Lecture Notes for CS347: Operating Systems

Mythili Vutukuru, Department of Computer Science and Engineering, IIT Bombay

5. Process Scheduling

5.1 Basics of scheduling

- Process scheduling is used to timeshare a processor amongst multiple processes that are ready to run, enabling concurrent execution of several processes. It is one of the central mechanisms that enables multiprogramming and efficiency.

- When is a scheduler invoked? In a **non-preemptive** operating system, a scheduling decision occurs only when the currently running process blocks, or terminates. In a **preemptive** operating system, a scheduling decision can be made at other times as well, e.g., when a system call or interrupt handling completes, or when an event that causes a process to unblock occurs. The kernel may also choose to invoke the scheduler at any other point in the kernel-mode execution of a process. Note that all system calls or switches to kernel mode from user mode do not result in calls to the scheduler and a subsequent context switch. For example, if an interrupt occurs while a process is running, it is perfectly possible for the process to service the interrupt and go back to user mode. A transition to kernel mode is a necessary but not sufficient condition for a context switch.

- The classification of preemptive vs. non-preemptive can apply to the kernel as well. A kernel can choose to not preempt itself. That is, it can turn off interrupts while in kernel mode (while continuing to preempt userspace processes). However, such kernel designs tend not to be responsive to interrupts, and cannot handle real-time services. Therefore, most modern kernels are preemptible, i.e., their execution can be safely interrupted without compromising the integrity of shared data structures, *except* when the kernel explicitly turns off preemption (e.g., while holding a spinlock).

- When the scheduler is invoked, it must pick one of the several ready processes to run. A **scheduling policy** specifies how a process is picked. The scheduling policy also determines the data structure that is used to store the list of ready processes (or PCBs). Different scheduling policies have different goals. Some may want to support fast response time for interactive processes. Some may want to maximize the efficiency and throughput of the CPU and I/O devices. Some may want to maximize fairness across processes. Different operating systems have different goals, and may end up with different scheduling policies in their implementations. Some operating systems let users configure **priorities** for processes that can help influence the scheduling decision.

- Picking a process to run is not enough: the scheduler must actually perform the work required for the **context switch**. A context switch is typically done as follows, by a piece of code called the **dispatcher**. When a process (in kernel mode) invokes the scheduler, the context of the running process (its CPU registers, program counter, virtual memory information, and so on) is saved. Once the scheduler runs its algorithm to pick a process to execute, the context of the new process is loaded, so that the new process can resume execution where it left off. Note that a context switch happens from the kernel mode of one process to the kernel mode of another. A process in user mode must first enter kernel mode before a context switch. Similarly, a newly switched in process will eventually move back to user mode after the context switch. The dispatcher itself deals with switching between kernel modes of processes.

- Where is a process context saved during a context switch? The context of a process is typically saved on the kernel stack of the process, which is in turn accessible from the PCB by the scheduler.

- On which stack, and in which context, does the scheduler run? This answer depends on the implementation of the operating system. The scheduler could just be another piece of kernel code invoked by a process in kernel mode, so it could run on the kernel stack of the invoking process itself. Alternately, there could be a separate per-CPU scheduler thread, with its own context and stack. In this case, during a context switch, a switch first happens from process context to scheduler context, and then from scheduler context to the new process context. The scheduler stack saves the scheduler context (the CPU registers and other state of the scheduler) between its invocations.

- An undesirable interaction of locks and scheduling is a phenomenon called priority inversion. Consider three processes L, M, H with low, medium and high priorities of scheduling respectively. Suppose L holds a lock, but is frequently preempted by M. Now, H is waiting for the lock that L holds. So, in effect, the execution of H is hampered by the execution of M, which violates the fact that H has a higher priority than M. Several solutions exist to avoid this problem. For example, the OS scheduler could temporarily elevate the priority of L to that of H, considering the fact that L holds a lock H needs. Priority inversion also does not occur in systems with only two priority levels (because H could never be ready to run and preempt L when L is running with the lock, enabling L to run to completion quickly without preemption and release the lock).

5.2 Scheduling Policies

- The simplest scheduling policy is the **First-Come-First-Serve (FCFS)** policy. Every process that is ready to run is placed at the end of a ready queue, and the scheduler goes over each process and runs it until it finishes or gives up the CPU voluntarily (i.e., blocks). This policy usually leads to longer average waiting times, e.g., when a lot of short processes get stuck in the queue behind a long one.

- A preemptible version of FCFS is the **Round Robin (RR)** policy. Here, the scheduler runs every process for a certain **time quantum** or **slice**, and moves on to the next process in the queue once the slice is finished. The performance of the RR algorithm heavily depends on the size of the quantum. Very small slices lead to good response times, but waste a lot of CPU cycles on context switching.

