Università degli studi di Pisa

Facoltà di Ingegneria

Anno Accademico 1998/99

Tesi di Laurea in Ingegneria Informatica

Progetto e Realizzazione di un Filesystem Modulare per Sistemi Operativi Real-time

Candidato: Massimiliano Giorgi

Relatori: Prof. Ing. Paolo Ancilotti

Prof. Ing. Cosimo A. Prete

Dr. Ing. Giorgio C. Buttazzo

Ai miei genitori

*Desidero ringraziare tutti i membri del RETIS Lab.
della Scuola Superiore S.Anna per l'aiuto datomi
nella realizzazione del presente lavoro;
un particolare grazie al Prof. Giorgio Buttazzo
per i consigli e la disponibilità dimostrata.*

Capitolo 1

Introduzione

Il termine filesystem di solito viene usato con due significati diversi: si può intendere il "sistema di gestione degli archivi", cioè la parte di un sistema operativo che si occupa di gestire le risorse necessarie all'input/output o si può intendere una particolare organizzazione degli "archivi", cioè dei files, fatta su un dispositivo di memorizzazione come un hard-disk. Nel seguito sarà comprensibile dal contesto a quale di questi due significati ci si riferisce.

1.1 Scopi

In questo lavoro non si intende creare/implementare/studiare un filesystem adatto a sistemi real-time ma si intende costruire un sistema modulare nel quale si possano facilmente implementare/testare nuovi filesystems per studiare il loro comportamento in sistemi real-time con applicazioni reali.

Per raggiungere questi scopi è utile avere un tool per la misurazione delle caratteristiche temporali del sistema; per questo è stato sviluppato anche un tool per la misurazione dei tempi che verrà descritto nel capitolo 6; questo tool permette di tenere traccia degli "eventi" non solo di questo sistema, cioè del filesystem, ma anche di ogni altra parte del kernel per il quale è stato sviluppato.

Come applicazione reale per testare il comportamento del sistema si è usato il riproduttore MPEG-1 realizzato alla "The Regents of the University of California" del quale si è fatto il porting dal vecchio al nuovo kernel di *S.Ha.R.K.*.

1.2 Requisiti

I requisiti fondamentali per raggiungere questi scopi sono:

Modularità Tutto il sistema deve avere una struttura modulare; ogni modulo non deve conoscere la struttura interna degli altri moduli (principio dell' "information hidding") e perciò deve interagire con gli altri moduli in maniera controllata.

Standarizzazione Le interfacce fra i vari moduli interni al filesystem devono essere ben documentate in maniera da poter aggiungere nuovi moduli o da poter modificare/sostituire vecchi moduli con nuovi moduli più efficienti o con una diversa struttura interna. Verso l'esterno il filesystem deve fornire una interfaccia "standard" più eventualmente nuove interfacce sperimentali per poter testare nuove caratteristiche.

Compatibilià Posix Il sistema deve disporre di una interfaccia compatibile *Posix*per facilitare l'apprendimento/uso del sistema e per facilitare il porting di applicazioni già sviluppate sul sistema.

L'analisi dei sorgenti del sistema operativo *Linux*è stato un punto di partenza per organizzare questo lavoro; alcuni dei nomi di funzioni e strutture dati pur essendo uguali o simili a funzioni o strutture dati di *Linux*possono avere una semantica (e sicuramente una implementazione) profondamente diversa.

Questo sistema è stato realizzato utilizzando il linguaggio "C"; è stato portato e testato utilizzando il sistema operativo real-time *S.Ha.R.K.* sviluppato al RETIS Lab. della Scuola Superiore S.Anna di Pisa e ne è diventato una parte integrante.

Questo lavoro è cosi organizzato: nel Capitolo 1 è presente questa breve introduzione; nel Capitolo 2 viene descritta l'architettura generale del sistema, cioè come il sistema è stato scomposto e modularizzato, descrivendo brevemente anche l'interazione fra i vari moduli; nel Capitolo 3 viene descritta la parte di basso livello di questo sistema, cioè i device driver, vengono descritte le interfaccie fra i vari moduli, le politiche di schedulazioni delle richieste fatte ai device driver e l'unico device driver realizzato (IDE); nel Capitolo 4 viene descritta la parte di alto livello, cioè la gestione dei vari filesystem, come per il capitolo precedente, vengono descritte le interfaccie fra i vari moduli, le strutture dati utilizzate e l'unico filesystem implementato (FAT16); nel Capitolo 5 è presente una breve introduzione a *Posix*con particolare riferimento ai sistemi operativi real-time ed alla relazione fra questo sistema e lo standard; nel Capitolo 6 viene descritto il "tracer", cioè il tracciatore di eventi realizzato per *S.Ha.R.K.*, viene descritto sia da un punto di vista architetturale che da un punto di vista implementativo; nel Capitolo 7 sono descritti i risultati di alcuni test fatti.

Capitolo 2

Architettura del sistema

In questo capitolo si intende descrivere l'architettura di sistema, cioè come il sistema è strutturato, in quali e quanti moduli è stato suddiviso, quali sono le relazioni fra i vari moduli, confrontando questa struttura con quella di altri sistemi operativi.

2.1 Caratteristiche generali

Il sistema, come mostrato in Figura 2.1 nella pagina seguente, è strutturato in due grossi livelli:

- Una parte di alto livello (filesystems) contenente tutte le strutture dati per poter gestire gli archivi, cioè i files che sono risorse usate per l'input/output da parte delle applicazioni.
- Una parte di basso livello (blockdevices) che gestisce direttamente l'hardware dei dispositivi a blocchi come gli hard-disk e fornisce una interfaccia verso la parte di alto livello in maniera che questa "veda" i dispositivi a blocchi in maniera uniforme.

Sia la parte di alto livello che quella di basso livello sono strutturate in maniera modulare come mostrato in Figura 2.2 a pagina 5: ci sono dei moduli che implementano specifici filesystem (come FAT16) o specifiche interfacce fisiche (come l'interfaccia IDE) che vengono "mascherati" da una parte di uno strato superiore, il cui scopo è isolare i moduli (fornendo una interfaccia omogenea al resto del sistema) e fornire delle funzionalità generali ai moduli stessi.

Tutto il sistema si collega al resto del kernel mediante uno strato detto "glue" il cui scopo è di permettere diversi tipi di interazioni fra il sistema ed il kernel come spiegato successivamente; inoltre è presente uno strato di "wrappers" ad alto livello con analoghe funzionalità del "glue", cioè per permettere diversi tipi di interazione fra le applicazioni ed il filesystem.

Fra la parte di alto livello e quella di basso livello c'è uno strato che isola le due parti implementando una cache: questa cache, che può essere di varia natura, può essere anche esclusa; ci sono stati molti studi sull'opportunità di avere una cache (specialmente per il processore) in sistemi real-time da cui sono emerse due filosofie di pensiero: escludere la cache per avere un comportamento deterministico o usare la cache e dilatare i tempi computazionali (se il sistema non è critico), o calcolarli senza la cache ed usarli quali stime pessimistiche (se il sistema è critico).

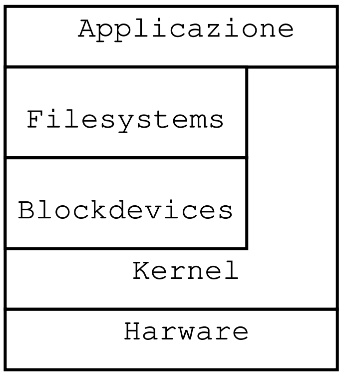

Figura 2.1: Architettura

Questa architettura è tipica, ad esempio viene implementata una architettura simile in *Linux*, perchè allora non si è deciso di fare semplicemente il porting di una architettura per l'I/O esistente? perchè queste architetture, progettate per sistemi time-sharing, hanno come finalità le performance, cioè cercano di massimizzare le prestazioni nel caso medio massimizzando l'efficienza a scapito della semplicità, mentre questo progetto ha come scopo la progettazione e l'implementazione di un filesystem modulare per sistemi real-time, nel quale gli obbiettivi fondamentali sono il determinismo temporale, cioè poter "assegnare tempi di calcolo ai task" e la modularità a più livelli, come specificato nella sezione 1.1 a pagina 1, cioè la facilità di inserimento di nuovi filesystems, nuovi device drivers e nuovi algoritmi di scheduling in maniera da poter comparare le prestazioni di applicazioni reali in varie condizioni.

2.2 Requisiti

Questa architettura per funzionare ha bisogno di essere supportata dal kernel; in particolare necessita di un sub-set delle funzionalità supportate dalla libreria "C" standard come le funzioni di manipolazione della memoria, di gestione delle stringhe, di accesso a funzioni con un numero di argomenti variabile e di classificazione dei caratteri.

Gli strati "glue" e "wrappers", descritti brevemente nella sezione 2.1 nella pagina precedente, servono per isolare le funzionalità specifiche del kernel. Naturalmente devono essere supportate dal kernel anche le seguenti funzionalità che possono essere emulate/implementate in specifici moduli del kernel:

- Meccanismi semaforici per la sincronizzazione e meccanismi per ottenere la mutua esclusione su regioni critiche.
- Meccanismi per l'acquisizione o il rilascio di risorse di basso livello, come lo spazio di I/O o gli interrupt.
- Meccanismi per la protezione della memoria, cioè meccanismi che consentano ad un processo privilegiato di accedere a diversi spazi di indirizzamento nella memoria.

In particolare le uniche funzionalità veramente necessarie sono quelle relative ai meccanismi di sincronizzazione e di mutua esclusione che vanno emulate/implementate

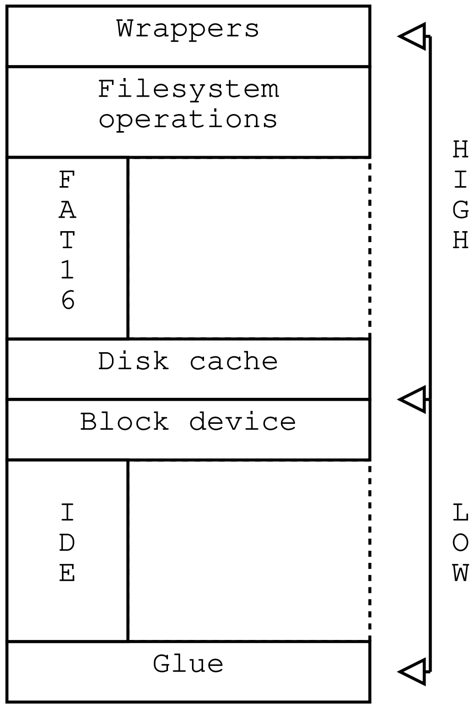

Figura 2.2: Architettura - dettagli

mentre le altre possono essere omesse: ad esempio se il sistema operativo non supporta la protezione della memoria e quindi ogni processo "vede" tutta la memoria, i meccanismi per la protezione possono essere implementati tramite funzioni che o non fanno nulla o utilizzano le funzionalità elementari fornite dal sistema operativo: se non c'è protezione della memoria, ad esempio, la funzione *__verify_write()* può ritornare sempre successo, la funzione *__copy_from_user()* può essere implementata come una macro che chiama la funzione *memcpy()*.

2.3 Basso livello: device drivers

Il basso livello, come mostrato nella Figura 2.3 nella pagina seguente, è strutturato in due parti:

- Una parte generale che esporta i servizi del basso livello verso l'esterno e che fornisce una serie di routine generali ai vari drivers.
- Una parte composta dai vari device drivers, il compito dei quali è la gestione di una o più classi di periferiche; da sottolineare che i vari device drivers sono isolati e che quindi possono essere inseriti o rimossi a tempo di compilazione del sistema.

Questa struttura modulare consente di poter inserire nuovi drivers (vedi la sezione 3.1.3 a pagina 16) in maniera semplice, consente inoltre di poter fare il porting di device drivers da altri sistemi operativi in maniera quasi indolore.

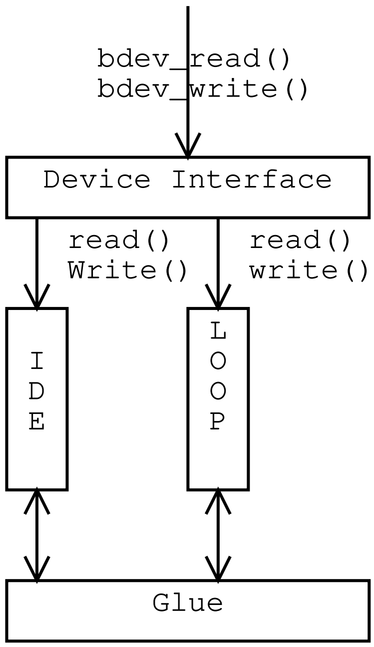

Figura 2.3: Architettura - Basso Livello

Volendo scendere ulteriormente più in profondità, come mostrato in Figura 2.4 nella pagina seguente, la parte generale è composta da tre sottomoduli e da un sottomodulo variabile (cioè che può essere sostituito):

Block device La parte principale, che si occupa di interfacciarsi verso l'esterno, dell'inizializzazione del sottosistema e di registrare tutti i driver presenti nel sistema.

Logical disk Fornisce ai vari device drivers delle funzionalità utili per trovare la struttura logica di un disco rigido, cioè l'inizio, la fine, le dimensioni ed il tipo di ogni partizione presente sull'hard-disk.

Physical disk Mantiene una base di dati contenente le caratteristiche fisiche di tutti gli hard-disk presenti nel sistema (i dispositivi fisici devono essere registrati dai vari device drivers); queste informazioni possono essere utilizzate dai moduli che si occupano di gestire le code delle richieste pendenti fatte al basso livello.

Queue policy É il modulo che gestice le politiche di gestione delle code di richieste nel basso livello; cambiando questo sottomodulo è possibile implementare qualsiasi tipo di gestione delle richieste, come spiegato nella sezione 3.3 a pagina 27.

2.4 Alto livello: filesystems

La parte di alto livello del sistema, mostrata in Figura 2.5 a pagina 8 ha una struttura più complessa di quella di basso livello; in particolare è divisa in tre sottostrati:

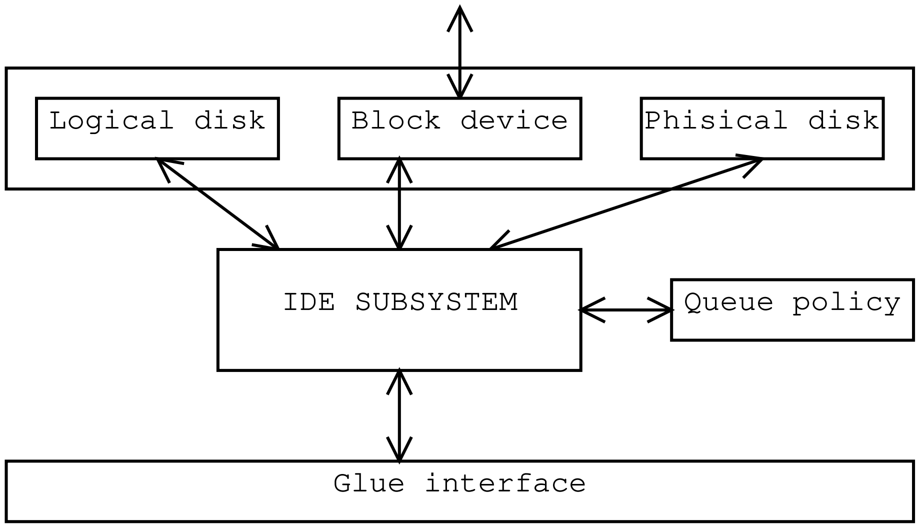

Figura 2.4: Architettura - Basso Livello - dettagli

- Una parte che gestisce i filesystems, l'inizializzazione del sistema, l'interfaccia con le routine esterne al kernel e che si interfaccia con i moduli che implementano i vari filesystems; i moduli di questo sottostrato richiedono servizi ai vari moduli di filesystems.
- Una parte composta dai vari moduli che implementano particolari filesystems che, a loro volta, possono essere strutturati in sottomoduli.
- Un modulo di cache che fa da interfaccia fra i moduli dei vari filesystem e la parte di basso livello; questo modulo può essere rimpiazzato (vedi la sezione 4.2.2 a pagina 62) in modo da poter gestire la cache con vari algoritmi, descritti nella sezione 4.2.2 a pagina 63, al limite per non avere una cache.

Il primo sottostrato è diviso in sottomoduli in maniera che sia possibile cambiare gli algoritmi interni ai vari sottomoduli in maniera trasparente rispetto al resto del sistema, ad esempio si potrebbe cambiare l'organizzazione interna del sottomodulo che gestisce gli inodes (sezione 4.2.1 a pagina 58) in maniera da memorizzare gli inodes non in liste doppie a chiave hash ma in un albero binario. Questi sottomoduli, mostrati nella Figura 2.5 nella pagina seguente, sono organizzati nella seguente maniera:

Filesystems É responsabile della inizializzazione e della terminazione di tutta la parte ad lato livello, registra tutti i filesystems presenti e contiene le routines per montare e smontare le partizioni.

File Gestisce le tabelle dei descrittori e i files; in particolare definisce ed organizza nella struttura *file*, descritta nella sezione 4.2.1 a pagina 60, tutte le informazioni necessarie all'utilizzo di un file.

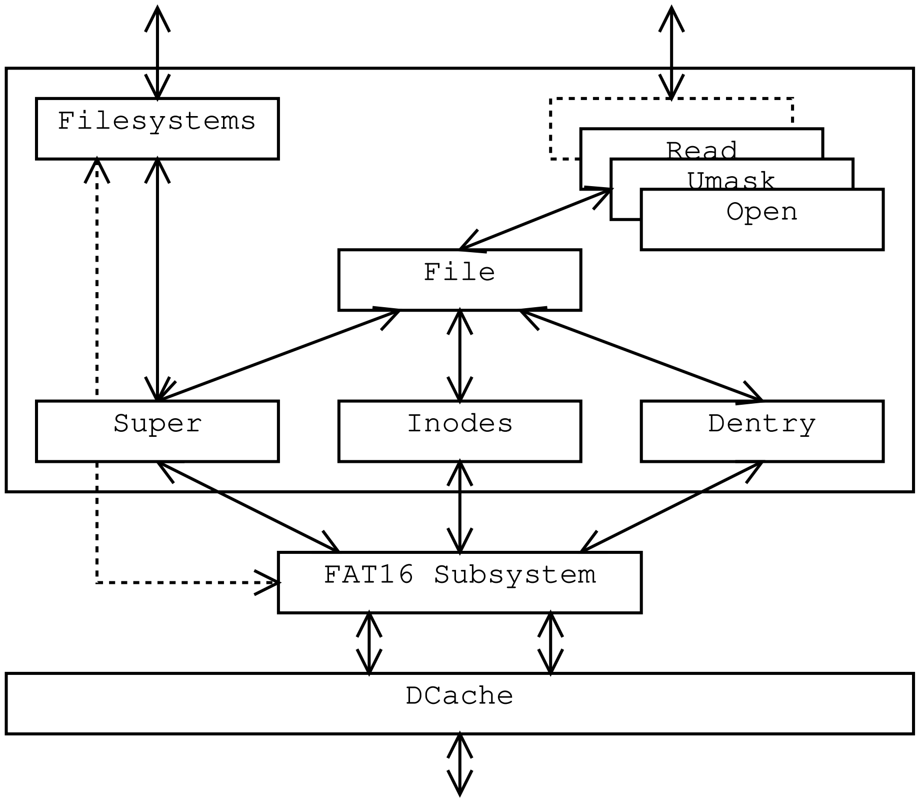

Figura 2.5: Architettura - Alto livello - dettagli

Super Gestisce i super blocchi (4.2.1 a pagina 53) cioè tutte le informazioni globali ad un particolare device fisico sul quale è presente un determinato filesystem.

Inodes Gestisce gli inodes (4.2.1 a pagina 58), cioè le informazioni che un filesystem possiede su un file.

Dentry Mantiene i nomi dei file organizzati in una struttura ad albero.

Open, read, umask, ... Implementano primitive di sistema; ad ogni primitiva di sistema corrisponde un file che la implementa utilizzando le informazioni dei vari sottomoduli o richiedendo servizi ai vari filesystems.

L'interfaccia fra i task utente e le primitive del filesystem viene realizzata dal modulo "wrappers", come già brevemente spiegato nella sezione 2.2 a pagina 4, questo modulo può essere facilmente modificato per consentire varie tipi di interazioni come spiegato nella successiva sezione.

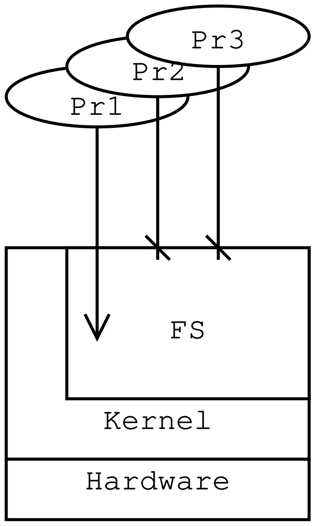

Figura 2.6: Interazione classica (1)

2.5 Interazione processi utente/sistema di I/O

L'interazione fra funzioni utente, cioè funzionalità per l'I/O presenti nelle librerie "standard", e sistema, cioè funzioni presenti all'interno del kernel, può avvenire in varie maniere; nel modello di interazione classico, mostrato in Figura 2.6, il filesystem è una parte di un kernel monolitico e quindi le primitive di sistema sono bloccanti per qualsiasi altra funzionalità offerta del kernel (il termine "primitiva" deriva proprio dal fatto che le funzionalità offerte dai kernel erano eseguite in mutua esclusione rispetto a tutte le altre funzionalità offerte dal kernel, erano operazione "atomiche" rispetto alla macchina astratta che il kernel mostrava ai processi utente).

L'interazione fra task utente e sistema può avvenire nel modo precedentemente descritto o in un modo più flessibile, usato da quasi tutti i sistemi operativi recenti: tutte le sezioni critiche, cioè le sezioni nelle quali viene letta/scritta una risorsa condivisa, possono essere isolate in maniera da ottenere un kernel multi-threads, mostrato in Figura 2.7 nella pagina seguente, nel quale cioè le primitive di sistema non vengono eseguite in mutua esclusione ma in maniera concorrente, anche se rimangono sezioni critiche che devono essere eseguite in mutua esclusione.

Il sistema può funzionare nelle due modalità precedentemente descritte perciò è necessario che ogni modulo aggiunto al sistema per implementare nuovi device drivers o nuovi filesystems sia "rientrante" (sia possibile, cioè, eseguire una routine quando un'altro task è stato bloccato, dallo schedulatore, all'interno di quella stessa routine) e multi-threads.

Sarebbe interessante, e anche di non difficile realizzazione, utilizzare l'approccio seguito dai sistemi a micro-kernel, come RT-Mach [1] o Nemesis [3], mostrato in figura 2.8 a pagina 11, dove il filesystem è visto come un task di sistema che fornisce le proprie funzionalità ai task utente mediante uno scambio di messaggi.

Un altro possibile approccio, mostrato in figura 2.9 a pagina 11, usato ad esempio in [14], è di "inglobare" il filesystem all'interno dei processi utente; il sistema che gestisce l'I/O è visto come una libreria da collegare a tutti i processi e che utilizza

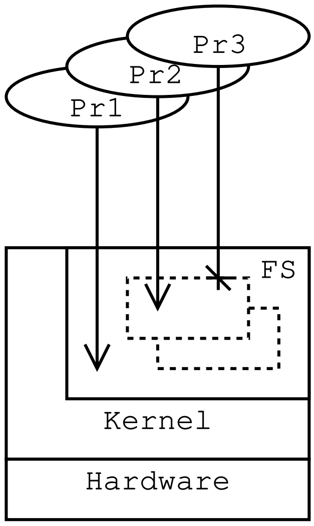

Figura 2.7: Interazione classica (2)

alcune funzionalità specifiche del kernel per coordinare le varie copie delle librerie; si potrebbe dire che viene utilizzato, in contrapposiszione al metodo mostrato in figura 2.8 nella pagina successiva, una interazione a memoria comune.

Per facilitare l'implementazione di nuovi modelli di interfaccia le chiamate di sistema vengono fatte attraverso una macro che può essere ridefinita; ad esempio la primitiva *open()*, definita nella libreria utente standard, chiama la macro *SYSCALL()* passandole come primo argomento il nome della funzione interna al kernel (*k_open()* in questo caso); attualmente questa macro espande in una chiamata alla funzione passata come primo argomento ma potrebbe essere modificata per spedire un messaggio ad un task (si potrebbe implementare una semantica simile alla "chiamata di procedura remota").

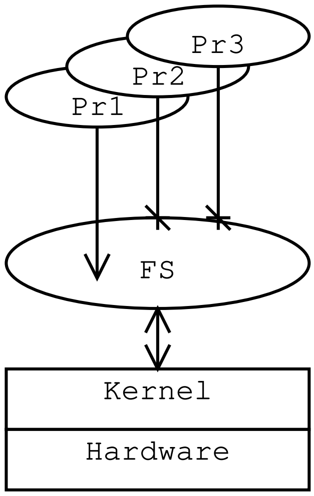

Figura 2.8: Interazione a scambio di messaggi

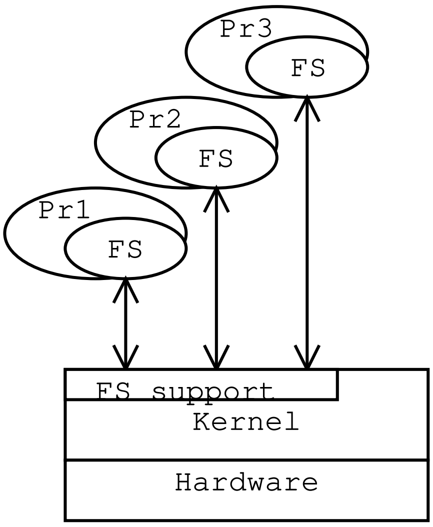

Figura 2.9: Interazione “fra task”

Capitolo 3

Device drivers

Un device driver è un modulo di programma responsabile della gestione di un dispositivo fisico; in questo capitolo si vuol mettere in luce l'inizializazzione di questo sottosistema, l'interfaccia con la parte di alto livello, l'interazione fra un generico device drivered il resto del sistema, come costruire un device driver per un sistema operativo real-time e come interfacciarsi con le politiche di scheduling delle code di richieste.

3.1 Interfaccia

Bisogna prendere in considerazione l'interfaccia fra la parte di alto livello ed il sottosistema che gestisce i device drivers e l'interfaccia fra questo sottosistema e i vari device drivers.

3.1.1 Inizializzazione

Per inizializzare la parte di basso livello è necessario chiamare la funzione *bdev_init()*, passandole una struttura *BDEV_PARMS*, mostrata nella Tabella 3.1 nella pagina seguente; il sistema dovrebbe usare le funzioni, mostrate nella stessa tabella, per manipolare i campi di questa struttura poichè potrebbero in futuro essere modificati.

Questa struttura è composta da due parti: una parte generale ed una parte specifica per poter passare parametri particolari ad ogni device drivers; i campi della struttura hanno il seguente significato:

showinfo Flag che se settato richiede al sistema di essere "verbose" durante l'inizializzazione del sistema; la macro *bdev_dev_showinfo* può essere usata per modificare questo flags.

config É una stringa non avente significato per la parte generale del basso livello ma che può avere un significato per i vari devices drivers; ad esempio potrebbe esserci un device drivers che ha necessità di un parametro numerico che vogliamo non specificare all'interno di un sorgente ma che vogliamo assegnare a run-time; potremmo fare in modo che questo drivers analizzi la stringa di configurazione per trovare un pattern del tipo "foo:[nnn]", dove "nnn" è il parametro numerico del quale necessita; questo campo dovrebbe essere impostato tramite la macro *bdev_def_configstring* per i motivi visti prima.

```
typedef struct bdev_parms {
  __uint32_t showinfo:1;
  char       *config;
  void       *bmutexattr;
  __uint16_t dummy;

#ifdef IDE_BLOCK_DEVICE
  IDE_PARMS ideparms;
#endif

} BDEV_PARMS;

void bdev_def_showinfo    (BDEV_PARMS s, int v);
void bdev_def_configstring(BDEV_PARMS s, char *v);
void bdev_def_mutexattrptr(BDEV_PARMS s,void *v);
```

Tabella 3.1: Struttura bdev_parms e macro

bmutexattr É un parametro che non ha significato per la parte di basso livello che è stato introdotto per facilitare l'integrazione del sistema con *S.Ha.R.K.*: questo sistema operativo mette a disposizione degli oggetti "mutex", per implementare la mutua esclusione, aventi varie semantiche che sono specificate attraverso una struttura certa dati; questo campo è un puntatore a questa struttura dati; le macro per manipolare i mutex, usati internamente al basso livello, possono usare questo parametro per gestire correttamete i mutex.

dummy Parametro senza significato utilizzato per allineare i campi all'interno della struttura per non far fallire l'espansione della macro *BASE_BDEV* spiegata più avanti.

La parte specifica della struttura *bdev_parms* è sotto il controllo ed ha significato solo per lo specifico device driver; la funzione *bdev_init()* può essere chiamata con un puntatore a tale struttura, che deve essere inizializzata assegandogli il valore di *BASE_BDEV*, o con il parametro NULL, nel qual caso sono usati i valori di defaults.

Per inserire un nuovo device drivers in questa architettura bisogna:

1. Aggiungere un nuovo numero "major" all'interno del file *include/fs/major.h*; vedi la sezione 3.1.2 nella pagina successiva per ulteriori spiegazioni.

2. Aggiungere un nuovo "#define" nel file *include/fs/bdevconf.h* in maniera da poter facilmente, a tempo di compilazione, decidere se compilare/collegare questo nuovo device driver.

3. Aggiungere, se è necessario, una struttura dati per l'inizializzazione all'interno della struttura *bdev_parms*, e modificare di consequenza la macro *BASE_BDEV* con i nuovi parametri di inizializzazione di default.

4. Modificare, in maniera condizionale, la fine della funzione *bdev_init()* in maniera che chiami la funzione di inizializzazione del device drivers alla quale viene passato un puntatore alla struttura *bdev_parms*.

