Daniel González Pérez
Guillermo Cebrián Serrano
Grupo 3

16/02/16

Minix instalado sin problemas siguiendo las instrucciones.

Creado un usuario (con vi en /etc/passwd)
<ID:password:UID:GID:nombre_de_usuario:carpeta_usuario:shell>
carpeta_usuario: /usr/<ID> (Carpeta a la que accede al iniciar)
shell: /usr/bin/ash
el password se puede cambiar con: passwd <ID>

Cambiar el Mensaje de inicio.
Fichero: /usr/src/kernel/tty.c
Función: tty_task() (Buscar con /tty_task)
Mensaje cambiado -> compilar el Kernel.

make hdboot compila el Kernel si se ejecuta en usr/src/tools.
Entre los procesos ejecutados estan:
install -S 0 kernel
installboot -image image ../kernel/kernel ../mm/mm ../fs/fs init
cp image /dev/hd1a:/minix/2.0.0r13
(Lo que indica que se ha compilado correctamente)

Al reiniciar el mensaje editado aparece.
Al cambiar el inicio al disquete, se inicia como en la version original de Minix.
El mensaje editado no aparece pero el nuevo usuario sigue existiendo.
Tampoco pide la tecla ¡ para iniciar.

Añadido un mensaje de inicio para indicar que inicia de Disco Duro.

Domingo 21/02/16

Creada la segunda maquina virtual Minix (Sin uso alguno actualmente)

Martes 23/02/16

Creado el archivo "hijos.c" en "/root/PracticaA" que crea la cantidad de hijos dada en el argumento al llamarla.
Compilado con: cc -o hijos hijos.c
Llamada a la funcion: ./hijos <Numero_de_hijos>
Sobre hijos.c:
- Se asegura que solo se use 1 argumento en la llamada.
- Pasa el argumento a int con "atoi(<argumento>)"
- Con un bucle for, tantas veces como diga el argumento, crea un hijo y guarda su ID en un int.

Añadido una linea para indicar el numero maximo de hijos cuando falle hijos.c.

Práctica B

Cuando se llama a fork, se usa un _syscall(MM,FORK,&m)
Donde:

-MM: Identificador del gestor de memoria. (Destinatario de la llamada al sistema)
-FORK: Tipo de llamada sobre MM.
-&m: Direccion de memoria donde esta el resto de los argumentos de la llamada.

A su vez, _syscall() llama a _sendrec(who, msgptr), la cual envia un mensaje al servidor MM.
_sendrec usa un int "SYSVEC", que es la interrupcion software.
El argumento de la llamada es 33
(Linea 8 : "SYSVEC = 33")

RESUMEN:
syscall --> sendrec --> SYSVEC

A partir de aqui el procesador cambia a modo privilegiado.

La funcion "prot_init()" de src/kernel/protect.c inicializa la tabla de vectores de interrupcion.
Ejemplo: Vector de interrupcion "SYS386_VECTOR"
```
int_gate(SYS386_VECTOR, (phys_bytes) (vir_bytes) s_call,
      PRESENT | (USER_PRIVILEGE << DPL_SHIFT) | INT_GATE_TYPE);
```

En src/kernel/const.h se define esta constante con el valor indicado anteriormente (33)
"#define SYS386_VECTOR 33   /*except 386 system calls use this*/"

El vector interrupcion llama a s_call, que es una funcion de "src/kernel/mpx386.s"
A su vez, s_call llama a sys_call():
```
call  _sys_call  !sys_call(function,src_dest,m_ptr)
```
Y sus argumentos han sido previamente insertados en la pila:
```
push  ebx   !pointer to user message    (Message Pointer/m_ptr)
push  eax   !src/dest              (src_dest)
push  ecx   !SEND/RECEIVE/BOTH          (function)
```

sys_call es una funcion de "src/kernel/proc.c" y es la implementacion de la llamada al sistema.
Nota. sys-call funciona a base de SEND y RECIEVE.

Domingo 28/02/16
---
s_call llama a _restart, metodo de src/kernel/mpx386.s
_restart elimina las interrupciones actuales y las reactiva.
El procesador cambia de modo privilegiado a modo usuario.

