Parte 3 simulador de kernel SESO

Alejandro Pérez

Durante el desarrollo del curso de Sistemas Operativos hemos aprendido cómo funcionan las entrañas de un sistema operativo, desde el scheduler y las políticas de cola de procesos, hasta la gestión de la memoria.

Además, hemos desarrollado un proyecto que, sin duda, ha sido la manera más directa de relacionarnos con el temario trabajado, ya que tener que implementar mecanismos y políticas es la manera más eficaz de familiarizarse con el funcionamiento de estas. En las partes 1 y 2, nos encargamos de definir la estructura general: diferenciar componentes del kernel y lograr que se comuniquen, así como la arquitectura de la máquina. Para esta tercera parte nos encargamos de cargar programas, asignarlos a hilos hardware y simular su ejecución. Este es el informe que explica qué cambios se han dado desde la última entrega, así como una explicación del código.

Índice

Diseño del sistema

En este apartado explicaré qué hace cada fichero, y también mencionaré el diseño general del código y partes de este que merecen mencionar

Kernel_Simulator.c

Este fichero c actúa como el main. Se encarga de inicializar 'el hardware' y de crear los
hilos. Recibe varios parámetros en el formato:
num_CPU, num_core/CPU, num_hilos/core, f_pGenerator, f_Scheduler
Así se determina la estructura del hardware que se simula, y también la frecuencia a la que
se generan los procesos y a la que se planifican. No ha sufrido cambios que merezcan mencionar desde la parte anterior.

Clock.c

Este fichero se encarga de dar los pulsos del reloj. Aunque no es realista, el reloj acaba a que los timers acaben sus tareas antes de dar otro pulso de reloj, así la simulación queda más sencilla de seguir. En cada pulso llama a la función reducirTtl, que mantiene el nombre de la parte anterior pero cambia funcionalidad.

Esta función se encarga de recorrer todos los hilos hardware, y para cada proceso en ejecución que encuentre, ejecuta la instrucción que el PC del proceso esté apuntando. Aumenta el PC en 4 Bytes para que apunte a la siguiente instrucción.

Código que merece mencionar:

```
//Sustituye la primera traducción no válida del TLB con la traducción encontrada en la pgb
//Devuelve -1 si no está en el pgb
int actualizar_TLB(unsigned int dir,HwThread_t hilo){
    int res = -1;
    //Busca el primer TLB no válido, si todos son válidos sustituye el último arbitrariamente
    int done = -1;
    int index = 0;
    while (done < 0 && index < 4){
        if (hilo.MMU.TLB[index].valid == -1){
            done = 1;
        } else {
            index++;
        }
    }

    done = -1;
    int index2 = 0;
    while (done < 0 && index2 < hilo.proceso.mm.num_frames)
    {
        unsigned int mask;
        mask = (1 << 5) - 1;
        //Si los primeros 19 bits de los 24 de la dirección virtual coinciden con los primeros 19 del puntero del frame, significa que está en ese frame
        if ((dir & mask) == (hilo.proceso.mm.pgb[index2] & mask))
        {
            done = 1;
            hilo.MMU.TLB[index].physical = hilo.proceso.mm.pgb[index2];
            hilo.MMU.TLB[index].virtual = (dir & mask);
            hilo.MMU.TLB[index].valid = 1;
            res = index2;
        } else {
            index2++;
        }
        return res;
    }
}
```

Esta función se llama en la siguiente función:

```
//Función que busca si la memoria virtual está en la TLB
//Si no está, actualiza la TLB
```

```
//Devuelve el índice del TLB, devuelve -1 si la dirección no corresponde
al pgb
int esta_en_TLB(unsigned int dir,HwThread_t hilo){
    int res = -1;
    unsigned int mask;
    mask = (1 << 5) - 1;
    for (int i = 0; i < 4; i++)
    {
        //Si el TLB es válido y la dirección virtual está en la misma
página
        if (hilo.MMU.TLB[i].valid == 1 && ((hilo.MMU.TLB[i].virtual &
mask) == (dir & mask)))
        {
            res = i;
        }

    }
    if (res < 0)
    {
        res = actualizar_TLB(dir,hilo);
    }

    return res;
}
```

Como se explica en los comentarios, estas dos funciones sirven para saber en qué entrada TLB del MMU se encuentra la traducción de la memoria virtual proporcionada a la memoria física. Si no está en ninguna entrada, se trae desde la tabla de páginas,

Lo que merece mencionar es el diseño de la tabla de páginas. Según el enunciado, la tabla de páginas ocupa espacio físico dentro del kernel.

