# הטכניון – מכון טכנולוגי לישראל
# מעבדה במערכות הפעלה 046210
# תרגיל בית מס' 2

תאריך הגשה: 31.5.2021, עד 23:55

*Introduction* 3

*Working Environment* 3

*Compiling the Kernel* 3

*Detailed Description* 4

*Background Information* 5

**The Task Structure** 8

**Process Creation and Termination** 8

**System Calls** 8

**Memory Management** 9

**Kernel Linked Lists** 9

*New System Calls API* 5

*Useful Information* 10

*Testing Your Custom Kernel* 10

*Submission Procedure* 11

*Emphasis Regarding Grade* 12

## Introduction

In the previous assignment you created and install a basic char device. In this assignment you will customize the kernel code itself.

Your mission in this assignment is to implement a system that allows you to specify a list of files that no process can open, except ones that are explicitly marked as privileged.

A regular process will not be able to open the limited files and will not be able to change the list of privileged processes. A privileged process will be able to open limited files and will be able to mark processes as privileged.

## Working Environment

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

## 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.

Important 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.

In order to not forget any files when you submit and to reduce the risks of strange bugs, it is highly recommended to create a separate directory where you put the files you modified from the kernel source and any new files you added. To compile the kernel with your modifications, use the make_changed.py script that comes with the pygrader package. In addition to the steps mentioned above, this script will also create a backup of the original kernel source code, so you'll always have a clean copy in case you want to start from scratch.

Example of modifying a kernel include file and compiling with the changes:

```
mkdir my_changes
cd my_changes
mkdir -p include/linux
cp /usr/src/linux-2.4.18-14custom/include/linux/sched.h include/linux/
# …
# edit include/linux/sched.h
# …
# Build the new kernel with the changes from the current directory
make_changed.py .
# If we messed up, we can return to a clean copy of the kernel
make_changed.py --reset
```

## Detailed Description

The system will have a global list of file paths that can only be opened by processes that are marked as privileged. When an unprivileged process tries to open() one of them, it will get the error EPERM (Operation not permitted). A process inherits its privileged status from its parent.

Note: If we wanted to create a real security system, we'd need to handle all edge cases involving different kinds of paths. Since paths have a *lot* of edge cases, the only limitation that will be checked are for **absolute paths** (start with /) that point to **regular files** (no links, directories, devices, etc).

Following is a list of the new system calls that you need to implement (see detailed API below):

1. **sys_new_open**: A modification on the original open() system call. Opens the file only if the process is privileged or not trying to access a limited file.
2. **sys_block_add_file**: Add a file to the limited files list
3. **sys_block_clear**: Clear the limited files list
4. **sys_block_query**: Check whether a file is in the limited file list
5. **sys_block_add_process**: Mark a process as privileged.

You are required to implement both the system calls and their wrapper functions (wrapper functions simplify the invocation of system calls from user space). Detailed description of the new system calls and their wrapper functions is given in the next section.

For marking a process as privileged, it is recommended (but not mandatory) to use **task_struct**, which is the database that stores all the information about a process. Since there is no limit to the number of limited files, the list of files should be implemented though dynamic allocation. For this, it's recommended (but again not

mandatory) to use the kernel's linked list mechanism (see the implementation in "include/linux/list.h" and its use in the kernel code).

When a process is created (using the **fork** mechanism) its privileged status should be the same as its parent's.

You can assume all strings passed to your system calls are null ('\0') terminated.

## New System Calls API

You should implement the following functions:

1. **long open(const char * filename, int flags, int mode)**
    a. Description:
    Check if **filename** is in the limited-files list. If it's not, or the process is marked as privileged, continue with open() as usual.
    b. Return value:
        i. If the file is limited and the process is not privileged: -EPERM
        ii. Otherwise: same as open()

2. **int block_add_file(const char *filename)**
    a. Description:
    Add **filename** to the list of limited files. If **filename** is already in the list, do nothing.
    b. Return value:
        i. on failure: -1
        ii. on success: 0
    c. On failure **errno** should contain one of following values:
        i. "ENOMEM" (Out of memory): Failure allocating memory.
        ii. "EFAULT" (Bad address): Error copying from user space, including if **filename** is NULL.
        iii. "EPERM" (Operation not permitted): The process is not privileged.

דומה ל
(ssize_t highscore_list(const char *board, unsigned int *buffer, ssize_t size .2

3. **int block_clear()**
    a. Description:
    Clear the list of limited files.
    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 process is not privileged.

4. **bool block_query(const char *filename)**
    a. Description:
    Checks whether **filename** is in the list of limited files. Note that unprivileged processes can also use this command.

דומה ל
(int highscore_chleague (pid_t pid .3

b. Return value:
    i. true if **filename** is in the list, false otherwise or if **filename** is NULL.
    ii. This function should never fail.

