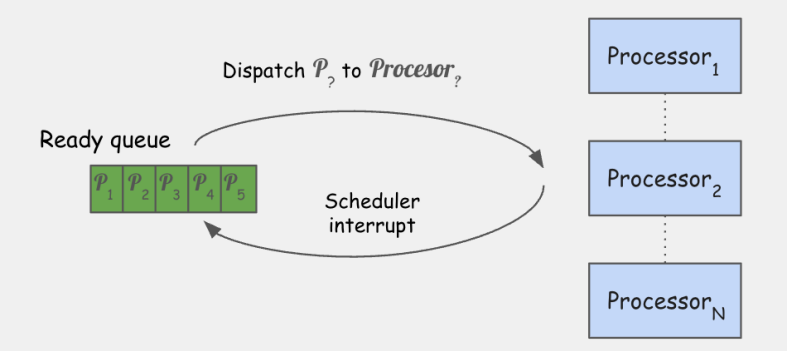
- Q1: What is scheduling?

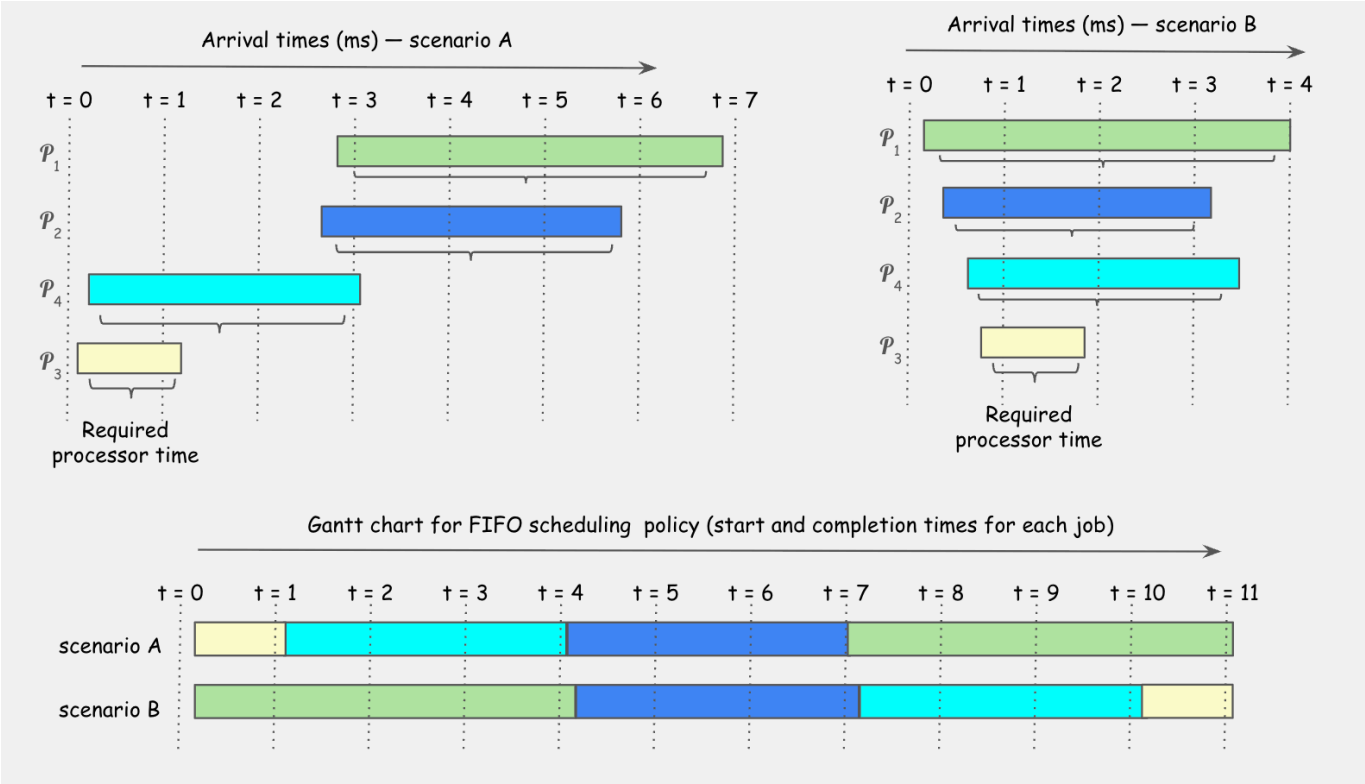
- Formally, the processor scheduling problem can be expressed as follows: "**Given k jobs ready to run in a system with N available processors, which job should be dispatched to which processor at any given point in time?"**
- The following diagram illustrates the scheduling problem, assuming, without loss of generality, that there are five jobs ready to run in a system, all mapped to one POSIX process each. (Recall that every process in the system is given the illusion that it has full exclusive access to all system resources, including all available processors, at any time.)



