San Sebastián / Donostia

Simulador de un kernel

Sistemas Operativos

Iñaki García Noya

Universidad del País Vasco
Facultad de Ingeniería Informática
Ingeniería del Software

19/01/2021

Índice

1 Introducción **2**

2 Introducción primera parte **3**

2.1 Objetivos 3

3 Políticas del scheduler **4**

4 Estructuras **5**

5 Diseño **6**

6 Guia de uso y Mejoras **8**

7 Introducción segunda parte **9**

7.1 Problemas en el desarrollo: 9

7.1.1 Librerias 9

7.1.2 Mutex 10

8 Estructuras **11**

9 Diseño **13**

9.1 Posibles mejoras 18

10 Ejemplos de ejecución: **20**

11 Conclusiones **23**

1. Introducción

En este documento se documenta el proyecto realizado en la asignatura de Sistemas Operativos en la carrera de Ingeniería Informática. El proyecto tiene como objetivo simular un *kernel*. Para lograr este objetivo, el proyecto se ha realizado en 2 partes, por lo que este documento está dividido en 2 partes:

- **Primera parte:** construir un programa multihilo que simule el funcionamiento del kernel de un sistema operativo. Desarrollar un *Scheduler* (y *Dispatcher*) con la política (o políticas) de planificación que se decidan.
- **Segunda parte:** sobre la base del marco de funcionamiento construido en la primera parte en la que se ha desarrollado un *Scheduler* (y *Dispatcher*) con unas determinadas políticas de planificación, construir un sistema de gestión de memoria virtual y memoria física.

Para ello, en la Sección 2 se detalla la primera parte y en la Sección 7 se detalla la segunda parte. Al realizar el proyecto en dos fases, se ha mejorado la implementación de la Primera Parte en la Segunda Parte (todo ello especificado en su correspondiente apartado).

2. Introducción primera parte

En esta primera parte, se han realiza los apartados 1 y 2 del proyecto. Los enunciados proporcionados eran los siguientes:

1ª parte: Arquitectura del Sistema En esta primera parte se definirá la arquitectura del sistema. Sobre todo aquí hay que preparar el sistema para realizar las siguientes partes de la práctica. Para ello es necesario definir las estructuras de datos necesarias para realizar las simulaciones e implementar el thread principal que gestionará todo el sistema. Este thread tendrá varias tareas, por ejemplo, gestionará el reloj del sistema.

2ª parte: Planificador El planificador (o *scheduler*) gestiona la ejecución de los procesos. En la construcción de este subsistema se pueden utilizar diversas políticas y arquitecturas. En esta parte, teniendo en cuenta la descripción del hardware entregado, se deberá implementar una (o varias) política que maximice la eficiencia y el rendimiento de los procesos.

2.1 Objetivos

- El objetivo principal de la primera parte fue que el resultado funcione.
- Hacer uso de los punteros, para hacer un buen uso del lenguaje de programación de C.
- Definir diferentes políticas del *scheduler*.
- Implementar un lenguaje basado en módulos; es decir, con el uso de las librerías que proporciona c.
- Hacer uso de diferentes estructuras, y argumentar el por qué de su selección.
- Utilizar los hilos justos y necesarios.
- Realizar una memoria que argumente todas las elecciones realizadas y todas aquellas descartadas.

3. Políticas del scheduler

El primer paso del proyecto era seleccionar el tipo de política que iba a utilizar el *scheduler*. En un primer lugar se iba a utilizar una política expulsora por eventos con colas de prioridad para los procesos en estado de espera. Sin embargo, el profesor recomendó el uso del *clock* para avisar al*scheduler*, por lo que se ha utilizado un **híbrido**, para realizar un política diferente.

En un primer lugar, el programa funciona con el *timer* avisando al *scheduler* por cada cierto aviso del reloj (parámetro de entrada). Pero una vez que el programa se ha ejecutado, si llega a ocurrir que todos los hilos de los *core* están ocupados, se pasa a un ***scheduler* de tipo por eventos**. Para ello, se hace uso de un flag llamado *scheduler_flag*, el cual informa a todo el programa de este cambio. De esta manera, se crean hilos que se saltan cada vez que un programa tenga como *quantum* 0. En el caso de que se quiera desprender de esta implementación, bastaría con no cambiar el valor de este *flag*, por lo que no afectaría al desarrollo del proyecto.

Por otro lado, se ha comentado que se ha hecho uso de colas de prioridades. Los PCB disponen de un atributo llamado prioridad, el cual es valor número que tiene que estar en el rango comprendido entre 0 y 3, siendo 0 el proceso con mayor prioridad. De esta manera, se tienen creadas 4 colas(llamadas *queue*), las cuales almacenan solo los PCB de su prioridad correspondiente. Se ha utilizado esta política ya que es muy común su uso en las máquinas actuales.

El problema que tienen las colas de prioridades es que pueden estar expuestas a interbloqueo. Es posible que por probabilidad, se creen muchos procesos de prioridad superior, y en el caso de que se dispongan de pocos hilos, se puede dar el caso de que los procesos con menor prioridad nunca sean seleccionados. Para ello, se ha creado un método el cual de manera periódico incrementa las prioridades a los procesos de las colas. De esta manera, se asegura que nunca quede pendiente ningún proceso. Para finalizar, solo se dispone de un *scheduler*. En un primer lugar, la idea era la implementación de un *scheduler* por cada CPU, pero a raíz de su complejidad de sincronización no se ha implementado.

