Lezione Kernel - Il Kernel Linux

31 gennaio 2007

Alessio Checcucci

Riccardo Aldinucci

Q.It Universita' degli Studi di Siena

1 Introduzione

Linux fa parte della famiglia dei sistemi operativi Unix. Quelli che derivano da System V release 4 (SVR4) sviluppato da AT&T e da 4.4 BSD distribuito da University of California at Berkeley, oltre ad un folto gruppo di Unix commerciali.

Linux e' stato inizialmente sviluppato da Linus Torvalds nel 1991 come un sistema operativo per macchine IBM-compatibili (i80386). L'attivita' di Linus al momento, oltre allo sviluppo, e' grandemente orientata al coordinamento delle centinaia di programmatori che conducono lo sviluppo dei vari rami del sistema. Negli anni le piattaforme supportate si sono estese fino ad includere HP Alpha, Intel Itanium, Amd AMD64, PowerPC, Sun Sparc, IBM zSeries, etc.

Da un pusto di vista tecnico, Linux e' uno Unix kernel, sebbene non possa essere definito un Sistema Operativo Unix, perche' non fornisce nessuna delle classiche applicazioni di Unix. In questo senso e' meglio parlare di GNU/Linux, ovvero il kernel affiancato da tutte le applicazioni fornite dalla GNU sotto l'egida della licenza GPL.

1.1 Linux kernel e gli altri Unix

I vari sistemi Unix-like presenti sul mercato derivano in qualche maniera o da System V release 4 oppure da 4.4BSD. Ogni Unix ha poi sviluppato caratteristiche peculiari che lo differenziano dagli altri. In ogni caso alcune caratteristiche comuni possono essere riscontrate, esse sono degli standard conosciuti come IEEE Portable Operating Systems (POSIX) e X/Open's Common Applications Environment (CAE). Questi standard riguardano l'API del kernel e il runtime environment, per cui non pongono limiti al progetto interno del kernel.

In relazione ad altri Unix kernel, Linux presenta le seguenti caratteristiche:

- Kernel Monilitico

E' un programma grande e complesso che svolge una grande serie di compiti, anche se composto da una serie di moduli logici distinti. La maggior parte degli Unix sono monolitici, mentre Mac OS X e GNU/Hurd sono sviluppati a microkernel derivato da Carnegie-Mellon Mach.

- Kernel compilato e linkato staticamente

La maggior parte dei kernel Unix sono in grado di caricare e togliere parti del kernel a runtime. Queste parti (in genere device drivers) sono comunemente chiamate moduli. Linux e' pero' uno dei pochi Unix capace di caricare e scaricare i moduli in maniera automatica quando e' necessario.

- Kernel Threading

Alcuni Unix sono organizzati a thread. Un thread non e' altro che un contesto di esecuzione che puo' essere schedulato indipendentemente dagli altri. Esso puo' essere associato ad un programma utente o ad alcune funzioni interne del kernel. Un context switch fra thread e' molto meno gravoso di quello fra i comuni processi, perche' i thread operano in uno spazio di indirizzamento comune. Linux fa un uso dei thread piuttosto limitato.

- Supporto per applicazioni Multithread

La maggior parte dei sistemi operativi moderni presenta il supporto per applicazioni multithread, ovvero programmi utente che sono organizzati in una serie di percorsi di esecuzione relativamente indipendenti e che condividono una gran parte delle strutture dati. L'applicazione in questo senso puo' essere vista come composta da una serie di lightweight processes (LWP) , ovvero processi che operano su strutture dati comuni. Linux definisce la propria variante dei LWP e li gestisce per mezzo della chiamata di sistema (non-standard) clone().

- Supporto Multiprocessore

Quasi tutte le varianti di Unix presentano il supporto multiprocessore. Linux 2.6 presenta il supporto Symmetric Multiprocessing (SMP) per una serie di modelli di memoria (incluse le architetture NUMA). Il sistema puo' usare piu' processori e ognuno di essi puo' eseguire qualunque task senza discriminazioni. Sebbene alcune parti del kernel debbano essere eseguite in maniera seriale e sia presente ancora il "Big Kernel Lock", l'uso dell'SMP in Linux e' quasi ottimale.

- Filesystem

I filesystem standard in Linux sono una notevole quantita'. Nativamente ci sono i filesystem Ext2/ext3. Altre scelte possono essere ReiserFS, IBM AIX JFS, SGI Irix XFS. Il porting di un filesystem verso Linux e' piuttosto agevole grazie all'architettura Object-oriented del Virtual Filesystem Switch (VFS).

- Stream

Linux non ha nessun corrispondente al sottosistema di STREAM I/O presente in altri Unix. Sebbene questa sia l'interfaccia preferenziale, anche se non la piu' efficiente, per scrivere device drivers, protocolli di rete e terminal drivers.

Oltre a questo va notato che la vitalita' di Linux e le molte altre caratteristiche che presenta, lo rendono competitivo, se non spesso superiore alla maggior parte degli Unix commerciali.

1.2 Dipendenza dall'hardware

I sorgenti di Linux mantengono una netta distinzione fra le parti hardware-dependent e quelle hardware-indipendent. Le directory arch e include hanno sotto di se 23 sub-directory relative ai vari tipi di hardware supportati:

- alpha
- arm, arm26
- cris
- frv
- h8300
- i386

- ia64
- m32r
- m68k
- mips
- parisc
- ppc, ppc64
- s390
- sh, sh64
- sparc, sparc64
- um
- v850
- x86_64

1.3 Versioni Linux[1]

Fino al kernel 2.5 l'identificazione dei kernel Linux avveniva attraverso un semplice sistema di numerazione. Ogni versione era caratterizzata da tre numeri separati da punti. I primi due numeri identificavano la versione, mentre il terzo era relativo alla release. Il primo numero, al momento 2, e' lo stesso dal 1996. Il secondo numero ha sempre identificato il tipo di kernel: se era pari il kernel era della serie stabile, se era dispari il kernel era della serie di sviluppo.

Le versioni stabili erano rilasciate dopo accurati controlli e test e presentavano, in genere, solo correzione di bug o aggiunta di nuovi device driver. Al contrario le versioni di sviluppo possono essere anche molto diverse da una versione all'altra e gli utenti che usassero questi kernel per le proprie applicazioni potrebbero trovarsi a fronteggiare grossi problemi.

Durante lo sviluppo della versione 2.6 e' avvenuto un cambio significativo dello schema di numerazione. Fondamentalmente la seconda cifra non identifica piu' la serie stabile, oppure quella di sviluppo, quindi ora gli sviluppatori introducono pesanti cambiamenti gia' nella serie 2.6. Una serie 2.7 verra' creata al momento in cui dovranno essere testate caratteristiche tali da poter essere potenzialmente dannose per il sistema.

Il nuovo modello di sviluppo di Linux implica che due kernel con la stessa versione, ma numero di release diverso, possano differire significativamente. Quindi quando una nuova release del kernel viene rilasciata essa e' potenzialmente instabile o presentare dei bug. Per cercare di restringere questi problemi, gli sviluppatori possono rilasciare versioni con patch di ogni kernel. Queste sono identificate da un quarto numero nello schema di numerazione.

[1] Vedi Appendice A

1.4 Concetti base di Sistemi Operativi

Ogni computer include un set di base di software detto sistema operativo. La parte piu' importante di questo set e' detta *kernel,* esso viene caricato in RAM al boot del sistema e contiene tutta una serie di procedure critiche necessarie per far funzionare il sistema.

L'attivita' degli altri programmi e' meno cruciale e in genere essi servono ad eseguire tutti quei compiti per cui un computer e' stato acquistato. Ma le capacita' di un sistema dipendono fondamentalmente dal suo kernel.

Il kernel fornisce tutta una serie di servizi al resto del sistema ed e' determinante per l'esecuzione del software sovrastante. In questo senso, spesso la parola Sistema Operativo e' usata come sinonimo di kernel.

I due principali compiti di un kernel sono:

- Interagire con l'hardware, gestendo tutti gli elementi programmabili di basso livello inclusi nella piattaforma hardware.
- Fornire un ambiente di esecuzione alle applicazioni che girano sul computer.

Esistono sistemi operativi che permettono ai programmi utente la gestione diretta dell'hardware. Al contrario Unix nasconde tutti i dettagli a basso livello dell'organizzazione fisica del computer alle applicazioni utente. Quando un programma vuole accedere ad una risorsa hardware deve farne richiesta al sistema operativo. Se il kernel accetta la richiesta si occupera' di interagire con l'hardware per conto del programma utente.

Per realizzare questo meccanismo i moderni sistemi operativi si affidano alle caratteristiche offerte dall'hardware, che impediscono ai programmi utente di interagire direttamente con i componenti di basso livello o di accedere indiscriminatamente alle locazioni di memoria. In particolare l'hardware presenta almeno due diverse modalita' di esecuzione per la CPU: una modalita' non privilegiata per i programmi utente e una modalita' privilegiata per il kernel. Unix chiama questi stati *User Mode* e *Kernel Mode*.

1.4.1 Sistemi Multiutente

Un sistema multiutente e' un computer capace di eseguire concorrentemente e indipendentemente varie applicazioni appartenenti a due o piu' utenti. *Concorrentemente* significa che le applicazioni possono essere attive nello stesso momento e servirsi delle stesse risorse come CPU, memoria, hard disk, etc. *Indipendentemente* significa che ogni applicazione puo' eseguire le proprie procedure senza essere al corrente di cio' che le altre applicazioni stanno facendo. Naturalmente lo switch da un'applicazione all'altra comporta il rallentamento di entrambe e influisce sul tempo di risposta che un utente percepisce.

La maggior parte della complessita' dei sistemi operativi moderni riguarda proprio la minimizzazione dei ritardi relativi ad ogni programma e rendere la risposta all'utente il piu' rapidamente possibile.

I sistemi operativi multiutente devono includere diverse caratteristiche:

- Un sistema di autenticazione per stabilire l'identita' dell'utente.

- Un meccanismo di protezione nei confronti dei programmi utente difettosi che possano bloccare l'intero sistema.
- Un meccanismo di protezione nei confronti di programmi utente che possano fraudolentemente interferire con le attivita' di altri utenti.
- Un meccanismo di accounting che limiti la quantita' di risorse assegnate ad ogni utente.

Per assicurare meccanismi di protezione sicuri, il sistema operativo deve avvalersi della protezione hardware associata alla modalita' protetta della CPU.

1.4.2 Utenti e Gruppi

In un sistema multiutente, ogni utente ha uno spazio privato su una macchina. Il sistema operativo deve assicurare che questa porzione privata di spazio utente sia visibile solo dal suo proprietario. In particolare deve essere garantito che nessuna applicazione di un altro utente possa violare questo spazio privato.

Tutti gli utenti di un sistema Unix sono identificati da uno *User ID* (UID). Quando un utente inizia una sessione di lavoro, deve fornire delle credenziali per accedere alla macchina, in genere un login name e una password. Se la coppia utente/password non e' corretta il sistema nega l'accesso. Visto che la password dovrebbe essere segreta, la privacy e' garantita.

Per condividere materiale con altri utenti, ogni user fa parte di uno o piu' gruppi, identificati da un *Group ID* (GID). Ogni file e' associato esattamente con un gruppo.

Ogni sistema Unix ha un particolare utente chiamato *superutente o root*. L'amministratore del sistema deve fare login come root per gestire gli account utente, eseguire operazioni di manutenzione del sistema, etc. L'utente root puo' fare praticamente ogni cosa, perche' il sistema operativo non applica nessun criterio di protezione: in particolare root puo' accedere a qualsiasi file e interagire con ogni programma in esecuzione.

1.4.3 Processi

Tutti i sistemi operativi utilizzano un fondamentale elemento di astrazione: il *processo*. Un processo puo' essere definito come l'"istanza di un programma in esecuzione" oppure come il "contesto di esecuzione" di un programma. Nei sistemi operativi tradizionali, un processo esegue una sequenza di istruzioni in uno spazio di indizzamento, ovvero un gruppo di locazioni di memoria a cui il processo puo' accedere. I sistemi operativi moderni permettono l'esecuzione di processi con flussi di esecuzione multipli, ovvero sequenze di istruzioni multiple eseguite nello stesso spazio di indirizzamento.

I sistemi multiutente devono fornire un ambiente di esecuzione nel quale diversi processi possano essere attivi allo stesso tempo e accedano alle stesse risorse, principalmente la CPU. I sistemi che permettono la contemporanea esecuzione di processi attivi sono detti *multiprogramming o multiprocessing*.

