Ain Shams University
Faculty of Engineering
Computer Engineering and Software Systems Program

CSE232: Advance Software Engineering – Fall 2021

OPERATING SYSTEMS

Final project

Names:

Mohamed Hatem Zakaria Elafifi	19P7582
Mustafa Mohamed Zeidan	19P7998
Mahmoud Abdalla Mohaseb	20P2787
Rana Mohamed Walid	19P8994
Omar Mohamed Ibrahim Alsayed	19p7813
Yasmin Haitham Abdelmoaty	18P2102

Table of Contents

Requirement 1: 2

Introduction: 2

Internal structure: 2

Algorithms: 4

Evaluation of the algorithms: 5

Comparative study between different algorithms that are used for each component: 8

Requirement 2: 11

1- Round Robin 11

2- Shortest job first 12

3- Priority 13

4- Multi-level feedback 14

Testing 14

Round robin: 15

Shortest job first: 17

Priority: 19

Multilevel Feedback: 21

Requirement 3: 25

Paging configuration: 30

Hierarchical paging: 27

Page replacement: 31

FIFO algorithm: 31

LRU Algorithm: 33

Requirement 4: 36

User extent size: 36

Allocating/freeing consecutive block: 37

Requirement 1:

Introduction:

An operating system is the linkage between software and hardware. It is responsible for controlling the hardware in a way that makes it easier and more efficient for the user of the computer system. It has the responsibility to utilize every resource that the computer has in the most efficient way possible. The operating system also creates an interface between the user and the hardware. Controlling programs is also the responsibility of the operating system.

The operating system consists of some main components like the kernel, the user interface and system calls which will all be discussed in more details later in this document.

Internal structure:

We can consider the operating system as one system that is composed of several components that build it. Without any of these component there will be chaos and the whole system will fail. The main components of the operating system are:

- The kernel,
- The user interface,
- Process management,
- Memory management,
- System interrupts,
- The computer security system
- Networking system etc.

These components are divided into smaller components. Each one has its specific goal to achieve so that the operating system works well.

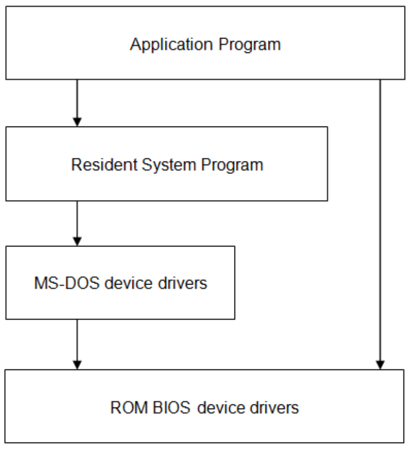
Operating systems vary in their types according to their uses and internal structure. The main reason of having different structures of operating system is because of the way that the components of the operating system are connected to each other. We have four main approaches to structure an operating system which are:

- Simple structure.
- Complex structure.
- Layered structure.
- Micro kernel.
- Modular structure.

1- <u>Simple structure</u>

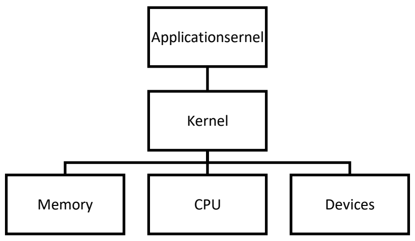
These operating systems don't have a well-defined structure because the components are not divided into any kind of modules. They are very simple and limited in functionality which is not separated well.

This is not a good thing because if one user program fails the whole system will indeed fail although all user programs can only access limited and basic I/O components. This makes the interfaces between the operating system and a user program tend to minimum.



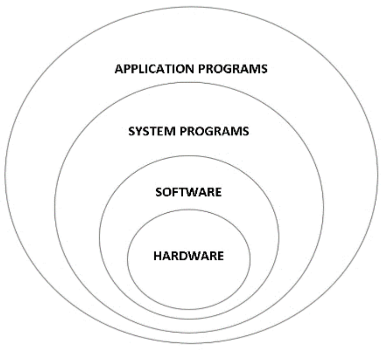
2- <u>Complex Structure</u>

These systems are divided into two main parts which are the system programs and the kernel. System programs are the programs that come with the operating system. The kernel is a program that runs all the time in the memory and supposed to be the core of the operating system. The kernel has multiple essential jobs like scheduling the CPU, file system and memory management.



3- <u>Layered Structure</u>

These operating systems divide its components into smaller pieces in the shape of layers making the structure of layers concatenated on top of each other. Level 0 is the hardware going up each layer depends on the functions provided by the layer beneath it.



4- <u>Modular structure</u>

In this structure the operating system is divided into modules. Each module consists of a set of components relative to each other's. It's like the layered approach but with more flexibility. You can also add more modules to the system as you upgrade it. It is considered the best approach until now.

5- <u>Micro kernel structure</u>

These operating systems tend to minimize the kernel space as much as possible. This gives the operating system a lot of advantages which will be discussed later while we talk about the MINIX3 operating system. But it has some disadvantages. The main disadvantage is the overhead on the system because for a process to get the desired function it has to send it to the kernel to search for the other process responsible of this function and return it.

Minix3 is a micro kernel structured operating system. It tries to reduce the size of the kernel to the minimum size it can. This is achieved by executing most of the processes in user-mode. The kernel mode has only about 4000 lines of code comparing with other Monolithic systems which have millions. This gives the operating system more reliability as less number of lines of code means less probability of having errors or mistakes achieving a bug free system.

It also makes the operating system more secure because each user-mode process is separated from other processes so if any harmful code is inserted the whole system is safe because it is away from the kernel. Micro kernel operating systems are also more expandable and executable in various architectures.

The micro kernel in Minix3 controls a small number of components which are:

a) CPU Scheduling.
b) Interrupts.
c) Inter communication of processes (IPC).

This makes it the only component that runs in kernel mode so that the micro kernel has full access of the hardware, memory management, and more of the essential components of the system.

Inter-process-communication is achieved by message passing which makes the process send messages to the rest of the system and receive from it to communicate with other processes. The problem with direct message passing is that the size of the message is limited (64 bytes). If the process wished to send/receive larger messages there is another method called Memory grant. In this method the message is sent to the memory and only the reference of it is sent to the receiving process. The kernel is responsible of reading/writing the message to the memory.

Algorithms:

Minix3 picked the most suitable algorithms for its functionality for each of its components in a way that doesn’t interfere with its structure. We will try to summarize most of these algorithms in this table then we will discuss and evaluate each of them in the next section.

Functionality	Algorithm
Process management	Multilevel feedback, priority.
Deadlock handling	The ostrich algorithms
Memory management	Managed by process manager
File system	MINIX file-system

Evaluation of the algorithms:

1- CPU scheduling

CPU scheduling is a specific part of any operating systems and the algorithm used for it may cause a huge difference between different operating systems. MINIX3 uses multi-level scheduling algorithm with some polices that may make it appear as it is a multi-level feedback although it is not 100% working with the feedback policies.

This algorithm prevents starvation and implements suitable queues for their suitable processes. It has some problems on the other hand like the complexity of implementing them and the overhead on the CPU.

The scheduling system contains 16 queues numbered from 0 up to 15. These queues differ in their uses as we will explain. The lowest level is for idle processes. Idle processes are not in use at the current time waiting for input or other messages. The highest queue is used for system tasks and the clock.

Processes may be moved from one queue to another. Queues are connected with the priority of the process. User processes are in the queues between the lowest and highest queues in the multi-level system. Server processes are higher in priority and lower than driver processes. Both of them have larger quantums than user processes because they must not be interrupted until they finish.

1-System processes
2-Driver processes
3-Server process
4-User processes

2- Deadlock handling

Deadlocks are considered one of the problems that can occur rarely in an operating system. The cost of solving this problem is so high that it is better to assume that it will never happen. Even if it happened the system will just have to boot up again which happens very rarely but loses all progress.

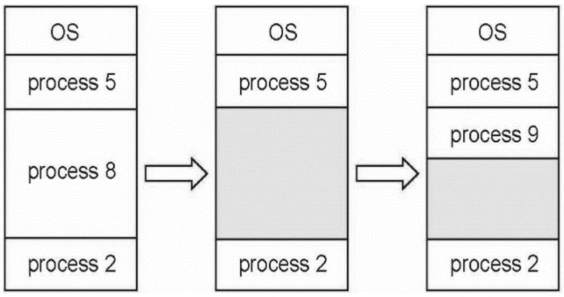
MINIX3 uses what’s called The Ostrich Algorithm. Simply this algorithm means to pretend that deadlocks never occurs. This algorithm is used in windows and UNIX because it is the most reasonable according to most of software engineers.

3- Memory management

MINIX3 does not have paging technique. It also does not use the swapping feature although its code is available. In MINIX3 the memory management is the responsibility of the process manager.

The process manager handles all system calls related to managing the processes. In MINIX3 it also holds a list of holes which are block of available memory of various sizes scattered throughout the memory. Keep in mind that the process manager is outside if the kernel and in user space

When a process enters the process manager or creates a child it assigns it to the first hole to fit it (first-fit) from the list. The process is forced to be in this exact space until it finishes due to the absence of the swapping function. It also cannot expand its memory space or shrink it.



