Operating System Project CSE335

Dr: Gamal A. Ebrahim
T.A: Sally Shaker

Names	*IDs*
Ahmed Mohamed El Sayed Tabbash	20P1076
Yaman Abdelhamid Elewa	2002078
Paula Emil Alexan Aziz	2200750
Ahmed Hassan Nofull	22P0255
Patrick hany adel	2200557

Table of Contents

Requirement 1

Scheduling implementation

In this Requirement, We Will modify MINIX 3 to include the following scheduling algorithms:

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

1-Round Robin:

- Minix 3 ,By Default, Has Round Robin Scheduling implemented in it. We modified the code So that, All Processes to be assigned to only one Queue.
- In the Round-Robin Algorithm, Processes are executed in sequential order and each process runs for a certain amount of time (determined by the operating system developer). After the amount of time has passed the process is then interrupted by the process next in line to be executed for the same amount of time, even if it hasn't finished execution.
- In schedule.c, We edited the **do_noquantam()**"This Function is called when a process finishes its quantum. Concerning the the part of lowering priority of a process when consumes its entire quantum, We disabled it.

```
99      }
100
101     rmp = &schedproc[proc_nr_n];
102         /* disable feedback part in the function  for lowering priority*/
103         if(rmp->priority < MIN_USER_Q){
104         printf("Process consumed its quantam and the priority = %d\n",rmp->proirity);
105         //Deleted the line responsible for lowering priority
106         }
107
108         if((rv = schedule_process_local(rmp)) != OK){
109          return rv;
110          }
111
112         return OK;
113 }
```

- For making the Algorithm Running on a single queue, edited the Queues to run only on a single queue. So, In **config.h** we changed the user maximum queue to be 15, Because by default minix 3 uses 16 queues and the-lowest priority queue- 16th queue represents the idle, Because **MIN_USER_Q** is the **NR_SCHED_QUEUES -1** will be 16 – 1 equals 15, making all processes running only in queue 15 as **MIN_USER_Q** equals **MAX_USER_Q**.

```
 */
#define NR_SCHED_QUEUES   16    /* MUST equal minimum priority + 1 */
#define TASK_Q         0    /* highest, used for kernel tasks */
#define MAX_USER_Q         15    /* highest priority for user processes */
#define USER_Q        ((MIN_USER_Q - MAX_USER_Q) / 2 + MAX_USER_Q) /* default
                       (should correspond to nice 0) */
#define MIN_USER_Q    (NR_SCHED_QUEUES - 1) /* minimum priority for user
                           processes */
/* default scheduling quanta */
#define USER_QUANTUM 200
```

2-Shortest Job First (SJF)

- In Shortest Job first algorithm, processes are scheduled to be executed according to the burst time of each process.
- In **schedule.c** we edited the **do_noquantum()** function so we can choose the process with the lowest minimum burst time among other processes.

```
105       rmp = &schedproc[proc_nr_n];
106            if(rmp->priority >= MAX_USER_Q && rmp->priority <= MIN_USER_Q)
107            {
108              int MinBurstTime = 0;
109              for(int i = 0 ; i <NR_PROCS ; i++)
110              {
111                temp = &schedproc[i];
112                if( temp->flags == IN_USE ){
113                 if( temp->BRUST_TIME < MinBurstTime || i == 0)
114                       {
115                          MinBurstTime = temp->BRUST_TIME;
116                          temp = &schedproc[i];
117
118                       }
119                }
120              }
121
122
123              rmp = temp;
124
125            }
126           else if (rmp->priority < MIN_USER_Q) {
127            rmp->priority += 1; /* lower priority */
128         }
129            if((rv = schedule_process_local(rmp)) != OK){
130            return rv;
131            }
132
```

- We added **BRUST_TIME** variable in **schedproc** struct in **schedproc.h.**

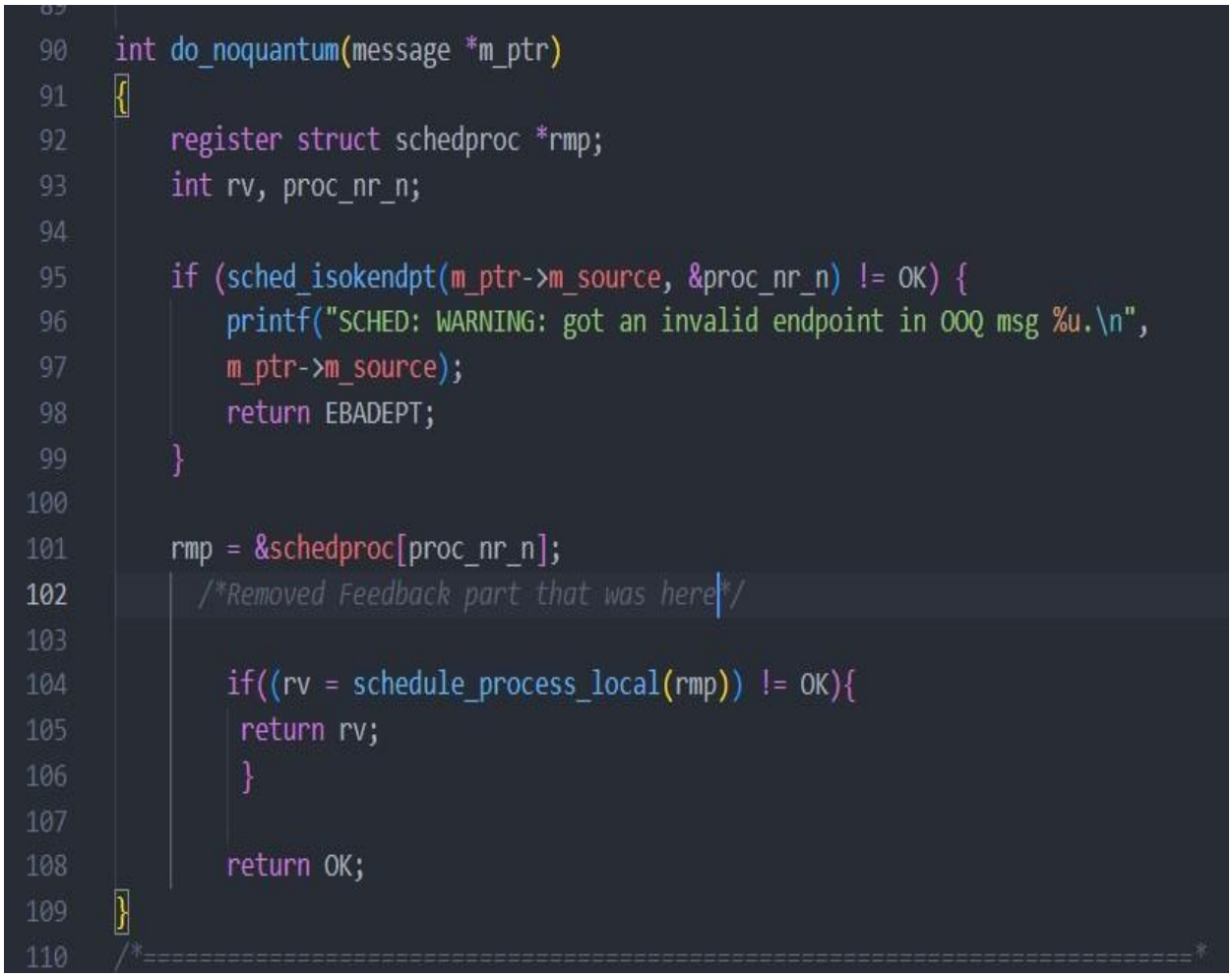
```
EXTERN struct schedproc {
    endpoint_t endpoint;    /* process endpoint id */
    endpoint_t parent;  /* parent endpoint id */
    unsigned flags;     /* flag bits */

    /* User space scheduling */
    unsigned max_priority;  /* this process' highest allowed priority */
    unsigned priority;      /* the process' current priority */
    unsigned time_slice;        /* this process's time slice */
    unsigned cpu;       /* what CPU is the process running on */

        unsigned BRUST_TIME;/* Brust time of process */


    bitchunk_t cpu_mask[BITMAP_CHUNKS(CONFIG_MAX_CPUS)]; /* what CPUs is hte
                            process allowed
                            to run on */
} schedproc[NR_PROCS];

/* Flag values */
#define IN_USE      0x00001 /* set when 'schedproc' slot in use */

```

- We made the user processes work on queues 16 to 18,
  So we edited **NR_SCHED_QUEUES** to be set to 19, So **MIN_USER_Q** is equal to 18 "**NR_SCHED_QUEUES -1"** , and the process with highest priority to 16.

```
#define NR_SCHED_QUEUES   19    /* MUST equal minimum priority + 1 */
#define TASK_Q          0    /* highest, used for kernel tasks */
#define MAX_USER_Q          16    /* highest priority for user processes */
#define USER_Q        ((MIN_USER_Q - MAX_USER_Q) / 2 + MAX_USER_Q) /* default
                        (should correspond to nice 0) */
#define MIN_USER_Q    (NR_SCHED_QUEUES - 1) /* minimum priority for user
                            processes */
```

3-Priority based:

- In this algorithm, each process is given a priority number. Priority scheduling in operating systems is a method of scheduling processes based on their given priority. Where the higher priority processes are executed before the lower priority processes.
- To do this we disabled the feedback part in **schedule.c** in **do_noquantum()** that lowers the priority whenever the process used consecutive quantums.
- By Doing this processes with higher priority will executed first at the queue and the job with the minimum priority will be executed at last. Each process has its own priority in the Process Control Block (PCB) .



```
90  int do_noquantum(message *m_ptr)
91  {
92      register struct schedproc *rmp;
93      int rv, proc_nr_n;
94
95      if (sched_isokendpt(m_ptr->m_source, &proc_nr_n) != OK) {
96          printf("SCHED: WARNING: got an invalid endpoint in OOQ msg %u.\n",
97          m_ptr->m_source);
98          return EBADEPT;
99      }
100
101     rmp = &schedproc[proc_nr_n];
102       /*Removed Feedback part that was here*/
103
104         if((rv = schedule_process_local(rmp)) != OK){
105          return rv;
106          }
107
108         return OK;
109 }
110 /*===========================================================================*
```

4-Multi-Level Feedback Queue:

- The Multi-Level feedback queue is a scheduling algorithm that divides processes into multiple ready queues based on processor demand, prefers processes with short CPU burst, And prefers processes that have high I/O bursts. (The I/O bound processes sleep in the waiting queue to give the other processes some CPU time).
- We added 3 queues 16-18 and 19 which represents the IDLE, Where processes Work on them. When the process consumes the 5 quantum at queue 16 it goes to queue 17(priority decreased) and if it consumed 15 quantum it goes to queue 18(priority decreased) if it consumed 35 quantum it goes back to queue 16.
- To Make the processes work on queues 16-18 we edited in the **config.h** file, We made the total number of schedules equals 19 (**NR_SCHED_QUEUES**) and the queue with highest priority equals to 16(**NR_SCHED_QUEUES-1**), then minimum priority for user processes will be 18 (**MIN_USER_Q = NR_USER_QUEUES -1**)

```
71  #define NR_SCHED_QUEUES   19    /* MUST equal minimum priority + 1 */
72  #define TASK_Q         0    /* highest, used for kernel tasks */
73  #define MAX_USER_Q        16    /* highest priority for user processes */
74  #define USER_Q        ((MIN_USER_Q - MAX_USER_Q) / 2 + MAX_USER_Q) /* default
75                          (should correspond to nice 0) */
76  #define MIN_USER_Q    (NR_SCHED_QUEUES - 1) /* minimum priority for user
```

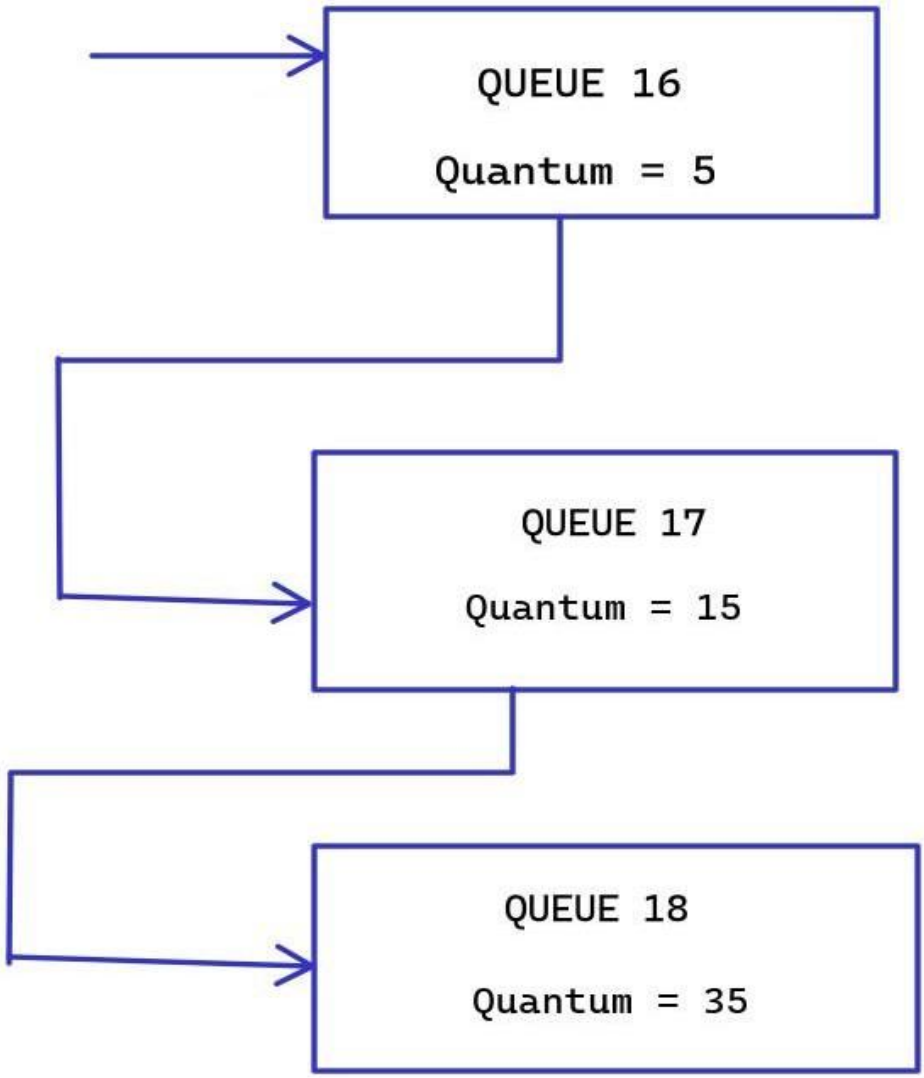
- In **schedule.**c we modified the **do_noquantum()** function to do the following: Processes will work on queues 16,17,18, when the process consume the quantum for each queue we lower their priority by increasing it by 1. when process in queue 18 is terminated we set the priority to 16.

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

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

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

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

- For the quantum spent in each queue by the process entered, we made a variable named **quantum** in **schedproc.h**.

```c
EXTERN struct schedproc {
    endpoint_t endpoint;    /* process endpoint id */
    endpoint_t parent;  /* parent endpoint id */
    unsigned flags;     /* flag bits */

    /* User space scheduling */
    unsigned max_priority;  /* this process' highest allowed priority */
    unsigned priority;      /* the process' current priority */
    unsigned time_slice;        /* this process's time slice */
    unsigned cpu;       /* what CPU is the process running on */
    unsigned quantum;
    bitchunk_t cpu_mask[BITMAP_CHUNKS(CONFIG_MAX_CPUS)]; /* what CPUs is hte
                                process allowed
                                to run on */
} schedproc[NR_PROCS];

/* Flag values */
#define IN_USE      0x00001 /* set when 'schedproc' slot in use */
```

- **A design Illustrating MLFQ**



- ***The Directories of files we used:*** /usr/src/servers/sched/schedule.c
  /usr/src/include/minix/config.h
  /usr/src/servers/sched/schedproc.h
  .

5-Tests and Results:

We created a Directory Named Algorithms in mnt file where for each Algorithm has its own Directory in it there's the test files and the modified versions of the used files.

/mnt/Algorithms/SJF

/mnt/Algorithms/MFQ

/mnt/Algorithms/RoundRobin

/mnt/Algorithms/Priority

In order to run tests for Round Robin , Priority, SJF:

We must first make , build and clean the changed directories then write the following commands:

cc longrun0.c -o longrun0 cc mytest.c -o mytest

./mytest

In order to run tests for MFQ:

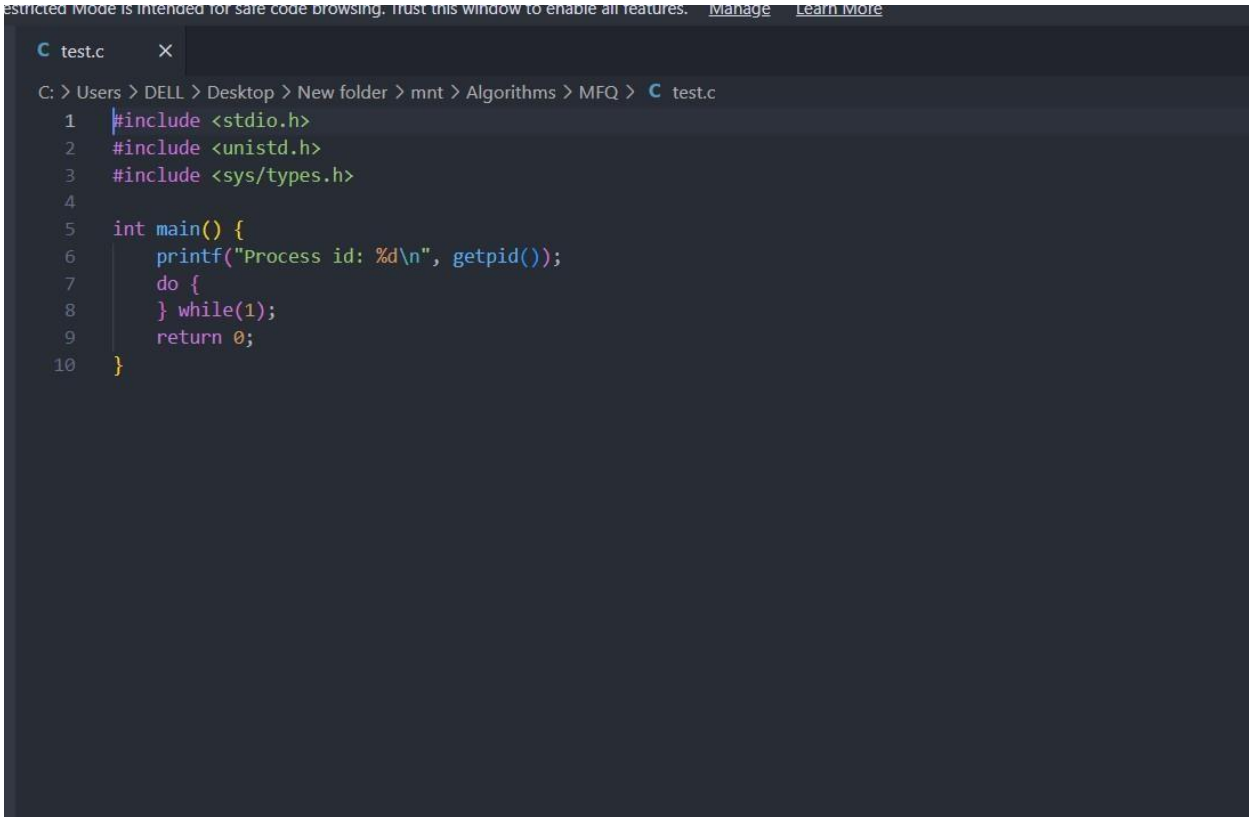
We must first make , build clean the changed directories then write the following commands:

cc test.c -o test

./test

The Test files are:

- **test.c:**



- **longrun0.c:**


longrun0.c
C: > Users > DELL > Desktop > New folder > mnt > Algorithms > Priority > longrun0.c
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <time.h>
#define LOOP_COUNT_MIN 100
#define LOOP_COUNT_MAX 100000000
long long subtime(struct timeval start,struct timeval end){
return 1e6*(end.tv_sec-start.tv_sec)+(end.tv_usec-start.tv_usec);
}
int main (int argc, char *argv[])
{
char *idStr;
unsigned int v;
int i = 0;
int iteration = 1;
int loopCount;
int maxloops;
if (argc < 3 || argc > 4)
{
printf ("Usage: %s <id> <loop count> [max loops]\n", argv[0]);
exit (-1);
}
v = getpid ();
idStr = argv[1];

longrun0.c

C: > Users > DELL > Desktop > New folder > mnt > Algorithms > Priority > longrun0.c

```c
  loopCount = atoi (argv[2]);
  if ((loopCount < LOOP_COUNT_MIN) || (loopCount > LOOP_COUNT_MAX))
  {
    printf ("%s: loop count must be between %d and %d (passed %s)\n", argv[0], LOOP_COUNT_MIN, LOOP_COUNT_MAX, argv[2]);
    exit (-1);
  }
  if (argc == 4)
  {
    maxloops = atoi (argv[3]);
  }
  else
  {
    maxloops = 0;
  }


  struct timeval start,end,wait,curr; //store start, end (turnaround time) and temporary times for calculations
  double waiting_time=0;
  gettimeofday(&start,NULL);

  while (1)
  {
    v = (v << 4) - v;
    if (++i == loopCount)
    {
      if (iteration == maxloops)
      {
     break;
      }

      gettimeofday(&wait,NULL);
      fflush(stdout);
      gettimeofday(&curr,NULL);

      waiting_time=waiting_time+subtime(wait,curr);// add to waiting time

      iteration += 1;
      i = 0;
    }

  }


