Proof of Concept of Lottery Scheduling

Machida Hiroaki
himachid@bu.edu
CS575 A1 Operating Systems (Fall 2019)

Table of Contents

Chapter 1 Abstract

Lottery Scheduling can be implemented but there are some unexpected behaviors.

Problem:
Is Lottery Scheduling implementable by a usual developer?
What are challenges to implement?

Conclusion:
Lottery Scheduling can be implemented, but there are some unexpected behaviors.
The challenges range from setting up development environment to implementation itself.

Chapter 2 Implementation

Implementation: About Minix 3

Lottery Scheduling implemented on Minix 3, very simple open-source OS for educational purpose.

- Minix 3 was originally developed for educational purpose by Andrew S. Tanenbaum. (See Operating Systems Design and Implementation)

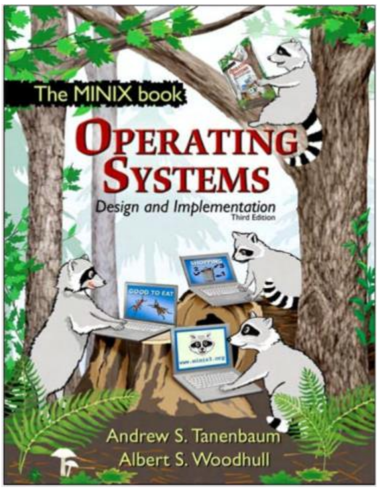


Figure 1. Operating Systems Design and Implementation

- Minix 3 is based on microkernel architecture, that has only about 4,000 lines of executable kernel code.
- Linus installed MINIX on his computer, wanted to read MINIX newsgroups, but some features lacking, so he wrote a program to do so, terminal driver, and disk driver, file system, then a primitive kernel. That was how Linux was born.

Implementation: Internal structure of Minix 3

Minix 3 is structured in four layers. Only processes in the bottom layer may use kernel mode instructions.

Internal layers:

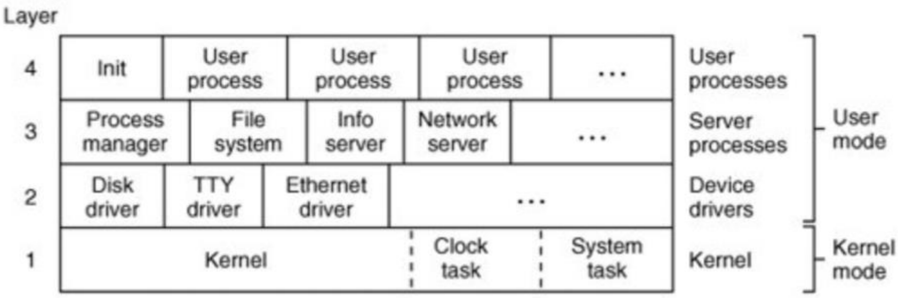


Figure 2. Minix 3 layers [1]

Minix 3 directory structure:

The main directories are as follows.
kernel/ layer 1 (scheduling, messages, clock and system tasks).
drivers/ layer 2 (device drivers for disk, console, printer, etc.).
servers/ layer 3 (process manager, file system, other servers).
src/lib/ source code for library procedures (e.g., open, read).
src/tools/ Makefile and scripts for building the MINIX 3 system.
src/boot/ the code for booting and installing MINIX 3.