Dentro de cada cola; como su propio nombre indica, se sigue el sistema FSFS, por lo que no se tiene en cuenta el *quantum* del proceso.

4. Estructuras

Se ha seguido las recomendaciones del tutor para la estructura del programa. Todas las estructuras que se utilicen en el programa se encuentran definidas en **definitions.h** y tienen en su mayoría un tamaño máximo asignado de manera estática.

- En primer lugar se dispone de un lista de **CPU** *(arr_cpu[NUM_CPU])*.
- Este a su vez tiene una lista de **núcleos** *(arr_core[NUM_CORE])*.
- Los núcleos; *core*, disponen de una lista de **hilos** *(arr_th[MAXTHREAD])*.
- Los hilos: *thread*, están formados por:
  - *is_process:* booleano que indica si se encuentra un proceso en ejecución o no. Se considera que un proceso está en ejecución cuando su *quantum* es superior a 0.
  - *t_pcb:* proceso que se encuentra en ejecución.
- Los **procesos** se denominan PCB, los cuales están formados por:
  - *id:* identificador del proceso.
  - *quantum:* tiempo de vida del proceso.
  - *prioridad:* valor número que tiene que estar en el rango comprendido entre 0 y 3, siendo 0 el proceso con mayor prioridad.
- Las **colas** disponen de los comandos básicos que disponen las estructuras de tipo FIFO o FCFS.
- Se dispone de una estructura adicional llamada **parámetros**, la cual nos ayuda el paso de múltiples parámetros a los hilos.

Por otro lado, para la correcta sincronización de los hilos, se disponen de los siguientes *mutex*.

- ***mutexT:*** *mutex* del *scheduler*, que gestiona el acceso a la variable de control que sincroniza a el *timer* y al *scheduler*.
- ***mutexC:*** *mutex* del *timer*, que gestiona el acceso a la variable de control que sincroniza a el *timer* y al *clock*.
- ***mutexPCB:*** *mutex* que da acceso a las estructuras de la CPU *hardware*.

5. Diseño

Como se ha comentado previamente, se ha hecho uso de las **librerías** de C. En este apartado se indican el diseño del programa, mientras que en el siguiente se detallan algunos aspectos de programación. El proyecto se reparte en las siguientes :

kernel.c Programa principal en el cual se encuentra el *main*. Desde este, se invocan al resto de las librerías (por lo que no dispone de variables). Aún así, dispone de algún método extra, ya que no se sabía en cual de las librerías restantes era recomendado introducir. Los métodos:

- *int main(int argc, char *argv[]):* recibe los parámetros de entrada del *script*, gestión el formato de los parámetros y crea los hilos. Por defecto se ha definido un tiempo de espera, para que el programa termine en ese tiempo. Se puede eliminar esta ejecución para que el programa se ejecute de manera indefinida.
- *void inicializar():* inicializa las estructuras necesarias.
- *void asignarPCB(struct PCB pPcb):* asigna un PCB a un hilo del *core*.
- *void decrementarQ_ListaPCB():* decrementa en uno el *quantum* de cada PCB. Si es 0, asigna al hilo del *core* que no está ocupado.
- *void aumentarPrioridad():* aumenta las prioridades a las colas.
- *todosHilosOcupados():* recorre todas los procesos de los hilos, comprobando si todos los *cores* están ocupados.

thread.c Librería que contiene todos los métodos y relacionados con los *pthread*.

- *void *kernelClock(void *arg):* hilo que gestiona los ciclos del reloj. Multiplica el parámetro de entrada que indica la frecuencia del clock; el cual tiene que ser un múltiplo de 10, por un número que equivale a un segundo en mi ordenador.
- *void *timerScheduler(void *arg): timer* que avisa periódicamente al *scheduler*, para ello se comunica con el *clock* mediante el uso de un *mutex* y con el *scheduler* con el uso de otro *mutex*. También se encarga de gestionar el aumento de prioridades y el cambio de los *flags*.
- *void *processGenerator(void *arg):* hilo que crea procesos aleatorios continuamente. Cuando crea esos procesos, se encarga de introducirlos teniendo en cuenta su prioridad.
- *void *schedulerTiempo(void *arg):* hilo que gestiona el *scheduler* que se activa con el *timer*.
- *void *schedulerEvento(void *c_ptr):* hilo que se crea cuando todos los hilos están ocupados, y en el caso de que un nuevo proceso tenga el *quantum* a cero. Cuando acaba su cometido, se elimina el hilo.

queue.c Librería que contiene todos los métodos asociados al manejo de las colas.

- *struct Queue* createQueue():* crea una cola y devuelve su puntero.
- *int isFull(struct Queue* pQueue):* comprueba si la cola está llena. *int isEmpty(struct Queue* pQueue):* comprueba si la cola está vacía.
- *void enqueue(struct Queue* pQueue, struct PCB pcb):* introduce un elemento en la cola.
- *struct PCB dequeue(struct Queue* pQueue):* extrae un elemento de la cola.
- *struct PCB front(struct Queue* pQueue):* devuelve la cabeza de la cola.
- *struct PCB rear(struct Queue* pQueue):* devuelve el último elemento de la cola.
- *void printQueue(struct Queue* pQueue):* imprime la cola.
- *void subirPrioridadColas(struct Queue* pQueue1, struct Queue* pQueue2):* sube los procesos de una cola a la siguiente.

mensajes.c Librería para gestionar los mensajes y una prueba que indica en la salida estándar la clasificación de las estructuras.

