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.

1.2 General Questions

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

b) What is the advantage of SJF?

c) What is the benefit of round robin?

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

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

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.

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.

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?

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 then 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?

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?

How can you fix that?

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

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 threads 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 threads 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.