Universidad de Valladolid

Escuela Ingeniería Informática Valladolid

UVa

ESCUELA DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA

MATERIA: ESTRUCTURA DE SISTEMAS OPERATIVOS

CUADERNO DE BITÁCORAS - LABORATORIO PARTE 2

AUTOR: Miguel Chaveinte García

GRUPO: T1

FECHA:20/05/2021

1. Práctica 3: Instalación y programación en MINIX

1.1. Día 1: Primer laboratorio - 08/04/2021

Resumen: Instalación de VirtualBox y creación de una máquina virtual Minix siguiendo los pasos de las instrucciones de la práctica.

Informe de situación: He tenido problemas a la hora de la instalación de Minix **EXPLICACIÓN**, era necesario habilitar la virtualización desde la BIOS del ordenador. Una vez hecho esto, se ha procedido exitosamente.

Tras comprobar que esta funciona correctamente, nos disponemos a cambiar el shell por defecto por el `ash`. Para ello tenemos que modificar el fichero `/etc/passwd`. Para hacer esto tan solo hay que modificar la línea donde se indica que se está usando `sh` por `ash`. Gracias al nuevo shell, ya funciona correctamente la función autocompletar al pulsar el tabulador.

Para crear un nuevo usuario lo primero que haremos será crear el directorio para este nuevo usuario. Lo hemos creado en `/usr/usuario`. Lo siguiente que hemos hecho ha sido entrar en el fichero `passwd` y hemos creado el usuario. Le hemos asignado el nombre usuario , le hemos asignado el Id de usuario 9 y el de grupo 3. También le hemos asignado su directorio y por último el shell `ash`. Por último le hemos asignado una contraseña con el comando `passwd`.

Modificación del mensaje de inicio modificando el archivo `/usr/src/kernel/tty.c`
Posteriormente hemos compilado el núcleo utilizando `make hdboot` en el directorio donde se encuentra la imágen del núcleo, en `/usr/src/tools/`.
Informe de situación: Todo ha procedido correctamente.

1.2. Día 2: Segundo laboratorio - 15/04/2021

He creado la carpeta `/root/Practica3` que usaré para trabajar en la parte de programación de esta práctica.Para hacer el programa, hemos creado el fichero `creaProcesos.c` adjunto captura del código que es el que el profesor mostró y explicó en clase:

```c
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <errno.h>
#include <stdarg.h>

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

        int numP;
        int i;
        int pid;

        /*Comprobarcion argc*/
        if(argc!=2){
                printf("Usar: CreaProcesos NumeroDeProcesos\n");
                exit(1);
        }
        numP=atoi(argv[1]);

        /*creo nump procesos hijo*/
        for(i=0;i<numP;i++){
                pid=fork();
                if(pid==-1){
```

