eman ta zabal zazu

Universidad del País Vasco
Euskal Herriko Unibertsitatea

Documentación Kernel

por

Aitor González González

Sistemas Operativos
Grado en Ingeniería Informática
UPV/EHU
11/11/2022

Índice

Introducción

Un sistema operativo es un software que actúa como intermediario entre el hardware y los demás programas que se ejecutan en una computadora. Su función principal es gestionar los recursos del sistema, como la memoria, los procesadores, los dispositivos de entrada y salida y los periféricos, y proporcionar servicios comunes para que los demás programas puedan ejecutarse de manera eficiente y correcta. Algunos ejemplos de sistemas operativos populares son Microsoft Windows, macOS, Linux y Android.

El Scheduler, es un componente del sistema operativo que se encarga de asignar recursos del sistema, como la CPU y la memoria, a los procesos que están en ejecución. El objetivo del Scheduler es asegurar que cada proceso obtenga una cantidad justa y equitativa de tiempo de uso de la CPU y otros recursos, para que ningún proceso se bloquee o retrase demasiado tiempo.

Existen diferentes algoritmos de programación que utilizan los sistemas operativos para determinar cómo deben asignarse los recursos a los procesos. Algunos de estos algoritmos son el algoritmo de planificación de prioridades, el algoritmo de planificación round-robin y el algoritmo de planificación multinivel. Cada algoritmo tiene sus propias ventajas y desventajas, y el sistema operativo puede elegir el que mejor se ajuste a sus necesidades y requisitos.

El loader es un componente del sistema operativo que se encarga de cargar los programas en la memoria y prepararlos para su ejecución. Cuando un usuario quiere ejecutar un programa, el loader primero carga el código del programa en la memoria, luego realiza una serie de tareas necesarias para preparar el programa para su ejecución, como enlazar las bibliotecas y cargar los datos necesarios.

El loader también se encarga de gestionar el espacio de memoria disponible y asegurar que los programas tengan suficiente espacio para ejecutarse de manera eficiente. Por ejemplo, si un programa necesita más memoria de la que está disponible, el loader puede tratar de liberar espacio desechando algunos programas o datos que ya no se están utilizando.

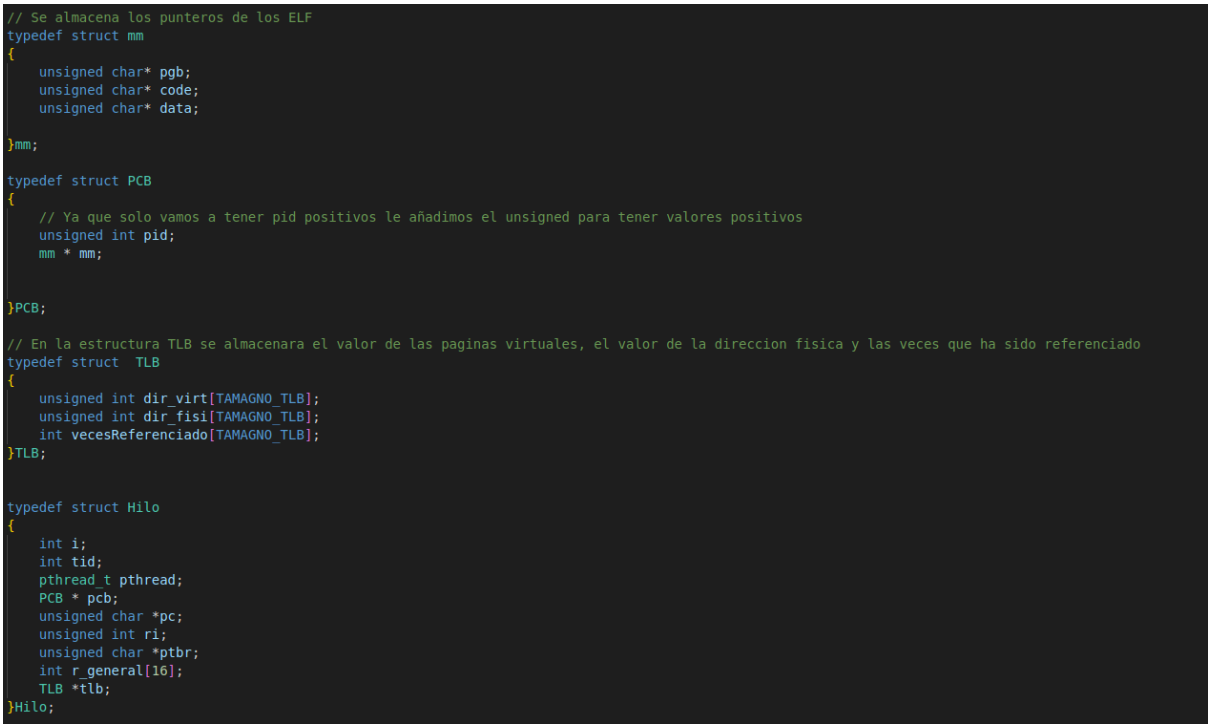
En general, el loader es un componente esencial del sistema operativo que ayuda a asegurar que los programas se ejecuten de manera correcta y eficiente en la computadora.

