

Table of Contents

1	REQUIRNMET 1:	4
1.1	Operating System Structure:	4
1.1.1	Introduction:	4
1.1.2	MINIX 3.0 Structure:	6
1.2	Memory Management in Minix 3.0	8
1.2.1	Overview	8
1.2.2	Memory Layout	8
1.2.3	Process Manager Data Structures and Algorithms	9
1.3	Process Management	12
1.3.1	Overview	12
1.3.2	Minix start up	12
1.3.3	Interprocess Communication in MINIX 3	12
1.3.4	Process Scheduling in MINIX 3	13
1.3.5	The System Task in MINIX 3	14
1.3.6	The Clock Task in MINIX 3	14
1.4	File System in MINIX 3	18
1.4.1	Overview	18
1.4.2	Messages	18
1.4.3	File System Layout	18
1.4.4	Bitmaps	20
1.4.5	I-Nodes	21
1.4.6	The Block Cache	22
1.4.7	Directories and Paths	23
1.4.8	File Descriptors	23
1.4.9	File Locking	24
1.4.10	Pipes and Special Files	24
1.4.11	Free Space Management	25
2	REQUIRNMET 2:	28
2.1	LITTLE HISTORY:	28
2.2	How scheduling works now:	28
2.3	The default user mode scheduler	29
2.4	Functions involved	29
2.5	CODE IMPLEMENTATION	32
2.6	OUTPUT SCREENSHOTS OF PERFORMANCE ANAYLSIS	38

3	REQUIRNMMENT 3:	40
3.1	Memory Management.....	40
3.2	Memory allocation	40
3.2.1	Contiguous memory allocation	40
3.2.2	Multiple partition allocation	41
3.2.3	Paging.....	41
3.3	Page replacement.....	44
3.3.1	FIFO (First-In-First-Out)	44
3.3.2	LRU (Least recently used)	45
3.4	Memory management in Minix	47
3.4.1	VM server	48
3.5	Implementing Hierarchal Paging	49
3.5.1	STEP 1.....	49
3.5.2	STEP 2	50
3.5.3	Step 3.....	50
3.5.4	Step 4.....	51
3.5.5	BONUS STEP	51
3.6	IMPLEMENTING FIFO.....	51
3.6.1	STEP 1.....	52
3.6.2	STEP 2	54
3.7	IMPLEMENTING LRU.....	55
3.7.1	STEP 1.....	55
3.7.2	STEP 2	59
3.8	OUTPUT SCREENSHOT OF PERFORMANCE ANALYSIS	60
4	REQUIRNMMENT 4:	61
4.1	Minix Implementation of file System.....	61
4.2	VFS Internals	61
4.3	Minix Disk Partitions.....	62
4.4	File Systems	63
4.4.1	EXT2 File System	63
4.4.2	EXT2 allocating blocks overview	64
4.5	Functions Involved in block allocation	64
4.5.1	Write.C file	66
4.6	Extent-based allocation code implementation.....	68
4.7	Free disk space management.....	69
5	REFERENCES:	72

1 REQUIREMENT 1:

“The details of internal structure of operating system”

1.1 Operating System Structure:

1.1.1 Introduction:

General-purpose Operating System is very large program so there is various ways to structure it: Simple structure, more complex, Layered, Microkernel, and Modules.

- Simple structure; written to provide the most functionality in the least space, it isn't divided into modules. Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated, and finally this structure is efficient because it is one unit but difficult in editing [1].
- The Complex structure: OS consists of two separable parts:
 - 1- Systems programs
 - 2- The kernel: consists of everything above the physical hardware and below the system-call interface and it provides the file system, CPU scheduling, memory management, and other operating-system functions [1].
- Layered structure: The operating system has been divided into a number of layers (it is also called levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware and the highest (layer N) is the user interface. Layers are selected such that each layer uses functions (operations) and services of only the lower-level layer underneath it [1].
- Microkernel Structure: Moves as much from the kernel into user space. Communication takes place between user modules using message passing through the microkernel

Benefits:

- Easier to extend a microkernel
- Easier to port the operating system to new architectures
- More reliable (less code is running in kernel mode)
- More secure

Disadvantages:

- Performance overhead of user space to kernel space communication
- Modules Structure: Many modern operating systems implement **loadable kernel modules**. Module structure is similar to layers but with more flexibility because any module can call any other module, also it is similar to the microkernel but it is more efficient, because modules in order to communicate do not need to invoke message passing [1].

Note: Most modern operating systems are actually not only one pure model. Examples for this structure:

- 1- Linux and Solaris kernels in kernel address space, so monolithic, plus modular for dynamic loading of functionality
- 2- Windows mostly monolithic, plus microkernel for different subsystem personalities
- 3- Apple Mac OS X hybrid, layered

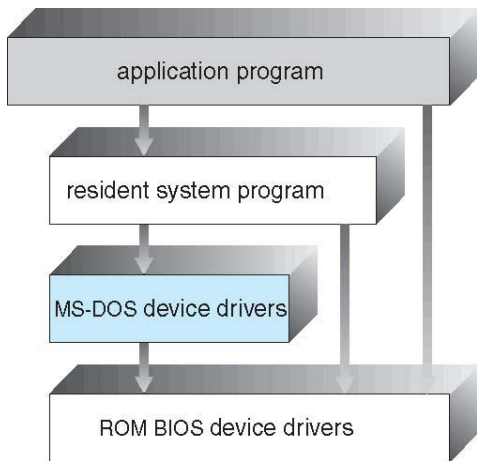


Figure 1: Simple Structure

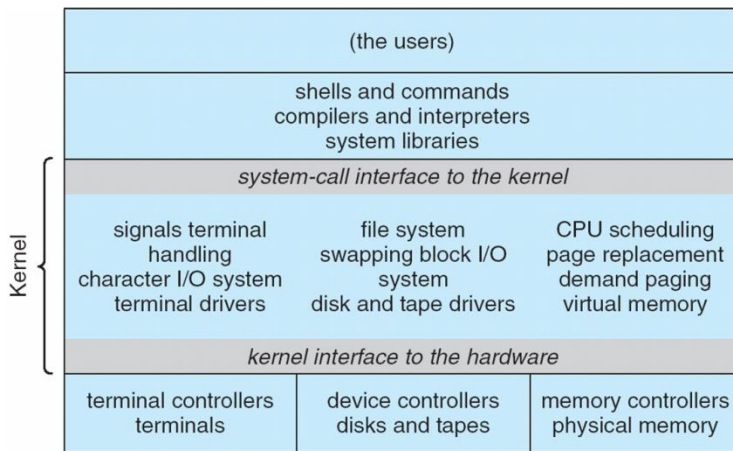


Figure 2: UNIX System Structure, example of Complex Structure

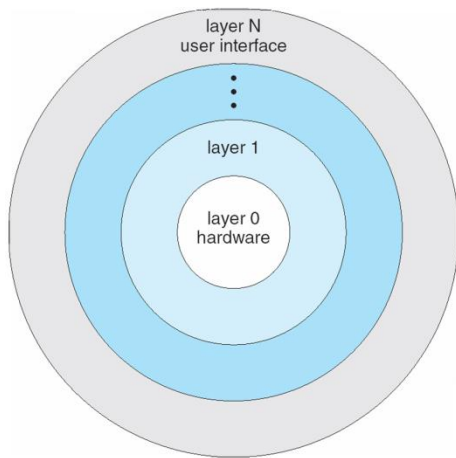


Figure 3: Layered Structure

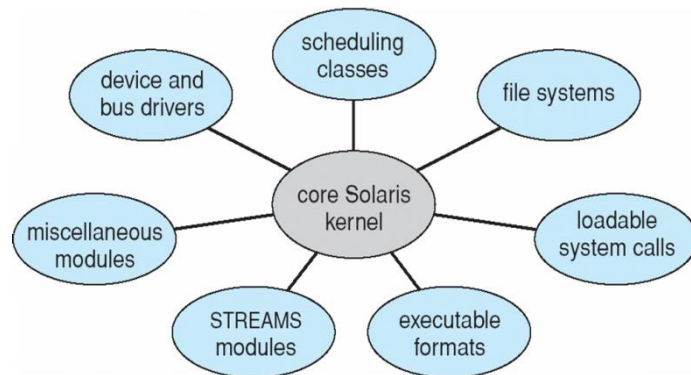


Figure 4: Solaris Modular Approach, example of Module Structure

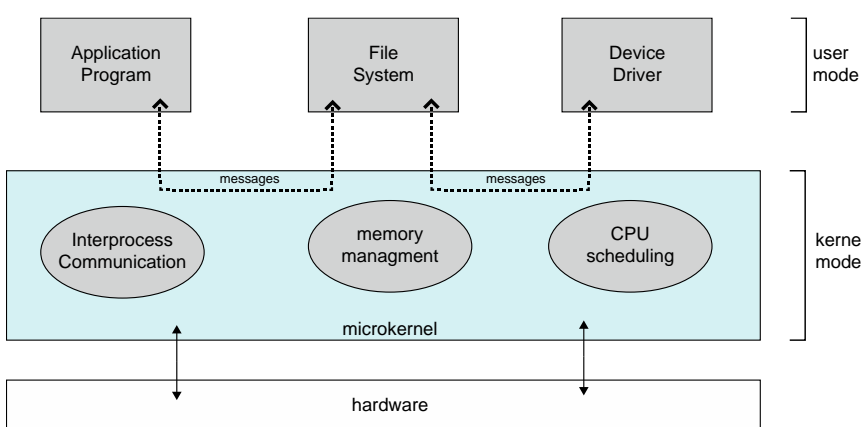


Figure 5: Microkernel System Structure

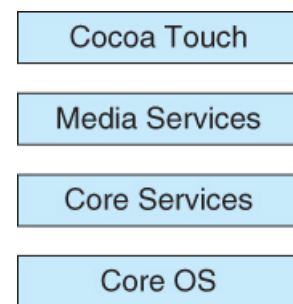


Figure 6: Architecture of Apple's iOS.

1.1.2 MINIX 3.0 Structure:

The Minix operating system implements a layered, micro-kernel architecture as shown in figure [7]. Minix combines two structures: the layered structure and client/server structure.

- The layered system structure divides the system into a number of layers (levels) that implement specific functions; the higher level layers relay on the services provided by the lower level layers. Minix has four layers, each with a specific and well defined function that will be discussed later.
- In micro-kernel architecture the key functions of the operating system most of them are implemented as servers which run separately from the core kernel; Key services provided by a Micro-Kernel are thread management, address space management, timer management, and inter-process communication. This design makes the operating system extensible and module since additional services can also be improved and developed without so many changes to the kernel[2][3].

❖ Structure of Minix: “Minix 3” is structured into **four layers** as follows:

1- Layer 1: Kernel

This layer (level) provides the lowest level services which are needed to run the system. These include traps, management of interrupts, communication, and scheduling. Take into consideration that the lowest part of this layer that details with interrupts is written in assembly language but the rest of it is written in C.

This layer does the following: Manage interrupts and Traps, Scheduling, Providing Services to the next layer, saving and restoring registers, and Messaging and communication services [2]

2- Layer 2: Device Drivers

This layer contains the Input/Output devices. The Input/Output devices which are found in this layer (level) are such peripherals like keyboards, printers, disks, CD drivers etc [2].

3- Layer 3: Server Processes

This layer (level) provides services which are used by all programs which are running in layer 4 (level 4). The Processes in this layer (level) are able to access services of the device driver layer but cannot have direct access to Layer 2 (level 2) processes programs in Layer 4 (level 4).

Example of some of these services include: Reincarnation server, file system, information server, network server, process manager, memory manager etc. In layered approach, the servers provide services to programs /processes which run in layer 4 (level 4) while consuming services from lower layers [2].

4- Layer 4: User Processes

This layer forms the userland section of Minix operating system in which user programs are executed. These programs totally relay on the services which are provided by services of the lower level. Programs which are found in this layer include shells, daemons of various types, commands and any other program that may want to run by the user.

The processes that are in layer 4 have the least privileges to access resources and gets to privileged resources through lower level processes. For example, user can execute the ping command that requires the networking

server process usage. The ping command; does not call the networking server process directly, So instead it goes through the file system server process.

❖ Merits of Minix Architecture

- **Modularity:** the relationship between various components is equally well defined and the system is well structured.
- **Security:** combination of the micro-kernel design and layered system structure makes for architecture which allows for better incorporation of security. Layers 2 and Layer 3 run in User mode but layer 1, runs only in Kernel mode which has all the privileges that are needed to access any part of the system.
- **Extensibility:** For having a functional system, one needs the key services as well as the kernel setup which are needed to start off. All the other functions can be added as and when it is needed. This makes it simpler to specialize and/or extend the function of the system.
- **Performance & Stability:** most problems which cause instability on a computer system are a result of poorly compiled drivers and user programs and/or poor designed. The micro kernel architecture allows these programs to be implemented and executed independent of the core of the components of the operating system, which means that a failure in any of those programs is not catastrophic to the system. The system can maintain uptime, despite errors [2].

❖ Demerits of the Architecture

- **Complexity:** the architecture is complex, so that it makes it harder to design in first place and to evolve as time moves on. The computer industry is amongst the fastest moving industry of the global economy and is a real need to adjust to innovations in software as well as new developments in hardware.
- **Communications and Messaging:** the architecture needs efficient and fast communications architecture so to ensure faster communication between the various processes which are running in their individual address spaces as well as with varied security levels. Poor communications implement would negatively impact on the system's performance [2].

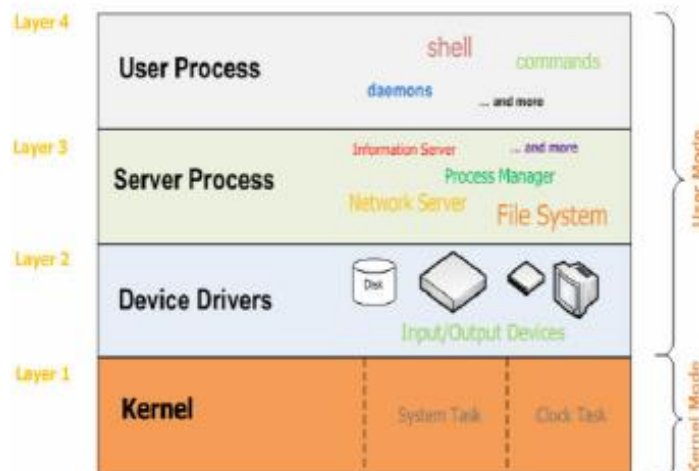


Figure 7: Minix Layered Micro Kernel Architecture

1.2 Memory Management in Minix 3.0

1.2.1 Overview

In MINIX 3 Memory management is simple: paging is not used at all. Paging means that the process's Physical address space can be noncontiguous; process is allocated physical memory whenever the latter is available.

MINIX 3 memory management does not include swapping either. **Note:** Swapping code is available in the complete source and for sure could be activated to make MINIX 3 work on a system with limited physical memory, but in practice, memories are so large now so swapping is rarely needed.

User-space server designated the process manager (PM); The process manager handles THE system calls that are related to process management. MINIX 3 process manager handles querying the real time clock and setting. In MINIX 3 the process management and memory management “the two functions” are merged into one process. **Note:** The PM is not part of the kernel. The actual setting of memory maps for processes is done by the system task within the kernel. As a result for the spilt it is easy to change the memory management policy (algorithms) without any need to modify the lowest layers of the operating system. Most of the PM code handles the MINIX 3 system calls which involve creating processes, primarily fork and exec, rather than just manipulating lists of processes and holes [1].

1.2.2 Memory Layout

In MINIX 3, the default is to compile programs to use separate I and D space but it may compile combined I and D space which means all parts of the process (text, data, and stack) share block of memory which is allocated and released as one block (as default for the original version of MINIX).

Normally MINIX 3 operation memory is allocated on two occasions. First, when a process forks, the amount of memory that are needed by the child is allocated. Second, when a process changes its memory image via the exec system call, the space occupied by the old image is automatically returned to the free list as a hole, then the memory is allocated for the new image; “new image may be in part of memory different from the released one” [1].

Disadvantages of combined:

Doing it this way is not trivial. Consider

the possible error condition that there

is not enough memory to perform

an exec.

What has been declared so far applies for programs which have been compiled with combined I and D space.

Programs with separate I and D space take great advantage of an enhanced mode of memory management

which is called **shared text**. When a process does a fork, only the amount of memory needed for copy of the new process' stack and data is allocated. The parent and the child both of them share executable code in use already by the parent. When such a process does an exec, the process table is searched in order to see if there is another process already using the executable code needed. If one is found, new memory is allocated only for the stack and data, and the text already in memory is shared.

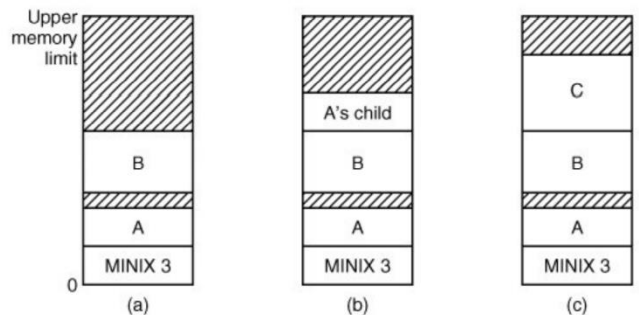


Figure 8: Memory allocation. (a) Originally. (b) After a fork. (c) After the child does an exec.

Shared text complicates termination of a process. Process releases the memory occupied by its stack and data when it terminate. But we need to note that it only releases the memory occupied by its text segment but for sure after a search of the process table reveals that there isn't another current process is sharing that memory. So process may be allocated more memory at starting than releasing when it terminates, if it loaded its own text when it started but that text is being shared by one or more other processes when the first process terminates [1].

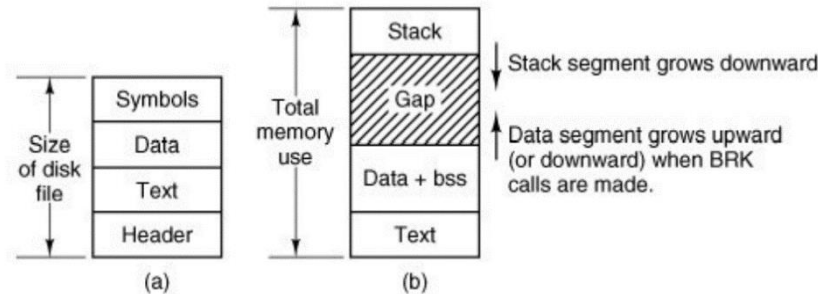


Figure 9: (a) A program as stored in a disk file. (b) Internal memory layout for a single process.

1.2.3 Process Manager Data Structures and Algorithms

Two key data structures are used by the process manager: the process table and the hole table.

In MINIX 3, each of these three pieces (kernel, process manager, file system) of the operating system has its own process table, containing just fields that it needs. With few exceptions, entries correspond exactly, to keep things simple. So slot k of the PM's table refers to the same process as slot k of the file system's table. Also when a process is created or destroyed, all three parts update their tables in order to reflect the new situation, to keep them synchronized.

Note: The exceptions are processes which are not known outside of the kernel, because they are compiled into the kernel, like the SYSTEM tasks and CLOCK, or because they are place holders like KERNEL, and IDLE [1].

1.2.3.1 Message Handling

In MINIX 3, the process manager is message driven. After the system has been initialized, PM enters its main loop that consists of waiting for a message, carrying out request contained in the message, and sending a reply.

