CS3103 Project (Option B)

Note: We highly recommend using the Ubuntu environment on your own machine. Please DO NOT use the gateway server. Compiling errors may occur when using other operating systems, such as macOS or Windows.

The topics to be covered in this project:

- How to coordinate the execution of multiple processes with scheduling.
- How to achieve inter-process communication.

This project is based on a simple operating system kernel, basekernel, which is available on Github: *https://github.com/ruoxianglee/basekernel.* Basekernel is *not* a complete operating system, but it is a starting point for us to study and develop new operating system code. And this is raw low-level code, not a production system. Basekernel can boot an Intel PC-compatible virtual machine in 32-bit protected mode, with support for VESA framebuffer graphics, ATA hard disks, ATAPI optical devices, process management, memory protection, simple graphics, and basic filesystem. From there, it's your job to expand the system with other functionalities. The functionalities to be implemented for you includes:

a) a process scheduler with the priority scheduling policy.
b) inter-process communication with named pipe.

Introduction

1. Prepare environment

On your **Ubuntu OS**, install QEMU virtual machine and dependent packages:

```
sudo apt-get update
sudo apt-get install make
sudo apt-get install gcc
sudo apt-get install genisoimage
sudo apt-get install qemu-system-x86
```

QEMU (Quick Emulator) is an open-source virtualization software. It provides hardware emulation for a wide range of CPUs, including x86, ARM, PowerPC, and more, enabling you to run guest operating systems that are designed for different architectures.

2. Clone and build

Download the source code of basekernel and compile:

```
git clone https://github.com/ruoxianglee/basekernel
cd basekernel
make
```

3. Launch basekernel in QEMU virtual machine

```
qemu-system-i386 -cdrom basekernel.iso
```

If everything is OK, you will see the interface where you can interact with basekernel, just like Linux terminal. Follow *basekernel/README.md*, you can execute commands once you see the kernel shell prompt ("kshell>"). Try to take some actions and play with it. See *kshell_execute()* function in *kernel/kshell.c* for all supported commands, such as: automount, list, run, etc. Refer to *https://github.com/dthain/basekernel/wiki* for detailed introduction of basekernel (in cases where the detailed explanations deviate from the actual code implementation, it is essential to prioritize the code implementation as the authoritative reference).

Problem 1: Process Scheduler with Priority Scheduling Policy

In this problem, you should implement the priority scheduling policy into basekernel for process scheduling. Please implement the priority scheduling policy into basekernel, which **must** comply with the following requirements:

- Implement the priority scheduling policy into */kernel/process.c*, and also declare the corresponding functions in */kernel/process.h*.
- The priority scheduling policy must schedule all processes created by the system call *syscall_process_run()*, which are put into the ready queue (i.e., *ready_list* in */kernel/process.c*).
- The priority of a process should be specified as an input argument by user-level program when invoking the system call *syscall_process_run().*
- The supported priority of a process can be 0, 1, 2, 3, 4, ... Smaller integer represents higher priority.
- Implement 5 test programs into /user/*process1.c*, /user/*process2.c*, /user/*process3.c*, /user/*process4.c*, and /user/*process5.c*, with a priority **9, 7, 2, 1, and 5**, respectively.
- Each of the above 5 test programs should print its *pid* and *priority* once it is scheduled for execution and invoke the below *runForSeconds()* function to run for several seconds before exiting.
- Implement a start-up program into */user/schedulertest.c* to launch the above 5 test programs with the same order as their indexes. This program just serves as a start-up code for the above 5 test programs and doesn't create any process. The following is the high-level example about how to launch one of the test programs for your reference.

```
// invoked in test programs
void runForSeconds(int seconds)
{
    unsigned int startTime; // seconds
    syscall_system_time(&startTime);

    unsigned int timeElapsed;

    do {
        syscall_system_time(&timeElapsed);
        timeElapsed -= startTime;
    } while(timeElapsed < seconds);
}
```

```
//start-up program
const char * exec = "bin/process1.exe";

int argc;
const char *argv[3];
argv[0] = exec;
argv[1] = argument1;
argv[2] = argument2;

int pfd = syscall_open_file(KNO_STDDIR,exec,0,0);

if(pfd>=0) {
    int pid = syscall_process_run(pfd, argc,
&argv);
    if(pid > 0) {
        printf("started process %d\n", pid);
    } else {
        printf("couldn't run %s: %s\n",
argv[0],strerror(pid));
    }
    syscall_object_close(pfd);
} else {
    printf("couldn't find %s: %s\n",
argv[0],strerror(pfd));
}
```

Hint:

- Remember to modify *user/Makefile* for compiling your test program.

