

Operating System

CSE335



Made by:

|  |  |
| --- | --- |
| Abdel-rahman megahed | 18P7423 |
| Maryam Ahmed Nouh | 18P2824 |
| Yomna Hussein | 18P6546 |
| Youssef Mohamed Mansi | 18P5848 |
| Seif Ahmed Elewa | 18P5662 |
| Sherif Naiem | 18P5794 |

[I. Requirement 1 - CPU Schedulers 3](#_Toc61038563)

[A. Round Robin 3](#_Toc61038564)

[1. Explanation 3](#_Toc61038565)

[2. Code Implementation 3](#_Toc61038566)

[3. Test Cases and Results 5](#_Toc61038567)

[4. Test Cases and Results 6](#_Toc61038568)

[B. Shortest Job First 7](#_Toc61038569)

[1. Explanation 7](#_Toc61038570)

[2. Code Implementation 7](#_Toc61038571)

[3. Test Cases and Results 9](#_Toc61038572)

[C. Priority Based 9](#_Toc61038573)

[1. Explanation 9](#_Toc61038574)

[2. Code Implementation 10](#_Toc61038575)

[3. Test Cases and Results 10](#_Toc61038576)

[D. Multi-level Feedback 11](#_Toc61038577)

[1. Explanation 11](#_Toc61038578)

[2. Code Implementation 12](#_Toc61038579)

[3. Test Cases and Results 15](#_Toc61038580)

[II. Requirement 2 - Paging 17](#_Toc61038581)

[A. FIFO Page Replacement 17](#_Toc61038582)

[1. Explanation 17](#_Toc61038583)

[2. Code Implementation 17](#_Toc61038584)

[3. Conclusion 18](#_Toc61038585)

[B. LRU Page Replacement 18](#_Toc61038586)

[1. Explanation 18](#_Toc61038587)

[2. Code Implementation 19](#_Toc61038588)

[3. Conclusion 19](#_Toc61038589)

[C. Hierarchical Paging Algorithm 19](#_Toc61038590)

[1. Explanation 19](#_Toc61038591)

[2. Code Implementation 21](#_Toc61038592)

[3. Code 21](#_Toc61038593)

[III. Requirement 3 - Memory Management 22](#_Toc61038594)

[A. Memory Management in MINIX 3 22](#_Toc61038595)

[B. User-Defined Disk Allocation 23](#_Toc61038596)

[1. Contiguous Allocation 23](#_Toc61038597)

[2. Linked Allocation 24](#_Toc61038598)

[3. Indexed Allocation 26](#_Toc61038599)

[C. Compare Performances 27](#_Toc61038600)

[D. Code Implementation 28](#_Toc61038601)

# Requirement 1 - CPU Schedulers

## Round Robin

### Explanation

Round Robin scheduling implements the concept of sharing equally one thing amongst all parties. Here, used in multitasking, this algorithm tackles dividing the processor between all processes in the ready state for a limited time. In Round Robin scheduling, each task in the ready queue gets executed for a time period known as time quantum or time slice, then the processor is taken away involuntarily, characterizing this algorithm as preemptive. The Round Robin algorithm works as follows, it takes the first process in the ready queue and assigns a CPU to it. Once the time slice is over, the process involuntarily yields control of the processor. Here, the cycle restarts with the following process positioned in the ready queue. This is done till each process in the ready queue gets its total burst time with the processor and finishes its execution. The Round robin scheduling method is independent from the burst time; thus, it is implemented without restrictions.

This algorithm, like any other, has its advantages and disadvantages, such as eliminating the starvation problem, since all processes get an equal time share when assigned a processor. One of this method’s most famous fortes is that knowing the total number of tasks in the ready queue, one can get a prediction on the worst-case response time. It also happens to be an exquisite path to multitasking, hence producing the best average response time, an important factor for the overall performance. Round Robin scheduling is known for its fairness, dealing with all tasks equally, no priority taken into consideration, assuring fair allocation of the processor to all tasks. Yet, this might be a problem if a priority was to be set for a process. This algorithm allows the operating system to save each process’ state, through context switch, but with all the swapping between tasks, this makes the algorithm spend more time switching rather than executing the processes. Unfortunately, if the time quantum is too small, the performance will be much worse, as there will be a much higher context switch overhead; meaning that the performance of the Round Robin scheduling is dependent on the slicing time. However, perfecting the value of the time quantum is an underestimated job, that’s very hard to reach.

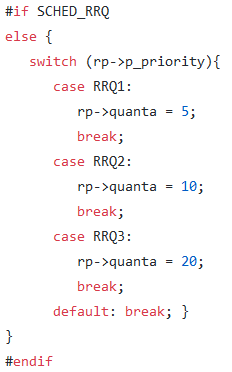
### Code Implementation

Note that user processes will exist in queues 16-18.

* Demotion from Queue 1: Complete 5 quanta in this queue.
* Demotion from Queue 2: Complete 10 quanta in this queue.
* Promotion to Queue 1: Complete 20 quanta in this queue.

Running strategy: Run all processes in Queue 1, then a process from Queue 2. Once all processes in Queue 2 have run, run a process from Queue 3.

Changes have been made to functions “sched()” and “pick\_proc” in “proc.c” that can be found through the path “/usr/src/minix/kernel/proc.c”. The function “sched()” checks if the process's queue has only the sentinel, then the function enqueues to head, otherwise, it enqueues to tail. Moreover, another condition is checked, which is “!time\_left”, in this case, the quanta is increased by one, then the function evaluates for promotion/demotion, else, it refreshes quanta.



The function “pick\_proc()” is used to instantiate a sentinel process in each queue to keep track of whether we have completed the queue's contents or not.

The following pseudocode explains the changes that should be done to the file.

start = 1;

for(int i = RRQ1; i <= RRQ3; i++)

If head[i] is sentinel... Requeue it.

Start = ++i;

break;

for(int i = start; i <= RRQ3; i++)

if(head[i] is a sentinel)

Requeue it.

else

Return head[i].

if global variable repeat == 1

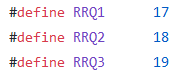
return (repeat = 0);

else

repeat = 1;

return pick\_proc();

Changes have been made to the header file as well “proc.h”. We need to define an extra user queue and trick the original process scheduler into keeping system processes out of the user queue. This is done because there is no harm in having them there. We defined three extra user queues so we can use the same “proc.c” for the other scheduling algorithm. This lets us speed up compile times by only recompiling the kernel instead of all the code that was created in the past. This can be found through the path “/usr/src/minix/kernel/proc.h”.



### Test Cases and Results

User processes will exist in queues 16-18.

Demotion from Queue 1: Complete 5 quanta in this queue.

Demotion from Queue 2: Complete 10 quanta in this queue.

Promotion to Queue 1: Complete 20 quanta in this queue.

Running strategy: Run all processes in Queue 1, then a process from Queue 2. Once all processes in Queue 2 have run, run a process from Queue 3.

