

Guillermo Anta Alonso

Adrian Calvo Rojo

Grupo 11.

Martes 9/2/2016

PRACTICA A

Hemos comenzado con la primera práctica, descargando e instalando Virtualbox y los archivos de disco Minix y el disquete con el núcleo. Hemos creado la maquina virtual añadiendo el disquete y teniendo en cuenta el orden en el que esta iba a arrancar. Desactivado arranque por CD. Arrancamos la maquina y accedemos por primera vez como root. Conseguimos cambiar entre terminales con alt + Fi.

Tutorial de vi:

- esc : Cambiamos de edición a modo comandos.

- i : Cambiamos a modo edición en la posición del cursor.

-a : Cambiamos a modo edición en la siguiente posición a la del cursor.

-x : Borra en la posición del cursor.

-.w : Guardar.

-.q : Salir.

-xyy : Copia x líneas desde el cursor.

-p : Pegar

-xdd : Borra x líneas desde el cursor.

-/x : Busca x en el archivo.

-.:x : Avanza hasta la línea x.

-% : Cambia entre parejas de corchetes.

-u : Deshacer.

-.! : Forzar comando.

-xG :Ir a la línea x.

`--r` :nombreDeFichero: copia un fichero desde la posición de cursor.

`--set number` :Muestra el número de línea en el archivo que estamos viendo.

Continuando con la práctica, copiamos los archivos de imagen y disquete en otro directorio y cambiamos el UUID con "VBoxManage internalcommands sethuid minix...vdi"

Cambiamos el shell por defecto a ash, editando /etc/passwd y sustituyendo sh por ash en la primera línea, creamos otro usuario añadiendo la línea "login:contraseña:9:3: Datos usuario:Directorio del usuario:"

Reiniciamos con el comando "shutdown -r now" o con "reboot".

Apagamos con el comando "shutdown -h now".

Cambiamos el mensaje de inicio del terminal editando el archivo /usr/src/kernel/tty.c (línea 146). Nos movemos al directorio usr/src/tools y compilamos el núcleo con el comando "make hdbboot". Vemos que la revisión ha cambiado y se ha copiado el nuevo núcleo a /minix.

El mensaje no ha cambiado. EXPLICACION: nos damos cuenta de que en la segunda máquina estábamos arrancando desde disquete por lo que las modificaciones hechas no tenían ningún efecto, cambiamos el orden de arranque y vemos que el mensaje también ha cambiado.

Siguiendo el guión hacemos el programa a.out, necesitamos investigar sobre la función fork(), comprendemos que crea una copia exacta de el programa actual en otra posición de memoria y que devuelve el PID del hijo creado. Gestionaremos los errores con "errno".

El programa recibe un argumento de tipo entero y crea tantos procesos como indica este, cada hijo ejecuta la función hijo y termina. El padre espera a que todos terminen.

Modificamos la constante NR_PROCS (usr/include/minix/config.h) de 40 a 50 para cambiar el número de hijos que pueden crearse que en un principio eran 27. Compilamos y reiniciamos.

Martes 23/2/2016

PRACTICA B

Comenzamos la práctica B. Grupo 11. Muestra el número de procesos actuales, modificamos el programa anterior para que cada hijo ejecute una función en vez de no hacer nada. Solucionado, ahora se pueden ejecutar 36 hijos. Ahora si comenzamos la práctica B.

Vamos a hacer el seguimiento de una llamada a fork():

1 - (Modo Usuario) Cuando llamamos a `fork()` (`/usr/src/lib/posix/_fork.c`) desde un programa se efectúa una llamada a la función `_syscall(MM, FORK, &m)`.

Sabemos que la función `fork()` devuelve el PID del hijo que recibe el padre al ejecutar la llamada, indagamos en el código de otras llamadas al sistema como `exit()` o `wait()` y vemos que se repite la llamada a `_syscall(MM,[algo],&m)`, variando `[algo]` en función del tipo de llamada que se realiza.

2 - (Modo Usuario) La función `_syscall()` (`/usr/src/lib/other/syscall.c`) llama a la función `_sendrec()` que también envía un mensaje a un servidor (en este caso también MM).

El parámetro `m` declarado en `fork()`, se define en la función `_syscall()`, llenando el campo `m_type` (`m_source`: emisor del mensaje y `m_type` tipo de mensaje), a `_sendrec()` se le pasa el servidor que tratará la excepción MM y el mensaje con el tipo ya definido.

3 - (Modo Usuario) En `_sendrec()` (`/usr/src/lib/i386/rts/_sendrec.s`) en ensamblador, encontramos la interrupción de software `int SYSVEC`.

En esta encontramos las instrucciones `push`, `mov`, `pop` las cuales guardan en pila, mueven entre registros y memoria y sacan de la pila. Encontramos también `SYSVEC = 33`. A partir de este punto el procesador pasa a Modo Privilegiado.

4 - (Modo Privilegiado) Se comienza a tratar la interrupción, la función `prot_init()` (`/usr/src/kernel/protect.c`) inicializa la tabla de vectores de interrupción, encontramos una llamada a la función `s_call()`. Y una mención a `SYS386_VECTOR` (`src/kernel/const.h`) en el cual se define con el valor de `SYSVEC` (en este caso 33).

