Ain Shams University

Faculty of Engineering

Computer and Systems Engineering Department

CSE335:Operating Systems (Fall 2022)

Operating System – Project Document

Team Members

Kareem Wael Hasan	2001151
Shady Emad Sabry	20P7239
Karim Bassel Samir	20P6794
Ahmed Tarek Aboelmakarem	20P6897
Fady Fady Fouad Fahim	20P7341
Matthew Sherif Shalaby	20P6785
Mostafa ayman mostafa	20p9883
Mina Fadi Mohsen	20P6996
Omar Hisham Gouda	20P6484
Philopateer Samuel Nassif	19P7542
Judy Khaled	20P6630
Aya Mohamed	20P7066

Table of Contents

Table of Figures

1. Requirement 1

1.1. Implementation

By default, there are 16 queues in Minix scheduler, each queue represents a priority where priority 0 is the highest (used for system tasks) and priority 15 is the lowest (used for idle process that runs when there are no processes ready to run.

Each queue runs by Round-Robin algorithm with default quantum = 200ms.

The goal of this requirement is to implement Round Robin (RR), Shortest Job First (SJF), Priority, and Multilevel Feedback Queue (MFQ) separately.

The user-defined scheduling parameters are defined in the configuration file **config.h** in */usr/src/include/minix*

Scheduling Parameters:

1. SCHEDULING_ALGORITHM: determines the scheduling algorithm to be used by the scheduler, 'r' = Round-Robin, 'p' = Priority, 's' = Shortest Job First, 'm' = Multilevel Feedback Queue.
2. MFQ_NUMBER_OF_QUEUES: number of levels in MFQ algorithms.
3. RR_QUANTUM: process' quantum in RR algorithm
4. BALANCE_TIMEOUT: how often to increase priority (aging effect) in seconds.

```c
/*=========================================================================*
 *  ASU Project Edit: User defined parameters          *
 *=========================================================================*/

#define SCHEDULING_ALGORITHM 's'        // determines the scheduling algorithm
                                        // r->round robin
                                        // p->priority
                                        // s->shortest job first
                                        // m->multilevel feedback queue

#define MFQ_NUMBER_OF_QUEUES 4          // number of levels in mfq algorithm

#define RR_QUANTUM 250                  // quantum in round-robin algorithm

#define BALANCE_TIMEOUT 5               // how often to balance queues (used for Aging in Priority algorithm)
                                        // in seconds

#define EXTENT_SIZE 4                   // number of blocks per extent (Requirement 3)
```

Figure 1 : Configuration File

5. NR_SCHED_QUEUES: determines number of queues used to represent priority in Priority scheduling and expected time in SJF scheduling, already implemented in Minix.

The parameters are assigned to variables in **schedule.c** in */usr/src/servers/sched* to be used inside that file and to add constraints on the parameters, one constraint is to keep number of levels in MFQ less than 15 levels and at least 2 levels.

For testing purposes, we implemented a rand() function to generate random numbers for Priority and SJF tests.

```
// Variables to configure scheduling
char asu_schedulingAlgorithm = SCHEDULING_ALGORITHM;            // determines the scheduling algorithm

unsigned int asu_numberOfQueues = MFQ_NUMBER_OF_QUEUES % 15;    // number of levels in mfq
                                                                // % 15 ensures no more than 14 level

unsigned int asu_quantum = RR_QUANTUM;                          // quantum in round-robin

int rand(){                                                     // Random number generator for test purposes
    static int prev = 3221;                                     // Anything
    prev = ((prev * prev) / 324) % 25238;
    return prev;
}
```

```
    if(asu_numberOfQueues < 2) asu_numberOfQueues = 2;  // Ensures no less than 2 levels in MFQ algorithm
```

Figure 2 : schedule.c variables and rand() function

Before implementing any algorithm, some modifications were made to show test results clearly and to prevent unwanted changes in processes' priority.

In **proc.c** in */usr/src/kernel,* we disabled the ability of a prosses to return to the head of the ready queue and assumed the only get enqueued in the back of the queue, just to show the test results clearly.

```
not_runnable_pick_new:
    if (proc_is_preempted(p)) {
        p->p_rts_flags &= ~RTS_PREEMPTED;
        if (proc_is_runnable(p)) {
            if (!is_zero64(p->p_cpu_time_left))
                // removed enqueue_head, all pre-emptions put processes at the back of the queue
                enqueue(p);
                // enqueue_head(p);
            else
                enqueue(p);
        }
    }
```

Figure 3 : proc.c Adjustments

In **schedule.c** we disabled the function balance_queues() that increases processes' priority every 5 seconds, this was made to prevent changes in priority in algorithms that requires a single priority (Queue) for all processes, which are RR and SJF, it is re-enabled in Priority algorithm to add the effect of aging.

```
/*===========================================================================*
 *              balance_queues                          *
 *===========================================================================*/

/* This function in called every 100 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 processes that have been bumped down,
 * and pulls them back up. This default policy will soon be changed.
 */
static void balance_queues(struct timer *tp)
{
    // Disable balance queues
    // 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, balance_queues, 0);
}
```

Figure 4 : disabled balance_queues() function

In **config.h,** we modified MIN_USER_Q to be 14 instead of 15, not allowing user processes to descend to the idle queue to prevent the idle process for interfering with the test results.

1.1.1. Round Robin

For the implementation of Round Robin algorithm, all user processes were assigned to a single arbitrary priority (10) so that all processes are in just one ready queue (Queue 10). This queue runs by RR algorithm by default.

All user processes are assigned quantum = RR_QUANTUM which is defined in the configuration file.

We avoided messing with system processes and system tasks to allow minix to work efficiently.

In **schedule.c,** inside function **do_start_scheduling()**, which is called by a process to start scheduling it for the first time, all user processes are given priority = 10 and time_slice = asu_quantum which maps to the user-defined quantum.

```
        //asu_edits
        if(asu_schedulingAlgorithm=='r'){
            rmp->priority = 10;
            rmp->time_slice = asu_quantum;
        }
```

Figure 5 : Assigning priorities and time slices

Inside function **do_noquantum()**, which is called by a process upon finishing its quantum, we disabled the line responsible for lowering priority every time a process consumes its entire quantum. Again, this was made to prevent unwanted changes in priority and keep all user processes in the same queue.

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

Figure 6 : Disabling the lowering of priorities

1.1.2. Shortest Job First (SJF)

For the implementation of Shortest Job First algorithm, we borrowed the idea of multiple queues for priority. Instead, the scheduling queues will represent the expected time for each process. Processes with lower expected time will have higher priority during scheduling which achieves the target of the algorithm.

To make the scheduling non-preemptive, we set the quantum to a very high value (INT_MAX = 232 –1) so that a process will never end its allowed time slice and be forced to release the CPU.

In schedproc.c in /usr/src/servers/sched where the process table exists, we added extra variable to store the expected time for the process. Since this algorithm is optimal and expecting the burst time of a process is likely very hard. For now, we randomized it for each process.

```
EXTERN struct schedproc {

    // Added attribute expectedTime to sort processes in running queue in SJF
    unsigned int asu_expectedTime;
```

Figure 7 : Extra variable added in schedproc.c

In schedule.c, inside function **do_start_scheduling()**, which is called by a process to start scheduling it for the first time, all user processes are given priority equal to the expected time, the expected time is randomized but optimally it is assumed to be calculated and within the range of 1 to number of queues defined by the user.

```
else{                                                          // SJF
    rmp->time_slice = INT_MAX;                                 // Non preemptive
    rmp->asu_expectedTime=rand()%(NR_SCHED_QUEUES-1) +1;       // Assign random expected time for a process,
                                                               // assume all processes do not exceed The number
                                                               // of queues defined by the user - 1 (15ms b default)
    rmp->priority = rmp->asu_expectedTime;
}
```

Figure 8: SJF version of do_start_scheduling()

1.1.3. Priority Based

The implementation of the Priority algorithm is almost like the default algorithm implemented in Minix with minor changes. The process will inherit its priority from the parent process (only in testing priority is randomized) and were given very large time_slice (INT_MAX = 2^{32} –1) so that the process doesn't release the CPU unless it is terminated.

These changes were made in **schedule.c**, inside function **do_start_scheduling().**

```c
    else if(asu_schedulingAlgorithm=='p'){
        rmp->priority = schedproc[parent_nr_n].priority;  // Inherit parent priority
        // rmp->priority = rand()%14 +1;   // random priority from 1 to 14, just for testing
        rmp->time_slice = INT_MAX;      // will not be preempted unless a process with higher priority comes
    }
```

Figure 9: Priority version of do_start_scheduling()

The balancing code, which was disabled for the previous algorithms, was reenabled to add the effect of aging to lower priority processes. in **schedule.c**, inside function **balance_queues().** This function is responsible for increasing priority for all ready processes to prevent starvation of low-priority processes. It is called periodically every number of seconds equal to what the user defined in BALANCE_TIMEOUT.

```c
static void balance_queues(struct timer *tp)
{
    struct schedproc *rmp;
    int proc_nr;

    // Disable balance queues for all algorithms other than Priority
    if(asu_schedulingAlgorithm == 'p'){
        for (proc_nr=0, rmp=schedproc; proc_nr < NR_PROCS; proc_nr++, rmp++) {  // All processes in ready queues
            if (rmp->flags & IN_USE) {
                if (rmp->priority > rmp->max_priority) {
                    rmp->priority -= 1; /* increase priority, Aging effect */
                    schedule_process_local(rmp); // reschedule the process with the new priority
                }
            }
        }
    }

    set_timer(&sched_timer, balance_timeout, balance_queues, 0);
}
```

Figure 10: balance_queue() for aging effect

The BALANCE_TIMEOUT is assigned to the variable balance_timout after multiplying it by system clock frequency to map to number of clock ticks, then the timer for balancing (aging) starts. This is already implemented in **schedule.c**, inside function **init_scheduling().**

```c
/*===========================================================================*
 *              start_scheduling                     *
 *===========================================================================*/

void init_scheduling(void)
{
    balance_timeout = BALANCE_TIMEOUT * sys_hz();
    init_timer(&sched_timer);
    set_timer(&sched_timer, balance_timeout, balance_queues, 0);
}
```

Figure 11: init_scheduling()

1.1.4. Multi-Level Feedback Queue

For the implementation of Multi-Level Feedback Queue algorithm, the number of levels is defined in the configuration file. Each level runs by Round-Robin with a quantum that increases as the process goes down in levels until in the last level it has a very large time_slice (INT_MAX = 2^{32} –1) which is practically First-Come-First-Serve (FCFS).

The chosen levels start from level 14 (one level higher than idle queue) and upward levels are chosen according to the number of levels the user defined in the configuration file. For example, if the number of levels = 4 then the chosen queues (priorities, or levels) are 11, 12, 13, and 14.

The quantum in each level is calculated from the simple formula $quantum = 10 \cdot priority$, where priority is the current level, except for the last level which has a quantum of (INT_MAX = 2^{32} –1). For example, if the number of levels = 4 then the chosen queues (priorities, or levels) are 11, 12, 13, and 14. And the quantum in each level are:

1. Quantum_{11} = 110ms.
2. Quantum_{12} = 120ms.
3. Quantum_{13} = 130ms.
4. Quantum_{14} = INT_MAX = (2^{32} –1)ms.

In **schedule.c,** inside function **do_start_scheduling()**, user processes are given priority higher than the lowest queue by the number of levels and given a quantum according to the formula above.

First level = (15 – asu_numberOfQueues): the bottom level is always 14, the top level is ***asu_numberOfQueues*** levels over the lowest queue.

Example: if asu_numberOfQueues = 4, then 1st level = 15-4=11; and the levels are 11,12,13,14.

```
else if(asu_schedulingAlgorithm=='m'){
    rmp->priority = (NR_SCHED_QUEUES-1) - asu_numberOfQueues; // Using queues from the lowest upwards, starts from
                                                               // number of levels higher than the lowest queue (15)
                                                               // equals what the user defined
    rmp->time_slice = rmp->priority * 10;                      // Every level runs by round-robin with increasing quantum
                                                               // except the lowest level runs by FCFS
}
```

Figure 12 : Inside do_start_scheduling()

Inside function **do_noquantum()**, which is called by a process upon finishing its quantum in all levels other than the last level, the priority of the process is lowered thus moving to the next level (Feedback) and the new quantum assigned to the process according to the formula or INT_MAX if in the last level.

```c
if(asu_schedulingAlgorithm=='m'){
    if (rmp->priority < MIN_USER_Q) {
        rmp->priority += 1;                      // lower priority, increase level

        if(rmp->priority < MIN_USER_Q)
            rmp->time_slice = rmp->priority * 10;  // If not in the lowest level, assign slightly increasing
                                                   // quantum (10 * the new (increased) level)
        else
            rmp->time_slice = INT_MAX;             // If in the lowest level -> FCFS
    }
}
```

Figure 13 : Inside do_noquantum()

1.2. Analysis of the Algorithms

1.2.1. Round Robin

1.2.1.1. Introduction

This algorithm is named after the round-robin principle. In the round-robin principle, each person takes turns sharing something evenly. It is known as the oldest and simplest scheduling algorithm used primarily for multitasking. In Round-Robin scheduling, each task is executed sequentially for a limited time slot (determined by the operating system developer) in a circular queue. This algorithm also offers starvation free execution of processes, which means that no process will have to wait for a very large amount of time to reach its turn.

The Round-Robin algorithm is a pre-emptive algorithm which means that in this algorithm, Processes can be interrupted during their execution.

1.2.1.2. How the algorithm works

In the Round-Robin Algorithm, Processes are executed in sequential order and each process runs for a certain amount of time (determined by the operating system developer). After the amount of time has passed the process is then interrupted by the process next in line to be executed for the same amount of time, even if it hasn't finished execution.

Let’s assume the following processes want to run in our operating system using the Round – Robin algorithm.

Process Queue	Burst time	Time Slot
P1	3	3
P2	6	
P3	9	

These processes will run sequentially. Each process will run for the selected time slot (3) until it has finished execution. P1 will begin execution first for 3 seconds. While the other processes remain in the waiting queue.

P1							
0	3	6	9	12	15	18	21

As P1 only has a burst time of 3 seconds, this means that it has finished execution and now our algorithm will iterate on P2 and P3 which are the remaining processes in our waiting queue.

P1	P2						
0	3	6	9	12	15	18	21

P1	P2	P3					
0	3	6	9	12	15	18	21

P1	P2	P3	P2				
0	3	6	9	12	15	18	21

At this moment, P2 has also finished executing and now P3 is the only process left.

P1	P2	P3	P2	P3			
0	3	6	9	12	15	18	21

As all the other processes were finished, P3 will continue taking a time slot of 3 seconds until it has finished execution.

P1	P2	P3	P2	P3	P3		
0	3	6	9	12	15	18	21

1.2.1.3. Advantages of Round – Robin Algorithm

- Treats all the processes equally without any priority as it allows all the processes to get an equal amount of allocation of CPU, which means that it never faces the issue of starvation.
- No priority scheduling is involved (In some cases, this might be seen as a disadvantage).
- Easily implementable.
- Best performance in terms of average response time.
- Doesn't depend on burst time of processes.

1.2.1.4. Disadvantages of Round – Robin Algorithm

- The throughput depends heavily on the time quantum selected. Therefore, if the chosen time quantum is low, the processor output will be reduced, and the scheduler will spend more time on context switching.
- Important processes are not given priority, which may not be convenient in some cases.
- If the processes have varied burst times among them, determining a suitable quantum time will be difficult.
- Selecting a large quantum time will make this algorithm act as a **F**irst **C**ome **F**irst **S**erved algorithm which comes with its own disadvantages.
- The average waiting time is often long.

