Sistemas Operativos
Simulador de un Kernel

Universidad del País Vasco
Euskal Herriko Unibertsitatea

Ander Carrasco del Río

Índice:

Objetivo:

El objetivo de la práctica de la asignatura de Sistemas Operativos consta en desarrollar un programa multihilo (haciendo uso de la librería *pthread.h*) que simule el comportamiento de un kernel, utilizando el lenguaje de programación **C**. Además, debemos apoyarnos en mutex con variables de condición para la sincronización y comunicación entre hilos.

Para ello, el primer paso que hemos de completar es definir la arquitectura del sistema.

Arquitectura del Sistema:

Nuestro simulador tendrá 4 hilos principales: un reloj, un temporizador, un generador de procesos y un *scheduler/dispatcher*. Además, contará con una serie de estructuras de datos que ayudarán al correcto funcionamiento de los hilos y una serie de parámetros de entrada que definiré a continuación.

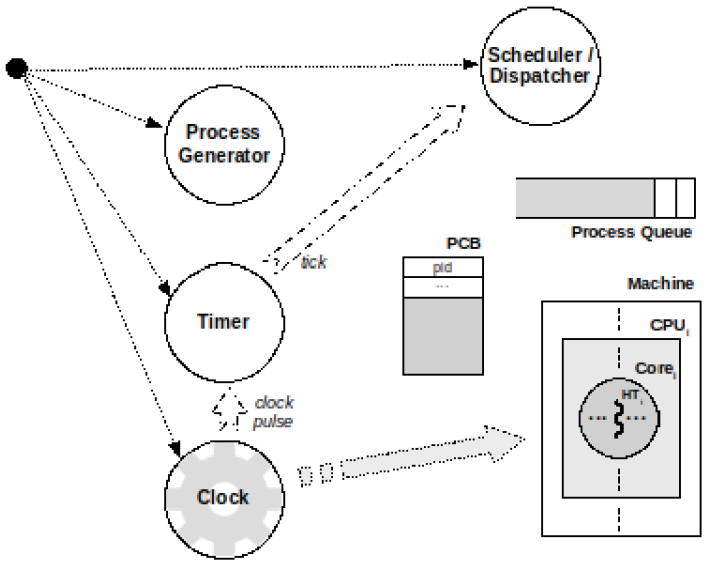

Figura 1: Arquitectura del sistema

Fuente: Documentación en eGela

- Estructuras de Datos:

Máquina: Contendrá el número de CPUs de la máquina y un *array* con dichos CPUs. Además, llevará la hora.

CPU: Contendrá el número de *cores* que tiene cada CPU y un *array* apuntando a cada uno de ellos.

Core: Le corresponde un único hilo *hardware*, de momento sólo indicará si éste está ocupado o no.

PCB (Process Control Block): Estructura que representa a un proceso. Contendrá su *pid*, tiempo de vida...

Cola de procesos: Cada nodo tendrá la información de un proceso (en una variable PCB) y estará apuntando al siguiente elemento de la cola (a una variable de Cola de Procesos, o a *NULL*).

- Parámetros de Entrada:

Frecuencia: En hercios, indicará cada cuanto tiempo deberá el temporizador producir una interrupción. Será un valor

Número de CPUs: El número de CPUs con los que trabajará la maquina.

Número de *cores*: El número de *cores* que tendrá cada CPU. Cada *core* contará con un hilo *hardware*.

- Hilos:

Reloj: El reloj se encargará de generar los ciclos que controlan el tiempo del sistema, envíando señales a la máquina (CPU, cores e hilos) para que

Temporizador: Se encargará de mandar una señal al resto de hilos de la máquina en función de la frecuencia indicada. Señales más lentas que el clock.

Generador de procesos: Generará procesos (PCBs) con una frecuencia aleatoria.