5 - (Modo Privilegiado) La función `s_call()` (`usr/src/kernel/mpx386.s`) llama a la función `sys_call()` (`call _sys_call`), la cual será encargada de enviar el mensaje al servidor (en este caso MM) y el mensaje (en este caso FORK)

6 - (Modo Privilegiado) Una vez enviado el mensaje se ejecuta `_restart()` (`usr/src/kernel/mpx386.s`). Esta función permite continuar la ejecución de otro proceso, que

en este caso es MM ya que ha de tratar el mensaje mandado por `sys_call()`. En este punto el procesador vuelve a Modo Usuario.

7 - (Modo usuario) El main de MM (`/usr/src/mm/main.c`) queda a la espera de recibir un mensaje ejecutando en un bucle infinito `get_work()`, una vez recibido llama a la función que ha de tratarlo a través de el vector de funciones `call_vec[]`.

Miércoles 24/2/2016

Vamos a buscar la función del gestor de memoria que se ejecuta cuando se recibe la llamada al sistema `fork()`:

Lo primero que hacemos es `cd /usr/src/mm` y `ls` para ver el contenido de la carpeta del gestor de memoria. Creemos que lo mejor es comenzar analizando el código de `main.c`, vemos que la función `mm_init()` inicializa las variables que usará el gestor de memoria, en esta función encontramos los `printf` con la información de memoria que encontramos al arrancar el sistema (Memory, MINIX, RAM, Available). Lo siguiente que hace el gestor de memoria es ejecutar en un bucle infinito `get_work()` (como ya habíamos visto en la parte anterior), la cual espera al siguiente mensaje (mediante la función `receive(ANY, &mm_in)`), y extrae la información necesaria de este, `who = mm_in.m_source` que será el que envía el mensaje y `mm_call = mm_in.m_type` que es el tipo de mensaje enviado `call number`, que para `fork()` será 2, definido en `/usr/include/minix/callnr.h`, este es un valor válido:

```
if (mm_call < 0 || mm_call >= NCALLS){  
    error = EBADCALL;  
  
    else  
  
    error = (*call_vec[mm_call})();
```

Lo siguiente que se nos ocurre es buscar por el vector `call_vec[]` en el directorio del gestor de memoria (`usr/src/mm`), hacemos `grep call_vec *` para buscar en todos los archivos del directorio, que nos devuelve los archivos `glo.h`, `main.c` y `table.c` con la línea `_PROTOTYPE (int (*call_vec[NCALLS]),(void))...` buscaremos en este último (`usr/src/mm/table.c`) encontramos el vector `*call_vec[NCALLS]`, el cual tiene en la posición 2, `do_fork`. Repetiremos el proceso anterior y buscaremos `do_fork` en el directorio, `grep do_fork *`, que nos devuelve los archivos `forkexit.c` y `proto.h` con la línea `_PROTOTYPE(int do_fork, (void))`;; buscaremos en este último (`usr/src/mm/proto.h`), solo encontramos el prototipo de la función, buscaremos

en forkexit.c (usr/src/mm/forkexit.c), encontramos do_fork() y una constante que habíamos modificado anteriormente NR_PROCS en la siguiente línea:

```
if (procs_in_use == NR_PROCS) return(EAGAIN);
```

Lo que explica porque cuando cambiamos en la práctica anterior el valor se nos permitía crear más procesos al ejecutar a.out. Añadimos un printf en la función do_fork(), vamos a comprobar si saluda ejecutando a.out, no funciona, se nos ha olvidado compilar el núcleo, hacemos make hdbboot desde (usr/src/tools), compila sin problemas, la imagen se ha copiado en /dev/hd1a:/minix/2.0.0r14, vamos a comprobar si funciona de nuevo. Funciona, no hace falta ejecutar a.out al reiniciar encontramos el mensaje de saludo muchas veces en la pantalla, suponemos casi sin miedo a equivocarnos que esto es debido a que se crean muchos procesos para el Sistema Operativo. Modificamos el printf para añadir un salto de línea al final del mensaje para que todo resulte un poco más legible. Compilamos y reiniciamos. 2.0.0r15.

Cada vez que ejecutamos un comando aparece el mensaje. Con cd no pasa?????.
EXPLICACION: Porque cd es una rutina del shell, no hace falta crear un proceso.

Martes 1/3/2016

PRACTICA C

Práctica C: Vamos a crear una nueva llamada al sistema, ESOPS, la cual modificará e imprimirá los parámetros que reciba del mensaje. Esta llamada pasará del Usuario (nivel 4) -> MM (nivel 3) -> SYSTEM (nivel 2-1) -> MM (nivel 3) -> Usuario (nivel 4).

Lo primero que haremos será añadir la constante alfanumérica que identificará a nuestra llamada en el sistema (usr/include/minix/callnr.h) que será la 77 ESOPS.

En src/usr/mm/main.c encontramos el bucle infinito que espera un mensaje, que cuando recibe pasa a buscarlo en call_vec[] con el número de llamada al sistema (77 en el caso de ESOPS), la función call_vec[] inicializada en /usr/src/mm/table.c es el lugar donde tendremos que añadir la entrada correspondiente a nuestra función.

