Carlos Martin Sanz Grupo: T2 Grupo Prácticas: X4

CUADERNO DE BITACORA

/*/**

******************************PRÁCTICA 3****************************/**

/*/**

DÍA 17/04/2021

Hoy he empezado con la segunda parte de las prácticas, y para empezar he creado un directorio "Practica 3" dentro de /root y como consta de 2 programas, dentro de estos otros dos directorios. Llamados "Parte 1" para "CreaProcesos.c" y "Parte 2" para "leeAout.c".

Ahora con todo estructurado voy a empezar a programar "CreaProcesos.c". Cuando más o menos pienso que tengo la estructura básica del programa compilo y me da un montón de errores, entre ellos con la declaración de variables y además me faltan algunos #include como <errno.h> .

EXPLICACIÓN: en MINIX la declaración de variables se realiza justo después de declarar el main, y yo algunas de las variables las declaraba después de comprobar el número de argumentos entonces, me daba errores, o dentro del for que he dejado de forma esquemática para tener como "la estructura".

He dejado hecho que me creé procesos e imprime un mensaje cada vez que crea uno, pero nunca acaba el programa, no funciona bien.

DÍA 18/04/2021

He continuado con lo de ayer, intentando solucionar porque me creaba hijos infinitos y el programa no paraba. He cambiado bastantes partes del programa porque no entiendo porque no finaliza y por tanto me crea hijos de forma infinita.

EXPLICACIÓN: He buscado en internet y me he dado cuenta de que no había "parseado" el argumento a int, es decir argv[1], aunque le pusiese ./CreaProcesos 5, me cogía el 5 como si fuese una cadena entonces tenía que hacer un atoi, int numHijos = atoi(argv[1]). Entonces eso podría hacer que el bucle for no parara.

Al compilar me ha saltado un error, y es porque no he importado <stdlib.h>.

Ahora ya el programa para, pero sigue sin funcionar, pero he decidido dejarlo porque entre cambiar el programa varias veces hasta que he dado con la solución de lo de atoi he perdido bastante tiempo. Y tengo que seguir mirando porque me sigue creando hijos de más...

DÍA 21/04/2021

Durante el horario de las clases de laboratorio he conseguido hacer que el programa funcione correctamente.

EXPLICACIÓN: no esperaba a que terminase uno de los hijos para crear otro, entonces se me superponían los hijos y se crean más de los que le paso como argumento.

He probado con waitpid(pid, &stado, 1); pero no funciona, también con waitpid(pid, &stado, 0); pero no funciona, y me he comido la cabeza y perdido bastante tiempo con este error, hasta que por casualidad di con el error. Poniendo un exit(0); ya estaría.

Solución: pid = fork(); cuando pid == 0 (todo correcto), se imprime el número de hijo y se sale con exit(0), y si no es todo correcto (Por ejemplo, cuando has llegado al límite de hijos que te permite crear) pid == -1 entonces te dice el número de hijos que has creado y te imprime el error. Voy a escribir el código de la "chicha" prescindiendo de la declaración de variables y del if que comprueba el número de argumentos.

```
for(int i=0; i<numHijos;i++){
pid = fork();
        if (pid == 0){
printf("He creado el hijo Nº %d",i);
exit(0);
}else if(pid == -1){
printf("Has creado estos procesos (limite): %d\n", i);
perror("Error "); (Mensaje que suelta cuando superas el numero de hijos)
exit(1);
}
}
```

He comprobado que me crea el número de hijos que le pido, pero ahora me falta probar que funcionan los errores y saber cuál es el numero límite de hijos que permite crear. Aunque justo se me ha acabado la clase de laboratorio y ya no he podido continuar, así que mañana compruebo esto.

