Simulador de un Kernel

Sistemas Operativos

23/06/2023

Ian Fernandez Hermida

Índice

Introducción

En el marco de esta práctica, se ha desarrollado un simulador de kernel. Mediante la implementación de diversas funcionalidades, se ha buscado emular de manera precisa y detallada el comportamiento de una máquina física.

Una de las principales áreas de enfoque en la implementación del simulador ha sido la planificación de tareas. Se ha desarrollado un sistema de colas con políticas de planificación para el scheduler, lo que permite asignar eficientemente los recursos del sistema a los procesos en ejecución.

Se ha implementado tanto una memoria física como una memoria virtual para simular la interpretación de instrucciones y el almacenamiento de datos. Esto ha implicado diseñar y desarrollar algoritmos y estructuras de datos que permitan la administración eficiente de los recursos de memoria, incluyendo estrategias de paginación, segmentación y algoritmos de reemplazo de páginas.

Objetivos

La finalidad de esta práctica es tratar de replicar de forma efectiva el comportamiento de un sistema operativo real, para ello se deben tener en cuenta los siguientes objetivos.

1. Arquitectura del sistema: Diseñar una arquitectura que simule una máquina real, incluyendo, CPUs, cores, threads, el clock del sistema y timers.

2. Planificación: Implementar un sistema de colas con políticas de planificación para el scheduler, asegurando una ejecución adecuada de los procesos.

3. Gestión de memoria: Desarrollar una memoria física y virtual para simular instrucciones y datos. Esto implica la asignación y liberación eficiente de memoria física, así como la implementación de una tabla de páginas y traducción de direcciones virtuales a físicas para la gestión de memoria virtual.

Ficheros

La práctica está dividida en dos carpetas, puesto que el gestor de memoria modifica completamente el funcionamiento del simulador. Por lo tanto, en la carpeta *first_part* se encuentra el simulador que emplea el generador de procesos, mientras que en la de *second_part,* está la que implementa el gestor de memoria.

main.c: Contiene la función main del programa que llama a las funciones de inicialización, lanza los pthreads necesarios para el programa y la implementación del reloj que se encarga de mover la máquina.

init.c: Proporciona una interfaz al usuario e inicializa datos y estructuras necesarias para el funcionamiento del programa, como las especificaciones de la máquina.

process_generator.c: Contiene la implementación del temporizador encargado de generar los procesos junto a sus PCBs e introducirlos en las colas correspondientes de cada core.

process_queue.c: Funciones relacionadas con las operaciones de las colas, encolar, desencolar…

scheduler.c: Aquí se ubica el segundo temporizador. Este fichero tiene las funciones necesarias para poner en marcha tanto scheduler como dispatcher y simular la ejecución de los procesos.

loader.c: Es la sustitución del generador de procesos. Interpreta los datos de un programa almacenado en un fichero.

globals.h: Guarda las definiciones que puedan ser de propósito general para el programa, como librerías, variables globales…

structs.h: Almacena todas las estructuras de datos necesarias para el funcionamiento del simulador.

Desarrollo

Arquitectura del sistema

Para llevar a cabo el desarrollo de la arquitectura del sistema, se debía implementar un reloj y uno o varios temporizadores. El reloj es un simple componente hardware que emite pulsos a una cierta frecuencia, mientras que el temporizador es el componente software encargado de recibir esos pulsos y contarlos. Para sincronizar estos dos componentes es necesario implementar un mutex condicional con el que se garantiza la exclusión mutua y se consigue tener los datos perfectamente distribuidos. Con este método se logra sincronizar no uno, sino múltiples temporizadores junto al mismo reloj.

En este caso se han implementado tres funciones para simular el reloj y dos temporizadores. La función del *cpu_clock* produce los pulsos y las de *timer1* y *timer2* los consumen. Se hace uso de la librería pthreads y serán hilos simulados los que ejecuten cada una de estas funciones.

El *timer1* se encarga de generar procesos mediante la función *generate_process*, a una frecuencia variable indicada por el usuario. Por otro lado, el *timer2* es el encargado de llamar al scheduler cada cierta frecuencia también introducida por el usuario.

Para terminar con la arquitectura, han sido necesarias ciertas estructuras que se encuentran en *structs.h*. En principio solo han hecho falta la *pcb_t* para crear los procesos y almacenar la información de estos y la de *args_t* que es la estructura que se le pasa al reloj y a los temporizadores, con información que deben tener en común.

Planificador

Para el planificador, se pedía realizar el funcionamiento tanto del scheduler como del dispatcher. Para ello, es necesario realizar un sistema de colas que permita a estos dos componentes introducir y sacar procesos de la CPU, simulando de esta forma su ejecución.

Para el sistema de colas, se ha creado en el fichero *structs.h* la estructura *proces_queue*, que va dentro de cada core, por lo tanto, cada core de la máquina tiene una cola de procesos implementada. Para tratar de simplificar esta tarea, se ha creado una sola cola de procesos sin tener en cuenta las prioridades que pueda tener cada proceso, pese a que la idea inicial era usar una política multinivel.