Para que nuestra función sea operativa debemos añadir el prototipo de la misma en /usr/src/mm/proto.h y escribir el código do_esops en /usr/src/mm/utility.c que lo que hará es: imprimir los valores que llegan al gestor de memoria MM, invocar a la tarea del sistema y guardar los cambios realizados.

Aprendemos que los structs de tipo message usan las constantes m1_i1, m1_i2... definidas en /usr/include/minix/type.h y que en este caso el argumento es un puntero a mensaje por

lo que es necesario usar -> y no . para acceder a estos campos. También que mm_in y mm_out son estructuras globales para el envío y recepción de mensajes desde MM.

En /usr/src/kernel/system.c añadimos la línea en sys_task():

```
case ESOPS:    r = do_esops(&m); break;
```

Más adelante en el mismo archivo escribimos la función do_esops(), que será la que ejecute el kernel y lo que hará será recibir los parámetros del mensaje que le pasa el gestor de memoria y sumarle al primer campo 10, al segundo 100 y al tercero 100 e imprimir los valores antes y después de ser modificados. Compilamos el kernel y reiniciamos (2.0.0r16).

Domingo 6/3/2016

Continuando con la práctica C, creamos el fichero PracticaC.c (root/Practicas/Practica-C) en el cual incluimos los ficheros de cabecera:

<stdio.h>, <lib.h>, <sys/types.h>, <unistd.h> y <minix/syslib.h>

Declaramos una variable del tipo mensaje e inicializamos los tres primeros campos con 1, 2 y 3. Utilizamos la función _taskcall(MM,ESOPS,&msg); para realizar la llamada al sistema, imprimimos los valores de los campos. Compilamos, reiniciamos y ejecutamos. Problema: No sabemos desde donde se están imprimiendo los parámetros del mensaje, añadimos identificadores, compilamos el núcleo y reiniciamos (2.0.0r17). Perfecto, ahora se ve desde que parte se imprimen los valores del mensaje.

Tercera parte: vamos a añadir nuestra propia llamada al sistema, para ello pasamos un argumento a nuestra función ESOSP, dependiendo de el valor del argumento invocaremos a una u otra función. Para ello escribimos nuestra nueva función en /usr/src/kernel/system.c y añadimos en ESOPS un switch que actuará de distribuidor. Por lo tanto no es necesario incrementar la variable NCALLS, ya que la llamada será gestionada a través de ESOPS. Escribimos el código de NEWCALL para que nos devuelva el PID y el nombre del proceso en ejecución.

Martes 8/3/2016

Revisando el apartado anterior nos damos cuenta de que no devolvemos los valores modificados desde el kernel, lo arreglamos en (usr/src/kernel/system.c), compilamos y reiniciamos. No compila porque tenemos varias llamadas a

NEWCALL la cual no está todavía terminada. Aprovechamos para quitar el mensaje que imprimía do_fork() realizado en la prácticaB para ver el terminal mas claro. (2.0.0r18).

Después compilamos y reiniciamos para ver si funciona, no funciona debido a un error de sintaxis, solucionado, imprimimos pid y nombre del proceso en ejecución.(2.0.0r20).

Viernes 11/3/2016

Seguimos intentando que nuestra nueva llamada al sistema devuelva más información del proceso que está en ejecución, como el tiempo que lleva ejecutándose etc.. Añadimos estas nuevas características, compilamos y reiniciamos (2.0.0r21). No sale el tiempo, intentamos solucionarlo, CyR (2.0.0r22), no lo conseguimos. (2.0.0r23), seguimos en ello (2.0.0r24). Lo medio conseguimos.

Martes 15/3/2016

Terminamos la practica C (2.0.0r25), imprimimos, nombre del proceso, PID, user_time y sys_time.

PRACTICA D

Comenzamos con la práctica D, leyendo las funciones pick_proc(), ready(), unready(), sched() (usr/src/kernel/proc.c)

PROBLEMA: Hemos modificado la práctica C (Y la entrada del Bitacora) para que desde do_esops() invoque a la función que se le indique en el argumento que se le pase, siendo 100 do_newcall(). (2.0.0r26).

pick_proc(): Es la encargada de decidir a que proceso de la cola de listos se le va a asignar CPU, nos fijamos que la prioridad asignada es TASK_Q > SERVER_Q > USER_Q, por tanto la función tratará de asignar CPU a los procesos listos de la tarea del sistema, después de las tareas de MM y FS y por último de las tareas de usuario. Encontramos en esta función proc_ptr y bill_ptr, siendo proc_ptr la utilizada en do_newcall() para imprimir los datos del proceso actual en ejecución, se nos ocurre sustituir proc_ptr por bill_ptr, compilamos y reiniciamos (2.0.0r27). Ejecutamos practicaC y le pasamos como argumento 100 para que desde do_esops se invoque a NEWCALL. Vemos que el nombre que imprime es SYS y el PID 0.