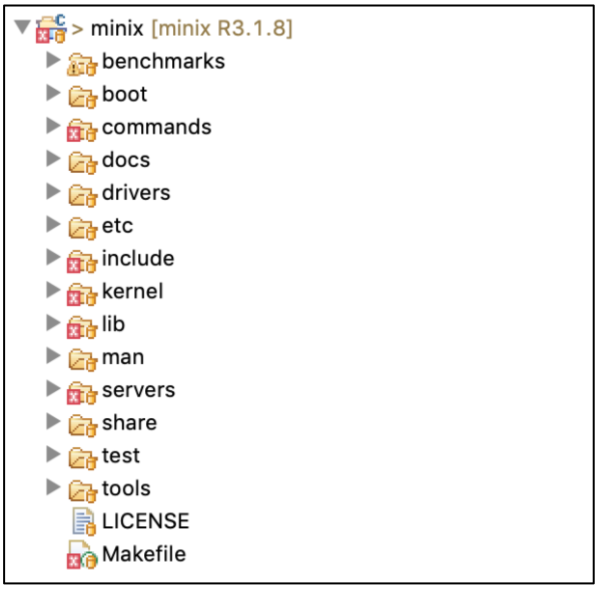


Figure 3. Minix 3 directory structure

[1] Operating Systems Design and Implementation (3rd Edition)

Implementation: Layer 1 (Kernel)

Layer 1 provides a set of privileged kernel calls.
The main directories are as follows.
arch/   Assembly files
system/   Kernel API
proc.c   process and message handling.

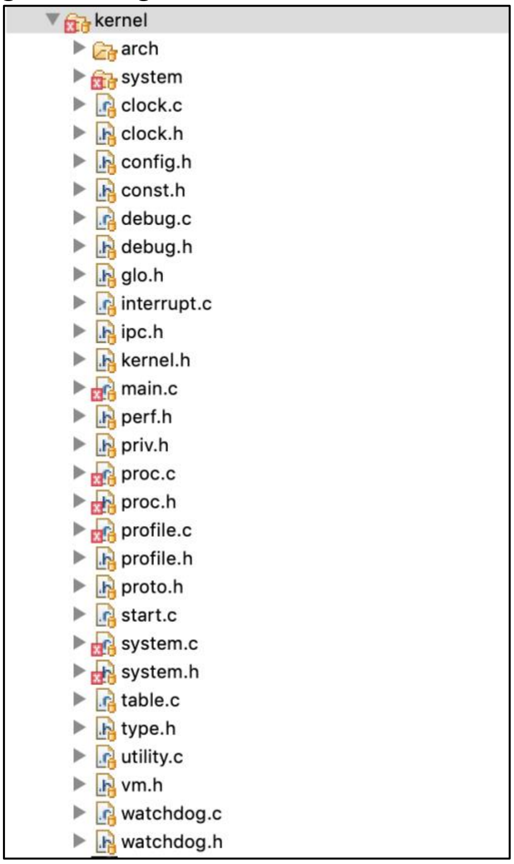


Figure 4. Layer 1 structure

Implementation: Layer 2 (Drivers)

Layer 2 can request read from or write to I/O ports.
The main directories are as follows.
audio/
floppy/
printer/
tty/ Console and keyboard driver

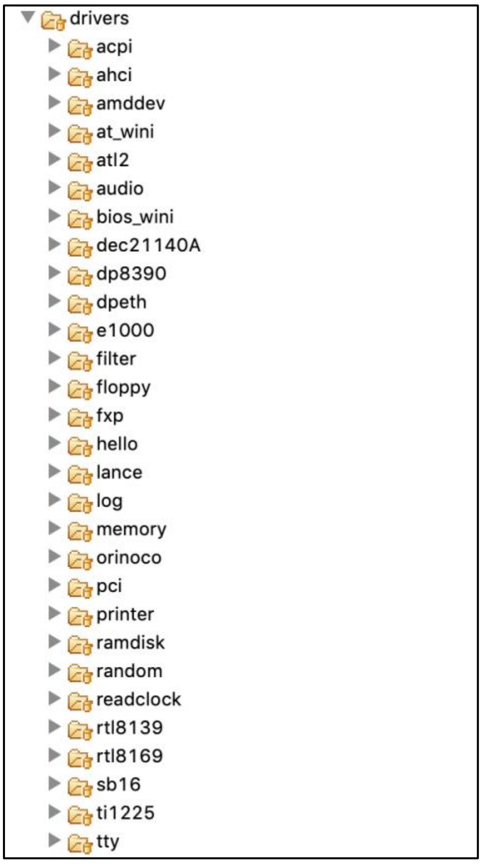


Figure 5. Layer 2 structure

Implementation: Layer 3 (Servers)

