Chapter 9

CPU scheduling

In general, scheduling is deciding how to allocate a single temporal resource among multiple clients, in what order and for how long. This is a highly complex subject, as well as being a problem much older and broader than computer science. Here we focus on CPU scheduling, but the OS also schedules other resources (e.g., disk and network IO).

CPU scheduling involves deciding which task to run next on a given CPU, how long to run it for, and why CPU a given task should run on. “Task” here is intentionally vague: it might be a process, thread, batch job, dispatcher, etc.

Definition 9.1 (Scheduling). Scheduling is the problem of deciding, at any point in time, which process or thread on every core (or hardware thread) in a system is currently executing.

Definition 9.2 (Dispatch). In contrast to scheduling, dispatching refers to the mechanism for (re)starting a particular process or thread running on a particular core.

Remarks:
- We distinguish between scheduling and dispatch here so we can focus on actual scheduling algorithms in this chapter, and ignore dispatch (which we dealt with earlier in chapters 4 and 5), but it’s a useful distinction in practice. You will see people use the term “scheduling a process” to mean actually running it, but we’ll try and avoid that usage.

To break it down, we start here with very simple scheduling problems and gradually complicate them; this happens to roughly coincide with the computing chronology as well.

9.1 Non-preemptive uniprocessor batch-oriented scheduling

Definition 9.3 (Uniprocessor scheduling). Uniprocessor scheduling is the problem of scheduling tasks on a single processor core (or, more precisely, hardware execution context).
9.1.2 Batch scheduling metrics

What makes a good scheduler? A scheduler is generally trying to optimize some metric relative to some workload.

Definition 9.13 (Batch scheduler throughput). The throughput of a batch scheduler is the number of jobs the scheduler completes per unit time.

Definition 9.14 (Overhead). The overhead of a scheduler is the proportion of CPU time spend running the scheduler itself, as opposed to a client job. Overhead consists of the context switch time (strictly speaking, the time to do two half-context-switches) plus the scheduling cost.

Remarks:
- For run-to-completion (non-preemptive) uniprocessor schedulers, throughput for a given workload really only the overhead. However, as things get more complex the throughput starts to depend on the algorithmic properties of the scheduler as well.
- There are plenty of more interesting scheduler metrics, even in the simplistic case we are considering here: mean (or median, or maximum, or ...) wait time, for example. Other examples are various definitions of “fairness”, or keeping to some kind of external policy for sharing resources between paying clients.
- Overhead can be a problem for jobs with short run times.
- You should have already spotted that we’re assuming the CPU runs at fixed speed. This is not true in reality for power reasons (and cache effects, etc.). In many batch cases, however, the most efficient way to use the CPU is to run it flat-out until there are no more runnable jobs.

Example 9.15. Suppose the scheduling cost plus context switch time is 1ms, and each job runs for 4ms. The overhead is therefore 1/(4 + 1) = 20%.

Algorithm 9.16 First-come-first-served (FCFS) scheduling

1. Assume each job \( P_i \) arrives at time \( t_i \)

When the scheduler is entered:
2. Dispatch the job \( P_j \) with the earliest arrival time \( t_j \)

Remarks:
- This pretty much the simplest scheduling algorithm possible, though it is used in some cases.
- It’s low-overhead, but even for simple cases its performance is unpredictable.

9.1. NON-PREEMPTIVE UNIPROCESSOR BATCH-ORIENTED SCHEDULING

Example 9.17. Three jobs a, b, c arrive in quick succession, with execution times of 2, 3, and 3 seconds respectively. Dispatching them in this order leads to a mean wait time of \( (24 + 27 + 30)/3 = 27s \). If the arrival order instead was \( c, b, a \), we would see a mean wait time of \( (3 + 6 + 24)/3 = 13s \).

Definition 9.18 (Convoicing). The convoy phenomenon occurs in FIFO schedulers (among others) when many short processes back up behind long-running processes, greatly inflating mean wait times.

Remarks:
- This is a well-known (and widely seen!) problem, famously first identified in databases with disk I/O.
- It is a simple form of self-synchronization, a wider class of generally undesirable effects in systems.
- Despite this, FIFO is still used, for example in memcached, Facebook’s front-tier cache.

