Operating Systems Notes - Chapter 5

June 9, 2025

Contents

1 CPU Scheduling

1.1 Basic Concepts

- CPU scheduling is fundamental to multiprogrammed operating systems, maximizing CPU utilization by switching the CPU among processes.
- On modern operating systems, kernel-level threads are scheduled, though "process scheduling" and "thread scheduling" are often used interchangeably.
- A process executes on a CPU's core; general terminology of scheduling a process to "run on a CPU" implies running on a core.

1.2 Basic concepts

- In a single CPU core system, only one process runs at a time; others wait. Multiprogramming aims to keep a process running at all times to maximize CPU utilization.
- When one process waits (e.g., for I/O), the OS gives the CPU to another process. This continues, ensuring productive use of waiting time.
- On multicore systems, this concept extends to all processing cores.
- CPU scheduling is a fundamental OS function, central to its design.

1.2.1 CPU-I/O burst cycle

- Process execution consists of a **cycle** of CPU execution and I/O wait, alternating between **CPU burst** and **I/O burst**.
- The final CPU burst ends with a system request to terminate execution.
- Durations of CPU bursts vary but tend to have an exponential or hyperexponential frequency curve (many short, few long bursts).
- I/O-bound programs have many short CPU bursts; CPU-bound programs have a few long CPU bursts. This distribution is important for CPU-scheduling algorithms.

1.2.2 CPU scheduler

- When the CPU becomes idle, the operating system selects a process from the ready queue to execute.
- The **CPU scheduler** performs this selection and allocates the CPU to the chosen process.
- The ready queue is not necessarily FIFO; it can be a FIFO queue, priority queue, tree, or unordered linked list.
- Records in queues are generally process control blocks (PCBs).

1.2.3 Preemptive and nonpreemptive scheduling

- CPU-scheduling decisions occur under four circumstances:
  1. Process switches from running to waiting state (e.g., I/O request, `wait()` for child termination).
  2. Process switches from running to ready state (e.g., interrupt occurs).
  3. Process switches from waiting to ready state (e.g., I/O completion).
  4. Process terminates.
- For circumstances 1 and 4, no scheduling choice; a new process must be selected. Choices exist for 2 and 3.
- **Nonpreemptive** or **cooperative** scheduling: CPU allocated to a process until it releases it (terminates or switches to waiting state).
- **Preemptive** scheduling: CPU can be taken away from a process. Most modern OS (Windows, macOS, Linux, UNIX) use preemptive scheduling.
- Preemptive scheduling can cause race conditions with shared data (e.g., one process updates, is preempted, second process reads inconsistent data).
- Preemption affects OS kernel design:
  - Nonpreemptive kernel: waits for system call completion or process block before context switch, ensuring simple kernel structure and consistent data. Poor for real-time computing.
  - Preemptive kernel: requires mechanisms (e.g., mutex locks) to prevent race conditions when accessing shared kernel data structures. Most modern OS are fully preemptive in kernel mode.
- Sections of code affected by interrupts must be guarded (e.g., disable interrupts at entry, reenable at exit) to prevent simultaneous use and data loss.

1.2.4 Dispatcher

- The **dispatcher** is a component of the CPU-scheduling function.
- It gives control of the CPU's core to the process selected by the CPU scheduler.
- Functions include:
  - Switching context from one process to another.
  - Switching to user mode.

  - Jumping to the proper location in the user program to resume that program.
- The dispatcher should be fast, as it's invoked during every context switch.
- **Dispatch latency**: time for the dispatcher to stop one process and start another.
- Context switch frequency can be observed using tools like `vmstat` on Linux (system-wide) or `/proc` file system (per-process).
- **Voluntary context switch**: process gives up CPU (e.g., blocking for I/O).
- **Nonvoluntary context switch**: CPU taken from process (e.g., time slice expired, preempted by higher-priority process).

Section glossary

Term	Definition
cycle	Repeating loop
CPU burst	Scheduling process state in which the process executes on CPU.
I/O burst	Scheduling process state in which the CPU performs I/O.
CPU scheduler	Kernel routine that selects a thread from the threads that are ready to execute and allocates a core to that thread.
nonpreemptive	Under nonpreemptive scheduling, once a core has been allocated to a thread the thread keeps the core until it releases the core either by terminating or by switching to the waiting state.
cooperative	A form of scheduling in which threads voluntarily move from the running state.
preemptive	A form of scheduling in which processes or threads are involuntarily moved from the running state (by for example a timer signaling the kernel to allow the next thread to run).
dispatcher	The kernel routine that gives control of a core to the thread selected by the scheduler.
dispatch latency	The time it takes for the dispatcher to stop one thread and start another running.

2 Scheduling criteria

2.1 Overview

- Different CPU-scheduling algorithms have varying properties, favoring certain process classes.
- The choice of algorithm depends on the desired characteristics for comparison.

2.2 Criteria for Comparison

- **CPU utilization**: Keep the CPU as busy as possible (ideally 40-90% in real systems).
- **Throughput**: Number of processes completed per unit time.
- **Turnaround time**: Total time from process submission to completion (includes waiting in ready queue, CPU execution, and I/O).
- **Waiting time**: Total time a process spends waiting in the ready queue.
- **Response time**: Time from request submission until the first response is produced (for interactive systems).