Changes have been made to functions sched() and pick\_proc in proc.c

Path: /usr/src/minix/kernel/proc.c

sched(): f the the process's queue has only the sentinel, enqueue to head. Else, enqueue to tail. If !time\_left, ++quanta Evaluate for promotion/demotion else, refresh quanta

A picture containing graphical user interface

Description automatically generated

pick\_proc(): We'll have to instantiate a sentinel process in each queue to keep track of whether or not we've completed the queue's contents.

start = 1; for(int i = RRQ1; i <= RRQ3; i++) If head[i] is sentinel... Requeue it. Start = ++i; break;

for(int i = start; i <= RRQ3; i++) if(head[i] is a sentinel) Requeue it. else Return head[i].

if global variable repeat == 1 return (repeat = 0); else repeat = 1; return pick\_proc();

Changes have been made to the header file as well proc.h we need to define an extra user queue and trick the original process scheduler into keeping system processes out of the user queue. Because there is no harm in having them there, I defined three extra user queues so we can use the same proc.c for the other scheduling algorithm. This lets us speed up compile times by only recompiling the kernel instead of the whole world. Path: /usr/src/minix/kernel/proc.h

A picture containing graphical user interface

Description automatically generated

### Test Cases and Results

Given the following table of processes with their corresponding arrival time, burst time, and priority, the average waiting time and turnaround time for the processes was found.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Process | Arrival Time | Burst Time | Priority | Turnaround time | Wait time |
| P1 | 0 | 10 | 5 | 28 – 0 = 28 | 28 – 10 = 18 |
| P2 | 1 | 5 | 1 | 19 – 1 = 18 | 18 – 5 = 13 |
| P3 | 3 | 2 | 3 | 10 – 3 = 7 | 7 – 2 = 5 |
| P4 | 3 | 3 | 4 | 16 – 3 = 13 | 13 – 3 = 10 |
| P5 | 5 | 4 | 2 | 23 – 5 = 18 | 18 – 4 = 14 |
| P6 | 7 | 4 | 2 | 24 – 7 = 17 | 17 – 4 = 13 |

The average turnaround time is 16.83, while the average wait time is 12.17.

A screenshot of a computer

Description automatically generated with medium confidence

## Shortest Job First

### Explanation

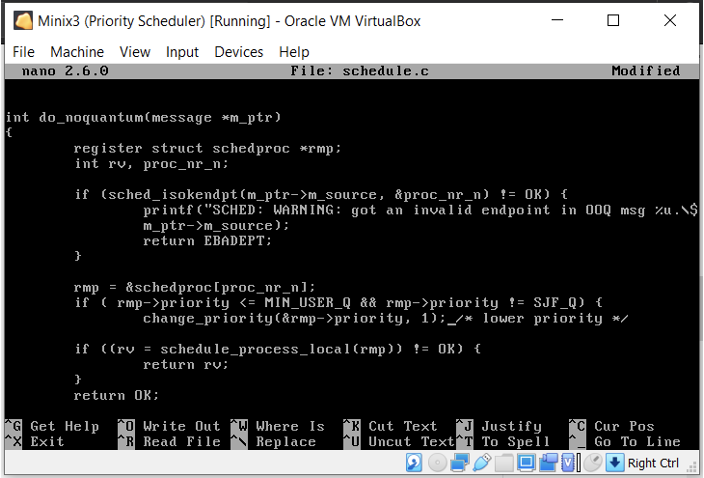
The shortest job first (SJF) is a scheduling algorithm that tries to increase the throughput of the processes inside the CPU. This is achieved by giving processes with short burst times the priority of execution over those with higher burst times. As a result, a shorter turnaround time is obtained as the short processes do not wait for long inside the CPU. This is useful in batch-type processing, where process waiting is not critical.

Using the SJF algorithm comes with a few drawbacks. Firstly, knowing the exact burst time of the process is unachievable; therefore, an equation is used to determine the next estimated burst time interval.

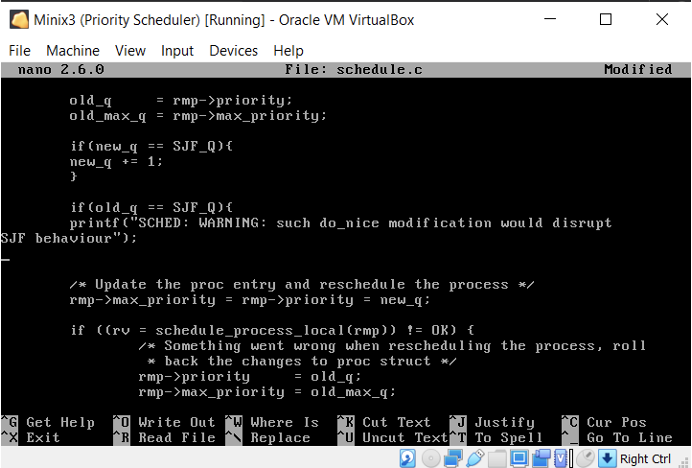
Also, SJF could lead to process starvation. This is when processes with large burst times remain waiting inside the CPU without being executed. This is caused by the constant addition of processes with a short burst time into the CPU. These processes will have a higher priority of execution than the longer processes that were already inside the CPU. This problem can be solved by adding priority aging to the processes where each process that spends a certain amount of time in the CPU gets a priority over newer processes.

### Code Implementation

To be able to edit the CPU scheduler to make it follow the shortest job first algorithm, the path in minix3.4 was found to be “/usr/src/minix/servers/sched/schedule.c”. The change made was in the function under the name “do\_noquantum()”, where a function was added, “change\_priority”, to modify lowering priority in SJF.



Afterwards, changes should also be made to the function “do\_nice()” by adding a “new\_q” if statement to compare the time quantum to the SJF quantum, while also having a comparison of “old\_q” to “SJF\_q” to issue warning that this disrupts the behavior.



Text

Description automatically generatedMoreover, a function was added to implement the shortest job first algorithm , which was named “do\_setsjf()”.

### Test Cases and Results

Given the following table of processes with their corresponding arrival time, burst time, and priority, the average waiting time and turnaround time for the processes was found.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Process | Arrival Time | Burst Time | Priority | Turnaround time | Wait time |
| P1 | 0 | 10 | 5 | 10 – 0 = 10 | 10 – 10 = 0 |
| P2 | 1 | 5 | 1 | 28 – 1 = 27 | 27 – 5 = 22 |
| P3 | 3 | 2 | 3 | 12 – 3 = 9 | 9 – 2 = 7 |
| P4 | 3 | 3 | 4 | 15 – 3 = 12 | 12 – 3 = 9 |
| P5 | 5 | 4 | 2 | 19 – 5 = 14 | 14 – 4 = 10 |
| P6 | 7 | 4 | 2 | 23 – 7 = 16 | 16 – 4 = 12 |

The average turnaround time is 14.67, while the average wait time is 10.

