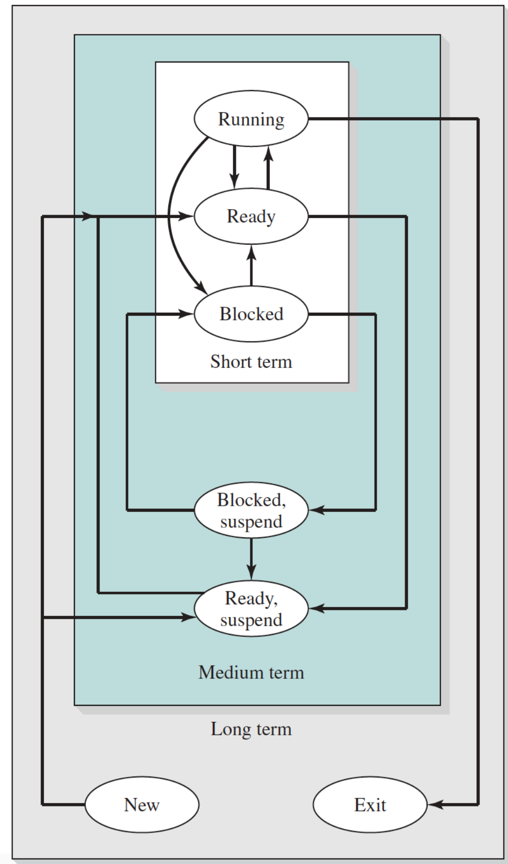
<u>**Chapter 9 (9.1 – 9.3)**</u>



- **Long-term scheduling** – deals with new processes
  - Can another process be created?
  - Which process to admit?
- **Medium-term scheduling** – deals with swapping
- **Short-term scheduling** – deals with which process to execute
  - Aka. the dispatcher
- **I/O scheduling** – which process's pending I/O request to handle

- Short-term scheduling criteria:

	User-oriented	*System-oriented*
Performance related	• Turnaround time • Response time • Deadlines	• Throughput • Processor utilization
Other	• Predictability	• Fairness (avoid starvation) • Enforcing priorities • Balancing resources

- Scheduling policies
  - Pure priority scheduling may cause starvation
  - Quantities:
    - *w* = time spent waiting so far
    - *e* = time spent executing so far
    - *s* = total service time (including *e*)
  - Turnaround time = waiting time + service time
  - Decision modes:
    - Non-preemptive – process will continue execution until it terminates or is blocked
    - Preemptive – currently running process may be pre-empted
  - First-come first-served
    - Bad for short processes
  - Round robin
    - Inefficient for I/O processes
    - Virtual round robin adds an <u>auxiliary queue</u> – dispatcher favours I/O processes that just became unblocked
  - Shortest process next – needs to know *s*
    - Long processes have less predictability, may be starved
  - Shortest remaining time – needs to know *s*
    - Long processes may still be starved
    - Better turnaround time than SPN
  - Highest response ratio next – needs to know *s*

- Picks based on normalized turnaround time
- Feedback
  - Every time a process is pre-empted it is demoted in priority
  - Long processes can be starved

	FCFS	Round Robin	SPN	SRT	HRRN	Feedback
Selection Function	$\max[w]$	constant	$\min[s]$	$\min[s - e]$	$\max\left(\frac{w + s}{s}\right)$	(see text)
Decision Mode	Non-preemptive	Preemptive (at time quantum)	Non-preemptive	Preemptive (at arrival)	Non-preemptive	Preemptive (at time quantum)
Throughput	Not emphasized	May be low if quantum is too small	High	High	High	Not emphasized
Response Time	May be high, especially if there is a large variance in process execution times	Provides good response time for short processes	Provides good response time for short processes	Provides good response time	Provides good response time	Not emphasized
Overhead	Minimum	Minimum	Can be high	Can be high	Can be high	Can be high
Effect on Processes	Penalizes short processes; penalizes 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

- Fair share scheduling
  - In some systems the process pool need to be regarded as a collection of process sets, each with some weighting that determines their share of system resources
- **Unix scheduling**
  - Uses multi-level feedback
  - Round robin within priority queues
  - Processes are pre-empted & priorities recalculated every second
  - Processes are restricted within their priority bands: (decreasing priority)
    - Swapper
    - Block I/O device control
    - File manipulation
    - Character I/O device control
    - User processes

