



Eidgenössische Technische Hochschule Zürich  
Swiss Federal Institute of Technology Zurich  
Spring Term 2014

# Operating Systems and Networks

## Assignment 2

Assigned on: **27th February 2014**

Due by: **6th March 2014**

### 1 Scheduling

*The following table describes tasks to be scheduled. The table contains the entry times of the tasks, their duration/execution times and their deadlines. All time values are given in ms.*

Task Number	Entry	Execution Time	Deadline
1	0	30	70
2	0	20	90
3	20	20	50
4	30	10	40
5	50	30	120

*Scheduling decisions are performed every 10ms. You can assume that scheduling decisions take about no time. The deadline values are absolute.*

#### 1.1 Creating schedules

*In the following you are asked to create different types of schedules. Please visualize your schedule (like in the lecture) and also answer the questions below.*

*Types of schedules:*

- a) RR (Round Robin)*
- b) EDF (Earliest Deadline first)*
- c) SRTF (Shortest Remaining Time First)*

*Please answer the following questions for each of the schedules:*

- a) How big is the wait time per task?*
- b) How big is the average wait time?*
- c) How big is the turnaround time per task?*

- d) How is the response time computed for this scheduler? If possible, calculate the response time per task.

**Answer:** Definitions of the terms “turnaround time”, “waiting time” and “response time” according to the Silberschatz book:

- The Turnaround time is the time between the submission or arrival of a job until it completes execution.
- Waiting time is the time the job spends runnable but not executing. This is the sum of all periods the jobs spends in the runqueue, but it is not actually running.
- Response time is the time it takes from the point where a job becomes runnable to the point where output appears. In general this applies to interactive tasks. For example, how long does it take from the user pressing a key to the character being displayed on the screen?

- a) *RR (Round Robin)* We assume, that a task which enters the system can be immediately scheduled and it is at the beginning of the scheduling ring. This leads to the following schedule:

Task	00	10	20	30	40	50	60	70	80	90	100
1											
2											
3											
4											
5											

- (a) How big is the wait time per task?

1: 60ms, 2: 50ms, 3: 40ms, 4: 0ms, 5: 30ms

- (b) How big is the average wait time?

$$(60\text{ms} + 50\text{ms} + 40\text{ms} + 0\text{ms} + 30\text{ms}) / 5 = 36\text{ms}$$

- (c) How big is the turnaround time per task?

1: 90ms, 2: 70ms, 3: 60ms, 4: 10ms, 5: 60ms

- (d) The response time

In the worst case the task just lost its timeslice (is being preempted by the scheduler) when the user pressed a key. If the task is very fast, it can produce output immediately when it becomes running again. So with 5 jobs we have to wait for 4 jobs which are running in the meantime.

1: 5 jobs scheduled in RR with a timequanta of  $(5 - 1) * 10\text{ms} = 40\text{ms}$

- b) *EDF (Earliest deadline first)* The deadlines are absolute if the tasks are non-periodic. If two tasks have the same deadline, we assume that the first found task is going to run. This leads to the following schedule:

Task	00	10	20	30	40	50	60	70	80	90	100	110
1												
2												
3												
4												
5												

(a) *How big is the wait time per task?*

1: 30ms, 2: 60ms, 3: 10ms, 4: 0ms, 5: 30ms

(b) *How big is the average wait time?*

$$(30\text{ms} + 60\text{ms} + 10\text{ms} + 0\text{ms} + 30\text{ms}) / 5 = 26\text{ms}$$

(c) *How big is the turnaround time per task?*

1: 60ms, 2: 80ms, 3: 30ms, 4: 10ms, 5: 60ms

(d) *The response time*

This is less obvious than in RR scheduling. Since the task are scheduled by their deadline we can say, that if a schedule is feasible, the response time is at most (deadline - entry time - execution time). This leads to:

1: 40ms, 2: 70ms, 3: 10ms, 4: 0ms, 5: 40ms

c) *SRTF (Shortest remaining time first)* The job with the shortest execution time is always chosen to be executed. This scheduling might lead to starvation as long jobs might never be scheduled due to short running ones. SRTF can lead to the following schedule:

Task	00	10	20	30	40	50	60	70	80	90	100	110
1												
2												
3												
4												
5												

(a) *How big is the wait time per task?*

1: 50ms, 2: 0ms, 3: 0ms, 4: 10ms, 5: 30ms

(b) *How big is the average wait time?*

$$(50\text{ms} + 0\text{ms} + 0\text{ms} + 10\text{ms} + 30\text{ms}) / 5 = 18\text{ms}$$

(c) *How big is the turnaround time per task?*

1: 80ms, 2: 20ms, 3: 20ms, 4: 20ms, 5: 60ms

(d) *The response time*

There is no formalism that can be developed for SRTF for computing the response time. As SRTF can lead to starvation, it might happen that some jobs are never scheduled. As such, the response time can not be estimated.

