Scheduling in minix3

Gabriele Modena, mat. 108742, gabriele.modena@gmail.com

June 11, 2006

Contents

Abstract

Questo progetto mi ha portato ad approfondire il funzionamento dello scheduler di minix3 e la gestione delle syscall. In particolare, ho modificato la modalitá con cui i processi utente vengono consegnati al dispatcher, imponendo un criterio di selezione basato sulla quantitá di tempo di cpu usato. Questo mi ha portato a implementare un meccanismo di inserimento nelle code di scheduling basato su prioritá.

Parallelamente ho aggiunto una nuova system call, quantum(), che consente di cambiare (aumentare o diminuire) a run time la dimensione del quanto temporale associata ad un processo.

Chapter 1

Funzionamento dello scheduler

1.1 Introduzione

Lo scheduler di minix3 si basa su una struttura a code multilivello, 16 nell'implementazione attuale, in cui i processi possono spostarsi dinamicamente una coda all'altra (possono cioé modificare la loro prioritá).

La natura a microkernel del sistema porta ad un distinguo tra quattro classi di processi (in ordine di prioritá):

- **task**
- **driver**
- **server** (o *servizi di sistema*)
- **user**.

Appartengono alla famiglia server i processi di sistema (gestione dei processi, log del sistema, is, file system...) che vengono eseguiti in user space. I device driver (tty0, schede di rete etc.) fanno parte della categoria driver e vengono eseguiti con una prioritá maggiore rispetto ai server. I task SYSTEM, CLOCK e IDLE sono invece eseguiti in kernel space e costituiscono il cuore del sistema minix3. Tutto il resto rientra nella categoria user.
La prioritá di un processo e determinata dalla coda in cui lo stesso viene posizionato: 0 indicata la prioritá massima, 15 la minima. I *task* CLOCK e SYSTEM vengono esegeuiti unicamente con prioritá 0, IDLE occupa costantemente la coda 15 ed esegue quando non sono presenti processi ready. I processi *di sistema* e i *driver* si posizionano nelle code 0-6. I processi *user* si vedono assegnata la coda 7, con la possibilitá di salire a 0 e scendere fino a 14. La tabella dei processi con le prioritá predefinite al boot del sistema si puotrovare nel file *kernel/table.c*.

All'interno di ogni coda i processi sono gestiti tramite una versione modificata di **round robin**. Ogni processo ha a disposizione un quanto temporale, durante il quale gli viene assegnata la cpu. Allo scadere del quanto temporale, il kernel effettua una preemption passando il controllo del processore ad un nuovo processo.

I nuovi processi vengono accodati secondo uno schema FCFS.

Il quanto varia in base al tipo di processo e rimane costante per tutta la durata di esecuzione, anche in caso di modifica della prioritá.

Il server **clock** genera, ad intervalli regolari, degli impulsi hardware che, tradotti messaggi software, vengono utilizzati per scandire il tempo di vita del processo in eseguzione. Vengono cioé incrementati il suo tempo di esecuzione (utente e di sistema) e viene decrementato il valore della variable **p_ticks_left**, che indica la porzione di quanto rimasta. Allo scadere del tempo, il sistema provvede ad eseguire un nuovo ciclo di scheduling. Un processo running puó essere bloccato dal sistema (ad esempio per la notifica di un segnale). Nel momento in cui il processo diventa di nuovo ready, lo scheduler determina la sua posizione nella coda in base al quanto temporale rimasto. Se il processo ha ancora tempo da spendere viene messo in testa alla coda, altrimenti in fondo. Parallelamente, il sistema verifica che il processo non sia entrato in un ciclo abbassando (incrementando p_priority) la sua prioritá in caso affermativo o aumentandola (decrementando p_priority) nel caso in cui la sua esecuzione non abbia impedito ad altri processi di ottenere la cpu.

Tipicamente ai processi task, driver e server é associato un quanto maggiore rispetto a quello dei processi utente ed é loro consetita l'esecuzione sequenziale fino al termine dell'operazione. Nel caso in cui il tempo di esecuzione sia troppo alto, viene effettuata preemption per evitare cicli infiniti.