Miércoles 16/3/2016

Investigamos acerca de NIL_PROC, para ello hacemos en el mismo directorio grep NIL_PROC * | more y vemos que NIL_PROC está definido en proc.h y que es un casting del tipo ((struct proc *) 0), es decir de 0 a struct proc *.

Hacemos lo mismo con `rdy_head[]` y vemos que es una lista de punteros a la primera posición de las colas.

`ready()`: Mete un proceso al final de la cola de listos. Mantiene 3 colas, `TASK_Q` de mayor prioridad para los procesos del sistema, `SERVER_Q` de prioridad media para los procesos de los gestores de memoria, `USER_Q` de baja prioridad para los procesos de usuario. Para ello identifica el tipo de proceso y lo mete en la cola correspondiente. Esto lo hace mediante `istaskp()`, `isuserp()`. Si la cola esta vacía lo meterá en la primera posición (cabeza), sino simplemente lo insertará en esta. Investigamos acerca de `SHADOW_Q` que es una cola adicional que se usará si la constante alfanumérica `SHADOWING` tiene valor 1.

`unready()`: Saca un proceso de la cola de listos, da a `rdy_head[]` el valor del siguiente proceso de la cola a través de `p_nextready`. Después invoca a `pick_proc()` que decidirá cual es el siguiente proceso que se ejecutará. La forma de expulsar el proceso de la cola, es buscarlo en esta y sustituirlo por el siguiente listo. En esta función también encontramos `SHADOW_Q`.

Jueves 24/3/2016

Continuamos con la Práctica D.

`sched()`: Se invoca a este método cuando un proceso ha estado demasiado tiempo en ejecución (ha superado el quantum). Inserta el proceso actual al final de la cola y pone el siguiente listo en la cabeza. Este método solo funciona con `USER_Q`, no se desalojan procesos de sistema ni de gestor de memoria. Después de esto invoca a `pick_proc()` para que decida cual será el siguiente proceso para ejecutar.

Vamos a leer y analizar las funciones: `init_clock()`, `clock_handler()`, `clock_task()`, `do_clocktick()` en `usr/src/kernel/clock.c` e `interrupt()` en `usr/src/kernel/proc.c`.

La primera función que encontramos es `init_clock()` encargada de inicializar el reloj y hacer que funcionen de manera continua, carga el tiempo en el canal 0 y elige quien será el encargado de manejar las interrupciones del reloj (`clock_handler()`), del tipo del chip depende la inicialización de este método.

Viernes 1/4/2016

Después analizamos `clock_handler()`, encargada de manejar el reloj, incrementando el tiempo de los procesos de sistema y de generar una interrupción cuando se agota el quantum. Cambia el valor de ticks de la siguiente forma:


```
ticks = lost_ticks + 1; y lost_ticks = 0; pending_ticks += ticks;
```

También modifica la variable now de tipo clock_t:

```
now = realtime + pending_ticks;
```

La variable realtime se actualiza desde la función clock_task() ya que está protegida, es la variable utilizada para determinar si ha de producirse una interrupción if (next_alarm <= now) o porque se agota el quantum del proceso, determinado por la variable sched_ticks, si llega a 0 se vuelve a reiniciar el quantum.

Pasamos a analizar clock_task(), encargada de la gestión de las tareas del reloj. Inicializa el reloj mediante una llamada a init_clock(), después entra en un bucle infinito en el que actualiza de forma protegida mediante lock() y unlock() es decir:

```
lock();  
  
realtime += pending_ticks;  
  
pending_ticks = 0;  
  
unlock();
```

También actúa como distribuidor de funciones relacionadas con las interrupciones de reloj.

Veremos ahora do_clocktick() que es invocada desde clock_task() cuando el opcode es HARD_INT, por error escribimos un comando del tipo comando() y vemos que se abre una especie de editor de texto, nos damos cuenta de que podemos crear nuestro propio comando personalizado, lo primero que se nos ocurre es un comando para que el núcleo compile automáticamente. Lo hemos llamado compilanucleo. Siguiendo con la práctica deducimos que la utilidad de do_clocktick() es comprobar si se han producido interrupciones y tratarlas. Encontramos una llamada a la función lock_sched() definida en proc.c, la cual realiza una llamada a sched() pero cambiando el valor de la variable switching a TRUE antes de hacer la llamada y volviendo a cambiarlo a FALSE después de hacerla.

Analizamos ahora interrupt() y vemos que es una función invocada cuando el quantum de un proceso se agota. Si k_reentrar != 0 o switching tiene valor TRUE, el proceso se añade a una cola de retenidos. Si el proceso espera una interrupción se bloquea, se manda el mensaje HARD_INT desde HARDWARE y se añade rp a TASK_Q.

Consultamos ahora la inicialización de la variable sched_ticks que es la que contiene los ticks restantes del proceso en ejecución. Encontramos en su definición (MILLISEC*HZ/1000), buscamos el valor de HZ, la encontramos en /usr/include/minix/const.h con un valor de 60. Por lo tanto la variable SCHED_RATE = 6;

Viernes 8/4/2016

Vamos a crear el programa para crear 3 hilos ligeros (1seg) y 1 pesado (10-20seg). Hemos conseguido terminar el programa, los ligeros tardan alrededor de 1 segundo y el pesado alrededor de 14 segundos.

