Ampliación Sistemas Operativos

Práctica 1: Minikernel

Alumno: Santiago Ramos Gómez

Titulación: Ingeniería de computadores

Universidad: Rey Juan Carlos

Campus: Móstoles

Curso: 2021-2022

Introducción

En esta primera práctica se debían añadir unas funcionalidades a un código facilitado.

Estas son:

- Sleep
- Round Robin
- Mutex

A lo largo del trabajo se irá explicando paso a paso el código utilizado en cada funcionalidad, así como los ficheros secundarios que han sido modificados.

Sleep

La función “Sleep” o Dormir se centra en tomar un proceso y bloquearlo durante un tiempo especificado. Como parámetros recibe únicamente una cantidad de segundos, los cuales deben ser multiplicados por una constante ya definida “**TICKS”**. Esto es debido a que la frecuencia de reloj establecida es de 100 (ticks/segundo), como bien se explica en **const.h.**

```
int dormir(unsigned int segundos){
    segundos = (unsigned int)leer_registro(1);
    int nivel;
    nivel=fijar_nivel_int(NIVEL_3);

    p_proc_actual->estado = BLOQUEADO;
    p_proc_actual->segundos =  segundos * TICK;

    eliminar_elem(&lista_listos, p_proc_actual);
    insertar_ultimo(&lista_bloqueados,p_proc_actual);


    BCP *p_proc_dormir = p_proc_actual;
    p_proc_actual = planificador();
    cambio_contexto(&(p_proc_dormir->contexto_regs), &(p_proc_actual->contexto_regs));
    fijar_nivel_int(nivel);
    return 0;
}
```

Lo primero que se realiza es utilizar la función “leer_registro(1)”, ya que de otra manera obtenía error y no recibía bien los parámetros.

A continuación, se declara el nivel de interrupciones a nivel 3, siendo el más alto de todos. Esto se realiza para que se tenga la máxima prioridad posible y no se vea interrumpido el proceso por otros de menor importancia en este momento.

Tras esto, se toma el proceso actual (proceso con el que se está trabajando en este momento) definido como ”p_proc_actual”, este es un dato que viene definido por la estructura BCP, la cual tiene los siguientes atributos mostrados en la imagen.

```
typedef struct BCP_t {
        int id;                         /* ident. del proceso */
        int estado;                     /* TERMINADO|LISTO|EJECUCION|BLOQUEADO*/
        contexto_t contexto_regs;       /* copia de regs. de UCP */
        void * pila;                    /* dir. inicial de la pila */
        BCPptr siguiente;               /* puntero a otro BCP */
        void *info_mem;                 /* descriptor del mapa de memoria */
        //nuevos
        unsigned int segundos;          //SLEEP

        int ticks_rodaja;               //ROUNDROBIN

        mutex* descriptores[NUM_MUT_PROC]; //MUTEX
        int num_descriptores;          //MUTEX

} BCP;
```

Se accede a los atributos estado y segundos; en uno para establecer que desde ese momento se encuentra bloqueado, y en el otro para definir la cantidad de tiempo que debe estar bloqueado según la frecuencia de reloj.

Ahora se debe eliminar este proceso de la lista de procesos listos y moverlo a la lista de procesos en estado bloqueado, es por ello que se utilizan las siguientes dos funciones.

Por último, se genera una variable “p_proc_dormir” y obtiene los datos del proceso actual, tras tener estos datos guardados, se procede a llamar al planificador para obtener un nuevo proceso, el cual será asignado a la variable p_proc_actual.

Por último, se realiza el cambio de contexto para guardar los registros de la UCP mientras está bloqueado y se retira el nivel 3 de interrupción que previamente se había almacenado.

Cabe resaltar que Sleep no termina aquí, ya que falta el verdadero nucleo de la función, el cual se encuentra en la función “int_reloj()”.

```
static void int_reloj(){
//printk("-> TRATANDO INT. DE RELOJ\n");

//PROCESOS ROUND ROBIN

p_proc_actual->ticks_rodaja--;
if(p_proc_actual->ticks_rodaja<=0){
    activar_int_SW();
}

//PROCESOS DORMIDOS

    BCP *p_proc_dormir = lista_bloqueados.primero;
    while(p_proc_dormir != NULL){
        p_proc_dormir->segundos--;
        if(p_proc_dormir->segundos <= 0){


            int nivel=fijar_nivel_int(NIVEL_3);

            p_proc_dormir->estado = LISTO;
            p_proc_dormir->segundos = 0;

            eliminar_elem(&lista_bloqueados, p_proc_dormir);
            insertar_ultimo(&lista_listos, p_proc_dormir);
            fijar_nivel_int(nivel);
        }
        p_proc_dormir = p_proc_dormir->siguiente;
    }

    return;
}
```