1.2 Struttura del codice

La parte di scheduling da me analizzata viene gestita interamente a livello kernel. I file interessati si trovano nella directory */usr/src/kernel*

1.2.1 glo.h

Questo header file contiene definizioni di variabli e strutture dati utilizzate globalmente nel codice del livello kernel.

In particolare vengono dichiarati:

- **proc_ptr** puntatore al processo attualmente in esecuzione.

- **next_ptr** puntatore al prossimo processo che verrá eseguito
- **prev_ptr** puntatore al processo eseguito in precedenza
- **bill_ptr** puntatore al processo a cui si dovranno accreditare ticks di cpu.

1.2.2 proc.h

In questo header si trovano le dichiarazioni della struttura **proc**, che viene usata per rappresentare l'istanza di un processo in memoria. All'interno di proc vengono definite, tra le altre:

- il numero del processo, *proc_nr*, a livello kernel. Tale identificativo non corrisponde al PID, che in minix3 é un'informazione fornita dal server PM.
- la prioritá attuale del processo, *p_priority*
- la prioritá massima ottenibile dal processo, *p_max_priority*
- informazioni sulla natura del processo: utente (USR_T), task o (TSK_T) server (SRV_T), preemptive etc.)
- tempo di vita, *p_user_time* e *p_sys_time*
- dimensione del quanto e tempo rimasto *p_quantum_size*, *p_ticks_left*
- nome del processo *p_name*

In proc.h sono definite anche le infromazioni relative alle code di scheduling:

- numero di code presenti, *NR_SCHED_QUEUES*
- coda con la massima prioritá *TASK_Q*
- coda predefinita per i processi utente *USER_Q*, massima *MAX_USER_Q* e minima *MIN_USER_Q* coda raggiungibile
- coda con la minima prioritá, *IDLE*. Tale coda è riservata unicamente al task IDLE che viene eseguito quando nessun altro processo richiede la cpu.

Una coda di prioritá è costituita da un'array di puntatori a strutture *proc* indicizzate dai puntatori rdy_head[Q] e rdy_tail[Q] che rappresentano rispettivamente la cima e il fondo della coda Q.

1.2.3 proc.c

Questo file contiene le funzioni per la gestione dei processi.

- *sched()* detta la politica di scheduling per un processo. In particolare determina la coda (prioritá) da assegnare e la sua posizione nella stessa (capo o fondo).
- *enqueue()* inserisce un processo in una coda
- *dequeue()* rimuove un processo da una coda
- *pick_proc()* svolge la funzione di dispatcher. Un ciclo scorre tutte le code assegnando la cpu al primo processo pronto ad eseguire, posizionato in testa (rdy_head) alla prima coda non vuota trovata. Una volta trovato un processo utile, pick_proc() aggiorna i riferimenti di next_ptr e bill_ptr. Se non ci sono processi pronti ad eseguire, il controllo della cpu passa al task IDLE (coda 15).

1.2.4 clock.c

Questo file contiene il codice relativo al task CLOCK. CLOCK si occupa, tra l'altro, di aggiornare il tempo di esecuzione, in ticks, del processo (p_user_time) e la porzione di quanto rimasta (p_ticks_left). Se il processo ha la flag BILLABLE attiva, come tipicamente avviene per i processi utente, il sistema provvede a far pagare un costo aggiuntivo di tempo di sistema (p_sys_time) per compensare l'overhead dovuto allo scheduling.

Chapter 2

Modifica dello scheduler

2.1 Politica

Lo scopo del mio lavoro é stato modificare lo scheduler per fare in modo che, nel momento in cui nessun task o server sia pronto ad eseguire, il sistema selezioni il processo utente che recentemente ha usato la cpu per meno tempo. É stato necessario implementare un meccanismo di invecchiamento dei processi in coda per evitare starvation.

2.2 Meccanismo

La struttura a code multilivello é stata modificata, isolando i processi utente nella sola coda 7 (USER_Q).

In *kernel/glo.h* ho dichiarato due nuovi array con lo scopo di tenere traccia del tempo di esecuzione del processo e di fornire un indice dinamico di prioritá:

prio_estimate[NR_PROCS] e procs_lifetime[NR_PROCS];

L'etichetta *NR_PROCS*, definita in *minix/com.h*, indica il numero massimo di processi utente consentiti dal sistema.