Desarrollo del proyecto

En este primer desarrollo hemos creado la arquitectura del sistema, para ello hemos dividido este desarrollo en subdesarrollados, facilitando así su implementación.

Arquitectura del sistema

En esta imagen podemos ver las estructuras creadas para la implementación del kernel:



```c
// Se almacena los punteros de los ELF
typedef struct mm
{
    unsigned char* pgb;
    unsigned char* code;
    unsigned char* data;

}mm;

typedef struct PCB
{
    // Ya que solo vamos a tener pid positivos le añadimos el unsigned para tener valores positivos
    unsigned int pid;
    mm * mm;


}PCB;

// En la estructura TLB se almacenara el valor de las paginas virtuales, el valor de la direccion fisica y las veces que ha sido referenciado
typedef struct  TLB
{
    unsigned int dir_virt[TAMAGNO_TLB];
    unsigned int dir_fisi[TAMAGNO_TLB];
    int vecesReferenciado[TAMAGNO_TLB];
}TLB;


typedef struct Hilo
{
    int i;
    int tid;
    pthread_t pthread;
    PCB * pcb;
    unsigned char *pc;
    unsigned int ri;
    unsigned char *ptbr;
    int r_general[16];
    TLB *tlb;
}Hilo;
```

```c
typedef struct Core
{
    int i;
    Hilo *hilos;
}Core;

typedef struct CPU
{
    int i;
    Core *cores;
}CPU;


typedef struct Machine
{
    CPU *cpus;
    int elf;
    int ult_elf;
}Machine;
```

La primera parte ha sido la obtención de los diferentes parámetros necesarios para la inicialización del kernel.

```c
// Primero vamos a recibir los parametros
int numCpus = atoi(argv[1]);
int numCores = atoi(argv[2]);
int numThreads = atoi(argv[3]);
int tiempoReloj = atoi(argv[4]);
int tiempoDisp = atoi(argv[5]);
int tiempoLoader = atoi(argv[6]);
int numTotalThreads = numCpus * numCores * numThreads;
```

A continuación vamos a inicializar los hilos del sistema y vamos a crear los elementos necesarios para realizar la sincronización del kernel.

```c
int i, j, k, l, cont=0;
srand(time(NULL));
inicializar_lista(&processQueue);
pthread_mutex_init(&m_clk, NULL);
pthread_cond_init(&c1_clk, NULL);
pthread_cond_init(&c2_clk, NULL);
pthread_mutex_init(&m_disp, NULL);
pthread_cond_init(&c1_disp, NULL);
pthread_cond_init(&c2_disp, NULL);
pthread_mutex_init(&m_load, NULL);
pthread_cond_init(&c1_load, NULL);
pthread_cond_init(&c2_load, NULL);
basic_threads = malloc(sizeof(pthread_t)*NB_THREADS);
pthread_create(&basic_threads[0], NULL, clk, NULL);
hilos = malloc(sizeof(struct Hilo *)*numTotalThreads);

machine.cpus = (CPU*)malloc(sizeof(CPU) * numCpus);
for (i = 0; i < numCpus; i++) {
    machine.cpus[i].cores = (Core*)malloc(sizeof(Core) * numCores);
    for (j = 0; j < numCores; j++) {
        machine.cpus[i].cores[j].hilos = (Hilo*)malloc(sizeof(Hilo) * numThreads);
        for (k = 0; k < numThreads; k++) {
            pthread_create(&machine.cpus[i].cores[j].hilos[k].pthread, NULL, crearHilo, &machine.cpus[i].cores[j].hilos[k]);
            machine.cpus[i].cores[j].hilos[k].i=cont/numThreads;
            machine.cpus[i].cores[j].hilos[k].tid=cont;
            machine.cpus[i].cores[j].hilos[k].pcb=NULL;
            machine.cpus[i].cores[j].hilos[k].tlb = malloc(sizeof(TLB));

            for(l=0; l<TAMAGNO_TLB;l++){
                machine.cpus[i].cores[j].hilos[k].tlb->dir_fisi[l]= -1;
                machine.cpus[i].cores[j].hilos[k].tlb->dir_virt[l]= -1;
                machine.cpus[i].cores[j].hilos[k].tlb->vecesReferenciado[l]= -1;
            }

            hilos[cont]= &machine.cpus[i].cores[j].hilos[k];

            cont++;

        }
    }
}

iniciarPageTable();
```