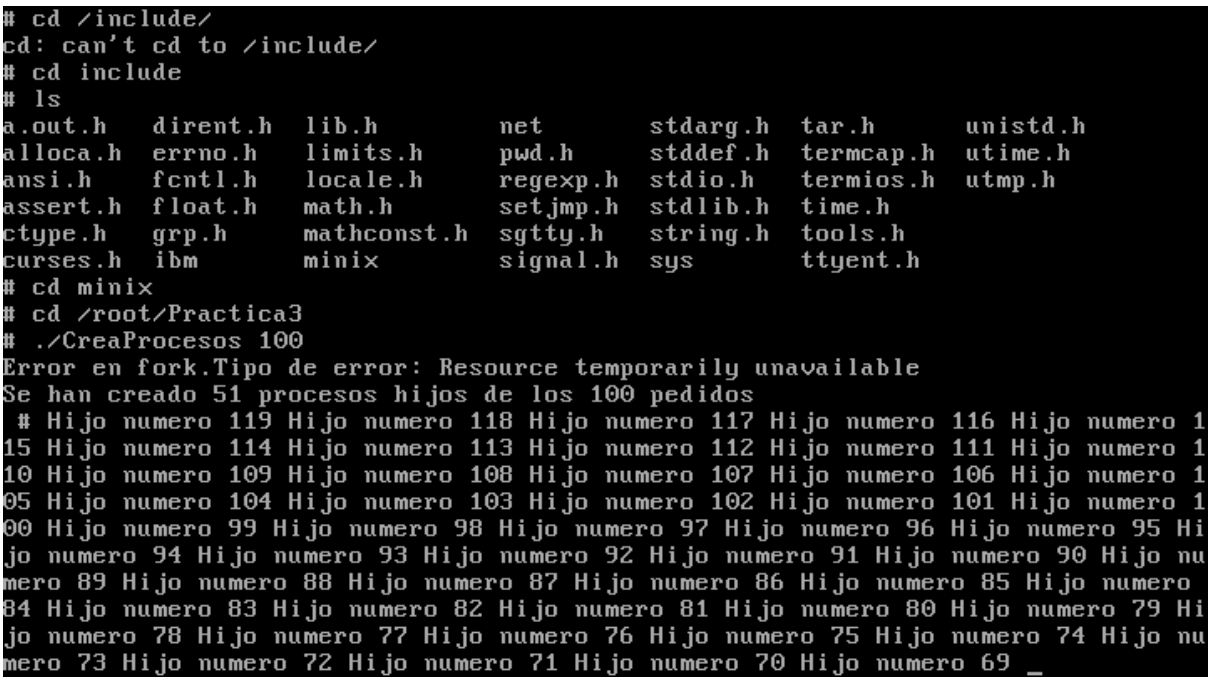
"CreaProcesos.c" 43 lines, 711 chars

```c
        /*creo nump procesos hijo*/
        for(i=0;i<numP;i++){
                pid=fork();
                if(pid==-1){
                        printf("Error en fork.Tipo de error: %s\n",strerror(errn
                        printf("Se han creado %d procesos hijos de los %d pedido
                        exit(1);
                }
                if(pid==0){
                        Hijo();
                        exit(0);
                }
        }
        for(i=0;i<numP;i++){
                wait(0);
        }
}

Hijo(){
        pid_t PIDHijo;
        PIDHijo=getpid();
        printf("Hijo numero %d ",PIDHijo);
}
```

Informe de situación: Hay un problema a la hora de compilar CreaProcesos.c, **EXPLICACIÓN** al trabajar con vi eliminamos uno de los include (`<sys/types>`) por lo que nos daba error. Tras introducirlo de nuevo y compilar el programa con `cc -o`, ejecutamos el programa que nos muestra que se han creado 23 procesos, aunque al ejecutarlo le hayamos pedido que nos ejecutara 60. **EXPLICACIÓN** el número de procesos viene limitado por el número NR_PROCS que viene en `<minix/config.h> (en /usr/include/minix)`, aunque en realidad los hijos que se nos permiten para nuestro programa son menores que ese número debido a que hay otros procesos del sistema que se están ejecutando. Ampliamos ese número a 64 y compilamos de nuevo el núcleo y

volvemos a ejecutar nuestro programa que ya nos permite un número mayor como se ve en la foto siguiente,(51 en nuestro caso ya que hay otros 14 del sistema de los 64 máximos).



1.3. Día 3: Tercer laboratorio - 22/04/2021

Estudiamos en clase el pdf El núcleo de Minix I, y cómo se producen el paso de mensajes entre las diferentes capas.

Además ese día, intento realizar el programa LeeAout, pero solo hago lo de la lectura del argumento del fichero ya que no termino de entender lo de la cabecera y los campos.

