



CSE 335 Operating Systems

MINIX 3

OS Course Project

PROJECT DOCUMENTATION

Submitted to:

Prof. Dr. Gamal Abdel Shafy

Eng. Sally Shaker

Submitted by:

Youssef George Fouad 19P9824

Mostafa Nasrat Metwally 19P4619

Kerollos Wageeh Youssef 19P3468

Anthony Amgad Fayek 19P9880

Table of Contents

| | |
|--|----|
| <i>Table of Contents</i> | 2 |
| <i>List of figures</i> | 4 |
| 0. INTRODUCTION | 5 |
| 1. MINIX OS | 5 |
| 1.1. Internal Structure of MINIX 3 | 5 |
| 1.2. Kernel Layer..... | 6 |
| 1.3. Devices Drivers Layer | 6 |
| 1.4. Server Processes Layer | 6 |
| 1.5. User Processes Layer | 6 |
| 2. CPU SCHEDULING | 7 |
| 2.1. CPU Scheduling in MINIX 3 | 7 |
| 2.2. Round Robin..... | 8 |
| 2.2.1. Advantages..... | 8 |
| 2.2.2. Disadvantages | 8 |
| 2.2.3. RR in MINIX 3..... | 8 |
| 2.3. Shortest Job First (SJF)..... | 9 |
| 2.3.1. Advantages..... | 9 |
| 2.3.2. Disadvantages | 9 |
| 2.3.3. Implementing SJF in MINIX 3 | 10 |
| 2.4. Priority-Based Scheduling..... | 19 |
| 2.4.1. Advantages..... | 19 |
| 2.4.2. Disadvantages | 19 |
| 2.4.3. Implementing priority scheduling in MINIX 3 | 20 |
| 2.5. Multi-Level Feedback Queue..... | 21 |
| 2.5.1. Implementing MFQ in MINIX 3 | 21 |

| | | |
|--------|---|----|
| 2.6. | Testing Scheduling Algorithms | 25 |
| 2.6.1. | Testing RR..... | 26 |
| 2.6.2. | Testing SJF | 27 |
| 2.6.3. | Testing priority-based queue..... | 28 |
| 2.6.4. | Testing MFQ | 29 |
| 2.6.5. | Comparing & evaluating testing results..... | 30 |
| 3. | MEMORY MANAGEMENT | 31 |
| 3.1. | Memory management in xv6 | 31 |
| 3.2. | Hierachal paging in xv6 (2-level paging)..... | 33 |
| 3.3. | How we modified hierachal paging (3-level paging) | 35 |
| 3.4. | Implementing page replacement algorithms in xv6 | 38 |
| 3.4.1. | First In First Out (FIFO) algorithm | 44 |
| 3.4.2. | Last Recently Used (LRU) algorithm..... | 45 |
| 3.5. | Performance results for hierachal paging and replacement algorithms | 46 |
| 4. | FILE SYSTEM MANAGEMENT..... | 47 |
| 4.1. | Requirement Statement..... | 47 |
| 4.2. | File System in MINIX 3 | 47 |
| 4.3. | How MINIX 3 manages empty space | 51 |
| 4.4. | How we edited the free space management..... | 51 |
| 4.4.1. | Extents..... | 51 |
| 4.4.2. | Code Edits..... | 52 |
| 4.4.3. | Performance testing | 53 |

List of figures

| | |
|--|-------------------------------------|
| FIGURE 1: MINIX 3 STRUCTURE | 5 |
| FIGURE 2: PRIORITY SCHEDULING CODE | 20 |
| FIGURE 3: MFQ QUEUES 16-18 DEFINITION | 21 |
| FIGURE 4: MFQ QUANTUM | 22 |
| FIGURE 5: MFQ CODE 3 | 22 |
| FIGURE 6: MFQ CODE 4 | 23 |
| FIGURE 7: MFQ CODE 5 | 23 |
| FIGURE 8: MFQ DO_NICE() | 24 |
| FIGURE 9: MFQ BALANCE_QUEUES() | ERROR! BOOKMARK NOT DEFINED. |
| FIGURE 10: MFQ EXECUTION EXAMPLE | 25 |
| FIGURE 11: TESTING DEFAULT RR | 26 |
| FIGURE 12: TESTING SJF | 27 |
| FIGURE 13: TESTING PRIORITY BASED | 28 |
| FIGURE 14: TESTING MFQ | 29 |
| FIGURE 15: LOGICAL ADDRESS DIVISION | 32 |
| FIGURE 16: SENDING ADDRESSES TO MMU | 32 |
| FIGURE 17: MULTI-LEVEL PAGING | 33 |
| FIGURE 18: HIERARCHICAL PAGING XV6 | 34 |
| FIGURE 19: 3-LEVEL HIERARCHICAL PAGING | 35 |
| FIGURE 20: 3-LEVEL PAGING CODE 1 | 36 |
| FIGURE 21: 3-LEVEL PAGING CODE 2 | 37 |
| FIGURE 22: 3-LEVEL PAGING CODE 3 | 37 |
| FIGURE 23: PAGE SWAPPING CODE 1 | 38 |
| FIGURE 24: PAGE SWAPPING CODE 2 | 39 |
| FIGURE 25: PAGE SWAPPING CODE 3 | ERROR! BOOKMARK NOT DEFINED. |
| FIGURE 26: PAGE SWAPPING CODE 4 | 41 |
| FIGURE 27: PAGE SWAPPING CODE 5 | 42 |
| FIGURE 28: PAGE SWAPPING CODE 6 | 42 |
| FIGURE 29: PAGE SWAPPING CODE 7 | 43 |
| FIGURE 30: PAGE SWAPPING CODE 8 | 43 |
| FIGURE 31: FIFO REPLACEMENT ALGORITHM CODE | 44 |
| FIGURE 32: LRU ALGORITHM CODE | 45 |
| FIGURE 33: MINIX SUPERBLOCK | 48 |
| FIGURE 34: MINIX I-NODE | 50 |
| FIGURE 35: FS CODE 1 | 52 |
| FIGURE 36: FS BUILDING SUCCESSFULLY | 53 |

0. INTRODUCTION

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

1. MINIX OS

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

1.1. *Internal Structure of MINIX 3*

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

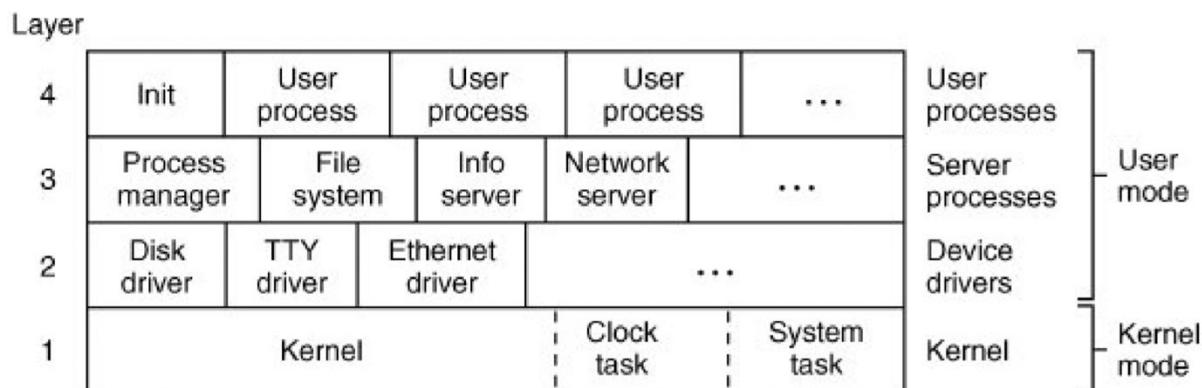


Figure 1: MINIX 3 Structure

1.2. *Kernel Layer*

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

1.3. *Devices Drivers Layer*

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

1.4. *Server Processes Layer*

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

1.5. *User Processes Layer*

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

2. CPU SCHEDULING

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

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

2.1. CPU Scheduling in MINIX 3

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

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

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

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

2.2. Round Robin

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

2.2.1. Advantages

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

2.2.2. Disadvantages

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

2.2.3. RR in MINIX 3

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

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

2.3. Shortest Job First (SJF)

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

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

Where:

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

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

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

2.3.1. Advantages

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

2.3.2. Disadvantages

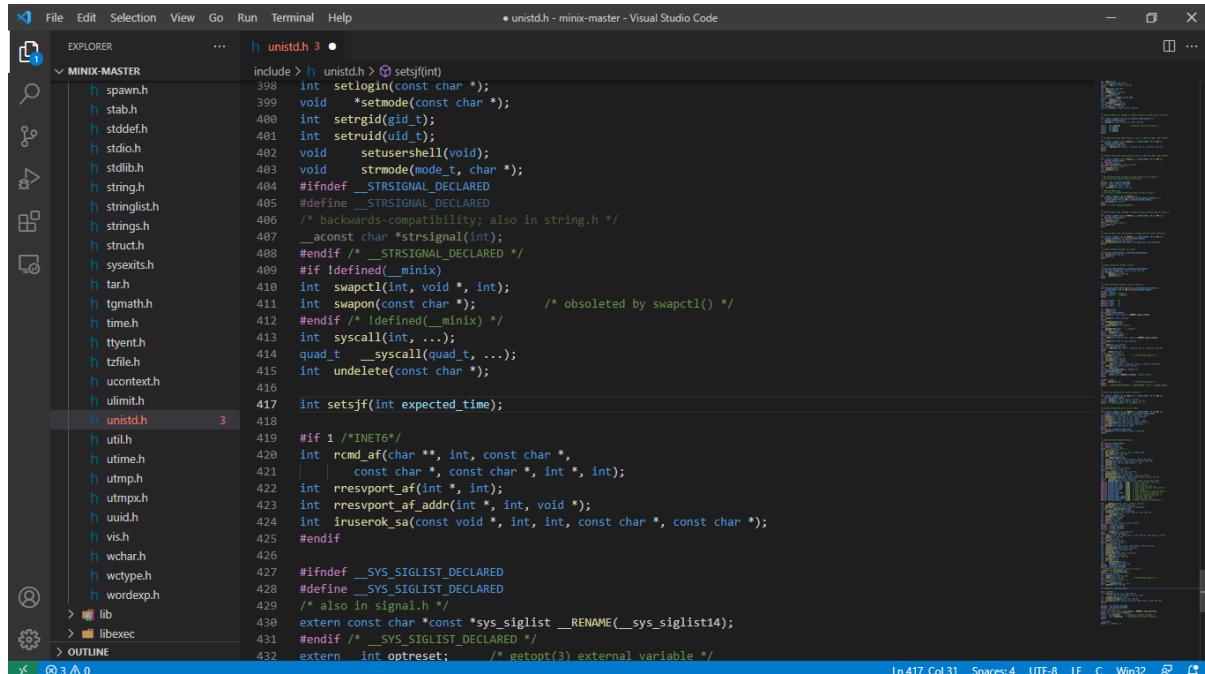
- Might cause starvation of long processes in case shorter processes keeps entering the queue.
- Difficult to predict the next CPU burst length.

2.3.3. Implementing SJF in MINIX 3

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

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

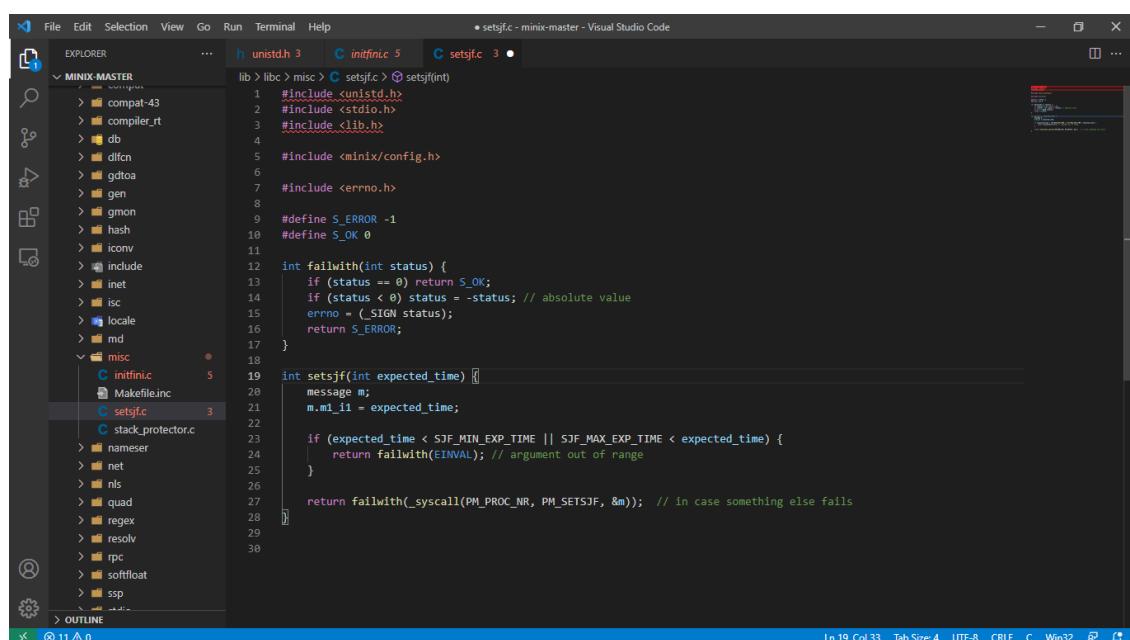
Line 417: We put the prototype of setsjf function which validates the expected time.



```
File Edit Selection View Go Run Terminal Help
unistd.h - minix-master - Visual Studio Code
EXPLORER MINIX-MASTER
unistd.h 3
include > h unistd.h > setsjf(int)
398 int setlogin(const char *); 
399 void *setmode(const char *); 
400 int setregid(gid_t); 
401 int setreuid(uid_t); 
402 void setusershell(void); 
403 void strmode(mode_t, char *); 
404 #ifndef __STRSIGNALL_DECLARED 
405 #define __STRSIGNALL_DECLARED 
406 /* backwards-compatibility; also in string.h */ 
407 __acronst char *strsignal(int); 
408 #endif /* __STRSIGNALL_DECLARED */ 
409 #if !defined(__minix) 
410 int swapct1(int, void *, int); 
411 int swapon(const char *); /* obsoleted by swapct1() */ 
412 #endif /* !defined(__minix) */ 
413 int syscall(int, ...); 
414 quad_t __syscall(quad_t, ...); 
415 int undelete(const char *); 
416 
417 int setsjf(int expected_time); 
418 
419 #if 1 /*INET6*/ 
420 int rcmd_af(char **, int, const char *, 
421             const char *, const char *, int *, int); 
422 int rresvport_af(int *, int); 
423 int rresvport_af_addr(int *, int, void *); 
424 int iruserok_sa(const void *, int, int, const char *, const char *); 
425 #endif 
426 
427 #ifndef __SYS_SIGLIST_DECLARED 
428 #define __SYS_SIGLIST_DECLARED 
429 /* also in signal.h */ 
430 extern const char *const *sys_siglist __RENAME(__sys_siglist14); 
431 #endif /* __SYS_SIGLIST_DECLARED */ 
432 extern int optreset; /* getopt(3) external variable */ 
Ln 417, Col 31 Spaces:4 UTF-8 LF C Win32
```

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

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



```
File Edit Selection View Go Run Terminal Help
setsjf.c - minix-master - Visual Studio Code
EXPLORER MINIX-MASTER
unistd.h 3
lib > libc > misc > setsjf.c > setsjf()
1 #include <unistd.h>
2 #include <stdio.h>
3 #include <lib.h>
4 
5 #include <minix/config.h>
6 
7 #include <errno.h>
8 
9 #define S_ERROR -1
10 #define S_OK 0
11 
12 int failwith(int status) {
13     if (status == 0) return S_OK;
14     if (status < 0) status = -status; // absolute value
15     errno = (_SIGN status);
16     return S_ERROR;
17 }
18 
19 int setsjf(int expected_time) {
20     message m;
21     m.m1_id = expected_time;
22 
23     if (expected_time < SCHED_MIN_EXP_TIME || SCHED_MAX_EXP_TIME < expected_time) {
24         return failwith(EINVAL); // argument out of range
25     }
26 
27     return failwith(_syscall(PM_PROC_NR, PM_SETSJF, &m)); // in case something else fails
28 }
```

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

Line 12: Add setsjf.c in the Makefile.

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** On the left, it displays the project structure under "MINIX-MASTER". The "misc" folder is expanded, showing files like "initfini.c", "Makefile.inc", "setsjf.c", and "stack_protector.c".
- Code Editor:** The main area shows the content of "setsjf.c". The code includes comments about NetBSD 1.2 and the stack protector, along with declarations for "initfini.c", "stack_protector.c", and "setsjf.c".
- Status Bar:** At the bottom, it shows "L12, Col 17" and "Spaces: 4 - UTF-8 - LF".