Supplementary Information:

The followings are some background information about process in basekernel, which has already been implemented, based on which you would figure out how to achieve the scheduler.

In generally, a process is an instance of an executing program. From the kernel's point of view, a process consists of: (1) user-space memory containing program code, (2) the variables used by that code, and (3) a set of kernel data structures that maintain information about the process's state (e.g. page tables, table of open files, process resource usage and limits...).

In basekernel, the process has five states: cradle, ready, running, blocked, grave. It is actually a struct data type, which contains a process id (*pid*) as well as a process parent id (*ppid*). (see */kernel/process.h*) The *pid* is allocated by a next fit search through an array of 1024 available ids. This array also serves to allow easy lookup of processes by id. In basekernel, the first process of the system is created with the kernel-level function *process_init()* (see */kernel/main.c*), which is the C initialization point of the kernel. Every newly created process with function *sys_process_run()* or *sys_process_wrun()* will be the child of the first process, and will be launched and be put into back of a ready queue (see *process_launch()*). Then, the processes in the ready queue will be scheduled with the First Come First Served policy for CPU execution. Please read the source code in */kernel/process.h* and */kernel/process.c* to better understand how the process is managed.

Problem 2: Named PIPEs

In this problem, you are going to implement the **Named PIPE** mechanism into basekernel, which should be used by the user-level programming with specific well-designed system calls. Please implement the Named PIPE into basekernel, which **must** comply with the following requirements:

- Implement your Named PIPE in *kernel/named_pipe.h* and *kernel/named_pipe.c* files.
- Add a new system call for creating a new Named PIPE, named as *syscall_make_named_pipe(const char *fname)*. The new file associated with the Named PIPE is named by the file name pointed to by the input argument *fname*.
- Add a new system call for opening a Named PIPE, named as *syscall_open_named_pipe(const char * fname)*. The file associated with the Named PIPE is specified by the file name pointed to by the input argument *fname*.
- Implement two test programs separately into */user/sender.c* and */user/receiver.c* to evaluate the correctness of your Named PIPE. The *sender* program should open a Named PIPE and then write some characters to *receiver* program, while the *receiver* program creates the Named PIPE and read the characters from the *sender* program. The followings are the high-level example of the test programs for your reference.

```
// receiver program
char * fname = "/path/to/file/filename";

int res = syscall_make_named_pipe(fname);
int fd = syscall_open_named_pipe(fname);

char buffer[20];
syscall_object_read(fd, buffer, 20);
printf("%s\n", buffer);
```

```
//sender program
char *fname = "/path/to/file/filename";

int fd = syscall_open_named_pipe(fname);
char buffer[] = "Hello World\n";

syscall_object_write(fd,buffer,strlen(buffer));
```

Hints:

- To implement the Named PIPE, you should call the APIs in */kernel/fs.h* for file operation, such as *fs_dirent_mkfile()* for new file creation.

- Remember to modify *user/Makefile* for compiling your test program.
- Remember to add named pipe to the struct *kobject* in */kernel/kobject.h*.
- To add new system calls, please modify */include/library/syscalls.h*, */library/syscalls.c*, */include/kernel/syscall.h*, and */kernel/syscall_handler.c*. Refer to the implementation of the open PIPE system call *syscall_open_pipe()*.

Supplementary Information:
The followings are some background about PIPE, which has already been implemented in basekernel, based on which you would figure out how to achieve a **Named PIPE**.

A PIPE is a byte stream (technically speaking, it is a buffer in kernel memory), which allows processes to exchange bytes. A PIPE has the following properties:

- *It is unidirectional*. Data travels only in one direction through a PIPE. One end of the PIPE is used for writing, the other one for reading.
- *Data passes through the PIPE sequentially*. Bytes are read from a PIPE in exactly the order they were written.
- *No concept of messages, or message boundaries*. The process reading from a PIPE can read blocks of data of any size, regardless of the size of blocks written by the writing process.
- Attempts to read from an empty PIPE blocks the reader until, either at least one byte has been written to the PIPE, or a no-terminating signal occurs.
- If the write-end of a PIPE is closed, then a process reading from the PIPE will see end-of-file once it has read all remaining data in the PIPE.
- A write is blocked until, either sufficient space is available to complete the operation atomically#, or a no-terminating signal occurs. (#on Linux, pipe capacity is 65536 bytes.)
- Writes of data blocks larger than PIPE_BUF* bytes may be broken into segments of arbitrary size (which may be smaller than PIPE_BUF bytes). (*on Linux, PIPE_BUF has the value 4096 bytes.)

