diegochine

APPUNTI DI SO

Capitolo 1: introduzione

Introduzione

Per definire cos'è un **sistema operativo** si possono utilizzare due differenti definizioni.
La prima considera il sistema operativo come una **macchina estesa** (*visione top-down),* che si basa sul concetto di astrazione come chiave per gestire la complessità. Dato che l'architettura (insieme di istruzioni, organizzazione della memoria, I/O e bus) della maggior parte dei computer è primitiva e complicata, almeno a livello di linguaggio macchina, si preferisce astrarre quelle che sono le attività "complesse", ad esempio la scrittura su disco dei dati. Una buona astrazione suddivide un'attività quasi impossibile in due attività gestibili: la prima riguarda la definizione e implementazione delle astrazioni, la seconda l'impiego di queste astrazioni per risolvere un problema reale. I file sono un esempio di questa astrazione: permettono di scrivere in modo semplice dei dati su disco, senza la necessità di conoscere i dettagli di come l'operazione viene effettivamente eseguita sul disco. Compito del sistema operativo è creare una buona astrazione e successivamente implementare e gestire gli oggetti astratti così creati.
Una seconda definizione vede il SO come **gestore di risorse** *(visione bottom-up)*: il SO esiste per gestire tutti i pezzi di un sistema complesso, e il suo lavoro è fornire un'allocazione ordinata e controllata di processori, memorie e unità I/O fra i vari programmi che lo richiedono. La gestione delle risorse include il **multiplexing** delle risorse, nello spazio e nel tempo. Quando una risorsa è condivisa temporalmente, programmi o utenti diversi fanno a turno ad usarla (ad esempio, CPU assegnata per un certo tempo massimo ad un processo). Per quanto riguarda lo spazio, invece di alternarsi gli utenti utilizzano ognuno una parte della risorsa (ad esempio, memoria principale suddivisa fra i vari programmi in esecuzione; disco fisso che contiene file di più utenti).

Storia dei Sistemi Operativi

Va in mona.Volentieri

Tipologie di sistemi operativi

Ordinati dal in ordine decrescente, dal più grande al più piccolo:

- **SO per mainframe:** sono strettamente orientati all'esecuzione di numerosi lavori alla volta, molti dei quali richiedono ingenti quantità di I/O. Tre servizi principali: batch (lavori di routine senza la presenza di utenti che interagiscono con la macchina), elaborazione delle transazioni (trattazione di moltissime unità di lavoro di dimensione ridotta), timesharing (consentire a molteplici utenti remoti di eseguire lavori sul computer tutti insieme). Questi tre servizi sono spesso eseguiti tutti insieme. Esempio è l'OS/390, ma stanno venendo sostituiti da Linux.
- **SO per server:** girano su server, servono molteplici utenti allo stesso tempo attraverso una rete e consentono agli utenti di condividere risorse hw e sw. Tipici SO server sono Solaris, FreeBSD, Linux, Windows Server 201x.
- **SO multiprocessore:** spesso sono varianti dei SO per server, con caratteristiche per comunicazione, connettività e coerenza in quanto è necessario gestire più CPU sulla stessa macchina. Molti SO, come Windows e Linux, sono multiprocessore.
- **SO per personal computer:** supportano la multiprogrammazione, il loro scopo è fornire un valido supporto ad un utente singolo. Esempi sono Linux, FreeBSD, Windows 7/8/10, OSX.
- **SO per computer palmari (PDA):** mercato dominato da Android e iOS. Necessaria comunicazione con molti sensori: GPS, fotocamera..
- **SO integrati (embedded):** girano su computer che controllano dispositivi elettronici come forni a microonde, TV, automobili. Non accettano sw installato dall'utente, tutto il software è su ROM - non è quindi necessario un livello di protezione tra le applicazioni. Esempi diffusi sono QNX e VxWorks.
- **SO per sensori:** utilizzati in reti di piccoli sensori (nodi) che comunicano tramite WiFi, ogni nodo è un piccolo computer alimentato a batterie. Devono lavorare per molto tempo in autonomia, la rete deve tollerare la rottura di singoli nodi. Ogni nodo esegue un SO che generalmente è event driven, ovvero risponde ad eventi esterni o esegue misure periodiche sulla base di un clock interno. Il più famoso è TinyOS.
- **SO real-time:** per questi sistemi il tempo rappresenta un parametro chiave (ad esempio, sistemi di controllo del processo industriale). Possono essere **hard real-time system** (l'operazione deve avvenire entro un dato tempo o il sistema fallisce), oppure **soft real-time system** (mancare una scadenza, anche se non auspicabile, è tollerabile). Esempio è il sistema e-Cos.
- **SO per smart card:** girano su smart card, dispositivi delle dimensioni di una carta di credito contenenti un chip CPU. Presentano problematiche di alimentazione e memoria; alcune sono Java oriented (è presente un interprete per la JVM). Sono SO spesso molto primitivi.

Concetti di base dei SO

Un concetto chiave è quello di **processo**, che in sostanza è un programma in esecuzione. Ogni processo ha associato un suo **spazio degli indirizzi**, un elenco di locazioni di memoria da 0 ad un massimo che il processo può leggere e scrivere. Lo spazio degli indirizzi contiene il programma eseguibile, i dati del programma e il suo stack. Inoltre, ogni processo ha associato un insieme di risorse (registri, come PC e SP), elenco di file aperti, allarmi in sospeso, processi relativi e tutte le informazioni necessarie a far girare il programma. Un processo quindi è un contenitore che raccoglie tutto ciò che serve per far girare un programma. In molti SO, tutta l'informazione riguardante ciascun processo è salvata in una tabella chiamata **tabella di processo**, sostanzialmente un array (o linked list) di strutture, una per ogni processo in essere. Generalmente un processo, tramite chiamate di sistema, può creare processi figli, generando una struttura ad albero; Il termine **IPC - interprocess communication** indica la comunicazione che può avvenire tra un processo padre ed i suoi processi figli.
Ad ogni persona autorizzata ad usare il sistema è assegnato uno **UID** (user identification); ogni processo che parte ha l'UID della persona che lo ha fatto partire. Processi figli hanno l'UID del processo padre. Gli utenti possono inoltre appartenere a gruppi, ognuno dei quali con un proprio **GID** (group identification). Un preciso UID, il ***superuser*** (UNIX) o ***Administrator*** (Windows) ha un potere speciale e può violare molte delle regole di protezione.

Un altro concetto chiave comune a tutti i SO è il **file system**. Esso è una struttura che controlla come i dati vengono salvati e recuperati dalla memoria, offrendo un'interfaccia semplice e astratta, indipendente dalle peculiarità dei dischi e dei dispositivi di I/O. Per creare, cancellare, leggere e scrivere files sono necessarie chiamate di sistema. *Concetti di directory, path name, root directory.* Prima che un file possa essere letto o scritto, deve essere aperto; se è consentito l'accesso il sistema restituisce un numero, il **descrittore del file**, da usare come referenza per il file stesso.
Un altro concetto importante in UNIX è il **file speciale**: essi sono pensati per far sì che i dispositivi di I/O siano visti come file. Due tipi:

- **File speciali a blocchi**: usati per modellare dispositivi costituiti da un insieme di blocchi indirizzabili casualmente, come i dischi.
- **File speciali a caratteri**: usati per modellare stampanti, modem e altri dispositivi che accettano una sequenza di caratteri in input o output.

L'ultima caratteristica importante riguarda sia file che processi ed è la **pipe**. Essa è una specie di pseudofile che può essere usato per connettere due processi che vogliono comunicare. Quando due processi A e B vogliono comunicare, A scrive sulla pipe come fosse un file di

output, mentre B la legge come fosse un file di input: quindi le comunicazioni tra processi in UNIX sono molto simili alla normale scrittura e lettura di un file.

Chiamate di sistema (syscall)

Se un processo sta eseguendo un programma utente in modalità utente e necessita di un servizio del sistema, come leggere i dati da un file, deve eseguire un'operazione di trap per trasferire il controllo al SO. Il SO analizza i parametri, realizza la chiamata e poi restituisce il controllo all'istruzione successiva. Quindi, in un certo senso una syscall è una speciale chiamata di procedura che entra in modalità kernel per eseguire operazioni che appunto possono essere eseguite solo in modalità kernel.

Struttura dei SO

Sistemi monolitici. L'organizzazione più vecchia e comune; in questo approccio l'intero SO viene eseguito come un programma singolo e in modalità kernel. Il SO è scritto come una raccolta di procedure, linkate all'interno di un solo grande programma binario eseguibile. Ogni componente del SO fa parte del kernel. La comunicazione diretta tra i componenti rende questo tipo di sistemi operativi molto efficiente, però può risultare molto difficile isolare fonti di errore o malfunzionamenti proprio per il fatto che tutti i componenti sono raggruppati nel kernel. Inoltre, dato che l'intero codice viene eseguito con piena possibilità di accesso al sistema, i sistemi con kernel monolitici sono particolarmente esposti a danni derivati da codice accidentalmente o intenzionalmente errato (come quello dei virus).

Sistemi a livelli. L'approccio a livello nel progetto dei sistemi operativi tenta di risolvere queste problematiche raggruppando in strati i componenti che svolgono funzioni simili. Ogni strato comunica esclusivamente con quello immediatamente inferiore o superiore. Gli strati di livello inferiore forniscono servizi a quelli di livello superiore per mezzo di un'interfaccia che nasconde il modo in cui il lavoro viene svolto. I SO a strati sono molto più modulari di quelli monolitici, in quanto ogni strato è modificabile senza apportare necessariamente modifiche agli altri strati. Un sistema modulare è composto da componenti auto contenuti riutilizzabili in tutto il sistema. Ogni componente nasconde il modo in cui svolge il proprio lavoro e presenta un'interfaccia standard utilizzabile dagli altri componenti per richiedere i suoi servizi. La modularità impone che il sistema sia ben strutturato e consistente, spesso semplificando le operazioni di validazione, debug e modifica. Tuttavia una richiesta di un processo utente in un sistema a strati potrebbe doverne attraversare molti prima di poter essere servita. Le prestazioni possono dunque risultare peggiori rispetto a quelle di un kernel monolitico, inoltre dato che i vari strati hanno libero accesso al sistema, i kernel a strati sono anche esposti a danni derivati da codice accidentalmente o intenzionalmente errato.

Sistemi microkernel. L'idea alla base del progetto del microkernel è di raggiungere un'alta stabilità suddividendo il SO in piccoli moduli ben definiti, uno solo dei quali - il microkernel - eseguito in modalità kernel. In particolare, facendo girare ogni driver e file system come processo utente separato, un loro difetto potrà bloccare un singolo componente ma non l'intero sistema. Al contrario, in un sistema monolitico con tutti i driver nel kernel, un driver difettoso può facilmente provocare l'immediato blocco del sistema. Quindi, un SO a microkernel fornisce solo un piccolo numero di servizi nel tentativo di mantenere il kernel piccolo e scalabile. Questi servizi riguardano in genere la gestione a basso livello della memoria, la comunicazione tra processi (IPC) e i meccanismi base di sincronizzazione che permettono ai processi di cooperare. Il microkernel offre un altro grado di modularità che lo rende espandibile, portabile e scalabile. Tale modularità si ottiene tuttavia al prezzo di un elevato numero di comunicazioni tra i moduli che finisce per peggiorare le prestazioni del sistema.

SO di rete. Un SO di rete permette ai processi di accedere a risorse che si trovano su altri computer indipendenti collegati in rete. La struttura di molti SO di rete e distribuiti si basa spesso sul modello client/server. I computer client richiedono risorse attraverso un appropriato protocollo e i server rispondono fornendo quelle appropriate. In tali reti, i progettisti devono considerare attentamente le problematiche di gestione dei dati e delle comunicazioni tra computer. In un ambiente di rete un processo è eseguibile sul computer in cui è stato creato o su un altro computer della rete. In alcuni SO di rete gli utenti possono specificare esattamente dove i loro processi saranno eseguiti, in altri invece è il SO a prendere queste decisioni. I file system di rete sono componenti molto importanti nei SO di rete: al livello più basso gli utenti acquisiscono le risorse su un'altra macchina collegandosi esplicitamente a quella e recuperando i file, mentre quelli di alto livello, invece, permettono agli utenti di accedere a file remoti come se questi fossero nel sistema locale.

SO distribuiti. Un SO distribuito è un singolo SO in grado di gestire risorse distribuite su più sistemi di elaborazione. I sistemi distribuiti forniscono l'illusione che diversi computer costituiscano un unico, potente computer, così che un processo possa accedere a tutte le risorse del sistema indipendentemente dalla locazione del processo all'interno della rete di computer del sistema distribuito. I SO distribuiti sono spesso difficili da realizzare e richiedono algoritmi complessi che permettano ai processi di comunicare e condividere dati.

Macchine virtuali (VMs). È un'astrazione software di un sistema di elaborazione, spesso in esecuzione sopra ad un SO nativo. Una VM permette la coesistenza di diverse istanze di SO anche differenti ma eseguibili contemporaneamente. Inoltre è possibile utilizzare

l'emulazione, ovvero software o hardware che imita le funzionalità di hardware o software non presente nel sistema. Importante è la macchina virtuale di Java, chiamata JVM.

Obiettivi dei SO

- **Efficienza.** Un SO efficiente raggiunge un alto throughput e un basso tempo medio di turnaround. Il throughput misura la quantità di lavoro che un processore può completare in un certo intervallo di tempo. Uno dei compiti di un SO è quello di fornire servizi a molte applicazioni, un SO efficiente minimizza il tempo necessario a fornire tali servizi.
- **Robustezza.** Un SO robusto è tollerante ai guasti ed affidabile: il sistema non deve bloccarsi del tutto a causa di errori in applicazioni isolate o nell'hardware e, nel caso in cui dovesse bloccarsi, lo deve fare in modo graduale e controllato nel tentativo di minimizzare la quantità di lavoro perso e di evitare danni all'hardware del sistema. Tale SO continuerà a fornire servizi alle applicazioni a meno di un guasto all'hardware.
- **Scalabilità.** Un SO scalabile è in grado di utilizzare a pieno nuove risorse che vengono aggiunte al sistema. Se un sistema non fosse scalabile si arriverebbe al punto in cui l'aggiunta di ulteriori risorse al sistema non porterebbe alcun beneficio. Un SO scalabile può rapidamente variare il suo grado di multiprogrammazione. La scalabilità è una caratteristica particolarmente importante nei sistemi multiprocessore, in quanto al crescere del numero di processori nel sistema dovrebbe idealmente corrispondere la crescita delle capacità di elaborazione in proporzione al numero di processi.
- **Espandibilità.** Un SO espandibile è in grado di adattarsi bene alle nuove tecnologie e presenta la capacità di svolgere compiti che vanno oltre a quelli pensati in fase di progetto del SO stesso.
- **Portabilità.** Un SO portabile è progettato per poter funzionare su un gran numero di configurazioni hardware. La portabilità delle applicazioni è anche importante, in quanto sviluppare applicazioni è costoso; il vantaggio di rendere eseguibile una stessa applicazione su diverse configurazioni hardware permette di ridurre i costi di sviluppo. Il SO è fondamentale nel raggiungere questo tipo di portabilità.
- **Sicurezza.** Un SO sicuro impedisce agli utenti ed al software di accedere a servizi e risorse senza l'opportuna autorizzazione. Con il termine protezione ci si riferisce ai meccanismi che realizzano la politica di sicurezza del sistema.
- **Interattività.** Un SO interattivo permette alle applicazioni di rispondere velocemente alle azioni dell'utente o agli eventi.
- **Usabilità.** Un SO usabile ha la potenzialità di essere adatto ad un gran numero di utenti. Sistemi di questo tipo sono normalmente provvisti di un'interfaccia utente molto facile da usare.

Capitolo 2: processi e thread

Processi

In ogni sistema multiprogrammato la CPU passa da processo a processo rapidamente; nonostante essa ne possa eseguire solo uno alla volta, nel corso di un secondo può lavorare su parecchi di loro, dando l'illusione di parallelismo. In questo contesto si parla di ***pseudoparallelismo***, in contrapposizione con il vero parallelismo hardware dei sistemi multiprocessore.

Per semplificare la gestione di questo pseudoparallelismo è stato sviluppato un modello secondo il quale tutto il software eseguibile sul computer (talvolta anche il SO) è organizzato in un certo numero di **processi sequenziali**, o processi per semplicità. Un processo è un'istanza di un programma in esecuzione, compresi i valori attuali del PC, dei registri e delle variabili. Concettualmente ogni processo ha la sua CPU virtuale, anche se nella realtà essa viene continuamente scambiata tra i vari processi; questo meccanismo è detto **multiprogrammazione**.

Ogni processo ha un proprio **spazio di indirizzamento** composto da:

- **Code segment**: memorizza il codice che viene eseguito dal processore
- **Data segment**: memorizza variabili e memoria allocata dinamicamente
- **Stack**: memorizza istruzioni e variabili locali per le chiamate di procedura attive

I SO hanno bisogno di un modo per creare i processi. Gli eventi che possono causare la creazione di un processo sono quattro:

1. Inizializzazione del sistema;
2. Esecuzione di una chiamata di sistema di creazione di un processo;
3. Richiesta dell'utente di creare un processo;
4. Inizio di un job in modalità batch.

All'avvio del sistema operativo vengono in genere creati parecchi processi. Alcuni sono attivi, in primo piano ed interagiscono con gli utenti (umani); altri invece (i **daemon**) sono eseguiti in background e non sono associati ad un utente in particolare ma hanno funzioni specifiche.

Tecnicamente, un nuovo processo è creato avendone un altro che esegue una chiamata di sistema di creazione di processo. Ciò che il processo fa è eseguire una chiamata di sistema operativo per creare un nuovo processo.

In UNIX, per creare un processo nuovo esiste una sola chiamata di sistema: **fork**. Questa chiamata crea un clone esatto del processo che esegue la chiamata. Solitamente il processo figlio esegue una chiamata di sistema per cambiare la propria immagine di memoria ed eseguire un nuovo programma.

In Windows, una sola chiamata di funzione Win32, la **CreateProcess**, gestisce sia la creazione del processo sia il caricamento del programma corretto nel nuovo processo. Sia in UNIX che in Windows, il genitore ed il figlio hanno il loro spazi degli indirizzi personali.

Una volta creato, un processo comincia ad eseguire il suo compito. Prima o poi il processo terminerà, di solito a causa di una delle condizioni seguenti:

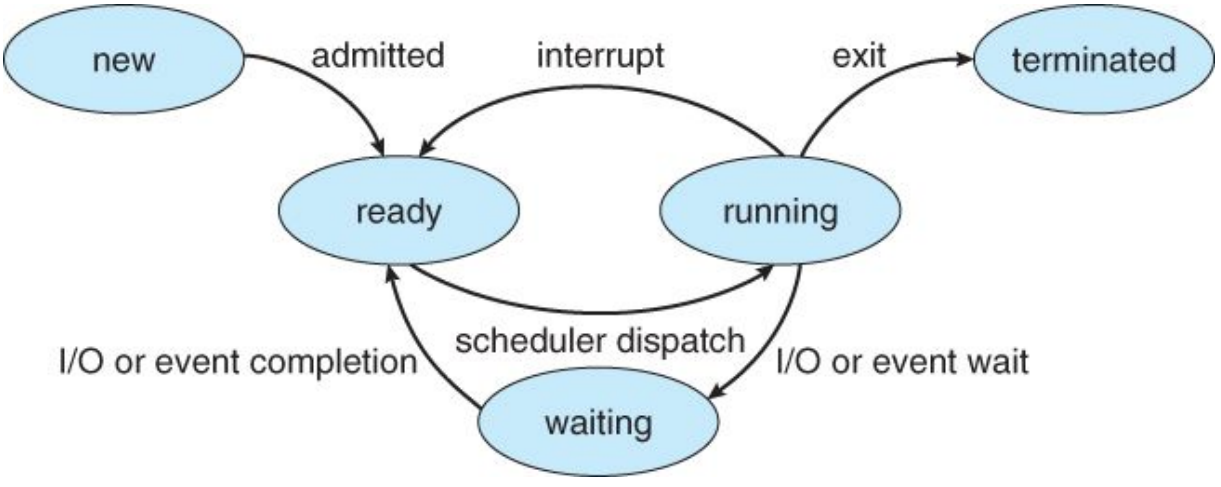
1. Uscita normale (condizione volontaria);
2. Uscita su errore (condizione volontaria);
3. Errore critico (condizione involontaria);
4. Terminazione da parte di un altro processo (condizione involontaria).

