

--- CAPITOLO 1 : INTRODUZIONE ---

Vedremo come è strutturato ed organizzato un sistema operativo, l'architettura.

Kernel è il core del sistema operativo.

Non c'è una definizione del sistema operativo.

Gestisce tutte le risorse.

Che cos'è un sistema operativo?

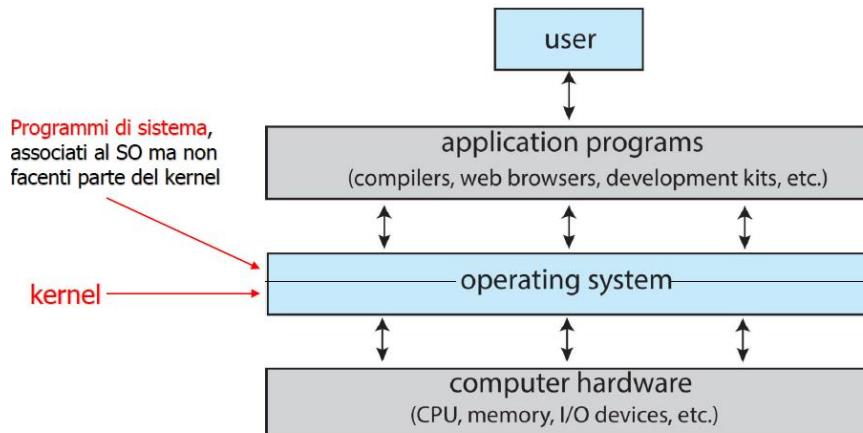
E' un programma che fa da intermediario tra l'utente e tutto l'hardware del computer.

Gli obiettivi del sistema operativo :

- Eseguire programmi per l'utente.
- Rendere il sistema informatico il più conveniente possibile.
- Usare l'hardware del computer in maniera efficiente.

Il sistema informatico può essere diviso in 4 componenti:

- 1) **Hardware** → CPU, memoria, dispositivi di Input e output.
- 2) **Sistema operativo** → Controlla e orchestra l'uso dell'hardware e varie applicazioni e utenti
- 3) **Applicazioni** → Definisce le modalità con cui le risorse di sistema vengono utilizzate per risolvere i problemi informatici degli utenti
- 4) **Utenti** → Persone, macchine, altri computer



Cosa fa un sistema operativo?

Dipende dal punto di vista.

L'utente vuole che **sia facile da usare** e che **offra ottime performance**, non vuole interessarsi come risolve la **condivisione delle risorse**.

Ma i computer condivisi come mainframe o minicomputer devono accontentare tutti gli utenti.

- Il sistema operativo è un programma di allocazione di risorse e di controllo che fa un uso efficiente dell'HW e gestisce l'esecuzione dei programmi utente.

Gli utenti di sistemi dedicati come le **workstation** dispongono di risorse dedicate ma utilizzano frequentemente risorse condivise dai **server**.

I dispositivi mobili come smartphone e tablet sono poveri di risorse, ottimizzati per l'usabilità e la durata della batteria.

- Interfacce utente mobili come touch screen, riconoscimento vocale

Definizione sistema operativo

Come vedete sono molto prolissi sulla definizione. Non c'è una definizione universale.

Possiamo conoscere la parte centrale che è il kernel, è quella parte del sistema operativo che è sempre in esecuzione.

Tutto quello che non è kernel, sono o i programmi di sistema e i programmi applicativi.

- Un **programma di sistema** è fornito dal sistema operativo (ma non fa parte del kernel).
- Un **programma applicativo**, tutti i programmi non associati ad un sistema operativo.

I **middleware** sono programmi intermedi che risolvono determinati problemi. Per esempio è un dmbs per i database. Cioè i **middleware** sono un insieme di framework software che forniscono servizi aggiuntivi agli sviluppatori di applicazioni come database, multimedia, grafica.

Organizzazione di un sistema informatico

C'è una o più CPU che soddisfano le richieste degli utenti tramite il sistema operativo.

Ci sono le periferiche hardware come mouse, tastiera, dischi per la memorizzazione. Ognuno di questi ha bisogno di un programma per essere usato dal sistema operativo.

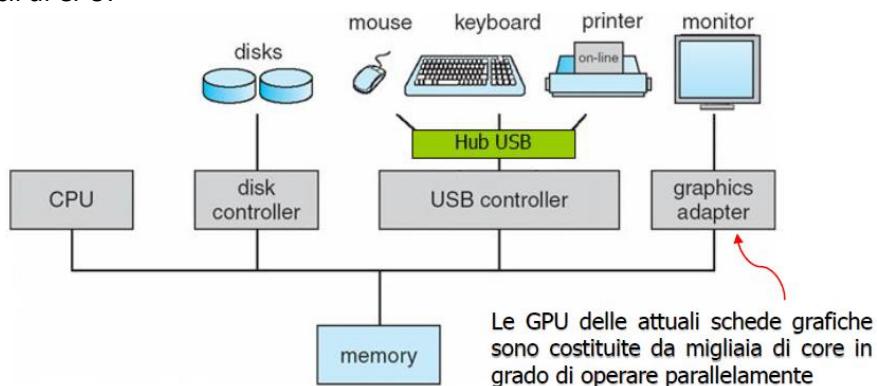
Ci sono i controller connessi tramite le porte USB.

C'è un adattatore grafico che è il monitor.

Tutte queste componenti sono messi in collegamento, tramite **bus** comune fornendo accesso alla memoria condivisa.

A supporto del sistema operativo c'è la memoria principale ed è utile per gestire le risorse e quant'altro.

I programmi applicativi competono per l'utilizzo delle risorse, cioè sia la memoria, sia le varie risorse di I/o e sia i vari cicli di CPU.



Operazioni del sistema informatico

- Gestire i vari dispositivi di I/O e la CPU deve eseguirle concorrentemente.
- Gestisce ogni controller device che è collegato. (Tramite **driver**)
- C'è un driver del device per poterlo gestire.
- La CPU muove i dati da/a memoria principale ai buffer locali
- I/O sono dal device al buffer locale del controller
- Causa un **interrupt**: Il device informa la CPU che l'esecuzione del programma è finito.

Interrupt : Device informa la CPU che è appena accaduto un evento che richiede attenzioni.

(C'è sia l'interrupt lato hardware e lato software).

L'interrupt serve per notificare alla CPU che si è verificato un evento che va gestito.

In generale l'interrupt trasferisce il controllo attraverso un **vettore degli interrupt**, che contiene gli indirizzi di tutte le routine di servizio.

Nella gestione dell'interrupt la CPU deve memorizzare lo stato sospeso.

Struttura I/O

Per la gestione delle operazioni di I/O ci sono 2 approcci diversi:

- 1) Il primo caso è bloccante, ovvero parte l'operazione di I/O e il controllo ritorna all'utente solo nel momento in cui che l'operazione di I/O è completata.
- 2) Nel secondo caso viene fatta l'operazione di I/O, si lascia al suo destino e il programma continua (come per esempio quando effettuo una stampa posso fare altro sul pc durante la stampa).

Alcune considerazioni su questi due approcci :

Nel primo prevede che ci sia un istruzione di wait che lascia la cpu in uno stato di attesa fino al prossimo interrupt. C'è un loop di attesa, il problema è che non puoi far partire altre richieste di I/O.

Nel secondo caso invece, quando viene fatta partire l'operazione di I/O ma l'utente ottiene comunque il controllo, c'è una **system call** che chiede al sistema operativo di attendere il completamento dell'operazione di I/O ma molto più importante il sistema operativo deve ottenere per ogni device il tipo, l'indirizzo e lo stato, che tramite questa **tavella** mi dice quale device è pronto per eseguire un'operazione di I/O e quali stanno eseguendo operazioni di I/O.

AVVIO DEL SISTEMA OPERATIVO NEL MOMENTO IN CUI VIENE ACCESO IL CALCOLATORE

C'è un **programma di bootstrap** che è caricato all'accensione o al riavvio del calcolatore.

Ed è quello a cui è assegnato il compito di far partire il kernel del caricatore.

Il kernel è solitamente nella ROM (memoria di sola lettura) è noto anche come **firmware**.

Inizializza tutti gli aspetti del sistema.

Carica il kernel del sistema operativo che a sua volta carica tutti i moduli.

STRUTTURE DELLA MEMORIA

All'interno di un calcolatore abbiamo:

- La memoria principale, l'unica memoria accessibile tramite la CPU.

Caratteristiche della memoria principale :

1) **Volatile**.

2) Memoria **RAM** nella forma **DRAM (Dynamic Random Access Memory)**.

- **Memoria secondaria**, estensione della **memoria principale** che non è volatile, il dato persiste anche allo spegnimento del PC.
- **HDD Hard Disk Drives**, rigidi di metallo o in fibra di vetro, ricoperti di materiale magnetico, è sicuramente più lenta di una RAM.
- **Non volatile memory (NVM)** memorie non volatili, prive di organi meccanici, più veloci degli hard disk, sempre più diffusi.

GERARCHIA DELLE MEMORIE:

Le unità di misura che usiamo per fare la gerarchia:

- 1) Velocità
- 2) Costo
- 3) Volatilità.

Definizione di caching : E' quell'operazione che sposta dalla memoria più lenta a quella veloce un pò di dati.

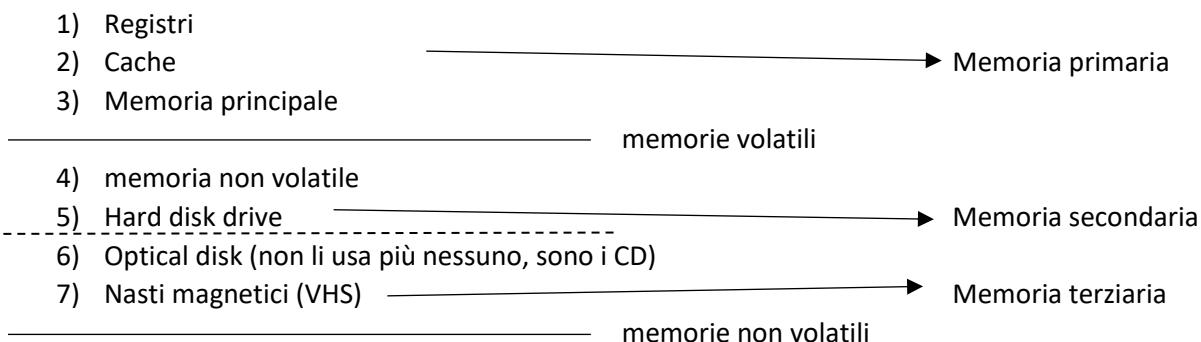
E' utile così ottimizzzo le prestazioni del computer dei dati che vengono utilizzati più frequentemente.

E' fondamentale per ottimizzare il calcolatore.

Ogni tipologia di memoria ha il suo driver per gestire il caching per gestire un interfaccia uniforme per le operazioni di I/O.

GERARCHIA DI MEMORIA:

(Cosa vede per primi la CPU)



6 e 7 (memoria terziaria, che è tutto quello che può essere estratto dal calcolatore (CD e VHS))

L'architettura di Von Neuman è utilizzata oggi.

In memoria ci sono istruzioni e dati.

STRUTTURA DIRECT MEMORY ACCESS

DMA → è una metodologia che consente ai dispositivi di I/O di bypassare il trasferimento dei dati attraverso alla CPU, quindi vado direttamente in memoria centrale.

Ha il vantaggio che non si interrompe la CPU, è ovviamente più veloce.

- Utilizzato per dispositivi I/O ad alta velocità in grado di trasmettere informazioni a velocità prossime a quelle della memoria
- Il controller del dispositivo trasferisce blocchi di dati dalla memoria buffer direttamente alla memoria principale senza l'intervento della CPU
- Viene generato un solo interrupt per blocco, anziché quello

OPERAZIONI DEL SISTEMA OPERATIVO

- 1) Programma di bootstrap che fa partire il kernel del sistema operativo.
- 2) Caricamento kernel, una volta che il kernel si avvia fa partire tutta una serie di demoni, perché girano in autonomia per fornire ulteriori servizi di sistema.
- 3) Il kernel fa partire una **serie di demoni**
- 4) Il **kernel è guidato da interventi di interrupt** che possono essere di tipo hardware o software.

È per esempio un interrupt di tipo software un errore ad un software, una system call, loop infiniti, programmi che cercano di modificarsi l'uno con l'altro. (loop infiniti e l'ultimo sono interruputi).

MULTIPROGRAMMAZIONE

Coesistenza di diversi programmi in modo che tale da impiegare la CPU sempre in esecuzione.

Vengono implementate politiche di **scheduling**.

È lo scheduler che si occupa di applicare politiche di programmi da eseguire.

Anche nel momento in cui uno di questi fa richieste di I/O.

E conservare lo stato dei processi in modo tale di alternare l'esecuzione senza perdere i dati.

MULTITASKING (TIMESHARING)

E' legato alla multiprogrammazione, è un'estensione logica

L'idea di fondo è che la CPU ad intervalli di tempo regolari alterna i vari processi in esecuzione in modo tale che l'interazione che chiede l'utente è in tempo reale con uno di essi, come per esempio quando ho tanti software aperti.

E' la CPU che la gestisce nel minor tempo possibile così sembra che sia in tempo reale agli occhi dell'utente (< 1 secondo).

Distinzione tra programma e processo:

- Il **programma** è un pezzo di software che è passivo
- Il **processo** è un'istanza in esecuzione di un programma.

Avere più processi in esecuzione adotta politiche di **CPU scheduling**

Deadlock: Processo che è in attesa di una risorsa perché è utilizzata da un'altra risorsa o un processo che non riesce ad accedere ad una risorsa .

Uno dei limiti è che la memoria non è infinita. Questo significa che è sempre il SO a fare swapping.

Più processi possono fare riferimento ad un'unico programma.

DOPPIA MODALITÀ CON CUI OPERA IL SISTEMA OPERATIVO

Ci sono due modi, la modalità utente e la modalità kernel.

Ed è il modo con cui il sistema operativo si mette a protezione di eventuali azioni malavoli eseguiti dagli applicativi dell'utente, o da applicativi che non fanno parte nel suo ambiente, come porzioni di codice del kernel di vari moduli che fanno parte del kernel.

Per distinguere questa modalità c'è un bit.

Nel caso in cui l'applicativo utente richiede un'operazione che è un'operazione privilegiata passa attraverso una system call, il sistema operativo valuta se l'operazione può essere eseguita e si prende in carico l'operazione, esegue quell'operazione e restituisce il controllo al programma applicativo con l'eventuale risultato di quell'operazione.

Quando parte il sistema operativo dopo l'operazione di bootstrap il bit è in modalità kernel mode.

Il sistema operativo è l'unico che può cambiare il bit di modalità.

L'esecuzione viene assegnato un programma applicativo indicandogli il bit di modalità.

TIMER

Con il multitasking ci può essere più di un processo.

Uno dei modi che un processo ottimizza l'utilizzo delle risorse.

Il timer attraverso l'interrupt avvisa che è scaduto il tempo dell'esecuzione

Il timer previene i loop infiniti.

E' il SO a settare il timer.

Quando scade il tempo viene generato un interrupt e la CPU sa che è terminata l'esecuzione del processo e il timer fa partire il nuovo processo.

C'è quindi un intervallo di esecuzione.

GESTIONE PROCESSO

Il processo è un'istanza di un programma in esecuzione.

Il programma non ha bisogno di risorse perché è un pezzo di codice scritto, il processo sì, perché usa cicli di CPU, memoria, ecc...

Quando il processo termina le risorse vanno recuperate per assegnarle ad altri processi.

Il processo ha un **program counter** (che dice qual'è l'istruzione successiva da eseguire), il program counter serve per eseguire sequenzialmente tutte le istruzioni del processo così com'è scritto nel programma.

I processi multithreading hanno un program counter per ogni thread in esecuzione.

In un sistema coesistono diversi processi, sia per gli utenti e sia del sistema operativo ed è compito del SO di assegnarli in modo efficiente ed evitare problemi di deadlock, loop infiniti e starvation.

ATTIVITA' DI GESTIONE DEI PROCESSI

Il sistema operativo è responsabile delle seguenti attività in relazione alla gestione dei processi:

- Creazione ed eliminazione di processi utente e di sistema
- Sospensione e ripresa dei processi
- Fornire meccanismi per la sincronizzazione dei processi
- Fornire meccanismi per la comunicazione di processo
- Fornire meccanismi per la gestione dello stallo

GESTORE DELLA MEMORIA

Per poter essere eseguiti i processi devono risiedere interamente o in parte nella memoria principale.

Deve anche ottimizzare l'utilizzo della CPU e il caching.

E' il gestore della memoria che gestisce quali risorse spostare nella memoria secondaria.

E' il gestore della memoria che alloca e libera le risorse dalla memoria per i processi.

Attività di gestione della memoria

- Tenere traccia di quali parti della memoria sono attualmente utilizzate e da chi
- Decidere quali processi (o parti di essi) e dati spostare dentro e fuori dalla memoria
- Allocazione e deallocazione dello spazio di memoria secondo necessità

GESTORE DEI FILE SYSTEM

Il SO da una visione logica e uniforme della memoria secondaria, si memorizzano byte nelle varie tracce, per il SO però esistono file e directory che è un'astrazione di quello che c'è nel disco.

Il gestore del file system è in grado di controllare chi ha accesso ai vari file.

Il SO include:

- Creazione ed eliminazione dei file.
- Primitive per manipolare file e directory.
- Mapping dei file nella memoria secondaria.
- Garantisce la non volatilità dei file aperti.

GESTORE DELLA MEMORIA DI MASSA

Soltanamente, dischi utilizzati per memorizzare dati che non rientrano nella memoria principale o dati che devono essere conservati per un "lungo" periodo di tempo

Una corretta gestione è di fondamentale importanza

L'intera velocità di funzionamento del computer dipende dal sottosistema del disco e dai suoi algoritmi

Attività del sistema operativo:

- Montaggio e smontaggio
- Gestione dello spazio libero
- Allocazione dello spazio di archiviazione
- Programmazione del disco
- Partizionamento
- Protezione Attività del SO:

CACHING

E' importante per rendere il sistema più performante possibile.

E' implementato sia a livello hardware che software.

La politica è spostare dalla memoria più lenta a una più veloce per i file usati frequentemente.

MIGRAZIONE DEI DATI DA UN DISCO A UN REGISTRO

Il problema di fondo è che in un ambiente multitasking può esistere il caso più di un processo utilizza lo stesso dato.

In presenza di più processi va garantita la consistenza del dato in tutte le varie copie.

In ambiente multiprocessore, la coerenza della cache si ottiene garantendo che l'aggiornamento di un dato in una qualsiasi cache si rifletta immediatamente in tutte le cache in cui il dato in questione risiede.

Le cose si complicano in ambienti distribuiti, non lo vedremo noi, nel libro è nel capitolo 19.

E' il SO che gestisce questa cosa con il gestore della memoria.

SISTEMA CHE SI OCCUPA DELLE OPERAZIONE DI I/O (SOTTOSISTEMA I/O)

Una delle caratteristiche del SO è quella di nascondere all'utente tutta la complessità che ci sta dietro, di tutto l'hardware che deve gestire.

La componente che si occupa dei dispositivi di I/O maschera una serie di operazioni che includono la gestione della memoria e dei vari dispositivi di I/O includendo le operazioni di **buffering**.

Il **buffering** è una operazione complementare al caching, nel senso che memorizzo un pò di dati e aspetto l'istante più opportuno per trasferirli, caching e lo spooling, ovvero le richieste simultanee di più processi di operazioni di I/O verso lo stesso dispositivo.

Ha a disposizione diversi driver per specifici dispositivi hardware.

PROTEZIONE E SICUREZZA

- **Protezione** → Serie di meccanismo e controlli con cui il SO concede o protegge l'accesso alle risorse a processi o utenti.
- **Sicurezza** → Meccanismo con cui il SO si difende da attacchi interni ed esterni (malware, DOS, worms, ...), furto d'identità

PROTEZIONE

In generale il SO distingue gli utenti in base ai propri privilegi che accedono tramite **ID** e password.

L'user ID viene associato con tutti i file, processi e quegli utenti a cui hanno l'accesso a quel file.

Ci sono anche gli **identificativi** per i **gruppi**.

I gruppi sono una categoria di utenti per associare **privilegi** a tutti i membri appartenenti a quel gruppo.

VIRTUALIZZAZIONE

Consente di eseguire applicazioni all'interno di altri sistemi operativi.

Emulazione: Pratica con cui si utilizzano programmi all'interno di architetture di CPU diverse (es. PowerPC nell'intel x86).

Virtualizzazione: Applicativo che gira sul SO che ti consente a sua volta di installarci sopra un altro sistema operativo.

Per esempio: VMware emula Windows XP e le sue applicazioni che ci sono al suo interno.

Qual'è l'utilità della virtualizzazione?

- Provare altri SO, come Mac OS x su windows o viceversa.
- Sviluppare e testare altri applicativi senza il bisogno di procurarsi altri PC con quei SO reali.

SISTEMI DISTRIBUITI

E' legato al discorso della virtualizzazione, per esempio creo 3 macchine virtualizzate.

Consiste in sistemi operativi diversi tramite macchine diverse e comunicano tramite la rete.

Sono comunicati tra loro tramite la **rete**, il protocollo più usato è **TCP/IP** :

- **Local Area Network (LAN)**
- **Wide Area Network (WAN)**
- **Metropolitan Area Network (MAN)**
- **Persona Area Network (PAN)**

E' il **sistema operativo di rete** che gestisce in maniera trasparente di gestire il tutto.

