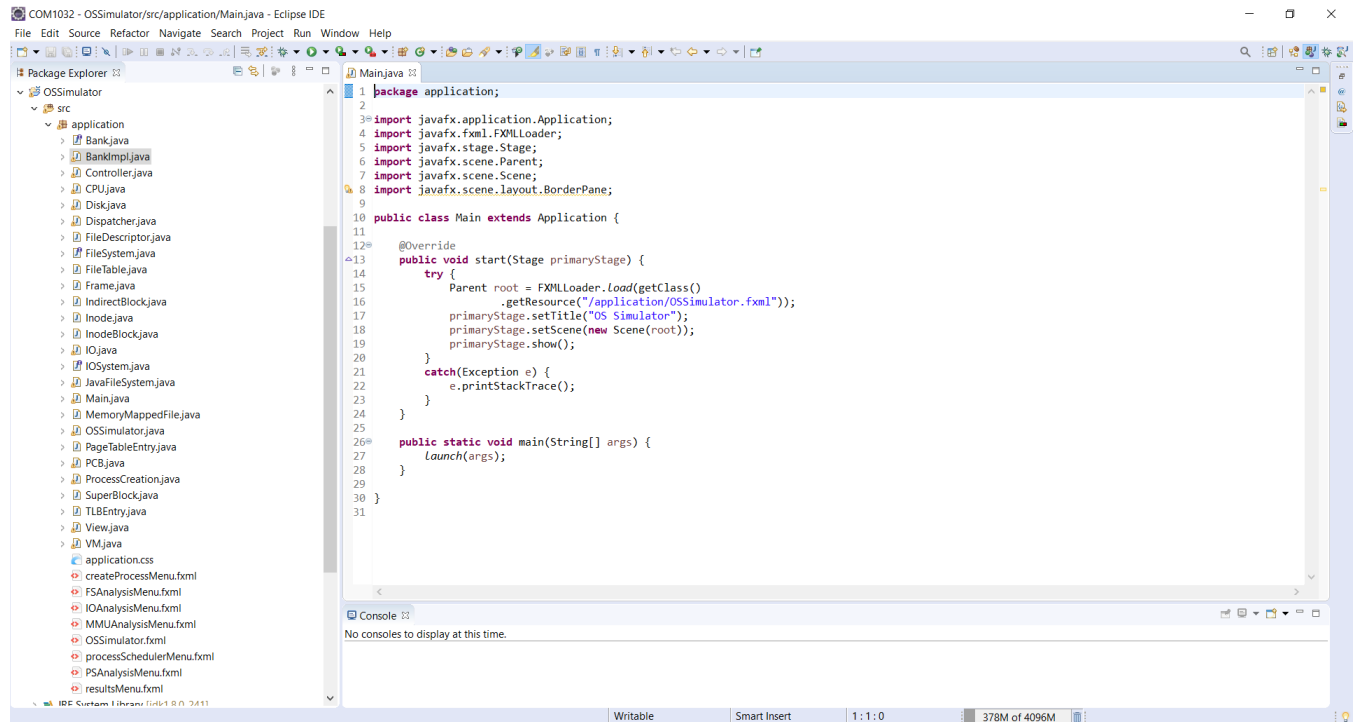


## OS simulator report

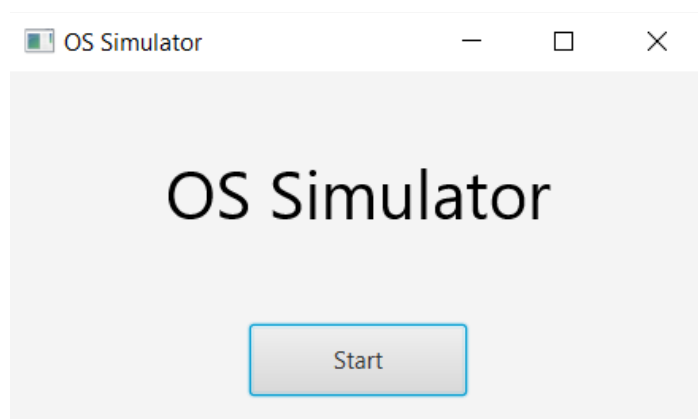
### Compiling and running

The operating system simulator has been written in Java 8. The name of the Java project is OSSimulator. The JavaFX library [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] has been used to create the GUI; therefore, JavaFX must be installed on the user's computer to run the OS simulator code.

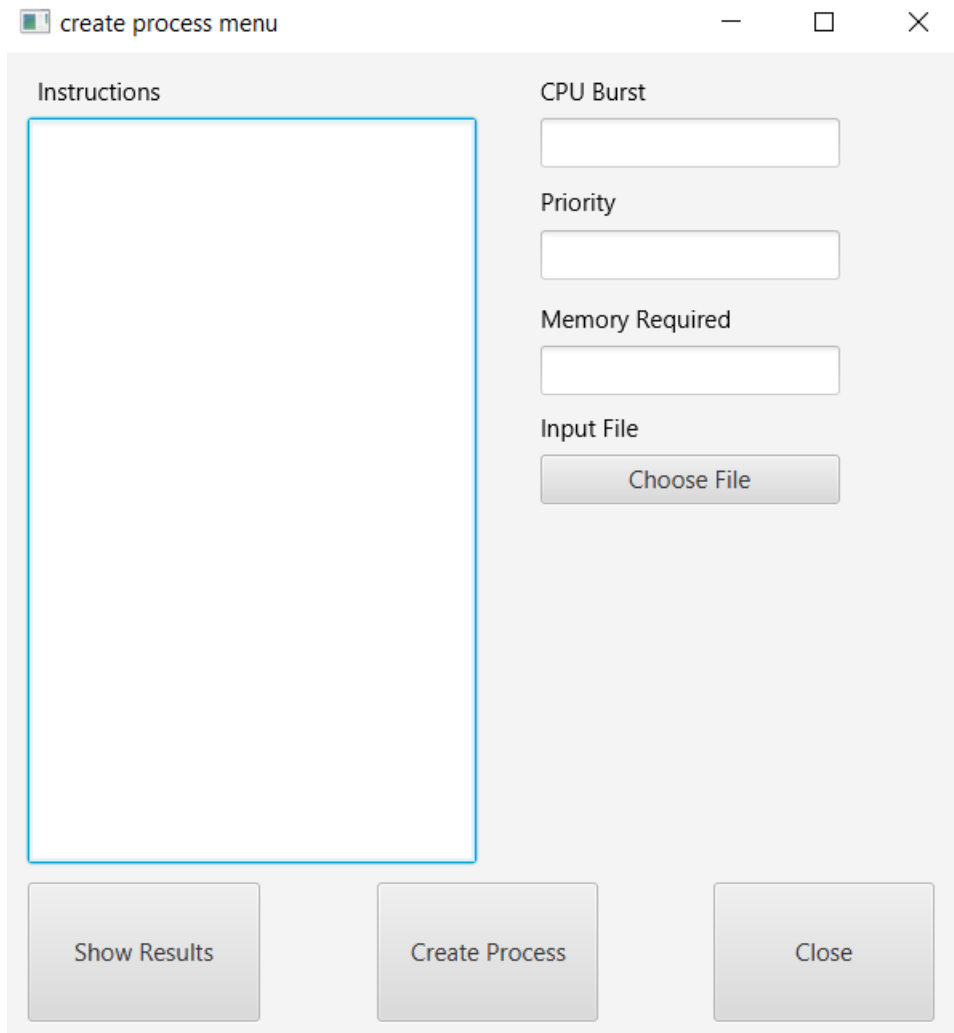
Run the Main.java class to run the OS simulator code. The file path of Main.java is src/application/Main.java. The following is the Main.java file open in the eclipse IDE:



When the Main class is run, the following window will appear:




Click on the Start button to run the OS Simulator. After the Start button has been clicked, the following window will appear:



The screenshot shows a window titled "create process menu". It features a large text area on the left labeled "Instructions". To the right of this area are four input fields: "CPU Burst", "Priority", "Memory Required", and "Input File". Below the "Input File" field is a button labeled "Choose File". At the bottom of the window, there are three buttons: "Show Results", "Create Process", and "Close".

To create processes, the instructions field, CPU Burst field and priority field will need to be filled in. The Memory Required field and Input File are optional. The CPU burst and priority is measured in milliseconds, the memory required is measured in bytes. Each process can include multiple instructions. When the instructions are input, they are separated by semicolons (;). For example, if the user wants to create a process, with a CPU burst of 50 and a priority of 5, which runs the print(1) instruction and then the print(2) instruction, the fields will be filled in the following way:

 create process menu



### Instructions

```
print(1);  
print(2);
```

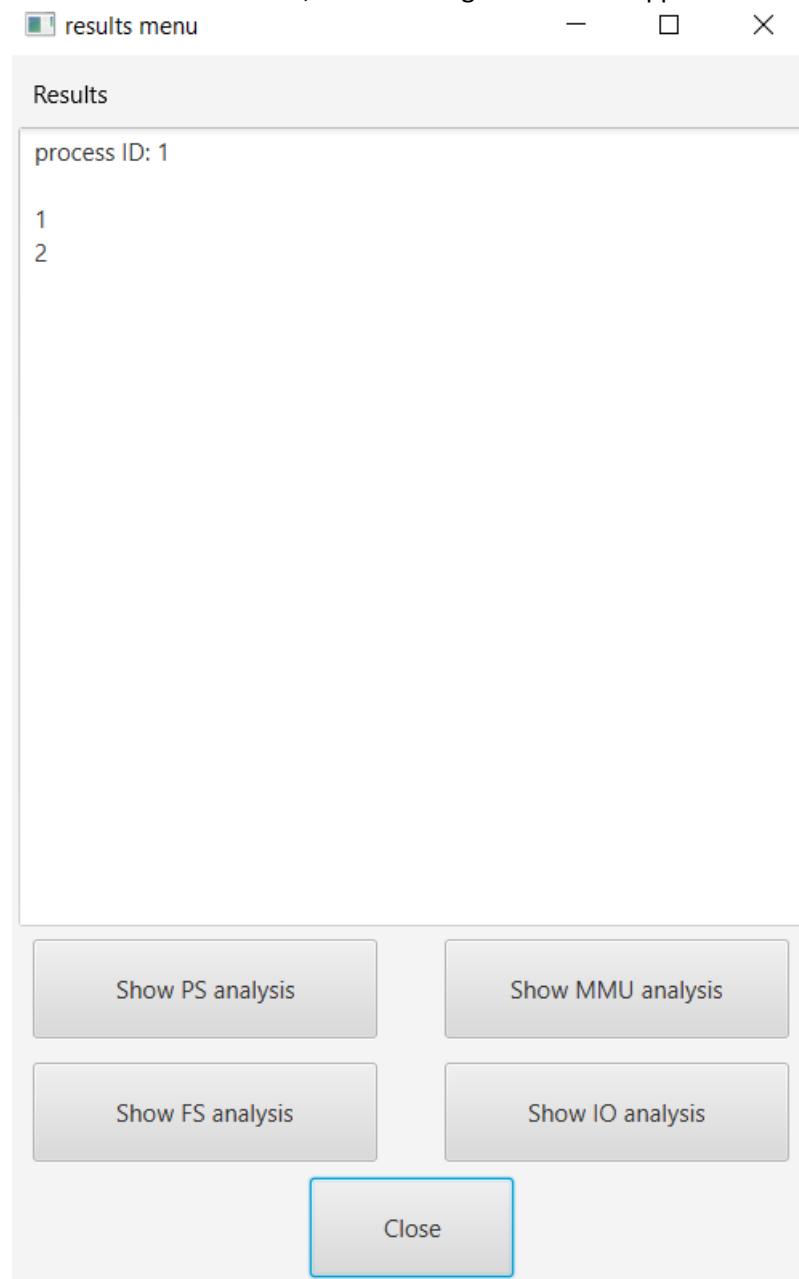
### CPU Burst

### Priority

### Memory Required

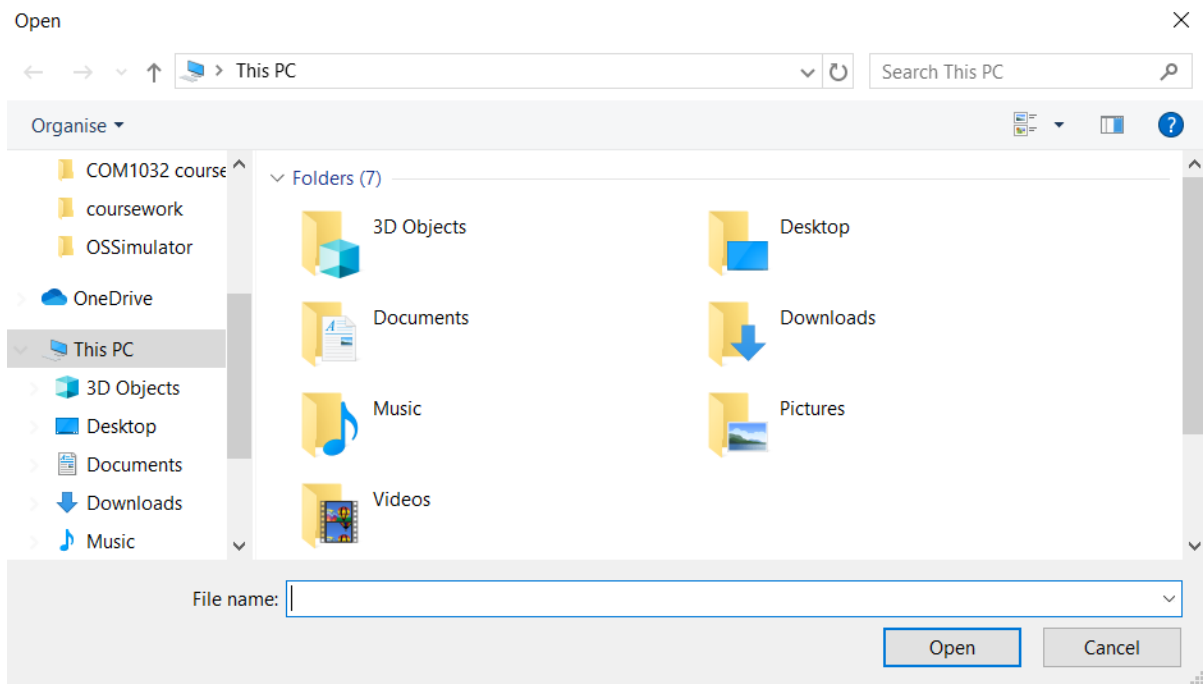
### Input File

Click on the Create Process button so that the OS simulator runs the process. Afterwards click on the Show Results Button, the following window will appear:



The Results field outputs the process ID of the process and the result of running the process. In the example, the process ID is 1. The print(1) instruction is run first and the print(2) instruction is run afterwards. The result of running the print(1) instruction is 1, therefore the Results field outputs 1. The result of running the print(2) instruction is 2, therefore the Results field outputs 2.

If the user wants to run an input file containing a list of processes, the user can click on the Choose File button. The following window will appear:



Choose the input file and click on the Open button. Afterwards the processes will run.

## User Manual

### Overview

The OS simulator contains four subsystems, namely process scheduler, MMU, file system and I/O system.

Before the OS simulator is run, you will need to configure a text file called ini.txt which contains the configuration parameters. The ini.txt file needs to be saved in the main project folder. The following configuration parameters are included in the ini.txt file:

1. Maximum number of processes in buffers containing processes
2. Process scheduling algorithm
3. Name of the Initial input file containing a list of processes
4. Time quantum (only included if scheduling algorithm equals RR or HPFSP)
5. Page table entries
6. Number of frames
7. Page size
8. TLB size
9. Page replacement algorithm
10. Block size
11. Number of blocks
12. Node size
13. Maximum number of files
14. Disk scheduling algorithm

The input file contains a list of processes, with each line representing a process. Each line contains the following information, a semicolon is input after each information:

- a. The instructions which the process contains, a semicolon is input after each instruction.
- b. The exit instruction.

- c. The CPU burst of the process.
- d. The priority of the process.
- e. The memory required by the process.

The exit instruction can optional be included at the end of a process when the GUI is used to create a process, however it is required in the input file. The exit instruction must not be included in the start of middle of a process otherwise the process will terminate before all the instructions have run.

The program will remove the contents of the input file once it is used, therefore a copy of the input file is required before the program is run to reuse the input file.

Unlike the GUI, the memory required is not optional. If the process does not require memory, then the memory required should be set the 0. The following is an example of an input file:

```
print(1);print(2);exit;50;5;0;
x = 5;print(x);exit;60;6;0;
```

The first process in the input file contains the instruction print(1) and print(2), has a CPU burst of 50, a priority of 5 and does not require memory. The second process in the input file contains the instructions x = 5 and print(x), has a CPU burst of 60, a priority of 6 and does not require memory.