## 1.2 General Questions

a) *What is the problem with shortest job first (SJF) scheduling policy?*

**Answer:** Long jobs are potentially never scheduled, if short jobs are entering the system continuously and if there is no preemption.

b) *What is the advantage of SJF?*

**Answer:** It minimizes the waiting time and the turnaround time.

c) *What is the benefit of round robin?*

**Answer:** It is easy to implement, understand and analyze. It has a good response time.

d) What is the big conceptual difference between EDF and RR?

**Answer:** EDF is a realtime-based scheduling strategy. Therefore tasks have priorities. RR treats all tasks the same. This is not a realtime scheduling strategy and there is no notion of priorities.

e) Why do hard realtime systems often not have dynamic scheduling?

**Answer:** In a dynamic setup it is not possible to guarantee the feasibility of a correct schedule. That means, if new tasks enter the system and the system allows that, it cannot be guaranteed that every task still meets the deadline. The other approach is to first compute whether there is enough time to allow a new task. If not, creation of a new task will fail. This guarantees that the already running task will always meet the deadlines, however it is possible that an important task cannot be created.

### 1.3 Realtime Scheduling

You are designing an HD-TV. To keep the production costs low, it has only one CPU which must perform the following tasks:

- Decode video chunks (takes 50 ms, has to be done every 200ms)
- Update Screen (takes 30 ms, has to be done every 200 ms)
- Handle user input (takes 10 ms, has to be done every 250 ms)

a) Show that it is possible to schedule all tasks in such a way that all deadlines are met, using Rate Monotonic Scheduling. Do not present a working schedule.

**Answer:** It can be shown that  $N$  periodic tasks with the computation times  $C_i$  and periods  $P_i$  can be scheduled by ordering the tasks according to their periods if

$$U = \sum_{i=1}^N \frac{C_i}{P_i} \leq N(2^{\frac{1}{N}} - 1)$$

For the given values we get:  $0.44 \leq 0.78$  (cf. last question).

b) Show that, when the number of tasks approaches infinity, all tasks can be scheduled without violating deadlines if the utilization is below 69.3%. Explain how one can arrive at this result.

**Answer:**  $\lim_{N \rightarrow \infty} (N(2^{\frac{1}{N}} - 1)) = \ln(2) \approx 0.693$

c) Now assume your boss wants you to add DRM to your TV. This requires an extra task with a period of 300 ms and a execution time of 100 ms.

(a) What is the utilization of the system now?

(b) What is the upper bound according to the theorem used in b)?

(c) If you try to build a working schedule by hand, you will see that it is still possible to create one. How can this be explained?

**Answer:**

Task $i$	Execution Time $C_i$	Period $P_i$	$C_i/P_i$	Utilization	Upper Bound
1	30	200	0.15	0.15	1.00
2	50	200	0.40	0.40	0.82
3	10	250	0.44	0.44	0.78
4	100	300	0.77	0.77	0.75

The upper bound given above states that if utilization  $U$ , for a set of tasks, is below  $N(2^{\frac{1}{N}} - 1)$  then it is always possible to schedule those tasks - however, it is not a tight upper bound, i.e., there are many cases where feasible schedules can be found even though the utilization reaches 1.

## 2 Processes Revisited

*Creating new processes in the Unix/Linux world is done using `fork()`. The `fork()` function clones an existing process and adds it to the runqueue, rather than really creating a new one. Since `fork` clones a process, they both execute the line after the `fork()` call. Now they need to distinguish whether they are parent or child process. This can be done by checking the return value of `fork()`: to the parent process, `fork()` returns the PID of the child process, to the child process, `fork()` returns 0*

### 2.1 Call `fork()` multiple times in a row

*Write a program which calls `fork()` multiple times in a row, e.g. three times. Each forked process shall print his level in the process tree and next wait for all child processes using the `waitpid()` method. Note that you can use `ps f` in order to verify if your program output matches the actual process tree.*

*What process tree do you expect?*

**Answer:** Calling `fork()` multiple times in a row creates an imbalanced tree. While the root process forks `n` child processes, the first child will fork `n-1` child process, the second child `n-2` child processes and so on.