Il sistema operativo di rete fornisce funzionalità tra i sistemi attraverso la rete

- Lo schema di comunicazione consente ai sistemi di scambiare messaggi
- Illusione di un unico sistema

ARCHITETTURA SISTEMI INFORMATICI

Molti dei sistemi utilizzavano (almeno fino a pochi anni fa) un singolo processore general-purpose (cioè che facevano operazioni diverse).

Si è poi passati a sistemi con più processori, **multiprocessori**, noti anche come processori paralleli.

Posso fare più **operazioni contemporaneamente e si migliorano le prestazioni**.

Si dividono in :

- 1) **Multiprocessore asimmetrico** → Se c'è un singolo processore che coordina tutti gli altri processori.
- 2) **Multiprocessore simmetrico** → Se tutti i processori sono allo stesso livello.

Dual core :

Singolo processore fisico con al suo interno due CPU, hanno i registri autonomi, la cache di livello 1 autonoma e condividono la cache di livello 2 e la memoria principale.

ARCHITETTURA NUMA - Non Uniform Access Memory → Cioè ci sono CPU che non condividono la memoria ma ognuno ha la memoria sua. (Es. Solo la CPU 1 può accedere alla memoria 1).

SISTEMI CLUSTER

Ho sempre a disposizione due processori diversi che risiedono in sistemi distinti e anche in questo caso collegati tramite una rete.

Condividono uno spazio di memoria tramite lo **storage area network (SAN)**. Da la possibilità di avere più risorse e protetti da anomalie e risorse pronte all'uso.

Anche qui abbiamo :

- **Cluster asimmetrici:** Quando c'è un nodo che si fa carico del bilanciamento del degli altri nodi del cluster.
- **Cluster simmetrici:** Quando tutti i nodi del cluster sono equivalenti, cioè si autogovernano per bilanciarsi tra loro.

I sistemi cluster vengono realizzati per scopi più disparati.

Vengono utilizzati per l'**HPC (High performance computing)**.

Scheda madre contiente : Socket del processore, slot della ram, tutti i vari connettori per i dispositivi di I/O.

AMBIENTI DEI SISTEMI INFORMATICI

Ci sono ambienti:

- Tradizionali
- Mobile
- Client-Server
- Peer-to-Peer
- Cloud Computing
- Real-time embedded → Sistemi che portano a termine i compiti

TRADIZIONALI

Per esempio macchina general-purpose

Quella serie di calcolatori connessi tramite una LAN/WAN o wireless

MOBILE COMPUTING

Smartphone, tablet, ecc...

Non è così netta la differenza con il sistema "tradizionale".

Ha caratteristiche extra (GPS, giroscopio, ecc...)

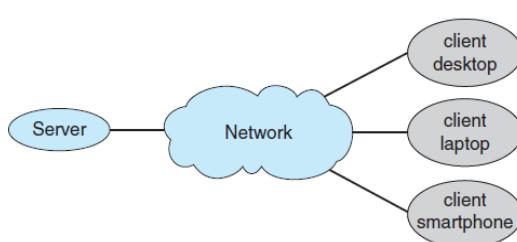
Ha nuovi tipi di applicazioni come per esempio la realtà aumentata.

Usa l'architettura Wi-fi IEEE 802.11 wireless

I leader sono **Apple iOS** e **Google Android**.

SISTEMI Client-Server

E' una architettura dove c'è un **server** che mette a disposizione le risorse nella rete e ci sono i **client** che si collegano al server per recuperare le informazioni da quel server.



Peer-to-Peer

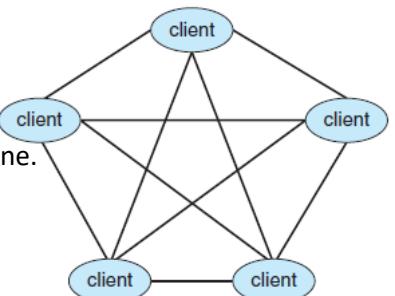
Architettura connessi tramite client, sistemi distribuiti, in questo caso non c'è una distinzione tra client e server, il singolo nodo ricopre entrambi i ruoli, cioè fornire e chiedere servizi, come emule, bit torrent, dove io posso fare sia l'upload per esempio di un film o il download di un film caricato da un altro.

Ci sono 2 distinzioni:

- 1) C'è un nodo centrale che fornisce un registro delle risorse offerte.
- 2) Quando uno dei nuovi client si registra e informa quali sono le risorse a disposizione.

Ogni nodo avrà un registro dei servizi messi a disposizione.

Napster e Gnutella sono Voice over IP (VOIP) come Skype.



CLOUD Computing

In questo caso sono servizi offerti alla rete, come Gmail, Google Drive, OneDrive.

C'è Amazon **EC2** che mette a disposizione risorse in rete con funzionalità diverse, come la memoria, ecc...

Dropbox appena è nato si appoggia a server di amazon.

Ci sono 3 tipologie:

- 1) **Cloud pubblico** → Disponibile via internet per tutti.
- 2) **Cloud privato** → Es. Servizio di un'azienda disponibile solo tramite i dipendenti all'interno della azienda o fuori tramite VPN.
- 3) **Cloud ibrido** → Ha funzioni limitate gratuite che una volta pagato si ha più funzionalità (Come google drive che gratuitamente ha solo 15 GB a disposizione).

Altre distinzioni:

- Software as a Service (**SaaS**) → Applicazioni disponibili attraverso la rete (Es. Blocco note online)
- Platform as a Service (**PaaS**) → Mi collego tramite la rete ad esempio ad un servizio di database in rete.
- Infrastrucure as a Service (**IaaS**) → Mi collego per esempio ad un nodo hpc di una rete.

Cloud pubblico e privato possono coesistere.

Sistemi real time Embedded

Sono sistemi per rispondere a task molto specifici, devono rispondere ad un intervallo regolare.

Per esempio, l'auto a guida autonoma.

SISTEMI OPERATIVI OPEN SOURCE E FREE

Sistema operativo Open source → Hai accesso al codice sorgente al sistema operativo.

Hai la possibilità sia di virtualizzare che vedere come sono implementate certe soluzioni.

STRUTTURE DATI DEI Kernel

Torneranno utili quando faremo lo scheduling.

Le strutture sono la lista singola e la lista circolare

Un'altra struttura dati è l'albero binario i nodi sinistra sono sempre \leq di quello di destra, la ricerca ha il costo $O(n)$

Le funzioni hash sono usati, che hanno un numero che è la chiave che riconduce a un valore.

--- CAPITOLO 2: STRUTTURA DEL SISTEMA OPERATIVO ---

SERVIZI DEL SISTEMA OPERATIVO

Vedremo i servizi del SO, l'interfaccia del SO, **system** call → Meccanismo in cui riescono a chiedere servizi a basso livello, servizi offerti dal SO.

Fornisce una serie di servizi di diversa natura che offrono all'utente funzionalità avanzate per facilitare la sua interazione col SO.

- 1) Uno dei servizi che fornisce per interagire con il SO è **l'interfaccia**. Abbiamo l'interfaccia grafica (GUI) e a linea di comando (CLI).
- 2) **Esecuzione di programmi** → capacità di caricare un programma in memoria ed eseguirlo, eventualmente rilevando, ed opportunamente gestendo, situazioni di errore.
- 3) **Operazioni di I/O** → il SO fornisce ai programmi utente i mezzi per effettuare l'I/O su file o periferica
- 4) **Gestione del file system** → capacità dei programmi di creare, leggere, scrivere e cancellare file e muoversi nella struttura delle directory
- 5) **Comunicazioni** → scambio di informazioni fra processi in esecuzione sullo stesso elaboratore o su sistemi indipendenti, connessi via rete. Le comunicazioni possono avvenire utilizzando memoria condivisa o con scambio di messaggi
- 6) **Rilevamento di errori** → il SO deve tenere il sistema di calcolo sotto controllo costante, per rilevare errori, che possono verificarsi nella CPU e nella memoria, nei dispositivi di I/O o durante l'esecuzione di programmi utente.
 - Per ciascun tipo di errore, il SO deve prendere le opportune precauzioni per mantenere una modalità operativa corretta e consistente
 - I servizi di debugging possono facilitare notevolmente la programmazione e, in generale, l'interazione con il sistema di calcolo

Esistono funzioni addizionali atte ad assicurare l'efficienza del sistema (non esplicitamente orientate all'utente) :

- **Allocazione di risorse** → quando più utenti o più processi vengono serviti in concorrenza, le risorse disponibili devono essere allocate equamente ad ognuno di essi.
- **Accounting e contabilizzazione dell'uso delle risorse** → tener traccia di quali utenti usano quali e quante risorse del sistema (utile per ottimizzare le prestazioni del sistema di calcolo).
- **Protezione e sicurezza** → i possessori di informazioni memorizzate in un sistema multiutente o distribuito devono essere garantiti da accessi indesiderati ai propri dati processi concorrenti non devono interferire fra loro:
 - **Protezione:** assicurare che tutti gli accessi alle risorse di sistema siano controllati
 - **Sicurezza:** si basa sull'obbligo di identificazione tramite password e si estende alla difesa dei dispositivi di I/O esterni (adattori di rete, etc) da accessi illegali

INTERFACCIA UTENTE CLI

L'interfaccia utente a linea di comando permette di impartire direttamente comandi al SO (istruzioni di controllo)

- Talvolta viene implementata nel kernel, altrimenti attraverso programmi di sistema (UNIX/Linux)
- Può essere parzialmente personalizzabile, ovvero il SO può offrire più **shell** più ambienti diversi, da cui l'utente può impartire le proprie istruzioni al sistema.
- La sua funzione è quella di interpretare ed eseguire le istruzioni di comando (siano esse istruzioni built-in del SO o nomi di eseguibili utente) → interprete dei comandi

INTERFACCIA UTENTE GUI

Interfaccia user-friendly che realizza la metafora della scrivania (**desktop**):

- Interazione semplice tramite mouse, tastiera, monitor
- Le **icone** rappresentano file, **directory**, programmi, etc
- I diversi tasti del mouse, posizionato su oggetti differenti, provocano diversi tipi di azione (forniscono informazioni sull'oggetto in questione, eseguono funzionitiche dell'oggetto, aprono directory - **folder** o **cartelle** nel gergo GUI)
- Realizzate per la prima volta, all'inizio degli anni 70 dai laboratori di ricerca Xerox PARC di Palo Alto (computer Xerox Alto, 1973).

INTERFACCIA UTENTE GUI - TOUCHSCREEN

I device mobili con touchscreen richiedono nuovi tipi di interfaccia:

- Accesso senza il supporto del mouse (impossibile da usare o poco pratico).
- Azioni ed operazioni di selezione realizzate tramite "gesti" (pressioni e strisciamenti delle dita).
- Tastiera virtuale per l'immissione di testo
- Comandi vocali

CHIAMATE DI SISTEMA

Le chiamate al sistema forniscono l'interfaccia fra i processi e i servizi offerti dal SO.

- Sono realizzate utilizzando linguaggi di alto livello (C o C++)
Normalmente, vengono richiamate dagli applicativi attraverso **API (Application Programming Interface)**, piuttosto che per invocazione diretta.
- Alcune API molto diffuse sono la Win 64 API per Windows, la POSIX API per i sistemi POSIX-based (tutte le versioni di UNIX, Linux, e Mac OS), e la Java API per la Java Virtual Machine (JVM)

IMPLEMENTAZIONI CHIAMATE DI SISTEMA

Normalmente, a ciascuna system call è associato un numero

- L' **interfaccia alle chiamate di sistema** mantiene una tabella indicizzata dal numero di system call, effettua la chiamata e ritorna al chiamante lo stato del sistema dopo l'esecuzione (ed eventuali valori restituiti)

L'utente non deve conoscere i dettagli implementativi delle system call deve conoscere la modalità di utilizzo dell'API (ed eventualmente il compito svolto dalle chiamate di sistema)

- L'intermediazione della API garantisce la portabilità delle applicazioni
- Molto spesso una system call viene chiamata tramite una funzione di funzione di libreria standard (ad esempio contenuta in stdlibc).

PASSAGGIO DI PARAMETRI ALLE SYSTEM CALL

Spesso l'informazione necessaria alla chiamata di sistema non si limita al solo nome (o numero di identificazione)

- Il tipo e la quantità di informazione varia per chiamate diverse e diversi sistemi operativi

Esistono tre metodi generali per passare parametri al SO:

- Il più semplice passaggio di parametri nei registri → Talvolta, possono essere necessari più parametri dei registri presenti
- Memorizzazione dei parametri in un blocco in memoria e passaggio dell'indirizzo del blocco come parametro in un registro

È un'approccio seguito da Linux (per 5 parametri) e Solaris

- **Push** dei parametri nello stack da parte del programma; il SO recupera i parametri con un **pop**
- Gli ultimi due metodi non pongono limiti al numero ed alla lunghezza dei parametri passati

TIPO DI CHIAMATE DI SISTEMA

- 1) Controllo dei processi
 - Creazione e terminazione di un processo fork exit
 - Caricamento ed esecuzione exec execve
 - Lettura/modifica degli attributi di un processo (priorità, tempo massimo di esecuzione - get/set process attributes)
 - Attesa per il tempo indicato o fino alla segnalazione di un evento wait waitpid
 - Assegnazione e rilascio di memoria alloc free
 - Invio di segnali signal kill
 - Dump della mappa di memoria in caso di errore
 - **Debugger** ed esecuzione a passo singolo
 - Gestione di **lock** per l'accesso a memoria condivisa
- 2) Gestione dei file
 - Creazione e cancellazione di file (create, delete)
 - Apertura e chiusura di file (open, close)
 - Lettura, scrittura e posizionamento (read, write, seek)
 - Lettura/modifica degli attributi di un file (nome, tipo, codici di protezione, informazioni di contabilizzazione - get/set file attributes)
- 3) Gestione dei dispositivi di I/O
 - Richiesta e rilascio di un dispositivo (request, release)
 - Lettura, scrittura e posizionamento (read, write, seek)
 - Lettura/modifica degli attributi di un dispositivo (ioctl)
 - Connessione/disconnessione logica dei dispositivi
- 4) Gestione delle informazioni
 - Lettura/modifica dell'ora e della data (time, date)
 - Informazioni sul sistema
 - Lettura/modifica degli attributi di processi, file e dispositivi (ps, getpid)
- 5) Comunicazione
 - Apertura e chiusura di una connessione (open connection, close connection, pipe)
 - Invio e ricezione di messaggi (send, receive)
 - Informazioni sullo stato dei trasferimenti
 - Inserimento ed esclusione di dispositivi remoti
 - Condivisione della memoria (shm_open, shmget, mmap)
- 6) Protezione
 - Controllo di accesso alle risorse
 - Lettura/modifica dei permessi di accesso (chown, chmod)
 - Accreditamento degli utenti

PROGRAMMI DI SISTEMA

I programmi di sistema forniscono un ambiente conveniente per lo sviluppo e l'esecuzione di programmi utente.
Esistono programmi di sistema per :

- Gestione di file
- Informazioni di stato
- Editing di file
- Supporto a linguaggi di programmazione
- Caricamento ed esecuzione di programmi
- Comunicazioni
- Supporto alla realizzazione di applicativi

L'aspetto del SO per la maggioranza degli utenti è definito dai programmi di sistema, non dalle chiamate di sistema vere e proprie.

1) **Gestione di file** → per creare, cancellare, copiare, rinominare, listare, stampare e, genericamente, gestire le operazioni su file e directory.

2) **Informazioni di stato:**

- Per ottenere dal sistema informazioni tipo data, spazio di memoria disponibile, spazio disco, numero di utenti abilitati.
- Per ottenere informazioni sulle statistiche di utilizzo del sistema di calcolo (prestazioni, logging, etc.) e sul debugging.
- Per effettuare operazioni di formattazione e stampa dei dati
- Per ottenere informazioni sulla configurazione del sistema (registry)

3) **Modifica di file:**

- Editori di testo, per creare e modificare file
- Comandi speciali per cercare informazioni all'interno di file o effettuare trasformazioni sul testo.

4) **Supporto a linguaggi di programmazione** → assembler, compilatori e interpreti.

5) **Caricamento ed esecuzione di programmi** → linker, loader, debugger per linguaggio macchina e linguaggi di alto livello.

6) **Comunicazioni** → per creare connessioni virtuali tra processi, utenti e sistemi di elaborazione: Permettono agli utenti lo scambio di messaggi video e via e-mail, la navigazione in Internet, il login remoto ed il trasferimento di file

7) **Servizi background:**

Lanciati durante la fase di boot:

- Taluni utili solo nella fase di startup del sistema
- Altri in esecuzione dal boot allo shutdown
- Supportano servizi quali controllo del disco, scheduling dei processi, logging degli errori, stampa, connessioni di rete
- Vengono eseguiti in modalità utente
- Noti anche come sottosistemi daemon

8) **Programmi applicativi**

- Non fanno parte del sistema operativo
- Eseguiti dagli utenti
- Lanciati da linea di comando, dal click del mouse o da pressione sul touchscreen

LINKER E LOADERS

Codice sorgente compilato in file oggetto progettati per essere caricati in qualsiasi posizione di memoria fisica - **file oggetto rilocabili**:

- Combinati dal **linker** in un singolo file eseguibile binario in cui vengono incluse anche le funzioni di libreria
- I programmi risiedono quindi nella memoria secondaria come **eseguibili** binari
- Devono essere caricati in memoria centrale dal **loader** per **essere eseguiti**
- Durante la fase di **rilocazione** si assegnano gli indirizzi assoluti alle parti del programma e ai dati

I moderni sistemi general purpose non collegano le librerie ai file eseguibili:

Piuttosto, le librerie collegate dinamicamente (e.g., le DLL di Windows) vengono caricate secondo necessità e condivise da tutti i programmi che utilizzano la stessa versione della stessa libreria (presente in memoria in un'unica copia)

- I file oggetto e eseguibili hanno formati standard, perché il sistema operativo sappia come caricarli e avviarli
- Nei sistemi UNIX-like, il formato standard è l'**ELF** per Executable and Linkable Format → Contiene il punto di inizio del programma cioè l'indirizzo della prima istruzione da eseguire
- I sistemi Windows utilizzano il formato PE Portable Executable mentre Mac OS usa file Mach-O

LIBRERIE STATICHE VS LIBRERIE DINAMICHE

- Le librerie statiche resistono alla vulnerabilità perché risiedono all'interno del file eseguibile
- La velocità in fase di esecuzione si verifica più velocemente perché il suo codice oggetto (binario) è in un file eseguibile
- Le modifiche apportate ai file e al programma richiedono il ricollegamento e la ricompilazione
- La dimensione del file è molto più grande

VS

- Più applicazioni in esecuzione utilizzano la stessa libreria senza bisogno di ciascun file con una propria copia
- Tuttavia, cosa succede se la libreria dinamica viene danneggiata? Il file eseguibile potrebbe non funzionare perché vive al di fuori dell'eseguibile ed è vulnerabile alla rottura
- Contengono file più piccoli
- Le librerie dinamiche sono collegate in fase di esecuzione. Non richiede ricompilazione e ricollegare quando il programmatore apporta una modifica.

PERCHE' LE APPLICAZIONI SONO SO-SPECIFIC?

Le applicazioni compilate su un sistema di solito non sono eseguibili su altri sistemi operativi

Ogni sistema operativo offre le proprie chiamate di sistema uniche, i propri formati di file eseguibili, etc.

Le applicazioni possono essere multi-SO se:

- scritte in un linguaggio interpretato come Python, coninterprete disponibile su più sistemi operativi
- scritte in un linguaggio (come Java) che include una VM contenente l'app in esecuzione
- scritte in un linguaggio standard (come C), ma compilate separatamente su ciascun SO

Application Binary Interface ABI è l'equivalente di API, ma definisce in che modo i diversi componenti del codice binario possono interrarsi per un dato SO su una data architettura, CPU, etc

PROGETTAZIONE E REALIZZAZIONE DI SO

Progettare il SO "perfetto" è un compito che non ammette soluzione, ma alcuni approcci implementativi si sono dimostrati comunque validi.

La struttura interna dei diversi SO può variare notevolmente :

- in dipendenza dall'hardware
- e dalle scelte progettuali che, a loro volta, dipendono dallo scopo del sistema operativo e ne influenzano i servizi offerti

Richieste utente ed obiettivi del SO:

- **Richieste utente** → il SO deve essere di semplice utilizzo, facile da imparare, affidabile, sicuro e veloce
- **Obiettivi del sistema** → il SO deve essere semplice da progettare, facile da realizzare e manutenere, flessibile, affidabile, error-free ed efficiente
- Progettare e realizzare un SO è un compito di **ingegneria del software** eminentemente creativo

Per la progettazione e la realizzazione di un sistema operativo è fondamentale mantenere separati i due

concetti di:

- **Politiche** → Quali sono i compiti e i servizi che il SO dovrà svolgere/fornire?
(Es scelta di un particolare algoritmo per lo scheduling della CPU)
- **Meccanismi** → Come realizzarli? (Es timer)
I meccanismi determinano "come fare qualcosa", le politiche definiscono "cosa è necessario fare".

La separazione della politica dal meccanismo è un principio molto importante, consente la massima flessibilità se le decisioni politiche devono essere modificate in seguito.

SVILUPPO/IMPLEMENTAZIONE SISTEMA OPERATIVO

Tradizionalmente i SO venivano scritti in linguaggio assembly successivamente, furono utilizzati linguaggi specifici per la programmazione di sistema, quali Algol e PL/1; attualmente vengono invece sviluppati in linguaggi di alto livello, particolarmente orientati al sistema C o C++.

