Università degli studi di Pisa

Facoltà di ingegneria

Corso di laurea in ingegneria informatica

Analisi e ottimizzazione di uno scheduler Round Robin per unità a disco

Autore:

Tommaso Caprai

Relatori:

Prof. Luigi Rizzo

Prof. Marco Avvenuti

Anno Accademico 2010-2011
Sessione di laurea del 03/03/2011

Ai miei genitori,

per avermi sempre sostenuto

e per aver assecondato tutte le scelte che ho fatto nella vita.

A Paola,

che col suo amore e la sua pazienza ha sempre creduto in me

e mi ha aiutato a raggiungere i traguardi più importanti.

Indice

Elenco delle illustrazioni

Introduzione

In ogni calcolatore elettronico è presente un sistema operativo, il quale ha tra i propri compiti, quello di gestire l'hardware e le varie risorse nel modo più efficiente possibile. Tra queste risorse vi sono le unità a disco, che essendo condivise tra molti utenti, rendono necessario l'utilizzo di un algoritmo di scheduling che permetta di scegliere in maniera appropriata la prossima richiesta da mandare in esecuzione tra quelle in attesa.

Lo scopo di questa tesi è quello di sviluppare e testare uno scheduler per unità a disco che dia garanzie di servizio per l'utente e che assicuri il più possibile un utilizzo equo di tale risorsa da parte dei vari processi che effettuano operazioni di I/O. Questo nasce anche dal fatto che un singolo processo, se non controllato in maniera opportuna, può acquisire l'uso del disco per effettuare le proprie attività per tempi che possono essere anche molto lunghi, negando perciò ad altri la possibilità di utilizzare lo stesso e riducendo così di fatto il parallelismo tra i vari processi o addirittura generando situazioni di stallo parziale (*Starvation*) del tutto indesiderate.

Nel realizzare uno scheduler non esiste un'unica strada da poter seguire, ma vi sono svariati algoritmi che portano al suddetto scopo, ciascuno dei quali presenta dei pregi e dei difetti, dipendenti da svariati fattori implementativi e dall'ambito operativo in cui ci si trova. In linea di massima comunque, due tra gli scopi principali per cui si introducono delle tecniche di scheduling sono la massimizzazione del throughput globale del disco e la riduzione del tempo medio di risposta alle varie richieste di I/O.

Questi due obiettivi vanno però in conflitto tra di loro, perché per aumentare il throughput si tende a servire sequenzialmente le varie richieste in base a dove esse vanno ad operare fisicamente sul disco e questo comportamento tende a ridurre l'equità (*fairness*) del servizio tra i vari processi poiché non prende in

considerazione i tempi di attesa. Al contrario, per ridurre il tempo di risposta a un qualsiasi processo, l'effettiva capacità di trasmissione dei dati da parte del disco non può che diminuire, poiché l'ordine per servire le varie richieste diventa quello temporale e c'è bisogno di fare un maggior numero di *seek*, ovvero di movimenti delle testine del disco. Tutto sta quindi, sebbene vadano analizzati anche altri fattori, nel trovare il giusto equilibrio tra questi due parametri fondamentali. Molti sistemi operativi privilegiano solo il throughput, utilizzando algoritmi di scheduling come ad esempio *Elevator* (o algoritmo dell'ascensore), che consiste semplicemente nello scandire il disco in un'unica direzione servendo man mano le richieste che ci si trova di fronte, andando così a gravare in molti casi su quelli che sono i tempi di attesa. Per processi però che fanno ad esempio rendering video, si hanno vincoli temporali molto stringenti e quindi si rendono necessari algoritmi differenti.

Per le nostre prove abbiamo lavorato in ambiente FreeBSD, un sistema operativo avanzato derivato da BSD, la versione di UNIX sviluppata dall'università della California (Berkeley). Anch'esso utilizza in maniera nativa Elevator (nella sua variante C-LOOK) come algoritmo di scheduling disco, ma dispone anche di un sistema sperimentale che andava testato al fine di poter dimostrare la sua stabilità e il fatto che potesse garantire all'utente un servizio fruibile e sicuro dal punto di vista operativo.

Questo sistema è basato su un framework modulare per l'accesso al disco chiamato GEOM, che consente la manipolazione delle richieste attraverso una serie di moduli che sono interconnessi tra loro come i nodi di un grafo. Un modulo può ad esempio effettuare il riordinamento delle richieste di I/O e quindi svolgere funzioni di scheduling. In particolare il modulo *geom_sched* offre numerosi vantaggi tra cui la possibilità di implementare più politiche di scheduling differenti, utilizzare un'unica istanza per tutti i dispositivi a disco (poiché non richiede la specifica dettagliata delle caratteristiche del singolo disco), e la possibilità di abilitare o disabilitare lo scheduler a seconda delle necessità che si hanno al momento.

Nel nostro caso specifico abbiamo analizzato uno scheduler GEOM che implementa *Round-Robin*, un algoritmo preemptive che esegue i vari processi in base all'ordine di arrivo delle rispettive richieste di I/O, ma che effettua anche preemption sul processo attivo, ponendolo alla fine della coda dei processi in attesa e passando l'esecuzione a quello successivo, nel caso in cui essa superi un determinato budget temporale o dati. Lo scheduler in questione è stato realizzato in lavori precedenti ma non è mai stato completamente testato, per cui il mio compito è stato quello di verificarne il corretto funzionamento e le prestazioni.
Effettuando dei test operativi, è venuto a galla il fatto che, in determinate situazioni, si verificava il mancato rispetto delle *deadline*, cioè il processo in esecuzione, al termine del budget temporale, non subiva preemption ma continuava in maniera imprecisata la sua esecuzione. E' stata quindi necessaria una approfondita ricerca all'interno del codice sorgente dello scheduler delle cause per cui veniva a verificarsi questo funzionamento indesiderato, che ha portato all'individuazione di una particolare situazione in cui, a causa di una erronea verifica e aggiornamento dei parametri, il settaggio dell'istante di fine budget non avveniva correttamente. Analizzando i vari dati raccolti riguardo questa anomalia di funzionamento, è stato possibile risolvere il problema mediante la modifica di una serie di controlli effettuati dallo scheduler sulla coda di richieste attiva.
Per quanto riguarda i test prestazionali effettuati, nonostante non vi sia alcun guadagno in caso di utilizzo da parte di un singolo utente, in caso di utilizzo multiutente si sono raggiunte con lo scheduler attivo delle velocità del disco in lettura raddoppiate o addirittura quasi triplicate rispetto a prove equivalenti con scheduler non attivo, e in scrittura, nonostante un calo delle prestazioni intorno al 50%, si è notato un notevole aumento per quanto riguarda l'equità di servizio.
Questa analisi approfondita ha permesso quindi, di rendere il funzionamento dello scheduler conforme alle aspettative, e di poter garantire all'utente che andrà ad utilizzarlo, un software stabile e in grado di gestire in maniera equa la risorsa disco, il tutto con un sensibile miglioramento rispetto al semplice scheduler Elevator offerto da FreeBSD.

Capitolo 1.
Algoritmi di scheduling

L’evoluzione tecnologica recente ha portato a un notevole sviluppo di dispositivi elettronici come microprocessori e memorie, che gli hanno permesso di raggiungere velocità estremamente elevate. Non altrettanto si può dire invece di dispositivi elettromeccanici come i dischi rigidi i quali, anche per fattori fisici legati proprio alla natura del dispositivo, come la necessità di far ruotare i dischi o spostare le testine, non hanno subito una simile evoluzione.
Inoltre all'interno di un sistema, diversi processi possono generare contemporaneamente richieste di interazione col disco (siano esse in ingresso o in uscita), cosa che avviene più velocemente della sua capacità di servirle. A seguito di questo, tali processi tendono a subire un ritardo di servizio che se non controllato può essere anche piuttosto elevato, e per ridurre il problema è doveroso operare ottimizzando le richieste implementando degli algoritmi di scheduling. Per fare un facile esempio basti pensare al caso di un lettore video il quale, se non vengono rispettati stringenti vincoli temporali, non sarà in grado di fornire all'utente un servizio fruibile, offrendo una visione 'a scatti' del filmato che sta visionando.

1.1 Cenni sulla storia dei dischi

Il primo prototipo di disco fisso (in inglese *Hard Disk Drive* o abbreviato *HDD*) fu inventato da ingegneri dell'IBM nel 1956: era costituito da 50 piatti da 24" con una capacità di memorizzazione complessiva di circa 5MB, ed originariamente il nome era *Fixed Disk,* giustificato dal fatto che aveva un peso e una grandezza (come un armadio di medie dimensioni) che non ne permettevano

una facile mobilità. Nel 1963 sempre IBM introdusse il meccanismo di sollevamento delle testine mediante il cuscinetto d'aria sviluppato dalla rotazione stessa dei dischi, mentre all'inizio degli anni 70 nacque la denominazione Hard Disk per contrapporre questa tecnologia a quella degli appena nati *Floppy Disk*. Il modello che andò comunque a segnare il predominio degli Hard Disk nel campo della memorizzazione digitale nacque nel 1973, denominato "*3340 Winchester*" per analogia con un noto fucile, ed era composto da due piatti in grado di memorizzare 30MB ciascuno. Il primo disco per PC destinato all'uso domestico e predecessore di quelli moderni, è stato invece nel 1980 l'*ST-506* creato da Seagate Technologies, ed era costituito da dischi da 5,25", una capacità complessiva di 5MB e da un motore passo passo per il movimento delle testine sulla superficie.

1.2 Struttura e funzionamento di un disco

Le unità a disco presenti negli attuali calcolatori, sono normalmente gestite come array monodimensionali di blocchi logici che costituiscono le unità elementari di trasferimento. C'è anche una metodologia ben definita per l'organizzazione dei dati al suo interno, ed iniziamo una sua breve analisi dicendo innanzitutto che quando si parla di disco si parla di un'unità che ha in sé non uno ma un certo numero di dischi uguali tra loro e sistemati parallelamente uno all'altro. Ciascun disco ha poi entrambe le superfici utilizzabili, ciascuna chiamata *piatto* e su ciascun piatto è presente una testina di lettura/scrittura situata a pochi micron dalla superficie del piatto stesso che si muove lateralmente su di essa: le varie testine si muovono in blocco, essendo dipendenti l'una dall'altra. Ogni piatto si compone a sua volta di diversi anelli concentrici numerati chiamati *tracce*, e si parla di *cilindro* per intendere l'insieme delle tracce equidistanti dal centro (quindi identificate dallo stesso numero) che si trovano però sui vari piatti. Per finire ogni traccia è suddivisa in *settori* che rappresentano la quantità d'informazione minima memorizzabile (che può andare solitamente da 32 byte a 4Kb), a seconda di come viene formattato il disco, ovvero in base al suo *filesystem*. Il filesystem determina quindi il tipo di organizzazione dei dati,

fornendo una corrispondenza tra file ed uno specifico tipo di coordinate, ad esempio la sequenza piatto-traccia-settore, che individuano fisicamente il file sul disco. La dimensione scelta per il settore andrà anche ad influire sulle prestazioni, poiché se i dati memorizzati sono più grandi del singolo settore saranno necessarie un maggior numero di operazioni di I/O, mentre se sono più piccoli l'accesso sarà più rapido a discapito di un maggior spreco di spazio sul disco (quello che resta vuoto tra il dato utile e la fine del settore). Per questo motivo e anche considerando le capacità elevate raggiunte dagli attuali hard disk, per semplificare le operazioni i settori vengono anche raggruppati tra loro in *cluster*, che sono semplicemente una concatenazione di settori contigui. Per finire, si parla di *blocco* per intendere l'insieme dei settori posti nella stessa posizione su ogni piatto.

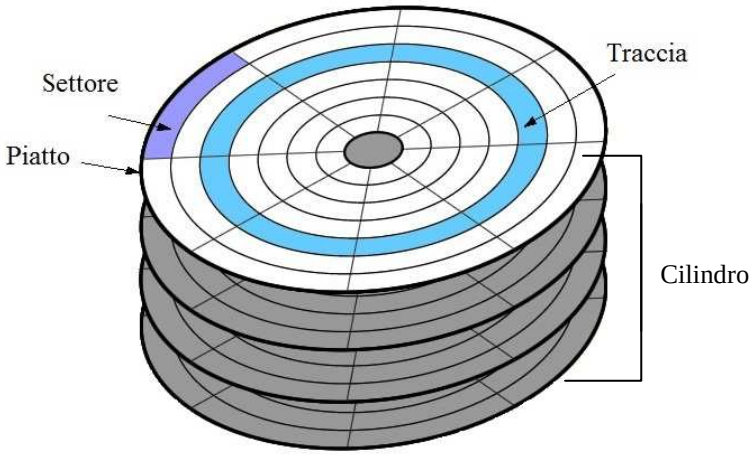


Fig 1.1 - Struttura di un disco

L'array monodimensionale di blocchi logici è quindi realizzato sequenzialmente nei settori del nostro disco e solitamente si parte dalla parte più esterna del disco, intendendo come settore 0 il primo settore della traccia più esterna.

1.3 Obiettivi dello scheduling

In base agli obiettivi e al tipo di interazioni che si debbono avere con il disco, è opportuno scegliere un algoritmo di scheduling piuttosto che un altro. Un fattore

molto importante da prendere in considerazione è l'equità del servizio offerto o *fairness*, che indica la capacità di servire una qualsiasi richiesta senza che si verifichino situazioni di stallo parziale (*starvation*) in cui cioè non si riesce ad ottenere l'accesso al disco, attendendo per tempi indefinitamente lunghi. In generale comunque ciò che si cerca sempre di ottimizzare sono i seguenti parametri:

- ***Throughput***, che rappresenta il numero di richieste servite nell'unità di tempo;
- ***Tempo medio di risposta***, che rappresenta quanto deve attendere mediamente una richiesta prima di essere servita;
- ***Varianza del tempo di risposta***, che indica la probabilità che le varie richieste siano servite con un tempo di attesa simile (è una sorta di misura dell'equità e della prevedibilità).

Inoltre, si cerca anche di memorizzare le varie richieste di I/O nella maniera più semplice possibile, in modo che lo scheduler abbia il minimo impatto sulle prestazioni di sistema.

Un modo comune per incrementare il throughput globale è quello di servire sequenzialmente le richieste, analizzando dove esse devono andare ad operare fisicamente sul disco: servendo infatti richieste non contigue, si va incontro a ritardi rotazionali e penalizzazioni dovute alle operazioni di *seek* eseguite (con seek si intende lo spostamento delle testine del disco al fine di individuare il punto su cui si deve andare a lavorare). Seguendo solo questa strada però, è possibile andare incontro ad una forte penalizzazione dei tempi di risposta nonché dell'equità di servizio.