DÍA 22/04/2021

Hoy no pretendo avanzar, y estar poco tiempo, simplemente comprobar que CreaProcesos.c funciona correctamente. Como ya pude ver ayer me crea los hijos de forma correcta. Ya lo probé ayer, pero he vuelto a probar si solo pasaba un argumento y te imprime "Uso: CreaProcesos numeroHijos", también funciona correctamente, Y por último ver el límite de hijos que puedes crear, he ido probando de 10 en 10 ya que no sé exactamente cuántos pueden ser, me sonaba que Julián dijo en clase 27 o 28, entonces por eso me voy a guiar de 10 en 10.

EXPLICACIÓN: Pero he tenido un problema, es que como me imprime una línea por cada hijo, no me salta ningún error cuando supera los permitidos (ya qué he probado con números

muy grandes como, 600 y me muestra por pantalla que crea los 23 primeros, entonces no puedo saber cuál es el límite) , ya que toda la consola esta ocupada por esos prints así que para poder probar esto he modificado el código y comentado esta línea ahora he probado y con 10 y 20 me los crea bien ya que no muestra nada por pantalla, al probar con 30, la salida es:

Has creado estos procesos (limite): 27

Error: Not enough core

Por tanto, el número de procesos hijos máximos que puedes crear es 27, y cuando intenta crear el 28 da un error que es Not enough core.

DÍA 23/04/2021

Hoy no tengo pensado escribir código con respecto a leeAout, sino que quiero leer que requiere el programa y pensar como debo empezar el código, o como debo encaminar el programa. Y ya cuando tenga esto claro empezar mañana o pasado a escribir el programa.

Después de 30-40 minutos pensando y buscando información en internet he llegado a la conclusión de que:

-Necesito crear una struct exec que va a ser la que tenga los datos

-Necesito abrir el fichero en modo lectura que se pasa como argumento para sacar los datos

-Un método que se llame información o algo similar para que printease toda esa información.

DÍA 24/04/2021

He empezado haciendo la estructura básica del programa, pero no pretendo avanzar en la parte difícil del programa.

Hoy he dedicado poco tiempo a escribir código, pero lo he dejado bastante bien armado para empezar a entrar con la parte difícil mañana. Simplemente he declarado el main, un struct exec y una variable tipo FILE (para el archivo que se pasa como parámetro). La estructura se llama datos (“struct exec datos” pues a informacion se le pasa datos como argumento informacion(datos)). Además, también he hecho el if que sirve para comprobar el número de parámetros y el if que comprueba si le has pasado un fichero o no. Ambos imprimen un mensaje de error. También he pensado que, para leer los datos del fichero, se necesitará la función fread, pero mañana cuando empiece con la funcionalidad ya veré.

DÍA 25/04/2021

En la struct que definí ayer, voy a almacenar todos los datos, y para poder leer del fichero voy a utilizar fread.

Tras implementar fread me daba errores de compilación.

EXPLICACIÓN: estaba pasando argumentos de forma incorrecta a fread por ejemplo, la forma correcta de argumentos es así: fread(&datos,sizeof(struct exec),1,fichero), pues al principio en vez de &datos, le pasaba directamente datos, el orden de sizeof() y de 1. Al consultar el manual de Linux (man fread) me di cuenta de que los estaba utilizando mal y solucioné el error.

Ahora por la tarde, he empezado con el método que tenía que printear los distintos datos sobre el fichero. Pero no he sabido cómo hacerlo, así que antes de escribir nada de código he buscado información en internet. Me ha llevado un rato bueno, hasta que he encontrado una página que me explica los distintos campos del exec header, la estructura que yo he llamado datos. La dejo aquí "https://www.freebsd.org/cgi/man.cgi/a.out(5)" además la página me da el paquete que hay que incluir para que no diese error en compilación.

Como ahora ya se mas o menos por donde hay que tirar. Voy a dejarlo ya avanzaré otro día, ya que me ha costado encontrar una página que me ayudase a conocer los distintos campos a printear y me ha llevado bastante tiempo.

DÍA 27/04/2021

He consultado la página del último día que me puse, y ahí me salen distintos campos de la estructura. Entonces he creado el método void informacion, donde le paso la estructura exec. Para printear un dato, he probado a hacer prinft("Bss size (bytes): %d\n", m.a_bss); y aunque sí que me imprimía bien la información, aunque he vuelto a mirar la página y... Estaba imprimiendo los datos como %d (decimal) y no como %ld (long).