2.3 Optimization Goals

- Maximize CPU utilization and throughput.
- Minimize turnaround time, waiting time, and response time.
- Often, the goal is to optimize the average measure, but sometimes minimum or maximum values are preferred (e.g., minimizing maximum response time for guaranteed service).
- For interactive systems, minimizing the variance in response time may be more important than minimizing the average.

Section glossary

Term	Definition
throughput	Generally, the amount of work done over time. In scheduling, the number of threads completed per unit time.

3 Scheduling algorithms

3.1 Overview

- CPU scheduling involves deciding which process in the ready queue is allocated the CPU's core.
- Various CPU-scheduling algorithms exist.
- These algorithms are described in the context of a single processing core, capable of running one process at a time. Multiprocessor scheduling is discussed in Section 5.5.
- For demonstration, each process is associated with a CPU burst time, although in practice, this is not known in advance.

3.2 First-come, first-served scheduling

- The simplest CPU-scheduling algorithm: **first-come, first-served** (**FCFS**).
- The process that requests the CPU first is allocated the CPU first.
- Implemented with a FIFO queue: process PCBs are linked to the tail, and the CPU is allocated to the process at the head.
- Average waiting time under FCFS is often long and can vary substantially with CPU burst times.
- **Convoy effect**: A CPU-bound process holds the CPU, causing I/O-bound processes to wait in the ready queue, leading to lower CPU and device utilization.
- FCFS is nonpreemptive; once allocated, a process keeps the CPU until it terminates or requests I/O. This is problematic for interactive systems.
- A **Gantt chart** is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes.

3.3 Shortest-job-first scheduling

- The **shortest-job-first** (**SJF**) scheduling algorithm associates each process with the length of its next CPU burst.
- The CPU is assigned to the process with the smallest next CPU burst. FCFS breaks ties.
- More accurately called the **shortest-next-CPU-burst** algorithm.
- SJF is provably optimal, providing the minimum average waiting time for a given set of processes.
- Cannot be implemented at the CPU scheduling level directly, as the length of the next CPU burst is unknown.
- Can be approximated by predicting the next CPU burst using an **exponential average** of previous CPU bursts:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\,\tau_n.$$

  - t_n: length of the nth CPU burst.
  - τ_{n+1}: predicted value for the next CPU burst.
  - α: parameter ($0 \leq \alpha \leq 1$) controlling the weight of recent vs. past history.
  - If $\alpha = 0$, $\tau_{n+1} = \tau_n$ (recent history has no effect).
  - If $\alpha = 1$, $\tau_{n+1} = t_n$ (only most recent CPU burst matters).
  - Commonly, $\alpha = 1/2$, weighting recent and past history equally.
- SJF can be preemptive or nonpreemptive.
- Preemptive SJF is called **shortest-remaining-time-first** scheduling. It preempts the current process if a new process has a shorter remaining CPU burst.

3.4 Round-robin scheduling

- The **round-robin** (**RR**) scheduling algorithm is similar to FCFS but includes preemption.
- A small unit of time, called a **time quantum** or **time slice** (typically 10-100 milliseconds), is defined.
- The ready queue is treated as a circular queue. The CPU scheduler allocates the CPU to each process for up to 1 time quantum.
- If a process finishes its CPU burst before the time quantum expires, it voluntarily releases the CPU.
- If a process's CPU burst is longer than 1 time quantum, it is preempted by a timer interrupt and moved to the tail of the ready queue.
- RR is a preemptive scheduling algorithm.
- If there are n processes and a time quantum of q, each process gets $1/n$ of the CPU time in chunks of at most q time units. Each process waits no longer than $(n-1)q$ time units.
- Performance depends heavily on time quantum size:
  - Large quantum: RR degenerates to FCFS.
  - Small quantum: results in many context switches, increasing overhead.
- Time quantum should be large relative to context-switch time (e.g., context-switch time ¡ 10% of time quantum).
- Average turnaround time does not necessarily improve with increasing time quantum; it improves if most processes finish their CPU burst within one quantum.
- Rule of thumb: 80% of CPU bursts should be shorter than the time quantum.

3.5 Priority scheduling

- The SJF algorithm is a special case of the general **priority-scheduling** algorithm.
- A priority is associated with each process; the CPU is allocated to the process with the highest priority.
- Equal-priority processes are scheduled in FCFS order.
- SJF is a priority algorithm where priority is the inverse of the (predicted) next CPU burst (shorter burst = higher priority).
- Priorities can be defined internally (e.g., time limits, memory requirements) or externally (e.g., importance of process, payment).
- Priority scheduling can be preemptive (new higher-priority process preempts current) or nonpreemptive (new higher-priority process goes to head of queue).
- Major problem: **indefinite blocking** or **starvation**, where low-priority processes may wait indefinitely for the CPU.
- Solution to starvation: **aging**, which gradually increases the priority of processes that wait for a long time.
- Can combine RR and priority scheduling: highest-priority process executes, and processes with the same priority use RR.

3.6 Multilevel queue scheduling

- **Multilevel queue** scheduling partitions the ready queue into several separate queues.
- Each queue may have its own scheduling algorithm (e.g., foreground/interactive processes with RR, background/batch processes with FCFS).
- Scheduling among queues is commonly fixed-priority preemptive scheduling (e.g., real-time queue has absolute priority over interactive queue).
- Processes are typically permanently assigned to a queue.
- Can also time-slice among queues (e.g., foreground queue gets 80% CPU time, background gets 20%).