Teniendo las colas creadas, el *process_generator*, se encarga de introducir los procesos creados en dichas colas, dando pie a que el scheduler pueda realizar la planificación prevista para repartir los recursos del sistema.

Como se ha mencionado, no se ha realizado una política multinivel, a pesar de ello, se ha decidido hacer uso de la política de arbitraje más justa para una sola cola, Round-Robin. Con esta política se asegura de que cada proceso pase un tiempo determinado en ejecución y no abarque todos los recursos del sistema para sí mismo hasta que llegue a su fin. Cabe mencionar, que todos los hilos del sistema ejecutan el proceso nulo cuando no tienen ningún proceso real en ejecución, de manera que la máquina está siempre en funcionamiento.

La implementación de la política de cola tiene lugar en el scheduler y se realiza de la siguiente manera. Cuando el scheduler es llamado, verifica cada hilo de la máquina para ver si los procesos no nulos que están ejecutando se han quedado sin *quantum*. De encontrar alguno, se llama al *dispatcher* para que evalúe su estado y determine si ha de volver a la cola de procesos, o en su lugar el proceso ha finalizado, en cuyo caso se da por terminado. En ambos casos, el *dispatcher* tratará de introducir un nuevo proceso en ejecución y en caso de que la cola se encuentre vacía, ejecutará el proceso nulo. Después de haber tratado cada hilo individualmente, por cada core, el scheduler inspecciona si existe algún proceso en la cola que pueda entrar a ejecución y llama al dispatcher para tratar de mandarlo a ejecución.

Para realizar una simulación más adecuada, se ha tratado de implementar un *pthread* que se encargue de la ejecución del *scheduler*, puesto que el único trabajo del *timer2*, debería ser contar pulsos y avisar al *scheduler* al alcanzar la frecuencia establecida. Esto se ha llevado a cabo de forma correcta, pero ha quedado comentado en el fichero *scheduler.c* debido a un imprevisto que se explica a continuación.

Anteriormente, los procesos que finalizaban, no lo hacían realmente. Es decir, si a un proceso se le acababa el tiempo de vida en medio de una ejecución, este no abandonaba la CPU hasta llamar nuevamente al scheduler, perdiendo ciclos de ejecución en dicho hilo. Se desconoce si la decisión tomada es acertada o no, pero el cambio realizado, comprueba en cada ciclo de reloj, junto con la reducción de *quantum* y tiempo de vida, si cualquier proceso ha finalizado. De ser así, se crea una interrupción llamando al scheduler en ese mismo instante y se reinicia la frecuencia con la que este es llamado. (Esta implementación ha llevado a una pérdida de tiempo innecesaria).

Gestor de memoria

Para la gestión de memoria, se debían implementar nuevas estructuras para recrear la correcta simulación de la memoria física, memoria virtual, sistema de páginas y sus correspondientes traducciones de direcciones.

Con las especificaciones exigidas para este apartado, la estructura *physical_memory_t* representa la memoria física y *mem_layout_t* la distribución de esta misma. La estructura *mmu_t* es la representación de la unidad de gestión de memoria y *tlb_t* su correspondiente TLB que funciona como una memoria caché para cada hilo. Se ha creado una estructura tanto para las páginas como para la tabla de páginas y se ha añadido a la pcb una estructura *mm_t,* que contiene los punteros a las direcciones virtuales de *.text, .data* y a la tabla de páginas.

Dentro de la función loader se leen los ficheros de la carpeta *programs* que simulan programas y se interpretan, de forma que se obtiene la dirección de comienzo de las instrucciones del programa (.text) y la de los datos (.data). De la misma forma también se obtiene en un formato legible, las instrucciones que a ejecutar y los datos.

El resto de la gestión de memoria no se ha podido llevar a cabo, por lo que, a partir de este punto se describirán una serie de pasos que se deberían seguir para finalizar el simulador.

Una vez recogidos los datos, primeramente habría que mirar si hay espacio en la tabla de páginas para almacenar el programa. En caso de que si hubiera espacio disponible, se debería almacenar las páginas en la memoria física y crear la pcb donde se guardarán las direcciones virtuales de *.text* y *.data* y también la dirección de la tabla de páginas en la sección del kernel.

Durante el cambio de contexto se deberá guardar la información de ese proceso en la tabla de páginas para que el siguiente en ejecutarse pueda cargar esa información.

