Practica 1 - Minikernel

Ampliación de sistemas operativos

Doble grado en Ing. Informática + Ing. Computadores

Javier García Borrego

Contenido

Llamada que bloquea al proceso un plazo de tiempo:

Lo primero que hacemos es añadir la rutina en kernel.c además de añadirlo en la tabla de servicios de kernel.h y sumar uno en llamsis.h en NSERVICIOS dejándolo a 5 luego añadir la interfaz en serv.c y añadir el prototipo de la llamada dormir a servicios.h.

A parte se ha añadido lista_BCPs lista_dormidos = {NULL,NULL}; en kernel.h para cómo se pide en el enunciado definir una lista de procesos esperando plazos y en la función int_reloj se ha añadido el código que se ve en el anexo para contabilizar el tiempo que lleva dormido y cuanto le queda además de que lo despierta cuando llega este a 0.

Ahora se procederá a explicar brevemente el código del anexo:

Lo primero que hacemos es crear unas variables para guardar los segundos que se va a dormir el proceso y otra variable para el nivel de interrupción luego actualizamos el BCP al estado de bloqueo y con el numero de segundos que va a estar dormido, se guarda el nivel de interrupción y se fija en 3, tenemos que sacar de la lista los procesos listos que es el primer proceso y lo ponemos en la lista de bloqueados aquí he decidido restaurar el nivel de la interrupción anterior aunque creo que se puede realizar antes del return 0; y para terminar se obtienen un nuevo proceso actual con el planificador para realizar un cambio de contexto guardando el contexto del proceso que se va a dormir para restaurarlo mas tarde cuando se despierte.

Cuando se produce la interrupción del reloj se mira si el proceso es NULL si no se decrementa el tiempo en uno y se muestra cuanto tiempo le queda y se pasa al siguiente proceso cuando los procesos quedan a 0 se elevan las interrupciones en el reloj para pararlo y se dispone a despertar el proceso.

Mutex:

Lo primero que hacemos es añadir las rutinas en kernel.c ademas de añadirlo en la tabla de servicios de kernel.h y sumar uno en llamsis.h en NSERVICIOS dejándolo a 10 luego añadir la interfaz en serv.c y añadir el prototipo de cada llamada a servicios.h.

Ahora se procederá a explicar brevemente el código del anexo:

Función Crear_mutex:

Miramos que no haya demasiados caracteres, que no hay otro con el mismo nombre y si se han alcanzado el número máximo de mutex creados en el sistema para esto se ha tenido que añadir el código en el anexo (Cambio en crear tarea en el índice) ya que si no se inicializaban los descriptores a -1 daba error en la prueba mutex 1 y 2 y decía que no había descriptores disponibles y en el caso de que dormir se bloquea el proceso hasta que se encuentre disponible un mutex después de esto se crea al mutex.

Función abrir_mutex:

Se abre el mutex ya creado, se busca si hay un descriptor libre y una vez encontrado el descriptor libre se guarda, se asigna y se actualizan las variables.

Función lock:

Se lee el descriptor del mutex, se mira si el mutex ha sido abierto anteriormente y si no da error, ahora miramos si el mutex es recursivo si lo es comprobamos si ha sido bloqueado por el proceso actual si ha sido bloqueado el proceso actual es el dueño y se puede volver a bloquear si no ha sido bloqueado se tiene que bloquear el proceso porque estaríamos en la cola del mutex aplicando algo parecido a lo que hacemos en dormir, si resulta que el mutex no es recursivo se mira si ha sido bloqueado por el proceso actual si esta bloqueado se sale si no el mutex ha sido bloqueado por otro proceso con lo que hay que bloquear el proceso actual utilizando algo parecido a dormir y por ultimo si el mutex no está bloqueado se bloquea y si ha salido todo bien y sin errores se asigna el proceso de ese momento como propietario del mutex.

Función unlock:

Desbloquea el mutex bloqueado por lock que al igual que en lock solo puede desbloquearlo el dueño del mutex.

El código de lock es igual que lock lo único que en vez de bloquear el mutex se desbloquea, pero esencialmente es lo mismo como podemos comprobar en el anexo con el código.

Función Cerrar_mutex:

El mutex se cierra cuando el proceso no lo necesita si el mutex estuviera bloqueado este se desbloquea y en caso de ser recursivo se desbloqueará las veces que haga falta y en caso de que no haya procesos que hayan abierto mutex este será borrado.

