Simulador del Kernel

Unai Sainz de la Maza Gamboa

6 de junio de 2021

Resumen

El objetivo de esta práctica es la realización de un simulador del Kernel, se ha realizado la arquitectura del sistema sobre el que se cimienta todo el Kernel, el planificador (o scheduler) para la gestión de los procesos y por último, se ha realizado el gestor de memoria.

Índice

1. Arquitectura del sistema

En esta sección se detallan todos los aspectos relevantes en lo que a la arquitectura del sistema respecta. Se ha optado por dividir en dos la sección. En la primera parte se muestra un esquema completo de la arquitectura para poder tener una referencia visual del sistema, y en la segunda se introducen todas las estructuras de datos necesarias para construir dicho sistema.

1.1. Esquema del sistema

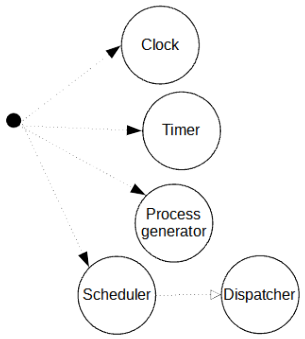

Figura 1: Esquema de los workers

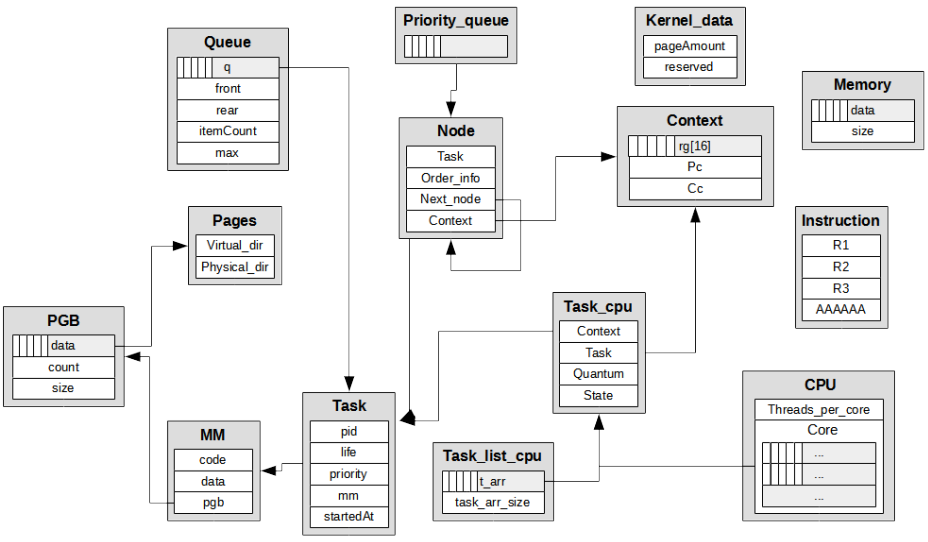

Figura 2: Esquema de las estructuras de datos

1.2. Estructuras de datos

El sistema consta de varias estructuras de datos para poder simular el Kernel, en esta sección tratamos de explicar la finalidad de cada una y las funciones asociadas a dichas estructuras.

Nota: todas las estructuras de datos están almacenadas en el fichero structs.h, acudir a él para más información sobre las estructuras.

1.2.1. CPU

Las estructuras relacionadas con la CPU, contienen toda la información necesaria para poder simularla, así como las funciones necesarias para dicho fin. Estas funciones se encuentran en el fichero *cpu.h*, consideramos esenciales las siguientes funciones:

- clock_phase_cpu(): esta función se encarga de que todos los procesos que están en la cpu y sigan vivos (o working), consuman su quantum por cada ciclo del clock.
- sched_cpu(Cpu *cpu): esta función busca en la cpu indicada los procesos que ya han finalizado, libera los recursos y devuelve dichos procesos en una lista.

1.2.2. Memoria

Estas estructuras contienen la información relativa al *Memory Mapping*, *PGB* y las *Páginas del PGB*.

- Memory Mapping (MM): contiene la información del memory mapping de un proceso, es decir, la dirección de memoria virtual tanto de code, como de data y el pgb.
- PGB: contiene cuantas páginas tiene el pgb, el tamaño de la lista reservada y la lista misma.
- Páginas del PGB (Pages): se utiliza para relacionar una dirección virtual (virtual_d), con una dirección física (physical_d).

Las funciones relacionadas con estás estructuras se pueden encontran en los ficheros *pgb.h* y *ram.h*, ambos fichero incluyen documentación explicativa sobre su cometido.