Después vamos a crear los hilos necesarios para el clock, scheduler, loader y los dos timers.

```c
pthread_create(&basic_threads[1], NULL, timDisp, NULL);
pthread_create(&basic_threads[2], NULL, timLoader, NULL);
pthread_create(&basic_threads[3], NULL, loader, NULL);
pthread_create(&basic_threads[4], NULL, dispatcher, NULL);
```

Nuestro reloj va a simular el paso del tiempo, esto va a ser muy importante para poder generar los ciclos para controlar el sistema.

```c
void* clk(){

    done_clk = 0;
    while (1)
    {
        // Primero bloqueamos todo los hilos, esto quiere decir que todos los hilos han acabado
        pthread_mutex_lock(&m_clk);

        while (done_clk < numTotalThreads + NB_THREADS -3 -killed){
            // Esperamos a que todos los hilos se ejecuten, que nos manden el flag en la condicion de que han acabado los hilos
            pthread_cond_wait(&c1_clk, &m_clk);
        }
        done_clk = 0;
        // Primero se lo manda a todos, que ya tienen disponible
        pthread_cond_broadcast(&c2_clk);
        // El segundo desbloquea el mutex y va a volver al while
        pthread_mutex_unlock(&m_clk);

    }


}
```

Tenemos distintos timers, uno que se encarga del scheduler y otro que se encarga del loader.

```c
void * timScheduler(){
    // Este timer se activa gracias al reloj, por tanto necesamos un mutex para permitir esa comunicacion
    // Por un lado tenemos la comunicacion clock timer dispacher
    // por otro tenemos el timer dispacher con el dispacher
    int contador = 0, periodo = 0;
    pthread_mutex_lock(&m_clk);
    periodo = frec_dis;
    while(1){
        done_clk++;
        contador++;
        // printf("\nContador= %d", contador);
        // Cada vez que se despierta el hilo la variable contador suma 1, si el contador es == al periodo aleatorio
        // entonces se le llama al process generator para que genere un nuevo proceso
        if (contador == periodo){
            printf("\nHa saltado el temporizador del dispacher");
            pthread_mutex_lock(&m_disp);
            while (!doneDisp){

                pthread_cond_wait(&c1_disp, &m_disp);

            }
            doneDisp=false;

            pthread_cond_broadcast(&c2_disp);
            pthread_mutex_unlock(&m_disp);

            periodo = frec_dis;
            contador = 0;
        }

        pthread_cond_signal(&c1_clk);
        pthread_cond_wait(&c2_clk, &m_clk);

    }
}
```

Esta es la función del loader, para almacenar los pcb hemos creado una lista enlazada de PCBs, esta lista a su vez utiliza funciones desarrolladas en el linkedList.c, una de las funciones inicializa la lista, la segunda añade los PCB a la

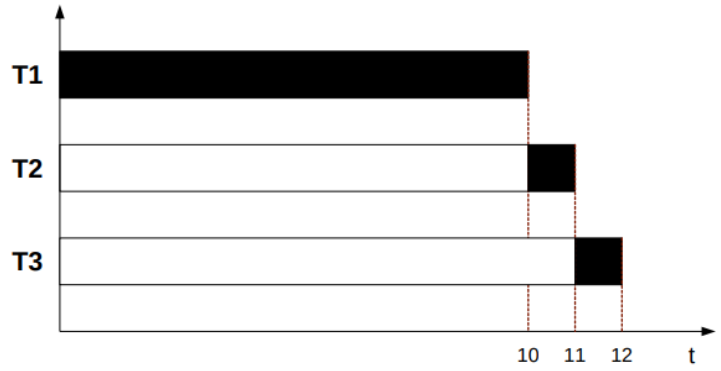
lista y la tercera los obtiene, he decidido no poner esta parte del código en la memoria porque me parecía trivial.

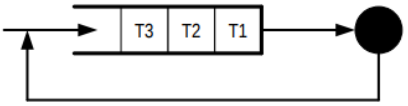
```
void *loader(){
    int pids = 0;//variable que almacena los pids almacenados
    machine.elf = -1;
    leerELF(); //almacenamos el ultimo elf
    pthread_mutex_lock(&m_load);
        while(1){
            doneLoader=true;
            printf("\nLoader= %d", doneLoader);
            if (machine.ult_elf+1 == contador_p){
                int k;
                for(k=0;k<numTotalThreads;k++){
                    pthread_join(hilos[k]->pthread, NULL);
                }
                for(k=0;k<numTotalThreads;k++){
                    if (k!=4){
                        pthread_join(basic_threads[k], NULL);
                    }
                }
                pthread_cond_signal(&c1_load);
                pthread_mutex_unlock(&m_load);
                pthread_exit(NULL);
            }
            if(processQueue.estado == 0){
                processQueue.estado = 1;
                PCB * pcb = (PCB* )malloc(sizeof(PCB));
                pcb->pid=pids;
                machine.elf++;
                cargarELF(pids, processQueue, pcb);
                printf("\n Pc:code C %s\n", pcb->mm->code);
                pids++;
                anadirNodo(&processQueue, pcb, -1);//añadimos en la ultima pos
                printf("\n %s", "tamaño de la process queue");
                printf("\n %d", processQueue.tamagno);
                processQueue.estado = 0;
            }
            pthread_cond_signal(&c1_load);
            pthread_cond_wait(&c2_load, &m_load);
        }

}
```

