

Diario de A Bordo

Estructura de Sistemas Operativos

Grupo X5-10

Javier Gómez Herrero

Javier Ramiro de Santos

### Práctica 3

- 03/04/2020

Aprovechando el tiempo libre que nos brinda la situación actual unida a las vacaciones de Semana Santa decidimos comenzar con la práctica. Mediante videollamada comenzamos a leer las guías de MINIX para comenzar con la instalación de la máquina virtual, y concluimos en que los primeros pasos de la instalación debemos hacerlos por separado, al menos hasta que necesitemos ayuda mutua.

- 12/04/2020

Tras mas de una semana de trabajo propio ponemos en común los avances. Aunque la instalación de VirtualBox no trajo ningún problema, la configuración de MINIX sí. Ambos tuvimos dificultades a la hora del cambio de UUID; primero pensábamos que el comando debía ejecutarse dentro de la máquina virtual, y cuando por fin nos dimos cuenta de que era en el equipo huésped donde se utilizaba, MacOS no le permitía a Javier ejecutarlo. Tras algún que otro reinicio y cambio de configuración, terminamos por conseguirlo.

A partir de aquí, uno de los problemas más complicados fue la distinta disposición de teclado, sobretodo para navegar por las distintas terminales ya que en la disposición de teclado de Apple no existe la tecla *alt*. Tras bastantes intentos acabamos descubriendo la función de cada tecla. Intentamos también configurarlo en un portátil *hp-omen* pero también tiene una disposición de teclado diferente y seguía dando errores.

También encontramos problemas al crear el nuevo usuario ya que al añadirlo en el fichero *passwd* el segundo (la contraseña) no permitía un asterisco como representación de la no contraseña. Tras averiguar eso se pudo ejecutar correctamente el comando *passwd*.

En cuanto a la compilación del núcleo resultó bastante simple, aunque olvidamos cambiar el orden de arranque, con lo cual no veíamos las modificaciones del fichero *tty.c*. Tras cambiar el orden de arranque todo funcionó correctamente.

```
Minix Release 2.0 Version 0

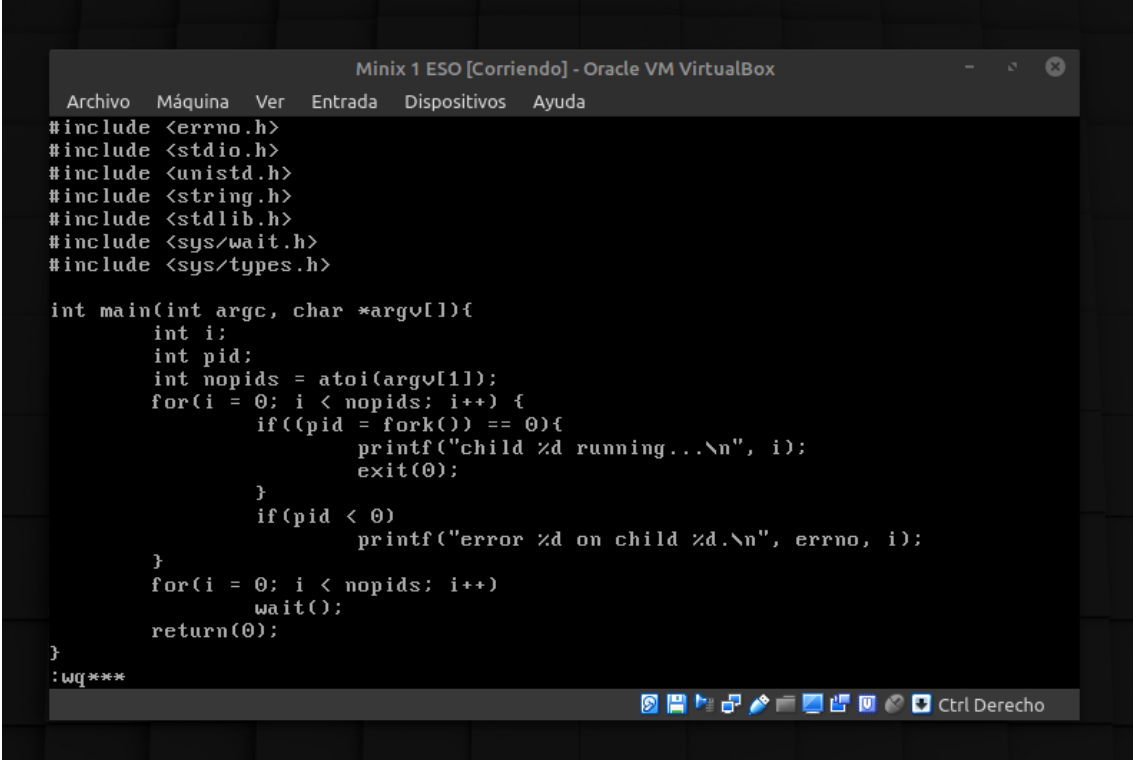
minix2.0.0 login: javrami
Password:
$ ls
.ellepro.b1  .profile  boot  etc  fd1  mnt  tmp
.exrc        bin       dev   fd0  minix  root  usr
$ cd minix
$ ls
2.0.0  2.0.0r12
$
```