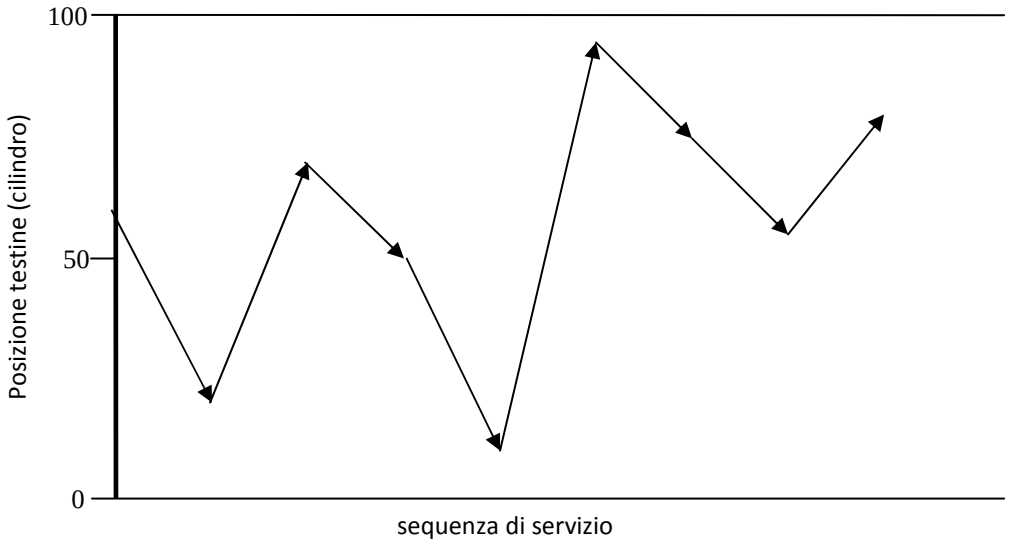
Per ridurre invece il tempo medio di risposta è sufficiente servire le richieste in base al loro ordine di arrivo. In questo caso però è il throughput che può risentirne enormemente, in quanto per andare a posizionarci ogni volta su richieste che possono essere anche molto distanti tra loro sul disco, sorge la necessità di effettuare molte seek, riducendo quindi di fatto quelle che sono le prestazioni del disco rispetto alle sue potenzialità.

Questi due parametri vanno quindi in conflitto tra loro, poiché fondamentalmente sono direttamente proporzionali, ma in teoria ciò che si vorrebbe sarebbe massimizzare il throughput e minimizzare i tempi medi di risposta, rendendo così necessario il raggiungimento di un compromesso che deve essere quello che più si addice ai nostri scopi. La possibile via da seguire non è quindi per forza una, ma è preferibile decidere a seconda delle condizioni operative in cui ci troveremo a lavorare, che possono dipendere dal carico di lavoro, o da altre necessità che potremmo dover soddisfare.

1.4 FCFS

Uno dei più semplici algoritmi di scheduling, ovvero l'*FCFS* (First Come First Served), provvede a servire le richieste nell'ordine di arrivo, indipendentemente dalla loro posizione sul disco, immagazzinandole semplicemente in una coda FIFO. Questo permette sì di garantire un servizio equo per ogni processo, ma quando il carico si fa più pesante, ovvero si ha un grosso numero di richieste di I/O, a causa di un forte aumento delle operazioni di seek, si riduce notevolmente il throughput causando un forte calo delle prestazioni.

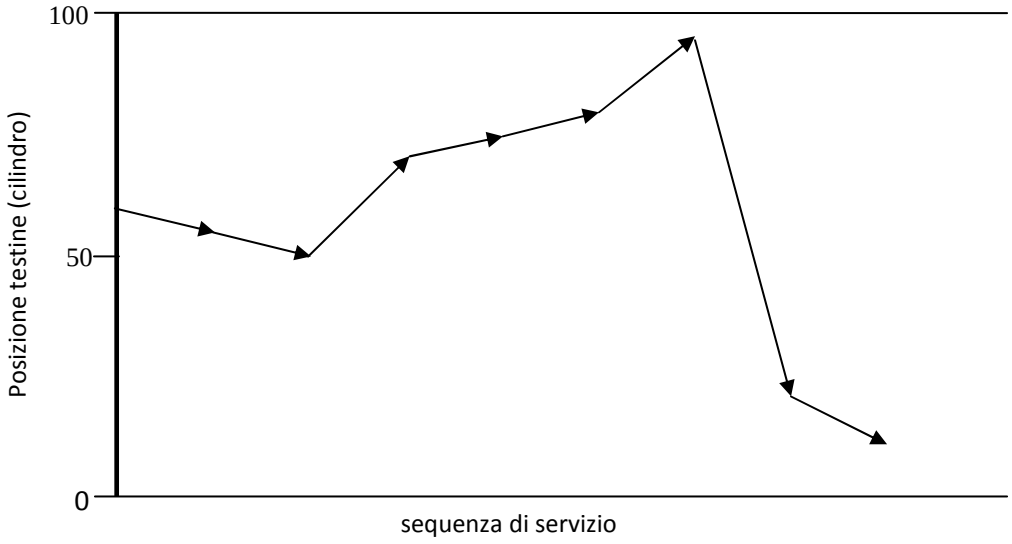
Supponiamo che il nostro disco sia costituito da 100 cilindri, che arrivino delle richieste di I/O nell'ordine 22, 68, 50, 10, 97, 73, 54, 79, e che le testine siano posizionate inizialmente sul cilindro 60. Otterremo il seguente andamento:



1.2 Movimenti testine con FCFS

1.5 SSTF

L'SSTF (Shortest Seek Time First), serve le richieste in base al relativo tempo di seek, ovvero la decisione viene presa in base alla vicinanza rispetto all'attuale cilindro del disco su cui stiamo lavorando. Il throughput ottiene in questo modo un notevole incremento e, pur dipendentemente dal carico, anche il tempo di risposta si mantiene su dei buoni valori. Quella che ne risente maggiormente però è in questo caso la varianza del tempo di risposta dato che la scelta in base alla distanza crea una discriminazione tra le zone esterne del disco e quelle interne che saranno più favorite. Questo algoritmo pertanto non garantisce un servizio equo, e nel caso particolare numerose richieste si concentrino nelle tracce interne del disco, si può verificare il fenomeno della starvation relativo a quelle poche richieste che andrebbero a operare nelle zone più esterne che potrebbero non essere mai servite. Supponendo che arrivi una sequenza di richieste analoga a quella mostrata nel caso precedente, avremo il seguente andamento delle testine:



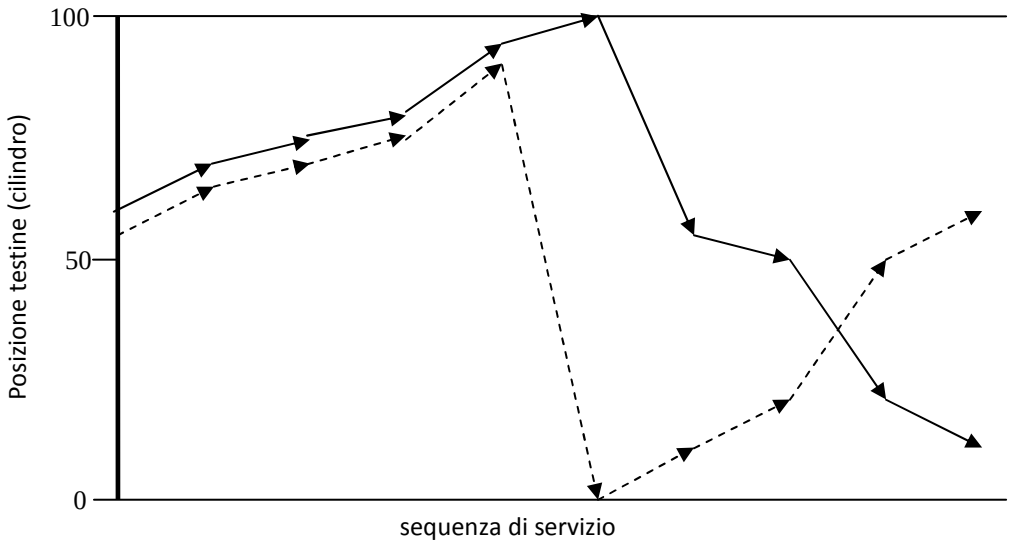
1.3 Movimenti testine con SSTF

1.6 SCAN e C-SCAN

L'algoritmo di scheduling SCAN, noto anche con il nome di *elevator* (ascensore) per la particolare sequenza di movimento delle testine che genera, è una variante di SSTF che scandisce il disco in una sola direzione alla volta, cioè sceglie la

successiva richiesta da evadere come quella che necessita del seek minore in una determinata direzione preferenziale. In pratica si inizia dal cilindro più piccolo del disco servendo le varie richieste fino ad arrivare al più grande, e giunti a quel punto si cambia direzione. Così facendo si mantiene un buon throughput come nel caso SSTF sfruttando la località delle richieste, e si riduce, anche se solo entro certi limiti la varianza eliminando il fenomeno della starvation. L'unico inconveniente che rimane è che una richiesta, arrivata di poco in ritardo, può attendere anche molto tempo prima che si torni a servirla.

Circular SCAN (C-SCAN) è invece una variante di SCAN dove in cui il disco viene scandito sempre nella stessa direzione, partendo dal cilindro più piccolo al più grande per poi ripartire dal più piccolo, cercando di mantenere i vantaggi di SCAN, ovvero mantenere un buon throughput in andata sfruttando la località delle richieste nonché fornire tempi di attesa più uniformi anche per richieste arrivate in ritardo, e diminuire l'attesa per le richieste più vecchie, riducendo di conseguenza la varianza. L'inconveniente in questo caso è che non c'è riduzione del movimento delle testine. Il loro comportamento all'arrivo della solita sequenza di richieste è il seguente:



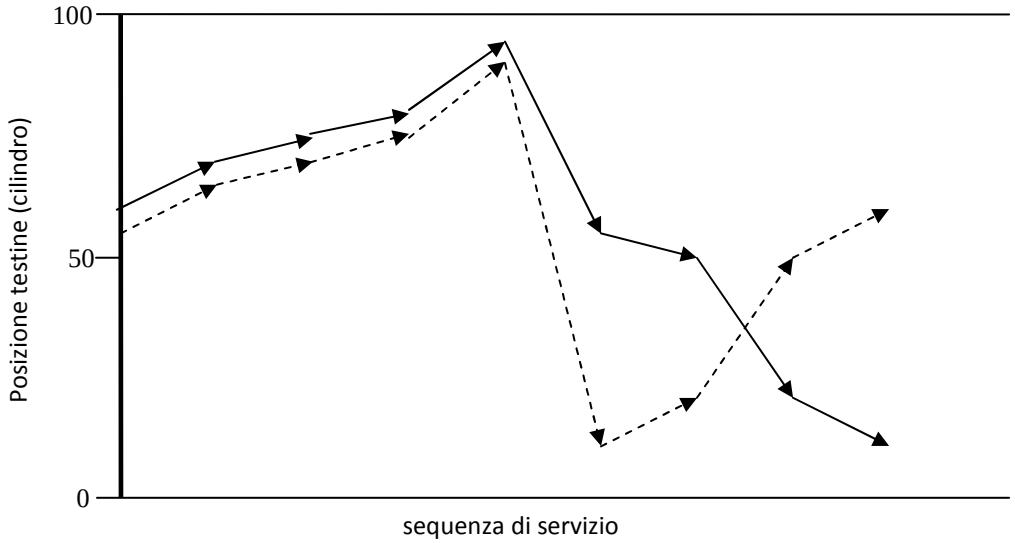
1.4 Movimenti testine con SCAN e C-SCAN (tratteggiato)

1.7 LOOK e C-LOOK

Gli algoritmi LOOK e C-LOOK sono varianti ottimizzate rispettivamente di SCAN e C-SCAN che sfruttano la conoscenza della posizione su disco delle varie richieste.

Con LOOK ci si comporta esattamente come con SCAN a differenza del fatto che ogni volta che si scandisce il disco in una determinata direzione, prima di andare avanti, si verifica che effettivamente ci sia una richiesta da servire in quel senso, altrimenti si torna indietro. In questo caso si sente ancora di più l'analogia con l'andamento di un ascensore quindi anche questo algoritmo viene chiamato Elevator. A differenza di SCAN si eliminano in questo modo tutte le seek superflue.

Allo stesso modo C-LOOK effettua una scansione del disco in maniera circolare, partendo dal cilindro più basso corrispondente ad una richiesta fino al più alto con richieste pendenti, per poi ripartire con lo stesso criterio. Le caratteristiche prestazionali sono le stesse di C-SCAN, ma con una riduzione dei tempi di seek. C-LOOK è caratterizzato da una minor varianza rispetto a LOOK, e un throughput piuttosto alto anche se leggermente inferiore se paragonato con quello di LOOK. In questo caso il movimento delle testine col solito pattern di richieste sarà:



1.5 Movimenti testine con LOOK e C-LOOK (tratteggiato)

1.8 Round Robin

Passiamo ora ad analizzare il comportamento generale dell'algoritmo di scheduling Round Robin, su cui è incentrata la tesi, che ha un funzionamento del tutto diverso dai precedenti. Esso è realizzato su due livelli, ovvero come nel caso FCFS, da una coda FIFO in cui anziché le richieste, vengono inseriti i processi che hanno necessità di effettuare operazioni di I/O sul disco. In più, ad ogni processo, corrisponde una coda in cui vengono memorizzate le relative richieste. Viene inoltre stabilito un tempo massimo di esecuzione, detto *budget* o *quantum* temporale, valido per ciascun elemento della coda FIFO, allo scadere del quale viene tolto a quel processo il diritto di eseguire operazioni sul disco, reinserendo lo stesso elemento al termine della coda. Se invece il budget temporale non viene consumato completamente si azzera il budget e si passa all'elemento successivo presente in coda. In questo modo si garantisce un servizio equo per tutti i processi, eliminando completamente il fenomeno della starvation. In questo caso il throughput resta comunque elevato, poiché si continua a sfruttare la località delle richieste, ed inoltre si rende più uniforme anche la varianza del tempo di risposta, dato che comunque dopo un determinato numero di budget l'esecuzione tornerà nelle mani di chi è in attesa.

1.9 Distribuzione delle richieste

Ci sono però anche delle precisazioni da fare che cambiano non di poco l'effettivo andamento delle prestazioni. Infatti tutte le considerazioni fatte sui vari algoritmi sono valide prevedendo una distribuzione uniforme delle richieste, ovvero che la probabilità che si vada ad operare su di un cilindro del disco piuttosto che su un altro sia la stessa in ogni caso. In realtà però questa situazione non è poi così realistica dato che, ad esempio, i file dello stesso utente vengono solitamente impacchettati in zone contigue del disco. Questo può migliorare di molto le prestazioni dei vari scheduler sfruttando il fatto che le seek da effettuare si riducono enormemente. Inoltre, un'altra tecnica adottata per ridurre il movimento delle testine consiste nella fusione delle richieste (*merging*) in cui ad

esempio due richieste per settori del disco adiacenti vengono unite in un'unica richiesta per due settori contigui. Sfruttando quindi questi due aspetti si riescono a ridurre molto i tempi di attesa e le operazioni di seek garantendo un throughput elevato e facilitando notevolmente il compito degli scheduler, indipendentemente dall'algoritmo che implementano.

Capitolo 2.

FreeBSD e GEOM

2.1 Introduzione a FreeBSD

FreeBSD è un sistema operativo avanzato per architetture compatibili x86, amd64, Alpha/AXP, IA-64 e UltraSPARC: esso è derivato da BSD (Berkeley Software Distribution), la versione Unix sviluppata dall'università della California, è disponibile gratuitamente e viene fornito con il codice sorgente completo.

