

**Practica 1 Ampliación de Sistemas Operativos
Minikernel**

**Autor: Gabriel Morales Dato
Titulación: Doble grado en Ingeniería de
Computadores e Ingeniería Informática
Curso: 2023/2024**

Fecha: 31 de diciembre de 2023

Índice

1. Introducción	2
2. Llamada que bloquea a un proceso	2
3. Servicio basado en mutex	3
4. Algoritmo de planificación Round-Robin	9

1. Introducción

Esta práctica consisten en modificar una versión inicial de minikernel, incluyendo nuevas funcionalidades y añadiendo multiprogramación.

Se piden las siguientes modificaciones:

- Llamada que bloquea al proceso: introducir una llamada que bloquee un proceso durante x segundos, poniéndolo en estado bloqueado.
- Mutex: introducir un servicio de sincronización basado en mutex. Se pide realizar un mutex recursivo y otro no recursivo, así como métodos para crear, abrir y cerrar los mutex, y las primitivas lock y unlock.
- Round-Robin: sustituir el algoritmo de planificación FIFO por uno basado en Round-Robin.

2. Llamada que bloquea a un proceso

Para poder controlar y gestionar los procesos bloqueados he creado una lista de tipo lista_BCPs llamada lista_bloqueados. Esta lista contendrá todos los procesos en estado bloqueado.

Para este apartado he creado dos métodos:

- dormir: este método se encarga de cambiar el estado de proceso actual a bloqueado y asignarle el tiempo que tiene que dormir. Este tiempo se guarda en la variable segundos_dormir de la estructura BCP. Después mueve el proceso actual de la lista de procesos listos a la lista de procesos bloqueados, y llama al planificador para que asigne un nuevo proceso actual.

Antes de realizar estos cambios se guarda el nivel previo de interrupción para poder restaurarlo al terminar y realizar el cambio de contexto.

Código del metodo:

```
int dormir(unsigned int segundos) {
    int n_interr_previo = fijar_nivel_int(NIVEL_3);

    BCPptr dormido = p_proc_actual;

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

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

    p_proc_actual = planificador();

    /* Restaurar el nivel de interrupcion y realizar el cambio de contexto */
    fijar_nivel_int(n_interr_previo);
    cambio_contexto(&(dormido->contexto_regs),
                    &(p_proc_actual->contexto_regs));

    return 0;
}
```

- esperar_bloqueados: este metodo realiza el conteo de cuanto tiempo le queda a los procesos de la lista de bloqueados para dejar de estar bloqueados.

Cada vez que es llamado, itera sobre toda la lista de procesos bloqueados, restando una unidad a su variable segundos_dormir. Si esta variable es igual a 0, mueve el proceso de la lista de bloqueados a la lista de listos.

Código del método:

```
int esperar_bloqueados() {
    BCPptr actual;
    BCPptr siguiente;
    actual = lista_bloqueados.primeros;
    while (actual != NULL) {
        siguiente = actual->siguiente;

        actual->segundos_dormir--;
        if (actual->segundos_dormir <= 0) {
            actual->estado = LISTO;
            eliminar_elem(&lista_bloqueados, actual);
            insertar_ultimo(&lista_listos, actual);
        }

        actual = siguiente;
    }
    return 0;
}
```

3. Servicio basado en mutex

Para poder trabajar con mutex, he creado una estructura de datos para definirlos. Esta estructura contiene:

- nombre
- ocupado: si esta ocupado o no
- recursivo: si es recursivo o no
- proc_utilizado: puntero al proceso que esta utilizando el mutex.
- num_proc_esperando: numero de procesos que lo esperan.
- num_bloqueos: numero total de bloqueos.

Estructura de datos mutex:

```
typedef struct mutex_t {
    char nombre[MAX_NOM_MUT];
    int ocupado;
    int recursivo;
    BCPptr proc_utilizando;
    int num_proc_esperando;
    lista_BCPs lista_proc_esperando;
    int num_bloqueos;
} mutex;
```

También he modificado la estructura de datos BCP, que ahora contiene:

- descriptores[]: descriptores de fichero de los mutex que esta utilizando.
- num_descriptores

He creado dos listas nuevas:

- lista_mutex: contiene los mutex creados. Tiene tamaño máximo de NUM_MUT.

- `lista_bloq_mutex`: contiene los procesos bloqueados por un mutex.

