Práctica:
Simulador de un kernel

Curso 24-25

Sistemas Operativos

By dalferr

Índice

Introducción

Esta práctica es la implementación de un simulador de un Kernel en C. En esta documentación comentaremos su compilación, su uso, la estructura del código y el desarrollo de cada parte del kernel: Arquitectura del sistema, planificador y gestor de memoria. El código se ubica en el siguiente repositorio:

- https://github.com/dalferr/SimuladorKernel

El repositorio cuenta con dos directorios, uno con la implementación de las partes 1 y 2, funcionando, y el otro directorio con parte de la implementación de la parte 3, pero esta no he conseguido que funcione correctamente.

Compilación

1. Nos ubicamos dentro del directorio de la parte que queramos ejecutar (partes 1 y 2 o parte 3):

```
cd Partes1y2
```

2. Necesitaremos tener instalado make, en caso contrario instalamos:

```
sudo apt install make
```

3. Ejecutamos make:

```
make
```

Esto compilará el programa según la definición del fichero makefile:

```
CC = gcc
OBJ = source/threads.o source/funciones.o

sim: source/sistema.c $(OBJ)
	$(CC) -o sim source/sistema.c $(OBJ)

funciones.o: source/funciones.c
	$(CC) -c source/funciones.c

threads.o: source/threads.c
	$(CC) -c source/threads.c

clean:
	rm -f $(OBJ)
```

4. Para eliminar los ficheros .o creados ejecutamos:

```
make clean
```

Ejecución

Si ejecutamos el programa veremos que nos pedirá los siguientes parámetros:

```
Uso:  ./sim <frecuencia_clock> <frecuencia_temp> <frecuencia_processGen>
<num_cpus> <num_cores> <num_hilos>
```

Explicación de los parámetros

- **<frecuencia_clock>:** Entero > 0. Esto indicará la frecuencia del reloj, es decir, la frecuencia de los ciclos que controlan el tiempo del sistema.
- **<frecuencia_temp>:** Entero > 0. Esto indicará la frecuencia del temporizador que llamará al scheduler, es decir, cada cuantos ciclos de reloj se ejecutará el scheduler.
- **<frecuencia_processGen>:** Entero > 0. Indicará la frecuencia con la que se le llame al proccess generator, es decir, la frecuencia con la que se generará un nuevo proceso.
- **<num_cpus>:** Entero > 0. Indicará el número de procesadores que tendrá la máquina que simularemos.
- **<num_cores>:** Entero > 0. Indicará el número de núcleos que tendrá cada procesador.
- **<num_hilos>:** Entero > 0. Indicará el número hilos que tendrá cada core.

Salida del código

Al ejecutarlo con los parámetros correspondientes, veremos que por cada pulso de reloj imprime por pantalla el estado del Scheduler/Dispatcher y del Process Generator, los procesos en ejecución junto al estado de cada proceso, y las colas con su contenido.

El programa imprime el código muy rápido, por lo que para analizarlo una solución es redirigir la salida a un fichero:

```
./sim 1 2 1 1 2 3 > ej.txt
```

La ejecución del programa es infinita, por lo que cuando queramos que pare debemos pulsar Ctrl+C.

Ejemplo

Ejecutamos el programa con los siguientes parámetros:

```
./sim 1 2 1 1 2 3 > ej.txt
```

