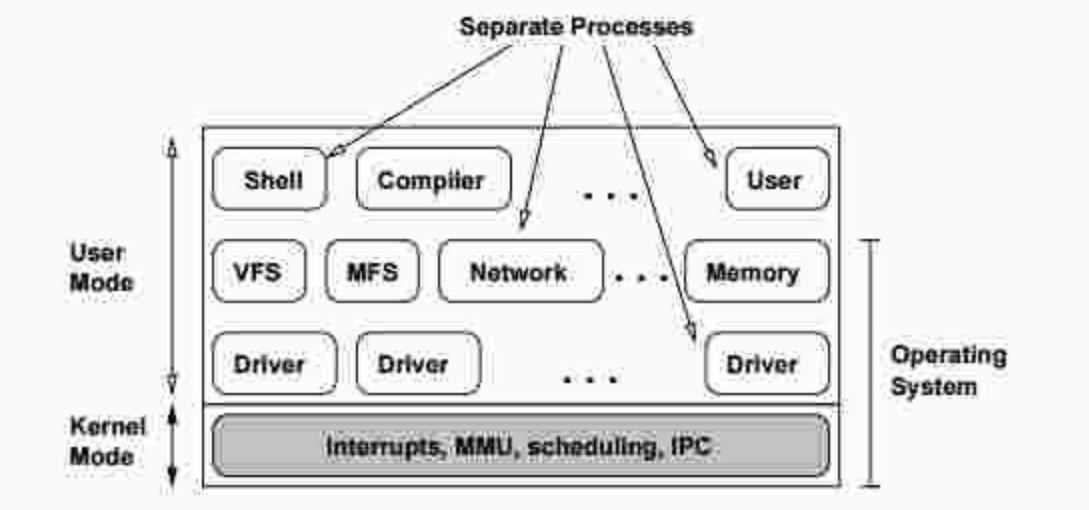
Minix Schedulers - Technical report

The goal of this assignment is to learn how to modify MINIX3, create and implement system calls, modify kernel to some extent and most importantly, gain familiarity with scheduling. Thus, once known how MINIX3 does process scheduling, modify the MINIX's current scheduler (Round-Robin) and test its performance in different cases.

In this assignment, we need to implement two versions of the **Earliest-Deadline-First scheduling - Preemptive and Non-preemptive**.

Introduction

MINIX3 is a microkernel based POSIX submissive, flexible and easy to understand operating system designed to be highly reliable, yielding, and docile. The approach of implementation of both PM and vFS (Virtual file system) is based on concepts of modularity and **fault isolation**. This is achieved by breaking the system into many **self contained modules**.



The above diagram shows the structure of the operating system. A minimal kernel provides interrupt handlers, a mechanism for starting and stopping processes, a scheduler, and interprocess communication. MINIX3 handles deadlocks quite exceptionally as well. Deadlock detection is implemented in the kernel. If a process unexpectedly were or is going to cause a deadlock, the offending is denied and an error message is returned to the caller.

Now coming to the vFS in MINIX, it is built in a **distributed, multiserver** manner. It consists primarily of a top-level vFS process and separate FS process for each mounted partition.

The top-level vFS process receives the requests from user programs through system calls. If actual file system operation is involved requests are made by the vFS to the corresponding FS process to do the job.
This same procedure is followed in my implementation of the system-call, **do_setdeadline**, which I use in passing a message containing the deadline as well as the PID of the forked process from user space to the **pm** and further, to the **scheduler**.

The vFS plays an important role in implementing system calls. The following are the steps followed when implementing the **do_setdeadline** system call :

- The user process (here **longrun.c**) calls the mysyscall function (which further calls do_setdeadline function in **/usr/src/servers/pm/misc.c**) of the POSIX library which builds the request message and sends it to the VFS process.

- The VFS process copies the information in message struct from userspace.