1.3. Día 4: Cuarto laboratorio - 29/04/2021

El profesor explica en clase el concepto de cabecera y remata el ejercicio. Que básicamente incluimos el fichero a.out el cual contiene un struct exec que contiene todos los datos acerca de los ficheros ejecutables.

```c
#include <a.out.h>
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <string.h>

/* LeeAout fichero_ejecutable */

main(int argc,char *argv[]){
        struct exec pp;
        FILE *fd;
        int valor;

        if(argc!=2){
                printf("Usar: LeeAout fichero_ejecutable\n");
                exit(0);
        }
        fd=fopen(argv[1],"r");
        if(fd==NULL){
                printf("Error en fichero %s %s\n",argv[1],strerror(errno));
                exit(0);
        }
        valor=fread(&pp,sizeof(struct exec),1,fd);
        if(valor!=0){ /* ahora puedo imprimiir cualquier campo de struct exec */
```

```c
/* LeeAout fichero_ejecutable */

main(int argc,char *argv[]){
        struct exec pp;
        FILE *fd;
        int valor;

        if(argc!=2){
                printf("Usar: LeeAout fichero_ejecutable\n");
                exit(0);
        }
        fd=fopen(argv[1],"r");
        if(fd==NULL){
                printf("Error en fichero %s %s\n",argv[1],strerror(errno));
                exit(0);
        }
        valor=fread(&pp,sizeof(struct exec),1,fd);
        if(valor!=0){ /* ahora puedo imprimiir cualquier campo de struct exec */
                printf("Tam. del codigo %d-B\n",pp.a_text);
                printf("Tam. de datos %d-B\n",pp.a_data);
                printf("Dir. de la primera instruccion codigo maquina: 0x%x\n",p
        }
}
```

2. Práctica 4: Llamadas al sistema

2.1. Día 4: Cuarto laboratorio - 29/04/2021

Leemos la práctica 4 y nos vamos desplazando por los diferentes ficheros para entender las llamadas al sistema.

Seguimiento completo de una llamada fork();

Modo Usuario

1º- (MU) Cuando invocamos desde un programa la llamada al sistema fork() (posix/_fork.c), se ejecuta una llamada a la función _syscall(MM, FORK, &m).

EXPLICACION:

Es interesante señalar que la función fork() devuelve un valor del tipo pid_t, que es el PID del hijo que se da al padre al ejecutar la llamada.
Observando el código de otras llamadas al sistema como wait(), podemos ver que en todas ellas se sigue el patrón de llamada a _syscall(MM, *, &m), por lo que * es el argumento más importante de este paso, ya que es el que identifica el tipo de llamada al sistema que estamos realizando (mediante las constantes alfanuméricas de callnr).
El parámetro m se declara de tipo message en la función fork();

2º- (MU) Esta función invoca a su vez a _sendrec().

EXPLICACION:

El parámetro m, ya declarado en fork(), se empieza a definir en la función _syscall(), llenando el campo m_type(El tipo de datos message corresponde a una estructura que se basa en dos variables: m_source, que indica quién manda el mensaje y m_type, que indica qué tipo de mensaje es).En este caso el tipo de mensaje definido es FORK.
A la función _sendrec() se le pasan como parámetros el servidor que ha de tratar la llamada al sistema (MM) y el mensaje con el tipo ya definido.

3º- (MU) Desde _sendrec() lanza la interrupción software int SYSVEC,almacenando en un registro el número 33 como argumento para la llamada al sistema, la cual hace que a partir de este punto el procesador cambie a modo privilegiado.

EXPLICACION:

El compilador de C guarda en la pila los argumentos de llamada a una función, por eso la función en ensamblador sendrec guarda esos valores en los registros de propósito general sacándolos de la pila y después realiza la interrupción (que

utilizará los valores de estos registros para definir cómo se ha de tratar la interrupción).

Modo Privilegiado

4º- El sistema pasará al tratamiento de la interrupción para ello la función prot_init() inicializa la tabla de vectores de interrupción con el argumento 33 (el valor de Sysvec). La recepción de la interrupción está asociada con la función s_call().

