# Sistemas Operativos

---

Informe y Resumen

Simulador de un Kernel

**Facultad de Informática**
**Grado de Ingeniería Informática**

Josu Lorenzo Hernández

9 Mayo 2022

# Índice

# Introducción y Resumen

Todo sistema informático necesita de un administrador básico. Este administrador básico tiene que ser capaz de comunicarse con el hardware físico, y dar soporte seguro para poder ejecutar programas. En la ejecución de programas, los recursos son esenciales, y gracias al administrador, las aplicaciones pueden hacerlo de forma segura.

El mencionado administrador básico, se trata del Kernel. Es el componente más importante de cualquier sistema moderno, y es el encargado de la comunicación con el hardware. Los recursos proporcionados, en la mayoría de casos son el hardware; cpu, memoría física, almacenamiento no volátil, dispositivos, etc.

En este contexto, en la asignatura Sistemas Operativos (SO), se ha propuesto desarrollar un simulador de un Kernel. El alcance y envergadura del proyecto es limitado por razones obvias, pero se va a intentar realizar una aproximación. Para ello, en la asignatura se han explicado los diferentes modelos, mecanismos, políticas y principios en los que se basan los Kernels, haciendo especial hincapié en el de Linux.

# Objetivos y Estructura

El Kernel tiene que ser capaz de dar soporte al sistema operativo. Tiene que ser capaz de proporcionar una base segura y robusta para la ejecución de programas. Por lo tanto, el objetivo principal del proyecto es crear un Kernel seguro y fiable.

La estructura del sistema es clara; se tiene un reloj central, un Scheduler/Dispatcher, un Loader (cargador de programas), y los timers de ambos Scheduler y Loader. Cada uno de estos sistemas es un hilo (pthread) independiente.

***Reloj central (clock)***
Es el encargado de mover todo el sistema. Simula los ciclos del reloj, permitiendo al resto de componentes del sistema trabajar cuando les toque. La velocidad del reloj es configurable por el usuario. Simula la ejecución de instrucciones en cada hilo hardware, llevando a cabo todas las fases; fetch, decodificación, búsqueda de operandos, operar y dejar resultados.

***Timers (Scheduler y Process Generator)***
El timer es el responsable de llamar al Scheduler/Dispatcher y Loader (cargador de programas). Realmente no es un timer, sino dos, ya que el Scheduler y Loader se mueven por separado. Cada cierta cantidad de ciclos, generan ticks, es decir, interrupciones que señalan a ambos Scheduler y Loader.

***Scheduler***
El Scheduler es el administrador de colas y flujos de ejecución, es decir, los threads. Es el componente que planifica y organiza la ejecución de los programas, o mejor dicho, de los pcb's que simulan programas.

***Loader (cargador de programas)***
Por último, está el Loader. Este componente es el encargado de cargar los programas y ponerlos en circulación (encolar el PCB representante). Estos programas son leídos de ficheros .elf simplificados, en los que se hace uso de un conjunto de instrucciones reducido. Este sistema carga las instrucciones y datos del programa en memoria principal, y encola los respectivos PCB.

A su vez, también es necesario mencionar las estructuras de datos que se han creado. Haciendo una vista rápida, principalmente serían las listas con sus respectivos nodos, los pcb's ya mencionados, y la estructura de cpus, cores y threads englobados como "la máquina". Dentro de los threads, podemos encontrar las estructuras propias de un hilo hardware; registros generales (RGs), program counter (PC), instruction register (IR), translation lookaside buffer (TLB) y puntero a la tabla de páginas del proceso en ejecución (PTBR). A su vez, los PCB's contienen una estructura destinada a controlar la posición del código del programa en memoria principal; memory management (MM). Más adelante se explicarán en detalle.

Por último, me gustaría hablar brevemente de la estructura de los archivos del proyecto. Es una estructura estándar como la que se vería en cualquier otro proyecto; directorio "include" con los archivos de cabecera, directorio "src" con los archivos fuente, directorio "obj" con los archivos .o y .d de dependencias, y finalmente el archivo "makefile" para generar el ejecutable. Como apunte, el ejecutable se crea al mismo nivel que está el archivo makefile.

# Funcionamiento y Sincronización

## *Funcionamiento general*

El funcionamiento y política del sistema es simple; el Loader lee y carga los programas .elf en memoria y crea los pcb's que los representan. Después, el Scheduler los "pone en circulación", y cuándo su ejecución concluye éste mismo los saca de los threads, en vistas a cargar otro programa. La política que se ha elegido para las colas y la planificación de los pcb's ha sido FIFO, es decir, el primer pcb que llegue entra en ejecución siempre que sea posible. Una vez resumida brevemente la política y funcionamiento del sistema, voy a pasar a explicarlo más en detalle, empezando desde la planificación y cómo se ponen los pcb's en circulación, funcionamiento del Scheduler y Loader, ejecución de instrucciones/programas y sincronización de los timers, hasta las funciones auxiliares y el archivo makefile.

Para empezar, se va a explicar el flujo de ejecución general. Primero, el Loader lee los programas y los carga en memoria principal. Mientras lee los .elf's, lleva el seguimiento del comienzo y final del programa, junto con la configuración de la tabla de páginas que se ha hecho para ese programa en concreto. Con estos datos y el programa cargado en memoria, el Loader crea los pcb's y hace un wrapper metiéndolos en un nodo. Estos nodos son la unidad con la que va a trabajar la máquina. Con esto me refiero a que la máquina no hace referencias directas a los pcb's, sino que lo hace al wrapper nodo correspondiente. Esto se ha hecho en vistas a facilitar la implementación de las colas. Una vez creado el nodo, se encola en la "cola de preparados", la cual guarda los pcb's creados y que aún no se han podido poner en circulación. El siguiente código es de la función del hilo del Loader:

```
while(1) {
    if (loaderKill == 0) {
        buscarSiguienteElf();   // buscar siguiente program
        if (loaderKill == 0) {
            actualizarNombreProgALeer();    // actualizar e
            loadProgram(++pidActual);       // cargar el .e
            printf("[ Loader ] Programa cargado en memoria
```

Más adelante se explicará el mecanismo de carga de programas. Una vez encolado el nodo en "preparados", el Scheduler/Dispatcher cogerá los nodos disponibles que estén en la cola y los tratará de poner en ejecución. Es importante este matiz, tratará, ya que si no hay hilos de ejecución vacíos, no se podrá poner en ejecución el pcb por falta de recursos de la máquina. En este caso, se espera a que algún hilo de ejecución quede libre, y el pcb en cuestión se carga en el hilo en cuestión. Gracias a que la cola de preparados se gestiona con política FIFO, los pcb's que no se han podido poner en circulación estarán ordenados según lo viejos que son, lo que permitirá una planificación más justa que si no fuera FIFO.

```
if (cmpnode(maquina.cpus[i].cores[j].hilos[k].executing, nullNode) && maquina.cpus[i].cores[j].hilos[k].flag_ocioso == 1 && listas.preparados.size > 0) {
    node_t* sugerencia = desencolar();
    if (cmpnode(sugerencia, nullNode)) {
        printf("\n[Scheduler Warning] Se ha intentado cargar el nodo nulo en un hilo de ejecucion\n");
    } else {
        loadThread(sugerencia, i, j, k);  // Cargar informacion del pcb y programa en el hilo
        printf("[Scheduler] PCB cargado en hilo[%d][%d][%d]: PCB PID: %d\n", i, j, k, maquina.cpus[i].cores[j].hilos[k].executing->data->pid);
    }
}
```

Los programas y sus respectivos pcb's se ejecutan hasta encontrar la instrucción de salida en el programa. La ejecución de instrucciones ocurre en cada ciclo en el reloj central. La operación se realiza para todos los pcb's no nulos que se encuentren en ejecución.

```
if ((maquina.cpus[i].cores[j].hilos[k].flag_ocioso == 0) && (!cmpnode(maquina.cpus[i].cores[j].hilos[k].executing, nullNode)))
    executeProgram(i, j, k);    // ejecutar instrucciones
```