- *void subirPrioridadColas(struct Queue* pQueue1, struct Queue* pQueue2):* imprime el mensaje que recibe como parámetro y termina la ejecución del programa.
- *void imprimirEstructura():* imprime las estructuras del programa de manera visual y ordenada.

6. Guia de uso y Mejoras

Prerequisitos: Es necesario tener instalado el compilador de C y el paquete *make*.

Instalación: En primer lugar, se compila el programa. Para ello, se utiliza el comando *make*. Si la ejecución ha sido la correcta, se puede ejecutar el script. Para ello:

./kernel frecuenciaClock frecuenciaTimer frecuenciaProcessGen

Un ejemplo de ejecución sería el siguiente. A recalcar que la frecuencia del clock tiene que ser un múltiplo de 10. *./kernel 10 10 10*

Mejoras:

- Añadir más listas dinámicas.
- Realizar una parametrización adecuada, que otorgue más opciones al usuario.
- Utilizar señales para notificar de ciertas acciones.
- Utilizar más punteros.

7. Introducción segunda parte

Los objetivos principales de esta parte del proyecto eran los siguientes:

- Después de realizar una autoevaluación de la anterior parte, y además por la petición del profesor, era necesario realizar una parametrización adecuada.
- Hacer que las listas de las estructuras (cpu, núcleos e hilos) sean dinámicas, para que se puedan actualizar con los parámetros.
- Obtener un código que funcione correctamente, sin errores de compilación (por ejemplo, *core-dumps*). Este objetivo es importante ya que el uso de los *threads* puede generar que en ciertas ejecuciones se encuentren problemas, cuando en otras no ocurren.
- Ofrecer más *feedback* al usuario. Para ello, se notificará en todo momento de la ejecución del programa para que el usuario sepa de su funcionamiento.
- Utilizar los *mutex* necesarios para evitar interbloqueo.

7.1 Problemas en el desarrollo:

7.1.1 Librerias

Esta parte del proyecto a resultado ser la más difícil, debido al número de errores obtenidos en el desarrollo. Gracias al *debugger* de C *(gdb)* se ha logrado solventar la mayoría de ellos. Aún así, uno no ha sido posible solventar. En la anterior parte, y gran parte del desarrollo de esta, se han utilizado las librerías de C (Véase Capítulo 5). Durante el desarrollo, se obtuvo un error que imposibilitó el uso de las mismas (Figura 7.1). No ha sido posible arreglar el error, ya que aunque se ha buscado una solución por los foros, el error no es muy común. Por ende, se ha decidido unificar todo el código (todos los ficheros ".c") en uno único, y hacer uso de la librería ***definitions.h***.

Como se puede ver en la Figura a continuación, se indica que cada una de las librerías vuelve a inicializar todas las estructuras.Se han eliminado los *include* sobrantes, pero no se ha podido arreglar.

