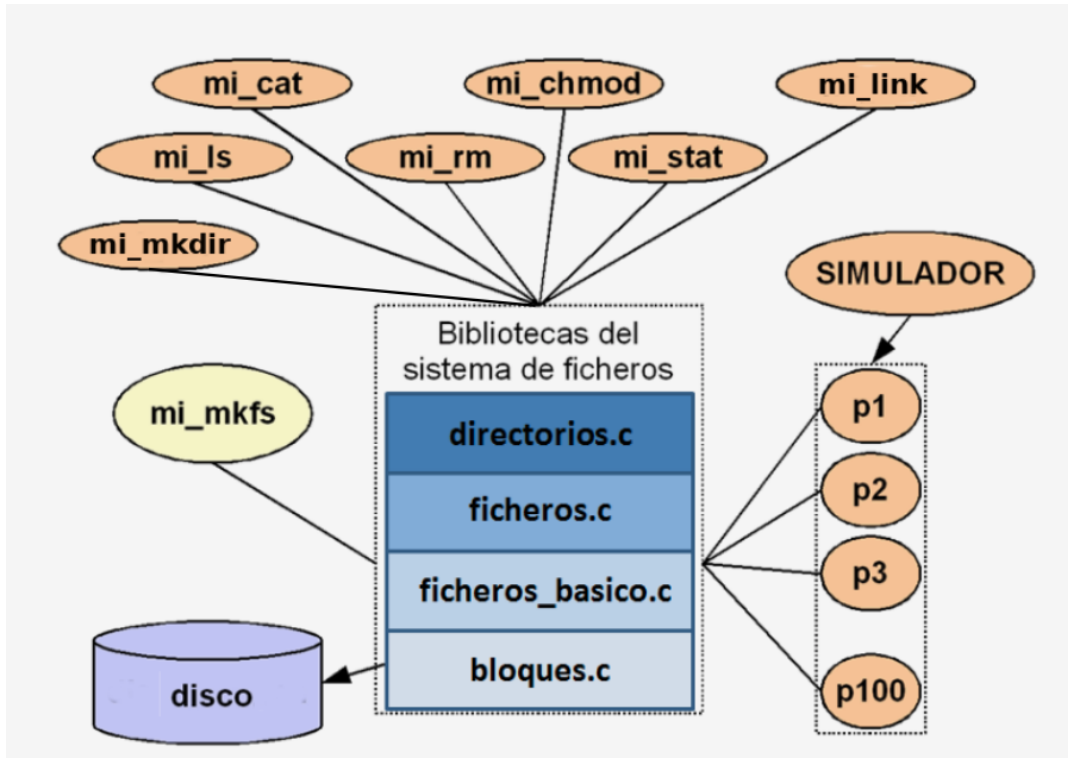


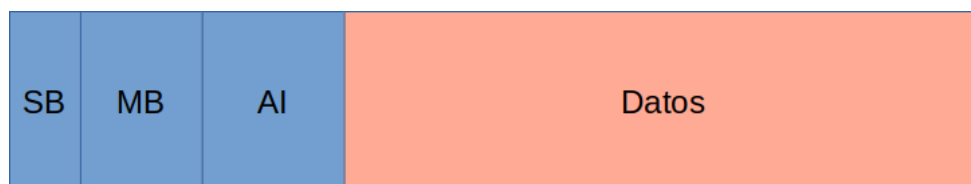
Nivel 11: semáforos y exclusión mutua



Para que el sistema de ficheros pueda ser utilizado concurrentemente por más de un proceso, hay que controlar, mediante un **semáforo** (exclusión mutua es suficiente), el **acceso concurrente a los metadatos**, es decir al **superbloque**, al **mapa de bits** y al **array de inodos**.

Esta responsabilidad es del sistema de ficheros, nunca de los programas cliente.

En cambio **no es responsabilidad del sistema de ficheros controlar el acceso concurrente a los bloques de datos**, sino de aquellos programas cliente que así lo deseen.



Las **secciones críticas** que tendremos que controlar serán las porciones de código donde:

- **se reserven o liberen inodos** (afecta a los campos `SB.posprimerInodoLibre`, `SB.cantInodosLibres`).
- **se reserven o liberen bloques** (afecta a los campos `SB.cantBloquesLibres` y al mapa de bits),

- **y cuando se modifiquen campos de los inodos** (tipo, permisos, sellos de tiempo, nlinks, tamEnBytesLog, numBloquesOcupados, punteros).