The processes in the initial input file are run when the OS simulator starts running. The initial input file must be saved in the main project folder. If there is no initial input file, the line containing the name of the initial input file in the ini.txt file must be blank, and the program will create [12] and use an input file called processes.txt which is saved in the main project folder.

If the scheduling algorithm in the ini.txt file does not equal RR or HPFSP, then the time quantum line must not be included otherwise the program will throw an exception.

The following is an example of an ini.txt file:

```
5
FCFS

256
256
256
16
LRU
512
1000
64
21
SSTF
```

In the example, the maximum number of processes in buffers containing processes is 5. The process scheduling algorithm is FCFS. There is not initial input file. The number of page table entries is 256. The number of frames is 256. The page size is 256. The TLB size is 16. The page replacement algorithm is LRU. The block size is 512. The number of blocks is 1000. The node size is 64. The maximum number of files is 21. The disk scheduling algorithm is SSTF.

The OS simulator can run a mixture of instructions including simple JavaScript instructions, simple shell instructions such as `cd`, memory management instructions, file system instructions and I/O system instructions. The code from the lab2 solutions has been used to run simple shell instructions [13].

### Process scheduler

The process scheduler includes good scheduling for multitasking. The process scheduler and the process scheduling algorithms are explained in more detail in the technical manual.

Two test cases have been used to show the functionality of the process scheduler. The process scheduling algorithm used in these tests is FCFS. To run the test cases, the `ini.txt` file must be configured like the following:

```
5
FCFS

256
256
256
16
LRU
512
1000
64
21
SSTF
```

### Test case 1

To run the following test case, the input file `processSchedulerTest1.txt` must be used. The program removes the processes included in the file which means the copy of the file must be saved to rerun the test multiple times. To run the test case, click on the Choose File button and select the input file. The steps required to choose an input file has been explained in the compiling and running section of this report.

The input file contains the following processes:

```
x = 5;print(x);exit;20;2;0;
print(4*3);exit;10;1;0;
cd ..\;cd OSSimulator;print(10);exit;30;3;0;
history;exit;40;4;0;
invalid command;cd a;exit;50;5;0;
```

After the processes have run, click on the show results menu. The following window should appear:

results menu



## Results

process ID: 1

5

process ID: 2

12

process ID: 3

10

process ID: 4

0 x = 5

1 print(x)

2 print(4\*3)

3 cd ..\

4 cd OSSimulator

5 print(10)

process ID: 5

Show PS analysis

Show MMU analysis

Show FS analysis

Show IO analysis

Close



The following is the output of the processes:

```
process ID: 1
5
process ID: 2
12
process ID: 3
10
process ID: 4
0 x = 5
1 print(x)
2 print(4*3)
3 cd ..\
4 cd OSSimulator
5 print(10)
process ID: 5
'invalid' is not recognised as an internal or external command,
operative program or batch file
the directory C:\Users\armat\Documents\Ali\COM1032\OSSimulator.zip_expanded\OSSimulator\..\OSSimulator\ does not exist
```

The order which the processes have run is correct because the process scheduling algorithm is FCFS. Process 1 is run first, then process 2, then process 3, then process 4 and then process 5. The output of each of the processes is correct.

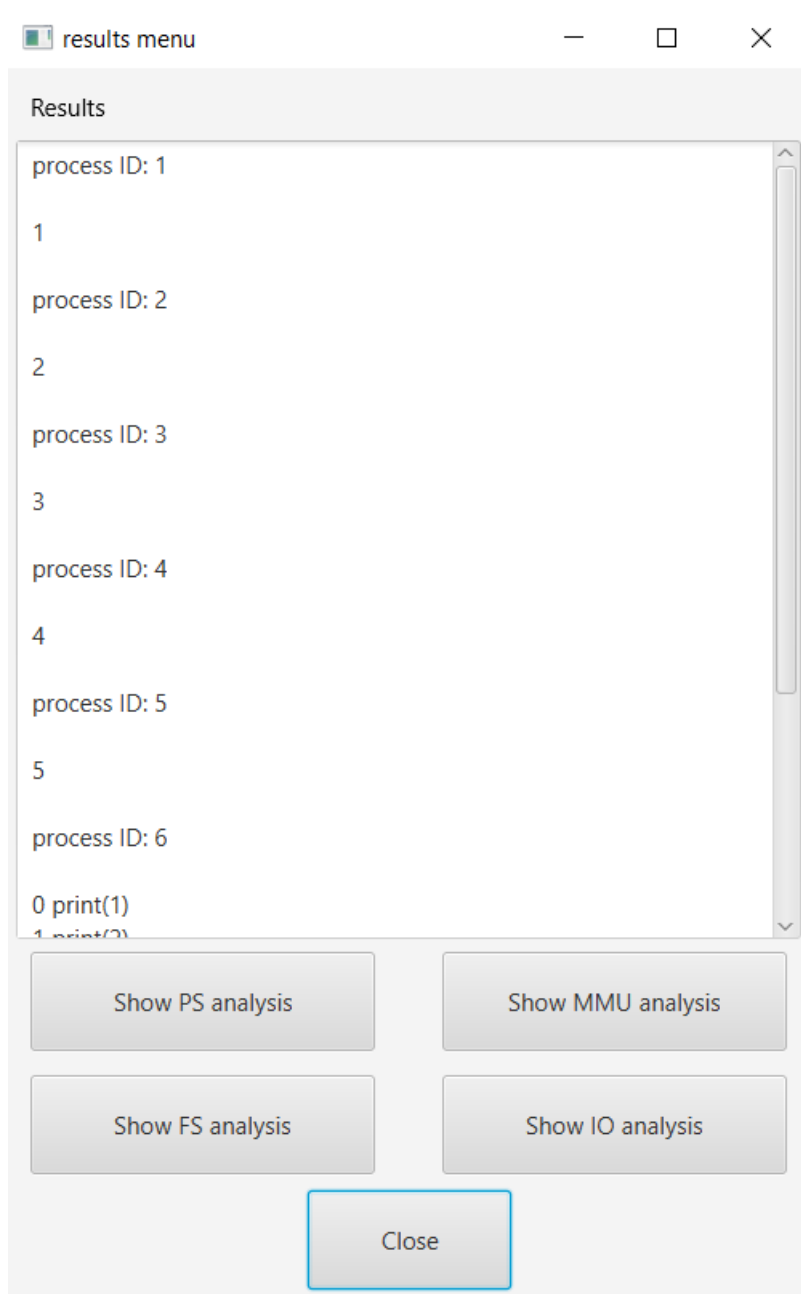
### Test case 2

To run the following test case, the input file processSchedulerTest2.txt must be used. The program removes the processes included in the file which means the copy of the file must be saved to rerun the test multiple times. To run the test case, click on the Choose File button and select the input file. The steps required to choose an input file has been explained in the compiling and running section of this report.

The input file contains the following processes:

```
print(1);exit;10;1;0;
print(2);exit;10;1;0;
print(3);exit;10;1;0;
print(4);exit;10;1;0;
print(5);exit;10;1;0;
history;exit;10;1;0;
!1;exit;10;1;0;
```

After the processes have run, click on the show results menu. The following window should appear:



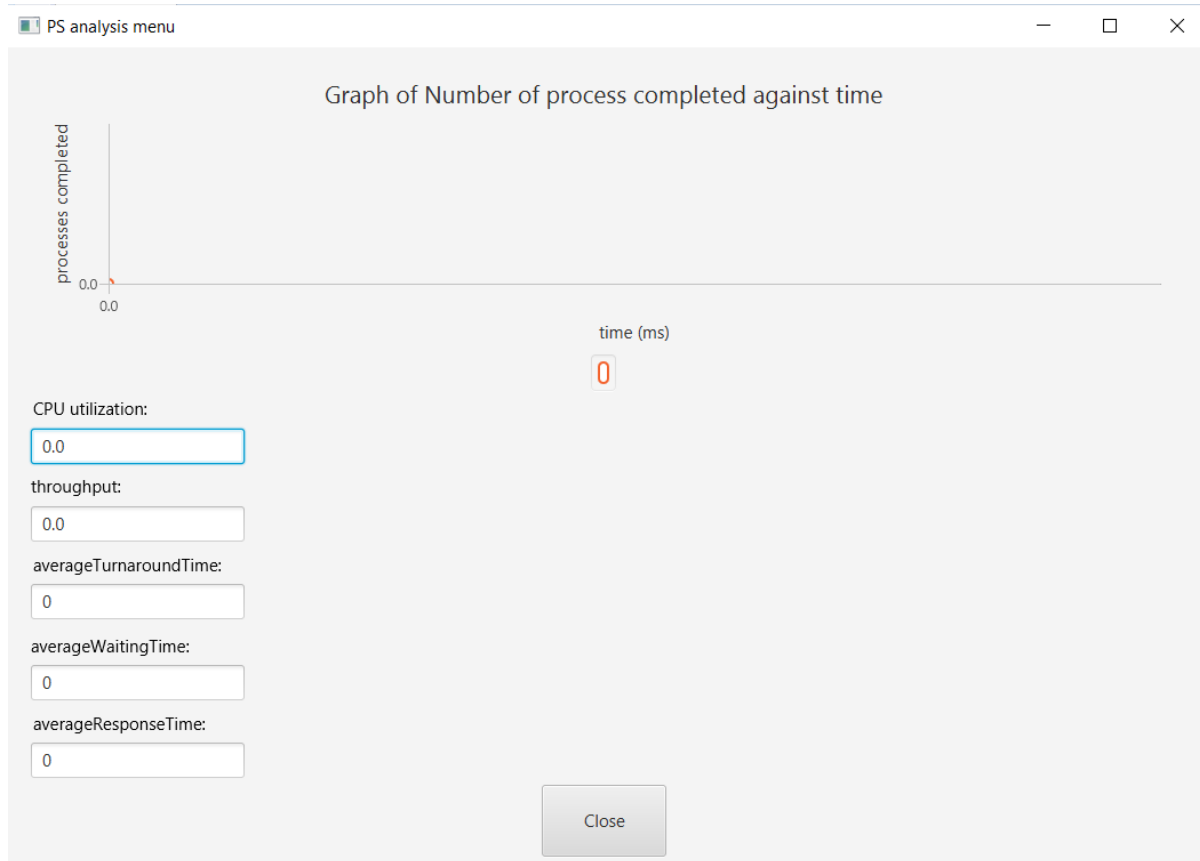
The following is the output of the processes:

```
process ID: 1
1
process ID: 2
2
process ID: 3
3
process ID: 4
4
process ID: 5
5
process ID: 6
0 print(1)
1 print(2)
2 print(3)
3 print(4)
4 print(5)
process ID: 7
2
```

The order which the processes have run is correct because the process scheduling algorithm is FCFS. Process 1 is run first, then process 2, then process 3, then process 4, then process 5, then process 6 and then process 7. The output of each of the processes is correct.

#### [Analysis Menu](#)

For statistics relating to the file system, click the ShowPSAnalysis button. The following window will appear:



The PS analysis menu shows a line chart of process completed against time, the CPU utilization, the throughput, the average turnaround time of the processes, the average waiting time of the processes, and the average response time of the processes. [14] In the above example no processes have run which is why the line chart does not contain data. The CPU utilization is 0 because no processes have run. The throughput, average turnaround time, average waiting time and average response time are 0 because no process has terminated.

## MMU

A good MMU, with new data structure for pages has been implemented. Page replacement algorithms have also been implemented and are explained in more detail in the technical manual. The function of the MMU is to map a virtual address to the corresponding physical address. The number of memory locations the RAM contains is equal to the size of the physical address space. The size of the virtual address space can be different to the size of the physical address space. Each memory location in the RAM stores 1 byte, which means the size of the physical address space and the size of the virtual address space is measured in bytes. The code from the lab 8 solutions has been used to implement the MMU. [13]

The MMU supports the following instructions:

- allocateMemory
- deallocateMemory

The CPU stores the virtual address of the next process to be stored in memory. When the OS simulator starts, the virtual address of the next process to be stored in memory is 0.

The allocateMemory instruction will store the process currently running in the RAM, and the instructions after the allocateMemory instruction will not run. The virtual address of the next

process to be stored in memory is equal to the virtual address of the process currently running. The MMU will map the virtual address of the process to the physical address of the process and the process will be stored in the memory location corresponding to the physical address of the process. Afterwards the virtual address of the next process to be stored in memory is incremented by 1. If the virtual address of the next process to be stored in memory is equal to the size of the virtual address space, the program will not store the process in memory and the next instruction is run.

The deallocateMemory instruction will remove the last process that was stored in the RAM from the RAM and run that process after the process that is currently running is complete. The virtual address of the next process to be stored in memory is decreased by 1. Afterwards, the MMU will map the virtual address of the process to be removed from the RAM to the physical address of the process, to remove the process from the memory location it is stored in. After the deallocateMemory instruction is run, the remaining instructions which the process contains is run. If the virtual address of the next process to be stored in memory is equal to 0, the program will not remove any process from the RAM because there are no processes currently stored in the RAM.

Every time a process calls the allocateMemory instruction, 1 byte is stored in the RAM. Therefore, the value of memory required for every process is equal to the number of times the allocateMemory instruction is called in the process.

Two test cases have been used to show the functionality of the MMU. To run the test cases, the ini.txt file must be configured like the following:

```
5
FCFS

256
256
256
16
LRU
512
1000
64
21
SSTF
```

### Test case 1

In the following test case, two processes must be created.

The first process contains the instructions:

- `x = 5`
- `y = 6`
- `allocateMemory`
- `print(y)`
- `print(x)`

The CPU burst of the first process is 50. The priority of the first process is 5. The memory required of the first process is 1.

Fill the fields in the create process menu in the following way:

create process menu

Instructions

```
x = 5;
y = 6;
allocateMemory;
print(y);
print(x);
```

CPU Burst

50

Priority

5

Memory Required

1

Input File

Choose File

Show Results

Create Process

Close

Click the Create Process button.

The second process contains the instruction:

- deallocateMemory

The CPU burst of the second process is 50. The priority of the second process is 5. The second process does not require memory.

Fill the fields in the create process menu in the following way:

create process menu

Instructions

deallocateMemory;

CPU Burst

50

Priority

5

Memory Required

Input File

Choose File

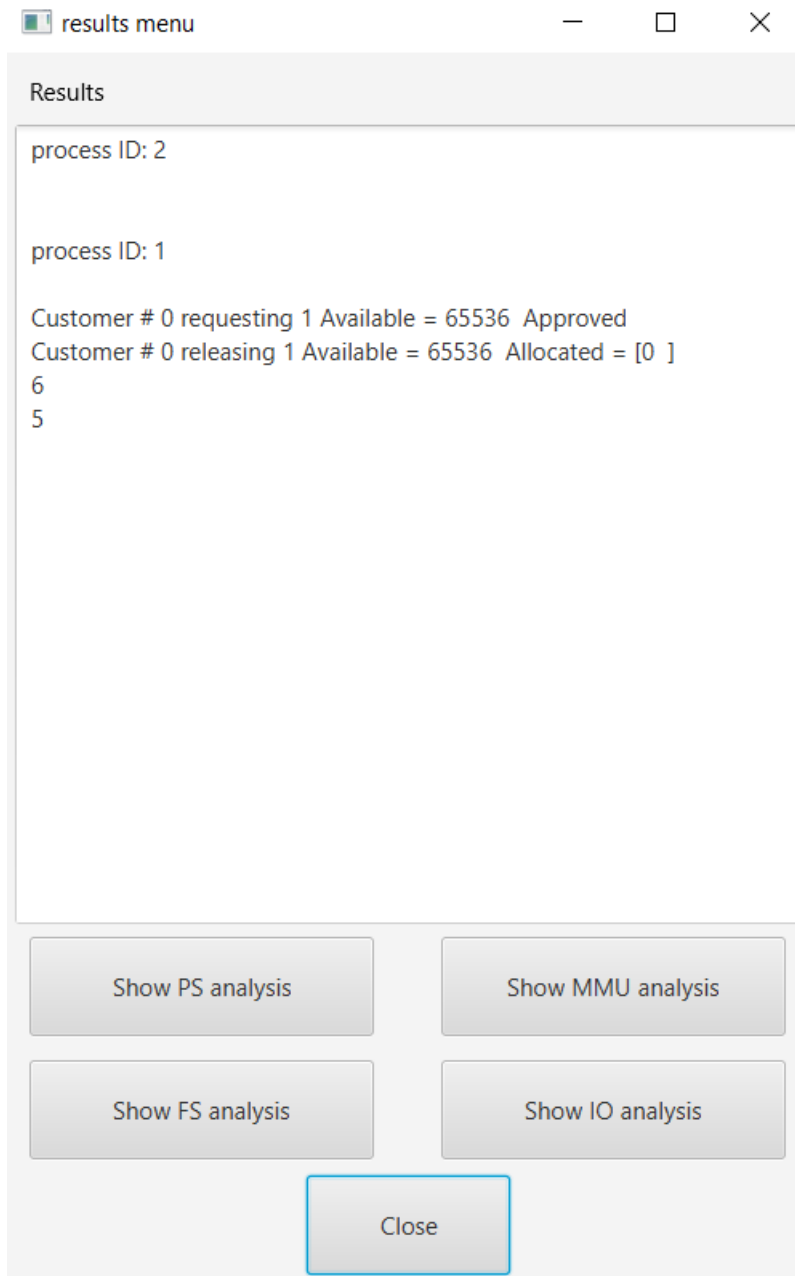
Show Results

Create Process

Close

Click the Create Process button.