La salida será la siguiente:

```
Scheduler: NO  //Indica que el scheduler no se ejecuta
Pg: 3175 Cola: 1  //Se crea el proceso 3175 y se le asigna la cola 1
-------------------------------------------------------
***PROCESOS EN EJECUCIÓN***        //Como no se ha ejecutado
CPU: 0, Core: 0, Thread: 0 --->    //el scheduler/dispatcher
CPU: 0, Core: 0, Thread: 1 --->    //el proceso no entra en ejecucción
CPU: 0, Core: 0, Thread: 2 --->
CPU: 0, Core: 1, Thread: 0 --->
CPU: 0, Core: 1, Thread: 1 --->
CPU: 0, Core: 1, Thread: 2 --->
-------------------------------------------------------
***COLAS***
#####Cola 1#####
PID: 3175, VIDA: 7, Q: 5   //Se indica que el proceso está en la cola 1
#####Cola 2#####           // y su vida y quantum
#####Cola 3#####
-------------------------------------------------------
Se ha llamado al Scheduler   //Se ejecuta el Scheduler/Dispatcher
Pg: 24367 Cola: 2            //Se crea proceso nuevo
-------------------------------------------------------
***PROCESOS EN EJECUCIÓN***
CPU: 0, Core: 0, Thread: 0 ---> PID: 3175, VIDA: 7, Q: 5, Cola: 1
CPU: 0, Core: 0, Thread: 1 --->   //El proceso entra en ejecución
CPU: 0, Core: 0, Thread: 2 --->
CPU: 0, Core: 1, Thread: 0 --->
CPU: 0, Core: 1, Thread: 1 --->
CPU: 0, Core: 1, Thread: 2 --->
-------------------------------------------------------
***COLAS***
#####Cola 1#####
#####Cola 2#####
PID: 24367, VIDA: 14, Q: 10  //El proceso nuevo se queda en la cola
#####Cola 3#####
-------------------------------------------------------
Pg: 13438 Cola: 1
Scheduler: NO   //No se ejecuta el scheduler/dispatcher
-------------------------------------------------------
***PROCESOS EN EJECUCIÓN***
CPU: 0, Core: 0, Thread: 0 ---> PID: 3175, VIDA: 6, Q: 4, Cola: 1
CPU: 0, Core: 0, Thread: 1 --->   //Disminuye la vida y quantum de
```

```
CPU: 0, Core: 0, Thread: 2 --->   //los procesos en ejecución
CPU: 0, Core: 1, Thread: 0 --->
CPU: 0, Core: 1, Thread: 1 --->
CPU: 0, Core: 1, Thread: 2 --->
------------------------------------------------------------
***COLAS***
#####Cola 1#####
PID: 13438, VIDA: 13, Q: 5
#####Cola 2#####
PID: 24367, VIDA: 14, Q: 10
#####Cola 3#####
------------------------------------------------------------
Pg: 3633 Cola: 2
Se ha llamado al Scheduler  //Mete a ejecución los procesos en las colas
------------------------------------------------------------
***PROCESOS EN EJECUCIÓN***
CPU: 0, Core: 0, Thread: 0 ---> PID: 3175, VIDA: 5, Q: 3, Cola: 1
CPU: 0, Core: 0, Thread: 1 ---> PID: 13438, VIDA: 13, Q: 5, Cola: 1
CPU: 0, Core: 0, Thread: 2 ---> PID: 24367, VIDA: 14, Q: 10, Cola: 2
CPU: 0, Core: 1, Thread: 0 ---> PID: 3633, VIDA: 3, Q: 10, Cola: 2
CPU: 0, Core: 1, Thread: 1 --->
CPU: 0, Core: 1, Thread: 2 --->
------------------------------------------------------------
***COLAS***
#####Cola 1#####
#####Cola 2#####
#####Cola 3#####
```

Estructura del código

El código está separado en dos directorios, el directorio “include” y el directorio “source”.

- **Include:** Contiene los ficheros .h con las cabeceras y estructuras.
- **Source:** Incluye el código (ficheros .c).

Ficheros relevantes:

- **structs.h:** Incluye las definiciones de las estructuras que se utilizarán a lo largo del programa.
- **sistema.c:** Contiene las inicializaciones de las variables y estructuras que se utilizarán. Además incluye la función main(), que se encargará de inicializar los hilos.
- **threads.c:** Contiene los hilos que se encargarán de simular el reloj, el Process Generator y el Scheduler/Dispatcher.
- **funciones.c:** Contiene las funciones auxiliares a las que llamarán los hilos.

Implementación

structs.h

Estructuras que se utilizarán a lo largo del programa.