EXPLICACION:

idt es una variable global declarada en protect.c como un vector de struct gatedesc_s de tamaño IDT_SIZE. Siguiendo la pista a IDT_SIZE lo encontramos definido en protect.h con el valor IRQ8_VECTOR + 8, estando esta definida en const.h con el valor 48 (0x30), por lo que el vector tiene tamaño 56.
En la función int_gate() es donde se referencia a idt[33] (el argumento formal vec_nr tomar el valor 33) asignándole la correspondencia con s_call.

Comprobando la línea 206 del archivo /usr/src/kernel/protect.c con el nuevo comando añadido a la lista de Vi, observamos que el vector SYS386_VECTOR esta asociado con la función s_call().

5º- La función s_call() invoca a la función sys_call(), cuyos argumentos habían sido insertados previamente en la pila.

6º- La función sys_call() es la encargada de enviar el mensaje al servidor apropiado en el ejemplo del fork() el servidor es MM y el mensaje es FORK (Este FORK es una constante que representará la llamada a la función que tratará a fork() en call_vec[]).

EXPLICACION:

La función sys_call() recibe tres parámetros, el más relevante es el primero de ellos "function", que determina si se va a enviar un mensaje, recibir un mensaje o ambos (send, receive o sendrec). En nuestro caso llega sendrec.
La función que se invoca para enviar el mensaje al servidor es mini_send(), la cual se encarga de comprobar que el receptor del mensaje está esperando a recibir dicho mensaje, tras enviar el mensaje, bloquea el proceso que envía el mensaje. La función que se invoca para recibir el mensaje es mini_rec(), que comprobará que tiene un mensaje para recibir y desbloquea el emisor de este mensaje.

7º- Una vez que sys_call() ha enviado el mensaje a MM, se ejecuta _restart() en mpx386.s. Esta función permitirá continuar con la ejecución de otro proceso, normalmente en nuestro caso será MM ya que ha de tratar el mensaje que previamente le había mandado sys_call(), en este punto el procesador vuelve de nuevo a modo usuario (MU)

2.2. Día 5: Quinto laboratorio - 06/05/2021

Modo Usuario

8º- El main del MM que estaba a la espera de recibir un mensaje ejecutando en un bucle infinito la función get_work(), una vez ha recibido el mensaje se invoca a la función que ha de tratarlo a través de un vector de funciones llamado call_vec[], en nuestro caso call_vec[2] = do_fork, esta será la función encargada del tratamiento de la llamada al sistema.

EXPLICACION:

En el main de `usr/src/mm/main.c` se declaran e inicializan las variables que usará el gestor de memoria llamando a la función mm_init() (podemos ver que ésta inicialización se realiza con el arranque del sistema porque están escritos los printf con información de la memoria que se muestran en el login). Después de esta inicialización el gestor entra en un bucle infinito: lo primero que hará será llamar a la función get_work, que intentará leer un mensaje mediante la función receive y en caso de conseguirlo lo "decodificará" definiendo los nuevos valores para who y mm_call. Esta última variable será especialmente importante ya que almacena el tipo de mensaje enviado (call number) que en nuestro caso, recordemos, será FORK = 2. Este valor es un "call number" válido por lo que se ejecutará (*call_vec[mm_call])(). En este punto debemos buscar dónde se encuentra inicializado el vector call_vec[], por lo que hacemos "grep call_vec *" en el directorio mm en el que nos encontramos: el resultado es table.c. En este fichero vemos el vector y buscamos la posición 2 correspondiente al mensaje de FORK, que nos conduce a la función do_fork. De nuevo hacemos "grep do_fork *" para buscar donde se encuentra esta función y obtenemos como resultado el fichero forkexit.c. Modificamos esta función para que salude introduciendo un printf después de la declaración de las variables. Una vez introducido el printf, recompilamos el núcleo con `make hdboot` y reiniciamos el sistema. Ahora cada vez que ejecutamos cualquier comando, el gestor de memoria nos saluda porque se realiza un fork(), ya que para satisfacer la ejecución de los comandos que introducimos en la terminal esta genera hijos para hacerlo. Muestro las imágenes de la modificación y tras su posterior compilación como aparece el mensaje.

```
PRIVATE pid_t next_pid = INIT_PID+1;    /* next pid to be assigned */

FORWARD _PROTOTYPE (void cleanup, (register struct mproc *child) );

/*===========================================================================*
 *                                do_fork                                    *
 *===========================================================================*/
PUBLIC int do_fork()
{
/* The process pointed to by 'mp' has forked.  Create a child process. */

  register struct mproc *rmp;   /* pointer to parent */
  register struct mproc *rmc;   /* pointer to child */
  int i, child_nr, t;
  phys_clicks prog_clicks, child_base = 0;
  phys_bytes prog_bytes, parent_abs, child_abs; /* Intel only */

  /*MODIFICA:6-5*/

  printf("MM:do_fork\n");


 /* If tables might fill up during FORK, don't even start since recovery half
_ * way through is such a nuisance.
```

```
MM:do_fork
MM:do_fork
MM:do_fork


Minix  Release 2.0 Version 0

minix2.0.0 login: root
MM:do_fork
MM:do_fork
# ls
MM:do_fork
.ellepro.b1  .profile  boot  etc  fd1    mnt   tmp
.exrc        bin       dev   fd0  minix  root  usr
# pwd
MM:do_fork
/
# cd /usr/src/mm
# ls
MM:do_fork
Makefile  break.o  forkexit.c  glo.h   mm.h     putk.c    table.c  type.h
alloc.c   const.h  forkexit.o  main.c  mproc.h  putk.o    table.o  utility.c
alloc.o   exec.c   getset.c    main.o  param.h  signal.c  trace.c  utility.o
break.c   exec.o   getset.o    mm      proto.h  signal.o  trace.o
# _
```

Una vez comprobado que el mensaje salía y por tanto la modificación era correcta, eliminamos el printf y recompilamos.

Informe de situación: Ha habido un primer problema al compilar el núcleo dando un error en el archivo forkexit.c el cual decía que la constante LAST_FEW no estaba definida.
EXPLICACIÓN al modificar el archivo forkexit.c eliminé la línea donde definía dicha constante. Tras volverla a poner compila, y muestra el mensaje sin problema.

3. Práctica 5: Programación de Llamadas al Sistema en Minix

3.1. Día 5: Quinto laboratorio - 06/05/2021

Lo primero que debo hacer para la creación de esta nueva llamada es añadirla al fichero de las listas de llamadas al sistema, definiéndolo con el número 77 con el nombre ESOPS e incrementando el número de ellas a 78.

Posteriormente debo introducir en el vector de llamadas la rutina asociada a la llamada 77 que en este caso la he llamado do_esops, añadiendolo al final del todo después de do_reboot.

También debo acceder para modificar el fichero de la función de prototipos al final del fichero de forma análoga a do_reboot , pero en este caso poniendo como función do_esops y sin argumentos.

Ahora abro con vi el archivo utility.c e implemento la función al final de este. Utilizo el código proporcionado en el guión de la práctica. Esta función envía un mensaje a la tarea que se identifica con la constante con SYSTASK, la cual está definida como -2.

Informe de situación: De momento todo sin problemas. Se verá si los hay cuando se compile el núcleo.

3.2. Día 6: 12/05/2021

Continuación con Práctica 5. Abro con vi el fichero /usr/src/kernel/system.c y añado el prototipo de nuestra función después de los demás prototipos FORWARD _PROTOTYPE(int do_esops, (message *m_ptr)); Después añado nuestra llamada al sistema en el switch de sys_task(): case ESOPS: r = do_esops(&m); break;

Aunque tengamos dos funciones con el mismo nombre, no pasa nada porque cada una está en una capa distinta. Escribo la función do_esops utilizando el código proporcionado en el guión. Ahora como se están utilizando punteros tenemos que fijarnos en utilizar ->. El código proporcionado en el guión hay que modificarlo. Para empezar tiene un fallo sintáctico en el que utiliza : en vez de ;. Además al final de la función cambio los valores contenidos en la estructura de memoria para comprobar que se ha recibido en el nivel 4. Por último, compilo el núcleo, sin problemas tras la ejecución.