3.7 Multilevel feedback queue scheduling

- The **multilevel feedback queue** scheduling algorithm allows a process to move between queues.
- Separates processes by CPU burst characteristics: processes using too much CPU time are moved to lower-priority queues.
- This keeps I/O-bound and interactive processes (short CPU bursts) in higher-priority queues.
- Aging is implemented by moving processes that wait too long in lower-priority queues to higher-priority queues, preventing starvation.
- Example: Queue 0 (8ms quantum), Queue 1 (16ms quantum), Queue 2 (FCFS). Processes start in Queue 0, move to Queue 1 if not finished, then to Queue 2.
- Defined by: number of queues, scheduling algorithm for each queue, methods for upgrading/demoting processes, and method for initial queue entry.
- Most general and configurable CPU-scheduling algorithm, but also the most complex to define optimally.

Section glossary

Term	Definition
First-come first-served (FCFS)	The simplest scheduling algorithm - the thread that requests a core first is allocated the core first, and others following get cores in the order of their requests.
Gantt chart	A bar chart that is used in the text to illustrate a schedule.
convoy effect	A scheduling phenomenon in which threads wait for the one thread to get off a core, causing overall device and CPU utilization to be suboptimal.
shortest-job-first (SJF)	A scheduling algorithm that associates with each thread the length of the threads next CPU burst and schedules the shortest first.
exponential average	A calculation used in scheduling to estimate the next CPU burst time based on the previous burst times (with exponential decay on older values).
shortest-remaining-time-first (SJRF)	Similar to SJF, this scheduling algorithm optimizes for the shortest remaining time until thread completion.
round-robin (RR)	A scheduling algorithm that is designed especially for time-sharing systems - similar to FCFS scheduling, but preemption is added to enable the system to switch between threads.
time quantum	A small unit of time used by scheduling algorithms as a basis for determining when to preempt a thread from the CPU to allow another to run.
time slice	See time quantum.
priority-scheduling	A scheduling algorithm in which a priority is associated with each thread and the free CPU core is allocated to the thread with the highest priority.
infinite blocking	See starvation.
starvation	A scheduling risk in which a thread that is ready to run never gets put onto the CPU due to the scheduling algorithm - it is starved for CPU time.
aging	Aging is a solution to scheduling starvation and involves gradually increasing the priority of threads as they wait for CPU time.
multilevel queue	A multilevel queue scheduling algorithm partitions the ready queue into several separate queues.
foreground	A thread that is interactive and has input directed to it (such as a window currently selected as active or a terminal window that is currently selected to receive input).
background	A thread that is not currently interactive (has no interactive input directed to it) such as one in a batch job or not currently being used by a user.
multilevel feedback queue	The multilevel feedback queue scheduling algorithm that allows a process to move between queues.

4 Thread scheduling

4.1 Overview

- Modern operating systems schedule kernel-level threads, not processes.
- User-level threads are managed by a thread library and must be mapped to kernel-level threads (possibly via a lightweight process, LWP) to run on a CPU.
- This section explores scheduling issues for user-level and kernel-level threads, with Pthread examples.

4.2 Contention scope

- Distinction in scheduling between user-level and kernel-level threads:
  - **Process-contention scope** (**PCS**): Thread library schedules user-level threads onto available LWPs. Competition for the CPU occurs among threads within the same process.
  - **System-contention scope** (**SCS**): Kernel schedules kernel-level threads onto a CPU. Competition for the CPU occurs among all threads in the system.
- Systems using the one-to-one model (e.g., Windows, Linux) schedule threads using only SCS.
- PCS typically uses priority-based scheduling, preempting lower-priority threads. No guarantee of time slicing among equal-priority threads.

4.3 Pthread scheduling

- POSIX Pthread API allows specifying PCS or SCS during thread creation using:
  - `PTHREAD_SCOPE_PROCESS`: schedules threads using PCS.
  - `PTHREAD_SCOPE_SYSTEM`: schedules threads using SCS.
- On many-to-many systems, `PTHREAD_SCOPE_PROCESS` schedules user-level threads onto LWPs managed by the thread library.
- `PTHREAD_SCOPE_SYSTEM` on many-to-many systems creates and binds an LWP for each user-level thread, effectively using a one-to-one policy.
- Functions for setting/getting contention scope:
  - `pthread_attr_setscope(pthread_attr_t *attr, int scope)`
  - `pthread_attr_getscope(pthread_attr_t *attr, int *scope)`
- Some systems (e.g., Linux, macOS) only allow `PTHREAD_SCOPE_SYSTEM`.

Section glossary

Term	Definition
process-contention scope (PCS)	On systems implementing the many-to-one and many-to-many threading models, the thread library schedules user-level threads to run on an available LWP (and thus threads contend with others within the same process contend for CPU time).
system-contention scope (SCS)	A thread scheduling method in which kernel-level threads are scheduled onto a CPU, regardless of which process they are associated with (and thus contending with all other threads on the system for CPU time).

5 Multi-processor scheduling

5.1 Overview

- Scheduling in systems with multiple processing cores introduces **load sharing** and increased complexity.
- The term **multiprocessor** now applies to multicore CPUs, multithreaded cores, NUMA systems, and heterogeneous multiprocessing.
- This section discusses multiprocessor scheduling concerns in these architectures, focusing on homogeneous processors first, then heterogeneous.