In basekernel source code, there is an implemented PIPE mechanism (see *kernel/pipe.h* and *kernel/pipe.c*) and also the associated user-level test program (see *user/test/pipetest.c*). You will see how the PIPE is used for reading and writing between two child processes.

A **Named PIPE** (a.k.a, FIFO) is also a byte stream. Semantically, a Named PIPE is similar to a PIPE. The principal difference is that a Named PIPE has a name within the *file system*, and is opened and deleted in the same way as a *regular file*. This allows a Named PIPE to be used for **communication between *unrelated processes***, as long as the processes have the *file name*. Just as with PIPEs, a Named PIPE has a write-end and a read-end, and data is read from the Named PIPE in the same order as it is written.

Grading

We will grade your work with the following three considerations:

● Design (Project Report)	30%
● Implementation (Code)	30%
● Evaluation (Test Result)	40%

Design (30 %)

Your project report explaining your design details will account for 30% of your project grade. The project report should encompass the following content:

a) **Abstract idea and mechanism design (15%).** Please provide a detailed explanation of the abstract idea behind your mechanism design for both problems. Describe how you implemented the priority scheduling policy and the named pipe. You can utilize various methods to present your design, such as textual descriptions, flow diagrams, pseudo-code, or any other suitable means.

b) **Implemented functions (15%).** List and explain all the functions you have implemented for both problems. Ensure to include the input and output arguments for each function, as well as a clear description of their functionality. It is recommended to provide comprehensive coverage of all the relevant functions.

You could also discuss any difficulties encountered and the lessons learned in solving them.

In case your code cannot be compiled or does not give the correct result, you may still get part of the design scores according to the quality of your project report.

Implementation (30 %)

The implementation of your two programs will account for 30% of your project grade. Your code must be written by you own. The code should be nicely formatted with sufficient comments. Each function should be preceded by a header comment that describes what the function does. The code should be easy to read, properly indented, employ good naming standards, and structure, and correctly implement the design. Your code should match your design.

In case your code cannot be compiled or does not give the correct result, you may still get part of the implementation scores according to the quality and degree of correctness of the codes.

Evaluation (40 %)

The evaluation of your two programs will account for 40% of your project grade, with 20% for each Problem 1 and Problem 2. Your programs will be tested for correctness validation, ensuring that they work correctly. For each problem, the grading team will test your program based on both the test programs you have implemented and additional test cases specific to each problem.

Submission

1. Please submit the following files:

- Source code files of basekernel together with your own implemented source code for both problems (including the modified Makefile).
- Project report in PDF format based on report template for Project B. It should be named as "**XX-report.pdf**", where XX represents your group number. For example, if you are in

group #12, then your submitted report should be named as "12-report.pdf". Please list the names, student numbers, and emails of all group members at the beginning of the project report and describe each team member's contribution.

Please organize and compress the above listed files in a **zip format file** (named as **"XX-project.zip"**, where XX is your group number):

```
/XX-project
|── source code
|── XX-report.pdf     <- The project report file.
```

2. Submit to CANVAS
Self-sign-up is enabled for groups. Instead of all students having to submit a solution to the project, Canvas allows one person from each group to submit on behalf of their team. If you work with partner(s), both you and your partner(s) will receive the same grade for the project.

When you're ready to hand in your work, go to the course site in Canvas, choose the "Assignments" section > "Project" item > "Start Assignment" button, and upload your compressed zip file named as "**XX-project.zip**", where XX is your group number.

Academic Honesty

All work must be developed by each group independently. Please write your own code. **All submitted source code will be scanned by anti-plagiarism software.** We will both test your code and **carefully check your source code**. If your submitted code does not work, please indicate it in the report clearly.

Questions?

If you have questions, please first post them on Canvas so others can also get the benefit of the TA's answer. If the posts on Canvas do not resolve your issue, please contact the TA, Mr. LI Ruoxiang <ruoxiang.li@my.cityu.edu.hk>.