Paso a la fase del proceso de usuario: En mi directorio de trabajo (/root/Practica5) creo el fichero practica5.c, incluyo todas las librerías necesarias: además de las especificadas en el guión, incluyo también stdio.h, declaro una estructura mensaje e inicializo sus valores arbitrariamente. Usando _taskcall(MM, ESOPS, &msj) se realiza la llamada al sistema. Se pasa la dirección memoria del mensaje porque no hay comunicación directa entre niveles. Después se imprimen los campos del mensaje para comprobar que han sido modificados

por el kernel. Compilo y ejecuto. **EXPLICACIÓN :** Todo según lo previsto el valor msj.m1_i1=1 cambia al valor que habíamos puesto en la tarea del sistema do_esops.

3.2. Día 6: 13/05/2021

Primero añadÍa usr/include/minix/callnr.h el nombre de la función que quería implementar, en mi caso " PCBINFO " con el número 78, y también incrementé el número total de llamadas disponibles NCALLS, a 79 con esta nueva llamada.

Después, ya en /usr/src/kernel/system.c creamos una función do_pcbinfo() de prueba, que simplemente escribía en pantalla un mensaje, para comprobar que todo funcionaba.

EXPLICACIÓN: Después modifiqué la función do_esops que creamos para los puntos anteriores, exactamente la situada en /usr/src/kernel.c, ya que las funciones y cambios que hice en el gestor de memoria anteriormente valían para esta parte de la práctica, y que solo había que añadir alguna sentencia, en nuestro caso switch que comparara m1_i1 con las constantes introducidas en callnr de las nuevas llamadas que queramos crear, que nos permitiera acceder a las nuevas llamadas al sistema fácilmente, así pues, la función esops quedó:

```
/*===========================================================================*
 *                                do_esops                                   *
 *===========================================================================*/
PRIVATE int do_esops(m_ptr)
register message *m_ptr;
{
        int a2,a3;
        a2 = m_ptr->m1_i2;
        a3 = m_ptr->m1_i3;

        switch(m_ptr->m1_i1){
                case PCBINFO:   do_pcbinfo(a2,a3);
                                break;
                default: return(E_BAD_FCN);
        }

        return(OK);
}

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

En nuestro ejemplo, si la llamada es correcta, se puede ver que el switch comparará el primer campo del mensaje (por eso m_ptr apunta a m1_i1, siendo este último una macro), si no es correcta, retornará un error.

Después hicimos los cambios en practica5.c, situado en nuestro directorio root/Practica5

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

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

        message msj;

        msj.m1_i1=PCBINFO;

        msj.m1_i2=2;
        msj.m1_i3=3;

        _taskcall(MM,ESOPS,&msj);

        printf("Practica5: %d,%d,%d\n",msj.m1_i1,msj.m1_i2,msj.m1_i3); /*deberia
}
~
~
~
~
~
 "practica5.c"  19 lines, 346 chars
```

Después nos movimos a /usr/src/tools, y compilamos el núcleo. Ahora al compilar practica5.c y después ejecutarlo nos salió por pantalla el mensaje que habíamos puesto en la función do_pcbinfo, así que ya estamos seguros de que los cambios en do_esops funcionan.

