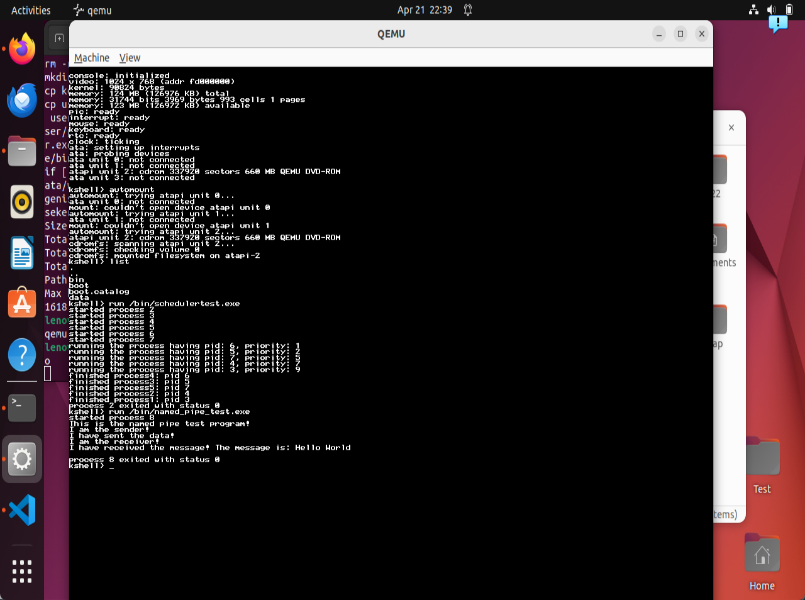
**CS3103 Project B Report**

**Group Information**

|  |  |  |  |
| --- | --- | --- | --- |
| Member | Name | SID | Email |
| #1 (contact person) | LIU Hengche | 57854329 | hengchliu2-c@my.cityu.edu.hk |
| #2 | WANG Ying | 57854262 | ywang3843-c@my.cityu.edu.hk |
| #3 | WAN Zimeng | 57222904 | zimengwan2-c@my.cityu.edu.hk |

Note: Please specify each team member's contribution if not all members make significant contributions to this project.

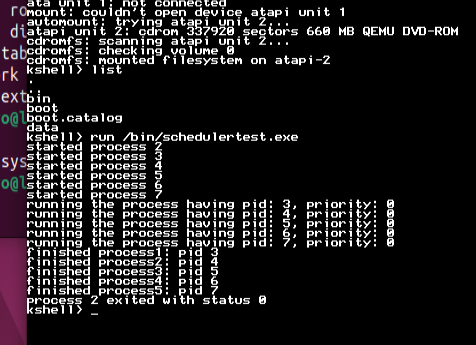
Everyone in the team contributed equally. Though as you can see in the screenshot below that we have successfully finished the two questions, we do acknowledge that we do not fully comprehend every detail of basekernel, and that’s why we also finished project A.