Text

Description automatically generated with medium confidence

## Priority Based

### Explanation

Priority based scheduling is a method of scheduling based on priority. The tasks with the highest priority are selected first.

Characteristics:

* Priority based.
* Used to perform batch processes.
* If the processes have the same priority, it selects based on a first come first serve basis.

1. Lower number indicates a higher priority.

Advantages:

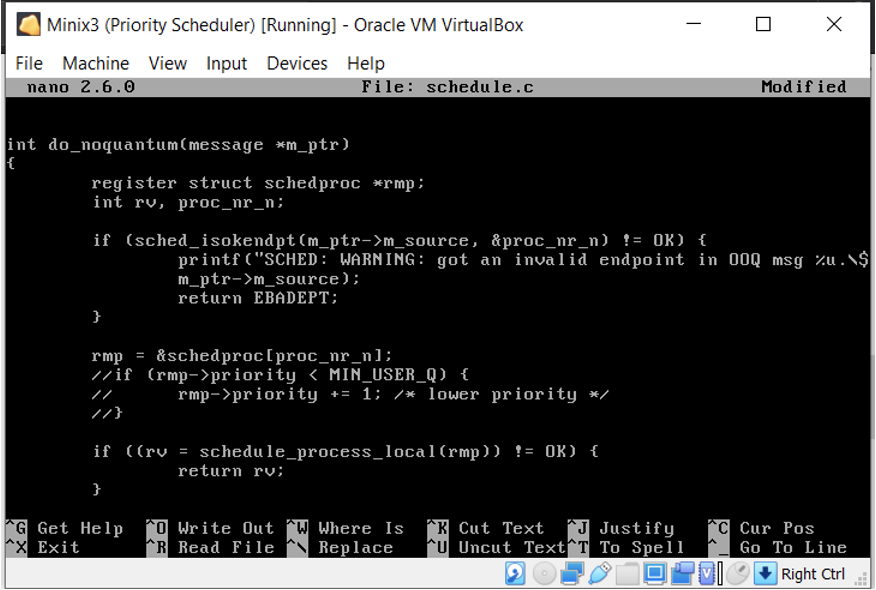
* Easy to use.
* Good handling mechanism as the tasks with high priority are executed first.
* Suitable for applications with differing time and resources.

Disadvantages:

1. If the program crashes all low priority tasks are lost.
2. Starvation may occur to low priority processes.
3. Ready processes must wait for CPU.
4. If a new higher priority process keeps on coming in the ready que1ue, then the process which is in the waiting state may need to wait for a long duration of time.

### Code Implementation

Here, not much was added. A single change to the file “schedule.c” was made in the function “do\_noquantum()” by disabling the feedback.



### Test Cases and Results

Given the following table of processes with their corresponding arrival time, burst time, and priority, the average waiting time and turnaround time for the processes was found.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Process | Arrival Time | Burst Time | Priority | Turnaround time | Wait time |
| P1 | 0 | 10 | 5 | 28 – 0 = 28 | 28 – 10 = 8 |
| P2 | 1 | 5 | 1 | 6 – 1 = 5 | 5 – 5 = 0 |
| P3 | 3 | 2 | 3 | 16 – 3 = 13 | 13 – 2 = 11 |
| P4 | 3 | 3 | 4 | 19 – 3 = 16 | 16 – 3 = 13 |
| P5 | 5 | 4 | 2 | 10 – 5 = 5 | 5 – 4 = 1 |
| P6 | 7 | 4 | 2 | 14 – 7 = 7 | 7 – 4 = 3 |

The average turnaround time is 12.33, while the average wait time is 6.

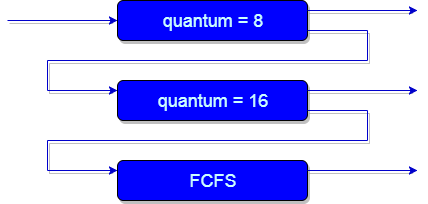
Text

Description automatically generated

## Multi-level Feedback

### Explanation

In a multilevel queue-scheduling algorithm, processes are permanently assigned to a queue on the entrance to the system. Processes do not travel among queues. Multilevel feedback queue scheduling, however, enables a process to migrate among queues. The purpose is to depart processes with various CPU-burst features. If a process utilizes too much CPU time, it will be moved to a lower-priority queue. Likewise, a process that remains too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging stops starvation.



In general, a multilevel feedback queue scheduler is defined by the following parameters:

* The number of queues.
* The scheduling algorithm for every queue.
* The technique utilized to decide when to promote a process to a higher-priority queue.
* The technique utilized to decide when to demote a process to a lower-priority queue.
* The technique utilized to decide which queue a process will access when that process requires service.

The representation of a multilevel feedback queue scheduler presents it as the most prevailing CPU-scheduling algorithm. It can be configured to resemble a particular system under the scheme. Unluckily, it also needs some means of choosing values for all the parameters to determine the most suitable scheduler. Although a multilevel feedback queue is the **most prevailing scheme**, it is also the **most complex**.

#### Advantages

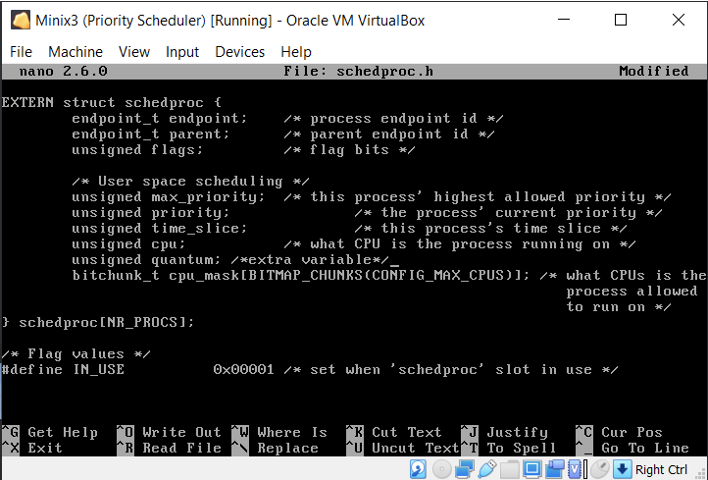
* Low scheduling overhead.
* Allows aging, thus no starvation.