In realtà, normalmente, si utilizza un mix di linguaggi :

- Componenti di basso livello sviluppate in assembly (es driver dei dispositivi)
- Kernel in C
- Programmi di sistema realizzati tramite, C, C++ e linguaggi di scripting, quali PERL Python shell script

Caratteristiche:

- Veloci da codificare codice compatto, di facile comprensione, messa a punto e manutenzione
- Portabilità
- Potenziale minor efficienza del codice C rispetto all'assembly

STRUTTURA DEL SISTEMA OPERATIVO

Il sistema operativo generico è un programma molto grande.

Ci sono vari modi per strutturarli :

- Struttura semplice - MS-DOS
- Più complesso - UNIX
- A strati: un'astrazione
- Microkernel - Mach

STRUTTURA MONOLITICA - UNIX ORIGINALE

UNIX - limitato dalla funzionalità hardware, l'originale sistema operativo UNIX aveva una strutturazione limitata.

Il sistema operativo UNIX è costituito da due parti separabili

- Programmi di sistema
- Il kernel:
 - Consiste in tutto ciò che si trova al di sotto dell'interfaccia di chiamata di sistema e sopra l'hardware fisico
 - Fornisce il file system, lo scheduling della CPU, la gestione della memoria e altre funzioni del sistema operativo; un grande numero di funzioni per un livello!

APPROCCIO STRATIFICATO

- 1) Il SO è suddiviso in un certo numero di strati (ciascuno costruito sopra gli strati inferiori).
Il livello più basso (strato 0) è l'hardware, il più alto (strato N) è l'interfaccia utente
- 2) L'architettura degli strati è tale che ciascuno strato impiega esclusivamente funzioni (e servizi di strati di livello inferiore (usati come black-box). Ogni strato è un "oggetto astratto", che incapsula i dati e le operazioni che trattano tali dati.

MICROKERNEL

Quasi tutte le funzionalità del kernel sono spostate nello spazio utente

Mach è un esempio di **microkernel** → Kernel di Mac OS X (Darwin) basato in parte su Mach.

Un microkernel offre i servizi minimi di gestione dei processi, gestione della memoria e di comunicazione

Scopo principale → fornire funzioni di comunicazione fra programmi client e servizi (implementati esternamente).

Le comunicazioni hanno luogo tra moduli utente mediante scambio di messaggi (mediati dal kernel)

Vantaggi:

- Funzionalità del sistema più semplici da estendere i nuovi servizi sono programmi di sistema che si eseguono nello spazio utente e non comportano modifiche al kernel

- Facilità di modifica del kernel
- Sistema più facile da portare su nuove architetture
- Più sicuro e affidabile (meno codice viene eseguito in modo kernel)

Svantaggi:

- Possibile decadimento delle prestazioni a causa dell'overhead di comunicazione fra spazio utente e spazio kernel

SISTEMI IBRIDI

La maggior parte dei SO attuali non adotta un modello "puro"

- I modelli ibridi combinano diversi approcci implementativi allo scopo di migliorare le performance, la sicurezza e l'usabilità
- I kernel di Linux e Solaris sono fondamentalmente monolitici, perché mantenere il SO in un unico spazio di indirizzamento garantisce prestazioni migliori sono però anche modulari, per cui le nuove funzionalità possono essere aggiunte dinamicamente al kernel
- Windows è perlopiù monolitico, ma conserva alcuni comportamenti tipici dei sistemi microkernel, tra cui il supporto per sottosistemi separati (detti **personalità** che vengono eseguiti come processi in modalità utente)

iOS

Anche se progettati per hardware diversi, **Mac OS** e **iOS** hanno architettura simile

- Strato dell'interfaccia utente user experience Aqua per Mac OS, progettata per interazione con mouse e trackpad, Springboard per iOS, per touch screen.
- Strato degli ambienti applicativi Cocoa per Mac OS, Cocoa Touch per iOS, forniscono un'API per i linguaggi di programmazione Objective-C e Swift.
- Ambienti di base (supportano grafica e contenuti multimediali, inclusi Quicktime e OpenGL).
- Ambiente kernel Darwin include il microkernel Mach e il kernel BSD UNIX.

ANDROID

Sviluppato dalla Open Handset Alliance guidata da Google ed è open-source

Costituito da una "pila" di strati software (come iOS)

Basato su un kernel Linux modificato (al di fuori delle distribuzioni standard) :

- Gestione dei processi, della memoria, delle periferiche
- Ampliato per includere l'ottimizzazione dei consumi energetici

L'ambiente di runtime include include un insieme di librerie di base e la Dalvik virtual machine

- App sviluppate in Java con il supporto dell'Android API
- Class file di Java compilati in bytecode e quindi tradotti in eseguibili per la Dalvik virtual machine

Le librerie includono ambienti per lo sviluppo di browser webkit di supporto ai database SQLite, ambienti multimediali e una libreria C standard minimale.

GENERAZIONE ED AVVIO DEL SO

I sistemi operativi sono generalmente progettati per funzionare su una classe di architetture, che possono disporre di una grande varietà di periferiche.

Comunemente, il sistema operativo è già installato quando si acquista un computer :

Nulla vieta tuttavia che se ne possano realizzare e installare altri.

Se si genera un sistema operativo from scratch occorre (scrivere il codice sorgente del sistema operativo) :

- 1) Configurare il sistema operativo per il sistema di calcolo su cui verrà eseguito
- 2) Compilare il sistema operativo
- 3) Installare il sistema operativo
- 4) Avviare il computer con il nuovo SO

GENERAZIONE ED AVVIO DI LINUX

- 1) Accedere a <http://www.kernel.org> per scaricare il codice sorgente di Linux
- 2) Configurare il kernel tramite "make menuconfig" si genera il file di configurazione .config
- 3) Compilare il kernel usando "make" (che utilizza i parametri di configurazione presenti in config)
 - Si produce vmlinuz, l'immagine del kernel
 - Compilare i moduli del kernel tramite "make modules"
 - Installare i moduli del kernel in vmlinuz tramite "make modules_install"
 - Installare il nuovo kernel sul sistema tramite "make install"

AVVIO DEL SISTEMA OPERATIVO

- Quando si accende un computer, l'esecuzione inizia da una posizione di memoria prefissata
- Il sistema operativo deve essere reso disponibile all'hardware affinché l'hardware possa avviarlo:
 - Piccolo pezzo di codice - bootstrap loader parte del BIOS memorizzato in ROM o nella EEPROM, individua il kernel, lo carica in memoria e lo avvia
 - A volte il bootstrap avviene in due passi si carica il bootloader contenuto nel blocco di avvio che provvede a caricare il kernel
 - I sistemi moderni sostituiscono il BIOS con l'interfaccia UEFI Unified Extensible Firmware Interface
- I comuni bootstrap, come GRUB di Linux, permettono la selezione del kernel in varie versioni, con differenti opzioni e da dischi diversi
- Il kernel viene caricato e il sistema è quindi in esecuzione
- I loader di avvio spesso consentono vari stati di avvio, ad esempio modalità utente singolo o modalità di ripristino

DEBUGGING DEL SISTEMA OPERATIVO

- Il **debugging** è l'attività di individuazione e risoluzione di errori nel sistema, i cosiddetti **bug**.
- Il SO può generare **file di log** che danno informazioni sugli errori rilevati durante l'esecuzione di un processo.
- Il SO può anche acquisire e memorizzare in un file un'immagine del contenuto l'errore di un'applicazione della memoria utilizzata dal processo, chiamata **core dump**.
- Il SO può anche acquisire e memorizzare in un file un'immagine del contenuto l'errore del sistema operativo della memoria del kernel, chiamata **crash dump**.

PERFORMANCE TUNING

- Il performance tuning è l'insieme delle tecniche atte ad ottimizzare le prestazioni del sistema, eliminando i colli di bottiglia.
- Introduzione, all'interno del SO, di strumenti interattivi che permettano ad amministratore ed utenti di monitorare il sistema.
- A volte utilizzando elenchi di tracce di attività, registrati per l'analisi.
- **Profiling** → campionamento periodico dell'IP (Instruction Pointer) per valutare, per esempio, quali chiamate di sistema siano utilizzate maggiormente

TRACCIAMENTO

- Raccoglie i dati per un evento specifico, come i passaggi coinvolti in una chiamata di sistema

Gli strumenti includono:

- strace - Trace chiamate di sistema invocate da un processo
- gdb - debugger a livello di sorgente
- perf - raccolta di strumenti per le prestazioni di Linux
- tcpdump - raccoglie i pacchetti di rete

LINUX BBC

- Il debugging a livello di interazioni tra codice utente e codice kernel è quasi impossibile senza un set di strumenti integrati al SO
- BCC (BPF Compiler Collection) è un ricco toolkit che fornisce funzionalità di tracciamento per Linux
- Ad esempio, disksnoop.py traccia l'attività di I/O su disco

- BCC include strumenti di monitoraggio di tutti i moduli del sistema

---CAPITOLO 3 – PROCESSI---

DEFINIZIONE DI PROCESSO

Il **processo** è un **programma in esecuzione**, l'esecuzione di un processo deve **avvenire in modo sequenziale**, o, in alternativa, la sequenza di eventi che avvengono in un elaboratore quando opera sotto il controllo di un particolare programma.

Un **processo** include:

- una **sezione di testo** (il **codice del programma** da eseguire)
- una **sezione dati** (**variabili globali**)
- lo **stack** (**dati temporanei** - parametri per i sottoprogrammi, variabili locali e indirizzi di rientro, un record di attivazione)
- uno **heap** (letteralmente "mucchio" grande quantità **memoria dinamicamente allocata** durante l'esecuzione del task)
- il **program counter** ed il **contenuto dei registri della CPU**

Il **programma** è un'entità "**passiva**" memorizzata su disco (**un file eseguibile**), un **processo** è "**attivo**":

I **programmi** divengono **processi** quando vengono caricati nella **memoria principale**.

L'**esecuzione** di un **programma** viene **attivata** dal **doppio click** del mouse in una GUI, o immettendo il nome del file eseguibile da linea di comando.

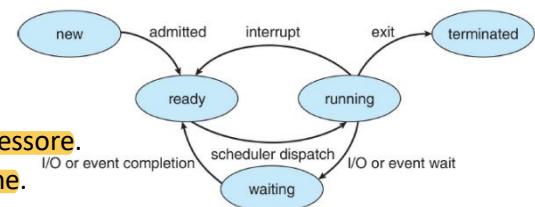
Un programma può corrispondere a diversi processi. Si pensi a un insieme di utenti che utilizzano uno stesso **editor di testo** in relazione a file diversi.

Quindi, **un programma descrive non un processo, ma un insieme di processi - istanze del programma** - ognuno dei quali è relativo all'esecuzione del programma da parte dell'elaboratore per un particolare insieme di dati in ingresso. Un processo può rappresentare un ambiente di esecuzione di altri processi → Java Virtual Machine e programmi scritti in Java.

STATO DEL PROCESSO

Mentre viene eseguito, un processo è soggetto a **transizioni di stato**, definite in parte dall'attività corrente del processo ed in parte da eventi esterni, asincroni rispetto alla sua esecuzione:

- **New** (nuovo): il **processo viene creato**.
- **Running** (in esecuzione): se ne eseguono le **istruzioni**.
- **Waiting** (in attesa): il **processo è in attesa di un evento**.
- **Ready** (pronto): il **processo è in attesa di essere assegnato ad un processore**.
- **Terminated** (terminato): il **processo ha terminato la propria esecuzione**.



PROCESS CONTROL BLOCK

Ad **ogni processo** sono **associate delle informazioni**, il descrittore di processo - **Process Control Block (PCB)**.

Informazione associata a ciascun processo:

- **Stato del processo**.
- **Nome** (del processo).
- **Contesto del processo** **program counter**, **registri della CPU** (accumulatori, registri indice, stack pointer, registri di controllo).
- **Informazioni sullo scheduling della CPU**, (priorità, puntatori alle code di scheduling).
- **Informazioni sulla gestione della memoria allocata al processo** (registri base e limite, tabella delle pagine o dei segmenti).
- **Informazioni di contabilizzazione delle risorse**: utente proprietario, tempo di utilizzo della CPU, tempo trascorso dall'inizio dell'esecuzione, etc...
- **Informazioni sull'I/O**: elenco dispositivi assegnati al processo, file aperti, etc...

THREAD

Alla base del concetto di **thread** sta la constatazione che la **definizione di processo** è basata su due aspetti:

- 1) **Possesso delle risorse**
- 2) **Esecuzione**

I due aspetti sono indipendenti e come tali possono essere gestiti dal SO:

- L'elemento che viene eseguito è il thread
- L'elemento che possiede le risorse è il processo

Il termine **multithreading** è utilizzato per descrivere la situazione in cui ad un **processo** sono associati più **thread**.

SCHEDULING DEL PROCESSO

Meccanismo messo in atto **per massimizzare l'utilizzo della CPU**, che consiste nel **passare rapidamente dal'esecuzione di un processo al successivo, garantendo il time sharing**.

È il modulo del SO noto come **CPU scheduler** che esegue questo compito.

Code per lo scheduling dei processi:

- **Ready queue** → Coda dei processi pronti Insieme dei processi pronti ed in attesa di essere eseguiti, che risiedono in memoria centrale
- **Code di attesa (wait queues)** → Insieme dei processi in attesa per un dispositivo di I/O o per il verificarsi di un evento

CONTEXT SWITCH

Quando la CPU passa da un processo all'altro, il sistema deve salvare il **contesto** del vecchio processo e caricare il **contesto**, precedentemente salvato, per il nuovo processo in esecuzione.

Il **contesto** è rappresentato all'interno del PCB del processo e comprende i valori dei registri della CPU, lo stato del processo e le informazioni relative all'occupazione di memoria :

- Il tempo di context-switch è un sovraccarico (over-head); il sistema non lavora utilmente mentre cambia contesto.
Più sono complicati il SO e, conseguentemente, il PCB, più è lungo il tempo di context-switch.

Il tempo di context-switch (msec) dipende dal **supporto hardware** (velocità di accesso alla memoria, numero di registri da copiare, istruzioni speciali, gruppi di registri multipli).

Nel caso di registri multipli, più contesti possono essere presenti, contemporaneamente, nella CPU.

MULTITASKING PER SISTEMI MOBILI

Alcuni SO per mobile (per es le prime versioni di iOS) prevedevano un solo processo utente in esecuzione (sospensione di tutti gli altri).

A causa dello spazio sullo schermo, i limiti dell'interfaccia utente che iOS prevede a:

- Un unico **processo in foreground**, controllabile mediante GUI.
- **Più processi in background**, in memoria centrale, in esecuzione, ma impossibilitati ad utilizzare il display (quindi con "capacità" limitate)
- Applicazioni eseguibili in background se....
 - realizzano un unico task di lunghezza finita (completamento di un download dalla rete)
 - ricevono notifiche sul verificarsi di un evento (ricezione di email)
 - impongono attività di lunga durata (lettore audio)

Android non pone particolari limiti alle applicazioni eseguite in background

- 1) Un'applicazione in elaborazione in background deve utilizzare un servizio un componente applicativo separato, che viene eseguito per conto della app.
- 2) Il servizio continua a funzionare anche se l'app in background viene sospesa.
- 3) I servizi non hanno un'interfaccia utente e hanno un ingombro di memoria moderato.

Questa è una tecnica efficace per il mobile multitasking

OPERAZIONE SUI PROCESSI

Nella maggior parte dei SO, i processi si possono eseguire in concorrenza, e si devono creare e cancellare dinamicamente. Il SO deve fornire meccanismi per la

- 1) creazione di processi
- 2) terminazione di processi

CREAZIONE DI PROCESSI

Un processo può creare altri processi.

Il processo **padre** crea processi **figli** che, a loro volta, creano altri processi, formando un **albero** di processi.

I processi vengono identificati all'interno del sistema tramite un **pid** (per process identifier).

Condivisione di risorse:

- Il padre e i figli condividono tutte le risorse
- I figli condividono un sottoinsieme delle risorse del padre
- Il padre e i figli non condividono risorse

Esecuzione:

- Il padre e i figli vengono eseguiti in concorrenza
- Il padre attende la terminazione dei processi figli

Spazio degli indirizzi:

- Il processo figlio è un duplicato del processo padre
- Nel processo figlio viene caricato un diverso programma

In Linux/Unix:

- La system call fork() crea un nuovo processo
- Quindi, la exec() viene impiegata dopo la fork() per sostituire lo spazio di memoria del processo con un nuovo programma
- Il genitore attende il completamento del processo figlio con la chiamata di sistema wait()

LA CHIAMATA DI SISTEMA fork()

Dopo aver creato un nuovo processo figlio, entrambi i processi lo faranno eseguire l'istruzione successiva dopo la chiamata di sistema fork().

Quindi bisogna distinguere il genitore dal figlio. Questo può essere fatto testando il valore restituito di fork():

- fork() restituisce uno zero al processo figlio appena creato
- fork() restituisce un valore positivo, l'ID del processo di processo figlio, al genitore. Un processo può utilizzare funzione getpid() per recuperare l'ID del processo assegnato a questo processo.

Il processo figlio eredita i privilegi e gli attributi di scheduling dal genitore, così come certe risorse, quali i file aperti

TERMINAZIONE DI PROCESSI

Un processo termina quando esegue l'ultima istruzione e chiede al sistema operativo di essere cancellato per mezzo di una specifica chiamata di sistema (exit() in UNIX) che compie le seguenti operazioni:

- può restituire dati (output) al processo padre (attraverso la system call wait())
- le risorse del processo vengono deallocate dal sistema operativo
- Con abort() il processo può terminare volontariamente segnalando una terminazione anomala al padre

Il padre può terminare l'esecuzione dei processi figli se:

- se il figlio ha ecceduto nell'uso delle risorse ad esso allocate
- se il compito assegnato al figlio non è più richiesto

- se il padre sta terminando → perché alcuni sistemi operativi non consentono ad un processo figlio di continuare l'esecuzione.

Questo fenomeno è detto **terminazione a cascata** e viene avviato dal SO.

Il processo padre può attendere la terminazione di un figlio invocando la system call `wait()`

La chiamata restituisce informazioni di stato ed il pid del figlio terminato

```
pid = wait(&status);
```

Nel sistema esiste una tabella dei processi:

- Un processo terminato, il cui padre non ha ancora invocato la `wait()` è uno **zombie**
- Dopo la chiamata alla `wait()` il pid del processo **zombie** e la relativa voce nella tabella dei processi vengono rilasciati
- Se il padre termina senza invocare la `wait()` il processo è un **orfano**.

GERARCHIA DEI PROCESSI ANDROID

A causa delle limitate risorse di calcolo, i SO per mobile possono dover terminare dei processi per utilizzare un altro processo:

In Android è definita una **gerarchia dalla più importante alle meno importante** :

- 1) Processo foreground: con cui l'utente sta interagendo direttamente.
- 2) Processo visibile che esegue un'attività a cui il processo foreground fa riferimento.
- 3) Processo di servizio che esegue in background ma svolgendo un'attività evidente per l'utente.
- 4) Processo in background svolge un'attività non evidente per l'utente.
- 5) Processo vuoto non contiene componenti attive associate ad una app.

Android interrompe i processi in base alla gerarchia, ma salva lo stato di un processo prima della sua terminazione forzata, per poterlo ripristinare quando l'utente ritorna all'applicazione.

ARCHITETTURA MULTIPROCESSO - IL BROWSER CHROME

I browser attuali supportano la navigazione a schede, che permette di aprire contemporaneamente, in una singola istanza del browser, più siti web.

Se si blocca un'applicazione web in una qualsiasi scheda, si blocca l'intero processo.

Google Chrome ha un'architettura multiprocesso, con tre diversi tipi di processi:

- 1) **Il browser** che gestisce l'interfaccia utente, l'I/O da disco e da rete
- 2) **I renderer** che contengono la logica per il rendering di pagine web, per la gestione di HTML, Javascript, immagini, etc.:
 - Si crea un renderer per ogni sito aperto
 - Vengono eseguiti in una sandbox con accesso a disco e rete limitati per minimizzare gli effetti negativi di exploit di sicurezza
 - Solo il renderer "crasha" se si verifica un problema in una scheda.
- 3) **I plug-in** che agevolano il browser nell'elaborazione di parti colari contenuti web, come i file Flash o Windows Media.

COMUNICAZIONE TRA PROCESSI

Un processo è **indipendente** se la sua esecuzione non può influire sull'esecuzione di altri processi nel sistema o subirne gli effetti.

I processi **cooperanti** possono, invece, influire sull'esecuzione di altri processi e/o esserne influenzati.

Vantaggi della cooperazione fra processi :

- 1) **Condivisione di informazioni**: ambienti con accesso concorrente a risorse condivise
- 2) **Accelerazione del calcolo**: possibilità di elaborazione parallela (in presenza di più CPU)
- 3) **Modularità**: funzioni distinte che accedono a dati condivisi (es. kernel modulari)
- 4) **Convenienza**: anche per il singolo utente, possibilità di compiere più attività in parallelo

I processi di cooperazione richiedono la comunicazione interprocesso (IPC):

Due modelli di IPC:

- 1) **Memoria condivisa:** massima efficienza nella comunicazione
- 2) **Scambio di messaggi:** utile per trasmettere piccole quantità di dati, nessuna conflittualità, utilizzo di system call per la messaggeria

MODELLO DI COMUNICAZIONE

- 1) **Shared memory** → i processi o i thread condividono dati in memoria e accedono in lettura e scrittura a tali dati condivisi
- 2) **Message passing** → i processi o i thread si scambiano informazioni tramite messaggi (simile a quanto avviene sulla rete)

PROBLEMA DEL PRODUTTORE-CONSUMATORE

È un paradigma classico per processi cooperanti il processo produttore produce informazioni che vengono consumate da un processo consumatore.

L'informazione viene passata dal **produttore** al **consumatore** attraverso un buffer.

Utile metafora del meccanismo client-server.