EXPLICACIÓN: como bien decía la página, solo que no me fije. Muchos de los datos de la estructura exec eran long por lo tanto no podía utilizar %d, si no que %ld, ahora ya el printf que había puesto de prueba no solo funcionaba, sino que también era correcto, solo faltaba seguir imprimiendo más campos.

DÍA 28/04/2021

Durante el horario del laboratorio, y en apenas 15 minutos ya he hecho todos los prints de los datos que me salía en la página web. Pero he seguido buscando y se pueden imprimir algunos más como CPU ID, longitud y la versión entre otros. Así que he añadido los respectivos prints en el método void informacion(struct exec m):

printf("Magic number: %u\n", m.a_magic);

printf("CPU ID: %u\n", m.a_cpu);

printf("Lenght: %ld\n", m.a_text);

printf("Text size (bytes): %u\n", m.a_text);

… entre otros

Compila y al ejecutar aparentemente me imprime todos los datos de forma correcta, esta es la salida para un archivo en concreto que he creado yo para probar el programa:

Magic number: 138944

CPU ID: 97

Length: 97

Text size (bytes): 1835101549

Data size(bytes): 1835101537

...

así que he acabado la práctica 3, y justo hoy, día de laboratorio toca presentar la práctica 4. Así que he dejado de hacer cosas y he estado atento para enterarme de como se hace la siguiente práctica.

/*/**

******************************PRACTICA 4****************************/**

/*/**

DÍA 30/04/2021

He empezado leyendo la práctica y lo que había que hacer. También según he ido leyendo he ido revisando las líneas de código que decía la práctica, y cual se ejecutaba en modo usuario y cual en modo Kernel o privilegiado, y tomando nota de hasta qué punto era en modo usuario y después modo privilegiado. Viendo por encima el guion y siguiéndolo hasta la ejecución de int 33 es en modo usuario y luego cambia. Pero como hoy he dedicado poco tiempo, ya cuando me ponga más tiempo con la practica lo explicaré mejor.

DÍA 02/05/2021

Voy a empezar con una breve explicación de los pasos que realiza cuando se produce una llamada al sistema.

Primero cuando se produce una llamada al sistema, la función fork() se enlaza con _fork.c que, observando su código, retorna _syscall, con 3 argumentos. El primero, MM, el destinatario de la llamada al sistema, FORK es la operación que MM queremos que realice y por último, &m donde se va a depositar los valores devueltos por MM, es una dirección de memoria. Observando el código de _syscall.c se observa en la línea “status = _sendrec(who,msgptr) que es otra función, lo que va a hacer es enviar un mensaje al servidor que es MM, who seria MM quien lo va a recibir y msgptr el mensaje que va a recibir. Y esto nos lleva a otra función _sendrec .

Dentro del código de _sendrec.c se puede observar _send, _recieve y _sendrec, dentro de esta última esta lo que interesa, todo el código está escrito en lenguaje ensamblador. Hay instrucciones mov push etc, pero la más importante es int SYSVEC la que va a colocar en registros los argumentos de las funciones y luego realiza la llamada al sistema, el argumento de esta es 33 como se ve en la cabecera del código.

Todas las llamadas que llaman a otras o a funciones vistas anteriormente, se ejecutan en modo usuario, a partir de ahora con la instrucción, a través de todo el hardware, se pasa a modo privilegiado. Por tanto, ya no va a ser /usr/scr/lib sino /usr/src/kernel, la ubicación de las funciones y llamadas.