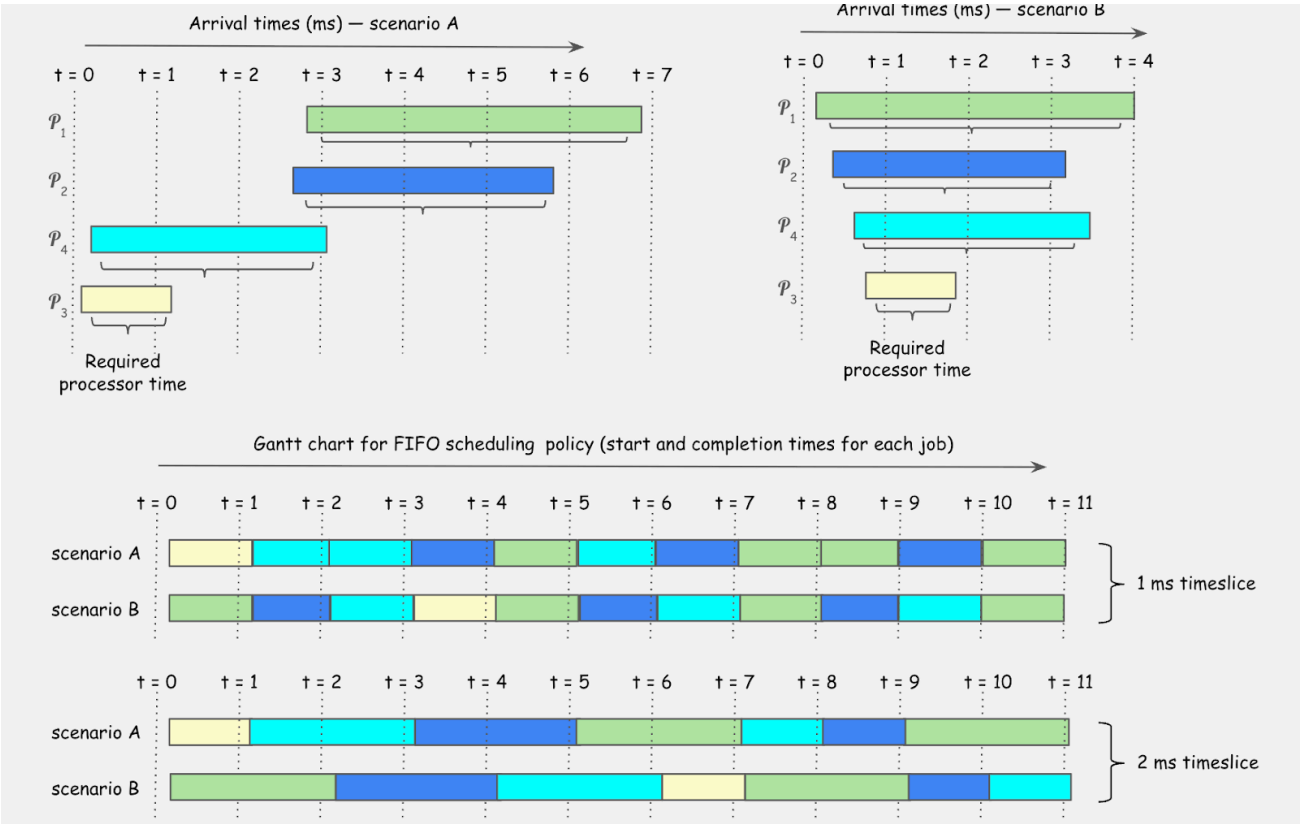
- Quantitative goals of scheduling
  - Minimize the average **completion time** of all jobs: $\text{Min}\{1/n \sum_{1}^{n} (T_i^{completion} - T_i^{arrival})\}$
  - Minimize the average **response time** of all jobs: $\text{Min}\{1/n \sum_{1}^{n} (T_i^{firstrun} - T_i^{arrival})\}$
  - Maximize **throughput**, jobs completed per unit of time: $\text{Max}\{ n / \text{Max}\{T_i^{completion}\}\}$

