סמסטר חורף תשע"ו

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

תאריך הגשה: 10.12.2015, עד 23:55

Introduction

In the previous assignment you learned how to create and install a basic char device. In this assignment you will learn how to customize the kernel itself. You will learn how to modify and compile the source of the Linux kernel.

Your mission in this assignment is to implement a TODO management system for processes. Each process will have its own TODO's queue and will manage it. Managing TODO's queue involves adding a new TODO to its queue, mark a TODO as completed and delete a TODO. A process will be able to manage its own TODO's queue and the TODO's queues of its descendant processes. TODO management will be done using a new set of system calls.

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.

Detailed Description

Each process should hold a queue of TODO tasks. When the process is created the queue is empty. Any process can manage, with some restrictions, its own TODO's queue or the TODO's queue of its descendants. A TODO is composed of a description, string of chars, and a status, int value. When created the status of a TODO is 0, that signifies that the TODO is uncompleted. The management of the TODO's queue is done through system calls. Following is a list of the new system calls that you need to implement:

1. **sys_add_TODO(pid, TODO_description, description_size)**: Add a TODO to the TODO's queue of a process identified by **pid**. **TODO_description** is a string of chars of size **description_size**. A process can add a TODO only to itself or to its descendants.
2. **sys_read_TODO(pid, TODO_index, TODO_description, description_size, status)**: Get the description of TODO identified by its position, **TODO_index**, in the TODO's queue of process **pid**. The description is returned in **TODO_description** which is a string of chars of size **description_size.** The status of the TODO is returned in **status**. A process can query the description of a TODO that belongs to itself or to one of its descendants.
3. **sys_mark_TODO(pid, TODO_index, status):** Set the status of a TODO identified by its position, **TODO_index**, in the TODO's queue of process **pid**. The status can be set to either uncompleted (**status** = 0) or completed (**status** != 0). A process can change the status of a TODO that belongs to itself or to one of its descendants.
4. **sys_delete_TODO(pid, TODO_index):** Delete a TODO identified by its position, **TODO_index**, in the TODO's queue of process **pid**. A process can delete a TODO that belongs to itself or to one of its descendants.

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 a later section.

The kernel stores the TODO's queue of each process in its **task_struct** which is the data base that stores all the information required for managing the process. There is no limit to the number of TODO's in the TODO's queue and to their length (except the available memory on your machine). Therefore the queue should be implemented through dynamic allocation. It is recommended to use the linked list mechanism included in the kernel (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 queue is empty (it does NOT inherit the TODO's of its parent). When a process is terminated all the TODO's in its queue are deleted and their memory is freed.

Your code should not assume that the **TODO_description** passed to **sys_add_TODO** includes a null char ('\0') at the end (even though C strings include the null char). This means that your code should handle the **TODO_description** as a buffer with a known size (**description_size**) and not as a null terminated string.

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 implement a TODO’s queue for each process. The natural location for this queue 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).