Layer 3 contains servers, processes that provide useful services to the user processes. The main directories are as follows.

pm/ Process Manager
sched/ Scheduler
vfs/ File system
vm/ Virtual Memory Manager
init/ Parent of all user processes

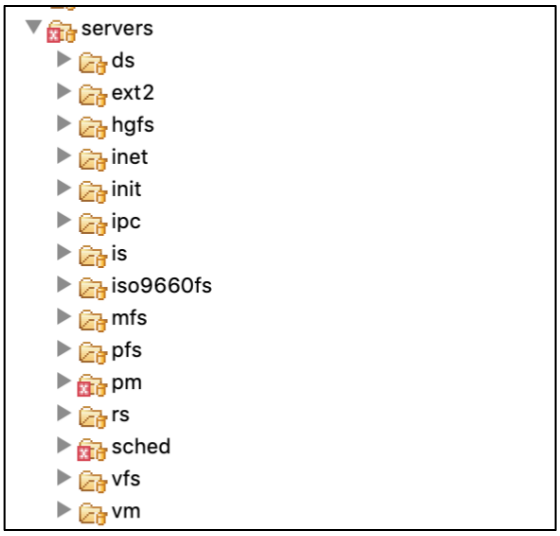


Figure 6. Layer 3 structure

Implementation: Message flow of changing niceness

At 3.1.7 release, Process Scheduler was separated from the kernel for easy customization. The flow of changing niceness.[2]

1. A user process invokes the setpriority system routine, which sends a message to PM
2. PM will look up the scheduler for this particular process and send a message to SCHED with the new value.
3. SCHED will validate this new nice level and may choose to alter this process' priority or quantum. In that case, the kernel will be notified with sys_schedule. Once SCHED replies to PM, PM will store the new nice value locally and can serve getpriority requests without contacting SCHED.

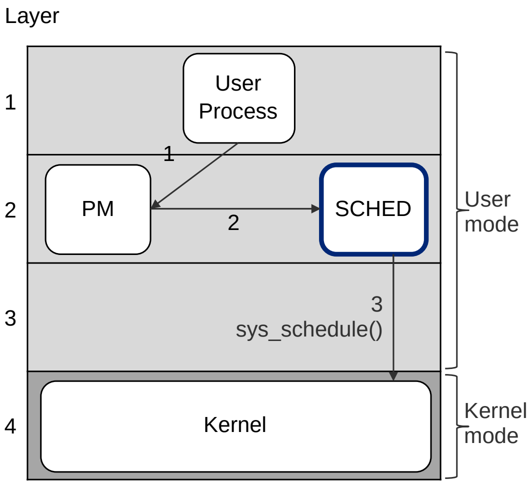


Figure 7. Message flow of changing niceness

[2] http://www.minix3.org/docs/scheduling/report.pdf

Implementation: SCHED overview

SCHED uses the following methods to handle messages from the Process Manager. The methods are as follows. [3]

do_noquantum()

Called on behalf of process' that run out of quantum

do_stop_scheduling()

Request to start scheduling a proc

do_start_scheduling()

Request to stop scheduling a proc

do_nice()

Request to change the nice level on a proc

Init_scheduling()

Called from main.c to set up/prepare scheduling

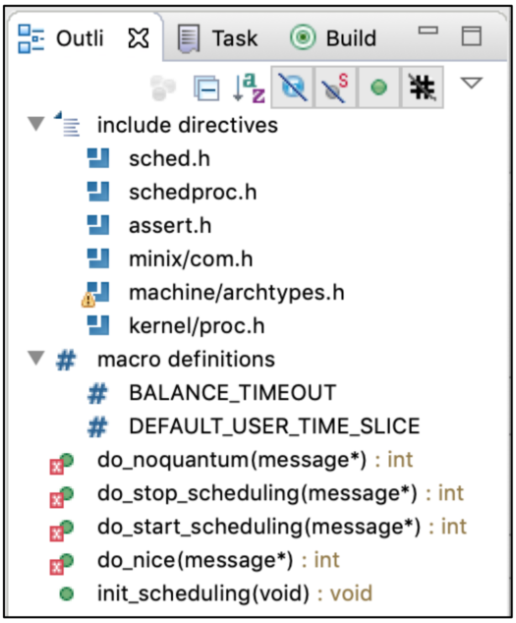


Figure 8. methods of sched/schedule.c

[3] https://github.com/Stichting-MINIX-Research-Foundation/minix/blob/R3.1.8/servers/sched/schedule.c