Minix 2.0.0 - Disco duro V2 Copyright 1997 Prentice-Hall, Inc.  
Javier Ramiro de Santos

.Executing in 32-bit protected mode

- 20/04/2020

Nos reunimos una última vez para crear los programas *CreaProcesos* y *leeAout*. El primero parece no ser demasiado problema, pues ya estábamos familiarizados con el uso de *fork()* gracias a la anterior práctica (*UVash*). De hecho, a priori el mayor reto era el uso del editor vi dentro de la máquina virtual que, lejos de ser igual de versátil que en la máquina huésped, nos hacía arduo el camino de programar (amén del ya mencionado hecho de los problemas con el teclado). Tras unos minutos dominando el teclado y algunos más programando un bucle que invocase a *fork()*, compilamos el programa con *cc* y vimos muchos mas errores de los que esperábamos (no esperábamos ninguno). Sin embargo, el problema tenía fácil solución: declarar todas las librerías que habíamos olvidado declarar.



```
Minix 1 ESO [Corriendo] - Oracle VM VirtualBox
Archivo  Máquina  Ver  Entrada  Dispositivos  Ayuda

#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <sys/types.h>

int main(int argc, char *argv[]){
    int i;
    int pid;
    int nopids = atoi(argv[1]);
    for(i = 0; i < nopids; i++) {
        if((pid = fork()) == 0){
            printf("child %d running...\n", i);
            exit(0);
        }
        if(pid < 0)
            printf("error %d on child %d.\n", errno, i);
    }
    for(i = 0; i < nopids; i++)
        wait();
    return(0);
}

:wq***
```

En cuanto a *leeAout.c* tuvimos muchos problemas para comprender el propósito del programa, al principio pensábamos que era similar a *cat* pero la salida era obviamente irreconocible.

Aunque leímos en la guía de la práctica que la estructura era *exec*, no conseguimos pasarle el parámetro de archivo y tampoco que imprimiera los tipos de variables de cabecera. Buscando en internet encontramos la estructura *stab*, que comprueba que la cabecera sea correcta e imprime los tipos de variables de esta.

```

#include <sys/types.h>
#include <fcntl.h>

#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>

int main(int argc, char *argv[]){

    char *archivo;
    struct stat sb;

    if(stat(archivo, &sb) != -1){
        printf("%s ocupa %lld bytes.\n", archivo, sb.st_size);
        printf("%s Tipo de archivo: %u\n", archivo, sb.st_mode);
        printf("Numero de bloques de 512B asignados en %s: %lld\n", arch
    }

    ~
    ~
    ~
    "leeAout.c" 21 lines, 432 chars

```

Aún no comprendemos muy bien todas las cabeceras pero en los siguientes días seguiremos intentando conseguirlo con la estructura *exec* de *a.out.h*.

#### Práctica 4

- 02/05/2020

Tras unos días bastante ajetreados nos reunimos para comenzar con la práctica 4 por videollamada. Comenzamos obviamente investigando todos los ficheros que se mencionan a lo largo de ésta como indica la primera parte de la misma.

El primer archivo que abrimos es *\_fork.c*, que no necesita permisos de administrador, y comprobamos como ejecuta la llamada al sistema con los argumentos MM, FORK y m.

```

#include <lib.h>
#define fork      _fork
#include <unistd.h>

PUBLIC pid_t fork()
{
    message m;

    return(_syscall(MM, FORK, &m));
}
~

```

Le seguimos la pista hasta `_syscall.c`, que tampoco necesita privilegios y comprobamos que desde aquí se realiza otra llamada a `_sendrec()`, con los argumentos `who` y `msgptr`.

```
#include <lib.h>

PUBLIC int _syscall(who, syscallnr, msgptr)
int who;
int syscallnr;
register message *msgptr;
{
    int status;

    msgptr->m_type = syscallnr;
    status = _sendrec(who, msgptr);
    if (status != 0) {
        /* 'sendrec' itself failed. */
        /* XXX - strerror doesn't know all the codes */
        msgptr->m_type = status;
    }
    if (msgptr->m_type < 0) {
        errno = -msgptr->m_type;
        return(-1);
    }
    return(msgptr->m_type);
}
~
~
"/usr/src/lib/other/syscall.c" [READONLY] 22 lines, 424 chars
```