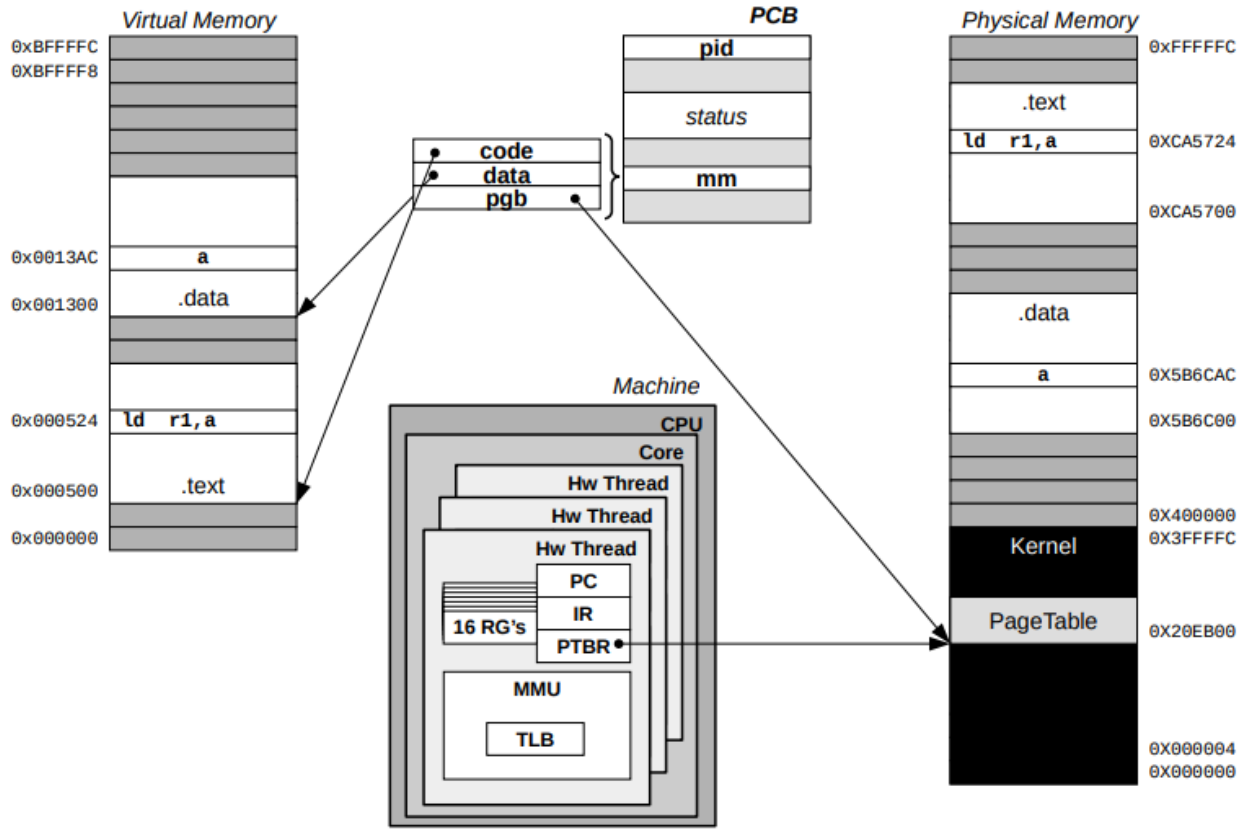


Figura 2. Esquema del Sistema de Memoria Paginada.

Implementarlo así SIGNIFICA tener que hacer los cambios en la tabla de páginas directamente en la memoria física, y resultaría demasiado complejo. Por tanto, la tabla de páginas se ha implementado para cada proceso. Cada proceso tiene su tabla de páginas en el mm.pgb, y el PTBR apunta al mismo sitio donde apunta el pgb.