- 1) **Buffer illimitato** non vengono posti limiti pratici alla dimensione del buffer:
 - Il consumatore può trovarsi ad attendere nuovi oggetti, ma il produttore può sempre produrne
- 2) **Buffer limitato** si assume che la dimensione del buffer sia fissata:
 - Il consumatore attende se il buffer è vuoto, il produttore se è pieno.

SISTEMI A MEMORIA CONDIVISA

Memoria condivisa: inizialmente, parte dello spazio di indirizzi del processo che la alloca.

I processi cooperanti - che la usano per comunicare - dovranno annettere la zona di memoria al loro spazio di indirizzi.

La gestione della memoria condivisa, una volta allocata, non dipende dal SO :

- Il tipo e la collocazione dei dati sono determinati dai processi
- che hanno anche la responsabilità di non scrivere simultaneamente nella stessa locazione

Fornire opportuni meccanismi per la sincronizzazione dei processi nell'accesso a dati condivisi.

I processi cooperanti devono sincronizzare le loro azioni per garantire la coerenza dei dati

BUFFER LIMITATO E MEMORIA CONDIVISA

Dati condivisi:

```
#define BUFFER_SIZE 10

typedef struct {

    ... ...
}

} elemento;

elemento buffer[BUFFER_SIZE];

int in = 0;

int out = 0;
```

La soluzione ottenuta è corretta, ma consente l'utilizzo di soli BUFFER_SIZE-1 elementi

MEMORIA CONDIVISA

Di solito, la comunicazione di processo interconnessa viene eseguita utilizzando **pipe** o **named pipe**.

I processi non correlati (diciamo un processo in esecuzione in un terminale e un altro processo in un altro

terminale) può essere la comunicazione eseguita utilizzando Named Pipes o attraverso le tecniche IPC popolari di memoria condivisa

- `#include <sys/ipc.h>`
- `#include <sys/shm.h>`
- Creare il segmento di memoria condivisa o utilizzarne uno già creato segmento di memoria condivisa `shmget()`
- Allegare il processo alla memoria condivisa già creata segmento `shmat()`
- Scollegare il processo dalla memoria condivisa già collegata segmento `shdet()`
- Operazioni di controllo sul segmento di memoria condivisa `shmctl()`

IPC E SCAMBIO DI MESSAGGI

I processi comunicano fra loro senza far uso di variabili condivise.

La funzionalità IPC consente due operazioni:

- 1) `send(messaggio)` - la dimensione del messaggio può essere fissa o variabile
- 2) `receive(messaggio)`

La dimensione del messaggio è fissa o variabile.

Se i processi P e Q vogliono comunicare, devono:

- stabilire fra loro un canale di comunicazione
- scambiare messaggi per mezzo di send/receive

Problemi di implementazione:

- Come vengono stabiliti i canali (connessioni)?
- È possibile associare un canale a più di due processi?
- Quanti canali possono essere stabiliti fra ciascuna coppia di processi comunicanti?
- Qual è la capacità di un canale?
- Il formato del messaggio che un canale può gestire è fisso o variabile?
- Sono preferibili canali monodirezionali o bidirezionali?

Implementazione del canale di comunicazione:

- fisica → memoria condivisa, bus hardware, rete
- logica → comunicazione diretta o indiretta, sincrona o asincrona, con bufferizzazione assente o automatica

COMUNICAZIONE DIRETTA

I processi devono "nominare" esplicitamente i loro interlocutori (modalità simmetrica):

- `send (P, messaggio)` - invia un messaggio al processo P
- `receive (Q, messaggio)` - riceve un messaggio dal processo Q

Proprietà del canale di comunicazione:

- I canali vengono stabiliti automaticamente
- Ciascun canale è associato esattamente ad una coppia di processi
- Tra ogni coppia di processi comunicanti esiste esattamente un canale
- Il canale può essere unidirezionale (ogni processo collegato al canale può soltanto trasmettere/ricevere), ma è normalmente bidirezionale

COMUNICAZIONE INDIRETTA

Per ovviare ai difetti della nominazione diretta si usa la nominazione indiretta basata su porte o mailbox.

I messaggi vengono inviati/ricevuti a/di mailbox o porte:

- Ciascuna mailbox è identificata con un id unico
- I processi possono comunicare solo se condividono una mailbox

Proprietà dei canali di comunicazione:

- Un canale viene stabilito solo se i processi hanno una mailbox in comune
- Un canale può essere associato a più processi
- Ogni coppia di processi può condividere più canali di comunicazione
- I canali possono essere unidirezionali o bidirezionali

Operazioni:

- Creare una nuova mailbox
- Inviare/ricevere messaggi attraverso mailbox
- Rimuovere una mailbox

Primitive di comunicazione:

- **send(A, messaggio)** → invia un messaggio alla mailbox A
- **receive(A, messaggio)** → riceve un messaggio dalla mailbox A

Condivisione di mailbox:

- P1, P2 e P3 condividono la mailbox A
- P1 invia P2 e P3 ricevono
- Chi si assicura il messaggio?

Soluzioni:

- Permettere ad un canale di essere associato ad al più di due processi
- Permettere ad un solo processo alla volta di eseguire un'operazione di ricezione
- Permettere al SO di selezionare arbitrariamente il ricevente o in base ad uno scheduling (es round robin) il sistema ne comunica l'identità al mittente

SINCRONIZZAZIONE

Lo scambio di messaggi può essere sia bloccante che non-bloccante.

In caso di scambio di messaggi **bloccante**, la comunicazione è **sincrona**:

- **Blocking send** - Il processo mittente si blocca nell'attesa che il processo ricevente, o la porta, riceva il messaggio
- **Blocking receive** - Il ricevente si blocca nell'attesa dell'arrivo di un messaggio

In caso di scambio di messaggi **non-bloccante** la comunicazione è **asincrona**:

- **Non-blocking send** - Il processo mittente invia il messaggio e riprende la propria esecuzione
- **Non-blocking receive** - Il ricevente riceve un messaggio valido o un valore nullo

Diverse combinazioni possibili:

- Se sia l'invio che la ricezione stanno bloccando, abbiamo un **Rendezvous**

BUFFERING

La coda dei messaggi legata ad un canale può essere implementata in tre modi:

- 1) Capacità zero → Il canale non può avere messaggi in attesa al suo interno; il trasmittente deve attendere che il ricevente abbia ricevuto il messaggio (rendezvous)
- 2) Capacità limitata → Il canale può avere al più n messaggi in attesa; se il canale è pieno, il trasmittente deve attendere
- 3) Capacità illimitata → Il canale può "contenere" infiniti messaggi il trasmittente non attende mai.

SISTEMI PER IPC

- UNIX utilizza lo standard POSIX, che prevede **IPC** sia tramite memoria condivisa che scambio di messaggi
- Mach implementa la comunicazione tramite scambio di messaggi:

- I messaggi si inviano/ricevono tramite porte, create tramite `mach_port_allocate()`
- Anche le chiamate di sistema si realizzano tramite messaggi
- Quando si crea un nuovo task si creano anche due porte speciali Kernel e Notify
- I messaggi vengono ricevuti ed inviati tramite la funzione `mach_msg()`
- Invia e ricevi sono flessibili, ad esempio quattro opzioni se la casella di posta è piena:
 - 1) Aspetta a tempo indeterminato
 - 2) Attendere al massimo n millisecondi
 - 3) Restituisce immediatamente
 - 4) Memorizza temporaneamente nella cache un messaggio

- Windows usa la memoria condivisa come meccanismo per supportare alcune forme di message passing

PIPE

Una pipe o una **named pipe** agisce come un canale di comunicazione fra processi.

Problemi :

- Come implementarla?
- La comunicazione permessa dalla pipe è unidirezionale o bidirezionale?
Se è ammessa la comunicazione a doppio senso, essa è di tipo half-duplex o full-duplex ?
- Deve esistere una relazione (tipo **padre-figlio**) tra i processi in comunicazione?
- Le pipe possono comunicare in rete o i processi comunicanti devono risiedere sulla stessa macchina?

Pipe convenzionali (ordinary pipes):

- Non possono essere accedute se non dal processo che le ha create.
- Esistono solo per la durata del tempo di comunicazione.
- Permettono ai processi di comunicare seguendo il paradigma del produttore-consumatore
- Sono unidirezionali se viene richiesta la comunicazione a doppio senso devono essere utilizzate due pipe
- Nel gergo di Windows, sono dette pipe anonime

Named pipe:

- Possono essere utilizzate da processi qualunque (non necessariamente in relazione padre-figlio)
- La comunicazione può essere bidirezionale e la pipe continua ad esistere anche quando i processi comunicanti sono terminati
- Più processi possono usarla per comunicare
- Disponibili sia in Windows che in UNIX

COMUNICAZIONI IN SISTEMI CLIENT-SERVER

- Socket
- Chiamate a procedura remota (Remote Procedure Call - RPC)

SOCKET

- Una **socket** è definita come l'estremità di un canale di comunicazione.
- Ogni socket è identificata da un indirizzo IP concatenato ad un numero di **porta** → la porta è un numero incluso all'inizio del pacchetto di messaggi a differenziare i servizi di rete su un host
- Esempio: la socket **161.25.19.8:1625** si riferisce alla porta **1625** sull'host **161.25.19.8**
- La comunicazione si stabilisce fra coppie di socket (una per ogni processo → tutte le connessioni devono essere uniche)
- Tutte le porte inferiori a 1024 sono **ben note**, utilizzate per servizi standard:
 - 80: Hypertext Transfer Protocol (HTTP)
 - 110: Post Office Protocol (POP3)
 - 143: Internet Message Access Protocol (IMAP)
 - 156: Structured Query Language (SQL)
 - 443: Hypertext Transfer Protocol Secure (HTTPS)
- L'indirizzo IP speciale 127.0.0.1 (**loopback**) fa riferimento al sistema su cui è in esecuzione il processo.

CHIAMATE A PROCEDURA REMOTA

Il concetto di chiamata a procedura remota estende il paradigma della chiamata di procedura a processi residenti su sistemi remoti collegati in rete:

- Gestite tramite porte che offrono servizi diversi
- **Stub** sia lato client che lato server che consentono la comunicazione e strutturano i parametri marshalling:
 - Stub lato client → individua il server e "impacchetta" i parametri
 - Stub lato server → riceve il messaggio, spacchetta i parametri ed esegue la procedura sul server
- Uno stub è una interfaccia di comunicazione che implementa il protocollo RPC e specifica come i messaggi vengono costruiti ed interscambiati.
- In Windows, il codice stub viene compilato dalle specifiche scritte in **Microsoft Interface Definition Language (IDL)**.
- La semantica delle RPC permette ad un client di invocare una procedura presente su un sistema remoto nello stesso modo in cui esso invocherebbe una procedura locale.

Perché stub esegue il marshalling dei parametri?

La rappresentazione dei dati è gestita tramite l'utilizzo di un formato standard di trasmissione (XDR - External Data Representation) per tenere conto di diverse architetture per gestire l'eventuale diversità di rappresentazione dei dati fra sistemi remoti (es CPU 32 64 bit, **big-endian little-endian**).

La comunicazione remota ha più scenari di errore rispetto a quella locale.

- I messaggi possono essere consegnati esattamente una volta (timestamp + ack) piuttosto che al massimo una volta (timestamp)

Il sistema operativo in genere fornisce un servizio di rendezvous (o **matchmaker**) a collegare client e server.

--- CAPITOLO 4 - THREAD E CONCORRENZA ---

MOTIVAZIONI

La maggior parte delle applicazioni attuali sono multithread

I thread vengono eseguiti "all'interno" delle applicazioni

La gestione dei processi può diventare molto onerosa, sia dal punto di vista computazionale che per l'utilizzo di risorse :

- Creazione: allocazione dello spazio degli indirizzi e successiva popolazione
- Context-switch: salvataggio e ripristino degli spazi di indirizzamento (codice, dati, stack) di due processi

La programmazione multithread può produrre codice più semplice e più efficiente

I kernel attuali sono normalmente multithread

Il multithreading si adatta perfettamente alla programmazione nei sistemi multicore

DEFINIZIONE

Un thread (trama, filo) è l'unità base d'uso della CPU e comprende un identificatore di thread (TID), un contatore di programma, un insieme di registri ed uno stack:

- Condivide con gli altri thread che appartengono allo stesso processo la sezione del codice, la sezione dei dati e le altre risorse allocate al processo originale (file aperti e segnali)

Un processo tradizionale - heavyweight process - è composto da un solo thread:

- Un processo è "pesante", in riferimento al contesto (spazio di indirizzamento, stato) che lo definisce

Poiché i thread appartenenti ad uno stesso processo ne condividono codice, dati e risorse.

Un processo multithread è in grado di eseguire più compiti in modo concorrente

VANTAGGI

- 1) **Tempi di risposta/Reattività** → Rendere multithread un'applicazione interattiva permette ad un programma di continuare la propria esecuzione, anche se una parte di esso è bloccata o sta eseguendo un'operazione particolarmente lunga, riducendo il tempo medio di risposta. E' utile per le interfacce utente.
- 2) **Condivisione delle risorse** → I thread condividono la memoria e le risorse del processo cui appartengono il vantaggio della condivisione del codice risiede nel fatto che un'applicazione può consistere di molti thread relativi ad attività diverse, tutti nello stesso spazio di indirizzi (comunicazioni molto più rapide che con memoria condivisa o message passing)
- 3) **Economia** → Assegnare memoria e risorse per la creazione di nuovi processi è oneroso poiché i thread condividono le risorse del processo cui appartengono, è molto più conveniente creare thread e gestirne i cambiamenti di contesto (il TCB dei thread non contiene informazioni relative a codice e dati, quindi alla gestione della memoria)
- 4) **Scalabilità** → I vantaggi della programmazione multithread aumentano notevolmente nelle architetture multiprocessore, dove i thread possono essere eseguiti in parallelo l'impiego della programmazione multithread in un sistema con più unità di elaborazione fa aumentare il grado di parallelismo

PROGRAMMAZIONE MULTICORE

Architetture **multicore** diverse unità di calcolo sullo stesso chip: Ogni unità appare al SO come un processore separato.

Sui sistemi multicore i thread possono essere eseguiti in parallelo, poiché il sistema può assegnare thread diversi a ciascuna unità di calcolo.

I sistemi multicore si occupano anche di:

- **Divisione delle attività**
- **Bilanciamento**
- **Separazione dei dati**
- **Dipendenza dai dati**
- **Test e debug**

Distinzione fra **parallelismo e concorrenza**:

- Un sistema è parallelo se può eseguire simultaneamente più task
- Un sistema concorrente invece, supporta più task, consentendo a tutti di progredire nell'esecuzione grazie al multiplexing della CPU (unica).

Tipi di **parallelismo**:

- Con il termine **data parallelism** ci si riferisce a scenari in cui la stessa operazione viene eseguita contemporaneamente (ovvero in parallelo) sull'insieme dei dati; nelle operazioni in parallelo sui dati, l'insieme originale viene suddiviso in partizioni in modo che più thread (che compiono la stessa operazione) possano agire simultaneamente su segmenti diversi.
- Con il termine **task parallelism** si indica una forma di parallelizzazione del codice tra più processori in ambienti di calcolo parallelo; il task parallelism si concentra cioè sulla distribuzione di thread diversi in esecuzione nei diversi nodi di calcolo (che possono agire su dati comuni)

LA LEGGE DI AMDAHL

Permette di determinare i potenziali guadagni in termini di prestazioni ottenuti dall'aggiunta di core, nel caso di applicazioni che contengano sia componenti seriali (non parallelizzabili) sia componenti parallele

- S → porzione seriale
- N → core

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

Esempio: Se un'applicazione è al 75% parallela ed al 25% seriale, passando da 1 a 2 core, si ottiene uno speedup pari a 1.6;

- se i core sono 4 lo speedup è invece di 2.28

In generale per N → INFINITO lo speedup tende a 1/S

La porzione seriale di un'applicazione ha un effetto dominante sulle prestazioni ottenibili con l'aggiunta di nuovi core.

THREAD A LIVELLO UTENTE E THREAD A LIVELLO KERNEL

Thread a livello utente:

- Sono gestiti come uno strato separato sopra il nucleo del sistema operativo
- Sono realizzati tramite librerie di funzioni per la creazione, lo scheduling e la gestione dei thread, senza alcun intervento diretto del nucleo

Le tre librerie principali sono:

- POSIX **Pthreads** → è la libreria per la realizzazione di thread utente in sistemi UNIX like
- **Windows threads** → per sistemi Windows
- **Java threads** → per la JVM

Thread a livello kernel:

- Sono gestiti direttamente dal SO il nucleo si occupa di creazione, scheduling, sincronizzazione e cancellazione dei thread nel suo spazio di indirizzi.
- Supportati da tutti i sistemi operativi attuali
 - Windows
 - Solaris
 - Linux
 - Mac OS
 - iOS
 - Android

MODELLI DI PROGRAMMAZIONE MULTITHREAD

- Modello molti-a-uno (M:1)
- Modello uno-a-uno (1:1)
- Modello molti-a-molti (M:M)

MODELLO MOLTI A UNO

Molti thread a livello utente vanno a corrispondere ad un unico thread a livello kernel.

Il kernel "vede" una sola traccia di esecuzione.

In altre parole i thread sono implementati a livello di applicazione, il loro scheduler non fa parte del SO, che continua ad avere solo la visibilità del processo.

Il blocco di un thread (a livello di utente) provoca il blocco di tutti.

Più thread (a livello di utente) potrebbero non essere eseguiti in parallelo sul sistema multicore perché solo uno (thread del kernel) può essere nel kernel alla volta.

Pochi sistemi attualmente utilizzano questo modello

MODELLO UNO A UNO

Ciascun thread a livello utente corrisponde ad un thread a livello kernel.

La creazione di un thread a livello utente crea un thread del kernel

Maggior concorrenza rispetto a molti-a-uno

Numero di thread per processo a volte limitato a causa del sovraccarico

Esempi : Windows 95/98/2000/XP/Vista e versioni attuali, Linux, Solaris (versione 9 e successive)

MODELLO MOLTI A MOLTI

Si mettono in corrispondenza più thread a livello utente con un numero minore o uguale di thread a livello kernel.

Consente al sistema operativo di creare un numero sufficiente di thread del kernel.

Windows con il pacchetto ThreadFiber.

MODELLO A DUE LIVELLI

Simile al modello M:M, offre però la possibilità di vincolare un thread utente ad un thread del kernel.

LIBRERIE DEI THREAD

Forniscono al programmatore una API per la creazione e la gestione dei thread

Librerie a livello utente:

- Codice e strutture dati nello spazio utente
- Invocare una funzione di libreria si traduce in una chiamata locale a funzione, non in una system call

Librerie a livello kernel:

- Codice e strutture dati nello spazio del kernel.
- È supportata dal sistema operativo.
- Invocare una funzione della API provoca, generalmente, una chiamata di sistema.

PTHREADS

- Può essere fornito sia a livello di utente che a livello di kernel
- È lo standard POSIX (IEEE 1003.1.c) che definisce la API per la creazione e la sincronizzazione dei thread
- È una specifica, non implementazione
- L'API specifica il comportamento della libreria di thread, l'implementazione spetta allo sviluppo della libreria
- Comune nei SO UNIX like (Linux, Mac OS)

THREAD IN JAVA

- I thread Java sono gestiti dalla JVM
- Tipicamente implementato utilizzando il modello di thread fornito dal sottostante Sistema operativo
- I thread Java possono essere creati da:
 - Estensione della classe Thread
 - Implementazione dell'interfaccia eseguibile
 - La pratica standard consiste nell'implementare l'interfaccia Runnable

THREADING IMPLICITO

Le applicazioni multithread sono più difficili da programmare ed aumentano in complessità al crescere del numero di thread.

Trasferimento della creazione e della gestione del threading dagli sviluppatori di applicazioni ai compilatori ed alle librerie di runtime.

Ci sono 5 metodi di Threading implicito :

1. Gruppi di thread (Thread pool)
2. Fork-Join
3. OpenMP
4. Grand Central Dispatch
5. Intel Threading Building Blocks

1 - GRUPPI DI THREAD (THREAD POOL)

Un numero illimitato di thread presenti nel sistema potrebbe esaurirne le risorse → Creare un certo numero di thread worker alla creazione del processo ed organizzarli in un gruppo in cui attendano il lavoro che verrà loro

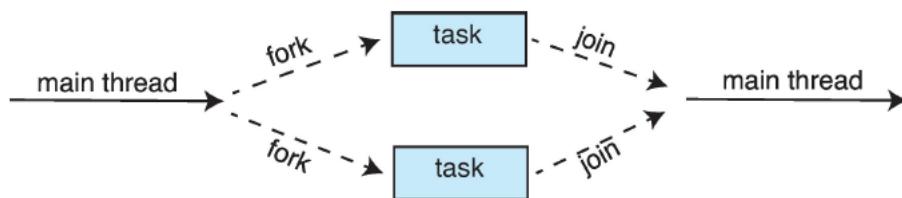
successivamente richiesto.

Vantaggi:

- Di solito il servizio di una richiesta tramite un thread esistente è più rapido, poiché elimina l'attesa della creazione di un nuovo thread
- Si limita il numero di thread relativi a ciascun processo alla dimensione prestabilita (rilevante per sistemi che non possono sostenere un numero elevato di thread concorrenti)
- Separare il task da eseguire dal meccanismo di creazione dello stesso permette strategie diverse di elaborazione
 - o Esempio i task possono essere schedulati per essere eseguiti periodicamente o dopo un dato intervallo di tempo.

2 - FORK-JOIN

Il genitore crea uno o più figli e attende che tutti terminino prima di riprendere l'esecuzione.



