Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Operating Systems and Networks	Spring 2015
Assignment 2	Due 6 March 2015

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.

   (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.

   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.

   Solution:

   - Turnaround time: The time between arrival and completion of a job.
   - Waiting time: The time a job is runnable but not executing.
   - Response time: The time from arrival of a job to the point where output appears. In general, this is only for interactive jobs. For instance, how long does it take for a character to appear on the screen after a user presses a key?

   i. RR: Assume a job which enters the system can be scheduled immediately and is the next job to be run.

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

- Waiting time for each job: 1: 60, 2: 50, 3: 40, 4: 0, 5: 30
- Average waiting time: $(60 + 50 + 40 + 0 + 30)/5 = 36$
- Turnaround time for each job: 1: 90, 2: 70, 3: 60, 4: 10, 5: 60
- Response time: In the worst case, the job's time slice is used up when the user presses a key. Then it must wait for 4 other jobs before it will be run again. Assuming the job can produce output as soon as it is run, the response time is at most $(5-1) \times 10 = 40$ ms.

ii. EDF: The deadlines are absolute if the jobs are non-periodic. If two jobs have the same deadline, assume the first jobs we find is run.

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

- Waiting time for each job: 1: 30, 2: 60, 3: 10, 4: 0, 5: 30
- Average waiting time: $(30 + 60 + 10 + 0 + 30)/5 = 26$
- Turnaround time for each job: 1: 60, 2: 80, 3: 30, 4: 10, 5: 60
- Response time: Since jobs must meet their deadlines, a job is scheduled to start at (deadline - execution time) at the latest. Thus the maximum response time is (deadline - execution time - entry time). Assuming jobs produce output as soon as they are run, we get 1: 40, 2: 70, 3: 10, 4: 0, 5: 40.

iii. SRTF: The job with the shortest remaining time is always chosen to be executed.

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

- Waiting time for each job: 1: 50, 2: 0, 3: 0, 4: 10, 5: 30
- Average waiting time: $(50 + 0 + 0 + 10 + 30)/5 = 18$
- Turnaround time for each job: 1: 80, 2: 20, 3: 20, 4: 20, 5: 60
- Response time: The response time cannot be estimated for SRTF since starvation may cause certain jobs to never be scheduled.

(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?

Solution:

i. Long jobs may suffer from starvation if there is a continuous stream of incoming short jobs.

ii. It minimizes the waiting time and the turnaround time. This can be shown using the rearrangement inequality (proof given below).

Suppose there are n jobs and the execution times are $t_1, \ldots, t_n$ (in order of execution). Then the average waiting time is $\frac{(n-1)t_1+(n-2)t_2+\ldots+t_{n-1}}{n}$. Clearly, the average waiting time is minimal if and only if the total waiting time $(n-1)t_1 + (n-2)t_2 + \ldots + t_{n-1}$ is minimal, and this is guaranteed when $t_1 \leq \ldots \leq t_n$. Hence SJF is optimal is terms of average waiting time.
Since the turnaround time of a job is equal to (waiting time + execution time) and the execution time is a constant value, the average turnaround time is also minimal if and only if the average waiting time is minimal.

Theorem 1. *(Rearrangement inequality.) Let $n \in \mathbb{N}$. Suppose $a_1, \ldots, a_n, b_1, \ldots, b_n \in \mathbb{R}$ such that $a_1 \leq \ldots \leq a_n$ and $b_1 \leq \ldots \leq b_n$. Define $S_n = \{1, \ldots, n\}$. Let $\sigma : S_n \to S_n$ be a permutation of S_n. Then $a_1b_1 + \ldots + a_nb_n \geq a_{\sigma(1)}b_1 + \ldots + a_{\sigma(n)}b_n \geq a_nb_1 + \ldots + a_1b_n$.*

Proof. Suppose σ is not the identity map. Let j be the smallest element in S_n such that $\sigma(j) \neq j$. Then $\sigma(i) = i$ for all $i \in S_n$ satisfying $i < j$, so we have $\sigma(j) > j$. Furthermore, there exists a unique $k \in S_n$ such that $k > j$ and $\sigma(k) = j$. It follows that $(a_{\sigma(j)} - a_j)(b_k - b_j) \geq 0$ implies

$$a_jb_j + a_{\sigma(j)}b_k \geq a_{\sigma(j)}b_j + a_jb_k. \tag{1}$$

Thus we can obtain

$$a_1b_1 + \ldots + a_nb_n \geq a_{\sigma(1)}b_1 + \ldots + a_{\sigma(n)}b_n \tag{2}$$

by repeatedly exchanging elements using (1). Applying (2) to $a_1 \leq \ldots \leq a_n$ and $-b_n \leq \ldots \leq -b_1$ completes the second half of the rearrangement inequality. □

iii. It is easy to understand, analyze, and implement. It also has a good response time.
iv. EDF is a real-time scheduling strategy. Therefore, jobs have priorities. In contrast, RR treats all jobs as the same and does not prioritize.
v. In a dynamic setup, it is not possible to guarantee the feasibility of a correct schedule. That means if new tasks are allowed to be entered, we cannot guarantee all deadlines will be met. A possible solution is to compute whether there is sufficient time to meet deadlines before creating a new task. This guarantees existing tasks will always meet deadlines, but new important tasks may failed to be created.

