CSE 335 Operating Systems

MINIX 3

OS Course Project

PROJECT DOCUMENTATION

Submitted to:

Prof. Dr. Gamal Abdel Shafy

Eng. Sally Shaker

Submitted by:

Youssef George Fouad	19P9824
Mostafa Nasrat Metwally	19P4619
Kerollos Wageeh Youssef	19P3468
Anthony Amgad Fayek	19P9880

Table of Contents

List of figures

0. INTRODUCTION

The course project aims to discover Minix 3 Operating System in terms of internal structure, components, and algorithms used in each of them. In addition, this document covers modifying some of the internal structure of MINIX 3.2 to implement additional algorithms and compare different CPU scheduling algorithms, file management, and memory management types.

1. MINIX OS

MINIX 3 is a free open-source operating system based on a small (about 12K lines of code) microkernel that runs in kernel mode. The rest of the operating system runs as a collection of server processes, each one protected by the hardware MMU (Memory Management Unit). These processes include the virtual file system, one or more actual file systems, the memory manager, the process manager, the reincarnation server, and the device drivers, each one running as a separate user-mode process.

1.1. Internal Structure of MINIX 3

Like other operating system, MINIX 3 is divided in 4 well-defined layers with the kernel layer working in the kernel mode as the bottom most layer below the 3 user mode layers: device drivers layer, server processes layer, and the user processes layer. Figure 1 helps us visualize the structure of MINIX 3.

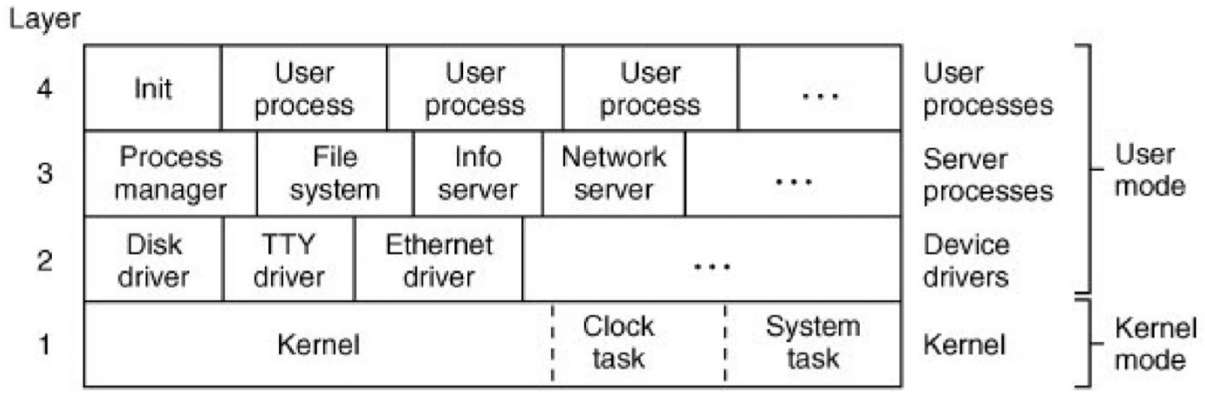


Figure 1: MINIX 3 Structure

1.2. Kernel Layer

Layer 1 running in the kernel mode consists of 3 modules. Kernel module is responsible for processes management to schedule them and handle messages passing between these processes, which uses privileged kernel mode instructions that are not available to the other 3 layers. Kernel instructions (calls) are low-level functions from the system task allowing the drivers and servers to work properly. Moreover, the clock task module generates timing signals for the system, but it is not available for communication with user process as it only works with the kernel through its interfaces. Last but not least, system task module implements a set of kernel calls to be provided to the drivers and servers in the higher level.

1.3. Devices Drivers Layer

Layer 2 has the most privileges out of layers 2,3,4 running in user mode. Processes in this layer can let the system tasks write/ read data to I/O ports. In addition, device drivers can request newly read data to be copied to the address of a different process, besides other kernel calls this layer can call.

1.4. Server Processes Layer

Layer 3 containing servers and modules that work as service providers to users' processors as the process manager (PM) available in this layer calls the operating system calls that manage all processes lifecycle from forking, executing to exiting. Furthermore, File system (FS) using all system calls responsible for file management including, but not limited to, *read*, *chdir* (change directory), and *mount*.

1.5. User Processes Layer

Layer 4 includes all the user processes and programs beside background processes, known as, daemons. Moreover, system processes that runs at all times are also considered to be in this layer. For instance, **init** process acts as the root process for all the process in the processes tree of MINIX 3 operating system.

2. CPU SCHEDULING

In the second requirement of the project, we should modify MINIX 3 to include all of the following CPU scheduling algorithms, where the user can choose which one of them to be used through a configuration file. In addition, results of running these algorithms on a set of processes should be discussed after the calculation of average turnaround and waiting time of each process. The four CPU scheduling algorithms to be implemented are:

- Round Robin
- Shortest Job First (SJF)
- Priority based
- Multi-Level Feedback Queue

2.1. CPU Scheduling in MINIX 3

In MINIX 3, CPU scheduling is one of the responsibilities of the "Process manager" module running in layer 3 of the OS internal structure, as shown in (*Figure 1: MINIX 3 Structure*), running in user space and implemented in the folder "/minix/servers/sched". Meanwhile, the user processes themselves run in the highest layer (layer 4) of the operating system structure.

MINIX 3, by default, has a multilevel queuing system with 16 queues, where processes are classified among them as follows:

1- System clock and system task run in the queue with the highest priority.
2- Drivers' processes have the second highest priority.
3- Server processes comes after the drivers in priority.
4- User processes run in queues of several higher priority level than idle ones but lower queues than those of the servers.
5- Idle processes run in the lowest priority queue.

Normally, not all 16 queues are normally working as some of them are kept for cases where additional drivers / servers are added or for experimental purposes. In addition, a process can be moved to a different queue by the OS or by the user, within certain limits.

2.2. Round Robin

Round Robin is one of the oldest, simplest, and widely used preemptive scheduling algorithms. Each process is given a time period, as known as a quantum, to run before preempting the CPU and giving it to the next process in the ready queue, in addition to putting the preempted process at the end of the ready queue. If the process finishes execution, or is blocked, before the end of its quantum, the context switching takes place in order to start the quantum of the next ready process in the queue.

2.2.1. Advantages

- High fairness as all processes utilizes the CPU for equal periods before continuing.

2.2.2. Disadvantages

- If quantum time is relatively small, then the time needed for context switch between processes will take up most of the time slice thus reducing CPU utilization.
- Average waiting time is usually the highest among the scheduling algorithms.

2.2.3. RR in MINIX 3

Each of the previously mentioned queues run in a “Round-Robin” manner and have a different quantum, where the quantum of the queue increases as the priority of the queue increase. Therefore, the lowest queue has the minimum quantum, and the highest queue has the maximum quantum.

Also, the drivers and servers should normally run until they block or finish execution but in MINIX, they are preemptable, given a large but limited quantum to prevent the multifunction of the system (deadlock).

2.3. Shortest Job First (SJF)

SJF is one of the non-preemptive scheduling algorithms. CPU scheduler chooses the process that needs the least time to execute. Practically, the scheduler does not know the process execution time before it actually finishes execution; therefore, an approximation can be calculated for the execution time of the process before it actually executes using the following equation:

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\ \tau_n$$

Where:

t_n: actual length of the n^{th} CPU burst

α: a parameter set between 0 and 1, commonly set to 0.5

τ_{n+1}: predicted value for the next CPU burst

2.3.1. Advantages

- Average waiting time is usually the least among the scheduling algorithms.

2.3.2. Disadvantages

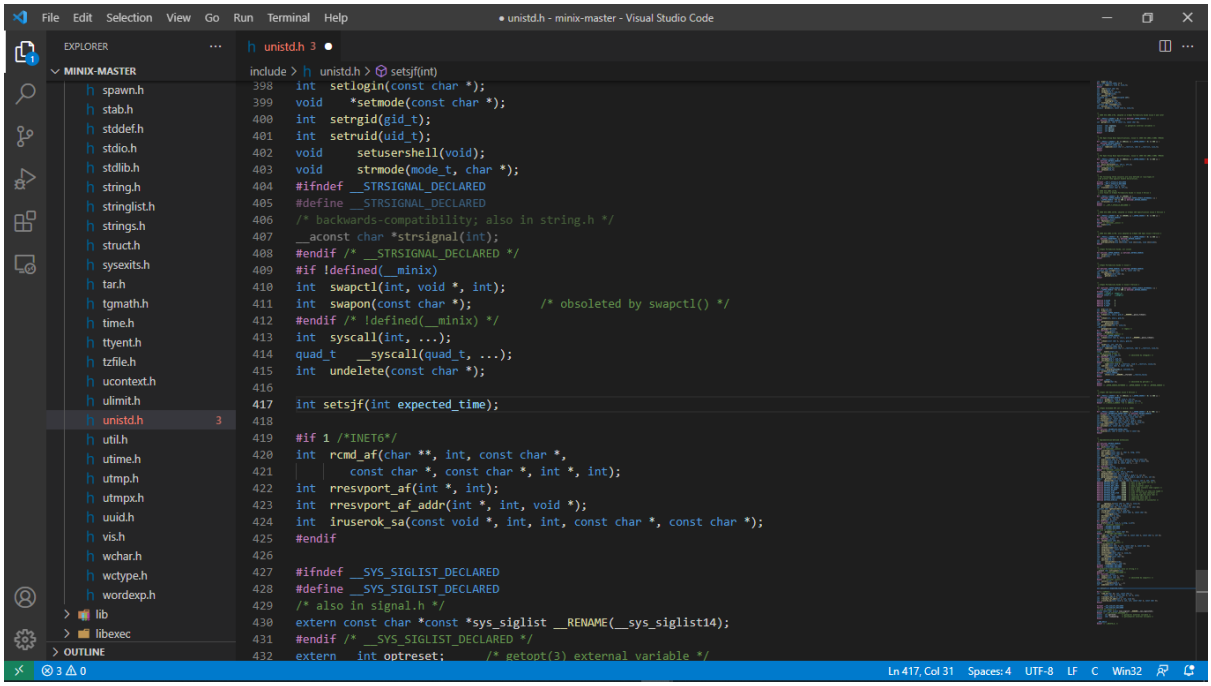
- Might cause starvation of long processes in case shorter processes keeps entering the queue.
- Difficult to predict the next CPU burst length.

2.3.3. Implementing SJF in MINIX 3

As discussed earlier, MINIX 3 does not originally have a SJF scheduling algorithm implemented in the operating system; therefore, we implemented the SJF algorithm to run user processes in queues x, x, x by editing the code as follows:

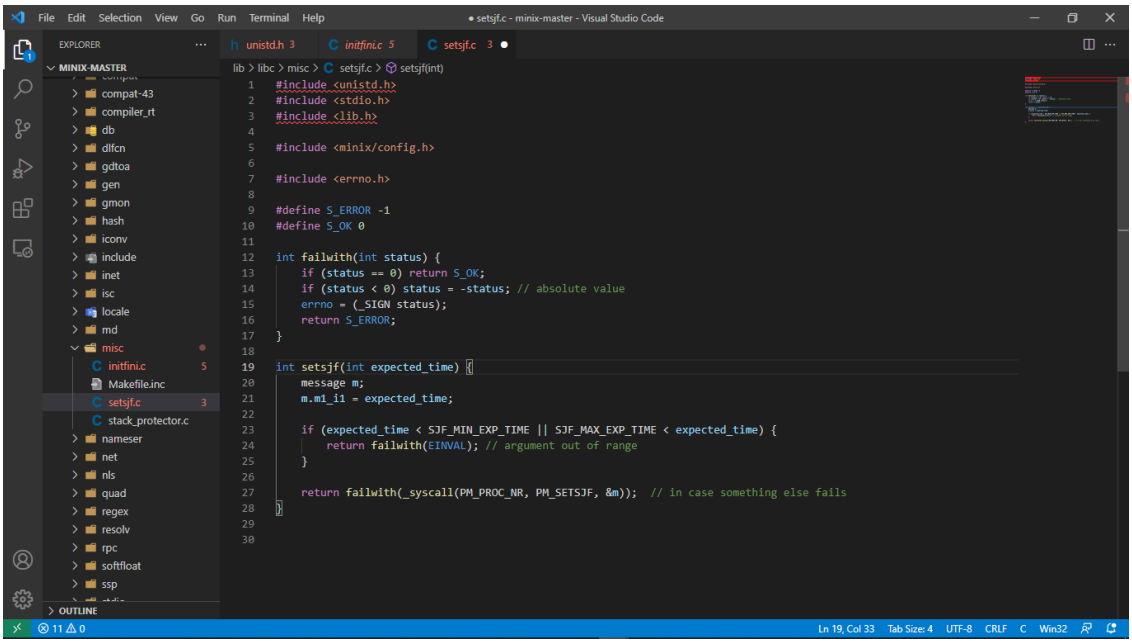
1. In the file “unistd.h” located in “/include”

Line 417: We put the prototype of setsjf function which validates the expected time.



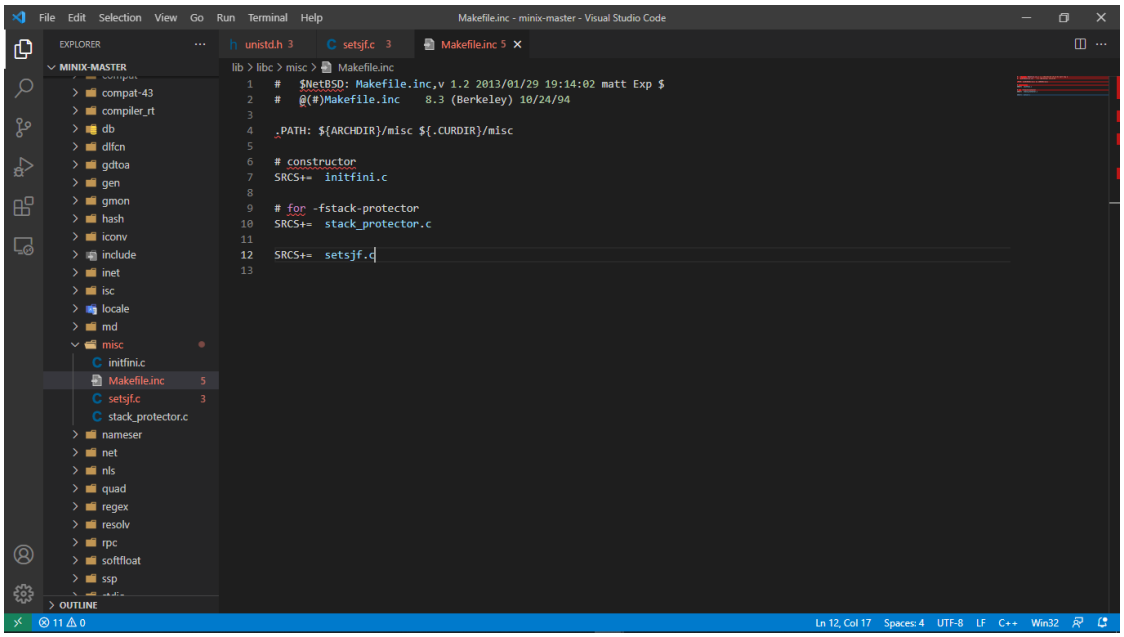
2. Add file “setsjf.c” located in “/lib/libc/misc”

All Lines: We write 2 functions to validate the expected time within the bounds or not.



3. In the file “Makefile” located in “/lib/libc/misc”

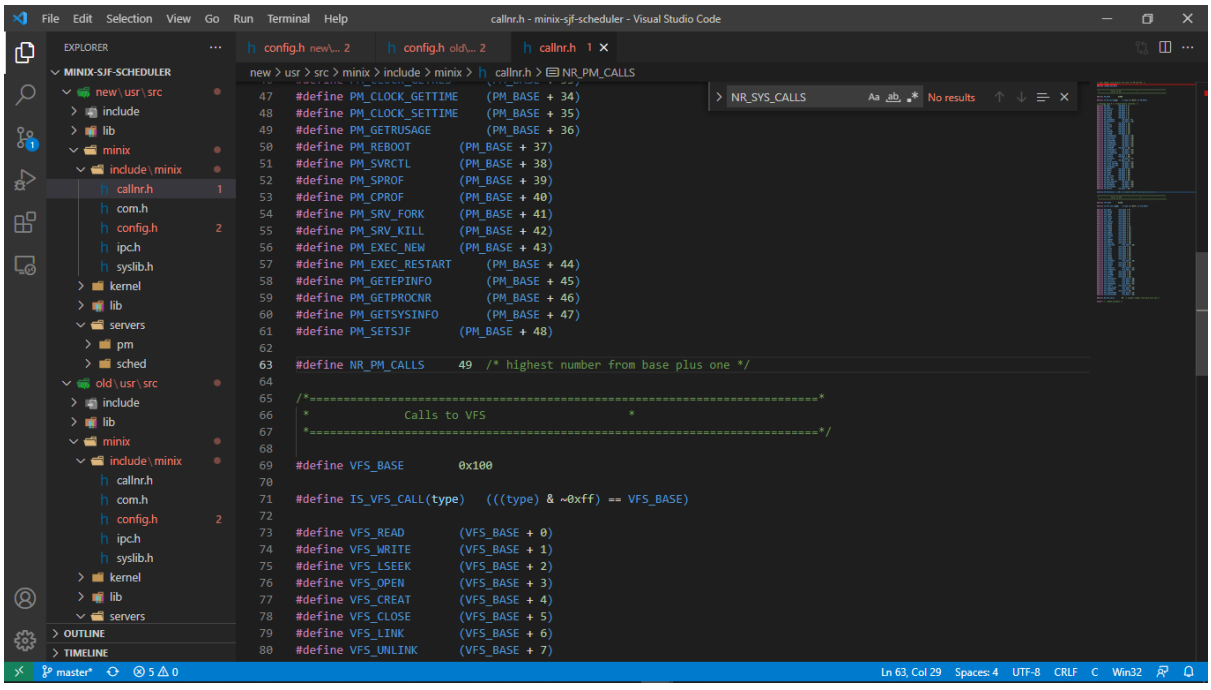
Line 12: Add setsjf.c in the Makefile.



4. In the file “callnr.h” located in “/minix/include/minix”

Line 61: add new process manager call (PM_SETSJF).

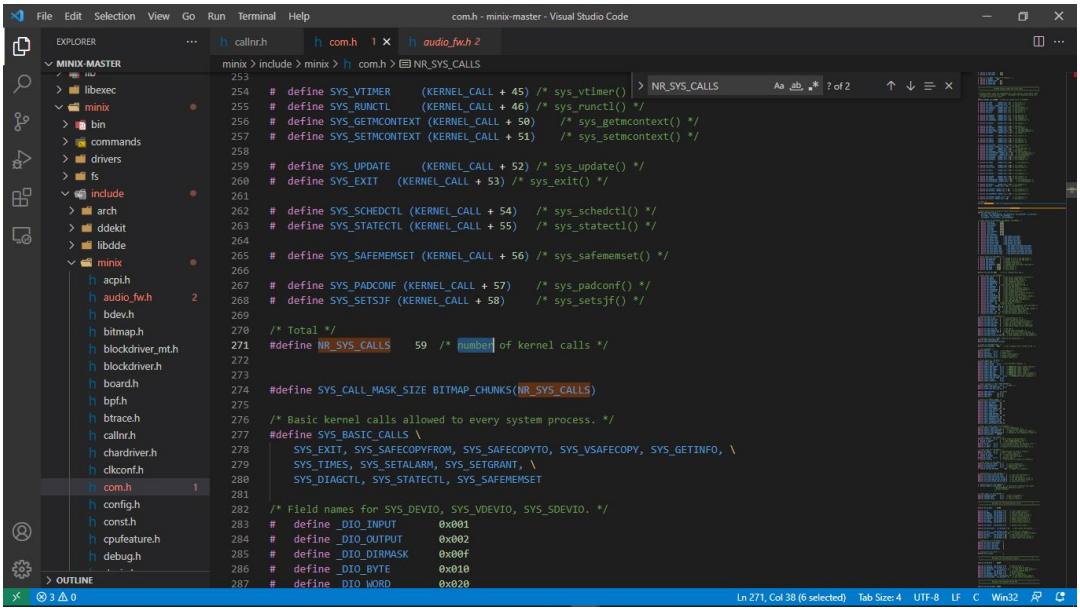
Line 63: increment the process manager number of calls by 1.