Nos disponemos a crear la función `cambiarQ()`. Vamos a llamar a esta función directamente desde `system.c`.

Para ello lo que hacemos es crear en el fichero `clock.c` la función `cambiarQ(nq)`, esta función recibirá como parámetro un `int` que asignará directamente a la variable `sched_ticks`.

Ahora dentro de `system.c` llamamos a la función anterior pasándole el mensaje que se generará en `m1_i2`. Solo necesitamos llamar a esta función desde el repartidor para conseguir que funcione.

Lunes 11/4/2016

Después de pensar que no conseguíamos llegar hasta la última función en `clock.c:cambiarQ()`, tras hacer algunas comprobaciones en las funciones y poner varios `printf()` por los tramos que pasa la llamada... conseguimos que funcione y que llegue a cambiar la variable `sched_ticks`. Para hacer llegar hasta la función de cambiar el cuanto hemos creado una variable alfanumérica a la cual hemos asignado el valor 1. De esta forma escribimos una llamada a la función `system.c:do_esops()` enviando el valor 1 y el valor del nuevo cuanto para cambiarlo.

Vamos a comprobar ahora el efecto que tiene cambiar este cuanto; ejecutamos primero el programa que creaba 3 hilos ligeros y uno pesado.

Tenemos unos `printf` de tal forma que vemos cuando empieza y cuando acaba el hilo. Si lo ejecutamos sin cambiar nada, lo que ocurre es lo siguiente:

Se crea proceso pesado

Se crea proceso ligero

Acaba proceso ligero (1 segundo)

Se crea proceso ligero

Acaba proceso ligero (1 segundo)

Se crea proceso ligero

Acaba proceso ligero (1 segundo)

Acaba proceso pesado (21 segundos)

Ahora ponemos un cuanto de 200000 y volvemos a ejecutar el mismo programa, pasa esto:

Se crea proceso pesado

Acaba proceso pesado (21 segundos)

Se crea proceso ligero

Acaba proceso ligero (1 segundo)

Se crea proceso ligero

Acaba proceso ligero (1 segundo)

Se crea proceso ligero

Acaba proceso ligero (1 segundo)

Comprobado, todo funciona correctamente. EXPLICACION: Al cambiar el cuanto lo que conseguimos es cambiar de RR a FIFO, por tanto hasta que no finalice un proceso, no empieza el siguiente. Aquí se acaba la práctica D.

Martes 12/4/2016

PRACTICA E

Empezamos la practica E. Para ellos nos dirigimos al gestor de memoria de minix en `/usr/src/mm/`. Como tenemos que imprimir la lista `hole_head` y la tabla de segmentos de las tareas del sistema cuando se inicie la memoria, nos dirigimos al archivo `/usr/src/mm/alloc.c` donde estan definidos el numero de holes asi como su estructura (almacena dónde empieza, qué tamaño tiene y un puntero hacia el siguiente).

Tenemos que llamar a la funcion que creemos desde `/usr/src/mm/main.c:mm_init()`.

Una vez creada la funcion en `/usr/src/mm/alloc.c`, que simplemente recorre la lista `hole_head` hasta que encuentra un agujero nulo (`NIL_HOLE`) e imprime los parametros `h_base` y `h_len`, la añadimos al distribuidor `do_esops` para despues ser llamada desde un comando. A dicha funcion tambien la llamamos desde `/usr/src/mm/main.c:mm_init()` que se llama cuando minix arranca el mm. Compilamos el nucleo y no nos da ningun error. Creamos

un comando llamado MemLibre y funciona perfectamente mostrando cinco agujeros. Nos fijamos que tambien funciona cuando arranca minix, salen dos agujeros.

Ahora vamos a hacer las funciones que impriman las tablas de segmentos de todos los procesos de usuario. Primero creamos una nueva constante alfanumerica a la que llamamos IMPRIME_SEG en /usr/include/minix/callnr.h. Los procesos se definen en el archivo /usr/src/mm/mproc.h, donde se definen todas sus propiedades (cuándo acaba, ID, PID del grupo, indice del padre...) y se almacenan en una tabla.

Vamos a crear ahora una función en utility.c llamada imprime_tabla_seg() que recorrera todos los procesos de usuario y otra para que recorra cada tabla de segmentos dentro de cada proceso de usuario. El tipo que describe el mapa de memoria es struct mem_map mp_seg[NR_SEGS];" Vamos a imprimir por pantalla cada atributo de la estructura mem_map.

Una vez hechas las dos funciones, solo nos queda añadir la llamada al repartidor y comprobar si funciona. Despues de varios errores de compilacion y dos malditas horas comiendonos la cabeza conseguimos que compile correctamente. Hacemos un reboot y comprobamos que funciona e imprime todos los segmentos segun lo esperado. Para finalizar solo queda que imprima tambien el estado de la lista hole_head al realizar esta última llamada.

Jueves 14/4/2016

Hemos retocado el distribuidor de MM para que cuando hacemos la llamada a ESOPS pasando como primer argumento 3, y como segundo 0 imprima todas las tablas de segmentos de los procesos y que si el segundo argumento es un 1 imprima solo las del proceso que lo invoca a MM. De esta manera continuamos con la ultima parte de la práctica E.