5. **Int block_add_process(pid_t pid)**
    a. Description:
    Mark the requested process as privileged. Only privileged processes can mark others as privileged, **unless** there are no privileged processes at all in the system. In this case, any process can mark any other process as privileged.
    b. Return value:
        i. On success: the current number of privileged processes.
        ii. On failure: -1

איזה סוג כישלון יכול להחזיר -1?

    c. On failure **errno** should contain one of the following values:
        i. "EPERM" (Operation not permitted): The process is not privileged (and there is at least one privileged process in the system).
        ii. "ESRCH" (No such process): The process is allowed to mark another process as privileged, but the process was not found.

Notes:

- If there are multiple errors that can be returned, you may return any of them.

Your wrapper functions should follow the example in the next page (note: this is an example from a previous HW):

```
int add_message(int pid, const char *message, ssize_t message_size)
{
        int res;
         __asm__
        (
                "pushl %%eax;"
                "pushl %%ebx;"
                "pushl %%ecx;"
                "pushl %%edx;"
                "movl $243, %%eax;"
                "movl %1, %%ebx;"
                "movl %2, %%ecx;"
                "movl %3, %%edx;"
                "int $0x80;"
                "movl %%eax,%0;"
                "popl %%edx;"
                "popl %%ecx;"
                "popl %%ebx;"
                "popl %%eax;"
                : "=m" (res)
                : "m" (pid) ,"m" (message) ,"m"(message_size)
        );

        if (res >= (unsigned long)(-125))
        {
                errno = -res;
                res = -1;
        }
        return (int) res;
}
```

This code uses inline assembler to call the 0x80 interrupt. Explanation:

1. **"pushl %%eax;"…**: Store any used registers in the stack.
2. **"movl $243, %%eax;"**: Store the system call number in register **eax.**
3. **"movl %1, %%ebx;"...**: Store the first parameter of the function in register **ebx.**
4. **"int $0x80;"**: Invoke the 0x80 interrupt.
5. **"movl %%eax,%0;"**: Store the return value (**eax**) in the output variable **%0**.
6. **"popl %%edx;"**: Pop back the stored registers.
7. **: "=m" (res)**: Map the output variable to variable **res**.
8. **: "m" (pid) ,…**: Map the input parameter **%1** to variable **pid**.

You can read more about inline assembly in the following link.

The wrapper functions should be stored in a file called “block_api.h”.

You should also implement the system calls. The new system calls should use the following numbering:

| System call | Number |
|---|---|
| sys_block_add_file | 243 |
| sys_block_clear | 244 |
| sys_block_query | 245 |
| sys_block_add_process | 246 |

dolev sys_highscore_list 244

# Background Information

## The Task Structure

The kernel stores all the information about a process in the **task_struct** structure. This information includes its pid, task state, file handles etc. **task_struct** is defined in “include/linux/sched.h”.

In this assignment you are required to keep track of each process’ privilege status. The natural location for this association is in the **task_struct**. Note that you will need to update both the **task_struct** definition and the **INIT_TASK** macro (defined in the same header file).

