Operating Systems – spring 2023
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.

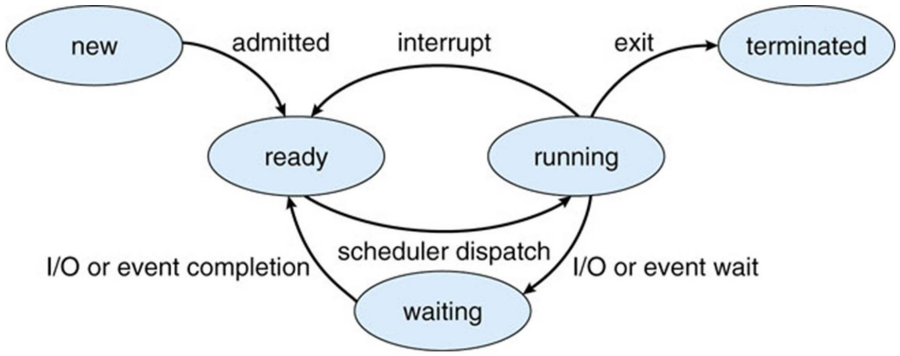
b. What is the difference between a long and short-term scheduler?

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: Link list implementation in C

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

```
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>

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

struct ListItem *listHead = NULL;

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

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

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

int isEmpty() {
    // implement this
    // return true (1) if list is empty, false (0) otherweise
    return 0;
}

int main() {
    appendItem(42);
    appendItem(4711);
    removeFirstItem();
    appendItem(42);
    appendItem(4);
    for(int i=0; i<5; i++) printf("%d\n", removeFirstItem());
}
```

You should implement the empty functions in the template.

Solution:

```
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
```

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

struct ListItem *listHead = NULL;

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

    // Allocate memory for the new element
    struct ListItem *item = (struct ListItem *)malloc(sizeof(struct ListItem));
    // Missing here: error handling
    // Note that the assignment 3 specifies a specific way errors shall be handled

    // Set the content of the new item to value
    item->value = value;

    // We append at the end, so the next pointer of the new element shall be NULL
    item->next = NULL;

    // Special case: if the list is empty, we just set listHead to our new element,
    // making it the first and only element in the list
    if (listHead == NULL) {
        listHead = item;
        return;
    }

    // Otherwise, we have to find the end of the list
    // We go through the list until we find the last element (the one with next==NULL).
    struct ListItem *lastel = listHead;
    while (lastel->next)
        lastel = lastel->next;

    // and then we append our new element to the last
    lastel->next = item;
}

int
removeFirstItem()
{
    //implement this
    // removes the first list item from the list and returns its value; returns - 1 if list is empty
    if (listHead == NULL)
        return -1;

    // We remove the first element by updating the listHead to point to the element after the current
    // listHead. We also remember the removed element (the "old" listhead) in tbr ("to be removed")
    struct ListItem *tbr = listHead;
    listHead = listHead->next;
```

```
    // We get the value from the list head first, as after the free, the value is no longer available
    int val = tbr->value;
    free(tbr);
    return val;
}

int
containsItem(int value)
{
    //implement this
    // return true(1) if list contains value, false(0)

    // we iterate over the list, starting with the first element
    struct ListItem *listElement = listHead;
    int isContained = 0;

    // while we have not yet reached the end
    while(listElement) {
        if(listElement->value == value)
            isContained = 1;
        // advance to the next element
        listElement = listElement->next;
    }
    return isContained;
}

int isEmpty() {
    // Shorthand notation in C for a comparison plus assignment
    // if the condition is true, the result is the value after the "?" / before the ":"
    // if it is false, the result is the value after the ":"
    return listHead == NULL ? 1 : 0;

    // equivalent to:
    // if (listHead==NULL) { return 1; } else { return 0; }

    // equivalent to
    // (because C uses 1 as "true" and 0 as "false")
    return (listHead==NULL);
}

int main() {
    appendItem(42);
    appendItem(4711);
    removeFirstItem();
    appendItem(42);
    appendItem(4);
    for(int i=0; i<5; i++) printf("%d\n", removeFirstItem());
}
```