3 - OPENMP

- OpenMP è un insieme di direttive del compilatore ed una API per programmi scritti in C, C++ e FORTRAN.
- Fornisce supporto per la programmazione parallela in ambienti a memoria condivisa
- Identifica le regioni parallele cioè i blocchi di codice eseguibili in parallelo
- OpenMP permette inoltre di impostare manualmente il numero di thread ed il livello di condivisione dei dati fra i thread

4 - GRAND CENTRAL DISPATCH (Grande invio centrale)

- Tecnologia Apple per sistemi operativi macOS e iOS
- Estensioni a linguaggi C, C++ e Objective-C, API e biblioteca runtime
- Consente l'identificazione di sezioni parallele
- Gestisce la maggior parte dei dettagli del threading (in modo simile a ciò che accade con OpenMP)
- Il blocco è in "```" → `printf("Io sono un blocco");`
- Blocchi posizionati in coda di spedizione:
 - Assegnato al thread disponibile nel pool di thread quando viene rimosso dalla coda

Due tipi di code di spedizione:

- **seriale** - blocchi rimossi in ordine FIFO, la coda è per processo, chiamata coda principale.
 - I programmati possono creare code seriali aggiuntive all'interno del programma
- **simultanei**: rimossi in ordine FIFO, ma possono essere rimossi più di uno alla volta.

- Per il linguaggio Swift, un'attività è definita come una chiusura, simile a un blocco, meno il cursore (cioè "```")

5 - INTEL THREADING BUILDING BLOCKS (TBB)

- Libreria di modelli per la progettazione di programmi C++ paralleli
- Una versione seriale di un ciclo for semplice:

```
for(int i = 0; i < n; i++) {  
    apply(v[i]);  
}
```

- Lo stesso ciclo for scritto usando TBB con l'istruzione parallel_for:

```
parallel_for (range body)
```

PROBLEMI DI THREADING

- Semantica delle chiamate di sistema **fork()** ed **exec()**
- Gestione dei segnali (sincroni vs asincroni)
- Cancellazione dei thread (asincrona vs differita)
- Dati specifici dei thread
- Attivazione dello scheduler

SEMANTICA DI fork() ED exec()

In un programma multithread la semantica delle system call **fork()** cambia:

- Se un thread in un programma invoca la **fork()** il nuovo processo potrebbe, in generale, contenere un **duplicato** di tutti i thread oppure del solo thread invocante.
- La scelta fra le due opzioni dipende dalla immediatezza o meno della successiva chiamata ad **exec()** - la cui modalità di funzionamento non cambia.
- Se la chiamata di **exec()** avviene immediatamente dopo la chiamata alla **fork()** la duplicazione dei thread non è necessaria, poiché il programma specificato nei parametri della **exec()** sostituirà in toto il processo → si duplica il solo thread chiamante
 - Altrimenti, tutti i thread devono essere duplicati
- Alcune release di UNIX rendono disponibili entrambe le implementazioni della **fork()**

GESTIONE DEI SEGNALI

Nei sistemi UNIX si usano **segnali** per comunicare ai processi il verificarsi di determinati eventi.

I segnali vengono elaborati da un gestore dei segnali :

- 1) All'occorrenza di un particolare evento, si genera un segnale
- 2) Si invia il segnale ad un processo
- 3) Una volta ricevuto, il segnale deve essere gestito da un gestore, che può essere :
 - predefinito (gestito dal SO)
 - definito dall'utente

I segnali possono essere...

- **sincroni** se vengono inviati allo stesso processo che ha eseguito l'operazione causa del segnale (es.: divisione per 0)
- **asincroni** se causati da eventi esterni al processo in esecuzione (es.: scadenza di un timer o ^C.)

Per ogni segnale esiste un gestore predefinito del segnale che il kernel esegue in corrispondenza del particolare evento:

- Il gestore definito dall'utente, se presente, si sostituisce al gestore standard
- Nei processi a singolo thread, i segnali vengono inviati al processo

Cosa accade nel caso di processi multithread?

Nei sistemi multithread è possibile:

- inviare il segnale al thread cui il segnale si riferisce
- inviare il segnale ad ogni thread del processo
- inviare il segnale a specifici thread del processo
- definire un thread specifico per ricevere tutti i segnali diretti al processo
- I segnali sincroni devono essere recapitati al thread che ha generato l'evento causa del segnale.
- Alcuni segnali asincroni (per esempio il segnale di terminazione di un processo, ^C) devono essere inviati a tutti i thread

CANCELLAZIONE DEI THREAD

- È l'operazione che permette di terminare un thread prima che completi il suo compito:

- Esempio: pressione del pulsante di terminazione di un programma di consultazione del Web per interrompere il caricamento di una pagina
- Il thread da cancellare viene definito **thread bersaglio (target thread)**.
- La **cancellazione** di un thread bersaglio può avvenire:
 - **in modalità asincrona:** terminazione immediata del thread bersaglio
 - Problemi se si cancella un thread mentre sta aggiornando dati che condivide con altri thread
 - **in modalità differita:** il thread bersaglio può periodicamente controllare se deve terminare, in modo da riuscirvi al momento opportuno (cancellation point, in Pthreads).
- La chiamata della `pthread_cancel()` comporta solo una richiesta di cancellazione del thread bersaglio l'effettiva cancellazione dipende dallo stato del thread
- In caso di cancellazione **in modalità differita:** disabilitata, l'operazione rimane pendente finché il thread non procede all'abilitazione.

THREAD LOCAL STORAGE

- Il **Thread Local Storage (TLS)** permette a ciascun thread di possedere una propria copia dei dati (non necessariamente tutti i dati del processo di origine)
- Utili quando non si ha controllo sul processo di creazione dei thread (per esempio, nel caso dei gruppi di thread)
- Diversamente dalle variabili locali (che sono visibili solo durante una singola chiamata di funzione), i dati TLS sono visibili attraverso tutte le chiamate.

ATTIVAZIONE DELLO SCHEDULER

- Sia il modello M M che il modello a due livelli richiedono che sia mantenuta una comunicazione costante per garantire un numero sufficiente di thread del kernel allocati ad una particolare applicazione
- In genere, si alloca una struttura dati intermedia nota come **processo leggero o (LWP) LightWeight Process:**
 - Per la libreria dei thread a livello utente, LWP è un processore virtuale a cui l'applicazione può richiedere lo scheduling di un thread a livello utente.
 - Ogni LWP è collegata al thread del kernel.
 - Quanti LWP dovranno essere creati per una data applicazione?
- L'attivazione dello scheduler mette a disposizione la procedura di **upcall** - un meccanismo di comunicazione dal kernel alla libreria di gestione dei thread.
- Tale comunicazione garantisce all'applicazione la capacità di mantenere attivi un numero opportuno di thread a livello kernel.

THREAD IN WINDOWS

- API di Windows: API primaria per le applicazioni Windows
- Implementa la mappatura uno-a-uno, a livello di kernel
- Ogni thread contiene:
 - Un ID thread
 - Insieme di registri che rappresenta lo stato del processore
 - Stack utente e kernel separati per quando il thread viene eseguito in modalità utente o in modalità kernel
 - Area di archiviazione dati privata utilizzata dalle librerie di runtime e dalle librerie a collegamento dinamico (DLL)
- Il set di registri, gli stack e l'area di archiviazione privata sono noti come il **contesto del thread**
- Le strutture dati primarie di un thread includono:
 - ETHREAD (executive thread block) – include un puntatore al processo a cui appartiene il thread e a KTHREAD, nello spazio del kernel.
 - KTHREAD (kernel thread block) – informazioni sulla pianificazione e sincronizzazione, stack in modalità kernel, puntatore a TEB, nello spazio del kernel
 - TEB (thread environment block) – ID thread, stack in modalità utente, archiviazione thread-local, nello spazio utente.

THREAD IN LINUX

- Linux li definisce **task** (come i processi)
- La creazione dei task avviene con la chiamata della system call **clone()**
- All'invocazione, **clone()** riceve come parametro un insieme di indicatori (flag), che definiscono quante e quali risorse del task genitore saranno condivise dal task figlio:
 - **clone()** può permettere al task figlio la condivisione totale dello spazio degli indirizzi e delle risorse del task padre: si crea un **thread**.

--- CAPITOLO 5 - SCHEDULING DELLA CPU ---

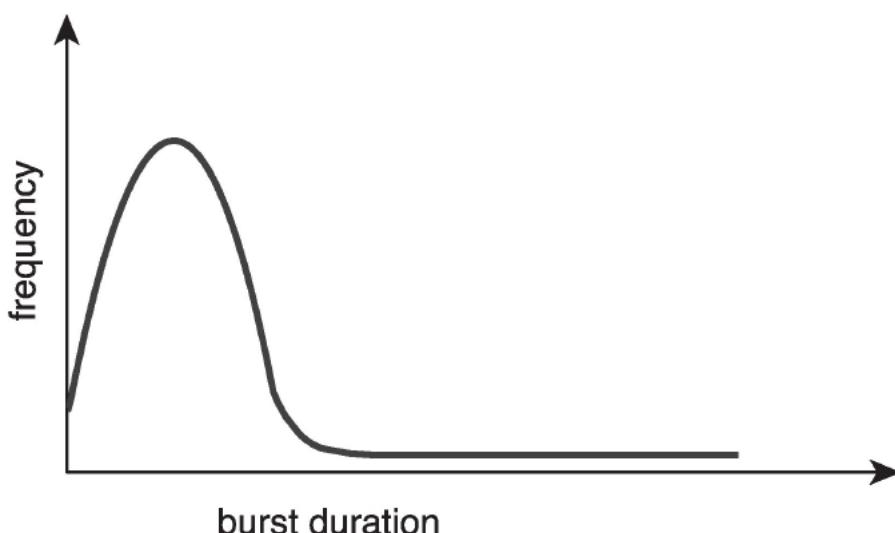
OBBIETTIVI

- Descrivere vari algoritmi di scheduling della CPU
- Valutare gli algoritmi in base ai criteri di scheduling
- Spiegare le problematiche relative allo scheduling multiprocessore e multicore
- Descrivere vari algoritmi di scheduling real-time
- Utilizzare modellazione e simulazioni per valutare gli algoritmi di scheduling della CPU

CONCETTI FONDAMENTALI

- Lo scheduling è una funzione fondamentale dei sistemi operativi:
 - o Si sottopongono a scheduling quasi tutte le risorse di un calcolatore.
- Lo **scheduling della CPU** è alla base dei sistemi operativi multiprogrammati:
 - o Attraverso la commutazione del controllo della CPU tra i vari processi, il SO rende più "produttivo" il sistema di calcolo.
- Il massimo impiego della CPU è pertanto ottenuto con la multiprogrammazione
- Ciclo di CPU-I/O burst – L'elaborazione di un processo consiste di cicli di esecuzione nella CPU ed attese/utilizzo ai/dei dispositivi di I/O.
- I CPU-burst si susseguono agli I/O-burst durante tutta la vita del processo.
- La distribuzione delle CPU-burst è la principale preoccupazione.
- Un programma con prevalenza di I/O si definisce I/O-bound e produce generalmente molte sequenze di operazioni della CPU di breve durata.
- Un programma con prevalenza di elaborazione si definisce CPU-bound e produce (poche) sequenze di esecuzione molto lunghe.

GRAFICO DELLA DISTRIBUZIONE DEI CPU BURST



LO SCHEDULER DELLA CPU

Lo scheduler della CPU gestisce la coda dei processi pronti, selezionando il prossimo processo cui verrà allocata la CPU.

- La coda pronta può essere ordinata in vari modi.
- Lo scheduler della CPU deve prendere una decisione quando un processo:
- 1) passa da stato running a stato wait (es.: richiesta di I/O o attesa terminazione di un processo figlio)
 - 2) passa da stato running a stato ready (interrupt).
 - 3) passa da stato wait a stato ready (es.: completamento di un I/O)
 - 4) termina
- Per le situazioni 1 e 4, non c'è scelta in termini di programmazione.

Deve essere presente un nuovo processo (se presente nella coda dei processi pronti) selezionato per l'esecuzione.

- Per le situazioni 2 e 3, invece, c'è una scelta
- Se lo scheduling viene effettuato solo nei casi 1 e 4 si dice che lo schema di scheduling è **nonpreemptive** (senza diritto di prelazione) o cooperativo.
- Altrimenti si ha uno scheduling **preemptive**.

Quando lo scheduling è **nonpreemptive**, una volta che la CPU è stata allocata a un processo, il processo mantiene la CPU fino a quando non la rilascia terminando o passando allo stato di attesa.

- Qual è il potenziale problema?

- Praticamente tutti i moderni sistemi operativi incluso Windows, MacOS, Linux e UNIX utilizzano gli algoritmi di **scheduling preemptive**.
- Lo scheduling **preemptive** può avere race condition (situazioni di corsa) quando i dati vengono condivisi tra più processi.
- Si consideri il caso di due processi che condividono dati. Mentre un processo aggiorna i dati, viene anticipato in modo che il secondo processo possa essere eseguito. Il secondo processo tenta quindi di leggere i dati, che sono in uno stato incoerente.

IL DISPATCHER

- Il modulo dispatcher passa il controllo della CPU al processo selezionato dallo scheduler; il dispatcher effettua:
 - Context switch
 - Passaggio a modo utente
 - Salto alla posizione corretta del programma utente per riavviare l'esecuzione
- **Latenza di dispatch** - è il tempo impiegato dal dispatcher per sospendere un processo e avviare una nuova esecuzione.

CRITERI DI SCHEDULING

- **Utilizzo della CPU** → la CPU deve essere più attiva possibile
- **Throughput (produttività)** → numero dei processi che completano la loro esecuzione nell'unità di tempo.
- **Tempo di turnaround (tempo di completamento)** → tempo impiegato per l'esecuzione di un determinato processo
- **Tempo di attesa** → tempo passato dal processo in attesa nella ready queue
- **Tempo di risposta** → tempo che intercorre tra la sottomissione di un processo e la prima risposta prodotta

CRITERI DI OTTIMIZZAZIONE PER GLI ALGORITMI DI SCHEDULING

- Massimo utilizzo della CPU
- Massimo throughput
- Minimo tempo di turnaround
- Minimo tempo di attesa
- Minimo tempo di risposta

SCHEDULING First-Come-First-Served (FCFS)

- La CPU viene assegnata al processo che la richiede per primo la realizzazione del criterio FCFS si basa sull'implementazione della ready queue per mezzo di una coda FIFO
- Quando un processo entra nella ready queue, si collega il suo PCB all'ultimo elemento della coda quando la CPU è libera, viene assegnata al processo che si trova alla testa della ready queue, rimuovendolo da essa.

SCHEDULING Shortest-Job-First (SJF)

- Si associa a ciascun processo la lunghezza del suo burst di CPU successivo si opera lo scheduling in base alla brevità dei CPU-burst.
- SJF è ottimo - minimizza il tempo medio di attesa:
 - Difficoltà nel reperire l'informazione richiesta

LUNGHEZZA DEL SUCCESSIVO CPU-burst

Può essere stimata come una media esponenziale delle lunghezze dei burst di CPU precedenti (impiegando una combinazione convessa)

1. t_n = lunghezza dell' n -esimo CPU burst
2. τ_{n+1} = valore stimato del prossimo CPU burst
3. $0 \leq \alpha \leq 1$

⇒ Si definisce $\tau_{n+1} = \alpha t_n + (1-\alpha) \tau_n$

τ_1 = valore costante o stimato come media dei burst del sistema

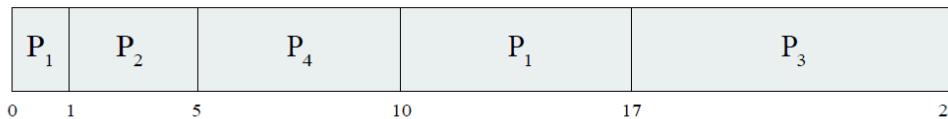
⇒ Solitamente, si usa $\alpha=1/2$

SHORTEST REMAINING TIME FIRST SCHEDULING

- Versione preventiva di SJF
- Ogni volta che un nuovo processo arriva nella coda di pronto, la decisione su quale processo programmare successivamente viene ripetuta utilizzando l'algoritmo SJF
- SRT è più "ottimale" di SJF in termini di tempo di attesa medio minimo per un determinato insieme di processi?
 - Now we add the concepts of varying arrival times and preemption to the analysis

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- Preemptive SJF Gantt Chart



26

SCHEDULING ROUND ROBIN

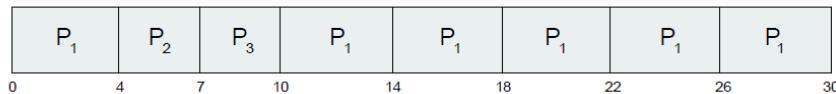
- A ciascun processo viene allocata una piccola quantità di tempo di CPU, un **quanto di tempo** (time slice), generalmente 10-100 millisecondi:
 - Dopo un quanto di tempo, il processo è forzato a rilasciare la CPU e viene accodato alla ready queue
 - La ready queue è trattata come una coda (circolare)
- Se vi sono n processi nella ready queue ed il quanto di tempo è q, ciascun processo occupa 1/n del tempo di CPU in frazioni di, al più, q unità di tempo; nessun processo attende per più di $(n-1)*q$ unità di tempo.
- Il timer interrompe la CPU allo scadere del quanto per programmare il prossimo processo.

Prestazioni:

- q grande → FIFO
- q piccolo → q deve essere grande rispetto al tempo di context switch (tuttavia, è normalmente < 10 nano secondi), altrimenti l'overhead è troppo alto.

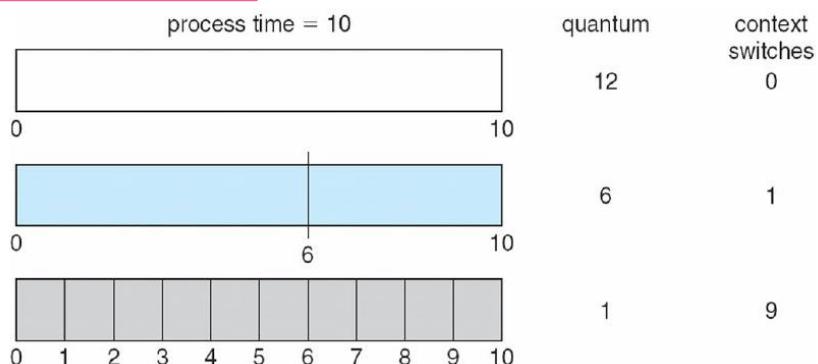
Processo	Tempo di burst
P ₁	24
P ₂	3
P ₃	3

- Il diagramma di Gantt con $q = 4$ è:



- In genere si ha un tempo medio di attesa e di turnaround maggiore rispetto a SJF, tuttavia si ottiene un miglior tempo medio di risposta
- Tempi di attesa: P₁→6, P₂→4, P₃→7
- Tempo medio di attesa T_a=(6+4+7)/3=5.6msec

QUANTO DI TEMPO E CONTEXT-SWITCH



Un quanto di tempo minore incrementa il numero di context switch

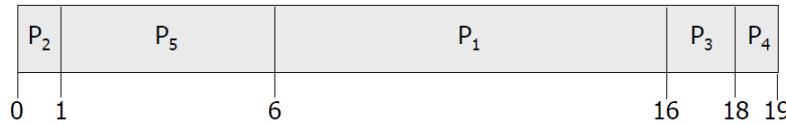
SCHEDULING A PRIORITA'

- Un valore di priorità (intero) viene associato a ciascun processo.
- La CPU viene allocata al processo con la priorità più alta (intero più basso → priorità più alta):
 - Preemptive
 - Non-preemptive
- Problema: **Starvation** ("inedia", blocco indefinito) - i processi a bassa priorità potrebbero non venir mai seguiti.
- Soluzione: **Aging** (invecchiamento) - aumento graduale della priorità dei processi che si trovano in attesa nel sistema da lungo tempo.
- SJF è uno scheduling a priorità in cui la priorità è calcolata in base al successivo tempo di burst

Processo	Tempo di burst	Priorità
P ₁	10	3
P ₂	1	1
P ₃	2	4
P ₄	1	5
P ₅	5	2

Priorità massima

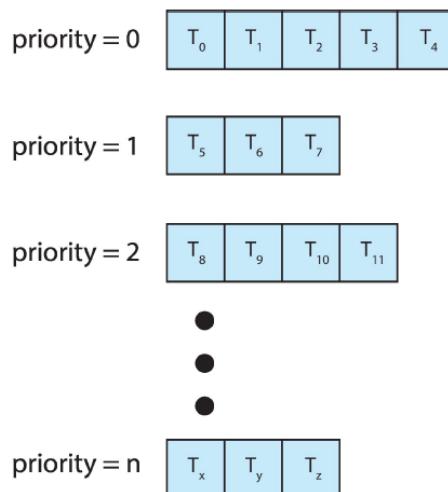
- Il diagramma di Gantt è:



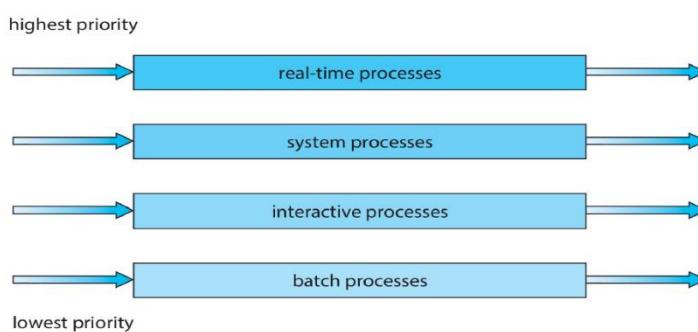
- Tempi di attesa: P₁ → 6, P₂ → 0, P₃ → 16, P₄ → 18, P₅ → 1
- Tempo medio di attesa $T_a = (6+0+16+18+1)/5 = 8.2\text{msec}$

CODA MULTILIVELLO (Multilevel Queue) / Scheduling con code multiple

- La ready queue è composta da più code.
- Esempio:
 - Scheduling a priorità, in cui ciascuna priorità ha la propria coda separata
 - Pianifica il processo nella coda con la priorità più alta!