***Scheduler/Dispatcher*:** Llevará a cabo los cambios de contexto de los procesos.

Función **coreWork**, se inicializará una por cada core de la máquina. Esperará a una señal del scheduler, que se emitirá cuando se le haya asignado un proceso al core. Ésta, de momento, se limitará a contar los ticks del clock y reducir el tiempo restante del proceso.

Planificador (*Scheduler*):

Para llevar a cabo la implementación del planificador es importante tener claras las políticas que éste va a aplicar sobre los procesos. En mi caso, en cuanto a las políticas de planificación me he decantado por el ***Round Robin*** puesto que es considerada una política fácil de implementar, simple y que no puede provocar inanición.

La política de *scheduling Round Robin*, trata de organizar todos los procesos en una cola circular y asignar un segmento de tiempo fijo (*quantum*) a cada uno de ellos. En mi caso, el *quantum* corresponderá a ticks del clock, es decir, cuantos ticks del clock deben suceder antes de pasar al siguiente proceso.

Si un proceso no finaliza tras consumir todo el *quantum* se moverá al final de la cola y se liberará el core encargado del mismo pasando a ejecutar el próximo proceso disponible en la cola.

De esta manera, todos los procesos tendrán igualdad de oportunidades y no será necesario aplicar unas políticas de planificación expulsoras.

En esta fase, he configurado tanto el generador de procesos como el Scheduler/Dispatcher.

Generador de procesos:

Esperará a la señalización del temporizador.

Entonces generará un nuevo PCB, con un identificador (el primero será id=1), un tiempo de vida aleatorio e indicará el estado del proceso (0, no ejecutándose).

Creará un nuevo nodo para la cola de procesos, con la información del proceso (PCB) y apuntando a NULL. Añadirá éste nodo al final de dicha cola.

Scheduler/Dispatcher:

De igual manera, primero esperará a la señalización del temporizador.

Comprobará que haya procesos en la cola. Si los hay, comprobará el estado del primero de ellos, si no está en ejecución procederá.

Buscará un núcleo libre en la máquina. Cuando lo encuentré lo marcará como ocupado y calculará el índice del mismo. Entonces le enviará una señal

Apuntará al primer proceso de la cola, consultará el tiempo de vida que le queda, si el tiempo es menor que el *quantum*, elimnará el proceso de la cola, en caso contrario, lo mandará al final de ésta.

Mandará el proceso ha ejecutarse a una función auxiliar que agote el tiempo del *quantum* y que libere el proceso una vez haya pasado este tiempo. Ésta también "desbloqueará" el núcleo encargado del proceso, cuando finalice.

Los *mutex* controlarán que el acceso tanto a la cola de procesos por parte del generador y del *scheduler*, como al estado de los núcleos y al estado de los procesos se den de manera sincronizada.

Gestión de la Memoria:

Para este apartado, primero hemos tenido que construir el entorno para poder emular el sistema. Añadiendo nuevas estructuras de datos e hilos, de acuerdo a los siguientes dos esquemas:

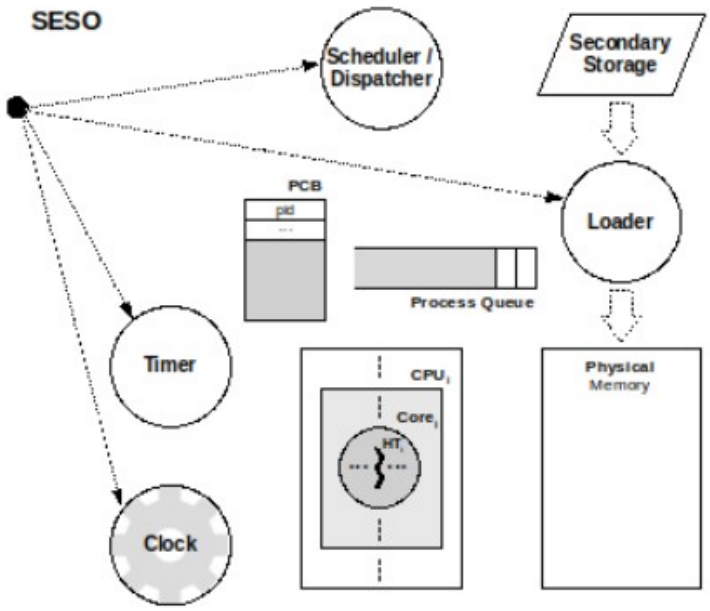

Figura 2: Nueva arquitectura del sistema

Fuente: Documentación de eGela

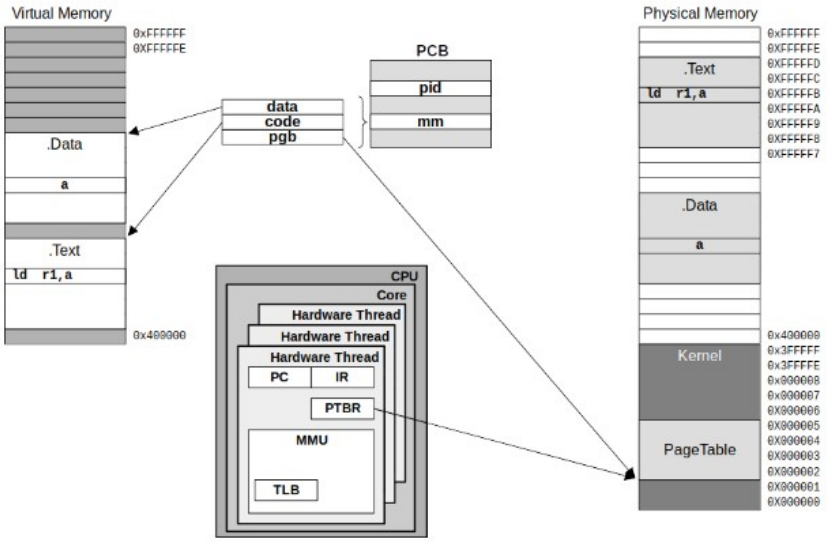

Figura 3: Esquema de la memoria

Fuente: Documentación de eGela

- Nuevas Estructuras de Datos:

Memoria física: Con una variable indicando el tamaño total de la memoria física, el numero de particiones y un *array* de punteros apuntando a las particiones de la memoria y un puntero a la propia memoria física "real" del tipo *uint8_t*.

Particion: El particionado será fijo. Por tanto, esta estructura contendrá variables indicando el tamaño de la partición, la dirección de inicio de la misma, el registro base y un bit indicando si está ocupada o no.

Tabla de páginas: Cada proceso contará con una. Formada por un *array* apuntando a todas las entradas de la tabla y el número total de páginas virtuales.

Entrada de la tabla de páginas: Contará con varios bits, para indicar si esa entrada la usa el proceso, si está presente en la memoria física, si ha sido accedida y/o si ha sido modificada. Además de la dirección física a la que señala.

mm: Estructura auxiliar para la estructura **PCB**. Contiene tres punteros; señalando la dirección virtual de comienzo del código, señalando la dirección virtual de comienzo de los datos y señalando la dirección física de la tabla de páginas correspondiente.

Hilo: Habrá un hilo por cada *core*. Tenfrá un ID, un *buffer* que guardará las direcciones físicas recientes (TLB) y un puntero a la tabla de páginas (PTBR).

TLB: Con un puntero a las entradas TLB y el número de las mismas que tiene.

entrada_TLB: Contendrá la dirección virtual y física correspondientes. Y un bit indicando si la entrada es válida o no.

Además de éstas, algunas de las estructuras definidas en la primera fase del proyecto han sido modificadas:

Core: Se le agrega un puntero a un hilo. Y un puntero a un PCB.

PCB: Se le agrega una variable de la estructura auxiliar **mm**. Además de una variable indicando el *quantum* restante, otra el PC y otra para los registros (R0..R16).