Tenemos dos tipos de hilos en el sistema, por un lado están los hilos básicos que son estructuras de pthread que brindan soporte básico al sistema, como el reloj, dos timers, el loader y el scheduler. Y por otro lado tenemos los hilos de la máquina que son los hilos que hacen referencia a la máquina. Se almacenan en la estructura "machine", aunque también existe una estructura llamada "hilos" donde cada índice apunta a la misma dirección que "machine", lo que facilita el acceso a los hilos. Estos hilos trabajan a la frecuencia del reloj y cada uno ejecutará un proceso elegido por el scheduler hasta su finalización.

Política Seleccionada

La política FIFO (First-In, First-Out) es una política de planificación de procesos utilizada en sistemas operativos para determinar el orden en que se deben ejecutar los procesos que están en la cola de ejecución. Según esta política, los procesos se ejecutan en el orden en que llegaron a la cola de ejecución, es decir, el primer proceso en llegar será el primer proceso en ser atendido y ejecutado.
La política FIFO es muy simple y fácil de implementar, pero puede no ser muy eficiente en términos de tiempo de ejecución de procesos, ya que los procesos que llegaron primero pueden tardar mucho tiempo en ser completados, lo que puede retrasar la ejecución de los procesos que llegaron después. Además, esta política no tiene en cuenta la prioridad de los procesos ni el tiempo que cada uno necesita para ser completado.





Aun así, la política FIFO es útil en algunas situaciones, como cuando se desea garantizar un procesamiento justo y equitativo de todos los procesos o cuando se quiere evitar que algunos procesos tengan prioridad sobre otros.

Gestión de memoria

La gestión de memoria en el kernel es un proceso crucial para el correcto funcionamiento de un sistema operativo. El kernel es responsable de asignar y administrar la memoria del sistema, asegurando que los procesos y programas tengan acceso a la memoria que necesitan para ejecutarse.

Hay varios mecanismos de gestión de memoria utilizados por los kernels, pero algunos de los más comunes son:

- Particionamiento estático: El sistema de particionado estático divide la memoria en partes fijas de tamaño establecido, cada una de las cuales se asigna a un proceso específico. Este es un enfoque simple pero inflexible, ya que un proceso puede ocupar más o menos memoria de lo que realmente necesita.
- Particionamiento dinámico: El sistema de particionado dinámico divide la memoria en partes que pueden variar de tamaño, y los procesos pueden solicitar y liberar memoria en tiempo de ejecución. Este es un enfoque más flexible y eficiente, ya que permite una mejor utilización de la memoria.
- Paginación: La paginación es un sistema de gestión de memoria que divide la memoria en páginas de tamaño fijo. Cada proceso tiene un conjunto de páginas asignadas, y el kernel administra qué páginas están en la memoria principal y cuáles están en el disco duro. La paginación permite una mejor utilización de la memoria, ya que los procesos sólo tienen acceso a las páginas que realmente necesitan en un momento dado.
- Memoria virtual: La memoria virtual es un sistema de gestión de memoria que permite a los procesos acceder a más memoria de la que realmente está disponible en el sistema. El kernel simula la existencia de más memoria mediante el uso de una técnica llamada "swapping", que consiste en intercambiar las páginas de memoria que se encuentran en disco duro con las que se encuentran en memoria principal.

En resumen, la gestión de memoria en el kernel es un proceso clave para el funcionamiento de un sistema operativo, ya que permite a los procesos y programas acceder a la memoria que necesitan para ejecutarse de manera eficiente y asegurando un correcto funcionamiento del sistema.

Las páginas físicas son bloques de memoria físicamente existentes en el hardware del sistema, mientras que las páginas virtuales son bloques de memoria asignados a un proceso por el sistema operativo.

La principal diferencia entre las páginas físicas y las páginas virtuales es que las páginas físicas están limitadas por la cantidad de memoria física disponible en el sistema, mientras que las páginas virtuales pueden ser mayores debido a la existencia de la memoria virtual.

