Simulador de un kernel

Sistemas Operativos

Grado en Ingeniería Informática

Facultad de Informática UPV/EHU

Fecha: junio de 2025

Autor: Diego Idígoras

1. Contexto

En este proyecto se debe crear una simulación de un kernel desde cero, para ello el proyecto se divide en 3 partes donde se trabajarán distintas partes del mismo:

1. **Arquitectura del Sistema**: crear las bases del kernel, sus estructuras, CPU's, cores, hilos hardware, timers, reloj, …
2. **Scheduler o planificador**: definir políticas para aumentar la eficiencia del sistema, gestionando la entrada y salida de los distintos threads al procesador
3. **Gestor de memoria**: crear una memoria física, cargar programas desde ficheros y ejecutarlos, traducción de direcciones virtuales a físicas y viceversa, …

El código está accesible desde mi repositorio de GitHub mediante este enlace:
https://github.com/DiegoIdig/Proyecto_SO

2. Parte 1: Arquitectura del Sistema

En esta primera parte el objetivo principal era que el timer y el clock se comunicasen entre ellos de forma indefinida, simulando un sistema real. Además, se debe ejecutar una función según la frecuencia recibida por parámetro al ejecutar el programa. El timer, llevará la cuenta de los ciclos del reloj, y cuando se alcanza dicha frecuencia la función correspondiente se activa, en este caso el scheduler y el processGenerator. En esta fase del proyecto los archivos de my_scheculer_dispatcher y my_processGenerator no tenían una implementación avanzada, simplemente mostraban un mensaje por pantalla avisando de su activación. Para el caso de my_PCB es muy similar, contiene la estructura base del pcb con su identificador, tiempo de vida generado de forma aleatoria y para su uso en la segunda parte, el quantum y ciclos ejecutados.

Un ejemplo para ejecutar esta versión del programa: .\main.exe 3 4 16 4 10
Generando así un sistema con 3 procesadores, 4 cores por procesador, 16 threads por core, y una frecuencia de activación de 4 ciclos para el scheduler y 10 para el processGenerator.

2.1 Comunicación entre el clock y el timer

Para esto me he basado principalmente en los contenidos vistos en clase y en el archivo "Semáforos" disponible en eGela, donde se muestra un ejemplo del uso de *mutex* con condiciones (*cond* y *cond2*) y variables compartidas (*done*) para realizar la sincronización.

El funcionamiento es el siguiente:

```
while(machine->global_cycles < 100)
{
  pthread_mutex_lock(&mutex);

  while(done < totalTimers) pthread_cond_wait(&cond, &mutex);
  ...
  done=0;
```

```
  pthread_cond_broadcast(&cond2);
  pthread_mutex_unlock(&mutex);
  }
  stop_system = 1;
}
```

```
while(!stop_system)
{
  pthread_mutex_lock(&mutex);
  done++;
  ...
  pthread_cond_signal(&cond);
  pthread_cond_wait(&cond2, &mutex);

  pthread_mutex_unlock(&mutex);
}
```

De esa forma se consigue comunicar de forma sincronizada el clock y el timer, haciendo avanzar el sistema ciclo a ciclo en base al reloj.

Para las pruebas establezco un límite de 100 ciclos en total, pero se puede dejar de forma indefinida modificando del while de ambos por while(1).

Por otra parte, utilizo dos estructuras para pasar los parámetros a la función *pthread_create* desde el main, ya que no he encontrado otra forma de hacerlo mediante un vector de argumentos o similar, porque cada argumento puede ser de un tipo de dato distinto. De esta forma creo *clock_params* y *timer_params*, que contienen el total de timers con los que debe comunicarse el reloj y la estructura de la máquina para hacerla avanzar ciclo a ciclo. Para el caso del timer, contiene la frecuencia de activación de la función y la función que debe activar.

2.2 Estructura my_machine

Se encarga de guardar todos los CPU's, cores y threads, además de los ciclos totales del sistema, cantidad de cores por CPU, cantidad de threads por core, etc. Está definida en el archivo my_clocktimer.h, donde establezco una estructura para parte del sistema (CPU, core y thread), agrupados mediante un vector. Cada uno contiene un id distinto que se asignan al crear la máquina con la función *create_my_machine*. Esta función simplemente reserva memoria para la máquina y crea vectores en función de la cantidad de CPU's, cores y threads, para los que también reserva memoria e inicializa con los valores recibidos al ejecutar el programa desde línea de comandos.