Una vez los programas ejecutan “F0000000”, terminan su conjunto de instrucciones, por lo que la ejecución termina, y el Scheduler los saca de los hilos de ejecución, encolándolos en la cola de terminados (terminated). Antes de eso, se hace una “foto finish” del estado del hilo al terminar el programa. En lo que respecta al encolamiento, al igual que la cola de preparados, la cola de “terminated” también es FIFO, pero no importa mucho ya que esta es solamente para hacer registro de los pcb’s terminados. Después, se limpia el hilo y sus estructuras para el siguiente programa a cargar en él, y por último se libera la memoria ocupada originalmente.

```
if ((maquina.cpus[i].cores[j].hilos[k].flag_ocioso == 1) && !cmpnode(maquina.cpus[i].cores[j].hilos[k].executing, nullNode)) {
    terminateStatus(i, j, k);                                       // Hacer una "foto finish" del estado del hilo cuando el programa termina
    encolar(maquina.cpus[i].cores[j].hilos[k].executing, terminated);       // Encolar nodo en la cola de terminados
    printf("[Scheduler] Se ha liberado el hilo[%d][%d][%d] del PCB %d\n", i, j, k, maquina.cpus[i].cores[j].hilos[k].executing->data->pid);
    freeThread(i, j, k);                                            // Limpiar estructuras del hilo
    freeProgram(maquina.cpus[i].cores[j].hilos[k].executing);               // Liberar memoria
}
```

## *Funciones del sistema*

El sistema se mueve gracias a los cinco componentes ya mencionados; el clock central, los dos timers, y el Scheduler y Loader. Ahora se van a analizar en detalle.

Empezando por el clock central, la mayor parte del código ya ha sido explicada anteriormente. Lo más importante de la función es la sincronización, de la cual se hablará más adelante. En lo que respecta a su quehacer, se limita a llamar a la función de ejecución de programas, asegurar que el simulador termina ordenadamente y desbloquear los elementos de sincronización para el resto del sistema.

```
if (kills == (maquina.info.cpus * maquina.info.cores * maquina.info.threads)) {
    sigKill = 1;   // activar secuencia de muerte
    if (killedThreads == 4)
        break;
} else {
    // caso general; ejecutar instrucciones
    for (i = 0; i < maquina.info.cpus; i++) {
        for (j = 0; j < maquina.info.cores; j++) {
            for (k = 0; k < maquina.info.threads; k++) {
                if ((maquina.cpus[i].cores[j].hilos[k].flag_ocioso == 1) && (loaderKill == 1) && (listas.preparados.size == 0)) {
                    kills++;  // contar cuantos hilos han terminado de trabajar finalmente
                }
                if ((maquina.cpus[i].cores[j].hilos[k].flag_ocioso == 0) && (!cmpnode(maquina.cpus[i].cores[j].hilos[k].executing, nullNode))) {
                    executeProgram(i, j, k);    // ejecutar instrucciones
                }
            }
        }
    }
}
```

```
// abrir barreras y esperar a que el resto del sistema termine de ejecutarse en esta iteracion
while(contTimers<2) {
    if (killedThreads == 4)
        break;
    pthread_cond_wait(&condTimers, &mutexTimers);
}
```

Una vez explicado el clock, pasamos a los timers. Estos simplemente son funciones de conteo y acumulación, y cuando se llega a un threshold (punto límite) configurable de ciclos de reloj, permiten que el Scheduler y el Loader trabajen. Importante: se trata de dos timers, uno para el Scheduler y otro para el Loader. Dado que la implementación es la misma para los dos, con el código de uno se pueden analizar ambos.

```
// funcion principal del TimerLoader; desbloquear el Loader cada calls_LoaderTick ticks del clock central
contLocalTimer++;
if ((killedThreads < 2) && (contLocalTimer >= maquina.info.calls_LoaderTick)) {
    pthread_cond_signal(&condLoader);
    pthread_cond_wait(&condLoader, &mutexLoader);
    contLocalTimer = 0;
}
```

Ahora, hablaremos de las piezas más importantes; el Scheduler y el Loader. Primero, el Scheduler se encarga de meter pcb's en hilos hardware o "flujos de ejecución", junto con sacarlos de ahí una vez terminen.

```
// Meter pcb's en hilos vacios
for (i = 0; i < maquina.info.cpus; i++) {
    for (j = 0; j < maquina.info.cores; j++) {
        for (k = 0; k < maquina.info.threads; k++) {
            if (cmpnode(maquina.cpus[i].cores[j].hilos[k].executing, nullNode) && maquina.cpus[i].cores[j].hilos[k].flag_ocioso == 1 && listas.preparados.size > 0) {
                node_t* sugerencia = desencolar();
                if (cmpnode(sugerencia, nullNode)) {
                    printf("\n[Scheduler Warning] Se ha intentado cargar el nodo nulo en un hilo de ejecucion\n");
                } else {
                    loadThread(sugerencia, i, j, k);  // Cargar informacion del pcb y programa en el hilo
                    printf("[Scheduler] PCB cargado en hilo[%d][%d][%d]: PCB PID: %d\n", i, j, k, maquina.cpus[i].cores[j].hilos[k].executing->data->pid);
```

```
// Sacar pcb's acabados de los hilos y limpiar memoria
for (i = 0; i < maquina.info.cpus; i++) {
    for (j = 0; j < maquina.info.cores; j++) {
        for (k = 0; k < maquina.info.threads; k++) {
            if ((maquina.cpus[i].cores[j].hilos[k].flag_ocioso == 1) && !cmpnode(maquina.cpus[i].cores[j].hilos[k].executing, nullNode)) {
                terminateStatus(i, j, k);                                   // Hacer una "foto finish" del estado del hilo cuando el programa termina
                encolar(maquina.cpus[i].cores[j].hilos[k].executing, terminated);       // Encolar nodo en la cola de terminados
                printf("[Scheduler] Se ha liberado el hilo[%d][%d][%d] del PCB %d\n", i, j, k, maquina.cpus[i].cores[j].hilos[k].executing->data->pid);
                freeThread(i, j, k);                                        // Limpiar estructuras del hilo
                freeProgram(maquina.cpus[i].cores[j].hilos[k].executing);               // Liberar memoria
```

En lo que respecta al Loader, éste busca los programas .elf, después actualiza el nombre del programa a cargar actualmente, y por último carga el programa. Como ya se verá más adelante, al cargar el programa se crea el pcb junto con su nodo correspondiente, a la vez que se carga el código del .elf en memoria principal.

```
if (loaderKill == 0) {
    buscarSiguienteElf();   // buscar siguiente programa a cargar en ./progs/
    if (loaderKill == 0) {
        actualizarNombreProgALeer();    // actualizar el nombre del programa a leer con el nombre del .elf que se ha encontrado
        loadProgram(++pidActual);       // cargar el .elf y crear pcb
        printf("[ Loader ] Programa cargado en memoria (prog%d.elf) y PCB creado (pid %d) ···\n", elfActual, pidActual);
    }
}
```

## *Finalización del simulador*

El proyecto y su sincronización ha tenido como cimientos la librería <*pthread.h*>. Como es natural, esta librería también incluye funciones de finalización. Dado que al comienzo de la ejecución del simulador se crean cinco “hilos maestros” ejecutándose concurrentemente, se ha decidido que la forma más correcta de terminar el programa sería usar la función *pthread_exit*. El uso de la función se ha tenido que hacer de forma ordenada y escalonada, ya que, tal y como está implementada la sincronización, el clock central desbloquea a los timers, y los timers al Scheduler y Loader respectivamente. Es decir, si la finalización de hilos no se hiciera de forma ordenada, el programa fallaría, imposibilitando el final correcto del programa.