Funciones auxiliares:

descriptor_libre(): Busca y devuelve la posición libre en el array de descriptores de cada proceso del sistema.

descriptor_mutex():Busca y devuelve los mutex libres.

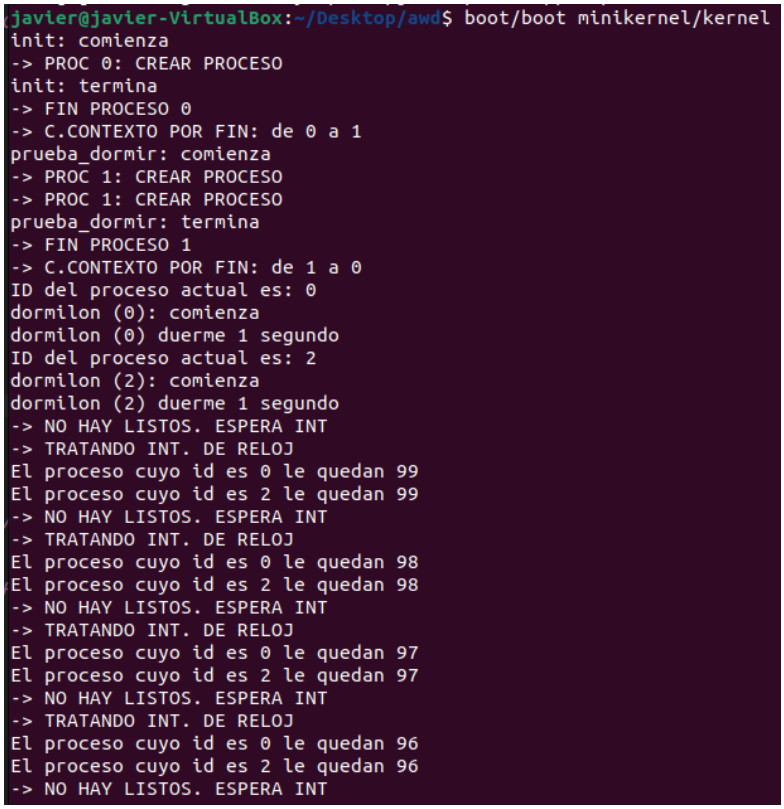
buscar_nombres_iguales(char* nombre): Busca en todos los mutex en busca de un nombre igual para generar un error

Round Robin:

No se ha podido implementar este apartado.

Ejemplos de ejecución:

DORMIR:



```
-> TRATANDO INT. DE RELOJ
El proceso cuyo id es 0 le quedan 1
El proceso cuyo id es 2 le quedan 1
-> NO HAY LISTOS. ESPERA INT
-> TRATANDO INT. DE RELOJ
El proceso cuyo id es 0 le quedan 0
El proceso con id = 0 se despierta
El proceso cuyo id es 2 le quedan 0
El proceso con id = 2 se despierta
dormilon (0) duerme 1 segundos
dormilon (2) duerme 3 segundos
-> NO HAY LISTOS. ESPERA INT
-> TRATANDO INT. DE RELOJ
El proceso cuyo id es 0 le quedan 99
El proceso cuyo id es 2 le quedan 299
-> NO HAY LISTOS. ESPERA INT
-> TRATANDO INT. DE RELOJ
El proceso cuyo id es 0 le quedan 98
El proceso cuyo id es 2 le quedan 298
-> NO HAY LISTOS. ESPERA INT
```

```
-> TRATANDO INT. DE RELOJ
El proceso cuyo id es 0 le quedan 1
El proceso cuyo id es 2 le quedan 201
-> NO HAY LISTOS. ESPERA INT
-> TRATANDO INT. DE RELOJ
El proceso cuyo id es 0 le quedan 0
El proceso con id = 0 se despierta
El proceso cuyo id es 2 le quedan 200
dormilon (0): termina
-> FIN PROCESO 0
-> NO HAY LISTOS. ESPERA INT
-> TRATANDO INT. DE RELOJ
El proceso cuyo id es 2 le quedan 199
-> NO HAY LISTOS. ESPERA INT
```

```
-> TRATANDO INT. DE RELOJ
El proceso cuyo id es 2 le quedan 4
-> NO HAY LISTOS. ESPERA INT
-> TRATANDO INT. DE RELOJ
El proceso cuyo id es 2 le quedan 3
-> NO HAY LISTOS. ESPERA INT
-> TRATANDO INT. DE RELOJ
El proceso cuyo id es 2 le quedan 2
-> NO HAY LISTOS. ESPERA INT
-> TRATANDO INT. DE RELOJ
El proceso cuyo id es 2 le quedan 1
-> NO HAY LISTOS. ESPERA INT
-> TRATANDO INT. DE RELOJ
El proceso cuyo id es 2 le quedan 0
El proceso con id = 2 se despierta
-> C.CONTEXTO POR FIN: de 0 a 2
dormilon (2): termina
-> FIN PROCESO 2
```

