Facoltà di Scienze Matematiche, Fisiche e Naturali

Corsi di Laurea in Informatica e in Tecnologie informatiche

Sistemi Operativi I

Schede integrative

Proff. L. V. Mancini e F. Massaioli

Anni accademici 2006-2008

1 Creazione e terminazione di processi sotto UNIX

La creazione di processi in un sistema operativo può avere svariate cause, che includono: comandi interattivi dell'utente, comandi eseguiti a tempo fissato, lavori avviati da un sistema di code *batch*, processi avviati in risposta ad eventi verificatisi (per esempio una richiesta di connessione su un *port* TCP/IP). Tuttavia, dal punto di vista del sistema operativo la creazione di un nuovo processo viene sempre richiesta da un altro processo, un demone che fa parte di un sistema di code, o un demone che sovrintende alle connessioni di rete, o un interprete di comandi, grafico o testuale che sia. A questa regola fa ovviamente eccezione un particolare processo, sotto UNIX detto *init* (da non confondere con il comando dallo stesso nome). Tale processo è il primo processo vero e proprio ad apparire nel sistema, ha *pid* 1, è "assemblato" direttamente dal *kernel* durante la fase di *boot*, ed è in ultima istanza il progenitore di tutti i processi che verranno eseguiti nel sistema, secondo le relazioni di parentela che verranno descritte nel seguito.

La terminazione di un processo può avvenire per cause interne o esterne allo stesso. Sono cause interne la richiesta di terminazione (implicita o esplicita) del processo al suo completamento, o la richiesta del processo di terminare a seguito di una condizione erronea (per esempio un argomento errato nella linea di comando) che gli impedisce di svolgere i suoi compiti. Sono cause esterne la richiesta di terminazione da linea di comando da parte dell'utente possessore del processo o di un utente con privilegi di *superuser*, o la decisione del sistema operativo di terminare un processo quando questo genera un'eccezione, quando una richiesta di memoria non è soddisfabile, quando il processo eccede i limiti di tempo di esecuzione assegnati, quando viene identificata una condizione di stallo nella quale il processo è coinvolto.

Nel sistema operativo UNIX queste funzionalità sono implementate tramite alcune chiamate di sistema specifiche. I paragrafi successivi di questa scheda ne introducono quelle più significative rispetto agli obiettivi del corso.

1.1 Creazione: `fork()` ed `exec()`

Il meccanismo di creazione di processi in UNIX si fonda sulle chiamate di sistema **`fork()`** ed **`exec()`**.

La chiamata di sistema **`fork()`** crea a tutti gli effetti un nuovo processo, in tutto e per tutto identico al processo chiamante: l'immagine in memoria del nuovo processo (dati, *stack* e codice) è identica in dimensione e contenuti a quella del processo chiamante, i descrittori di *file* aperti vengono condivisi dal nuovo processo, etc. Il processo appena creato è il processo figlio, o *child process*, che riceve un nuovo *pid*. Il processo che ha chiamato la **`fork()`** ne è il processo genitore (*parent process*).

Il nuovo processo inizia la sua esecuzione, concorrente a quella del processo genitore, ritornando anche lui dalla **`fork()`**. Nel processo genitore, la **`fork()`** restituisce il *pid* del processo figlio appena creato. Nel processo figlio la **`fork()`** restituisce 0 (ovviamente, il processo figlio ha il proprio *pid* ed il *pid* del processo genitore registrati all'interno del proprio *Process Control Block*, e può quindi facilmente ottenerli invocando le chiamate di sistema **`getpid()`** e **`getppid()`**). Nel caso che la chiamata non abbia successo, la **`fork()`** restituisce un valore negativo e non viene creato alcun nuovo processo.

Un esempio tipico di uso della **fork()** è il seguente:

```
pid = fork();
if (pid==0) {
      /* siamo nel processo figlio */
} else if (pid > 0) {
      /* siamo nel processo genitore */
} else {
      /* insuccesso */
      int saveerror = errno;
      perror("Error in fork()");
      /* gestione dell'errore */
}
```

Ovviamente, la **fork()** è utile per creare più istanze concorrenti del medesimo programma (per esempio, per servire richieste indipendenti in un *web server*).

Per eseguire un programma differente si ricorre alla **exec()**, che sostituisce completamente l'immagine in memoria del processo chiamante con l'eseguibile specificato dagli argomenti, lasciandone inalterato *pid* e processo genitore. In realtà **exec()** non è una chiamata, ma una famiglia di chiamate (**execl()**, **execv()**, …), che corrispondono a varie modalità di specificare l'eseguibile e gli argomenti da passare su riga di comando.

Una chiamata ad **exec()** ritorna al programma chiamante solo in caso di fallimento. In caso di successo, non ritorna a nessun programma: il codice chiamante non esiste più, essendo stato sostituito dal nuovo eseguibile, il quale a sua volta deve essere avviato.

Un esempio tipico di uso della **exec()** è il seguente:

```
execl("/bin/ls", "ls", "-l", NULL);
/* insuccesso */
saveerror = errno;
perror("Error in execl()");
/* gestione dell'errore */
```

La chiamata di sistema **execl()** richiede come primo argomento il nome del *file* contenente l'eseguibile da utilizzare, come secondo argomento il parametro da passare al nuovo programma come **argv[0]** (per convenzione il nome del programma stesso), e opzionalmente a seguire gli argomenti successivi. Il **NULL** indica la fine degli argomenti da passare.