#### Disadvantages

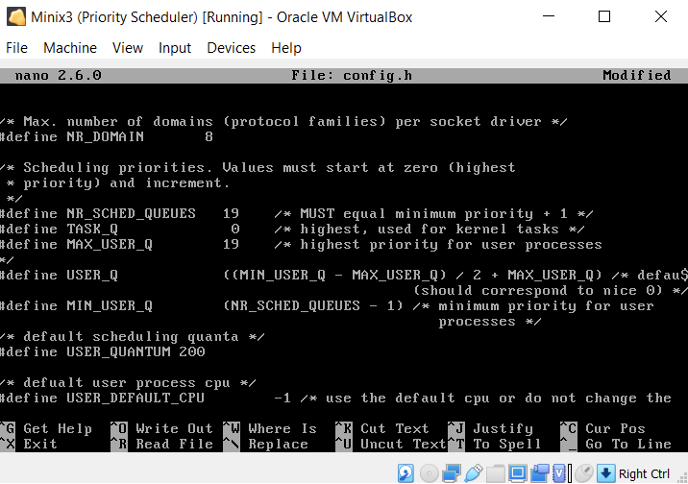
* It is not flexible.
* It also needs some means of choosing values for all the parameters to determine the most suitable scheduler, thus it is also the most complex.

### Code Implementation

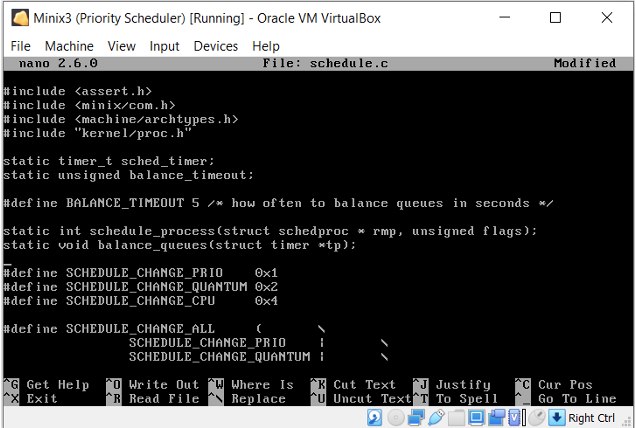
To implement the multilevel feedback queue, changes were made to the existing file named “scheduler.c” and “scheduler.h”, while also modifying in the file “config.h”. In addition, changed were done to the file “schedproc.h”, to be able to add an unsigned quantum. The path to these changed is “/usr/src/servers/sched/config.h”.



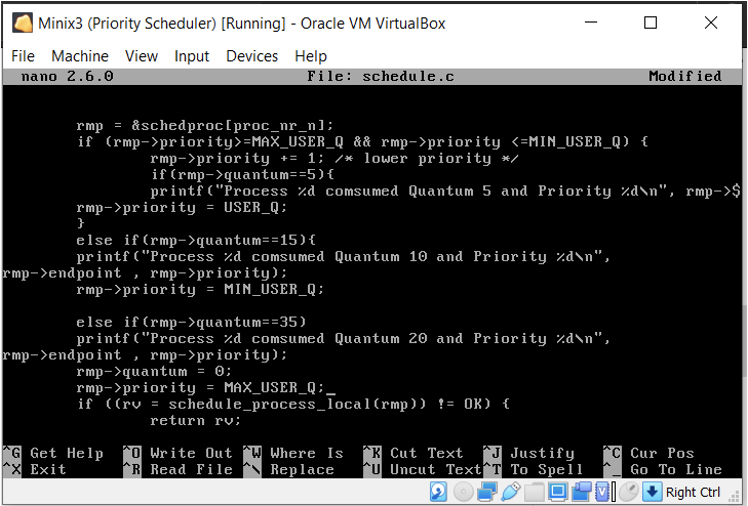
Changes made in the file “config.h” allowed us to add 3 queues for different levels of MFQ. This made the number of original queues range from sixteen to nineteen, NR queues and user queues, respectively. The path to this file is “/usr/src/minix/include/minix/config.h”.



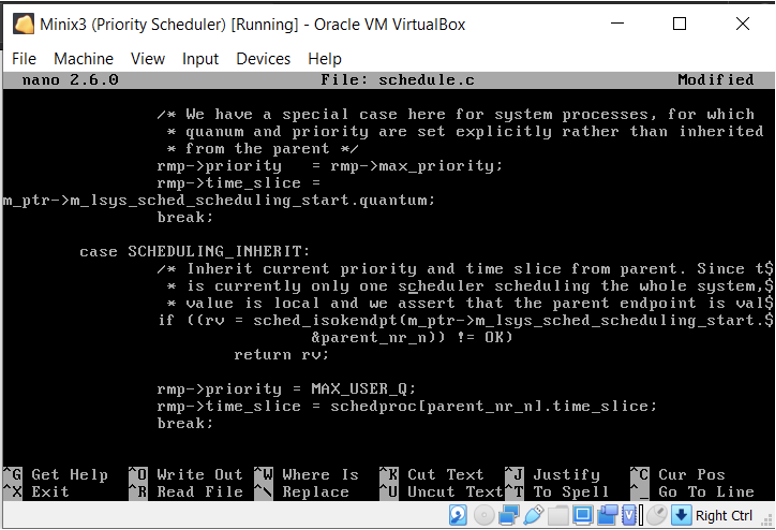
Meanwhile, the changes in the scheduler to be able to implement the MFQ was, found through the path “/usr/src/minix/servers/sched/schedule.c”, adding a static timer for the scheduler.



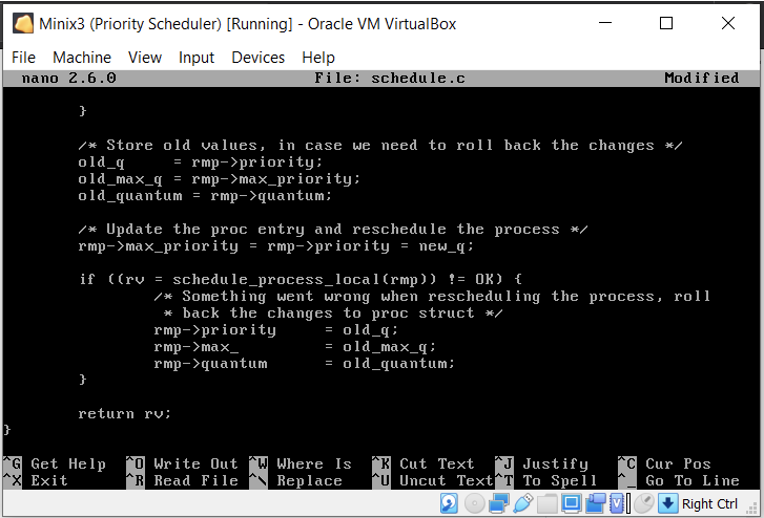
Nevertheless, more changes were made to the function “do\_noquantum()”, mainly to increase the priority of processes between queue 16 and 18, by one.