Primero, hay que aclarar dónde y porqué se termina el programa. Esto lo “decide” la función *buscarSiguienteElf*. En concreto, cuándo no se encuentra en la carpeta *./progs/* ningún programa con número de programa superior al actual, se da por terminada la ejecución de todos los programas.

```
// no se ha encontrado un numero de programa mayor al actual
if (maxMinLocal == 1000) {
    printf("\n=================================================
    printf("Se han ejecutado todos los programas de ./progs/. No se
    printf("=================================================
    loaderKill = 1;  // Hacer una especie de Sigkill de la maquina
    return;
}
```

La activación del flag *loaderKill* afecta directamente a dos sistemas; el hilo del Loader, y el clock central. En el caso del primero, con la activación se evita cargar un hipotético siguiente programa el cual realmente no existe.

```
if (loaderKill == 0) {
    buscarSiguienteElf();   // buscar siguiente programa a cargar en ./progs/
    if (loaderKill == 0) {
        actualizarNombreProgALeer();    // actualizar el nombre del programa a leer con el nombre del .elf que se ha encontrado
        loadProgram(++pidActual);       // cargar el .elf y crear pcb
        printf("[ Loader ] Programa cargado en memoria (prog%d.elf) y PCB creado (pid %d) ···\n", elfActual, pidActual);
    }
}
```

En lo que respecta al hilo del clock, se desencadena la finalización de hilos. Para esto, si *loaderKill* está activado, se cuentan cuántos hilos han terminado de trabajar, y cuándo el número de hilos finalizados equivale al número de hilos hardware totales del sistema, se activa el flag de finalización para los hilos maestros.

```
// si todos los hilos hardware han terminado de ejecutar programas, y el loader ha detectado que ya no hay mas programas a cargar en ./progs/, activar secuencia de finalizacion
if (kills == (maquina.info.cpus * maquina.info.cores * maquina.info.threads)) {
    sigKill = 1;   // activar secuencia de muerte
    if (killedThreads == 4)
        break;
} else {
    // caso general; ejecutar instrucciones
    for (i = 0; i < maquina.info.cpus; i++) {
        for (j = 0; j < maquina.info.cores; j++) {
            for (k = 0; k < maquina.info.threads; k++) {
                if ((maquina.cpus[i].cores[j].hilos[k].flag_ocioso == 1) && (loaderKill == 1) && (listas.preparados.size == 0) && (maquina.cpus[i].cores[j].hilos[k].killed == 0)) {
                    kills++;  // contar cuantos hilos han terminado de trabajar finalmente
                    maquina.cpus[i].cores[j].hilos[k].killed = 1;   // matar hilo hardware
```

Nuevamente, la activación de otro flag desencadena otra finalización del funcionamiento. En este caso, se trata de la finalización de los threads maestros; Loader / Scheduler, Timers y Clock central. En orden cronológico, se empezará hablando del Loader y Scheduler. Primero, el Loader. Tal y como se ha mencionado antes, realmente este hilo es el que desencadena la finalización del programa. Sin embargo, no tiene porqué ser el primer

sistema en apagarse. Esto es debido a la sincronización entre sistemas; si la configuración con la que se ha lanzado el programa dictamina que, por ejemplo, el Loader se ejecuta más a menudo que el Timer, y el Loader muere antes, la sincronización de mutex y condiciones fallaría causando un interbloqueo fatal; el clock central se quedaría en espera a una *signal* que nunca llegará. Lo mismo ocurriría si la configuración favoreciera la ejecución del Scheduler en lugar del Loader.

En el caso del Loader:

```
// si se han cargado todos los programas que hay en ./progs/, matar hilo
if (sigKill == 1) {
    if (killedThreads == 0 && (maquina.info.calls_LoaderTick >= maquina.info.calls_SchedulerTick)) break;
    if (killedThreads == 1) break;  // si se ha finalizado el thread del Scheduler, finalizar este tambien
}
```

(fuera del while perpetuo)

```
// secuencia de finalizacion del hilo (muerte controlada del hilo)
killedThreads++;
pthread_cond_signal(&condLoader);
pthread_mutex_unlock(&mutexLoader);
printf("\nkilledThreads: %d\n", killedThreads);
printf("\nhLoader KILLED ---------------------------------------------------------------->\n");
pthread_exit(NULL);
```

En el caso del Scheduler:

```
// si se han cargado todos los programas que hay en ./progs/, matar hilo
if (sigKill == 1) {
    if (killedThreads == 0 && (maquina.info.calls_SchedulerTick >= maquina.info.calls_LoaderTick)) break;
    if (killedThreads == 1) break; // si se ha finalizado el thread del Loader, finalizar este tambien
}
```

(fuera del while perpetuo)

```
// secuencia de finalizacion del hilo (muerte controlada del hilo)
killedThreads++;
pthread_cond_signal(&condScheduler);
pthread_mutex_unlock(&mutexScheduler);
printf("\nkilledThreads: %d\n", killedThreads);
printf("\nhScheduler KILLED ------------------------------------------------------------->\n");
pthread_exit(NULL);
```

Subiendo en la jerarquía y orden de finalización, nos encontramos con los timers. Los timers se ejecutan con la misma frecuencia, ya que se desbloquean en cada ciclo del clock (otra cosa es que estos desbloqueen al Scheduler/Loader). Debido a esto, en este caso da igual cuál de los dos timers acaba antes, ya que al terminar uno siempre va a terminar el otro seguidamente. El código en ambos hilos es exactamente el mismo:

```
// si se ha dado la señal de muerte, matar hilo (se sale del while perpetuo, y se hace pthread_exit())
if (sigKill)
    if (killedThreads >= 2) break;
```

(fuera del while perpetuo)

```
// secuencia de finalizacion del hilo (muerte controlada del hilo)
contTimers++;
killedThreads++;
pthread_cond_signal(&condTimers);
pthread_mutex_unlock(&mutexTimers);
printf("\nkilledThreads: %d\n", killedThreads);
printf("\nhTimerScheduler KILLED -------------------------------------------------------->\n");
pthread_exit(NULL);
```

Por último, hablar sobre la finalización del clock central. Este es el corazón del simulador, y es el último sistema en ser matado. Es el encargado de mantener la sincronización correcta entre todos los sistemas del simulador, y dado a esto, es el último sistema en dejar de ejecutarse.

Tal y como se ha mencionado con anterioridad, cuándo todos los hilos hardware han terminado de ejecutar todos los programas de la carpeta de programas se activa el flag *sigKill*. Al activar el flag, se matan los hilos maestros (obviamente el clock no), y cuándo todos menos el clock han terminado, el clock sale de su while perpetuo para terminar de ejecutarse.

```
if (kills == (maquina.info.cpus * maquina.info.cores * maquina.info.threads)) {
    sigKill = 1;   // activar secuencia de muerte
    if (killedThreads == 4)
        break;
```

```
printf("\n\n\n================================================================================\n\n");
printf("   >> >> Se han cargado y ejecutado todos los programas de ./progs/ << <<\n\n");
printf("         >> Se ha detenido el reloj y las unidades de ejecucion <<\n\n");
printf("              >> Se ha terminado la ejecucion del simulador <<");
printf("\n\n\n================================================================================\n\n");
pthread_exit(NULL);
```

Para terminar con el programa, la ejecución vuelve al main, ya que los *pthread_join* han encontrado el código de retorno que esperaban de los hilos maestros, y finalmente se ejecuta el *return*.

```
pthread_join(masterThreads[0], NULL);
pthread_join(masterThreads[1], NULL);
pthread_join(masterThreads[2], NULL);
pthread_join(masterThreads[3], NULL);
pthread_join(masterThreads[4], NULL);

return 0;
```

# Ejecución de los programas

## *Carga de programas*

Llegados a este punto del informe, se van a analizar las partes del código que hacen las funcionalidades principales; desde buscar los archivos .elf en mediante llamadas al sistema, hasta ejecutar las instrucciones de esos mismos .elf simplificados.

Para empezar, vamos a analizar la carga de programas. Esto se realiza mediante tres funciones: *buscarSiguienteElf*, *actualizarNombreProgALeer*, y *loadProgram*. Las dos primeras se consideran funciones auxiliares, por lo que se encuentran en *funcAux.c,* mientras que la última se encuentra en *ejecProg.c*.