2.3 Colas de procesos

La idea inicial era crear un Round Robin (RR), aunque lo acabé modificando un poco para facilitar la implementación y gestión de las colas al añadir y eliminar procesos. Hay una cola

por procesador, que están agrupadas en la variable global llamada *global_queues*, que es un vector de punteros o matriz, según se quiera ver. Cada cola tiene una estructura (*my_processQueue*) que contiene un apuntador al primer y último elemento, además del total de procesos, número máximo de procesos que puede haber de forma simultánea en la cola, y la id de la CPU a la que pertenece la cola. Los elementos de la cola están formados por nodos que contienen el PCB del proceso y dos apuntadores, al anterior y al siguiente elemento, formando así una lista doblemente enlazada.

Diseñé 5 funciones aunque en la segunda fase del proyecto tuve que añadir alguna más. Sus nombres describen de forma sencilla que hace cada una de ellas, por lo que no entraré muy en detalle. Si que debo mencionar que, al tratarse de un RR los nuevos procesos se añaden al final de la cola, y los procesos en ejecución son los primeros de la cola que serán los que se eliminen cuando finalicen.

- create_my_processQueue
- add_process
- search_Process
- remove_Process
- print_queue

3. Parte 2: Planificador

En esta segunda parte las mayores modificaciones las he hecho en los archivos *my_scheduler_dispatcher* y *my_processGenerator* que son los componentes principales para simular los procesos.

Las políticas a seguir para la ejecución y reparto de tiempo de cómputo siguen la línea de RR, asignando un quantum fijo y rotando los procesos de la cola para repartir el tiempo de cómputo entre todos. Los procesos que se crean se asignan a la CPU que tenga la menor cantidad de procesos en la cola. Si se quieren añadir nuevos procesos y las colas están llenas, los nuevos procesos se descartan y se eliminan.

Para lograr esa rotación de procesos he tenido que añadir otra función en *my_processQueue*, donde simplemente cambio los punteros *first* y *last* de la cola, para indicar que el primer elemento ahora es el último de la cola.

Para simular el avance de la máquina simplemente aumenta en 1 la cantidad de ciclos ejecutados del primer proceso de la cola y el quantum que ha consumido cada vez que se genera un ciclo del clock. Estos cambios en los ciclos del pcb los hace el clock directamente.

3.1 Scheduler

Para el scheduler he definido una función que active todos los schedulers para cada cola, de tal forma que se recorre la variable global *global_queues* activando el scheduler para cada una de ellas. Una vez se llama al scheduler para gestionar una cola de procesos de una CPU concreta, se toma una decisión en función de 4 casos:

1. **Proceso finaliza**: si un proceso ha ejecutado la misma o mayor cantidad de ciclos que su lifetime asignado, se imprime su información y se elimina de la cola, estableciendo el siguiente proceso como el primero de la cola y el que entra en ejecución.
2. **Proceso agota el quantum**: simplemente se rota la cola para que el siguiente proceso de la cola entre en ejecución y se restablece a cero el *taken_quantum* para que cuando se vuelva a ejecutar este proceso tenga de nuevo la cuota de ejecución completa.
3. **Proceso no agota el quantum**: el proceso se sigue ejecutando hasta que se active el scheduler de nuevo y se vuelvan a barajar los posibles casos.
4. **Solo existe 1 proceso en la cola**: no se toman medidas, el proceso sigue ejecutándose.

Cada vez que se llama al scheduler se imprime la cola, para ver como van aumentando los tiempos de ejecución (*executed_cycles*), cuota consumida (*taken_quantum*), rotación de los procesos en la cola, etc.

Respecto al quantum me he decantado por asignarlo de forma constante para todos los procesos con un valor de 10 ciclos, está definido en el archivo *my_processGenerator.h*

Por otra parte, barajé la posibilidad de ejecutar todos los scheduler de cada procesador a la vez, para intentar optimizar un poco más el sistema evitando hacerlo de forma lineal. De ahí la función de *global_scheduler* que está comentada en el código. Pero tuve problemas para desarrollarlo ya que el programa finalizaba de repente y desconocía la razón. Para esta idea se utilizaba también la función *scheduler_wrapper*.

3.2 Generador de procesos

Cada vez que se llama a esta función desde el timer se crea un PCB pasándo por parámetro un quantum, que en este caso es fijo por la macro QUANTUM definida. Una vez creado el proceso, se consultan las colas de procesos de todos los procesadores y se guarda el índice de la cola con menor cantidad de procesos recorriendo *global_queues*. Por último, se muestra la información del PCB recién generado y se añade a la cola correspondiente.