DÍA 03/05/2021

Continuo donde lo dejé ayer, donde la instrucción int 33 (SYSVEC) hace que se pase de modo usuario a modo kernel. Siguiendo el guión de la práctica nos lleva a la función prot_init donde inicializa las tablas de vectores de interrupción (set up tables for protected mode como dice el propio código). En la línea 206 se asocia esta interrupción con s_call "int_gate(:qSYS386_VECTOR, (phys_bytes) (vir_bytes) s_call …" y esta s_call invoca a sys_call() . Dentro de proc.c tiene 2 funciones mini_rec y mini_send, basicamente las llamadas al sistema en Minix, se implementan pudiendo recibir y enviar estos mensajes, en este caso sys_call() envía el FORK mensaje, a MM nuestro servidor.

Después de que el mensaje haya sido enviado a MM el servidor, se ejecuta la función restart, que recupera y permite la ejecución de otro proceso, y de nuevo se pasa de modo kernel o privilegiado a modo usuario.

De forma resumida, la gestión se realiza en modo usuario, hasta antes de ejecutarse la instrucción int XX, que al ser una llamada al sistema XX es 33. Cuando se ejecuta esa instrucción se pasa a modo privilegiado, se ejecutan determinadas funciones que llaman a otras y después de que el servidor (MM) reciba el mensaje, se ejecuta la función restart() y continua con la ejecución de otro proceso.

Con la explicación paso a paso de las distintas funciones y llamadas ya he ido explicando el proceso y conjunto de pasos hasta que se realiza una llamada al sistema en Minix.

DÍA 04/05/2021

Para localizar la función gestora de memoria primero, tengo que localizar todos los archivos .c que se relacionan con el servidor, que va a ser quien reciba el mensaje. Como he explicado cuando he explicado las llamadas al sistema. He estado buscando donde podría estar el directorio, y durante un rato navegando por los directorios he llegado a /usr/src/mm, ya que mm es el destinatario y tiene sentido que esté relacionado. Para ver todos los .c, he accedido a este directorio poniendo en la terminal de Minix cd /usr/src/mm y haciendo un ls *.c para ver todos los archivos c que hay en ese directorio.

He pensado que podía ser el alloc.c pero he probado a poner el print dentro de alloc_mem ya que en la descripción de esta decía que era la gestora de memoria cuando se producía un fork y he compilado y reiniciado, pero no. He buscado cual era el correcto hasta que he llegado a forkexit.

EXPLICACIÓN: De todos los archivos c que se muestran el que hay que modificar es forkexit.c, ya que aunque, el gestor de memoria me cuadra más que sea alloc.c. De todos los archivos .c que hay el único que tiene la palabra fork es este, entonces será el responsable de crear los procesos. Y de todas las funciones que tiene la encargada de crear los procesos es

do_fork, entonces para hacer que salude tengo que añadir un print, que indique está saludando como printf("CreaProcesos: Estoy Saludando");. Para que estos cambios se apliquen he de compilar el Kernel y reiniciarla. Para ello voy al directorio /usr/src/tools y ejecuto make hdboot para compilar el Kernel. Una vez que muestra done, ya ha finalizado la compilación de este. Y para comprobar si funciona el cambio aplicado, reinicio la maquina con shutdown –r now. Y cuando se reinicia la maquina se ve el print que hemos escrito. Por tanto, la práctica 4 funciona perfectamente y cada vez que crea un proceso imprime correctamente lo que hemos puesto en print. Una vez hecho y explicado aquí, he decidido comentar la línea del print ya que se me hacía muy incómodo que imprimiese cada vez. Entonces la comento y de esta forma demuestro que he realizado la práctica y me ahorro que imprima todo el rato el mensaje.

Para ello he comentado la línea, he compilado el Kernel y he reiniciado la máquina, de nuevo.

/********PRACTICA 5**********/**

/********/**

DÍA 06/05/2021

Para empezar a hacer la practica 5, que consta de 3 fases, voy a empezar por la primera fase siguiendo el guion de la práctica y realizando las modificaciones necesarias. En esta primera fase no voy a añadir código, ya que se encuentra en el guion y solo hay que ir siguiéndolo.

Como dice el primer punto, acceder al archivo de cabecera callnr.h, y modificarlo para incluir una nueva llamada. Se inserta al final con el nombre que se indica.