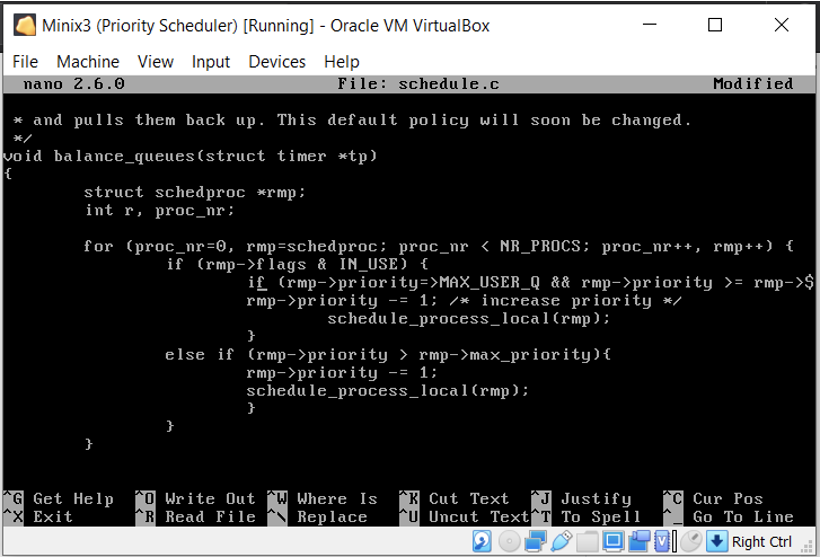
Also, more changes were done to the function “do\_start\_scheduling()” to include initialization of “priority”, “max\_priority,quantum”, and “time\_slice”, changing the values when needed.



Nevertheless, modifications were made for the function “do\_nice()”, as stated before. Those were that in case of no assignment of new priorities, to the processes located in the queues ranging from sixteen to eighteen, then, previous values of priorities are held. However, we keep in mind that an erro might occur, thus, the function should roll back.



Finally, one last modification was made to the function “balance\_queue()” to add one to the priority of a specific process.



### Test Cases and Results

Given the following table of processes with their corresponding arrival time, burst time, and priority, the average waiting time and turnaround time for the processes was found.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Process | Arrival Time | Burst Time | Priority | Turnaround time | Wait time |
| P1 | 0 | 10 | 5 | 8 – 0 + 2 = 10 | 10 – 10 = 0 |
| P2 | 1 | 5 | 1 | 15 – 1 = 14 | 14 – 5 = 9 |
| P3 | 3 | 2 | 3 | 17 – 3 = 14 | 14 – 2 = 12 |
| P4 | 3 | 3 | 4 | 20 – 3 = 17 | 17 – 3 = 14 |
| P5 | 5 | 4 | 2 | 24 – 5 = 19 | 19 – 4 = 15 |
| P6 | 7 | 4 | 2 | 28 – 7 = 21 | 21 – 4 = 17 |

The average turnaround time is 15.83, while the average wait time is 11.17.

Text

Description automatically generated

# Requirement 2 - Paging

In operating systems that use paging for memory management, page replacement algorithm is needed to decide which page needed to be replaced when new page comes in. Whenever a new page is referred and not present in memory, page fault occurs, and Operating System replaces one of the existing pages with newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce the number of page faults.

## FIFO Page Replacement

### Explanation

FIFO stands for “First In, First Out”. It is the simplest page replacement algorithm. The operating system saves all the pages of the memory in a queue, oldest being at the front. When replacement needs to happen, a page is removed from the front of the queue.

Advantages:

* Simplicity
* Easy to understand and implement
* Uses queue data structure

Disadvantages:

* When the number of pages is large, it performs poorly.
* When we increase the number of frames or capacity to store pages in the queue, it should give us a smaller number of page faults. Sometimes FIFO may behave abnormally, and it may increase the number of page faults, called Belady’s Anomaly.

### Code Implementation

To start implementing the FIFO Page Replacement algorithm on MINIX 3, the file responsible for page replacement should be located inside the directory “/usr/src/minix/servers/vm”. The page responsible with dealing with pagefaults is found under the name “pagefaults.c”. Moving on, the following steps should also be done:

1. Find the function(s) responsible for page replacement in this operating system.
2. Replace it with the algorithm to be implemented.

### Conclusion

Text

Description automatically generated

Graphical user interface, text

Description automatically generated

## LRU Page Replacement

### Explanation

Least Recently Used is a selfish algorithm as it replaces the least recently used page. It is basis is on the locality of reference.

#### Advantages:

* Open for full analysis
* Avoids the FIFO “Belady’s Anomaly”
* Easy to choose the faulted page

#### Disadvantages:

* Requires additional data structure to be implemented
* Requires high hardware assistance

### Code Implementation

Concerning code implementation, a location should be specified as a destination for the function that implements the LRU page replacement algorithm. This file, which is named “pagefaults.c” to be located can be found in the directory “/usr/src/minix/servers/vm”. Once the file responsible for page replacements is found successfully, these steps were done:

1. Find the function(s) responsible for page replacement algorithms.
2. Replace it with the algorithm to be implemented.

### Conclusion

Text

Description automatically generated

## Hierarchical Paging Algorithm

### Explanation

Hierarchical Paging (known as multilevel paging) is an algorithm used in operating systems’ memory management where page tables are consisted of two or more levels. Considering two level paging, tables in level one contain pointers that lead to level two whereas level 2 tables contain the page frame information. In multilevel paging, each table except the last one contains pointers to the tables inside the next level. The final table always contains the address and information for pages.



Page tables are saved inside the memory; therefore, at least two or more memory accesses are required to find out which physical memory address inside the main memory a specific page frame occupies.

Diagram

Description automatically generated

In operating systems, hierarchical paging is very essential. It is mostly used when a single page table is bigger than the page frame sizes. This problem means that that page table is not capable of being stored inside the main memory, as it would need to be stored inside pages.

#### How it works:

Multilevel Paging’s keyword is the hierarchical way that highlights how memory management is tackled. First off, pages are logical memory blocks that are divided to a constant size, meanwhile, frames are physical memory that are also fixed in size. A page table’s main role is to translate and map each logical address to a physical one. Here, in this specific case, the page table holds more space than the actual number of frames, which aids in page replacement. Moreover, all free frames should be kept stored somewhere safe, and each page is mapped to a frame in the main memory, that is not necessarily its corresponding one. What makes multilevel paging different from any other type is that a page table points to another one, which creates the hierarchy mentioned before.

#### Advantages:

* Dedicating memory for page tables is simple and cheap.
* The Operating System can use the first free page inside the table.
* External fragmentation is non-existent in the system.
* Page frames are able to spread all over the PM.
* Page mapping is executed in a precise manner.
* Grants the ability to utilize demand pre and demand paging.
* Swapping becomes more efficient.
* Swapping using the Least Recently Used Algorithm is easily implemented.
* Translation Lookaside Buffer can be used to modify the access time.