```

```c

  gettimeofday(&end,NULL);
  printf ("CPU Bound The final value of v is 0x%08x\n", v);
  printf("Turnaround time = %0.6lf\n",subtime(start,end)/1e6);
  printf("Waiting time = %0.6lf\n",waiting_time/1e6);
}

```

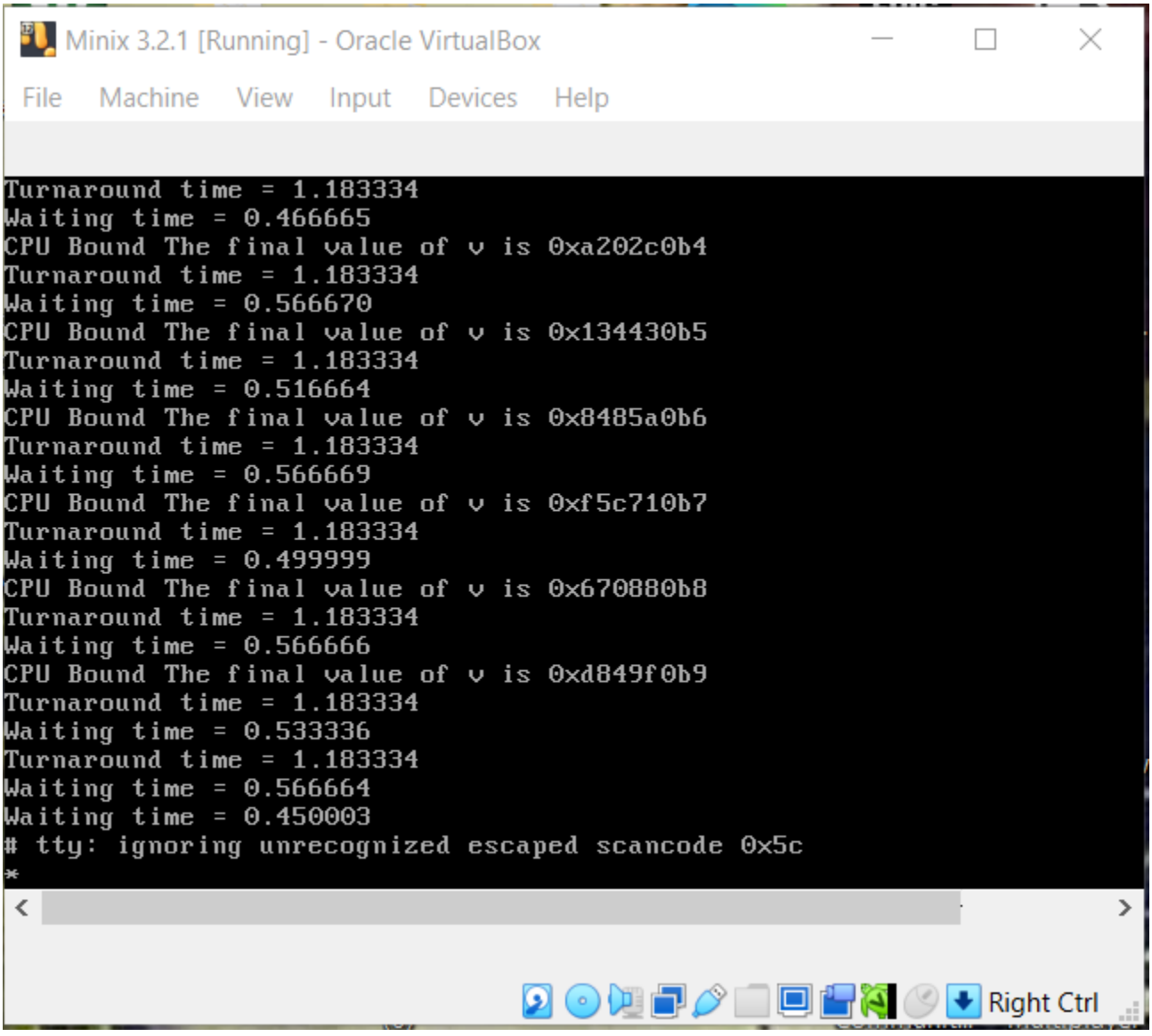
- **mytest.c:**

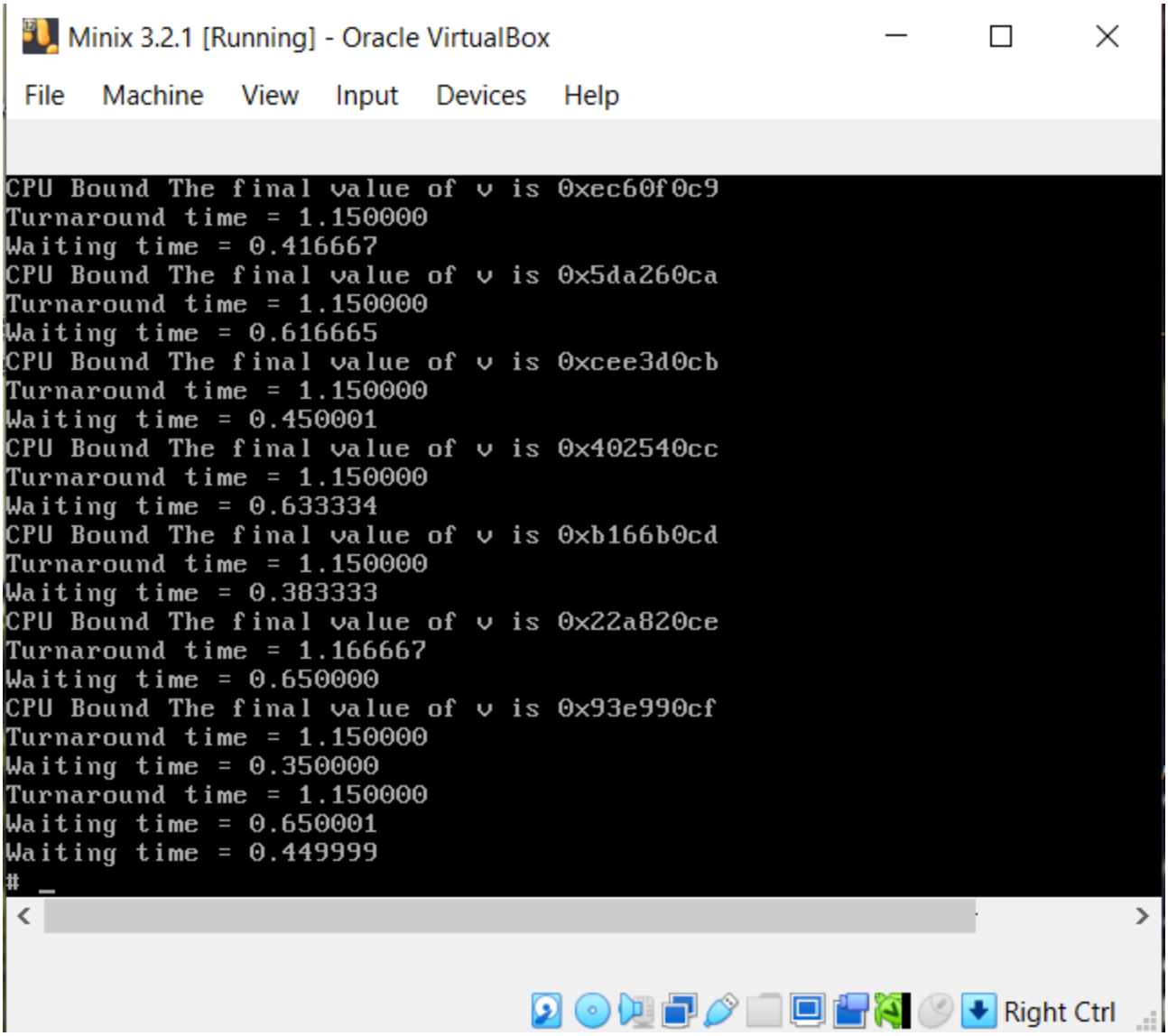
C mytest.c X

C: > Users > DELL > Desktop > New folder > mnt > Algorithms > Priority > C mytest.c

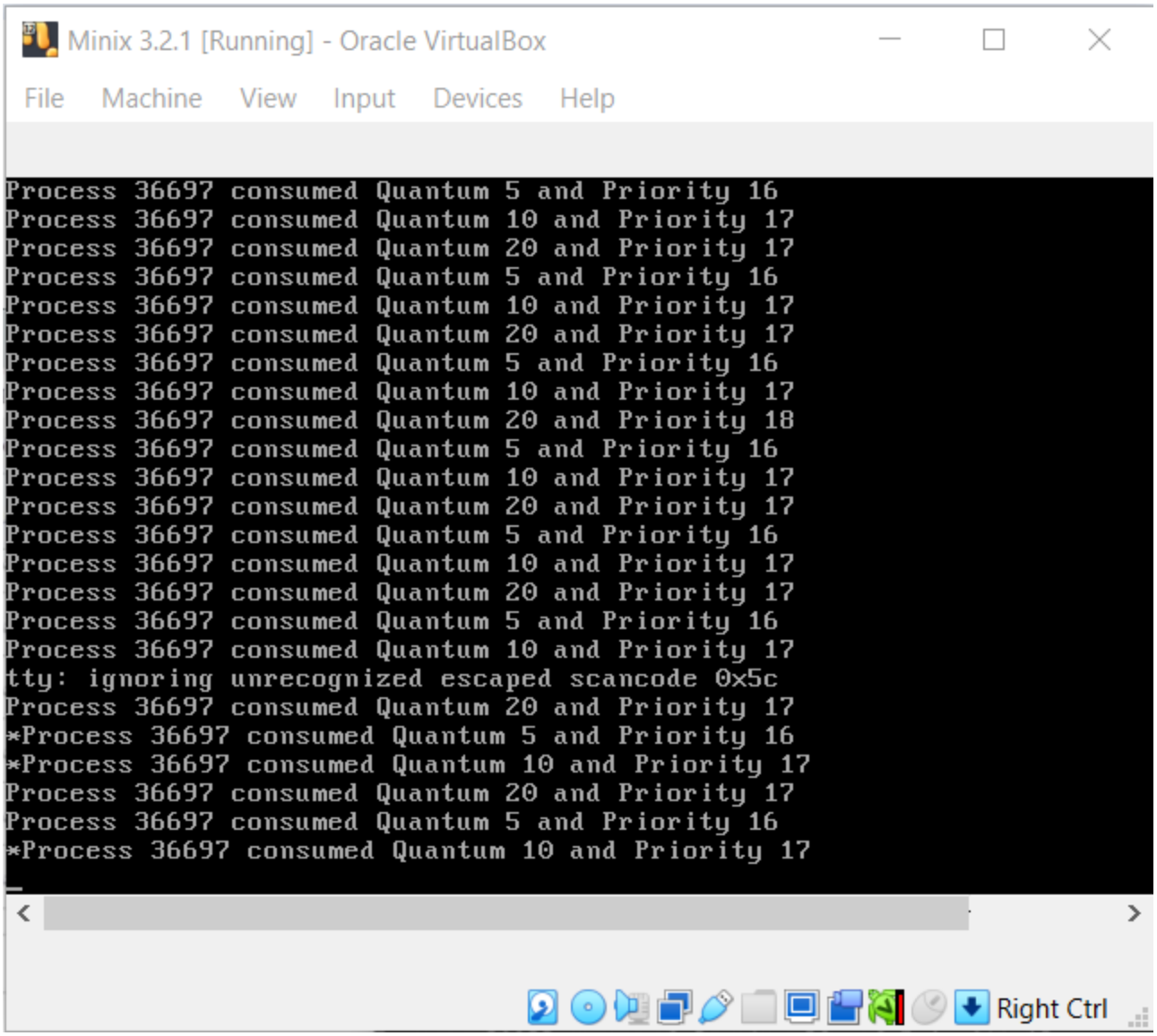
```c
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
    int pid[20];
    int i;
    char processid[100];

    for(i=1;i<=10;i++)
    {
        pid[i]=fork();
        if(pid[i]==0)
        {
            sprintf(processid,"%2d",i);
            execlp("./longrun0","./longrun0",processid,"100000","1000",NULL);

        }

    }
    for(i=1;i<=10;i++)
    {
        wait(NULL);
    }
    return 0;
}
```

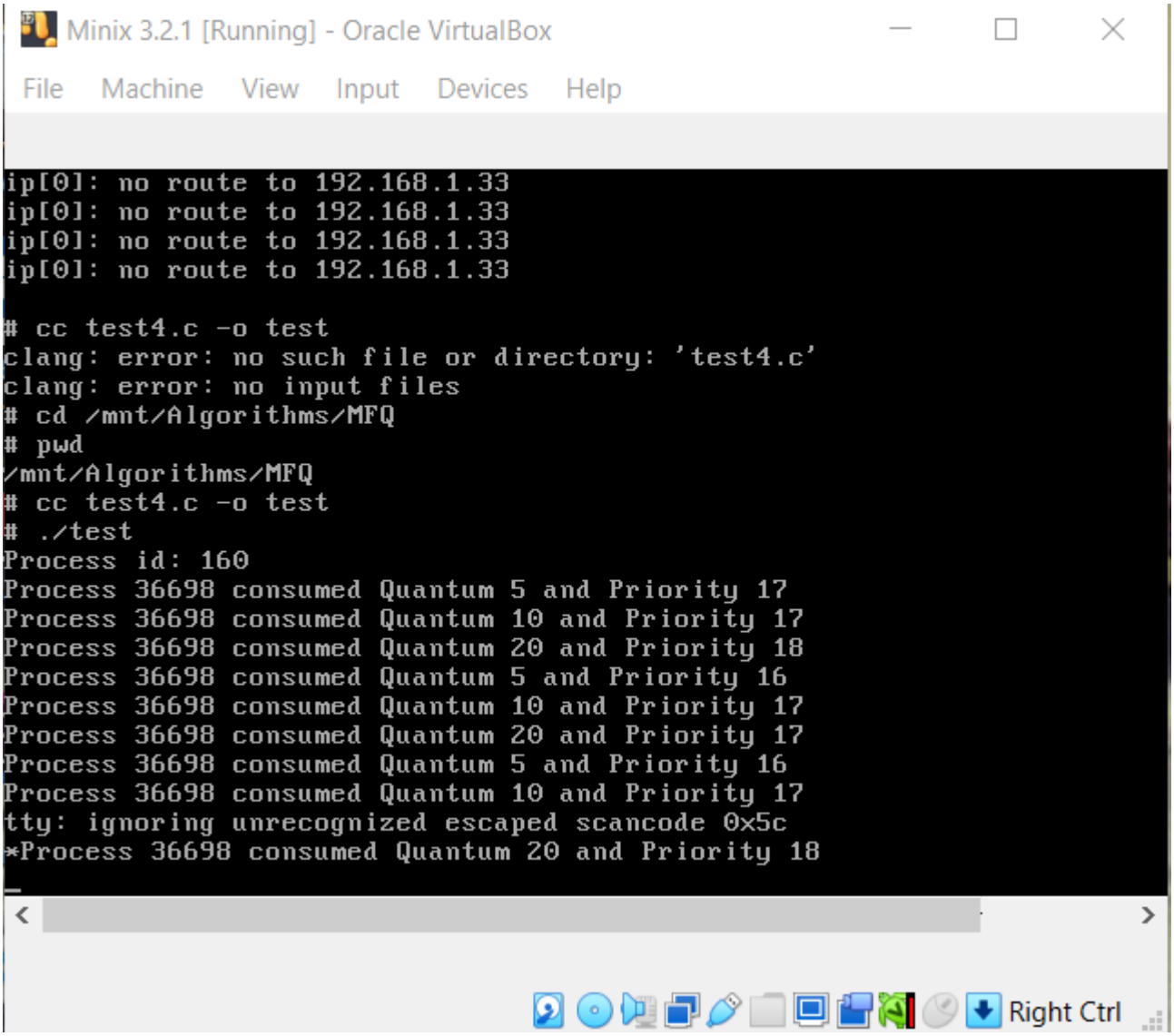
- **Round Robin Tests and Results:**



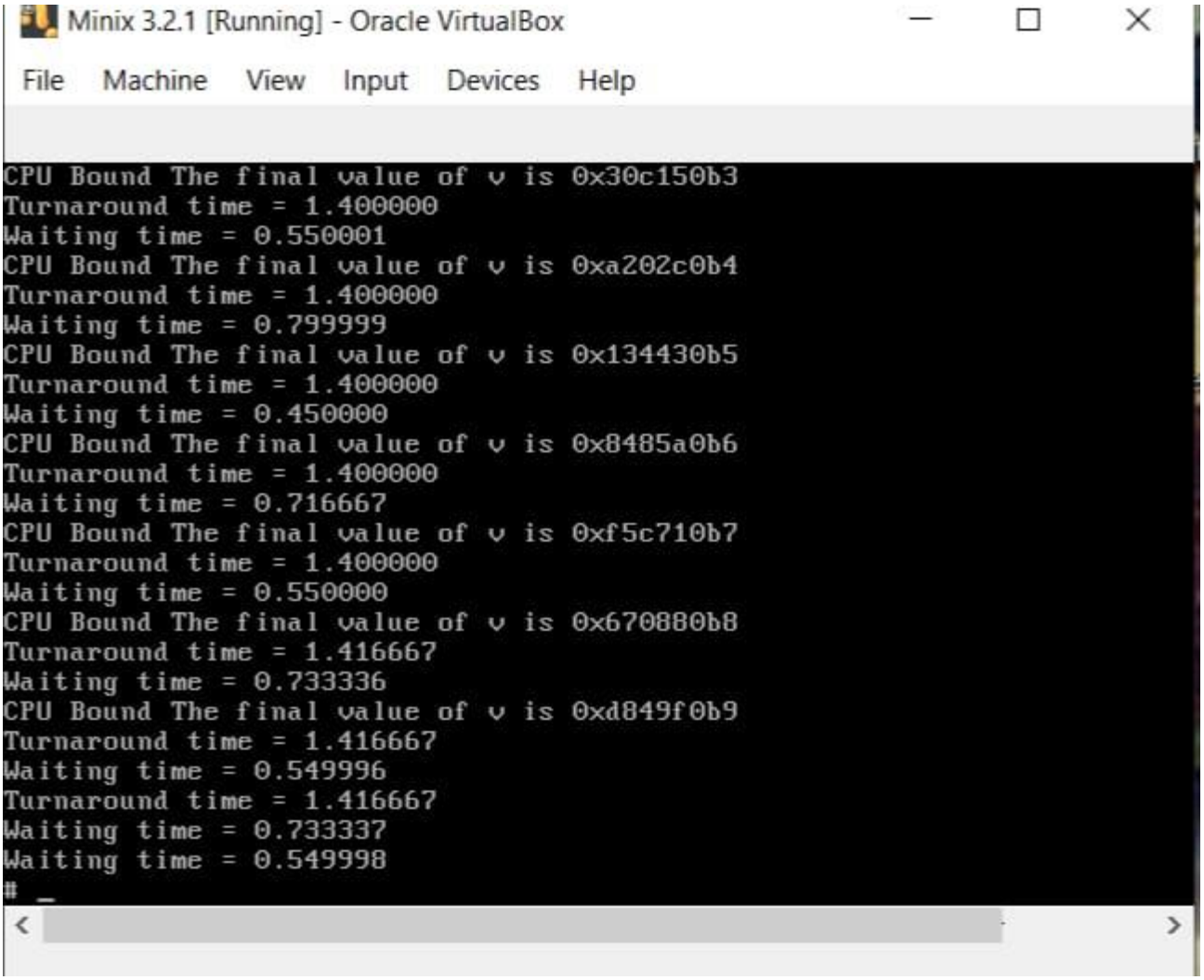

Minix 3.2.1 [Running] - Oracle VirtualBox
File Machine View Input Devices Help
CPU Bound The final value of v is 0xec60f0c9
Turnaround time = 1.150000
Waiting time = 0.416667
CPU Bound The final value of v is 0x5da260ca
Turnaround time = 1.150000
Waiting time = 0.616665
CPU Bound The final value of v is 0xcee3d0cb
Turnaround time = 1.150000
Waiting time = 0.450001
CPU Bound The final value of v is 0x402540cc
Turnaround time = 1.150000
Waiting time = 0.633334
CPU Bound The final value of v is 0xb166b0cd
Turnaround time = 1.150000
Waiting time = 0.383333
CPU Bound The final value of v is 0x22a820ce
Turnaround time = 1.166667
Waiting time = 0.650000
CPU Bound The final value of v is 0x93e990cf
Turnaround time = 1.150000
Waiting time = 0.350000
Turnaround time = 1.150000
Waiting time = 0.650001
Waiting time = 0.449999
_
Right Ctrl

Multi-Level Feed (MLFQ) Tests and Results:




Minix 3.2.1 [Running] - Oracle VirtualBox
File Machine View Input Devices Help
ip[0]: no route to 192.168.1.33
ip[0]: no route to 192.168.1.33
ip[0]: no route to 192.168.1.33
ip[0]: no route to 192.168.1.33
cc test4.c -o test
clang: error: no such file or directory: 'test4.c'
clang: error: no input files
cd /mnt/Algorithms/MFQ
pwd
/mnt/Algorithms/MFQ
cc test4.c -o test
./test
Process id: 160
Process 36698 consumed Quantum 5 and Priority 17
Process 36698 consumed Quantum 10 and Priority 17
Process 36698 consumed Quantum 20 and Priority 18
Process 36698 consumed Quantum 5 and Priority 16
Process 36698 consumed Quantum 10 and Priority 17
Process 36698 consumed Quantum 20 and Priority 17
Process 36698 consumed Quantum 5 and Priority 16
Process 36698 consumed Quantum 10 and Priority 17
tty: ignoring unrecognized escaped scancode 0x5c
*Process 36698 consumed Quantum 20 and Priority 18
Right Ctrl

Priority based tests and results:



Requirement 2: Memory Management

Required

In this requirement, you need to implement hierarchical paging in MINIX 3, with all the needed parameters (page size, number of levels, address format … etc.) are user-defined via a configuration file. Additionally, FIFO and LRU page replacement algorithms should be implemented (configuration parameters of these algorithms should be stored in configuration file too). The performance parameters (e.g., number of page faults, number of empty frames … etc.) of the hierarchical paging as well as the replacement algorithms should be collected as the size of the pages and the number of levels is changed, with a complete analysis should be provided in the report in addition to the collected results.