4. Parte 3: Gestor de Memoria

Esta parte no he podido completarla por lo que no es funcional, sin embargo he desarrollado la fase 1 de la parte 3 del proyecto y un poco más. Hay varias discrepancias en el código de esta parte porque no he podido finalizarlo, las iré explicando en el informe.

Al ser una memoria direccionable por 24 bits y palabras de 4 bytes, el tamaño total de la memoria es 4 * 2^24 = 67 MB aproximadamente. En mi caso me he decantado por un tamaño de página de 4096 o 4KB y al kernel le he asignado 4MB de memoria, quedando el resto para el usuario. En el pcb se añade la estructura mm que contiene tres direcciones de memoria, dos virtuales que hacen referencia al segmento .text y .data del archivo .elf que carga el loader, y el tercero es el pgb que es la dirección base de la tabla de páginas del kernel asociado al pcb.

4.1 Physical memory

En este par de archivos (*.c* y *.h*) he definido la estructura básica de la memoria física y algunas de sus funciones para su gestión.

Hay muchas macros definidas en el header que no se llegan a utilizar en las funciones como tal, pero las he definido por aclarar los distintos parámetros y valores de la memoria y por si me hacían falta durante el desarrollo. Algunas de las macros son:

- **NUM_PAGES**: número de páginas total de la memoria física.
- **PAGE_NUM_WORDS**: número de palabras por página.
- **KERNEL_PAGES**: número de páginas de la zona de memoria del kernel.
- **KERNEL_WORDS**: número de palabras de la zona de memoria del kernel.

A su vez he definido tres variables globales que son:

- **physical_memory**: la memoria física como tal, es un array de tamaño PHYSICAL_MEMORY_SIZE inicializado a ceros.
- **pages_bitmap**: array que indica las páginas ocupadas de la memoria (0=libre, 1=ocupado). Las del kernel quedan marcadas como ocupadas.
- **kernel_pageTable_ind**: es el índice que indica la siguiente zona disponible para hacer una tabla de páginas de un proceso que se vaya a crear.

Relacionado con la última está la función *create_pageTable* que recibe por parámetro la cantidad de entradas que debe tener la tabla de páginas, que como mínimo serán 2, una para el segmento text y otro para el data. El *kernel_pageTable_ind* se aumenta según el número de entradas añadidas. Aunque no esta implementación no es del todo correcta, ya que cada segmento puede tener varias páginas asignadas, por lo que es mejor guardar vectores en lugar de entradas sueltas.

Por otra parte está la función *allocate_page* que busca la primera página libre de la memoria, la reserva marcándola como ocupada en el bitmap y devuelve su índice. Esto lo utilizo en *load_segments* que leyendo de un buffer crea los segmentos correspondientes en memoria y escribe en ella los datos del buffer. Dichos datos corresponden a los segmentos de datos y texto/código que se obtienen de los archivos generados con prometheus. Se reciben la cantidad de palabras a copiar de ambos segmentos, se calcula el número de páginas de cada uno de ellos, ya que pueden ocupar más de una página cada segmento en caso de ser muy extensos. Después se van solicitando la cantidad de páginas necesarias y se van guardando su índices multiplicados por la cantidad de palabras por página (PAGE_NUM_WORDS) en los vectores *text_frames* y *data_frames*. Tras todos estos cálculos copio en la memoría física el contenido página a página y para la última de ellas es muy probable que no sea exactamente del tamaño de la página, por ello decremento la cantidad total de palabras (*words_to_copy*) a copiar. Los vectores de *text_frames* y *data_frames* son los que se deberían guardar en la tabla de páginas del kernel, pero no he podido terminar de implementarlo, de ahí la discrepancia entre ambas funciones. Debería crear un struct que devuelva ambos vectores en esta función y pasarla dicha estructura por parámetro a la función *create_pageTable*, esa era la idea. Tras crear la tabla de páginas en la zona de memoria del kernel, actualizo el pgb de la estructura mm del pcb recibido por parámetro.