Algorithm 9.19 Shortest-Job First (SJF) scheduling

1. Assume each job \( P_i \) has an execution time of \( t_i \) seconds

When the scheduler is entered:
2. Dispatch the job \( P_j \) with the shortest execution time \( t_j \)

Theorem 9.20. Shortest-job first is optimal in the sense that it minimizes the average (mean) waiting time for all jobs in the system, at least for the case when all jobs have the same release time.

Proof. By contradiction: consider a sequence of \( n \) jobs with execution times \( t_k \), for \( 0 < k \leq n \). Let \( w_k \) be the waiting time for job \( k \). Then \( w_k = w_{k-1} + t_k \), where \( w_0 = 0 \), and the average waiting time is \( W/n \) where \( W \) is the total waiting time:

\[
W = \sum_{k=1}^{n} w_k = \sum_{k=1}^{n} (n-k)t_k
\]

Suppose this sequence minimizes the mean wait time, but is not sorted by increasing execution time. Then \( \exists j: 0 < j < n \), such that \( t_j > t_{j+1} \).

Now consider the alternative sequence obtained by swapping the positions of jobs \( j \) and \( j + 1 \). The total wait time for this new sequence is:

\[
W' = \sum_{k=1}^{j} (n-k)t_k + (n-j)t_{j+1} + (n-j-1)t_j + \sum_{k=j+2}^{n} (n-k)t_k
\]

Subtracting:

\[
W - W' = (n-j)t_j + (n-j-1)t_{j+1} - (n-j-1)t_j = t_j - t_{j+1} > 0
\]

Consequently, \( W' < W \) and either the original sequence cannot have been optimal, or there was no \( j \) (i.e. the sequence was sorted).
Remarks:

• This algorithm is already more complex computationally: it required a sort. If execution times are discrete (e.g., integer minutes), this requires time linear in the number of jobs. This points to a fundamental tradeoff in scheduling complexity (which hopefully corresponds to efficiency of the resulting schedule!) and the scheduling overhead. A simpler, less theoretically efficient scheduler may be better than an optimal scheduler that takes too long to schedule.

• In the real world, jobs arrive at any time. SJF can only make a scheduling decision when a job terminates, which means that newly arrived jobs with short runtimes can experience long hold times because a long job is already running.

• SJF requires knowledge in advance of the job execution times. This is notoriously hard to estimate in practice, but can be finessed by having the clients guess it and penalize them if they get it wrong, or by preempting unexpected long jobs.

9.2 Uniprocessor preemptive batch scheduling

Definition 9.21 (Preemption). A scheduler which can interrupt a running job and switch to another one is preemptive.

Algorithm 9.22 SJF with preemption

When a new job enters the system or the running job terminates:
1. Preempt and suspend the currently running job (if there is one)
2. Dispatch (start or resume) the job \( P_j \) with the shortest execution time

Remarks:

• This requires the OS to have a mechanism for interrupting the current job, such as a programmable interrupt timer.

• SJF with preemption is still problematic: new, short jobs may preempt longer jobs already running, extending their wait time unacceptably.

• “Shortest remaining time next” is a variant of preemptive SJF which mitigates, but does not solve, this problem.

• Preemption means that jobs (and, below, processes) are dispatched and descheduled without warning. This is the norm in modern operating systems, and neatly handles other reasons why a job might stop running (page faults, device interrupts, blocking I/O operations, etc.).

• Despite this, there are cases where preventing preemption is a good thing: hard real-time scheduling, for example, or low-latency communication-intensive parallel jobs.

9.3 Uniprocessor interactive scheduling

Definition 9.23 (Interactive scheduling). In contrast to batch-oriented job scheduling, an interactive workload consists of long-running processes most of which are blocked waiting for an external event, such as user input.

Remarks:

• Interactive scheduling covers a wide range of workloads: almost anything that runs for a long time (such as online services, databases, media servers, user interfaces, etc.), and where the CPU demand is dynamic and unpredictable.

• In addition to cases where the OS preempts the running process, preemptive schedule also captures the case where the process is paused for other reasons: page faults, I/O requests, etc.

• There have always been interactive, non-preemptive systems. Often called “cooperative multitasking” systems, they have included Windows up to version 3.1, the Macintosh OS before version 7, and many embedded systems. Such systems require each process to explicitly give up the processor to the scheduler by performing an I/O request or executing a \( \text{yield}() \) system call every so often.

Definition 9.24 (Response time). The response time of an interactive program is the time taken to respond to a request for service.

