Program: CESS

Course Code: CSE335

Course Name: Operating Systems

Examination Committee

Ain Shams University
Faculty of Engineering
Fall Semester – 2021

Student Group Personal Information

Prepared By:

Ibrahim Ahmed Hassan Ahmed Hassab Elnaby	16p3062
Sohayla Ihab Abdelmawgoud Ahmed Hamed	19P7343
Seifeldin Sameh Mostafa Mohamed Elkholy	19P3954
Salma Ihab Abdelmawgoud Ahmad Hamed	19P8794
Noorhan Hatem Ibrahim Mohamed	19P5821
Mohamed Ahmed Ebrahim Elsaied	17P6089

Class/Year: **2021**

Plagiarism Statement

I certify that this assignment / report is my own work, based on my personal study and/or research and that I have acknowledged all material and sources used in its preparation, whether they are books, articles, reports, lecture notes, and any other kind of document, electronic or personal communication. I also certify that this assignment / report has not been previously been submitted for assessment for another course. I certify that I have not copied in part or whole or otherwise plagiarized the work of other students and / or persons.

Contents

1.0 Requirement One

1.1 Internal Structure of Operating Systems

1.1.1 Simple Monolithic Structure

The monolithic architecture is a very simple operating system structure in which all components of the operating system are located in the kernel space. System calls implement all the monolithic operating system's services including process management, memory management, and concurrency. Device drivers can be introduced to the kernel in the form of modules.

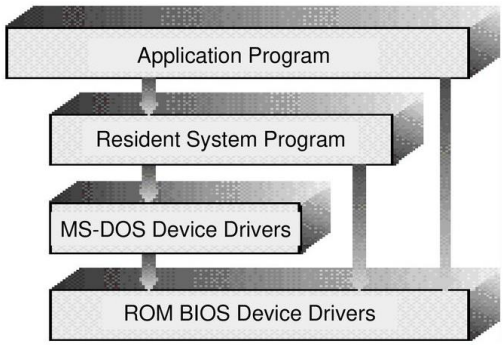


Figure 1: MS-DOS Monolithic Architecture

Advantages:

- Very fast execution as all the components are implemented under the same address space.
- Processes run entirely in one address space which reduces performance overhead.
- Increases application performance due to the small number of interfaces between an application and the hardware it requires.
- Easy to develop.

Disadvantages:

- If any component malfunctions, the entire system will fail.
- To add or remove a service, the entire operating system has to be modified.
- Does not allow for data hiding.
- There are no clear boundaries between the modules in the system.

1.1.2 Non-Simple Semi-Layered Structure

The non-simple semi-layered architecture is a modified and improved form of the simple architecture, in which the operating system is divided into two main separate parts: the system programs and the kernel. The kernel contains all components of the operating system that are above the hardware and below the system call interface, such as memory management and process management.

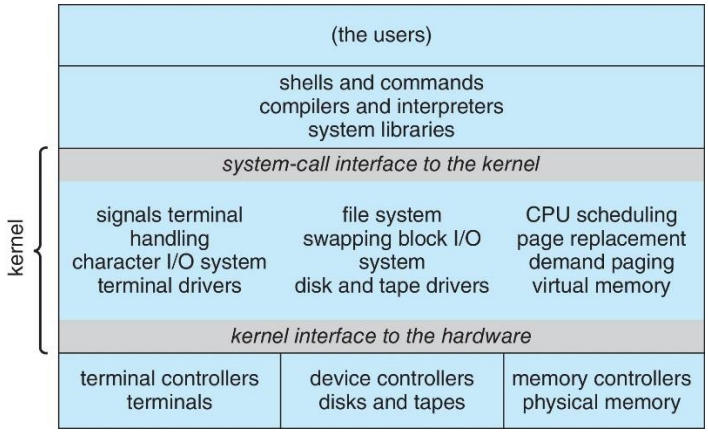


Figure 2: Traditional UNIX Semi-Layered Architecture

Advantages:

- Simple to implement.
- Has higher performance and less performance overhead than a fully-layered architecture.
- More extensible and traceable than the simple monolithic architecture.

Disadvantages:

- Much less flexible and extensible than a fully-layered architecture.
- Lower performance and execution speed than the simple monolithic architecture.

1.1.3 Layered Structure

The layered architecture is an operating system structure in which the system is separated into several layers, each of which is founded and constructed upon the layers below it. Therefore, each layer can only utilize the functions/operations of the layer below it; but not the layers above it. The topmost layer is the user interface of the computer system while the bottommost layer (layer 0) is its hardware.

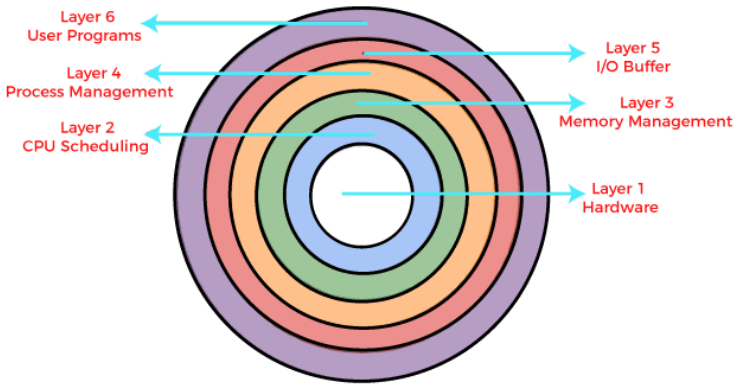


Figure 3: Layered Architecture

Advantages:

- Makes the debugging process and system verification process easier. Layers can be debugged from the bottom to the top, and if an error takes place while debugging a layer, the error must be in that specific layer, as the layers below it function properly (already debugged) and it does not use the services of the layers above it.
- Allows for the operating system to be modified and improved upon smoothly as a specific layer's implementation can be changed without disrupting other layers.

Disadvantages:

- Data must be modified and passed on at every layer. This increases the system's overhead.
- Worse application performance in comparison to the simple monolithic architecture.
- Careful planning and design of each layer is needed.

1.1.4 Microkernel Structure

The microkernel architecture is an operating system structure in which all non-essential components of the operating system are removed from the kernel entirely and made to be system programs and user programs in the user space. This greatly reduces the size of the kernel hence the name “microkernel”. Communication between the modules in the user space happens by message-passing via the microkernel. Microkernels are explained in more details on page #.

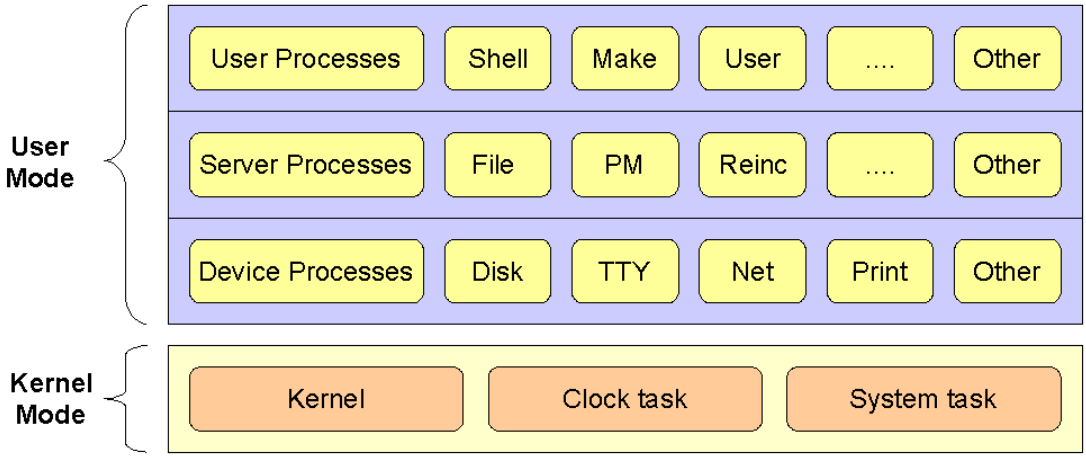


Figure 4: Microkernel Architecture

Advantages:

- New services can be added to the operating system without having to modify the kernel.
- Increases reliability and security; if one component of the operating system malfunctions, the rest of the system will not be harmed.
- The small size of the microkernel allows it to be tested effectively.
- Allows the operating system to be portable to different platforms.

Disadvantages:

- The system’s performance is degraded by the increased need of communication between the user space and the kernel space.
- Context switches are required when drivers are implemented and function calls are required when processes are implemented.

1.1.5 Modular Structure

The modular architecture is an operating system structure in which the kernel contains a few core components while the remaining components and services are added to the kernel (at run-time or boot-time) in the form of dynamically-loadable modules.

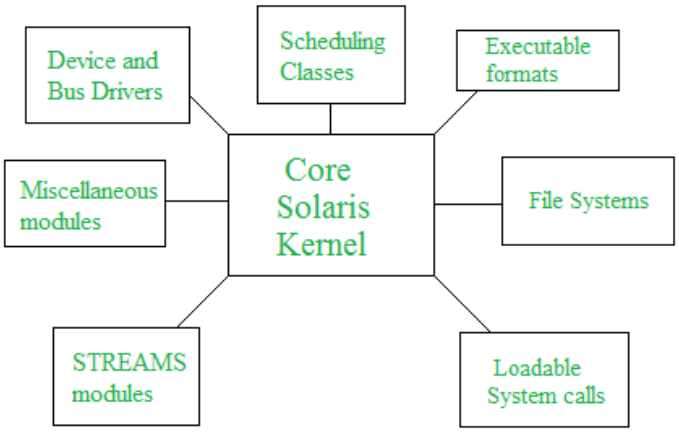


Figure 5: Solaris Modular Architecture

Advantages:

- High flexibility as it allows for adding and removing functionalities with great ease and security.
- The modules are separated and given their own address spaces which allows for pinpointing and isolating malfunctioning modules.
- Allows for hiding critical modules. This can be done by loading the critical modules only when they are needed and then removing them when they are no longer needed.

Disadvantages:

- Any module can view and interfere with the execution of all other modules.
- It is difficult to implement strong and effective security restrictions on critical modules.

1.1.6 Hybrid Structure

Most operating systems do not adhere strictly to a single architectural model; they combine elements of multiple structures to achieve their desired performance goals and security needs as well as various other standards.

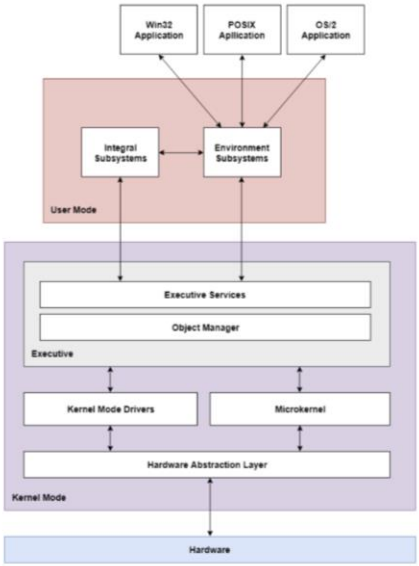


Figure 6: Windows Hybrid Architecture

Advantages:

- Allows the operating system to benefit from the advantages of two or more different architectures by applying each in the part of the operating system where it will be most efficient.
- Gives the operating system the ability to sidestep a few disadvantages of a particular architectural model but utilizing another model where the limitations are most significant.

Disadvantages:

- Combining multiple architectures can cause the limitations of more than one architectural model to affect the system at the same time; or can create new disadvantages unique to the hybrid.
- Careful planning, design, and testing of the system is needed to ensure the combined architectures work smoothly together.

1.2 Algorithms Used in Different Components of Operating Systems

MINIX 3, like most other operating systems, is made up of a plethora of components (small partitions of distinct roles and functionalities) that together allow all parts and elements of the computer system to function together smoothly. Among the most vital of those components are

- File Management
- Process Management
- Main Memory Management
- Secondary Storage Management
- Input/Output Device Management
- Network Management
- Security Management
- Command Interpreter System

For these components to carry out their required responsibilities, each component must be designed in such a way that accommodates the developers' end-goals for the component and how they want it to function. When designing each component, there are various approaches and algorithms that can be chosen from based on how the final operating system is intended to behave.

1.2.1 File Management: File Allocation Methods

File Management is mainly concerned with how and where files and directories (cataloging structures that store references to other files or even other directories) are stored, how disk space is allocated and managed, and how to ensure reliability, efficiency, and consistency with regards to file systems.

The most important aspect when it comes to storing files is overseeing and monitoring which disk blocks are assigned to which file. There are several different methods to accomplish this, including:

- **Contiguous Allocation**

It is the least complex method of disk block allocation. Contiguous Allocation is the process of storing each file as a connected set of consecutive disk blocks (see Fig.2).

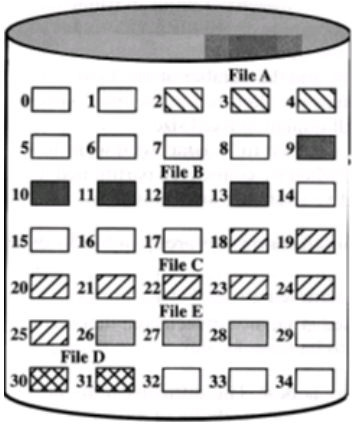


Figure 7: Contiguous Allocation

Advantages: It is simple to implement because the OS only has to keep track of the disk address of the first block and how many blocks are there in the file to be able to find any block allocated to the file. Another advantage is that it has high performance because only one seek is needed to read the entire file.

Disadvantages: The disk will be made up of file sections and empty spaces (fragmentation) which will cause the disk to become full quickly. Another disadvantage is that file size has to be specified when a file is created and the size cannot be extended or changed.

- **Linked List Allocation**

It is the process of storing files such that each file consists of a linked, but not contiguous, list of disk blocks where the first word in each block acts as a pointer to the successive block (see Fig.3).

Advantages: No disk fragmentation other than internal fragmentation in each file's final block. Plus, the directory enter only needs to store the address of the beginning block.

Disadvantages: This random-access method of reading a file is much slower than the straightforward sequential method of Contagious Allocation. Furthermore, the amount of data that is stored in each block is not a power of 2 anymore which is not ideal as most programs read and write in blocks of a size that is a power of 2.

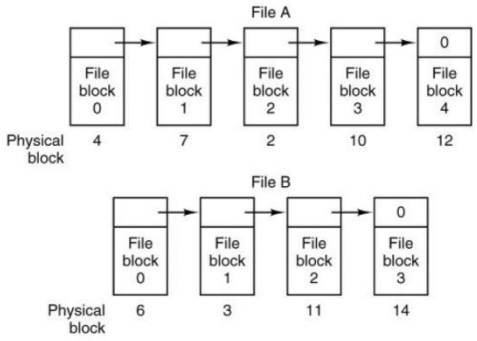


Figure 8: Linked List Allocation

- **Linked List Allocation Using a Table in Memory**

It is a modified form of Linked List Allocation where the first word of each disk block (the pointer word) is removed and put in a table memory known as a File Allocation Table (FAT) where each pointer word in a block of the table points to the next block containing the successive pointer word. This pattern repeats until a block containing a termination marker (marker that cannot be the valid marker of a successive block, such as -1) is reached (see Fig.4).

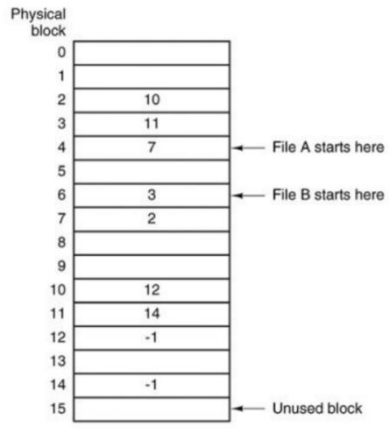


Figure 9: Linked List Allocation Using a Table in Memory

Advantages: eliminates the aforementioned disadvantaged of tableless Linked List Allocation. By placing the pointer words in a table in memory, reading a file will be much quicker and the data stored in each block can be a power of 2 once again.

Disadvantages: the entire FAT must stay in main memory at all times; therefore, it takes up a portion of the main memory.

- **I-Nodes**

It is the process of associating each file with an "index-node" that contains the disk addresses of all the file's assigned blocks; the I-node can then be used to track all the file's blocks.

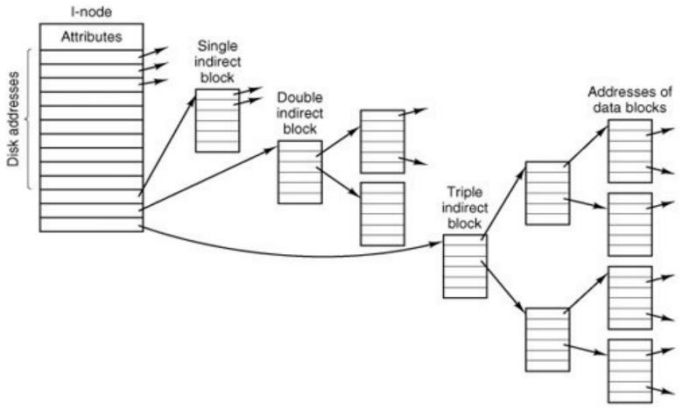


Figure 10: I-Nodes

Advantages: i-nodes only need to be in main memory when their corresponding files are open. Additionally, the space taken up i-nodes is much smaller than that taken up by a FAT.

Disadvantage: i-nodes have an unchangeable number of disk addresses which may cause a problem if a file exceeds the limited number of disk addresses. However, this can be solved by using some disk addresses at the bottom to store addresses of "indirect blocks" that hold even more data block addresses (see Fig.5).

1.2.2 Process Management: Process Scheduling Algorithms

Process Scheduling is the most important aspect of Process Management. The OS needs to be able to organize and handle processes competing for the CPU effectively and reasonably to achieve optimal CPU usage.

At any point in time, a process that has been admitted into the CPU exists in one of three states: "Ready" – meaning it is ready to execute once its turn comes, "Running" – meaning it has been allowed to execute and is currently doing so, or "Waiting" – meaning it is waiting for an input or output event to take place before it becomes ready to execute.

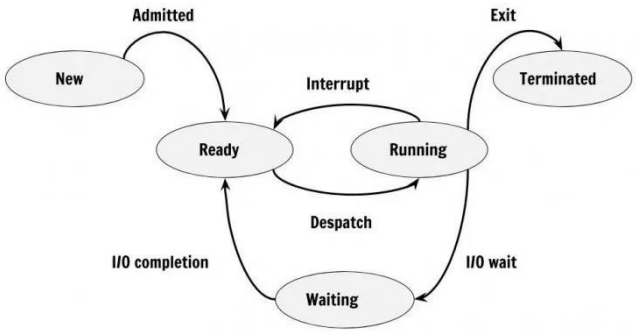


Figure 11: Process States

If there are several processes in the "Ready" queue waiting to be allowed to execute, the OS needs a method of suitably arranging those processes to determine which process should be allowed to execute first. To do so, several Scheduling Algorithms have been developed; each of which sets out to achieve a certain goal with its scheduling technique.

Process Scheduling Algorithms:

- **First-Come First-Served (FCFS)**

  FCFS is a non-preemptive algorithm that organizes ready processes in the exact order they arrived in. As more and more processes request the CPU, the are added to an ongoing queue of ready processes that will be made to execute sequentially.

Advantages: easy to understand, simple to implement, and does not cause starvation (when a process never gets a chance to execute).

Disadvantages: usually results in high average waiting time, low CPU utilization, and cannot be used in time-sharing systems due to lack of preemption.

- **Shortest Job First (SJF)**

SJF is a scheduling algorithm that operates on the assumption that the run-times of the ready processes are already known; based on that, the scheduler arranges ready processes in order of the time they will take to execute. Once a process is allowed to execute, it cannot be replaced by another process even if a shorter process arrives since SJF is non-preemptive.

Advantage: low average waiting time.

Disadvantages: time required by each process must be known by CPU before execution, and may cause starvation if new short processes keep entering the ready queue while a long process waits for its turn.

- **Shortest Remaining Time First (SRTF)**

SRTF is a preemptive variety of the SJF algorithm that also operates on the assumption that process run-times are already known; however, in SRTF, the scheduler always picks the process with the remaining shortest time. Meaning if a shorter process were to arrive while a process is executing, the scheduler would evict the currently executing process in favor of the quicker process.

Advantages: low average wait time.

Disadvantages: high CPU overhead, time required by each process must be known by CPU before execution, and may cause starvation if new short processes keep entering the ready queue while a long process waits for its turn.

- **Round Robin (RR)**

  RR is the considered to be the fairest and most used scheduling algorithm. It is a preemptive algorithm where each process is given a specific time limit to execute (quantum) and if it does not finish during its assigned quantum, it is evicted and placed at the back of the queue while the next process is allowed to execute for its assigned quantum.

  Advantages: prevents starvation, and allows each process to get an equal share of the CPU.

  Disadvantages: causes high average waiting time, and can lower CPU efficiency and utilization if the quantum is too short.

- **Priority Scheduling**

  Priority Scheduling is a preemptive scheduling technique where each process is assigned a priority, and the ready process with the highest priority is the one chosen by the scheduler to start executing. Priority Scheduling can be combined with other scheduling algorithms like RR such that the highest priority process is allowed to run for an assigned quantum and if it does not finish in time, it is evicted and the next highest priority process executes.

  Advantage: ensures essential and very important processes can execute quickly.

  Disadvantage: may cause starvation.

- **Multilevel Queue Scheduling (MLQ)**

  Since different types of processes can have different scheduling needs, it may be best to separate ready processes into different classes such that each class forms a separate queue of ready processes which would allow each queue to follow a different scheduling algorithm. Higher priority processes are put into top level queues and lower priority processes are put into low level queues. The queues are executed in order from top to bottom.