Conclusión

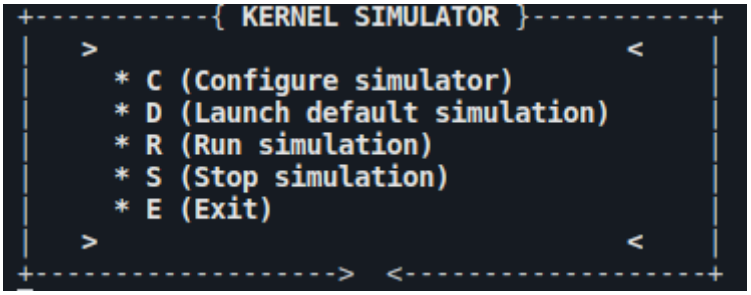
En resumen, el sistema cuenta con una arquitectura que incluye un reloj, temporizadores y sincronización. El planificador utiliza una cola de procesos y aplica la política de Round-Robin. La gestión de memoria requiere la implementación de una memoria física y virtual. Pese a que algunos aspectos no han sido completamente implementados, se ha tratado de realizar una correcta planificación para su desarrollo.

Manual de usuario

Primeramente ubicarse en la parte de la práctica que se desee ejecutar. Cada parte tiene en su interior un *makefile* para poder compilar el código que genera un ejecutable llamado *kernel_simulator*.

```
# make
# ./kernel_simulator
```

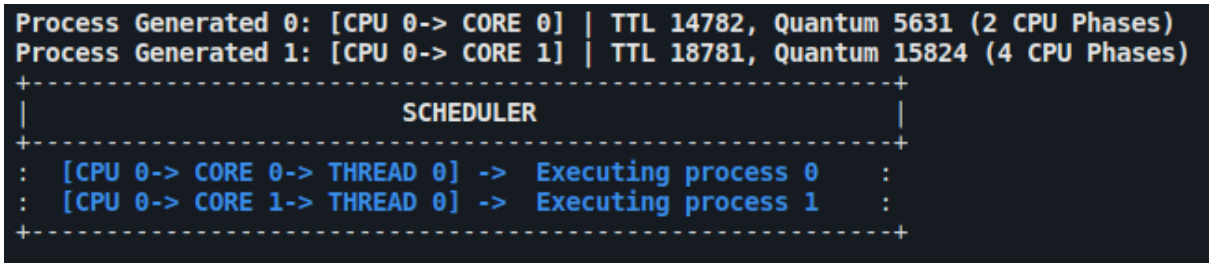
Cuando se ejecuta el programa ofrece una interfaz gráfica para interactuar con él de una manera básica.



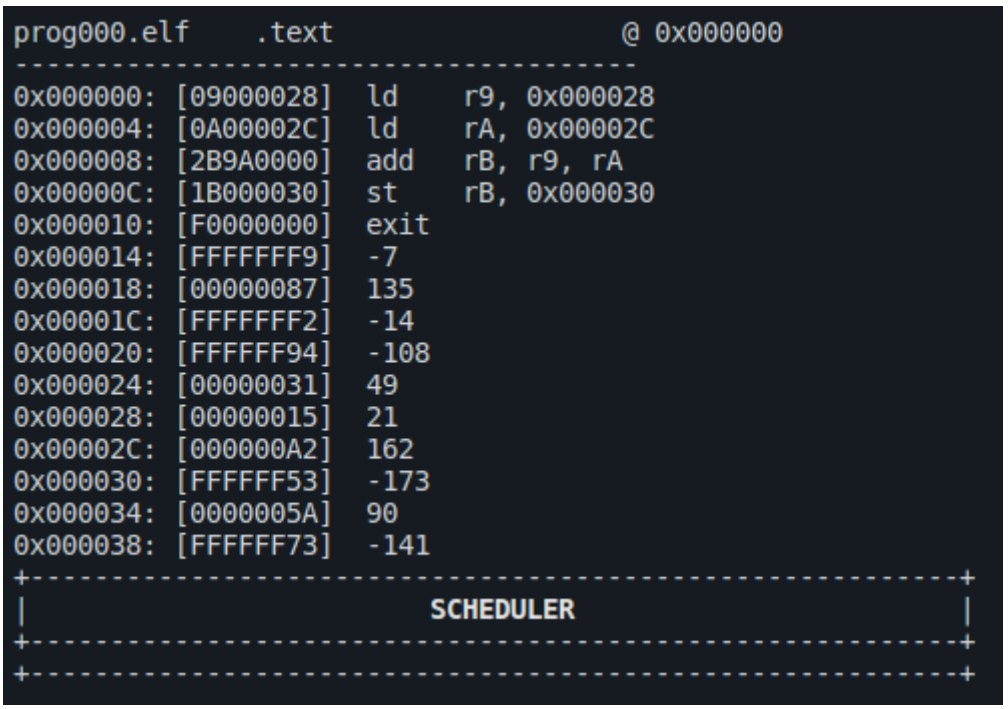
Interfaz de usuario.

Cabe mencionar que si se quiere realizar una simulación personalizada “R”, se deben configurar primeramente los parámetros del simulador “C”. En caso de que no sea así, se recomienda ejecutar con la configuración por defecto “D”. (La función de stop no está desarrollada)

Cuando se lanza lo que se visualiza por pantalla es la generación de procesos y las llamadas al scheduler como se muestra a continuación:



Fase de ejecución de *first_part*.



Fase de ejecución de *second_part*.