La maggior parte dei processi termina perchè ha completato il proprio lavoro, e lo indica al SO eseguendo una chiamata di sistema, che è **exit** in UNIX e **ExitProcess** in Windows.

In alcuni sistemi, quando un processo ne crea un altro, il processo genitore e il processo figlio restano in un certo modo associati. Il processo figlio può a suo volta creare altri processi, generando una gerarchia. In UNIX, un processo, tutti i suoi figli e gli ulteriori discendenti costituiscono un gruppo di processi. Quando un utente invia un segnale dalla tastiera, il segnale viene inviato a tutti i membri del gruppo di processi al momento associati alla tastiera; ogni processo può individualmente decidere come gestire il segnale. Windows invece non ha il concetto di gerarchia di processi, tutti i processi sono uguali. Quando viene creato un processo, al genitore viene dato un token (**handle**) che può usare per controllare il figlio. Tuttavia, esso può liberamente passare il token ad altri processi, invalidando la gerarchia; in UNIX ciò non è consentito.

Un processo può trovarsi in 5 differenti stati:

1. **new**: il processo è stato creato;
2. **ready**: può essere eseguito; è temporaneamente sospeso perchè sta venendo eseguito un altro processo.
3. **running**: in esecuzione;
4. **waiting**: in attesa di un qualche evento esterno (e.g. operazione di I/O)
5. **terminated**.



Per implementare il modello di processo, il SO mantiene una tabella (un array di strutture), la **tabella dei processi**, con una voce per processo (***process control block, PCB***). Questa voce contiene importanti informazioni sullo stato del processo, incluso il PC, lo SP, l'allocazione della memoria, lo stato dei suoi file aperti, le informazioni relative a gestione e scheduling e qualunque altra cosa relativa al processo da salvare quando passerà da running a ready/waiting.

Considerando l'uso della CPU da un punto di vista statistico si ottiene un buon modello per la multiprogrammazione. Supponendo che un processo impieghi una frazione *p* del suo tempo in attesa di I/O, ed avendo in memoria contemporaneamente *n* processi (***grado di multiprogrammazione***), risulta:

$$Uso\ CPU\ =\ 1\ -\ p^n$$

Thread

Un processo può essere visto come un modo per raggruppare risorse relazionate. Un processo ha uno spazio degli indirizzi contenente testo, programma, dati e potenzialmente altre risorse. L'altro concetto relativo ad un processo è quello del **thread**. Un thread ha un contatore di programma che tiene conto di quale istruzione eseguire come successiva; ha dei registri contenenti le sue variabili di lavoro attuali; ha uno stack proprio. Sebbene un thread debba essere eseguito in un processo, il thread e il relativo processo sono concetti differenti: i processi sono usati per raggruppare risorse, i thread sono entità schedulate per l'esecuzione sulla CPU. Il valore aggiunto dei thread è il fatto di consentire molteplici esecuzioni che hanno luogo nello stesso ambiente del processo, con un alto grado di indipendenza l'una dall'altra. Essi generalmente condividono spazio degli indirizzi, memoria fisica, dischi, stampanti e altre risorse. Dato che condividono alcune delle proprietà dei processi, a volte vengono chiamati **lightweight process**. Il termine **multithreading** è usato anche per descrivere la possibilità di molteplici thread nello stesso processo. Dato che ogni thread può accedere all'intero spazio degli indirizzi del processo, un thread potrebbe leggere, scrivere o anche cancellare lo stack di un altro thread. Non c'è protezione perchè è impossibile averla e perchè non dovrebbe essere necessaria: a differenza di processi diversi che potrebbero essere di utenti diversi ed entrare in conflitto, i thread di un singolo processo sono proprietà di un singolo utente, indi per cui dovrebbero cooperare e non entrare in conflitto.
Nonostante tutto, i thread causano anche alcune complicazioni nel modello di programmazione. Si consideri di avere un processo padre che voglia creare un processo figlio; se il padre ha più thread, dovrebbe averli anche il figlio? Come dovrebbero essere gestiti? Inoltre, un'altra complicazione deriva dal fatto che i thread condividono molte strutture dati. Cosa succede se un thread chiude un file mentre un altro lo sta leggendo? Questi problemi

sono risolvibili usando accortezza nella progettazione e programmazione dei programmi multithread.
IEEE ha definito uno standard per i thread, supportato dalla maggior parte dei sistemi UNIX che è **pthread**.

Implementare thread nello spazio utente

Per realizzare un pacchetto di thread abbiamo sostanzialmente due modalità: nello spazio utente e nel kernel. Consideriamo il primo metodo: il pacchetto di thread è interamente contenuto nello spazio utente; il kernel non ne è a conoscenza, infatti gestirebbe processi ordinari a singolo thread. Il primo vantaggio è che può essere implementato un pacchetto di thread utente su un SO che non supporta i thread. Con questo metodo i thread sono realizzati tramite una libreria. Utilizzando questo metodo è necessario che ogni processo abbia una **tabella dei thread** per tenere traccia di tutti i thread relativi a quel processo, la quale è gestita da un sistema run-time.
Quando un thread fa qualcosa che potrebbe causare il suo blocco richiama una procedura di sistema run-time che controlla se il thread deve entrare in stato di waiting; in caso affermativo, salva i registri del thread nella tabella dei thread e cerca un thread pronto da eseguire e ricarica i registri con i nuovi valori. Un simile scambio di thread è di almeno un ordine di grandezza più veloce che fare il trap nel kernel. Le procedure che salvano lo stato dei thread e lo scheduler sono solo locali, quindi richiamarle è molto più efficace che fare una chiamata al kernel; inoltre non sono richieste trap, cambi di contesto, ecc. Questo rende lo scheduling dei thread molto veloce. Altri vantaggi sono il consentire ad ogni processo di avere il proprio algoritmo di scheduling personalizzato e non pesare a livello di spazio nel kernel.
Il principale problema di questo approccio è il fatto che le chiamate di sistema bloccanti, anche se eseguite all'interno di un thread, possono bloccare l'intero processo e di conseguenza anche tutti gli altri thread, cosa che colliderebbe con il concetto stesso di thread. Stessa cosa succede nel caso di page fault causato da un singolo thread: il kernel, ignorando l'esistenza dei thread, blocca l'intero processo in attesa dell'I/O da disco, sebbene altri thread potrebbero essere eseguiti. Un'ultima ed importante ragione contro l'uso dei thread nello spazio utente è che i programmatori generalmente vogliono i thread proprio in applicazioni dove i thread si bloccano spesso e generano molte chiamate di sistema, come in un web server multithread.

Implementare thread nel kernel

Supponendo che il kernel sia a conoscenza dei thread e li possa gestire, non occorre un sistema di run-time in ogni kernel, e non c'è la tabella dei thread in ogni processo. Il kernel dispone di una propria tabella dei thread che tiene traccia di tutti i thread di sistema. Quando un thread vuole crearne uno nuovo o distruggerne uno esistente, fa semplicemente una chiamata kernel. La tabella dei thread tiene tutti i registri di tutti i thread, gli stati ed altre

informazioni; sono le stesse dei thread nello spazio utente, solo che ora stanno nel kernel. Tutte le chiamate che potrebbero bloccare un thread sono realizzate come chiamate di sistema, a costi notevolmente più alti di una chiamata di una procedura di un sistema run-time.
A causa dei costi superiori di creazione e distruzione dei thread nel kernel, alcuni sistemi "riciclano" i thread (segnandoli come non eseguibili quando al posto di distruggerli e riattivandoli quando vi è necessità di nuovi thread), risparmiando un po' di overhead. Inoltre, se un thread in un processo causa un page fault, il kernel può facilmente verificare se il processo ha altri thread eseguibili e in caso eseguirli. Il loro principale svantaggio è che il costo di una chiamata di sistema è considerevole, così se le operazioni sui thread (creazione, chiusura..) sono frequenti si incorre in molto overhead.

Altri approcci di implementazione dei thread

Scheduler activation. Gli obiettivi delle scheduler activation sono di imitare la funzionalità dei thread del kernel, ma con prestazioni migliori ed una maggiore flessibilità di solito associata ai pacchetti di thread implementati nello spazio utente. Usando le scheduler activation, il kernel assegna un certo numero di processori virtuali (possono essere CPU reali nei sistemi multiprocessore) ad ogni processo e fa sì che il sistema run-time dello spazio utente allochi i thread ai processori. Il numero di processori virtuali allocato ad un processo è inizialmente uno, ma il processo può richiedere di averne di più e restituirne nel caso non fossero più necessari. L'idea base che fa funzionare questo schema è che quando il kernel sa che un thread si è bloccato (page fault o syscall bloccante) esso notifica il sistema run-time del processo. Questo meccanismo è chiamato **upcall**. In questo il sistema run-time può rischedulare i suoi thread; più tardi, quando il kernel verifica che il thread originale può tornare ad essere eseguito, esegue un'altra upcall.
Thread pop-up. I thread sono molto utili nei sistemi distribuiti. Considerando un sistema di gestione dei messaggi in ingresso, con un thread bloccato in attesa di un messaggio; è possibile adottare un approccio in cui l'arrivo di un messaggio causa la creazione di un nuovo thread, detto thread pop-up. Il vantaggio principale è che, dato che sono creati da zero, non hanno storia (registri, stack..) che deve essere recuperata. Il risultato è che la latenza fra l'arrivo del messaggio e l'inizio dell'elaborazione può essere brevissima.

Scheduling

Quando un computer è multiprogrammato, è molto probabile che vi siano due o più processi o thread che competono per la CPU nello stesso istante (entrambi sono in stato ready). La parte del SO che effettua la scelta è chiamata **scheduler** e l'algoritmo è chiamato **algoritmo di scheduling**. Lo scheduling è relativamente poco importante sui PC (generalmente è attivo un solo processo principale e i computer sono diventati così veloci nel corso degli anni che la CPU

raramente è una risorsa scarsa), ma risulta invece sensibilmente importante ad esempio per i server in rete.

Quasi tutti i processi alternano passi di elaborazione (CPU) a richieste di I/O (disco). Quando un processo spende la maggior parte del suo tempo in elaborazione, viene definito **compute-bound** o **CPU-bound**, mentre se spende la maggior parte del tempo in attesa di I/O allora è definito **I/O-bound**. I processi CPU-bound hanno burst di CPU lungo e attese di I/O poco frequenti, mentre quelli I/O-bound hanno burst di CPU brevi e attese di I/O frequenti.

Un problema chiave dello scheduling è dato dai tempi. Le situazioni in cui è necessario schedulare sono varie. Primo, quando è creato un nuovo processo, si deve decidere se deve essere eseguito il processo padre o quello figlio. Secondo, quando un processo viene terminato va scelto un altro processo da eseguire. Terzo, quando un processo si blocca su I/O, su un semaforo o per qualunque altro motivo, bisogna selezionare un altro processo da eseguire. Quarto, una decisione di scheduling va presa quando si verifica un interrupt di I/O.
Gli algoritmi di scheduling sono suddivisibili in due categorie, rispetto al loro comportamento nei riguardi degli interrupt del clock. Un algoritmo di scheduling **non preemptive** preleva un processo per eseguirlo e poi lo lascia andare finché non si blocca (e.g. I/O o attesa di un altro processo) o finchè non rilascia volontariamente la CPU. Un algoritmo di scheduling **preemptive** prende un processo e lo lascia eseguire per un tempo massimo prefissato; se alla fine dell'intervallo il processo è ancora in esecuzione, il processo è sospeso e lo scheduler ne prende un altro da eseguire.
L'uso di un tipo di algoritmo di scheduling piuttosto che un altro dipende dal contesto in cui ci si trova. Vediamo tre casi differenti:

1. Sistemi batch: sono spesso accettabili algoritmi non preemptive o algoritmi preemptive con lunghi tempi di attesa, dato che i sistemi batch si occupano di elaborazioni lunghe.
2. Sistemi interattivi: l'uso della preemption è essenziale per evitare che un processo monopolizzi la CPU e neghi il servizio ad altri.
3. Sistemi real-time: la preemption non è sempre necessaria, poiché i processi sanno che non possono essere eseguiti per lunghi periodi di tempo e generalmente fanno il loro lavoro e si bloccano rapidamente.

Gli obiettivi degli algoritmi di scheduling variano da sistema a sistema, anche se alcuni sono generali:

- Tutti i sistemi
    - Equità - dare ad ogni processo una equa condivisione della CPU
    - Applicazione della policy - assicurarsi che la policy dichiarata sia attuata
    - Bilanciamento - tenere impegnate tutte le parti del sistema
- Sistemi batch

    - Massimizzare il throughput
    - Minimizzare il tempo di turnaround
    - Mantenere la cpu sempre impegnata
- Sistemi interattivi
    - Rispondere alle richieste rapidamente (minimizzare tempo di risposta)
    - Adeguatezza - far fronte alle aspettative dell'utente
- Sistemi real-time
    - Rispetto delle scadenze - evitare la perdita dei dati
    - Prevedibilità - evitare il degrado della qualità nei sistemi multimediali

Il **throughput** è il numero di lavori completato dal sistema fino ad un certo punto. Il **tempo di turnaround** è statisticamente il tempo medio trascorso da quando un lavoro batch è sottoposto al sistema a quando è terminato; corrisponde quindi al tempo di attesa medio dell'utente per rilevare l'output.

Scheduling nei sistemi batch

First-come first-served. Algoritmo non preemptive in cui i processi sono semplicemente assegnati alla CPU nell'ordine in cui la richiedono. Fondamentalmente c'è una singola coda di processi in stato ready, quando un processo termina se ne preleva uno nuovo dalla coda. È un algoritmo semplice da capire e da programmare, ma presenta un forte svantaggio nel caso in cui ci sia un solo processo CPU-bound e molti processi I/O bound.

Shortest job first. Algoritmo non preemptive che parte dal presupposto che i tempi di esecuzione siano conosciuti in anticipo. Quando parecchi lavori equivalenti sono sistemati nella coda di input in attesa di essere avviati, lo scheduler preleva per primo il lavoro più breve. Notiamo che l'algoritmo SJF è ottimale solo nel caso in cui tutti i lavori siano disponibili contemporaneamente.

Shortest remaining time next. È la versione preemptive del SJF; lo scheduler sceglie sempre il processo il cui tempo che manca alla fine dell'esecuzione è il più breve.

Scheduling nei sistemi interattivi

Scheduling round-robin. Ad ogni processo è assegnato un intervallo di tempo, chiamato **quanto**, durante il quale gli è consentito di essere eseguito. Se alla fine del quanto il processo è ancora in esecuzione, la CPU viene assegnata ad un altro processo. Se il processo si blocca o termina prima che sia trascorso il quanto la CPU viene riassegnata prima. Un quanto troppo breve causa troppi cambi di contesto e riduce l'efficienza della CPU; un quanto troppo alto

può causare una risposta scadente alle richieste rapide interattive. Un compromesso ragionevole è un quanto di 20-50 ms.

Scheduling a priorità. Lo scheduling round-robin presuppone che tutti i processi siano ugualmente importanti. L'idea di base dello scheduling a priorità è chiara: a ciascun processo è assegnata una priorità e il processo eseguibile con la priorità più alta sarà il prossimo ad essere eseguito. Per impedire che i processi ad alta priorità siano eseguiti indefinitamente, lo scheduler può abbassare la priorità del processo attualmente in esecuzione ad ogni scatto del clock, oppure assegnare un quanto di tempo massimo. Le priorità possono essere assegnate staticamente o dinamicamente. È spesso conveniente raggruppare i processi in classi di priorità e usare lo scheduling a priorità fra le classi, ma all'interno di ciascuna classe usare lo scheduling round-robin.

Code multiple. I processi sono suddivisi in classi di priorità. I processi nella classe maggiore sono eseguiti per un quanto, quelli della classe successiva per due quanti, la classe seguente quattro e così via. Quando un processo utilizza tutti i quanti assegnati viene spostato in basso di una classe.

Shortest process next. L'analogo del SJF dei sistemi batch su sistemi interattivi. L'idea è quella di fare delle stime basate sull'esperienza maturata ed eseguire il processo con il tempo di esecuzione stimato più breve. Attraverso un parametro a si considerano le stime $aT_0 + (1-a)T_1$. Questa tecnica è anche chiamata aging. Con $a = \frac{1}{2}$ si hanno queste stime:
$T_0 \text{---} \; T_0/2 + T_1/2 \; \text{---} \; T_0/4 + T_1/4 + T_2/2 \text{---} T_0/8 + T_1/8 + T_2/4 + T_3/2$

Scheduling garantito. In un sistema con n processi utente, si promette che ogni processo utente avrà circa 1/n della potenza della CPU. L'algoritmo calcola per ogni processo il rapporto fra tempo di CPU consumato e tempo di CPU di cui si ha diritto, quindi esegue il processo con il rapporto minore finché il suo rapporto non supera quello del suo vicino concorrente.

Scheduling a lotteria. Vengono dati ai processi "biglietti della lotteria" per le diverse risorse del sistema, come il tempo di CPU. Ogni volta che deve essere presa una decisione di scheduling si pesca un biglietto e il processo con quel biglietto si aggiudica la risorsa. Applicato allo scheduling della CPU, il sistema dovrebbe fare un'estrazione 50 volte al secondo, assegnando ogni volta come premio 20 ms di tempo di CPU. Ai processi più importanti possono essere assegnati biglietti extra. Lo scheduling a lotteria è molto reattivo e processi cooperanti possono, se vogliono, scambiarsi i biglietti.

Scheduling fair-share. Ad ogni utente viene assegnata una frazione di CPU e lo scheduling raccoglie i processi in modo tale da farla rispettare, indipendemente dal numero di processi che il singolo utente sta eseguendo.

Scheduling nei sistemi real-time

Gli eventi a cui un sistema real-time può dover rispondere possono essere categorizzati come periodici (avvengono a intervalli regolari) o non periodici (avvengono imprevedibilmente). A seconda di quanto tempo ogni evento richiede per essere elaborato, non sempre può essere possibile gestirli tutti. Per esempio, se ci sono m eventi periodici e l'evento i avviene con un periodo P_i e richiede C_i secondi di tempo di CPU per gestire ogni evento, allora il carico può essere gestito solo se $\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$. Un sistema real-time che soddisfa questo criterio è detto schedulabile.

Policy e meccanismo

Talvolta accade che un processo abbia molti figli eseguiti sotto il suo controllo; è del tutto possibile che il processo principale abbia un'idea eccellente di quale dei suoi figli sia il più importante o critico. Tuttavia nessuno degli scheduler visti finora accetta alcun input dai processi utente riguardo alle decisioni di scheduling; di conseguenza, lo scheduler raramente prende la decisione migliore. La soluzione a questo problema sta nel separare il **meccanismo di scheduling** dalla **politica di scheduling**, un principio stabilito da tempo. L'algoritmo di scheduling è quindi in qualche modo parametrizzato, ma i parametri possono essere riempiti dai processi utente.

Capitolo 3: gestione della memoria

La parte del SO che gestisce (in parte) la gerarchia della memoria è chiamata **gestore della memoria**. Il suo compito è gestire con efficienza la memoria: tener traccia di quali parti della memoria sono in uso, allocare memoria ai processi secondo le loro necessità e liberarla a lavoro terminato.

Spazio degli indirizzi (di nuovo)

È necessario un certo livello di astrazione per utilizzare in modo ottimale e sicuro la memoria. Questa astrazione è lo spazio degli indirizzi. Così come un processo crea una specie di CPU astratta per eseguire i programmi, lo spazio degli indirizzi crea una specie di memoria astratta per farci vivere i programmi. Uno **spazio degli indirizzi** è l'insieme degli indirizzi che un processo può usare per indirizzare la memoria. Ogni processo ha il suo spazio degli indirizzi

personale, indipendente da quello di altri processi. Non è semplice dare ad ogni programma il suo spazio degli indirizzi. Il metodo spiegato di seguito è semplice ma caduto in disuso.