Il progetto ha origine all'inizio del 1993 come evoluzione dell'*Unofficial 386BSD Patch Kit* e la prima release ufficiale è datata dicembre dello stesso anno, poco prima che gli sviluppi di una lunga controversia legale tra l'università di Berkeley e *Novell* (un'azienda produttrice di software proprietaria di alcune licenze Unix) vadano ad incidere direttamente anche su FreeBSD, che era inizialmente basato sul fork 386BSD contenente alcune parti originariamente derivate da Unix. Nonostante Novell accettò un accordo per chiudere il contenzioso, FreeBSD doveva comunque eliminare le porzioni di codice prese dal sistema proprietario. Il team accettò la sfida e nonostante l'impatto che tali modifiche avevano sull'intero sistema a novembre del 1994 rilasciò *FreeBSD 2.0*, ancora piuttosto instabile ma libero da codici proprietari.

Oggi il sistema è utilizzato principalmente in ambito server, poiché consente l'hosting di numerosi siti web (centinaia di migliaia) sulla stessa macchina, grazie alla stabilità e scalabilità del sottosistema di networking. Vista la sua specificità non stupisce la grande attenzione posta alle problematiche di sicurezza con l'integrazione di ben 3 firewall a partire dalla versione 6.0.

Nonostante la sua fama di sistema operativo per server, FreeBSD può essere comodamente utilizzato anche come sistema per desktop, grazie all'efficiente sistema di gestione dei pacchetti software (*Ports*), che permette di installare agevolmente centinaia di applicazioni: browser internet, ambienti grafici integrati, suite per ufficio, lettori multimediali e molto altro. I ports liberano l'utente dal problema delle dipendenze, visto che il sistema che li gestisce scarica i sorgenti più aggiornati del programma di cui c'è bisogno e non il suo eseguibile binario, in combinazione con i sorgenti aggiornati di tutti i programmi dai quali esso dipende, completando l'operazione di installazione con la loro ricompilazione ex novo sulla macchina in esame. Questo permette anche di ottimizzare il funzionamento del programma per l'uso su un determinato sistema, in base anche alle sue caratteristiche hardware.

2.2 Il Kernel di FreeBSD

La versione di FreeBSD utilizzata per questa tesi è la RELEASE 8.0, che all'inizio dello studio rappresentava la versione ufficiale disponibile più aggiornata (ad oggi è stata soppiantata dalla RELEASE 8.1).

Tradizionalmente il nucleo del sistema operativo di FreeBSD, cioè il *kernel*, è sempre stato di tipo *monolitico*, ovvero un programma di grosse dimensioni in grado di supportare una determinata lista di periferiche, che necessitava di una ricompilazione e del riavvio del sistema nel caso si avesse voluto un comportamento diverso da quello predefinito. Attualmente invece FreeBSD sta adottando un modello in cui gran parte delle funzionalità sono contenute in moduli che possono essere caricati e scaricati a seconda delle necessità, garantendo così dei vantaggi che permettono di aggiungere funzionalità al kernel che non erano disponibili o necessarie al momento della sua compilazione, così come di adattarsi ad un nuovo hardware che si è appena reso disponibile. Un kernel di questo tipo viene chiamato *modulare*. Nonostante questo però, in determinati casi è ancora necessario effettuare delle compilazioni statiche del kernel, o perché la funzionalità da aggiungere è talmente legata al nucleo che non può essere resa caricabile dinamicamente, o perché più semplicemente nessuno

ha ancora realizzato un modulo dinamico per essa. Il processo di ricompilazione del kernel è comunque un'operazione quasi obbligatoria per un qualsiasi utente di FreeBSD, poiché porta numerosi benefici al sistema che da generico e quindi in grado di supportare la maggior parte degli hardware, diventa specifico per la macchina su cui viene utilizzato, garantendo tempi di avvio ridotti, un minor uso della memoria e il supporto ad hardware addizionale.

2.3 Considerazioni generali sulla temporizzazione

Al fine di capire più avanti quelle che sarà la caratterizzazione temporale dello scheduler e le sue configurazioni, è necessario fare una precisazione riguardo la temporizzazione dei processi all'interno del sistema. I sistemi *unix-like*, come anche FreeBSD, hanno sempre mantenuto due distinti tipi di dati per la misure dei tempi all'interno del sistema: essi sono rispettivamente chiamati *calendar time* e *process time*.

Il calendar time, o tempo di calendario, è il numero di secondi trascorsi dalla mezzanotte del primo gennaio 1970, in tempo universale coordinato (o UTC), data che viene usualmente indicata con 00:00:00 Jan, 1 1970 (UTC) e chiamata *Epoch*. Questo tempo viene anche chiamato anche GMT (Greenwich Mean Time) dato che l'UTC corrisponde all'ora locale di Greenwich, ed è il tempo su cui viene mantenuto l'orologio del kernel e viene usato ad esempio per indicare le date di modifica dei file o quelle di avvio dei processi. Per memorizzare questo tempo è stato riservato il tipo primitivo *time_t*.

Il process time invece, viene misurato in *clock tick*. Un tempo questo corrispondeva al numero di interruzioni effettuate dal timer di sistema, ma adesso lo standard POSIX[1] richiede che esso sia pari al valore della costante *clocks_per_sec*, che deve essere definita come $1*10^6$, qualunque sia la risoluzione reale dell'orologio di sistema e la frequenza delle interruzioni del timer. Il tipo di dato primitivo usato per questo tempo è *clock_t*, che ha quindi

(1) POSIX : è uno standard il cui compito è quello di definire alcuni concetti base che vanno seguiti durante la realizzazione del sistema operativo, e deriva da un progetto finalizzato alla standardizzazione dei software sviluppati per le diverse varianti dei sistemi operativi UNIX. [Per info http://standards.ieee.org/]

una risoluzione del microsecondo. Il numero di tick al secondo può essere ricavato anche attraverso sysconf(). Il process time viene di solito espresso in secondi, ma provvede una precisione ovviamente superiore al calendar time (che è mantenuto dal sistema con una granularità di un secondo) e viene usato per tenere conto dei tempi di esecuzione dei processi.
Eseguendo nei sistemi utilizzati per l'analisi dello scheduler il comando "sysctl kern.clockrate" si ottengono valori di HZ = 1000 e tick=1000, che significano rispettivamente che i sistemi generano il segnale di clock alla frequenza di 1MHz, che corrisponde a 1 tick ogni millisecondo.

2.4 Cos'è GEOM

GEOM è un framework modulare per la manipolazione delle richieste rivolte al disco. Esso fornisce una infrastruttura in cui le varie classi controllate, possono effettuare modifiche sulle richieste di I/O del disco che vanno dal kernel ai driver di periferica e viceversa. Queste trasformazioni possono riguardare diversi aspetti, dal semplice spostamento geometrico dei dati verso un disco partizionato, alla gestione di algoritmi RAID[2], fino ad una completa protezione crittografica dei dati memorizzati. C'è da dire che, paragonato alle normali tecniche di gestione dei dischi, GEOM è estremamente differente, poiché oltre a non aver bisogno di informazioni rigorose sulla topologia del sistema e dei vari dispositivi, permette scrivere una nuova classe di trasformazione senza dover apportare modifiche a strutture che sono esterne a GEOM stesso, rendendola completamente trasparente al sistema.
La caratteristica principale di GEOM è data dal fatto che i vari moduli sono interconnessi tra loro come nodi di un grafo. Questa proprietà porta ad avere una struttura molto particolare su vari livelli, in cui ciascun nodo del grafo è costituito da una o più porte *provider* e una o più porte *consumer*, come mostrato in figura 2.1. Le porte consumer possono essere connesse ad un solo provider, mentre al contrario quelle provider possono avere più consumer collegati ad essa.

(2) RAID : Acronimo di "Redundant Array of Independet Disks" cioè insieme ridondante di dischi indipendenti, che è un sistema di archiviazione che utilizza un insieme di dischi rigidi per condividere o replicare le informazioni.

Le porte provider rappresentano il punto di accesso ai servizi offerti da GEOM, e quelle più a monte sono direttamente collegate con i sorgenti (ad esempio il filesystem). Le richieste di I/O al disco, ciascuna rappresentata da una struttura chiamata *bio* e definita in un file di libreria di FreeBSD che è bio.h, arrivano al disco stesso seguendo il percorso opportuno, entrando nei vari nodi attraverso le porte provider, ed uscendo dalle porte consumer. All'interno di ogni nodo può essere effettuata una qualsiasi tipo di manipolazione sui dati tra quelle definite dai vari moduli GEOM. Cosa molto importante, dato che a ciascun provider possono essere collegati uno o più consumer, essi possono fornire delle strutture bio che hanno subito tipi di manipolazioni diversi.
A seconda delle necessità le richieste vengono poi passate a valle finché non si incontra l'ultimo nodo del grafo che dialoga direttamente coi driver della periferica a cui vengono fornite le richieste opportunamente manipolate. All'interno del grafo non è permesso alcun tipo di ciclo.

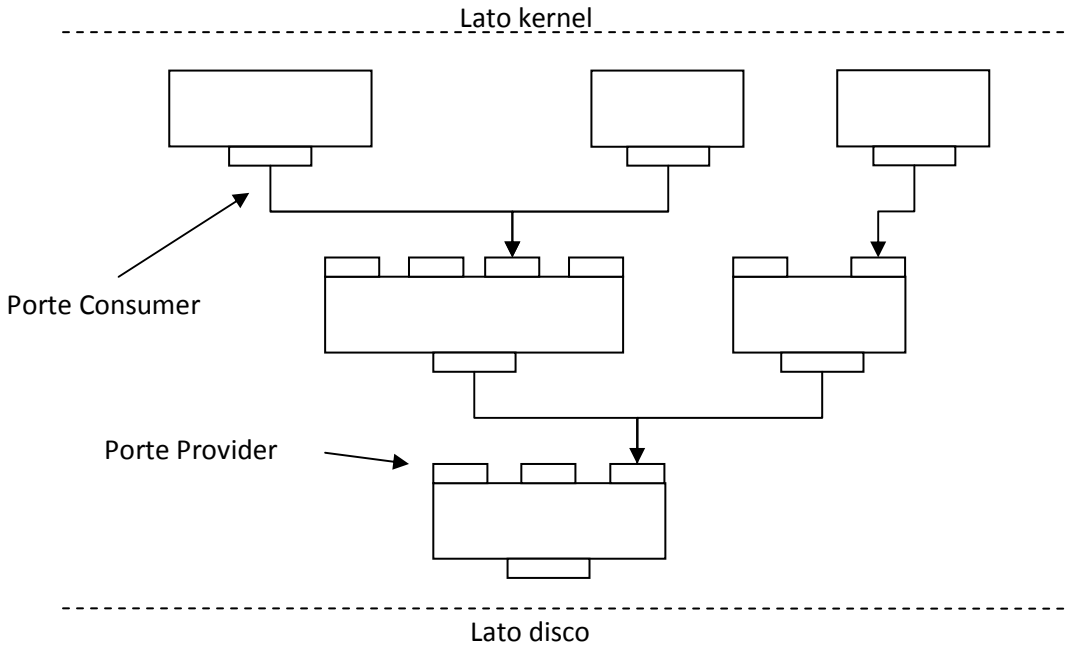


Fig. 2.1 - Struttura a grafo dei moduli GEOM

GEOM è piuttosto orientato agli oggetti, quindi la sua terminologia rispecchia quella della programmazione orientata agli oggetti. Più in generale, una classe di nome *class* è infatti rappresentata dalla struttura dati g_class (dove g sta per

GEOM) e implementa una particolare tipo di trasformazione. Ciascun provider, come detto, viene presentato a chi usufruisce del servizio, ed è rappresentato da una sorta di disco logico che si aggiunge nella cartella /dev del disco, caratterizzato da un nome, una dimensione totale e dalla dimensione dei suoi settori.

GEOM permette anche speciali operazioni topologiche che posso essere effettuate sul grafo. Tra queste, un esempio interessante che analizzeremo in seguito sono le INSERT/DELETE, che permettono a un nuovo nodo GEOM di essere dinamicamente inserito/rimosso tra una porta consumer e una provider già esistenti.

Capitolo 3.

Analisi dello scheduler GEOM

3.1 Obiettivi legati all'uso di geom_sched

Il fatto di effettuare la schedulazione del disco sfruttando un modulo GEOM, introduce già tutti i vantaggi portati da GEOM stesso e analizzati nel capitolo precedente, poiché con l'utilizzo di un un'unica istanza valida per tutte le unità a disco, si ha la possibilità di implementare differenti algoritmi di scheduling che è possibile abilitare o disabilitare a piacimento a seconda delle situazioni operative. Il modulo geom_sched offre un particolare tipo di trasformazione che si occupa di effettuare un'ottimizzazione delle richieste di accesso al disco. Esso va a costituire un nuovo nodo come quello mostrato nella figura 3.1, che si inserisce all'interno del grafo che costituisce la struttura di GEOM.

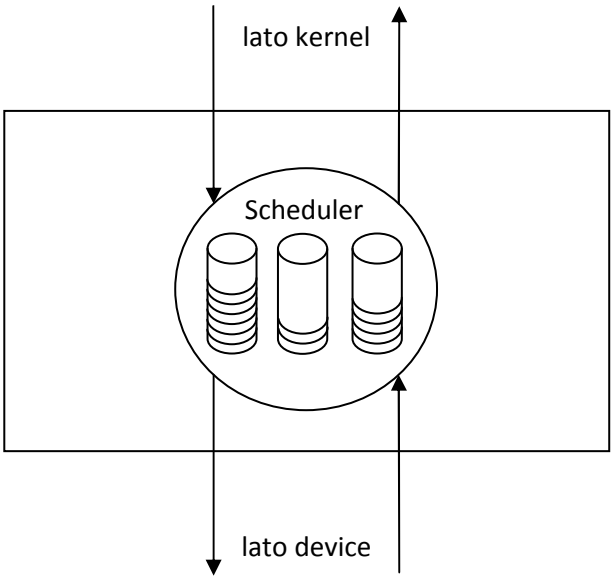


Fig. 3.1 - Nodo schematizzato di geom_sched

Esso riceve dall'alto delle richieste di accesso al disco sotto forma di 'bio' che provvede ad immagazzinare in code ordinate, che invia successivamente verso il disco scegliendo quelle previste dall'algoritmo in uso. Quando riceve poi dal disco una segnalazione di richiesta completata (tramite una bio_done) provvede a trasmetterla in direzione opposta, per darne comunicazione al consumer interessato.

Il modulo geom_sched presenta numerosi vantaggi rispetto allo scheduler standard implementato da FreeBSD. Innanzitutto è da sottolineare la sua facilità d'uso anche per un utente con poca esperienza, poiché per utilizzarlo è sufficiente caricare i moduli del kernel necessari al suo funzionamento, e inoltre la sua ridotta invadenza, data dal fatto che una volta attivato non necessiti di alcun tipo di intervento da parte dell'utente. L'aspetto fondamentale legato al suo uso è comunque il fatto che dovrebbe fornire dei benefici a livello prestazionale non indifferenti.

Questa cosa però, insieme al suo corretto funzionamento, va ancora comprovata definitivamente dato che sia il modulo geom_sched sia gli algoritmi di scheduling che implementa, sono stati realizzati in lavori precedenti e mai testati a fondo.

Il funzionamento di geom_sched è comunque caratterizzato da un ridotto incremento dell'*overhead*[3] sul sistema e nonostante inevitabilmente introduca un piccolo ritardo in più nella consegna delle varie richieste al disco, dovute all'attraversamento di un ulteriore nodo, le misure dimostreranno che esso influisce davvero poco sulle prestazioni ottenute.