1.2.2. Shortest Job First (SJF)

1.2.2.1. Introduction

Shortest Job First (or SJF) is a scheduling algorithm that selects the process with the smallest execution time in the ready queue to be executed next. This method is non-pre-emptive (Which means that it ***cannot*** interrupt processes during execution). However, it has a pre-emptive version (Which means that it ***can*** interrupt processes during execution) called Shortest Remaining Time First (SRTF).

Successful implementation of this algorithm would require the processor to be notified in advance of the execution/burst time of the processes, which is not always practical. It is also difficult to predict the execution time of processes. Therefore, the Shortest Job First algorithm is not easy to implement in the operating system.

1.2.2.2. How the algorithm works

The Shortest Job First algorithm works in a very simple manner. It checks the execution time of processes available to be executed in the ready queue then chooses the process with the lowest execution time and puts it in the CPU for execution.

Let's assume the following processes are to be executed by *Non-Pre-emptive* Shortest Job First:-

Process	Burst time
P1	21
P2	3
P3	6
P4	2

Assuming all processes arrived at time = 0. The shortest process will be executed first, Hence the Gantt chart in the end will be:-

P4	P2	P3	P1

0 2 5 11 32

As P4 had the lowest burst time (2), It was executed first. Then the lowest process from the remaining processes was P2 with a burst time of 3. Then P3 was executed as it had lower burst time than P1. Finally, P1 was executed, and all our processes were finished.

The problem with this non-pre-emptive SJF is that if all of our processes do not arrive at the same time, Processes with a very short burst time will have to wait for the running process to finish execution as we cannot interrupt the process execution in non-pre-emptive SJF. Which could lead to starvation.

Let’s assume the following processes are to be done by *Pre-emptive* Shortest Job First:

Process	Arrival time	Burst time
P0	3	2
P1	2	4
P2	0	6
P3	1	4

In this example we had to assume that the processes’ arrival time is different to show the ‘Pre-emptive’ part of SJF. The Gantt chart in the end will be:

P2	P3	P0	P1	P2	
0	1	5	7	11	16

As you can see P2 arrived first, and no other process had arrived yet, therefore it was chosen as the shortest process available in ready queue. But then P3 arrived and had a shorter burst time than the currently running process (P2). As this is a Pre-emptive SJF, the scheduler was able to interrupt P2 and run the process with shorter burst time which is P3. After that, many processes started to arrive, and they all had a shorter burst time than P2. Therefore, they were executed first and after they were done P2 was executed.

As you can see in this example, The starvation problem was avoided. Pre-emptive SJF is also known as Shortest Remaining Time First, because at any given time the process with the shortest remaining time will run first.

1.2.2.3. Advantages of SJF Algorithm

- Suitable for processes that have known burst times.
- Its algorithmic approach is useful for batch processing were waiting for job completion is not critical.
- Minimum average waiting time for a given set of processes among all scheduling algorithms.

1.2.2.4. Disadvantages of SJF Algorithm

- Difficult to implement, As the operating system may not know the burst time of the processes which will result in the operating system not being able to sort them.
- Non-pre-emptive SJF could lead to starvation of other processes or a very large turnaround time.
- Burst time of processes must be known in advance, which may not be possible for some processes.

1.2.3. Priority Based

1.2.3.1. Introduction

In this algorithm, each process is given a priority number. Priority scheduling in operating systems is a method of scheduling processes based on their given priority. Where the higher priority processes are executed before the lower priority processes.

There are several factors used to determine process/job priority such as the process's burst time, it's memory requirements and the ratio of average I/O to average CPU burst time. These process priorities are expressed as simple integers in a selected fixed range such as 0 to 10 or 0 to 4095. These numbers can vary from system to another.

1.2.3.2. How the algorithm works

There are two types of priority scheduling algorithms in OS which are: Pre-emptive Scheduling (Which means that it ***can*** interrupt processes during execution) and Non-Preemptive Scheduling (Which means that it ***cannot*** interrupt processes during execution). The priorities are determined in ascending order, which means that a process with priority 0 has higher priority than all of the other processes.

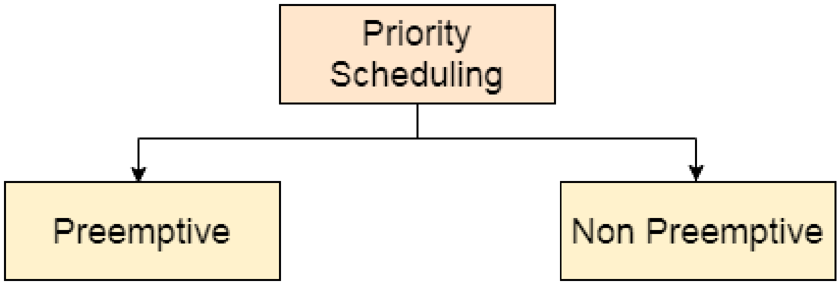


Figure 14 : Types of priority scheduling

Let’s assume the following processes will run using the Non-Preemptive Scheduling priority algorithm: -

Process	Arrival Time	Burst Time	Priority
P1	0	4	2
P2	0	3	3
P3	3	4	1

As we can see in the previous table, P1 and P2 both arrive at the same time (0), To determine which process begins execution first we must look at their priority. Since the priority of P1 higher than the priority of P2, P1 will begin execution first and will not be interrupted even if a process with higher priority arrives during its execution.

Time	0	1	2	3	4
Process	P1	P1	P1	P1	P1

As we can see from the previous table, P3 arrived when time was equal 3, And even though it has a higher priority than P1, It was not able to start execution as we are using a non-preemptive priority scheduling. Therefore, the final Gantt chart of this example will be:

P1	P3	P2

0 4 8 11

Now let’s assume the same example but this time, we will use Preemptive priority scheduling:

Time	0	1	2	3	4
Process	P1	P1	P1	P3	P3

As we can see from the previous table, Once P3 arrived P1 was interrupted and P3 was scheduled to be executed. This is because P3 has higher priority than P1 and therefore must finish execution first. Therefore, the final Gantt chart of this example will be:

P1	P3	P1	P2

0 3 7 8 11

1.2.3.3. Advantages of Priority Based Algorithm

- The relative importance of each process is precisely defined using priorities.
- Simple and easy to understand.
- Important processes in the operating system will not need to wait to be executed.

1.2.3.4. Disadvantages of Priority Based Algorithm

- Processes with low priority may be subject to starvation.
- In some cases, it is difficult to decide which processes are to be given higher priority.
- If the computer suddenly crashes, All the low priority processes will be lost.
- *(In Non-Pre-emptive priority scheduling)* Any new process that becomes ready for execution during the execution of another process will have to wait for that other process to finish. Which could be a long time.

1.2.4. Multi-Level Feedback Queue (MFQ)

1.2.4.1. Introduction

The Multi-Level feedback queue is a scheduling algorithm that divides processes into multiple ready queues based on processor demand, prefers processes with short CPU burst, And prefers processes that have high I/O bursts. (The I/O bound processes sleep in the waiting queue to give the other processes some CPU time).

In the Multi-Level feedback queue scheduling, the ***ready queue*** is divided into separate queues. The processes can then be moved between these queues which could result in decreasing the waiting time for other processes.

The movement of processes throughout the queues depends on their own burst time. If a process is consuming too much CPU time, it will be moved to a lower priority queue to give the other processes a chance to be executed. If the process is waiting for an input or an output, it will be moved to a higher priority queue. If a process starves after waiting too long in a low priority queue, The process will then be moved into a higher priority queue.

The queues themselves can contain different scheduling algorithms such as First Come First Served (FCFS), Round Robin, etc....

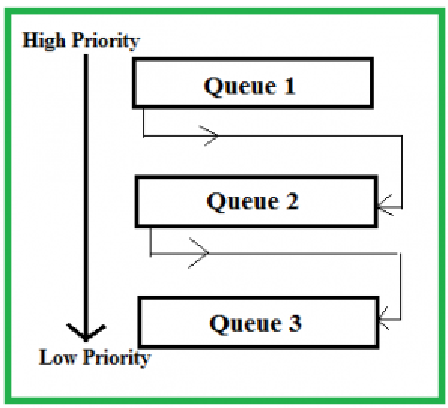


Figure 15 : Multi-level Feedback queue

1.2.4.2. How the algorithm works

A multi-level feedback queue scheduler is generally defined by the following parameters:

- Number of queues
- A selected scheduling algorithm for each queue. (Round Robin / FCFS / Priority)
- A method to determine when a process should be moved to a higher priority queue.
- A method to determine when a process should be moved to a lower priority queue.
- A method to determine which queue a process should enter.

In multi-level feedback queue, Multiple queues are used, where each queue uses a different scheduling algorithm.

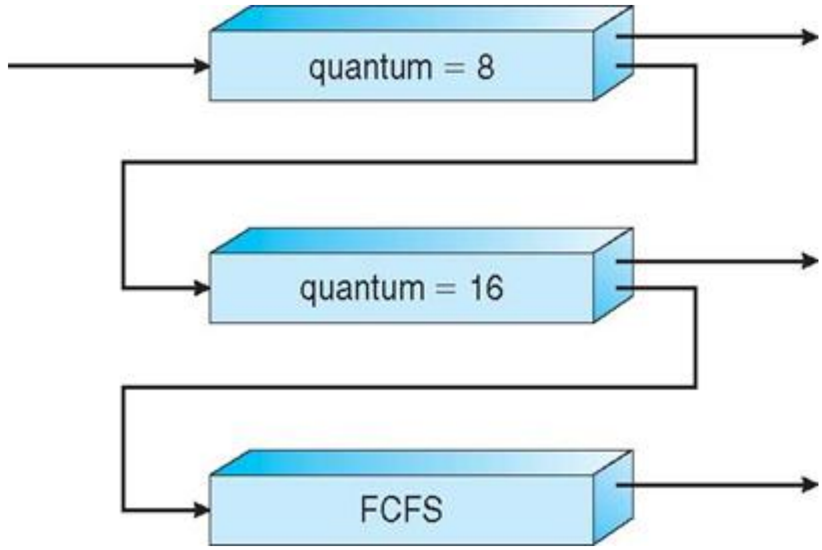


Figure 16 : Multi-level feedback queue visualization (MFQ)

If a process starts execution, then it enters the first queue, in our case this queue uses a Round Robin scheduling algorithm with a time quantum = 8 units. If that process finishes its execution during those 8 units. Then it will leave the system as shown in the figure below. (Assuming P1's burst time = 8 units).

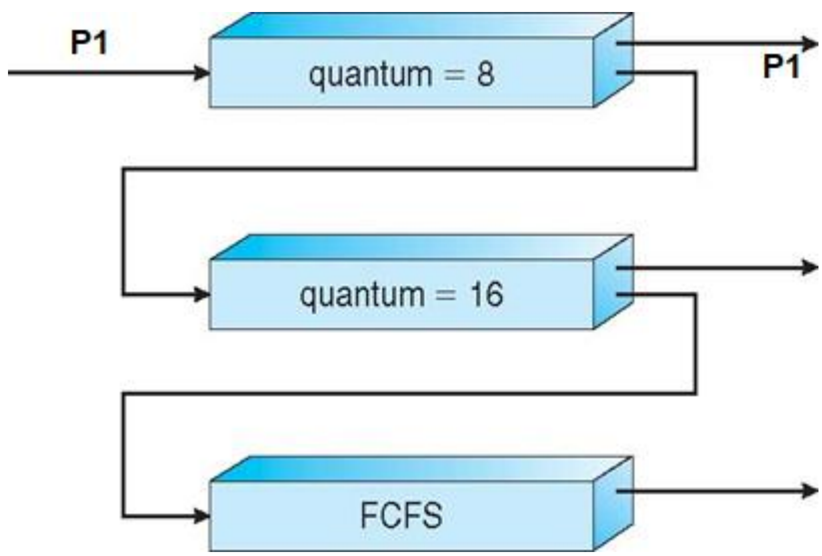


Figure 17 : Process with a burst time of 8 units (MFQ)

Now let's assume a process P2 whose burst time is 10 units. P2 will first enter the first queue, then execute for 8 units. After that, the priority of P2 will decrease and the process will be moved into queue 2 which has a quantum time of 16 units. Which means that P2 will be able to finish its execution in the second queue and exit the system as shown in the figure below.

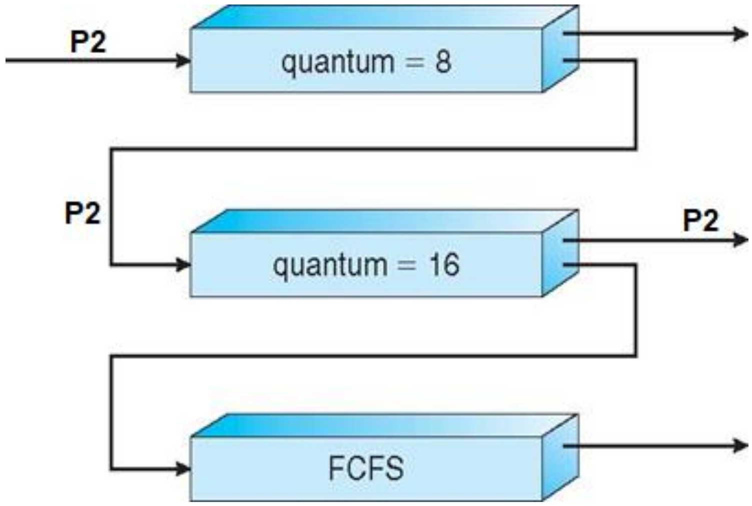


Figure 18 : Process with a burst time of 10 units (MFQ)

Now let's assume a process P3 whose burst time is 26 units. P3 will first enter the first queue, then execute for 8 units. After that, the priority of P3 will decrease and the process will be moved into queue 2 which has a quantum time of 16 units. P3 will still need to run for 2 more units after queue 2. Therefore, P3 will then be demoted to the lowest priority queue which is queue 3 which uses a FCFS (**F**irst **C**ome **F**irst **S**erved) scheduling algorithm. P3 will finish its execution in the third queue then exit the system as shown in the figure below.

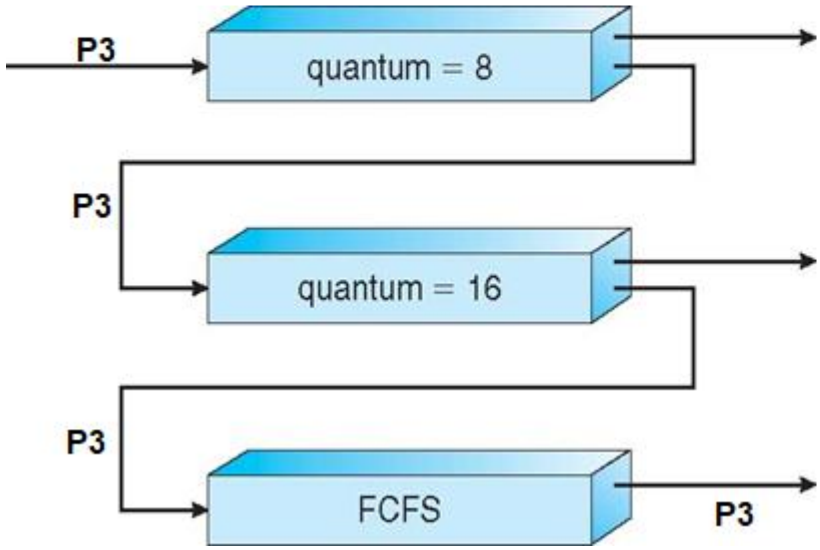


Figure 19 : Process with a burst time of 26 units (MFQ)

1.2.4.3. Advantages of Multi-Level Feedback Algorithm

- Processes are not permanently assigned to a certain queue as in multi-Level queue scheduling.
- Allows different processes to move through different queues. (Dynamic Priority)
- Prevents starvation by moving long waiting processes in a low priority queue to a high priority queue.
- More flexible than multi-Level queue scheduling.