Two message categories may be received by the process manager. In high priority communication between the kernel and system servers such as PM, a system notification message is used but the majority of messages are received by process manager result from system calls originated by user processes [1].

1.2.3.2 Processes in Memory

PM's process table is called "mproc" its definition is given in `src/servers/pm/mproc.h`. It contains all the fields related to a process' memory allocation, and some additional items. The most important field is the array `mp_seg`, which has three entries, for the data, text, stack, and segments. Each entry is a structure containing the physical address, virtual address, and length of the segment, all of them are measured in clicks not in bytes. The size of a click For MINIX 3 it is 1024 bytes. All of the segments must start on a click boundary and occupy an integral number of clicks [1].

1.2.3.3 Shared Text

Contents of the stack and data areas belonging to a process may change when the process executes, but the **text does not change**. Memory efficiency is improved by using shared text. When “exec” is about to load a process, it opens the file holding the disk image of the program to be loaded and reads the file header.

- If the process uses separate I and D space, search of mp_dev, mp_ino, and mp_ctime fields in each slot of “mproc” is directly made. These hold device and numbers changed-status times of the images being executed by other processes and i-node.
- If process in memory is found to be executing the same program that is about to be loaded, there is no need to allocate memory for another copy of the **text**. Instead the mp_seg [T] portion of the new process' memory map is initialized to point to the same place where the text segment is already loaded, and only the data and stack portions are set up in a new memory allocation.

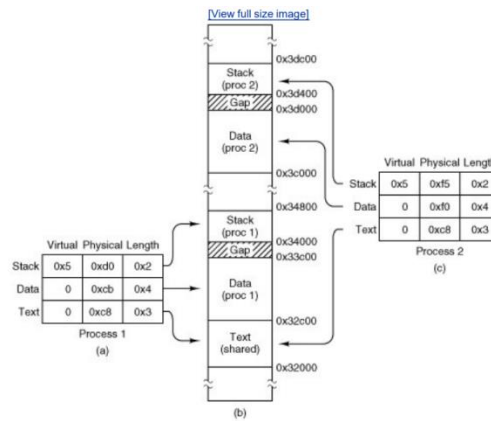


Figure 10: (a) The memory map of a separate I and D space process, as in the previous figure. (b) The layout in memory after a second process starts, executing the same program image with shared text. (c) The memory map of the second process.

1.2.3.4 The Hole List

The other major process manager data structure is the hole table. Hole is defined in src/servers/pm/alloc.c that lists every hole in memory to increasing memory address.

The gaps between the data and stack segments are not considered holes actually because they have already been allocated to processes. So, they are not contained in the free hole list. Each hole list entry has three fields: a pointer to the next entry on the list, the length of the hole, and the base address of the hole. The list is singly linked, because of that it is easy to find the next hole starting from any given hole. But you have to consider that to find the previous hole; you have to search the entire list from the beginning until you come to the given hole. as we said before we record everything about segments and holes in clicks rather than bytes because it is much more efficient [1].

- To allocate memory, hole list is searched, starting at the hole with lowest address, until a hole which is large enough is found (first fit). Then the segment is allocated by reducing the hole by the specific amount needed for the segment, or in the rare case of an exact fit, removing the hole from the list. This scheme is fast and simple but suffers from both a small amount of internal fragmentation.
- Process terminates and it is cleaned up, its stack memory and data are returned back to the free list. If it uses combined I and D, this releases all its memory, but If the program uses separate I and D and a no other process is sharing the text, the text allocation will be returned also. In shared text the text and data

regions are not necessarily contiguous; so two regions of memory may be returned. each region returned, both of the region's neighbors are holes, they are merged, as a result adjacent holes never occur. so, the location, number, and sizes of the holes vary continuously during system operation. But when all user processes have terminated, all of available memory is once again ready for allocation.

Note: This is not necessarily a single hole, since physical memory can be interrupted by regions unusable by the operating system, as in IBM compatible systems where read-only memory (ROM) and memory reserved for I/O transfers separate usable memory below address 640K from memory above 1MB.

1.2.3.5 *Signal Handler*

Signals are described as mechanism to convey information to process which is not necessarily waiting for input.

Three phases of dealing with signals:

- ❖ Preparation: program code prepares for possible signal.
- ❖ Response: signal is received and action is taken.
- ❖ Cleanup: restore normal operation of the process.

In MINIX 3 the preparation phase of signal processing is handled entirely by the PM, since the necessary data structures are all in the PM's part of the process table. We need to mention that a system process, such as the process manager itself, cannot catch signals. System processes use a new handler type "SIG_MESS" that tells PM to forward a signal by means of a "SYS_SIG" notification message.

For signal generation, multiple parts of MINIX 3 system may become involved. The response begins in the PM, which figures out which processes should get the signal using the data structures.

1.2.3.6 *User-Space Timers*

Alarm generation to wake up process after period of time is one of the most common uses of signals. In conventional operating system, alarms are managed entirely by the kernel, or clock driver running in kernel space. In MINIX 3 responsibility for alarms to user processes is by process manager to lighten the kernel's load, and simplify the code that runs in kernel space.

In MINIX 3 timers in kernel space are maintained only for system processes. The process manager maintains another queue of timers for user processes which have requested alarms. Process manager requests an alarm from the clock only for the timer at the head of its queue. When expiration of an alarm is detected after clock interrupt a notification comes to the PM. Then the PM does all the work of checking its signaling user processes own timer queue, and possibly requesting another alarm if there is an active alarm request still at the head of its list [1].

Advantages of this user space timer:

- ❖ First there is possibility that more than one timer may be found to have expired on a particular clock tick. It may improbable that two processes request alarms at the same time. However, although the clock checks for timer expirations on **every interrupt** from the clock chip, interrupts are sometimes disabled. If these are handled by the process manager kernel-space code does not have to traverse its own linked list, cleaning it up and generating multiple notifications.
- ❖ Second, alarms can be cancelled. When this happen it eases the load on the kernel-space code if cancellation of timers is done on a queue maintained by the process manager, and not in the kernel. The kernel-space queue only needs attention when the timer at its head expires or when the process manager makes a change to the head of its queue.

1.3 Process Management

1.3.1 Overview

In MINIX 3 processes follow the general process model. Processes can create sub processes that in turn can also create more sub processes, yielding a tree of processes. All the user processes in the whole system are actually part of a single tree with init at the root, but drivers and Servers are a special case, of course some of them must be started before any user process, including init.

1.3.2 Minix start up

The MINIX 3 bootstrap loads larger program, “boot” that then loads the operating system itself. MINIX 3 boot program looks for specific multipart file on the partition or diskette and loads the individual parts into the memory at proper locations. The most important parts are the kernel (because it includes the system task and the clock task), the file system, and the process manager. At least one disk driver have to be loaded as part of the boot image. Several other programs are loaded in the boot image. These include the console, the RAM disk, reincarnation server, and log drivers, and init.

It should be emphasized that all parts of the boot image are separate programs. After kernel starts the system and clock tasks, process manager and file system have been loaded they cooperate in starting other servers and drivers then many other parts are able to be loaded separately. Finally when all these have run and initialized, they will be blocked, waiting for something to do” waiting for the first user process to be executed”. **Note:** MINIX 3 scheduling prioritizes processes [1].

1.3.2.1 Initialization of the process tree

In MINIX 3, there is few system processes running by the time init gets to run. The tasks CLOCK and SYSTEM that run within kernel are unique processes which are not visible outside it. They receive no PIDs and are not considered part of any tree of processes. The process manager is the first process to run in the user space; it is given PID 0 and is neither a child nor a parent of any other process. The reincarnation server is made the parent of all the other processes started from the boot image [1].

1.3.3 Interprocess Communication in MINIX 3

There are three primitives that are provided for sending and receiving messages. They are called by the C library procedures:

- ❖ Send (dest, &message); to send a message to process dest,
- ❖ Receive (source, &message); to receive a message from process source (or ANY)
- ❖ sendrec (src_dst, &message); to send a message and wait for a reply from the same process

Note: User processes cannot send messages to each other. User processes in layer 4 can initiate messages only to servers in layer 3, servers in layer 3 can initiate messages only to drivers in layer 2.

MINIX 3 uses the rendezvous method to avoid problems of buffering sent, but not yet received, messages. Rendezvous method means both send and receive are blocking (Blocking is considered synchronous and it means; Blocking send -- the sender is blocked until the message is received, Blocking receive -- the receiver is blocked until a message is available)

The advantage of this approach:

- ❖ It is simple and eliminates our need for buffer management (including the possibility of running out of buffers).
- ❖ In addition, all messages are of fixed length determined at compile time, buffer overrun errors, a common source of bugs, are structurally prevented.

There is another important message-passing primitive that It is called by the C library procedure `notify(dest)`; it is used when process needs to make another process aware that has happened something important. A `notify` is nonblocking (means the sender continues to execute whether or not the recipient is waiting) [1].

1.3.4 *Process Scheduling in MINIX 3*

The interrupt system what keeps multiprogramming operating system still going. Processes block if they make requests for an input, allowing another processes to execute. When the input becomes already available, the current running process will be interrupted by the disk, keyboard, or other hardware. Also the clock generates interrupts which are used to ensure that the running user process has not requested input eventually relinquishes the CPU, to give other processes the chance to run. This is the job of lowest layer of MINIX 3 in order to hide these interrupts by turning them into messages. Also when an I/O device completes an operation it directly sends a message to some process, waking it up, moreover making it eligible for running.

Note: In MINIX 3 interruptions due to the clock or message passing or I/O operations occur more frequently than process termination.

The MINIX 3 scheduler uses a multilevel queuing system. Sixteen queues are defined, recompiling to use fewer or more queues is easy. The lowest priority queue is used by the IDLE process only that runs when there isn't anything else to do. User processes by default start in a queue several levels higher than the lowest one. Servers normally scheduled in queues with higher priorities than allowed for the user processes. Drivers in queues with higher priorities than those of the servers, and system tasks and the clock are scheduled in the highest priority queue [1].

Note: Not all the sixteen are available queues to be in use at any time. Processes are started in only few of them. Process may be moved to a different priority queue (in certain limits) by a user who invokes the `nice` command or by the system. The extra levels are available for experimentation, and for additional drivers are added to MINIX 3 the default settings can be adjusted for best performance.

Moreover to the priority that is determined by the queue on which a process is placed, there is another mechanism is used to give some processes edge over other processes. The quantum, the time interval allowed before a process is preempted, is not the equivalent in amount for all processes. User processes have low quantum, but Drivers and servers normally should run until they block. So, they are made preemptable, they are given a large quantum. Also they are allowed to run for a large but finite number of clock ticks, but if they use their entire quantum they are preempted in order not to hang the system [1].

Processes are scheduled using slightly modified round robin. If a process didn't use its entire quantum and it becomes unready, so it blocked waiting for I/O, when it becomes ready again it will be put on the head of the queue(to give user processes quick response to I/O), but with the left-over part of its previous quantum only, but normally the process that became unready because it used its entire quantum is placed at the end of the queue "pure round robin fashion".

Note: Picking a process to run, scheduler checks if any processes are queued in highest priority queue. If one or more are ready, the one at the head of the queue will run, but if none is ready the next lower priority queue is tested similarly, and so on.

All high priority processes should complete whatever work was requested of the servers. Process will then block and there is nothing to do until user processes get a turn to run and make more requests. In case no

process is ready, so the IDLE process is chosen. This puts the CPU in low-power mode until next interrupt occurs.

At each clock tick, there is a check made to make sure if the current process has been running for more than its allotted quantum. If this happened, the scheduler moves this process to the end of its queue. Then the next process to run is picked, as illustrated before. If there are no processes on higher-priority queues and the previous process is alone on its queue it will run again immediately, or the process at the head of the highest priority nonempty queue will run next. But if something goes wrong in the servers their priority temporarily lowered to prevent the system coming to a total standstill [1].

1.3.5 *The System Task in MINIX 3*

In MINIX 3; components of operating system run in user space, although they have special privileges like the system processes. We will still use name "system call" for any of the POSIX-defined system calls (and a few MINIX extensions) but user processes do not request services of the kernel directly. In MINIX 3 user processes' system calls are transformed into messages to server processes. Then Server processes communicate with each other, with the device drivers, and with kernel by messages [1].

The system task, receives all the requests for kernel services. We can call these requests system calls, but to be more exact we are going to refer to them as kernel calls. Kernel calls can't be made by user processes. In many cases system call that originates with a user process results in a kernel call with similar name is made by a server. This is because some part of the service being requested can be dealt only with by the kernel. To overview the System Task; It accepts 28 kinds of messages, Each one of them can be considered a kernel call, although, in some cases there are multiple macros defined with different names that all result in just one of the message types. In some other cases more than one of the message types is handled by a single procedure that does the work [1].

1.3.6 *The Clock Task in MINIX 3*

Clocks (timers) are essential to the operation of any timesharing system for a variety of reasons. Advantages: they prevent one process from monopolizing the CPU.

The MINIX 3 clock task it is driven by interrupts generated by a hardware device. The clock is neither a block device, like a disk, nor a character device, like a terminal. In fact, in MINIX 3 an interface to the clock is not provided by a file in the /dev/ directory, the clock task executes in kernel space and cannot be accessed by user-space processes directly. It has access to all data and kernel functions, but user-space processes can only access it by the system task. There are two types of clocks used in computers: hardware clock and software clock, and both are quite different from the clocks and watches used by people. All the clock hardware is generate interrupts at known intervals. Everything else involving time must be done by the software, the clock driver. The exact duties of the clock driver are different among operating systems:

1. Maintaining the time of day.
 2. Preventing processes from running longer than they are allowed to.
 3. Accounting for CPU usage.
 4. Handling the alarm system call made by user processes.
 5. Providing watchdog timers for parts of the system itself.
 6. Doing profiling, monitoring, and statistics gathering.
- The first clock function, maintaining the time of day (also called the real time) is not difficult. It only requires incrementing the counter at each clock tick. The only thing to watch for is the number of bits in time-of-day counter, using a clock rate of 60 Hz, a 32-bit counter will overflow in just over 2 years. The system cannot store the real time as the number of ticks since Feb. 1, 1970 in 32 bits. We can solve this problem using three approaches.

- The first way is to use a 64-bit counter, although this makes maintaining the counter more expensive.
 - The second way is to maintain the time of day in seconds, rather than in ticks.
 - The third approach is to count ticks, but to do that relative to the time the system was booted, rather than relative to a fixed external moment.
- The second clock function is preventing processes from running too long.
- The third clock function is doing CPU accounting. To do it accurately starts a second timer, distinct from the main system timer, whenever a process is started.

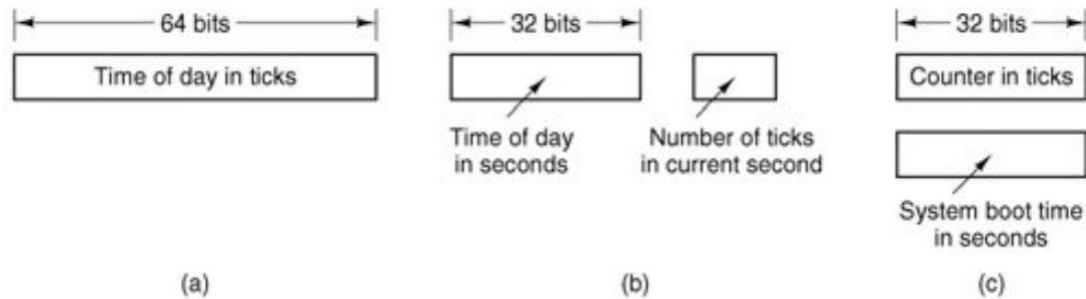


Figure 11: Three ways to maintain the time of day.

In MINIX 3, a process can request which the operating system give it warning after a certain interval. The warning is usually a interrupt, signal, message. For example: One application requiring such warnings is networking, another example is computer-aided instruction. If clock driver had enough clocks, it can set a separate clock for each request, and it must simulate multiple virtual clocks with single physical clock. There is only one way, is to maintain a table that the signal time for all pending timers is kept, and a variable giving time of the next one. If the time of day is updated, the driver checks to see if closest signal has occurred. If it has, table is searched for next one to occur. If there are many signals are expected, it is very efficient to simulate multiple clocks by chaining all the pending clock requests together, sorted on time, in a linked list. Each entry on the list tells us how many clock ticks following the previous one to wait before causing a signal [1]

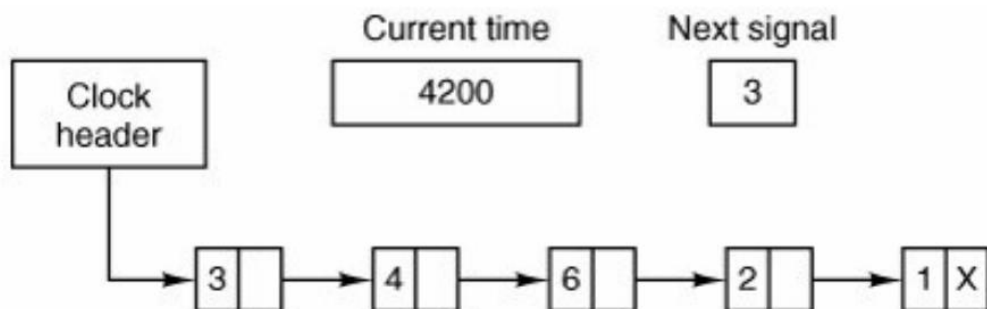


Figure 12: Simulating multiple timers with a single clock

In Figure [12], a timer has just expired. The next interrupt occurs in 3 ticks, and 3 has just been loaded. On every tick, the Next signal is decremented. When it gets to 0, signal corresponding to the first item on the list is caused, and that item is removed from the list. After that the Next signal is set to value in the entry now at the head of the list. Here is 4. The usage of absolute times rather than relative times is more convenient in many cases, and which can be approached by MINIX 3. **Note:** during clock interrupt, the clock driver has several things to do. These things include decrementing the quantum, incrementing the real time, and checking for 0, doing the CPU accounting, and also decrementing the alarm counter. Each of these operations has been carefully arranged to be very fast in order to they have to be repeated many times a second [1].

Note: Parts of operating system need to set timers. These parts are called watchdog timers.

Note: In the Clock task the main loop of clock task accepts only a single kind of message, `HARD_INT` that comes from interrupt handler. Anything else is an error. Moreover, it does not receive this message for every clock tick interrupt; despite subroutine called each time a message is received is named `do_clocktick`. A message is received, `do_clocktick` only is called if process scheduling is needed or a timer has expired.

1.3.6.1 *The Clock Interrupt Handler*

Interrupt handler runs every time counter in the clock chip reaches zero and generates an interrupt. This is where the basic timekeeping work is done. In MINIX 3 in `clock.c` only the counter for ticks since boot is maintained; records of the boot time are kept elsewhere. The clock software supplies only the current tick count to aid a system call for the real time. Then Processing is done by one of the servers. This is consistent with the MINIX 3 strategy of moving functionality to processes that run in user space [1].

1.3.6.2 *Clock Driver*

The MINIX 3 clock driver is contained in the file `kernel/clock.c`. It has three functional parts. First, like the device, task mechanism which runs in a loop, dispatching to subroutines that perform the action requested in each message and waiting for messages.