Per creare un nuovo processo corrispondente ad un eseguibile diverso da quello chiamante è necessario combinare l'uso di **fork()** ed **exec()**. Ad esempio, quando l'utente tramite una *shell* di comandi testuale dà un comando esterno alla stessa, la *shell* chiama la **fork()**, e resta in attesa che il processo figlio termini. Il processo figlio appena generato esegue una **exec()**, costruendone i parametri a partire dalla riga di comando fornita dall'utente.

1.2 Terminazione: exit() e kill()

Un processo richiede la propria terminazione con la chiamata **`exit()`**, che non è propriamente una chiamata di sistema, ma una *routine* della *Standard Library* del C. La chiamata di sistema vera e propria è **`_exit()`**. Nessuna delle due ritorna al programma chiamante, tutte e due provocano il rilascio di tutte le risorse, la chiusura di tutti i descrittori di *file* aperti e la terminazione del processo, tutte e due richiedono un argomento intero, i cui 8 bit meno significativi sono passati al processo genitore. La differenza tra le due è che **`exit()`** effettua anche una serie di operazioni di finalizzazione delle librerie utilizzate, per esempio l'effettiva scrittura su *file* dei dati ancora contenuti nei *buffer* di I/O, la terminazione corretta di connessioni di rete, etc. In generale, l'utilizzo di **`_exit()`** è sconsigliabile e riservato a situazioni particolarissime.

Per convenzione, un processo che termina con successo lo fa tramite **`exit(0)`**, uno stato di uscita diverso da zero indica in generale una terminazione dovuta ad una condizione erronea.

Frequentemente, **`exit()`** non viene invocata esplicitamente ma implicitamente. Per esempio, un programma in C termina quando la funziona **`main()`** esegue lo *statement* return. Il valore passato a return viene restituito da **`main()`** al codice di *start-up* del processo, inserito nell'eseguibile in fase di *linking* (se si utilizza per il *linking*, come avviene di solito, il compilatore, quest'ultimo specifica automaticamente al *linker* il codice di *start-up* corretto da includere nell'eseguibile). Il codice di *start-up* chiede la terminazione del processo, passando ad **`exit()`** il valore restituito da **`main()`**.

A seguito della chiamata ad **`exit()`** il processo viene terminato, la sua immagine in memoria distrutta, il suo *Process Control Block* liberato. Quest'ultima operazione non può avvenire prima che il processo genitore abbia ricevuto lo stato di uscita del figlio (registrato, appunto, nel PCB DEL FIGLIO). In alcune situazioni, è possibile che il genitore non sia pronto a richiedere lo stato di uscita del figlio, ed il PCB di quest'ultimo deve essere conservato nelle tabelle di sistema. Un processo in tale stato figura nel sistema in uno stato *zombie*.

Un processo può essere terminato dall'esterno inviandogli un opportuno *signal* tramite la chiamata di sistema **`kill()`**, tipicamente inviando i segnali SIGINT o SIGKILL. In realtà non è tanto **`kill()`** a terminare il processo, ma più propriamente quest'ultimo a richiedere di conseguenza la propria terminazione.

La ricezione di uno specifico *signal* è registrata in una serie di *flag* contenuti nel *Process Control Block*. Quando un processo passa nello stato *Running* o esce da una chiamata di sistema, verifica se sia arrivato un *signal* e chiama una *routine* di gestione appropriata (*signal handler*). La maggior parte dei *signal handler* di *default* chiedono la terminazione del processo, ma possono essere ridefiniti dal programmatore o da librerie utilizzate. Il *signal handler* di SIGKILL non può essere ridefinito, e per questo è l'estrema risorsa per ottenere la terminazione di un processo. Una conseguenza della gestione asincrona dei *signal* è che un processo in stato *Blocked* che riceva SIGINT o SIGKILL non può comunque terminare finché non esce da tale stato.

1.3 Considerazioni sul flusso di esecuzione

Le chiamate di sistema discusse in questa sessione si comportano in modo molto peculiare rispetto al flusso di esecuzione "abituale" di un programma. Infatti, quando normalmente in un programma si chiama una *subroutine* ci si aspetta che esegua il suo lavoro e poi ritorni al programma chiamante.

Al contrario, per **`exit()`**, **`_exit()`**, **`exec()`** e **`fork()`** bisogna sempre tenere presente che:

→ **`exit()`** ed **`_exit()`** non ritornano mai al programma chiamante

→ **`exec()`** ritorna al programma chiamante solo in caso di errore

→ **`fork()`** è chiamata in un processo ma ritorna in due processi, il genitore ed il figlio, tranne che in caso di fallimento, nel quale ritorna solo al processo chiamante.

ATTENZIONE: se non si tengono ben presenti i possibili flussi di esecuzione risultanti da queste chiamate e non si gestiscono correttamente tutti i diversi comportamenti che si possono verificare, è possibile inserire nel codice errori anche di natura banale, ma che possono manifestarsi in comportamenti "sorprendenti", se non addirittura mettere in crisi il sistema su cui il programma è in esecuzione.

2 I meccanismi di protezione sotto UNIX

I meccanismi di protezione di un sistema operativo hanno innanzitutto lo scopo di proteggere dati e processi di un utente da interventi indebiti di altri utenti (lettura o scrittura di *file*, invio di *signal*, etc…). Inoltre, per garantire flessibilità nell'uso del sistema, devono consentire, oltre all'esclusione degli accessi indebiti, un controllo dell'accesso a informazioni e processi da parte di più utenti, con una granularità sufficiente per gli usi tipici del sistema.

Il sistema operativo UNIX implementa i meccanismi di protezione su due fronti: autenticazione dell'utente (*User-Oriented Access Control*) e diritti o permessi di accesso ai dati (*Data-Oriented Access Control*).