- I processi si assegnano in modo permanente ad una coda, generalmente secondo qualche caratteristica (invariante) del processo :



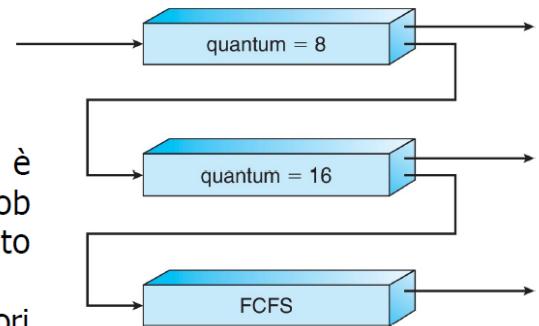
CODA MULTIPLE CON FEEDBACK

- Un processo può spostarsi fra le varie code se si può implementare l'aging
- Lo scheduler a code multiple con feedback è definito dai seguenti parametri:
 - Numero di code
 - Algoritmo di scheduling per ciascuna coda

- Metodo impiegato per determinare quando spostare un processo in una coda a priorità maggiore
- Metodo impiegato per determinare quando spostare un processo in una coda a priorità minore
- Metodo impiegato per determinare in quale coda deve essere posto un processo quando entra nel sistema

▪ Tre code:

- Q_0 – RR con quanto di tempo di 8 millisecondi
- Q_1 – RR con quanto di tempo di 16 millisecondi
- Q_2 – FCFS



Scheduling

- Un nuovo processo viene immesso nella coda Q_0 , che è servita con RR; quando prende possesso della CPU il job riceve 8 millisecondi; se non termina, viene spostato nella coda Q_1
- Nella coda Q_1 il job è ancora servito RR e riceve ulteriori 16 millisecondi; se ancora non ha terminato, viene spostato nella coda Q_2 , dove verrà servito con criterio FCFS all'interno dei cicli di CPU lasciati liberi dai processi delle code Q_0 e Q_1

SCHEDULING DEI THREAD

- Esiste una distinzione fondamentale fra lo scheduling dei thread a livello utente e a livello kernel ma, certamente, se i thread sono supportati dal sistema, lo scheduling avviene a livello di thread.
- Scheduling locale: nei modelli molti-a-uno e molti-a-molti, è l'algoritmo con cui la libreria dei thread decide quale thread (a livello utente) allocare a un LWP disponibile (**ambito della competizione ristretto al processo, PCS**) – normalmente un algoritmo a priorità con prelazione definito dal programmatore.
 - La libreria dei thread pianifica l'esecuzione su LWP
 - Tuttavia, il thread prescelto non è effettivamente in esecuzione su una CPU fisica finché il SO non pianifica l'esecuzione del thread a livello kernel corrispondente.
- Scheduling globale: l'algoritmo con cui il kernel decide quale thread a livello kernel dovrà venire eseguito successivamente ambito di competizione allargato al **sistema SCS**.

API Pthreads PER LO SCHEDULING DEI THREAD

- La API Pthreads di POSIX consente di specificare PCS (Process-Contetion-Scope) o SCS (System-Contetion-Scope) nella fase di generazione dei thread.
- Per specificare l'ambito della contesa, Pthreads usa i valori:
 - `PTHREAD_SCOPE_PROCESS` scheduling PCS
 - `PTHREAD_SCOPE_SYSTEM` scheduling SCS

SCHEDULING MULTIPROCESSORE

- Lo scheduling diviene più complesso quando nel sistema di calcolo sono presenti più CPU.
- Le architetture multiprocessore possono essere:
 - CPU multicore
 - Multicore multithread
 - Sistemi NUMA
 - Sistemi eterogenei
- Multielaborazione simmetrica (SMP, Symmetric Multi-Processing) → i thread pronti vanno a formare una coda comune oppure vi è un'apposita coda per ogni processore ciascun processore ha un proprio scheduler che esamina la coda opportuna per prelevare il prossimo thread da eseguire.

PROCESSORI MULTICORE

- Tendenza a posizionare più core del processore sullo stesso chip fisico
- Più veloce e consuma meno energia
- Anche più thread per core sono in crescita:

- Sfrutta lo **stallo della memoria** per fare progressi su un altro thread durante il recupero della memoria
- implementano unità di calcolo multithread, in cui due o più thread hardware sono assegnati ad un singolo core
- Se un thread ha un blocco di memoria, passa a un altro thread!
- Nei **chip multithreading (CMT)**, a ciascun core vengono assegnati un certo numero di thread hardware.
- In un quad-core, con 2 thread hardware per core, si hanno 8 processori logici.

Un processore multicore e multithread richiede due livelli di scheduling:

- Scheduling effettuato dal SO per stabilire quale thread software mandare in esecuzione su ciascun thread hardware - algoritmi standard
- Scheduling relativo a ciascun core per decidere quale thread hardware mandare in esecuzione

MULTIPLE PROCESSOR SCHEDULING - BILANCIAMENTO DEL CARICO (LOAD BALANCING)

Bilanciamento del carico → Ripartire uniformemente il carico di lavoro sui diversi processori.

- Nei sistemi con ready queue comune è automatico
- **Migrazione guidata (push migration)** un thread dedicato controlla periodicamente il carico dei processori per effettuare eventuali riequilibri.
- **Migrazione spontanea pull migration** un processore inattivo sottrae ad uno sovraccarico un thread in attesa

MULTIPLE PROCESSOR SCHEDULING - AFFINITÀ DEL PROCESSORE

- Mantenere un thread in esecuzione sempre sullo stesso processore, per riutilizzare il contenuto della cache
 - per burst successivi.
 - **Predilezione debole (soft affinity)** - il SO tenta di mantenere il thread su un singolo processore, ma la migrazione non è interdetta
 - **Predilezione forte (hard affinity)** - il thread è vincolato all'esecuzione su uno (o più) processori

NUMA E CPU SCHEDULING

In caso di **NUMA** (Non Uniform Memory Access), situazione standard in sistemi costituiti da diverse schede ognuna con una o più CPU e memoria, le CPU di una scheda accedono più velocemente alla memoria locale, rispetto alle memorie residenti sulle altre schede.

Se il sistema operativo è compatibile con NUMA, assegnerà le chiusure di memoria alla CPU su cui è in esecuzione il thread.

SCHEDULING REAL TIME

- **Sistemi hard real-time** richiedono il completamento dei processi critici entro un intervallo di tempo predeterminato e garantito.
- **Sistemi soft real-time** richiedono solo che ai processi critici sia assegnata una maggiore priorità rispetto ai processi di routine (nessuna garanzia sui tempi di completamento).

I sistemi real-time, specialmente hard real-time, sono per loro natura guidati dagli eventi:

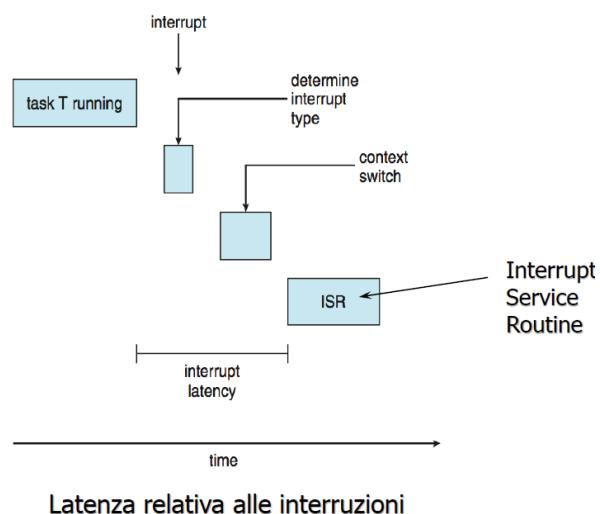
- Si attende che si verifichi un evento cui si deve reagire in tempo reale.
- Latenza dell'evento tempo che intercorre tra l'occorrenza dell'evento e il momento in cui il sistema ne effettua la gestione.

Categorie di latenza che influiscono sul funzionamento dei sistemi real-time:

- 1) **Latenza relativa alle interruzioni (Interrupt Latency)**: tempo compreso tra la notifica di un'interruzione alla CPU e l'avvio della routine che la gestisce.
 - Fondamentale minimizzare l'intervallo di tempo in cui le interruzioni sono disattivate per l'aggiornamento dei dati del kernel.
- 2) **Latenza di dispatch (Dispatch Latency)**: tempo necessario al dispatcher per interrompere un processo ed avviare il codice di gestione dell'evento.

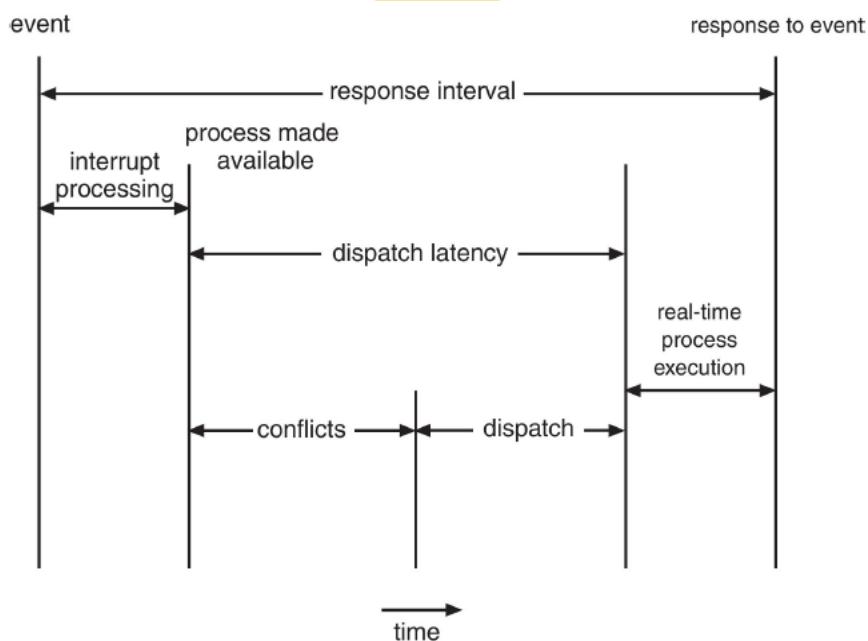
INTERRUPT LATENCY - LATENZA RELATIVA ALLE INTERRUZIONI

Latenza relativa alle interruzioni: tempo compreso tra la notifica di un'interruzione alla CPU e l'avvio della routine che la gestisce



DISPATCH LATENCY - LATENZA DI DISPATCH

Latenza di dispatch: tempo necessario al dispatcher per interrompere un processo ed avviare il codice di gestione dell'evento



La cosiddetta fase di conflitto nella latenza di dispatch consiste di due componenti:

- Prelazione di ogni processo in esecuzione nel kernel
- Cessione, da parte dei processi a bassa priorità, delle risorse richieste dal processo ad alta priorità.

SCHEDULING BASATO SULLE PRIORITÀ

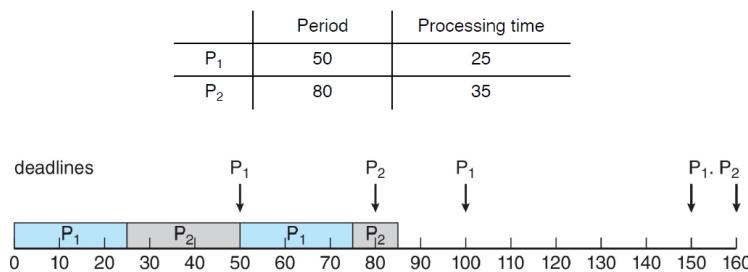
- Per lo scheduling real time, lo scheduler deve supportare la pianificazione preventiva e basata su priorità
- Ma garantisce solo soft real-time
- Per hard real-time deve anche fornire la capacità di rispettare le scadenze

- I processi hanno nuove caratteristiche: quelli **periodici** richiedono CPU a intervalli costanti:

- Ha tempo di elaborazione t , scadenza d , periodo p
- $0 \leq t \leq d \leq p$
- La frequenza dell'attività periodica è $1/p$

SCHEDULING CON PRIORITÀ PROPORZIONALE ALLA FREQUENZA (O RATE MONOTONIC)

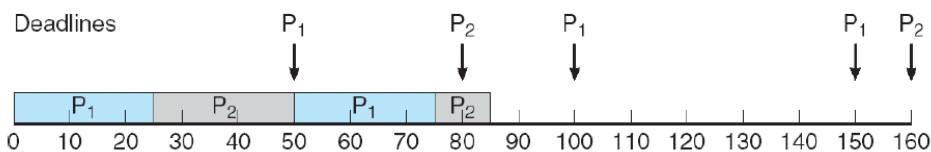
- La priorità è inversamente proporzionale al periodo:
 - Si assegnano priorità più elevate ai processi che fanno uso più frequente della CPU.
 - Periodi più brevi = priorità più alta
 - Periodi più lunghi = priorità più bassa



▪ CPU utilization: $25/50 + 35/80 \rightarrow 94\%$

Lo scheduling con priorità proporzionale alla frequenza è un algoritmo ottimale, nel senso che se non è in grado di pianificare l'esecuzione di una serie di processi rispettandone i vincoli temporali, nessun altro algoritmo che assegna priorità statiche vi riuscirà

- **Esempio:** siano P_1 e P_2 due processi con periodi rispettivamente pari a 50 e 80; Supponiamo che i tempi di elaborazione siano invece $t_1=25$ e $t_2=35$
- P_1 ha ancora priorità maggiore di P_2 , ma...



- ...in questo caso la scadenza relativa a P_2 non può essere rispettata (anche se si ha il 94% di utilizzo della CPU)

SCHEDULING EDF (Earliest-Deadline-First)

- Le priorità sono assegnate secondo le scadenze:

- Quanto prima è la scadenza, tanto maggiore è la priorità
- Più tardi è la scadenza, minore è la priorità

SCHEDULING A QUOTE PROPORZIONALI (Proportional Share Scheduling)

- Opera distribuendo un certo numero di quote, T fra tutte le applicazioni nel sistema

- Un'applicazione può ricevere N quote di tempo, assicurandosi così l'uso di una frazione N/T del tempo totale di CPU

- Esempio: T=100 quote a disposizione da ripartire fra i processi A B e C; se A dispone di 50 quote, B di 15 e C di 20 la CPU è occupata all' 85%.

- Lo scheduler deve lavorare in sinergia con un meccanismo di controllo dell'ammissione, per garantire che ogni applicazione possa effettivamente ricevere le quote di tempo che le sono state destinate (altrimenti non viene momentaneamente ammessa al sistema).

POSIX REAL TIME SCHEDULER

- Lo standard POSIX.1b è dedicato al scheduling real-time
- L'API fornisce funzioni per la gestione dei thread in tempo reale

- Definisce due (tre) classi di pianificazione per i thread in tempo reale:
 - 1) SCHED_FIFO - i thread vengono pianificati utilizzando una strategia FCFS con una coda FIFO. Non è previsto il time-slicing per i thread di uguale priorità.
 - 2) SCHED_RR - simile a SCHED_FIFO eccetto che il time-slicing si verifica per thread di uguale priorità.
 - 3) SCHED_OTHER - non implementato e definito dall'utente. Il comportamento può variare da sistema a sistema.
- Definisce due funzioni per ottenere e impostare la politica di pianificazione:
 - `pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`
 - `pthread_attr_setsched_policy(pthread_attr_t *attr, int politica)`

SCHEDULING LINUX FINO ALLA VERSIONE 2.5

- Prima della versione 2.5 del kernel era una variante dell'algoritmo tradizionale di UNIX (code multiple con feedback gestite perlomeno con RR).
- Con la versione 2.5 del kernel, lo scheduler è stato rivisitato per ottenere un algoritmo noto come O(1) indipendente dal numero di task presenti nel sistema.
 - Algoritmo a priorità preemptive
 - Due intervalli di priorità, relativi a task time-sharing, detti nice e (soft) real-time
 - Priorità dei task real-time con valori compresi fra 0 e 99, fra 100 e 139 per i task nice (valori numericamente inferiori indicano priorità maggiori)
 - Task a priorità maggiore ottengono time-slice più lunghi
 - I task vengono eseguiti (rimangono attivi) fino ad esaurimento del time-slice
 - Esaurito il quanto di tempo, il task viene considerato "scaduto" e non verrà più eseguito finché tutti gli altri task del sistema siano scaduti.

SCHEDULING LINUX DALLA VERSIONE 2.6.23 IN POI

Completely Fair Scheduler (CFS)

- Classi di scheduling:
 - Ciascuna dotata di una specifica priorità
 - Lo scheduler seleziona il processo a più alta priorità, appartenente alla classe di scheduling a priorità più elevata.
 - Invece di assegnare un quanto di tempo, lo scheduler CFS assegna ad ogni task una percentuale del tempo di CPU.
 - Il kernel Linux standard implementa due classi di scheduling, dette default e real-time (possono però essere aggiunte altre classi).
- Per la classe default, la percentuale del tempo di CPU viene calcolata sulla base dei valori nice associati a ciascun task (da -20 a +19 valori minori per priorità maggiori):
 - CFS non utilizza valori discreti per i quanti di tempo, ma definisce una latenza obiettivo, cioè un intervallo di tempo entro il quale ogni task eseguibile dovrebbe ottenere la CPU almeno una volta
 - La latenza obiettivo cresce con il numero di task
- Lo scheduler CFS non assegna direttamente le priorità, ma registra per quanto tempo è stato eseguito ogni task, mantenendo il tempo di esecuzione virtuale di ogni task nella variabile vruntime.
 - Il tempo di esecuzione virtuale è associato ad un fattore di decadimento che dipende dalla priorità del task (task a bassa priorità hanno fattori di decadimento più alti e vruntime maggiore rispetto al tempo effettivo di esecuzione).
 - Per i task con priorità normale (nice = 0 il tempo di esecuzione virtuale coincide con il tempo effettivo di esecuzione).
- Per decidere il prossimo task da eseguire, lo scheduler seleziona il task con il valore vruntime più piccolo.

SCHEDULING WINDOWS

- Windows utilizza lo scheduling preemptive (preventiva) basata sulla priorità.
 - Il thread con la priorità più alta viene eseguito successivamente
- Dispatcher è uno scheduler e il thread viene eseguito fino a:

- blocchi
 - utilizza l'intervallo di tempo
 - anticipato da thread con priorità più alta
- I thread real-time possono anticipare quelli non-real-time
 - Schema di priorità a 32 livelli e due classi
 - La classe variabile è nei livelli 1-15
 - La classe real-time è nei livelli 16-31
 - La priorità 0 è il thread di gestione della memoria
 - Coda per ogni priorità
 - Se nessun thread eseguibile, esegue il thread inattivo

PRIORITA' DELLE CLASSI WINDOWS

- L'API Win32 identifica diverse classi di priorità a cui può appartenere un processo (tutte variabili tranne REALTIME):
 1. REALTIME_PRIORITY_CLASS
 2. HIGH_PRIORITY_CLASS
 3. ABOVE_NORMAL_PRIORITY_CLASS
 4. NORMAL_PRIORITY_CLASS
 5. BELOW_NORMAL_PRIORITY_CLASS
 6. IDLE_PRIORITY_CLASS
- Un thread all'interno di una determinata classe di priorità ha una priorità relativa
 1. TIME_CRITICAL
 2. HIGHEST
 3. ABOVE_NORMAL
 4. NORMAL
 5. BELOW_NORMAL
 6. LOWEST
 7. IDLE
- La classe di priorità e la priorità relativa si combinano per dare priorità numerica
- La priorità di base è NORMALE all'interno della classe
- Se il quantum scade, la priorità viene ridotta, ma mai al di sotto della base.

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

- Se si verifica un'attesa, la priorità aumenta in base a ciò che è stato atteso.
- La finestra in primo piano ha ricevuto un aumento di priorità 3x.
- Aggiunta la pianificazione in modalità utente (UMS) di Windows 7:
 - Le applicazioni creano e gestiscono thread indipendentemente dal kernel
 - Per un gran numero di thread, molto più efficiente
 - Gli scheduler UMS provengono da librerie di linguaggi di programmazione come il framework C++ Concurrent Runtime (ConcRT).

SOLARIS

- SCHEDULING A PRIORITA'
- Sei classi disponibili:
 - Time sharing (default) (TS)

- Interactive (IA)
 - Real time (RT)
 - System (SYS)
 - Fair Share (FSS)
 - Fixed priority (FP)
- Il thread dato può essere in una classe alla volta
 - Ogni classe ha il proprio algoritmo di pianificazione
 - La condivisione del tempo è una coda di feedback a più livelli
 - Tabella caricabile configurabile da sysadmin

SCHEDULER SOLARIS

Lo scheduler converte le priorità specifiche della classe in una priorità globale per thread

- Il thread con la priorità più alta viene eseguito successivamente
- Funziona fino a:
 - blocchi
 - utilizza l'intervallo di tempo
 - anticipato da thread con priorità più alta
- Più thread con la stessa priorità selezionati tramite RR

VALUTAZIONE DEGLI ALGORITMI

- Come selezionare un algoritmo di scheduling della CPU per un particolare sistema operativo?
- Definizione dei criteri da usare per la scelta dell'algoritmo:
 - Esempio 1: massimizzare l'utilizzo della CPU, con tempo massimo di risposta inferiore a 300 millisecondi.
 - Esempio 2: massimizzare la produttività, con tempo di completamento proporzionale (in media) al tempo di esecuzione effettivo.
- Valutazione analitica - in base all'algoritmo ed al carico di lavoro del sistema, fornisce una formula per valutarne le prestazioni.
- Modelli deterministici - valutazione analitica, che considera un carico di lavoro predeterminato e definisce le prestazioni di ciascun algoritmo per quel carico di lavoro.