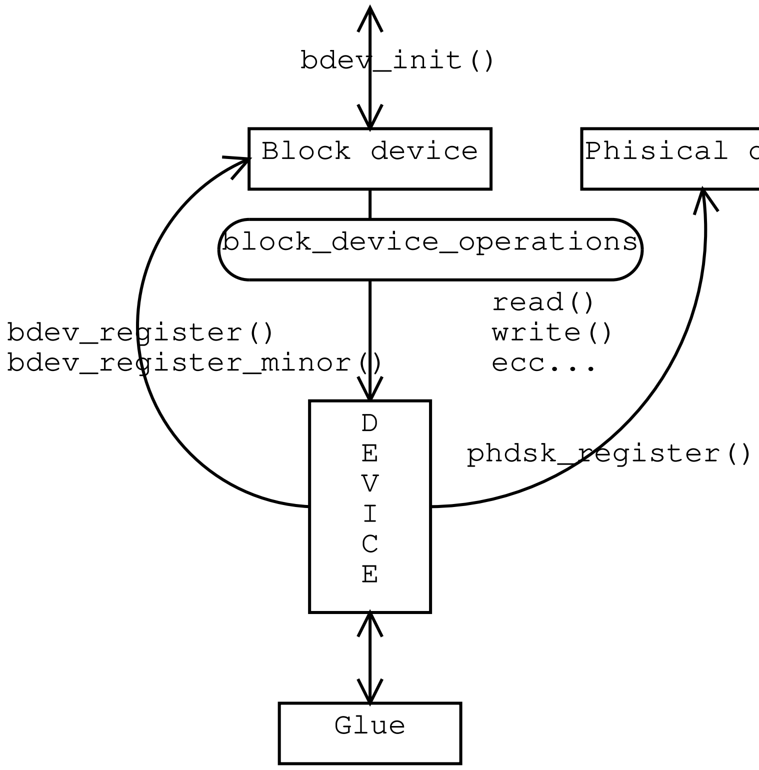

Figura 3.1: Inizializzazione dei device drivers

La funzione di inizializzazione del device driver deve compiere alcune azioni, come mostrato in Figura 3.1, oltre all'inizializzazione interna del device:

1. Deve registrarsi nel sistema chiamando la funzione *bdev_register()*; questa funzione deve essere chiamata per ogni tipo di dispositivo che il device driver intende gestire.

2. Deve registrare ogni dispositivo a blocchi (block device) che trova nelle interfaccie che deve gestire al modulo *Phisical disk* passandogli alcuni parametri.

3. Deve registrare ogni dispositivo logico che ha trovato chiamando la funzione *bdev_register_minor()*.

Le funzioni di registrazione nonchè le strutture dati usate durante la fase di inizializzazione sono descritte nella sezione 3.1.3 a pagina 16.

3.1.2 Device serial number

Ad ogni dispositivo logico, come una partizione logica di un hard-disk, viene associato un numero, di tipo *__dev_t*, che in terminologia *Posix*viene chiamato "device serial number"; le uniche caratteristiche che *Posix*richiede per questo numero è che sia unico, cioè che ad un device serial number corrisponda un ed un solo dispositivo logico.

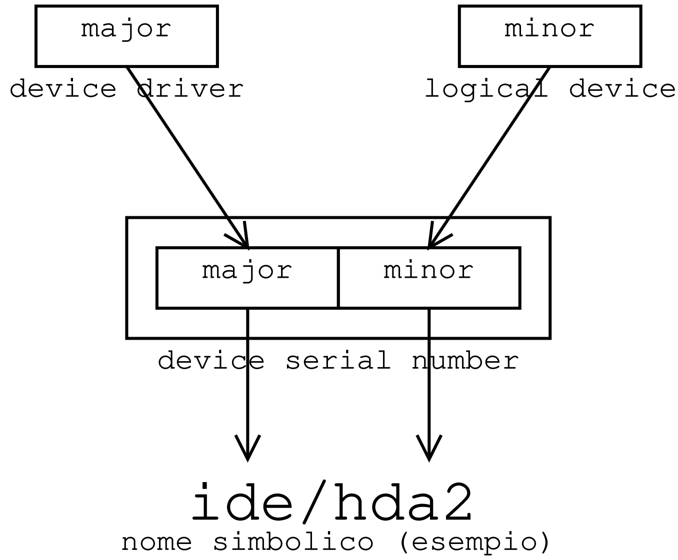

Figura 3.2: Device serial number

All'interno di questo sistema, come in ogni sistema *Unix*, questo numero ha una struttura interna, visibile nella Figura 3.2; è composto da due parti: una parte avente significato globale chiamato "major number", o più semplicemente "major", ed una parte avente significato solo per uno specifico device driver, chiamata "minor".

Esistono tre macro, i cui prototipi si trovano nel file *include/fs/sysmacro.h*, per manipolare questi device serial numbers:

makedev É una macro che accetta due numeri interi, il major ed il minor number, e restituisce un valore del tipo *__dev_t*, cioè un device serial number.

major Restituisce il major, cioè un numero intero, dato un device serial number.

minor Restituisce il minor dato un device serial number.

La parte di gestione generale dei device drivers usa il major per decidere a quale device drivers fare pervenire una richiesta come descritto nella sezione 3.1.3 nella pagina successiva; un device drivers può richiedere la gestione di diversi tipi di dispositivi a blocchi, ognuno contraddistinto da un diverso numero di major (i major si trovano elencati nel file *include/fs/major.h*); la gestione dei minor è lasciata al device driver, cioè è lasciata al driver la scelta dell'algoritmo di assegnazione dei minor ai dispositivi logici; da notare che la parte di gestione generale non controlla che un device esista, controlla solamente che ci sia un device driver che si è registrato per quella classe di dispositivi, cioè non controlla che la parte minor sia valida, cosa della quale deve assicurarsi il device drivers.

Per semplificare la "ricerca" di un determinato dispositivo, invece di implementare una serie di macro che ritornano il device serial number del dispositivo cercato, si è associato un nome simbolico ad ogni dispositivo, e si è fornito una serie di primitive, descritte dettagliatamente nella sezione 3.1.4 a pagina 22, per trovare questo device serial number. I nomi simbolici sono composti da due parti: un nome univoco assegnato

```
int bdev_register(__dev_t major, char *name,
    struct block_device *);
int bdev_register_minor(__dev_t device, char *name,
    __uint8_t fsind);

struct phdskinfo *phdsk_register(
       struct phdskinfo *disk);
```

Tabella 3.2: Device drivers - funzioni di registrazione

dalla parte generale ad ogni major, un nome specifico assegnato del device driver ad ogni specifico device; ad esempio il nome assegnato al major *MAJOR_B_IDE* è "ide", un nome di uno specifico device, fra quello controllati dal device driver che implementa l'interfaccia IDE, può essere "hda2", così con il nome "ide/hda2" si individua la seconda partizione dell'hard-disk master sulla prima interfaccia dei controllers IDE (vedi la sezione 3.4 a pagina 34).

3.1.3 Dettagli implementativi

Nella Tabella 3.2 sono mostrati i prototipi delle funzioni di registrazioni utilizzate dai device drivers:

bdev_register Questa funzione registra un tipo di device (vedi 3.1.2 a pagina 14); alla funzione viene passato il "major number" del device da registrare, un nome simbolico, usato per costruire i nomi simbolici delle periferiche, e un puntatore alla struttura *block_device*, il formato della quale viene spiegato più avanti; questa funzione ritorna zero in caso di successo e un numero diverso da zero in caso di fallimento.

bdev_register_minor Serve per registare ogni dispositivo logico trovato, usato solo per scopi informativi, ad esempio per implementare la funzione *bdev_finddevice_byname()* discussa nella sezione 3.1.4 a pagina 22; a questa funzione viene passato l'identificatore di device, il nome simbolico da assegnarle, e un valore che indica il tipo di filesystem presente su questo device (gli identificatori possono essere trovati nel file *include/fs/fsind.h*); questa funzione ritorna zero in caso di successo ed un numero diverso da zero in caso di fallimento.

phdsk_register Questa funzione registra ogni dispositivo fisico trovato da un device drivers con parametri che possono essere utili per implementare le politiche di schedulazione descritte nella sezione 3.2 a pagina 24; a questa funzione viene passato un puntatore ad una struttura *phdiskinfo*, contenete i parametri del dispositivo, e ritorna NULL in caso di errore o un puntatore alla struttura dati registrata (che può essere diverso da quello passato dall'utente).

La struttura *phdskinfo*, mostrata nella Tabella 3.3 nella pagina seguente, contiene i parametri fisici di un dispositivo; tutti i campi di questi struttura devono essere riempiti prima di chiamare la funzione *phdsk_register()*; in futuro altri campi potrebbero essere aggiunti, l'aggiunta di un nuovo campo comporta la modifica di tutti i device drivers o potrebbe essere usato un valore di default per i device driver non ancora aggiornati.

I campi di questa struttura hanno i seguneti scopi:

```
struct dskgeometry {
  __uint16_t    cyls;
  __uint16_t    heads;
  __uint16_t    sectors;
};

struct phdskinfo {
  char        pd_name[MAXPHDSKNAME];
  __dev_t     pd_device;
  __blkcnt_t pd_size;
  int         pd_sectsize;

  struct dskgeometry pd_phgeom;
  struct dskgeometry pd_logeom;
};
```

Tabella 3.3: La struttura phdsk

```
struct block_device {
  char                         *bd_name;
  int                          bd_sectorsize;
  __uint32_t                   bd_flag_used:1;

  struct block_device_operations *bd_op;
};
```

Tabella 3.4: Struttura block_device

pd_name Nome del dispositivo, non usato all'interno del sistema se non per scopi di "dump" dello stato del sistema.

pd_device Identificatore del device al quale queste informazioni si riferiscono.

pd_size Dimensione del dispositivo espressa mediante il numero di settori logici presenti sul dispositivo.

pd_sectsize Dimensione, in bytes, di un settore logico.

pd_phgeom e pd_logeom Queste due strutture mantengono la geometria del dispositivo; mantengono la geometria reale (o fisica) e la geometria logica, cioè quella vista dal sistema; le geometrie sono descritte mediante la struttura *dskgeometry*, mostrata nella Tabella 3.3, ed avente i seguenti campi:

  cyls Numero di cilindri (o traccie) del dispositivo.

  heads Numero di testine di lettura/scrittura presenti sul dispositivo.

  sectors Numero di settori presenti in ogni cilindro.

La struttura *block_device*, mostrata nella tabells 3.4, serve per registrare i dispositivi a blocchi:

```
struct block_device_operations {
  int (*_trylock)   (__dev_t device);
  int (*_tryunlock)(__dev_t device);
  int (*read)       (__dev_t device,
                    __blkcnt_t blocknum,
                    __uint8_t *buffer);
  int (*seek)       (__dev_t device,
                    __blkcnt_t blocknum);
  int (*write)      (__dev_t device,
                    __blkcnt_t blocknum,
                    __uint8_t *buffer);
};
```

Tabella 3.5: Struttura block_device_operations

bd_name Nome del dispositivo fisico; ad esempio, il device driver che gestisce l'interfaccia IDE si registra, ovviamente, con nome "ide".

bd_sectorsize Dimensione, in bytes, di un settore logico; questa informazione è ridondante ma utile in alcuni algoritmi.

bd_flag_used É usato dai moduli di basso livello generali e non dovrebbe essere utilizzato o modificato dai device drivers; se è settato significa che il corrispondente device driver è presente (per ottimizzare le performance viene usata una tabella per memorizzare tutti i possibili device drivers, quindi anche quelli non compilati o non presenti, alla quale si accede direttamente tramite un indice; questo campo permette di sapere immediatamente se un determinato device driver si è registrato).

bd_op Un puntatore ad una struttura statica contenete le operazioni "virtuali" dei device: scrivere un nuovo device driver significa implementare queste funzionalità.

Struttura block_device_operation

Questa struttura, mostrata in Tabella 3.5, contiene le funzionalità che devono essere fornite al sistema da ogni device driver:

read Questa funzione deve leggere sul dispositivo dato il blocco logico passato come parametro e deve scrivere queste informazioni sul buffer assegnato; deve ritornare 0 in caso di successo o un numero diverso da zero in caso di fallimento.

write Funzione duale della precedente: deve scrivere un blocco sul dispositivo.

seek Deve spostare la testina del dispositivo sul blocco richiesto senza effettuare nessuna operazione di lettura.

_trylock e _tryunlock Sono usate per provare a bloccare o sbloccare un dispositivo; ritornano zero in caso di successo o un numero diverso in caso di errore; dopo l'inizializzazione ogni dispositivo si deve trovare nello stato sbloccato, ad ogni bloccaggio effettuato con successo deve corrispondere uno sbloccaggio avente successo.

```
struct lodskinfo {
  __uint8_t  fs_ind;
  __blkcnt_t start;
  __blkcnt_t size;
};

typedef int (*lodsk_callback_func)(int,
                                   struct lodskinfo*,
                                   void *);

int lodsk_scan(__dev_t device,
               lodsk_callback_func func,
               void *data, int showinfo,
               char *lname);
```

Tabella 3.6: Il modulo "lodsk"

Cosa significa bloccare e sbloccare un dispositivo? il problema, descritto nella sezione 4.2.2 a pagina 63, deriva da un requisito richiesto dal modulo della cache di alto livello: questo modulo richiede che la coppia ¡numero dispositivo, numero settore logico¿ sia univoca cioè sia utilizzabile come chiave unica per individuare qualsiasi blocco; se ho due dispositivi logici che condividono alcuni settori potrei usare due chiavi per individuare quei settori; per evitare questo prima di "montare" un dispositivo logico la parte di alto livello prova a bloccare il dispositivo, se il blocco riesce allora il dispositivo passato come parametro alla funzione *_trylock()* è l'unico che può formare una chiave per quei blocchi a comune, se fallisce significa che ci sono blocchi a comune con un altro dispositivo che è attualmente bloccato (questi test vanno effettuati in mutua esclusione!). Se un device driver non ha dispositivi logici con blocchi condivisi allora le funzioni di bloccaggio e sbloccaggio possono ritornare sempre un successo. Può sembrare strano avere dispositivi con blocchi a comune, ma, ad esempio in *Linux*, i device /dev/hda e /dev/hda2 condividono dei blocchi: i blocchi di /dev/hda2 sono un sottoinsieme dei blocchi di /dev/hda.

Il modulo "lodsk"

Questo modulo, l'interfaccia del quale è mostrata nella Tabella 3.6, offre un servizio ai device drivers: è in grado di riconoscere la struttura logica, cioè le partizioni, di un dispositivo a blocchi (hard-disk) organizzato secondo l'architettura classica di un PC, descritta anche in [10]; una parte di questo codice è stata prelevata da *Linux*.

Prima di chiamare la funzione *lodsk_scan()* è necessario che il device driver che richiede il servizio si sia già registrato; questa funzione richiede cinque parametri:

1. L'identificatore del dispositivo sul quale effettuare la ricerca.
2. Una funzione di callback che viene chiamata per ogni partizione individuata.
3. Un puntatore a dati generici che non vengono usati ma vengono passati alla funzione di callback.
4. Un flag che indica se mostrare sul video la struttura del dispositivo analizzato.

```
typedef internal_sem_t __sem_t;

#define __sem_init    (ptr,val)
#define __sem_wait    (ptr)
#define __sem_trywait (ptr)
#define __sem_signal (ptr)
```

Tabella 3.7: L'interfaccia verso i semafori di sincronizzazione

5. Il nome del generico dispositivo logico, usato solo se il flag di visualizzazione è settato; ad esempio potrebbe essere "hdb".

Per ogni partizione logica la funzione di callback viene chiamata passandogli come argomenti:

1. Il numero logico della partizione trovata; viene usato lo schema di numerazione di *Linux*.
2. Una struttura contenente le informazioni sulla partizione trovata.
3. Il puntatore ai dati generici passato durante la chiamata alla funzione *lodsk_scan()*.

Le informazioni passate alla funzione di callback, come mostrato nella Tabella 3.6 nella pagina precedente, sono:

fs_ind Un numero che individua il filesystem presente sulla partizione; questi numeri sono elencati nel *include/fs/fsind.h*.

start Il numero del blocco a partire dal quale comincia questa partizione.

size La dimensione, in numero di blocchi, della partizione.

Le routine della parte di basso livello fanno sempre un controllo su questi valori prima di chiamare le routine specifiche di un device driver, in maniera da minimizzare la possibilità di una lettura, ma specialmente di una scrittura, fuori della partizione richiesta.

I semafori di mutua esclusione e di sincronizzazione

Come già descritto nella sezione 2.2 a pagina 4, questo sistema utilizza semafori di mutua esclusione e di sincronizzazione che devono essere forniti dal kernel; l'interfaccia che il sistema utilizza per utilizzare questi due tipi di oggetti si trova nei files *include/fs/mutex.h* e *include/fs/semaph.h* che sono usati sia dalla parte di alto livello che da quella di basso livello.

Questa interfaccia, mostrata nella Tabella 3.7, si compone di quattro macro e di una definizione di tipo (naturalmente al posto delle macro si possono usare funzioni funzionanti da wrappers verso le vere funzioni del kernel):

__sem_t È il tipo "semaforo di sincronizzazione", utilizzato per istanziare variabili usate per la sincronizzazione.

```
typedef internal_sem_t __mutex_t;

#define __mutex_init    (ptr,attr)
#define __mutex_lock    (ptr)
#define __mutex_trylock (ptr)
#define __mutex_unlock  (ptr)

typedef SYS_FLAGS __fastmutex_t;

#define __fastmutex_init   (ptr)
#define __fastmutex_lock   (ptr)
#define __fastmutex_unlock (ptr)
```

Tabella 3.8: L'interfaccia verso i semafori di mutua esclusione

__sem_init Inizializza un semaforo, passato mediante un puntatore, al valore positivo passato (un semaforo è "verde" quando il suo valore è diverso da zero, mentre è rosso quando il suo valore è nullo).

__sem_wait Primitiva semaforica "wait": aspetta che un semaforo sia "verde", ritorna dopo aver descrementato il contatore semaforico.

__sem_trywait Testa se un semaforo è verde: se è verde si comporta come la funzione *__sem_wait()* e ritorna il valore zero, altrimenti ritorna immediatamente ritornando un valore diverso da zero.

__sem_signal Incrementa la variabile semaforica: se un semaforo era rosso diviene verde, qualche task in attesa potrebbe essere svegliato.

I "mutex" sono semafori di mutua esclusione, di solito implementati mediante semafori binari; nel sistema vengono usati due tipi di mutex:

1. I mutex "fast", cioè veloci, quando si ha bisogno di garantire la mutua esclusione su regioni critiche composte da poche linee di codice.

2. I mutex "normali", non aventi nessuna restrizione.

Qeusta distinzione è fatta solamente per aumentare la performance del sistema; facendo il porting verso un sistema operativo si può dimenticare questa distinzione ed implementare nella stessa maniera i due tipi di mutex.

L'interfaccia, mostrata nella Tabella 3.8, è molto simile a quella dei semafori di sincronizzazione:

__mutex_t e __fastmutex ̂Per istanziare variabili di mutua esclusione.

__mutex_init e __fastmutex_init Per inizializzare un mutex; l'eventuale secondo parametro serve per passare ulteriori informazioni al kernel (vedi la descrizione del campo *bmutexattr* fatta nella sezione 3.1 a pagina 12).

__mutex_lock e __fastmutex_lock Chiamate prima di entrare in una regione critica.

```
int bdev_read  (__dev_t dev, __blkcnt_t blocknum,
                __uint8_t *buffer);
int bdev_write (__dev_t dev, __blkcnt_t blocknum,
                __uint8_t *buffer);
int bdev_seek  (__dev_t dev, __blkcnt_t blocknum);

int bdev_trylock(__dev_t device);
int bdev_tryunlock(__dev_t device);

__uint8_t bdev_getdeffs(__dev_t device);
__dev_t   bdev_find_byname(char *name);
__dev_t   bdev_find_byfs(__uint8_t fsind);

int       bdev_findname(__dev_t device,
                        char **majorname,
                        char **minorname);

int bdev_scan_devices(int(*callback)
                      (__dev_t,__uint8_t));
```

Tabella 3.9: Funzioni del basso livello

__mutex_unlock e __fastmutex_unclock Chiamate dopo essere usciti da una regione critica.

__mutex_trylock Tentativo di entrare in una regione critica: se ritorna zero il tentativo è riuscita, se ritorna un valore diverso il tantativo è fallito.

Da notare che all'interno del sistema non si utilizza direttamente questa interfaccia poichè si vuole rendere semplice la disabilitazione della mutua esclusione nel caso che questa non sia richiesta, ad esempio se si fa funzionare la parte di alto livello come un processo avente un proprio spazio di indirizzamento privato; per questo, ad esempio, invece di utilizzare direttamente la funzione *__mutex_lock()* si utilizzano le funzioni *__b_mutex_lock()* o *__fs_mutex_lock()*.

3.1.4 Interfaccia verso l'esterno

Il basso livello esporta due classi di funzioni verso l'esterno, mostrate nella Tabella 3.9: quelle per manipolare i device, cioè per richiedere un servizio ad un device drivers, che dovrebbero essere usate solo dalla parte del sistema di alto livello, e quelle per cercare/conoscere i device logici presenti in un sistema.

Le tre funzioni maggiormente utili per l'utente del sistema sono:

bdev_find_byname Questa funzione accetta un nome simbolico di device, descritto nella sezione 3.1.2 a pagina 14, e ritorna il suo "device serial number" se il device esiste, cioè è stato registrato nel sistema da un device drivers, altrimenti ritorna il valore zero.

bdev_find_byfs Questa funzione accetta un numero che rappresenta un identificatore di filesystems, che è possibile trovare nel file *include/fs/fsind.h* e ritorna il primo device che dovrebbe contenere quel filesystem, ritorna zero se non riesce a trovarene nessuno.

bdev_scan_device Questa funzione serve per ottenere un elenco di tutti i device presenti nel sistema; a questa funzione viene passato come argomento una funzione di callback che viene chiamata per ogni device presente nel sistema; questa funzione ritorna zero in caso di successo, meno uno in caso di fallimento o un valore diverso da zero (quello ritornato dalla funzione di callback) se la funzione di callback non ritorna zero; la funzione di callback viene chiamata passandogli un device serial number e un valore che specifica quale filesystem dovrebbe essere presente sul device.

Queste funzioni sono utili durante la fase iniziale del sistema quando, di solito, si deve montare il device logico contenente il "root" filesystem, vedi 4.1.3 a pagina 49, più eventuali altri device.

Le funzioni *bdev_getdeffs()* e *bdev_findname()* sono usate dalla parte di alto livello per:

bdev_getdeffs Conoscere il probabile filesystem presente su di un device logico.

bdev_findname Conoscere il nome logico di un device; a questa funzione devono essere passati device serial number da trovare e due puntatori a puntatori a stringa (non ad array di caratteri); se la funzione riesce ritorna zero ed imposta i due puntatori su dei buffer che contengono i nomi del major e del minor, se la funzione fallisce viene ritornato un numero diverso da zero.

Le restanti funzioni esportate dal basso livello servono per richiedere dei servizi ai device drivers; le seguenti funzioni controllano la correttezza degli argomenti passati come parametri e chiamano, eventualmente, le funzioni corrispondenti dei device drivers come descritto in 3.1.3 a pagina 18:

bdev_read Legge un blocco dal device corrispondente.

bdev_write Scrive un blocco nel device corrispondente.

bdev_seek Sposta la testina di lettura/scrittura nella posizione indicata.

bdev_trylock e bdev_tryunlock Cercano di bloccare o sbloccare un device; vedi la sescrizione fatta nella sezione 3.1.3 a pagina 18.

Tutte le richieste di servizio fatte mediante queste ultime funzioni possono bloccare il task chiamante per un tempo indeterminato; tutto il basso livello è stato pensato ed organizzato per essere rientrante ed adatto ad un sistema multithread, per questi motivi anche i vari device drivers dovrebbero essere realizzati in maniera da essere multi-threads, cioè in maniera da poter essere utilizzati da più task contemporaneamente.

Se un device drivers non è multi-thread è necessario utilizzare semafori di mutua esclusione all'inizio delle routine proprie dei device drivers in maniera che un solo task possa eseguire codice all'interno del device driver in un certo istante, poichè le routine di alto livello, essendo multi-threads, possono richiedere un servizio al basso livello prima che la precedente richiesta sia conclusa.

L'utilizzare semafori di mutua esclusione all'inizio delle routine del device drivers, però, causa un grave incoveniente: è l'algoritmo usato dal kernel per implementare i semafori di mutua esclusione che determina l'ordine di esecuzione delle richieste; non si possono, cioè, implementare politiche di scheduling per le richieste al basso livello ed è percio inutile interfacciare il drivers con uno degli algoritmi di scheduling delle richieste al disco, descritti nella sezione 3.2; di solito i semafori interni al kernel vengono implementati utilizzazndo una politica FIFO (first in first out), perciò utilizzare questo approccio e quasi equivalente ad utilizzare l'algoritmo FCFS, descritto nella sezione 3.3.1 a pagina 30.

La parte di basso livello è strutturata in maniera "verticale" [4]: questo significa anche che il tempo speso dal kernel per eseguire operazioni per i task è conteggiato come tempo usato dal task, si è fatto in modo che un task non debba eseguire certo codice arbitrario a causa di un altro task; come spiegato nel paragrafo successivo, ad esempio, la gestione delle interruzioni andrebbe fatta tramite task, le computazioni fatte per fornire un servizio ad un task andrebbero attribuite a quello specifico task.

Sarebbe opportuno realizzare i device drivers utilizzando un approccio, oltre che multi-threads, anche "a server": la parte di alto livello fa molte richieste di servizio alla parte di basso livello, queste probabilmente causeranno il bloccaggio di qualche task poichè ogni device driver può eseguire solo una richiesta alla volta; dopo che una richiesta è stata fatta all'hardware da parte delle routine del device driver e prima che questa richiesta sia completata, di solito, bisogna eseguire del codice; chi e quando deve eseguire questo codice? potremmo svegliare il task bloccato e fare eseguire il codice a questo task ma in questo modo questo codice sarebbe eseguito quando lo scheduler del sistema decide di eseguire il task sbloccato e molte interfaccie hardware richiedono una determinata tempistica per eseguire questo codice; potrebbe essere eseguito dalla routine di interrupt (di solito quando una interfaccia ha finito di servire una richiesta invia un segnale di interruzione) ma in questo modo verrebbe "rubato" del tempo per eseguire questa routine di interruzione e questo potrebbe portare problemi ai task in esecuzione che devono, essendo il sistema in tempo reale, eseguire il loro compito rispettando parametri temporali; l'approccio migliore è usare un task aperiodico, avente caratteristiche temporali impostate dal device driver, per eseguire questo codice quando lo scheduler di sistema gli dà il controllo, naturalmente dopo che è stato attivato dall'arrivo di una richiesta di interruzione.

3.2 Scheduling delle richieste di I/O

Gli algoritmi di scheduling normalmente usati per schedulare le richieste fatte ad un disco sono sensibilmente diversi da quelli usati per schedulare un processore, molti dei quali sono descritti in [5] a causa, principalmente, di due motivi:

1. Non è possibile fare preemption sulle richieste.

2. L'overhead per eseguire una richiesta non è trascurabile; da questo deriva che è importante l'ordine delle richieste.

Il primo punto significa che una volta che è stata richiesta la lettura/scrittura di una certa parte dell'hard-disk non è più possibile fare altre richieste fino a che la richiesta già fatta non sia stata eseguita o non sia fallita (a meno che il controller non abbia la capacità di gestire internamente una coda di richieste ma in questo caso la politica della gestione delle richieste è quella inglobata nel controller); anche se fosse possibile interrompere una richiesta per eseguirne un'altra questo non potrebbe rendere l'accesso al disco preemptionable poichè l'assunto fondamentale che fanno tutti gli algoritmi di scheduling preemptive del processore è che l'overhead sia trascurabile e che quindi sia possibile fare ripartire un task sul quale sia stata fatta preemption senza modificare il suo tempo di calcolo mentre il tempo necessario per eseguire una richiesta di accesso al disco dipende dalla storia passata delle richieste, cioè dallo stato dell'hard-disk (in particolare è importante conoscere la posizione attuale della testina di lettura scrittura).

Come già detto nel paragrafo precedente l'overhead per eseguire una richiesta non è trascurabile: il tempo necessario per leggere un certo numero di settori non dipende solamente dal numero di settori richiesto ma anche dalla posizione della testina di lettura, cioè dalla stato del disco.

Questo comporta che l'ordine con il quale viene deciso di effettuare gli accessi al disco è fondamentale: ad esempio se ho tre task che hanno un tempo di calcolo di 10ms e una dead-line relativa di 30ms, qualsiasi schedulazione decida di eseguire è fattibile, mentre se ho tre richieste di lettura dell'hard-disk di un solo settore l'ordine con il quale decido di effettuare le richieste è importante, ad esempio ordinando le richieste per numero di traccia crescente minimizzo il tempo medio di accesso (nei sistemi real-time di solito si è interessati al worst-case).

In letteratura sono stati proposti principalmente algoritmi che tentano di minimizzare un qualche parametro medio, solo recentemente sono stati studiati algoritmi che tentano di garantire la qualità del servizio di accesso all'hard-disk, qualità del servizio intesa non solo come minimizzazione del tempo medio di accesso ma come garanzia di un qualche parametro (come avere la possibilità di usare una certa percentuale del tempo disponibile del controller).

Gli algoritmi classici, usati in sistemi operativi non real-time, sono:

FCFS (First Come First Served) Le richieste vengono gestite in ordine FIFO, cioè in ordine di arrivo; questo algoritmo è usato, per la sua semplicità, in sistemi non multiprogrammati (come l'MS-DOS); le sue prestazioni sono pessime poichè non si tiene conto dello stato attuale del sistema.

SSTF (Shortest Seek Time First) Viene sempre eseguita la richiesta che minimizza il tempo di accesso tenendo conto dello stato attuale del sistema; poichè come stato del sistema si può considerare la posizione attuale della testina dell'hard-disk questo algoritmo sceglie la richiesta che porta la testina sulla traccia più vicina all'attuale; questo algoritmo migliora decisamente le prestazioni intese come tempo medio di accesso ma ha un grosso inconveniente: può introdurre "starvation", le richieste più vicine hanno sempre la precedenza sulle richieste più lontane quindi se continuano ad arrivare richieste vicine le richieste lontane possono subire un ritardo indefinito.

SCAN, LOOK, CSCAN, ... Ci sono molti algoritmi simili che sono varianti di una stessa idea: eseguimo le richieste in ordine crescente/descrescente di cilindro; questa classe di algoritmi non introduce starvation ed hanno buone prestazioni; ad esempio:

- **SCAN** Si "spazzola" la superficie del disco in una direzione eseguendo la richiesta più vicina alla direzione in cui si sta scandendo la superfice del disco; arrivati ad un estremo si comincia la scansione nella direzione opposta.
- **LOOK** Ci si comporta come in SCAN ma si può cambiare direzione di scansione se non ci sono altre richieste nella direzione corrente. Questi due algoritmi non introducono starvation se si interpreta una richiesta fatta sul cilindro corrente (quindi che avrebbe distanza zero) come se fosse una richiesta fatta nella direzione di scansione opposta all'attuale.

- **CSCAN (Circular-SCAN)** L'algoritmo SCAN favorisce i cilindri centrali rispetto a quelli estremi poichè serve una doppia scansione della superficie per ritornare ad un estremo ma nel frattempo siamo passati due volte sui cilindri centrali; in questo algoritmo non si favorisce nessun cilindro, si scandisce, come in SCAN, la superficie dell'hard-disk ma in una sola direzione, arrivati ad un estremo si ritorna al cilindro di partenza senza eseguire altre richieste, e si riprende a scandire nella stessa direzione.
- **CLOOK (Circular-LOOK)** Variante dell'algoritmo LOOK analoga alla CSCAN: si scandisce la superficie in una sola direzione, come in CSCAN, si può interrompere la scansione se non ci sono altre richieste pendenti nell'unica direzione di scansione usata, per ricominciare una nuova scansione dalla richiesta più lontana.