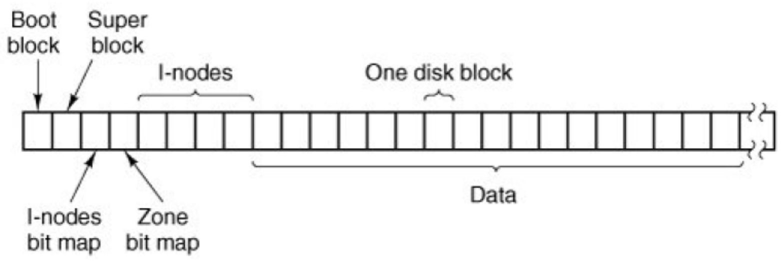
This algorithm makes the system simpler and easier to understand and more compatible with different architectures which gives the system the ability to load in embedded systems.

4- File system

The responsibility of the file system is to allocate and deallocate files on the disk and keeping track of their locations and other empty spaces on the disk. It also needs to provide some kind of security for these files from any unauthorized access.

MINIX3 tried to replicate the UNIX file system but in a much simpler way. The user sends a message to the file system with desired changes and the file system is responsible of making these changes happen and replying to the user with the results. The file system is divided into six main components:

- **The boot block:** contains the bootloader which loads and run the operating system
- **The superblock**: stores data about file system.
- **I-node bitmap:** stores the occupancy of each i-node (in use or empty).
- **Zone bitmap:** stores the occupancy of each zone
- **I-nodes:** represent files and directories.
- **Data:** contains all the data inside the file or directory and the largest part of the system.



How minix3 manages free space:

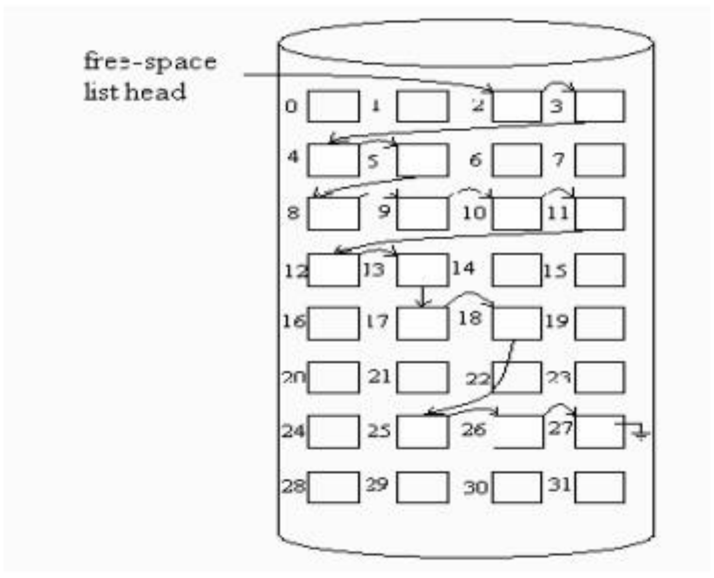
Bitmaps: It’s a group of bits that each bit represents a disk block. If the block is free it will be shown by 1s and by 0 if the block isn’t free. if there are lots of free blocks some of them may be used to hold the free list without any misuse of the disk.

Minix3 uses two bitmaps to find which blocks are free and which are not. The disk with n blocks will need a bitmap with n bits. There are two types of bitmaps (block bitmap and zone bitmap).

The zone bitmap is used to make sure that the disk blocks which belong to the same file are allocated in the same cylinder to develop the performance when the file is reading sequentially. We choose this to make it possible to allocate different blocks at a time.

Allocation:

While allocating a new block, minix3 allocates block by changing the bit which represents that the block in the bitmap to 1s that’s mean that this block now is allocating to a file. Also when emptying block the bit which represents the block will change to 0s.



Comparative study between different algorithms that are used for each component:

In this section we will compare MINIX3 algorithms with some alternatives that could have been used or are used in other operating systems.

1- <u>Process management:</u>

To try and compare the multi-level scheduling and other scheduling algorithms we will briefly discuss the other algorithms and then compare each one with the one used in MINIX3.

First: first come first served (FCFS) which works as the name says. The first process to reach the schedule is the first process to enter the CPU. This algorithm is very straight forward and it is easier to implement than the MFQ but it can cause an ineffective way of scheduling as the average waiting and turnaround time may increase noticeably.

Second: shortest job first which solved the problem of the FCFS algorithm by minimizing the waiting and turn around time. It has some defects though as we need to find an algorithm to pick the shortest of the available jobs and predict the next job using mathematical probability. It also might cause starvation as long processes may be always be delayed so we might also need to implement an aging algorithm.

Third: Priority scheduling which is kind of implemented in MINIX3 but instead of changing the processes positions in one queue it is used to move the processes between queues. Priority scheduling is easy to implement but it might cause starvation as well as the shortest job first (SJF) algorithm so it also needs aging algorithm.

Fourth: Round Robin scheduling is a scheduling algorithm that gives each process a limited time (quantum) to make progress before making it to wait for other processes. The queues in MINIX3 are round robin with different quantums so it is already implemented.

2- Deadlock prevention:

Some people think that it is not enough to run away from the problem instead of facing it and that deadlocks are brutal to the system and must be dealt with. There are two ways of handling deadlocks: preventing them from even happening or avoiding them.

To prevent a deadlock from happening we need to prevent its causes which are: mutual exclusion, hold and wait state, no preemption and circular wait. Breaking any of these properties will indeed affect the performance of the system by causing starvation. Some of the properties cannot even be broken due to the operating system responsibilities.

Avoiding deadlocks is by the banker's algorithm. This algorithm uses data structures to store the maximum resources needed for every process and their allocated ones. It also stores the available resources in the system. Then when a process requests resources it checks if the request does not exceed the process maximum or the available resources then allocates them to it.

This algorithm uses lots of data structures and raises the overhead on the system for its calculations so it's much better to assume deadlocks rarely occur instead of wasting all that time and resources.

3- <u>Memory management:</u>

The memory management system in MINIX3 is simple and easy to understand but it is not the most efficient comparing with other operating systems. One of the things we repeated a lot while talking about the memory management in MINIX3 is its lack of the paging algorithm.

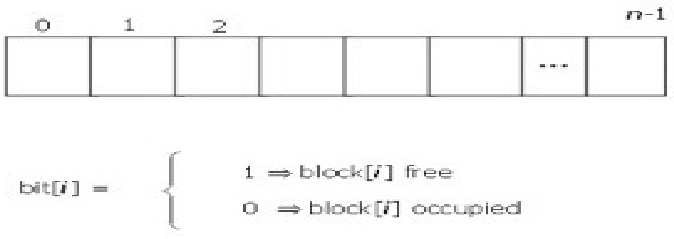
This algorithm divides the physical memory into powers of 2 of spaces called frames. Dividing the logical memory into pages. The algorithm keep track of empty frames and assigns them to whichever process that needs them. in conclusion paging solves external fragmentation.

This makes the memory more flexible so that a process can be in different spaces when returning to memory instead of in one bulk chunk in the memory so it can expand and shrink whenever it needs to. Also swapping is not used in MINIX3. Swapping is used to change the location of processes to find more room for others. Lacking of swapping leads to wasting parts of the memory due to not being able to move others.

4- <u>File system:</u>

The bitmap system and First fit allocation are the simplest algorithms that can be used in an operating system to manage files. They are easy to understand and very simple to implement and change in.

The bitmap system is also overhead friendly which means that it simplifies the searching for empty blocks which reduces the overall overhead on the system. The searching only starts from the last allocated bit which reduces the time of searching greatly. It is not only faster but also efficient.



Requirement 2:

In this section we are trying to change the Minix3 policy of scheduling and give the user different options to choose from. As we discussed before Minix3 uses multilevel queue. We are trying to implement 4 other algorithms of scheduling which are: Round Robin, Shortest job first, Priority and Multi level feedback scheduling.

When the kernel receives a process which need to be scheduled and finds that the process runs out of quantum, it sends a message to the user space scheduler, the user space scheduler receives the message in the main method. After that it calls the do_noquantum() function which has the responsibility to specify the new priority, new quantum, then call schedule_process_local() function with flags. Then schedule_process_local() function calls schedule_process() function which makes a system call to the kernel to execute this process.

We will let the user decide which algorithm to use by changing the variable algorithm as in first code and then check the value of algorithm in the do_noquantum which gives every process its quantum before sending it to execute.

```
* 1: Round Robin
* 2: Shortest job first
* 3: priority
* 4: Multi level feedback
```

MINIX3 used a multi-level queue with some flexibility of changing the level by changing the process priority. We commented the old code which lowered the queue of the process to control it using our new algorithms from line 102. We also commented the balance_queue function which raises the queue Then we added an if statement to check the algorithm variable and do the chosen algorithm. We added two new lines in the do_start_scheduling function to give our processes priority and time left just to help our algorithms.

```
if (rmp->priority < MIN_USER_Q) {
		rmp->priority += 1; // lower priority
```

```
	/*struct schedproc* rmp;
	int proc_nr;

	for (proc_nr=0, rmp=schedproc; proc_nr < NR_PROCS; proc_nr++, rmp++) {
		if (rmp->flags & IN_USE) {
			if (rmp->priority > rmp->max_priority) {
				rmp->priority -= 1; // increase priority

	schedule_process_local(rmp);
			}
		}
	}

	set_timer(&sched_timer, balance_timeout
```

```
rmp->in_queue_priority = rand() % 10; // random priority from 0- 9;
	rmp->expected_job_time = rand() % 900 + 1; // set random expected time from 1 - 900
```

1- Round Robin

As we removed the control of the queues now we are working with only one queue all we needed for implementing the round robin scheduling was giving any process that comes in the do_noquantum a fixed quantum to continue its cycle.

```
if (algorithm == 1) {
                        rmp->time_slice = 100; // giving the processes fixed quantum
            }
```