5.2 Approaches to multiple-processor scheduling

- **Asymmetric multiprocessing**: A single main server processor handles all scheduling, I/O, and system activities; other processors execute user code.
  - Simple, reduces data sharing.
  - Main server can become a performance bottleneck.
- **Symmetric multiprocessing** (**SMP**): Each processor is self-scheduling.
  - Processors examine a ready queue and select a thread to run.
  - Two strategies for organizing threads:
    1. Common ready queue: All threads in one queue. Potential race conditions require locking, which can be a performance bottleneck.
    2. Private per-processor queues: Each processor has its own queue. Avoids locking overhead and is common in SMP systems. Benefits from processor affinity.
  - Most modern OS (Windows, Linux, macOS, Android, iOS) support SMP.

5.3 Multicore processors

- Contemporary hardware uses **multicore processor** chips, where multiple computing cores reside on the same physical chip.
- Each core appears as a separate logical CPU to the operating system.
- Multicore processors are faster and consume less power than systems with separate physical CPU chips.
- **Memory stall**: Processor waits for data from memory (e.g., due to cache miss), wasting significant time.
- To mitigate memory stalls, many designs implement multithreaded processing cores with two or more **hardware threads** per core.
- **Chip multithreading** (**CMT**), also known as **hyper-threading** or **simultaneous multithreading** (SMT), allows a core to switch to another hardware thread if one stalls.
- Two ways to multithread a processing core:
  - **Coarse-grained multithreading**: Switches threads on long-latency events (e.g., memory stall). High switching cost due to pipeline flushing.
  - **Fine-grained multithreading**: Switches threads at a finer granularity (e.g., instruction cycle boundary). Low switching cost due to architectural design.
- Multithreaded, multicore processors require two levels of scheduling:
  1. Operating system schedules software threads onto hardware threads (logical CPUs).
  2. Each core decides which hardware thread to run (e.g., round-robin, urgency-based).
- OS scheduling can be more effective if it is aware of processor resource sharing (e.g., scheduling software threads on separate cores to avoid sharing resources).

5.4 Load balancing

- **Load balancing** attempts to keep the workload evenly distributed across all processors in an SMP system.
- Necessary for systems with private per-processor ready queues. Unnecessary for common ready queues.
- Two general approaches:
  - **Push migration**: A task periodically checks processor loads and moves threads from overloaded to idle/less-busy processors.
  - **Pull migration**: An idle processor pulls a waiting task from a busy processor.
- Both push and pull migration can be implemented in parallel.
- "Balanced load" can mean equal number of threads or equal distribution of thread priorities.

5.5 Processor affinity

- **Processor affinity**: A process has an affinity for the processor on which it is currently running.
- Benefits from "warm cache" (data recently accessed by the thread populates the processor's cache).
- Migrating a thread to another processor incurs cost of invalidating and repopulating caches.
- Private per-processor ready queues naturally provide processor affinity.
- Forms of processor affinity:
  - **Soft affinity**: OS attempts to keep a process on the same processor but does not guarantee it (migration can occur during load balancing).
  - **Hard affinity**: System calls allow a process to specify a subset of processors on which it can run.
- Load balancing often counteracts processor affinity benefits.
- Non-uniform memory access (NUMA) systems: CPUs have faster access to local memory. NUMA-aware scheduling and memory placement can improve performance by allocating memory closest to the CPU running the thread.
- There is a tension between load balancing and minimizing memory access times in modern multicore NUMA systems.

5.6 Heterogeneous multiprocessing

- **Heterogeneous multiprocessing** (HMP): Systems with cores that run the same instruction set but vary in clock speed and power management.
- Not asymmetric multiprocessing; both system and user tasks can run on any core.
- Intention: better power consumption management by assigning tasks to cores based on demands.
- ARM's **big.LITTLE** architecture combines high-performance "big" cores with energy-efficient "LITTLE" cores.
  - "Big" cores: higher energy consumption, for short, high-performance tasks.
  - "LITTLE" cores: lower energy consumption, for longer background tasks.
- Advantages: preserves battery, assigns appropriate cores for interactive vs. background tasks, allows disabling energy-intensive cores in power-saving mode.
- Windows 10 supports HMP scheduling.