1.2.4.4. Disadvantages of Multi-Level Feedback Algorithm

- Very complex algorithm to implement.
- Determining the best scheduling algorithms to use in queues in advance may be tough.
- Movement of processes through different queues generates CPU overheads

1.3. Comparative Analysis between the scheduling algorithms

Scheduling Algorithm	Round Robin	Shortest Job First	Priority	Multi-Level Feedback
Average Waiting Time	Longest Average waiting time	Small average waiting time comparted to FCFS	Small average waiting time comparted to FCFS	Small average waiting time comparted to FCFS
Preemptive/ Non-preemptive	Preemptive algorithm	Nom-Preemptive algorithm*	Both	Preemptive algorithm
Starvation occurs	No	Possible	Possible	Possible
Overhead	Minimum	May be high	Depends on priority policy	May be high
Usage	Used by process and network schedulers in computing	Used for batch-type processing where waiting for jobs to complete is not critical	Used in batch systems	Used to divide processes into different classes to meet their own scheduling needs

*: Has a Preemptive version: Shortest Remaining Time First (SRTF)

Scheduling Algorithm	Round Robin	Shortest Job First	Priority	Multi-Level Feedback queue
Advantages	• Treats all of the processes equally without any priority as it allows all the processes to get an equal amount of allocation of CPU • It never faces the issue of starvation. • No priority scheduling is involved (In some cases, This might be seen as a disadvantage). • Easily implementable. • Best performance in terms of average response time. • Doesn’t depend on	• Suitable for processes that have known burst times. • Its algorithmic approach is useful for batch processing where waiting for job completion is not critical. • Minimum average waiting time for a given set of processes among all scheduling algorithms.	• The relative importance of each process is precisely defined using priorities. • Simple and easy to understand. • Important processes in the operating system will not need to wait to be executed.	• Processes are not permanently assigned to a certain queue as in Multi-Level queue scheduling. • Allows different processes to move through different queues. (Dynamic Priority) • Prevents starvatio n • More flexible than Multi-Level queue scheduling.

	burst time of processes.			
Disadvantages	• The throughput depends heavily on the time quantum selected. • Important processes are not given priority • Determining a suitable quantum time could be difficult. • Selecting a large quantum time will make this algorithm act as a **F**irst **C**ome **F**irst **S**erved algorithm • The average waiting time is often long.	• Difficult to implement, As the operating system may not know the burst time of the processes which will result in the operating system not being able to sort them. • Non pre-emptive SJF could lead to starvation of other processes or a very large turnaround time. • Burst time of processes must be known in advance, Which may not be possible for some processes.	• Processes with low priority may be subject to starvation. • In some cases, It is difficult to decide which processes are to be given higher priority. • If the computer suddenly crashes, All the low priority processes will be lost. • *(In Non-Pre-emptive priority scheduling)* Any new process that becomes	• Very complex algorithm to implement. • Determining the best scheduling algorithms to use in queues in advance may be tough. • Movement of processes through different queues generates CPU overheads

			ready for execution during the execution of another process will have to wait for that other process to finish. Which could be a long time.	

1.4. Test cases and Results

The test is conducted by creating 5 processes in approximately the same time and observing how the scheduler deals with them according to each algorithm while calculating the turnaround time and waiting time.

For testing purposes, a printf() line added Inside function **do_noquantum()** and **do_stop_scheduling(),** which is called by a process upon finishing**,** to get details about each process releasing the CPU.

Turnaround time and waiting time results were collected in test files attached to the report.

The average turnaround time and average waiting time for each algorithm is calculated, compared and explained in this section.

1.4.1. Round Robin

Test in Action:

Processes take turns in the same queue with quantum = 250 that we defined.

IMPORTANT NOTE: the quantum in printf() refers to the quantum given to the process not the actual time it stayed in the CPU.

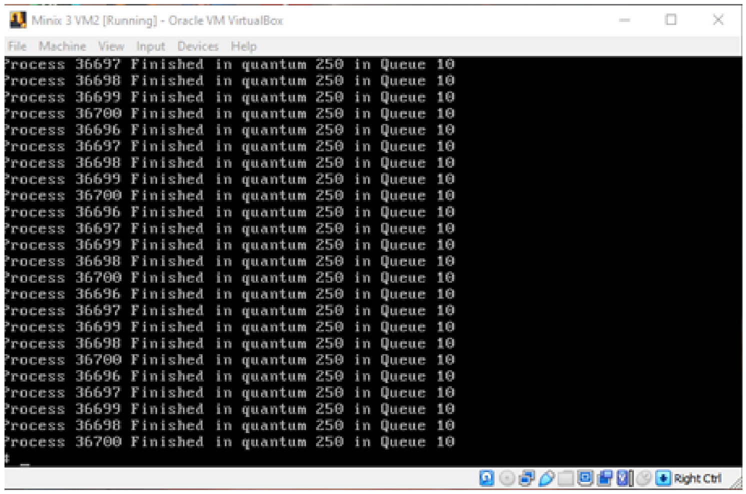


Figure 20 : RR Tests running on MINIX 3

Statistical Results:

process 1, Turnaround time = 9033.333000, Waiting time = 7823.000000 ms

process 2, Turnaround time = 9050.000000, Waiting time = 7840.000000 ms

process 3, Turnaround time = 9066.666000, Waiting time = 7856.000000 ms

process 4, Turnaround time = 9066.666000, Waiting time = 7856.000000 ms

process 5, Turnaround time = 9066.666000, Waiting time = 7856.000000 ms

test took: 9.066666

$$Average\ Turnaround\ Time\ =\ 9056.662ms$$

$$Average\ Waiting\ Time\ =\ 7851.6ms$$

1.4.2. Shortest Job First (SJF)

Test in Action:

Processes with smaller expected time get the CPU and don’t release it until it is finished first.

IMPORTANT NOTE: the last process is the parent process that creates the 5 child processes that are used to test, this parent process calls system call **wait()** to prevent its termination until all its children are terminated.

System processes often interfered with the test making results unclear, so the test was repeated several times until only 5 processes and their parent show up.

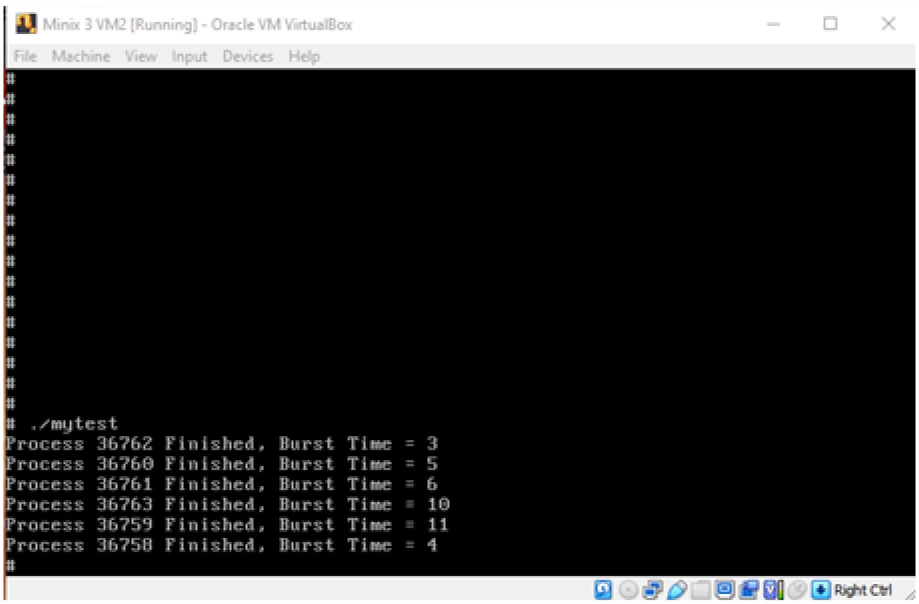


Figure 21 : SJF Tests on MINIX 3

Since this algorithm is optimal and impossible to implement seriously in real life, statistical results can’t be collected practically. However, the test in action indicates that the implemented algorithm conforms to what has been discussed in the earlier section.

1.4.3. Priority Based

Test in Action:

Processes with higher priorities finishes first then next lower-priority process takes the CPU.

IMPORTANT NOTE: the last process is the parent process that creates the 5 child processes that are used to test, this parent process calls system call **wait()** to prevent its termination until all its children are terminated.

System processes often interfered with the test making results unclear, so the test was repeated several times until only 5 processes and their parent show up.

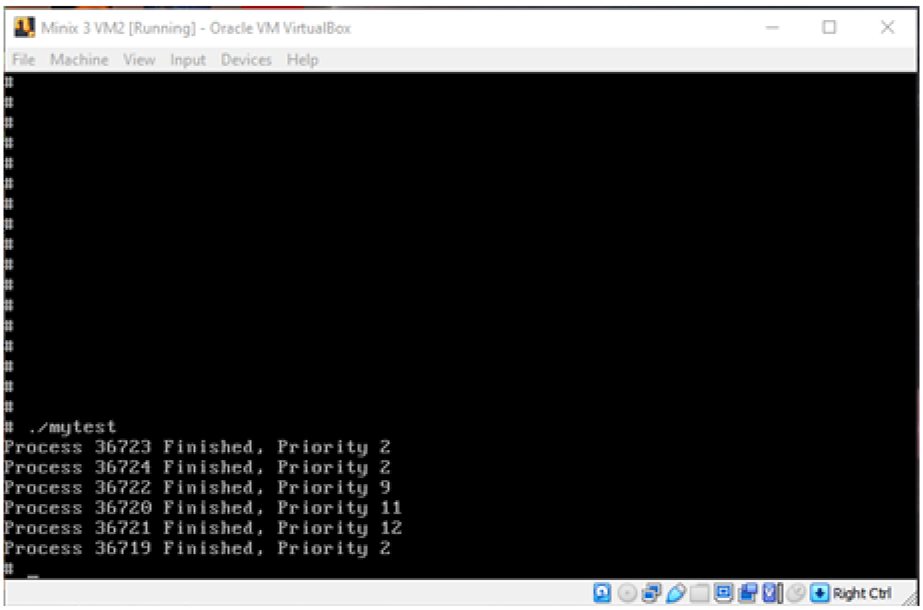


Figure 22 : Priority based tests on MINIX 3

Statistical Results:

process 4, Turnaround time = 1950.000000, Waiting time = 740.000000 ms

process 5, Turnaround time = 4016.667000, Waiting time = 2806.000000 ms

process 3, Turnaround time = 6183.333000, Waiting time = 4973.000000 ms

process 1, Turnaround time = 8300.000000, Waiting time = 7090.000000 ms

process 2, Turnaround time = 10433.333000, Waiting time = 9223.000000 ms

test took: 10.433333

$Average\ Turnaround\ Time$=6176.6666$ms$

$Average\ Waiting\ Time$=4966.4ms

1.4.4. Multi-Level Feedback Queue

Test in Action:

We defined number of levels = 4. Used queues are 11, 12, 13, and 14.

Processes take turn in each queue in round robin fashion, descending in levels every time a process finishes each quantum, the last level has ridiculously high quantum so that the process is guaranteed to finish before exhausting all its quantum.

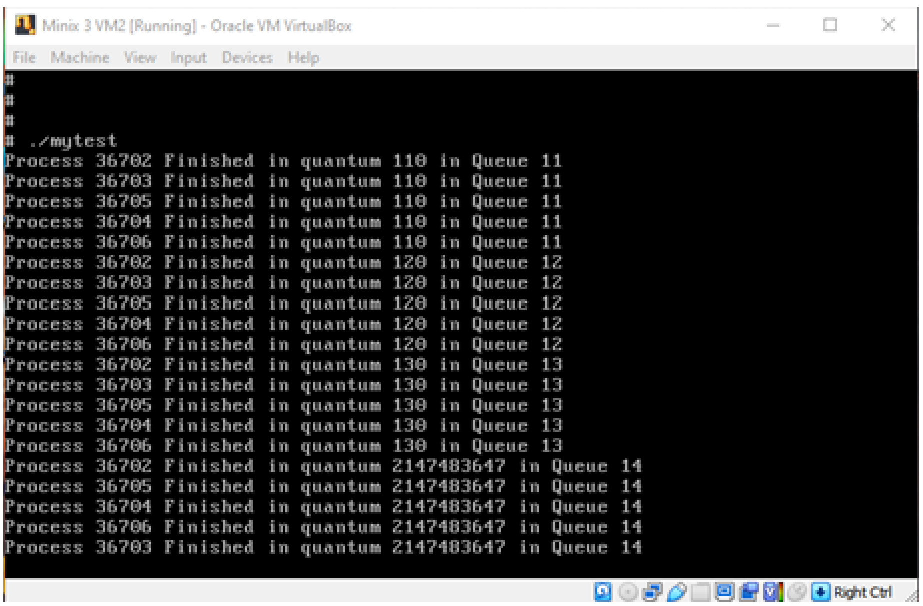


Figure 23 : MLFQ tests on MINIX 3

Statistical Results:

process 3, Turnaround time = 6916.667000, Waiting time = 5706.000000 ms

process 5, Turnaround time = 7916.667000, Waiting time = 6706.000000 ms

process 1, Turnaround time = 8933.334000, Waiting time = 7723.000000 ms process 4, Turnaround time = 9933.334000, Waiting time = 8723.000000 ms

process 2, Turnaround time = 10950.000000, Waiting time = 9740.000000 ms

test took: 10.905000

$$Average\ Turnaround\ Time = 8930.0004ms$$

$$Average\ Waiting\ Time = 7719.6ms$$

1.4.5. Comparative Analysis and Explanation of the Results

Round Robin algorithm guarantees fair share of the CPU for all processes, the turnaround time and waiting time were approximately equal for processes that has approximately equal burst time. The average turnaround time and average waiting time are very close to the turnaround time and waiting time of each process. Round Robin results in lower Average Response Time.

Shortest Job First is an optimal algorithm that doesn't exist in real life. It gives the priority to the processes with less expected burst time resulting in lower waiting time for them. SJF results in lower Average Waiting Time and thus lower Average Turnaround Time.

Priority algorithm sorts the processes in the ready queue based on their given priority. Processes with higher priority has lower waiting time and thus lower turnaround time and vice versa. The average waiting time and average turnaround time depends on the order of the processes, their respective priorities, start time, and burst time. However, it often results in lower Average Waiting Time and Average Turnaround Time than Round Robin, at the expense of very high waiting time, response time, and turnaround time for lower priority processes.

First Come First serve algorithm used in the last queue in the MFQ algorithm is a non-preemptive algorithm that guarantees little to no waiting time for processes that comes early in the ready queue, while causes very high waiting time for processes that comes later. FCFS is almost like our implementation of Priority algorithm except that the priority is the order of entering the ready queue. Thus, resulting in lower Average Waiting time and Average Turnaround Time than Round-Robin.