  Qualitative goals
  - Ensure fairness so that all processes receive a similar share of available processor time.
  - Impose an upper bound on the maximum response time a process may experience.
  - Uniformly distribute the load across all available processors, especially in cases of Symmetric Multi-Processor (SMP) architectures.
- Although the scheduling problem seems trivial to reduce to many well-known theoretical problems (e.g., the bin packing problem)—which have been studied for decades and have realistic heuristic-based solutions—in practice, its dynamic nature on real systems where jobs arrive in an a priori unknown manner, makes it difficult to derive a good generic solution.
- Most importantly, every solution must be designed and evaluated along with the relevant workloads of the target domain: for example, on interactive workloads (e.g., on a video call) scheduling must prioritize for small response; while, on the other hand, batched systems handling long-running jobs (e.g., a cluster of GPUs training Large Language Models) ought to, preferably, minimize average turnaround time in order to increase throughput.
- "And you have to realize that there are not very many things that have aged as well as the scheduler. Which is just another proof that scheduling is easy." –-Linus Torvald, 2001.
- Twenty-four years later: The Linux kernel began transitioning to the "Earliest Eligible Virtual Deadline First" (EEVDF) scheduler, first introduced in a scientific publication in 1995.

- **Q2: What are the default POSIX-compliant scheduling policies?**
  - The POSIX standard specifically requires four scheduling policies for an OS to be considered POSIX-compliant with respect to scheduling, including SCHED_FIFO (first-in first-out scheduling policy), SCHED_RR (round-robin scheduling policy), SCHED_SPORADIC (sporadic server scheduling policy), and SCHED_OTHER (another scheduling policy) defined here.
  - POSIX-compliant OS kernels should provide a sched_setscheduler interface to allow users to specify the desired scheduling policies for their tasks.
  - We will detail the two most relevant today—namely, **SCHED_FIFO** and **SCHED_RR**.