Advantage: allows the system to apply the most appropriate scheduling algorithm to each set of processes.

Disadvantage: may cause starvation of processes in the lowest queue.

- **Multilevel Feedback Queue Scheduling (MLFQ)**

It is a modified form of Multilevel Queue Scheduling in which processes are allowed to move between queues. In MLFQ, the scheduler continuously analyzes the execution (burst) time of different processes and changes their priority level based on the analysis.

Advantages: prevents starvation, and has a low scheduling overhead.

Disadvantages: very complex, and inflexible.

1.2.3 Memory Management: Memory Allocation Algorithms

There are several types of mechanisms and functionalities needed in memory management to ensure memory-related functions are efficient and dependable. One of those required mechanisms is a way to allocate memory to a process.

- **Single Contiguous Allocation**

It is the simplest memory allocation method. Single Contiguous Allocation is the process of making all of the memory available to a process, except for the part of the memory reserved for the operating system,

Advantages: it provides high performance as well as easy access to memory. Additionally, it also increases processing speed.

Disadvantages: causes memory wastage, causes memory inflexibility, and does not support multiprogramming.

- **Partitioned Allocation**

  It is a memory allocation method in which the memory is partitioned into several different blocks. Processes are allocated based on their needs.

  Advantage: easy to implement and design.

  Disadvantages: causes high internal fragmentation, and is limited by the number of partitions.

- **Paged Memory Management**

  It is a memory management method in which the memory is divided into page frames that are all of a fixed specific size. The page frames are then used in a virtual memory environment.

- **Segmented Memory Management**

  It is a memory management method in which a process is segmented based on logical groupings of a its code. It can sometimes be implemented along with paging such that the process's segments are divided into pages.

In operating systems that use **Partition Allocation**, if there are several available blocks to accommodate a process's request, only one must be chosen. Therefore, a partition allocation algorithm is required.

- **First Fit Algorithm**

  A partition allocation method in which the first suitable (large enough) partition from the top of main memory is selected to be allocated to the process.

  Advantage: very quick; it takes up very little CPU time.

Disadvantages: if there are several large blocks on the top of the main memory, the first fit algorithm may allocate them to small processes which would cause them to break up and thus, they will not be available for large processes that may later require them. It also has a high chance of creating several tiny internal fragments at the top of the main memory that will be wasted.

- **Next Fit Algorithm**

A modified form of the First Fit partition allocation method that looks for the first suitable partition starting from the last allocation point.

Advantage: reduces the possibility of causing tiny internal fragments at the top of the Main memory like the First Fit algorithm might do.

Disadvantage: like the First Fit algorithm, it breaks up large memory blocks that could have been utilized by large processes in the future.

- **Best Fit Algorithm**

A partition allocation method in which all available partitions are examined and the smallest one that can accommodate the size of the process is selected to be allocated to the process.

Advantage: it is memory efficient as it causes the least wastage of unutilized memory space.

Disadvantage: takes up a lot of CPU time to complete its search.

- **Worst Fit Algorithm**

A partition allocation method in which all available partitions are examined and the largest one is selected to be allocated to the process.

Advantage: will cause the creation of relatively large internal fragments than can still be utilized by small or medium-sized processes.

Disadvantage: quite time-consuming.

- **Quick Fit**

A partition allocation method in which different lists of commonly used partition sizes are kept. A table containing entries that point to the top of each list is also maintained.

Advantage: finding an available block of the needed size is done very quickly.

Disadvantage: after a process terminates or swaps out of a block, it is difficult to find the blocks adjacent to it and see if a merge is possible.

1.2.4 Memory Management: Page Replacement Algorithms

Another core functionality needed in memory management is a mechanism that can deal with page replacement in memory. When the main memory is full and a new page needs to be brought in, the OS needs to select a page to remove from memory to vacate a space for the new page. To decide the most appropriate page that should be removed, there are several theoretical and experimental Page Replacement Algorithms that have been developed.

- **The Optimal Page Replacement Algorithm**

It is the most ideal Page Replacement Algorithm; however, it is physically impossible to implement. The algorithm works by removing the page that will not be used (referenced) until the largest number of other instructions will be executed; therefore, the page that will be sitting in memory without being utilized for the longest amount of time will be the page removed. Unfortunately, this is algorithm cannot be implemented as there is no way for the OS to know when each page will be used next.

Advantage: requires low assistance as the data structures it uses are light and simple.

Disadvantage: impossible to implement.

- **The Not Recently Used (NRU) Page Replacement Algorithm**

The NRU algorithm works by removing a random page from a class of pages that have not been recently used. To approximate the "used" status of pages, each page is given two status bits: R and M. The R bit of a page is set to 1 whenever it is referenced. The M bit of a page is set to 1 whenever it is modified upon. When a process is first started, the OS sets the R and M bits of all the pages to 0. Throughout the execution of the process, R and M values of pages that are referenced or modified will be updated respectively. Furthermore, the OS will periodically (at constant time intervals) clear all R bits back to 0 to be able to distinguish recently referenced pages from pages that were referenced a while back. When a page replacement needs to occur, the OS will divide all pages into four classes based on their R and M bits.

  - Class 0: R=0 M=0 (Not Referenced, Not Modified)
  - Class 1: R=0 M=1 (Not Referenced, Modified)
  - Class 2: R=1 M=1 (Referenced, Not Modified)
  - Class 3: R=1 M=1 (Referenced, Modified)

The NRU algorithm will then select a random page from the lowest available class. It first goes for Class 0, if there are no pages in class 0, it will go for Class 1 and so on.

Advantage: moderately easy to implement.

Disadvantage: does not offer the best performance.

- **The First-In, First-Out (FIFO) Page Replacement Algorithm**

FIFO, as its name suggests, operates by keeping a list of all pages currently stores in the memory. The page on top is the oldest page that has been in memory (first-in) and the page at the very end is the newest page in memory. When a page replacement needs to occur, the OS removes the page on top and every page move up the list by a level. Then, the new page going in is placed at the bottom of the list.

Advantage: simple to comprehend and implement.

Disadvantages: it has low efficiency compared to various other algorithms and it also requires all pages to be accounted for. Additionally, it may lead to Belady's anomaly which is a glitch where increasing the number of pages leads to an increase in the number of page replacements needed.

- **The Least Recently Used (LRU) Page Replacement Algorithm**

The LRU algorithm operates on the basis that pages that have been recently used multiple times will most likely be heavily used again, while pages that have not been used for a while will probably not be used in the near future. The LRU algorithm is implemented by keeping a linked list of all pages currently in memory. The most recently used page is at the front and behind it are pages of gradually less and less usage, ending with the least recently used page at the very back. The list is modified every time there is memory reference. When a page replacement needs to happen, the OS simply chooses the page at the back.

Advantage: avoids Belady's anomaly.

Disadvantage: needs additional data structures and hardware to be implemented.

- **The Second Chance Page Replacement Algorithm**

It is a modified version of FIFO that implements elements of the NRU algorithm. It keeps a list of pages from oldest to newest like in FIFO; however, it also assigns each page an R bit and implements a periodic clearing cycle like in NRU. When a page replacement needs to occur, the OS analyzes the oldest page in memory (the page on top), if its R bit is 0 (meaning the page is old and also unused), the page is replaced. If its R bit is 1, the bit is cleared and the page is moved to the bottom of the list as if it had just arrived. The OS then inspects the new page on top and the process is repeated until a page with an R bit equal to 0 is found; that is the page that will be removed. If no page with an R value of 0 is found after running through the whole list, then the oldest page will be on top once again but now it has an R value of 0 so it gets removed.

Advantage: it offers higher efficiency than FIFO and is always ensured to terminate.

Disadvantage: more difficult to implement than FIFO.

- **The Clock Page Replacement Algorithm**

This algorithm is a better tuned version of the Second Chance algorithm. To avoid the inefficiency of constantly rearranging the list of pages, all pages currently in memory are kept on a circular list (in order of age) with a hand pointing to the oldest page. When a page replacement needs to occur, the OS analyzes the page currently being pointed at by the hand (the oldest page in memory). If the page's R bit is 0, then the OS removes it and the new page takes up the recently evacuated position and the handle moves to the following page (the current oldest page). Otherwise, its R bit is cleared and the handle advances to the second oldest page and the process repeats like in the Second Chance algorithm.

Advantage: more efficient than the Second Chance Algorithm.

Disadvantage: more difficult to implement than FIFO and Second Chance Algorithm.

1.3 Comparative Study

If you are a developer designing an operating system, it might be difficult to choose between all the various available internal structures, methods, and algorithms that can be implemented in different components of your operating system.

There is no single correct answer to the question of which algorithm or approach you should implement; however, you make an informed decision by analyzing how you wish your operating system to function, the resources you have, and your technical abilities.

Part One: Internal Structures

If you are still starting out in operating system design and are not very advanced in your technical abilities, and/or if you do not possess a great deal of resources, the Simple Monolithic Model is the best model for you to implement, as you will be readily able to understand its design and functionalities. However, you have to determine all the services your operating system is expected to have before you start development or will be forced to modify the entire system if you wish to make a change in any of its components.

Conversely, if you possess advanced technical abilities, you should make your decision based on your available resources and the qualities you want your operating system to have. If you value security and reliability above all else, the Microkernel Model might be your desired model, as it is the most dependable. However, you value performance over reliability, the Microkernel model may be of less value to you than reliability, you may be interested in the Non-Simple Semi-Layered Model.

If you want flexibility to modify your operating system, or if you are not fully-decided on the services the end product should have, you should choose between the Microkernel Model, the Layered Model, or the Modular Model. They all offer high flexibility that allows for altering specific parts of the system without having to modify all of it.

Finally, you can try a Hybrid Model if you possess very high skills and a lot of resources. If you have a very unique system in mind, combining two or more approaches in your operating system may help you in achieving your end-goals.

Part Two: File Allocation Methods

If you are not very skilled, Contiguous Allocation is the best method to choose, as it is very simple to implement. However, it is very inefficient and can cause severe fragmentation; therefore, if you have the required skills and want a smooth-running system you should choose Linked List Allocation Using a Table in Memory or i-nodes. You can try regular Linked List Allocation if you do not intend to design a very complex system but want a method more efficient than Contiguous Allocation.

Part Three: Process Scheduling Algorithms

Different process scheduling algorithms work best in different instances; therefore, if you are very skilled then you should pick MLQ or MLFQ. If you lack the technical skills to design multilevel algorithms, you should carefully consider what you value most in the operating system you wish to develop, as a single algorithm will be applied to all processes. If you prioritize stability and wish to avoid issues like starvation, you can choose between FCFS and RR. In case you want a system with low average waiting time then the best options for you are SJF and SRTF. Finally, if the system being designed is supposed to prioritize certain processes over others, then you should choose priority scheduling.

Part Four: Memory Allocation Algorithms

The Single Contiguous Algorithm is the best algorithm of choice only if you are designing an operating system that is not intended to be very complicated or if you are not very skilled. In almost all other circumstances, it is better to go for a non-contiguous approach to reduce memory waste and to allow for multiprogramming.

If you pick a non-contiguous (partition) approach, you should also choose a partition allocation algorithm to provide available memory blocks to processes properly. If you value avoiding internal fragmentation and waste reduction over speed, the Best Fit or Worst Fit algorithm is the right option for you. If you value speed over all else, the Quick Fit algorithm is the best choice for your operating system. You can also pick between the First Fit and Next Fit algorithms if you want a quick algorithm but are willing to sacrifice some speed in favor of a system that runs more smoothly.

Part Five: Page Replacement Algorithms

The best page replacement algorithm to implement in a system in the Optimal Page Replacement Algorithm; however, it is regrettably not feasible. Consequently, you must choose between the remaining algorithms can be practically implemented. First of all, consider the resources you have and the costs you are willing to pay. The LRU and NRU algorithms require specific additional hardware to function well; therefore, if you cannot afford or cannot obtain the required hardware you should eliminate those two choices.

FIFO is very easy to understand and implement, so if you are still a beginner or if you intend to design a simple system, it is probably the right option for you. However, you will be making a great sacrifice in terms of efficiency. If you want a more efficient system and are able to design complex algorithms, you can choose between the Second Chance algorithm or the Clock algorithm.

2.0 Requirement Two

2.1 Internal Structure of MINIX 3

MINIX 3 is an open-source OS that was designed with the primary intention of being highly self-healing (fault-tolerant), secure, and reliable; while also maintaining great flexibility. It was loosely structured and based on older versions of MINIX (MINIX 1 and MINIX 2); however, the older MINIX versions were designed to function more as teaching tools; not as serious, reliable systems like MINIX 3.

To accomplish the aforementioned qualities of reliability and fault-tolerance that MINIX 3 is required to have, the kernel of MINIX 3 was made to be very small. It is a microkernel (approximately 5000 lines of code) that runs in the privileged level (also known as kernel mode). The remainder of the OS runs in the less-privileged user mode in the form of several protected and isolated processes. This greatly reduces the probability of the whole kernel crashing if a bug where to occur, unlike in monolithic systems.

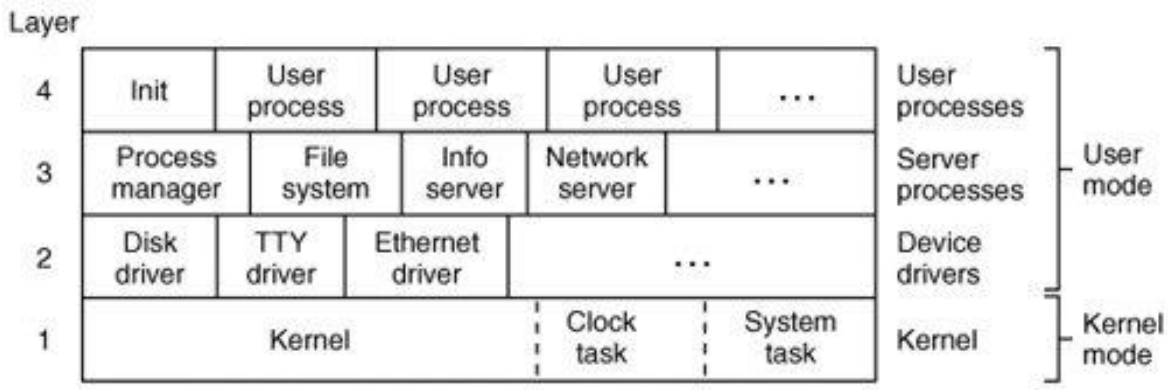


Figure 12: Internal Structure of MINIX 3

2.1.1 Kernel Mode

The microkernel is the main component of the kernel mode. It is responsible for scheduling processes and for handling the transitions between the states of those processes (ready, running, blocked, etc.); additionally, it carries out message handling between processes. This includes checking for legal destinations, finding “send” and “receive” buffers in RAM, and copying bytes from the sender to the receiver.

Besides the microkernel, there are 2 other modules found in kernel mode: “Clock Task” and “System Task”. They are separate processes from the microkernel and have their own stack calls despite being compiled into the address space of the microkernel. System Task and Clock Task operate in the same way device drivers do; but they are not user-accessible, unlike device drivers.

2.1.2 User Mode

There are several layers in the User Mode; though, they can sometimes all be considered one layer due to the microkernel essentially dealing with them all in the same manner. They are all confined to set user mode instructions, and none of the layers has the privilege to directly access input/output ports or access memory segments that were not allocated to it.

The separation between the User Mode layers is fundamentally based on the differences between the number of privileges the layers possess. The bottommost User Mode layer (labelled “Layer 2” in Fig.1) possesses the most privileges out of all the layers. The processes contained within it (device drivers) have the ability to make privileged kernel calls, such as calling the system task to read/write data on the input/output ports, or calling the system task to copy specific data between address spaces. The next layer (labelled “Layer 3” in Fig.1) also possesses some privileges, although less than those of layer 2. However, the topmost layer (labelled “Layer 4” in Fig.1) has no distinct privileges.

Layer 3 contains the server processes known as “servers” that offer several essential services to user processes. Due to being less privileged than device drivers in layer 3, servers are not able to communicate with input/output devices directly; however, they are able to communicate with device drivers and request from device drivers to carry out input/output communications on their behalf. Servers are also able to use the System Task to communicate with the microkernel.

Three of the most notable servers are the File System Server, the Process Manager Server, and the Reincarnation Server. The File System Server, as its name suggests, is responsible for all file system calls, while the Process Manager Server is responsible for all the system calls that manage process execution, such as starting or stopping a process, as well as causing a process to

fork or wait. The Reincarnation Server handles malfunctioning device drivers by killing devices that are failing and restarting new copies of them.

The topmost and least privileged layer, Layer 4, is where the user processes are contained. User processes are user-written programs to be run on the computer system.

2.2 Communication Between Processes in MINIX 3

For processes to be able to communicate with one another, MINIX 3 has three primitives that are responsible for receiving and sending messages. These primitives are called by the three specific C-Library procedures “send(destination, &message)”, “receive(source, &message)”, and sendrec(src_dst, &message).

The **send** procedure is used to send a message to a process; **receive** is used to receive a message from a process; and **sendrec** is used to send a message to a process and wait for a reply back.

Task processes, driver processes, and server processes are only permitted to send messages to and receive messages from certain processes. Processes can exchange messages with processes horizontally (in their same layer) or with processes that are in the layers adjacent to them. On the other hand, user processes cannot communicate horizontally with other user processes; but they can communicate with server processes in layer 3.

Process communication can result in problems when a process sends a message to another process that is not waiting for to receive a message. MINIX 3 utilizes the rendezvous approach to avoid theses problems; it blocks the sender until the receiving process is ready. This removes the need for “buffer management” and the various errors caused by it. Furthermore, a notify call can be made if the receiving message is not prepared yet since notify calls do not block. It notifies the receiving process of next time it should receive a message. All system processes have bitmaps to keep track of pending notifications, with a unique bit for each system process. When a sender process sends a notification to another process, the bit that is associated with the sender process in the receiving process’s bitmap is set and the kernel adds a timestamp

of when the notification was sent. Therefore, each process can keep track of the identity of processes that wish to send it messages, as well as when they intend to.

In some instances, additional information is added to the notification messages. For example, when a notification message is created to notify the receiving process about an interrupt, the message includes a bitmap of every possible source of interrupt. Moreover, when the system task sends a notification message, the message includes a bitmap of all the receiving process's pending signals. These bitmaps are located in kernel data structures, by knowing who sent the notification, the receiving process is able to tell which additional information is included, if any at all, and how to interpret it.

2.3 MINIX 3 Process Scheduling

The process scheduling algorithm in MINIX 3 is a multi-priority round robin. Therefore, it heavily relies upon the interrupt system. Processes are designed to block after they make input/output requests so that other processes have the ability to execute. When the request is granted, the current running process is interrupted and the original process is allowed to continue execution. Interrupts are also carried out by the clock to prevent a process from hoarding the CPU forever. Clock interrupts force running processes to stop after a certain period of time to allow other processes to also get a chance to execute. Software can generate a type of interrupts known as "traps" which have the same effect as hardware interrupts. The bottommost layer of MINIX 3 turns theses interrupts to messages that are sent to processes.

After an interrupt takes place or after a process terminates on its own, the next process to execute must be chosen. The MINIX 3 scheduler uses multilevel queueing to determine the process most deserving of the chance to execute. Sixteen queues are defined in the MINIX 3 scheduler. The queue with the lowest priority is reserved for usage by the IDLE process (process that executes when there is nothing to do) only. The clock and system tasks are scheduled in the queue with the highest priority. Next in line are drivers which are scheduled in queues with priority higher than the queues that servers are scheduled in, but lower than the highest priority queue. User processes are scheduled in a queue with priority significantly higher than the queue reserved for the IDLE process, but lower than the queues where servers are scheduled. The

system can move processes to different priority queues. Users also have the ability to do so (although limited) by using the nice command.

Another key aspect of how MINIX 3 schedules processes is by utilizing a quantum, which is a specific time interval allocated to processes throughout which they can execute. If a process is not done executing before the quantum ends, it is preempted and put at the end of its queue. Different process types are given different quanta; user processes have a low quantum compared to drivers and servers which are given a very large quantum. Essential drivers and servers are allocated very large quanta so that they are never normally preempted; however, MINIX 3 provides the ability for them to be preempted to prevent the system from being stuck if a malfunction occurs.

If a high priority process is stuck in a loop and preventing lower priority processes from executing, it is placed at the end of a lower priority queue (its priority is lowered) to give another process the chance to run. On the other hand, a process can be promoted to a higher priority queue (within its allowed limits) if it uses up all its quantum but does not prevent other processes from running.

If a process blocks after an input/output request and therefore does not use up its entire quantum, it will be placed on the head of the queue when it becomes ready; however, this is only for the remaining duration of its interrupted quantum.

When it is time for the scheduler to pick the next process to execute, it checks the highest priority queue first; if it contains any queued processes, the one at the head to chosen to execute. If not, the scheduler checks the second highest priority queue, and so on until the lowest priority queue is reached and the IDLE process runs.

