Operating Systems – sprint 2022
Tutorial-Assignment 3

Instructor: Hans P. Reiser

Question 3.1: Scheduling Basics

a. What is the purpose of scheduling?

Solution:
The purpose of scheduling is to find a mapping of processes to resources, so that each process will eventually get the resources it needs. Normally, scheduling tries to maximize some quality metrics, such as the resource utilization. In this assignment, we will focus on CPU scheduling—that is, mapping of processes/threads to CPUs.

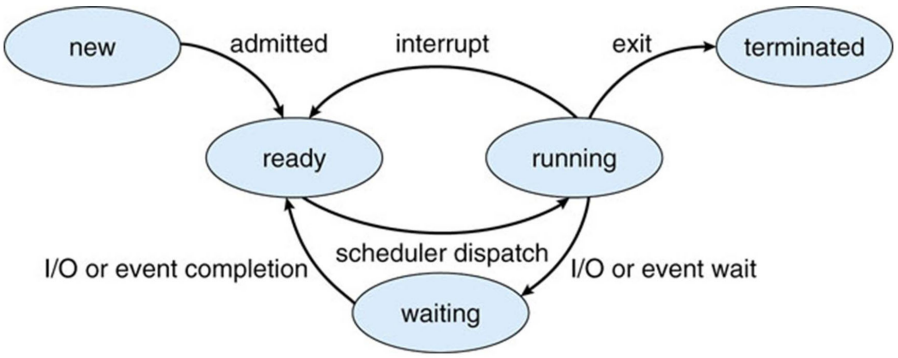
b. What is the difference between a long and short-term scheduler?

Solution:
The long-term scheduler is responsible for selecting the processes that are brought into the ready queue. The long-term scheduler is invoked very infrequently (seconds/minutes) and decides which jobs to load into memory and prepare for execution. Long-term schedulers usually can be found on job-processing systems such as the IBM mainframe, but not on regular desktop installations.

The short-term scheduler selects the next process that is allowed to run on the CPU from the ready queue. It is invoked very frequently (multiple times per second) and must be optimized for quick decisions. An O(1) scheduler is thus preferable and standard in modern operating systems.

c. Consider an operating system that supports the five task states "new", "running", "ready", "waiting", and "terminated". Depict the possible state transitions and the events that cause them.

Solution:



d. What quantitative metrics/criteria can be used to estimate the quality of a scheduling policy?

Solution:

Processor utilization: *Percentage of time that a processor is not idle.*

Throughput: *Number of processes/threads completed per unit of time.*

Turnaround time: *Time from submission of a process/thread to its completion — includes the time spent waiting in various queues and the time actually running on a CPU.*

Waiting time: *Time a process/thread spends in the ready queue.*

Response time: *Time from the submission of a request until the first response is produced. (e.g., key press to the begin of the console echo or submission of a job to the first time it is dispatched)*

e. What kind of hardware support is required for an operating system that implements a non-cooperative scheduling policy?

Solution:
For non-cooperative scheduling, the operating system must be able to interrupt a running thread (most probably executing in user mode). As the operating system cannot regain control by itself, hardware support in the form of interrupts is needed. Theoretically, any external interrupt (caused by some device) is suitable, as interrupts always return control to the operating system. In order to ensure interrupts (and thus OS invocations) at a constant rate, one normally employs a special timer device that can be programmed to generate interrupts periodically. Whenever such an interrupt "fires", the OS regains control and can select another thread to run next.

f. Discuss pros and cons of choosing a short timeslice length vs. choosing a longer timeslice length. What are common values for the length of a timeslice?

Solution:
The longer a timeslice is, the fewer context switches per time unit are required. As context switches require a certain overhead (entering the kernel, selecting a new process to run, switching address spaces if necessary, . . .), reducing the number of context switches can help to improve the throughput. However, making timeslices too long will negatively affect the responsiveness of the system: Each process will run for a relatively long time until it will be preempted, and consequently ready processes have to wait longer until they are allowed to run again.

Common timeslice lengths range from about 2 ms to 200 ms.

Configuring Windows to prefer foreground processes for example gives a timeslice length of 20 ms. This is the default on client versions. The server version is configured to prefer background processes by using a timeslice of 120 ms – 180 ms.