También mencionar que arbitrariamente se han elegido 4 entradas TLB en la cache de los procesos para las traducciones dirección física <-> dirección virtual. Esto se debe a que ya que los programas creados con prometeo no superan las 64 instrucciones por lo general, no van a tener que usar más de una página. Esto es configurable, pero dejando 4 entradas se puede simular la cache TLB de la MMU y mantener una implementación más sencilla de cara a este proyecto.

Por último, la última función de apoyo en Clock.c es

```
//Busca la dirección física que coincida con la dirección virtual en la
TLB que contiene la info
//No tiene mucho sentido, ya que como estamos manualmente guardando los
datos en una memoria
//vacía del mismo tamaño que la memoria virtual, las direcciones son las
mismas, pero es una función que debería estar
unsigned int buscar_en_TLB(int index, unsigned int dir, HwThread_t hilo){
    if (index == -1)
    {
        return 0x00;
```

```
    }

    unsigned int res=0x0;
    for (int i = 0; i < 256; i++)
    {
        if (hilo.MMU.TLB[index].physical+i == dir)
        {
            res = hilo.MMU.TLB[index].physical+i;
        }

    }
    return res;
}
```

Esta función busca en el bloque de 256 direcciones que representa el TLB si la dirección está ahí, y devuelve el valor de la dirección física.

Por otro lado, la función principal *es reducirTtl()*. El mismo código está comentado. La dirección virtual corresponde al PC del proceso, se busca la traducción a una dirección de la memoria física y se obtiene la instrucción (se guarda en el Instruction Register). Después, según qué instrucción sea, se hace el ld, st, add o exit. Por último, se incrementa el PC para que apunte a la siguiente instrucción.

Timer.c

Este programa se encarga de recibir los pulsos del reloj, y cada cierta frecuencia despierta a loader y a dispatcher. No se ha modificado significativamente desde la parte anterior.

Schedule_dispatcher.c

Este programa se encarga de planificar los procesos en espera a los hilos. Estos procesos se encuentran en una lista dinámica, *lista_procesos*. Recorre todos los hilos, y cuando encuentra el primer hilo con un programa que ha terminado ya (*PCB.status* se lo indica), crea la información del hilo hardware y le asigna el primer proceso en la cola. En caso de que no hay procesos en la cola, lo notifica por consola con un mensaje.

```
if (machine.cpu_array[i].core_array[j].hilos[k].proceso.status == -1)
//Significa que el proceso ha acabado, es decir, el hilo está libre
{
        HwThread_t new_thread;
```

```
        new_thread.proceso = lista_procesos.first->me;
        new_thread.PC = lista_procesos.first->me.mm.data;
        new_thread.PTBR = lista_procesos.first->me.mm.pgb;
        new_thread.proceso.status = 1;
        for (int p = 0; p < 4; p++)
        {
                new_thread.MMU.TLB[p].valid = -1;
        }

        machine.cpu_array[i].core_array[j].hilos[k] = new_thread;
        asignado=true;

        // Si solo hay un proceso en la lista, el último va a ser null
        if (lista_procesos.last == lista_procesos.first)
        {
                lista_procesos.last = NULL;
        }
        // En cualquier caso, el rimero debe apuntar al siguiente (si es
el único se queda en null, como debería)
        lista_procesos.first = lista_procesos.first->next;
}
```

Cuando asigna el proceso, notifica por consola con un mensaje que dice en qué CPU, core e hilo se ha asignado el proceso con qué identificador. También dice dónde empiezan los segmentos *.data* y *.text*.

Loader.c

Este ha sido el programa más interesante de implementar. Se encarga de leer de ficheros los programas, crea los PCBs y los añade a la lista de programas en espera de ser asignados.

```
//Prepara el nombre del archivo, puede ser dentro del rango 000-999
char code[21] = {"Programas/prog000.elf"};
int resto = 0;
int resultado = ultimoid;
//ultimoid = xyz
resto = resultado % 10;    //resto = xyz mod 10 = z
resultado = resultado/10; //resultado = xyz div 10 = xy
code[16] = '0' + resto;   //prog00z
resto = resultado % 10;   //resto = xy mod 10 = y
resultado = resultado/10; //resultado = xy div 10 = x
code[15] = '0' + resto;   //prog0yz
code[14] = '0' + resultado; //progxyz
```

Ya que conocemos la sintaxis de los programas, podemos usar este código para conseguir el nombre. Todos los programas se encuentran en la carpeta **Programas** dentro de la carpeta *Source*.