Seguimos tirando del hilo y entramos al fichero `_sendrec.s`. Observamos que está escrito en ensamblador y buscamos la función que nos atañe: `__sendrec`. Vemos en ella la instrucción que nos importa ahora mismo: `int SYSVEC`, que es la interrupción del sistema. Es aquí donde el procesador cambia de modo a modo privilegiado.

```
__sendrec:
    push    ebp
    mov     ebp, esp
    push    ebx
    mov     eax, SRCDEST(ebp)      ! eax = dest-src
    mov     ebx, MESSAGE(ebp)     ! ebx = message pointer
    mov     ecx, BOTH             ! _sendrec(srcdest, ptr)
    int     SYSVEC                ! trap to the kernel
    pop     ebx
    pop     ebp
    ret
```

Nos dirigimos ahora a la función que inicializa la tabla de vectores de interrupción: *protect.c*. Aquí se encuentra el vector *SYS386\_VECTOR*, definido en *const.h*.

```
#if _WORD_SIZE == 4
/* Complete building of main TSS. */
tss.iobase = sizeof tss;      /* empty i/o permissions map */

/* Complete building of interrupt gates. */
int_gate(SYS386_VECTOR, (phys_bytes) (vir_bytes) s_call,
        PRESENT | (USER_PRIVILEGE << DPL_SHIFT) | INT_GATE_TYPE);
#endif
}

/*=====
 *
 *=====
init_codeseg
/*=====
/* Fixed system call vector. */
#define SYS_VECTOR      32      /* system calls are made with int SYSVEC */
#define SYS386_VECTOR   33      /* except 386 system calls use this */
#define LEVEL0_VECTOR   34      /* for execution of a function at level 0 */
```

La pista nos lleva ahora a *s\_call()*, que vuelve a estar escrita en ensamblador. Ésta a su vez llama a *sys\_call()* con los argumentos insertados en la pila.

```
?=====
?*
?*=====
        .align 16
_s_call:
_p_s_call:
        cld                                ! set direction flag to a known value
        sub     esp, 6*4                    ! skip RETADR, eax, ecx, edx, ebx, esi
        push    ebp                        ! stack already points into proc table
        push    esi
        push    edi
        o16 push    ds
        o16 push    es
        o16 push    fs
        o16 push    gs
        mov     dx, ss
        mov     ds, dx
        mov     es, dx
        incb    (_k_reenter)
        mov     esi, esp                    ! assumes P_STACKBASE == 0
        mov     esp, k_stktop
        xor     ebp, ebp                    ! for stacktrace
        ? end of inline save
        sti                                ! allow SWITCHER to be interrupted
        ? now set up parameters for sys_call()
        push    ebx                        ! pointer to user message
        push    eax                        ! src/dest
        push    ecx                        ! SEND/RECEIVE/BOTH
        call    _sys_call                  ! sys_call(function, src_dest, m_ptr)
        ? caller is now explicitly in proc_ptr
        mov     AXREG(es), eax             ! sys_call MUST PRESERVE si
        cli                                ! disable interrupts

! Fall into code to restart proc/task running.
```

Para terminar con todo esto, se ejecuta la función *\_restart()* de *mpk386s.s*, lo que permite continuar ejecutando otro proceso, terminando aquí el modo privilegiado, con lo que el procesador vuelve a modo usuario.

```

/*=====
 *
 *=====
sys_call
 *=====
PUBLIC int sys_call(function, src_dest, m_ptr)
int function;          /* SEND, RECEIVE, or BOTH */
int src_dest;          /* source to receive from or dest to send to */
message *m_ptr;        /* pointer to message */
{
/* The only system calls that exist in MINIX are sending and receiving
 * messages. These are done by trapping to the kernel with an INT instruction
 * The trap is caught and sys_call() is called to send or receive a message
 * (or both). The caller is always given by proc_ptr.
 */

    register struct proc *rp;
    int n;

    /* Check for bad system call parameters. */
    if (!isoksrc_dest(src_dest)) return(E_BAD_SRC);
    rp = proc_ptr;

    if (isuserp(rp) && function != BOTH) return(E_NO_PERM);

    /* The parameters are ok. Do the call. */
    if (function & SEND) {
        /* Function = SEND or BOTH. */
        n = mini_send(rp, src_dest, m_ptr);
        if (function == SEND || n != OK)
            return(n);          /* done, or SEND failed */
    }

    /* Function = RECEIVE or BOTH.
     * We have checked user calls are BOTH, and trust 'function' otherwise.
     */
    return(mini_rec(rp, src_dest, m_ptr));
}

```

Las llamadas al sistema en Minix se realizan mediante mensajes (*send* y *rec*) como se entiende en el código de *sys\_call()*. Se realizan mediante hardware, para lo cual el procesador debe encontrarse en modo privilegiado, es decir, haciendo “trapping to the kernel with an INT instruction”. Ésta instrucción INT de ensamblador se encuentra en la función *\_sendrec()* del fichero *\_sendrec.s*. Es en esa función donde se hace el “trapping” del que se sale más tarde (y tras haber ejecutado las llamadas pertinentes al sistema) mediante la función *\_restart()* del fichero *mpk386s.s*.