EXPLICACIÓN: Por último, modifico la función do_pcbinfo, para que diera información acerca del proceso, ya que, hasta ahora, esta función simplemente la había hecho para comprobar que los cambios de esops funcionaban. Para ello, primero quise comprobar la estructura de proc_ptr y de bill_ptr, al hacerlo pude observar que ambas eran de tipo struct "proc", para ver proc me moví al fichero /usr/src/kernel/proc.h, donde al comprobarlo, pude observar dos campos interesantes para mostrar, por un lado el entero p_nr que dice el número del proceso, y por otro lado p_pid, que dice el PID del proceso en cuestión. Al final la función do_pcbinfo() quedó así:

```
/*===========================================================================*
 *                                do_pcbinfo                                 *
 *===========================================================================*/
PRIVATE int do_pcbinfo(int arg1,int arg2){

        struct proc *tmp=bill_ptr;
        printf("hola soy pcbinfo con argumentos : %d , %d\n",arg1,arg2);
        printf("Numero de proceso: %d, PID: %D\n",tmp->p_nr,tmp->p_pid);

}




/*===========================================================================*
 *                                do_esops                                   *
 *===========================================================================*/
PRIVATE int do_esops(m_ptr)
register message *m_ptr;
{
```

Al compilar el núcleo y ejecutar practica5 mostró por pantalla que el número de proceso era el 12 y el PID era 51, tal como tenemos en do_pcbinfo.

3.2. Día 7: 20/05/2021

El profesor nos hace un repaso por la práctica 5, lo que hace que modifique cosas en mi código para adaptarlo y mejorarlo; además de incluir también el distribuidor en el gestor de memoria en el archivo utility.c .

Además en el proceso de usuario (/root/Practica5/practica5.c) hacemos que la llamada sea pasada como argumento cuando se ejecute lo que permitirá poder implementar más llamadas.

Posteriormente,realizo el distribuidor en utility.c como se puede ver en la imagen y posteriormente en la llamada de do_pcbinfo añado lo que se explico en clase, previa comprobación del slot está libre,imprimo más información sobre los procesos que se encuentran en el pcb, su nombre y su pid. (En clase se imprimió más información sobre las direcciones virtuales de, fisicas y longitud de la pila,datos,código…).

Añado las imágenes del proceso de usurio, de do_esops en memoria y kernel:

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

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

        message msj;

        if(argc==1){
                printf("Warning : Write ./practica5 PCBINFO\n");
        }

        if(strcmp(argv[1],"PCBINFO")==0){
                msj.m1_i1=PCBINFO;
                msj.m1_i2=2;
                msj.m1_i3=3;
        }

        _taskcall(MM,ESOPS,&msj);

        printf("Practica5: %d,%d,%d\n",msj.m1_i1,msj.m1_i2,msj.m1_i3); /*deberia
}
```

```
Wrote "practica5.c"  24 lines, 456 characters
```

```c
*                               do_esops
 *===========================================================================*/
 PUBLIC int do_esops() {
        int tipo,arg1,arg2;
        tipo= mm_in.m1_i1;
        arg1 = mm_in.m1_i2;
        arg2 = mm_in.m1_i3;

        /* Mensaje para comprobar el funcionamiento */
        printf("MM:do_esops: %d %d %d\n",tipo,arg1,arg2);

        switch(tipo){
        case PCBINFO:
                do_pcbinfo(arg1,arg2);
                break;
        default:
                printf("mm:do_esops:tipo desconocido:%d\n",tipo);
        }

        /* 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,e
        /* result2 es una variable externa que se coiara en mm_out.m1_i1*/
```

```
        /* Mensaje para comprobar el funcionamiento */
        printf("MM:do_esops: %d %d %d\n",tipo,arg1,arg2);

        switch(tipo){
        case PCBINFO:
                do_pcbinfo(arg1,arg2);
                break;
        default:
                printf("mm:do_esops:tipo desconocido:%d\n",tipo);
        }

        /* 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,e
        /* result2 es una variable externa que se coiara 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;
}

PUBLIC int do_pcbinfo(int arg1,int arg2){
        printf("mm: argumentos %d,%d\n",arg1,arg2);
}
```