1.3.6.3 *Watchdog Timers*

Ordinarily are thought of as user-supplied procedures that are part of the user's code and are executed when the timer expires, but this cannot be done in MINIX 3. But we can use a synchronous alarm to bridge the gap from the kernel to user space.

This is a good time to explain what is meant by a synchronous alarm. A signal may arrive or a conventional watchdog may be activated without any relation to what part of a process is currently executing, so these mechanisms are asynchronous.

Watchdog timers take advantage of “timer_t” type “s_alarm_timer” field which exists in each element of the “priv table”. Each process in the system has slot in the “priv table” in order to set a timer, system process in user space makes “sys_setalarm call” that is handled by the system task. System task is compiled in the kernel space, so it can initialize a timer on behalf of the calling process. Initialization entails by putting the address of procedure to execute when timer expires into the correct field, After that inserting the timer into a the list of timers, as shown in Figure 11. The procedure for executing has to be in the kernel space too. The system task contains watchdog function, “cause_alarm” that generates notify when it goes off, that causes a synchronous alarm for the user [1].

This alarm is able to invoke the user-space watchdog function. Within kernel binary this is true watchdog, but for process which requested the timer, it is synchronous alarm. But It is not the same as having timer execute procedure in target's address space. There is an overhead bit more, but it is simpler than interrupt.

We have just written that the system task can set alarms on behalf of some user-space processes. The mechanism just described only works for the system processes. Each system process has copy of the “priv” structure, but single copy only is shared by all the user (non system) processes. The parts of “priv table” that cannot be shared, because the bitmap of pending timer and the notifications are not usable by the user processes.

The solution for this is that: the process manager manages timers on behalf of user processes in way similar to way of the system task manages timers for system processes. Each process has a “timer_t” field of its own in process manager's part of process table. When user process makes alarm system call for asking for an alarm to be set, it is handled by process manager that sets up the timer and also inserts it into its list of timers. The process manager asks system task to send it notification when first timer in the PM's list of timers is scheduled to expire. The process manager has to ask for help only when head of its chain of timers changes, either because the first timer has expired or has been cancelled, or because there is a new request has been received which must go on the chain before the current head. This is used to support the POSIX-standard alarm system call. Procedure to execute is within address space of the process manager. When executed, the user process which requested the alarm is sent a signal, rather than a notification [1].

1.3.6.4 Millisecond Timing

Procedure is provided in clock.c which provides microsecond resolution timing. The short Delays “few microseconds” may be needed by various I/O devices. No practical way can be done using the message passing interface and alarms. Counter which is used for generating clock interrupts can directly be read. It is decremented every 0.8 microseconds approximately, also reaches zero 60 times a second, or every 16.67 milliseconds. This function is used as a source of randomness only for the random number generator [1].

1.3.6.5 Summary of Clock Services

There are several functions declared PUBLIC that can be called from kernel or system task. The other services are available indirectly only, by system calls ultimately handled by the system task. And the Other system processes can ask directly the system task, but the user processes have to ask the process manager, that also relies on the system task.

Service	Access	Response	Clients
get_uptime	Function call	Ticks	Kernel or system task
set_timer	Function call	None	Kernel or system task
reset_timer	Function call	None	Kernel or system task
read_clock	Function call	Count	Kernel or system task
clock_stop	Function call	None	Kernel or system task
Synchronous alarm	System call	Notification	Server or driver, via system task
POSIX alarm	System call	Signal	User process, via PM
Time	System call	Message	Any process, via PM

Figure13: The time-related services supported by the clock driver

The kernel or the system task can get current uptime, or reset or set timer without the overhead of a message. The system task or the kernel can also call read_clock, that reads the counter in the timer chip, to get time in units of approximately 0.8 microseconds. The clock_stop function is intended to be called only when MINIX 3 shuts down. It restores the BIOS clock rate. A system process, either a driver or a server, is able to request a synchronous alarm that causes activation of a watchdog function in kernel space and a notification to requesting process. A POSIX-alarm is requested by user process by the way of asking the process manager that then asks

the system task to activate a watchdog. When the timer expires, system task notifies the process manager, and also the process manager delivers a signal to the user process.

1.4 File System in MINIX 3

1.4.1 Overview

The file system in MINIX 3 is just a big C program which runs in the user space figure[1]. For reading and writing files, user processes send messages to the file system to tell it what they want done. So the file system does the work after that sends back a reply. File system is, in fact, is a network file server that happens to be running on the same machine as the caller. The design of the file system in minix has some important implications: the file system can be almost modified, experimented with, and tested completely independently of the rest of MINIX 3. Moreover it is very easy to move file system to any computer that has a C compiler, and compile it there; also you can use it as a free-standing UNIX-like remote file server. Only changes which need to be made are how messages are received and sent, which differs from system to system [1].

1.4.2 Messages

File system accepts 39 types of messages all of them requesting work. All but only two are for MINIX 3 system calls. These two exceptions are messages that generated by other parts of MINIX 3. From the system calls, already 31 are accepted from user processes. However, six system call messages are for system calls that are handled first by process manager, which then calls file system for doing part of the work. There are two other messages are also handled by the file system. Figure [14] shows the messages. File system's structure is basically the same as that of process manager and all I/O device drivers. It has a main loop which waits for a message to arrive. When the message arrives, the message's type is extracted and used as an index into a table which contains pointers to the procedures within the file system which handle all the types. After that the appropriate procedure is called, the procedure does its work then returns a status value. Then the file system sends a reply back to caller and goes back to top of the loop to wait for a next message [1].

1.4.3 File System Layout

MINIX 3 file system is logical, self-contained entity with i-nodes, data blocks, and directories. It is able to be stored on any block device, as hard disk or floppy disk partition. In all of the cases, the layout of file system has same structure. Figure [15] shows the layout for small hard disk or floppy disk partition with a 1-KB block size and 64 i-nodes. In this example, the zone bitmap is only one 1-KB block, it can keep track of no more than 8192 1-KB zones (blocks), so this limit the file system to 8 MB. Even for floppy disk, 64 i-nodes only puts severe limit on the number of the files, as a result rather than the four blocks reserved for i-nodes in the figure, more would be used properly. Reserving eight blocks for i-nodes would be practical more but our diagram would not look nice like now. For modern hard disk, both zone

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
setuid	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)

Figure 24: File system messages

bitmaps and i-node will be much larger than 1 block. Relative size of the various components in the figure can vary from file system to another file system, that depend on their sizes, how many files are allowed maximum, and many others. But all components are always present and in the same order [1].

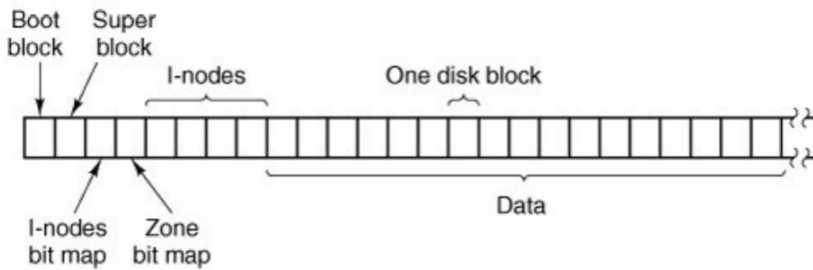


Figure 15: Disk layout for a floppy disk or small hard disk partition

As shown in figure [15], each file system begins with boot block that contains executable code. Boot block's size is always 1024 bytes (which is two disk sectors), even MINIX 3 can (and by default does) use larger block size. When computer is turned on, hardware reads boot block from boot device into the memory, and begins executing its code. Boot block code begins the process by loading operating system itself. Once the system has been booted, boot block is not used any more. Not every disk drive can be used as a boot device, but we do this to keep the structure uniform; every block device has a block reserved for boot block code. At the worst case this strategy wastes one block. For preventing the hardware from trying booting an unbootable device, a magic number is placed at known location in boot block; it is used only when executable code is written to the device. When booting from a device, hardware (the BIOS code) will refuse to attempt to load from a device lacking the magic number. By doing this we prevent inadvertently using garbage as a boot program [1].

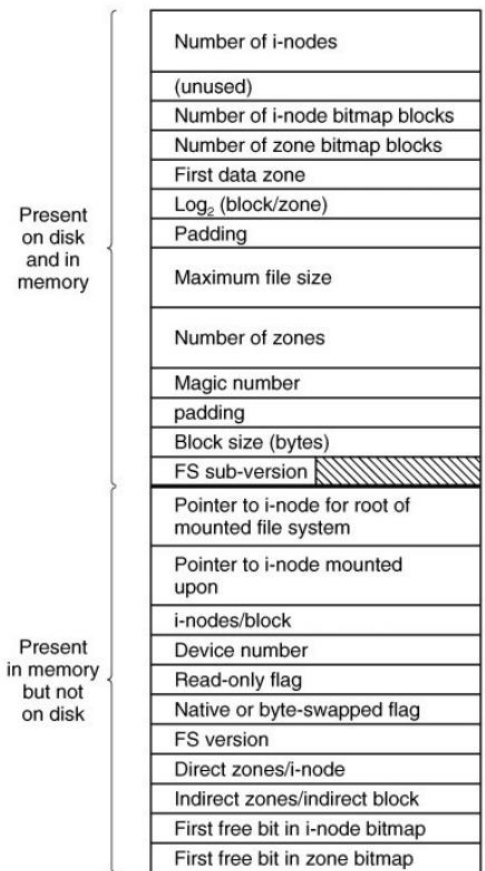


Figure 16: The MINIX 3 superblock

1.4.3.1 Superblock

Superblock contains information which describes the layout of file system. Like the boot block, the superblock is always 1024 bytes, regardless of block size which is used for the rest of the file system. It is illustrated in figure [16]

Main function of the superblock is to tell the file system how big the various pieces of the file system are. The block size and the number of i-nodes are given, so it is easy to calculate size of the i-node bitmap and also the number of blocks of inodes. As an example, for a 1-KB block, each block of bitmap has 1024 bytes (8192 bits), so it can keep track of the status of up to 8192 inodes. (in fact, the first block can handle only up to 8191 i-nodes, since there is no 0th i-node, but it is given a bit in the bitmap). For 10,000 i-nodes, two bitmap blocks are needed. Since i-nodes each occupy 64 bytes, a 1-KB block holds up to 16 i-nodes. With 64 i-nodes, four disk blocks are needed to contain them all [1].

The difference between zones and blocks we be explained in detail later, but for now we need to say that disk storage can be allocated in units (zones) of 1, 2, 4, 8, or in general 2^n blocks. The zone bitmap keeps track of free storage in zones, not the blocks. All standard disks used by MINIX 3 the zone and block sizes are both the

same (by default 4 KB), so for first approximation zone is the same as a block on these devices. Until we come to the details of storage allocation later, it is adequate to think "block" whenever you see "zone." **Note:** the number of blocks per zone is not stored in the superblock, because it is never needed. But all that is needed is the base 2 logarithm of the zone to block ratio that is used as shift count to convert zones to blocks and vice versa. We can take example, with 8 blocks per zone, $\log_2 8 = 3$, so to find the zone containing block 128 we shift 128 right 3 bits to get zone 16 [1].

Zone bitmap includes the data zones only (blocks used for the i-nodes and bitmaps are not in the map), with first data zone designated zone 1 in the bitmap. As with the i-node bitmap, bit 0 in the map is unused, because of that the first block in zone bitmap can map 8191 zones and subsequent blocks can map 8192 zones each. If we examine bitmaps on newly formatted disk, it will be found that both the zone bitmaps and i-node have 2 bits set to 1. There is one is for the nonexistent 0th i-node or zone and the other is for i-node and zone that are used by the root directory on the device that is placed there when file system is created. Information in the superblock is redundant because sometimes it is needed in one form and sometimes in another. With 1 KB devoted to the superblock, we can compute this information in all the forms it is needed, rather than we have to recompute it frequently during execution. Zone number of the first data zone on the disk, can be calculated from the zone size, block size, number of i-nodes, and the number of zones, but it is faster just to keep it in the superblock. The rest of the superblock is wasted anyhow, so using up another word of it costs nothing [1].

At the time when MINIX 3 is booted, the superblock for the root device is read into the table in memory. Same, as other file systems that are mounted; their superblocks are also brought into memory. The superblock table holds a number of fields which not present on the disk. These include flags which allow device to be specified as read-only or as following a byte-order convention opposite to the standard, and fields for speeding the access by indicating points in bitmaps below which all bits are marked used. Moreover, there is field that describes the device from which the superblock came. Before disk can be used as a MINIX 3 file system, it must be given the structure of figure [3]. The utility program mkfs has been provided to build file systems. This program can be called either by command like "mkfs /dev/fd1 1440" for building empty 1440 block file system on floppy disk in drive 1, or it can be given a prototype file listing directories and files for including it in the new file system. Also this command puts a magic number in the superblock for identifying the file system as a valid MINIX file system. The MINIX file system has evolved, and some aspects of file system (for example, the size of inodes) were different previously [1].

The magic number importance that it identifies the version of "mkfs" that created the file system, so differences can be accommodated. Attempts to mount a file system not in MINIX 3 format, such as an MS-DOS diskette, will be rejected by the mount system call that checks the superblock for a valid magic number and other things.

1.4.4 *Bitmaps*

MINIX 3 keeps tracks of which i-nodes and zones are free by the usage two bitmaps. When file is removed, so it is a simple matter to calculate which block of the bitmap contains the bit for the i-node that has been freed and to find it by the usage of the normal cache mechanism. Once the block is found, the bit corresponding to freed i-node will be set to 0. Zones are released from zone bitmap by the same way.

When a file is to be created, file system must search in bit-map blocks one at a time for the first free i-node. Then this i-node is allocated for the new file. In fact, inmemory there is copy of the superblock has a field that points to the first free i-node, in fact, as a result no search is necessary until a node is used, when the pointer must be updated to point to the new next free i-node that will often turn out to be the next one, or a the closest one. The same happens, when an i-node is freed, a check is made to see if the free i-node comes before the currently-pointed-to one, and the pointer is updated if necessary. If every i-node slot on the disk is full, search routine returns a 0, that clarifies why i-node 0 is not used (it can be used to indicate the search failed). (When "mkfs" creates a new file system, it zeroes i-node 0 and sets the lowest bit in bitmap to 1, so file system will never attempt to allocate it. Everything which has been said here about the i-node bitmaps also is applied to the

zone bitmap; it is searched for the first free zone when space is needed, but a pointer to the first free zone is maintained in order to eliminate most of the need for sequential searches through the bitmap [1].

Now we can now explain the difference between zones and blocks. The idea behind zones is to ensure that disk blocks that belong to the same file are located on the same cylinder, in order to improve performance when the file is read sequentially. The approach chosen is to make it possible to allocate several blocks at a time. As an example, block size is 1 KB and the zone size is 4 KB, the zone bitmap keeps track of zones, not blocks. A 20-MB disk has 5K zones of 4 KB, hence 5K bits in its zone map. Most of the file system works with blocks. Transfers of the Disk are always a block at a time, and the buffer cache also works with individual blocks. Only few parts of the system which keep track of physical disk addresses (the i-nodes and the zone bitmap) know about zones. Some of the design decisions had to be made while developing the MINIX 3 file system [1].

1.4.5 I-Nodes

The layout of the MINIX 3 which is given in Figure [17], is almost the same as a standard UNIX i-node. The disk zone pointers are 32-bit pointers, and there are 9 pointers only, which are 7 direct and 2 indirect. The MINIX 3 i-nodes occupy 64 bytes, and there is space available for a 10th (triple indirect) pointer. The MINIX 3 i-node access, modification time and i-node change times are standard. The last of these is updated for almost every file operation except a read of the file.

When a file is opened, its i-node is located and brought into the inode table in memory, where it remains until the file is closed. The inode table has few additional fields doesn't present on the disk, as the i-node's device and number, as a result the file system knows where to rewrite the i-node if it is modified while in memory. Also it has a counter per i-node. If same file is opened more than once, one copy only of the i-node is kept in memory, but the counter is incremented each time the file is opened and also decremented each time the file is closed. Only when the counter reaches zero finally, the i-node is removed from the table. If it has been modified since being loaded into memory, also it is rewritten to the disk [1].

Main function of a file's i-node is to tell where the data blocks are. The first seven zone numbers are given right in the i-node itself. In the standard distribution, zones and blocks both 1 KB. **Note:** Files up to 7 KB do not need indirect blocks, but beyond 7 KB, indirect zones are needed, by the usage of the scheme of Figure [18], except that the single and double indirect blocks are only used With 1-KB blocks and zones and 32-bit zone numbers, a single indirect block holds 256 entries, representing a quarter megabyte of storage. The double indirect block points to 256 single indirect blocks, giving access to up to 64 megabytes. With 4-KB blocks, the double indirect block leads to 1024 x 1024 blocks, which is over a million 4-KB blocks, making the maximum file size over 4 GB. In practice the use of 32-bit numbers as file offsets limits the maximum file size to 232 1 bytes. As a result for these numbers, when 4-KB disk blocks are used MINIX 3 has no need for the triple indirect blocks; because the maximum file size is limited by the pointer size, not the ability to keep track of enough blocks [1].

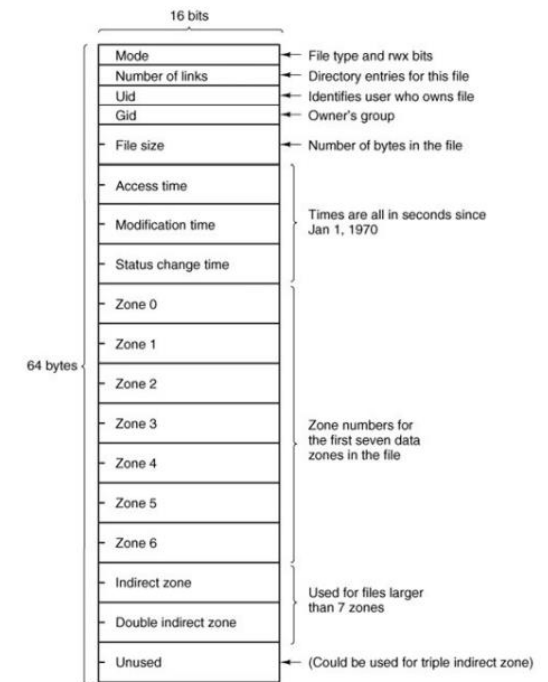


Figure 17: The MINIX i-node

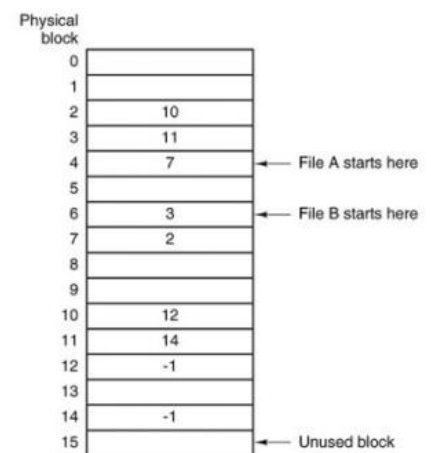


Figure 18: Linked list allocation using a file allocation table in main memory

Also the i-node holds mode information that tells what is the kind of the file is it (regular, pipe, block special, character special, or directory), and it gives the protection and SETGID bits and SETUID. Link field in the i-node records how many directory entries point to i-node, as a result file system knows when to release the file's storage. Also this field should not be confused with the counter (which present only in the inode table in memory, not on the disk) which tells how many times the file is currently open, typically by different processes. **Note:** i-nodes, which we mentioned its structure may be modified for special purposes. For example example used in MINIX 3, is the i-nodes for block and character device special files. These do not need zone pointers, as they don't have to reference data areas on the disk. The minor and major device numbers are stored in Zone-0 space. In another way that an i-node could be used, although not implemented in MINIX 3, is as an immediate file with a small amount of data stored in the i-node itself [1].