Questa sezione illustra i meccanismi fondamentali disponibili su un sistema UNIX base, considerato come a se stante. La situazione reale è tuttavia più varia. Le informazioni che descrivono le utenze e i gruppi possono essere organizzate, invece che su ogni singola macchina, su *server* centralizzati, in modo da consentire autenticazione distribuita su rete (NIS, NIS+, LDAP). L'autenticazione può essere basata su sistemi più sicuri della *password* standard (Kerberos, *one-time password*, *challenge based*). Il meccanismo di autenticazione può essere implementato in maniera più complessa (per esempio astraendo più modalità tramite PAM) o più sicura (rimpiazzando i meccanismi standard tramite SSH). Il controllo dell'accesso ai *file* può essere realizzato con granularità molto più fine sui *file system* che supportano l'uso di *Access Control List*.

La discussione che segue è limitata a quella parte dei meccanismi fondamentali di UNIX sufficiente ad illustrare i concetti necessari nell'ambito delle finalità del corso.

2.1 Utenze e gruppi

Ogni utente di un sistema UNIX è univocamente identificato da un *username* alfanumerico e da un *user identifier* (*uid*) numerico. Lo *uid* è utilizzato in tutte le strutture dati del sistema (PCB, *i-node*, etc…) nelle quali sia necessario identificare l'appartenenza di una risorsa (processo, *file*, etc…) ad un utente specifico.

Ogni utente appartiene anche ad uno o più gruppi (il cui significato sarà chiarito nel seguito). Ogni gruppo è univocamente identificato da un nome alfanumerico e da un *group identifier* (*gid*) numerico. I gruppi sono definiti nel *file* **`/etc/group`**.

Ogni utenza è completamente definita da una riga del *file* **`/etc/passwd`**, contenente nell'ordine: *username*; *password* criptata; *uid*; il *gid*; nome vero ed altre informazioni in formato libero; *home directory* dell'utente; interprete di comandi (*shell*) utilizzato. Dato che **`/etc/passwd`** deve essere leggibile da tutti i processi, su alcuni sistemi la *password* è conservata in un altro *file*, ad accesso ristretto, **`/etc/shadow`**. Oltre che al gruppo specificato in **`/etc/passwd`**, detto gruppo principale, un utente può far parte di altri gruppi, detti secondari. Tale appartenenza è specificata in **`/etc/group`**.

Una tipica riga di **`/etc/passwd`** si presenta in questo modo (la x nel campo *password* indica l'uso di uno *shadow file*):

```
sabinar:x:6335:283:Sabina Rossi:/home/sabinar:/bin/csh
```

Una tipica riga di **`/etc/group`** si presenta in questo modo:

```
aan:x:283:
```

2.2 La procedura di login

Per accedere al sistema (ossia per ricevere il permesso di eseguire comandi), l'utente deve essere autenticato, ovvero la sua identità deve essere stabilita ad un ragionevole livello di certezza.

Su un sistema UNIX, a seconda del tipo di terminale utilizzato (seriale, via rete, etc…), un processo specifico (**`getty`**, **`telnetd`**, etc…) presenta una richiesta di *username*. Dopo l'inserimento della *username* il controllo viene passato al comando **`login`**, che richiede una *password*. Inserita quest'ultima, il processo **`login`** verifica la coppia *username* e *password* nei *file* di configurazione pertinenti. In caso di corrispondenza, sostituisce se stesso con l'interprete di comandi definito per l'utenza in questione, utilizzando come *directory* di esecuzione la *home directory* specificata nel *file* **`/etc/passwd`**. Il processo di *shell* così creato appartiene all'utente appena autenticato, ed al gruppo principale di utenti al quale l'utente appartiene.

Nel caso di connessione via rete, quando l'utente termina l'esecuzione della *shell* con il comando **`exit`**, o quando l'autenticazione fallisce, la connessione viene chiusa. Nel caso di un terminale seriale, viene lanciato un nuovo processo **`getty`** che ne riprende il controllo.

2.3 Diritti di accesso

In un sistema UNIX, ogni *file* (e quindi ogni dispositivo, essendo questi rappresentati nel sistema come *file*), ha associato un proprietario ed un insieme di diritti di accesso (o permessi).

La proprietà di un *file* è normalmente attribuita utilizzando *uid* e *gid* del processo che lo ha creato.

I diritti di accesso sono di tre tipi:

- lettura: **`r`**
- scrittura: **`w`** (include la cancellazione)
- esecuzione: **`x`**

Per ogni *file* sono presenti tre terne di questi diritti, verificabili tramite il comando **`ls -l`**:

```
-rwxr-xr-x     1    federico  em        5120 Nov  7 11:03 a.out
-rw-r--r--     1    federico  em         233 Nov  7 11:03 test.c
```

La prima terna corrisponde ai diritti dell'utente proprietario (*user*), la seconda ai diritti concessi agli altri utenti del gruppo proprietario del *file* (*group*), la terza a tutti gli altri utenti del sistema (*other*). In questo modo, è possibile controllare tre livelli di permessi di accesso. Alcuni utenti privilegiati (*root*, *operator*, …) hanno diritti di lettura e scrittura su qualsiasi *file* del sistema, indipendentemente dai diritti di accesso ad esso associati.

I diritti di accesso possono essere modificati con il comando **`chmod`**, dal proprietario del *file* o da un utente privilegiato (*superuser*). Quest'ultimo può anche modificare la proprietà di un *file* con il comando **`chown`**.

In maniera analoga (con differenze specifiche, da verificare caso per caso) sono trattati i diritti di accesso ad aree di memoria condivisa (*shared memory segment*) e i diritti di inviare *signal* a processi.