2- Shortest job first

We will need a variable to hold the shortest time of the available processes which we will name minBurst and a process struct to hold the shortest process which will be chosen. Then we made a for loop on all available processes to check if it has lower time than the previous shortest process and change the variables and that goes on until we get the shortest process and pass it.

```
else if (algorithm == 2) {
                        int minTime = 2000;
                        schedproc minProcess; // process with minimum job time

                        for (int i = 0; i < NR_PROCS; i++) {// for loop checking all processes to get the
shortest of them

                                    schedproc currentProc = schedproc[i];//process being checked

                                    if (currentProc->expected_job_time < minBurst && currentProc->flags
& IN_USE) {
                                                minBurst = currentProc->expected_job_time;
                                                minProcess = currentProc;
                                    }
                        }

                        minProcess->quantum = 200;// giving the shortest process enough quntum to
finish

                        if ((rv = schedule_process_local(minProcess)) != OK) {
```

3- Priority

We will need a variable to hold the highest priority of the available processes which we will name minPrio and a process struct to hold the highest priority process which will be chosen. Then we made a for loop on all available processes to check if it has higher priority than the previous highest priority process and change the variables and that goes on until we get the highest priority process and pass it.

```
else if (algorithm == 3) {
                int minPrio = 11; // max priority
                schedproc minProcess; // process with minimum priority

                for (int i = 0; i < NR_PROCS; i++) {// for loop checking all prcesses for highest
priority
                        schedproc currentProc = schedproc[i];
                        if (currentProc->in_queue_priority < minPrio && currentProc->flags &
IN_USE) {
                                minPrio = currentProc->in_queue_priority;
                                minProcess = currentProc;
                        }
                }
                minProcess->quantum = 200;// giving the shortest process enough quntum to
finish

                if ((rv = schedule_process_local(minProcess)) != OK) {
                        return rv;
```

4- Multi-level feedback

MINIX3 already has a multi-level scheduler with multiple queues all we needed to do was implement a policy to change the queue that the process is in when it finishes its quantum given in the previous queue. We implemented a 4 level queue which are all round robin with different quantums except the last level which is shortest job first. The first level has quantum of 80 the second 100 the third 160. The last level uses the same algorithm used in shortest job first part but with additional condition in the if statement to only compare processes in the last queue only.

```
else if (algorithm == 4) {

                if (rmp->priority != (USER_Q + 3))
                        rmp->priority += 1; // check if process reached last queue (level 4)

                switch (rmp->priority) {// switch for giving the process the suitable scheduling queue depending on
the queue priority

                case USER_Q://level1
                        rmp->time_slice = 80;
                        break;

                case (USER_Q+1) ://level2
                        rmp->time_slice = 100;
                        break;

                case (USER_Q + 2)://level3
                        rmp->time_slice = 160;
                        break;

                case (USER_Q + 3)://level4
                        int minBurst = 2000;
                        schedproc minProcess; // process with minimum job time

                        for (int i = 0; i < NR_PROCS; i++) {// for loop checking all processes to get the shortest of
them

                                schedproc currentProc = schedproc[i];//process being checked

                                if (currentProc->expected_job_time < minBurst && currentProc->flags & IN_USE
&& currentProc->priority == USER_Q + 3) {// check only processes in this level
                                        minBurst = currentProc->expected_job_time;
                                        minProcess = currentProc;
                                }
                        }

                        minProcess->quantum = 200;// giving the shortest process enough quntum to finish

                        if ((rv = schedule_process_local(minProcess)) != OK) {
```

Testing

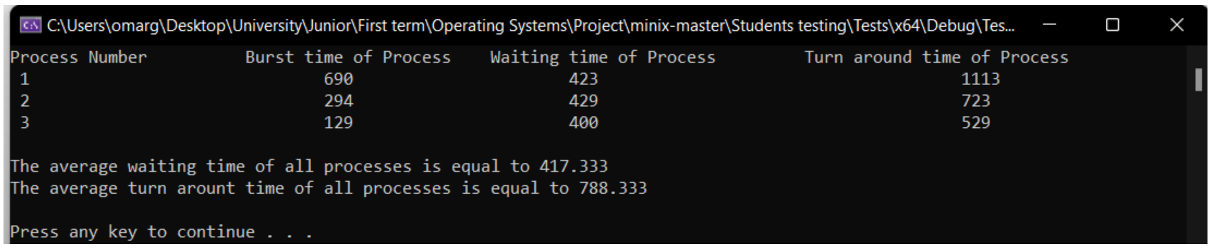
We used normal C++ to try and test the round robin algorithm by calculating the waiting and turnaround tine of each process and selecting the brust time manually. Then sjf, priority and multi-level feedback. (all test codes are in \minix-master\Students testing)

Round robin:

```
using namespace std;

class process {
public:
	char name[2];
	int priority;
	float timeLeft;
};

void RR_TurnAroundTime(int process[], int numberOfProcesses, int burst_time[], int waiting_time[], int
turnaround_time[])
{
	for (int i = 0; i < numberOfProcesses; i++)
		turnaround_time[i] = burst_time[i] + waiting_time[i];
}

void RR_waitingtime(int process[], int numberOfProcesses,
	int burstTime[], int waitingTime[], int quantum)
{
	int* remaining_burstTime = new int[numberOfProcesses];
	for (int i = 0; i < numberOfProcesses; i++)
		remaining_burstTime[i] = burstTime[i];

	int current_time = 0;

	//while loop untill all of the processes are executed
	while (true)
	{
		bool executed = true;

		for (int i = 0; i < numberOfProcesses; i++)
		{

			if (remaining_burstTime[i] > 0)
			{
				executed = false; // not all processes are done

				if (remaining_burstTime[i] > quantum)
				{
					current_time += quantum;

					remaining_burstTime[i] -= quantum;
				}


				else
				{

					current_time = current_time + remaining_burstTime[i];

					waitingTime[i] = current_time - burstTime[i];

					// when it is fully executed the remaining time becomes 0
```

```cpp
void RR_averageTime(int process[], int numberOfProcesses, int burst_time[], int quantum)
{
	int* waiting_time = new int[numberOfProcesses];
	int* turnaround_time = new int[numberOfProcesses];
	int total_waitingTime = 0, total_turnaround = 0;
	RR_waitingtime(process, numberOfProcesses, burst_time, waiting_time, quantum);
	RR_TurnAroundTime(process, numberOfProcesses, burst_time, waiting_time, turnaround_time);
	cout << "Process Number " << " \tBurst time of Process"
		<< "\t Waiting time of Process" << "\t Turn around time of Process" << endl;


	for (int i = 0; i < numberOfProcesses; i++)
	{
		total_waitingTime = total_waitingTime + waiting_time[i];
		total_turnaround = total_turnaround + turnaround_time[i];
		cout << " " << i + 1 << "\t\t\t\t" << burst_time[i] << "\t\t\t "
			<< waiting_time[i] << "\t\t\t\t\t " << turnaround_time[i] << endl;
	}

	cout << endl << "The average waiting time of all processes is equal to "
		<< (float)total_waitingTime / (float)numberOfProcesses << endl;
	cout << "The average turn arount time of all processes is equal to "
		<< (float)total_turnaround / (float)numberOfProcesses << endl;
	cout << endl;
}

int main()
{
	// Id of the processes
	int processes[] = { 1, 2, 3 };
	int numberOfProcesses = sizeof processes / sizeof processes[0];
	int burst_time[] = { 690, 294, 129 };
	int quantum = 100;
	RR_averageTime(processes, numberOfProcesses, burst_time, quantum);

	system("pause");
```

The output


C:\Users\omarg\Desktop\University\Junior\First term\Operating Systems\Project\minix-master\Students testing\Tests\x64\Debug\Tes...

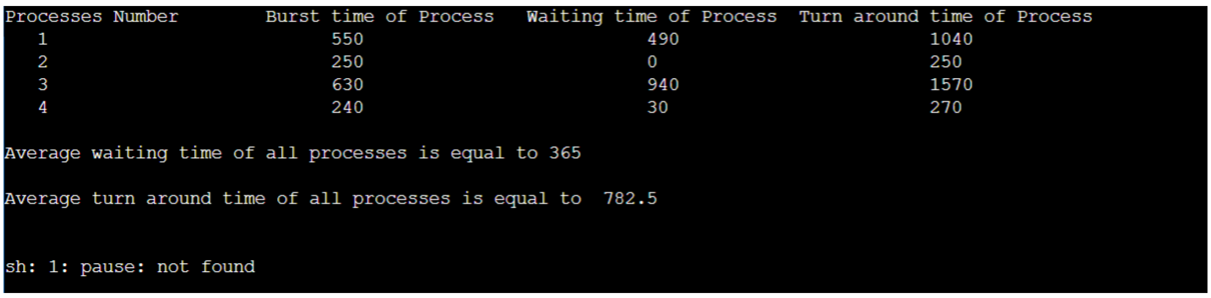
```
Process Number          Burst time of Process    Waiting time of Process         Turn around time of Process
 1                              690                      423                             1113
 2                              294                      429                             723
 3                              129                      400                             529

The average waiting time of all processes is equal to 417.333
The average turn arount time of all processes is equal to 788.333

Press any key to continue . . .
```