Multilevel Feedback Queue Algorithm works as a set of ready queues that runs by Round Robin algorithm and a single queue running by FCFS algorithm. Thus, the results conform to an expected Average Waiting Time and Average Turnaround Time somewhere between RR and FCFS.

2. Requirement 2

2.1. Hierarchical Paging

Here we applied the hierarchical paging by adding a second level, we modified the 32 bit virtual address, page table entry is now taking 7 bits instead of 8 and the extra bit used to decide the second level, to do that we changed the page table entry from 256 to 128 to make it use 7 bits only and leave 1 bit for level 2, here are some snapshots to the modifications we did in the code:

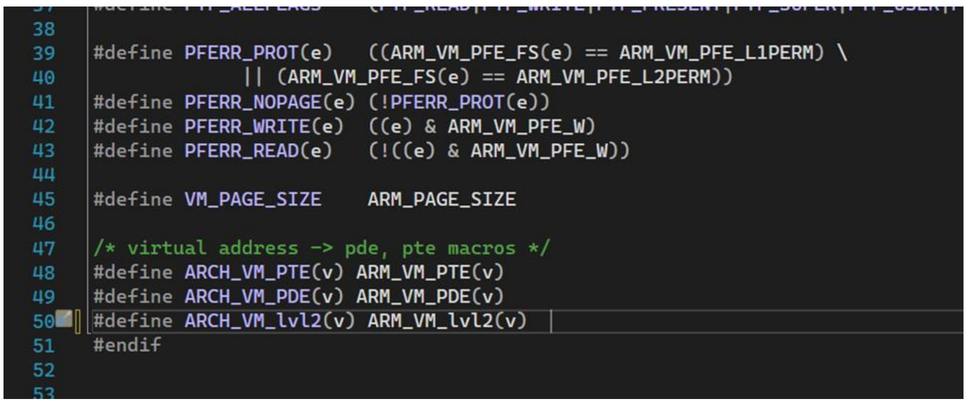


Figure 24 : Level 2 VM address

Lvl2 function definition to get the level2 bit by taking the unused bit of the page table entry

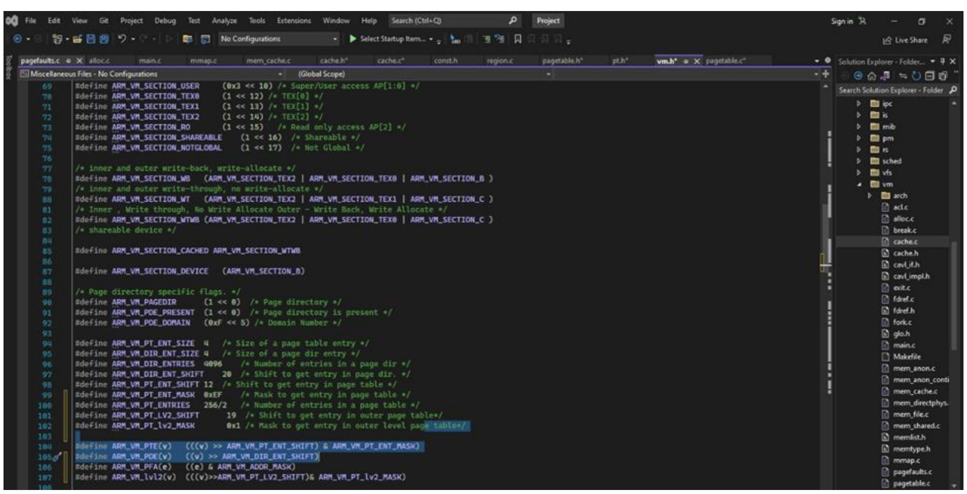


Figure 25 : Shifting marks

First 12 bits for page directory entry

The next bit for level2 as we changed the page table entry to 128 instead of 265

The next 7 bits for page table entry

The last 12 bits for page offset

```c
#ifndef _PT_H
#define _PT_H 1

#include <machine/vm.h>

#include "vm.h"
#include "pagetable.h"

/* A pagetable. */
typedef struct {
    /* Directory entries in VM addr space - root of page table.  */
    u32_t *pt_dir;      /* page aligned (ARCH_VM_DIR_ENTRIES) */
    u32_t pt_dir_phys;  /* physical address of pt_dir */

    /* Pointers to page tables in VM address space. */
    u32_t * pt_pt[ARCH_VM_DIR_ENTRIES][2];
    /* When looking for a hole in virtual address space, start
     * looking here. This is in linear addresses, i.e.,
     * not as the process sees it but the position in the page
     * page table. This is just a hint.
     */
    u32_t pt_virtop;
} pt_t;
#define CLICKSPERPAGE (VM_PAGE_SIZE/CLICK_SIZE)
#endif
```

Figure 26 : Page table struct after modification

Pt_pt was 1d array of pointers, we modified it to a 2D array of pointers, directory is pointing on the second level and second level is pointing on the page table, we modified page table entry to be 128 instead of 265 and used the 8th bit of the 265 as the second level

```c
995
996  /*========================================================================*
997   *              pt_new                          *
998   *========================================================================*/
999  int pt_new(pt_t * pt)
1000 {
1001     /* Allocate a pagetable root. Allocate a page-aligned page directory
1002      * and set them to 0 (indicating no page tables are allocated). Lookup
1003      * its physical address as we'll need that in the future. Verify it's
1004      * page-aligned.
1005      */
1006     int i, r;
1007
1008     /* Don't ever re-allocate/re-move a certain process slot's
1009      * page directory once it's been created. This is a fraction
1010      * faster, but also avoids having to invalidate the page
1011      * mappings from in-kernel page tables pointing to
1012      * the page directories (the page_directories data).
1013      */
1014     if (!pt->pt_dir &&
1015         !(pt->pt_dir = vm_allocpages((phys_bytes*)&pt->pt_dir_phys,
1016             VMP_PAGEDIR, ARCH_PAGEDIR_SIZE / VM_PAGE_SIZE))) {
1017         return ENOMEM;
1018     }
1019
1020     assert(!((u32_t)pt->pt_dir_phys % ARCH_PAGEDIR_SIZE));
1021     for (int k = 0; k < 2; k++) {
1022         for (i = 0; i < ARCH_VM_DIR_ENTRIES; i++) {
1023             pt->pt_dir[i] = 0; /* invalid entry (PRESENT bit = 0) */
1024             pt->pt_pt[i][k] = NULL;
1025         }
1026     }
1027     /* Where to start looking for free virtual address space? */
1028     pt->pt_virtop = 0;
1029
1030     /* Map in kernel. */
1031     if ((r = pt_mapkernel(pt)) != OK)
1032         return r;
1033
1034     return OK;
1035 }
```

Figure 27 : pt_new function

New page table allocated in this function, nested for loop at 1021 to iterate over the second dimension

```c
    int p, pages;

    assert(!(bytes % VM_PAGE_SIZE));

    pages = bytes / VM_PAGE_SIZE;

    for (p = 0; p < pages; p++) {
        int pde = ARCH_VM_PDE(v);
        int pte = ARCH_VM_PTE(v);
        int lvl2 = ARCH_VM_lvl2(v);

        assert(!(v % VM_PAGE_SIZE));
        assert(pte >= 0 && pte < ARCH_VM_PT_ENTRIES);
        assert(pde >= 0 && pde < ARCH_VM_DIR_ENTRIES);

        /* Page table has to be there. */
        if (!(pt->pt_dir[pde] & ARCH_VM_PDE_PRESENT))
            return EFAULT;

        /* Make sure page directory entry for this page table
         * is marked present and page table entry is available.
         */
        assert((pt->pt_dir[pde] & ARCH_VM_PDE_PRESENT) && pt->pt_pt[pde]);

        if (!(pt->pt_pt[pde][lvl2][pte] & ARCH_VM_PTE_PRESENT)) {
            return EFAULT;
        }

#if defined(__i386__)
        if (write && !(pt->pt_pt[pde][pte] & ARCH_VM_PTE_RW)) {
#elif defined(__arm__)
        if (write && (pt->pt_pt[pde][pte] & ARCH_VM_PTE_RO)) {
#endif
            return EFAULT;
        }

        v += VM_PAGE_SIZE;
    }

    return OK;
}
```

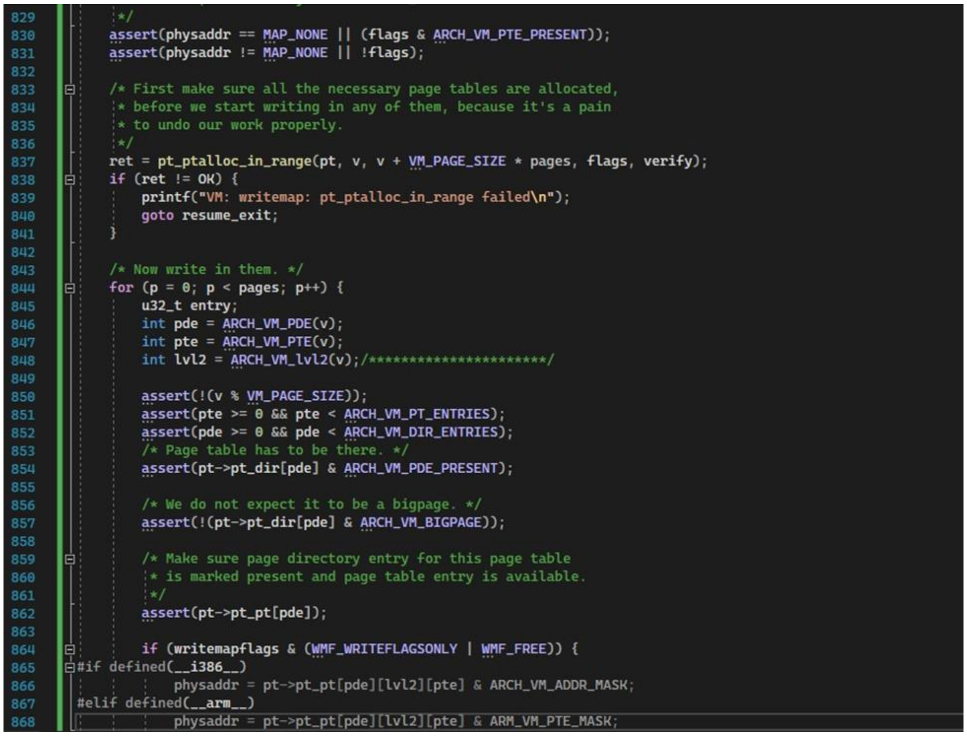
Figure 28 : pt_new function cont

Access method to the array changed to 3 levels

```c
 *                    pt_checkrange                    *
 *=========================================================================*/
int pt_checkrange(pt_t * pt, vir_bytes v, size_t bytes,
    int write)
{
    int p, pages;
    assert(!(bytes % VM_PAGE_SIZE));
    pages = bytes / VM_PAGE_SIZE;
    for (p = 0; p < pages; p++) {
        int pde = ARCH_VM_PDE(v);
        int pte = ARCH_VM_PTE(v);
        int lvl2 = ARCH_VM_lvl2(v);
        assert(!(v % VM_PAGE_SIZE));
        assert(pte >= 0 && pte < ARCH_VM_PT_ENTRIES);
        assert(pde >= 0 && pde < ARCH_VM_DIR_ENTRIES);
        /* Page table has to be there. */
        if (!(pt->pt_dir[pde] & ARCH_VM_PDE_PRESENT))
            return EFAULT;
        /* Make sure page directory entry for this page table
         * is marked present and page table entry is available.
         */
        assert((pt->pt_dir[pde] & ARCH_VM_PDE_PRESENT) && pt->pt_pt[pde]);
        if (!(pt->pt_pt[pde][lvl2][pte] & ARCH_VM_PTE_PRESENT)) {
            return EFAULT;
        }
#if defined(__i386__)
        if (write && !(pt->pt_pt[pde][pte] & ARCH_VM_PTE_RW)) {
#elif defined(__arm__)
        if (write && (pt->pt_pt[pde][pte] & ARCH_VM_PTE_RO)) {
#endif
            return EFAULT;
        }

        v += VM_PAGE_SIZE;
    }

    return OK;
}
```

Figure 29 : pt_checkpage function

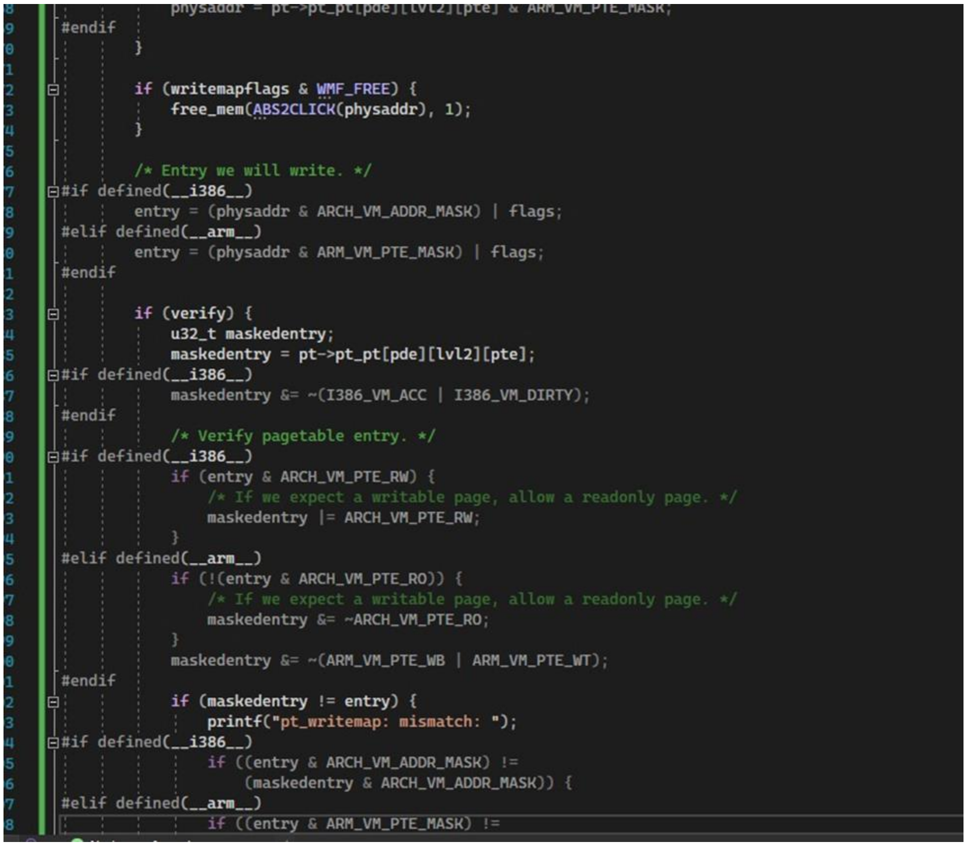
Lvl2 variable to get the 8th unused bit of the page table entry and use it as second level page table



```c
	 */
	assert(physaddr == MAP_NONE || (flags & ARCH_VM_PTE_PRESENT));
	assert(physaddr != MAP_NONE || !flags);

	/* First make sure all the necessary page tables are allocated,
	 * before we start writing in any of them, because it's a pain
	 * to undo our work properly.
	 */
	ret = pt_ptalloc_in_range(pt, v, v + VM_PAGE_SIZE * pages, flags, verify);
	if (ret != OK) {
		printf("VM: writemap: pt_ptalloc_in_range failed\n");
		goto resume_exit;
	}

	/* Now write in them. */
	for (p = 0; p < pages; p++) {
		u32_t entry;
		int pde = ARCH_VM_PDE(v);
		int pte = ARCH_VM_PTE(v);
		int lvl2 = ARCH_VM_lvl2(v);/**********************/

		assert(!(v % VM_PAGE_SIZE));
		assert(pte >= 0 && pte < ARCH_VM_PT_ENTRIES);
		assert(pde >= 0 && pde < ARCH_VM_DIR_ENTRIES);
		/* Page table has to be there. */
		assert(pt->pt_dir[pde] & ARCH_VM_PDE_PRESENT);

		/* We do not expect it to be a bigpage. */
		assert(!(pt->pt_dir[pde] & ARCH_VM_BIGPAGE));

		/* Make sure page directory entry for this page table
		 * is marked present and page table entry is available.
		 */
		assert(pt->pt_pt[pde]);

		if (writemapflags & (WMF_WRITEFLAGSONLY | WMF_FREE)) {
#if defined(__i386__)
			physaddr = pt->pt_pt[pde][lvl2][pte] & ARCH_VM_ADDR_MASK;
#elif defined(__arm__)
			physaddr = pt->pt_pt[pde][lvl2][pte] & ARM_VM_PTE_MASK;
```

Figure 30 : pt_checkpage function cont.

