Universidad Rey Juan Carlos

Escuela Técnica Superior
Ingeniería Informática

Practica 1
Minikernel

Grado de Ingeniería de Computadores.
Asignatura: Ampliación de sistemas operativos
Curso: 2021-2022

Autor: Yaroslav Dytko

Contenido

Introducción

La realización de la práctica de minikernel consiste en trabajar sobre una versión inicial del minikernel que se entrega como material de apoyo para así incluir nuevas funcionalidades y modificar algunas de las ya existentes, también se añadirá multiprogramación como uno de los puntos a desarrollar en la práctica.

Un breve resumen de nuevas funcionalidades que se van a incluir al minikernel durante el desarrollo de la practica:

- Llamada que bloquea un proceso durante un determinado tiempo.
- Servicio de sincronización basado en mutex.
- Sustituir el algoritmo de planificación FIFO por un Round-Robin.

a continuación, se irán detallando los aspectos de desarrollo de esas nuevas funcionalidades.

Llamada que bloquea un proceso durante un determinado tiempo

De acuerdo con el enunciado de la practica en este apartado se desarrolla la llamada al sistema *dormir ()* , la cual bloquea al proceso en ejecución un plazo de tiempo determinado recibido por parámetro, aparte del desarrollo de la propia función *dormir* se ha tenido que crear una lista de procesos adicional donde se trasladan los procesos bloqueados, también en la propia definición de tipo proceso se ha incluido un *contador de tiempo*(llamado tiempo_dormir) que el proceso va estar dormido. Dado que de alguna manera hay que ir controlando el tiempo que va durmiendo el proceso se ha desarrollado una función auxiliar llamada *cuentaAtrasBloqueados* la cual es llamada en cada interrupción de reloj, recorre la lista de procesos bloqueados y decrementa en una unidad su contador de tiempo para dormir, cuando el contador llega a 0 se realiza el cambio de estado de proceso bloqueado a listo con su respectivo cambio a lista de procesos listos.

También se ha tenido que modificar la función *crear_tarea* para inicializar a 0 el contador de tiempo cuando se crea el proceso.

Las modificaciones y el código desarrollado se podrán ver en el apartado correspondiente del anexo.

Servicio de sincronización basado en mutex

En esta ocasión se trata de desarrollar un servicio de manejo de mutex en el minikernel.
Este servicio consta de varias llamadas al sistema para poder crear, abrir, bloquear, desbloquear y cerrar los mutex de manera segura y sin producir interbloqueos entre procesos. Para este servicio se ha desarrollado el tipo de dato llamado mutex sobre el cual interaccionan las llamadas al sistema antes mencionadas, también se ha creado una lista de procesos adicional para almacenar ahí a los procesos que se han bloqueado en la creación de mutex. En este punto de la practica hay que prestar especial atención a la hora de desarrollar las llamadas al sistema y usar las interrupciones pertinentes, el uso de las interrupciones es bastante critico en este servicio ya que sin ellas no se puede conseguir el máximo grado de coherencia sobre todo a la hora de mover los procesos entre listas de procesos ya que sin uso de interrupciones podría dejar un proceso “en el aire” es decir fuer de su lista de procesos originar pero sin entrar en la lista destino.
Este servicio a ser el mas extenso que se pide realizar en la práctica conlleva con ello una gran cantidad de modificaciones y creación de variables y funciones nuevas.

Se han desarrollado 5 llamadas al sistema principales:

- crear_mutex – crea el mutex y lo almacena en lista mutex, esta versión de minikernel tiene un numero limitado de mutex que se pueden crear, esto implica que si algún proceso quiere crear un mutex y el numero de mutex creado ya es el máximo, el proceso es bloqueado y almacenado en una lista de procesos creada para ese propósito llamada *lista_bloqueados_mutex* al igual se incrementa en 1 el contador de procesos bloqueados para dicha lista llamado: *contador_lista_bloqueados_mutex*.
- abrir_mutex – abre el mutex ya creado, lo que permite que el proceso que lo abre tenga acceso a dicho mutex, esto se consigue asignando a una de las posiciones del array llamado *descriptores* la posición de dicho mutex en lista mutex.
- lock – bloquea el mutex, esta acción solo la puede realizar el proceso que anteriormente ha abierto ese mutex, sobre los mutex recursivos se permite hacer varios bloqueos, si el lock sobre el mutex se realiza con éxito, la variable de control *mutex_lock* se pone a 0 lo cual es como si fuera true en C, si el proceso que intenta hacer lock sobre el mutex no es su propietario en ese momento, ese proceso se queda bloqueado y almacenado en una lista interna que tiene el propio mutex llamada *lista_procesos_esperando* a la espera de ser despertado cuando se desbloquee el mutex por su propietario.
- unlock – desbloquea el mutex bloqueado por la llamada al sistema lock, de mismo modo que en lock solo lo puede desbloquear el proceso propietario del mutex en ese momento, es decir el proceso que lo ha bloqueado, en caso de mutex recursivos se puede desbloquear varias veces, cuando el mutex ya esta completamente desbloqueado es decir cuando su contador de bloqueos es igual a 0 la variable de control *mutex_lock* se pone a 1 lo cual es como si fuera false en C, es cuando deja de ser propiedad del proceso que lo desbloqueo, en ese momento se comprueba por si hay algún proceso dentro de *lista_procesos_esperando* para poder despertarlo y cederle el mutex desbloqueado.
- cerrar_mutex – cuando el proceso no necesita el mutex lo cierra, como consecuencia de esto en caso de estar bloqueado el mutex se desbloquea antes del cierre lo cual se controla mediante la variable interna de mutex *mutex_lock* que funciona a modo de booleano permitiendo saber si el mutex esta o no bloqueado, en caso de estarlo se desbloquea las veces que haga falta, ya sea 1 vez para mutex no recursivo o N veces para mutex recursivo. Como fase final del cierre de mutex se comprueba la *lista_bloqueados_mutex* por si hay algún proceso bloqueado esperando para crear mutex, es caso de si haberlo ese proceso se desbloquea y se cambia a lista de procesos listos.

  Dado que los mutex también se tienen que cerrar de manera implícita cuando un proceso acaba, se ha tenido que modificar la función *sis_terminar_proceso* implementando una lógica para en caso de haber mutex no cerrados cerrarlos en ese momento.

  Además, se han creado una serie de funciones auxiliares que disminuyen la cantidad de código en las llamadas al sistema, lo que implica reutilización de código y una lectura y comprensión más fácil de dichas llamadas al sistema.

  Al igual que en el apartado interior en la función *crear_tarea* se han inicializado los valores del contador de descriptores abiertos del proceso y el array de descriptores.

Las modificaciones y el código desarrollado se podrán ver en el apartado correspondiente del anexo.

Funciones auxiliares para servicio mutex:

- buscarPosicionMutexLibre – recorre la lista_mutex y devuelve la primera posición libre encontrada en esa lista.
- buscarMutexPorNombre – busca un mutex por su nombre y devuelve su posición en lista_mutex
- buscarDescriptorLibrePorceso – busca y devuelve la posición libre en el array de descriptores que tiene cada proceso del sistema
- buscarMutexPorID – busca el descriptor de mutex dentro del array de descriptores del proceso, devuelve 2 valores, la posición relativa al array de descriptores y la posición relativa a lista_mutex que es donde se encuentra el mutex realmente, he optado por esta implementación algo no común ya que simplifica bastante la tarea de buscar y obtener información del mutex, ya que en una llamada a una función se obtienen tanto el lugar donde se guarda el id del mutex por el proceso actual, como su ubicación dentro de la lista_mutex
- iniciar_lista_mutex – teniendo en cuenta que una de las cosas añadidas fue la lista_mutex, que es un array de tamaño fijo donde se guardan los mutex, a la hora de iniciar el minikernel hay que inicializar esa lista y mas en concreto los mutex almacenados ahí con una serie de valores iniciales

Sustituir el algoritmo de planificación FIFO por un Round-Robin

En este apartado se desarrolla el algoritmo de planificación de procesos Round-Robin, el funcionamiento de ese algoritmo se basa en que a cada proceso de sistema se le asigna un valor de tiempo llamado "TICKS_POR_RODAJA", ese valor de tiempo es el mismo para todos y se va reduciendo con cada interrupción de reloj, esto implica que cuando a un proceso se le acaba el tiempo y aun no ha terminado su ejecución, ese proceso se expulsa de ejecución poniéndose al final de la cola de procesos listos.

Como punto de entrada en este apartado se ha incluido en el tipo de dato BCP un contador de ticks llamado *contadorTicks*, del mismo modo que antes ese contador de ticks se inicializa con su valor pertinente en la función *crear_tarea* y se le asigna el valor de la constante TICKS_POR_RODAJA.