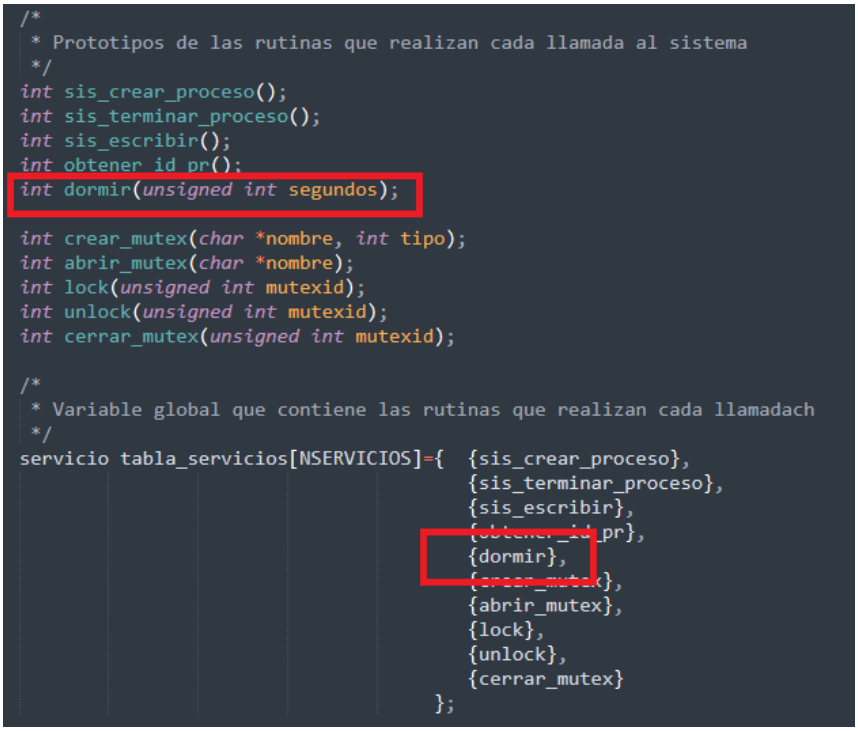
En este apartado tan solo se comentará lo que procede después del comentario "//PROCESOS DORMIDOS".

Comienza con un puntero de tipo BCP llamado p_proc_dormir, cuya función es similar a la de un iterador. Toma el primer valor de la lista de los procesos bloqueados, si no es NULL es porque hay alguno, por lo que entra al bucle while, en el cual se le restara una unidad de tiempo cada vez que se ejecute, si ha llegado a 0 o inferior (por asegurar, ya que en las primeras pruebas salían números negativos y al final decidí dejarlo así) quiere decir que no necesita estar más tiempo bloqueado, por lo que se procede a cambiar su estado a LISTO y resetear su contador de segundos a 0, todo esto habiendo marcado previamente el nivel de interrupción 3, asociado al reloj.

Finalmente se libera el proceso de la lista de bloqueados y se añade al final de lista de procesos listos, se quita el nivel 3 de interrupción y se comprueba si hay algún proceso más que esté bloqueado, a partir del atributo siguiente.

Los archivos modificados para el funcionamiento de esta función han sido:

- **Kernel.c** para añadir y completar las funciones previas.
- **Kernel.h** para añadir el atributo "segundos" a las estructuras de tipo BCP, así como modificar la tabla de servicios y su prototipo.



```
/*
 * Prototipos de las rutinas que realizan cada llamada al sistema
 */
int sis_crear_proceso();
int sis_terminar_proceso();
int sis_escribir();
int obtener_id_pr();
int dormir(unsigned int segundos);

int crear_mutex(char *nombre, int tipo);
int abrir_mutex(char *nombre);
int lock(unsigned int mutexid);
int unlock(unsigned int mutexid);
int cerrar_mutex(unsigned int mutexid);

/*
 * Variable global que contiene las rutinas que realizan cada llamadach
 */
servicio tabla_servicios[NSERVICIOS]={  {sis_crear_proceso},
                                        {sis_terminar_proceso},
                                        {sis_escribir},
                                        {obtener_id_pr},
                                        {dormir},
                                        {crear_mutex},
                                        {abrir_mutex},
                                        {lock},
                                        {unlock},
                                        {cerrar_mutex}
                                     };
```

- **Servicios.h** para añadir la función: "int dormir(unsigned int segundos)"
- **Llamsis.h** para modificar el NSERVICIOS y añadir DORMIR 4
- **Serv.c** para añadir la llamada al sistema en la interfaz.

