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

תאריך הגשה: 1.07.2021, עד 23:55

## Introduction

In this assignment you will learn about the scheduling algorithm and how to update it. This assignment builds upon the previous one, so be sure to finish that before starting this one.

In the previous assignment you implemented a file-access limitation system and introduced the concept of privileged processes. However, since you finished the assignment some things have happened. The privileged processes have seized control of the government and started a dictatorship. They plan to change the rules of which processes are chosen to run and oppress the unprivileged processes. In the meantime, a resistance movement is starting to emerge.

Your mission in this assignment is to update the scheduling algorithm to prefer running processes that are privileged.

## Working Environment

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

## Compiling the Kernel

In this assignment you will apply modifications to the Linux kernel. Errors in the kernel can render the machine unusable. Therefore you will apply the changes to a ‘custom’ kernel. The sources of the custom kernel can be found in /usr/src/linux-2.4.18-14custom. All files that you will work with are in this directory. This kernel has already been configured and compiled. Below are the steps needed for recompiling the kernel after applying your changes:

1. Make your changes to the kernel source file(s).
2. Invoke **cd /usr/src/linux-2.4.18-14custom**
3. Invoke **make bzImage**. The bzImage is the compressed kernel image created with command **make bzImage** during kernel compilation. The name bzImage stands for "Big Zimage". Both zImage and bzImage are compressed with gzip. The kernel includes a mini-gunzip to uncompress the kernel and boot into it.
4. Invoke **make modules**
5. Invoke **make modules_install**
6. Invoke **make install**
7. Invoke **cd /boot**
8. Invoke **mkinitrd –f 2.4.18-14custom.img 2.4.18-14custom**
9. Invoke **reboot**. This command will restart the machine.
10. After rebooting choose "custom kernel" in the Grub menu.

The system should boot properly with your new custom kernel.

*Note*: Steps 4 & 5 are necessary only in case you touched any header files (*.h & *.S) since the last time you compiled the kernel. If you modify only kernel source files (*.c) then you can skip these steps and save compilation time.

## Detailed Description

This assignment builds on the previous assignment. For this work you should use the privilege system you implemented in the previous assignment and implement a scheduling algorithm for it. The updated scheduling algorithm should include the following rules:

1. For any two processes $P$ and $A$, if $P$ is privileged and $A$ is not, then $A$ is not allowed to run as long as $P$ wants to run (is in a RUNNING state).
2. Privileged processes are ordered by seniority: if two processes $P_1$ and $P_2$ are both privileged, then $P_2$ cannot run as long as $P_1$ wants to run if $P_2$ was marked as privileged after $P_1$.
    a. A process that was born privileged will use its creation time.
    b. Use the **jiffies** variable for this. See *Kernel Timing* section below.
3. If all RUNNING processes are not privileged, they are scheduled as usual.
4. A process that currently sleeps should not be scheduled for execution regardless of the new scheduling policy. Take for example, process A that is privileged and process B which is not . If process A is sleeping, then process B should still be scheduled and run.
5. If process $A$ kills process $B$ using SIGTERM (the Terminate signal, see *Signals* below), it inherits its privilege status and its seniority, but only if it's status will be improved as a result. Examples: $A$ is unprivileged and $B$ is privileged and is the most senior process (was marked as privileged first). $B$ goes to sleep, allowing $A$ to run and kill $B$. $A$ then become the most senior privileged process. If $B$ kills $A$ instead, nothing happens because $B$ is already better off than $A$.

## Background Information

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 and 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**.

### 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". 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 next task is selected according to its **priority**. This value plays an important part in the scheduling algorithm. The kernel uses it to distinguish between different types of processes:

- <u>Interactive processes</u>: Processes that interact with the user. An interactive process spends most of its life time in the **TASK_INTERRUPTIBLE** state waiting for user activity (e.g. key press). But when the event for which it is waiting occurs, it should become the current running process quickly or else the operating system will appear unresponsive to the user. Therefore, the priority of these tasks should be high.
- <u>Batch process</u>: These processes don't need the user's interaction, and usually run in the background. Typical batch processes are compilers and scientific applications.
- <u>Real-Time process</u>: RT processes should never be interrupted by a normal process (Interactive and Batch processes). These types of processes are used in time critical applications like video and motor controllers. These processes have the highest priority. As long as there are runnable RT processes normal processes are not allowed to run.

