiate between batch/interactive
    - Interactive if it satisfies either of these equivalent statements

$$P_d <= 3 * P_s / 4 + 28$$

or

$$b - 5 >= P_s / 4 - 28$$

Average sleep time



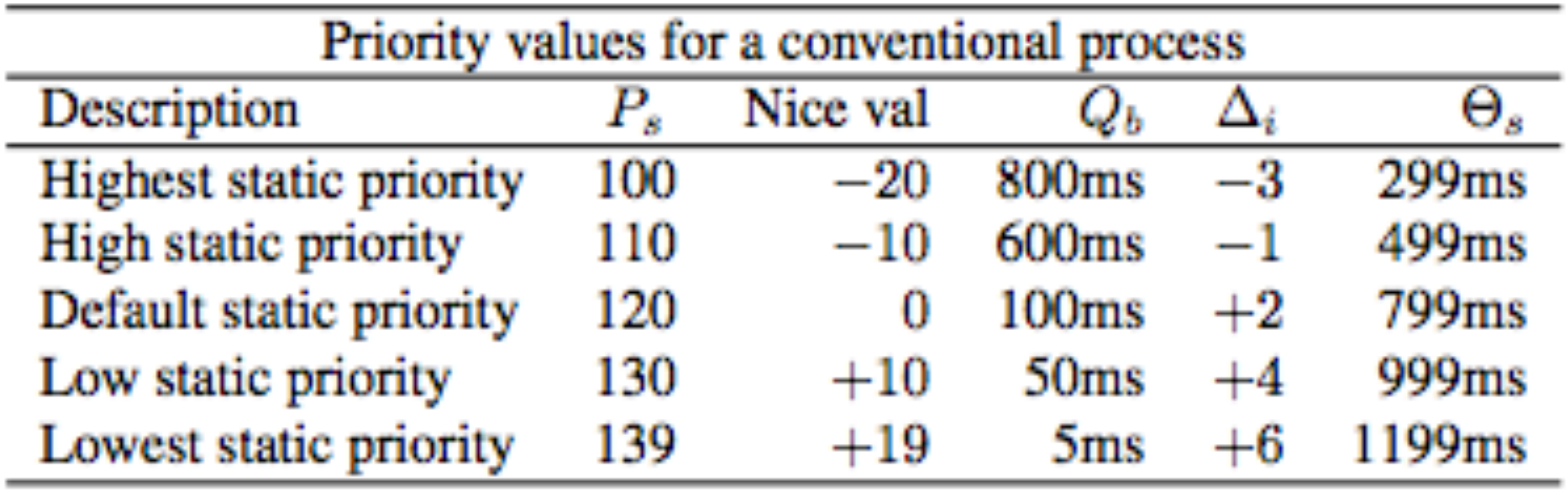
Avg sleep time	Bonus	Granularity
$0\text{ms} \le T_s < 100\text{ms}$	0	5120
$100\text{ms} \le T_s < 200\text{ms}$	1	2560
$200\text{ms} \le T_s < 300\text{ms}$	2	1280
$300\text{ms} \le T_s < 400\text{ms}$	3	640
$400\text{ms} \le T_s < 500\text{ms}$	4	320
$500\text{ms} \le T_s < 600\text{ms}$	5	160
$600\text{ms} \le T_s < 700\text{ms}$	6	80
$700\text{ms} \le T_s < 800\text{ms}$	7	40
$800\text{ms} \le T_s < 900\text{ms}$	8	20
$900\text{ms} \le T_s < 1000\text{ms}$	9	10
1s	10	10

Linux Scheduling

- Interactive delta Δ_i
  - Given by P_s `/ 4 - 28`
  - Easier for higher priority processes to become interactive
  - A process with 100 priority is considered interactive if b > 2
    - Or average sleep time > 200ms
  - Process with 139 priority is never considered interactive
    - b is always less than 100
  - Process with priority of 120 becomes interactive as soon as average sleeptime exceeds 700ms

Linux Scheduling

- Sleep threshold Θ_s



Priority values for a conventional process

Description	P_s	Nice val	Q_b	Δ_i	Θ_s
Highest static priority	100	-20	800ms	-3	299ms
High static priority	110	-10	600ms	-1	499ms
Default static priority	120	0	100ms	$+2$	799ms
Low static priority	130	$+10$	50ms	$+4$	999ms
Lowest static priority	139	$+19$	5ms	$+6$	1199ms