5. In the file “com.h” located in “/minix/include/minix”

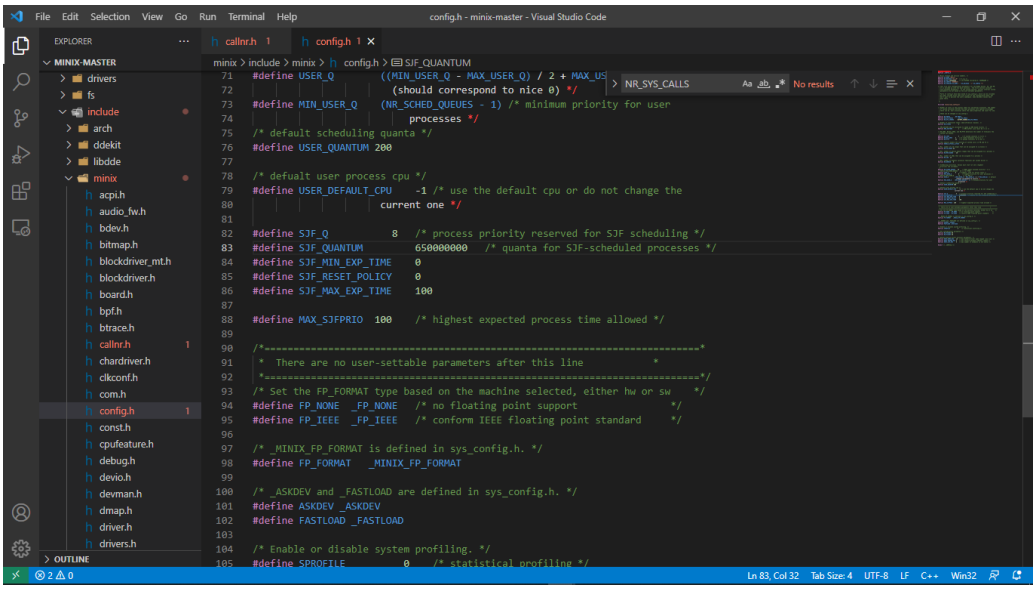
Line 268: add new kernel (system) call (PM_SETSJF).

Line 271: increment the kernel number of calls by 1.



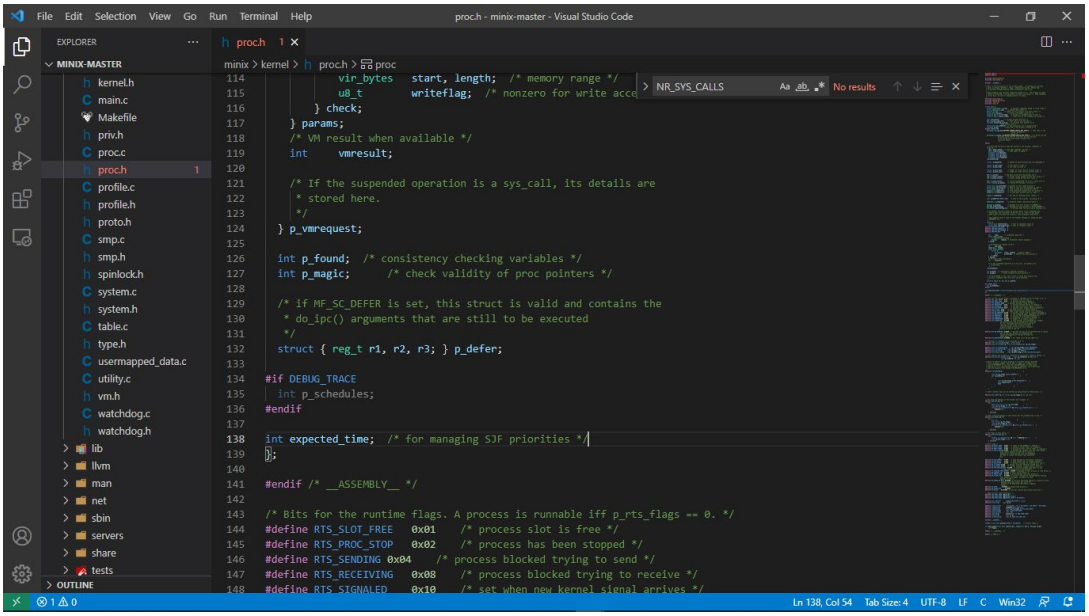
6. In the file “config.h” located in “/minix/include/minix”

Lines 82 - 88: add some definitions for the shortest job first, such as: the queue number 8, the quantum of this queue and minimum and maximum expected time.



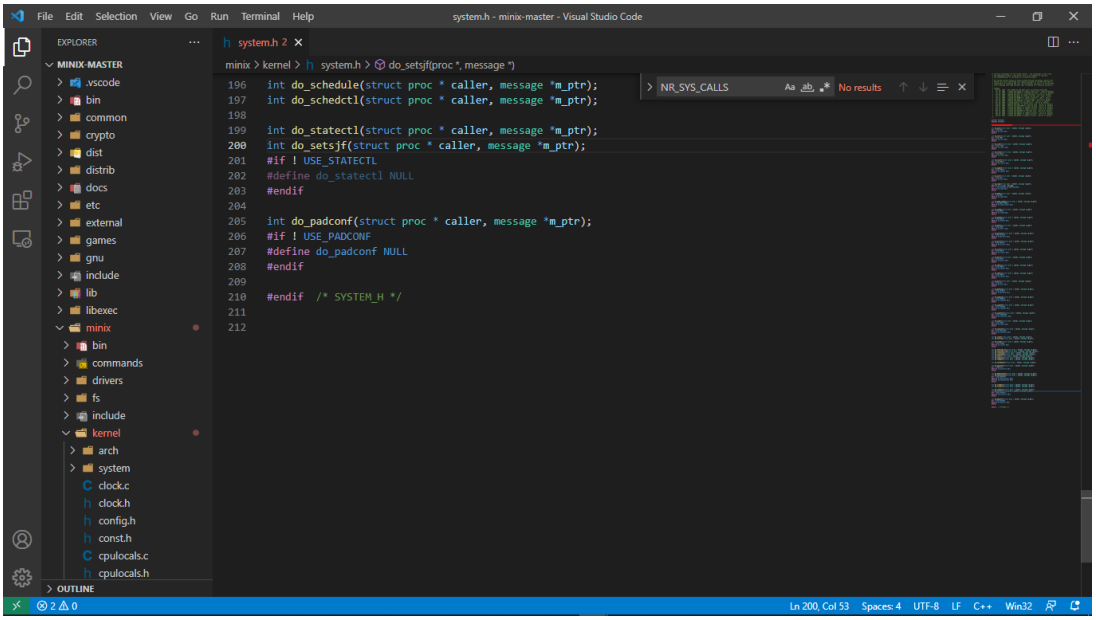
7. In the file “proc.h” located in “/minix/kernel”

Line 138: add expected time attribute for the process struct.



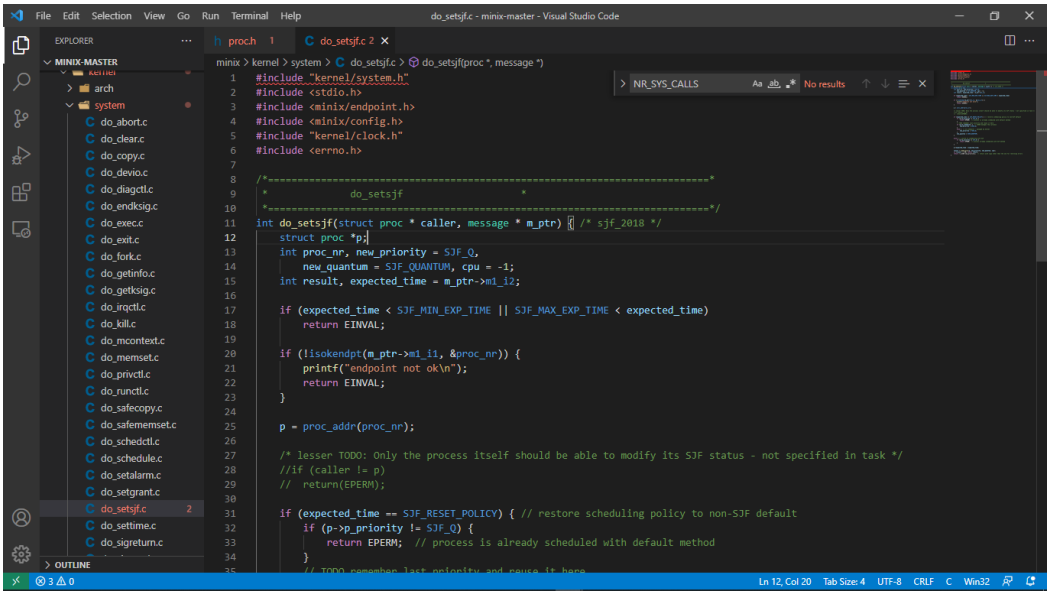
8. In the file “system.h” located in “/minix/kernel”

Line 200: add the prototype of do_setsjf.



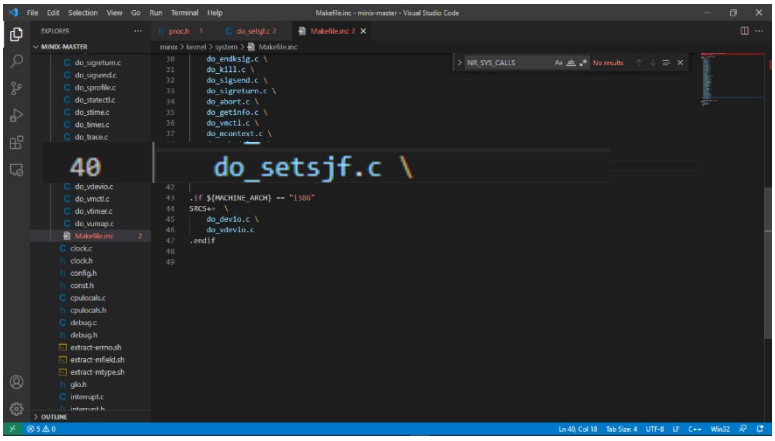
9. In the file “do_setsjf.c” located in “/minix/kernel/system”

All lines: this is the function that the kernel call will execute. This function assigns the process to shortest job first queue and give it quantum.



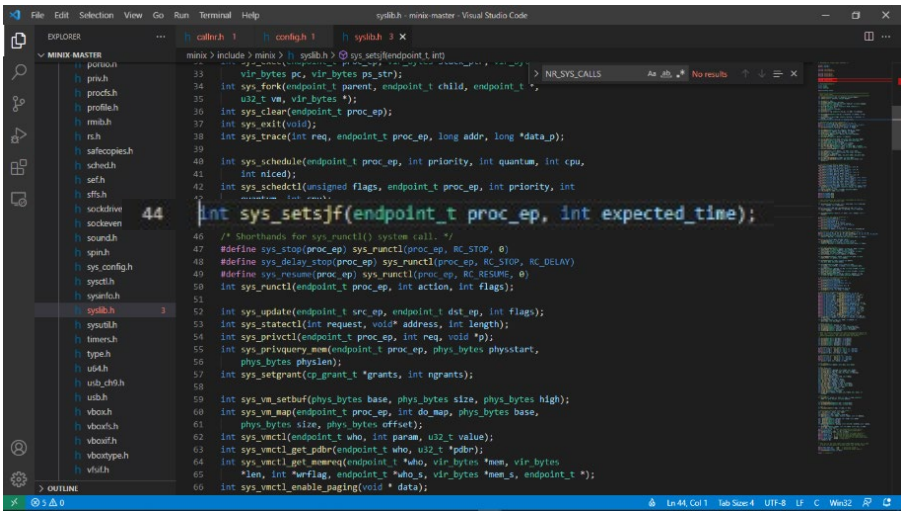
10. In the file “Makefile” located in “/minix/kernel/system”

Line 40: add do_setsjf.c to the Makefile to be compiled.



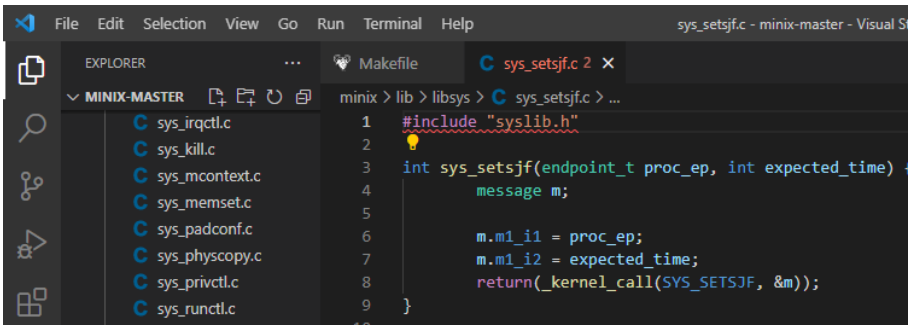
11. In the file “syslib.h” located in “/minix/include/minix”

Line 44: add the prototype of sys_setsjf() which is the new system call.



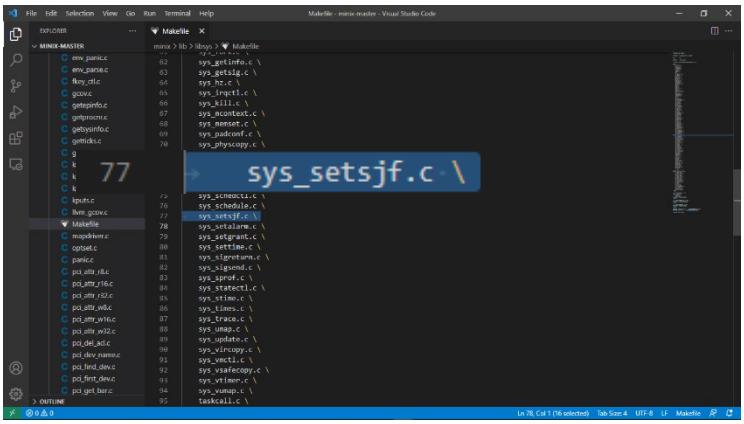
12. the file “sys_setsjf.c” located in “/minix/lib/libsys”

All Lines: write the implementation of the system call sys_setsjf.



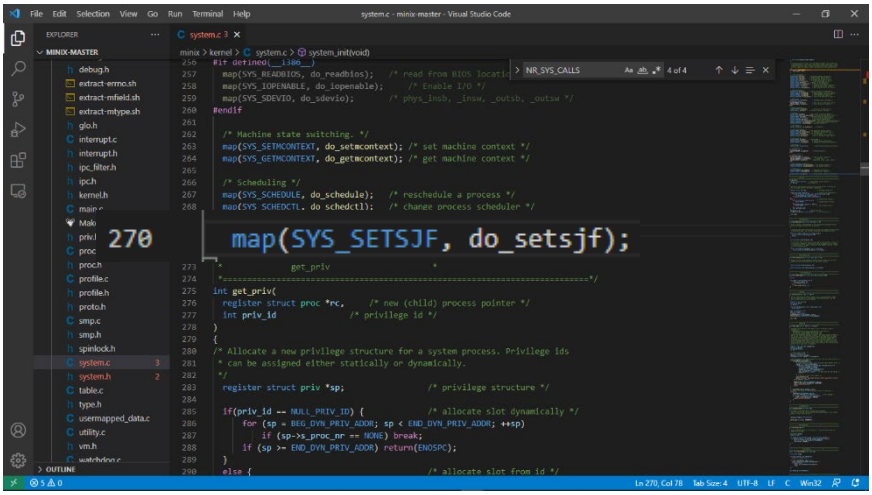
13. the file “Makefile” located in “/minix/lib/libsys”

Line 77: add sys_setsjf.c to the Makefile to be compiled.



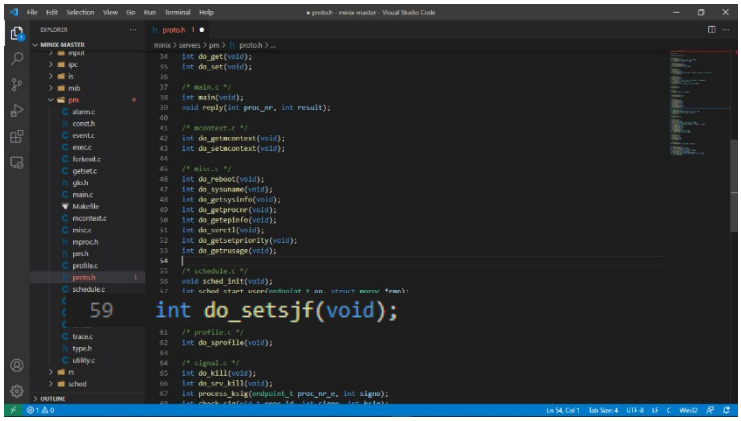
14. the file “system.c” located in “/minix/kernel”

Line 270: map between the sys_setsjf (system call) and do_setsjf function.



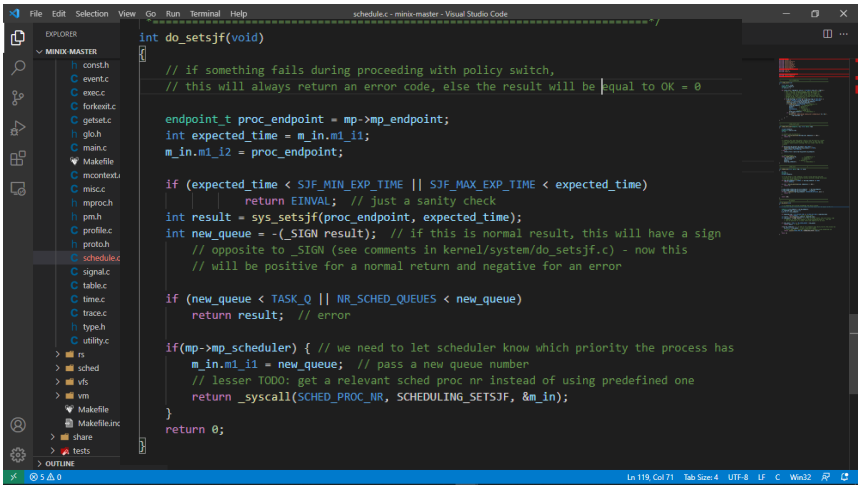
15. the file “proto.h” located in “/minix/servers/pm”

Line 59: add do_setsjf prototype to of the process manager.



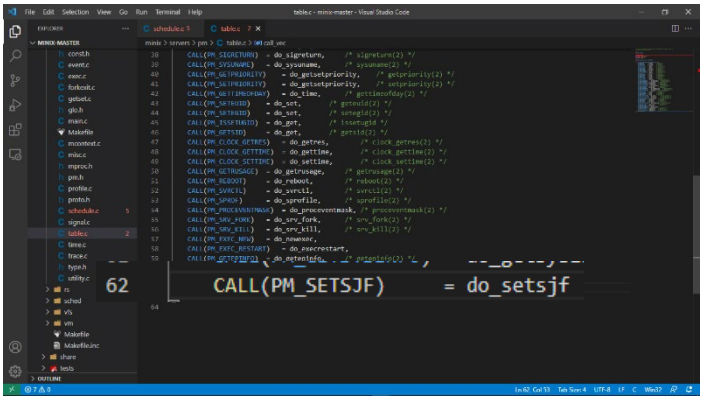
16. the file “schedule.c” located in “/minix/servers/pm”

Lines 116 - 141: add the implementation of do_setsjf to the schedule of the process manager to keep track of the process and call the system call made for the SJF, then return a message for the scheduler to set SJF.

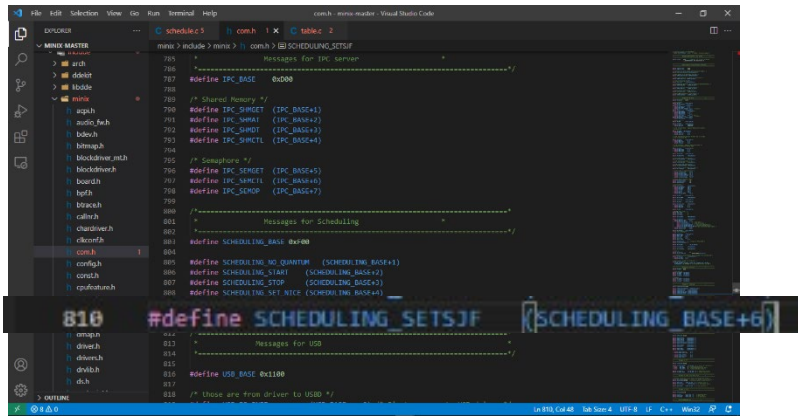


17. the file “table.c” located in “/minix/servers/pm”

Line 62: map do_setsjf implemented in schedule.c to the process manager call on step 4.