E' importante distinguere il programma dal processo: vari processi possono eseguire lo stesso programma concorrentemente, mentre lo stesso processo puo' eseguire piu' programmi in sequenza.

Nei sistemi monoprocessore solo un processo puo' utilizzare la CPU in un dato momento, quindi un solo flusso di esecuzione andra' avanti in ogni momento. Piu' in generale il numero di CPU in un sistema e' sempre limitato, e quindi solo pochi processi saranno in esecuzione in ogni momento. Un componente del sistema operativo detto *scheduler* e' reponsabile della decisione di quale processo mandare in esecuzione. Alcuni sistemi operativi permettono solo processi *non-preemptable,* ovvero lo scheduler viene invocato solo quando un processo rilascia volontariamente le risorse. I processi di un sistema multiutente devono essere *preemptable*, spetta allora al sistema operativo tenere traccia di quanto ogni processo abbia utilizzato la CPU e periodicamente attivare lo scheduler.

Unix e' un sistema operativo multiutente con processi preemptable. Anche se nessun utente e' loggato sul sistema e nessuna applicazione e' in esecuzione, diversi processi di sistema monitorano le periferiche. In particolare una serie di processi sono in ascolto sui terminali di sistema in attesa del login degli utenti.

Tutti i sistemi Unix-like adottano un modello *processo/kernel.* Ogni processo ha l'illusione di essere solo sulla macchina e di avere accesso esclusivo alle risorse del sistema operativo. Quando un processo esegue una *system call* (ovvero richiede un servizio del sistema operativo), l'hardware cambia il livello di privilegio da User Mode a Kernel Mode, il processo mette in esecuzione una procedura del kernel strettamente connessa all'attivita' richiesta. In questa maniera il kernel agisce nel contesto di esecuzione del processo al fine di soddisfarne la richiesta. Quando la richiesta e' completata, la procedura del kernel forza l'hardware a tornare in User Mode e il processo continua la sua esecuzione dall'istruzione che seguiva la *system call.*

1.4.4 Architettura del kernel

La maggior parte dei kernel Unix sono monolitici: ogni substrato del kernel e' integrato nel kernel stesso e eseguito in Kernel Mode su richiesta dei processi. Al contrario i sistemi operativi a microkernel necessitano di una ristretta serie di funzioni dal kernel, in genere alcune primitive di sincronizzazione, uno scheduler e un meccanismo di interprocess communication. Diversi processi di sistema girano al di sopra del microkernel e implementano le altre funzioni.

Linux e' un sistema operativo a kernel monolitico. Per ovviare ad una serie di penalizzazioni prestazionali e poter avere una serie di vantaggi delle architetture a microkernel, il kernel di Linux implementa i *moduli.* Un modulo e' un file oggetto il cui codice puo' essere linkato e unlinkato dal kernel a runtime. Il codice oggetto in genere implementa una serie di funzioni che implementano un filesystem, un device driver o altre caratteristiche dello strato piu' alto del kernel. Il modulo e' eseguito in Kernel Mode per conto dei processi che ne richiedano le funzioni, cosi' come ogni altra funzione del kernel linkata staticamente.

I principali vantaggi di usare l'approccio a moduli sono:

- Approccio modulare

Visto che ogni modulo puo' essere linkato e unlinkato a runtime, gli sviluppatori devono introdurre delle interfacce ben definite per accedere alle strutture dati gestite dai moduli.

- Indipendenza dalla piattaforma

Sebbene possa dipendere da alcune caratteristiche dell'hardware, un modulo non dipende da una particolare piattaforma hardware.

- Uso minimale della memoria

Un modulo puo' essere linkato al kernel quando ce ne sia necessita' e unlinkato quando non serve piu'.

- Nessuna penalizzazione della performance

Una volta linkato il codice oggetto del modulo e' equivalente a quello del resto del kernel linkato staticamente.

1.5 Il Filesystem Unix

Il progetto del sistema operativo Unix e' incentrato sul filesystem, che ha diverse caratteritiche interessanti.

1.5.1 File

Un file Unix non e' altro che un contenitore di informazioni strutturato come una sequenza di byte, il kernel non interpreta il contenuto del file. Esistono molte librerie che implementano astrazioni di alto livello, come per esempio i record. In ogni caso le routine di queste librerie devono affidarsi alle system call che offre il kernel. Dal punto di vista dell'utente, i file sono organizzati in un namespace strutturato ad albero.

Tutti i nodi dell'albero ad eccezione delle foglie rappresentano nomi di directory. Un nodo di tipo directory contiene informazioni a riguardo di file e directory al di sotto di esso. Un nome di file o di directory consiste di una sequenza di caratteri ascii (esistono perlatro una serie di filesystem che permettono l'uso di Unicode), con l'eccezione di / e del null terminator \0. Classicamente i filesystem impongono un limite alla lunghezza dei nomi a 255 caratteri. La directory radice dell'albero e' chiamata *root directory*. Per convenzione il suo nome e' /. Lo stesso nome puo' essere utilizzato in directory diverse, mentre nella stessa directory i nomi devono essere distinti.

Unix associa una *current working directory* con ogni processo; esso appartiene al contesto di esecuzione del processo stesso e identifica la directory attualmente utilizzata dal processo. Per identificare un file, il processo usa un *pathname*, che consiste di una serie di nomi di directory separate da /, che conducono fino al file. Se il primo carattere del pathname e' / allora il pathname e' detto assoluto, perche' il suo punto di inizio e' la root directory. Altrimenti, se il primo oggetto del pathname e' il nome di una directory o di un file, il pathname e' detto relativo.

Spesso, quando si specificano nomi di file, si utilizzano le espressioni . e .. Esse denotano la current working directory e la sua parent directory. Nel caso in cui la current working directory sia la root directory, allora . e .. coincidono.

1.5.2 Hard e soft link

Un filename incluso in una directory viene chiamato hard link o semplicemente un link. Lo stesso file potrebbe avere piu' link nella stessa directory o in piu' directory, cosi' da avere piu' filename.

Il comando:
ln file1 file2
viene utilizzato per creare un nuovo hard link che ha il pathname file2 per un file identificato dal pathname file1.

Gli hard link hanno due limitazioni:

- Non e' possibile creare hard link per le directory. Questo potrebbe comportare la trasformazione dell'albero delle directory in un grafo con cicli. Rendendo di fatto impossibile localizzare un file in base al suo nome.
- I link possono essere creati solo fra file inclusi nello stesso filesystem.

Al fine si superare queste limitazioni sono stati creati i *soft link (o link simbolici)*. I link simbolici sono brevi file che contengono un pathname arbitrario di un altro file. Il pathname puo' referenziare ogni file o directory in ogni filesystem, perfino file non esistenti.

Il comando
ln -s file1 file2
crea un soft link con il pathname file2 che referenzia il pathname file1. Il nuovo file che viene creato contiene il nome indicato dal pathname di file1. In questo modo ogni riferimento a file2 viene automaticamente tradotto in un riferimento a file1.

1.5.3 Tipi di file

I file Unix possono appartene ad uno dei seguenti tipi:

- Regular file
- Directory
- Symbolic link
- Device file block-oriented
- Device file character-oriented
- Pipe e named pipe (dette anche Fifo)
- Socket

I primi tre tipi sono i costituenti di ogni filesystem Unix.

I device file sono relativi ai dispositivi di I/O e ai device drivers che fanno parte del kernel. Per esempio quando un programma accede ad un device file, esso in realta' agisce direttamente sul dispositivo di I/O associato con questo file.

Le pipe e le socket sono file speciali usati per interprocess communication.

1.5.4 Descrittori di file e inode

Unix fa una chiara distinzione fra il contenuto di un file e le informazioni che riguardano il file. Con l'eccezione di device file e file di particolari filesystem, ogni file consiste di una sequenza di byte. Il file non include nessuna informazione di controllo, come la lunghezza oppure un delimitatore di fine file (EOF).

Tutte le informazioni necessarie al filesystem per gestire un file sono incluse in una struttura detta *inode*. Ogni file ha il suo inode, che il filesystem utilizza per identificare il file.

Sebbene i filesystem e le funzioni del kernel che li gestiscono possano variare molto da un sistema Unix all'altro, esistono una serie di attributi (definiti dagli standard POSIX) che devono sempre essere presenti:

- Tipo di file
- Numero di hard link associato con il file
- Lunghezza del file in byte
- Device ID (identificatore del dispositivo che contiene il file)
- Inode number che identifica il file nel filesystem
- UID del proprietario del file
- GID del file
- Timestamp che identificano: inode status change time, last access time, modify time
- Diritti di accesso e file mode

1.5.5 Diritti di accesso e file mode

I potenziali utenti di un file ricadono in tre caterogie:

- L'utente proprietario del file
- Gli utenti che fanno parte del gruppo a cui appartiene il file, escluso il proprietario
- Tutti gli altri utenti (others)

Ci sono tre tipi di diritti di accesso: *read, write e execute* per ognuna delle tre classi. Quindi il set di diritti di accesso associati con un file consiste di nove flag binari. Tre flag addizionali, chiamati *suid (set user ID), sgid (set group ID) e sticky* definiscono il file mode. Quando sono applicati a file eseguibili i seguenti bit hanno il significato:

- SUID: Un processo che esegue un file mantiene lo UID del proprietario del processo. Comunque, se i file eseguibili hanno il suid bit attivo, il processo acquisisce lo UID del proprietario del file.
- SGID: Un processo che esegue un file mantiene il GID del proprietario. Comunque, se il file ha il sgid bit attivo, il processo acquisisce il GID del file.

Sticky: Un file eseguibile con lo sticky bit attivo corrisponde ad una richiesta al kernel di mantenere il programma in memoria dopo che la sue esecuzione e' terminata.[2]

Quando un file viene creato da un processo, lo owner ID e' quello del proprietario del processo. Il group owner ID puo' essere quello del processo oppure quello della directory in cui viene creato, a seconda del valore dello sgid flag della directory.

1.6 Una visione di insieme del kernel Unix

Il kernel Unix fornisce l'ambiente di esecuzione in cui le applicazioni possono girare. Quindi, il kernel deve implementare un set di servizi e le corrispondenti interfacce. Le applicazioni utilizzano questi servizi e, in genere, non fanno un accesso diretto all'hardware.

1.6.1 Il modello Process/Kernel

Una CPU puo' lavorare in Kernel Mode oppure User Mode. Al momento, alcune CPU hanno piu' di due modalita' di esecuzione. Per esempio i processori della famiglia 80x86 ne hanno 4. In ogni caso tutti i kernel Unix ne usano soltanto due.

Quando un programma viene eseguito in User Mode, esso non puo' accedere alle strutture dati del kernel o al codice del kernel. Quando un'applicazione gira in Kernel mode queste limitazioni non agiscono. Ogni architettura di CPU mette a disposizione alcune particolari istruzioni per passare da User mode a Kernel mode e viceversa. Un programma in genere gira in User mode e passa in Kernel mode solo quando richiede un servizio fornito dal kernel. Quando il kernel ha soddisfatto la richiesta del programma, fa tornare il programma in User mode.

I processi sono entita' dinamiche che in genere hanno una vita limitata nel sistema. Il compito di creare, eliminare e sincronizzare i processi esistenti e' delegato a un gruppo di routine del kernel.

Il kernel in se non e' un processo, ma un manager di processi. Il modello process/kernel assume che i processi che richiedono i servizi del kernel usino delle particolari interfacce chiamate *system call*. Ogni system call predispone i parametri che identificano la richiesta del processo e poi esegue le istruzioni (dipendenti dall'architettura) per passare da User mode a Kernel mode.

A fianco dei processi utente, il sistema Unix include alcuni processi privilegiati, chiamati kernel threads, con le seguenti caratteristiche:

- Sono eseguiti in Kernel mode nell'address space del kernel
- Non interagiscono con gli utenti e non richiedono terminali

[2] Questo flag e' obsoleto, oggi si usa un approccio piu' moderno basato sulla condivisione delle pagine di codice. Peraltro lo sticky bit conserva un certo valore per le directory, in questo caso se tale bit è impostato un file potrà essere rimosso dalla directory soltanto se l'utente ha il permesso di scrittura su di essa ed inoltre è vera una delle seguenti condizioni:

- l'utente è proprietario del file
- l'utente è proprietario della directory
- l'utente è l'amministratore

- Sono generalmente creati durante lo startup del sistema e rimangono attivi fino allo shutdown

Sui sistemi monoprocessore solo un processo e' in esecuzione in un certo istante e puo' essere in esecuzione in User o in Kernel mode. Se gira in Kernel mode, allora il processo sta eseguendo qualche routine del kernel. In realta' il kernel Unix fa molto di piu' che gestire le system call, infatti le routine di sistema possono essere attivate in molti modi:

- Un processo invoca una system call
- La CPU mentre esegue un processo solleva un eccezione, che in genere e' una condizione inusuale. Il kernel gestisce l'eccezione per conto del processo che l'ha causata.
- Una periferica invia un segnale di interruzione alla CPU per notificare che un certo evento necessita attenzione, un cambio di stato o il completamento di un operazione di I/O. Ogni segnale di interrupt viene gestito da un programma del kernel chiamato *interrupt handler*. Siccome le periferiche operano in modo asincsrono rispetto alla CPU, le interruzioni possono avvenire in ogni momento.
- Viene eseguito un thread del kernel. Siccome esso gira in Kernel mode, il programma corrispondente deve essere considerato parte del kernel.