En este caso se han creado las siguientes funciones:

- actualizarTick – funcion de actualización de contador de ticks, esta función decrementa el contador de ticks del proceso que se encuentra en ejecución en este momento, dicha función es llamada en cada interrupción de reloj, en caso de que el contador de ticks llega a cero se activa la interrupción software mediante llamada a *activar_int_SW()*
- tratarIntSW – como su propio nombre indica esta función trata la interrupción software activada por la función *actualizarTick*, se tratan dos situaciones, la primera es cuando el proceso en ejecución es el único proceso que se esta ejecutando en el sistema, entonces no hace falta realizar replanificación y moverlo al final de la lista de procesos listos, simplemente se actualiza su contador de ticks, en caso de haber mas de un proceso el proceso en ejecución se expulsa ya que se le ha acabado el tiempo y pasa al final de la cola de procesos listos, guardando el contexto y actualizándose su contador de ticks. Esta función es llamada desde *int_sw.*

Las modificaciones y el código desarrollado se podrán ver en el apartado correspondiente del anexo.

Conclusiones y opinión personal

En el transcurso de desarrollo de esta practica de minikernel se ha aprendido(al menos por mi parte), manejo de llamadas del sistema, analizar código de otros autores sobre el que se trabaja(el código inicial de la practica), manejo de makefiles, nociones sobre programación de sistemas operativos aunque sean de nivel rudimentario, manejo de interrupciones, mejora de habilidades de programacion en lenguaje C, funcionamiento de un kernel de sistema operativo, la multiprogramación es una gran parte de la practica ya que el apartado más extenso justo trata ese aspecto.

Como opion personal es una de las practicas mas extensas de la carrera, y su nivel de dificultad es bastante alto, se agradecería también que se proporcionara algo más de información explicativa sobre el código inicial desde el cual partimos ya que son infinidad de cosas que hay que tener en cuenta o llegar a ellas a modo de idea feliz.

Anexo

Llamada que bloquea un proceso durante un determinado tiempo

Funciones creadas:

```
/* Llamada que bloquea al proceso un plazo de tiempo */
int dormir(unsigned int segundos){

	/* lectura de registro 1 */
	unsigned int segs = (unsigned int)leer_registro(1);

	/* guardar el nivel de interrupcion */
	int n_interrupcion = fijar_nivel_int(NIVEL_1);
	BCPptr proceso_dormido = p_proc_actual;

	/* actualizacion de estructura de datos */
	proceso_dormido -> tiempo_dormir = segs*TICK;
	proceso_dormido -> estado = BLOQUEADO;

	/* cambio de lista de procesos */
	eliminar_primero(&lista_listos);
	insertar_ultimo(&lista_bloqueados_dormir,proceso_dormido);

	/* usando el planificador se obtiene el proceso a ejecutar */
	p_proc_actual=planificador();

	/* se restaura el nivel de interrupcion guardado */
	cambio_contexto(&(proceso_dormido->contexto_regs),&(p_proc_actual->contexto_regs));
	fijar_nivel_int(n_interrupcion);

	return 0;
}

/* funcion auxiliar para la llamada dormir, actualiza los tiempos de los procesos dormidos */
void cuentaAtrasBloqueados(){

	BCPptr auxiliar = lista_bloqueados_dormir.primero;
	/* recorro la lista y actualizo los tiempos */
	while(auxiliar != NULL){
		BCPptr siguiente = auxiliar->siguiente;
		auxiliar->tiempo_dormir--;
		if(auxiliar->tiempo_dormir ==0){
			auxiliar->estado = LISTO;
			eliminar_elem(&lista_bloqueados_dormir, auxiliar);
			insertar_ultimo(&lista_listos, auxiliar);
		}
		auxiliar = siguiente;
	}
}
```

Modificaciones realizadas:

En kerel.h -> BCP se incluye tiempo_dormir:

```
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 */
         /* ----------cosas añadidas---------- */
         /* añadidos para la llamada dormir */
         int tiempo_dormir;                         /* tiempo que se queda el proceso dormido */
} BCP;
```

En kernel.h se crea la lista de procesos bloqueados para dormir

```
/* ---------llamada al sistema dormir--------- */

/* Variable global que representa la cola de procesos bloqueados por la llamada al sistema dormir()*/

lista_BCPs lista_bloqueados_dormir = {NULL,NULL};
```