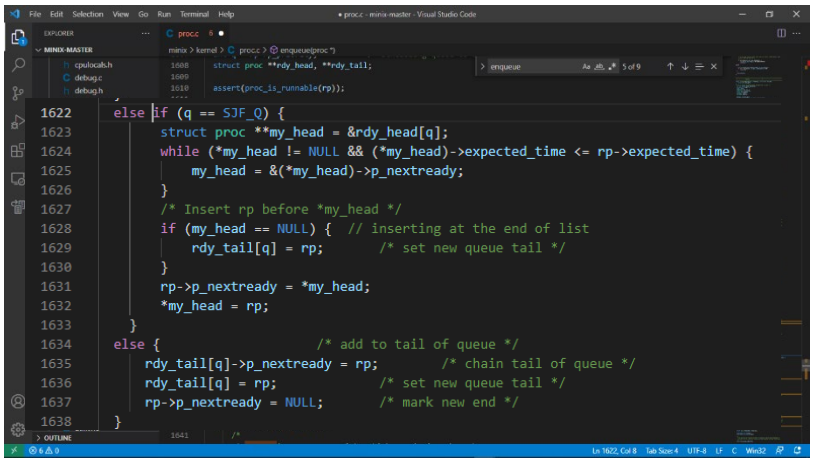


18. the file “com.h” located in “/minix/include/minix”Line 810: add a message to the scheduling messages for SJF which is used in step 16 .



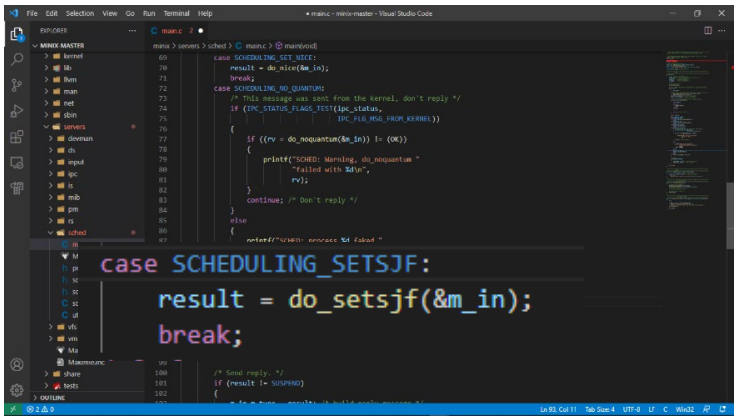
19. the file “proc.c” located in “/minix/kernel”

Lines 1622 - 1633: for a new process started the function enqueue() is called. If the queue of this process is SJF queue then search for its appropriate index in the ready queue to be allocated into it.



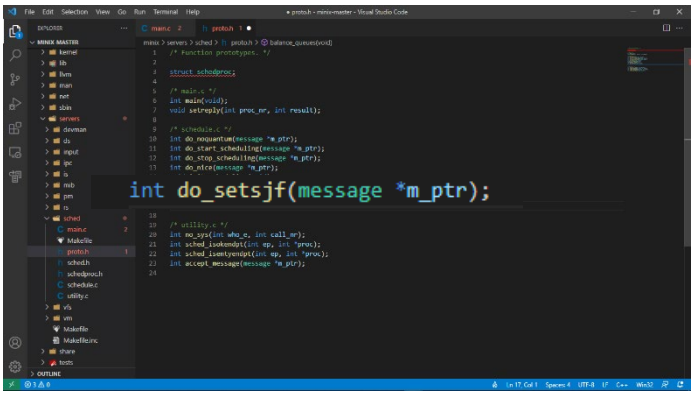
20. the file “main.c” located in “/minix/servers/sched”

Lines 93 - 95: in case of SJF message is sent to the scheduler from the process manager, do_setsjf in schedule.c is called.



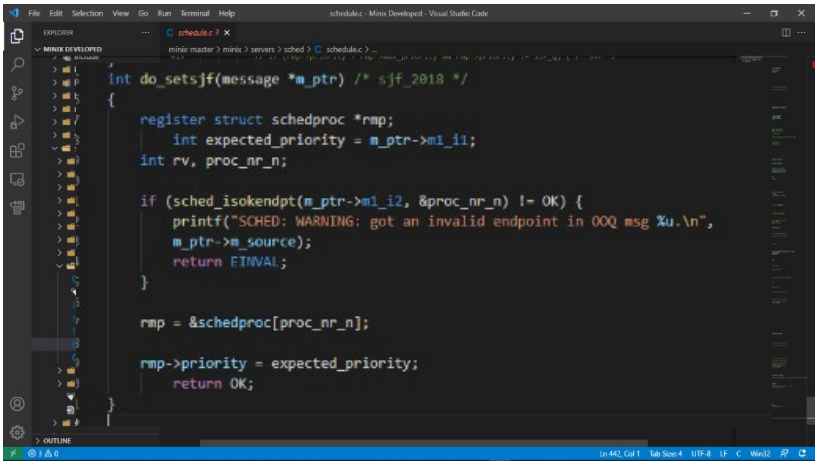
21. the file “proto.h” located in “/minix/servers/sched”

Line 16: add do_setsjf prototype to the scheduling functions prototypes.



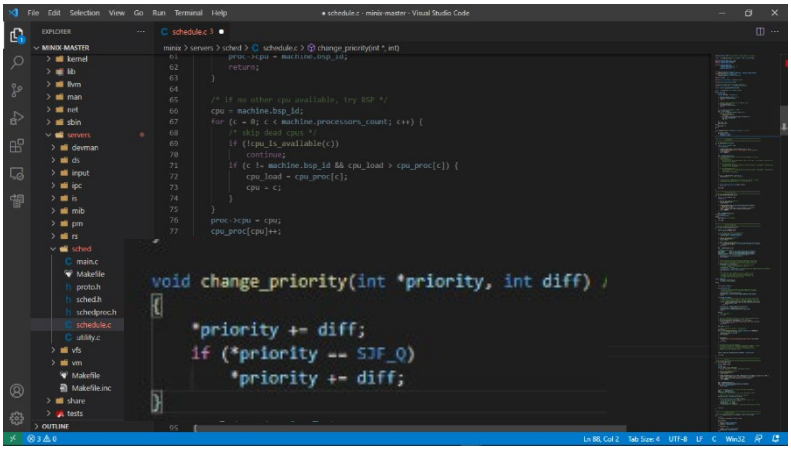
22. the file “schedule.c” located in “/minix/servers/sched”

Lines 425 - 441: verifies the message sent as an attribute with the function.



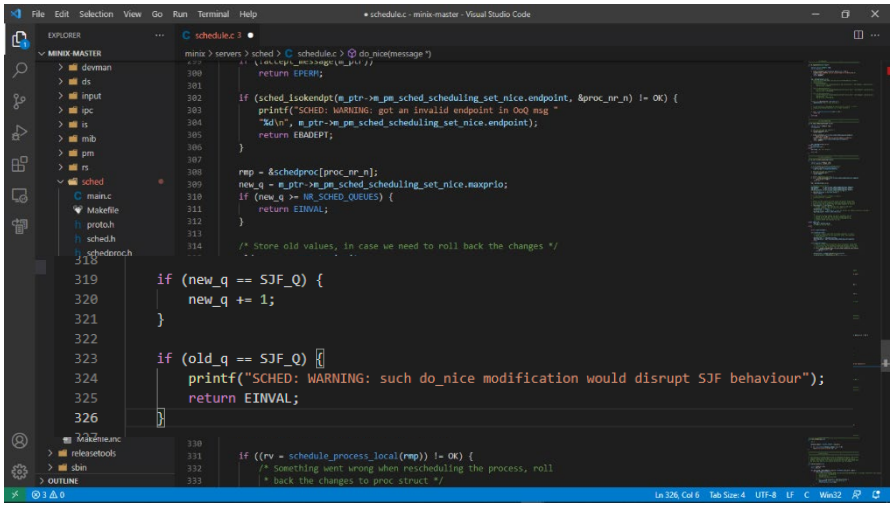
23. the file “schedule.c” located in “/minix/servers/sched”

Lines 83 - 88: add whenever need to change the queue don’t assign a process to SJF queue unless it was assigned to it. This was done by implementing change priority function which skips the SJF q.



24. the file “schedule.c” located in “/minix/servers/sched”

Lines 319 - 326: in do_nice function, if the new queue is SJF queue, then skip it and assign the next queue to be the new queue and if the old queue is the SJF queue, then don’t change the queue and return INVALID.



2.4. Priority-Based Scheduling

A scheduling algorithm where each process is assigned a priority to it on initialization, where the scheduler chooses the process with the highest priority in the ready queue to execute. One of the main problems associated with priority scheduling is starvation, where processes with the least priorities wait in the queue indefinitely and can never execute. There are many solutions concerned with preventing starvation of processes in a priority-based scheduling queue. For instance, one solution is aging which is increasing the priority of the ready processes that are not currently running.

2.4.1. Advantages

- Provides an algorithm by which the OS can execute processes with higher importance more quickly.
- Provides the easiest way to implement a scheduling algorithm.

2.4.2. Disadvantages

- Might cause starvation of less important processes in case processes of higher importance keeps entering the queue.

2.4.3. Implementing priority scheduling in MINIX 3

Implementing priority-based scheduling algorithm in MINIX 3, we only had to disable the movement of user processes between the user processes' queues. Therefore, we edited the source code of the operating system as follows:

- In file "schedule.c" located in "/minix/servers/sched/"

Lines 106-109: We commented these lines to prevent the feedback behavior of the scheduling algorithm. Thus, every process enters the scheduler would have a specific priority, therefore specific queue, with priority of 7 for user processes. That will make a priority queue by running the higher priority processes (the least value) before the lower priority processes. To prevent starvation, every constant number of clock cycles the processes which didn't finish will increase its priority by 1.

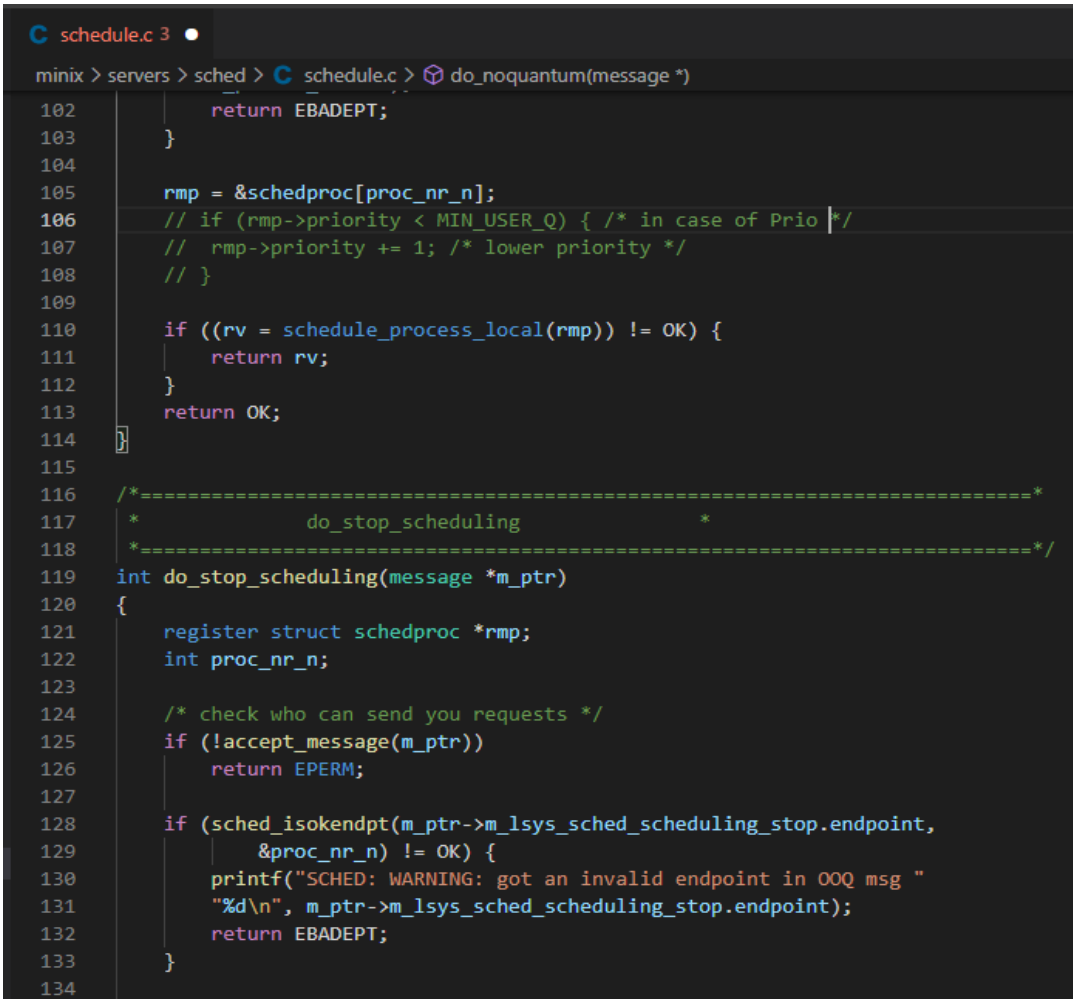


Figure 2: Priority Scheduling Code

2.5. Multi-Level Feedback Queue

A scheduling system where processes can move between several queues. A new process is assigned the highest priority; therefore, it is put in the top queue. When the process reaches the lowest queue and finishes its quantum there, it is moved again to the highest priority queue where it started initially. A multilevel feedback queue scheduling algorithm is defined according to several parameters, which are

- Number of queues
- Scheduling algorithm used in each queue
- How a process is upgraded
- How a process is demoted

2.5.1. Implementing MFQ in MINIX 3

Implementing the MFQ system in MINIX 3, we edited several files of the OS source code as follows.

1- In file “config.c” located in “/minix/include/minix/”

Lines 66-68: We changed the number of queues from 16 to 19 to define 3 extra queues (16-18) where all the user processes can execute in a MFQ manner in these queues.

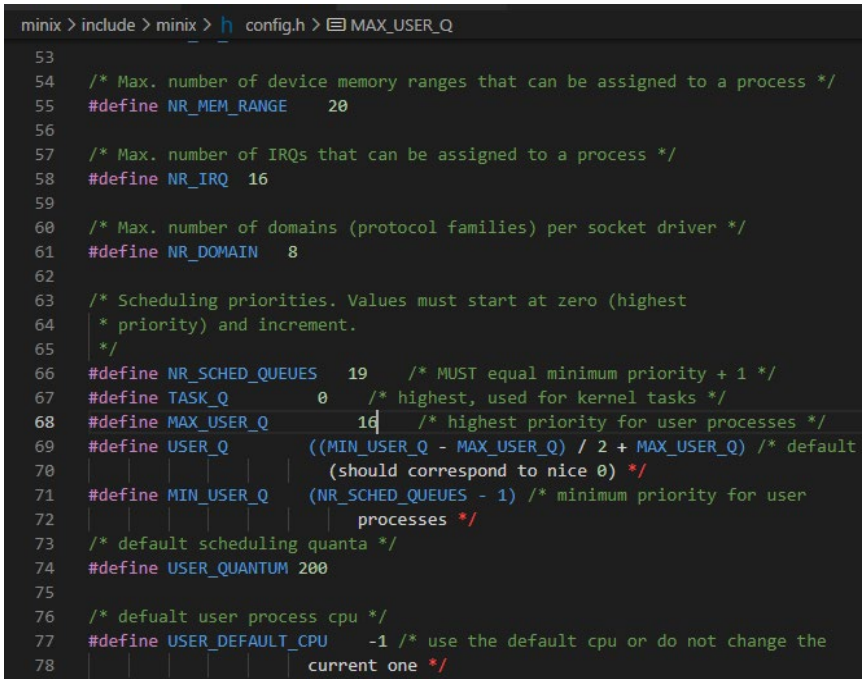


Figure 3: MFQ queues 16-18 definition

2- In file “schedproc.h” located in “/minix/servers/sched”

Line 33: We defined “quantum” of type “unsigned”, which will be assigned the quantum value used in the MFQ algorithm and calculate the amount of time a process spent in a queue.

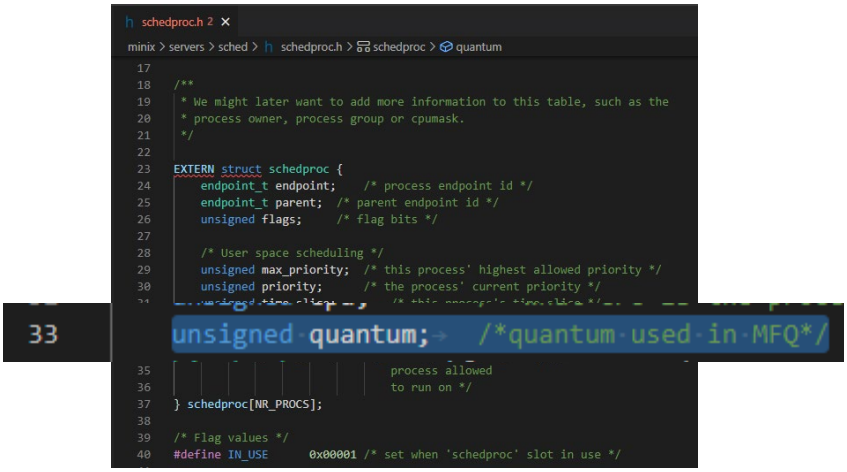


Figure 4: MFQ quantum

3- In file “schedule.c” located in “/minix/servers/sched/”

Lines 99-114: if the process is in the 3 queues of the MFQ, we print the quantum and its priority. If the quantum is 5 then the process enters the next queue of priority 17. If quantum is 10, then the process enters the next queue of priority 18. If the quantum is 1, then the process enters the queue of priority 16. Thus, the process spends 5 quantum in queue number 16, 10 quantum in queue number 17 and 15 quantum in queue number 18.


```
91
92      if (sched_isokendpt(m_ptr->m_source, &proc_nr_n) != OK) {
93          printf("SCHED: WARNING: got an invalid endpoint in OOQ msg %u.\n",
94          m_ptr->m_source);
95          return EBADEPT;
96      }
97
98      rmp = &schedproc[proc_nr_n];
99      if(rmp->priority>=MAX_USER_Q && rmp->priority<=MIN_USER_Q){
100         rmp->quantum+=1;
101         if(rmp->quantum==5){
102             printf("Process %d consumed Quantum 5 and Priority %d\n", rmp->endpoint, rmp->priority);
103             rmp->priority = USER_Q;
104         }
105         else if(rmp->quantum==15) {
106             printf("Process %d consumed Quantum 10 and Priority %d\n", rmp->endpoint, rmp->priority);
107             rmp->priority = MIN_USER_Q;
108         }
109         else if(rmp->quantum==35) {
110             printf("Process %d consumed Quantum 20 and Priority %d\n", rmp->endpoint, rmp->priority);
111             rmp->quantum = 0;
112             rmp->priority = MAX_USER_Q;
113         }
114     }
115     else if (rmp->priority < MIN_USER_Q) {
116         rmp->priority += 1; /* lower priority */
117     }
118
119     if ((rv = schedule_process_local(rmp)) != OK) {
120         return rv;
121     }
122     return OK;
123 }
```

Figure 5: MFQ Code 3

4- In file “schedule.c” located in “/minix/servers/sched/”

Lines 165: We initialize the priority of the process to be 0.

