**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.

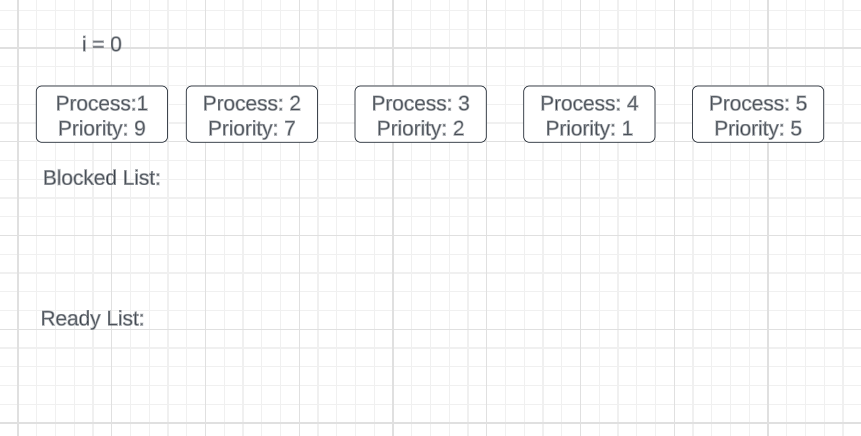
# Problem 1

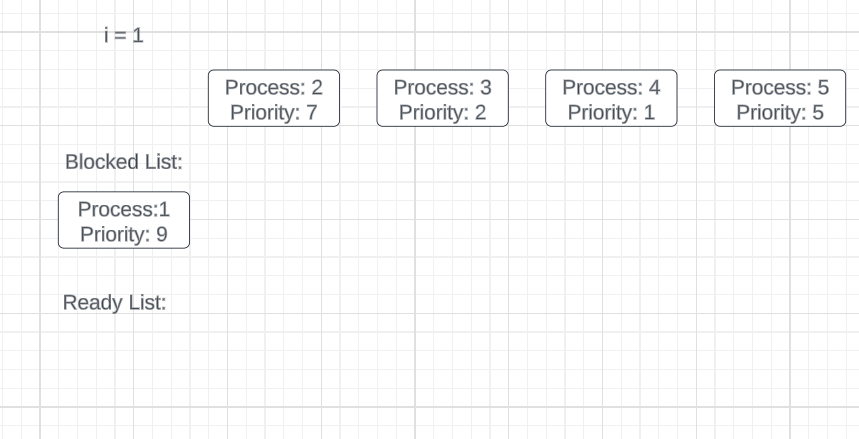
# 1) Have you successfully implemented the priority scheduling into the basekernel? Have your test programs executed properly? If not, please provide potential reasons for the issues.

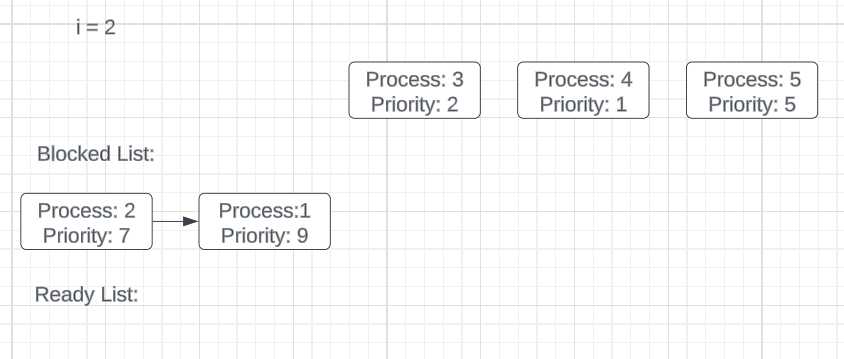
Yes, we have implemented the priority scheduling into the base kernel and have properly tested out the programs and confirmed that they are executing properly.