Memory management in minix 3.2.1

In this version of minix the memory management is handled in the Virtual Memory (VM) server (unlike previous versions where it was handled in the Process Management (PM) server.

The pagetable structure

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

    pt_pt pt_pt;

    /* When looking for a hole in virtual address space, start
     * looking here. This is in linear addresses, i.e.,
     * not as the process sees it but the position in the page
     * page table. This is just a hint.
     */
    u32_t pt_virtop;
} pt_t;
```

The pagetable (pt) contains 1 directory (a directory is like an array of pages and this directory points to the pages stored in this pt) and a pointer to an array of 1024 other

directories. This structure implements the idea of 2 level paging where the Page Table Entry (pte) is the inner table, and the Page Directory Entry (pde) is the outer level.

The structure of the minix memory address

Minix uses a 32 bit address and this address is stored inside an int.
The 32 bits are divided as follows:

- The 12 leftmost bits are the page offset, these represent the offset inside the page itself and these point to the word inside the page. (Why 12 bits? Because the page size is 4KB which is equivalent to 2^{12} bytes, we need 12 bits to represent an entire page).
- The middle 10 bits represent the page table entry inside a directory.
- The 10 rightmost bits represent the page directory entry.

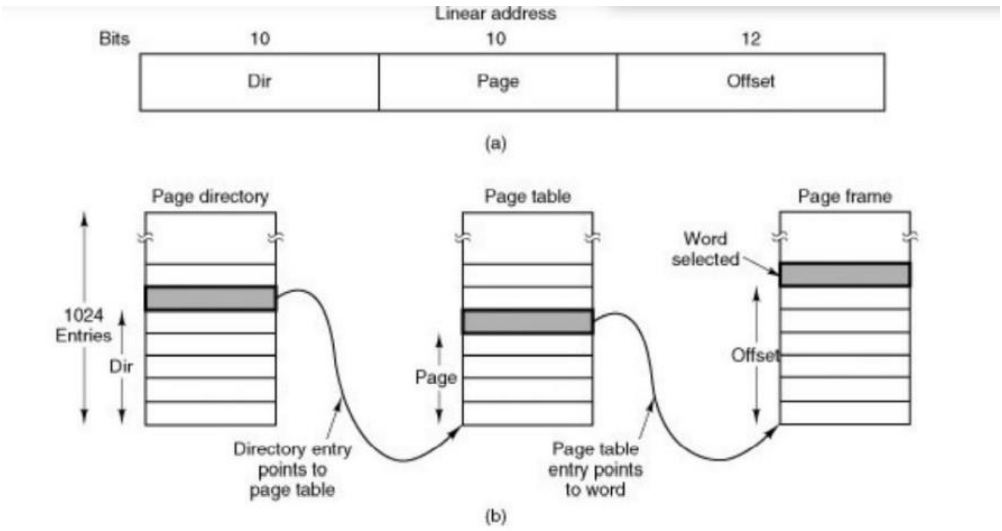
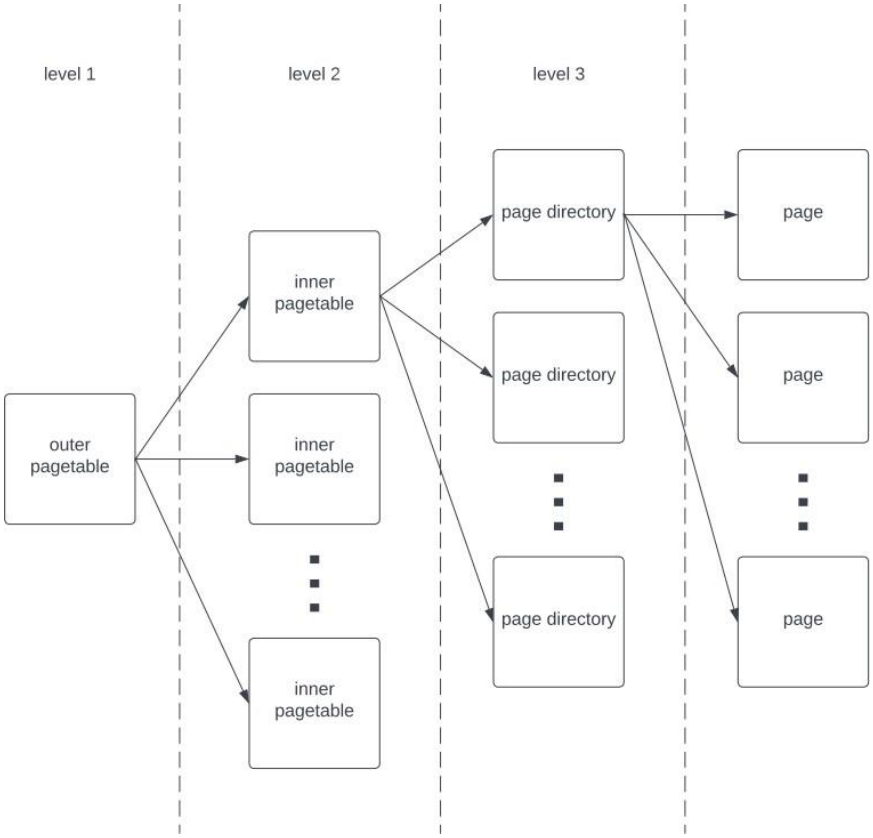


Figure 1The structure of the minix memory address

What we will do

We will add another level to the hierarchy by creating an outer pagetable that points to the inner page tables which point to the directories that point to the page frames.
The structure should look something like this:



How?

The 10 bits that are used to describe the directory entry in the address we will split them into 2 halves. The first 5 bits will represent the index of the inner pagetable inside the outer pagetable and the last 5 bits will represent the directory and the next 10 bits represent the page number.

Why?

This trades off between better memory and performance. By introducing a new level, we don't need all the directories to be loaded into the memory, we only need the outer pagetable. But of course, this influences the performance because now instead of 2 steps to get a page (one to get the directory and one to get the page) we now need 3 steps (one for the inner pagetabled and one for the directory and one for the page).

The implementation

vm.h

in this step we changed some shifting constants and masks and functions to allow the implementation of the new level.

```
#define I386_VM_OPT_ENTRIES 32 /* number of entries in the outer pagetable */
#define I386_VM_DIR_ENTRIES 32  /* Number of entries in a page dir */
```

We defined a new constant I386_VM_OPT_ENTRIES to be equal to 32 (2^5 represented by the 5 bits in the new address). This is the number of inner page entries inside 1 outer pagetable

And we changed the directory entries from 1024 to 32 (the other 5 bits). This is the number of directory entries inside 1 inner pagetable.

These 2 levels make up the old 1024 directory entries (32 x 32 = 1024).

```
#define I386_VM_OPT_ENT_SHIFT   27  /* Shift to get the outer pagetable. */
#define I386_VM_DIR_ENT_SHIFT   22  /* Shift to get entry in page dir. */
#define I386_VM_DIR_ENT_MASK    0x1F    /* Mask to get entry in page dir */
```

We also introduced a new shift I386_VM_OPT_ENT_SHIFT of 27 to get the 5 rightmost bits in the address.

And we introduced a new mask I386_VM_DIR_ENT_MASK 0x1F to extract the 5 directory bits.

```
#define I386_VM_OPT(v)  ( (v) >> I386_VM_OPT_ENT_SHIFT)
#define I386_VM_PTE(v)  (((v) >> I386_VM_PT_ENT_SHIFT) & I386_VM_PT_ENT_MASK)
#define I386_VM_PDE(v)  ( ((v) >> I386_VM_DIR_ENT_SHIFT) & I386_VM_DIR_ENT_MASK)
```

We added a new function that will be used to extract the inner pagetable entry by shifting the address to the right 27 times to extract the bits 0-4 inclusive.

We modified the directory entry function. The directory entry bits are bits from 5-9 inclusive in the 32 bit address so to extract them we shift to the right 22 times then we “and” this value with 11111 mask (0x1F) to eliminate the 0-4 bits.

opt.h

In this step we implemented our new structure, the outer pagetable.
We will start by creating an outer pagetable in servers/vm/opt.h

```
#ifndef MINIX_R3_2_1_OPT_H
#define MINIX_R3_2_1_OPT_H

#include <machine/vm.h>

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

/* A pagetable. */
typedef struct {
    //this is an array of the inner pagetables which contain the directories
    pt_t *inner_page_tables[I386_VM_OPT_ENTRIES];
} opt_t;


#endif //MINIX_R3_2_1_OPT_H
```

This outer page table contains an array of the inner page tables.

vmproc.c

Now we need to make the processes use the outer page table instead of the inner page table so instead of a pt_t member inside the process we will put an opt_t.

```
1
2   #ifndef _VMPROC_H
3   #define _VMPROC_H 1
4
5   #include <minix/bitmap.h>
6   #include <machine/archtypes.h>
7
8   #include "opt.h"
9   #include "vm.h"
10  #include "regionavl.h"
11
12  struct vmproc;
13
14  typedef void (*callback_t)(struct vmproc *who, message *m);
15
16  struct vmproc {
17      int     vm_flags;
18      endpoint_t  vm_endpoint;
19      opt_t       vm_pt;  /* page table data */
20      struct boot_image *vm_boot; /* if boot time process */
21
22      /* Regions in virtual address space. */
23      region_avl vm_regions_avl;
24      vir_bytes  vm_region_top;   /* highest vaddr last inserted */
25
26      bitchunk_t vm_call_mask[VM_CALL_MASK_SIZE];
27
```

```
28      /* State for requests pending to be done to vfs on behalf of
29       * this process.
30       */
31      callback_t vm_callback;   /* function to call on vfs reply */
32      int vm_callback_type; /* expected message type */
33
34      int vm_slot;        /* process table slot */
35      int vm_yielded;     /* yielded regions */
36
37      union {
38          struct {
39              cp_grant_id_t gid;
40          } open; /* VM_VFS_OPEN */
41      } vm_state;     /* Callback state. */
42  #if VMSTATS
43      int vm_bytecopies;
44  #endif
45  };
46
47  /* Bits for vm_flags */
48  #define VMF_INUSE   0x001   /* slot contains a process */
49  #define VMF_EXITING 0x002   /* PM is cleaning up this process */
50  #define VMF_WATCHEXIT   0x008   /* Store in queryexit table */
51
52  #endif
53
```

pagetable.c

Now we need to adapt the the pagetable functions to use our new outer pagetable and to access the inner pagetable from it.

findhole

```
static u32_t findhole(int pages) {
/* Find a space in the virtual address space of VM. */
    u32_t curv;
    int pde = 0, try_restart;
    static u32_t lastv = 0;
    opt_t *pt = &vmprocess->vm_pt;
    vir_bytes vmin, vmax;
#if defined(__arm__)
    u32_t holev;
#endif

    vmin = (vir_bytes)(&_end); /* marks end of VM BSS */
    vmin += 1024 * 1024 * 1024;    /* reserve 1GB virtual address space for VM heap */
    vmin &= ARCH_VM_ADDR_MASK;
    vmax = VM_STACKTOP;

    /* Input sanity check. */
    assert(vmin + VM_PAGE_SIZE >= vmin);
    assert(vmax >= vmin + VM_PAGE_SIZE);
    assert((vmin % VM_PAGE_SIZE) == 0);
    assert((vmax % VM_PAGE_SIZE) == 0);
#if defined(__arm__)
    assert(pages > 0);
#endif

    curv = lastv;
```

We changed the pt_t pointer here to an opt pointer to point to the outer pagetable instead of the inner pagetable

```
    if(curv < vmin || curv >= vmax)
        curv = vmin;

    try_restart = 1;

    /* Start looking for a free page starting at vmin. */
    while(curv < vmax) {
        int pte;
#if defined(__arm__)
        int i, nohole;
#endif

        assert(curv >= vmin);
        assert(curv < vmax);

#if defined(__i386__)
        int opt = I386_VM_OPT(curv);
        pde = I386_VM_PDE(curv);
        pte = I386_VM_PTE(curv);
#elif defined(__arm__)
        holev = curv; /* the candidate hole */
        nohole = 0;
        for (i = 0; i < pages && !nohole; ++i) {
            if(curv >= vmax) {
            break;
            }
#endif
```

Here we added a new variable opt to carry the inner pagetable index. This calls the definition above to extract the 5 bit outer pagetable address.

```
#if defined(__i386__)
        if(!(pt->inner_page_tables[opt]->pt_dir[pde] & ARCH_VM_PDE_PRESENT) ||
           !(pt->inner_page_tables[opt]->pt_pt[pde][pte] & ARCH_VM_PAGE_PRESENT)) {
#elif defined(__arm__)

        /* if page present, no hole */
        if((pt->inner_page_tables[opt]->pt_dir[pde] & ARCH_VM_PDE_PRESENT) &&
           (pt->inner_page_tables[opt]->pt_pt[pde][pte] & ARCH_VM_PTE_PRESENT))
        nohole = 1;

        /* if not contiguous, no hole */
        if (curv != holev + i * VM_PAGE_SIZE)
        nohole = 1;

        curv+=VM_PAGE_SIZE;
    }

    /* there's a large enough hole */
    if (!nohole && i == pages) {
#endif
        lastv = curv;
#if defined(__i386__)
        return curv;
#elif defined(__arm__)
        return holev;
#endif
    }
```

We modified those 2 if statements to access the pages through the outer page table

```
#if defined(__i386__)
        curv+=VM_PAGE_SIZE;

#elif defined(__arm__)
        /* Reset curv */
#endif
        if(curv >= vmax && try_restart) {
            curv = vmin;
            try_restart = 0;
        }
    }

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

    return NO_MEM;
}
```

vm_allocpages

```
void *vm_allocpages(phys_bytes *phys, int reason, int pages)
{
/* Allocate a page for use by VM itself. */
    phys_bytes newpage;
    vir_bytes loc;
    opt_t *pt;
    int r;
    static int level = 0;
    void *ret;
    u32_t mem_flags = 0;

    pt = &vmprocess->vm_pt;
    assert(reason >= 0 && reason < VMP_CATEGORIES);

    assert(pages > 0);

    level++;

    assert(level >= 1);
    assert(level <= 2);

    if((level > 1) || !pt_init_done) {
        void *s;

        if(pages == 1) s=vm_getsparepage(phys);
        else if(pages == 4) s=vm_getsparepagedir(phys);
        else panic("%d pages", pages);
```

Here we adjusted the allocate page function to use our outerpage

```
        level--;
        if(!s) {
            util_stacktrace();
            printf("VM: warning: out of spare pages\n");
        }
        if(!is_staticaddr(s)) vm_self_pages++;
        return s;
    }

#if defined(__arm__)
    if (reason == VMP_PAGEDIR) {
        mem_flags |= PAF_ALIGN16K;
    }
#endif

    /* VM does have a pagetable, so get a page and map it in there.
     * Where in our virtual address space can we put it?
     */
    loc = findhole(pages);
    if(loc == NO_MEM) {
        level--;
        printf("VM: vm_allocpage: findhole failed\n");
        return NULL;
    }

    /* Allocate page of memory for use by VM. As VM
     * is trusted, we don't have to pre-clear it.
     */
```

```
    if((newpage = alloc_mem(pages, mem_flags)) == NO_MEM) {
        level--;
        printf("VM: vm_allocpage: alloc_mem failed\n");
        return NULL;
    }

    *phys = CLICK2ABS(newpage);

    /* Map this page into our address space. */
    if((r=pt_writemap(vmprocess, pt, loc, *phys, VM_PAGE_SIZE*pages,
        ARCH_VM_PTE_PRESENT | ARCH_VM_PTE_USER | ARCH_VM_PTE_RW
#if defined(__arm__)
        | ARM_VM_PTE_WB | ARM_VM_PTE_SHAREABLE
#endif
        , 0)) != OK) {
        free_mem(newpage, pages);
        printf("vm_allocpage writemap failed\n");
        level--;
        return NULL;
    }

    if((r=sys_vmctl(SELF, VMCTL_FLUSHTLB, 0)) != OK) {
        panic("VMCTL_FLUSHTLB failed: %d", r);
    }

    level--;


    /* Return user-space-ready pointer to it. */
    ret = (void *) loc;

    vm_self_pages++;
    return ret;
}
```

vm_pagelockv

```
void vm_pagelock(void *vir, int lockflag)
{
/* Mark a page allocated by vm_allocpage() unwritable, i.e. only for VM. */
    vir_bytes m = (vir_bytes) vir;
    int r;
    u32_t flags = ARCH_VM_PTE_PRESENT | ARCH_VM_PTE_USER;
    opt_t *pt;

    pt = &vmprocess->vm_pt;

    assert(!(m % VM_PAGE_SIZE));

    if(!lockflag)
        flags |= ARCH_VM_PTE_RW;
#if defined(__arm__)
    else
        flags |= ARCH_VM_PTE_RO;
    flags |= ARM_VM_PTE_WB | ARM_VM_PTE_SHAREABLE;
```

addrok

this function verifies the passed address, we just modified it to use out outer pagetable

```
int vm_addrok(void *vir, int writeflag) {
    opt_t *pt = &vmprocess->vm_pt;
    int pde, pte, opt;
    vir_bytes v = (vir_bytes) vir;

#if defined(__i386__)
    opt = I386_VM_OPT(v);
    pde = I386_VM_PDE(v);
    pte = I386_VM_PTE(v);
#endif

    if (!(pt->inner_page_tables[opt]->pt_dir[pde] & ARCH_VM_PDE_PRESENT)) {
        printf("addr not ok: missing pde %d\n", pde);
        return 0;
    }

#if defined(__i386__)
    if(writeflag &&
        !(pt->inner_page_tables[opt]->pt_dir[pde] & ARCH_VM_PTE_RW)) {
        printf("addr not ok: pde %d present but pde unwritable\n", pde);
        return 0;
    }

#endif
    if (!(pt->inner_page_tables[opt]->pt_pt[pde][pte] & ARCH_VM_PTE_PRESENT)) {
        printf("addr not ok: missing pde %d / pte %d\n",
               pde, pte);
        return 0;
    }

#if defined(__i386__)
    if(writeflag &&
        !(pt->inner_page_tables[opt]->pt_pt[pde][pte] & ARCH_VM_PTE_RW)) {
        printf("addr not ok: pde %d / pte %d present but unwritable\n",
#endif
    opt, pde, pte);
    return 0;
}

return 1;
}
```

pt_ptalloc

this function allocates a page table and writes its address to the page directory, we need to modify it to search the outer pagetable to find our directory

```
static int pt_ptalloc(opt_t *pt, int pde, u32_t flags) {
/* Allocate a page table and write its address into the page directory. */
    int i;
    phys_bytes pt_phys;

    /* Argument must make sense. */
    assert(pde >= 0 && pde < ARCH_VM_DIR_ENTRIES);
    assert(!(flags & ~(PTF_ALLFLAGS)));

    /* We don't expect to overwrite page directory entry, nor
     * storage for the page table.
     */
    assert(!(pt->pt_dir[pde] & ARCH_VM_PDE_PRESENT));
    assert(!pt->pt_pt[pde]);

    /* Get storage for the page table. */
    if (!(opt->inner_page_tables[opt]->pt_pt[pde] = vm_allocpage(&pt_phys, VMP_PAGETABLE)))
        return ENOMEM;

    for (j = 0; j < ARCH_VM_DIR; j++)
        for (i = 0; i < ARCH_VM_PT_ENTRIES; i++)
            pt->inner_page_tables[j]->pt_pt[pde][i] = 0;   /* Empty entry. */
```

This loop used to initialize the pagetable so we added a nested loop to initialize all the inner pagetables and their directories.

```
    /* Make page directory entry.
     * The PDE is always 'present,' 'writable,' and 'user accessible,'
     * relying on the PTE for protection.
     */
#if defined(__i386__)
    pt->inner_page_tables[j]->pt_dir[pde] = (pt_phys & ARCH_VM_ADDR_MASK) | flags
        | ARCH_VM_PDE_PRESENT | ARCH_VM_PTE_USER | ARCH_VM_PTE_RW;
#endif

    return OK;
}
```

pt_ptmap

```
int pt_ptmap(struct vmproc *src_vmp, struct vmproc *dst_vmp) {
/* Transfer mappings to page dir and page tables from source process and
 * destination process. Make sure all the mappings are above the stack, not
 * to corrupt valid mappings in the data segment of the destination process.
 */
    int pde, r;
    phys_bytes physaddr;
    vir_bytes viraddr;
    opt_t *pt;

    pt = &src_vmp->vm_pt;

#if LU_DEBUG
    printf("VM: pt_ptmap: src = %d, dst = %d\n",
        src_vmp->vm_endpoint, dst_vmp->vm_endpoint);
#endif

    /* Transfer mapping to the page directory. */
    viraddr = (vir_bytes) pt->inner_page_tables[j]->pt_dir;
    physaddr = pt_dir_phys & ARCH_VM_ADDR_MASK;
#if defined(__i386__)
    if((r=pt_writemap(dst_vmp, &dst_vmp->vm_pt, viraddr, physaddr, VM_PAGE_SIZE,
        ARCH_VM_PTE_PRESENT | ARCH_VM_PTE_USER | ARCH_VM_PTE_RW,
#elif defined(__arm__)
    if((r=pt_writemap(dst_vmp, &dst_vmp->vm_pt, viraddr, physaddr, ARCH_PAGEDIR_SIZE,
        ARCH_VM_PTE_PRESENT | ARCH_VM_PTE_USER | ARCH_VM_PTE_RW |
        ARM_VM_PTE_WB | ARM_VM_PTE_SHAREABLE,
#endif
    WMF_OVERWRITE)) != OK) {
        return r;
```

This function transfers the mapping of the directories and pages to the destination process, we modified it to properly access the directories and pages from the outer page.