1.4.6 *The Block Cache*

MINIX 3 uses block cache in order to improve the file system performance. The cache is implemented as a fixed array of buffers, each consist of a header containing pointers, flags, and counters, and a body with room for one disk block. All of the buffers which are not in use chained together in a double-linked list, from most recently used (MRU) to least recently used (LRU) as illustrated in figure [19].

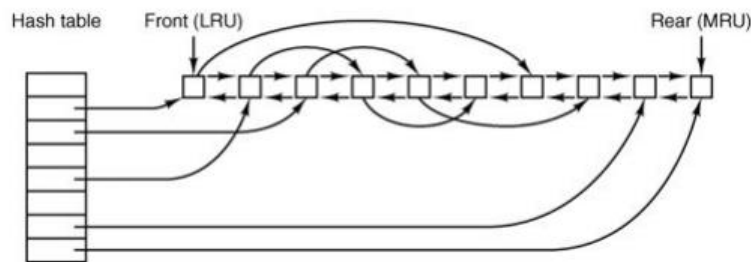


Figure 19: The linked lists used by the block cache.

To quickly determine if a given block is in the cache or not, a hash table is used. All of the buffers contains a block that has hash code k are linked together on a single-linked list pointed to by entry k in the hash table. The hash function is just to extract the low-order n bits from the block number, so blocks from different devices appear on the same hash chain. Every buffer is on one of these chains. After the initialization, MINIX 3 is booted, and all buffers are unused, and all are in a single chain pointed to by the 0th hash table entry. At that time all the other hash table entries contain a null pointer, but once the system starts, buffers will be removed from the 0th chain and other chains will be built [1].

When file system needs to acquire a block, it calls a procedure, `get_block` that computes the hash code for that block and searches the appropriate list. `Get_block` is called with a device number the same as the block number, and the search compares both numbers with the corresponding fields in the buffer chain. If a buffer containing the block is found, a counter in the buffer header will be incremented to show that the block is now in use, and a pointer to it is returned. If block is not found on the hash list, first buffer on the LRU list can be used; it is guaranteed not to be still in use, and block which it contains can be evicted to free up the buffer. Once a block has been chosen for eviction from the block cache, and there is another flag in its header is checked to see if the block has been modified since being read in. If it is rewritten to the disk. In this point the block needed is read in by the way of sending a message to the disk driver. File system is suspended until the block arrives, at which time it continues and a pointer to the block is returned to the caller [1].

When the procedure that requested the block has completed its job, it will call another procedure, "`put_block`", in order to to free the block. And automatically, a block will be used immediately and then released, but because it is possible that additional requests for a block will be made before it has been released,"`put_block`" will

decrement the counter's use and puts the buffer back onto the LRU list only when the use counter goes back to zero. But while the counter is nonzero, the block remains in limbo [1].

1.4.7 Directories and Paths

Another subsystem in the file system manages path names and directories. A lot of system calls, like open, have a file name as a parameter. What is needed only is the i-node for that file, so it is up to the file system to look up for the file in the directory tree and locate its inode. In MINIX 3 the V3 file system provides 64 bytes directory entries, with 4 bytes for the inode number and 60 bytes for the file name. So having up to 4 billion files per disk partition is effectively infinite. The MINIX 3 scheme is very simple; also it is very fast for both storing new ones and looking up names, since no heap management is required [1].

The usual configuration for MINIX 3 is to have a small root file system containing the files needed to start the system and for doing the basic system maintenance, and also to have the majority of the files, including users' directories, on a separate device mounted on /usr. Now we need to look at how mounting is done. When the user types the command `mount /dev/c0d1p2 /usr` on the terminal, the file system contained on hard disk 1, partition 2 is mounted on top of /usr/ in the root file system. The file systems before and after mounting are shown in Figure [20].

The key to the whole mount business is the flag which is set in the memory copy of the i-node of /usr after a successful mount. This flag indicates that the i-node is mounted on. The mount call loads the super-block also for the newly mounted file system into the super_block table and sets two pointers in it. Moreover, it puts the root i-node of the mounted file system in the inode table. In the super-blocks in memory contain two fields related to mounted file systems. First of these, is the i-node-for-root-of-mounted-file-system, which is set to point to the root i-node of the newly mounted file system. Second, is the i-node-mounted-upon, which is set to point to the i-node mounted on, in this case, the i-node for /usr. These two pointers serve to connect the mounted file system to the root and represent the "glue" which holds the mounted file system to the root [as shown in Figure[20]] This glue is what makes mounted file systems work [1].

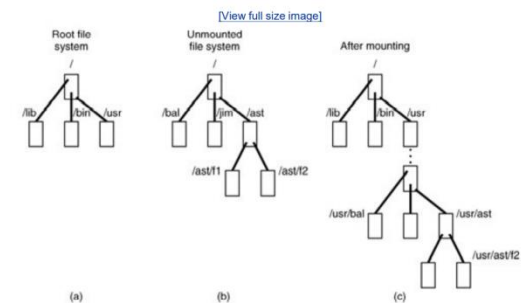


Figure 20: (a) Root file system. (b) An unmounted file system. (c) The result of mounting the file system of (b) on /usr/.

Note: When path such as /usr/ast/f2 is being looked up, the file system sees a flag in the i-node for /usr/ and it must continue searching at the root i-node of the file system mounted on /usr/; by searching all the superblocks in memory until it finds the one which its i-node mounted on field points to /usr/. This must be the superblock for the file system mounted on /usr/. And once it has the superblock, it will easily follow the other pointer to find the root i-node for the mounted file system. Now the file system can continue searching [1].

1.4.8 File Descriptors

Once file has been opened, file descriptor is returned to the user process for use in subsequent read and write calls. Like the process manager and the kernel, the file system maintains part of the process table within its address space. Three of its fields are of interest. The first two are pointers to the i-nodes for the working directory and the root directory. Path searches always begin at one or the other, depending on whether the path is absolute or relative. These pointers are changed by the chroot and chdir system calls to point to the new root or new working directory, respectively [1].

The third interesting field in the process table is an array indexed by file descriptor number. It is used to locate the proper file when a file descriptor is presented. In Minix 3, it is not possible to put the file position in the process table. So we introduce a new, shared table, file that contains all the file positions Figure [21]. By having the file position truly shared, the semantics of fork can be implemented correctly, and shell scripts will work properly [1].

Although the only thing that the filp table really has to contain is the shared file position, it is more convenient to put the i-node pointer there. By this way, all the file descriptor array in the process table contains is a pointer to a filp entry. Also The filp entry contains the file mode (permission bits), some flags indicating if the file was opened in a special mode, and a count of the number of processes using it, so the file system can tell when the last process using the entry has terminated, to reclaim the slot.

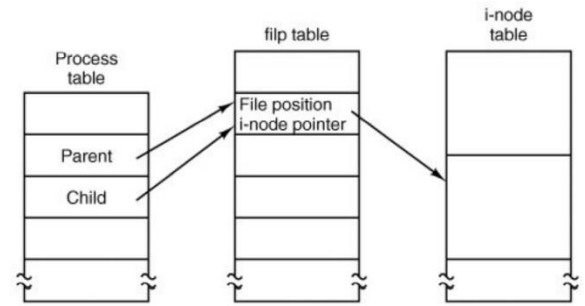


Figure 21: How file positions are shared between a parent and a child

1.4.9 File Locking

File locking in the file system management requires a special table. MINIX 3 supports the POSIX interprocess communication mechanism of advisory file locking. This permits any part, or many multiple parts, of a file to be marked as locked. Operating system does not enforce locking, but the processes are expected to be well behaved and to look for locks on a file before doing anything that would conflict with another process. A single process can have more than one lock active, and different parts of a file may be locked by more than one process (although, , the locks cannot overlap), so neither the process table nor the filp table is a good place to record locks. Since a file may have more than one lock placed upon it, the i-node is not a good place either. MINIX 3 uses another table, file_lock table, in order to record all locks. Each slot in this table has space for a lock type, indicating if the file is locked for reading or writing, the process ID holding the lock, a pointer to the i-node of the locked file, and the offsets of the first and last bytes of the locked region [1].

1.4.10 Pipes and Special Files

Pipes and special files differ from ordinary files, when a process tries to read or write a block of data from a disk file, it is almost certain that the operation will complete at most in a few hundred milliseconds, and worst case is that two or three disk accesses might be needed not more than that. When reading from a pipe, the situation is different: if the pipe is empty, the reader must wait until some other process puts data in the pipe that might take hours. The same happens when reading from a terminal, a process must wait until somebody types something.

As a result, the file system's normal rule of handling a request until it is finished does not work. It is necessary to suspend these requests and restart them later. When a process tries reading or writing from a pipe, the file system check the state of the pipe immediately to see if the operation can be completed. If it can be, it is, but if it cannot be, then the file system records the parameters of the system call in the process table, as a result it can restart the process when the time comes. **Note:** the file system doesn't need to take any action to have the caller suspended. It has only to refrain from sending a reply, leaving the caller blocked waiting for the reply. So after suspending a process, the file system goes back to its main loop waiting for the next system call. When another process modifies the pipe's state the suspended process can complete, the file system sets a flag so that next time through the main loop it extracts the suspended process' parameters from the process table and executes the call [1].

The situation with terminals and other character special files is slightly different. The i-node for each special file contains two numbers, the major device and the minor device. The major device number indicates the device class (RAM disk, floppy disk, hard disk, terminal), and It is used as an index into a file system table which maps it onto the number of the corresponding I/O device driver. In fact the major device determines which I/O

driver to call. The minor device number is passed to the driver as a parameter. It specifies which device is to be used, for example, terminal 2 or drive 1.

In some cases, most notably terminal devices, the minor device number encodes some information about a category of devices handled by a driver. In MINIX 3 devices, `ttyp0`, `ttyp1`, etc., are assigned device numbers such as 4, 128 and 4, 129. These pseudo devices each have an associated device, `ptyp0`, `ptyp1`, and so on. The major, minor device number pairs for these are 4,192 and 4,193, etc. These numbers are chosen for making it easy for the device driver to call the lowlevel functions required for each group of devices. It is not expected that anyone is going to equip a MINIX 3 system with 192 or more terminals. When a process reads from a special file, the file system extracts the major and minor device numbers from the file's i-node, and uses the major device number as an index into a file system table in order to map it onto the process number of the corresponding device driver. Once it has identified the driver, the file system sends it a message, including as parameters the minor device, the operation to be performed, the caller's process number and buffer address, and the number of bytes to be transferred [1].

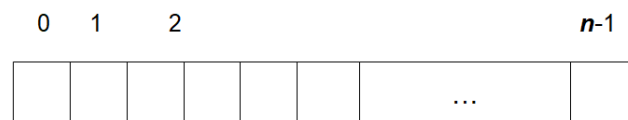
If the driver is able to carry out the work immediately (a line of input has already been typed on the terminal), it copies data from its own internal buffers to the user and sends the file system a reply message saying that the work is done. Then The file system then sends a reply message to the user, and the call is finished. **Note:** the driver does not copy the data to the file system. Data from block devices go through the block cache, but data from character special files do not. But if the driver is not able to carry out the work, so it records the message parameters in its internal tables, and immediately sends a reply to the file system saying that the call could not be completed. At this time, the file system is in the same situation as having discovered that someone is trying to read from an empty pipe. It records the fact that the process is suspended and waits for the next message. When the driver has acquired enough data to complete the call, it transfers them to the buffer of the still-blocked user and then sends the file system a message reporting what it has done. All the file system has to do is send a reply message to the user to unblock it and also report the number of bytes transferred [1].

1.4.11 Free Space Management

The file system always keeps tracks of free disk blocks for allocating space to files when they are created. Moreover, to reuse the space released from deleting the files, free space management becomes important. The system maintains free space list that keeps track of the disk blocks that are not allocated to some file or directory. The free space list can be implemented mainly by one of these 4 methods:

- Bitmap or Bit vector

Bitmap or Bit Vector is a series or a collection of bits where each bit corresponds to a disk block. The bit can take two values: 0 and 1; 0 indicates that the block is allocated and 1 indicates a free block. The given instance of disk blocks/extents on the disk in Figure [22] (where green blocks are allocated) can be represented by a bitmap of 16 bits as: 0000111000000110.



$$\text{bit}[i] = \begin{cases} 1 \Rightarrow \text{block}[i] \text{ free} \\ 0 \Rightarrow \text{block}[i] \text{ occupied} \end{cases}$$

Figure 23: Bitmap or Bit vector representation

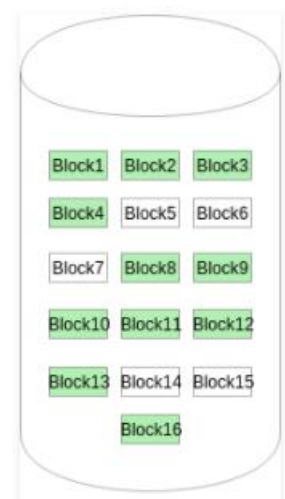


Figure 22: Bitmap or Bit vector

Advantages:

- Finding the first free block is efficient.
- It requires scanning the words (a group of 8 bits) in a bitmap for a non-zero word.
- (A 0-valued word has all bits 0). The first free block is then found by scanning for the first 1 bit in the non-zero word.
- The block number can be calculated as:
- $(\text{number of bits per word}) * (\text{number of 0-values words}) + \text{offset of bit first bit 1 in the non-zero word}$

- Linked List

In this Method, the free disk blocks are linked together, a free block contains a pointer to the next free block. The block number of the very first disk block is stored at a separate location on disk and is also cached in memory.

As shown in Figure [24], the free space list head points to Block 5 which points to Block 6, the next free block and so on. The last free block would contain a null pointer indicating the end of free list.

Advantages:

- No waste of space

Disadvantages:

- The I/O required for free space list traversal.
- Cannot get contiguous space easily

- Grouping

This method stores the address of the free blocks in the first free block. The first free block stores the address of some, for example we have n free blocks. Out of these n blocks, the first n-1 blocks are actually free and the last block contains the address of next free n blocks.

Advantages:

The addresses of a group of free disk blocks can be found easily.

- Counting

This method stores the address of the first free disk block and a number n of free contiguous disk blocks that follow the first block.

Every entry in the list would contain:

- ★ Address of first free disk block
- ★ A number n

For example, in Figure [23], the first entry of the free space list would be: ([Address of Block 5], 2), because 2 contiguous free blocks follow block 5.

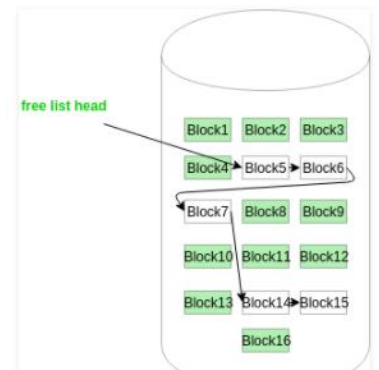


Figure 24: Linked List

To sum up after illustrating the 4 methods which we can use in managing the free space in file system we need to clarify that Minix 3 has chosen the first method Bitmap or Bit vector to manage its free space in file management and this method was chosen specifically for its great and important advantages. **Note:** The detail description of how Minix 3 uses this method can be found in section 1.4.4.

2 REQUIREMENT 2:

“CPU SCHEDULING”

2.1 *LITTLE HISTORY:*

MINIX 3 is a micro-kernel operating system structured in four layers [4]. Device drivers, networking and memory management are examples of services running in layers above the kernel, but until recently scheduling was still handled in-kernel. This IPA explored possibilities of moving scheduling to user mode (Figure 2). There are several motives, most importantly decoupling scheduling policy from kernel. With a user mode scheduler, it is possible to change the scheduling policy without modifying the kernel. Furthermore, it is possible to run multiple user mode schedulers each with their own policy. A user mode scheduler in a multi-core environment can also afford to spend more time evaluating scheduling decisions than the kernel can, since the system (and kernel) keep running on other cores. Long story short, the scheduling was moved to the user space using EVENT-DRIVEN APPROACH which means the scheduler would largely sit idle and only react to event messages sent from kernel. For example, when a process runs out of quantum.

2.2 *How scheduling works now:*

It was found that many popular policies can be implemented through a very simple interface. Two system library routines are exposed to user mode, *sys_schedctl* and *sys_schedule*. These send messages to kernel requesting to start scheduling a particular process and to give a particular process quanta and priority, respectively. Once a process runs out of its quantum, the user mode scheduler is notified with message.

This message contains system and process feedback that the scheduler may use for making scheduling decisions

The kernel defines ‘n’(usually 16) priority queues and implements a simple preemptive round robin scheduler, always choosing the process at the head of the highest priority queue. When a process runs out of quantum, the kernel sets the process's `RTS_NO_QUANTUM` runtime flag, effectively dequeuing the process and marking it as not runnable.

The scheduler is notified when a process runs out of quantum by sending a message to the scheduler on the process' behalf and this message is called `OUT_OF_QUANTUM`. Two system calls are exposed to user mode. *sys_schedctl* is called by a user mode scheduler to take over scheduling of a particular process. The kernel will make note of the scheduler's endpoint in its process table and send an out of quantum messages to that scheduler.

The scheduler will make its scheduling decision and reschedule the process using the *sys_schedule* system call. The scheduler may choose to place the process in a different priority queue or give it a different quantum, all depending on its policy. The *sys_schedule* call can also be used to modify the priority and quantum of a currently runnable process.

Another source can send a signal to the user mode scheduler, and this is the PM (process manager) which sends *mp_scheduler* which points to the scheduler's endpoint, or to *KERNEL* if the process is scheduled by the kernel's default policy. `ENDPOINTS` represent which user mode scheduler will handle the process if there are several user mode schedulers

2.3 The default user mode scheduler

When the scheduler receives an out of quantum message it needs to reschedule the process using `sys_schedule`. This is where the user mode scheduling policy kicks in. The policy will dictate in which priority queue the process should be scheduled and given what quantum. The current SCHED implementation mimics the policy that was previously implemented in kernel. Each time a process runs out of quantum, it will be bumped down in priority by one. Then, periodically, the scheduler will run through all processes that have been bumped down and push them up, one queue at a time. This way, a CPU bound process will quickly be pushed down to the lowest priority queue, but gradually make its way back up once it stops hawking the CPU.

2.4 Functions involved

To know which functions are involved in the scheduling process which will be of concern to us, we need to know the process lifecycle within the system. First, when a process is initialized or forked it goes to the PM hoping it could be scheduled. The PM then sends a message to the user mode scheduler to schedule the newly made or forked process. This message invokes our first function which is the `do_start_scheduling()` function present in `SCHEDULE.C` file which gives the process a few attributes which will be needed in making scheduling decisions such as level priority, time_slice, etc... then the function schedules the process for the FIRST TIME by giving it max priority and a certain time_slice. After the process finishes its quantum then the kernel itself sends the `out_of_quantum` message to the scheduler which will be received in the `do_no_quantum()` function present in the `SCHEDULER.C` file which will take the scheduling decisions from here on out by invoking `schedule_process_local()` which in turn calls another function `schedule_process()` which gives the process the priority and time_slice sent to it from `do_no_quantum()`. After its done, it sends the kernel the `SYS_SCHEDULE` system call to notify it that it has finished scheduling.

