**CS3103 Project B Report**

**Group Information**

|  |  |  |  |
| --- | --- | --- | --- |
| Member | Name | SID | Email |
| #1 (contact person) | Yonge Bai | 40151091 | yongebai2-c@my.cityu.edu.hk |
| #2 | Mahir Labib | 56749556 | mlabib2-c@my.cityu.edu.hk |
| #3 |  |  |  |

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

Table of Contents

[Problem 1 3](#_Toc164381664)

[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. 3](#_Toc164381665)

[2) Abstract idea and mechanism design. 3](#_Toc164381666)

[3) Implemented functions. 6](#_Toc164381667)

[3.1 schedulertest.c 6](#_Toc164381668)

[3.2 Makefile 8](#_Toc164381669)

[3.3 list.c 8](#_Toc164381670)

[3.4 process.c 9](#_Toc164381671)

[3.5 sys\_call\_handler.c 9](#_Toc164381672)

[3.6 syscalls.c 10](#_Toc164381673)

[3.7 runForSeconds(int seconds) 12](#_Toc164381674)

[Problem 2 12](#_Toc164381675)

[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. 12](#_Toc164381676)

[2. Abstract idea and mechanism design. 12](#_Toc164381677)

[Creating the Named Pipe: 12](#_Toc164381678)

[Using the Named Pipe: 13](#_Toc164381679)

[Deleting the Named Pipe: 13](#_Toc164381680)

[3. Implemented functions. 13](#_Toc164381681)

[3.1 ‘named\_pipe\_create(char \*fname)’ 14](#_Toc164381682)

[3.2 ‘named\_pipe\_addref(struct named\_pipe \*p)’ 14](#_Toc164381683)

[3.3 ‘named\_pipe\_flush(struct named\_pipe \*p) 14](#_Toc164381684)

[3.4 named\_pipe\_delete(struct named\_pipe \*p) 14](#_Toc164381685)

[3.5 ‘named\_pipe\_write(struct named\_pipe \*p, char \*buffer, int size)’ 15](#_Toc164381686)

[3.6 ‘named\_pipe\_write\_nonblock(struct named\_pipe \*p, char \*buffer, int size)’ 15](#_Toc164381687)

[3.7 named\_pipe\_read(struct named\_pipe \*p, char \*buffer, int size) 15](#_Toc164381688)

[3.8 ‘named\_pipe\_read\_nonblock(struct named\_pipe \*p, char \*buffer, int size)’ 16](#_Toc164381689)

[3.9 ‘named\_pipe\_size(struct named\_pipe \*p)’ 16](#_Toc164381690)

[3.10 t syscall\_make\_named\_pipe(const char \*fname) 16](#_Toc164381691)

[3.11 named\_pipe\_close(int fd) 16](#_Toc164381692)