The **priority** of a process is a dynamic value that is determined both by the user (by setting **nice values**) and the kernel (by collecting statistics on the activity of the process).

The **nice value** of a process is a number between -20 and +19, saying how the user would like to prioritize the process compared to other (non-RT) processes in the system. Nice value of -20 is highest priority and nice value of +19 is lowest priority. The nice value of a child process is the same as its parent. The **nice** command can run a command with a different nice value. The `NI` field of the command `ps -le` shows the nice value of all processes. These can be fetched also using the `getpriority` syscall. The `renice` command can change a running process' nice value, along with the `setpriority` syscall.

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 processes which are yet to exhaust their time slice and those whose time slice has expired. The runnable processes are listed in two lists **active** and **expired** (according to the mentioned division) in the **runqueue**. The **runqueue** also points to the current running process (which obviously resides in the **active** list). Each time the **schedule()** function is called it selects the next running process from the **active** list. Once the time slice of a process expires it is moved to the **expired** list. When the **active** is empty, the **expired** and **active** lists are switched and the **active** processes are assigned new time slices.

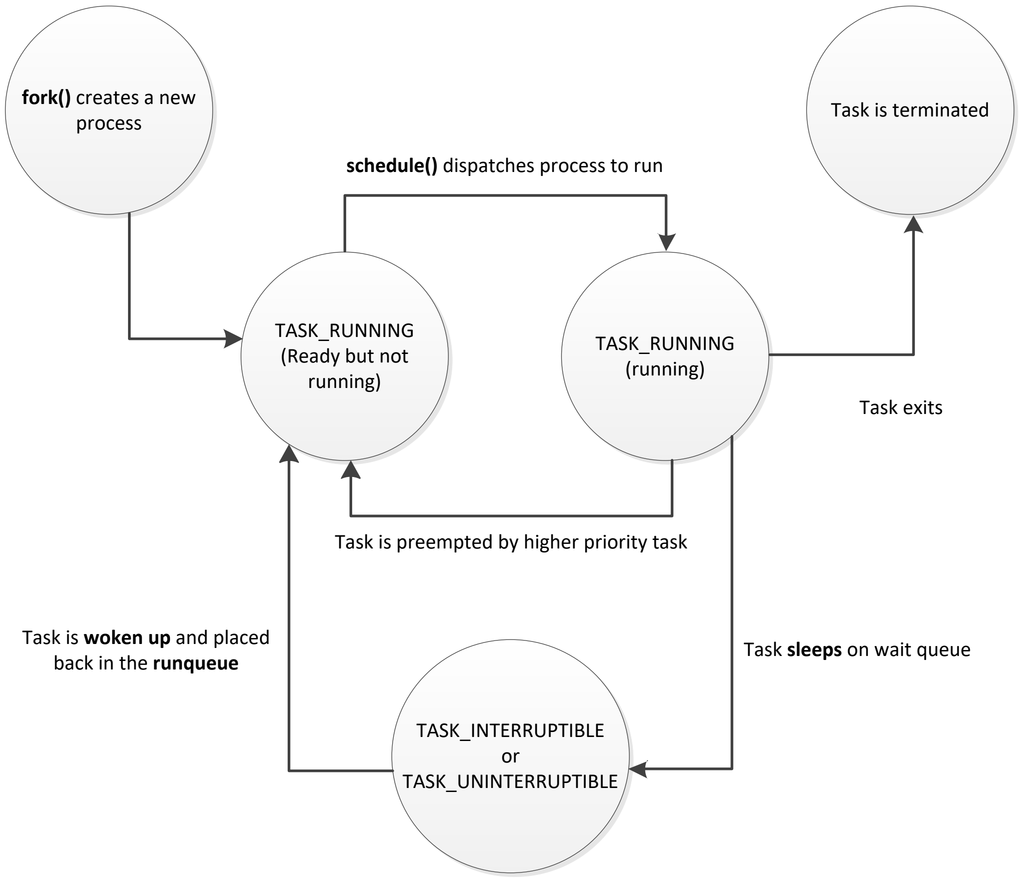
A process can yield its execution to the kernel. This usually happens when the process needs to wait for some event or for a specified period of time. A process can yield its execution by several means. The simplest one is to set its state to **TASK_INTERRUPTIBLE** and then call the **schedule()** function. For example process A can **sleep** by:

```
set_current_state(TASK_INTERRUPTIBLE);
schedule();
//
// After waking up, the process execution continues from here…
//
```

