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

תאריך הגשה: 30.06.2019, עד 23:55

Introduction

In the previous assignment you have implemented a simple Message Passing Interface (MPI) between process-es. In this assignment you will learn about the scheduling algorithm and how to update its policy.

Your mission in this assignment is to add scheduling mechanism to the MPI system. The system call **sys_receive_mpi_message** should be updated to be synchronous, i.e. it should not return immediately (as in HW 2) instead it should return only when an appropriate message is received.

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.

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

This assignment builds on the previous assignment. For this work you should use the Message Passing Interface (MPI) you implemented in the previous assignment and implement the scheduling algorithm for it. When a process issues the system call **sys_receive_mpi_message** the kernel should check if the process received a message from a process identified by its rank. If such message exists its value is returned. If not, the process (that issued **sys_receive_mpi_message**) is put to sleep till the appropriate message is received. A process sets a timeout to avoid being put to sleep indefinitely.

Below are the modifications to the system calls, required for the deadline mechanism:

1. **sys_receive_mpi_message(rank, timeout, message_size)**: check if the process received a message from the process identified by the rank, **rank**. If such message exists its values is returned in the buffer **message**, of length, **message_size**. If not, the process is put to sleep until an appropriate message is received or till **timeout** expires or when the process with rank, **rank**, terminates. The received message is deleted.
2. The system calls **sys_register_mpi** and **sys_send_mpi_message** are not changed.

The above system calls replace the corresponding system calls that you implemented in the previous assignment. 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 updated system calls and their wrapper functions is given in a later section.

Notes

1. The timeout is set in seconds and is relative to the value returned by the **C** function **time()**. For example if you want to set a timeout 60 seconds away, then the value you should use in the function **sys_receive_mpi_message** is 60.
2. The **C** function **time()** can be called only by User Mode applications. To get the current time from inside the kernel you can use the **CURRENT_TIME** macro defined in “sched.h”.

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 found in the recommended books and the following link (from the CS Operating System 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 **wait4()** 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

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). 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:

- Interactive processes: 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.
- Batch process: These processes don't need the user's interaction, and usually run in the background. Typical batch processes are compilers and scientific applications.
- Real-Time process. 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 and the kernel (by collecting statistics on the activity of the process).

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.

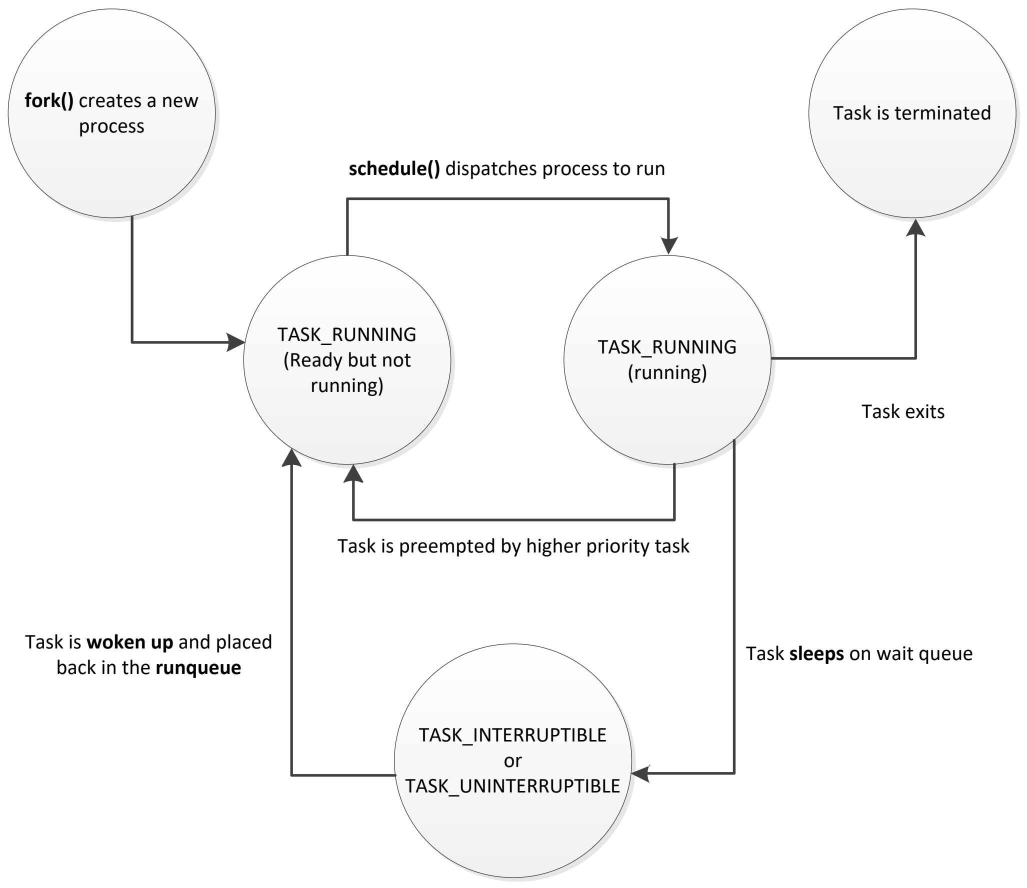
A process can yield its execution to the kernel. This usually happens when the process needs to wait for some event or for a specified period of time. A process can yield its execution by several means. The simplest one is to set its state to **TASK_INTERRUPTIBLE** and then call the **schedule()** function. 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.c”) counts the system ‘tick’s event since start up. To ‘tick’ period duration depends on the specific linux version. The **HZ** variable is use for converting the ‘tick’s to seconds:

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

Modified System Call API

You should implement the following wrapper function:

1. **int receive_mpi_message(int rank, int timeout, char *message, ssize_t message_size)**
   a. Description:
      Check if the calling process received a message from a process with the rank, **rank**. If such a message exists it is deleted from the messages queue and its value is copied into the buffer **message** of length **message_size**. If no such message exists in the queue the issuing process is put to sleep until an appropriate message arrives or **timeout** expires. If a new appropriate message arrives during **timeout** the sleeping process should wake up, return the received message value in **message** and delete the received message from the messages queue. Messages are processed in the order they are received.
      If **message_size** is shorter than the message available in the queue the function copies only the requested **message_size** bytes. If **message_size** is larger than the message available in the queue the function copies only the available bytes to **message**. You may assume that the buffer **message** is large enough to hold at least **message_size** bytes. The message is deleted from the messages queue after it is received even if it wasn't fully read.
      The function returns the <u>actual</u> number of bytes copied to the **message** string.
   b. Return value:
      i. on failure: -1
      ii. on success: size of the string copied into **message.**
   c. On failure **errno** should contain one of following values:
      i. "ETIMEDOUT" (Connection timed out): No message from a process with the rank, **rank**, exists in the message queue of the calling process, and an appropriate message hasn't arrived for **timeout** time.
      ii. "EFAULT" (Bad address): Error copying message to the user space.
      iii. "EINVAL" (Invalid argument) **message** is NULL or **message_size**<1 or **timeout**<0.
      iv. "ESRCH" (No such process): No such process exists. This can happen in one of three cases:
         1. There is no process with rank, **rank** available.
         2. Or that the receiving process is not in the MPI communication list.
         3. Or that the process with the rank, **rank**, has terminated before the timeout expiration period without sending a message to the receiving process.

<u>Notes:</u>

- In all system calls, if there are multiple errors that may be valid, e.g., <u>EINVAL</u> due to message being null and <u>ESRCH</u> due to the rank not being valid for the process, any of the expected faults would be considered a valid errno value.

Your wrapper functions should follow the example given in the previous assignment. The wrapper functions should be stored in a file called "mpi_messages_api.h". The system call should use the following numbering:

System call	Number
sys_register_mpi	243
sys_send_mpi_message	244
sys_receive_mpi_message	245

Useful Information

- You can assume that the system is with a single CPU and you don't need to handle concurrency issues in this exercise.
- More on system calls and task scheduling 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.
- 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 in HW2.

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

- The timeout mechanism should be independent of the test, i.e. it is not enough to test for a late timeout whenever one of your system calls is used.
- To put a process into sleep, you need to force the kernel not to schedule it for a period of time. One possible way to do this is to change the status of the process to **TASK_INTERRUPTIBLE** and set a timeout after which it will be woken. Setting a timeout can be done using the **schedule_timeout()** function (inside the **timer.c** file). Note that this function calls the **schedule()** function therefore you should not call it from inside the **schedule()** function itself. If you want to initiate a timeout from inside the **schedule()** function, you will need to manually set up a timer (see how it is done in the **sched-ule_timeout()** function).

Submission Procedure

1. Submission deadline: 30/06/19 till 23:55.
2. Submissions allowed in pairs or individuals.
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
            |
            +- mpi_messages_api.h
            |
            +- kernel/ -+
            |           |
            |           +-...
            |
            +- include/ -+
            |            |
            |            +-...
           ...
   ```
   b. The wrapper functions file "mpi_messages_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. 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 right 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 dynamic allocated memory.