UNIVERSITA' "CA' FOSCARI" DI VENEZIA

DIPARTIMENTO DI MATEMATICA APPLICATA

MARCO CORAZZA - SERENA FACCO

IL GESTORE *CACHE* IN WINDOWS NT

n. 15/2003

Il gestore *cache* in Windows NT

Marco Corazza
Dipartimento di Matematica Applicata - Università Ca' Foscari di Venezia

Serena Facco

Sintesi - In questo scritto, dopo aver sinteticamente introdotto il sistema operativo Windows NT e la sua struttura architetturale, si propone una elementare presentazione del gestore della memoria *cache* in questo stesso sistema operativo. In particolare, si presentano alcuni approfondimenti relativamente alla sua gestione del *Virtual Address Control Block* e delle operazioni di lettura e di scrittura.

Parole chiave e frasi - Windows NT, memoria, gestore *cache*.

1 La memoria *cache* nella gerarchia delle memorie

In un sistema di calcolo si possono usare vari dispositivi per la memorizzazione dei dati. Sostanzialmente, questi dispositivi si dividono in due categorie:

dispositivi primari: sono gli unici dispositivi di memoria direttamente accessibili dal processore. In questa categoria rientrano la RAM, la più piccola ma più veloce *cache* ed i registri interni allo stesso processore. Tali dispositivi hanno una capacità di memorizzazione limitata, costi elevati, ma forniscono un veloce accesso ai dati.

dispositivi secondari: in questa categoria rientrano i dischi magnetici, i dischi ottici ed i nastri magnetici. Tali dispositivi, generalmente, hanno una elevata capacità di memorizzazione, bassi costi, ma un tempo di accesso più elevato rispetto ai dispositivi primari. Ciò è dovuto al fatto che i dispositivi primari sono gli unici dispositivi accessibili direttamente dalla CPU; quindi, affinché la CPU possa accedere ai dati presenti nei dispositivi secondari, è necessario che essi siano prima trasferiti nei dispositivi primari attraverso delle richieste di *input/output* (nel seguito: I/O).

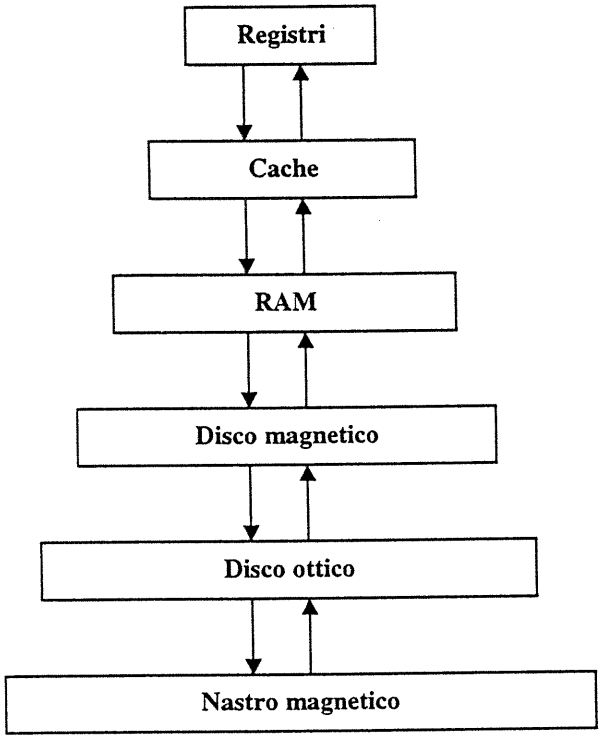
Ecco allora che si può introdurre una struttura gerarchica tra i dispositivi per la memorizzazione sulla base della loro velocità e del loro costo (si veda la figura che segue).

Un'altra caratteristica che sottolinea la differenza fra le due categorie di dispositivi per la memorizzazione dei dati è la volatilità delle informazioni. In particolare, i dispositivi primari sono spesso chiamati dispositivi di memoria volatile, in quanto perdono il proprio contenuto in presenza di caduta del sistema. Ecco allora che la persistenza delle informazioni viene salvaguardata tramite i dispositivi secondari. Ne segue che, per avere un buon sistema di memorizzazione, si deve trovare una buona combinazione tra i dispositivi di entrambe le categorie.

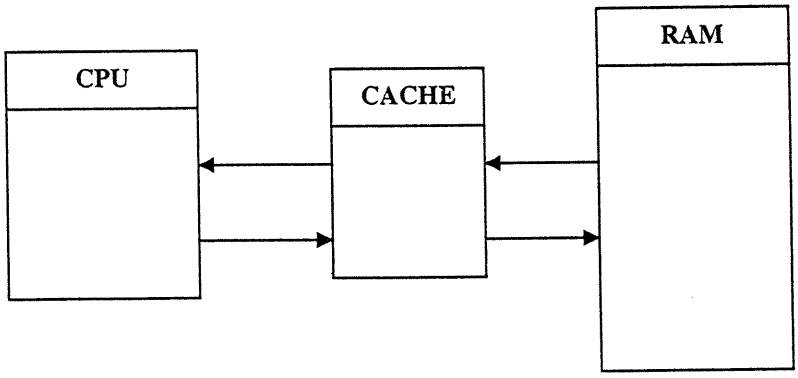
Focalizziamo ora la nostra attenzione sulla memoria *cache* al fine di comprendere, a livello generale, qual è la sua funzione.

Come noto, la maggior parte delle CPU oggi in commercio è in grado di decodificare una o più istruzioni in un solo ciclo di *clock*, e in tale ciclo eseguire una o più operazioni sul contenuto dei registri programmabili presenti all'interno della CPU (come, ad esempio, i registri indice). Tuttavia, per completare invece un accesso in memoria centrale (o, peggio, in memoria secondaria) possono invece essere richiesti molti cicli, poiché la CPU accede alla memoria centrale tramite una

transazione sul *bus* di memoria. Ecco che, in questi casi, il processore deve sospendere la fase di esecuzione poiché non dispone dei dati necessari per portare a termine l'istruzione in corso. Quindi, ecco che si crea un divario fra la velocità della CPU e quella della RAM, divario che in alcuni casi può risultare notevole.



Il rimedio a questo problema consiste nel "frapporre" tra la CPU e la RAM la memoria *cache*, che sappiamo essere più piccola, ma più veloce rispetto alla RAM.



Il concetto che sta alla base della memoria *cache* è il seguente: se si accede ad un'informazione in memoria centrale, allora esiste una non trascurabile probabilità che questa informazione possa nuovamente servire. In particolare, quando si deve accedere ad una informazione, per prima cosa il sistema operativo (nel seguito: S.O.) verifica la presenza dell'informazione nella memoria *cache*. Se l'informazione è presente, allora può essere messa direttamente a disposizione della CPU senza dover accedere alla memoria centrale, altrimenti l'informazione viene letta dalla memoria centrale e caricata nella memoria *cache*, per poi essere messa a disposizione della CPU che la ha richiesta.

Come già accennato, il requisito di avere memorie grandi e veloci contrasta con quello di avere dei costi ragionevolmente contenuti; quindi, preso atto che a causa dei suoi costi la memoria *cache* non può avere le dimensioni tipiche della memoria principale, rimane il problema pratico di poter disporre di spazi logici più grandi di quelli fisici. Per risolvere questo problema si rende

necessario effettuare degli scambi fra la memoria principale e la memoria cache con conseguenti problemi di sincronizzazione per mantenere la consistenza e l'integrità dell'informazione stessa.

Ecco che, ragionando in questi termini, potremmo interpretare i registri programmabili della CPU come una *cache* veloce per la memoria centrale, e potremmo considerare la memoria centrale come una *cache* veloce per la memoria secondaria; infatti, alcuni sistemi riservano una sezione separata della memoria centrale per la funzione di "*cache* del disco", dove memorizzare i blocchi in previsione di un loro utilizzo entro breve tempo. Più precisamente, quando un processo richiede la lettura di un blocco dal disco, per prima cosa il S.O. verifica la eventuale presenza di questo blocco nella *cache*; se il blocco è presente, può essere messo a disposizione del processo senza accedere al disco, altrimenti viene letto il blocco direttamente dal disco e caricato nella *cache*, per poi essere messo a disposizione del processo che lo ha richiesto. Lo stesso meccanismo è utilizzato anche per le operazioni di scrittura, nel senso che il blocco che deve essere scritto viene "intercettato" e, invece di essere memorizzato su disco, viene posto nella *cache*, per essere scritto su disco solo quando il sistema è costretto a rimuoverlo o, eventualmente, al "CLOSE" del file.