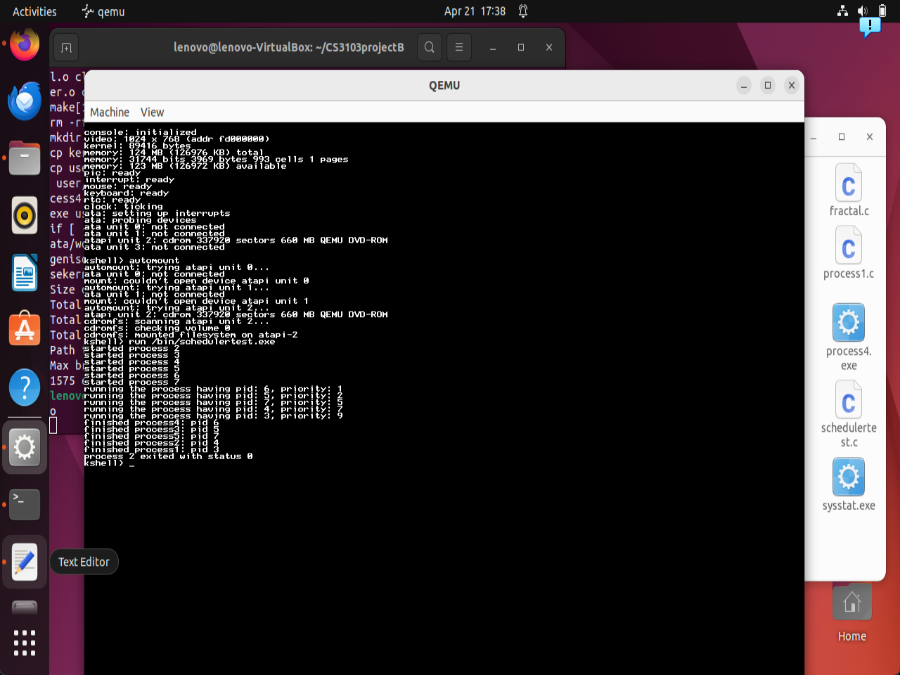


**Problem 1**

1) Have you successfully implemented the priority scheduling into the basekernel? Have your test programs executed properly? If not, please provide potential reasons for the issues.

yes. We run the /bin/schedulertest.exe, the results are as the following.





2) Abstract idea and mechanism design.

In order to sort process with its priority, we create a new list called priority\_waiting\_list, which will carry all process and sort them with its priority number. After we add all process, we will put them on ready list to run them one by one.

3) Implemented functions.

1.**Create user/schedulartest.c** (Top-most program)

Following are user/schedulartest.c, in which I have a **create\_process\_priority()** method to create process with it priority. It has three system call in this method. We don't need to change syscal\_open file and syscal\_object\_close. Since originally, there is not syscall\_process\_run take priority as a parameter, we **create a new system call named, syscall\_process\_prun**, which takes priority as a parameter, which will be introduced in "lib/Syscalls.c", "kernel/Syscal\_handler.c".

In the **main** method, it will call the create\_process\_priority() method and create 5 process with corresponding priority shown in "cs3103-project-b" document. After creating five process with priority, it will ask the system to run all the process that I created in the order of their priority. In this kernel program, when we run the process, we will print the pid and priority number and put process in to ready list, which will be down in kernel/process.c and kernel/list.c documents. Since we don't have specific program to put all waiting process into ready list, we **create syscall\_prun\_all** program to do it.

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

2.Create user/Process 1 2 3 4 5

process 1:

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

3.Change lib/Syscalls.c & h

Next step, in order to use the the new system call that we added in user/schedulartest.c, we need to add new system call in lib/Syscalls.c and lib/Syscalls.h.

This is new syscall\_process\_prun() in lib/Syscalls.h, which takes priority as a parameter.

图形用户界面, 文本, 应用程序

描述已自动生成

Next one is what we add in lib/Syscalls.c.

图片包含 图表

描述已自动生成

This is new syscall\_prun\_all() in lib/Syscalls.h, which runs all process in priority waiting list.

图形用户界面, 文本, 应用程序

描述已自动生成

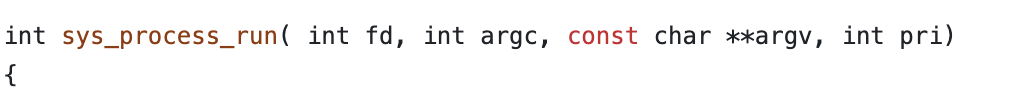
Next one is what we add in lib/Syscalls.c.

图片包含 徽标

描述已自动生成

4.Change kernel/Syscall\_handler.c & h

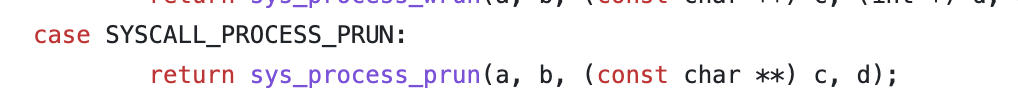
In kernel/Syscall\_handler.c, we create a new system called syscall\_process\_prun, which is similar with the "syscall\_process\_run". We only add a new parameter called "pri" standing for priority, and change the "process\_launch()" to "process\_priority\_launch()" which takes one more priority paramter.



图形用户界面, 文本

描述已自动生成

We also add it in syscall\_handler() in kernel/Syscall\_handler.c.



We also add sys\_prun\_all() in kernel/Syscall\_handler.c, which has "process\_run\_waiting\_process()" moving all processes that wait in priority list into ready list. We introduced "process\_run\_waiting\_process()" in kernel/process.c.

图形用户界面, 文本, 应用程序

描述已自动生成

图片包含 文本

描述已自动生成

5.Change kernel/process.c & h

we add process\_priority\_launch() and process\_run\_waitting\_process() in kernel/process.c. and kernel/process.h. process\_priority\_launch() put process in priority\_waiting\_list. We also add a list called priority\_waiting\_list to carry the process with priority.