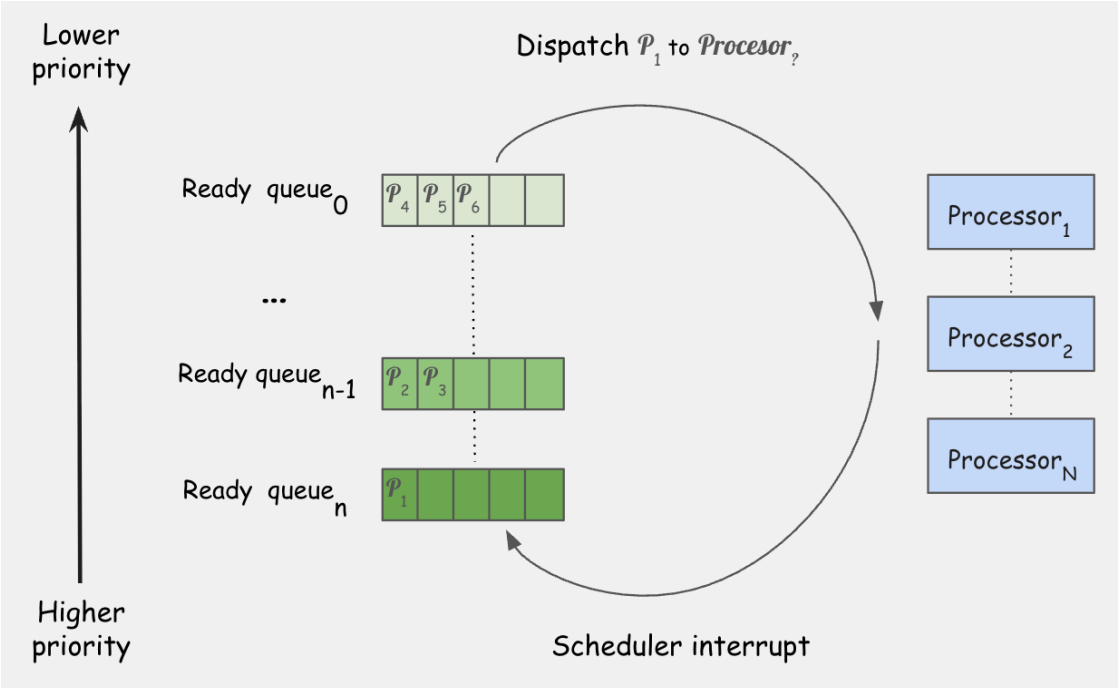


  - **SCHED_FIFO**: Each job gets assigned the processor in a First-In-First-Out (FIFO) order, based on its arrival time, and runs to completion (no preemption).
  - Not good for general workloads because both the average waiting time and the average response time for each job depend on its arrival time and the state that the system will be in.
    - **Avg. response time;** Scenario A: (0ms + 1ms + 1ms + 4ms) / 4 = 1.5ms [worst: 4ms]
    - **Avg. response time;** Scenario B: (0ms + 4ms + 7ms + 10ms) / 4 = 5.25ms [worst: 10ms]
    - **Avg. completion time;** Scenario A: (1ms + 4ms + 4ms + 8ms) / 4 = 4.25ms [worst: 8ms]
    - **Avg. completion time;** Scenario B: (4ms + 7ms + 10ms + 11ms) / 4 = 8ms [worst: 11ms]
  - The variance in the response time and the lack of any lower bound guarantees is the bigger issue of using FIFO by itself for scheduling real-time tasks because latency (i.e., lag) beyond a few microseconds is perceivable by the users and will make the systems appear as laggy / non-responsive / unpleasant to use.
  - Jitter tolerance below noticeable thresholds: ~1ms for audio, ~10ms for keyboard and mouse I/O, and ~20ms for video calls before humans notice lip-to-sound delay drift.



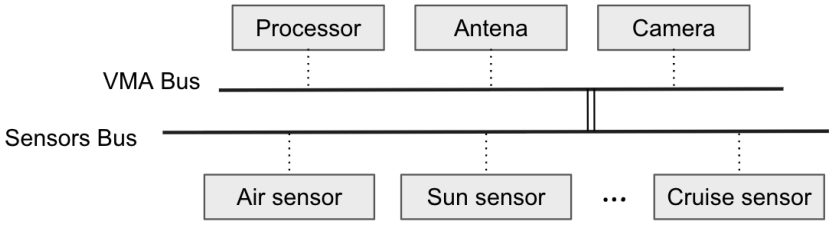
- **SCHED_RR**: Run each ready process for a fixed system-wide time interval—called time-slice, or also time quantum—and if the job is still running at the end of the time-slice, preempt it and pick another one from the ready queue in a FIFO manner.
- Practical approach to support time sharing and multitasking
    - **Avg. response time (1ms t-s):** Sc.-A: 0.5ms [worst: 1ms] ; Sc.-B: 1.5ms [worst: 3ms]
    - **Avg. response time (2ms t-s):** Sc.-A: 0.75ms [worst: 2ms]; Sc.-B: 3 ms [worst: 6ms]
    - **Avg. completion time (1ms t-s):** Sc.-A: 5.25ms [worst: 8ms]; Sc.-B: 5.5ms [worst: 11ms]
    - **Avg. completion time (2ms t-s):** Sc.-A: 5.5ms [worst: 8ms]; Sc.-B: 9.25ms [worst: 11ms]
- Better avg. response time compared to FIFO
    - Sc.-A: 0.5/0.75ms (RR) vs. 1.5ms (FIFO); Sc.-B: 1.5/3ms (RR) vs. 5.25ms (FIFO).
- Better worst-case response time compared to FIFO
    - Sc.-A: 1/2ms (RR) vs. 4ms (FIFO); Sc.-B: 3/6ms (RR) vs. 10ms (FIFO).
- Poor average completion time and worst-case completion time similar to FIFO
    - Sc.A: 5.25/5.5ms (RR) vs. 4.25ms (FIFO); Sc.-B: 5.5/9.25ms (RR) vs. 8ms (FIFO)
- The behavior of this scheduling policy depends on the length of the time-slice, and generally, a small time-slice can offer worst-case guarantees regarding response time; although, it may increase completion-time-related metrics.
- A more severe problem with SCHED_RR (not shown in any of the above graphs) is the cost of context switching: switching from the execution of one process or thread is non-trivia: especially if the switch is on a different process, the overhead is enormous not only because the OS kernel has to execute many instructions to do repetitive book-keeping work, but also because there are plenty hardware side-effects on the cache hierarchies, and most importantly, on the TLB.