VSCAN(R) É un algoritmo parametrico: si scandisce il disco come in LOOK, però abbiamo un parametro (R), compreso fra zero ed uno, che indica la possibilità all'algoritmo di invertire la direzione di scansione; in particolare se R vale uno ci si riduce all'algoritmo LOOK, cioè si cambia direzione di scansione come specificato dal LOOK, più il parametro si riduce più si è disposti ad invertire la direzione di scansione per favorire una richiesta vicina che si trova nella direzione di scansione opposta all'attuale invece di proseguire la scansione per una richiesta più lontana; il parametro R indica quando si può invertire la direzione di scansione, al limite, per R uguale a zero, si può sempre invertirla, riducendo l'algoritmo ad un SSTF.

SPTF (Shortest Positioning Time First) É un SSTF nel quale si considera, oltre al cilindro corrente, anche la posizione all'interno del cilindro, scegliendo la richiesta che minimizza il tempo di posizionamento; per poterlo implementare si deve disporre di hardware adeguato; una variante (Aged SPTF) cerca di evitare la starvation insita in questo algoritmo "invecchiando" le richieste: il tempo di posizionamento effettivo viene ridotto in maniera proporzionale al tempo di permanenza della richiesta, cioè in base alla differenza fra il tempo attuale ed il tempo nel quale è stata effettuata la richiesta.

I primi algoritmi di scheduling per i dischi per sistem real-time sono stati pensati adattando algoritmi di scheduling per processori:

EDF (Earlier Deadline First Si eseguono le richieste in base alla deadline della richiesta stessa; quindi si ha un ordinamento che non considera minimamente lo stato attuale dell'hard-disk ma considera solamente caratteristiche temporali delle richieste; questo algoritmo è pessimo poichè per i dischi non sono valide due ipotesi: che il tempo di overhead sia trascurabile e che non sia importante l'ordine di esecuzione di richieste con la stessa dead-line assoluta, come già precedentemente esposto; questo algoritmo, pur introducendo un grande overhead, è ancora abbastanza diffuso a causa della sua semplicità e della non esistenza di un algoritmo universalmente accettato, ben studiato e stabile.

PSCAN (Priotiry SCAN) Si mantengono varie code e si schedulano le richieste della coda a più alta priorità tramite SCAN, passando, quando sono terminate le richieste ad alta priorità, a quelle delle code con priorità descrescente; naturalmente se durante la scansione di una di queste code arriva una richiesta a più alta priorità si passa subito alla coda di questa richiesta; è opportuno avere un basso numero di code, avere più di quattro code non incide sulle prestazioni globali del sistema.

SCAN-EDF (SCAN-Earlier Deadline First) Si tenta di integrare l'algoritmo EDF con l'algoritmo SCAN per cercare di avere prestazioni migliori; le richieste si schedulano secondo EDF, se una serie di richieste, come spesso accade, hanno la stessa dead-line, si schedulano utilizzando SCAN.

SCAN-EDF (variante) Un altro modo per cercare di integrare EDF con SCAN: si ordinano le richieste secondo EDF, cioè secondo la propria dead-line, e si schedulano le prime "n" richieste utilizzando SCAN.

Tutti questi algoritmi non garantiscono niente, cercano solo di ottimizzare le richieste da un punto di vista temporale abbassando inevitabilmente le prestazioni medie del sistema, prestazioni intese come disk bandwidth, tempo medio di risposta o latenza media; una nuova generazione di algoritmi sono allo studio, algoritmi che tentano di conciliare le prestazioni medie del sistema con i vincoli temporali propri di un sistema real-time, cercando anche di garantire una certa qualità del servizio.

Nel sistema RT-Mach (RealTime-Mach) si è implementato un'altra variante dell'algoritmo SCAN-EDF, descritta in [1], nel quale si tenta di organizzare le richieste di accesso al disco come una qualsiasi altra risorsa gestita dal kernel: prima di poter accedere al disco un task deve richiedere una certa "banda" del disco, c'è un algoritmo di "disk bandwidth reservation" che accetta o rifiuta le richieste; una volta che un task è riuscito a riservarsi una banda può fare richieste al sottosistema che gestisce gli accessi al disco; le richieste al disco vengono schedulate utilizzando una variante dell'algoritmo EDF, chiamata EDF/JIT (Earlier Deadline First/Just In Time), nel quale ad ogni richiesta viene associato uno "slack" temporale [20], cioè un ritardo massimo che la richiesta può subire senza violare la propria dead-line, utilizzando queste informazioni si cerca di ordinare le richieste secondo l'algoritmo SCAN.

In Cello [19] (Cello non è un sistema operativo ma una infrastruttura per l'accesso al disco) si struttura il sistema di accesso al disco in due livelli interdipendenti: una parte di basso livello nel quale si gestisce una coda delle richieste con algoritmo FCFS e nel quale si esportano informazioni temporali sulle richieste presenti nella coda ed una parte di alto livello nel quale sono presenti varie code ognuna gestita con algoritmo specifico ed alle quali viene riservata una certa banda "temporale", cioè una certa parte del tempo totale, o una certa banda "dimensionale", cioè una certa parte dei settori trasferiti in un periodo; in base a queste informazioni la parte di basso livello decide quale coda di alto livello attivare, la coda di alto livello attivata inserisce la propria prima richiesta in una certa posizione della coda di basso livello rispettando un algoritmo descritto in Cello in maniera da non violare i parametri temporali delle altre richieste già presenti in questa coda.

3.3 Algoritmi di scheduling presenti nel sistema

Attualmente nel sistema sono presenti cinque diversi algoritmi di scheduling intercambiabili a tempo di compilazione modificando uno switch presente all'interno del makefile; l'aggiunta di un nuovo algoritmo di scheduling richiede solamente l'aggiunta di un nuovo modulo.

Attualmente, per semplicità, tutte le code sono dello stesso tipo, cioè non è possibile avere code gestite con algoritmi diversi: ogni device driver deve usare un modello

```
int bqueue_init          (bqueue_t *);
int bqueue_numelements   (bqueue_t *);
int bqueue_removerequest (bqueue_t *);
int bqueue_insertrequest (bqueue_t *,
                          struct request_prologue *);

struct request_prologue *bqueue_getrequest(
                         bqueue_t *);
```

Tabella 3.10: Funzioni per gestire le code

di coda del quale non conosce nulla; se si volesse sarebbe possibile, ridefinendo il file header di ogni modulo di schedulazione, far si che un device driver utilizzi un certo algoritmo di schedulazione mentre un altro ne utilizza un altro, si potrebbe anche "virtualizzare" le operazioni agenti sulle code tramite puntatori (come fa il C++) e inserendo in ogni coda un campo che ne indica il tipo.

I device drivers presenti nel sistema dovrebbero utilizzare l'interfaccia presente nel file *bqueue.h*, mostrata nella Tabella 3.10, per gestire le code; la strttura *bqueue_t* è specifica di ogni algoritmo di scheduling, cioè ogni algoritmo di scheduling deve definire questa struttura secondo le proprie esigenze;queste funzioni hanno la seguente semantica:

bqueue_init Inizializza la coda di richieste passata come parametro; deve ritornare zero in caso di successo un altro valore in caso di fallimento.

bqueue_numelements Deve ritornare il numero di elementi presenti nella coda; funzione utile ma non necessaria per implementare alcuni device drivers.

bqueue_insertrequest La richiesta specificata deve essere inserita in coda secondo l'algoritmo prorio di quel tipo di coda; la funzione deve ritornare zero in caso di successo un numero diverso da zero in caso di fallimento; da notare che le richieste vengono passate tramite un puntatore alla struttura *request_prologue_t* 3.11 nella pagina successiva, cioè è necessario che ogni device driver definisca una struttura dati per memorizzare le richieste, che il primo campo di questa struttura sia di tipo *request_prologue* e che venga effettuato un cast a questa struttura prima di chiamare la funzione.

bqueue_getrequest Preleva la prima richiesta disponibile nella coda, cioè la richiesta che dovrà essere eseguita per prima; questa funzione restituisce un puntatore alla richiesta se ha successo e NULL se fallisce o se non vi sono richieste pendenti; le richieste prelevate non vanno rimosse dalla coda e devono rimanere sempre disponibili, cioè, una volta che una richiesta è stata prelevata, questa funzione deve ritornare sempre quella richiesta fino a quando non sarà esplicitamente rimossa.

bqueue_removerequest Rimuove la prima richiesta della coda, cioè quella che sarebbe ritornata dalla funzione *bqueue_getrequest()*; ritorna, come al solito, zero in caso di successo, un numero diverso da zero in caso di fallimento.

```
#define REQ_DUMMY  0
#define REQ_SEEK   1
#define REQ_READ   2
#define REQ_WRITE  3

struct request_prologue {
  unsigned sector;
  unsigned head;
  unsigned cylinder;
  unsigned nsectors;
  unsigned operation;
  /*--*/
  struct request_specific x;
};

#define request_prologue_t struct request_prologue
```

Tabella 3.11: La struttura request_prologue_t

Un device drivers, prima di poter chiamare la funzione *bdev_insertrequest()* per inserire una richiesta, deve riempire tutti i campi presenti nella struttura *request_prologue_t*, mostrati nella Tabella 3.11, poichè questi campi possono essere usati dall'algoritmo di scheduling:

sector È il settore che sul quale si desidera operare.

head È la testina con la quale effettuare l'operazione.

cylinder È il cilindro dell'hard-disk sul quale si trova il settore.

nsectors È il numero di settori che si desidera trasferire.

operation È un valore che indica quale operazione si intende fare; può valere:

- **REQ_DUMMY** Richiesta non specificata.
- **REQ_SEEK** Richiesta di spostamento della testina di lettura/scrittura sul cilindro specificato.
- **REQ_READ** Richiesta di lettura.
- **REQ_WRITE** Richiestadi scrittura.

È possibile aggiungere altri campi se sono necessari ad un certo algoritmo di scheduling? è possibile alla condizione di dover modificare tutti i device drivers presenti in modo che utilizzino questi campi aggiuntivi; altrimenti si può ricorrere all'approccio usato ad esempio dall'algoritmo EDF illustrato in 3.3.5 a pagina 33 che ha bisogno delle dead-line delle richieste.

Cosa è la struttura *request_specific* visibile nella Tabella 3.11? È una struttura dati che deve essere definita da ogni specifico algoritmo di scheduling e che contiene informazioni utili a questo algoritmo per gestire la richiesta, di solito per implementare le

```
typedef struct TAGfcfs_queue_t {
  struct phdskinfo        *disk;
  /**/
  __b_fastmutex_t         mutex;
  struct request_prologue *head;
  struct request_prologue *tail;
  int counter;
} fcfs_queue_t;

struct request_specific {
  void *next;
};
```

Tabella 3.12: La struttura fcfs_queue_t

strutture dati necessarie a creare la coda (con coda, in questo caso, si intende una struttura dati ordinata nella quale inserire le richieste, non che queste strutture dati debbano essere realizzate mediante code).

La struttura *bqueue_t*, che deve contenere una coda di scheduling, deve essere definita in ogni modulo che implementa una politica di scheduling; può contenere un numero qualsiasi di campi ma il primo campo deve essere un puntatore alla struttura *phdskinfo*, descritta in 3.1.3 a pagina 16 e mostrata nella Tabella 3.3 a pagina 17, poichè durante l'inizializzazione della coda il primo campo viene sempre impostato a questa struttura dati, contenente informazioni sull'hard-disk ed utilizzabili dagli algoritmi di scheduling per conoscere le caratteristiche degli hard-disk stessi.

Nelle successive sezioni si intende descrivere brevemente come un particolare algoritmo di scheduling è stato implementato e non descrivere l'algoritmo stesso, una descrizione del quale può essere trovato nella sezione 3.2 a pagina 24.

3.3.1 FCFS

Questa politica di sceduling viene implementata facilmente utilizzando la struttura dati mostrata nella Tabella 3.12:

mutex Un semaforo di mutua esclusione per accedere a questa struttura dati.

head e tail Due puntatori alla testa ed alla coda della lista che mantiene le richieste: le richieste vengono prelevate in testa e vengono inserite in coda.

counter Un contatore che conta le richieste presenti in questa struttura dati.

La struttura *request_specific* contiene un puntatore per poter collegare le richieste in una lista.

3.3.2 SSTF

Questa politica di scheduling viene implementata utilizzando due liste mostrate nella Tabella 3.13 nella pagina successiva:

mutex Il solito semaforo di mutua esclusione.

```
typedef struct TAGsstf_queue_t {
  struct phdskinfo        *disk;
  /**/
  __b_fastmutex_t         mutex;
  struct request_prologue *actual;
  struct request_prologue *lqueue;
  struct request_prologue *hqueue;
  int counter;
} sstf_queue_t;

struct request_specific {
  void *next;
};
```

Tabella 3.13: La struttura sstf_queue_t

actual La richiesta attuale cioè quella che verrà ritornata dalla funzione *bqueue_getrequest()*.

lqueue e hqueue Due liste di richieste, la prima ordinata in ordine discendente la seconda in ordine ascendente: quando arriva una nuova richiesta questa viene inserita in una delle due code a seconda del fatto che il cilindro della richiesta sia minore o maggiore del cilindro della richiesta attuale; quando si rimuove una richiesta la nuova richiesta attuale diventa la prima richiesta della coda che ha la distanza minore della richiesta attuale; questo algoritmo, come già ricordato, può introdurre starvation.

counter Il contatore delle richieste contenute complessivamente nelle due liste.

Anche in questo caso la struttura *request_specific* contiene un puntatore per poter collegare le richieste in una lista.

3.3.3 LOOK

Per poter realizzare questa politica di scheduling si utilizza la struttura dati mostrata nella Tabella 3.14 nella pagina seguente:

mutex Il semaforo di mutua esclusione.

dir È la direzione di scansione: zero significa ascendente mentre uno significa discendente.

queue Due liste ordinate rispettivamente in maniera ascendente e discendente: quando *dir* vale zero si utilizza la lista *queue[0]* per schedulare le richieste, analogamente quando vale uno; se una lista si svuota, si inverte il campo *dir* e si procede con la nuova lista, se non vuota; una richiesta si inserisce nella lista di scansione attuale se la sua posizione viene dopo/prima la posizione attuale altrimenti la si inserisce nell'altra lista.

counter Il solito contatore.

Questo algoritmo può essere implementato in due varianti:

```
typedef struct TAGlook_queue_t {
  struct phdskinfo        *disk;
  /**/
  __b_fastmutex_t         mutex;
  int                     dir;
  struct request_prologue *queue[2];
  int counter;
} look_queue_t;

struct request_specific {
  void *next;
};
```

Tabella 3.14: La struttura look_queue_t

```
typedef struct TAGclook_queue_t {
  struct phdskinfo        *disk;
  /**/
  __b_fastmutex_t         mutex;
  int                     act;
  struct request_prologue *queue[2];
  int counter;
} clook_queue_t;

struct request_specific {
  void *next;
};
```

Tabella 3.15: La struttura clook_queue_t

1. Si inseriscono le richieste che hanno la posizione uguale a quella attuale nella lista di scansione attuale: in questo modo si può introdurre starvation, poichè le richieste che hanno la stessa posizione sono privilegiate rispetto alle altre.

2. Le richieste sopra specificate si inseriscono nella lista che non si sta attualmente scandendo: si elimina la starvation al prezzo di un peggioramento delle prestazioni medie del sistema; questo è la scelta fatta per default, modificando un flag all'inizio del file *look.c* si può operare l'altra scelta.

3.3.4 CLOOK

Questo politica è realizzata utilizzando una struttura dati, visibile nella Tabella 3.15, i campi dei quali hanno lo stesso nome di quelli della struttura *look_queue_t* ma un significato diverso:

mutex Il semaforo di mutua esclusione.

```
typedef struct TAGbd_edf_queue_t {
  struct phdskinfo        *disk;
  /**/
  __b_fastmutex_t         mutex;
  struct request_prologue *queue;
  int                     inservice;
  int                     counter;
} bd_edf_queue_t;

struct request_specific {
  void *next;
  long dl;
};
```

Tabella 3.16: La struttura edf_queue_t

dir Indica quale lista si sta utilizzando per la scansione e non in quale direzione si sta effettuando la scansione poichè la scansione è sempre fatta in ordine ascendente.

queue Due liste ordinate in maniera ascendente; si utilizza la lista *queue[dir]* per la scansione fino a quando non si svuota, dopodichè si commuta *dir* e si procede sull'altra se non vuota; una nuova richiesta è inserita nella lista attuale se la sua posizione è successiva alla posizione attuale altrimenti viene inserita nell'altra lista.

counter Il contatore delle richieste complessive inserite.

Anche in questo caso c'è il problema della starvation: una richiesta che ha distanza nulla dalla richiesta attuale può essere inserita nella lista di scansione attuale, introducendo starvation, o nell'altra, peggiorando le prestazioni.

3.3.5 EDF

Questa è l'unica politica di schedulazione per sistemi real-time presente nel sistema; questa implementazione è specifica del sistema operativo real-time *S.Ha.R.K.*, cioè questo modulo non funziona, a differenza degli altri, in un sistema operativo diverso da *S.Ha.R.K.*; il perchè di questo è illustrato più avanti.

La struttura dati per implementare questa politica di schedulazione è mostrata nella Tabella 3.16:

mutex Il semaforo di mutua esclusione.

queue Una lista di richieste ordinata per dead-line crescenti.

inservice Un flag che viene settato quando una richiesta va in servizio (in maniera che un'altra richiesta non possa inserirsi in testa alla lista).

counter Il contatore delle richieste in coda.

```
typedef struct {
  RES_MODEL r;
  TIME dl;
} BDEDF_RES_MODEL;

#define BDEDF_res_default_model(res)
#define BDEDF_res_def_level(res,l)
#define BDEDF_res_def_dl(res,dl)
```

Tabella 3.17: Il modello di risorsa per le dead-lines

Come viene ricavata la dead-line della richiesta? Tramite una risorsa usata durante la creazione del task, per questo motivo questa politica di schedulazione funziona solo con *S.Ha.R.K.*; per farlo funzionare con altri OS basterebbe però definire una funzione di nome *bd_edf_getdl()* che restituisca la dead-line relativa delle richieste del task in esecuzione.

Attualmente si è scelto di utilizzare un modello di risorsa di *S.Ha.R.K.*, mostrato nella Tabella 3.17; per utilizzarlo basta definire, durante la creazione di un task, una variabile di tipo *BDEF_RES_MODEL*, inizializzarla mediante la macro *BDEF_res_default_model* ed assegnarle una dead-line relativa, che può essere diversa da quella del task, mediante la macro *BDEV_res_def_dl*; se non si setta una dead-line mediante tale macro o non si usa questo modello di risorsa per un task, la dead-line viene posta implicitamente all'infinito (perciò, se si utilizza questo modulo e non si settano le dead-line, questo algoritmo si riduce ad un FCFS).

3.4 Device driver: l'interfaccia IDE

3.4.1 Descrizione

L'interfaccia IDE è largamente utilizzata nell'industria, infatti ogni personal computer compatibile IBM ne possiede una; pur essendo nata altre 10 anni fà le sue specifiche sono in continua evoluzione a causa della sua enorme diffusione e perciò dei suoi bassi costi.

Una descrizione approfondita delle caratteristiche dell'interfaccia e delle sue caratteristiche può essere trovata in [9] o in [22].

Questa interfaccia permette di collegare su ogni controller, al massimo, due periferiche IDE e permette di comunicare con queste tramite il protocollo ATA o una sua estensione (ATAPI), descritto in [21]; di questo protocollo ne esistono, a tutt'oggi, cinque versioni ma per l'implementazione si è scelto di utilizzare la quarta versione del protocollo essendo più stabile anche se è ancora in evoluzione (la maggior parte delle periferiche IDE attuali sono conformi alle specifiche ATA-3).

Esistono due classi di periferiche che possono essere collegate ad una interfaccia IDE: le periferiche "a pacchetti", come i cd-rom, che devono, per essere conformi a queste specifiche (ATAPI-4), fornire una serie di funzionalità e le periferiche "normali", come gli hard-disk, che devono fornire un'altra serie di funzionalità per essere conformi ATA-4.

Rispetto alle versione precedenti del protocollo ATA, la versione 4 possiede le seguenti caratteristiche aggiuntive:

1. Supporto della modalità Ultra DMA, per collegamenti sul bus IDE ad alta velocità.
2. Integrazione con il protocollo ATAPI per la comunicazione con dispositivi lenti, aventi caratteristiche particolari come i cd-rom.
3. Possibilità di avere comandi "overlap", cioè i comandi che richiedono molto tempo per essere eseguiti possono rilasciare il bus IDE in maniera che il bus possa essere usato per l'altra periferica presente.
4. Possibilità di incodare i comandi, cioè di passare più comandi contemporaneamente ad una stessa periferica.
5. Gestione del risparmi energetico tramite una serie di comandi che sono in grado di portare la periferica in condizioni di basso consumo energetico.
6. Supporto ad un protocollo di sicurezza, per permettere l'accesso alle periferiche in maniera sicura.
7. Supporto ad una serie di funzionalità per il self-monitoring e la self-diagnostic delle periferiche.
8. Supporto per la memorizzazione su supporti di memorizzazione allo stato solido.
9. Supporto alle unità removibili.

Il documento ATA-4 [21] specifica 6 diverse classi di comandi aventi caratteristiche diverse (queste classi vengono chiamte "commands protocol" nella documentazione ufficiale):

PIO data in Comandi di ingresso utilizzando una tecnica "polled", nei quali lo spostamento effettivo dei dati viene effettuato dall'host, cioè dalla cpu del computer.

PIO data out Comandi di uscita in modalità "polled"

Non-data Comandi che non coinvolgono il trasferimento di dati.

DMA Comandi di ingresso o uscita nel quale i dati vengono trasferiti utilizzando il DMA, cioè l'accesso diretto alla memoria.

Packet Comandi che trasferiscono i dati a pacchetti in momenti disgiunti; questi comandi vengono utilizzati per la modalità overlapped accennata precedentemente.

DMA Queued Comandi che sfruttuno le capacità di gestire una coda di richieste multiple utilizzando la modalità DMA.

Il dialogo fra host e l'interfaccia IDE avviene mediante l'uso di 9 registri visibili dall'host nello spazio di indirizzamento di I/O e mediante un segnale di interruzione (che può essere disabilitato). Per i comandi funzionanti in modalità "polled" l'host scrive nei registri le informazioni necessarie e, se il comando è di uscita, riempie il buffer di uscita scrivendo in uno di quei 9 registri, a questo punto l'host fa partire il comando scrivendo in un particolare registro; bisogna notare che l'host si comporta sempre come master del bus IDE, che il bus non viene rilasciato fino al completamento del comando

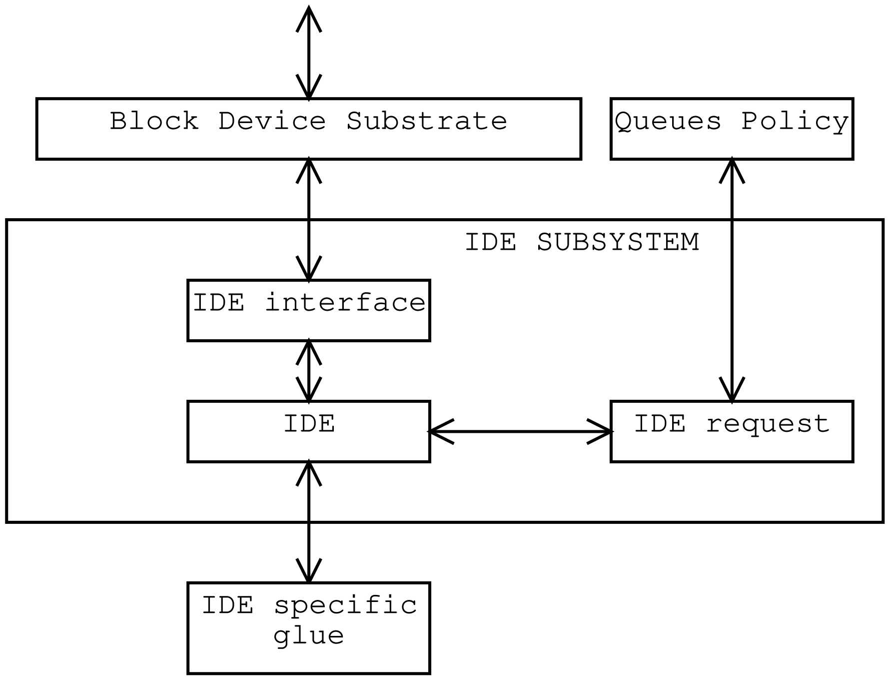

Figura 3.3: Il sottosistema IDE

e che l'host non ha la possibilità di abortire un comando; al completamento di questa fase viene generato un interrupt al quale l'host deve rispondere leggendo il registo di stato, per i comandi di uscita, leggendo questo registro e leggendo il buffer di ingresso, per i comandi di ingresso; a questo punto l'interfaccia è pronta per un nuovo comando; bisogna notare che una volta dato un comando ad una periferica non è possibile inviare un comando all'altra periferica. Per i comandi "non-data" si utilizza un protocollo simile però non si legge o scrive il buffer.

Per i comandi "DMA" l'algoritmo da seguire è simile, prima però bisogna preparare il controller dma al traferimento di dati, bisogna inserire le informazioni necessarie nei registri e quindi fare partire il comando; all'arrivo dell'interruzione bisogna leggere il registro di stato, i dati sono già stati trasferiti.

L'indirizzo del settore/settori da leggere/scrivere può essere fornito in due maniere:

1. Tramite il numero di cilindro, testina e di settore (indirizzamento C/H/S).
2. Tramite il numero di settore logico, come nei sistemi SCSI (indirizzamento LBA, Logical Block Address).

Il secondo metodo di indirizzamento è preferito, poichè più semplice, se la periferica lo supporta.

3.4.2 Implementazione

Il device driver che controlla l'interfaccia IDE è strutturato come illustrato nella Figura 3.3: c'è una parte di interfaccia il cui compito è di inizializzare il sistema, cioè ad

```
typedef struct ide_parms {
  void *parm_initserver;
} IDE_PARMS;

struct ide_server_model {
  TIME dl;
  TIME wcet;
  TIME mit;
};
```

Tabella 3.18: Le strutture dati per inizializzare il driver IDE

esempio di trovare le interfaccie IDE presenti nel sistema, trovare i device presenti e la loro struttura logica, e di trasformare le richieste fatte dalla parte generica di basso livello in comandi; c'è un modulo designato a gestire i comandi ed un modulo per interfacciarsi con il modulo che gestisce le politiche delle code di scheduling; c'è una parte specifica ad ogni sistema operativo e che serve per "incollare" questo driver ad un particolare sistema operativo e che quindi va riscritta se viene eseguito il porting verso altri OS.

Il driver cerca di utilizzare la modalità DMA, se possibile, comunque è possibile forzare l'uso della modalità polled, questo a causa dei problemi che un controller DMA causa ad un sistema real-time: il controller DMA ha la priorità sull'uso del bus di sistema, questo può causare ritardi imprevedibili nell'esecuzione di task aventi caratteristiche temporali precise.

La modalità DMA è supportata solo in modalita bus-master, modalità presente nei controller integrati nelle schede madri dei computer degli ultimi anni: questi controller hanno al loro interno un controllore DMA completo che una volta programmato, come specificato in [9], è in grado di diventare master del bus di sistema per effettuare i trasferimenti dati; la modalità più vecchia, nella quale c'è bisogno di un controller DMA sulla scheda madre che si interfaccia tramite alcuni segnali con il controller IDE che deve supportare la modalità DMA, non è supportata attualmente.

Attualmente il sistema usa solamente alcuni comandi delle classi "PIO data in", "PIO data out", "Non-data" e "DMA"; utilizza i comandi "read", "write", "seek", "dma read", "dma write", "identify", "pidentify" e "set feature"; i comandi di identify servono per identificare il device presente sul bus IDE mentre il comando "set feature" è utilizzato per impostare alcune nei device; i comandi a pacchetti non sono supportati per cui i device ATAPI, come i cd-rom, non sono attualmente supportati.

Inizializzazione

Il sistema viene inizializzato chiamando la funzione *ide_init()*, la chiamata a questa funzione viene effettuata all'interno della routine *bdev_init()* se è attivato il supporto IDE, quindi l'utente del sistema non è responsabile della chiamata di questa funzione; l'utente del sistema deve inizializzare la struttura *bdev_parms*, come specificato nella sezione 3.1 a pagina 12, impostando i campi necessari se vuole ottenere un comportamento diverso da quello di default.

Come mostrato in Tabella 3.18, per ora esiste solo un parametro all'interno della struttura *IDE_PARMS*:

```
void ide_glue_send_request         (int ideif);
int  ide_glue_activate_interface   (int ideif);
void ide_glue_unactivate_interface (int ideif);
```

Tabella 3.19: Il “glue” per il driver IDE

parm_initserver È un puntatore non specificato per l’inizializzazione; il puntatore è non specificato poichè dipende ed è utilizzato nella parte di “glue” del sistema e quindi è specifico ad ogni sistema operativo; questo parametro è stato introdotto proprio a questo scopo.

Questa struttura dati è presente nel campo *ideparms* della struttura dati generale di inizializzazione.

La parte dipendente dal sistema operativo del driver IDE è contenuta nel file *ide-glue.c* che contiene una implementazione delle funzioni mostrate nella Tabella 3.19, questa implementazione è relativa al sistema operativo *S.Ha.R.K.*:

ide_glue_activate_interface Deve attivare l’interfaccia passata come parametro: crea un task soft aperiodico, avente determinate caratteristiche temporali, e gli associa l’interruzione usata dall’interfaccia, cioè fa in maniera che all’arrivo di una richiesta di interruzione questo task sia attivato.

ide_glue_activate_interface Deve disattivare l’interfaccia: distrugge il task che è stato creato durante l’attivazione.

ide_glue_send_request Deve mandare in esecuzione la prima richiesta disponibile: attiva esplicitamente il task dell’interfaccia; quando il task viene attivato a causa di una interruzione il task finisce di eseguire la richiesta pendente e, se un’altra richiesta è in coda, attiva la nuova richiesta.

Il campo *parm_initserver* contiene, nell’implementazione del driver per *S.Ha.R.K.*, un puntatore alla struttura dati *ide_server_model*, mostrata sempre nella Tabella 3.18 nella pagina precedente, usata durante la creazione dei task delle interfaccie:

dl È la dead-line del task in microsecondi.

wcet È il tempo peggiore di esecuzione del server IDE in microsecondi.

mit È il tempo medio di esecuzione.

Se questi parametri non vengono specificati vengono usati dei valori di default tipici che non potrebbero causare problemi su computer lenti.

All’interno dei moduli che implementano il device driver vi sono dei flags che se definiti compilano il modulo in maniera particolare:

FORCE_SCANNING Se definito forza lo scanning, cioè la ricerca, delle periferiche sul bus IDE anche se il reset del bus fallisce; alcune periferiche non seguono le tempistiche specificate facendo credere al driver che l’interfaccia IDE non sia presente o che non vi siano periferiche sul bus.