2.4 Cambiamento di identità

Un utente interattivo di un sistema UNIX può cambiare la propria identità, ossia la proprietà del processo dell'interprete di comandi che sta utilizzando, con il comando **su** *username*. In questo modo assume tutti i diritti associati alla nuova identità.
Se il cambiamento avviene tra utenti normali, o da un utente normale ad uno degli utenti privilegiati (*root*, *operator*, …) che hanno diritti completi su tutti i *file*, viene richiesta la *password* associata alla identità da assumere. In caso di passaggio da un'utenza privilegiata ad un utenza non privilegiata, non viene richiesta alcuna *password*, dato che si stanno limitando, e non ampliando o cambiando, i diritti di accesso di cui si è in possesso.

2.5 Diritti dei processi: utenza e gruppo reali ed efficaci

Nonostante l'uso colloquiale dell'espressione "accesso da parte di un utente", in un sistema operativo non è tanto l'utente ad accedere ai *file* (che si tratti di dati o di eseguibili), quanto i suoi processi. Di conseguenza, ogni volta che un processo tenta l'accesso ad un *file*, il successo o meno dell'operazione dipenderà dalla corrispondenza dei permessi associati al *file* stesso con *uid* e *gid* del processo.

Se lo *uid* del processo è la stesso del proprietario del *file*, verrà presa in esame la prima terna di diritti. In caso contrario, viene presa in considerazione il *gid* del processo: se questo corrisponde al *gid* associato al *file*, verrà presa in considerazione la seconda terna di permessi, altrimenti la terza.

In alcuni casi particolari può essere necessario attribuire ad un processo più diritti o diritti diversi da quelli dell'utente che lo ha lanciato. Un esempio è il comando **passwd**, che consente di modificare la *password* associata all'utenza corrente. Tale comando deve modificare il *file* **/etc/passwd**, che per motivi di accesso deve essere leggibile da tutti. Tuttavia la modifica di tale *file* da parte di un utente normale deve essere limitata al campo *password* della riga che si riferisce all'utente. La modifica di informazioni relative ad altre utenze deve essere possibile solo ed esclusivamente ad utenti privilegiati.

Per risolvere problemi di questo genere, UNIX associa ad ogni processo, oltre a *uid* e *gid* reali dell'utente che lo ha lanciato, una coppia di *uid* e *gid* efficaci che sono quelli effettivamente presi in esame nel momento dell'accesso ai *file*. Tali *uid* e *gid* efficaci sono normalmente uguali a quelli reali, a meno che il *file* contenente l'eseguibile non abbia abilitati i permessi speciali SETUID (abilitabile col comando **chmod u+s *nome_file***) e/o SETGID (abilitabile con **chmod g+s *nome_file***), nel qual caso *uid* e/o *gid* efficaci saranno quelli del proprietario del *file* eseguibile. Per esempio, il comando **passwd** ha i seguenti permessi:

```
-r-sr-sr-x  1  root     sys    21964  Apr  7  2002 /bin/passwd
```

Quando il *file* eseguibile passato come argomento alla chiamata di sistema **exec()** ha i permessi SETUID e/o SETGID abilitati, il processo risultante sarà del tipo SETUID e/o SETGID. Nell'esempio, il processo creato dall'esecuzione del comando **passwd** avrà fin dall'inizio i permessi efficaci dell'utente *root*, e quindi potrà modificare il *file* **/etc/passwd**. Il programma è tuttavia scritto in modo da modificare solo la riga corrispondente alla *uid* reale, ossia all'utente che ha lanciato il comando.

Un processo con permessi SETUID e/o SETGID può alternare i propri *uid* e/o *gid* efficaci tra quelli del proprietario del processo e quelli del proprietario del *file* eseguibile tramite le chiamate di sistema **setuid()**,**setgid()**, **seteuid()** e **setegid()**. Per ovvie ragioni di sicurezza, ad un processo il cui *uid* efficace corrisponde al *superuser* (*root*) non è consentito alternare tra i due proprietari, ma solo il degrado definitivo del processo ai diritti più bassi dell'utente che lo ha lanciato, senza poter più riacquisire i diritti di *root*..

È quasi superfluo osservare che se da un canto questo meccanismo consente in modo flessibile la temporanea estensione o modifica dei diritti di un utente, d'altro canto presta il fianco ad attacchi di sicurezza di vario tipo, tanto più pericolosi se un eseguibile SETUID è proprietà di un utente privilegiato. Pertanto si tratta di un meccanismo da usare con estrema cautela, quando non siano disponibili alternative ragionevoli, e con particolare attenzione agli aspetti di sicurezza nella scrittura dei codici relativi.

3 Confronto delle politiche di sostituzione

Un sistema operativo, per assolvere alle proprie funzioni, utilizza inevitabilmente risorse del sistema, in particolare tempo di CPU e memoria, sottraendole alle applicazioni. Ogni qual volta si deve scegliere quale algoritmo o politica utilizzare per implementare una determinata funzionalità, non è sufficiente considerare solo la "qualità" dei risultati prodotti da ognuno dei possibili algoritmi, è necessario valutare anche il costo di ciascuno in termini di risorse occupate.
Il caso delle politiche di sostituzione utilizzate quando il numero di *frame* di memoria fisica non è sufficiente ad ospitare le pagine di memoria dei processi è un esempio particolarmente istruttivo. Per semplicità, nel seguito si fa riferimento ad una politica di sostituzione locale, nella quale tutte le operazioni di sostituzione riguardano esclusivamente i *frame* assegnati al processo che ha richiesto la nuova pagina. Le conclusioni sono comunque valide in generale.