MM: Código gestor de memoria, almacenado en /usr/src/mm/main.c, metodo main()
Añadida la sentencia "System.out.printf("Gestor de memoria!")" debajo de:
```
if (mm_call== EXEC && error == OK) continue;
```
Para que solo la imprima si no hay errores.
Al compilar da un error.

Lunes 29/02/16

Error solucionado, se ha borrado la linea y se ha escrito:
"printf("Llamada al sistema!\n")"
encima de:

return(next_pid);
Compila y funciona correctamente.

Martes 01/03/16

Práctica C
> Creación de una llamada al sistema.
+ Modificaciones en el nucleo:
- Gestor de memoria:
En /usr/include/minix/callnr.h añade la llamada 77:ESOPS
y en la primera linea "#define NCALLS 77 /*number of system calls allowed*/
se aumenta la capacidad a 78.

Funcion main() de usr/src/mm/main.c invoca una funcion del vector call_vec.
"error = (*call_vec[mm_call])();" (mm_call: dirección de la función)
(call_vec[] está en /mm/table.c)

Se añade:
do_esops, /* 77 = nueva llamada ESO*/
Para añadir la llamada en table.c
Para añadirla a proto.h se escribe la linea:
_PROTOTYPE(int do_esops, (void));

Se añade la funcion do_esops() en utility.c tal como se da el ejemplo en la practica.
(¡Cuidado con la línea de _taskcall, cambiar ASOPS por ESOPS (nuestra función)!)

- Tarea de sistema:
En el archivo /kernel/system.c se añaden:
- "FORWARD _PROTOTYPE(int do_esops, (message *m_ptr));" en la lista de prototipos.
- "case ESOPS: r = do_esops(&m);   break;" en el switch de sys_task()
- La función do_esops, que imprime en la consola el argumento dado.

```
PRIVATE int do_esops(m_ptr)
register message *m_ptr;
{
int m = m_ptr->m1_i1;
printf("%d",m);
return(OK);
}
```

*******Comentado el printf de los fork, para que no moleste********

Sabado 05/03/16

- Crear un proceso de usuario
(Programa en C que llame a _taskcall(MM,ESOPS,&msj))
Creado en root/PracticaC.
Resumen del programa:
-Se crea un mensaje: "message men;"
-Se le da el dato: "men.m1_i1=atoi(argv[1]);"
-Se llama a taskcall: "_taskcall(MM,ESOPS,&men);"

Lunes 07/03/16

!!! No funcionaba correctamente debido a la funcion do_esops, cambiada:
int m = m_ptr->m1_i1;
a:
m_ptr->m1_i1=99;
Para que transforme m1_i1 a 99, sea cual sea el dato que tenga.

Martes 08/03/16

Práctica D

usr/src/kernel/proc.c ---> Maneja los procesos y llamadas a funciones.
-[300]pick_prock(): Decide que proceso ejecutar.
-[331]ready(rp): Añade rp al final de una cola de procesos a ejecutar.
  --> TASK_Q: (Prioridad alta) tareas ejecutables.
  --> SERVER_Q (Prioridad media) MM y FS.
  --> USER_Q (Prioridad baja) procesos de usuario.
-[390]unready(rp): Bloquea rp.
-[461]sched(): El proceso en ejecucion lleva demasiado tiempo.
  Si otro proceso de usuario se puede ejecutar, se pone el actual al final de
  la cola.

src/kernel/clock.c ---> Codigo de la tarea del reloj.
-[103]clock_task(): Programa principal del reloj.
-[145]do_clocktick(): Añade un tick (Para cuando se necesita hacer mucho trabajo).
-[382]clock_handler(irq): Se ejecuta en cada tick.
  Hace algo de trabajo para que clock_task no necesite ser llamado en cada tick.
-[494]init_clock(): Inicializa el reloj.

En proc.c:
-[49]interrupt(task): Interrumpe task.

Domingo 13/03/16

(Vuelta a la Practica C)
Crear una nueva funcion a la que llama esops:
  - En usr/include/minix/callnr.h:
      Se añade "INFO   500"
  - En usr/src/kernel/system.c:
      Se edita do_esops para añadir un (distribuidor) switch para detectar que función llama.
      ****
      int v1 = m_ptr->m1_i1;
      switch(v1){
            case(INFO):
              do_info();
              break;
      ****
      Y creamos la función do_info que imprima el nombre del mensaje
      "printf("%s\n",proc_ptr->p_name);"
Al ejecutar el programa practicaC.c con el dato 500 (ID de INFO) imprime el nombre del proceso en ejecucion (SYS).