Questi due array vengono inizializzati al boot del sistema (*kernel/main.c*) e alla creazione di un nuovo processo tramite fork(*kernel/system/do_fork.c*).

procs_lifetime svolge il ruolo di cronometro: ad ogni interrupt viene incrementato il valore nella posizione associata al processo attualmente in esecuzione (equivale a calcolare lo user time). Al momento di inserire un nuovo processo in coda, si aggiorna la stima di prioritá sommando il tempo di vita del processo e si verifica che sia trascorso un tempo minimo di attesa. In caso affermativo il cronometro viene azzerato e le stime di prioritá definite

da prio_estimate decrementate. Questo evita che un processo abbia un tempo di vita tendente a infinito e rimanga bloccato in coda.

A questo punto, il processo viene inserito nella coda utente rispettando l'ordinamento (crescente) basato sulla prioritá. Un processo con valore piú basso ha eseguito per meno tempo e si vede associata una prioritá piú alta.

Il codice che gestisce questo meccanismo é contenuto in:

- **kernel/clock.c** - *clock_handler()*: il task CLOCK incrementa il cronometro.
- **kernel/proc.c** - *enqueue()*: gestisce l'inserimento del processo in coda in base alla politica di scheduling opportuna. Si vedano i commenti al codice per effettuare un test sulla starvation dei processi.

2.3 Visualizzazione

Premendo il tasto F9 viene effettuato un dump delle code di scheduling e di prio_estimate sul terminale tty0. Alla pressione del tasto, il server is (information server) esegue la funzione *sched_dmp()* definita in *is/dmp_kernel.c* Come tutte le funzioni implementate in questo file, *sched_dmp()* esegue il dump di strutture dati definite in kernel space stampando l'output su terminale.

Per poter accedere a prio_estimate é stato necessario intervenire sulla system call do_getinfo del task SYSTEM e sugli header della libreria di sistema ad essa associata. In particolare:

- aggiungere l'etichetta *SYS_PRIOINFO* all'header *minix/com.h*
- aggiungere la macro *do_priogetinfo* all'header *minix/syslib.h*. Questa macro fa da wrapper alla funzione di libreria sys_getinfo, richiamandola con il segnale opportuno
- modificare *system/do_getinfo()* per gestire il segnale e inviare una copia di prio_estimate in user space.
- modificare la funzione *dmp_sched()*.

Si vedano i commenti al codice per i dettagli sul funzionamento di do_getinfo.

In seguito alle modifiche, is é in grado di visualizzare, per ogni processo in coda 7, nome, numero e prioritá.

2.4 Considerazioni

Inzialmente era mia intenzione usare un timer per notificare SYSTEM e CLOCK dello scadere dell'intervallo di tempo. L'uso dei timer é peró risultato in sporadiche corruzioni degli array, in situazioni che non sono riuscito a riprodurre. Questa situazione mi ha portato a cercare una soluzione alternativa, meno elegante ma che dai test che ho effettuato si é rivelata piú robusta. Ho adottato un meccanismo analogo a quello usato dal task CLOCK per decretare la preemption del processo che sta usando la cpu. Parallelamente all'incremento di procs_lifetime, decremento un contatore time_elapsed. Nella funzione enqueue(), al momento di stimare la prioritá del processo entrante verifico lo stato di time_elapsed. Se il contatore é arrivato a zero o meno, faccio ripartire il cronometro azzerando procs_lifetime di tutti i processi e riduco le loro prioritá di $\frac{1}{4}$. L'uso di due array con informazioni ridondanti si eŕeso necessario perché lavorare direttamente su procs_lifetime ha portato a delle incongruenze nel calcolo prioritá. Dati i ridottissimi tempi di esecuzione, capitava che al momento di inserire in coda un nuovo processo non fosse piú quello da me atteso.

Per verificare il funzionamento delle modifiche ho usato la suite di test presente in */usr/src/test*. Per stressare il sistema ho eseguito due test in parallelo, con il risultato di far fallire test5 (file buffer overflow), mandandolo in un ciclo di durata indefinita . Questo esperimento ha rivelato che il cambio dinamico di prioritá ha permesso di evitare il blocco del processo in coda con conseguente starvation.