DORMIR es su nombre en **Llamsis.h,** 1 es el número de argumentos a pasar, y segundos el nombre del argumento.

```
int dormir(unsigned int segundos){
   return llamsis(DORMIR, 1, (long)segundos);
}
```

Round-Robin

En segundo lugar, se va a explicar el algoritmo Round-Robin, ya que el tercer apartado es extremadamente extenso.

Para comenzar se propone utilizar la constante TICKS_POR_RODAJA disponible en const.h, los cuales equivalen a 10. Estos ticks siguen el mismo funcionamiento que en el caso del Sleep, medidas de tiempo, y serán implementados en la estructura BCP como el atributo “ticks rodaja”.

Al asignar un proceso en la función “planificador ()” se le concederán siempre estos ticks.

```
static BCP * planificador(){
    while (lista_listos.primero==NULL)
        espera_int();
    lista_listos.primero->ticks_rodaja=TICKS_POR_RODAJA;
    return lista_listos.primero;
}
```

En la función int_reloj() hay un apartado dedicado a contar estos ticks para poder cambiar entre procesos la prioridad. Sencillamente si los ticks_rodaja del proceso llegan a 0 se activa la interrupción software.

```
static void int_reloj(){
//printk("-> TRATANDO INT. DE RELOJ\n");

//PROCESOS ROUND ROBIN

p_proc_actual->ticks_rodaja--;
if(p_proc_actual->ticks_rodaja<=0){
    activar_int_SW();
}
```

En esta interrupción software se planteó usar el nivel 1 de interrupción, que es la adecuada para software según las constantes, pero ya que se buscaba que ante cualquier situación fuera específicamente este proceso el que se expulsara ya que el módulo HAL carecía de una funcionalidad, se decidió subirlo al máximo para prevenir cualquier riesgo.

Dejando eso atrás, el funcionamiento es el mismo que la parte final del Sleep. Se da un nivel de interrupción, se genera la variable proceso_rr y se le asignan los datos del proceso actual, se elimina de la primera posición de lista_listos y se añade al final de la misma (ya que así funciona el algoritmo Round Robin). Por último, el proceso actual es el siguiente de la lista_listos, donde en esa llamada al planificador se le está concediendo los nuevos ticks de rodaja. Para finalizar se realiza un cambio de contexto y se restaura el nivel.

```
static void int_sw(){

    printk("-> TRATANDO INT. SW\n");
    //int nivel = fijar_nivel_int(NIVEL_1);
    int nivel = fijar_nivel_int(NIVEL_3);
    BCP* proceso_rr=p_proc_actual;
    eliminar_primero(&lista_listos);
    insertar_ultimo(&lista_listos, proceso_rr);
    p_proc_actual=planificador();

    cambio_contexto(&(proceso_rr->contexto_regs), &(p_proc_actual->contexto_regs));
    fijar_nivel_int(nivel);


    return;
}
```

Los únicos archivos modificados son los comentados a lo largo del apartado, kernel.c y kernel.h.

Mutex

Tercer y último apartado de la práctica que se divide en 5 funciones.

Es importante constatar que este apartado es muy extenso y por ello pienso que en mi código puede haber algún error o fallo en cuanto a contenido.

Para poder dar entrada a los mutex, primero se deben hacer unos cuantos ajustes en el fichero **kernel.h,** en el cual añade una nueva estructura, mutex.

```
#define NO_RECURSIVO 0
#define RECURSIVO 1

#define OCUPADO 0
#define LIBRE 1


typedef struct{

        char* nombre[MAX_NOM_MUT];
        int estado;  //LISTO OCUPADO
        int tipo;//RECURSIVO NO RECURSIVO
        int id_propietario;
        int num_veces_lock;

        lista_BCPs procesos_espera; //Procesos bloqueados porque no tenian el Mutex
}mutex;

int total_mutex;
mutex lista_mutex[NUM_MUT];
```

Esta estructura tiene como atributos un nombre de una longitud especifica, un estado que representa si está OCUPADO o LISTO, un tipo el cual puede ser RECURSIVO o NO RECURSIVO, un id_propietario, el cual vendrá definido por un proceso y num_veces_lock, que guarda el número de veces que se ha bloqueado; además, hay una lista de BCPs procesos_espera que almacenará a ciertos procesos que interactúen con el mutex en cierto momento.