*BuscarSiguienteElf*, tal y como dice el nombre, busca el siguiente programa a cargar en la carpeta *./progs/*, la cual se encuentra en la raíz del proyecto. La metodología de la función es la siguiente: mediante llamadas al sistema, se abre el directorio *./progs/* y se va mirando el nombre de cada archivo en la carpeta. Dado que la nomenclatura de los .elf con los que se trabaja es *progXYZ.elf*, el número de programa siempre se va a encontrar en las posiciones 4, 5, y 6 del nombre. Se transforman los números *char* de esas posiciones en un número *int*, y se realiza una comparación para deducir si el programa siendo analizado se trata del siguiente a cargar (orden de menor a mayor, 0-999). A pesar de que en la documentación del proyecto se especificara que los números de programa serían consecutivos, la función se ha implementado de esta forma para permitir introducir programas sin números consecutivos.

```
if (directorio) {
    // while de lectura del directorio ./progs/
    while ((dir_struct = readdir(directorio)) != NULL) {
        // conversion de numero de programa de char a int
        numProgEnBusqueda[0] = dir_struct->d_name[4];
        numProgEnBusqueda[1] = dir_struct->d_name[5];
        numProgEnBusqueda[2] = dir_struct->d_name[6];
        numProgEnBus = strtol(numProgEnBusqueda, NULL, 10);

        // comprobar si el programa siendo analizado es el siguiente en orden
        if (numProgEnBus < maxMinLocal && numProgEnBus > elfActual) {
            maxMinLocal = numProgEnBus;
        }
    }
```

Tras ejecutarse *buscarSiguienteElf*, se llama a *actualizarNombreProgALeer*. El propósito de esta función es sencillo: actualizar el nombre del programa a cargar en memoria. Podría haberse combinado con el método anterior, pero por problemas en la implementación es una función separada. Se hace uso de la función *sprintf* para hacer conversión de tipos (*int* a *char*).

```
sprintf(&nombreProg[4], "%ld", elfActual);

if (elfActual < 10) {
    // solo hace falta coger un numero del nombre (numProg < 10)
    nombreProg[6] = nombreProg[4];
    nombreProg[5] = '0';
    nombreProg[4] = '0';
} else if (elfActual > 9 && elfActual < 100) {
    // solo hace falta coger dos numero del nombre (numProg > 9 && numProg < 100)
    nombreProg[6] = nombreProg[5];
    nombreProg[5] = nombreProg[4];
    nombreProg[4] = '0';
} else if (elfActual > 99 && elfActual < 1000) {
    // hay que coger todos los numeros del nombre (numProg > 99 && numProg < 1000)
    /* nada, el sprintf lo coloca ya bien */
}
```

Acabando con la carga de programas, *loadProgram*. La función se encarga de cargar el programa antes deducido en memoria. Primero, mediante llamadas al sistema se abre el fichero .elf (haciendo control de errores), y se consiguen los metadatos del archivo: el tamaño en bytes (para saber cuánta memoria va a necesitar), dónde comienza su segmento de instrucciones (normalmente *000000*), y dónde comienza su segmento de datos.

```
// Calcular tamaño en bytes que va a necesitar en physical
    stat(ruta, &st);
    tamaño = st.st_size;
    tamaño -= 26;
    tamaño = tamaño - (tamaño / 9);
    tamaño = (tamaño / 8) * 4;

// Leer comienzo de instrucciones (siempre 000000)
    read(fd, buf13, 13);
    buf13[12] = '\0';
    printf("\n%s\n", buf13); //write(1, buf13, 13);

    // caracteres 6 - 11 tienen la informacion
    bdir[0] = buf13[6]; bdir[1] = buf13[7]; bdir[2] = buf13[8]; bdir[3] = buf13[9]; bdir[4] = buf13[10]; bdir[5] = buf13[11];
    textSeg = (int)strtol(bdir, NULL, 16);

// Leer comienzo de datos
    read(fd, buf13, 13);
    buf13[12] = '\0';
    printf("%s\n", buf13); //write(1, buf13, 13);

    // caracteres 6 - 11 tienen la informacion
    bdir[0] = buf13[6]; bdir[1] = buf13[7]; bdir[2] = buf13[8]; bdir[3] = buf13[9]; bdir[4] = buf13[10]; bdir[5] = buf13[11];
    dataSeg = (int)strtol(bdir, NULL, 16);
```

Una vez conseguidos los metadatos, gracias a que el simulador ya sabe cuántos bytes va a ocupar el programa, se crea la tabla de páginas del programa. Para ello, primero se busca espacio en ella. Dado que en cada página entran hasta 64 instrucciones, se van a buscar *(tamaño del programa en bytes / 256)* páginas disponibles seguidas en la tabla de páginas. Una vez encontrado el número de páginas necesarias seguidas, se ocupan. Esto se indica poniendo un *“1”* en el primer byte del PTE correspondiente.

```
    // buscar en toda la tabla de paginas
    for (i = 0, ip = 0; i <= (49151 * 4); i+=4, ip++) {
        c0 = physical[(0x20EB00 + i+0)];

        // c0 == 1, es decir, el PTE indica que la pagina esta ocupada
        if (c0) {
            pagPosible = -1;
            pagCont = 0;
            continue;
        }

        // el contador esta vacio, y la pagina actual esta disponible
        if (!pagCont)
            pagPosible = i;

        pagCont++;

        // se tienen tantas paginas como son necesarias
        if (pagCont == npags)
            break;
    }

    // la tabla de paginas esta llena
    if (pagPosible == -1) {
        printf("Memoria llena. No hay sitio en memoria en el que cargar el programa. Esperando
        return;
    }

    printf("\n :: Pagina encontrada :: pagina %d | %x | dir physical: %x\n", ip, (0x20EB00 + pa

    // Ocupar PTEs en la tabla de paginas
    for (i = (0x20EB00 + pagPosible); i < (0x20EB00 + pagPosible + (npags * 4)); i+=4)
        physical[i] = 0x01;
```

Ocupada la tabla de páginas, se crea y configura el pcb del proceso; el struct *mm* se configura con los punteros al código y datos en memoria principal, junto con el puntero a la tabla de páginas también en memoria principal. Después, se lee y carga todo el código del .elf en la memoria reservada. Por último, se crea el nodo, se encola y se informa de la carga del programa:

```
p->mm = malloc(sizeof(mm_t));
p->mm->code = &(physical[numDirCode]);  // puntero a instrucciones
numDirCode += dataSeg;
p->mm->data = &(physical[numDirCode]);  // puntero a datos
p->mm->pgb = &(physical[(0x20EB00 + pagPosible)]);  // puntero a tabla de paginas
p->mm->psize = tamaño;



crear_node(&localNode, p);
encolar(localNode, preparados);
printNode(localNode);

printf("\n <--------------------------->\n");
printf("\nPrograma cargado en memoria principal ---> %s\n\n", ruta);
printf("Inicio del segmento de instrucciones: %x", textSeg);
printf("\nInicio de segmento de datos: %x", dataSeg);
printf("\nTamaño del programa: %d bytes", tamaño);
printf("\nTamaño del programa: %d lineas", tamaño/4);
printf("\n <--------------------------->\n\n");
```

```
while ((n = read(fd, buf, BUFSIZE)) > 0) {
------------------- primer byte de la palabra -------------
    bbyte[0] = buf[0]; bbyte[1] = buf[1];
    bb = (unsigned char)strtol(bbyte, NULL, 16);
    physical[contPhysical++] = bb;
------------------- segundo byte de la palabra -------------
    bbyte[0] = buf[2]; bbyte[1] = buf[3];
    bb = (unsigned char)strtol(bbyte, NULL, 16);
    physical[contPhysical++] = bb;
------------------- tercer byte de la palabra -------------
    bbyte[0] = buf[4]; bbyte[1] = buf[5];
    bb = (unsigned char)strtol(bbyte, NULL, 16);
    physical[contPhysical++] = bb;
------------------- cuarto byte de la palabra -------------
    bbyte[0] = buf[6]; bbyte[1] = buf[7];
    bb = (unsigned char)strtol(bbyte, NULL, 16);
    physical[contPhysical++] = bb;

    printf(" Physical: [%2x][%2x][%2x][%2x] | contPhysical:

    read(fd, buf, 1);   // Leer el '\n' del final de linea
}
```