Nota: cuando se crea la memoria, se calcula cuantas páginas hay en el tamaño de memoria, se calcula el espacio que va a necesitar la tabla de páginas y se reserva: tamaño de memoria (seleccionado) + tamaño de la tabla de páginas.

1.2.3. Priority Queue

Esta estructura se utiliza como una lista ordenada en base a la prioridad (de ejecución) de cada nodo en la lista. Se han implementado las funciones básicas ligadas al funcionamiento de una lista, es decir: insertar nodo, retirar nodo, comprobar si la lista está vacía, etc.

A su vez, tenemos dos funciones que requieren de especial atención:

- priority_q_sort(): esta función se encarga de ordenar la lista en base al atributo *order_info*, dicho atributo nos determina la prioridad de cada nodo de la lista.
- priority_q_normalize(): es la encargada de normalizar los nices, se ejecuta justo antes de sacar los procesos de la lista. Su funcionamiento viene bien documentado en el fichero *prio_q.h*, pero resumiendo, en caso de que el primero de la lista tenga nice negativo, puede haber nices muy negativos, por lo tanto normalizamos.

1.2.4. Queue

Estructura que representa una lista *FIFO (First In, First Out)* para los procesos en espera de ser atendidos por el dispatcher. A diferencia de la *Priority Queue*, esta lista solo contiene las funciones básicas para su correcto funcionamiento, las funciones denominadas *CRUD (Create, Read, Update, Delete)*.

2. Workers

Los workers (o servidores), son los encargados de realizar el trabajo que hace falta para mantener en marcha el Kernel, diferenciamos entre dos tipos de workers: los que realizan (simulan) tareas externas al Kernel (*Clock* y *Process generator*), y los que realizan (simulan) las tareas internas del Kernel (*Timer* y *Scheduler*).

2.1. Clock

Es el encargado de simular el consumo de recursos (tiempo) hardware real que tendría la maquina. La función encargada de simular dicho consumo es *clock_phase_cpu()*, está incluida en el fichero *cpu.h*, donde se incluye la documentación sobre su funcionamiento.
Por otra parte, el clock debe generar interrupciones cada cierto número de ciclos, esto lo hace mediante el uso de semáforos.

2.2. Process generator

Este worker se encarga de simular las tareas que llevaría a cabo el sistema operativo, es decir, crear procesos. Estos procesos contienen código a ejecutar, dicho código viene dado por los archivos binarios (roms). A su vez, la función *new_program(Task t, char *file)*, que se encarga de leer el fichero indicado, también vuelca el contenido sobre la memoria asignada al proceso indicado.

2.3. Timer

Es el hilo de ejecución (worker) encargado de generar las interrupciones que activan el Scheduler.

2.4. Scheduler

El scheduler es el encargado de gestionar los procesos (*Task*) generados por el *Process generator*, es decir, calcular las cuotas de ejecución (cuanto tiempo le pertenece a cada proceso), insertar/retirar procesos de la CPU, etc.

Los pasos para poder llevar a cabo estas tareas son los siguientes:

- Extraer los procesos de la *queue* normal e insertar ordenados por prioridad en la *priority queue*. Esta tarea la lleva a cabo un hilo de ejecución llamado *Dispatcher*.
- Extraer los procesos que han terminado de la CPU (pueden haber terminado su tiempo asignado, pero no su trabajo al completo) e insertarlos en la *priority queue*. Es un paso importante para el *scheduler*, se pide a las CPUs mediante la función *sched_cpu(Cpu cpu)* la lista de procesos terminados, si alguno de los procesos no ha terminado su trabajo, lo introducimos en la *priority queue* con un valor de *order_info* (valor necesario para ordenar según prioridades) igual a la prioridad + el quantum.
- Asigna los procesos contenidos en la *priority queue* de manera equitativa en las CPUs, esto se hará mientras haya procesos en la lista o haya huecos libres en las CPUs.

2.5. Planificación

La planificación es esencial para distribuir los procesos a la espera de entrar a ejecutar en la CPU. Todos estos procesos se introducirán ordenados por prioridad + nice, a cuanto mayo es esta suma, más prioritario es el proceso. Hay casos en los que puede haber desajustes, por ejemplo, cuando un proceso entra en la lista tras haber estado en una CPU, se calula la suma en base a la prioridad + quantum. Pero en caso de que el *scheduler* se haya despertado tarde, y el proceso haya consumido más tiempo de CPU que el asignado, el quantum toma valor negativo.