En kernel.c -> int_reloj(), se incluye la llamada al cuentaAtrasBloqueados()

```
static void int_reloj(){

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

         /* procesos dormidos */
         cuentaAtrasBloqueados();

   return;
}
```

Servicio de sincronización basado en mutex

Variables y tipos de datos creados

En kernel.h se crean una serie de defines para la lógica de programación de los mutex, tipo de datos mutex, lista_mutex, el contador de lista_mutex, también la lista de procesos bloqueados a la espera para crear mutex y su contador correspondiente.

```
/* ----------cosas añadidas para mutex---------- */
#define NO_RECURSIVO 0/* tipo de mutex no recursivo */
#define RECURSIVO 1		/* tipo de mutex recursivo */
#define LOCKED 0		/* mutex bloqueado */
#define UNLOCKED 1		/* mutex desbloqueado */

* ---------mutex--------- */
/* definicion de tipo que corresponde con un mutex */

typedef struct MUTEX_t *MUTEXptr;

typedef struct MUTEX_t{
	char nombre[MAX_NOM_MUT];		/* nombre del mutex */
	int estado;				 /* estado de mutex LIBRE|OCUPADO */
	int tipo;				/* tipo de mutex NO RECURSIVO|RECURSIVO */
	int num_procesos_esperando;		/* contador de procesos esperando al mutex */
	lista_BCPs lista_procesos_esperando;	/* lista de procesos esperando al mutex */
	int	id_proceso_propietario;		/* puntero al proceso actual poseedor del mutex */
	int contador_bloqueos;			/* contador de bloqueos del mutex */
	int mutex_lock;				/* 1 - LOCK | 0 - UNLOCK */
} mutex;

mutex lista_mutex[NUM_MUT];			/* lista de nombres de mutex en el sistema */
int contador_lista_mutex;			/* contador de mutex en el sistema */

/* Variable global que representa la cola de procesos esperando al mutex*/
lista_BCPs lista_bloqueados_mutex = {NULL,NULL};
int contador_lista_bloqueados_mutex;
```

En kernel.c funciones auxiliares:

```
int buscarPosicionMutexLibre(){
	int i;
	for (i = 0; i < NUM_MUT; i++)
	{
		if (lista_mutex[i].estado==LIBRE)
		{
			return i; /* devuelve la posicion del mutex libre */
		}
	}
	return -1;
}
```

```
int buscarMutexPorNombre(char *nombre){
	int i;
	for ( i = 0; i < NUM_MUT; i++)
	{
		if (strcmp(lista_mutex[i].nombre,nombre)==0)
		{
			return i; /* devuelve la posicion del mutex encontrado */
		}
	}
	return -1;				/* devuelve -1 si no encuentra mutex */
}

int buscarDescriptorLibrePorceso(){
	int i;
	for ( i = 0; i < NUM_MUT_PROC; i++)
	{
		if (p_proc_actual->descriptores[i]==-1)
		{
			return i; /* devuelve la posicion del descriptor libre */
		}
	}
	return -1;
}

int *buscarMutexPorID(int mutexid){
	int i;
	static int retorno[2] = {-1,-1};
	for ( i = 0; i < NUM_MUT_PROC; i++)
	{
		if (p_proc_actual->descriptores[i]==mutexid)
		{
			retorno[0]=i;
			retorno[1]=p_proc_actual->descriptores[i];
			return retorno;
		}
	}
	return retorno;
}

void iniciar_lista_mutex(){
	int i;
	for ( i = 0; i < NUM_MUT; i++)
	{
		lista_mutex[i].contador_bloqueos=0;
		lista_mutex[i].estado=LIBRE;
		lista_mutex[i].id_proceso_propietario=-1;
		lista_mutex[i].lista_procesos_esperando.primero=NULL;
		lista_mutex[i].mutex_lock=UNLOCKED;
		lista_mutex[i].num_procesos_esperando=0;
	}
}
```