Librerías:

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <time.h>
```

Se definen las librerías que utilizará el programa

PCB:

```
// Estructura de PCB
typedef struct PCB {
  int pid;
  int vida;
  int quantum;
  int cola;
  struct PCB* sig;
} PCB;
```

Estructura de PCB, con los atributos pid, vida, quantum, cola y un puntero al siguiente PCB.

cola:

```
// Estructura de Cola de Procesos
typedef struct {
  PCB* prim;
  PCB* ult;
  int cant;
  int quantum;
} cola;
```

Estructura cola, es donde se almacenarán los procesos que no están en ejecución. Habrá tres tipos, lo veremos más adelante.

process_queue:

```
// Estructura de Cola de Procesos
typedef struct {
  cola cola1;
  cola cola2;
  cola cola3;
} process_queue;
```

Estructura que almacena las tres colas.

machine:

```
// Estructura Machine
typedef struct {
  int cpus;
  int cores;
  int hilos;
} machine;
```

Estructura que representa la arquitectura de la máquina simulada.

sistema.c

Librerías:

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <time.h>
#include <semaphore.h>
#include "../include/structs.h"
```

Declaraciones de variables:

```
// Definimos las variables de sincronización
pthread_mutex_t mutex;
sem_t sem;
sem_t sem1;
sem_t sem2;
sem_t sem3;
```

```
pthread_cond_t cond;
pthread_cond_t cond1;
pthread_cond_t cond2;

// Definimos más variables
int done = 0;
int tenp_kont = 2;
int freq_c;
int mul_t;
int mul_p;
int cont_t;
int cont_p;
int prioridad;

//Definimos las colas
cola cola1;
cola cola2;
cola cola3;

//Definimos la cola de procesos
process_queue cola_procesos;

machine hardware;
PCB**** maquina; // Declaración de puntero para memoria dinámica

// Incluimos las cabeceras de los hilos
void* reloj(void* arg);
void* timer(void* arg);
void* timer1(void* arg);
void* process_gen(void* arg);
void* sche_dispa(void* arg);
```

Declaramos las variables que utilizaremos más adelante. Además incluimos las cabecerás de las funciones que utilizarán los hilos.

Función main:

```
int main(int argc, char* argv[]) {
```

Paso de parámetros:

```
  if (argc != 7) {
      printf("\nUso: %s <frecuencia_clock> <frecuencia_temp>
<frecuencia_processGen> <num_cpus> <num_cores> <num_hilos>\n", argv[0]);
      exit(1);
```

```
  }

  int freq_c = atoi(argv[1]);
  int freq_t = atoi(argv[2]);
  int freq_p = atoi(argv[3]);
  int cpus = atoi(argv[4]);
  int cores = atoi(argv[5]);
  int hilos = atoi(argv[6]);
  mul_t = freq_c * freq_t;
  mul_p = freq_c * freq_p;
  cont_t = 0;
  cont_p = 0;
```

Se hace un control de parámetros sencillo, y se le asignan los parámetros definidos a las variables de frecuencia y las variables del hardware de la máquina.
Además, se le asignan valores a las variables cont y mul, estas les servirán a los hilos par saber cuando tienen que ejecutarse.

Inicialización del hardware:

```
hardware.cpus = atoi(argv[4]);
hardware.cores = atoi(argv[5]);
hardware.hilos = atoi(argv[6]);

//Inicializamos la estructura machine
//maquina[hardware.cpus][hardware.cores][hardware.hilos];
// Inicializamos la estructura machine (arreglo tridimensional dinámico)
maquina = (PCB****)malloc(hardware.cpus * sizeof(PCB***));
for (int i = 0; i < hardware.cpus; i++) {
      maquina[i] = (PCB***)malloc(hardware.cores * sizeof(PCB**));
      for (int j = 0; j < hardware.cores; j++) {
            maquina[i][j] = (PCB**)malloc(hardware.hilos * sizeof(PCB*));
            for (int k = 0; k < hardware.hilos; k++) {
                  maquina[i][j][k] = NULL; // Inicializamos cada puntero a
NULL
            }
      }
      }
```

Inicializamos la estructura hardware con los valores que recibimos como parámetros que será la encargada de simular los procesadores. Además Inicializamos un array tridimensional de PCBs donde se almacenarán los PCBs en ejecución. El array tiene la siguiente forma -> maquina[cpus][cores][hilos].