1.6.2 Implementazione dei processi

Il kernel per poter amministrare i processi, rappresenta ogni processo per mezzo di un *process descriptor* che include informazioni circa lo stato del processo.

Quando il kernel arresta l'esecuzione di un processo, salva il contenuto dei vari registri del processore nel process descriptor, fra cui:

- Il program counter e lo stack pointer
- I registri di uso generale
- I registri floating point
- I registri di controllo del processore (Processor Status Word)
- I registri di memory management utilizzati per tenere traccia della RAM utilizzata dal processo

Quando il kernel decide di rimettere in esecuzione un processo, esso utilizza l'opportuno process descriptor per caricare i registi della CPU. Siccome il valore salvato del program counter punta all'istruzione successiva a quella dell'ultima istruzione eseguita, il processo riprende l'esecuzione dal punto in cui era stato arrestato.

Quando un processo non e' in esecuzione sulla CPU, e' in attesa di qualche evento. I kernel Unix distinguono molti stati di attesa, che sono generalmente implementati da code di descittori di processo; ogni coda corrisponde al set di processi in attesa di uno specifico evento.

1.6.3 Reentrant kernel

Tutti i kernel Unix sono *reentrant*. Questo significa che diversi processi possono essere in esecuzione in Kernel mode nello stesso momento. Naturalmente sui sistemi monoprocessore solo un processo alla volta va avanti, ma altri possono essere bloccati in Kernel mode in attesa della CPU o del completamento di un operazione di I/O.

Un modo per garantire la reentrancy e' scrivere funzioni che modifichino solo variabili locali e non alterino le strutture dati globali. Queste funzioni si chiamano *reentrant functions*. Ma un kernel reentrant non puo' limitarsi solo a queste funzioni. Il kernel quindi include anche funzioni non reentrant e usa dei meccanismi di locking per essere sicuro che solo un processo alla volta possa eseguire una nonreentrant function.

Se un interrupt hardware si verifica, un kernel reentrant e' in grado di sospendere il processo in esecuzione anche in Kernel mode. Questa capacita' e' molto importante, perche' accresce il throughput del device controller che ha generato l'interrupt. Una volta che un device ha generato un interrupt, esso aspetta l'acknowledge da parte della CPU. Se il kernel risponde rapidamente il device controller potra' eseguire altre operazioni mentre la CPU gestisce l'interrupt.

Un *kernel control path* identifica la sequenza di istruzioni eseguite dal kernel per gestire una system call, un'eccezione o un interrupt.

Nel caso piu' semplice, la CPU esegue un kernel control path sequenzialmente dalla prima instruzione all'ultima. Quando una delle seguenti condizioni si verifica, il kernel interlaccia i kernel control path:

- Un processo in esecuzione in User Mode invoca una system call e il corrispondente kernel control path verifica che la richiesta non puo' essere soddisfatta immediatamente; quindi viene invocato lo scheduler per selezionare un nuovo processo da eseguire. Il primo kernel control path e' lasciato non terminato e la CPU riprende l'esecuzione di qualche altro kernel control path. In questo caso due kernel control path sono eseguiti per conto di due diversi processi.

- La CPU rileva un'eccezione. Il primo control path e' allora sospeso e la CPU inizia l'esecuzione di una procedura di gestione. Quando la procedura termina il precedente control path puo' essere ripreso. In questo caso i due control path sono eseguiti per conto dello stesso processo.

- Un hardware interrupt arriva mentre la CPU sta eseguendo un kernel control path con le interruzioni abilitate. Il primo control path viene lasciato non finito e la CPU inizia l'esecuzione di un altro control path per gestire l'interrupt. Il primo kernel control path viene ripreso quando il gestore dell'interrupt finisce. In questo caso i due kernel control path girano nel contesto di esecuzione dello stesso processo e il tempo totale di uso della CPU viene conteggiato ad esso, sebbene non sia detto che il gestore dell'interrupt agisse per conto del processo.

- Un interrupt viene generato mentre la CPU sta girando con la kernel preemption abilitata e un processo a priorita' piu' alta e' nello stato eseguibile. In questo caso il primo kernel control path viene lasciato non finito e la

CPU riprende l'esecuzione di un altro kernel control path per conto del processo a priorita' piu' alta.

1.6.4 Process Address Space

Ogni processo ha il proprio spazio di indirizzamento privato. Un processo in esecuzione in User Mode fa riferimento ad uno stack e aree dati e codice private. Quando l'esecuzione e' in Kernel Mode, il processo indirizza le aree dati e codice del kernel e utilizza un altro stack.

Siccome il kernel e' reentrant, vari kernel control path relativi a diversi processi possono essere eseguiti a turno. In questo caso ogni kernel control path fa riferimento al suo kernel stack privato.

Sebbene ogni processo abbia la sensazione di avere uno spazio di indirizzamento privato, ci sono volte in cui parte dell'address space e' condiviso con altri processi. In alcuni casi la condivisione e' esplicitamente richiesta dal processo, in altri e' fatta automaticamente dal kernel per ridurre l'occupazione di memoria.

I processi possono condividere parte del loro spazio di indirizzamento e realizzare in questo modo una sorta di interprocess communication, utilizzando la tecnica della "shared memory" introdotta da SysV.

Linux supporta inoltre la chiamata di sistema mmap(), che permette a parti di un file o ad informazioni immagazzinate in un block device di essere mappate in una parte dell'address space di un processo. Questa tecnica puo' essere un'alternativa alle comuni read e write per trasferire dati. Se un file e' condiviso da piu' processi, la sua mappatura in memoria viene inclusa nello spazio di indirizzamento di ogni processo.

1.6.5 Sincronizzazione e regioni critiche

L'implementazione di un kernel reentrant comporta l'uso di tecniche di sincronizzazione. Se un kernel control path viene sospeso mentre sta manipolando una struttura dati del kernel, nessun altro control path dovrebbe agire su quella struttura finche' non sia stata riportata ad uno stato consistente. Altrimenti l'interazione dei due control path potrebbe corrompere i dati.

Quando il risultato di una computazione dipende da come due o piu' processi sono schedulati, il codice e' scorretto. Questa viene chiamata una *race condition*.

In genere, l'accesso sicuro alle variaili globali e' assicurato dall'uso di *atomic operations*. Ovvero singole operazioni non interrompibili. In ogni caso il kernel contiene strutture dati a cui non e' possibile accedere attraverso singole operazioni. Ogni porzione di codice che debba essere terminata da ogni processo che la inizia prima che qualsiasi altro processo possa entrarci e' detta *critical region*.

Questi tipi di problemi non riguardano solo i kernel control path, ma anche i processi che condividono dati comuni. Esistono diverse tecniche di sincronizzazione per il kernel e i processi.

Disabilitare la kernel preemption Per fornire una soluzione drastica ai problemi si sincronizzazione gli Unix tradizionali sono nonpreemptive: quando un processo e' in esecuzione in Kernel mode, non puo' essere arbitrariamente sospeso e sostituito con un altro. Quindi, sui sistemi monoprocessore, tutte le strutture dati del kernel che non siano aggiornate dai gestori di interrupt e eccezioni sono ad accesso sicuro per il kernel.

Naturalmente un processo in kernel mode puo' volontariamente rilasciare la CPU, ma, in questo caso, deve assicurare che tutte le strutture dati siano lasciate in uno stato consistente. In piu', quando ripende l'esecuzione, deve controllare i valori della struttura a cui accedeva precedentemente perche' potrebbero essere cambiati.

Un meccanismo di sincronizzazione appliacabile ai kernel preemptive e' quello di disabilitare la preemption prima di accedere ad una critical region e riabilitarla dopo averla lasciata.

La disabilitazione della preemption non e' sufficiente per i sistemi multiprocessore, in quanto due kernel control path in esecuzione su due diverse CPU potrebbero accedere alla stessa struttura dati.

Disabilitare gli interrupt Un altro meccasimo di sincronizzazione per i sistemi uniprocessore consiste nel disabilitare tutti gli hardware interrupt prima di accedere ad una critical region e riabilitarli subito dopo. Questo meccanismo, sebbene semplice, e' molto rudimentale. Se la critical region e' ampia, gli interrupt possono rimanere disabilitati per un periodo relativamente lungo, causando un blocco di tutte le attivita' hardware.

In piu', sui sistemi multiprocessore, disabilitare gli interrupt sulla CPU locale non e' sufficiente e altre tecniche di sincronizzazione dovrebbero essere usate.

Semafori Un meccanismo largamente usato, capace di agire sia per i sistemi monoprocessore che multiprocessore, e' costituito dall'uso dei *semafori*. Un semaforo e' semplicemente un contatore associato con una struttura dati. Esso viene controllato da tutti i thread del kernel prima di accedere alla struttura. Ogni semaforo puo' essere visto come composto da:

- Una variabile intera
- Una lista di processi in attesa
- Due metodi atomici: up() e down()

il metodo down decrementa il valore del semaforo. Se il nuovo valore e' minore di 0, il metodo aggiunge il processo in esecuzione alla lista e blocca (invoca lo scheduler). Il metodo up() incrementa il valore del semforo, se il nuovo valore e' maggiore o uguale a zero, uno o piu' processi nella lista vengono riattivati.

Ogni struttura dati che deve essere protetta ha il proprio semaforo, inizializzato a 1. Quando un kernel control path vuole accedere alla struttura dati, esegue il metodo down() sul semaforo appropriato. Se il valore non e' negativo, l'accesso alla struttura e' concesso. Altrimenti il processo che sta eseguendo il kernel control path viene aggiunto alla lista del semaforo e bloccato. Quando un altro processo esegue il metodo up() sul semaforo ad uno dei processi nella semaphore list e' concesso l'accesso.

Spin Locks Nei sistemi multiprocessore i semafori non sono sempre la migliore soluzione ai problemi di sincronizzazione. Alcune strutture dati del kernel dovrebbero essere protette dall'essere contemporaneamente accedute dai kernel control path che girano su CPU diverse. In questo caso, se il tempo richiesto per

aggiornare la struttura e' breve, il semaforo puo' essere molto inefficiente, perche' le operazioni connesse sono computazionalmente dispendiose, e allo stesso tempo l'altro kernel control path potrebbe aver gia' rilasciato il semaforo.

In questi casi, i sistemi multiprocessore utilizzano gli *spin lock*. Uno spin lock e' molto simile ad un semaforo, ma non ha la lista dei processi; quando un processo trova il lock chiuso da un altro processo, esso "*spins around*" eseguendo un breve ciclo di istruzioni finche' il lock non si apre.

Naturalmente gli spin locks sono inutili in ambienti monoprocessore. Quando un kernel control path cerca di accedere a una struttura in lock, inizia un ciclo senza fine. Quindi il kernel control path che sta aggiornando la struttura dati protetta non ha nessuna possibilita' di continuare la sua esecuzione e rilasciare lo spin lock. Il risultato e' il blocco del sistema.

Evitare i deadlock I processi e i kernel control path che si sincronizzano fra di se possono entrare nello stato di *deadlock*. Il caso piu' semplice di deadlock avviene quando un processo p1 puo' accedere ad una struttura dati a e il processo p2 puo' accedere alla struttura b, dopo di che p1 attende che b sia disponibile e p2 aspetta di accedere ad a. Un deadlock causa il blocco totale dei processi o dei kernel control path coinvolti.

Per come il kernel e' progettato, i deadlock diventano un problema quando il numero di kernel lock utilizzato e' alto. In questo caso e' molto difficile essere certi che nessun deadlock possa essere raggiunto per ogni possibile interlacciamento dei kernel control path. Vari sistemi operativi, tra cui Linux, evitano questo problema ordinando i lock.