Adicionalmente se tienen las variables de tipo define “RECURSIVO, NO_RECURSIVO, OCUPADO, LIBRE”, un contador de mutex, total_mutex y dos listas adicionales, una de tipo mutex en la cual se guardan todos los mutex y lista_bloqueados_mutex, en la cual se guardarán procesos circunstancialmente.

```
lista_BCPs lista_listos= {NULL, NULL};
lista_BCPs lista_bloqueados= {NULL, NULL};
lista_BCPs lista_bloqueados_mutex= {NULL, NULL};
```

También se encuentran dos nuevos atributos en la estructura BCP, siendo un array de tipo mutex llamado descriptores de tamaño definido y un contador de número de descriptores.

```
typedef struct BCP_t {
        int id;                         /* ident. del proceso */
        int estado;                     /* TERMINADO|LISTO|EJECUCION|BLOQUEADO*/
        contexto_t contexto_regs;       /* copia de regs. de UCP */
        void * pila;                    /* dir. inicial de la pila */
        BCPptr siguiente;               /* puntero a otro BCP */
        void *info_mem;                 /* descriptor del mapa de memoria */
        //nuevos
        unsigned int segundos;          //SLEEP

        int ticks_rodaja;               //ROUNDROBIN

        mutex* descriptores[NUM_MUT_PROC]; //MUTEX
        int num_descriptores;          //MUTEX

} BCP;
```

A continuación, se explicarán las 5 funciones que conforman mutex.

Crear_mutex( ):

```
int crear_mutex(char *nombre, int tipo){

    nombre = (char*)leer_registro(1);
    if(strlen(nombre)>MAX_NOM_MUT){printk("Nombre demasiado largo\n");return -1;} //nombre demasiado largo
    tipo = (int)leer_registro(2);

    //En caso de que el nombre sea valido, se debe comprobar si exite ya un mutex con el mismo nombre
    if(total_mutex>0){
        for(int i =0; i<total_mutex;i++){
            if(strcmp(lista_mutex[i]->nombre, nombre == 0)){
                printk("Nombre igual a otro\n");
                return -1;
            }
        }
    }

    int descriptor = descriptores_libres();
    if(descriptor<0){printk("No hay descriptores libres\n");return -1;}

    int mutex = mutex_libres();
    if(mutex<0){
    printk("No hay mutex libres, se procede a bloquear el proceso.\n");

    int nivel = fijar_nivel_int(NIVEL_3);
    BCP* p_proc_bloquear = p_proc_actual;
    p_proc_bloquear->estado = BLOQUEADO;
    eliminar_primero(&lista_listos);
    insertar_ultimo(&lista_bloqueados_mutex, p_proc_bloquear);

    p_proc_actual = planificador();
    cambio_contexto(&(p_proc_bloquear->contexto_regs), &(p_proc_actual->contexto_regs));
    fijar_nivel_int(nivel);

    }else{

        p_proc_actual->descriptores[descriptor] = &lista_mutex[mutex];
        p_proc_actual->num_descriptores++;
        lista_mutex[mutex].nombre=nombre;
        lista_mutex[mutex].estado=OCUPADO;
        lista_mutex[mutex].tipo=tipo;
        lista_mutex[mutex].num_veces_lock=0;

        total_mutex++;

        }

    return descriptor;
}
```

Esta función recibe dos parámetros, un nombre y un tipo, los cuales se guardan en dos variables. Al comienzo del todo se hace una comprobación con la longitud del nombre para asegurar que no se pasa del límite. Adicionalmente se comprueba en el bloque “if” si ese nombre ya existe.

```
//FUNCIONES AUXILIARES MUTEX

int descriptores_libres(){
        for(int i=0;i<NUM_MUT_PROC; i++){
        if(p_proc_actual->array_descriptores[i]==NULL)
            return i;
    }
    return -1;
}

int mutex_libres(){
            for(int i=0;i<NUM_MUT; i++){
                if(lista_mutex[i]->estado==1){ //1 = LIBRE //if(lista_mutex[i].estado==LIBRE)
                    return i;
                }
    return -1;

}
```

Tras esto se añaden dos nuevas variables al código descriptor y mutex.