Inicialización de las colas

```
  //Asignamos los valores a las colas
  cola1.quantum=5;
  cola1.cant=0;
  cola2.quantum=10;
  cola2.cant=0;
  cola3.quantum=-1; // -1 para indicar que es infinito, ya que la ultima
cola es FCFS
  cola3.cant=0;
  cola_procesos.cola1 = cola1;
  cola_procesos.cola2 = cola2;
  cola_procesos.cola3 = cola3;
```

Inicializamos las colas con el quantum que tendrá cada una y la cantidad de elementos (0 porque empiezan vacías).
Las colas siguen una planificación con degradación paulatina, es decir, siempre entrarán los procesos a ejecución en el orden de las colas. Los de la primera cola tienen un quantum de 5, cuando lo consumen pasan a la segunda, donde tienen un quantum de 10, y si lo vuelven a consumir, pasan a la tercera, donde recibirán un quantum ilimitado (FCFS).

Creación e inicialización de los hilos:

```
//Iniciamos las variables de los hilos
  pthread_mutex_init(&mutex, NULL);
  sem_init(&sem, 0, 0);
  sem_init(&sem1, 0, 0);
  sem_init(&sem2, 0, 0);
  sem_init(&sem3, 0, 0);
  pthread_cond_init(&cond, NULL);
  pthread_cond_init(&cond1, NULL);
  pthread_cond_init(&cond2, NULL);

  // Creamos los hilos
  pthread_t th_clock;
  pthread_t th_timer;
  pthread_t th_timer1;
  pthread_t th_sche_dispa;
  pthread_t th_process_gen;
  pthread_create(&th_clock, NULL, reloj, NULL);
  pthread_create(&th_timer, NULL, timer, NULL);
  pthread_create(&th_timer1, NULL, timer1, NULL);
  pthread_create(&th_sche_dispa, NULL, sche_dispa, NULL);
  pthread_create(&th_process_gen, NULL, process_gen, NULL);

  // Esperar a que los hilos se junten
```

```
  pthread_join(th_clock, NULL);
  pthread_join(th_timer, NULL);
  pthread_join(th_timer1, NULL);
  pthread_join(th_sche_dispa, NULL);
  pthread_join(th_process_gen, NULL);
}
```

Inicializamos las variables de sincronización y creamos los hilos. A partir de este momento el simulador de la estructura creada empezará a correr.
Después esperamos a que los hilos se junten, es simplemente una formalidad ya que esto nunca sucederá, pués ejecutan funciones infinitas.

threads.c

Incluye las funciones que ejecutan los hilos.

Librerías:

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <time.h>
#include <semaphore.h>
#include "../include/structs.h"
#include "../include/threads.h"
```

void* reloj(void* arg):

```
// Función clock
void* reloj(void* arg) {
  while(1){
      pthread_mutex_lock(&mutex);
      while(done<tenp_kont){
      pthread_cond_wait(&cond1, &mutex);
      }
      done=0;

      imprimirProcesos();
      imprimirColas();
      moverEstructura();

      pthread_cond_broadcast(&cond2);
```

```
        pthread_mutex_unlock(&mutex);
    }
}
```

Es el motor del simulador, se encarga de mover la máquina. Simula el reloj de los procesadores, es decir, cada vez que se ejecuta significa un ciclo de ejecución de cada hilo hardware. Además de hacer avanzar la estructura, señala a los timers para que estos funcionen. Esto lo hace mediante las variables de sincronización de hilos (cond y mutex).

void* timer(void* arg):