(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).
   γ) If you try to construct a working schedule by hand, you will see it is still possible to create one. How can this be explained?
iii. For a general setting show the following: if the number of tasks approaches infinity and the utilization is below 69.3%, then all tasks can be scheduled without violating deadlines. Explain how you derive this result.

Solution:

i. It can be shown that N periodic tasks with computation times C_i and periods P_i can be scheduled by ordering the tasks according to their periods if

$$U = \sum_{i=1}^{N} \frac{C_i}{P_i} \leq N(2^{\frac{1}{N}} - 1). \tag{3}$$

For the given values, we get $U = 0.44 \leq 0.78$.

ii. If the utilization U does not exceed the upper bound given in Equation 3, then a working schedule can be constructed. However, the converse is not true, i.e., a working schedule need not have a utilization less than or equal to the upper bound. There are cases where a feasible schedule can be found even when $U = 1$.

Task	Execution time	Period	$\sum C_i/P_i$	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

iii. We want to compute $\lim_{N\to\infty} N(2^{\frac{1}{N}} - 1)$. By reformulating the problem and then applying L'Hôpital's rule, we obtain

$$\lim_{N\to\infty} N(2^{\frac{1}{N}} - 1) = \lim_{x\to 0} \frac{2^x - 1}{x} = \lim_{x\to 0} (\ln 2 \cdot 2^x) = \ln 2 \approx 0.693.$$

2. **Processes revisited**

Creating new processes in Linux (and other Unix-like operating systems) is done using `fork()`. The system call `fork()` clones an existing process instead of creating a completely new one.

Since `fork()` clones a process, both the parent and the child execute the same code after the invocation. It is necessary to distinguish to two processes, and this can be done by checking the return value of `fork()`. The parent receives the PID of the child process while the child gets 0.

(a) **Calling `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 its own level in the process tree and wait for all child processes using `waitpid()`.

What process tree do you expect? Note that you can use `ps f` to verify if your program output matches the actual process tree.

(b) **Executing `ls -l` from your program**

Write a program which executes `ls -l`. Look up the man page of the `exec()` family of functions using `man 3 exec`. (Note that each man page is associated with a section, and all the sections are described in the man page of `man`. Here we must specify section 3 to get the correct man page.)

After calling `exec()` (or one of its variants) in your program, add a `printf()` statement to show that `exec()` has completed.

When you run your program, what problem do you observe? How can it be fixed?

(c) **Reading the output of `ls` from a pipe** Create a program which opens a pipe. Execute `ls` and redirect the output into the pipe. Then read data from the pipe and print it. Your program should print a statement before writing data read from the pipe, e.g., "Output from ls: ...".

Solution:

(a) The process tree will be unbalanced. The root process will spawn n children, the first child will spawn $n-1$ children, the second child will spawn $n-2$ children, and so on.

(b) The function `exec()` never returns since it replaces the current process with a new program. Therefore, `printf()` will never be reached assuming `exec()` does not fail.
The problem can be fixed by calling `fork()` and let the child process call `exec()`. The parent can use `wait()` to wait until the child has completed.

(c) A sample implementation is available on the course website.
Note that shells make use of `pipe()` to implement pipelines. File descriptors are manipulated using `dup2()` after `fork()` but before `exec()`. For an example, see `http://pdos.csail.mit.edu/6.828/2014/xv6/xv6-rev8.pdf`.

3. **Processes and the kernel**

(a) If a process terminates but its parent has not called `wait()` (or a similar function) or "ignored" it, the process enters the zombie state. Explain why the process is put in the zombie state instead of being cleaned up in the kernel.

(b) Write a program which calls `fork()`, and make the parent process terminate while the child runs indefinitely. Does the child process get a new parent? If so, which process becomes its parent and is there anything special about the new process?

Solution:

(a) If the process is cleaned up, then the exit status will not be available if the parent requests it using `wait()` or `waitpid()`.

(b) The process receives a new parent whose PID is 1. The process with PID 1 is the first process started during boot and is typically `init` or `systemd` depending on the operating system.

4. **User-level threads**

(a) **Implementing user-level threads**

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

You will need to implement 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 (`struct thread` in the template) 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 `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 the 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;
}
```

(b) **Testing the user-level thread library**

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

Write two functions which maintain separate counters. One prints 0–9 while the other prints 1000-1009

Templates of `thread.h` and `main.c` are available from the course website. The template defines prototypes of functions used in `main.c`, but you may use your own ones.

Solution: A sample implementation is available on the course website.