## *Ejecución de instrucciones*

Hemos acabado de analizar la carga de programas. Ahora, siguiendo con el flujo de ejecución del simulador, se explicará la ejecución de los programas. Para empezar, el Scheduler carga el programa en un hilo vacío usando la función *loadThread*. Esta función se limita a asignar las estructuras de control del pcb en el hilo:

```
// Cargar hilo general
maquina.cpus[i].cores[j].hilos[k].executing = n;
for (w = 0; w < NUM_RGs; w++)
    maquina.cpus[i].cores[j].hilos[k].rgs[w] = 0;
maquina.cpus[i].cores[j].hilos[k].pc = n->data->mm->code;
maquina.cpus[i].cores[j].hilos[k].ir = 0;
maquina.cpus[i].cores[j].hilos[k].ptbr = n->data->mm->pgb;
maquina.cpus[i].cores[j].hilos[k].flag_ocioso = 0;

// Limpiar TLB
for (w = 0; w < TAM_TLB; w++) {
    maquina.cpus[i].cores[j].hilos[k].tlb->virtualPage[w] = -1;
    maquina.cpus[i].cores[j].hilos[k].tlb->physicalPage[w] = -1;
    maquina.cpus[i].cores[j].hilos[k].tlb->score[w] = -1;
}
```

Ahora el hilo hardware ya tiene la información necesaria para empezar a trabajar. La ejecución de este trabajo, es decir, la ejecución de instrucciones, se hará mediante el clock central, el cual llama a la función *executeProgram*. Antes de continuar, aclarar que el método hace uso constante de manejo y desplazamiento de bits. La función comienza con la fase de *Fetch*, en la cual se consigue la instrucción accediendo al contenido de la dirección que apunta el pc. La instrucción se guarda en el registro de instrucción, IR:

```
// Fase de Fetch. Buscar la instruccion y conseguirla
/* el pc esta apuntando a la direccion del array physical con el trozo de codigo (la palabra objetivo actual) del programa */
maquina.cpus[i2].cores[j2].hilos[k2].ir = 0;
maquina.cpus[i2].cores[j2].hilos[k2].ir += *(maquina.cpus[i2].cores[j2].hilos[k2].pc + 0);
maquina.cpus[i2].cores[j2].hilos[k2].ir = maquina.cpus[i2].cores[j2].hilos[k2].ir << 8;
maquina.cpus[i2].cores[j2].hilos[k2].ir += *(maquina.cpus[i2].cores[j2].hilos[k2].pc + 1);
maquina.cpus[i2].cores[j2].hilos[k2].ir = maquina.cpus[i2].cores[j2].hilos[k2].ir << 8;
maquina.cpus[i2].cores[j2].hilos[k2].ir += *(maquina.cpus[i2].cores[j2].hilos[k2].pc + 2);
maquina.cpus[i2].cores[j2].hilos[k2].ir = maquina.cpus[i2].cores[j2].hilos[k2].ir << 8;
maquina.cpus[i2].cores[j2].hilos[k2].ir += *(maquina.cpus[i2].cores[j2].hilos[k2].pc + 3);

maquina.cpus[i2].cores[j2].hilos[k2].pc += 4;
```

La siguiente fase se trata de la *fase de decodificación*. En esta se analiza de qué tipo de instrucción se trata la instrucción conseguida anteriormente; *ld, st, add o exit*. Para ello, se hace un *switch* con el primer byte de la palabra, y se actúa en consecuencia:

<u>*LOAD*</u>: se necesita el registro de destino, y la dirección de origen.

<u>*STORE*</u>: se necesita el registro de destino, y la dirección de destino.

<u>*ADD*</u>: se necesita el registro de origen 1, origen 2, y destino.

<u>*EXIT*</u>: nada.

```
switch(c0) {
    case 0:
        // Conseguir del IR el registro destino, direccion virtual de origen
        r0 = maquina.cpus[i2].cores[j2].hilos[k2].ir << 4;
        r0 = r0 >> 28;
        dirvirtual = maquina.cpus[i2].cores[j2].hilos[k2].ir << 8;
        dirvirtual = dirvirtual >> 8;
        printf("    >> Instruccion de tipo LD | %x %x(%d) %x |", c0, r0, r0, dirvirtual);
        break;
    case 1:
        // Conseguir del IR el registro origen, direccion virtual de destino
        r0 = maquina.cpus[i2].cores[j2].hilos[k2].ir << 4;
        r0 = r0 >> 28;
        dirvirtual = maquina.cpus[i2].cores[j2].hilos[k2].ir << 8;
        dirvirtual = dirvirtual >> 8;
        printf("    >> Instruccion de tipo ST | %x %x(%d) %x |", c0, r0, r0, dirvirtual);
        break;
    case 2:
        // Conseguir del IR el registro destino, registro origen 1, registro origen 2
        r0 = maquina.cpus[i2].cores[j2].hilos[k2].ir << 4;
        r0 = r0 >> 28;
        r1 = maquina.cpus[i2].cores[j2].hilos[k2].ir << 8;
        r1 = r1 >> 28;
        r2 = maquina.cpus[i2].cores[j2].hilos[k2].ir << 12;
        r2 = r2 >> 28;
        printf("    >> Instruccion de tipo ADD | %x %x(%d) %x(%d) %x(%d)\n", c0, r0, r0,
        break;
    case 0xF:
        // Nada; acabar con la ejecucion en la fase de operacion
        printf("    >> Instruccion de tipo EXIT | %x\n", c0);
        break;
    default:
        printf("    >> Error. Tipo de instruccion no compatible con la arquitectura. | IR:
        break;
}
```

Terminada la fase de decodificación, se entra en la *búsqueda de operandos*. La fase se divide en dos segmentos claramente separados, todo dependiendo de las necesidades de la dirección. Como es natural, casi todo el trabajo de implementación se la ha llevado la parte en la que se necesita traducir una dirección virtual en una física.

En esta parte, primero se consiguen el offset dentro de la página y la página virtual a la que hay que acceder, y después se mira la TLB del hilo hardware para intentar traducir la dirección de forma rápida, evitando un acceso a memoria. Si la traducción *página virtual - página física* se encuentra en la TLB, se coge directamente el PTE de la misma y se resuelve la dirección, accediendo al valor de la dirección. De lo contrario, si la traducción *página virtual - página física* no se encuentra en la TLB, hay que acceder a la tabla de páginas que se encuentra en la memoria principal para ver el valor del PTE objetivo. Al igual que en caso anterior, al final se resuelve la dirección y se accede al valor de la misma.
Por otra parte, si la instrucción siendo ejecutada no requiere de traducción de direcciones, sino que sólamente requiere trabajar con registros, se consigue el valor necesario y se almacena en los registros pertinentes. No se va a enseñar código de esta fase debido a su longitud, pero la implementación está completamente comentada en *ejecProg.c*.

Una vez se ha completado la búsqueda de operandos, se llega a la *fase de operación.* Sólamente se opera con la instrucción *add*, en la que se suman números.

```
// Fase de Operar. Operar y comprobar si finalizar la ejecucion del programa
if (c0 == 0xF) {
    flag_exit = 1;  // activar secuencia de finalizacion
} else if (c0 == 2) {
    r3 = r1 + r2;   // sumar registros
}
```

Por último, la *fase de resultados*. Tal y como indica el nombre, dependiendo del tipo de instrucción se guarda el resultado de la instrucción en un registro o en una dirección física. Obviamente, la instrucción *exit* no tiene resultado numérico, pero en la fase anterior se comenzó su cadena de eventos particular.

```
// Fase de Resultados. Dejar resultado en registros o memoria
switch(c0) {
    case 0:
        maquina.cpus[i2].cores[j2].hilos[k2].rgs[r0] = r1;
        break;
    case 1:
        /* Dejar en physical el valor de r0 */
        printf(" \n              >> st %x, %x\n", r0, dirObjetivo);
        int r00, r01, r02, r03;
        r00 = r01 = r02 = r03 = r0;
        r00 = r00 >> 24;
        physical[dirObjetivo + 0] = r00;
        r01 = r01 << 8;
        r01 = r01 >> 24;
        physical[dirObjetivo + 1] = r01;
        r02 = r02 << 16;
        r02 = r02 >> 24;
        physical[dirObjetivo + 2] = r02;
        r03 = r03 << 24;
        r03 = r03 >> 24;
        physical[dirObjetivo + 3] = r03;
        break;
    case 2:
        maquina.cpus[i2].cores[j2].hilos[k2].rgs[r0] = r3;
        break;
    case 0xF:
        printf("    >> Instruccion de tipo EXIT | No se va a operar\n");
        break;
```