- After this, the message reaches the process manager, PM, which is received by the function do_setdeadline function in **misc.c**. The code is given below :

```
int do_setdeadline(){
     m_in.m_type = SYS_SETDL;
     m_in.m1_i2 = mp->mp_endpoint;
     return _taskcall(SCHED_PROC_NR, SYS_SETDL, &m_in);
}
```

Here, the **SYS_SETDL** is macro defined in **/usr/include/minix/com.h**, which defined the message type and prompts the scheduler that process requesting to be scheduled is of type **longrun**.

- Once the parameters are set for the global message struct, **m_in**, a system call named **_taskcall** is invoked again, with first parameter being another macro, SCHED_PROC_NR, specifying that the system call is regarding scheduling a process, the second argument being the message type and third, begin the message itself.

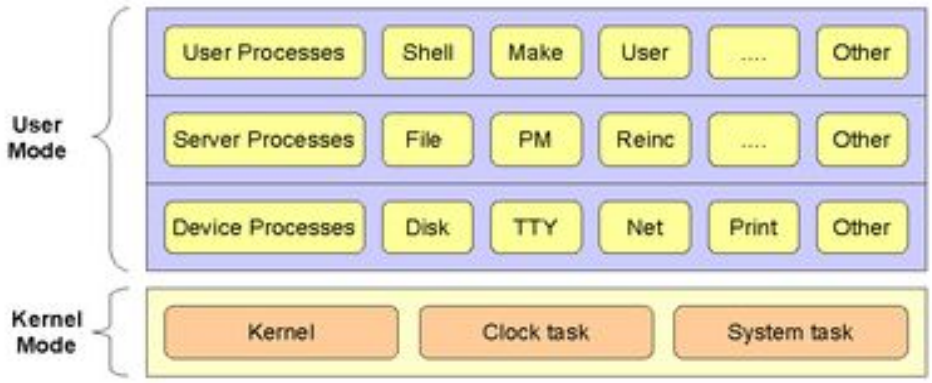
- The **_taskcall** system-call successfully transfers the message to the scheduler, in the function `_do_set_deadline(message *m_ptr)` function in **/usr/src/servers/sched/schedule.c**, and the process's deadline and endpoint is set accordingly as shown below :

```
int _do_set_deadline(message *m_ptr) {
     struct schedproc *rmp;
     int rv; int proc_nr_n;
     unsigned old_q, old_max_q; //m_ptr->m1_i2 is the
endpoint passed from message

     proc_nr_n = _ENDPOINT_P(m_ptr->m1_i2); //setting
     the endpoint here from the message here, used in
     getting the process, rmp

     rmp = &schedproc[proc_nr_n];
     rmp->deadline = m_ptr->m1_i1; //setting the
     deadline from the message here
     /*
     *
     */
}
```

Now we look at kernel functionality on MINIX here. The policy followed while scheduling the processes is handled by the server in the user space, and the kernel does the actual job of scheduling processes on the CPU and **context-switching**. The MINIX Kernel comprises of 4 sections, with the top 3 being User mode and the bottom layer being the Kernel mode as shown below :



The MINIX 3 Microkernel Architecture

As is evident from the above figure, the scheduler is under layer 2, under the director sched. The actual schedulers are present in the **User mode** in layer 2, thus allowing the user to run multiple schedulers, each with their independent policy.

In MINIX, the scheduling is added in the User mode of the kernel, thus giving us the authority of change the scheduling policy in MINIX. First, let's have a look at the default scheduling policy in MINIX.

For scheduling, the Kernel defines 16 priority queues, and follows **preemptive Round-Robin** scheduling as its default scheduling policy. It, by default, always chooses the process at the head of the highest priority queue. The processes have time quanta assigned to them, and when a process runs out of quantum, it is preempted and is appended at the end of its current priority queue immediately. Note that this policy not starvation free, it can still occur if some of the processes have large time quantum, but it is meant to schedule servers during the system startup, in order to avoid the problem of running out of time quantum before the scheduler starts at the start of the system.

The following is the procedure, how a user interacts with the scheduler :

- The user after invoking a **do_setdeadline** system call, he makes a library call to the scheduler.

- This request goes to the system wrapper present in **/usr/src/servers/pm/misc.c**, which has the definition of the system call as well. The user has basically reached the in the PM part of MINIX.