Implementation: Lottery Scheduling implementation

SCHED assigns as many tickets to processes as specified by nice, and does the lottery. Tickets are assigned to processes in do_nice().

```c
PUBLIC int do_nice(message *m_ptr)
{
    struct schedproc *rmp;
    int rv;
    int proc_nr_n;
    unsigned new_q, old_q, old_max_q;
    int t_tickets;
    int nice;

    printf("do_nice() called.\n");


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

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

    rmp = &schedproc[proc_nr_n];

    /* Store old values, in case we need to roll back the changes */
    old_q     = rmp->priority;
    old_max_q = rmp->max_priority;

    /* Increase tickets by nice */
    nice = m_ptr->SCHEDULING_MAXPRIO;
    rmp->tickets += nice;

```

Figure 9. sched/schedule.c#do_nice()

Lottery is done in lottery().

```c
PUBLIC int lottery()
{
    struct schedproc *rmp;
    int proc_nr;
    int rv;
    int winner;
    int total_tickets = 0;

    /* Add each participants tickets to the total pile of tickets. */
    for (proc_nr=0, rmp=schedproc; proc_nr < NR_PROCS; proc_nr++, rmp++) {
        if ((rmp->flags & IN_USE && rmp->is_sys_proc != 1) && rmp->priority == 15) {
            total_tickets += rmp->tickets;
        }
    }

    /* This is the basic loop logic for picking a winning process. */
    if (total_tickets > 0) {
        winner = rand() % total_tickets + 1; /* min of 1 ticket */
        for (proc_nr=0, rmp=schedproc; proc_nr < NR_PROCS; proc_nr++, rmp++) {
            if ((rmp->flags & IN_USE && rmp->is_sys_proc != 1) && rmp->priority == 15) {
                if (winner > 0) {
                    winner -= rmp->tickets;
                    if (winner <=0 && rmp->priority == 15) {
                        printf("Process:%d won the lottery! tickets:%d/%d priority:%d.\n",
                                rmp->endpoint, rmp->tickets, total_tickets, rmp->priority);
                        schedule_process(rmp);
                    }
                }
            }
        }
    }
    return OK;
}
```

Figure 10. sched/schedule.c#lottery()

Implementation is done by referencing minix-lottery-scheduler project. [4]

[4] https://github.com/HashTag-PurpleTeam/minix-lottery-scheduler

Chapter 3 Result

Result: Same niceness

Lottery Scheduling successfully assigns tickets to processes and do the lottery.

Performance test:

Longrun1.c is used for performance test. A not initialized variable v is 4 bit left shifted and deducted original v value for maxloop times, specified in the argument, and for each loopCount, also specified in the argument, the program print the progress. Implementation is done by referencing CMPS111-project project. [5]

```c
            argv[0], LOOP_COUNT_MIN, LOOP_COUNT_MAX, argv[2]);
        exit (-1);
    }
    /* max loops is third argument (if present) */
    if (argc == 4) {
        maxloops = atoi (argv[3]);
    } else {
        maxloops = 0;
    }

    /* Loop forever - use CTRL-C to exit the program */
    while (1) {
        /* This calculation is done to keep the value of v unpre
           the compiler can't calculate it in advance (even fron
           value of v and the loop count), it has to do the loop
        v = (v << 4) - v;
        if (++i == loopCount) {
          /* Exit if we've reached the maximum number of loops.
         0 (or negative), this'll never happen... */
          if (iteration == maxloops) {
        break;
          }
          n_print=n_print+1;
          if((n_print-o_print)>100){
              o_print=n_print;
            printf ("%s:%06d\n", idStr, iteration);
          }
          fflush (stdout);
          iteration += 1;
          i = 0;
          for(no_p=0;no_p<10000;no_p++)
              ;
        }
    }
    /* Print a value for v that's unpredictable so the compile
       optimize the loop away.  Note that this works because t
       can't tell in advance that it's not an infinite loop. *
    printf ("The final value of v is 0x%08x\n", v);
}
```