Martes 15/03/16

Para ampliar la informacion que da do_info() se ha editado:
  "printf("Nombre: %s\n",proc_ptr->p_name);"
  "printf("PID: %d\n",proc_ptr->p_pid);"
  "printf("Nombre de usuario: %d\n",proc_ptr->user_time);"

_____ Practica D_____
(Para info sobre las funciones mirar el Martes 08/03/16)
usr/src/kernel/clock.c -----> Codigo del reloj.
-[63]TIMER_COUNT ((unsigned)(TIMER_FREQ/HZ)): Valor inicial del contador.
-[64]TIMER_FREQ (1193182L): Frequencia del reloj.

usr/include/minix/const.h ----> Constantes de Minix.
-[15]HZ (60) Frecuencia del reloj.
FRECUENCIA: 60 HZ -> 60 interrupciones x segundo.
TICK: 1/FREC --> Tiempo entre interrupciones.

_____Practica C_____
Añadida la funcion PPMM con ID 501.
Añadido un distribuidor en usr/src/mm/utility.c (do_esops) para las funciones.

NOTA: Distribuidores en:
usr/src/mm/utility.c (do_esops())
usr/src/kernel/system.c (do_esops())

Martes 05/04/16

_____Practica C_____
El distribuidor del gestor daba un error porque llamaba a las funciones do_info() y do_ppmm(), las cuales estan
en el nucleo, no en el gestor de memoria. Dado que no estaban en el gestor, no habia funciones a las que llamar.
_____Practica D_____
usr/src/kernel/clock.c
-[187]sched_ticks = SCHED_RATE   (Ticks restantes del proceso en ejecucion para agotar el quantum)
-[55]SCHED_RATE (MILLISEC*HZ/1000)
<Empezando practicaD>

Sabado 09/04/16

Creado el programa C para calcular primos.
Crea 3 procesos ligeros y uno pesado. Aún siendo el último en el código, el proceso pesado se inicia el primero
y se termina el último.

_____ Añadir una llamada de sistema para cambiar el Quantum_____
Se crea la llamada al sistema en include/minix/callnr.h (CAMQ 502)
En kernel/system.c se añade CAMQ al distribuidor:
```
case CAMQ:
      cambiaQ(m_ptr->m1_i2);
      break;
```
en clock.c:
- Se añade un "int nuevoQ = SCHED_RATE" como variable para cambiar el quantum.
- Se añade la funcion a la lista:
```
      FORWARD _PROTOTYPE( void cambiaQ, (int nq) );
```
- Se crea la función:

```
PUBLIC void cambiaQ(nq){
      nuevoQ = nq;
      sched_ticks = nuevoQ;
      printf("Quantum cambiado a %d",nuevoQ");
}
```
NO COMPILA (cambiaQ undefined): Hay que borrar el FORWARD _PROTOTYPE

Editado el programa PracticaC.c (El que llama a esops) para usar 2 argumentos en vez de 1.
Al cambiar el quantum a 50000 el proceso pesado empieza y acaba el primero.

Martes 12/04/16

_____Práctica E: Estructuras de descripción de la memoria_____

usr/src/mm/alloc.c tiene la lista de los "agujeros" (holes) de memoria libre.
Se crea una función para que imprima la lista de los agujeros (print_hh()):
-Se carga el agujero con: register struct hole *hp;
-Se imprimen los datos:
      Base: hp->h_base (En que posición de memoria empieza)
      Tamaño: hp->h_len (En clicks. Se divide entre 4 para darlo en KB)
-Se pasa al siguiente agujero con : hp=hp->h_next;

(Creado un comando [quantum <x>] para cambiar el quantum. Para ello se crea y compila
el programa en /bin)

Añadida la función al Distribuidor del Gestor de Memoria, a mm/main.c/mm_init() y creado
un comando (MemLib) como en la práctica anterior.
(Para ello también se crea otra llamada al sistema, MEMLIB 503)

Jueves 14/04/16

Para imprimir la tabla de procesos se usa la estructura mproc del gestor de memoria.
mproc contiene los procesos y la información de la memoria que usan, mediante estructuras mem_map.