Remarks:

• Response time is different from wait time: it refers to long-running processes which handle a sequence of external requests (possibly in addition to other computation). Examples include a game responding to user control, a word processor responding to typing, or Facebook responding to a request for a home page.

• Response time, or some statistical measure of it, is often the key metric of interest for interactive scheduling.
Algorithm 9.25 Round-robin (RR) scheduling

1. Let $R$ be a double-ended queue of runnable processes.
2. Let $q$ be the scheduling quantum (a fixed time period).

When the scheduler is entered:
3. Push the previously-running job on the tail of $R$.
4. Set an interval timer for an interrupt $q$ seconds in the future.
5. Dispatch the job at the head of $R$.

Remarks:
- RR is the simplest interactive scheduling algorithm—in the absence of I/O or other activity, it runs all runnable tasks for a fixed quantum in turn. It is the interactive counterpart to FIFO.
- RR is easy to implement, understand, and analyze.
- Unless you’re testing an OS, it’s rarely what you want. For one thing, it allocates all processes in the system the same share of CPU time. Moreover, if a process blocks, it implicitly donates the rest of its current time quantum to the rest of the system.
- The response time of a process highly unpredictable, since it essentially depends on where the process is in the run queue.
- RR has a fundamental tradeoff between response time and scheduling overhead, determined by the choice of quantum $q$. However, it is exacerbated that RR usually switches the running process more than is necessary for the system to make progress.

Example 9.26. Suppose we have 50 processes, the process switch time is 10µs, and the scheduling quantum is 100µs. This leads to a scheduling overhead of about 9%, but an average response time of $50 \times 110/2 = 2750µs$.

Alternatively, if we increase the quantum to 1000µs, the overhead is reduced to 0.99%, but average response time increases to $50 \times 1010/2 = 25250µs$ or 50ms.

9.3.1 Priority-based scheduling

Definition 9.27 (Priority). Priority-based scheduling is a broad class of scheduling algorithms in which each process is assigned a numeric priority, and the scheduler always dispatches the highest priority runnable task. A strict priority scheduling algorithm is one where these priorities do not change.

Remarks:
- Processes with the same priority can be scheduled using some other algorithm, such as RR.

Definition 9.28 (Starvation). Strict priority scheduling can lead to starvation: low-priority processes may be starved by high-priority ones which remain runnable and do not block. For this reason, strict priority systems are rare, and processes that run at high priority in such systems are carefully written to avoid hogging the processor. Instead most priority-based schedulers are not strict but dynamic: the priorities of tasks change over time in response to system event and application behavior.

Definition 9.29 (Process aging). Aging is one solution to starvation: Tasks which have waited a long time are gradually increased in priority. Eventually, any starving task ends up with the highest priority and runs. The original priorities periodically reset.

Definition 9.30 (Multi-level queues). In practice, priority-based schedulers are based on multi-level queues: there are a finite number priorities (e.g. 256), and each has a queue of processes at that priority. Priority levels are grouped into classes; queues in different classes are scheduled differently. For example, interactive queues are high priority and scheduled using round-robin, batch and background tasks are low-priority and scheduled FCFS, etc.

Definition 9.31 (Priority Inversion). Priority inversion occurs when a low-priority process blocks, a runnable medium-priority process then attempts to acquire the lock. If when $P_h$ blocks, a runnable medium-priority process $P_m$ gets to run, this inverts the effect of priority in the schedule.

Remarks:
- In the worst case, the medium-priority process can prevent the high-priority process from running for an arbitrarily long duration.
- Priority inversion is an old and well-studied problem, but it recurs with disturbing frequency as in the infamous case of the Mars Pathfinder rover.
- Classically, there are two approaches to dealing with priority inversion.

Definition 9.32 (Priority ceiling). In a system with priority inheritance, a process holding a lock temporarily acquires the priority of the highest-priority process waiting for the lock until it releases the lock.

Remarks:
- Priority ceiling incurs much less runtime overhead than priority inheritance, but potentially requires static analysis of the entire system to work. Its use is therefore restricted to embedded real-time systems.
A (rather conservative) approximation to priority ceiling is to disable interrupts during the lock hold time, but this is only applicable in limited situations.

Definition 9.34 (Hierarchical scheduling). A hierarchical scheduler is a further generalization of multi-level queues: queues are instead organized in a nested hierarchy or tree of scheduling domains. Within each domain (node in tree), sub-nodes are scheduled according to a potentially different policy.