Lvl2 variable to get the 8th unused bit of the page table entry and use it as second level page table



```c
			physaddr = pt->pt_pt[pde][lvl2][pte] & ARM_VM_PTE_MASK;
#endif
		}

		if (writemapflags & WMF_FREE) {
			free_mem(ABS2CLICK(physaddr), 1);
		}

		/* Entry we will write. */
#if defined(__i386__)
		entry = (physaddr & ARCH_VM_ADDR_MASK) | flags;
#elif defined(__arm__)
		entry = (physaddr & ARM_VM_PTE_MASK) | flags;
#endif

		if (verify) {
			u32_t maskedentry;
			maskedentry = pt->pt_pt[pde][lvl2][pte];
#if defined(__i386__)
			maskedentry &= ~(I386_VM_ACC | I386_VM_DIRTY);
#endif
			/* Verify pagetable entry. */
#if defined(__i386__)
			if (entry & ARCH_VM_PTE_RW) {
				/* If we expect a writable page, allow a readonly page. */
				maskedentry |= ARCH_VM_PTE_RW;
			}
#elif defined(__arm__)
			if (!(entry & ARCH_VM_PTE_RO)) {
				/* If we expect a writable page, allow a readonly page. */
				maskedentry &= ~ARCH_VM_PTE_RO;
			}
			maskedentry &= ~(ARM_VM_PTE_WB | ARM_VM_PTE_WT);
#endif
			if (maskedentry != entry) {
				printf("pt_writemap: mismatch: ");
#if defined(__i386__)
				if ((entry & ARCH_VM_ADDR_MASK) !=
					(maskedentry & ARCH_VM_ADDR_MASK)) {
#elif defined(__arm__)
				if ((entry & ARM_VM_PTE_MASK) !=
```

Figure 31 : pt_checkpage function cont.

Access method to the array changed to 3 levels

```c
                    if ((entry & ARM_VM_PTE_MASK) !=
                        (maskedentry & ARM_VM_PTE_MASK)) {
#endif
                        printf("pt_writemap: physaddr mismatch (0x%lx, 0x%lx); ",
                            (long)entry, (long)maskedentry);
                    }
                    else printf("phys ok; ");
                    printf(" flags: found %s; ",
                        ptestr(pt->pt_pt[pde][lvl2][pte]));
                    printf(" masked %s; ",
                        ptestr(maskedentry));
                    printf(" expected %s\n", ptestr(entry));
                    printf("found 0x%x, wanted 0x%x\n",
                        pt->pt_pt[pde][lvl2][pte], entry);
                    ret = EFAULT;
                    goto resume_exit;
                    }
                }
            else {
                /* Write pagetable entry. */
                pt->pt_pt[pde][lvl2][pte] = entry;
            }

            physaddr += VM_PAGE_SIZE;
            v += VM_PAGE_SIZE;
            }

resume_exit:

#ifdef CONFIG_SMP
    if (vminhibit_clear) {
        assert(vmp && vmp->vm_endpoint != NONE && vmp->vm_endpoint != VM_PROC_NR &&
            !(vmp->vm_flags & VMF_EXITING));
        sys_vmctl(vmp->vm_endpoint, VMCTL_VMINHIBIT_CLEAR, 0);
    }
#endif

    return ret;
    }
```

Figure 32 : pt_checkpage function cont.

Access method to the array changed to 3 levels

```c
                | ARM_VM_PTE_CACHED
#endif
                ,
                WMF_OVERWRITE)) != OK) {
                return r;
            }
        }

        return OK;
}

void pt_clearmapcache(void)
{
        /* Make sure kernel will invalidate tlb when using current
         * pagetable (i.e. vm's) to make new mappings before new cr3
         * is loaded.
         */
        if (sys_vmctl(SELF, VMCTL_CLEARMAPCACHE, 0) != OK)
            panic("VMCTL_CLEARMAPCACHE failed");
}

int pt_writable(struct vmproc* vmp, vir_bytes v)
{
        u32_t entry;
        pt_t* pt = &vmp->vm_pt;
        assert(!(v % VM_PAGE_SIZE));
        int pde = ARCH_VM_PDE(v);
        int pte = ARCH_VM_PTE(v);
        int lvl2 = ARCH_VM_lvl2(v);
        assert(pt->pt_dir[pde] & ARCH_VM_PDE_PRESENT);
        assert(pt->pt_pt[pde]);

        entry = pt->pt_pt[pde][lvl2][pte];

#if defined(__i386__)
        return((entry & PTF_WRITE) ? 1 : 0);
#elif defined(__arm__)
        return((entry & ARCH_VM_PTE_RO) ? 0 : 1);
#endif
}
```

Figure 33 : clear map function and justify

Lvl2 variable to get the 8th unused bit of the page table entry and use it as second level page table

```c
*                   pt_map_in_range                        *
*==========================================================================*/
int pt_map_in_range(struct vmproc* src_vmp, struct vmproc* dst_vmp,
    vir_bytes start, vir_bytes end)
{
    /* Transfer all the mappings from the pt of the source process to the pt of
     * the destination process in the range specified.
     */
    int pde, pte,lvl2;
    vir_bytes viraddr;
    pt_t* pt, * dst_pt;
    pt = &src_vmp->vm_pt;
    dst_pt = &dst_vmp->vm_pt;
    end = end ? end : VM_DATATOP;
    assert(start % VM_PAGE_SIZE == 0);
    assert(end % VM_PAGE_SIZE == 0);
    assert( /* ARCH_VM_PDE(start) >= 0 && */ start <= end);
    assert(ARCH_VM_PDE(end) < ARCH_VM_DIR_ENTRIES);
#if LU_DEBUG
    printf("VM: pt_map_in_range: src = %d, dst = %d\n",
        src_vmp->vm_endpoint, dst_vmp->vm_endpoint);
    printf("VM: pt_map_in_range: transferring from 0x%08x (pde %d pte %d) to 0x%08x (pde %d pte %d)\n",
        start, ARCH_VM_PDE(start), ARCH_VM_PTE(start),
        end, ARCH_VM_PDE(end), ARCH_VM_PTE(end));
#endif
    /* Scan all page-table entries in the range. */
    for (viraddr = start; viraddr <= end; viraddr += VM_PAGE_SIZE) {
        pde = ARCH_VM_PDE(viraddr);
        if (!(pt->pt_dir[pde] & ARCH_VM_PDE_PRESENT)) {
            if (viraddr == VM_DATATOP) break;
            continue;   }
        lvl2 = ARCH_VM_lvl2(viraddr);//*******************************************************//
        pte = ARCH_VM_PTE(viraddr);
        if (!(pt->pt_pt[pde][lvl2][pte] & ARCH_VM_PTE_PRESENT)) {
            if (viraddr == VM_DATATOP) break;
            continue;}
        /* Transfer the mapping. */
        dst_pt->pt_pt[pde][lvl2][pte] = pt->pt_pt[pde][lvl2][pte];
        assert(dst_pt->pt_pt[pde]);
        if (viraddr == VM_DATATOP) break;}
    return OK;
```

Figure 34 : pt_map_in_range function

Access method to the array changed to 3 levels

```c
static int pt_ptalloc(pt_t * pt, int pde, u32_t flags)
{
    /* Allocate a page table and write its address into the page directory. */
    int i;
    phys_bytes pt_phys;
    u32_t * p;
    /* Argument must make sense. */
    assert(pde >= 0 && pde < ARCH_VM_DIR_ENTRIES);
    assert(!(flags & ~(PTF_ALLFLAGS)));
    /* We don't expect to overwrite page directory entry, nor
     * storage for the page table.
     */
    assert(!(pt->pt_dir[pde] & ARCH_VM_PDE_PRESENT));
    assert(!pt->pt_pt[pde]);
    /* Get storage for the page table. The allocation call may in fact recursively create
    if (!(p = vm_allocpage(&pt_phys, VMP_PAGETABLE)))
        return ENOMEM;
    if (pt->pt_pt[pde]) {
        vm_freepages((vir_bytes)p, 1);
        assert(pt->pt_pt[pde]);
        return OK;
    }
    pt->pt_pt[pde] = p;
    for (int k = 0; k < 2; k++)
        for (i = 0; i < ARCH_VM_PT_ENTRIES; i++)
            pt->pt_pt[pde][k][i] = 0;
    }   /* Empty entry. */
    /* Make page directory entry.
     * The PDE is always 'present,' 'writable,' and 'user accessible,'
     * relying on the PTE for protection.
     */
#if defined(__i386__)
    pt->pt_dir[pde] = (pt_phys & ARCH_VM_ADDR_MASK) | flags
        | ARCH_VM_PDE_PRESENT | ARCH_VM_PTE_USER | ARCH_VM_PTE_RW;
#elif defined(__arm__)
    pt->pt_dir[pde] = (pt_phys & ARCH_VM_PDE_MASK)
        | ARCH_VM_PDE_PRESENT | ARM_VM_PDE_DOMAIN; //LSC FIXME
#endif
    return OK;
}
```

Figure 35 : pt_map_in_range function cont.

Nested for loop at 520 to access the second level

```c
 *==========================================================================*/
int vm_addrok(void* vir, int writeflag)
{
    pt_t* pt = &vmprocess->vm_pt;
    int pde, pte,lvl2;
    vir_bytes v = (vir_bytes)vir;
    pde = ARCH_VM_PDE(v);
    pte = ARCH_VM_PTE(v);
    lvl2 = ARCH_VM_lvl2(v);
    if (!(pt->pt_dir[pde] & ARCH_VM_PDE_PRESENT)) {
        printf("addr not ok: missing pde %d\n", pde);
        return 0;}
#if defined(__i386__)
    if (writeflag &&
        !(pt->pt_dir[pde] & ARCH_VM_PTE_RW)) {
        printf("addr not ok: pde %d present but pde unwritable\n", pde);
        return 0;}
#elif defined(__arm__)
    if (writeflag &&
        (pt->pt_dir[pde] & ARCH_VM_PTE_RO)) {
        printf("addr not ok: pde %d present but pde unwritable\n", pde);
        return 0;}
#endif
    if (!(pt->pt_pt[pde][lvl2][pte] & ARCH_VM_PTE_PRESENT)) {
        printf("addr not ok: missing pde %d / pte %d\n",
            pde,lvl2, pte);
        return 0;}
#if defined(__i386__)
    if (writeflag &&
        !(pt->pt_pt[pde][lvl2][pte] & ARCH_VM_PTE_RW)) {
        printf("addr not ok: pde %d / pte %d present but unwritable\n",
            pde, lvl2 ,pte);
#elif defined(__arm__)
    if (writeflag &&
        (pt->pt_pt[pde][lvl2][pte] & ARCH_VM_PTE_RO)) {
        printf("addr not ok: pde %d / pte %d present but unwritable\n",
            pde,lvl2,pte);
#endif
        return 0;}
    return 1;}
```

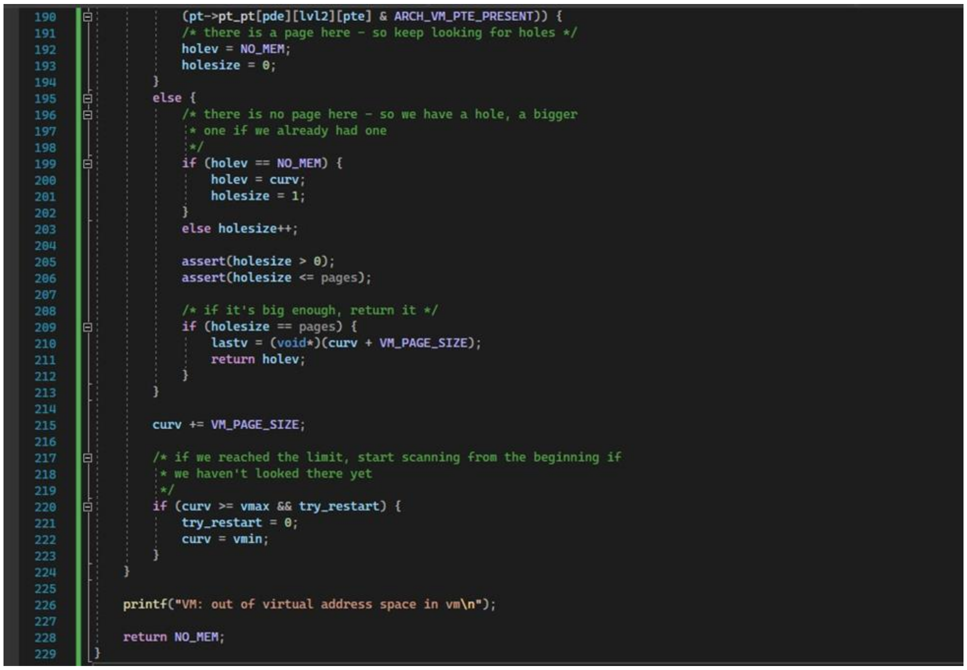
Figure 36 : vm_addrok function

Lvl2 variable to get the 8th unused bit of the page table entry and use it as second level page table

```c
/*==========================================================================*
 *              findhole                          *
 *==========================================================================*/
static u32_t findhole(int pages)
{
    /* Find a space in the virtual address space of VM. */
    u32_t curv;
    int pde = 0, try_restart;
    static void* lastv = 0;
    pt_t* pt = &vmprocess->vm_pt;
    vir_bytes vmin, vmax;
    u32_t holev = NO_MEM;
    int holesize = -1;

    vmin = VM_OWN_MMAPBASE;
    vmax = VM_OWN_MMAPTOP;

    /* Input sanity check. */
    assert(vmin + VM_PAGE_SIZE >= vmin);
    assert(vmax >= vmin + VM_PAGE_SIZE);
    assert((vmin % VM_PAGE_SIZE) == 0);
    assert((vmax % VM_PAGE_SIZE) == 0);
    assert(pages > 0);
    int lvl2 = 0;
    curv = (u32_t)lastv;
    if (curv < vmin || curv >= vmax)
        curv = vmin;

    try_restart = 1;

    /* Start looking for a free page starting at vmin. */
    while (curv < vmax) {
        int pte;
        assert(curv >= vmin);
        assert(curv < vmax);
        pde = ARCH_VM_PDE(curv);
        lvl2 = ARCH_VM_lvl2(curve);//********here we recognized which page table in outer level we will access***********//
        pte = ARCH_VM_PTE(curv);
        if ((pt->pt_dir[pde] & ARCH_VM_PDE_PRESENT) &&
            (pt->pt_pt[pde][lvl2][pte] & ARCH_VM_PTE_PRESENT)) {
            /* there is a page here - so keep looking for holes */
```

Figure 37 : findhole function

At 187 we recognized which page table in outer level we will access



```c
                (pt->pt_pt[pde][lvl2][pte] & ARCH_VM_PTE_PRESENT)) {
                /* there is a page here - so keep looking for holes */
                holev = NO_MEM;
                holesize = 0;
            }
            else {
                /* there is no page here - so we have a hole, a bigger
                 * one if we already had one
                 */
                if (holev == NO_MEM) {
                    holev = curv;
                    holesize = 1;
                }
                else holesize++;

                assert(holesize > 0);
                assert(holesize <= pages);

                /* if it's big enough, return it */
                if (holesize == pages) {
                    lastv = (void*)(curv + VM_PAGE_SIZE);
                    return holev;
                }
            }

            curv += VM_PAGE_SIZE;

            /* if we reached the limit, start scanning from the beginning if
             * we haven't looked there yet
             */
            if (curv >= vmax && try_restart) {
                try_restart = 0;
                curv = vmin;
            }
        }

        printf("VM: out of virtual address space in vm\n");

        return NO_MEM;
}
```

Figure 38 : findhole function cont.