SO we now know that any change I need to be made to scheduling policy can be made either to `do_no_quantum` or the `schedule_process` functions either way it will give me the same result but we chose to alter the `do_no_quantum()`.

The back and forth between the kernel and the scheduler continues until the process finishes and terminates but the PM is only visited once when the process is first made so the `do_start_scheduling()` is only provoked when a process is newly made or newly forked. Another function which is my concern is the `balance_queues()` function present in the `SCHEDULE.C` file, this functions job is to limit starvation and it achieves this by looping after a set time from the lowest priority queues to upper ones and increase the level priority of processes present inside them to allow them to be executed in higher priority level queues so that the processes in lower level priority queues don't suffer from starvation. This process is called aging. Unfortunately, this function have to be commented out because in algorithms such as multi-level feedback, the processes must stay in their designated queues and not move up the level priority against what the multilevel feedback is looking to achieve.

do_start_scheduling():

```
/*=====*/
/*      do_start_scheduling      */
/*=====*/
int do_start_scheduling(message *m_ptr)
{
    register struct schedproc *rmp;
    int rv, proc_nr_n, parent_nr_n;

    /* we can handle two kinds of messages here */
    assert(m_ptr->m_type == SCHEDULING_START ||
           m_ptr->m_type == SCHEDULING_INHERIT);

    /* check who can send you requests */
    if (!accept_message(m_ptr))
        return EPERM;

    /* Resolve endpoint to proc slot. */
    if ((rv = sched_isemptyendpt(m_ptr->m_lsys_sched_scheduling_start.endpoint,
                                &proc_nr_n)) != OK)
    {
        return rv;
    }
    rmp = &schedproc[proc_nr_n];

    /* Populate process slot */
    rmp->endpoint = m_ptr->m_lsys_sched_scheduling_start.endpoint;
    rmp->parent = m_ptr->m_lsys_sched_scheduling_start.parent;
    rmp->max_priority = m_ptr->m_lsys_sched_scheduling_start.maxprio;
}
```

do_no_quantum():

```
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 OQ msg %u.\n",
               m_ptr->m_source);
        return EBADEPT;
    }

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


Schedule_process_local():

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

Schedule_process_local() calling schedule_process():

```
#define schedule_process_local(p) \
    schedule_process(p, SCHEDULE_CHANGE_PRIO | SCHEDULE_CHANGE_QUANTUM)
#define schedule_process_migrate(p) \
    schedule_process(p, SCHEDULE_CHANGE_CPU)
```

Schedule_process():

```
/*=====
 *          schedule_process          *
 *=====*/
static int schedule_process(struct schedproc *rmp, unsigned flags)
{
    int err;
    int new_prio, new_quantum, new_cpu, niced;

    pick_cpu(rmp);

    if (flags & SCHEDULE_CHANGE_PRIO)
        new_prio = rmp->priority;
    else
        new_prio = -1;

    if (flags & SCHEDULE_CHANGE_QUANTUM)
        new_quantum = rmp->time_slice;
    else
        new_quantum = -1;

    if (flags & SCHEDULE_CHANGE_CPU)
        new_cpu = rmp->cpu;
    else
        new_cpu = -1;

    niced = (rmp->max_priority > USER_Q);

    if ((err = sys_schedule(rmp->endpoint, new_prio,
                           new_quantum, new_cpu, niced)) != OK)
    {

```

2.5 CODE IMPLEMENTATION

- A Process *priority* denotes which queue it is in in the ready queues
- A few values have been defined by us in the **SCHEDULE.C** file which will be referenced in each algorithm we implement. These values can be found in :

```
minix_3.3.0 > minix > minix > servers > sched > C schedule.c
```

```
// student edits
// quantums
#define RRQ 200
#define PRIOQ 200

// value denoting each algorithm
#define MLFQ 1
#define RR 2
#define PRIO 3
#define SJF 4

int algorithm = MLFQ; // value denoting which algo should be used
// end student edits
```

ROUND ROBIN:

Changes have been made to *do_noquantum* function which can be found in

```
minix_3.3.0 > minix > minix > servers > sched > C schedule.c > do_noquantum(message *)
```

The idea was that when a process is sent to the function, we gave it a **RRQ** quantum which can be set to a certain value (200). we assigned that value to the variable *time_slice* which is a variable that denotes the quantum that each process must possess .



Code Screenshot:

```
case RR:
    // keep in same queue, so don't change priority
    // give constant time slice
    rmp->time_slice = RRQ;
    break;
```

As you can see, we didn't change priority as to keep the process in its current running queue while we changed the *time_slice* of the process by giving it **RRQ** which is set by us

MULTI-LEVEL FEEDBACK:

Changes have been made to *do_noquantum* function which can be found in

```
minix_3.3.0 > minix > minix > servers > sched >  schedule.c >  do_noquantum(message *)
```

The idea was that a process needs to change queue level each time it finishes its quantum so each time a process needs scheduling we check if it is in the last queue, if not, then we lower its priority level which means that it will be handled in a lower priority level queue so multilevel feedback is achieved.

Code Screenshot:

```
case MLFQ:
    // check current queue
    // if not last queue reduce queue by 1
    if (rmp->priority > NR_SCHED_QUEUES - 1)
    {
        rmp->priority = rmp->priority - 1;
    }
    else
    {
        rmp->priority=rmp->priority+1
    }
    break;
```

As you can see, we check the process's queue level if not in last queue then we lower its queue priority level .otherwise, we lower the priority value to increase priority so that it wont go to the idle queue which is a queue that executes when the system has no processes to execute on any level

PRIORITY SCHEDULING:

Here we encountered a problem and that is the priority associated with the process is the level queue priority but we need a priority variable that denotes the priority of the process itself within the queue

So we defined a variable called *inQueuePriority* in the *do_start_scheduling()* function which as mentioned before associate each process with few variables such as priority , time_slice and in this case *inQueuePriority* which would have a value between 0 and 9

```
minix_3.3.0 > minix > minix > servers > sched > C schedule.c >
```

```
/*=====
 *          do_start_scheduling          *
 *=====*/
int do_start_scheduling(message *m_ptr)
{
    register struct schedproc *rmp;
    int rv, proc_nr_n, parent_nr_n;

    /* we can handle two kinds of messages here */
    assert(m_ptr->m_type == SCHEDULING_START ||
           m_ptr->m_type == SCHEDULING_INHERIT);

    /* check who can send you requests */
    if (!accept_message(m_ptr))
        return EPERM;

    /* Resolve endpoint to proc slot. */
    if ((rv = sched_isemptyendpt(m_ptr->m_lsys_sched_scheduling_start.endpoint,
                                &proc_nr_n)) != OK)
    {
        return rv;
    }
    rmp = &schedproc[proc_nr_n];

    /* Populate process slot */
    rmp->endpoint = m_ptr->m_lsys_sched_scheduling_start.endpoint;
    rmp->parent = m_ptr->m_lsys_sched_scheduling_start.parent;
    rmp->max_priority = m_ptr->m_lsys_sched_scheduling_start.maxprio;
    // student edits
    rmp->inQueuePriority = rand() % 10;           // prio from 0 - 9
}
```

After that , the next change will be in the *do_no_quantum()* function in which the idea was that each time a process is sent to the scheduler , the scheduler would loop through all the processes present in the system and find the process with the highest priority and assign that process a certain quantum I assigned to priority Algorithm called **PRIOQ** with value 200 defined in **SCHEDULE.C** and send it back to kernel to execute this process.

Code Screenshot:

```
case PRIO:
    int minPrio = 10;
    schedproc minPrioProc;
    for (int i = 0; i < NR_PROCS; i++)
    {
        schedproc cp = schedproc[i];
        if (cp->flags & IN_USE && cp->inQueuePriority < minPrio)
        {
            minPrio = cp->inQueuePriority;
            minPrioProc = cp;
        }
    }
    minPrioProc->time_slice = PRIOQ;
    if ((rv = schedule_process_local(minPrioProc)) != OK)
    {
        return rv;
    }
    break;
```

We declare a **minPrio** variable with value 10 which is higher value than any process can have then we loop through all processes to find the process with that lowest priority value which means highest priority (**CP**) with the IF condition which check if the process is **INUSE** and its **inQueuePriority** lower than **minPrio**. We then assign this process to a **SCHEDPROC** variable called **minPrioProc** which will have the process I have chosen, and I will send that process back to the kernel to execute.

SHORTEST JOB FIRST (SJF):

Here we have encountered a different issue since each process must have a certain CPU burst time that is associated to it and since knowing how much time each process will spend executing is nearly impossible so one of the solutions that we came up with was to hardcode a random CPU burst time to each process and we shall choose the process with the least value to execute.

A variable name as **expectedExecTime** have been defined in **do_start_schedule()** to denote each process's expected CPU burst . this value would be between 2000 and 3000.

```

/*=====
 *      do_start_scheduling
 *=====*/
int do_start_scheduling(message *m_ptr)
{
    register struct schedproc *rmp;
    int rv, proc_nr_n, parent_nr_n;

    /* we can handle two kinds of messages here */
    assert(m_ptr->m_type == SCHEDULING_START ||
           m_ptr->m_type == SCHEDULING_INHERIT);

    /* check who can send you requests */
    if (!accept_message(m_ptr))
        return EPERM;

    /* Resolve endpoint to proc slot. */
    if ((rv = sched_isemptyendpt(m_ptr->m_lsys_sched_scheduling_start.endpoint,
                                &proc_nr_n)) != OK)
    {
        return rv;
    }
    rmp = &schedproc[proc_nr_n];

    /* Populate process slot */
    rmp->endpoint = m_ptr->m_lsys_sched_scheduling_start.endpoint;
    rmp->parent = m_ptr->m_lsys_sched_scheduling_start.parent;
    rmp->max_priority = m_ptr->m_lsys_sched_scheduling_start.maxprio;
    // student edits
    rmp->inQueuePriority = rand() % 10;           // prio from 0 - 9
    rmp->expectedExecTime = rand() % 1000 + 2001; // expected time from 2000 to 3000;
    // end students edits
}

```

The next change would be in the `do_no_quantum()` function in which the idea was the same as priority algorithm in which we loop through all processes in the system and find the process with the minimum time and send it back to the kernel to execute it

```

case SJF:
    int minExpectedTime = 4000;
    schedproc shortestJob;
    for (int i = 0; i < NR_PROCS; i++)
    {
        schedproc cp = schedproc[i];
        if (cp->flags & IN_USE && cp->expectedExecTime < minExpectedTime)
        {
            minExpectedTime = cp->expectedExecTime;
            shortestJob = cp;
        }
    }
    shortestJob->time_slice = 200;
    if ((rv = schedule_process_local(shortestJob)) != OK)
    {
        return rv;
    }
    break;
}

// Student edits done

```

Here we assign certain variable called `minExpectedTime` a certain value 4000 which is higher than any process can have then we loop through all processes in the system and find the shortest process and assign it to `SCHEDPROC` called `shortestJob` which is a `PROC` struct. By this we have chosen the shortest job and we send that process back to kernel to execute it

NB:

Having implemented all these algorithms , we have placed all these algorithms in a switch case statement in `do_no_quantum()` so that can be choice by the user to choose which algorithm to run by giving a value to a variable we defined in `SCHEDULE.C` called `ALGORITHM` and depending on which value its given , a certain algorithm will run

```
// student edits
// quantums
#define RRQ 200
#define PRIOQ 200

// value denoting each algorithm
#define MLFQ 1
#define RR 2
#define PRIO 3
#define SJF 4

int algorithm = MLFQ; // value denoting which algo should be used
// end student edits
```

```
// Student edits

switch (algorithm)
{
case MLFQ:
    // check current queue
    // if not last queue reduce queue by 1
    if (rmp->priority > NR_SCHED_QUEUES - 1)
    {
        rmp->priority = rmp->priority - 1;
    }
    else
    {
        rmp->priority=rmp->priority+1
    }
    break;
case RR:
    // keep in same queue, so don't change priority
    // give constant time slice
    rmp->time_slice = RRQ;
    break;
case PRIO:
    int minPrio = 10;
    schedproc minPrioProc;
    for (int i = 0; i < NR_PROCS; i++)
    {
        schedproc cp = schedproc[i];
        if (cp->flags & IN_USE && cp->inQueuePriority < minPrio)
        {
            minPrio = cp->inQueuePriority;
            minPrioProc = cp;
        }
    }
    minPrioProc->time_slice = PRIOQ;
    if ((rv = schedule_process_local(minPrioProc)) != OK)
    {
        return rv;
    }
    break;
case SJF:
    int minExpectedTime = 4000;
    // student edits
}
```


2.6 OUTPUT SCREENSHOTS OF PERFORMANCE

ANAYLSIS

RR:

Processes	Burst Time	Waiting Time	Turnaround Time
1	8	12	20
2	6	10	0
3	12	14	0
Average waiting time = 12.000000			
Average turnaround time = 6.666667			

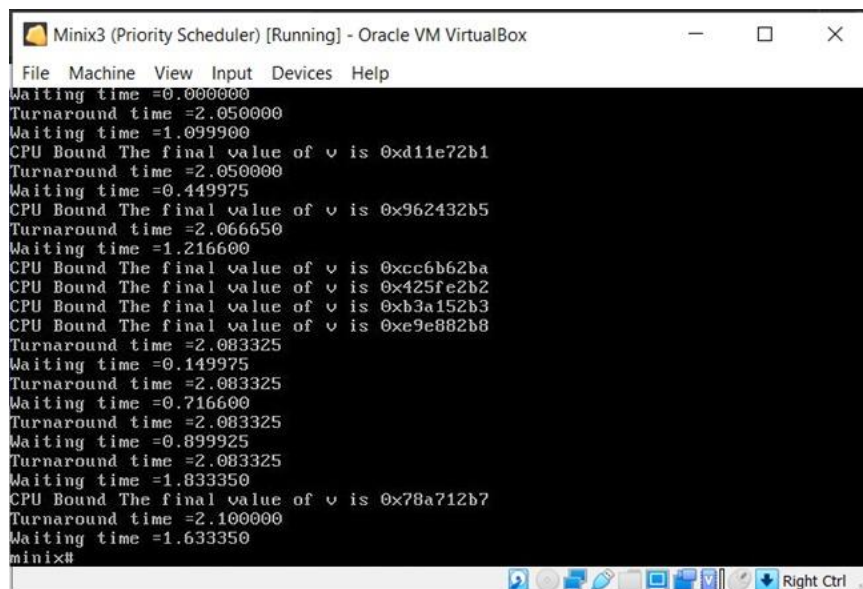
All values are approximates as the output was printed as a double with a stream of decimals.

SJF:

```
Enter number of process:3
Enter Burst Time:np1:9
p2:4
p3:3
sh: 1: pause: not found
Average Waiting Time=3.333333Average Turnaround Time=8.666667
```

All values are approximates as the output was printed as a double with a stream of decimals.

Priority Based:



```
Minix3 (Priority Scheduler) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Waiting time =0.000000
Turnaround time =2.050000
Waiting time =1.099900
CPU Bound The final value of v is 0xd11e72b1
Turnaround time =2.050000
Waiting time =0.449975
CPU Bound The final value of v is 0x962432b5
Turnaround time =2.066650
Waiting time =1.216600
CPU Bound The final value of v is 0xcc6b62ba
CPU Bound The final value of v is 0x425fe2b2
CPU Bound The final value of v is 0xb3a152b3
CPU Bound The final value of v is 0xe9e882b8
Turnaround time =2.083325
Waiting time =0.149975
Turnaround time =2.083325
Waiting time =0.716600
Turnaround time =2.083325
Waiting time =0.899925
Turnaround time =2.083325
Waiting time =1.833350
CPU Bound The final value of v is 0x78a712b7
Turnaround time =2.100000
Waiting time =1.633350
minix#
```

All values are approximates as the output was printed as a double with a stream of decimals.

Multilevel Feedback:

```
Process 13251 consumed Quantum 5 and Priority 16
Process 13251 consumed Quantum 5 and Priority 16
Process 13251 consumed Quantum 10 and Priority 17
Process 13251 consumed Quantum 20 and Priority 18
Process 13251 consumed Quantum 5 and Priority 16
Process 13251 consumed Quantum 5 and Priority 16
Process 13251 consumed Quantum 10 and Priority 17
Process 13251 consumed Quantum 20 and Priority 18
Process 13251 consumed Quantum 5 and Priority 16
Process 13251 consumed Quantum 5 and Priority 16
Process 13251 consumed Quantum 10 and Priority 17
Process 13251 consumed Quantum 20 and Priority 18
Process 13251 consumed Quantum 5 and Priority 16
Process 13251 consumed Quantum 5 and Priority 16
Process 13251 consumed Quantum 10 and Priority 17
```

Process kept going down priority and then going back up because of aging technique used by minix

3 REQUIRNMMENT 3:

“Memory Management”

3.1 *Memory Management*

The part supposed to manage the system is called memory manager. It is used to figure out which parts is used, and which are not in use, allocate and deallocate memory and swapping between memory and disk.

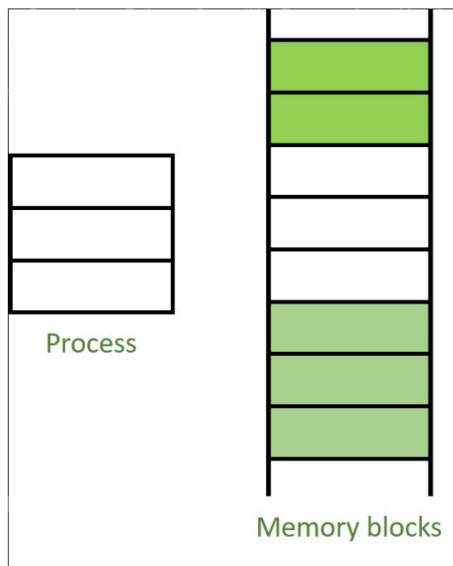
In MINIX 3 it is not in the kernel.

3.2 *Memory allocation*

Memory allocation happens when a process needs to run , so the memory manager allocates space in the memory for the process to run , this not so simple procedure is called memory allocation. There are several ways memory can be allocated to a certain process.

3.2.1 *Contiguous memory allocation*

Contiguous memory allocation is basically a method in which a single contiguous section/part of memory is allocated to a process or file needing it. Because of this all the available memory space resides at the same place together, which means that the freely/unused available memory partitions are not distributed in a random fashion here and there across the whole memory space.



3.2.2 *Multiple partition allocation*

This allocation is when the holes (free spaces in memory) are of different sizes and according to different algorithms which we will discuss in the next section memory will be allocated to files / process that needs memory space

3.2.2.1 *Different multiple partition allocating algorithms*

First fit: it is the algorithm which use the first hole that is big enough

Next fit: it is the algorithm which use the next hole that is big enough

Best fit: it is the algorithm which search list for smallest hole big enough

Worst fit: it is the algorithm which search list for largest hole available

Quick fit: it is the algorithm which separate lists of commonly requested sizes

3.2.3 *Paging*

Memory is divided up into sections or chunks called frames. A frame is fixed in size and is a unit of physical memory. Frames are the placeholders or storage spaces for pages, the parts or chunks of an executing process or task and represent a logical unit of memory. paging is a scheme of memory management which ends and eliminates the need of contiguous allocation of the physical memory, and this prevents the physical address from being contiguous as it permits the physical address of process to be non-contiguous since pages are in the end a set number of memory blocks that are allocated to the file or process. These blocks can be non-contiguous

Paging opens possibilities for options such as:

- Physical address space of a process can be noncontiguous.
- Part of process in the memory (executable part) and the other part in the backing store.

There are two types of addresses in paging:

- The logical address or the virtual address: it is the address which is generated by the CPU.
- The physical address: it is the address which is on the memory chip

The logical address space or the virtual address space: it is all of the logical addresses generated by a program

The physical address space: it is all the physical space which the logical space is corresponding to

The address is divided into two parts:

First part: page number

Indicates which page the memory need from the program.

Second part: Page offset

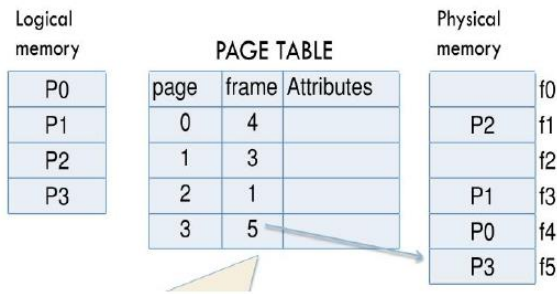
Indicates the location inside the page.

Logical address 2^m page size 2^n

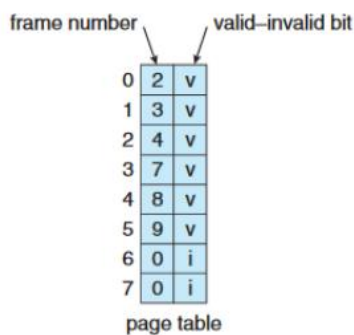
Then the page address needs $M-n$ bits, and page offset is needs n bits.

3.2.3.1 Page Table

Translates page number to frame number. Index is page number while inside the table is frame number. Bits of the frame number is not equal to bits of page number.



Page table base register points to page table. Page table length register indicates size of the table. Valid and invalid bit in page table, Indicates the validity of every page in the page frame.

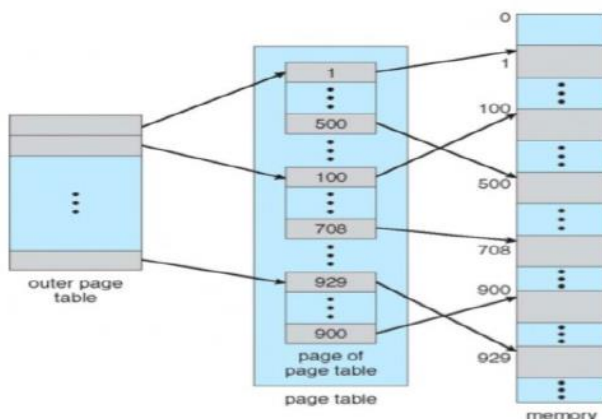


NOTE:

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

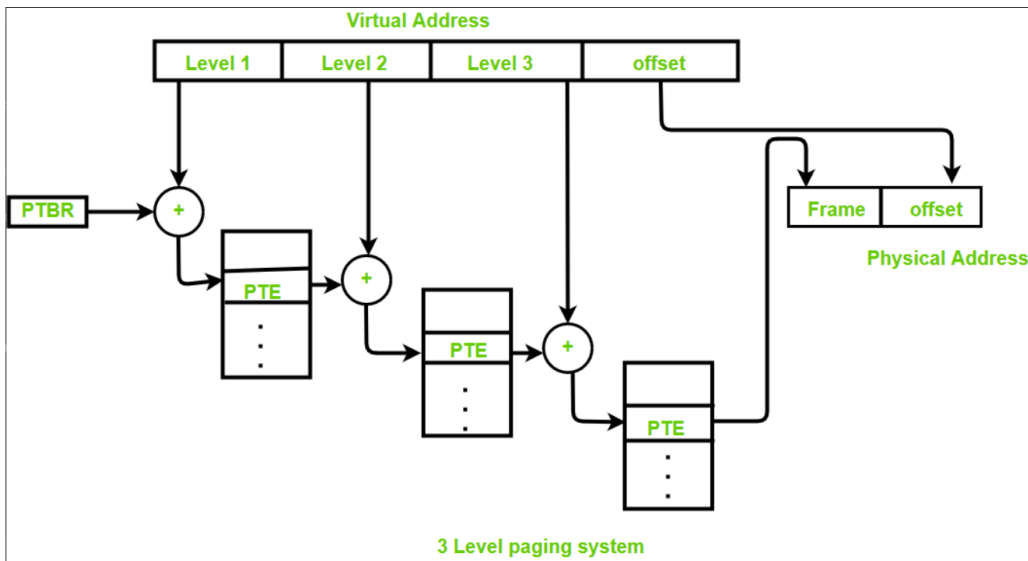
3.2.3.1.1 Page table Structure

Consider we have 32-bit for logical address and page size 4kb, so we need offset to indicate the location inside the page 2^{12} there is 20 remaining bits so we need page table of 2^{20} and it will be a huge paging table, So the usage of hierarchical page tables will solve this problem. The hierarchical page tables divide one level of paging into many levels.



3.2.3.2 Hierarchical paging

Multilevel Paging is a paging scheme which consist of two or more levels of page tables in a hierarchical manner. It is also known as hierarchical paging. The entries of the level 1 page table are pointers to a level 2 page table and entries of the level 2 page tables are pointers to a level 3 page table and so on. The entries of the last level page table are storing actual frame information. Level 1 contain single page table and address of that table is stored in PTBR (Page Table Base Register). In multilevel paging whatever may be levels of paging all the page tables will be stored in main memory. So it requires more than one memory access to get the physical address of page frame. One access for each level needed. Each page table entry except the last level page table entry contains base address of the next level page table.



Reference to actual page frame:

- Reference to PTE in level 1 page table = PTBR value + Level 1 offset present in virtual address.
- Reference to PTE in level 2 page table = Base address (present in Level 1 PTE) + Level 2 offset (present in VA).
- Reference to PTE in level 3 page table = Base address (present in Level 2 PTE) + Level 3 offset (present in VA).
- Actual page frame address = PTE (present in level 3).

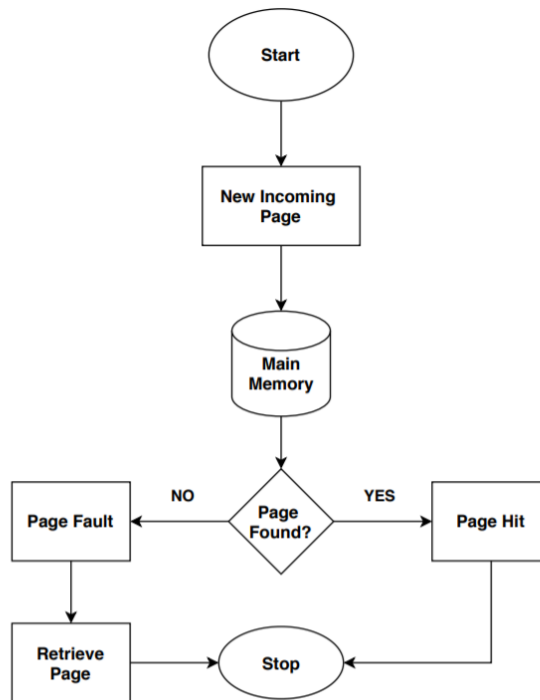

```
Number of entries in page table:
= (virtual address space size) / (page size)
= Number of pages

Virtual address space size:
=  $2^n B$ 

Size of page table:
<>= (number of entries in page table)*(size of PTE)
```

3.3 Page replacement

The page replacement algorithm decides which memory page is to be replaced. The process of replacement is sometimes called swap out or write to disk. Page replacement is done when the requested page is not found in the main memory (page fault). There are few algorithms that help us choose the victim frame (page which will be swapped out) but the two algorithms that we will focus on will be the FIFO (First-In-First-Out) algorithm and LRU (Least recently used) algorithm.



3.3.1 FIFO (First-In-First-Out)

In this algorithm, a queue is maintained. The page which is assigned the frame first will be replaced first. In other words, the page which resides at the rear end of the queue will be replaced on the every page fault.

3.3.1.1 *FIFO 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 (P[k] not in S) then

 S.add(P[k]) ;

 PF = PF + 1;

 QPage.put(P[k]) ;

 end

 else

 if (P[k] not in S) then

 val = QPage. Queue [0];

 QPage.get ();

 S.remove(val) ;

 S.add(P[k]) ;

 QPage.put(P[k]) ;

 PF = PF + 1 ;

 end

 end

end

return PF ;

explanation:

- Here S denotes the set of current pages in the system. We created a queue and named it as QPage to store the incoming pages in a FIFO manner. Initially, we set the number of page fault PF to zero.
- Now when a new page comes in, we check the capacity of set S. Here, we have three cases. The first one is when the set can store the new incoming page, and the page isn't already present in the set. In this case, we normally store the new page in the set S.
- The second scenario is when the set can store the new incoming page, but the page is already present in the set. In this case, we'll mark it as a page hit and won't increase the count of the variable PF.
- The third case occurs when the set is full. Here we need to perform FIFO to remove pages from the queue.
- Finally, when all the pages are either stored or removed from the set S, the algorithm returns the total number of page fault PF and terminates.

3.3.2 *LRU (Least recently used)*

This algorithm helps the Operating system to search those pages that are used over a short duration of time frame. The page that has not been used for the longest time in the main memory will be selected for replacement.

3.3.2.1 *LRU Steps example*

Let capacity be the number of pages that memory can hold. Let set be the current set of pages in memory.

1- Start traversing the pages.

i) If set holds less pages than capacity.

a) Insert page into the set one by one until the size of set reaches capacity or all page requests are processed.

b) Simultaneously maintain the recent occurred index of each page in a map called indexes.

c) Increment page fault

ii) Else If current page is present in set, do nothing.

Else

a) Find the page in the set that was least recently used. We find it using index array. We basically need to replace the page with minimum index.

b) Replace the found page with current page.

c) Increment page faults.

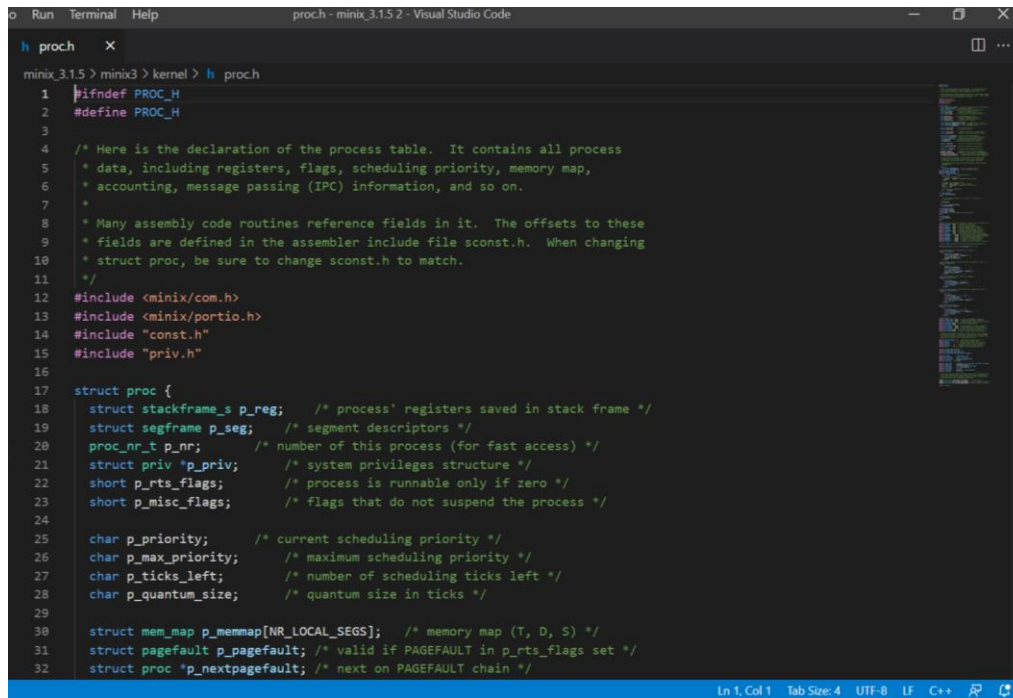
d) Update index of current page.

2. Return page faults.

3.4 Memory management in Minix

MINIX combine both the memory allocation and CPU allocation data in the same structure called `proc` placed in

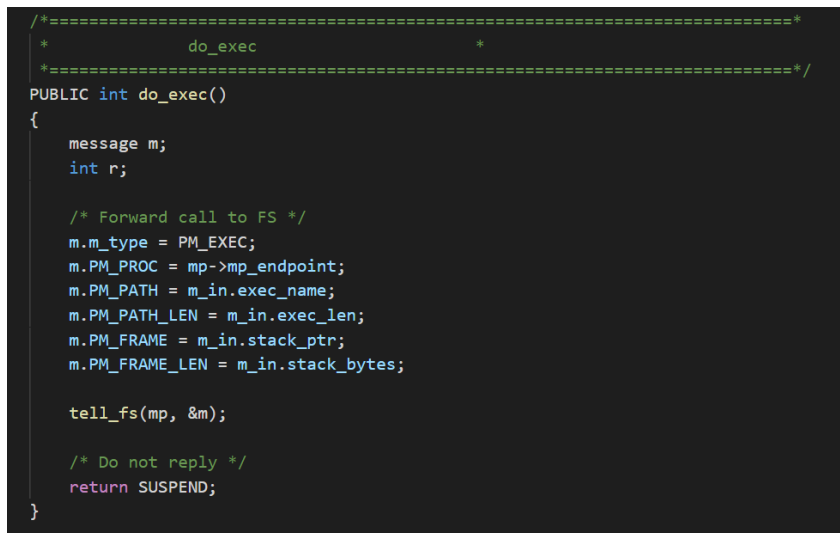
```
minix_3.1.5 > minix3 > kernel > h proc.h
```



```
h proc.h X
minix_3.1.5 > minix3 > kernel > h proc.h
1 #ifndef PROC_H
2 #define PROC_H
3
4 /* Here is the declaration of the process table. It contains all process
5  * data, including registers, flags, scheduling priority, memory map,
6  * accounting, message passing (IPC) information, and so on.
7  *
8  * Many assembly code routines reference fields in it. The offsets to these
9  * fields are defined in the assembler include file sconst.h. When changing
10  * struct proc, be sure to change sconst.h to match.
11  */
12 #include <minix/com.h>
13 #include <minix/portio.h>
14 #include "const.h"
15 #include "priv.h"
16
17 struct proc {
18     struct stackframe_s p_reg; /* process' registers saved in stack frame */
19     struct segframe p_seg; /* segment descriptors */
20     proc_nr_t p_nr; /* number of this process (for fast access) */
21     struct priv *p_priv; /* system privileges structure */
22     short p_rts_flags; /* process is runnable only if zero */
23     short p_misc_flags; /* flags that do not suspend the process */
24
25     char p_priority; /* current scheduling priority */
26     char p_max_priority; /* maximum scheduling priority */
27     char p_ticks_left; /* number of scheduling ticks left */
28     char p_quantum_size; /* quantum size in ticks */
29
30     struct mem_map p_memmap[NR_LOCAL_SEGS]; /* memory map (T, D, S) */
31     struct pagefault p_pagefault; /* valid if PAGEFAULT in p_rts_flags set */
32     struct proc *p_nextpagefault; /* next on PAGEFAULT chain */
33 }
```

while the actual memory allocation operations are placed in both files `forkexit.c` and `exec.c` which are both placed in

```
minix_3.1.5 > minix3 > servers > pm >
```



```
/*=====
 *
 * do_exec
 *
 *=====*/
PUBLIC int do_exec()
{
    message m;
    int r;

    /* Forward call to FS */
    m.m_type = PM_EXEC;
    m.PM_PROC = mp->mp_endpoint;
    m.PM_PATH = m_in.exec_name;
    m.PM_PATH_LEN = m_in.exec_len;
    m.PM_FRAME = m_in.stack_ptr;
    m.PM_FRAME_LEN = m_in.stack_bytes;

    tell_fs(mp, &m);

    /* Do not reply */
    return SUSPEND;
}
```

```

/*=====
 *      do_fork      *
 *=====*/
PUBLIC int do_fork()
{
    /* The process pointed to by 'mp' has forked. Create a child process. */
    register struct mproc *rmp; /* pointer to parent */
    register struct mproc *rmc; /* pointer to child */
    pid_t new_pid;
    static int next_child;
    int i, n = 0, r, s;
    endpoint_t child_ep;
    message m;

    /* If tables might fill up during FORK, don't even start since recovery half
     * way through is such a nuisance.
     */
    rmp = mp;
    if ((procs_in_use == NR_PROCS) ||
        (procs_in_use >= NR_PROCS-LAST_FEW && rmp->mp_effuid != 0))
    {
        printf("PM: warning, process table is full!\n");
        return(EAGAIN);
    }

    /* Find a slot in 'mproc' for the child process. A slot must exist. */
    do {
        next_child = (next_child+1) % NR_PROCS;
        n++;
    } while (next_child == next_child);
}

```

Those two files contains the code that handle 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 firstly 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.

3.4.1 VM server

As the virtual memory controls the memory (as it track all the memory the used and the unused one and assigns memory to processes and frees it) 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 places in them . Each entry is a memory block with a size of 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 refer to the physical block and the physical page of memory is described by this block also.

The system calls of the user space goes 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. mmap.c, cache.c
3. update of the data structures and the pagetable

3.5 Implementing Hierarchal Paging

Minix versions prior to 3.1.4 didn't have paging in their memory allocation implementation. However, this was changed from 3.1.4 and onwards and paging was introduced. The most recent versions of minix (3.3) not only have paging but it also has 3 level hierarchal paging, so we would have chosen to go back to version 3.1.5 which had single level paging which we will extend its page table to support two level page tables. 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.

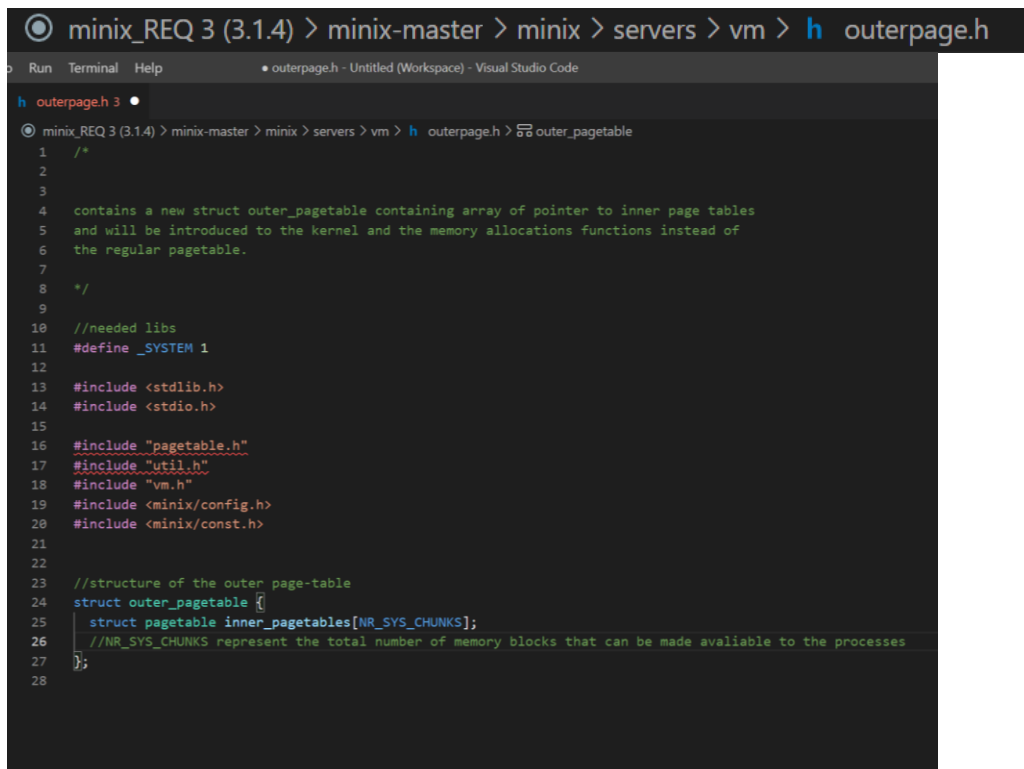
Our solution would include an implementation of an outer page table which will contain an array of pointers referring to a group of page-tables where the whole group will be called the inner page table. This solution brings the possibility of breaking a large page table into smaller ones by keeping track of the locations of those smaller page-tables in the actual memory.

Finally, all functions accessing allocation would be edited to support two level page tables by tracing down the memory allocation sequence.

Our solution would contain several steps and would be explained one by one in the coming sections.

3.5.1 STEP 1

Create a file name `outer_pagetable.h` which that will contain new struct of the `outer_pagetable`. This file will be in