Prueba mutex 1 y prueba mutex 2 resultados:

```
javier@javier-VirtualBox:~/Desktop/awd$ boot/boot minikernel/kernel
init: comienza
-> PROC 0: CREAR PROCESO
-> PROC 0: CREAR PROCESO
init: termina
-> FIN PROCESO 0
-> C.CONTEXTO POR FIN: de 0 a 1
prueba_mutex1: comienza
-> PROC 1: CREAR PROCESO
-> PROC 1: CREAR PROCESO
-> PROC 1: CREAR PROCESO
-> PROC 1: CREAR PROCESO
-> PROC 1: CREAR PROCESO
prueba_mutex1: termina
-> FIN PROCESO 1
-> C.CONTEXTO POR FIN: de 1 a 2
prueba_mutex 2 comienza
Creando mutex
Mutex creado correctamente
Haciendo lock
El mutex no existe
error en lock de mutex. DEBE APARECER
Creando mutex
Mutex creado correctamente
Haciendo lock
Lock realizado sobre mutex
Haciendo lock
segundo lock en mutex no recursivo. DEBE APARECER
Haciendo lock
Lock realizado sobre mutex
Haciendo lock
Lock realizado sobre mutex
-> PROC 2: CREAR PROCESO
excepcion de memoria cuando estaba dentro del kernel
```

(no se porque aparece una excepción de memoria pero por lo demás funciona todo bien)

Anexo:

Se aporta capturas de imagen de los cambios realizados en cada archivo.

Kernel.c

Función dormir:

```
int dormir (unsigned int segundos){

        int nivel_int;
        segundos = (unsigned int)leer_registro(1);

        //Guardamos interrupcion
        nivel_int = fijar_nivel_int(NIVEL_3);

        //Actualizamor BCP
        p_proc_actual->estado = BLOQUEADO;
        p_proc_actual->t_dormir = segundos*TICK;
        BCP* p_proc_dormido = p_proc_actual;



        // Sacar de la lista
        eliminar_primero(&lista_listos);
        // Insertarlo
        insertar_ultimo(&lista_dormidos, p_proc_dormido);

        // Restaurar interrupcion
        fijar_nivel_int(nivel_int);


        //cambio de contexto

        p_proc_actual=planificador();
        cambio_contexto(&(p_proc_dormido->contexto_regs), &(p_proc_actual->contexto_regs));

        return 0;
}
```

Cambio en int_reloj descrito anteriormente:

```
static void int_reloj(){

        printk("-> TRATANDO INT. DE RELOJ\n");


        BCP* proceso = lista_dormidos.primero;

        while (proceso != NULL) {
                //Disminuir tiempo en 1
                proceso->t_dormir--;
                printk("El proceso cuyo id es %d le quedan %d\n", proceso-> id, proceso->t_dormir);
                //Si se acaba desbloqueamos pero antes guardamos el proceso al que apunta porque su no podemos apuntar a otro proceso sin dormir
                BCP* proceso_siguiente = proceso->siguiente;
                if(proceso->t_dormir <= 0) {
                        //Elevar nivel interrupcion para parar el reloj
                        int nivel_int = fijar_nivel_int(NIVEL_3);
                        printk("El proceso con id = %d despierta\n", proceso->id);
                        //Cambiar estado quitarlo de dormidos y ponerlo el ultimo en la cola y volver al nivel anterior de interrupcion
                        proceso->estado = LISTO;
                        eliminar_elem(&lista_dormidos, proceso);
                        insertar_ultimo(&lista_listos, proceso);
                        fijar_nivel_int(nivel_int);
                }
                proceso = proceso_siguiente;
        }

        return;
}
```