4.2 Loader

Esta función es utilizada por *my_processGenerator*, consiste en cargar desde los archivos que genera prometheus en formato .elf toda la información necesaria. Primero leo las dos primera líneas del fichero con *fgets* y las desgloso mediante *sscanf* para obtener la etiqueta y la dirección virtual correspondiente, buscando un string y un valor hexadecimal respectivamente. De esta forma, sé que si la *label* es igual a la cadena de texto ".text" la cadena hexadecimal que obtengo es la dirección virtual de inicio del segmento .text y de igual forma para el segmento de datos. Después, calculo la cantidad de palabras de cada uno de los segmentos y obtengo la cantidad total de palabras con un bucle hasta leer todo el documento, el *fp* antes del bucle ya ha saltado las dos primera líneas con los *fgets* por lo que solo se van a leer instrucciones o datos. A continuación, creo el buffer que *load_segments* recibirá como parámetro y vuelvo al inicio del fichero para ir leyendo palabra a palabra, copiandolas a su vez en el buffer. Por si acaso miro si las líneas tienen el carácter de salto de línea "\n" y lo sustituyo por el de fin de cadena de carácteres para que cada lectura sea del tamaño exacto de la palabra, o sea 4 bytes (*WORD_SIZE*). Tras copiar todos los datos al buffer, actualizo los campos de la estructura mm del pcb con la dirección virtual del segmento de datos y texto. Por último, llamo a la función *load_segments* para que partiendo del pcb, del buffer con todos los datos y de la cantidad de palabras del segmento .text y .data pueda volcar los datos del proceso en la memoría física.

4.3 Ejecución de procesos

En este apartado no me ha quedado claro si debía utilizar heracles o si la herramienta era simplemente para visualizar el desensamblado de los archivos .elf en instrucciones en ensamblador y sus correspondientes datos. Me he decantado por que es simplemente una herramienta visual y que nosotros debíamos hacer la implementación de lectura de palabras, interpretación y ejecución de la instrucción con los registros correspondientes. Aun así, no he podido finalizar esta parte tampoco.

Como se dice en el enunciado, solo debemos gestionar 4 operaciones: LD, ST, ADD y EXIT. Los valores en hexadecimal de cada uno de ellos los he definido en el header de *my_clocktimer* como:

```
#define LOAD_VALUE 0x000000
#define STORE_VALUE 0x100000
#define ADD_VALUE 0x200000
#define EXIT_VALUE 0xF00000
```

La idea era ir leyendo los datos con la función *read_word* e ir interpretando los valores para determinar qué operación se debía ejecutar y con qué datos. En caso de que la operación fuera un store, utilizaría *write_word*. Como en las instrucciones las direcciones de los datos son relativas, o sea, virtuales, utilizaría la función *translate_vaddr_to_paddr* pasando dicha dirección relativa y el pcb como parámetros para obtener la real y así poder cargar la palabra correspondiente a dicho dato. De igual forma con las instrucciones, además de que hago una operación lógica con las macros definidas para obtener los valores de los bits que me interesan para conocer la instrucción, por ejemplo:

```
opcode = instruction & 0xF00000; //obetener el bit correspondiente a la operacion
```

5. Conclusiones y reflexiones

Con este proyecto hemos visto de forma práctica todas las partes esenciales y a su vez básicas que componen un kernel. Empezando desde la creación y carga de procesos (con sus respectivos PCBs y segmentos de memoria), lo que facilita la comprensión de cómo se gestionan y programan los procesos en un sistema real. Siguiendo con la asignación dinámica de procesos, en mi caso a las distintas CPU's y cores. Todo ello gracias a un scheduler que destaca la importancia de gestionar de forma adecuada las colas de procesos y la sincronización entre los distintos componentes. Por último, la gestión de la memoria, desde la creación de los segmentos y carga de los mismos en memoria, hasta la interpretación de los mismos para ejecutar los procesos ciclo a ciclo.

Me hubiera gustado terminar el proyecto en su totalidad pero no me ha sido posible por falta de tiempo y sobre todo mala organización. Aún así, todo ello me ha aportado una visión de cómo funciona la base de un sistema operativo, que al fin y al cabo es el kernel. A su vez, he aprendido mucho sobre el lenguaje de programación C, que era otro de los objetivos principales de esta práctica. Desde definir macros en los headers para evitar sobreescrituras de las funciones al compilar con #ifndef, hasta la importancia de llevar una programación organizada y modular, el uso de punteros en C, la creación de estructuras desde cero, la declaración y uso de variables globales y en memoria compartida, etc.

En resumen, considero que ha sido un buen proyecto del que he aprendido mucho y amplía mi perspectiva sobre el diseño y funcionamiento de un sistema operativo, resultando fundamental para mi formación en múltiples áreas de la ingeniería informática