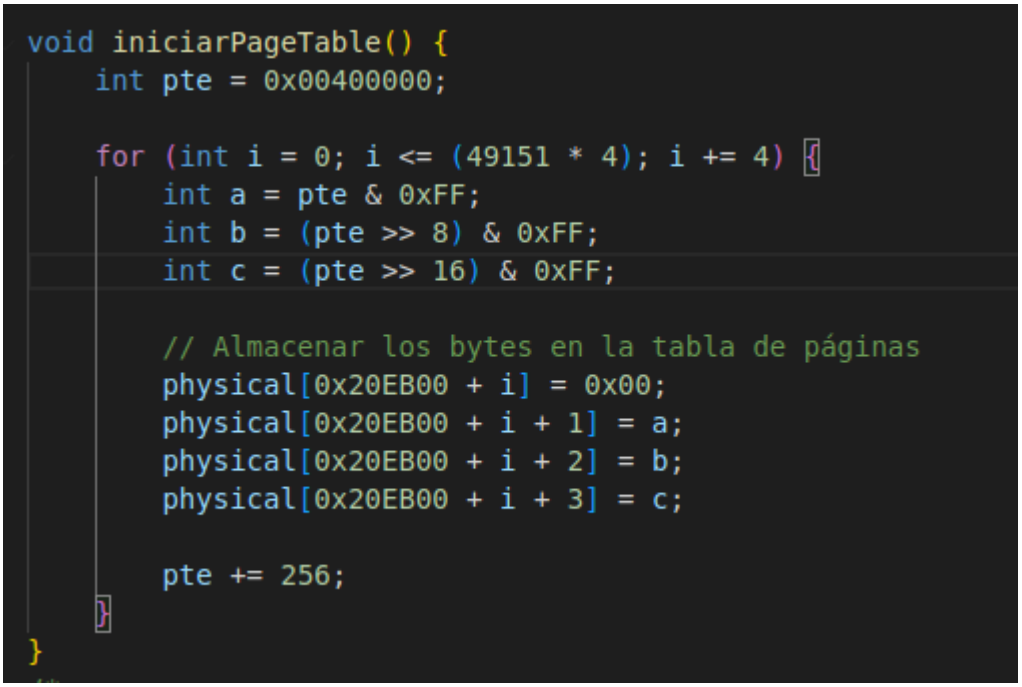
La memoria virtual permite al sistema operativo simular la existencia de más memoria de la que realmente está disponible en el sistema, permitiendo a los procesos acceder a más memoria de la que físicamente está presente. El sistema operativo hace esto mediante el uso de técnicas de swapping, donde las páginas de memoria que no están siendo utilizadas en ese momento son intercambiadas con otras que están en el disco duro.

En resumen, las páginas físicas son bloques de memoria físicamente existentes en el sistema, mientras que las páginas virtuales son bloques de memoria asignados a un proceso por el sistema operativo a través de la memoria virtual.

iniciarPageTable

Se utilizan operaciones de máscara de bits para extraer los 3 bytes menos significativos de la dirección física en lugar de operaciones de desplazamiento de bits. Esto hace que el código sea más fácil de leer y entender. Se hace uso del operador & con una máscara de 0xFF (o 255 en decimal) para obtener los últimos 8

bits del número que corresponde al byte menos significativo de un número de 32 bits. Finalmente, se agrega la instrucción de incrementar la dirección física al final del bucle.



```
void iniciarPageTable() {
    int pte = 0x00400000;

    for (int i = 0; i <= (49151 * 4); i += 4) {
        int a = pte & 0xFF;
        int b = (pte >> 8) & 0xFF;
        int c = (pte >> 16) & 0xFF;

        // Almacenar los bytes en la tabla de páginas
        physical[0x20EB00 + i] = 0x00;
        physical[0x20EB00 + i + 1] = a;
        physical[0x20EB00 + i + 2] = b;
        physical[0x20EB00 + i + 3] = c;

        pte += 256;
    }
}
```

leerELF

Este código está leyendo un archivo ELF desde un directorio llamado "./elfs/". Comienza abriendo el directorio utilizando la función opendir(). Si el directorio se abre correctamente, declara una variable llamada "directorio" que se utilizará para recorrer los archivos en el directorio. Luego, crea una variable llamada "num_elf" que es una lista de caracteres de tamaño 3, y una variable llamada "n_elf" que es un entero. A continuación, utiliza un bucle while para leer todos los archivos en el directorio utilizando la función readdir(). Dentro del bucle, extrae los tres dígitos del nombre del archivo y lo almacena en la variable "num_elf". Luego, convierte la variable "num_elf" a un entero y la almacena en la variable "n_elf". A continuación, compara el valor de "n_elf" con el valor actual de "machine.ult_elf" y si el valor de "n_elf" es mayor que "machine.ult_elf" o machine.elf es igual a cero, asigna el valor de "n_elf" a "machine.ult_elf". Este proceso se realiza para todos los archivos en el directorio. Finalmente, después de que el bucle haya terminado, el código cierra el directorio utilizando la función closedir().

```c
void leerELF(){
    DIR * directorio = opendir("./elfs/");
    if(directorio!=NULL){
        struct dirent *entry;
        char num_elf[3]= "   ";
        int n_elf;
        // Vamos a recorrer el directorio completo
        while ((entry = readdir(directorio))!=NULL)
        {
            num_elf[2] = entry->d_name[6];
            num_elf[1] = entry->d_name[5];
            num_elf[0] = entry->d_name[4];

            n_elf = strtol(num_elf,NULL,10);

            if(n_elf>machine.ult_elf || machine.elf==0){

                machine.ult_elf = n_elf;
            }

        }

        closedir(directorio);

    }
}
```