Funciones de mutex:

```
int crear_mutex (char *nombre, int tipo){
        printk("Creando mutex\n");
        nombre=(char *)leer_registro(1);
        tipo=(int)leer_registro(2);

        if(strlen(nombre) > MAX_NOM_MUT){
                return -1;
        }


        int i;
        for (i = 0; i < NUM_MUT; i++){
                if(array_mutex[i].nombre != NULL &&
                        strcmp(array_mutex[i].nombre, nombre) == 0){
                        return -1;
                }
        }

        int descriptor_proc = descriptor_libre();
        if(descriptor_proc == -1) {
                printk("El proceso no tiene descriptores libres\n");
                return -1;
        }

        |
        while(mutex_creados==NUM_MUT) {

                int nivel_int = fijar_nivel_int(NIVEL_3);

                p_proc_actual->estado=BLOQUEADO;

                eliminar_primero(&lista_listos);
                insertar_ultimo(&lista_bloq_mutex, p_proc_actual);

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

                cambio_contexto(&(p_proc_bloq->contexto_regs), &(p_proc_actual->contexto_regs));
                fijar_nivel_int(nivel_int);
        }


        int descriptor_mut = descriptor_mutex();


        p_proc_actual->descriptores[descriptor_proc] = descriptor_mut;

        array_mutex[descriptor_mut].nombre = nombre;
        array_mutex[descriptor_mut].tipo = tipo;
        array_mutex[descriptor_mut].abierto++;
        mutex_creados++;
        p_proc_actual->n_descriptores++;

        printk("Mutex creado correctamente\n");
        return descriptor_proc;
}
```

```c
int abrir_mutex(char *nombre) {
        printk("Abriendo mutex\n");
        nombre=(char *)leer_registro(1);

        if(strlen(nombre) > MAX_NOM_MUT){
                return -1;
        }


        if(buscar_nombres_iguales(nombre) == 0) {
                printk("No se ha encontrado un mutex con este nombre\n");
                return -1;
        }

        int descriptor_proc = descriptor_libre();
        if(descriptor_proc == -1) {
                printk("El proceso no tiene descriptores libres\n");
                return -1;
        }


        int descriptor_mut;
        for(int i=0; i<mutex_creados; i++) {
                if(strcmp(array_mutex[i].nombre, nombre) == 0) {
                        descriptor_mut = i;
                }
        }


        p_proc_actual->descriptores[descriptor_proc]=descriptor_mut;
        p_proc_actual->n_descriptores++;

        array_mutex[descriptor_mut].abierto++;

        printk("Mutex abierto correctamente\n");
        return descriptor_proc;
}
```

```
int lock (unsigned int mutexid) {
        printk("Haciendo lock\n");
        int desc_proc=(unsigned int)leer_registro(1);
        mutexid = p_proc_actual->descriptores[desc_proc];
        int proceso_esperando = 1;

        if(mutexid == -1) {
                printk("El mutex no existe \n");
                return -1;
        }

        while (proceso_esperando) {


                if(array_mutex[mutexid].locked > 0) {
                        if((array_mutex[mutexid].tipo) == RECURSIVO) {
                                if(array_mutex[mutexid].propietario == p_proc_actual->id) {

                                        array_mutex[mutexid].locked++;
                                        proceso_esperando = 0;
                                }

                                else {
                                        int nivel_int = fijar_nivel_int(NIVEL_3);


                                        p_proc_actual->estado=BLOQUEADO;

                                        eliminar_primero(&lista_listos);
                                        insertar_ultimo(&(array_mutex[mutexid].lista_proc_esperando_lock), p_proc_actual);


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


                                        cambio_contexto(&(p_proc_bloq->contexto_regs), &(p_proc_actual->contexto_regs));


                                        fijar_nivel_int(nivel_int);
                                }

                        }

                        else {

                                if(array_mutex[mutexid].propietario == p_proc_actual->id) {
                                        return -1;
                                }

                                else {