Descripto por su parte comprueba si el proceso actual tiene algún hueco disponible en su array de descriptores, si lo hay devuelve su posición.

Mutex por otro lado comprueba si de la lista general de mutex hay alguno que esté disponible, si lo hay, devuelve su posición. En caso de no haber ningún hueco disponible, se procede a bloquear al proceso que ha solicitado crear un mutex y una vez más como en todas las funciones anteriores se sigue el mismo procedimiento para bloquear a un proceso. En caso de que SÍ existiera un hueco, se añade al array de descriptores la referencia a la posición del mutex que se va a crear, se aumenta su contador de descriptores y el mutex recibe sus atributos nombre, estado, tipo y num_veces_lock según lo especificado. Para finalizar se aumenta el contador total_mutex y se devuelve la posición del descriptor que guarda dicho mutex.

Abrir_mutex( ):

```
int abrir_mutex(char *nombre){
    nombre = (char *)leer_registro(1);

    if(existe_nombre_mutex(nombre)<0)return-1;

    int descriptor = descriptores_libres();
    if(descriptor<0)return -1;
    int mutex = mutex_libres();
    if(mutex<0)return -1;

    p_proc_actual->descriptores[descriptor] = &lista_mutex[mutex];
    p_proc_actual->num_descriptores++;

    return descriptor;
}
```

Esta función recibe únicamente un parámetro, el nombre del mutex que se quiere abrir.

Se comprueba a partir de una función auxiliar “existe_nombre_mutex” si existe el nombre del mutex (parecido al apartado anterior).

```
int existe_nombre_mutex(char* nombre){
    if(total_mutex>0){
        for(int i =0; i<total_mutex;i++){
            if(strcmp(lista_mutex[i].nombre, nombre == 0))
                return 1;
        }
    }else{return -1}
}
```

Si existe el nombre se comprueba que el proceso actual tenga espacio en el array de descriptores, así como comprobar que el mutex no esté ocupado. SI ambas condiciones se cumplen, se añade el mutex a la posición del array de descriptores correspondiente y se aumenta el número de descriptores, por último, se devuelve la posición del descriptor en la cual podemos obtener el mutex.

Aunque el nombre pueda confundir, sencillamente es añadir un mutex a la lista de descriptores de un proceso.

Lock( ):

Esta función recibe un parámetro, y es el mismo que recibirán las dos siguientes funciones. Se podía interpretar de varias maneras, pero yo lo interpreté como la posición de un descriptor en el array de estos; pero también estaba la opción de que fuera la posición en la lista global de mutex, pero no me parecía demasiado razonable.

```
int lock(unsigned int mutexid){
    mutexid = (unsigned int)leer_registro(1);
    //comprobamos si nuetro proceso contiene ese mutex
    if(p_proc_actual->descriptores[mutexid]==NULL)return -1;

    //si no ha sido nunca bloqueado
    if(p_proc_actual->descriptores[mutexid]->num_veces_lock==0){
        p_proc_actual->descriptores[mutexid]->id_propietario=p_proc_actual->id;
        p_proc_actual->descriptores[mutexid]->num_veces_lock++;
    }else{

        //si ya ha sido bloqueado mas veces
        if(p_proc_actual->descriptores[mutexid]->tipo==NO_RECURSIVO){ //NO ES RECURSIVO

            if(p_proc_actual->descriptores[mutexid]->id_propietario==p_proc_actual->id)return -1; //interbloqueo

        //BLOQUEAR PROCESO
            int nivel = fijar_nivel_int(NIVEL_3);
            p_proc_actual->estado=BLOQUEADO;
            eliminar_primero(&lista_listos);
            insertar_ultimo(&(proc_actual->descriptores[mutexid]->procesos_espera), p_proc_actual);

            BCP* p_proc_bloquear =p_proc_actual;
            p_proc_actual=planificador();

            cambio_contexto(&(p_proc_bloquear->contexto_regs), &(p_proc_actual->contexto_regs));
            fijar_nivel_int(nivel);

        }else{ //ES RECURSIVO

            if(p_proc_actual->descriptores[mutexid]->id_propietario==p_proc_actual->id){
                p_proc_actual->descriptores[mutexid]->num_veces_lock++;

            }else{
                //si es recursivo y no es el propietario y hay mas de un "locked", se bloquea el proceso.
                int nivel = fijar_nivel_int(NIVEL_3);
                p_proc_actual->estado=BLOQUEADO;
                eliminar_primero(&lista_listos);
                insertar_ultimo(&(proc_actual->descriptores[mutexid]->procesos_espera), p_proc_actual);

                BCP* p_proc_bloquear =p_proc_actual;
                p_proc_actual=planificador();

                cambio_contexto(&(p_proc_bloquear->contexto_regs), &(p_proc_actual->contexto_regs));
                fijar_nivel_int(nivel);
            }
        }
    }
    return 0;
}
```

En primer lugar, se comprueba si en la posición del descriptor dado en el array de descriptores del proceso actual es NULL o tiene algún mutex asociado.