Access method to the array changed to 3 levels

2.2. LRU and FIFO replacement algorithms

2.2.1. LRU algorithm

LRU page replacement algorithm associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period.

It's implemented using a double link list.

```
struct cached_page {
    /*  - The (dev, dev_offset) pair are unique;
     *    the (ino, ino_offset) pair is information and
     *    might be missing. duplicate do not make sense
     *    although it won't bother VM much.
     *  - dev must always be valid, i.e. not NO_DEV
     *  - ino may be unknown, i.e. VMC_NO_INODE
     */
    dev_t dev;          /* which dev is it on */
    u64_t dev_offset;       /* offset within dev */

    ino_t ino;          /* which ino is it about */
    u64_t ino_offset;       /* offset within ino */
    int flags;          /* currently only VMSF_ONCE or 0 */
    struct phys_block* page;    /* page ptr */
    struct cached_page* older;  /* older in lru chain */
    struct cached_page* newer;  /* newer in lru chain */
    struct cached_page* hash_next_dev; /* next in hash chain (bydev) */
    struct cached_page* hash_next_ino; /* next in hash chain (byino) */
};
```

Figure 39 : Cached page struct

Where it has a pointer to older page before it and the newer page after it and a pointer to the page itself

And three functions are used to implement that lru_add,lru_rm and lru_touch

Where lru_add is used to add a node to the linked list

Lru_rm is used to remove a node from the linked list

Lru_touch is used to remove a node then add the same node again

The point to use lru_touch is to reach the goal of the algorithm that when you need to access an already available page (node) its moved to the front of the linked list so the least recently used is always at the back

We now can move to implementation of those functions

```c
static void lru_add(struct cached_page *hb)
{
    if(lru_newest) {
        assert(lru_oldest);
        assert(!lru_newest->newer);
        lru_newest->newer = hb;
    } else {
        assert(!lru_oldest);
        lru_oldest = hb;
    }

    hb->older = lru_newest;
    hb->newer = NULL;
    lru_newest = hb;

    cached_pages++;
}
```

Figure 40 : lru_add function

Where lru_newest and lru_oldest are pointers to the back and front of the linked list and they are defined global to the file so you will see them used in lru_rm as well

Cached_pages is the number of the pages that are cached and its also a global variable

```c
static void lru_rm(struct cached_page *hb)
{
    struct cached_page *newer = hb->newer, *older = hb->older;
    assert(lru_newest);
    assert(lru_oldest);
    if(newer) {
        assert(newer->older == hb);
        newer->older = older;
    }
    if(older) {
        assert(older->newer == hb);
        older->newer = newer;
    }

    if(lru_newest == hb) { assert(!newer); lru_newest = older; }
    if(lru_oldest == hb) { assert(!older); lru_oldest = newer; }

    if(lru_newest) assert(lru_newest->newer == NULL);
    if(lru_oldest) assert(lru_oldest->older == NULL);

    cached_pages--;
}
```

Figure 41 : lru_rm function

Same as lru_add there is no new variables here

```c
void cache_lru_touch(struct cached_page *hb)
{
    lru_rm(hb);
    lru_add(hb);
}
```

Figure 42 : cache_lru_touch function

Just removing the page then adding it to the front of the LinkedList

2.2.2. FIFO algorithm

To understand how we implemented FIFO you should look fist at the previous section where the LRU is Implemented.

When the process first enter it is added to LinkedList so at this point the back of the linked list will be the first in process and we will cancel the lru_touch so even if you accessed the page again while it's in the list its place won't change.

```
#define __FIFO__ 1
struct cached_page {
    /*  - The (dev, dev_offset) pair are unique;
     *    the (ino, ino_offset) pair is information and
     *    might be missing. duplicate do not make sense
     *    although it won't bother VM much.
     *  - dev must always be valid, i.e. not NO_DEV
     *  - ino may be unknown, i.e. VMC_NO_INODE
     */
    dev_t dev;          /* which dev is it on */
    u64_t dev_offset;       /* offset within dev */

    ino_t ino;          /* which ino is it about */
    u64_t ino_offset;       /* offset within ino */
    int flags;          /* currently only VMSF_ONCE or 0 */
    struct phys_block* page;    /* page ptr */
    struct cached_page* older;  /* older in lru chain */
    struct cached_page* newer;  /* newer in lru chain */
    struct cached_page* hash_next_dev; /* next in hash chain (bydev) */
    struct cached_page* hash_next_ino; /* next in hash chain (byino) */
};
```

Figure 43 : FIFO Const active

We added a __FIFO__ constant when its set to other than 0 the FIFO algorithm will work and when its 0 LRU algorithm will work

Here at the function lru_touch we made the preprocessor make the code inactive when __FIFO__ is active

```c
void cache_lru_touch(struct cached_page *hb)
{
#if __FIFO__
    return;
#else
    lru_rm(hb);
    lru_add(hb);
#endif // __FIFO__
}
```

Figure 44 : Cache LRU touch inactive

2.3. Performance parameter Using Replacement Algorithms

- **FIFO:** Number of Page faults is greater while using FIFO algorithm
- **LRU:** Number of Page faults is less while using FIFO algorithm

2.4. Performance parameters Hierarchical paging Algorithms

- **Multilevel:** The page table can consume much less space if there are large regions of unused memory.

Any regions of memory that are not mapped for a region covered by a top-level page number do not need to have a lower-level page table allocated. This avoids the need to allocate the lower-level page table for that region.

- **Single level**: Access time is faster.

3. Requirement 3

3.1. Introduction to the File Systems

Perhaps the most important problem in implementing file storage is keeping track of which blocks on disk belong to which file. Different operating systems use different methods to keep track. In this section we will examine a few of them.

3.1.1. Contiguous Allocation

It is the simplest allocation scheme in operating systems. It stores each file as a common sequence of disk blocks. If we assume that our disk contains 1-KB blocks, A 100-KB file would be allocated in 100 consecutive blocks. In the figure below we have 2 files stored on disk that contains 1-KB blocks, Where file 1 is a 5-KB file and file 2 is a 3-KB file.

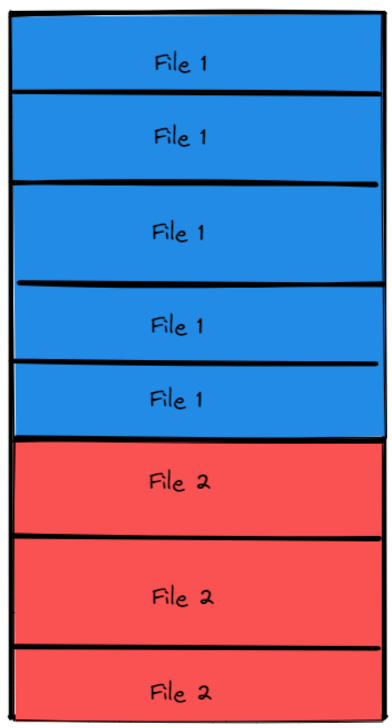


Figure 45 : Storing files using Contiguous Allocation

3.1.1.1. Advantages

- Very simple to implement
- Excellent read performance as the file can be read from disk in a single operation

- Only one ***seek*** is needed which goes to the address of the first block of our file. Then it is read at the full bandwidth of the disk

3.1.1.2. Disadvantages

- As time passes, This could cause ***fragmentation*** where the disk becomes full of files and holes.
- Needs to keep a list of holes to be able to reuse their free space which is not efficient.
- Necessary to know the final size of new files in order to choose a suitable hole of correct size to place it in, Which may not be doable in files that change their sizes frequently.

3.1.2. Linked List Allocation

The second method for storing files in operating system is to keep the file blocks as elements of a linked list as shown in the figure below. Where we start with block 0 which is the initial block which the operating system ***seeks*** to, Then using pointers, The initial block points to the next block. This continues until the next pointer of the file block is ***null*** which indicates that the whole file has been read.

The first word of each block is used as a pointer to the next. The rest of the block used for data.

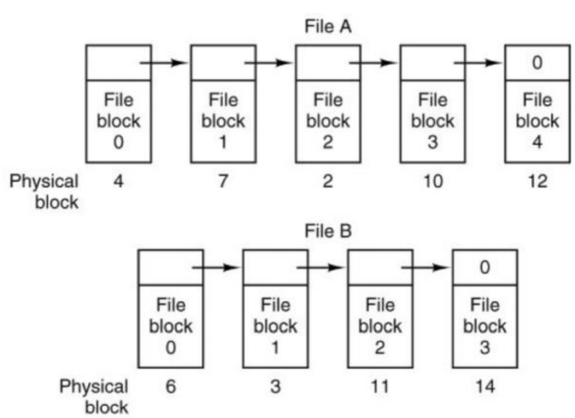


Figure 46 : Storing files using Linked List

Compared to the ***[contiguous allocation](#)*** method, In the linked list every disk block can be used without losing space due to fragmentation. But they are similar in that the operating system will need to ***seek*** only once Where in both linked list and contiguous methods, The operating system will only need to ***seek*** to the disk address of the first block to find the file in the disk.

3.1.2.1. Advantages

1. Only one ***seek*** is needed which goes to the address of the first block of our file.
2. Very easy to increase file size as we can just add another block to our file's linked list. So, the file can grow if memory blocks are available.
3. Less load on directory.

3.1.2.2. Disadvantages

- Random/Direct memory access is not allowed.
- Pointers stored in linked lists incur additional overhead.
- Needs to traverse each block of the file which takes time.
- If a pointer is broken for any reason the file becomes corrupted.

3.1.3. I – Nodes

In UNIX-based operating systems, each file is indexed by its inode. An inode is a special disk block created when a file system is created. The number of files or directories in the file system depends on the number of inodes in the file system.

Each inode table contains the following information :-

1. File Size
2. File type
3. Attributes (Permissions, ownership details, etc..)
4. Access Times (When a file was last read, Or when a file was last written)
5. Several direct blocks (usually 12) containing pointers to the first 12 blocks of the file.
6. A single indirect pointer that points to a certain disk block (The index block). This pointer is used if the file is too large to be indexed entirely by direct blocks.
7. A double indirect pointer that points to a disk block that contains a collection of pointers to the index blocks (The single pointers in the previous step). This pointer is used if the file is too large to be indexed by direct blocks and single indirect pointers
8. A triple index pointer that points to a disk block that contains a collection of pointers (double indirect pointers), Where each pointer of that collection points to a disk block

of another collection of pointers (single indirect pointers), Where each pointer of that collection points to an index block.

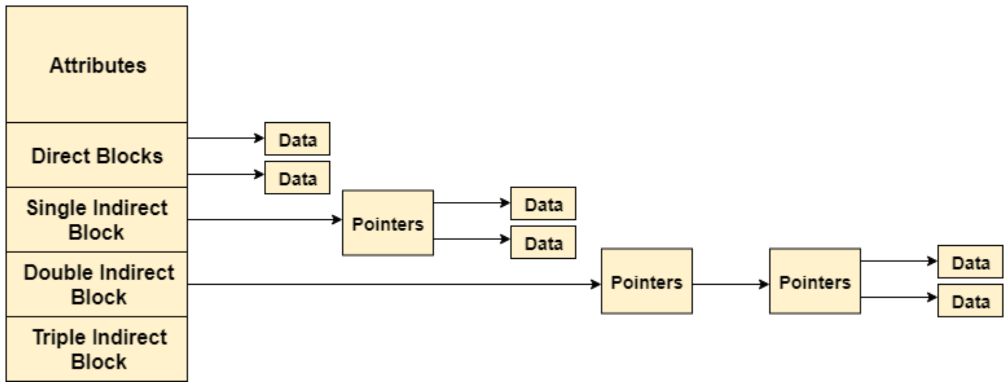


Figure 47 : I-nodes with three levels of indirect blocks

3.2. Bitmaps

Bitmaps are going to be mentioned several times in the next sections, Therefore they will be defined in this section.

The term bitmap means a map of bits. Which means that bitmaps are simply a collection of bits (an array). Each bit in bitmaps corresponds to a disk block. Bits in bitmaps can take 2 values, Either 0 or 1.

3.3. How MINIX 3 manages empty spaces

MINIX 3 keeps track of which I-nodes and zones are free using 2 bitmaps (One bitmap for each of them). MINIX can determine when a space is empty by looking at that space's bit in the bitmap. If that space's bit is = 0, This indicates a free, If that space's bit is = 1, This indicates an allocated block. The following figure showcases several disk blocks on our disk.

If a file is deleted, you can easily calculate which block of the bitmap contains the freed inode bits and use normal caching mechanisms to find it. When the block is found, The bits corresponding to the freed inode are set to 0 (To indicate that it is now free and can be used).