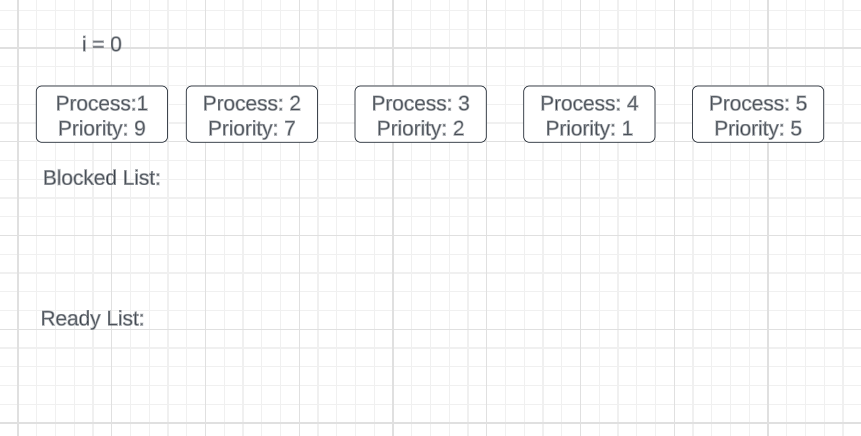
# **Problem 1**

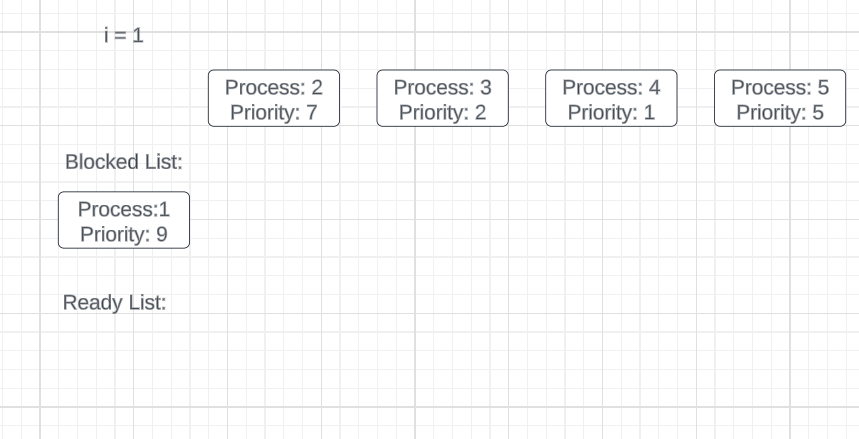
## **1) Have you successfully implemented the priority scheduling into the basekernel? Have your test programs executed properly? If not, please provide potential reasons for the issues.**

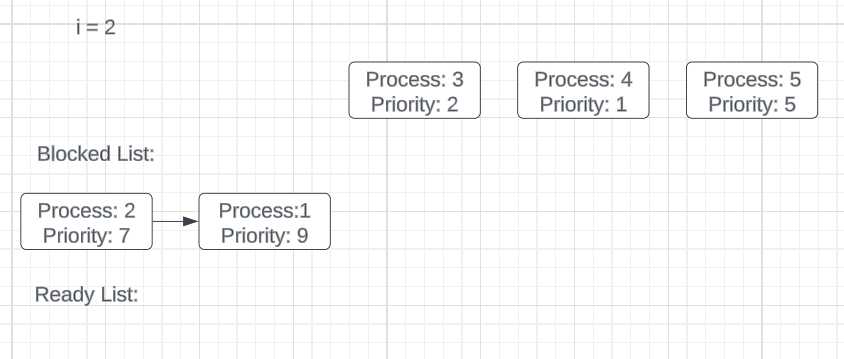
Yes, we have implemented the priority scheduling into the base kernel and have properly tested out the programs and confirmed that they are executing properly.

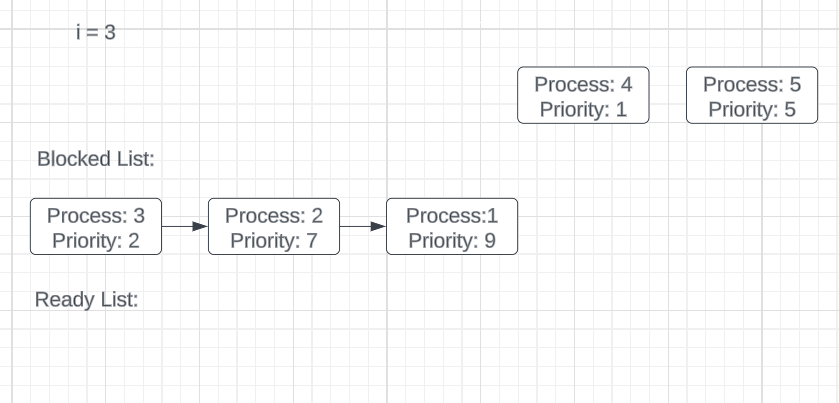
## **2) Abstract idea and mechanism design.**

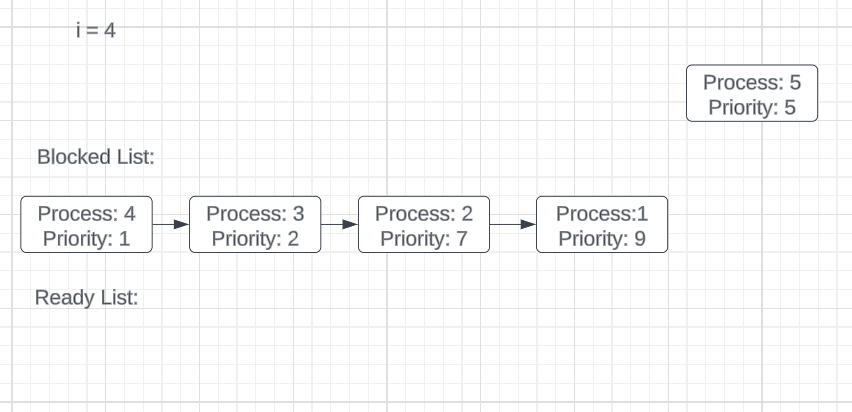
Creating processes:

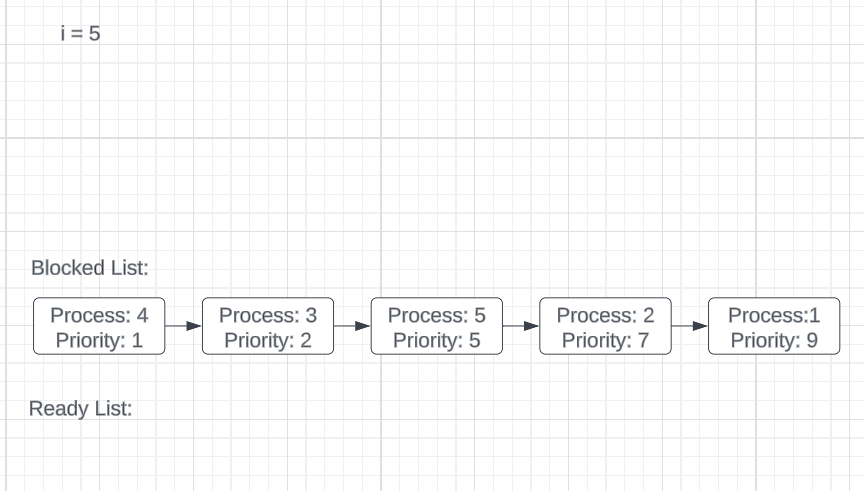


****

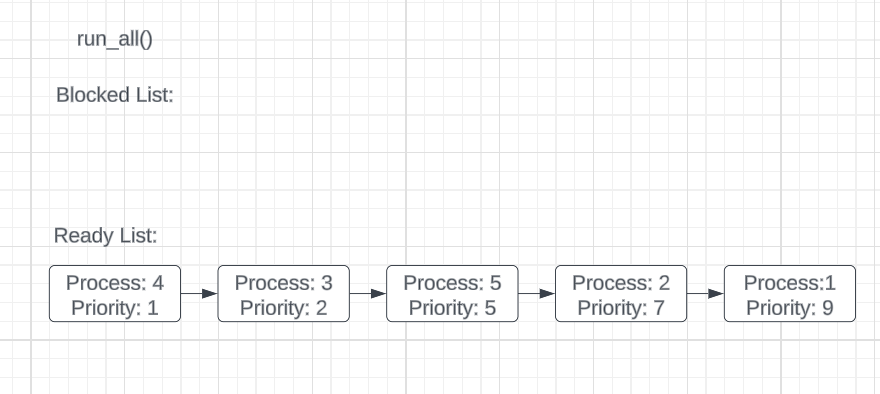




****



After all processes are created and added to blocked queue:

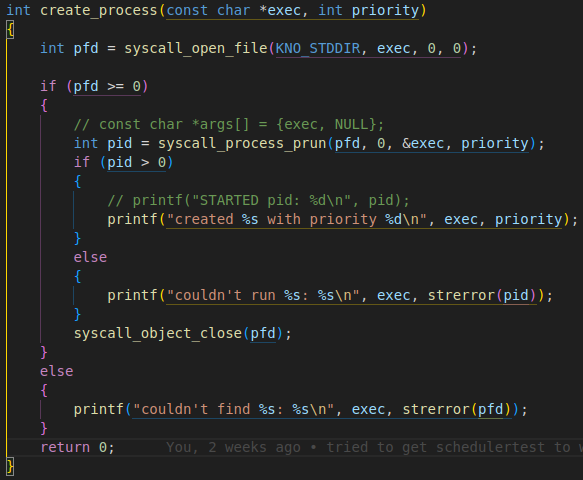


## **3) Implemented functions.**

### **3.1 schedulertest.c**

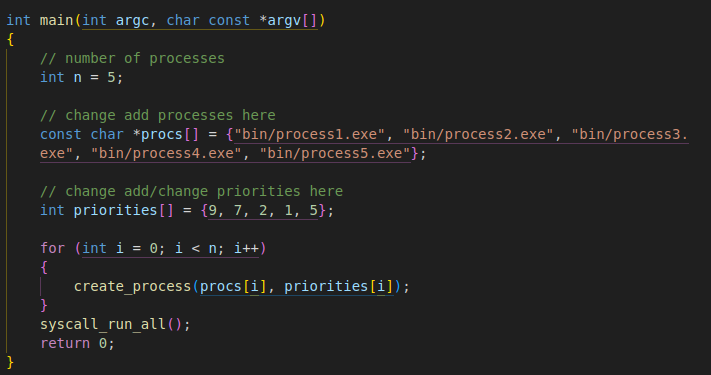
create\_process(const char \*exec, int priority), takes in a path to a process and a priority.

It opens the file specified in exec then creates a process using sys\_process\_prun(), a custom syscall that adds a process with priority to the blocked list in sorted non-decreasing order.



The main function sequentially creates each process with priority using the create\_process() function

Then once all the processes are created, it calls another custom syscall, run\_all() pushes all processes from the blocked list to the ready list.



### **3.2 Makefile**

In the Makefile, we have added additional executable targets to the “**USER\_PROGRAMS**” variables, as these are the new user programs that we have developed and added to the build process. and they are as follows. ***‘process1.exe’, ‘process2.exe’, ‘process1.exe’, ‘process2.exe’, ‘process3.exe’, ‘process4.exe’, ‘process5.exe’, ‘schedulertest.exe’, ‘named\_pipe\_test.exe’***

### **3.3 list.c**

**changed function:**

void list\_push\_head\_priority(struct list \*list, struct list\_node \*node, int pri)

to sort in nondecreasing order vs nonincreasing order to match with the project specification

Before:   
A black background with white text

Description automatically generated  
After:   
A black background with white text

Description automatically generated

### **3.4 process.c**

**Added Variable:**

struct list blocked\_list = {0, 0}

that stores blocked processes

**Added Function:**

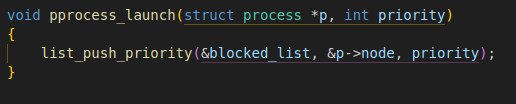
pprocess\_lauch() that uses the list\_push\_priority() function from list.h which takes in a:

destination list,

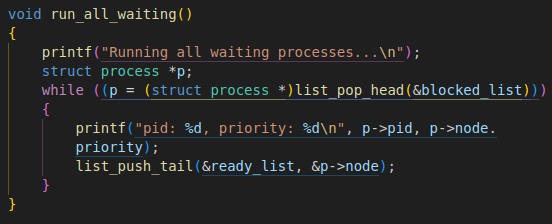
process’s listnode,

priority

and adds the listnode to the destination list in nondecreasing order.



run\_all\_waiting() that pushes all processes in the blocked list to the ready list, printing their priority and PID’s



Also added the function declaration to process.h

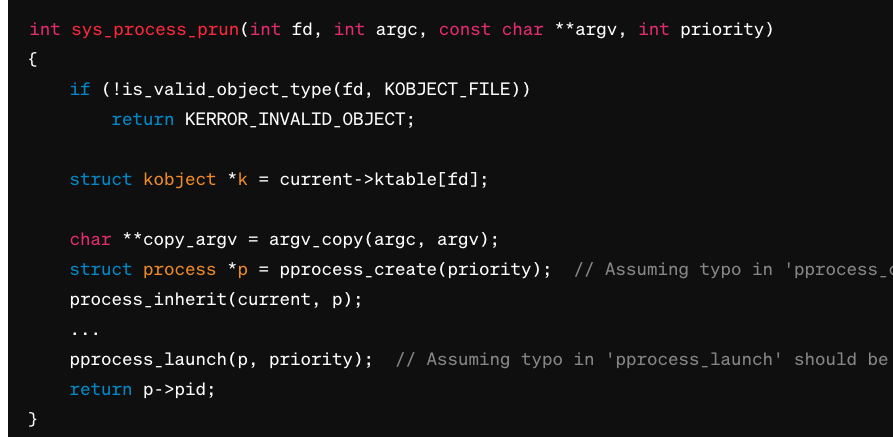
void pprocess\_launch(struct process \*p, int priority);

void run\_all\_waiting();