1.6.6 Segnali e Interprocess Communication

I segnali Unix forniscono un meccanismo per rendere noti ai processi gli eventi del sistema. Ogni evento ha il proprio numero di segnale, a cui e' associata anche una costante simbolica (per es. SIGKILL). Ci sono due tipi di eventi di sistema:

- Notifiche Asincrone: per es. un utente puo' inviare un segnale ad un processo.
- Notifiche Sincrone: per es. il kernel puo' inviare il segnale SIGSEGV ad un processo che accede una locazione di memoria ad un indirizzo non valido.

Lo standard POSIX definisce circa 20 diversi segnali, due dei quali sono definibili dall'utente e possono essere usati come un meccanismo primitivo per la comunicazione e sincronizzazione fra i processi in User mode. In generale un processo puo' rispondere alla ricezione di un segnale in due modi:

- Ignorare il segnale
- Eseguire in modo asincrono una certa procedura (signal handler)

Se il processo non specifica nessuna delle alternative, il kernel effettua una *default action* che dipende dal segnale. Le cinque possibili azioni sono:

- Terminare il processo.

- Scrivere il contesto di esecuzione e il contenuto dello spazio di indirizzamento in un file (core dump) e terminare il processo.
- Ignorare il segnale.
- Sospendere il processo.
- Riprendere l'esecuzione del processo, se era sospeso.

La gestione dei segnali del kernel e' piuttosto elaborato, siccome la semantica POSIX permette ai processi di bloccare temporaneamente i segnali. In piu' i segnali SIGKILL e SIGSTOP non possono essere ignorati o gestiti in proprio dal processo.

AT&T Unix System V introdusse altri generi di interprocess communication fra i processi in User Mode. Queste modalita' sono state adottate dalla maggior parte dei kernel Unix: *semafori, message queues e shared memory.* Tutto questo e' globalmente conosciuto come System V IPC.

1.6.7 Process Management

Unix fa una netta distinzione fra il processo e il programma che sta eseguendo. Le system call fork() e _exit() sono usate rispettivamente per creare un nuovo processo e per terminarlo, quando una chiamata ad una funzione della famiglia exec() viene fatta per caricare un nuovo programma. Dopo che questa chiamata e' stata fatta, il processo riprende l'esecuzione con un nuovo address space che contiene il programma caricato.

Il processo che invoca la fork() e' il *parent* e il nuovo processo il *child*. Parent e child possono referenziarsi perche' la struttura dati che descrive ogni processo include un puntatore all'oggetto padre e puntatori a tutti gli oggetti figli.

Una rozza implementazione della fork() richiederebbe che il codice e i dati del processo parent fossero duplicati e le copie assegnate ai child. I kernel moderni invece possono contare sulle unita' di paginazione delle CPU e seguono l'approccio Copy-On-Write, che pospone la duplicazione delle pagine all'ultimo momento possibile.

La chiamata _exit() termina un processo. Il kernel la gestisce rilasciando le risorse relative al processo e inviando il segnale SIGCHLD al padre, che per default viene ignorato.

Processi Zombie Un processo padre puo' informarsi dello stato dei propri processi figlio per mezzo della system call wait4(), quando esso termina la procedura ritorna il PID del child terminato.

Quando viene eseguita questa funzione il kernel controlla se un processo figlio e' gia' terminato. Uno stato particolare dei processi detto *zombie* e' stato introdotto per identificare i processi terminati: il processo rimane in questo stato finche' il processo padre esegue una wait4() su di esso. Il gestore della system call estrae i dati sulle risorse utilizzate dai campi del process descriptor; il process descriptor puo' essere rilasciato una volta che i dati sono stati raccolti.

Il problema potrebbe verificarsi se il processo padre termina, mentre i figli sono ancora in esecuzione. Non sarebbe possibile allora eseguire la chiamata a wait4() e rilasciare le risorse ad essi associate. La soluzione a questo problema sta in un particolare processo chiamato *init*, creato durante durante l'avvio dei

sistema. Quando un processo termina, il kernel cambia i puntatori nel process descriptor di tutti i figli per farli diventare figli del processo init. Questo processo controlla l'esecuzione di tutti i suoi figli e ogni tanto richiama la system call wait4(), il cui effetto e' quello di occuparsi di tutti gli zombie orfani.

Gruppi di processi e login sessions Gli Unix moderni introducono il concetto di "gruppo di processi" per rappresentare l'astrazione di un *job*. Per eseguire il comando

ls | sort | more

una shell che supporti i process groups, crea un nuovo process group per i tre processi corrispondenti ai tre comandi in esecuzione. In questo modo la shell puo' agire sui tre comandi come se fossero una sola entita' (il job). Ogni descrittore di processo include un campo contenente il process group ID. Ogni gruppo di processi puo' avere un group leader, il cui PID coincide con il process group ID. Un nuovo processo viene inserito nel process group del parent.

I kernel Unix moderni introducono anche le login sessions. Una login session contiene tutti i processi che sono discendenti del processo che ha creato la sessione di lavoro su uno specifico terminale, in genere il primo processo shell creato per l'utente. Tutti i processi in un process group devono essere nella stessa login session. Una login session puo' avere piu' process group attivi simultaneamente, uno di questi e' sempre in foreground, ovvero ha acceso al terminale. Gli altri processi attivi sono in background.

1.6.8 Gestione della memoria

La gestione della memoria e' di gran lunga l'attivita' piu' complessa di un kernel Unix.

Virtual Memory Tutti i kernel Unix recenti mettono a disposizione un'utile astrazione chiamata memoria virtuale. La memoria virtuale agisce come un livello logico fra le richieste di memoria e la hardware Memory Managment Unit (MMU). La memoria virtuale ha molti scopi e vantaggi:

- Vari processi possono essere eseguiti contemporaneamente (concorrentemente)
- E' possibile eseguire applicazioni le cui esigenze di memoria siano maggiori della memoria fisica disponibile
- I processi possono eseguire un programma il cui codice e' parzialmente caricato in memoria
- Ad ogni processo e' consentito di accedere ad una porzione della memoria fisica disponibile
- I processi possono condividere una singola immagine di una libreria o di un programma
- I programmi possono essere rilocabili (relocatable), cosi' da poter essere messi ovunque in memoria fisica

- I programmatori possono scrivere codice indipendente dalla piattaforma, perche' non devono preoccuparsi dell'organizzazione della memoria fisica

L'ingrediente principale della memoria virtuale e' la nozione di *virtual address space.* Il set di riferimenti di memoria che un processo puo' usare e' differente dagli indirizzi di memoria fisica. Quando un processo usa un indirizzo virtuale, il kernel e la MMU cooperano per trovare la vera locazione fisica dell'oggetto richiesto nella memoria.

Le moderne CPU includono una circuiteria che traduce automaticamente gli indirizzi virtuali in indirizzi fisici. La Ram disponibile viene partizionata in page frames tipicamente di 4 o 8KB. Un set di page tables viene introdotto per specificare come gli indirizzi virtuali e quelli fisici corrispondano. Questi circuiti rendono l'allocazione di memoria semplice, visto che la richiesta di un blocco di memoria virtuale contigua puo' essere soddisfatta allocando un gruppo di page frames che abbiano indirizzi fisici non contigui.

Uso della RAM Tutti i sistemi operativi Unix distinguono chiaramente fra due porzioni di memoria RAM. Alcuni megabyte sono dedicati all'immagine del kernel (codice del kernel e strutture dati statiche). Il resto della RAM e' generalmente gestita dal sistema di memoria virtuale e viene usata in tre possibili modi:

- Per soddisfare le richieste del kernel per buffer, descrittori e altre strutture dati dinamiche
- Per soddisfare le richieste dei processi per aree generiche di memoria e per la mappatura di file
- Per avere migliori prestazioni dai dischi e da altri dispositivi bufferizzati per mezzo di cache

ogni tipo di richiesta e' preso in considerazione. D'altro canto, visto che la RAM disponibile e' limitata, qualche tipo di bilanciamento fra i tipi di richiesta deve essere fatto, in particolare quando e' rimasta poca memoria. In piu' quando vengono raggiunte alcune soglie critiche della memoria disponibile e un algoritmo di reclaim di page frame viene invocato occorre scegliere quali page frames liberare; non esistono soluzioni banali a questi problemi, le uniche soluzioni derivano da algoritmi empirici resi efficienti nel tempo.

Uno dei problemi maggiori di cui si deve occupare il sistema di memoria virtuale e' la *frammentazione della memoria.* Idealmente, una richiesta di memoria dovrebbe fallire solo quando il numero di page frame liberi e' troppo piccolo. Pero' il kernel e' spesso costretto ad usare aree di memoria fisica contigue. Quindi la richiesta di memoria potrebbe fallire anche se c'e' abbastanza memoria disponibile, ma non e' disponibile in una sola porzione.

Kernel Memory Allocator L'allocatore di memoria del kernel (KMA) e' un sottosistema che cerca di soddisfare le richieste per aree di memoria da tutte le parti del sistema. Alcune di queste richieste vengono da altri sottosistemi del kernel che necessitano di memoria per il funzionamento del kernel stesso, altre arrivano per mezzo delle system call dai programmi utente per richiedere l'aumento dello spazio di indirizzamento del processo. Un buon KMA deve avere le seguenti caratteristiche:

- Deve essere rapido. Questo e' l'attributo principale, perche' viene utilizzato da tutti i sottosistemi del kernel (incluso il gestore degli interrupt)
- Deve minimizzare la quantita' di memoria persa
- Dovrebbe ridurre la frammentazione della memoria
- Dovrebbe cooperare con gli altri sottosistemi di gestione della memoria per assegnare e rilasciare i page frame a essi

Gestione del process virtual address space Lo spazio di indirizzamento di un processo contiene tutti gli indirizzi virtuali che un processo puo' referenziare. Il kernel di solito memorizza un process virtual address space come una lista di descrittori di aree di memoria. Per esempio quando un processo esegue un programma attraverso una chiamata ad una system call della famiglia exec(), il kernel assegna al processo un virtual address space che include aree di memoria per:

- Il codice eseguibile del programma
- I dati inizializzati del programma
- I dati non inizializzati del programma
- Lo User Mode stack
- Il codice eseguibile e i dati delle librerie condivise necessarie
- Lo heap (la memoria dinamicamente richiesta dal programma)

Tutti i moderni sistemi operativi adottano una strategia di allocazione di memoria detta *demand paging*. In questa maniera un processo puo' iniziare l'esecuzione con nessuna pagina in memoria virtuale. Quando viene fatto un accesso ad una pagina non presente, la MMU genera un eccezione; il gestore dell'eccezione trova la regione di memoria coinvolta, alloca una pagina libera e la inizializza con i dati appropriati. Nello stesso modo, quando un processo richiede dinamicamente memoria utilizzando le chiamate di sistema malloc() o brk(), il kernel si limita ad aggiornare la dimensione dello heap del processo. Un page frame viene assegnato al processo solo quando genera un'eccezione mentre referenzia i suoi indirizzi di memoria virtuali.

Gli spazi di memoria virtuale permettono anche altre efficienti strategie, come per esempio il Copy-on-Write. Per esempio quando un nuovo processo viene creato, il kernel assegna i page frame del processo padre all'address space del figlio, ma li marca read-only. Un'eccezione viene sollevata quando il processo padre o il processo figlio cerca di modificare il contenuto di una pagina. Il gestore delle eccezioni assegna un nuovo page frame al processo coinvolto e lo inizializza con il contenuto della pagina originale.

Caching Una buona parte della memoria fisica disponibile e' usata come cache per gli hard disk e gli altri device a blocchi. Questo perche' gli hard disk sono piuttosto lenti: un accesso al disco richiede diversi millisecondi, che e' un tempo molto lungo in rapporto al tempo di accesso alla RAM. Quindi i dischi sono spesso il collo di bottiglia nelle prestazioni di un sistema. Come regola generale,

una delle politiche del kernel e' quella di ritardare la scrittura su disco il piu' possibile. Come risultato i dati letti dal disco e non piu' utilizzati dai processi restano in memoria.

La strategia e' basata sul fatto che c'e' una buona probabilita' che un nuovo processo richieda l'acceso ai dati letti o scritti da un processo che non esiste piu'. Quando un processo richiede un accesso al disco, il kernel controlla se i dati richiesti sono presenti in cache. Ogni volta che questo accade (cache hit) il kernel puo' soddisfare la richiesta senza accedere al disco.

La chiamata di sistema sync() forza la sincronizzazione, scrivendo tutti i "dirty buffers" (quelli il cui contenuto e' diverso dai corrispondenti disk block) sul disco. Per evitare la perdita di dati, tutti i sistemi operativi si prendono cura di scrivere periodicamente i "dirty buffers" su disco.