2.4 Testing

2.4.1 MINIX Scheduling Policies

Background Information:

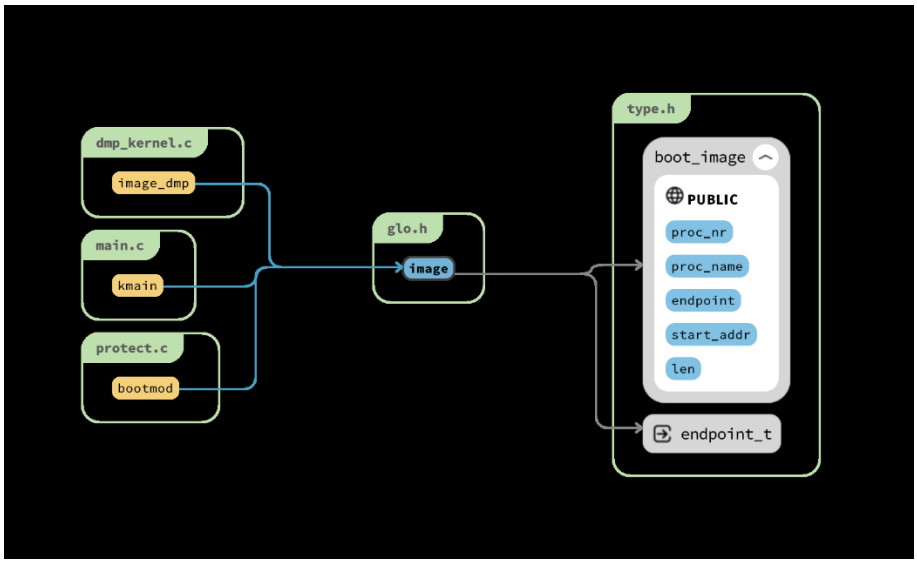
- MINIX 3 provides an interface by which user can modify the policies of scheduler.
- Through the following system calls: sys_schedctl and sys_schedule.
- They send messages to kernel requesting to start scheduling a particular process and to give a particular process quantum and priority, respectively.
- Once a process runs out of its quantum, the user mode scheduler is notified with a message.
- This message contains system and process feedback that the scheduler may use for making scheduling decisions.

Scheduler implementation (sched):

- PM has its own process table which is the user mode counterpart of the in-kernel process table. PM's process table has two attributes related to scheduling: mp_nice and mp_scheduler.
- PM is the only user mode process that knows who is scheduled by whom. This information is stored in mp_scheduler which points to the scheduler's/or KERNEL endpoint. Neither it is scheduled by user-apace-scheduler or kernel scheduler.
- As PM is the only user mode process that knows who is scheduled by whom, operations aecting scheduling are routed through PM. An example of this would be changing a process' niceness.

After Boot:

- When the system is booted the kernel's main() function will populate its process table from the boot_image struct as described




```
115  void kmain(kinfo_t *local_cbi)
116  {
117  /* Start the ball rolling. */
118    struct boot_image *ip;     /* boot image pointer */
```

```
44   struct boot_image image[NR_BOOT_PROCS] = {
45   /* process nr, name */
46   {ASYNCM,        "asyncm"},
47   {IDLE,          "idle"  },
48   {CLOCK,         "clock" },
49   {SYSTEM,        "system"},
50   {HARDWARE,      "kernel"},
51
52   {DS_PROC_NR,    "ds"    },
53   {RS_PROC_NR,    "rs"    },
54
55   {PM_PROC_NR,    "pm"    },
56   {SCHED_PROC_NR, "sched" },
57   {VFS_PROC_NR,   "vfs"   },
58   {MEM_PROC_NR,   "memory"},
59   {TTY_PROC_NR,   "tty"   },
60   {MIB_PROC_NR,   "mib"   },
61   {VM_PROC_NR,    "vm"    },
62   {PFS_PROC_NR,   "pfs"   },
63   {MFS_PROC_NR,   "mfs"   },
64   {INIT_PROC_NR,  "init"  },
65   };
```

- In the kernel/main.c of the kernel, we find the following where an image of boot processes is initialized.

```
/* Set up proc table entries for processes in boot image. */
for (i=0; i < NR_BOOT_PROCS; ++i) {
  int schedulable_proc;
  proc_nr_t proc_nr;
  int ipc_to_m, kcalls;
  sys_map_t map;

  ip = &image[i];             /* process' attributes */
  DEBUGEXTRA(("initializing %s... ", ip->proc_name));
  rp = proc_addr(ip->proc_nr);        /* get process pointer */
  ip->endpoint = rp->p_endpoint;      /* ipc endpoint */
  rp->p_cpu_time_left = 0;
  if(i < NR_TASKS)            /* name (tasks only) */
      strlcpy(rp->p_name, ip->proc_name, sizeof(rp->p_name));

  if(i >= NR_TASKS) {
      /* Remember this so it can be passed to VM */
      multiboot_module_t *mb_mod = &kinfo.module_list[i - NR_TASKS];
      ip->start_addr = mb_mod->mod_start;
      ip->len = mb_mod->mod_end - mb_mod->mod_start;
  }
```

- Processes are given a fixed quantum but no scheduler, meaning that scheduling is handled by the kernel.
- Once the kernel switches to user mode the servers included in the boot image rs,ds,vfs,pfs, or sched- start initializing.
- The user mode scheduler does not know which processes are currently running in the system, and therefore does not pro-actively take over scheduling of any processes.
- It will go straight to receive incoming messages and wait for requests to start scheduling.
- PM populates its local process table from the same boot_image struct.

```
54
55      /* SEF local startup. */
56      sef_local_startup();
57
```

```
111  /*===========================================================================*
112   *                sef_local_startup                  *
113   *===========================================================================*/
114  static void
115  sef_local_startup(void)
116  {
117    /* Register init callbacks. */
118    sef_setcb_init_fresh(sef_cb_init_fresh);
119    sef_setcb_init_restart(SEF_CB_INIT_RESTART_STATEFUL);
120
121    /* Register signal callbacks. */
122    sef_setcb_signal_manager(process_ksig);
123
124    /* Let SEF perform startup. */
125    sef_startup();
126  }
127
128  /*===========================================================================*
129   *                  sef_cb_init_fresh                                *
130   *===========================================================================*/
131  static int sef_cb_init_fresh(int UNUSED(type), sef_init_info_t *UNUSED(info))
132  {
133  /* Initialize the process manager. */
134    int s;
135    static struct boot_image image[NR_BOOT_PROCS];
136    register struct boot_image *ip;
137    static char core_sigs[] = { SIGQUIT, SIGILL, SIGTRAP, SIGABRT,
138                  SIGEMT, SIGFPE, SIGBUS, SIGSEGV };
139    static char ign_sigs[] = { SIGCHLD, SIGWINCH, SIGCONT, SIGINFO };
140    static char noign_sigs[] = { SIGILL, SIGTRAP, SIGEMT, SIGFPE,
141                  SIGBUS, SIGSEGV };
142    register struct mproc *rmp;
143    register char *sig_ptr;
144    message mess;
145
146    /* Initialize process table, including timers. */
147    for (rmp=&mproc[0]; rmp<&mproc[NR_PROCS]; rmp++) {
148      init_timer(&rmp->mp_timer);
149      rmp->mp_magic = MP_MAGIC;
150      rmp->mp_sigact = mpsigact[rmp - mproc];
151      rmp->mp_eventsub = NO_EVENTSUB;
152    }
```

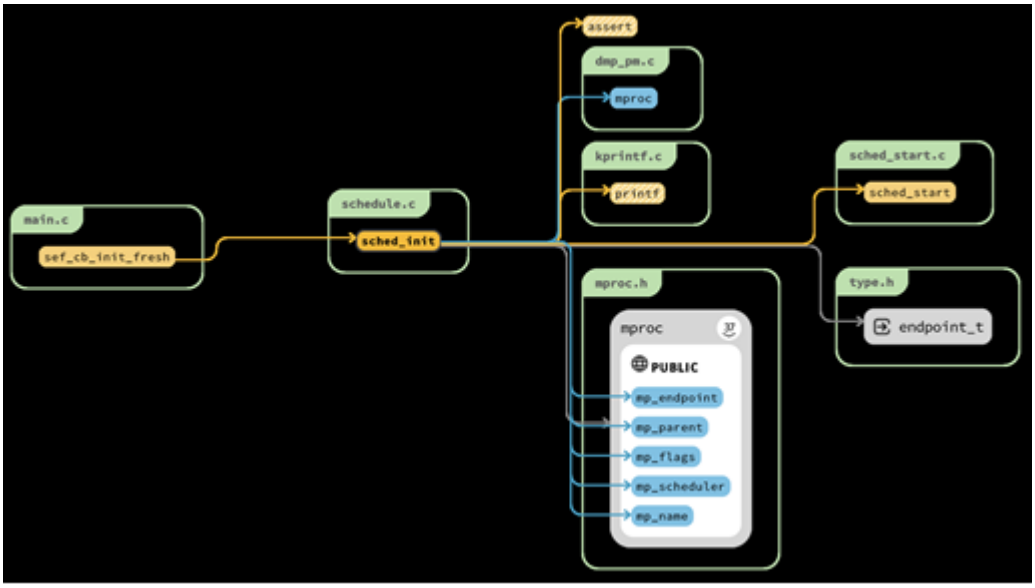
- At the end of its main() function it will send the user mode scheduler a request to start scheduling processes.

```
240     /* Initialize user-space scheduling. */
241     sched_init();
242
243     return(OK);
244   }
```

- At this time, init will be the the only non-privileged processes in PM's process table, and without quantum.
- This is the only process that PM requests the scheduler to schedule, but as all user processes are forked from init (or its children), they inherit init's assigned scheduler.

```
17  /*===========================================================================*
18   *              init_scheduling                  *
19   *===========================================================================*/
20  void sched_init(void)
21  {
22      struct mproc *trmp;
23      endpoint_t parent_e;
24      int proc_nr, s;
25
26      for (proc_nr=0, trmp=mproc; proc_nr < NR_PROCS; proc_nr++, trmp++) {
27          /* Don't take over system processes. When the system starts,
28           * init is blocked on RTS_NO_QUANTUM until PM assigns a
29           * scheduler, from which other. Given that all other user
30           * processes are forked from init and system processes are
31           * managed by RS, there should be no other process that needs
32           * to be assigned a scheduler here */
33          if (trmp->mp_flags & IN_USE && !(trmp->mp_flags & PRIV_PROC)) {
34              assert(_ENDPOINT_P(trmp->mp_endpoint) == INIT_PROC_NR);
35              parent_e = mproc[trmp->mp_parent].mp_endpoint;
36              assert(parent_e == trmp->mp_endpoint);
37              s = sched_start(SCHED_PROC_NR,  /* scheduler_e */
38                  trmp->mp_endpoint,  /* schedulee_e */
39                  parent_e,       /* parent_e */
40                  USER_Q,         /* maxprio */
41                  USER_QUANTUM,       /* quantum */
42                  -1,         /* don't change cpu */
43                  &trmp->mp_scheduler);   /* *newsched_e */
44              if (s != OK) {
45                  printf("PM: SCHED denied taking over scheduling of %s: %d\n",
46                      trmp->mp_name, s);
47              }
48          }
49      }
50  }
```

```
79      /* A user-space scheduler must schedule this process. */
80      memset(&m, 0, sizeof(m));
81      m.m_lsys_sched_scheduling_start.endpoint    = schedulee_e;
82      m.m_lsys_sched_scheduling_start.parent      = parent_e;
83      m.m_lsys_sched_scheduling_start.maxprio     = maxprio;
84      m.m_lsys_sched_scheduling_start.quantum     = quantum;
85
```



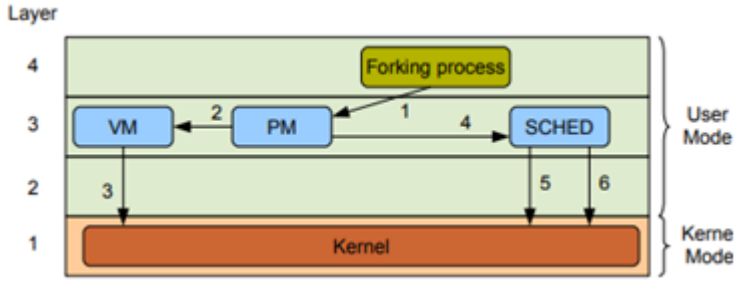
- RS will act similarly to PM, requesting the user mode scheduler to schedule device drivers and certain system servers.
- In theory the user mode scheduler can schedule any process, but care must be taken to avoid deadlocks.
- The scheduler should not schedule servers that itself depends on.
- The scheduler should not schedule itself, nor for example VM if it uses dynamic memory allocation.
- Messages sequence summary:
    1. PM and RS send a SCHEDULING_START start message to SCHED through the sched_start library routine.
    2. SCHED populates its local process table and send a message to kernel, notifying that it will take over scheduling the process.
    3. SCHED will make a scheduling decision based on its local policy and schedule the process for the first time, allocating it quantum and placing it in a given priority queue.

2.4.2 User Processes Lifecycle

Initialization:

- Previously, processes inherited their parent's priority and half their parent's quantum.

- This was revisited when moving scheduling to user mode. When a process is forked it is now spawned without quantum, is therefore not runnable and has no real priority.



- The process will not be placed in a run queue before a user mode scheduler has taken over scheduling the process and given it a priority and queue.
- In which priority the process is placed is entirely up to the user mode scheduler to decide. However, which ever policy gets implemented, it is often desirable for child processes to inherit not only their parent's base priority but also their current priority.
- The base priority denotes the general priority, essentially the highest priority this process can have at any given time. This value is local to the scheduler, is never exposed to kernel, and is typically affected by the nice level. The current priority denotes in which priority queue the process is currently scheduled.
- Messages sequence:

    1. POSIX program issues fork(). This sends a message to PM, which prepares its process table for the new process and forwards the message on to VM.
    2. VM prepares memory for the new process and forwards the message to kernel.
    3. The kernel forks the process, basing its process table entry on its parent entry. The parent process' quantum remains unchanged but the child is spawned without quantum (blocked with RTS_NO_QUANTUM flag).

```
58    /* Copy parent 'proc' struct to child. And reinitialize some fields. */
59    gen = _ENDPOINT_G(rpc->p_endpoint);
60  #if defined(__i386__)
61    old_fpu_save_area_p = rpc->p_seg.fpu_state;
62  #endif
63    *rpc = *rpp;                /* copy 'proc' struct */
```

```
..  do_fork
87      strcat(rpc->p_name, FORKSTR);
88
89    /* the child process is not runnable until it's scheduled. */
90    RTS_SET(rpc, RTS_NO_QUANTUM);
91    reset_proc_accounting(rpc);
92
93    rpc->p_cpu_time_left = 0;
```

4. At this point PM will asynchronously notify VFS of the new process and wait for its reply before moving on. This is left out of Figure 6 for the sake of clarity. Once VFS replies, PM will notify SCHED that a new process has been spawned and it needs to be scheduled.
5. SCHED will populate its local process table and send a message to kernel, taking over the scheduling of the process.

```
23  EXTERN struct schedproc {
24      endpoint_t endpoint;    /* process endpoint id */
25      endpoint_t parent;  /* parent endpoint id */
26      unsigned flags;     /* flag bits */
27
28      /* User space scheduling */
29      unsigned max_priority;  /* this process' highest allowed priority */
30      unsigned priority;      /* the process' current priority */
31      unsigned time_slice;        /* this process's time slice */
32      unsigned cpu;       /* what CPU is the process running on */
33      bitchunk_t cpu_mask[BITMAP_CHUNKS(CONFIG_MAX_CPUS)]; /* what CPUs is the
34                                  process allowed
35                                  to run on */
36  } schedproc[NR_PROCS];
```

```
216       /* Take over scheduling the process. The kernel reply message populates
217        * the processes current priority and its time slice */
218       if ((rv = sys_schedctl(0, rmp->endpoint, 0, 0, 0)) != OK) {
219           printf("Sched: Error taking over scheduling for %d, kernel said %d\n",
220               rmp->endpoint, rv);
221           return rv;
222       }
```

6. SCHED will make a scheduling decision based on its local policy and schedule the process for the first time, allocating it quantum and placing it in a given priority queue.

```
189     switch (m_ptr->m_type) {
190
191     case SCHEDULING_START:
192         /* We have a special case here for system processes, for which
193          * quanum and priority are set explicitly rather than inherited
194          * from the parent for drivers and init */
195         rmp->priority   = rmp->max_priority;
196         rmp->time_slice = m_ptr->m_lsys_sched_scheduling_start.quantum;
197         break;
198
199     case SCHEDULING_INHERIT:
200         /* Inherit current priority and time slice from parent. Since there
201          * is currently only one scheduler scheduling the whole system, this
202          * value is local and we assert that the parent endpoint is valid */
203         if ((rv = sched_isokendpt(m_ptr->m_lsys_sched_scheduling_start.parent,
204                 &parent_nr_n)) != OK)
205             return rv;
206
207         rmp->priority = schedproc[parent_nr_n].priority;
208         rmp->time_slice = schedproc[parent_nr_n].time_slice;
209         break;
210
211     default:
212         /* not reachable */
213         assert(0);
214     }
```

2.4.3 Different Scheduling Policies

PRIORITY:

- There is not a big difference between the current policy in MINIX 3.4 and priority algorithm all we need to transfer to priority is to cancel the FEEDBACK block of code.

```
97
98      rmp = &schedproc[proc_nr_n];
99      if (rmp->priority < MIN_USER_Q) {
100         rmp->priority += 1; /* lower priority */
101     }
102
```

- The priority of each process is predefined by user, using the method setpriority; either for built-in crucial processes, or for user processes.

```
39  int setpriority(int which, id_t who, int prio)
40  {
41      message m;
42
43      memset(&m, 0, sizeof(m));
44      m.m_lc_pm_priority.which = which;
45      m.m_lc_pm_priority.who = who;
46      m.m_lc_pm_priority.prio = prio;
47
48      return _syscall(PM_PROC_NR, PM_SETPRIORITY, &m);
49  }
```

ROUND ROBIN:

1.0) What is the change?

- We just modify the feedback in do_noquantum which change the priority by increasing it by one. if the process hadn't been finished yet and it needed more time slice from CPU.
- we will not change priority of crucial built-in processes like. VFS, or CLOCK; that will cause the system to get crashed or deadlocked, we will add feedback for user processes which work as FCFS-like; depending on the arrival time of each process.
- The system now will work in a priority-like manner, but not for all processes, only built-in processes which operate the system, but will work as FCFS-like manner for the none built-in processes like INIT, and its children.
- The change is to detect the end point at first if it is a none built-in process so we will give it a quantum until it will be finished.
- Else if it was a built-in then it will be scheduled upon the original feedback.
- So, any endpoint equal to 11 or greater will be considered as none bult-in.

```
41
42  /*===========================================================================*
43   *               Process numbers of processes in the system image            *
44   *===========================================================================*/
45
46  /* Kernel tasks. These all run in the same address space. */
47  #define ASYNCM  ((endpoint_t) -5) /* notifies about finished async sends */
48  #define IDLE    ((endpoint_t) -4) /* runs when no one else can run */
49  #define CLOCK   ((endpoint_t) -3) /* alarms and other clock functions */
50  #define SYSTEM  ((endpoint_t) -2) /* request system functionality */
51  #define KERNEL  ((endpoint_t) -1) /* pseudo-process for IPC and scheduling */
52  #define HARDWARE     KERNEL /* for hardware interrupt handlers */
53
54  /* Number of tasks. Note that NR_PROCS is defined in <minix/config.h>. */
55  #define MAX_NR_TASKS    1023
56  #define NR_TASKS      5
57
58  /* User-space processes, that is, device drivers, servers, and INIT. */
59  #define PM_PROC_NR   ((endpoint_t) 0)   /* process manager */
60  #define VFS_PROC_NR  ((endpoint_t) 1)   /* file system */
61  #define RS_PROC_NR   ((endpoint_t) 2)   /* reincarnation server */
62  #define MEM_PROC_NR  ((endpoint_t) 3)   /* memory driver (RAM disk, null, etc.) */
63  #define SCHED_PROC_NR ((endpoint_t) 4)  /* scheduler */
64  #define TTY_PROC_NR  ((endpoint_t) 5)   /* terminal (TTY) driver */
65  #define DS_PROC_NR   ((endpoint_t) 6)   /* data store server */
66  #define MIB_PROC_NR  ((endpoint_t) 7)   /* management info base service */
67  #define VM_PROC_NR   ((endpoint_t) 8)   /* memory server */
68  #define PFS_PROC_NR  ((endpoint_t) 9)  /* pipe filesystem */
69  #define MFS_PROC_NR  ((endpoint_t) 10)  /* minix root filesystem */
70  #define LAST_SPECIAL_PROC_NR    11  /* An untyped version for
71                                         computation in macros.*/
72  #define INIT_PROC_NR ((endpoint_t) LAST_SPECIAL_PROC_NR)  /* init
73                                                    -- goes multiuser */
```

2.0) The modified code:

The following figure 20 clarify the if condition which differentiate between built-in and none built-in processes by the end point, then according to that some feedback is changing priority by lowering it.

```
97  rmp = &schedproc[proc_nr_n];///assign the process object from element number "proc_nr_n" in the local process table "schedproc[]" by reference
98  if(rmp->endpoint<11)
99
100     {
101         if (rmp->priority < MIN_USER_Q) {/// still the process priority does n't exceed 15 which is the minimum priority.
102             rmp->priority += 1; /* lower priority */
103         }
104
105
106     }
107 else
108 {
109     /// no thing to do just give the process whatever quantum it asked for.
110 }
111 if ((rv = schedule_process_local(rmp)) != OK) {
112             return rv;
113         }
114         return OK;
115
116 }
```

3.0) We shall add printf function for debugging causes that mechanism is known as log messages debugging through pipe com1.

```
331
332     if ((err = sys_schedule(rmp->endpoint, new_prio,
333         new_quantum, new_cpu, niced)) != OK) {
334         printf("PM: An error occurred when trying to schedule %d: %d\n",
335         rmp->endpoint, err);
336     }
337     else(!err)
338     {
339         printf("Process %d consumed Quantum %d and Priority %d\n", rmp->endpoint,rmp->time_slice rmp->priority);
340     }
341
342
```

MFQ:

1.0) What is the change?

```
/* Max. number of I/O ranges that can be assigned to a process */
#define NR_IO_RANGE 64

/* Max. number of device memory ranges that can be assigned to a process */
#define NR_MEM_RANGE    20

/* Max. number of IRQs that can be assigned to a process */
#define NR_IRQ  8

/* Scheduling priorities. Values must start at zero (highest
 * priority) and increment.
 */
#define NR_SCHED_QUEUES   19    /* MUST equal minimum priority + 1 */
#define TASK_Q          0    /* highest, used for kernel tasks */
#define MAX_USER_Q         19    /* highest priority for user processes */
#define USER_Q        ((MIN_USER_Q - MAX_USER_Q) / 2 + MAX_USER_Q) /* default
                        (should correspond to nice 0) */
#define MIN_USER_Q    (NR_SCHED_QUEUES - 1) /* minimum priority for user
                           processes */
/* default scheduling quanta */
#define USER_QUANTUM 200

/* defualt user process cpu */
#define USER_DEFAULT_CPU    -1 /* use the default cpu or do not change the
                      current one */
```

- There were several modifications made in the schedule.c in the directory /usr/src/servers/schedule.c.
- do_noquantum( ): For processes between queue 16 and 18, increase their priority by 1, as soon as their quantum expires.(line 102)
- For processes in queue 18, set the new priority as 16.(line
- For all other processes, use default policy.(line 118)

```
int do_noquantum(message *m_ptr)
{
    register struct schedproc *rmp;
    int rv, proc_nr_n;

    if (sched_isokendpt(m_ptr->m_source, &proc_nr_n) != OK) {
        printf("SCHED: WARNING: got an invalid endpoint in OOQ msg %u.\n",
        m_ptr->m_source);
        return EBADEPT;
    }

    rmp = &schedproc[proc_nr_n];
    if(rmp->priority>=MAX_USER_Q && rmp->priority<=MIN_USER_Q){
        rmp->quantum+=1;
        if(rmp->quantum==5){
            printf("Process %d consumed Quantum 5 and Priority %d\n", rmp->endpoint, rmp->priority);
            rmp->priority = USER_Q;
        }
        else if(rmp->quantum==15) {
            printf("Process %d consumed Quantum 10 and Priority %d\n", rmp->endpoint, rmp->priority);
            rmp->priority = MIN_USER_Q;
        }
        else if(rmp->quantum==35) {
            printf("Process %d consumed Quantum 20 and Priority %d\n", rmp->endpoint, rmp->priority);
            rmp->quantum = 0;
            rmp->priority = MAX_USER_Q;
        }
    }
    else if (rmp->priority < MAX_USER_Q-1) {
        rmp->priority += 1; /* lower priority */
    }

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

- do_start_scheduling( ): In this function we mainly initialize the priority, max_priority, quantum, and time_slice. Changing the value's when required.
- do_nice( ): If we cannot assign new priorities to the jobs in queues 16 -18, we assign previous priority value's. In case of an error we roll back.