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

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

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

```
File Edit Selection View Go Run Terminal Help callnr.h - minix-sjf-scheduler - Visual Studio Code

EXPLORER h config.h new\...\_2 h config.h old\...\_2 h callnr.h 1 x
new > usr > src > minix > include > minix > h callnr.h > NR_PM_CALLS
47 #define PM_CLOCK_GETTIME (PM_BASE + 34)
48 #define PM_CLOCK_SETTIME (PM_BASE + 35)
49 #define PM_GETRUSAGE (PM_BASE + 36)
50 #define PM_REBOOT (PM_BASE + 37)
51 #define PM_SVRCRTL (PM_BASE + 38)
52 #define PM_SPROF (PM_BASE + 39)
53 #define PM_CPROF (PM_BASE + 40)
54 #define PM_SRV_FORK (PM_BASE + 41)
55 #define PM_SRV_KILL (PM_BASE + 42)
56 #define PM_EXEC_NEW (PM_BASE + 43)
57 #define PM_EXEC_RESTART (PM_BASE + 44)
58 #define PM_GETEPINFO (PM_BASE + 45)
59 #define PM_GETPROCNR (PM_BASE + 46)
60 #define PM_GETSYSINFO (PM_BASE + 47)
61 #define PM_SETSJF (PM_BASE + 48)
62
63 #define NR_PM_CALLS 49 /* highest number from base plus one */
64
65 /*-----*
66 * Calls to VFS
67 *-----*/
68
69 #define VFS_BASE 0x100
70
71 #define IS_VFS_CALL(type) (((type) & ~0xFF) == VFS_BASE)
72
73 #define VFS_READ (VFS_BASE + 0)
74 #define VFS_WRITE (VFS_BASE + 1)
75 #define VFS_LSEEK (VFS_BASE + 2)
76 #define VFS_OPEN (VFS_BASE + 3)
77 #define VFS_CREAT (VFS_BASE + 4)
78 #define VFS_CLOSE (VFS_BASE + 5)
79 #define VFS_LINK (VFS_BASE + 6)
80 #define VFS_UNLINK (VFS_BASE + 7)

Ln 63 Col 29 Spaces: 4 UTF-8 CRLF C Win32
```

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

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

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

```

File Edit Selection View Go Run Terminal Help com.h - minix-master - Visual Studio Code
EXPLORER ... h com.h 1 x h audio.fwh 2
MINIX-MASTER
libexec
minix
bin
commands
drivers
fs
include
arch
didekit
libdde
minix
acpi.h
audio.fwh
bdev.h
bitmap.h
blockdriver_mth.h
board.h
bpf.h
btrace.h
callnr.h
chardriver.h
clkconf.h
com.h 1
config.h
const.h
cpufeature.h
debug.h
...
OUTLINE > ① ② ▲ 0
Ln 271, Col 38 (6 selected) Tab Size 4 UTF-8 LF C Win32 ⌂

```

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

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

```

File Edit Selection View Go Run Terminal Help config.h - minix-master - Visual Studio Code
EXPLORER ... h config.h 1 x
MINIX-MASTER
drivers
fs
include
arch
didekit
libdde
minix
acpi.h
audio.fwh
bdev.h
bitmap.h
blockdriver_mth.h
blockdriver.h
board.h
bpf.h
btrace.h
callnr.h
chardriver.h
clkconf.h
com.h
config.h 1
const.h
cpufeature.h
debug.h
devio.h
devman.h
dimap.h
driver.h
drivers.h
...
OUTLINE > ① ② ▲ 0
Ln 83, Col 32 Tab Size 4 UTF-8 LF C++ Win32 ⌂

```

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

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

```

File Edit Selection View Go Run Terminal Help
proc.h - minix-master - Visual Studio Code
EXPLORER ... h proc.h 1 x
MINIX-MASTER
n kernel
C main.c
Makefile
h priv.h
h proc.h
h proc
C profile.c
C profile.h
h proto.h
C smp.c
h smp.h
h spinlock.h
C system.c
h system.h
C table.c
h type.h
C usermapped_data.c
C utility.c
h vm.h
C watchdog.c
h watchdog.h
> lib
> lvm
> man
> net
> sbin
> servers
> share
> tests
> OUTLINE
x 1 △ 0
minix > kernel > h proc.h 1 x
114     vir_bytes start, length; /* memory range */
115     u8_t writeflag; /* nonzero for write access */
116 } check;
117 } params;
118 /* VM result when available */
119 int vmresult;
120
121 /* If the suspended operation is a sys_call, its details are
122 * stored here.
123 */
124 } p_vmrrequest;
125
126 int p_found; /* consistency checking variables */
127 int p_magic; /* check validity of proc pointers */
128
129 /* if MF_SC_DEFER is set, this struct is valid and contains the
130 * do_ipc() arguments that are still to be executed
131 */
132 struct { reg_t r1, r2, r3; } p_defer;
133
134 #if DEBUG_TRACE
135     int p_schedules;
136 #endif
137
138 #endif /* __ASSEMBLY__ */
139
140 /* Bits for the runtime flags. A process is runnable iff p_rts_flags == 0. */
141 #define RTS_SLOT_FREE 0x01 /* process slot is free */
142 #define RTS_PROC_STOP 0x02 /* process has been stopped */
143 #define RTS_SENDING 0x04 /* process blocked trying to send */
144 #define RTS_RECEIVING 0x08 /* process blocked trying to receive */
145 #define RTS_SIGNALLED 0x10 /* set when new kernel signal arrives */
146
147 #endif /* SYSTEM_H */
Ln 138, Col 54 Tab Size:4 UTF-8 LF C Win32 ⚡ 🔍

```

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

Line 200: add the prototype of do_setsjf.

```

File Edit Selection View Go Run Terminal Help
system.h - minix-master - Visual Studio Code
EXPLORER ... h system.h 2 x
MINIX-MASTER
> .vscode
> bin
> common
> crypto
> dist
> distrib
> docs
> etc
> external
> games
> gnu
> include
> lib
> libexec
> minix
> bin
> commands
> drivers
> fs
> include
> kernel
> arch
> system
C clock.c
h clock.h
h config.h
h const.h
C cpulocals.c
h cpulocals.h
> OUTLINE
x 2 △ 0
minix > kernel > h system.h 2 x
196 int do_schedule(struct proc * caller, message *m_ptr);
197 int do_schedctl(struct proc * caller, message *m_ptr);
198
199 int do_statectl(struct proc * caller, message *m_ptr);
200 int do_setsjf(struct proc * caller, message *m_ptr);
201 #if ! USE_STATECTL
202 #define do_statectl NULL
203 #endif
204
205 int do_padconf(struct proc * caller, message *m_ptr);
206 #if ! USE_PADCONF
207 #define do_padconf NULL
208 #endif
209
210 #endif /* SYSTEM_H */
211
212
Ln 200, Col 53 Spaces: 4 UTF-8 LF C++ Win32 ⚡ 🔍

```

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

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

```
File Edit Selection View Go Run Terminal Help do_setsjf.c - minix-master - Visual Studio Code

EXPLORER
MINIX-MASTER
  - RECENTS
    - arch
    - system
      C do_abort.c
      C do_clear.c
      C do_copy.c
      C do_devio.c
      C do_diagctl.c
      C do_endksig.c
      C do_exec.c
      C do_exit.c
      C do_fork.c
      C do_getinfo.c
      C do_getksig.c
      C do_irqlc.c
      C do_kill.c
      C do_mcontext.c
      C do_memsetc.c
      C do_privl.c
      C do_rundlc.c
      C do_safecopy.c
      C do_safememset.c
      C do_schedlct.c
      C do_schedule.c
      C do_setalarmc.c
      C do_setgrantc.c
      C do_setsjf.c
      C do_settime.c
      C do_sigreturn.c
    > OUTLINE
      C do_setsjf.c 2

minix > kernel > system > C do_setsjf.c => do_setsjf(proc *, message *)
1 #include <kernel/system.h>
2 #include <stdio.h>
3 #include <minix/endpoint.h>
4 #include <minix/config.h>
5 #include <kernel/clock.h>
6 #include <errno.h>
7
8 /**
9 *          do_setsjf
10 */
11 int do_setsjf(struct proc * caller, message * m_ptr) { /* sjf_2018 */
12     struct proc *p;
13     int proc_nr, new_priority = SJF_Q;
14     new_quantum = SJF_QUANTUM, cpu = -1;
15     int result, expected_time = m_ptr->m_id;
16
17     if (expected_time < SJF_MIN_EXP_TIME || SJF_MAX_EXP_TIME < expected_time)
18         return EINVAL;
19
20     if (!isokendpt(m_ptr->m_id, &proc_nr)) {
21         printf("endpoint not ok\n");
22         return EINVAL;
23     }
24
25     p = proc_addr(proc_nr);
26
27     /* lesser TODO: Only the process itself should be able to modify its SJF status - not specified in task */
28     //if (caller != p)
29     //    return EPERM;
30
31     if (expected_time == SJF_RESET_POLICY) { // restore scheduling policy to non-SJF default
32         if (p->p_priority != SJF_Q) {
33             return EPERM; // process is already scheduled with default method
34         }
35         // TODO: remember last priority and pause it here
36     }

NR_SYS_CALLS
Aa ab No results ↑ ↓ ⌂
```

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

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

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

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

```
File Edt Selection View Go Run Terminal Help syslib.h - minix-master - Visual Studio Code

EXPLORER
MINKI MASTER
  position
    .pinch
    .profits
    .profitch
    .profitch
    .rmbuh
    .rsh
    .safeoplesh
    .schdush
    .seth
    .sfhs.h
    .sodckdrive
    .sodeeven
    .soundch
    .spinlh
    .sys.config
    .syslibh
    .sysrnlch
    .sysstat
    .sysstat
    .timersh
    .type.h
    .u64h
    .usb_chdh
    .usbh
    .vbouth
    .vbouthsh
    .vbouthsh
    .vbostype.h
    .vifdh

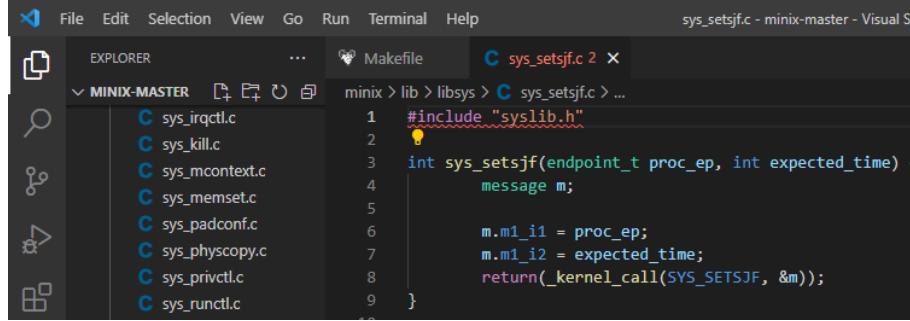
  44
  3
  51
  52
  53
  54
  55
  56
  57
  58
  59
  60
  61
  62
  63
  64
  65
  66

[minix > include > minix > syslibh > sys_setsjf(endpoint_t, int)
33     vir_bytes pc, vir_bytes ps_str);
34     int sys_fork(endpoint_t parent, endpoint_t child, endpoint_t *, 
35     u32_t va, vir_bytes * );
36     int sys_clear(endpoint_t proc_ep);
37     int sys_exit(void);
38     int sys_trace(int req, endpoint_t proc_ep, long addr, long *data_p);
39
40     int sys_schedule(endpoint_t proc_ep, int priority, int quantum, int cpu,
41     int nice);
42     int sys_schedctl(unsigned flags, endpoint_t proc_ep, int priority, int
43     int sys_setsjf(endpoint_t proc_ep, int expected_time);

44     /* Shortcuts for sys_muncl() system call. */
45 #define sys_ston(proc,ep) sys_runcnt(proc,ep, RC_STOP, 0)
46 #define sys_delay_stop(proc,ep) sys_runcnt(proc,ep, RC_STOP, RC_DELAY)
47 #define sys_resume(proc,ep) sys_runcnt(proc,ep, RC_RESUME, 0)
48
49     int sys_runcnt(endpoint_t proc_ep, int action, int flags);
50
51     int sys_update(endpoint_t src_ep, endpoint_t dst_up, int flags);
52     int sys_startred(int request, void *address, int length);
53     int sys_vmemctl(endpoint_t proc_ep, int req, void *p);
54     int sys_privary_ass(endpoint_t proc_ep, phys_bytes physstart,
55     phys_bytes physend);
56     int sys_grant(endpoint_t *grants, int ngrants);
57
58     int sys_vmembuf(phys_bytes base, phys_bytes size, phys_bytes high);
59     int sys_vmem_map(endpoint_t proc_ep, int do_map, phys_bytes base,
60     phys_bytes size, phys_bytes offset);
61     int sys_vmemcl(endpoint_t who, int param, u32_t value);
62     int sys_vmemcl(endpoint_t who, int param, u32_t value);
63     int sys_vmemcl_get_sweep(endpoint_t *who, vir_bytes *mem, vir_bytes
64     *len, int *wflag, endpoint_t *who_s, vir_bytes *mem_s, endpoint_t *);
65     int sys_vmemcl_enable_paging(void * data);
```

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

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



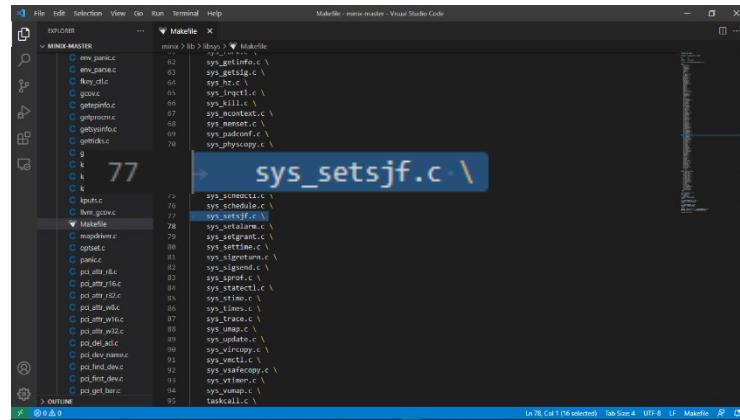
```

File Edit Selection View Go Run Terminal Help
EXPLORER Makefile C sys_setsjf.c 2 ...
MINIX-MASTER minix > lib > libsys > C sys_setsjf.c > ...
C sys_irqctl.c
C sys_kill.c
C sys_mcontext.c
C sys_memset.c
C sys_padconf.c
C sys_physcopy.c
C sys_privctl.c
C sys_runctl.c
1 #include "syslib.h"
2
3 int sys_setsjf(endpoint_t proc_ep, int expected_time) {
4     message m;
5
6     m.m1_i1 = proc_ep;
7     m.m1_i2 = expected_time;
8
9     return(_kernel_call(SYS_SETSJF, &m));
}

```

13. the file "Makefile" located in "/minix/lib/libsys"

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



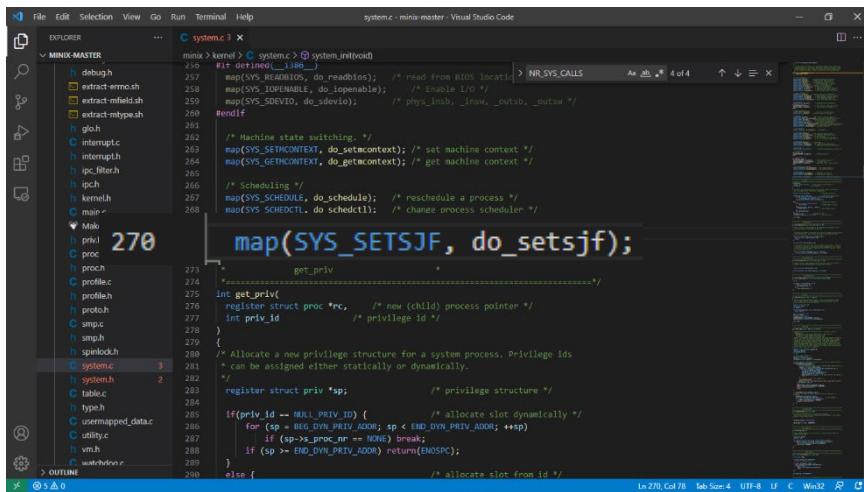
```

File Edit Selection View Go Run Terminal Help
MINIX-MASTER ... Makefile
minix > lib > libsys > Makefile
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77 sys_setsjf.c \
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95

```

14. the file "system.c" located in "/minix/kernel"

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



```

File Edit Selection View Go Run Terminal Help
MINIX-MASTER system.c
minix > kernel > C system.c > system_initvoid
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270 map(SYS_SETSJF, do_setsjf);
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290