Una vez hechos estos cambios he seguido observando la función main() de main.c u la línea error = (*call_vec[mm_call])();

Después, he modificado table.c que estaba en el mismo directorio que el main.c así que con hacer un vi table.c ya estoy. Creo que la función principal de este fichero es definir el vector de las llamadas al sistema, por ello cuando creamos una en callnr.h después hay que venir a este fichero y añadirla en orden correcto, sino es como si no hubieses creado nada porque no está definida en el vector. La añadí al final, en el lugar correspondiente con respeto a callnr.h.

Luego, he modificado el archivo de cabecera proto.h, añadiendo la declaración de do_esops() de forma similar a do_reboot(), añadiendo esta línea justo después de la declaración de do_reboot: _PROTOTYPE(int do_esops, (void));

He probado a guardar todos los cambios, compilar el Kernel y reiniciar la máquina, pero, me da un error. He revisado de nuevo los pasos que he hecho y ...

EXPLICACIÓN: me daba un error, relacionado con el tamaño del vector. Y el error era que en el #define NCALLS 77, y al añadir una nueva llamada al sistema había que incrementarlo en 1 y poner #define NCALLS 78, he guardado cambios y recompilado el Kernel y reiniciado la máquina.

Estos cambios me han llevado alrededor de una hora, sobre todo por el error que no he sabido exactamente el porqué, hasta que he revisado todo y lo he visto, no he podido seguir aplicando el resto así que lo he dejado para otros días, ya que no voy mal de tiempo.

También he probado los comandos "yy" y "p" del guion, que básicamente sirven para copiar una línea y pegarla.

DÍA 10/05/2021

Voy a seguir aplicando los cambios necesarios. He modificado el archivo utility.c añadiendo la función que envía el mensaje a la tarea del sistema, escribiendo el código que dice el guión. Una vez realizados estos cambios "sin riesgo" ahora toca modificar el Kernel, donde tengo que ir con más cuidado para no destrozar la máquina virtual.

Entonces vamos al directorio correcto "usr/src/kernel" añadir el prototipo de la función llamada do_esops() en el fichero system.c. Escribiendo "FORWARD _PROTOTYPE (int do_esops, (message *m_ptr)); Después de realizar todos los cambios como añadir ESOPS al switch y definir la función escribiendo el código del guion voy a compilar el kernel a ver si no da ningún error. He compilado el núcleo y no da ningún error.

Mucha suerte haber realizado todos los cambios de golpe y que no diese ningún error, aunque esto que he hecho es un error.

EXPLICACIÓN: ya que si vas realizando cambios poco a poco y compilando y reiniciando la maquina en caso de que diese algún error sabes dónde puede estar de forma acotada, ya que los cambios que has hecho son "pocos". Sin embargo, si realizas todos los cambios y luego compilas el error puede estar en cualquier cambio de todos los que has hecho. Por suerte como los cambios están guionizados en la práctica los he ido haciendo con cuidado y no he cometido ningún cambio. Pero no es la forma correcta de hacerlo.

Ahora cree un subdirectorio dentro de root y cree un fichero practica5.c, donde empezaría a realizar la fase "Proceso de usuario".

Para empezar con el código, primero, he consultado en este fichero de cabecera como se declaraba una variable tipo message (/usr/include/minix/type.h,) y he tirado unas fotos con el móvil para poder consultarlas mientras escribo mi código en la práctica 5. Como indica el guion he empezado poniendo los includes correspondientes.

DÍA 12/05/2021

Durante la clase del laboratorio de hoy he comenzado, lo primero he borrado el archivo que empecé haciendo el último día porque era un desastre y he preferido hacerlo de cero de forma correcta añadiendo los includes que se piden, declarando la variable de tipo mensaje, asignando a msj.m1_i1, msj.m1_i2, msj.m1_i3, unos valores,100,200,300 respectivamente, haciendo la llamada al sistema con _taskcall(MM, ESOPS, &msj), y por último imprimiendo estos campos como por ejemplo con la línea, printf("M4: campo1= %d, campo2=%d, campo3=%d\n", msj.m1_i1 …"), en los puntos suspensivos faltarían las otras 2 variables de tipo message.