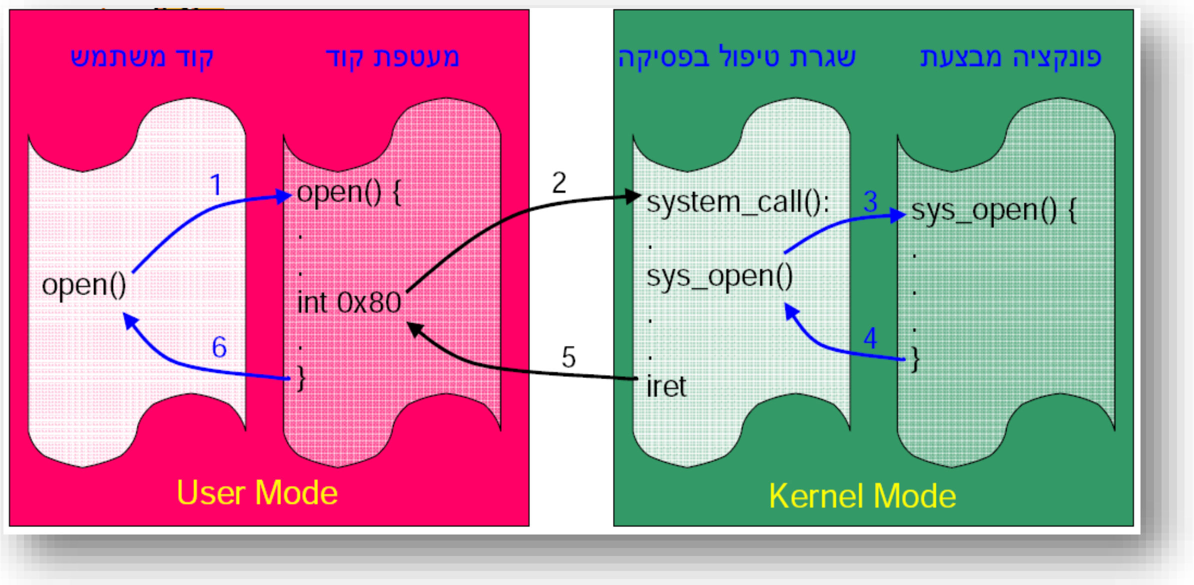
Process Creation and Termination

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. In this assignment each process should hold its own queue

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 TODO’s 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 (implementing a queue), remove an element, access the data structure of the list element etc.

New System Calls API

You should implement the following wrapper functions:

1. **int add_TODO(pid_t pid, const char *TODO_description, ssize_t description_size)**
   a. Description:
      Add a TODO to the TODO's queue of a process identified by **pid**. **TODO_description** is a string of chars of size **description_size**.
   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. "ESRCH" (No such process): No such process exists or the current process is not allowed to manage the TODO's queue of the process identified by **pid**.
      iii. "EFAULT" (Bad address): Error copying message from user space.
      iv. "EINVAL" (Invalid argument) **TODO_description** is NULL or **description_size** < 1.

2. **ssize_t read_TODO(pid_t pid, int TODO_index, char *TODO_description, ssize_t description_size, int* status)**
   a. Description:
      Get the description of TODO identified by its position, **TODO_index**, in the TODO's queue of process **pid**. The description is returned in **TODO_description** which is a string of chars of size **description_size**. The status of the TODO is returned in **status**.
      Note: It is possible to call this function with **description_size** larger than the size of the actual TODO's description. It is the responsibility of this function to copy only valid chars (it shouldn't copy chars beyond the TODO's description). The function returns the actual number of bytes copied to the TODO_description string.
   b. Return value:
      i. on failure: -1
      ii. on success: size of the string copied into **TODO_description**.
   c. On failure **errno** should contain one of following values:
      i. "ESRCH" (No such process): No such process exists or the current process is not allowed to manage the TODO's queue of the process identified by **pid**.
      ii. "EFAULT" (Bad address): Error copying message from user space.
      iii. "EINVAL" (Invalid argument) The TODO's queue size is smaller than **TODO_index**, **TODO_description** is NULL or **description _size** < the size of the description of the TODO.

3. **int mark_TODO(pid_t pid, int TODO_index, int status)**
    a. Description:
    Set the status of a TODO identified by its position, **TODO_index**, in the TODO's queue of process **pid**. The status can be set to either uncompleted (**status** = 0) or completed (**status** != 0).
    b. Return value:
        i. on failure: -1
        ii. on success: 0
    c. On failure **errno** should contain one of following values:
        i. "ESRCH" (No such process): No such process exists or the current process is not allowed to manage the TODO's queue of the process identified by **pid**.
        ii. "EINVAL" (Invalid argument) The TODO's queue size is smaller than **TODO_index**.

4. **int delete_TODO(pid_t pid, int TODO_index)**
    a. Description:
    Delete a TODO identified by its position, **TODO_index**, in the TODO's queue of process **pid**.
    b. Return value:
        i. on failure: -1
        ii. on success: 0
    c. On failure **errno** should contain one of following values:
        i. "ESRCH" (No such process): No such process exists or the current process is not allowed to manage the TODO's queue of the process identified by **pid**.
        ii. "EINVAL" (Invalid argument) The TODO's queue size is smaller than **TODO_index**.

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 “todo_api.h”.

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

System call	Number
sys_add_TODO	243
sys_read_TODO	244
sys_mark_TODO	245
sys_delete_TODO	246

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. Submission deadline: 10/12/15 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. 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
         |
         +- todo_api.h
         |
         +- kernel/ -+
         |           |
         |           +-...
         |
         +- include/ -+
         |            |
         |            +-...
        ...
```

   b. The wrapper functions file “todo_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.
- Pay attention to all the requirements including error values.
- 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 2 points for each day (up to 15 points).