Alejandro Pulido Sánchez Grupo: T2 Lab:X4

CUADERNO DE BITÁCORA

En este cuaderno de bitácora de las prácticas de ESO, se relatará el trabajo diario para realizar las prácticas de minix vistas en clase desde el inicio hasta el final, analizando los problemas encontrados y proponiendo las soluciones oportunas.

Tabla de contenido

19/04/2021

Tras instalar y configurar la máquina virtual con los pasos pertinentes explicados anteriormente en el laboratorio y escritos en la práctica 3, dediqué el día de hoy a organizar los directorios como indica esta. Es decir, cree el directorio Practica 3 y dentro de este, dos programas llamados "CreaProcesos.c" y "leeAout.c". El resto del tiempo empleado, lo he invertido en pensar la estructura y la manera de completar el programa de manera exitosa. He pensado en hacer un bucle con el número de iteraciones igual al número introducido a la hora de ejecutar el programa. Por cada iteración crear un proceso hijo del padre.

21/04/2021

Tras el esbozo pensado el último día me puse a escribir el código, empecé incluyendo los #include que vi necesarios, como "stdio.h" o "stdlib.h" entre otros. Después inicialicé las variables imprescindibles como pid de tipo pid_t o un int que almacené el número de procesos. Inicié un bucle for con el número de iteraciones igual al número introducido. Dentro del bucle he introducido un condicional, en el que si el pid es -1 imprime por pantalla que hay un error. Si no entra en el condicional entrará en el else if. Si se cumple que el pid es 1 entonces estaremos en un proceso hijo, por lo que se introducirá en una función llamada hijo(), que aún no he implementado.

23/04/2021

Hoy creé la función hijo, la cual imprime un mensaje diciendo hola soy un hijo. Tras ello me di cuenta de que no había tratado los errores, por ello he introducido otro condicional fuera del bucle for. Si el número de argumentos introducidos es distinto a dos se salía del programa. Tras compilar me encontré con muchos errores de compilación. Entre ellos se me había olvidado importar algunas librerías como errno. También había un error de tipos bastante importante por el cuál no compilaba, esto se debe a que había tomado el número introducido y lo había guardado en una variable de tipo int sin hacer uso de la función atoi para convertirlo de String a int. Tras compilarlo de nuevo seguía habiendo algunos errores.

24/04/2021

Tras volver a revisar el código no sabía muy bien porque fallaba. Al final descubrí que era debido a los argumentos de la función waitpid. Cambie los argumentos de la función para función por pid,0,WNOHANG respectivamente. Ahora me daban algunos warning pero ningún error. Probé a ejecutarlo, pero no cumplía la funcionalidad pedida. Al entrar en el bucle se hacía un hijo por cada proceso, por lo que en vez de x procesos pedidos se creaban $n^x + n^{x-1} + \ldots + 1$. Tras probar algunas cosas no he podido solucionar el problema.

27/04/2021

Intenté ver cuál era el problema y solucionarlo, pero no lo conseguí, entonces para poder seguir avanzando y no quedarme en el primer ejercicio decidí pasar al segundo. En este empecé igual que en el primero, es decir, empecé elaborando un pequeño mapa mental de como completaría el programa pedido. Tras pensar lo que iba a hacer lo primero que escribí fueron las importaciones de las distintas librerías que creía que me iban a ser necesarias, aunque según iba escribiendo el programa tuve que añadir alguna que se me había olvidado. Tras las importaciones empecé a escribir el main en el cual había 2 condicionales. Uno trataba el error de número de argumentos pasados como parámetro distinto de los necesarios y el restante devolvía otro error cuando no se podía abrir el fichero pasado.

28/04/2021

En el main declaré una variable de tipo FILE para poder abrir el fichero que se pasa por parámetro y un struct exec. Para utilizar este tipo de struct tuve que buscar información por internet, ya que no sabía muy bien cómo funcionaba. Encontré esta página que me sirvió para aclarar los conceptos. Después de los dos condicionales utilicé fread para leer del fichero previamente abierto, los argumentos que utilicé fueron un puntero a p, la longitud del struct, la longitud de unidad 1 y el fichero fp. Antes del main definí la función para imprimir todos los datos del fichero, así que después de los if, pasé por la función al struct exec para que imprima por pantalla todos los datos. Para acabar cerré el fichero y retorno 0. Tras compilarlo y ejecutarlo el programa funcionaba bien. Al intentar compilar me daban muchos errores, pero dejé la corrección para el día siguiente.