Después se ve cuántos frames son necesarios para el programa, se redondea hacia arriba.

```
//Abre el archivo y mira cuántos frames son necesarios
FILE* programFile;

programFile = fopen(code,"r");
if (programFile == NULL)
{
printf("No hay más archivos que leer\n");
} else {
struct stat programInfo;
stat(code,&programInfo);
int framesProceso = (((programInfo.st_size-26)/9)/64); //He comprobado
que un archivo con las dos líneas iniciales y el último salto de línea
son 26 bytes, y por cada línea de instrucciones y el salto de línea son 9
bytes
if (((programInfo.st_size-26)/9)%64)
{
framesProceso++; //Esto es para redondear para arriba, he visto que es
común usar (x+(y-1))/y para redondear para arriba pero como no lo veo del
todo me quedo con esto
}
```

A continuación, deben leerse las líneas del fichero, sabiendo que las dos primeras líneas nos dan información sobre dónde empiezan los segmentos, y a la vez han de guardarse las entradas a los frames y todos los datos en la memoria física. Por último, se crea el PCB y se añade a la cola.

```
while (fgets(read,14,programFile))
        {
                printf("el dato leído es: %s\n",read);
                memcpy(valores,&read[6],6); // CONTROL : Antes estaba
puesto a copiar 7 bytes, que funcionaba, pero a saber qué pasa ahora
                if (i == 1) //La segunda línea nos dice dónde empieza el
código, anotar
                {
                        new_mm.code = (int)strtol(valores,NULL,16);  //
(NOTA) He probado con stroul pero no me funciona bien siempre, no sé
usarlo bien
                } else if (i == 0) //La primera nos dice el dato, anotar
```

```
                {
                        new_mm.data = (int)strtol(valores,NULL,16);
                } else // A partir de ahí guardar todo en memoria física
                {
                        //Si hemos leído un múltiplo de 256 de líneas
reservamos otro frame
                        if (instruccionesYaLeidas%256 == 0)
                        {
                                frameActual++;
                                new_mm.pgb[frameActual] =
lista_frames.first->address;
                                dondeGuardar = lista_frames.first-
>address;
                                lista_frames.first = lista_frames.first-
>next;
                        }
                        //Guardamos en memoria física los datos
                        unsigned int linea = (int)strtol(read,NULL,16);
                        memcpy(&memoria_fisica[dondeGuardar],&linea,8);
// CONTROL : Puede que falle esto (?)
                        dondeGuardar+=4;
                        instruccionesYaLeidas+=4;
                }
                i++;
        }
        // Crear el nuevo proceso
        process_info_t new_PCB;
        new_PCB.mm = new_mm;
        new_PCB.pid = ultimoid;
        new_PCB.status = 0;

        //Meterlo en la cola de procesos para el dispacher
        process_node_t *nodo =
(process_node_t*)malloc(sizeof(process_node_t));
        nodo->me = new_PCB;
        nodo->next = NULL;

        if (lista_procesos.first == NULL)
        {
                lista_procesos.first = nodo;
        } else {
                lista_procesos.last->next = nodo;
        }
        lista_procesos.last = nodo;
}
```

Se muestra por pantalla los datos que ha leído, así como información sobre el proceso al haberse creado completamente.