Para la gestión de mutex he creado los siguientes métodos:

- `crear_mutex`: crea un mutex y devuelve el descriptor de fichero para acceder a él. En primer lugar se lee el nombre y el tipo de mutex que se va a crear de los registros 1 y 2, respectivamente. Después se guarda el nivel de interrupción anterior. A continuación se comprueba que el nombre proporcionado sea válido, y que el mutex no exista ya con ese mismo nombre. Si es válido, se procede a crear el mutex, inicializando sus parámetros y guardando su dirección en la lista de mutex. Después se abre el mutex, devolviendo el descriptor de fichero.

En el caso de que no queden mutex disponibles, el proceso se bloqueará.

Código de `crear_mutex`:

```
int crear_mutex(char *nombre, int tipo) {
    mutex m;
    int posicion; /* Para buscar el hueco libre */
    char *nombreMutex = (char *) leer_registro(1);
    int tipoMutex = (int) leer_registro(2);

    int nivelInterrupcion = fijar_nivel_int(NIVEL_1);

    /* Si el nombre es muy largo, se corta a las 7 letras */
    if (strlen(nombreMutex) >= MAX_NOM_MUT) {
        nombreMutex[MAX_NOM_MUT] = '\0';
    }

    if (existe_mutex(nombreMutex) == -1) {
        /* El mutex no existe y se puede crear */
        posicion = buscar_hueco_lista_mutex();
        if (posicion >= 0) { /* Hay un hueco libre */
            /* Se crea el mutex para meterlo en la lista */
            strcpy(m.nombre, nombreMutex);
            m.ocupado = 1;
            m.recursivo = tipoMutex;
            m.proc.utilizando = NULL;
            m.num_proc.esperando = 0;
            m.num_bloqueos = 0;

            lista_mutex[posicion] = &m;

            int fd = abrir_mutex(nombreMutex);
            fijar_nivel_int(nivelInterrupcion);

            return fd;
        } else {
            /* No quedan mutex disponibles. El proceso se bloquea */
            BCPptr actual = p_proc.actual;

            actual->estado = BLOQUEADO;
            eliminar_elem(&lista_listos, actual);
            insertar_ultimo(&lista_bloq_mutex, actual);
            p_proc.actual = planificador();

            fijar_nivel_int(nivelInterrupcion);
            cambio_contexto(&(actual->contexto_regs),
                           &(p_proc.actual->contexto_regs));

            return -1;
        }
    } else {
        /* Ya existe un mutex con el mismo nombre */
        fijar_nivel_int(nivelInterrupcion);

        return -1;
    }

    return 0;
}
```

- `existe_mutex`: busca el mutex indicado por parámetro y devuelve su posición. En caso de que no exista devuelve -1.

Código de `existe_mutex`:

```

int existe_mutex(char *nombre) {
    int i;
    for (i = 0; i < NUMMUT; i++) {
        if (!strcmp(lista_mutex[i]->nombre, nombre)) {
            return i;
        }
    }
    return -1;
}

```

- `buscar_hueco_lista_mutex`: este método itera sobre la lista de mutex y comprueba si hay alguno que no este ocupado, devolviendo su posición. En caso de que todos estén ocupados devuelve -1.

Código de `buscar_hueco_lista_mutex`:

```

int buscar_hueco_lista_mutex() {
    int i;
    for (i = 0; i < NUMMUT; i++) {
        if (lista_mutex[i]->ocupado == 0) {
            return i;
        }
    }
    return -1;
}

```

- `abrir_mutex`: asigna el mutex indicado por parámetro al proceso actual. Para ello, primero comprueba que el proceso actual no haya alcanzado el número máximo de mutex utilizados, y que el mutex existe en el listado. En caso de que se permita la utilización de este mutex, se procede asignárselo al proceso actual y se devuelve el descriptor de fichero del mutex.

Código de `abrir_mutex`:

```

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

    int nivelInterrupcion = fijar_nivel_int(NIVEL_1);

    if (p_proc_actual->num_descriptores < NUMMUT_PROC) {
        posicion = existe_mutex(nombreMutex);
        if (posicion >= 0) {
            p_proc_actual->descriptores[p_proc_actual->num_descriptores] =
                posicion;
            p_proc_actual->num_descriptores++;
            fijar_nivel_int(nivelInterrupcion);

            return posicion;
        } else {
            /* No existe el mutex */
            fijar_nivel_int(nivelInterrupcion);
            return -1;
        }
    } else {
        /* El proceso no tiene fd libres */
        fijar_nivel_int(nivelInterrupcion);
        return -1;
    }

    return 0;
}

```

- `lock`: ejecuta la primitiva lock para el mutex indicado por parámetro. En primer lugar comprueba si el mutex existe. En caso afirmativo se procede a bloquear procesos hasta que el proceso actual sea el que esta utilizando el mutex. Después se comprueba si es recursivo, y si no lo es, se comprueba que el numero de bloqueos es 1 para retornar de la función con un valor

negativo. En caso de que sea recursivo o sea no recursivo, pero con mas bloqueos, se anota en el valor num_bloqueos de la lista del mutex.