```
/*===========================================================================*
 *                              do_pcbinfo
 *===========================================================================*/
 PRIVATE int do_pcbinfo(int arg1,int arg2){

        int i;
        int NUso=0;
        struct proc *tmp=bill_ptr;

        printf("k: pcbinfo con argumentos : %d , %d\n",arg1,arg2);
        /* recorremos el pcb*/
        for(i=0;i<NR_PROCS+NR_TASKS;i++){
                if(!(proc[i].p_flags & P_SLOT_FREE)){/* slot en uso */
                        NUso++;
                        printf("Nombre:%s - , PID: %d",proc[i].p_name,proc[i].p_
                }
        }/* bucle for */

        printf("Entradas usadas :%d - Totales: %d\n",NUso,NR_PROCS+NR_TASKS);
        printf("Numero de proceso: %d, PID: %D\n",tmp->p_nr,tmp->p_pid);

 }

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

```
# cd /usr/src/tools
# make hdboot
cd ../kernel && exec make -
make: 'kernel' is up to date
cd ../mm && exec make -
exec cc -c -I/usr/include utility.c
"utility.c", line 131: (warning) implicit declaration of function do_pcbinfo
exec cc -o mm -i main.o forkexit.o break.o exec.o \
        signal.o alloc.o utility.o table.o putk.o trace.o getset.o
install -S 256w mm
cd ../fs && exec make -
make: 'fs' is up to date
installboot -image image ../kernel/kernel ../mm/mm ../fs/fs  init
    text    data     bss     size
   38656    6752   59860   105268  ../kernel/kernel
   12704    1268   39560    53532  ../mm/mm
   28576    2204 4344684  4375464  ../fs/fs
    6828    2032    1356    10216  init
  ------  ------  ------  -------
   86764   12256 4445460  4544480  total
exec sh mkboot hdboot
rm /dev/hd1a:/minix/2.0.0r22
cp image /dev/hd1a:/minix/2.0.0r23
Done.
# reboot
```

```
minix2.0.0 login: root
# cd /root/Practica5
# cc -o practica5 practica5.c
"practica5.c", line 15: (warning) implicit declaration of function strcmp
# ./practica5
practica5 PCBINFO
MM:do_esops: 0 0 0
mm:do_esops:tipo desconocido:0
Practica5: 0,0,0
# ./practica5 PCBINFO
MM:do_esops: 78 2 3
mm: argumentos 2,3
k: pcbinfo con argumentos : 2 , 3
Nombre:TTY - , PID: 0Nombre:WINCH - , PID: 0Nombre:SYN_AL - , PID: 0Nombre:IDLE
- , PID: 0Nombre:PRINTER - , PID: 0Nombre:FLOPPY - , PID: 0Nombre:MEMORY - , PID
: 0Nombre:CLOCK - , PID: 0Nombre:SYS - , PID: 0Nombre:HARDWAR - , PID: 0Nombre:M
M - , PID: 0Nombre:FS - , PID: 0Nombre:INIT - , PID: 1Nombre:ash - , PID: 32Nomb
re:update - , PID: 21Nombre:getty - , PID: 33Nombre:getty - , PID: 34Nombre:gett
y - , PID: 35Nombre:getty - , PID: 36Nombre:getty - , PID: 37Nombre:getty - , PI
D: 38Nombre:getty - , PID: 39Nombre:practica5 - , PID: 50Entradas usadas :23 - T
otales: 74
Numero de proceso: 12, PID: 50
Practica5: 78,2,3
#
```

Informe de situación: **EXPLICACIÓN** Al intentar compilar (se ha hecho cada vez que modificaba un fichero) nos han surgido errores porque faltaba una ; , y otra vez porque en el archivo utility.c que no habíamos definido en él la función do_pcbinfo. Tras esto compila y funciona como se ve en las imágenes anteriores.