- **Q3: What is priority-based scheduling?**
  - The problem with the scheduling algorithms discussed so far is that, because all jobs are treated with equal priority, their waiting and completion times depend on their arrival time as well as on the state of the system upon their arrival.
  - We need a scheduling policy such that users have the ability to explicitly prioritize some jobs over others, if they wish to do so.
  - To this end, we could design a solution that would be a hierarchical, priority-based scheduling policy where jobs are grouped into buckets of different priorities and the ones belonging to a high-precence bucket always take priority over those belonging to a lower-precedence bucket.
  - Furthermore, within each bucket, the OS kernel could give users fine-grained control to apply different scheduling policies. (For example, jobs of top-priority buckets could always be scheduled in a FIFO fashion, whereas jobs of medium-priority buckets could be scheduled in a round-robin manner.)

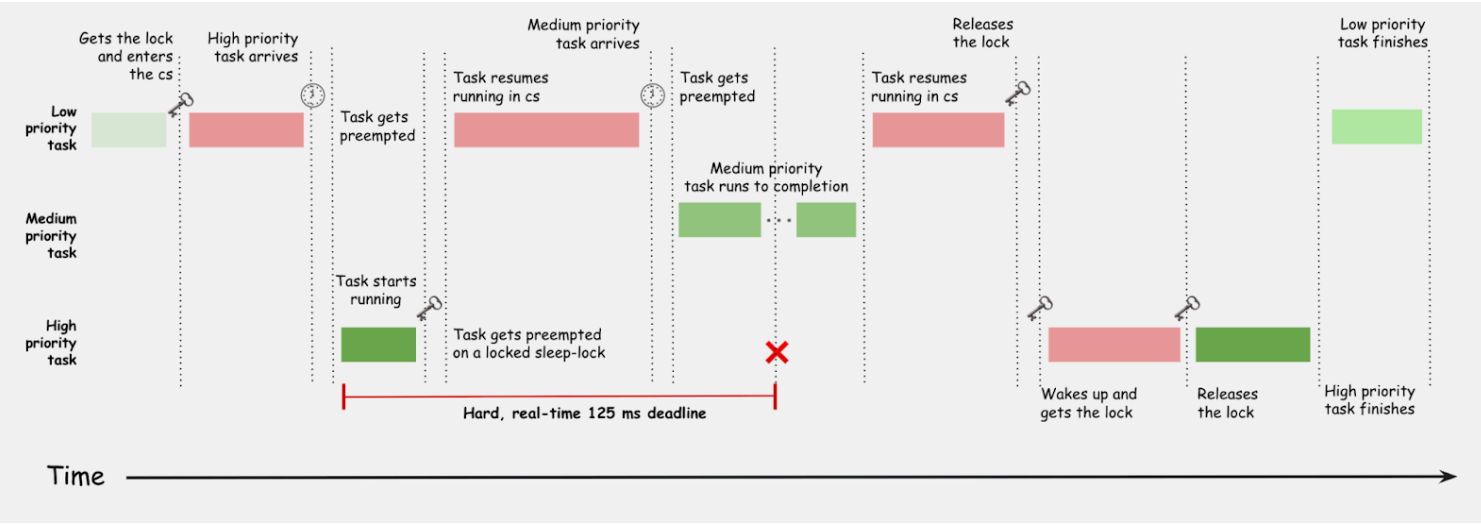


  - Since having the users group different jobs across the appropriate priority buckets and assign priorities is an overwhelming and error-prone task, by default, priority-based scheduling and the corresponding priority assignment are reserved for system tasks and users with special privileges and capabilities. (In contrast, user tasks of normal users are scheduled based on more generic scheduling policies, which do not depend on having the users select and optimize too many hyperparameters.)
  - Priority-based scheduling algorithms, when used along with locking and critical sections, introduce the possibility for a very subtle class of bugs, known as priority inversion bugs, which is one of the few bug classes known to have taken place on other planets!
  - Have you heard of the Mars Rover? NASA had to reproduce a priority inversion bug in a Rover's replica on Earth and remotely send a patch to its OS on Mars to fix it. (See this for more details.)

- The software of the Mars Rover uses the Wind River vxWorks Real-Time Operating System (RTOS), designed so that tasks can meet strict timing constraints with preemptive priority-based scheduling with a cyclic scheduler tick at 8 Hz (i.e., every 125ms).
- The hardware architecture of the Mars Pathfinder included two communication buses: the VME bus, connected to the processor, the antenna (for communication), and the camera; and the sensor bus, connected to the VME bus, and to the cruise and other sensors.
- Data from sensors had to be transmitted through the sensors' bus to the VMA bus, and eventually to the antenna to be transmitted to the Earth. On the other hand, the command signal from the processor had to move to the opposite direction, to the cruise and sensors part.



- The management of the sensors' bus was dependent on two critical tasks
  - The bc_sched task—the bus scheduler task, which decides who will transmit data next and transmits the schedule for the next cycle.
  - The bc_dist task—the bus distribution task, which decides who will receive data next.
- Furthermore, the comm_task was using the antenna to transmit data to Earth, and the asi_task was using the air sensor for scientific computations.