Definition 9.35 (Multilevel Feedback Queues). A multilevel feedback queue scheduler is a class of multi-level queue which aims to deliver good response for interactive jobs plus good throughput for background tasks. The key idea is to penalize CPU-bound tasks in favor of I/O bound tasks. Processes which do not block but run continuously during a time interval have their priority reduced (a form of aging). I/O bound (including interactive) tasks tend to block, and therefore remain at high priority. CPU-bound tasks are eventually re-promoted.

Remarks:
- MLFQ schedulers are a very general class of algorithm. Almost any non-real-time scheduling algorithm can be approximated by multi-level feedback queues.

Example 9.36 (The Linux on(1) scheduler). This version of the Linux scheduler is a 140-level Multilevel Feedback Queue. Levels 0-99 (high priority) are static, fixed, “real-time” priorities scheduled with FCFS or RR. Levels 100-199 are user tasks scheduling using RR, with priority aging for interactive (I/O intensive) tasks.

This makes the complexity of scheduling independent of the number of tasks. The scheduler uses two arrays of queues: “runnable” and “waiting”; when no tasks remain in the runnable array, the two arrays are simply swapped.

Example 9.37 (The Linux “completely fair scheduler”). In the CFS as described in the documentation, a task’s priority is determined by how little progress it has made adjusted by jiffes per second. The task gets a “bonus” if it yields or blocks early (this time is distributed evenly). The implementation uses a Red-Black tree to maintain a sorted list of tasks, meaning that operations are now \(O(\log n)\), but still fast.

In fact, this is the very old idea of “fair queuing” from packet networks, also known as “generalized processor scheduling”. It ensures a guaranteed service rate for all processes, although CFS does not expose (or maintain) this guarantee.

Remarks:
- Stepping back a bit from the details, the schedulers we have seen in Unix conflate protection domains with resource principals: priorities and scheduling decisions are per-process (or threads). In practice, applications may span multiple processes, and at the same time share server processes with other applications. This means we may want to allocate resources across processes, or provide separate resource allocations within a single process – think of a web server, for instance.

9.4. REAL-TIME SCHEDULING

- Scheduling processes can also lead to unfairness between users: If I run more compiler jobs than you, I get more CPU time.
- The algorithms we have seen do not deal cleanly with in-kernel processing, for example interrupts or the overhead of scheduling itself.
- Some (though not all) of these issues are addressed by virtual machines or containers.

9.4 Real-time scheduling

Definition 9.38 (Hard real-time). An application is hard real-time if its correctness depends not only on the I/O actions it performs, but also the time it takes to execute. Hard real-time task correctness is often expressed in terms of deadlines: each task has a specific point in time by which it must have completed in order to be correct.

Remarks:
- Hard real-time systems include engine management units (EMUs) for cars, control systems for critical machinery, avionics, etc.
- In the general case, hard real-time scheduling is impossible: tasks can appear at any time, with any deadlines. Hard real-time systems in practice must impose constraints on the set of tasks they are prepared to schedule so that they can guarantee correctness (including every task meeting its deadline).
- In hard real-time systems, the execution time of each task is generally known in advance along with the deadline.
- Tasks in a hard real-time system can be periodic (they recur at regular intervals) or aperiodic.
- If the task set is not known in advance, the system must reject tasks for which no feasible schedule is possible, a process called admission control.
- Real-time does not mean fast! Both hard- and soft-real-time scheduling are about predictability, not performance. Hard-real-time systems in particular are often quite slow.

Definition 9.39 (Rate-monotonic scheduling). Rate-monotonic scheduling (RMS) schedules periodic tasks by always running the task with shortest period first. This is a static (offline) scheduling algorithm.

Suppose there are \(n\) (periodic) tasks, each task \(i\) has execution time \(C_i\) and period \(P_i\). Then RMS will find a feasible schedule if:

\[
\sum_{i=1}^{n} \frac{C_i}{P_i} \leq n(2^{1/n} - 1)
\]
Remarks:

- The proof [LL73] is beyond scope of this course, but worth reading.
- This condition puts a limit on the processor utilization (the left-hand side of the inequality) before deadlines get missed. As the number of tasks increases, this tends to:

\[
\lim_{n \to \infty} n(\sqrt{2} - 1) = \ln 2 \approx 0.693147 \ldots
\]

The implication here is that it’s hard to use more than 69% of the system under RM.