IDE_OLDATAPIDELAY Alcune vecchie periferiche ATAPI, come vecchi cdrom players, sembra che non rispettino le tempistiche per la cancellazione di un interrupt pendente cosicchè dopo la cancellazione, ottenuta leggendo il registro di stato, bisogna fare un piccolo ritardo; questo ritardo viene attivato solo se il simbolo *IDE_OLDATAPIDELAY* è definito.

IDE_FORCERESET Con questo simbolo definito una operazione di soft reset non può venire abortita a causa di un timeout: se una periferica non rispetta le tempistiche previste per un soft reset o può andare in stand-by è necessario specificare questo simbolo.

IDE_SKIPSTATUSTEST Con questo simbolo definito non si testa che un comando stia procedendo nel modo corretto durante la sua esecuzione andando a leggere il registro di stato (con "durante la sua esecuzione" si intende mentre il driver sta scrivendo nei registri); tutte le schede madri base sul chipset TX che ho provato necessitano di questo simbolo.

IDE_SKIPALLTEST Con questo simbolo settato i comandi vengono eseguiti scrivendo nei registri senza fare test per aspettare dei cambiamenti nel registro di stato; questo simbolo implica il precedente; questa opzine è stata necessaria su una scheda madre basata sul chipset TX nella quale la modalità "polled" per i comandi non era affidabile.

Per settare o rimuovere questi flags di compilazione è necessario andare a modificare direttamente i sorgenti.

Interfaccia con le policy queues

Nella Tabella 3.20 nella pagina successiva è mostrata la struttura *idereq_t*, questa struttura è usata dal device driver per memorizzare una richiesta di servizio; si può notare che il primo campo di questa struttura,*info*, è di tipp *request_prologue_t* come richiesto dal modulo che gestice le politiche di schedulazione (vedi 3.3 a pagina 27).

Il driver dovrebbe creare una coda di richieste di servizio per ogni periferica e gestirla in maniera indipendentemente dalle altre ma, come ricordato precedentamente, l'interfaccia IDE permette di gestire un solo comando alla volta; questo pone un problema: se ho due periferiche connesse ad una sola interfaccia se decido di servire una richiesta su una certa periferica non posso servire richieste sull'altra periferica; tutti gli algoritmi di scheduling prendono in considerazione la gestione di una sola periferica e non considerano mai il fatto che la gestione di una periferica può bloccare quella di un'altra; per questi motivi pur potendo avere due code di richieste di servizio per una stessa interfaccia, queste code sono gestite, per semplicità, a priorità: la coda della prima periferica, la periferica master, ha sempre la precedenza sulla coda della seconda periferica, la periferica slave, di una stessa interfaccia IDE.

I campi, mostrati in nella Tabella 3.20 nella pagina successiva, contengono le seguenti informazioni:

info Per poter gestire le richieste tramite il modulo che gestisce la politica di schedulazione.

next Usato internamente al modulo di interfaccia per gestire un pool di richieste.

resetonerror Se questo flag è settato, un errore durante l'esecuzione di un comando provoca un reset software della periferica.

```
typedef struct TAGidereq_t {

  request_prologue_t info;

  int         next;

  __uint8_t   resetonerror;

  __uint8_t   cmd;
  __b_sem_t   wait;
  int         result;

  __uint8_t   features;
  __uint8_t   cyllow;
  __uint8_t   cylhig;
  __uint8_t   seccou;
  __uint8_t   secnum;
  __uint8_t   devhead;

  __uint8_t   *buffer;
} idereq_t;
```

Tabella 3.20: La struttura idereq_t

cmd Il comando da inviare alla periferica; i valorti possibili sono specificati nel documento [21].

wait Semaforo di sincronizzazione: quando un task ha "inviato" un comando, si pone in attesa su questo semaforo che il il server completi l'esecuzione della richiesta.

result È il risultato della richiesta: vale zero in caso di successo, un valore diverso da zero, che può essere trovato in una tabella interna al file *idelow.c*, in caso di errore.

features, cyllow, cylhigh, seccou, secnum e devhead Sono i valori da inserire nei registri di I/O dell'interfaccia IDE, hanno un significato diverso a seconda del tipo di comando e del fatto che si gestisca la periferica in LBA o in C/H/S (vedi 3.4.1 a pagina 34).

buffer Dove leggere/scrivere i dati se il comando comporta un trasferimento di dati.

I "minor number"

In questa sezione si vuol descrivere brevemente come sono gestiti i "minor number" all'interno di questo device driver per rendere più semplice l'implementazione di nuovi device drivers.

Come già descritto nella sezione 3.1.4 a pagina 22, la parte generale di basso livello decide a quale device drivers passare una richiesta utilizzando la parte "major" del device serial number; un device drivers deve individuare a quale specifico device inoltrare la richiesta utilizzando il "minor number".

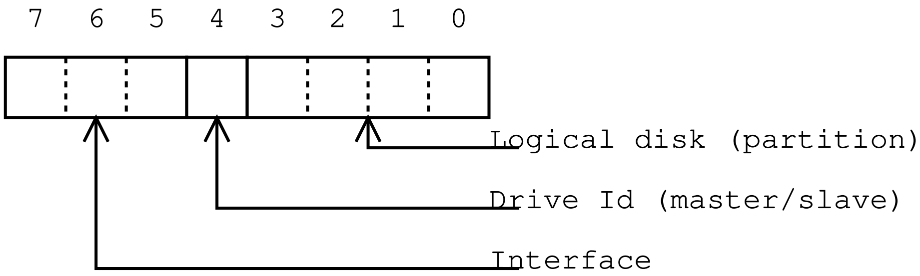

Figura 3.4: IDE "minor number"

Nella Figura 3.4 è mostrato come è strutturato internamente il minor number in questo specifico device driver:

- I primi tre bit indicano qual'è l'interfaccia, è possibile, perciò, gestire al massimo 8 interfaccie; ogni interfaccia è descritta in una struttura dati memorizzata all'interno di una tabella, perciò è possibile individuare una interfaccia utilizzando questa informazione come un indice all'interno della tabella.
- Un bit indica la periferica (ogni interfaccia IDE ha al massimo due periferiche per interfaccia): questo valore indica se la richiesta va inviata al master device od allo slave device di quella particolare interfaccia, cioè questo valore indica in quale coda delle richieste di quella particolare interfaccia inserire la richiesta.
- Gli ultimi quattro bits indicano quale partizione utilizzare; questa informazione è utilizzata per convertire gli indirizzi dei settori in forma assoluta cioè rispetto all'inizio di una periferica (gli indirizzi dei settori passati si riferiscono sempre alla partizione).

A questi tre valori non si accede direttamente ma tramite macro, quindi è possibile ridefinire la struttura interna dei "minor numeber" modificando solamente queste macro senza modificare altro codice.

Capitolo 4

Filesystem

Con il termine "filesystem" si indica l'organizzazione interna di un dispositivo a blocchi; questa struttura interna è mantenuta con lo scopo di memorizzarvi dei files, cioè per poter individuare le informazione contenute dentro ad un file.

Un dispositivo a blocchi è visto da un filesystem come un dispositivo avente dei blocchi di dimensione fissa ed ordinati in maniera che sia possibile leggere/scrivere l'n-esimo blocco; un filesystem mantiene informazioni per rispondere alla domanda fondamentale "su quali blocchi è memorizzato questo file?", queste informazioni sono mantenute sul dispositivo a blocchi stesso in maniera da poter essere sempre disponibili.

4.1 Interfaccia

In questa sezione si intende descrivere l'interfaccia fra il sistema ed i task utente e l'interfaccia fra il sistema ed uno specifico filesystem in maniera che sia possibile espandere il sistema con nuovi filesystem, magari studiati e progettati appositamente per sistemi real-time.

4.1.1 Inizializzazione

Per inizializzare la parte di alto livello del sistema (che si occupa di gestire i vari filesystem ed interfacciarsi con l'utente) si usa una metodologia simile a quella usata per i device driver e descritta nella sezione 3.1.1 a pagina 12.

Per inizializzare la parte di alto livello bisogna chiamare la funzione *filesystem_init()* passandole i parametri di inizializzazione che sono mostrati nella Tabella 4.1 nella pagina seguente, questi parametri andrebbero manipolati tramite le macro mostrate nella stessa tabella:

filesystem_def_rootdevice Setta la partizione da usare come "root device", cioè a quale directory effettiva associare la directory "/"; questo campo deve essere sempre specificato, questo implica che l'inizializzazione della parte di alto livello deve essere fatto dopo l'inizializzazione dei device drivers.

filesystem_def_fs Seleziona il filesystem da usare con il device definito precedentemente: deve essere uno dei simboli definiti nel file *include/fs/fsind.h*; il simbolo

```
typedef struct filesystem_parms {
  __dev_t           device;
  __uint8_t         fs_ind;
  __uint32_t        fs_showinfo:1;
  void              *fs_mutexattr;
  struct mount_opts *fs_opts;
} FILESYSTEM_PARMS;

#define filesystem_def_rootdevice(par,rootdev)
#define filesystem_def_fs(par,fs)
#define filesystem_def_showinfo(par,show)
#define filesystem_def_options(par,opts)

int filesystem_init(FILESYSTEM_PARMS *);
```

Tabella 4.1: La struttura filesystem_parms

```
struct mount_opts {
  __uint32_t flags;
  union {
    struct msdosfs_parms   msdos;
  } x;
};

/* Flags */
#define MOUNT_FLAG_RW 0x01
```

Tabella 4.2: La struttura mount_opts

speciale *FS_DEFAULT* può essere usato se si vuol far scegliere al sistema quale filesystem usare, comunque è bene specificarlo sempre.

filesystem_def_showinfo Un flag che se settato provoca la comparsa sul video dei messaggi di inizializzazione.

filesystem_def_options Definisce delle opzioni per il "montaggio" del "root device" che saranno descritte successivamente.

Il campo *fs_mutexattr* della struttura precedente ha uno scopo analogo al corrispondente campo della struttura *bdev_parms*, descritta nella sezione 3.1.1 a pagina 12, e non sarà qui ripetuta.

Questa struttura deve essere inizializzata, prima di poter essere usata con le macro precedentemente descritte, usando il valore *BASE_FILESYSTEM*; non è possibile chiamare la funzione *filesystem_init()* passadole un valore NULL poichè alcuni campi devono essere inizializzati.

Le opzioni per il "montaggio" di qualsiasi device sono contenute nella struttura *mount_opts* mostrata nella Tabella 4.2 (nel seguito l'operazione di "montaggio", che

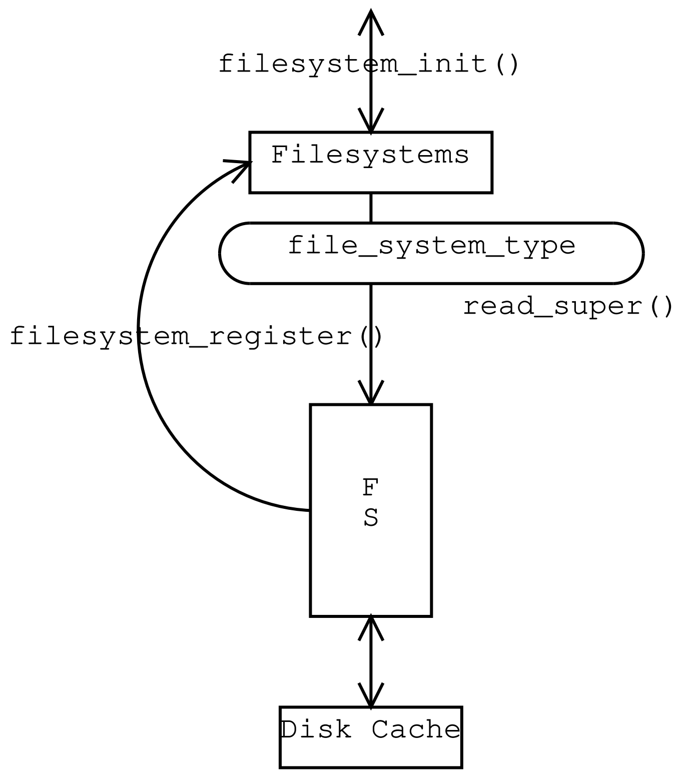

Figura 4.1: L'inizializzazione dei filesystems

consiste nell'associare una directory di partenza ad un device, sarà brevemente indicata con il termine "operazione di mount" o "mount"); questa struttura è composta da due parti: una parte generale ed una parte specifica di ogni filesystem; la parte generale attualmente contiene un solo campo:

flags Questo campo contiene i flags da usare per il montaggio; attualmente l'unico flag definito è *MOUNT_FLAG_RW* che, se settato, fa si che il filesystem sia montato in modalità di lettura/scrittura altrimenti, per default, i filesystem sono montati in modalità solo lettura per sicurezza.

Come mostrato nella Figura 4.1 l'inizializzazione di questo sottosistema avviene in varie fasi:

1. L'utente chiama la funzione *filesystem_init()* passandole i parametri descritti precedentemente.

2. La funzione *filesystem_init()*, dopo aver inizializzazto tutti i moduli interni, chiama le funzioni di inizializzazione dei vari filesystems; se si introduce un nuovo filesystem è necessario modificare questa funzione per far chiamare anche la funzionione di inizializzazione di questo nuovo filesystem.

3. Le funzioni di inizializzazione dei filesystems devono "registrarsi" chiamando la funzione *filesystem_register()*.

```
struct file_system_type {
  const char  *name;
  int         fs_flags;
  __uint8_t   fs_ind;

  struct super_block *(*read_super) (
         __dev_t device,
         __uint8_t fs_ind,
         struct mount_opts *options
         );
};
```

Tabella 4.3: Struttura file_system_type

La funzione *filesystem_register()* viene chiamata passandole un puntatore ad una struttura del tipo *file_system_type* descritta qui di seguito.

Struttura file_system_type

Questa struttura, mostrata in Tabella 4.3, mantiene le informazioni usate durante la "registrazione" di un filesystem: ogni filesystem presente nel sistema deve registarsi passando questa struttura come parametro durante la fase di inizializzazione; il meccanismo di inizializzazione è mostrato nella sezione 4.1.1 a pagina 42 dove si mostra come creare un nuovo modulo di filesystem da aggiungere al sistema.

I campi della struttura hanno il seguente significato:

name Contiene un puntatore al nome del filesystem, ad esempio "FAT16".

fs_flags Flags per i filesystems (attualmente non utilizzato).

fs_ind É un identificatore per il filesystem; ogni filesystem deve possedere un diverso identificatore.

read_super É la funzione di inizializzazione del filesystem: viene chiamata quando una nuova partizione viene montata per inizializzare la struttura *super_block*.

4.1.2 Interfaccia interna

Con interfaccia interna si vuole indicare l'interfaccia che esiste fra la parte generale di alto livello, che gestisce i filesystems, e i vari moduli, che implementano specifici filesystems.

Quando un utente decide di montare un device in una particolare directory utilizzando uno specifico filesystem, come descritto in 4.1.3 a pagina 49, il sistema trova la struttura *file_system_type* di quello specifico filesystem e chiama la funzione *read_super()* per leggere il superblocco di quello specifico device.

La struttura *super_block*, come molte altre strutture, contiene al suo interno un puntatore ad una struttura contenente operazioni "virtuali": tutte ls strutture dati utilizzate all'interno del sistema, descritte dettagliatamente nella sezione 4.2 a pagina 52, sono manipolate da queste funzioni "virtuali".

```
struct super_operations {
  int (*init_inode)   (struct inode *);
  int (*read_inode)   (struct inode *);
  int (*write_inode)  (struct inode *);
  int (*put_super)    (struct super_block *);
  int (*delete_inode) (struct inode *);
};
```

Tabella 4.4: Struttura super_operations

Implementare un nuovo filesystem significa definire le funzioni per operare “virtualmente” sulle strutture dati dati generali; dei puntatori queste funzioni vengono memorizzate in strutture dati per poter essere utilizzate, quando richiesto, dalla parte generale di alto livello. Queste strutture dati saranno analizzate nel seguito di questa sezione senza un particolare ordine, mentre le strutture dati usate internamente nel sistema saranno descritte nella sezione *sec:fs:intstruct*.

Struttura super_operations

La struttura, mostrata nella Tabella 4.4, serve per manipolare, in maniera “virtuale”, la struttura *super_block*. Tutte le seguenti funzioni devono ritornare uno zero in casi di successo, cioè se riescono a svolgere il proprio compito, un numero diverso da zero in caso di fallimento.

init_inode Questa funzione deve inizializare l’inode, cioè deve riempire tutti i campi della variabile passata come parametro, che è un puntatore alla struttura *inode*. Gli unici campi già inzializzati sono il campo *i_sb* ed il campo *i_st.st_ino* (vedi 4.2.1 a pagina 58).

read_inode Questa funzione deve leggere dal super blocco l’inode passato come parametro, cioè deve riempire tutti i campi della struttura *inode*, come fa la funzione *init_inode()*, con la differenza che questa funzione deve reperire le informazioni dal filesystem; anche a questa funzione viene passato un inode con i due campi precedentemente menzionati validi.

write_inode Questa funzione, alla quale viene passato un puntatore ad un inode valido, cioè con tutti i campi della struttura riempiti, deve scrivere l’inode nel super blocco, cioe’ nel filesystem corrispondente.

delete_inode Questa funzione deve rimuovere l’inode passato come parametro dal filesystem; naturalmente viene passato un inode attualmete esistente.

put_super Questa funzione deve trasferire il super blocco passato come parametro sul supporto di memorizzazione di massa; questa funzione viene chiamata, ad esempio, quando un filesystem viene smontato. La funzione duale di questa, cioè una funzione per leggere lo stato di un filesystem e trasferirlo all’interno di una struttura *super_block* non esiste poichè questa funzionalità, che serve solo durante la fase di montaggio di un filesystem, si trova nella struttura *file_system_type* (4.1.1 nella pagina precedente).

```
struct dentry_operations {
  int (*d_compare) (struct dentry *,
                    struct qstr *,
                    struct qstr *);
};
```

Tabella 4.5: Struttura dentry_operations

```
struct inode_operations {
  struct file_operations *default_file_ops;

  struct inode* (*create)   (struct inode *,
                             struct dentry *);
  struct inode* (*lookup)   (struct inode *,
                             struct dentry *);
  int           (*unlink)   (struct dentry *);
  void          (*destroy)  (struct inode *);
  int           (*truncate) (struct inode *,
                             __off_t len);
};
```

Tabella 4.6: Struttura inode_operations

Struttura dentry_operations

Questa struttura, mostrata nella Tabelle 4.5, contiene le funzioni per manipolare la struttura *dentry*; attualmente è presente una sola funzione:

d_compare Questa funzione deve comparare le due stringhe passate come argomento e restituire zero se sono uguali, un numero diverso da zero se sono diverse; oltre a questi due parametri viene passata anche un puntatore ad una struttura *dentry*.

Perchè non usare direttamente la funzione *strcmp()()* che compara due stringhe? Poichè in alcuni filesystems nei nomi dei files non si fà distinzione fra lettere maiuscole e minuscole ed alcuni impongono che devono essere composti sempre da lettere maiuscole.

Struttura inode_operations

Questa struttura contiene le operazioni "virtuali" operanti sulla struttura *inode*; tutte le funzioni ritornano zero in caso di successo o un numero diverso da zero in caso di errore, che verra interpretato come un codice per la variabile utente *errno*; è necessario usare numeri negativi in caso di errore (cioè l'errore *EPERM* deve essere ritornato come -*EPERM*).

default_file_ops Sono le *file_operations*, descritte nel paragrafo 4.1.2 nella pagina seguente, da usarsi per operare con questo *inode*; per default questo campo viene inizializzato con un puntatore alle *file_operations* del filesystem nel quale questo file risiede; può essere utile modificarle per implementare dei "filesystem virtuali" operanti su device; ad esempio si può fare in modo che la lettura/scrittura sul file "/dev/ide/hda1", che dovrebbe rappresentare un device, operi mediante funzioni ad hoc e non tramite le funzioni del filesystem nel quale questo file risiede.

create Questa funzione deve creare un nuovo *inode*, cioè un nuovo file, all'interno dell'*inode* passato come parametro, che deve essere un inode di directory, avente il nome passato come parametro; questa funzione deve ritornare il nuovo inode creato o *NULL* in caso di errore.

lookup Questa funzione deve cercare l'*inode* che ha il nome passato come parametro all'interno del'*inode* passato che deve essere una directory; deve ritornare l'*inode* trovato o *NULL* se non può essere trovato o in caso di errore.

unlink Questa funzione deve disassociare il nome del file passato come parametro dall'inode corrispondente; la cancellazione del file, nel caso che l'inode associato rimanga senza link, viene fatta dalle routine generiche di sistema.

destroy Questa funzione deve distruggere il file associato all'*inode* passato, cioè deve liberare lo spazio occupato da questo file; spesso è sufficiente chiamare la funzione *truncate()*, descritta successivamente, passando zero come lunghezza del file.

truncate Questa funzione deve troncare la lunghezza del file passato (identificato tramite la struttura *inode*) alla lunghezza specificata nel secondo parametro; se la lunghezza passata è superiore alla lunghezza attuale del file questa funzione non deve fare niente.

Struttura file_operations

Questa struttura contiene le principali operazioni "virtuali" che operano sui file; se non diversamente specificato, queste operazioni devono poter operare sia sui files regolari che sulle directory, in particolare la funzione *open()* deve poter aprire sia files che directories.

La struttura, come è mostrato nella Tabella 4.7 nella pagina successiva, contiene i seguenti campi:

lseek Deve spostare la posizione di input/output nel file passato come parametro a quella specificata; la posizione è specificata come nella primitiva di sistema *lseek()*, descritta in [11] o in altri testi sulla progrrammazione in *Unix*, quindi tramite un valore che specifica il delta, cioè l'incremento o il decremento, dalla posizione indicata (inizio file, fine file o posizione corrente); come già descritto nella sezione 4.2.1 a pagina 60 durante la descrizione del campo *f_pos*, la nuova posizione può essere oltre la fine del file; questa funzione deve ritornare la nuova posizione all'interno del file o un codice di errore.

read Deve leggere e posizionare nel buffer specificato un certo numero di bytes; questa funzione viene chiamata solo per i files regolari; deve ritornare il numero di byte letti, che possono essere meno di quelli richiesti, o un codice di errore.

```
struct file_operations {
  __off_t   (*lseek)    (struct file *,
                         __off_t, int);
  __ssize_t (*read)     (struct file *,
                         char *,
                         __ssize_t);
  __ssize_t (*write)    (struct file *,
                         char *,
                         __ssize_t);
  int       (*readdir)  (struct file *,
                         void *);
  int       (*open)     (struct inode *,
                         struct file *);
  int       (*close)    (struct inode *,
                         struct file *);
};
```

Tabella 4.7: Struttura file_operations

write Duale della precedenta, deve scrivere sul file un certo numero di byte prelevandoli dal buffer specificato; viene chiamata solo per file regolari; deve ritornare il numero di byte scritti, che possono essere meno di quelli richiesti, o un codice di errore.

readdir Deve leggere il nome del prossimo file presente nella directory passata come parametro tramite un puntatore alla struttura *file*, e inserirlo nella struttura *dirent* passata come parametro; deve ritornare zero se non ci sono errori, un codice negativo in caso di errore.

open Deve aprire il file associato all'inode passato come parametro riempiendo i campi specifici del filesystem interni alla struttura *file*; deve ritornare un codice di errore se l'apertura fallisce, zero in caso di successo.

close Deve chiudere il file passato come parametro; deve ritornare zero in caso di successo, un codice di errore in caso di fallimento.

Molte delle funzioni descritte precedentemente operano su buffers passati dall'utente; le routine generali controllano che questi buffer siano "corretti" e li passano direttamente alle routine specifiche di un filesystem perciò è opportuno utilizzare le funzioni *copyfromuser()* e *copytouser()*, aventi una sintassi uguale alla funzione *memcpy()*, per operare su questi buffer altrimenti il sistema potrebbe non funzionare se è presente un meccanismo di protezione della memoria che separa lo spazio di memoria del kernel da quello delle applicazioni utente. L'attuale implementazione per *S.Ha.R.K.* non ha questo problema poichè la memoria non è protetta e quindi si potrebbe utilizzare direttamente la funzione *memcpy()*.

4.1.3 Interfaccia esterna

Funzioni non standard

```
int mount      (dev_t device,
                u_int8_t fsind,
                char *where,
                struct mount_opts *options);

int umount     (dev_t device);

dev_t fdevice (char *device_name);
```

Tabella 4.8: Le funzioni mount/umount

Ci sono alcune funzioni che vengono esportate verso l'utente che non sono standard: le funzioni per fare il mount di dispositivi, il prototipo delle quali si trova nel file *include/sys/mount.h*. Queste funzioni hanno una interfaccia molto semplice visibile nella Tabella 4.8:

fdevice Questa funzione serve per sapere il device serial number di un device del quale si conosce il nome simbolico; vedi la sezione 3.1.2 a pagina 14.

mount Viene usata per montare una partizione; questa funzione ha bisogno di conoscere:

- Il device serial number del *device* che si vuole montare.
- Il filesystem (*fsind*) che si trova su questo particolare device; gli indicatori dei filesystem si trovano nel file *include/fs/fsind.h*.
- La directory (*where*) dove questo dispositivo deve essere montato.
- Eventuali parametri opzionali; già descritti nella sezione 4.1.1 a pagina 43.

Questa funzione ritorna zero se ha successo, un numero diverso da zero settando la variabile *errno* se fallisce.

umount Questa funzione serve per smontare un device precedentemente montato; questa funzione non può smontare il "root device"; non è necessario smontare i device montati: durante la fase di shutdown tutti i device vengono smontati nell'ordine giusto.

4.1.4 Inizializzazione: un esempio

Un esempio di inizializzazione globale di tutto il sistema all'interno di *S.Ha.R.K.*; per inizializzare il sistema bisogna innanzitutto definire la funzione di registrazione dei moduli del kernel:

```
TIME __kernel_register_levels__(void *arg)
{
  struct multiboot_info *mb=(struct multiboot_info *)arg;

  /* registrazione moduli di schedulazione */
  EDF_register_level(EDF_ENABLE_ALL);
```

```
    RR_register_level(RRTICK, RR_MAIN_YES, mb);
    CBS_register_level(CBS_ENABLE_ALL, 0);
    dummy_register_level();

    /* registrazione semafori, porte e mutex */
    SEM_register_module();
    CABS_register_module();
    NOPM_register_module();

    return TICK;
}
```

In questa funzione di registrazione dei moduli del kernel non abbiamo usato nessuna funzione per inizializzare i filesystem; abbiamo registrato moduli per la schedulazione e per alcune risorse; il porting di questo sistema verso *S.Ha.R.K.* richiede che sia presente almeno un livello che accetti i task soft aperiodici (modulo "CBS" in questo caso) e che sia disponibile un protocollo per i mutex (modulo "NOPM").

```
TASK __init__(void *arg)
{
    extern int __bdev_sub_init(void);
    extern int __fs_sub_init(void);
    struct multiboot_info *mb=(struct multiboot_info *)arg;

    /* inizializzazione ``block devices'' */
    __bdev_sub_init();
    /* inizializzazione ``filesystems'' */
    __fs_sub_init();

    /* chiamata della funzione main() */
    __call_main__(mb);

    return (void *)0;
}
```

Questa funzione serve per inizializzare il sistema; viene chiamata dal sistema operativo quando siamo entrati in multiprogrammazione e siamo pronti a fare partire il task "main"; durante questo codice chiamiamo due funzioni che servono ad inizializzare la parte di basso livello e quella di alto livello.

```
int __bdev_subinit(void)
{
    extern __dev_t root_device;
    BDEV_PARMS bdev=BASE_BDEV;

    /* inizializzazione basso livello */
    bdev_def_showinfo(bdev,TRUE);
    bdev_init(&bdev);

    /* ricerca del ``root device'' */
    root_device=bdev_scan_devices(choose_root_callback);
    if (root_device<0) {
        /* in caso di errore... */
```

```
    sys_end();
    return -1;
  }
  return 0;
}
```

Questa funzione inizializza la parte di basso livello: imposta i parametri della variabile *bdev*, chiama la funzione di inizializzazione del basso livello e cerca il “root device”; il device da montare come root deve essere specificato in qualche modo, in questo caso facciamo si che il basso livello ci elenchi tutti i device presenti e che chiami la funzione *choose_root_callback()* per ogni device presente; sarà questa funzione a scegliere il “root device”.

```
int choose_root_callback(dev_t dev, u_int8_t fs)
{
  if (fs==FS_MSDOS) return dev;
  return -1;
}
```

La scelta del root device è molto semplice: la prima partizione che contiene un filesystem MSDOS diventa il nostro root device; l’MS-DOS, di solito, chiama “C:” questo device.

```
int __fs_sub_init(void)
{
  extern __dev_t root_device;
  FILESYSTEM_PARMS fs=BASE_FILESYSTEM;
  struct mount_opts opts;

  /* paramtrei per il mount */
  memset(&opts,0,sizeof(struct mount_opts));
  opts.flags=MOUNT_FLAG_RW;

  /* inizializzazione dei ``filesystems'' */
  filesystem_def_rootdevice(fs,root_device);
  filesystem_def_fs(fs,FS_MSDOS);
  filesystem_def_showinfo(fs,TRUE);
  filesystem_init(&fs);

  /* inizializzazione della libreria ``C'' */
  libc_initialize();

  return 0;
}
```

Questo codice inizializza la parte di alto livello: viene montata in modalità lettura/scrittua la partizione precedentemente scelta e quindi viene inizializzata la libreria “C”.

4.2 Struttura interna

Molte delle strutture dati sono composte da due parti: una parte generale contenente informazioni necessarie alle primitive di sistema per svolgere le proprie funzionalità ed avente significato per ogni filesystem e da una parte particolare contenete informazioni utili o necessarie per uno specifico filesystem; queste informazioni particolari sono mantenute in una “union”.

Quasi tutte le strutture dati hanno associata un’altra struttura contenente puntatori a funzioni utilizzate per manipolare le strutture associate in maniera specifica (il loro nome è ottenuto dal nome della struttura associata con l’aggiunta del suffisso “operations”); questo permetette di “virtualizzare” alcune operazioni sulla struttura in maniera che ogni filesystem possa implementarle in maniera diversa; ad esempio associata alla struttura *super_block* (4.2.1) vi è la struttura *super_operations* (4.1.2 a pagina 46) contenente, ad esempio, la funzione *create_inode()* che serve ad un filesystem per creare un inode (ovviamente).

Le strutture dati presenti nella parte di alto livello sono molto importanti per l’efficienza complessiva del sistema, per questo motivo alcune informazioni sono duplicate in più strutture cioè alcuni campi interni alle strutture sono ridondanti.

Questo sistema è strutturato internamente ad lato livello utilizzando un approccio Unix-like: esistono 4 strutture dati fondamentali: la struttura *super*, contenete informazioni su ogni device montato e memorizzata in una tabella, la struttura *inode*, contenete informazioni sui files presenti in un filesystem e memorizzata in liste doppie a chiave hash, la struttura *dentry*, contenete i nomi dei file ed organizzata ad albero e la struttura *file*, contenente informazioni su ogni file aperto nel sistema ed organizzata in tabelle (tabelle dei descrittori).