01/05/2021

Como me pasó en el anterior programa me había olvidado importar algunas librerías, como por ejemplo la del struct exe, es decir <a.out.h>. Ahora el programa parecía compilar y funcionar correctamente. Al final el programa, obviando el tratamiento de errores, las importaciones y demás, quedó una cosa como esto…

```
fp = fopen(argv[1],”r”);
fread(&p,sizeof(struct exec),1,fp);
cabecera(p);
fclose(fp);
return0;
```

La función cabecera algo parecido a…

```
Void cabecera(struct exec a){
        Printf(“CPU ID: %u\n”, a.a_cpu);
                …
}
```

Cómo acabé antes de lo previsto dediqué el tiempo sobrante a leer parte de la siguiente práctica.

04/05/2021

Seguí leyendo la práctica, pero al final la tuve que volver a releer la práctica para localizar todas las líneas de código y completar la tarea 1. Los pasos que se ejecutan cuando se realiza una llamada son los siguientes.

Cuando realizamos un fork(), este está enlazado con una biblioteca(_fork.c) la cual contiene la función _syscall(), la que toma tres argumentos. El primero es el identificador de gestión de memoria, el segundo el tipo de operación que queremos que realice el gestor y el tercero es la dirección de memoria en la cual se encontrarán los argumentos devueltos por el gestor.

Si intentamos editar el fichero syscall.c este invocará a la función sendrec(), la cual envía un mensaje a un proceso servidor(MM). Sendrec se encarga de poner en ciertos registros los argumentos que trae de las funciones anteriores y realizar la llamada al sistema.

Ahora la CPU cambia a modo superusuario y se llama a prot_init(), función que inicializa la tabla de vectores de interrupción, en los que se encuentra SYS386_VECTOR. En el fichero const.h se define la constante SYS386_VECTOR con el valor de SYSVEC que en este caso es 33. Al llegar esta interrupción se llama a s_call(), la que a su vez llama a sys_call(),que está dentro de proc.c, cuyos argumentos ya están insertados en la pila.Dentro de proc.c también se encuentran las funciones mi ni_rec y mini_send, es decr, las llamadas al sistema, mediante estas sys_call envía FORK a MM(el servidor). Después se recupera y se pasa a la ejecución de otro proceso.

05/05/2021

Retomando el primer programa de la primera práctica, decidí preguntar al profesor sobre mi problema de funcionalidad. Tras preguntarle, me dijo que el problema podría estar en el waitpid y por ello me creaba procesos infinitos. Tras estar intentándolo arreglar al final la solución más fácil fue cambiar el waitpid por un wait. Tras hacerlo el programa ya funcionaba correctamente. El “algoritmo” del primer ejercicio quedó algo parecido a esto…

```
numProcesos = atoi(argv[1]);
for(i=0; i<numProcesos;i++){
        pid = fork();
        if(pid == -1){
                printf(“nº de procesos %d\n”, i);
                perror(“Error: “);
                exit(1);
        else if(pid==0){
                hijo(0);
                exit(0);
}}
for(i=0; i<numProcesos;i++){
        wait(0);
}
```

Tras probarlo, el número máximo de procesos que podía ejecutar a la vez eran 27. En el resto de la clase seguí explicando las llamadas en minix pero lo dejé todo en el mismo día para no dividir la explicación

07/05/2021

Ahora que ya queda explicado más o menos cómo funcionan las llamadas en Minix toca pasar a la segunda tarea. Esta se trata de modificar la función del gestor de memoria que se ejecuta cuando se llama a fork() para que al utilizarla salude. Lo primero que hay que hacer es localizar la función que gestiona la memoria. Para ello hay que ver que archivos son los que se relacionan con el servidor. En concreto archivos de tipo .c, así que nos vamos al directorio mm que se encuentra en los directorios usr/src. Si hacemos un ls filtrando por el nombre, es decir ls *.c podremos ver los archivos que estamos buscando. De todos los archivos que aparecen el que más me llamó la atención fue alloc, ya que en teoría hemos visto en teoria que malloc (Memory Allocation) sirve para gestionar la memoria. Si entramos a ver su contenido y leemos su descripción podemos confirmar que esta sí que es la función que se encarga de gestionar la memoria de las llamadas a fork, y además también de exec.