También agregué función auxiliar **MMU (*Memory Management Unit*)**: Esta se encargará de las traducciones de direcciones virtuales a físicas. Como parametros de entrada recibirá la dirección virtual en cuestión, un puntero al TLB, y otoro al PTBR. Con la dirección virtual, calculará la página y el desplazamiento, entonces buscará en la TLB la traducción, si no la encuentra la buscará usando el PTBR y actualizará la TLB.

- Nuevos Hilos:

Loader: Sustituye al generador de procesos. En vez de generar procesos aleatoriamente, leerá los programas a ejecutar desde un fichero de texto.

Primero, esperará hasta el *tick* del temporizador.

Abrirá el directorio *Secondary_Storage* y lo recorrerá, leyendo un fichero *.elf* cada vez que reciba un *tick* del temporizador.

Obtendrá la dirección virtual del segmento de código (junto a **.text**) y la dirección virtual del segmento de datos (junto a **.data**). Además, contará el número de líneas del programa y también las guardará.

Creará la tabla de páginas y la guardará en el espacio de *kernel* de la memoria. Inicializará todas las entradas de la tabla de páginas.

Creará un nuevo PCB: con un identificador, el estado (0, no procesando) y punteros a la dirección física de la tabla de páginas, a la dirección virtual del segmento de código y a la dirección virtual del segmento de datos.

Entonces, buscará una partición libre en la memoria física, donde copiará el programa. Actualizará las entradas de la tabla de páginas del proceso para que apunten a la dirección correspondiente en la memoria física.

Entonces, creará un nuevo nodo con el nuevo proceso y lo añadirá al final de la cola.

Ahora, si el ***scheduler/dispatcher*** encuentra un proceso con el estado "completado" lo retirará de la cola.

Según el enunciado, la memoria física tendrá un bus de direcciones de 24 bits y sus posiciones serán del tamaño de una palabra (4 Bytes). Por tanto, la memoria total será:

$$2^{24} \cdot 4B = 67{,}108{,}864B = \mathbf{64MiB}$$

Cada partición será de 512B, por tanto, habrá un total de 64MiB/512B = 131072 particiones.

De estos 64MiB, reservaré para el *kernel* 64KiB. El *kernel* no necesitará mucha memoria puesto que solo necesita guardar las tablas de páginas, de los procesos. Además, estos procesos son muy pequeños (no más de 50 instrucciones).

En los programas que genera *prometheus*, las líneas (de código o de datos) son de 4 bytes, es decir, ocupa exactamente lo mismo que una palabra de la memoria física. Por tanto, para la tabla de páginas he optado por páginas de 16 bytes.

La función **coreWork** se encargará de ejecutar estos programas. Cada tick del reloj hará que los cores ejecuten una instrucción del programa y el QUANTUM será de 5 ticks del reloj, es decir, cada 5 instrucciones volverá a la cola de procesos (si no termina antes). Se hará la traducción de la dirección de cada instrucción, y una vez obtenida se llevará a cabo el *load,store,add*...

Compilación y Ejecución:

La frecuencia del clocks será de 10k hercios, por tanto la frecuencia del *timer* (parámetro de entrada), ha de ser menor que 10k y un múltiplo de este (si no dará error). Además, el máximo de cores (CPUs * cores por CPU) con el que puede trabajar es de 64.

Compilación: ***gcc -Iinclude src/*.c -o simulador -pthread***.

El *scheduler* está programado para comenzar con algo de retardo respecto al loader, para que cuando empiece a trabajar ya haya procesos en la cola.

Ejecución (ejemplo): ***./simulador 10000 2 2***

He añadido muchos *prints* a lo largo del programa, para que cuando se ejecute muestre por pantalla cuando el loader añade un nuevo proceso, que proceso se está ejecutando, que core se está encargando de hacerlo, que instrucciones está llevando a cabo…

Jugando con los parámetros de entrada se puede poner a prueba la máquina.