Student: Cristiam Martin Jackson
[SOI] - Systemy Operacyjne

REPORT
LABORATORY N. 2 - SCHEDULING OF PROCESSES

PURPOSE

The purpose of the laboratory is to implement a scheduler algorithm for the user's processes, in which the processes should be classified into two groups depending on the number of the mentioned process, in this way:

- A - if process number is divisible by 2
- B - if process number is not divisible by 2

Also a system service should be implemented in order to set the time proportions in which the scheduler should choose to perform the tasks of group A as well as group B, the number (X) should be entered by the user and consists of a number between 0 and 100

- A will get X part of the window destined for the process and B will get 100 - X.

Conditions:

The call should return 0 if it was possible to change the proportion of time, otherwise it will obtain -1, this will happen in the case the input is given when any process B is being executed.

SOLUTION

In order to access to the kernel, a system call should be created, through the memory manager (MM). The memory manager will access the kernel using the function (_taskcall(SYSTASK,SYS_SETSCHEDULER,&msg);), by this way the messages can be sended in both directions, through &msg and the returned value from the functions.

First, the system call is implemented in the Memory Manager, in the header (callnr.h) is defined the system call with the number 69, as it was available.

/usr/include/minix/callnr.h

```
#define SETSCHEDULER 69
```

MM - table.c

```
set_scheduler,/* 69 = set_scheduler */
```

MM - proto.h

```
/* scheduler's system call */
_PROTOTYPE( int set_scheduler, (void)                                    );
```

MM - main.c

```
/*===========================================================*
 *                                set_scheduler                                *
 *===========================================================*/
int set_scheduler(){
  message msg;
  int i = mm_in.m1_i1;
```

```
/* m_in is a global variable set to MM's incoming message,
 * m_in.m1_i1 is the integer parameter set in the test program. */
    printf("System call set_scheduler called with value %d\n", i);
/* Send the int to the system service sys_scheduler() in system.c*/
   msg.m1_i1=i;
  return _taskcall(SYSTASK,SYS_SETSCHEDULER,&msg);
}
```

Once the system call is created, we can create the functions in the kernel that can be accessed through the MM's system call already implemented. The function _taskcall(SYSTASK,SYS_SETSCHEDULER,&msg) sends messages to the kernel.
In the address: /usr/include/minix/com.h
Define a new system service called SYS_SETSCHEDULER with an available and subsequent number.

```
# define SYS_SETSCHEDULER 22
```

The communication to the kernel will be handled by the system.c c-function, which is called using the taskcall function.

- In proc.h, new variables are added and modification of the processes' structure is done.
- In main.c, new variables are initialized.
- In proc.c, the scheduling function is implemented.
- In clock.c, the call to the system scheduler is done

KERNEL - proc.h

```
struct proc {
.
.
.
 int evenodd;                    /* if even 0, odd 1 */
  };
  int proca;                   /* counter por even and odd processes */
  int procb;                   /* counter por even and odd processes */
  int tickspproc;          /* control of ticks from clock.c */
  int timefora;                /* portion of time given to process A */
  int scheduling;              /* if scheduling 1 FALSE 2 TRUE */
```

KERNEL - main.c

```
/*===================================*
 *                main               *
*====================================*/
PUBLIC void main()
{
.
.
.
  /* Initialize the number of calls of each process */
```

```
  proca=0;
  procb=0;
  /* Initialize the tick counter */
  ticksproc=0;
  /* Initialize the default proportion of time for A process */
  timefora=50;
  scheduling=1;
.
.
.
```

System.c handles the interface between the MM and the kernel, in here the sys_scheduler function is written; this function will validate whether if the change of the portions of time can be done or not, as well as validate the number of processes and reset of some variables every time the system call is used.

KERNEL - system.c

```
FORWARD _PROTOTYPE( int sys_scheduler, (message *m_ptr) );
```

```
switch (m.m_type) {    /* which system call */
              .
              .
              .
              case SYS_SETSCHEDULER:       r = sys_scheduler(&m);          break;
              default:                     r = E_BAD_FCN;
          }
```

```
/*===========================================================*
 *                                sys_scheduler                                  *
 *===========================================================*/
PRIVATE int sys_scheduler(message *m_ptr)
{
          int n;
          int m;
          struct proc *p,*q;
          proca=0;
          procb=0;
          ticksproc=0;
          /*printf("\nI am in sys_scheduler\nand received this as x: %d\n",m_ptr->m1_i1);*/
          /*PROC1 is process father and PROC2 is process child*/
          p=proc_addr(m_ptr->PROC1);
          q=proc_addr(m_ptr->PROC2);
          m=proc_number(p);
          n=proc_number(q);
          printf("\nProcess FATHER address: %d\tProcess number: %d",p,m);
          printf("\nProcess CHILD address: %d\tProcess number: %d\n",q,n);
          if (n%2==0){
             timefora=m_ptr->m1_i1;
             rdy_head[USER_Q]->evenodd=10;
             return 0;
```

```
        }
        else{
          rdy_head[USER_Q]->evenodd=20;
          return -1;
        }
}
```