```
    /* Scan all non-reserved page-directory entries. */
    for (opt = 0; opt < ARCH_VM_DIR_ENTRIES; opt++)
        for (pde = 0; pde < ARCH_VM_DIR_ENTRIES; pde++) {
            if (!(pt->inner_page_tables[opt]->pt_dir[pde] & ARCH_VM_PDE_PRESENT)) {
                continue;
            }

            /* Transfer mapping to the page table. */
            viraddr = (vir_bytes) pt->pt_pt[pde];
#if defined(__i386__)
            physaddr = pt->pt_dir[pde] & ARCH_VM_ADDR_MASK;
#elif defined(__arm__)
            physaddr = pt->pt_dir[pde] & ARCH_VM_PDE_MASK;
#endif
            if ((r = pt_writemap(dst_vmp, &dst_vmp->vm_pt, viraddr, physaddr, VM_PAGE_SIZE,
                                 ARCH_VM_PTE_PRESENT | ARCH_VM_PTE_USER | ARCH_VM_PTE_RW
#ifdef __arm__
                    | ARCH_VM_PTE_PRESENT | ARCH_VM_PTE_USER | ARCH_VM_PTE_RW |
                    ARM_VM_PTE_WB | ARM_VM_PTE_SHAREABLE
#endif
                    ,
                                 WMF_OVERWRITE)) != OK) {
                return r;
            }
        }

    return OK;
```

We also added this nested for loop to loop over the inner pagetables in the outer pagetable to check if the directory is reserved or not

opt_new

this is a new function we created that creates a new outer pagetable

```
int opt_new(opt_t *pt) {
    int i, r;

    //here we verify the inner pagetables array then we allocate memory to it
    if (!pt->inner_page_tables &&
        !(pt->inner_page_tables = vm_allocpages((phys_bytes *), VMP_PAGEDIR, ARCH_PAGEDIR_SIZE / VM_PAGE_SIZE))) {
        return ENOMEM;
    }

    //here we initialize the inner pagetables
    for (i = 0; i < ARCH_VM_DIR_ENTRIES; i++) {
        pt->inner_page_tables[i] = 0; /* invalid entry (PRESENT bit = 0) */
        pt->inner_page_tables[i] = NULL;
    }

    /* Map in kernel. */
    if ((r = pt_mapkernel(pt)) != OK)
        return r;

    return OK;
}
```

*Page r*eplacement algorithm

```
#define MAX_PAGES_IN_MEM 262144 //that is the size of the memory (4GB) / the size of the page (4KB)



//-----------------------------------------------------------------------------------------------------------------
//                                          LRU
//-----------------------------------------------------------------------------------------------------------------

//first we will implement LRU algorithm
/* LRU uses a doubly linked list of pages
 * the front of the list is the least recently used page and in the back is the most recently used page
 */


struct page {
    int pageno;
    page *next, *prev;
};
```

We defined the maximum number of pages that can be inside the memory using the equation:
(size of the memory) / (page size)
We added a new file replacement.c which is dedicated to the replacement algorithms.
First, we implemented the least recently used algorithm:

LRU:

To implement the least recently used algorithm we used a doubly linked list, at the head of the list is the least recently used page and at the tail we have the most recently used page. We chose to keep a pointer on the tail for faster insertions at the back of the list.

Every time a page is used the function use_page is called. This function first checks if the page is in the memory or not, if so the page is sent to the back of the list because it is the most recently used page. If the page was not found in the memory, we check if the memory is full or not, if not we simply add the page to the back of the list. If the memory is full, we need to use our page replacement algorithm. We remove the least recently used page (the head of the list) and then we add the new page to the back of the list.

Note: this make the list always sorted from least recently used to most recently used.

```
void lru_add_page(page *new_page) {
    assert(page);
    if (!lru) {
        mru = lru = new_page;
        return;
    }
    //basic linked list insertion
    //the new page will be inserted in the back because it is the most recently used
    mru->next = new_page;
    new_page->prev = mru;
    mru = new_page;

    pages_in_memory++;

    return;
}

void lru_remove_page() {
    //simply pop the head of the linked list
    assert(lru); //to make sure lru is not a null pointer
    lru = lru->next;
    pages_in_memory--;
}
```

```
void lru_use_page(page *page) {
    //check if the page is cached or not
    for (page *cur = lru; cur; cur = cur->next) {
        if (cur->pageno == page->pageno) {
            //page is already in the memory now we need to make it the most recently used
            cur->next->prev = cur->prev;
            cur->prev->next = cur->next;
            cur->prev = mru;
            cur->next = 0;
            mru = cur;
            return;
        }
    }
    //page is not in the memory so either there is place in the memory or we will need to replace the least recently used
    if (pages_in_memory >= MAX_PAGES_IN_MEM) {
        lru_remove_page(); //remove the least recently used to be replaced by the new page
    }
    lru_add_page(page); //add the new page
}
```

FIFO

The First In First Out is simpler than the LRU. We also use a doubly linked queue with 2 pointers, one pointing on the head and the other pointing on the tail

Every time a page is used we first check if the page is already in the memory, If it is in the memory we do not do anything. If the page is absent, we check if the memory is full or not, if the memory is full we pop the head of the queue and add the new page. If the memory is not full we just add the page to the end of the queue.

```
/* now we will implement the FIFO algorithm
 * this time we will simply use a linked queue with the same struct used for LRU
 */

page *fi = 0; //this is the front of the queue (first in)
page *li = 0; //this is the back of the queue (last in)

void fifo_add_page(page *page) {
    assert(page);
    if (!fi) {
        //empty queue
        fi = li = page;
        return;
    }
    //insert the page at the end of the queue
    li->next = page;
    page->prev = li;
    li = page;

    pages_in_memory++;

    return;
}

void fifo_remove_page() {
    fi = fi->next;
    pages_in_memory--;
}

void fifo_use_page(page *page) {
    //check if the page is already in the memory
    for (page *cur = fi; cur; cur = cur->next) {
        if (cur->pageno == page->pageno) {
            //page found no need for replacement
            return;
        }
    }
    //page not found
    if (pages_in_memory >= MAX_PAGES_IN_MEM)
        fifo_remove_page(); //if the memory is full pop the first in element
    fifo_add_page(page); //push the new page
}
```

Requirement 3

File System implementation

File system in Minix 3

MINIX 3 file system is a logical, self-contained entity with i-nodes, directories, and data blocks. It can be stored on any block device, such as a floppy disk or a hard disk partition.

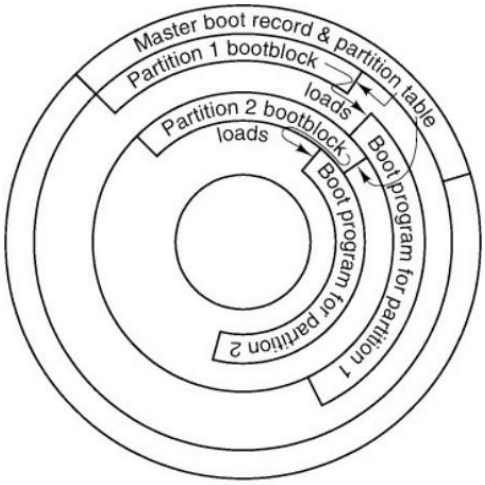


Figure 2Minix disc layout

The Boot Block contain executable code, where the size of the block is always 1024 bytes (two disk sectors).
When the computer is turned on, the hardware reads the boot block from the boot device into memory, jumps to it, and begins executing its code.
The boot block code begins the process of loading the operating system itself.
Once the system has been booted, the boot block is not used any more.

Super block

Like the boot block, the superblock is always **1024** bytes.
The main function of the superblock is to tell the file system how big the various pieces of the file system are.
Given the block size and the number of i-nodes, it is easy to calculate the size of the i-node bitmap and the number of blocks of i-nodes.
The superblock table holds a number of fields not present on the disk. These include flags that allow a device to be specified as read-only or as following a byte-order convention opposite to the standard, and fields to speed access by indicating points in the bitmaps below which all bits are marked used.

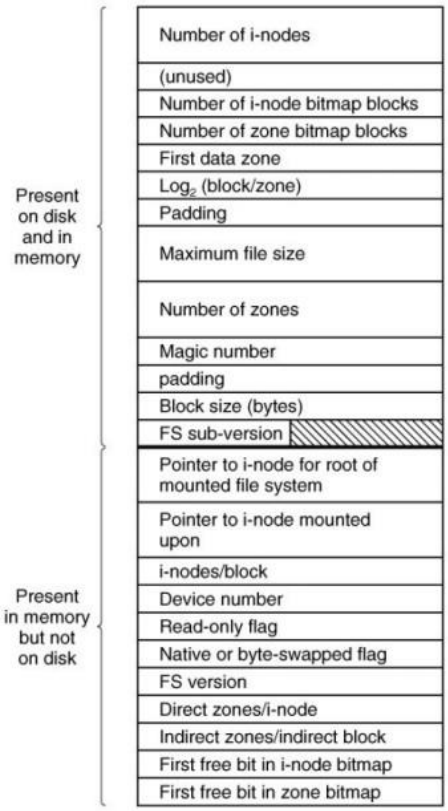


Figure 3 Super block

Bitmaps

Minix tracks free i-nodes and zones using bitmaps.
When a file is deleted, its corresponding i-nodes is deleted by marking the coresponding bits to 0
Disk storage is allocated in terms of 2^n blocks called zones.
A **zone** is a method to allows allocating as many blocks next to each other (on the same cylinder) to safe load time.
To create a new file the disk is searched for the first free i-node then it starts allocating its.
The first free block is already cached in the super block

I-nodes

Minix i-node tracks meta-data and data blocks of the file.
When a file is opened, its i-node is located and brought into the i-node table in memory, where it remains until the file is closed.
The link field is the ref-count

- records how many directory entries point to the i-node, so the file system knows when to release the file's storage.

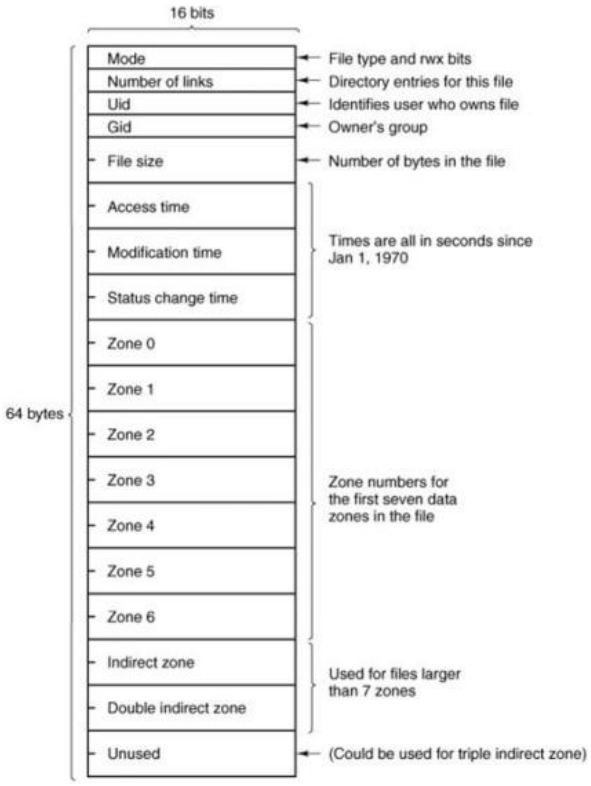


Figure 4 Nodes

Implementing User Extent in Minix3 File System

Following files are changed:

- **/usr/src/servers/mfs/super.h**
- **/usr/src/servers/mfs/super.c**

In order to implement this in minix file system, we need to look closely at the code in which file system is implemented

Like the kernel and process manager, various data structures and tables used in the file system are defined in header files. In **super.h**, we have the declaration of the superblock table. When the system is booted, the superblock for the root device is loaded here. As file systems are mounted, their superblocks go here as well. As with other tables, **super_block** is declared as EXTERN.

To implement user extent, we defined a variable named **USER_DEFINED_SIZE** which indicates the user extent we are going to use in allocating files. Declared in **/usr/src/servers/mfs/super.h**

```
/* Super block table.  The root file system and every mounted file system
 * has an entry here.  The entry holds information about the sizes of the bit
 * maps and inodes.  The s_ninodes field gives the number of inodes available
 * for files and directories, including the root directory.  Inode 0 is
 * on the disk, but not used.  Thus s_ninodes = 4 means that 5 bits will be
 * used in the bit map, bit 0, which is always 1 and not used, and bits 1-4
 * for files and directories.  The disk layout is:
 *
 *    Item        # blocks
 *    boot block      1
 *    super block     1    (offset 1kB)
 *    inode map     s_imap_blocks
 *    zone map      s_zmap_blocks
 *    inodes        (s_ninodes + 'inodes per block' - 1)/'inodes per block'
 *    unused        whatever is needed to fill out the current zone
 *    data zones    (s_zones - s_firstdatazone) << s_log_zone_size
 *
 * A super_block slot is free if s_dev == NO_DEV.
 */

#define USER_DEFINED_SIZE 3

EXTERN struct super_block {
  ino_t s_ninodes;    /* # usable inodes on the minor device */
  zone1_t  s_nzones;    /* total device size, including bit maps etc */
  short s_imap_blocks;    /* # of blocks used by inode bit map */
```

Then, moving to **/usr/src/servers/mfs/super.c**

The following is added:

Include <machine/param.h> and <machine/vmparam.h> in order to use its parameters in code

```
#include "fs.h"
#include <string.h>
#include <assert.h>
#include <minix/com.h>
#include <minix/u64.h>
#include <minix/bdev.h>
#include <machine/param.h>
#include <machine/vmparam.h>
#include "buf.h"
#include "inode.h"
#include "super.h"
#include "const.h"
```

Then, declare a static variable which indicates the number of used blocks in order to keep track of the available space in our file system. The variable named **usedBlocks** and initialized with 0.

```
26
27   static u32_t usedBlocks = 0; /* to indicate the number of blocks that are used */
28
```

Since the file **super.c** contains procedures that manage the superblock and the bitmaps. These procedures are defined in this file:

Procedure	Function
alloc_bit	Allocate a bit from the zone or i-node map
free_bit	Free a bit in the zone or i-node map
get_super	Search the superblock table for a device
get_block_size	Find block size to use
read_super	Read a superblock

The following procedures are edited:

alloc_bit()

alloc_bit() is used to search the relevant bitmap. The search involves three nested loops, as follows:

1. The outer loops on all the blocks of a bitmap
2. The middle loops on all the words inside the block.
3. The inner loops on all the bits of the word.

The middle loop works by seeing if the current word is equal to the one's complement of zero, that is, a complete word full of 1s. If so, it has no free i-nodes or zones, so the next word is tried.

When a word with a different value is found, it must have at least one 0 bit in it, so the inner loop is entered to find the free (0) bit.

And in order to make work on a user extent the inner loop is changed. Now the loop searches the word for several free space (0) equals to the number of the user extent defined (**USER_DEFINED_SIZE**)

```
77        /* Does this word contain a free bit? */
78        if (*wptr == (bitchunk_t) ~0) continue;
79
80        /* Find and allocate the free bit. */
81        int count = 0;
82        k = (bitchunk_t) conv4(sp->s_native, (int) *wptr);
83        for (i = 0; count < USER_DEFINED_SIZE && i<16; ++i) {
84          if((k & (1 << i)) == 0){
85            count++;
86          }
87        }
88
89        if(count!= USER_DEFINED_SIZE) continue;
90
91        /* Bit number from the start of the bit map. */
92        b = ((bit_t) block * FS_BITS_PER_BLOCK(sp->s_block_size))
93            + (wptr - &b_bitmap(bp)[0]) * FS_BITCHUNK_BITS
94            + i;
95
```

Now the allocation will only happen if the free space equals the number of user extent
When the number of free spaces is equal to the user defined size, the allocation is also based on the user extent

```
98
99        /* Allocate and return bit number. */
100       for(int j=i;j>i-USER_DEFINED_SIZE;j--){
101         k |= 1 << i;
102       }
103       *wptr = (bitchunk_t) conv4(sp->s_native, (int) k);
104       MARKDIRTY(bp);
105       put_block(bp, MAP_BLOCK);
```

Here we check if the block is used block, if it is then increase the number of used blocks by 1

```
106     if(map == ZMAP) { /* in case of a used block, increase the number of usedBlocks variable */
107       usedBlocks++;
108       lmfs_blockschange(sp->s_dev, 1);
109     }
```

free_bit()

free_bit() calculates which bitmap block contains the bit to free and sets the proper bit to 0 by calling get_block(), zeroing the bit in memory and then calling put_block(). In free_bit() function we don't need to search
Since the allocation is now based on a user extent which does not allocate a single bit, we needed to change free_bit() also to work the same
Make it loop a number of times equals to the user extent

```
146
147     bit = bit_returned % FS_BITCHUNK_BITS;
148     for(int j=bit;j>bit-USER_DEFINED_SIZE;j--){
149         mask = 1 << j;
150     }
151
152     bp = get_block(sp->s_dev, start_block + block, NORMAL);
```

Here we check if the block is now not used, then decrease the number of used blocks by 1

```
165
166     if(map == ZMAP) { /* decrease the number of usedBlocks variable */
167     usedBlocks--;
168     lmfs_blockschange(sp->s_dev, -1);
169     }
170   }
```

rw_super()

Here we changed the type of **sbbuf** pointer from **static char** to **char**, and also defined a new pointer to a struct of buf named **bp**

```
202   /*===========================================================================*
203    *        rw_super                 *
204    *===========================================================================*/
205   static int rw_super(struct super_block *sp, int writing)
206   {
207   /* Read/write a superblock. */
208     int r;
209     char *sbbuf;
210     dev_t save_dev = sp->s_dev;
211     struct buf *bp;
212
```

Before reading the super block, the cache block size should be big enough that the whole super block is in block 0. So, we assert the following

```
236     assert(lmfs_fs_block_size() >= sizeof(struct super_block) + SUPER_BLOCK_BYTES);
237     assert(SUPER_BLOCK_BYTES >= sizeof(struct super_block));
238     assert(SUPER_BLOCK_BYTES >= ondisk_bytes);
```

Then, we take a copy between the disk block and the super block buffer that we created

```
239
240     if(!(bp = get_block(sp->s_dev, 0, NORMAL)))
241     panic("get_block of superblock failed");
```

Then, make sbbuf point to the disk block at the super block offset. To do that we call b_data(bp) which returns the pointer to the first byte of it and then add the super block size to it

```
242
243   /* sbbuf points to the disk block at the superblock offset */
244   sbbuf = (char *) b_data(bp) + SUPER_BLOCK_BYTES; /* a pointer to the disk block
245                   (b_data returns a pointer to the first byte + super block bytes) */
246
```

Then, we check if the condition is writing, accordingly the writing case is edited to the following, else leave it as default

```
246
247   if(writing) {
248     memset(b_data(bp), 0, lmfs_fs_block_size()); /* delete data from cache */
249     memcpy(sbbuf, sp, ondisk_bytes); /* copy the buffer which contains the data into the disk */
250   lmfs_markdirty(bp);
251   } else {
252   memset(sp, 0, sizeof(*sp));
253     memcpy(sp, sbbuf, ondisk_bytes);
254     sp->s_dev = save_dev;
255   }
256
```

Then, call put_block()

```
256
257     put_block(bp, FULL_DATA_BLOCK);
258     lmfs_flushall();
```

read_super()

read_super() is partially analogous to rw_block() and rw_inode(), but it is called only to read.

The superblock is not read into the block cache at all, a request is made directly to the device for 1024 bytes starting at an offset of the same amount from the beginning of the device. Writing a superblock is not necessary in the normal operation of the system. read_super() checks the version of the file system from which it has just read and performs conversions, if necessary, so the copy of the superblock in memory will have the standard structure even when read from a disk with a different superblock structure or byte order.

But there are versions of the file system that should be handled: Version1, Version2, and Version3.

When a file is opened its i-node is read into memory and kept there until the file is closed. The inode structure definition provides for information that is kept in memory but is not written to the disk i-node.

Since there is only one version and nothing is version-specific here, When the i-node is read in from the disk, differences between V1 and V2/V3 file systems should be handled.

When reading, calling functions conv2 and conv4 to swap byte orders might be necessary. This is not used by MINIX 3, since it does not support the V1 filesystem to the extent that V1 disks can be used.
So, we changed **read_super()** to only support **V3**