Chapter 3

Struttura delle system call

3.1 syscall in minix3

L'interazione tra i processi utente, server e kernel (e i driver) avviene tramite scambio di messaggi. In un sistema monolitico, il termine *system call* si riferisce a tutte le funzioni che consentono di accedere a servizi forniti dal kernel.

In minix3 le chiamate di processi utente vengono trasformate in messaggi a processi server, che a loro volta comunicano tra loro e con i driver e il kernel sempre tramite messaggi.

Il sistema prevede un insieme di chiamate a livello kernel implementate dal task SYSTEM (kernel/system). Per ognuna di queste chiamate, esiste un corrispettivo in user space implementato nella libreria *syslib* (minix/com.h, minix/syslib.h e /usr/src/lib/syslib) e accessibile dai processi server. Ad ogni chiamata eássociato un id in forma SYS_CALLNAME e il kernel, al boot, crea una tabelle di associazioni (*kernel/system.c*) tra funzioni di libreria e chiamate kernel. Queste funzioni non possono essere richiamate da utente e prendono il nome di *kernel calls*.

Ad un livello superiore si trovano le chiamate relative ai servizi di sistema e le chiamate POSIX, che si appogiano ai due strati sottostanti. Queste funzioni formano l'insieme delle *system call* fornite all'utente.

In generale, una chiamata ad una system call in minix3 risulta nella chiamata al suo analogo in kernel space.

3.2 Aggiunta di una nuova system call: quantum()

La seconda parte del progetto eśtata aggiungere il comando qnt che consente di modificare la dimensione del quanto temporale di un processo.

I passi necessari sono stati:

- Aggiungere la funzione *sys_quantum()* alla libreria di sistema (*syslib*) e modificare il vettore delle chiamate affinché la richiesta sia inviata correttamente al task SYSTEM. Questa funzione non viene richiamata direttamente dall'utente, ma dal server PM (*Process Manager*).

- Aggiungere la kernel call *do_quantum()* che, una volta ricevuto un messaggio da *sys_quantum()* modifichi opportunamente le strutture dati dei processi. Possiamo vedere questa procedura come l'implementazione in kernel space di *sys_quantum()*.

- Aggiungere la system call *quantum()*, che invia un messaggio con la richiesta *QUANTUM* al server PM. Quest'ultimo, associa la chiamata alla funzione do_quantum() che a sua volta richiama *sys_quantum()* con i parametri opportuni.

- Scrivere un comando qnt che richiama la funzione *quantum()* prendendo in input pid del processo da modificare e dimensione del quanto. *qnt* é stato aggiunto ai comandi di sistema presenti in */usr/src/commands/simple* e viene installato in */usr/bin*.

3.3 Modifiche agli header

Per implementare la chiamata é stato necessario effettuare delle modifiche agli header presenti in */usr/include/minix*.

3.3.1 minix/com.h

Qui sono definite delle etichette usate globalmente da varie parti del sistema. Le modifiche apportate sono:

- definita *PR_QUANTUM*: fa da alias al campo m1_i2 (int) della struttura dati usata per inviare messaggi tra processi. Rappresenta la dimensione del quanto.

- definite *MAX_QUANTUM_SIZE* e *MIN_QUANTUM_SIZE*: dimensione massima e minima (in ticks) assegnabile al quanto.
- aggiunta una kernel call *SYS_QUANTUM* e incrementato il valore di *NR_SYS_CALL*.

3.3.2 minix/callnr.h

Questo file contiene una lista di system call richiamabili da utente. Ho aggiunto la chiamata numero 91, *QUANTUM*, e incrementato il numero di *NR_CALLS*

3.3.3 minix/syslib.h

Dichiarato il prototipo della funzione *sys_quantum()*

3.4 do_quantum()

É la kernel call vera e propria, dichiarata in *kernel/system.h*.

Viene eseguita a livello kernel e pertanto ha accesso diretto a tutte le strutture dati presenti in memoria. In particolare, puoó modificare i singoli elementi presenti nella tabella dei processi.

Come detto prima, in minix3 le system call sono implementate via message passing.

do_quantum() riceve da sys_quantum() e procede ad estrarre i campi PR_PROC_NUM (numero del processo in kernel space) e PR_QUANTUM.

A questo punto vengono effettuati dei test sulla validitá del numero di processo, in particolare viene verificato che il processo selezionato sia in user space (non task).