Ad una prima analisi può sembrare che, in questo ambito, l'unico elemento da considerare nella valutazione sia l'occupazione di memoria necessaria per implementare la politica prescelta. Infatti, alle velocità dei processori di oggi, il tempo di CPU utilizzato al momento della sostituzione sarà sicuramente trascurabile a fronte dell'accesso a disco necessario per caricare la pagina richiesta (e dell'eventuale accesso per scrivere il contenuto della pagina sostituita).
Tuttavia, alcune politiche richiedono ad ogni accesso in memoria l'aggiornamento delle strutture dati relative alla pagina acceduta. L'impatto di questi aggiornamenti potrebbe essere tale da rallentare da 2 a 5 volte la velocità di elaborazione del sistema. Questo aspetto dovrà essere tenuto in debito conto.

Il caso della politica FIFO è senz'altro il più semplice. L'ordine di sostituzione è predeterminato, una volta per tutte. Assumendo che sia lo stesso nel quale le pagine sono elencate nella *page table*, per implementare la politica è sufficiente un puntatore che ne percorra ciclicamente le righe. Non è necessaria alcuna operazione aggiuntiva nel corso dei normali accessi in memoria alle pagine già presenti in memoria. Il consumo di risorse è quindi sostanzialmente trascurabile. Purtroppo, anche l'efficienza della politica è scarsa.

La politica LRU ha un'efficienza decisamente più elevata, e più elevato è ovviamente il suo costo di realizzazione. Il modo meno oneroso di implementarla è di realizzare una *doubly linked list* delle pagine del processo che sono presenti in *frame* di memoria, mantenendola ordinata in modo che la testa sia la pagina acceduta da minor tempo, e la coda sia sempre la pagina da sostituire. Ad ogni riga della *page table* dovranno pertanto essere associati due puntatori, ognuno di almeno $\log_2 N$ bit, dove N è il numero di pagine nella tabella. Altri due puntatori saranno necessari per identificare la testa e la coda della lista.
Per mantenere la lista ordinata è sufficiente che ad ogni accesso in memoria, la pagina corrispondente sia portata in testa. È facile convincersi che nessuna operazione è richiesta per un accesso in memoria nella pagina usata più di recente, ma che per un accesso ad una pagina che si trova all'interno della lista è necessario modificare 5 puntatori (4 nel caso della pagina di coda). Per fare un accesso in memoria ne sono necessari molti di più, con un impatto rilevante sulle prestazioni del sistema.

La politica del *clock* è un compromesso tra i due estremi precedenti. È innanzitutto importante osservare che in tale politica le pagine vengono esaminate sempre nello stesso ordine, esattamente come nella politica FIFO. In realtà, una politica FIFO non è nient'altro che una politica del *clock* con nessun bit di uso per pagina, nella quale quindi ogni pagina è sempre sostituibile. La politica del *clock* sarà tanto più efficiente quanto maggiore è il numero di bit di uso: più bit di uso implicano più informazioni disponibili per ogni pagina, e migliore capacità di discriminare tra pagine differenti. È ragionevole pensare che per raggiungere prestazioni identiche a quelle della politica LRU sia necessario usare un numero di bit di uso paragonabile con il numero di bit necessario per implemen-

tare la LRU. L'evidenza sperimentale mostra però che già pochi bit di uso (≤8) assicurano un'efficienza di poco inferiore alla LRU.
Perché la politica funzioni correttamente, è necessario che ad ogni accesso in memoria il contatore formato dai bit di uso sia incrementato (fino ad eventuale saturazione). Quindi, per ogni accesso in memoria, è necessario effettuare una sola lettura ed una sola scrittura aggiuntive, situazione assai favorevole rispetto a quella della politica LRU. Inoltre, dato che l'intervento va effettuato solo su dati corrispondenti alla pagina acceduta, senza aggiornamenti relativi ad altre pagine come avviene per la LRU, e dato che i bit di uso necessari sono pochi, tali bit possono essere conservati per ogni pagina nella riga corrispondente della *page table*. Ciò consente di implementare l'aggiornamento dei bit di uso con impatto di tempo nullo sull'elaborazione del processo. Infatti, affinché una pagina sia accessibile dal processore, la sua *entry* nella *page table* deve obbligatoriamente essere presente nel TLB. Di conseguenza, l'incremento dei bit di uso potrà essere demandato all'*hardware* che lo esegue contemporaneamente al processo di traduzione dell'indirizzo.

4 Il *file system* FAT

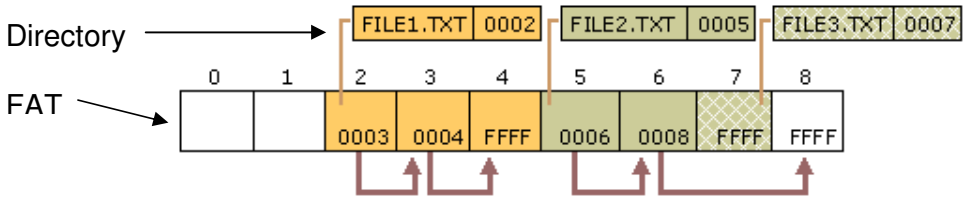
Il *file system* FAT è stato introdotto con MS-DOS. Originariamente concepito per dischi di piccole dimensioni (i *floppy* il cui uso era dominante all'epoca), è stato successivamente esteso per supportare i dischi rigidi di dimensione crescente. Le limitazioni sempre più evidenti al crescere delle dimensioni del *file system* ne stanno portando alla sostituzione con sistemi dalle migliori prestazioni. È tuttavia ancora in uso, oltre che per i dischi *floppy*, per dispositivi di memoria quali le "chiavi" USB basate su tecnologia FLASH.