Section glossary

Term	Definition
load sharing	The ability of a system with multiple CPU cores to schedule threads on those cores.
asymmetric multiprocessing	A simple multiprocessor scheduling algorithm in which only one processor accesses the system data structures and others run user threads, reducing the need for data sharing.
symmetric multiprocessing	A multiprocessor scheduling method, where each processor is self-scheduling and may run kernel threads or user level threads (contending for access to kernel data structures and requiring locking).
memory stall	When a thread is on CPU and accesses memory contents that is not in the CPU's cache, the thread execution stalls while the contents of that memory is fetched.
hardware threads	A given CPU core can run a single thread (one hardware thread per core) or more than one per core (multiple hardware threads per core) to optimize core use, for example to avoid memory stalls by switching hardware threads if the current thread causes a stall.
chip multithreading (CMT)	A CPU with multiple cores, where each core supports multiple hardware threads supports chip multithreading.
hyper-threading	See chip multithreading.
simultaneous multithreading (SMT)	See chip multithreading.
Load balancing	Load balancing attempts to keep the workload evenly distributed across all processors in an SMP system.
push migration	With push migration, a specific task periodically checks the load on each processor and—if it finds an imbalance—evenly distributes the load by moving (or pushing) threads from overloaded to idle or less-busy processors.
pull migration	Pull migration occurs when an idle processor pulls a waiting thread from a busy processor.
processor affinity	A kernel scheduling method in which a process has an affinity for the processor in which it is currently running (to keep the cache warm for example).
soft affinity	When an operating system has a policy of attempting to keep a process running on the same processor—but not guaranteeing that it will do so.
hard affinity	When an operating system supports or allows a process's threads to run on the same processor at all times (as opposed to being moved to various processors as the thread is scheduled onto CPU).
heterogeneous multiprocessing (HMP)	A feature of some mobile computing CPUs in which cores vary in their clock speed and power management.
big.LITTLE	ARM processor implementation of HMP in which high performance big cores are combined with energy efficient LITTLE cores.

6 Real-time CPU scheduling

6.1 Overview

- CPU scheduling for real-time operating systems involves special considerations.
- **Soft real-time systems**: Provide no guarantee on when a critical real-time process will be scheduled, only preference over noncritical processes.
- **Hard real-time systems**: Have stricter requirements; a task must be serviced by its deadline, or it's considered a failure.
- This section explores scheduling issues in both soft and hard real-time OS.

6.2 Minimizing latency

- Real-time systems are often event-driven and must respond quickly to events.
- **Event latency**: The time elapsed from when an event occurs to when it is serviced.
- Two types of latencies affect real-time system performance:
  1. **Interrupt latency**: Time from interrupt arrival at CPU to the start of the interrupt service routine (ISR).
     - Involves completing current instruction, determining interrupt type, and saving current process state.
     - Must be minimized (and bounded for hard real-time systems).
     - Interrupts should be disabled for very short periods.
  2. **Dispatch latency**: Time for the scheduling dispatcher to stop one process and start another.
     - Must be minimized for real-time tasks to get immediate CPU access.
     - Achieved with preemptive kernels.
     - For hard real-time systems, typically measured in microseconds.
- Dispatch latency has a **conflict phase** (preemption of kernel processes, release of resources by low-priority processes) and a dispatch phase (scheduling high-priority process).

6.3 Priority-based scheduling

- Real-time OS schedulers must support a priority-based algorithm with preemption to respond immediately to real-time processes.
- Higher-priority processes preempt lower-priority ones.
- Providing a preemptive, priority-based scheduler guarantees only soft real-time functionality.
- Hard real-time systems require additional features to guarantee tasks meet deadlines.
- Characteristics of processes for hard real-time scheduling:
  - **Periodic**: Require CPU at constant intervals (periods).
  - Fixed processing time (t), deadline (d), and period (p). Relationship: $0 \leq t \leq d \leq p$.
  - **Rate** of a periodic task: $1/p$.
- Schedulers can assign priorities based on deadline or rate.
- **Admission-control** algorithm: Scheduler admits a process only if it can guarantee completion by its deadline; otherwise, it rejects the request.

6.4 Rate-monotonic scheduling

- Schedules periodic tasks using a static priority policy with preemption.
- Priority is inversely based on its period: shorter period = higher priority.
- Rationale: Assign higher priority to tasks requiring CPU more often.
- Assumes processing time is constant for each CPU burst.
- Considered optimal: if a set of processes cannot be scheduled by rate-monotonic, it cannot be scheduled by any other static-priority algorithm.
- Limitation: CPU utilization is bounded. Worst-case CPU utilization for N processes is $N\left(2^{1/N}-1\right)$.
  - For one process, 100%.
  - For two processes, approx. 83%.
  - Approaches 69% as $N \to \infty$.

6.5 Earliest-deadline-first scheduling

- **Earliest-deadline-first** (**EDF**) scheduling assigns priorities dynamically based on deadline: earlier deadline = higher priority.
- When a process becomes runnable, it announces its deadline. Priorities are adjusted dynamically.
- Unlike rate-monotonic, EDF does not require processes to be periodic or have constant CPU burst times.
- Theoretically optimal: can schedule processes to meet deadlines with 100% CPU utilization (in practice, limited by context switching and interrupt handling).

6.6 Proportional share scheduling

- **Proportional share** schedulers allocate shares among applications.
- An application with N shares out of a total T shares receives N/T of the total processor time.
- Works with an admission-control policy: a client is admitted only if sufficient shares are available.