Quindi viene verificato che la dimensione del quanto rispetti i limiti imposti a compile time, in caso contrario il valore viene ridimensionato e la funzione stampa un messaggio di errore su tty0.

A questo punto il processo da modificare viene selezionato grazie alla macro *proc_addr()* (*kernel/proc.c*). Una volta rimosso dalla coda, il campo *p_quantum_size* viene modificato, e il processo reinserito in coda.

In ultimo, ho aggiunto la label *USE_QUANTUM* all'header *kernel/config.h* in modo da poter decidere o meno l'inclusione delle chiamata a compile time.

3.5 sys_quantum()

```
int sys_quantum(int proc, int qnt)
```

La funzione di libreria é implementata in */usr/src/lib/syslib/sys_quantum.c* ed é richiamabile in user space.

Il suo scopo edi creare un mesaggio contente i campi PR_PROC_NUM e PR_QUANTUM impostati, con i valori proc e qnt passati, e inviarlo al task SYSTEM tramite la funzione *_taskcall()* (*/usr/src/lib/sysutil/*) notificando la richiesta di esecuzione della chiamata *SYS_QUANTUM*.

Il task SYSTEM mantiene un vettore delle chiamate (*calls vector*), inzializzato a boot time dalla funzione *initialize* (*kernel/system.c*). Lo scopo di tale vettore é associare le chiamate dichiarate in (*minix/com.h*) alla loro implementazione in kernel space. Ad esempio, la chiamata *SYS_QUANTUM* viene associata alla funzione *do_quantum()* tramite:

```
map(SYS_QUANTUM, do_quantum)
```

3.6 quantum()

La struttura a livelli di minix3 non consente ad un processo di dialogare direttamente con il task SYSTEM (kernel space). In particolare, la gestione dei processi é demanata al servizio PM.

Per poter accedere alla funzione sys_quantum é stato necessario modificare tale servizio aggiungendo una nuova system call: *QUANTUM* (91).

L'implementazione della system call si trova nel file */usr/src/lib/other/quantum.c* e la sua dichiarazione nell'header *sys/resources.h*:

```
int quantum(int pid, int qnt)
```

La funzione prende in input il PID del processo da modificare e la dimensione del quanto, crea un messaggio aggiornando i campi opportuni e lo invia al server PM notificando la system call QUANTUM.

A questo punto, é stato necessario modificare il server PM aggiungendo la nuova system call al *call_vector* (*table.c*) e associandola alla funzione do_quantum().

Questa funzione é implementata nel file *misc.c*.

Una volta ricevuto un messaggio da un processo utente, verifica che l'utente che ha effettuato la richiesta sia **root**, estrae il campo PR_PID e PR_QUANTUM, converte il pid selezionato nel corrispettivo numero di processo a livello kernel e invoca sys_quantum() con i parametri opportuni.

A questo punto, viene stampato un messaggio di debug su tty0 che riporta il numero del processo modificato.

Chapter 4

Approccio e metodo di lavoro

Ho dedicato la prima settimana di lavoro allo studio della documentazione sul funzionamento del sistema e ha prendere confidenza con gli strumenti.

Per prima ho affrontato la parte sullo scheduler, qui ho avuto alcuni problemi dovuti alla difficoltá nel fare debug.

Per aiutarmi in questo senso sono ricorso all'inserimento di kprintf nel codice, un metodo rozzo ma utile in mancanza di un debugger in kernel space. Di fondamentale importanza é stato il servizio IS, che consente di avere una vista immediata e in tempo reale su tutte le strutture dati del kernel.

In particolare, al posto degli array prio_estimate e procs_lifetime avevo modificato la struttura **proc** per tenere traccia delle informazioni. Questo, solo in alcuni casi non sempre riproducibili, é risultato in immagini corrotte. Le difficoltá nel tracciare questo tipo di errori mi hanno fatto preferire una soluzione basata sugli array esterni, lo spreco di memoria é maggiore ma la comoditá nella gestione e nel debug é sensibilmente aumentata.

Per quanto riguarda l'aggiunta della system call quantum(), ho analizzato il funzionamento del comando nice e da li ho individuato le porzioni di codice su cui intervenire per ottenere il risultato desiderato.