### 2.2 Executing `ls -l` from your program

*Write a simple program (main function) which executes the `ls` program. Look up the manual page for the `exec` family (man 3 `exec`). After the `exec` call in your main function, have a `printf` which says that you have called `exec` now.*

*What do you notice?*

**Answer:** The line after `exec()` of your program will not be executed. `exec()` replaces the currently executed program by a new one. It does not automatically fork a new process to execute `ls -l`.

*How can you fix that?*

**Answer:** If you want your program to continue to run, first fork and execute `exec()` in the child. If you want your program to wait until the child process terminated, you can call one of the `wait()` functions (see man 2 `wait`).

### 2.3 Reading the `ls` output from a pipe

*Create a simple application which opens a pipe, executes `ls` and reads its output via the pipe. Your application should write a sentence and the output of `ls` to the console (example: "Output from `ls`: <ls output>").*

## 3 Implement a User-Level Cooperative Thread-Scheduler

**Answer:** You can download an example implementation from the web page.

### 3.1 Threads-package

In this section you should implement a user-level cooperative thread-scheduler. To keep it simple, you can implement a round-robin-based scheduler.

You will need to implement at least the following functions:

- `thread_create`
- `thread_add_runqueue`
- `thread_yield`
- `thread_exit`
- `schedule`
- `dispatch`
- `thread_start_threading`

Each thread should be represented by a TCB (a `struct thread` in the skeleton) which contains at least a function pointer to the thread's function and an argument of type `void *`. The thread's function should take this `void *` as argument whenever it is executed. This struct should also contain a pointer to the thread's stack and two fields which store the current stack pointer and base pointer when it calls `yield`.

`thread_create()` should take a function pointer and a `void *arg` as parameters. It allocates a TCB, allocates a new stack for this thread and sets default values. It is important that the initial stack pointer (set by this function) is at an address divisible by 8. The function returns the initialized structure.

`thread_add_runqueue()` adds an initialized TCB to the runqueue. Since we implement a round robin scheduler, it is easiest if you maintain a ring of those structures. You can do that by having a `next` field which always points to the next to be executed thread.

The static variable `current_thread()` always points to the currently executed thread.

`thread_yield()` suspends the current thread by saving its context to the TCB and calling the scheduler and the dispatcher. If the thread is resumed later, `thread_yield()` returns to the calling place in the function.

`thread_exit()` removes the calling thread from the ring, frees its stack and the TCB, sets the `current_thread` variable to the next to be executed thread and calls `dispatch()`. It is important to dispatch the next thread right here before returning, because we just removed the current thread.

`schedule()` decides which thread to run next. This is actually trivial, because it is a round robin scheduler. You can just follow the `next` field of the current thread. For convenience (for example for the dispatcher), it might be helpful to have another static variable which points to the last executed thread.

`dispatch()` actually executes a thread (the thread to run as decided by the scheduler). It has to save the stack pointer and the base pointer of the last thread to its TCB and it has to restore the stack pointer and base pointer of the new thread. This involves some assembly code (see recitation session). In case the thread has never run before, it may have to do some initializations, rather than just returning to the thread's context. In case the thread's function just returns, the thread has to be removed from the ring and the next one has to be dispatched. The easiest thing to do here is calling `thread_exit()`, since this function does that already.

`thread_start_threading()` initializes the threading by calling `schedule()` and `dispatch()`. This function should be called by your main function (after having added the first thread to the runqueue). It should never return (at least as long as there are threads in your system).

So in summary, to create and run a thread, you should follow the steps below:

```
static void thread_function(void *arg)
{
```

```

// ...
// may create threads here and add to the runqueue;
// ...

while(some condition, maybe forever) {
    do_work();
    thread_yield();
    if (exit-condition) {
        thread_exit();
    }
}
}

int main(int argc, char **argv)
{
    struct thread *t1 = thread_create(f1, NULL);
    thread_add_runqueue(t1);
    // ...
    // may create more threads and add to runqueue here;
    // ...
    thread_start_threading();
    printf("\nexited\n");
    return 0;
}

```

## 3.2 Test your threads-package

As a second step, implement a main function which creates a couple of threads which perform some operations so that we can see on the console that the threads are really running interleaved. Note: Since this is cooperative threading, your threads have to call `thread_yield()` from time to time.

Two simple functions might have a counter, one counting from 0-9 and another one counting from 1000-1009.

You may download `main.c` and a **skeleton** of `threads.h` from the courses website. The skeleton provides you the prototypes assumed by `main.c` (you don't need to follow this, you can have your own prototypes and can have an own `main.c`). The skeleton does not define the contents of TCB.