Ovviamente, quando si presenta la necessità di caricare un blocco in una *cache* piena, uno dei blocchi già presenti in essa deve essere rimosso ed eventualmente riscritto su disco (analogamente a quanto avviene nel meccanismo di *swapping* delle pagine nella gestione della memoria virtuale); se la copia del blocco residente nella *cache* ha subito modifiche, l'eliminazione deve essere preceduta da una scrittura sul disco per aggiornare la versione preesistente.

La maggior parte dei sistemi è dotata di una o più memorie *cache* e, fra queste, ce ne sono alcune interamente gestite dall'*hardware*. È da sottolineare il fatto che una buona gestione della memoria *cache* può comportare un notevole miglioramento a livello di prestazioni del sistema

2 Introduzione a Windows NT

Windows NT della Microsoft è un S.O. che riprende concetti e principi già noti da anni, i quali effettivamente sono stati implementati in molti altri S.O. attualmente in commercio. In altri termini, si può vedere Windows NT come il risultato di una confluenza di idee, principi e metodi derivanti da prodotti commerciali e progetti di ricerca condotti da vari Enti. Ad esempio, risulta chiaro che Windows NT sfrutta i principi di progettazione e le metodologie dei S.O. UNIX, OpenVMS e Mach; inoltre, si può rilevare anche l'influenza di sistemi operativi meno sofisticati come MS_DOS e OS/2.

Comunque, non si deve pensare che Windows NT sia poco affidabile in quanto unione di principi e idee di progettazione già rimaneggiati; anzi, ciò ha condotto allo sviluppo di una piattaforma di elaborazione più stabile e completa rispetto ai precedenti S.O proposti dalla Microsoft. In particolare, l'influenza del S.O. Mach si riscontra nello sforzo di minimizzare la dimensione del *kernel*, cioè di adottare una architettura *microkernel*; in questo modo è possibile apportare migliorie ad una parte del S.O. senza che ciò abbia ripercussioni sul resto del sistema.

Ci sono due versioni di NT, Windows NT Workstation e Windows NT Server, le quali usano lo stesso *kernel* e lo stesso codice di sistema; NT Server, però, è configurato per applicazioni *client-server*.

Windows NT non è un S.O. multiutente ed è in grado di controllare al più otto processori. Alcuni degli obiettivi chiave di questo sistema sono l'estendibilità, la portabilità, l'affidabilità (nel senso della sicurezza), e la compatibilità con le applicazioni MS-DOS. Per quanto riguarda l'estendibilità, vedremo che Windows NT ha una struttura stratificata, e che ciò facilita eventuali cambiamenti del sistema stesso nel corso del tempo e quindi gli permette di stare al passo coi tempi a livello tecnologico.

Windows NT è progettato per essere portabile, cioè il suo trasferimento da un'architettura *hardware* ad un'altra comporta un numero di cambiamenti relativamente ridotto; infatti, così come

per UNIX, la maggior parte del sistema è scritto in C o C++, e tutto il codice dipendente dal processore è isolato in una libreria dinamica, detta strato di astrazione dell'*hardware* (*Hardware Abstraction Layer*, HAL). Lo HAL agisce direttamente sull'*hardware*, isolando il resto di NT dalle differenze tra le piattaforme sulle quali il sistema è eseguito. Infatti, il sistema Windows NT può "girare" su Intel X86, DEC Alpha, e anche MIPS-based.

Windows NT è anche progettato per essere affidabile; in tal senso Windows NT è in grado di proteggere discretamente sè stesso ed i suoi utenti da software difettosi e da attacchi volontari al sistema, usando la protezione *hardware* per la memoria virtuale e meccanismi di protezione *software*.

3 L'architettura di Windows NT

Come già accennato, Windows NT ha una struttura stratificata per facilitare eventuali aggiornamenti nel corso del tempo.

Gli strati principali sono quello dell'astrazione dell'*hardware*, il *kernel* e l'esecutivo, operanti in modo protetto (*kernel mode*), ed un'ampia raccolta di sottosistemi (*subsystem*) eseguiti in modo utente.

Nella figura che segue è sinteticamente illustrata tale architettura.

3.1. Lo strato di astrazione dell'*hardware* (HAL)

Gli sviluppatori di Windows NT hanno creato un ridotto strato di software detto strato di astrazione dell'*hardware* (*Hardware Abstraction Layer*, HAL), che agisce direttamente sull'*hardware*, isolando i livelli superiori di NT (*kernel* ed esecutivo) dalle differenze tra le piattaforme sulle quali il sistema è eseguito. Ciò ha contribuito ad ottenere un sistema operativo portabile; infatti, ad esempio, è sufficiente una sola versione di ogni *driver*, la quale si può eseguire su tutte le piattaforme senza doverne modificare il codice.

3.2. Il kernel

Il *kernel* di Windows NT, sebbene di dimensioni ridotte rispetto all'esecutivo, è il cuore del S.O e rappresenta le basi sulle quali poggiano l'esecutivo ed i sottosistemi.

I compiti principali a cui assolve sono:

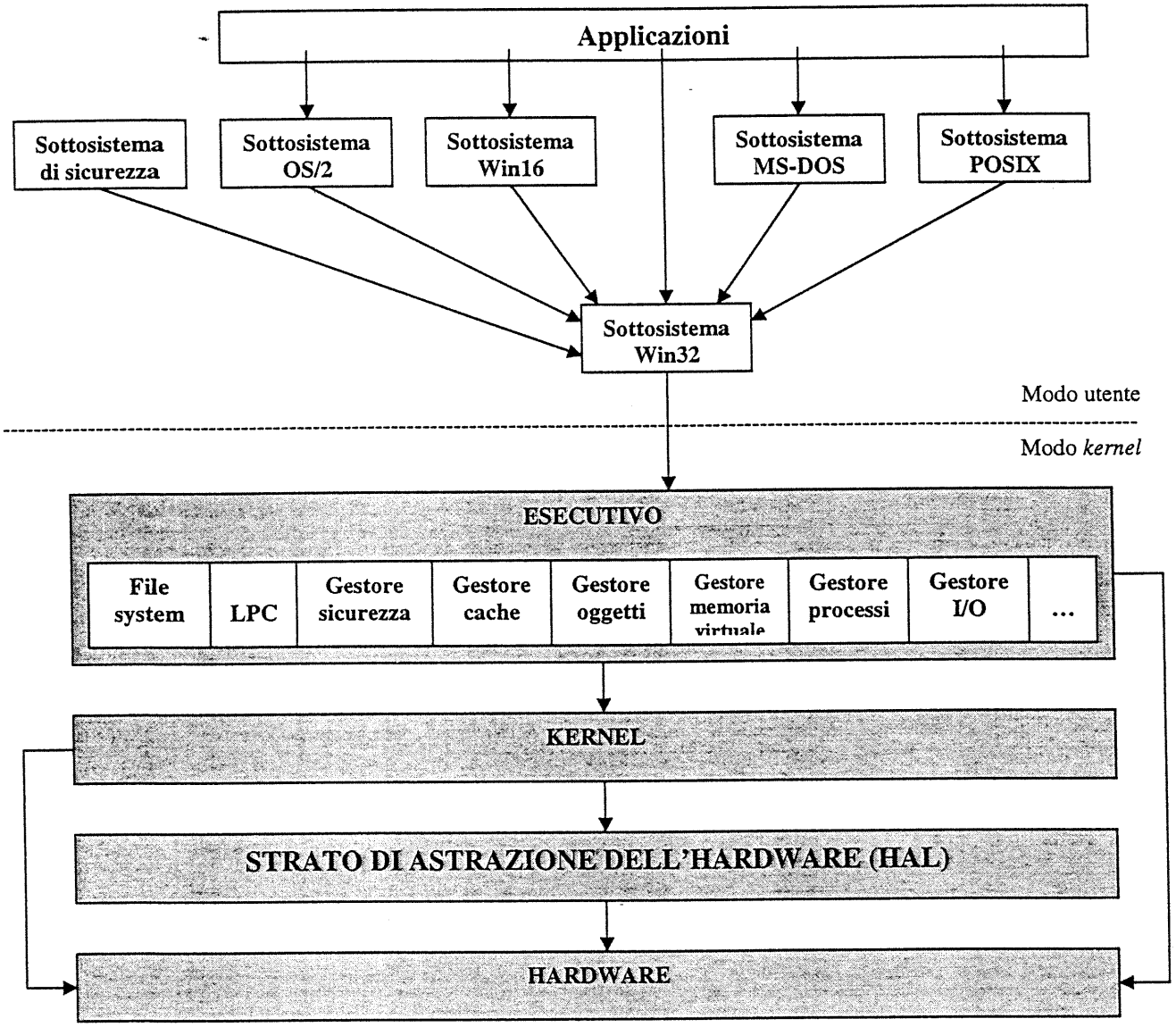
- lo scheduling dei processi e dei *thread*,
- la sincronizzazione del processore,
- la gestione degli *interrupt*,
- la gestione dei trap per le eccezioni,
- la gestione di problemi che sorgono in caso di caduta del sistema.