Shortest job first:

```
void SJF_WaitingTime(int noOfProcceses, int waiting_time[], Process process[]) {
	int* remainingTime = new int[noOfProcceses];
	for (int k = 0; k < noOfProcceses; k++)
		remainingTime[k] = process[k].burst_time;
	int isComplete = 0;
	int time = 0;
	int minimum = INT_MAX;
	int minIndex = 0;
	int execTime; //the execution time
	bool checkProccessAvailability = false;
	while (isComplete != noOfProcceses) {
		for (int f = 0; f < noOfProcceses; f++) {
			if ((process[f].arrivalTime <= time) && (remainingTime[f] < minimum) && remainingTime[f] > 0) {
				minimum = remainingTime[f];
				minIndex = f;
				checkProccessAvailability = true;
			}
		}
		if (checkProccessAvailability == false) {
			time++;
			continue;
		}
		remainingTime[minIndex]--;
		minimum = remainingTime[minIndex];
		if (minimum == 0)
			minimum = INT_MAX;
		if (remainingTime[minIndex] == 0) {
			execTime = time + 1;
			isComplete++;
			checkProccessAvailability = false;
			waiting_time[minIndex] = execTime - process[minIndex].burst_time - process[minIndex].arrivalTime;
			if (waiting_time[minIndex] <= 0)
				waiting_time[minIndex] = 0;
```

```
void SJF_AverageTime(int noOfProcceses, Process process[]) {
	int* turnAroundTime = new int[noOfProcceses];
	int* waiting_time = new int[noOfProcceses];
	int totalWaitingTime = 0,
		totalTurnAroundTime = 0;
	SJF_WaitingTime(noOfProcceses, waiting_time, process);
	SJF_turnAround(noOfProcceses, waiting_time, turnAroundTime, process);
	cout << "Processes Number" << " \tBurst time of Process" << " \tWaiting time of Process" << "\t Turn around
time of Process\n";
	for (int h = 0; h < noOfProcceses; h++) {
		totalTurnAroundTime = totalTurnAroundTime + turnAroundTime[h];
		totalWaitingTime = totalWaitingTime + waiting_time[h];
		cout << "   " << process[h].processID << "\t\t\t      " << process[h].burst_time
			<< "\t\t\t   " << waiting_time[h] << "\t\t\t     " << turnAroundTime[h] << endl;
	}
	cout << endl << "Average waiting time of all processes is equal to " << (double)totalWaitingTime /
(double)noOfProcceses;
	cout << endl << endl << "Average turn around time of all processes is equal to  " <<
(double)totalTurnAroundTime / (double)noOfProcceses << endl << endl;
}

int main() {
	Process process[] = { { 1, 550, 120 },{ 2, 250, 130 },{ 3, 630, 220 },{ 4, 240, 350 } };
	int noOfProcesses = sizeof(process) / sizeof(process[0]);
	SJF_AverageTime(noOfProcesses, process);
	cout << endl;
	system("pause");
	return 0;
```

Output



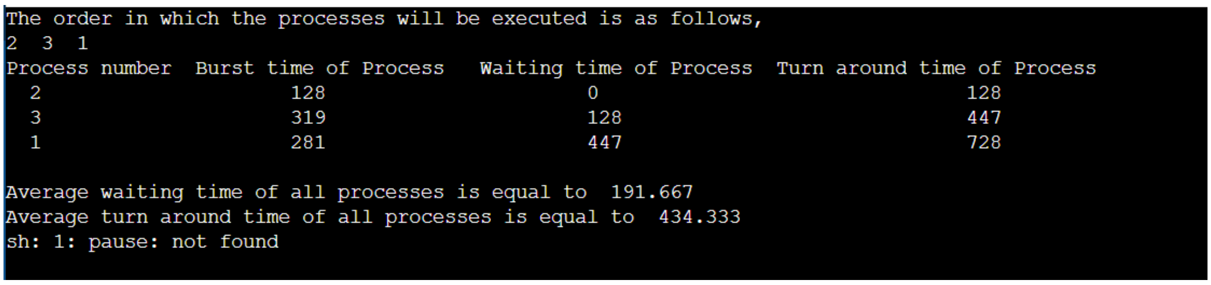
```
Processes Number        Burst time of Process   Waiting time of Process  Turn around time of Process
   1                          550                        490                       1040
   2                          250                        0                         250
   3                          630                        940                       1570
   4                          240                        30                        270

Average waiting time of all processes is equal to 365

Average turn around time of all processes is equal to  782.5


sh: 1: pause: not found
```

Priority:

```
struct Process2
{
        int processID;
        int burst_time;
        int processPriority;
};
bool comparator(Process2 process1, Process2 process2)
{
        bool x = (process1.processPriority < process2.processPriority);
        return x;
}
void calculateTat(int noOfProccesses, int waitingTime[], int turnAroundTime[], Process2 process[])
{
        for (int g = 0; g < noOfProccesses; g++)
                turnAroundTime[g] = process[g].burst_time + waitingTime[g];
}
void CalculateWT(int noOfProccesses, int waitingTime[], Process2 process[])
{
        waitingTime[0] = 0;
        for (int o = 1; o < noOfProccesses; o++)
                waitingTime[o] = process[o - 1].burst_time + waitingTime[o - 1];
}
void calculateAverageTime(Process2 process[], int noOfProccesses)
{
        int* waitingTime = new int[noOfProccesses];
        int* turnAroundTime = new int[noOfProccesses];
        int totalWaitingTime = 0;
        int totalTurnAroundTime = 0;
        CalculateWT(noOfProccesses, waitingTime, process);
        calculateTat(noOfProccesses, waitingTime, turnAroundTime, process);
        cout << endl << "Process number" << " \tBurst time of Process"
                << " \tWaiting time of Process" << "\t Turn around time of Process" << endl;
        for (int k = 0; k < noOfProccesses; k++)
        {
                totalWaitingTime = totalWaitingTime + waitingTime[k];
                totalTurnAroundTime = totalTurnAroundTime + turnAroundTime[k];
                cout << "  " << process[k].processID << "\t\t\t"
                        << process[k].burst_time << "\t\t\t " << waitingTime[k]
                        << "\t\t\t\t " << turnAroundTime[k] << endl;
        }

        cout << endl << "Average waiting time of all processes is equal to  "
                << (double)totalWaitingTime / (double)noOfProccesses;
        cout << endl << "Average turn around time of all processes is equal to  "
                << (double)totalTurnAroundTime / (double)noOfProccesses << endl;
}

void prioritySchedular(Process2 process[], int noOfProcesses)
{
        sort(process, process + noOfProcesses, comparator);

        cout << "The order in which the processes will be executed is as follows," << endl;;
        for (int u = 0; u < noOfProcesses; u++)
```

```
int main()
{
	Process2 process[] = { { 1, 12, 2 },{ 2, 8, 0 },{ 3, 9, 1 } };
	int noOfProcesses = sizeof process / sizeof process[0];
	prioritySchedular(process, noOfProcesses);
	system("pause");
	return 0;
}
```

Output:



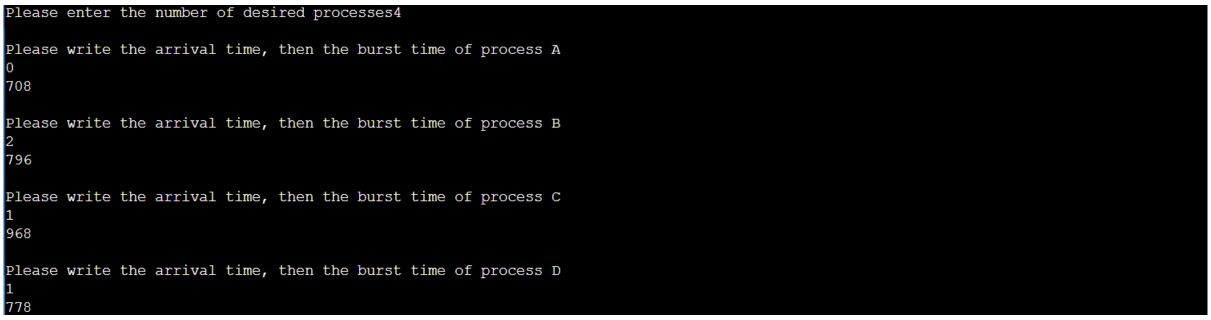
```
The order in which the processes will be executed is as follows,
2  3  1
Process number  Burst time of Process   Waiting time of Process  Turn around time of Process
  2                     128                         0                               128
  3                     319                         128                             447
  1                     281                         447                             728

Average waiting time of all processes is equal to  191.667
Average turn around time of all processes is equal to  434.333
sh: 1: pause: not found
```

Multilevel Feedback:

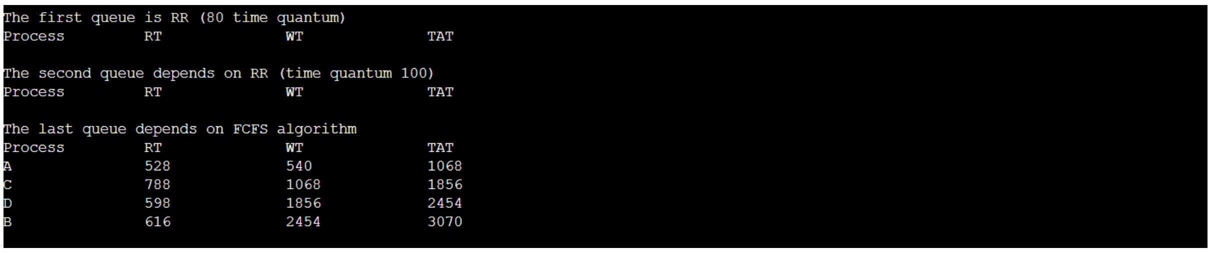
```
struct Process
{
        char processName;
        int arrivalTime;
        int currentTime;
        int turnAroundTime;
        int waitingTime;
        int burstTime;
        int remainingTime;
};
void sortByArrival(Process queue1[], Process queue2[], Process queue3[], int noOfProcesses)
{
        struct Process processTemp;
        for (int iCounter = 0; iCounter < noOfProcesses; iCounter++)
        {
                for (int jCounter = iCounter + 1; jCounter < noOfProcesses; jCounter++)
                {
                        if (queue1[iCounter].arrivalTime > queue1[jCounter].arrivalTime)
                        {
                                processTemp = queue1[iCounter];
                                queue1[iCounter] = queue1[jCounter];
                                queue1[jCounter] = processTemp;
}}}}
```

```
int main()
{
	Process queue1[10], queue2[10], queue3[10];
	int noOfProcesses;
	int  kCounter = 0, rCounter = 0;
	int timee = 0;
	int quantum1 = 5, quantum2 = 8, flag = 0;
	char c;
	cout << "Please enter the number of desired processes";
	cin >> noOfProcesses;
	for (int i = 0, c = 'A'; i < noOfProcesses; i++, c++)
	{
		queue1[i].processName = c;
		cout << endl << "Please write the arrival time, then the burst time of process " << queue1[i].processName << endl;
		cin >> queue1[i].arrivalTime >> queue1[i].burstTime;
		queue1[i].remainingTime = queue1[i].burstTime;

	}
	sortByArrival(queue1, queue2, queue3, noOfProcesses);
	timee = queue1[0].arrivalTime;
	cout << "The first queue is RR (5 time quantum)" << endl;
	cout << "Process\t\tRT\t\tWT\t\tTAT" << endl;
	for (int i = 0; i < noOfProcesses; i++)
	{

		if (queue1[i].remainingTime <= quantum1)
		{
			timee += queue1[i].remainingTime;
			queue1[i].remainingTime = 0;
			queue1[i].waitingTime = timee - queue1[i].arrivalTime - queue1[i].burstTime;
			queue1[i].turnAroundTime = timee - queue1[i].arrivalTime;
			cout << queue1[i].processName << "\t\t" << queue1[i].burstTime << "\t\t" << queue1[i].waitingTime << "\t\t" << queue1[i].turnAroundTime << endl;

		}
		else
		{
			queue2[kCounter].waitingTime = timee;
			timee += quantum1;
			queue1[i].remainingTime = queue1[i].remainingTime - quantum1;
			queue2[kCounter].burstTime = queue1[i].remainingTime;
			queue2[kCounter].remainingTime = queue2[kCounter].burstTime;
			queue2[kCounter].processName = queue1[i].processName;
			kCounter++;
			flag = 1;
		}
	}
	if (flag == 1)
	{
		cout << endl << "The second queue depends on RR (time quantum 8)" << endl;
		cout << "Process\t\tRT\t\tWT\t\tTAT" << endl;
	}for (int i = 0; i < kCounter; i++)
	{
		if (queue2[i].remainingTime <= quantum2)
		{
			timee += queue2[i].remainingTime;
			queue2[i].remainingTime = 0;
			queue2[i].waitingTime = timee - quantum1 - queue2[i].burstTime;
			queue2[i].turnAroundTime = timee - queue2[i].arrivalTime;
			cout << queue2[i].processName << "\t\t" << queue2[i].burstTime << "\t\t" << queue2[i].waitingTime << "\t\t" << queue2[i].turnAroundTime << endl;

		}
		else
		{
			queue3[rCounter].arrivalTime = timee;
			timee += quantum2;
			queue2[i].remainingTime -= quantum2;
			queue3[rCounter].burstTime = queue2[i].remainingTime;
			queue3[rCounter].remainingTime = queue3[rCounter].burstTime;
			queue3[rCounter].processName = queue2[i].processName;
			rCounter++;
			flag = 2;
		}
	}

	if (flag == 2) {
		cout << endl << "The last queue depends on FCFS algorithm ";
		cout << "\nProcess\t\tRT\t\tWT\t\tTAT" << endl;
	}
	for (int i = 0; i < rCounter; i++)
	{
		if (i == 0)
			queue3[i].currentTime = queue3[i].burstTime + timee - quantum1 - quantum2;
		else
			queue3[i].currentTime = queue3[i - 1].currentTime + queue3[i].burstTime;

	}

	for (int i = 0; i < rCounter; i++)
	{
		queue3[i].turnAroundTime = queue3[i].currentTime;]
		queue3[i].waitingTime = queue3[i].turnAroundTime - queue3[i].burstTime;
		cout << queue3[i].processName << "\t\t" << queue3[i].burstTime << "\t\t" << queue3[i].waitingTime << "\t\t" << queue3[i].turnAroundTime << endl;
```

Output:



```
Please enter the number of desired processes4

Please write the arrival time, then the burst time of process A
0
708

Please write the arrival time, then the burst time of process B
2
796

Please write the arrival time, then the burst time of process C
1
968

Please write the arrival time, then the burst time of process D
1
778
```



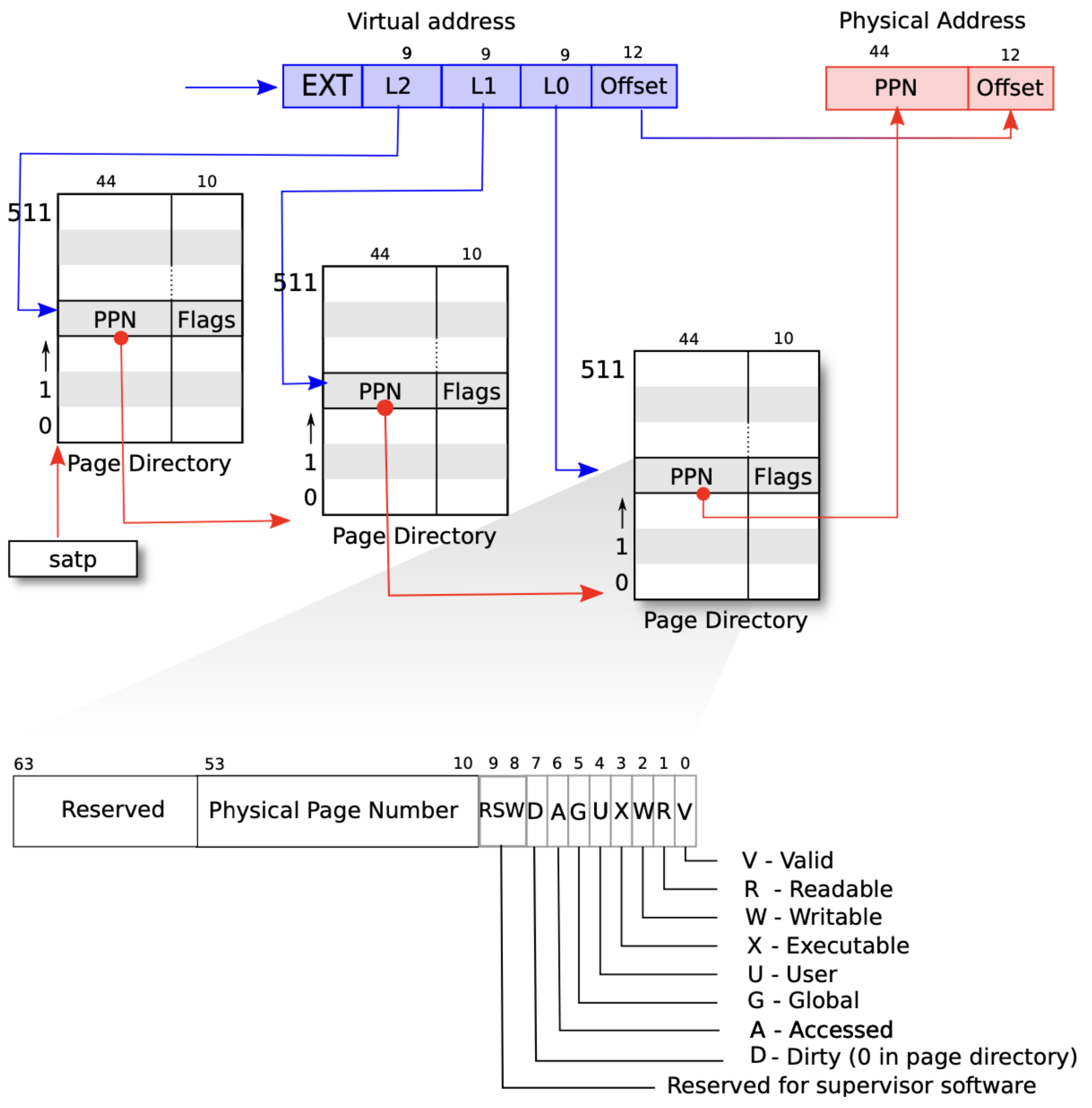
```
The first queue is RR (80 time quantum)
Process         RT              WT              TAT

The second queue depends on RR (time quantum 100)
Process         RT              WT              TAT

The last queue depends on FCFS algorithm
Process         RT              WT              TAT
A               528             540             1068
C               788             1068            1856
D               598             1856            2454
B               616             2454            3070
```

Requirement 3:

Paging in XV6

What is paging ?