3.2 Architettura di geom_sched

Geom_sched è costituito da tre elementi distinti:

1. Un file di libreria (geom_sched.so), che permette di settare e modificare le varie configurazioni;
2. Un modulo generico del kernel (geom_sched.ko), che ne permette il

(3) Overhead : Definisce le risorse accessorie necessarie ad ottenere un determinato scopo a seguito dell'introduzione di un metodo o di un processo più evoluto che ha una maggiore complessità.

funzionamento in FreeBSD e supporta i vari algoritmi di scheduling;

3. Vari altri moduli (gsched_rr.ko, gsched_as.ko, ecc..), ciascuno dei quali permette di attuare una determinata politica di scheduling del disco.

Durante il caricamento, il modulo geom_sched.ko inizializza le varie strutture dati necessarie a comunicare con il kernel e in particolare crea il nuovo percorso dati all'atto di attivazione dello scheduler oltre a memorizzare le configurazioni dei parametri e dei vari algoritmi di scheduling. Tali algoritmi, vengono invocati mediante l'uso dell'*API*[4] di geom_sched, interfaccia che offre sia dei meccanismi di controllo del flusso dei dati, sia il supporto necessario all'attivazione dello scheduler e alla gestione dei vari client. Le funzioni rese disponibili sono le seguenti:

gs_init(), invocata quando un determinato algoritmo di scheduling inizia ad essere utilizzato da un nodo geom_sched.

gs_fini(), al contrario della precedente, viene invocata quando lo scheduler viene rilasciato.

gs_init_class(), chiamata quando al nodo si collega un nuovo client (determinato univocamente da un classificatore)

gs_fini_class(), eseguita quando un client scompare.

gs_start(), invocata all'arrivo di una nuova richiesta di accesso al disco, che si occupa di accodarla e ritorna 0 in caso di esito positivo, o un valore diverso da 0 in caso di errore che determinerà il bypass dello scheduler.

gs_next(), che può essere invocata in 3 casi distinti: subito dopo gs_start() in un ciclo di *g_sched_dispatch()* che è la funzione che si occupa dell'effettiva evasione della richiesta, in caso di timeout, o a seguito del ricevimento di una 'done' che indica un'operazione di I/O dello scheduler andata a buon fine.

Essa ritorna immediatamente o un puntatore alla 'bio' relativa al prossimo processo da servire, o NULL in caso al momento non si debba servire alcuna richiesta. C'è anche un parametro 'FORCE' che se settato, se c'è una richiesta da servire restituisce comunque un puntatore relativo ad essa.

(4) API : Acronimo che sta per Application Programming Interface, ovvero l'interfaccia di programmazione di un'applicazione, che indica l'insieme di procedure disponibili al programmatore.

gs_done(), chiamata ogni qualvolta una richiesta in servizio viene completata, che dopo aver verificato che il timeout relativo alla coda di processi in esecuzione non è ancora scattato, si occupa di invocare nuovamente il ciclo di dispatch per servire le eventuali richieste pendenti.

Uno schema interno delle strutture utilizzate da geom_sched è riportato in figura 3.2, dove si può distinguere un blocco 'g_sched_softc{}' che viene creato con il comando "*geom sched insert*".

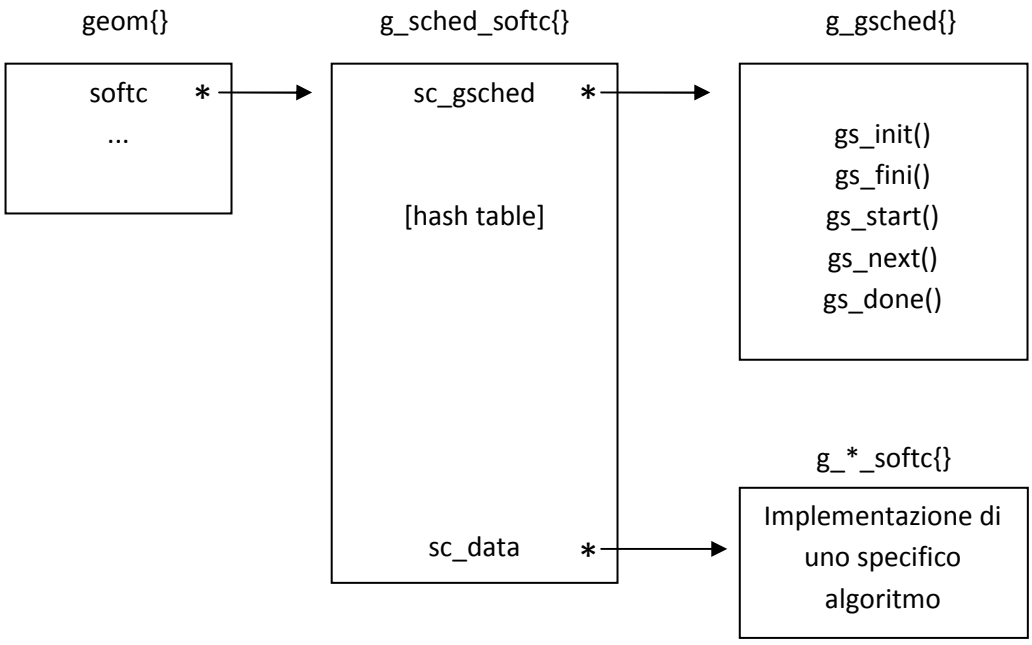


Fig. 3.2 - Struttura del modulo geom_sched.ko

A seguito di tale comando, si imposta sc_gsched in modo da puntare alla struttura g_gsched{}, contenente l'interfaccia comune utilizzata dai vari algoritmi di scheduling. Viene poi immediatamente effettuata una chiamata a gs_init() per creare g_*_softc{} contenente l'implementazione dello specifico algoritmo di scheduling che si intende attivare (specificato all'attivazione), che non ha bisogno di conoscere niente a proposito di GEOM, ma che si occupa solo di manipolare opportunamente le 'bio' che riceve, relative alle varie richieste di accesso al disco. Le altre funzioni contenute nel blocco g_gsched{} (gs_start, gs_next, ...)

vengono chiamate durante il funzionamento dello scheduler a seconda delle necessità.

3.3 Meccanismo di INSERT/DELETE trasparente

Come detto, geom_sched rappresenta un modulo GEOM utilizzato per la gestione di algoritmi di scheduling del disco. Come tutti i moduli ha quindi bisogno di essere caricato, anche se conviene inserirlo come nodo del grafo che caratterizza la struttura di GEOM in maniera del tutto trasparente, in modo che si possa abilitare o disabilitare su un determinato filesystem del sistema nel modo più indolore possibile. In FreeBSD infatti i vari filesystem montati sono elencati nel file /etc/fstab e ciò che si vuole è che essi non cambino a seconda della presenza o meno dello scheduler.
In GEOM abbiamo dei provider e degli oggetti GEOM e supponiamo di voler inserire uno scheduler che controlli un'unità disco USB. Seguendo la terminologia utilizzata in FreeBSD, che utilizza il prefisso "da" come codice dispositivo per un disco USB (a differenza di quella "ad" utilizzata per un disco IDE), esso sarà identificato ad esempio dal provider "da0" ed accessibile mediante il puntatore 'pp': originariamente 'pp' è assegnato al geom "da0" (che ha lo stesso nome ma che rappresenta un'istanza della classe di trasformazione) accessibile attravero il puntatore old_gp:

Prima -->[pp -->old_gp...]

Una normale operazione di creazione dello scheduler, effettuata tramite il comando "geom sched create da0", creerebbe un nuovo nodo costituito da geom_sched al di sopra del provider "da0" puntato da 'pp' costituito da una nuova porta provider indentificata con "da0.sched." e indirizzata da new_pp come indicato:

Dopo il 'create' -->[newpp -->gp -->cp] --->[pp -->old_gp...]
|____nuovo nodo_____|

Verrebbe quindi automaticamente creato un intero albero, a monte di 'new_pp', sul quale noi potremmo fare le nostre operazioni di I/O, ma in questo modo tutte le richieste rivolte a "da0" continuerebbero ad essere indirizzate tramite il puntatore 'pp' mentre per far sì che le nuove richieste subiscano la schedulazione voluta si dovrà deviarle usando il puntatore 'new_pp'.
Con il meccanismo di inserimento trasparente invece, e quindi utilizzando il comando "geom sched insert da0", il provider originale "da0" viene invece collegato con il nodo di geom_sched nel seguente modo:

Dopo l'insert -->[pp -->gp -->cp] -->[newpp -->old_gp...]
|_____nuovo nodo_____|

ovvero l'intero blocco viene inserito all'interno di quello che avevamo prima e quindi ogni operazione di I/O rivolta al provider "da0" viene automaticamente fatta passare attraverso lo scheduler senza bisogno di doverla reindirizzare. L'intero processo è rappresentato in figura:

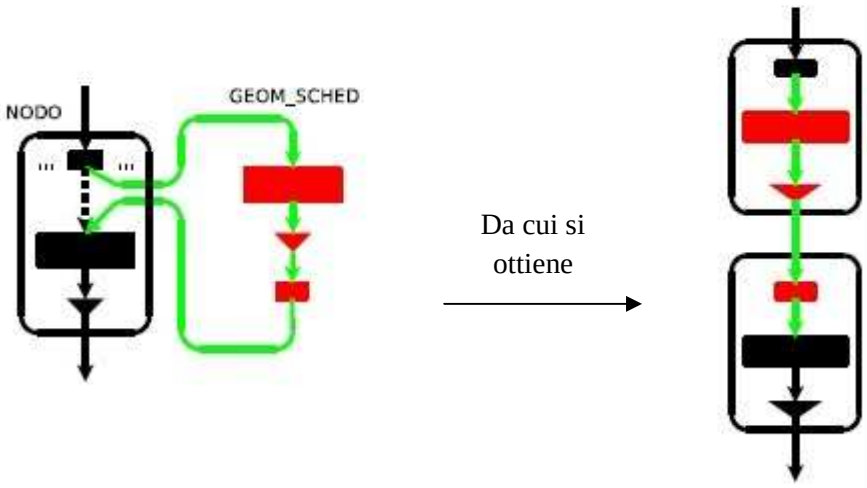


Fig. 3.3 - Meccanismo di INSERT trasparente

Quando l'utilizzo dello scheduler non si rende più necessario, con il comando "geom sched destroy da0.sched." si può altrettanto semplicemente effettuare un'operazione di delete trasparente che va a ripristinare la catena originale e

quella che è la normale gestione delle richieste di accesso al disco realizzata in FreeBSD.

3.4 Classificazione delle richieste di I/O in geom_sched

Lo scheduler si affida ad un classificatore per raggruppare in code le varie richieste di I/O rivolte al disco, ed effettua tale raggruppamento basandosi su di un attributo del creatore della richiesta stessa chiamato *flowid* che ne indica univocamente il processo di appartenenza. Tale meccanismo è rappresentato in figura 3.4:

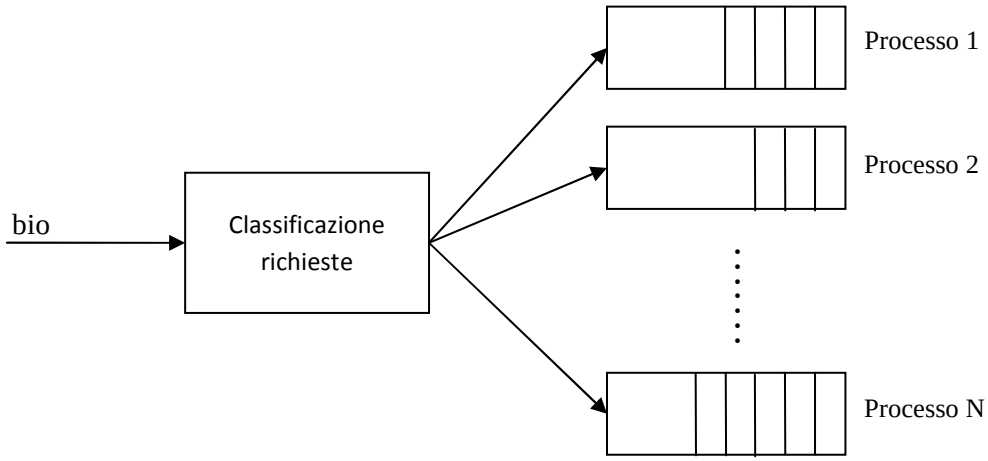


Fig. 3.4 - Classificazione delle richieste

All'interno di ciascuna coda si cerca inoltre di sfruttare la località delle varie richieste, infatti esse non vengono memorizzate nell'ordine di arrivo ma riordinate utilizzando *disksort*, che è una funzione che implementa l'algoritmo di scheduling C-SCAN. Le code non sono quindi di tipo FIFO ma ordinate in modo da minimizzare i tempi di seek.

Al momento attuale c'è però da dire che, per classificare le richieste, geom_sched all'atto del suo caricamento và ad eseguire la modifica di una funzione del kernel di FreeBSD che è g_io_request(). In pratica le varie richieste di I/O, come detto rappresentate da una struttura chiamata 'bio', portano con sé oltre alle informazioni standard anche l’etichetta flowid contenente le informazioni utili ai

fini del raggruppamento delle richieste stesse. Questa etichetta viene inserita dalla g_io_request() modificata, all'interno di un campo già esistente all'interno della struttura 'bio', *bio_caller1*, che non è nato per trasportare questo tipo di informazione, ma essendo usato raramente si presta bene a tale scopo, rendendo comunque l'uso di geom_sched incompatibile con quello di altri moduli GEOM come ZFS e Journal (proprio perché essi usano il campo in cui lo scheduler inserisce il flowid).

In questo modo lo scheduler, a seconda delle proprie necessità, è in grado di risalire la catena di bio per controllare appunto il campo flowid ed effettuare le dovute valutazioni al fine di decidere quale richiesta debba essere scelta per andare in esecuzione.

All'atto della disattivazione dello scheduler viene invece ripristinato il normale funzionamento di g_io_request(), rendendo così nuovamente disponibile il campo contenente flowid per un'eventuale utilizzo da parte di altri moduli.

Questa soluzione è stata introdotta poiché permette l'utilizzo di geom_sched senza dover ricompilare l'intero nucleo del sistema operativo, anche se la patch in questione è attualmente valida solo per kernel i386/amd64 compilati con i flag standard, mentre per altri tipi di configurazione hardware sarà necessario andare a modificare manualmente un file sorgente del kernel (/sys/geom/geom_io.c).

Per risolvere definitivamente questa situazione, è comunque prevista per il futuro l'aggiunta di un campo all'interno della struttura bio, che andrà a contenere un classificatore dedicato a contenere il flowid, ed anche una modifica finale alla funzione g_io_request() che permetta di andare a lavorare direttamente sul campo in questione. Questo di conseguenza comporterà anche una piena compatibilità di geom_sched con gli altri moduli GEOM sopra menzionati, con cui al momento non è compatibile, proprio perché la modifica renderà nuovamente disponibile il campo a loro dedicato.

3.5 Analisi del modulo gsched_rr

Passiamo ora all'analisi del modulo *gsched_rr*, che realizza l'implementazione dell'algoritmo di scheduling Round Robin per l'utilizzo in geom_sched, il cui

obiettivo è quello di migliorare il throughput rispetto al puro algoritmo Elevator offerto da FreeBSD, assicurando al tempo stesso equità di servizio tra i vari processi.