4.2.1 Le strutture dati

Struttura super_block

Dal punto di vista dell’utente, cioè dell’utilizzatore delle primitive di sistema per l’input/output, il sistema è simile ad un sistema Unix: non esistono “drives” (come nell’MSDOS), ma il sistema permette di “montare” delle partizioni all’interno dell’albero delle directory. Alla partenza il sistema monta la partizione che è stata scelta come radice nella “root directory” (che convenzionalmente viene indicata con “/”), cioè associa alla directory radice la partizione scelta; tutte le altre partizioni sono visibili montandole in una sottodirectory di una directory già montata.

Questa struttura mantiene le informazioni fondamentali di un filesystem (in ambito Unix queste informazioni vengono chiamate “super block” e vengono duplicate più volte all’interno di un hard-disk); ogni volta che una partizione viene montata una nuova struttura viene allocata ed inizializzata. Queste strutture vengono mantenute ad albero, cioè se una partizione viene montata all’interno di una directory, la nuova struttura diventa una “foglia” della struttura che contiene la directory di montaggio.

La radice, cioè il super block della partizione di root, si trova nella variabile puntatore *root_superblock*; un possibile albero di super bloccchi è rappresentato in Figura 4.2 a pagina 55.

I campi della struttura *super_block*, visibili nella Tabella 4.9 nella pagina successiva, sono utilizzati per i seguenti scopi:

sb_parent, sb_childs, sb_next Servono per mantenere la struttura organizzata ad albero: *sb_parent* punta al genitore di questa struttura, *sb_childs* punta alla prima struttura dei figli delle struttura attuale e *sb_next* serve per costruire la lista dei fratelli cioè la lista dei figli della struttura genitore.

```
struct super_block {
  struct super_block        *sb_parent;
  struct super_block        *sb_childs;
  struct super_block        *sb_next;

  __dev_t                   sb_dev;
  struct inode              *sb_root;
  struct dentry             *sb_droot;
  struct file_system_type   *sb_fs;
  struct super_operations   *sb_op;
  struct dentry_operations  *sb_dop;
  struct mount_opts         sb_mopts;
  int                       sb_busycount;

  __uint32_t                sb_used:1;
  __uint32_t                sb_blocked:1;

  union {
    struct msdos_super_block msdos_sb;
  } u;
};
```

Tabella 4.9: Struttura super_block

sb_dev Contiene il "device", cioè la partizione contenete questo filesystem (la parte di basso livello è responsabile dei device e di solito li associa ad una partizione su un hard-disk; si potrebbe estendere la parte a basso livello per far associare ai device altre "entità", as esempio un file su un altro filesystem; questa operazione di solito è prevista nei sistemi Unix tramite device di loopback).

sb_root É l'"inode" principale (vedi 4.2.1 a pagina 58), cioè della directory root, di questo filesystem.

sb_droot É la "directory entry" della directory root; con directory entry si intende un nome di file (vedi 4.2.1 nella pagina successiva).

sb_fs É un puntatore al filesystem al quale questa partizione appartiene; ogni filesystem presente nel sistema si registra (vedi la sezione 4.1.1 a pagina 42) durante la fase di inizializzazione.

sb_op Puntatore ad una struttura contenete le funzioni specifiche di un filesystem per agire su questa struttura super_block.

sb_dop Operazione specifiche di un filesystem per agire sulla "directory entry".

sb_mopts Opzioni usate durante la fase di montaggio della partizione.

sb_busycount Incrementato e decrementato per segnalare alle altre parti del sistema che questa struttura è in uso (ad esempio per evitare si smontare la partizione quando si stà creando un nuovo file).

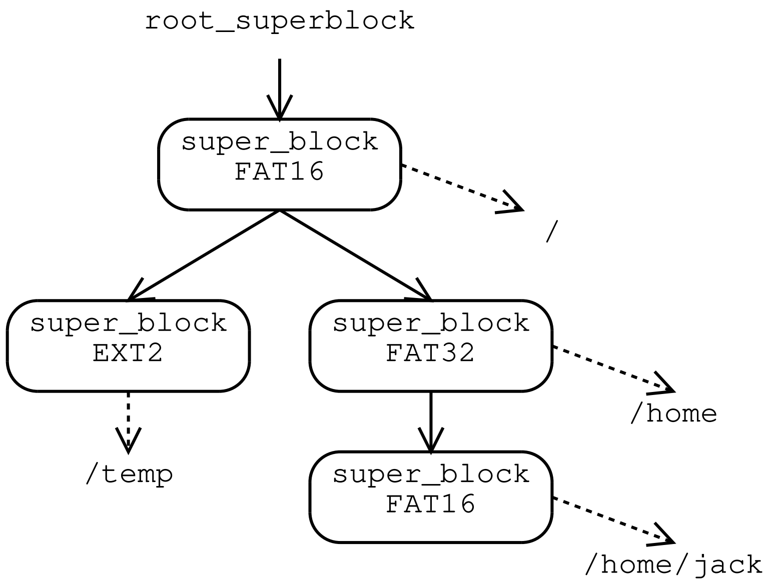

Figura 4.2: Albero di super blocchi

sb_used É un flag che indica se questa struttura è usata.

sb_blocked Alcune operazioni devono essere eseguite in mutua esclusione su questa struttura; se questo flag è settato significa che qualche operazione è in esecuzione (questo flag non può essere testato o modificato direttamente dalle funzioni utente).

Struttura dentry

La struttura *dentry* contiene i nomi dei files, indipendentemente dal tipo di file, cioè sia che il file sia regolare, sia che sia una directory, ecc... ("dentry" è la contrazione di "directory entry"). Ogni file presente in un filesystem viene sempre memorizzato in due strutture:

1. nella struttura *dentry* è presente il nome del file.

2. nella struttura *inode* sono presenti tutte le altre informazioni che servono a caratterizzare un file.

Questa suddivisione è dovuta al fatto che in molti filesystem (non nel filesystem FAT16 dell'MS-DOS) sono possibili i "links", cioè ad un file si possono associare più nomi e quindi la relazione fra nome del file e file fisico non è uno a uno ma è molti a uno. Quindi ad un file fisico, che in ambito unix viene chiamato "inode" possono essere associati più nomi.

Queste strutture dati sono mantenute internamente al sistema mediante una organizzazione ad albero, come è naturale visto che molto spesso i files sono memorizzati nei filesystem ad albero; da notare che gli inodes, pur essendo i "file fisici" non sono

```
struct dentry {
  struct dentry *d_next;
  struct dentry *d_prev;
  struct dentry *d_parent;
  struct dentry *d_child;

  __time_t       d_acc;
  struct qstr    d_name;
  short          d_lock;

  struct dentry_operations  *d_op;
  struct inode              *d_inode;
  struct super_block        *d_sb;
};
```

Tabella 4.10: Struttura dentry

```
struct qstr {
  char name[MAXFILENAMELEN+1];
  char *nameptr;
};
```

Tabella 4.11: Struttura qstr

strutturati ad albero e che perciò ogni struttura dentry mantiene un puntatore all'inode corrispondente (naturalmente un inode potrebbe non avere un solo puntatore alle dentry).

Nella Figura 4.3 nella pagina successiva è mostrato un esempio di albero di "directory entries". Per chiarezza le strutture che rappresentano directory sono state disegnate a tratto continuo, mentre quelle che rappresentano files regolari sono state disegnate a tratteggio; internamente, però, non viene fatta alcuna distinzione fra i due tipi di file.

Internamente al sistema sono mantenute solamente le *dentry* in uso o usate di recente, non tutte le informazioni presenti in un filesystem, perciò internamente al sistema si ha sempre un albero parziale che viene continuamente aggiornato aggiungendo o rimuovendo *dentry*.

I nomi associati alle voci di directory sono mantenuti nella struttura *qstr*. Ci si può chiedere: perchè non utilizzare direttamente una stringa (cioè un array di caratteri)? ciò non viene fatto perchè, altrimenti, molte routines che manipolano i nomi dei files dovrebbero copiarli molte volte; questa struttura è composta, come mostrato nella tabella 4.11 da due campi:

nameptr Se diverso da NULL, è il nome da usare.

name É il nome da usare se il campo precedente è NULL.

La macro *QSTRNAME* può essere usata sulla struttura *qstr* per avere un puntatore a caratteri; quando si vuole riempire questa struttura si può scegliere di usare uno di

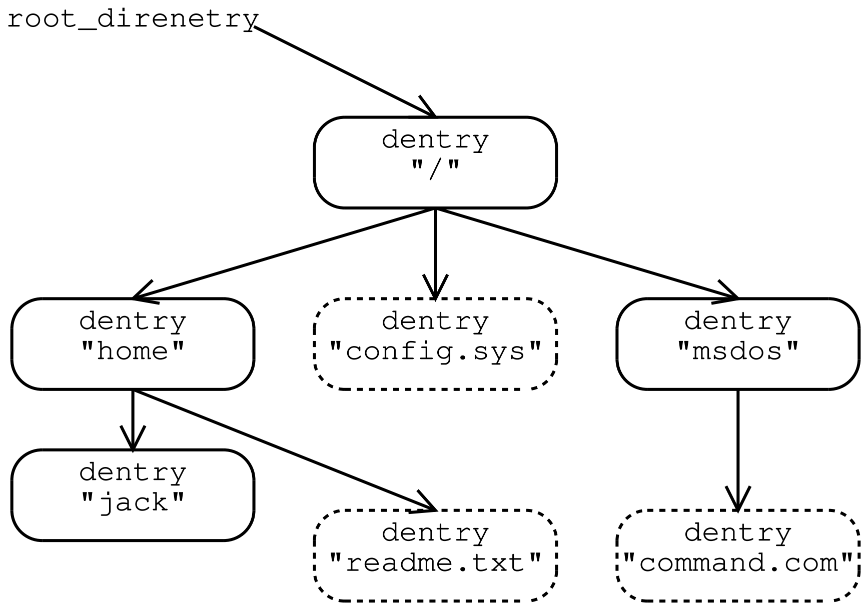

Figura 4.3: Albero di directory entries

questi due campi: se si sceglie *nameptr* bisogna ricordarsi che la stringa non può essere dinamica ma deve essere statica, se si sceglie di utilizzare *name* bisogna ricordarsi di scrivere NULL nel campo *nameptr* e bisogna copiare la stringa all'interno della struttura.

I campi di questa struttura mantengono le seguenti informazioni:

d_next, d_prev, d_parent e d_child Servono per mantenere la struttura ad albero delle *dentry*: *d_parent* è un puntatore alla *dentry* padre, *d_child* è un puntatore alla lista delle *dentry* figlie, *d_next* e *d_prev* servono per creare la lista doppia (cioè una lista che si può scorrere sia in avanti che all'indietro) delle figlie di una *dentry*.

d_acc Mantiene l'istante dell'ultimo accesso a questa *dentry*.

d_name É il nome associato a questa *dentry*.

d_lock É un contatore di bloccaggio: ogni volta che qualche routine ha bisogno di questa *dentry* incrementa questo contatore, quando ha finito di usarlo lo decrementa; in questo modo è possibile sapere quali sono le strutture attualmente in uso. Le routine di un particolare filesystem non possono (non ne hanno bisogno) modificare questo campo direttamente.

d_op Puntatore alla tabella delle operazioni virtuali per agire su questa struttura.

d_inode É l'inode associato a questa struttura.

d_sb É il *super_block* nel quale si trova questa voce di directory; è una informazione ridondante ma utile per velocizzare il codice.

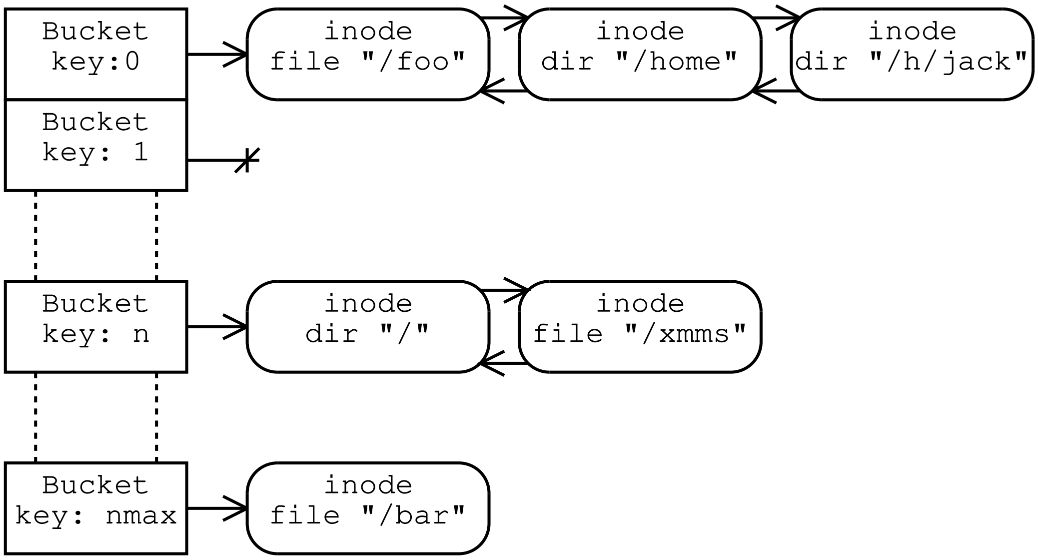

Figura 4.4: Struttura dati per gli inodes

Struttura inode

Un *inode* è un "file" presente all'interno di un filesystem: contiene tutte le informazioni che il sistema dispone su un file eccetto il nome che viene mantenuto nella *dentry* (vedi 4.2.1 a pagina 55); una descrizione più approfodità degli inode può essere trovata in [15].

Internamente al sistema tutti gli inode presenti in un certo istante sono memorizzati all'interno di una struttura dati a "hash buckets" come mostrato in Figura 4.4: per ogni inode viene calcolato un numero tramite una funzione hash avente come parametri il numero univoco dell'inode all'interno di un filesystem (file serial number) ed il numero univoco del device sul quale l'inode risiede. Tutti gli inodes con lo stesso valore della funzione hash, che viene utilizzata come una chiave, vengono memorizzati in una lista; l'inizio della lista, corrispondente ad un certo valore della funzione hash, viene memorizzato all'interno di una tabella nella posizione corrispondente al valore hash.

Un *inode* contiene i seguenti campi:

i_st É una struttua contenente tutte le informazioni "standard" per un file (come la lunghezza ed il tipo); queste informazioni possono essere conosciute da un processo utilizzando la funzione standard *stat()*, per questo motivo queste informazioni si trovano in una struttura a parte: la struttura *stat* (descritta più avanti in questa sezione) è condivisa dalle librerie utente e dal sistema di gestione del filesystem, cosicchè sia possibile utilizzare una semplice copia per trasferire queste informazione ai processi.

i_next, i_prev e i_hash Questi campi servono per implementare le code doppie a chiave hash utilizzate per mantenere queste strutture: *i_next* e *i_prev* implementato la coda doppia mentre *i_hash* è il valore della chiave hash (informazione ridondante ma utile).

```
struct inode {
  struct stat   i_st;
  struct inode *i_next;
  struct inode *i_prev;
  int           i_hash;

  __rwlock_t    i_lock;
  __uint16_t    i_dlink;

  __uint32_t    i_dirty:1;

  struct inode_operations *i_op;
  struct super_block      *i_sb;

  union {
    struct msdos_inode_info msdos_i;
  } u;

};
```

Tabella 4.12: Struttura inode

i_lock É un "locker" per poter leggere/modificare questa struttura (4.2.2 a pagina 66).

i_dlink Numero di directory entries che puntanto a questo *inode* (vedi 4.2.1 a pagina 55).

i_dirty Flag che indica se questo *inode* è "sporco" cioè se è stato modificato.

i_op Funzioni "virtuali" per operare su questo *inode*; ogni filesystem utilizza le proprie funzioni virtuali per operare sugli inodes.

i_sb Super blocco (vedi 4.2.1 a pagina 53) al quale questo inode appartiene.

La struttura *stat*, mostrata nella Tabella 4.13 nella pagina seguente, contenuta all'interno della struttura *inode*, della quale una descrizione approfondita può essere trovata nel paragrafo 5.6 di [11], contiene i seguenti campi:

st_mode É il "modo" del file associato, cioè contiene informazioni circa i diritti sul file (chi può leggerlo, modificarlo o eseguirlo).

st_ino É un numero univoco all'interno di un filesystem che consente di individuare il file (chiamato "file serial number" nella terminologia *Posix*).

st_dev É il numero del dispositivo che contiene il file (device serial number); *st_dev* e *st_ino* sono necessari e sufficienti per individurare in maniera univoca un qualsiasi file all'interno del sistema.

st_nlink Numero di collegamenti al files ("hard link"); questo campo vale sempre 1 per il filesystem FAT16.

```
struct stat {
  __dev_t       st_dev;
  __ino_t       st_ino;
  __mode_t      st_mode;
  __nlink_t     st_nlink;
  __uid_t       st_uid;
  __gid_t       st_gid;
  __off_t       st_size;
  unsigned long st_blksize;
  __time_t      st_atime;
  __time_t      st_mtime;
  __time_t      st_ctime;
};
```

Tabella 4.13: Struttura stat

st_uid e st_gid Identificatori dell'utente e del gruppo ai quali appartiene il file; per convenzione 0 rappresenta l'identificatore dell'amministratore di sistema; questi campi sono sempre impostati a 0 dal filesystem FAT16.

st_size Dimensione del file in bytes.

st_atime, st_mtime e st_ctime Tempi di un files: *st_atime* è il tempo dell'ultimo accesso, *st_mtime* dell'ultima modifica e *st_ctime* dell'ultimo cambio di stato del file (ad esempio quando il file è creato o è stato modificato tramite *chmod()*).

Struttura file

Ogni file aperto ha associato una struttura *file*, mostrata nella tabella 4.14 nella pagina successiva, contenente le informazioni necessarie per manipolare il file; sono presenti i seguenti campi:

f_next Contiene un puntantatore che viene utilizzato internamente al modulo e che perciò non deve essere utilizzato dalle routine proprie di un filesystem.

f_dentry Contiene il nome del file (vedi la sezione 4.2.1 a pagina 55) dal quale si puo risalire all'inode corrispondente.

f_op Contiene un puntatore alle funzioni "virtuali" operanti sulle strutture *file* e descrtitte nel paragrafo 4.1.2 a pagina 48.

f_pos Contiene la posizione corrente all'interno del file: le operazioni di lettura/scrittura devono essere fatte a partire da questa posizione; le specifiche *Posix*permettono di fare un "seek", cioè di spostare la posizione di lettura/scrittura, oltre la fine del file specificando che in questo caso una lettura debba ritornare un errore ma una scrittura debba scrivere a questa posizione riempiendo lo spazio fra la fine del file e questa posizione con degli zeri; perciò le funzioni specifiche di un filesystem che implementano la lettura, scrittura o il posizionamento all'interno di un file devono considerare queste specifiche e quindi devono prendere in considerazione il fatto che il campo *f_pos* possa essere posizionato oltre la fine del file.

```
struct file {
  struct file            *f_next;

  struct dentry          *f_dentry;
  struct file_operations *f_op;

  __loff_t               f_pos;
  unsigned int           f_count;
  __uint32_t             f_flag_isdir:1;

  union {
    struct msdos_file_info msdos_f;
  } u;
};
```

Tabella 4.14: Struttura file

f_count Contiene il numero di descrittori di files che utilizzano questa struttura per le operazioni di input/output sul file in oggetto.

f_flag_isdir Informazione ridondante per conoscere se questo file è una directory, in maniera da poter fare dei check più veloci.

msdos_f Informazioni specifiche all'implementazione del filesystem FAT16 ed utilizzate internamente da questo.

Le applicazioni utilizzano un "descrittore di file" per indicare su quale file operare; un descrittore di file non è altro che un numero, avente significato solo per il processo specifico che l'ha ottenuto (ad esempio tramite una chiamata alla funzione *open()*), che viene associato ad una struttura *file*; questa associazione è fatta all'interno del sistema mediante un tabella. Naturalmente ad ogni processo viene associata una diversa tabella contenuta nella struttura *file_descriptors* descritta nella sezione 4.2.1. In *S.Ha.R.K.* esiste una sola tabella poichè tutti i task fanno parte dello stesso processo (applicazione).

Struttura file_descriptors

Questa struttura, mostrata nella Tabella 4.15 nella pagina successiva, viene usata per mantenere i parametri generali usati dal sistema e associati ad ogni processo come ad esempio la directory di lavoro corrente; in *S.Ha.R.K.*è presente una sola struttura poichè il sistema è monoprocesso multithread; quando questa struttura è usata in un sistema multithread, come *S.Ha.R.K.*, bisogna fare attenzione alla concorrenza dei threads. In particolare se bisogna modificare o accedere a questa struttura bisogna assicurarsi di essere gli unici a fare operazioni sulla struttura tramite il semaforo di mututa esclusione, che non viene utilizzato in caso di sistemi monothread.

Questa struttura contiene i seguenti campi:

fd_file É la tabella dei descrittori; ad ogni possibile descrittore (un numero compreso fra 0 e MAXOPENFILES) è associato un elemento della tabella che può contenere i seguenti valori:

```
struct file_descriptors {
  struct file   *fd_file[MAXOPENFILES];

  __uint16_t    fd_flags[MAXOPENFILES];
  __uint32_t    fd_fflags[MAXOPENFILES];

  __mode_t      fd_umask;
  char          fd_cwd[MAXPATHNAMELEN+1];
  struct dentry *fd_cwden;

  __mutex_t     fd_mutex;
};
```

Tabella 4.15: Struttura file_descriptors

- NULL se il descrittore è libero,
- RESERVED se è riservato per usi speciali (come i descrittori associati alla tastiera o al terminale)
- altrimenti è un puntatore al file associato.

fd_flags Sono i flags associati al file, come ad esempio FD_CLOSEXEC, e descritti dettagliatamente in [11] nel paragrafo dove viene descritta la funzione *fcntl()*.

fd_fflags Sono i flag che sono stati utilizzati durante l'apertura del file associato, come ad esempio O_APPEND, O_RDONLY, ecc...

fd_umask É la maschera di creazione dei file usata durante la creazione di un nuovo file; può essere modificata tramite la funzione *umask()*.

fd_cwd É il nome della directory corrente di lavoro utilizzata da tutte le primitive alle quali viene passato un pathname non assoluto, cioè senza lo slash iniziale. Questa informazione è ridondante in quanto potrebbe essere ricavata dal campo successivo.

fd_cwden É la directory di lavoro corrente modificabile tramite la funzione *chdir()*.

fd_mutex É il semaforo di mutua esclusione utilizzato nei sistemi multitread per coordinare l'accesso a questa struttura.

4.2.2 La cache del disco

Come accedere alla cache

I moduli che gestiscono i vari filesystems devono poter accedere alla cache per reperire le informazioni delle quali necessitano.

L'interfaccia verso il modulo che gestisce la cache è mostrata nella Tabella 4.16 nella pagina seguente:

dcache_lock Questa funzione serve per richiedere l'accesso in "sola lettura" alla cache, più task possono avere l'accesso in lettura contemporaneamente; le viene

```
dcache_t *dcache_lock    (__dev_t dev,
                          __blkcnt_t lsector);
void      dcache_unlock  (dcache_t *);

dcache_t *dcache_acquire (__dev_t dev,
                          __blkcnt_t lsector);
void      dcache_release (dcache_t *);

void dcache_dirty (dcache_t *d);
void dcache_skipwt(dcache_t *d);
```

Tabella 4.16: L'interfaccia verso la cache

passato il device sul quale il settore risiede ed il numero di tale settore; restituisce un puntatore alla struttura *dcache_t* in caso di successo o il valore NULL in caso di errore; questa funzione non ritorna fino a quando o l'accesso richiesto è consentito o si è verificato un errore, quindi, per evitare problemi di blocchi critici è necessario, se si deve richiedere l'accesso a più blocchi contemporaneamante, utilizzare un protocollo di accesso: si devono effettuare i bloccaggi in ordine crescente di numero di blocco; comunque è conveniente, se possibile, bloccare un solo settore alla volta.

dcache_unlock Questa funzione server per sbloccare un'entry di cache precedentemente bloccata con la funzione sopradescritta.

dcache_acquire Questa funzione si comporta come la funzione *dcache_lock()*, solamente richiede un accesso in "lettura/scrittura"; ad un solo task alla volta è consentito l'accesso in queste modalità. Questo tipo di accesso non comporta automaticamente che la cache entry venga ritenuta "sporca", cioè modificata, vedi la funzione *dcache_dirty()*.

dcache_release La funzione analoga alla *dcache_unlock()* per la funzione *dcache_acquire()*.

dcache_dirty Segnala al sistema che la cache entry indicata è stata modificata; questa funzione va chiamata solo se si possiede il permesso di "lettura/scrittuta" se si è modificata la cache.

dcache_skipwt Questa funzione indica al sistema che la cache entry indicata contiene informazioni di sistema, che viene modificata spesso e che perciò il flag *write_trought* non andrebbe onorato; vedi la sezione 4.2.2 a pagina 66 per una descrizione più dettagliata.

Struttura dcache

Questa struttura dati, usata dal modulo che implementa la cache, rappresenta una entry di cache: tutte le funzioni della parte di alto livello usano il modulo *cache* per interfacciarsi con la parte di basso livello; questa struttura dati è usata per passare i dati richiesti alla parte di alto livello.

Queste strutture dati sono mantenute all'interno del sistema con una organizzazione simile a quella adottata per gli inodes: una serie di liste alle quali è possibile

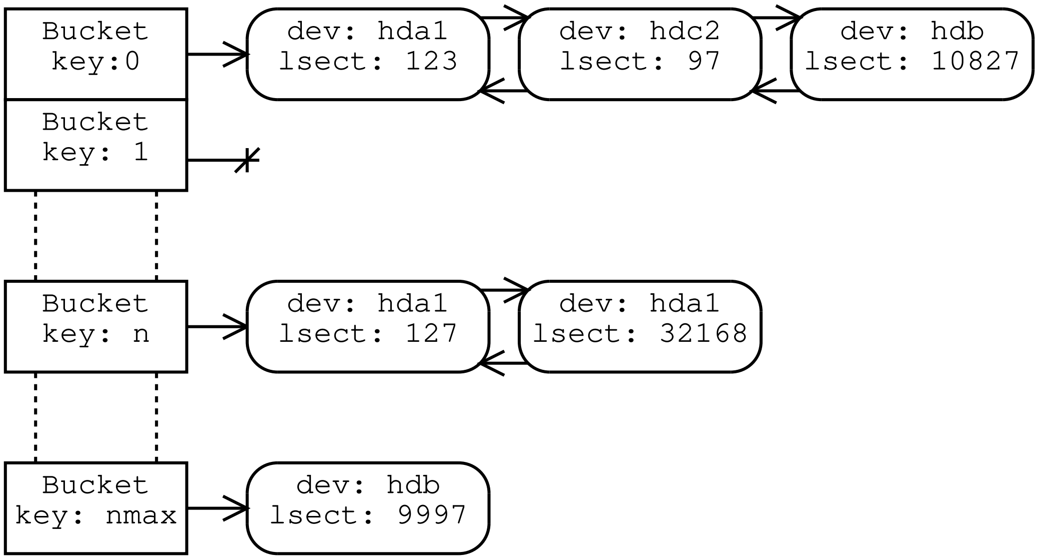

Figura 4.5: Struttura dati per la disk cache

accedere tramite una chiave hash, come mostrato nella Figura 4.5; la chiave hash, che è mantenuta anche dentro la struttura dati stessa, è calcolata usando il "device", cioè il dispositivo fisico, e il "lsector". cioè il numero di settore logico.

Ogni struttura *dcache* rappresenta un blocco di dati elementare che può essere trasferito da un device; non è possibile avere due strutture che rappresentano lo stesso blocco dati, per questo è necessario che la coppia (device,lsector), usata per calcolare la chiave hash, individui in maniera univoca un blocco all'interno del dispositivo fisico (bisogna stare attenti al fatto che, ad esempio, il blocco 64 del device "ide/hda" potrebbe corrispondere al blocco 0 del device "ide/hda1", come viene spiegato nel paragrafo 3.1.2 a pagina 14, e quindi bisogna in qualche modo garantire questa unicità).

I campi presenti all'interno di questa struttura, come mostrato nella Tabella 4.17 nella pagina seguente, sono i seguenti:

prev, next e hash Sono usati per mantenere queste strutture dentro le liste ad accesso con chiave hash: *prev* e *next* sono usati per implementare una lista doppia, mentre *hash* è la chiave hash di questa struttura.

device, lsector e buffer Questi campi contengono rispettivamente il dispositivo fisico di questa "cache entry", il suo numero di settore logico e i dati presenti nel blocco fisico corrispondente; questi sono gli unici campi che le routine di alto livello dovrebbero usare, cioè questi sono gli unici campi che qualsiasi implementazione di una strategia di caching del disco deve garantire.

used É il numero di task che stanno usando questo blocco; -1 ha un significato speciale: significa che nessun task sta usando questo blocco e il blocco è libero.

time É l'istante dell'ultimo accesso al blocco.

numblocked É il numero di task che stanno aspettando che questo blocco diventi disponibile, cioè che il flag *ready* venga settato: se un task chiede di poter accedere

```
typedef struct TAGdcache {
  int         prev;
  int         next;
  int         hash;

  __dev_t     device;
  __blkcnt_t lsector;
  __uint8_t  buffer[MAXSECTORSIZE];

  int         used;
  __time_t   time;
  int         numblocked;
  __fs_sem_t sync;
  __fs_sem_t esync;
  __rwlock_t rwlock;

  __uint32_t dirty:1;
  __uint32_t ready:1;
  __uint32_t esyncreq:1;
  __uint32_t skipwt:1;

} dcache_t;
```

Tabella 4.17: Struttura dcache

ad un blocco già richiesto da un altro task, ma non ancora disponibile (poichè non ancora reso disponibile dalla parte di basso livello), il task si blocca su un semaforo di sincronizzazione ed incrementa questo campo.

sync É il semaforo di sincronizzazione usato per i task bloccati in attesa che *ready* sia settato.

esync É un semaforo di sincronizzazione sugli errori nessario durante la fase di "purging" della cache spiegato successivamente.

rwlock É un locker, vedi la sezione 4.2.2 nella pagina successiva, usato per richiedere l'accesso al blocco in lettura/scrittura.

dirty Settato se il blocco è stato modificato.

ready Settato se il blocco è disponibile.

esyncreq Settato se è richiesta la sincronizzazione sugli errori e quindi se è richiesto l'utilizzo del campo *esync*.

skipwt Indica se onorare od ignorare la gestione tramite "write throught", spiegata successivamente in questa sezione.