<u>Chapter 10 (10.1 – 10.5)</u>

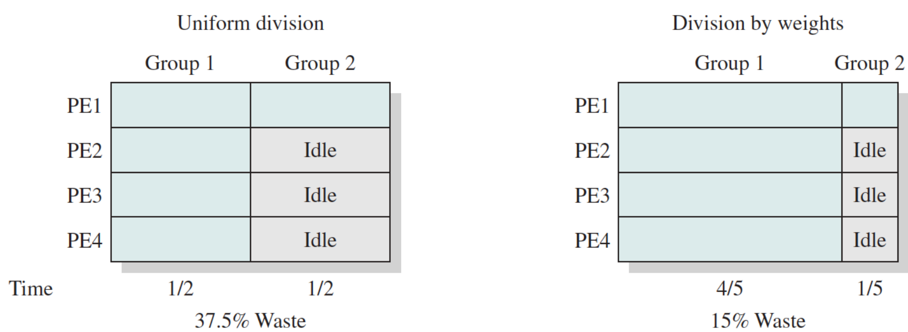
- Types of multiprocessor systems
  - Loosely coupled/distributed multiprocessor (aka. cluster)
  - Functionally specialized processors
  - Tightly coupled multiprocessor
- **Granularity** – frequency of synchronization

Grain Size	Description	Synchronization Interval (Instructions)
Fine	Parallelism inherent in a single instruction stream	< 20
Medium	Parallel processing or multitasking within a single application	20–200
Coarse	Multiprocessing of concurrent processes in a multiprogramming environment	200–2,000
Very Coarse	Distributed processing across network nodes to form a single computing environment	2,000–1M
Independent	Multiple unrelated processes	Not applicable

  - Independent parallelism – no synchronization among processes
    - E.g. time sharing system
  - Coarse/very coarse – handled as set of concurrent processors on a uniprocessor
    - Can be supported on multiprocessor with little change
  - Medium/fine – coordination & interaction between threads in a program
- Design issues
  - Assignment of processes to processors
    - Static assignment vs. global queue vs. dynamic load balancing
    - <u>Master/slave</u> – kernel functions occupy a single processor
      - Master processor controls all memory & I/O, handles scheduling
      - Disadvantage: master can be a bottleneck; failure of master → failure of system
    - <u>Peer architecture</u> – kernel can execute on any processor; processor self-schedules
      - Disadvantage: complicates OS
  - Multiprogramming on individual processors
    - With many processors and higher granularity, it's no longer necessary for every processor to be busy all the time
  - Process dispatching
    - Sophisticated scheduling algorithms may be unnecessary

- **Thread scheduling**
  - Load sharing
    - Global queue of ready threads
    - Advantages:
      - No processor is left idle

            - No centralized scheduler required
        - Disadvantages:
            - Central queue requires mutual exclusion – bottleneck
            - Preemptive threads tend to resume on another process – inefficient for caches
            - Unlikely all threads of a process will be active at the same time
    - Gang scheduling
        - Set of threads simultaneously scheduled to set of processors
        - Great for highly synchronized threads (medium/fine grained)
        - Single-threaded applications can cause inefficient processor use