Para conseguir que imprima un mensaje hay que localizar la función exacta que se ejecuta al realizar un fork(). Tras leer las descripciones de todas, a que se encarga de gestionar la memoria es alloc_mem. Al código ya existente de esta función añadí un printf con un mensaje de saludo, así cuando se ejecute se imprimirá por pantalla dicho mensaje. Mala mi suerte que al probarlo no conseguí que me imprimiese nada. Tras hablarlo con un compañero me dijo que es que estaba modificando un archivo que no era, debía modificar el archivo forexit.c que se encuentra la ruta usr/src/mm. Sabiendo esto borré la modificación que había hecho y me dispuse a modificar forexit. Una vez abierto el fichero para editarlo, fui a la función del fork y añadí una línea al principio del código de la función, la línea era un printf como el comentado anteriormente. Tras hacerlo, inserté los cambios en el kernel, y reinicié la máquina. Ahora al hacer un ls ya me imprimía el mensaje, es decir, ya funcionaba correctamente.

12/05/2021

Tras la explicación que nos dio el profesor sobre la fase 2 y 3 de la practica 5 yo comencé la fase 1 de esta. En esta fase solo había que seguir los pasos indicados en el fichero explicativo de la práctica. Tras seguir todos los pasos he intentar compilar el nuevo kernel me daba un error. Por lo visto en system.c no había introducido un nuevo valor para comprobar que es recibido en el nivel 4. Tras areglarlo intenté compilarlo de nuevo, pero ahora daba otro error.

13/05/2021

Intenté averiguar cuál era el fallo que provocaba el error de compilación, pero no fui capaz, por lo que decidí revisar todos los pasos que había que realizar en la fase 1 para corroborar que los hice correctamente. Tras repasarlos no vi ningún fallo ni ninguna cosa rara. Para no quedarme atascado en esta fase pasé directamente a la segunda, aunque debido al fallo de la primera no podré corroborar que la he hecho bien. En esta fase había que crear un pequeño programa con las indicaciones del enunciado. Primero importé las librerías necesarias además de las indicadas y dentro del main incorporé el código.

```
message msj;
msj.m1_i1=10;
msj.m1_i2=20;
msj.m1_i3=30;
_taskcall(MM, ESOOPS, &msj);
Printf("M4: campo1=%d, campo2=%d, campo3=%d\n", msj.m1_i1, msj.m1_i2, msj.m1_i3);
```

Con esto la fase 2 creo que estaría bien pero no puedo comprobarlo debido a los errores en la fase 1.

14/05/2021

Hoy empecé la fase 3 de la última práctica, que tras las explicaciones que dio ayer el profesor parecía no ser demasiado complicada como parecía en un inicio. Empecé a realizar los cambios en el archivo System.c. Lo primero que hice fue definir do_caso1 y do_caso2. Ambos códigos eran similares con excepción de los valores de m_ptr->m1_i2 y m_ptr->m1_i3. El código era algo similar a …

```
PRIVATE int do_caso1(m_ptr)
Reguster message *m_ptr;
{
m_ptr->m1_i2= 2000;
m_ptr->m1_i3= 100;
printf("KERNEL:do_caso1: a1=%d, a2=%d, a3=%d\n", m_ptr->m1_i1, m_ptr->m1_i2, m_ptr-
>m1_i3);
}
```

Este es el código de do_caso1, el de do_caso2 es prácticamente el mismo, lo único que cambia son los valores de m_ptr->m1_i2 y m_ptr->m1_i3, que pasan a valer 3000 y 300 respectivamente. Después modifiqué el código de do_esops añadiendo el switch. Lucía algo como…

```
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("KERNEL:do_caso1: a1=%d, a2=%d, a3=%d\n", a1,a2,a3);

switch (a1) {
        case CASO1: do_caso1(&m);     break;
        case CASO2: do_caso2(&m);     break;
        case CASO3: m_ptr->m1_i2=a2+15;      break;
        default:          printf("error en llamada\n);
}
}
```

Lo último que hice en este archivo fue añadir los prototipos de do_caso1, do_caso2 y do_esops.

El otro archivo que he cambiado ha sido callnr.h, añadiendo al final esops con el número 77 y caso1 caso2 y caso 3 con 1,2 y 3 respectivamente.

19/05/2021

Hoy volví a revisar la fase 1, tras estar un buen rato me di cuenta de lo que fallaba era un comentario que no estaba bien indicado. Tras corregirlo ya me funcionaba la fase 1. Después probé la 3 con el código hecho en la 2, modificando una variable, y también funcionaba correctamente. Por lo que la práctica 5 estaba hecha correctamente y aparentemente terminada.