```
274
275     magic = sp->s_magic;    /* determines file system type */
276
277     /* make it only supports version 3 of file system */
278     if(magic == SUPER_V2 || magic == SUPER_MAGIC) {
279     printf("MFS: only supports V3 filesystems.\n");
280     return EINVAL;
281     }
282
283     if(magic != SUPER_V3) {
284     return EINVAL;
285     }
286     version = V3;
287     native = 1;
288
289   /* If the super block has the wrong byte order, swap the fields; the magic
290    * number doesn't need conversion. */
291   sp->s_ninodes =           (ino_t) conv4(native, (int) sp->s_ninodes);
292   sp->s_nzones =          (zone1_t) conv2(native, (int) sp->s_nzones);
293   sp->s_imap_blocks =       (short) conv2(native, (int) sp->s_imap_blocks);
294   sp->s_zmap_blocks =       (short) conv2(native, (int) sp->s_zmap_blocks);
295   sp->s_firstdatazone_old =(zone1_t)conv2(native,(int)sp->s_firstdatazone_old);
296   sp->s_log_zone_size =     (short) conv2(native, (int) sp->s_log_zone_size);
297   sp->s_max_size =          (off_t) conv4(native, sp->s_max_size);
298   sp->s_zones =             (zone_t)conv4(native, sp->s_zones);
299
300   /* since there are differences between deifferent versions which will confuse data,
301    change it to version 3 only which is the latest version */
302   assert(version == V3);
303   sp->s_block_size = (unsigned short) conv2(native,(int) sp->s_block_size); /* converted variables
304                                                                               compatible only with V3 */
305   if (sp->s_block_size < PAGE_SIZE) {
306   return EINVAL;
307   }
308   sp->s_inodes_per_block = V2_INODES_PER_BLOCK(sp->s_block_size);
309   sp->s_ndzones = V2_NR_DZONES;
310   sp->s_nindirs = V2_INDIRECTS(sp->s_block_size);
311
```

Change here to **PAGE_SIZE** instead of MIN_BLOCK_SIZE

```
327
328     if (sp->s_block_size < PAGE_SIZE)
329       return(EINVAL);
330
```

And finally, added a new function to return the number of used blocks