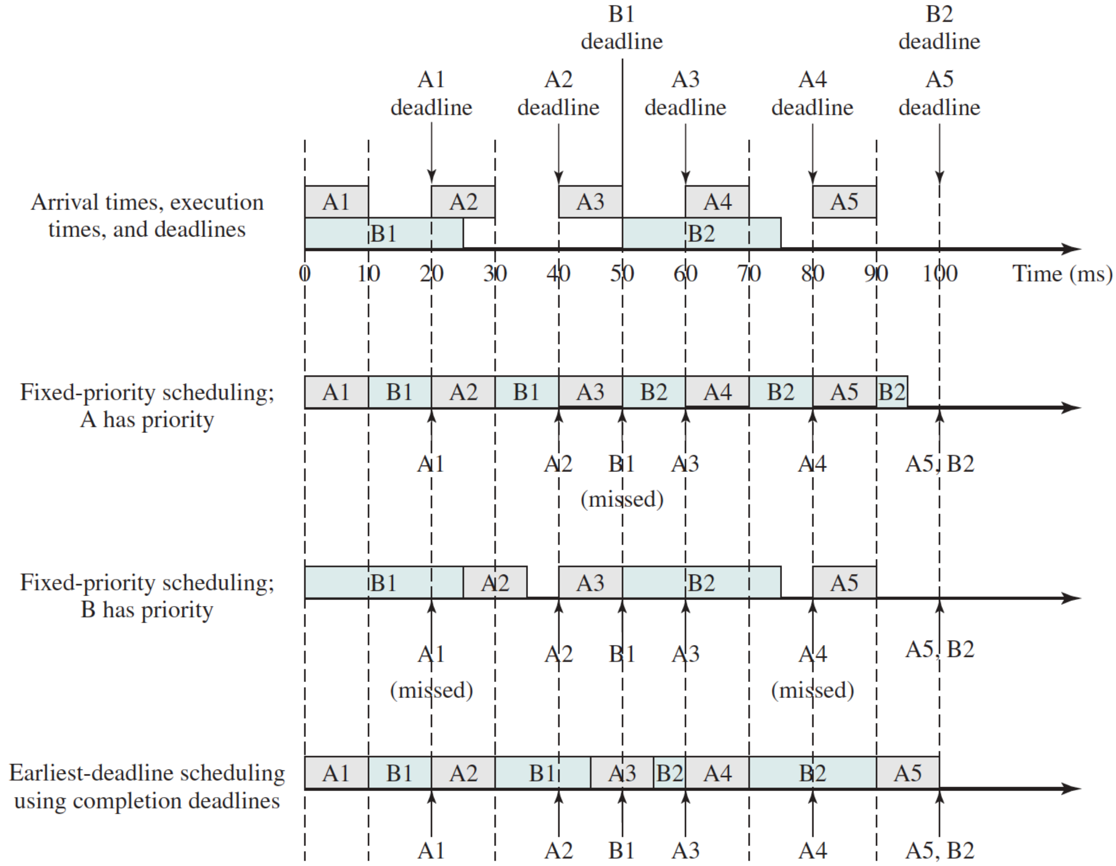


    - Dedicated processor assignment
        - Dedicate group of processors to an application; each thread gets a processor
        - Useful for highly parallel, many processor system; no processor switching ever
    - Dynamic scheduling
        - # of threads in each application can be altered dynamically
    - In multicore systems, minimizing access to off-chip memory (i.e. caches) takes precedence over maximizing processor utilization

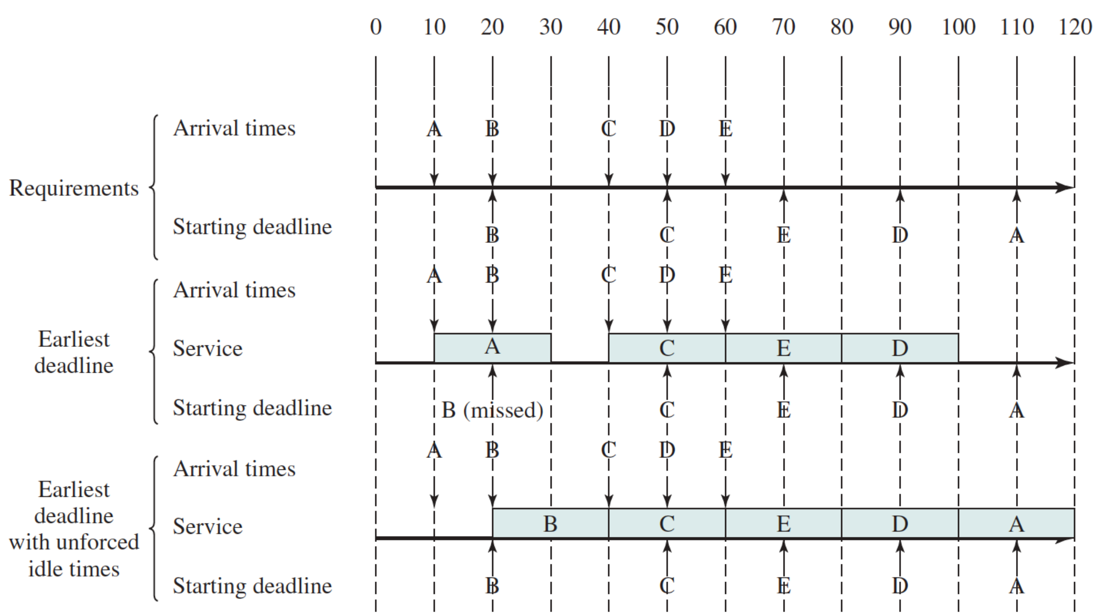
- **Real-time systems**
    - Correctness of the system depends on logical result as well as the time at which result is produced
    - Real-time tasks must be able to keep up with deadlines
        - Hard (necessary, else fault) vs. soft (desirable)
        - Periodic vs. aperiodic
- Characteristics of RT OS:
    - Determinism – perform operations at fixed, predetermined times/time intervals
        - Depends on interrupt response speed & capacity to handle requests
        - How long before interrupt is acknowledged
    - Responsiveness – after acknowledging, how long before interrupt is serviced
        - Depends on time to setup & perform the interrupt service routine, and interrupt nesting
    - User control – users have more fine-tuned control over priority, time, memory (e.g. paging), I/O, etc.
    - Reliability – degradation of performance have much more severe consequences
    - Fail-soft operation – ability to fail in a way that preserves as much capability and data as possible