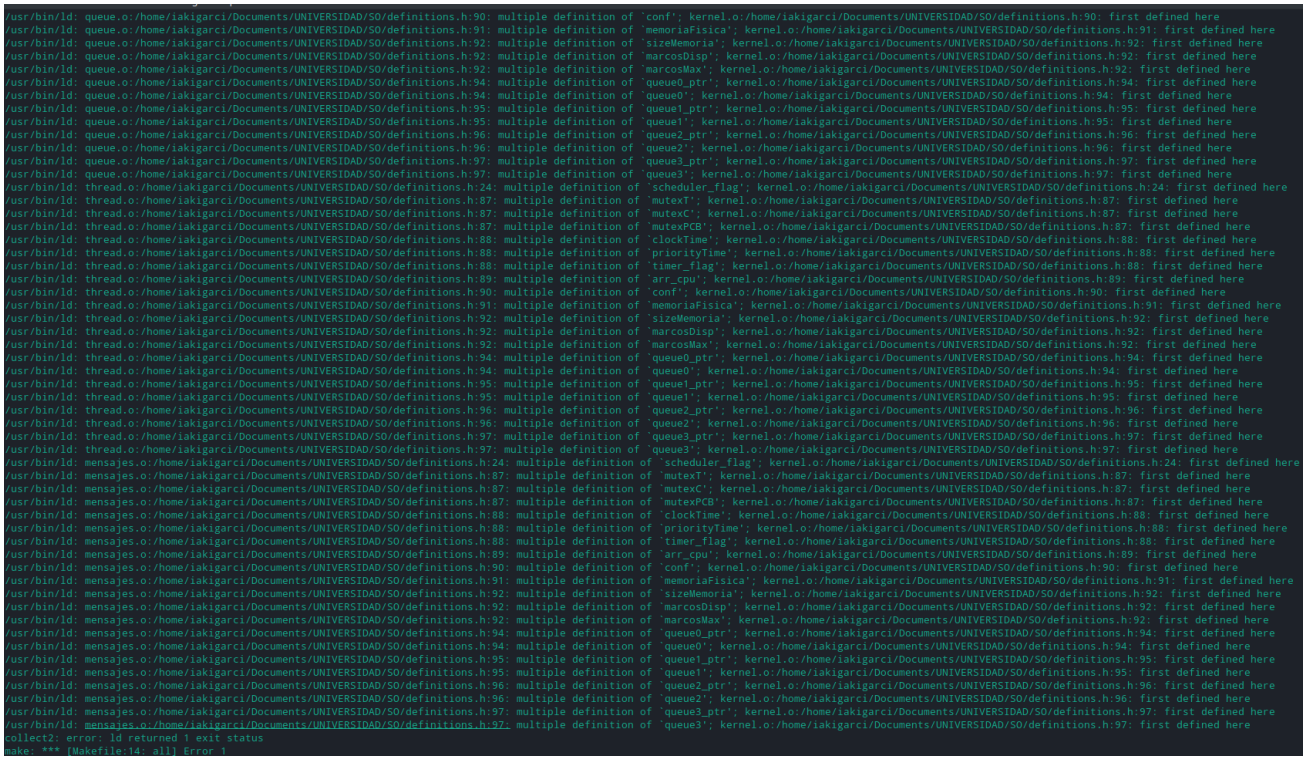


Figure 7.1: Error de librerías

7.1.2 Mutex

Por otro lado, se obtiene un error de lo más peculiar. De manera aleatoria, existe un momento en la ejecución del código, que se bloquea el *mutex* de la memoria física. Este *mutex* restringe la entrada de la memoria física, y es usado por 3 hilos: *scheduler*, *loader* y *timer*. Cuando el programa lleva un tiempo de ejecución, los 3 mutex se bloquean, pero no pasa en todas las ejecuciones.

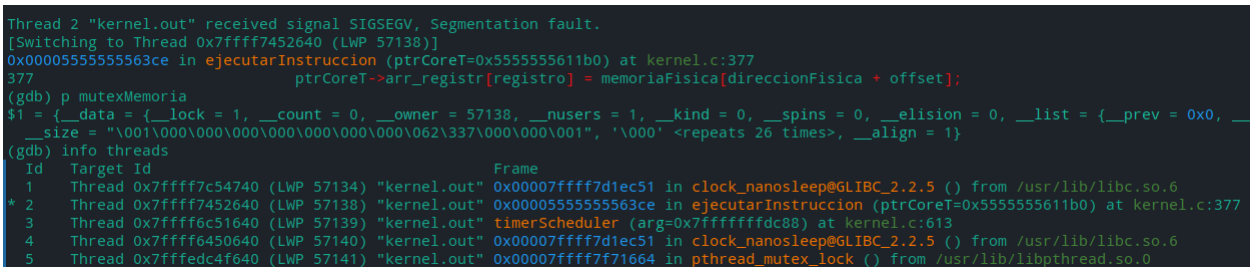


Figure 7.2: Error de mutex memoria

En la figura se puede puede comprobar como el *mutex* tiene como dueño el hilo actual (*owner=57138*). Una vez revisados todos los mutex, se puede observar que tiene interbloqueo. Se han repasado todos los *mutex* y todas las operaciones de acceso a memoria, y no se ha encontrado ningún error, por lo que no se le encuentra solución. Soy consciente que no es buena práctica tener *mutex* anidados, pero en el caso de borrarlos, se obtiene un *segmentation_fault*. Se ha realizado innumerable de pruebas: cambiar el *mutex* a métodos completos, poner muchos *mutex* pero solo donde es es específicamente necesarios, borrar todos los mutex... pero sin éxito.

8. Estructuras

Las estructuras han tenido un cambio significante. Las estructuras : *arr_capu*, *arr_core* y *arr_th* son dinámicas. Para ello, se solicita al usuario que indique las cantidades solicitadas, y si no quiere indicar ninguna cantidad, se dispone de valores por defecto:

- *NUM_CPU:* 2
- *NUM_CORE:* 4
- *MAXTHREAD:* 4

En su anterior parte, estas definiciones eran constantes, pero como se tiene que modificar su valor se han convertido en valores *int*.

Por otra parte, se ha modificado las estructuras del PCB:

- **PCB:**
  - *id:* identificador del PCB.
  - *quantum:* tiempo que dispone el proceso cuando es cargado. Cuando se acaba el *quantum* y no ha terminado de ejecutarse todas las instrucciones, se introduce en una nueva cola con uno nuevo (aleatorio).
  - *prioridad:* prioridad del PCB del 0 al 3.
  - **mm:**
    - *code:* donde empieza la sección del código del fichero.
    - *data:* donde empieza la sección de los datos del fichero.
    - *pgb:* tabla de páginas que tiene el propio PCB, con longitud dinámica.
  - **pcb_Status:**
    - *arr_registr:* 16 registros del PCB.
    - *IR:* ultima instrucción ejecutada.
    - *PC:* dirección virtual de IR.
    - **TLB:**
      - *virtual:* dirección virtual.
      - *física:* dirección física.

Para la gestión de la memoria, se han añadido los siguientes estructuras:

- *sizeMemoria:* tamaño de la memoria. Se calcula elevando al cuadrado el argumento que se pasa por parámetro. Este parámetro tiene que ser mayor que 8.