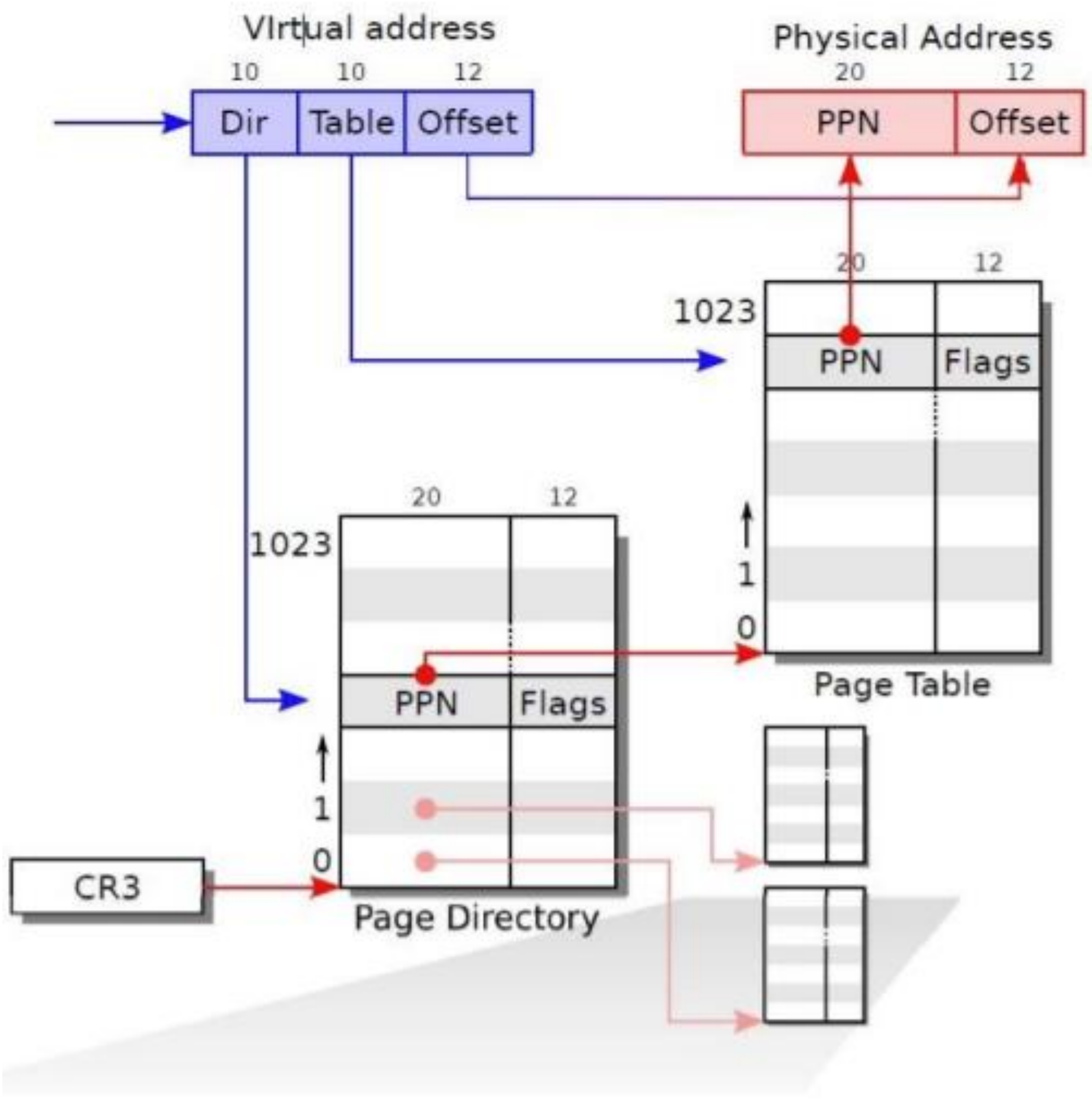
Paging is basically a memory management method in computer operating systems that allows computers to store and retrieve data from secondary storage –Disk– to main memory –RAM–. The OS recovers data from secondary storage in the form of equal sized blocks known as pages. It is a non-contiguous allocation (virtual memory), so it is mainly used to increase the CPU utilization. Paging is an incredibly significant aspect of the implementation of the virtual memory nowadays. It helps the computer to surpass the size of usable physical memory using the secondary storage.



Page table:

Page Table is a data structure that a virtual memory use in the process of the mapping that keeps track of these virtual address to the physical address in a computer operation system.

In XV6 the 2-level hierarchy gives both pages and page tables and their directories some flags to see if it is present PTE_P And if it can be accessed by user level processes PTE_U also if it is allowed to other processes to write in this page or only able to read PTE_W.



Hierarchical paging:

Multilevel Paging is a paging scheme which consist of two or more levels of page tables in a hierarchical manner. It is also known as hierarchical paging. The entries of the level (1) page table are pointers to a level (2) page table and entries of the level (2) page tables are pointers to a level (3) page table and so on. The entries of the last level page table are stores actual frame information. Level (1) contain single page table and address of that table is stored in PTBR (Page Table Base Register).

Note: Xv6 doesn't implement Hierarchical paging by default .

Code

```
typedef struct inner
{
	int id;
	int innerp;
	int frame_no;
}inner;
typedef struct outer

{
	int outerp;
	int id;
}outer;
void print(int f[], int offset[])
{
	cout << "-----------------------------------------------------------------------" << endl;
	cout << "| FRAME NUMBER | OFFSET |" << endl;
	cout << "| ";
	for (int i = 20; i >= 0; i--)
		cout << f[i];
	cout << "| ";
	for (int i = 0; i < 12; i++)
		cout << offset[i];
	cout << "|" << endl;
	cout << "-----------------------------------------------------------------------" << endl;
}
long long int BinaryToDecimal(int arr[], int size)
{
	int k = 0;
	long long int sum = 0;
	for (int i = size - 1; i >= 0; i--)
	{
		sum = sum + arr[i] * pow(2, k++);
	}
	return sum;
}
void DecimalToBinary(int f, int size, int offset[])
{
        int arr[26];
        int k = 0;
        memset(arr, 0, sizeof(arr));
        while (f > 0)

        {
                int rem = f % 2;
                arr[k++] = rem;
                f = f / 2;
        }
        print(arr, offset);
}
void multi()
{
        cout << "\t\t\t\tTWO LEVEL PAGING\t\t\t" << endl << endl;
        //OUTER PAGE TABLE
        cout << "Enter outer page table" << endl;
        outer o[20];
        for (int i = 0; i < 5; i++)
        {
                cin >> o[i].outerp >> o[i].id;
        }
        //INNER PAGE TABLE
        cout << "Enter inner page table" << endl;
        inner inn[20];
        for (int i = 0; i < 5; i++)
        {
                for (int j = 0; j < 3; j++)
                {
                        inn[j].id = i;
                        cin >> inn[j].innerp >> inn[j].frame_no;
                }
        }
        int p[20]; //20 bits
        int d[12]; //12 bits offset
        cout << "Enter the logical address generated by CPU" << endl;
        for (int i = 0; i < 20; i++)
                cin >> p[i];
        for (int i = 0; i < 12; i++)
                cin >> d[i];

                long long int pagenumber = BinaryToDecimal(p, 20);
        long long int offset = BinaryToDecimal(d, 12);
        cout << endl << "------------------------" << endl;
        cout << "| PAGE NUMBER | OFFSET |" << endl
                ;
        cout << "| " << pagenumber << " | " << offset << " |" << endl;
        cout << "------------------------------" << endl;
        int p1[10];
        int p2[10];
        for (int i = 0; i < 20; i++)
        {
                if (i < 10)
                        p1[i] = p[i];
                else
```

```
                        p2[i - 10] = p[i];
            }
            long long int outerpage = BinaryToDecimal(p1, 10);
            long long int innerpage = BinaryToDecimal(p2, 10);
            cout << endl << "----------------------------------------------------" << endl;
            cout << "| OUTER PAGE NUMBER | INNER PAGE NUMBER | OFFSET |" << endl;
            cout << "| " << outerpage << " | " << innerpage << " |
                        "<<offset<<" | "<<endl;
                        cout << "-----------------------------------------------------------" << endl;
            int frame;
            int flagO = 0;
            int flagI = 0;
            for (int i = 0; i < 5; i++)
            {
                        if (o[i].outerp == outerpage)
                        {
                                    flagO = 1;
                                    int id = o[i].id;
                                    for (int j = 0; j < 15; j++)
                                    {
                                                if (inn[j].id == id && inn[j].innerp == innerpage)

                                                {
                                                            frame = inn[j].frame_no;
                                                            flagI = 1;
                                                            break;
                                                }
                                    }
                                    if (flagI == 0)
                                                cout << "There is no frame associated with " << innerpage << " inner
page
                                                number"<<endl;
                                                break;
                        }
            }
            if (flagO == 0)
                        cout << "There is no inner page table associated with " << outerpage << " outer page
                        number"<<endl;
            else if (flagI == 1)
            {
                        cout << "Frame Number =" << frame << endl;
                        cout << "Page Size =" << pow(2, 12) << endl;
                        cout << "Offset =" << offset << endl;
                        long long int physical_address = frame * pow(2, 12) + offset;
                        cout << "Physical Address =" << physical_address << endl;
                        DecimalToBinary(frame, 26, d);
            }
            else
                        cout << "Page Fault" << endl;
}
```

Paging configuration:

Page size is normally 4KB in XV6 which requires the least significant 12 bits of the page indexing address and the rest 20 bits themselves are used to index the page from the page table by using the other 2 additional addresses for accessing. This way of multiplexing the address space of the different processes onto a single physical memory provides protection to the memories of different processes, so changing the page sizes is not advisable. Kalloc.c is the physical memory allocator, used to allocate 4096-byte per page, using the PGSIZE. The translation from virtual to physical addresses is performed by the MMU.