Llamadas al sistema:

```
/* llamada al sistema para crear mutex */
int crear_mutex(char *nombre, int tipo){
	nombre = (char*) leer_registro(1);
	tipo = (int) leer_registro(2);
	int n_interrupcion, desc_lista_mutex, desc_libre_proc, mutex_creado;
	n_interrupcion = fijar_nivel_int(NIVEL_1);
	/* comprobacion de longitud de nombre */
	if (strlen(nombre)>MAX_NOM_MUT)
	{
		printk("Error, nombre de mutex sobrepasa la logintud establecida\n");
		fijar_nivel_int(n_interrupcion);
		return -1;
	}
	/* comprobacion de duplicidad de nombres */
	if (buscarMutexPorNombre(nombre)!=-1)
	{
		printk("Error, mutex %s ya existe en el sistema\n",nombre);
		fijar_nivel_int(n_interrupcion);
		return -1;
	}
	/* comprobacion de descriptores libres en el proceso actual */
	desc_libre_proc = buscarDescriptorLibrePorceso();
	if (desc_libre_proc==-1)
	{
		printk("Error, el proceso id: %d no tiene descriptores libres\n",p_proc_actual->id);
		fijar_nivel_int(n_interrupcion);
		return -1;
	}
	/* comprobacion de hueco libre en lista mutex */
	mutex_creado = 0;
	while (mutex_creado==0)
	{
		desc_lista_mutex = buscarPosicionMutexLibre();
		if (desc_lista_mutex==-1)
		{
			mutex_creado=0;
			printk("Error, alcanzado maximo de mutex creados en el sistema\n");
			printk("Bloqueando proceso id: %d\n",p_proc_actual->id);
			contador_lista_bloqueados_mutex++;
			BCPptr p_proc_bloqueado = p_proc_actual;
			p_proc_bloqueado->estado = BLOQUEADO;
			eliminar_primero(&lista_listos);
			insertar_ultimo(&lista_bloqueados_mutex,p_proc_bloqueado);
			p_proc_actual = planificador();
			printk("C.CONTEXTO POR BLOQUEO de %d a %d\n",p_proc_bloqueado-
>id,p_proc_actual->id);
			cambio_contexto(&(p_proc_bloqueado->contexto_regs),&(p_proc_actual-
>contexto_regs));
		}
```

```
                /* si ha pasado todas las pruebas y no se ha bloqueado */
                /* crea el mutex */
                MUTEXptr m = &lista_mutex[desc_lista_mutex];
                strcpy(m->nombre,nombre);
                m->tipo=tipo;
                m->estado = OCUPADO;
                contador_lista_mutex++;
                /* abre el mutex */
                p_proc_actual->descriptores[desc_libre_proc]=desc_lista_mutex;
                p_proc_actual->num_descriptores_abiertos++;
                printk("Mutex %s CREADO y ABIERTO\n",m->nombre);
                mutex_creado=1;
        }

        fijar_nivel_int(n_interrupcion);
        return desc_libre_proc;
}

/* llamada al sistema para abrir mutex */
int abrir_mutex(char *nombre){

        nombre = (char*) leer_registro(1);
        int n_interrupcion,pos_lista_mutex, desc_libre_proc;

        n_interrupcion = fijar_nivel_int(NIVEL_1);

        /* busqueda de mutex por su nombre */
        pos_lista_mutex = buscarMutexPorNombre(nombre);
        if (pos_lista_mutex==-1)
        {
                printk("Error, mutex %s no encontrado\n",nombre);
                fijar_nivel_int(n_interrupcion);
                return -1;
        }
        /* comprobacion de mutex disponibles en proceso actual */
        desc_libre_proc = buscarDescriptorLibrePorceso();
        if (desc_libre_proc==-1)
        {
                printk("Error, el proceso id: %d no tiene descriptores libres\n",p_proc_actual->id);
                fijar_nivel_int(n_interrupcion);
                return -1;
        }

        p_proc_actual->descriptores[desc_libre_proc] = pos_lista_mutex;
        p_proc_actual->num_descriptores_abiertos++;
        printk("Mutex %s ABIERTO\n",nombre);
        fijar_nivel_int(n_interrupcion);

        return desc_libre_proc;

}
```

```
/* llamada al sistema para boquear mutex */
int lock(unsigned int mutexid){
	int n_interrupcion, desc_proc, pos_lista_mutex, lock;
	int *retorno;
	unsigned int m_id = (unsigned int) leer_registro(1);
	if (m_id<NUM_MUT)
	{
		mutexid=m_id;
	}
	n_interrupcion = fijar_nivel_int(NIVEL_1);
	retorno = buscarMutexPorID((int)mutexid);
	desc_proc = *retorno;
	pos_lista_mutex = *(retorno+1);
	if (desc_proc==-1)
	{
		printk("Error, mutex con mutexid: %d no encontrado\n",mutexid);
		fijar_nivel_int(n_interrupcion);
		return -1;
	}
	MUTEXptr m = &lista_mutex[pos_lista_mutex];
	//printk("Mutex %s encontrado | num_bloqueos: %d, id propietario: %d\n",m->nombre,m-
>contador_bloqueos,m->id_proceso_propietario);
	lock=0;
	while (lock==0)
	{
		if (m->id_proceso_propietario==-1 && m->contador_bloqueos==0)
		{
			m->contador_bloqueos++;
			m->id_proceso_propietario=p_proc_actual->id;
			m->mutex_lock=LOCKED;

			printk("Mutex %s BLOQUEADO\n",m->nombre);

			fijar_nivel_int(n_interrupcion);
			return desc_proc;
		}
		if (m->id_proceso_propietario==p_proc_actual->id)
		{
			if (m->mutex_lock==LOCKED && m->tipo==RECURSIVO)
			{
				lock=1;
				m->contador_bloqueos++;
				printk("Mutex RECURSIVO %s BLOQUEADO\n",m->nombre);
				return desc_proc;
			} else if (m->mutex_lock==LOCKED)
			{
				printk("Error, intento de bloquear mutex %s ya bloqueado y de tipo NO
RECURSIVO\n",m->nombre);
				fijar_nivel_int(n_interrupcion);
				return -1;
			}
		}
```

```
                        /* si llega hasta aqui esque proceso actual no es propietario del mutex y sera bloqueado*/