```
/// Funcion timer
void* timer(void* arg) {
  pthread_mutex_lock(&mutex);
  while(1){
      cont_t++;
      if (cont_t==mul_t){
      sem_post(&sem); //señal para que se ejecute sche_dispa
      sem_wait(&sem1);
      cont_t = 0;
      }
      else{
      printf("Scheduler: NO\n");
      fflush(stdout);
      }

      done++;
      pthread_cond_signal(&cond1);
      pthread_cond_wait(&cond2,&mutex);
  }
}
```

Tiene en cuenta la frecuencias que le hemos pasado al programa como parámetros para dar paso o no al hilo que hace de Scheduler/Dispatcher. Esto lo hace mediante la utilización de semáforos.

void* timer1(void* arg):

```
// Funcion Process Generator
void* timer1(void* arg) {
  pthread_mutex_lock(&mutex);
  while(1){
      cont_p++;
```

```
      if (cont_p==mul_p){
      sem_post(&sem2); //señal para que se ejecute process_gen
      sem_wait(&sem3);
      cont_p = 0;
      }
      else{
      printf("Pg: NO\n");
      fflush(stdout);
      }
      done++;
      pthread_cond_signal(&cond1);
      pthread_cond_wait(&cond2,&mutex);
  }
}
```

Igual que el anterior, pero para dar paso al hilo del Process Generator.

void* sche_dispa(void* arg):

```
void* sche_dispa(void* arg) {
  while(1){
      sem_wait(&sem); //se queda esperando a timer
      printf("Se ha llamado al Scheduler\n");
      fflush(stdout);

      sacarDeEstructura(); //Elimina los procesos acabados de la
estrcutura y los a los que se les ha acabado el quantum los manda a la
siguiente cola.
      asignarEstructura(); //Mueve los PCBs de las colas a la
estructura.

      sem_post(&sem1);
  }
}
```

Hilo que se encarga de planificar y realizar los cambios de contexto de los procesos. Para ello utiliza las funciones sacarDeEstructura() y asignarEstructura().

void* process_gen(void* arg):

```
void* process_gen(void* arg) {
  while(1){
      sem_wait(&sem2); //se queda esperando a timer
      //PCB proc;
      PCB* proc = (PCB*)malloc(sizeof(PCB));
```

```
      proc->pid = (rand() % 32668) + 100; //Para simular un pid
aleatorio
      proc->vida = (rand() % 20) + 1; //Para simular un tiempo de vida
aleatorio
      prioridad = (rand() % 3) + 1; //Para simular un nivel de prioridad
aleatorio del 1 al 3
      //
      anadirACola(proc, prioridad);
      sem_post(&sem3);
  }
}
```

Hilo que se encarga de generar procesos de manera aleatoria. Después los añade a la cola que les haya tocado con la función anadirACola(proc, prioridad).

funciones.c

Librerías:

```
#include "../include/structs.h"
#include "../include/funciones.h"
```

void anadirACola(PCB* proc, int prioridad):

```
void anadirACola(PCB* proc, int prioridad) {
  if (prioridad == 1) {
      if (cola_procesos.cola1.cant == 0){
      proc->sig = NULL;
      proc->cola = 1;
      proc->quantum = cola_procesos.cola1.quantum;
      cola_procesos.cola1.prim = proc;
      cola_procesos.cola1.ult = proc;
      cola_procesos.cola1.cant = 1;
      }
      else {
      cola_procesos.cola1.ult->sig = proc;
      cola_procesos.cola1.ult = proc;
      proc->sig = NULL;
      proc->cola = 1;
      proc->quantum = cola_procesos.cola1.quantum;
      cola_procesos.cola1.cant += 1;
```