1.6.9 Device drivers

Il kernel interagisce con i dispositivi di I/O per mezzo di *device drivers*. I device drivers sono inclusi nel kernel e sono costituiti da strutture dati e funzioni che controllano uno o piu' dispositivi. Ogni driver interagisce con il resto del kernel per mezzo di ben precise interfacce.

Questo approccio ha i seguenti vantaggi:

- Il codice per il particolare dispositivo puo' essere incluso in uno specifico modulo
- I venditori di hardware possono aggiungere un nuovo dispositivo senza conoscere il codice del kernel, ma sono necessarie solo le specifiche delle interfacce
- Il kernel tratta tutti i device in modo uniforme e compie gli accessi attraverso la stessa interfaccia
- E' possibile scrivere un device driver come modulo da caricare dinamicamente senza il reboot del sistema. E' possibile anche rimuovere il modulo non piu' utilizzato. In questo modo l'immagine del kernel in memoria viene minimizzata.

La parte visibile agli utenti dell'interfaccia dei device driver e' costituita dalla directory /dev. Ogni device file fa riferimento ad un device driver, che viene invocato dal kernel per eseguire l'operazione richiesta sul componente hardware.

2 Appendice A - Version numbering

Il numero di versione del kernel Linux al momento consiste di quattro numeri, a seguito di un recente cambiamento nella usuale politica di numbering che prevedeva un versioning scheme a tre valori. Quindi il version number e' composto da: **A.B.C[.D]**.

- Il numero *A* denota la *versione del kernel*. Viene cambiato con la frequenza piu' bassa e solo quando avvengono modifiche generali del codice e dei concetti del kernel. Il cambiamento e' avvenuto due volte nella storia del kernel: nel 1994 (version 1.0) e nel 1996 (version 2.0).
- Il numero *B* identifica la *major revision del kernel*.
  - Prima della versione 2.6.x di Linux, i numeri pari identificavano una release stabile, come 1.2, 2.4 o 2.6. I numeri dispari storicamente hanno identificato release di sviluppo, come 1.1 o 2.5. Quest'ultime erano dedicate al test di nuove caratteristiche e driver fino al punto in cui esse fossero diventate sufficientemente stabili da essere incluse nelle stable release.
  - A partire dal kernel 2.6 nessun significato viene attribuito a numeri pari o dispari, con nuove caratteristiche che sono direttamente introdotte nelle stesse serie del kernel. Linus Torvalds ha affermato che questo sara' il modello per il prossimo futuro.
- Il numero *C* indica la *minor revision del kernel*. Nel vecchio versioning scheme a tre numeri, questo valore veniva cambiato quando patch di sicurezza, bugfix, nuove caratteristiche e driver venivano implementate nel kernel. Con la nuova politica, esso viene cambiato solo quando nuovi driver o caratteristiche vengono introdotti. Le minor fix vengono gestite con il valore D.
- Un valore *D* viene introdotto quando si verifica un serio errore che richiede una correzione immediata. Al tempo stesso non ci sono abbastanza cambiamenti che legittimino il rilascio di una nuova minor version. Dalla versione 2.6.11, questa e' stata adottata come la nuova versioning policy. I bug fix e le patch di sicurezza sono gestite dal quarto numero, mentre i cambiamenti piu' corposi vengono introdotti solo con il cambio del minor revision number.

Oltre a questi valori, a volte e' possibile che ci siano delle combinazioni di caratteri dopo la versione (come *rc1* o *mm2*). I valori *rc* si riferiscono alle release candidate e indicano un rilascio non ufficiale. Altre lettere sono spesso, ma non sempre, le iniziali di qualche sviluppatore, questo indica il ramo di sviluppo mantenuto da una certa persona: ck per Con Kolivas, ac per Alan Cox, mm per Andrew Morton, etc.

Il modello di sviluppo del kernel 2.6 ha rappresentato un grosso cambiamento rispetto a quello del kernel 2.5. In precedenza c'era un ramo stabile (2.4) in cui solo cambiamenti piccoli e sicuri venivano introdotti nel codice e un ramo di sviluppo in cui cambiamenti drastici erano permessi. Questo significa che gli utenti avevano a disposizione una versione 2.4 che era sempre profondamente

testata con gli ultimi bug fix e patch di sicurezza, mentre avrebbero dovuto attendere per le nuove caratteristiche, che venivano introdotte nel ramo 2.5. Il ramo 2.5 ad un certo punto e' stato dichiarato stabile e trasformato in 2.6. Invece di aprire un nuovo ramo 2.7 di sviluppo, gli sviluppatori del kernel hanno scelto di continuare ad introdurre grossi cambiamenti nella serie stabile 2.6. Questo presenta il vantaggio di non dover mantenere due rami separati (di cui uno "vecchio") e di avere a disposizione le nuove caratteristiche molto rapidamente, oltre a permettere un piu' ampio test del codice introdotto.

Dall'altro lato, il nuovo modello di sviluppo 2.6 ha voluto anche dire che non esiste alcun ramo stabile per le persone che vogliono solo fix di bug e di sicurezza, senza aver bisogno di nuove caratteristiche. Quindi e' necessario prendere oltre a queste fix anche tutte le nuove caratteristiche introdotte, alcune delle quali potrebbero essere non sufficientemente testate e portare all'instabilita' del sistema. Una parziale correzione a questo e' stata l'introduzione della quarta cifra (y in *2.6.x.y*), essa identifica release puntuali create dallo stable team (Greg Kroah-Hartman, Chris Wright e altri). Il team rilascia aggiornamenti solo per il kernel piu' recente e quindi non risolve la questione della mancanza di una serie stabile. Le distribuzioni Linux mantengono peraltro dei propri kernel a cui un utente puo' affidarsi.

3 Appendice B - I processi

Il concetto di processo e' fondamentale per ogni sistema operativo *multiprogramming*. Un processo viene in genere definito come l'istanza di un programma in esecuzione. I processi sono usualmente chiamati *task* o *threads* nel codice sorgente di Linux.

Il termine processo viene spesso utilizzato con significati diversi, ma in genere indica l'istanza di un programma in esecuzione. E' possibile pensare ad esso come ad una collezione di strutture dati che descrivono il punto fino a cui l'esecuzione del programma e' andata avanti. Cosi' come per gli umani: i processi vengono generati, hanno una vita piu' o meno significativa, generando uno o piu' figli e eventualmente muoiono. Ogni processo ha un solo processo genitore (*parent process*).

Dal punto di vista del kernel, lo scopo di un processo e' quello di agire come un'entita' alla quale devono essere allocate risorse di sistema.

Quando un processo viene creato, e' quasi identico al proprio parent process. Esso riceve una copia logica dello spazio di indirizzamento del processo genitore ed esegue lo stesso codice del processo padre, iniziando con l'istruzione che segue la system call di creazione del processo. Sebbene padre e figlio possano condividere le pagine che contengono il codice del programma (*text*), essi avranno copie separate dei dati (*stack* e *heap*), cosi' che le modifiche fatte alle locazioni di memoria dal figlio sia invisibili al padre e vice versa.

I kernel dei sistemi operativi Unix hanno tradizionalmente utilizzato questo modello semplice, gli Unix moderni invece se ne differenziano. Essi supportano le *multithreaded applications*, ovvero programmi utente che hanno molti flussi di esecuzione relativamente indipendenti e che condividono ampie porzioni delle strutture dati dell'applicazione. In questi sistemi un processo e' composto di molti *user thread* (detti semplicemente *thread*), ognuno dei quali rappresenta un flusso di esecuzione del processo. Attualmente la maggior parte delle applicazioni multithread sono scritte usando un gruppo di funzioni di libreria dette *pthread* (*Posix thread*).

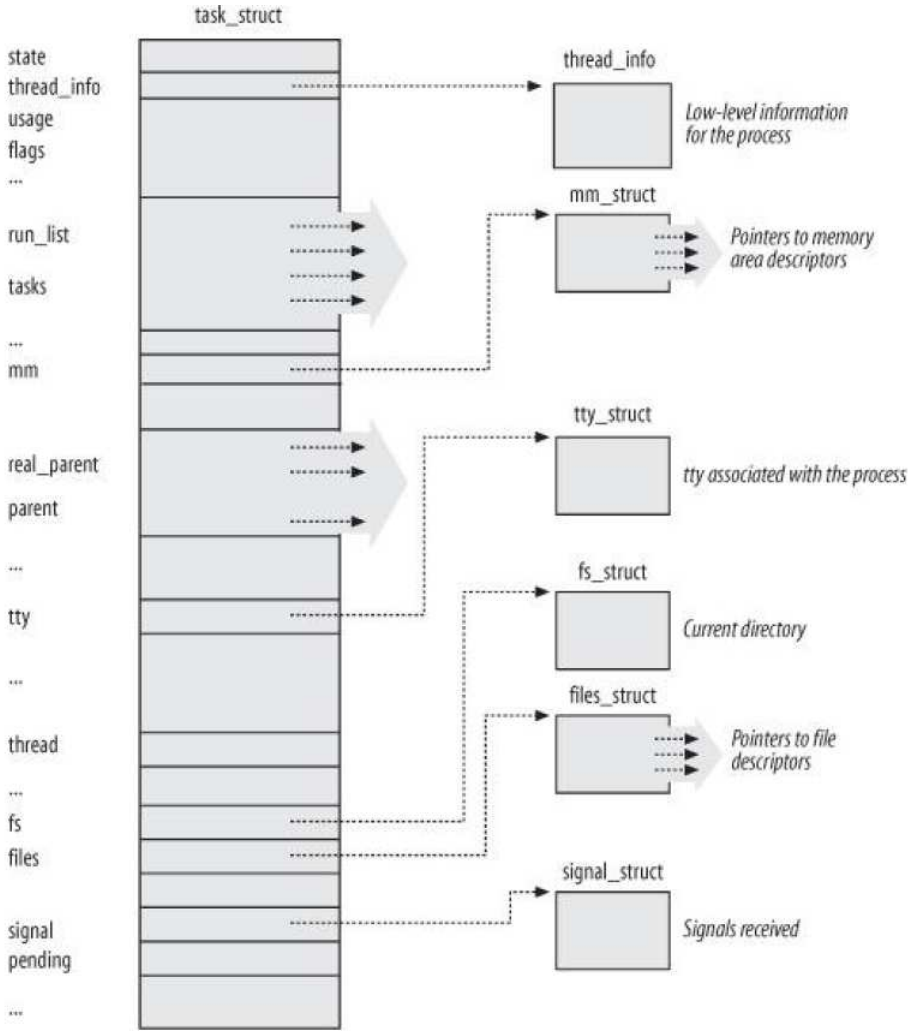
Le versioni piu' vecchie di Linux non fornivano alcun supporto alle applicazioni multithreaded. Dal punto di vista del kernel un'applicazione multithreaded e' un normale processo, I flussi multipli di esecuzione di un'applicazione multithreaded vengono creati, gestiti e schedulati completamente in User Mode, in genere mediante una libreria *pthread* Posix-compliant. Un'implementazione del genere di un'applicazione multithreaded non e' molto soddisfacente.

Linux fornisce i *lightweight process* (LWP) per offrire un supporto migliore per le applicazioni multithreaded. Due lightweight process possono condividere alcune risorse, come l'address space, i file aperti, ecc. Qualora uno di essi modifichi una risorsa condivisa, l'altro immediatamente nota i cambiamenti. Naturalmente a questo punto nasce il problema della sincronizzazione nell'accesso alla risorsa condivisa.

Un metodo molto diretto per implementare un'applicazione multithreaded e' quello di associare un lightweight process con ogni thread. In questo modo, i thread possono accedere allo stesso set di strutture dati dell'applicazione semplicemente condividendo lo stesso memory address space, lo stesso set di file aperti, ecc. Allo stesso tempo ogni thread puo' essere schedulato indipendentemente dal kernel cosi' che uno possa andare nello stato sleep, mentre un altro resta in esecuzione.

Le applicazioni multithreaded sono gestite meglio dai kernel che supportano i *thread group*, ovvero un set di lightweight process che implementano un'applicazione multithreaded e che agiscono come un tutt'uno rispetto a system call come: getpid(), kill() e _exit().