Creamos el programa para monitorizar el funcionamiento de FORK, lo primero que hacemos es invocar la función que muestra hole_head mediante una llamada a ESOPS pasándole como primer argumento MEMLIBRE, El siguiente paso es invocar a FORK, e imprimir las tablas de segmentos para el padre y para el hijo, haciendo una llamada a ESOPS pasándole como primer argumento IMPRIME_SEG y como segundo argumento 1. Compilamos y comprobamos que funciona.

Viernes 15/4/2016

Aunque parezca difícil y después de mucho darle vuelta a algunos problemas hemos conseguido completar todas las prácticas, finalizando con un MINIX "personalizado" que dispone de los dos distribuidores tanto el del gestor de memoria, como el del núcleo, con comandos como MemLibre y cambiaQ y a mayores algunos añadidos por nuestra propia comodidad como compilanucleo, que nos permite compilar desde cualquier directorio. Finalizamos así con la etapa de prácticas A-B-C-D-E.

Martes 26/4/2016

PRACTICA ABIERTA

Hemos decidido que la práctica que vamos a realizar sea la gestión de memoria en minix. Dentro de la función `/usr/src/alloc.c:alloc_mem()` se encuentra la forma en la que gestiona minix la memoria. Este método por defecto, usa el algoritmo "first fit". Recibe como parámetro el tamaño que ocupa un proceso (en clicks), y procede a buscar el hueco en la tabla de agujeros libres según uno de los siguientes métodos:

First-Fit: Al recorrer la tabla asigna el primer agujero que encuentra con espacio disponible para ubicar el proceso.

Best-Fit: Recorre toda la lista de huecos y de todos los agujeros en los que cabe el proceso elige el de menor tamaño.

Worst-Fit: Recorre toda la lista de huecos y de todos los agujeros en los que cabe el proceso elige el de mayor tamaño.

La idea es crear un distribuidor que cambie entre los algoritmos de gestión de memoria de los procesos: "Best-fit", "Worst-Fit", "First-Fit".

Jueves 28/4/2016

En el fichero `/usr/src/mm/glo.h` hemos creado una variable global:

```
EXTERN int gestor_mem;
```

Esta variable será la que modifiquemos para cambiar el plan de gestión de memoria para los procesos.

Los procesos al ser creados (mediante FORK) son gestionados por el archivo `/usr/src/mm/forkexit.c`. Este archivo tiene diferentes funciones:

Do_fork(): maneja la llamada al sistema FORK. Recibe el puntero al proceso padre con la variable `mp`. Comprueba que haya sitio en la tabla de procesos. Se calcula el tamaño del hijo

en función del padre. Se reserva este espacio en hole head mediante `alloc_mem()`. Después se copia el espacio de memoria del proceso padre en el hijo. Finalmente se informa al servidor de ficheros y al núcleo sobre la creación del nuevo proceso.

`Mm_exit()` y `do_waitpid()`: Liberación de recursos y espera a la finalización de otros procesos. No las vamos a usar.

Martes 3/5/2016

Creamos una nueva variable alfanumérica llamada `CAMBIA_MEM`. Modificamos el distribuidor de memoria que teníamos que `/usr/src/mm/utility.c` y añadimos un nuevo caso al switch para que llame al método que va a cambiar la variable global `gestor_mem`. El método encargado de esta tarea será `cambia_mem()` en `/usr/src/mm/utility.c`

Lo siguiente que haremos será comenzar a modificar `alloc_mem` para que elija el algoritmo de planificación de memoria en función de la variable global que hemos definido anteriormente, para ello implementaremos una función que será la invocada desde `forkexit.c` y `exec.c`, la cual comprobará el valor de la variable `gestor_mem` y en función de este cambiará el algoritmo de asignación de memoria.

Comenzamos creando el algoritmo “Best Fit” en `/usr/src/mm/alloc.c` en una función llamada `alloc_mem_BF`, lo que hará esta función será recorrer la lista de holes y elegir el agujero mas pequeño en el que cabe el proceso, para ello compararemos cada agujero con el tamaño del proceso y con el tamaño del agujero anterior, eligiendo el menor entre estos dos y repitiendo el proceso.

Martes 10/5/2016

Una vez que tenemos hecho el código para Best Fit guardamos el estado del disco duro por si nos lo cargamos, compilamos (compila a la primera), y cambiamos las llamadas a la funcion `/usr/src/mm/alloc.c:alloc_mem` directamente en `/usr/src/mm/forkexit.c` y `/usr/src/mm/exec.c`. Lo haremos así solo por esta vez, mas adelante crearemos un distribuidor. Reiniciamos la maquina y aparentemente funciona todo correctamente.