Otros ficheros

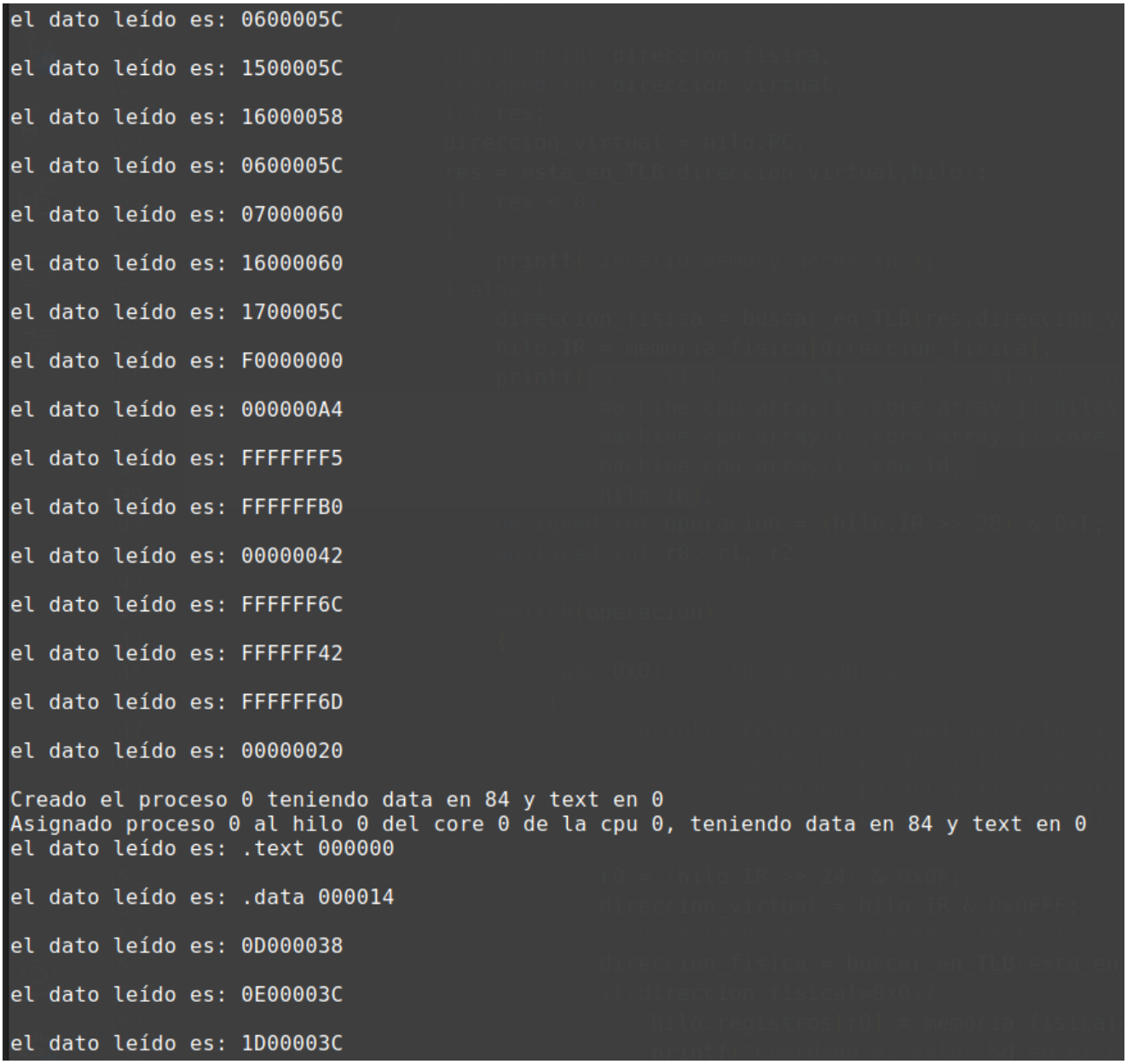
Globals.h contiene las variables globales, que, por lo general, se declaran e inicializan en *ernel_Simulator.c*.
data_structures.h contiene las definiciones de las estructuras de datos. Se han hecho cambios desde la parte anterior acorde con las especificaciones de este enunciado. En un principio, he optado por no usar punteros, pero han sido necesarios a medida que implementaba los programas principales.
Makefile permite compilar el proyecto de manera sencilla,
test.c es un programa de prueba usado para hacer experimentos y resolver dudas. me ha parecido oportuno dejarlo como parte del proyecto por si interesa ver el porqué se ha elegido programar de cierta manera en algunos casos (consiguiendo el nombre de los programas a abrir *rogXYZ.elf* por ejemplo).
process_generator.c se ha incluido sin cambiar a modo de tener comparación de los cambios del loader.
La carpeta **Programas** contiene 50 programas generados con prometeo, sin modificar las opciones que ofrece. El último programa, *prog050.elf*, lo he usado para hacer comprobaciones.