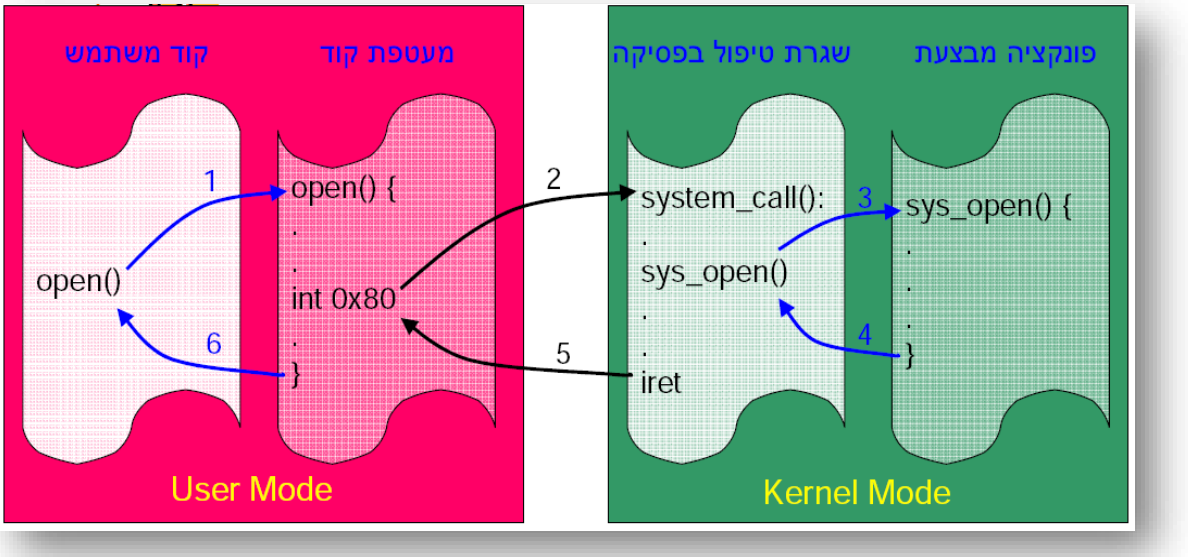
## Process Creation and Termination

Processes are created using the ‘fork’ mechanism (see the “Process” tutorial for more details on the fork system call). The function that does the actual ‘forking’ is called **do_fork()** and is defined in “kernel/fork.c”. Note that when the fork mechanism creates a child process, it copies the **task_struct** of the parent.

The **do_exit()** function is invoked each time a process terminates. It is defined in “kernel/exit.c”. This function handles freeing of allocated memory and any other necessary cleanup tasks.

## System Calls

A process uses the system calls mechanism to perform actions that require kernel functionality/privileges. The flow of a system call is described in the figure below. An application invokes a system call by calling its wrapper function (**open()** in the figure). The wrapper function passes control to the kernel using the 0x80 interrupt. The parameters of the system call, including its type (open, read, write, etc.), are passed by registers and not over the stack. The type of the system call is stored in register **aex** and the rest of the parameters are passed using registers **ebx**, **ecx,** etc. The 0x80 interrupt is handled using the **system_call()** interrupt handler. This handler uses the value in register **aex** as an index into the system calls handlers table, **sys_call_table**. This table maps each system call to its handler function (**sys_open** in the example below). Control is passed back to user space using the **iret** command. The return value is stored in register **aex**. The convention is that 0 means OK and negative values mean that an error occurred (The list of error codes is available in “include/asm-i386/errno.h”.




Creating a new system call involves the following steps:

1. Create the system call function. The new system call should be created in a new file that is compiled and linked with the kernel. The preferred place to put the source file is under the folder "kernel" and the header file under the folder "include/linux".
2. Add the system call source file to the kernel build mechanism. This is done by modifying the "obj-y" variable in the "kernel/Makefile" to include the object file of your system call source file.
3. Register the new system call handler in the **sys_call_table** (found in "arch/i386/kernel/entry.S"). This table links the system call number with the corresponding system call handler. It is used by the **system_call()** interrupt handler. You should add your new system call handler at the bottom of the table (as number 243 and up).
4. Create a wrapper function for the new system call. The wrapper functions are defined in a separate header file which you include in your application (in user space).

## Memory Management

The memory for the list of files should be allocated dynamically in the kernel. The kernel allocates space in physical memory using the **kamlloc** and **kfree** commands. These commands are used in the same manner as the **malloc** and **free** commands.

## Kernel Linked Lists

The Linux kernel uses its own implementation of a double linked list which is defined in "include/linux/list.h". The list is linked using elements of type **list_head**. To make a list of some data structure one needs to add a field of type **list_head** to the data structure, and use it to link to the **list_head** of next and previous data elements. The header file, "include/linux/list.h", defines several operations that can be performed on a list: add new element to the tail of the list, add an element after/before an existing element, remove an element, access the data structure of the list element, etc.

## Useful Information

- You can assume that the system is with a single CPU.
- More on system calls can be found in the "Understanding the Linux Kernel" book.
- Use **printk** for debugging (see link). It is easiest to see **printk**'s output in the textual terminals: Ctrl+Alt+Fn (n=1..6). Note, due to the fact that you are using the VMplayer you might need to press Ctrl+Alt+Space, then release the Space while still holding Ctrl+Alt and then press the required Fn.
- Use **copy_to_user** & **copy_from_user** to copy buffers between User space and Kernel space (see link).
- 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.

## Testing Your Custom Kernel

You should test your new kernel thoroughly (including 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.

## Submission Procedure

1. Submissions allowed in pairs only.
2. You should submit through the Moodle website (**Only one** submission per pair).
3. You should submit one zip file containing:
   a. All files you added or modified in your custom kernel. 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 names, **emails** and IDs 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

- Submission deadline: 31/05/21 till 23:55
- Your grade for this assignment makes 30% of final grade.
- Pay attention to all the requirements including error values.
- Your submissions will be checked using an automatic checker, pay attention to the submission procedure. Each submission error will reduce the grade by 5 points.
- 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).
- Wrong or partial implementation of system calls might make them work on your computer, but not on others. Therefore, it is recommended to verify that your submission works on other computers before submitting.
- The kernel is sophisticated and complex. Therefore, it is **highly important** to use its programming conventions. Not using them might cause your code and **other kernel mechanisms** to malfunction. **Note that failing to do so might harm your grade.**
- Obviously, you must free all the dynamically allocated memory.