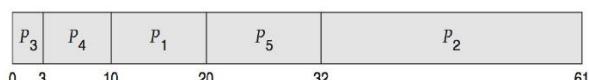
VALUTAZIONE DETERMINISTICA

- Per ciascun algoritmo si calcola il tempo medio di attesa
- Metodo semplice e veloce che richiede però una conoscenza delle dinamiche del sistema normalmente non disponibile.

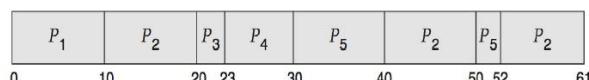
- FCS is 28ms:



- Non-preemptive SJF is 13ms:



- RR is 23ms:



QUEUEING MODELS (MODELLI IN CODA LETTERALMENTE) - RETI DI CODE

- Descrive l'arrivo di processi, CPU e burst di I/O probabilisticamente
 - Comunemente esponenziale e descritto per mezzo
 - Calcola il throughput medio, l'utilizzo, il tempo di attesa, ecc.
- Il sistema di calcolo viene descritto come una rete di server, ciascuno con una coda d'attesa:

- La CPU è un server con la propria coda dei processi pronti, ed il sistema di I/O ha le sue code dei dispositivi
- Se sono noti l'andamento degli arrivi e dei servizi (sotto forma di distribuzioni di probabilità), si può calcolare utilizzo di CPU e dispositivi, lunghezza media delle code, tempo medio d'attesa, throughput medio, etc...

LA FORMULA DI LITTLE

- n = lunghezza media di una coda
- W = tempo medio di attesa nella coda
- λ = andamento medio di arrivo dei nuovi processi
- Legge di Little: in regime stazionario, i processi in uscita dalla coda devono essere uguali ai processi in arrivo, quindi:

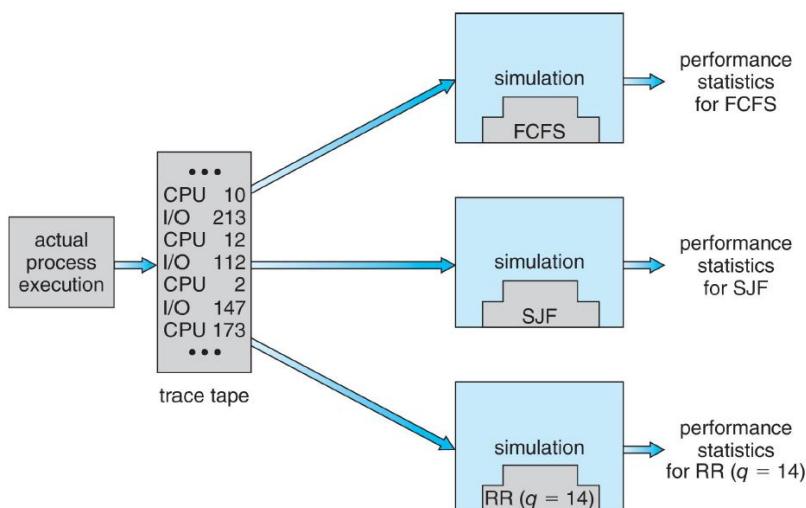
$$n = \lambda \times W$$

- È utilizzabile per il calcolo di una delle tre variabili, quando sono note le altre due per esempio, se ogni secondo arrivano 7 processi e la coda ne contiene mediamente 14 il tempo medio d'attesa per ogni processo è pari a 2 secondi

SIMULAZIONE

Simulazione - implica la programmazione di un modello del sistema di calcolo le strutture dati rappresentano gli elementi principali del sistema.

- Il simulatore dispone di una variabile che rappresenta il clock e modifica lo stato del sistema in modo dadescrire le attività dei dispositivi, dei processi e dello scheduler
- Durante l'esecuzione della simulazione, si raccolgono e si stampano statistiche che descrivono le prestazioni degli algoritmi
- I dati per la simulazione possono essere generati artificialmente, modellati da distribuzioni matematiche o empiriche o raccolti da un sistema reale mediante un trace tape.



IMPLEMENTAZIONE

Implementazione - implica la codifica effettiva degli algoritmi di scheduling da valutare, ed il loro inserimento nel sistema operativo, per osservarne il comportamento nelle condizioni reali di funzionamento del sistema.

- Difficoltà nel fare accettare agli utenti un sistema in continua evoluzione
- Alti costi, alti rischi
- Adeguamento del comportamento degli utenti alle caratteristiche dello scheduler
- Gli scheduler più flessibili possono essere modificati per-site o per-system.

CONSIDERAZIONI FINALI

- Gli algoritmi di scheduling più adattabili sono quelli che possono essere tarati dagli amministratori di sistema, che li adattano al particolare ambiente di calcolo, con la propria specifica gamma di applicazioni
- Molti degli attuali sistemi operativi UNIX-like forniscono all'amministratore la possibilità di calibrare i parametri di scheduling in vista di particolari configurazioni del sistema (es. Solaris).

--- CAPITOLO 6 - STRUMENTI DI SINCRONIZZAZIONE

OBBIETTIVI

- Descrivere il problema della sezione critica ed illustrare come avvengono le race condition.
- Mostrare soluzioni hardware alla mutua esclusione.
- Dimostrare come lock mutex, semafori, monitor e variabili condizionali possono essere utilizzati per la mutua esclusione.
- Valutare gli strumenti che risolvono il problema della sezione critica in scenari a bassa, moderata e alta contesa.

INTRODUZIONE

- Nei moderni SO, i task vengono eseguiti concorrentemente, su un core, e in parallelo, su core distinti.
- Un processo cooperante può influenzare gli altri processi in esecuzione nel sistema o subirne l'influenza
- I processi cooperanti possono...
 - condividere uno spazio logico di indirizzi, cioè codice e dati (memoria condivisa)
 - oppure solo dati, attraverso file o messaggi
- L'accesso concorrente a dati condivisi può causare inconsistenza nei dati
- Per garantire la coerenza (l'integrità) dei dati occorrono meccanismi che assicurano l'esecuzione ordinata dei processi cooperanti.
- I processi possono essere interrotti dallo scheduler in qualsiasi punto del loro flusso di esecuzione, per assegnare la CPU (il core) ad un altro processo.

RACE CONDITION

Race condition - Più processi accedono in concorrenza e modificano dati condivisi l'esito dell'esecuzione (il valore finale dei dati condivisi) dipende dall'ordine nel quale sono avvenuti gli accessi.

Per evitare le corse critiche occorre che i processi concorrenti vengano sincronizzati.

IL PROBLEMA DELLA SEZIONE CRITICA

- n processi {P0,P1,..., Pn} competono per utilizzare dati condivisi
- Ciascun processo è costituito da un segmento di codice, chiamato **sezione critica** (o regione critica), in cui accede a dati condivisi
- **Problema:** assicurarsi che, quando un processo accede alla propria sezione critica, a nessun altro processo sia concessa l'esecuzione di un'azione analoga.
- L'esecuzione di sezioni critiche da parte di processi cooperanti è mutuamente esclusiva nel tempo.

Soluzione → Progettare un protocollo di cooperazione fra processi:

- Ogni processo deve chiedere il permesso di accesso alla sezione critica, tramite una **entry section**.
- La sezione critica è seguita da una **exit section**
- Il rimanente codice è non critico

```
do {  
    entry section  
    /* critical section */  
    exit section  
    /* remainder section */
```

```
} while (true);
```

SOLUZIONE AL PROBLEMA DELLA SEZIONE CRITICA

Requisiti per la soluzione del problema della sezione critica:

1. **Mutua esclusione** → Se il processo Pi è in esecuzione nella sua sezione critica, nessun altro processo può eseguire la propria sezione critica.
2. **Progresso** → Se nessun processo è in esecuzione nella propria sezione critica ed esiste qualche processo che desidera accedervi, allora la selezione del processo che entrerà prossimamente nella propria sezione critica non può essere rimandata indefinitamente.
3. **Attesa limitata** → Se un processo ha effettuato la richiesta di ingresso nella sezione critica, è necessario porre un limite al numero di volte che si consente ad altri processi di entrare nelle proprie sezioni critiche, prima che la richiesta del primo processo sia stata accordata (politica fair per evitare la starvation).
 - Si assume che ciascun processo sia eseguito ad una velocità diversa da zero.
 - Non si fanno assunzioni sulla **velocità relativa** degli n processi.

SOLUZIONE BASATA SU INTERRUPT

- Sezione di ingresso: disabilita gli interrupt
- Sezione di uscita: abilita gli interrupt
- Questo risolverà il problema?
 - Cosa succede se la sezione critica è codice che viene eseguito per un'ora?
 - Alcuni processi possono - non entrare mai nella loro sezione critica.
 - Cosa succede se ci sono due CPU?

SOLUZIONE SOFTWARE 1

- Soluzione a due processi
- Si supponga che le istruzioni in linguaggio macchina di caricamento e memorizzazione siano atomiche; cioè, non possono essere interrotte.
- I due processi condividono una variabile:
 - int turn;
- La variabile turn indica a chi tocca entrare nella sezione critica
- Viene inizializzata turn = 1;

ALGORTIMO PER IL PROCESSO Pi

```
while (true){  
    while (turn == j)  
    ;  
    /* critical section */  
    turn = i;  
    /* remainder section */  
}
```

CORRETTEZZA DELLA SOLUZIONE SOFTWARE

- Viene preservata la mutua esclusione.
 - Pi entra nella sezione critica se e solo se:
turn == i
e turn non può essere contemporaneamente 0 e 1
- E il requisito di progresso?
 - Se il Processo 1 vuole entrare nella sezione critica e il Processo 2 non è interessato ad entrare nella sezione critica, il Processo 1 può entrare?
- Che dire del requisito di attesa vincolata?

ALGORTIMO PER IL PROCESSO Pi(2)

```

while (true) {
    turn = i;
    while (turn == j)
    ;
    /* critical section */
    turn = j;
    /* remainder section */
}

```

SOLUZIONE DI PETERSON

- Soluzione per due processi P₀ e P₁ (i=1-j)
- I due processi condividono due variabili
 - int turno
 - boolean flag[2]
- La variabile turno indica il processo che “è di turno” per accedere alla propria sezione critica
- L’array flag si usa per indicare se un processo è pronto ad entrare nella propria sezione critica
 - flag[i] = TRUE implica che il processo P_i è pronto per accedere alla propria sezione critica.

ALGORITMO PER IL PROCESSO P_i

```

while(true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
    sezione critica
    flag[i] = false;
    sezione non critica
}

```

Soluzione: Mutua esclusione garantita dal valore di turn; P_i entra nella sezione critica progresso al massimo dopo un ingresso da parte di P_j attesa limitata: alternanza stretta.

CORRETTEZZA DELLA SOLUZIONE DI PETERSON

- Viene dimostrato che i tre requisiti CS sono soddisfatti:

1. Viene preservata la mutua esclusione

P_i entra in CS solo se:
flag[j] == false o turn == i

2. Il requisito di progresso è soddisfatto
3. Il requisito dell'attesa limitata è soddisfatto.

SOLUZIONE DI PETERSON

- La soluzione funziona per 2 processi
- Che ne dici di modificarlo per gestire 10 processi?
- La soluzione richiede un **busy waiting**:
 - I processi sprecano i cicli della CPU per chiedere se possono entrare nella sezione critica

SOLUZIONE DI PETERSON E ARCHITETTURE MODERNE

- La soluzione di Peterson è basata sul software perché l’algoritmo garantisce la mutua esclusione senza richiedere alcun supporto speciale dal SO, né istruzioni hardware specifiche
- Sebbene rappresenti un utile approccio algoritmico, il funzionamento della soluzione di Peterson non è garantito sulle architetture attuali
- Capire perché l’approccio potrebbe non funzionare aiuta anche a comprendere le race condition
- Per migliorare le prestazioni, i processori e i compilatori possono riordinare le operazioni che non hanno interdipendenze.

- Nel caso di processi single-thread il risultato non cambia.
- Tuttavia, nel multithreading, il riordino può produrre risultati inconsistenti o inaspettati.

ESEMPIO:

- Due thread condividono i dati:

```
boolean flag = false;
int x = 0;
```

- Il thread 1 esegue:

```
while (!flag);
print x
```

- Mentre il thread 2 calcola:

```
x = 100;
flag = true
```

- Quale sarà l'output prodotto?

L'output dovrebbe essere 100 ma se le operazioni di thread 2 che sono indipendenti, vengono riordinate, si può stampare 0.

- Tuttavia, poiché le variabili flag e x sono indipendenti l'uno dall'altro, le istruzioni:

```
flag = true;
x = 100;
```

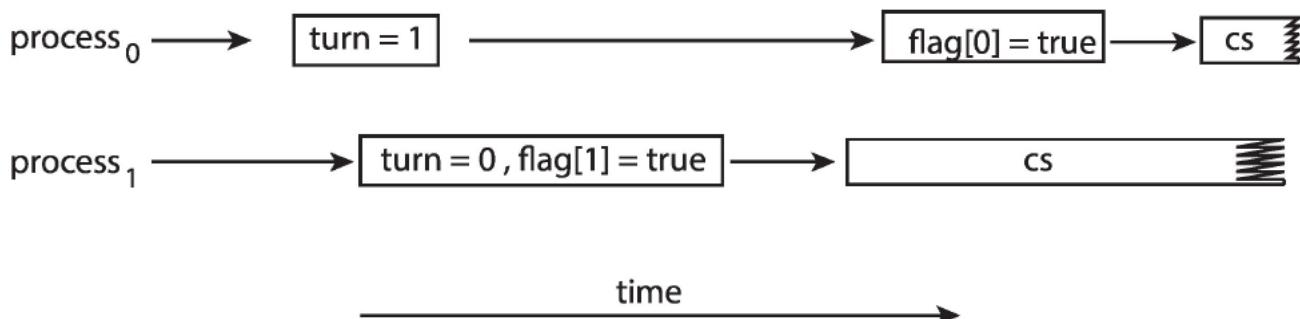
per il thread 2 può essere riordinato

- Se ciò si verifica, l'output potrebbe essere:

0

LA SOLUZIONE DI PETERSON RIVISITATA

- Effetto dello scambio di istruzioni nella soluzione di Peterson



- Questo consente ai processi l'accesso concorrente alla sezione critica
- Per garantire che la soluzione di Peterson funzioni correttamente sulla moderna architettura dei computer, dobbiamo utilizzare Memory Barrier (barriera di memoria).

HARDWARE DI SINCRONIZZAZIONE

- Molti sistemi forniscono supporto hardware per risolvere efficacemente il problema della sezione critica.
- In un sistema monoprocesso è sufficiente interdire le interruzioni mentre si modificano le variabili condivise.
 - Il codice attualmente in esecuzione viene eseguito senza possibilità di prelazione.
 - Soluzione inefficiente nei sistemi multiprocessore → SO non scalabili.
- Tre forme di supporto hardware

- Barriere di memoria
 - Istruzioni hardware
 - Variabili atomiche
- Queste primitive possono essere utilizzate direttamente come strumenti di sincronizzazione o costituire la base per costruire meccanismi di sincronizzazione più astratti.

ISTRUZIONI MEMORY BARRIER (BARRIERE DI MEMORIA)

- Il modello di memoria è il modo in cui l'architettura di un computer determina quali garanzie relative alla memoria vengono fornite ai programmi applicativi.
- Il modello di memoria può essere
 - Fortemente ordinato - se una modifica alla memoria di un processore è immediatamente visibile a tutti gli altri.
 - Debolemente ordinato - se una modifica alla memoria di un processore può non essere immediatamente visibile a tutti gli altri.
- Una barriera di memoria (o recinzione di memoria) è un'istruzione che forza la propagazione a tutti i processori di ogni cambiamento alla memoria.
- Viene utilizzata un'istruzione di barriera di memoria per garantire che tutte le istruzioni di caricamento e memorizzazione siano completate prima che vengano eseguite operazioni di caricamento o memorizzazione successive.
- Pertanto, anche se le istruzioni sono state riordinate, la barriera di memoria garantisce che le operazioni di memorizzazione siano completate in memoria e visibili ad altri processori prima che vengano eseguite future operazioni di caricamento o memorizzazione.

ESEMPIO MEMORY BARRIER (BARRIERE DI MEMORIA)

Relativamente all'esempio precedente, potremmo aggiungere una barriera di memoria sia in thread 1 che in thread 2 per garantire che thread 1 stampi il valore 100.

- thread 1:

```
while (!flag)
    memory_barrier();
print x
```

- thread 2:

```
x=100;
memory_barrier();
flag(true);
```

- Nella soluzione di Peterson, inserire una barriera di memoria fra le due istruzioni che costituiscono la entry section garantisce l'esecuzione ordinata dei processi.

ISTRUZIONI HARDWARE

- Istruzioni hardware speciali che ci consentono di testare e modificare il contenuto di una parola o di scambiare il contenuto di due parole in modo atomico (ininterrottamente):
 - Istruzioni Test-and-Set → permettono di controllare e modificare il contenuto di una parola di memoria.
 - Istruzioni Compare-and-Swap → permette di scambiare il contenuto di due parole di memoria.

ISTRUZIONE Test-and-Set

```
boolean test_and_set (boolean *target)
{
    boolean rv = *target;
    *target = true;
```

```

        return rv;
    }
}

```

Proprietà di TestAndSet:

- Eseguita atomicamente
- Ritorna il valore originale del parametro passato
- Pone il nuovo valore del parametro a TRUE

SOLUZIONE USANDO test_and_set()

- Variabile booleana condivisa lock, inizializzata a FALSE
- Soluzione:

```

do {
    while (test_and_set(&lock)
          ; /* do nothing */
          /* critical section */
          lock = false;
          /* remainder section */
    } while (true);
}

```

- Basato sul busy waiting.
- Risolve il problema della sezione critica?

ISTRUZIONE Compare-and-Swap

- Definizione:

```

int compare_and_swap(int *value, int expected, int new_value)
{
    int temp = *value;
    if (*value == expected)
        *value = new_value;
    return temp;
}

```

- Proprietà:

- Eseguita atomicamente
- Restituisce il valore originale del valore del parametro passato.
- Valorizza la variabile con il valore del parametro passato new_value, ma solo se la condizione *value == expected è vera. Cioè, lo scambio avviene solo in questa condizione.

SOLUZIONE USANDO compare_and_swap

- Variabile intera condivisa lock, inizializzata a FALSE
- Soluzione:

```

while (true) {
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */
        /* critical section */
        lock = 0;
        /* remainder section */
}

```

- Basato sul busy waiting.
- Risolve il problema della sezione critica?

VARIABILI ATOMICHE

- In genere, le istruzioni hardware (quelli di prima) per garantire la mutua esclusione vengono utilizzate come blocchi costruttivi per altri strumenti di sincronizzazione.
- Uno di questi sono le **variabili atomiche** normalmente definite su tipi di dati quali interi e booleani, che vengono aggiornate senza soluzione di continuità.
- I SO che supportano le variabili atomiche forniscono anche tipi speciali di dati atomici e funzioni per l'accesso e la manipolazione di variabili di tali tipi
- Le variabili atomiche sono comunemente utilizzate nei SO e nelle applicazioni concorrenti, ma il loro uso è spesso limitato ad aggiornamenti di singoli dati condivisi, come contatori e generatori random.
- Per esempio:
 - Sia una variabile atomica
 - Sia **increment()** un'operazione sulla variabile atomica **sequence**
 - Il comando:

```
increment(&sequence);
```

assicura che la **sequenza** venga incrementata senza interruzioni.

- La funzione **increment()** può essere implementata come segue:

```
int compare_and_swap(int *value, int expected, int new_value)
{
    int temp = *value;
    if (*value == expected)
        *value = new_value;
    return temp;
}

void increment	atomic_int *v)
{
    int temp;
    do {
        temp = *v;
    }
    while (temp != (compare_and_swap(v,temp,temp+1)));
}
```

LOCK MUTEX / MUTEX LOCK (NELLA SLIDE IN INGLESE)

- Le soluzioni hardware al problema della sezione critica sono complicate e generalmente inaccessibili ai programmati di applicazioni.
- I progettisti di SO hanno implementato strumenti software per risolvere il problema: il più semplice è il **lock mutex**.
 - Si definisce una variabile booleana **available** che indica se il lock è disponibile o meno.
- Ci sono due operazioni:
 - Si proteggono le regioni critiche prima acquisendo (**acquire()**) e quindi rilasciando (**release()**) il lock.
- Le chiamate ad **acquire()** e **release()** devono essere atomiche:
 - Normalmente vengono implementate tramite istruzioni (atomiche) hardware come compare-and-swap.
- La soluzione tramite lock impone l'attesa attiva (**busy waiting**) mentre un processo si trova nella propria sezione critica, ogni altro processo che ne tenti l'accesso deve ciclare effettuando la chiamata ad **acquire()**
 - Per questo motivo si parla anche di **spinlock**.

```
acquire() {
    while (!available)
        /* busy wait */
```

```

        available = false;
    }
    release() {
        available = true;
    }

    do {
        acquire()
            sezione critica
    release()
            sezione non critica
    } while (true);
}

```

SEMAFORI

- I semafori rappresentano un approccio più sofisticato alla sincronizzazione fra processi
- Il semaforo S è una variabile intera
- Si può accedere al semaforo S (dopo l'inizializzazione) solo attraverso due operazioni indivisibili (operazioni atomiche, primitive) wait() e signal()
- Definizione dell'operazione wait():

```

wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}

```

- Definizione dell'operazione signal():

```

signal(S){
    S++;
}

```

- **Semaforo contatore:** intero che può assumere valori in un dominio non limitato
- **Semaforo binario:** assume soltanto i valori 0 e 1, come il lock mutex.
- Si può implementare un semaforo contatore S utilizzando semafori binari.
- Permettono l'approccio a diversi problemi di sincronizzazione.