- There is another file with the same name, **schedule.c** (which is present in **/usr/src/servers/sched** directory, which is the main scheduling unit), present in the same directory, and the library call made earlier passes through the functions of this code.

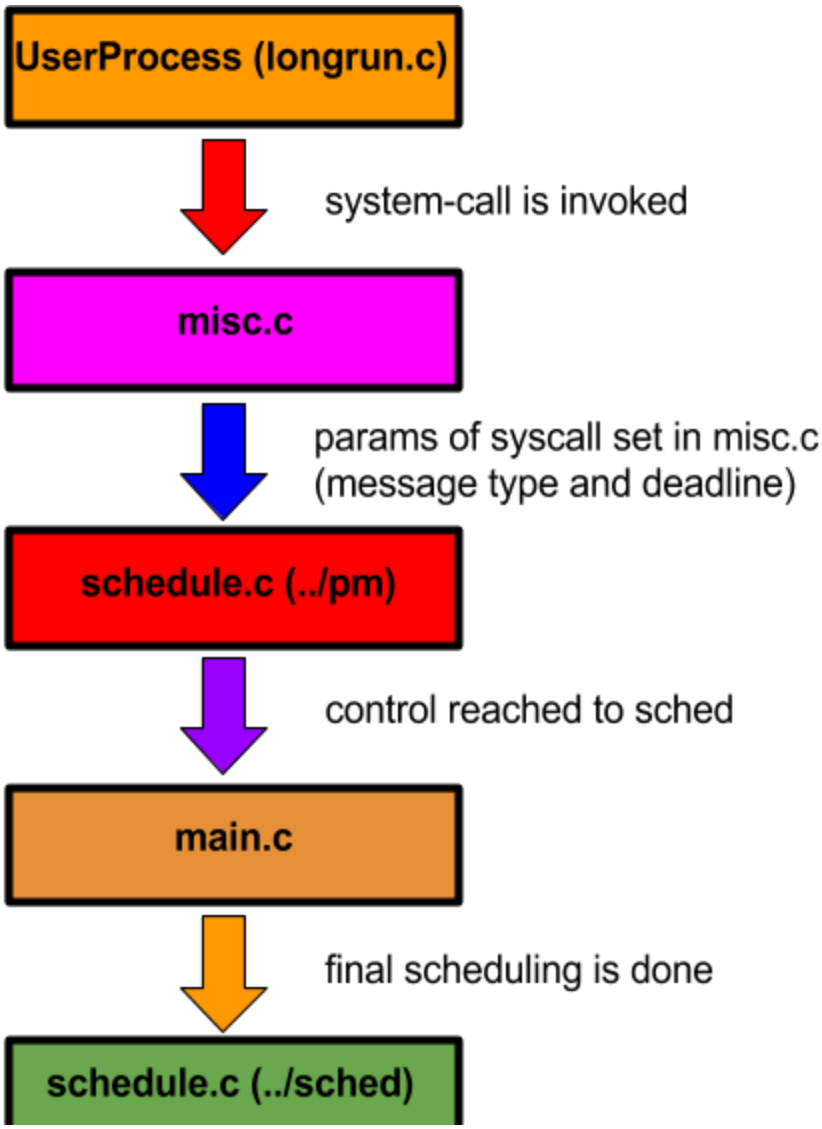
- Now, the call reaches the file **main.c** in the directory **/usr/src/servers/sched**, or shortly, the directory **sched**, which is the main directory, and depending upon the type of the message passed to **main.c**, which can SCHEDULING_START, SCHEDULING_INHERIT or SYS_SETDL, where the first one is generally for scheduling system processes, like login process (/usr/bin/login), ls, cd, etc., the second one being for the normal user space programs which **don't** invoke the **do_setdeadline** system call, and are hence not supposed to be scheduled using the **EDF** policy and the last one, which we defined for the purposes of this assignment, and hence, the programs (of the type **longrun**), who invoke the **do_setdeadline** system call, set the type of `message m_in` to **SYS_SETDL** in **misc.c**, and which are hence supposed to be scheduled using the EDF policy.

- The scheduler is called finally.