Manual de usuario

Para compilar el proyecto, basta con hacer make. Después habrá que llamar a Kernel_Simulator y pasarle los parámetros. Los parámetros indican el número de CPUs, cores e hilos por core que tendrá la máquina. También determina la frecuencia a la que se llama a loader y al dispatcher.
Por ejemplo:

```
./Kernel_Simulator 1 2 2 50000 50000
```

Esto nos dará una ejecución en la que se puede apreciar qué programas se leen de fichero y qué PCBs se crean, así como a qué hilos se asignan.



PENDIENTE : usar tag para ver información del clock

Conclusiones y propuestas de mejora

Los hilos se crean y se comunican correctamente, los programas se leen desde los ficheros de forma apropiada y con valores ciertos. Los PCBs se crean sin problema. Sin embargo, el proyecto cuenta con dos problemas principales que afectan a su correcto funcionamiento.

Se puede apreciar que los procesos se asignan a hilos libres, pero solamente se asignan 3 procesos. El resto de procesos se quedan en la cola, ya que no salta el mensaje de que la cola está vacía. No entiendo dónde puede estar la causa del error.

Con los primeros tres procesos funciona bien, con los siguientes no. Cambiando las frecuencias del loader y el scheduler no cambia nada.

El segundo problema es que los procesos que se asignan solo ejecutan el load. Más concretamente, se lee una fila de cuatro ceros, lo cual hace que la única operación que se lleva a cabo es de la forma 'ld 0,0x000'. Se puede apreciar en la captura:

```
Hilo 3 del core 1 de la cpu 0
la instrucción es: 0
Hilo 0 del core 0 de la cpu 0
la instrucción es: 0
Hilo 1 del core 0 de la cpu 0
la instrucción es: 0
Hilo 2 del core 1 de la cpu 0
la instrucción es: 0
Hilo 3 del core 1 de la cpu 0
la instrucción es: 0
Hilo 0 del core 0 de la cpu 0
la instrucción es: 0
Hilo 1 del core 0 de la cpu 0
la instrucción es: 0
Hilo 2 del core 1 de la cpu 0
la instrucción es: 0
Hilo 3 del core 1 de la cpu 0
la instrucción es: 0
Hilo 0 del core 0 de la cpu 0
la instrucción es: 0
Hilo 1 del core 0 de la cpu 0
la instrucción es: 0
Hilo 2 del core 1 de la cpu 0
la instrucción es: 0
Hilo 3 del core 1 de la cpu 0
la instrucción es: 0
Hilo 0 del core 0 de la cpu 0
```

Aquí se ha hecho que el clock.c imprima la operación que está llevando a cabo y escribe un '0'. Se puede apreciar que se ejecutan uno a uno los tres procesos asignados, como he mencionado antes. El error debe de estar al guardar los datos en la memoria física, ya que se leen bien (código del loader.c).

```
//Si hemos leído un múltiplo de 256 de líneas reservamos otro frame
if (instruccionesYaLeidas%256 == 0)
{
        frameActual++;
        new_mm.pgb[frameActual] = lista_frames.first->address;
        dondeGuardar = lista_frames.first->address;
        lista_frames.first = lista_frames.first->next;
}
//Guardamos en memoria física los datos
unsigned int linea = (int)strtol(read,NULL,16);
```

```
memcpy(&memoria_fisica[dondeGuardar],&linea,8);    // CONTROL : Puede que
falle esto (?)
dondeGuardar+=32;
instruccionesYaLeidas+=4;
```

He intentado de todo, pero no soy capaz de arreglar este error.

Arreglar estos dos aspectos supondría un funcionamiento correcto, y es es la mejora vital que necesita el proyecto.

Ha sido un proyecto muy interesante, y me hubiese gustado dedicarle más tiempo durante el primer cuatrimestre, ya que lo dejé para el segundo y ha resultado más ocupado de lo planeado. De todas formas, he aprendido mucho, y el proyecto de IOS me ha servido mucho en cuanto al aspecto de implementación. Por último, me gustaría agradecer a Olatz por las oportunidades que me ha dado durante el curso respecto a la asignatura.

Apéndice

Carpeta Source
Código en Github
Enunciado original