```
272   *             do_nice                       *
273   *===========================================================================*/
274  int do_nice(message *m_ptr)
275  {
276      struct schedproc *rmp;
277      int rv;
278      int proc_nr_n;
279      unsigned new_q, old_q, old_max_q, old_quantum;
280
281      /* check who can send you requests */
282      if (!accept_message(m_ptr))
283          return EPERM;
284
285      if (sched_isokendpt(m_ptr->SCHEDULING_ENDPOINT, &proc_nr_n) != OK) {
286          printf("SCHED: WARNING: got an invalid endpoint in OOQ msg "
287          "%ld\n", m_ptr->SCHEDULING_ENDPOINT);
288          return EBADEPT;
289      }
290
291      rmp = &schedproc[proc_nr_n];
292      new_q = (unsigned) m_ptr->SCHEDULING_MAXPRIO;
293      if (new_q >= NR_SCHED_QUEUES) {
294          return EINVAL;
295      }
296
297      /* Store old values, in case we need to roll back the changes */
298      old_q     = rmp->priority;
299      old_max_q = rmp->max_priority;
300      old_quantum = rmp->quantum;
301
302      /* Update the proc entry and reschedule the process */
303      rmp->max_priority = rmp->priority = new_q;
304
305      if ((rv = schedule_process_local(rmp)) != OK) {
306          /* Something went wrong when rescheduling the process, roll
307           * back the changes to proc struct */
308          rmp->priority     = old_q;
309          rmp->max_priority = old_max_q;
310          rmp->quantum = old_quantum;
311      }
312
313      return rv;
314  }
```

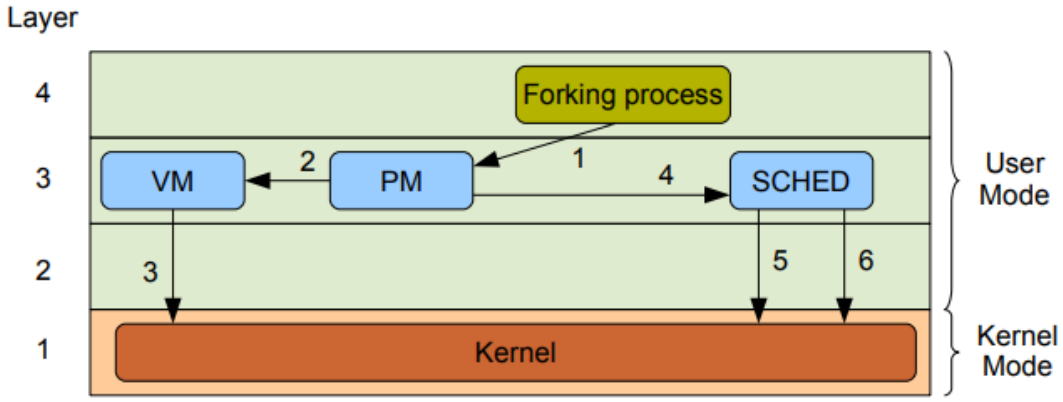
- balance_queues( ): increase the priorities of all the jobs by one.

2.0) Test Snaps

```
# ./test
Process id: 166
Process 36704 consumed Quantum 5 and Priority 16
Process 36704 consumed Quantum 5 and Priority 16
Process 36704 consumed Quantum 10 and Priority 17
Process 36704 consumed Quantum 20 and Priority 18
Process 36704 consumed Quantum 5 and Priority 16
Process 36704 consumed Quantum 5 and Priority 16
Process 36704 consumed Quantum 10 and Priority 17
Process 36704 consumed Quantum 20 and Priority 18
Process 36704 consumed Quantum 5 and Priority 16
Process 36704 consumed Quantum 5 and Priority 16
Process 36704 consumed Quantum 10 and Priority 17
Process 36704 consumed Quantum 20 and Priority 18
Process 36704 consumed Quantum 5 and Priority 16
Process 36704 consumed Quantum 5 and Priority 16
Process 36704 consumed Quantum 10 and Priority 17
```

SJF:

1. The hardest part in the algorithm is to calculate the process CPU time precisely.
2. So the algorithm was not applicable; which urges us to use another ways to evaluate the process.
3. One of the ways we used is by comparing the sizes of the processes -its width of memory-
4. The shortest in width is shortest in CPU time; because it executes less machine instructions.
5. However that, it was not precise enough; because the nature of process either it was I/O bound or CPU bound will indeed influence the time to execute; I/O operations depend on readiness of the user, or device and their state -available or not- and more factors are not controllable.



6. Due to the synchronization between messages from PM to SCHED and VM we can't use the size of process because it is not initialized yet.
7. So we differentiate between I/O bound and CPU bound by the synchronous IPC in which the process is blocking waiting for some input from user or device.
8. The attribute is named m_krn_lsys_scedule.acnt_ipc_sync in each message
9. Depending on the number of ipc calls the priority and time slice are calculated.
10. The I/O bound need small time slice and high priority due to its realtime nature.
11. Where CPU bound need long time slices and can wait in lower priorities than I/O bound.
12. By using the IPC number we calculate burst time by the following formula.figure 26
13. Burst is the average amount of work in milliseconds between IPC calls.

$$burst = \frac{process\ quantum}{number\ of\ IPC\ calls}$$

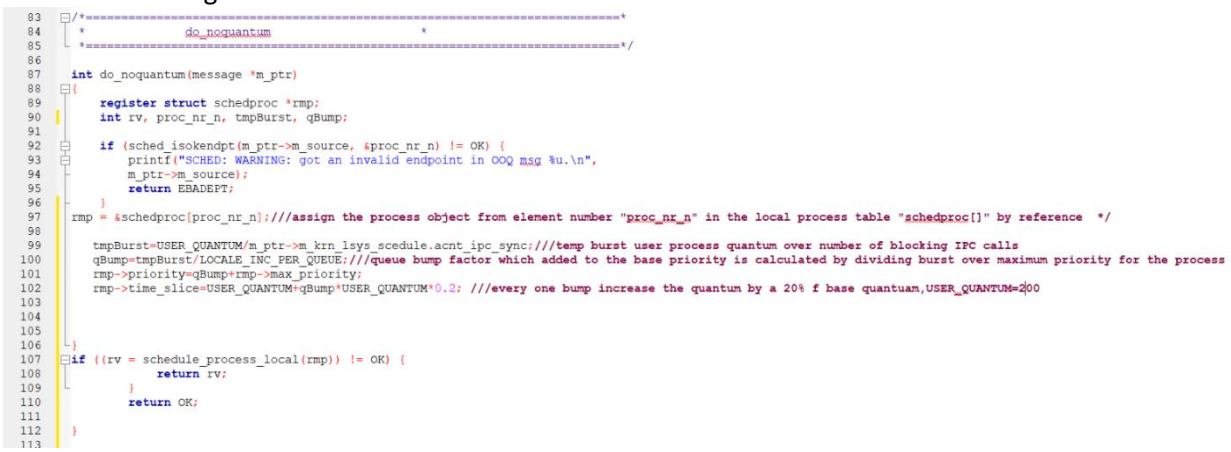
14. we calculate a moving average based on the last 10 samples, and prioritize the process based on this value.
15. based on the last 10 samples, and prioritize the process based on this value. Each process has both a base priority and base quantum.
16. These define the process' highest priority and the quantum it should have at that priority.
17. For each 10 ms of burst (INC_PER_QUEUE) we penalize the process by lowering its priority by one. When lowering priority, we also increase the quantum by 20%.

$$burst = \frac{process\ quantum}{number\ of\ IPC\ calls} = \frac{200\,\mathrm{ms}}{6} = 33,34\,\mathrm{ms}$$

18. we show a process p with a base priority of 2 and base quantum of 200 ms. When p runs out of quantum the scheduler will inspect the number of IPC calls and calculate its burst.
19. And we use the burst to calculate the queue bump factor as a priority penalty.

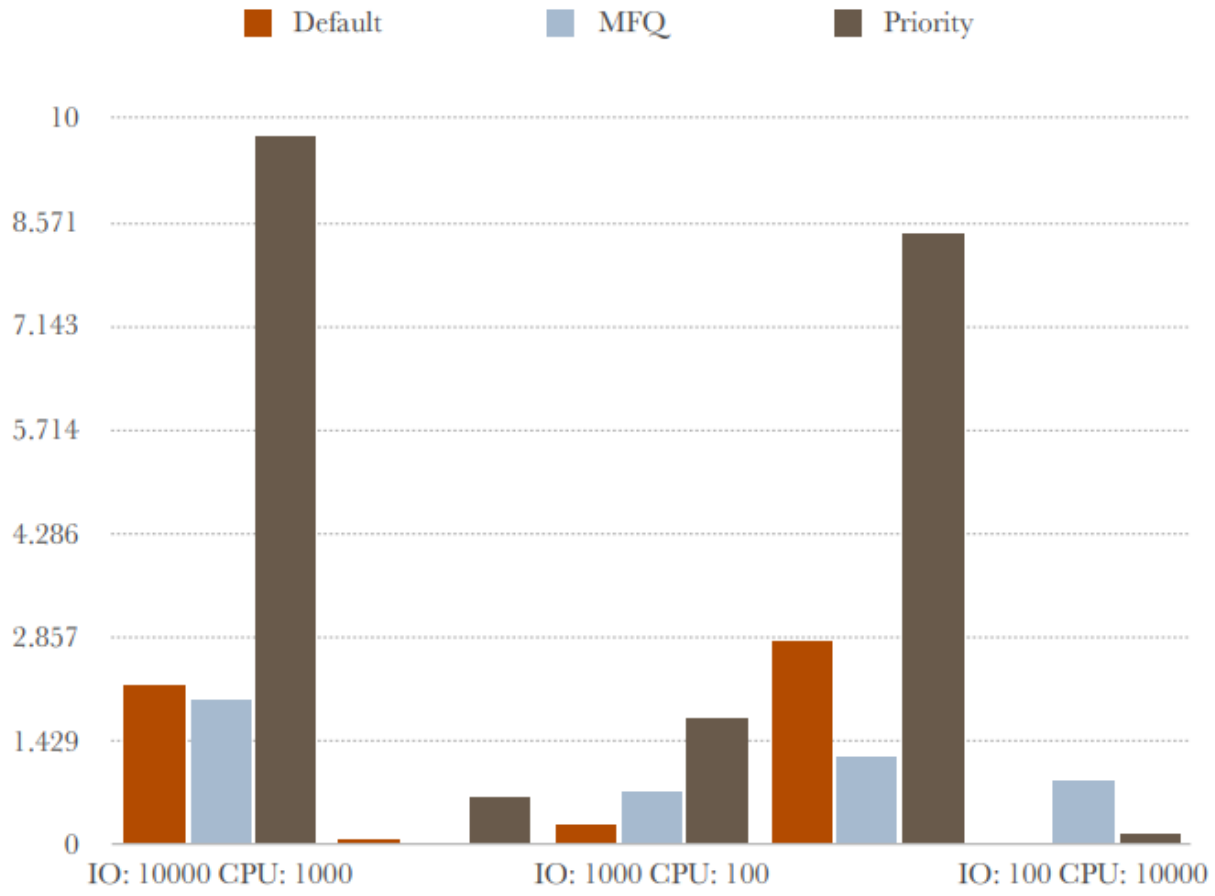
$$queue\ bump = \left\lfloor \frac{burst}{INC\_PER\_QUEUE} \right\rfloor = \left\lfloor \frac{33,34\,\text{ms}}{10\,\text{ms}} \right\rfloor = 3$$

20. we increase the quantum by 20% of the base quantum.
21. Then current priority = base priority + queue bump = 2 + 3 = 5
22. And then current quantum = 200 ms + 3 · 200 ms · 20% = 320 ms
23. After these estimations, the scheduler will schedule p in priority queue 5 with a quantum of 320 ms.
24. What is the change?



```
/*===========================================================================*
 *              do_noquantum                 *
 *===========================================================================*/

int do_noquantum(message *m_ptr)
{
    register struct schedproc *rmp;
    int rv, proc_nr_n, tmpBurst, qBump;

    if (sched_isokendpt(m_ptr->m_source, &proc_nr_n) != OK) {
        printf("SCHED: WARNING: got an invalid endpoint in OOQ msg %u.\n",
        m_ptr->m_source);
        return EBADEPT;
    }
rmp = &schedproc[proc_nr_n];///assign the process object from element number "proc_nr_n" in the local process table "schedproc[]" by reference  */

   tmpBurst=USER_QUANTUM/m_ptr->m_krn_lsys_scedule.acnt_ipc_sync;///temp burst user process quantum over number of blocking IPC calls
   qBump=tmpBurst/LOCALE_INC_PER_QUEUE;///queue bump factor which added to the base priority is calculated by dividing burst over maximum priority for the process
   rmp->priority=qBump+rmp->max_priority;
   rmp->time_slice=USER_QUANTUM+qBump*USER_QUANTUM*0.2; ///every one bump increase the quantum by a 20% f base quantuam,USER_QUANTUM=200



}
if ((rv = schedule_process_local(rmp)) != OK) {
            return rv;
        }
        return OK;