```
minix_REQ 3 (3.1.4) > minix-master > minix > servers > vm > h outerpage.h
h outerpage.h 3
minix_REQ 3 (3.1.4) > minix-master > minix > servers > vm > h outerpage.h > o outer_pagetable
1  /*
2
3
4  contains a new struct outer_pagetable containing array of pointer to inner page tables
5  and will be introduced to the kernel and the memory allocations functions instead of
6  the regular pagetable.
7
8  */
9
10 //needed libs
11 #define _SYSTEM 1
12
13 #include <stdlib.h>
14 #include <stdio.h>
15
16 #include "pagetable.h"
17 #include "util.h"
18 #include "vm.h"
19 #include <minix/config.h>
20 #include <minix/const.h>
21
22
23 //structure of the outer page-table
24 struct outer_pagetable {
25     struct pagetable inner_pagetables[NR_SYS_CHUNKS];
26     //NR_SYS_CHUNKS represent the total number of memory blocks that can be made available to the processes
27 };
28
```

3.5.2 STEP 2

We need to keep track of starts and ends of the page tables in the memory, so we will add variables in the page tables file to keep track of starts and ends of the smaller page tables. These variables will be needed in `do_exec()` and `do_fork()` functions and will be given proper values. These values will be applied in

```
minix_REQ 3 (3.1.4) > minix-master > minix > servers > vm > C pagetable.c
```

```
101  /* May as well require them to be equal then.
102  */
103  ▾ #if CLICK_SIZE != VM_PAGE_SIZE
104      #error CLICK_SIZE must be page size.
105  #endif
106
107  vir_bytes inner_pagetable_start;
108  vir_bytes inner_pagetable_end;
109
110  static void *spare_pagequeue;
111  static char static_sparepages[VM_PAGE_SIZE*STATIC_SPAREPAGES]
112      __aligned(VM_PAGE_SIZE);
113
```

3.5.3 Step 3

We need to update any allocation functions to support our two-level paging.

In `alloc.c` we find page structure are defined so we need to alter this with to make them adapt with the new outer-level paging , moreover allocating and control functions must be updated because they are called in `do_exec()` and `do_fork()` . these will help us connect between the levels of our page tables.

Another thing we will add will be our new list of inner page tables to keep track of our inner pages etc.

Changes will be made to

```
minix_REQ 3 (3.1.4) > minix-master > minix > servers > vm > C alloc.c
```

A static structure must be added before any functions , this structure is a linked list holding inner page table addresses

```
//student edit
▾ 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 function that manage actual allocation and mapping */
} list_of_smaller_pagetables[MAXRESERVEDQUEUES]
//till here
```


3.5.4 Step 4

allocmem() is a function in minix that one of its jobs that is allocates the page table so we need to update it to place any page table that needs to be inserted in our list of inner page tables

changes are made also to alloc.c file in line 270

```
do {  
    mem = alloc_pages(clicks, memflags);  
    //student edit  
    page_table_update();  
    list_of_smaller_pagetables[number_of_pagetables++] = page_table_query(mem);  
    //till here  
} while(mem == NO_MEM && cache_freepages(clicks) > 0);
```

3.5.5 BONUS STEP

We can change our page size to any size if we want

This can be found in

```
minix_REQ 3 (3.1.4) > minix-master > minix > servers > vm > arch > earm > h pagetable.h > ...  
#define VM_PAGE_SIZE    ARM_PAGE_SIZE
```

Can be changed

```
#define VM_PAGE_SIZE    200
```

And hence physical pages in memory would be changed as we can see in

```
minix_REQ 3 (3.1.4) > minix-master > minix > servers > vm > c alloc.c > ...  
  
32  /* Number of physical pages in a 32-bit address space */  
33  #define NUMBER_PHYSICAL_PAGES (int)(0x100000000ULL/VM_PAGE_SIZE)
```

We can also change cache size per page and maxreserved pages number by changing their constants

```
57  #define MAXRESERVEDPAGES    300  
36  #define PAGE_CACHE_MAX    10000
```

3.6 IMPLEMENTING FIFO

Most memory management happens in PM since it contains the two main functions that initialize memory management (do_exec() and do_fork()), our solution included making a new file and naming it fifo.c and

place this file in PM server so that its functions can be called in during the execution of the `do_exec()` and `do_fork()` functions since these contain the memory allocation and have the possibilities where page replacement might be needed.

3.6.1 STEP 1

Changes made can be found in

```
minix_REQ 3 (3.1.4) > minix-master > minix > servers > pm > C fifo.c > ...  
  
1  /*  
2  STUDENT EDIT  
3  
4  FIFO algorithm is implemented in this file  
5  
6  */  
7  //needed libs  
8  #include <stdio.h>  
9  #include <stdlib.h>  
10 #include "pm.h"  
11 #include <sys/stat.h>  
12 #include <minix/callnr.h>  
13 #include <minix/endpoint.h>  
14 #include <minix/com.h>  
15 #include <minix/vm.h>  
16 #include <signal.h>  
17 #include <libexec.h>  
18 #include <sys/ptrace.h>  
19 #include "mproc.h"  
20 #include "vm.h"  
21
```

These are libs needed in minix and we need in our implementation

CODE:

```

21
22
23
24 int isInMemory(int pageRequest, int *pageTable, int tableSize)
25 {
26     int i;
27     for (i = 0; i < tableSize; i++)
28     {
29         if(pageRequest == pageTable[i])
30         {
31             return 1;
32         }
33     }
34     return 0;
35 }
36
37
38
39
40 void fifo_page_replace(smaller_inner_pagetable* pt)
41 {
42     //initialize all needed variables to check if a replacement is needed.
43     int tableSize = pt[0].number_of_pagetables;
44     int pageRequest, pageTableIndex = 0, numRequests = 0, numMisses = 0;
45     ssize_t bytesRead = &pt[0];
46     int i;
47     //int numHits = numRequests - numMisses;
48     //float hitRate = numHits/numRequests;
49
50     while(bytesRead) != -1)
51     {
52         pageRequest = list_of_smaller_pagetables[pageTableIndex].page_table_query;
53         if (pageRequest == 0)
54         {
55             continue;
56         }
57         numRequests++;
58         if (!isInMemory(pageRequest, pt, tableSize))
59         {
60             numMisses++;
61             if (pageTableIndex < tableSize)
62             { //still have room in page table
63                 pt[pageTableIndex++] = pageRequest;
64             }
65             else
66             {
67                 // Algorithm for FIFO
68                 for (i=0; i<tableSize;i++)
69                 {
70                     pt[i]=pageTable[i+1];
71                 }
72                 pt[tableSize-1]=pageRequest;
73             }
74             //update something in pageTable so that lru and second chance work correctly
75             else
76             {
77                 //printf("%d is already in table, no page faults.\n", pageRequest);
78             }
79         }
80         //printf("Hit rate = %f\n", (numRequests - numMisses)/(double)numRequests);
81         //printf("page faults = %f\n", numMisses);
82     }

```

CODE EXPLANATION:

- We used a queue which will contain our pages, this queue will place the newest page at the end of it and the oldest pages at the beginning of it in increasing order (ascending) .
- When checking for memory several conditions will be checked

- As you can guess from the name, the `isinmemory()` function iterates over the queue to search for the page requested with same number as the ones in the queue ,if page found then skip this iteration
- If page is not found and there is enough space , the requested page will be inserted in the queue at the end of the queue behind the last page as seen in line 61.
- Page shifting then happens in the for loop in line 68 when there is no space left and fifo algorithm needs to act so we shift pages and insert the page requested in the end while the page at the front (oldest) is shifted out to be stored in disk
- `numMisses` (page faults) is calculated along the code as well as the hit rate

3.6.2 STEP 2

Next we need to include our `fifo.c` file in the file that contains the `do_exec()` and `do_fork()` functions to be called to handle page replacement

```
minix_REQ 3 (3.1.4) > minix-master > minix > servers > pm > C exec.c > ...
30  #include "fifo.c"
```

```
minix_REQ 3 (3.1.4) > minix-master > minix > servers > pm > C forkexit.c > ...
31  #include "fifo.c"
```

After that we need to call `fifo_page_replace()` function inside `do_exec()` and `do_fork()` and give it appropriate parameter which is our list of inner page replacement

```

36  /*=====*/
37  *          do_exec          *
38  *=====*/
39  int
40  do_exec(void)
41  {
42      message m;
43
44      /* Forward call to VFS */
45      memset(&m, 0, sizeof(m));
46      m.m_type = VFS_PM_EXEC;
47      m.VFS_PM_ENDPT = mp->mp_endpoint;
48      m.VFS_PM_PATH = (void *)m_in.m_lc_pm_exec.name;
49      m.VFS_PM_PATH_LEN = m_in.m_lc_pm_exec.namelen;
50      m.VFS_PM_FRAME = (void *)m_in.m_lc_pm_exec.frame;
51      m.VFS_PM_FRAME_LEN = m_in.m_lc_pm_exec.framelen;
52      m.VFS_PM_PS_STR = m_in.m_lc_pm_exec.ps_str;
53      fifo_page_replace(&list_of_smaller_pagetables);
54      tell_vfs(mp, &m);
55
56      /* Do not reply */
57      return SUSPEND;
58  }

```

Do_fork()

```

131
132      tell_vfs(rmc, &m);
133      fifo_page_replace(&list_of_smaller_pagetables);

```

3.7 IMPLEMENTING LRU

Same as the fifo implementation a new file has been made named lru.c and placed in PM and then is called in the functions do_exec() and do_fork()

3.7.1 STEP 1

Changes made can be found in

```

minix_REQ 3 (3.1.4) > minix-master > minix > servers > pm > lru.c

```

```

1  /*
2  STUDENT EDIT
3
4  This file contains the LRU algorithm implementation
5
6
7  */
8
9  #include <stdio.h>
10 #include <stdlib.h>
11 #include "pm.h"
12 #include <sys/stat.h>
13 #include <minix/callnr.h>
14 #include <minix/endpoint.h>
15 #include <minix/com.h>
16 #include <minix/vm.h>
17 #include <signal.h>
18 #include <libexec.h>
19 #include <sys/ptrace.h>
20 #include "mproc.h"
21 #include "vm.h"
22
23 #include <stdio.h>
24 #include <stdlib.h>

```

These are the needed libs in minix and in our implementation

CODE:

```

24 #include <stdlib.h>
25
26 struct Node
27 {
28     int pageNum;
29     struct Node* next;
30     struct Node* prev;
31 };
32
33 struct Node* head;
34 struct Node* tail;
35
36 struct Node* GetNewNode(int pageRequest)
37 {
38     struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
39
40     newNode->pageNum = pageRequest;
41     newNode->next = NULL;
42     newNode->prev = NULL;
43     return newNode;
44 }
45
46 void InsertAtTail(int pageRequest)
47 {
48     struct Node* myNode = GetNewNode(pageRequest);
49     if (head == NULL)
50     {
51         head = myNode;
52         tail = myNode;
53         return;
54     }
55     tail->next = myNode;

```

```

56     myNode->next = NULL;
57     myNode->prev = tail;
58     tail = myNode;
59     return;
60 }
61
62 void PopAtHead(void)
63 {
64     struct Node* temp = head;
65     head = head->next;
66     head->prev = NULL;
67     temp->next = NULL;
68     free(temp);
69     return;
70 }
71
72 }
73
74 struct Node* isInMemory(int pageRequest, int tableSize)
75 {
76     {
77         struct Node* check = head;
78         while (check != NULL)
79         {
80             if (check->pageNum == pageRequest)
81             {
82                 return check;
83             }
84             check = check->next;
85         }
86         return NULL;
87     }
88 }
89
90 void lru_page_replace(smaller_inner_pagetable* pt)
91 {
92     //initialize all needed variables to check if a replacement is needed.
93     int tableSize = pt[0].number_of_pagetables;
94     int pageRequest, pageTableIndex = 0, numRequests = 0, numMisses = 0;
95     ssize_t bytesRead = &pt[0];
96     int i;
97
98     head = NULL;
99     tail = NULL;
100     //int numHits = numRequests - numMisses;
101     //float hitRate = numHits/numRequests;
102
103     while(bytesRead != -1)
104     {
105         pageRequest = list_of_smaller_pagetables[pageTableIndex].page_table_query;
106         if (pageRequest == 0)

```



```

114     pageRequest = list_of_smaller_pagetables[tableIndex].page_table_query;
115     if (pageRequest == 0)
116     {
117         continue;
118     }
119     numRequests++;
120     struct Node* nodeSelected = isInMemory(pageRequest, tableSize);
121     if (nodeSelected == NULL)
122     {
123         numMisses++;
124         //printf("before if pagetab<tableSize\n");
125         if (pageTableIndex < tableSize)
126         { //still have room in page table
127             InsertAtTail(pageRequest);
128             pageTableIndex++;
129         }
130         else
131         { // TODO implement a page replacement algorithm
132             // Algorithm for LRU
133             PopAtHead();
134             InsertAtTail(pageRequest);
135         }
136     }
137     //update something in pageTable so that lru and second chance work correctly
138     else
139     {
140         if (nodeSelected == head && nodeSelected != tail)
141         {
142             head = head->next;
143             head->prev = NULL;
144             nodeSelected->next = NULL;
145             nodeSelected->prev = tail;
149         else if (nodeSelected != head && nodeSelected != tail)
150         {
151             nodeSelected->prev->next = nodeSelected->next;
152             nodeSelected->next->prev = nodeSelected->prev;
153             nodeSelected->prev = tail;
154             nodeSelected->next = NULL;
155             tail->next = nodeSelected;
156             tail = nodeSelected;
157         }
158     }
159 }
160
161 //printf("page faults = %f\n", numMisses);
162 free(input);
163

```

CODE EXPLANATION:

In our implementation , LRU means getting the least recently used page as the victim page. These pages needed to be stored in a data structure , we used a double linked list to decrease number of shifting and to be easier to insert a new node(page) at the tail of the data structure and easier to pop the node at the head of the data structure. We implemented a function called GetNewNode() which creates a node with a page request that the function receives .

- The function lru_page_replace() gets called and it initializes every variable it needs it starts its job
- It checks for every inner page table to see if it contains the page requested
- how does this happen?
- It calls the isInmemory() function
- This isInmemory() function checks if the page is in a certain page table
- If page is found the functions returns check which is the page found
- If page not found then it returns null
- Back to lru_page_replacement() it takes the variable returned from isInmemory()
- If the returned value is null,this means that it is not found so the algorithm then checks if there is space available to place the page requested

- If there is no space so LRU must act and it pops the page at the head of the list (which is the least recently used page) and it inserts the requested page in the tail of the list (which represent the most recently used page)
- If the value returned from `isinmemory()` is check which means the page has been found
- It checks if the page found is in head or tail of the list or in the middle
- If It is in the head, then this means that this page is the most recently used pages so it places it in the tail of the list
- If it is in the tail then nothing happens since it is already most recently used
- Finally if it is not in the head or tail, it places this page at the tail of the list since it is the most recently used page

3.7.2 *STEP 2*

Next we need to call the function in the `do_exec()` and `do_fork()` instead of the fifo algorithms depending on which algorithm we want to use

```
53      fifo_page_replace(&list_of_smaller_pagetables); // OR lru_replace_page(&list_of_smaller_pagetables)
133     fifo_page_replace(&list_of_smaller_pagetables); // OR lru_replace_page(&list_of_smaller_pagetables)
```

3.8 OUTPUT SCREENSHOT OF PERFORMANCE ANALYSIS

```
File Machine View Input Devices Help

Page Replacement Algorithms
1.Enter data
2.FIFO
3.LRU
4.Exit
Enter your choice:1

Enter length of page reference sequence:8

Enter the page reference sequence:5
1
3
4
5
7
2
5
```

```
File Machine View Input Devices Help

Enter no of frames:3

Page Replacement Algorithms
1.Enter data
2.FIFO
3.LRU
4.Exit
Enter your choice:2

For 5 : 5
For 1 : 5 1
For 3 : 5 1 3
For 4 : 1 3 4
For 5 : 3 4 5
For 7 : 4 5 7
For 2 : 5 7 2
For 5 :No page fault
Total no of page faults:7
```