Click the Show Results button, the following window will appear:



The order which the processes terminate is correct. Process 1 terminates after process 2. The output of each process is correct.

#### Test case 2

In the following test case, two processes must be created.

The first process contains the instructions:

- print(1)
- allocateMemory
- print(2)

The CPU burst of the first process is 50. The priority of the first process is 5. The memory required of the first process is 1.

Fill the fields in the create process menu in the following way:



create process menu

Instructions

```
print(1);
allocateMemory;
print(2);
```

CPU Burst

50

Priority

5

Memory Required

1

Input File

Choose File

Show Results

Create Process

Close

Click the Create Process button.

The second process contains the instructions:

- $x = 3$
- $y = 4$
- deallocateMemory
- $z = x * y$
- print(x)
- print(y)
- print(z)

The CPU burst of the second process is 50. The priority of the second process is 5. The second process does not require memory.

Fill the fields in the create process menu in the following way:

create process menu

Instructions

```
x = 3;  
y = 4;  
deallocateMemory;  
z = x*y;  
print(x);  
print(y);  
print(z);
```

CPU Burst

Priority

Memory Required

Input File

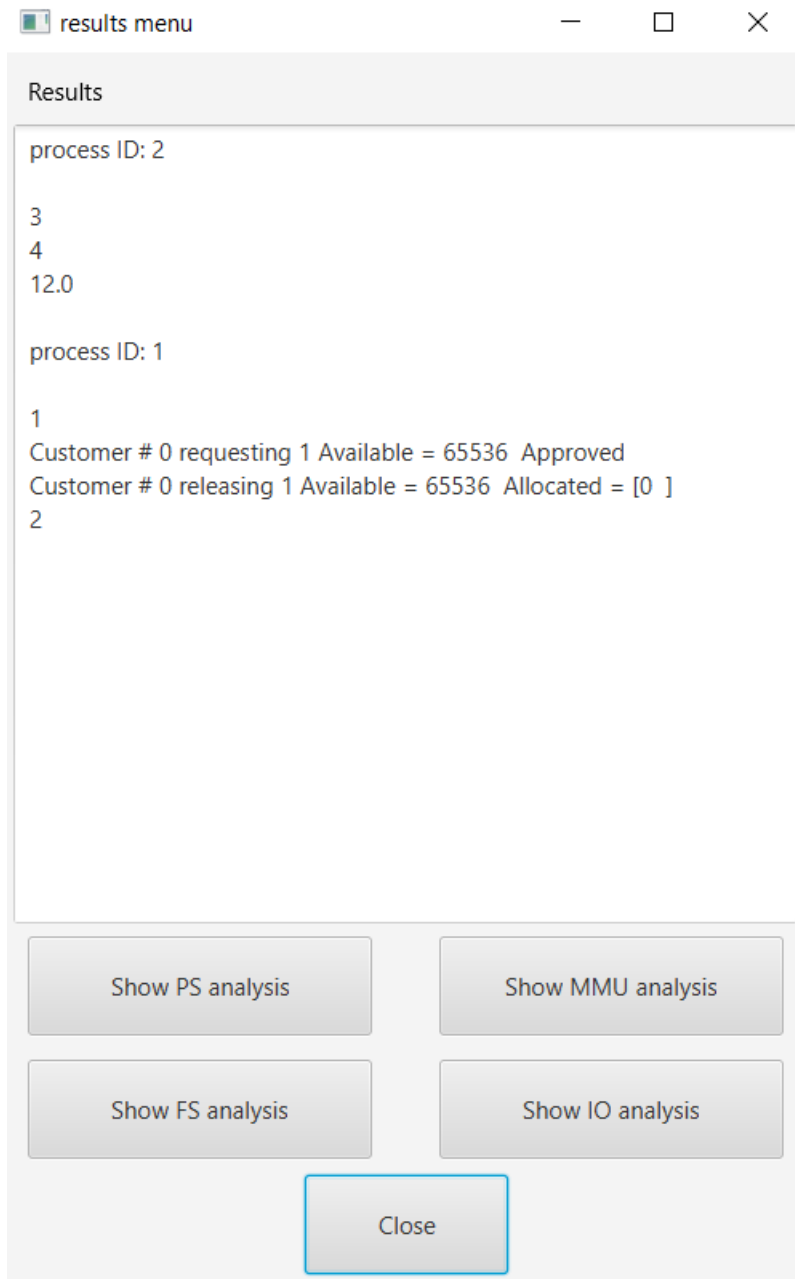
Show Results

Create Process

Close

Click the Create Process button.

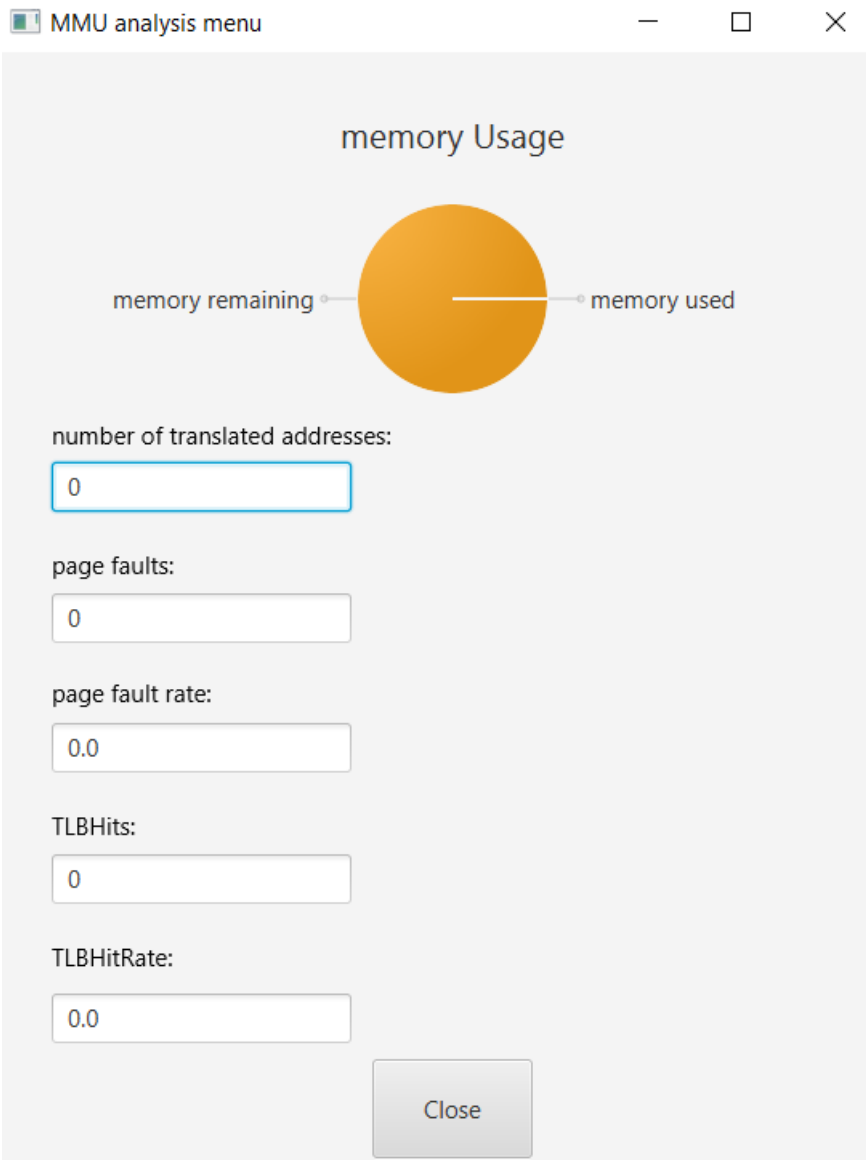
Click the Show Results button, the following window will appear:



The order which the processes terminate is correct. Process 1 terminates after process 2. The output of each process is correct.

#### Analysis Menu

For statistics relating to the MMU, click the ShowMMUAnalysis button. The following window will appear:



The MMU analysis menu shows a pie chart showing memory used and memory remaining, the number of translated addresses, the page faults, the page fault rate, the TLB hits, and the TLB hit rate. In the above example no processes have run which is why the memory remaining is the maximum value and the memory used is 0. In the above example the number of translated addresses, page faults, page fault rate, TLB hits, and TLB hit rate are 0.

### [File system](#)

The OS simulator is completely interfaced with the file system simulator in week 9. The file system simulator is week 9 has been updated to handle a queue of disk requests. The code from the lab 9 solutions has been used to implement the file system. [13]

The file system can run the following instructions:

- formatDisk size iSize
- shutdown
- create
- open inum
- inumber fd

- read fd size
- write fd pattern size
- seek fd offset whence
- close fd
- delete inum
- fileSystemVars

As explained in lab 9, “formatDisk initialises the disk to the state representing an empty file system: It fills in the superblock and links all the data blocks to the free list.”. The size argument is the “number of disk blocks in the file system” and the isize argument is “the number of inode blocks”. [13]

The shutdown instruction “closes all open files and shuts down the simulated disk”. [13]

The create instruction “creates a new empty file” and open “locates an existing file”. “Each method creates an integer in the range from 0 to 20 called a file descriptor (fd for short)”. “The file descriptor is an index into an array called a file descriptor table containing open files. Each array is associated with one open file and also contains a file pointer, which is initially set to 0.” The inum argument which open contains “is the inumber of an exiting file”. Inumber “returns the inumber of the file corresponding to an open file descriptor.” [13]

The read instruction reads “bytes starting at the current seek pointer”. The size argument is the number of bytes the instructions reads up to. “The return value is the number of bytes read”. If the number of bytes “between the current seek pointer and the end of the file” is less than the size argument, “only the remaining bytes are read”. “In particular, if the current seek pointer is greater than or equal to the file size”, the read instructions “returns 0, and the buffer is unmodified”. “The seek pointer is incremented by the number of bytes read.” [13]

The write instruction transfers “bytes from buffer to the file starting at the current seek pointer, advancing the seek pointer by that amount”. The number of bytes transferred is the size argument. [13]

The seek instruction “modifies the seek pointer” as explained in lab 9. If the seek instruction “results in a negative value for the seek pointer, the seek pointer is unchanged”, and the instruction returns “-1”. “Otherwise, the value returned is the new seek pointer, representing the distance in bytes from the start of the file.” [13]

The close instruction “writes the inode back to disk and frees the file table entry”. [13]

The delete instruction “frees the inode and all of the blocks of the file. An attempt to delete a file that is currently open results in an error”. [13]

The shutdown instruction “closes all open files, flushes all in-memory copies of disk structures out to disk, calls the stop method on the disk, and prints debugging or statistical information”. [13]

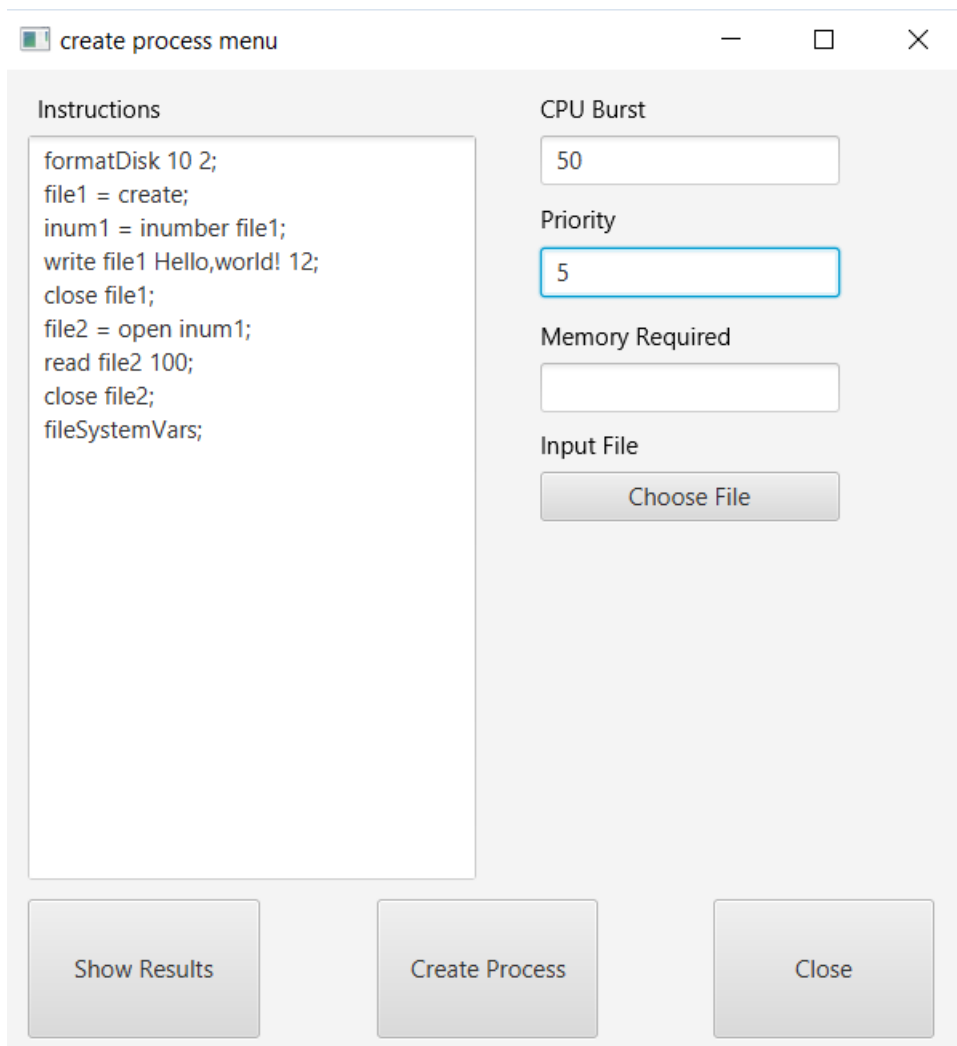
Like lab 9, variables can be created, and the result of file system instructions can be assigned to the variables. For example, the instruction file1 = create, creates a new file and assigns the file descriptor of the new file to file1. The filesystem vars instruction outputs all the variables used in the file system and their values. [13]

Two test cases have been used to show the functionality of the file system. To run the test cases, the ini.txt file must be configured like the following:

5  
FCFS  
  
256  
256  
256  
16  
LRU  
512  
1000  
64  
21  
SSTF

### Test case 1

To run the following test case, fill the fields in the create process menu in the following way:



create process menu

Instructions

```
formatDisk 10 2;  
file1 = create;  
inum1 = inumber file1;  
write file1 Hello,world! 12;  
close file1;  
file2 = open inum1;  
read file2 100;  
close file2;  
fileSystemVars;
```

CPU Burst

Priority

Memory Required

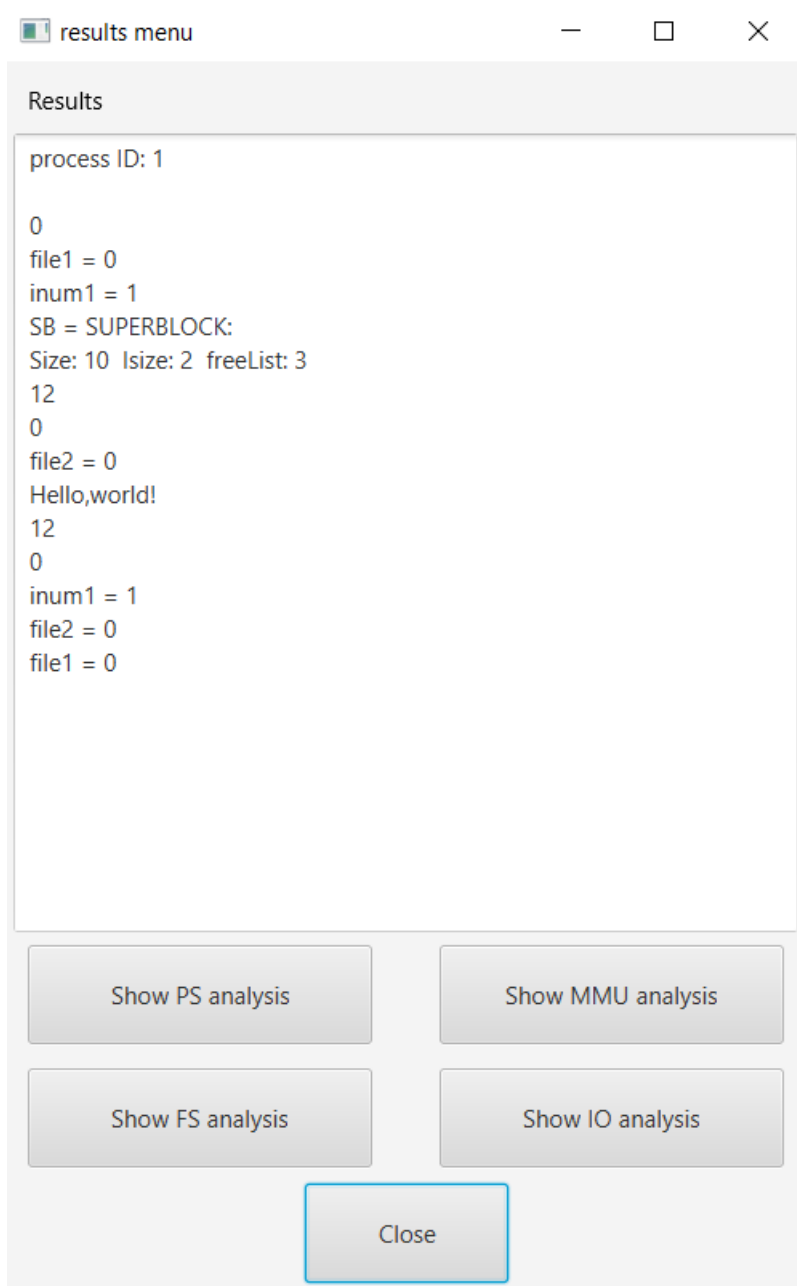
Input File

Choose File

Show Results   Create Process   Close

Afterwards the Create Process button must be clicked.