I campi *esync* e *esyncreq* vengono utilizzati durante la fase di "purging": un task richiede di leggere/scrivere in/su un blocco, il blocco non è già presente in cache e la cache è piena, il sistema, allora, decide di scaricare un blocco che non è attualmente usato dalla cache, secondo un determinato algoritmo; per scaricare questo blocco può essere necessario fare una scrittura sul disco e può accadere che, mentre la scrittura sul disco è in corso, un altro task richieda il blocco che si è appena deciso di liberare (proprio un worst-case); questo secondo task, più eventuali altri tasks, possono bloccarsi in attesa di sincronizzazione sul blocco che si stà liberando, si potrebbero far fallire le richieste su questo blocco in queste condizioni ma questo fallimento è necessario solo se si riesce a liberare il blocco in questione, se non si riesce a liberare il blocco non è necessario far fallire queste richieste; quindi se non si riesce a liberare il blocco il task che stava facendo il purging della cache può risvegliare questi altri task senza problemi, mentre se riesce a liberare il blocco deve svegliare gli altri task facendogli credere che si è verificato un errore e dopo può usare questo elemento di cache che è stato liberato; il problema è che questo task dopo aver svegliato tutti i task in attesa deve "aspettare" che tutti questi task abbiano fatto il test per verificare un errore; per far questo imposta il flag *esyncreq* e si blocca sul semaforo *esync*; l'ultimo task a fare il test per gli errori, trovando questo flag settato, fa un signal sul semaforo svegliando il task in attesa.

La cache può funzionare in tre modi diversi: può funzionare in "copy-back", in "write-througth" e in un write-throught modificato; nei normali sistemi time-sharing, di solito, si usa una cache per il disco usata in copy-back, cioè nella quale le scritture vengono effettuate solamente in cache, e si usa un processo di sistema a bassa priorità per scaricare la cache, cioè per sincronizzare i dati in cache modificati con quelli presenti nel disco fisso, ciò naturalmente non garantisce che ci sia spazio in cache per altri dati. In un sistema time-sharing, se viene effettuata un richiesta in cache e la cache è piena, si libera un blocco di cache, eventualmente effettuando una scrittura sul disco; in un sistema real-time, nel quale riveste una importanza fondamentale schedulare i task in maniera da garantire i requisiti del task (come la propria dead-line), il liberare un blocco effetuando una scrittura sul disco può dare dei problemi, poichè i tempi per l'accesso al disco vengono conteggiati nei tempi di calcolo del task che effettua la richiesta e quindi un task che, ad esempio, effettua solo letture può subire cospicui ritardi a causa di un altro task che riempie "velocemente" la cache mediante scritture.

Per risolvere questi problemi sono possibili due approcci: non si usa la cache (o, in altre parole, si fa in maniera che la cache funzioni in maniera "dummy", cioè riversando tutte le richieste al basso livello senza effettuare caching) o si fa funzionare la cache in modalità write-throught, cioè sincronizzando subito le scritture sulla cache con il disco, in maniera che chi "sporca" la cache è anche responsabile della sua sincronizzazione con il disco e quindi della scrittura su disco.

Usando la cache del disco in write-throught si manifesta un altro problema: spesso si modificano pochi bytes sempre sugli stessi blocchi (ad esempio per aggiornare le informazioni di un filesystem sull'allocazione di un file nel disco); usando la cache in write-throught si genererebbero continue scritture su disco, per questo si può utilizzare la terza modalità di uso della cache: in questa modalità la cache viene utilizzata in write-throught tranne che per quei blocchi per i quali è settato il flag *skipwt* che vengono gestiti in copy-back; per default, questo flag viene resettato, se un filesystem vuole disabilitare la politica di caching write-throught basta che chiami la funzione *dcache_skipwt()* sulla cache-entry prima di rilasciarla.

Struttura __rwlock_t

Sia il modulo che implementa la cache che altri modulit hanno bisogno di un "locker", cioè un lucchetto per risolvere il problema dei lettori/scrittori: ho una risorsa

```
typedef struct {
  __mutex_t  mutex;
  int        active_readers;
  int        active_writers;
  int        blocked_readers;
  int        blocked_writers;
  __sem_t    readers_semaph;
  __sem_t    writers_semaph;
} __rwlock_t;
```

Tabella 4.18: Struttura __rwlock_t

```
void __rwlock_init      (__rwlock_t *ptr);
void __rwlock_rdlock    (__rwlock_t *ptr);
void __rwlock_wrlock    (__rwlock_t *ptr);
int  __rwlock_tryrdlock (__rwlock_t *ptr);
int  __rwlock_trywrlock (__rwlock_t *ptr);
void __rwlock_rdunlock  (__rwlock_t *ptr);
void __rwlock_wrunlock  (__rwlock_t *ptr);
```

Tabella 4.19: Funzioni per i "lockers"

condivisa sulla quale ho dei processi che devono accedervi in lettura o scrittura, poichè la scrittura può rendere la struttura momentaneamente inconsistente devo imporre che:

1. Se qualcuno sta scrivendo, nessun altro è autorizzato a leggere o scrivere.

2. Se qualcuno sta leggendo, altri sono autorizzati a leggere ma non a scrivere.

Esistono vari modi per risolvere il problema: alcuni introducono "starvation" privilegiando i lettori rispetto agli scrittori, altri invece privilegiano gli scrittori, altri evitano la starvation cercando di non privilegiare nessuna classe di processi.

Questo problema è analizzato in maniera piu approfondita in [2] nel quale sono proposte varie soluzioni al problema e nel quale vengono analizzati i vantaggi e gli svantaggi delle varie soluzioni; attualmente, nel sistema, viene usato il terzo algoritmo del paragrafo 4.2.3.1 di [2]; naturalmente è possibile modificare questo algoritmo (rimpiazzando il modulo *rwlock.c*) con un altro algoritmo senza che il sistema ne risenta, a patto di mantenere la stessa interfaccia con gli altri moduli. Non verrà discussa l'implementazione fatta o la struttura interna mostrata nella tabella 4.18. Per implementare un "locker" è necessario che tutti i dati necessari siano inseriti in una struttura *__wrlock_t* e che vengono fornite le seguenti funzioni, mostrate in Tabella 4.19, alle quali viene sempre passata la struttura sulla quale agire:

__rwlock_init Deve inizializzare la struttura passata come argomento; viene sempre chiamata prima di ogni altra funzione.

__rwlock_rdlock Richiesta di lettura (read lock); quando questa funzione ritorna, il task chiamante è autorizzato a leggere i dati della risorsa condivisa protetta mediante il *__rwlock_t* passato come argomento.

__rwlock_wrlock Richiesta di scrittura (write lock).

__rwlock_tryrdlock Richiesta di lettura condizionale (try read lock); la funzione ritorna zero se il task chiamante può accedere in lettura, un altro numero in caso contrario; se un task può accere in lettura, la risorsa è già stata assegnata in lettura al task chiamante e quindi non vi è necessità di chiamare la funzione *__rwlock_rdlock()*, ma bisogna chiamare la funzione *__rwlock_rdunlock()* al termine della lettura.

__rwlock_tryrdunlock Richiesta di scrittura condizionale (try write lock); valgono le stesse regole della funzione precedente.

__rwlock_rdunlock Fine lettura (read unlock); il task ha finito di utilizzare la risorsa condivisa.

__rwlock_wrunlock Fine scrittura (write unlock); il task ha finito di scrivere sulla risorsa condivisa e la risorsa condivisa è nuovamente in stato consistente.

Le funzioni *__rwlock_rdlock()* e *__rwlock_rdunlock()* vanno sempre usate in coppia: ad ogni chiamata della prima funzione deve corrispondere una chiamata alla seconda funzione; non sono permesse chiamate innestate; lo stesso discorso vale per le funzioni *__rwlock_wrlock()* e *__rwlock_wrunlock()*.

Da notare che le funzioni di "write lock" e "read lock" possono causare un bloccaggio del task chiamante di duranta sconosciuta, anche infinita se l'algoritmo utilizzato non previene la starvation; tutte le altre funzioni non provocano un bloccaggio del task chiamante. Potrebbe essere implementato un qualche algoritmo ad hoc per sistemi realtime in maniera da privilegiare i task in base al loro tipo (se hanno la deadline imminente, se appartengono ad una classe ad alta priorità, ecc...).

4.3 Filesystem: il filesystem MS-DOS (FAT16)

4.3.1 Descrizione

Una descrizione tecnica ed approfondita del filesystem FAT16 può essere trovata in [10] o [12]. Quali sono le caratteristiche salienti di questo filesystem? Le caratteristiche migliori sono la sua semplicità, la sua diffusione e la documentazione disponibile; la sua semplicità e sia un punto di forza, l'implementazione di questo filesystem non è molto complessa, sia un punto di debolezza, le sue prestazioni e le sue potenzialità sono molto limitate: non supporta i link, non supporta i "device special file", non supporta la protezione dei files, i nomi dei files sono limitati a 8+3 caratteri (anche se esiste una estensione che permette di avere dei nomi lunghi, VFAT, al prezzo di una complicazione della struttura delle directory) e la sua organizzazione può generare una allocazione estremamente frammentata dei files.

Questo filesystem è stato pensato e realizzato per sistemi monoutete e monotask, noi invece lo utilizzeremo per un sistema monoutente, multithreads e real-time; questo comporta inevitabili problemi proprio poichè non è stato progettato per sistemi realtime come invece ad esempio il filesystem del sistema operativo RT-11 per macchine PDP-11 della Digital (del quale non ho trovato documentazione se non brevi note ed accenni); il motivo per il quale si è scelto di utilizzare questo filesystem è proprio dovuta alla sua semplicità e diffusione.

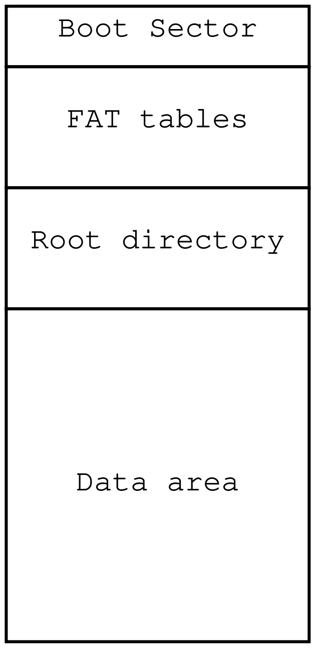

Figura 4.6: Struttura interna del filesystem MS-DOS

Struttura interna

Come mostrato nella Figura 4.6, questo filesystem è strutturato internamente in quattro "aree":

Boot sector È il settore di avviamento: contiene un breve programma per avviare il sistema operativo MS-DOS, come specificato in [10], più altre informazioni come, ad esempio, il numero di FATS, il tipo di supporto e la dimensione della "Root directory"; questo settore viene usato per mantenere quelle informazioni che, usando una terminologia *Unix*, verrebbero mantenute nel "super blocco"; queste informazioni sono mostrate con maggior dettagli nella Tabella 4.20 nella pagina successiva.

FAT tables Questa parte mantiene pìu copie della FAT, la File Allocation Table; queste informazioni servono per sapere dove si trova un file sull'hard-disk.

Root directory È la directory di root, cioè la directory principale; questo blocco è strutturao come un qualsiasi files contenente una directory, descritto più avanti, con la particolarità di essere allocato in maniera contigua e di essere di dimensione fissa.

Data area È l'area dove sono memorizzati i files, sia i files regolari che le directorries; l'unita di allocazione minima utilizzata da questo filesystem non è il blocco, o settore fisico, ma il cluster: un cluster è un insieme contiguo di settori fisici: quanti settori formano un cluster è una delle informazioni memorizzate nel boot sector.

```
struct bootrecord {
  __uint8_t  reserved[3];
  char       oemname[8];
  __uint16_t bytespersector;
  __uint8_t  sectorspercluster;
  __uint16_t hiddensectors;
  __uint8_t  fatsnumber;
  __uint16_t rootentry;
  __uint16_t sectors;
  __uint8_t  media;
  __uint16_t sectorsperfat;
  __uint16_t sectorpertrak;
  __uint16_t headsperdisk;
  __uint32_t hiddensectors32;
  __uint32_t sectors32;
  __uint16_t physicaldrive;
  __uint8_t  signature;
  __uint8_t  serialnumber[4];
  char       volumelabel[11];
  char       fattype[8];
};
```

Tabella 4.20: Il “boot sector” del filesstem MS-DOS

Il “boot sector”

La Tabella 4.20 mostra la struttura di un boot sector, non si descriverà qui il significato di tutti questi campi ma solo di alcuni più importanti; il boot sector ha sempre la dimensione di 512 bytes, la tabella mostra solo la prima parte del boot sector, la seconda parte contiene il programma di avviamento sopramenzionato.

La maggior parte di questi campi mantiene informazioni sulle dimensioni di qualche entita: *sectorpervluster* mantiene, evidentemente, il numero di settori che formano un cluster, *sectorsperfat* il numero di settori per ogni FAT, *fatsnumber* il numero di FAT presenti sul disco, ecc...

Una particolarità: quando l’MS-DOS formatta un disco (“formatta” è usato nel senso MS-DOS, cioè quando crea un filesystem FAT16), dà la precedenza a queste informazioni, se già trovate, rispetto a quelle che gli fornisce la macchina: ad esempio, se un hard-disk informa l’MS-DOS di avere 8 testine di lettura/scrittura, ma nel campo *headsperdisk* viene trovato il numero 13, l’MS-DOS si comporterà come se il disco avesse 13 testine.

La ‘”File Allocation Table”

Ogni FAT è organizzata come una tabella avente elementi di 16 bits; ogni elemento rappresenta un cluster, il valore contenuto nella tabella indica che informazioni sono mantenute nel cluster corrispondente:

- Il valore 0 indica che il cluster non è usato e quindi è disponibile.

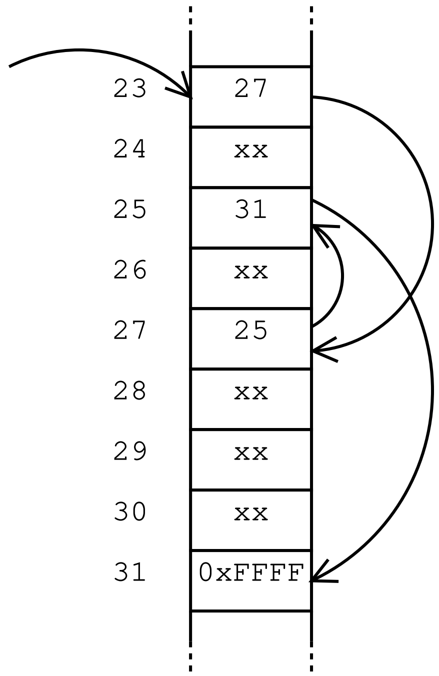

Figura 4.7: Un esempio di File Allocation Table (FAT)

- I valori nell'intervallo 0xfff0-0xfff7 sono usati per indicare un cluster riservato; ad esempio il valore 0xfff7 viene usato per indicare che il cluster non va utilizzato in quanto il supporto fisico dove dovrebbe risiedere questo cluster è danneggiato.
- I valori 0xfff8-0xffff indicano che il cluster è l'ultimo cluster di un file.
- Qualsiasi altro valore indica qual'è il successivo cluster del file.

La FAT è usata per creare le "catene" di cluster che costituiscono un file; ad esempio, nella Figura 4.7, si sono mostrate le informazioni contenute nella FAT per un ipotetico file memorizzato su 4 cluster, nei cluster 23,27,25 e 31. Come si fà a conoscere qual'è il primo cluster di un file? Queste informazioni sono contenute nelle directory.

Da notare che il primo cluster dell'area dati è il cluster numero 2, cioè non esistono i cluster 0 ed 1 ma le FAT iniziano dalla posizione 0: nei primi quattro bytes di queste tabelle sono memorizzati dei valori che indicano il tipo di FAT (esistono le FAT a 12 e 16 bits) e il formato del disco.

Le directory, tranne la "root directory", sono files normali, cioè sono memorizzate come normali files; il sistema impone una specifica struttura interna per i file che contegono directory. Per motivi di sicurezza (sicurezza intesa come "safety"), i sistemi operativi non danno mai l'accesso diretto ai files che contengono le directory, ma forniscono delle chiamate di sistema per manipolare indirettamente questi files speciali, ad esempio, non si può creare un file scrivendo informazioni direttamente all'interno di una directory ma si può utilizzare la funzione *creat()*, che indirettamente scrive informazioni all'interno di un file di directory.

```
struct directoryentry {
  __uint8_t  name[8];
  __uint8_t  ext[3];
  __uint8_t  attribute;
  __uint8_t  reserved[10];
  __uint16_t time;
  __uint16_t date;
  __uint16_t cluster;
  __uint32_t size;
};
```

Tabella 4.21: Le "direcotry entry" del MS-DOS

La "root directory"

Questa area ha la stessa struttura di un file di directory tranne il fatto che è esterna all'area dati, è allocata in maniera contigua ed ha dimensione fissa; perciò le informazioni contenute in questa sottosezione si applicano anche ai files di directory tranne il fatto che questi sono allocati nell'area dati in cluster non contigui.

Un file di directory è strutturato come una tabella di "directory entry"; ogni directory entry contiene le informazioni su uno specifico file; questa struttura è mostrata nella Tabella 4.21 e contiene le seguenti informazioni:

name e ext Contengono il nome del file nel formato classico 8+3; i caratteri non usati devono essere riempiti con degli spazi.

attribute Flags per il file: questi flags specificano se il file è regolare, è una directory, è un "volume label", è stato archiviato o è a sola lettura.

time Ora della creazione del file con una precisione di 2 secondi; lo standard *Posix*richiede la memorizzazione di tre diverse date: la data di creazione, quella di ultima modifica, quella di ultimo accesso; utilizzando un filesystem MS-DOS queste tre date vengono impostate all'unico data memorizzata per un file; la codifica di questo campo è specifico dell'MS-DOS.

date Data della creazione del file; questo campo permette di specificare date fino al 2099; al codifica del campo è specifica dell'MS-DOS.

cluster È il cluster iniziale del file; se questo valore vale 0 significa che il file è vuoto (il cluster numero 0 non esiste, il primo cluster disponibile è il cluster numero 2).

size La dimensione in bytes del file; la dimensione massima di un file è percio di 4GBytes (utilizzando la struttura FAT16 si cominciano ad avere problemi quando la dimensione dell'hard-disk supera il gigabytes).

Queste strutture contengono anche i nome dei files mentre nella maggior parte dei filesystem sviluppati per *Unix*i nomi e le informazioni sui files vengono memorizzati in parti diverse: le informazioni generali sui files vengono memorizzati negli inodes mentre i nomi dei files vengono memorizzati nelle directory.

Le entry di directory non sono memorizzate in maniera contigua: se un file viene cancellato deve essere cancellata la sua entry e le entry che seguono non vengono spostate per consentire una memorizzazione contigua; il primo carattere del nome di un file viene usato per stabilire se una entry è valida: se contiene il valore 0xe5 significa che l'entry è stata cancellata, se contiene il valore 0x00 che non è stata mai usata, ogni altro carattere indica che l'entry è valida. Il valore 0x00 viene usato per motivi di efficienza: la scansione di una directory può terminare quando si trova una entry con il primo bytes a 0x00 anche se ci sono altri settori nel cluster contenente la directory.

La "data area"

Su questa area c'è poco da dire: è divisa in cluster, cioè in un insieme di settori, parte dal cluster numero 2, nei cluster sono memorizzati i files regolari e di directory.

Per individuare un settore di un hard-disk, e quindi un cluster, si usano degli indirizzi linerari, cioè i settori di un hard-disk vengono numerati sequenzialmente partendo dal numero 0. Dove si trova il settore n-esimo? cioè in quale cilindro, in quale disco, in quale settore di una traccia si trova? Il filesystem specifica un algoritmo che associa in maniera univoca ad ogni indirizzo linerare un indirizzo C/H/S (Cylinder/Head/Sector).

Alcune implementazioni di sistemi MS-DOS compatibili suppongono che il filesystem sia strutturato in un determinato modo in funzione del tipo di supporto usato: ad esempio, su di un floppy ad alta densita da 3 pollici e 1/4, si suppone di avere cluster composti da un settore e di avere una directory principale di 14 settori. Il modo migliore di implementare un filesystem è di ignorare queste supposizioni e di utilizzare, invece, le informazioni contenute nel filesystem stesso; ad esempio e bene non supporre di avere due copie della FAT, come accade in quasi tutti i casi.

4.3.2 Note implementative

Si vuole qui accennare ad un problema che nasce quando si tenta di inserire il filesystem FAT16 all'interno di questa architettura.

Questo sistema ha due strutture dati per mantenere le informazioni sui files (*inode*) e le informazioni sui nomi dei files (*dentry*); questa separazione fra informazioni sui files e nomi dei files è stata fatta per poter espandere il sistema in maniera da poter facilmente supportare i "link"; infatti il sistema non suppone mai che fra nomi e inode ci sia una corrispondenza uno a uno.

Nel filesystem MS-DOS però c'è una corrispondenza univoca fra nomi dei file e inode infatti tutte queste informazioni si trovano nelle directory entry di questo filesystem.

Il sistema, quando deve riferirsi ad un files, lo fà attraverso il suo "file serial number", cioè con il numero di inode; questo comporta che il filesystem debba assegnare ad ogni file un numero, che questo numero sia unico all'interno del filesystem e che questo numero sia sufficiente per individuare il file all'interno del filesystem.

Per fare questo il file serial number, che è un numero a 32 bits, è stato diviso in due parti, come mostrato nella Figura 4.8 nella pagina successiva:

- La parte più significativa rappresenta il numero di cluster.
- La parte meno significativa rappresenta il numero di directory entry.

Bisogna notare che un file serial number deve servire per individuare tutte le informazioni su di un file, quindi il numero di cluster che si inserisce non è quello nel quale

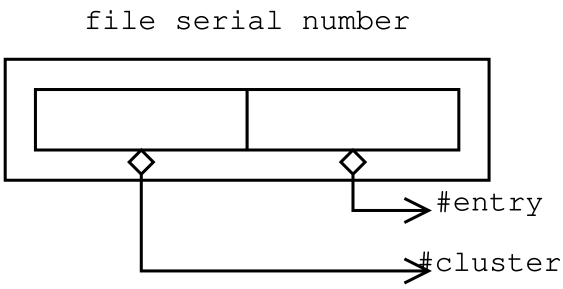

Figura 4.8: Il “file serial numeber” nella FAT16

comincia il file ma è il numero di cluster della directory che contiene il file; quindi, ad esempio, se il numero di inode vale 0x01230008, significa che ci stiamo riferendo al file la directory entry del quale si trova nell'ottava directory entry del 123esimo cluster.

Esistono due valori di cluster speciali: il cluster 1, che come spiegato precedentemente non esiste in un filesystem FAT16, è usato per indicare le directory entry contenuti nella radice (nella root directory) ed il cluster numero 0 che è usato per indicare la roor directory entry stessa.

Capitolo 5

Posix

Posix è una sigla che significa “Portable Operating System Interface”; questa sigla è stata coniata da una specifica commissione dell’IEEE (the Institute of Electrical and Electronics Engineers) per indicare una serie di documenti correlati riguardanti l’interfaccia fra un sistema operativo e le applicazioni utente, alcuni dei quali sono diventati standard ISO/IEC.

Posix si occupa solo dell’interfaccia e non di come questa interfaccia debba essere implementata, in particolare, specifica i prototipi di tutte le funzioni che una interfaccia “*Posix* compliance” deve fornire, specifica dettagliatamente la semantica di queste funzioni e specifica anche una metodologia di test per verificare la “conformance” a queste specifiche.

5.1 Posix ed i sistemi real-time

Lo standard *Posix* può essere trovato in [11]: questo documento è molto voluminoso, infatti descrive tutte le possibili funzionalità che un sistema operativo può esportare verso una applicazione, per questo è stato definito uno standard [8] che descrive quale sub-set delle funzionalità *Posix* dovrebbero essere rese disponibili in un sistema real-time.

In questo documento vengono definiti 4 diversi sub-sets (o profili di sistema) per sistemi operativi real-time:

Minimal Realtime System Profile (PSE51) Il profilo minimo, adatta per sistemi embedded o per il controllo di dispositivi di I/O speciali; in questo profilo, ad esempio, non sono richieste funzionalità per la gestione dei file o per la protezione della memoria.

Realtime Controller System Profile (PSE52) È analogo al precedente con l’aggiunta del supporto ai filesystems, cioè a tutte le funzionalità di I/O su supporti di memorizzazione di massa.

Dedicated Realtime System Profile (PSE53) Una estensione del profilo minimo nel quale è stato aggiunto il supporto di più processori, una interfaccia comune ad alcuni device drivers, un supporto più esteso per la gestione della memoria,

Multi-Purpose Realtime System Profile (PSE4) Il profilo più complesso nel quale è richiesto il supporto di buona parte dello standard *Posix* compreso il supporto al multi-threading.

Il sistema descritto in questo lavoro cerca di conformarsi, usando primitive di sistema e funzioni di libreria, al profilo “Realtime Controller System Profile (PSE52)” per conseguire gli obbiettivi già descritti nella sezione 1.1 a pagina 1; naturalmente vengono trattate solamente quelle parti riguardanti i filesystems, ad esempio le funzioni di creazioni di file, e non parti più specifiche di un sistema operativo, come la gestione dei segnali o della memoria.

Il sistema, però, non può definirsi conformato al profilo PSE52 di *Posix* poichè sul sistema non sono stati eseguiti i test di “conformance” *Posix* alcune funzionalià richieste sono mancanti e la semantica di alcune funzioni di interfaccia è stata leggermente modificata; nonostante questo siamo riusciti a compilare ed eseguire applicazioni abbastanza complesse con poco sforzo, come decodificatori MPEG e programmi di analisi FFT, inoltre tutte le principali funzionalità sono presenti e funzionanti.

La commissione che si occupa dello standard *Posix* all'interno della IEEE sta lavorando su delle estensioni allo standard ([18] e [17]) in maniera da includere funzionalità specifiche per i sistemi in tempo reale; in particolare nello standard 1003.1b annex H, che si trova in [11], vengono proposte delle funzionalità per l'I/O in sistema real-time che consentono di:

- Accedere ai file in maniera più efficiente.
- Preallocare buffers per migliorare le performance.
- Accedere direttamente al sotto-sistema di I/O.
- Usare la cache del disco in maniera “intelligente”.
- Specificare/richiedere la dimensione o l'allineamento dei trasferimenti dati.

Questa interfaccia, pur essendo stadard, è poco usata/implementata a causa della sua complessità; la commissione si sta orientando sul fornire delle funzionalita di “advice”, cioe delle funzioni che avvertono il sistema sull'uso che si intende fare di un file, come ad esempio, il tipo di accesso o la lettura a velocità costante; poi sarà responsabilità del sistema usare queste informazioni per aumentare le prestazioni. Da notare che si sta lavorando per cercare di aumentare le performance di un sistema ma non la sua prevedibilità: in un sistema real-time la prevedibilità temporale è più importante delle performance medie o di picco del sistema.

5.2 Il “Realtime Controller System Profile”

Questo profilo richiede che le seguenti funzionalità, che sono raggruppate in “classi”, siano presenti:

POSIX_FILE_SYSTEM Funzioni per rimuovere file o creare file temporanei (che si cancellano automaticamente alla chiusura).

POSIX_FILE_ATTRIBUTES Funzioni per cambiare gli attributi dei files.

POSIX_FD_MGMT Funzioni per la gestione dei descrittori dei file, come, ad esempio, la duplicazioni di descrittori o lo spostamento del file pointer di un descrittore.

POSIX_DEVICE_IO Funzioni principali per l'I/O su file, come la funzione *read()*.

Naturalmente si è fatto riferimento solo alle funzionalità che riguardano l'input/output; all'interno di queste classi di funzionalità devono essere definiti dei "flags" che determinano funzionalità aggiuntive che una classe deve fornire:

_POSIX_FSYNC Funzioni di sincronizzazione fra i buffers interni al sistema ed il dispositivo di memorizzazione.

_POSIX_SYNCHRONIZED_IO Questo identificatore richiede che vengano onorati alcuni flags come, ad esempio, il flag *O_DSYNC*.

_POSIX_ASYNCHRONOUS_IO Funzioni per l'I/O asincrono, come ad esempio la funzione *lio_listio()*; questa classe di funzione attualmente non è supportata.

_POSIX_MAPPED_FILE Questa classe di funzioni servono per poter "mappare" un file in un certo spazio di memoria; data l'alta integrazione che queste funzioni devono avere con il sistema di gestione della memoria, non sono supportate.

_POSIX_NOTRUNC I filenames non devono essere troncati.

_POSIX_FILE_LOCKING Funzioni di bloccaggio utili in sistemi multithreads.

5.3 Note implementative

5.3.1 Primitive di sistema

In questa sezione si intedono descrivere le deviazioni dello standard delle funzioni *Posix* implementate, non si intende descrivere l'implementazione di queste funzioni o la loro semantica, informazioni che possono essere trovate in [11], [13] o [16]; ne si intende fare un elenco delle funzioni implementate nel sistema.

Però è utile avere un elenco delle funzioni che dovrebbero essere implementate per aderire maggiormente allo standard:

link Attualmente non sono supportati i link hard o soft fra file anche se il sistema è stato predisposto per una espansione futura.

rmdir, mkdir e rename Funzioni "poco utili" per un sistema di studio.

fsync e fdatasync Richiedono una ristrutturazione interna del sistema per poter essere implementate in maniera efficiente.

mmap,munmap e msync L'implementazione delle funzioni per mappare i files richiede una stretta integrazione con il codice di gestione della memoria; in particolare queste funzioni sono più facilmente implementabili se si dispone della protezione della memoria.

tmpfile La funzione di libreria del OSKit (vedi 5.3.2 a pagina 79) non funziona.

Le seguenti funzioni si comportano in maniera diversa da quanto specificato dallo standard.

access()

Il flag *X_OK* ritorna sempre l'errore *EACCESS*; se il file è presente la funzione ha sempre successo.

fcntl()

Il flag *FD_CLOEXEC* anche se viene gestito viene sempre ignorato su files; le richieste di bloccaggio, fatte usando i flags *F_GETLK*, *F_SETLK* o *F_SETLKW*, ritornano sempre l'errore *EINVAL*.

fdevice()

Questa funzione non è standard *Posix*: ritorna il "device serial number", vedi la sezione 3.1.2 a pagina 14, del device passato tramite un nome simbolico.

fstat()

Ritorna l'errore EBADF se si tenta di usarla sui primi tre descrittori di file.

getumask()

Il valore della *umask* non viene usato.

mount()

Questa funzione non è standard.

open()

I seguenti flags non sono supportati e ritornano un errore se si tenta di usarli: *O_*, *O_DSYNC*, *O_NOCTTY*, *O_NONBLOCK*, *O_RSYNC* e *O_RSYNC*; i diritti di accesso sui files non sono testati,usati o impostati; non tutte le condizioni di errore sono testate; i primi tre descrittori dei files sono riservati: anche se sono liberi non vengono usati per aprire nuovi files.

read

Usando i primi tre descrittori di file si puo causare un blocco indefinito del task chiamante: si usa la funzione di sistema *keyb_getchar()* per prelevare i caratteri e questa funzione usa dei semafori interni che non sono punti di cancellazione.

umask()

Il parametro passato non viene usato.

umount()

Questa funzione non è standard.

utime()

I tempi restituiti possono essere non esatti: i tempi negli inode non vengono aggiornati nella maniera specificata nello standard.