Tras acabar esta primera parte de la práctica, decidimos continuar en otro momento, ya que ambos teníamos exámenes al caer.

- 08/05/2020

Tras librarnos de las tareas más acuciantes, volvemos a la carga dispuestos a acabar la práctica 4. Una vez más, nos reunimos por videollamada y comenzamos a trabajar en la segunda parte de ésta práctica.

En *\$usr/src/mm* se encuentra el archivo *forkexit.c*. Es en éste directorio donde se implementa la llamada al sistema, y trasteando aquí con los ficheros, encontramos en *forkexit.c* la función *do\_fork()*, que es la que se encarga de realizar la orden *fork()*.

Teniendo esto en cuenta, sólo tuvimos que conseguir acceso de administrador e introducir un *printf* en ella para poder imprimir un mensaje cada vez que nuestra máquina quisiera dividir un proceso.

Tras ello hicimos un *make hdbboot* para que el cambio surtiera efecto y con ello terminar.

```
Minix Release 2.0 Version 0

minix2.0.0 login: Somos Javier Ramiro de Santos y Javier Gomez Herrero y hemos m
odificado fork().
Somos Javier Ramiro de Santos y Javier Gomez Herrero y hemos modificado fork().
Somos Javier Ramiro de Santos y Javier Gomez Herrero y hemos modificado fork().
Somos Javier Ramiro de Santos y Javier Gomez Herrero y hemos modificado fork().
Somos Javier Ramiro de Santos y Javier Gomez Herrero y hemos modificado fork().
Somos Javier Ramiro de Santos y Javier Gomez Herrero y hemos modificado fork().

Minix Release 2.0 Version 0

minix2.0.0 login: _
```

Comprobamos que al reiniciar la máquina ésta hace varios *fork()* y nuestro mensaje se imprime correctamente.

## Práctica 5

- 20/05/2020

Al acabar un examen que ambos teníamos, aprovechamos para continuar con el laboratorio. Comenzamos con la fase 1, editando el fichero *callnr.h*, para añadir una nueva llamada *ESOPS* de nivel 3 al sistema.

```
/* Posix signal handling. */
#define SIGACTION      71
#define SIGSUSPEND     72
#define SIGPENDING     73
#define SIGPROCMASK    74
#define SIGRETURN      75

#define REBOOT         76
#define ESOPS          77
```

Incrementamos también el número de llamadas permitidas.

```
#define NCALLS          78    /* number of system calls allowed */

#define EXIT            1
#define FORK            2
#define READ            3
#define WRITE           4
```



Añadimos en *table.c* la nueva llamada al sistema, al igual que en *proto.h* (en este último caso y como dice el guión, siguiendo el ejemplo de *do\_reboot()*).

```
do_sigsuspend, /* 72 = sigsuspend */
do_sigpending, /* 73 = sigpending */
do_sigprocmask, /* 74 = sigprocmask */
do_sigreturn, /* 75 = sigreturn */
do_reboot, /* 76 = reboot */
do_esops, /* 77 = interrupcion prueba */

PROTOTYPE( int do_alarm, (void) );
PROTOTYPE( int do_kill, (void) );
PROTOTYPE( int do_ksig, (void) );
PROTOTYPE( int do_pause, (void) );
PROTOTYPE( int set_alarm, (int proc_nr, int sec) );
PROTOTYPE( int check_sig, (pid_t proc_id, int signo) );
PROTOTYPE( void sig_proc, (struct mproc *rmp, int sig_nr) );
PROTOTYPE( int do_sigaction, (void) );
PROTOTYPE( int do_sigpending, (void) );
PROTOTYPE( int do_sigprocmask, (void) );
PROTOTYPE( int do_sigreturn, (void) );
PROTOTYPE( int do_sigsuspend, (void) );
PROTOTYPE( int do_reboot, (void) );
PROTOTYPE( int do_esops, (void) );
```

Como se recomienda en el guión, escribimos la función *do\_esops()* al final del fichero *utility.c*, enviando el mensaje a la tarea del sistema y bajando así un nivel mas hasta el nivel 2. Para ello copiamos el código que se nos facilita.

```
PUBLIC int do_esops() {

    int a1, a2, a3;

    a1 = mm_in.m1_i1;
    a2 = mm_in.m1_i2;
    a3 = mm_in.m1_i3;

    /* Mensaje para comprobar el funcionamiento */
    printf("MM:do_esops: %d %d %d\n", a1, a2, a3);

    /* Reenvio del mensaje a la tarea del sistema */
    _taskcall(SYSTASK, ESOPS, &mm_in);

    /* Preparacion de la respuesta a enviar hacia arriba, nivel 4,
    es decir, el proceso de usuario */
    /* result2 es una variable externa que se copiara en mm_out.m1_i1 */
    result2 = mm_in.m1_i1;
    mm_out.m1_i2 = mm_in.m1_i2;
    mm_out.m1_i3 = mm_in.m1_i3;

}
```