Figure 11. longrun1.c implementation

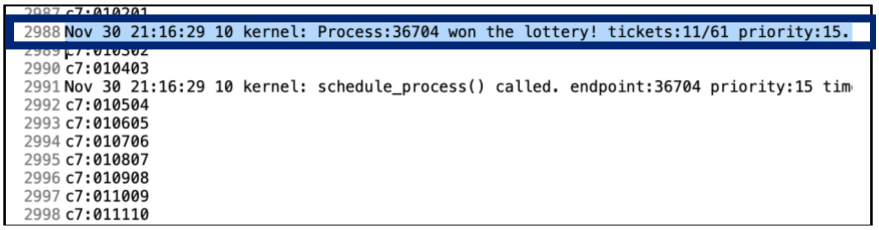
Command:

nice -n 10 ./a.out a7 100 30000 >> /var/log/messages & nice -n 10 ./a.out b7 100 30000 >> /var/log/messages & nice -n 10 ./a.out c7 100 30000 >> /var/log/messages & nice -n 10 ./a.out d7 100 30000 >> /var/log/messages

Log:

You can see the lottery done in /var/log/messages.

[5] https://github.com/freedombird9/CMPS111-project



```
2987 c7:010201
2988 Nov 30 21:16:29 10 kernel: Process:36704 won the lottery! tickets:11/61 priority:15.
2989 c7:010302
2990 c7:010403
2991 Nov 30 21:16:29 10 kernel: schedule_process() called. endpoint:36704 priority:15 tim
2992 c7:010504
2993 c7:010605
2994 c7:010706
2995 c7:010807
2996 c7:010908
2997 c7:011009
2998 c7:011110
```

Figure 12. /var/log/messages

Progress:

The progress is visualized. The x axis is the line number of log file and the y axis is the progress of each process.

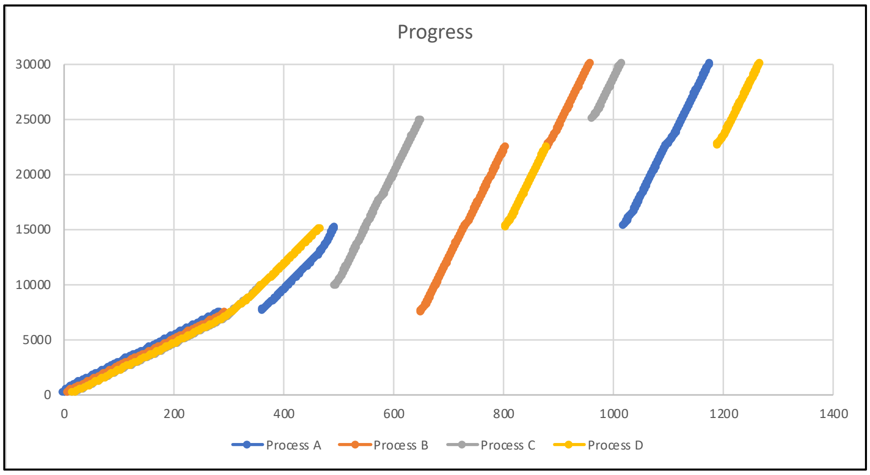


Figure 13. Progress

Result: Different niceness

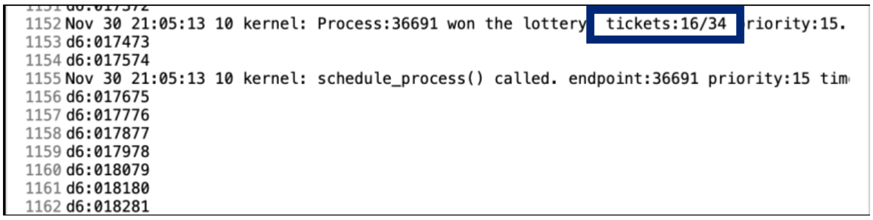
Also it is confirmed that the lottery is based on the number of tickets.