L'idea si basa su una versione particolare della **rilocazione dinamica**. Quello che fa è mappare lo spazio degli indirizzi di ogni processo su una parte diversa di memoria fisica in modo semplice. Si equipaggia la CPU con due registri hw speciali, il **registro base** e il **registro limite**. I programmi sono caricati in posizioni di memoria consecutive dovunque ci sia spazio e senza riposizionamento durante il caricamento. Al momento dell'esecuzione di un processo, il registro base è caricato con l'indirizzo fisico dove comincia il suo programma in memoria e il registro limite è caricato con la lunghezza del programma. Ogni volta che un processo consulta la memoria, prima di inviare l'indirizzo sul bus di memoria l'hw della CPU aggiunge automaticamente il valore di base all'indirizzo generato tramite il processo. Contemporaneamente controlla se l'indirizzo offerto sia uguale o maggiore del valore nel registro limite, nel cui caso è generato un errore e l'accesso interrotto. Uno svantaggio di questo approccio è la necessità di eseguire una somma e un confronto con ogni riferimento alla memoria.

Swapping

Se la memoria fisica del computer è abbastanza ampia da contenere tutti i processi, il metodo visto finora funziona. Nella realtà tuttavia la quantità totale di RAM necessaria per tutti i processi è spesso molto di più di quanto possa stare in memoria. Per gestire il sovraccarico di memoria sono stati sviluppati negli anni due approcci generali. La prima strategia, chiamata **swapping**, consiste nel prendere ciascun processo nella sua totalità, eseguirlo per un certo tempo, quindi rimetterlo nuovamente su disco. L'altra strategia, chiamata **memoria virtuale**, consente ai programmi di essere eseguiti anche quando sono solo parzialmente nella memoria principale. Quando lo swapping crea molteplici spazi vuoti nella memoria, è possibile combinarli tutti in un unico spazio vuoto spostando tutti i processi il più in basso possibile. Questa tecnica è conosciuta come **memory compaction**. Di solito non viene fatta perché richiede molto tempo di CPU. Bisogna inoltre tenere in considerazione quanta memoria allocare per un processo quando viene creato o quando ne viene fatto lo swap da disco: se i processi hanno dimensione fissa allora l'allocazione è semplice, se invece possono variare allora bisogna agire di conseguenza, ad esempio allocando memoria extra ad ogni swap.

Gestione della memoria libera

Quando la memoria è assegnata dinamicamente, il SO deve gestirla. In termini generali, ci sono due modalità di tenere traccia dell'utilizzo della memoria: bitmap e liste.

Gestione della memoria con bitmap. La memoria, con una bitmap, è divisa in unità di allocazione piccole come qualche parola o grandi come molti kilobyte. Ad ogni unità di

allocazione corrisponde un bit della bitmap, che è 0 se l'unità è libera e 1 se è utilizzata. Poiché la dimensione della bitmap dipende solo dalla dimensione della memoria e dalla dimensione dell'unità di allocazione, una bitmap fornisce un modo semplice per tener traccia di parole di memoria in una quantità fissa di memoria. Il problema principale è che, dovendo portare un processo di k unità in memoria, il gestore deve cercare nella bitmap una serie di k bit uguali a zero: questa operazione è molto lenta e gioca a sfavore della bitmap.

Gestione della memoria con liste collegate. Un altro sistema per tenere traccia della memoria è mantenere delle liste di segmenti di memoria allocati e liberi, in cui un segmento o contiene un processo o è uno spazio vuoto fra due processi. Ogni voce della lista specifica il tipo di spazio (vuoto/processo), l'indirizzo da cui parte, la lunghezza e il puntatore alla voce successiva.
Quando processi e spazi vuoti sono tenuti su una lista ordinata per indirizzo, diversi algoritmi possono essere usati per allocare la memoria per un processo:

- **First fist**: il gestore della memoria scorre la lista dei segmenti finchè non trova uno spazio vuoto abbastanza grande. Lo spazio è suddiviso in due parti, una per il processo e l'altra per la memoria inutilizzata. Dato che cerca il meno possibile, first fit è un algoritmo veloce.
- **Next fit**: lavora allo stesso modo del first fit, di cui è una variante minore, ma tiene traccia di ogni posto dove ha trovato uno spazio adatto. La volta seguente cerca nella lista a partire dal punto dove era rimasto l'ultima volta, invece di partire dal principio. Ha prestazioni leggermente peggiori del first fit.
- **Best fit**: cerca all'interno della lista lo spazio più piccolo tra quelli adatti; l'idea è ottimizzare la richiesta di spazi disponibili.
- **Worst fit**: prende sempre lo spazio disponibile più grande. Prestazioni non ottimali.
- **Quick fit**: mantiene liste divise per alcune delle più comuni dimensioni richieste (e.g., una tabella con n voci, di cui la prima punta ad una lista di spazi di 4 KB, la seconda punta alla lista degli spazi da 8 KB e via così).

Best fit è più lento di first fit e genera anche un maggior spreco di memoria rispetto a first fit e next fit. First fit genera, in media, spazi vuoti più grandi. Con quick fit, la ricerca degli spazi è molto veloce, ma presenta lo stesso svantaggio di tutti gli schemi ordinati per dimensione, ovvero trovare i vicini di un processo è un processo dispendioso. I primi quattro algoritmi possono essere velocizzati mantenendo liste separate per i processi e gli spazi. In questo caso si può anche ordinare per dimensione la lista degli spazi; così facendo first fit e best fit hanno le stesse prestazioni.

Memoria virtuale

L'idea alla base della **memoria virtuale** è che ogni programma ha il proprio spazio degli indirizzi personale, suddiviso in pezzi chiamati **pagine**. Ogni pagina è un intervallo di indirizzi contigui. Queste pagine sono mappate sulla memoria fisica, ma non tutte le pagine devono stare nella memoria fisica per eseguire il programma. Quando il programma fa riferimento ad una parte del proprio spazio degli indirizzi che è nella memoria fisica, l'hw esegue il necessario mappaggio diretto; quando fa riferimento a una parte che *non* è nella memoria fisica, il SO è allertato di andare a prendere il pezzo mancante e rieseguire l'operazione fallita.

Paginazione

La maggior parte dei sistemi di memoria virtuale usa una tecnica chiamata **paginazione** o **paging**. Su qualsiasi computer i programmi referenziano un insieme di indirizzi di memoria. Questi indirizzi generati dal programma sono chiamati **indirizzi virtuali** e formano lo **spazio virtuale degli indirizzi**. Sui computer senza memoria virtuale, l'indirizzo virtuale è messo direttamente sul bus di memoria e provoca la lettura/scrittura della parola della memoria fisica con lo stesso indirizzo. Quando è usata la memoria virtuale, gli indirizzi virtuali non vanno direttamente al bus di memoria, ma ad una **MMU** (**memory management unit**) che mappa gli indirizzi virtuali sugli indirizzi di memoria fisica. Lo spazio degli indirizzi virtuali è suddiviso in unità di dimensione fissa, chiamate **pagine**. Le unità corrispondenti nella memoria fisica sono chiamate **frame** o **page frame**. Le pagine e i frame sono generalmente della stessa dimensione. Nell'hw effettivo, un **bit presente/assente** tiene traccia di quali pagine sono presenti fisicamente in memoria. Se il programma referenzia degli indirizzi non mappati, allora la MMU rileva che la pagina non è mappata e causa una trap della CPU verso il SO. Questa trap è chiamata **page fault**. Il SO preleva il frame da sostituire e scrive il suo contenuto su disco; poi prende la pagina appena referenziata e la mette nel suddetto frame, cambia la mappa e riavvia l'istruzione che era in trap.
Il numero di pagina è usato come indice nella **page table**: se il bit presente/assente è 0, avviene una trap al sistema operativo per recuperare il frame; se il bit è 1, il numero di frame trovato viene usato per calcolare l'indirizzo fisico.

Page table

In una semplice implementazione si può sintetizzare il mappaggio degli indirizzi virtuali su indirizzi fisici in questo modo: l'indirizzo virtuale è diviso in un numero di pagina virtuale (bit più significativi) e offset (bit meno significativi). Per esempio, un indirizzo di 16 bit potrebbe avere 4 bit come numero di pagina e 12 come offset. Il numero di pagina virtuale è utilizzato come un indice nella page table per trovare il numero di frame (se esiste). Il numero di frame è

allegato all'offset per formare un indirizzo fisico che può essere inviato alla memoria. In questo modo lo scopo della page table è mappare le pagine virtuali sui frame delle pagine.

Struttura di una voce della page table. Il campo più importante è il numero del frame. Segue il bit *presente/assente*. I bit di *protezione* specificano quali tipi di accesso sono consentiti. I bit *dirty* e *referenziato* tengono traccia dell'uso della pagina. L'ultimo bit permette di disabilitare la cache per la pagina.

Velocizzare la paginazione

In ogni sistema di paginazione devono essere affrontate due questioni principali:

1. Il mappaggio dall'indirizzo virtuale all'indirizzo fisico deve essere veloce;
2. Se lo spazio virtuale degli indirizzi è grande, la page table sarà grande.

Il primo punto deriva dal fatto che il mappaggio virtuale-fisico deve avvenire per ogni riferimento alla memoria. Il secondo punto deriva dal fatto che tutti i computer moderni usano indirizzi virtuali di almeno 32 bit, e lo standard sta diventando 64 bit; si raggiungono velocemente page table di milioni di voci (ricordiamo che ogni processo ha la propria page table).

Per quanto riguarda il primo punto, è stato osservato che la maggior parte dei programmi tende a fare un gran numero di riferimenti ad un piccolo numero di pagine e non il contrario; dunque solo una piccola parte delle voci della page table viene letta frequentemente, il resto è poco utilizzato. La soluzione escogitata per ovviare al primo problema è quindi stata quella di equipaggiare i computer di un piccolo dispositivo hw (dentro la MMU) per mappare gli indirizzi virtuali sugli indirizzi fisici senza passare per la page table, chiamato **TLB, translation lookaside buffer**. In sostanza si tratta di una cache per la page table, che contiene informazioni necessarie per mappare un numero di pagina virtuale ad un numero di frame. Quando un numero di pagina virtuale non è nel TLB, la MMU rileva un **TLB miss** e fa una normale ricerca sulla tabella delle pagine, quindi sostituisce una delle voci presenti con quella appena cercata.
In realtà, non è necessario avere un dispositivo hardware che svolga la funzione di TLB, dato che questa può essere gestita a livello software dal SO. Il SO carica esplicitamente le voci della TLB; quando c'è un miss, viene generato un errore e il SO si arrangia a trovare la pagina ricercata e inserirla nella TLB. Vi sono due tipi di miss: **soft miss** quando la pagina di riferimento non è nel TLB ma è in memoria, basta aggiornare il TLB (10-20 istruzioni macchina, pochi ns) e non serve I/O; **hard miss** quando la pagina non è in memoria, quindi è necessaria un'operazione di I/O dal disco che dura in media parecchi ms. Cercare il mappaggio nella gerarchia delle page table è un'operazione che prende il nome di **table walk**.

Il secondo problema riguarda il comportamento da assumere con spazi degli indirizzi virtuali molto grandi.

Tabelle delle pagine multilivello. È l'equivalente del "paginare" la page table. Sono presenti più livelli di page table; la voce localizzata tramite l'indicizzazione nella page table di primo livello produce l'indirizzo o il numero di frame di una page table di secondo livello. Nell'indirizzo virtuale ci saranno due campi, uno per la prima page table e uno per la seconda.

Tabelle delle pagine invertite. In questa progettazione c'è una sola voce per frame nella memoria reale, piuttosto che una voce per pagina dello spazio virtuale degli indirizzi. Sebbene risparmino una gran quantità di spazio, quando lo spazio virtuale degli indirizzi è molto superiore rispetto alla memoria fisica la traduzione diventa difficile. Per ovviare a questo problema si usa una TLB. Queste tabelle sono comuni nelle macchine a 64 bit.

Algoritmi di sostituzione delle pagine

Quando si verifica un page fault, per far spazio alla pagina entrante il SO deve scegliere una pagina da rimuovere dalla memoria. Se la pagina da rimuovere è stata modificata mentre era in memoria deve essere riscritta su disco per mantenere la copia aggiornata; se non è stata modificata questa operazione non è necessaria. In tutti gli algoritmi di sostituzione nasce una certa questione: quando una pagina deve essere rimossa dalla memoria, deve essere una delle pagine dello stesso processo (quello che ha generato il page fault) o può appartenere ad un altro processo? Nel primo caso si limita ciascun processo ad un numero fisso di pagine; nel secondo caso no. Entrambe sono possibilità valide.

L'algoritmo ottimale di sostituzione delle pagine. Il miglior algoritmo è semplice da descrivere ma impossibile da implementare. Al momento del page fault, nella memoria si trova un insieme di pagine. Una di queste sta per essere referenziata in un'istruzione molto vicina; le altre non saranno referenziate prima di un certo numero di istruzioni. Si etichetta quindi ciascuna pagina con il numero di istruzioni da eseguire prima che quella pagina sia referenziata per la prima volta. L'algoritmo quindi sceglie di sostituire la pagina con il più alto numero di etichetta. Nella realtà, ciò è irrealizzabile poichè, al momento del page fault, il SO non ha modo di sapere quando ciascuna delle pagine sarà referenziata la volta successiva.

Not recently used (NRU). Molti computer con memoria virtuale dispongono di due bit di stato: R, che viene impostato quando si fa riferimento alla pagina (lettura o scrittura) e M, che viene impostato quando la pagina viene modificata. L'algoritmo di paginazione NRU funziona così: all'inizio, entrambi i bit sono impostati a 0 dal SO. Periodicamente (per esempio ogni interrupt del clock), il bit R è ripulito. Quando avviene un page fault, il sistema operativo ispeziona tutte le pagine e le divide in 4 categorie in base ai valori di R e M:

- Classe 0: non referenziato, non modificato;

- Classe 1: non referenziato, modificato;
- Classe 2: referenziato, non modificato;
- Classe 4: referenziato, modificato.

Si rimuove quindi una pagina a caso dalla classe non vuota con il numero più basso. NRU è facilmente comprensibile, discretamente efficiente da implementare e fornisce prestazioni che, anche se non ottimali, possono risultare adeguate.

FIFO. Algoritmo di paginazione con basso overhead. Il SO tiene una lista delle pagine attualmente in memoria, con l'arrivo più recente in coda e il più vecchio in testa. A una page fault la pagina di testa è rimossa e la nuova aggiunta in coda. Raramente utilizzato in questa forma, perché potrebbe rimuovere pagine presenti in memoria da molto tempo ma ancora referenziate.

Seconda chance. Modifica di FIFO, consiste nel controllare il bit R della pagina più vecchia: se è 0, si rimuove la pagina normalmente; se è 1, il bit è azzerato e la pagina è messa in fondo all'elenco come se fosse appena arrivata in memoria, quindi si continua la ricerca.

Clock. Sebbene l'algoritmo seconda chance sia ragionevole, è inutilmente inefficiente poiché fa scorrere di continuo le pagine lungo la sua lista. Un approccio migliore è tenere tutti i frame su una lista circolare a forma di orologio, in cui una lancetta indica la pagina più vecchia. Quando avviene il page fault, se il bit R della pagina indicata dalla lancetta è 0 allora si sostituisce quella pagina e si sposta avanti la lancetta, altrimenti si azzera R e si ripete il procedimento spostando avanti la lancetta.

Least recently used (LRU). L'idea di LRU è quella di sostituire la pagina che è stata referenziata più tempo fa. Sebbene teoricamente possibile, non è economico e sarebbe necessario un hardware particolare per implementarlo. Si preferisce implementarlo via software, usando il NFU.

Not frequently used (NFU). Richiede un contatore software associato ad ogni pagina e inizialmente impostato a zero. Il SO fa la scansione di tutte le pagine in memoria, ad ogni interrupt del clock. Per ciascuna pagina viene sommato il bit di R al contatore; il contatore quindi tiene traccia all'incirca di quante volte la pagina è stata referenziata. Quando avviene un page fault la scelta della pagina da sostituire cade su quella con il contatore più basso. Il problema di NFU è che non dimentica nulla: ad esempio, con un compilatore multipass, se il primo passaggio è quello con il tempo di esecuzione maggiore tra tutti i passaggi, le pagine dei passaggi successivi potrebbero avere sempre conteggi inferiori alle pagine del primo passaggio; di conseguenza il SO rimuoverebbe pagine utili invece di pagine non più utilizzate.

Aging. Una modifica semplice di NFU lo rende in grado di simulare abbastanza bene LRU. La modifica consta di due parti: innanzitutto i contatori sono shiftati a destra di 1 bit prima che

sia aggiunto il bit R; quindi si aggiunge il bit R al bit più a sinistra. Questa modifica prende il nome di aging.

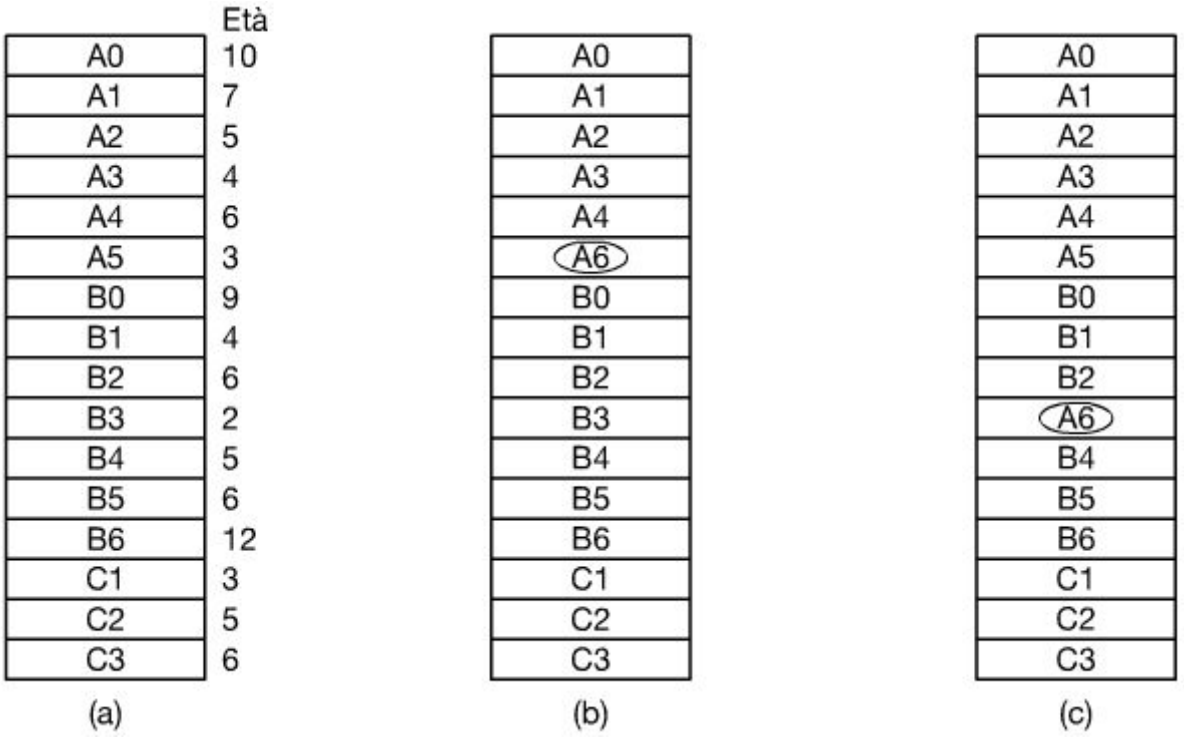
Working set. I processi, nella forma più pura di paginazione, sono avviati senza nessuna delle loro pagine in memoria. Appena la CPU prova a prelevare la prima istruzione, genera un page fault, seguito da altri page fault per le variabili globali e lo stack. Dopo un certo tempo, il processo ha la maggior parte delle pagine di cui ha bisogno. Questa strategia è detta **demand paging** poiché le pagine sono caricate a richiesta (on demand) e non in anticipo. L'insieme delle pagine che un processo sta attualmente usando è noto come il suo **working set**. Se l'intero working set è in memoria, il processo sarà eseguito senza causare molti page fault; se la memoria disponibile è troppo piccola per contenere l'intero working set, il processo provocherà molti page fault e sarà lento nell'esecuzione. Un programma che causa page fault frequenti è detto essere **thrashing**.
Tecnicamente, è lento avere 20, 100 o 1000 page fault ogni volta che un processo è caricato; lo spreco di tempo è considerevole. Molti sistemi di paginazione cercano quindi di tenere traccia del working set di ciascun processo e di essere certi che sia in memoria prima di consentirne l'esecuzione: questo approccio è chiamato **working set model**. Caricare le pagine prima di eseguire i processi è detto anche **prepaging**.
Per implementare il modello a working set serve che il SO tenga traccia di quali pagine sono nel working set. Avere questa informazione porta immediatamente ad un possibile algoritmo di sostituzione delle pagine: quando accade un page fault, cerca una pagina al di fuori del working set e la rimuove.