#### Disadvantages:

* An increase of access time inside the page tables is noticable
* Page tables can become inverted
* Memory specifications (one entry per VM page)
* Update utilizing Multilevel page tables and variable page sizes (super-pages)
* Secured page tables
* Page Table Length Register (PTLR) to limit virtual memory size
* Internal fragmentation

### Code Implementation

By default, MINIX 3 implements a one level paging algorithm. By definition, paging is an efficient memory management operation that is used virtually to organize and keep track of virtual memory. Operating systems use page replacement algorithms to properly allocate memory pages for processes in the ready queue. However, one level of paging might not be enough to manage multiple large processes; therefore, we need to implement at least two-level paging.

Some operating systems use two levels of paging to increase the number of pages for allocation, yet two level paging may also come with its constraints. For that reason, more levels were introduced. Multilevel paging can be organized in various ways to form a hierarchy. It can be composed of two or more levels of page tables, where the top-level page table contains pointers to the next level of tables, and the final table level contains information about pages’ data.

Upon locating the file “pagetable.c” inside the directory “/usr/src/minix/servers/vm”, to change MINIX’s memory management implementation to a multilevel implementation, we must:

1. Find the function responsible for page allocation for processes.
2. Find the page table definition.
3. Define the size for the inner page table.
4. Deduce the size for the outer page table.

The function for page allocation was found to be “pt\_ptalloc” on line 374, responsible for allocating a page table and recording the page address into the directory. This function should be edited in a way such that page allocation is done on two separate levels, where the outermost level points to the innermost page level. This can be implemented by using a data structure that points to the predefined variable “pt\_t \*pt”. There are three parameters passed to “pt\_ptalloc”: the page table, page directory entry (page address), and the page’s corresponding flags that determine if the page is in use. The function first checks the address validity by verifying that the address is a positive integer less than the virtual memory’s maximum. Then the page flags are checked to make sure it is free. To ensure that no data is lost in the memory, we must check that the page directory entry and the page table where the process data is stored is free. Once these validation tests are completed successfully, virtual memory allocation is finalized.

A new page table can be created through the function “pt\_new”. This newly created page table by default points directly to a page; however, we aim to keep track of these page tables by using another page table. To do so, we will use two nested for loops where the outer for loop maps each inner page table to a location in the outer page table. The second for loop will remain as previously implemented. We assumed the inner and outer table sizes are equal. From our studies of the code, we deduced that “I386\_VM\_DIR\_ENTRIES” is the page table size; thus, we divided it by two for equal distribution of space over the table levels. The page offset will not change as it is not affected by the changes made inside the code.

### Code

//We use pt\_tcustom\* pt as a parameter passed to the function to //create a table inside a table

for (i =0; i< I386\_VM\_DIR\_ENTRIES/2;i++) {

for (j = 0; j < I386\_VM\_DIR\_ENTRIES/2; j++) {

pt->pt[i]->pt\_dir[j] = 0; /\* invalid entry I386\_VM\_PRESENT bit = 0) \*/

pt->pt[i]->pt\_pt[j] = NULL;

}

# Requirement 3 - Memory Management

## Memory Management in MINIX 3

MINIX has a uniquely implemented file system by Andrew S. Tanenbaum in the 1980s aiming to copy and replace the usually used file system in Unix. It was created simplifying the complex Unix file system for educational purposes but because its features are bounded it has lost its popularity compared to other systems released in the 1990s.

The MINIX file system is split into six parts.

* + - 1. The Boot Block: this block houses the bootstrap that is required to boot the OS at startup , it’s saved in the first block.
      2. The Superblock: This block stores all the data of the file system where it helps the OS to identify and recognize all the components such as the starting block of the data , the different inode and zone bitmaps sizes , and their true present number.
      3. The inode Bitmap: It is formed of a bitmap that shows whether the inode is in use represented by “1” or free represented by “0”.
      4. The Zone Bitmap: similar to the inode bit map it shows whether the zone is in use represented by “1” or free represented by “0”.
      5. The inode Table: inode” is the data structure that represent any directory or file , it contains the main file or directory attributes such as permissions and metadata while also storing the addresses to facilitate access.
      6. The Data Area: This are storing the files and directories and needs to be the largest in the file system. It contains all the other data blocks found on the hard drive.

16 bytes are allocated for every file entry where 14 are used for its name and 2 are used for the inode part of data ; that’s why filename maximum is fourteen characters. The directories on the other hand can consist of 60 bytes hence a limit of 60 characters for the name. The inoodes number is limited to about 65,535 which limits the number of files since each inode is linked to a file and also limits number of files on each partition. Inside each inode exists a number of zones where each one of them points to a block of address. Inode contains a basic pointer for each memory block up to seven pointers while for larger files an eighth pointer can be found where it points to a block of memory that contains 8 more pointers to fully grasp the data.

The file consists of four zones split into

* 1. First Zone : This zone points to 7 blocks.
  2. Second Zone: This zone points to 512 blocks.
  3. Third Zone: This zone points to 512 blocks.
  4. Fourth Zone: This zone points to 512 blocks.

This indicates that by theory we have around 262,657 total blocks of memory per file calculated as (7+512+512\*512) which rounds up to 256MB since each block is about 1KB. MINIX rationally uses short unsigned integers in the file system ranging from 0 to 65,535 rendering the maximum file size on MINIX to 64 MB which is called indexed allocation further explained in the section below.

## User-Defined Disk Allocation

Methods of file allocation determine how the files are saved in the disk blocks. There are three leading methods which are contiguous allocation, linked allocation, indexed allocation. Those methods are done to make the disk space more efficient and to let us access file blocks faster.

### Contiguous Allocation

This method is the most popular method for file allocation. In this method, every file fill contiguous set of addresses on the disk. To clarify this method, if we have a file that needs n blocks and block a is the beginning, then the file will be in blocks like a, a+1, a+2, a+3,…..a+n-1. Therefore, if the beginning block address and the length of the file are known, then we can know the blocks which have the files. In contiguous allocation method, the directory entry of a file has file name, first block address, length in blocks.

#### Advantages:

* This method gives us fast performance, that’s because reading consecutive blocks of the same file doesn’t need changing place of the disk heads.
* Implementing this method is easy
* This method backs up two access methods: sequential and direct, where in direct access, we can access the nth block of a specified file that begins at memory block j by (j+n)
* This method is very fast because of the low number of seeks and that is a result of contiguous allocation.

#### Disadvantages:

* This method  can cause disk sometime to be fragmented, it endures from external and internal fragmentation and this makes the memory utilization inefficient.
* Increasing file size is complex as it depends on the contingency of memory at that moment.
* Over estimating a file’s size would lead to external fragmentation that wastes unused disk space.
* If the file size was underestimated, it needs to be moved.

A picture containing table

Description automatically generated

*Figure 1: Contiguous allocation of disk space*

### Linked Allocation

Linked allocation is a type of disk allocation used by many operating systems. This type of allocation overpasses the problem that was faced in contiguous allocation as it tackles the disk allocation from a different perspective. This technique is used in a way where disk files become linked. A pointer is dedicated to the start of the directory, and from that part, the link points to the following link, where the end is marked in the directory as well. This lowers the cost of this technique as the storage space used drops by more than 80%. Moreover, this linked allocation gives freedom to distribution of disk blocks as they are not necessarily stored on the disk contiguously, unlike contiguous allocation. Deduced from its name, each block in this method links, or points, to the next block, which kills any external fragmentation.