Nos desplazamos a *system.c* para escribir la función prototipo que dará servicio a nuestra llamada al sistema y que escribimos a imagen de *do\_getmap()*.

```

FORWARD _PROTOTYPE( int do_getsp, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_kill, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_mem, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_newmap, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_sendsig, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_sigreturn, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_endsig, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_times, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_trace, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_umap, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_xit, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_vcopy, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_getmap, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_esops, (message *m_ptr) );

#if (SHADOWING == 1)
FORWARD _PROTOTYPE( int do_fresh, (message *m_ptr) );
#endif

/*=====
 *                               sys_task                               *
 *=====
PUBLIC void sys_task()
:wq
#

```

Nos dirigimos ahora a `sys_task()` para añadir nuestra llamada al sistema mediante su introducción en el `switch`. A estas alturas hemos llegado ya al nivel 2.

```

        case SYS_GETSP:      r = do_getsp(&m);      break;
        case SYS_TIMES:     r = do_times(&m);      break;
        case SYS_ABORT:     r = do_abort(&m);      break;
#if (SHADOWING == 1)
        case SYS_FRESH:     r = do_fresh(&m);      break;
#endif
        case SYS_SENDSIG:   r = do_sendsig(&m);    break;
        case SYS_SIGRETURN: r = do_sigreturn(&m);  break;
        case SYS_KILL:      r = do_kill(&m);       break;
        case SYS_ENDSIG:    r = do_endsig(&m);     break;
        case SYS_COPY:      r = do_copy(&m);       break;
        case SYS_UCOPY:     r = do_vcopy(&m);      break;
        case SYS_GBOOT:     r = do_gboot(&m);     break;
        case SYS_MEM:       r = do_mem(&m);        break;
        case SYS_UMAP:      r = do_umap(&m);       break;
        case SYS_TRACE:     r = do_trace(&m);     break;
        case ESOPS          r = do_esops(&m);     break;
        default:            r = E_BAD_FCN;

    }

    m.m_type = r;          /* 'r' reports status of call */
    send(m.m_source, &m); /* send reply to caller */
}

```

Aún estando en `system.c` definimos la función `do_esops()`, eligiendo para ello el espacio justo anterior a la función `do_fork()` y utilizando el código facilitado para ello.

```

/*=====
 *
 *                               do_esops
 *=====*/
PRIVATE int do_esops(m_ptr)
register message *m_ptr;
{
    int a1, a2, a3;

    a1 = m_ptr->m1_i1;
    a2 = m_ptr->m1_i2;
    a3 = m_ptr->m1_i3;

    printf("K:do_esops:a1=%d, a2=%d, a3=%d\n", a1, a2, a3);

    m_ptr->m1_i1 = 250;
}

/*=====
 *
 *                               do_fork
 *=====*/

```

Tras todo esto, utilizamos el comando *make hdbboot* para compilar el núcleo y descubrimos que existen errores (De hecho, uno de ellos se puede apreciar en la imagen anterior).

```

# cd ..
# cd tools
# make hdbboot
cd ../kernel && exec make -
exec cc -c -m -I/usr/include system.c
"system.c", line 175: : missing before identifier
"system.c", line 198: unknown selector m1
"system.c", line 198: : deleted
make: Error code 256
make: made 'system.o' look old.
make: Error code 256
#

```

Tras buscar esos errores encontramos que son errores de código y los solucionamos, pudiendo compilar así el núcleo sin errores.

```

exec cc -c -I/usr/include device.c
exec cc -c -I/usr/include path.c
exec cc -c -I/usr/include link.c
exec cc -c -I/usr/include protect.c
exec cc -c -I/usr/include time.c
exec cc -c -I/usr/include misc.c
exec cc -c -I/usr/include table.c
exec cc -o fs -i main.o open.o read.o write.o pipe.o \
    device.o path.o mount.o link.o super.o inode.o \
    cache.o cache2.o filedess.o stadir.o protect.o time.o \
    lock.o misc.o utility.o table.o putk.o
install -S 512w fs
installboot -image image ../kernel/kernel ../mm/mm ../fs/fs init
      text    data    bss      size
38512    6648    54196    99356    ../kernel/kernel
12656     1204    32072    45932    ../mm/mm
28576     2200  4341612  4372388    ../fs/fs
 6828       2032    1356     10216    init
-----
86572     12084  4429236  4527892    total
exec sh mkboot hdbboot
rm /dev/hd1a:/minix/2.0.0r11
cp image /dev/hd1a:/minix/2.0.0r12
Done.
#