The Linux completely fair scheduler *uses dynamic timeslice lengths derived from the number of ready processes, the priority (nice value) of the process to be scheduled, and adjustable parameters, in a complex way that tries to balance responsiveness with overhead.*

Question 3.2: Scheduling Policies

a. How does SJF scheduling work? What is the difference between preemptive and non-preemptive SJF?

Solution:
SJF always selects the job with the shortest remaining time next. Instead of considering the total time to completion–which is hard to determine for many applications–, the (estimated) length of the next CPU-burst can be considered.

A preemptive SJF scheduler will make a new scheduling decision periodically and when a new process arrives in the ready queue: If the length of the next CPU burst of the new process is shorter than the remaining length of the current process, the new one is scheduled. With a non-preemptive SJF scheduler, a job always runs until its CPU burst ends.

b. What is the basic idea of priority scheduling?

Solution:
With priority-based scheduling, each process is assigned a priority, and out of all ready processes, the one with the highest priority is chosen. A second scheduling policy is needed if there are multiple ready processes with the same priority; these processes can for example be scheduled using round robin scheduling.

c. What is the major problem of priority-based algorithms? What is a possible solution?

Solution:
*The main problem of priority-based scheduling algorithms is **starvation**: A process won't be able to run as long as there is at least one process with a higher priority.*

To weaken the effects of starvation, an aging policy can be employed by which the priority of processes is increased that have not run for a certain amount of time. This technique is implemented in the Windows thread scheduler.

d. Explain the multilevel feedback queue scheduling algorithm. What kind of processes can be found in the high-level queues, what kind of processes sink to the lower-level queues?

Solution:
As the name suggests, multilevel feedback queue scheduling uses multiple scheduling queues with different characteristics. Processes will be moved between queues, depending on their runtime behavior. Processes are always taken from the highest non-empty queue. Processes in a higher queue get shorter timeslices than ones in a lower queue. If a process fully uses its timeslice it descends to the next-lower queue. As a consequence, only I/O intensive processes will remain in the highest queue. As I/O-intensive processes have short CPU-bursts, it is preferable to schedule them first, supplying jobs for the I/O devices, which can work asynchronously. CPU-bound processes will descend to lower-level queues getting longer timeslices.

e. Describe the idea of lottery scheduling.

Solution:
With lottery scheduling, each process is assigned a specific number of lottery tickets. When the scheduler is invoked, it randomly chooses one of the tickets, and the process that owns it is allowed to run.

f. Enumerate possible advantages of lottery scheduling over "classical" priority-based scheduling algorithms.

Solution:

No starvation *If one assumes that each process is granted at least one ticket and that the scheduler picks numbers uniformly distributed, each process will be allowed to run sooner or later. This is not ensured with (static) priorities, where low-priority processes will never run if there is always at least one higher-priority process runnable.*

CPU time *Lottery scheduling can also be used to grant each process a specific amount of CPU time, simply by assigning an according number of tickets: If the scheduler has a total of 100 tickets and a process shall get* 20% *of the CPU time, it is assigned 20 tickets.*

Hierarchical scheduling *Lottery scheduling allows a hierarchical distribution of CPU time. For example, each user can be assigned a certain amount of tickets, and the user can distribute these tickets among his applications.*

Ticket donation *Another interesting feature of lottery scheduling is that a process could be allowed to lend its tickets to another. This is especially useful in a client/server scenario. Whenever a client sends a request to a server, it lends its tickets to the server so that the server will run when the client normally would. Consequently, the client (and not the server) is accounted for the CPU time the server requires to process the client's request.*

Question 3.3: Link list implementation in C

a. Implement a linked list in C, using the following data structures:

```
struct ListItem {
    struct ListItem *next;
    int value;
}

struct ListItem *listHead = NULL;

void appendItem(int value) {
    // ... implement this
    // append to the end of the list
}

int removeListItem() {
    // implement this
    // removes the first item from the list and returns its value
    // returns -1 if list is empty
}

int containsItem(int value) {
    // implement this
    // return true (1) if list contains value, false (0) if not
}

int isEmpty() {
    // implement this
```

```
    // return true (1)  if  list  is  empty, false (0) otherweise
}
```

You should implement the empty functions in the template.