Nel *file system* FAT, lo stato di uso e l'allocazione dei blocchi sono descritti in un'unica struttura dati, detta appunto *File Allocation Table*, memorizzata nella parte iniziale della partizione del disco assegnata al *file system*. Si tratta di una tabella lineare, ad una sola colonna, con tante righe quanti sono i blocchi (ovvero le unità minime di allocazione) del *file system* (detti *cluster* nella terminologia Microsoft). Ogni elemento, a seconda delle dimensioni del supporto fisico, è composto di 12, 16 o 32 bit (FAT-12, FAT-16, FAT-32).

Lo stato del blocco *i*-esimo è definito dal valore contenuto nell'*i*-esima riga della tabella:

0	blocco libero
≠0	blocco in uso

Normalmente il valore assegnato ad un blocco in uso è l'indice del blocco successivo all'interno del *file* di cui fa parte. Un valore speciale (tutti i bit a 1) marca l'ultimo blocco di un file. Un altro insieme di valori speciali è utilizzato per marcare come in uso blocchi fisicamente danneggiati, e prevenirne l'uso da parte del sistema operativo.



La rappresentazione di un *file* è completata dalla riga corrispondente in una delle *directory* del *file system*, che oltre al nome del file, alle date di modifica e accesso ed ai *flag* che ne definiscono informazioni accessorie, dovrà riportare l'indice del primo blocco che lo compone. La *directory* radice del *file system*, che segue immediatamente la FAT all'inizio della partizione assegnata, è di dimensioni fissate in FAT-12 e FAT-16, mentre può crescere dinamicamente, come un qualunque *file*, nel caso di FAT-32.

Sono evidenti i limiti di questo approccio. Da un lato, il numero massimo di blocchi gestibili è fissato dal numero di bit assegnato ad ogni elemento della FAT. All'aumentare delle dimensioni del *file system* sarà ad un certo punto necessario aumentare le dimensioni dei singoli blocchi per mantenerne limitato il numero, con un aumento conseguente della frammentazione interna. D'altro canto, il *caching* della FAT non è agevole. Un disco da 64 GB, gestito con FAT-32 e blocchi da 4 KB richiederà una FAT di dimensioni pari a 4 *byte* (32 bit) × 16 M (numero di blocchi), ossia 64 MB, una porzione tutt'altro che trascurabile della memoria RAM del sistema. Ovviamente è concepibile il *caching* parziale di porzioni (contigue!) di FAT, ma l'efficacia di tale approccio sarà scarsa per *file* estremamente frammentati, ossia i cui blocchi siano sparpagliati per il disco.

Di conseguenza, le prestazioni di un *file system* FAT peggioreranno non solo al crescere delle dimensioni del *file system* stesso, ma, in maniera imprevedibile, in base allo stato di frammentazione dei *file* sul supporto fisico. Un approccio con tale caratteristica si dice "non scalabile".

5 Ripristino della consistenza di un *file system*

I dati che descrivono un *file* (metadati) devono essere memorizzati su disco in modo da consentire l'accesso al contenuto del *file* a seguito di un *reboot*, e devono in linea di principio essere aggiornati ogni volta che un *file* viene modificato. Indipendentemente dal *file system* usato (FAT, NTFS, *i-node*, …), l'aggiornamento sincrono dei metadati ad ogni modifica del *file* comporterebbe un rallentamento intollerabile delle operazioni: essendo i metadati rappresentati in pochi *byte*, ed avvenendo l'I/O su disco per blocchi da qualche KB, un aggiornamento costante comporterebbe un grande traffico di dati la maggior parte dei quali restano immutati.

Per ovviare a questo problema, il SO aggiorna tempestivamente solo le copie in memoria RAM dei metadati dei *file* attualmente in uso, differendo a momenti di traffico limitato con il disco l'aggiornamento dei metadati sul supporto fisico. Dato che questo aggiornamento del supporto fisico è determinato da considerazioni di efficienza dell'I/O, e non dalla successione temporale delle operazioni sui *file*, è possibile che ad un dato istante di tempo i metadati sul supporto fisico non solo non corrispondano al reale contenuto dei *file*, ma che siano anche contraddittori tra loro, ovvero inconsistenti. Le possibili inconsistenze possono essere di vario genere, ad esempio: un blocco risulta in uso ma non far parte di nessun *file*; un blocco risulta libero ma appartenere ad un *file*; un blocco risulta appartenente a due *file* contemporaneamente; un *file* appare descritto correttamente, ma nessuna *directory* lo elenca; e così via. Queste situazioni aprono un problema molto serio: se un fallimento *hardware* del sistema, o un'improvvisa mancanza di alimentazione elettrica, o un'operazione indebita di un utente (rimozione forzosa di un supporto fisico) impediscono l'aggiornamento completo dei metadati su disco, l'accesso al *file system* ne può risultare compromesso.

Il verificarsi (anche potenziale) del problema può essere rilevato con un semplice meccanismo. All'atto dell'apertura (*mount*) di un *file system*, in una particolare locazione del disco, utilizzata come *flag*, viene registrato un valore non nullo. Al momento della chiusura (*unmount*) del *file system*, nella locazione in questione viene scritto il valore 0. In questo modo, se ad una successiva apertura, il valore in questione risulta non nullo, evidentemente il *file system* non è stato chiuso correttamente, ed i metadati sono potenzialmente inconsistenti.