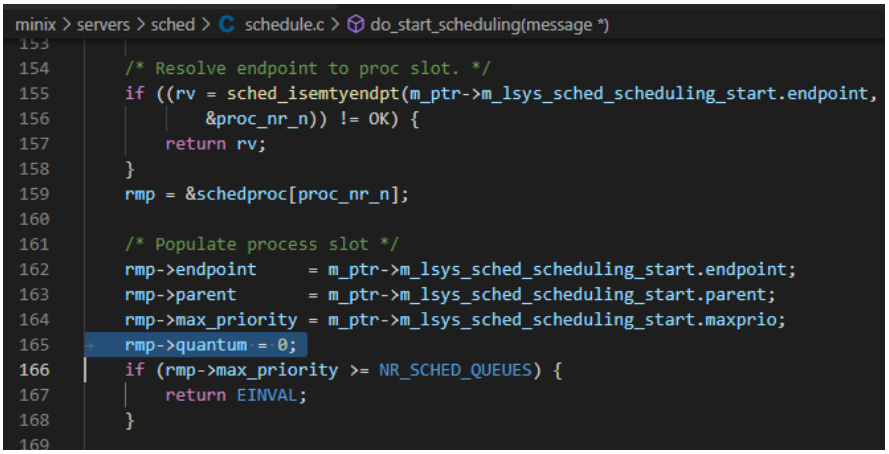


Figure 6: MFQ Code 4

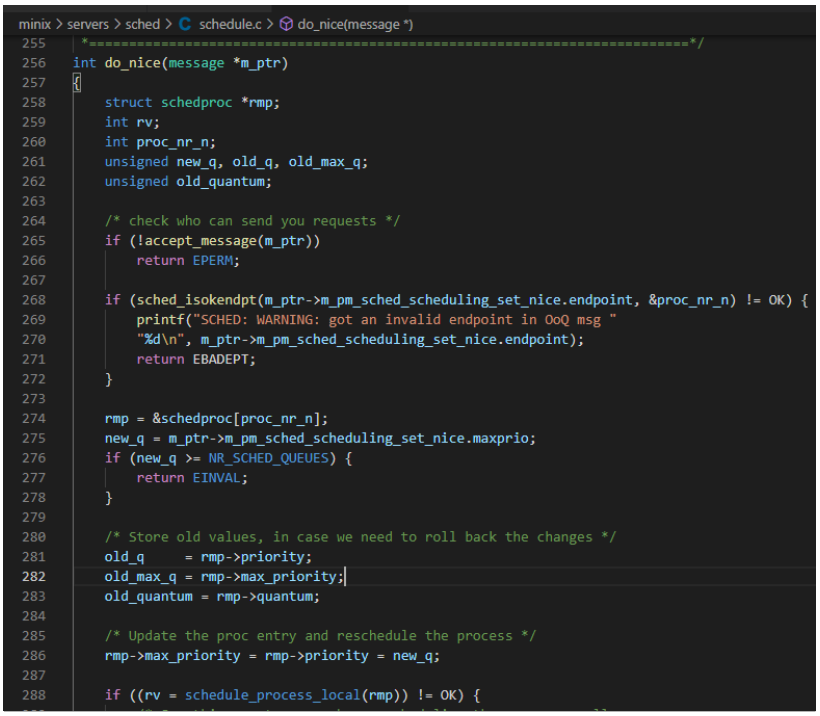
5- In file “schedule.c” located in “/minix/servers/sched/”

Line 176: We put the process initially at the first queue (queue number 16).

Figure 7: MFQ Code 5

6- In file “schedule.c” located in “/minix/servers/sched/”

Lines x-x: We edited the function “do_nice()” in order to assign previous priority values in case we cannot assign new priorities to the processes in queues 16-18.

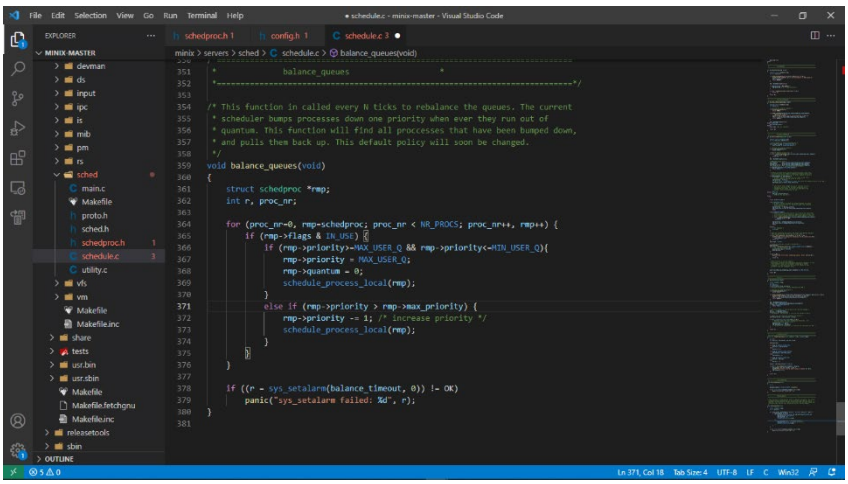


```
minix > servers > sched > schedule.c > do_nice(message *)
255   *===========================================================================*/
256  int do_nice(message *m_ptr)
257  {
258      struct schedproc *rmp;
259      int rv;
260      int proc_nr_n;
261      unsigned new_q, old_q, old_max_q;
262      unsigned old_quantum;
263
264      /* check who can send you requests */
265      if (!accept_message(m_ptr))
266          return EPERM;
267
268      if (sched_isokendpt(m_ptr->m_pm_sched_scheduling_set_nice.endpoint, &proc_nr_n) != OK) {
269          printf("SCHED: WARNING: got an invalid endpoint in OoQ msg "
270          "%d\n", m_ptr->m_pm_sched_scheduling_set_nice.endpoint);
271          return EBADEPT;
272      }
273
274      rmp = &schedproc[proc_nr_n];
275      new_q = m_ptr->m_pm_sched_scheduling_set_nice.maxprio;
276      if (new_q >= NR_SCHED_QUEUES) {
277          return EINVAL;
278      }
279
280      /* Store old values, in case we need to roll back the changes */
281      old_q     = rmp->priority;
282      old_max_q = rmp->max_priority;
283      old_quantum = rmp->quantum;
284
285      /* Update the proc entry and reschedule the process */
286      rmp->max_priority = rmp->priority = new_q;
287
288      if ((rv = schedule_process_local(rmp)) != OK) {
```

Figure 8: MFQ do_nice()

7- In file “schedule.c” located in “/minix/servers/sched/”

Lines 359-380: We edited the function “balance_queues()” to rescheduling a process after N clock ticks assigning it to the queue number 16 and reset its quantum.



```
/* This function in called every N ticks to rebalance the queues. The current
 * scheduler bumps processes down one priority when ever they run out of
 * quantum. This function will find all proccesses that have been bumped down,
 * and pulls them back up. This default policy will soon be changed.
 */
void balance_queues(void)
{
    struct schedproc *rmp;
    int r, proc_nr;

    for (proc_nr=0, rmp=schedproc; proc_nr < NR_PROCS; proc_nr++, rmp++) {
        if (rmp->flags & IN_USE) {
            if (rmp->priority>=MAX_USER_Q && rmp->priority<=MIN_USER_Q){
                rmp->priority = MAX_USER_Q;
                rmp->quantum = 0;
                schedule_process_local(rmp);
            }
            else if (rmp->priority > rmp->max_priority) {
                rmp->priority -- 1; /* increase priority */
                schedule_process_local(rmp);
            }
        }
    }

    if ((r = sys_setalarm(balance_timeout, 0)) != OK)
        panic("sys_setalarm failed: %d", r);
}
```

Figure 9 MFQ balance_queues()

2.5.2. Testing MFQ algorithm

As shown in figure …, as process with pid 496 enters the highest queue 16, it executes a quantum of 5 before moving to the lower queue 17, due to the feedback mechanism, to execute for a quantum of 10 before moving to the lowest priority queue 18 as it didn’t finish

its work in the first 2 quantums executed in queues 16,17. After it finishes its quantum in queue 18 without finishing its work, it moves again to the highest priority queue 16 and so on.

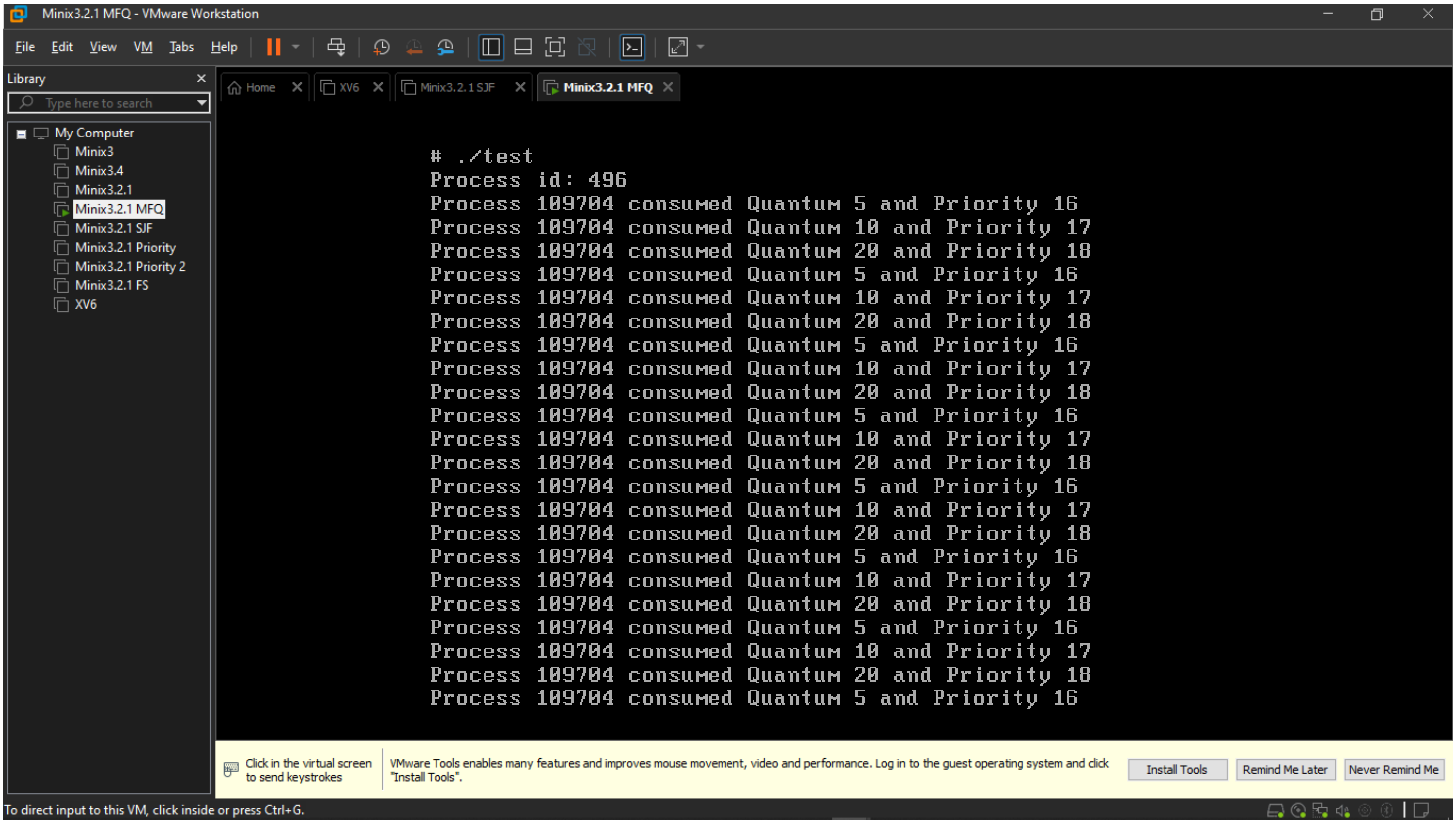


Figure 10: MFQ execution example

2.6. Testing Scheduling Algorithms

A group of random processes are executed to test and compare the different scheduling algorithms we implemented. A testing driver program was made to test and output the waiting time and the turnaround time of each process.

Turnaround time = Time a process took during its lifecycle, from admitted until termination.