Si può dire che lo scheduler è realizzato su due diversi livelli: al primo livello si crea, per ciascun processo che ha necessità di effettuare operazioni sul disco, una coda di richieste che viene gestita e riordinata utilizzando l'algoritmo C-SCAN utilizzando come detto disksort. Al livello successivo viene invece implementato il Round Robin vero e proprio, facendo in modo che ciascuna coda di richieste non venga servita per più di un determinato budget, sia temporale che di dati. Quindi, in pratica, a ciascun processo viene concesso l'uso del disco per un certo intervallo di tempo o per trasferire una certa quantità di dati, durante il quale esso andrà ad effettuare operazioni di I/O riducendo, e talvolta anche svuotando, la coda di richieste ad esso associata.

Inoltre c'è da dire che l'algoritmo differisce leggermene dal puro e semplice Round Robin, poiché viene effettuata anche anticipazione sulla coda di richieste attiva, al fine di ridurre le operazioni di seek.

Il codice sorgente completo di questo modulo è contenuto all'interno del file gs_rr.c. La struttura dati che realizza ciascuna coda è chiamata *g_rr_queue* ed è definita come segue:

```
struct g_rr_queue {
        struct g_rr_softc       *q_sc;                  /* link to the parent */

        enum g_rr_state         q_status;
        unsigned int            q_service;              /* service received so far */
        int                     q_slice_end;            /* actual slice end in ticks */
        enum g_rr_flags         q_flags;                /* queue flags */
        struct bio_queue_head q_bioq;

        /* Scheduling parameters */
        unsigned int            q_budget;               /* slice size in bytes */
        unsigned int            q_slice_duration;       /* slice size in ticks */
        unsigned int            q_wait_ticks;           /* wait time for anticipation */

        /* Stats to drive the various heuristics. */
        struct g_savg  q_thinktime;                     /* Thinktime average. */
        struct g_savg  q_seekdist;                      /* Seek distance average. */
```

```
        int             q_bionum;               /* Number of requests. */
        off_t           q_lastoff;              /* Last submitted req. offset. */
        int             q_lastsub;              /* Last submitted req. time. */

        /* Expiration deadline for an empty queue. */
        int             q_expire;

        TAILQ_ENTRY(g_rr_queue) q_tailq;        /* RR list link field */
};
```

Essa è composta quindi da numerosi elementi, sia utili alla gestione della coda come q_bioq che punta al primo elemento, q_flags che indica se è stata selezionata o meno una nuova coda di richieste o q_service e q_slice_end che indicano i dati elaborati finora e il tick di fine slice, sia di configurazione come q_budget e q_slice_duration che indicano rispettivamente la dimensione in bytes e in ticks del budget associato alla coda, sia statistici ed utili all'anticipazione come q_thinktime che indica il thinktime medio e q_seekdist che indica la distanza di seek media. Il campo q_status indica invece lo stato attuale della coda, e distingue tra READY, in cui si passa immediatamente a servire la prima richiesta pendente, BUSY che indica che una richiesta è in servizio ed è quindi necessario attendere il termine della sua esecuzione, e IDLE che segnala di non servire immediatamente le richieste in arrivo ma di aspettare finché la coda non avrà i requisiti per ripristinare la propria esecuzione.

La struttura che si occupa invece di implementare il Round Robin tra le varie code è *g_rr_softc*, così definita:

```
struct g_rr_softc {
        struct g_geom       *sc_geom;

        struct g_rr_queue   *sc_active;
        struct callout      sc_wait;            /* timer for sc_active */

        struct g_rr_tailq   sc_rr_tailq;        /* the round-robin list */
        int                 sc_nqueues;         /* number of queues */

        /* Statistics */
        int                 sc_in_flight;       /* requests in the driver */

LIST_ENTRY(g_rr_softc)      sc_next;
};
```

La coda su cui stiamo attualmente effettuando la schedulazione è indicata dal campo sc_active, che può essere modificato solo dalla funzione *g_rr_next()* che si occupa di scegliere la prossima coda da mandare in esecuzione. Le altre code su cui viene eseguito il Round Robin sono invece immagazzinate nella struttura sc_rr_tailq, il cui numero è indicato da sc_nqueues.

Un'altra struttura interessante da analizzare, poiché contiene vari parametri di configurazione dello scheduler, è g_rr_params definita nel seguente modo:

```
struct g_rr_params {
        int     queues;                 /* total number of queues */
        int     w_anticipate;           /* anticipate writes */
        int     bypass;                 /* bypass scheduling writes */

        int     units;                  /* how many instances */
        /* sc_head is used for debugging */
        struct g_scheds         sc_head;        /* first scheduler instance */

        struct x_bound queue_depth;     /* max parallel requests */
        struct x_bound wait_ms;         /* wait time, milliseconds */
        struct x_bound quantum_ms;      /* quantum size, milliseconds */
        struct x_bound quantum_kb;      /* quantum size, Kb (1024 bytes) */

        /* statistics */
        int     wait_hit;               /* success in anticipation */
        int     wait_miss;              /* failure in anticipation */
};
```

Essa contiene il numero di processi con code di richieste attive, alcuni flag che controllano il funzionamento dello scheduler in scrittura, statistiche sull'anticipazione, ma soprattutto permette di immagazzinare i parametri d'impostazione dello scheduler medianti i campi queue_depth, wait_ms che indica il tempo di anticipazione e quantum_ms e quantum_kb che contengono i possibili valori temporali e di dati relativi ai budget del Round Robin.

Capitolo 4.

Test prestazionali e risultati

Giunti a questo punto possiamo passare ad analizzare quelle che sono le prestazioni che si riescono a raggiungere grazie all'utilizzo del modulo geom_sched. I test prestazionali sono stati eseguiti al fine di valutare l'incremento o meno di prestazioni con l'utilizzo dello scheduler Round Robin confrontate con quelle offerte dall'algoritmo di scheduling standard offerto da FreeBSD, nonché per accertarsi dell'effettiva validità del modello di ottimizzazione delle richieste offerto da questo sistema che come già detto non è mai stato testato a fondo. Di seguito descriveremo le varie configurazioni hardware con cui abbiamo effettuato le varie prove velocistiche, per poi mostrare quelli che sono stati i risultati ottenuti.

4.1 Configurazioni di test

Per ottenere dei dati prestazionali in grado di fare delle considerazioni complete e sicure riguardo il comportamento dello scheduler Round Robin in FreeBSD, le prove sono state effettuate su più di una macchina con caratteristiche hardware diverse tra loro, permettendo di analizzare le performance offerte dallo scheduler in situazioni operative differenti.

Inizialmente, per garantire un buon livello di parallelismo, avevamo iniziato l'analisi dei moduli GEOM utilizzando *Virtualbox*, un software di virtualizzazione che permette di eseguire FreeBSD come sistema operativo ospite anche su una macchina sulla quale è installato un sistema operativo differente (nel nostro caso Microsoft Windows 7), emulando i dischi mediante speciali file con estensione *.VDI* (che sta per per Virtual Disk Image). Questa soluzione però

è stata presto abbandonata, sia per problemi di temporizzazione, sia per problemi legati alla gestione dell'I/O. Infatti utilizzando FreeBSD come sistema operativo ospite (così come qualsiasi altro sistema operativo del resto) si va incontro ad una doppia gestione delle richieste di accesso al disco, poiché esso adotta le proprie politiche di controllo del'I/O, ma quello su cui va a leggere e scrivere non è in realtà un disco fisico, bensì un file immagine che lo emula. Virtualbox fa da tramite tra i due sistemi operativi, anche se in realtà quello che si occupa di portare a termine fisicamente la lettura o scrittura sul disco è il sistema operativo ospitante, il quale segue i propri meccanismi. Questo tipo di comportamento può mascherare quindi quelle che sono le effettive prestazioni del sistema operativo ospite, rendendo inconsistenti i dati che si possono ottenere. Nel nostro caso infatti le performance raggiunte in lettura e scrittura erano incongruenti con le effettive potenzialità del disco utilizzato.

Le prove sono state perciò eseguite su due laptop in cui FreeBSD girava in maniera nativa: nel primo caso abbiamo impiegato una macchina, indicata d'ora in poi come "Macchina 1", dotata di processore Intel i7 720QM con una frequenza di clock di 1.6 GHz e 4GB di memoria RAM di tipo DDR3 alla quale abbiamo collegato un disco esterno con interfaccia USB di 500GB di capacità. La "Macchina 2" è invece dotata di un processore Intel Centrino Duo a 1,66GHz di clock e 1GB di memoria RAM di tipo DDR, e per le prove si è utilizzato il disco interno IDE da 100GB.

Per realizzare i test abbiamo dovuto inoltre ottenere dei pattern di richieste differenti, sia sequenziali (in lettura e in scrittura), per cui si è utilizzato il comando *dd* che esegue operazioni su settori adiacenti del disco (impostando come dimensioni del blocco 128KB), sia casuali, per cui si è utilizzato *Subversion* (o *SVN*) che è un programma che esegue accessi sparsi in zone diverse del disco (generando così numerose seek) effettuando il checkout di un *repository*[5] appositamente creato. I valori riportati di seguito rappresentano una media ottenuta ripetendo più volte i test e mantenendo inalterati i parametri.

(5) Repository : E' un archivio di dati che memorizza informazioni nella forma di un filesystem ad albero. Si occupa di memorizzare le modifiche ai vari file che contiene e può essere scritto per aggiornare tali file, o letto per recuperare le varie versioni modificate degli stessi.

4.2 Risultati ottenuti con richieste sequenziali

Nei grafici che seguono sono riportati i risultati dei test effettuati sulle due macchine effettuando sul disco operazioni di lettura o scrittura di soli blocchi contigui di dati. La figura 4.1 mostra le differenze in sola lettura tra le performance ottenute con scheduler disattivato e attivo: come si può notare, con questo pattern di richieste sequenziali, si ha un throughput molto elevato e l'attivazione dello scheduler non si fa sentire più di tanto poiché l'algoritmo di scheduling C-LOOK implementato in FreeBSD e il C-SCAN implementato nello scheduler GEOM per la coda di richieste relativa all'unico processo attivo, sono pressoché equivalenti. La leggera differenza che si nota con lo scheduler attivo è dovuta principalmente all'overhead introdotto interrompendo e rischedulando il processo al termine di ogni budget temporale previsto dal Round Robin.

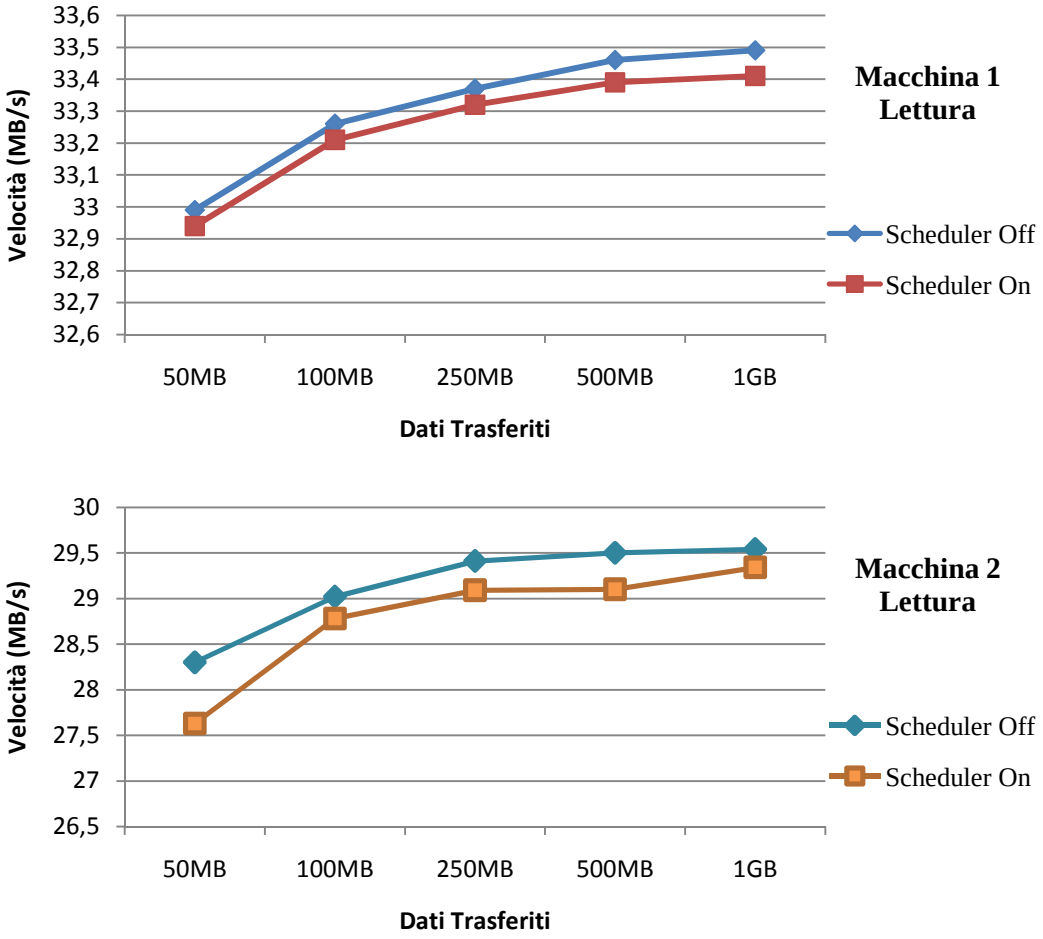


Fig. 4.1 - Risultati ottenuti con letture sequenziali

Di seguito sono invece riportati i risultati dei test eseguiti effettuando sul disco sole scritture su blocchi di dati contigui. Le modalità sono analoghe a quelle del test precedente, ed analizzando la figura 4.2 si può notare come anche in questo caso il throughput con scheduler attivo si mantenga ad un valore molto simile a quello che si otterrebbe senza il suo utilizzo, con il solito piccolo distacco che si ha a causa della maggiore complessità nella gestione dell'algoritmo Round Robin. Si nota infatti un leggero calo delle prestazioni che in percentuale resta però molto basso, addirittura quasi trascurabile, aggirandosi intorno allo 0,5% per quanto riguarda le prove effettuate sulla macchina 1 e intorno all'1,5-2% sulla macchina 2.