```
        }
        printf("Pg: %d Cola: 1\n", cola_procesos.cola1.ult->pid);
        fflush(stdout);
    }
    else if (prioridad == 2) {
        if (cola_procesos.cola2.cant == 0){
        proc->sig = NULL;
        proc->cola = 2;
        proc->quantum = cola_procesos.cola2.quantum;
        cola_procesos.cola2.prim = proc;
        cola_procesos.cola2.ult = proc;
        cola_procesos.cola2.cant = 1;
        }
        else {
        cola_procesos.cola2.ult->sig = proc;
        cola_procesos.cola2.ult = proc;
        proc->sig = NULL;
        proc->cola = 2;
        proc->quantum = cola_procesos.cola2.quantum;
        cola_procesos.cola2.cant += 1;
        }
        printf("Pg: %d Cola: 2\n", cola_procesos.cola2.ult->pid);
        fflush(stdout);
    }
    else if (prioridad == 3) {
        if (cola_procesos.cola3.cant == 0){
        proc->sig = NULL;
        proc->cola = 3;
        proc->quantum = cola_procesos.cola3.quantum;
        cola_procesos.cola3.prim = proc;
        cola_procesos.cola3.ult = proc;
        cola_procesos.cola3.cant = 1;
        }
        else {
        cola_procesos.cola3.ult->sig = proc;
        cola_procesos.cola3.ult = proc;
        proc->sig = NULL;
        proc->cola = 3;
        proc->quantum = cola_procesos.cola3.quantum;
        cola_procesos.cola3.cant += 1;
        }
        printf("Pg: %d Cola: 3\n", cola_procesos.cola3.ult->pid);
        fflush(stdout);
    }
}
```

Esta función agrega un proceso (proc) a una de las tres colas de prioridad, dependiendo de la prioridad especificada (prioridad). Si la cola está vacía, inicializa los punteros de inicio y fin de la cola. Si ya hay procesos, agrega el nuevo proceso al final, después, a22signa el quantum correspondiente y actualiza la cantidad de elementos en la cola.

void imprimirColas():

```
void imprimirColas(){

printf("\n-------------------------------------------------------------\n")
;
  printf("***COLAS***");
  PCB* proc;

  printf("\n#####Cola 1#####\n");
  if (cola_procesos.cola1.cant != 0){
      proc = cola_procesos.cola1.prim;
      printf("PID: %d, VIDA: %d, Q: %d\n", proc->pid, proc->vida,
proc->quantum);
      while (proc->sig != NULL){
      proc = proc->sig;
      printf("PID: %d, VIDA: %d, Q: %d\n", proc->pid, proc->vida,
proc->quantum);
      }
  }

  printf("#####Cola 2#####\n");
  if (cola_procesos.cola2.cant != 0){
      proc = cola_procesos.cola2.prim;
      printf("PID: %d, VIDA: %d, Q: %d\n", proc->pid, proc->vida,
proc->quantum);
      while (proc->sig != NULL){
      proc = proc->sig;
      printf("PID: %d, VIDA: %d, Q: %d\n", proc->pid, proc->vida,
proc->quantum);
      }
  }

  printf("#####Cola 3#####\n");
  if (cola_procesos.cola3.cant != 0){
      proc = cola_procesos.cola3.prim;
      printf("PID: %d, VIDA: %d, Q: %d\n", proc->pid, proc->vida,
proc->quantum);
      while (proc->sig != NULL){
      proc = proc->sig;
      printf("PID: %d, VIDA: %d, Q: %d\n", proc->pid, proc->vida,
```

```
proc->quantum);
      }
  }


printf("------------------------------------------------------------\n");
  fflush(stdout);
}
```

Esta función imprime el contenido de las tres colas de procesos en la consola. Para cada cola, muestra los procesos con su PID, vida y quantum.

PCB* sacarPCBDeCola():

```
PCB* sacarPCBDeCola(){
  PCB* proceso;
  if (cola_procesos.cola1.cant > 0){
      proceso = cola_procesos.cola1.prim;
      cola_procesos.cola1.prim = proceso->sig;
      cola_procesos.cola1.cant--;
      proceso->sig = NULL;
  }
  else if (cola_procesos.cola2.cant > 0){
      proceso = cola_procesos.cola2.prim;
      cola_procesos.cola2.prim = proceso->sig;
      cola_procesos.cola2.cant--;
      proceso->sig = NULL;
  }
  else if (cola_procesos.cola3.cant > 0){
      proceso = cola_procesos.cola3.prim;
      cola_procesos.cola3.prim = proceso->sig;
      cola_procesos.cola3.cant--;
      proceso->sig = NULL;
  }
  else {
      proceso = NULL;
  }
  return proceso;
}
```

Esta función extrae y devuelve el primer proceso (PCB) disponible de las colas, comenzando por la cola de mayor prioridad (1). Si una cola está vacía, pasa a la siguiente. Si no hay procesos en ninguna cola, devuelve NULL.

void asignarEstructura():