```

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

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

```

MINIX MASTER
proto.h ... proto.h - minix - minix master - Visual Studio Code
proto.h
1 // servers > pm > proto.h
2
3 int do_getpid();
4 int do_set(pid);
5
6 /* Main system calls */
7 int do_reboot(void);
8 void reply(int proc_nr, int result);
9
10 /* Process management */
11 int do_execve(void);
12 /* Context switching */
13 int do_getcontext(void);
14 int do_setcontext(void);
15
16 /* Memory management */
17 int do_realloc(void);
18 int do_rebind(void);
19 int do_sysname(void);
20 int do_getsysinfo(void);
21 int do_getcpuinfo(void);
22 int do_getcpuinfo(void);
23
24 /* Miscellaneous */
25 int do_vfork(void);
26 int do_getpriority(void);
27 int do_getusage(void);
28
29 /* Scheduling */
30 int do_setsjf(void);
31
32 /* Profile */
33 int do_profile(void);
34
35 /* Signals */
36 int do_kill(void);
37 int do_svrlkill(int proc_nr, int signal);
38
39 int process_kill(endpoint_t proc_nr, int signal);
40
41 int do_setscheduler(endpoint_t proc_nr, int priority);
42
43 int do_setitimer(endpoint_t proc_nr, int interval);
44
45 int do_getitimer(endpoint_t proc_nr, int interval);
46
47 int do_setSIG(SIG_BLOCK, endpoint_t proc_nr, int sigmask);
48
49 int do_getSIG(SIG_BLOCK, endpoint_t proc_nr, int sigmask);
50
51 int do_setsig(SIG_BLOCK, endpoint_t proc_nr, int sigmask);
52
53 int do_getsig(SIG_BLOCK, endpoint_t proc_nr, int sigmask);
54
55 int do_setsig(SIG_BLOCK, endpoint_t proc_nr, int sigmask);
56
57 int do_getsig(SIG_BLOCK, endpoint_t proc_nr, int sigmask);
58
59 int do_setsjf(void);
59
60
61 /* profile.c */
62
63 /* misc.c */
64
65 /* misc.h */
66
67 /* misc.c */
68
69 /* misc.h */
70
71 /* misc.c */
72
73 /* misc.h */
74
75 /* misc.c */
76
77 /* misc.h */
78
79 /* misc.c */
80
81
82 /* misc.h */
83
84
85 /* misc.c */
86
87
88 /* misc.h */
89
90
91 /* misc.c */
92
93
94 /* misc.h */
95
96
97 /* misc.c */
98
99
100 /* misc.h */
101
102
103 /* misc.c */
104
105
106 /* misc.h */
107
108
109 /* misc.c */
110
111
112 /* misc.h */
113
114
115 /* misc.c */
116
117
118 /* misc.h */
119
120
121 /* misc.c */
122
123
124 /* misc.h */
125
126
127 /* misc.c */
128
129
130 /* misc.h */
131
132
133 /* misc.c */
134
135
136 /* misc.h */
137
138
139 /* misc.c */
140
141
142 /* misc.h */
143
144
145 /* misc.c */
146
147
148 /* misc.h */
149
150
151 /* misc.c */
152
153
154 /* misc.h */
155
156
157 /* misc.c */
158
159
160 /* misc.h */
161
162
163 /* misc.c */
164
165
166 /* misc.h */
167
168
169 /* misc.c */
170
171
172 /* misc.h */
173
174
175 /* misc.c */
176
177
178 /* misc.h */
179
180
181 /* misc.c */
182
183
184 /* misc.h */
185
186
187 /* misc.c */
188
189
190 /* misc.h */
191
192
193 /* misc.c */
194
195
196 /* misc.h */
197
198
199 /* misc.c */
200
201
202 /* misc.h */
203
204
205 /* misc.c */
206
207
208 /* misc.h */
209
210
211 /* misc.c */
212
213
214 /* misc.h */
215
216
217 /* misc.c */
218
219
220 /* misc.h */
221
222
223 /* misc.c */
224
225
226 /* misc.h */
227
228
229 /* misc.c */
230
231
232 /* misc.h */
233
234
235 /* misc.c */
236
237
238 /* misc.h */
239
240
241 /* misc.c */
242
243
244 /* misc.h */
245
246
247 /* misc.c */
248
249
250 /* misc.h */
251
252
253 /* misc.c */
254
255
256 /* misc.h */
257
258
259 /* misc.c */
259
260
261 /* misc.h */
262
263
264 /* misc.c */
265
266
267 /* misc.h */
268
269
270 /* misc.c */
271
272
273 /* misc.h */
274
275
276 /* misc.c */
277
278
279 /* misc.h */
279
280
281 /* misc.c */
282
283
284 /* misc.h */
285
286
287 /* misc.c */
288
289
289 /* misc.h */
290
291
292 /* misc.c */
293
294
295 /* misc.h */
296
297
298 /* misc.c */
299
300
301 /* misc.h */
302
303
304 /* misc.c */
305
306
307 /* misc.h */
308
309
309 /* misc.c */
310
311
312 /* misc.h */
313
314
315 /* misc.c */
316
317
318 /* misc.h */
319
320
321 /* misc.c */
322
323
324 /* misc.h */
325
326
327 /* misc.c */
328
329
329 /* misc.h */
330
331
332 /* misc.c */
333
334
335 /* misc.h */
336
337
338 /* misc.c */
339
340
341 /* misc.h */
342
343
344 /* misc.c */
345
346
347 /* misc.h */
348
349
350 /* misc.c */
351
352
353 /* misc.h */
354
355
356 /* misc.c */
357
358
359 /* misc.h */
359
360
361 /* misc.c */
362
363
364 /* misc.h */
365
366
367 /* misc.c */
368
369
369 /* misc.h */
370
371
372 /* misc.c */
373
374
375 /* misc.h */
376
377
378 /* misc.c */
379
380
381 /* misc.h */
382
383
384 /* misc.c */
385
386
387 /* misc.h */
388
389
389 /* misc.c */
390
391
392 /* misc.h */
393
394
395 /* misc.c */
396
397
398 /* misc.h */
399
400
401 /* misc.c */
402
403
404 /* misc.h */
405
406
407 /* misc.c */
408
409
409 /* misc.h */
410
411
412 /* misc.c */
413
414
415 /* misc.h */
416
417
418 /* misc.c */
419
420
421 /* misc.h */
422
423
424 /* misc.c */
425
426
427 /* misc.h */
428
429
429 /* misc.c */
430
431
432 /* misc.h */
433
434
435 /* misc.c */
436
437
438 /* misc.h */
439
440
441 /* misc.c */
442
443
444 /* misc.h */
445
446
447 /* misc.c */
448
449
449 /* misc.h */
450
451
452 /* misc.c */
453
454
455 /* misc.h */
456
457
458 /* misc.c */
459
460
461 /* misc.h */
462
463
464 /* misc.c */
465
466
467 /* misc.h */
468
469
469 /* misc.c */
470
471
472 /* misc.h */
473
474
475 /* misc.c */
476
477
478 /* misc.h */
479
480
481 /* misc.c */
482
483
484 /* misc.h */
485
486
487 /* misc.c */
488
489
489 /* misc.h */
490
491
492 /* misc.c */
493
494
495 /* misc.h */
496
497
498 /* misc.c */
499
500
501 /* misc.h */
502
503
504 /* misc.c */
505
506
507 /* misc.h */
508
509
509 /* misc.c */
510
511
512 /* misc.h */
513
514
515 /* misc.c */
516
517
518 /* misc.h */
519
520
521 /* misc.c */
522
523
524 /* misc.h */
525
526
527 /* misc.c */
528
529
529 /* misc.h */
530
531
532 /* misc.c */
533
534
535 /* misc.h */
536
537
538 /* misc.c */
539
540
541 /* misc.h */
542
543
544 /* misc.c */
545
546
547 /* misc.h */
548
549
549 /* misc.c */
550
551
552 /* misc.h */
553
554
555 /* misc.c */
556
557
558 /* misc.h */
559
560
561 /* misc.c */
562
563
564 /* misc.h */
565
566
567 /* misc.c */
568
569
569 /* misc.h */
570
571
572 /* misc.c */
573
574
575 /* misc.h */
576
577
578 /* misc.c */
579
580
581 /* misc.h */
582
583
584 /* misc.c */
585
586
587 /* misc.h */
588
589
589 /* misc.c */
590
591
592 /* misc.h */
593
594
595 /* misc.c */
596
597
598 /* misc.h */
599
600
601 /* misc.c */
602
603
604 /* misc.h */
605
606
607 /* misc.c */
608
609
609 /* misc.h */
610
611
612 /* misc.c */
613
614
615 /* misc.h */
616
617
618 /* misc.c */
619
620
621 /* misc.h */
622
623
624 /* misc.c */
625
626
627 /* misc.h */
628
629
629 /* misc.c */
630
631
632 /* misc.h */
633
634
635 /* misc.c */
636
637
638 /* misc.h */
639
640
641 /* misc.c */
642
643
644 /* misc.h */
645
646
647 /* misc.c */
648
649
649 /* misc.h */
650
651
652 /* misc.c */
653
654
655 /* misc.h */
656
657
658 /* misc.c */
659
660
661 /* misc.h */
662
663
664 /* misc.c */
665
666
667 /* misc.h */
668
669
669 /* misc.c */
670
671
672 /* misc.h */
673
674
675 /* misc.c */
676
677
678 /* misc.h */
679
680
681 /* misc.c */
682
683
684 /* misc.h */
685
686
687 /* misc.c */
688
689
689 /* misc.h */
690
691
692 /* misc.c */
693
694
695 /* misc.h */
696
697
698 /* misc.c */
699
700
701 /* misc.h */
702
703
704 /* misc.c */
705
706
707 /* misc.h */
708
709
709 /* misc.c */
710
711
712 /* misc.h */
713
714
715 /* misc.c */
716
717
718 /* misc.h */
719
720
721 /* misc.c */
722
723
724 /* misc.h */
725
726
727 /* misc.c */
728
729
729 /* misc.h */
730
731
732 /* misc.c */
733
734
735 /* misc.h */
736
737
738 /* misc.c */
739
740
741 /* misc.h */
742
743
744 /* misc.c */
745
746
747 /* misc.h */
748
749
749 /* misc.c */
750
751
752 /* misc.h */
753
754
755 /* misc.c */
756
757
758 /* misc.h */
759
760
761 /* misc.c */
762
763
764 /* misc.h */
765
766
767 /* misc.c */
768
769
769 /* misc.h */
770
771
772 /* misc.c */
773
774
775 /* misc.h */
776
777
778 /* misc.c */
779
780
781 /* misc.h */
782
783
784 /* misc.c */
785
786
787 /* misc.h */
788
789
789 /* misc.c */
789
790
791 /* misc.h */
792
793
794 /* misc.c */
795
796
797 /* misc.h */
798
799
799 /* misc.c */
799
800
801 /* misc.h */
802
803
804 /* misc.c */
805
806
807 /* misc.h */
808
809
809 /* misc.c */
809
810
811 /* misc.h */
812
813
814 /* misc.c */
815
816
817 /* misc.h */
818
819
819 /* misc.c */
819
820
821 /* misc.h */
822
823
824 /* misc.c */
825
826
827 /* misc.h */
828
829
829 /* misc.c */
829
830
831 /* misc.h */
832
833
834 /* misc.c */
835
836
837 /* misc.h */
838
839
839 /* misc.c */
839
840
841 /* misc.h */
842
843
844 /* misc.c */
845
846
847 /* misc.h */
848
849
849 /* misc.c */
849
850
851 /* misc.h */
852
853
854 /* misc.c */
855
856
857 /* misc.h */
858
859
859 /* misc.c */
859
860
861 /* misc.h */
862
863
864 /* misc.c */
865
866
867 /* misc.h */
868
869
869 /* misc.c */
869
870
871 /* misc.h */
872
873
874 /* misc.c */
875
876
877 /* misc.h */
878
879
879 /* misc.c */
879
880
881 /* misc.h */
882
883
884 /* misc.c */
885
886
887 /* misc.h */
888
889
889 /* misc.c */
889
890
891 /* misc.h */
892
893
894 /* misc.c */
895
896
897 /* misc.h */
898
899
899 /* misc.c */
899
900
901 /* misc.h */
902
903
904 /* misc.c */
905
906
907 /* misc.h */
908
909
909 /* misc.c */
909
910
911 /* misc.h */
912
913
914 /* misc.c */
915
916
917 /* misc.h */
918
919
919 /* misc.c */
919
920
921 /* misc.h */
922
923
924 /* misc.c */
925
926
927 /* misc.h */
928
929
929 /* misc.c */
929
930
931 /* misc.h */
932
933
934 /* misc.c */
935
936
937 /* misc.h */
938
939
939 /* misc.c */
939
940
941 /* misc.h */
942
943
944 /* misc.c */
945
946
947 /* misc.h */
948
949
949 /* misc.c */
949
950
951 /* misc.h */
952
953
954 /* misc.c */
955
956
957 /* misc.h */
958
959
959 /* misc.c */
959
960
961 /* misc.h */
962
963
964 /* misc.c */
965
966
967 /* misc.h */
968
969
969 /* misc.c */
969
970
971 /* misc.h */
972
973
974 /* misc.c */
975
976
977 /* misc.h */
978
979
979 /* misc.c */
979
980
981 /* misc.h */
982
983
984 /* misc.c */
985
986
987 /* misc.h */
988
989
989 /* misc.c */
989
990
991 /* misc.h */
992
993
994 /* misc.c */
995
996
997 /* misc.h */
998
999
999 /* misc.c */
999
1000
1001 /* misc.h */
1002
1003
1004 /* misc.c */
1005
1006
1007 /* misc.h */
1008
1009
1009 /* misc.c */
1009
1010
1011 /* misc.h */
1012
1013
1014 /* misc.c */
1015
1016
1017 /* misc.h */
1018
1019
1019 /* misc.c */
1019
1020
1021 /* misc.h */
1022
1023
1024 /* misc.c */
1025
1026
1027 /* misc.h */
1028
1029
1029 /* misc.c */
1029
1030
1031 /* misc.h */
1032
1033
1034 /* misc.c */
1035
1036
1037 /* misc.h */
1038
1039
1039 /* misc.c */
1039
1040
1041 /* misc.h */
1042
1043
1044 /* misc.c */
1045
1046
1047 /* misc.h */
1048
1049
1049 /* misc.c */
1049
1050
1051 /* misc.h */
1052
1053
1054 /* misc.c */
1055
1056
1057 /* misc.h */
1058
1059
1059 /* misc.c */
1059
1060
1061 /* misc.h */
1062
1063
1064 /* misc.c */
1065
1066
1067 /* misc.h */
1068
1069
1069 /* misc.c */
1069
1070
1071 /* misc.h */
1072
1073
1074 /* misc.c */
1075
1076
1077 /* misc.h */
1078
1079
1079 /* misc.c */
1079
1080
1081 /* misc.h */
1082
1083
1084 /* misc.c */
1085
1086
1087 /* misc.h */
1088
1089
1089 /* misc.c */
1089
1090
1091 /* misc.h */
1092
1093
1094 /* misc.c */
1095
1096
1097 /* misc.h */
1098
1099
1099 /* misc.c */
1099
1100
1101 /* misc.h */
1102
1103
1104 /* misc.c */
1105
1106
1107 /* misc.h */
1108
1109
1109 /* misc.c */
1109
1110
1111 /* misc.h */
1112
1113
1114 /* misc.c */
1115
1116
1117 /* misc.h */
1118
1119
1119 /* misc.c */
1119
1120
1121 /* misc.h */
1122
1123
1124 /* misc.c */
1125
1126
1127 /* misc.h */
1128
1129
1129 /* misc.c */
1129
1130
1131 /* misc.h */
1132
1133
1134 /* misc.c */
1135
1136
1137 /* misc.h */
1138
1139
1139 /* misc.c */
1139
1140
1141 /* misc.h */
1142
1143
1144 /* misc.c */
1145
1146
1147 /* misc.h */
1148
1149
1149 /* misc.c */
1149
1150
1151 /* misc.h */
1152
1153
1154 /* misc.c */
1155
1156
1157 /* misc.h */
1158
1159
1159 /* misc.c */
1159
1160
1161 /* misc.h */
1162
1163
1164 /* misc.c */
1165
1166
1167 /* misc.h */
1168
1169
1169 /* misc.c */
1169
1170
1171 /* misc.h */
1172
1173
1174 /* misc.c */
1175
1176
1177 /* misc.h */
1178
1179
1179 /* misc.c */
1179
1180
1181 /* misc.h */
1182
1183
1184 /* misc.c */
1185
1186
1187 /* misc.h */
1188
1189
1189 /* misc.c */
1189
1190
1191 /* misc.h */
1192
1193
1194 /* misc.c */
1195
1196
1197 /* misc.h */
1198
1199
1199 /* misc.c */
1199
1200
1201 /* misc.h */
1202
1203
1204 /* misc.c */
1205
1206
1207 /* misc.h */
1208
1209
1209 /* misc.c */
1209
1210
1211 /* misc.h */
1212
1213
1214 /* misc.c */
1215
1216
1217 /* misc.h */
1218
1219
1219 /* misc.c */
1219
1220
1221 /* misc.h */
1222
1223
1224 /* misc.c */
1225
1226
1227 /* misc.h */
1228
1229
1229 /* misc.c */
1229
1230
1231 /* misc.h */
1232
1233
1234 /* misc.c */
1235
1236
1237 /* misc.h */
1238
1239
1239 /* misc.c */
1239
1240
1241 /* misc.h */
1242
1243
1244 /* misc.c */
1245
1246
1247 /* misc.h */
1248
1249
1249 /* misc.c */
1249
1250
1251 /* misc.h */
1252
1253
1254 /* misc.c */
1255
1256
1257 /* misc.h */
1258
1259
1259 /* misc.c */
1259
1260
1261 /* misc.h */
1262
1263
1264 /* misc.c */
1265
1266
1267 /* misc.h */
1268
1269
1269 /* misc.c */
1269
1270
1271 /* misc.h */
1272
1273
1274 /* misc.c */
1275
1276
1277 /* misc.h */
1278
1279
1279 /* misc.c */
1279
1280
1281 /* misc.h */
1282
1283
1284 /* misc.c */
1285
1286
1287 /* misc.h */
1288
1289
1289 /* misc.c */
1289
1290
1291 /* misc.h */
1292
1293
1294 /* misc.c */
1295
1296
1297 /* misc.h */
1298
1299
1299 /* misc.c */
1299
1300
1301 /* misc.h */
1302
1303
1304 /* misc.c */
1305
1306
1307 /* misc.h */
1308
1309
1309 /* misc.c */
1309
1310
1311 /* misc.h */
1312
1313
1314 /* misc.c */
1315
1316
1317 /* misc.h */
1318
1319
1319 /* misc.c */
1319
1320
1321 /* misc.h */
1322
1323
1324 /* misc.c */
1325
1326
1327 /* misc.h */
1328
1329
1329 /* misc.c */
1329
1330
1331 /* misc.h */
1332
1333
1334 /* misc.c */
1335
1336
1337 /* misc.h */
1338
1339
1339 /* misc.c */
1339
1340
1341 /* misc.h */
1342
1343
1344 /* misc.c */
1345
1346
1347 /* misc.h */
1348
1349
1349 /* misc.c */
1349
1350
1351 /* misc.h */
1352
1353
1354 /* misc.c */
1355
1356
1357 /* misc.h */
1358
1359
1359 /* misc.c */
1359
1360
1361 /* misc.h */
1362
1363
1364 /* misc.c */
1365
1366
1367 /* misc.h */
1368
1369
1369 /* misc.c */
1369
1370
1371 /* misc.h */
1372
1373
1374 /* misc.c */
1375
1376
1377 /* misc.h */
1378
1379
1379 /* misc.c */
1379
1380
1381 /* misc.h */
1382
1383
1384 /* misc.c */
1385
1386
1387 /* misc.h */
1388
1389
1389 /* misc.c */
1389
1390
1391 /* misc.h */
1392
1393
1394 /* misc.c */
1395
1396
1397 /* misc.h */
1398
1399
1399 /* misc.c */
1399
1400
1401 /* misc.h */
1402
1403
1404 /* misc.c */
1405
1406
1407 /* misc.h */
1408
1409
1409 /* misc.c */
1409
1410
1411 /* misc.h */
1412
1413
1414 /* misc.c */
1415
1416
1417 /* misc.h */
1418
1419
1419 /* misc.c */
1419
1420
1421 /* misc.h */
1422
1423
1424 /* misc.c */
1425
1426
1427 /* misc.h */
1428
1429
1429 /* misc.c */
1429
1430
1431 /* misc.h */
1432
1433
1434 /* misc.c */
1435
1436
1437 /* misc.h */
1438
1439
1439 /* misc.c */
1439
1440
1441 /* misc.h */
1442
1443
1444 /* misc.c */
1445
1446
1447 /* misc.h */
1448
1449
1449 /* misc.c */
1449
1450
1451 /* misc.h */
1452
1453
1454 /* misc.c */
1455
1456
1457 /* misc.h */
1458
1459
1459 /* misc.c */
1459
1460
1461 /* misc.h */
1462
1463
1464 /* misc.c */
1465
1466
1467 /* misc.h */
1468
1469
1469 /* misc.c */
1469
1470
1471 /* misc.h */
1472
1473
1474 /* misc.c */
1475
1476
1477 /* misc.h */
1478
1479
1479 /* misc.c */
1479
1480
1481 /* misc.h */
1482
1483
1484 /* misc.c */
1485
1486
1487 /* misc.h */
1488
1489
1489 /* misc.c */
1489
1490
1491 /* misc.h */
1492
1493
1494 /* misc.c */
1495
1496
1497 /* misc.h */
1498
1499
1499 /* misc.c */
1499
1500
1501 /* misc.h */
1502
1503
1504 /* misc.c */
1505
1506
1507 /* misc.h */
1508
1509
1509 /* misc.c */
1509
1510
1511 /* misc.h */
1512
1513
1514 /* misc.c */
1515
1516
1517 /* misc.h */
1518
1519
1519 /* misc.c */
1519
1520
1521 /* misc.h */
1522
1523
1524 /* misc.c */
1525
1526
1527 /* misc.h */
1528
1529
1529 /* misc.c */
1529
1530
1531 /* misc.h */
1532
1533
1534 /* misc.c */
1535
1536
1537 /* misc.h */
1538
1539
1539 /* misc.c */
1539
1540
1541 /* misc.h */
1542
1543
1544 /* misc.c */
1545
1546
1547 /* misc.h */
1548
1549
1549 /* misc.c */
1549
1550
1551 /* misc.h */
1552
1553
1554 /* misc.c */
1555
1556
1557 /* misc.h */
1558
1559
1559 /* misc.c */
1559
1560
1561 /* misc.h */
1562
1563
1564 /* misc.c */
1565
1566
1567 /* misc.h */
1568
1569
1569 /* misc.c */
1569
1570
1571 /* misc.h */
1572
1573
1574 /* misc.c */
1575
1576
1577 /* misc.h */
1578
1579
1579 /* misc.c */
1579
1580
1581 /* misc.h */
1582
1583
1584 /* misc.c */
1585
1586
1587 /* misc.h */
1588
1589
1589 /* misc.c */
1589
1590
1591 /* misc.h */
1592
1593
1594 /* misc.c */
1595
1596
1597 /* misc.h */
1598
1599
1599 /* misc.c */
1599
1600
1601 /* misc.h */
1602
1603
1604 /* misc.c */
1605
1606
1607 /* misc.h */
1608
1609
1609 /* misc.c */
1609
1610
1611 /* misc.h */
1612
1613
1614 /* misc.c */
1615
1616
1617 /* misc.h */
1618
1619
1619 /* misc.c */
1619
1620
1621 /* misc.h */
1622
1623
1624 /* misc.c */
1625
1626
1627 /* misc.h */
1628
1629
1629 /* misc.c */
1629
1630
1631 /* misc.h */
1632
1633
1634 /* misc.c */
1635
1636
1637 /* misc.h */
1638
1639
1639 /* misc.c */
1639
1640
1641 /* misc.h */
1642
1643
1644 /* misc.c */
1645
1646
1647 /* misc.h */
1648
1649
1649 /* misc.c */
1649
1650
1651 /* misc.h */
1652
1653
1654 /* misc.c */
1655
1656
1657 /* misc.h */
1658
1659
1659 /* misc.c */
1659
1660
1661 /* misc.h */
1662
1663
1664 /* misc.c */
1665
1666
1667 /* misc.h */
1668
1669
1669 /* misc.c */
1669
1670
1671 /* misc.h */
1672
1673
1674 /* misc.c */
1675
1676
1677 /* misc.h */
1678
1679
1679 /* misc.c */
1679
1680
1681 /* misc.h */
1682
1683
1684 /* misc.c */
1685
1686
1687 /* misc.h */
1688
1689
1689 /* misc.c */
1689
1690
1691 /* misc.h */
1692
1693
1694 /* misc.c */
1695
1696
1697 /* misc.h */
1698
1699
1699 /* misc.c */
1699
1700
1701 /* misc.h */
1702
1703
1704 /* misc.c */
1705
1706
1707 /* misc.h */
1708
1709
1709 /* misc.c */
1709
1710
1711 /* misc.h */
1712
1713
1714 /* misc.c */
1715
1716
1717 /* misc.h */
1718
1719
1719 /* misc.c */
1719
1720
1721 /* misc.h */
1722
1723
1724 /* misc.c */
1725
1726
1727 /* misc.h */
1728
1729
1729 /* misc.c */
1729
1730
1731 /* misc.h */
1732
1733
1734 /* misc.c */
1735
1736
1737 /* misc.h */
1738
1739
1739 /* misc.c */
1739
1740
1741 /* misc.h */
1742
1743
1744 /* misc.c */
1745
1746
1747 /* misc.h */
1748
1749
1749 /* misc.c */
1749
1750
1751 /* misc.h */
1752
1753
1754 /* misc.c */
1755
1756
1757 /* misc.h */
1758
1759
1759 /* misc.c */
1759
1760
1761 /* misc.h */
1762
1763
1764 /* misc.c */
1765
```