Tutto il codice e i dati del *kernel* sono sempre residenti in memoria centrale. Inoltre, il codice del *kernel* non può essere prelazionato, perciò i *context switch* non sono permessi quando il processore esegue codice appartenente al *kernel*. Altro fattore importante è che il *kernel* di NT cerca di mantenere un ambiente basato sugli oggetti: un "tipo di oggetto" in NT è un tipo di dato definito dal sistema che ha un insieme di attributi (valori dei dati) e possiede un insieme di metodi, cioè funzioni o operazioni per accedere e manipolare tali tipi di oggetti. In particolare, un "oggetto" è un'istanza di uno specifico tipo di oggetto. L'insieme di oggetti del *kernel* possono essere usati successivamente dall'esecutivo, il quale, a sua volta, li può usare per costruire eventualmente oggetti più complessi e renderli disponibili agli utenti.

Gli oggetti del kernel sono di due classi:

oggetti dispatcher: questi oggetti controllano la sincronizzazione e la comunicazione (*dispatching*) all'interno del sistema. Essi includono tipi di oggetti come gli eventi, i semafori, i

thread, i *timer* e i *mutex* (semafori di mutua eslusione). In particolare, l'oggetto evento è usato per registrare il verificarsi di un evento e per sincronizzare l'evento stesso con qualche altra azione eventualmente in corso di esecuzione; l'oggetto semaforo si comporta come un contatore al fine di controllare il numero di *thread* che accedono ad una risorsa; l'oggetto *thread* è l'entità eseguita dal *kernel* e associata ad un oggetto processo (oggetto di controllo); infine, gli oggetti *timer* sono usati come orologi per segnalare, ad esempio, la necessità di interrompere operazioni che si prolungano oltre un certo limite;



oggetti di controllo: questi oggetti sono relativi alle operazioni del codice in modalità *kernel*, ma non sono relativi alla sincronizzazione ed alla comunicazione. Essi includono tipi di oggetti come le chiamate di procedure asincrone (*asynchronuos procedure call*, APC), gli *interrupt*, la notifica di cali di tensione ed i processi. In particolare, la chiamata di procedura asincrona è usata per interrompere un *thread* in esecuzione; l'oggetto *interrupt* associa una procedura di servizio di un *interrupt* a una fonte di *interrupt*; l'oggetto di notifica di un calo di tensione è usato per chiamare automaticamente una specifica procedura a seguito appunto di un calo di tensione; ed infine, un oggetto processo rappresenta lo spazio di indirizzamento virtuale e le informazioni di controllo necessarie per eseguire l'insieme di *thread* associato a un processo.

3.3. L'esecutivo.

L'esecutivo è composto da moduli distinti e usa i servizi offerti dal *kernel* e dallo HAL. In particolare, esso offre ai vari sottosistemi (d'ambiente e di protezione) un ricco insieme di chiamate ai servizi del sistema. Inoltre, offre anche un supporto agli sviluppatori che desiderano estendere le funzionalità già esistenti del S.O. (solitamente tali estensioni coinvolgono la parte dedicata ai *device driver* ed ai *system driver*).

I vari componenti (moduli) che costituiscono l'esecutivo sono ben delineati, ed ogni modulo non può accedere direttamente alle strutture dati interne degli altri moduli. Le componenti più importanti dell'esecutivo sono: il gestore degli oggetti, il gestore della memoria virtuale, il gestore dei processi, il gestore dell'I/O, il gestore della sicurezza, la chiamata di procedura locale ed il gestore della memoria cache.

Daremo ora una descrizione sintetica di ognuno di questi componenti, soffermandoci con maggiore attenzione sul gestore della memoria *cache*.

• Il gestore degli oggetti

Abbiamo già accennato al fatto che Windows NT è un sistema orientato agli oggetti; esempi di oggetti sono: gli eventi, i semafori, i processi, i *thread*, le *directory*, i *file*, ... In particolare, tutte le componenti dell'esecutivo per creare, "aprire", "chiudere", cancellare e gestire un oggetto devono usare i servizi del gestore degli oggetti, in quanto è il gestore degli oggetti che supervisiona l'uso degli oggetti.

Inoltre, il gestore degli oggetti:

- permette di aggiungere dinamicamente un nuovo tipo di oggetto al sistema. È anche possibile distinguere fra oggetto temporaneo e oggetto permanente; gli oggetti permanenti rappresentano entità fisiche, come le unità a disco, quindi non vengono mai cancellati;
- permette di effettuare dei controlli di sicurezza e protezione sugli oggetti.

Ad esempio, quando un processo tenta di aprire un "oggetto", il gestore:

- controlla i diritti di accesso del processo;
- fissa la quantità massima di memoria allocabile ad un processo;
- fornisce una metodologia consistente per mantenere i riferimenti fra gli oggetti;
- fornisce una gerarchia di denominazione globale.

L'esecutivo di NT permette di associare un nome ad ogni oggetto; ecco allora che il gestore degli oggetti conserva uno spazio di nomi che è globale per Windows NT. E si può accedere a tutti gli oggetti nominati nel sistema attraverso tale spazio. I nomi degli oggetti hanno la stessa struttura dei nomi di *file* in MS-DOS od in UNIX. Quindi, c'è una *directory* radice globale creata dal gestore degli oggetti, ed i componenti dell'esecutivo possono creare *directory* e *subdirectory* a partire dalla *directory* radice, e creare gli oggetti sotto una qualsiasi *directory*.

• Il gestore della memoria virtuale

Il gestore della memoria virtuale (*Virtual Memory Manager*, VMM) è concepito assumendo che l'*hardware* sottostante sia in grado di associare indirizzi virtuali ad indirizzi fisici, ed adotta un meccanismo di paginazione. Tale gestore usa indirizzi a 32 bit (infatti Windows NT è un S.O. a 32 bit). In questo modo, ogni processo è dotato di uno spazio di indirizzamento virtuale di 4GB; poiché la dimensione di una pagina è di 4K (4096 byte), ecco allora che i 20 bit più significativi di un indirizzo logico indicano il numero di pagina, e i rimanenti 12 bit meno significativi indicano l'*offset* di pagina.

In particolare, dei 4GB di indirizzi virtuali associati ad ogni processo, i 2GB superiori costituiscono il cosiddetto "spazio *kernel*"; esso è identico per tutti i processi ed è dedicato ai dati e al codice del S.O., quindi è accessibile da NT in modalità *kernel*. Invece, i 2GB inferiori costituiscono lo "spazio utente", che è usato per informazioni specifiche del processo ed è accessibile in modalità utente e in modalità *kernel*. Lo spazio utente si riferisce ai dati privati del

processo (e quindi tali dati non sono accessibili da altri processi nel sistema); invece, lo spazio *kernel* si riferisce sempre alle stesse pagine fisiche le quali contengono il codice e i dati del S.O..

Al fine di non penalizzare eccessivamente le prestazioni, il gestore della memoria virtuale permette ai processi privilegiati di garantirsi un certo numero di pagine virtuali in memoria fisica, assicurandosi così il fatto che queste non saranno trasferite su disco.