Another process can wake up process A by calling:

```
wake_up_process(process_A);
```

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


fork() creates a new process
Task is terminated
schedule() dispatches process to run
TASK_RUNNING (Ready but not running)
TASK_RUNNING (running)
Task exits
Task is preempted by higher priority task
Task is woken up and placed back in the runqueue
Task sleeps on wait queue
TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE

## 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:

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

## Signals

Processes can send each other signals, which are a basic way to interact. For example, when you press Ctrl-C to stop a command, you send the Interrupt signal (or SIGINT in short). Another common signal is the Terminate signal (SIGTERM), which is used to tell a process to quit. Signals are sent using the `kill` system call, defined as `sys_kill` in `kernerl/signal.c`. Another relevant function in the same file is `kill_proc_info` function, which is called by `sys_kill`. In the command line, you can use the `kill` command, which sends a SIGTERM to the given PID. In Python, you can use the `os.kill` function. We'll briefly describe the mechanism of sending signals.

When process A sends a signal to process B, it calls `sys_kill`, which finds process B and marks that it has a pending signal. Process B then wakes up, sees that it has a signal waiting and handles it according to which signal it is. For example, if the pending signal is SIGTERM the default action is to exit using the normal `do_exit` function in `kernel/exit.c` .

## Synchronization

A shared memory application means that multiple threads might access the same resources. Problem arises when several threads are trying to access the same resources concurrently. Code paths that manipulate shared data are called **Critical Paths**. This situation might result in one thread overwriting the data of another thread, or accessing data which is in unpredictable state. The solution is to use synchronizations methods that protect shared resources from concurrent access. Synchronization means that only a single thread (or a predetermined number of threads) can access a resource concurrently. Other threads wait till the first thread finishes and only then they are granted access to the resource. Synchronization should be applied carefully as bad synchronization can cause deadlocks. A deadlock is a condition involving one or more threads of execution and one or more resources, such that each thread waits for one of the resources, but all the resources are already held. The threads all wait for each other, but they never make any progress toward releasing the resources that they already hold. Therefore, none of the threads can continue, which results in a deadlock.

In the lab we concentrate on the linux kernel version 2.4. This version of the kernel is non preemptive, i.e. only a single process per cpu can use the kernel at a single moment. In a <u>multi-processor</u> system several processes can reside in the kernel at the same time, one per processor. But the specific version of the kernel we use is compiled in <u>uni-processor</u> mode; this means that in practice we don't need to worry about two processes that run in kernel at the same time. Note, however, a kernel control path can be interrupted by interrupts and therefore the currently executed function may be preempted and control will be returned to it at a later time. If you modify/access a value that may be changed during interrupt you should consider using synchronization primitives (e.g., mutex/spinlocks) for it.

## Useful Information

- You can assume that the system is with a single CPU.
- 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

## 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
```

## Tips for the Solution

- Start working on this exercise soon.
- Spend time on understanding how the kernel uses the processes' priority to make scheduling decisions.
- Use the functions **activate_task** and **deactivate_task** (definded in 'sched.c') to add or remove a process to/from the **runqueue**.
- To check if a process is running (in the **runqueue**) you can check the **array** field of its **task_struct** (if the field is set to NULL, then the process is not in the **runqueue**).
- Take the time to think where to put your changes so as to avoid breaking existing functionality

## Submission Procedure

1. Submission deadline: 1/07/21 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
            |
            +- block_api.h
            |
            +- kernel/ -+
            |           |
            |           +-...
            |
            +- include/ -+
            |            |
            |            +-...
           ...
   ```
   b. The wrapper functions file "block_api.h".
   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 40% 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.
- Delayed submissions will be penalized 4 points for each day (up to 24 points).
- Incorrect submission format will be penalized by 5 points.
- Obviously, you should free all dynamically allocated memory.