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

```

    ...
    #define IPC_BASE 0x0000
    ...
    /* Shared Memory */
    #define IPC_SHMGET (IPC_BASE+1)
    #define IPC_SHMSET (IPC_BASE+2)
    #define IPC_SHMOT (IPC_BASE+3)
    #define IPC_SHMCTL (IPC_BASE+4)
    ...
    /* Sockets */
    #define IPC_SOCKET (IPC_BASE+5)
    #define IPC_NFSCTL (IPC_BASE+6)
    #define IPC_SCMOP (IPC_BASE+7)
    ...
    /* Messages for Scheduling */
    #define SCHEDULING_BASE 0x100
    ...
    #define SCHEDULING_NO_QUANTUM (SCHEDULING_BASE+1)
    #define SCHEDULING_START (SCHEDULING_BASE+2)
    #define SCHEDULING_STOP (SCHEDULING_BASE+3)
    #define SCHEDULING_SETSJF (SCHEDULING_BASE+4)
    ...
    #define USB_BASE 0x1100
    ...
    /* Those are from driver to USB */
    ...

```

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

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

```

    ...
    1608 struct proc **my_head = &rdy_head[q];
    1609 assert(proc_is_runnable(rp));
    1610
    1622 else if (q == SJF_Q) {
    1623     struct proc **my_head = &rdy_head[q];
    1624     while (*my_head != NULL && (*my_head)->expected_time <= rp->expected_time) {
    1625         my_head = &(*my_head)->p_nextready;
    1626     }
    1627     /* Insert rp before *my_head */
    1628     if (my_head == NULL) { // inserting at the end of list
    1629         rdy_tail[q] = rp; /* set new queue tail */
    1630     }
    1631     rp->p_nextready = *my_head;
    1632     *my_head = rp;
    1633 }
    1634 else { /* add to tail of queue */
    1635     rdy_tail[q]->p_nextready = rp; /* chain tail of queue */
    1636     rdy_tail[q] = rp; /* set new queue tail */
    1637     rp->p_nextready = NULL; /* mark new end */
    1638 }
    ...

```

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

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

```

    ...
    60     case SCHEDULING_SETSJF:
    61         result = do_setsjf(&m_in);
    62         break;
    63     case SCHEDULING_NO_QUANTUM:
    64         /* This message was sent from the kernel, don't reply */
    65         if ((IPC_STATUS_FLAGS_TEST(ipc_status,
    66             IPC_FLAG_NO_REPLY) || ipcm_flag == IPC_FLAG_FROM_KERNEL))
    67         {
    68             if (((m_in.do_noquantum&1)<1)) {
    69                 /* If (m_in.do_noquantum&1)<1, then do_noquantum is 0 */
    70                 /* If do_noquantum is 0, then do_noquantum<1 */
    71                 /* So if (m_in.do_noquantum&1)<1, then (m_in.do_noquantum<1) is true */
    72                 /* So if (m_in.do_noquantum&1)<1, then (m_in.do_noquantum<1) is true */
    73                 /* So if (m_in.do_noquantum&1)<1, then (m_in.do_noquantum<1) is true */
    74                 /* So if (m_in.do_noquantum&1)<1, then (m_in.do_noquantum<1) is true */
    75                 /* So if (m_in.do_noquantum&1)<1, then (m_in.do_noquantum<1) is true */
    76                 /* So if (m_in.do_noquantum&1)<1, then (m_in.do_noquantum<1) is true */
    77                 /* So if (m_in.do_noquantum&1)<1, then (m_in.do_noquantum<1) is true */
    78                 /* So if (m_in.do_noquantum&1)<1, then (m_in.do_noquantum<1) is true */
    79                 /* So if (m_in.do_noquantum&1)<1, then (m_in.do_noquantum<1) is true */
    80                 /* So if (m_in.do_noquantum&1)<1, then (m_in.do_noquantum<1) is true */
    81                 /* So if (m_in.do_noquantum&1)<1, then (m_in.do_noquantum<1) is true */
    82                 /* So if (m_in.do_noquantum&1)<1, then (m_in.do_noquantum<1) is true */
    83                 /* So if (m_in.do_noquantum&1)<1, then (m_in.do_noquantum<1) is true */
    84             }
    85             continue; /* Don't reply */
    86         }
    87         else {
    88             /* m_in.do_noquantum is 1 */
    89             /* So if (m_in.do_noquantum&1)<1, then (m_in.do_noquantum<1) is false */
    90             /* So if (m_in.do_noquantum&1)<1, then (m_in.do_noquantum<1) is false */
    91             /* So if (m_in.do_noquantum&1)<1, then (m_in.do_noquantum<1) is false */
    92             /* So if (m_in.do_noquantum&1)<1, then (m_in.do_noquantum<1) is false */
    93             /* So if (m_in.do_noquantum&1)<1, then (m_in.do_noquantum<1) is false */
    94             /* So if (m_in.do_noquantum&1)<1, then (m_in.do_noquantum<1) is false */
    95             /* So if (m_in.do_noquantum&1)<1, then (m_in.do_noquantum<1) is false */
    ...

```

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

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

```
int do_setsjf(message *m_ptr);
```

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

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

```
int do_setsjf(message *m_ptr) /* sjf_2018 */ {
    register struct schedproc *rmp;
    int expected_priority = m_ptr->m1_il;
    int rv, proc_nr_n;

    if (sched_isokendpt(m_ptr->m1_i2, &proc_nr_n) != OK) {
        printf("SCHED: WARNING: got an invalid endpoint in OCQ msg %u.\n",
               m_ptr->m_source);
        return EINVAL;
    }

    rmp = &schedproc[proc_nr_n];

    rmp->priority = expected_priority;
    return OK;
}
```

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

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

```
void change_priority(int *priority, int diff) {
    *priority += diff;
    if (*priority == SJF_Q)
        *priority += diff;
}
```

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

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

```
File Edit Selection View Go Run Terminal Help • schedule.c - minix-master - Visual Studio Code

EXPLORER
> MINIX MASTER
> > domain
> > dr
> > input
> > i44
> > is
> > mib
> > pm
> > rs
> > sched
> > man4c
> > Makefile
> > protos
> > schedt
> > utilproct
318
319     if (new_q == SJF_Q) {
320         new_q += 1;
321     }
322
323     if (old_q == SJF_Q) {
324         printf("SCHED: WARNING: such do_nice modification would disrupt SJF behaviour");
325         return EINVAL;
326     }
327
328     if ((rv = schedule_process_local(mp)) != OK) {
329         /* Something went wrong when rescheduling the process, roll
330         * back the changes to proc struct */
331
332         if (new_q > NR_SCHED_QUEUES) {
333             return EINVAL;
334         }
335
336         /* Store old values, in case we need to roll back the changes */
337
338         if (new_q == SJF_Q) {
339             new_q -= 1;
340         }
341
342         if (old_q == SJF_Q) {
343             new_q -= 1;
344         }
345
346         if ((rv = schedule_process_local(mp)) != OK) {
347             /* Something went wrong when rescheduling the process, roll
348             * back the changes to proc struct */
349             return EIO;
350         }
351
352         /* Success */
353         return OK;
354     }
355
356     /* Success */
357     return OK;
358 }
```

2.4. Priority-Based Scheduling

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

2.4.1. Advantages

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

2.4.2. Disadvantages

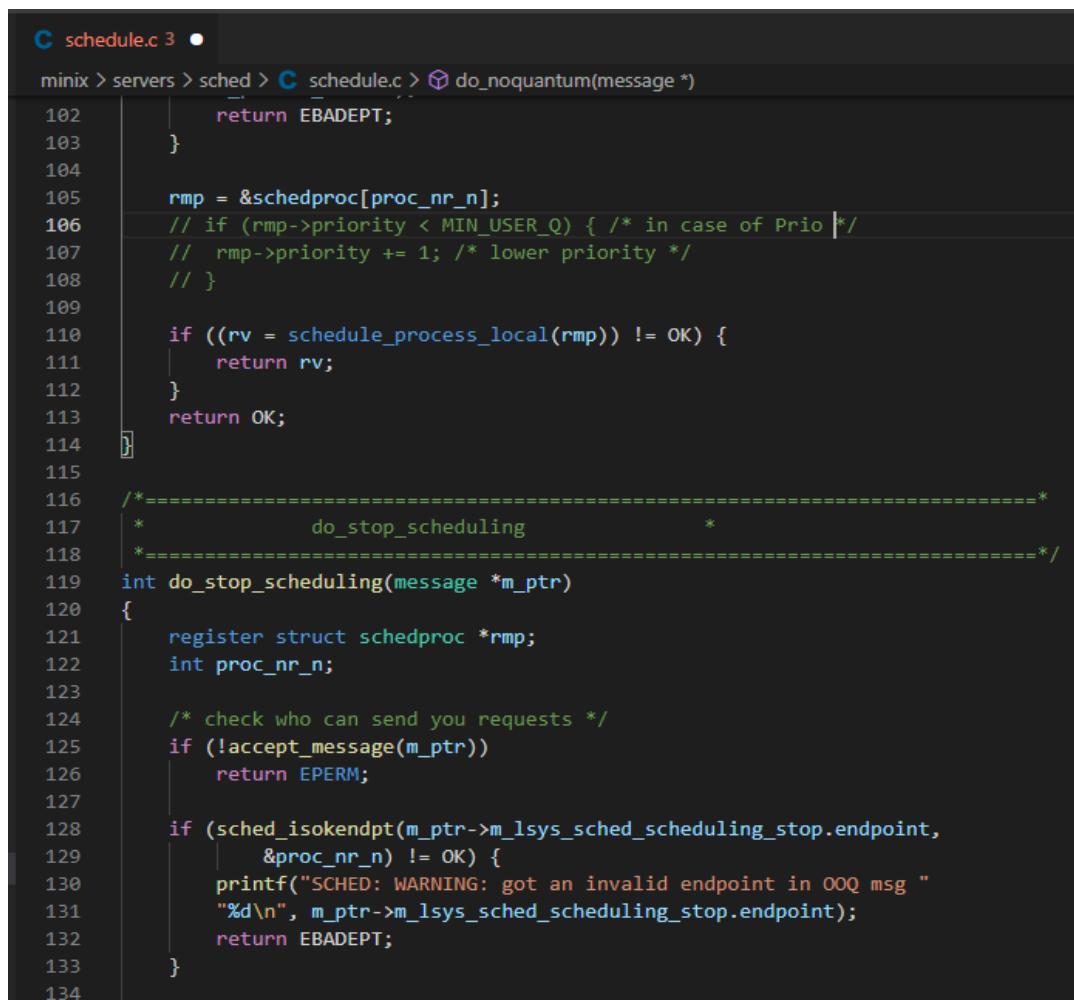
- Might cause starvation of less important processes in case processes of higher importance keeps entering the queue.