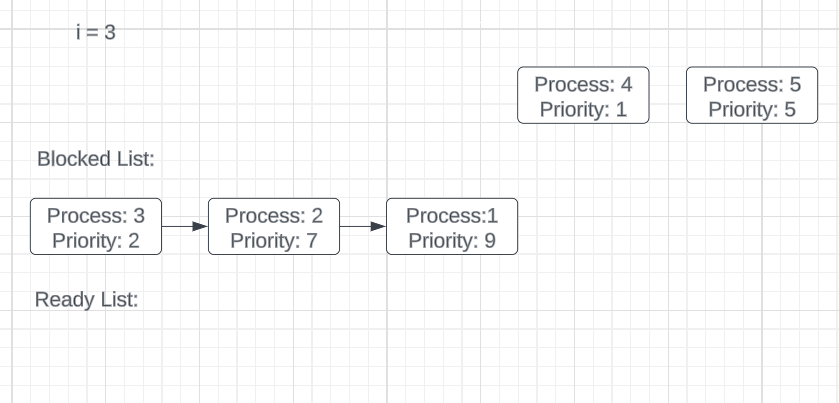
# 2) Abstract idea and mechanism design.

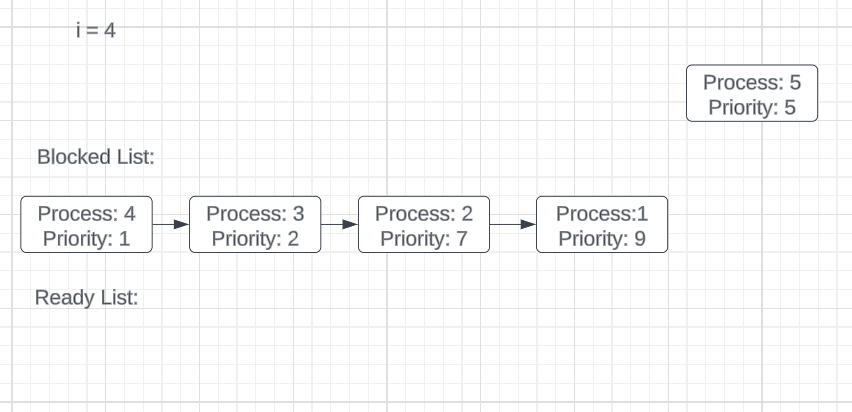
Creating processes:

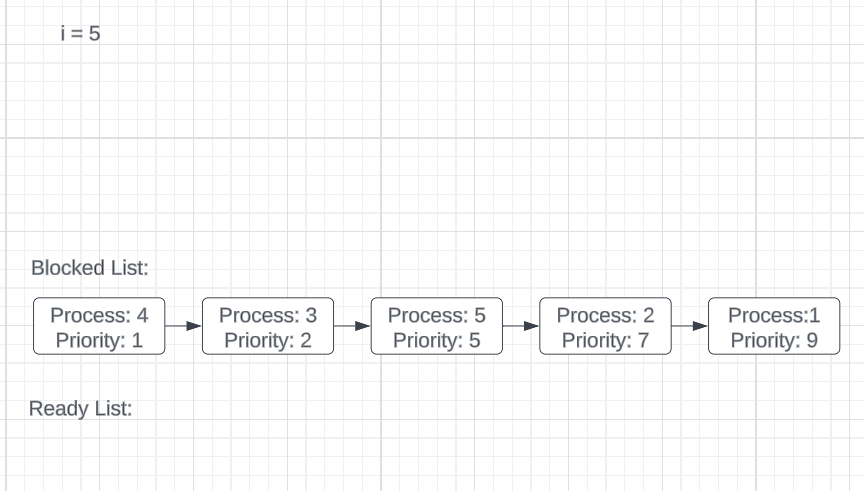


****

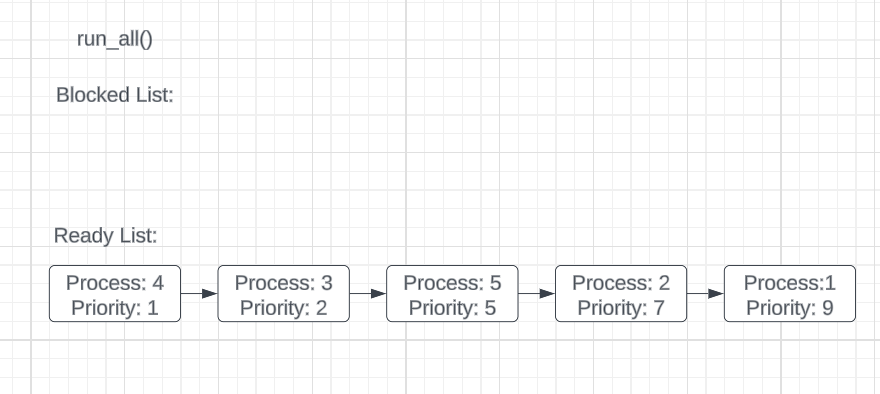




****



After all processes are created and added to blocked queue:

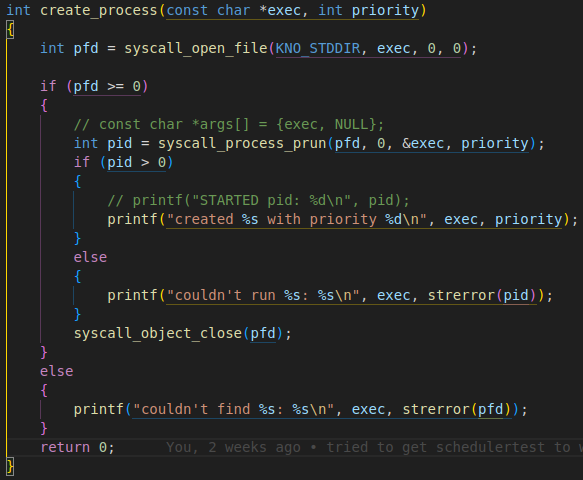


# 3) Implemented functions.

**3.1 schedulertest.c**

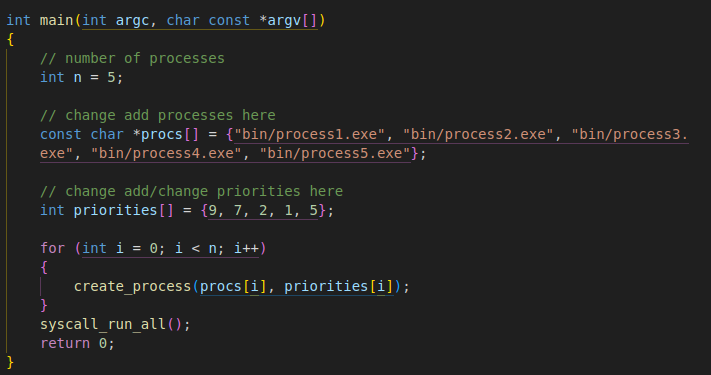
create\_process(const char \*exec, int priority), takes in a path to a process and a priority.

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:**

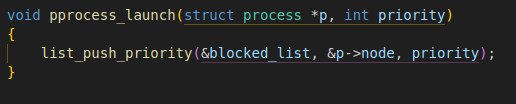
pprocess\_lauch() that uses the list\_push\_priority() function from list.h which takes in a:

destination list,

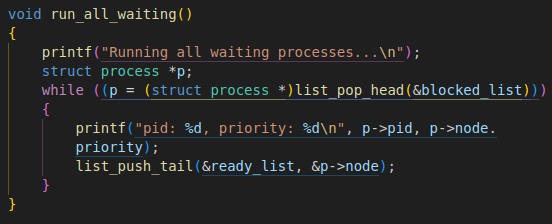
process’s listnode,

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();

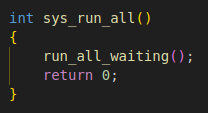
**3.5 sys\_call\_handler.c**

**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)

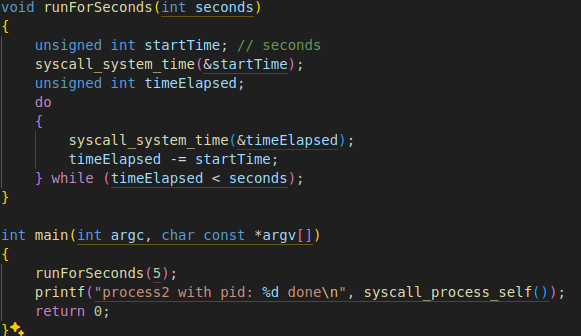
{

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

}

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

**3.7 process’s**

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.

# 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.c file

### **3.1.1 ‘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.

### **3.1.2 ‘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.

### **3.1.3 ‘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.

### **3.1.4 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.

3.1.5 ‘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)

### **3.1.6 ‘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

### **3.1.7 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)

### **3.1.8 ‘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.

### **3.1.9 ‘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).