                        //printk("Error, proceso id: %d no es propietario de mutex %s, id propietario real:
%d\n",p_proc_actual->id,m->nombre,m->id_proceso_propietario);
                        //printk("Bloqueando proceso id: %d\n",p_proc_actual->id);
                        lock=0;
                        m->num_procesos_esperando++;
                        BCPptr p_proc_bloqueado = p_proc_actual;
                        p_proc_bloqueado->estado = BLOQUEADO;
                        eliminar_primero(&lista_listos);
                        insertar_ultimo(&(m->lista_procesos_esperando),p_proc_bloqueado);
                        p_proc_actual = planificador();
                        printk("C.CONTEXTO POR BLOQUEO de %d a %d\n",p_proc_bloqueado->id,p_proc_actual->id);
                        cambio_contexto(&(p_proc_bloqueado->contexto_regs),&(p_proc_actual->contexto_regs));
                }

                fijar_nivel_int(n_interrupcion);
                return -1;
}

/* llamada al sistema para desbloquear mutex */
int unlock(unsigned int mutexid){
                int n_interrupcion, desc_proc, pos_lista_mutex;
                int *retorno;
                unsigned int m_id = (unsigned int) leer_registro(1);

                if (m_id<NUM_MUT)
                {
                        mutexid=m_id;
                }

                n_interrupcion = fijar_nivel_int(NIVEL_1);

                retorno = buscarMutexPorID((int)mutexid);
                desc_proc = *retorno;
                pos_lista_mutex = *(retorno+1);
                if (desc_proc==-1)
                {
                        printk("Error, mutex con mutexid: %d no encontrado\n",mutexid);
                        fijar_nivel_int(n_interrupcion);
                        return -1;
                }
                MUTEXptr m = &lista_mutex[pos_lista_mutex];
                if (m->mutex_lock==LOCKED)
                {
                        //printk("Mutex %s esta bloqueado, desbloqueando\n",m->nombre);
                        m->contador_bloqueos--;
                        //printk("Contador de bloqueos decrementado: %d, mutex %s\n",m->contador_bloqueos,m-
>nombre);
                        if (m->contador_bloqueos==0)
                        {
                                m->mutex_lock=UNLOCKED;
                                m->id_proceso_propietario=-1;
```

```
                        printk("Mutex %s DESBLOQUEADO\n",m->nombre);
                        if (m->num_procesos_esperando>0)
                        {
                                m->num_procesos_esperando--;
                                BCPptr p_proc_bloqueado = m->lista_procesos_esperando.primero;
                                p_proc_bloqueado->estado = LISTO;
                                eliminar_primero(&(m->lista_procesos_esperando));
                                insertar_ultimo(&lista_listos,p_proc_bloqueado);
                                printk("Proceso id: %d DESBLOQUEADO\n",p_proc_bloqueado->id);
                        }
                        fijar_nivel_int(n_interrupcion);
                        return desc_proc;
                }

        } else {  /* si el mutex no esta bloqueado el intento de desbloquearlo producira un error */
                printk("Error, intento de desbloquear mutex %s no bloqueado\n",m->nombre);
                fijar_nivel_int(n_interrupcion);
                return -1;
        }
        fijar_nivel_int(n_interrupcion);
        return 0;
}

/* llamada al sistema para cerrar mutex */
int cerrar_mutex(unsigned int mutexid){
        int n_interrupcion, desc_proc, pos_lista_mutex;
        int *retorno;
        unsigned int m_id = (unsigned int) leer_registro(1);
        if (m_id<NUM_MUT)
        {
                mutexid=m_id;
        }
        n_interrupcion = fijar_nivel_int(NIVEL_1);
        //printk("MUTEXID: %d\n",mutexid);
        retorno = buscarMutexPorID((int) mutexid);
        desc_proc = *retorno;
        pos_lista_mutex = *(retorno+1);
        if (desc_proc==-1)
        {
                printk("Error, mutex con mutexid: %d no encontrado\n",mutexid);
                fijar_nivel_int(n_interrupcion);
                return -1;
        }
        //pos_lista_mutex = p_proc_actual->descriptores[desc_proc];
        MUTEXptr m = &lista_mutex[pos_lista_mutex];
        //printk("Mutex %s encontrado\n",m->nombre);

        while (m->mutex_lock==LOCKED)
        {
                unlock(mutexid);
        }
```

```
		m->estado = LIBRE;
		m->contador_bloqueos=0;
		m->num_procesos_esperando=0;
		p_proc_actual->descriptores[desc_proc]=-1;
		p_proc_actual->num_descriptores_abiertos--;
		contador_lista_mutex--;
		printk("Mutex %s CERRADO\n",m->nombre);

		if (contador_lista_bloqueados_mutex>0)
		{
			contador_lista_bloqueados_mutex--;
			BCPptr p_proc_bloqueado = lista_bloqueados_mutex.primero;
			p_proc_bloqueado->estado = LISTO;
			eliminar_primero(&lista_bloqueados_mutex);
			insertar_ultimo(&lista_listos,p_proc_bloqueado);
			printk("Proceso id %d DESBLOQUEADO\n",p_proc_bloqueado->id);
		}


		fijar_nivel_int(n_interrupcion);

		return 0;
}
```