Command:

nice -n 1 ./a.out a6 100 30000 >> /var/log/messages & nice -n 1 ./a.out b6 100 30000 >> /var/log/messages & nice -n 1 ./a.out c6 100 30000 >> /var/log/messages & nice -n 15 ./a.out d6 100 30000 >> /var/log/messages

Log:

You can see the lottery done in /var/log/messages.



```
1152 Nov 30 21:05:13 10 kernel: Process:36691 won the lottery tickets:16/34 riority:15.
1153 d6:017473
1154 d6:017574
1155 Nov 30 21:05:13 10 kernel: schedule_process() called. endpoint:36691 priority:15 tim
1156 d6:017675
1157 d6:017776
1158 d6:017877
1159 d6:017978
1160 d6:018079
1161 d6:018180
1162 d6:018281
```

Figure 14. /var/log/messages

Progress:

The progress is visualized. The x axis is the line number of log file and the y axis is the progress of each process.

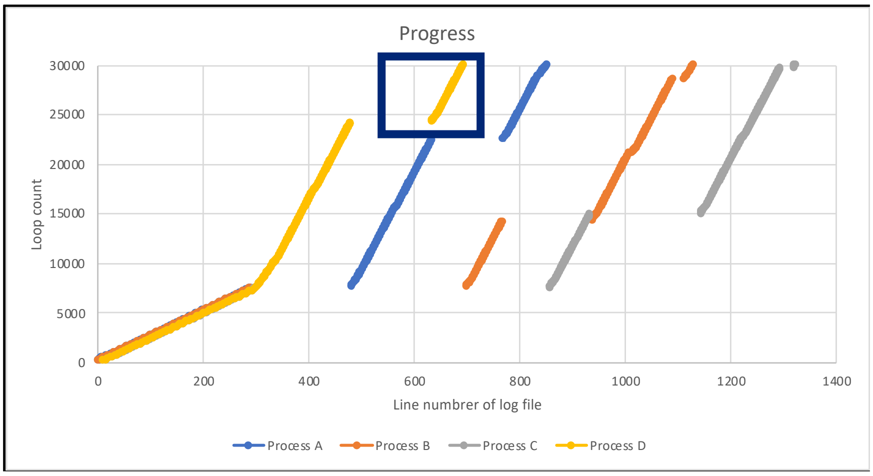


Figure 15. Progress

Chapter 4 Challenges

There were several challenges to implement and test lottery scheduling.

Challenge 1: Choose OS

Challenge:

Needed to choose an open source operating system to implement.

OS	Comment	Result
Minix	✓ Very light, easy to build. ✓ Lot of reference info.	✓
Mach	✓ Already implemented. ✓ Not much of information.	×
Linux	✓ Too heavy, hard to build.	×
OpenBSD	✓ Too heavy	×

Figure 16. Pros and cons for choosing OS

Solution:

Chose Minix.

Challenge 2: VirtualBox setup

Challenge:

Encountered a VirtualBox bug that the hard disk image cannot be created.