Waiting time = Turnaround time – Execution time

2.6.1. Testing RR

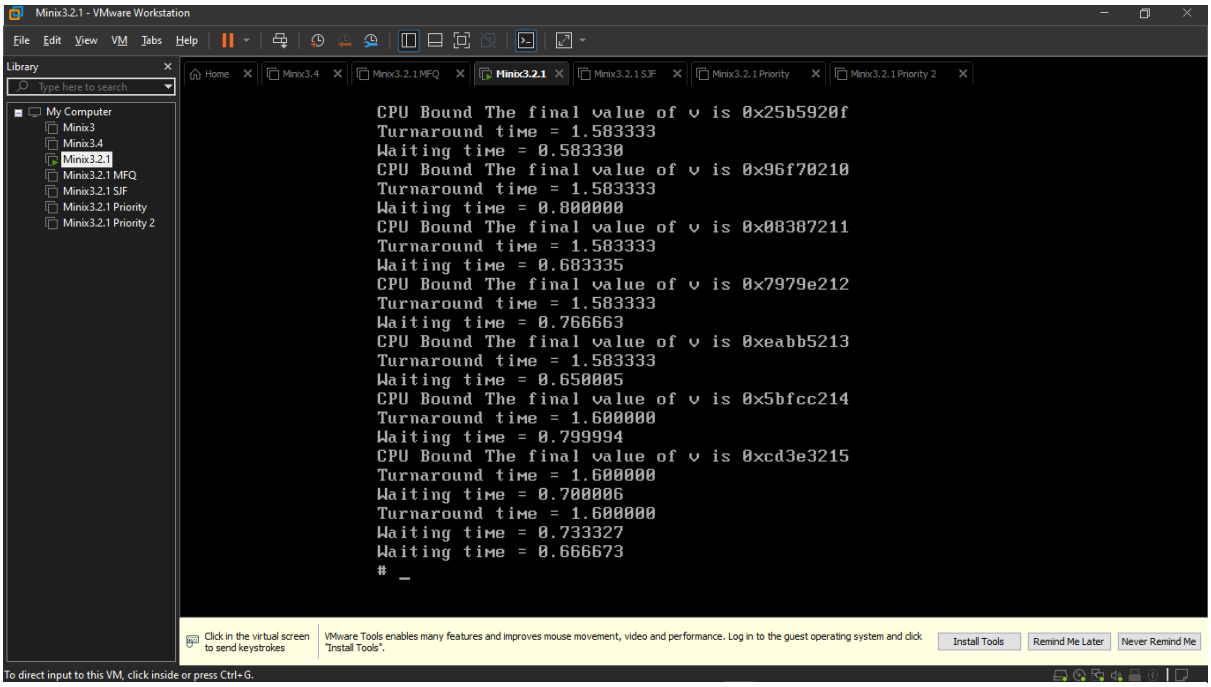


Figure 11: Testing Default RR

2.6.2. Testing SJF

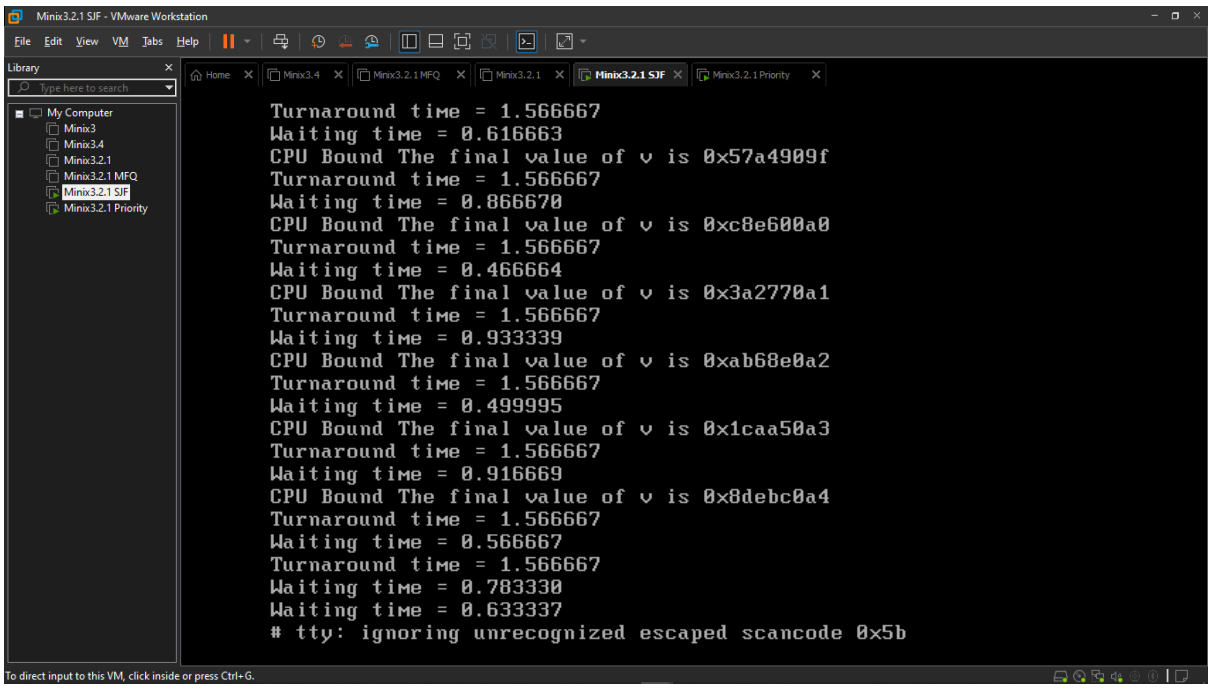


Figure 12: Testing SJF

2.6.3. Testing priority-based queue

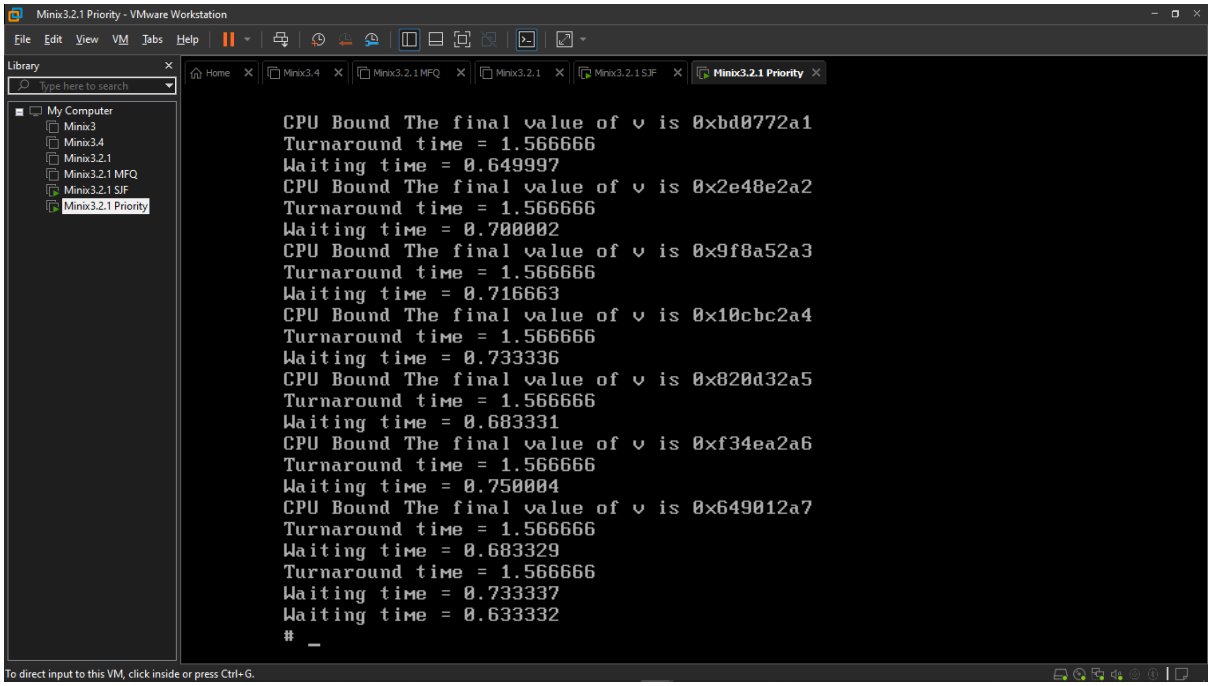


Figure 13: Testing priority based

2.6.4. Testing MFQ

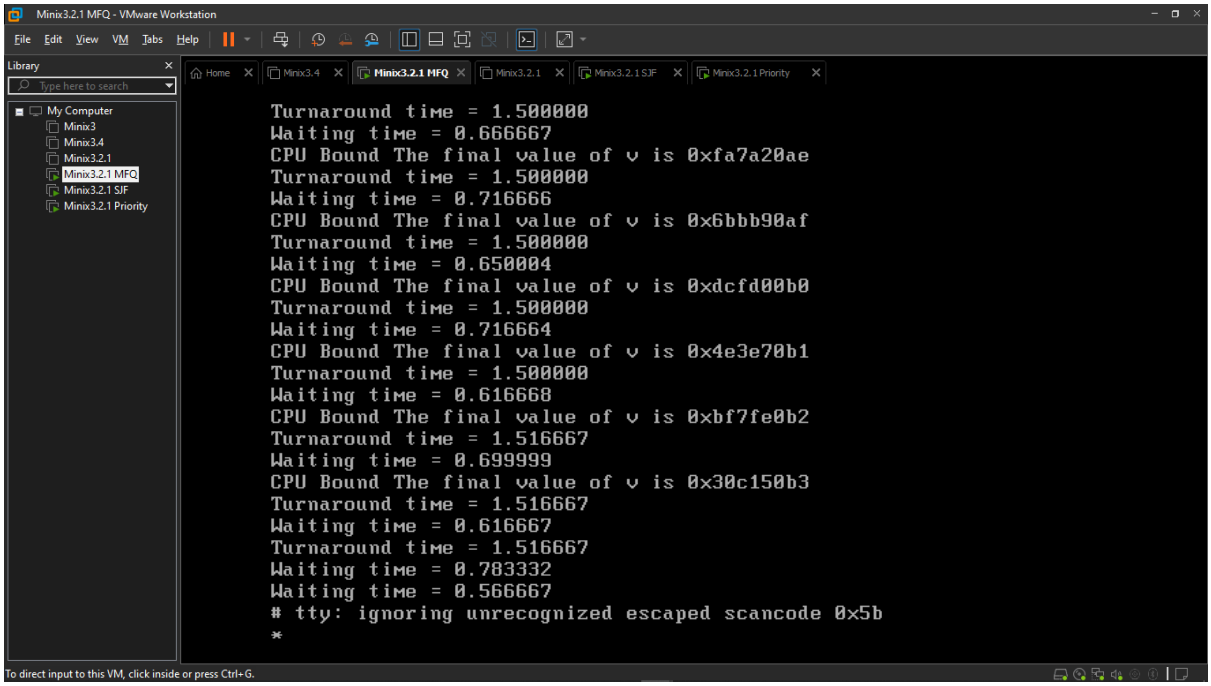


Figure 14: Testing MFQ

2.6.5. Comparing & evaluating testing results:

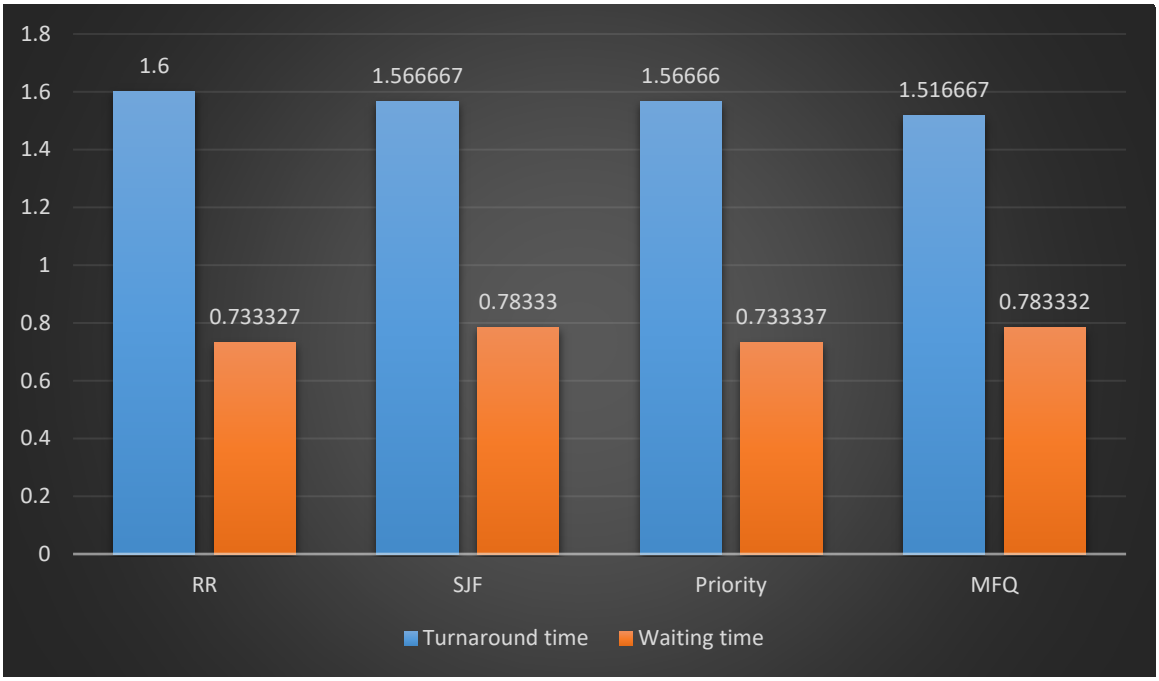
After collecting the average turnaround time and average waiting time of each scheduling algorithm, this chart is concluded. The differences of turnaround time and waiting time between the algorithms could seem to be small but it is significant because these records are measured in seconds and the clock cycle is measured with nanoseconds, thus there are enormous number of clock cycles difference between these algorithms.

Comparing the turnaround and waiting time of the studied algorithms, we found:

1- The turnaround time of RR is greater than that of SJF, while the waiting time of the RR is smaller than that of SJF. The reason is that in the SJF algorithm some processes could wait a long time before execution, but the average turnaround time of all processes would be small.

2- The turnaround time of SJF is approximately the same as that of priority, which would happen at certain cases, such as: longer processes have lower priority. While the waiting time of the SJF is greater than that of priority, the reason is that in the SJF some processes could wait a long time before execution than the processes wait in priority.

3- The turnaround time of priority is greater than that of MFQ, which is obvious from the behavior of these algorithms, as the process in the MFQ enters 3 queues while it enters only 1 queue in priority. On the other hand, the waiting time of the priority is smaller than that of MFQ, the reason is that in the MFQ some processes could wait a long time not assigned to a queue going from a queue to another, but this doesn't happen in priority.



25. MEMORY MANAGEMENT

In this requirement, we worked on a different operating system which is xv6. Xv6 is a small OS developed by MIT mainly for educational purposes. It is a re-implementation of Unix Version 6 (v6), so it follows the structure and style of v6, but is implemented in ANSI C for an x86-based multiprocessor. In xv6, we modified the hirarchical paging scheme to be implemented according to some parameters: page size, number of levels, address format, ... etc.) which are all user-defined via a configuration file. Additionally, FIFO and LRU page replacement algorithms were implemented.

3.1. Memory management in xv6

The memory management process in xv6 is achieved using paging technique. This is due to the wide range of advantages of paging over other memory management schemes such as Contiguous Memory Allocation and Segmentation. For instance, it avoids the external fragmentation, common in contiguous memory allocation schemes, and hence there is no need to for the time-consuming compaction process. However, internal fragmentation is still found but it is considered impactless. Paging also helps to protect the memories of different processes.

Paging is implemented by breaking the memory into fixed-sized blocks called pages (usually with sizes ranging from 0.5 KB to 16 MB). In addition, the logical memory of each executing process is also divided into logical pages of the same size. Each process's logical page is mapped to a free physical page in the main memory in which it is stored. Therefore, the physical address space of a process is totally different from the logical address space and can be non-contiguous, and hence allowing for a greater logical address space that can usually reach up to 2^{32} Bytes (in x86 systems).

For each process, there is a page table stored in the physical memory that contains multiple entries. Each entry maps a logical page of the process to the physical page in which it is stored. And using this page table, each logical address for that process is mapped to the proper physical address in the main memory.

Every logical address generated by the CPU is divided into two parts: a page number (p), and a page offset (d) (*Figure 15: Logical address division*). For a system with a logical address space of 2^m Bytes and page size of 2^n Bytes, the offset is the rightmost n-bits, while the page number is the leftmost m-n bits.

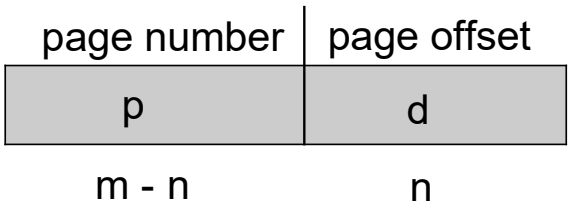


Figure 15: Logical address division

As *Figure 16: Sending addresses to MMU* shows, the page number (p) is used as an index into the page table to access the base address (f) of the physical page that is mapped to that logical address. This base address (f) is combined with the page offset (d) to define the physical memory address that is sent to the memory management unit (MMU).

This paging technique is called one-level paging. And usually with big logical address space (m>32), the number of page table entries (2^{m-n}) becomes too huge, and the physical memory needed to store the entire page table increases considerably.

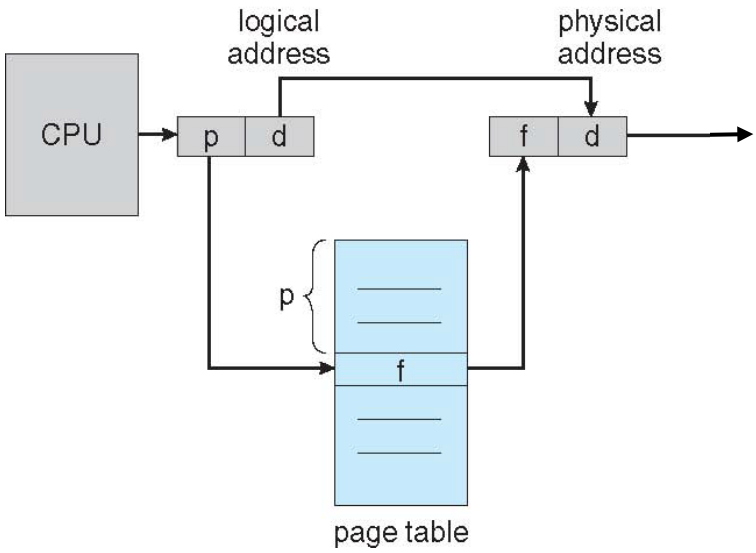


Figure 16: Sending addresses to MMU

To overcome this space overhead, two-level (or multi-level) paging is applied, where there is an outer page table that maps to more than one inner page tables, as *Figure 17: Multi-level paging* shows.

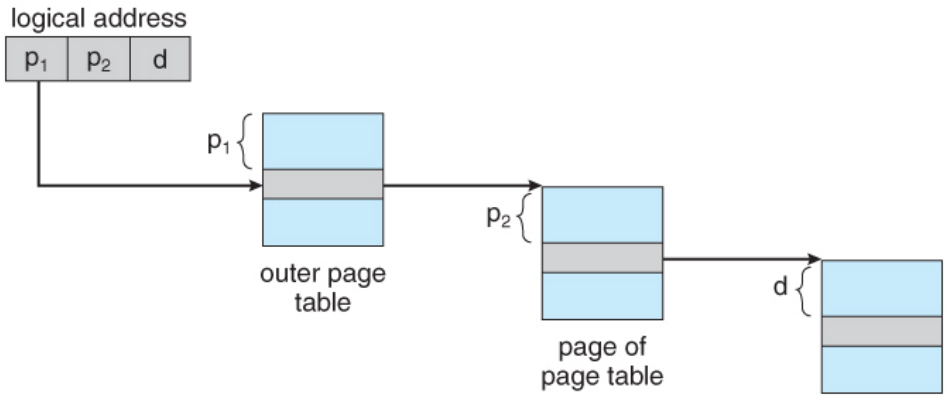


Figure 17: Multi-level paging

In this 2-level paging, the logical address is also divided into the n-bits offset (d) and the m-n bits page number (p). However, the page number is further divided into outer page number (p_1) and inner page number (p_2), where p_1 is an index into the outer page table, and $p_2$is the displacement within the page of the inner page table.

3.2. Hierarchal paging in xv6 (2-level paging)

In xv6, two-level hierarchal paging is implemented, where there is an outer page table that points to a page of inner page tables.

Since that xv6 is built to be used on x86 processors, both logical addresses and physical addresses are of length 32 bits. Each 32-bit logical address is divided into three parts: outer page number, inner page number, and the offset. Since the page size is 4 KB (2^{12} Byte), so the offset length is 12 bits. The remaining 20 bits is divided into 10 bits for both the outer page number p_1, and the inner page number p_2.

So, with this architecture, there is an outer table consisting of 1024 (2^{10}) entry, with each entry pointing to one of the 1024 inner tables. Each inner table also consists of 1024 (2^{10}) entry, with each one pointing to a physical page address in the main memory.

And because each table (either outer or inner) consists of 1024 32-bit entry, the total size of each table will be 4 KB (1024*4 Bytes) same as the page size. So, each table is exactly fitted in a physical page in the main memory.

In xv6, the outer page table is called the ***page directory,*** while the inner ones are simply called ***page tables***.

The whole page table is actually stored in physical memory as a two-level tree. The root of the tree is the 4 KB **page directory** that contains 1024 references to inner page tables. Each inner page table is an array of 1024 32-bit entry.

As shown in *Figure : Hierarchical paging xv6*, the paging hardware uses the top 10 bits (**Dir**) of a virtual address to select an entry in the **page directory**. If the page directory entry is present, the paging hardware uses the next 10 bits of the virtual address (**Table**) to select an entry from the inner **page table** that the page directory entry refers to. If either the page directory entry or the inner table entry is not present, the paging hardware raises a fault.

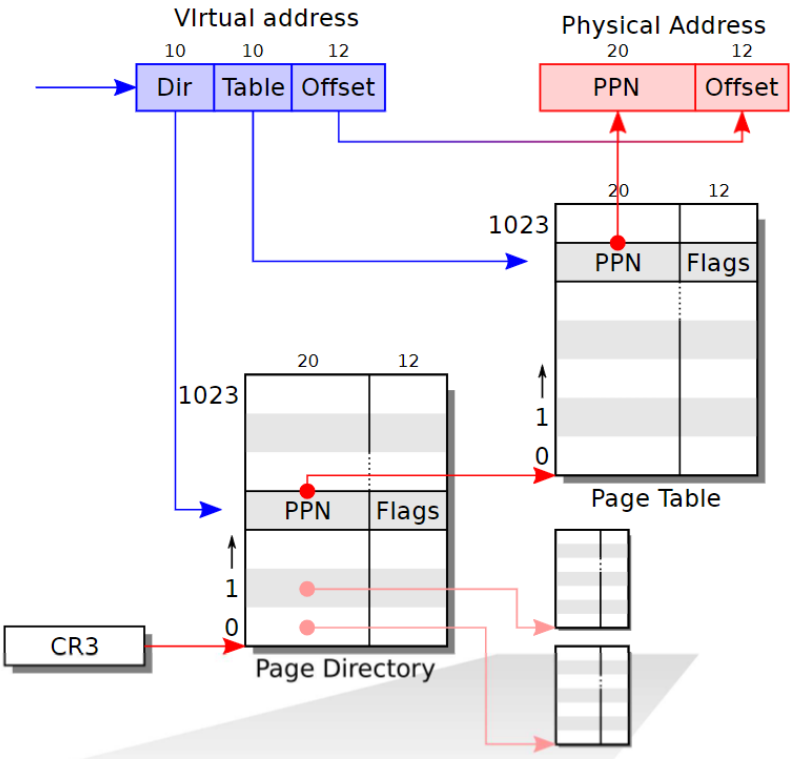


Figure 18: Hierarchical paging xv6

This target 32-bit entry contains a 20-bit physical page number (PPN) and the remaining 12 bits are reserved for some flags. The paging hardware copies this 20-bit **PPN** and combines it with the 12-bit **Offset** in the logical address to finally produce the 32-bit target physical address that will be accessed in the main memory.

3.3. How we modified hierarchal paging (3-level paging)

We modified the hierarchal paging scheme in xv6 to be 3-level hierarchy instead of the already implemented 2-level one. This means that there will be three levels for page tables: level 2 (outermost), level 1, and level 0 (innermost). Each entry in level 2-page table points to one of the level 1 page tables. Similarly, each entry in the level 1-page tables points to one of the level 0 page tables. Finally, each entry in the innermost level 0-page tables contains the base address of the physical page in the main memory.

As shown in *Figure 19: 3-Level Hierarchical paging*, this hierarchy is implemented by dividing the 32-bit logical address into 4 parts. The rightmost 12 bits are reserved for the **Offset,** while the remaining 20 bits represents the page number. These 20 bits are divided into 3 bits as level 2 index **(L2)**, 7 bits as level 1 index **(L1)**, and 10 bits for level 0 index **(L0)**.

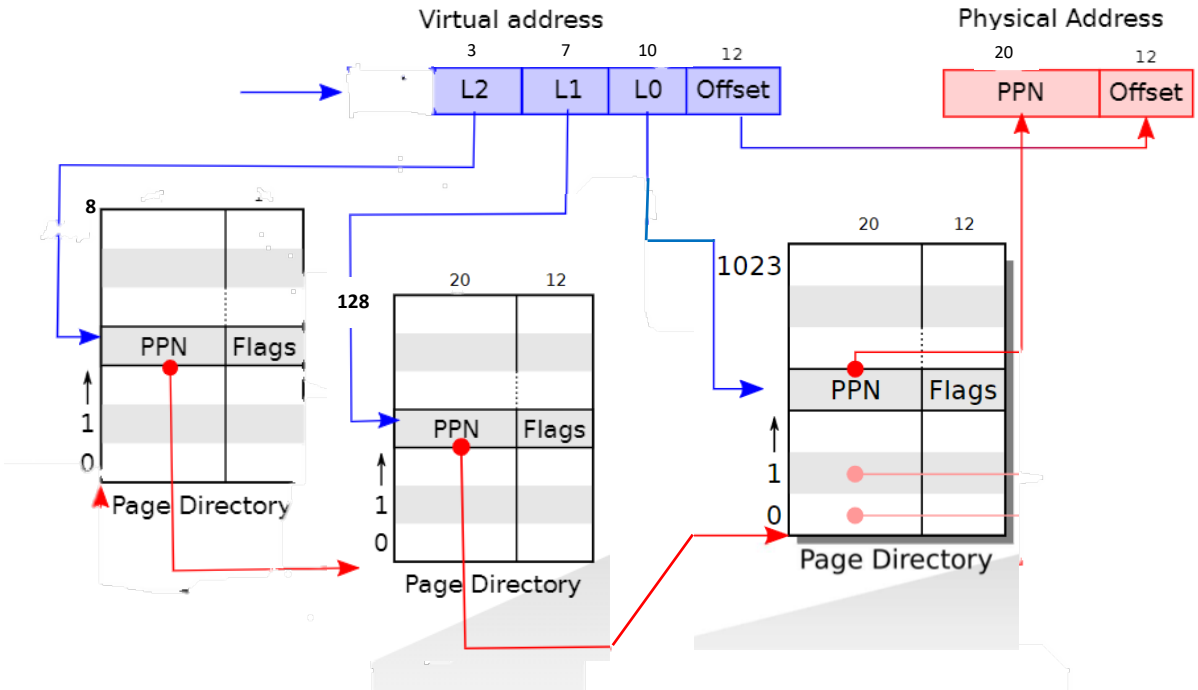


Figure 19: 3-Level Hierarchical paging

- **Code Edits:**

In the header file ***"mmu.h",*** all the parameters and functions related to the new 3-level hierarchy are modified. These parameters define the no. of bits allocated to each level and deal with virtual addresses in order to return the required new indexes for each level access. *Figure 20* shows all these modifications in "***mmu.h***" header file.