- *marcosDisp:* marcos disponibles.
- *marcosMax:* marcos máximos que tiene la memoria.
- *MEMORY_SIZE_DEFAULT:* multiplicador por defecto de la memoria, valor por defecto 20;

Se han introducido dos carpetas nuevas:

- **heracles:** dispone del programa *heracles*.
- **prometeo:** dispone del programa *prometeo* junto a sus ficheros creados. Es necesario que los ficheros permanezcan aquí ya que se leen de esta carpeta. En caso de que se quiera cambiar su ubicación, sería necesario modificar la siguiente instrucción del método *loader()*:

  ```
  snprintf(path, 21, "prometeo/prog%03d.elf", idFichero);
  ```

Por último, se ha modificado el **Makefile**, ya que al utilizar una librería matemática es necesario la opción *-lm*.

```
OBJS        = kernel.o
SOURCE        = kernel.c
HEADER        = definitions.h
OUT         = kernel.out
CC          = gcc
FLAGS         = -g -c -Wall
LFLAGS         = -lpthread -lm

all: $(OBJS)
        $(CC) -g $(OBJS) -o $(OUT) $(LFLAGS)


kernel.o: kernel.c
        $(CC) $(FLAGS) kernel.c

queue.o: queue.c
        $(CC) $(FLAGS) queue.c

thread.o: thread.c
        $(CC) $(FLAGS) thread.c

mensajes.o: mensajes.c
        $(CC) $(FLAGS) mensajes.c

clean:
        rm -f $(OBJS) $(OUT)
```

9. Diseño

Respecto al diseño, la idea ha sido la misma que la primera parte. Para realizar la gestión de memoria, y mejorar los fallos de la primera parte. En esta sección, se enumera las funciones que han sido modificadas y también las añadidas. Para no saturar el documento con código, se han introducido solo los fragmentos de código importantes. El código se dispone en el archivo ZIP con en un repositorio de Git Hub.

- *int main(int argc, char *argv[]):* para poder realizar la parametrización, se ha tenido que modificar. Se ha introducido un bucle, que recorre todas las opciones disponibles (solo recorre aquellas que se han introducido). Para ello, se dispone de los valores por defecto que se ha comentado en la anterior sección. También se ha añadido una llamada al nuevo método *display_header()*.

```
    {"help",       no_argument,       0,  'h' },
        {"clock",        required_argument,        0,        'c' },
        {"frencuencia",        required_argument,        0,        'f'},
        {"p",        required_argument,        0,        'p'},
        {"n",        required_argument,        0,        'n'},
        {"t",        required_argument,        0,        't'},
        {"m",        required_argument,        0,        'm'},
    {0,            0,                  0,   0  }
};

while ((opt = getopt_long(argc, argv,":h:c:f:p:n:m:",
                    long_options, &long_index )) != -1) {
  switch(opt) {
    case '?':
        printf ("Uso %s [OPTIONS]\n", argv[0]);
        printf ("  -c  --clock=NNN\t"
            "Frecuencia del clock [%d]\n", CLOCK_DEFAULT);
        printf ("  -t  --timer=NNN\t"
            "Frecuencia del timer [%d]\n", TIMER_DEFAULT);
                printf ("  -h, --help\t\t"
                "Ayuda\n");
                printf ("  -nCPU"
            "Numero de CPU [%d]\n", NUM_CPU);
        printf ("  -nC  --nCores=NNN\t"
            "Numero de cores/nucleos [%d]\n", NUM_CORE);
                printf ("  -nT  --nThreads=NNN\t"
            "Numero de threads/hilos [%d]\n", MAXTHREAD);
                printf ("  -m  --memoria=NNN\t"
            "Multiplicador de tamaño memoria fisifca (2^m), mínimo 8[%d]\n",
            MEMORY_SIZE_DEFAULT);
```

```
            printf ("Ejemplos:\n");
            printf ("  ./kernel -c100 -t100  -nCPU2 -C2 -nT4 -m8\n");
            printf ("  ./kernel -nprog -f60 -l1000 -p1\n");
            printf ("  ./kernel -nprog -f61 -l20   -p60\n");
            exit (2);
        case 'c':   /* -c or --clock */
                    p1.tid=idclock.tid;
                    p1.frec=atoi(optarg);
            break;
        case 'f':   /* -f or --frecuenciaTimer */
            p2.tid=idtimer.tid;
                    p2.frec=atoi(optarg);
            break;
            case 'p':   /* -p or --procesador */
                    NUM_CPU = atoi(optarg);
            break;
            case 'n':   /* -n or --nucleo */
                    NUM_CORE = atoi(optarg);
            break;
            case 'h':   /* -h or --hilo */
                    MAXTHREAD =atoi(optarg);
            break;
            case 'm':
                    printf("m");
                    if (atoi(optarg)<8)
                    {
                            mensaje_error("El multiplicador de la memoria tiene que ser
                            mayor que 8");
                    }else{
                            MEMORY_SIZE_DEFAULT = atoi(optarg);
                    }
                    break;
        default:
            mensaje_error("Argumento no valido");
                    break;
      }
    }
```