Quando si verifica un *page fault,* il gestore della memoria virtuale trasferisce in memoria la pagina mancante ponendola sul primo *frame* della lista dei *frame* liberi. Se non vi sono *frame* disponibili sulla lista, allora NT adotta una strategia di sostituzione FIFO per ogni processo, cioè viene scaricata la pagina che, fra quelle presenti in memoria, è stata caricata per prima.

• Il gestore dei processi

Questo gestore è responsabile della creazione e della cancellazione dei processi e dei *thread.* Esso utilizza i servizi forniti dal *kernel* di NT per eseguire compiti come la sospensione di un processo, l'acquisizione di informazioni del processo e così via.

È da notare che il gestore dei processi non possiede alcuna informazione sulle relazioni parentali o sulle gerarchie fra i processi (questi dettagli sono lasciati alla gestione del particolare sottosistema d'ambiente relativo al processo). Ad esempio, se un'applicazione necessita la creazione di un processo nell'ambiente Win32, e quindi chiama **CreateProcess( )**, ecco allora che il sottosistema Win32 riceve un messaggio al quale reagisce chiamando il gestore dei processi al fine di creare un processo; a sua volta, il gestore dei processi chiama il gestore degli oggetti che crea un oggetto processo e restituisce a Win32 l'accesso all'oggetto. Il sottosistema Win32 chiama nuovamente il gestore dei processi per creare un *thread* relativo al processo e così via.

• Il gestore dell'I/O

Il gestore dell'I/O in NT è responsabile, tra le altre cose, della gestione del *file system,* dei *driver* dei dispositivi, dei *driver* di rete, e della gestione dei *buffer* per le richieste di I/O. Inoltre, collabora col gestore della memoria virtuale e col gestore della memoria cache.

Il gestore dell'I/O mette a disposizione sia operazioni sincrone che asincrone, e possiede strumenti per mettere in comunicazione due *driver.*

Tutte le richieste di I/O vengono ricevute e convertite dal gestore dell'I/O in una forma standard detta "pacchetto di richiesta di I/O" (*I/O Request Packet,* o IRP), dopo di che il gestore dell'I/O spedisce (inoltra) l'IRP al *driver* appropriato affinché esso possa soddisfare la richiesta. In particolare, ogni IRP mandato ad un *driver* rappresenta una richiesta di I/O pendente su quel *driver,* e rimane tale fino a che il *driver* non invoca la routine **IoCompleteRequest( )** per quel particolare IRP. Tale routine si conclude notificando che l'operazione richiesta è stata completata; ecco allora che il gestore dell'I/O dichiara evasa la richiesta.

In Windows NT non c'è alcun limite per quanto riguarda il tipo e il numero di periferiche che possono essere usate; questa caratteristica è dettata dal fatto che un S.O. al passo coi tempi deve essere progettato anche per essere estensibile, infatti ogni giorno vengono progettati e sviluppati nuovi tipi di dispositivi periferici, ognuno con il suo insieme di caratteristiche uniche.

• Il gestore della sicurezza

Questo modulo è responsabile del rafforzamento della politica di sicurezza del sistema. Infatti, ogni qualvolta un processo apre un oggetto, viene controllato se il processo gode dei diritti necessari per accedere a tale processo.

• La chiamata di procedura locale

La chiamata di procedura locale (*Local Procedure Call,* o LPC) è un meccanismo per lo scambio di messaggi tra due processi all'interno della stessa macchina. La LPC è modellata sul meccanismo di chiamata di procedura remota (*Remote Procedure Call,* o RPC) usata da molti S.O. per l'elaborazione distribuita in rete. Tuttavia, la LPC è ottimizzata per la comunicazione all'interno di un unico sistema, dove tutti i processi hanno accesso alla stessa memoria fisica. In altri termini, la

LPC è usata per trasferire richieste e risultati fra processi *client* e processi *server* all'interno di una stessa macchina.

• **Il gestore della memoria cache**
Verrà discusso in dettaglio nella prossima sezione.

3.4. I sottosistemi di ambiente e di protezione

Gerarchicamente, sopra l'esecutivo operano molti sottosistemi eseguiti in modo utente che si dividono in due categorie: i sottosistemi d'ambiente e i sottosistemi di protezione.

3.4.1. I sottosistemi d'ambiente

I sottosistemi d'ambiente sono processi eseguiti in modo utente, basati sui servizi dell'esecutivo di Windows NT, che permettono l'esecuzione di programmi sviluppati per altri S.O.; in questo modo, un programma scritto per MS_DOS (as esempio, Windows a 16 bit - Win16 -, o POSIX, o OS/2) può essere eseguito da NT nell'ambiente appropriato. In particolare, NT usa il sottosistema a 32 bit (Win32) per avviare ogni processo e per gestire tutto l'I/O relativo alla tastiera, al mouse ed allo schermo. Quando viene eseguita un'applicazione, il sottosistema Win32, tramite il gestore della memoria virtuale, verifica se il codice dell'applicazione è eseguibile in Win32; se non lo è avvia il sottosistema appropriato come processo in modo utente. Quindi, Win32 crea un processo per eseguire l'applicazione e trasferisce il controllo al sottosistema d'ambiente.

Ogni sottosistema d'ambiente fornisce una propria API (*Application Programming Interface*).

I sottosistemi che costituiscono NT possono ottenere i servizi dal *kernel* grazie al meccanismo di chiamata di procedura locale (LPC) che, come abbiamo visto, permette la trasmissione di messaggi ad alte prestazioni; infatti, i parametri passati ad una *system call* possono essere sottoposti a controlli di correttezza.

Grazie alla struttura modulare di NT è possibile aggiungere nuovi sistemi d'ambiente, senza che ciò abbia ripercussioni sull'esecutivo. Tuttavia, la compatibilità di NT a livello di codice eseguibile non è perfetta: in MS_DOS, ad esempio, le applicazioni possono accedere direttamente alle porte *hardware*, mentre per motivi di sicurezza e di affidabilità ciò non è permesso in NT.

3.4.2. I sottosistemi di protezione

I sottosistemi di protezione forniscono servizi di sicurezza. In particolare, prima che un utente possa accedere ad un oggetto in NT, esso deve avere un *account* e fornire la *password* associata all'*account*; a questo punto il sottosistema di sicurezza, attraverso le informazioni ottenute dal sottosistema di accesso o dal *server* di rete, determina l'eventuale convalida.

4. La memoria condivisa ed il *file mapping*

Supponiamo di avere due processi sullo stesso sistema che accedano allo stesso *file* presente su disco ed, in particolare, alla stessa porzione di dati del *file*; supponiamo, inolte, che i dati che vengono letti dal *file* su disco siano destinati ad essere riscritti sullo steso *file* su disco. Ecco allora che il S.O. Windows NT mette a disposizione delle funzionalità per condividere delle sezioni di *file* tra più processi in maniera consistente, utilizzando un metodo detto *file mapping*.

Con questo metodo, un processo non utilizza le *system call* per accedere ai dati, ma semplicemente mappa il *file* (o una sua porzione) presente su disco nel proprio spazio virtuale degli indirizzi. Se il processo prova ad accedere allo spazio di indirizzi virtuali connesso al *file* considerato (o ad una sua porzione) ed ottiene un *page fault*, il S.O. lo risolve allocando della memoria fisica e ottenendo i dati appropriati dal file del disco.