图表

描述已自动生成

文本, 信件

描述已自动生成

图形用户界面, 文本, 应用程序

描述已自动生成

图形用户界面, 文本

描述已自动生成

图片包含 图示

描述已自动生成

6.Change kernel/list.c & h

Change list\_push\_priority(), from > to <, since the smaller the number, the larger the priority.

图形用户界面

低可信度描述已自动生成

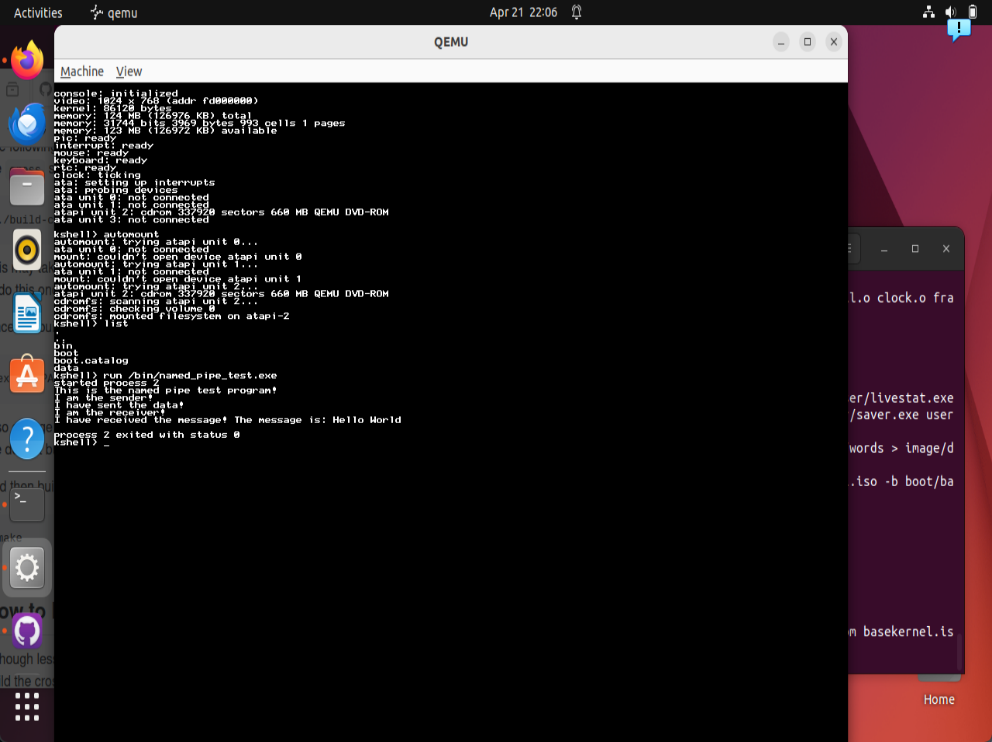
7.Change user/Makefile

we add schedulertest.exe process1.exe process2.exe process3.exe process4.exe process5.exe schedulertest.exe to it.

**Problem 2**

1. Have you successfully implemented the named pipe into the basekernel? Have your test programs executed properly? If not, please provide potential reasons for the issues.

Yes,



1. Abstract idea and mechanism design.

In this system, the central concept is the creation of a special type of file known as a “named pipe.” A named pipe facilitates communication between different programs by allowing them to read from and write to this file as if they were exchanging messages. Here’s a high-level overview of how it typically operates:

1. **Named Pipe Creation**:

A user-level program or process creates a named pipe using a specific system call (e.g., syscall\_make\_named\_pipe).

The named pipe is associated with a file name (specified by the user) and is stored in the kernel. It named pipe acts as a communication channel between different processes.

1. **Receiver Program**:

A receiver program (e.g., the one in /user/receiver.c) also opens the same named pipe.

The receiver reads data from the named pipe using the corresponding system call (e.g., syscall\_object\_read).

The data read from the named pipe buffer can be processed or displayed by the receiver program.

1. **Sender Program**:

A sender program (such as the one in /user/sender.c) opens the named pipe using another system call (e.g., syscall\_open\_named\_pipe).

The sender writes data (e.g., a message) into the named pipe using the appropriate system call (e.g., syscall\_object\_write).

The data is placed into the named pipe buffer.