                                        int nivel_int = fijar_nivel_int(NIVEL_3);


                                        p_proc_actual->estado=BLOQUEADO;

                                        eliminar_primero(&lista_listos);
                                        insertar_ultimo(&(array_mutex[mutexid].lista_proc_esperando_lock), p_proc_actual);


                                        BCP* p_proc_bloq = p_proc_actual;
                                        p_proc_actual=planificador();
                                        cambio_contexto(&(p_proc_bloq->contexto_regs), &(p_proc_actual->contexto_regs));
```

```
                                        fijar_nivel_int(nivel_int);
                                }
                        }
                }

                else {
                        array_mutex[mutexid].locked++;
                        proceso_esperando = 0;
                }
        }

        array_mutex[mutexid].propietario = p_proc_actual->id;
        printk("Lock realizado sobre mutex\n");
        return 0;
```

```
int unlock (unsigned int mutexid) {

	int desc_proc=(unsigned int)leer_registro(1);
	mutexid = p_proc_actual->descriptores[desc_proc];


	if(array_mutex[mutexid].abierto == 0) {
		return -1;
	}


	if(array_mutex[mutexid].locked > 0) {

		if((array_mutex[mutexid].tipo) == RECURSIVO) {
			if(array_mutex[mutexid].propietario == p_proc_actual->id) {

				array_mutex[mutexid].locked--;

				if(array_mutex[mutexid].locked == 0) {
					if(((array_mutex[mutexid].lista_proc_esperando_lock).primero) != NULL) {

						int nivel_int = fijar_nivel_int(NIVEL_3);
						BCP* proc_esperando = (array_mutex[mutexid].lista_proc_esperando_lock).primero;
						proc_esperando->estado = LISTO;
						eliminar_primero(&(array_mutex[mutexid].lista_proc_esperando_lock));
						insertar_ultimo(&lista_listos, proc_esperando);
						fijar_nivel_int(nivel_int);
					}

					array_mutex[mutexid].propietario = -1;
				}
			}
			else {
				return -1;
			}
		}
		else {

			if(array_mutex[mutexid].propietario == p_proc_actual->id) {

				array_mutex[mutexid].locked--;
				array_mutex[mutexid].propietario = -1;

				if(((array_mutex[mutexid].lista_proc_esperando_lock).primero) != NULL) {
					int nivel_int = fijar_nivel_int(NIVEL_3);

					BCP* proc_esperando = (array_mutex[mutexid].lista_proc_esperando_lock).primero;
					proc_esperando->estado = LISTO;
					eliminar_primero(&(array_mutex[mutexid].lista_proc_esperando_lock));
					insertar_ultimo(&lista_listos, proc_esperando);

					fijar_nivel_int(nivel_int);

				}
			}
			else {
				return -1;
			}
		}
	}

	else {
		return -1;
	}

	printk("Unlock realizado correctamente\n");
```

```
int cerrar_mutex (unsigned int mutexid) {

        unsigned int registro=(unsigned int)leer_registro(1);
        if(registro < 16) {
                mutexid = registro;
        }

        int desc_mut = p_proc_actual->descriptores[mutexid];

        if(array_mutex[desc_mut].nombre == NULL) {
                return -1;
        }


        p_proc_actual->descriptores[mutexid] = -1;
        p_proc_actual->n_descriptores--;


        if(array_mutex[desc_mut].propietario == p_proc_actual->id) {

                array_mutex[desc_mut].locked = 0;

                while((array_mutex[desc_mut].lista_proc_esperando_lock).primero != NULL) {

                        int nivel_int = fijar_nivel_int(NIVEL_3);

                        BCP* proc_esperando = (array_mutex[desc_mut].lista_proc_esperando_lock).primero;
                        proc_esperando->estado = LISTO;
                        eliminar_primero(&(array_mutex[desc_mut].lista_proc_esperando_lock));
                        insertar_ultimo(&lista_listos, proc_esperando);

                        fijar_nivel_int(nivel_int);

                }

        }
        array_mutex[desc_mut].abierto--;


        if(array_mutex[desc_mut].abierto <= 0) {
                mutex_creados--;
                while(lista_bloq_mutex.primero != NULL) {
                        int nivel_int = fijar_nivel_int(NIVEL_3);

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

                        fijar_nivel_int(nivel_int);
                        printk("Se ha desbloqueado el proceso\n");
                }
        }



        return 0;
}
```