Modificaciones: kernel.c -> main – inclusion de llamada para inicializar la lista mutex

```
int main(){
		/* se llega con las interrupciones prohibidas */

		instal_man_int(EXC_ARITM, exc_arit);
		instal_man_int(EXC_MEM, exc_mem);
		instal_man_int(INT_RELOJ, int_reloj);
		instal_man_int(INT_TERMINAL, int_terminal);
		instal_man_int(LLAM_SIS, tratar_llamsis);
		instal_man_int(INT_SW, int_sw);

		iniciar_cont_int();				/* inicia cont. interr. */
		iniciar_cont_reloj(TICK);	/* fija frecuencia del reloj */
		iniciar_cont_teclado();		/* inici cont. teclado */

		iniciar_tabla_proc();		/* inicia BCPs de tabla de procesos */

		/* --------cosas añadidas-------- */
		iniciar_lista_mutex();		/* inicia lista_mutex del sistema */

		/* crea proceso inicial */
		if (crear_tarea((void *)"init")<0)
			panico("no encontrado el proceso inicial");

		/* activa proceso inicial */
		p_proc_actual=planificador();
		cambio_contexto(NULL, &(p_proc_actual->contexto_regs));
		panico("S.O. reactivado inesperadamente");
		return 0;
}
```

kernel.c -> sis_terminar_proceso – se ha añadido lógica de programación para realizar cierre implícito de mutex

```
int sis_terminar_proceso(){
        /* comprobando si el proceso tiene mutex abiertos */
        int i = 0;
        while (p_proc_actual->num_descriptores_abiertos>0) {
                if (p_proc_actual->descriptores[i]!=-1) {
                        //printk("Cierre implicito de mutexid: %d\n",i);
                        cerrar_mutex(p_proc_actual->descriptores[i]);
                }
                i++;
        }
        printk("-> FIN PROCESO id: %d\n", p_proc_actual->id);
        liberar_proceso();
    return 0; /* no deberia llegar aqui */
}
```