In clock.c the action that calls the scheduler is performed, after 100ms a new process is called if it has been running for too long, in this function a new counter is implemented, every time the function detects that the old process is still running, it will call the scheduler and sum one unit until the ticksproc variable reaches 100, in which case it will be reset to 0.

KERNEL - clock.c

```
/*===========================================================*
 *                                do_clocktick                                    *
*===========================================================*/
PRIVATE void do_clocktick()
{
.
.
.
  /* If a user process has been running too long, pick another one. */
  if (--sched_ticks == 0) {
        if (bill_ptr == prev_ptr){
        if (scheduling==1) ticksproc=0;
          ticksproc=ticksproc+1;
          lock_sched();/* process has run too long */
          if (ticksproc >= 100)
                ticksproc = 0;
        }
        sched_ticks = SCHED_RATE;                   /* reset quantum */
        prev_ptr = bill_ptr;                        /* new previous process */

  }
}
```

In proc.c the part of scheduling is modified, so that a new process will be selected after the previous one reached the portion of time assigned for the user, the processes are enqueued and once the change of process is done, the scheduler calls the function that picks the process, depending on the hierarchy (the lowest priority is given to the users' processes).

The implemented solution compares the number of ticks given by variable implemented in clock.c, with the time for a established for the user and brought to the kernel through the system call implemented in system.c and the group of the current process.

KERNEL - proc.c

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

```
 *                                sched                                  *
*=========================================================*/
PRIVATE void sched()
{
/* The current process has run too long.  If another low priority (user)
 * process is runnable, put the current process on the end of the user queue,
 * possibly promoting another user to head of the queue.
 */
  int i;
  register struct proc *rp;         /* Struct table */
  if (rdy_head[USER_Q] == NIL_PROC){
            ticksproc=0;
            return;
}
  rp=rdy_head[USER_Q];
                                /* PRINT QUEUE */
while (rp != NIL_PROC){
   printf("%d-",rp->p_nr);
   rp=rp->p_nextready;
}
printf("| ");
                                /*END PRINT QUEUE*/
                                /*processes numbers*/
/* printf(" a: %d ",timefora);*
 * printf("%d ",rp->p_nr);*/
                                /*Ticks per process*/
/* printf("%d",ticksproc);*/

if (rdy_head[USER_Q]->p_nr%2==0 && ticksproc>=1 && ticksproc < timefora){
 rdy_head[USER_Q]->evenodd=28;
}
 else if (rdy_head[USER_Q]->p_nr%2==1 && ticksproc >= timefora){
 rdy_head[USER_Q]->evenodd=27;
}
 else if (rdy_head[USER_Q]->p_nr%2==1 && ticksproc>=1 && ticksproc < timefora)
 {
   rdy_head[USER_Q]->evenodd = 11;
   rdy_tail[USER_Q]->p_nextready = rdy_head[USER_Q];
   rdy_tail[USER_Q] = rdy_head[USER_Q];
   rdy_head[USER_Q] = rdy_head[USER_Q]->p_nextready;
   rdy_tail[USER_Q]->p_nextready = NIL_PROC;
  /* printf("\nodd ");*/
 }
 else if (rdy_head[USER_Q]->p_nr%2==0 && ticksproc >= timefora)
 {
```

```
   rdy_head[USER_Q]->evenodd = 12;
   rdy_tail[USER_Q]->p_nextready = rdy_head[USER_Q];
   rdy_tail[USER_Q] = rdy_head[USER_Q];
   rdy_head[USER_Q] = rdy_head[USER_Q]->p_nextready;
   rdy_tail[USER_Q]->p_nextready = NIL_PROC;
  /* printf("\neven ");*/
 }
/*printf("V ");*/
pick_proc();
}
```

TESTING

In order to control the number of processes created to test the implementation, the next functions were created:

- 2test.c : (compiled as cc -o 2t 2test.c) contains a function created to set the portion of time required for the implementation, the input is done by the user (./2t INPUT), where input is the desired portion of time for the A process, since this change is only possible when the process is divisible by 2, this function must be executed in the two terminals when the change is desired, the terminal will display 0 or -1 whether if the change was possible or not. The time process can also be checked using this function, but since there is a time since the user changes from the first terminal to the second and executes “parallel” processes, it is not very effective solution for measuring times.

