

1. Scheduling

The following table describes jobs to be scheduled. The table contains the entry times, duration, execution times, and deadlines of jobs. All values are given in milliseconds.

Job	Entry time	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 10 ms. Assume scheduling decisions take no time. The deadlines are absolute. Jobs that enter the system can be scheduled immediately.

(a) Creating schedules

Create schedules for the different types of policies listed below. Please visualize your schedules (like in the lecture) and also answer the questions below. If a policy does not clearly specify which job should be scheduled, always schedule the job with the lowest number.

Types of schedules:

- i. RR (round robin)
- ii. EDF (earliest deadline first)
- iii. SRTF (shortest remaining time first)

Please answer the following questions for each of the schedules:

- What is the waiting time for each job?
- What is the average waiting time for all job?
- What is the turnaround time for each job?
- How is the response time computed for this scheduler? If possible, calculate the response time for each job.

(b) General questions

- i. What is the problem with the SJF (shortest job first) policy?
- ii. What is an advantage of SJF?
- iii. What is a benefit of RR?
- iv. What is a major conceptual difference between EDF and RR?
- v. Why do hard real-time systems often not have dynamic scheduling?

(c) Real-time scheduling

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

- Decode video chunks (takes 50 ms and has to be done at least every 200 ms)
 - Update Screen (takes 30 ms and has to be done at least every 200 ms)
 - Handle user input (takes 10 ms and has to be done at least every 250 ms)
- i. Show that it is possible to schedule all tasks in such a way that all deadlines are met using rate-monotonic scheduling (RMS). Do not present a working schedule.
 - ii. Suppose DRM must be added to your TV. This requires an extra task with a period of 300 ms and an execution time of 100 ms.
 - α) What is the utilization of the system now?
 - β) What is the upper bound according to the theorem used in (ii).

`thread_create()` should take a function pointer and `void *arg` as arguments. It allocates a TCB and a new stack for the 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 run queue. Since we implement a round robin scheduler, it is easiest if you maintain a ring of `struct thread`'s. This can be done by having a linked list where the last node points to the first node.

The static variable `current_thread` always points to the currently executing thread.

`thread_yield()` suspends the current thread by saving its context to the TCB and calling the scheduler and the dispatcher. If the process is resumed later, it continues executing from where it stopped.

`thread_exit()` removes the caller from the ring, frees its stack and the TCB, sets `current_thread` to the next thread to be executed, 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. Simply select the next thread in the ring. For convenience (e.g., for the dispatcher), it may 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 restore the stack pointer and base pointer of the new thread. This involves some assembly code. In case the thread has never run before, it may have to do some initialization instead. If the thread's function 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 call `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 adding the first thread to the run queue). It should never return (at least as long as there are threads in your system).

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

```
static void
thread_function(void *arg)
{
    // Create threads here and add to the run queue if necessary

    while (condition) {
        do_work();
        thread_yield();
        if (condition)
            thread_exit();
    }
}

int
main(int argc, char **argv)
{
    struct thread *t = thread_create(f, NULL);
    thread_add_runqueue(t);
    // Create more threads and add to run queue if necessary
    thread_start_threading();
    printf("Done\n");
    return 0;
}
```