5.3.2 La libreria “C”

Lo standard *Posix* per il profilo PSE52 stabilisce che la libreria standard “C” definita in [13] debba essere fornita; fra le funzionalità fondamentali ci sono quelle relative all'input/output ad alto livello definito nell'header file *stdio.h*; le funzioni di questa libreria andrebbero non usate in task real-time, per questo motivo, non essendo critiche, si è scelto di non implementarle ma di affidarsi ad una libreria già esistente.

Si è scelto di usare le librerie del Flux OSKit [7] [6]: queste kit è composto da tutte le librerie necessarie per implementare un sistema operativo; il problema principale di questo kit è che è stato scritto in maniera generica, non ottimizzata, solo per fornire il supporto a funzionalità non critiche di un nuovo sistema operativo, inoltre queste librerie sono molto interdipendenti.

Il porting di queste librerie non è stato molto complesso a causa del fatto che la libreria “stdio” è molto autocontenuta e utilizza per la sua implementazione funzionalità di supporto standard. Questa libreria ha il grosso difetto, per l'uso con *S.Ha.R.K.*, di non essere nativamente multithread; per questo è stata dovuta compilare con una opzione che attiva un supporto al multithreading rudimentale: praticamente molte funzioni diventano regioni critiche controllate da un semaforo di mutua esclusione, in questa maniera si è risolto il problema della consistenza delle risorse.

Capitolo 6

Il tracer

Per studiare il comportamenteo di un sistema è molto utile avere a disposizione un "tracer" cioè uno strumento per tracciare gli eventi che si verificano all'interno del sistema.

Per verificare gli algoritmi di scheduling delle richieste al disco, per conoscere i movimenti delle testine degli hard-disk o, più in generale per conoscere, l'evoluzione temporale di un sistema, cioè quale e quando un task viene schedulato, è necessario memorizzare una traccia con tutte queste informazioni in maniera trasparente, cioè in maniera automatica, cercando di perturbare il sistema il meno possibile.

Per questi motivi ho realizzato un tracer che, essendo strettamente integrato in *S.Ha.R.K.*, può esser utilizzato solo con questo specifico sistema operativo.

6.1 Caratteristiche

Le caratteristiche fondamentali di questo sottosistema devono essere:

Semplicità *S.Ha.R.K.* è un sistema estremamente modulare, ogni utente può estendere le funzionalità del kernel aggiungendo nuovi moduli, cioè nuovi algoritmi di schedulazione, nuovi modelli per la gestione di gestire risorse, nuovo tipi di oggetti per la sincronizzazione, ecc... Questo tracer è organizzato come un modulo di *S.Ha.R.K.* che fornisce le proprie funzionalità sia al kernel che agli altri moduli, per questo deve essere semplice: deve essere semplice poter tracciare qualsiasi tipo di evento e richiedere le funzionalità di questo modulo senza dover ricompilare tutto il sistema.

Modularità Esistono molti modi per poter memorizzare una serie di eventi: possiamo memorizzarli in una area di memoria, su un hard-disk, possiamo volerli trasmettere ad un altro computer; deve essere possibile estendere questo sotto-sistema in maniera da poter decidere una diversa politica di gestione degli eventi, per questo la modularità è necessaria.

Efficienza L'efficienza è un altro requisito imperativo: il tempo "perso" a gestire il tracer non deve incidere sulle altre attività del sistema in maniera che gli eventi registrati siano significativi e non influenzati dalla presenza del tracer stesso; questo, naturalmente, non è possibile in assoluto: bisogna bilanciare la modularità e l'efficienza in maniera da rendere il sotto-sistema il più modulare possibile senza inficiarne l'efficienza.

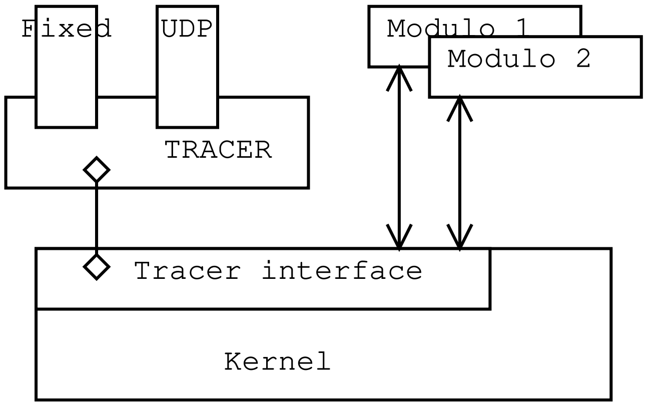

Figura 6.1: Architettura del “tracer”

6.2 Architettura

La struttura del tracer è mostrata nella Figure 6.1; il tracer è divisibile in tre sottosistemi:

1. Una parte inglobata permanentemente nel kernel che offre una interfaccia verso il kernel stesso e verso i moduli; questa parte serve per impedire di dover ricompilare tutto il kernel e tutti i moduli del kernel ogni volta che si decide di cambiare il comportamento del tracer, magari escludendolo del tutto: i kernel e i vari moduli possono sempre chiamare le funzionalità del tracer, se questo è stato compilato e collegato fornirà il proprio servizio, se questo non è stato collegato non verrà fornito nessun servizio; in questo modo non è necessario introdurre la compilazione condizionale nel kernel e nei moduli, questi devono comportarsi come se il tracer fosse sempre presente; l'overhaed di questa soluzione è trascurabile.

2. Una parte che fornisce le funzionalità generali e globali al sistema e che si integra con la parte inglobata nel kernel se il tracer viene compilato e collegato; questa parte fornisce le funzionalità generali per una corretta inizializzazione, chiusara e gestione degli eventi.

3. Dei moduli opzionali del tracer che implementano varie politiche di gestione degli eventi; il tracer utilizza varie code per gestire gli eventi ed ogni coda può essere gestita in maniera diversa: l'utente può decide quali e quante code creare, come gestirle e quali eventi registrare in ogni coda.

Per il tracer un evento è una condizione che si verifica ad un determinato istante, il tracer non interpreta gli eventi li registra solamente; in particolare, registra l'evento stesso (ogni evento è rappresentato da un numero), quando si è verificato ed eventualmente altri parametri opzionali che hanno significato solo per i programmi che leggono la traccia prodotta, ad esempio, ad un evento di “inizio schedulazione” viene associato un numero intero che rappresenta quale task sta iniziando la schedulazione.

Per motivi di efficienza gli eventi non sono modulari ma sono globali: ogni numero che identifica un evento è globale al sistema, se un modulo del kernel ha bisogno

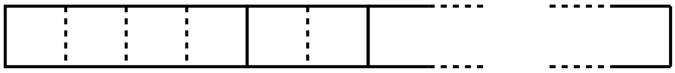

Figura 6.2: Formato delle tracce del tracer

di aggiunger altri eventi, per farlo, deve modificare il file (*include/trace/types.h*) che contiene gli identificatori di tutti gli eventi, quindi bisogna stare attenti a non duplicare i numeri usati per identificare gli eventi; non è neanche possibile utilizzare numeri casuali per identificare gli eventi poichè, per motivi implementativi, devono essere sequenziali.

Gli eventi, per favorire una gestione più semplice, sono divisi in classi: ogni classe raggruppa degli eventi aventi significati correlati; questa suddivisione in classi non implica un diverso trattamento degli eventi da parte del tracer, le classi servono solamente per poter informare il tracer come trattare determinati eventi (ad esempio se registrali o ignorali), è più semplice chiedere di non registrare una classe di eventi che chiedere di non registrare ogni singolo evento.

Attualmente sono presenti cinque classi di eventi:

tracer Gli eventi del tracer: una classe di eventi riservata per l'uso interno del tracer.

system Gli eventi di sistema sono generati dal kernel di *S.Ha.R.K.*e riguardano principalmente la gestione dei task: creazione, terminazione, attivazione, schedulazione, ecc...

user La classe degli eventi utente è riservata per l'uso da parte delle applicazioni di un utente; sono eventi che non hanno nessun significato e che non vengono mai generati all'interno di *S.Ha.R.K.*.

ll La classe degli eventi di basso livello; attualmente questa classe comprende un solo evento: l'evento "interrupt" generato durante la ricezione di una richiesta di interruzione.

semaph Per gli eventi semaforici: generati durante le wait, le signal, ecc...

6.3 Implementazione

6.3.1 Formato delle tracce

In maniera indipendente da come le code di eventi vengono gestite, alla terminazione del sistema le code vengono trasferite su di un supporto permanente (una file su di un hard-disk), in maniera da poter essere analizzate.

In Figura 6.2 è visibile il formato di ogni "evento" registrato su di un file:

1. L'istante temporale nel quale l'evento si è verficato, espresso come un numero in complemento a 2 su 32 bit in ordine "big endian" espresso in microsecondi.

2. Il numero che identifica l'evento, espresso come il campo precedente solamente su 16 bit.

```
/* generics event */
typedef union TAGtrc_allevents {
  trc_tracer_event_t trc;
  trc_system_event_t sys;
  trc_user_event_t   usr;
  trc_ll_event_t     ll;
  trc_sem_event_t    sem;
} trc_allevents_t;

/* event struct */
typedef struct TAGtrc_event_t {
  u_int32_t       time;
  u_int16_t       event;
  trc_allevents_t x;
} trc_event_t;
```

Tabella 6.1: Struttura di un evento del tracer

3. Una parte contenente informazioni opzionali per l'evento ma che comunque sono sempre presenti sul file, cioè ogni evento occupa sul file sempre lo stesso numero di bytes, indipendentemente dal fatto che la parte opzionale contenga informazioni rilevanti.

Nella Tabella 6.1 è mostrata la struttura usata dal tracer per memorizzare un evento, questa struttura è memorizzata nel file *include/trace/types.h* che è stato costruito in maniera da poter essere incluso in qualsiasi programma C ANSI dopo aver definito i tipi *u_int32_t* e *u_int16_t* rispettivamente a interi senza segno in complemento a 2 di 32 e 16 bits.

La lunghezza in bytes della parte contente informazioni opzionali può essere ricavata includendo questo file e usando l'operatore *sizeof()* sulla struttura *trc_allevents_t*, in altre parole, la lunghezza totale di ogni pacchetto di dati nel file che contiene la traccia è di *sizeof(trc_event_t)* bytes.

Come realizzare un analizzatore di tracce

Realizzare un analizzatore di traccie è estremamente semplice:

1. Si include il file *include/trace/types.h* dopo aver definito, come precedentemente specificato, i tipi *u_int8_t*, *u_int16_t* e *u_int32_t*.

2. Si legge il file contenente la traccia memorizzando i dati letti in una variabile di tipo *trc_event_t*.

3. Si analizzano i dati contenuti nella variabile per ricavarne le informazioni cercate.

Si possono usare come esempio i programmi *tdump* e *jdump* contenuti nella directory *examples/trace/* di *S.Ha.R.K.*: il primo programma mostra sullo schermo il

```
/* funzioni per l'input*/
int read_trace (char *filename,
                int (*func)(trc_event_t *))

/* funzioni informative */
int event_class  (int ev)
char *event_name (int ev)
```

Tabella 6.2: Funzioni di utilità per il tracer

contenuto di un file di traccia, il secondo programma converte la traccia in una traccia utilizzabile con un visualizzatore grafico (*jtracer*).

Questi due programmi utilizzano un modulo di nome *util.c* contenuto nella stessa directory che esporta alcune funzioni di utilità mostrate nella tabella 6.2:

read_trace Questa funzione è in grado di leggere un file di traccia, il nome del quale è passato come parametro, chiamando la funzione indicata per ogni evento trovato; alla funzione chiamata viene passato l'evento trovato; questa funzione deve ritornare zero se vuole continuare a processare il file, un altro numero se vuole terminare.

event_class Trova la classe di un evento.

event_name Trova il nome di un evento e lo ritorna come una stringa.

6.3.2 Le code di eventi

L'architettura semi-modulare di questo tracer permette di avere diverse politiche di gestione delle code degli eventi; attualmente sono state implementate quattro diverse politiche di gestione degli eventi:

dummy Una coda di tipo dummy è una coda nella quale gli eventi non vegono memorizzati: la coda si comporta come un "pozzo"; il modulo nel quale viene implementata questa politica (*trcdummy.c* e *qdummy.h*) può essere usato come base per la realizzazione di nuovi moduli.

fixed Una coda di dimensioni fisse che vengono decise durante la creazione della coda; gli eventi vengono memorizzati fino al riempimento della coda, dopodichè gli eventi vengono persi.

circular Una coda avente una dimensione fissa gestita in maniera circolare; esistono due tipi di code circolari: quelle nelle quali il salvataggio delle informazioni è fatto on-line e quelle nel quale le informazioni vengono salvate durante la terminazione del sistema; in queste ultime, quando la coda si riempie, gli ulteriori eventi vengono memorizzati cancellando gli eventi più vecchi.

Nelle code on-line il salvataggio viene effettuato tramite un task che funziona da server: gli eventi vengono memorizzati nella coda, quando la coda si riempie, gli eventi successivi vanno persi (si aspetta che la coda si svuoti); il server cerca di fare in modo che la coda non si riempia mai scrivendo i dati della coda su di un

```
int trc_logevent(int event, void *info);

int trc_suspend (void);
int trc_resume  (void);
```

Tabella 6.3: Le funzioni di "logging"

file durante il funzionamento del sistema; naturalmente non c'è nessuna garanzia sul fatto che non venga perso nessun evento. Bisogna notare che il task che funziona da server può interferire con il funzionamento del sistema specialmente se altri task utilizzano il filesystem.

udp Tutti gli eventi che arrivano in questa coda vengono spediti, utilizzando il protocollo udp/ip, verso un'altra macchina (attualmente *S.Ha.R.K.* supporta un piccolo numero di schede di rete ethernet).

6.3.3 Espansione del tracer

Con espansione del tracer si intende come il tracer può essere esteso per registrare nuove classi di eventi o come creare nuove politiche di gestione delle code.

Il tracere esporta tre funzioni per l'uso all'interno del kernel e degli altri moduli mostrate nella tabella 6.3:

trc_logevent È la funzione principale di questo sottomodulo: quando viene chiamata registra l'evento nell'appropriata coda; questa funzione deve essere chiamata per generare gli eventi; il secondo parametro, se l'evento non ha parametri opzionali, può essere NULL.

trc_suspend Sospende la registrazione di tutti gli eventi; ritorna zero in caso di successo e un numero diverso da zero in caso di fallimento; il kernel chiama questa funzione prima di entrare nella fase di shutdown.

trc_resume Riprende la registrazione degli eventi che era stata interrotta dalla funzione precedente.

Registrare nuove classi di eventi

Come spiegato precedentemente nella sezione 6.2 a pagina 81, gli eventi non sono stati modularizzati ma sono globali a tutto il sistema, per questa ragione aggiungere una nuova classe di eventi o modificare una classe già esistente significa modificare dei file interni al tracer; nonostante questo l'aggiunta o modifica di una nuova classe è abbastanza semplice.

Bisogna modificare il file *include/trace/types.h* per aggiungere i nuovi identificatori di eventi e una nuova struttura dati:

- Bisogna aggiungere dei nuovi identificatori di eventi in maniera sequenziale rispetto a quelli già presenti, bisogna aggiungere un altro identificativo di classe (*TRC_CLASS_???*) e modificare di conseguenza le costanti *TRC_F_LAST*, *TRC_NUMCLASSES* e *TRC_NUMEVENTS*.

```
typedef struct TAGtrc_queue_t {
  int          type;
  trc_event_t  *(*get)(void *);
  int          (*post)(void *);
  void         *data;
} trc_queue_t;

int trc_register_queuetype(
            int queuetype,
            int(*create)(trc_queue_t *,void *),
            int(*activate)(void *,int),
            int(*terminate)(void *)
            );
```

Tabella 6.4: La struttura trc_queue_t

- Bisogna creare una nuova struttura dati per gli evntuali parametri opzionali di questa classe di eventi ed inserire nella union *trc_allevents_t* una variabile ti tale tipo.
- Bisogna modificare la tabella *classtable* all'interno del file *trace.c* in maniera da includere il nuovo identificativo di classe.

In questo modo si è creata una nuova classe di eventi, naturalmente i files di traccia generati potrebbero non essere compatibili con quelli precedenti ed è necessario ricompilare tutto il sistema. Questi nuovi eventi possono essere usati nel modulo per il quale sono stati creati o anche in altre parti del kernel.

Inserire nuovi moduli per le code

Per inserire nuovi moduli per le code è necessario conoscere l'interfaccia fra la parte generale del tracer ed i moduli e l'inizializzazione di tutto il sottosistema, che verrà descritta dettagliatamente nella sezione 6.4 nella pagina successiva. Per ora basti sapere che ogni modulo dispone di una funzione di inizializzazione che deve registare il tipo di coda.

La funzione di registrazione è mostrata nella tabella 6.4 insieme alla struttura *trc_queue_t*; questa funzione deve essere chiamata durante la fase di inizializzazione passandole un numero, che specifica il tipo di coda, e tre funzioni che vengono utilizzate dalla parte generale del tracer per i seguenti scopi:

create Chiamata quando l'utente richiede la creazione di una nuova coda di quello specifico tipo; gli viene passato un puntatore alla struttura *trc_queue_t* che deve essere inizializzato e un puntatore a dati specifici di quel tipo di coda; l'inizializzazione è descritta in seguito.

activate Chiamata per attivare ogni coda quando il sistema è già in multiprogrammazione; gli viene passato un puntatore al campo *data* della coda e un numero progressivo di identificazione.

```
int TRC_init_phase1_standard(void);
int TRC_init_phase2_standard(void);
```

Tabella 6.5: Le funzioni di inizializzazione del tracer

terminate Chiamata prima che il sistema entri nella fase di shutdown; gli viene passato, come nella funzione precedente, un puntatore al campo *data*.

Durante la fase di creazione di una nuova coda, cioè quando il tracer generico chiama la funzione *create()* di uno specifico tipo di coda, il modulo di gestione delle coda deve riempire i campi della struttura *trc_queue_t* passata come parametro. Questi campi, mostrati nella Tabella 6.4 nella pagina precedente, servono alla parte generale del tracer per registrare gli eventi:

type È il tipo di coda.

get Questa funzione viene chiamata quando il sotto-sistema sta processando un nuovo evento: deve ritornare un puntatore ad una struttura *trc_event_t* che la parte generica riempirà con i dati dell'evento, può ritornare NULL se la coda non è in grado di accogliere un altro evento; a questa funzione viene passato un puntatore al campo generico *data*.

post Chiamata dopo che la parte del tracer generico ha riempito la struttura dati ritornanta dalla funzione *get()*; questa funzione deve "memorizzare" l'evento; "memorizzare" può significare spedire l'evento ad un altro computer, trasferire l'evento su un file, ecc...

data Puntatore ad una generica struttura dati contenente tutte le informazioni per gestire quel particolare tipo di coda.

Le funzioni *get()* e *post()* sono chiamate dall'interno del kernel e dei moduli ad interruzioni disabilitate, per questo devono essere molto efficienti se non si vuole pertubare il funzionamento del sistema; in particolare non devono bloccarsi.

6.4 Inizializzazione ed uso

Esistono due modi per attivare il tracer all'interno di *S.Ha.R.K.*: un modo semplice ma poco flessibile ed un modo completamente controllabile dall'utente e quindi più complesso.

Il primo modo consiste nell'utilizzare le due funzioni mostrate nella Tabella 6.5: la prima funzione va chiamata durante la fase di registrazione dei moduli di *S.Ha.R.K.* (prima possibile), la seconda va chiamata durante la fase di inizializzazione del sistema prima della chiamata alla funzione *main()* (dopo tutte le altre funzioni di inizializzazione , poichè la gestione delle code potrebbe richiedere le funzionalità di qualche altro modulo).

L'inizializzazione standard consiste nel creare due code standard: la prima di tipo "fixed", la seconda di tipo "circolar"; nel rendere tracciabili gli eventi di sistema e nell'assegnarli alla prima coda.

Le funzioni mostrate nella Tabella 6.6 nella pagina successiva servono per modificare alcune il modo di tracciare gli eventi:

```
int trc_assign_event_to_queue(int event, int queue)
int trc_assign_class_to_queue(int class, int queue)

int trc_notrace_event(int event)
int trc_trace_event(int event)

int trc_notrace_class(int class)
int trc_trace_class(int class)
```

Tabella 6.6: Le funzioni di utilità del tracer

```
int TRC_init_phase1(void);
int TRC_init_phase2(void);

int trc_create_queue(int queuetype, void *arg);
```

Tabella 6.7: Le funzioni di "logging"

trc_assign_event_to_queue Serve per assegnare un certo evento ad una certa coda; le code vengono identificate mediante un "handler" ritornato dalla funzione *trc_create_queue()*; usando l'inizializzazione standard alla prima coda viene assegnato 0 come handler e 1 alla seconda.

trc_assign_class_to_queue Assegna una classe di eventi ad una coda, cioè fa in modo che tutti gli eventi di quella particolare classe vengano registrati in quella particolare coda.

trc_notrace_event e trc_trace_event Servono per informare il tracer di non tracciare o di tracciare gli eventi passati come parametri.

trc_notrace_class e trc_trace_class Attivano e disattivano il tracciamento degli eventi di una determinata classe.

L'inizializzazione flessibile avviene utilizzando le funzioni della Tabella 6.7 e le funzioni già descritte e mostrate nella tabella 6.6; si chiama la funzione *TRC_init_phase1()* durante la fase di registrazione dei moduli di *S.Ha.R.K.*, quindi si registrano i tipi di code usati, si creano le code necessarie, si assegnano gli eventi alle varie code; infine si chiama la funzione *TRC_init_phase2()* prima di chiamare il *main()*. La funzione *trc_create_queue()* viene chiamata passandole il tipo di coda da creare e un puntatore a parameri specifici della coda o a NULL se si intendono utilizzare i dati di default; questa funzione ritorna un numero negativo in caso di errore, un numero maggiore o uguale a 0 in caso di successo, questo numero rappresenta l'handle della coda da utillizare con le funzioni della Tabella 6.6.

Le funzioni per registrare ogni tipo di coda ed i parametri che si posssono utilizzare durante la creazione di queste code sono descritte nei prossimi paragrafi.

```
typedef struct TAGtrc_fixed_queue_args_t {
  char    *filename;
  int     size;
} TRC_FIXED_PARMS;

#define trc_fixed_default_parms(m)
#define trc_fixed_def_filename(m,s)
#define trc_fixed_def_size(m,s)

int trc_register_fixed_queue(void);
```

Tabella 6.8: La coda "fixed" del tracer

Coda "dummy"

Non ha parametri opzionali e la funzione di registrazione *trc_register_dummy_queue()* non accetta parametri e riesce sempre ritornado il valore zero.

Coda "fixed"

Nella Tabella 6.8 è mostrata la funzione per registrare questo tipo di coda e la struttura dati che può esser utilizzata per inizializzare la coda; questa struttura dati andrebbe manipolata solamente tramite le macro elencate nella stessa Tabella:

trc_fixed_default_parms Memorizza i parametri di default nella struttura passata come parametro: imposta filename a NULL e size a 8192.

trc_fixed_def_filename Definisce il nome del file sul quale salvare la traccia; se viene settato a NULL viene scelto il nome "fix" seguito da un numero progressivo.

Coda "circular"

Come descritto in 6.3.2 a pagina 84, la coda di tipo "circular" esiste in due varianti: nella prima variante la coda viene salvata durante la terminazione del sistema mentre nella seconda variante la coda viene salvata durante l'evoluzione del sistema, cioè viene salvata online.

Nella Tabella 6.9 nella pagina successiva è mostrata la struttura dati utilizzabile durante la creazione di una coda di questa classe; anche in questo caso la struttura andrebbe manipolata tramite le macro mostrate nella stessa tabella:

trc_circular_default_parms Setta i parametri di default elencati in seguito.

trc_circular_def_filename Definisce il nome del file nel quale salvare la traccia; se NULL, come descritto precedentemente, si utilizza un nome di default.

trc_circular_def_size Dimensione della coda che vale 8192 per default.

trc_circular_def_onlinetask Setta un flag in maniera che la coda venga salvata online.

trc_circular_def_period È il periodo dell'eventuale task che salva la coda online; per default vale 500 millisecondi. Il task per il salvtaggio online è un task periodico soft.

```
typedef struct TAGtrc_circular_queue_args_t {
  char *filename;
  int  size;

  long period;
  long slice;

  int  flags;
} TRC_CIRCULAR_PARMS;

#define trc_circular_default_parms(m)

#define trc_circular_def_filename(m,s)
#define trc_circular_def_size(m,s)

#define trc_circular_def_onlinetask(m)
#define trc_circular_def_period(m,p)
#define trc_circular_def_slice(m,s)

int trc_register_circular_queue(void);
```

Tabella 6.9: La coda “circular” del tracer

trc_circular_def_slice È la fetta temporale assegnata a questo task dallo scheduler, cioè è il suo tempo di calcolo; per default viene impostato a 25 millisecondi.

Capitolo 7

Conclusioni

In questo ultimo capitolo si vuole mostrare i risultati di alcuni esperimenti effettuati su questo sistema per metterne in mostra alcune caratteristiche e limiti.

7.1 Test: device driver

Gli esperimenti sono stati effettuati utilizzando un computer avente le seguenti caratteristiche: scheda madre basata sul chipset MVP3 della Via Technologies con processore K6-2 a 500 MHz della AMD, hard-disk Quantum Fireball ST da 3.2 GBytes della Quantum.

Le caratteristiche di questo hard-disk più rilevanti dichiarate dal produttore sono mostrate nella tabella 7.1 nella pagina seguente; sono stati effettuati dei test per mostrare le performance, intese come velocità di trasferimento sequenziale massima, ottenibile da questo hard-disk nel sistema descritto precedentemente. Come metro di paragone sono state prese le performance ottenibili su un sistema *Debian/Linux* con kernel 2.2.14.

Nella tabella 7.2 nella pagina successiva sono mostrate le velocità di trasferimento dati di picco ottenute in varie condizioni operative; i test sono stati effettuati utilizzando il programma standard di *Linux* “hdparm” che può essere usato per cambiare le caratteristiche dei device drive oltre che per fare test di velocità; questi test sono stati fatti con il sistema in modalità monoutente cercando di fermare più servizi possibile.

Il programma “hdparm” efettua un test accedendo direttamente ad un device driver senza passare attraverso la cache e “mappando” il file (il block special file che rappresenta il device driver) direttamente in memoria tramite la funzione *mmap()*; i trasferimenti vengono effettuati facendo richieste da 1Mbytes in maniera sequenziale in maniera da ottenere le velocià di punta.

Le prove di questo sistema sono state fatte con il sistema operativo *S.Ha.R.K.* utilizzando un programma scritto per questo scopo; viene sfruttato il fatto che *S.Ha.R.K.*non protegge la memoria del sistema per usare direttamente le funzionalià a basso livello di lettura di blocchi, cioè viene utilizzata la funzione *bdev_read()*, per leggere blocchi consecutivi dello stesso device driver.

Confrontando queste due tabelle si può notare che nelle migliori condizioni possibili *Linux*ha una velocità quasi doppia: cerchiamo di capire il perchè.

Nella tabella 7.2 nella pagina seguente sono riportati i test fatti in varie modalità operative: usando la modalità DMA e usando l modalità PIO, abilitando e disabilitando

Cilindri	6256
Testine	16
Settori	63
Seek time medio	10 ms
Seek time traccia/traccia	2 ms
Seek time completo	20 ms
Latenza media rotazionale	5.56 ms
Velocità di rotazione	5400 RMP
Trasferimento dati di picco	16.6 MB/s
Buffer	128 KB

Tabella 7.1: Caratteristiche dell'hard-disk "Quantum Fireball St"

Condizioni				Risultati (MB/s)			
dma	drive ahead	kern ahead	multi	1st	2nd	3rd	media
1	0	0	0	3.33	2.26	2.26	2.62
1	0	0	0	X	X	X	?
0	1	0	0	5.01	5.12	5.05	5.06
1	1	0	0	X	X	X	?
0	0	1	0	2.26	2.26	2.26	2.26
1	0	1	0	X	X	X	?
0	1	1	0	5.18	5.16	5.05	5.13
1	1	1	0	8.76	8.91	8.89	8.85
0	1	1	1	5.45	5.54	5.51	5.50
1	1	1	1	8.91	8.86	8.90	8.89

Tabella 7.2: Device driver performance con Linux (1)

il "read-a-head" dell'hard-disk (in modalità "read-a-head" l'hard-disk memorizza nella propira cache interna i settori successivi a quello reichiesto'), abilitando e disabilitando il "read-a-head" del kernel di *Linux*(il device driver di *Linux* è in grado di richiedere settori successivi a quello richiesto), e abilitando e disabilitanto il "multi-sector transfert" (l'hard-disk può generare un solo interrupt per un trasferimento di settori multipli). Alcune prove, quelle contrassegnate con una "X", sono fallite: il driver andava in timeout, provocando un reset software delle periferiche connesse al bus ed un restart in modalità PIO.

Da questi dati mettono in evidenza che:

1. Il "multi-sector transfert" non ha un impatto significativo sulle performance.
2. La modalità ultra-dma dell'hard-disk ha un grosso impatto sulle performance complessive: disabilitandola si ha un calo del 40% circa.
3. Il "read-a-head" dell'hard-disk contribuisce in buona parte alle performance complessive.

Nella Tabella 7.4 nella pagina successiva sono mostrati i test effettuati con una versione di "hdparm" modificata per far si che le richieste vengano effettuate non con blocchi di 1 MBytes ma a blocchi di 512 bytes (un settore), come attualmente avviene su *S.Ha.R.K.*.

Condizioni				Risultati (MB/s)			
wcet (us)	dl (ms)	banda	dma	1st	2nd	3rd	media
200	7000	2.8%	0	2.393	2.394	2.392	2.393
200	7000	2.8%	1	4.668	4.642	4.671	4.660
200	2000	10.0%	1	4.677	4.671	4.671	4.775
100	500	20.0%	1	4.658	4.671	4.681	4.670

Tabella 7.3: Device driver performance con Hartik (1)

Condizioni				Risultati (MB/s)			
dma	drive ahead	kern ahead	multi	1st	2nd	3rd	media
0	0	0	0	0.09	0.09	0.09	0.09
1	0	0	0	X	X	X	?
0	1	0	0	5.00	4.76	4.65	4.80
1	1	0	0	X	X	X	?
0	0	1	0	0.17	0.17	0.17	0.17
0	1	1	0	5.24	5.24	5.28	5.25
1	1	1	0	8.91	8.86	8.72	8.83
0	1	1	1	6.32	6.20	6.36	6.29
1	1	1	1	X	X	X	?
1	1	1	1	8.86	8.89	8.86	8.87

Tabella 7.4: Device driver performance con Linux (2)

Da questa tabella si può notare che la dimensione del blocco da trasferire non ha nessun impatto nella modalità DMA ma l'ha quando il trasferimento avviene in modalità PIO: si guadagna circa 1 MB/s in velocità. Si può anche notare il crollo delle prestazioni in alcune condizioni dovute probabilmente alla non ottimizzazione delle tempistiche del driver in situazioni particolari mai usate.

