

הטכניון – מכון טכנולוגי לישראל  
מעבדה במערכות הפעלה 046210  
תרגיל בית מס' 3

---

תאריך הגשה: 1.6.2023, עד 23:55

---

<b>Introduction</b>	<b>3</b>
<b>Working Environment</b>	<b>3</b>
<b>Detailed Description</b>	<b>3</b>
<b>Useful Information</b>	<b>4</b>
<b>Tips for the Solution</b>	<b>4</b>
<b>Testing Your Custom Kernel</b>	<b>4</b>
<b>Submission Procedure</b>	<b>4</b>
<b>Emphasis Regarding Grade</b>	<b>5</b>
<b>Background Information</b>	<b>6</b>
Task States	6
Kernel Timing	8
Task Scheduling	8

## Introduction

In the previous assignment you have added magic capabilities to processes. In this assignment you will learn about the scheduling algorithm and how to modify it.

Your mission in this assignment is to add another capability to magical processes – giving themselves more time. You will add a new system call **magic\_clock**, which will block the rest of the system for a limited amount of time, allowing the process calling it to use the CPU exclusively.

## Working Environment

You will be working on the same REDHAT 8.0 Linux virtual machine, as in the previous assignments.

## Background Information

Introduction to the scheduling can be found at the end of the file. More details are in the “The O(1) Scheduler” document in Moodle.

## Detailed Description

This assignment builds on the previous assignment. For this work you should use the magic system you implemented in the previous assignment.

When a process issues the system call **magic\_clock** the kernel should give the process exclusive access to the CPU for a given amount of time and starve all other running processes. After the given amount of time passes, the CPU should return to time-sharing mode as usual. If the calling process goes to sleep while given exclusive access, the CPU will **remain idle** until it wakes up. If the calling process dies while given exclusive access, scheduling should return to normal.

The system call should implement this interface:

**int magic\_clock(unsigned int seconds)**

- a. Description:  
Give the calling process exclusive CPU access for **seconds** seconds.  
The new system call should use the number 247.
- b. Return value
  - i. on failure: -1
  - ii. on success: 0
- c. On failure **errno** should contain one of following values:
  - i. “EPERM” (Operation not permitted): The calling process doesn’t have a wand.
  - ii. “ENOMEM” (No memory): Error allocating memory.

## Useful Information

- You can assume that the system is with a single CPU.
- You can assume there are no real-time processes in the system.
- More on system calls and task scheduling can be found in the “Understanding The Linux Kernel” book.
- Use **printk** for debugging.
- You are not allowed to use **syscall** functions to implement code wrappers, or to write the code wrappers for your system calls using the macro **\_syscall1**. You should write the code wrappers according to the example of the code wrapper given above.
- Use Bootlin (see link in Moodle) to easily find where some function/variable is defined in the kernel

## Tips for the Solution

- Exclusive CPU access can in practice be seen as giving a process a high priority and disabling any starvation-preventing mechanisms. Use the kernel’s existing priorities to implement this and modify it to your needs.
- Use the **add\_timer()** function from timer.c to set up a kernel timer. See example of how to use it in **schedule\_timeout()** in the same file.
- Time in the kernel is kept in jiffies, not seconds. See ex0 for an example of converting seconds to jiffies.
- Take the time to think where to put your changes to avoid breaking existing functionality.

## Testing Your Custom Kernel

You should test your new kernel thoroughly (all functionality and error messages that you can simulate). Note that your code will be tested using an automatic tester. This means that you should pay attention to the exact syntax of the wrapper functions, their names and the header file that defines them. You can use whatever file naming you like for the source/header files that implement the system calls themselves, but they should compile and link using the kernel make file. To do so add your source file in the following line inside the “Makefile” file located in the “kernel” folder:

```
obj-y += sched.o dma.o ... <your_file_name>.o
```

## Submission Procedure

1. Submission deadline: 1/6/23 till 23:55.
2. Submissions allowed in pairs only.
3. You should submit through the moodle website (one submission per pair).
4. You should submit one zip file containing:
  - a. All files you added or modified in your custom kernel (including relevant files from the previous exercise). The files should be arranged in folders that preserve their relative path to the root path of the kernel source, i.e:

```
zipfile -+  
|
```

```

+- submitters.txt
|
+- magic_api.h
|
+- kernel/ -+
|           |
|           +-...
|
+- include/ -+
|           |
|           +-...
...