L'inconsistenza dei metadati viene corretta con appositi programmi (**`fsck`** sotto UNIX) che effettuano il raffronto incrociato di tutti i metadati, correggendo le contraddizioni nei modi più ragionevoli. Per esempio: assegnando un nome di *file* arbitrario ad un *file* che non è definito ma non elencato in nessuna *directory*, lasciando all'utente o all'amministratore decisioni ulteriori in merito; marcando come libero un blocco che risulti in uso ma non appartenente a nessun *file*; marcando come in uso un blocco libero che risulti appartenente ad un *file*; e così via. È chiaro che non c'è alcuna garanzia di salvaguardia dei dati (l'unico approccio ragionevole ad alcune inconsistenze è la cancellazione del *file*), ma perlomeno si ripristina uno stato coerente del *file system* e si rende possibile l'accesso sicuro ai *file* non danneggiati.

L'operazione di ripristino della consistenza è tanto più complessa ed onerosa quanto maggiori sono le dimensioni del *file system* ed il numero di *file* in esso contenuti. Per velocizzare il procedimento in applicazioni nelle quali è cruciale il ripristino tempestivo di un servizio dopo un *crash* del sistema, si utilizzano approcci come il *journaling*. Per un esempio di questa tecnica si rimanda al paragrafo 12.9 dello Stallings, che descrive NTFS.

6 Equivalenza dei meccanismi per la programmazione concorrente

I meccanismi per la programmazione concorrente visti durante il corso (condivisione di risorse + mutua esclusione, condivisione di risorse + sincronizzazione, e comunicazione o *message passing*) sono di fatto equivalenti, nel senso che se un'applicazione può essere implementata in termini di uno di essi è possibile implementarla utilizzandone un altro qualsiasi. Ovviamente, a seconda della particolare applicazione e del contesto in cui verrà sviluppata ed utilizzata, un particolare meccanismo potrà rivelarsi più conveniente degli altri, in termini di facilità di sviluppo, di prestazioni, e di gestione.

Scopo di questa sezione è di illustrare in forma succinta e schematica tale equivalenza, suggerendo operativamente come realizzarla, piuttosto che dimostrandola formalmente. Sarà sufficiente mostrare come sia possibile implementare un meccanismo in termini degli altri.

La mutua esclusione è equivalente alla sincronizzazione

L'implementazione della mutua esclusione tramite semafori è ovvia, utilizzando un semaforo inizializzato al valore 1, come discusso nel libro di testo.

L'implementazione della sincronizzazione tramite mutua esclusione è più complessa. Ai fini di questo corso, è sufficiente considerare come sia possibile risolvere il problema produttore-consumatore utilizzando la mutua esclusione, e ricorrendo ad una variabile che conta gli elementi nel *buffer* ed all'attesa attiva (*busy waiting*) per evitare che il consumatore prelevi un elemento che non è ancora stato prodotto, ossia per sincronizzare il consumo in modo che segua la produzione. La generalizzazione a più processi consumatori o un'implementazione più efficiente che eviti l'attesa attiva ovviamente richiedono codici più complessi, che esulano dagli obiettivi del corso.

La sincronizzazione + condivisione di risorse è equivalente alla comunicazione

Per convincersi che è possibile implementare la comunicazione tramite i meccanismi di sincronizzazione è sufficiente prendere in considerazione il problema produttore-consumatore, che altro non è che un modo di implementare comunicazioni monodirezionali tra due processi in esecuzione sullo stesso sistema.

Per mostrare l'equivalenza opposta è innanzitutto necessario mostrare come emulare la condivisione di risorse (per esempio di locazioni di memoria) con la comunicazione. Il risultato può essere facilmente ottenuto realizzando un processo *server* al quale si affidano le locazioni di memoria da condividere, e realizzando un semplice protocollo con cui altri processi *client* possono, tramite scambio di messaggi, ottenere dal *server* i valori memorizzati nelle differenti locazioni o richiederne la modifica (ossia la memorizzazione di un nuovo valore fornito dal *client*). In secondo luogo, è necessario mostrare come implementare un semaforo, utilizzando un sistema di comunicazione ad indirizzamento indiretto:

```
mailbox mb;

commsem_signal(sem) {
      nonblocking_send(sem, "");
}

commsem_wait(sem) {
      blocking_receive(sem, dummyvar);
}

commsem_init(sem, n) {
      for(j = 0; j<n; ++j)
            commsem_signal(sem);
}
```

```
/* esempio di uso*/
mailbox mb;

int main() {
      …
      /* per inizializzare il semaforo */
      commsem_init(mb, 5);
      …
      /* per una wait */
      commsem_wait(mb);
      …
      /* per una signal */
      commsem_signal(mb);
      …
}
```

Dalle due equivalenze discusse si deduce ovviamente quella tra comunicazione e condivisione di risorse + mutua esclusione.

7 Il problema dei filosofi a cena

Le soluzioni del problema dei filosofi a cena discusse sullo Stallings di fatto cambiano i termini del problema, introducendo più forchette, o cambiando le regole di comportamento dei filosofi (imparano a mangiare la pasta con una forchetta sola), o introducendo un supervisore (il processo cameriere). È interessante vedere una soluzione che, rispettando i termini originali del problema, evita lo stallo applicando una strategia di prevenzione del problema. Da un punto di vista logico, la numerazione delle forchette in modo che ogni filosofo acceda alle medesime in ordine crescente di numerazione, equivale semplicemente a rompere la "circolarità" destra-sinistra di accesso alle forchette. Sarà sufficiente che uno dei filosofi acceda in ordine diverso dagli altri. A tal fine basta fornire una funzione che riporti il "numero" d'ordine del filosofo, compreso tra 0 ed N:

```
semaphore fork[N] = {1, 1, … 1};

philosopher() {
      int me, left, right, first, second;

      me = mynumber();
      left = me;
      right = (me + 1) % N;

      first = right < left ? right : left;
      second = right < left ? left : right;

      while (1) {
            think_for_a_while();

            wait(fork[first]);
            wait(fork[second]);

            eat();

            signal(fork[first]);
            signal(fork[second]);
      }
}
```