Per gestire i processi il kernel necessita una chiara visione di cosa ogni processo stia facendo. Deve sapere, per esempio, la priorita' del processo, se e' in esecuzione su una CPU (*running*) oppure e' bloccato in attesa di un evento, quale address space gli e' stato assegnato, ecc. Questo e' il ruolo del *process descriptor*, una struttura di tipo *task_struct* i cui campi contengono tutte le informazioni relative ad un singolo processo. Come depositario di una quantita' cosi' grande di informazioni, il process descriptor e' piuttosto complesso. Oltre al grande numero di campi che contengono gli attributi del processo, il process descriptor contiene diversi puntatori ad altre strutture dati che, a loro volta, contengono puntatori ad ulteriori strutture.



Le sei strutture dati sulla destra della figura fanno riferimento a risorse

specifiche possedute dal processo.

3.1 Lo stato dei processi

Come il nome implica, il campo *state* del process descriptor descrive cosa sta accadendo ad un processo in un certo istante. Esso consiste di un array di flags, ognuno dei quali descrive un possibile stato del processo. Allo stato attuale, in Linux questi stati sono mutuamente esclusivi e quindi un solo flag di *state* e' sempre attivo, i restanti sono inattivi. I possibili process state sono i seguenti:

- **TASK_RUNNING**: il processo e' in esecuzione sulla CPU oppure sta aspettando di essere eseguito.
- **TASK_INTERRUPTIBLE**: il processo e' sospeso (*sleeping*) finche' alcune condizioni non diventano vere. Esempi delle condizioni che possono risvegliare un processo (farlo ritornare allo stato TASK_RUNNING) sono: l'arrivo di un hardware interrupt, il rilascio di una risorsa di sistema che il processo sta aspettando o l'arrivo di un segnale.
- **TASK_UNINTERRUPTIBLE**: Come per TASK_INTERRUPTABLE, solo che l'arrivo di un segnale al processo lascia il suo stato inalterato. E' uno stato usato raramente. Puo' essere pero' utile sotto certe specifiche condizioni nelle quali un processo deve attendere fino all'accadere di un certo evento senza essere interrotto.
- **TASK_STOPPED**: L'esecuzione del processo e' stata fermata. Il processo entra in questo stato dopo aver ricevuto un SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU.
- **TASK_TRACED**: L'esecuzione del processo e' stata fermata da un debugger. Quando un processo viene monitorato da un altro, ogni segnale puo' mettere il processo nello stato TASK_TRACED.

Due ulteriori stati possono essere salvati sia nel campo *state* che in quello *exit_state* del process descriptor. Come il nome suggerisce, un processo raggiunge questi due stati solo quando l'esecuzione e' terminata:

- **EXIT_ZOMBIE**: l'esecuzione del processo e' terminata, ma il processo parent non ha ancora eseguito una system call *wait4()* o *waitpid()* per raccogliere le informazioni sul processo morto. Prima che la system call della famiglia wait() sia stata eseguita, il kernel non puo' scartare i dati contenuti nel process decriptor del processo morto perche' il parent process potrebbe averne bisogno.
- **EXIT_DEAD**: lo stato finale, il processo e' stato rimosso dal sistema perche' il parent process ha appena eseguito una system call *wait4()* o *waitpid()* su di esso. Il cambio dello stato da EXIT_ZOMBIE a EXIT_DEAD evita che si verifichino race conditions dovute ad altri thread che possono eseguire le system call della famiglia wait() sullo stesso processo.

Il kernel utilizza due macro *set_task_state* e *set_current_state* per impostare il valore dello state di un processo.

3.2 Identificare un processo

Come regola generale, ogni contesto di esecuzione che possa essere schedulato indipendentemente deve avere il suo process descriptor, quindi anche i lightweight process, che condividono un ampia porzione delle loro kernel structures, hanno una propria struttura *task_struct*.

La stretta corrispondenza uno-a-uno fra il processo e il suo descrittore fa dell'indirizzo a 32 bit della struttura *task_struct* un utile mezzo per il kernel per l'identificazione dei processi. La maggior parte dei riferimenti ai processi che il kernel fa avvengono attraverso puntatori ai process descriptor.

Da un altro punto vista i sistemi operativi Unix permettono agli utenti di referenziare i processi per mezzo di un numero chiamato *Process ID* o PID che viene salvato nel campo *pid* del process descriptor. I PID sono assegnati sequenzialmente: il PID di un processo appena creato e' normalmente il PID dell'ultimo processo creato piu' uno. Naturalmente c'e' un limite superiore al valore del PID, quando questo viene raggiunto il kernel ricicla i valori piu' bassi non utilizzati. Per default il massimo PID e' 32767, questo valore puo' essere modificato agendo sul valore contenuto nel file virtuale */proc/sys/kernel/pid_max*.

Quando vengono riciclati i PID number, il kernel deve gestire una bitmap detta *pidmap_array* che identifica quali sono i PID attualmente utilizzati e quali liberi. Siccome un page frame contiene 32768 bit nelle architetture a 32bit, la bit map viene memorizzata in una sola pagina.

Linux associa un PID diverso ad ogni processo o lightweight process nel sistema. Questo approccio consente la massima flessibilita', perche' ogni execution context nel sistema puo' essere singolarmente identificato. D'altro canto i programmatori si aspettano che i thread che fanno parte dello stesso gruppo abbiano un PID comune. Per esempio dovrebbe essere possibile inviare un segnale specificando un PID in modo che influenzi tutti i thread nel gruppo (lo standard POSIX definisce uno stesso PID per tutti i thread di un'applicazione multithreaded).

Per soddisfare questo standard Linux fa uso dei thread group. L'identificatico che viene condiviso fra i thread e' il PID del group leader, ovvero il PID del primo lightweight process del gruppo, che viene salvato nel campo *tgid* del process descriptor. La funzione getpid() restituisce proprio questo campo invece che quello *pid* cosi' tutti i thread dell'applicazione condividono lo stesso identificativo. La maggior parte dei processi che appartengono ad un thread group consistono di un solo membro, che, come group leader, ha il campo *tgid* uguale a quello *pid*.

3.3 Switch dei processi

Per controllare l'esecuzione dei processi, il kernel deve essere in grado di sospendere il processo in esecuzione sulla CPU e di riprendere l'esecuzione di qualche altro processo che era stato preventivamente sospeso. Quest'attivita' e' conosciuta con il nome di *process switch, task switch* o *context switch*.

Sebbene ogni processo possa avere il proprio address space, tutti i processi condividono i registri della CPU. Quindi prima di riprendere l'esecuzione di un processo, il kernel deve assicurare che ognuno di questi registri venga caricato con il valore che aveva quando il processo e' stato sospeso.

Il set di dati che deve essere caricato nei registri prima che il processo riprenda la propria esecuzione sulla CPU e' detto *hardware context*. Esso costituisce un sottoinsieme del process execution context, che include tutte le informazioni necessarie al processo in esecuzione. In Linux una parte dell'hardware context viene salvato nel process descriptor. Mentre la restante parte viene salvata nel Kernel Mode Stack.

Un process switch puo' avvenire solo in alcuni punti ben definiti: la funzione *schedule()*. Essenzialmente, ogni process switch consiste di due passi:

1. Sostituire la Page Global Directory per installare un nuovo address space.
2. Sostituire il Kernel mode stack e l'hardware context.

3.4 Lo scheduling dei processi

Come ogni sistema time-sharing, Linux produce il magico effetto di un esecuzione apparentemente simultanea di piu' processi, passando da uno ad un altro in un tempo molto breve. Lo *scheduling* ha a che fare con "*quando*" cambiare processo e "*quale*" processo scegliere.

L'algoritmo di scheduling degli Unix tradizionali deve soddisfare molti obiettivi che possono confliggere: un tempo rapido di risposta, buon throughput per i processi in background, evitare la process starvation (ovvero l'attesa da parte di un processo per risorse che sono possedute da un'altro), riconciliare le necessita' dei processi a bassa e ad alta priorita', ecc. L'insieme di regole utilizzate per determinare quando e come selezionare un nuovo processo e' detta *scheduling policy*.

Lo scheduling in Linux e' basato su una tecnica di *time sharing*: vari processi possono girare in "time multiplexing" perche' il tempo della CPU viene diviso in *slice*, una per ogni ogni runnable process[3]. Naturalmente, un singolo processore puo' eseguire solo un processo in un certo istante. Se un processo in esecuzione in un certo momento non e' terminato quando la propria *time slice* (o *quantum*) termina, deve avvenire un process switch. Il time sharing fa affidamento ai timer interrupt e quindi e' trasparente nei confronti dei processi, ovvero nessuna linea di codice deve essere inserita nei programmi per assicurarsi che venga utilizzata la CPU time sharing.

La policy di scheduling e' basata anche sulla classificazione dei processi in base alla loro priorita'. Algoritmi complessi sono spesso utilizzati per derivare la priorita' corrente di un processo, il risultato finale e' pero' sempre lo stesso: ogni processo viene associato con un valore che dice allo scheduler "quanto appropriato" sia far girare il processo su una CPU.

La priorita' dei processi in Linux e' dinamica. Lo scheduler tiene traccia di cosa i processi stiano facendo e aggiusta la loro priorita' periodicamente; in questo modo i processi che hanno avuto l'uso della CPU negato per un lungo intervallo di tempo vengono "accelerati" incrementando dinamicamente la loro priorita'. Allo stesso modo, i processi che sono stati eseguiti per un lungo periodo vengono "penalizzati" diminuendo la loro priorita'.

Quando si parla di scheduling, i processi vengono tradizionalmente classificati in *I/O-bound* o *CPU-bound*. I primi fanno un uso pesante dei dispositivi di

[3] I processi in stato stopped e suspendend non possono essere selezionati dall'algoritmo di scheduling per essere messi in esecuzione su una CPU

I/O e trascorrono la maggior parte del tempo in attesa che le operazioni di I/O siano completate; gli altri portano avanti applicazioni di number-crunching che richiedono molto tempo della CPU.

Una classificazione alternativa distingue tre classi di processi:

- *Processi interattivi*: Sono quelli che interagiscono costantemente con i propri utenti e quindi passano molto tempo in attesa di pressione di tasti, movimenti del mouse, ecc. Quando l'input viene ricevuto, il processo deve essere risvegliato velocemente, o l'utente percepira' il sistema come poco responsivo. Tipicamente il ritardo medio deve cadere fra i 50 e i 150 ms. Anche la deviazione standard di questo ritardo dovrebbe essere limitata o l'utente percepira' il sistema come casuale nel comportamento.

- *Processi Batch*: Sono quelli che non necessitano dell'interazione dell'utente e che quindi sono di solito eseguiti in background. Siccome questi processi non necessitano di essere molto responsivi, sono in genere penalizzati dallo scheduler.

- *Processi real-time*: Sono quelli che hanno requisiti temporali molto stringenti. Essi non dovrebbero mai essere bloccati da processi a priorita' piu' bassa e dovrebbero avere un breve tempo di risposta garantito entro una minima varianza.

Le due classificazioni sono in qualche modo indipendenti. I processi real-time sono esplicitamente riconosciuti dall'algoritmo di scheduling, mentre non esiste un modo semplice per distinguere i processi batch e quelli interattivi. Lo scheduler Linux-2.6 implementa un sofisticato algoritmo euristico basato sul comportamento che ha tenuto in passato un processo per decidere come il processo stesso debba essere considerato. Naturalmente lo scheduler tende a favorire i processi interattivi rispetto a quelli batch.

3.4.1 Process preemption

I processi Linux sono *preemptable*. Quando un processo entra nello stato TASK_RUNNING, il kernel controlla se la sua priorita' dinamica e' superiore a quella del processo attualmente in esecuzione. Se lo e', l'esecuzione del processo corrente viene sopesa e lo scheduler viene invocato per selezionare un altro processo da mandare in esecuzione (in genere il processo che e' appena diventato *runnable*). Naturalmente un processo puo' anche essere *preempted* quando il suo time quantum termina. Quando questo accade il TIF_NEED_RESCHED flag nella struttura thread_info del processo corrente viene attivato, cosi' lo scheduler viene invocato e il timer interrupt handler termina.

Va notato che un processo preeempted non e' sospeso, perche' resta nello stato TASK_RUNNING, esso si limita a non utilizzare la CPU. Oltre a questo va notato che il kernel Linux 2.6 e' preemptive, questo significa che un processo puo' essere preempted sia quando e' in esecuzione in User Mode che quando e' in esecuzione in Kernel Mode.

3.4.2 Durata di un *quantum*

La durata del *quantum* e' fondamentale per le performance del sistema: non deve essere ne troppo lungo ne troppo corto.