Lo más simple es realizar un **big lock**, es decir poner un **wait** al inicio de cada función implicada y un **signal** antes de cualquier salida de las mismas, pero hay que llevar especial cuidado con las funciones más frecuentes que son, en primer lugar el `mi_read()` y en segundo el `mi_write()`, para **no serializarlas**.

En **directorios.c**:

- **mi_creat()**: implica **reservar un inodo** para el fichero/directorio que queremos crear al llamar a `buscar_entrada()` y también puede implicar **reservar un bloque** al llamar a `mi_write_f()` para escribir la entrada de directorio, quien a su vez llama a `traducir_bloque_inodo()` con `reservar=1`.
- **mi_link()**: implica **reservar un inodo** al llamar a `buscar_entrada()` y **liberarlo** después. También puede implicar **reservar un bloque** al llamar a `mi_write_f()` para escribir la entrada de directorio del enlace. Y además se modifica la información del inodo (`nlinks`, `ctime`)
- **mi_unlink()**: Puede implicar **liberar un bloque** al eliminar una entrada de directorio llamando a `mi_truncar_f()`. Y también puede implicar **liberar un inodo y liberar sus bloques** si no había más enlaces. Además se modifica la información del inodo (`tamEnBytesLog`, `nlinks`, `ctime`)
- **mi_write()**: si no se sobrescribe implica **reservar bloques** al llamar a `traducir_bloque_inodo()` y por tanto se modifica `numBloquesOcupados` y el `ctime`. También se modifica la información del inodo: `mtime` y, si el fichero crece, también `tamEnBytesLog` y por tanto `ctime`.

- Se puede hacer más granular poniendo la sección crítica en `mi_write_f()` pero que sólo afecte a la porción de código donde se actualiza la información del inodo (leyendo primero de nuevo el inodo y escribiéndolo justo a continuación):

wait

leer_inodo

actualizar `ctime`, `mtime` (y `tamEnBytesLog` si ha variado el tamaño)

escribir_inodo

signal

Además se ha de poner una sección crítica que englobe la llamada a `traducir_bloque_inodo()` con `reservar=1` ya que eso implica **reservar bloques**.

- **mi_read()** si no se tiene en cuenta el `atime` no sería necesario poner sección crítica.

Si lo tenemos en cuenta hay que evitar serializarla:

- Se puede granularizar poniéndola en `mi_read_f()` pero que sólo afecte a la porción de código donde se actualiza la información del inodo (incluyendo

la seccion crítica **al principio de la función** de tal manera que sólo se lea el inodo una vez):

wait

leer el inodo

actualizar el **atime**

escribir el inodo

signal

En **mi_stat()** no hace falta semáforo.

En **mi_dir()** no hace falta semáforo si ya se utiliza en **mi_read_f()** cuando se actualiza **atime**.

En **mi_chmod()** no es necesario poner semáforo pero sí en **mi_chmod_f()** ya que se actualizan **permisos** y **ctime**.

Además para garantizar la consistencia del sistema de ficheros frente a caídas del sistema, habría que llevar cuidado en que el **orden de las acciones** de ciertas funciones (que deberían ser atómicas) sea correcto¹, por ejemplo:

- al borrar un fichero/directorio, eliminar la entrada de directorio antes de liberar el inodo,
- al truncar, marcar los punteros del inodo como "null" antes de liberar los bloques,
- al escribir, grabar 1º los bloques antes de modificar el tamaño del fichero.

Utilizaremos **semáforos POSIX con nombre** (ver código [semaforo_mutex_posix.h](#) y [semaforo_mutex_posix.c](#)).

Para linkar el semáforo al programa que lo va a utilizar hay que utilizar la librería **pthread**.

Ejemplo:

```
$ gcc -pthread programa.c semaforo_mutex_posix.c -o programa
```

En **semaforo_mutex_posix.h** se declaran las siguientes funciones:

```
sem_t *initSem();
void deleteSem();
void signalSem(sem_t *sem);
void waitSem(sem_t *sem);
```

¹ +info: [Consistencia y mantenimiento de un sistema de archivos de UNIX](#) (págs 33 a 35)

que a su vez llaman a las funciones de `<semaphore.h>`: `sem_open()`, `sem_unlink()`, `sem_post()`, `sem_wait()`.