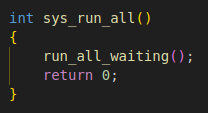
### **3.5 sys\_call\_handler.c**

**Added Function:**

sys\_process\_prun() that does the same as sys\_process\_run() but added an argument priority which gets passed into pprocess\_launch()



sys\_run\_all() which calls run\_all\_waiting() declared in process.h



### **3.6 syscalls.c**

**Added Function:**

**int syscall\_run\_all()**

**{**

**return syscall(SYSCALL\_RUN\_ALL, 0, 0, 0, 0, 0);**

**}**

and

int syscall\_process\_prun(int fd, int argc, const char \*\*argv, int priority)

{

return syscall(SYSCALL\_PROCESS\_PRUN, fd, argc, (uint32\_t) argv, priority, 0);

}

syscalls.h: adding SYSCALL\_PROCESS\_PRUN and SYSCALL\_RUNALL enums

**Case 1: Successful Process Creation**

**Input:**

A valid file descriptor (e.g., 3) representing an executable file.

An argument count (e.g., 2) and array of arguments (e.g., ["/path/to/executable", "--option"]).

A priority value (e.g., 10)

**Expected Output:**

A positive integer representing the new process ID.

**Case 2: Invalid File Descriptor**

**Input:**

An invalid file descriptor (e.g., -1).

An argument count and array of arguments.

A priority value.

**Expected Output:**

KERROR\_INVALID\_OBJECT, a predefined error code indicating an invalid object (assuming it is a negative integer).

**Case 3: Process Run Error (Hypothetical)**

Assuming further error checking is implemented in the missing parts of the function:

**Input:**

A valid file descriptor.

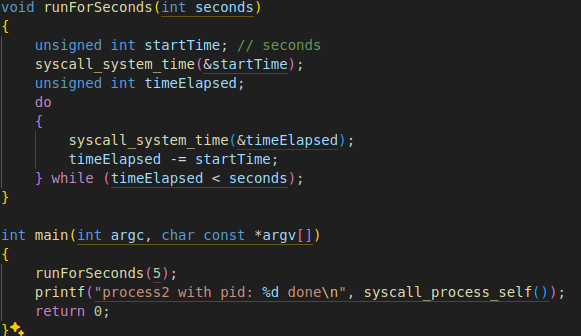
An argument count and array of arguments.

A priority value.

**Expected Output:**

A negative integer representing an error code if process creation fails for any reason (e.g., insufficient privileges).

### **3.7 runForSeconds(int seconds)**

Simply created processes outlined in project specification which run for a variable amount of time then print their PID

# **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.**

## **2. Abstract idea and mechanism design.**

In this system, the main idea is to create a special kind of file called a "named pipe". A named pipe allows different programs to communicate with each other by reading and writing data to this file as if they were sending messages back and forth. Here’s how it generally works:

### **Creating the Named Pipe:**

**Create a File:** First, a file is created in the file system. This isn’t a regular file; it's a placeholder that represents the named pipe.

**Create a Pipe**: Alongside the file, a pipe mechanism is created in the background. This pipe isn't visible like files in a folder but is a part of the system's way of handling data flow.

**Map Them Together:** The file and the pipe are then linked or "mapped" together. This means that whenever a program interacts with the file, it’s actually sending or receiving data through the pipe.

### **Using the Named Pipe:**

**Open:** Programs that need to communicate will open this named pipe (like opening a file) to send or receive data.

**Read/Write:** Data can be written to or read from the pipe. If a program writes data, it sends it through the pipe. If a program reads, it receives whatever data is sent from another program.

**Close:** Once the communication is done, the pipe can be closed, just like closing a file.

### **Deleting the Named Pipe:**

If the named pipe is no longer needed, it can be deleted. This involves removing the file and disassembling the pipe, cleaning up any resources used.