    - Stability – when unable to meet all deadlines, prioritize most critical tasks
- Common features of RT OS:
  - Stricter use of priorities
  - Interrupt latency is bounded & short
  - More precise and predictable timing
  - Fast process/thread switching
  - Preemptive priority-based scheduling
- **Real-time scheduling**
  - Static table-driven
    - Perform static analysis, produces a schedule that fits requirements
    - Useful for periodic tasks
  - Static priority-driven preemptive
    - Perform static analysis and assign priorities to tasks; let traditional scheduler schedule them
  - Dynamic planning-based
    - Feasibility of task & schedule determined at run-time
    - Task is only accepted if deadline is feasible
  - Dynamic best effort
    - No feasibility analysis; system tries to meet all deadlines at runtime
    - <u>Commonly used in practice</u>
- **Deadline scheduling**
  - Real-time is less concerned with speed, and more with completing tasks at the right time
  - Schedule tasks based on info about them:
    - Ready time
    - Starting deadline
    - Completion deadline
    - Processing time
    - Resource requirements
    - Priority
    - Subtask structure
  - Scheduling based on <u>earliest (starting/completion) deadlines</u> minimizes tasks that miss their deadlines (EDF)
    - Starting deadlines → use non-preemptive scheduling
    - Completion deadlines → use preemptive scheduling
    - Can achieve 100% CPU utilization with preemption

- Example:
  - A has execution time = 10, deadline every 20
  - B has execution time = 25, deadline every 50



- Example with aperiodic tasks:



- **Rate monotonic scheduling**
    - Tasks with shorter periods → higher priority
    - Commonly adopted in industry
- Priority inversion
    - Circumstances in system forces a higher-priority task to wait for a lower-priority task
    - E.g. lower-p locks a resource, and higher-p tries to lock the same resource
    - Unbounded priority inversion
        - Duration of priority inversion depends on unpredictable actions of unrelated tasks
    - Priority inheritance
        - Lower-p task temporarily inherits the priority of any higher-p task that is waiting on a resource they have
    - Priority ceiling – resource & process that accesses it is given a temporary higher priority

- **Linux scheduling**
    - SCHED_FIFO – FIFO real-time threads
    - SCHED_RR – round robin real-time threads
        - Real-time threads have priorities 0 – 99
    - SCHED_OTHER – non-real-time threads
        - Priorities 100 – 139
        - No preemption
    - 140-bit priority array each for the active and expired priority queues (140 queues each)
        - Each array cell points to the queue for that priority
        - Aka. the O(1) scheduler
- **UNIX SVR4 scheduling**
    - Queue for each priority is executed in round robin

Priority class	Global value	Scheduling sequence
Real time	159 • • • • 100	First ↓
Kernel	99 • • 60	↓
Time shared	59 • • • • 0	↓ Last

- **UNIX FreeBSD scheduling**

Priority Class	Thread Type	Description
0–63	Bottom-half kernel	Scheduled by interrupts. Can block to await a resource
64–127	Top-half kernel	Runs until blocked or done. Can block to await a resource
128–159	Real-time user	Allowed to run until blocked or until a higher-priority thread becomes available. Preemptive scheduling
160–223	Time-sharing user	Adjusts priorities based on processor usage
224–255	Idle user	Only run when there are no time sharing or real-time threads to run
Note: Lower number corresponds to higher priority.		