kernel.c -> crear_tarea – inicialización de variables de procesos añadidas para mutex

```
static int crear_tarea(char *prog){
        void * imagen, *pc_inicial;
        int error=0;
        int proc, i;
        BCP *p_proc;
        proc=buscar_BCP_libre();
        if (proc==-1)
                return -1;          /* no hay entrada libre */
        /* A rellenar el BCP ... */
        p_proc=&(tabla_procs[proc]);
        /* crea la imagen de memoria leyendo ejecutable */
        imagen=crear_imagen(prog, &pc_inicial);
        if (imagen)
        {
                p_proc->info_mem=imagen;
                p_proc->pila=crear_pila(TAM_PILA);
                fijar_contexto_ini(p_proc->info_mem, p_proc->pila, TAM_PILA,
                        pc_inicial,
                        &(p_proc->contexto_regs));
                p_proc->id=proc;
                p_proc->estado=LISTO;
                p_proc->tiempo_dormir = 0;
                /* mutex */
                p_proc->num_descriptores_abiertos = 0;
                for (i = 0; i < NUM_MUT_PROC; i++)
                {
                        p_proc->descriptores[i]=-1;
                }
                /* lo inserta al final de cola de listos */
                insertar_ultimo(&lista_listos, p_proc);
                error= 0;
        }
        else
                error= -1; /* fallo al crear imagen */
        return error;
}
```

Sustituir el algoritmo de planificación FIFO por un Round-Robin

En kernel.h se añade el contador de ticks al BCP

```
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 */
          /* ----------cosas añadidas---------- */
          /* añadidos para la llamada dormir */
          int tiempo_dormir;                        /* tiempo que se queda el proceso dormido */
          /* añadidos para mutex */
          int descriptores[NUM_MUT_PROC];           /* conjunto de descriptores vinculados con los mutex usados
por el proceso */
          int num_descriptores_abiertos;            /* guarda el numero de descriptores abiertos por el proceso
*/
          /* añadidos para round-robin */
          int contadorTicks;
} BCP;
```

En kernel.c se crean las funciones actualizarTick y tratarIntSW

```
/* rutina para actualizar el contador de ticks */
void actualizarTick(){
          if (p_proc_actual->estado==LISTO) /* si hay proceso */
          {
                    /* si se acaban los ticks se activa interrupcion software */
                    if (p_proc_actual->contadorTicks>0)
                    {
                              /* decrementar contador de ticks */
                              p_proc_actual->contadorTicks--;
                              printk("Porceso id: %d, contador de ticks restantes: %d\n",p_proc_actual-
>id,p_proc_actual->contadorTicks);
                    }
                    if (p_proc_actual->contadorTicks==0)
                    {
                              printk("Proceso id: %d, activando interrupcion SW\n",p_proc_actual->id);
                              activar_int_SW();
                    }
          }
}
```

```
/* rutina para tratar interrupcion SW en round-robin */
void tratarIntSW(){
	if (lista_listos.primero==lista_listos.ultimo)
	{
		p_proc_actual->contadorTicks=TICKS_POR_RODAJA;
		printk("Proceso id: %d, contador de ticks actualizado\n",p_proc_actual->id);
	} else {
		BCPptr p_proc_expulsado = p_proc_actual;
		int n_interrupcion = fijar_nivel_int(NIVEL_1);
		eliminar_elem(&lista_listos,p_proc_expulsado);
		insertar_ultimo(&lista_listos,p_proc_expulsado);
		fijar_nivel_int(n_interrupcion);
		p_proc_actual = planificador();
		p_proc_actual->contadorTicks=TICKS_POR_RODAJA;
		printk("C.CONTEXTO POR EXPULSION de %d a %d\n",p_proc_expulsado->id,p_proc_actual->id);
		cambio_contexto(&(p_proc_expulsado->contexto_regs),&(p_proc_actual->contexto_regs));
	}
}
```

En kernel.c se incluyen las llamadas a las funciones anteriormente declaradas.
desde int_reloj() se llama a actualizarTick

```
static void int_reloj(){
	printk("-> TRATANDO INT. DE RELOJ\n");
	int n_interrupcion = fijar_nivel_int(NIVEL_3);
	/* procesos dormidos */
	cuentaAtrasBloqueados();
	/* round robin */
	actualizarTick();
	fijar_nivel_int(n_interrupcion);
   return;
}
```

y desde int_sw() se llama a tratarIntSW

```
static void int_sw(){
	printk("-> TRATANDO INT. SW\n");
	/* round-robin */
	tratarIntSW();
	return;
}
```

Dado que se trata de un documento de memoria y puede resultar algo incomodo leer tanto código dejo el enlace a mi repositorio de Github con el proyecto minikernel para su mas cómodo análisis.

https://github.com/yarosdytko/minikernel