- In the **Shortest Job First (SJF)** policy, the process with the smallest execution time in the ready queue is picked for execution. This policy is provably optimal, achieving the lowest average wait time. However, predicting the execution time of a process based on past sizes of its CPU bursts is somewhat inaccurate, hence this policy is somewhat impractical to implement. A preemptive version of this policy is called the **shortest remaining time first** policy. Under this policy, when a new process arrives with a shorter execution time than that of the currently running process, the running process is preempted in favor of the new process. A heap-like data structure is more suited to storing the list of ready processes with such policies.

- With **priority scheduling**, processes are assigned a numerical priority, and the highest priority processes are scheduled before the lower priority ones. SJF is a special case of priority scheduling, where the priority is inversely proportional to the run time. Priority scheduling can be preemptive or non-preemptive. Priority can be defined by the user, or can be internally arrived at by the kernel. For example, it makes sense to prioritize I/O-bound processes over CPU-bound processes, in order to fully utilize the I/O devices. A potential problem with this policy is the starvation of low priority processes, which can be fixed by increasing the priority of processes as they wait longer.

- In **multilevel queue scheduling**, processes are placed in multiple queues, each corresponding do a different class (e.g., foreground processes, background processes, interactive processes), each with a different priority. The scheduler then selects a class to schedule from, and a process from the class using any of the earlier policies. The multilevel queue structure can also be adaptive, i.e., processes can move across classes based on their behavior.

- The policies implemented in real operating systems are often complex, and are a mashup of many of the simple policies discussed above. Linux maintains processes in different scheduler classes with varying priority. Each scheduler class can have its own scheduling policy. The default class with normal priority uses the Completely Fair Scheduler (CFS) policy. CFS tries to give each process its *fair share* of the scheduler. That is, with N processes, each should get $1/N$-th of the processor time. In fact, CFS uses "nice" values set by users as weights, and runs a weighted fair sharing scheduler. That is, a process with weight w_i should get a fraction $\frac{w_i}{\sum w_i}$ of the total run time.

This idea is implemented by maintaining a *virtual run time* counter of each process in a *red-black tree*. Think of time as divided into rounds, where each round goes over all processes once, giving each process one slot (or a number proportional to their weight) in each round. The virtual run time can be thought of as the number of such rounds that a process has completed so far. The scheduler does not do a strict round-robin to realize this idea of rounds. Instead, when a process runs for a while and comes to the scheduler, the scheduler simply computes its virtual run time based on its actual run time. Since our goal is to make sure that all processes run equally in all rounds, the scheduling decision boils down to extracting the process with the lowest virtual run time (i.e., one that has fallen behind and has run for the least number of rounds), and schedule it. Thus, every time the scheduler is invoked, CFS simply picks the process with the smallest virtual run time. In a red-black tree, this operation is O(1). Inserting a new process into a red-black tree is, however, O(log N) in the number of runnable processes.

Now, how is the virtual run time calculated? The virtual run time is calculated from the actual run time, adjusting for process priority and the number of other processes in the system. For example, for the same actual run time, the virtual run time of a higher priority process will be lower. That is, it would appear that the process has run for fewer rounds, causing it to be scheduled often. Similarly, for the same actual run time, the virtual run time may appear higher when there are more runnable processes in the system and the fair share goes down. Linux preserves the virtual run time of processes even when they block, so that a process that has blocked for a long time can reclaim its fair share of the CPU when it becomes ready to run. Linux also enforces a maximum executable time for a process, based on how many other processes there are and how long a process would want to wait, and preempts the running process upon the expiry of the maximum time.

5.3 Scheduling in multiprocessor systems

- Computer systems with more than one processor are referred to as **symmetric multiprocessor (SMP)** systems. These processors can be separate CPUs assembled together, or separate cores on the same CPU. Further, a core can have multiple hardware threads of execution to fully utilize the CPU while waiting to fetch data from memory. Process scheduling must efficiently schedule processes or kernel threads across all these processing entities.

- When a process runs on different processors at different times, the benefits of cache locality might be lost. To avoid this performance penalty, most SMP systems try to schedule a process onto the same processor and avoid migrating it. This property is called **processor affinity**.

- All processes in an SMP system can share a common data structure of ready processes, or each can have a separate list and schedule independently. Load balancing across processors becomes easy with a common queue. Otherwise, processes may have to be migrated across processor queues to balance load. However, a common queue of ready processes increases contention because all schedulers have to lock to access the shared list of processes.

- All memory access is not equal in a multiprocessor system. Accessing memory that is closer to the processor leads to lower access times and hence better performance. This property is called Non Uniform Memory Access (NUMA). NUMA-aware schedulers take this aspect into account when scheduling processes onto processors.