He probado a compilar y ejecutar, y muestra por pantalla los valores correctos que le he asignado, como todo funciona voy a seguir con la fase 3.

Salida:

MM: do_esops: 100 200 300

KERNEL: do_esops: a1 = 100, a2=200, a3=300;

Ahora para seguir con la práctica, hay que convertir la llamada en un distribuidor utilizando switch y añadiendo nuevos casos. Para ello voy a empezar modificando el archivo /usr/ src/kernel/system.c añadiendo ahí el switch de forma correcta para ello voy a añadir este código:

```
switch(a1){    /*según la llamada al sistema*/

case CASO1: do_caso1(&m); break;

        case CASO2: do_caso2(&m); break;

        case CASO3: m_ptr->m1_i2 = a2+65; break;

        default: printf("error en la llamada al sistema\n");

        }
```

Una vez que he modificado el código he intentado compilar el Kernel para ver si daba errores, pero me decía que CASO1, CASO2 y CASO3 estaban indefinidos y por tanto daba error. Para añadirlos me di cuenta revisando lo que había hecho en la fase 1 de esta práctica que tenía que ir a /usr/include/minix/callnr.h y añadir las etiquetas CASO1, CASO2 y CASO3 con los números asociados 1,2,3, poniendo al final, después de definir ESOPS: #define CASO1 1, #define CASO2 2, #define CASO3 3.

Ahora ya compilaba el núcleo, y podía seguir modificando cosas.

DÍA 13/05/2021

Quiero continuar modificando cosas, porque en la sesión de ayer del laboratorio me aclaró bastantes dudas de como seguir, para que funcionase. Y con las explicaciones creo que podría realizarlas y acabar hoy la practica 5, al menos hasta la parte de hacer do_esops() distribuida.

Para añadir las nuevas llamadas al sistema que se producen en el caso1 y caso2 del switch, al igual que cuando hice do_esops, me he ido al fichero system.c "vi /usr/src/kernel/system.c" y añadir debajo de do_esops, dos líneas de código con las 2 nuevas llamadas.

```
FORWARD _PROTOTYPE (int do_caso1, (message *m_ptr));

FORWARD _PROTOTYPE(int do_caso2, (message *m_ptr));
```

Una vez que han sido añadidas en el "inicio" del archivo, voy a hacer algo de forma similar a do_esops, añadir las 2 nuevas llamadas que lo que van a hacer sea modificar los valores de forma distinta y luego imprimirlos cuando ejecute practica5.c, de esta forma se van a ver en funcionamiento las 3 llamadas (gracias al switch), do_esops, do_caso1, do_caso2.

En el caso de do_caso1, el código que he implementado modifica la variable 2 y 3 asignándoles valores aleatorios que he elegido y deja la 1 como estaba, para hacerlo me he fijado en el código de do_esops y en el código que explico ayer Julián, modificando distintas variables para ver el funcionamiento, el código es este:

```
/*============do_caso1==========*/

PRIVATE int do_caso1(m_ptr)

register message *m_ptr;

{

m_ptr->m1_i1 = 50;

m_ptr-> m1_i2 = 4000;

m_ptr-> m1_i3 = 540;

printf("KERNEL: do_caso1: a1=%d, a2=%d, a3=%d\n", m_ptr->m1_i1, m_ptr->m1_i2, m_ptr->m1_i3);

}
```

Ahora el código de do_caso2 que en este caso modifica, las 3 variables dándoles valores distintos:

```
/*============do_caso2==========*/

PRIVATE int do_caso2(m_ptr)

register message *m_ptr;

{

m_ptr-> m1_i1 = 237;

m_ptr-> m1_i2 = 2100;

m_ptr-> m1_i3 = 1240;

printf("KERNEL: do_caso2: a1=%d, a2=%d, a3=%d\n", m_ptr->m1_i1, m_ptr->m1_i2, m_ptr->m1_i3);

}
```

Ahora creo que están todos los cambios hechos de forma correcta y no falta nada, así que voy a compilar el kernel y reiniciar la máquina. Compila por tanto voy a ejecutar la ./practica5 y me imprime los campos y todo como al principio y un "error en la llamada al sistema" es decir ejecuta el caso default del switch de do_esops, ya que no había utilizado ninguno de los "case" entonces ejecutaba el default.