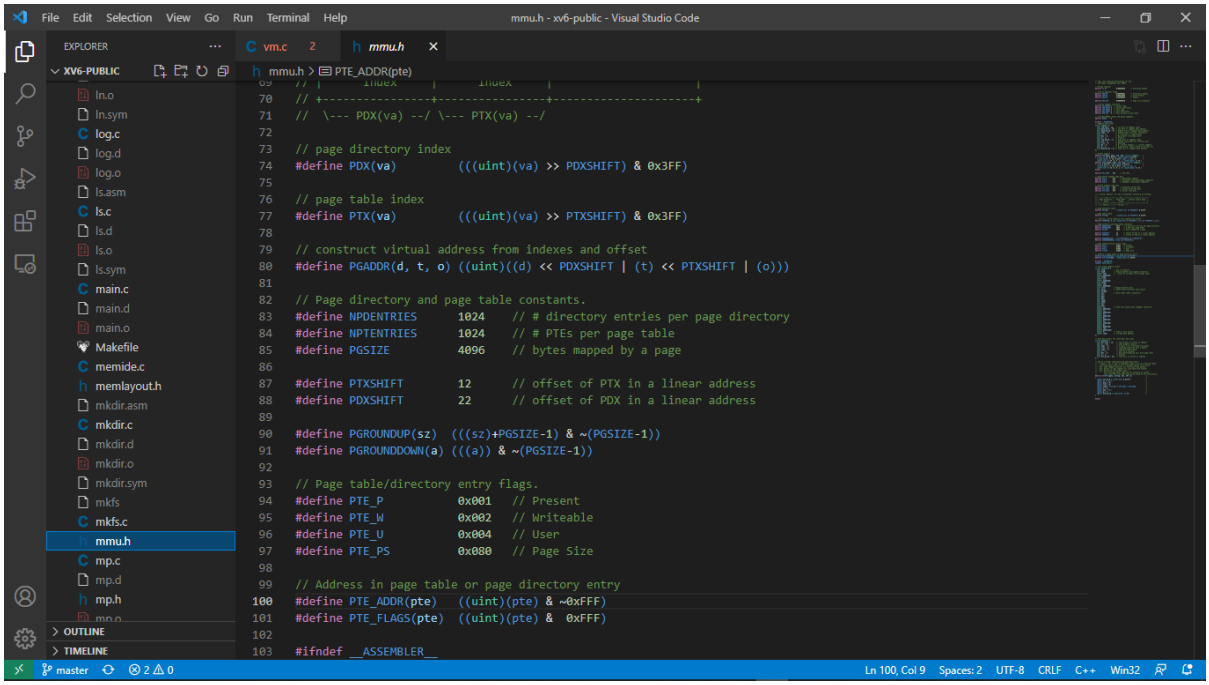


Figure 20: 3-Level Paging Code 1

In ***"vm.c",*** the main function responsible for walking in the page directory to map each virtual address to the physical page address in memory is ***walkpgdir().*** This function takes in a virtual address used by a process (***va***), the page-table of that process (***pgdir***) and returns the page table entry that points to the target physical address.

There is also a parameter called (**alloc**) which is used to specify whether the process can write to the target physical page or not.

Using this format, the ***walkpgdir()*** function is able to traverse through all the pages and return the final page-table entry. The returned page-table entry is essentially an unsigned 32-bit integer, which acts as a pointer to the physical address.

Our modification in the code allows the user to choose between the already implanted 2-level hierarchy or our new 3-level one. This can be achieved by modifying the **THREE_LEVEL_PAGING** macro found in **"param.h"** file. Figure 21 and Figure 22 in the next page show the modified ***walkpgdir()*** function***.***

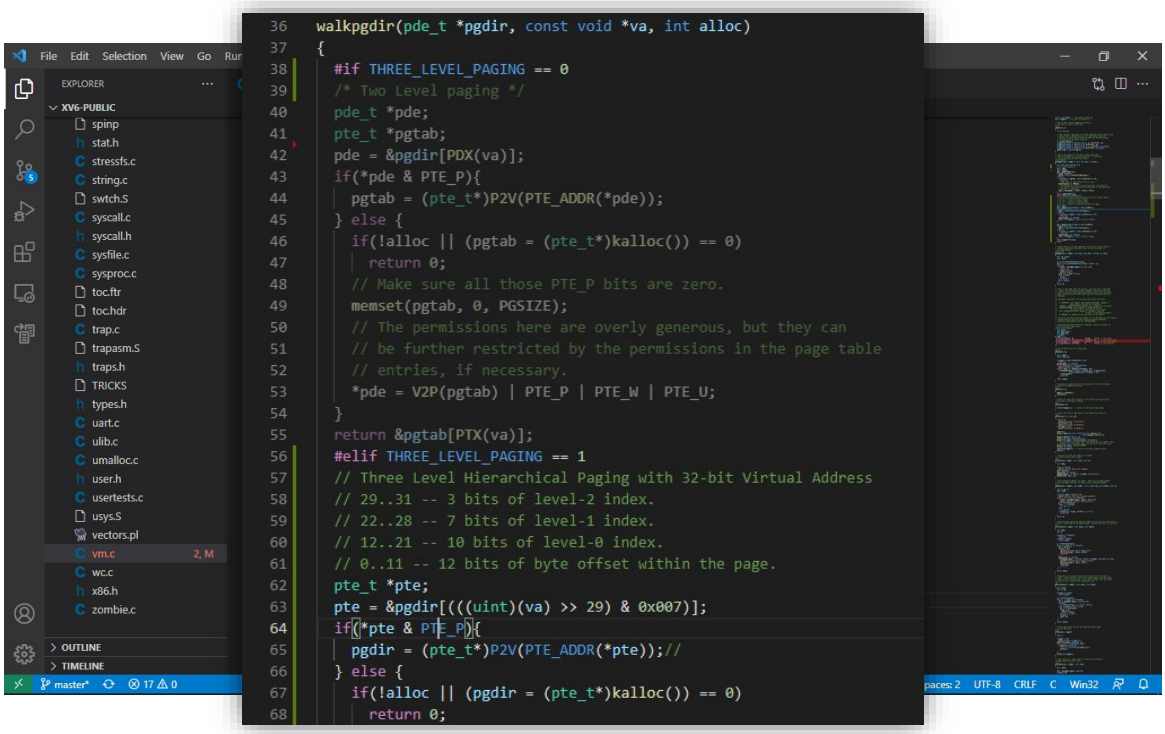


Figure 21: 3-Level Paging Code 2

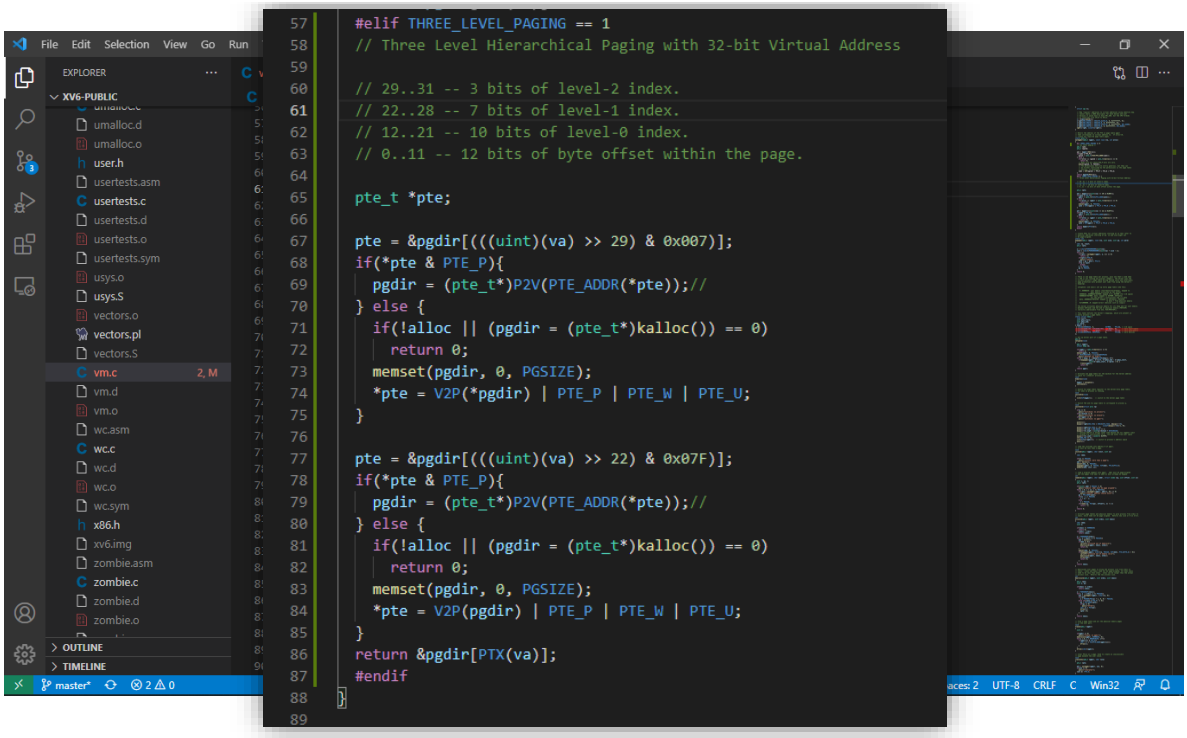


Figure 22: 3-Level Paging Code 3

3.4. Implementing swapping & page replacement algorithms in xv6

In any operating system that uses paging, a page replacement algorithm is needed to decide which page will be replaced and swapped out to the backing store when a new page comes in while the main memory is full.

Whenever a new page is referenced and it is not loaded in the memory, page fault occurs, and the OS replaces one of the existing pages (victim page) with this referenced page. So, a replacement algorithm must be utilized to determine which page in the memory will be replaced and swapped out. The target for all algorithms is to reduce number of page faults.

Our implementation of memory swapping can be turned on/off by changing the parameter ***ENABLE_SWAPPING*** in the ***"param.h"*** file, as shown in Figure 23. The maximum number of pages per process is set using ***MAXPAGES*** parameter, and the type of swapping (LRU or FIFO) is changed using the parameter ***LRU_SWAPPING***.

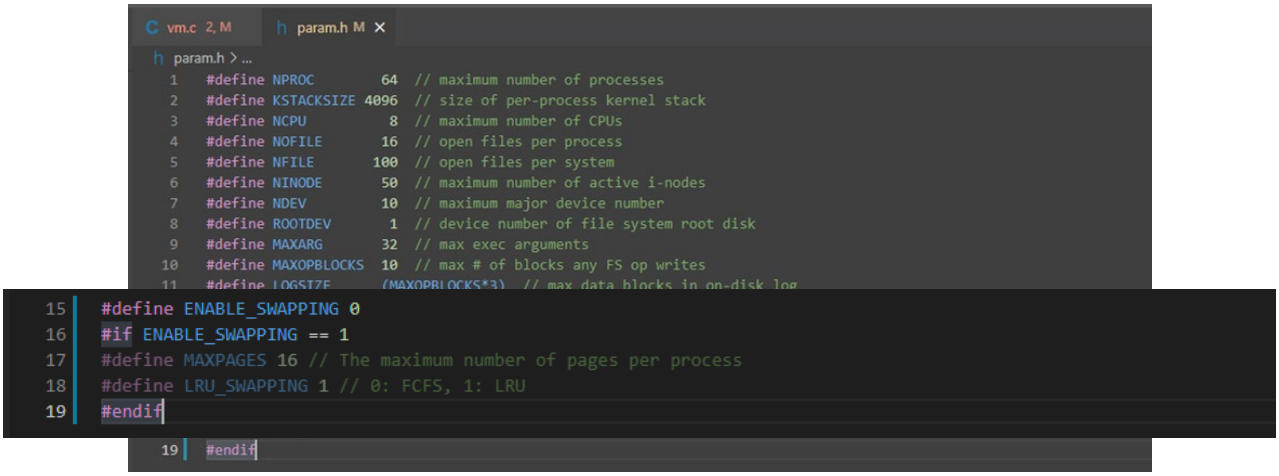


Figure 23: Page Swapping Code 1

In order to swap memory pages to the backing store, we implemented a function called ***swapToMem()*** in ***"proc.h"*** file, as shown in Figure 24. It takes the virtual address that the process is trying to access along with the process's page-table. The virtual address is stripped of any flags before being passed to the function to ensure it is consistent between swaps.

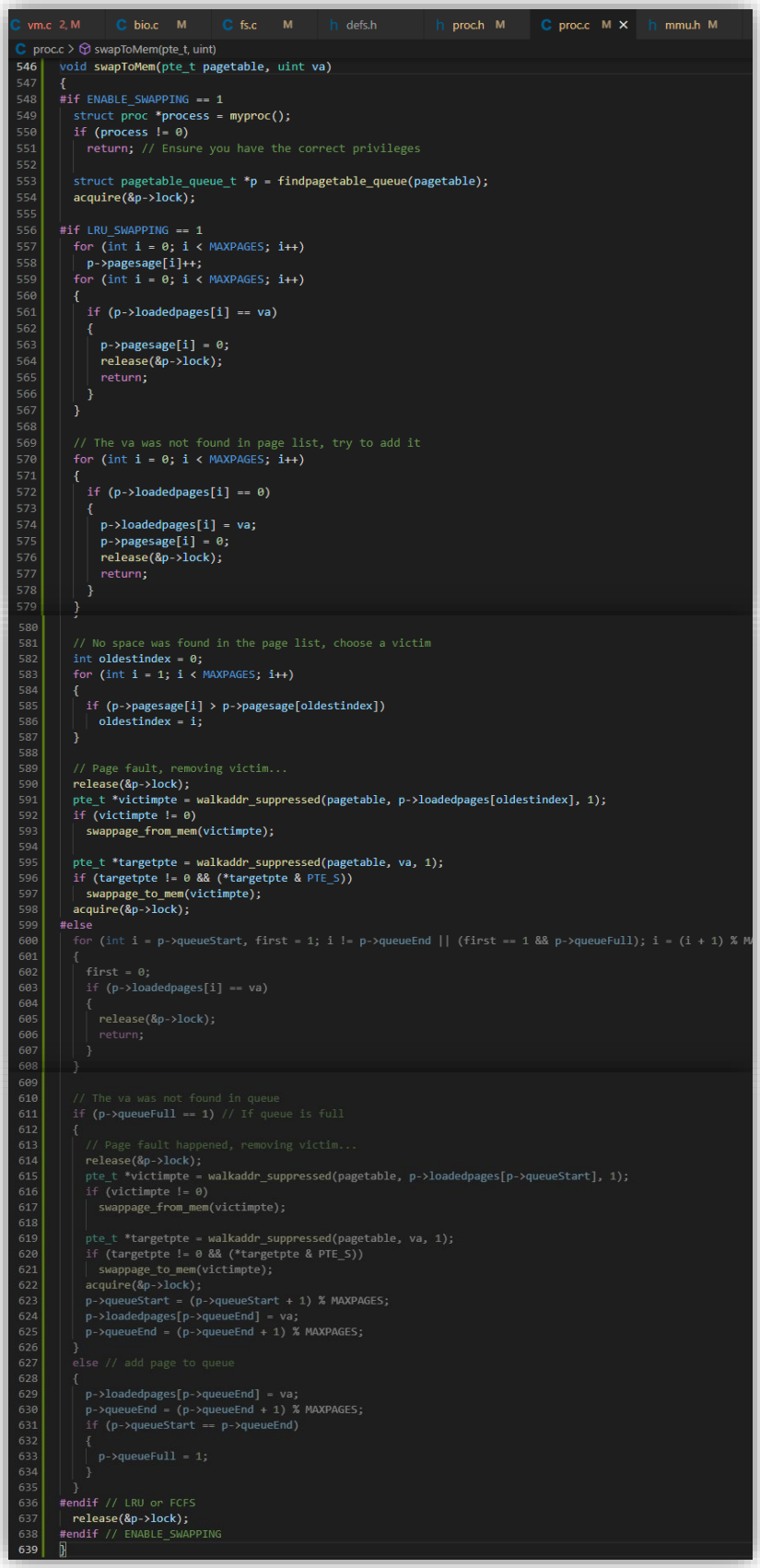


Figure 24: Page Swapping Code 2

The **walk()** function was also changed to call another modified function called ***walk_suppressed()***, shown in Figure 25 . It is a modified version of the ***walk()*** function that performs a call to the ***swapToMem()*** if the parameter ***supressSwapping*** is set to zero.

In order for the kernel to know the order of access of the memory pages of each process, a new data structure called ***pagetable_queue_t*** was made in the ***"proc.h"*** file.

Figure 25: Page Swapping Code 3

In our implementation, page swapping was implemented by calling the two functions shown in figure Figure 26: **swappage_from_mem()** and **swappage_to_mem()**, both located in **"vm.c"** file. Those two functions perform all the actions needed to swap pages into and out of ram.

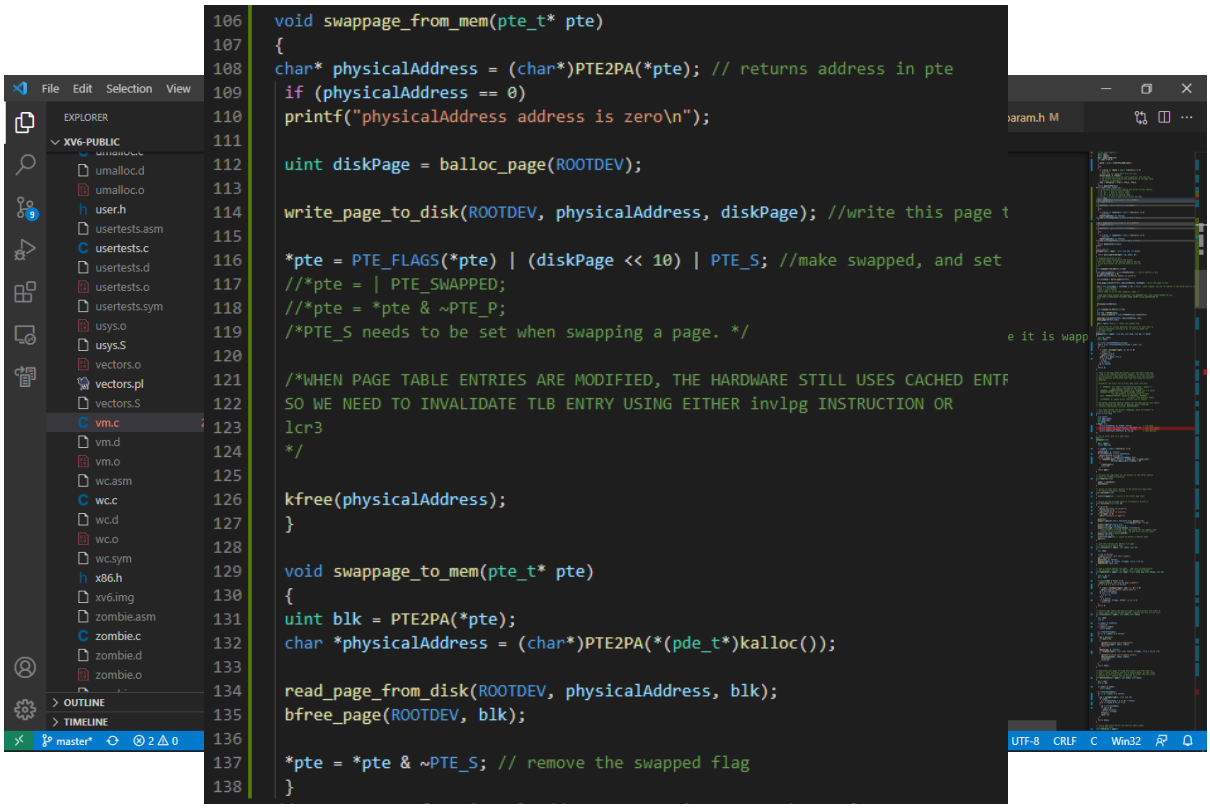
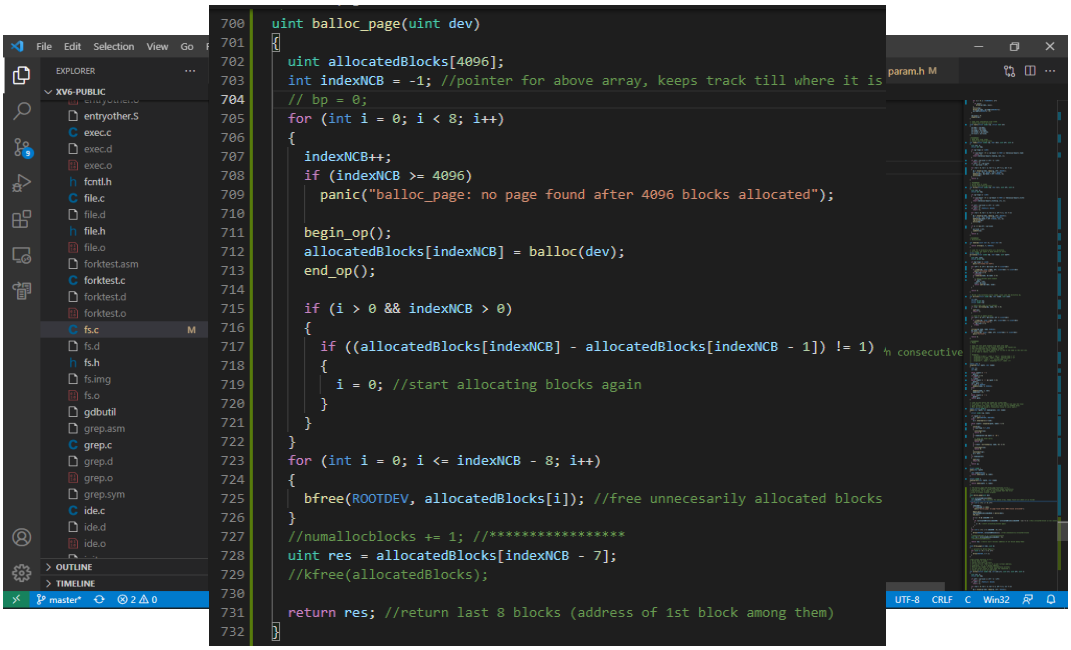