#### Advantages:

* Higher throughput and memory utilization.
* The directory stores only two pointers.
* Dynamic allocation allowing almost unlimited file size.
* High flexibility that allows increasing and decreasing of file size easily.

#### Disadvantages:

* Delay is present to locate a block that is near the end.
* To reach a specific block, individual traversal is required.
* Slow due to the delay.
* We need to traverse each block.
* Pointers take up space of their own.
* Internal fragmentation.
* Losing a pointer loses the entire file.
* A group of blocks that are back-to-back make the pointers useless.

Diagram

Description automatically generated

*Figure 2: Linked allocation of disk space.*

From the figure above, which illustrates linked list allocation of disk space, an illustration of the directory organization is shown, which highlights that both the start and end pointers are defined. Therefore, here, the start is noted to be the integer nine, which is shown to be the block that holds number one. Meanwhile, the end is stored to be twenty-five, which, through the figure, is the first block with a negative integer, “-1”. This negative number marks the end of the linked list, which is equivalent to a NULL pointer. Hence, we can conclude that the allocation consists of 25 blocks. While analyzing the figure, we notice in the directory something else under the name of “file jeep”. This is the name of the file with the corresponding start and end pointers.

Chart

Description automatically generated

*Figure 3: File-allocation table.*

The file allocation table portrayed above, abbreviated to FAT, stores in another individual table the links, which helps with memory cache, which increases the speed for accessing a specific file through its disk blocks. As shown above, the directory entry holds a number that sends us to the FAT, which is composed of linked pointers. This methodology is tackled by DOS.

### Indexed Allocation

In the index allocation method, we have an additional block, and that block is known as the index block. For each file, there is an individual index block that contains the pointers to all the blocks occupied by a file. In the index block, the ith entry contains the disk address of the ith file block. We can see in the below figure that the directory entry holds the address of the index block.

Diagram

Description automatically generated

*Figure 4: Indexed allocation of disk space*

#### Advantages of Index Allocation

* Ends external fragmentation issue.
* It provides immediate access and therefore allows direct access to chunks of files.

#### Disadvantages of Index Allocation

* The overhead cursor is higher.
* If an index block is not right, we may ruin the whole file.
* Some disk space is lost (comparable to linked lists or FAT tables) because, regarding how many information blocks the file holds, a whole index block must be reserved for each file.
* Creating an index for a tiny file is a loss

For files of large sizes, a single index block cannot contain all the pointers. To overcome this problem, there is a multiple mechanism which we can use:

1. Linked Scheme: – In the linked design, to hold the pointer, two or more than two index blocks are connected together. Each block holds the address of the next index block or a pointer.

2. Multilevel Index: – in a multilevel index, we use a first-level block to refer to the second-level index block, which in turn points to the disk blocks kept by the file. Depending on the file's maximum size, we can expand this up to 3 or more than 3 levels.

3. Combined Scheme: There is a special block called a knowledge node in a combined design (Inode). The inode contains all the file related data, such as authority, name, height, etc. Inode's leftover space is used to save the addresses of the Disk Block that contain the original file as seen in the image below. The starting pointer in the inode is used to point out the direct blocks.

This means that the pointer contains the addresses of the disk blocks that are made up of data from the file. The next few pointers are used to denote the indirect blocks. The indirect blocks are of three forms, which are single indirect, double indirect, and triple indirect. The single indirect block is the disk block that does not contain the data from the file, but the address of the blocks holding the data from the file. Similarly, the file data is not stored in double indirect blocks, but rather the disk address of the blocks containing the block address, including the file data.

Diagram

Description automatically generated

*Figure 5: The UNIX inode.*

Methods of file allocation determine how the files are saved in the disk blocks. There are three leading methods:

* Contiguous Allocation
* Linked Allocation
* Indexed Allocation

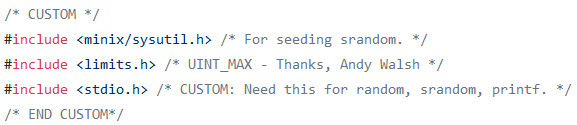
These methods are done to make the disk space more efficient and to let us access file blocks faster.

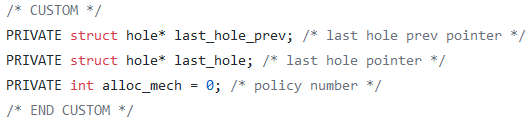
## Compare Performances

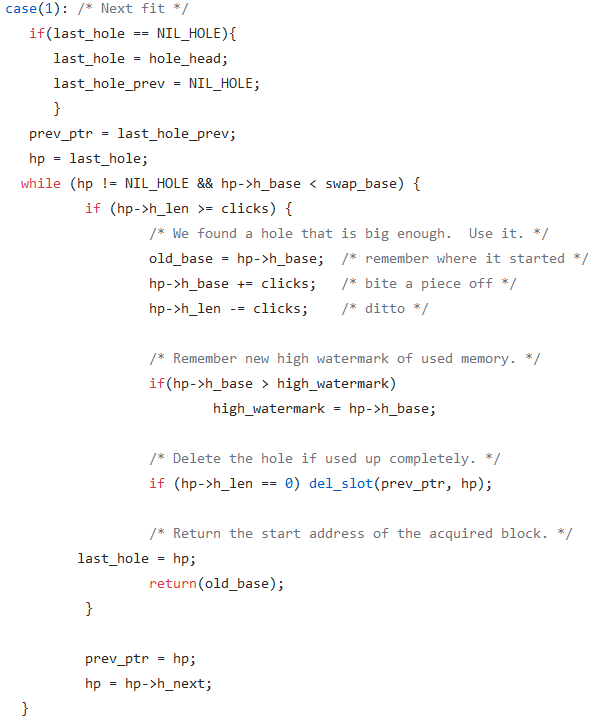
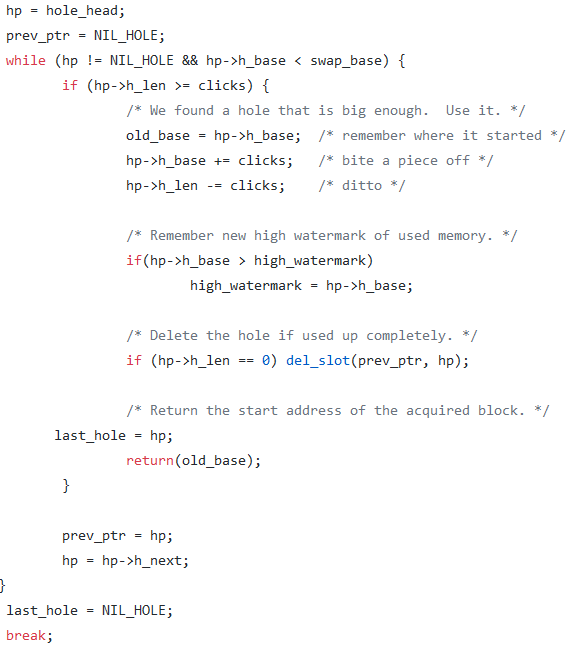
For sequential access files, the optimal allocation method is different than for random access files and is also different for small files than for large files. Some systems support more than one allocation method, which may require defining how to use the file at the time it is allocated (sequential or random access). These programs provide facilities for reformation as well. In the meantime, it has been noted that other systems use contiguous access for small files and turn to an indexed scheme automatically when file sizes reach a certain threshold. And, of course, some systems change their allocation schemes to better fit the hardware characteristics for best output (e.g., block sizes).