Se il quantum medio e' troppo breve, l'overhead causato dallo switch dei processi diventa eccessivamente alto. Al contrario se questa durata e' troppo lunga, i processi non sembrano piu' eseguiti simultaneamente. In genere si tende a credere che un lungo quantum degradi il tempo di risposta delle applicazioni interattive, in realta' questo e' falso. I processi interattivi hanno infatti un priorita' relativamente alta, quindi essi sotituiranno (*preempt*) i processi batch, a prescindere da quanto sia lunga la durata del quantum.

La scelta del quantum medio e' sempre un compromesso. La scelta effettuata da Linux e' quella di scegliere una durata che sia il piu' lunga possibile, mantenendo pero' un buon tempo di risposta.

3.5 L'algortimo di scheduling

L'algoritmo di scheduling utilizzato dalla precedenti versioni di Linux era piuttosto semplice: ad ogni process switch il kernel fa uno scan della lista dei processi *runnable*, calcola la priorita' e seleziona il "best process" da eseguire. Il principale difetto di questo algoritmo e' che il tempo speso per scegliere il "best process" dipende dal numero dei processi in stato runnable[4]. L'algoritmo diventa allora troppo costoso per i sistemi high-end che fanno girare migliaia di processi.

L'algoritmo di scheduling di Linux 2.6 e' molto piu' sofisticato, per specifiche di progetto esso reagisce meglio al crescere del numero dei runnable process, perche' seleziona il processo da mettere in esecuzione in tempo costante[5] indipendentemente dal numero dei processi in stato runnable. Inoltre l'algoritmo scala bene al crescere del numero di CPU perche' ogni processore ha la sua coda di processi runnable. Oltre a questo il nuovo algoritmo distingue meglio i processi batch da quelli interattivi.

Lo scheduler finisce sempre per selezionare un processo da eseguire, infatti c'e' sempre almeno un processo eseguibile, il processo *swapper* il cui PID e' 0 e che la CPU esegue quando non ci sono altri processi. Ogni CPU di un sistema multiprocessore ha il proprio processo *swapper*.

Ogni processo Linux viene sempre schedulato in base ad una delle seguenti *scheduling class*:

- **SCHED_FIFO**: Un processo real-time First-in, First-Out. Quando lo scheduler assegna la CPU al processo, lascia il process descriptor nella sua posizione nella runqueue. Se nessun altro processo real-time ad alta priorita' e' in stato runnable, il processo continuera' ad usare la CPU per quanto vuole, anche se esistono altri processi real-time con la stessa priorita' in stato runnable.
- **SCHED_RR**: Un processo real-time Round Robin. Quando lo scheduler assegna la cpu al processo mette il process decriptor alla fine della runqueue.
- **SCHED_NORMAL**: Un normale processo time-shared.

[4] L'algoritmo e' di tipo O(n)
[5] L'algoritmo e' di tipo O(1)

3.5.1 Scheduling dei processi convenzionali

Ogni processo convenzionale la la propria *priorita' statica*, che e' un valore utilizzato dallo scheduler per classificare un processo rispetto agli altri processi convenzionali nel sistema. Il kernel rappresenta la priorita' statica di un processo convenzionale con un numero in un range da 100 (priorita' piu' alta) a 139 (priorita' piu' bassa); la priorita' statica decresce al crescere del valore del numero.

Un nuovo processo eredita sempre la priorita' statica del suo parent. Un utente puo' comunque cambiare la priorita' statica dei processi in suo possesso passando alcuni "nice values" alle system call nice() e setpriority().

Base time quantum La priorita' statica essenzialmente determina il *base time quantum* di un processo, ovvero la durata del time quantum assegnata al processo quando ha esaurito il suo time quantum precedente. Come regola: piu' alta la priorita' statica (valore numerico piu' basso) piu' lungo il time quantum.

Priorita' dinamica e average sleep time A fianco della priorita' statica, un processo convenzionale possiede anche una *priorita' dinamica*, che e' un valore in un range da 100 (priorita' piu' alta) a 139 (priorita' piu' bassa). La priorita' dinamica e' il numero a cui lo scheduler guarda quando seleziona un nuovo processo da mettere in esecuzione, esso e' un valore legato alla priorita' statica.

Active e expired process Anche se i processi convenzionali che hanno una priorita' statica piu' alta ottengono slice di tempi di cpu piu' grandi, essi non dovrebbero completamente escludere i processi con priorita' statica piu' bassa. Per evitare la process starvation, quando un processo esaurisce il suo time quantum, puo' essere sostituito da un processo a piu bassa priorita' il cui time quantum ancora non si sia esaurito. Per implementare questo meccanismo lo scheduler mantiene due set disgiunti di processi runnable:

- **Active process**: I processi runnable che non hanno esaurito il proprio time quantum e quindi hanno la possibilita' di essere eseguiti
- **Expires process**: Questi runnable process hanno esaurito il proprio time quantum e quindi non possono essere eseguiti finche' tutti gli active process non sono diventati expire

Lo schema reale e' ancora piu' complesso perche' lo scheduler cerca di massimizzare le performance dei processi interattivi. Un active process batch che finisce il suo time quantum diventa sempre expired, mentre un active process interattivo che finisce il suo time quanum in genere resta attivo: lo scheduler riempie di nuovo il time quantum e lo lascia nel set degli active process. Comunque lo scheduler sposta un processo interattivo che finisce il suo time quantum nel set di expired process se il piu' vecchio expired process ha gia' atteso a lungo oppure se un expired process ha una priorita' statica piu' alta dell'interactive process. Come conseguenza il set di active process diventera' eventualmente vuota e gli expired process avranno una possibilita' di essere eseguiti.

3.5.2 Scheduling dei processi real-time

Ogni processo real-time viene associato con una *priorita' real-time* che e' un valore compreso fra 1 (massima priorita') e 99 (minima priorita'). Lo scheduler favorisce sempre un processo ad alta priorita' in stato runnable su un processo a bassa priorita'; in altre parole, un processo real-time inibisce l'esecuzione di ogni processo a bassa priorita' finche' resta runnable. Al contrario dei processi convenzionali, i processi real-time sono sempre considerati active.

Se diversi processi real-time in stato runnable hanno la stessa massima priorita', lo scheduler sceglie il processo che appare prima nella corrispondente lista della runqueue locale della CPU.

Un processo real-time viene sostituito da un altro processo solo quando accade uno dei seguenti eventi:

- Il processo viene sostituito (preempted) da un altro che ha una piu' alta priorita' real-time
- Il processo porta avanti un'operazione bloccante e viene messo in stato sleep (TASK_INTERRUPTABLE o TASK_UNINTERRUPTABLE)
- Il processo e' stoppato (TASK_STOPPED o TASK_TRACED) o ucciso (EXIT_ZOMBIE o EXIT_DEAD)
- Il processo rilascia volontariamente la CPU invocando la system call sched_yield()
- Il processo e' Round Robin real-time (SCHED_RR) e ha esaurito il suo time quantum

Le system call nice() e setpriority() quando applicate a un processo real-time Round Robin, non cambia la priorita' real-time ma piuttosto la durata del base time quantum. Infatti, la durata del base time quantum dei processi Round Robin real-time non dipende dalla priorita' real-time, ma piuttosto dalla priorita' statica del processo.

4 Appendice C - La gestione della memoria

4.1 Memory addressing

Sulle architetture Intel 80x86 la gestione della memoria viene fatta da tutti i sistemi operativi moderni sfruttando alcune circuiterie che offre il processore stesso.

I programmatori in genere fanno riferimento ai *memory address* come il metodo con cui accedono alle celle di memoria. Sui processori 80x86 dobbiamo distinguere tre tipi di indirizzi:

- **Indirizzi Logici**: Ogni indirizzo logico consiste di un *segmento* e di un *offset* (o *displacement*), che denota la distanza dall'inizio del segmento dell'indirizzo.
- **Linear Address**: Un intero a 32 bit che puo' essere utilizzato per referenziare fino a 4GB di memoria. Detti anche *virtual address*.
- **Physical Address**: Utilizzati per indirizzare celle nei chip di memoria. Essi sono rappresentatati da indirizzi a 32 o 36 (PAE) bit.

La Memory Management Unit (MMU) trasforma un indirizzo logico in uno lineare per mezzo di un'opportuna circuiteria detta *segmentation unit*, un secondo circuito hardware detto *paging unit* trasforma l'indirizzo lineare in quello fisico.

Linux utilizza la segmentazione in un modo molto limitato. Segmentazione e paging sono in qualche modo ridondanti, perche' entrambi possono essere utilizzati per separare lo spazio di indirizzamento fisico, la segmentazione puo' assegnare un differente linear address space ad ogni processo, mentre la paginazione puo' mappare lo stesso linear address space in physical address space diversi. Linux preferisce la paginazione alla segmentazione per le seguenti ragioni:

- La gestione della memoria e' piu' semplice quando tutti i processi utilizzano lo stesso valore del segment register, ovvero quando condividono lo stesso set di linear address.
- Uno degli obiettivi del progetto di Linux e' la portabilita' ad un ampio range di architetture e i sistemi RISC hanno un supporto per la segmentazione molto limitato.

Vediamo come viene utilizzato il paging in hardware. L'unita' di paginazione traduce gli indirizzi lineari in indirizzi fisici. Uno dei compiti principali dell'unita' e' il controllo del tipo di accesso richiesto nei confronti dei diritti di accesso del linear address. Se l'accesso alla memoria non e' valido, viene generata un'eccezione di *Page Fault*.

Per motivi di efficienza, gli indirizzi lineari sono raggruppati in intervalli di lunghezza fissa dette *pagine* (*pages*); indirizzi lineari contigui in una pagina vengono mappati in indirizzi fisici contigui. In questo modo il kernel puo'

specificare l'indirizzo fisico e i diritti di accesso di una pagina invece di quelli di tutti gli indirizzi lineari inclusi in essa. Convenzionalmente, viene utilizzato il temine *page* per riferirsi sia al set di indirizzi lineari che ai dati contenuti in questo gruppo di indirizzi.

L'unita' di paginazione immagina tutta la RAM come divisa in *page frame* di lunghezza fissa (detti anche *physical page*). Ogni page frame contiene una pagina, ovvero la lunghezza di un page frame coincide con quella di una pagina. Un page frame e' un elemento della memoria principale e quindi e' un area di memorizzazione. E' importante distinguere una pagina da un page frame: la prima e' solo un blocco di dati, che puo' essere memorizzata in un page frame o su disco.

4.2 Memory management

Linux utilizza quindi la circuiteria di segmentazione e di paginazione dei processori 80x86 per tradurre gli indirizzi logici in indirizzi fisici. Una parte della memoria viene permanentemente assegnata al kernel e utilizzata per salvare sia il codice che le strutture dati del kernel stesso.

La restante parte della memoria viene detta *dynamic memory*. Essa costituisce una notevole risorsa , necessaria non solo per i processi ma anche per il kernel stesso. Infatti, le performance del sistema dipendono dall'efficienza della gestione della memoria dinamica. Quindi, tutti i sistemi operativi multitasking cercano di ottimizzare l'uso della memoria dinamica, assegnandola solo quando necessario e liberandola il prima possibile.

4.2.1 Page frame management

I processori di classe Intel Pentium possono utilizzare due diverse dimensioni di page frame: *4kB* e *4MB*. Linux adotta la dimensione di 4kB come dimensione standard dell'unita' di allocazione. Questo semplifica le cose per due ragioni:

- Le eccezioni di Page Fault prodotte dall'unita' di paginazione vengono interpretate facilmente. O la pagina esiste, ma il processo non puo' referenziarla oppure la pagina non esiste. Nel secondo caso, il memory allocator deve trovare un page frame da 4kB libero e assegnarlo al processo.
- Sebbene sia 4kB che 4MB siano multipli di tutte le disk block size, i trasferimenti di dati fra la memoria principale e il disco sono nella maggior parte dei casi piu' efficienti quando viene utilizzata la dimensione minore.

4.2.2 Page descriptors

Il kernel deve tenere traccia dello stato corrente dei page frame. Deve, per esempio, essere in grado di distinguere i page frame che sono utilizzati per contenere le pagine che appartengono ai processi da quelle che contengono il codice del kernel. Allo stesso modo deve essere in grado di determinare se un page frame e' libero. Un page frame in dynamic memory e' libero quando non contiene nessun dato utile, mentre non e' libero quando contiene dati di un processo User Mode, dati di una software cache, dati allocati dinamicamente alle strutture del kernel, dati bufferizzati di un device driver, codice di un modulo del kernel, ecc.

Le informazioni del page frame sono mantenute in un page descriptor di tipo *page*. Tutti i descrittori sono memorizzati nell'array *mem_map*.