- The priority ordering of the above tasks was statically assigned as follows:
  prio[bc_sched] > prior[bc_dist] > prio[comm_task] > prio[asi]
- Most importantly, the bc_sched task used a watchdog to check at the beginning of its execution whether the bc_dist task had completed its execution in the previous cycle. This test was violated periodically, and it was causing a system-wide reset.
- **Why did this happen?** The antenna signal was better than expected, and the medium-priority antenna task had a lot of data to transmit. Consequently, due to the high load on the antenna, the medium-priority task preempted the low-priority task for a long period with the bus lock held, and thus, the high-priority task missed its deadline, leading to the bc_sched task resetting the system.



- Q4: What are the scheduling requirements for tasks of common workloads?

- **Real-time workloads:** Their tasks have strict timing constraints and must finish within specific deadlines, or otherwise the system fails.
  - Examples include airbag deployment software in cars (must respond within milliseconds), medical devices such as pacemakers, and robotics control systems.
  - The scheduling goal for such workloads is to meet strict deadlines reliably, every time.
  - Common scheduling algorithms include the Earliest Deadline First (EDF), which assigns dynamic priorities based on deadlines, and priority-based preemptive scheduling schemes, where higher-priority tasks immediately preempt lower-priority ones.
- **Latency-sensitive workloads:** Their tasks are interactive and require frequent human-computer interaction with quick system responses to keep the user experience smooth.
  - Examples include Graphical User Interfaces (GUIs), such as video games or text editors.
  - The scheduling goal for such workloads is to minimize the response time.
  - Common scheduling algorithms include round-robin with small quanta.
- **I/O-bound workloads:** Their tasks spend most of their time waiting for I/O (e.g., storage or network communication), do short processor bursts, and then block again waiting for I/O.
  - Examples include downloading a file, fetching data from disk while executing a database query, or streaming a video.
  - The scheduling goal for such workloads is to minimize the idle periods of the respective I/O devices by promptly allocating the processor for the brief time needed to initiate I/O requests (usually via DMA), ensuring continuous utilization of the respective I/O device.
  - Common scheduling algorithms include priority-based scheduling schemes, favoring I/O-bound tasks either with static priority assignment or dynamically via I/O boosting.
- **Processor-bound workloads:** Their tasks spend most of their time doing intensive computation, rarely yield voluntarily, and rarely need to perform I/O.
  - Examples include video rendering, scientific simulations, and training of ML models.
  - The scheduling goal for such workloads is to achieve a fair (balanced) processor use among all tasks, and to avoid starvation.
  - Common scheduling algorithms include round-robin with large quanta.

