Lecture Notes - Scheduling Policy

Debadatta Mishra
Indian Institute of Technology Kanpur

Scheduling policy

- First come first served (FCFS) is one of the simplest scheduling policies where the process arriving first in the ready queue is run to completion. While FIFO policy does not make any assumptions regarding the CPU burst of processes, FIFO policy suffers from convoy effect where a process with long CPU burst delays all the other processes with some of the processes requiring very small CPU bursts. Therefore, FIFO policy is not suitable for interactive applications. Because of the inherent non-preemptive nature of the FIFO policy, application response time suffers along with other performance metrics like average turnaround time and average waiting time.
- Non-preemptive shortest job first (SJF) scheduling policy selects the shortest job in the ready queue. Once the shortest job is scheduled, any new processes arriving with a shorter burst wait until the time the scheduled process finishes its execution due to termination or moves to waiting state. SJF is optimal on average waiting time and average turnaround time. A preemptive version of SJF, shortest remaining time first (SRTF) can schedule a newly arriving process with a shorter CPU burst time after descheduling the currently executing process. More preemption points (e.g., timer interrupt) can be introduced to allow the scheduler to schedule a process with shortest remaining time at the time of timer interrupt. SRTF is suitable for interactive applications as it results in low response time. One issue with these scheduling is the requirement of advance knowledge regarding the CPU burst times of all the processes.
- One way to approximate the SRTF without the knowledge of process burst duration is to employ a round robin (RR) scheduling scheme. In RR, every process is scheduled for a fixed time quantum before it is moved to the end of a FIFO queue and the next process in the ready queue is scheduled. Also, if a process relinquishes the CPU before its time quantum, the next process in the queue is scheduled. A newly arriving process is added to the tail of the ready queue. One design choice in RR scheduling is the length of the time quantum. A large value for time quantum may make the RR scheduling equivalent to FIFO scheduling where all processes finish before their time quantum expires. A very small time quantum may result in a lot of context switches which in turn results in performance overheads and CPU wastage. Another issue with RR scheduling is that there is no scope to incorporate user inputs like priority into the scheduling scheme.
- Priority scheduling implements process priority in the CPU scheduling scheme. The priority value of a process can either be assigned by the user (static priority) or dynamically decided by the OS based on the program behavior or can be a function of both static priority and the dynamic priority. Like SJF, priority scheduling can be both preemptive and non-preemptive. In the preemptive priority scheme, when a new job with higher priority than the currently executing process arrives in the ready

queue, the new process is scheduled by switching out the currently executing process. While priority scheduling does not make any assumptions regarding the CPU burst time, it suffers from the problem of starvation. If a lot of high priority processes dominate the CPU, low priority processes can be starved of CPU for a long duration. One technique to handle the issue of starvation is to consider aging (age of a process in the ready queue) along with the priority to derive dynamic priority for processes.

- An extension of priority scheduling is to maintain different queues for processes with different priorities. While selecting the next process, the scheduler picks a process from the highest priority non-empty process queue. Different priority queues may implement different scheduling policies at individual queue level. This scheme also suffers from the starvation problem in a case when high priority processes dominate the system.
- A variation of multiple queues to address starvation issues is to have a feedback channel between different queues to migrate processes from one queue to another, a.k.a. Multilevel feedback queue (MLFQ). Dynamic priority calculation is an important aspect of MLFQ as that determines how a process makes its transition across different queues. One example scheme can be to implement a priority demotion mechanism to reduce priority of processes consuming their full time quantum in a RR scheduling scheme. Consider the case where a process in the highest priority queue scheduled with a time quantum 5ms exhausts its time quantum. The scheduler moves the process to the next lower level priority queue. MLFQ favors interactive and I/O bound applications as these processes have short CPU bursts (less than the time quantum), and therefore remain in highest priority queue during their lifetimes. CPU bound applications, on the other hand, gets demoted as they consume their allocated time quantum. There are certain issues with MLFQ as listed below,
    - If applications start as being CPU bound, they get demoted to a lower priority and remain in low priority even if the applications change their behavior (to be I/O bound). This is not uncommon in practical application scenarios. This issue can be addressed by employing *periodic priority boost* technique where all processes in the ready queue are moved to the highest priority queue.
    - The scheduler can be tricked by malicious users by performing a carefully orchestrated blocking operation just before their time quantum expires. For example, a process can sleep for some microseconds (using APIs like usleep) just before their time quantum expires. A possible solution to this problem is to keep an account of consumed CPU time instead of a binary indication regarding the CPU quantum usage. If a process across all its CPU bursts consume the time quantum, it is moved to a lower priority queue.

Scheduling in Linux

- Linux OS implements a variation of multi-level feedback scheduling mechanism to meet the scheduling requirements of real time, interactive and batch applications. Linux provides two broad scheduling classes,
    - For real time applications, SCHED_FIFO and SCHED_RR can be used in which the priority value can be between 0 (highest) and 99 (lowest). The

    difference between the two classes is due to the queue level scheduling algorithms used. For processes in SCHED_FIFO class, FIFO scheduling policy is employed while round robin scheme is applied for processes in SCHED_RR class. The policy is applied at individual priority level. Please refer the man page of **sched_setscheduler** for more details.
    - There are three other scheduling classes---SCHED_OTHER, SCHED_BATCH and SCHED_IDLE, for normal applications. Linux allows users to specify static priority for different processes through **nice( )** system call for normal applications. The Linux kernel derives dynamic effective priority based on the static priority and the program behavior.
- Linux kernel provides two special scheduling queues---STOP class for system shutdown which supersedes all tasks in the scheduling queue and, IDLE class to have the lowest priority task (swapper) in system scheduled when no other process is available.
- Every process in the NORMAL scheduling class starts with static priority value 120 which can be adjusted to a value between 100 and 139 using nice( ) system call. Dynamic priority of a process is calculated by the Linux kernel which is a function of static priority and sleep time. If a process spends a lot of time in waiting state, the Linux kernel boosts the priority of the process and vice versa.
- Some time back (in Linux 2.6), kernel developers proposed an O(1) scheduler where two sets of lists (active list and expired list) are maintained for 40 priority levels (100 to 139). The scheduler picks the task from the highest priority non-empty queue from the active lists, schedules it for a given time quantum. After expiry of the time quantum, the Linux kernel recalculates the priority and time quantum, and inserts the process in the expired list. When all the queues in active list became empty, the expired list and the active list are swapped.