cargarELF

Este código está cargando un archivo ELF (Formato Ejecutable y Enlazable) desde una ruta específica ("./elfs/progXXX.elf") donde XXX es un número de tres dígitos. Abre el archivo, obtiene su tamaño en bytes y lee las primeras dos líneas del archivo, que contienen las direcciones de memoria de las secciones ".text" y ".data" del archivo ELF. Luego, calcula el número de instrucciones en el archivo y el número de páginas necesarias para almacenarlas en la memoria. El código luego busca en la Tabla de Páginas por páginas vacías y guarda las direcciones de esas páginas.

```
void cargarELF(int pids, linkedList processQueue,  PCB * pcb){
    char path[19] = "./elfs/progXXX.elf\0";
    path[13] = '0'+machine.elf%10;
    path[12] = '0'+machine.elf/10;
    path[11] = '0'+machine.elf/100;
    int fd;
    if((fd = open(path, O_RDONLY) )<0){
        exit(-1);
    }
    struct stat sta;
    int tamagno = 0;
    stat(path, &sta);
    tamagno = sta.st_size;
    printf("\nEste el el tamaño del fichero en bytes,  %d",tamagno);
    tamagno = tamagno -26;
    int longlinea = 13;
    //vamos a leer la primera linea del fichero
    char buff[longlinea];
    read(fd, buff, longlinea);
    buff[12] = '\0';
    char dirTxt[7];
    int i;
    for(i=0;i<6;i++){
        dirTxt[i]=buff[6+i];
    }
    dirTxt[6]='\0';
    //vamos a leer la segunda linea del fichero
    read(fd, buff, longlinea);
    buff[12] = '\0';
    char dirData[7];
    for(i=0;i<6;i++){
        dirData[i]=buff[6+i];
    }
    dirData[6]='\0';
```

```
int numInstruc = tamagno / 9;
tamagno = numInstruc * 4;
int numPag = numInstruc/64 + 1;
int pagObjectivo, contador=0;
printf("\nNumero de pagina %d\n", numPag);
unsigned int *dir_pte = malloc(sizeof(unsigned int)*numPag);
for(i=0; i < 49152 *4; i+=4){
    if(physical[(0x20EB00 + i)] !=1){
        dir_pte[contador] = 0x20EB00 + i;
        contador++;
        if (contador==numPag)break;
    }
}
if(contador!=numPag){
    machine.elf--;
    return;
}
for(i = 0; i<numPag; i++){
    physical[dir_pte[i]] = 1;
}

unsigned int a, b, c;
a = physical[dir_pte[0]+1];
b = physical[dir_pte[0]+2];
c = physical[dir_pte[0]+3];

int direccion_codigo = 0;
direccion_codigo += a;
direccion_codigo = (direccion_codigo << 8) + b;
direccion_codigo = (direccion_codigo << 8) + c;
```

```
pcb->mm = malloc(sizeof(mm));
pcb->mm->code = &(physical[direccion_codigo]);
int data_d = (int)strtol(dirData, NULL, 7);
direccion_codigo = direccion_codigo + data_d;
pcb->mm->data = &(physical[direccion_codigo]);
pcb->mm->pgb = &(physical[(dir_pte[0])]);
direccion_codigo = direccion_codigo - data_d;
char * buffer = malloc(sizeof(char)*9);
char hexa[3];
hexa[2]='\0';
unsigned char pal;

while ((read(fd, buffer, 8)) > 0) {
    buffer[8]='\0';

    for(i=0; i<4;i++){
        hexa[0]= buffer[i*2];
        hexa[1]= buffer[i*2+1];
        pal = (unsigned char)strtol(hexa, NULL, 16);
        physical[direccion_codigo++] = pal;

    }
    read(fd, buffer, 1);
}

close(fd);
free(dir_pte);
free(buffer);
```