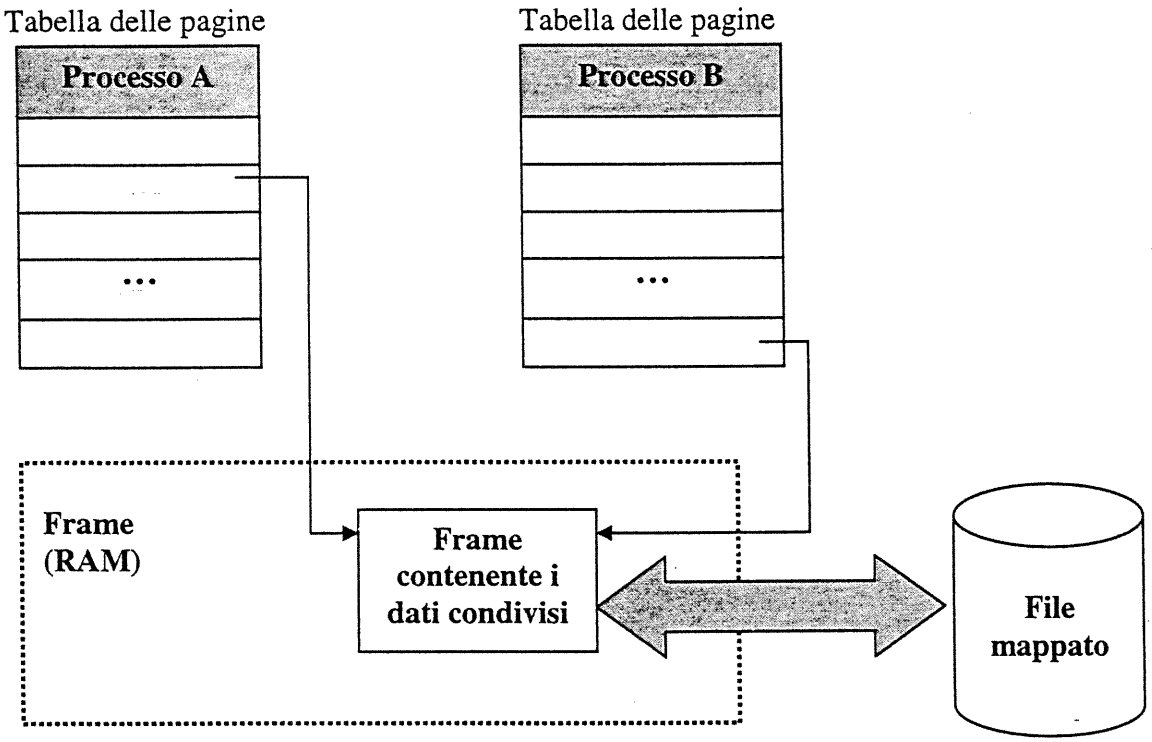
Analogamente, il processo può modificare i dati in memoria e il S.O. può, se richiesto, riscrivere i dati modificati sul *file* del disco.

Questo metodo ha un ulteriore vantaggio: tutti i processi che provano a mappare lo stesso *file* si ritrovano ad avere una mappa di indirizzi virtuale a cui corrisponde lo stesso insieme di pagine fisiche (*frame*); ecco che in questo modo viene implementata la condivisione del file.

In questo modo, le operazioni di lettura e di scrittura in quella regione di memoria vengono considerate come letture e scritture nel *file*, semplificando così l'uso del *file* medesimo, ed ogni scrittura da parte di ciascuno dei processi può essere vista da tutti gli altri processi che mappano la stessa porzione del *file*. In altri termini, tutti i processi vedono i dati in modo consistente, senza badare al fatto che un qualsiasi processo possa modificare i dati in qualsiasi istante (ricordiamo, comunque, che per garantire la coordinazione degli accessi ai dati condivisi, i processi in questione devono sincronizzarsi, in modo tale da non avere inaspettate conseguenze).

In particolare, per quanto riguarda l'operazione di chiusura del *file* condiviso essa comporta la scrittura su disco di tutti i dati presenti in memoria centrale e la loro rimozione dalla memoria virtuale dei processi.

La componente dell'esecutivo che supporta il *file mapping* è il gestore della memoria virtuale. Questa condivisione di memoria è illustrata nella seguente figura:



In Windows NT un blocco di memoria condivisa è detto "oggetto di sezione", ed è creato dal gestore della memoria virtuale in seguito alla richiesta di un *file mapping*.

In particolare, quando un processo non deve accedere a tutto il *file* mappato, ma ad una sua porzione, il gestore della memoria virtuale mappa nello spazio di indirizzi virtuali del processo solo la parte necessaria, questa parte è detta *view*. Concettualmente, una *view* è come una finestra sul *file* mappato, che permette l'accesso ad un campo limitato di *byte*.

Naturalmente, per uno stesso processo è possibile richiedere più *view* per lo stesso *file* mappato, e più processi possono avere differenti *view* sullo stesso *file* mappato. Quindi, con il meccanismo appena descritto, il sistema stesso può usare una *view* per esaminare lo spazio di indirizzamento di un oggetto un pezzo per volta.

Comunque, rimane la possibilità di proteggere una sezione di memoria condivisa. Ad esempio, le pagine della sezione possono essere di sola lettura, o di sola esecuzione, o ancora vi si può applicare il meccanismo di copia su scrittura (*copy on write*; fondamentale per l'operazione di **fork()** in cui lo spazio di indirizzi è inizialmente condiviso tra i processi padre e figlio). In

quest'ultimo caso, se due processi vogliono copie indipendenti dello stesso oggetto sezione, il gestore della memoria virtuale pone solo una copia condivisa in memoria fisica; se uno dei due processi tenta di scrivere su una delle pagine di tale sezione, il gestore fornisce al processo una copia privata della pagina che può poi essere modificata.

5 Il gestore della memoria *cache* in Windows NT

In molti S.O. il *caching* è eseguito dal *file system*; in Windows NT, invece, è eseguito dal gestore della memoria *cache*. Il gestore della memoria *cache* è una componente distinta dell'esecutivo e interagisce con i *file system driver*, il gestore della memoria virtuale, e il gestore dell'I/O.

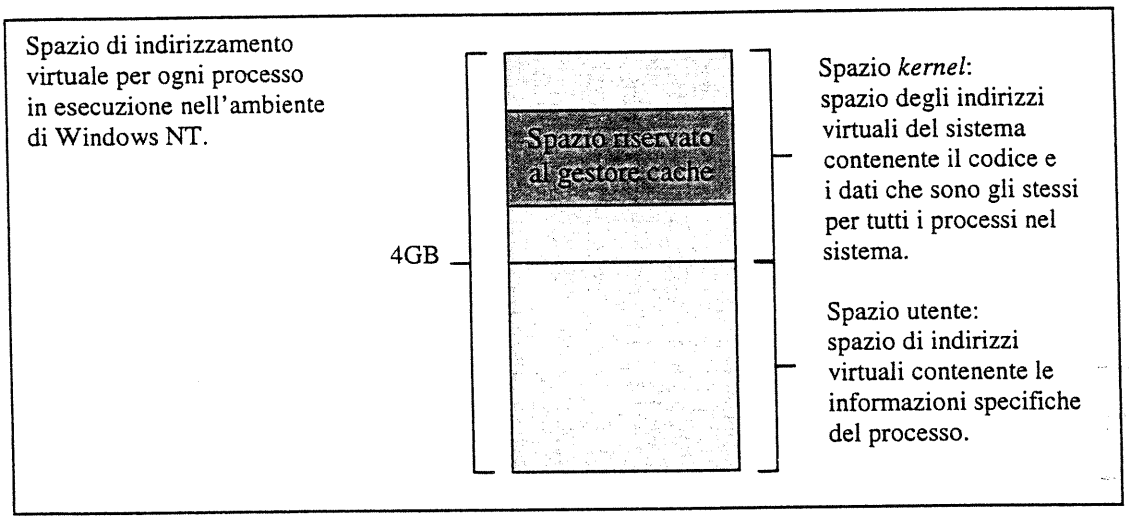
In particolare, il gestore della memoria *cache*:

- *permette di ottenere maggiori performance quando si hanno applicazioni che gestiscono grandi quantità di dati presenti in memoria secondaria.*
  Basti pensare, ad esempio, alle applicazioni che gestiscono i *database*;
- *permette di eseguire la lettura anticipata (read-ahead) su file di dati.*
  L'idea di base è di avere in anticipo nella *cache* quei dati che, presumibilmente, saranno richiesti in seguito. Ad esempio, se un'applicazione utente legge i primi 10K byte di un *file*, tipicamente il gestore della memoria *cache* legge in anticipo e, quindi, carica nella *cache* anche i successivi 64 Kbyte (poiché, normalmente, NT esegue le operazioni di I/O a blocchi di 64 Kbyte o 16 pagine). In questo modo, se l'applicazione utente necessita di accedere a quei dati, li trova già in memoria *cache* e non deve invece attendere parecchi cicli affinché i dati siano prelevati dalla memoria secondaria (tipicamente il disco). Ecco che i dati risultano già presenti nella *cache* prima ancora che l'applicazione richieda un accesso a tali dati.
  Ovviamente, la politica di lettura anticipata fornita dal gestore della *cache* trova la sua massima espressione di efficienza quando i file sono ad accesso sequenziale;
- *permette di differire la scrittura (delayed-write) su disco dei dati modificati presenti nella cache.*
  Mantenendo i dati modificati in memoria centrale per qualche tempo (4/5 secondi) prima di scriverli effettivamente su disco, il gestore della memoria *cache* fornisce una maggiore efficienza. Infatti, così facendo, esso può eventualmente accorpare più operazioni di scrittura che sono contigue in memoria e, di conseguenza, scrivere tutti i byte modificati con una sola operazione di I/O; ciò è tipicamente più efficiente che eseguire, di volta in volta, ogni piccola operazione di scrittura.
  Comunque, in Windows NT, volendo, si può adottare la strategia *write through caches* (presente anche nel S.O. MS_DOS), che consiste nel riscrivere il blocco su disco non appena il blocco viene modificato. Tale strategia offre, ovviamente, maggiore sicurezza poiché, come è noto, la memoria *cache* è una memoria volatile, ma richiede un maggior numero di accessi su disco. In particolare, quando è necessario il *caching write through*, il processo può impostare uno specifico *flag* all'apertura del *file* o chiamare esplicitamente una funzione per il trasferimento su disco dei dati della *cache*.

Come già accennato, nell'ambito del gestore della memoria virtuale, ogni processo in esecuzione nell'ambiente Windows NT ha a disposizione 4Gbyte di spazio di indirizzamento virtuale. In particolare, i 2Gbyte inferiori costituiscono lo spazio utente, mentre i 2Gbyte superiori costituiscono lo spazio *kernel*, il quale è riservato al S.O. ed è condiviso fra tutti i processi in esecuzione nel sistema. Il gestore della memoria virtuale alloca al gestore *cache* uno spazio di indirizzamento virtuale all'interno dei 2Gbyte superiori, che, solitamente, corrisponde a 512Mbyte. Ecco allora che ogni processo in esecuzione nel sistema ha accesso al campo di indirizzi virtuali riservati al gestore *cache*.

La seguente figura descrive la posizione dello spazio di indirizzi virtuali riservati al gestore della memoria *cache*.

Sebbene un certo spazio di indirizzi virtuali sia riservata per l'esclusivo uso del gestore *cache*, non si ha necessariamente una controparte a livello di pagine fisiche. Il numero di pagine fisiche che sono allocate al gestore *cache* è determinato (e costantemente aggiornato) dal gestore della memoria virtuale. In assenza di richiesta di memoria fisica da parte di altri processi utenti o di componenti di sistema, il gestore della memoria virtuale può scegliere di aumentare la quantità di memoria fisica allocata al gestore della *cache*. D'altra parte, su un sistema caricato "pesantemente", e con scarsa memoria fisica disponibile, il gestore della memoria può ridurre la quantità di memoria fisica allocata al gestore *cache*. In altri termini, la dimensione della memoria *cache* cambia dinamicamente, secondo la quantità di memoria libera nel sistema.



È importante notare che queste decisioni riguardanti l'allocazione della memoria fisica sono solo prerogativa del gestore della memoria virtuale.

5.1. Il *Virtual Address Control Block*

Il gestore della memoria *cache* usa il metodo *file mapping* per la gestione dei *file*. In particolare, ogni *view* mappata del *file* è descritta all'interno del gestore della *cache* tramite un blocco di controllo (o struttura dati) detto *Virtual Address Control Block* (VACB).

La dimensione di ogni *view* mappata è settata ad un valore costante dal gestore della *cache*, ed è uguale per tutti i *file* (solitamente corrisponde a 256K, cioè 64 pagine).

I VACB risiedono in un singolo vettore mantenuto dal gestore della *cache*. In particolare, ogni VACB contiene l'indirizzo virtuale associato alla *view* e lo scostamento (od *offset*) relativo al *file* della *view*, nonché il numero di processi che stanno usando la *view*.

Inoltre, il gestore della *cache* mantiene un vettore di puntatori ai VABC distinto per ogni *file* aperto. Questo vettore ha un elemento per ogni blocco del *file* della dimensione di una *view*; quindi, se, ad esempio, assumiamo che ogni *view* mappata sia di dimensione pari a 256K, allora un *file* di 2 MB sarebbe descritto da un vettore di puntatori di VACB di 8 elementi; ogni elemento di questo vettore punterebbe al VACB di quella *view*, se la *view* interessata fosse mappata, e sarebbe nullo altrimenti.

Ciò permette al gestore della *cache* di determinare velocemente se esiste già una *view* mappata contenente il campo di byte richiesto dall'applicazione utente. Se una tale *view* non esiste, il gestore della *cache* può creare una nuova *view* e allocare un VACB per rappresentarla.

Poiché lo spazio di indirizzamento virtuale riservato al gestore della *cache* è di dimensione fissa, è evidente che anche il vettore globale dei VACB ha dimensione fissa; ecco allora che il gestore della *cache* può non aver alcun VACB libero da allocare ad un *file* che necessita della creazione di una nuova *view*. In questo caso, il gestore della *cache* deve scaricare una *view* mappata

precedentemente per un *file* (tale *view* può riferirsi allo stesso *file* che richiede la nuova *view* da mappare, oppure può riferirsi ad un altro *file*); ovviamente, la sostituzione della *view* comporta anche l'aggiornamento del VACB e dei relativi puntatori, ossia si rimuove il VACB dal vettore di puntatori allocato per quel *file*, e poi si riassegna il VACB al nuovo *file*. Comunque, questa operazione non è richiesta frequentemente, poiché i VACB si liberano ogni volta che viene eseguita una operazione di *close* sul *file*; quindi, una richiesta di un VACB libero è generalmente soddisfatta.

Da notare che un campo di byte richiesto da un'applicazione utente può essere di dimensioni tali da richiedere la creazione di più *view* e, quindi, di più VACB, poiché la dimensione della *view* è costante. Comunque, il gestore della *cache* è sempre in grado determinare velocemente quale porzione del campo di byte richiesto è già contenuta in *view* mappate del file (se una qualche *view* esiste) e quale sottoinsieme necessita di essere mappato.

5.2. Operazioni di lettura e scrittura

Tutte le operazioni di *caching* su un file in Windows NT richiedono che il file a priori sia aperto.

Supponiamo che il processo utente faccia una richiesta di I/O su un *file* aperto. La richiesta è passata all'appropriato *file system driver*, ed è il *file system driver* stesso che dà il via all'operazione di *caching*. In particolare, per evitare sovraccarico inutile, il *file system driver* non inizia il *caching* per un *file* fino a che per esso non è prevista un'operazione di I/O. Perciò il *caching* viene generalmente iniziato per un *file* solo quando la prima operazione di I/O è ricevuta dal *file system driver*.

Quindi, se l'operazione di *caching* non è stata iniziata in precedenza per quel determinato *file*, il *file system driver* invoca il gestore *cache*. Su ricevimento di tale richiesta, il gestore *cache* chiama il gestore della memoria virtuale per creare un oggetto sezione rappresentante il *file mapping*; questo è fatto per l'intero *file*.

Successivamente, quando un processo tenta di accedere ai dati appartenenti al *file*, il gestore *cache* dinamicamente mappa le *view* del file nel proprio spazio di indirizzamento virtuale. In particolare, essendo il campo di indirizzi virtuali riservato al gestore della *cache* di dimensione fissa, il gestore *cache* può dover scaricare una o più viste mappate precedentemente per essere in grado di creare una nuova vista.