En caso de no ser NULL, se empiezan a ramificar las opciones, teniendo que comprobar si es la primera vez que se bloquea, si es así se asigna al proceso actual como proietario del mutex y se incrementa el contador del mutex.

Si esta no fuera la primera vez, se debería comprobar si el mutex solicitado es del tipo RECURSIVO o NO_RECURSIVO.

Si es NO_RECURSIVO se debe comprobar que el proceso que solicita el lock NO es el propietario del mutex, ya que si fuera el mismo proceso se daría un caso de interbloqueo. Por lo cual, si es un proceso distinto, se fija el máximo nivel de interrupción para no ser interrumpido en el proceso y se añade al final de la lista de espera del mutex solicitado al proceso en acción, tras esto se obtiene un nuevo proceso, cambio de contexto y se recupera el nivel de interrupción.

Por otra parte, si es del tipo RECURSIVO, el propietario si puede bloquearlo más de una vez, incrementando así el contador del mutex, en caso contrario, si el proceso no es propietario del mutex se realizan los mismos pasos que en el NO_RECURSVO, se añade a la lista de espera del mutex.

Unlock( ):

Esta función obtiene nuevamente el parámetro mutexid y casi idéntica a la anterior, pero restando contadores.

```
int unlock(unsigned int mutexid){

mutexid = (unsigned int)leer_registro(1);
if(p_proc_actual->descriptores[mutexid]==NULL)return -1;
if(p_proc_actual->descriptores[mutexid]->id_propietario!=p_proc_actual->id)return -1;
if(p_proc_actual->descriptores[mutexid]->num_veces_lock==0)return -1;


if(p_proc_actual->descriptores[mutexid]->tipo==RECURSIVO){

p_proc_actual->descriptores[mutexid]->num_veces_lock--;

    if(p_proc_actual->descriptores[mutexid]->num_veces_lock==0 && p_proc_actual->descriptores[mutexid]->procesos_espera->primero!=NULL){

        int nivel = fijar_nivel_int(NIVEL_3);
        BCP* p_proc_bloqueado  = p_proc_actual->descriptores[mutexid]->procesos_espera->primero;
        p_proc_actual->estado=LISTO;
        eliminar_primero(&(p_proc_actual->descriptores[mutexid]->procesos_espera));
        insertar_ultimo(&lista_listos, p_proc_bloqueado);

        p_proc_actual->descriptores[mutexid]->id_propietario=p_proc_bloqueado->id; //¿?

        fijar_nivel_int(nivel);
    }
}else{ //NO RECURSIVO

p_proc_actual->descriptores[mutexid]->num_veces_lock--;
if(p_proc_actual->descriptores[mutexid]->procesos_espera->primero!=NULL){


        int nivel = fijar_nivel_int(NIVEL_3);
        BCP* p_proc_bloqueado  = p_proc_actual->descriptores[mutexid]->procesos_espera->primero;
        p_proc_bloqueado->estado=LISTO;
        eliminar_primero(&(p_proc_actual->descriptores[mutexid]->procesos_espera));
        insertar_ultimo(&lista_listos, p_proc_bloqueado);

        p_proc_actual->descriptores[mutexid]->id_propietario=p_proc_bloqueado->id; //¿?

        fijar_nivel_int(nivel);
}
    return 0;
}
```

Para comenzar se realizan tres comprobaciones esenciales: si el proceso actual contiene ese mutex, es propietario y si dicho mutex está bloqueado. Si se pasan todas las restricciones se diferencian los procesos RECURSIVOS de los NO_RECURSIVOS.