6.7 POSIX real-time scheduling

- POSIX.1b defines two scheduling classes for real-time threads:
  - `SCHED_FIFO`: First-come, first-served policy with a FIFO queue. No time slicing among equal-priority threads; highest-priority thread runs until termination or blocking.
  - `SCHED_RR`: Round-robin policy, similar to `SCHED_FIFO` but provides time slicing among equal-priority threads.
- `SCHED_OTHER` is an additional, system-specific scheduling class.
- POSIX API functions for getting/setting scheduling policy:
  - `pthread_attr_getschedpolicy(pthread_attr_t *attr, int *policy)`
  - `pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy)`

Section glossary

Term	Definition
soft real-time systems	Soft real-time systems provide no guarantee as to when a critical real-time thread will be scheduled - they guarantee only that the thread will be given preference over noncritical threads
hard real-time systems	Hard real-time systems have strict scheduling facilities - a thread must be serviced by its deadline and service after the deadline has expired is the same as no service at all.
event latency	The amount of time that elapses from when an event occurs to when it is serviced.
Interrupt latency	The period of time from the arrival of an interrupt at the CPU to the start of the routine that services the interrupt.
dispatch latency	The amount of time the dispatcher takes to stop one thread and put another thread onto CPU.
conflict phase	During scheduling, the time the dispatcher spends moving a thread off of a CPU and releasing resources held but lower-priority threads that are needed by the higher-priority thread that is about to be put onto CPU.
periodic	A type of real-time process that repeatedly moves between two modes at fixed intervals- needing CPU time and not needing CPU time.
rate	A periodic real-time process has a scheduling rate of 1 / p (where p is the length of its running period).
admission-control	In real-time scheduling, the scheduler may not allow a process to start if its scheduling request as impossible - if it cannot guarantee that the task will be serviced by its deadline.
rate-monotonic	The rate-monotonic scheduling algorithm schedules periodic tasks using a static priority policy with preemption.
Earliest-Deadline-First (EDF)	A real-time scheduling algorithm in which the scheduler dynamically assigns priorities according to completion deadlines.
proportional share	Proportional share schedulers operate by allocating shares among all applications assuring each gets a specific portion of CPU time.

7 Operating-system examples

7.1 Overview

- This section describes the scheduling policies of Linux, Windows, and Solaris operating systems.
- The term "process scheduling" is used generally, referring to kernel threads (Solaris, Windows) or tasks (Linux).

7.2 Example: Linux scheduling

- Linux scheduling history:
  - Prior to Version 2.5: Traditional UNIX scheduling (poor SMP support, poor performance with many runnable processes).
  - Version 2.5: Introduced O(1) scheduler (constant time regardless of tasks), improved SMP support, processor affinity, and load balancing.
  - Version 2.6.23: **Completely Fair Scheduler** (CFS) became default.
- Linux scheduling is based on **scheduling classes**, each with a specific priority.
- Scheduler selects the highest-priority task from the highest-priority scheduling class.
- Standard Linux kernels implement two classes: default (CFS) and real-time.
- CFS scheduler:
  - Assigns a proportion of CPU time to each task based on its **nice value** (range -20 to +19; lower value = higher priority).
  - Uses a **targeted latency**: an interval during which every runnable task should run at least once.
  - Maintains **virtual run time** (`vruntime`) for each task, which decays based on priority (lower priority = higher decay rate).
  - Selects the task with the smallest `vruntime` to run next.
  - Higher-priority tasks can preempt lower-priority tasks.
  - Handles I/O-bound vs. CPU-bound tasks by giving I/O-bound tasks higher priority due to their lower `vruntime` (they run for shorter bursts).
  - Uses a red-black tree (balanced binary search tree) to store runnable tasks, keyed by `vruntime`. The leftmost node is the highest priority.
- Linux real-time scheduling:
  - Uses POSIX standard (`SCHED_FIFO` or `SCHED_RR`).
  - Real-time tasks run at higher priority than normal tasks.
  - Priority ranges: 0-99 for real-time (static), 100-139 for normal (based on nice values). Lower numeric value = higher priority.
- CFS load balancing:
  - Equalizes load among processing cores, is NUMA-aware, and minimizes thread migration.
  - Load of a thread: combination of priority and average CPU utilization.
  - Uses hierarchical **scheduling domains**: sets of CPU cores balanced against each other based on shared resources (e.g., L1, L2, L3 caches, **NUMA node**s).
  - Balances loads within domains, starting at the lowest level. Reluctant to migrate threads between NUMA nodes to avoid memory latency penalties.

7.3 Example: Windows scheduling

- Windows uses a priority-based, preemptive scheduling algorithm.
- The **dispatcher** handles scheduling.
- A thread runs until preempted by higher priority, terminates, time quantum ends, or calls a blocking system call.
- 32-level priority scheme:
  - Variable class: priorities 1-15.
  - **Real-time class**: priorities 16-31.
  - Priority 0: memory management thread.
- Dispatcher uses a queue for each priority; executes an **idle thread** if no ready thread is found.
- Windows API priority classes for processes: `IDLE_PRIORITY_CLASS`, `BELOW_NORMAL_PRIORITY_CLASS`, `NORMAL_PRIORITY_CLASS`, `ABOVE_NORMAL_PRIORITY_CLASS`, `HIGH_PRIORITY_CLASS`, `REALTIME_PRIORITY_CLASS`.
- Thread priority is based on its process's priority class and its relative priority within that class (`IDLE`, `LOWEST`, `BELOW_NORMAL`, `NORMAL`, `ABOVE_NORMAL`, `HIGHEST`, `TIME_CRITICAL`).
- Base priority: default is `NORMAL` relative priority for its class. Can be modified by `SetThreadPriority()`.
- Dynamic priority adjustments:
  - Variable-priority threads: priority lowered when quantum expires (never below base priority).
  - Priority boosted when released from a wait operation (amount depends on what it was waiting for).