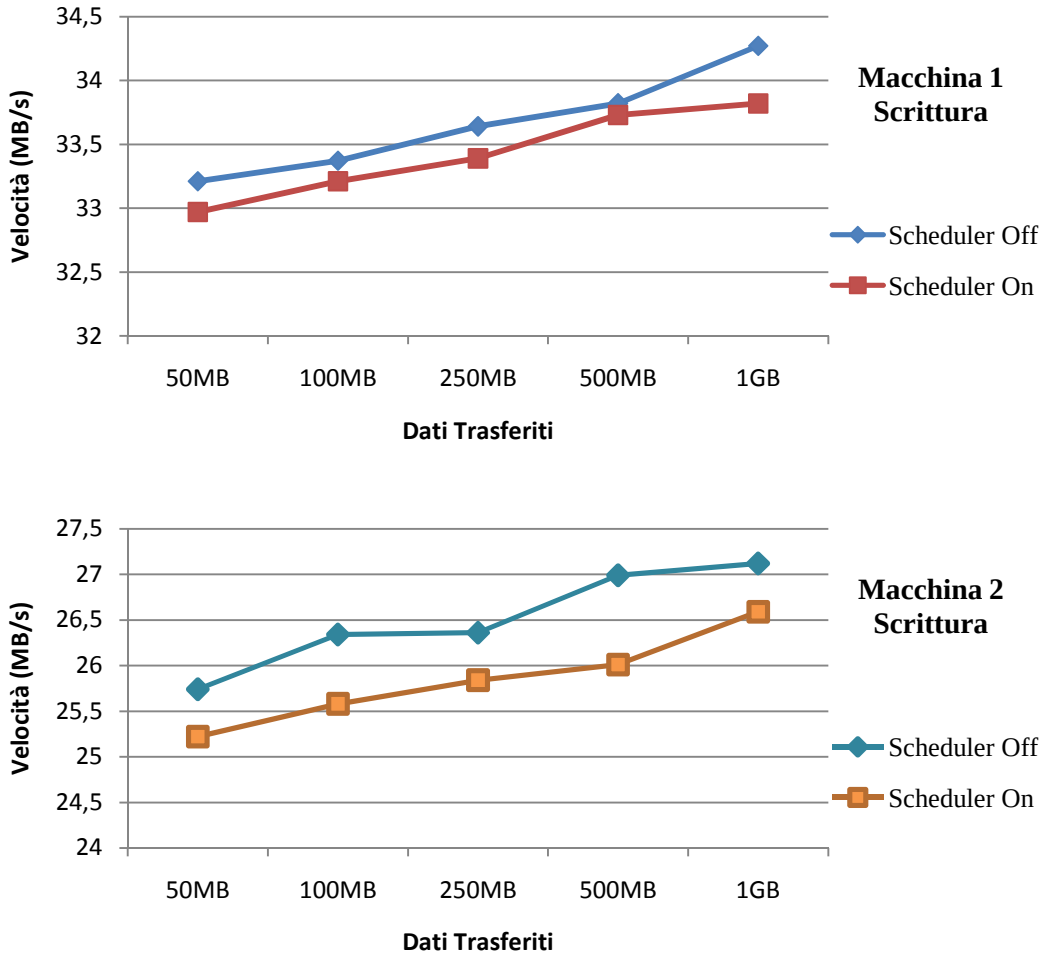


Fig. 4.2 - Risultati ottenuti con scritture sequenziali

Come era possibile immaginare si nota comunque chiaramente che accedendo al disco solo in maniera sequenziale, l'attivazione dello scheduler GEOM non è

affatto indispensabile, benché non sconveniente, poiché gli algoritmi di scheduling tendono a comportarsi in maniera analoga e non si notano differenze a livello prestazionale .

4.3 Risultati ottenuti con richieste casuali e sequenziali

Passiamo ora ad analizzare i dati relativi all'esecuzione concorrente di un processo che genera pattern di richieste casuali ed uno che, come nel caso precedente, effettua sul disco accessi sequenziali.

In figura 4.3 sono riportati i risultati ottenuti eseguendo, oltre a subversion, delle operazioni di lettura. Va sottolineato che i dati relativi a subversion rappresentano un valore medio calcolato utilizzando le dimensioni del repository e il tempo necessario ad effettuarne il checkout.

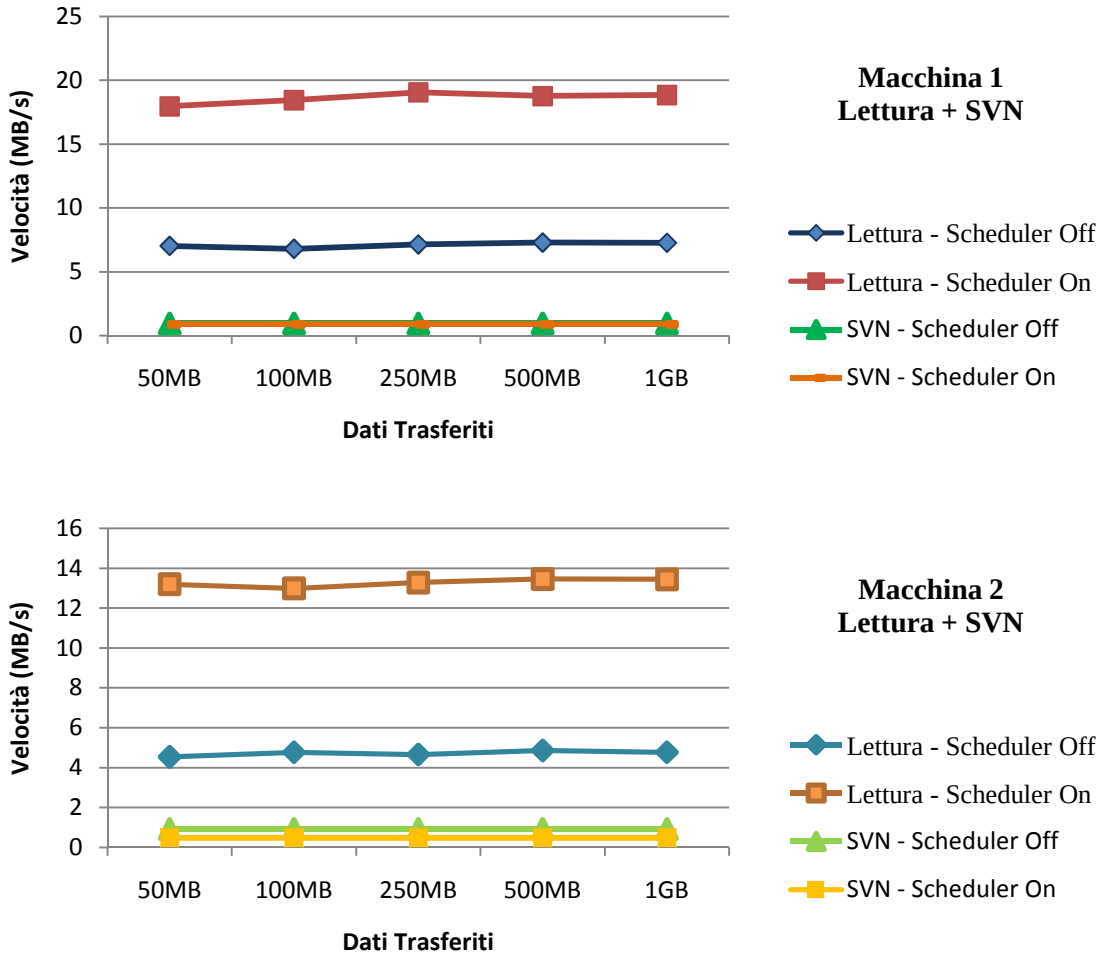


Fig. 4.3 - Risultati ottenuti con letture sequenziali e accessi casuali

In questo caso la presenza dello scheduler si fa notare molto infatti, nonostante non si noti una grossa differenza (anche a causa della scala del grafico) nell'esecuzione degli accessi casuali con subversion che rimane comunque abbastanza fluida in entrambi i casi, in lettura si ottiene un notevole incremento prestazionale andando a raddoppiare ed oltre la velocità di elaborazione. In pratica si ha, per una riduzione delle prestazioni dal 10 al 30% con accessi casuali al disco (che comunque non raggiunge mai neanche 1MB/s), un incremento della velocità di lettura del 160 - 180% circa con pattern sequenziali.
Lo stesso tipo di analisi può essere effettuata nel caso in cui oltre all'accesso casuale ai dati si operi anche sequenzialmente in scrittura, scenario riportato nella seguente figura 4.4:

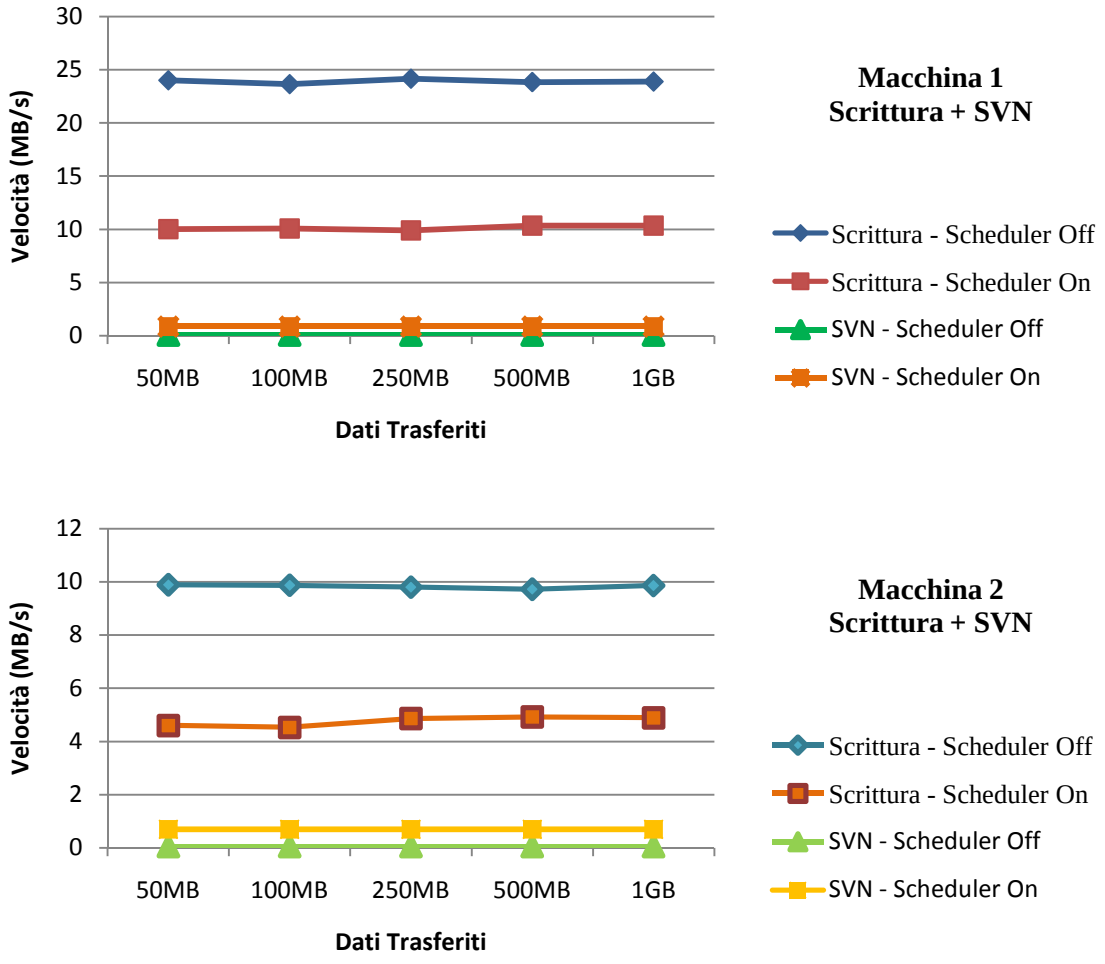


Fig. 4.4 - Risultati ottenuti con scritture sequenziali e accessi casuali

Qui la situazione risulta ribaltata, poiché attivando lo scheduler si nota un netto decremento delle prestazioni per quanto riguarda le scritture sequenziali (di circa il 50% o di poco superiori) ma un aumento notevole nella fluidità delle operazioni ad accesso casuale, dato che a scheduler disattivato questo tipo di pattern viene praticamente scartato dall'algoritmo C-LOOK nativo di FreeBSD andando a rallentare enormemente l'esecuzione di questo tipo di richieste. Questo avviene perché tale algoritmo evita di effettuare seek (introdotte eseguendo appunto Subversion) eseguendo tali richieste solo quando la testina di lettura/scrittura si trova nella posizione corrispondente sul disco, e riducendo di fatto anche l'equità di servizio garantita invece dallo scheduler Round Robin.

Capitolo 5.

Test funzionali e risultati

Passiamo a questo punto a studiare l'effettivo funzionamento dello scheduler Round Robin implementato in geom_sched, andando ad analizzare nel dettaglio se il funzionamento corrisponde effettivamente con quello voluto.

5.1 Ricompilazione del kernel per l'abilitazione di KTR

Per poter monitorare il comportamento dello scheduler sono state necessarie alcune operazioni preliminari, e la prima tra queste è stata l'abilitazione di un meccanismo di tracciamento degli eventi del kernel offerto da FreeBSD cioè *KTR*, che permette di memorizzare una determinata serie di eventi che accadono mentre il kernel è in esecuzione, su di un file log che viene creato e scritto in tempo reale e poi memorizzato su disco al fine di analizzarne il contenuto. KTR non è costituito da un modulo che è possibile caricare quando il sistema operativo è in esecuzione, ma necessita della ricompilazione statica dell'intero kernel di FreeBSD, poiché per abilitare tale funzionalità è necessario aggiungere delle opzioni di compilazione nel file di configurazione del kernel stesso. Per far questo siamo quindi andati a prendere quelle che sono le opzioni comuni contenute nel file di configurazione del kernel standard di FreeBSD (chiamato *GENERIC* e situato nella directory /usr/src/sys/i386/conf) e ad inserirle in un nuovo file chiamato *MIOKERNEL*, in cui abbiamo aggiunto quelle necessarie ad attivare le funzionalità di KTR che sono le seguenti:

- *options KTR* Abilita l'attività di KTR
- *options ALQ* Permette di abilitare il salvataggio del file log su disco

- *options KTR_ALQ*	Dipende dalla precedente definendo il modo in cui salvare i dati
- *options KTR_ENTRIES*	Determina la dimensione del buffer di eventi da memorizzare
- *options KTR_COMPILE*	Indica il tipo di eventi da compilare nel kernel
- *options KTR_MASK*	Determina il tipo di eventi da memorizzare durante l'esecuzione
- *options KTR_CPUMASK*	Stabilisce di quale CPU si debbano memorizzare gli eventi
- *options KTR_VERBOSE*	Opzionale, indica se mostrare o meno i risultati a run-time

A questo è seguita l'esecuzione dalla directory /usr/src dei comandi:

 # make buildkernel KERNCONF=MIOKERNEL

 # make installkernel KERNCONF=MIOKERNEL

necessari rispettivamente a compilare ed installare il kernel personalizzato utile ai nostri scopi.

5.2 Script per KTR e tracce

Giunti a questo punto siamo andati ad inserire, sia all'interno del file contenente il codice sorgente di geom_sched (che è gsched.c), sia in quello di gsched_rr che implementa l'algoritmo di scheduling Round Robin (che è gs_rr.c), delle tracce relative agli eventi più significativi, che possono permetterci di capire l'effettivo andamento delle cose, associando ad esse informazioni temporali e specificando a quale processo è associata la coda in esecuzione in un determinato momento.

Gli eventi tracciati sono i seguenti:

- START, che indica l'arrivo di una richiesta di I/O al disco;
- NEXT, che rappresenta il meccanismo di selezione della prossima richiesta da mandare in esecuzione;
- DISPATCH, che indica invece l'effettivo inizio di esecuzione di una determinata richiesta;

- DONE, che rappresenta il termine dell'esecuzione della richiesta attuale.