```

- b. The wrapper functions file “magic\_api.h” that includes, in addition to the system calls from the previous exercise, the wrapper for **magic\_clock**.
- c. A file named “submitters.txt” which lists the name **email** and ID of the participating students. The following format should be used:

```

ploni almoni ploni@t2.technion.ac.il 123456789
john smith john@gmail.com 123456789

```

Note: that you are required to include your email.

## Emphasis Regarding Grade

- Your grade for this assignment makes 35% of final grade.
- Your submissions will be checked using an automatic checker, pay attention to the submission procedure.
- You are allowed (and encouraged) to consult with your fellow students but you are not allowed to copy their code.
- Your code should be adequately documented and easy to read.
- Incorrect submission format will be penalized by 5 points.
- Obviously, you should free all dynamically allocated memory.

## Introduction to Scheduling

This assignment requires basic understanding of the task scheduling in the Linux kernel. Below is some information to get you started. More information can be found in the recommended links in the lab Moodle and the OS course.

Linux is a multitasking operating system. A multitasking operating system achieves the illusion of concurrent execution of multiple processes, even on systems with single CPU. This is done by switching from one process to another very quickly. Linux uses **Preemptive Multitasking**. This means that the kernel decides when a process is to cease running and a new process is to begin running. Tasks can also intentionally **block** or **sleep** until some event occurs (keyboard press, passage of time etc.). This enables the kernel to better utilize the resources of the system and give the user a responsive feeling.

## Task States

The state field of the process descriptor describes what is currently happening to the process. The process can be in one of the following states:

**TASK\_RUNNING:** The process is either executing on a CPU or waiting to be executed.

**TASK\_INTERRUPTIBLE:** The process is suspended (sleeping) until some condition becomes true. Raising a hardware interrupt, releasing a system resource the process is waiting for, or delivering a signal are examples of conditions that might wake up the process (put its state back to TASK\_RUNNING).

**TASK\_UNINTERRUPTIBLE:** Like TASK\_INTERRUPTIBLE, except that delivering a signal to the sleeping process leaves its state unchanged. This process state is seldom used.

**TASK\_STOPPED:** Process execution has been stopped; the process enters this state after receiving a SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU signal.

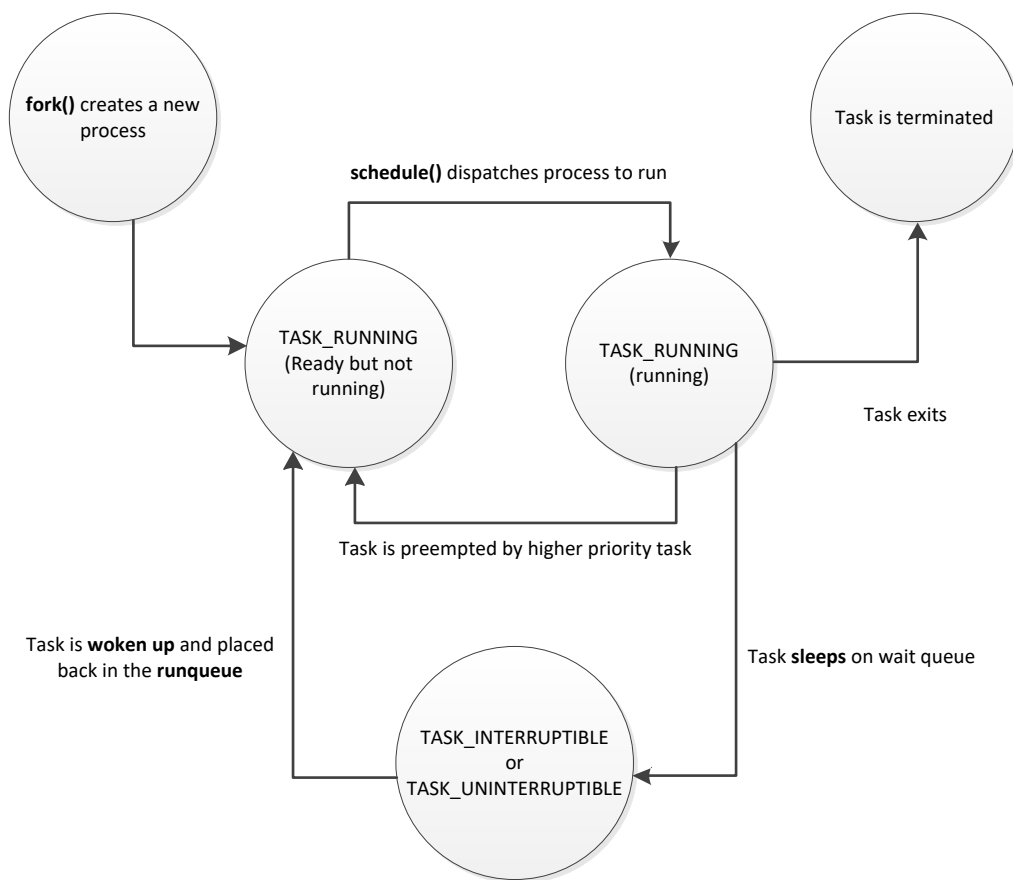
**TASK\_ZOMBIE:** Process execution is terminated, but the parent process has not yet issued a **wait()** or **waitpid()** system call to return information about the dead process.[\*] Before the wait()-like call is issued, the kernel cannot discard the data contained in the dead process descriptor because the parent might need it.