  - Foreground process (currently selected window) receives increased scheduling quantum (e.g., 3x longer).
- **User-mode scheduling** (**UMS**): Allows applications to create and manage threads independently of the kernel (Windows 7+).
  - More efficient for many threads, no kernel intervention.
  - Predecessor: **fibers** (limited use due to shared thread environment block).
  - UMS provides each user-mode thread with its own thread context.
  - Not for direct programmer use; built upon by language libraries (e.g., Microsoft **Concurrency Runtime** (ConcRT) for C++).
- Multiprocessor scheduling in Windows:
  - Attempts to schedule threads on optimal cores, maintaining preferred/most recent processor.
  - Uses **SMT sets** (sets of logical processors on the same CPU core, e.g., hyper-threaded cores).
  - Assigns an **ideal processor** (preferred processor) to each thread, incrementing a seed value for new threads to distribute load across logical processors and SMT sets.

7.4 Example: Solaris scheduling

- Solaris uses priority-based thread scheduling with six classes: Time sharing (TS), Interactive (IA), Real time (RT), System (SYS), Fair share (FSS), Fixed priority (FP).
- Each class has different priorities and scheduling algorithms.
- Default class: Time sharing. Dynamically alters priorities and time slices using a multilevel feedback queue (inverse relationship: higher priority = smaller time slice).
- Interactive class: Same as time-sharing, but gives windowing applications higher priority.
- Dispatch table for TS/IA threads includes:
  - Priority: Class-dependent (higher number = higher priority).
  - Time quantum: Inverse relationship with priority.
  - Time quantum expired: New (lower) priority for CPU-intensive threads.
  - Return from sleep: Boosted priority for threads returning from I/O wait.
- Real-time class: Highest priority; real-time processes run before any other class, guaranteeing bounded response time. Few processes belong to this class.
- System class: For kernel threads (e.g., scheduler, paging daemon); static priorities, reserved for kernel use.
- Fixed-priority class (Solaris 9+): Same priority range as time-sharing, but priorities are not dynamically adjusted.
- Fair-share class (Solaris 9+): Uses CPU **shares** (entitlement to CPU resources) instead of priorities, allocated to a **project** (set of processes).
- Scheduler converts class-specific priorities to global priorities and selects the highest global priority thread.
- Selected thread runs until it blocks, uses its time slice, or is preempted.
- Multiple threads with same priority use a round-robin queue.
- Solaris traditionally used many-to-many model, switched to one-to-one model with Solaris 9.

Section glossary

Term	Definition
scheduling classes	Scheduling in the Linux system is based on scheduling classes - each class is assigned a specific priority.
nice value	Nice values range from -20 to +19, where a numerically lower nice value indicates a higher relative scheduling priority.
targeted latency	Targeted latency is an interval of time during which every runnable thread should run at least once.
virtual run time	A Linux scheduling aspect in which it records how long each task has run by maintaining the virtual run time of each task.
scheduling domain	A set of CPU cores that can be balanced against one another.
NUMA node	One or more cores (for example, cores that share a cache) that are grouped together as a scheduling entity for affinity or other uses.
real-time class	A scheduling class that segreates real-time threads from other threads to schedule them separate and provide them with their needed priority.
idle thread	In some operating systems, If no ready thread is found, the dispatcher will execute a special thread called the idle thread that runs on the CPU until the CPU is needed for some other activity.
User-Mode Scheduling (UMS)	A Microsoft Windows 7 feature that allows applications to create and manage threads independently of the kernel.
fibers	The Microsoft Windows predecessor to User-Mode Scheduling allowed several user-mode threads (fibers) to be mapped to a single kernel thread.
Concurrency Runtime (ConcRT)	A Microsoft Windows concurrent programming framework for C++ that is designed for task-based parallelism on multicore processors.
shares	A scheduling concept in which CPU shares instead of priorities are used to make scheduling decisions, providing an entitlement to CPU time for a process or a set of processes.
project	A Solaris scheduling concept in which processes are grouped into a project and the project is scheduled.

8 Algorithm evaluation

8.1 Overview

- Selecting a CPU-scheduling algorithm is challenging due to various algorithms and parameters.
- First step: Define criteria for selection (e.g., maximizing CPU utilization under response time constraints, maximizing throughput with proportional turnaround time).
- This section describes various evaluation methods.

8.2 Deterministic modeling

- **Analytic evaluation**: Uses an algorithm and system workload to produce a formula or number for performance evaluation.
- **Deterministic modeling**: A type of analytic evaluation that takes a predetermined workload and defines each algorithm's performance for that workload.
- Example: Comparing FCFS, SJF, and RR (quantum = 10ms) for a given workload of 5 processes.
  - FCFS average waiting time: 28 milliseconds.
  - SJF average waiting time: 13 milliseconds.
  - RR average waiting time: 23 milliseconds.
- Advantages: Simple, fast, provides exact numbers, useful for describing algorithms and identifying trends.
- Limitations: Requires exact input, results apply only to the specific workload, may not reflect real-world variability.

8.3 Queueing models

- Useful when processes vary daily, but distributions of CPU/I/O bursts and arrival times can be measured/estimated.
- System described as a network of servers (CPU, I/O) with queues.
- **Queueing-network analysis**: Area of study to compute utilization, average queue length, average wait time, etc., from arrival and service rates.
- **Little's formula**: $n = \lambda \times W$
  - n: average long-term queue length (excluding serviced process).
  - λ: average arrival rate for new processes.
  - W: average waiting time in the queue.
  - Valid for any scheduling algorithm and arrival distribution.
- Limitations: Limited classes of algorithms/distributions, complex mathematics, often relies on unrealistic independent assumptions, results may be approximations.