Código de lock:

```
int lock(unsigned int mutexid) {
    int i;
    int existe;
    unsigned int id;
    int nivelInterrupcion;
    int nivelInterrupcion2;
    BCPptr actual;

    id = (unsigned int) leer_registro(1);
    nivelInterrupcion = fijar_nivel_int(NIVEL1);

    existe = 0;
    for (i = 0; i < NUM_MUT_PROC; i++) {
        if (p_proc_actual->descriptores[i] == id) {
            existe = 1;
        }
    }
    if (!existe) {
        /* El mutex no existe */
        fijar_nivel_int(nivelInterrupcion);

        return -1;
    }

    while (lista_mutex[id]->proc_utilizando != p_proc_actual &&
           lista_mutex[id]->proc_utilizando != NULL) {
        p_proc_actual->estado = BLOQUEADO;
        nivelInterrupcion2 = fijar_nivel_int(NIVEL3);
        eliminar_elem(&lista_listos, p_proc_actual);
        insertar_ultimo(&lista_mutex[id]->lista_proc_esperando,
                       p_proc_actual);
        lista_mutex[id]->num_proc_esperando++;
        fijar_nivel_int(nivelInterrupcion2);

        actual = p_proc_actual;
        p_proc_actual = planificador();
        cambio_contexto(&(actual->contexto_regs),
                       &(p_proc_actual->contexto_regs));
    }
    lista_mutex[id]->proc_utilizando = p_proc_actual;

    if (lista_mutex[id]->recursivo == NO_RECURSIVO &&
        lista_mutex[id]->num_bloqueos == 1) {
        /* El mutex no es recursivo */
        return -1; /* TODO: return -2 */
    }
    /* Si es recursivo o tiene mas bloqueos, se anota */
    lista_mutex[id]->num_bloqueos++;
    fijar_nivel_int(nivelInterrupcion);

    return 0;
}
```

- unlock: ejecuta la primitiva unlock para el mutex indicado por parámetro. Al igual que en lock, se comprueba primero que el mutex existe. A continuación nos aseguramos que el proceso que utiliza el mutex sea el actual, y le restamos una unidad a su número de bloqueos. Comprobamos cuantos bloqueos le quedan al mutex, y en caso de que le queden mas de 0, significa que el mutex es recursivo, por lo que fijamos nuevamente el nivel previo de interrupción y continuamos la ejecución. Si el mutex es no recursivo se revisa si quedan procesos esperando al mutex.

Código de unlock:

```
int unlock(unsigned int mutexid) {
    int i;
    int existe;
    unsigned int id;
    int nivelInterrupcion;
    int nivelInterrupcion2;
    BCPptr siguiente;

    id = leer_registro(1);
    nivelInterrupcion = fijar_nivel_int(NIVEL_1);

    existe = 0;
    for (i = 0; i < NUM_MUT_PROC; i++) {
        if (p_proc_actual->descriptores[i] == id) {
            existe = 1;
        }
    }

    if (!existe) {
        fijar_nivel_int(nivelInterrupcion);
        return -1;
    }

    if (lista_mutex[id]->proc_utilizando != p_proc_actual) {
        fijar_nivel_int(nivelInterrupcion);
        return -1;
    }

    lista_mutex[id]->num_bloqueos--;
    if (lista_mutex[id]->num_bloqueos != 0) {
        /* El mutex es recursivo */
        fijar_nivel_int(nivelInterrupcion);
        return 0;
    }
    lista_mutex[id]->proc_utilizando = NULL;

    if (lista_mutex[id]->lista_proc_esperando.primeros == NULL) {
        /* No quedan procesos esperando al mutex */
        fijar_nivel_int(nivelInterrupcion);
        return -1;
    }

    siguiente = lista_mutex[id]->lista_proc_esperando.primeros;
    siguiente->estado = LISTO;
    nivelInterrupcion2 = fijar_nivel_int(NIVEL_3);
    insertar_ultimo(&lista_listos, siguiente);
    fijar_nivel_int(nivelInterrupcion2);
    lista_mutex[id]->proc_utilizando = siguiente;
    fijar_nivel_int(nivelInterrupcion);

    return 0;
}
```

- **cerrar_mutex**: cierra el mutex indicado por parámetro. Primero se comprueba que el mutex existe. Después se comprueba si el proceso actual está utilizando el mutex, para así cerrarlo directamente, poniendo `proc_utilizando` a `NULL`. A continuación se libera marcándolo como no ocupado y ajustando las variables correspondientes. Si es recursivo, se liberan todos los procesos que estaban esperando para utilizar el mutex y se restablece el número de bloqueos. Si había algún proceso esperando en la lista de bloqueo de mutex, se saca de la lista y se pone como listo.