## *Finalización de programas*

En el punto en el que se llega a la instrucción de salida *F0000000*, en la función *executeProgram* se pone en marcha la finalización de ejecución del programa en cuestión. Resumidamente, al leer dicha instrucción, en la fase de operación se activa un flag, *flag_exit*, y al final de la función se activa otro flag. En este caso se trata un flag especial, *flag_ocioso*, ya que hay uno por cada hilo hardware. Se usa para indicar que el hilo ha terminado su tarea actual.

```
// Fase de Operar. Operar y comprobar si finalizar la ejecucion del programa
if (c0 == 0xF) {
    flag_exit = 1;  // activar secuencia de finalizacion
} else if (c0 == 2) {
    r3 = r1 + r2;   // sumar registros
}

if (flag_exit == 1)  {
    /* Liberar adecuadamente el hilo para que el Scheduler cargue otro programa */
    printf("\n\n\n\n ==== Se ha acabado con la ejecucion del programa con PID %d===
    maquina.cpus[i2].cores[j2].hilos[k2].flag_ocioso = 1;
    flag_exit = 0;
}
```

La activación del flag_ocioso repercute principalmente en dos aspectos; el clock central no intentará ejecutar instrucciones de ese thread, y cuándo se llame al Scheduler, este sacará el proceso del hilo, liberando la memoria ocupada por el programa.

```
if ((maquina.cpus[i].cores[j].hilos[k].flag_ocioso == 0
    executeProgram(i, j, k);
}
```

```
if ((maquina.cpus[i].cores[j].hilos[k].flag_ocioso == 1) && !cmpnode(maquina.cpus[i].cores[j].hilos[k].executing, nullNode)) {
    terminateStatus(i, j, k);                                       // Hacer una "foto finish" del estado del hilo cuar
    encolar(maquina.cpus[i].cores[j].hilos[k].executing, terminated);       // Encolar nodo en la cola de terminados
    printf("[Scheduler] Se ha liberado el hilo[%d][%d][%d] del PCB %d\n", i, j, k, maquina.cpus[i].cores[j].hilos[k].executing-
    freeThread(i, j, k);                                            // Limpiar estructuras del hilo
    freeProgram(maquina.cpus[i].cores[j].hilos[k].executing);               // Liberar memoria
}
```

## *Tablas de páginas y traducción de direcciones*

Uno de los componentes a los que iba dirigido este proyecto es precisamente la tabla de páginas y la traducción de direcciones virtuales-físicas. Tal y como se puede observar en la función *executeProgram*, es uno de los aspectos fundamentales de la ejecución de instrucciones.

La tabla de páginas se empieza en la dirección que se da en el ejemplo del documento aclarativo del proyecto, *0x20EB00*, mientras que el espacio de direccionamiento del usuario comienza en *0x400000,* también extraída del mismo documento. Teniendo en cuenta que cada PTE son 4 bytes, es decir, una palabra en memoria, la inicialización de la tabla de páginas es sencilla; se completan todos los PTEs (49151 entradas) con la dirección física en la que empieza cada página:

```
int pte = 0x00400000;   // 00 00 00 00 (3 bytes menos significativos direccion physical)

for (i = 0; i <= (49151 * 4); i+=4) {
    physical[(0x20EB00 + i+0)] = 0x00;

    // Calcular nueva pagina; romper direccion en 3 bytes (3 unsigned chars)
    d1 = pte << 8;
    d1 = d1 >> 24;
    d2 = pte >> 8;
    d2 = d2 << 24;
    d2 = d2 >> 24;
    d3 = pte << 24;
    d3 = d3 >> 24;

    physical[(0x20EB00 + i+1)] = (unsigned char)d1;
    physical[(0x20EB00 + i+2)] = (unsigned char)d2;
    physical[(0x20EB00 + i+3)] = (unsigned char)d3;

    pte += 256;
}
```

Con la tabla de páginas inicializada correctamente, ya se puede empezar a usar. El principal sistema que hará uso de la tabla de páginas será la TLB de cada hilo hardware. El propósito de la TBL es simple, pero muy útil; se trata de una memoria caché, en la que se guarda la correspondencia *dirección virtual-dirección física* de las páginas de las tablas de páginas de los programas. Gracias a esto, se ahorran accesos a memoria en instrucciones *load* y *store*. La TBL consiste de dos arrays; uno en el que se almacena la página virtual, y otro que almacena correspondencia de la dirección física de esa página virtual. A su vez, cabe mencionar que hay un sistema de puntuaciones para elegir una PTE víctima cuándo la TLB se llena.

```
typedef struct tlb_ {
    unsigned int* virtualPage;      // un array de numeros (numero de pagina virtual)
    unsigned int* physicalPage;     // un array de copias de PTEs
    int* score;                     // un array de numeros (puntuaciones de las PTEs)
} tlb_t;
```

La TLB tiene dos mecánicas; hit o miss. Es decir, que la página que se requiere se encuentra en la TLB o no. Si se encuentra en la TBL, simplemente se accede a la entrada correspondiente y se mira la dirección. De lo contrario, si la traducción no se encuentra en la caché, hace falta acceder a memoria principal para conocer la página virtual. En ambos casos, una vez se consigue la dirección se le suma el offset para llegar a la dirección final. Con la dirección final resuelta, se accede al valor de la dirección (con una indirección *).
Cuándo se tiene un miss, se actualiza la tabla de páginas con una nueva entrada, y ya sea miss o hit, se actualiza los valores de puntuación de las PTEs en la TLB. La TLB y la consecuente traducción de direcciones se puede observar en la función *executeProgram.*

## *Sincronización del sistema*

El funcionamiento explicado previamente funciona de la manera que se esperaba gracias a la correcta sincronización de los diferentes hilos que forman la máquina. Para ello, se ha utilizado la librería pthread y las estructuras de control “mutex” y variables de condición “cond” que proporciona, además de las funciones.

Dicho esto, paso a explicar cómo se ha llevado a cabo la sincronización. Toda la máquina se mueve gracias al reloj central, es decir, el hilo del clock. Este abre el “mutex” “mutexTimers” para que los timers puedan seguir moviendo la máquina, y se queda esperando la señal de los dos timers que certifican que han terminado. Una vez ha recibido una señal de terminado de cada timer, transmite la señal de que cada timer continúe a su siguiente iteración, pero gracias a que el clock tiene en propiedad el mutex “mutexTimers”, ninguno de los dos podrá hacer una iteración hasta que el clock les abra la barrera nuevamente. El siguiente código es del hilo clock:

```
// abrir barreras y esperar a que el resto del sistema termine de ejecutarse en esta iteracion
while(contTimers<2) {
    if (killedThreads == 4)
        break;
    pthread_cond_wait(&condTimers, &mutexTimers);
}

//printf("\n%d | Alemania del Este | kills: %d\n", cont, kills);

if (killedThreads == 4)
    break;

contTimers = 0;
usleep(maquina.info.frec);

// A terminado la iteracion actual, habilitar la siguiente iteracion
pthread_cond_broadcast(&condAB);
pthread_mutex_unlock(&mutexTimers);
```

Los timers, como ya se ha explicado, sólo pueden continuar sus iteraciones cuando el clock se lo permite. En estas iteraciones, se cuenta hasta un número establecido por el usuario, el cual decide la cantidad de ciclos necesarios para que ya sea el Scheduler o el Process Generator se llame y realice su tarea correspondiente. Los números son independientes, pero se puede poner el mismo valor en ambos si se desea. Cuando el contador de ciclos ha contado lo suficiente, crea un “tick”, es decir, una interrupción. Entonces, se abre el mutex para que ya sea el Scheduler o Process Generator pueda hacer su trabajo. Al igual que con el clock, los timers también esperan a que les lleguen las señales desde el hilo al que se le abren. Ambos hilos son idénticos, menos por la cantidad de ciclos necesarios y los mutexes que abren.