Figure 26: Page Swapping Code 4

The function **swappage_from_mem()** is responsible for swapping the victim page to the backing store on the disk by doing the following:

1- Maps the victim page to its physical address on the memory by calling ***PTE2PA***

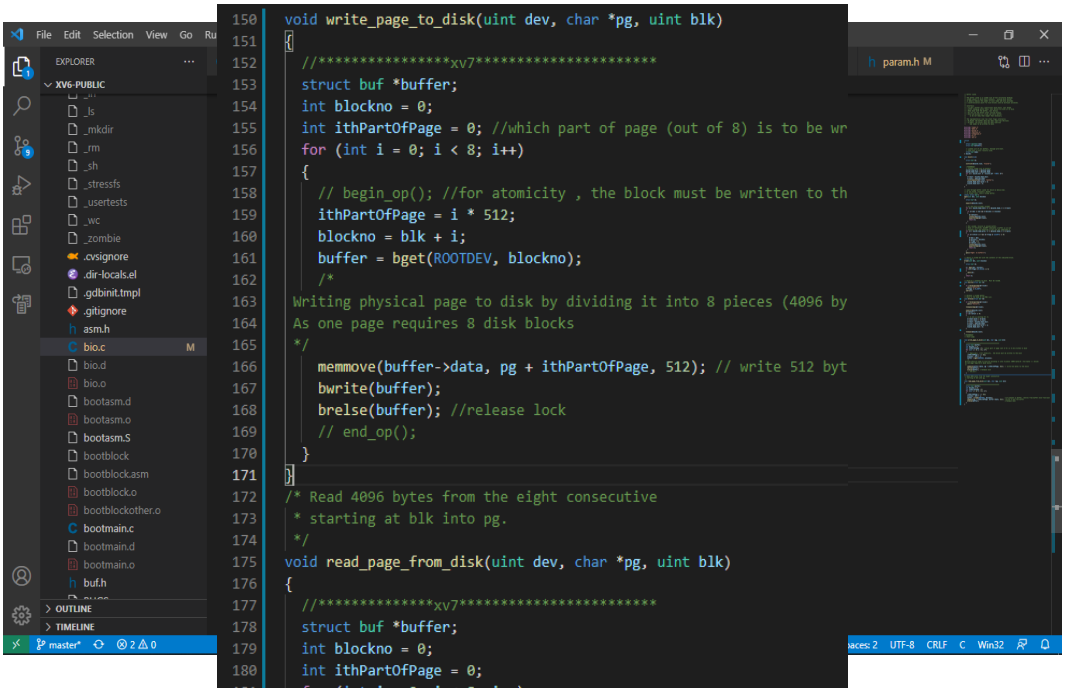
2- Allocates eight consecutive blocks on the disk (at which the victim page will be stored) by calling ***balloc_page()*** located in ***"fs.c"***



```
uint balloc_page(uint dev)
{
  uint allocatedBlocks[4096];
  int indexNCB = -1; //pointer for above array, keeps track till where it is
  // bp = 0;
  for (int i = 0; i < 8; i++)
  {
    indexNCB++;
    if (indexNCB >= 4096)
      panic("balloc_page: no page found after 4096 blocks allocated");

    begin_op();
    allocatedBlocks[indexNCB] = balloc(dev);
    end_op();

    if (i > 0 && indexNCB > 0)
    {
      if ((allocatedBlocks[indexNCB] - allocatedBlocks[indexNCB - 1]) != 1)
      {
        i = 0; //start allocating blocks again
      }
    }
  }
  for (int i = 0; i <= indexNCB - 8; i++)
  {
    bfree(ROOTDEV, allocatedBlocks[i]); //free unnecesarily allocated blocks
  }
  //numallocblocks += 1; //*****************
  uint res = allocatedBlocks[indexNCB - 7];
  //kfree(allocatedBlocks);

  return res; //return last 8 blocks (address of 1st block among them)
}
```

Figure 27: Page Swapping Code 5

3- Calls **write_page_to_disk()**, located in ***"bio.c",*** to write the victim page to newly allocated blocks



```
void write_page_to_disk(uint dev, char *pg, uint blk)
{
  //***************xv7*********************
  struct buf *buffer;
  int blockno = 0;
  int ithPartOfPage = 0; //which part of page (out of 8) is to be wr
  for (int i = 0; i < 8; i++)
  {
    // begin_op(); //for atomicity , the block must be written to th
    ithPartOfPage = i * 512;
    blockno = blk + i;
    buffer = bget(ROOTDEV, blockno);
    /*
  Writing physical page to disk by dividing it into 8 pieces (4096 by
  As one page requires 8 disk blocks
  */
    memmove(buffer->data, pg + ithPartOfPage, 512); // write 512 byt
    bwrite(buffer);
    brelse(buffer); //release lock
    // end_op();
  }
}
/* Read 4096 bytes from the eight consecutive
 * starting at blk into pg.
 */
void read_page_from_disk(uint dev, char *pg, uint blk)
{
  //**************xv7************************
  struct buf *buffer;
  int blockno = 0;
  int ithPartOfPage = 0;
```

Figure 28: Page Swapping Code 6

4- Changes the PTE (page-table entry) to point to the newly created block.

5- Flags this PTE as swapped out of ram

After swapping out the victim page, **swappage_to_mem()** moves the target page to the freed frame in the memory by doing the following:

1- Extracts the page number from the PTE by calling **PTE2PA**

2- Allocates a new page in memory

3- Calls **read_page_from_disk()**, located in **"bio.c"** to copy the page data from disk to the newly allocated page in memory

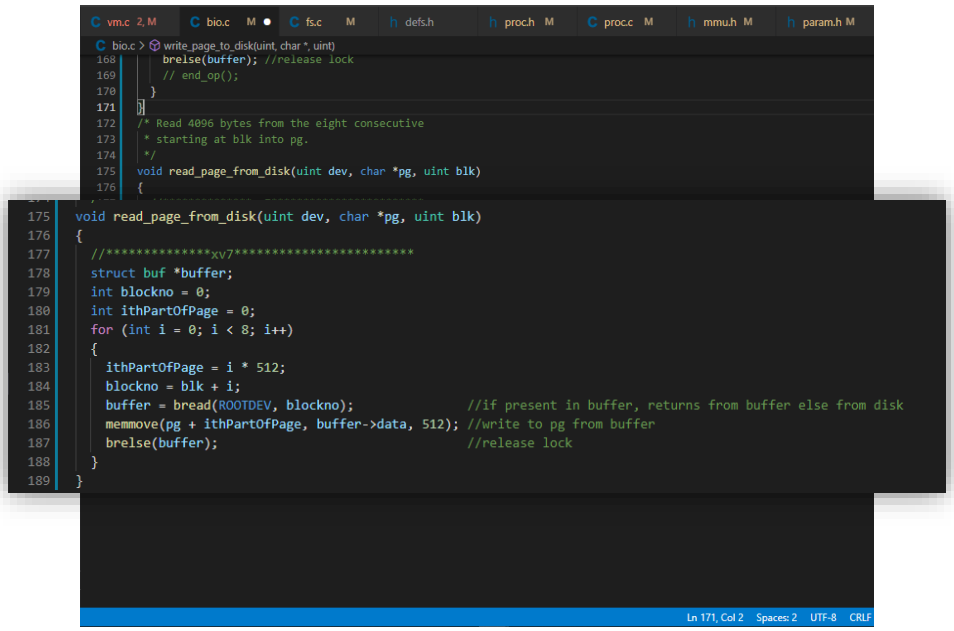


Figure 29: Page Swapping Code 7

4- Frees the previously allocated disk space by calling **bfree_page()** located in **"fs.c"**

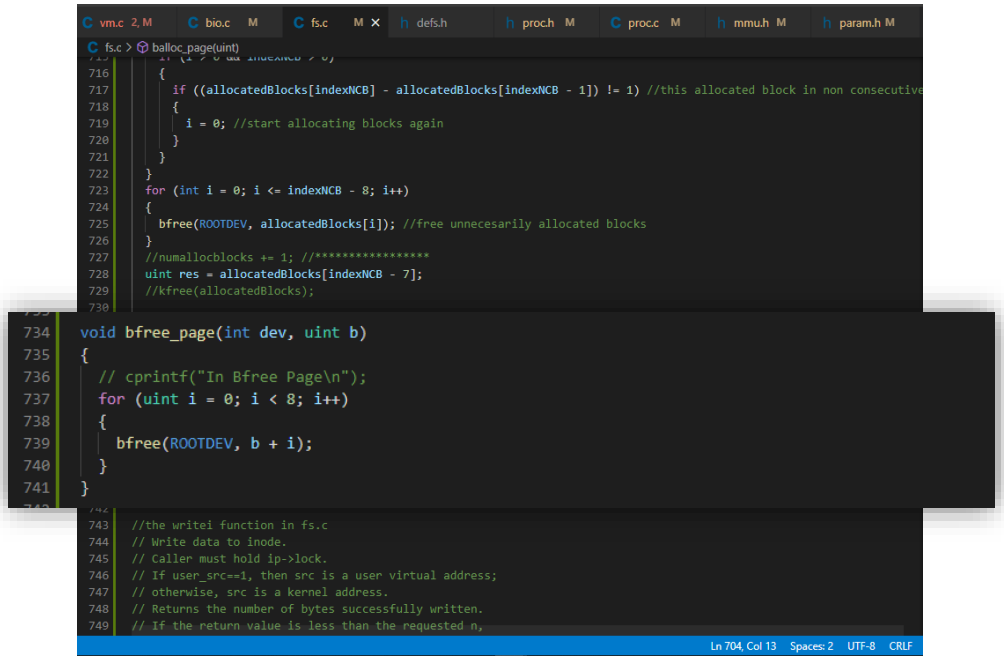


Figure 30: Page Swapping Code 8

5- Flags the PTE as not swapped out of ram

3.4.1. First In First Out (FIFO) algorithm

The most basic page-replacement algorithm is a first-in, first-out (FIFO) algorithm. A FIFO replacement algorithm associates with each page the time when that page was taken into the memory. When a page must be replaced, the oldest page is chosen. It is not exactly mandatory to record the time when a page is brought in. We can create a FIFO queue to hold all pages in memory. We replace the page at the front of the queue When a page is brought into memory, we enqueue it at the tail of the queue.

If the ***LRU_SWAPPING*** parameter in ***"param.h"*** is set to zero, the ***loadedpages*** array in the ***pagetable_queue_t*** , found in ***"proc.h"*** as shown in Figure 31, is treated as a queue. The

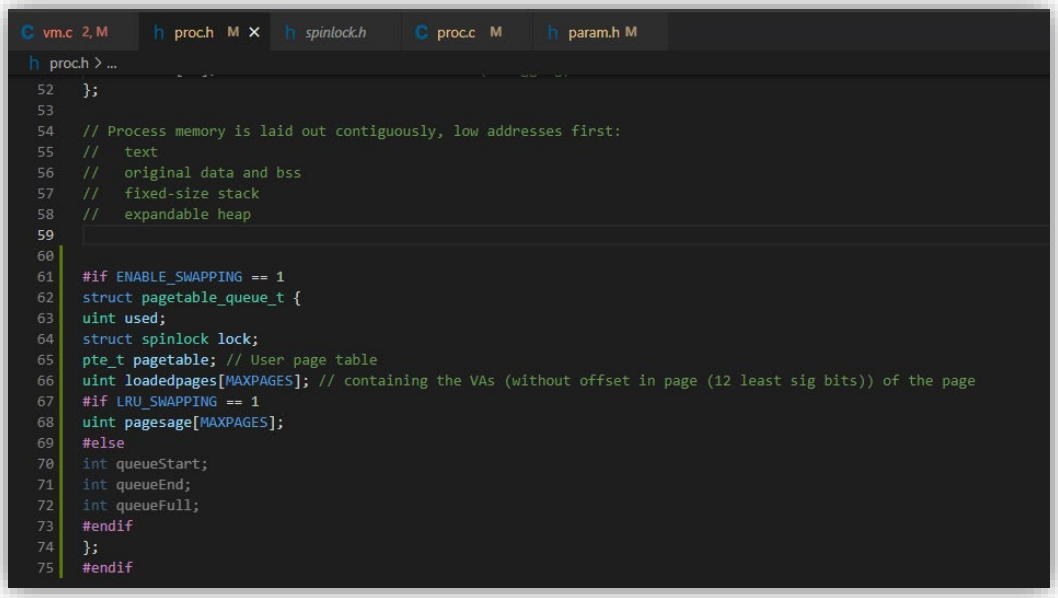


Figure 31: FIFO Replacement Algorithm Code

variables *queueStart*, *queueEnd* and *queueFull* act as metadata for the queue. Once the ***pagetable_queue_t*** for the corresponding ***pagetable*** is found, the function checks if the queue of the given ***pagetable*** is full. If it is not full, the accessed page is enqueued to the queue and no page fault occurs. If the queue is full, a victim page must be selected to be swapped out to the backing store on disk and replaced with the new loaded page. In the case of the FIFO algorithm, the first page in the queue (the oldest enqueued one) is selected as the victim. The victim page in then swapped to the disk and replaced with the new loaded page using **swappage_from_mem()** and **swappage_to_mem()** as mentioned before.

3.4.2. Last Recently Used (LRU) algorithm

The last recently used (LRU) algorithm associates with each page the last time at which the page is used. When a new page comes in, LRU chooses the last used page residing in the main memory. (i.e., the one which has the oldest usage time).

In our implementation, LRU algorithm is applied when the ***LRU_SWAPPING*** parameter is set to 1. In this case, the ***loadedpages*** array in the ***pagetable_queue_t*** is treated as a list of accessed pages (in no particular order). However, the age of each page in ***loadedpages*** array is stored as an arbitrary integer in a new array called ***pagesage,*** as shown in Figure 32***.*** This age integer is incremented whenever any page in a ***pagetable*** is accessed, and the age of the process that was accessed is set to zero. This way, the last recently used page is easily obtained by finding the page with the highest age.

Figure 32: LRU Algorithm Code

When using LRU algorithm, the ***swapToMem()*** function first tries to find any free location in the ***loadedpages*** array. If found a free one, the new page is added to that empty location. However, if no free location is found, the function selects a victim by finding the page with the highest age. The victim is swapped to disk, and the new page is loaded into memory as explained before using **swappage_from_mem()** and **swappage_to_mem()**.

3.5. Performance results for hierarchal paging and replacement algorithms:

Firstly, we build the system image by using WSL (windows subsystem Linux), which allows us to use Linux terminal on our operating system to use its commands to build XV6.
Before we run the system to examine the FIFO and LRU algorithms we predict that the LRU must have higher hit ratio than FIFO.

Secondly, we run the system to examine its performance and calculate number of page faults and hits for every algorithm. Thus, we activate the FIFO algorithm and make this input and record the output as shown below. Then, we activate the LRU algorithm and make the same input, to compare between the output of the 2 algorithms, and record the output as shown below.

For the same input, FIFO get 46.67% hit ratio, while LRU get 50% hit ratio, which was predicted before.

```
Enter number of page requests: 30
Enter number of pages: 5
Enter the page requests separated by space:
1 1 4 4 5 9 5 7 6 5 1 1 9 3 0 9 7 2 5 9 5 8 3 2 8 3 5 4 8 3
LRU:
        Page faults : 15
               Hits : 15
          Hit Ratio : 50.00%
```

```
Enter number of page requests: 30
Enter number of pages: 5
Enter the page requests separated by space:
1 1 4 4 5 9 5 7 6 5 1 1 9 3 0 9 7 2 5 9 5 8 3 2 8 3 5 4 8 3
FIFO:
        Page faults : 16
               Hits : 14
          Hit Ratio : 46.67%
```

26. FILE SYSTEM MANAGEMENT

4.1. Requirement Statement

In this requirement, you need to find how MINIX 3 manages empty space, then you need to modify disk-space management code in MINIX 3 to use user-defined extents. Also, the disk allocation should be modified to provide the allocation in terms of the user-defined extents. The extent itself consists of a set of disk blocks where they are handled as a single unit. User should provide all needed configuration in a configuration file that is going to be read by MINIX 3 and adopted accordingly. The performance results of this method with respect to the number of blocks in the extent should be collected, explained, and presented in the report.

4.2. File System in MINIX 3

In MINIX 3, File system management is a system C program in layer 3 of the operating system internal structure, as shown in (*Figure 1: MINIX 3 Structure*), running in user space. It is concerned with space allocation and deallocation for all files, in addition to keeping track of free space and disk blocks. Initially, we must know that any disk is divided into blocks and zones where in MINIX, each block is 1 KB of storage, and what follow are the different parts of the file system.

1- **Boot Block:**

File system starts with a boot block of size 1 KB (regardless of the OS block size) including an executable code which runs on machine start to only boot the operating system; therefore, once the operating system is booted, it is not accessed anymore.
Moreover, all disks managed by the file system has a boot block but the hardware only attempts booting from the disk with a “magic number” placed at in the boot block.

2- **Superblock:**

With a size 1 KB, a superblock comes after the boot block where it holds information regarding the file system layout as shown in *(Figure 19: MINIX Superblock)* and is loaded into a memory table after the OS is booted. Its function is to define the size of the file system parts. For instance, storing the block size and the number of i-nodes, the size of the i-node bitmap and the number of blocks of i-nodes are calculated.