/usr/usr/sched/2test.c

```
#include <time.h>
#include <sys/times.h>
#include <sys/types.h>
/* provides _syscall and message*/
#include <lib.h>
#include </usr/include/minix/type.h>
#include <stdio.h>
/* provides atoi*/
#include <stdlib.h>
int meta(void);
int main(int argc, char **argv) {
    int i,n;
    time_t a,b;
    double c,t;
    clock_t start, end;
/* Minix uses message to pass parameters to a system call*/
    message m;
/* expecting positive numbers*/
    i = atoi(argv[1]);
    printf("\nYou entered: %d\n",i);

    if (i>=0&&i<=100){
```

```
    m.m1_i1 = i;

    n=_syscall(MM_PROC_NR, SETSCHEDULER, &m);
        /* _syscall asks the kernel to ask the server process identified by
         * MM_PROC_NR (the MM server process) to execute the system call
         * identified by call number SETSCHEDULER with the parameter in the
         * message m (accessed through its address &m).
         */
            printf("scheduler replies: %d\n",n);
            start=clock();
            a=time(0);
            meta();
            end=clock();
            b=time (0);
            c=(double)(end-start)/CLOCKS_PER_SEC;
            printf("\ntime difference: %f\n",c);
            t=b-a;
            printf("TIME: %f\n",t);
      }
        else{
        printf("\nNumber not valid\n");
        }
    return 0;
}
int meta(){
int j;
int k;
for(j;j<2000000000;j++){
for(k;k<4;k++){
}

}
return 0;
}
```

- tfork.c: Uses the fork function in order to create two processes, every process is timed using the clock and time functions, belonging to the time.h library, the clock function retrieves the processor time of every process, while the time function retrieves the real time an user perceive for that function, in order to perform a long time process.

```
#include <time.h>
#include <unistd.h>
#include <sys/times.h>
#include <sys/types.h>
/* provides _syscall and message*/
#include <lib.h>
```

```
#include </usr/include/minix/type.h>
#include <stdio.h>
/* provides atoi*/
#include <stdlib.h>

int meta(void);
int main(int argc, char **argv) {

    int i,pidf,n;
    time_t ap,bp,ac,bc;
    double tp,tc,cp,cc,port,num,den;
    clock_t startp, endp,startc,endc;
/* Minix uses message to pass parameters to a system call*/
    message m;
/* expecting positive numbers*/
    i = atoi(argv[1]);
    printf("\nYou entered: %d\n",i);
    if (i>=0&&i<=100){
    m.m1_i1 = i;
    n=_syscall(MM_PROC_NR, SETSCHEDULER, &m);
       /* _syscall asks the kernel to ask the server process identified by
        * MM_PROC_NR (the MM server process) to execute the system call
        * identified by call number SETSCHEDULER with the parameter in the
        * message m (accessed through its address &m).
        */
          printf("scheduler replies: %d\n",n);
          if(fork()==0){
          startp=clock();
          ap=time(0);
          meta();
          endp=clock();
          bp=time(0);
          cp=(double)(endp-startp)/CLOCKS_PER_SEC;
          tp=bp-ap;
          den=(cp/tp)*100;
          if (i>=50){
          printf("\ntime difference parent: %f\nTIMEp: %f\n",cp,tp);

                    printf("Portion: %f%%\n",den);
          }
          /*den=tp-cp;*/
          /*den=(cp/tp)*100;
          printf("Percentage: %f%%", den);*/
          }
          else {

          ac=time(0);
          startc=clock();
          meta();
          bc=time(0);
```

```
        endc=clock();
        cc=(double)(endc-startc)/CLOCKS_PER_SEC;
        tc=bc-ac;
        num=((tc/cc)-1)*100;
        if (i<50){
        printf("\ntime difference child: %f\nTIME: %f\n",cc,tc);

                printf("Portion: %f%%\n-",num);
        }
        /*num=tc-cc;*/
        }
        /*port=(tc-cc)/((tp-cp)+(tc-cc))*100;
        printf("\nPercentage: %f, %f",num,den);
        */
    }
     else{
     printf("\nNumber not valid\n");
     }
   return 0;
}
int meta(){
int j,k;
for(j=0;j<2;j++){
for(k=0;k<2000000000;k++);
}
return 0;
}
```

RESULTS

In order to perform the calculations, were required the values added in the table 1. The way the percentage was found is as follows.