The value of the state field can be set using simple assignment, i.e,

```
p->state = TASK_RUNNING;
```

or using the macros **set\_current\_state** and **set\_task\_state**.

The life cycle of a process is shown in the following diagram:



## Task Scheduling

Note: The scheduling algorithm, called the O(1) scheduler, in our kernel (version 2.4) is taken from a kernel version 2.6. If you want to look online for information on the algorithm, look for information relevant for kernel version 2.6 (or use the algorithm's name).

The scheduling algorithm is implemented in “kernel/sched.c”, and most of the logic is in the **scheduler\_tick** and **schedule** functions. Linux scheduling is based on “time sharing” technique. The CPU time is divided into *slices*, one for each runnable process (processes with the **TASK\_RUNNING** state). A duration of the time slice depends on the priority of the process and ranges between 10ms to 300ms. Each CPU runs only one process at a time. The kernel keeps track of time using timer interrupts. When the time slice of the currently running process expires, the kernel scheduler is invoked and another task is set to run for the duration of its time slice. Switching between tasks is done through **context** switch. Switching of the currently running task can also occur before the expiration of its time slice. This can occur due to interrupts that wake up processes with higher priority or when the currently running process yields execution to the kernel (e.g. **blocks** or **sleeps**).

The kernel holds all runnable processes (processes with the **TASK\_RUNNING** state) in a data structure called a **runqueue**. The runnable processes are further divided to two arrays: processes that have yet to exhaust their time slice are in the **active** array, processes those whose time slice has expired are in the **expired** array. Each array is a collection of linked lists, one for each possible priority. The **runqueue** also points to the current running process (which obviously resides in the **active** array). Each time the **schedule()** function is called it selects the next running process by taking the first process from the first non-empty list in the **active** array. Once the time slice of a process expires it is moved to the **expired** array. When the **active** is empty, the **expired** and **active** arrays are switched, and the **active** processes are assigned new time slices.

When a process is put in the runqueue, the the **prio** field of **task\_t** is consulted to choose the correct queue number. For normal, non-realtime processes, this field is derived from two other values: user preference and a dynamic bonus. The user preferences is called a **nice value** and is encoded in the **static\_prio** field of **task\_t**, while the dynamic bonus is calculated based on how often the process needs the CPU and rewards processes that use it less often.

## Kernel Timing

The kernel keeps track of time using the *timer interrupt*. The timer interrupt is issued by the system timer (implemented in hardware). The period of the system timer is called ‘tick’. The *timer interrupt* advances the tick counter (called **jiffies**), and initiates time dependent activities in the kernel (decrease the time slice of the current running process, wake up processes that **sleep** waiting for a timer event etc.). The **jiffies** variable (defined in “include/linux/sched.h”) counts the system ‘tick’s event since start up. The ‘tick’ period duration depends on the specific linux version. The **HZ** variable is use for converting the ‘tick’s to seconds:

$$time\ in\ seconds = \frac{jiffies}{HZ}$$