2.4.3. Implementing priority scheduling in MINIX 3

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

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

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



The screenshot shows a terminal window with the title 'C schedule.c 3'. The path 'minix > servers > sched > C schedule.c > do_noquantum(message *)' is displayed above the code. The code itself is a C program with line numbers from 102 to 134. Lines 106 through 109 are highlighted with a grey background, indicating they have been commented out. The code handles scheduling logic, including a check for a minimum user priority (MIN_USER_Q), a call to 'schedule_process_local', and a return statement. It also includes a section for 'do_stop_scheduling' and a check for accepting messages.

```
minix > servers > sched > C schedule.c > do_noquantum(message *)
102     return EBADEPT;
103 }
104
105 rmp = &schedproc[proc_nr_n];
106 // if (rmp->priority < MIN_USER_Q) { /* in case of Prio */
107 //   rmp->priority += 1; /* lower priority */
108 // }
109
110 if ((rv = schedule_process_local(rmp)) != OK) {
111     return rv;
112 }
113 return OK;
114 }
115
116 /*=====
117 *           do_stop_scheduling
118 *=====
119 int do_stop_scheduling(message *m_ptr)
120 {
121     register struct schedproc *rmp;
122     int proc_nr_n;
123
124     /* check who can send you requests */
125     if (!accept_message(m_ptr))
126         return EPERM;
127
128     if (sched_isokendpt(m_ptr->m_lsys_sched_scheduling_stop.endpoint,
129                         &proc_nr_n) != OK) {
130         printf("SCHED: WARNING: got an invalid endpoint in OQ msg "
131               "%d\n", m_ptr->m_lsys_sched_scheduling_stop.endpoint);
132         return EBADEPT;
133     }
134 }
```

Figure 2: Priority Scheduling Code

2.5. Multi-Level Feedback Queue

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

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

2.5.1. Implementing MFQ in MINIX 3

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

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

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

```
minix > include > minix > h config.h > MAX_USER_Q
      -_
53
54  /* Max. number of device memory ranges that can be assigned to a process */
55  #define NR_MEM_RANGE 20
56
57  /* Max. number of IRQs that can be assigned to a process */
58  #define NR_IRQ 16
59
60  /* Max. number of domains (protocol families) per socket driver */
61  #define NR_DOMAIN 8
62
63  /* Scheduling priorities. Values must start at zero (highest
   | * priority) and increment.
   | */
64
65  #define NR_SCHED_QUEUES 19 /* MUST equal minimum priority + 1 */
66  #define TASK_Q 0 /* highest, used for kernel tasks */
67  #define MAX_USER_Q 16 /* highest priority for user processes */
68  #define USER_Q ((MIN_USER_Q - MAX_USER_Q) / 2 + MAX_USER_Q) /* default
   | (should correspond to nice 0) */
69  #define MIN_USER_Q (NR_SCHED_QUEUES - 1) /* minimum priority for user
   | processes */
70  #define USER_QUANTUM 200
71
72  /* default scheduling quanta */
73  #define USER_DEFAULT_CPU -1 /* use the default cpu or do not change the
   | current one */
74
75
76  /* default user process cpu */
77  #define USER_DEFAULT_CPU -1 /* use the default cpu or do not change the
   | current one */
78
```

Figure 3: MFQ queues 16-18 definition

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

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

```
h schedproc.h 2 x
minix > servers > sched > h schedproc.h 56 schedproc > quantum

17
18 /**
19 * We might later want to add more information to this table, such as the
20 * process owner, process group or cpumask.
21 */
22
23 EXTERN struct schedproc {
24     endpoint_t endpoint; /* process endpoint id */
25     endpoint_t parent; /* parent endpoint id */
26     unsigned flags; /* flag bits */
27
28     /* User space scheduling */
29     unsigned max_priority; /* this process' highest allowed priority */
30     unsigned priority; /* the process' current priority */
31     /* ... */
32     /* this pointer's type is always */
33     /* ... */
34     /* ... */
35     /* ... */
36     /* ... */
37 } schedproc[NR_PROCS];
38
39 /* Flag values */
40 #define IN_USE      0x00001 /* set when 'schedproc' slot in use */

33
```

Figure 4: MFQ quantum

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

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

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

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

Figure 5: MFQ Code 3

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

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

```
minix > servers > sched > C schedule.c > do_start_scheduling(message *)
153
154     /* Resolve endpoint to proc slot. */
155     if ((rv = sched_isemptyendpt(m_ptr->m_lsys_sched_scheduling_start.endpoint,
156         &proc_nr_n)) != OK) {
157         return rv;
158     }
159     rmp = &schedproc[proc_nr_n];
160
161     /* Populate process slot */
162     rmp->endpoint      = m_ptr->m_lsys_sched_scheduling_start.endpoint;
163     rmp->parent        = m_ptr->m_lsys_sched_scheduling_start.parent;
164     rmp->max_priority  = m_ptr->m_lsys_sched_scheduling_start.maxprio;
165     + rmp->quantum = 0;
166     if (rmp->max_priority >= NR_SCHED_QUEUES) {
167         return EINVAL;
168     }
169
```

Figure 6: MFQ Code 4

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

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

```
minix > servers > sched > C schedule.c > do_start_scheduling(message *)
151     if (!accept_message(m_ptr))
152         return EPERM;
153
154     /* Resolve endpoint to proc slot. */
155     if ((rv = sched_isemptyendpt(m_ptr->m_lsys_sched_scheduling_start.endpoint,
156         &proc_nr_n)) != OK) {
157         return rv;
158     }
159     rmp = &schedproc[proc_nr_n];
160
161     /* Populate process slot */
162     rmp->endpoint      = m_ptr->m_lsys_sched_scheduling_start.endpoint;
163     rmp->parent        = m_ptr->m_lsys_sched_scheduling_start.parent;
164     rmp->max_priority  = m_ptr->m_lsys_sched_scheduling_start.maxprio;
165     // rmp->quantum = 0; /* initializing quantum for MFQ */
166     if (rmp->max_priority >= NR_SCHED_QUEUES) {
167         return EINVAL;
168     }
169
170     /* Inherit current priority and time slice from parent. Since there
171     * is currently only one scheduler scheduling the whole system, this
172     * value is local and we assert that the parent endpoint is valid */
173     if (rmp->endpoint == rmp->parent) {
174         /* We have a special case here for init, which is the first
175         * process scheduled, and the parent of itself. */
176         rmp->priority = MAX_USER_Q;
177         rmp->time_slice = DEFAULT_USER_TIME_SLICE;
178
179         /*
180          * Since kernel never changes the cpu of a process, all are
181          * started on the BSP and the userspace scheduling hasn't
182          * changed that yet either, we can be sure that BSP is the
183          * processor where the processes run now.
184          */
185 #ifdef CONFIG_SMP
```

Figure 7: MFQ Code 5

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

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

Figure 8: MFQ do_nice()

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

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

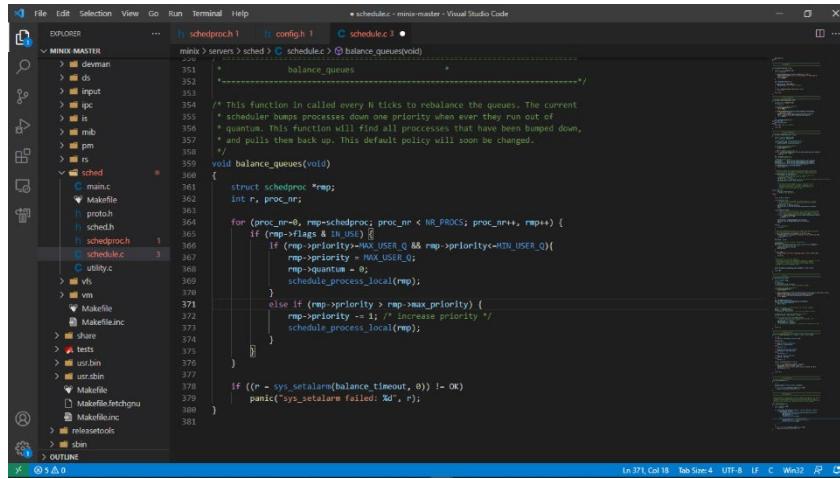


Figure 9 MFQ balance_queues()

2.5.2. Testing MFQ algorithm

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

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

Figure 10: MFO execution example

2.6. Testing Scheduling Algorithms

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

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

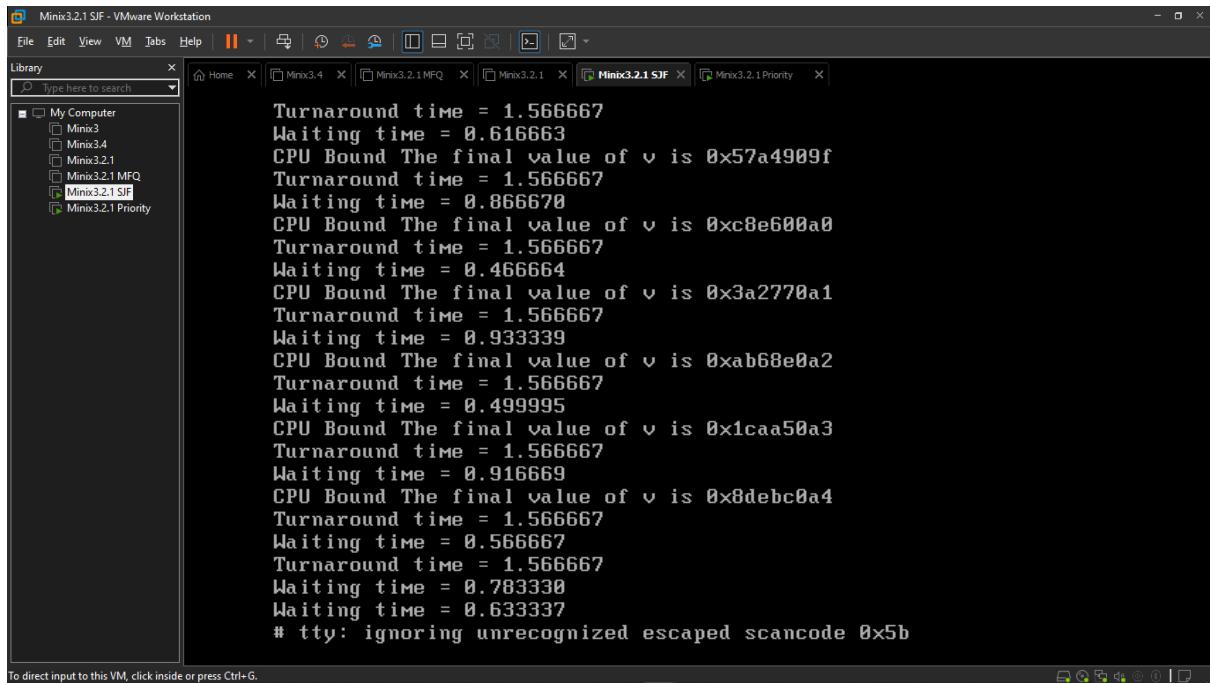
Waiting time = Turnaround time – Execution time

2.6.1. Testing RR

```
CPU Bound The final value of v is 0x25b5920f
Turnaround time = 1.583333
Waiting time = 0.583330
CPU Bound The final value of v is 0x96f70210
Turnaround time = 1.583333
Waiting time = 0.800000
CPU Bound The final value of v is 0x00387211
Turnaround time = 1.583333
Waiting time = 0.683335
CPU Bound The final value of v is 0x7979e212
Turnaround time = 1.583333
Waiting time = 0.766663
CPU Bound The final value of v is 0xeabb5213
Turnaround time = 1.583333
Waiting time = 0.650005
CPU Bound The final value of v is 0x5bfcc214
Turnaround time = 1.600000
Waiting time = 0.799994
CPU Bound The final value of v is 0xcd3e3215
Turnaround time = 1.600000
Waiting time = 0.700006
Turnaround time = 1.600000
Waiting time = 0.733327
Waiting time = 0.666673
# _
```

Figure 11: Testing Default RR

2.6.2. Testing SJF

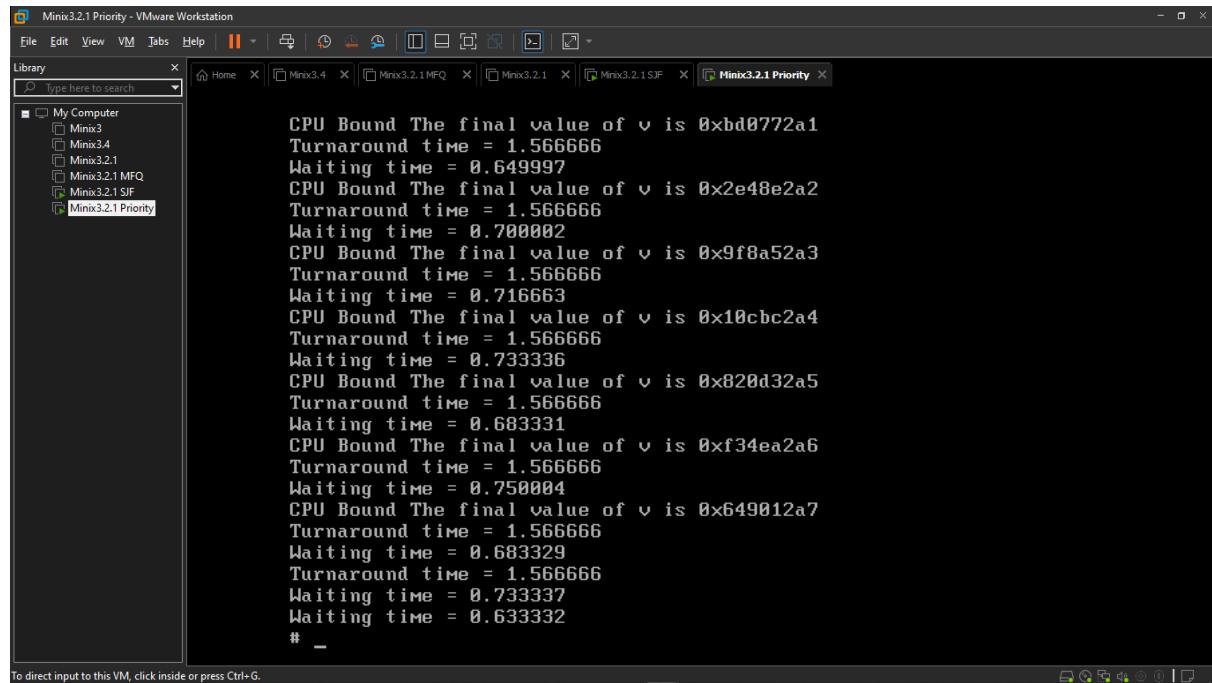


```
Minix3.2.1 SJF - VMware Workstation
File Edit View VM Tabs Help || Library Home Minix3.4 Minix3.2.1 MFQ Minix3.2.1 Minix3.2.1 SJF Minix3.2.1 Priority
Turnaround time = 1.566667
Waiting time = 0.616663
CPU Bound The final value of v is 0x57a4909f
Turnaround time = 1.566667
Waiting time = 0.866660
CPU Bound The final value of v is 0xc8e600a0
Turnaround time = 1.566667
Waiting time = 0.466664
CPU Bound The final value of v is 0x3a2770a1
Turnaround time = 1.566667
Waiting time = 0.933339
CPU Bound The final value of v is 0xab68e0a2
Turnaround time = 1.566667
Waiting time = 0.499995
CPU Bound The final value of v is 0x1caa50a3
Turnaround time = 1.566667
Waiting time = 0.916669
CPU Bound The final value of v is 0x8debc0a4
Turnaround time = 1.566667
Waiting time = 0.566667
Turnaround time = 1.566667
Waiting time = 0.783330
Waiting time = 0.633337
# tty: ignoring unrecognized escaped scancode 0x5b

To direct input to this VM, click inside or press Ctrl+G.
```