Screenshot of the mmu.h where PGSIZE is defined, as well as directory entries per page and table entries per page. Which will be required to change, if any edit is made to the page size default value
.

```
// page directory index
#define PDX(va)         (((uint)(va) >> PDXSHIFT) & 0x3FF)

// page table index
#define PTX(va)         (((uint)(va) >> PTXSHIFT) & 0x3FF)

// construct virtual address from indexes and offset
#define PGADDR(d, t, o) ((uint)((d) << PDXSHIFT | (t) << PTXSHIFT
| (o)))

// Page directory and page table constants.
#define NPDENTRIES      1024    // # directory entries per page
directory
#define NPTENTRIES      1024    // # PTEs per page table
#define PGSIZE          4096    // bytes mapped by a page
```

To enable paging on xv6 access file mmu.h through this flag

```
#define CR0_PG          0x80000000      // Paging
```

Page replacement:

Page replacement is used when the main memory is fully occupied, so a page must

be replaced to create a room for the required page. It occurs by swapping out the existing

page from the main memory and replace it by the required page from the secondary

memory. Alongside This, when a page referred to by the CPU in the main memory and

it’s not found, a Page Fault occur so the paging replacement occurs in such condition.

The following are few examples of page replacement algorithms:

• First-In-First-Out (FIFO)

• Last Recently Used (LRU)

FIFO algorithm:

FIFO method refers to the First in First Out concept.

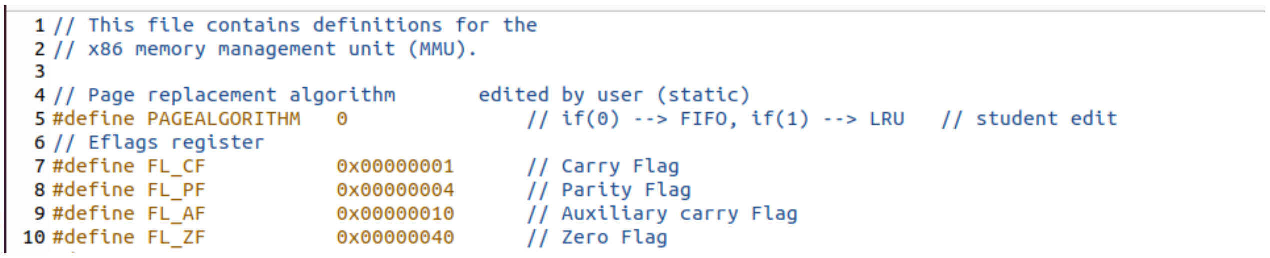
This is the simplest page replacement method in which the operating system maintains

all the pages in a queue. Oldest pages are kept in the front, while the newest is kept at

the end. On a page fault, these pages from the front are removed first, and the pages in

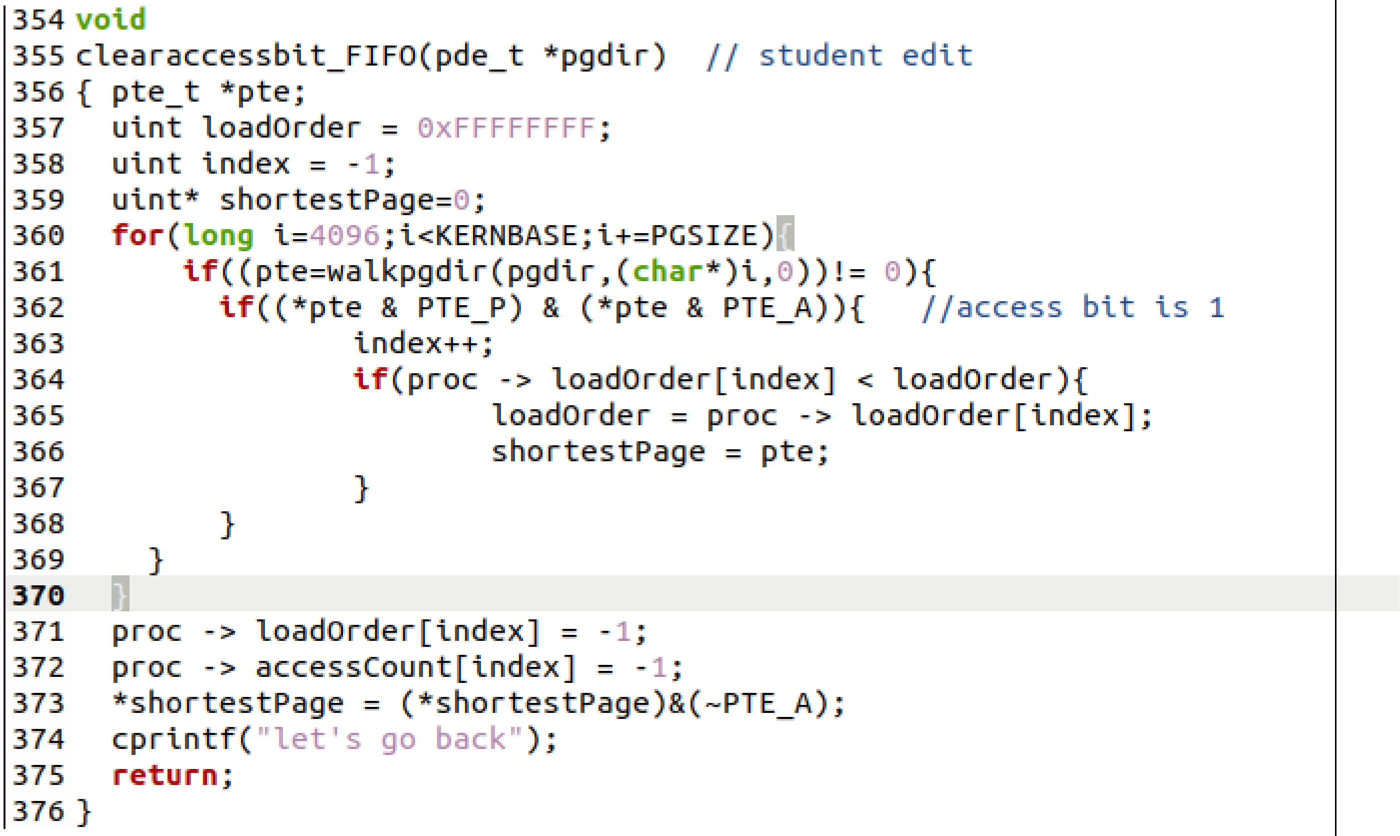
demand are added.

Code



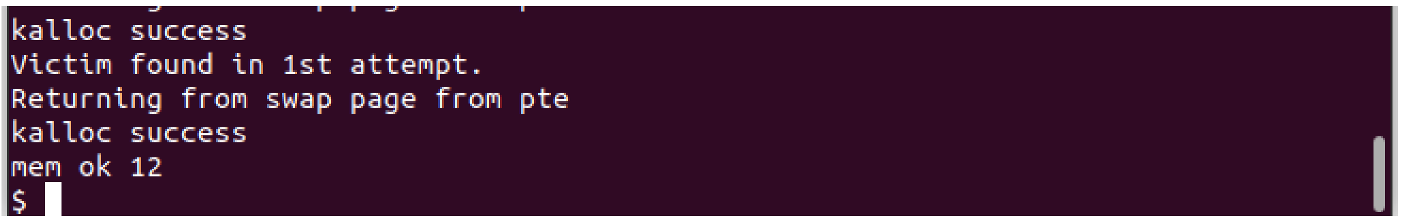
```
 1 // This file contains definitions for the
 2 // x86 memory management unit (MMU).
 3
 4 // Page replacement algorithm      edited by user (static)
 5 #define PAGEALGORITHM   0             // if(0) --> FIFO, if(1) --> LRU   // student edit
 6 // Eflags register
 7 #define FL_CF           0x00000001      // Carry Flag
 8 #define FL_PF           0x00000004      // Parity Flag
 9 #define FL_AF           0x00000010      // Auxiliary carry Flag
10 #define FL_ZF           0x00000040      // Zero Flag
```

Here if the user wants the page replacement algorithm to be FIFO the PAGEALGORITM in line 5 should be kept 0, otherwise if the user wants the algorithm to be LRU he will change the PAGEALGORITHM in line 5 to 1.



```
void
clearaccessbit_FIFO(pde_t *pgdir)  // student edit
{ pte_t *pte;
  uint loadOrder = 0xFFFFFFFF;
  uint index = -1;
  uint* shortestPage=0;
  for(long i=4096;i<KERNBASE;i+=PGSIZE){
      if((pte=walkpgdir(pgdir,(char*)i,0))!= 0){
        if((*pte & PTE_P) & (*pte & PTE_A)){   //access bit is 1
               index++;
               if(proc -> loadOrder[index] < loadOrder){
                      loadOrder = proc -> loadOrder[index];
                      shortestPage = pte;
               }
        }
    }
  }
  proc -> loadOrder[index] = -1;
  proc -> accessCount[index] = -1;
  *shortestPage = (*shortestPage)&(~PTE_A);
  cprintf("let's go back");
  return;
}
```

FIFO test:

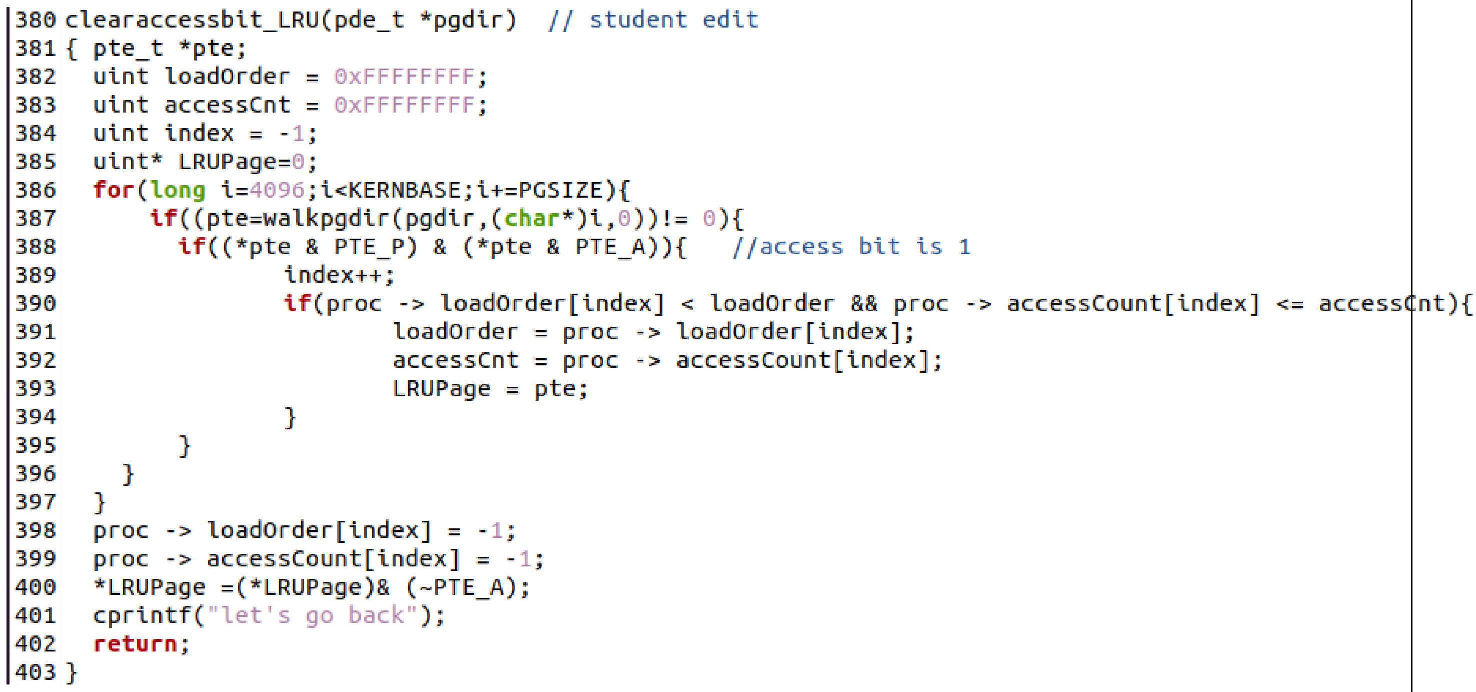


```
kalloc success
Victim found in 1st attempt.
Returning from swap page from pte
kalloc success
mem ok 12
$
```

LRU Algorithm:

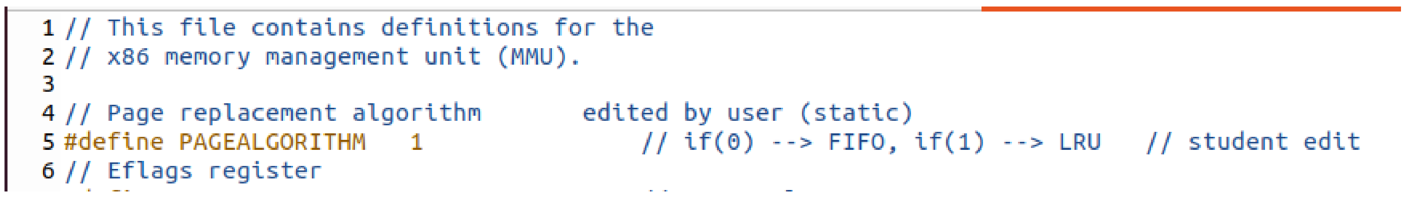
This algorithm is based on the observation that pages that have been heavily used in the last few instructions will probably be heavily used again in the next few. Conversely, pages that have not been used for ages will probably remain unused for a long time. This idea suggests a realizable algorithm: when a page fault occurs, throw out the page that has been unused for the longest time.

Code



```
clearaccessbit_LRU(pde_t *pgdir)  // student edit
{ pte_t *pte;
  uint loadOrder = 0xFFFFFFFF;
  uint accessCnt = 0xFFFFFFFF;
  uint index = -1;
  uint* LRUPage=0;
  for(long i=4096;i<KERNBASE;i+=PGSIZE){
      if((pte=walkpgdir(pgdir,(char*)i,0))!= 0){
        if((*pte & PTE_P) & (*pte & PTE_A)){   //access bit is 1
               index++;
               if(proc -> loadOrder[index] < loadOrder && proc -> accessCount[index] <= accessCnt){
                       loadOrder = proc -> loadOrder[index];
                       accessCnt = proc -> accessCount[index];
                       LRUPage = pte;
               }
        }
    }
  }
  proc -> loadOrder[index] = -1;
  proc -> accessCount[index] = -1;
  *LRUPage =(*LRUPage)& (~PTE_A);
  cprintf("let's go back");
  return;
}
```

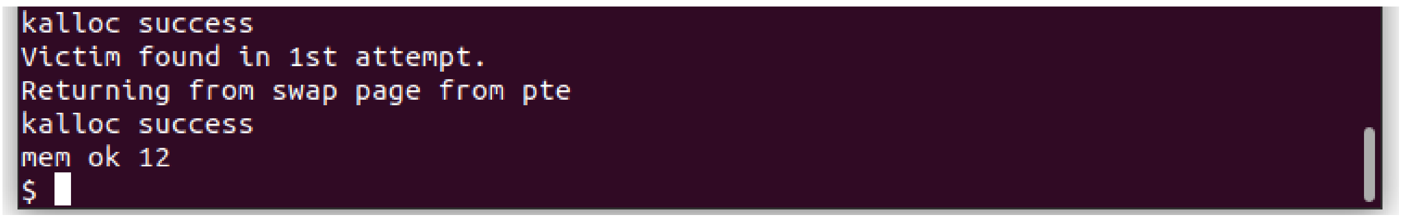
LRU test



```
// This file contains definitions for the
// x86 memory management unit (MMU).

// Page replacement algorithm       edited by user (static)
#define PAGEALGORITHM    1               // if(0) --> FIFO, if(1) --> LRU    // student edit
// Eflags register
```

Here we want to test LRU algorithm so, we changed the PAGEALGORITHM to 1.

Then we run the test to obtain the following :



```
kalloc success
Victim found in 1st attempt.
Returning from swap page from pte
kalloc success
mem ok 12
$ 
```

Modified files:

1) bio.c (for swapping)
• write_page_to_disk()
• read_page_from_disk()
2) fs.c (for swapping)
• balloc_page()
• bfree_page()
3) paging.c (for demand paging)
• swap_page()
• swap_page_from_pte()
4) vm.c (for demand paging)
• select_a_victim()
• clearaccessbitFIFO()
• clearaccessbitLRU()

All edits in XV6 code:

line 53 proc.h #define MAX_PHYC_PN 15 // maximum number of physical pages per process

line 67 proc.h uint loadOrderCounter; // load counter

line 68 proc.h uint loadOrder[MAX_PHYC_PN]; // for fifo

line 69 proc.h uint accessCounter; // access counter for LRU

line 70 proc.h uint accessCount[MAX_PHYC_PN]; // for LRU

line 71 proc.h uint pageFaultCnt = 0;

line 5 mmu.h #define PAGEALGORITHM 0 // if(0) --> FIFO, if(1) --> LRU

line 354 vm.c void clearaccessbit_FIFO(pde_t *pgdir)

line 377 vm.c void clearaccessbit_LRU(pde_t *pgdir)

line 407	**vm.c**	void updatePageCount (struct proc* p)
line 421	**vm.c**	void printState(struct proc *p)
line 83	**paging.c**	if(PAGEALGORITHM) clearaccessbit_LRU(pgdir);
line 84	**paging.c**	else clearaccessbit_FIFO(*pgdir);
line 176	**paging.c**	updatePageCount (*curproc);
line 188	**defs.h**	void clearaccessbit_LRU(pde_t *pgdir);
line 189	**defs.h**	void clearaccessbit_FIFO(pde_t *pgdir);
line 190	**defs.h**	void updatePageCount (struct proc* p);
line 191	**defs.h**	void printState(struct proc *p);
line 194	**defs.h**	int sys_bstat(void);

Requirement 4:

In this section we are not trying to change the whole file system of MINIX3 we are just trying to adjust it to use extents defined by the user instead of blocks. So we need to allocate and free blocks in terms of extents.

User extent size:

For the user to change the extent size he must go to the file const.h in the path \minix-master\minix\fs\ext2 then change the number next to the last constant EXT2_PREALLOC_BLOCKS. The default is 8 and we changed it to 16 just to check the algorithm.

```
#define EXT2_PREALLOC_BLOCKS            8
```

Allocating/freeing consecutive block:

After that we went to the write.c file which is responsible for writing blocks and added a for loop to acquire the desired number of consecutive blocks for our extent. We added the for loop in the new_block Function which allocates/frees consecuitive blocks for our file automatically from the functions alloc_block, free_block which are located in balloc.c.

```
for (int i = 0; i < EXT2_PREALLOC_BLOCKS; i++) {// for loop to return the new number of blocks instead of one

                    if ((b = alloc_block(rip, goal)) == NO_BLOCK) {
                              err_code = ENOSPC;
                              return(NULL);
                    }
                    if ((r = write_map(rip, position, b, 0)) != OK) {
                              free_block(rip->i_sp, b);
                              err_code = r;
                              ext2_debug("write_map failed\n");
                              return(NULL);
                    }
                    rip->i_last_pos_bl_alloc = position;
                    if (position == 0) {
                              /* rip->i_last_pos_bl_alloc points to the block position,
                               * and zero indicates first usage, thus just increment.
                               */
                              rip->i_last_pos_bl_alloc++;
                    }
          }
 }
```