Chapter 5: CPU scheduling

Andrés Ponce

November 8, 2020

> CPU scheduling refers to the process by which we decide which programs to run. One of the most important OS functions is scheduling programs, so allocating one of the most important system resources is a challenging task.

Idea

The CPU works in what are called **bursts**. Bursts are a short period of time where the CPU is executing a certain type of function. The CPU usually works with **CPU bursts**, followed by **I/O bursts**. Usually program execution alternates between the two.

When does scheduling take place? Depends on the OS. There are usually four conditions where that happens:

- When the state of the program changes from running state to waiting. [1]
- When the process goes from running state to ready state(e.g. when interrupt occurs).
- When the process goes from waiting state to ready state(e.g. I/O completion.)
- Process termination.

[1] Maybe b/c of I/O or a child process running.

When the first or the fourth condition occurs, we say the scheduling is **non-preemptive**, since essentially the OS waits for the program to either terminate or to be idle to perform a context switch. The other conditions cause what is called **preemptive** scheduling. This means the program can be in the middle of execution when it is rescheduled and another program is loaded.

This can result in **race conditions**, for example if the data is required by two programs, and gets changed during execution.

Thread Scheduling

We recall that the OS schedules kernel threads to run on the CPU, and the user multi-threading library creates the user-level threads which eventually get mapped to the kernel-level threads.

The OS usually selects the kernel thread with the highest priority to schedule on the physical CPU core.

How do we schedule programs to run on a CPU with multiple threads? We could have a common ready queue across all cores, which

physical cores have to access to schedule programs to run. This approach would maybe require some locking to occur every time we want to access the ready queue to avoid race conditions. This locking process would probably become a bottleneck whenever we need to access the queue.

Another idea is having a ready queue for each core, where the data is local. This approach is called **symmetirc multiprocessing**. CPUs using this approach are faster and consume less power, since they more efficiently use memory. One area where there still is lots of wasted time remains calling things from memory. Since the CPU runs much faster than memory, the CPU might spend a lot of valuable time waiting for items from memory to become available.

A remedy for this is **hardware threads**. The hardware threads are assigned to each core and so when there is a memory stall, another hardware thread might be scheduled to run during that time. Otherwise, up to half of the time might be spent waiting for info to be retrieved from memory. The result is that there might be twice as many logical CPUS available to the OS.

Thus a CPU might have two different levels of scheduling: one for deciding which core to schedule a thread on and inside the core there is a scheduler to decide how to schedule the instructions.

Load Balancing

Now that we have general approaches to scheduling on multi-threaded CPU cores, how do we schedule the instrutions in the most balanced way possible? If we don't do so, then we might be using the system resources in suboptimal ways. There is **push migration** and **pull migration**. The former involves a task checks the load on each processor and if there is an inbalance then *pushes* threads from some processors to less busy ones. The latter relies on idle cores looking for threads which it can bring over to itself to balance the load.

When a process has been running on a core, the cache registers have also already been populated with the process's data. This is called **processor affinity**, and if the cache already contains most of the information we need, we say it is **warm cache**. There are different policies of scheduling processes based on affinity. **Soft affinity** means that the scheduler might try to keep the process on the same thread, but it can be moved if necessary. The opposite, **hard affinity**, indicates to the scheduler that a thread only run on a subset of cores.

Real-Time CPU Scheduling

To refresh, a real-time system opertes with strict time-limit requirements. Processes have a certain time they need to be processed in. This changes our considerations for our OS design. **Soft real-time systems** provide no guarantee as to when a critical process will be scheduled, only that it will be given preference over non-critical processes. **Hard real-time systems** make processes be scheduled before their deadline, otherwise that process is no longer of use.

There are two types of latency: interrupt latency and dispatch latency. The former refers to how fast an interrupt can de handled once it occurs. The latter refers to how fast a scheduler can stop one process and start another.

Rate-Monotonic Scheduling is an algorithm that assigns static priorities to incoming processes. This also relies on some other terms: the **period** of a process refers to how often it needs to execute, and the **rate** is defined as $1/p$. The scheuduler will then pre-emptively schedule the processes with a higher priority. It can be shown that using this algorithm, CPU utilization will not be 100%. The higher the number of processes, the CPU utilizaiton might reach some sort of standstill.

Earliest-Deadline First scheduling involves the process announcing its time limit to the scheduler. Earlier deadline means higher priority. In this scheme we're not concerned with periodicity or rate, only that the process anounces its deadline to the scheuduler.

OS Examples

In the olden days, Linux used the traditional UNIX scheduler, but this did not take into account MSP, so it was revised. The Linux *Completely-Fair Scheduler* will assign a relative importance to each process, and schedule them in increasing numbers, meaning decreased importance. However, since a process can change its importance, it can allow different processes to run instead, which gives it the name nice.

Algorithm Evaluation

As we saw, there are different ways of evaluating, such as utilizaiton, thorughput, etc... The algorithm we choose depends on what our system requires.

Deterministic Modeling

Analytic evaluation uses a formula to determine the performance of an algorithm given a certain workload. To determine the different scheduling algorithms, we would need the burst time for each process, and our analysis would only work in that scenario.

Queueing models try to estimate the average waiting time and processes based on the distribution of CPU bursts. This is an exponential distribution. We can also use **Little's formula** to estimate the number of arriving processes while our CPU waits.

$$n = \lambda x W$$

where λ is the rate of arrival of a new process and W is the average waiting time. This says that the average queue length n is equal to the rate of arrival of new processes times the average waiting time for each process.

We can also run a simulation, where we use software and random number generators to set CPU bursts, incoming processes, wait times and so on to see the performance of a certain algorithm.