This "choosing" inside **main.c** is done as shown in the code below (the switch clause, `call_nr`, is the type of the message struct) :

```
switch(call_nr) {
     case SYS_SETDL:
          result = _do_set_deadline(&m_in); break;
     case SCHEDULING_INHERIT:
     case SCHEDULING_START:
          result = do_start_scheduling(&m_in); break;
     /* other procedures */
}
```

The following is the flow chart which pretty-much sums up the procedure :



The first three phases belong to PM, while the last two phases belong to SCHED, doing the main scheduling part.

Preemptive EDF scheduling : code changes

1. Setting the deadline obtained as a command-line argument to **longrun.c** :

   ➔ A system-call whose "front-end" is defined in **mysyscallscheduling.h** (present in **/usr/include** for the purposes of **#include**), is invoked in **longrun.c**. (This front-end part of the system-call is just an interface for user to communicate with the kernel, the main implementation is done in **/usr/src/servers/pm/misc.c**)

   The function is called in longrun.c as,

```
mysyscall(atoi(argv[2]), getpid()); //argv[2] is
the deadline supplied as commandline-arg and getpid() gives
the PID of the current process
```

   and, with, **mysyscall** defined in **mysyscallscheduling.h**, as follows :

```
int mysyscall(int deadline, int pid){
     message m;
     m.m1_i1 = deadline;
     m.m1_i2 = pid;
     return _syscall(PM_PROC_NR, SETDEADLINE, &m);
}
```

2. Setting the parameters passed through the system-call defined above in the PM and passing them to the scheduler in SCHED using another system-call named **_taskcall** as shown :

   ➔ The do_setdeadline acts as the final phase of the system-call, which transfers the message struct, **m_in**, by-reference to **main.c** in sched as shown in the code below :

```
int do_setdeadline(){
     m_in.m_type = SYS_SETDL;
     m_in.m1_i2 = mp->mp_endpoint;
     return _taskcall(SCHED_PROC_NR, SYS_SETDL,
&m_in);
}
```

3. Adding extra switch case, corresponding to the type **SYS_SETDL** of message **call_nr** in main.c, and the corresponding function, **_do_set_deadline(message *m_ptr)**, schedules the syscall invoking processes (longrun), according to **EDF** policy.

4. Once everything is set and control is passed to the scheduler, the following are the changes (highlighted in **yellow**) made in the main scheduling unit, **schedule.c** :

➔ **do_noquantum(message *m_ptr)**

```
if (rmp->priority < MIN_USER_Q && rmp->deadline ==
LONG_MAX){
     rmp->priority += 1; /* lower priority */
}
```

➔ **do_start_scheduling(message *m_ptr)**

```
/* add deadline here */
rmp->deadline = LONG_MAX;

switch (m_ptr->m_type) {
     case SCHEDULING_START:
          rmp->deadline = LONG_MAX;
          rmp->priority = rmp->max_priority;
          rmp->time_slice = (unsigned)
          m_ptr->SCHEDULING_QUANTUM;
          break;

     case SCHEDULING_INHERIT:
          if ((rv =
     sched_isokendpt(m_ptr->SCHEDULING_PARENT,
     &parent_nr_n)) != OK)
               return rv;
          rmp->deadline = LONG_MAX;
          rmp->priority =   schedproc[parent_nr_n].priority;
          rmp->time_slice =
          schedproc[parent_nr_n].time_slice;
          break;
```

```
        default:
                assert(0);
}
```

➔ A new function, **_do_set_deadline(message *)**, is added to schedule the processes invoking the syscall, according to **EDF** policy :

```
int _do_set_deadline(message *m_ptr) {
        struct schedproc *rmp;
        int rv; int proc_nr_n;
        unsigned old_q, old_max_q;

        /* m_ptr->m1_i2 is endpoint passed from message */
        proc_nr_n = _ENDPOINT_P(m_ptr->m1_i2);

        rmp = &schedproc[proc_nr_n];
        printf("proc with pid : %d and deadline : %d has been
        added to the ready queue.\n", m_ptr->m1_i2,
        m_ptr->m1_i1);

        /* setting the deadline here */
        rmp->deadline = m_ptr->m1_i1;

        int add_factor = (m_ptr->m1_i1)/1000000;
        rmp->priority = rmp->max_priority + add_factor;
        if (rmp->priority > MIN_USER_Q) {
                rmp->priority = MIN_USER_Q;
        }
        old_q = rmp->priority;
        old_max_q = rmp->max_priority;
        rmp->max_priority = rmp->priority;
        if ((rv = schedule_process_local(rmp)) != OK) {
                rmp->priority = old_q;
                rmp->max_priority = old_max_q;
                rmp->deadline = LONG_MAX;
        }
        rmp->flags = IN_USE;
        return rv;
}
```