Para hacer la función que imprima la tabla, se hace un bucle:
  for(procNum=0;procNum<NR_PROCS;procNum++)
Se carga el proceso de la tabla de procesos:
  struct mproc *aux;
  aux=&mproc[procNum];
Se comprueba si el proceso está en ejecución:
  if ((aux->mp_flags)&&IN_USE)
Se carga el mem_map del proceso:
  struct mem_map *mm;
  mm=aux->mp_seg;
Y se imprimen los datos del mapa de memoria:
  *NOTA: mm[T]: text, mm[D]: data, mm[S]: stack
  Posición de memoria (Física): mm[T].mem_phys;
  Tamaño de memoria (Virtual) en clicks:
mm[T].mem_len+mm[D].mem_len+mm[S].mem_len;
  (Estos son los datos del grupo, para ver los de los segmentos se imprimirían

los datos de mm[T], mm[D] y mm[S] por separado.)

Se añade esta función (segMem()) al comando MemLib previamente creado (En el
distribuidor del gestor de memoria)

Para "Monotorizar" FORK:
Se crea la tabla de huecos de memoria libres como se ha hecho con anterioridad.
Para imprimir los datos del proceso en ejecución, se usa mp como variable m_proc.
mp es una variable global del gestor de memoria que apunta al proceso en ejecución.

Los datos para la práctica E se han obtenido de:
http://www.infor.uva.es/~benja/mm/mm.html

Viernes 15/04/16

/*===
=======================*
* Resumen
*

*===
=======================*/
Carpetas y archivos de interés:
_____
* /elleprob.b1 [El editor de texto propio de Minix]
* /.exrc [Opciones del editor de texto Vim]
* /bin [Archivos compilados del sistema (Comandos)]
* /etc [Configuración de Minix]
* /minix [Versiones de Minix]
* /usr ["Unix System Resource" donde se almacena la mayor parte de archivos de Minix]
* /etc/passwd [Usuarios de Minix]
* /usr/include/minix [Archivos del sistema operativo]
* /usr/src/kernel [Núcleo de Minix]
* /usr/src/mm [Gestor de memoria]
* /usr/src/tools [Herramientas del sistema]

Añadir un usuario:
_____
En /etc/passwd
nombre:contraseña:uid:gid:gecos:dir:shell
nombre: Nombre de usuario
contraseña: Contraseña del usuario (Encriptada)
uid: "User ID" ID de usuario (Número)
gid: "Group ID" ID del grupo de usuario (Número) (Los grupos están en /usr/src/etc/group)
gecos: Información del usuario (Ej: Nombre completo)
dir: Ruta del directorio del usuario
shell: Ruta del login shell del Usuario

El password se puede cambiar con el comando: passwd <ID>

Compilar el núcleo (Kernel):

_____

En usr/src/tools se usa el comando "make hdboot".

*Los cambios realizados no se producen hasta que se reinicie el sistema.

FORK:

_____

*Modo usuario:

Para crear el proceso, fork hace una llamada al sistema con _syscall() y se accede al gestor de memoria para crear la interrupción.

*Modo privilegiado:

Una vez la interrupción es atendida, se crea otra llamada al sistema para pasar el mensaje.

Se atiende el proceso y una vez acabado se eliminan sus interrupciones de la lista y se vuelve al modo privilegiado.

Crear una llamada al sistema:

_____

En /usr/include/minix/callnr.h se añade la llamada que queremos crear.

En mm/main.c (O otro archivo del gestor de memoria) se añade la función de la llamada.

En el caso de la práctica (do_esops) se crea un distribuidor en el gestor de memoria y en el núcleo para ejecutar funciones varias según el mensaje que le demos a esops.

**El resto consta de crear funciones varias, guardarlas en los distribuidores y ejecutarlas con esops.

Gestión de memoria:

_____

La lista de procesos están almacenados en el gestor de memoria como esctructuras "mproc", y los segmentos de datos se guardan en estructuras "mem_map".

Los segmentos son Text, Data y Stack. De cada uno se puede obtener su localización en memoria (Física) y su tamaño (memoria virtual)

Los espacios libres en memoria son almacenados como estructuras "hole" de los cuáles también podemos obtener su localización en memoria y su tamaño.