T_odd = tc - cc

T_even = tp - cp

Percentage = T_odd / (T_odd + T_even)

Table 1. Results of different tests

Time for A	ODD		EVEN		Measured percentage
	clock (cc)	time (tc)	clock (cp)	time (tp)	
90%	20.03	31	20.15	22	85.57%
80%	19.95	32	19.96	24	74.89%
70%	20.03	33	20.00	26	68.37%
60%	19.98	34	20.00	33	51.89%
50%	19.95	35	19.95	35	50.00%

40%	20.05	32	20.06	40	37.47%
30%	19.95	26	20.00	40	23.22%
20%	20.01	24	20.03	40	16.65%
10%	20.00	22	19.98	40	9.08%

The evidence for the process with 80% of the time for A processes is shown, in the image 1 the time is set to 80%, the system replies 0, meaning that the change was possible. The terminal also displays the way the queue is being moved and the process being executed.

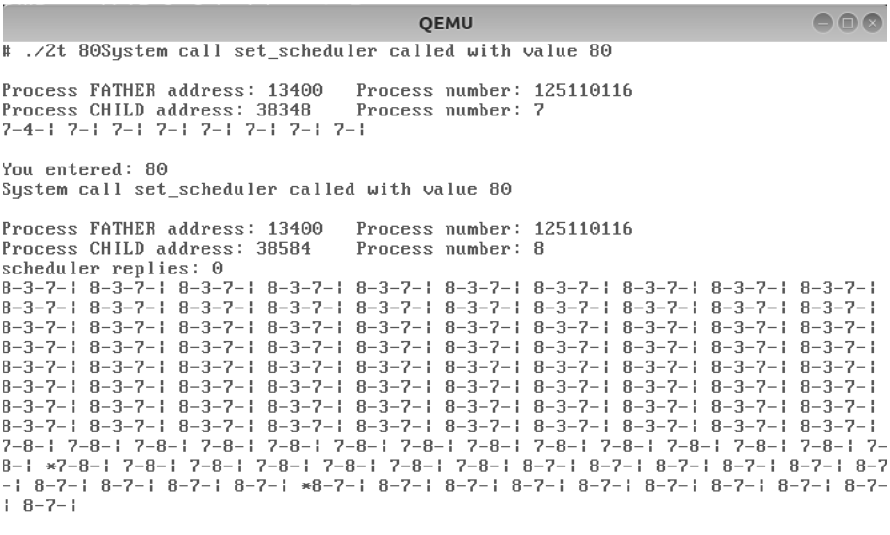

Image 1.

The image 2 shows the second testing-function implemented, the scheduler replies -1, however, since this function was executed right after setting the portion into 80%, there will be no changes.

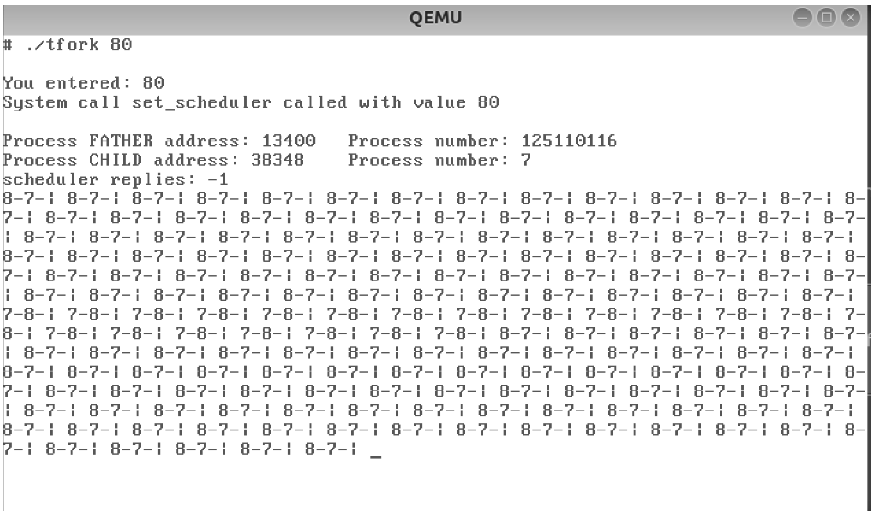

Image 2.

The implementation of the scheduling algorithm threw a good approach of what was expected, some differences in time are due to different processes that appeared during the

test, some external processes the user perform could have been added to the queue, eg.: typing, pressing some keys from the keyboard, printing some useful variable, etc. However those differences in time where small in compare to the time of execution of the test which was in general close to one minute.

REFERENCES

- Modern Operating Systems (4th Edition) by Andrew S. Tanenbaum (Author), Herbert Bos (Author)