1. **Inter-Process Communication**:

The sender and receiver programs can run independently and asynchronously.

The named pipe acts as an intermediary, allowing data to flow between them.

It provides a simple and efficient way for processes to exchange information without needing to explicitly manage sockets or other communication mechanisms.

In summary, named pipes enable inter-process communication by providing a shared channel where data can be transmitted between programs. They serve as a bridge for sending messages or data across different processes within the system.

3) Implemented functions.

Since named\_pipe is very similar to the pipe.c already implemented in the basekernel, we implemented the kernel/named\_pipe.c in the same concept. The named\_pipe struct is defined with three members: fname (the name of the pipe), file (a pointer to the file structure), and mes (a pointer to the message to be sent or received). There's also an array named\_pipes of pointers to named\_pipe structures, and an index to keep track of the next available slot in the array. The create\_file\_at\_path function takes a full path to a file and creates a file at that location. It first finds the last slash in the path to separate the directory and the filename. It then resolves the directory and creates the file in that directory with the given filename. The named\_pipe\_write and named\_pipe\_read functions are used to write a message to and read a message from the named pipe, respectively. They simply use the strcpy function to copy the message to/from the mes member of the named\_pipe structure. The named\_pipe\_create function creates a new named pipe with a given name. It first resolves the "bin" directory and creates a file named "named\_pipe" in that directory. It then allocates memory for a new named\_pipe structure and its members, and adds it to the named\_pipes array. Finally, the named\_pipe\_open function opens an existing named pipe with a given name. It searches the named\_pipes array for a named pipe with the given name and returns a pointer to it. If no such named pipe is found, it returns a pointer to the first named pipe in the array.

And then we have the header file of the named\_pipe.c with the function declared.

For the system\_call functions, we have sys\_make\_named\_pipe and sys\_open\_named\_pipe, which are system calls for creating and opening a named pipe, respectively. The sys\_make\_named\_pipe function takes a filename as an argument. It calls the named\_pipe\_create function, which creates a named pipe with the given filename. If the named pipe creation fails (i.e., named\_pipe\_create returns NULL), the function returns KERROR\_NOT\_FOUND. Otherwise, it returns 0 indicating successful creation of the named pipe. The sys\_open\_named\_pipe function also takes a filename as an argument. It first calls the process\_available\_fd function to get an available file descriptor for the current process. If no file descriptor is available (i.e., process\_available\_fd returns a negative value), the function returns KERROR\_NOT\_FOUND. Next, it calls the named\_pipe\_open function to open the named pipe with the given filename. If the named pipe opening fails (i.e., named\_pipe\_open returns NULL), the function again returns KERROR\_NOT\_FOUND. If the named pipe is successfully opened, the function creates a new kernel object for the named pipe by calling kobject\_create\_named\_pipe and assigns it to the current process's kernel object table at the index of the available file descriptor. Finally, it returns the file descriptor. In summary, these two functions provide a way for processes to create and open named pipes for inter-process communication. The error handling in these functions ensures that appropriate error codes are returned when the named pipe cannot be created or opened, or when no file descriptor is available.

And the first test program for the named\_pipe, i.e. name\_pipe\_test.c, we first tried to have two sub functions namely senderProgram and receiveProgram, but then we figured this is relatively simple and does not align with the test program for pipe given in pipetest.c, so we changed the format into forked processes. The main function starts by printing a message to indicate that it is the named pipe test program. It then defines a filename for the named pipe and calls the syscall\_make\_named\_pipe function to create a named pipe with that filename. It opens the named pipe by calling syscall\_open\_named\_pipe and stores the returned file descriptor in w. Next, the main function forks a new process by calling syscall\_process\_fork. The syscall\_process\_fork function returns 0 in the child process and the process ID of the child process in the parent process. Therefore, the if (x) condition is true in the parent process and false in the child process. In the parent process (the sender), the program prints a message to indicate that it is the sender. It then defines a buffer with the message to be sent, writes the message to the named pipe using syscall\_object\_write, and prints a message to indicate that it has sent the data. It then sleeps for 1000 milliseconds by calling syscall\_process\_sleep. In the child process (the receiver), the program prints a message to indicate that it is the receiver. It then reads the message from the named pipe using syscall\_object\_read and prints the received message. In summary, the test program demonstrates how to use named pipes for IPC in a parent and child process. The parent process writes a message to the named pipe, and the child process reads the message from the named pipe.