Figure 12: Testing SJF

2.6.3. Testing priority-based queue



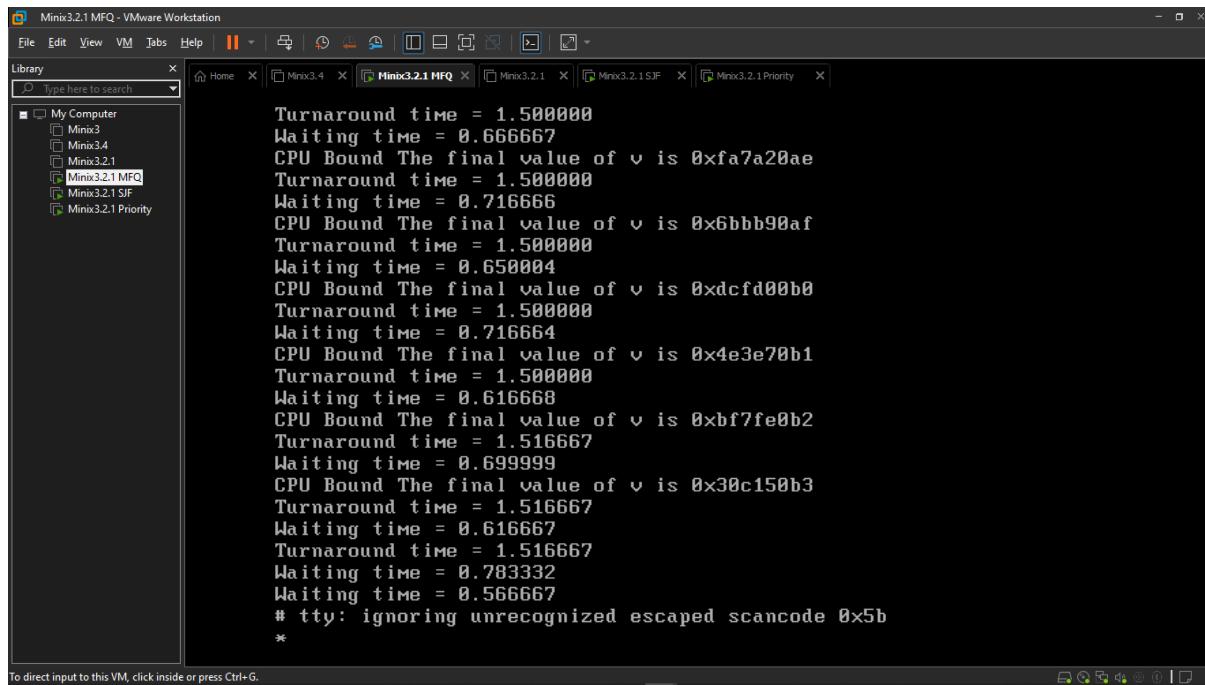
The screenshot shows a VMware Workstation interface with a single VM named "Minix3.2.1 Priority". The VM window displays a terminal session with the following output:

```
CPU Bound The final value of v is 0xbd0772a1
Turnaround time = 1.566666
Waiting time = 0.649997
CPU Bound The final value of v is 0x2e48e2a2
Turnaround time = 1.566666
Waiting time = 0.700002
CPU Bound The final value of v is 0x9f8a52a3
Turnaround time = 1.566666
Waiting time = 0.716663
CPU Bound The final value of v is 0x10cbc2a4
Turnaround time = 1.566666
Waiting time = 0.733336
CPU Bound The final value of v is 0x820d32a5
Turnaround time = 1.566666
Waiting time = 0.683331
CPU Bound The final value of v is 0xf34ea2a6
Turnaround time = 1.566666
Waiting time = 0.750004
CPU Bound The final value of v is 0x649012a7
Turnaround time = 1.566666
Waiting time = 0.683329
Turnaround time = 1.566666
Waiting time = 0.733337
Waiting time = 0.633332
# _
```

To direct input to this VM, click inside or press Ctrl+G.

Figure 13: Testing priority based

2.6.4. Testing MFQ



```
Turnaround time = 1.500000
Waiting time = 0.666667
CPU Bound The final value of v is 0xfa7a20ae
Turnaround time = 1.500000
Waiting time = 0.716666
CPU Bound The final value of v is 0x6bbb90af
Turnaround time = 1.500000
Waiting time = 0.650004
CPU Bound The final value of v is 0xdcfd00b0
Turnaround time = 1.500000
Waiting time = 0.716664
CPU Bound The final value of v is 0x4e3e70b1
Turnaround time = 1.500000
Waiting time = 0.616668
CPU Bound The final value of v is 0xbf7fe0b2
Turnaround time = 1.516667
Waiting time = 0.699999
CPU Bound The final value of v is 0x30c150b3
Turnaround time = 1.516667
Waiting time = 0.616667
Turnaround time = 1.516667
Waiting time = 0.783332
Waiting time = 0.566667
# tty: ignoring unrecognized escaped scancode 0x5b
*
```

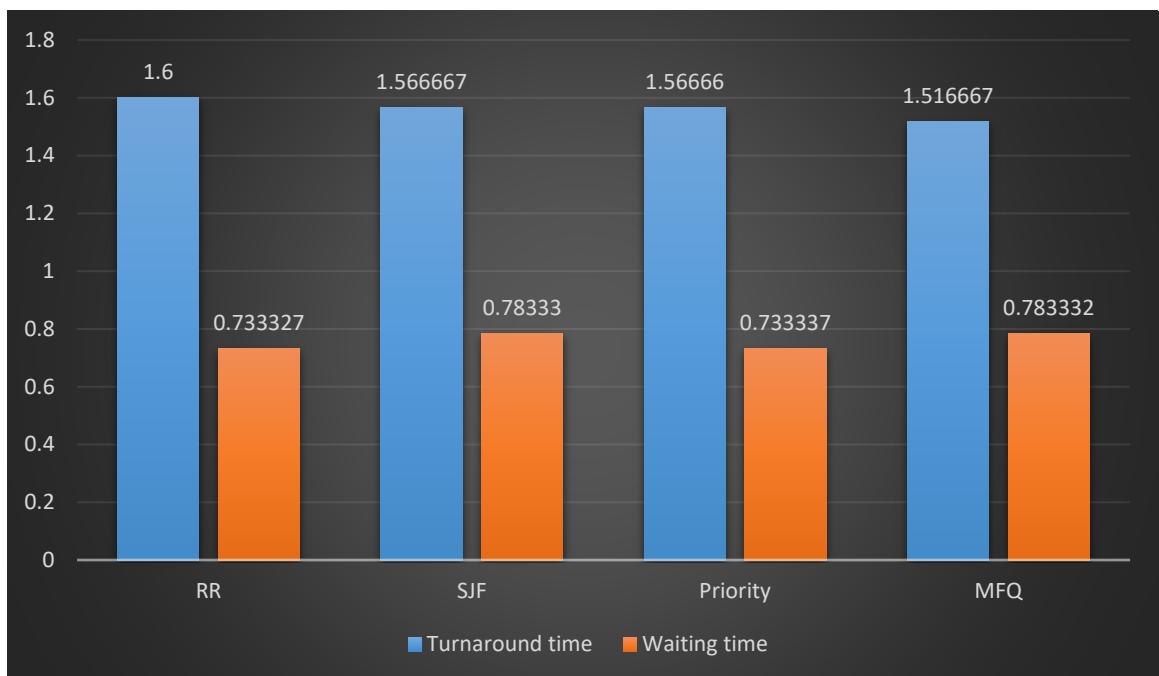
Figure 14: Testing MFQ

2.6.5. Comparing & evaluating testing results:

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

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

- 1- The turnaround time of RR is greater than that of SJF, while the waiting time of the RR is smaller than that of SJF. The reason is that in the SJF algorithm some processes could wait a long time before execution, but the average turnaround time of all processes would be small.
- 2- The turnaround time of SJF is approximately the same as that of priority, which would happen at certain cases, such as: longer processes have lower priority. While the waiting time of the SJF is greater than that of priority, the reason is that in the SJF some processes could wait a long time before execution than the processes wait in priority.
- 3- The turnaround time of priority is greater than that of MFQ, which is obvious from the behavior of these algorithms, as the process in the MFQ enters 3 queues while it enters only 1 queue in priority. On the other hand, the waiting time of the priority is smaller than that of MFQ, the reason is that in the MFQ some processes could wait a long time not assigned to a queue going from a queue to another, but this doesn't happen in priority.



25. MEMORY MANAGEMENT

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

3.1. Memory management in xv6

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

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

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

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

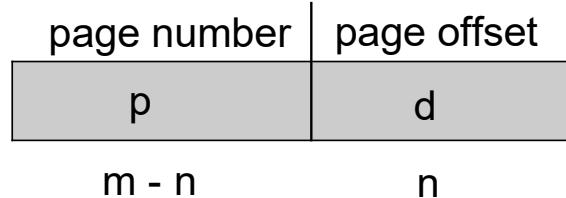


Figure 15: Logical address division

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

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

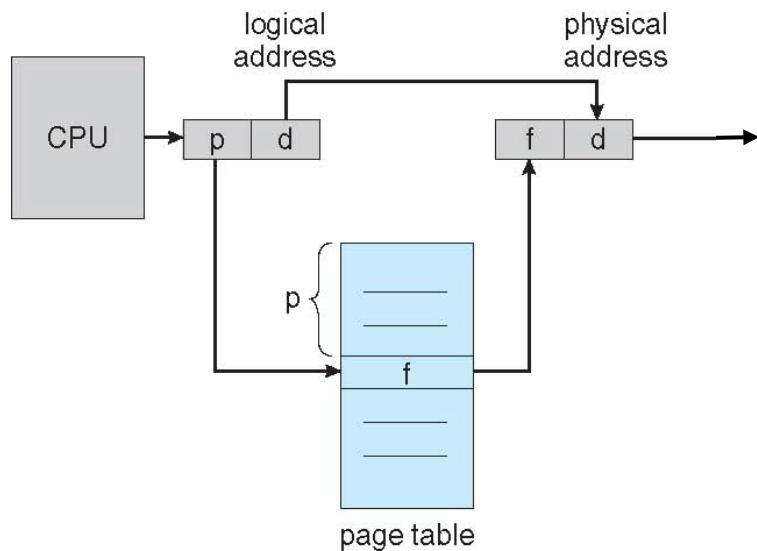


Figure 16: Sending addresses to MMU

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

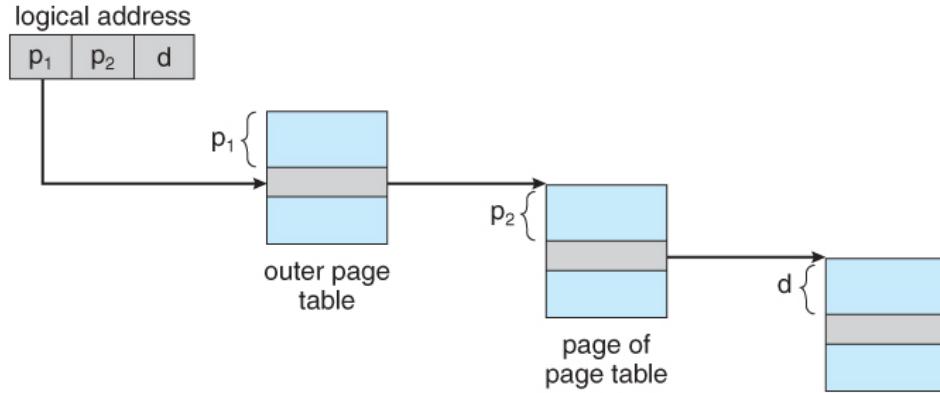


Figure 17: Multi-level paging

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

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

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

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

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

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

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

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

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

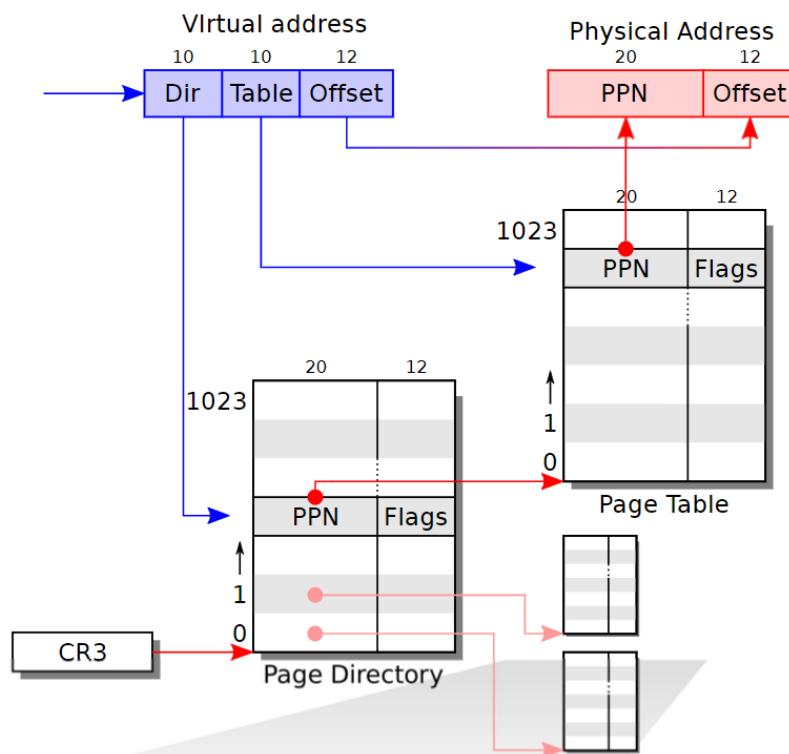


Figure 18: Hierarchical paging xv6

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

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

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

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

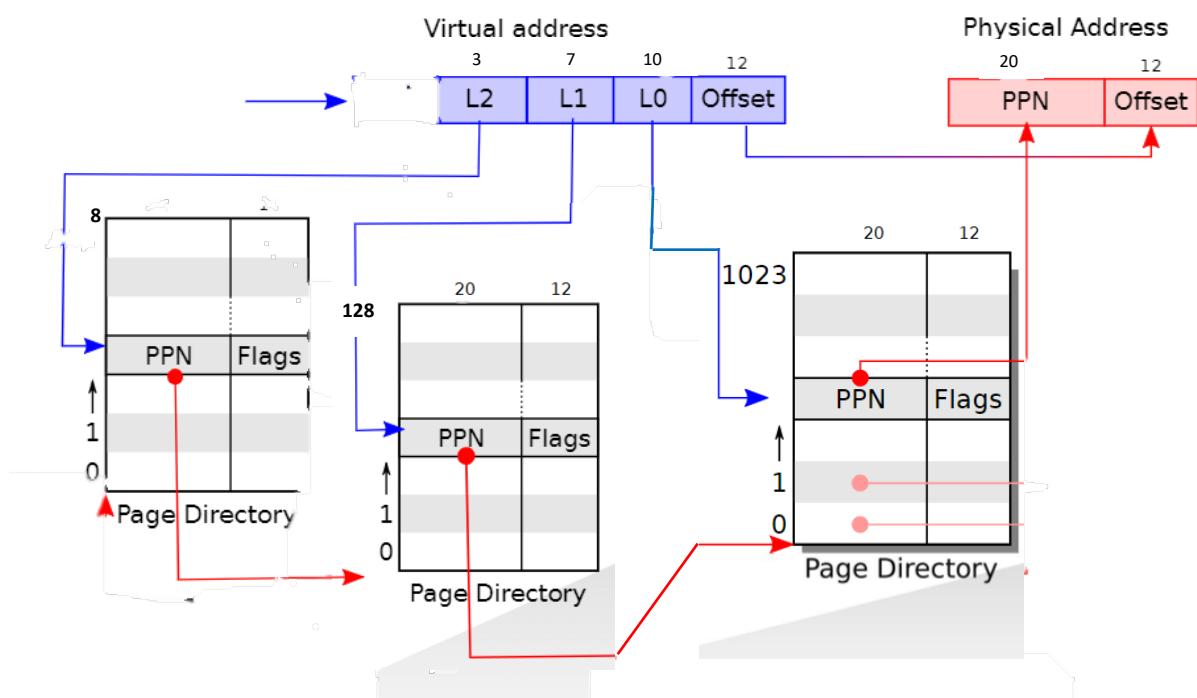


Figure 19: 3-Level Hierarchical paging

- Code Edits:

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

Figure 20 shows all these modifications in “*mmu.h*” header file.

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer (Left):** Shows the project structure under "XV6-PUBLIC". The "mmu.h" file is currently selected.
- Editor Area (Top):** Displays the content of the "mmu.h" file. The code defines constants and macros for memory management, including PDXSHIFT, PTXSHIFT, and PGSHIFT, as well as page table entry flags (PTE_P, PTE_W, PTE_U, PTE_PS).
- Bottom Status Bar:** Shows file information: "master", "2 Δ 0", "Ln 100 Col 9", "Spaces: 2", "UTF-8", "CRLF", "Win32".