- *void display_header():* imprime todas las opciones que se han seleccionado.
- *void inicializar():* se ha modificado para que inicie las estructuras del kernel. Para ello, hace uso del *malloc()*. También se inicia el nuevo *mutex*: *mutexMemoria*.

```
arr_cpu = malloc(sizeof(struct cpu)*NUM_CPU);
for (int i = 0; i < NUM_CPU; i++)
{
        arr_cpu[i].arr_core = malloc(sizeof(struct cpu_core)*NUM_CORE);
        for (int j = 0; j < NUM_CORE; j++)
        {
                arr_cpu[i].arr_core[j].arr_th = malloc(sizeof(struct core_thread)*MAXTHREAD);
```

```
        }
}

marcosDisp = sizeMemoria / 256;
marcosMax = marcosDisp;
memoriaFisica = malloc(sizeof(long)*sizeMemoria);
```

- *void asignarPCB(struct PCB pPcb):* se ha añadido la llamada a *volcarRegistros()*, para que se vuelquen los registros del PCB en el hilo hardware.

```
while (k < MAXTHREAD && seguir)
{
        if(arr_cpu[i].arr_core[j].arr_th[k].is_process) { // Se mira si hay proceso
                k++;
        }else{
                printf("[SCHEDULER] Metido [%d] quantum [%d] en [%d][%d][%d]",
                pPcb.id,pPcb.quantum, i,j,k);
                arr_cpu[i].arr_core[j].arr_th[k].t_pcb = pPcb;
                seguir = false;
                arr_cpu[i].arr_core[j].arr_th[k].is_process = true;
                volcarRegistros(&arr_cpu[i].arr_core[j].arr_th[k]);
        }
}
```

- *void decrementarQuantumYEjecutar():* se han añadido las llamadas de *guardarRegistros()* y *ejecutarInstruccion()*. Por una parte, se guarda el estado del hilo hardware en el PCb si el *quantum* llega a 0, y por el otro, se ejecuta la instrucción.

```
if (arr_cpu[i].arr_core[j].arr_th[k].is_process)
{
        pthread_mutex_lock(&mutexPCB);
        if (arr_cpu[i].arr_core[j].arr_th[k].t_pcb.quantum>0)
        {
                if (arr_cpu[i].arr_core[j].arr_th[k].t_pcb.quantum==1)
                {
                        arr_cpu[i].arr_core[j].arr_th[k].t_pcb.quantum=0;
                        arr_cpu[i].arr_core[j].arr_th[k].is_process=false;
                        guardarRegistros(&arr_cpu[i].arr_core[j].arr_th[k]);
                        enqueue(queue0_ptr,arr_cpu[i].arr_core[j].arr_th[k].t_pcb);
                }
                ejecutarInstruccion(&arr_cpu[i].arr_core[j].arr_th[k]);
                arr_cpu[i].arr_core[j].arr_th[k].t_pcb.quantum--;
        }
        pthread_mutex_unlock(&mutexPCB);
}
```

- *void guardarRegistros(struct core_thread *ptrCoreT):* guarda los registros del hilo en el PCB.

```
void guardarRegistros(struct core_thread *ptrCoreT) {
        for (int i = 0; i < 16; i++)
```

```
        {
                ptrCoreT->t_pcb.pcb_status.arr_registr[i] = ptrCoreT->arr_registr[i];
                ptrCoreT->arr_registr[i] = 0;
        }
}
```

- *void volcarRegistros(struct core_thread *ptrCoreT):* guarda los registros del PCB en el hilo hardware.

```
void volcarRegistros(struct core_thread *ptrCoreT) {
        //pthread_mutex_lock(&mutexPCB);
        for (int i = 0; i < 16; i++)
        {
                ptrCoreT->arr_registr[i] = ptrCoreT->t_pcb.pcb_status.arr_registr[i];
        }
        //pthread_mutex_unlock(&mutexPCB);
}
```

- *void ejecutarInstruccion(struct core_thread *ptrCoreT):* ejecuta la instrucción del hilo que se le pasa por parámetro. Para ello, realiza las mascaras necesarios para extraer todos los componentes necesarios de la instrucción.
  - Instrucción: instrucción que tiene el hilo hardware, que se obtiene por el PC del PCB.
  - Código: Se aplica la mascara *0xF0000000* y se desplazan 28b a la instrucción.
  - Registro: Se aplica la mascara *0x0F000000* y se desplazan 24b a la instrucción para obtener el primer registro. No se obtienen los otros 2 registros en este momento ya que el código no siempre es 2.
  - Dirección absoluta. Se aplica la mascara *0x00FFFFFF* a la instrucción.
  - Offset: Se aplica la mascara *0x0000FF* a la dirección absoluta. De esta manera, será posible moverse por los marcos.
  - Dirección virtual: Se realiza la mascara *0xFFFF00* y de desplazan 8b para obtener la dirección virtual.

```
long instruccion = ptrCoreT->t_pcb.pcb_status.PC;
long codigo = (instruccion & 0xF0000000) >> 28;
long registro = (instruccion & 0x0F000000) >> 24;
long direccionAbsoluta = (instruccion & 0x00FFFFFF);
long offset = (direccionAbsoluta & 0x0000FF);
long direccionVirtual = (direccionAbsoluta & 0xFFFF00) >> 8;
```

Después, dispone de una estructura condicional dependiendo del código de la instrucción.