Ahora que sabemos que Best Fit no da errores (no sabemos si aun funciona) volvemos a cambiar el código anterior que modificamos en `forkexit.c` y `exec.c` para que cargue siempre el código por defecto de minix (First Fit). En `alloc.c` hacemos un distribuidor sencillo con un switch que, basándose en la variable global “`gestor_mem`”, llama a las distintas funciones seleccionando el tipo de gestión de memoria que queremos: First Fit, Best Fit o Worst Fit. El camino de gestion de memoria es el siguiente:

1. Se crea un proceso mediante `fork` que llama a la funcion `alloc.c:alloc_mem()` enviando el tamaño en clicks del proceso.
2. La función `alloc_mem` ahora es un distribuidor con un switch que comprueba la variable “`gestor_mem`” (por defecto es 0) y:

3. Si "gestor_mem" = 0 llama a First Fit
4. Si "gestor_mem" = 1 llama a Best Fit
5. Si "gestor_mem" = 2 llama a Worst Fit
6. Si "gestor_mem" toma cualquier otro valor, llama a First Fit (default)
7. Estos métodos reciben el tamaño en clicks y devuelven un puntero indicando dónde empieza el hueco destinado al proceso.

Jueves 12/5/2016

Basándonos en el código de Best Fit, lo copiamos y creamos el método para Worst Fit cambiando la clausula "<" que elige el hueco menor, porque en el caso de Worst Fit debe seleccionar el mayor.

Ponemos unos cuantos printf para seguir el camino de gestion de memoria y asi ver en alloc.c el tamaño en clicks del proceso que se pide y en qué hueco se mete. Creamos ahora un comando mediante el que podemos cambiar o ver el algoritmo de gestión de memoria de esta forma:

- "gestor ver" nos muestra el algoritmo de gestión de memoria actual.
- "gestor cambiar [0,1,2]" cambiar el algoritmo.

Domingo 15/5/2016

Lo que vamos a hacer ahora es modificar el programa de una práctica anterior que calculaba números primos para que al principio muestre la tabla de agujeros vacíos y durante la ejecución vaya creando los procesos uno a uno. Asi podemos ver qué proceso se crea, cuánto ocupa y en qué agujero se mete.

Probando el mismo programa con los distintos métodos de gestión de memoria parece que funciona todo como debe ser. También corregimos un error a la hora de mostrar los huecos libres, que no mostraba los clicks del tamaño del hueco bien. Ahora usamos la función de minix `click_to_round_k()` y ya lo muestra bien.

Martes 17/5/2016

Comenzamos a implementar otro algoritmo, en este caso será Last Fit que alojará el proceso en el último hole de la lista en el que entre. Conseguimos implementar este nuevo algoritmo, a mayores hemos sustituido en el distribuidor (alloc_mem) los enteros del switch por variables alfanuméricas. Comprobamos que funciona.

Viernes 20/5/2016

Empezamos a implementar el siguiente algoritmo, en este caso será NEXTFIT que consistirá en guardar el último hole en el que hemos alojado el último proceso y cuando tengamos que asignar memoria comenzaremos a recorrer hole_head desde el hole que hemos guardado.

Lo primero que hacemos es definir una variable global en glo.h de tipo struct hole, que será donde guardemos el hole que hemos utilizado.

Martes 24/5/2016

Continuando con el algoritmo Nextfit, intentamos solventar el problema que teníamos añadiendo un contador y limitando las vueltas del bucle con el número máximo de entradas de la table NR_HOLES. Probamos el algoritmo con la carga de trabajo del cálculo de primos que no nos permite ver con claridad si de verdad funciona, sin embargo vemos que al cambiar de un algoritmo a otro se crean procesos de distintos tamaños y los aloja siguiendo el patrón next fit establecido correctamente.

Sábado 28/5/2016

Empezamos a crear una carga de trabajo con procesos de distintos pesos para comprobar cómo los distintos algoritmos asignan memoria a procesos de distintos tamaños, para ello crearemos un proceso ligero, uno mediano y uno pesado y los ejecutaremos desde un comando intercalando MemLibre antes y después de cada proceso para ver cómo se alojan en la memoria.

Martes 31/5/2016

La carga que hicimos el día anterior no da muchos resultados, ya que los procesos ocupan como mínimo 77KB y la tabla de agujeros es la siguiente:

***** Tabla de agujeros *****

Base: 0x1AE, Tamaño: 21 KB

Base: 0x21B, Tamaño: 17 KB

Base: 0x2A1, Tamaño: 62 KB

Base: 0x756, Tamaño: 117 KB

Base: 0x63BC, Tamaño: 10001 KB

Por lo que no conseguimos ver bien del todo el funcionamiento de nuestros algoritmos. Así que decidimos efectuar la carga con una llamada al sistema, que nos crea procesos de distintos tamaños que caben en todos los agujeros. Los procesos de esta llamada tienen los siguientes tamaños: 77 KB, 78 KB, 51 KB, 19 KB, 51 KB y 129 KB.

Ahora vamos a probar esta carga de trabajo con los distintos algoritmos.

- **First Fit**

Tamaño del proceso: 77 KB. Se guarda en el agujero con tamaño: 117 KB (proceso padre)

Tamaño del proceso: 78 KB. Se guarda en el agujero con tamaño: 117 KB (aquí se ha liberado la memoria, por eso lo vuelve a meter en el mismo.)