Figure 20: 3-Level Paging Code 1

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

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

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

Our modification in the code allows the user to choose between the already implanted 2-level hierarchy or our new 3-level one. This can be achieved by modifying the

THREE_LEVEL_PAGING macro found in “**param.h**” file. Figure 21 and Figure 22 in the next page show the modified **walkpgdir()** function.

```

36 walkpgdir(pde_t *pgdir, const void *va, int alloc)
37 {
38     /* Two Level paging */
39     pde_t *pde;
40     pte_t *pgtab;
41     pde = &pgdir[PDX(va)];
42     if(*pde & PTE_P){
43         pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
44     } else {
45         if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
46             return 0;
47         // Make sure all those PTE_P bits are zero.
48         memset(ptab, 0, PGSIZE);
49         // The permissions here are overly generous, but they can
50         // be further restricted by the permissions in the page table
51         // entries, if necessary.
52         *pde = V2P(ptab) | PTE_P | PTE_W | PTE_U;
53     }
54     return &ptab[PTX(va)];
55 }
56 #elif THREE_LEVEL_PAGING == 1
57 // Three Level Hierarchical Paging with 32-bit Virtual Address
58 // 29..31 -- 3 bits of level-2 index.
59 // 22..28 -- 7 bits of level-1 index.
60 // 12..21 -- 10 bits of level-0 index.
61 // 0..11 -- 12 bits of byte offset within the page.
62 pte_t *pte;
63 pte = &pgdir[((uint)(va) >> 29) & 0x007];
64 if(*pte & PTE_P){
65     pgdir = (pte_t*)P2V(PTE_ADDR(*pte));
66 } else {
67     if(!alloc || (pgdir = (pte_t*)kalloc()) == 0)
68         return 0;
69 }

```

Figure 21: 3-Level Paging Code 2

```

57 #elif THREE_LEVEL_PAGING == 1
58 // Three Level Hierarchical Paging with 32-bit Virtual Address
59
60 // 29..31 -- 3 bits of level-2 index.
61 // 22..28 -- 7 bits of level-1 index.
62 // 12..21 -- 10 bits of level-0 index.
63 // 0..11 -- 12 bits of byte offset within the page.
64
65 pte_t *pte;
66
67 pte = &pgdir[((uint)(va) >> 29) & 0x007];
68 if(*pte & PTE_P){
69     pgdir = (pte_t*)P2V(PTE_ADDR(*pte));
70 } else {
71     if(!alloc || (pgdir = (pte_t*)kalloc()) == 0)
72         return 0;
73     memset(pgdir, 0, PGSIZE);
74     *pte = V2P(pgdir) | PTE_P | PTE_W | PTE_U;
75 }
76
77 pte = &pgdir[((uint)(va) >> 22) & 0x007F];
78 if(*pte & PTE_P){
79     pgdir = (pte_t*)P2V(PTE_ADDR(*pte));
80 } else {
81     if(!alloc || (pgdir = (pte_t*)kalloc()) == 0)
82         return 0;
83     memset(pgdir, 0, PGSIZE);
84     *pte = V2P(pgdir) | PTE_P | PTE_W | PTE_U;
85 }
86
87 return &pgdir[PTX(va)];
88 }
89

```

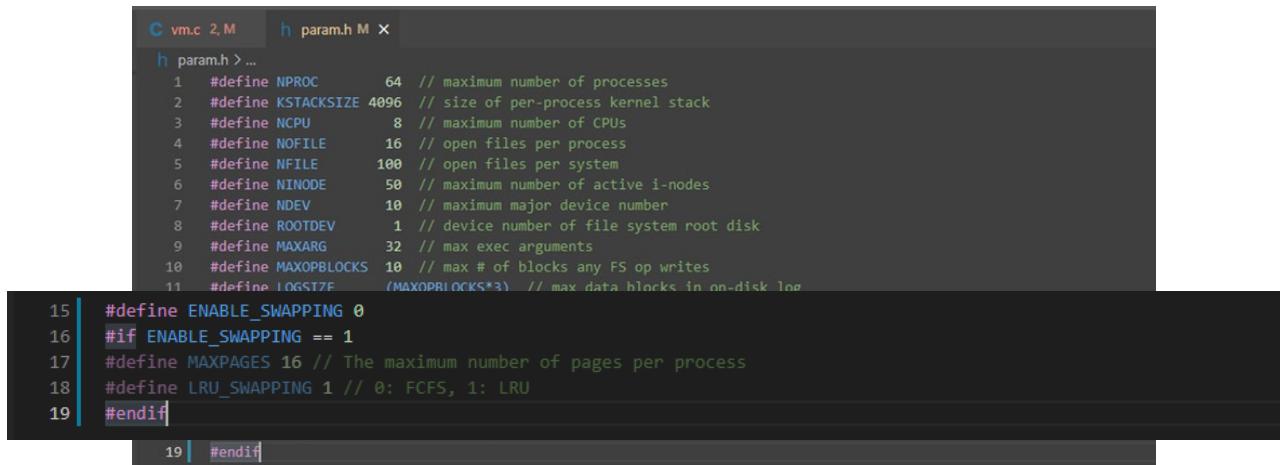
Figure 22: 3-Level Paging Code 3

3.4. Implementing swapping & page replacement algorithms in xv6

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

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

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



```
C vm.c 2, M h param.h M x
h param.h > ...
1 #define NPROC      64 // maximum number of processes
2 #define KSTACKSIZE 4096 // size of per-process kernel stack
3 #define NCPU       8 // maximum number of CPUs
4 #define NOFILE     16 // open files per process
5 #define NFILE      100 // open files per system
6 #define NINODE     50 // maximum number of active i-nodes
7 #define NDEV        10 // maximum major device number
8 #define ROOTDEV    1 // device number of file system root disk
9 #define MAXARG     32 // max exec arguments
10 #define MAXOPBLOCKS 10 // max # of blocks any FS op writes
11 #define LOGSIZEF   (MAXOPBLOCKS*3) // max data blocks in on-disk log
15 #define ENABLE_SWAPPING 0
16 #if ENABLE_SWAPPING == 1
17 #define MAXPAGES 16 // The maximum number of pages per process
18 #define LRU_SWAPPING 1 // 0: FCFS, 1: LRU
19 #endif
19 | #endif
```

Figure 23: Page Swapping Code 1

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

```

C vm.c M C bio.c M C fs.c M h def.h h proc.h M C proc.c M X h mmu.h M
C proc.c > swapToMem(pte_t uint)
546 void swapToMem(pte_t pagetable, uint va)
547 {
548 #if ENABLE_SWAPPING == 1
549 struct proc *process = myproc();
550 if (process != 0)
551     return; // Ensure you have the correct privileges
552
553 struct pagetable_queue_t *p = findpagetable_queue(pagetable);
554 acquire(&p->lock);
555
556 #if LRU_SWAPPING == 1
557 for (int i = 0; i < MAXPAGES; i++)
558     p->pagesage[i]++;
559 for (int i = 0; i < MAXPAGES; i++)
560 {
561     if (p->loadedpages[i] == va)
562     {
563         p->pagesage[i] = 0;
564         release(&p->lock);
565         return;
566     }
567 }
568
569 // The va was not found in page list, try to add it
570 for (int i = 0; i < MAXPAGES; i++)
571 {
572     if (p->loadedpages[i] == 0)
573     {
574         p->loadedpages[i] = va;
575         p->pagesage[i] = 0;
576         release(&p->lock);
577         return;
578     }
579 }
580
581 // No space was found in the page list, choose a victim
582 int oldestindex = 0;
583 for (int i = 1; i < MAXPAGES; i++)
584 {
585     if (p->pagesage[i] > p->pagesage[oldestindex])
586         oldestindex = i;
587 }
588
589 // Page fault, removing victim...
590 release(&p->lock);
591 pte_t *victimpte = walkaddr_suppressed(pagetable, p->loadedpages[oldestindex], 1);
592 if (victimpte != 0)
593     swappage_from_mem(victimpte);
594
595 pte_t *targetpte = walkaddr_suppressed(pagetable, va, 1);
596 if (targetpte != 0 && (*targetpte & PTE_S))
597     swappage_to_mem(victimpte);
598 acquire(&p->lock);
599
600 for (int i = p->queueStart, first = 1; i != p->queueEnd || (first == 1 && p->queueFull); i = (i + 1) % MAXPAGES)
601 {
602     first = 0;
603     if (p->loadedpages[i] == va)
604     {
605         release(&p->lock);
606         return;
607     }
608 }
609
610 // The va was not found in queue
611 if (p->queueFull == 1) // If queue is full
612 {
613     // Page fault happened, removing victim...
614     release(&p->lock);
615     pte_t *victimpte = walkaddr_suppressed(pagetable, p->loadedpages[p->queueStart], 1);
616     if (victimpte != 0)
617         swappage_from_mem(victimpte);
618
619     pte_t *targetpte = walkaddr_suppressed(pagetable, va, 1);
620     if (targetpte != 0 && (*targetpte & PTE_S))
621         swappage_to_mem(victimpte);
622     acquire(&p->lock);
623     p->queueStart = (p->queueStart + 1) % MAXPAGES;
624     p->loadedpages[p->queueEnd] = va;
625     p->queueEnd = (p->queueEnd + 1) % MAXPAGES;
626 }
627 else // add page to queue
628 {
629     p->loadedpages[p->queueEnd] = va;
630     p->queueEnd = (p->queueEnd + 1) % MAXPAGES;
631     if (p->queueStart == p->queueEnd)
632     {
633         p->queueFull = 1;
634     }
635 }
636 #endif // LRU or FCFS
637 release(&p->lock);
638 #endif // ENABLE_SWAPPING
639

```

Figure 24: Page Swapping Code 2

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

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

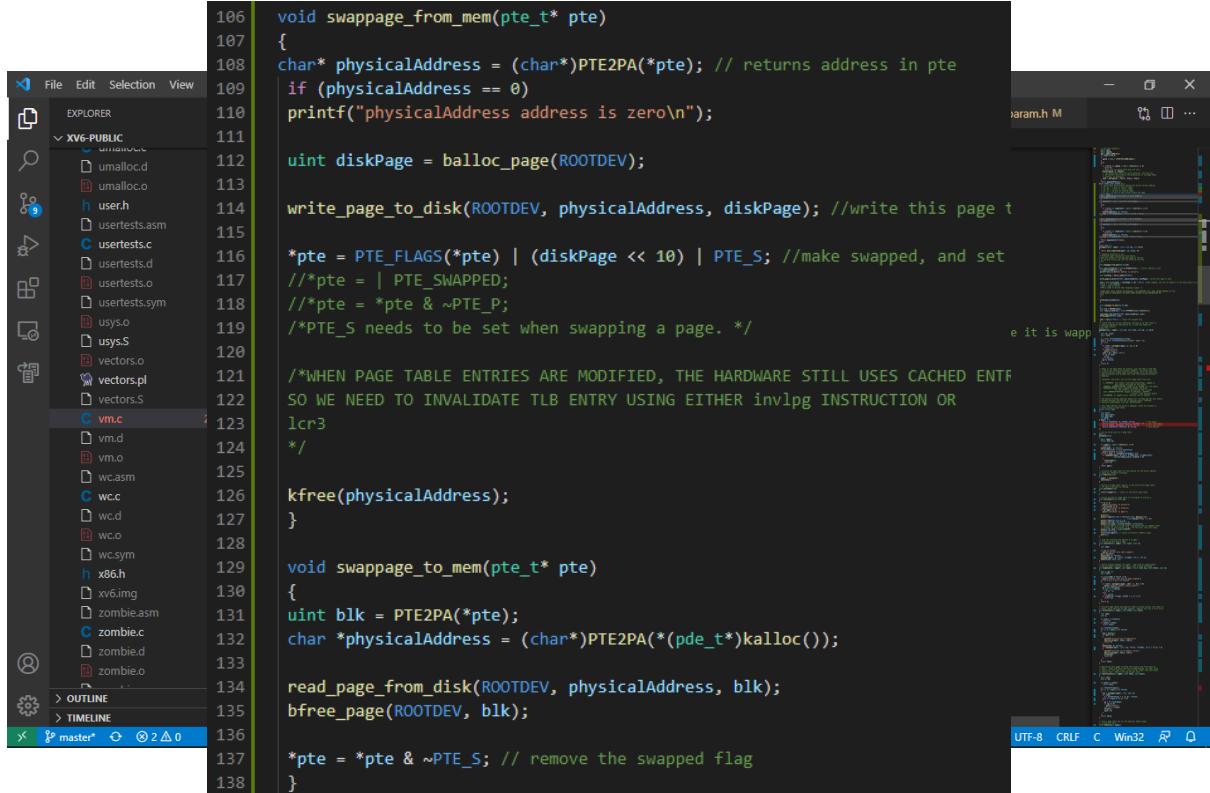
```

C vm.c 2 M X C bio.c M C fs.c M h def.h h proc.h M C proc.h M h mmu.h M
C vm.c > walkpgdir(pde_t *, const void *, int)
34
35 pte_t *walk_suppressed(pte_t *pagetable, uint va, int alloc, uint suppressSwapping)
36 {
37     if (suppressSwapping == 0)
38         swapToMem(*pagetable, (va >> 12) << 12);
39 #if THREE_LEVEL_PAGING == 0
40     /* Two Level paging */
41     pde_t *pde;
42     pte_t *pgtab;
43     pde = &pgd[PDX(va)];
44     if (*pde & PTE_P)
45     {
46         pgtab = (pte_t *)P2V(PTE_ADDR(*pde));
47     }
48     else
49     {
50         if (!alloc || (pgtab = (pte_t *)kalloc()) == 0)
51             return 0;
52         // Make sure all those PTE_P bits are zero.
53         memset(pgtab, 0, PGSIZE);
54         // The permissions here are overly generous, but they can
55         // be further restricted by the permissions in the page table
56         // entries, if necessary.
57         *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
58     }
59     return &pgtab[PTX(va)];
60 #elif THREE_LEVEL_PAGING == 1
61     // Three Level Hierarchical Paging with 32-bit Virtual Address
62     // 29..31 -- 3 bits of level-2 index.
63     // 22..28 -- 7 bits of level-1 index.
64     // 12..21 -- 10 bits of level-0 index.
65     // 0..11 -- 12 bits of byte offset within the page.
66     pte_t *pte;
67     pte = &pagetable[((uint)(va) >> 29) & 0x0007];
68     if (*pte & PTE_P)
69     {
70         pagetable = (pte_t *)P2V(PTE_ADDR(*pte)); //
71     }
72     else
73     {
74         if (!alloc || (pagetable = (pte_t *)kalloc()) == 0)
75             return 0;
76         memset(pagetable, 0, PGSIZE);
77         *pte = V2P(*pagetable) | PTE_P | PTE_W | PTE_U;
78     }
79
80     pte = &pagetable[((uint)(va) >> 22) & 0x000F];
81     if (*pte & PTE_P)
82     {
83         pagetable = (pte_t *)P2V(PTE_ADDR(*pte)); //
84     }
85     else
86     {
87         if (!alloc || (pagetable = (pte_t *)kalloc()) == 0)
88             return 0;
89         memset(pagetable, 0, PGSIZE);
90         *pte = V2P(pagetable) | PTE_P | PTE_W | PTE_U;
91     }
92     return &pagetable[PTX(va)];
93 #endif
94
95 static pte_t *
96 walkpgdir(pde_t *pgdir, const void *va, int alloc)
97 {
98     return walk_suppressed(pgdir, va, alloc, 0);
99 }

```

Figure 25: Page Swapping Code 3

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



```

106 void swappage_from_mem(pte_t* pte)
107 {
108     char* physicalAddress = (char*)PTE2PA(*pte); // returns address in pte
109     if (physicalAddress == 0)
110         printf("physicalAddress address is zero\n");
111
112     uint diskPage = balloc_page(ROOTDEV);
113
114     write_page_to_disk(ROOTDEV, physicalAddress, diskPage); //write this page to disk
115
116     *pte = PTE_FLAGS(*pte) | (diskPage << 10) | PTE_S; //make swapped, and set
117     ///*pte = | PTE_SWAPPED;
118     ///*pte = *pte & ~PTE_P;
119     /*PTE_S needs to be set when swapping a page. */
120
121     /*WHEN PAGE TABLE ENTRIES ARE MODIFIED, THE HARDWARE STILL USES CACHED ENTRIES
122     SO WE NEED TO INVALIDATE TLB ENTRY USING EITHER invlpg INSTRUCTION OR
123     lcr3
124
125     */
126
127     kfree(physicalAddress);
128
129 void swappage_to_mem(pte_t* pte)
130 {
131     uint blk = PTE2PA(*pte);
132     char *physicalAddress = (char*)PTE2PA(*(&pde_t*)kalloc()));
133
134     read_page_from_disk(ROOTDEV, physicalAddress, blk);
135     bfree_page(ROOTDEV, blk);
136
137     *pte = *pte & ~PTE_S; // remove the swapped flag
138 }

```

Figure 26: Page Swapping Code 4

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

- Maps the victim page to its physical address on the memory by calling **PTE2PA**
- Allocates eight consecutive blocks on the disk (at which the victim page will be stored) by calling **balloc_page()** located in “**fs.c**”

```

700 uint balloc_page(uint dev)
701 {
702     uint allocatedBlocks[4096];
703     int indexNCB = -1; //pointer for above array, keeps track till where it is
704     // bp = 0;
705     for (int i = 0; i < 8; i++)
706     {
707         indexNCB++;
708         if (indexNCB >= 4096)
709             panic("balloc_page: no page found after 4096 blocks allocated");
710
711         begin_op();
712         allocatedBlocks[indexNCB] = balloc(dev);
713         end_op();
714
715         if (i > 0 && indexNCB > 0)
716         {
717             if ((allocatedBlocks[indexNCB] - allocatedBlocks[indexNCB - 1]) != 1) // consecutive
718             {
719                 i = 0; //start allocating blocks again
720             }
721         }
722         for (int i = 0; i <= indexNCB - 8; i++)
723         {
724             bfree(ROOTDEV, allocatedBlocks[i]); //free unnecessarily allocated blocks
725         }
726         //numallocblocks += 1; //*****
727         uint res = allocatedBlocks[indexNCB - 7];
728         //kfree(allocatedBlocks);
729
730         return res; //return last 8 blocks (address of 1st block among them)
731     }
732 }

```

Figure 27: Page Swapping Code 5

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

```

150 void write_page_to_disk(uint dev, char *pg, uint blk)
151 {
152     //*****XV7*****
153     struct buf *buffer;
154     int blockno = 0;
155     int ithPartOfPage = 0; //which part of page (out of 8) is to be wr
156     for (int i = 0; i < 8; i++)
157     {
158         // begin_op(); //for atomicity , the block must be written to th
159         ithPartOfPage = i * 512;
160         blockno = blk + i;
161         buffer = bget(ROOTDEV, blockno);
162         /*
163             Writing physical page to disk by dividing it into 8 pieces (4096 by
164             As one page requires 8 disk blocks
165         */
166         memmove(buffer->data, pg + ithPartOfPage, 512); // write 512 byte
167         bwrite(buffer);
168         brelse(buffer); //release lock
169         // end_op();
170     }
171
172     /* Read 4096 bytes from the eight consecutive
173      * starting at blk into pg.
174      */
175     void read_page_from_disk(uint dev, char *pg, uint blk)
176     {
177         //*****XV7*****
178         struct buf *buffer;
179         int blockno = 0;
180         int ithPartOfPage = 0;
181     }