3.4. Modifying disk-space management in MINIX 3

"alloc_bit" is the MINIX function responsible for allocating free disk-space to a certain file. It does so by finding a free bit and the allocating this bit to the desired file. To use extents of 4 blocks we need to make sure that instead of allocating a single bit, the alloc_bit function finds 4 consecutive free bits.

```c
/*===========================================================================*
 *                alloc_bit                                                  *
 *===========================================================================*/
bit_t alloc_bit(sp, map, origin)
struct super_block *sp;     /* the filesystem to allocate from */
int map;            /* IMAP (inode map) or ZMAP (zone map) */
bit_t origin;           /* number of bit to start searching at */
{
/* Allocate a bit from a bit map and return its bit number. */

  block_t start_block;      /* first bit block */
  block_t block;
  bit_t map_bits;       /* how many bits are there in the bit map? */
  short bit_blocks;     /* how many blocks are there in the bit map? */
  unsigned word, bcount;
  struct buf *bp;
  bitchunk_t *wptr, *wlim, k;
  bit_t i, b;

  if (sp->s_rd_only)
    panic("can't allocate bit on read-only filesys");

  if (map == IMAP) {
    start_block = START_BLOCK;
    map_bits = (bit_t) (sp->s_ninodes + 1);
    bit_blocks = sp->s_imap_blocks;
  } else {
    start_block = START_BLOCK + sp->s_imap_blocks;
    map_bits = (bit_t) (sp->s_zones - (sp->s_firstdatazone - 1));
    bit_blocks = sp->s_zmap_blocks;
  }

  /* Figure out where to start the bit search (depends on 'origin'). */
  if (origin >= map_bits) origin = 0;    /* for robustness */

  /* Locate the starting place. */
  block = (block_t) (origin / FS_BITS_PER_BLOCK(sp->s_block_size));
  word = (origin % FS_BITS_PER_BLOCK(sp->s_block_size)) / FS_BITCHUNK_BITS;

  /* Iterate over all blocks plus one, because we start in the middle. */
  bcount = bit_blocks + 1;
  do {
    bp = get_block(sp->s_dev, start_block + block, NORMAL);
    wlim = &b_bitmap(bp)[FS_BITMAP_CHUNKS(sp->s_block_size)];

    /* Iterate over the words in block. */
    for (wptr = &b_bitmap(bp)[word]; wptr < wlim; wptr++) {

        /* Does this word contain a free bit? */
        if (*wptr == (bitchunk_t) ~0) continue;

        /* Find and allocate the free bit. */
        k = (bitchunk_t) conv4(sp->s_native, (int) *wptr);
```

Figure 48 : alloc_bit function (1)

To do so we added two loops to the function. First one is responsible to check for 4 free consecutive bits instead of a single one. The second loop is responsible of allocating the 4 free bits to the file. This is done by setting all bits to 1 by ORing every bit of them with the value 1.

```

            /*** ------------------ ADDED LOOP ------------------ **/
            /* Look for EXTENT_SIZE free consecutive bits*/
            int count = 0;
            for (i = 0; ((k & (1 << i)) != 0 || count < EXTENT_SIZE) && i < 16; i++) {
                if ((k & (1 << i)) == 0) count++;
                if (count < EXTENT_SIZE) continue;
            }

            /* Bit number from the start of the bit map. */
            b = ((bit_t) block * FS_BITS_PER_BLOCK(sp->s_block_size))
                + (wptr - &b_bitmap(bp)[0]) * FS_BITCHUNK_BITS
                + i;

            /* Don't allocate bits beyond the end of the map. */
            if (b >= map_bits) break;

            /* Allocate and return bit number. */

            /*** ------------------ ADDED LOOP ------------------ **/
            /* Allocate the EXTENT_SIZE bits by ORing every bit with the value 1*/
            for (int j = i; j > i - EXTENT_SIZE; j--) {
                k |= 1 << j;
            }

            *wptr = (bitchunk_t) conv4(sp->s_native, (int) k);
            MARKDIRTY(bp);
            put_block(bp, MAP_BLOCK);
            return(b);
        }
        put_block(bp, MAP_BLOCK);
        if (++block >= (unsigned int) bit_blocks) /* last block, wrap around */
            block = 0;
        word = 0;
    } while (--bcount > 0);
    return(NO_BIT);        /* no bit could be allocated */
}

```

Figure 49 : alloc_bit function (2)

Another edit that we found to be important is adding a loop in the free_bit function, which is responsible of freeing unneeded bits by files, to make sure that 4 bits are freed upon function call instead of just freeing a single bit to make sure there is no disk space used without being allocated to a certain file.

```c
/*===========================================================================*
 *              free_bit                         *
 *===========================================================================*/
void free_bit(sp, map, bit_returned)
struct super_block *sp;     /* the filesystem to operate on */
int map;            /* IMAP (inode map) or ZMAP (zone map) */
bit_t bit_returned;     /* number of bit to insert into the map */
{
/* Return a zone or inode by turning off its bitmap bit. */

  unsigned block, word, bit;
  struct buf *bp;
  bitchunk_t k, mask;
  block_t start_block;

  if (sp->s_rd_only)
    panic("can't free bit on read-only filesys");

  if (map == IMAP) {
    start_block = START_BLOCK;
  } else {
    start_block = START_BLOCK + sp->s_imap_blocks;
  }
  block = bit_returned / FS_BITS_PER_BLOCK(sp->s_block_size);
  word = (bit_returned % FS_BITS_PER_BLOCK(sp->s_block_size))
     / FS_BITCHUNK_BITS;

  bit = bit_returned % FS_BITCHUNK_BITS;

  /*** ------------------ ADDED LOOP ------------------ **/
  /* Free EXTENT_SIZE bits by ANDing every bit with 0*/
  for (int i = bit; i > bit - EXTENT_SIZE; i--) {
      mask |= 1 << i;
  }

  bp = get_block(sp->s_dev, start_block + block, NORMAL);

  k = (bitchunk_t) conv4(sp->s_native, (int) b_bitmap(bp)[word]);
  if (!(k & mask)) {
    if (map == IMAP) panic("tried to free unused inode");
    else panic("tried to free unused block: %u", bit_returned);
  }

  k &= ~mask;
  b_bitmap(bp)[word] = (bitchunk_t) conv4(sp->s_native, (int) k);
  MARKDIRTY(bp);

  put_block(bp, MAP_BLOCK);
}
```

Figure 50 : free_bit function

3.5. How MINIX 3 can create, read, and write in files and Directories

The open.c file (available in servers/fs/ open.c according to the manual) contains code for six system calls: creat, open, mknod, mkdir, close and lseek.

In older versions of UNIX, the creat and open calls had different purposes. Attempting to open a file that didn't even exit usually resulted in an error and had to create a new file with the creat system call. But since MINIX 3 is POSIX (**P**ortable **O**perating **S**ystem **I**nterface) compliant, The open system call can now be used to create a new file or truncate an old file, Therefore, The creat system call is considered a subset of the open call and is only needed for compatibility with older programs.

When a file is created or opened in MINIX 3 it involves the following steps :-

1. Find an inode (allocate and initialize it if the file is being created)
2. Find or create a directory entry
3. Set and return the file descriptor for the file

3.5.1. Creating a File in MINIX 3

Both the creat and open calls do two things : Get the name of the file and call common_open which handles tasks common to both calls. Common_open makes sure that free file descriptor and flip table slots are available. We have a bit named O_CREAT, When this bit is set the calling function specifies the creation of a new file. In this section, We will set the O_CREAT bit.

Since the calling function specified the creation of a new file, new_node is called on line 24594 in the block of code below. new_node will then return a pointer to an existing inode if the directory entry already exists. Otherwise, both a new directory entry and an ainode are created. If the inode cannot be created, new_node sets the err_code global variable. Setting err_code does not always mean an error in MINIX. If new_node finds an existing file, The error code returned indicates that the file trying to be created already exists.

3.5.2. Opening a File in MINIX 3

In this section we will be using a set of bits called rwx (**R**ead **W**rite E**x**ecute) bits. These bits help us determine whether a file can be read, written on, Or executed or not.

We will also be mentioning a bit called O_TRUNC, During system calls if O_TRUNC is set, All of the file's data is truncated to length 0, Which means that all of the file's contents will be deleted without deleting the file itself.

At first,The O_CREAT bit will not be set (Because we are not creating a new file in this section). Therefore, the search for inode will be made using another way which is called eat_path function in the path.c file. Before the file system opens the file, The file system should first test the file type, mode, etc.. to see if it can be opened. The file system can see this by a call to forbidden() function which makes a general check on the rwx bits mentioned before. If the file is a normal file and common_open was called with the O_TRUNC bit was set. The file is truncated to length 0 and the forbidden() function is called again to make sure that the file may be written in.

3.5.3. Reading a File in MINIX 3

When a file is opened (By following the sequence in [the previous section](#)), It can be either read or written. Many functions are used for both reading and writing. These functions are located in the read.c and write.c files.

First, do_read calls the common procedure read_write with a flag set to READING (This indicates that we want to read the file), During the read_write procedure some validity checks are done (Such as reading a file that is opened only for writing , etc..). The normal file reading mechanism is done through the loop in the figure below.

```
        /* Split the transfer into chunks that don't span two blocks. */
        while (m_in.nbytes != 0) {

                off = (unsigned int) (position % block_size);/* offset in b
                if (partial_pipe) {  /* pipes only */
                        chunk = MIN(partial_cnt, block_size - off);
                } else
                        chunk = MIN(m_in.nbytes, block_size - off);
                if (chunk < 0) chunk = block_size - off;
[Page 976]
                if (rw_flag == READING) {
                        bytes_left = f_size - position;
                        if (position >= f_size) break;  /*  we are beyond E
                        if (chunk > bytes_left) chunk = (int) bytes_left;
                }

                /* Read or write 'chunk' bytes. */
                r = rw_chunk(rip, position, off, chunk, (unsigned) m_in.nby
                             rw_flag, m_in.buffer, seg, usr, block_size, &c

                if (r != OK) break;     /* EOF reached */
                if (rdwt_err < 0) break;

                /* Update counters and pointers. */
                m_in.buffer += chunk;   /* user buffer address */
                m_in.nbytes -= chunk;   /* bytes yet to be read */
                cum_io += chunk;        /* bytes read so far */
                position += chunk;      /* position within the file */

                if (partial_pipe) {
                        partial_cnt -= chunk;
                        if (partial_cnt <= 0) break;
                }
        }
    }
```

Figure 51 : The mechanism for normal file reading

This loop divides the request into chunks, Where each chunk fits on one disk block only. The chunk begins at the current position and keeps extending until one of these conditions is met :-

1. All of the bytes have been read
2. The EOF (**E**nd **O**f **F**ile) is hit.
3. A block boundary is encountered

After the request has been divided into multiple chunks, The reading of these chunks is done by rw_chunk. rw_chunk involves taking the inode and file positions, converting them to physical disk block numbers, and requesting that the block (or part of it) be transferred to the user space. When control returns, various counters and the pointers are incremented, and the next iteration begins.

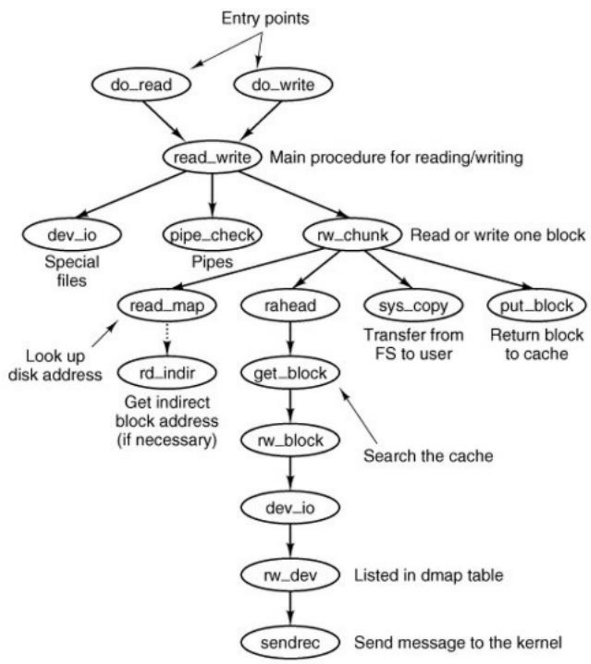


Figure 52 : Procedures involved in reading a file

3.5.4. Writing a File in MINIX 3

As we said before, When a file is opened, It can be either read or written. The code for writing files is available in the write.c file. The do_write function does the same as do_read in the previous section, But with a change in the flag being sent. do_write calls read_write with the WRITING flag. The difference between writing and reading files in MINIX is that in writing we will need to allocate new disk blocks to the new data being written. Write_map is called which is similar to ***read_map*** but instead of searching for physical block numbers in the i-node and indirect blocks, It enters new ones there.

If the zone to be inserted is near the beginning of the opened file, It is simply inserted into the inode.

But writing in a file is not that simple, As we need to deal with many cases that might occur when writing new data in a file. For example, what if a file exceeds the size that can be handled by a single-indirect block?, In that case we will need to use a double-indirect block to fix this issue. Then the single-indirect block will be allocated and its address put into the double-indirect block. But this raises another issue which is that the disk might be full so the single-indirect block cannot be allocated. In this case the double-indirect block is returned to avoid corrupting the bitmap.

3.5.5. Creating a directory in MINIX 3

To create a directory in MINIX, The create system call is called. After the system call is called, A directory is created. It is empty except for dot and dotdot which are automatically placed there by the system.

3.5.6. Opening a directory in MINIX 3

Like files, Directories can be read in MINIX 3. 'Reading' a directory simply lists all the files in that directory. A listing program is used to open the directory and read out all the files' names which are located in the directory. Before being read, Directories must be opened first (Same as files).

To open a directory in MINIX, The Opendir system call is called.

3.5.7. Reading a directory in MINIX 3

To read a directory in MINIX, The Readdir system call is called. Calling this system call will return it the next entry in the currently open directory. Previously, Directories could be read with the normal read system call but this had a major disadvantage which is that the programmers had to be aware of and work with the internal structures of the directory. In contrast, the readdir system call always returns entries in an established form no matter which directory structure is being used.

3.5.8. Writing in a directory in MINIX 3

Technically, The term for writing in directories is called 'Linking'. Linking is a method that allows files to appear in multiple directories. This system call, given an existing file and pathname, creates a link from the existing file to the name specified by the path. This allows the same file to appear in multiple directories. This type of link that increments a counter in the file's inode (To keep track of the number of directory entries containing the file) is sometimes called a hard link.

Linking is done by calling the Link system call.

The opposite of linking can also be done by the Unlink system call. When Unlink is called, A directory entry is removed. If the file to be unlinked only exists in one directory, It is removed from the file system entirely. If it exists in multiple directories, Only the specified pathname is removed. The rest will remain.

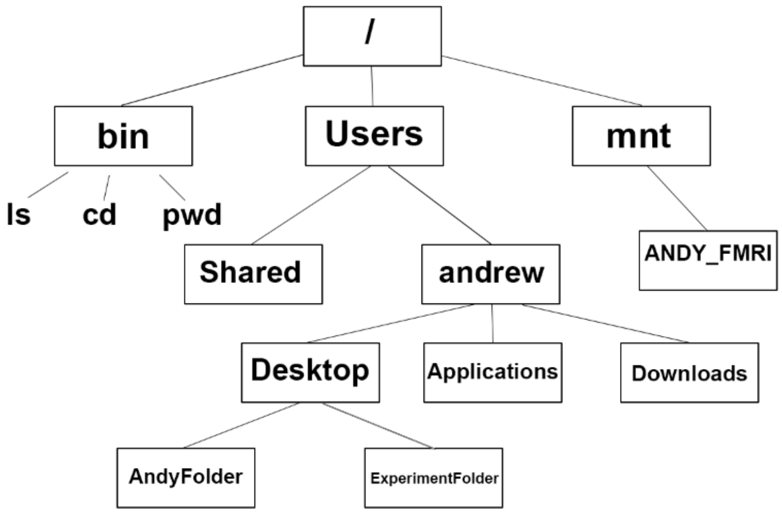


Figure 53 : Example of a MINIX Directory tree

4. Requirement 4 : Internal structure of MINIX

4.1. Introduction

Minix is an open-source Unix like operating system that was created with the intention of serving as a teaching tool. Minix was created by Andrew S. Tanenbaum, and in order to fulfil its function as an educational tool, it is modular and has well-commented source code that is made available. There are around 4000 lines of code in the Minix kernel.

The Linux kernel, the face of open source, was inspired by Minix, which has a long and distinguished history in the academic world. Since Linus Torvalds used a Minix system when he first began working on the Linux kernel, Linux and Minix in those early days share a lot in terms of design and drivers. Minix uses a micro-kernel architecture, but Linux is developed using a monolithic structure. This difference in implementation is an important one.