```
384
385  static bool visited = 0;
386
387  u32_t get_usedBlocks(struct super_block *sp)
388  {
389    if(!visited)  {
390      /* how many blocks are in use? */
391      usedBlocks = sp->s_zones - count_free_bits(sp, ZMAP);
392      visited = 1;
393    }
394    return usedBlocks;
395  }
```

Requirement 4

MINIX

A brief history on MINIX:

Before MINIX came, the up-and-coming operating system was UNIX. But as UNIX evolved, it was noticeable that only the theoretical part of the operating system was used, and the actual study of UNIX was neglected. This dilemma left the users without sufficient knowledge on the operating system as a whole. To remedy this situation, Andrew S. Tanenbaum decided to write a whole new operating system from scratch that would be compatible to UNIX from the user's point of view, but the inside would be completely different. By doing this, it was ensured that the system could be of full use, and that way, users can dissect a real operating system and see what is on the inside. This operating system is MINIX, and it was made so that users can study and modify it as they please.

An advantage MINIX had over UNIX is that it was written a decade after UNIX and was structured in a more modular way. For instance, from the very beginning, in the first version of MINIX, the file system and memory manager were not part of the operating system at all but, they ran as user programs. In the current version of MINIX, this modularization was extended to I/O device drivers, which – except for the clock driver – are all run as user programs. MINIX is also designed to be readable, in the sense that its code has thousands of comments in it.

The Internal Structure of MINIX:

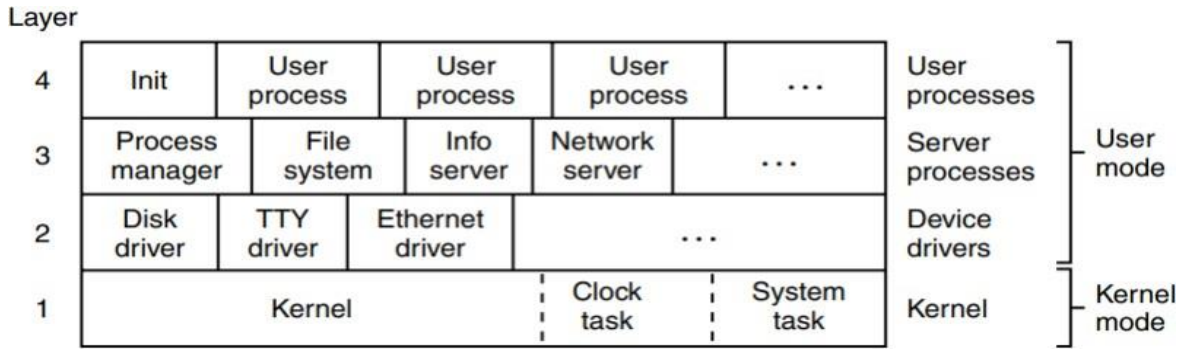
The MINIX operating system is built upon a layered, micro-kernel architecture.

The primary reason for using the micro-kernel approach is that is removes all nonessential components from the kernel and implements them as user-level programs. Hence, producing a smaller kernel.

Therefore, by using the micro-kernel architecture, most of the operating system functions are implemented as servers and are run separately form the core kernel. The main functions of the micro-kernel include inter-process communication, timer management and address space management.

The layered structure means that the system is constructed of several layers, with the upper layers relying on the services provided by the lower layers. MINIX has four layers, each performing a specific and well-defined function.

Layers of MINIX structure:



Layer 1- bottom most layer- Kernel:

Figure 5 Layers of minix structure

This kernel is responsible for scheduling processes and managing the transitions between the process states, which are ready, running, and blocked states. The kernel is also responsible for handling all communication between processes (in the MINIX's case, inter process communication occurs through message passing). Message handling requires checking for legal destinations, locating the send and receive buffers in physical memory, and copying bytes from sender to receiver. The kernel is also responsible for granting access to I/O ports and interrupts. This requires privilege to kernel mode instructions that are not available in ordinary processes.

In addition to the kernel, layer 1 also contains two more modules like the device drivers. The first one is the **clock task**; it resembles an I/O device driver as it interacts with the hardware that generates timing signals, but it only interacts with the kernel, it is not accessible like a disk or communications line driver.

The primary function of layer 1 is to provide a set of privileged kernel calls to drivers and servers above it. These calls include reading and writing I/O ports, copying data between address spaces. The **system task** is responsible for implementing these kernel calls. While the clock task and system task are compiled into the kernel's address space, they are scheduled as separate processes and have their own call tasks.

How layer 1 is written in code:

Most of the kernel and all the clock and system tasks are written in C, but a small amount of the kernel is written in assembly code. This part of the kernel that is written in assembly is done in that way to be able to handle interrupts, manage context switching between processes (like saving and restoring registers), handle inter-process communication, and manipulate the MMU (Memory Management Unit) hardware.

The assembly code handles the parts of the kernel that must interact directly with the hardware at a low level and so, cannot be expressed in C. The kernel also offers a set of kernel calls to allow the rest of the operating system to continue doing its work. These calls perform functions such as hooking handlers to interrupts, moving data between address spaces, and installing memory maps for the new processes.

The other three layers above the kernel could be considered as a single layer as the kernel treats all three of them in the same way. All three are limited to user mode instructions, and then they are scheduled to run by the kernel. None of them can access I/O ports directly. Also, none of them can access memory outside the segments allotted to it.
The real difference between each layer though, is the privileges (like the ability to make a kernel call). Processes in layer 2 have the most privileges, those in layer 3 have some privileges and those in layer 4 have no privileges at all.

Layer 2- Device Drivers:

The processes in layer 2 which are called device drivers, can request from the system to read data or write data to I/O ports on their behalf. A driver is needed for each device type (which include disks, printers, terminals, and network interfaces). If there are more I/O devices, a driver is needed for each one of them as well. Device drivers can also make other kernel calls, like requesting that newly read data be copied to the address space of a different process.

In this layer, resource management is also being done, with help from the kernel layer when privileged access to I/O ports or interrupt system is required.

The device drivers – which are typically started when the system is started - can also be started later.

Layer 3- Servers:

The third layer contains the servers, which are processes that provide useful services to the user process. There are two essential servers. The **process manager (PM),** this server carries out all the MINIX system calls that have to do with starting or stopping process execution, such as fork, exec, and exit. It also performs system calls related to signals, such as alarm and kill, which can alter the execution state of the process. This server is also responsible for managing memory, such as using the brk system call. The other server - **file system (FS)** - carries out all the file system calls, such as read, mount, and chdir.

In this layer, we go into further details about the difference between kernel calls and POSIX system calls and try to understand them better.

Kernel calls: They are low-level functions provided by the system task to allow the drivers and servers to do their work. For example, a typical kernel call could be reading a hardware I/O port.

POSIX system calls: They are high-level calls defined by the POSIX standard and are available to user programs in layer 4. Some of these calls include read, fork, and unlink.

User programs contain many POSIX calls but no kernel calls. On occasion, an error could be made in calling a kernel call a system call, as the methods used to make these calls happen to have a lot of similarities, and kernel calls can be considered a special subset of system calls.

Aside from PM and FS, layer 3 has other servers which perform functions that are specific to MINIX. The **information server (IS)** is used in handling jobs that include debugging and status information about other drivers and servers, something that is necessary in an operating system like MINIX, as it is used in experimentation. Another server is the **reincarnation server (RS)**, it starts and restarts device drivers that are not loaded into memory at the same time as the kernel. Particularly, if a driver fails during operation, the reincarnation server detects this failure, kills the driver if it is not already dead, and starts a fresh copy of the driver, making the system highly fault tolerant.

Servers are not able to do I/O directly, but they can communicate with drivers to request I/O. They can also communicate with the kernel using system tasks.

Another server is the **network server (inet),** this is mainly present in networked systems.

In layer 3, system call interpretation is done by the process manager and file system servers. The file system has been carefully designed as a file and could easily be removed to a remote machine with few changes.

The addition of more servers in the operating system does not require recompilation. The process manager and the file system can be supplemented with the network server and other servers by attaching additional servers as required when MINIX is started up.

To sum up both layer 2 and layer 3, device drivers and servers are compiled and stored on disk as ordinary executable files, but when properly started up, they are granted access to special privileges needed. A user program called service provides an interface to the reincarnation server which manages this. Even though, drivers and servers are independent processes, they differ from user process as they never terminate while the system is still up and running.

It is often referred to drivers and servers (layer 2 and layer 3) as system processes. It can be argued that system processes are part of the operating system, as they do not belong to a user, and almost all of them, if not all will be activated before the user even logs onto the system. Another difference between system processes and user processes is that the system processes have higher execution priority than servers, but not always. Execution priority in MINIX is assigned on a case-by-case basis; there is a possibility that a driver that services a slow device to be give lower priority than that of a server that must reply quickly.

Layer 4- User Processes:

Layer 4 contains all user processes – shells, editors, compilers, and user -written programs. User processes come and go with ease as a user log onto the system, does work, and logs out. A running system usually has some user processes that are started when the system is booted, and which run forever – like the process init. There is also a **daemon** present, this is a background process that is executed periodically or is always

waiting for some event, like the arrival of a packet from a network. In other words, the daemon could be described as a server that starts independently and runs as a user process. This daemon or multiple daemons are also likely to be running. The daemon could also have a higher priority that other ordinary user processes.

To better understand how the structure of MINIX works, we delve more into tasks and device drivers. In older versions of MINIX, device drivers were compiled together with the kernel, giving them all access to data structures that belong to the kernel and each other. The drivers also had the ability to access I/O ports directly. They were referred to as “tasks” to differentiate between them and pure independent user-space processes. In MINIX, device drivers were implemented in user-space. The only exception is the clock task, which is arguably not a device driver as drivers can be accessed through device files by user processes.

Benefits of micro-kernel:

1. It facilitates the extension of the operating system. All new services do not require modification of the kernel as they are added to the user space. But even so, the kernel sometimes need modification, but the changes tend to be few, because the micro-kernel is a smaller kernel.
2. It makes the resulting Operating system easy to port between hardware designs.
3. It provides higher security and reliability as most of the services are running as user processes. Therefore, if one service fails, the rest of the operating system functions normally and is untouched.
4. It gave a new meaning to modularity; it is so by breaking up the operating system into several independent user-level processes.

While the micro-kernel is very efficient, has many benefits, and is widely used, it still has a weak spot. The micro-kernel’s performance can suffer because of high system-function overhead. When communication occurs between two user-level services, messages are copied between the services, which happen to be in different address spaces. Also, the operating system may have to switch from one process to another in order to exchange messages. The overhead resulting from copying messages and process switching happens to be a large impediment to the growth of the micro-kernel-based operating systems. For example, in the case of layered micro-kernel organization, and the solution to this problem can be as easy as moving some layers from user space to kernel space and integrating them more closely.

Apple macOS

Brief History on MacOS:

Apple macOS (the original name, OS X, was changed in 2016 to match the naming scheme of other Apple products).MacOS is considered as fusion of macOS classic and NEXTSTEP, where its kernel became the core of Darwin The macOS is a UNIX operating system(as it is named Darwin, composed of kernel and XNU and runtime), where it makes use of the BSD codebase and XNU kernel and uses the apple open source code Darwin operating system.

The Apple Macintosh computer was arguably the first computer with a GUI designed for home users. The original Mac OS ran only on Apple computers and slowly was eclipsed by Microsoft Windows (starting with Version 1.0 in 1985), which was licensed to run on many different computers from a multitude of companies.

Work on Mach began in the mid 1980, where its architecture was monolithic kernel. The operating system was designed with the following three critical goals in mind:

1. Be a modern operating system that supports many memory models, as well as parallel and distributed computing.
2. Have a kernel that is simpler and easier to modify than 4.3 BSD
3. The Mach code was initially developed inside the 4.2 BSD kernel, so Emulate 4.3 BSD UNIX so that the executable files from a UNIX system can run correctly under Mach.

Through Release 2, Mach provided compatibility with the corresponding BSD systems by including much of BSD's code in the kernel and the new features and capabilities of mach made the kernels larger than the corresponding BSD kernels. Release 3 moved the BSD code outside the kernel, leaving a much smaller microkernel, where the kernel implements only Mach basic operations, and all UNIX-specific code has been evicted to run in user-mode servers. Excluding UNIX code from kernel allows the replacement of BSD with the simultaneous execution of multiple operating system interfaces on top of the microkernel. Throughout the upgraded versions of macOS the new features advertised are mostly user mode, kernel resources remained closed even though it maintained XNU. As form of security the system is hardened to not allow any access to the underlying UNIX APIs but the application's own, the layered structure helped in maintaining this security manner.

The architecture of the latest version of OS X is layered structure

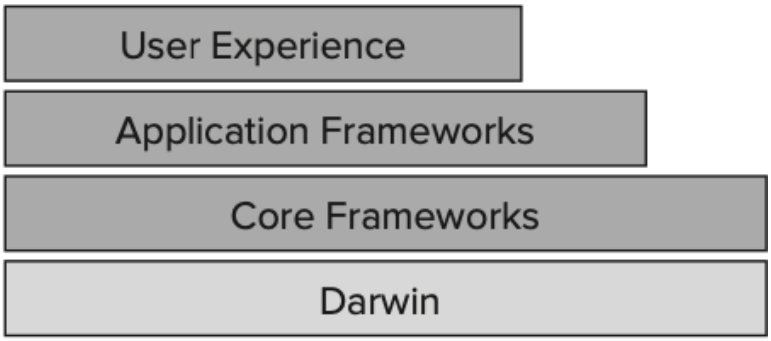


FIGURE 2-1: OS X and iOS architectural diagram

Figure 6 Layer Mac structure

1. The User Experience Layer (user interface)
2. The Application Framework Layer
3. The Core Frameworks (Graphics and Media layer)
4. Darwin: The OS core- kernel and UNIX shell environment

The User Experience Layer (user interface): the layer that defines the software interface that allows users to interact with computing devices.

The Application Framework Layer: the layers that that includes Cocoa and Cocoa Touch frame- works, providing an API for Objective-C and Swift programming languages.

The Core Frameworks: the layer that defines frameworks that support graphics and media, including QuickTime and OpenGL.

Darwin: The Apple code name for its open- source kernel, the OS core.

The layered architecture:

The operating system is made of number of layers, where the nth layer is the user interface and layer 0 is hardware. Each layer only inter-communicate with the above or below layer. This approach simplifies debugging and system verification, thus the implementation and design is simplified.

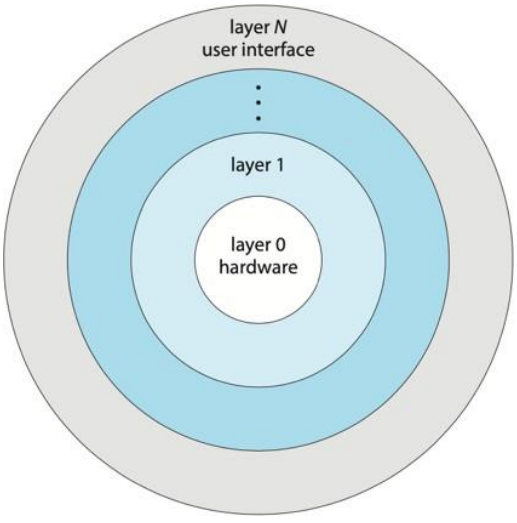


Figure 7 Layered Architecture

Internal Structure of MacOS:

A. Kernel

B. User Interface

C. API

D. Hardware

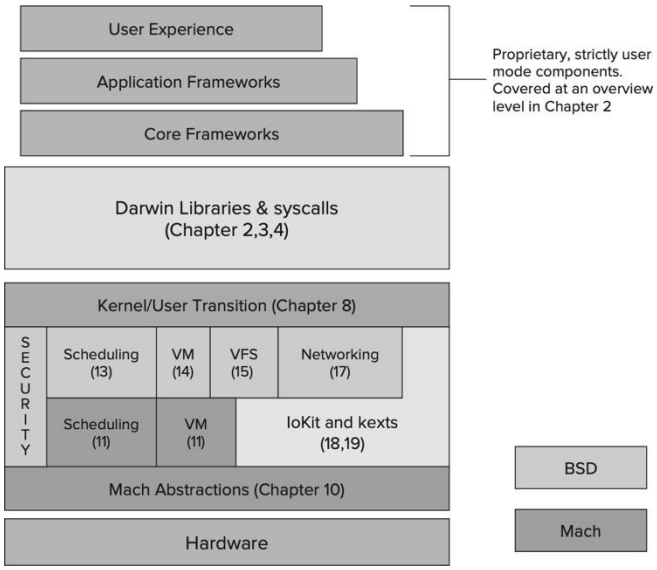


Figure 8Internal Structure

Kernel:

Kernel is the one part of the operating system that is always working where it handles every aspect of the operating system. The kernel provides a level of virtualization. This is accomplished by an API that deals with abstract objects like, virtual memory, network interfaces, and generic devices.

Kernel is also considered the process scheduler which determines when and for how long a process executes on a processor, not only this. But also, memory manager, which determines when and how memory is allocated to processes and what to do when main memory becomes full and I/O manager which services input and output requests from and to hardware devices, respectively. Moreover, interposes communication (IPC) manager, which allows processes to communicate with one another and file system manager, which organizes named collections of data on storage devices and provides an interface for accessing data on those devices.

The XNU kernel is in the lowest layer, Darwin, where the Darwin is a fully open sourced and serves as the foundation and low-level APIs for the rest of the system, where it is considered kernel environment. Kernels are subject to a more complicated, real-time, and hardware constrained environment.
The Darwin is not singled layer it is hybrid, where it has Mach and BSD with clean separation. Where it provides microkernel flexibility and performance of monolithic kernel.

Mach:

The kernel is modular and allows for pluggable Kernel Extensions to be dynamically loaded on demand. The core of XNU is Mach, where Mach is a system that was originally developed as a research project into creating a lightweight and efficient platform for operating systems. This results in Mach microkernel, which handles only the most primitive responsibilities of the operating system:

- Process and thread abstraction
- Virtual memory management
- Task scheduling
- Intercrosses communication and messaging

Mach limited APIs, they are discouraged by apple, but they are fundamental, any additional functionality must be implemented on top of it in the BSD layer.

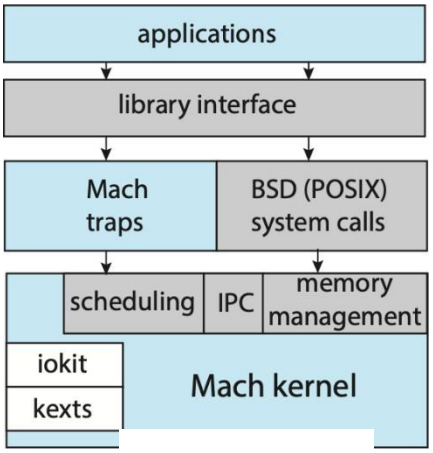


Figure 9 Mach

BSD Layer:

It's inseparable from the Mach, this layer presents a solid API that provides the POSIX (Portable Operating System Interface) compatibility. The BSD layer wraps the Mach kernel, but its native system calls are still accessible from user mode. XNU's BSD implementation is largely compatible with FreeBSD's, but does have some noteworthy changes

The BSC layer provide higher-level abstraction:

- Process Model
- Threading model and its synchronization primitives
- Users and groups
- The Network stack (BSD socket API)
- File System access
- Device access

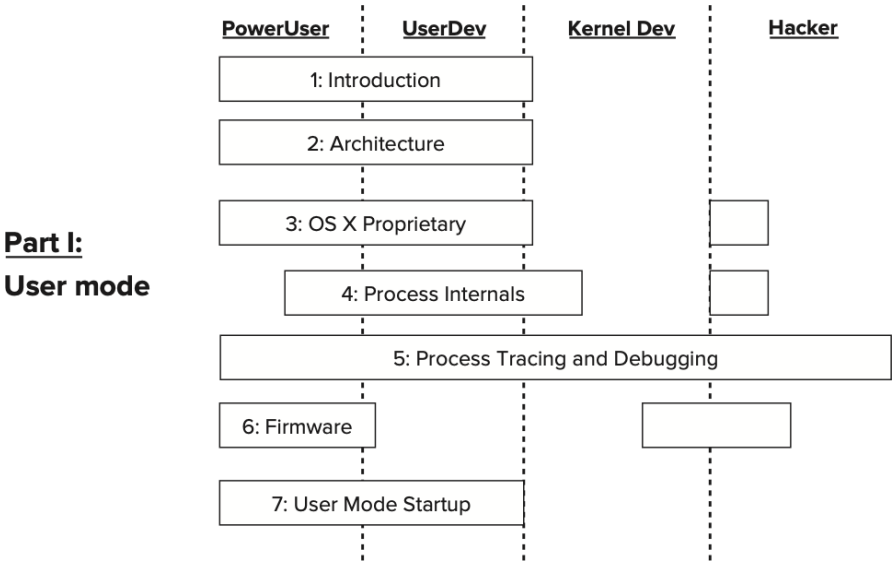


Figure 10 MachOS user mode

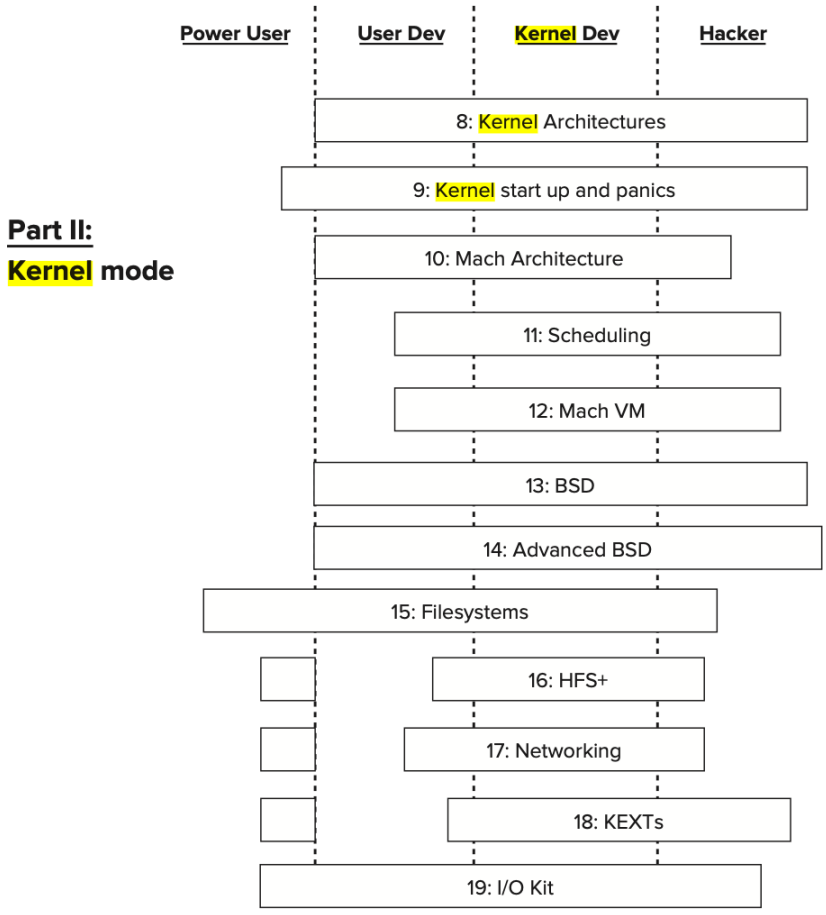


Figure 11 MachOS Kernel mode

The Monolithic kernel architecture:

- Monolithic is that every component of the operating system is contained in one place in the kernel
- can directly communicate with each other simply by function calls
- Switching from user mode to kernel mode is highly efficient

so direct intercommunication (controlled by the kernel) is highly efficient

The Microkernel Architecture:

- Most of the operating system components execute outside exported to external serers
- Only critical aspects are carried out by the kernel
- Communication occurs by message passing
- Benefits:

1. Exhibit a high degree of modularity, makes them extensible, portable and scalable
2. Microkernel is independent, any fails does not affect other components
3. Error free
4. Flexible and adaptivity

The Hybrid Kernels:

Like in this case the Darwin is, as it takes the advantages of both architectures. Where the innermost core of the kernel, supporting the lowest level services of scheduling, inter-process communication (IPC) and virtual memory, is self-contained, as would be a microkernel. All other services are implemented outside this core, though also in kernel mode and in the same memory space as the cores. Also, the whole operating system has layered architecture.

User Interface:

Increasingly, many users interact with mobile devices such as smartphones and tablets—devices that are replacing desktop and laptop computer systems for some users. These devices are typically connected to networks through cellular or other wireless technologies. Many devices also allow users to interact through a voice recognition interface, such as Apple's Siri.

The choice of whether to use a command-line or GUI interface is mostly one of personal preference. System administrators who manage computers and power users who have deep knowledge of a system frequently use the command-line interface. For them, it is more efficient, giving them faster access to the activities they need to perform.

The User Experience Layer, that debuted with cheetah, has been target for imitation and it influenced other GUI-based operating systems. The user interface contains several parts:

- Aqua:
- the familiar, distinctive GUI of OS X, its features translucent windows and graph- ics effects.
- is mouse-driven and designed for windowing
- is part of the Core Graphics frameworks buried deep within another framework, Application Services
- Quick Look:

is a feature that was introduced to enable a quick preview from inside the Finder, of various file types, it is an extensible architecture, allowing most of the work to be done by plugins.

- Spotlight:
- is the quick search technology that Apple introduced; it has been seamlessly integrated into Finder.
- The brain behind spotlight is an indexing server, mds, located in the Metadata framework, which is part of the system's core services.
- Accessibility options

The system's first user-mode process launched is responsible for starting the GUI, the main process that maintains the GUI is the Window Server. The same object-orientation was prevalent all throughout the operating system. The system offered frameworks and kits, which allowed for rapid GUI development using a rich object library the operating system now provides both an Aqua GUI and a command-line interface.

API:

The API specifies a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect.

The Mach APIs in the kernel, these APIs are largely hidden from view, with most applications using the much more popular BSD APIs. The Mach APIs are, nonetheless, critical for the system, and virtually all applications would break down if they were to be suddenly removed. The Mach APIs tends to be slower, its message-passing microkernel-

based architecture may be elegant, but it is hardly as effective as contemporary monolithic kernels. That resulted in removing Mach altogether and solidifying the kernel to be fully BSD.

Throughout the versions one of the significant changes is **Grand Central Dispatch**, it enabled multi-core programming through a central API. Other changes are that the APIs exposed by the kernel are provided through a common file system adaptation layer, called the Virtual File system Switch (VFS), VFS is a uniform interface for all file systems in the kernel, both UNIX based and foreign.

Frameworks are a lot like libraries but are unique to Apple's systems and provide a full runtime interface serve to hide the underlying system and library APIs. Apple keeps most frameworks in tightly closed source. This is because the frameworks are responsible for providing the unique look-and-feel, as well as other advanced features that are offered only by Apple's operating systems. The "traditional" libraries still exist in Apple's systems and, in fact, provide the basis on top of which the frameworks are implemented.

Top level framework in OS X is carbon and cocoa, they are in the Application Framework Layer. Where the carbon is the name given to the OS 9 legacy programming interfaces and many new interfaces have been added into. Cocoa is the preferred application programming environment, cocoa is a "fat" binary with all 3 architectures. Both are built on top of other frameworks and essentially serve as a wrapper.

System Calls These are entry points into predefined functions exported
by the kernel and accessible in user mode. The primary interface between processes and the operating system, providing a means to invoke services made available by the operating system. OS X system calls are unusual in that the system exports two distinct "personalities" — that of Mach and that of POSIX. POSIX is a standard API that defines, System call prototypes and system call numbers, other than that another subset of APIs — Mach Traps, which remains OS X specific.