  - 0 ó 1: en primer lugar, comprueba que la dirección virtual obtenida de la instrucción coincide con el TLB. Si es diferente, se actualiza el TLB. Si no lo es, se obtiene la dirección física. Con esta dirección física, se pueden realizar las operaciones de *ld* y *st*, accediendo a la memoria haciendo uso de la dirección física + offset.
  - 2: se obtienen los 2 registros restantes con las mascaras pertinentes, y se actualiza el registro.
  - 15: se cambia el *boolean* del PCb a False, para que borrar el PCB del hilo. Después, se limpian los marcos utilizados por el PCB, con el método *limpiarMarcos()*.

```
if (codigo==0 || codigo==1)
{
        long direccionFisicaAux;
        if (ptrCoreT->t_pcb.pcb_status.TLB.virtual != direccionVirtual) // Actualizar TLB o no
        {
                direccionFisicaAux = ptrCoreT->t_pcb.mm.pgb[direccionVirtual];
                ptrCoreT->t_pcb.pcb_status.TLB.virtual = direccionVirtual;
                ptrCoreT->t_pcb.pcb_status.TLB.fisica =
                ptrCoreT->t_pcb.mm.pgb[direccionVirtual];
        }else{
                direccionFisicaAux = ptrCoreT->t_pcb.pcb_status.TLB.fisica;
        }
        long direccionFisica = (direccionFisicaAux << 8) + offset;

        if (codigo==0)        // ld
        {
                pthread_mutex_lock(&mutexMemoria);
                ptrCoreT->arr_registr[registro] = memoriaFisica[direccionFisica];
                pthread_mutex_unlock(&mutexMemoria);

        }else{        // st
                pthread_mutex_lock(&mutexMemoria);
                memoriaFisica[direccionFisica] = ptrCoreT->arr_registr[registro];
                pthread_mutex_unlock(&mutexMemoria);
        }

}else if (codigo==2)        // add
{
        long registro1 = instruccion & (0x00F00000) >> 20;
        long registro2 = instruccion & (0x000F0000) >> 16;
        ptrCoreT->arr_registr[registro] = ptrCoreT->arr_registr[registro1] +
        ptrCoreT->arr_registr[registro2];
}else if (codigo==15)        // exit
{
        ptrCoreT->is_process=false;
        limpiarMarcos(&ptrCoreT->t_pcb);
}else{
        mensaje_error("Codigo de instruccion incorrecto");
}
```

- *void limpiarMarcos(struct PCB *ptrPCB)():* limpia los marcos, con el PGB del PCB obtenido como parámetro. Para ello, cambia el valor del rango de valores de la memoria física a ”0”.

```
void limpiarMarcos(struct PCB *ptrPCB) {
        pthread_mutex_lock(&mutexPCB);
        for (long i = 0; i < sizeof(ptrPCB->mm.pgb); i++)
        {
                long marco = ptrPCB->mm.pgb[i];
                for (long j = 0; j < 256; i++)
                {
```

```
                        pthread_mutex_lock(&mutexMemoria);
                        memoriaFisica[marco+j] = 0;
                        pthread_mutex_unlock(&mutexMemoria);
                }
                marcosDisp++;
        }
        pthread_mutex_unlock(&mutexPCB);
}
```

- *void *loader(void *arg):* el loader es un método muy extenso, ya que es necesario releer varias veces las mismas lineas del fichero.
  - Se abre el fichero con el identificador de *idFichero*. Por defecto, se empieza por el *idFichero=0*.

    ```
    snprintf(path, 21, "prometeo/prog%03d.elf", idFichero);
                    FILE *fichero = fopen(path, "r");
                    int nLineas = 2;
                    if (fichero == NULL)
                    {
                            break;
                    }else{
    ```

  - Se obtiene el número de lineas que tiene el fichero.

    ```
    while (fgets(buffer, 120, fichero) != NULL) // Obtener numero de lineas
                            {
                                    nLineas++;
                            }
    ```

  - Se obtiene el número de marcos, teniendo en cuenta de que cada bloque tiene 64 instrucciones.

    ```
    int nMarcos;
                            if (nLineas % 64 == 0) // Obtener marcos necesarios
                            {
                                    nMarcos = nLineas / 64;
                            }
                            else
                            {
                                    nMarcos = 1 + nLineas / 64;
                            }
    ```

  - Si el número de marcos es menor al número de marcos disponibles, se realiza la lectura del fichero. En este proceso, se crea un PCB, con todas sus estructuras. Además, se asignan los marcos necesarios.
  - Si el número de marcos es mayor al número máximo de marcos, se desprecia ese por pantalla y se notifica al usuario.
- *void printPCB(struct PCB* ptrPCB)():* imprime el PCB para que se visible en tiempo de ejecución.

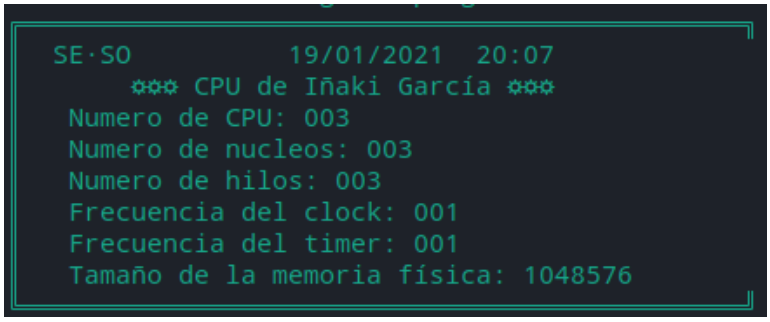
9.1 Posibles mejoras

- Sería mejor utilizar todo con punteros.