WSClock. Non ho voglia di scriverlo.

FINE ARGOMENTI PRIMO COMPITINO

Problemi di progettazione dei sistemi di paginazione

Abbiamo discusso parecchi algoritmi per la scelta della pagina da sostituire quando accade un page fault, ma un'importante questione è rimasta scoperta: come dovrebbe essere allocata la memoria fra i processi eseguibili concorrentemente?
Supponiamo di aver la seguente situazione, dove A, B e C sono l'insieme dei processi eseguibili:



Situazione iniziale (a): il processo A genera un page fault, quindi è necessario far spazio in memoria per la nuova pagina A6, rimuovendo la pagina che è stata usata meno recentemente. A questo punto si prospettano due possibilità:

- *Situazione (b)*: si considerano solo le pagine del processo che ha generato il page fault; in questo caso quindi si rimuove A5. Questo algoritmo è detto **algoritmo di sostituzione delle pagine locali**;
- *Situazione (c)*: si considerano tutte le pagine in memoria, indipendentemente dal processo a cui appartengono; in questo caso quindi si rimuove B3. Questo algoritmo è detto **algoritmo di sostituzione delle pagine globali**.

Gli algoritmi locali assegnano ad ogni processo una frazione fissa della memoria. Gli algoritmi globali assegnano dinamicamente frame fra i processi eseguibili; in questo modo il numero di frame assegnato ad ogni processo varia nel tempo. Gli algoritmi globali funzionano generalmente meglio, specie quando la dimensione del working set può variare durante la vita di un processo. Se è usato un algoritmo locale e il working set aumenta, ne risulta del thrashing, anche in presenza di frame liberi; se diminuisce, risulta esserci uno spreco di memoria. Se viene usato un algoritmo globale, il sistema deve continuamente decidere quanti frame assegnare a ciascun processo tramite un algoritmo. Un modo è determinare periodicamente il numero di processi in esecuzione e assegnare a ciascun processo una parte

uguale. Sebbene questo metodo sembri equo, ha poco senso assegnare parti uguali di memoria ad un processo di 10 KB e ad uno di 10 MB, ad esempio; le pagine possono invece essere assegnate in proporzione alla dimensione totale del processo (è saggio dare ad ogni processo un limite minimo di pagine). Con gli algoritmi globali è possibile avviare ogni processo con un certo numero di pagine proporzionale alla sua dimensione, ma quando il processo è eseguito, l'assegnazione deve essere aggiornata dinamicamente. Un modo per gestire l'assegnazione è di usare l'algoritmo **PFF** (**Page Fault Frequency**), che indica quando aumentare o diminuire l'allocazione delle pagine di un processo, ma non dice nulla riguardo quale pagina sostituire in caso di page fault; esso controlla solo la dimensione dell'insieme di allocazione.
È importante notare che alcuni algoritmi di sostituzione delle pagine possono funzionare con una politica di sostituzione sia locale sia globale: ad esempio il FIFO può sostituire le pagine vecchie in tutta la memoria (globale) o le vecchie pagine di proprietà del processo attuale (locale).

Dimensione delle pagine

La dimensione delle pagine è spesso un parametro che può essere scelto dal sistema operativo. Determinare la migliore dimensione delle pagine richiede il bilanciamento di parecchi fattori in concorrenza fra loro. Non esiste quindi un risultato che vada bene in qualsiasi situazione. Innanzitutto ci sono due fattori a favore delle pagine piccole:

- Mediamente metà della pagina finale è vuota; lo spazio in più è sprecato. Questo spreco è detto **frammentazione interna**. Con n segmenti in memoria e una dimensione di pagina di p byte, $np/2$ byte sono sprecati in frammentazione interna.
- Pensiamo ad un programma che consiste di otto fasi sequenziali di 4 KB ciascuna. Con una dimensione di pagina di 32 KB, il programma deve allocare ogni volta 32 KB; con una dimensione di pagina di 16 KB, ha bisogno di soli 16 KB. Con una dimensione minore o uguale a 4 KB, saranno richiesti solo 4 KB in ogni momento. In generale una dimensione di pagina grande fa sì che vi sia nella memoria più spazio inutilizzato rispetto a pagine di piccole dimensioni.

D'altra parte, le pagine piccole comportano che il programma richieda molte pagine, ossia una page table estesa. I trasferimenti da e verso il disco sono generalmente di una pagina per volta, con la maggior parte del tempo impiegato per ricerca e il ritardo della rotazione; trasferire pagine piccole dunque richiede quasi tanto tempo quanto trasferire una pagina grande.
In alcune macchine, è necessario che il sistema operativo carichi nei registri hardware la tabella delle pagine ogniqualvolta la CPU passa da un processo ad un altro. Su queste macchine, una dimensione delle pagine ridotta significa che il tempo richiesto a caricare i registri delle pagine aumenta con il ridursi della dimensione delle pagine. Lo spazio occupato

dalla page table, inoltre, aumenta al diminuire della dimensione delle pagine. Quest'ultimo può essere analizzato da un punto di vista matematico: supponiamo sia *s* byte la dimensione media del processo e *p* byte la dimensione della pagina; ogni voce della tabella richiede *e* byte. Il numero di pagine necessario per processo è circa *s/p*, che occupa *se/p* byte di spazio di page table. La memoria sprecata a causa di frammentazione interna nell'ultima pagina del processo è *p/2*. L'overhead totale risulta quindi: $overhead = se/p + p/2$. I computer disponibili in commercio hanno usato dimensioni di pagina che vanno dai 512 byte ai 64 KB. I valori più comuni oggi sono 4 KB e 8 KB.

La paginazione lavora al meglio quando c'è abbondanza di frame liberi che possono essere richiamati se si verificano dei page fault; se tutti i frame sono pieni e anche modificati, prima che possa essere introdotta una nuova pagina occorre che una pagina vecchia sia scritta su disco. Per assicurare un rifornimento abbondante di frame liberi, molti sistemi di paginazione dispongono di un processo in background chiamato **paging daemon**, che resta dormiente per la maggior parte del tempo ma periodicamente si risveglia per verificare lo stato della memoria. Se la quantità di frame liberi è troppo bassa, il paging daemon comincia a selezionare pagine da eliminare usando un qualche algoritmo di sostituzione delle pagine; se queste sono state modificate, sono anche scritte su disco.

Segmentazione

La memoria virtuale illustrata finora è monodimensionale poichè gli indirizzi virtuali vanno da 0 ad un certo massimo, uno dopo l'altro. Per molti problemi, avere due o più spazi degli indirizzi virtuali può essere molto meglio che averne uno. Una soluzione diretta molto generica è fornire la macchina di molti spazi degli indirizzi completamente indipendenti, chiamati **segmenti**. Ogni segmento consiste di una sequenza lineare di indirizzi, da 0 ad un certo massimo; segmenti diversi possono avere lunghezze diverse ed inoltre la lunghezza può variare durante l'esecuzione. La lunghezza di un segmento di stack può essere aumentata ogni volta che viene inserito qualcosa nello stack e diminuita ogni volta che qualcosa ne è estratto. Poiché ogni segmento costituisce uno spazio degli indirizzi separato, segmenti diversi possono crescere o compattarsi indipendentemente, senza influenzarsi l'un l'altro. Per specificare un indirizzo in questa memoria segmentata o bidimensionale, il programma deve fornire un indirizzo costituito da due parti: il numero di segmento e un indirizzo nel segmento. Un segmento può contenere una procedura o un array o uno stack o una raccolta di variabili scalari, ma solitamente non contiene mix di tipologie diverse.
Oltre a semplificare il trattamento delle strutture di dati che crescono o si compattano, una memoria segmentata ha altri vantaggi. Ad esempio, se ogni procedura occupa un segmento separato, con indirizzo 0 al suo punto di partenza, il linking di procedure compilate separatamente è estremamente semplificato.

CAPITOLO 4: FILE SYSTEM

Si possono identificare tre requisiti essenziali per il salvataggio delle informazioni a lungo termine:

1. Capacità di poter salvare una **grande quantità** di informazioni: per molte applicazioni (in genere quelle che richiedono la gestione di database/grande quantità di dati) è necessaria una grande dimensione dello spazio degli indirizzi per poter lavorare adeguatamente.
2. Le informazioni devono **permanere** una volta la terminazione del processo: es. gestione di database, dove le info devono essere mantenute per lunghi periodi di tempo e persistere anche in caso di un'inaspettata terminazione del processo.
3. Le informazioni devono poter essere **accedute contemporaneamente** da più processi. Un modo per soddisfare questo requisito è rendere tali informazioni **indipendenti** da qualunque processo.

È possibile, come fatto in precedenza con il processore e la gestione della memoria, **astrarre** il concetto di informazione con il concetto di **file**. Tale astrazione insieme a quella di processo e di spazio di indirizzi sono i fondamenti della comprensione dei SO.

I file

*I **file** sono **unità logiche di informazioni** create da processi. Possono essere pensati come uno spazio degli indirizzi usati per modellare il disco. I file sono **persistenti**, ovvero indipendenti dalla creazione e terminazione di un dato processo.*

La parte del SO che a che fare (accesso, utilizzo, protezione, gestione, ecc.) con i file viene chiamata **file system**. Per un utente non è ovviamente importante l'implementazione dei file, in quanto esso vede esclusivamente come appare, da cosa è costituito, quali operazioni sono possibili ecc. Le successive considerazioni sui file verranno fatte sotto il punto di vista utente.

Nomi dei file.
Per gestire al meglio ogni tipo di astrazione sono importanti i **criteri e l'utilizzo della nomenclatura** degli oggetti gestiti. Ogni volta che un processo crea un file gli assegna un nome. Tale nome persiste alla terminazione del processo e può essere utilizzato da altri processi per accedere a tale file.
I criteri per la definizione dei nomi definiscono la lunghezza massima della stringa, l'utilizzo di caratteri speciali o meno, la distinzione tra maiuscole e minuscole, la composizione del nome

in più parti (*nome*.estensione). Tali criteri differiscono in base al file system considerato (i più vecchi MS-DOS, FAT e FAT-16 non distinguevano le maiuscole dalle minuscole).

Struttura dei file.

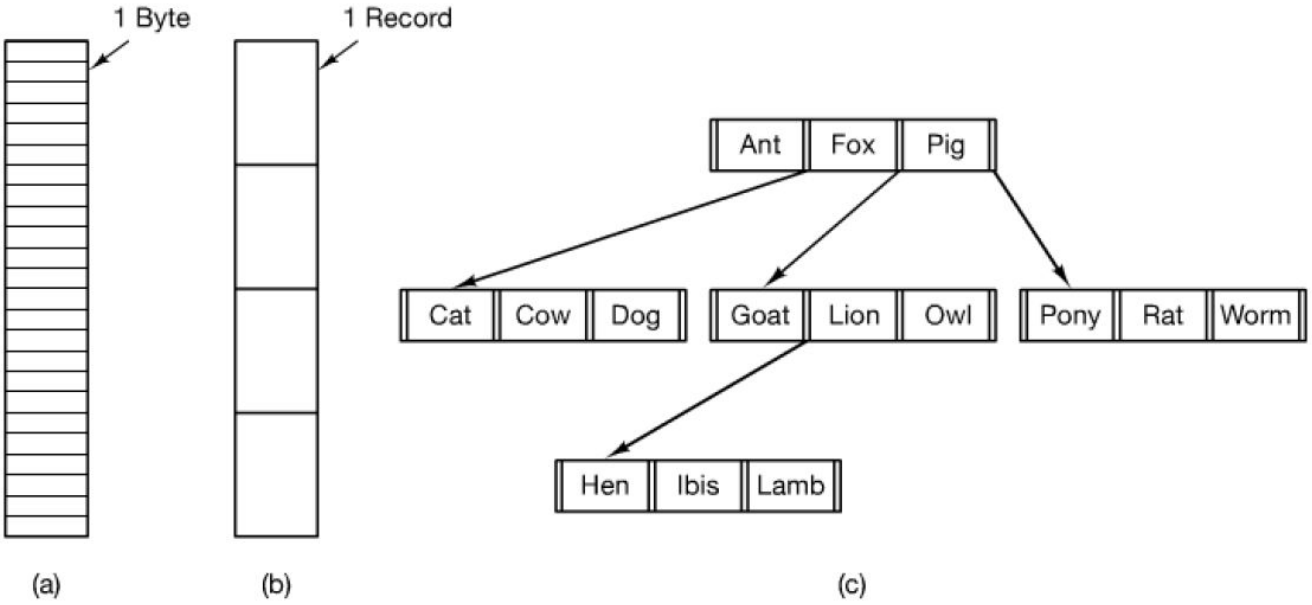


Figura 4.2 Tre tipi di file. (a) Sequenza di byte. (b) Sequenza di record. (c) Albero.

- **4.2(a) SEQUENZA NON STRUTTURATA DI BYTE**: il SO non conosce, e non è interessato al contenuto del file, in quanto vede solo i byte da cui è formato. Il significato viene dato dai programmi a livello utente. Approccio molto flessibile, adottato da tutte le versioni di UNIX, MS-DOS e WINDOWS;
- **4.2(b) SEQUENZA DI RECORD A LUNGHEZZA FISSA**: ogni record ha la propria struttura interna. Un'operazione di lettura su tale struttura restituisce un record, una di scrittura aggiunge o aggiorna record. Struttura inutilizzata oggigiorno;
- **4.2(c) ALBERO DI RECORD**: i record hanno lunghezza variabile, ed ognuno contiene un campo chiave. L'albero è ordinato su tale campo permettendo una ricerca rapida per particolari chiavi. L'aggiunta di nuovi record è gestita dal SO e non da un programma utente. Struttura usata principalmente su grossi mainframe.

Tipi di file.

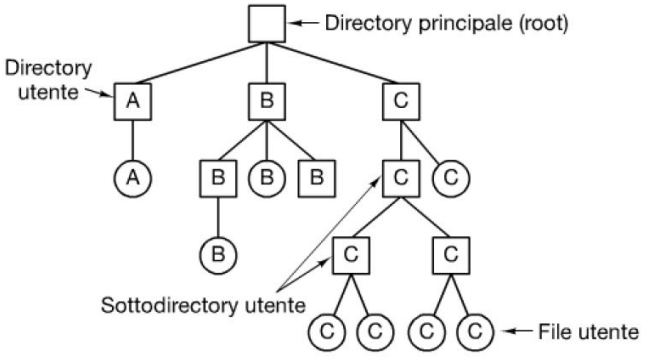
- **File normali:** contengono informazioni utente. Possono essere file ASCII o file binari. Quelli ASCII sono molto semplici da visualizzare e modificare, in quanto composti da stringhe. E' inoltre molto semplice collegare l'output di un programma come input di un altro quando tali file vengono usati come input/output. I file binari sono ovviamente interpretabili dai programmi che ne conoscono la struttura.
- **Directory:** file di sistema usati per mantenere la struttura del file system

- **File speciali a caratteri:** usati per la gestione di dispositivi I/O
- **File speciali a blocchi**: usati per modellare i dischi

Directory

Come si possono strutturare le directory:

- **Sistemi di directory a livello singolo**: è la forma più semplice, c'è **una sola directory** che contiene tutti i file e viene detta principale o root. La **semplicità e** la **facilità di ricerca** (c'è un solo posto dove cercare) di questa implementazione fanno sì che sia prevalentemente usata nei dispositivi embedded
- **Sistemi di directory gerarchici**: un singolo livello non è però ideale per gestire più utenti e migliaia di file; la soluzione è una **gerarchia**. Le directory A,B e C appartengono ciascuna a un utente differente, e ognuno di essi può creare un numero arbitrario di sottodirectory.



Nome di percorso.

Ci sono due metodi per specificare univocamente un file:

- **Nome di percorso assoluto**: composto dalla directory principale al file, ad esempio /usr/ast/mailbox (N.B. il separatore varia in base al SO considerato)
- **Nome di percorso relativo**: viene designata dall'utente una **directory di lavoro (detta anche corrente)** e tutti i percorsi che non iniziano dalla root sono considerati relativi a questa directory di lavoro. Ad esempio: dir di lavoro: */usr/ast***,** allora *mailbox* equivale a */usr/ast/mailbox*

Preferenze nell'utilizzo del tipo di percorso.

Utilizzando il percorso assoluto si ha la certezza che funzionerà sempre.

Tuttavia si usa anche cambiare la directory di lavoro con una chiamata a sistema, in modo da utilizzare un parametro più corto per la open (apre/legge una directory, vedi paragrafo successivo). Questa tecnica è vantaggiosa con i processi: ogni processo ha la propria directory di lavoro, e può cambiarla a piacimento. Si tratta di un'operazione sicura perché

Operazioni sulle directory.
Le chiamate di sistema che permettono di gestire le directory sono molto diverse in base al SO considerato. Consideriamo quelle di UNIX.

1.**Create**. Crea un directory vuota
2.**Delete**. Cancella una directory sse la directory è vuota
3.**Opendir**. Analogamente ad un file, una directory deve essere aperta prima di poter essere letta
4.**Closedir** Chiude una directory aperta (viene liberato lo spazio delle tabelle interne)
5.**Readdir**. restituisce la voce successiva all'interno di una directory aperta. Superiore alla chiamata di sistema read in quanto non necessita che l'utilizzatore conosca la struttura interna della directory
6.**Rename**. Rinomina una directory
7.**Link**. Permette ad un file di apparire in più directory. Questo collegamento è detto **hard link**, poiché incrementa un contatore (chiamato i-node, che in realtà è un array con un contatore per ogni file) che conta in quante directory contengono quel file
8.**Unlink**.Viene rimossa una voce della directory. Se il file scollegato era presente in una sola directory, viene rimosso dal file system, altrimenti solo il percorso specificato.

Un altro tipo di link è il **link simbolico**, che permette di attraversare dischi diversi e/o collegare computer remoti. Invece di avere due nome che puntano alla stessa struttura rappresentante un file,viene creato un piccolo file, di tipo LINK, che contiene solo il nome di percorso del file a cui si collega.

Implementazione del file system

I file system sono memorizzati sul disco. Il disco è suddivisibile in più partizioni, ognuna con file system indipendenti. Il **MasterBootRecord** (**MRB**) o settore 0 è utilizzato per avviare il computer e contiene la tabella delle partizioni, che indica gli indirizzi di inizio e fine di ciascuna partizione.
All'avvio: **BIOS** avvia **l'MBR** -> **l'MBR** esegue il **boot block** -> il boot block carica il **SO**

Layout di una partizione di disco (può cambiare molto in base al file system considerato)

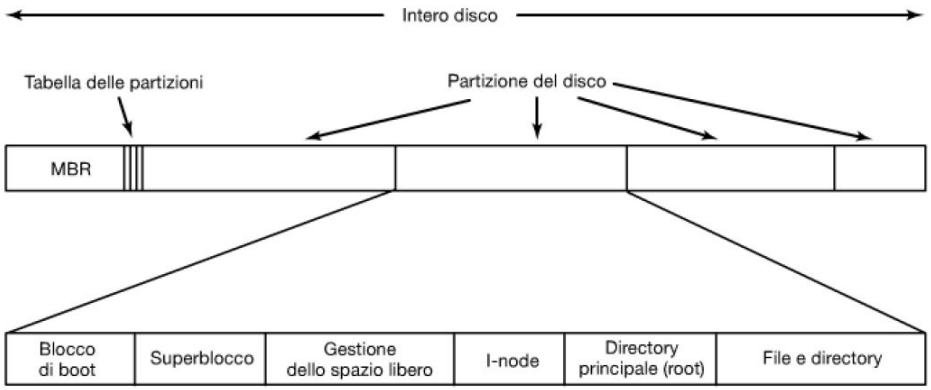


Figura 4.9 Possibile layout di un file system.