Código del timer del Scheduler:

```
pthread_mutex_lock(&mutexScheduler);

// funcion principal del TimerScheduler; desbloquear el Scheduler cada calls_Sched
contLocalTimer++;
if ((killedThreads < 2) && (contLocalTimer >= maquina.info.calls_SchedulerTick)) {
    pthread_cond_signal(&condScheduler);
    pthread_cond_wait(&condScheduler, &mutexScheduler);
    contLocalTimer = 0;
}

pthread_mutex_unlock(&mutexScheduler);

// si se ha dado la señal de muerte, matar hilo (se sale del while perpetuo, y se
if (sigKill)
    if (killedThreads >= 2) break;

// se ha hecho todo lo que se tenia que hacer en esta iteracion; se indica que se
contTimers++;
pthread_cond_signal(&condTimers);
pthread_cond_wait(&condAB, &mutexTimers);
```

Código del timer del Process Generator:

```
pthread_mutex_lock(&mutexLoader);

// funcion principal del TimerLoader; desbloquear el Loader cada calls_LoaderTick
contLocalTimer++;
if ((killedThreads < 2) && (contLocalTimer >= maquina.info.calls_LoaderTick)) {
    pthread_cond_signal(&condLoader);
    pthread_cond_wait(&condLoader, &mutexLoader);
    contLocalTimer = 0;
}

pthread_mutex_unlock(&mutexLoader);

// si se ha dado la señal de muerte, matar hilo (se sale del while perpetuo, y se
if (sigKill)
    if (killedThreads >= 2) break;

// se ha hecho todo lo que se tenia que hacer en esta iteracion; se indica que se
contTimers++;
pthread_cond_signal(&condTimers);
pthread_cond_wait(&condAB, &mutexTimers);
```

Por último, el Scheduler y el generador de procesos esperan a que se les deje trabajar mediante una espera a la señal correspondiente. Al igual que con los timers, el código que se encarga de la sincronización es el mismo, por lo que sólo se va a poner el del Scheduler:

```
//Aqui se acaba todo lo que tengo que hacer, y signaleo que ya he acabado mi tarea/iteracion
pthread_cond_signal(&condScheduler);
pthread_cond_wait(&condScheduler, &mutexScheduler);
```

# Código y Uso

## *Funciones auxiliares*

Las partes más importantes ya se han explicado, siendo estas las el clock, Scheduler y Process Generator. También se ha detallado la sincronización del sistema, por lo que ahora se procederá a explicar brevemente las funciones y utilidades que usan los sistemas vitales de la máquina, junto con el main, archivos header y el makefile.

Para empezar, se va explicar las funciones auxiliares encontradas en el fichero “funcAux.c”. Antes se mencionó que en el generador de procesos se crea un pcb y que se le hace un wrapper convirtiéndolo realmente en un nodo. Estas son las funciones que lo administran, a la vez que la definición del struct pcb y el struct node encontrados en “types.h” junto con la función que determina si dos nodos son iguales mediante el pid del pcb:

```c
typedef struct pcb {
    int pid;
    status_t* status;
    mm_t* mm;
} pcb_t;
```

```c
typedef struct node {
    node_t* next;
    pcb_t* data;
} node_t;
```

```c
// crear nodo nuevo
void crear_node(node_t** n, pcb_t* p) {
    *n          = malloc(sizeof(node_t));
    (*n)->data  = p;
    (*n)->next  = nullNode;
}
```

```c
// comparar nodos; son iguales si el pid del pcb que contienen son iguales
int cmpnode(node_t* a, node_t* b) {
    int pidA = a->data->pid;
    int pidB = b->data->pid;
    if (pidA == pidB)
        return true;

    return false;
}
```

A su vez, previamente se hizo referencia a las colas de preparados y terminados. Para implementarlas, primero se tuvieron que crear y definir las listas y sus unidades, los nodos. Esto se encuentra en el fichero de cabecera “types.h”, mientras que las funciones de encolar y desencolar están en “funcAux.c”.

Definición de las estructuras:

```c
typedef struct lista {
    node_t* first;
    node_t* last;
    int size;
} lista_t;

typedef struct coleccion_listas {
    lista_t preparados;
    lista_t terminated;
} coleccion_listas_t;
```

## *Función main*

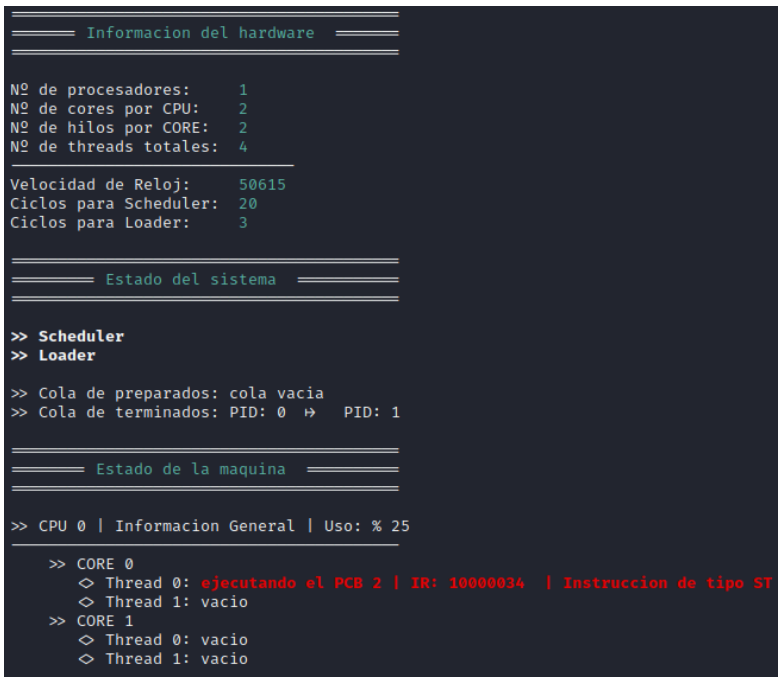
La función main es simple. Se recogen los parámetros de la máquina mediante argv[], se realiza el control de errores y se llama a la función encargada de inicializar las estructuras del sistema. En esta función, se inicializan las estructuras de sincronización proporcionadas por la librería “pthread”, se crean las unidades de trabajo nulas de la máquina (nullProc y nullNode), se inicializa el sistema de colas y por último se hace lo propio con la estructura de cpus, cores e hilos.

Una vez se completa la puesta en marcha de las estructuras, se crean los hilos, con lo que la máquina se pone en marcha de forma definitiva.

Por último, en el directorio raíz del proyecto se encuentra el makefile, el cual ha sido creado siguiendo las indicaciones de un video referenciado al final del documento. Esto se ha hecho debido a que nunca he creado un makefile, ni se me ha enseñado cómo hacerlo, a pesar de que no fuera una tarea difícil en este caso.

## *Salida e Interfaz*

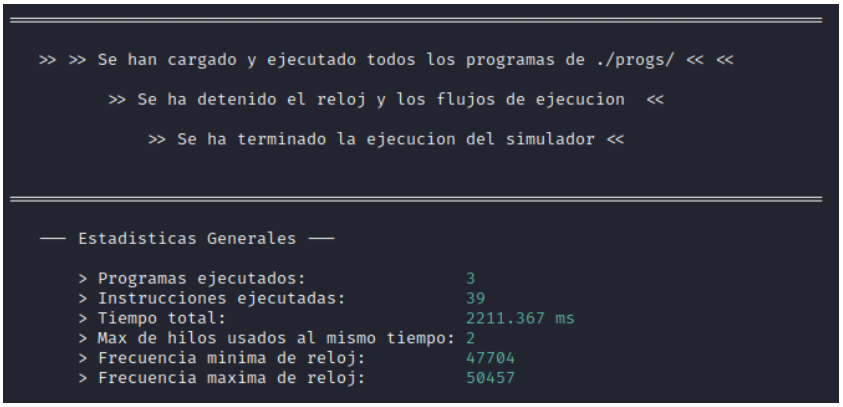
La interfaz del programa es simple y se entiende fácilmente. La interfaz es estática, pero los datos en ella se actualizan en tiempo real. En total hay tres secciones; la primera es para recordar al usuario la configuración de la máquina que ha escogido, la segunda informa sobre actividad en el Scheduler / Loader, y la última sección es la más interesante, ya que es dónde se puede apreciar el uso de la máquina junto con la ejecución de instrucciones.