- Para seleccionar los ficheros generados por *prometeo*, no es muy seguro. Con el diseño implementado, solo coge aquellos ficheros que empiezan por el 0, y no adquiere aquellos que estén desordenados.
- Utilizar las señales para notificar de ciertas notificaciones.

10. Ejemplos de ejecución:

En primer lugar, se informa al usuario de las opciones que ha realizado, y si los valores por defecto de los que no ha seleccionado.



Después, se imprime los identificadores de los hilos creados.

```
[New Thread 0x7ffff6c51640 (LWP 56909)]
[New Thread 0x7ffff6450640 (LWP 56910)]
[New Thread 0x7fffeffff640 (LWP 56911)]
```

Durante la ejecución del código, el LOADER informa de los PCB que ha creado, y el SCHEDULER informa cuales ha metido.

```
[LOADER]Introducido PCB:
    ID: 061
    Prioridad: 3
    Quantum: 087
    IR: 0
    PC: 16000048
    TLB
         Virtual:     0 | Fisica:     0
    Code: 0
    Data: 24
    PGB: 0
[SCHEDULER] Metido [61] quantum [87] en [0][0][0]
```

Por otra parte, cuando ha transcurrido el tiempo del timer, se ejecutan las instrucciones, que se informa de su estructura. Además, se informa del estado de los registros previos y posteriores.

```
[INTRUCCION] 16000048   > Registros previos: [ 0 | 0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |
   > Codigo: 1
```

```
   > Registro: 6
   > Direccion Absoluta: 48
   > Direccion Virtual: 0
   > Offset: 48
   > Registros: [ 0 | 0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |
```

Para finalizar, se imprimen las estructuras del kernel, junto al estado de las colas. De esta manera, se sabe en qué estado de la ejecución ha terminado el programa. Por cada PCB, se imprime: id, prioridad y *quantum*.

```
Estructura
Número de CPU [3]
Número de CORE [3]
Número de thread [3]
CPU [0]
         CORE [0]
                  T [0]            PCB [61]|[3]|[86]
                  T [1]            PCB [62]|[1]|[15]
                  T [2]            PCB [63]|[1]|[35]
         CORE [1]
                  T [0]            PCB [64]|[2]|[93]
                  T [1]            PCB [65]|[1]|[22]
                  T [2]            PCB [66]|[2]|[28]
         CORE [2]
                  T [0]            PCB [67]|[2]|[60]
                  T [1]            PCB [68]|[3]|[27]
                  T [2]            PCB [0]|[0]|[0]
CPU [1]
         CORE [0]
                  T [0]            PCB [0]|[0]|[0]
                  T [1]            PCB [0]|[0]|[0]
                  T [2]            PCB [0]|[0]|[0]
         CORE [1]
                  T [0]            PCB [0]|[0]|[0]
                  T [1]            PCB [0]|[0]|[0]
                  T [2]            PCB [0]|[0]|[0]
         CORE [2]
                  T [0]            PCB [0]|[0]|[0]
                  T [1]            PCB [0]|[0]|[0]
                  T [2]            PCB [0]|[0]|[0]
CPU [2]
         CORE [0]
                  T [0]            PCB [0]|[0]|[0]
                  T [1]            PCB [0]|[0]|[0]
                  T [2]            PCB [0]|[0]|[0]
         CORE [1]
                  T [0]            PCB [0]|[0]|[0]
                  T [1]            PCB [0]|[0]|[0]
                  T [2]            PCB [0]|[0]|[0]
         CORE [2]
```

```
                T [0]           PCB [0]|[0]|[0]
                T [1]           PCB [0]|[0]|[0]
                T [2]           PCB [0]|[0]|[0]
Queue[ 100 ]
Queue[  ]
Queue[  ]
Queue[  ]
```

Hay 3 casos en los que el programa puede terminar:

- El programa principal se "despierta" después de superar el tiempo de espera (notificando al usuario).
- Todos los PCB han sido cargados en memoria y se han ejecutado todas las instrucciones, pero el tiempo de vida no se ha terminado. En ese caso, se espera a que termine.
- No se ha podido introducir ningún PCB, así que se espera a que el tiempo de vida termine.

11. Conclusiones

Para finalizar este documento, me gustaría expresar las conclusiones del proyecto. En primer lugar, el proyecto me ha parecido muy interesante. Desde que entré en la universidad, mi interés por el mundo de *hardware* ha ido creciendo, a pesar de ser de la rama *Software*. Me parece fascinante la estructura de los procesadores internamente, y me ha parecido interesante el uso de la memoria virtual y física.

Por otra parte, me ha frustrado el resultado del programa. A pesar de revisar el código en una gran cantidad de veces, no he sido capaz de solucionar los problemas detallados anteriormente. A pesar de ello,me ha ayudado a mejorar mis cualidades del lenguaje de C y el domino con soltura su *Debugger* (gdb). Respecto al código, creo que es un código óptimo. Por una parte, el hecho de utilizar un *scheduler híbrido*, me ha ayudado a buscar diferentes alternativas de *scheduler*. Gracias a eso, he aprendido un poco más como funcionan los procesadores de ahora (como puede ser el uso de la cola de prioridades). Por otra parte, se ha hecho uso de todas las estructuras solicitadas de los PCB (TLB, IR, PC...), por lo que la ejecución no se diferencia en demasía con una simulación real.