He pensado que podría estar mal, y me he dado cuenta de que para probar los nuevos casos debería de en msj.m1_i1 = 10; en vez de poner un valor, llamar a uno de los case del switch poniendo msj.m1_i1 = CASO1; para ir probándolas.

Funciona de forma correcta ya que la salida ahora es:

MM: do_esops: 1 200 300

KERNEL: do_esops: a1 = 1, a2=200, a3=300;

KERNEL: do_caso1: a1 = 50, a2 = 4000, a3 = 540

M4: Primer Campo=4000, Segundo Campo=4000, Tercer campo=540

Si en vez de poner msj.m1_i1 = CASO1, pongo CASO2, me cambia las variables también de forma correcta, ahora la salida es esta:

MM: do_esops: 2 200 300

KERNEL: do_esops: a1 = 2, a2=200, a3=300;

KERNEL: do_caso2: a1 = 237, a2 = 2100, a3 = 1240

M4: Primer Campo =2100, Segundo Campo =2100, Tercer campo =1240

Y lo mismo con CASO3 que deja el Primer campo con valor 3 ya que es el CASO3, el segundo campo le suma 65 al valor que tenía la variable, y el Tercer campo, lo deja como está (300), la salida es esta:

MM: do_esops: 3 200 300

KERNEL: do_esops: a1 = 3, a2=200, a3=300;

M4: Primer Campo =3, Segundo Campo =265, Tercer campo =300

Ejecutando los códigos me he dado cuenta de que, aunque me lo modifica todo bien, al final en el último print siempre campo1 y campo2 toma el mismo valor. Eso es porque me he confundido en algún archivo y en vez de poner msj.m1_i1 he puesto msj.m1_i2 .

He mirado y todo parece estar aparentemente correcto sin errores y con todos los cambios bien realizados, así que he decidido mandar un correo a Julián para que me ayude a arreglarlo que, aunque es un cambio menor, ya que do_caso1 imprime bien los resultados, me gustaría que funcionase todo de forma correcta.

DÍA 14/05/2021

Hoy he visto la respuesta al correo que envié ayer, de nuevo he vuelto a revisar todo de nuevo la fase 2, y todos los cambios. Y todas las asignaciones de m_ptr->m1_i1 estaban bien, en especial he revisado system.c como decía la respuesta al correo. Tras revisar todo y no ver ninguna asignación mal, he mirado toda la práctica desde el principio, las 3 fases. Y todo estaba bien hasta que he llegado a revisar de nuevo la fase 1 y en utility.c me he dado cuenta del error, copié mal el código que había que escribir, la variable externa result2 tomaba el valor de m1_i2.

EXPLICACIÓN: sustituir result2 = mm_in.m1_i2 por result2 = mm_in.m1_i1, ahora faltaba guardar los cambios en el fichero, recompilar el núcleo y reiniciar la máquina. Ahora al ejecutar ./practica5 ya ejecutaba de forma correcta y los 3 campos tomaban los valores correctos en función del CASO. Imprimía los valores correctos.

El error se encontraba en la parte guiada de la práctica, es decir, había copiado mal el código que me daba el guion. Por tanto, cuando probé la parte de la llamada al sistema sin ser distribuidor, el error debía de estar presenté solo que no me di cuenta. Pero al probar la llamada como distribuidor si y lo he solucionado. Sustituyendo lo que he comentado anteriormente.

De esta forma ya he acabado la practica 5 y con ello el diario/cuaderno de bitácora.