➔ The function, **balance_queues(struct schedproc *)** is changed in such a manner, that, those processes, which have their deadline equal to **LONG_MAX**, i.e., are system processes, would be scheduled using the default **Round-Robin** scheduling policy, whereas the others using the **EDF** policy, for which we decrease the deadline by a constant factor every time the function is called, which is every 100 ticks.

```
static void balance_queues(struct timer *tp) {
	struct schedproc *rmp;
	int proc_nr;
	for (proc_nr=0, rmp=schedproc; proc_nr < NR_PROCS;
	proc_nr++, rmp++) {
		if (rmp->deadline == LONG_MAX) {
			if (rmp->flags & IN_USE) {
				if (rmp->priority > rmp->max_priority) {
					rmp->priority -= 1;
					/* increase priority */
					schedule_process_local(rmp);
				}
			}
		}
		else {
			if (rmp->flags & IN_USE) {
				if (rmp->deadline <= DECREASE_FACTOR) {
					printf("Process killed, missed its
					deadline.\n");
					sys_kill(rmp->endpoint, SIGKILL);
				}
				else {
					/* decrement deadline */
					rmp->deadline -= DECREASE_FACTOR;
					if (rmp->priority >
				rmp->max_priority) {
						rmp->priority -= 1;
						schedule_process_local(rmp);
					}
				}
			}
		}
```

```
            set_timer(&sched_timer, balance_timeout,
            balance_queues, 0);
        }
}
```

Minor changes made in the following files for implementing the system-call :

- **callnr.h** => Added this line : `#define SETDEADLINE 69.` This macro is added for the **do_setdeadline** function defined in **misc.c**, used to define the system call number.

- **proto.h** => Added this line : `int do_setdeadline(void);` This defines the prototype of the function defined in **misc.c**, acts as a forward declaration for it.

- **schedproc.h** => Added a new field to store the deadline of the process : `long int deadline;`

- **table.c** => Added a new entry for our system call. **table.c** includes the declaration of a table of system calls implemented by the PM server. There is one entry in the table for each system call and the position in the table is the system call number. To add a system call to the PM server, I replaced one of the unused table entries with the name of the system-call function.
  I replaced the line :
  ```
  no_sys,              /* 69 = unused  */
  ```
  with :
  ```
  do_setdeadline,        /* 69 = setdeadline  */
  ```

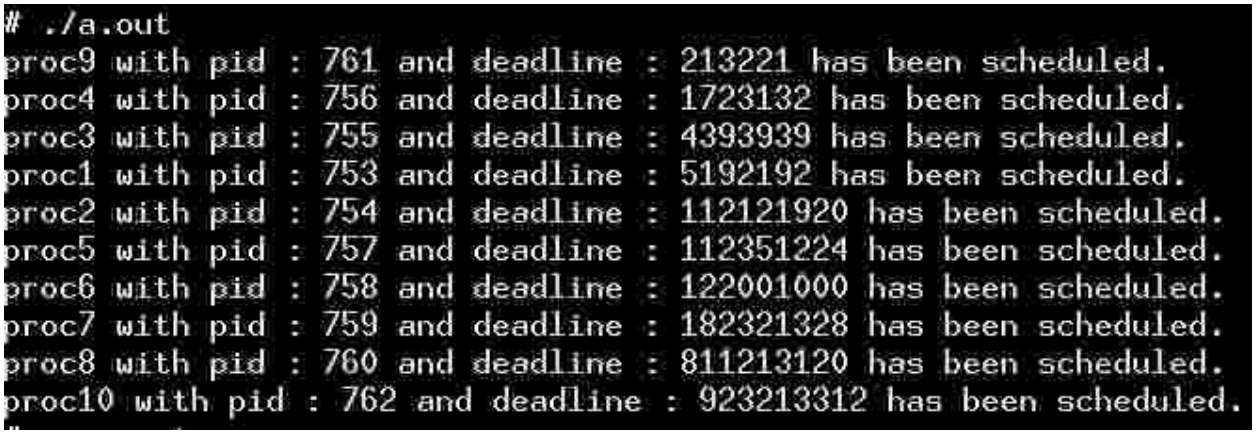
Testing the functionality of code :