Click the Show Results button and the following window will appear:



The instructions output the expected result.

The result of the formatDisk instruction is 0 which means the instruction has run successfully.

The variable file1 has been assigned the value 0 which means the file descriptor of the new file created is 0.

Inum1 has been assigned the variable 1 which means the inumber of file1 is 1.

The write instruction returns the value 12 which means the string Hello,world! Has successfully been written to file1. The string Hello,world! Contains 12 bytes.

The close file1 instruction returned 0 which means file1 has successfully been closed.

The variable file2 has been assigned the value 0 which means the file descriptor of the file opened is 0.

The read instruction output the string Hello,world! And the number 12 which means the string Hello,world! Has been successfully read from file2.

The close file2 instruction returned 0 which means file2 has successfully been closed.

The fileSystemVars instruction output all the variables used in the file system which are inum1 = 1, file2 = 0 and file1 = 0.

### Test case 2

To run the following test case, add the following instructions in the instructions field in the create process menu:

```
formatDisk 100 10;
file1 = create;
inum1 = inumber file1;
write file1 HiThere 10000;
file2 = create;
inum2 = inumber file2;
seek file2 1000 0;
write file2 Yo 1000;
seek file1 -10 1;
read file1 50;
seek file2 -1010 2;
read file2 50;
close file2;
inumber file1;
close file1;
file3 = create;
write file3 Foo 1000;
seek file3 10 1;
write file3 Bar 1000;
seek file3 990 0;
read file3 30;
file1 = open inum1;
read file1 50;
inumber file2;
close file2;
delete inum2;
open inum2;
shutdown;
```

Add 50 to the CPU Burst field and add 5 to the priority field. The create process menu should therefore be filled in the following way:



create process menu

Instructions

```
formatDisk 100 10;
file1 = create;
inum1 = inumber file1;
write file1 HiThere 10000;
file2 = create;
inum2 = inumber file2;
seek file2 1000 0;
write file2 Yo 1000;
seek file1 -10 1;
read file1 50;
seek file2 -1010 2;
read file2 50;
close file2;
inumber file1;
close file1;
file3 = create;
write file3 Foo 1000;
seek file3 10 1;
write file3 Bar 1000;
seek file3 990 0;
read file3 30;
file1 = open inum1;
```

CPU Burst

50

Priority

5

Memory Required

Input File

Choose File

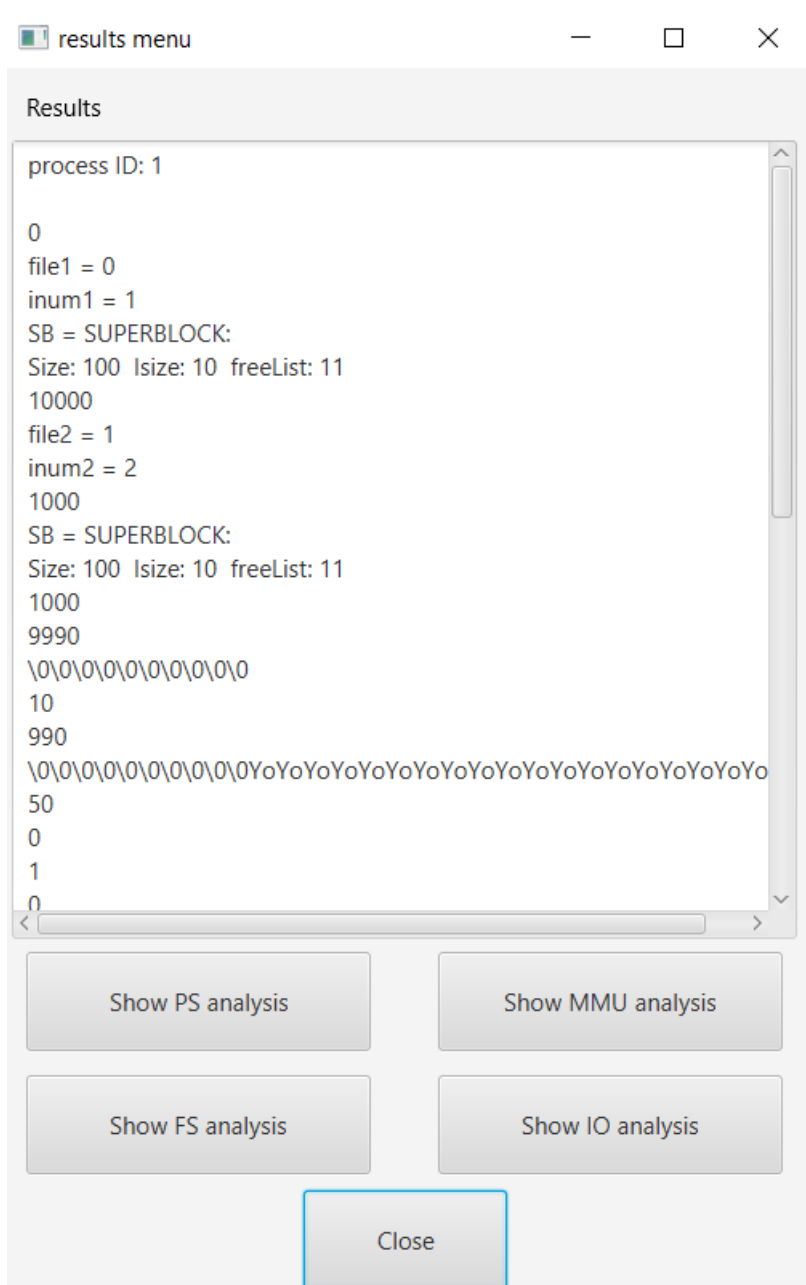
Show Results

Create Process

Close

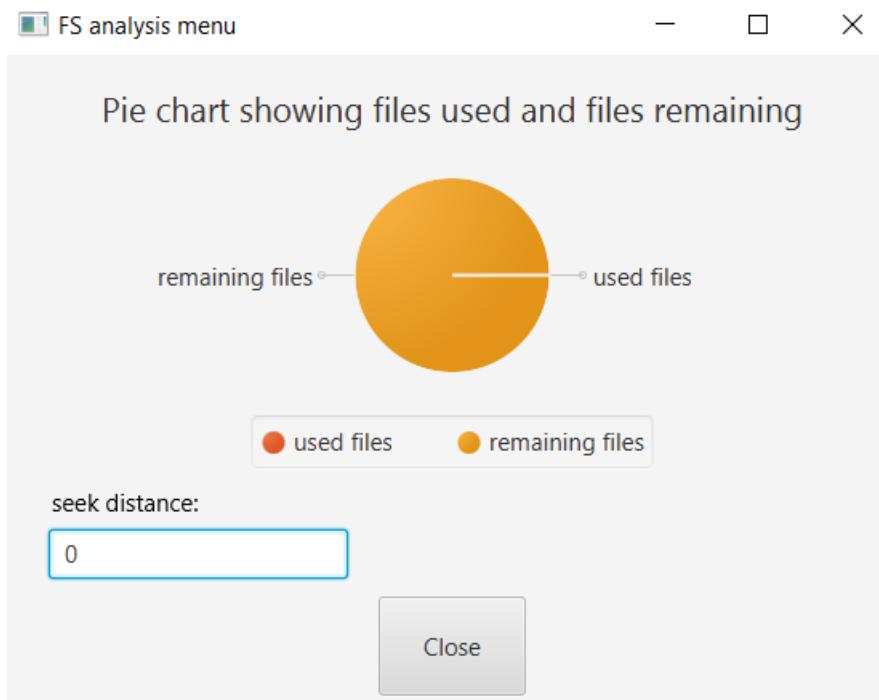
Click on the Create Process button.

Click on the Show Results button. The following window will appear:



The following is the result of running the process:

[illegible]



The FS analysis menu shows a pie chart showing files used and files remaining, and the seek distance. In the above example no processes have run which is why the remaining files is the maximum value and the used files is 0. The seek distance in the above example is 0.

### I/O system

An I/O manager has been implemented. The I/O manager has been properly integrated with to the other subsystems which have been implemented. The OS simulator contains one instruction set that is supporting all 4 of the implemented subsystems. The I/O system manages a list of peripherals and memory mapped files are buffers for specific peripherals, as explained in lab 11. The code from the lab 11 solutions has been used to implement the I/O system. [15]

The I/O system can run the following instructions:

- addDevice DeviceName DeviceBufferFileName
- input deviceName position size
- output deviceName position data1 data2 ...
- removeDevice deviceName
- IOVars

The addDevice instruction adds a new device to the list of peripherals. The DeviceName argument is the name of the device added. The DeviceBufferFileName argument is the name of the memory mapped file for the peripheral. [15]

The input instruction reads data from the memory mapped file of the device. The DeviceName argument is the name of the device. The position argument is the position to start reading from. The size argument is the size of the data to be read. [15]

The output instruction writes data to the memory mapped file of the device. The DeviceName argument is the name of the device. The position argument is the position to start writing to. The data arguments are the data to write to the file. [15]

The removeDevice instruction removes a device from the list of peripherals. The DeviceName argument is the name of the file to remove. [15]

Like lab 11, variables can be created, and the result of IO instructions can be assigned to the variables. For example, the instruction size = output keyboard 0 a, writes the data "a" to the memory mapped file and assigns the size of the data written to the variable size. The IO vars instruction outputs all the variables used in the file system and their values. [15]

Two test cases have been used to show the functionality of the I/O system. To run the test cases, the ini.txt file must be configured like the following:

```
5
FCFS

256
256
256
16
LRU
512
1000
64
21
SSTF
```

#### Test case 1

To run the following test case, add the following instructions in the instructions field in the create process menu:

```
addDevice keyboard keyboardBuffer;
output keyboard 0 this is the input for the keyboard to this system;
input keyboard 0 20;
```

Add 50 to the CPU Burst field and add 5 to the priority field. The create process menu should therefore be filled in the following way:

create process menu

Instructions

```
addDevice keyboard keyboardBuffer;  
output keyboard 0 this is the input for the  
input keyboard 0 20;
```

CPU Burst

50

Priority

5

Memory Required

Input File

Choose File

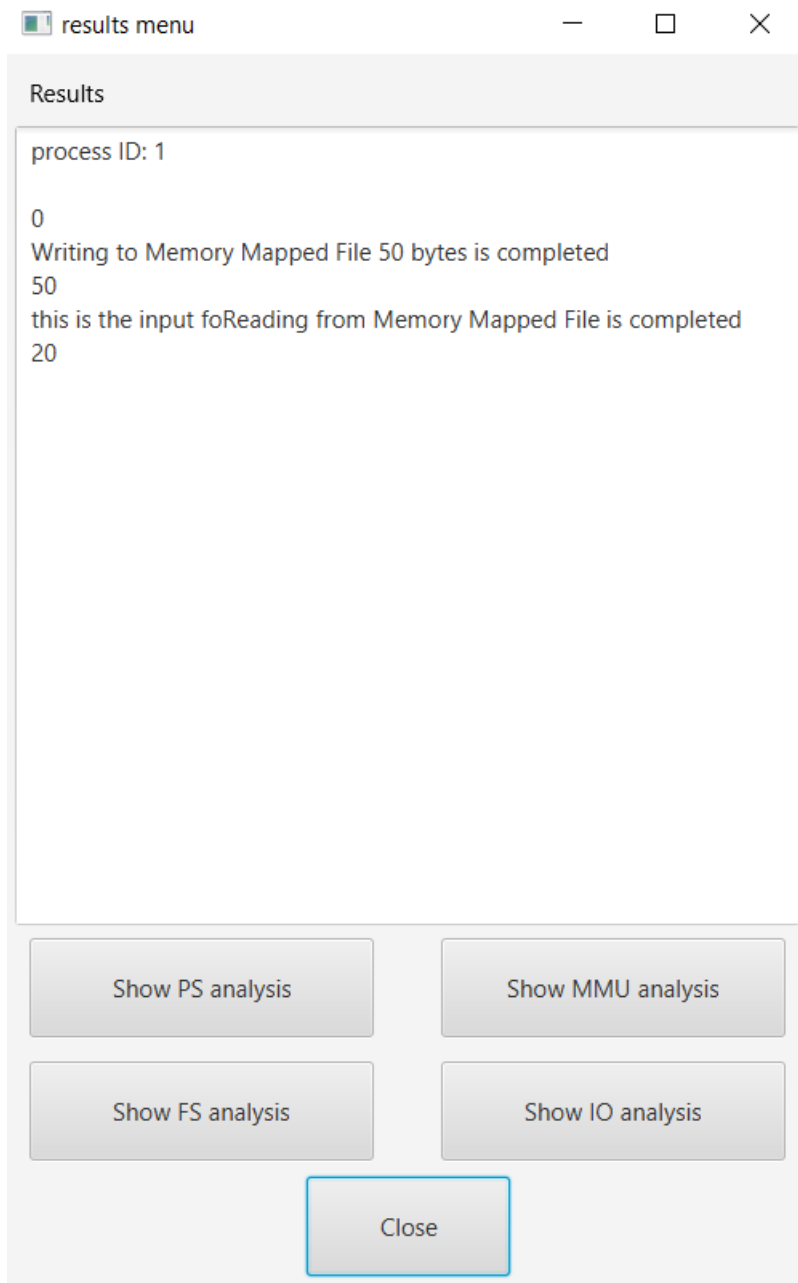
Show Results

Create Process

Close

Click on the Create Process button.

Click on the Show Results button. The following window will appear:



The instructions output the expected result.

The result of the addDevice instruction is 0 which means the device has successfully been added to the list of peripherals.

The result of the output instruction is 50, which means 50 bytes have successfully been written to the memory mapped file.

The result of the input instruction is the data that has been read from the memory mapped file, and the number 20. This means 20 bytes have successfully been read from the memory mapped file. The data that has been read from the memory mapped file is "this is the input fo".