Il tracciamento degli eventi avviene quindi secondo la sequenza illustrata in figura:

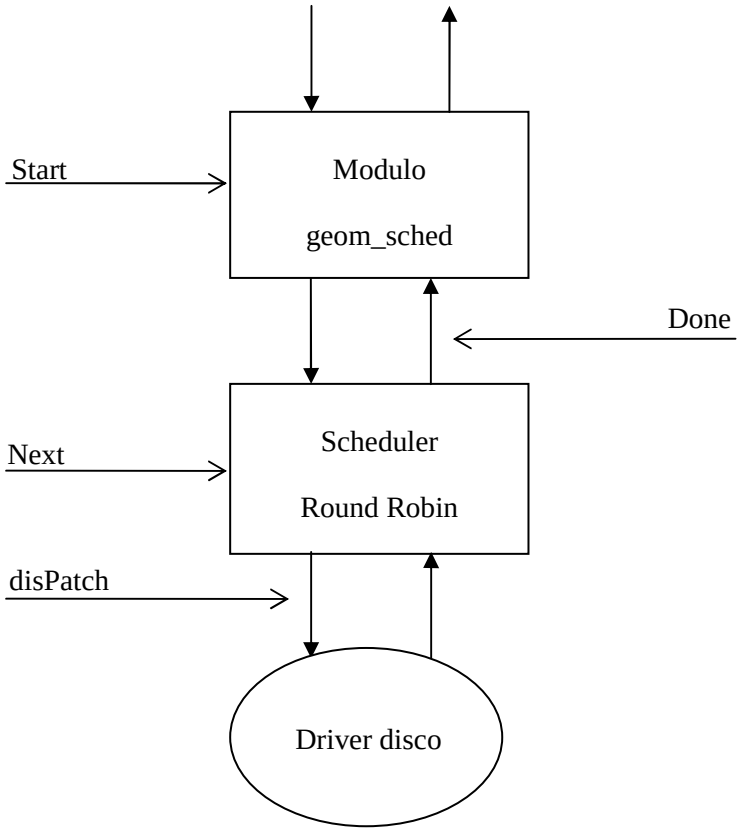


Fig. 5.1 – Schema di tracciamento degli eventi

Le tracce inserite all'interno dei file sono del tipo:

```
void
g_sched_trace_bio_EVENT(struct bio *bp){
        CTR5(KTR_GSCHED, "E %d %c %lu/%lu %lu", g_sched_issuer_pid(bp),
                        g_sched_type(bp), ticks, bp->bio_offset, bp->bio_length);
}
```

dove, al posto di EVENT, si va a sostituire il nome dell'evento che si desidera tracciare, e al posto della prima lettera tra virgolette quello dell'iniziale dell'evento, che sarà quella mostrata successivamente nel file di debug. Il dispatch della richiesta è stato indicato con 'P' per distinguerlo dal done che ha la stessa iniziale. I dati visualizzati nella traccia vanno ad indicare nell'ordine:

- Il processo a cui appartiene la coda di richieste in questione, indicato da g_sched_issuer_pid(bp);
- Il tipo di operazione effettuata (R per lettura o W per scrittura), indicato da g_sched_type(bp);
- Il tick relativo all'istante in cui si verifica l'evento;
- L'offset relativo al file su cui andremo ad operare, indicato da bp->bio_offset;
- La dimensione della richiesta espressa in bytes, indicata da bp->bio_length.

Per ottenere dei risultati significativi è stato necessario ricorrere anche in questo caso all'esecuzione concorrente sia di un processo che genera un pattern di richieste sequenziali come 'dd', sia di uno che ne genera di casuali portando all'esecuzione di numerose seek come 'subversion', in modo da creare artificialmente una condizione di carico che può verificarsi anche in un contesto di uso normale, in cui non si ha solo un certo tipo di pattern di richieste di accesso ma si ha una situazione che è abbastanza varia, per rispecchiare anche il caso in cui più utenti fanno uso (in maniera diversa) del disco. Per ricreare questo tipo di situazione è stato necessario aprire due diverse *shell*[6] (o terminali) in FreeBSD, eseguendo in una il comando dd e nell'altra svn, ma per poter memorizzare i dati ottenibili con ktr è comunque necessario effettuare una *sysctl* (che nei sistemi BSD è una chiamata di sistema che viene utilizzata per esaminare e modificare dinamicamente diversi parametri del kernel) per abilitarlo ad effettuare la scrittura su file. Per far questo si è utilizzato lo script *BASH*[7] riportato di seguito, contenuto nel file *ktr_trace.sh*:

```
#!/bin/sh

#Mostra come usare lo script e controlla se l'invocazione e' corretta
istruzioni() {
        echo "Utilizzo: ./ktr_trace.sh CMD ARGS"
```

(6) Shell : E' l'ambiente di lavoro attraverso il quale un utente può comunicare con il sistema eseguendo comandi e richiedendo l'esecuzione di programmi. Insieme al kernel rappresenta una delle componenti principali del sistema operativo.

(7) BASH : Acronimo per Bourne Against SHell, che è un interprete di comandi testuale usato nei sistemi Unix-like che utilizza la reindirezione dell'I/O per eseguire più programmi in cascata. [Per info www.gnu.org]

```
        exit
}

if [ "$#" -lt 1 ] ; then
        istruzioni
fi

#Crea un ramdisk e lo monta su /mnt
mdmfs -s 256m /dev/md0 /mnt

#Abilita KTR e lo fa loggare su /mnt/trace
sysctl debug.ktr.alq_file=/mnt/trace

#Viene impostata la maschera KTR_SPARE4 che e' quella usata dallo
#scheduler per lasciare delle tracce a noi utili
sysctl debug.ktr.mask=65536

#Abilita il log su file
sysctl debug.ktr.alq_max=0
sysctl debug.ktr.alq_enable=1
#sysctl debug.ktr.verbose=2

#Esegue il comando da tracciare che gli passo da linea di comando
$*

#Disabilita il logging su file
sysctl debug.ktr.alq_enable=0

#Gestisce in modo pulito l'uscita e l'unmount
sleep 5
cp /mnt/trace /tmp/trace

i=0
while [ $i -le 10 ] ; do
        if umount /mnt ; then
                break;
        fi
        sleep 2
        i = $(expr $i + 1)
done

mdconfig -d -u 0

#Converte la traccia binaria contenuta in /tmp/trace in file di testo
#aggiungendo per ogni risultato un timestamp assoluto
```

Per ottenere la traccia desiderata viene eseguito Subversion con il comando:

 # svn checkout file:///repos_svn/archivio_svn archivio_test_X

che va ad operare su un repository di adeguate dimensioni appositamente creato per eseguire i test. Contemporaneamente a tale esecuzione si va a lanciare anche lo script con la riga di comando:

 # ./ktr_trace.sh dd if=/dev/da0 of=/dev/null bs=128k count=800

Quello che si va ad ottenere è una traccia del tipo:

index	*timestamp*	*trace*			
...					
131	*262516536534*	*S 100085*	*R*	*127695/1835008*	*131072*
132	*262517427168*	*N 100085*	*R*	*127695/1835008*	*131072*
133	*262519753866*	*P 100085*	*R*	*127697/1835008*	*131072*
134	*262526141706*	*D 100085*	*R*	*127701/1835008*	*131072*
135	*262526657385*	*N 100056*	*W*	*127701/2628812288*	*131072*
136	*262527206778*	*P 100056*	*W*	*127701/2628812288*	*131072*
137	*262527864468*	*S 100085*	*R*	*127702/1966080*	*131072*
138	*262533921291*	*D 100056*	*W*	*127705/2628812288*	*131072*
139	*262539447534*	*N 100056*	*R*	*127709/971734528*	*114688*
140	*262541831724*	*P 100056*	*R*	*127710/971734528*	*114688*
141	*262546504656*	*S 100056*	*W*	*127713/2629041664*	*131072*
142	*262547703276*	*D 100056*	*R*	*127714/971734528*	*114688*
143	*262548281568*	*N 100059*	*W*	*127714/1475542528*	*16384*
144	*262548858723*	*P 100059*	*W*	*127715/1475542528*	*16384*
...					

su cui ovviamente dovremo effettuare delle opportune elaborazioni al fine di ottenere delle informazioni consistenti sul comportamento dello scheduler.

In questo esempio siamo andati ad operare in lettura con dd su di un disco USB dal quale vengono letti 100MB in blocchi da 128KB e scritti in /dev/null che è il cosiddetto *null device* (o dispositivo nullo), che ha la caratteristica di scartare tutti i dati che vi vengono scritti.

5.3 Programma awk per l'analisi delle tracce

Giunti a questo punto possiamo andare ad analizzare i dati ottenuti per vedere se lo scheduler rispetta il comportamento previsto. Per fare ciò risulta immediato andare a controllare se vengono rispettati i budget imposti dall'algoritmo Round Robin, cosa che è stata effettuata in maniera automatica mediante un altro script che verifica che al termine dei ticks o dei dati disponibili per ciascun processo venga effettivamente tolto l'uso del disco al processo attivo. Andando ad

analizzare i parametri dello scheduler, il budget temporale corrisponde a 100 ticks (che corrispondono a 100ms) e per il budget dati la soglia massima corrisponde a 8MB, quindi ci si aspetterà che al raggiungimento della minore tra queste due soglie, l'esecuzione venga trasferita ad una coda di richieste diversa da quella attiva.

Il programma utilizzato in questo caso è in linguaggio *awk*, che essendo orientato alla manipolazione delle stringhe di dati e vista la traccia che abbiamo a disposizione, fa esattamente al caso nostro. In pratica awk è un eseguibile che legge il programma vero e proprio da riga di comando o da un file con estensione .awk (come nel nostro caso), e va a cercare determinate sequenze di caratteri specificate nel programma, all'interno di un file di dati (per noi è quello contenente la traccia), effettuando le dovute elaborazioni in caso ne verifichi la presenza. La sintassi di awk è simile a quella utilizzata in C, e quindi di immediata comprensione per chi ha dimestichezza con questo linguaggio di programmazione. Il programma awk utilizzato è il seguente, contenuto nel file controllo_budget.awk:

```
# Programma awk che calcola il budget associato ai vari processi
#
# I simboli che il programma va ad analizzare sono le tracce lasciate
# dallo scheduler GEOM e indicano le seguenti situazioni:
#
# S - Start       : una nuova richiesta arriva allo scheduler (che
#                   inizialmente si occupa di accodarla)
# N - Next        :esecuzione di g_rr_next in gs_rr.c
# P – disPatch:seleziona quale delle richieste accodate deve essere
#                 servita e ne attiva l'esecuzione
# D - Done        :la richiesta in servizio ha terminato la propria
#                 esecuzione (si puo' andare a selezionare la prossima)
#
# Al termine dell'elaborazione, file stats.txt vengono inseriti
# i risultati ottenuti.

/ S /{    #Sincronizzo l'inizio dell'analisi
          #con l'arrivo di una Start
          if(!start_tick){
                    start_tick = substr($6, 0, 7)
          }
}
```

```
/ N /{   if(start_tick){
                tick_attuale = substr($6, 0, 7)
                dati_attuali = ($7 / 1024)
                #Controllo se si tratta dello stesso processo
                #che ho mandato in esecuzione l'ultima volta
                if(last_pid_sched == $4){
                        #Aggiorno i budget
                        budget_temporale = tick_attuale - first_tick_this_pid
                        budget_dati += dati_attuali
                        #Verifico di non aver superato il budget temporale
                        if(budget_temporale >= 100){
                                #Memorizzo il numero di sforamenti
                                #e il massimo budget ottenuto
                                time_exceeds++
                                if(durata_max <= budget_temporale){
                                        durata_max = budget_temporale
                                        pid_max = $4
                                        line_max = $1
                                }
                        }
                        if(budget_dati >= 8192){
                                #Memorizzo il numero di sforamenti
                                #e la quantita' di dati trasferita
                                data_exceeds++
                                if(trasferiti_max <= budget_dati){
                                        trasferiti_max = budget_dati
                                        pid_dati_max = $4
                                        line_dati_max = $1
                                }
                        }
                }else{  #Se il processo e' cambiato, nella Done
                        #aggiorno il tick di inizio budget e
                        #la quantita' di dati trasferiti
                        aggiorna_parametri = 1
                        this_data = $7
                }
        }
}

/ D / {  if(start_tick){
                #Verifico se aggiornare il budget o meno
                if(aggiorna_parametri == 1){
                        last_pid_sched = $4
                        first_tick_this_pid = substr($6, 0, 7)
                        budget_dati = 0 + (this_data / 1024)
                        aggiorna_parametri = 0
```

```
                        if(time_exceeds > 0){
                                time_outs ++
                                time_exceeds = 0
                        }
                        if(data_exceeds > 0){
                                data_outs ++
                                data_exceeds = 0
                        }
                }
        }
}


/EOF/{  printf "\n\n________STATISTICHE RISULTATI________\n\n" > "stats.txt"
        printf "Durata massima budget temporale:\t\t%i\n\n",
                        durata_max > "stats.txt"
        printf "relativa al PID:\t%s alla riga:\t%s\n\n",
                        pid_max, line_max > "stats.txt"
        printf "Numero complessivo sforamenti budget temporale: %i\n\n\n",
                        time_outs > "stats.txt"
        printf "Quantita' massima dati nel budget (KB):\t\t%i\n\n",
                        trasferiti_max > "stats.txt"
        printf "relativa al PID:\t%s alla riga:\t%s\n\n", pid_dati_max,
                        line_dati_max > "stats.txt"
        printf "Numero complessivo sforamenti budget dati: %i\n",
                        data_outs > "stats.txt"
}
```

Il file "stats.txt" riportato di seguito, contiene i risultati di una delle tante prove effettuate, e mostra dati significativi solo in caso di mancato rispetto dei budget:

```
_____________________________STATISTICHE RISULTATI_____________________________

Durata massima budget temporale:                602

relativa al PID: 100048         alla riga:      17887

Numero complessivo sforamenti budget temporale: 37


Quantita' massima dati nel budget (KB):         0

relativa al PID: 0                      alla riga:      0

Numero complessivo sforamenti budget dati: 0
```

Nonostante i risultati dei test siano stati molto vari si può dire che quelli riportati rappresentano molto bene la situazione che ci siamo trovati di fronte, poiché nel 90% dei casi c'è stato uno sforamento del budget temporale, a differenza di quello dati che non si è mai verificato.

5.4 Monitoraggio della coda e individuazione errore

Per cercare di capire i motivi di questo mancato rispetto delle soglie temporali abbiamo deciso di inserire in gs_rr.c, all'interno della funzione g_rr_next() che è quella che oltre a scegliere il prossimo processo da servire si occupa anche di verificare se sono stati consumati i budget o meno, una traccia che all'atto dello sforamento della soglia temporale ci permettesse di visualizzare lo stato della coda, in modo da vedere se la situazione che ci siamo trovati di fronte era imputabile o meno a qualche controllo o aggiornamento erroneo dei parametri della coda. In pratica siamo andati a stampare quasi tutti gli elementi contenuti nella struttura g_rr_queue, descritta nel capitolo 3 di questa tesi e contenente i parametri relativi alla coda di richieste associata al processo in servizio al momento. La traccia è stata suddivisa in 4 parti, dato che ktr permette di registrare fino ad un massimo di 6 valori contemporaneamente ma a noi non erano sufficienti per avere un quadro completo della situazione, ed è riportata di seguito:

```
void
g_sched_trace_queue_SCHEDSTAT_A(struct g_rr_queue *qp){
        CTR5(KTR_GSCHED, "I1 %i\t%i\t%i\t%i\t%i", qp->q_status,
                qp->q_service, qp->q_slice_end,
                qp->q_flags, debug_flags);
}

void
g_sched_trace_queue_SCHEDSTAT_B(struct g_rr_queue *qp){
        CTR5(KTR_GSCHED, "I2 %i\t%i\t%i\t%i\t%i", qp->q_budget,
                qp->q_slice_duration, qp->q_wait_ticks,
                qp->q_thinktime.gs_avg, qp->q_thinktime.gs_smpl);
}
```

```
void
g_sched_trace_queue_SCHEDSTAT_C(struct g_rr_queue *qp){
        CTR5(KTR_GSCHED, "I3 %i\t%i\t%i\t%i\t%i", qp->q_seekdist.gs_avg,
                qp->q_seekdist.gs_smpl, qp->q_bionum,
                qp->q_lastsub, qp->q_expire);
}

void
g_sched_trace_queue_SCHEDSTAT_D(struct g_rr_queue *qp){
        CTR6(KTR_GSCHED, "I4 %i\t%i\t%i\t%i\t%i\t%i",
                qp->q_service - qp->q_budget, ticks - qp->q_slice_end, qp->q_flags,
                G_FLAG_COMPLETED, qp->q_flags & G_FLAG_COMPLETED,
                gs_bioq_first(&qp->q_bioq));
}
```

Oltre agli elementi propri della struttura e già descritti in precedenza, ne abbiamo aggiunti altri di cui alcuni derivati, come (q_service - q_budget) e (ticks - q_slice_end) che indicano rispettivamente i dati e il tempo ancora disponibili nell'attuale budget, ed altri ex novo come debug_flags inserito e modificato in ogni ramo di g_rr_next che si occupa di controllare le situazioni possibili in cui ci si può trovare, in modo da capire man mano quali rami di scelta erano stati seguiti.