```

Inmediatamente tras acabar la fase 1 comenzamos con la 2, creando el subdirectorio *C*, el fichero *practica5.c* en él e incluyendo en éste las librerías *lib.h*, *sys/types.h*, *unistd.h* y *minix/syslib.h*. Escribimos ahora en el fichero el código pertinente para poder hacer la llamada al sistema que hemos programado y comprobar que los campos han sido cambiados desde el núcleo.

```
#include <lib.h>
#include <sys/types.h>
#include <unistd.h>
#include <minix/syslib.h>

int main(int argc, char *argv[]) {

    message mensaje;

    mensaje.m1_i1 = 1;
    mensaje.m1_i2 = 2;
    mensaje.m1_i3 = 3;

    _taskcall(MM, ESOPS, &mensaje);

    printf("m1: %d.\nm2: %d.\nm3: %d.", mensaje.m1_i1, mensaje.m1_i2, mensa
```

Tras solucionar dos errores de compilación, el programa está listo.

```
#include <lib.h>
#include <sys/types.h>
#include <unistd.h>
#include <minix/syslib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {

    message mensaje;

    mensaje.m1_i1 = 1;
    mensaje.m1_i2 = 2;
    mensaje.m1_i3 = 3;

    _taskcall(MM, ESOPS, &mensaje);

    printf("m1: %d.\nm2: %d.\nm3: %d.", mensaje.m1_i1, mensaje.m1_i2, mensa

}
}
```

Reiniciamos la máquina virtual y ejecutamos *practica5*.

```
Minix 2.0.0 - Modificado Copyright 1997 Prentice-Hall, Inc.

Executing in 32-bit protected mode

bios-hd0: 81 cylinders, 16 heads, 63 sectors per track

Memory size =15946K   MINIX =4422K   RAM disk = 1024K   Available =10500K

Tue May 26 18:24:02 MET DST 2020
/dev/hd1c is read-write mounted on /usr
Starting standard daemons: update.

Minix Release 2.0 Version 0

minix2.0.0 login: root
# cd Practica5/
# ./practica5
m1: 0.
m2: 0.
m3: 0.
# _
```

Paramos aquí de momento, dejando la tercera fase para el próximo día, ya que se empieza a hacer tarde.

- 26/05/2020

Terminada la fase 2, pasamos a la fase 3 de la práctica, que si hablamos de desconfinamiento aquí en la ancha Castilla no hemos entrado aún en la fase 1. Primero creamos otra llamada al sistema *-INFOESOPS-* con número 78 que nos dará algunos de los campos de información de un proceso.

Después, mediante un *switch*, transformamos *do\_esops()* en un distribuidor. Para ello modificamos la función existente en *system.c*. En este *switch* utilizamos los casos de ejemplo que vimos en el laboratorio, *LEEPCB1 (do\_esops(arg1, arg2))* y *LEEPCB2 (do\_infosops(arg1, arg2))*.

```
PUBLIC int do_esops() {  
  
    int a1, a2, a3;  
  
    int tipo, arg1, arg2;  
  
    a1 = mm_in.m1_i1;  
    a2 = mm_in.m1_i2;  
    a3 = mm_in.m1_i3;  
  
    tipo = mm_in.m1_i1;  
    arg1 = mm_in.m1_i2;  
    arg2 = mm_in.m1_i3;  
  
    printf("MM:do_esops: %d %d %d\n", a1, a2, a3);  
  
    switch(tipo) {  
  
        case LEEPCB1: do_esops(arg1, arg2);  
                      break;  
        case LEEPCB2: do_infosops(arg1, arg2);  
                      break;  
        default: printf("MM_llamada al sistema %d no definida.\n", tipo)
```

En *infosops* utilizamos un *printf* como primitivo método de depuración para saber si funcionaba o no antes de implementar la estructura que extraería la información del *PCB*.

Una vez comprobado que nuestro mensaje se imprimía pasamos a implementar la estructura para imprimir campos del *PCB* que encontramos en las guías de la asignatura, en este caso, introducción a *minix*. Elegimos dos campos de ejemplo, *pid\_t* para extraer el *id* desde el *MM* y *p\_name* para conocer el nombre del proceso.

Al compilar encontramos un error en el uso de la estructura, ya que intentábamos imprimir los parámetros de la estructura directamente mediante *printf*. Utilizamos variables para guardar los parámetros en ellas, como por ejemplo *pid\_t id* utilizando un *long*, *float* y *char p\_name[16]* nombre y compilamos correctamente el programa.