Cargar y limpiar hilo

Una función carga los valores del hilo y la otra las resetea.

```c
//carga los valores del pcb al hilo
void cargarHilo(Hilo *hilo, PCB *pcb){
    printf("\nhilo : %d, cargado", hilo->tid);
    hilo->pcb = pcb;
    //se asigna la direccion inicial
    hilo->pc = pcb->mm->code;
    hilo->ri = 0;
    hilo->ptbr = pcb->mm->pgb;
    //reseteamos registros y tlb
    for(int i=0; i<16;i++){
        hilo->r_general[i]=0;
    }
    for(int i=0; i<TAMAGNO_TLB;i++){
        hilo->tlb->dir_fisi[i]= -1;
        hilo->tlb->dir_virt[i]= -1;
    }
}

/*
Resetea los valores
*/
void limpiarHilo(Hilo *hilo){
    printf("\nhilo : %d, limpiado", hilo->tid);
    hilo->pc = NULL;
    hilo->ri = -1;
    hilo->pcb = NULL;
    hilo->ptbr = NULL;
    for(int i=0; i<16;i++){
        hilo->r_general[i]=0;
    }
    for(int i=0; i<TAMAGNO_TLB;i++){
        hilo->tlb->dir_fisi[i]= -1;
        hilo->tlb->dir_virt[i]= -1;
        hilo->tlb->vecesReferenciado[i]=-1;
    }

}
```

ejecFases

Este código implementa una tabla de páginas y una tabla de traducción (TLB) para un sistema de memoria virtual. Recupera la instrucción del contador de programa (PC) y recupera el código de operación de la instrucción. El código de operación se utiliza para determinar el tipo de instrucción, como una LOAD, STORE, ADD o EXIT.

Si la instrucción es una LOAD o STORE, el código recupera la dirección virtual de la instrucción y la utiliza para buscar la dirección física correspondiente en la TLB. Si no se encuentra la dirección virtual en la TLB, el código busca en la tabla de páginas y actualiza la TLB con la dirección física.

```
int ejecFases(Hilo *hilo){

    int i;

    hilo->ri = 0;
    for(i = 0;i<TAM_PAL-1;i++){

        hilo->ri += *(hilo->pc + i);

        hilo->ri = hilo->ri <<8;

    }

    hilo->ri += *(hilo->pc + 3);
    hilo->pc +=4;
    char a = hilo->ri >> 28;
    char ADD[3] = "ADD";
    char EXIT[3] = "EXT";
    char LOAD[3] = " LD";
    char STORE[3] = " ST";
    unsigned char b, c, d ,e;
    unsigned int dir_virtual = -1;


    switch (a) {
        case 0:
        case 1:
            b = hilo->ri >> 28;
            dir_virtual = hilo->ri & 0x0FFFFFFF;
            break;
        case 2:
            b = hilo->ri >> 28;
            c = (hilo->ri >> 24) & 0xF;
            d = (hilo->ri >> 20) & 0xF;
            break;
        case 0xF:
            break;
        default:
            break;
    }
```