A C program named **testrun.c** is developed in order to fork the processes of type **longrun** which is run on an input file named **inp-params.txt** made using provided **n**, **μ1**, **μ2** and λ as input.

The expected output is supposed to be the messages, printed in the increasing order of the deadlines of the processes begin scheduled. Note that the output doesn't include the processes which miss their deadlines, a **SIGKILL** is sent to the processes which miss their deadlines, and hence, they don't print the final line which shows their deadlines and the corresponding PID.

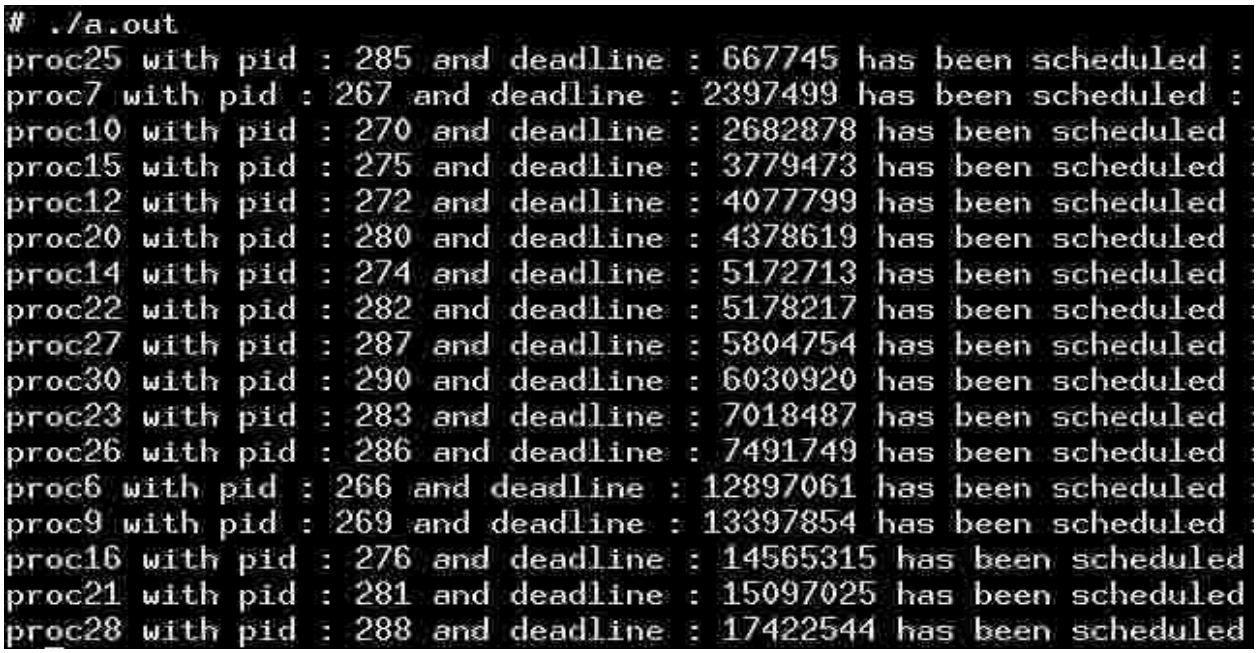
Preemptive scheduler : A test file was generated with **n** = 15, **μ1** = 90, **μ2** = 60, and λ, varying from 10 to 50 for the purposes of making graphs. The following is the

screenshot of one of the outputs when the **testrun.c** is run on the input file made using above mentioned parameters :