Código de cerrar_mutex:

```
int cerrar_mutex(unsigned int mutexid) {
    unsigned int id;
    int nivelInterrupcion;
    int nivelInterrupcion2;
    int i;
    int existe;
    mutex *m;
    BCPptr proc;

    id = (unsigned int) leer_registro(1);
    nivelInterrupcion = fijar_nivel_int(NIVEL_1);

    existe = 0;
    for (i = 0; i < NUM_MUTPROC; i++) {
        if (p_proc_actual->descriptores[i] == id) {
            existe = 1;
        }
    }
    if (!existe) {
        fijar_nivel_int(nivelInterrupcion);
        return -1;
    }

    m = lista_mutex[id];
    if (m->proc.utilizando == p_proc_actual) {
        m->proc.utilizando = NULL;
        m->num_proc.esperando = 0;
    }
    p_proc_actual->num_descriptores--;
    p_proc_actual->descriptores[i] = -1;
    m->ocupado = 0;

    if (m->recursivo == RECURSIVO) {
        /* Si es recursivo se liberan todos los procesos asociados */
        if (liberar_todos_mutex(m) == -1) {
            return -1;
        }
    }
    m->num_bloqueos = 0;
    if (lista_bloq_mutex.primeros != NULL) {
        proc = lista_bloq_mutex.primeros;
        nivelInterrupcion2 = fijar_nivel_int(NIVEL_3);
        eliminar_primeros(&lista_bloq_mutex);
        insertar_ultimo(&lista_listos, proc);
        proc->estado = LISTO;
        fijar_nivel_int(nivelInterrupcion2);
    }
    fijar_nivel_int(nivelInterrupcion);

    return 0;
}
```

- **liberar_todos_mutex**: método auxiliar que se utiliza en el método cerrar_mutex para liberar todos los procesos asociados a un mutex.

Código de liberar_todos_mutex:

```
int liberar_todos_mutex(mutex *m) {
    BCPptr actual;
    BCPptr siguiente;
    int nivelInterrupcion;
    int ok;

    actual = m->lista_proc.esperando.primeros;
    siguiente = NULL;
    while(m->lista_proc.esperando.primeros != NULL) {
        siguiente = actual->siguiente;
        nivelInterrupcion = fijar_nivel_int(NIVEL_3);
        eliminar_primeros(&m->lista_proc.esperando);
        insertar_ultimo(&lista_listos, siguiente);
        actual->estado = LISTO;
        fijar_nivel_int(nivelInterrupcion);
        actual = siguiente;
    }

    ok = 0;
    if (m->lista_proc.esperando.primeros != NULL) {
        ok = -1;
    }

    return ok;
}
```

4. Algoritmo de planificación Round-Robin

Este algoritmo se basa en calcular rodajas de tiempo y asignarlas a los procesos. Cada rodaja consta de unos ticks de duración. Estos ticks están predefinidos por el sistema, y es necesario conocerlos cada vez que a un proceso le llega su turno de ejecución. Para ello he creado una variable llamada ticksRestantes, a la que se asigna la constante TICKS_POR_RODAJA cada vez que se llama al planificador:

```
static BCP * planificador(){  
    /* Para round robin se asignan los ticks por rodaja */  
    ticksRestantes = TICKS_POR_RODAJA;  
    while (lista_listos.primeros==NULL)  
        espera_int(); /* No hay nada que hacer */  
    return lista_listos.primeros;  
}
```

Para comprobar si un proceso ha terminado su turno de planificación, he modificado el método int_reloj para que ahora se llame a un nuevo método llamado round_robin. Este método round_robin resta una unidad a la variable ticksRestantes y comprueba si los ticksRestantes son 0, en cuyo caso se activa una interrupción de software llamando a activar_int_SW.

Código de round_robin:

```
/* ROUND ROBIN */  
int round_robin() {  
    ticksRestantes--;  
    if (ticksRestantes <= 0) {  
        activar_int_SW();  
    }  
    return 0;  
}
```