## Code Implementation

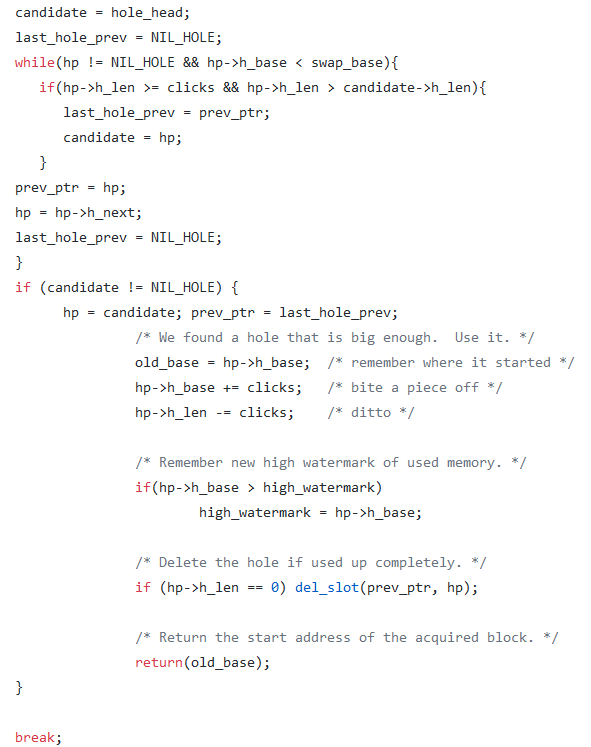
Changes were made to the files “alloc.c”, “proto.h”, and “table.c”, with the path “/usr/src/minix/servers/pm“



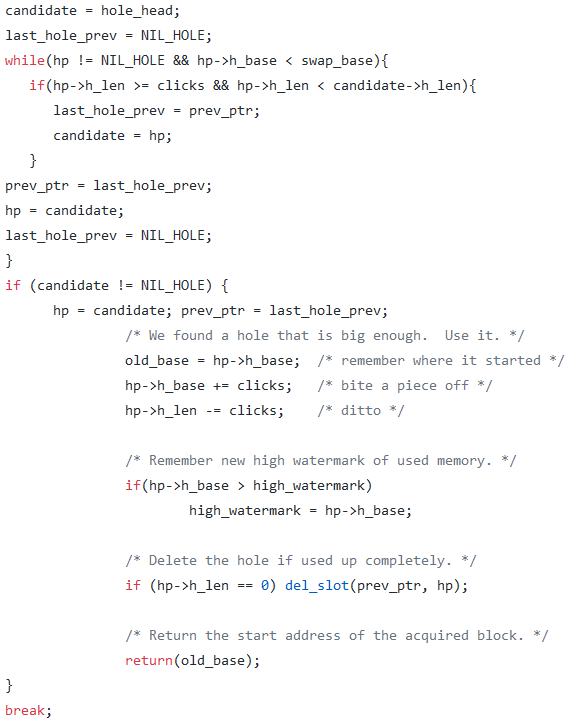


Next fit Implementation:

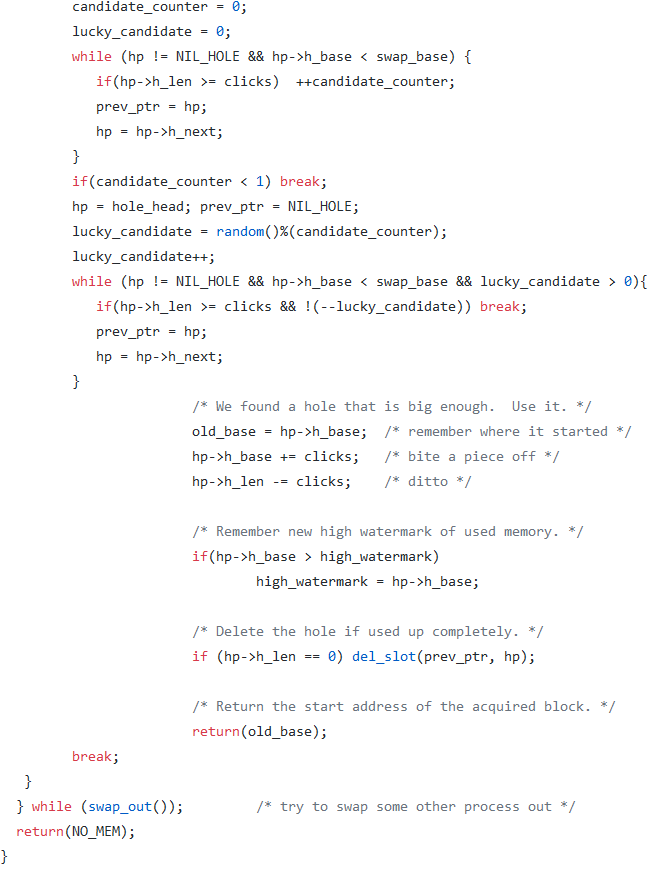
Worst Fit Implementation:



Implementation of best fit:

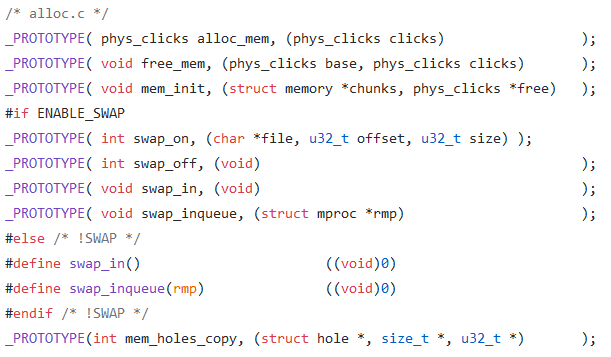


Implementation of random fit:



Change in proto.h: edited the prorotypes for alloc.c

Path: /usr/src/minix/servers/pm/proto.c



In table.c these were added to the call\_vec array