Modifying the MINIX Scheduler

Daniel Rodríguez García ICOM 5007

To comprehend in a deeper how an operating system works we must address one of the must import parts of it, its process scheduler. The process scheduler is the one that decides what program runs at any time. This decision isn't made at random, they rely on a predetermined algorithm. There are several algorithms that are frequently used for this, some of them are the First-Come-First-Served which takes the processes as they arrive at the process queue. The FCFS algorithm is one of the most basic ones, from it we can derive others like Round-Robin, the scheduling algorithm used by **MINIX** or to Earliest Deadline First(EDF) the one that we are going to implement in this laboratory to compare against the RR used by **MINIX**.

The main problem that is presented with this algorithm is that they can be more useful than the others for certain applications. To decide which one to choose we must first compare them with each other. To do this, certain variables must be taken in account, examples of this are execution time, throughput, latency, fairness and many other ones depending on the desired use of the system.

For this experiment, we are going to use execution time analysis, represented by the Big-O notation. This will give us a deep understanding on how a sequential machine like a computer can handle processes in average.

From the class, we know that Round-Robin is one of the most efficient algorithms for average day applications. It bends and changes depending on the needs of the system, giving the user a better experience. From the definition of both, RR and EDF we could assume that RR is going to outperform any test given against EDF since it can adapt back and forth to the needs of the execution system.

Here we propose an efficient solution in order to compete against RR, **MINIX** default scheduling algorithm.

I. Implement MINIX kernel call

- Add a new kernel call for the Earliest Deadline First Scheduling Algorithm. For this, modify the header file **com.h** located in /usr/src/include/minix

```
#  define SYS_UPDATE   (KERNEL_CALL + 52)   /* sys_update() */
#  define SYS_EXIT     (KERNEL_CALL + 53)   /* sys_exit() */

#  define SYS_SCHEDCTL (KERNEL_CALL + 54)   /* sys_schedctl() */
#  define SYS_STATECTL (KERNEL_CALL + 55)   /* sys_statectl() */
#  define SYS_SCHEDEDF (KERNEL_CALL + 56)   /* sys_schededf() */
```

- Add the prototype for the kernel call. For this modify the header **system.h** located in /usr/src/kernel

```
_PROTOTYPE( int do_schedule,    (struct proc * caller, message
*m_ptr) );
_PROTOTYPE( int do_schedctl, (struct proc * caller, message
*m_ptr) );
_PROTOTYPE( int do_schededf, (struct proc * caller, message
*m_ptr) );

_PROTOTYPE( int do_statectl, (struct proc * caller, message
*m_ptr) );
```

- Added the call to **system.c** located in /usr/src/kernel to map it in the system.

```
#endif

  /* Scheduling */
  map(SYS_SCHEDULE, do_schedule);     /* reschedule a process */
  map(SYS_SCHEDCTL, do_schedctl);     /* change process scheduler
*/

  map(SYS_SCHEDEDF, do_schedctl);     /* give process a deadline
*/


}
```

- Implement the call in a file called **do_schededf.c** and save it in /usr/src/kernel/system and add its entry to the Makefile.inc. Use the minix attributes that were used in past labs in order to pass relevant information through the call, which in this case is the given process deadline.

```
#include "kernel/system.h"

#include <minix/endpoint.h>

/
*===========================================================
==========*
 *                    do_schededf                            *

*===========================================================
==========*/
PUBLIC int do_schededf(struct proc * caller, message * m_ptr)
{

  struct proc *rp;
  int proc_nr = 0;
  endpoint_t e_proc_nr;

  if(isokendpt(m_ptr->m1_i3, &proc_nr))
     return EINVAL;
  rp = proc_addr(proc_nr);
  rp->deadLine = m_ptr->m1_i2;
  return(OK);
}
```

- Add a protoype for the EDFSA, called **sys_edf.c** and save it in /usr/src/lib/libsys. Add an entry for this file inthe **Makefile** in the same folder.
- Add the service **SYS_SCHEDEDF** in the **parse.c** file, located in /usr/src/commands/service.

```
    { "STIME",      SYS_STIME },
    { "VMCTL",      SYS_VMCTL },
    { "EDF",        SYS_SCHEDEDF },
    { NULL,         0 }
};
```

- Execute make hdboot in /usr/src/tools. Then sync and reboot.
- Update with make and make install from /usr/src/commands/service
- make and make install from /usr/src/lib/libsys

II. EDF

Now that we have the kernel calls in place we can start implementing our earliest deadline first algorithm. This algorithm dequeues the process that is closer to end its execution given by the deadline. This is clearly a disadvantage; this means that it would give a priority or preference to processes who use a small amount of time from the given deadlines.