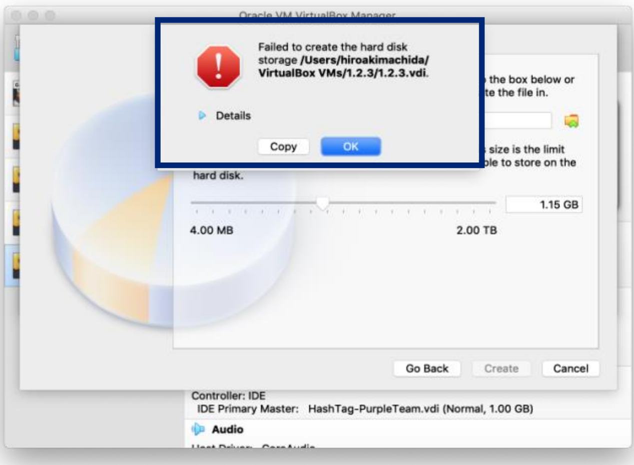


Figure 17. VirtualBox error

Solution:

Searched a forum, that took some time, and specified the disk size by bytes.

Challenge 3: Development environment setup

Challenge:

Cannot make the index with Remote System Explorer on Eclipse, neither run debug.

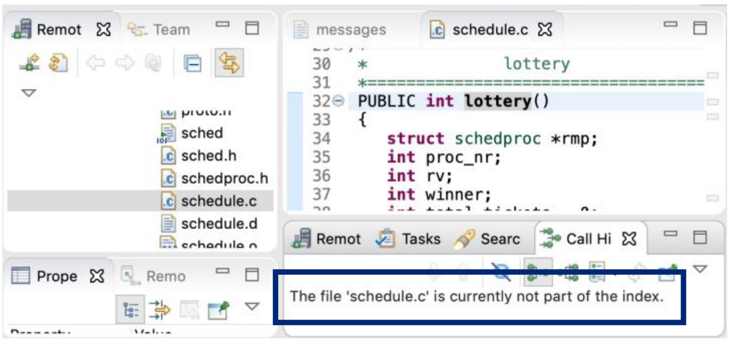


Figure 18. Index cannot be created on Eclipse

Solution:

Copy the sources into my local and check the codes.
Put printf() function into codes and see if the program works.

Challenge 4: Rereference project

Challenge:

Reference project from UCSC didn't work.

The processes do not work based on tickets.

Reference project
CMPS111-project[6]

Command
nice -n 0 ./a.out a 100 300000 >> test1 & nice -n 16 ./a.out b 100 300000 >> test1

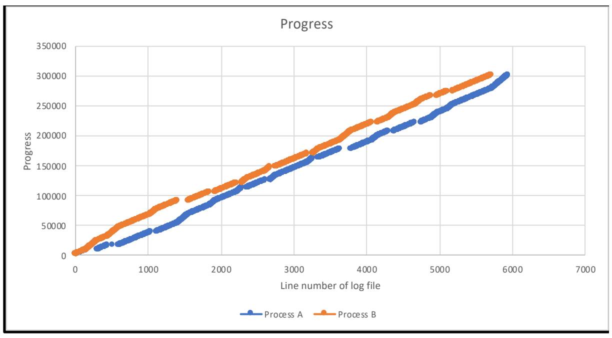


Figure 19. Progress of CMPS111-project

Solution:

Find other reference projects.

[6] https://github.com/freedombird9/CMPS111-project

Challenge 5: Unintentional running process change

Challenge:

Running process changed without calling sys_schedule() by scheduler.