[Test case 2](#)

To run the following test case, fill the fields in the create process menu in the following way:

create process menu

Instructions

```
addDevice keyboard keyboardBuffer;  
size = output keyboard 0 outputting;  
removeDevice keyboard;  
input keyboard 0 size;  
IOVars;
```

CPU Burst

50

Priority

5

Memory Required

Input File

Choose File

Show Results

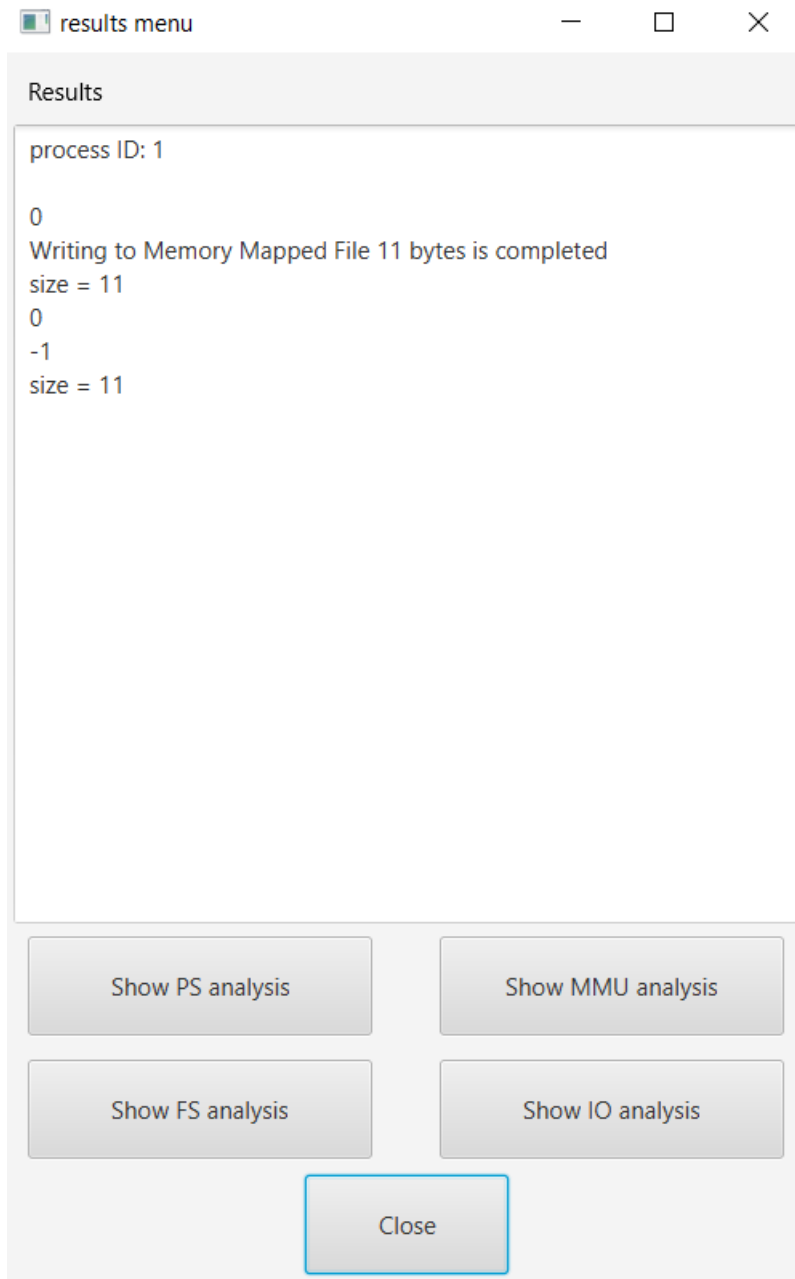
Create Process

Close

Click on the Create Process button.

Click on the Show Results button. The following window will appear:





The instructions output the expected result.

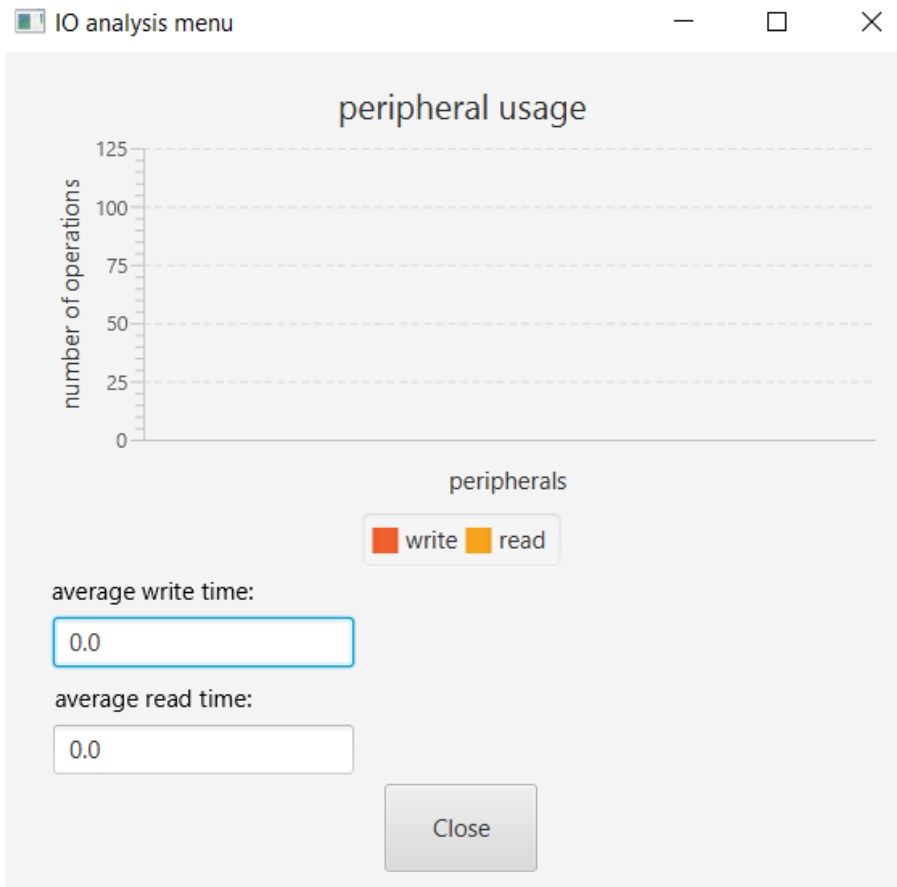
The result of the removeDevice instruction is 0, which means the device has successfully been removed from the list of peripherals.

The result of the input instruction is -1 which means bytes have not successfully been read from the memory mapped file. This is because the keyboard device has been removed from the list of peripherals before the input instruction is called.

The IOVars instruction output all the variables used in the I/O system which are size = 11.

#### Analysis Menu

For statistics relating to the IO, click the ShowIOAnalysis button. The following window will appear:



The IO analysis menu shows a pie chart showing the number of write operations and the number of read operations performed by each peripheral, the average write time of the operations, and the average read time of the operations. In the above example no processes have run which is why the bar chart does not contain data. In the above example the average write time is 0 because no write operations have been performed. The average read time is 0 because no read operations have been performed.

## [Technical Manual](#)

### [Overview](#)

The producer-consumer problem in lab 5 has been used to implement the OS simulator. [16]

The program contains six threads, namely ProcessCreation, Dispatcher, CPU, IO, VM and JavaFileSystem.

The program contains 6 buffers, namely readyBuffer, runningBuffer, terminatedBuffer, VMBuffer, fileSystemBuffer and IOBuffer. A class named PCB has been used to represent processes. The buffers store PCB objects.

The ProcessCreation thread is a:

- producer to the readyBuffer

The Dispatcher thread is a:

- consumer to the readyBuffer
- producer to the runningBuffer

The CPU thread is a:

- consumer to the runningBuffer
- producer to the readyBuffer
- producer to the terminatedBuffer
- producer to the VMBuffer
- producer to the fileSystemBuffer
- producer to the IOBuffer

The IO thread is a:

- consumer to the IOBuffer
- producer to the readyBuffer

The VM thread is a:

- consumer to the VMBuffer
- producer to the readyBuffer

The JavaFileSystem thread is a:

- consumer to the fileSystemBuffer
- producer to the readyBuffer

The above design has been inspired by the suggested design in lab 5. [16]

Details about how to start the OS simulator is explained in the compiling and running section. The OS simulator starts when the create process menu appears.

When the OS simulator starts, the configuration parameters are read from the ini.txt file. After the configuration parameters are read from the ini.txt file, the thread objects are created and the start() method in each thread object is called. The threads are always running, which is why each thread has a while(true) loop in its run() method.

In the create process menu, when the user has input the instructions, CPU burst, priority, and optionally the memory required for the process, and clicked on the Create Process button, the program will write the information associated with each process to the input file. The format of the input file is explained in the user manual. As explained in the user manual, the input file is either the file specified by the user in the ini.txt file, or the processes.txt file created by the program if the user did not specify an input file.

The role of the ProcessCreation thread is to read the input file and create processes. If the file contains a line representing a process, the thread will read the line and create a PCB object which stores the information which the line contains. The thread will add the PCB object to the readyBuffer and the line which represents the process is removed from the input file.

The role of the Dispatcher thread is to schedule the running of the processes. The Dispatcher thread uses the process scheduling algorithm specified in the ini.txt file to select a PCB object in the readyBuffer and removes that object from the readyBuffer. The thread will add the PCB object to the running buffer.

The role of the CPU thread is to simulate the running of processes. The CPU thread will remove a PCB object from the runningBuffer. The CPU will retrieve the instructions stored in the PCB object. For each instruction, the CPU will determine the command of the instruction and perform the

appropriate action based on the command. After an instruction has run, the CPU will run the next instruction. When an exit instruction is encountered, the CPU will stop running the process and add the PCB object to the terminatedBuffer. If the instruction allocateMemory is encountered, the CPU thread will add the PCB object to the VMBuffer so that the VM thread can store the PCB object in the RAM, and the CPU will stop running the process. If the instruction deallocateMemory is encountered, the CPU will add a copy of the PCB object to the VMBuffer, afterwards the CPU will run the remaining instructions in the process. If the current instruction in a process is a read instruction or a write instruction, the process is a disk request, therefore the PCB object will be added to the fileSystemBuffer, and the CPU will stop running the process. The fileSystemBuffer contains all the disk requests which the file system needs to handle. If the instruction encountered is a file system instruction which is not a disk request, the CPU will call the runTranslation method in the JavaFileSystem thread with the PCB object as an parameter to the method, and the CPU will stop running the process. If the instruction encountered is a I/O system instruction, the PCB object will be added to the IOBuffer and the CPU will stop running the process.

The role of the VM thread is to either store a process in the RAM or remove a process from the RAM. The code from the lab 8 solutions has been updated so that Frame objects also have an array of PCB objects as an attribute. The VM thread will remove a PCB object from the VMBuffer. If the current instruction in the process is allocateMemory, the thread will retrieve the virtual address of the process to store in the RAM. The MMU will map the virtual address of the process to store in the RAM to the physical address of the process to store in the RAM. The thread will store the process in the array of PCB objects which is an attribute of a Frame object, the Frame object which the process is stored in and the position of the PCB object in the array depends on the physical address of the PCB object. If the current instruction in the process is deallocateMemory, the thread will retrieve the virtual address of the process to remove from the RAM. The MMU will map the virtual address of the process to remove from the RAM to the physical address of the process to remove from the RAM. The thread will remove the process stored in the array of PCB objects which is an attribute of the Frame object, the Frame object which the process is stored in and the position of the PCB object in the array depends on the physical address of the PCB object. The PCB object which has been removed from the RAM is added to the readyBuffer.

The role of the JavaFileSystem thread is to run file system instructions. If the fileSystemBuffer contains a disk request, the JavaFileSystem thread will remove a PCB object from the fileSystemBuffer and run the current instruction in the PCB object. After the file system instruction has run, the thread will add the PCB object to the readyBuffer. If the run translation method has been called, the thread will run the current instruction of the PCB object which is the attribute of the run translation method, after the file system instruction has run, the thread will add the PCB object to the ready buffer.

The role of the IO thread is to run I/O system instructions. The IO thread will remove a PCB object from the IOBuffer and run the current instruction in the PCB object. After the I/O system instruction has run, the thread will add the PCB object to the readyBuffer.

### [Process scheduling algorithms](#)

The process scheduling algorithms implemented are FCFS, SJF, RR, HPFSP and HPFSN. These algorithms are used to select the next algorithm to put in the runningBuffer. No source code has been used to implement these algorithms.

## FCFS

The readyBuffer acts as a queue of PCB objects. Each PCB object created is added to the end of the readyBuffer. The PCB object created first is stored at the start of the readyBuffer, the PCB object created last is stored at the end of the readyBuffer.

When the readyBuffer contains more than one PCB object, the PCB object at the start of the readyBuffer is removed from the readyBuffer and added to the runningBuffer.

To run the following test case, the ini.txt file must be configured like the following:

```
10
FCFS

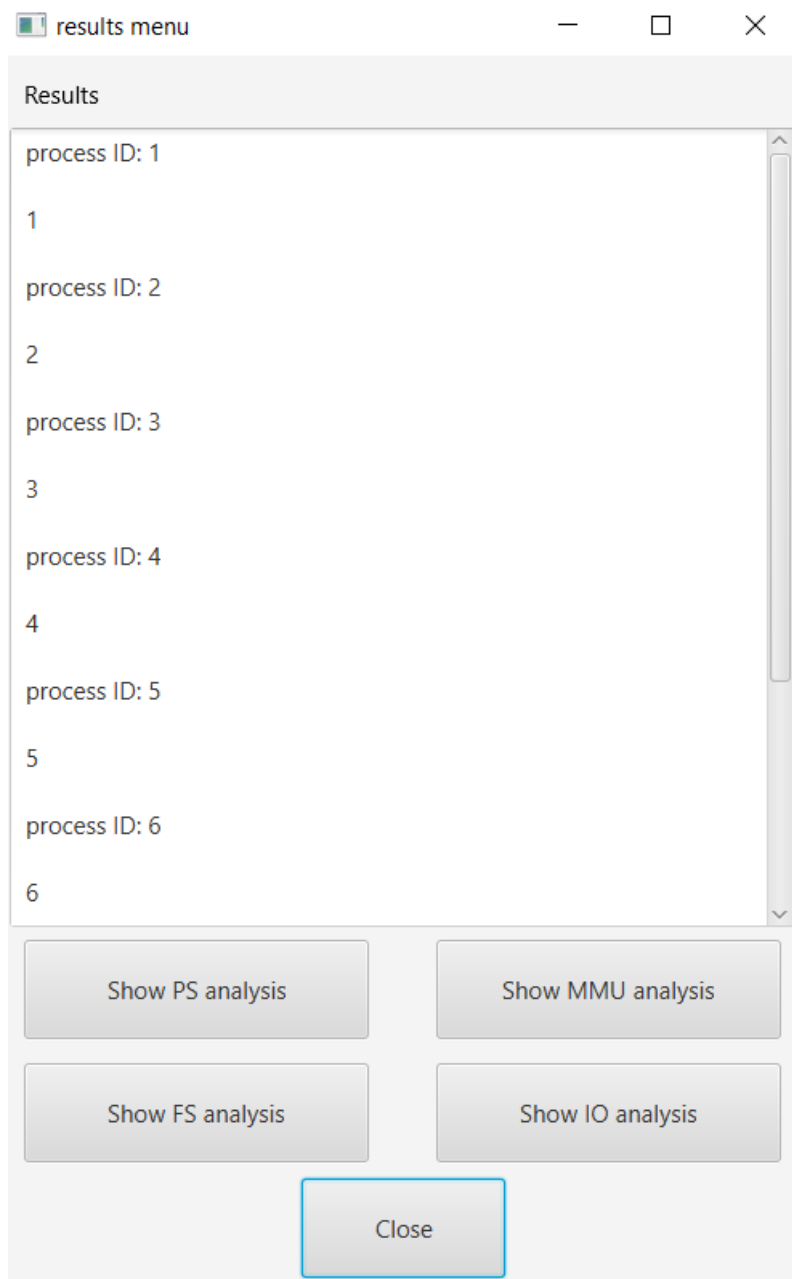
256
256
256
16
LRU
512
1000
64
21
SSTF
```

To run the following test case, the input file processSchedulerFCFSTest.txt must be used. The program removes the processes included in the file which means the copy of the file must be saved to rerun the test multiple times. To run the test case, click on the Choose File button and select the input file. The steps required to choose an input file has been explained in the compiling and running section of this report.

The input file contains the following processes:

```
print(1);exit;50;6;0;
print(2);exit;20;1;0;
print(3);exit;40;8;0;
print(4);exit;10;3;0;
print(5);exit;60;5;0;
print(6);exit;30;2;0;
print(7);exit;80;7;0;
print(8);exit;70;4;0;|
```

After the processes have run, click on the show results menu. The following window should appear:



The following is the output of the processes:

```
process ID: 1
1
process ID: 2
2
process ID: 3
3
process ID: 4
4
process ID: 5
5
process ID: 6
6
process ID: 7
7
process ID: 8
8
```

The order which the processes have run is correct because the process scheduling algorithm is FCFS. Process 1 is run first, then process 2, then process 3, then process 4, then process 5, then process 6, then process 7, and then process 8. The output of each of the processes is correct.

#### SJF

When the readyBuffer contains more than one PCB object, the SJF algorithm retrieves the CPU burst of each PCB object and the PCB object with the lowest CPU burst is removed from the readyBuffer and added to the runningBuffer.