```
File Machine View Input Devices Help

Page Replacement Algorithms
1.Enter data
2.FIFO
3.LRU
4.Exit
Enter your choice:3

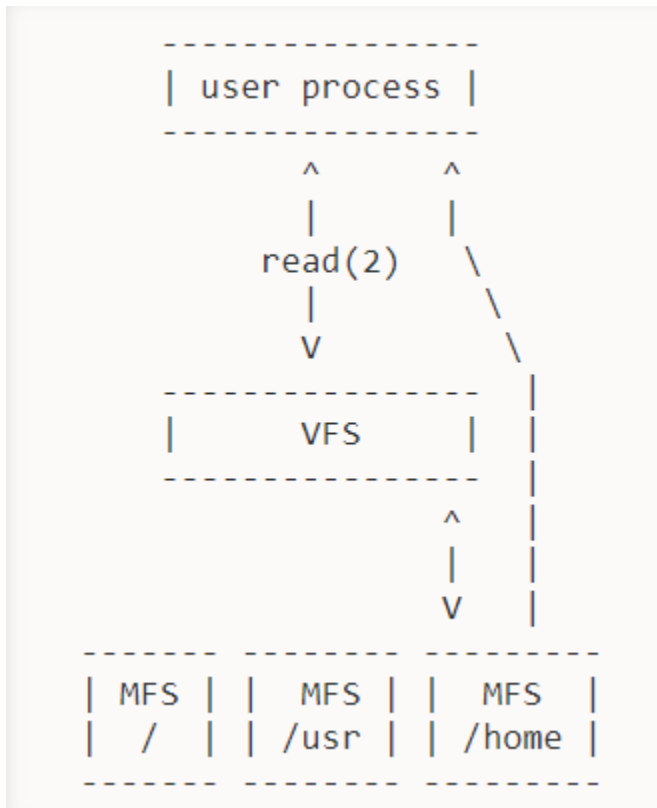
For 5 : 5
For 1 : 5 1
For 3 : 5 1 3
For 4 : 4 1 3
For 5 : 4 5 3
For 7 : 4 5 7
For 2 : 2 5 7
For 5 :No page fault!
Total no of page faults:7
```

4 REQUIREMENT 4:

“File Management System”

4.1 *Minix Implementation of file System*

The minix file system implementation doesn't contain several file system which communicate with user processes directly, instead there exists a layer (server) named as Virtual File System (VFS). This layer communicate usually between the user processes and the file systems themselves as shown in the figure below.

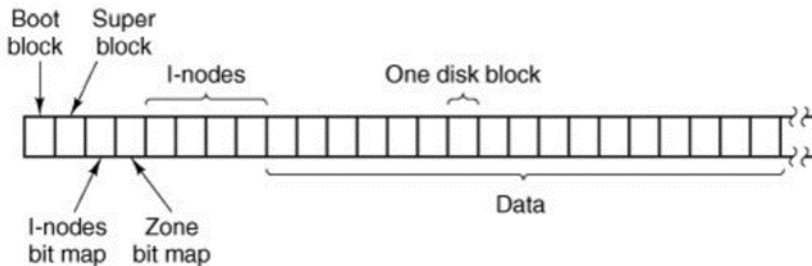


4.2 *VFS Internals*

VFS implements the file system in cooperation with one or more File Servers (FS). The File Servers take care of the actual file system on a partition. That is, they interpret the data structure on disk, write and read data to/from disk, etc. VFS sits on top of those File Servers and communicates with them. Looking inside VFS, we can identify several roles. First, a role of VFS is to handle most POSIX system calls that are supported by Minix. Additionally, it supports a few calls necessary for libc. VFS works roughly identical to every other server and driver in Minix; it fetches a message (internally referred to as a job in some cases), executes the request embedded in the message, returns a reply, and fetches the next job. There are several sources for new jobs: from user processes, from PM, from the kernel, and from suspended jobs inside VFS itself (suspended operations on pipes, locks, or character special files).

4.3 Minix Disk Partitions

Minix divides disc partitions into several sections:



A MINIX file system has six components:

The Boot Block which is always stored in the first block. It contains the boot loader that loads and runs an operating system at system startup.

The second block is the Superblock which stores data about the file system, that allows the operating system to locate and understand other file system structures. For example, the number of inodes and zones, the size of the two bitmaps and the starting block of the data area.

The inode bitmap is a simple map of the inodes that tracks which ones are in use and which ones are free by representing them as either a one (in use) or a zero (free).

The zone bitmap works in the same way as the inode bitmap, except it tracks the zones.

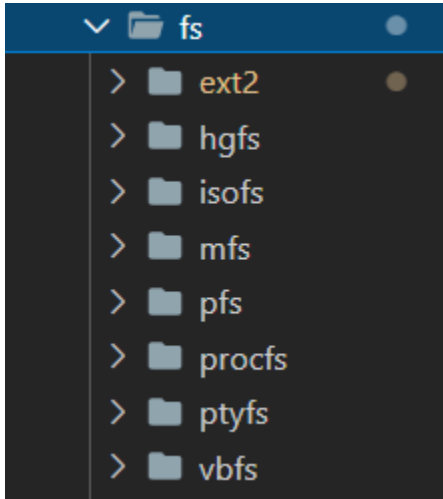
The inodes area. Each file or directory is represented as an inode, which records metadata including type (file, directory, block, char, pipe), IDs for user and group, three timestamps that record the date and time of last access, last modification, and last status change. An inode also contains a list of addresses that point to the zones in the data area where the file or directory data is actually stored.

The data area is the largest component of the file system, using the majority of the space. It is where the actual file and directory data are stored.

Each Partition Defines a certain file system

4.4 *File Systems*

The file system we will focus on in our implementation would be the **EXT2** file system but there are several file systems that are implemented in minix, and these are shown below:



Each file system is concerned with specific type of files,

4.4.1 *EXT2 File System*

The ext2 or second extended file system is a file system for the Linux kernel but it was also implemented in MINIX 3 as one of the file systems. The space in ext2 is split up into blocks. These blocks are grouped into block groups, analogous to cylinder groups in the Unix File System. There are typically thousands of blocks on a large file system. Data for any given file is typically contained within a single block group where possible. This is done to minimize the number of disk seeks when reading large amounts of contiguous data.

Each block group contains a copy of the superblock and block group descriptor table, and all block groups contain a block bitmap, an inode bitmap, an inode table, and finally the actual data blocks.

The superblock contains important information that is crucial to the booting of the operating system. Thus backup copies are made in multiple block groups in the file system. However, typically only the first copy of it, which is found at the first block of the file system, is used in the booting.

The group descriptor stores the location of the block bitmap, inode bitmap, and the start of the inode table for every block group. These, in turn, are stored in a group descriptor table.

4.4.2 *EXT2 allocating blocks overview*

When a new file or directory is created, ext2 must decide where to store the data. If the disk is mostly empty, then data can be stored almost anywhere. However, clustering the data with related data will minimize seek times and maximize performance.

ext2 attempts to allocate each new directory in the group containing its parent directory, on the theory that accesses to parent and children directories are likely to be closely related. ext2 also attempts to place files in the same group as their directory entries, because directory accesses often lead to file accesses. However, if the group is full, then the new file or new directory is placed in some other non-full group.

The data blocks needed to store directories and files can be found by looking in the data allocation bitmap. Any needed space in the inode table can be found by looking in the inode allocation bitmap which will be explained later in free space management.

4.5 *Functions Involved in block allocation*

To understand functions involved in allocating blocks to a requesting file, we have to understand the lifecycle of a file in EXT2.

Firstly, when a file needs to be created a function called `fs_open()` (with a certain argument that indicates that a file needs to be created) is called which in turn calls a function name `new_node()` found in

```
minix_3.3.0 > minix > minix > fs > ext2 >  open.c >  new_node(inode *, char *, mode_t, uid_t, gid_t, block_t)
```

Which allocates an Inode to this file and returning a pointer to that Inode as shown below and explained by comments.


```

/*=====
 *      new_node      *
 *=====*/
static struct inode *new_node(struct inode *ldirp,
    char *string, mode_t bits, uid_t uid, gid_t gid, block_t b0)
{
    /* New_node() is called by fs_open(), fs_mknod(), and fs_mkdir().
    * In all cases it allocates a new inode, makes a directory entry for it in
    * the ldirp directory with string name, and initializes it.
    * It returns a pointer to the inode if it can do this;
    * otherwise it returns NULL. It always sets 'err_code'
    * to an appropriate value (OK or an error code).
    */

    register struct inode *rip;
    register int r;

    if (ldirp->i_links_count == NO_LINK) { /* Dir does not actually exist */
        err_code = ENOENT;
        return(NULL);
    }

    if (S_ISDIR(bits) && (ldirp->i_links_count >= USHRT_MAX ||
        ldirp->i_links_count >= LINK_MAX)) {
        /* New entry is a directory, alas we can't give it a "." */
        err_code = EMLINK;
        return(NULL);
    }

    /* Get final component of the path. */
    rip = advance(ldirp, string);

    if (rip == NULL && err_code == ENOENT) {
        /* Last path component does not exist. Make new directory entry. */
        if ((rip = alloc_inode(ldirp, bits, uid, gid)) == NULL) {
            /* Can't creat new inode: out of inodes. */
            return(NULL);
        }

        /* Force inode to the disk before making directory entry to make
        * the system more robust in the face of a crash: an inode with
        * no directory entry is much better than the opposite.
        */
        rip->i_links_count++;
        rip->i_block[0] = b0; /* major/minor device numbers */
        rw_inode(rip, WRITING); /* force inode to disk now */

        /* New inode acquired. Try to make directory entry. */
        if ((r=search_dir(ldirp, string, &rip->i_num, ENTER,
            rip->i_mode & I_TYPE)) != OK) {
            rip->i_links_count--; /* pity, have to free disk inode */
            rip->i_dirt = IN_DIRTY; /* dirty inodes are written out */
            put_inode(rip); /* this call frees the inode */
            err_code = r;
            return(NULL);
        }

    } else {
        /* Either last component exists, or there is some problem. */
        if (rip != NULL)
            r = EEXIST;
        else
            r = err_code;
    }

    /* The caller has to return the directory inode (*ldirp). */
    err_code = r;
    return(rip);
}

```

Inside this implementation `alloc_inode()` function is called which tries to allocate an inode in the parents dev as explained in the previous section

```
/*=====
 *      alloc_inode
 *=====*/
struct inode *alloc_inode(struct inode *parent, mode_t bits, uid_t uid,
                          gid_t gid)
{
    /* Allocate a free inode on parent's dev, and return a pointer to it. */

    register struct inode *rip;
    register struct super_block *sp;
    int inumb;
    bit_t b;
    static int print_oos_msg = 1;

    sp = get_super(parent->i_dev); /* get pointer to super_block */
    if (sp->s_rd_only) { /* can't allocate an inode on a read only device. */
        err_code = EROFS;
        return(NULL);
    }

    /* Acquire an inode from the bit map. */
    b = alloc_inode_bit(sp, parent, (bits & I_TYPE) == I_DIRECTORY);
    if (b == NO_BIT) {
        err_code = ENOSPC;
        if (print_oos_msg)
            ext2_debug("Out of i-nodes on device %d/%d\n",
                major(sp->s_dev), minor(sp->s_dev));
        print_oos_msg = 0; /* Don't repeat message */
        return(NULL);
    }
}
```

After the file is allocated to an inode , each inode must be allocated some preallocated blocks which can be found inside inode struct in

```
minix_3.3.0 > minix > minix > fs > ext2 > h inode.h > ...
```

```
block_t i_prealloc_blocks[EXT2_PREALLOC_BLOCKS]; /* preallocated blocks */
int i_prealloc_count; /* number of preallocated blocks */
int i_prealloc_index; /* index into i_prealloc_blocks */
int i_preallocation; /* use preallocation for this inode, normally
```

This constant number of blocks is defined in

```
minix_3.3.0 > minix > minix > fs > ext2 > h const.h > ...
```

```
#define EXT2_PREALLOC_BLOCKS 4
```

This is an important number which we will need later in our extend-based allocation implementation

After that step our file is ready and allocated some blocks to start with. During its lifetime definitely the file would need more blocks to store more data this happens in the `Write.c` file when a file request a new block and this is where our small adjustment would be.

4.5.1 *Write.C file*

This file is important since it's the file where block allocation happen during the file lifecycle within the system. its operation include a function called `new_block ()` which is called when a file need a new block . This function when called tries to allocate a new block for the file and return a pointer to that new block to be stored in the file's inode section which stores all the blocks used by the file. This function can be found in

```

/*-----*/
struct buf *new_block(rip, position) register struct inode *rip; /* pointer to inode */
off_t position; /* file pointer */
{
    /* Acquire a new block and return a pointer to it. */
    struct buf *bp;
    int r;
    block_t b;

    /* Is another block available? */
    if ((b = read_map(rip, position, 0)) == NO_BLOCK)
    {
        /* Check if this position follows last allocated
        * block.
        */
        block_t goal = NO_BLOCK;
        if (rip->i_last_pos_bl_alloc != 0)
        {
            off_t position_diff = position - rip->i_last_pos_bl_alloc;
            if (rip->i_bsearch == 0)
            {
                /* Should never happen, but not critical */
                ext2_debug("warning, i_bsearch is 0, while\
                i_last_pos_bl_alloc is not!");
            }
            if (position_diff <= rip->i_sp->s_block_size)
            {
                goal = rip->i_bsearch + 1;
            }
            else
            {
                /* Non-sequential write operation,
                * disable preallocation
                * for this inode.
                */
                rip->i_preallocation = 0;
                discard_preallocated_blocks(rip);
            }
        }

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

    r = lufs_get_block_ino(&bp, rip->i_dev, b, NO_READ, rip->i_num,
    | | | | | rounddown(position, rip->i_sp->s_block_size));
    if (r != OK)
    {
        panic("ext2: error getting block (%llu,%u): %d", rip->i_dev, b, r);
        zero_block(bp);
        return (bp);
    }
}

```

In short this function checks for a goal block which the file wishes to acquire if found it places this block in the buffer variable created in the function by calling the

```
r = lmfs_get_block_ino(&bp, rip->i_dev, b, NO_READ, rip->i_num,
                      rounddown(position, rip->i_sp->s_block_size));
```

Then it returns that buffer containing the free block.

```
return (bp);
```

4.6 Extent-based allocation code implementation

Simply, having an Extent-based allocation algorithm allow the system to allocate certain number of blocks to the file each time a file request extra space these number of blocks that are allocated each time are called EXTENTS.

We would implement this by simply seeing when does a file request a block and place this request in a for loop with a stopping condition which would be our EXTENT size. For simplicity we would make this stopping condition as the constant defined as **EXT2_PREALLOC_BLOCKS** which was given to the inode which as first created and is found in const.c as shown in section 4.5 we can even simply exchange this constant as we want

1st modification:

```
// #define EXT2_PREALLOC_BLOCKS 8
#define EXT2_PREALLOC_BLOCKS 4
```

2nd modification:

```
struct buf *bp[EXT2_PREALLOC_BLOCKS];
```

Make the buffer that was holding the block to be allocated as an array of pointers which will be pointing to each block that will be allocated

3rd modification:

```
// student edit for loop
for (int i = 0; i < EXT2_PREALLOC_BLOCKS; i++)
{
    if ((b = alloc_block(rip, goal)) == NO_BLOCK)
    {
        err_code = ENOSPC;
        return (NULL);
    }
    if ((r = write_map(rip, position, b, 0)) != OK)
    {
        free_block(rip->i_sp, b);
        err_code = r;
        ext2_debug("write_map failed\n");
        return (NULL);
    }
    rip->i_last_pos_bl_alloc = position;
    if (position == 0)
    {
        /* rip->i_last_pos_bl_alloc points to the block position,
        * and zero indicates first usage, thus just increment.
        */
        rip->i_last_pos_bl_alloc++;
    }
}

r = lmfs_get_block_ino(&bp[i], rip->i_dev, b, NO_READ, rip->i_num,
                      roundup(position, rip->i_sp->s_block_size));
}
// till here
```

For loop to find a **EXT2_PREALLOC_BLOCKS** number of blocks that will be allocated to file and place them into the buffer array then return buffer normally and change needed modification in receiving files

```
return (bp);
```

4.7 Free disk space management

BITMAPS!!

Minix tracks free inodes and zones using bitmaps. When a file is deleted, its corresponding inodes is deleted by marking the corresponding bits to 0, Disk storage is allocated in terms of 2n blocks called zones. A zone is a method to allows allocating as much blocks next to each other (on the same cylinder) to save load time. To create a new file the disk is searched for the first free inode then it starts allocating it. The first free block is already cached in the super block. Also the superblock keeps count of free blocks and inodes count needed for faster allocation and free space management

These are defined in

```
minix_3.3.0 > minix > minix > fs > ext2 > h super.h > ...
struct group_desc
{
    u32_t    block_bitmap;        /* Blocks bitmap block */
    u32_t    inode_bitmap;        /* Inodes bitmap block */
    u32_t    inode_table;         /* Inodes table block */
    u16_t    free_blocks_count;    /* Free blocks count */
    u16_t    free_inodes_count;    /* Free inodes count */
    u16_t    used_dirs_count;      /* Directories count */
    u16_t    pad;
    u32_t    reserved[3];
};
```

Obviously free space management is needed in allocation , deallocation of every block and inode as they are seen throughout all files of the file systems

Few examples are :

- When allocating an inode:

```
/*=====
 *          alloc_inode
 *=====*/
struct inode *alloc_inode(struct inode *parent, mode_t bits, uid_t uid,
                          gid_t gid)
{
    /* Allocate a free inode on parent's dev, and return a pointer to it. */

    register struct inode *rip;
    register struct super_block *sp;
    int inumb;
    bit_t b;
    static int print_oos_msg = 1;

    sp = get_super(parent->i_dev);    /* get pointer to super_block */
    if (sp->s_rd_only) {    /* can't allocate an inode on a read only device. */
        err_code = EROFS;
        return(NULL);
    }

    /* Acquire an inode from the bit map. */
    b = alloc_inode_bit(sp, parent, (bits & I_TYPE) == I_DIRECTORY);
    if (b == NO_BIT) {
        err_code = ENOSPC;
        if (print_oos_msg)
            ext2_debug("Out of i-nodes on device %d/%d\n",
                major(sp->s_dev), minor(sp->s_dev));
        print_oos_msg = 0;    /* Don't repeat message */
        return(NULL);
    }
}
```

Alloc_inode_bit() in turn uses this to check inode bitmap:

```
bp = get_block(sp->s_dev, gd->inode_bitmap, NORMAL);
```

- Similarly when Allocating blocks:

```

/*===== alloc_block_bit =====*/
static block_t alloc_block_bit(sp, goal, rip)
struct super_block *sp; /* the filesystem to allocate from */
block_t goal; /* try to allocate near this block */
struct inode *rip; /* used for preallocation */
{
    block_t block = NO_BLOCK; /* allocated block */
    int word; /* word in block bitmap */
    bit_t bit = -1;
    int group;
    char update_bsearch = FALSE;
    int i;

    if (goal >= sp->s_blocks_count ||
        (goal < sp->s_first_data_block && goal != 0)) {
        goal = sp->s_bsearch;
    }

    if (goal <= sp->s_bsearch) {
        /* No reason to search in a place with no free blocks */
        goal = sp->s_bsearch;
        update_bsearch = TRUE;
    }

    /* Figure out where to start the bit search. */
    word = ((goal - sp->s_first_data_block) % sp->s_blocks_per_group)
        / FS_BITCHUNK_BITS;

    /* Try to allocate block at any group starting from the goal's group.
     * First time goal's group is checked from the underneath after all
    */

    bp = get_block(sp->s_dev, gd->block_bitmap, NORMAL);
}

```

Moreover , free space management is made more efficient by means of counting as it stores the first free block to speed up search results and counts of free inodes and blocks as mentioned above

Found in

```

minix_3.3.0 > minix > minix > fs > ext2 > h inode.h
block_t i_bsearch; /* where to start search for new blocks,
                  * also this is last allocated block.

```

This is used so next time search for a new block is needed the search begins with this block's bitmap position to speed up the search process

Also this found in

```

minix_3.3.0 > minix > minix > fs > ext2 > h super.h >
block_t s_bsearch; /* all data blocks below this block are in use*/
int s_igsearch; /* all groups below this one have no free inodes */

```

Present in superblock to speed up search as well in processes.

5 **REFERENCES:**

- [1] Silberschatz, A., Galvin, P. and Gagne, G., 2012. Operating System Concepts. 9th ed. United States of America: John Wiley&Sons, p.944.
- [2] imma. 2021. Presentation: Internal Structure of Minix. [online] Available at: <<https://imma.wordpress.com/2007/04/02/presentation-internal-structure-of-minix/>> [Accessed 3 December 2021].
- [3] En.wikipedia.org. 2021. Minix 3 - Wikipedia. [online] Available at: <https://en.wikipedia.org/wiki/Minix_3> [Accessed 3 December 2021].