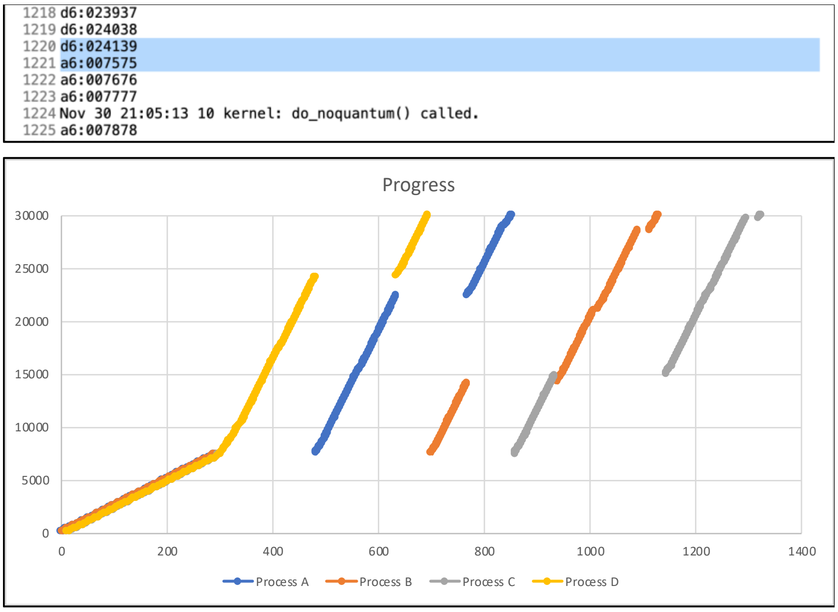


Figure 20. Running process changed without calling sys_schedule()

This is because the kernel does round robin even if sys_schedule() is called by a scheduler.

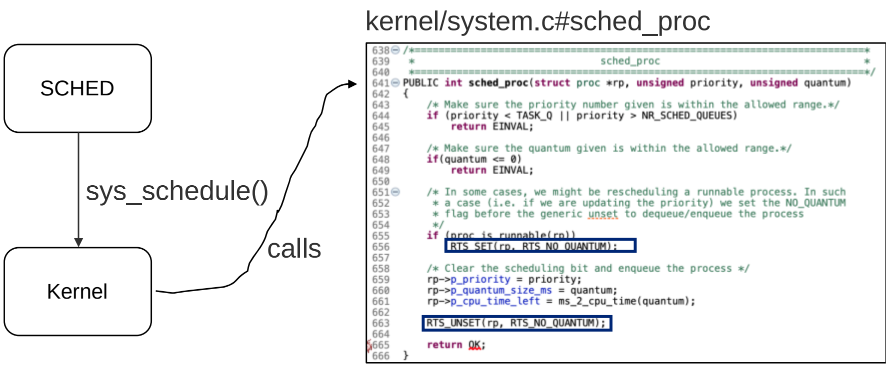


Figure 21. Kernel behavior

Solution:

(This is specification)

Challenge 6: All processes running at a time

Challenge:

At the beginning all process running at a time, but later running separately.

This is because the kernel starts round robin a while after the beginning.

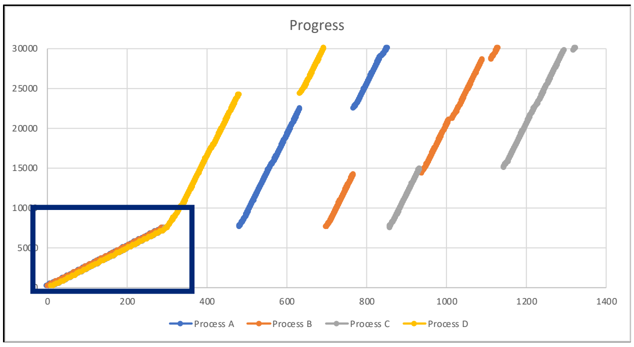


Figure 22. Progress

Solution:
(This is specification)

Chapter 5 Conclusion

Lottery Scheduling can be implemented but there are some unexpected behaviors.

Problem:
Is Lottery Scheduling implementable by a usual developer?
What are challenges to implement?

Conclusion:
Lottery Scheduling can be implemented, but there are some unexpected behaviors.
The challenges range from setting up development environment to implementation itself.

Chapter 6 References

"Operating Systems Design and Implementation, Third Edition"

Original Lottery Scheduling Paper
https://www.usenix.org/legacy/publications/library/proceedings/osdi/full_papers/waldspurger.pdf

Minix Official Site
https://www.minix3.org/

Minix Message Flow
http://www.minix3.org/docs/scheduling/report.pdf

Minix Customization
https://github.com/gosaliajigar/CustomizingMINIX3

Reference code from UCSC
https://github.com/freedombird9/CMPS111-project

Reference code from UCSC2
https://github.com/HashTag-PurpleTeam/minix-lottery-scheduler