To run the following test case, the ini.txt file must be configured like the following:

```
10
SJF

256
256
256
16
LRU
512
1000
64
21
SSTF
```

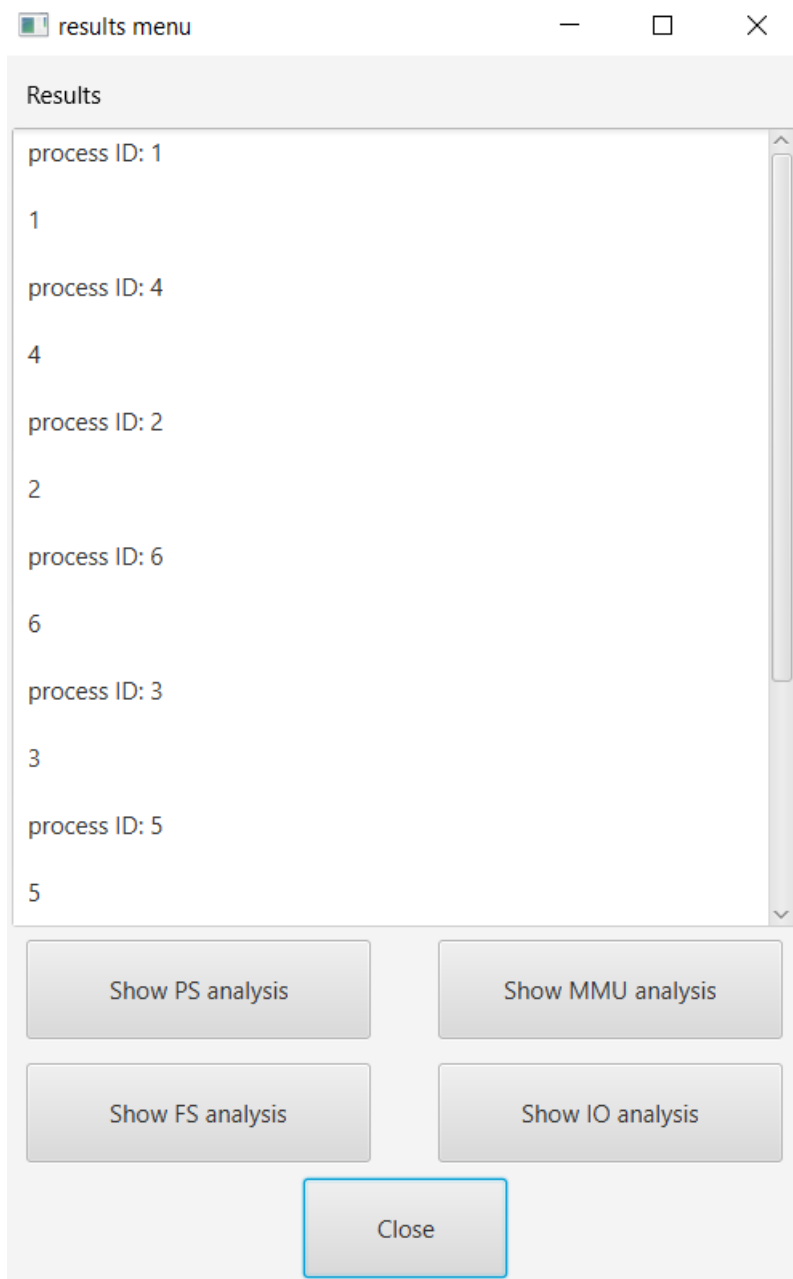
To run the following test case, the input file processSchedulerSJFTest.txt must be used. The program removes the processes included in the file which means the copy of the file must be saved to rerun the test multiple times. To run the test case, click on the Choose File button and select the input file. The steps required to choose an input file has been explained in the compiling and running section of this report.

The input file contains the following processes:

```
print(1);exit;50;6;0;
print(2);exit;20;1;0;
print(3);exit;40;8;0;
print(4);exit;10;3;0;
print(5);exit;60;5;0;
print(6);exit;30;2;0;
print(7);exit;80;7;0;
print(8);exit;70;4;0;
```

After the processes have run, click on the show results menu. The following window should appear:





The following is the output of the processes:

```
process ID: 1
1
process ID: 4
4
process ID: 2
2
process ID: 6
6
process ID: 3
3
process ID: 5
5
process ID: 8
8
process ID: 7
7
```

Process 1 is the first process to be added to the readyBuffer, which is why it is run first. Afterwards, the processes with the lowest CPU bursts are run first. Process 4 is run first, then process 2, then process 6, then process 3, then process 5, then process 8, and then process 7. The output of each of the processes is correct.

#### HPFSN

When the readyBuffer contains more than one PCB object, the HPFSN algorithm retrieves the priority of each PCB object and the PCB object with the highest priority is removed from the readyBuffer and added to the runningBuffer.

To run the following test case, the ini.txt file must be configured like the following:

```
10
HPFSN

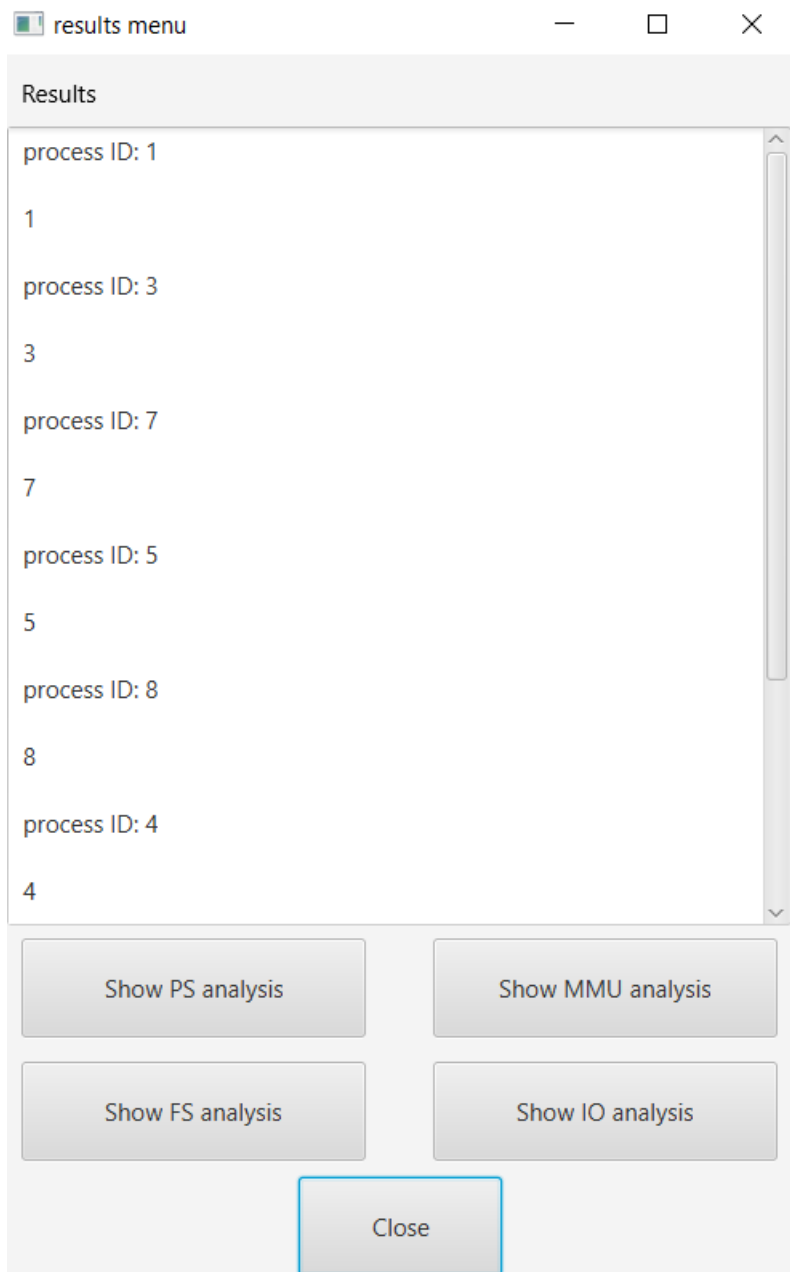
256
256
256
16
LRU
512
1000
64
21
SSTF
```

To run the following test case, the input file processSchedulerHPFSNTest.txt must be used. The program removes the processes included in the file which means the copy of the file must be saved to rerun the test multiple times. To run the test case, click on the Choose File button and select the input file. The steps required to choose an input file has been explained in the compiling and running section of this report.

The input file contains the following processes:

```
print(1);exit;50;6;0;
print(2);exit;20;1;0;
print(3);exit;40;8;0;
print(4);exit;10;3;0;
print(5);exit;60;5;0;
print(6);exit;30;2;0;
print(7);exit;80;7;0;
print(8);exit;70;4;0;
```

After the processes have run, click on the show results menu. The following window should appear:



The following is the output of the processes:

```
process ID: 1
1
process ID: 3
3
process ID: 7
7
process ID: 5
5
process ID: 8
8
process ID: 4
4
process ID: 6
6
process ID: 2
2
```

The processes are run in the correct order. The output of each of the processes is correct.

#### RR

After each instruction is run, if the time taken to run the process is higher than the time quantum, the process stops running and is added to the ready queue.

To run the following test case, the ini.txt file must be configured like the following:

```
10
RR
1

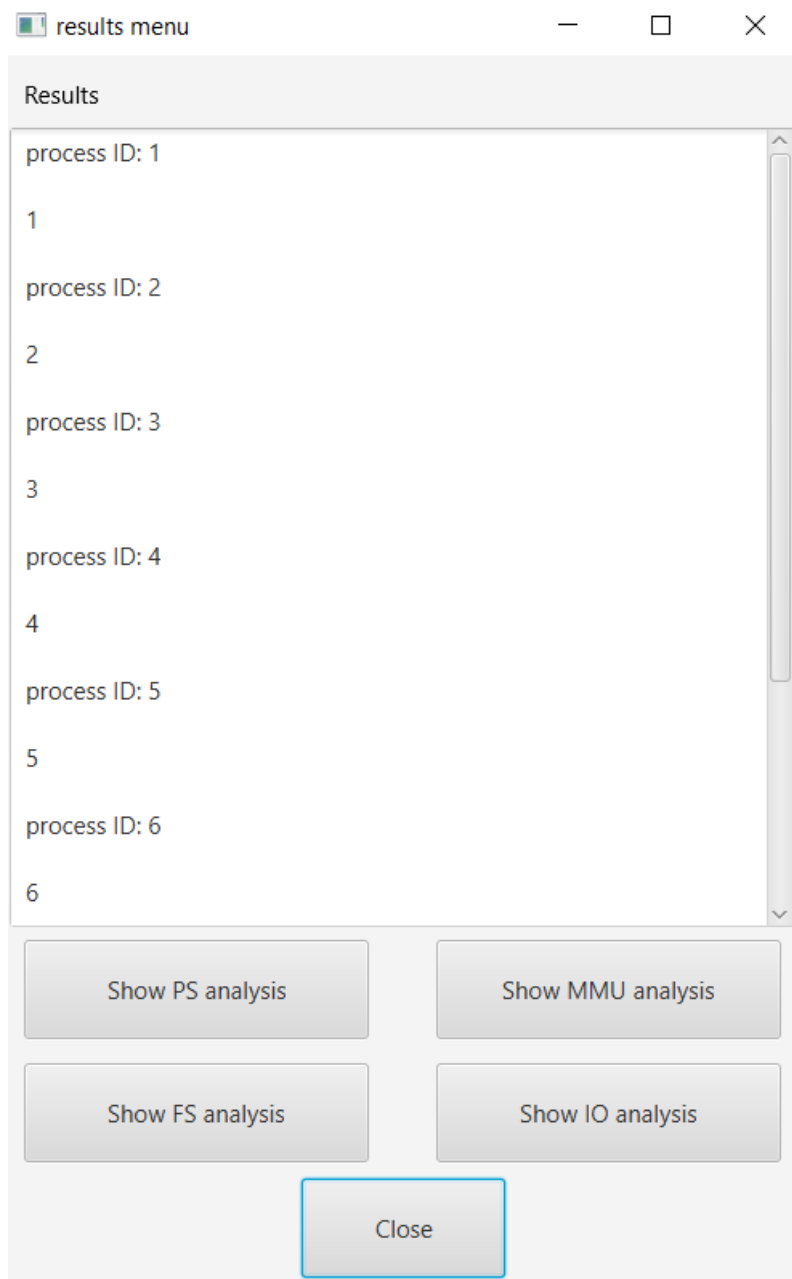
256
256
256
16
LRU
512
1000
64
21
SSTF
```

To run the following test case, the input file processSchedulerRRTest.txt must be used. The program removes the processes included in the file which means the copy of the file must be saved to rerun the test multiple times. To run the test case, click on the Choose File button and select the input file. The steps required to choose an input file has been explained in the compiling and running section of this report.

The input file contains the following processes:

```
print(1);exit;50;6;0;
print(2);exit;20;1;0;
print(3);exit;40;8;0;
print(4);exit;10;3;0;
print(5);exit;60;5;0;
print(6);exit;30;2;0;
print(7);exit;80;7;0;
print(8);exit;70;4;0;
```

After the processes have run, click on the show results menu. The following window should appear:



The following is the output of the processes:

```
process ID: 1
1
process ID: 2
2
process ID: 3
3
process ID: 4
4
process ID: 5
5
process ID: 6
6
process ID: 7
7
process ID: 8
8
```

The order which the processes have run is correct because the process scheduling algorithm is RR. Process 1 is run first, then process 2, then process 3, then process 4, then process 5, then process 6, then process 7, and then process 8. The output of each of the processes is correct.

#### HPFSP

After each instruction is run, if the time taken to run the process is higher than the time quantum, the process stops running and is added to the ready queue.

To run the following test case, the ini.txt file must be configured like the following:

```
10
HPFSP
1

256
256
256
16
LRU
512
1000
64
21
SSTF
```

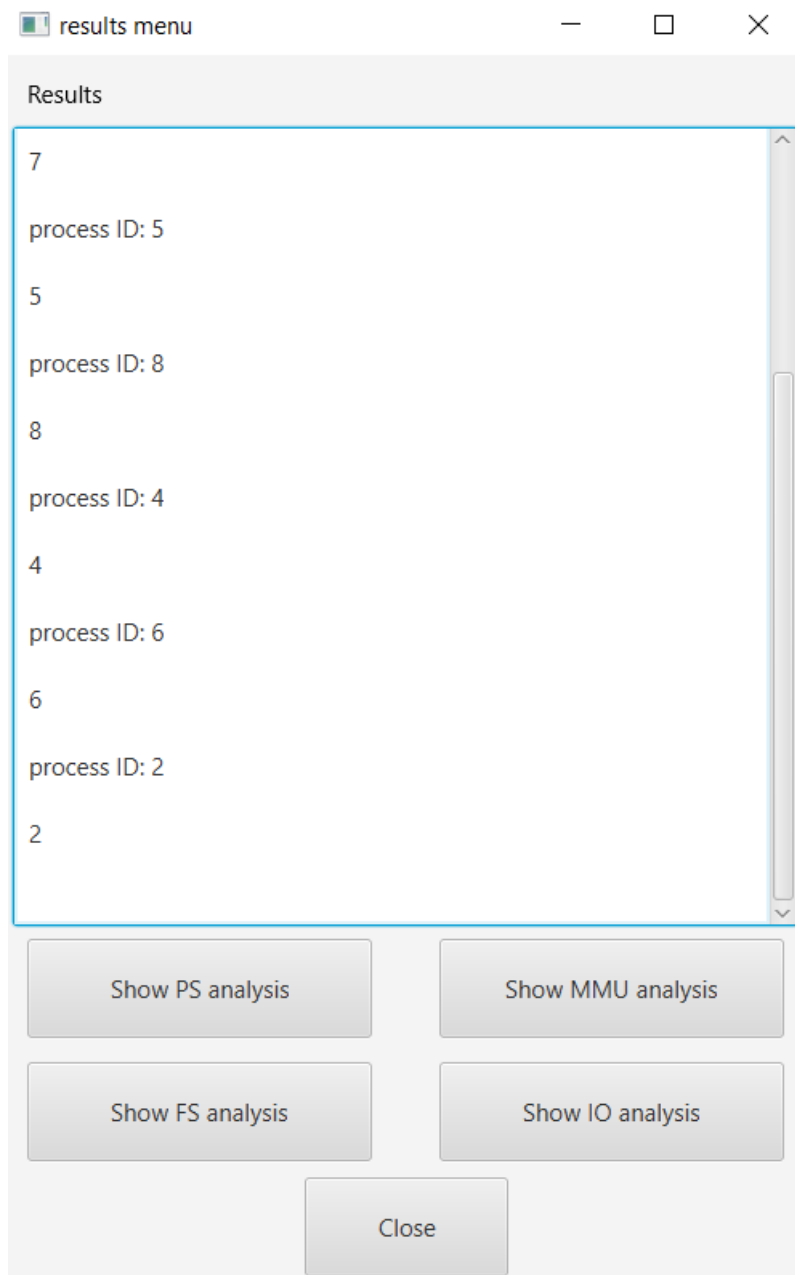


To run the following test case, the input file processSchedulerHPFSPTest.txt must be used. The program removes the processes included in the file which means the copy of the file must be saved to rerun the test multiple times. To run the test case, click on the Choose File button and select the input file. The steps required to choose an input file has been explained in the compiling and running section of this report.