Hardware:

Apple has done its utmost to keep OS X closed, so this strips down the operating system to allow developers only the functionality Apple deems as “safe” or “recommended,” rather than allow full use of the hardware, which is comparable to any decent desktop computer. But these limitations are artificial at its core. There BSD files that represent hardware devices on the system.

Derwin is open-source code that does not include many of the defining macOS elements, so cannot run Mac applications, but it does support known features. Apart from the macOS elements Darwin support 64-bit x86-64 variant of the intel x68 processors used in Intel-based Macs. So, Darwin supports the POSIX API and large number of programs written for various other UNIX-like systems.

Windows

A brief history on Windows

In the mid-1980s, Microsoft and IBM cooperated to develop the **OS/2 operating system**, which was written in assembly language for single-processor Intel 80286 systems. In 1988, Microsoft decided to end the joint effort with IBM and develop its own "new technology" (or NT) portable operating system to support both the OS/2 and POSIX application programming interfaces (APIs). In October 1988, Dave Cutler, the architect of the DEC VAX/VMS operating system, was hired and given the charter of building Microsoft's new operating system.

Originally, the team planned to use the OS/2 API as NT's native environ- ment, but during development, NT was changed to use a new 32-bit Windows API (called Win32), based on the popular 16-bit API used in Windows 3.0. The first versions of NT were Windows NT 3.1 and Windows NT 3.1 Advanced Server. (At that time, 16-bit Windows was at Version 3.1.) Windows NT Ver- sion 4.0 adopted the Windows 95 user interface and incorporated Internet web-server and web-browser software. In addition, user-interface routines and all graphics code were moved into the kernel to improve performance (with the side effect of decreased system reliability and significant loss of security). Although previous versions of NT had been ported to other microprocessor architectures (including a brief 64-bit port to Alpha AXP 64), the Windows 2000 version, released in February 2000, supported only IA-32-compatible pro- cessors due to marketplace factors. Windows 2000 incorporated significant changes. It added Active Directory (an X.500-based directory service), better networking and laptop support, support for plug-and-play devices, a dis- tributed file system, and support for more processors and more memory.

Windows Structure:

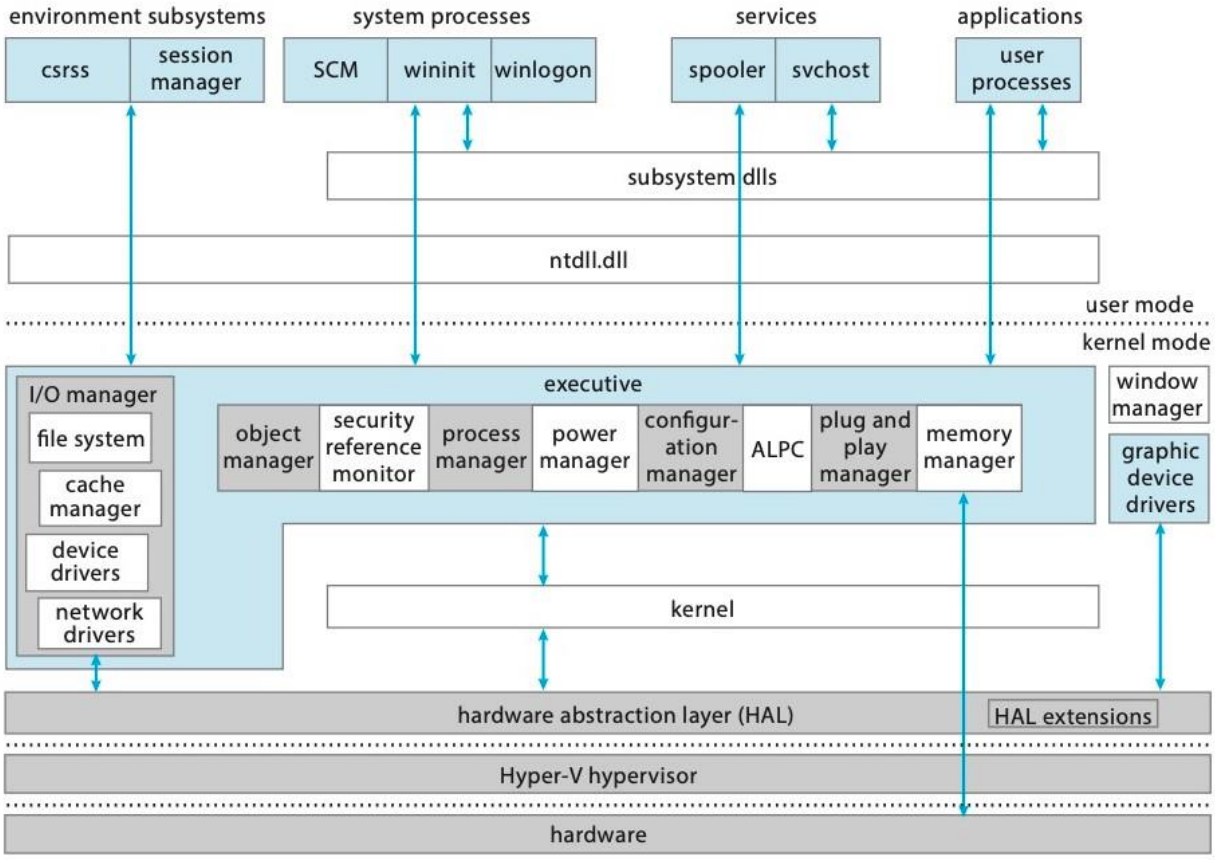


Figure 21.1 Windows block diagram.

Figure 12 Windows structure

The architecture of Windows is a layered system of modules operating at specific privilege levels, as shown earlier in Figure 21.1. By default, these privilege levels are first implemented by the processor (providing a "vertical" privilege isolation between user mode and kernel mode). Windows 10 can also use its Hyper-V hypervisor to provide an orthogonal (logically independent) security model through **Virtual Trust Levels** (**VTLs**). When users enable this feature, the system operates in a Virtual Secure Mode (VSM). In this mode, the layered privileged system now has two implementations, one called the **Normal World**, or VTL 0, and one called the **Secure World**, or VTL 1. Within each of these worlds, we find a user mode and a kernel mode.

Let's look at this structure in somewhat more detail.

- **In the Normal World, in kernelmodeare**
  (1) the HAL and its extensions
  (2) the kernel and its executive, which load drivers and DLL dependencies.
  In user mode are a collection of system processes, the Win32 environment sub-system, and various services.

- **In the Secure World, if VSM is enabled, are a secure kernel and executive (within which a secure micro-HAL is embedded). A collection of isolated Trustlets (discussed later) run in secure user mode.**

- **Finally, the bottom most layer in Secure World runs in a special processor mode (called, for example, VMX Root Mode on Intel processors), which contains the Hyper-V hypervisor component, which uses hardware virtu- alization to construct the Normal-to-Secure-World boundary. (The user-to- kernel boundary is provided by the CPU natively.)**

One of the chief advantages of this type of architecture is that interactions between modules, and between privilege levels, are kept simple, and that iso- lation needs and security needs are not necessarily conflated through privilege. For example, a secure, protected component that stores passwords can itself be unprivileged. In the past, operating-system designers chose to meet isolation needs by making the secure component highly privileged, but this results in a net loss for the security of the system when this component is compromised.

Hyper-V Hypervisor

The hypervisor is the first component initialized on a system with VSM enabled, which happens as soon as the user enables the Hyper-V component. It is used both to provide hardware virtualization features for running separate virtual machines and to provide the hardware's Second Level Address Translation (SLAT) functionality (discussed shortly).

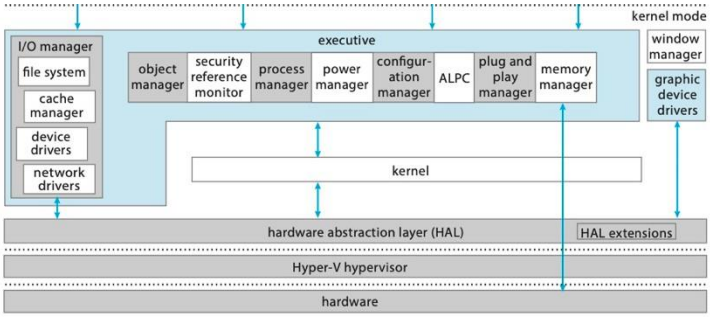


Figure 13 Windows block diagram kernal mode

The hypervisor uses a CPU-specific virtualization extension, such as AMD's Pacifica (SVMX) or Intel's Vanderpool (VT-x), to intercept any interrupt, exception, memory access, instruction, port, or register access that it chooses and deny, modify, or redirect the effect, source, or destination of the operation. It also provides a **hypercall** interface, which enables it to communicate with the kernel in VTL 0, the secure kernel in VTL 1, and all other running virtual machine kernels and secure kernels.

Secure Kernel

The secure kernel acts as the kernel-mode environment of isolated (VTL 1) user- mode Trustlet applications (applications that implement parts of the Windows security model). It provides the same system-call interface that the kernel does, so that all interrupts, exceptions, and attempts to enter kernel mode from a VTL 1 Trustlet result in entering the secure kernel instead. However, the secure kernel is not involved in context switching, thread scheduling, memory man- agement, interprocess-communication, or any of the other standard kernel tasks. Additionally, no kernel-mode drivers are present in VTL 1. In an attempt to reduce the attack surface of the Secure World, these complex implementa- tions remain the responsibility of Normal World components. Thus, the secure kernel acts as a type of "proxy kernel" that hands off the management of its resources, paging, scheduling, and more, to the regular kernel services in VTL 0. This does make the Secure World vulnerable to denial-of-service attacks, but that is a reasonable tradeoff of the security design, which values data privacy and integrity over service guarantees.

Hardware-Abstraction Layer

The HAL is the layer of software that hides hardware chipset differences from upper levels of the operating system. The HAL exports a virtual hard- ware interface that is used by the kernel dispatcher, the executive, and the device drivers. Only a single version of each device driver is required for each CPU architecture, no matter what support chips might be present. Device drivers map devices and access them directly, but the chipset-specific details of mapping memory, configuring I/O buses, setting up DMA, and coping with motherboard-specific facilities are all provided by the HAL interfaces.

Kernel

The kernel layer of Windows has the following main responsibilities: thread scheduling and context switching, low-level processor synchronization, inter- rupt and exception handling, and switching between user mode and kernel mode through the system-call interface. Additionally, the kernel layer imple- ments the initial code that takes over from the boot loader, formalizing the tran- sition into the Windows operating system. It also implements the initial code that safely crashes the kernel in case of an unexpected ex- ception, assertion, or other inconsistency. The kernel is mostly implemented in the C language, using assembly language only when absolutely necessary to interface with the lowest level of the hardware architecture and when direct register access is needed.

Dispatcher

The dispatcher provides the foundation for the executive and the subsystems. Most of the dispatcher is never paged out of memory, and its execution is never preempted. Its main responsibilities are thread scheduling and context switching, implementation of synchronization primitives, timer management, software interrupts (asynchronous and deferred procedure calls), interproces- sor interrupts (IPIs) and exception dispatching. It also manages hardware and software interrupt prioritization under the system of **inter- rupt request levels**

(**IRQLs**).

Switching Between User-Mode and Kernel-Mode Threads

What the programmer thinks of as a thread in traditional Windows is actually a thread with two modes of execution: a **user-mode thread** (**UT**) and a **kernel- mode thread** (**KT**). The thread has two stacks, one for UT execution and the other for KT. A UT requests a system service by executing an instruction that causes a trap to kernel mode. The kernel layer runs a trap handler that switches UT stack to its KT sister and changes CPU mode to kernel. When thread in KT mode has completed its kernel execution and is ready to switch back to the corresponding UT, the kernel layer is called to make the switch to the UT, which continues its execution in user mode. The KT switch also happens when an interrupt occurs.

The dispatcher is not a separate thread running in the kernel. Rather, the dispatcher code is executed by the KT component of a UT thread. A thread goes into kernel mode in the same circumstances that, in other operating systems, cause a kernel thread to be called. These same circumstances will cause the KT to run through the dispatcher code after its other operations, determining which thread to run next on the current core.

Threads

Like many other modern operating systems, Windows uses threads as the key schedulable unit of executable code, with processes serving as containers of threads. Therefore, each process must have at least one thread, and each thread has its own scheduling state, including actual priority, processor affinity, and CPU usage information.

There are eight possible thread states: **initializing, ready, deferred- ready, standby, running, waiting, transition, and terminated. ready indicates that the thread is waiting to execute,** while deferred-ready indicates that the thread has been selected to run on a specific processor but has not yet been scheduled. A thread is running when it is executing on a processor core. It runs until it is preempted by a higher-priority thread, until it terminates, until its allotted execution time (quantum) ends, or until it w[illegible] an event signaling I/O completion. If a thread is preempting another thread on a different processor, it is placed in the standby state on that processor, which means it is the next thread to run.

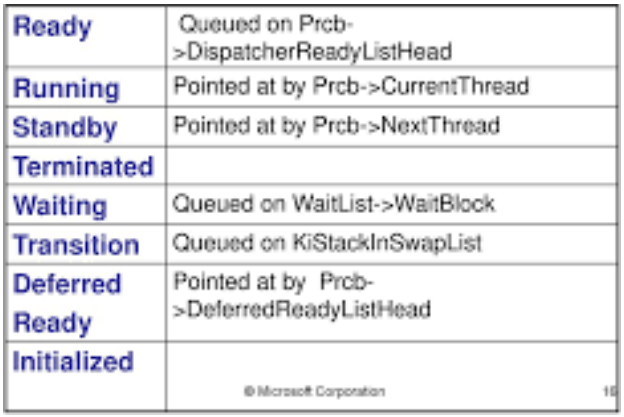


Figure 14 Threads states

Preemption is instantaneous—the current thread does not get a chance to finish its quantum. Therefore, the processor sends a software interrupt—in this case, a **deferred procedure call** (**DPC**)—to signal to the other processor that a thread is in the standby state and should be immediately picked up for execution. Interestingly, a thread in the standby state can itself be preempted if yet another processor finds an even higher-priority thread to run in this processor. At that point, the new higher-priority thread will go to standby, and the previous thread will go to the ready state. A thread is in the waiting state when it is waiting for a dispatcher object to be signaled. A thread is in the transition state while it waits for resources necessary for execution; for example, it may be waiting for its kernel stack to be paged in from secondary storage. A thread enters the terminated state when it finishes execution, and a thread begins in the initializing state as it is being created, before becoming ready for the first time.

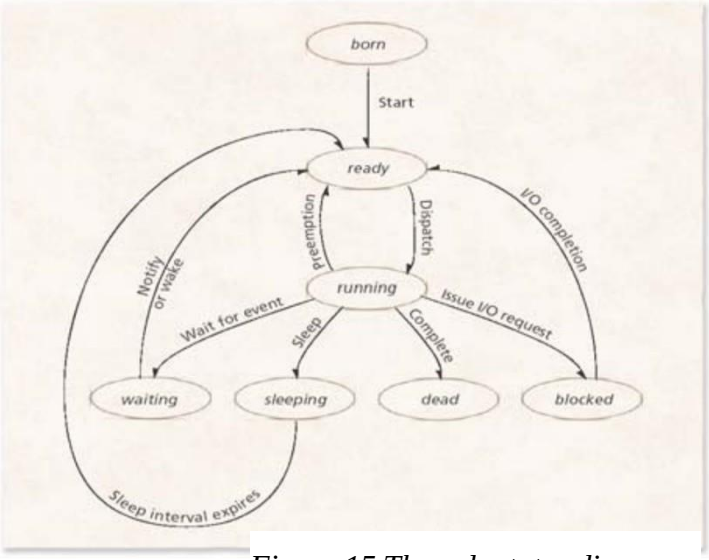


Figure 15 Threads states diagram

Thread Scheduling

Scheduling occurs when a thread enters the ready or waiting state, when a thread terminates, or when an application changes a thread's processor affinity. As we have seen throughout the text, a thread could become ready at any time. If a higher-priority thread becomes ready while a lower-priority thread is running, the lower-priority thread is preempted immediately. This preemption gives the higher-priority thread instant access to the CPU, without waiting on the lower-priority thread's quantum to complete.

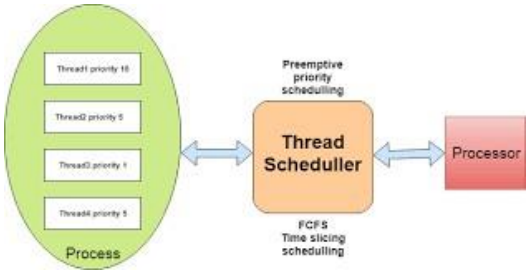


Figure 16 Threads Scheduler

It is the lower-priority thread itself, performing some event that caused it to operate in the dispatcher, that wakes up the waiting thread and immedi- ately context-switches to it while placing itself back in the ready state. This model essentially distributes the scheduling logic throughout dozens of Win- dows kernel functions and makes each currently running thread behave as the scheduling entity. In contrast, other operating systems rely

on an external “scheduler thread” triggered periodically based on a timer. The advantage of the Windows approach is latency reduction, with the cost of added overhead inside every I/O and other state-changing operation, which causes the current thread to perform scheduler work.

Windows is not a hard-real-time operating system, however, because it does not guarantee that any thread, even the highest-priority one, will start to execute within a particular time limit or have a guaranteed period of execution. Threads are blocked indefinitely while DPCs and **interrupt service routines** (**ISRs**) are running (as further discussed below), and they can be preempted at any time by a higher-priority thread or be forced to round-robin with another thread of equal priority at quantum end.

Traditionally, the Windows scheduler uses sampling to measure CPU utilization by threads. The system timer fires periodically, and the timer inter- rupt handler takes note of what thread is currently scheduled and whether it is executing in user or kernel mode when the interrupt occurred. This sam- pling technique originally came about because either the CPU did not have a high-resolution clock or the clock was too expensive or unreliable to access frequently. Although efficient, sampling is inaccurate and leads to anomalies such as charging the entire duration of the clock (15 milliseconds) to the currently running thread (or DPC or ISR). Therefore, the system ends up completely ignoring some number of milliseconds—say, 14.999—that could have been spent idle, running other threads, running other DPCs and ISRs, or a combi- nation of all of these operations. Additionally, because quantum is measured based on clock ticks, this causes the premature round-robin selection of a new thread, even though the current thread may have run for only a fraction of the quantum.

Linux

A brief history on Linux:

Linux is used primarily for process, memory, and device-driver support for hardware and has been expanded to include power management. The Android runtime environment includes a core set of libraries as well as the Dalvik virtual machine. Software designers for Android devices develop applications in the Java language. However, rather than using the standard Java API, Google has designed a separate Android API for Java development. The Java class files are first compiled to Java bytecode and then translated into an executable file that runs on the Dalvik virtual machine. The Dalvik virtual machine was designed for Android and is optimized for mobile devices with limited memory and CPU processing capabilities. The set of libraries available for Android applications includes frameworks for developing web browsers (webkit), database support (SQLite), and multimedia. The lab library is like the standard C library but is much smaller and has been designed for the slower CPUs that characterize mobile devices.

Linux treat the command interpreter as a special program that is running when a process is initiated or when a user first logs on (on interactive systems). On systems with multiple command interpreters to choose from, the interpreters are known as shells. For example, on Linux systems, a user may choose among several different shells, including the C shell, Bourne-Again shell, Korn shell, and others. Third-party shells and free user-written shells are also available. Most shells provide similar functionality, and a user's choice of which shell to use is generally based on personal preference.

Shell:

Although Linux systems have a graphical user interface, most programmers and sophisticated users still prefer a command-line interface, called the shell. Often, they start one or more shell windows from the graphical user interface and just work in them. The shell command-line interface is much faster to use, more powerful, easily extensible, and does not give the user RSI from having to use a mouse all the time. Below we will briefly describe the bash shell (bash). It is heavily based on the original UNIX shell, Bourne shell (written by Steve Bourne, then at Bell Labs). Its name is an acronym for Bourne Again SHell. Many other shells are also in use (ksh, csh, etc.), but bash is the default shell in most Linux systems.

When the shell starts up, it initializes itself, then types a prompt character, often a percent or dollar sign, on the screen and waits for the user to type a command line. When the user types a command line, the shell extracts the first word from it, where word here means a run of characters delimited by a space or tab. It then assumes this word is the name of a program to be run, searches for this program, and if it finds it, runs the program. The shell then suspends itself until the program terminates, at which time it tries to read the next command. What is important here is simply the observation that the shell is an ordinary user program. All it needs is the ability to read from the keyboard and write to the monitor and the power to execute other programs. Commands may take arguments, which are passed to the called program as character strings. For example, the command line cp src dest invokes the cp program with two arguments, src and dest. This program interprets the first one to be the name of an existing file. It makes a copy of this file and calls the copy dest.

Not all arguments are file names. In head –20 file the first argument, –20, tells head to print the first 20 lines of file, instead of the default number of lines, 10. Arguments that control the operation of a command or specify an optional value are called flags, and by convention are indicated with a dash. The dash is required to avoid ambiguity, because the command head 20 file is perfectly legal, and tells head to first print the initial 10 lines of a file called 20, and then print the initial 10 lines of a second file called file. Most Linux commands accept multiple flags and arguments. To make it easy to specify multiple file names, the shell accepts magic characters, sometimes called wild cards. An asterisk, for example, matches all possible strings, so ls *.c tells ls to list all the files whose name ends in.c. If files named x.c, y.c, and z.c all exist, the above command is equivalent to typing ls x.c y.c z.c Another wild card is the question mark, which matches any one character. A list of characters inside square brackets selects any of them, so ls [ape]* lists all files beginning with ‘‘a’’, ‘‘p’’, or ‘‘e’’.

A program like the shell does not have to open the terminal (keyboard and monitor) in order to read from it or write to it. Instead, when it (or any other program) starts up, it automatically has access to a file called standard input (for reading), a file called standard output (for writing normal output), and a file called standard error (for writing error messages). Normally, all three defaults to the terminal, so that reads from standard input come from the keyboard and writes to standard output or standard error go to the screen.

Many Linux programs read from standard input and write to standard output as the default. For example, sort invokes the sort of program, which reads lines from the terminal (until the user types a CTRL-D, to indicate end of file), sorts them alphabetically, and writes the result to the screen. It is also possible to redirect standard input and standard output, as that is often useful. The syntax for redirecting standard input uses a less-than symbol (). It is permitted to redirect both in the same command. For example, the command sort out causes sort to take its input from the file in and write its output to the file out. Since standard error has not been redirected, any error messages go to the screen. A program that reads its input from standard input, does some processing on it, and writes its output to standard output is called a filter. Consider the following command line consisting of three separate commands: sort temp; head –30 foo Here all the lines containing the string ‘‘ter’’ in all the files ending in .t are written to standard output, where they are sorted. The first 20 of these are selected out by head, which passes them to tail, which writes the last five (i.e., lines 16 to 20 in the sorted list) to foo. This is an example of how Linux provides basic building blocks (numerous filters), each of which does one job, along with a mechanism for them to be put together in almost limitless ways. Linux is a general-purpose multiprogramming system. A single user can run several programs at once, each as a separate process. The shell syntax for running a process in the background is to follow its command with an ampersand. Thus wc –l b & runs the word-count program, wc, to count the number of lines (–l flag) in its input, a, writing the result to b, but does it in the background. As soon as the command has been typed, the shell types the prompt and is ready to accept and handle the next command. Pipelines can also be put in the background, for example, by sort

Utilities:

The command-line (shell) user interface to Linux consists of many standard utility programs. Roughly speaking, these programs can be divided into six categories, as follows:

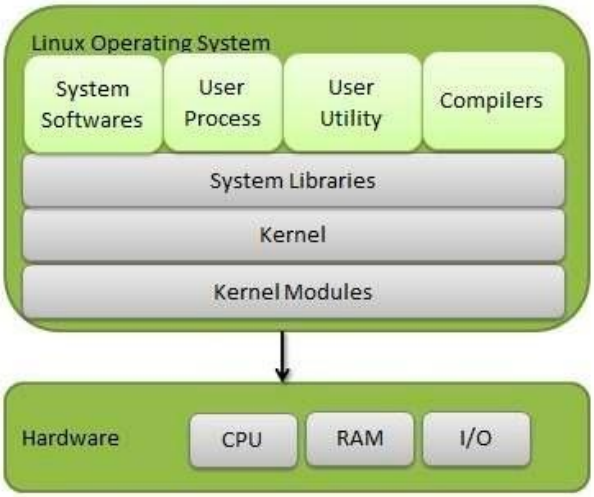


Figure 17 Linux Structure

1. File and directory manipulation commands.

Starting with file and directory manipulation. cp a b copies file a to b, leaving the original file intact

removes the original. In effect, it moves the file rather than really making a copy in the usual sense. Several files can be concatenated using cat, which reads each of its input files and copies them all to standard output, one after another. Files can be removed by the rm command. The chmod command allows the owner to change the rights bits to modify access permissions. Directories can be created with mkdir and removed with rmdir.

To see a list of the files in a directory, ls can be used. It has a vast number of flags to control how much detail about each file is shown (e.g., size, owner, group, creation date), to determine the sort order (e.g., alphabetical, by time of last modification, reversed), to specify the layout on the screen, and much more.

2. Filters.

We have already seen several filters: grep extracts lines containing a given pattern from standard input or one or more input files; sort sorts its input and writes it on standard output; head extracts the initial lines of its input; tail extracts the final lines of its input. Other filters defined by 1003.2 are cut and paste, which allow columns of text to be cut and pasted into files; od, which converts its (usually binary) input to ASCII text, in octal, decimal, or hexadecimal; tr, which does character translation (e.g., lowercase to uppercase), and pr, which formats output for the printer, including options to include running heads, page numbers, and so on.

3. Program development tools, such as editors and compilers.

Compilers and programming tools include gcc, which calls the C compiler, and ar, which collects library procedures into archive files. Another important tool is made, which is used to maintain large programs whose source code consists of multiple files. Typically, some of these are header files, which contain type, variable, macro, and other declarations. Source files often include these using a special include directive. This way, two or more source files can share the same declarations. However, if a header file is modified, it is necessary to find all the source files that depend on it and recompile them. The function of make is to keep track of which file depends on which header, and similar things, and arrange for all the necessary compilations to occur automatically. Nearly all Linux programs, except the smallest ones, are set up to be compiled with make, along with a short description of each. All Linux systems have them and many more.