## Práctica 6

Empezamos la primera fase investigándolas siguientes funciones de *proc.c*:

- *pick\_proc()* – Ésta función decide qué proceso será ejecutado el siguiente. Lo selecciona mediante el *proc\_ptr* y lo guarda en *bill\_ptr*.
- *ready()* – Ésta función es añadir un proceso a la cola de procesos listos para ejecutarse, con lo cual cambia el estado de dicho proceso y se pone en listo.
- *unready()* – Ésta función hace lo contrario a la anterior, como dice en su descripción: “this process is no longer runnable, a process has blocked”; cambia el estado de un proceso a no ejecutable.
- *sched()* – Ésta última función controla si el proceso que ha estado ejecutándose tiene que salir por el quantum para que otro proceso entre, realiza este cambio y pone el proceso en la cola de listos.

Tras comprender estas funciones, pasamos a la gestión del reloj, donde podemos encontrar las siguientes funciones:

- *clock.c: init\_clock()* – Aquí encontramos *TIMER\_COUNT*, que es una cuenta atrás que refleja la duración del quantum. También encontramos *TIMER\_FREQ* y *HZ*, que determinan el número de pulsos del reloj por segundo, es decir, las veces que se interrumpe al procesador.
- *clock.c: clock\_handler()* – Ésta es la rutina de servicio de interrupción que en este caso se ejecuta 60 veces por segundo. Aparece cuando acaba el quantum de un proceso que todavía no ha terminado su ejecución, o cuando el proceso termina su ejecución antes del quantum.
- *clock.c: clock\_task()* – Ésta es la función principal de la tarea del reloj, cuando se produce una interrupción, *clock\_handler* envía un mensaje de error y se ejecuta la rutina *do\_clocktick()*.
- *proc.c: interrupt()* – Ésta es la función encargada de enviar el mensaje a una tarea, en este caso, envía el mensaje de error *HARD\_INT* a *clock\_task()*.

Para averiguar el quantum por defecto nos dirigimos al fichero *clock.c* que nos indica el guion y buscamos la variable *SCHED\_RATE*.

```
/* Constant definitions. */  
#define MILLISEC      100      /* how often to call the scheduler (msec) */  
#define SCHED_RATE (MILLISEC*HZ/1000) /* number of ticks per schedule */
```

En este caso el quantum por defecto es el siguiente:

$SCHED\_RATE = (MILLISEC * HZ / 1000)$

Con los datos del código (imagen de arriba), los HZ los vemos en *init\_clock()*:

$SCHED\_RATE = (100 * 60 / 1000) = 6$

Una vez comprendida la planificación en *minix*, podemos pasar a la segunda fase: el efecto convoy. Sin embargo, esto era tarea para otra jornada puesto que ambos teníamos una semana ajetreada.

- 29/05/2020

Tras una intensa semana, volvemos a la carga con el resto de la sexta práctica. Comenzamos diseñando, como la práctica sugiere, un programa que calcula tantos números primos como el primer parámetro le indica. Las capturas de este código no están hechas en *minix* puesto que es demasiado largo, se salía de la pantalla y tras varios intentos fallidos de plasmarlo bien en el cuaderno de bitácora decidimos hacerlo desde otro entorno.

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

int main (int argc, char *argv[]) {
    if(argc!=2){
        printf("Must enter the number of primes desired.");
        exit(1);
    }

    int nprimos = atoi(argv[1]);
    if(nprimos<1){
        printf("Minimum number of primes to calculate is 1.");
        exit(1);
    }

    int *lista;
    int current=3, currentlista=3, i;
    bool primo;

    lista=(int*)malloc(nprimos*sizeof(int));
    if(lista==NULL){
        printf("Memory not allocated.");
        exit(1);
    }

    printf("Starting calculation of first %d prime numbers.\n", nprimos);
    time_t inicio=time(NULL);

    switch(nprimos){
        case 1:
            lista[0]=0;
            break;
        case 2:
            lista[0]=0;
            lista[1]=1;
            break;
        case 3:
            lista[0]=0;
            lista[1]=1;
            lista[2]=2;
            break;
        default:
            lista[0]=0;
            lista[1]=1;
            lista[2]=2;
            lista[nprimos-1]=0;

            while(lista[nprimos-1]==0){
                primo=true;
                i=2;
                while(primo && lista[i]<=(current/2)){
                    if((current%lista[i])==0)
                        primo=false;
                    i++;
                }
                if(primo){
                    lista[currentlista]=current;
                    currentlista++;
                }
                current++;
            }
    }

    printf("Calculation of first %d prime numbers finished. Last number calculated: %d. Seconds used: %d.\n",
        nprimos, lista[nprimos-1],(time(NULL)-inicio));
    free(lista);
}
```