Linux Scheduling

- Scheduling of real-time processes
    - Real-time priority ranges from 1 (highest) to 99 (lowest)
    - Scheduler always favors higher priority runnable process over a lower priority one
    - Real-time processes always considered active
    - If there are several real-time runnable processes at same priority, scheduler chooses the process that occurs first in the corresponding list of local CPU's `runqueue`

Linux Scheduling

- Active and expired processes
  - processes with higher $\mathtt{P_s}$ get larger slice of CPU time but they should not completely lock out processes with lower $\mathtt{P_s}$
  - Process starvation is avoided by scheduling a lower priority process whose time quantum has not been exhausted when a higher priority process finishes its quantum
  - Active processes:
    - Runnable processes that have not exhausted their time quantum and are allowed to run
  - Expired processes:
    - Runnable processes that have exhausted their time quantum and are forbidden to run until all active processes expire

runqueue

```
struct runqueue {
    spinlock_t     lock;        /* spin lock which protects this
                                   runqueue */
    unsigned long    nr_running;     /* number of runnable tasks */
    unsigned long    nr_switches;     /* number of contextswitches */
    unsigned long    expired_timestamp;  /* time of last array swap */
    unsigned long    nr_uninterruptible; /* number of tasks in
                                                uinterruptible sleep */
    struct task_struct *curr;        /* this processor's currently
                                        running task */
    struct task_struct *idle;        /* this processor's idle task */
    struct mm_struct  *prev_mm;      /* mm_struct of last running task
                                        */
    struct prio_array  *active;       /* pointer to the active priority
                                          array */
    struct prio_array  *expired;      /* pointer to the expired
                                          priority array */
    struct prio_array  arrays[2];      /* the actual priority arrays */
    int         prev_cpu_load[NR_CPUS];/* load on each processor */
    struct task_struct *migration_thread;  /* the migration thread on this
                                               processor */
    struct list_head  migration_queue;   /* the migration queue for this
                                              processor */
    atomic_t      nr_iowait;      /* number of tasks waiting on I/O
                                   */
}
```

Linux Scheduling

- Data structures used by scheduler
    - `runqueue`
        - Each CPU has its own `runqueue`
        - Every runnable process in system belongs to exactly one `runqueue`
        - Runnable processes may migrate from one `runqueue` to another
            - ie: Move to another CPU

Linux Scheduling

- runqueue cont.
    - each structure also contains two `arrays` of 140 doubly linked list heads
        - One list corresponding to each priority (0-139)
        - `arrays[0]` contains expired processes
        - `arrays[1]` contains active processes
        - Role of each of these swaps periodically
            - `arrays[0]` holds active processes, `arrays[1]` holds expired processes

Linux Processes

- Linux process information cont.
  - `tasks->prev` field of `init_task` points to the `tasks` field of process descriptor inserted last in the list
  - Entire process list is scanned by following macro:

```
#define for_each_process(p) \
   for (p = &init_task; \
      (p = list_entry((p)->tasks.next,struct task_struct,tasks))\
         != &init_task; )
```

Linux Processes

- Lists of TASK_RUNNING processes
  - When looking for a new process to run on CPU, kernel has to consider only runnable processes
    - Processes in TASK_RUNNING state
  - Early Linux versions put all runnable processes in the same list called `runqueue`
- Scheduler speedup is achieved by splitting `runqueue` into many lists of runnable processes
  - One list per priority

Linux Processes

- `runqueue` implemented through `prio_array_t`

```
struct prio_array_t {
   int    nr_active;   // Number of active processes
   unsigned long bitmap[5]   // Priority bitmap; set if corresponding
                             // priority list is not empty
   struct list_head queue[140];   // head of each priority list
}
```

- Inserting a process p into the runqueue:

```
list_add_tail ( &p->run_list, &array->queue[p->prio] );
_set_bit ( p->prio, array->bitmap);
array->nr_active++;
p->array = array;
```

Linux Scheduling

- Allocating time slices to children
  - Both parent and child get half the number of ticks left to the parent
  - Done to prevent children from getting unlimited amount of CPU time
    - Parent creates a child that runs the same code and kills itself
    - If creation rate is adjusted properly, child can get full quantum before parent's quantum expires
    - Continuing this process allows infinite time to single process family
  - Also prevents single process from using excessive time
    - By starting several processes in background

Scheduling done!

- Any questions?