}
```

2.4.4 Comparison



As expected, on an average the turnaround time for the original scheduler was the least. This may be due the fact that this scheduler can set higher priorities than the ones that can be set in the user mode. It is interesting to see that the MFQ worked almost as well as the default. This is because we allowed a job to increase its priorities after certain iterations, i.e. when its quantum got over in the bottommost queue and it was pushed to the topmost queue with an increase in priority. The priority scheduler had the worst turn around times amongst all of them. This may be due to the fact that a number of shorter jobs with relatively low priority would have to wait for the jobs which are longer but have higher priority. All this would increase the average turn around time.

3.0) Requirement Three

3.1 Memory management

The memory manager is mostly used to figure out which parts are currently in use, and which are not. It is used to allocate and deallocate memory and swap between memory and disk in MINIX 3, not in the kernel.

Memory management systems are divided into two basic classes: 1- Systems that move processes between memory and hard disk (Swapping and Paging), 2- Systems that do not. The second type of systems, which does not use swapping and paging, is simpler.

Swapping and paging are mainly used when the main memory cannot hold all processes and their data at once.

3.1.1 Paging

Paging is a scheme of memory management, which ends and eliminates the need for contiguous memory allocation of the physical memory, which prevents the physical address from being contiguous as it permits the physical address of processes to be non-contiguous.

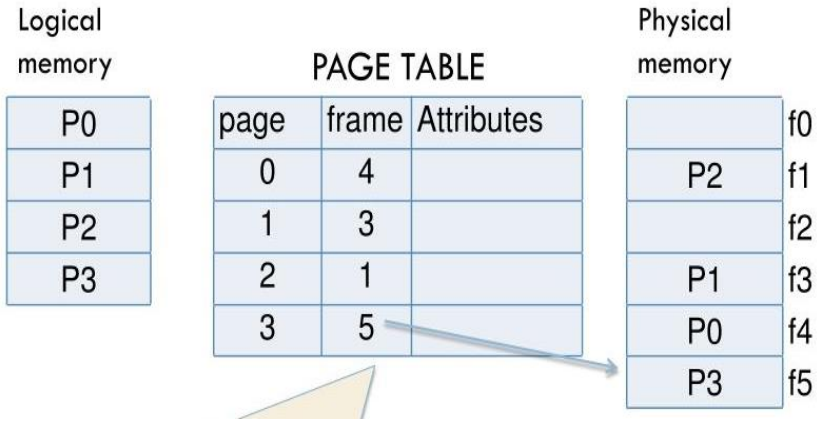
- Memory Management Unit(MMU) divides physical memory into frames and manages logical memory to pages, while the CPU is responsible for creating the logical memory.
- Size of the frame must be equal to the size of page (0.5 to 16) MB.
- Pages can fit any frame.
- Physical address space of a process can be noncontiguous.
- The executable part of the process is stored in the memory, while its remaining parts are stored in the backing store.
- Local/Virtual/Logical address is generated by the CPU.
- The physical address is the address on the actual memory chip
- Each logical address must correspond to an existing physical address, or an error might occur

The address is divided into two parts: 1- **Page number**, indicates which page the memory needs from the program; 2- **Page offset**, indicates the location inside the page.

If the Logical address is of size 2^m and pages are of size 2^n, then the page address needs M-n bits, and page offset is n bits.

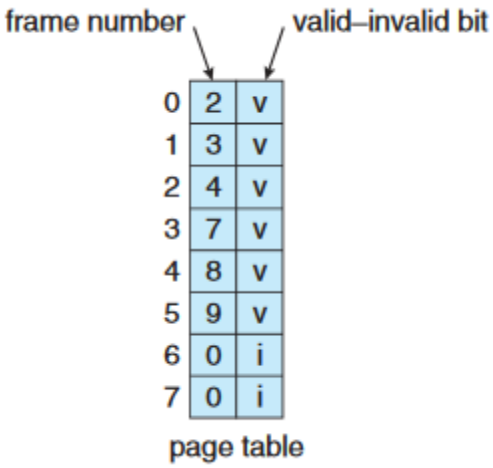
3.1.2 Page table

- Correlates each page to a frame
    - Bits of the frame number is not equal to bits of the page number.



Implementation:

- Page table base register points to the page table
- Page table length register indicates the size of the table
- Valid and invalid bits in the page table indicate the validity of every page in the page frame



Shared Pages:

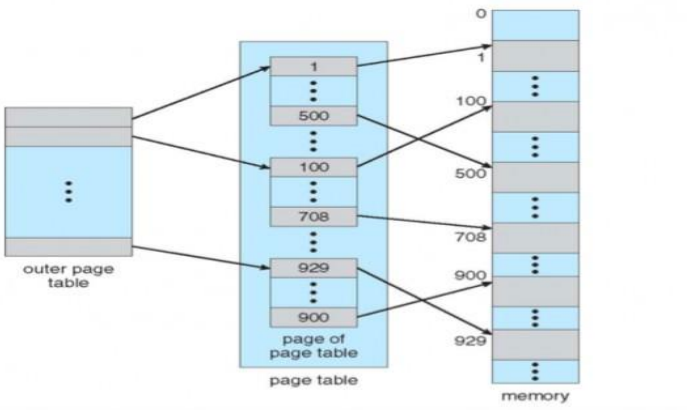
If many processes share the same page, we can allocate these pages in the same frame.

Structure of Page Table:

Considering that we have 32-bits for logical addresses and a page size of 4kb, we need offset to indicate the location inside the page, so 2^12 bits are used; thus, there are 20 remaining bits. We need a page table of 2^20. This will take up so much memory just to load the page table, which will be very inefficient, thus the use of **hierarchical page tables** will solve this problem.

Simply put, the hierarchical page table divides one level of paging into multiple levels.

In two-level paging, the address will be divided into three parts: 1-P1, 2-P2, 3-offset



As levels of hierarchy increase, **overhead to accessing memory substantially increases.**

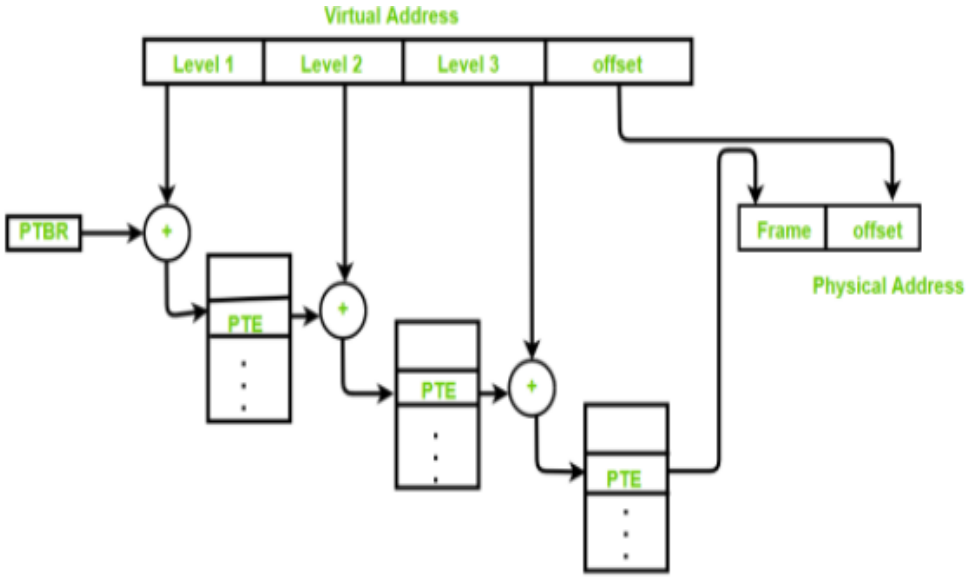
Thus, having too many levels is not efficient either and creates a new problem.

Hierarchal Paging:

A paging scheme in which each page in the outer levels of page tables contains page tables of inner levels, which is put in levels as the entries of each level points to the following level as if we have three levels the entries of the first level will point to the second level and the entries of the second level will point to the third level.

And the first level must contain a single page table and it is stored in what is called page table base register

Here all the pages will be saved in the main memory so more than one memory is required as one memory to access the physical address of the page frame and one memory to access each level needed as each table entry except the last page table entry will contain the base address of the next page table level



3 Level paging system

The levels of the paging and how to get it

There is a rule to get that number of the levels

Number of entries = (virtual address space size) / (page size)

= Number of pages

Virtual address space size: = 2^n B

Size of page table = (number of entries in page table) *(size of PTE)

Frames of the Memory are Full:

When there are no free frames available, we need page replacement, Search for memory pages but not really in use, and page it out.

Since the page size is between 0.5 to 16 MB so it may to a lot of time to reallocate this page to the memory.

So, to avoid this problem, we will use another bit in the page table called a modified bit (Dirty bit) which indicates whether the page is modified or not

If the page is modified (Dirty bit is 1) we will need to reallocate it to the memory if it doesn't (Dirty bit is 0), so we will leave this page at the backing store.

Page Fault:

Page fault is important in page replacement. Page fault happens when there is a new page requested by the program and this page is not found in the main memory.

Page fault sends an alert for the operating system when the operating system retrieves this alert it gets the page from the virtual memory and transfers it to the main memory and all the processes are now run in the background. This operation takes milliseconds but has a huge impact on the operating system so high numbers of page faults may slow the operating system or cause the program to crash or terminate.

3.2 Memory management in MINIX 3

Memory management in MINIX 3.1 is simple it doesn't use paging at all but after this version, MINIX 3 started to use paging.

Process manager allocates memory using a principle called first fit in which it never changes it.

The process manager tracks the memory to assign it to the processes, as the memory mapping is done by the kernel

the memory is allocated by two system calls the fork system call and the exec system call and this is done by the process manager.

It has two modes (the choice of the model depends on the program itself)

the first mode: that the code and the data are in the same segment or the code in a segment that can be used by processes

the second mode: the exec system calls searches for the code to determine whether it is loaded and can be shared with the processes or not

the Minix Memory Allocation

there is what is called a hole table and it is in Minix in include/Minix/type. has it tracks the empty available memory that can be used and the memory which is already allocated will remain tracked in the process table of the process manager as every entry points to the following entry and also points to the hole length and base

there is an algorithm which is called first-fit allocation and it is implemented in the file

servers/pm/alloc.c

the freeing of the memory has more complexity due to more than one reason:

1. The holes may not be available in the descriptors (as Minix store the hole descriptors away from the holes themselves)
2. The free list must be in sorted order, so any block is inserted in a correct position
3. The new memory which is freed may be adjacent to block which is free so they must be merged as the merging is done by seeing that the block whether is adjacent to the one before it or not and if they are adjacent then merging of them and the operation is repeated to the next operations

Process and Memory Management:

Memory management is done by process management as the processes is what allocates memory

In Minix versions before 3.1, the memory management is merged with the process management but the functionalities that support the memory management has evolved over time

And here we deduce that the process is a package of (stack, text, heap) segments

As every segment is allocated contiguously in the RAM so the memory will act as a collection of holes and allocated segments

Before minix 3.1 the paging, virtual memory, and demand paging weren't supported due to:

Developed techniques of memory management that need support by the hardware so the Minix 3.1 is for older CPUs that doesn't use or support paging

But after Minix 3.2 the paging, virtual memory, and demand paging and also that virtual memory is a new one this memory server is separated from the memory management

After minix 3.2 Minix must use segmented paging with the usage of the virtual memory

Virtual Memory Server:

As the virtual memory controls the memory (as it tracks all the memory the used and the unused one and assigns memory to processes and frees it)

In the virtual machine code in the virtual region, there is a division between the architecture-dependent and independent code of the virtual machine

The virtual region is the region in which there is a range of virtual address space that has a specific type and parameters and it has a static array of pointers to physical memory placed in them

Each entry is a memory block with a size of the page this block is initialized and points to the physical region, as the block of memory is described by it.

the physical region is what refers to the physical block and the physical page of memory is described by this block also

Call Structure:

The system calls of the userspace go to the virtual memory to allocate memory for heaps

The process manager with the system call (fork exit)

And the kernel control is

1. receive a message in main.c

2. call of certain job in a certain file, e.g. map.c, cache.c

3. update of the data structures and the page table

Handling Absent Memory:

Sometimes the process tries to use a memory address or a page that is not in the memory here the usage of page fault is necessary and the virtual memory can control the absent memory by two conditions

1. That the page fault is anonymous memory
2. Here the page fault will be found in the file mapped region in this condition the virtual machine will query cache else go to the virtual file system

Process Manager Server

The text segments can be shared among the processes by the process manager so the operating system will get that text segment only once

There is a part in process manager that handles memory management we can refer to this part as it is the “memory manager”. Memory management and process management are not separated in MINIX 3 where the functions are merged into one process.

The process manager handles system calls related to process management some of these calls are specified to memory management such as fork, exec, brk.

Memory is allocated in MINIX 3 on one of the two occasions. If the process forks the child allocate an amount of memory or when a process changes its memory image via an exec system call. The old memory image will be returned as a whole. The new image may be allocated into different allocations from the old image. This location will be determined when the new image finds an acceptable hole. The process deallocates the memory when it terminates or being killed by a signal or exiting.

System calls related to memory in MINIX 3

System Call	Input parameters	Reply value
Fork	(none)	Child's PID, (to child: 0)
exit	Exit status	(No reply if successful)
wait	(none)	Status
Waitpid	Process identifier and flags	Status
brk	New size	New size
exec	Pointer to initial stack	(No reply if successful)
kill	Process identifier and signal	Status
alarm	Number of seconds to wait	Residual time

pause	(none)	(No reply if successful)
sigaction	Signal number, action, old action	Status
sigsuspend	Signal mask	(No reply if successful)
sigpending	(none)	Status
sigprocmask	How, set, old set	Status
Sigreturn	Context	Status

The required tasks in this requirement are to implement hierarchal paging and different page replacement algorithms (LRU and FIFO Algorithms). Since MINIX 3 memory management does not use paging at all so we need to discuss in detail what is paging and discuss everything linked to it to complete these tasks.

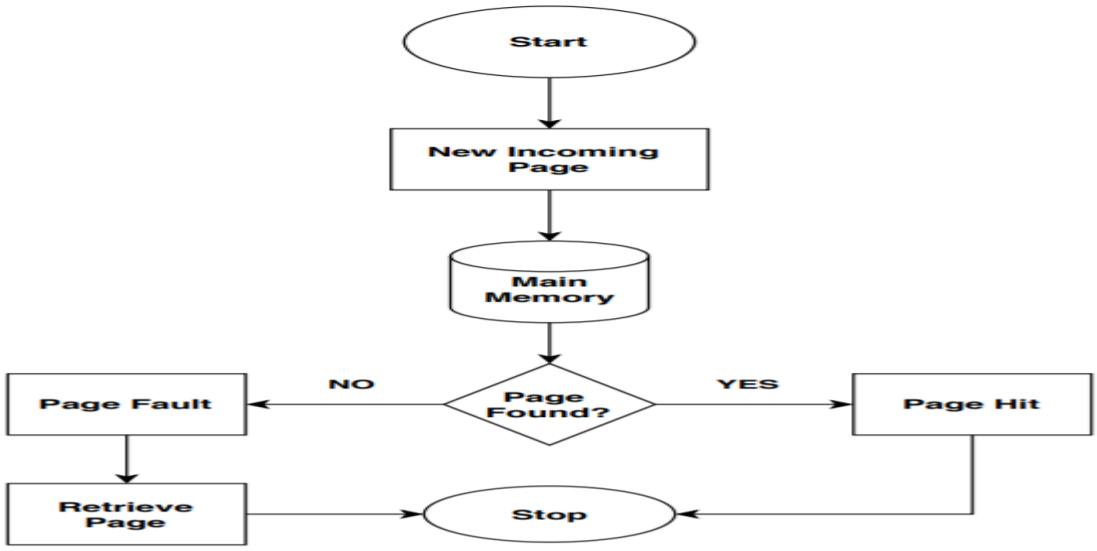
Page Replacement algorithms

Page replacement algorithm is a technique used for paging. Paging is mainly used for virtual memory management. It is used when there is a new page request and there is not enough space in the memory.

So, the page replacement algorithm decides which page it should replace.

Steps required to make page replacement:

- We firstly need a Frame allocation algorithm.
- Determine how many frames to each process which frames to replace.
- Need lowest page fault.
- String is just page numbers, not full addresses.
  -Repeated access to the same page does not cause a page fault.
- Here is a flowchart discussing the page replacement algorithm.



The effect of the algorithm is measured by the total number of page faults that occurred. The more effective algorithm has the least number of page faults generated by the algorithm itself.

Basic Page Replacement

1. Find the location of the desired page on disk.

2. Find a free frame:

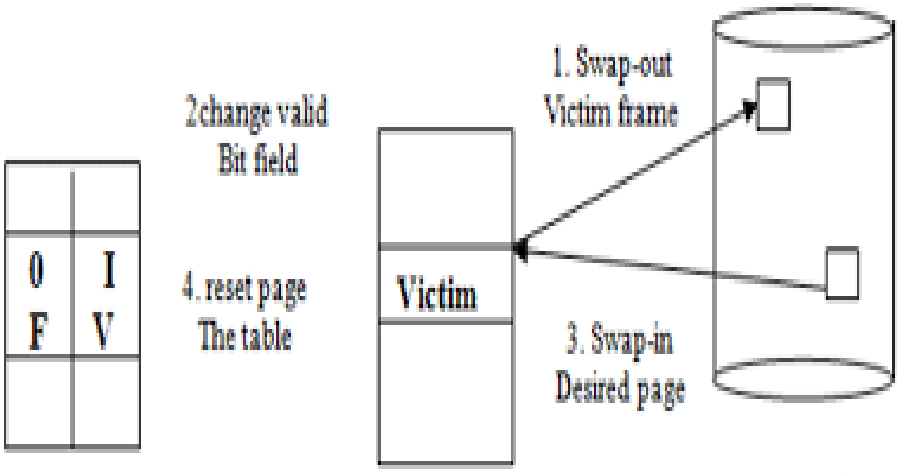
-If there is a free frame, use it.

-If there is no free frame, use a page replacement algorithm to select a victim frame.

-Write victim frame to disk if dirty.

3. Bring the desired page into the (newly) free frame; update the page table

4.Continue the process by restarting the instruction that caused the trap (page fault)



First-In-First-Out (FIFO) Algorithm

It is the simplest page replacement algorithm. The operating system tracks all pages in the queue of the memory, The oldest page will be in the front of the queue. When the page is needed to be replaced the page in front of the queue will be removed.

Here is the pseudocode for First-In-First-Out (FIFO) Algorithm

Algorithm 1: FIFO Page Replacement Algorithm Pseudocode

Data: Pages *P,* Number of Pages *N,* Capacity *C* **Result:** Number of page fault **PF**

```
Function FindPageFault(P, N, C) S = set();

QPage = Queue();

PF = 0;

for k = 0 to length(N) do

    if length(S) < C then

        if Plk] not in S then

            S.add(P1k1);

            PF = PF +1;

            QPage.put(P14); end

    else

        if Pik] not in S then

            val = QPage. queue/0J; QPage.get();

            S.remove(val);

            S.add(P1k1);

            QPage.put(114); PF = PF +1;

        End     return PF;
```

Illustration For pseudocode of First-In-First-Out (FIFO) Algorithm

1. **S** denotes for the current pages in the system, A queue **Q** is created to store the incoming pages in FIFO form.

3. The page faults **PF** initialized to zero.

4. When a new page is received we check for the capacity of **S** . We have three cases:

A-The Set **S** has free space to store these new page, The new page is successfully stored in the set.

B-When the set **S** has free space to store the new page, but the set already contains this page, in this case, we will mark page hit and the number of page faults won't increase.

C-When the set **S** is full, We will need to remove pages from the Set and replace them with new pages in the queue **Q** by using the FIFO algorithm. The algorithm returns the total page faults **PF** and then terminates.

Advantages and Disadvantages of FIFO Algorithm

- The main advantage of the FIFO Algorithm is its simplicity, where it is easy to be implemented as it uses a queue data structure.

- The Disadvantage of the FIFO Algorithm is that when the number of incoming pages is large, it might not provide excellent performance. As the capacity to store pages in queue increase, it should provide the least number of page faults but sometimes the FIFO behave abnormally and may increase the number of page faults. And another disadvantage of FIFO is that we should keep track of all frames which results in slow process execution.

Least Recently Used Algorithm

The LRU algorithm depends on observing the pages that have been used frequently in the last few instructions it predicts that these pages will be heavily used again in the next instructions and it also predicts that the pages that have not been used for a long time will remain unused for another long duration. It is a realizable algorithm when a page fault occurred it removes the unused page for the longest time.

Here is the pseudocode for Least Recently Used (LRU) Algorithm

Algorithm 2: LRU Page Replacement Algorithm Pseudocode

$T_v = (Fr)$

INPUT PAGES

FOR Pin TO Pin(n): P I+1

IF Frame = 0 THEN

Replace page in a frame

Else // search of page in memory

IF Pin = Pm THEN

Take the Next Page (P_m)

Else (P_{in} != P_m) //Page fault

Boolean Found = (T_v) [Last page and smallest frequently]

IF Found THEN Replace Page in Frame

V. P = Pm // victim pages

Remove Pm

Replace P_n

End for

End for

Illustration For pseudocode of Least Recently Used (LRU) Algorithm

1. **Tv** will be equal to **Fr** which represents the frequency of each page when it tries to enter the main memory. For example, if the incoming page is already in the memory and it tries to enter the main memory again the frequency of the page will increase.

2. We will loop on the queue which contains the incoming pages.

3. When we try to put the page in the main memory there will be three cases:

A. IF there is an empty frame that has the same size as the page we will put the page in this frame.

B. IF the incoming page **Pin** is already in the main memory **(Pin = Pm)**

We will get the next incoming page in the queue **(Pin)** and increment the frequency of this page.

C.IF the incoming page is not in the main memory **(Pin != Pm)**, then we will get into a condition to check the frequency of each page and the page which has the least frequency will be selected as the victim page **(V.P=Pm)** and this page will be removed and the incoming page (**Pin)** will replace it. IF there are two pages or more having the same frequency (Least frequency) the page that occurred first will be the victim page.

Modifications

The goal in this requirement is to implement hierarchical paging

Memory Allocation in MINIX concerning the source code:

MINIX combines both the memory allocation and CPU allocation data in the same structure called proc placed in /usr/src/kernel/proc.h while the actual memory allocation operations are placed in both files forkexit.c and exec.c which are both placed in /usr/src/servers/pm/.

Those two files contain the code that handles allocating the memory to the process as their inner functions are called in the two main scenarios where the processes need memory allocations, these scenarios are 1. When the process is first executed so the do_exec() function is called which is in file exec.c. 2. When a child process is forked from a parent executing process where the function do_fork() is called which is placed in file forkexit.c.

Implementing Hierarchical Paging in MINIX:

Single-level paging was firstly introduced to MINIX in version 3.1.4, We will implement two-level paging which is one of the most coinvent variances of multi-level paging and allow the creation of multiple levels also, To properly implement we will need to extend the page table implementation to support two-level page tables. The virtual address space of each process will be limited to 16MB and any virtual address referenced by a process is translated into a physical address using the two-level page table. So Minix now will be able to support 512 pages of 4KB each which can be further modified from the configuration file in /usr/src/include/Minix/config.h

So to wrap this up, The outer page tables will contain an array of pointers referring to a group of page tables where the whole group will be called the inner page table. By doing so, we will be able to successfully divide the big page table into smaller ones by keeping track of the locations of those smaller page tables in the actual memory. Lastly, all accessing functions are updated to allow the two-level page accessing by tracing down the memory allocation sequence.

Step #1: Creating a whole structure of the outer level table.

In /usr/src/servers/vm a new file will be created called outerpagetable.h that will contain the new structure of the outer page table.

/*

This whole file is a part of the implementation of Requirement #2 - Modification #1

Creating a structure that will contain an array of pointers of the inner page-page tables

and will be introduced to the kernel and the memory allocations functions instead of

the regular page table.

*/

//needed for OS-file-awareness

#define _SYSTEM 1

#include <stdlib.h>

#include <stdio.h>

#include "pagetable.h"

#include "util.h"

#include "vm.h"

#include <minix/config.h>

#include <minix/const.h>

//structure of the outer page-table containing an array of all inner page tables.

struct outer_pagetable {

  struct pagetable inner_pagetables[NR_SYS_CHUNKS]; /*NR_SYS_CHUNKS is pre-defined Minix variable from the config file that represents the total number of the memory block that can be made available to the processes.*/

};

Step #2: Create two variables in the page table to keep track of this start and end location in memory:

In /usr/src/servers/vm/pagetable.c at line 107 add the following two variables:

vir_bytes inner_pagetable_start;

vir_bytes inner_pagetable_end;

where they will be given proper values when allocating the memory in do_exec() and do_fork() to keep track of the locations of those smaller page-tables in the actual memory.

Step #3: Update Memory Allocation Functions to adapt with two-level paging:

In /usr/src/servers/vm/alloc.c:

Here is where all pages structures are defined with their reserve and control functions. To make them adapt with the new outer-level paging and allocation function are updated which are then automatically called whenever allocation is needed in do_exec() and do_fork() placed in /usr/src/servers/pm/exec.c and /usr/src/servers/pm/forkexit.c to act as a bridge connecting the two levels and also a new list of smaller-inner page tables is implemented to help keep tracking of everything.

At line 59, before the implementing or calling of any allocation functions, a static structure is implemented to act as a linked list holding all smaller-inner page tables addresses.

```
//Requirement #2 - Modification #3.

static struct smaller_inner_pagetable {

        struct pagetable *next;    /* next pointer to point on the next pagetable in the list */

        int max_available;         /* maximum number of pagetables supported */

        int number_of_pagetables;                  /* number of consecutive pagetables */

        int allocflags;            /* allocflags for alloc_mem pre-defined funciton that manage actual
allocation and mapping*/

} list_of_smaller_pagetables[MAXRESERVEDQUEUES]
```

At line 269, Update the alloc_mem() function to place the page-table in the new list_of_smaller_pagetables after allocating pages that are completely based on the original MINIX page-support code introduced in MINIX 3.1.4.