- **Superblocco**: contiene tutti i parametri chiave riguardanti il file system (numero magico che identifica il tipo di file system, numero di blocchi ecc.)
- **I-node**: array di strutture dati (che contengono info su un file), una per file
- **Directory e file**: tutto il resto

Implementazione dei file.
È importante tener traccia di quali blocchi del disco siano associati ad un determinato file.

Allocazione contigua

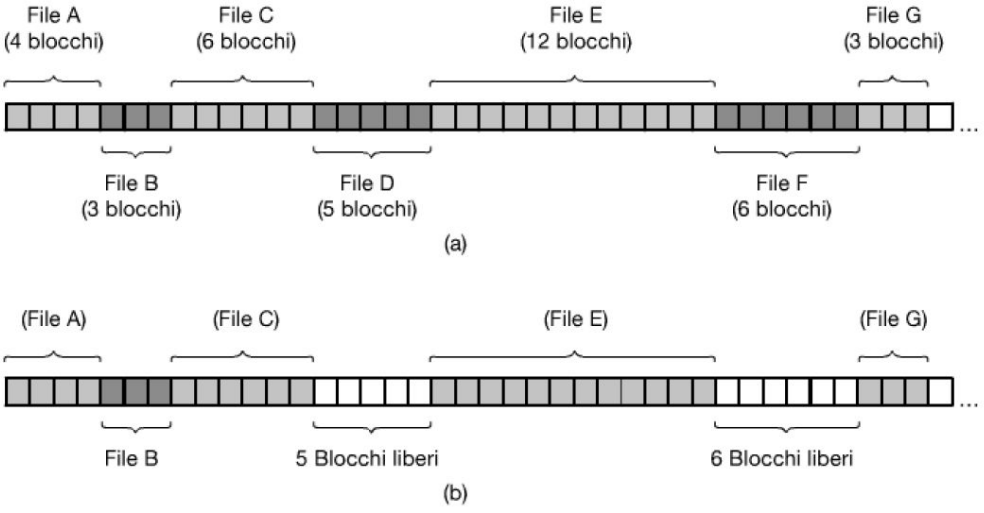


Figura 4.10 (a) Allocazione contigua dello spazio del disco per sette file. (b) Lo stato del disco dopo la rimozione dei file *D* e *F*.

La più semplice, si memorizza ciascun file come una sequenza contigua di blocchi del disco.
Vantaggi:

- **Semplice** da implementare: basta l'indirizzo del disco del primo blocco e il numero dei blocchi del file per tenerne traccia
- L'intero file può essere **letto da disco in una singola operazione**

Svantaggi:

- Nel corso degli anni **i dischi si frammentano** (vedi 4.10(b)). Inizialmente si può sempre scrivere alla fine del disco, diventa problematico una volta che questo si riempie

Questa implementazione si presta molto bene per i CD-ROM in quanto le dimensioni dei file sono conosciute in anticipo e non cambieranno nel tempo.

Allocazione a liste collegate

Si mantiene ciascun file come una lista collegata di blocchi del disco. La prima parola di ciascun blocco è usata come puntatore al successivo. Il resto del blocco sono i dati.

Vantaggi:

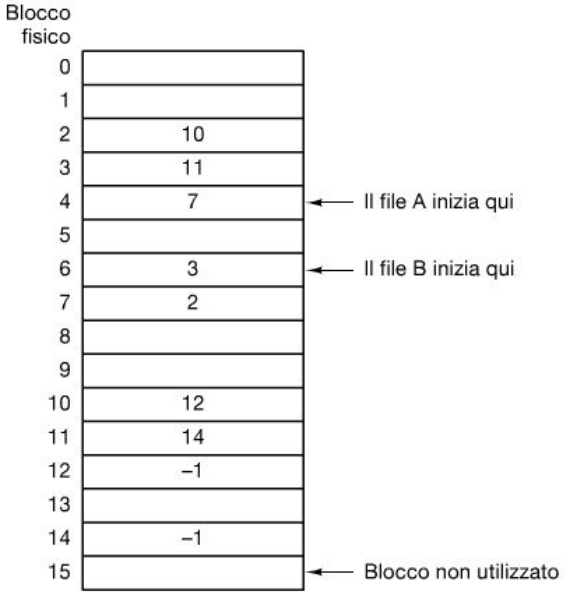
- Non soffre di frammentazione del disco

Svantaggi:

- L'accesso è estremamente **lento**
- La **quantità di spazio** necessaria è superiore in quanto i puntatori occupano alcuni byte

Allocazione a liste collegate usando una tabella in memoria

Ogni parola relativa al puntatore di ogni blocco del disco viene messa in una **tabella in memoria**. Questo tipo di tabella è chiamata **FAT** (file allocation table).



Ad esempio nell'immagine seguente il file a usa i blocchi 4,7,2,10,12.

Vantaggi:

- Non soffre dei difetti dell' allocazione con liste collegate

Svantaggi:

- La tabella deve permanere in memoria, e la dimensione occupata è direttamente proporzionale alla dimensione del disco. Es.: disco da un TB e dim blocco da 1KB → un miliardo di voci, ciascuna dai 3 ai 4 Byte → tabella occupa dai 2,4 ai 3 GB

diegochine

I-node

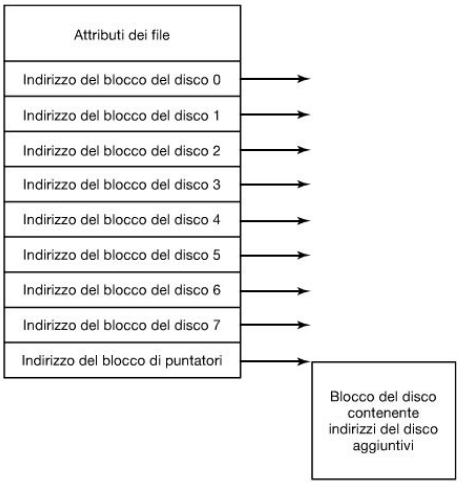
Un **array chiamato i-node** elenca gli attributi e gli indirizzi dei blocchi del file

Vantaggi:

- L'I-node ha bisogno di essere in memoria solo quando il file corrispondente è aperto
- Lo spazio occupato dall'I-node è direttamente proporzionale al numero massimo di file che potrebbe essere contemporaneamente aperto (ciascun I-node occupa n byte e si possono avere al massimo k file aperti, basta riservare kn byte)

Svantaggi:

- Ogni I-node ha spazio per un numero fisso di indirizzi del disco. Per ovviare a problemi legati al superamento di tale limite, l'ultimo indirizzo del disco è riservato ad un indirizzo che punta ad ulteriori blocchi del disco



Implementazione delle directory

Apertura file → SO usa il pathname per localizzare la voce della directory → la directory fornisce le info per trovare i blocchi del disco, in base all'implementazione usata (alloc contigua, liste collegate, i-node).

Gli attributi possono essere memorizzati direttamente nella voce della directory *(a)*, o negli i-node *(b)*.

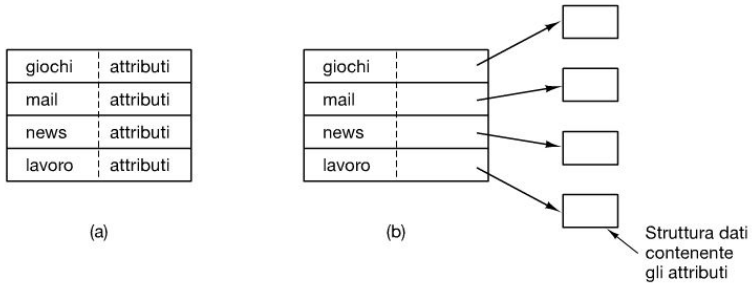


Figura 4.14 (a) Una semplice directory contenente voci a dimensione fissa con gli indirizzi del disco e gli attributi nella voce della directory. (b) Una directory in cui ogni voce fa semplicemente riferimento a un i-node.

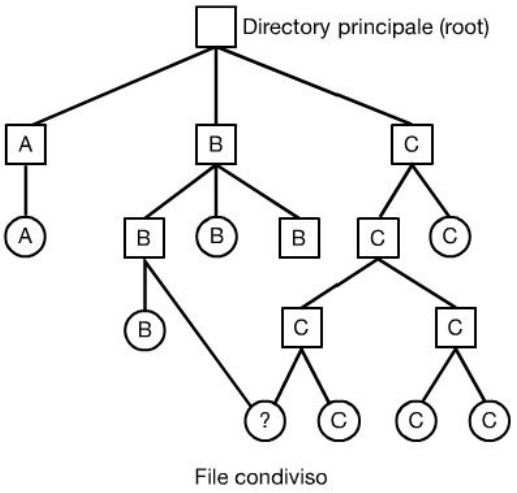
---

diegochine

File condivisi

Un file è **condiviso** quando appare in directory diverse **contemporaneamente**, appartenenti ad **utenti diversi**.

Il seguente è detto grafo aciclico orientato o **DAG**: **-->DArio Gay**



File condiviso

Vedi la fine delle operazioni su directory per informazioni sul linking (hard e simbolico)

File system strutturati a log

Con l'incremento della velocità delle CPU e della dimensione delle memorie la **cache** del file system può soddisfare buona parte delle richieste **senza bisogno di accedere al disco.**

L'idea dei **Log-Structured-file System (LFS)** è quindi quello di strutturare l'intero disco come un file di log. Tutte le **operazioni di scrittura sono bufferizzate** in memorie e scritte sul disco come un **singolo segmento contiguo** alla fine del log.

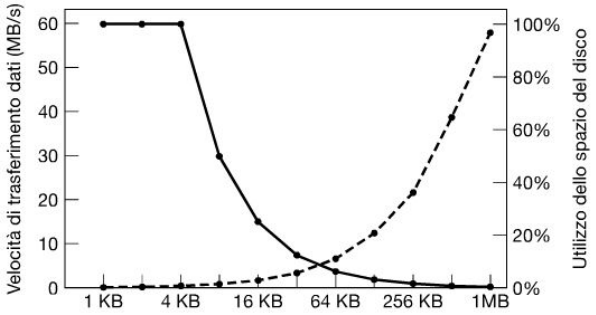
Dato che i dischi non hanno memoria infinita è necessario un thread chiamato cleaner, che scansiona circolarmente il log per compattarlo.

Gestione dello spazio su disco

La scelta della dimensione dei blocchi è importante: una dimensione troppo grande vuol dire che file di piccole dimensioni sprecano una grande quantità di spazio.

Una dimensione di blocco piccola tuttavia comporta la distribuzione della maggior parte dei file su molteplici blocchi, che comporta lentezza e ritardi nella lettura.

Il **rapporto tra efficienza e dimensione** dei blocchi è illustrata nel grafico sottostante:



La curva continua (lato sinistro della scala) fornisce la velocità di trasferimento dati di un disco. La curva tratteggiata (a destra) esprime l'efficienza nell'utilizzo dello spazio del disco. Tutti i file sono di 4 KB.

Tenere traccia dei blocchi liberi

Esistono principalmente due metodi:

- **Lista** (chiamata **free list**) collegate di blocchi del disco: ogni blocco contiene tanti numeri di blocchi di disco liberi quando possibile.
  Es. con un blocco da 1 KB e un numero di blocchi di disco a 32 bit ciascun blocco sulla free list contiene i numeri di 255 blocchi liberi
- **Bitmap:** un disco con n blocchi richiede una bitmap con n bit. I blocchi liberi sono indicati con il valore 1; con 0 altrimenti. La bitmap occupa meno spazio rispetto alla lista, in quanto utilizza un solo bit per blocco, invece che 32. Ciò comporta anche una maggior velocità nel controllare se ci sono blocchi liberi contigui. Il principale svantaggio è che in tale ricerca il file system potrebbe dover cercare per tutta la bitmap.

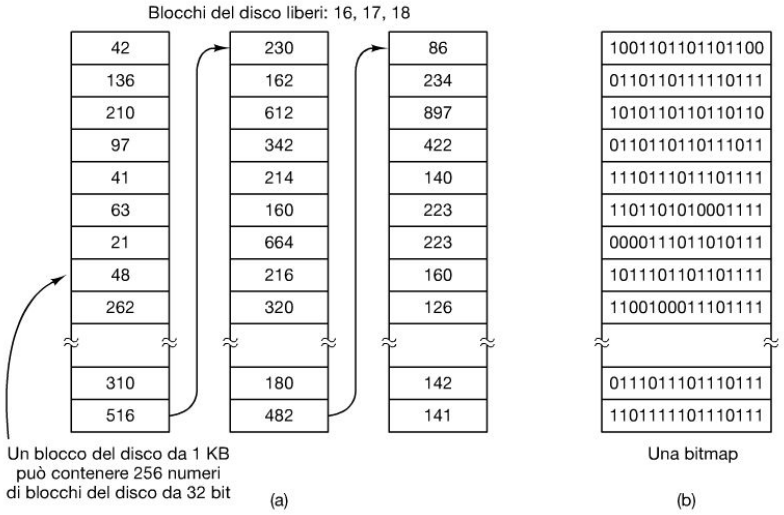


Figura 4.22 (a) Memorizzazione della free list in una lista collegata. (b) Una bitmap.

Backup

Un backup permette:

- Il **recupero di dati** persi a causa di una **catastrofe naturale** (crash del disco, incendio ecc.)
- Il **recupero di dati** persi a causa di un **errore di distrazione** (rimozione accidentale)

Alcune considerazioni sui backup

- Conviene non eseguire il backup di programmi che possono essere facilmente re installati, e **limitarlo solo a directory specifiche**.
- E' uno spreco di tempo e spazio fare backup di file che non sono cambiati dall'ultimo backup → **backup incrementale**: periodicamente un backup completo (mensilmente/settimanalmente), e un backup giornaliero ma solo dei file modificati.
- E' conveniente **comprimere** i dati di backup prima di memorizzarli
- E' difficile eseguire il backup di un filesystem attivo, e spegnere il sistema per lunghi periodi di tempo non è sempre possibile → viene fatta un "istantanea"(**snapshot**) del file system in quel momento, sulla quale viene eseguito il backup.
- Ovvie **precauzioni dovute alla sicurezza** aka non tenere il backup nello stesso luogo di dove si trovano i pc backuppati colione!

Tipi di backup

Backup fisico

Comincia dal blocco 0 del disco, **copia in ordine** tutti i blocchi del disco e termina quando copia l'ultimo. Il più semplice da realizzare ed **esente da errori**.
Tuttavia non è possibile saltare determinate directory (fare il backup di blocchi di disco inutilizzati, vuoti o danneggiati non ha alcun valore), rendendo impossibile l'esecuzione di backup incrementali e/o il recupero di file individuali su richiesta.

Backup logico

Parte da una o più directory specifiche e fa il **backup ricorsivo** di tutti i file e directory trovati e modificati a partire da una determinata data.
Risulta più semplice il recupero di un file o una directory specifica.

Esempio di backup logico (UNIX):

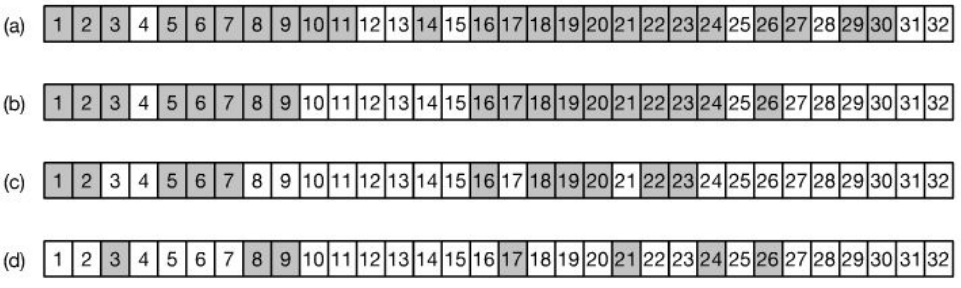


Figura 4.26 Le bitmap usate dall'algoritmo di backup logico.

(a) parte dalla directory principale e esamina tutti i nodi, directory e file

diegochine
indicando quelle modificate (indicati in grigio)
(b) deseleziona le directory che non hanno internamente file modificati
(c) backup delle directory selezionate in ordine di i-node
(d) backup dei file selezionate in ordine di i-node
Mantiene la struttura

Consistenza del file system

La consistenza del file system è un aspetto fondamentale dell'affidabilità di un sistema. Per eseguire una verifica di tale consistenza esistono utility specifiche (fsck per UNIX, scandisk per Windows), eseguite per esempio al riavvio dopo un crash.
Esistono due tipi di controllo della consistenza:

- **Controllo dei blocchi:** vengono create **due tabelle**, ciascuna contenente un **contatore** per ogni blocco (0 di default). La prima tab conta quante volte ogni blocco è presente in un file, la seconda tab registrano quanto spesso ogni blocco sia presente nell free list (o bitmap). Un file system è consistente se ogni blocco ha un 1 **o** nella prima tab **oppure** nella seconda.
- **Controllo delle directory:** Sempre una tabella di contatori, ma uno per ogni file. Si scende ricorsivamente dalla dir principale e di contatori vengono incrementati solo in presenza di un **hard link** per quel file. La tab viene quindi confrontata con i contatori presenti all'interno di ciascun i-node

---

Controllo degli accessi

I file possono contenere dati sensibili e personali come password, numeri di carte di credito ecc. E' importante gestire bene i permessi di accesso a tali file. Distinguiamo principalmente due metodi:

- **Matrice di controllo di accesso**: si utilizza una matrice bidimensionale dove $a_{ij} = 1$ se l'utente *i* può accedere al file *j*, = 0 altrimenti. Gli svantaggi sono ovvi, in quando per tanti utenti e/o tanti file questa matrice risulta essere molto grande e difficile da interrogare
- **Controllo di accesso per classi di utenti:** varie classi che identificano determinati privilegi di accesso per l'utente. Questa tecnica occupa molto meno spazio rispetto alla matrice in quanto i dati per il controllo di accesso possono essere memorizzati come parte del blocco di controllo di file (File Control Block)

Tecniche di accesso ai dati

Consideriamo due metodi per l'accesso ai dati:

- **Metodi di accesso a coda:**
  Dalle migliori prestazioni, utilizzati quando la sequenza di accesso ai record può essere prevista.
- **Metodi di accesso di base:**
  Se la sequenza di accesso non risulta prevedibile (es.: con l'accesso diretto)

Sicurezza (Terminologia)

- **Modello di sicurezza**
  Definisce soggetti, oggetti e privilegi di un sistema (come ad esempio le classi utente). Distinguiamo i controlli di accesso **a discrezione** (DAC → **D**ario **A**ssume **C**ocaina) dove il proprietario del file controlla i permessi, e quelli ad accesso **mandatorio** (MAC → **M**andarino **A**ssassinato dalla **C**ecilia), dove viene predefinito un sistema di autorizzazione centrale.

- **Politica di sicurezza**
  Definisce quali privilegi agli oggetti vanno assegnati ai soggetti → concetto di privilegio minimo: al soggetto viene dato l'accesso esclusivamente agli oggetti di cui ha bisogno per svolgere i suoi compiti

- **Meccanismo di sicurezza**
  E' il metodo con cui il sistema implementa la politica di sicurezza

Esempi di file system

I file system di UNIX e di Windows sono trattati nei capitoli 10 ed 11, qui parla dei loro predecessori.

Riepilogo

Visto dall'esterno un file system è un **insieme di file e directory** con le loro rispettive operazioni. La maggior parte dei filesystem moderni supporta un sistema di directory gerarchico, quindi con directory e sottodirectory annidate.