Nella Tabella 7.3 i test effettuati su *S.Ha.R.K.* sono stati eseguiti in varie modalità operative; il sistema è configurato con quattro livelli di schedulazione: un EDF, un RR con uno slice temporale di 10ms, un CBS e uno schedulatore dummy. Il task che effettua la lettura è di tipi NRT (Non Real Time) mentre il server che gestice il device driver è un task di tipo soft, quindi schedulato nel livello del CBS, con varie configurazioni temporali, come riportato nella tabella stessa; il test è stato effettuato sia in modalità DMA che in modalità PIO.

Com'era logico aspettarsi cambiando le caratteristiche del server non si ottiene un cambiamento delle performance mentre si ottiene un cambiamento simile a quello di *Linux* passando dalla modalità DMA alla modlità PIO.

Si può quindi concludere che le performance migliori di *Linux* sono dovute principalmene alla modalità ultra-DMA ed al "read-a-head" dell'hard-disk; implementando queste feature nel driver di *S.Ha.R.K.* probabilmente si otterrebbero risultati simili a quelli di *Linux*.

Per mettere in risalto la relazione fra la banda assegnata al server che serve le richieste del device driver e la banda totale disponibile nel sistema si sono tipetuti i test effettuati usando un sistema "carico": si sono usati dei task soft aventi certe caratteristiche temporali (*wcet2*, worst case execution time e *dl2* dead line) per saturare il sistema fino ad un certo livello; il task che serve il device driver ha sempre un wcet di $200\mu s$;

Condizioni					Risultati (MB/s)			
dl	banda	wcet2	dl2	banda	1st	2nd	3rd	media
7000	2.8%	25	500	0%	4.656	4.67	4.667	4.664
7000	2.8%	25	500	10%	3.637	3.647	3.636	3.640
7000	2.8%	25	500	25%	3.031	3.031	3.044	3.035
7000	2.8%	25	500	50%	1.982	1.980	1.978	1.980
7000	2.8%	25	500	75%	0.966	0.966	0.966	0.966
2000	10.0%	25	500	0%	4.668	4.649	4.651	4.656
2000	10.0%	25	500	10%	3.631	3.651	3.638	3.640
2000	10.0%	25	500	25%	3.030	3.023	3.023	3.025
2000	10.0%	25	500	50%	1.970	1.985	1.981	1.979
2000	10.0%	25	500	75%	0.965	0.966	0.966	0.966
2000	10.0%	0.5	10	0%	4.668	4.649	4.651	4.656
2000	10.0%	0.5	10	10%	4.054	4.071	4.065	4.053
2000	10.0%	0.5	10	25%	3.240	3.231	3.253	3.241
2000	10.0%	0.5	10	50%	2.040	2.030	2.032	2.034
2000	10.0%	0.5	10	75%	0.946	0.942	0.952	0.947

Tabella 7.5: Device driver performance con Hartik (2)

Condizioni	Risultati (MB/s)			
banda	1st	2ns	3rd	media
0%	4.061	4.067	4.069	4.066
10%	3.811	3.815	3.810	3.812
25%	3.273	3.273	3.276	3.274
50%	2.224	2.230	2.223	2.226
75%	1.177	1.177	1.178	1.177

Tabella 7.6: Device driver performance con Hartik (3)

nella Tabella 7.5 sono mostrati i risultati (i tempi sono sempre espressi in millisecondi se non diversamente specificato): le prestazione descrescono in maniera simile tranne nel caso nel quale si usano task con un breve periodo per disturbare il sistema, in quest'ultimo caso le prestazioni diminuiscono più lentamente.

Nella Tabella 7.6 si sono ripetuti i test con un'altra variante: il task che legge non è più un NRT ma è un soft con un tempo di calcolo di 25ms ogni 500ms: si può notare che rispetto ai casi precedenti le performance massime sono più basse ma le minime sono più alte; probabilmente questo è dovuto al fatto che essendo un task soft una certa banda gli viene riservata anche quando il sistema è carico.

7.2 Test: politiche di schedulazione

In questa ultima sezione si vuole mostrare come sia semplice effettuare delle misurazioni utilizzando il tracer descritto nel Capitolo 6 a pagina 80; non si vuole analizzare in profondità le caratteristiche delle politiche di schedulazione del basso livello.

In particolare si vuole mostrare come si sposta la testina di lettura durante il funzionamento del sistema: per fare questo si è inserito una linea di codice all'interno del modulo che gestisce le richieste del driver IDE in maniera che venga generato un evento ogni volta che viene accettata una richiesta di lettura.

Per effettuare i test si sono usati 7 task che, contemporaneamente, cercano di leggere diversi files per 10 volte consecutive; i task sono tutti di tipo NRT e sono schedulati utilizzando un Round-Robbin con 10ms si quanto temporale; il sistema dispone di 4MBytes di cache per l'accesso al disco.

Nelle pagine successive sono riportati i grafici ottenuti con diversi politiche di schedulazione: sulle ordinate è riportato il tempo in millisecondi e sulle ascisse il cilindro letto in quell'istante, cioè il cilindro che si è deciso di leggere ma che non è stato ancora letto; i grafici utilizzano scale diverse.

Da questi grafici si può notare che i files che si stanno leggendo sono concentrati in tre aree non contigue: vicino ai cilindri 1010, 1090 e 1110. Il FCFS si comporta in maniera piu regolare del SSTF: il SSTF ha richieste sovrapposte; il SSTF è anche il più veloce: l'applicazione termina prima ma, si nota, che alcuni task sono stati arbitrariamente ritardati. Il LOOK è testato con tre varianti:

"A" L'algoritmo LOOK senza possibilità di starvation.

"B" L'algoritmo LOOK con starvation.

"C" L'algoritmo LOOK con starvation con una cache 4 volte superiore al normale.

Si può notare che le prestazioni, intese come tempo per completare l'applicazione, crescono, senza mai raggiungere, però, le prestazioni del SSTF; l'algoritmo LOOK cerca di bilanciare il tempo di acceso al disco senza ritardare in maniera eccessiva i task con richieste lontane dalla posizione corrente. La cache gioca un ruolo fondamentale: all'inizio dei grafici la testina dell'hard-disk viene portata sui primi cilindri, che probabilmente contengono informazioni sui filesystem, per poi non farvi ritorno; queste informazioni sono mantenute in cache, se non vi fosse una cache le prestazioni decaderebbero dovendo il sistema leggere continuamente informazioni sui primi cilindri; si può inoltre notare che la cache favorisce una riduzione dei tempi di calcoli di circa il 15%.

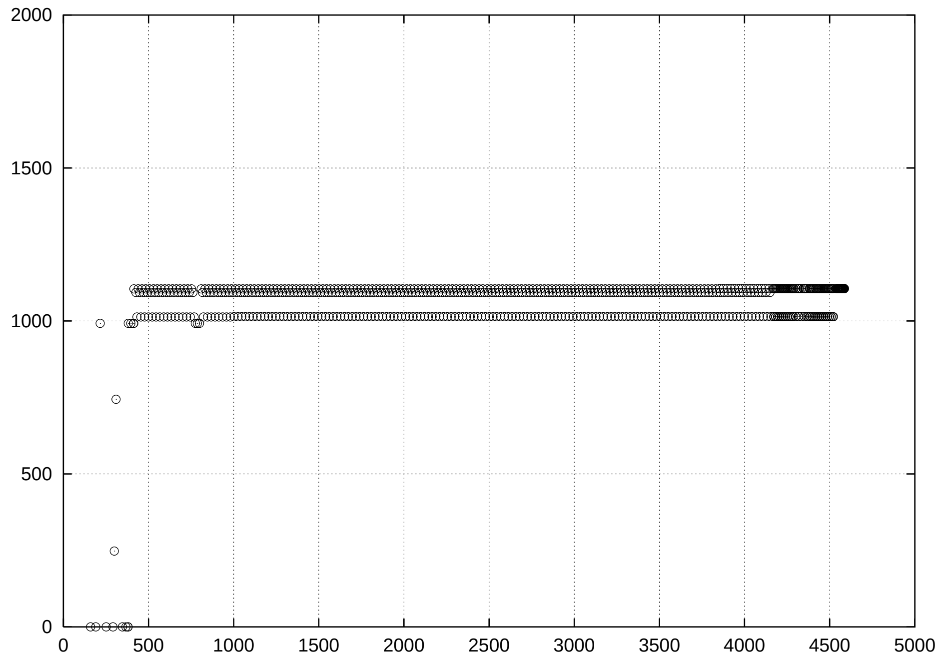

Figura 7.1: Test: richieste utilizzando FCFS

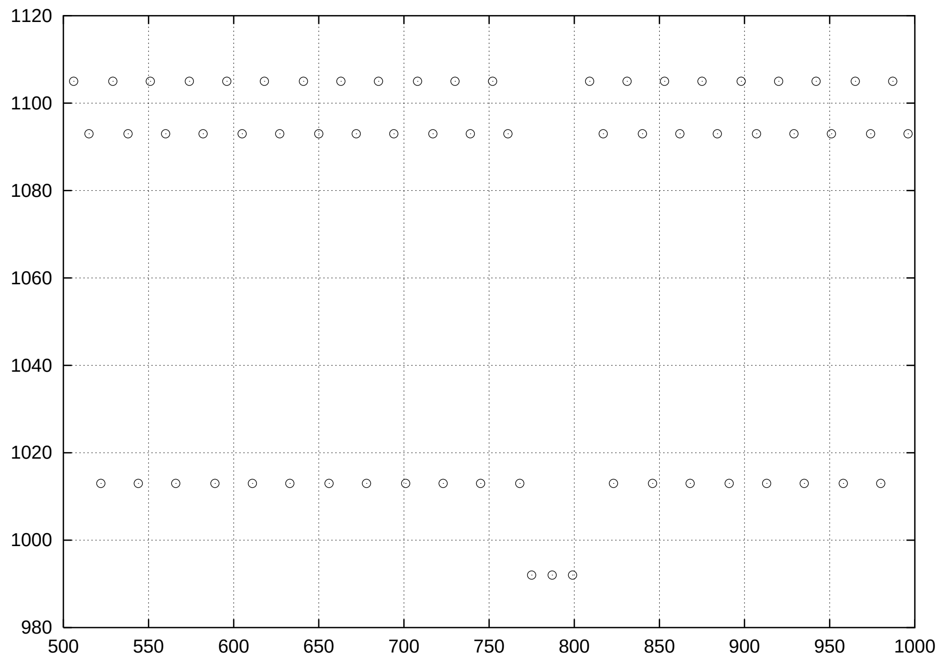

Figura 7.2: Test: richieste utilizzando FCFS (particolare)

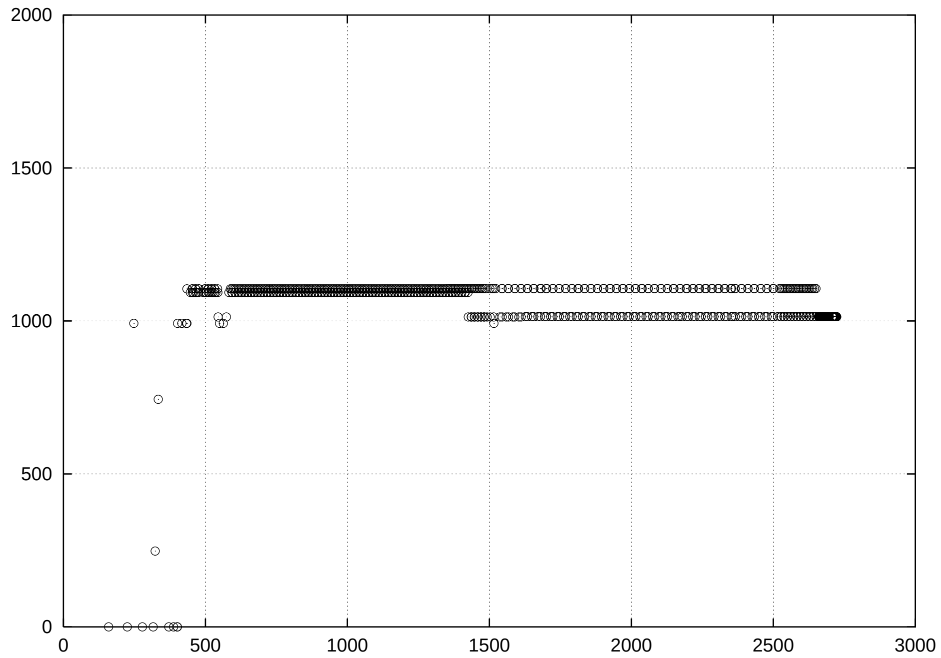

Figura 7.3: Test: richieste utilizzando SSTF

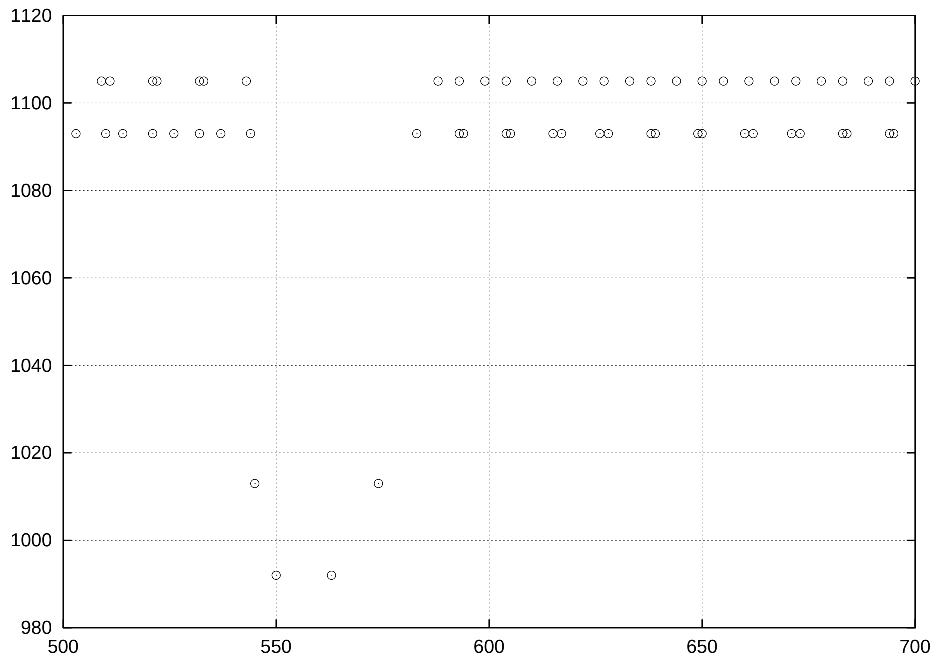

Figura 7.4: Test: richieste utilizzando SSTF (particolare)

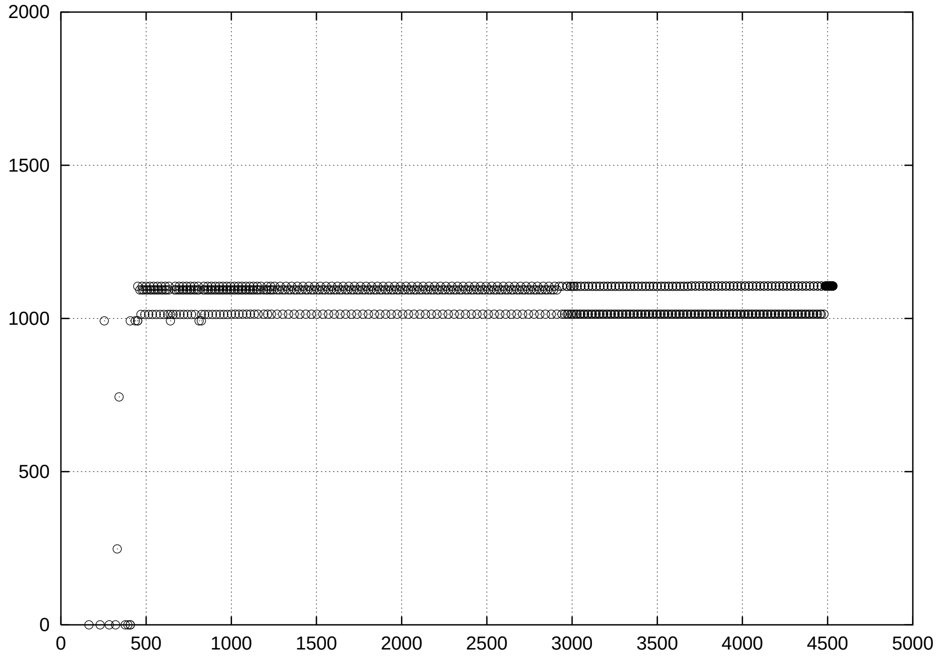

Figura 7.5: Test: richieste utilizzando LOOK “A”

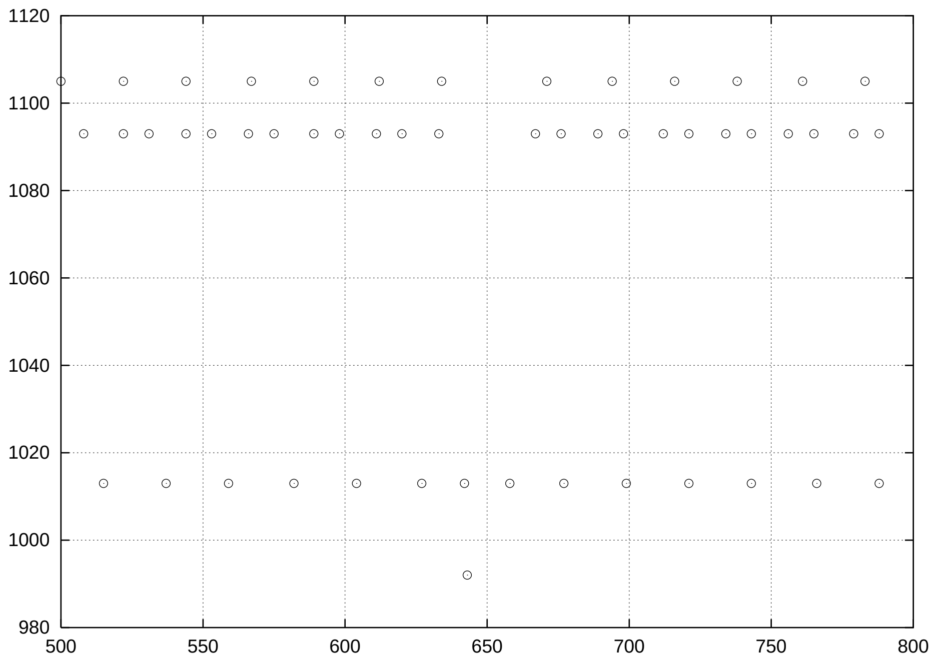

Figura 7.6: Test: richieste utilizzando LOOK “A” (particolare)

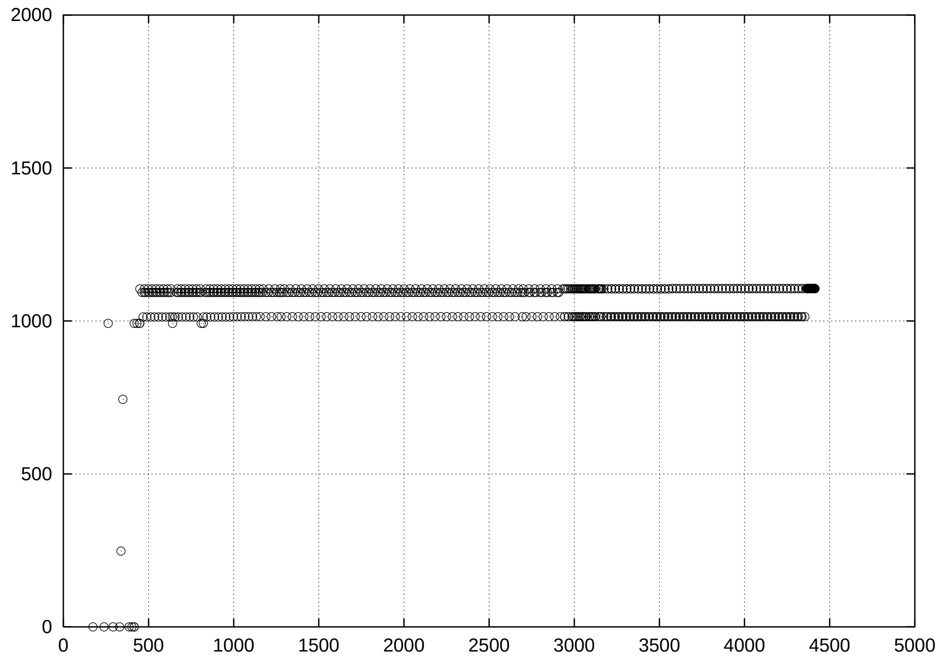

Figura 7.7: Test: richieste utilizzando LOOK “B”

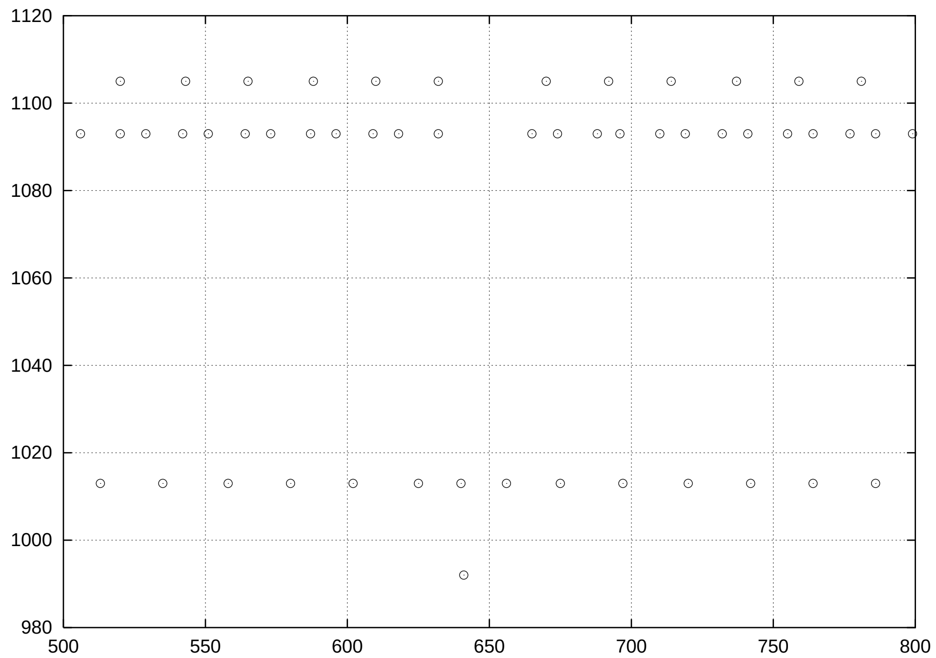

Figura 7.8: Test: richieste utilizzando LOOK “B” (particolare)

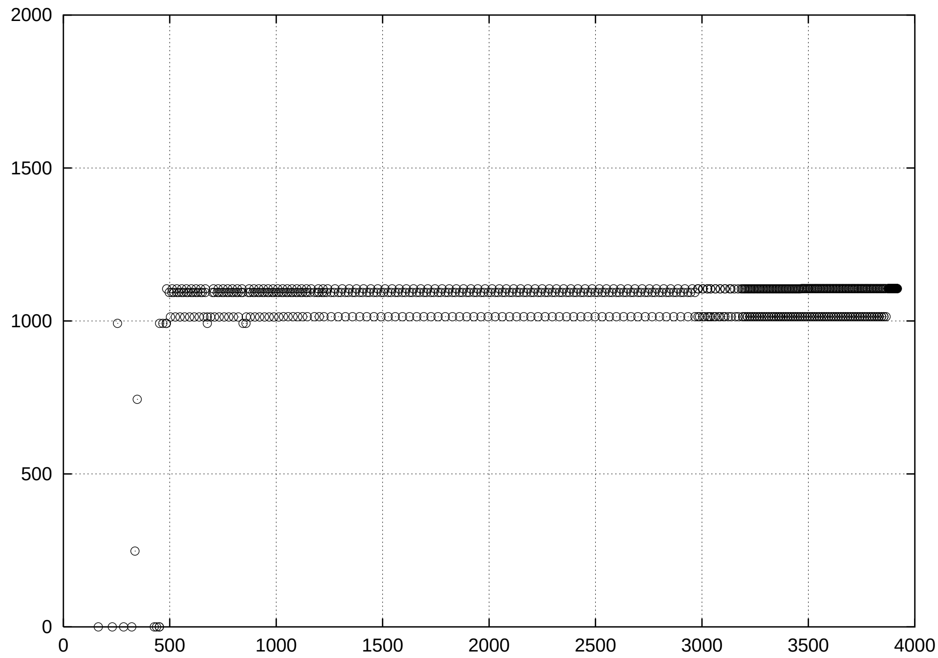

Figura 7.9: Test: richieste utilizzando LOOK “C”

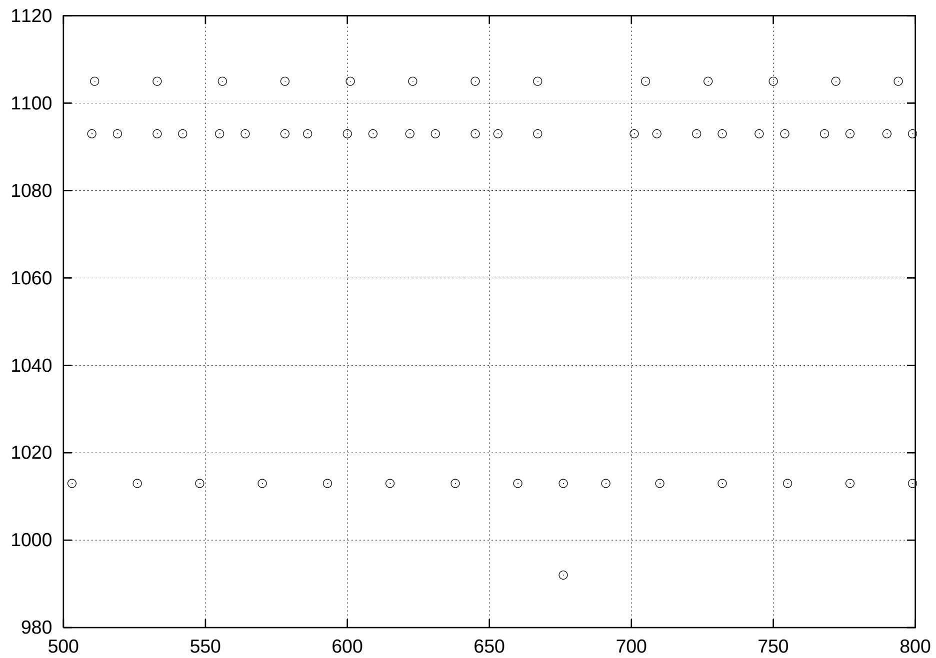

Figura 7.10: Test: richieste utilizzando LOOK “C” (particolare)

Indice

Elenco delle figure

Elenco delle tabelle

Bibliografia

[1] K. J. Anastasio Molano and R. Rajkumar. *Real-Time Filesystems – Guaranteeing Timing Constraints for Disk Accesses in RT-Mach*. Real-Time and Multimedia Laboratory - Departemente od Computer Scienze – Carnegie Mellon University, Pittsburgh, PA 15146.

[2] P. Ancilotti and M. Boari. *Principi e tecniche di programmazione concorrente*. Utet Libreria, 1987.

[3] P. R. Barham. *A Fresh Approach to File System Quality of Service*. University of Cambridge Computer Laboratory, Pembroke Street, Cambridge CB2 3QG ENGLAND.

[4] R. Black, P. Barham, A. Donnelly, and N. Stratford. *Protocol Implementation in a Vertical Structured Operating System*. Computer Laboratory, University of Cambridge, Cambridge, CB2 3QG, U.K.

[5] G. C. Buttazzo. *Sistemi in tempo reale*. Pitagore Editrice Bologna, 1995.

[6] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. *The Flux OSKit: A substrate for Kernel and Language Research*. University of Utah, University of California Davis, Massachusetts Institute of Technology.

[7] B. Ford, K. V. maren, J. Lepream, S. Clawson, B. Robinson, and J. Turner. *the Flux OS Toolkit: Reusable Components for OS Implementation*. Department of Computer Scienze, University of Utah, Salt Lake City, UT 84112.

[8] IEEE-SA Standard Board, editor. *IEEE Standard for Information technology–Standarized Application Environment Profile–POSIX Realtime Application Support (AEP)*. POSIX 1003.13. IEEE Computer Society, ieee std 1003.1 edition, March 1998.

[9] Intel©. *82371AB PCI ISA IDE Xcelerator (PIIX4)*.

[10] International Business Machine Corporation (IBM). *DOS 3.30 Technical Reference*, April 1987.

[11] International Standard ISO/IEC 9945-1, editor. *Information technology–Portable Operating System Interface (POSIX)–Part 1: System Application Program Interface (API) [C Language]*. POSIX 1003.1, 1003.1b, 1003.1c and 1003.1i. IEEE Computer Society, ieee std 1003.1 edition, 1996.

[12] *DOSref 3.3*. Ipertext on internet.

[13] ISO/IEC. *Programming languages–C*, iso/iec 9899 standard edition, 1990.

[14] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. *Application Performance and Flexibility on Exokernel Systems*. M.I.T. Laboratory for Computer Science, Cambridge, MA 02139, U.S.A.

[15] B. W. Kernighan and R. Pike. *Unix*. Zanichelli, 1989.

[16] B. W. Kernighan and D. Ritchie. *Linguaggio C*. Prenctice-Hall, 1989.

[17] Portable Application Standards Commitee, editor. *Draft Standard for Information Technology–Portable Operating System Interface (POSIX)–Part 1: System Application Program Interface (API)–Amendment x: Advanced Realtime Extensions [C Language]*. POSIX 1003.1j. IEEE Computer Society, ieee std 1003.1j edition, May 1999.

[18] Portable Application Standards Commitee, editor. *Information Technology–Portable Operating System Interface (POSIX)–Part 1: System Application Program Interface (API)–Amendment x: Additional Realtime Extensions [C Language]*. POSIX 1003.1d. IEEE Computer Society, ieee std 1003.1d edition, July 1999.

[19] P. J. Shenoy and H. M. Vin. *Cello: A Disk Scheduling Framework for Next Generation Operating Systems*. Distribuited Multimedia Computing Laboratory, Departement of Computer Scienze, University of Texas at Austin.

[20] H. Sprunt, L. Sha, and J.P.Lehoczky. Aperiodic task scheduling on hard real-time systems. *The Real-Time Systems Journal*, June 1989.

[21] T13 - Technical Committee of NCITS. *Information Techinology - AT Attachment with Packet Interface Extension (ATA/ATAPI-4)*, working draft - revision 18 edition, August 1998.

[22] VIA Technologies, Inc. *VT82C586B PIPC - PCI Integrated Peripheral Controller*, revision 1.0 edition, May 1987.