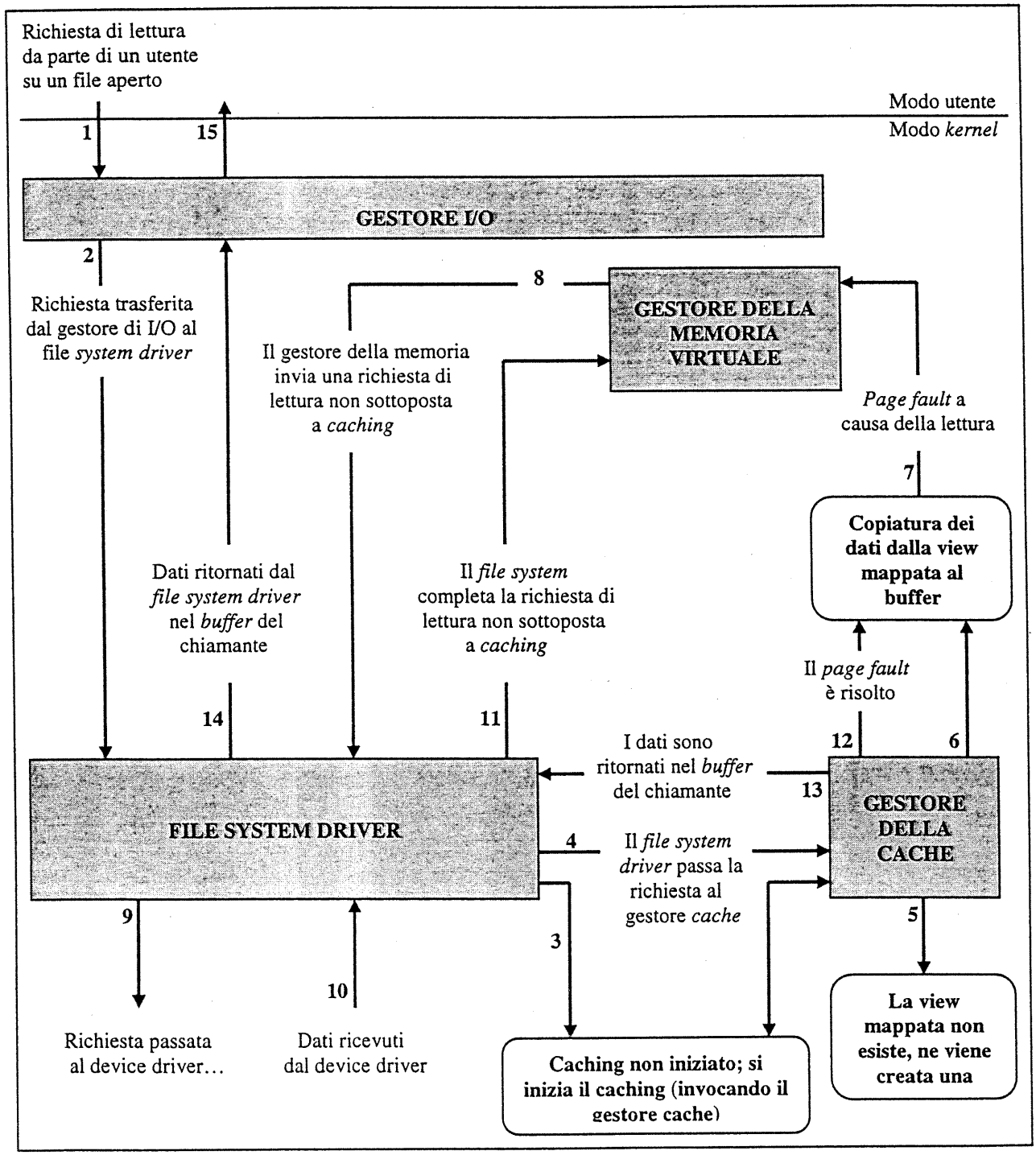
Per capire meglio il ruolo svolto dal gestore della *cache* nel servire le richieste di I/O, esamineremo di seguito la sequenza tipica di passi eseguita per le operazioni di lettura e scrittura.

5.2.1. Operazione di lettura

Consideriamo una operazione di lettura richiesta da un'applicazione utente. La figura che segue rappresenta graficamente la sequenza di operazioni eseguite per soddisfare tale richiesta. In particolare, una spiegazione di ogni passo nella figura è fornita di seguito.

1. L'applicazione utente esegue un'operazione di lettura, il cui controllo è trasferito al gestore di I/O.
2. Il gestore di I/O trasferisce la richiesta di lettura all'appropriato *file system driver* usando un IRP.
3. Il *file system driver* riceve la richiesta di lettura e verifica se l'operazione di lettura è diretta ad un *file* che è aperto. In particolare, se l'operazione di *caching* non è ancora iniziata per quel *file*, il *file system driver* la inizia invocando il gestore della memoria cache, il quale chiede al gestore della memoria virtuale di creare un *file mapping* (quindi un oggetto di sezione) per quel file, altrimenti si passa al passo **4**.
4. Il *file system driver* passa la richiesta di lettura al gestore della *cache* usando una routine chiamata **CcCopyRead()** (a meno che non si chieda esplicitamente che l'operazione di lettura non usi la *cache*). Il gestore della *cache* è ora responsabile dell'esecuzione di tutti i passi necessari per trasferire i dati nel *buffer* del chiamante.

5. Il gestore della *cache* esamina le sue strutture dati per determinare se c'è una *view* mappata del *file* contenente il campo di byte richiesto dall'utente. In particolare, il gestore determina l'elemento del vettore di puntatori di VACB del file corrispondente in base allo scostamento specificato dalla richiesta; l'elemento in questione o è nullo o punta ad un VACB. Nel primo caso la *view* mappata non esiste; ecco, allora, che il gestore della cache mappa la *view* e alloca un VACB per rappresentarla, ciò comporta l'aggiornamento dei VACB e dei relativi puntatori.
6. Il gestore della *cache* semplicemente esegue un'operazione di copiatura della memoria, dalla *view* mappata al *buffer* dell'utente. Se ciò riesce, si passa al passo **13**, altrimenti si passa al passo **7**.