Visto dall'interno invece si notano le strategie adottate dai progettisti per **allocare lo spazio** e di come **tener traccia di quale blocco associare a ciascun file**. Tali tecniche sono file contigui, liste collegate, tabelle di allocazione dei file, i-node.
Gli **attributi** possono essere assegnati alle directory o ad altri elementi (tipo dentro gli i-node).

Lo **spazio sul disco** può essere gestito mediante liste libere o bitmap.

L'**affidabilità** di un file system può essere migliorata eseguendo backup incrementali ed utilizzando un programma in grado di riparare file system danneggiati.

Per **migliorare le prestazioni** si può ricorrere a diversi metodi: utilizzo di una cache, la lettura anticipata e il posizionamento dei blocchi di un file uno vicino all'altro. Anche il file system basati su log strutturati migliorano le loro prestazioni scrivendo i dati in grosse unità.

Esempi di file system sono ISO 9660, MS-DOS e UNIX che differiscono in molti modi: come viene tenuta traccia di quali blocchi devono essere associati a ciascun file, come vengono strutturate le directory, come viene gestito lo spazio libero su disco ecc.

CAPITOLO 5: INPUT/OUTPUT

Dispositivi di I/O

I dispositivi di I/O possono essere sommariamente suddivisi in due categorie:

- **Dispositivi a blocchi**: è un dispositivo che archivia informazioni in blocchi di dimensioni fisse, ognuno con il proprio indirizzo. Tutti i trasferimenti sono in unità di uno o più blocchi (consecutivi) interi. La caratteristica essenziale di un dispositivo a blocchi è che ciascun blocco può essere letto o scritto indipendentemente da tutti gli altri. Dischi fissi, dischi blu-ray e chiavette USB sono esempi classici.
- **Dispositivi a caratteri**: un'unità a caratteri rilascia o accetta un flusso di caratteri, senza alcuna struttura a blocchi. Non è indirizzabile e non ha alcuna operazione di ricerca. Stampanti, interfacce di rete, mouse e molte altre unità, diverse dai dischi, possono essere viste come dispositivi a caratteri.

Questo schema di classificazione non è perfetto dal momento che alcuni dispositivi rimangono esclusi. I clock, ad esempio, non sono indirizzabili a blocchi e non generano né accettano flussi di caratteri: tutto ciò che fanno è produrre interrupt a intervalli ben definiti. Analogo ragionamento vale per i video a mappatura di memoria e per i touch screen. Questo modello è comunque abbastanza generico da poter essere usato come base per rendere indipendenti dai dispositivi alcuni dei software del SO che si occupano di I/O: il file system ad esempio si occupa solo di dispositivi a blocchi astratti e lascia la parte dipendente dai dispositivi a software di livello più basso.

Controller dei dispositivi

I dispositivi di I/O consistono tipicamente di una componente meccanica e di una elettronica; spesso è possibile separare queste due parti per fornire una progettazione più modulare e generale. La componente elettronica è detta **controller del dispositivo** *(device controller)* o ***adattatore***. Sui pc è spesso presente nella forma di un chip sulla scheda madre o di una scheda a circuiti stampati inseribile in un alloggiamento di espansione (PCI). La parte meccanica è il dispositivo stesso. L'interfaccia fra il controller e il dispositivo è uno stream seriale di bit, che parte con un **preambolo**, seguito dai 4096 bit in un settore e alla fine da un controllo numerico, chiamato **ECC - error correcting code** *(codice di correzione degli errori)*. Il preambolo contiene dati necessari al controller come numero di cilindri e di settori, informazioni di sincronizzazione, ecc. Il lavoro del controller è convertire il flusso seriale di bit in un blocco di byte ed eseguire ogni necessaria correzione di errore. Tipicamente il blocco dei byte viene prima assemblato, bit per bit, in un buffer interno al controller; successivamente si verifica il codice di correzione e una volta che il blocco viene dichiarato libero da errori può essere copiato nella memoria principale.

I/O mappato in memoria

Ogni controller dispone di pochi registri di controllo usati per le comunicazioni con la CPU. Essi possono essere usati dal SO in scrittura, per comandare al dispositivo di eseguire qualche operazione, e in lettura, per capire quale sia lo stato del dispositivo. Oltre ai registri di controllo, molti dispositivi hanno un buffer di dati su cui il SO può leggere o scrivere (*esempio: RAM video*).

La CPU può comunicare con i registri di controllo e i buffer dati del dispositivo in due modi:

1. A ciascun registro di controllo è assegnata un numero di **porta di I/O**; l'insieme di tutte le porte di I/O forma lo **spazio delle porte di I/O**, che è protetto in modo che i normali programmi utente non possano accedervi (lo può fare solo il SO). La CPU può usare istruzioni speciali per leggere e scrivere tali registri.
2. Si mappano tutti i registri di controllo nello spazio della memoria; a ciascun registro di controllo è assegnato un indirizzo di memoria univoco a cui non è assegnata memoria. Questo metodo è detto **memory-mapped I/O**. Generalmente gli indirizzi assegnati sono quelli nella parte superiore dello spazio degli indirizzi.
3. Si può anche usare, in realtà, un approccio ibrido; ad esempio, con buffer dei dati mappati in memoria e porte I/O separate per i registri di controllo.

In tutti i casi, quando la CPU vuole leggere una word, mette l'indirizzo di cui ha bisogno nelle linee degli indirizzi del bus e dichiara un segnale di READ sulla linea di controllo del bus. Una seconda linea di segnale è usata per indicare se si debba impiegare lo spazio degli indirizzi dell'I/O o lo spazio degli indirizzi della memoria.

Vantaggi e svantaggi delle due tecniche:

- Nel memory-mapped I/O i registri di controllo dei dispositivi sono solo variabili in memoria e possono essere utilizzati nei linguaggi di programmazione come ogni altra variabile. Senza i dispositivi mappati sarebbe necessaria una parte in assembly, dato che sono richieste istruzioni speciali.
- Sempre nel memory-mapped I/O non serve alcun meccanismo di protezione speciale per evitare che i processi utente eseguano l'I/O; tutto ciò che il SO deve fare è evitare di mettere quella parte dello spazio degli indirizzi contenente i registri di controllo nello spazio degli indirizzi virtuali di un qualunque utente. Meglio ancora, se ogni dispositivo ha i suoi registri di controllo mappati su una pagina diversa dello spazio degli indirizzi, il SO può dare un controllo utente su certi dispositivi e non su altri. Inoltre ogni istruzione che può referenziare della memoria può anche referenziare dei registri di controllo. Se non sono usati i dispositivi mappati in memoria, il registro di controllo deve prima essere letto dalla CPU, quindi testato, richiedendo due istruzioni invece di una, aumentando leggermente il tempo di risposta nel rilevare un dispositivo non attivo.
- Tuttavia, tutti i computer moderni hanno una qualche forma di cache; gestire la cache di un registro di controllo di dispositivi mappati potrebbe avere effetti disastrosi. L'hardware dovrebbe avere la capacità di disabilitare in maniera selettiva la cache, ad esempio a seconda delle pagine; questa proprietà aggiunge complessità sia all'hardware che al SO.
- Se siamo in presenza di un solo spazio degli indirizzi, tutti i moduli di memoria e tutti i dispositivi di I/O devono esaminare tutte le referenze di memoria per vedere a quali rispondere. Se il computer ha un singolo bus, questa operazione è semplice; tuttavia, i computer moderni hanno spesso un bus dedicato di memoria ad alta velocità, che non è visibile ai dispositivi di I/O e che quindi sono impossibilitati a rispondere. Ogni soluzione a questo problema è un compromesso che aggiunge complessità hw o sw.

Direct Memory Access (DMA)

Il **DMA** di una CPU è quel meccanismo che permette a periferiche esterne di accedere direttamente alla memoria interna per scambiare dati, in lettura e/o scrittura, senza coinvolgere l'unità di controllo della CPU per ogni byte trasferito tramite l'usuale meccanismo dell'interrupt e la successiva richiesta dell'operazione desiderata, ma generando un singolo interrupt per blocco trasferito.

ll DMA, tramite il **DMAC** (***Direct Memory Access Controller***), ha quindi il compito di gestire i dati passanti nel BUS permettendo a periferiche che lavorano a velocità diverse di comunicare senza assoggettare la CPU a un enorme carico di interrupt che ne interromperebbero continuamente il rispettivo ciclo di elaborazione.

Essenzialmente, in un trasferimento DMA un blocco di memoria viene copiato da una periferica a un'altra. La CPU si limita a dare avvio al trasferimento rilasciando il bus dati, mentre il trasferimento vero e proprio è svolto dal DMAC.

Il DMA gestisce i trasferimenti tra CPU e periferiche tramite l'utilizzo di diverse linee (Acknowledge, richiesta, controllo) e di due registri (DC e IOAR). Nel momento in cui la CPU necessita di dati presenti in memoria carica in IOAR l'indirizzo dal quale iniziare l'operazione e in DC il numero di dati consecutivi da trattare, informando il DMA su un ulteriore bit se si tratta di un'operazione di lettura o scrittura. A questo punto il DMA invia la richiesta alla periferica e nel momento in cui riceve il segnale di acknowledge inizia il trasferimento. Ad ogni passo viene incrementato IOAR e decrementato DC finché DC non è uguale a 0.
Il trasferimento tra DMA e I/O può avvenire in diversi modi:

- ***Burst Transfer***: Prevede che una volta iniziato il trasferimento il DMA mantiene il controllo del BUS a discapito della CPU, finché esso non è terminato;
- ***Cycle Stealing***: Il DMA esegue il trasferimento di parole un solo ciclo completo alla volta (cioè per ogni ciclo si interfaccia con la periferica ed esegue il trasferimento solo se è pronta, in altre parole effettuando un handshaking). Come risultato il tempo nel quale la CPU è bloccata è più frammentato;
- ***Transparent/Hidden***: Il DMA occupa il BUS solo quando la CPU non ne ha bisogno.

Ancora sugli interrupt

Quando un dispositivo di I/O finisce il lavoro che gli era stato assegnato, causa un **interrupt**, inviando un segnale sulla linea del bus che gli è stata assegnata. Questo segnale è rilevato da un chip di un controller degli interrupt della scheda madre che gestisce gli interrupt ricevuti in base all'ordine di arrivo e alla loro priorità. Per gestirlo, il controller assegna un numero alle linee degli indirizzi e lo usa come indice all'interno di una tabella chiamata **vettore degli interrupt** per prelevare un nuovo PC (program counter) che punta all'inizio della corrispondente procedura di servizio che risolve/gestisce l'interrupt.
L'hardware salva sempre determinate informazioni prima di avviare la procedura di servizio. In generale, vengono salvati il PC corrente e i registri in cui ci sono informazioni relative ai processi interrotti, così che possano essere riavviati.
Al momento di un interrupt, molte istruzioni potrebbero trovarsi in diversi stati di completamento. Un interrupt che lascia la macchina in uno stato ben definito è detto **interrupt preciso** e ha 4 proprietà:

1. Il PC è salvato in un posto conosciuto;
2. Tutte le istruzioni eseguite prima di quella puntata dal PC sono state completamente eseguite;
3. Nessuna istruzione oltre a quella puntata del PC è stata eseguita;
4. Lo stato di esecuzione dell'istruzione puntata del PC è conosciuto.

Un interrupt che non sottostà a queste specifiche è detto **interrupt impreciso**, costringendo il salvataggio di una grande quantità di registri che denotano lo stato attuale di tutte le attività

in corso al momento dell'interrupt, rendendo molto lente le operazioni di salvataggio e ripristino dello stato interno.
Il prezzo pagato per gli interrupt *precisi* è una logica di interrupt estremamente complessa all'interno della CPU. D'altra parte gli interrupt *imprecisi* rendono il SO più lento e più complicato e pertanto diventa difficile indicare quale sia l'approccio migliore.

Obiettivi del software di I/O

Un concetto fondamentale nella progettazione del software di I/O è conosciuto come **indipendenza dal dispositivo**. Il significato è che dovrebbe essere possibile scrivere programmi che possono accedere a qualsiasi dispositivo di I/O senza dover specificare in anticipo il dispositivo. Per esempio, un programma che legge un file dovrebbe essere in grado di leggere un file da disco fisso, da DVD o da chiavetta USB senza dover modificare il codice del programma per ciascun dispositivo.
L'obiettivo della definizione uniforme di nomi (**uniform naming**) è strettamente correlato all'indipendenza dal dispositivo: il nome di un file o di un dispositivo dovrebbe essere semplicemente una stringa o un numero e non dipendere in alcun modo dal dispositivo.
Un'altra importante questione è la **gestione degli errori**. In generale gli errori andrebbero gestiti il più possibile a livello hardware, dal controller o dal driver. In molti casi il recupero dell'errore avviene ai livelli più bassi, senza che i livelli superiori sappiano dell'errore.
Fondamentale è la gestione dei trasferimenti **sincroni** (bloccanti) rispetto a quelli **asincroni** (gestiti tramite interrupt). La maggior parte dell'I/O fisico è asincrono - la CPU inizia il trasferimento e smette di fare qualsiasi altra cosa finchè non arriva un interrupt; tuttavia i programmi utente sono molto più semplici da scrivere se le operazioni di I/O sono bloccanti. Sta al SO fare in modo che le operazioni asincrone appaiano bloccanti al programma utente.
Un altro problema del software di I/O è la **bufferizzazione** *(buffering)*: spesso i dati che escono da un dispositivo non possono essere memorizzati direttamente nella destinazione finale *(esempi: pacchetto che arriva dalla rete ma il SO deve prima analizzarlo e memorizzarlo; dispositivi con vincoli real-time stringenti come i quelli audio)*. L'uso di un buffer richiede molte operazioni di copia e provoca spesso un impatto pesante sulle prestazioni dell'I/O.
L'ultimo concetto suddivide i dispositivi in due categorie:

- **Dispositivi condivisi**: alcuni dispositivi di I/O, come i dischi, possono essere usati da molti utenti contemporaneamente.
- **Dispositivi dedicati**: altri dispositivi, come ad esempio le stampanti, devono essere dedicati ad un utente singolo finchè quell'utente non ha terminato. L'uso di questi dispositivi introduce una varietà di problematiche, come i deadlock.

Il SO deve ovviamente riuscire a gestire entrambi.

I/O Programmato

Per eseguire l'I/O ci sono fondamentalmente tre metodi. Il primo sistema è l'**I/O programmato**, che semplicemente delega tutto il lavoro alla CPU, la quale interroga continuamente il dispositivo di I/O per vedere se è pronto per l'uso. Questo comportamento è chiamato **polling** o **busy waiting**. l'I/O programmato è semplice ma ha lo svantaggio di occupare la CPU a tempo pieno finché tutta l'operazione di I/O non è terminata.
In un sistema embedded, in cui la CPU non ha altro da fare, il *busy waiting* è ragionevole, mentre, in sistemi più complessi, in cui la CPU ha altro da svolgere, il *busy waiting* è inefficiente.

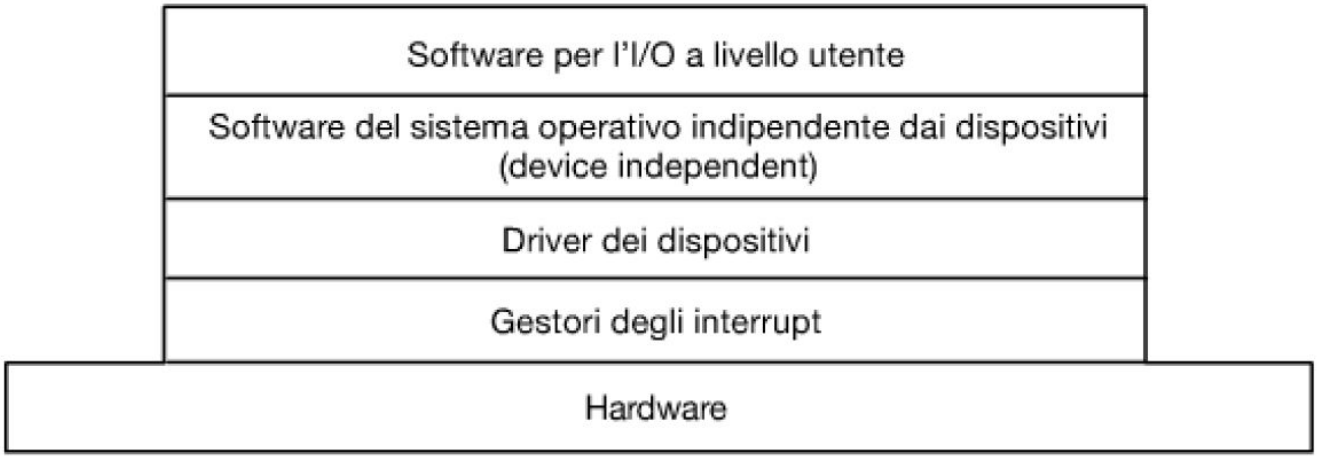
I/O Guidato dagli Interrupt

Vengono usati gli interrupt per permettere alla CPU di utilizzare i tempi "morti" durante i quali le periferiche eseguono il loro lavoro. *Esempio: una stampante che impiega 10 ms per stampare un carattere, dopo la scrittura la CPU esegue altri processi e viene bloccata da un interrupt generato dalla stampante.*
Svantaggio: gli interrupt richiedono molto tempo per essere gestiti.

I/O con DMA

Si delega il lavoro di I/O al DMAC: il DMA è I/O programmato, con il DMAC che fa tutto il lavoro, lasciando libera la CPU e riducendo il numero di interrupt.
Svantaggio: il DMAC è molto più lento della CPU.

Livelli Software I/O



Livelli del sistema del software per l'I/O.

Gestori degli Interrupt

Per la maggior parte dell'I/O vengono utilizzati gli interrupt. I passi sintetizzati che il SO segue nel gestire un interrupt sono i seguenti:

1. Salvataggio di tutti i registri (incluso il PSW) non ancora salvati dall'interrupt hardware
2. Impostazione di un contesto per la procedura di servizio dell'interrupt
3. Impostazione di uno stack per la procedura di servizio dell'interrupt
4. Avviso al controller degli interrupt. Qualora manchi il controller centralizzato degli interrupt, riabilitazione degli interrupt
5. Copia dei registri da dove erano stati salvati (eventualmente qualche stack) nella tabella dei processi
6. Esecuzione della procedura di servizio dell'interrupt. Estrarrà informazioni dai registri del controller del dispositivo che ha generato l'interrupt
7. Scelta di quale processo eseguire come successivo. Se l'interrupt ha fatto sì che qualche processo a priorità alta che era bloccato sia ora diventato pronto, potrebbe essere selezionato adesso per l'esecuzione
8. Impostazione del contesto della MMU per il processo successivo da eseguire. Potrebbe essere anche necessario impostare la TLB
9. Caricamento dei nuovi registri del processo, incluso il suo PSW
10. Avvio dell'esecuzione del nuovo processo

Driver di Dispositivo

Ciascun dispositivo di I/O connesso al computer necessita di un certo codice specifico al suo controllo. Questo codice, il **driver di dispositivo** (***device driver***), è solitamente scritto dal produttore del dispositivo stesso e rilasciato insieme a esso, con versioni differenti per i vari sistemi operativi più conosciuti. Ogni driver del dispositivo gestisce un tipo di dispositivo o al massimo una classe di dispositivi strettamente correlati. A volte, però, driver profondamente diversi sono basati sulla stessa tecnologia, ad esempio, i dispositivi che utilizzano la tecnologia USB: il trucco sta nella suddivisione a strati del driver USB (come lo stack TCP/IP).
Il driver del dispositivo deve essere parte del kernel del SO per poter accedere all'hardware del dispositivo, anche se esistono esempi di SO (come MINIX 3) che esegue i driver come procedure utente.
I SO classificano i driver in due categorie: **dispositivi a blocchi** come i dischi, contenenti molteplici blocchi di dati indirizzabili indipendentemente e i **dispositivi a carattere** come stampanti e tastiere, che generano o accettano un flusso di caratteri. La maggior parte dei SO definisce due interfacce standard, una per i dispositivi a blocchi e una per quelli a carattere, che devono essere supportate.