The beauty of this system lies in its simplicity and efficiency for allowing programs to interact with each other without needing to create more complex communication setups. It's like setting up a mailbox that programs can drop messages into or pick messages up from, facilitating easy and effective communication.

## **3. Implemented functions.**

**In this part of the report, we have compiled the functions added, in terms of the file names.**

### **3.1 ‘named\_pipe\_create(char \*fname)’**

**Purpose:** Creates and initializes a new named pipe structure.

**Input:** fname - A pointer to the name of the named pipe.

**Output:** Returns a pointer to a newly created named\_pipe structure if successful; otherwise, returns NULL.

* **Example:**
  + Input: "mypipe"
  + Output: Pointer to initialized named\_pipe structure.

### **3.2 ‘named\_pipe\_addref(struct named\_pipe \*p)’**

**Purpose:** Increments the reference count of the named pipe, indicating another part of the program is using the pipe.

**Input:** p - A pointer to the named\_pipe structure.

**Output:** Returns the same pointer passed as input after incrementing its refcount.

* **Example:**
* Input: Pointer to a named\_pipe
* Output: Same pointer with increased reference count.

### **3.3 ‘named\_pipe\_flush(struct named\_pipe \*p)**

**Purpose:** Flushes the named pipe by setting its flushed status, typically to allow any waiting processes to continue.

**Input:** p - A pointer to the named\_pipe.

**Output:** None.

### **3.4 named\_pipe\_delete(struct named\_pipe \*p)**

**urpose:** Decreases the reference count of a named pipe and frees its resources if the count reaches zero.

**Input:** p - A pointer to the named\_pipe.

**Output:** None.

### **3.5 ‘named\_pipe\_write(struct named\_pipe \*p, char \*buffer, int size)’**

**Purpose:** Writes data to the named pipe in a blocking manner.

**Input:**

* p - Pointer to the named\_pipe.
* buffer - Pointer to the data buffer to be written.
* size - Size of the data to write.

**Output:** Returns the number of bytes written.

**Example:**

**Input**: Pipe pointer, "Hello", 5

**Output**: 5 (if all bytes were written)

### **3.6 ‘named\_pipe\_write\_nonblock(struct named\_pipe \*p, char \*buffer, int size)’**

**Purpose:** Writes data to the named pipe in a non-blocking manner.

**Input:** Same as named\_pipe\_write

**Output:** Returns the number of bytes written or 0 if no space is available

### **3.7 named\_pipe\_read(struct named\_pipe \*p, char \*buffer, int size)**

**Purpose:** Reads data from the named pipe in a blocking manner.

**Input:**

p - Pointer to the named\_pipe.

buffer - Pointer to the buffer where the data will be stored.

size - Maximum number of bytes to read.

**Output:** Returns the number of bytes actually read.

**Example:**

Input: Pipe pointer, buffer, 10

Output: Number of bytes read (depends on available data)

### **3.8 ‘named\_pipe\_read\_nonblock(struct named\_pipe \*p, char \*buffer, int size)’**

**Purpose:** Reads data from the named pipe in a non-blocking manner.

**Input:** Same as named\_pipe\_read.

**Output:** Returns the number of bytes read or 0 if no data is available.

### **3.9 ‘named\_pipe\_size(struct named\_pipe \*p)’**

**Purpose:** Retrieves the size of the named pipe, which is set to the page size (PAGE\_SIZE).

**Input:** p - Pointer to the named\_pipe.

**Output:** Returns the size of the pipe (PAGE\_SIZE).

### **3.10 t syscall\_make\_named\_pipe(const char \*fname)**

**Purpose:** This function provides a simplified interface for making a system call to create a new named pipe. It abstracts away the details of the system call, allowing the programmer to request named pipe creation with a simple function call.

**Input:** fname - A pointer to a null-terminated string that represents the name or path of the named pipe to be created.

**Output:** The function returns an integer value. If the named pipe is successfully created, it typically returns 0. If there's an error during the creation process, it returns a negative error code corresponding to the type of error that occurred.

### **3.11 named\_pipe\_close(int fd)**

**Purpose:** Closes a named pipe and releases its resources.

**Input:** fd - File descriptor of the named pipe.

**Output:** None (function should typically return an integer status, but the provided snippet does not specify a return value).