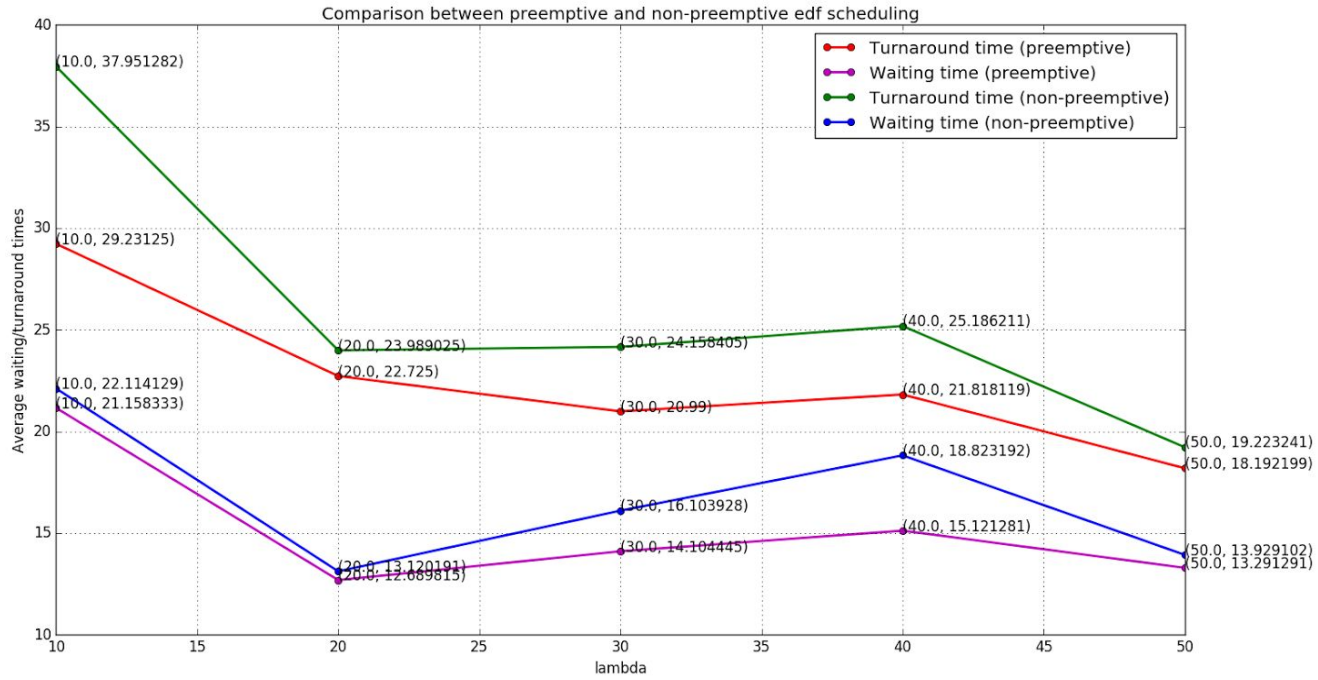
It can be seen clearly that the processes are scheduled according to the **EDF-policy**, and hence, correctly demonstrating the scheduling.