Tamaño del proceso: 51 KB. Se guarda en el agujero con tamaño: 62 KB

Tamaño del proceso: 19 KB. Se guarda en el agujero con tamaño: 21 KB

Tamaño del proceso: 51 KB. Se guarda en el agujero con tamaño: 62 KB

Tamaño del proceso: 129 KB. Se guarda en el agujero con tamaño: 10001 KB

- **Best Fit**

Tamaño del proceso: 77 KB. Se guarda en el agujero con tamaño: 117 KB

Tamaño del proceso: 78 KB. Se guarda en el agujero con tamaño: 117 KB

Tamaño del proceso: 51 KB. Se guarda en el agujero con tamaño: 62 KB

Tamaño del proceso: 19 KB. Se guarda en el agujero con tamaño: 21 KB

Tamaño del proceso: 51 KB. Se guarda en el agujero con tamaño: 62 KB

Tamaño del proceso: 129 KB. Se guarda en el agujero con tamaño: 10001 KB

Casualmente coincide con First Fit, pero esto es debido al orden en el que están los huecos en la tabla.

- **Worst Fit**

Tamaño del proceso: 77 KB. Se guarda en el agujero con tamaño: 10001 KB

Tamaño del proceso: 78 KB. Se guarda en el agujero con tamaño: 10001 KB

Tamaño del proceso: 51 KB. Se guarda en el agujero con tamaño: 9923 KB

Tamaño del proceso: 19 KB. Se guarda en el agujero con tamaño: 9923 KB

Tamaño del proceso: 51 KB. Se guarda en el agujero con tamaño: 9923 KB

Tamaño del proceso: 129 KB. Se guarda en el agujero con tamaño: 9923 KB

- Last Fit

Tamaño del proceso: 77 KB. Se guarda en el agujero con tamaño: 10001 KB
Tamaño del proceso: 78 KB. Se guarda en el agujero con tamaño: 10001 KB
Tamaño del proceso: 51 KB. Se guarda en el agujero con tamaño: 9923 KB
Tamaño del proceso: 19 KB. Se guarda en el agujero con tamaño: 9923 KB
Tamaño del proceso: 51 KB. Se guarda en el agujero con tamaño: 9923 KB
Tamaño del proceso: 129 KB. Se guarda en el agujero con tamaño: 9923 KB

También como en el otro caso, Last Fit coincide con Worst Fit porque el último agujero es el más grande. Last Fit escoge el último agujero con espacio disponible y Worst Fit el más grande.

Jueves 2/6/2016

- Next Fit

Tamaño del proceso: 77 KB. Se guarda en el agujero con tamaño: 117 KB
Tamaño del proceso: 78 KB. Se guarda en el agujero con tamaño: 10001 KB
Tamaño del proceso: 51 KB. Se guarda en el agujero con tamaño: 62 KB
Tamaño del proceso: 19 KB. Se guarda en el agujero con tamaño: 117 KB
Tamaño del proceso: 51 KB. Se guarda en el agujero con tamaño: 9923 KB
Tamaño del proceso: 129 KB. Se guarda en el agujero con tamaño: 9923 KB

Vemos perfectamente como el primer proceso lo asigna en el primer agujero con espacio suficiente y los demás procesos les va metiendo en el agujero siguiente con espacio disponible para el proceso. En el caso del último proceso, el algoritmo ha asignado al proceso anterior al agujero con tamaño 9923 KB; el algoritmo vuelve a empezar a recorrer la tabla (porque ya ha llegado al último agujero) y llega hasta el mismo agujero ya que es el único con espacio para dicho proceso.

Viernes 3/6/2016

Conclusiones:

En un principio pensábamos que nos iba a llevar más trabajo entender la gestión de memoria de minix, pero después de comprender el funcionamiento de todos los archivos mencionados durante la práctica, en especial alloc.c resulta que no es tan complicado. Al revés, es muy interesante a pesar de que sea un sistema operativo a pequeña escala.

En la práctica se nos proponía crear una serie de algoritmos para gestionar la memoria a partir del que venía por defecto, First Fit. Finalmente conseguimos realizar todo lo que nos propusimos; tanto Best Fit, Worst Fit, Last Fit y Next Fit han funcionado debidamente.

Alguno nos resultó más difícil que otro como Best Fit, que al ser el primero fue el que más horas de trabajo nos llevó.

Hemos creado una serie de comandos para ayudarnos durante la práctica:

- MemLibre: recorre la tabla hole_head y la imprime por pantalla, lo que nos muestra los huecos libres en la memoria.
- gestor cambiar 'x': cambia la variable global gestor_mem por el valor 'x'. De esta forma cambiamos de un algoritmo a otro. Tenemos métodos creados para los valores del 0 al 4 (incluidos) por lo que si introducimos el valor 10, se utilizará el algoritmo por defecto First Fit.
- monitorON/monitorOFF: con estos dos comandos cambiamos al modo monitor, que imprime por pantalla el tamaño que ocupa un proceso y el hueco al que se asigna.
- gestor ver: con este comando imprimimos por consola la variable gestor_mem y es el que aprovechamos para la carga de trabajo.
- Compilanucleo: gracias a este comando podemos compilar rápidamente. Ha sido el que más hemos utilizado por la cantidad de veces que hemos compilado.