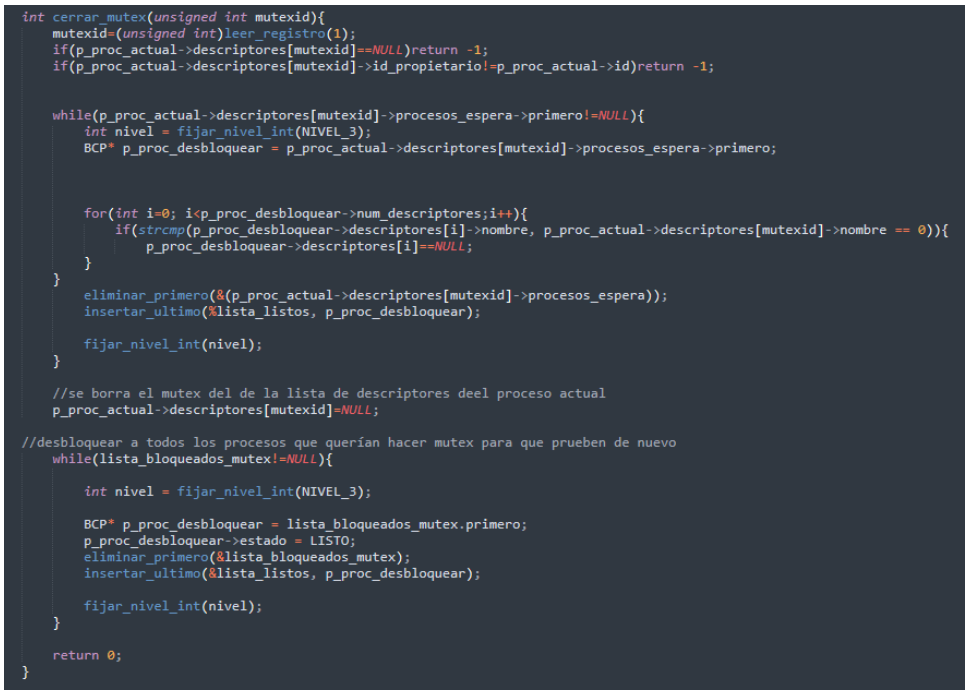
Comenzando con los primeros, se debe realizar una resta en el contador, ya que se está desbloqueando una de las varias veces que puede estar bloqueado.

A continuación, se comprueba si el proceso está totalmente desbloqueado, pero con procesos aun en espera. De ser así se desbloquea al primer proceso que estaba en la lista de espera del mutex (realizando una vez más el proceso de siempre) y se cede el id de propietario a este proceso desbloqueado.

En segundo lugar, aparecen los NO_RECURSIVOS, a los cuales se les resta una unidad (y única) del contador y se realizan las mismas acciones que en el procedimiento anterior para dar paso a un nuevo propietario.

Cerrar_mutex( ):

La última de las funciones que exigía el mutex recibe como único parámetro, otra vez más, mutex id.



```
int cerrar_mutex(unsigned int mutexid){
    mutexid=(unsigned int)leer_registro(1);
    if(p_proc_actual->descriptores[mutexid]==NULL)return -1;
    if(p_proc_actual->descriptores[mutexid]->id_propietario!=p_proc_actual->id)return -1;


    while(p_proc_actual->descriptores[mutexid]->procesos_espera->primero!=NULL){
        int nivel = fijar_nivel_int(NIVEL_3);
        BCP* p_proc_desbloquear = p_proc_actual->descriptores[mutexid]->procesos_espera->primero;



        for(int i=0; i<p_proc_desbloquear->num_descriptores;i++){
            if(strcmp(p_proc_desbloquear->descriptores[i]->nombre, p_proc_actual->descriptores[mutexid]->nombre == 0)){
                p_proc_desbloquear->descriptores[i]==NULL;
        }
    }
        eliminar_primero(&(p_proc_actual->descriptores[mutexid]->procesos_espera));
        insertar_ultimo(%lista_listos, p_proc_desbloquear);

        fijar_nivel_int(nivel);
    }

    //se borra el mutex del de la lista de descriptores deel proceso actual
    p_proc_actual->descriptores[mutexid]=NULL;

//desbloquear a todos los procesos que querían hacer mutex para que prueben de nuevo
    while(lista_bloqueados_mutex!=NULL){

        int nivel = fijar_nivel_int(NIVEL_3);

        BCP* p_proc_desbloquear = lista_bloqueados_mutex.primero;
        p_proc_desbloquear->estado = LISTO;
        eliminar_primero(&lista_bloqueados_mutex);
        insertar_ultimo(&lista_listos, p_proc_desbloquear);

        fijar_nivel_int(nivel);
    }

    return 0;
}
```

Para cerrar el mutex se debe comprobar primero si el proceso que realiza la acción es el propietario de este, y, por ende, si está en su array de descriptores.