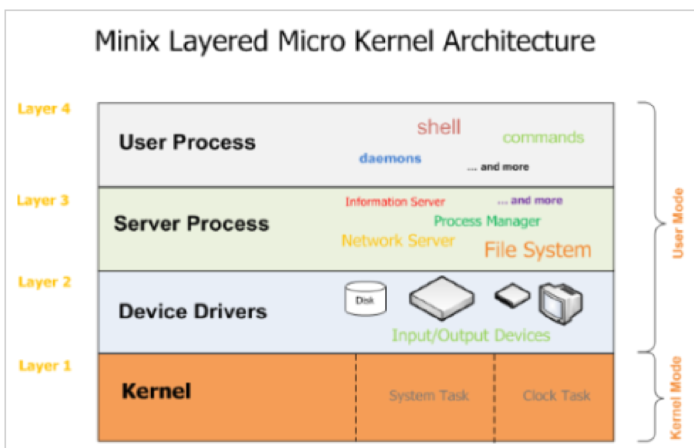


Figure 54 : MINIX Layered Micro Kernel Architecture

As shown in the diagram above, the Minix operating system has a layered, micro-kernel architecture. The core content of the unit discusses five different operating system architectures: client/server, layered, virtual machine, exokernel, and monolithic. The client/server structure and the layered structure are combined in Minix.

The system is divided into a number of layers by the layered system structure, each of which carries out a certain purpose. In such a system, the lower level layers often supply services that the upper level levels rely on. Each of Minix's four layers has an unique and clearly defined role. In a microkernel design, the majority of the operating system's crucial features are implemented as servers that operate independently of the main kernel. This structure makes the operating system flexible since new services may be created and enhanced without significantly altering the kernel Address space management, thread management, inter process communication, and timer management are some of the major functions offered by a Microkernel

4.2. Layer 4

4.2.1. User processes

This makes up the Minix operating system's user land, where user programs are run. These programs rely on the lower level providers' offerings for their service Daemons of all kinds, shells, commands, and any other program the user might want to launch are frequently found in this layer.

The layer 4 processes often gain access to privileged resources through the lower-level processes because they have the fewest access privileges to such resources.

For instance, a user can use the networking server process to run the ping command.

Ping does not make a direct call to the networking server process.

Instead, the file system server process is used.

4.3. Layer 3

4.3.1. Server process

All applications running at layer 4 rely on the services provided by this layer.

Although programs in layer 4 cannot directly access Layer 2 processes, processes in this layer can access the services provided by the device driver layer.

File system, reincarnation server, network server, information server, memory manager, process manager, etc. are a few examples of these services.

The servers often give services to layer 4 processes and applications while consuming services from the lower layers in a layered way.

4.4. Layer 2

4.4.1. Device drivers:

This layer contains Input/Output devices including discs, keyboards, printers, CD drivers, and etc..

4.5. Layer 1

The lowest level services required to keep the system running are provided by frist layer.

These include management of interrupts, traps, scheduling, and communication.

While the majority of this layer is written in C, the lowest portion that deals with interrupts is written in assembly language

It does the following:-

- Providing Services to the next layer
- Scheduling
- Messaging and communication services
- Manage interrupts and Traps
- Saving and restoring registers

5. Requirement 4 : The other operating system internal structures

5.1. Mac OS

The Mac OS is a graphical operating system developed by Apple Inc. The tenth version of the Mac OS is the Mac OS X which was launched in 2001.The structure of the Mac OS X includes multiple layers. The base layer is Darwin which is the Unix core of the system. Next layer is the graphics system which contains Quartz, OpenGL and QuickTime. Then is the application layer which has four components, namely Classic, Carbon, Cocoa and Java. The top layer is Aqua, which is the user interface.

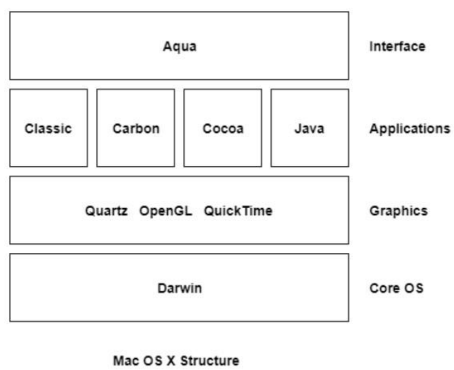


Figure 55 : Mac OS X Structure

5.1.1. Core OS

The Darwin Core is based on the BSD (Berkeley Software Distribution) version of Unix. Mach is the main part of the Darwin core and it performs operations such as memory use, data flow from and to CPU etc. Darwin is also open source i.e. anyone can obtain its source code and make modifications to it. Different versions of Darwin can be used to enhance the Mac OS X. Some of the major features of the Darwin core are protected memory, automatic memory management, preemptive multitasking, advanced virtual memory etc. It also provides I/O services for Mac OS X and supports plug-and-play, hot-swapping and power management.

5.1.2. Graphics Subsystem

Three components make up the Mac OS X graphics subsystem: Quartz, OpenGL, and QuickTime. Quartz controls the 2-D graphics in the graphics subsystem. It offers fonts, interface graphics, picture rendering, etc. System-wide 3-D graphics features including texture mapping, transparency, antialiasing, atmospheric effects, special effects, etc. are supported by OpenGL. Windows and Unix systems both use it. Digital media like as streaming audio and video, digital video, and more are all supported by QuickTime. Additionally, it makes creative software like iTunes and iMovie possible.

5.1.3. Application Subsystem

Mac OS X's application subsystem offers the traditional environment needed to run classic apps. The three supported application development environments are Carbon, Cocoa, and Java. The classic environment ensures that programs created for earlier operating systems can function properly. Existing programs are transferred to carbon application program interfaces using the carbon environment. The application called carbonised when this happens. The object-oriented application development environment is provided by the cocoa environment. The Mac OS X Structure's advantages are most frequently used by cocoa apps. The Java environment can be used to run Java programs and applets.

5.1.4. User Interface

Mac OS X's user interface is called Aqua. It offers both excellent visual features and the means to modify the user interface to suit the needs of the user. Along with making considerable use of color and texture, Aqua also features incredibly intricate iconography. It is efficient to use and pleasing to look at.

5.2. Linux

In 1991, the Linux history started with the starting of a particular project by the Finland student Linus Torvalds for creating a new free OS kernel. The final Linux Kernel was remarked by continuous development throughout the history since then.

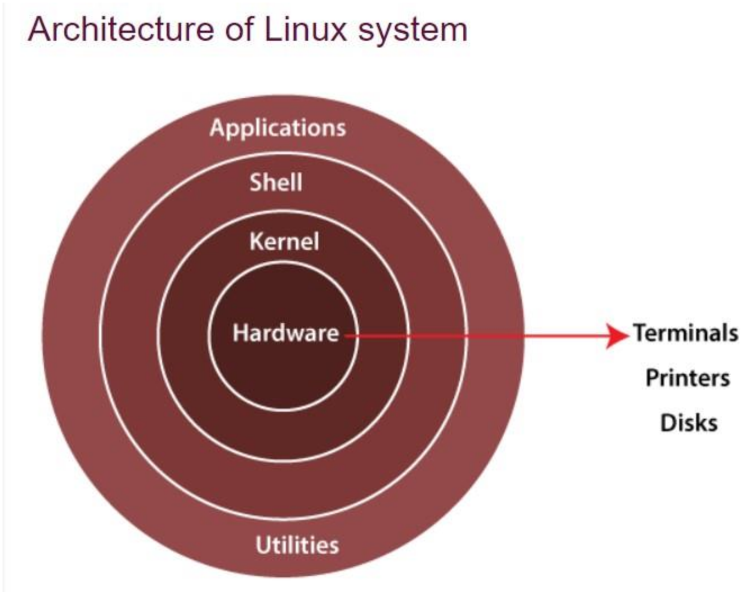


Figure 56 : Architecture of Linux System

The Linux operating system's architecture mainly contains some of the components: **the Kernel, System Libraries, Hardware layer, System,** and **Shell utility**.

5.2.1. Kernel

One of the core parts of an operating system is the kernel. It is in responsible of all of the Linux OS's major operations. This operating system works directly with the hardware resources and includes many types of modules. The kernel enables the necessary abstraction to protect the system from the specifics of low-level hardware or application programs. The following list includes some of the major kernel types:

1. Exo kernels
2. Hybrid kernels
3. Monolithic Kernel
4. Micro kernels

5.2.2. System Libraries:

these libraries can be specified as some special functions. These are applied for implementing the operating system's functionality and don't need code access rights of the modules of kernel.

5.2.3. System utility programs :

It is responsible for doing specialized level and individual activities.

5.2.4. Hardware layer :

Linux operating system contains a hardware layer that consists of several peripheral devices like CPU, HDD, and RAM.

5.2.5. Shell:

It serves as a user kernel interface. It can pay for kernel's services. It can accept commands from the user and performs kernel operations. There are various kinds of Operating systems that support the shell. These operating systems can be divided into two groups: command line shells and graphical shells. While command line shells support the command line interface, graphical line shells provide the graphical user interface. As a result, these two shells carry out operations. However, compared to command line interface shells, graphical user interface shells operate more slowly.

There are a few types of these shells which are categorized as follows:

1. Korn shell
2. Bourne shell
3. C shell
4. POSIX shell

6. Appendix :-

6.1. Ready Queue :-

It is the queue that keeps a set of all processes that reside in the main memory and ready to be executed.

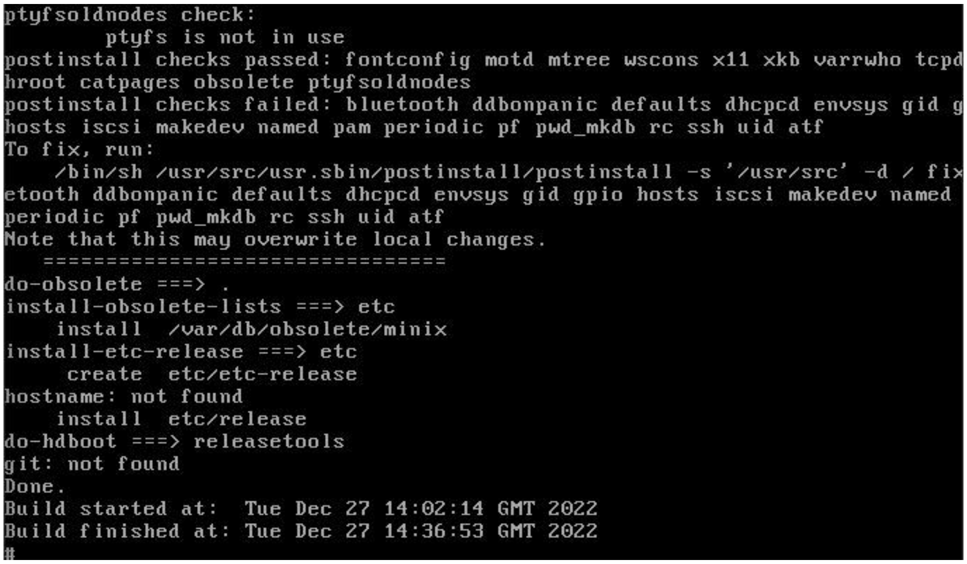
6.2. Seek :-

A seek operation allows an application program to change the value of the file pointer so that subsequent reads/writes are performed from a new position within the file. The new value of the file pointer is determined by adding the offset to the current value.

6.3. Read_map (Used in file reading) :-

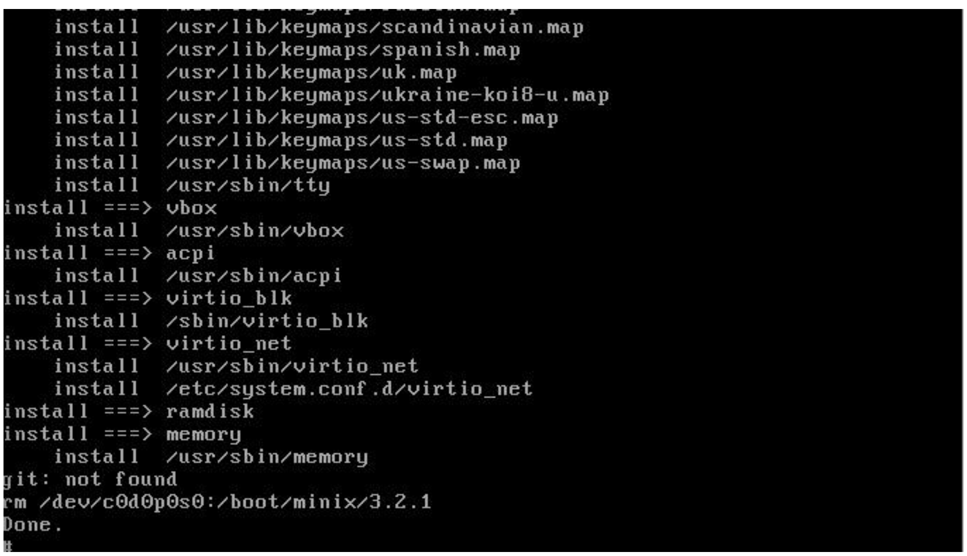
Looks at inode and converts logical file positions to physical block numbers. For blocks close enough to the beginning of the file to fit in one of the first seven zones (those in inodes), a simple calculation can determine which zone is needed and which block is needed next. Blocks further down in the file may need to read one or more indirect blocks.

7. Testing the OS :-



```
ptyfsoldnodes check:
        ptyfs is not in use
postinstall checks passed: fontconfig motd mtree wscons x11 xkb varrwho tcpd
hroot catpages obsolete ptyfsoldnodes
postinstall checks failed: bluetooth ddbonpanic defaults dhcpcd envsys gid g
hosts iscsi makedev named pam periodic pf pwd_mkdb rc ssh uid atf
To fix, run:
    /bin/sh /usr/src/usr.sbin/postinstall/postinstall -s '/usr/src' -d / fix
etooth ddbonpanic defaults dhcpcd envsys gid gpio hosts iscsi makedev named
periodic pf pwd_mkdb rc ssh uid atf
Note that this may overwrite local changes.
   ================================
do-obsolete ===> .
install-obsolete-lists ===> etc
    install  /var/db/obsolete/minix
install-etc-release ===> etc
     create  etc/etc-release
hostname: not found
    install  etc/release
do-hdboot ===> releasetools
git: not found
Done.
Build started at:  Tue Dec 27 14:02:14 GMT 2022
Build finished at: Tue Dec 27 14:36:53 GMT 2022
#
```

Figure 57 : Build test



```
    install  /usr/lib/keymaps/scandinavian.map
    install  /usr/lib/keymaps/spanish.map
    install  /usr/lib/keymaps/uk.map
    install  /usr/lib/keymaps/ukraine-koi8-u.map
    install  /usr/lib/keymaps/us-std-esc.map
    install  /usr/lib/keymaps/us-std.map
    install  /usr/lib/keymaps/us-swap.map
    install  /usr/sbin/tty
install ===> vbox
    install  /usr/sbin/vbox
install ===> acpi
    install  /usr/sbin/acpi
install ===> virtio_blk
    install  /sbin/virtio_blk
install ===> virtio_net
    install  /usr/sbin/virtio_net
    install  /etc/system.conf.d/virtio_net
install ===> ramdisk
install ===> memory
    install  /usr/sbin/memory
git: not found
rm /dev/c0d0p0s0:/boot/minix/3.2.1
Done.
#
```

Figure 58 : Installation test