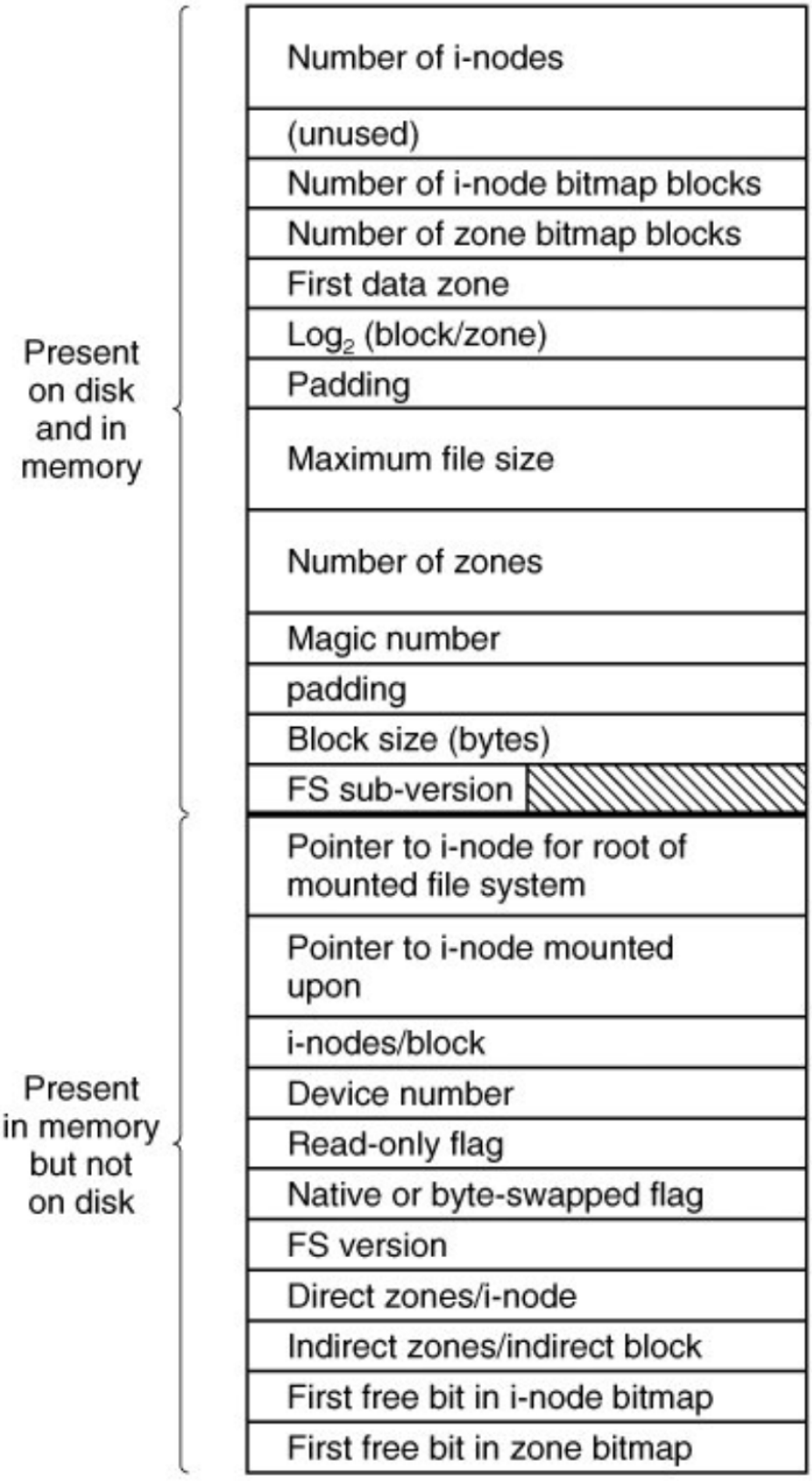


Figure 33: MINIX Superblock

3- Bitmaps:

Two bitmaps are located just after the superblock, the i-nodes bitmap and the zone bitmap, where the address to the first free bit of both bitmaps is stored to the superblock loaded in memory. The free i-nodes or zones are marked with 0 in their corresponding bitmap, while the occupied ones are denoted 1 in the bitmap.

In addition, the first bit in any bitmap is initially set to 1 to remain unused and the system never attempt to allocate It. Therefore, a 1 KB bitmap with 8192 bits can only map 8191 i-node or zone.

4- Zones vs Blocks:

Disk blocks are the smallest unit on a disk and is defined by the operating system, where in MINIX 3 the block size is a constant 1 Kilobytes of storage. Zones allocate one or more blocks at a time where a zone can be 1 ,2, 4, 8...2^n blocks. For instance, a system with 40 MB disk and 4 KB zones allocating 4 blocks at a time will have a 10K bits zone bitmap.

Zones ensure blocks of the same file are on the same physical cylinder in order to boost reading performance when sequentially reading a file on the disk.

5- I-nodes:

Each file has its own index block (i-node), which is a 64 bytes structure holding all information regarding the file as in (Figure 15: MINIX i-node) including the owner's id, file type, mode, access, modification, and status change timings, and most importantly zone numbers which tell where the file data blocks are, directly or indirectly.

The first 7 zone numbers directly locates 7 data blocks, each of size 1 KB; therefore, a file of size greater than 7 KB needs single, double or even triple levels indirect zones. For example, in a system with 32-bit zone number and 1 KB blocks:

- a single indirect block pointing to 256 entries has ->
  1 KB * 256 = 256 KB of storage
- a double indirect block pointing to 256 single indirect blocks ->
  256 * 256 KB = 64 MB of storage

If a system has a 4KB block size:

- a double indirect block pointing to 1024 single indirect blocks ->
  1024 * 1024 * 4 KB = 4 GB of storage

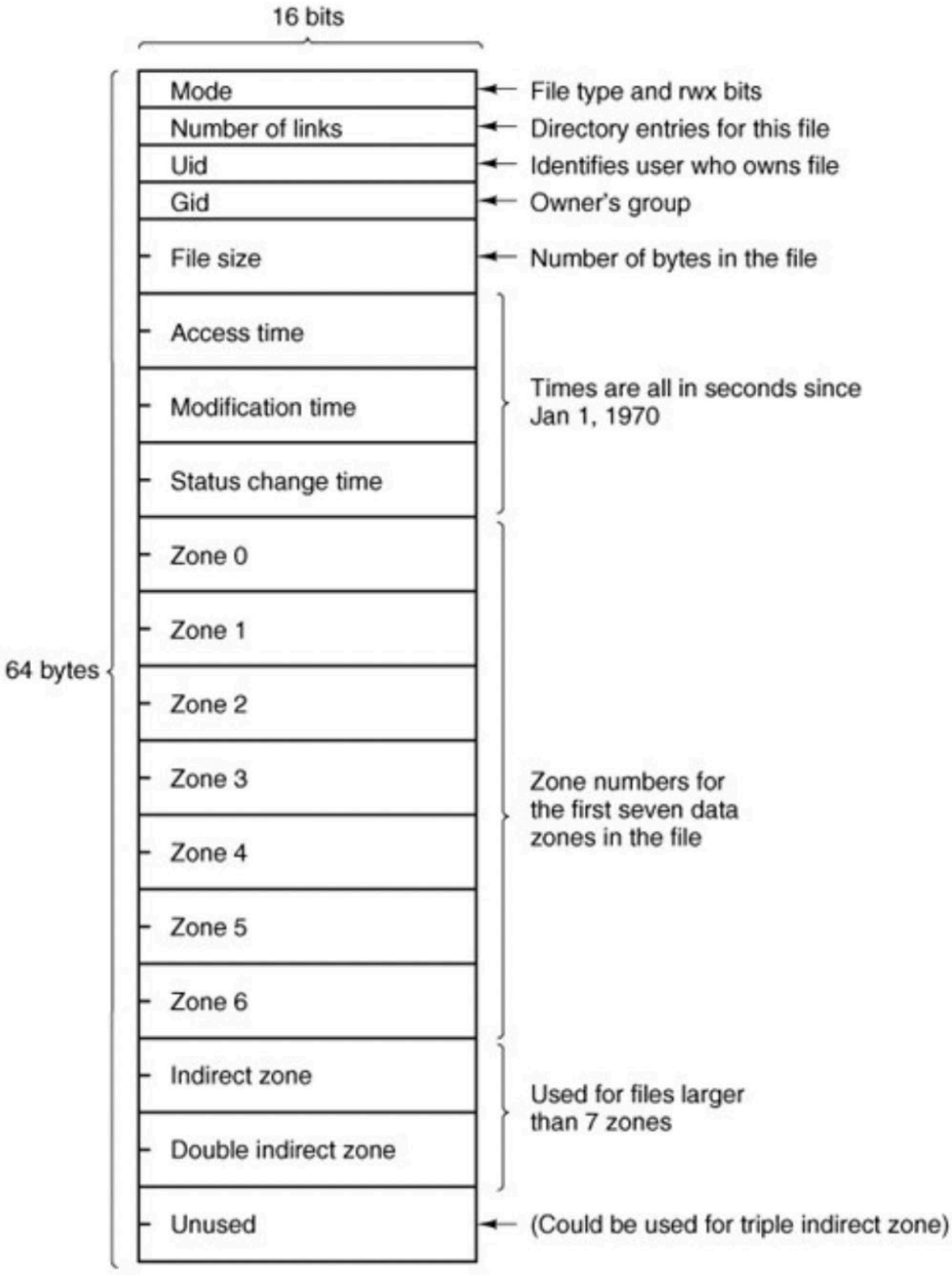


Figure 34: MINIX i-node

4.3. How MINIX 3 manages empty space

MINIX 3 manages free space by tracking i-nodes and zones in the two previously mentioned bitmaps, the i-nodes bitmap and the zone bitmap, where:

0 -> free i-node / zone

1 -> occupied i-node / zone

To remove a file, the mapped bit to the freed i-node (and zone) is set to 0. Afterwards, the "first free bit in i-node bitmap" pointer, already stored in the super block initially loaded in the memory, is updated to the new first free bit only if the freed bit is prior to the already stored bit in the bitmap i-node number.

To create / add a file, the first free bit in i-node bitmap is already stored in the super block loaded in the memory as shown in (*Figure 19: MINIX Superblock*); therefore, no need to search for a free bit in the bitmap and this first free bit is allocated the new file. Afterwards, the first free bit pointer is updated to the next free bit in the bitmap, which is usually the following bit in the bitmap.

All these steps about i-nodes and i-nodes bitmap apply to zones and zone bitmap as well.

4.4. How we edited the free space management

4.4.1. Extents

Extent is a set of blocks which are handled as a single unit. To allocate and free an extent some development must be made in "***super.c***" file specifically at "***alloc_bit***" and "***free_bit***" functions.

"***alloc_bit***" function is responsible for allocating blocks needed by a file by refereing to each block of size 1 KB as 1 bit in the bitmap. Allocating a block is done by setting its corresponding bit in the bit map to 1. This allocation in the bitmap is done by dividing the bitmap to blocks, starting with *origin* (pointer to the starting of the bitmap), then dividing the block to words, then every word consits of bit which we would iterate over them to find the appropraite bit for allocation.

"***free_bit***" function is responsible for freeing block, that is no longer used by a file, by the same mechanism of the "***alloc_bit***" function but only resetting the value of a bit to 0.

4.4.2. Code Edits

1- In header file “super.h” located in “/minix/fs/mfs/”

Line 64: We defined “EXTSZ” to be used in “super.c” as user-defined number of blocks per extent.

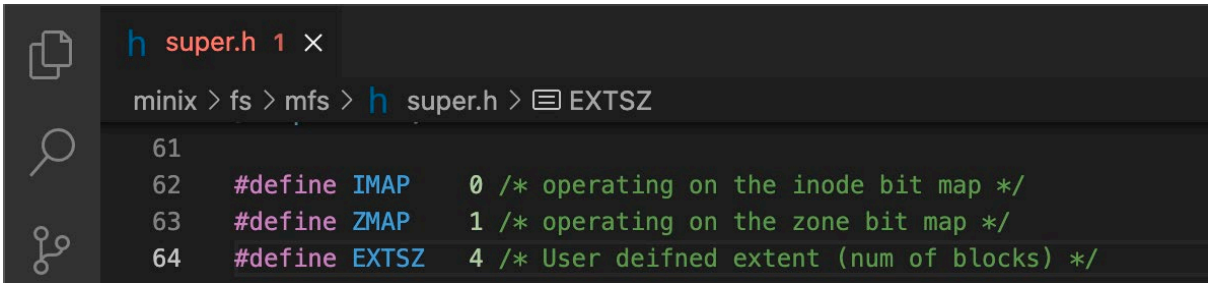


Figure 35: FS Code 1

2- In file “super.c” located in “/minix/fs/mfs/”

- ***alloc_bit*:** *Figure 23: FS Code 2*

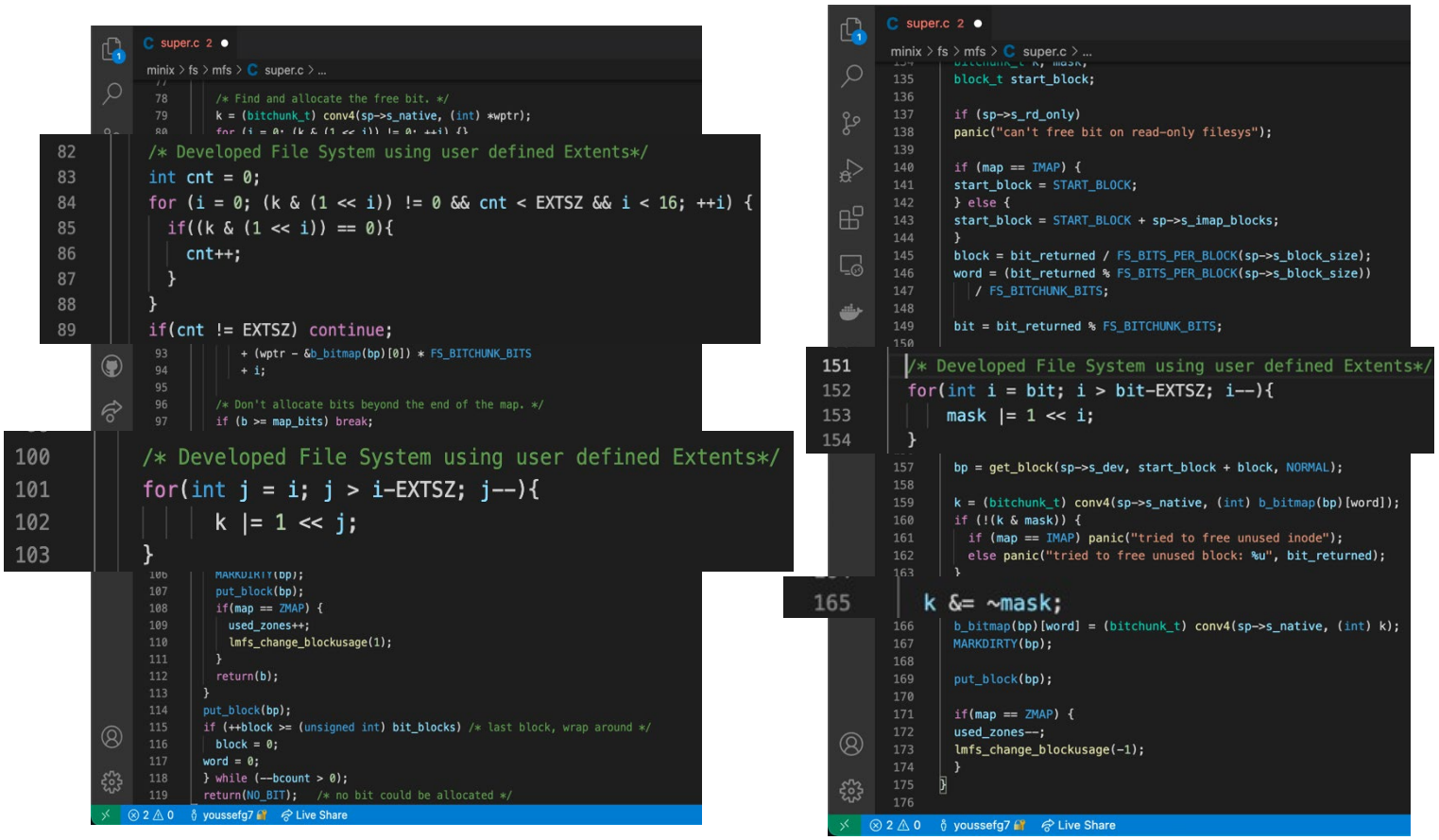
Lines 83-89: We are looping on the bits of *k* (every bit mapped with a block) to find *EXTSZ* free bits and allocate them later.

Lines 101-103: We are looping on size of *EXTSZ* to set bits of *k* to 1, these bits are correspinding to blocks, thus we are allocating blocks of extent size *EXTSZ*.

- ***free_bit:*** *Figure 22: FS Code 3*

Lines 152-154: We are looping starting from *bit* with size of *EXTSZ* to free bits corresponding to blocks by constructing *mask* (variable) setting the needed bits to be freed to 1 in *mask*.

Line 165: (k & ~mask) this makes all the ‘1’ bits *mask* to be 0 in *k*, thus setting them to be free bits in the bitmap and free blocks in the file system.



4.4.3. Performance testing

Firstly, we built MINIX 3 to confirm that these edits won't make any errors during compilation. The OS built successfully as shown in *Figure 24: FS Building successfully*.

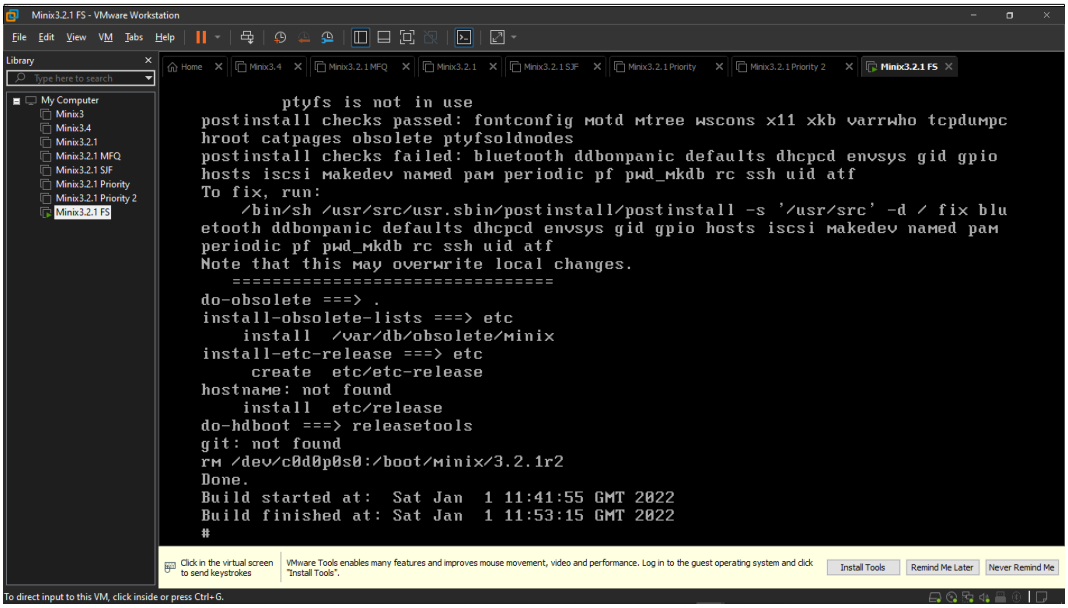


Figure 36: FS Building successfully

Secondly, the user changes number of blocks per extent in file “super.h” located in “fs/mfs” as in (*Figure 21: FS Code 1*). Generally, increasing the number of blocks per extents increases the read rate of the files but it can affect the disk utilization negatively due to internal fragmentation where a file is allocated a number of extents that gives him an extra space that can’t be used by other files.

All the following examples are tested on a **16 MB** disk with **1KB** block size.

- The following table shows number of files of size **20 KB** that can be saved on a disk of size **16 MB** (16*1024 = 16,384 KB) for different number of blocks per extent.

Blocks /extent	Files saved	Extents / file	Internal Fragmentation /file	Total Wasted Disk Space	Disk Utilization
1	819	20	0 KB	4 KB	99.9%
2	819	10	0 KB	4 KB	99.9%
3	780	7	1 KB	784 KB	95.2%
4	819	5	0 KB	4 KB	99.9%
8	682	3	4 KB	2744 KB	83.3%

- The following table shows number of files of size **21 KB** that can be saved on a disk of size **16 MB** (16*1024 = 16,384 KB) for different number of blocks per extent.

Blocks /extent	Files saved	Extents / file	Internal Fragmentation /file	Total Wasted Disk Space	Disk Utilization
1	780	21	0 KB	4 KB	99.9%
2	744	11	1 KB	760 KB	95.4%
3	780	7	0 KB	4 KB	99.9%
4	682	6	3 KB	2062 KB	87.4%
8	682	3	3 KB	2062 KB	87.4%

- The following table shows number of files of size **22 KB** that can be saved on a disk of size **16 MB** (16*1024 = 16,384 KB) for different number of blocks per extent.

Blocks /extent	Files saved	Extents / file	Internal Fragmentation /file	Total Wasted Disk Space	Disk Utilization
1	744	22	0 KB	16 KB	99.9%
2	744	11	0 KB	16 KB	99.9%
3	682	8	2 KB	1380 KB	91.6%
4	682	6	2 KB	1380 KB	91.6%
8	682	3	2 KB	1380 KB	91.6%

- The following table shows number of files of size **24 KB** that can be saved on a disk of size **16 MB** (16*1024 = 16,384 KB) for different number of blocks per extent.
  This case has an ideal disk utilization as files of this size have no internal fragmentation for all tested number of blocks per extent.

Blocks /extent	Files saved	Extents / file	Internal Fragmentation /file	Total Wasted Disk Space	Disk Utilization
1	682	24	0 KB	16 KB	99.9%
2	682	12	0 KB	16 KB	99.9%
3	682	8	0 KB	16 KB	99.9%
4	682	6	0 KB	16 KB	99.9%
8	682	3	0 KB	16 KB	99.9%