```

Figure 28: Page Swapping Code 6

- Changes the PTE (page-table entry) to point to the newly created block.
- Flags this PTE as swapped out of ram

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

- 1- Extracts the page number from the PTE by calling **PTE2PA**
- 2- Allocates a new page in memory
- 3- Calls **read_page_from_disk()**, located in “**bio.c**” to copy the page data from disk to the newly allocated page in memory

```

C vm.c 2 M C bio.c M C fs.c M h defsh h proch M C proc.c M h mmuh M h param.h M
C bio.c > write_page_to_disk(uint char * uint)
168 | brelse(buffer); //release lock
169 | // end_op();
170 |
171 | }
172 /* Read 4096 bytes from the eight consecutive
173 * starting at blk into pg.
174 */
175 void read_page_from_disk(uint dev, char *pg, uint blk)
176 {
177 //*****
178 struct buf *buffer;
179 int blockno = 0;
180 int ithPartOfPage = 0;
181 for (int i = 0; i < 8; i++)
182 {
183 ithPartOfPage = i * 512;
184 blockno = blk + i;
185 buffer = bread(ROOTDEV, blockno); //if present in buffer, returns from buffer else from disk
186 memmove(pg + ithPartOfPage, buffer->data, 512); //write to pg from buffer
187 brelse(buffer); //release lock
188 }
189 }

```

Ln 171, Col 2 Spaces: 2 UTF-8 CRLF

Figure 29: Page Swapping Code 7

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

```

C vm.c 2 M C bio.c M C fs.c M X h defsh h proch M C proc.c M h mmuh M h param.h M
C fs.c > alloc_page(uint)
716 | {
717 | if ((allocatedBlocks[indexNCB] - allocatedBlocks[indexNCB - 1]) != 1) //this allocated block is non consecutive
718 | {
719 | i = 0; //start allocating blocks again
720 | }
721 |
722 | for (int i = 0; i <= indexNCB - 8; i++)
723 | {
724 | bfree(ROOTDEV, allocatedBlocks[i]); //free unnecessarily allocated blocks
725 | }
726 //numallocblocks += 1; //*****
727 uint res = allocatedBlocks[indexNCB - 7];
728 //kfree(allocatedBlocks);
729 }

730 void bfree_page(int dev, uint b)
731 {
732 // cprintf("In Bfree Page\n");
733 for (uint i = 0; i < 8; i++)
734 {
735 bfree(ROOTDEV, b + i);
736 }
737 }

738 // the writel function in fs.c
739 // Write data to inode.
740 // Caller must hold ip->lock.
741 // If user_src==1, then src is a user virtual address.
742 // otherwise, src is a kernel address.
743 // Returns the number of bytes successfully written.
744 // If the return value is less than the requested n,
745

```

Ln 704, Col 13 Spaces: 2 UTF-8 CRLF

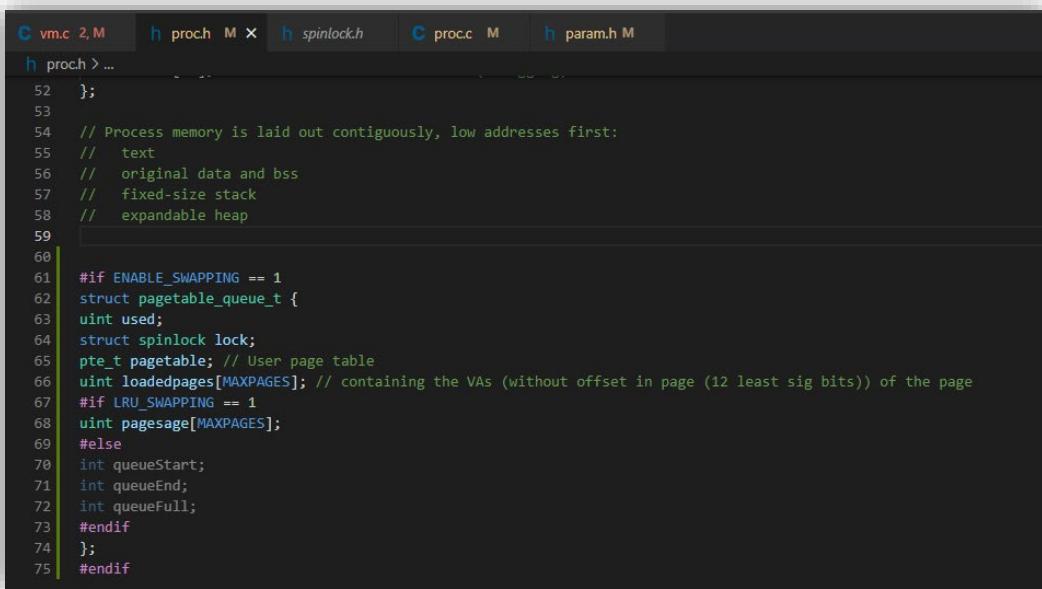
Figure 30: Page Swapping Code 8

- 5- Flags the PTE as not swapped out of ram

3.4.1. First In First Out (FIFO) algorithm

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

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



A screenshot of a terminal window displaying C code. The terminal tabs include "vm.c", "proc.h", "spinlock.h", "proc.c", and "param.h". The code itself is as follows:

```
52  };
53
54 // Process memory is laid out contiguously, low addresses first:
55 // text
56 // original data and bss
57 // fixed-size stack
58 // expandable heap
59
60
61 #if ENABLE_SWAPPING == 1
62 struct pagetable_queue_t {
63     uint used;
64     struct spinlock lock;
65     pte_t pagetable; // User page table
66     uint loadedpages[MAXPAGES]; // containing the VAs (without offset in page (12 least sig bits)) of the page
67 #if LRU_SWAPPING == 1
68     uint pagesage[MAXPAGES];
69 #else
70     int queueStart;
71     int queueEnd;
72     int queueFull;
73 #endif
74 };
75 #endif
```

Figure 31: FIFO Replacement Algorithm Code

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

3.4.2. Last Recently Used (LRU) algorithm

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

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

```
60
61 #if ENABLE_SWAPPING == 1
62 struct pagetable_queue_t {
63     uint used;
64     struct spinlock lock;
65     pte_t pagetable; // User page table
66 }
67 #if LRU_SWAPPING == 1
68     uint pagesage[MAXPAGES];
69 #else
70     int queueEnd;
71     int queueFull;
72 #endif
73 };
74 #endif
75 
```

Figure 32: LRU Algorithm Code

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

3.5. Performance results for hierachal paging and replacement algorithms:

Firstly, we build the system image by using WSL (windows subsystem Linux), which allows us to use Linux terminal on our operating system to use its commands to build XV6.

Before we run the system to examine the FIFO and LRU algorithms we predict that the LRU must have higher hit ratio than FIFO.

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

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

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

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

26. FILE SYSTEM MANAGEMENT

4.1. Requirement Statement

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

4.2. File System in MINIX 3

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

1- Boot Block:

File system starts with a boot block of size 1 KB (regardless of the OS block size) including an executable code which runs on machine start to only boot the operating system; therefore, once the operating system is booted, it is not accessed anymore.

Moreover, all disks managed by the file system has a boot block but the hardware only attempts booting from the disk with a “magic number” placed at in the boot block.

2- Superblock:

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

| | |
|--|--|
| Present on disk and in memory | Number of i-nodes |
| | (unused) |
| | Number of i-node bitmap blocks |
| | Number of zone bitmap blocks |
| | First data zone |
| | \log_2 (block/zone) |
| | Padding |
| | Maximum file size |
| | Number of zones |
| | Magic number |
| | padding |
| | Block size (bytes) |
| | FS sub-version |
| | Pointer to i-node for root of mounted file system |
| | Pointer to i-node mounted upon |
| | i-nodes/block |
| | Device number |
| Present in memory but not on disk | Read-only flag |
| | Native or byte-swapped flag |
| | FS version |
| | Direct zones/i-node |
| | Indirect zones/indirect block |
| | First free bit in i-node bitmap |
| | First free bit in zone bitmap |

Figure 33: MINIX Superblock

3- Bitmaps:

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

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

4- Zones vs Blocks:

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

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

5- I-nodes:

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

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

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

If a system has a 4KB block size:

- a double indirect block pointing to 1024 single indirect blocks ->
$$1024 * 1024 * 4 \text{ KB} = 4 \text{ GB of storage}$$

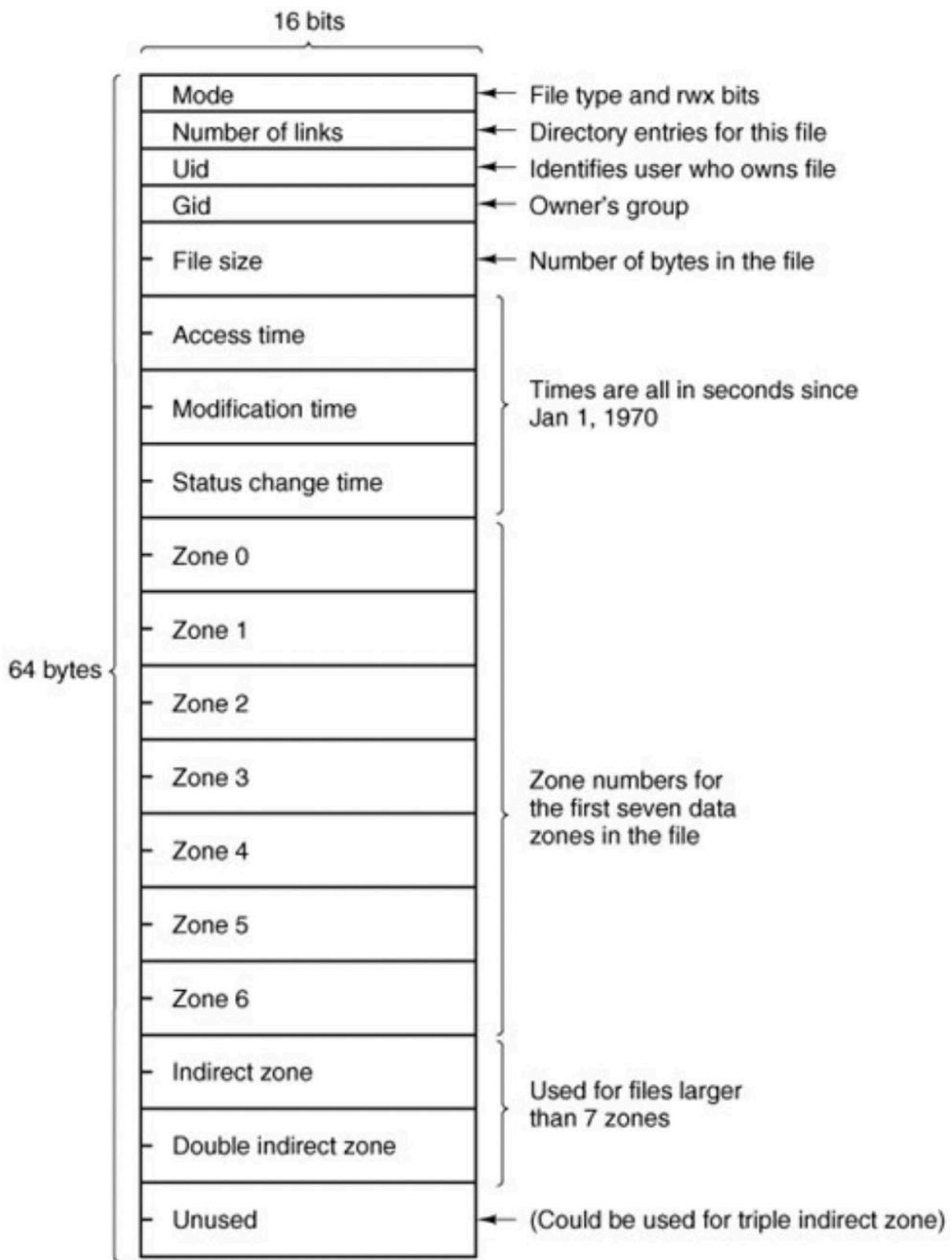


Figure 34: MINIX i-node

4.3. How MINIX 3 manages empty space

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

0 -> free i-node / zone

1 -> occupied i-node / zone

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

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

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

4.4. How we edited the free space management

4.4.1. Extents

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

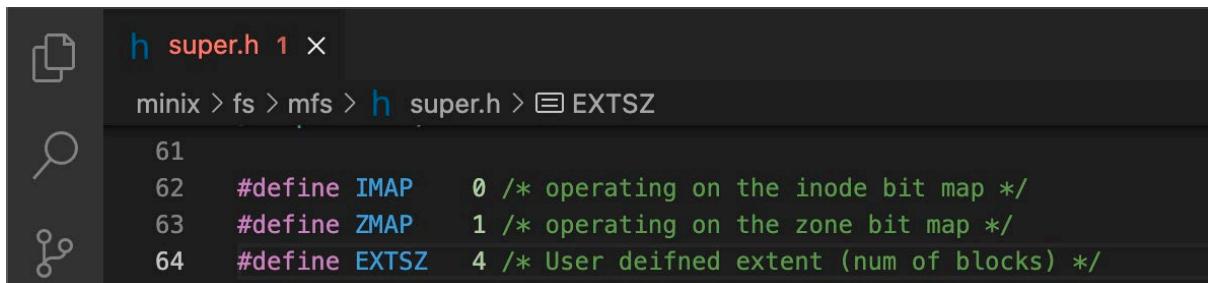
“*alloc_bit*” function is responsible for allocating blocks needed by a file by refereing to each block of size 1 KB as 1 bit in the bitmap. Allocating a block is done by setting its corresponding bit in the bit map to 1. This allocation in the bitmap is done by dividing the bitmap to blocks, starting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

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

4.4.2. Code Edits

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

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



```
h super.h 1 ×
minix > fs > mfs > h super.h > EXT SZ
61
62 #define IMAP    0 /* operating on the inode bit map */
63 #define ZMAP    1 /* operating on the zone bit map */
64 #define EXT SZ  4 /* User deifned extent (num of blocks) */
```

Figure 35: FS Code 1

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

- **alloc_bit:** [Figure 23: FS Code 2](#)

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

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

- **free_bit:** [Figure 22: FS Code 3](#)

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

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

```

C super.c 2 ●
minix > fs > mfs > C super.c > ...
82
83
84
85
86
87
88
89
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
165
166
167
168
169
170
171
172
173
174
175
176

```

/* Developed File System using user defined Extents*/
int cnt = 0;
for (i = 0; (k & (1 << i)) != 0 && cnt < EXTSZ && i < 16; ++i) {
 if((k & (1 << i)) == 0){
 cnt++;
 }
}
if(cnt != EXTSZ) continue;
 + (wptr - &b_bitmap(bp)[0]) * FS_BITCHUNK_BITS
 + i;
/* Don't allocate bits beyond the end of the map. */
if (b >= map_bits) break;

/* Developed File System using user defined Extents*/
for(int j = i; j > i-EXTSZ; j--){
 k |= 1 << j;
}

MAPKU1K1Y(bp);
put_block(bp);
if(map == ZMAP) {
 used_zones++;
 lmfs_change_blockusage(1);
}
return(b);
}
put_block(bp);
if (++block >= (unsigned int) bit_blocks) /* last block, wrap around */
block = 0;
word = 0;
} while (--bcount > 0);
return(NO_BIT); /* no bit could be allocated */
}

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

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

/* Developed File System using user defined Extents*/
for(int i = bit; i > bit-EXTSZ; i--){
 mask |= 1 << i;
}

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

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

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

put_block(bp);

if(map == ZMAP) {
 used_zones--;
 lmfs_change_blockusage(-1);
}

4.4.3. Performance testing

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

```

ptyfs is not in use
postinstall checks passed: fontconfig Mtd Mtree wscons x11 xkb varrwho tcpdumpc
hroot catpages obsolete ptyfsoldnodes
postinstall checks failed: bluetooth ddbonpanic defaults dhcpcd envsys gid gpio
hosts iscsi makedev named pam periodic pf pwd_Mkdb rc ssh uid atf
To fix, run:
    /bin/sh /usr/src/usr.sbin/postinstall/postinstall -s '/usr/src' -d / fix blu
ethooth ddbonpanic defaults dhcpcd envsys gid gpio hosts iscsi makedev named pam
periodic pf pwd_Mkdb rc ssh uid atf
Note that this may overwrite local changes.
=====
do-obsolete ==> .
install-obsolete-lists ==> etc
    install /var/db/obsolete/minix
install-etc-release ==> etc
    create etc/etc-release
hostname: not found
    install etc/release
do-hdboot ==> releasetools
git: not found
rm /dev/c0d0p0s0:/boot/Minix/3.2.1r2
Done.
Build started at: Sat Jan 1 11:41:55 GMT 2022
Build finished at: Sat Jan 1 11:53:15 GMT 2022
#

```

[Figure 36: FS Building successfully](#)

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

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

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

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

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

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

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

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

- The following table shows number of files of size **24 KB** that can be saved on a disk of size **16 MB** ($16 \times 1024 = 16,384$ KB) for different number of blocks per extent.

This case has an ideal disk utilization as files of this size have no internal fragmentation for all tested number of blocks per extent.

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