- Q5: How is multiprocessor scheduling performed?

- So far, we've been mostly concerned with how to assign jobs to processors and less about how to decide which processor to assign each job to.
- In an ideal world, using a **global ready queue** of all ready tasks along with a bitmap keeping track of available processors would be enough.
  - Conceptually easy solution with good resource utilization and balance.
  - Unfortunately, having a global queue is not a scalable solution: contention will increase as more processors are added, and most importantly, there will be poor cache and TLB locality, since tasks will often migrate from one processor to the other.
- An alternative design, which addresses the contention of having a single global ready queue, is to have one **ready queue per processor.**
  - This design also improves cache locality, assuming there is some care for processor affinity (i.e., try to schedule a task on the same processor that it ran on recently).
  - However, since there could be a load imbalance across processors over time, the OS kernel would periodically need to perform load balancing across processors.

- Hardware considerations for multiprocessor scheduling decisions
    - Non-Uniform Memory Access (NUMA): In multi-node systems, memory access times vary between CPUs, so the scheduler strives to keep a process near the CPU where its memory is allocated to minimize slow remote memory access.
    - Simultaneous Multi-Threading (SMT): Logical processors on the same physical core share execution units, so schedulers may prefer to fill separate physical cores first, before scheduling work on SMT siblings.

- Q6: How does the Linux kernel perform scheduling?

- The Linux scheduler is a hierarchical, policy-driven scheduler: there are five scheduler classes—which can be perceived as five different schedulers—each of which takes precedence over all lower-priority classes, and furthermore, each scheduler class implements multiple scheduling policies. (See the macro for_each_active_class used in the __pick_next_task of schedule core.c to iterate over the predefined precedence of all available scheduler classes.)
- By default, the Linux kernel is not fully preemptible, and the default preemption model is PREEMPT_VOLUNTARY, which allows kernel code to be preempted only at explicitly defined safe points. (However, Linux is configurable: its preemption behavior can be customized during compilation or at boot time if the kernel is built with CONFIG_PREEMPT_DYNAMIC).
- The following table summarizes all the scheduler classes implemented by the Linux kernel, along with their typical use cases as well as the policies that each scheduler class supports.
- As discussed earlier, the Linux kernel provides the sched_setscheduler interface, which allows users to specify the desired scheduling policies for their tasks.

Precedence Order	Scheduler class	Implemented policies	Usecase	POSIX compliance
1	stop_sched_class	Run Linux kernel-internal tasks	Only used internally by the kernel; preempts anything running in the local processor	No
2	dl_sched_class	SCHED_DEADLINE	Hard real-time tasks whose execution deadlines must be met	No
3	rt_sched_class	SCHED_FIFO, SCHED_RR	Soft real-time taks (e.g., audio daemon) with priorities [1–99]	Yes
4	cfs_sched_class, eevdf_sched_class	SCHED_NORMAL, SCHED_BATCH, SCHED_IDLE	User tasks with nice values	Partially Yes
5	idle_sched_class	Run the "idle" task	Runs only when the local processor is idle	No

- Ultimately, this hierarchy pushes to the users the responsibility of assigning hard and soft real-time tasks to the appropriate scheduling classes, and the OS kernel is only responsible for handling processor- and i/o-bound tasks as well as interactive tasks.