8.4 Simulations

- Provides more accurate evaluation than analytic methods.
- Involves programming a model of the computer system with software data structures representing components.
- Simulator uses a clock variable to modify system state and gather performance statistics.
- Data generation:
  - Random-number generator based on probability distributions (mathematical or empirical).
  - **Trace files**: Monitoring real system to record sequence of actual events, then using this sequence to drive the simulation.
- Advantages: Trace files provide excellent comparison of algorithms on identical real inputs, producing accurate results for those inputs.
- Limitations: Can be expensive (computer time, storage for trace files), complex to design, code, and debug.

8.5 Implementation

- The most accurate evaluation method: code the algorithm, integrate it into the OS, and test under real operating conditions.
- Costs: Coding, modifying OS, testing (often in virtual machines).
- **Regression testing**: Confirms changes haven't introduced new bugs or reintroduced old ones.
- Challenges:
  - Environment changes: New programs, changing problem types.
  - Scheduler performance can influence user behavior (e.g., breaking large processes into smaller ones if short processes are prioritized).
  - Human/program behavior can attempt to circumvent scheduling algorithms.
- Flexible scheduling algorithms can be altered by system managers or users (e.g., Solaris's `dispadmin` command, Java/POSIX/Win APIs).
- Downfall of API-based tuning: performance improvements are often not generalizable.

Section glossary

Term	Definition
analytic evaluation	A means of comparing scheduling algorithm effectiveness by analyzing an algorithm against a workload and assigning it a score.
deterministic modeling	One type of analytic evaluation - takes a particular predetermined workload and defines the performance of each algorithm for that workload.
queueing-network analysis	An area of computing study in which algorithms are analyzed for various aspects and effectiveness.
Little's formula	A scheduling equation ($n = \lambda \times W$) that is particularly useful because it is valid for any scheduling algorithm and arrival distribution.
trace files	A scheduling algorithm evaluation method in which thread details are captured on real systems and various scheduling algorithms analyzed to determine effectiveness.
Regression testing	Confirms that the changes haven't made anything worse, and haven't caused new bugs or caused old bugs to be recreated (for example because the algorithm being replaced solved some bug and changing it caused that bug to reoccur).

9 Summary

- CPU scheduling is the task of selecting a waiting process from the ready queue and allocating the CPU to it. The CPU is allocated to the selected process by the dispatcher.
- Scheduling algorithms may be either preemptive (where the CPU can be taken away from a process) or nonpreemptive (where a process must voluntarily relinquish control of the CPU). Almost all modern operating systems are preemptive.
- Scheduling algorithms can be evaluated according to the following five criteria: (1) CPU utilization, (2) throughput, (3) turnaround time, (4) waiting time, and (5) response time.
- First-come, first-served (FCFS) scheduling is the simplest scheduling algorithm, but it can cause short processes to wait for very long processes.
- Shortest-job-first (SJF) scheduling is provably optimal, providing the shortest average waiting time. Implementing SJF scheduling is difficult, however, because predicting the length of the next CPU burst is difficult.
- Round-robin (RR) scheduling allocates the CPU to each process for a time quantum. If the process does not relinquish the CPU before its time quantum expires, the process is preempted, and another process is scheduled to run for a time quantum.
- Priority scheduling assigns each process a priority, and the CPU is allocated to the process with the highest priority. Processes with the same priority can be scheduled in FCFS order or using RR scheduling.
- Multilevel queue scheduling partitions processes into several separate queues arranged by priority, and the scheduler executes the processes in the highest-priority queue. Different scheduling algorithms may be used in each queue.
- Multilevel feedback queues are similar to multilevel queues, except that a process may migrate between different queues.
- Multicore processors place one or more CPUs on the same physical chip, and each CPU may have more than one hardware thread. From the perspective of the operating system, each hardware thread appears to be a logical CPU.
- Load balancing on multicore systems equalizes loads between CPU cores, although migrating threads between cores to balance loads may invalidate cache contents and therefore may increase memory access times.
- Soft real-time scheduling gives priority to real-time tasks over non-real-time tasks. Hard real-time scheduling provides timing guarantees for real-time tasks.
- Rate-monotonic real-time scheduling schedules periodic tasks using a static priority policy with preemption.
- Earliest-deadline-first (EDF) scheduling assigns priorities according to deadline. The earlier the deadline, the higher the priority; the later the deadline, the lower the priority.
- Proportional share scheduling allocates shares among all applications. If an application is allocated N shares of time, it is ensured of having N/T of the total processor time.
- Linux uses the completely fair scheduler (CFS), which assigns a proportion of CPU processing time to each task. The proportion is based on the `virtual runtime (vruntime)` value associated with each task.
- Windows scheduling uses a preemptive, 32-level priority scheme to determine the order of thread scheduling.
- Solaris identifies six unique scheduling classes that are mapped to a global priority. CPU-intensive threads are generally assigned lower priorities (and longer time quantums), and I/O-bound threads are usually assigned higher priorities (with shorter time quantums.)
- Modeling and simulations can be used to evaluate a CPU scheduling algorithm.