El nombre de nuestro semáforo será `"/mymutex"` y estará inicializado a `1` (en `semaforo_mutex_posix.h`):

```
#define SEM_NAME "/mymutex" /* Usamos este nombre para el semáforo mutex */
#define SEM_INIT_VALUE 1 /* Valor inicial de los mutex */
```

En `bloques.c`:

```
#include "semaforo_mutex_posix.h"
```

- Utilizaremos una variable global para el semáforo:

```
static sem_t *mutex;
```

- Inicializaremos el semáforo desde `bmount()`:

```
if (!mutex) { // el semáforo es único en el sistema y sólo se ha de inicializar 1 vez (padre)
    mutex = initSem();
    if (mutex == SEM_FAILED) {
        return -1;
    }
}
```

- Eliminaremos el semáforo desde `bunmount()`:

```
deleteSem();
```

- Definiremos unas **funciones propias** para llamar a `waitSem()` y `signalSem()` (de esta manera todas las llamadas a las funciones de `semaforo_mutex_posix.c` estarán concentradas en `bloques.c`, y si cambiásemos el semáforo no habría que tocar el código del resto de programas):

```
void mi_waitSem() {
    waitSem(mutex);
}

void mi_signalSem() {
    signalSem(mutex);
}
```

En las funciones donde vayamos a definir las secciones críticas, habrá que realizar un **wait** y luego incorporar un **signal** en todas las posibles salidas de la función.

Ejemplo en `mi_creat()` de `directorios.c`:

```
int mi_creat(const char *camino, unsigned char permisos) {
    mi_waitSem();
    ...
    if ((error = buscar_entrada(camino, &p_inodo_dir, &p_inodo, &p_entrada, 1, permisos)) < 0) {
        mostrar_error_buscar_entrada(error);
        mi_signalSem();
        return -1;
    } else {
        mi_signalSem();
        return 0;
    }
}
```

Esquema de las llamadas a funciones de semáforos:

<semaphore.c>	semaforo_mutex_posix.c	bloques.c	directorios.c ficheros.c	comandos
sem_open()	← initSem()	← bmount()		≡
sem_unlink()	← deleteSem()	← bumount()		≡
sem_wait()	← waitSem()	← mi_waitSem()	≡	
sem_post()	← signalSem()	← mi_signalSem()	≡	

Cómo evitar que se haga el **wait** dos o más veces (código reentrante), por ejemplo para el `mi_creat()` que a su vez llama a `mi_read_f()`:

En `bloques.c`:

```
static unsigned int inside_sc = 0;
...
void mi_waitSem() {
    if (!inside_sc) { // inside_sc==0
        waitSem(mutex);
    }
}
```

```
    inside_sc++;  
}  
  
void mi_signalSem() {  
    inside_sc--;  
    if (!inside_sc) {  
        signalSem(mutex);  
    }  
}
```

Makefile para compilar con la librería `pthread`:

```
CC=gcc  
CFLAGS=-c -g -Wall -std=gnu99  
LDFLAGS=-pthread  
  
SOURCES=mi_mkfs.c bloques.c ficheros_basico.c leer_sf.c ficheros.c directorios.c  
mi_mkdir.c mi_touch.c mi_ls.c mi_chmod.c mi_stat.c mi_escribir.c mi_cat.c mi_link.c  
mi_rm.c semaforo_mutex_posix.c simulacion.c verificacion.c  
LIBRARIES=bloques.o ficheros_basico.o ficheros.o directorios.o semaforo_mutex_posix.o  
INCLUDES=bloques.h ficheros_basico.h ficheros.h directorios.h semaforo_mutex_posix.h  
simulacion.h verificacion.h  
PROGRAMS=mi_mkfs leer_sf mi_mkdir mi_touch mi_ls mi_chmod mi_stat mi_escribir  
mi_cat mi_link mi_rm simulacion verificacion  
  
OBJS=$(SOURCES:.c=.o)  
  
all: $(OBJS) $(PROGRAMS)  
  
$(PROGRAMS): $(LIBRARIES) $(INCLUDES)  
    $(CC) $(LDFLAGS) $(LIBRARIES) $@.o -o $@  
  
%.o: %.c $(INCLUDES)  
    $(CC) $(CFLAGS) -o $@ -c $<  
  
.PHONY: clean  
clean:  
    rm -rf *.o *~ $(PROGRAMS) disco* ext* res*
```