```
//Requirement #2 - Modification #4

page_table_update();

list_of_smaller_pagetables[number_of_pagetables++] = page_table_query(mem);
```

Step #4: Support Multiple Levels – more than two:

Add to file /usr/src/servers/vm/alloc.c the needed functions to allow multiple levels of page tables starting from line 559:

```
uint64_t *getOrCreateLevel(uint64_t **root) {
   if (*root == NULL) {
      *root = calloc(getTableSize(), sizeof(uint64_t));
   }
   return *root;
}

uint64_t *getOrCreateNextLevel(uint64_t *srcCell) {
   if (isValidAddress(*srcCell) == 0) {
      *srcCell = (alloc_page_frame() << 12u) | 0x1u;  // Add zero offset + valid bit
   }
   uint64_t **ptToTable = (uint64_t **) phys_to_virt(*srcCell);
   getOrCreateLevel(ptToTable);
   return *ptToTable;
}

uint64_t
handleLevel(uint64_t **curPagingTable, uint64_t curLevelBits, unsigned int readOnly) {
   uint64_t *curRoot = *curPagingTable + curLevelBits;
   if (readOnly == 1) {
      if (isValidAddress(*curRoot) == 0) return NO_MAPPING;  // If valid bit is zero
      else *curPagingTable = *(uint64_t **) (phys_to_virt(*curRoot));
```

```
    } else {
        *curPagingTable = getOrCreateNextLevel(curRoot);
    }
    return 0;
}

uint64_t walkAndUpdate(uint64_t *levelsRoot, uint64_t vpn, uint64_t ppn, unsigned int readOnly) {
    uint64_t *curPagingTable;
    uint64_t curLevelBits, *finalRoot, bitMask = getBitMask();
    curPagingTable = levelsRoot;
    for (int curLevel = 1; curLevel <= LEVELS_AMOUNT; curLevel++) {
        curLevelBits = (vpn >> ((LEVELS_AMOUNT - curLevel) * ENTRY_SIZE_BITS)) & bitMask;
        if (curLevel == LEVELS_AMOUNT) {
            finalRoot = curPagingTable + curLevelBits;
            if (readOnly == 1) return *finalRoot == 0 ? NO_MAPPING : *finalRoot;
            else *finalRoot = ppn == NO_MAPPING ? 0 : ppn;
        } else {
            if (handleLevel(&curPagingTable, curLevelBits, readOnly) == NO_MAPPING) return NO_MAPPING;
        }
    }
    return NO_MAPPING;
}

void page_table_update(uint64_t pt, uint64_t vpn, uint64_t ppn) {
    uint64_t *root = getOrCreateLevel((uint64_t **) phys_to_virt(pt));
    walkAndUpdate(root, vpn, ppn, 0);
}

uint64_t page_table_query(uint64_t pt, uint64_t vpn) {
    uint64_t **root = (uint64_t **) phys_to_virt(pt << 12u);
```

```
    if (root == NULL) return NO_MAPPING;

    if (*root == NULL) return NO_MAPPING;

    return walkAndUpdate(*root, vpn, -1, 1);

}
```

Step #5: Allow user to control the size of pages, and the address format from the configuration file:

1. User can change page size from /usr/src/servers/vm/arch/earm/pagetable.h at line 45 where he can change:

```
#define VM_PAGE_SIZE    ARM_PAGE_SIZE
```

To any preferred page size:

```
#define VM_PAGE_SIZE    200
```

2. User can also change the number of physical pages from /usr/src/servers/vm/alloc.c at line 33 and the maximum cache per page from line 36 and also the maximum reserved page number from line 57.

```
#define NUMBER_PHYSICAL_PAGES (int)(0x100000000ULL/VM_PAGE_SIZE)

#define PAGE_CACHE_MAX 10000

#define MAXRESERVEDPAGES       300
```

Step #6: Implement FIFO Page Replacement Algorithm in MINIX

As mentioned before, most memory management and memory allocation are completely managed by the process manager in MINIX, so to implement a FIFO page replacement algorithm, We made a new file called fifo.c and placed it in /usr/src/servers/pm to be called during the calling of do_exec() and do_fork() where the actual memory allocation is managed and potential page replacement may be needed.

```
/*

This file is responsible for the full implementation of the FIFO page replacement algorithm.

It is a part of the implementation of Requirement #2 - Modification #6.
```

```
*/

#include <stdio.h>
#include <stdlib.h>
#include "pm.h"
#include <sys/stat.h>
#include <minix/callnr.h>
#include <minix/endpoint.h>
#include <minix/com.h>
#include <minix/vm.h>
#include <signal.h>
#include <libexec.h>
#include <sys/ptrace.h>
#include "mproc.h"
#include "vm.h"

int parseArguments(int argc, char*argv[])
{
	int tableSize;
	if(argc == 2 && (tableSize = atoi(argv[1])))
	{
		return tableSize;
	}

}

int isInMemory(int pageRequest, int *pageTable, int tableSize)
{
	int i;
```

```
	for (i = 0; i < tableSize; i++)
	{
		if(pageRequest == pageTable[i])
		{
			return 1;
		}
	}
	return 0;
}



void fifo_page_replace(smaller_inner_pagetable* pt)
{
 //initialize all needed variables to check if a replacement is needed.
 int tableSize = pt[0].number_of_pagetables;
 int pageRequest, pageTableIndex = 0, numRequests = 0, numMisses = 0;
 ssize_t bytesRead = &pt[0];
	int i;
	//int numHits = numRequests - numMisses;
	//float hitRate = numHits/numRequests;

	while(bytesRead) != -1)
 {
		pageRequest = list_of_smaller_pagetables[pageTableIndex].page_table_query;
		if (pageRequest == 0)
		{
			continue;
		}
		numRequests++;
		if (!isInMemory(pageRequest, pt, tableSize))
```

```
		{
			numMisses++;
			if (pageTableIndex < tableSize)
			{	//still have room in page table
				pt[pageTableIndex++] = pageRequest;
			}
			else
			{
				// Algorithm for FIFO
				for (i=0; i<tableSize;i++)
				{
					pt[i]=pageTable[i+1];
				}
				pt[tableSize-1]=pageRequest;
			}
		}//update something in page table so that lru and second chance work correctly
	else
		{
			//printf("%d is already in table, no page faults.\n", pageRequest);
		}
	}
	//printf("Hit rate = %f\n", (numRequests - numMisses)/(double)numRequests);
	free(pt);
}
```

We have used a queue to insert the most recent page at the end of the array and the oldest pages at the beginning of the array in ascending order.

We made a few conditions for this implementation when checking for memory.

```
if (!isInMemory(pageRequest, pageTable, tableSize))
```

Where the function isInMemory() iterates over the queue to search for a page with the same page request number and if there are no matches and there is enough space, the function inserts the page after the lastly inserted page. All pages are then shifted using the following for loop:

```
for (i=0; i<tableSize;i++)
{
        pageTable[i]=pageTable[i+1];
}
```

And if a match is found then the function does nothing and skip this iteration.

Then all we needed to do is include that file in the exec.c and forkexit.c placed in /usr/src/servers/pm and call the fifo page replace each time do_exec() or do_fork() are called to handle replacement.

At line 30 in exec.c add the following following include statement:

```
#include "fifo.c"
```

And at line 54 call the fifo_page_replace() function and give it the appropriate parameters:

```
fifo_page_replace(&list_of_smaller_pagetables);
```

And the same in forkexit.c in do_fork().

Step #7: Implement LRU Page Replacement Algorithm in MINIX

Same as FIFO Page Replacement, a new file is created in /usr/src/servers/pm where all the actual algorithm is implemented, and the replace function is then called in do_exec() and do_fork() functions.

```
/*
```

This file is responsible for the full implementation of the LRU page replacement algorithm.

It is a part of the implementation of Requirement #2 - Modification #7.

```
*/

#include <stdio.h>
#include <stdlib.h>
#include "pm.h"
#include <sys/stat.h>
#include <minix/callnr.h>
#include <minix/endpoint.h>
#include <minix/com.h>
#include <minix/vm.h>
#include <signal.h>
#include <libexec.h>
#include <sys/ptrace.h>
#include "mproc.h"
#include "vm.h"

#include <stdio.h>
#include <stdlib.h>

struct Node
{
                        int pageNum;
                        struct Node* next;
                        struct Node* prev;
};

struct Node* head;
struct Node* tail;
```

```
struct Node* GetNewNode(int pageRequest)
{
                struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

                newNode->pageNum = pageRequest;
                newNode->next = NULL;
                newNode->prev = NULL;
                return newNode;
}

void InsertAtTail(int pageRequest)
{
                struct Node* myNode = GetNewNode(pageRequest);
                if (head == NULL)
                {
                head = myNode;
                tail = myNode;
                return;
                }
                tail->next = myNode;
                myNode->next = NULL;
                myNode->prev = tail;
                tail = myNode;
                return;
}

void PopAtHead(void)
{
                struct Node* temp = head;
                head = head->next;
```

```
                head->prev = NULL;
                temp->next = NULL;
                free(temp);
                return;
}

int parseArguments(int argc, char*argv[])
{
                int tableSize;
                if(argc == 2 && (tableSize = atoi(argv[1])))
                {
                return tableSize;
                }
                fprintf(stderr, "Wrong arguments. Pass tableSize as an argument\n");
                exit(-1);

}

struct Node* isInMemory(int pageRequest, int tableSize)
{
                struct Node* check = head;
                while (check != NULL)
                {
                if (check->pageNum == pageRequest)
                {
                        return check;
                }
                check = check->next;
                }
                return NULL;
```

```
}


void lru_page_replace(smaller_inner_pagetable* pt)
{
                        //initialize all needed variables to check if a replacement is needed.
 int tableSize = pt[0].number_of_pagetables;
 int pageRequest, pageTableIndex = 0, numRequests = 0, numMisses = 0;
 ssize_t bytesRead = &pt[0];
                        int i;


                        head = NULL;
                        tail = NULL;
                        //int numHits = numRequests - numMisses;
                        //float hitRate = numHits/numRequests;


                        while(bytesRead != -1)
 {
                        pageRequest = list_of_smaller_pagetables[pageTableIndex].page_table_query;
                        if (pageRequest == 0)
                        {
                                continue;
                        }
                        numRequests++;
                        struct Node* nodeSelected = isInMemory(pageRequest, tableSize);
                        if (nodeSelected == NULL)
                        {
                                numMisses++;
                                //printf("before if pagetabe<tablesize\n");
                                if (pageTableIndex < tableSize)
```

```
        {       //still have room in page table
                InsertAtTail(pageRequest);
                pageTableIndex++;
        }
        else
        {       // TODO implement a page replacement algorithm

                // Algorithm for LRU
                PopAtHead();
                InsertAtTail(pageRequest);
        }
}//update something in pageTable so that lru and second chance work correctly
else
{
        if (nodeSelected == head && nodeSelected != tail)
        {
                head = head->next;
                head->prev = NULL;
                nodeSelected->next = NULL;
                nodeSelected->prev = tail;
                tail->next = nodeSelected;
                tail = nodeSelected;
        }
        else if (nodeSelected != head && nodeSelected != tail)
        {
                nodeSelected->prev->next = nodeSelected->next;
                nodeSelected->next->prev = nodeSelected->prev;
                nodeSelected->prev = tail;
                nodeSelected->next = NULL;
                tail->next = nodeSelected;
```

```
                    tail = nodeSelected;
                }
            }
            struct Node* nodeP = head;
            }
            printf("Hit rate = %f\n", (numRequests - numMisses)/(double)numRequests);
            free(input);

            struct Node* nodeSelected;
            while(head != NULL)
            {
            nodeSelected = head;
            head = head->next;
            free(nodeSelected);
            }
}
```

To implement the LRU, we used a double linked list to manage page replacement to help decrease the number of shifting where the function GetNewNode() creates a new node with its page request and InsertAtTail() inserts it to the list, and lastly PopAtHead() that deletes the page from the list.

The same as FIFO, the function iterates over the page requests yet if a match is found, 2 scenarios may apply, The first is handled using:

```
if (nodeSelected == head && nodeSelected != tail)
```

And the second is handled using:

```
else if (nodeSelected != head && nodeSelected != tail)
```

Also, a matched node can be in the middle of the list and is neither the head nor tail. If the matched node is the tail then the function does nothing do nothing since it is already the tail, which should be set to the most recently used.

The function is lastly called in do_exec() and do_fork() instead of the FIFO function to start page replacing according to its written algorithm.

Performance Analysis of FIFO and LRU:

To compare both page replacement algorithms, we ran them 5 times with varying page sizes and recorded the hit rates of each by using MINIX pre-build utility functions. All the test was for the default system boot processes so all had the same processing situation for a more fair comparison. The results were as follows:

Page Size	FIFO	LRU
50	0.486111	0.513629
100	0.570789	0.597564
150	0.622332	0.646351
200	0.664779	0.720091
250	0.703559	0.753123

Table 1 : Page Size vs Hit Rates

Requirement 3

In this section, three different topics will be discussed thoroughly. These topics are disk allocation, disk space management, and file systems. Since the shine of operating systems in the computer world, designers and engineers have been concerned mostly with the topic of free space and allocating disk space. The whole competition relies on having the freest space available and optimizing the use of disk space as much as possible.

The working framework keeps up a list of free disk spaces to keep track of all disk blocks which are not being utilized by any file. At whatever point a record has got to be made, the list of free disk space is looked for and after that distributed to the unused file. The sum of space distributed to this file is at that point expelled from the free space list. When a file is erased, its disk space is included in the free space list. In this area, we'll examine two strategies to oversee free disk blocks.

Why files are implemented in operating systems?

Since applications always fetch and store information constantly, this information must be easy to access, modify, or delete. If the data is stored in the physical memory of the OS, a lot of problems might occur:

1- Size
2- Data vanishes with process termination
3- Only one process accesses the data at a time

Therefore, operating system developers decided to create new storage devices for this type of information. These storage units are called “files”. Every operating system has its way of managing its file system, but they have some features in common:

- Information should be present when the process terminates
- Store a large amount of data easily
- Multiple processes can access the data stored all at the same time

In the upcoming pages, the explanation will be divided into 2 main parts:

1- Explaining disk management in operating systems in general
    a. Different types of disk allocation
    b. Different types of free space management
    c. Different types of extent-based systems
2- Explaining disk management in MINIX 3 OS specifically
    a. Disk allocation in MINIX
    b. Free space management in MINIX
    c. Extent-based systems in MINIX

4.0 Requirement Four

4.1 Theory and Overview

For the purposes of this section of the report, we will ignore the user's point of view, and focus only on the implementation of file systems, meaning, we will not discuss file naming or user program-imposed file structure. Regarding file types: files are viewed here as a sequence of bytes, executable with the proper format. Regarding file access: we are referring only to random access to files, not sequential access, unless stated otherwise. Regarding directory systems: hierarchical multi-level organization is used in most server file systems.

4.2 Messages and VFS

The MINIX 3 file system works using a concept known as Virtual File System (VFS). The VFS is an abstract layer positioned in the user space. It is simply an interface that deals with user requests, indexes them into a table of procedure pointers, and then the procedure calls requested system call to be carried out. The VFS deals with the actual file system with the cooperation of multiple file servers. It is highly modular and can be separately tested or reused as a network file server with only minor modifications.

Messages from users	Input parameters	Reply value
access	File name, access mode	Status
chdir	Name of new working directory	Status
chmod	File name, new mode	Status
chown	File name, new owner, group	Status
chroot	Name of new root directory	Status
close	File descriptor of file to close	Status
creat	Name of file to be created, mode	File descriptor
dup	File descriptor (for dup2, two fds)	New file descriptor
fcntl	File descriptor, function code, arg	Depends on function
fstat	Name of file, buffer	Status
ioctl	File descriptor, function code, arg	Status
link	Name of file to link to, name of link	Status
lseek	File descriptor, offset, whence	New position
mkdir	File name, mode	Status
mknod	Name of dir or special, mode, address	Status
mount	Special file, where to mount, ro flag	Status
open	Name of file to open, r/w flag	File descriptor
pipe	Pointer to 2 file descriptors (modified)	Status
read	File descriptor, buffer, how many bytes	# Bytes read
rename	File name, file name	Status
rmdir	File name	Status
stat	File name, status buffer	Status
stime	Pointer to current time	Status
sync	(None)	Always OK
time	Pointer to place where current time goes	Status
times	Pointer to buffer for process and child times	Status

umask	Complement of mode mask	Always OK
umount	Name of special file to unmount	Status
unlink	Name of file to unlink	Status
utime	File name, file times	Always OK
write	File descriptor, buffer, how many bytes	# Bytes written

Messages from users	**Input parameters**	**Reply value**
Messages from PM	**Input parameters**	**Reply value**
exec	Pid	Status
exit	Pid	Status
fork	Parent pid, child pid	Status
setgid	Pid, real and effective gid	Status
setsid	Pid	Status
setuid	Pid, real and effective uid	Status
Other messages	**Input parameters**	**Reply value**
revive	Process to revive	(No reply)
unpause	Process to check	(See text)

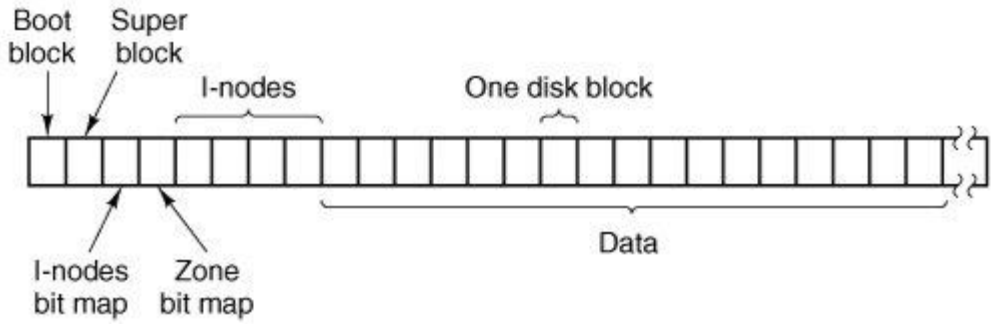
This is a table of all the 39 types of messages accepted by MINIX 3. The code status as a reply value can mean “OK” or “error”.

The way this is implemented is very similar to process manager. The main loop waits for a message to arrive. The message, when it arrives, its type is used as an index into a table containing pointers to the procedures. The appropriate procedure is decided upon and called, then, after its work is done, it returns a status value. A message is sent to the user and it returns to the main loop.

4.3 File System Layout

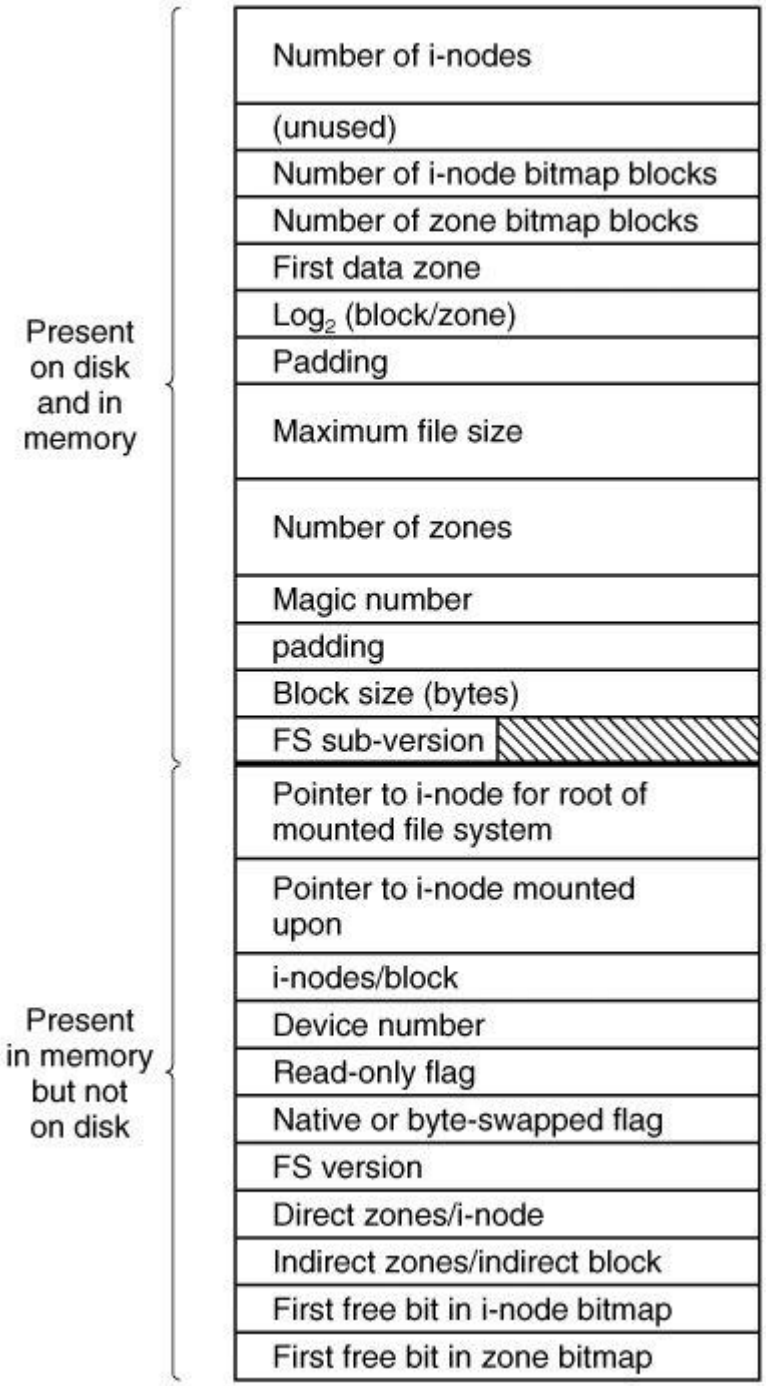
Why do we need file management in the first place? There are many problems with storing all the information of a system in main memory, mainly that it won’t fit and that it is volatile. So we need files to be saved from main memory to hard disk. As previously discussed, most disks are divided into partitions, each managed by independent file systems. In MINIX 3, a partition is allowed to contain a sub-partition table, which has almost the same structure as extended partition, a data structure that points to a linked list of logical partitions. Keep in mind that while logical partitions cannot be used by BIOS to start the operating system, initial start-up from a primary partition will manage the logical partitions. An advantage for MINIX 3 using sub partitions is that the problems in one sub partition cannot be carried to the next one, since they are different ones for root device, swapping, system binaries, and user’s files. Note that the layout of disk partitions can vary from file system to file system (except for starting with master boot block).

A MINIX 3 file system is a self-contained entity with i-nodes, directories and data blocks. It looks like this.



This file system can be stored in hard disk partition or in floppy disk. Regardless, even though the relative size of the components may vary from file system to file system, the structure remains the same.

- **The boot block** contains executable code. It is always 1024 bytes in size. It is the first thing in the boot device in the memory the hardware reads when the computer is turned on, and it loads the operating system. After booting, it is not used anymore.
- **The superblock** contains all the information describing the file system. It is also 1024 bytes in size. The superblock is used to find the size of various parts of the file system. The number of blocks per zone is not stored because all that is needed is the base 2 log of the zone to block ratio, in order to shift convert between zone and block count. Here is the structure of the superblock in MINIX 3.



4.4 Bitmaps

There are two general disk-space management methods. The first is to store an n bit file contiguously, with the same trade-offs between pure segmentation and paging as discussed in memory management systems. The second method is to chop files up into fixed-size blocks, then store blocks (not necessarily) non-adjacently. Blocks can vary in size between operating systems, with the conflict between favouring

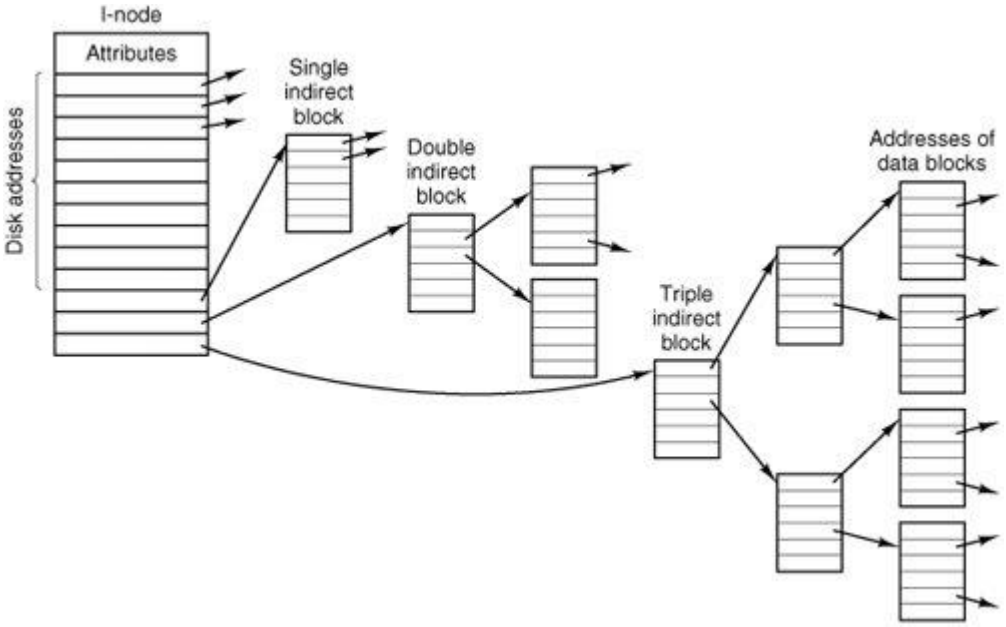
performance over space utilization or vice versa. A good compromise is chosen for MINIX, 1 KB- size block.

Bitmaps are a method of disk space management. They are simply a sequence of bits used to keep track of free blocks. If a disk has n blocks, then it needs a bitmap of n bits.

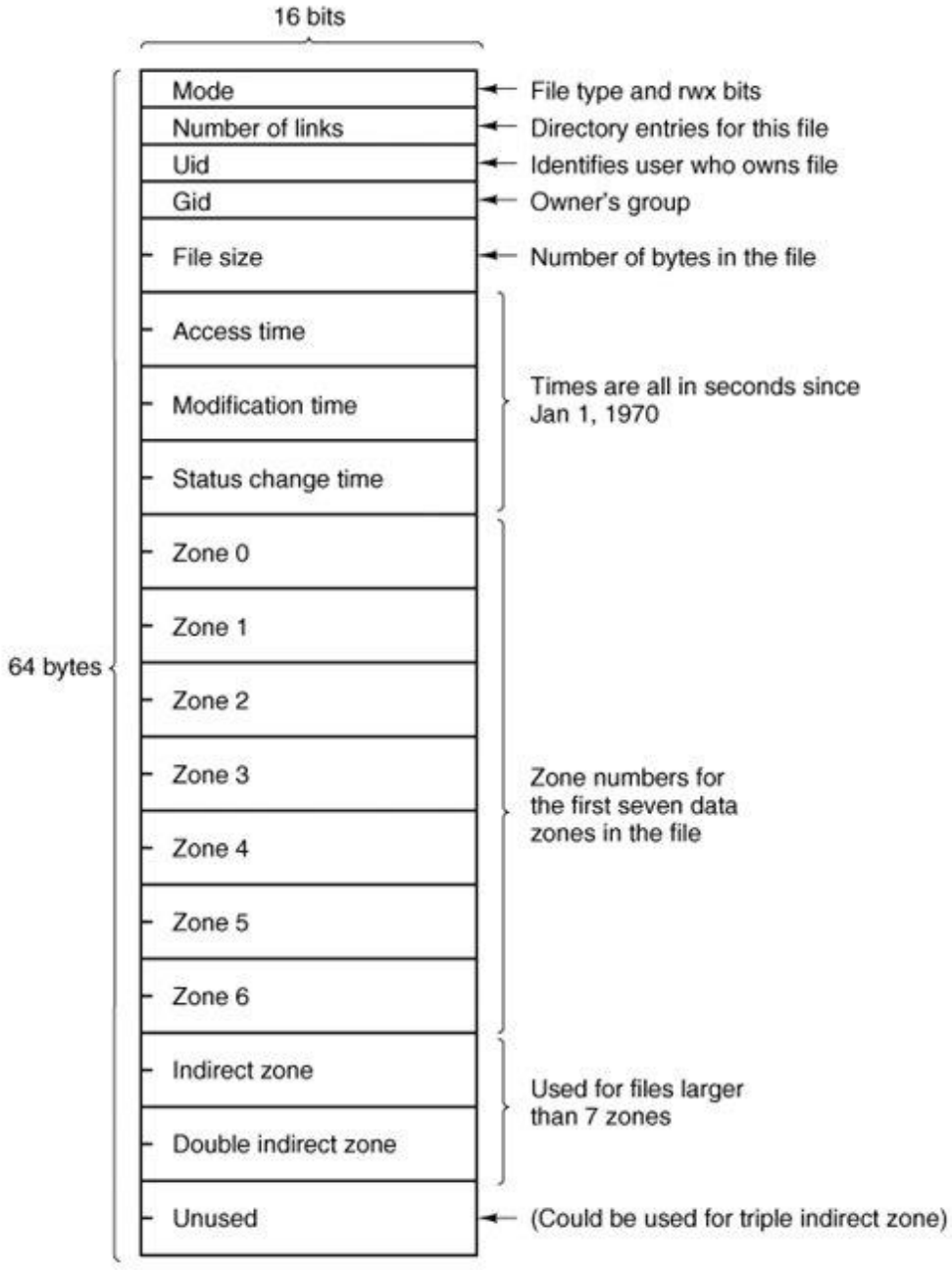
MINIX tracks free i-nodes and zones using two bitmaps. Disk storage is allocated in terms of 2^n blocks called **zones**. A zone allows allocating as much blocks next to each other (on the same cylinder) as possible to save load time. When a new file is created, the first free i-node is located on the disk, then it is allocated to. The first free block is cached already in the superblock in the memory. Whenever a file is deleted, MINIX deletes the i-node by marking its corresponding bit as 0. An i-node is a method of block management which will be discussed next.

4.5 I-Nodes

We need to know which disk blocks go with which file. There are many methods used to keep track of any file's blocks, such as contiguous allocation, linked list allocation, File Allocation Tables (FAT) kept in memory, and i-nodes (index-nodes). I-nodes are a type of data structure that lists all the file's blocks attributes and block addresses. It is almost like a File Control Block that we studied, but for file blocks. Thus, if we are given the i-node of a file, we can find all its blocks. The i-node is useful because it is only present in the memory if its file is open, so it takes up less space than FAT. The figure below describes how an i-node's indirect block works. Indirect block is a solution if the file needs more blocks than the fixed number allocated to be max for an i-node.



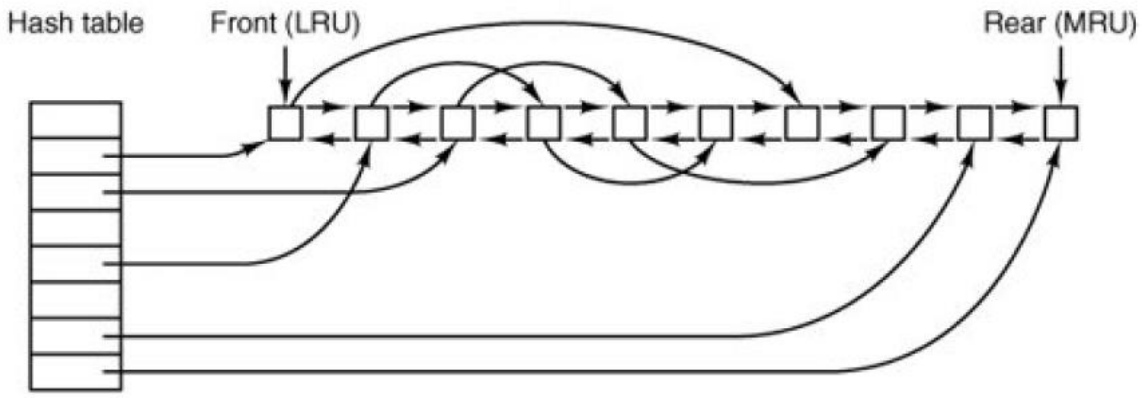
The MINIX 3 i-node layout is given below. There are 9 32-bit disk zone pointers, 7 direct and 2 indirect (files up to 7 KB do not need indirect blocks). When a file is opened, its i-node is found from i-node file and then brought into memory, where it remains until the file is closed. The mode field records the type of file (regular, directory, block special, character special, or pipe). The link field records how many directory entries. Note how the file name is not included in the i-node itself, but it is included in the directory structure.



4.6 The Block Cache

The block cache is used by MINIX to improve file system performance. A “cache” is a series of blocks being kept in memory for performance reasons, to load files. It consists of a fixed array of buffers, each with header info and poiners, counters, flags, as well as the space for one block. The double-linked list is

the chain of buffers not being used, ordered from least to most recently used. For MINIX to read a block, the cache is first searched with the block number hash. File system calls a procedure *get_block* with the device number and the block number, which computes the hash code for that block and searches the buffer chain. If the block is found in the buffer chain, a counter in the buffer header is incremented. Otherwise, the first block from the LRU list is used, because it is not still in use. The flag is used to tell whether the block has been rewritten or not, and if it has been, then it is copied again to the disk. At this point, another procedure, *put_block*, is called to decrement the use counter and free the block. It is also used to decide where to move the block in the LRU list. Note that the modified block is not rewritten unless it reaches the front of the LRU list, or a `sync` system call is executed to force all the modified blocks out onto the disk.



This situation is similar to paging, with the LRU algorithm applicable. Some other things may occur to ensure the consistency of the blocks in the file system, such as writing critical blocks quickly, or dividing the blocks into categories then putting the blocks that will probably not be used again soon in the front so their buffers will be reused quickly.

4.7 Mounting Files

We have already established that MINIX 3 file system, for the most part, looks like any other UNIX file system. Since it is not possible to put the file position in the i-node or in the process table without messing with `fork` system call semantics, a shared *filp* (file position) table must be used. The *filp* table is shared between a parent and the child, so the position where the first process left off writing in the table can be inherited. For similar reasons, a *file_lock* table to record all locks (file locked for reading or writing) is used as a separate way and for implementing POSIX advisory locking.

MINIX 3 partitions have to be mounted to the root of the file system before accessed, as shown below.

4.8 Testing

Types of file systems on FS module in minix3

1. **ext2: second extended file system**

ext File System

The ext file system stands for "Extended File System". It was the first file system designed to support the Linux kernel.

Virtual File System (VFS) was used for the ext file system. Its primary purpose was to allow the Linux kernel to access the ext file system. The ext file system restricted filename lengths to 255 characters and supported partitions up to 2GB.

While it managed to solve issues that the Minix file system had, it had one major flaw – timestamping. Unlike today where each Linux file has three timestamps (access timestamp, modified timestamp, and changed timestamp), the ext file system allowed only one timestamp per file.

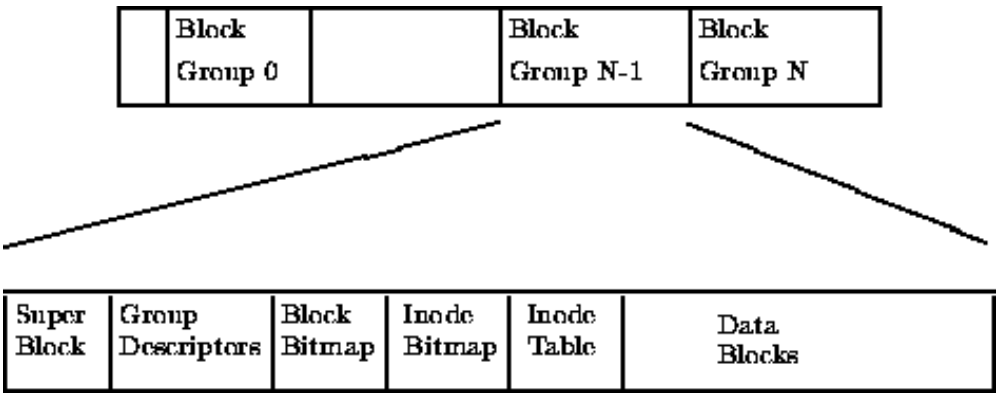
ext2 File System

The ext2 file system enabled the retention of the internal structure while the file system functionalities extended. Data from files were kept in data blocks of the same length. The ext2 file system supported the maximum file size of 2TiB. Filename lengths were not limited in characters, but in bytes – 255 bytes. It did not support journaling.

While this file system was largely used, it still had two major issues:

- **File corruption** – This phenomenon would occur if data were written to the disk at the time of a power loss or system crash.
- **Performance loss** – Disk fragmentation happens when a single file is broken into pieces and spread over several locations on the disk. As a result, files take longer to read and write, which leads to performance degradation.

The ext2 system was mostly used until the early 2000s when the ext3 file system was introduced. It is occasionally used today for USB devices because it does not support the journaling system.



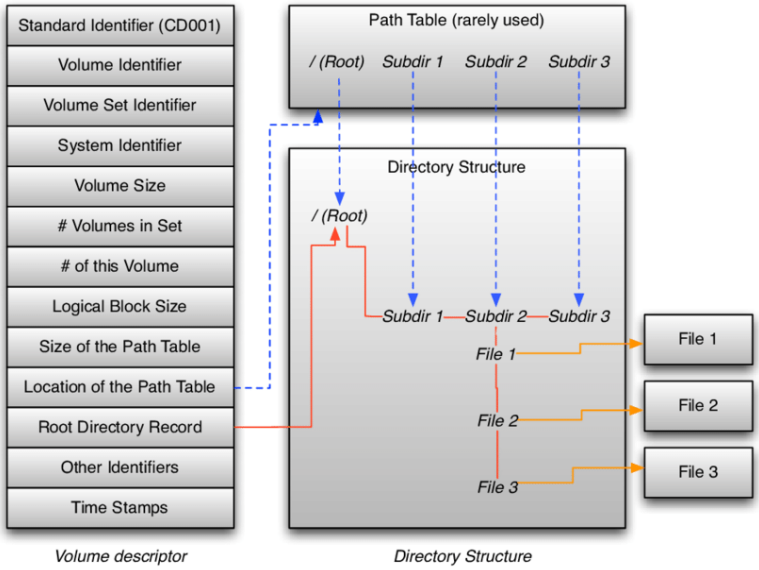
The Second Extended File system was devised (by Rémy Card) as an extensible and powerful file system for Linux. It is also the most successful file system so far in the Linux community and is the basis for all of the currently shipping Linux distributions.

The EXT2 file system, like a lot of the file systems, is built on the premise that the data held in files is kept in data blocks. These data blocks are all of the same length and, although that length can vary between different EXT2 file systems the block size of a particular EXT2 file system is set when it is created (using mke2fs). Every file's size is rounded up to an integral number of blocks. If the block size is 1024 bytes, then a file of 1025 bytes will occupy two 1024 byte blocks. Unfortunately this means that on average you waste half a block per file. Usually in computing you trade off CPU usage for memory and disk space utilisation. In this case Linux, along with most operating systems, trades off a relatively inefficient disk usage in order to reduce the workload on the CPU. Not all of the blocks in the file system hold data, some must be used to contain the information that describes the structure of the file system. EXT2 defines the file system topology by describing each file in the system with an inode data structure. An inode describes which blocks the data within a file occupies as well as the access rights of the file, the file's modification times and the type of the file. Every file in the EXT2 file system is described by a single inode and each inode has a single unique number identifying it. The inodes for the file system are all kept together in inode tables. EXT2 directories are simply special files (themselves described by inodes) which contain pointers to the inodes of their directory entries.

Figure gif shows the layout of the EXT2 file system as occupying a series of blocks in a block structured device. So far as each file system is concerned, block devices are just a series of blocks which can be read and written. A file system does not need to concern itself with where on the physical media a block should be put, that is the job of the device's driver. Whenever a file system needs to read information or data from the block device containing it, it requests that its supporting device driver reads an integral number of blocks. The EXT2 file system divides the logical partition that it occupies into Block Groups. Each group duplicates information critical to the integrity of the file system as well as holding real files and directories as blocks of information and data. This duplication is neccessary should a disaster occur and the file system need recovering. The subsections describe in more detail the contents of each Block Group.

2. isofs: ISO9660(International Standardisation Office) file system for optical drives

ISO 9660 is a file system for optical disc media.



3. **mfs :memory file system**

A filesystem that stored in memory. Memory filesystems are useful for caches, temporary data stores, unit testing, etc. Since all the data is in memory, they are very fast, but non-permanent

4. **pfs: physical/pipe file system**

Management of the open pipes and closed devices throughout the entire system

5. **procfs: process file system**

The proc filesystem is a special filesystem in Unix-like operating systems that presents information about processes and other system information in a hierarchical file-like structure

6. **ptyfs: file system for Unix98 pseudoterminal slave nodes**

7. **vbfs: File system for Virtual Box**

VFS-FS PROTOCOL

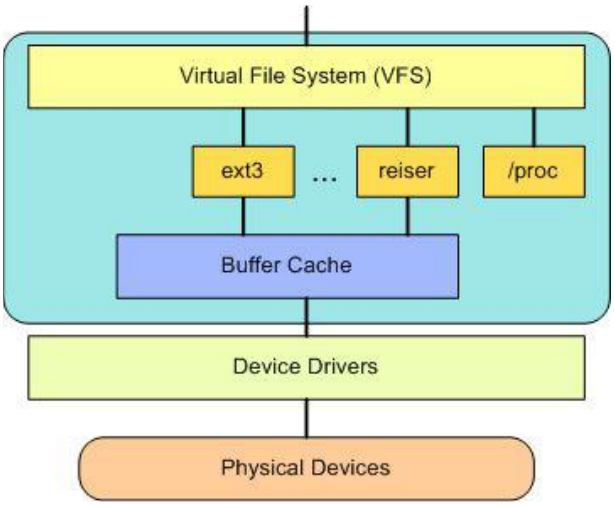
1. The user process executes the read system call which is delivered to VFS.

2. VFS verifies the read is done on a valid (open) file and forwards the request to the FS responsible for the file system on which the file resides.

3. The FS reads the data, copies it directly to the user process, and replies to VFS it has executed the request.

4. Subsequently, VFS replies to the user process the operation is done and the

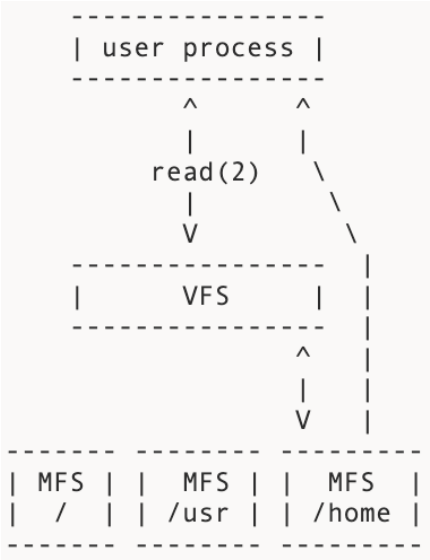
user process continues to run.



Requirement Implementation

A) SRC CODE AMENDMENT (of ext2)

- Change block allocation and extension to new disk space:
  //Change block allocation to allow extension (treat leftover inode as part of the one block) so when a new block is allocated it doesn't have the same address as the previous one
  1)CHANGES TO BLOCKS IN EXT2 MODULE OF FS



```
71  //find appropriate block to allocate
72  block_t alloc_block(struct inode *rip, block_t block)
73  {
74  /* Allocate a block for inode. If block is provided, then use it as a goal:
75   * try to allocate this block or his neghbors.
76   * If block is not provided then goal is group, where inode lives.
77   */
78    block_t goal; //to use as goal
79    block_t b; //block
80    struct super_block *sp = rip->i_sp;
81
82    if (sp->s_rd_only)
83      panic("can't alloc block on read-only filesys.");
84
85    /* Check for free blocks. First time discard preallocation,
86     * next time return NO_BLOCK
87     */
88    //Condition to check that:
89    //No use of reserved blocks//Free blocks count<Reserved blocks count
90   //WASTING BLOCKS!!!//So mark as no block
91    if (!opt.use_reserved_blocks &&
92        sp->s_free_blocks_count <= sp->s_r_blocks_count) {
93      discard_preallocated_blocks(NULL);
94    } else if (sp->s_free_blocks_count <= EXT2_PREALLOC_BLOCKS) {
95      discard_preallocated_blocks(NULL);
96    }
97
98    if (!opt.use_reserved_blocks &&
99        sp->s_free_blocks_count <= sp->s_r_blocks_count) {
100     return(NO_BLOCK);
101   } else if (sp->s_free_blocks_count == 0) {
```

alloc_block function in balloc.c in fs/ext2