The input file contains the following processes:

```
print(1);exit;50;6;0;  
print(2);exit;20;1;0;  
print(3);exit;40;8;0;  
print(4);exit;10;3;0;  
print(5);exit;60;5;0;  
print(6);exit;30;2;0;  
print(7);exit;80;7;0;  
print(8);exit;70;4;0;
```

After the processes have run, click on the show results menu. The following window should appear:



The following is the output of the processes:

```
process ID: 1
1
process ID: 3
3
process ID: 7
7
process ID: 5
5
process ID: 8
8
process ID: 4
4
process ID: 6
6
process ID: 2
2
```

The order which the processes have run is correct because the process scheduling algorithm is HPFSP. The output of each of the processes is correct.

#### Comparison

Dynamic priority scheduling and MLFQ prevent starvation but are harder to implement.

#### Page Replacement Algorithms

The page replacement algorithms which have been implemented are FIFO and LRU. If the size of the virtual address space is higher than the size of the physical address space and the virtual address of the process being stored in the RAM is equal to or higher than the size of the physical address space, a page replacement algorithm is used to select a victim frame and a new page is brought into the frame. In the following tests, the size of the RAM will be 1 and the page replacement algorithms will be used to store processes in the RAM. The same algorithms have been used to update the TLB. No source code has been used to write the page replacement algorithms.

#### FIFO

The VM thread stores an array of integers named FIFOQueue as an attribute. The integers are the page numbers of pages which have been used. The first integer in the array is removed and the victim frame corresponding to the page, which has the page number which has been removed, is selected.

To run the following test case, the ini.txt file must be configured like the following:

```
5
FCFS

256
1
1
16
FIFO
512
1000
64
21
SSTF
```

To run the following test case, create 8 processes, the information associated with each process is described below:

Process 1:

Instructions:

- print(1);
- allocateMemory;

CPU Burst : 50

Priority: 5

Memory required: 1

Process 2:

Instructions:

- print(2);
- allocateMemory;

CPU Burst : 50

Priority: 5

Memory required: 1

Process 3:

Instructions:

- print(3);
- allocateMemory;

CPU Burst : 50

Priority: 5

Memory required: 1

Process 4:

Instructions:

- print(4);
- allocateMemory;

CPU Burst : 50

Priority: 5

Memory required: 1

Process 5:

Instructions:

- deallocateMemory;

CPU Burst : 50

Priority: 5

Memory required: 0

Process 6:

Instructions:

- deallocateMemory;

CPU Burst : 50

Priority: 5

Memory required: 0

Process 7:

Instructions:

- deallocateMemory;

CPU Burst : 50

Priority: 5

Memory required: 0

Process 8:

Instructions:

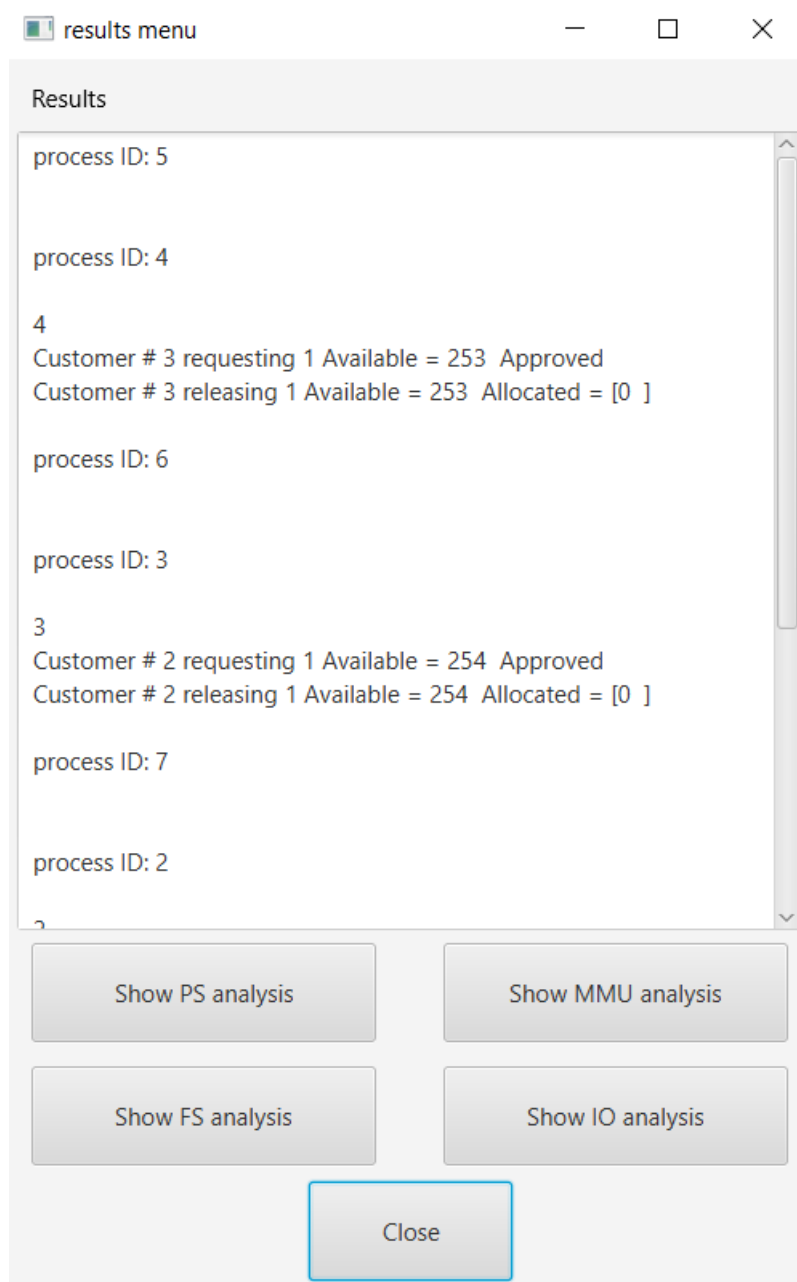
- deallocateMemory;

CPU Burst : 50

Priority: 5

Memory required: 0

After the processes have run, click on the show results menu. The following window should appear:



The following is the output of the processes:

```
process ID: 5
```

```
process ID: 4
```

```
4
Customer # 3 requesting 1 Available = 253 Approved
Customer # 3 releasing 1 Available = 253 Allocated = [0 ]
```

```
process ID: 6
```

```
process ID: 3
```

```
3
Customer # 2 requesting 1 Available = 254 Approved
Customer # 2 releasing 1 Available = 254 Allocated = [0 ]
```

```
process ID: 7
```

```
process ID: 2
```

```
2
Customer # 1 requesting 1 Available = 255 Approved
Customer # 1 releasing 1 Available = 255 Allocated = [0 ]
```

```
process ID: 8
```

```
process ID: 1
```

```
1
Customer # 0 requesting 1 Available = 256 Approved
Customer # 0 releasing 1 Available = 256 Allocated = [0 ]
```

---

Four processes have been stored in the RAM and removed from the RAM, despite the size of the physical address space being one. Therefore, the FIFO page replacement algorithm is functioning correctly.

### LRU

The VM thread stores an array of integers named LRUStack as an attribute. The integers are the page numbers of pages which have been used. The array acts as a stack, the page number of the most recently used page is at the end of the array and the page number of the least recently used page is at the start of the array. The page number of the least recently used page is removed and the victim frame corresponding to the least recently used page is selected.

To run the following test case, the ini.txt file must be configured like the following:

5  
FCFS

256  
1  
1  
16  
LRU  
512  
1000  
64  
21  
SSTF

To run the following test case, create 8 processes, the information associated with each process is described below:

Process 1:

Instructions:

- print(1);
- allocateMemory;

CPU Burst : 50

Priority: 5

Memory required: 1

Process 2:

Instructions:

- print(2);
- allocateMemory;

CPU Burst : 50

Priority: 5

Memory required: 1

Process 3:

Instructions:

- print(3);
- allocateMemory;

CPU Burst : 50

Priority: 5



Memory required: 1

Process 4:

Instructions:

- print(4);
- allocateMemory;

CPU Burst : 50

Priority: 5

Memory required: 1

Process 5:

Instructions:

- deallocateMemory;

CPU Burst : 50

Priority: 5

Memory required: 0

Process 6:

Instructions:

- deallocateMemory;

CPU Burst : 50

Priority: 5

Memory required: 0

Process 7:

Instructions:

- deallocateMemory;

CPU Burst : 50

Priority: 5

Memory required: 0

Process 8:

Instructions:

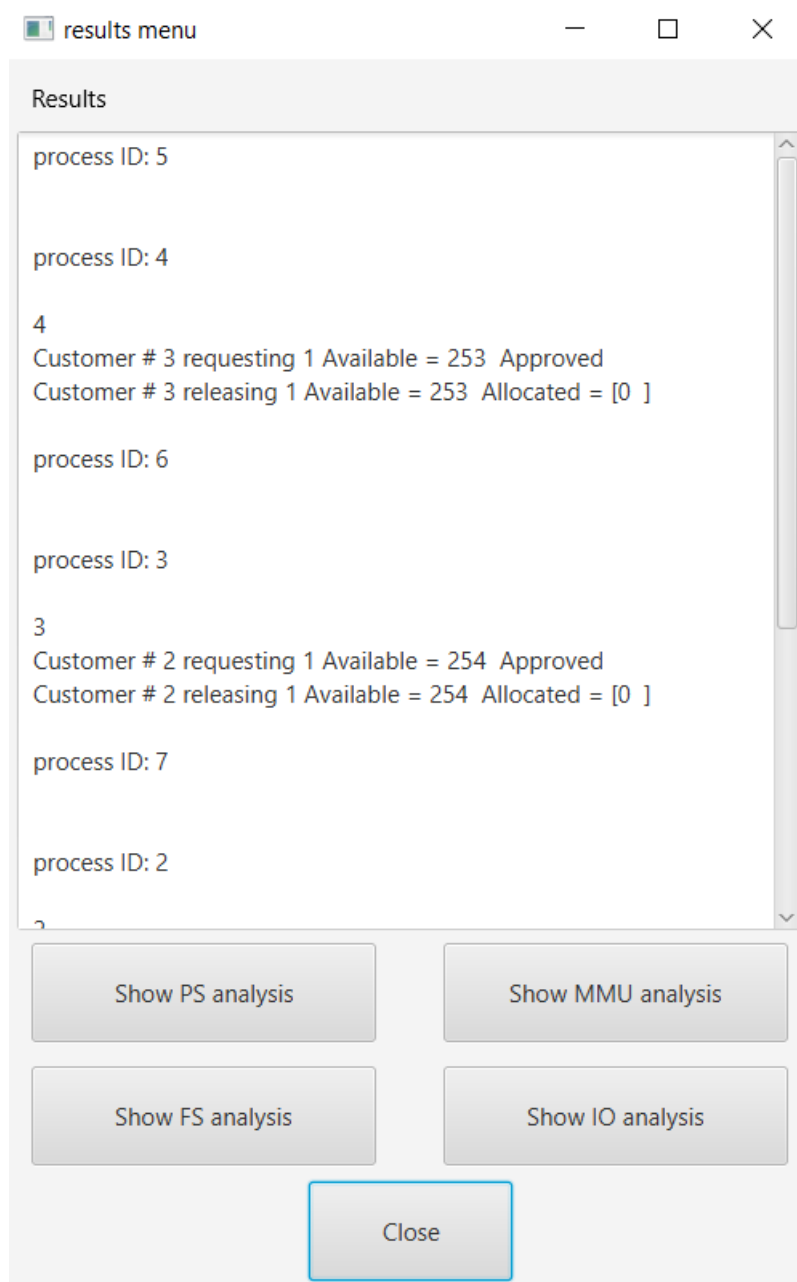
- deallocateMemory;

CPU Burst : 50

Priority: 5

Memory required: 0

After the processes have run, click on the show results menu. The following window should appear:



The following is the output of the processes:

```
process ID: 5
```

```
process ID: 4
```

```
4
Customer # 3 requesting 1 Available = 253  Approved
Customer # 3 releasing 1 Available = 253  Allocated = [0  ]
```

```
process ID: 6
```

```
process ID: 3
```

```
3
Customer # 2 requesting 1 Available = 254  Approved
Customer # 2 releasing 1 Available = 254  Allocated = [0  ]
```

```
process ID: 7
```

```
process ID: 2
```

```
2
Customer # 1 requesting 1 Available = 255  Approved
Customer # 1 releasing 1 Available = 255  Allocated = [0  ]
```

```
process ID: 8
```

```
process ID: 1
```

```
1
Customer # 0 requesting 1 Available = 256  Approved
Customer # 0 releasing 1 Available = 256  Allocated = [0  ]
```

---

Four processes have been stored in the RAM and removed from the RAM, despite the size of the physical address space being one. Therefore, the LRU page replacement algorithm is functioning correctly.

### Bankers algorithm

The memory allocated by a process is a resource request that is considered in the Bankers algorithm implementation. The code from the lab 6 solutions has been used to implement the banker's algorithm [13]. When the process requires more than 1 byte of memory, it is added to the bank. When the process is stored in memory, the process requests the memory resource. When the process is released from memory, it releases the memory resource. The banker's algorithm has been used to determine whether the storage of a process in the RAM will leave the process in a safe state or not.

## Disk scheduling algorithms

The disk scheduling algorithms which have been implemented are FCFS and SSTF. If the fileSystemBuffer contains more than one PCB object, the disk scheduling algorithm will select the disk request to run first. No source code has been used to write the disk scheduling algorithms.

### FCFS

The first PCB object in the fileSystemBuffer is removed and the disk request in the PCB object is run. The seek pointer of the disk request is determined and the head movement and cylinder position are updated.

To run the following test case, the ini.txt file must be configured like the following:

```
10
FCFS

256
256
256
16
LRU
512
1000
64
21
FCFS
```

To run the following test case, two input files must be used.

diskSchedulingAlgorithmFCFSTestpt1.txt must be used first, then

diskSchedulingAlgorithmFCFSTestpt2.txt. The program removes the processes included in the file which means the copy of the file must be saved to rerun the test multiple times. To run the test case, click on the Choose File button and select the input file. The steps required to choose an input file has been explained in the compiling and running section of this report.