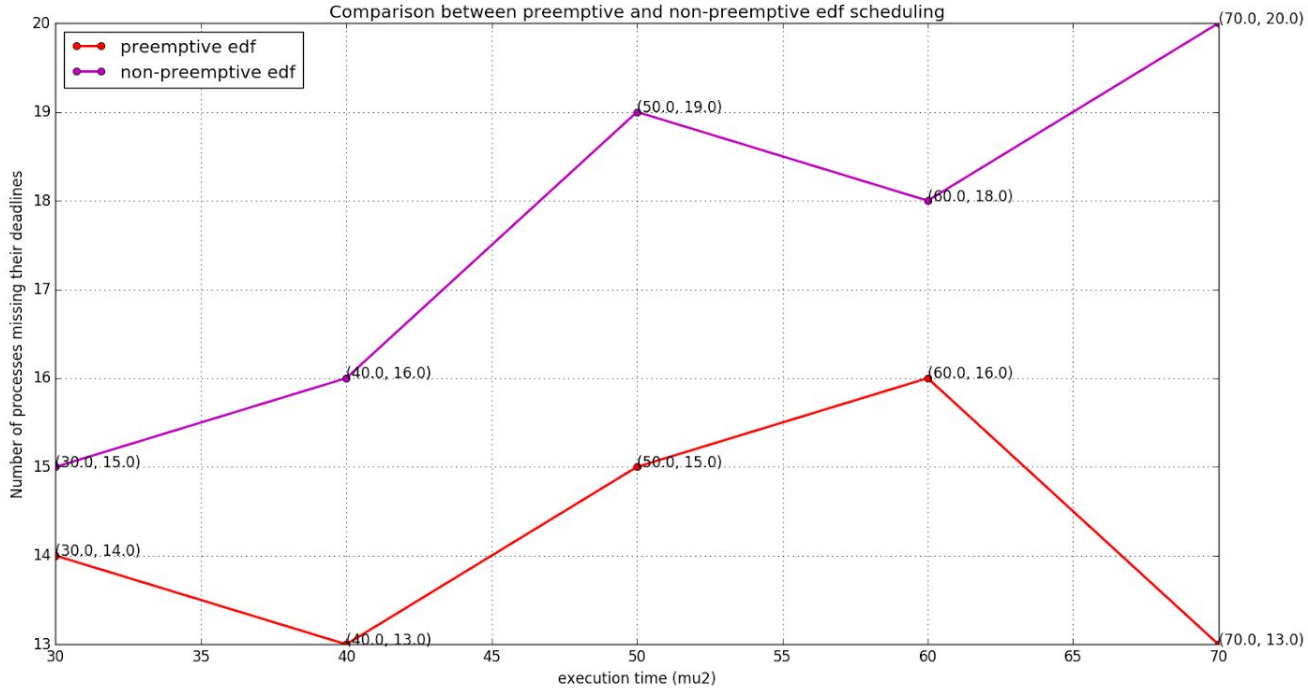
Here is another screenshot of the output run on following parameters as suggested in the question - **n** = 30, **μ1** = 30, **μ2** = 20 :



Clearly, this output also suggests that EDF policy is strictly followed while scheduling the processes. Both the test files used are attached along the assignment directory.

The following graphs demonstrate the average waiting and turnaround times taken by a process of the type **longrun** for preemptive **EDF** scheduler v/s λ , and the number of processes missing their deadlines v/s **μ2** :





The above readings are for preemptive scheduler only. The implementation of Non-preemptive scheduler is not done as the readings were inconsistent for it, and implementing it requires changing considerable amount of **kernel** code in **/usr/src/kernel/proc.c**, where we need to nearly change the complete **enqueue**, **dequeue** and **pick_proc** functions.

However, as discussed in the class, I implemented a static version of the **non-preemptive EDF** scheduling, which is more or less, similar to the normal **priority scheduling**, only that, the priorities are calculated depending on the supplied deadlines. The following code change was made :

```
/* m_ptr->m1_i2 is endpoint passed from message */
proc_nr_n = _ENDPOINT_P(m_ptr->m1_i2);

rmp = &schedproc[proc_nr_n];
printf("proc with pid : %d and deadline : %d has been
added to the ready queue.\n", m_ptr->m1_i2,
m_ptr->m1_i1);

/* setting the deadline here */
rmp->deadline = m_ptr->m1_i1;

/* setting the time-slice to infinite, in order to
avoid preemption */
rmp->time_slice = INT_MAX;
```

The performance of this version of scheduling is generally poorer as compared to the preemptive one, as the same can be observed from the graph. The number of processes missing their deadlines is always higher, regardless of the execution time calculated using **μ2**, since this is non-preemptive scheduling and the waiting and turnaround times, as expected, are also higher.

Thus, using the observations from the graphs above, we can state that preemptive earliest-deadline-first policy is better than the non-preemptive one.