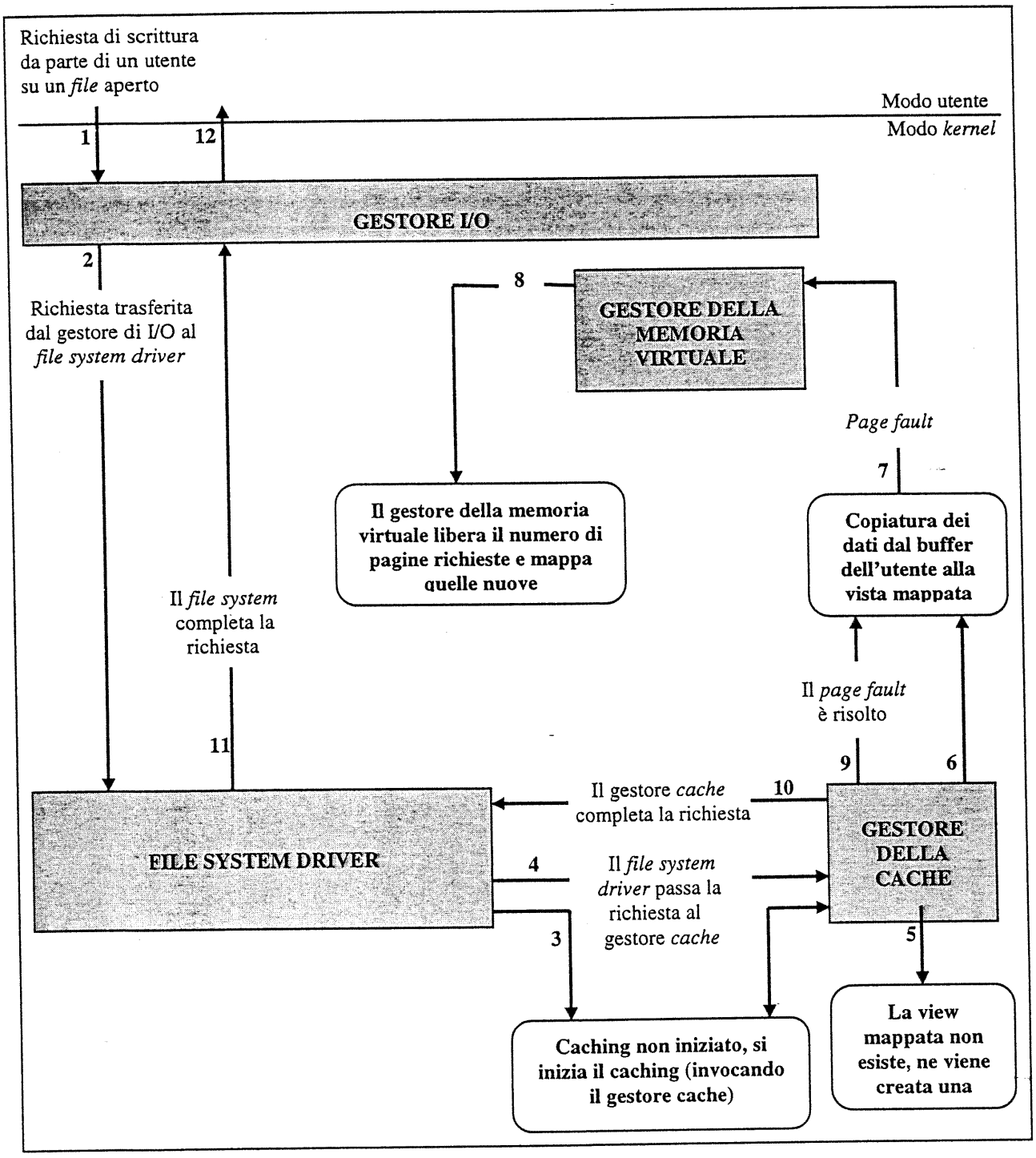
7. Se la *view* mappata del file non ha come controparte una pagina fisica contenente i dati richiesti, allora si verifica un *page fault* ed il controllo è trasferito al gestore della memoria virtuale.
8. Il gestore della memoria virtuale alloca dei *frame* che saranno usati per contenere i dati richiesti (i quali hanno causato il *page fault*), e poi invia una richiesta di lettura non sottoposta a *caching* (*noncached paging I/O*) al *file system* attraverso il gestore di I/O. Si noti che, per semplificare la rappresentazione grafica, quest'ultimo passaggio non è indicato nella figura, ma è quello che effettivamente accade.
9. Quando il *file system driver* riceve la richiesta di lettura non sottoposta a *caching*, crea una corrispondente richiesta di I/O per ottenere i dati dalla memoria secondaria e spedisce tale richiesta al *device driver* appropriato.
10. Il *driver* del dispositivo appropriato, sotto il controllo del *file system*, ottiene i dati dalla memoria secondaria (o, eventualmente, dalla rete) e li restituisce al *file system*.
11. Il *file system* completa la richiesta di lettura non sottoposta a *caching* fatta dal gestore della memoria virtuale al passo **8.**, in modo tale che esso possa caricare i dati nella memoria *cache*.
12. L'istruzione che era stata interrotta a causa del *page fault* viene rieseguita, poiché ora i dati si trovano nella memoria *cache*. Quindi, il gestore della *cache* completa l'operazione di copiatura dalla *view* mappata per il *file* nel *buffer* del chiamante; questa volta, la trascrizione si completa senza incorrere in un *page fault*.
13. Il gestore della *cache* ritorna il controllo al *file system driver* (dopo che i dati nella memoria cache sono stati copiati nel buffer dell'utente). Tali dati rimarranno comunque nello spazio di indirizzi virtuali riservati al gestore della memoria *cache*.
14. Il *file system* completa l'IRP originaria e la trasferisce al gestore dell'I/O.
15. Il gestore dell'I/O completa e dichiara evasa la richiesta di lettura originaria dell'utente.

5.2.2. Operazione di scrittura

Supponiamo ora che un'applicazione utente richieda una operazione di scrittura; come si potrà notare, la sequenza di operazioni che analizzeremo è simile a quella descritta precedentemente per l'operazione di lettura. Nella figura che segue è rappresentata graficamente tale sequenza, una spiegazione di ogni passo della figura è fornita di seguito.

1. L'applicazione utente esegue una operazione di scrittura, il cui controllo è trasferito al gestore di I/O.
2. Il gestore di I/O trasferisce la richiesta di scrittura all'appropriato *file system driver* usando un IRP.
3. Il *file system driver* riceve la richiesta di scrittura e verifica se l'operazione di scrittura è diretta ad un *file* che è aperto. In particolare, se l'operazione di *caching* non è ancora iniziata per quel file, il *file system driver* la inizia invocando il gestore della memoria *cache*, il quale chiede al gestore della memoria virtuale di creare un *file mapping* (quindi un oggetto di sezione) per quel *file*, altrimenti viene eseguito il passo **4**.
4. Il *file system driver* passa la richiesta di scrittura al gestore della cache usando una routine chiamata **CcCopyWrite( )**.
5. Il gestore della *cache* esamina le sue strutture dati per determinare se c'è una *view* mappata per il *file* contenente il campo di byte che devono essere sottoposti a modifica da parte dell'utente. In particolare, il gestore determina l'elemento del vettore di puntatori di VACB del *file* corrispondente in base allo scostamento specificato dalla richiesta; l'elemento in questione o è nullo o punta ad un VACB. Nel primo caso, la *view* mappata non esiste; ecco allora che il gestore della *cache* mappa la view e alloca un VACB per rappresentarla, ciò comporta l'aggironamento dei VACB e dei relativi puntatori.
6. Il gestore della *cache* semplicemente esegue un'operazione di copiatura della memoria, dal *buffer* dell'utente alla *view* mappata. Se ciò riesce, si va al passo **10**, altrimenti si va al passo **7**.

7. Se la *view* mappata del file non ha come controparte una pagina fisica contenente i dati richiesti, allora si verifica un *page fault* e il controllo è trasferito al gestore della memoria virtuale.
8. Il gestore della memoria virtuale alloca dei *frame* che saranno usati per contenere i dati richiesti (i quali hanno causato il *page fault*). Nel nostro caso, si assume che intere pagine vengano riscritte dall'utente; in un tale scenario, né il gestore della *cache*, né il gestore della memoria virtuale leggono precedentemente i dati esistenti nel disco prima di modificare tali dati.
9. L'istruzione che era stata interrotta a causa del *page fault* viene rieseguita, poiché i dati ora si trovano nella memoria *cache*. Quindi, il gestore della *cache* completa l'operazione di copiatura dal *buffer* del chiamante alla *view* mappata per il *file*; questa volta, la trascrizione si completa senza incorrere in un *page fault*.



10. Il gestore della *cache* ritorna il controllo al *file system driver*. È da notare che i dati dell'utente ora risiedono in memoria centrale, ma non sono ancora stati scritti nella memoria secondaria. L'effettivo trasferimento dei dati nella memoria secondaria verrà fatto in un secondo tempo dal gestore della *cache*.

11. Il *file system* completa l'IRP originaria e la trasferisce al gestore dell'I/O.

Il gestore dell'I/O completa e dichiara evasa la richiesta di scrittura originaria dell'utente.

Riferimenti bibliografici

[1] Silberschatz, A. e Baer Galvin, P., *Sistemi Operativi. Quinta Edizione*, Addison Wesley Longman, 1998.

[2] Nagar, R., *Windows NT File System Internals: A Developer's Guide* (prima edizione), O'Reilly, 1997.

RAPPORTI DIDATTICI
del DIPARTIMENTO DI MATEMATICA APPLICATA

n. 1/2002 Marco Li Calzi
Lecture notes on Financial Markets.

n. 2/2002 Fulvio Piccinonno
Conoscenze informatiche di base. Appunti per la preparazione all'esame.

n. 3/2002 Andrea Ellero
Appunti di "Maple"

n. 4/2002 Alberto Zorzi *(presentato da Ellero)*
Appunti di crittografia

n. 5/2002 Alberto Zorzi *(presentato da Ellero)*
Elementi grafici per matematica

n. 6/2002 Andrea Ellero
Problemi di programmazione biobiettivo e frazionaria

n. 7/2002 Marco Corazza
La revisione statistica del portafoglio azionario: i principali modelli classici.

n.8/2002 Stefania Funari
Applicazioni con EXCEL alle decisioni finanziarie

n. 9/2002 Stefania Funari
Un' introduzione al linguaggio HTML e alla creazione di una pagina Web

n. 10/2002 F.Mason, E.Moretti, F.Piccinonno
Appunti di logistica

n. 11/2002 F. Mason
MAT 55

n.12/2003 Alberto Zorzi (presentato da Ellero)
Esercizi di topologia

n.13/2003 D.Favaretto, E.Moretti
Metodi Matematici per la Gestione delle Aziende.
Complessità computazionale, programmmazione lineare intera, scheduling della produzione

n. 14/2003 Francesco Mason
Simulazione con il metodo Montecarlo

n. 15/2003 M. Corazza, S. Facco
Il gestore *cache* in Windows NT