Cambio en crear_tarea:

```
        //modificar para que funcione mutex
        for(int i = 0; i< NUM_MUT_PROC; i++){
                p_proc->descriptores[i] = -1;
        }

        p_proc->n_descriptores = 0;
```

Funciones auxiliares:

```
int descriptor_libre(){

        int i = 0;
        for(i=0; i<NUM_MUT_PROC; i++){
                //Si descriptor = -1, no ha sido utilizado
                if(p_proc_actual->descriptores[i] == -1) {
                        return i;
                }

        }

        return -1;
}

int descriptor_mutex(){

        int aux = -1;
        int i = 0;

        while((aux == -1) && (i <= NUM_MUT)) {
                //Si descriptor = -1, no ha sido utilizado
                if(array_mutex[i].abierto == 0) {
                        aux = i;
                }

                i++;
        }

        return aux;
}

int buscar_nombres_iguales(char* nombre){

        int i;
        for(i=0;i<mutex_creados;i++){
                if(array_mutex[i].abierto != 0){
                        if(strcmp(array_mutex[i].nombre,nombre)==0){
                                return -1;
                        }
                }
        }

        return 0;

}
```

Kernel.h:

```
#define NO_RECURSIVO 0
#define RECURSIVO 1
```

```
//Dormir
int dormir(unsigned int segundos);//llamada de dormir

lista_BCPs lista_dormidos = {NULL, NULL};

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

//Struct para el mutex
typedef struct {
        char *nombre;
        int tipo; //Recursivo o no recursivo
        int propietario; //Id del proceso
        int abierto;
        int locked;
        //Lista de cada mutex
        lista_BCPs lista_proc_esperando_lock;
} mutex;
//Array para guardar mutex utilizados. Tamaño = numero maximo de mutex
mutex array_mutex[NUM_MUT];
//Variable que almacena el numero de mutex creados
int mutex_creados;
//Lista de procesos bloqueados porque se habian creado el numero maximo de mutex permitidos
lista_BCPs lista_bloq_mutex = {NULL, NULL};

/*
 * Variable global que contiene las rutinas que realizan cada llamada
 */
servicio tabla_servicios[NSERVICIOS]={
                                        {sis_crear_proceso},
                                        {sis_terminar_proceso},
                                        {sis_escribir},
                                        {obtener_id_pr},
                                        {dormir}, //1) Rutina Dormir
                                        {crear_mutex},
                                        {abrir_mutex},
                                        {lock},
                                        {unlock},
                                        {cerrar_mutex}
                                };

#endif /* _KERNEL_H */
```

Servicios.h:

```c
//Llamada a la funcion dormir
int dormir (unsigned int segundos);

//Definicion de mutex recursivo o no
#define RECURSIVO 1
#define NO_RECURSIVO 0

//Llamadas de la funcion Mutex
int crear_mutex (char* nombre, int tipo);
int abrir_mutex (char* nombre);
int lock (unsigned int mutex_id);
int unlock (unsigned int mutex_id);
int cerrar_mutex (unsigned int mutex_id);
```

Serv.c:

```c
int dormir (unsigned int segundos) {
        return llamsis(DORMIR, 4, (long) segundos);
}
int crear_mutex (char* nombre, int tipo) {
        return llamsis(CREAR_MUTEX, 5, (long) nombre,(long) tipo);
}
int abrir_mutex (char* nombre) {
        return llamsis(ABRIR_MUTEX, 6, (long) nombre);
}
int lock (unsigned int mutex_id) {
        return llamsis(LOCK, 7, (long) mutex_id);
}
int unlock (unsigned int mutex_id) {
        return llamsis(UNLOCK, 8, (long) mutex_id);
}
int cerrar_mutex (unsigned int mutex_id) {
        return llamsis(CERRAR_MUTEX, 9, (long) mutex_id);
}
```

Llamsis.h:

```c
#ifndef _LLAMSIS_H
#define _LLAMSIS_H

/* Numero de llamadas disponibles */
#define NSERVICIOS 10

#define CREAR_PROCESO 0
#define TERMINAR_PROCESO 1
#define ESCRIBIR 2
#define OBTENERID 3
//Numero de llamada para dormir
#define DORMIR 4

//Numeros de llamada para las funciones MUTEX
#define CREAR_MUTEX 5
#define ABRIR_MUTEX 6
#define LOCK 7
#define UNLOCK 8
#define CERRAR_MUTEX 9

#endif /* _LLAMSIS_H */
```