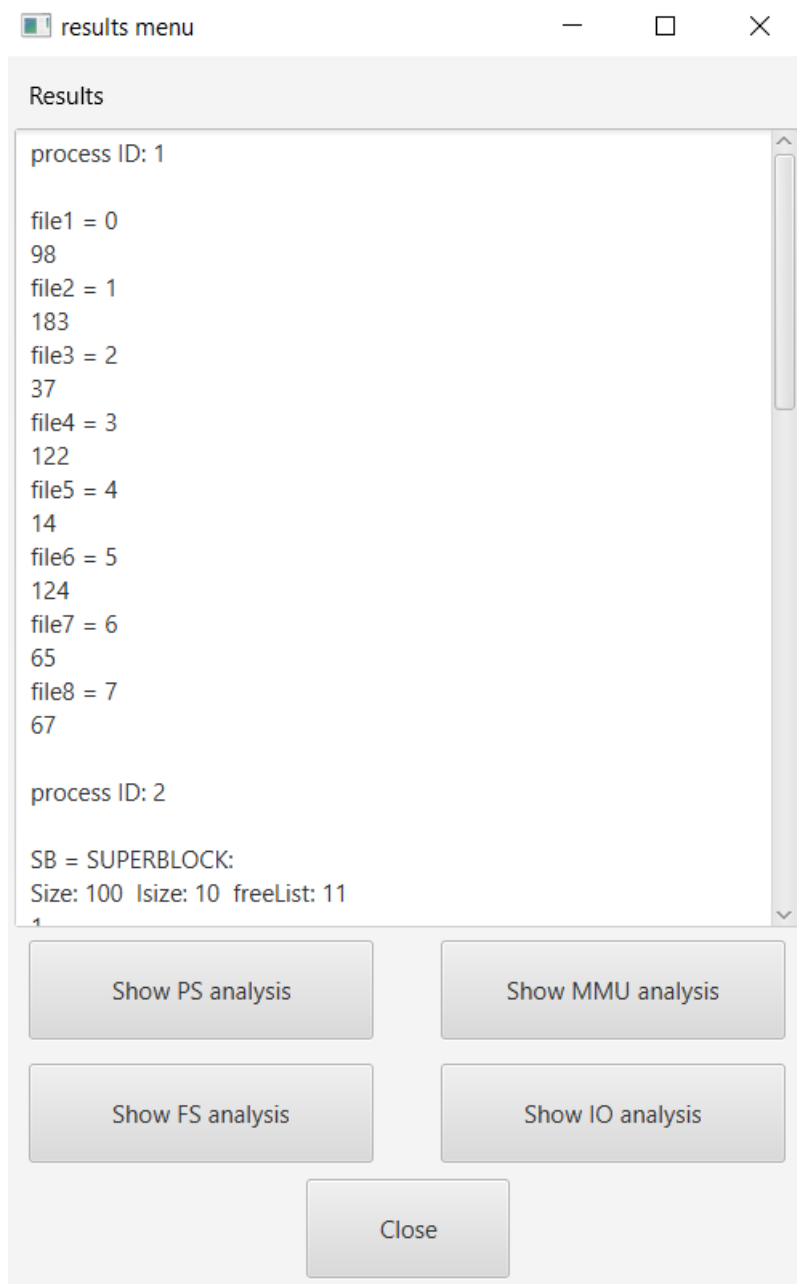
diskSchedulingAlgorithmFCFSTestpt1.txt contains the following process:

- file1 = create;seek file1 98 0;file2 = create;seek file2 183 0;file3 = create;seek file3 37 0;file4 = create;seek file4 122 0;file5 = create;seek file5 14 0;file6 = create;seek file6 124 0;file7 = create;seek file7 65 0;file8 = create;seek file8 67 0;exit;50;5;0;

diskSchedulingAlgorithmFCFSTestpt2.txt contains the following processes:

```
write file1 a 1;exit;50;5;0;
write file2 b 1;exit;50;5;0;
write file3 c 1;exit;50;5;0;
write file4 d 1;exit;50;5;0;
write file5 e 1;exit;50;5;0;
write file6 f 1;exit;50;5;0;
write file7 g 1;exit;50;5;0;
write file8 h 1;exit;50;5;0;
```

After the processes have run, click on the show results menu. The following window should appear:



The following is the output of the processes:

---

process ID: 1

file1 = 0

98

file2 = 1

183

file3 = 2

37

file4 = 3

122

file5 = 4

14

file6 = 5

124

file7 = 6

65

file8 = 7

67

process ID: 2

SB = SUPERBLOCK:

Size: 100 Isize: 10 freeList: 11

1

process ID: 3

SB = SUPERBLOCK:

Size: 100 Isize: 10 freeList: 11

1

process ID: 4

SB = SUPERBLOCK:

Size: 100 Isize: 10 freeList: 11

1

process ID: 5

SB = SUPERBLOCK:

Size: 100 Isize: 10 freeList: 11

1

```
process ID: 6
```

```
SB = SUPERBLOCK:  
Size: 100  Isize: 10  freeList: 11  
1
```

```
process ID: 7
```

```
SB = SUPERBLOCK:  
Size: 100  Isize: 10  freeList: 11  
1
```

```
process ID: 8
```

```
SB = SUPERBLOCK:  
Size: 100  Isize: 10  freeList: 11  
1
```

```
process ID: 9
```

```
SB = SUPERBLOCK:  
Size: 100  Isize: 10  freeList: 11  
1
```

If an error occurs which states the disk contains too many files, a new disk file must be saved in the main project file. A new disk file is provided in the input files folder. The disk file used is identical to the one in lab 9. The order which the disk requests have run is correct because the disk scheduling algorithm is FCFS. The disk request in process 2 is run first, then the disk request in process 3, then the disk request in process 4, then the disk request in process 5, then the disk request in process 6, then the disk request in process 7, then the disk request in process 8, then the disk request in process 9. The output of each disk request is correct. The FS analysis menu shows the seek distance is 693 which is correct.

### SSTF

The seek pointer of each disk request is determined, and the disk request with the lowest seek distance is run first. The head movement and cylinder position are updated.

To run the following test case, the ini.txt file must be configured like the following:

```
10  
FCFS  
  
256  
256  
256  
16  
LRU  
512  
1000  
64  
21  
SSTF
```

To run the following test case, two input files must be used.

diskSchedulingAlgorithmSSTFTestpt1.txt must be used first, then diskSchedulingAlgorithmSSTFTestpt2.txt. The program removes the processes included in the file which means the copy of the file must be saved to rerun the test multiple times. To run the test case, click on the Choose File button and select the input file. The steps required to choose an input file has been explained in the compiling and running section of this report.

diskSchedulingAlgorithmSSTFTestpt1.txt contains the following process:

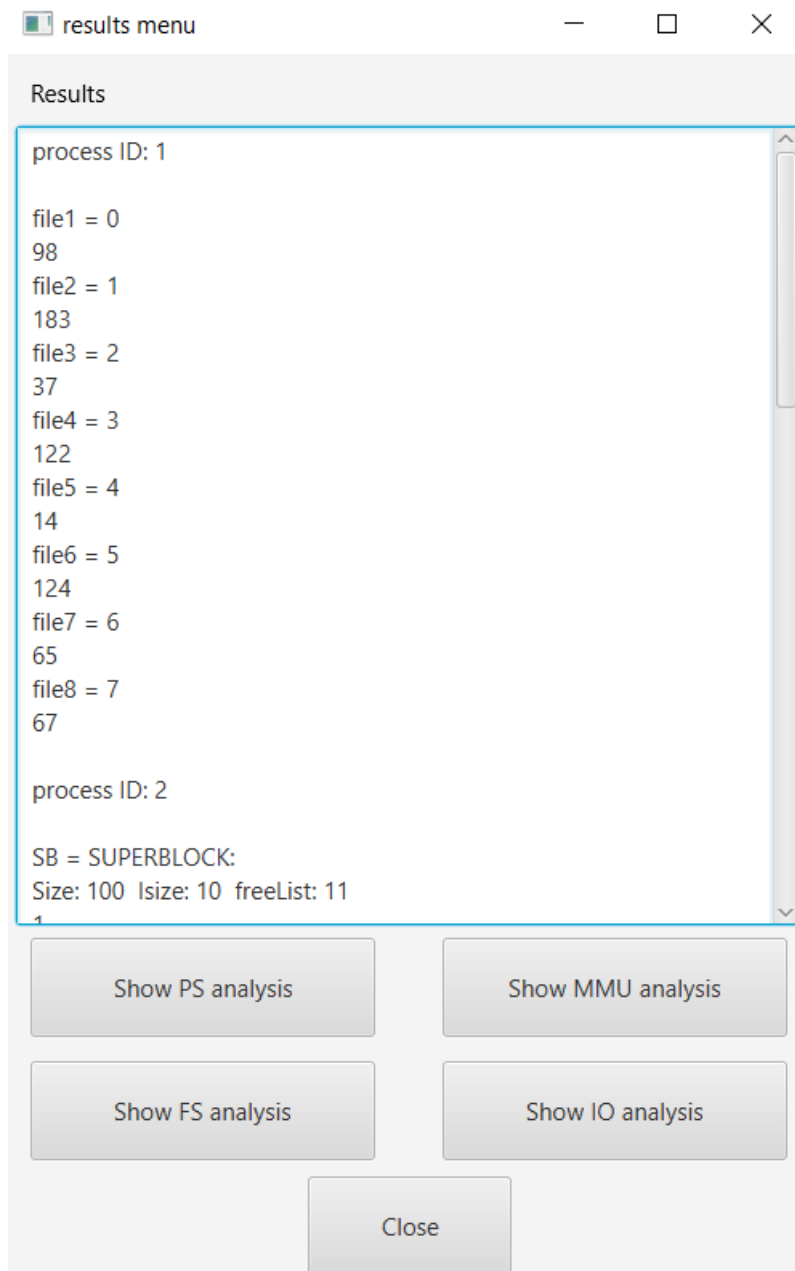
- file1 = create;seek file1 98 0;file2 = create;seek file2 183 0;file3 = create;seek file3 37 0;file4 = create;seek file4 122 0;file5 = create;seek file5 14 0;file6 = create;seek file6 124 0;file7 = create;seek file7 65 0;file8 = create;seek file8 67 0;exit;50;5;0;

diskSchedulingAlgorithmSSTFTestpt2.txt contains the following processes:

```
write file1 a 1;exit;50;5;0;
write file2 b 1;exit;50;5;0;
write file3 c 1;exit;50;5;0;
write file4 d 1;exit;50;5;0;
write file5 e 1;exit;50;5;0;
write file6 f 1;exit;50;5;0;
write file7 g 1;exit;50;5;0;
write file8 h 1;exit;50;5;0;
```

After the processes have run, click on the show results menu. The following window should appear:





The following is the output of the processes:

process ID: 1

file1 = 0  
98  
file2 = 1  
183  
file3 = 2  
37  
file4 = 3  
122  
file5 = 4  
14  
file6 = 5  
124  
file7 = 6  
65  
file8 = 7  
67

process ID: 2

SB = SUPERBLOCK:  
Size: 100 Isize: 10 freeList: 11  
1

process ID: 5

SB = SUPERBLOCK:  
Size: 100 Isize: 10 freeList: 11  
1

process ID: 7

SB = SUPERBLOCK:  
Size: 100 Isize: 10 freeList: 11  
1

process ID: 3

SB = SUPERBLOCK:  
Size: 100 Isize: 10 freeList: 11  
1

```
process ID: 9

SB = SUPERBLOCK:
Size: 100  Isize: 10  freeList: 11
1

process ID: 8

SB = SUPERBLOCK:
Size: 100  Isize: 10  freeList: 11
1

process ID: 4

SB = SUPERBLOCK:
Size: 100  Isize: 10  freeList: 11
1

process ID: 6

SB = SUPERBLOCK:
Size: 100  Isize: 10  freeList: 11
1
```

If an error occurs which states the disk contains too many files, a new disk file must be saved in the main project file. A new disk file is provided in the input files folder. The disk file used is identical to the one in lab 9. The first processes in the fileSystemBuffer are run first. The order which the disk requests have run is correct because the disk scheduling algorithm is SSTF. The disk request with the smallest seek distance is run before the disk requests with higher seek distances. The output of each disk request is correct. The FS analysis menu shows the seek distance is 352 which is correct.

### Synchronisation

Code from lab 5 has been used to synchronise the processes. [16] The synchronised keyword has been used. Wait/notify messaging has also been used. This synchronisation construct is easier to implement than using a semaphore to synchronise threads. The synchronised keyword and the wait/notify messaging allows the threads to always run. However, locks and conditions do not allow the threads to always run.

## Architecture and instruction set

### Process scheduler

The CPU supports multiprocessing. There is at most one CPU thread running in the program. Therefore, the OS simulator is assuming there is one CPU core, because parallel multitasking has not been included.

The process scheduler is assumed to have three components, a CPU, a process creation component, and a dispatcher component. Hence the process scheduler contains a CPU thread, a process creation thread, and a dispatcher thread. These components are run in parallel.

Modern processors used in computers have more than one CPU core. [17]. Therefore, the implemented process scheduler will not be compatible with any architecture for a computer, such as a laptop, in the market.

For example, the intel Core i7 has 8 CPU cores, which means this process scheduler will not be compatible with the intel i7. [18]

### MMU

Each process is assumed to store 1 byte of memory in the RAM. The size of the virtual address space can either be lower than, equal to, or higher than the size of the physical address space, depending on the values input in the ini.txt file. The MMU is assumed to support paging but not segmentation. The MMU is assumed to contain a TLB, the size of the TLB is a configuration parameter in the ini.txt file.

Processes in other architectures usually allocate more than 1 byte. The physical address space of an architecture is typically much smaller than the size of the virtual address space. Other architectures usually include both paging and segmentation. Therefore, the MMU will not be compatible with other computer architectures, segmentation needs to be implemented and processes will need to be able to allocate more than one byte of memory.

### File System

As explained in lab 9, “the structure of the file system closely follows UNIX-based file systems, where a superblock is used to describe the state of the file system. Internally, each file is stored in the file system using an inode data structure”. Lab 9 also explains that “each file in the system is described by an inode structure as used in Unix”.

Random access files are assumed to be used, which is why the disk file which simulates a disk is used as a random-access file. Random access files are usually used in random access files; therefore, the implemented file system is compatible with architectures of computers which use the Unix operating system.

### I/O System

Memory mapped I/O is assumed to be used. It is assumed that no DMA is used in the I/O system. Each peripheral is assumed to have both input and output functionality.

Most computer systems use memory mapped I/O. However, many computer systems also use DMA. Many i/o peripherals do not include either input functionality or output functionality. For example, a computer mouse is only an input device and does not have output functionality. Therefore, the I/O system is not compatible with other architectures.

## Instruction set

Memory management, file system, I/O system, simple JavaScript, and simple shell instructions are supported. The memory management, file system and I/O system instructions are explained in detail in the user manual. Simple JavaScript instructions and simple shell instructions are also supported.

Examples of supported JavaScript instructions are simple math calculations such as multiplication of variables. JavaScript variables are also supported such as  $x = 5$  and  $z = x * y$ . However, JavaScript functions are not supported.

Simple shell instructions such as `cd` are included in the OS simulator. The simple shell instructions which are supported depend on the operating system which the program is run on. For example, on a windows operating system, the simple shell instruction `print` is supported. However, on a Linux operating system, the simple shell instructions `pwd` and `ls` are supported.

Assembly language instructions are not supported. Therefore, the architecture of the computer the OS simulator is assuming is neither a reduced instruction set computer (RISC) or a complex instruction set computer (CISC).

The fetch-execute cycle has been completely omitted from the OS simulator. Registers have been excluded from the CPU. The data bus and memory bus have also been excluded. Therefore, the OS simulator is not compatible with any computer architectures because all CPUs contain registers. The fetch-execute cycle is included in every processor in a computer.

## References

- [1] "o7planning," [Online]. Available: <https://o7planning.org/en/10623/javafx-tutorial-for-beginners#a2746282>.
- [2] "Griffon," [Online]. Available: [http://griffon-framework.org/tutorials/5\\_mvc\\_patterns.html](http://griffon-framework.org/tutorials/5_mvc_patterns.html).
- [3] "stack overflow," [Online]. Available: <https://stackoverflow.com/questions/34853392/determine-clicked-button-in-javafx>.
- [4] "stack overflow," [Online]. Available: <https://stackoverflow.com/questions/32922424/how-to-get-the-current-opened-stage-in-javafx/49499114>.
- [5] "stack overflow," [Online]. Available: <https://stackoverflow.com/questions/35956527/javafx-javafx-scene-layout-anchorpane-cannot-be-cast-to-javafx-scene-layout-bo>.
- [6] "ProgramCreek," [Online]. Available: <https://www.programcreek.com/java-api-examples/?class=javafx.fxml.FXMLLoader&method=setController>.

- [7] "stack overflow," [Online]. Available: <https://stackoverflow.com/questions/25037724/how-to-close-a-java-window-with-a-button-click-javafx-project/41838183>.
- [8] "stack overflow," [Online]. Available: <https://stackoverflow.com/questions/32451986/how-to-hide-or-deactivate-a-textfield-and-a-label-with-javafx>.
- [9] A. Redko, "Oracle Docs," [Online]. Available: <https://docs.oracle.com/javafx/2/charts/line-chart.htm>.
- [10] "tutorialspoint," [Online]. Available: [https://www.tutorialspoint.com/javafx/pie\\_chart.htm](https://www.tutorialspoint.com/javafx/pie_chart.htm).  
]
- [11] A. Redko, "Oracle Docs," [Online]. Available: <https://docs.oracle.com/javafx/2/charts/bar-chart.htm>.  
]
- [12] "w3schools," [Online]. Available: [https://www.w3schools.com/java/java\\_files\\_create.asp](https://www.w3schools.com/java/java_files_create.asp).  
]
- [13] A. Silberschatz, P. B. Galvin and G. Gagne, Operating system concepts with Java, 8th ed, John Wiley & Sons, 2010.  
]
- [14] "AfterAcademy," 3 November 2019. [Online]. Available: <https://afteracademy.com/blog/what-is-burst-arrival-exit-response-waiting-turnaround-time-and-throughput>.
- [15] M. Helal, "SurreyLearn COM1032 19-20 Offering," I/O Systems, March 2020. [Online].  
]
- [16] M. Helal, "SurreyLearn COM1032 19-20 Offering," Threads Synchronisation, February 2020.  
] [Online].
- [17] S. Harding, "Tom's Hardware," 23 August 2018. [Online]. Available:  
] <https://www.tomshardware.com/uk/news/cpu-core-definition,37658.html#:~:text=Today%2C%20CPUs%20have%20been%20two,the%20more%20efficient%20it%20is..>
- [18] A. Williams, "Trusted Review," 21 May 2020. [Online]. Available:  
] <https://www.trustedreviews.com/best/best-intel-processor-3517396>.