Software per l'I/O indipendente dal dispositivo

La funzione base del software indipendente è quella di eseguire tutte quelle funzioni di I/O trasversali a tutti i dispositivi e di fornire un'interfaccia uniforme al software a livello utente.

Interfacciamento uniforme dei driver dei dispositivi

È necessario che l'interfaccia fra i driver di dispositivo e il resto del sistema sia uniforme, altrimenti per interfacciare un nuovo driver verrebbe richiesto un grande sforzo di programmazione. Per ciascuna classe di dispositivi il SO definisce un insieme di funzioni che il driver deve supportare; spesso il driver contiene una tabella con i puntatori a se stesso per queste funzioni. Una volta caricato il driver, il SO registra l'indirizzo di questa tabella di puntatori di funzioni, così quando ha bisogno di chiamarne una può fare una chiamata indiretta tramite questa tabella, nella quale è definita l'interfaccia fra il driver e il resto del SO.
Un altro aspetto dell'uniformità di un'interfaccia riguarda la denominazione dei dispositivi di I/O: il software indipendente dal dispositivo si prende cura di mappare i nomi simbolici dei dispositivi nel driver adatto. In UNIX, un nome di dispositivo, come */dev/disk0*, specifica in modo univoco l'i-node per un file speciale e questo i-node contiene il **major device number**, usato per localizzare il driver adeguato. L'i-node contiene anche il **minor device number**, passato come un parametro al driver al fine di specificare l'unità da leggere o da scrivere. Tutti i dispositivi hanno un major device number e un minor device number e l'accesso a tutti i driver avviene usando il major device number.
Sia in Windows che in UNIX, i dispositivi appaiono nel file system con nomi di oggetti e di conseguenza si applicano le regole di protezione abituali che si applicano anche ai file.

Bufferizzazione

Esistono diversi tipi di buffering:

A. **Input senza uso di buffer**: molto svantaggioso perché causa continui interrupt e il processo utente deve essere riavviato a ogni carattere in ingresso.
B. **Buffer nello spazio utente**: il processo fornisce un buffer di *n* caratteri nello spazio utente e fa una lettura di *n* caratteri. Il buffer rischia di essere bloccato in memoria con un conseguente calo delle prestazioni.
C. **Buffer nel kernel con copia nello spazio utente**: inefficiente perché mentre il buffer viene copiato nello spazio utente non è utilizzabile.
D. **Buffering doppio**: vengono usati due buffer nel kernel, mentre uno viene usato per copiare nello spazio utente, l'altro accumula i nuovi input, una volta che il buffer è pieno il ruolo viene invertito.

E. **Buffer circolare**: composto da una zona di memoria e due puntatori, uno dei quali punta alla parola libera successiva, mentre l'altro punta alla prima parola dei dati nel buffer che non è stata ancora rimossa.

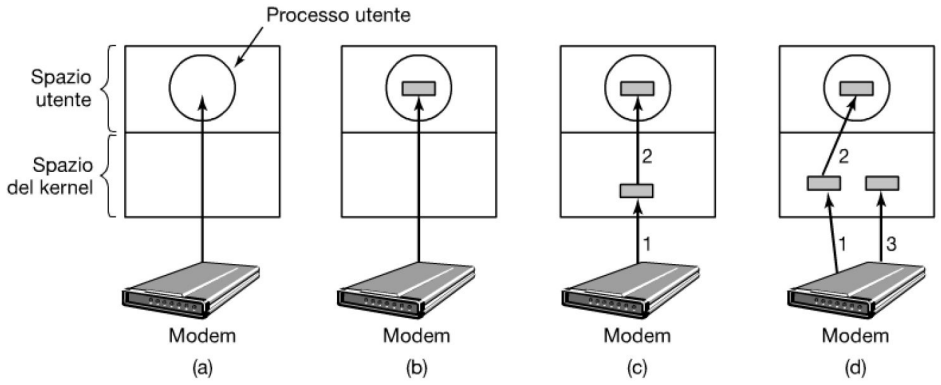


Figura 5.15 (a) Input senza uso di buffer. (b) Uso di buffer nello spazio utente. (c) Uso di buffer nello spazio del kernel seguito dalla copia nello spazio utente. (d) Doppio buffer nel kernel.

Segnalazione degli errori

Una classe di errori di I/O è quella degli errori di programmazione: questi si verificano quando un processo richiede qualcosa di impossibile(lettura da un dispositivo di OUT, scrittura su dispositivo di IN), fornire un indirizzo di buffer non valido o altri parametri e specificare un dispositivo errato(passare *disco3* quando il sistema ha solo 2 dischi). L'azione da intraprendere è semplice: riportare un codice d'errore al chiamante.
Un'altra classe di errori è quella degli errori effettivi di I/O: tentativo di scrittura su un blocco danneggiato del disco, provare a leggere da una videocamera che è stata spenta. In questi casi sta al driver decidere cosa fare e se non sa cosa fare, il problema passa alla parte di software indipendente dal dispositivo. Se è presente un utente interattivo disponibile, il software può visualizzare una finestra di dialogo con la richiesta all'utente sul da farsi; se non vi è un utente disponibile, la chiamata di sistema fallisce riportando un codice d'errore, oppure visualizzare un messaggio d'errore e terminare.

Allocazione e rilascio dei dispositivi dedicati

Alcuni dispositivi possono essere usati solo da un solo processo alla volta, sta al sistema operativo esaminare le richieste per l'uso del dispositivo e accettarle o rifiutarle, a seconda che il dispositivo sia disponibile o meno.

Dimensione dei blocchi indipendente dal dispositivo

Dischi differenti possono avere settori di diverse dimensioni. Sta al software indipendente dal dispositivo nascondere questo aspetto e fornire una dimensione di blocco uniforme ai livelli più alti.

Software di I/O nello spazio utente

Sebbene la maggior parte del software di I/O risieda nel SO, una piccola parte è composta da librerie collegate con i programmi utente e anche da interi programmi eseguiti al di fuori del kernel. Un esempio di libreria per I/O è la *stdio.h* di C.
Un'altra categoria importante è quella del sistema di **spooling**: esso è il modo di interagire con i dispositivi dedicati in un sistema multiprogrammato. Consideriamo una stampante: supponiamo che un processo apra il file speciale della stampante e poi non faccia nulla per ore: nessuno potrebbe stampare. In realtà quello che succede è che si crea un processo speciale, chiamato **daemon**, e una directory speciale detta **directory di spooling**. Per stampare un file, un processo prima genera per intero il file e poi lo mette nella directory di spooling; sarà compito del daemon, che è l'unico processo con il permesso di usare il file speciale della stampante, stampare i file nella directory. Altro esempio di processo che usa questa tecnica è il trasferimento di file su una rete.

Dischi

Hardware dei dischi

Dischi magnetici
I dischi magnetici sono organizzati in cilindri, ciascuno contenente tante tracce quante sono le testine impilate verticalmente; le tracce sono divise in settori. I dischi più vecchi facevano uscire in output un semplice flusso di bit in serie e il controller eseguiva la maggior parte del lavoro. Sui dischi più nuovi, gli **IDE** *(una volta)* e i **SATA** è l'unità del disco stesso a contenere un microcontroller che svolge una parte considerevole del lavoro e permette al controller reale di inviare un insieme di comandi di alto livello. Il controller spesso fa la cache delle tracce, rimappaggio dei blocchi difettosi e altro.
Una considerazione importante da fare è il fatto che la geometria specificata e usata dal software del driver è quasi sempre diversa dal formato fisico; i dischi moderni sono suddivisi in zone con più settori nelle zone esterne rispetto a quelle interne. Per nascondere questi dettagli, viene presentata al SO una geometria virtuale: il software si comporta come se ci siano *x* cilindri, *y* testine e *z* settori per traccia; sarà poi compito del controller rimappare la richiesta *(x, y, z)* nel disco. Oggi tuttavia i dischi supportano un sistema chiamato **indirizzamento logico dei blocchi** (***logical block addressing***), in cui i settori del disco sono numerati consecutivamente partendo da 0, senza alcuna considerazione sulla geometria del disco.
Gli indici di prestazione di un disco a testina mobile sono:

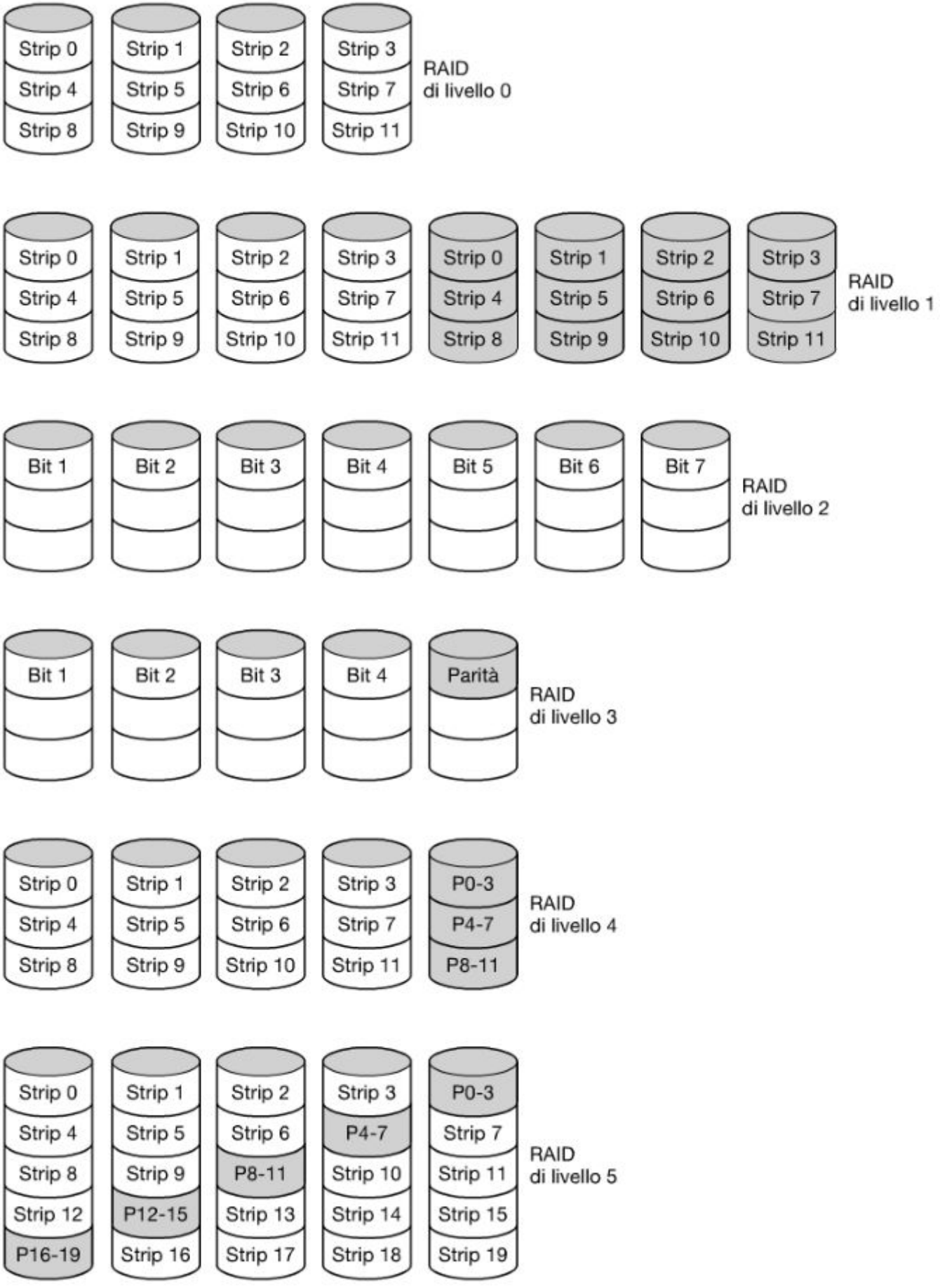
1. **Tempo di ricerca (seek)**: tempo che la testina di r/w impiega per spostarsi su un nuovo cilindro;
2. **Latenza rotazionale**: ritardo dovuto alla rotazione del disco;

3. **Tempo di trasmissione**: tempo di trasferimento effettivo dei dati.

RAID

Per velocizzare le prestazioni della CPU è stata sempre più utilizzata nel corso degli anni l'elaborazione parallela; molti pensarono che il parallelismo dell'I/O potesse essere un'idea altrettanto buona. Si arrivò così alla definizione di una nuova classe di dispositivi di I/O chiamati **RAID - Redundant array of independent disks** (*opposto allo **SLED** - single large expensive disk*). L'idea alla base del RAID è installare un contenitore pieno di dischi accanto al computer, sostituire la scheda controller dei dischi con un controller RAID, copiare di dati nel RAID e continuare con le normali operazioni. In altre parole, un RAID viene visto dal SO come uno SLED, ma con prestazioni e affidabilità migliorate.

diegochine


Strip 0
Strip 4
Strip 8
Strip 1
Strip 5
Strip 9
Strip 2
Strip 6
Strip 10
Strip 3
Strip 7
Strip 11
RAID
di livello 0
Strip 0
Strip 4
Strip 8
Strip 1
Strip 5
Strip 9
Strip 2
Strip 6
Strip 10
Strip 3
Strip 7
Strip 11
Strip 0
Strip 4
Strip 8
Strip 1
Strip 5
Strip 9
Strip 2
Strip 6
Strip 10
Strip 3
Strip 7
Strip 11
RAID
di livello 1
Bit 1
Bit 2
Bit 3
Bit 4
Bit 5
Bit 6
Bit 7
RAID
di livello 2
Bit 1
Bit 2
Bit 3
Bit 4
Parità
RAID
di livello 3
Strip 0
Strip 4
Strip 8
Strip 1
Strip 5
Strip 9
Strip 2
Strip 6
Strip 10
Strip 3
Strip 7
Strip 11
P0-3
P4-7
P8-11
RAID
di livello 4
Strip 0
Strip 4
Strip 8
Strip 12
P16-19
Strip 1
Strip 5
Strip 9
P12-15
Strip 16
Strip 2
Strip 6
P8-11
Strip 13
Strip 17
Strip 3
P4-7
Strip 10
Strip 14
Strip 18
P0-3
Strip 7
Strip 11
Strip 15
Strip 19
RAID
di livello 5

Formattazione dei dischi

Dopo la produzione, sul disco non c'è alcuna informazione di alcun genere; prima dell'uso ciascun piatto deve sottostare a una **formazione a basso livello** eseguita via software. Il formato consiste di una serie di tracce concentriche, ognuna contenente un certo numero di settori, con piccoli spazi tra loro. Ogni settore è composto da tre parti: il *preambolo*, che inizia con un determinato schema di bit che permette all'hw di riconoscere l'inizio del settore, insieme ad altre informazioni; i dati, solitamente 512 byte; infine il campo ECC che contiene informazioni ridondanti usate per ripristinare eventuali errori di lettura (solitamente 16 byte). La posizione del settore 0 è spostata di un certo offset rispetto alla traccia precedente; questo offset è chiamato **cylinder skew** (*pendenza del cilindro*) e serve a migliorare le prestazioni. L'idea è quella di consentire al disco di leggere tracce molteplici in un'operazione continua senza perdere dati. Il risultato della formattazione a basso livello è una riduzione della capacità del disco. *Uso di **interleaving singolo** e **doppio** per risolvere problemi di copia e buffer; per evitarlo, molti controller moderni bufferizzano un'intera traccia.*
Dopo la formattazione a basso livello il disco viene partizionato. Ogni partizione è vista logicamente come un disco separato e consentono la coesistenza di diversi SO. Nella maggior parte dei computer il settore 0 contiene il **MBR - Master Boot Record** contenente una parte del codice sorgente più la tabella delle partizioni. A causa dei limiti di dimensione imposti dalla MBR (*hdd di massimo 2 TB*), la maggior parte dei SO supportano anche la **GPT - GUID Partition Table**, che supporta dimensioni del disco fino a 9.4 ZB. La tabella delle partizioni fornisce il settore di avvio e la dimensione di ciascuna partizione.
Il passaggio finale è eseguire una **formattazione ad alto livello** di ciascuna partizione. Questa operazione stabilisce blocco di avvio, elenco dei blocchi liberi, la directory principale e un file system vuoto; viene inoltre messo un codice nella tabella delle partizioni indicando quale file system sia utilizzato nella partizione.
Una volta avviato il sistema, parte il BIOS che legge il MBR, in particolare il boot record della partizione attiva e la fa partire; viene quindi ricercato nel file system il kernel del SO che sarà poi caricato ed eseguito.

Algoritmi di scheduling del braccio del disco

Tre criteri per misurare le strategie:

1. **Throughput**: numero di richieste servite per unità di tempo;
2. **Tempo medio di risposta**: tempo medio di attesa che serve affinché la richiesta sia servita e utilizzabile;
3. **Varianza del tempo di risposta**: prevedibilità del tempo di risposta.

Obiettivi generali:

- **Massimizzare il throughput**;
- **Minimizzare il tempo di risposta e le relative varianze**.

I primi algoritmi di scheduling del disco si concentravano sulla minimizzazione del tempo di seek, i sistemi moderni ottimizzano anche il tempo di rotazione.

First Come First Served (FCFS)

Comporta notevoli svantaggi dato che le richieste vengono servite in ordine cronologico di arrivo. La ricerca dei dati con posizioni distribuite in modo casuale produce lunghi tempi d'attesa e scarso throughput, inoltre, sotto carico pesante, il sistema può comportarsi in modo critico (*trashing*).

Vantaggi:

- Equo
- Previene l'attesa infinita
- Basso *overhead*

Svantaggi:

- Possibile throughput estremamente basso

Shortest Seek Time First

Gestisce come richiesta seguente la più vicina fisicamente alla testina di lettura/scrittura.

Vantaggi:

- Throughput maggiore e tempi di risposta inferiori a FCFS
- Soluzione ragionevole per sistemi di elaborazione batch

Svantaggi:

- Non garantisce equità
- Possibilità di attesa infinita
- Alta varianza dei tempi di risposta
- Il tempo di risposta è generalmente inaccettabile per i sistemi interattivi

SCAN

Anche chiamato algoritmo dell'ascensore, è simile a SSTF ma gestisce come richiesta seguente la più vicina fisicamente alla testina di lettura/scrittura rispetto a una direzione preferita, che cambia quando si è raggiunto il limite del disco.

Vantaggi:

- Migliora la varianza dei tempi di risposta rispetto a SSTF

Svantaggi:

- Possibilità di attesa infinita

C-SCAN (Circolare)

Simile a SCAN, ma con la differenza che si muove sempre nella stessa direzione per servire le richieste: alla fine di una scansione verso l'interno, il braccio del disco salta al cilindro più esterno, senza servire richieste.

diegochine

Vantaggi:

- Migliora la varianza dei tempi di risposta rispetto a SCAN

Svantaggi:

- Peggiora il throughput e il tempo medio di risposta

FSSCAN

Congela periodicamente la coda delle richieste al disco e serve solo le richieste in coda per quel momento.

N-Step SCAN

Serve solo le prime *n* richieste nella coda in quel momento.

Vantaggi:

- Entrambe riducono la varianza dei tempi di risposta rispetto a SCAN
- Entrambe prevengono l'attesa infinita

LOOK (look ahead)

Migliora lo scheduling SCAN continuando fino al termine dell'attraversamento attuale per servire le richieste, se non ce ne sono cambia direzione.

Vantaggi:

- Migliora l'efficienza evitando operazioni inutili di ricerca
- Throughput elevato

C-LOOK

Migliora lo scheduling C-SCAN combinandolo con LOOK: quando non ci sono più richieste nell'attraversamento verso l'interno si sposta verso le richieste posizionate più all'esterno senza servirne altre in mezzo e inizia un nuovo attraversamento.

Vantaggi:

- Migliora la varianza dei tempi di risposta rispetto a LOOK

Svantaggi:

- Peggiora il throughput

Strategie recentemente sviluppate tentano di ottimizzare le prestazioni del disco riducendo la latenza rotazionale.