```

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>

int main (int argc, char *argv[])
{
    pid_t *pids=malloc(4 * sizeof(pid_t));

    char **arg1=(char**)malloc(3*sizeof(char*));
    char **arg2=(char**)malloc(3*sizeof(char*));

    char *primos="primos";
    char *narg1="30000";
    char *narg2="110000";
    arg1[0] = malloc((strlen(primos) + 1) * sizeof(char));
    strcpy(arg1[0],primos);
    arg2[0] = malloc((strlen(primos) + 1) * sizeof(char));
    strcpy(arg2[0],primos);
    arg1[1] = malloc((strlen(narg1) + 1) * sizeof(char));
    strcpy(arg1[1],narg1);
    arg2[1] = malloc((strlen(narg2) + 1) * sizeof(char));
    strcpy(arg2[1],narg2);
    arg1[2]=NULL;
    arg2[2]=NULL;

    for(int i=0; i<4; i++){
        pids[i]=fork();
        if (pids[i]==0){
            if(i!=3){
                if(execvp(*arg1, arg1)<0) {
                    printf("\nExecvp number %d failed.\n", i);
                    exit(1);
                }
            }
            else{
                if(execvp(*arg2, arg2)<0) {
                    printf("\nExecvp number %d failed.\n", i);
                    exit(1);
                }
            }
        }
        if (pids[i]<0){
            printf("\nFork number %d failed.\n", i);
            exit(1);
        }
    }
}

```

Ante la pregunta sobre por qué llamar primero a la función las tres veces con el menor parámetro, encontramos la solución no solo en el código de la función *ready()* en el fichero *proc.c*, sino también en el mismo título de la fase: el efecto convoy. Al poner el largo en ese orden, aunque se comienza a ejecutar el primero, en lo que acaba, los demás tienen tiempo de comenzar y acabar. En caso de que el cuanto de tiempo fuese más largo, se podría observar el efecto convoy.



Continuamos convirtiendo en repartidor a la función *do\_esops()*, para que ahora llame a otras funciones según indique su primer argumento y creando *cambiarQ(m\_ptr->m1\_i2)* en el fichero *clock.c*. Para ello comenzamos creandola nueva variable externa *nuevoQ*.

```
#include "proc.h"

/* Constant definitions. */
#define MILLISEC 100 /* how often to call the scheduler (msec) */
#define SCHED_RATE (MILLISEC*HZ/1000) /* number of ticks per schedule */
PRIVATE int nuevoQ = SCHED_RATE; _
```

Aquí tenemos la función *cambiaQ(nq)*:

```
/*===== cambiaQ =====*/
*
*=====*/
PRIVATE void cambiaQ(nq) {

    nuevoQ = nq;

    sched_ticks = nuevoQ;
}
/*===== clock_task =====*/
*
*=====*/
PUBLIC void clock_task()
: wq
#
```

Asignamos ahora *nq* a la nueva variable, y ésta a *sched\_ticks*.

```
    }

    /* If a user process has been running too long, pick another one. */
    if (--sched_ticks == 0) {
        if (bill_ptr == prev_ptr) lock_sched(); /* process has run too long */
        sched_ticks = nuevoQ; /* reset quantum */
        prev_ptr = bill_ptr; /* new previous process */
    }
    #if (SHADOWING == 1)
        if (rdy_head[SHADOW_Q]) unshadow(rdy_head[SHADOW_Q]);
    #endif
}

/sched_ticks

if (--sched_ticks == 0) {
    /* If bill_ptr == prev_ptr, no ready users so don't need sched(). */
    sched_ticks = nuevoQ; /* reset quantum */
    prev_ptr = bill_ptr; /* new previous process */
}
return 1; /* Reenable clock interrupt */
}
:wq
# _
```

Después de estos cambios, creamos la imagen del sistema mediante *make hdboot* y reiniciamos la máquina para que se hagan efectivos. Ejecutamos el fichero *a.out* que se genera al compilar *launcher.c* y observamos que comienza el fichero largo y en lo que termina se ejecutan los cortos. Al aumentar el cuanto de tiempo a 1000 milisegundos pasamos a un sistema FIFO y podemos observar como el proceso más largo bloquea a los demás.

## Práctica 7

- 30/05/2020

Decidimos realizar la última parte del laboratorio, la práctica opcional. Para ello comenzamos creando la función *imprimeHoleHead()* en el fichero *alloc.c*. Esta función imprime el comienzo y el tamaño de cada uno de los segmentos de la lista.

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

PUBLIC void imprimeHoleHead(){
    struct hole = segmento = hole_head;
    while(segmento != NIL_HOLE){
        printf("Base %d.\n", segmento->h_base);
        printf("Tamaño %d.\n", segmento->h_len);
        if (segmento->h_next == NIL_HOLE)
            break;
        segmento = segmento->h_next;
    }
}
```

Sin embargo, este código nos llevó algo más del tiempo que esperábamos, y sumando esto a los hechos de que el comando *MemInfo* se nos estaba haciendo cuesta arriba y ambos teníamos exámenes finales el lunes, decidimos que era mejor terminar aquí.

La realización de este laboratorio ha sido un placer por lo interesante que ha sido, a pesar de la complicación de escribir código en *MINIX*. Muchas gracias.