Questa traccia ci ha permesso di individuare un errore nell'aggiornamento di q_slice_end, che rappresenta il tick in cui il budget temporale risulterà terminato, e che in alcuni casi veniva impostato a zero anziché ad un valore consistente. Grazie a debug_flags abbiamo così ricostruito il percorso di scelta seguito all'interno della funzione g_rr_next(), scoprendo che il problema poteva sorgere (e comunque non in ogni caso) solo a seguito della selezione di una nuova coda da servire dato che anche q_flags era impostato a zero, cosa che avviene solo nel caso menzionato.

Nel codice, l'aggiornamento delle deadline avviene al termine dell'esecuzione della prima richiesta relativa al processo che è appena stato selezionato, all'interno della funzione gs_rr_done(), e non appena si cambia la coda. Questo perché si vuol evitare di sommare ai tempi di esecuzione anche quello necessario ad effettuare la prima seek, che non dipende direttamente dal processo scelto. La ricerca dell'errore funzionale dello scheduler si è quindi ristretta a tale funzione,

in cui l'aggiornamento veniva effettuato solo nel caso in cui la coda attiva risultasse nello stato BUSY e il cui flag di stato (q_flag) fosse ovviamente impostato a zero.

Analizzando il codice si è notato che, nel caso particolare di una coda formata da una sola richiesta a cui se ne fosse aggiunta un'altra subito dopo esser stata scelta per poter accedere al disco, lo stato della coda veniva impostato come READY rendendo così impossibile il corretto aggiornamento dei parametri previsto solo per le code nello stato BUSY.

L'impostazione dello stato della coda come READY avviene infatti congiuntamente con l'arrivo della nuova richiesta di I/O, dentro la funzione g_rr_start() di seguito riportata, che si occupa principalmente di inserire le varie richieste all'interno delle giuste code, per riordinarle poi utilizzando disksort.

```
static int
g_rr_start(void *data, struct bio *bp)
{
        struct g_rr_softc *sc = data;
        struct g_rr_queue *qp;

        if (me.bypass)
                return (-1);     /* bypass the scheduler */

        /* Get the queue for the request. */
        qp = g_rr_queue_get(sc, bp);
        if (qp == NULL)
                return (-1); /* allocation failed, tell upstream */

        if (gs_bioq_first(&qp->q_bioq) == NULL) {
                /*
                 * We are inserting into an empty queue.
                 * Reset its state if it is sc_active,
                 * otherwise insert it in the RR list.
                 */
                if (qp == sc->sc_active) {
                        qp->q_status = G_QUEUE_READY;
                        callout_stop(&sc->sc_wait);
                } else {
                        g_sched_priv_ref(qp);
                        TAILQ_INSERT_TAIL(&sc->sc_rr_tailq, qp, q_tailq);
                }
        }
```

```
        qp->q_bionum = 1 + qp->q_bionum - (qp->q_bionum >> 3);

        g_rr_update_thinktime(qp);
        g_rr_update_seekdist(qp, bp);

        /* Inherit the reference returned by g_rr_queue_get(). */
        bp->bio_caller1 = qp;
        gs_bioq_disksort(&qp->q_bioq, bp);

        TRC_QUEUE_EVENT(SCHEDSTAT_ST, qp);

        return (0);
}
```

La riga di codice che si occupa dell'aggiornamento dello stato è riportata in grassetto e viene eseguita appunto solo se la richiesta in arrivo va inserita nella stessa coda che è attualmente in esecuzione.

Va sottolineato che questo particolare tipo di situazione non è mai stata ottenuta volutamente, ma si è comunque verificata casualmente nella maggior parte delle prove funzionali effettuate, essendo forse stata generata durante l'esecuzione di qualche processo di sistema avviato direttamente dal kernel.

In ogni modo si è resa comunque necessaria una modifica ai controlli sulla coda che supportasse anche questo tipo di situazione, per far sì che lo scheduler si comportasse come previsto.

5.5 Modifica della funzione g_rr_done()

Per risolvere il problema, si è dovuto quindi spostare un controllo sullo stato della coda all'interno della funzione g_rr_done() in maniera che l'aggiornamento dei parametri avvenisse indipendentemente da esso, verificando esclusivamente il valore di q_flag che come detto indica un cambiamento della coda di processi in esecuzione.

Di seguito è riportato uno spezzone del codice originale della funzione g_rr_done(), contenuta come detto in precedenza nel file gs_rr.c, in cui avviene l'aggiornamento del budget temporale:

```
static void
g_rr_done(void *data, struct bio *bp){
        ...
        qp = bp->bio_caller1;
        if (qp == sc->sc_active && qp->q_status == G_QUEUE_BUSY) {
                if (!(qp->q_flags & G_FLAG_COMPLETED)) {
                        qp->q_flags |= G_FLAG_COMPLETED;
                        /* in case we want to make the slice adaptive */
                        qp->q_slice_duration = get_bounded(&me.quantum_ms, 2);
                        qp->q_slice_end = ticks + qp->q_slice_duration;
                }
                /* The queue is trying anticipation, start the timer. */
                qp->q_status = G_QUEUE_IDLING;
        ...
```

Il controllo in questione è mostrato in grassetto. A seguito del suo spostamento, il codice è invece diventato il seguente:

```
static void
g_rr_done(void *data, struct bio *bp){
        ...
        qp = bp->bio_caller1;
        if (!(qp->q_flags & G_FLAG_COMPLETED)) {
                qp->q_flags |= G_FLAG_COMPLETED;
                /* in case we want to make the slice adaptive. */
          qp->q_slice_duration = get_bounded(&me.quantum_ms, 2);
          qp->q_slice_end = ticks + qp->q_slice_duration;
        }
        if (qp == sc->sc_active && qp->q_status == G_QUEUE_BUSY) {
                /* The queue is trying anticipation, start the timer. */
                qp->q_status = G_QUEUE_IDLING;
        ...
```

Effettuando delle nuove prove con le stesse modalità di quelle descritte, il problema ha definitivamente smesso di presentarsi, rendendo quindi il comportamento dello scheduler in linea con quello previsto.

5.6 Verifica eccezione mediante I/O asincrono

Per poter fugare ogni dubbio, si è comunque deciso di procedere ricreando volutamente l'eccezione che portava alla luce l'anomalia di funzionamento dello scheduler, per applicarla alle due versioni di g_rr_done() e poter affermare quindi con sicurezza che il problema riscontrato era dipendente da quello.

Prima di analizzare come è stato effettuato questo test, è necessario fare una premessa: ricreare la situazione che portava al malfunzionamento dello

scheduler, ovvero di una coda formata da una sola richiesta a cui se ne aggiunge un'altra prima del termine della sua esecuzione, non è affatto immediato a livello utente poiché esse tendono ad essere sincrone, e quindi non viene inviata nessuna nuova richiesta finché quella attuale non è stata completata.

Per poter quindi generare I/O in maniera asincrona, si è dovuti ricorrere all'uso di *aio_read()*, che è una funzione C che effettua letture asincronamente manipolando una particolare struttura dati chiamata *aiocb* definita nel file di libreria *aio.h*. Il programma che ne è risultato (contenuto nel file aioread_p1.c) è riportato di seguito:

```
#include <fcntl.h>
#include <errno.h>
#include <aio.h>
#include <strings.h>

int main(){

        /*Definisco le variabili e le strutture *
         *dati che andrò ad utilizzare                */
        char buf[4096];
        ssize_t retval;
        ssize_t nbytes;
        struct aiocb myaiocb;
        /*Azzero myaiocb*/
        bzero(&myaiocb, sizeof(struct aiocb));
        /*Associo al descrittore di aiocb il mio       *
         *disco USB, aprendolo in sola lettura        */
        myaiocb.aio_fildes = open ("/dev/da0", O_RDONLY);
        myaiocb.aio_offset = 0;
        myaiocb.aio_buf = (void *) buf;
        myaiocb.aio_nbytes = sizeof(buf);
        myaiocb.aio_sigevent.sigev_notify = SIGEV_NONE;

        /*Effettuo adesso due letture consecutive*/
        retval = aio_read(&myaiocb);
        retval = aio_read(&myaiocb);

        /*Verifico l'esito positivo delle letture */
        if(retval)
                perror("aio_read:");

        /*Attendo il termine delle operazioni prima di uscire */
        while((retval = aio_error(&myaiocb)) == EINPROGRESS);
        nbytes = aio_return(&myaiocb);
}
```

A questo punto il programma è stato compilato generando il file eseguibile *aioread_p1* e, dopo aver caricato il modulo del kernel *aio.ko* necessario alla sua corretta esecuzione, è stato avviato con entrambe le versioni dello scheduler, in abbinamento allo script ktr_trace.sh riportato nel capitolo 5.2 con il comando

 # ./ktr_trace.sh ./aioread_p1

I risultati ottenuti sono riportati nelle due tracce seguenti. La prima è relativa alla vecchia versione dello scheduler:

index	*timestamp*	*trace*						
0	*1807104428910*	*S 100099*	*R*	*1054179/0*	*4096*			
1	*1807104887130*	*N 100099*	*R*	*1054179/0*	*4096*			
2	*1807105191640*	*P 100099*	*R*	*1054179/0*	*4096*			
3	*1807105498810*	*S 100099*	*R*	*1054179/0*	*4096*			
4	*1807106089250*	*D 100099*	*R*	*1054180/0*	*4096*			
5	*1807106410690*	*N 100099*	*R*	*1054180/0*	*4096*			
6	*1807106715160*	*I1 1 8192*	***0***	*0*	*11000*			
7	*1807106986340*	*I2 8388608*	*100*	*10*	*0*	*0*		
8	*1807107268580*	*I3 0 0*	*2*	*1054180*	*0*			
9	*1807107545750*	*I4 -8380416*	***1054181***	*0*	*1*	*0*	*0*	
10	*1807107864790*	*P 100099*	*R*	*1054181/0*	*4096*			
11	*1807108407310*	*D 100099*	*R*	*1054181/0*	*4096*			

Mentre per quanto riguarda la versione modificata il risultato è stato questo:

index	*timestamp*	*trace*						
0	*566427527904*	*S 100060*	*R*	*319713/0*	*4096*			
1	*566428247661*	*N 100060*	*R*	*319713/0*	*4096*			
2	*566428719345*	*P 100060*	*R*	*319714/0*	*4096*			
3	*566429192793*	*S 100060*	*R*	*319714/0*	*4096*			
4	*566430472809*	*D 100060*	*R*	*319715/0*	*4096*			
5	*566431327770*	*N 100060*	*R*	*319715/0*	*4096*			
6	*566431799337*	*I11 8192*	***319815***		*1*	*11000*		
7	*566432267844*	*I2 8388608*	*100*	*10*	*0*	*0*		
8	*566432713329*	*I3 0 0*	*2*	*319714*		*0*		
9	*566433123177*	*I4 -8380416*	***-99***	*1*	*1*	*1*	*0*	
10	*566433581796*	*P 100060*	*R*	*319717/0*	*4096*			
11	*566434866132*	*D 100060*	*R*	*319718/0*	*4096*			

In entrambi i casi siamo riusciti ad ottenere il risultato voluto visto che la START relativa alla seconda richiesta di I/O arriva prima del completamento della precedente. Inoltre andando ad analizzare i valori in grassetto presenti in

ambedue le tracce, che rappresentano rispettivamente il tick di fine slice e quelli disponibili rimanenti, si vede chiaramente che nel primo caso l'aggiornamento non avviene correttamente ma q_slice_end viene posto a zero, mentre nel secondo tutto funziona come dovrebbe, rendendo disponibili al nostro processo di lettura altri 99ms dal termine della prima istanza.

A questo punto è quindi possibile affermare con sicurezza che a seguito della modifica apportata, il problema rilevato nel funzionamento dello scheduler è stato risolto, e che adesso esso è effettivamente in grado di offrire il servizio previsto.

Capitolo 6.

Conclusioni

A seguito dei risultati ottenuti eseguendo sia i test prestazionali sia quelli funzionali descritti, si può affermare che l'utilizzo dello scheduler Round Robin, realizzato in GEOM mediante l'ausilio del modulo geom_sched, porta in generale dei sensibili benefici all'utente che andrà ad utilizzarlo, sia perché garantisce comunque delle ottime performance se paragonato con l'algoritmo di scheduling C-LOOK implementato in FreeBSD, ma soprattutto perché permette di offrire un servizio molto più equo a tutti gli utenti che hanno necessità di andare ad operare sul disco.

Per quanto riguarda le prestazioni si è visto che, se l'accesso deve essere garantito ad un solo processo che effettua letture o scritture sequenziali, questi vantaggi non vengono a galla (anche se le prestazioni restano comunque in linea con quelle senza scheduler), ma effettuando invece accessi multipli al disco, cosa che può rappresentare ad esempio una situazione in cui più utenti hanno necessità di accedervi contemporaneamente ed effettuare elaborazioni di diversa natura, i benefici introdotti saranno notevoli giustificando di fatto un irrisorio aumento della complessità di uso da parte di tali utenti che dovranno semplicemente provvedere a caricare i relativi moduli.

Per quanto riguarda invece l'effettivo funzionamento dello scheduler si può tranquillamente affermare che, a seguito del problema riscontrato e della modifica effettuata per risolverlo, il suo comportamento rispecchia pienamente quelle che sono le aspettative.

L'utilizzo di uno scheduler di questo tipo rappresenta quindi un notevole passo in avanti in quella che è la gestione del disco in un sistema FreeBSD, garantendo un servizio estremamente equo e fruibile.

Bibliografia

Oltre alle fonti riportate nelle note sono state consultate anche le seguenti:

Manuale, pagine man e altre informazioni su FreeBSD:
http://www.freebsd.org

Presentazione BSDCAN 2009 sugli scheduler GEOM:
http://www.bsdcan.org/2009/schedule/attachments/100_gsched.pdf

Tutorial GEOM:
http://phk.freebsd.dk/pubs/bsdcon-03.slides.geom-tutorial.pdf

Guide AWK, scripting BASH:
http://www.gnu.org/

Manuale di Subversion:
http://svnbook.red-bean.com/

Informazioni su Virtualbox:
http://www.virtualbox.org/

Per informazioni di carattere generale:
http://it.wikipedia.org/