Si es así, se procede a desbloquear a todos los procesos que estaban bloqueados en la lista de ese mutex y se les borra de su array de descriptores la referencia a este mutex que se va a cerrar, tras esto, le ocurre lo mismo al proceso actual.

Por último, se debe de desbloquear a todos los procesos (creo se podría desbloquear solo al primero) que estaban bloqueados en la lista de mutex.

Otra función que recibe modificaciones a partir del mutex es “sis_terminar_proceso”:

```
/*
 * Tratamiento de llamada al sistema terminar_proceso. Llama a la
 * funcion auxiliar liberar_proceso
 */
int sis_terminar_proceso(){

    //printk("-> FIN PROCESO %d\n", p_proc_actual->id);

    for(int i = 0; i<p_proc_actual->num_descriptores;i++){
        cerrar_mutex(i);
    }

    liberar_proceso();

        return 0; /* no deber?a llegar aqui */
}
```

La cual debe cerrar todos los mutex del proceso a terminar (por si fuera el propietario).

Finalmente, los archivos secundarios que han sido modificados han sido:

- **Kernel.c:** para añadir todas las funciones comentadas.
- **Kernel.h:** para añadir las nuevas estructuras, listas y prototipos, así como modificar la tabla de servicios.

```
/*
 * Prototipos de las rutinas que realizan cada llamada al sistema
 */
int sis_crear_proceso();
int sis_terminar_proceso();
int sis_escribir();
int obtener_id_pr();
int dormir(unsigned int segundos);

int crear_mutex(char *nombre, int tipo);
int abrir_mutex(char *nombre);
int lock(unsigned int mutexid);
int unlock(unsigned int mutexid);
int cerrar_mutex(unsigned int mutexid);

/*
 * Variable global que contiene las rutinas que realizan cada llamadach
 */
servicio tabla_servicios[NSERVICIOS]={  {sis_crear_proceso},
                                        {sis_terminar_proceso},
                                        {sis_escribir},
                                        {obtener_id_pr},
                                        {dormir},
                                        {crear_mutex},
                                        {abrir_mutex},
                                        {lock},
                                        {unlock},
                                        {cerrar_mutex}
                                     };
```

- **Servicios.h:** para añadir todas las funciones.

```
int crear_mutex (char *nombre, int tipo);
int abrir_mutex(char *nombre);
int lock (unsigned int mutexid);
int unlock (unsigned int mutexid);
int cerrar_mutex (unsigned int mutexid);
#endif /* SERVICIOS_H */
```

- **Llamsis.h:** para añadir las llamadas junto con sus números.

```
#define CREAR_MUTEX 5
#define ABRIR_MUTEX 6
#define LOCK 7
#define UNLOCK 8
#define CERRAR_MUTEX 9

#endif /* _LLAMSIS_H */
```

- **Serv.c:** modificando las interfaces de las llamadas al sistema.

```c
int crear_mutex(char* nombre, int tipo){
    return llamsis(CREAR_MUTEX, 2, (long) nombre, (long) tipo);
}
int abrir_mutex(char* nombre){
    return llamsis(ABRIR_MUTEX, 1, (long) nombre);
}
int lock(unsigned int mutexid){
    return llamsis(LOCK, 1, (long) mutexid);
}
int unlock(unsigned int mutexid){
    return llamsis(UNLOCK, 1, (long) mutexid);
}
int cerrar_mutex(unsigned int mutexid){
    return llamsis(CERRAR_MUTEX, 1, (long) mutexid);
}
```

Conclusiones

La práctica fue muy tediosa al principio del todo, pero una vez empiezas a trabajar, te das cuenta de que muchas cosas son repetidas y que el 30% es leer el manual y el enunciado, siguiendo los pasos escritos, poco a poco y finalmente generando un efecto bola de nieve donde todo está conectado y se tiene una imagen mucho más general.

Esto no quita que ha sido complicada por todo el estrés inicial y espero que la segunda práctica no sea ni de cerca tan complicada.

Por último, quiero dejar presente que este código no ha sido cedido a NINGÚN alumno y no se ha trabajado con NINGÚN otro alumno, en caso de cualquier similitud o inclusión de mi nombre en otro trabajo se me debe hacer sabe para la correspondiente defensa.