4. Text processing.

They are lightweight and have modular functionality. Even though these text manipulation tools differ in complexity and functionality, they come in handy in an environment where the graphical user interface isn't available.

Examples:

Sort / uniq / comm / cmp / diff / tr / sed / awk / perl / cut / paste / column / pr

5. System administration.

Linux system administration is a process of setting up, configuring, and managing a computer system in a Linux environment. System administration involves creating a user account, taking reports, performing backup, updating configuration files, documentation, and performing recovery actions. The user who manages the server, fixes configuration issues, recommends new software updates, and updates document is the system administrator.

6. Miscellaneous.

Finally, there are miscellaneous calls such as NtFlushBuffersFile. Like the UNIX sync call, it forces file-system data to be written back to disk. NtCancelIoFile cancels outstanding I/O requests for a particular file, and NtDeviceIoControlFile implements ioctl operations for devices. The list of operations is much longer. There are system calls for deleting files by name, and for querying the attributes of a specific file—but these are just wrappers around the other I/O manager operations we have listed and did not really need to be implemented as separate system calls. There are also system calls for dealing with I/O completion ports, a queuing facility in Windows that helps multithreaded servers make efficient use of asynchronous I/O operations by readying threads by demand and reducing the number of context switches required to service I/O on dedicated threads.

The POSIX 1003.1-2008 standard specifies the syntax and semantics of about 150 of these, primarily in the first three categories. The idea of standardizing them is to make it possible for anyone to write shell scripts that use these programs and work on all Linux systems. In addition to these standard utilities, there are many application programs as well, of course, such as Web browsers, media players, image viewers, office suites, games, and so on.

Kernel structure:

The kernel sits directly on the hardware and enables interactions with I/O devices and the memory management unit and controls CPU access to them. At the lowest level, it contains interrupt handlers, which are the primary way for interacting with devices, and the low-level dispatching mechanism. This dispatching occurs when an interrupt happens. The low-level code here stops the running process, saves its state in the kernel process structures, and starts the appropriate driver. Process dispatching also happens when the kernel completes some operations, and it is time to start up a user process again. The dispatching code is in assembler and is quite distinct from scheduling. Next, we divide the various kernel subsystems into three main components.

The I/O component contains all kernel pieces responsible for interacting with devices and performing network and storage I/O operations. At the highest level, the I/O operations are all integrated under a VFS (Virtual File System) layer. That is, at the top level, performing a read operation on a file, whether it is in memory or on disk, is the same as performing a read operation to retrieve a character from a terminal input. At the lowest level, all I/O operations pass through some device driver.

All Linux drivers are classified as either character-device drivers or block-device drivers, the main difference being that seeks, and random accesses are allowed on block devices and not on character devices. Technically, network devices are really character devices, but they are handled somewhat differently, so that it is probably clearer to separate them.

Above the device-driver level, the kernel code is different for each device type. Character devices may be used in two different ways. Some programs, such as visual editors like vi and emacs, want every keystroke as it is hit. Raw terminal (tty) I/O makes this possible. Other software, such as the shell, is line oriented, allowing users to edit the whole line before hitting ENTER to send it to the program. In this case the character stream from the terminal device is passed through a so-called line discipline, and appropriate formatting is applied. Networking software is often modular, with different devices and protocols supported. The layer above the network drivers handles a kind of routing function, making sure that the right packet goes to the right device or protocol handler.

Most Linux systems contain the full functionality of a hardware router within the kernel, although the performance is less than that of a hardware router. Above the router code is the actual protocol stack, including IP and TCP, but also many additional protocols. Overlaying all the network is the socket interface, which allows programs to create sockets for networks and protocols, getting back a file descriptor for each socket to use later. On top of the disk drivers is the I/O scheduler, which is responsible for ordering and issuing disk-operation requests in a way that tries to conserve wasteful disk head movement or to meet some other system policy. At the very top of the block-device column are the file systems.

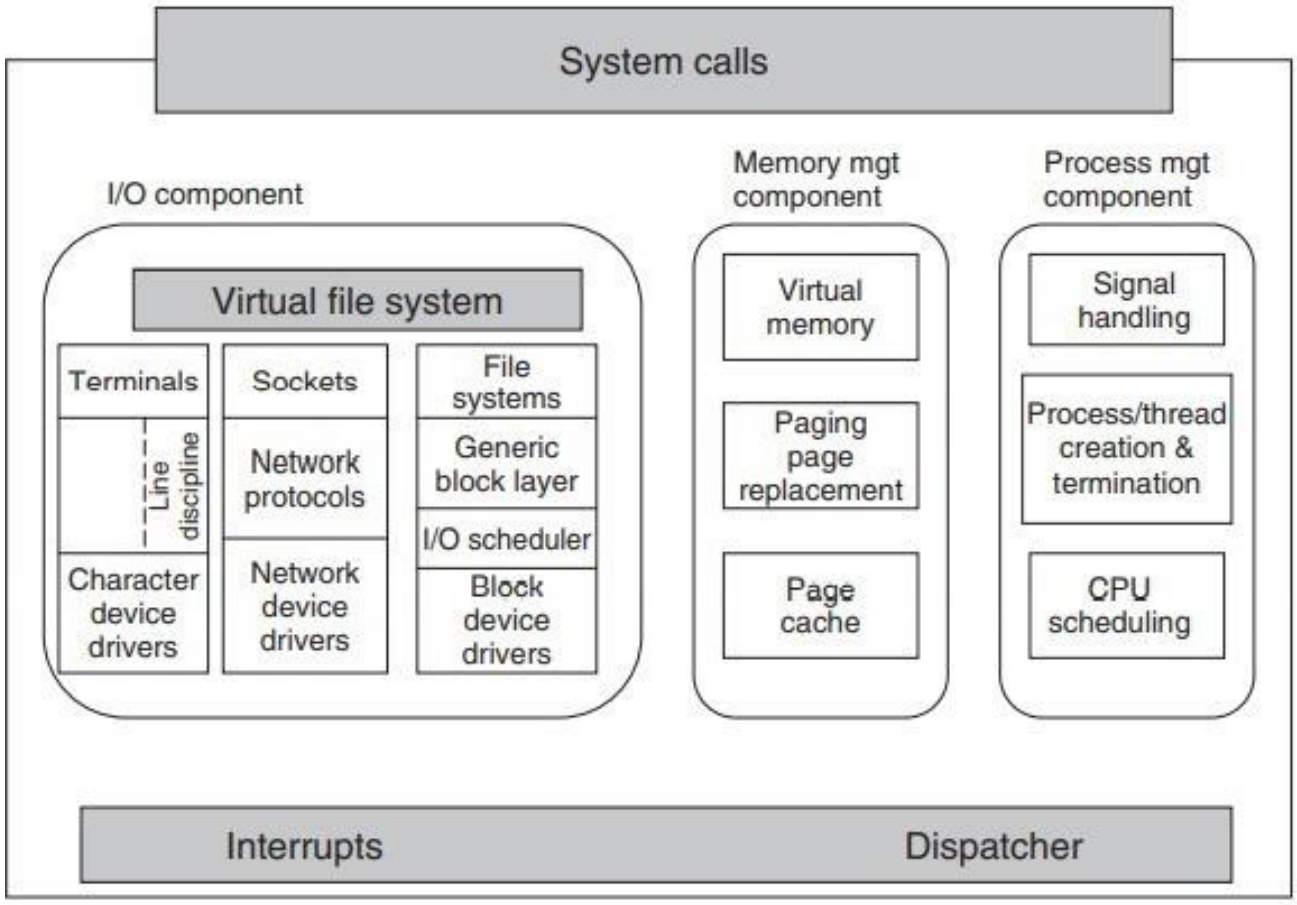


Figure 18Kernal

Linux may, and in fact does, have multiple file systems coexisting concurrently. In order to hide the gruesome architectural differences of various hardware devices from the file system implementation, a generic block-device layer provides an abstraction used by all

file systems. To the right in Fig. 10-3 are the other two key components of the Linux kernel.

These are responsible for the memory and process management tasks. Memory-management tasks include maintaining the virtual to physical-memory mappings, maintaining a cache of recently accessed pages and implementing a good page-replacement policy, and on-demand bringing in new pages of needed code and data into memory. The key responsibility of the process-management component is the creation and termination of processes. It also includes the process scheduler, which chooses which process or, rather, thread to run next., the Linux kernel treats both processes and threads simply as executable entities and will schedule them based on a global scheduling policy. Finally, code for signal handling also belongs to this component. While the three components are represented separately in the figure, they are highly interdependent. File systems typically access files through the block devices. However, in order to hide the large latencies of disk accesses, files are copied into the page cache in main memory. Some files may even be dynamically created and may have only an in-memory representation, such as files providing some run-time resource usage information. In addition, the virtual memory system may rely on a disk partition or in-file swap area to back up parts of the main memory when it needs to free up certain pages, and therefore relies on the I/O component. Numerous other interdependencies exist. In addition to the static in-kernel components, Linux supports dynamically loadable modules. These modules can be used to add or replace the default device drivers, file system, networking, or other kernel codes. The modules are not shown in Fig. 10-3. Finally, at the very top is the system call interface into the kernel. All system calls come here, causing a trap which switches the execution from user mode into protected kernel mode and passes control to one of the kernel components described above.

Scheduling Process in Minix3

Minix3 is a microkernel operating system that contains 4 layers and uses a multilevel queue scheduling system.
The bottom layer is for kernel mode and the three upper layers are for user mode.
The user mode implementation contains Process Manager (PM) and Scheduler (SCHED).
The scheduling process happens in the user mode which allows having multiple running schedulers.
There are 16 queues of runnable processes present in Minix3 and the scheduler maintains these queues, one per priority level.
Layer 1 is the highest priority level, and it contains the clock and the system tasks
Device Drivers in second layer gets a lower priority.
Server processes in third layer gets a lower priority than both clock and Device Drivers.
User processes has less priority than any of system processes and are all initially equal.
Lowest priority queue (Level 16) is set to be used by only idle processes
User processes' priority can be raised or lowered by the nice command.

IMPORTANT TERMS:

Completion Time: The time the process completes execution
Arrival Time: The time of the process arrival**.**
Waiting Time: The time the CPU has been waiting to be assigned the CPU and it equals Turnaround Time – Burst Time.
Turnaround Time: Amount of time to execute process starting from its arrival time till its completion.
Completion Time – Arrival Time OR
Waiting Time +Burst Time.
Response Time: amount of time a process takes from when a request was submitted until the first response. (Equals first time a process is assigned to CPU – Arrival Time).
Two level scheduling (CPU Scheduler and memory Scheduler) is common in Minix3.

Scheduling (CPU Scheduling) Processes Comparative Analysis:

Round Robin (assumes that all processes are equally important):

- It is a preemptive scheduling algorithm that makes a fair CPU time sharing and assumes that all processes are equally important.
- Each process is given a fixed execution time called quantum.
- When the time quantum finishes, if the process still didn't finish its execution time, the CPU is preempted from it and it goes to the end of the ready queue to be assigned the CPU again later or it goes to the terminal(end) state if it finished execution.
- The process in the ready queue after the previous preempted process is given the CPU for the specified time quantum and so on.
- Context switching is used to save the process's state when CPU is preempted.
- This algorithm allows multiple processes to run in parallel by making each process run for a small-time quantum.
- Schedular needs to maintain a list of all processes in running state whereas when a process finishes its quantum it is put at the end of the queue(list).
- Starvation free algorithm as all processes gets a fair share of CPU.

Advantages of Round Robin Scheduling:

- Gives the best Average response Time.
- Processes aren't exposed to starvation as each process gets equal time quantum.
- Each process is rescheduled after finishing the time quantum.
- Context Switching helps in saving the preempted processes states.

Disadvantages of Round Robin Scheduling:

- Has large waiting and response time.
- Spends a lot of time doing context switches.
- Will be a time-consuming scheduling algorithm if the process is given small time quantum thus higher context switching overhead.

Some Notes:

The number of context switches decreases while the response time increases by increasing the time quantum.

The number of context switches increases while the response time decreases by decreasing the time quantum (This lowers the CPU efficiency).

Conclusion:

Large time quantum will turn Round Robin to FCFS and making it to small increases the context switch overhead thus it must be in between.

Priority CPU Scheduling:

- Each process is given a priority and processes' execution happens with respect to the priority mentioned (Higher priority processes run first).
- To prevent high-priority processes from running indefinitely at each clock tick the schedular decreases the priority of some high priority processes to avoid some processes' starvation.
- FCFS is applied to incoming processes with the same priority.
- This priority can be (internally defined) decided based on number of open files, time limits, ratio of I/O burst to CPU burst, and memory requirements.
- Some processes are highly I/O bound (spend most time doing I/O operations. The CPU should be given immediately to this process when this process requires the CPU to allow it to start its next I/O request and then proceed in parallel with other running processes. Making I/O bound process wait long time for CPU will make it occupy memory for unnecessary time. To solve this problem, an algorithm is invented whereas the priority of I/O bound processes is set to 1/f (f is fraction of last processes used quantum).
- Minix3 puts I/O drivers and servers (memory manager, file system, and network) in the highest priority classes and the initial priority of each task or service is defined at compile time (slow device I/O processes may be given lower priority than fast I/O device processes).
- User processes, meanwhile, have lower priority than system components, but all priorities can change during execution.

Priority Scheduling is divided into two types:

Preemptive:

When a process with higher priority than the process currently running arrives, the CPU is preempted from this currently running process and given to the higher priority process.

Non-Preemptive:

The currently running process must finish its execution even if a higher priority process arrives to the system and the scheduler applies this higher priority process to the head of ready queue to be executed after the running process.

Advantages of Priority Scheduling:

- The importance of each incoming process is predefined
- Easy to use

Disadvantages of Priority Scheduling:

- May lead to starvation if a higher priority process has long execution time.
- If the system crashes, all low-priority processes will be lost

Aging can be used to solve the starvation of lower-priority processes by increasing their priority based on their waiting time.

Comparison between Round Robin and Priority:

	Priority Scheduling – Non pre-emptive	Priority Scheduling – Pre-emptive	Round Robin Scheduling
Model	Non pre-emptive	Pre-emptive	Pre-emptive
Criteria for Scheduling	Priority	Priority	**Time Quantum** & Arrival time (Generally from 10-100millisec)
Implementation.	Queue	Queue	Queue
No. of context switches	Low	High	Depends on time quantum If tq is high, no. of cs= low If tq is low, no. of cs= high
Average waiting time	Low priority processes starve.		In between
Avg turn around time	Depends on priority		In between
Throughput	Low		Depends on time quantum.
———	Equal priority processes scheduled in FCFS manner. Priority can be given internally or externally.		All process given same priority. If tq is very high, behaves as FCFS.

Figure 19 Comparison between Round Robin & Priority

Shortest Job First Scheduling (SJF):

It works on process with shortest burst time so the process's burst time should be known prior the process's execution.
It is optimal when all processes arrive at the same time.
Consist of two types:

- Non-Preemptive.
- Preemptive.

Non-Preemptive SJF :

In case the arrival time for all processes is the same:
The schedular takes the process with the shortest time and executes it without the CPU preemption of the process then takes the next shortest job and so on.
This approach gets an average waiting time way less than FCFS.
In case the arrival time for all processes is different:
This may lead to starvation as the process with the shorter burst time may be in the ready queue waiting for a process with a longer/longest burst time to finish execution without the process CPU preemption.
This starvation can be solved by aging (increasing some processes priority to be handled to CPU for execution).

Preemptive SJF:

All processes are put in the ready queue when arriving but when a process with a shorter burst time arrives the CPU is preempted from the currently running process and given to the process with the shorter burst time.
It has a shorter average waiting time than non-preemptive SJF.

Advantages of SJF Scheduling:

- Gives the minimal average waiting time for a process as jobs with shorter burst time are executed first.
- Better used when the processes' burst time is known in advance.
- Optimal considering average turnaround time.

Disadvantages of SJF Scheduling:

- May cause starvation for processes with shorter burst time.
- Hard because the coming processes' burst time must be predicted prior to execution.
- Can't be implemented at level of short-term CPU scheduling.

- May cause a major increase in the turnaround time.

Multi-Level Feedback Queue Scheduling:

Most general CPU scheduling algorithm and the processes in this algorithm are assigned to specific queues and allowed to move between queues.
It is used for separating processes with different characteristics of CPU burst.
Process that uses too much CPU time will be moved to lower priority queue.
If a process waits too long in lower priority queue, its priority will be increased, and it will be moved to a higher priority queue(aging).

Advantages of Multi-Level Feedback Queue Scheduling:

- Helps in response time reduction.
- Prevents starvation by increasing the priority of processes (moving process to higher priority) that have been waiting for so long because they have lower priority.
- Allows processes to move between different queues.
- Low scheduling overhead

Disadvantages of Multi-Level Feedback Queue Scheduling:

- Complex.
- CPU overheads because of moving processes between queues.
- Requires selecting values for all parameters to define the best scheduler.

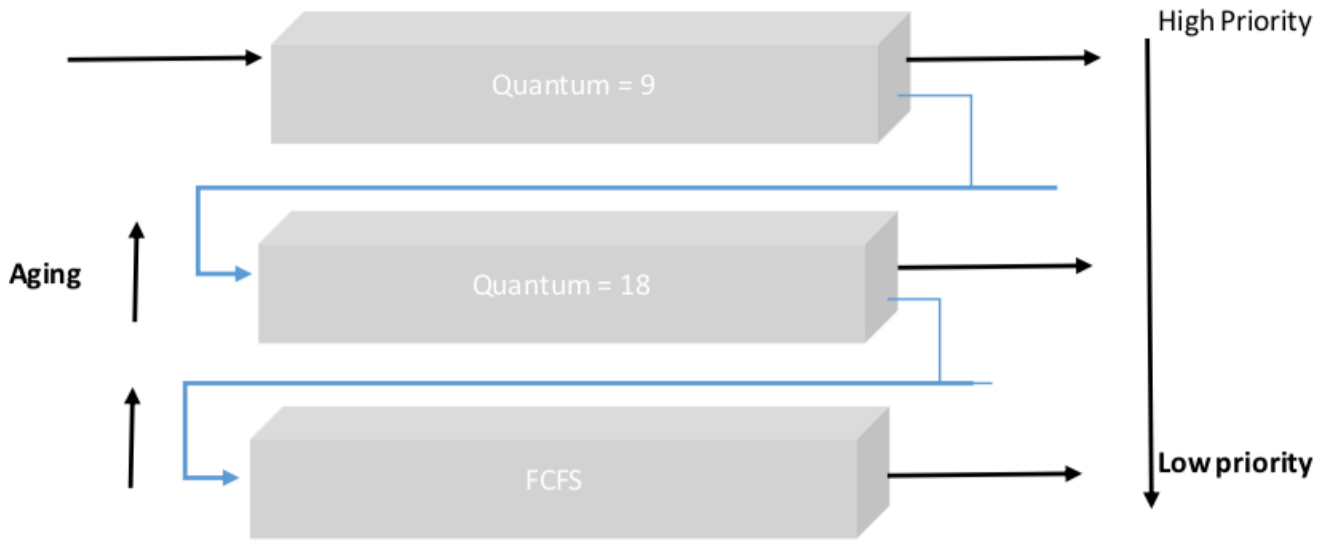


Figure 20 Queue Scheduling

Comparison between different Scheduling Algorithms:

TABLE I
COMPARISON OF VARIOUS SCHEDULING ALGORITHMS

Sr. No.	Parameters	FCFS Algorithm	SJF Algorithm	Priority Algorithm	R-R Algorithm	Multilevel Queue Algorithm	Multile Feedback Algorith
1	Preemption	This scheduling algorithm is non preemptive.	This scheduling algorithm is preemptive.	This scheduling algorithm is also preemptive.	This scheduling algorithm is also preemptive.	This scheduling algorithm is also preemptive.	This schedu algorithm is preemptive.
2	Complexity	This is simplest scheduling algorithm.	This algorithm is difficult to understand and code.	This algorithm is also difficult to understand.	In this scheduling algorithm, performance heavily depends upon the size of time quantum.	This algorithm is difficult to understand and code.	This algorit difficult to understand code and its performanc depends up size of time quantum.
3	Allocation	In this, it allocates the CPU in the order in which the process arrives.	In this, the CPU is allocated to the process with least CPU burst time.	It is based on the priority. The higher priority job can run first.	In this, the CPU is allocated in the order in which the process arrives but for fixed time slice.	In this, the CPU is allocated to the process which resides in the higher priority queue.	In this also, CPU is allo to the proce higher prior queue.
4	Waiting Time	In this, the average waiting time is large.	In this, the average waiting time is small as compared to FCFS scheduling algorithm.	In this, the average waiting time is small as compared to FCFS scheduling algorithm.	In this, the average waiting time is large as compared to all the three scheduling algorithms.	In this, the average waiting time is small as compared to FCFS scheduling algorithm.	In this, the average wai time is smal compared t FCFS sched algorithm.

Figure 21Comparison between different Scheduling Algorithms

Page Replacement Algorithms

Any operating system needs a way to manage the device's memory to provide the balance between efficiency in memory space and speed in random access. There are many implementations of memory management, one of which is paging. Paging is a memory management scheme that permits a process's physical address space to be non-contiguous. Paging avoids external fragmentation and the associated need for compaction, two problems that plague contiguous memory allocation. (Abraham Silberschatz). Additionally, the main memory is also divided into blocks of memory called frames and ideally, each frame is the same size as one page to insure almost no wasted space. The way paging is implemented differs from one Operating System to the other mainly in the algorithm of replacing an existing page in memory with a new one.

We are concerned in this report with two:

- LRU Algorithm
- FIFO algorithm

LRU (Last Recently Used) Algorithm

LRU is arguably the most deployed in practice. One example application is the popular Squid Web caching software. When LRU must evict a page from its cache, it chooses to evict the least recently requested page. LRU exploits temporal locality in request sequences and the recency property which states that recently requested objects are more likely to be requested next than objects that have not been requested recently. (E. Cohen, 2002)

FIFO (First in First Out) Algorithm

In the first-in-first-out page replacement policy, when a page is needed, the page that has been in memory for the longest period is chosen to replace. The rationale is that a page that has only recently been swapped in will have a higher probability of being used again soon. However, a frequently used page still gets swapped out when it gets to be old enough, even though it will have to be brought in again immediately (Gupta, 2014)

Comparative Analysis of LRU and FIFO algorithms

To compare algorithms, first, it is necessary to understand the behavior of real processes when interacting with the memory. In the article "Comparison of Cache Page Replacement Techniques to Enhance Cache Memory Performance", simulations were run to view how processes access memory. The simulations were some programs written in

plain C in an MS-DOS environment, and finally, a graph was generated to showcase the number of page faults relative to the number of frames.
The results for the FIFO Page replacement for three programs (bzip.txt, swim.txt, gcc.txt) were as follows:

No. of frames	Page faults for bzip.txt	Page faults for swim.txt	Page faults for gcc.txt
3	719	1397	1826
7	579	1131	1415
15	472	832	1121
31	378	319	928
63	329	125	858
127	278	82	828
255	244	82	803
511	244	82	575
1023	244	82	571

Figure 22 FIFO page replacement

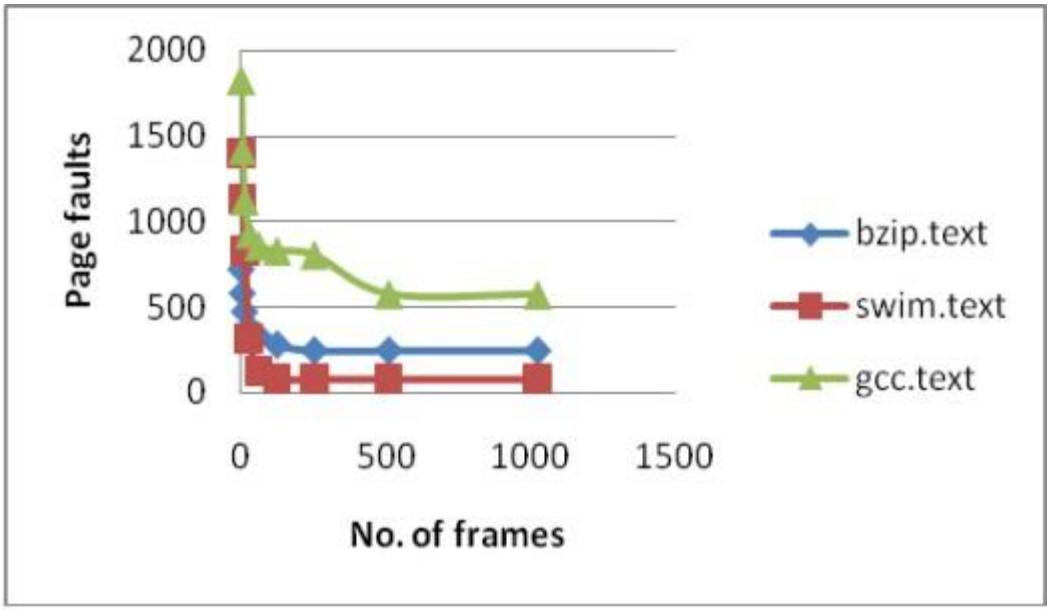


Figure 2 (for FIFO associative mapping)

Figure 23 FIFO associative mapping

(Gupta, 2014)
And for the LRU Algorithm the results were as follows:

Figure 4 (for LRU associative mapping)

No. of frames	Page faults for bzip.txt	Page faults for swim.txt	Page faults for gcc.txt
3	682	1143	1737
7	571	995	1310
15	457	660	1054
31	378	263	938
63	335	179	888
127	287	117	830
255	257	91	752
511	247	84	630
1023	244	82	583

Figure 24 LRU associative mapping

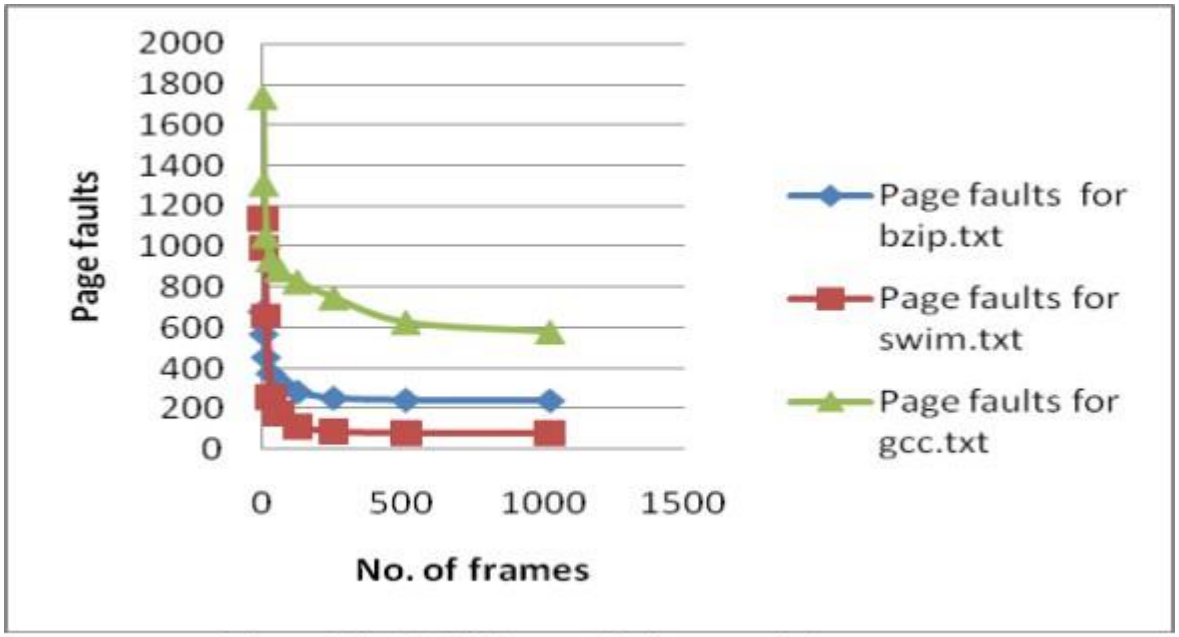


Figure 5 (for LRU Set-associative mapping)

Figure 25LRU Set associative mapping

Simulation conclusions

For two different programs (swim.txt, bzip.txt), the number of page faults is very close to each other and at some frame sizes they are identical, However, the one program that shows a big variation is the gcc.txt. With the LRU algorithm, the number of page faults decreases in an exponential-decay-like manner with a larger frame size. On the other hand, the FIFO algorithm with the gcc.txt starts with a higher number of page faults and maintains that behavior as the number of frames increases.

From such data and from the natures of the FIFO algorithm and the LRU algorithm, it is reasonable to deduce that the gcc.txt program depends a lot on irregular memory access patterns of memory compared to the other two programs. Given that conclusion, it is also reasonable to say that the LRU algorithm works best with programs that depend on irregular memory access patterns, and FIFO works best with programs with a sequential memory access nature.

Conclusion

The only way to determine which algorithm to implement in an Operating System is to analyze the behavior of the processes that will run on a said operating system when it comes to memory. Find which the most dominant behavior is and serve that the best you can.

Reference:

OPERATING SYSTEM CONCEPTS 10th EDITION by ABRAHAM SILBERSCHATZ and PETER BAER GALVIN and GREG GAGNE

MODERN OPERATING SYSTEMS 4th EDITION by ANDREW S. TANENBAUM HERBERT BOS Vrije Universiteit Amsterdam, The Netherlands

OPERATING SYSTEMS THREE EASY PIECES by REMZI H. ARPACI-DUSSEAUANDREA C. ARPACI-DUSSEAU UNIVERSITY OF WISCONSIN–MADISON

OPERATING SYSTEMS DESIGN AND IMPLEMENTATION 3rd Edition by ANDREW S. TANENBAUM Vrije Universiteit Amsterdam, The Netherlands and ALBERT S. WOODHULL Amherst, Massachusetts

Operating Systems 3rd edition by H. M. Deitel Deitel & Associates, Inc. And P. J. Deitel Deitel & Associates, Inc. And D. R. Choffnes Deitel & Associates, Inc.

Operating System Concepts by Abraham Silber Schatz, P. B. (n.d.). In P. B. Abraham Silberschatz, Operating System Concepts.

E. Cohen, H. K. (2002, April 19). Competitive Analysis of the LRFU Paging Algorithm1. Algorithmica, p. 1.
Gupta, R. (2014). Comparison of Cache Page Replacement Techniques to Enhance Cache Memory Performance. International Journal of Computer Applications.

Mac OS X and IOS Internals: To the Apple's Core Book by Jonathan Levin