```
void asignarEstructura(){
  for (int cpu = 0; cpu < hardware.cpus; cpu++){
      for (int core = 0; core < hardware.cores; core++){
      for (int hilo = 0; hilo < hardware.hilos; hilo++){
      PCB* proceso = maquina[cpu][core][hilo];
      if (proceso == NULL){
            maquina[cpu][core][hilo] = sacarPCBDeCola();
      }
      }
      }
  }
}
```

Recorre todas las CPUs, núcleos y hilos del sistema representados en la estructura maquina. Si un hilo no tiene un proceso asignado, extrae uno de las colas de procesos (utilizando sacarPCBDeCola()) y lo asigna al hilo correspondiente.

void sacarDeEstructura():

```
void sacarDeEstructura(){
  for (int cpu = 0; cpu < hardware.cpus; cpu++){
      for (int core = 0; core < hardware.cores; core++){
      for (int hilo = 0; hilo < hardware.hilos; hilo++){
      PCB* proceso = maquina[cpu][core][hilo];
      if (proceso != NULL){
            if (proceso->vida <= 0){
            free(proceso);
            maquina[cpu][core][hilo] = NULL;
            }
            else if ((proceso->quantum <= 0) && (proceso->cola == 1)){
//Si se le acaba el quantum vuelta a la cola
            maquina[cpu][core][hilo] = NULL;
            anadirACola(proceso, 2);
            }
            else if ((proceso->quantum <= 0) && (proceso->cola == 2)){
//Si se le acaba el quantum vuelta a la cola
            maquina[cpu][core][hilo] = NULL;
            anadirACola(proceso, 3);
            }
      }
      }
      }
  }
}
```

Elimina o reasigna procesos en ejecución de los hilos del sistema:

- Si la vida del proceso ha terminado, lo libera de la memoria.
- Si el quantum de un proceso en cola 1 o 2 se agota, lo reasigna a la siguiente cola.

Este método se utiliza para asegurar que los procesos sean gestionados correctamente según su vida y quantum.

void moverEstructura():

```
void moverEstructura(){
  for (int cpu = 0; cpu < hardware.cpus; cpu++){
      for (int core = 0; core < hardware.cores; core++){
      for (int hilo = 0; hilo < hardware.hilos; hilo++){
      PCB* proceso = maquina[cpu][core][hilo];
      if (proceso != NULL){
            if (proceso->vida != 0 ){
            if (proceso->cola == 3) {
            proceso->quantum = -1;
            }
            else {
            proceso->quantum--;
            }
            proceso->vida--; //Los borra el dispatcher
            maquina[cpu][core][hilo] = proceso;
            }
      }
      }
      }
  }
}
```

Recorre la estructura de ejecución y actualiza el estado de los procesos. Disminuye el quantum y la vida de cada proceso. Los procesos de la cola 3 no reducen su quantum porque es infinito.

void imprimirProcesos():

```
void imprimirProcesos(){

printf("------------------------------------------------------------\n");
  printf("***PROCESOS EN EJECUCIÓN***");
  for (int cpu = 0; cpu < hardware.cpus; cpu++){
      for (int core = 0; core < hardware.cores; core++){
      for (int hilo = 0; hilo < hardware.hilos; hilo++){
      printf("\nCPU: %d, Core: %d, Thread: %d --->", cpu, core, hilo);
```

```
      PCB* proceso = maquina[cpu][core][hilo];
      if (proceso != NULL){
            printf(" PID: %d, VIDA: %d, Q: %d, Cola: %d", proceso->pid,
proceso->vida, proceso->quantum, proceso->cola);
      }
      }
      }
  }
  fflush(stdout);
}
```

lmprime los procesos en ejecución para cada CPU, núcleo e hilo del sistema. Muestra el PID, la vida restante,el quantum y la cola de prioridad.