- Modify and add a new call in an empty slot. In this case EDFSA in slot 56.

```
#define IOCTL           54
#define FCNTL           55
#define EDFSA    56
#define FS_READY   57
#define EXEC            59
#define UMASK           60
#define CHROOT          61
```

- Modify **table.c** in /usr/src/servers/pm and add do_edfsa

```
no_sys,          /* 55 = fcntl    */
do_edfsa,        /* 56 = edfsa    */
no_sys,          /* 57 = unused   */
no_sys,          /* 58 = unused   */
do_exec,    /* 59 = execve   */
no_sys,          /* 60 = umask    */
```

- Add a prototype for the call in **proto.h** located in /usr/src/servers/pm

```
/* misc.c */
_PROTOTYPE( int do_reboot, (void)                              );
_PROTOTYPE( int do_sysuname, (void)                            );
_PROTOTYPE( int do_getsysinfo, (void)                          );
_PROTOTYPE( int do_getprocnr, (void)                           );
_PROTOTYPE( int do_getepinfo, (void)                           );
_PROTOTYPE( int do_getepinfo_o, (void)                         );
_PROTOTYPE( int do_svrctl, (void)                              );
_PROTOTYPE( int do_getsetpriority, (void)                      );
_PROTOTYPE( int do_edfsa, (void)                    );
```

- Add the call definition in the file **misc.c** located in /usr/src/servers/pm. In the headers include **unistd.h**

```
/
*===========================================================
==========*
 *                    do_edfsa                               *

*===========================================================
==========*/
PUBLIC int do_edfsa(int deadline)
{
      message m;
      m.m2_i1 = deadline;
      return (_syscall(PM_PROC_NR,MYEDF, &m));
}
```

- Create a library for the call named **edfsa.h** and save it in /usr/include

```
#include <lib.h>
#include <unistd.h>

PUBLIC int edfsa(int deadline)
{
      message m;
      m.m2_i1 = deadline;
      return ( _syscall(PM_PROC_NR, MYEDF, &m) );

}
```

- Compile, sync and reboot the system.

- Modify the file **proc.c** located in the folder /usr/src/kernel. Define the prototype and make the process picker use enqueueEDF instead of the default algorithm.

```
/
*===========================================================
===========*
 *                    enqueueEDF                          *

*===========================================================
===========*/
PRIVATE void enqueueEDF(struct proc *rp)
{
  const int q = rp->p_priority;

  struct proc **header, **trailer;
  assert(proc_ptr_ok(rp));
  assert(proc_is_runnable(rp));
  assert(!is_zero64(rp->p_cpu_time_left));
  assert(q >= 0);
  header = get_cpu_var(rp->p_cpu, run_q_head);
  trailer = get_cpu_var(rp->p_cpu, run_q_tail);
  if (!header[q]) {
      header[q] = trailer[q] = rp;
      rp->p_nextready = NULL;
  }
  else
      rp->p_nextready = header[q];
      header[q] = rp;
  read_tsc_64(&(get_cpulocal_var(proc_ptr->
p_accounting.enter_queue)));



  rp->p_accounting.dequeues--;
  rp->p_accounting.preempted++;

#if DEBUG_SANITYCHECKS
  assert(runqueues_ok_local());
#endif
```

- In order to test the implementation, create a file and called **tester.c** and run it. Which will assign deadlines linearly.

```
#include <stdio.h>
#include <edfsa.h>

int main(){

	int counter=4, pid=0, deadline = 0, i;
	for(i = 0; i < counter; i++){

		pid = fork();
		if(pid == 0){
			deadline = 100-i*10;
			myedf(deadline);
			sleep(1);
			break;
		}
	}
	if(pid == 0){
		printf("DEADLINE IS: %d\n",deadline);
	}
	return 0;
}
```

III. Results and conclusion

As we can see the tester shows a linear behavior where it assigns a deadline given a defined iterative value from a loop. This will create a linear behavior when analyzing the enqueueing times, therefore making this algorithm being bound by $O(n)$. This compared with the default algorithm in **MINIX** is poor since it uses a multi-priority Round-Robin algorithm and has a quantum assigned. This quantum is periodically decreased therefore it gives a good chance to run to most processes. It avoids having to execute in $O(n)$ time by using pre-emption and executes near $O(1)$. Therefore, the default scheduling algorithm is better than the Earliest Deadline First.