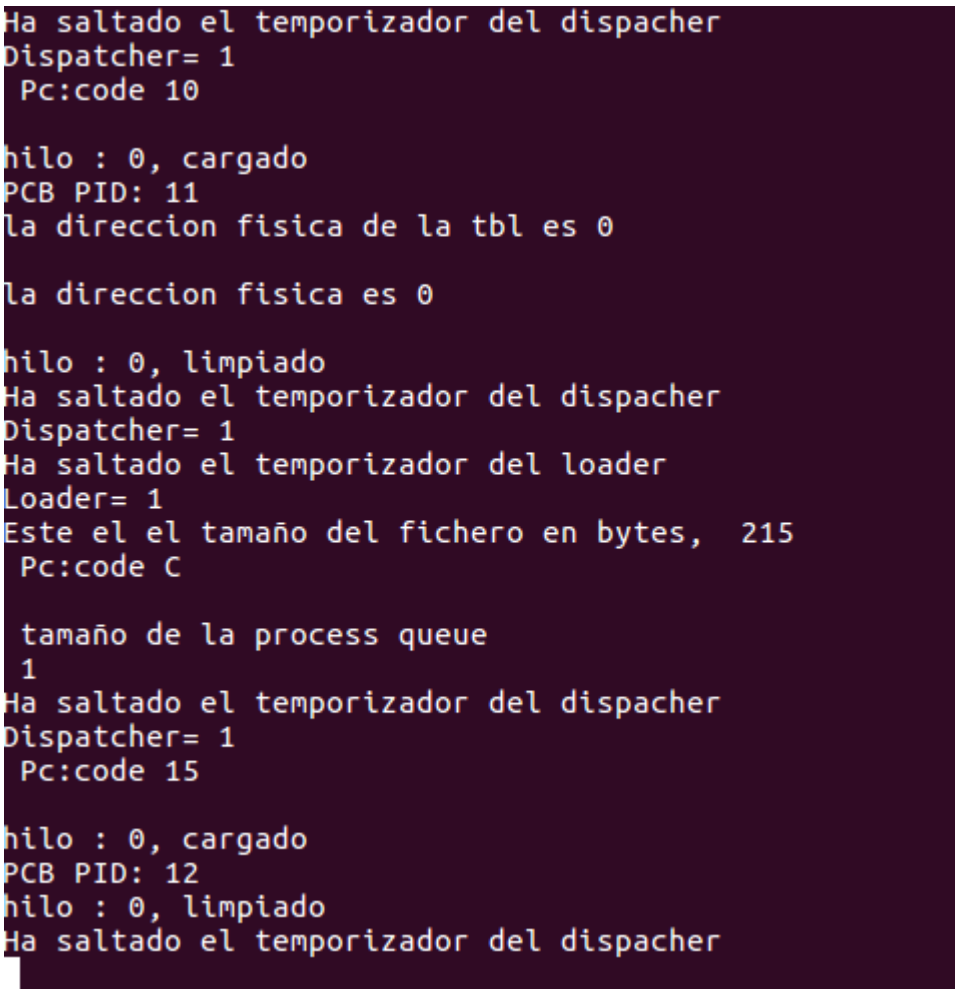
```c
unsigned int pagina, pagVirtual;
unsigned int direcFisica = 0, direcFin=0;
if (dir_virtual!=-1){
    pagina = (dir_virtual << 24) >>24 ;
    pagVirtual  = dir_virtual >> 8 ;
    int encontrado =0;
    int posicion = -1;
    for(i=0; i<TAMAGNO_TLB;i++){
        if(hilo->tlb->dir_virt[i]==pagVirtual){
            encontrado = 1;
            hilo->tlb->vecesReferenciado[i] ++;
            posicion = i;
            printf("\nla direccion fisica de la tbl es %x\n", hilo->tlb->dir_fisi[i]);
            break;
        }
    }


    unsigned char d0, d1, d2, d3;
    if (encontrado){
        printf("\nla direccion fisica es %x\n", direcFisica);
        direcFisica = ((hilo->tlb->dir_fisi[posicion] << 8) >> 8) + pagina;
        for(i = 0; i<TAM_PAL-1;i++){
            direcFin += physical[direcFisica + i];
            direcFin = direcFin << 8;
        }
        direcFin += physical[direcFisica + 3];

    }else{
    int minimo = 1000, pos=-1;
    for(i=0;i<TAMAGNO_TLB;i++){
        if(hilo->tlb->vecesReferenciado[i]<minimo){
            minimo = hilo->tlb->vecesReferenciado[i];
            pos = i;
        }
    }

    hilo->tlb->dir_virt[pos]=pagVirtual;
    hilo->tlb->vecesReferenciado[pos]=-1;

    unsigned char * pte = hilo->ptbr + (pagVirtual * 4);
```

```
unsigned int te = 0;
for(i=0;i<TAM_PAL;i++){

    te+=*(pte +i);

    te = te <<8;


}

te = te >> 8;

hilo->tlb->dir_fisi[pos]= te;

direcFisica = (hilo->tlb->dir_fisi[pos] << 8) >> 8 + pagina;
direcFisica = hilo->tlb->dir_fisi[pos] + pagina;

for(i = 0; i<TAM_PAL-1;i++){
        direcFin += physical[direcFisica + i];
        direcFin = direcFin << 8;
    }
direcFin += physical[direcFisica + 3];

}
```

```
switch(a){
    // Caso Load
    case 0:
        c = direcFin;
        hilo->r_general[b] = c;
        break;
    // Caso Store
    case 1:
        b = hilo->r_general[c];
        int bb, cc, dd, ee;
        bb = cc = dd = ee = b;
        bb = bb >> 24;
        physical[direcFisica + 0] = bb;
        cc = (cc << 8) >>24;
        physical[direcFisica + 1] = cc;
        dd = (dd << 16) >> 24;
        physical[direcFisica + 2] = dd;
        ee = (ee << 24)>> 24;
        physical[direcFisica + 3] = ee;
        break;
    // Caso suma
    case 2:
        hilo->r_general[b] = hilo->r_general[c]+hilo->r_general[d];
        break;
    // Caso exit
    case 0xF:
        return(-1);
        break;
    default:
        break;

}

return(0);
```

Ejecución del programa

En esta imagen se puede ver la ejecución del kernel



Conclusiones

En conclusión, el sistema operativo es un software esencial que actúa como intermediario entre el hardware y los programas que se ejecutan en una computadora. Su función principal es gestionar los recursos del sistema para asegurar que los programas se ejecuten de manera correcta y eficiente. El Scheduler, el loader y la memoria virtual son componentes clave del sistema operativo que ayudan a gestionar los recursos del sistema, cargar programas y aumentar la cantidad de memoria disponible.

Creo que la práctica ha tenido bastante complejidad, ya que al principio me encontraba perdido, aunque con esfuerzo una vez superado ese primer obstáculo he podido completar la tarea de forma satisfactoria.