4.2.3 Le Memory Zones

In un'architettura ideale un page frame e' un'unita' di memorizzazione in RAM che puo' essere usata per tutto: memorizzazione dati utente e kernel, buffer di disk data, ecc. Tutti i tipi di pagine di dati possono essere memorizzate in un page frame, senza alcuna limitazione.

Comunque, un'architettura reale ha dei vincoli hardware che possono limitare il modo in cui i page frame possono essere utilizzati. In particolare il kernel Unix deve operare con due vincoli delle architetture 80x86:

- I processori per il DMA per il vecchio bus ISA hanno grosse limitazioni, possono infatti indirizzare solo i primi 16MB di RAM.
- Nei moderni computer a 32 bit con molta RAM, la CPU non puo' accedere direttamente a tutta la memoria fisica perche' il linear address space e' troppo piccolo.

Per gestire questi limiti, Linux 2.6 suddivide la memoria fisica di ogni nodo di memoria in tre *zone*. Nelle architetture 80x86 di tipo UMA (Uniform Memory Architecture) le zone sono:

- **ZONE_DMA**: Contiene i page frame di memoria sotto i 16MB.
- **ZONE_NORMAL**: Contiene i page frame di memoria fra i 16MB e gli 896MB.
- **ZONE_HIGHMEM**: Contiene i page frame di memoria sopra gli 896MB.

Le zone ZONE_DMA e ZONE_NORMAL includono i page frame "normali" che possono essere indirizzati direttamente dal kernel attraverso il linear mapping del quarto gigabyte di linear address space. Al contrario la zona ZONE_HIGHMEM include i page frame che non possono essere indirizzati direttamente dal kernel.

4.3 Il Buddy System Algorithm

Il kernel deve stabilire una strategia robusta ed efficiente per allocare gruppi di page frame contigui. Per fare questo si scontra con il problema dell'*external fragmentation*. Ovvero potrebbe accedere che sia impossibile allocare un grande blocco di page frame contigui, sebbene ci siano abbastanza page frame liberi.

Esistono due metodi per evitare l'external fragmentation:

- Utilizzare la circuiteria di paginazione per mappare gruppi di page frame non contigui in un intervallo contiguo di linear address.
- Sviluppare una tecnica per tenere traccia di blocchi di page frame liberi e contigui, cercando di evitare al massimo la divisione di un grande blocco libero per soddisfare la richiesta per uno piu' piccolo.

Il secondo approccio e' quello seguito dal kernel e si basa sul *Buddy System Algorithm*. Tutte le pagine libere sono raggruppate in 11 liste di blocchi che contengono gruppi di 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 e 1024 page frame contigui, rispettivamente. La piu' grande richiesta di 1024 page frame corrisponde ad un blocco di 4MB di RAM contigua. L'indirizzo fisico del primo primo page frame del blocco e' un multiplo del group size (per esempio per il gruppo di 16 page-frame-block e' un multiplo di $16x2^{12}$ovvero 16x4k).

L'algoritmo opera nel seguente modo: assumiamo che ci sia una richiesta per un gruppo di 256 page frame contigui. L'algoritmo controlla, per prima cosa, se esiste un blocco libero nella lista da 256-page-frame. Se non esiste, l'algortimo va a vedere per il prossimo blocco di dimensione maggiore, un blocco libero nella lista da 512-page-frame. Se un blocco di tale tipo esiste, vengono allocati 256 dei 512 page frames per soddisfare la richiesta e inserisce i rimanenti 256 page frame nella lista dei blocchi da 256-page-frame liberi. Se non esiste alcun blocco libero da 512-page-frame, il kernel passa al successivo blocco piu' grande (1024-page-frame). Se tale blocco esiste, esso alloca 256 dei 1024 page frame per soddisfare la richiesta, inserisce 512 dei rimanenti 768 page frame nella lista dei blocchi liberi da 512-page-frame e inserisce gli ultimi 256 page frame nella lista dei blocchi liberi da 256-page-frame. Se la lista dei blocchi liberi da 1024-page-frame e' vuota, l'algoritmo rinuncia e notifica una condizione d'errore.

L'operazione inversa, il rilascio dei page frame, da' ragione al nome dell'algoritmo. Il kernel cerca di unire coppie di buddy block liberi di dimensione *b* in un singolo blocco di dimensione *2b*. Due blocchi sono considerati buddy, se:

- Entrambi hanno la stessa dimensione
- Sono allocati in physical address contigui
- L'indirizzo fisico del primo page frame del primo blocco e' un multiplo di $2xBx2^{12}$

L'algoritmo e' iterativo. Se riesce ad unire dei blocchi rilasciati, raddoppia b e va in esecuzione di nuovo per cercare di creare blocchi piu' grandi.

4.4 Memory Area Management

Le *memory area* sono sequenze di celle di memoria che hanno indirizi fisici contigui e lunghezza arbitraria.

Il buddy system algorithm adotta il page frame come la memory area base. Questo e' ottimo quando si abbia a che fare con richieste di memoria relativamente grandi, ma non funziona bene quando le aree di memoria richieste siano piccole, decine o centinaia di byte.

Chiaramente, sarebbe un grosso spreco allocare un intero page frame per memorizzare pochi byte. Un approccio migliore al problema consiste nell'introduzione di nuove strutture dati che descrivono come le piccole memory area siano allocate nello stesso page frame. In questo senso viene introdotto il problema dell'*internal fragmentation*. Causato da una discrepanza fra la dimensione della richiesta di memoria e la dimensione dell'area di memoria allocata per soddisfare la richiesta.

4.4.1 Lo Slab Allocator

Eseguire un memory area allocation algorithm al di sopra del buddy algorithm non e' particolarmente efficiente. Un miglior algoritmo e' stato derivato dallo schema dello *slab allocator* adottato per la prima volta da Sun Microsystems per Solaris 2.4. Esso e' basato sui seguenti punti:

- Il tipo di dati che devono essere memorizzati possono influenzare come le aree di memoria vengono allocate. Il concetto di slab allocator espande quest'idea e vede le memory area come *objects* che consistono sia di un set di strutture dati sia di una coppia di metodi detti *constructror* e *descructor*. Il primo inizializza l'area di memoria e l'altro la deinizializza.
  Per evitare di inizializzare continuamente gli oggetti, lo slab allocator non scarta gli oggetti che sono stati allocati e poi rilasciati, ma al contrario li salva in memoria. Quando viene richiesto un nuovo oggetto, esso puo' essere preso dalla memoria senza essere reinizializzato.

- Le funzioni del kernel tendono a richiedere memory area dello stesso tipo con frequenza (per esempio il process descriptor, o l'open file object quando un processo viene creato). Siccome i processi sono creati e distrutti frequentemente, senza lo slab allocator il kernel perderebbe tempo per allocare e deallocare i page frame che contengono le stesse aree di memoria frequentemente. Lo slab allocator permette di salvarli in cache e riutilizzati rapidamente.

- Le richieste per memory area possono essere classificate in base alla loro frequenza. Richieste di una particolare dimensione, che ci si attende avvengano frequentemente, possono essere gestite piu' efficientemente creando un set di objects "special-purpose" che abbiano la giusta dimensione, evitando cosi' la frammentazione interna. Allo stesso modo, dimensioni che siano state riscontrate raramente possono essere gestite per mezzo di uno schema di allocazione basato su oggetti in una serie geometrica (come per esempio le potenze di 2), anche se questo approccio conduce alla frammentazione interna.

- Introdurre gli objects le cui dimensioni non siano geometricamente distribuite comporta anche un altro vantaggio, gli indirizzi iniziali delle strutture dati sono meno soggetti a concentrarsi ad indirizzi fisici i cui valori sono potenze di 2. Questo comporta migliori performance anche nell'uso delle cache hardware dei processori.

- Le performance delle hardware cache creano un ulteriore ragione per limitare le chiamate al buddy algorithm il piu' possibile. Ogni chiamata ad una funzione del buddy system, invalida ("dirties") l'hardware cache, incrementando il livello medio del tempo di accesso alla memoria. L'impatto di una funzione del kernel su una cache hardware e' chiamata *function footprint*; viene definita come la percentuale di cache sovrascritta dalla funzione quando termina. Chiaramente, ampie footprint comportano una piu' lenta esecuzione del codice eseguito subito dopo la kernel funcion, visto che la cache e' riempita di informazioni non utili.

Lo slab allocator raggruppa gli oggetti in *cache*. Ogni cache e' un "magazzino" di oggetti dello stesso tipo. L'area di memoria che contiene una cache viene

divisa in *slab*, ogni slab consiste di uno o piu' page frame contigui che possono contenere oggetti allocati o liberi.

5 Appendice D - I/O scheduling

Il kerbel Linux 2.6 include un gruppo di I/O scheduler selezionabili. Gli I/O scheduler controllano il modo in cui il kernel impartisce (*commit)* ai dischi le operazioni di lettura e scrittura. Lo scopo per cui vengono forniti diversi scheduler e' di permettere una migliore ottimizzazione per diverse classi di workload.

Senza un I/O scheduler, il kernel non farebbe altro che passare ogni richiesta al disco nell'ordine in cui l'ha ricevuta. Questo potrebbe portare, fra le altre cose, ad una prematura usura del disco; per esempio se un processo sta leggendo su una parte del disco, mentre un differente processo sta scrivendo in un'altra, le testine sarebbero costrette a muoversi continuamente avanti e indietro fra le due aree per ogni operazione. Lo scopo principale degli I/O scheduler e' quello di ottimizzare il *disk access time*.

Un I/O scheduler puo' utilizzare le seguenti tecniche per migliorare le performance:

- *Request merging*: Lo scheduler unisce le richieste "adiacenti", in modo da ridurre il *disk seeking*.
- *Elevator*: Lo scheduler ordina le richieste in base alla locazione fisica sul block device e e cerca di fare il seek in una direzione finche' sia possibile.
- *Prioritarisation*: Lo scheduler ha il completo controllo su come gestire la priorita' delle richieste e puo' alterarle in molti modi,

Gli scheduler disponibili sono 4:

- **Noop Scheduler**: Questo scheduler implementa solo l'unione (merging) delle richieste. Il NOOP scheduler e' semplicemente una FIFO e utilizza il minimo numero di istruzioni della CPU per I/O per realizzare le funzionalita' base di merging e ordinamento necessarie a completare l'I/O. L'idea di fondo e' che le performance dell'I/O siano state ottimizzate a livello di block device o di controller intelligente.
- **Anticipatory I/O scheduler**: L'anticipatory scheduler era quello di default nei vecchi kernel 2.6. Implementa il request merging, un one-way elevator, batching delle read e write request e cerca di fare alcune anticipatory read cercando di anticipare i dati di cui l'utente potrebbe aver bisogno. L'ottimizzazione dal punto fisico viene fatta cercando di evitare i movimenti delle testine il piu' possibile. Uno svantaggio e' che questo sistema mal si adatta alle performance di database e sistemi di storage. L'Anticipatory elevator introduce un ritardo controllato prima di fare il dispatching dell'I/O, cercando di aggregare e/o riordinare le richieste migliorando la localita' e riducendo le operazioni di disk seek. L'algoritmo serve ad ottimizzare i sistemi con dischi piccoli o lenti. Uno svantaggio puo' essere una maggiore latenza dell'I/O.
- **Deadline scheduler**: Il deadline scheduler implementa il request merging, un one-way elevator e pone una deadline su tutte le operazioni per prevenire la resource starvation. Il deadline elevator utilizza un algoritmo deadline per minimizzare la latenza dell'I/O per una certa richiesta. Lo

scheduler fornisce un comportamento simil real-time e utilizza una policy di round robin per cercare di essere equo fra molte richieste di I/O e evitare la process starvation. Utilizzando cinque code di I/O, lo scheduler riordina aggressivamente le richieste per incrementare le performance.

- **Complete fair queueing scheduler (CFQ)**: E' lo scheduler di default dei nuovi kernel 2.6. Come dice il nome, CFQ mantiene una coda di I/O per processo di tipo scalabile e cerca di distribuire la banda di I/O disponibile in modo equanime fra tutte le richieste di I/O. CFQ e' ideale per i medi e grandi sistemi multiprocessore e per sistemi che richiedano una performance dell'I/O bilanciata su piu' controller.

Riferimenti bibliografici

[1] O'Reilly - Daniel P. Bovet, Marco Cesati "Understanding the Linux Kernel"

[2] Tigran Aivazian - Linux Kernel 2.4 Internals - http://www.tldp.org/LDP/lki/lki.pdf

Indice