Shortest Latency Time First

In un dato cilindro, serve le richieste con la minima latenza di rotazione. È molto facile da implementare e raggiunge prestazioni quasi ottimali per la latenza di rotazione.

Shortest Positioning Time First

Il tempo di posizionamento equivale alla somma del tempo di ricerca e la latenza di rotazione.
SPTF serve per prima la richiesta con il minor tempo di posizionamento.

Vantaggi:

- Buone prestazioni generali

Svantaggi:

- Può causare attesa infinita

Shortest Access Time First

Il tempo di accesso equivale alla somma del tempo di posizionamento e il tempo di trasmissione.
Variante di SPTF, in cui viene servita per prima la richiesta con il minor tempo di accesso.

Vantaggi:

- Throughput elevato

Svantaggi:

- Può causare attesa infinita

Sia SPTF e SATF possono implementare LOOK (ahead) per migliorare le prestazioni.

Svantaggi:

- Sia SPTF e SATF richiedono la conoscenza delle caratteristiche di prestazioni del disco che potrebbero non essere immediatamente disponibili (tempi di seek, latenza, posizioni dei settori)
- Possibile mascheramento al SO per controllo degli errori, correzione dei dati e riassegnazione trasparente dei settori danneggiati

Gestione degli errori

Due approcci generali:

- Trattarli a livello di controller: prima che il disco esca dalla fabbrica, viene verificato e viene scritta una lista di settori difettosi; ciascuno di questi è rimappato dal controller in uno dei settori di riserva
- Trattarli a livello di SO: il SO acquisisce l'elenco dei settori difettosi e costruisce delle tabelle di rimappaggio.

Memoria Stabile

La **memoria stabile** è un sottosistema del disco con la seguente proprietà: quando gli viene inviata un'operazione di scrittura, il disco o scrive correttamente i dati o non fa niente, lasciando i dati esistenti inalterati. La memoria stabile utilizza una coppia di dischi identici con i blocchi corrispondenti che lavorano insieme a formare un blocco esente da errori. In assenza

di errori i blocchi corrispondenti di entrambi i dischi sono identici. Per ottenere questo risultato sono definite le tre operazioni seguenti:

1. **Operazioni di scrittura stabili**: consiste nello scrivere il blocco sull'unità 1, quindi leggerlo per verificare se sia stato scritto correttamente; se non è stato scritto correttamente, si ripete per *n* volte il procedimento. Dopo *n* tentativi falliti il blocco è rimappato con uno di riserva e l'operazione ripetuta da capo. Una volta scritto correttamente il blocco sull'unità 1, si ripete l'intero procedimento sull'unità 2.
2. **Operazioni di lettura stabili**: si legge per primo un blocco dall'unità 1, se porta un ECC scorretto si ritenta *n* volte l'operazione di lettura; dopo *n* fallimenti viene letto il blocco corrispondente dall'unità 2.
3. **Crash recovery**: dopo un crash, un programma di ripristino fa la scansione di entrambi i dischi confrontando i blocchi corrispondenti. Se una coppia di blocchi è valida e sono identici non si fa nulla; se uno di loro presenta un errore ECC, il blocco difettoso è sovrascritto da quello validO; se sono entrambi senza errori ma diversi, il blocco del disco 1 sovrascrive quello del disco 2.

In assenza di crash della CPU questo schema funziona sempre.

Clock

Interfacce utente: tastiera, mouse, monitor

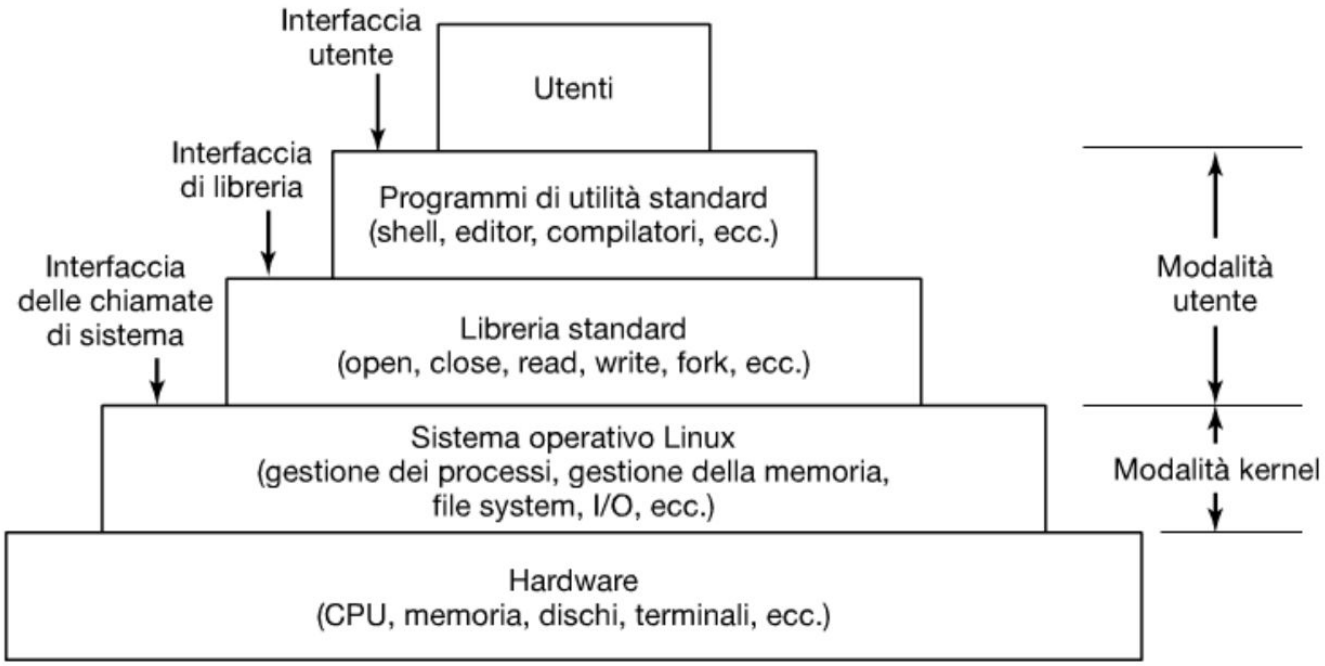
CAPITOLO 10: UNIX, LINUX E ANDROID

Partiamo con **Linux**, una variante molto diffusa di **UNIX**, eseguibile su moltissimi computer. È uno dei principali sistemi operativi di workstation e server di fascia alta, ma viene impiegato anche su sistemi che vanno dagli smartphone (**Android**) ai supercomputer.
Il numero di versione del kernel di Linux è composto da quattro numeri, come ad esempio 2.6.9.11. Il primo numero denota la versione del kernel; il secondo la versione principale (***major revision***); il terzo a versioni minori (***minor revision***), come l'aggiunta di nuovi driver. Il quarto indica la correzione di errori minori o patch di sicurezza. Una caratteristica insolita è il fatto che Linux sia un software libero. Nonostante ciò, la licenza **GPL** (**GNU public license**) è più lunga di quella di Windows e specifica che gli utenti possono usare, copiare, modificare e ridistribuire il codice sorgente e binario in maniera gratuita. Il limite fondamentale è che tutto il lavoro derivato dal kernel di Linux non può essere venduto o ridistribuito solo in forma binaria; il codice deve essere consegnato con il prodotto o può essere messo a disposizione a richiesta.

Obiettivi di Linux

xxx

Interfacce di Linux



Il sistema Linux può essere considerato come una sorta di piramide: alla base c'è l'hardware, che consiste di CPU, memoria, dischi, monitor tastiera e altri dispositivi; a seguire c'è il SO, la cui funzione è quella di controllare l'hardware e fornire un'interfaccia per le chiamate di sistema a tutti i programmi; queste chiamate di sistema consentono ai programmi utente di creare e gestire processi, file e altre risorse. I programmi fanno chiamate di sistema inserendo i parametri nei registri (o qualche volta nello stack) e trasmettendo istruzioni di trap per passare dalla modalità utente alla modalità kernel. Poiché non c'è modo di scrivere un'istruzione di trap in C, si fornisce una libreria con una procedura per chiamata di sistema, scritta in assembly.

Struttura del kernel

Il kernel si trova direttamente sull'hardware, abilita le interazioni con i dispositivi di I/O e l'unità di gestione della memoria e controlla l'accesso della CPU a essi. Da ricordare che il kernel di Linux è di tipo monolitico ma contiene componenti modulari. Al livello più basso esso contiene i gestori degli interrupt per interagire con i dispositivi e il meccanismo di distribuzione *(dispatch)* di basso livello; questo dispatch si verifica quando ha luogo un interrupt e sostanzialmente effettua il cambio di contesto del processo in esecuzione e avvia il driver appropriato.
Successivamente, suddividiamo i vari sottosistemi del kernel in tre componenti principali. Il **componente per l'I/O** contiene tutte le parti del kernel responsabili dell'interazione con i

dispositivi e dell'esecuzione delle operazioni di I/O e dell'esecuzione di operazioni di I/O verso la rete e verso la memoria. Al livello più alto, le operazioni di I/O sono integrate nel livello **VFS - virtual file system**, che è in sostanza un'astrazione delle operazioni sul file system. Al livello più basso tutte le operazioni di I/O passano per i driver di dispositivo. Il **componente per la gestione della memoria** include il mantenimento di una cache di pagine a cui si ha avuto accesso recentemente e l'implementazione di una buona politica di sostituzione delle pagine, oltre alla lettura in memoria su richiesta di nuove pagine di codice e di dati richiesti. Infine, il **componente per la gestione dei processi** si occupa di creare e distruggere processi. Include anche lo scheduler dei processi. Questi tre componenti sono strettamente interdipendenti.
Oltre ai componenti statici nel kernel, Linux supporta il caricamento dinamico di moduli, che possono essere usati per aggiungere o sostituire driver, il file system, il networking o altro codice del kernel.

Gestione dei processi in Linux

Le principali entità attive in un sistema Linux sono i processi, che sono molto simili ai classici processi sequenziali. Ciascun processo avvia un solo programma e all'inizio ha un solo thread di controllo (un solo PC), ma ad un processo è consentito creare thread aggiuntivi *(Linux è multiprogrammato)*. Ciascun utente può avere diversi processi attivi contemporaneamente, così su un grosso sistema ci possono essere moltissimi processi in esecuzione; anche quando l'utente è assente sono in esecuzione dozzine di processi in background ***(daemon)***.
I processi sono creati in modo molto semplice: la chiamata di sistema *fork* da parte di un ***processo genitore*** crea una copia esatta di tale processo, chiamata ***processo figlio***; i due processi hanno ognuno una immagine di memoria privata. Per distinguere i due processi (*bisogna sapere chi deve eseguire quale codice*) la chiamata a fork restituisce uno 0 al figlio e un valore diverso da 0, il **PID** (***process identifier***) del figlio al padre. I processi sono indentificati dal loro PID.
I processi in Linux possono comunicare tra loro usando dei canali su cui un processo può scrivere un flusso di byte che l'altro leggerà; sono chiamati **pipe**. Un altro modo che hanno per comunicare è tramite degli interrupt software chiamati **segnali**, che sostanzialmente sono messaggi ai quali un processo può rispondere in diversi modi. Un processo può inviare segnali solo ai membri del suo **gruppo di processi** (padre, antenati, fratelli e discendenti).

Implementazione di processi e thread in Linux. Il kernel di Linux rappresenta internamente i processi come task, attraverso una struttura *task_struct*. Questa struttura contiene una varietà di campi, di seguito riassunte con alcuni esempi (non completi):

1. **Parametri di scheduling**, come priorità del processo, tempo della CPU usato, tempo a riposo;
2. **Immagine della memoria**: puntatori ai segmenti testo, dati, stack, page table;
3. **Segnali**: maschere che mostrano come gestire i vari segnali;

4. **Registri della macchina** (salvati quando si verifica un trap nel kernel);
5. **Stato della chiamata di sistema**;
6. **Tabella dei descrittori dei file**;
7. **Accounting**: puntatore a una tabella che tiene traccia del tempo della CPU utente e di sistema impiegato dal processo;
8. **Stack del kernel**;
9. **Varie**: stato attuale del processo, PID, PID del padre..

Per quanto concerne i thread, nel 2000 Linux ha introdotto una nuova chiamata di sistema molto potente, la **clone**. Prima, quando si creava un nuovo thread, il thread originale e quello nuovo condividevano tutto ad eccezione dei registri, come descrittori di file aperti, gestori di segnali e altre proprietà specifiche del thread e non del processo; quello che permette di fare la clone è consentire a tutti questi aspetti di divenire specifici per il processo o per il thread.

Scheduling in Linux. I thread di Linux sono thread del kernel, quindi lo scheduling si basa sui thread e non sui processi. Vengono distinte tre classi di thread (*i real-time hanno livelli di priorità da 0 a 99, i time sharing da 100 a 139*):

1. **Real-time FIFO**: hanno la massima priorità e non sono preemptable se non da parte di nuovi thread real-time FIFO con priorità maggiore;
2. **Real-time round-robin**: identici ai precedenti, ma hanno associati ad essi dei quanti di tempo e sono preemptable da parte del clock. Se molteplici thread di questo tipo sono pronti, ciascuno viene avviato per il suo quanto e dopodichè va alla fine della lista dei thread real-time round-robin. Nessuna in queste classi in realtà è real-time, hanno semplicemente maggiore priorità della terza classe;
3. **Time sharing standard**: gestiti sempre con politica round-robin.

Una struttura dati usata dallo scheduler di Linux è una **runqueue** *(coda di esecuzione)*, la quale traccia tutti i task eseguibili dal sistema e seleziona quello da eseguire in seguito; ogni CPU ha associata una runqueue.
Uno scheduler Linux molto diffuso era lo **scheduler O(1)**, che era in grado di eseguire le operazioni di gestione dei task in tempo costante. *Funziona così: runqueue con due array di 140 liste (una per livello di priorità), active ed expired; lo scheduler sceglie un task dall'array dei task attivi di massima priorità. Se il suo quanto scade, viene spostato in una lista dei processi scaduti; se il task si blocca prima della fine del quanto, quanto torna in ready può riprendere la sua esecuzione con quanto ridotto e alla terminazione viene messo nei processi scaduti. Quando l'array dei processi attivi è vuoto, scambia i puntatori dei due array e ricomincia.* L'idea base di questo schema è che i processi CPU-bound ricevono in sostanza i servizi rimasti quando tutti i processi I/O-bound e interattivi sono bloccati.
Poiché Linux (come qualsiasi altro SO) non può sapere a priori se un task è I/O o CPU-bound, esso fa affidamento all'euristica dell'interattività. Ad ogni task è assegnato un valore

interpretabile come priorità statica (detto anche valore di **affinità** - **nice**); il default è 0 e varia in [-20, 19], con valori bassi che indicano priorità più alta.
Oggi lo scheduler più diffuso è il **CFS** - **Completely Fair Scheduler** *(usa un albero rosso-nero)*.

Gestione della memoria

Il modello di memoria di Linux è semplice, tale da rendere i programmi portabili; non è cambiata molto negli ultimi decenni poiché ha sempre funzionato bene.
Ogni processo Linux ha uno spazio degli indirizzi che consiste logicamente di tre segmenti: testo, dati e stack:

- Il **segmento testo** contiene le istruzioni macchina che formano il codice eseguibile del processo; normalmente è di sola lettura. La sua dimensione non cambia nel tempo.
- Il **segmento dati** contiene la memoria per tutte le variabili, stringhe e dati del programma; è formato da due parti, dati inizializzati e dati non inizializzati. Diversamente dal segmento testo, il segmento dati può variare; i programmi modificano le loro variabili continuamente e gestiscono la memoria in maniera dinamica. Il descrittore dello spazio degli indirizzi del processo contiene informazioni sull'intervallo di memoria allocata in modo dinamico nel processo, chiamato **heap**.
- **Segmento stack**: serve per permettere ad un programma di scoprire quali sono i suoi parametri (leggendoli dallo stack); nella maggior parte delle macchine inizia in cima allo spazio degli indirizzi virtuale e scende verso 0.

Molti sistemi Linux, inoltre, supportano i **segmenti testo condivisi**.

I/O in Linux

La soluzione di Linux per permettere ai programmi di accedere ai dispositivi di I/O consiste nell'integrare i dispositivi nel file system nei cosiddetti file speciali. Fondamentalmente, tutti i dispositivi di I/O sono creati per assomigliare ai file e vi si accede con le stesse chiamate di sistema *read* e *write* utilizzate per accedere a tutti i file comuni. Ad ogni dispositivo di I/O si assegna un *path name*, solitamente in */dev* (*/dev/hd1 potrebbe essere un disco, /dev/net una rete..*). Si utilizza il modello dei ***file speciali a blocco*** e ***a caratteri***. Associato a ciascun file c'è un driver del dispositivo che lo gestisce; ciascun driver ha un numero del ***dispositivo major*** per identificare il dispositivo e opzionalmente un numero del ***dispositivo minor*** se supporta molteplici driver.

Networking.
Un importante dispositivo di I/O è la rete, di cui UNIX Berkeley fu il pioniere, utilizzato poi da Linux più o meno allo stesso modo. Il concetto chiave sono i **socket**: sono connessioni logiche che permettono all'utente di interfacciarsi alla rete per inviare o ricevere messaggi.

File System in Linux

Il primo file system di Linux fu il file system MINIX 1. Tuttavia esso era fortemente limitante *(nomi dei file di massimo 14 caratteri; dimensione massima dei file esagerata)*, per cui venne sviluppato prima il file system *ext* e successivamente migliorato in ***ext2***, che divenne il file system principale.
Un file in Linux è una sequenza di 0 o più byte contenenti informazioni arbitrarie; non c'è distinzione fra file ASCII, binari o altro tipo, il significato dei file spetta interamente al proprietario dei file. *I nomi sono limitati a 255 chars e tutti i caratteri ASCII al di fuori di NULL sono consentiti*.

File system virtuale.
Per far sì che le applicazioni interagiscano con diversi file system Linux adotta un approccio denominato **VFS - Virtual File System**. Il VFS definisce un insieme di astrazioni del file system di base e le operazioni consentite su queste astrazioni. Le principali astrazioni sono:

- **Superblocco**: file system specifico, contiene le informazioni critiche sul layout del file system e la sua distruzione rende il file system illeggibile;
- **inode**: file specifico di cui è la descrizione (*directory e dispositivi fanno parte di questa categoria*);
- **Dentry**: voce della directory, singola componente di un percorso; è creata al volo dal file system e memorizzata in una cache. Ciò vuol dire che se diversi processi accedono allo stesso file attraverso lo stesso percorso, il loro oggetto file punterà alla stessa voce in cache.
- **File**: è la rappresentazione in memoria di un file aperto ed è creata in risposta alla chiamata di sistema *open*. Supporta *read, write, sendfile, lock* ed altre operazioni.

File system ext2 di Linux
Il blocco 0 non è utilizzato da Linux e spesso contiene un codice per avviare il pc. Dopo il blocco 0, la partizione del disco è divisa in gruppi di blocchi, indipendentemente da dove iniziano e finiscono i cilindri del disco e organizzato in questo modo:

- Il primo blocco è il **superblocco**. *Contiene info su layout del file system, numero di inode, blocchi liberi..*
- Successivamente c'è il descrittore del gruppo, *contiene info sulle bitmap e sugli inode.*
- Due bitmap tengono traccia rispettivamente dei blocchi liberi e degli inode liberi.
- A questo punto ci sono gli inode stessi, che contengono informazioni di gestione e dati del file.
- Infine ci sono i blocchi dei dati, dove sono memorizzati tutti i file e le directory. Se un file o directory è costituita da più di un blocco, questi non necessariamente devono essere contigui sul disco.

diegochine

File system /proc

File system creato per fornire informazioni in tempo reale sullo stato del nucleo e i processi di sistema; consente agli utenti di ottenere informazioni dettagliate che descrivono il sistema stesso, dalle informazioni di stato hardware ai dati che descrivono il traffico di rete. Esiste solo nella memoria principale e le chiamate procfs *read* e *write* possono accedere ai dati del nucleo, permettendo agli utenti di inviare i dati al nucleo.

CAPITOLO 11: WINDOWS 8

rip