Es necesario penalizar a dicho proceso (por consumir más tiempo que el debido) y premiar a los procesos en espera por no haber entrado en su turno a la CPU.

Cuando el *scheduler* extrae los procesos, puede haber procesos con suma de prioridad negativa, por lo tanto tendremos que normalizar todas las sumas de la lista. Para normalizar, necesitamos sumar a todos los nices el valor absoluto de el valor negativo máximo (el más negativo) de la lista, seguimos esta fórmula:

- $(nice_0 < 0) \rightarrow new\_nice_n = nice_n + (|nice_0| * 2)$

El tiempo asignado a cada proceso (quuantum) en la CPU viene dado por la siguiente fórmula:

- Para cada proceso p: $quantum_p = peso_p * 1000$
- Si $quantum_p > life_p \rightarrow quantum_p = life_p$

3. Intrucciones

3.1. Conjunto de instrucciones

Se han implementado las instrucciones dadas en el enunciado de la práctica, donde:

- C: código de operación.
- R: un registro general de los 16 disponibles (del 0 al 15).
- AAAAAA: dirección absoluta o virtual (24 bits).

Código	**Formato**	**Ensamblador**	**Comportamiento**
0 [0]	CRAAAAAA	ld rd, addr	rs = [addr]
1 [1]	CRAAAAAA	st rf, addr	addr = [rf]
2 [2]	CRRR - - - -	add rd, rf1, rf2	rd = [rf1] + [rf2]
3 [3]	CRRR - - - -	sub rd, rf1, rf2	rd = [rf1] - [rf2]
4 [4]	CRRR - - - -	mul rd, rf1, rf2	rd = [rf1] * [rf2]
5 [5]	CRRR - - - -	div rd, rf1, rf2	rd = [rf1] / [rf2]
6 [6]	CRRR - - - -	and rd, rf1, rf2	rd = [rf1] & [rf2]
7 [7]	CRRR - - - -	or rd, rf1, rf2	rd = [rf1] | [rf2]
8 [8]	CRR - - - - -	xor rd, rf1, rf2	rd = [rf1] $\wedge$ [rf2]
9 [9]	CRR - - - - -	mov rd, rf	rd = [rf]
10 [10]	CRR - - - - -	cmp rf1, rf2	cc = [rf1] - [rf2]
11 [11]	C - AAAAAA	b addr	branch to addr
12 [12]	C - AAAAAA	beq addr	branch to addr, if $cc == 0$
13 [13]	C - AAAAAA	bgt addr	branch to addr, if $cc > 0$
14 [14]	C - AAAAAA	blt addr	branch to addr, if $cc < 0$
15 [15]	C - - - - - - - -	exit	stop

Cuadro 1: Set de instrucciones

3.2. Funciones

Cada instrucción se ha implementado en una función, todas siguen una nomenclatura común del tipo: *inst_xx*, donde *xx ∈ (0, 1, 2, ..., 9, A, B, ... F)*. También se ha definido una función llamada *execute(Task t)*, que se encarga de obtener la instrucción que hay en *t.pc*, y crear la estructura correspondiente con sus campos, así como de llamar a dicha instrucción.

4. Modo de uso

Disponemos de varios parámetros de sistema que pueden ser seleccionados por el usuario: *Nota: todos los parámetros son de tipo entero.*

- -e [segundos], establece el tiempo en segundos que dura la ejecución (si *segundos* == 0, infinito).
- -c [cores], establece el numero de cores.
- -C [CPUs], establece el numero de cpus.
- -t [threads], establece el numero de threads.
- -q [tam], establece el tam maximo de la cola.
- -k [ciclos], establece la cantidad de ciclos para activar timer.
- -p [min_p], establece el minimo numero de procesos a generar.
- -P [max_], establece el maximo numero de procesos a generar.
- -s [min_s], establece el minimo tiempo a dormir.
- -S [max_s], establece el maximo tiempo a dormir.
- -r [tam], establece el tam maximo de la memoria.
- -v [0/1], establece si la memoria se muestra o no (1/0).
- -h, muestra la ayuda del sistema.

Junto al simulador, se ha implementado una herramienta que traduce a binario y setea los archivos correctamente. Dicha herramienta se llama *Binarize* y se encuentra en la carpeta *toolchain*, su modo de uso es el siguiente:

- Se colocan los archivos en la carpeta *files*, dentro de la carpeta *binarize*.
- Se ejecuta el script de bash incluído (*translate.sh*).
- Los archivos binarios traducidos estarán en la carpeta *binaries*, estos archivos están listos para ser colocados en la carpeta *programs* desde donde los va a ejecutar el simulador.