Il problema dei filosofi a cena può essere codificato utilizzando comunicazioni con indirizzamento indiretto invece dei semafori. I prefissi **b** ed **nb** indicano rispettivamente comunicazioni bloccanti e non bloccanti. Questa è la soluzione che può provocare uno stallo:

```
mailbox fork[N];

init_forks() {
      int i;

      for(i=0; i<N; ++i)
            bsend(fork[i], "fork");
}
```

```
philosopher() {
      int me, left, right;
      message fork1, fork2;

      me = mynumber();
      left = me;
      right = (me + 1) % N;

      while (1) {
            think_for_a_while();

            breceive(fork[left], fork1);
            breceive(fork[right], fork2);

            eat();

            nbsend(fork[left], fork1);
            nbsend(fork[right], fork2);
      }
}
```

La soluzione corretta, che previene lo stallo, del tutto equivalente a quella vista sopra per i semafori, è la seguente:

```
mailbox fork[N];

init_forks() {
      int i;

      for(i=0; i<N; ++i)
            bsend(fork[i], "fork");
}

philosopher() {
      int me, left, right, first, second;
      message fork1, fork2;

      me = mynumber();
      left = me;
      right = (me + 1) % N;


      first = right < left ? right : left;
      second = right < left ? left : right;

      while (1) {
            think_for_a_while();

            breceive(fork[first], fork1);
            breceive(fork[second], fork2);

            eat();

            nbsend(fork[first], fork1);
            nbsend(fork[second], fork2);
      }
}
```

L'uso delle comunicazioni consente una variante interessante. Supponendo che ogni filosofo pensi per un intervallo di tempo di durata casuale, indipendente e scorrelata da quella delle "riflessioni" degli altri commensali, si può scrivere la soluzione seguente:

```
mailbox fork[N];

init_forks() {
      int i;

      for(i=0; i<N; ++i)
            bsend(fork[i], "fork");
}

philosopher() {
      int me, left, right;
      message fork1, fork2;

      me = mynumber();
      left = me;
      right = (me + 1) % N;

      while (1) {
            think_for_a_random_time();

            if (nbreceive(fork[left], fork1)) {
                  if (nbreceive(fork[right], fork2)) {
                        eat();
                        nbsend(fork[left], fork2);
                  }
                  nbsend(fork[right], fork1);
            }
      }
}
```

Questa soluzione può dare luogo a *livelock* e *starvation*, ma non a stallo. In applicazioni reali si può ricorrere a una tecnica analoga (usando in alternativa protocolli di *locking*, o semafori, che non bloccano il processo se il *lock* non è disponibile o il semaforo è bloccato), quando l'azione corrispondente ad **`eat()`** è differibile e nel frattempo è possibile svolgere altro lavoro utile.

8 Una soluzione robusta per il barbiere “equo”

Il barbiere “equo”, così come definito dallo Stallings, è la versione nella quale si tiene conto della corrispondenza cliente-barbiere, in modo da evitare che un barbiere che ha finito il suo lavoro faccia alzare un cliente seduto su un’altra sedia.

La soluzione proposta dallo Stallings prevede la conoscenza del numero massimo di clienti serviti in un giorno (50, nel caso specifico). Tali assunzioni sono sempre insoddisfacenti, e da evitare. Se un giorno il programma fosse trasferito su un sistema più veloce, il valore massimo scelto potrebbe risultare insufficiente, e il programma manifesterebbe disfunzioni di cui bisognerebbe rintracciare la causa.

Una soluzione robusta si fonda non su stime ipotetiche e soggette a variazione in futuro, ma su dati sicuri. Il dato sicuro, in questo problema, non è il numero di clienti serviti in un giorno, ma il numero di barbieri. Lo pseudocodice che segue effettua anche un’ulteriore semplificazione rispetto allo Stallings, eliminando la strana pantomima del cassiere virtuale che blocca uno dei barbieri: è il barbiere stesso a ricevere il pagamento, aderendo letteralmente alle specifiche del problema.

```
semaphore max_capacity = 20;
semaphore sofa = 4;
semaphore barber_chair = 0;
semaphore cust_ready = 0;
semaphore payment = 0;
semaphore receipt = 0;
semaphore finished [3] = {0};
semaphore mb = 1;
int next_barber;

void Customer() {
      int my_barber;
      wait(max_capacity);
      enter_shop();
      wait(sofa);
      sit_on_sofa();
      wait(barber_chair);
      get_up_from_sofa();
      signal(sofa);
      my_barber = next_barber;
      sit_in_barber_chair();
      signal(cust_ready);
      wait(finished[my_barber]);
      leave_barber_chair();
      pay();
      signal(payment);
      wait(receipt);
      exit_shop();
      signal(max_capacity);
}
```

```
void Barber() {
      int barber_number= get_own_ID();
      while (true) {
            wait(mb);
            next_barber = barber_number;
            signal(barber_chair);
            wait(cust_ready);
            signal(mb);
            cut_hair();
            signal(finished[barber_number]);
            wait(payment);
            accept_pay();
            signal(receipt);
      }
}
```

9 Bibliografia

W. Stallings, *Operating Systems, Internals and Design Principles*, 5th ed, 2005, Prentice Hall

K. Haviland, B. Salama, *UNIX System Programming*, 1987, Addison Wesley

http://www.microsoft.com/technet/prodtechnol/Windows2000Pro/reskit/part3/proch17.mspx