Debido a limitaciones a la hora de presentar los datos en tiempo real, no se puede enseñar toda la información que se ha estado *debbugeando* y comprobando durante el desarrollo, especialmente la relacionada con el análisis de instrucciones y direcciones. Sin embargo, el código informante sigue en el código fuente, por lo que si se quiere, se puede descomentar. Una vez terminan todos los .elf’s y se acaba el programa, se nos indican varias estadisticas generales. Cabe aclarar que, debido a la precariedad de la interfaz, es posible que algunos eventos no se recogen y plasmen correctamente. En concreto, me refiero a momentos en

los que, por ejemplo, en un mismo ciclo el Scheduler carga dos programas en hilos distintos.

Se hace seguimiento de la cantidad de programas ejecutados, la cantidad de instrucciones totales ejecutadas entre todos los programas, el mayor número de hilos usados al mismo tiempo, y el tiempo que ha tomado la ejecución del simulador.




A su vez, y como inclusión extra de cara a darle más funcionalidades al proyecto, se ha hecho que la frecuencia que se indica en la configuración no sea la frecuencia absoluta real a la que se ejecutan las instrucciones; la frecuencia real puede distar hasta en un %20 de la frecuencia dictaminada al lanzar el simulador. Teniendo esto en cuenta, al final del programa también se informa al usuario de las frecuencias máxima y mínima a la que ha llegado el reloj central. El siguiente trozo de código, encontrado en el hilo del clock, es el que decide ciclo a ciclo el desplazamiento de la frecuencia:

```c
int delta = rand() % 500;
int masmenos = rand() % 2;

if (masmenos == 1) {    // sumar frecuencia generalmente
    if ((maquina.info.frec + delta) > (maquina.info.frecOG + (maquina.info.frecOG * 0.2))) {
        maquina.info.frec -= delta; // frecuencia demasiado alta, restar
    } else {
        maquina.info.frec += delta;
    }
} else {                // restar frecuencia generalmente
    if ((maquina.info.frec - delta) < (maquina.info.frecOG - (maquina.info.frecOG * 0.2))) {
        maquina.info.frec += delta; // frecuencia demasiado baja, sumar
    } else {
        maquina.info.frec -= delta;
    }
}
```

Toda la interfaz del programa está contenida en el fichero *printMaquina.c*. Este contiene las dos funciones de printeo, *printMachine y printStatus.*

## *Uso del programa*

El programa se usa de forma sencilla. A la hora de ejecutarlo, requiere de 6 parámetros proporcionados por el usuario. Estos son, en este orden:

| | |
|---|---|
| <*cpus*> | Nº de cpus de la máquina |
| <*cores*> | Nº de cores por cada cpu |
| <*threads*> | Nº de threads por cada core |
| <*ciclosGenerator*> | Nº de ciclos del reloj simulado necesarios para un tick del gen. |
| <*ciclosScheduler*> | Nº de ciclos del reloj simulado necesarios para un tick del Sch. |
| <*frecuencia*> | Velocidad del reloj del sistema, cada cuando ocurre un ciclo<br>Cuanto más alto sea el valor, más lento<br>Cuanto más bajo sea el valor, más rápido<br>Se recomienda como mínimo 250.000 - 300.000 |

Para que el usuario sepa qué está ocurriendo, la información del estado se muestra en pantalla mediante printf's con la información de las colas y el Scheduler y Process Generator.

Por último, el proyecto se puede compilar mediante el makefile antes mencionado usando la utilidad "make". Usando la opción "clean" en el make, se puede limpiar la carpeta raíz del proyecto y eliminar la carpeta "obj". Cuando se "hace make", se crea en la carpeta raíz del proyecto el ejecutable con el nombre "simulador". Este nombre se puede cambiar mediante la variable correspondiente en el makefile.

# Conclusiones

La principal idea del proyecto era crear un Kernel básico, lo cual ha sido muy atractivo a la vez que muy instructivo. Con esto me refiero a que he aprendido mucho acerca de la estructura básica de los Kernel, a pesar de que el simulador esté configurado para memoria paginada *sin swapping.*

A pesar de que en las primeras clases pareciera una tarea imposible, gracias al profesorado y sus explicaciones a la vez que a la ayuda del internet he aprendido la estructura y funcionamiento interno de un Kernel básico limitado, ya que, como ya he dicho antes, el horizonte del proyecto es limitado.

A su vez, una de las características que más me ha llamado la atención ha sido lo vital que son las colas y su gestión a la hora de diseñar un sistema así; Las características y políticas de estas junto con una política general al final determinan el funcionamiento general del sistema. A pesar de que antes de cursar la asignatura ya sabía que las colas y listas eran un aspecto importante a la hora de desarrollar un sistema de esta área, la importancia de éstas ha resultado mayor de lo que esperaba.

Por último, me gustaría hablar sobre las posibles mejoras y horas dedicadas.
En lo que respecta a las mejoras, me siento en la obligación de mencionar que el funcionamiento original que tenía pensado para el sistema era diferente al que finalmente se ha terminado implementando. Descrito brevemente, la idea original era tener un sistema que cargara los pcb's con política FIFO, como se ha terminado haciendo, pero estos pcb's tendrían la capacidad de quedarse bloqueados por un número aleatorio de ciclos. Cuando el Scheduler detectara esto, marcaría el pcb como "avisado", y si siguiera bloqueado, lo expulsaría a una cola de bloqueados, desde la cual volverían a ejecución tras ciertos ciclos. También se había pensado en hacer que un pcb fuera dependiente de otro, lo cual se traduciría en que el proceso A necesitaría que primero acabara el proceso B para entrar en ejecución. Al final no se implementó este sistema debido a ciertas dificultades y obstáculos en el tiempo que se nos dió de desarrollo.

En lo que respecta a las horas dedicadas, decir brevemente que primero se pensó en la estructura final (lo cual llevó 4 horas), después se hicieron pruebas de sincronización para ver cómo usar los mutex correctamente (8 horas), y finalmente se procedió a desarrollar el código implementando el sistema previamente diseñado (20 horas). Antes de dar por concluída la implementación, se hicieron pruebas para comprobar el correcto funcionamiento del sistema (2 horas). Al final, se procedió a redactar esta memoria y comentar el código (15 horas). También hay que aclarar que los sistemas no funcionaron a la primero, y arreglar esto tomó su tiempo (10 horas). Por último, la implementación de la finalización del programa mediante la función *pthread_exit* fue problemática (5 horas). Dicho esto, al proyecto finalmente se le han dedicado en torno a 65 horas. Esta figura y las horas por apartado descritas antes sólo tienen en cuenta la versión del proyecto entregada. Es decir, no se han considerado las horas invertidas en el otro sistema antes descrito en el apartado de mejoras. Si se tuvieran en cuenta, la cantidad de horas dedicadas incrementaría sustancialmente.

# Bibliografía y Referencias


Egela; conocimiento, funcionamiento y políticas proporcionado por el profesorado:
OS - Tema 2-1 Control de procesos.pdf (ehu.eus)
OS - Tema 2-2 Sincronización.pdf (ehu.eus)
OS - Tema 2-3 Interbloqueo.pdf (ehu.eus)
OS - Tema 2-4 Planificación - Indicadores.pdf (ehu.eus)
OS - Tema 2-5 Planificación - Políticas.pdf (ehu.eus)
OS - Tema 2-6 Linux Scheduler.pdf (ehu.eus)
OS - Tema 3-1 Sistema de gestión de memoria.pdf (ehu.eus)
OS - Tema 3-2 Memoria virtual.pdf (ehu.eus)
OS - Tema 3-3 Memoria cache.pdf (ehu.eus)

Egela y manual de linux; apuntes de programación y uso de la librería “pthread” de C:
OS - Tema 2-7 pthread.pdf (ehu.eus)
OS - Tema 2-7 pthread 2.pdf (ehu.eus)
pthreads(7) - Linux manual page (man7.org)

Egela; clase c con ejemplos de *cooking* del terminal
OS - Visualization.c

Github; control de versiones y repositorio para transferir archivos cómodamente a VMs:
https://github.com

Stackoverflow; buscar y resolver dudas y errores de código:
https://es.stackoverflow.com

Youtube; tutorial, construcción de un makefile:
https://youtu.be/0XlVyZAfQEM

Draw.io; hacer diagramas de la estructura interna del sistema:
https://app.diagrams.net/