- RM is one of the two classic hard real-time schedulers: it is extremely efficient provided that tasks are periodic and the full workload is known in advance – this is the case in many embedded control applications.

So what should we do if we need to run online (that is, we don’t know the job mix in advance)?

Definition 9.40 (Earliest-deadline first). **Earliest deadline first (EDF)** scheduling sorts tasks by deadline and always runs the earliest deadline first. It is dynamic and online, and tasks are not necessarily periodic.

EDF is guaranteed to find a feasible schedule if:

\[
\sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1
\]

I.e. it can use 100% of a processor, if we ignore the overhead of scheduling and context switching.

Remarks:

- EDF is more complex (scheduling decisions can be \(O(\log n)\) in the number of tasks). It is typically implemented by maintaining a priority queue of jobs sorted by deadline, often represented as a heap.
- It is much more flexible, and admission control can be performed online.
- If utilization exceeds 100%, however, EDF is unstable: its behavior is unpredictable.

Definition 9.41 (Soft real-time scheduling). **In contrast to hard real-time systems, a soft real-time task has timing requirements which are non-strict, but nevertheless affect the quality and utility of the result.**

9.5 Multiprocessor scheduling

So far we’ve considered scheduling a single processor (and its workload) in isolation. As soon as we have more than one processor to manage, things get much more complicated. Fully general multiprocessor scheduling is NP-hard - it tends to reduce to 2-dimensional bin-packing. The two-dimensionality comes from having to decide which core to run a given thread on as well as when to dispatch it on that core.

In general, multiprocessor scheduling is beyond the scope of this course. We just present a simplified overview here, starting with some simplifying assumptions:

- The system can always preempt a task. This rules out some very small embedded systems or hard-real-time systems (and early PC and Macs, it turns out) but otherwise is reasonable.
- The scheduler is work-conserving.

Definition 9.43 (Work conserving). A scheduler is **work conserving** if no processor is ever idle when there is a runnable task.

9.5.1 Sequential programs on multiprocessors

Scheduling a collection of sequential programs on multiprocessors is relatively simple, although more complex than uniprocessor scheduling.

Definition 9.44 (Naive sequential multiprocessor scheduling). The simplest model for multiprocessor scheduling maintains a single system-wide run queue. Whenever an individual processor makes a scheduling decision, it picks a thread from the run queue to remove and dispatch.
Remarks:

- As described, this scheduler is work-conserving (modulo overhead).
- We haven’t said anything about the per-core scheduling algorithm used when a core looks at the run queue. It can be almost any uniprocessor scheme, but note that most of the guarantees vanish. For example, priority invariants might not be maintained across the whole system.
- Basic multi-queue models from queuing theory can be applied to analyze a system like this, but one must also take into account the overheads of locking and sharing the queue.
- In many situations, the single run queue becomes a significant bottleneck due to the need to globally lock it whenever any core needs to reschedule.
- Threads or tasks in this scheme also end up being allocated arbitrarily to cores, and so tend to move frequently between cores and, more critically, between different groups of cores sharing a cache. This dramatically reduces locality and hence performance.

Definition 9.45 (Affinity-based scheduling).

To remove the bottleneck of a single run queue and improve cache locality of running processes, affinity-based scheduling tries to keep jobs on one core as much as possible. Each core has its own run queue, and jobs are periodically re-balanced between all the individual queues.

Remarks:

- This is much more efficient, but note that it is not work conserving any more. A processor can end up with an empty run queue when other queues have jobs which are runnable, but not currently running.
- One way to mitigate this is for a processor which is idle to “steal” a job from a more heavily loaded processor.

Definition 9.46 (Work-stealing).

A work-stealing scheduler allows one core which would otherwise be idle to “steal” runnable jobs from neighboring cores so as keep doing useful work.

9.5.2 Parallel programs on multiprocessors

Things get much more complex when we consider jobs not as single threads, but as collections of parallel threads which coordinate among themselves. For example, global barriers in parallel applications present a significant challenge: one slow thread has huge effect on performance.

Multiple threads in many (but not all) applications would benefit from cache sharing, and different competing applications on the same machine can pollute each others caches. However, in other cases, it’s better to have each thread have its own cache and thereby maximize use of the cache across the whole machine.