```
/*===========================================================================*
 *              write_map                    *
 *===========================================================================*/
int write_map(rip, position, new_zone, op)
struct inode *rip;      /* pointer to inode to be changed */
off_t position;         /* file address to be mapped */
zone_t new_zone;        /* zone # to be inserted */
//while current inode has no allocated zone
while(rip->i_zone==NO_ZONE ){
    int op;             /* special actions */
    {
    /* Write a new zone into an inode.
     *
     * If op includes WMAP_FREE, free the data zone corresponding to that position
     * in the inode ('new_zone' is ignored then). Also free the indirect block
     * if that was the last entry in the indirect block.
     * Also free the double indirect block if that was the last entry in the
     * double indirect block.
     */
      int scale, ind_ex = 0, new_ind, new_dbl,
            zones, nr_indirects, single, zindex, ex;
      zone_t z, z1, z2 = NO_ZONE, old_zone;
      register block_t b;
      long excess, zone;
      struct buf *bp_dindir = NULL, *bp = NULL;

      IN_MARKDIRTY(rip);
      scale = rip->i_sp->s_log_zone_size;       /* for zone-block conversion */
        /* relative zone # to insert */
      zone = (position/rip->i_sp->s_block_size) >> scale;
      zones = rip->i_ndzones;   /* # direct zones in the inode */
```

Added condition to make sure the present inode has no dedicated zone from the disk in fs/mfs/write.c

```
58        /* Is 'position' to be found in the inode itself? */
59        if (zone < zones) {
60          zindex = (int) zone;    /* we need an integer here */
61          //full rip->zone[zindex]
62          if(rip->i_zone[zindex] != NO_ZONE && (op & WMAP_FREE)) {
63              free_zone(rip->i_dev, rip->i_zone[zindex]);
64              rip->i_zone[zindex] = NO_ZONE;
65              printf("No zone in this inode%s",str);
66          } else {
67              rip->i_zone[zindex] = new_zone;
68              printf("This inode has a new zone%s",str);
69          }
70          //return(OK);
71          continue;
72        }
73
74        /* It is not in the inode, so it must be single or double indirect. */
75        excess = zone - zones;    /* first Vx_NR_DZONES don't count */
76        new_ind = FALSE;
77        new_dbl = FALSE;
78
79        if (excess < nr_indirects) {
80          /* 'position' can be located via the single indirect block. */
81          z1 = rip->i_zone[zones];    /* single indirect zone */
82          single = TRUE;
83        } else {
84          /* 'position' can be located via the double indirect block. */
```

PRINT STATUS OS ZONE ALLOCATION and change all checks that a free zone has been found (return (OK);) except last one in line 190

```
169               }
170           } else {
171               wr_indir(bp, ex, new_zone);
172           }
173           /* z1 equals NO_ZONE only when we are freeing up the indirect block. */
174           if(z1 != NO_ZONE) MARKDIRTY(bp);
175           put_block(bp);
176         }
177
178         /* If the single indirect block isn't there (or was just freed),
179          * see if we have to keep the double indirect block, if any.
180          * If we don't have to keep it, don't bother writing it out.
181          */
182         if(z1 == NO_ZONE && !single && z2 != NO_ZONE &&
183            empty_indir(bp_dindir, rip->i_sp)) {
184           free_zone(rip->i_dev, z2);
185           rip->i_zone[zones+1] = NO_ZONE;
186         }
187
188         put_block(bp_dindir);         /* release double indirect blk */
189
190         return(OK);
191       }
192   }//end of while loop
193
```

Define number of blocks to 5 in include/minix/config.h
(no need for sys_config.h)
Nrblocks references to alloc_zone and read_and_set and main and all are located in mkfs.c(microkernel file system . c)

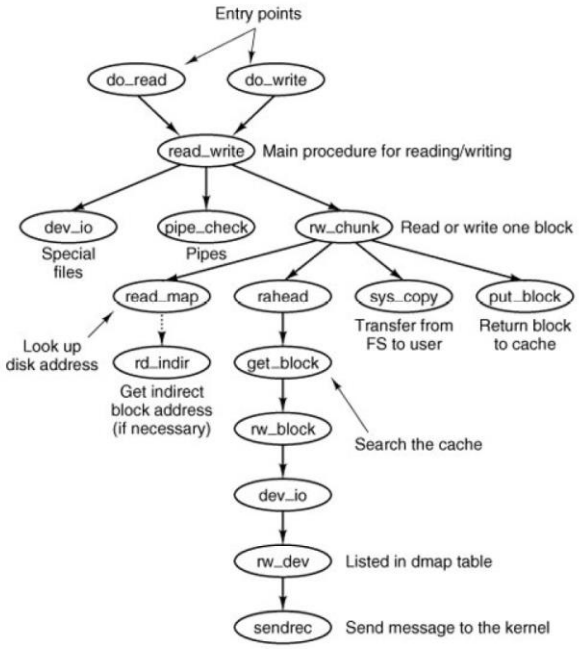

```
73 /* default scheduling quanta */
74 #define USER_QUANTUM 200
75
76 /* defualt user process cpu */
77 #define USER_DEFAULT_CPU    -1 /* use the default cpu or do not change the
78                                   current one */
79 /*define number of blocks*/
80 #define nrblocks            5
81 /*===========================================================================*
82  *  There are no user-settable parameters after this line                    *
83  *===========================================================================*/
```

B) ADDITION OF NEW SYSTEM CALL TO EASE TESTING

```
428 /*file_rsrc*/   //CREATED SYSTEM CALL TO VIEW FILE SYSTEM USAGE
429 int fs_rsrc(char* path);
430 #endif
```

Created system call that uses the string (fs path) as a parameter

List of system calls by vfs/table.c



```
    CALL(VFS_UTIMENS)   = do_utimens,       /* [fl]utime[n]s(2) */
    CALL(VFS_VMCALL)    = do_vm_call,
    CALL(VFS_GETVFSSTAT)    = do_getvfsstat,    /* getvfsstat(2) */
    CALL(VFS_STATVFS1)  = do_statvfs,       /* statvfs(2) */
    CALL(VFS_FSTATVFS1) = do_fstatvfs,      /* fstatvfs(2) */
    CALL(VFS_GETRUSAGE) = do_getrusage,     /* (obsolete) */
    CALL(VFS_SVRCTL)    = do_svrctl,        /* svrctl(2) */
    CALL(VFS_GCOV_FLUSH)    = do_gcov_flush,    /* gcov_flush(2) */
    CALL(VFS_MAPDRIVER) = do_mapdriver,     /* mapdriver(2) */
    CALL(VFS_COPYFD)    = do_copyfd,        /* copyfd(2) */
    CALL(VFS_SOCKETPATH)    = do_socketpath,    /* socketpath(2) */
    CALL(VFS_GETSYSINFO)    = do_getsysinfo,    /* getsysinfo(2) */
    CALL(VFS_SOCKET)    = do_socket,        /* socket(2) */
    CALL(VFS_SOCKETPAIR)    = do_socketpair,    /* socketpair(2) */
    CALL(VFS_BIND)      = do_bind,      /* bind(2) */
    CALL(VFS_CONNECT)   = do_connect,       /* connect(2) */
    CALL(VFS_LISTEN)    = do_listen,        /* listen(2) */
    CALL(VFS_ACCEPT)    = do_accept,        /* accept(2) */
    CALL(VFS_SENDTO)    = do_sendto,        /* sendto(2) */
    CALL(VFS_SENDMSG)   = do_sockmsg,       /* sendmsg(2) */
    CALL(VFS_RECVFROM)  = do_recvfrom,      /* recvfrom(2) */
    CALL(VFS_RECVMSG)   = do_sockmsg,       /* recvmsg(2) */
    CALL(VFS_SETSOCKOPT)    = do_setsockopt,    /* setsockopt(2) */
    CALL(VFS_GETSOCKOPT)    = do_getsockopt,    /* getsockopt(2) */
    CALL(VFS_GETSOCKNAME)   = do_getsockname,   /* getsockname(2) */
    CALL(VFS_GETPEERNAME)   = do_getpeername,   /* getpeername(2) */
    CALL(VFS_FS_RSRC)   =   file_rsrc,     /* file_rsrc(char* path)*/
    CALL(VFS_SHUTDOWN)  = do_shutdown,      /* shutdown(2) */
};
```

```
#define VFS_GCOV_FLUSH      (VFS_BASE + 44)
#define VFS_MAPDRIVER       (VFS_BASE + 45)
#define VFS_COPYFD      (VFS_BASE + 46)
#define VFS_SOCKETPATH      (VFS_BASE + 47)
#define VFS_GETSYSINFO      (VFS_BASE + 48)
#define VFS_SOCKET      (VFS_BASE + 49)
#define VFS_SOCKETPAIR      (VFS_BASE + 50)
#define VFS_BIND        (VFS_BASE + 51)
#define VFS_CONNECT     (VFS_BASE + 52)
#define VFS_LISTEN      (VFS_BASE + 53)
#define VFS_ACCEPT      (VFS_BASE + 54)
#define VFS_SENDTO      (VFS_BASE + 55)
#define VFS_SENDMSG     (VFS_BASE + 56)
#define VFS_RECVFROM        (VFS_BASE + 57)
#define VFS_RECVMSG     (VFS_BASE + 58)
#define VFS_SETSOCKOPT      (VFS_BASE + 59)
#define VFS_GETSOCKOPT      (VFS_BASE + 60)
#define VFS_GETSOCKNAME     (VFS_BASE + 61)
#define VFS_GETPEERNAME     (VFS_BASE + 62)
#define VFS_SHUTDOWN        (VFS_BASE + 63)

#define VFS_FS_RSRC         (VFS_BASE + 64)
#define NR_VFS_CALLS        65  /* highest number from base plus one */

#endif /* !_MINIX_CALLNR_H */
```

Above is the number of system calls to the vfs

proto.h | table.c | callnr.h | vfs_fs_rsrc.c

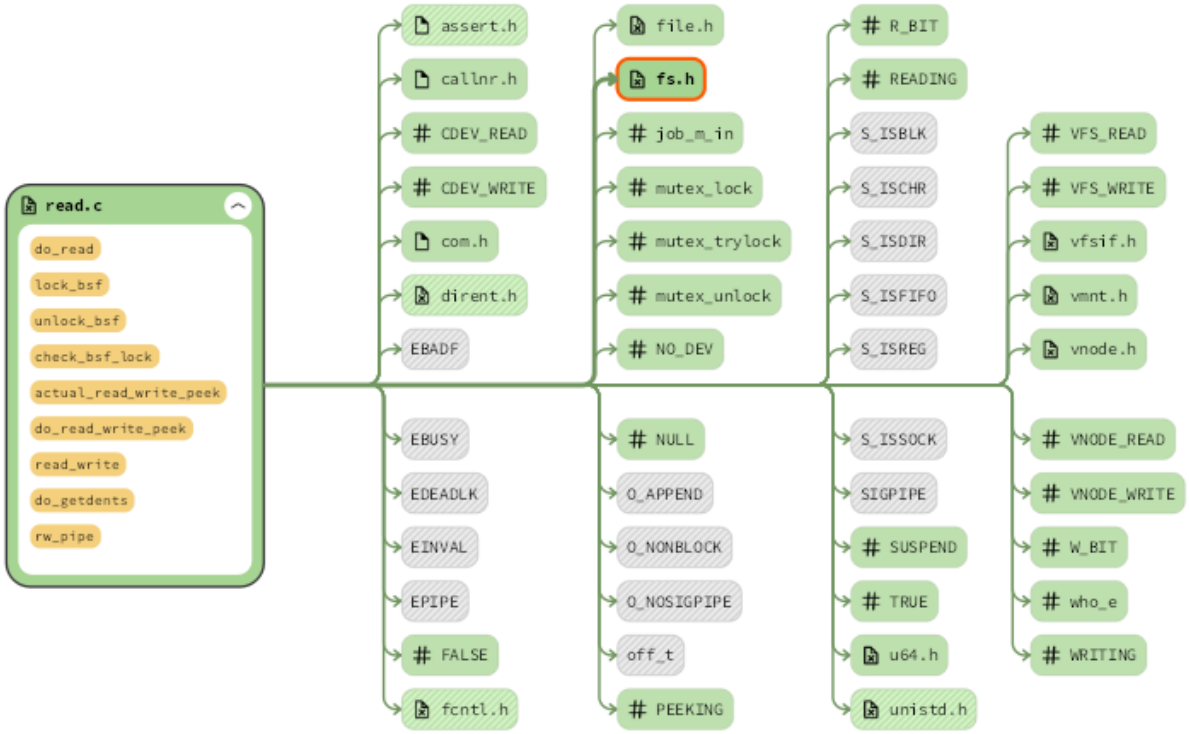
```c
#include <stdio.h>

int fs_rsrc(char* path) {
    printf("Executing Block Allocation %s",str);
    return 0;
}
```

DIRECTORIES IN VFS

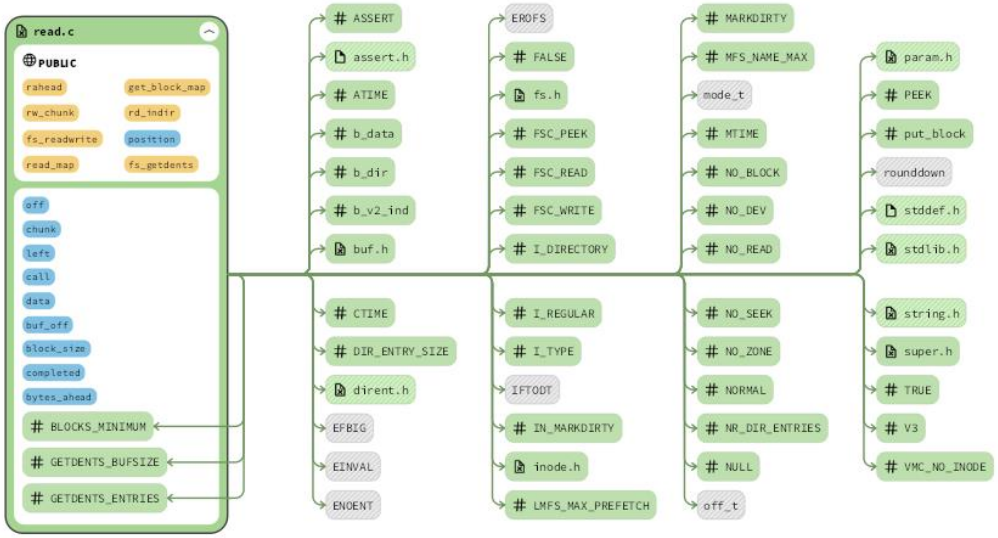
USED TO ADD AND IMPLEMENT SYSTEM CALL TO CHECK FOR REQ 4

```
        bdev.c cdev.c sdev.c smap.c socket.c

.if ${MKCOVERAGE} != "no"
SRCS+=  gcov.c
CPPFLAGS+= -DUSE_COVERAGE
.endif

.if defined(__MINIX)
#LSC: -Wno-maybe-uninitialized while compiling with -DNDEBUG -O3
CWARNFLAGS.gcc+=      -Wno-maybe-uninitialized
.endif # defined(__MINIX)

CFLAGS+= -Wall -Wextra -Wno-sign-compare -Werror
DPADD+=    ${LIBSYS} ${LIBTIMERS} ${LIBEXEC}
LDADD+=    -lsys -ltimers -lexec -lmthread
SRCS+= vfs_fs_rsrc.c #added system call
.include <minix.service.mk>
```

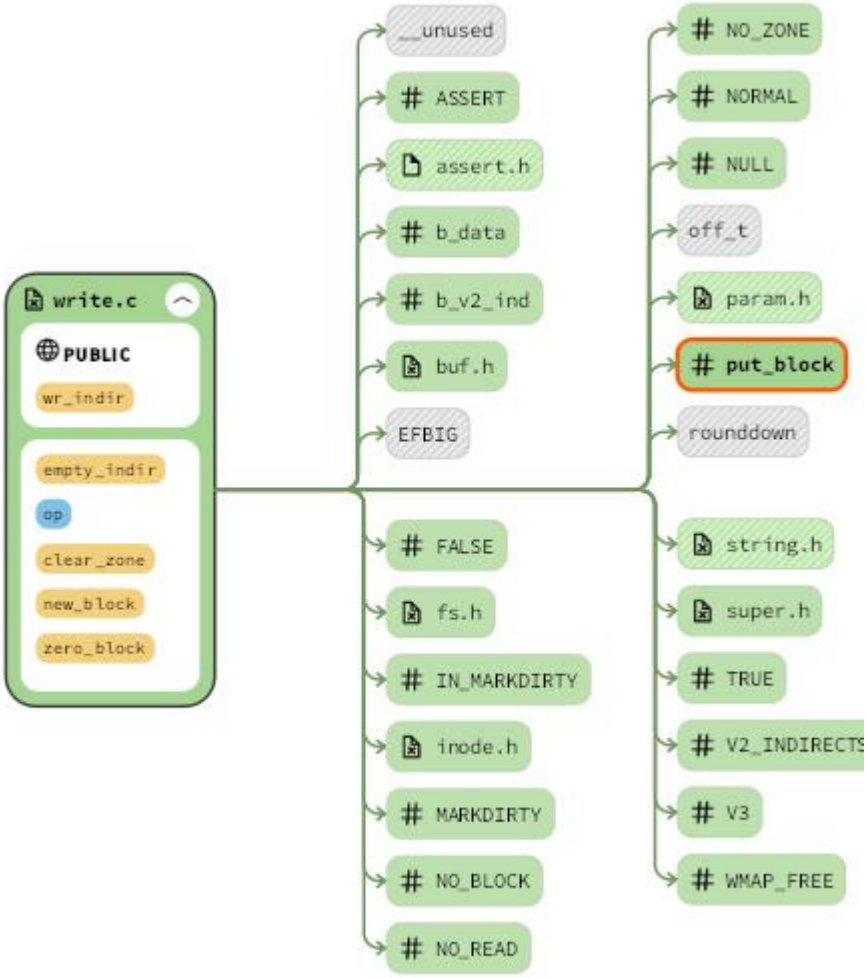
MakeFile Edited



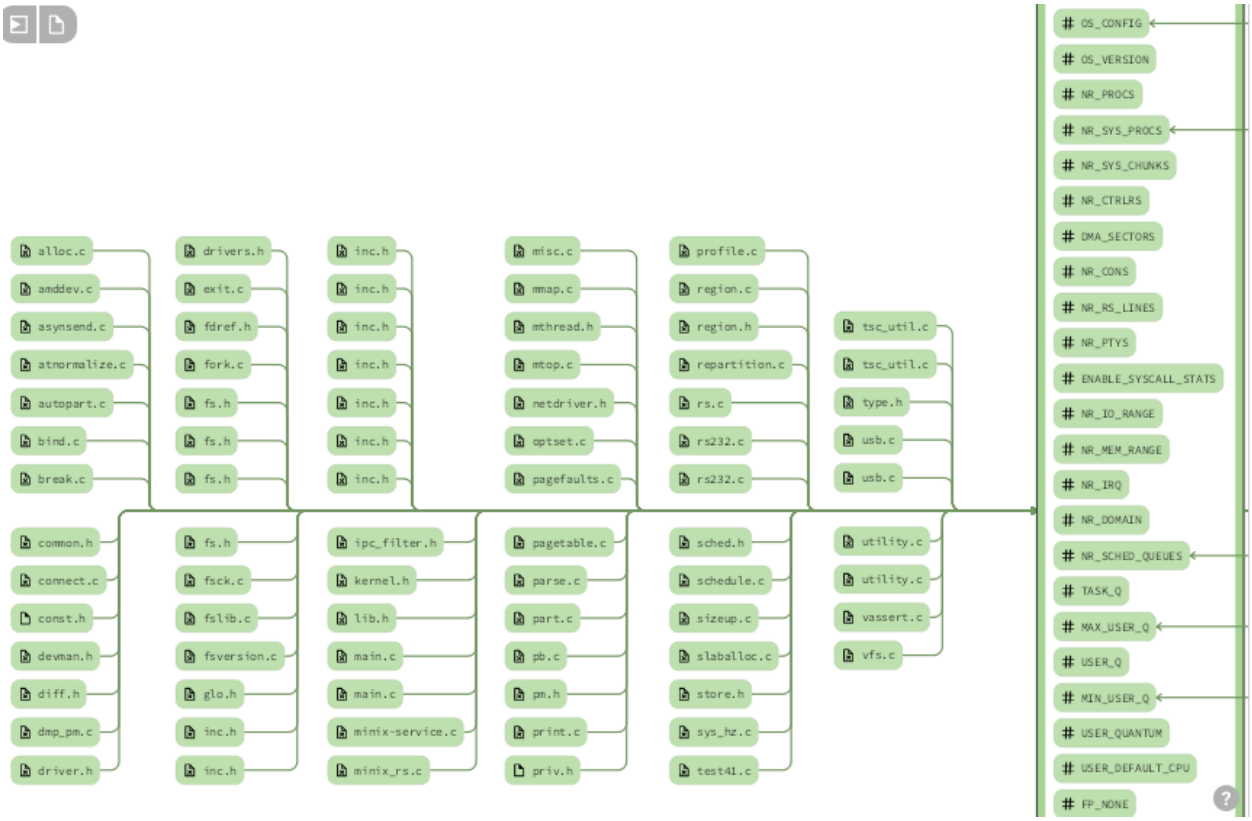
1st pic: READ FROM VFS (REFER TO VFS-FS PROTOCOL)



2nd pic: READ from mfs(REFER TO TYPES OF FILE SYSTEMS)



3rd pic:write from mfs



4th pic:change in config file