- Details on each scheduling class and policy
    - **stop_sched_class.** Used internally by the kernel for tasks that must absolutely not be preempted (e.g., stopping CPUs, context switching, migration).
    - **dl_sched_class [policy: SCHED_DEADLINE].** Used for hard real-time workloads.
        - Implements an Earliest Deadline First (EDF) scheduling algorithm combined with Constant Bandwidth Server (CBS) semantics, which guarantees tasks receive CPU time according to their specified runtime and deadlines.
        - Not part of the POSIX standard, and is, thus, accessed through a custom syscall (sched_setattr) rather than standard libc interfaces. (Requires root privileges due to the strict resource guarantees it provides.)
    - **rt_sched_class [policies: SCHED_RR/SCHED_FIFO].** Used for soft real-time workloads.
        - Implements a hierarchy of priority-based scheduling queues [with range 1–99], and within each queue, supports either a SCHED_RR policy with a 100ms default time-slice or a non-voluntarily preemptive SCHED_FIFO policy (task runs until it blocks, yields, or a higher-priority task preempts it).
        - An rt task can only be preempted either by a higher-priority rt task or by tasks from the dl_ and stop_sched_class.
        - POSIX-standard and accessible via sched_setscheduler with a parameter for the rt priority value. (Careful: Since the maximum non-perceivable latency jitter is ~20ms, flooding the systems with as many such tasks as the available processors will make it visibly laggy.)
    - **cfs_sched_class [policies: SCHED_NORMAL/SCHED_BATCH/SCHED_IDLE].** The Completely Fair Scheduler (CFS) scheduler class and the SCHED_NORMAL policy are the defaults for regular user tasks.
        - When a scheduler interrupt ticks (every 4 ms by default), the CFS scheduler picks the task with the smallest virtual runtime, or vruntime—a metric that tracks processor time used by each task weighted according to each task's nice value.
        - By picking the task with the lowest (weighted) vruntime, the scheduler ensures that all tasks in the system experience a fair (balanced) processor time allocation.
        - **SCHED_NORMAL**: Used for interactive workloads.
            - Uses wake-up boosting (heuristic): temporarily increases the priority (lowers the vruntime) of a waking task to help it run sooner by giving it a short-term priority boost, which, in the grand scheme of things, may make interactive tasks feel more responsive.
            - Uses wake-up preemption (heuristic): when a task wakes up, it may immediately preempt the currently running task.
            - Each task's vruntime is further weighted by the task's nice value.
        - **SCHED_BATCH**: Used for background jobs.
            - Wake-up boosting and wake-up preemption are disabled (waking tasks are queued at the end of the processor's run queue).
            - Each task's vruntime is still weighted by the task's nice value.
        - **SCHED_IDLE**: Used for extremely low-priority background daemons.
            - Uses a fixed IDLE_TASK_WEIGHT regardless of the task's nice value.
            - Wake-up boosting and wake-up preemption are disabled.
            - Runs only when no other task is runnable and may, thus, starve completely on busy systems.

- **cfs_eevdf_class [policies: SCHED_NORMAL/SCHED_BATCH/SCHED_IDLE].** In the most recent Linux kernels, the Earliest Eligible Virtual Deadline First (EEVDF) scheduler class is substituting the CFS scheduler.
    - Fairness is good for most workloads, but latency-sensitive tasks may be laggy.
    - The CFS scheduler has been enhanced, over the years, with various heuristics to boost the responsiveness of interactive workloads and to make their task appear less laggy (see "heuristics" above).
    - The paper "**Earliest Eligible Virtual Deadline First (EEVDF): A Flexible and Accurate Mechanism for Proportional Share Resource Allocation**," published in 1995 by Ion Stoica and Hussein Abdel-Wahab, proposed a weighted proportional-share algorithm for time-shared resources with near-ideal fairness and theoretical worst-case latency guarantees.
    - Similar to the CFS scheduler, an EEVDF scheduler aims to distribute processor time equally among all runnable tasks with the same priority by (1) assigning a vruntime to each task, and (2) creating a “lag” value that can be used to determine whether a task has received its fair share of processor time. (A task with a positive lag is owed processor time, while a task with a negative lag has exceeded its fair portion of processor time.)
    - The EEVDF scheduler picks tasks with lag >= 0 and calculates a virtual deadline for each, selecting the task with the earliest VD to execute next.
    - It is important to note that this allows latency-sensitive tasks with shorter time slices to be prioritize—which helps with their responsiveness—since every time the scheduler picks the tasks with the closest deadline in a virtual time space calculated for each task as its relative position compared to other tasks, plus how long it may take to finish if it runs.
- **idle_sched_class.** Used to run the idle task when the processor has no runnable tasks and needs to enter power-saving states.