ESEMPIO USO SEMAFORO

- Soluzione al problema CS
 - Creare un semaforo “**mutex**” inizializzato a 1
 - Codice:

```

wait(mutex);
    CS
signal(mutex);

```

- Si considerino i processi P1 e P2 che con due statement (blocco di istruzioni) S1 e S2 e il requisito che S1 avvenga prima di S2:
 - Creare un semaforo “**synch**” inizializzato a 0.
 - Codice:

P1:

```

S1;
signal(synch);

```

P2:

```

wait(synch);
S2;

```

IMPLEMENTAZIONE DEI SEMAFORI

- Dato che occorre garantire che due processi non eseguano mai **wait()** e **signal()** sullo stesso semaforo allo stesso istante (ovvero occorre garantire l'atomicità delle due operazioni)
 - L'implementazione dei semafori impone che **wait()** e **signal()** vengano eseguite esse stesse all'interno di una sezione critica → possibile il **busy waiting**
 - Codice di implementazione molto ridotto
 - Brevi attese se la sezione critica è raramente occupata
- Viceversa, con i semafori spinlock, si crea una situazione di busy waiting (attesa attiva) dei processi in attesa ad un semaforo.
 - Poiché i processi possono passare anche molto tempo all'interno delle loro sezioni critiche, il semaforo spinlock può provocare notevole spreco di CPU da parte dei processi in attesa sullo stesso semaforo.
- Si noti che per le applicazioni "normali", in cui le applicazioni possono trascorrere molto tempo in sezioni critiche, questa non è una buona soluzione.

IMPLEMENTAZIONE SENZA BUSY WAITING

- Ad ogni semaforo è associata una coda di attesa (waiting queue)
- Ogni voce in una coda di attesa ha due elementi di dati:
 - Valore (di tipo intero)
 - Puntatore al record successivo nell'elenco
- Coda di attesa (waiting queue):

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

IMPLEMENTAZIONE DELL'OPERAZIONE WAIT

- L'operazione **wait**:

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        sleep();
    }
}
```

- **sleep()** sospende il processo che lo ha invocato

IMPLEMENTAZIONE DELL'OPERAZIONE SIGNAL

- L'operazione **signal**:

```
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

- L'operazione **wakeup(P)** riprende l'esecuzione del processo P

PROBLEMI CON I SEMAFORI

- Uso scorretto delle operazioni del semaforo:
 - **signal(mutex) ... wait(mutex)**
 - **wait(mutex) ... wait(mutex)**

- Omitting of wait(mutex) and/or signal(mutex)
- Questi - e altri - sono esempi di cosa può accadere quando i semafori e altri strumenti di sincronizzazione vengono utilizzati in modo errato.
- Soluzione: introdurre costrutti di programmazione di alto livello.

BUSY WAITING

- In altre parole... Mentre un processo si trova nella propria sezione critica, qualsiasi altro processo che tenti di entrare nella sezione critica si trova nel ciclo del codice della entry section
- Il **busy waiting o attesa attiva** costituisce un problema per i sistemi multiprogrammati, perché la condizione di attesa spreca cicli di CPU che altri processi potrebbero sfruttare produttivamente.
- L'attesa attiva è vantaggiosa solo quando è molto breve perché, in questo caso, può evitare i context-switch.

I MONITOR

- Il monitor è un costrutto di alto livello (un ADT per Abstract Data Type) che fornisce un meccanismo efficiente per la sincronizzazione dei processi.
- Incapsula dati mettendo a disposizione metodi per operare su di essi
- Tipo di dati astratto, variabili interne accessibili solo da codice all'interno della procedura.
- All'interno del monitor può essere attivo un solo processo alla volta.
- La rappresentazione di un tipo monitor è costituita da:
 - dichiarazioni di variabili i cui valori definiscono lo stato di un'istanza del tipo
 - procedure o funzioni che realizzano le operazioni sulle variabili definite nel monitor.
- Supportati (implementati in certa misura) da C#, Java, Concurrent Pascal

```

monitor monitor name
{
    // dichiarazione delle variabili condivise
    function P 1 (...) {
        ...
    }
    function P 2 (...) {
        ...
    }
    function P n (...) {
        ...
    }
    {
        // codice di inizializzazione
    }
}

```

LE VARIABILI CONDITION

- Per permettere ad un processo di attendere dopo l'ingresso al monitor, causa occupazione della risorsa richiesta, si dichiarano apposite variabili condition.
- **condition x, y;**
- Le variabili condition possono essere usate solo in relazione a specifiche operazioni wait() e signal().
- L'operazione:

x.wait();

fa sì che il processo chiamante rimanga sospeso fino a che un diverso processo non effettui la chiamata:

- Mentre l'operazione

```
x.signal();
```

L'operazione x.signal() attiva esattamente un processo sospeso; se non esistono processi sospesi l'operazione signal() non ha alcun effetto

SCELTA DELLE VARIABILI CONDITION

- Se il processo P invoca **signal()** su una variabile condition ed esiste un processo Q in coda su quella variabile, cosa accadrebbe?
 - Sia Q che P non possono essere eseguiti in parallelo. Se Q viene ripreso, P deve attendere
- Le possibilità sono:
 - **Signal and wait** (Segnalare ed attendere) - nel monitor entra Q e P attende (fuori dal monitor) che Q abbia terminato o si metta in attesa su una condition.
 - **Signal and continue** (Segnalare e proseguire) - Q attende che P lasci il monitor o si metta in attesa su una condition.
- Entrambi hanno dei pro e dei contro:
 - Dato che P era già in esecuzione nel monitor segnalare e proseguire sembra più ragionevole.
 - Però... quando P termina, la condizione attesa da Q potrebbe non essere più valida.
- Soluzione adottata da Concurrent Pascal: P deve eseguire la signal() come ultima istruzione prima di uscire dal monitor
- I Monitor sono implementati in altri linguaggi come Mesa, C# e Java

- Variables

```
semaphore mutex;          // initially = 1
semaphore next;           // initially = 0
int next_count = 0;        // number of processes
                          // waiting inside the
                          // monitor
```

- Each function **F** will be replaced by

```
wait(mutex);
...
body of F;
...
if (next_count > 0)
    signal(next)
else
    signal(mutex);
```

- Mutual exclusion within a monitor is ensured

- For each condition variable **x**, we have:

```
semaphore x_sem;      // initially = 0
int x_count = 0;
```

- The operation **x.wait()** can be implemented as:

```
x_count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x_count--;
```

- The operation **x.signal()** can be implemented as:

```
if (x_count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```

RIPRENDERE I PROCESSI ALL'INTERNO DI MONITOR

- Se più processi sono accodati sulla variabile condition x, e x.signal() viene eseguito, quale processo dovrebbe essere ripreso?
- FCFS spesso non adeguato

- costrutto **conditional-wait** della forma `x.wait(c)` :
 - Dove c'è il **numero di priorità**
 - Il processo con il numero più basso (priorità più alta) è programmato il prossimo.

ESEMPIO DI ALLOCAZIONE DI RISORSE SINGOLE

- Allocare una singola risorsa tra processi concorrenti utilizzando numeri di priorità che specificano la quantità massima di tempo che un processo prevede di utilizzare la risorsa.
- Dove `R` è un'istanza del tipo **ResourceAllocator** il protocollo di accesso al monitor per l'uso della risorsa per un tempo `t` è:

```
R.acquire(t);
...
access the resource;
...
R.release;
```

UN MONITOR PER ALLOCARE UNA SINGOLA RISORSA

```
monitor ResourceAllocator {
    boolean busy;
    condition x;
    void acquire(int time) {
        if (busy)
            x.wait(time);
        busy = TRUE;
    }

    void release() {
        busy = FALSE;
        x.signal();
    }

    initialization code() {
        busy = FALSE;
    }
}
```

- Utilizzo:


```
acquire
...
release
```
- Uso scorretto delle operazioni di monitoraggio:
 - `release() ... acquire()`
 - `acquire() ... acquire()`
 - Omissione di `acquire()` e/o `release()`

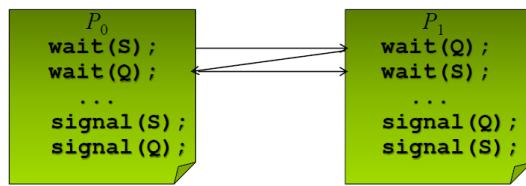
LIVENESS

- I processi potrebbero dover attendere indefinitamente durante il tentativo di acquisire uno strumento di sincronizzazione come un lock mutex o un semaforo
- L'attesa indefinita viola i requisiti di progresso e attesa limitata
- Il concetto di **liveness** fa riferimento a un insieme di proprietà che un sistema deve soddisfare per garantire che i processi progrediscano
- L'attesa indefinita è un esempio di fallimento della liveness

DEADLOCK

- La realizzazione di un semaforo con coda di attesa può condurre a situazioni in cui ciascun processo attende l'esecuzione di un'operazione `signal()` che solo uno degli altri processi in coda può causare.

- Più in generale, si verifica una situazione di **deadlock** o stallo quando due o più processi attendono indefinitamente un evento che può essere causato soltanto da uno dei processi in attesa.
- Siano S e Q due semafori inizializzati entrambi ad 1



- Considera se P0 esegue wait(S) e P1 wait(Q). Quando P0 esegue wait(Q), deve attendere che P1 esegua il segnale(Q).
- Tuttavia, P1 attende fino a quando P0 esegue il segnale(S)
- Poiché queste operazioni signal() non verranno mai eseguite, P0 e P1 sono in **deadlock**.

ALTRE FORME DI DEADLOCK

- **Starvation** - Un processo può attendere per un tempo indefinito nella coda di un semaforo senza venir mai rimosso.
- **Inversione della priorità**:
 - Quando: un processo a priorità più alta ha bisogno di leggere o modificare dati a livello kernel utilizzati da un processo, o da una catena di processi, a priorità più bassa.
 - Perché: poiché i dati a livello kernel sono tipicamente protetti da un lock, il processo a priorità maggiore dovrà attendere finché il processo a priorità minore non avrà finito di utilizzare le risorse.

PROTOCOLLO DI EREDITÀ PRIORITARIA

- Considera lo scenario con tre processi P1, P2 e P3. P1 ha la priorità più alta, P2 la seconda più alta e P3 la più bassa.
- Si suppone che a P3 sia assegnata una risorsa R che P1 desidera:
 - Pertanto, P1 deve attendere che P3 finisca di utilizzare la risorsa. Tuttavia, P2 diventa eseguibile e prevale su P3.
 - Quello che è successo è che P2 - un processo con una priorità inferiore a P1 - ha indirettamente impedito a P1 di accedere alla risorsa.
- Per evitare che ciò accada, viene utilizzato un **protocollo di eredità prioritaria**. Ciò consente semplicemente di assegnare la priorità del thread più alto in attesa di accedere a una risorsa condivisa al thread che sta attualmente utilizzando la risorsa. Pertanto, all'attuale proprietario della risorsa viene assegnata la priorità del thread con la priorità più alta che desidera acquisire la risorsa.

---- CAPITOLO 7A : ESEMPI DI SINCRONIZZAZIONE ----

OBBIETTIVI

- Illustrare il problema della sincronizzazione del buffer limitato
- Spiegare il problema di sincronizzazione dei lettori-scrittori
- Spiegare e risolvere i problemi di sincronizzazione tra ristoranti e filosofi

PROBLEMI CLASSICI DI SINCRONIZZAZIONE

- Problema del produttore-consumatore con buffer limitato.
 - Problema dei lettori-scrittori
 - Problema dei cinque filosofi
- ➔ Esempi di una vasta classe di problemi di controllo della concorrenza.

PROBLEMA DEL BUFFER LIMITATO

- Variabili condivise:
 - intero **n** → Numero di buffer, ciascuno può contenere un elemento.
 - Semaforo **mutex** inizializzato a 1

- Semaforo **full** inizializzato a 0
- Semaforo **empty** inizializzato a n
- Il buffer ha n posizioni, ciascuna in grado di contenere un elemento
- Il semaforo mutex garantisce la mutua esclusione sugli accessi al buffer
- I semafori empty e full contano, rispettivamente, il numero di posizioni vuote ed il numero di posizioni piene nel buffer.

PROBLEMA DEI LETTORI-SCRITTORI

Un insieme di dati (ad es un file, una base di dati, etc) deve essere condiviso tra più processi concorrenti che possono

- richiedere la sola **lettura** del contenuto
- o l'accesso ai dati sia in **lettura** che in **scrittura**
- Problema: permettere a più lettori di accedere ai dati contemporaneamente solo uno scrittore, viceversa, può accedere ai dati condivisi (accesso esclusivo).
- Variabili condivise (oltre ai dati):

```
semafori mutex, rw_mutex
int read_count
inizializzazioni : mutex=1 rw_mutex=1 read_count=0
```

```
while(true) {
    wait(rw_mutex);
    ...
    /* scrittura */
    ...
    signal(rw_mutex);
}
```

Scrittore

```
while(true) {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);
    signal(mutex);

    ...
    /* lettura */

    wait(mutex);
    read_count--;
    if (read_count == 0)
        signal(rw_mutex);
    signal(mutex);
}
```

Lettore

- Il semaforo **rw_mutex** è comune ad entrambi i tipi di processi
- Il semaforo **mutex** serve per assicurare la mutua esclusione all'atto dell'aggiornamento (o del controllo del valore) di **read_count**
- La variabile **read_count** contiene il numero dei processi che stanno attualmente leggendo l'insieme di dati
- Il semaforo **rw_mutex** realizza la mutua esclusione per gli scrittori e serve anche al primo ed all'ultimo lettore che entra/esce dalla sezione critica
 - Non serve ai lettori che accedono alla lettura in presenza di altri lettori
- Diverse varianti: la soluzione proposta riguarda il primo problema dei lettori- scrittori, in cui si richiede che nessun lettore attenda, almeno che uno scrittore abbia già ottenuto l'accesso ai dati condivisi.
 - Priorità ai lettori → gli scrittori possono essere soggetti a starvation
- Il secondo problema dei lettori-scrittori richiede invece che uno scrittore, se pronto, esegua il proprio compito di scrittura al più presto; in altre parole, se uno scrittore attende l'accesso all'insieme di dati, nessun nuovo lettore deve iniziare la lettura
 - Priorità agli scrittori → i lettori possono essere soggetti a starvation

PROBLEMA DEI CINQUE FILOSOFI

- Cinque filosofi passano la vita pensando e mangiando, attorno ad una tavola rotonda.
- Non interagiscono con i loro vicini; occasionalmente tentano di procurarsi due bacchette per mangiare.
- Al centro della tavola vi è una zuppiera di riso e la tavola è apparecchiata con cinque bacchette.

- Quindi quando un filosofo pensa, non interagisce con i colleghi, quando gli viene fame, tenta di impossessarsi delle bacchette che stanno alla sua destra ed alla sua sinistra.
- Il filosofo può appropriarsi di una sola bacchetta alla volta e non può strapparla dalle mani di un vicino.
- Quando un filosofo affamato possiede due bacchette contemporaneamente, mangia terminato il pasto, appoggia le bacchette e ricomincia a pensare.
- Variabili condivise:
 - La zuppiera di riso (l'insieme dei dati)
 - Cinque semafori, bacchetta[5], inizializzati ad 1.

Codice per l'i-esimo filosofo

```

while(true) {
    wait (bacchetta[i]);
    wait (bacchetta[(i + 1) % 5]);
        // mangia
    signal (bacchetta[i]);
    signal (bacchetta[(i + 1) % 5]);
        // pensa
}

```

- Soluzione che garantisce che due vicini non mangino contemporaneamente, ma non riesce ad evitare lo stallo.
- Stallo: i filosofi hanno fame simultaneamente ed ognuno si impossessa della bacchetta alla propria sinistra.
- Soluzioni:
 - Solo quattro filosofi possono stare a tavola contemporaneamente
 - Un filosofo può prendere le bacchette solo se sono entrambe disponibili (operazione da eseguire all'interno di una sezione critica).
 - I filosofi di posto "oari" raccolgono prima la bacchetta alla loro sinistra, quelli di posto "dispari" la bacchetta alla loro destra.

```

monitor filosofo
{
    enum{PENSA, AFFAMATO, MANGIA} stato[5];
    condition self[5];

    void prende (int i) {
        stato[i] = AFFAMATO;
        test(i);
        if (stato[i] != MANGIA) self[i].wait;
    }

    void posa (int i) {
        stato[i] = PENSA;
        // controlla i vicini a sinistra e a destra
        test((i+4) % 5);
        test((i+1) % 5);
    }
}

```

```

void test (int i) {
    if ( (stato[(i+4) % 5] != MANGIA) &&
        (stato[i] == AFFAMATO) &&
        (stato[(i+1) % 5] != MANGIA) ) {
            stato[i] = MANGIA;
            self[i].signal();
        }
}

codice_iniz() {
    for (int i = 0; i < 5; i++)
        stato[i] = PENSA;
}
}

```

- Il filosofo i-esimo invocherà le operazioni prende() e posa() nell'ordine

filosofo prende(i)
 mangia
 filosofo posa(i)

- La soluzione non provoca deadlock, ma la starvation è ancora possibile!

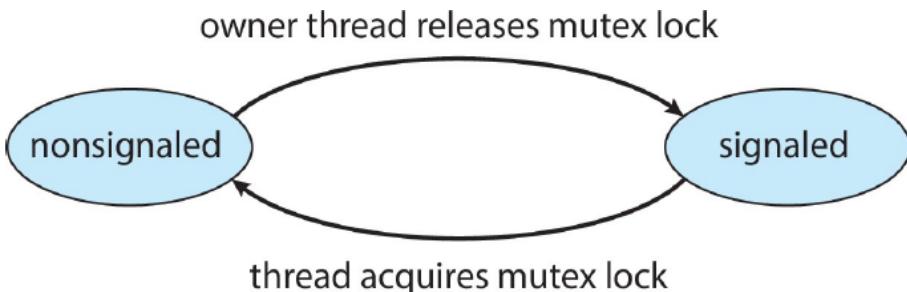
---- CAPITOLO 7B: ESEMPI DI SINCRONIZZAZIONE AVANZATA ----

OBBIETTIVI

- Descrivi gli strumenti utilizzati da Linux e Windows per risolvere i problemi di sincronizzazione
- Illustrare come POSIX e Java possono essere utilizzati per risolvere i problemi di sincronizzazione dei processi

SINCRONIZZAZIONE DEL KERNEL - WINDOWS

- Utilizza le maschere di interruzione per proteggere l'accesso alle risorse globali sui sistemi monoprocessoressi
- Utilizza **spinlock** su sistemi multiprocessore
 - Il filo di spinlock non sarà mai anticipato
- Fornisce anche **oggetti dispatcher** user-land che possono agire su mutex, semafori, eventi e timer:
 - **Eventi** → Un evento agisce in modo molto simile a una variabile di condizione
 - **Timers** → Avvisare uno o più thread quando il tempo è scaduto.
 - Oggetti Dispatcher in **stato segnalato** (oggetto disponibile) o stato **non segnalato** (il thread si bloccherà)
- Oggetto dispatcher Mutex



SINCRONIZZAZIONE IN WINDOWS

- Prima della versione 2.6 Linux adoperava un kernel senza diritto di prelazione
 - Gli interrupt venivano disabilitati per ottenere sezioni critiche di breve durata.
- Attualmente, Linux fornisce diversi meccanismi per la sincronizzazione nel kernel:
 - Lock mutex una variabile booleana che indica se il lock è disponibile, modificata tramite le chiamate di sistema mutex_lock() e mutex_unlock()
 - Spinlock e semafori (con variante lettore-scrittore)
 - Interi atomici rappresentati mediante il tipo opaco atomic_t
 - Versioni reader-writer di entrambi
- Sul sistema a CPU singola, gli spinlock sono stati sostituiti abilitando e disabilitando la prelazione del kernel (kernel preemption).

SINCRONIZZAZIONE POSIX

- La API Pthreads (lo standard POSIX) fornisce diversi tipi di lock e variabili condizionali per la sincronizzazione dei thread a livello utente.
- Ampiamente usato su UNIX, Linux e macOS.
- I lock mutex rappresentano la tecnica di sincronizzazione fondamentale
- POSIX prevede due versioni di semafori (definiti nell'estensione POSIX SEM), **con e senza nome**.
 - I semafori con nome possono essere utilizzati da processi non correlati, diversamente da quelli senza nome.
- Creazione e inizializzazione del lock:

```
#include <pthread.h>
pthread_mutex_t mutex;
/* create and initialize the lock mutex */
pthread_mutex_init(&mutex,NULL);
```

- Acquisizione e rilascio del lock:

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);
/* critical section */
/* release the lock mutex */
pthread_mutex_unlock(&mutex);
```

SEMAFORI CON NOME IN POSIX

- Creazione e inizializzazione del semaforo:

```
#include <semaphore.h>
sem_t *sem;
/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
```

- Un diverso processo può riferirsi al semaforo usandone il nome, SEM
- Acquisizione e rilascio del semaforo

```
/* acquire the semaphore */
sem_wait(sem);
/* critical section */
/* release the semaphore */
sem_post(sem);
```

SEMAFORI SENZA NOME IN POSIX

- Creazione e inizializzazione del semaforo:

```
#include <semaphore.h>
sem_t sem;
/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
```

- Un diverso processo può riferirsi al semaforo usandone il nome, SEM
- Acquisizione e rilascio del semaforo

```
/* acquire the semaphore */
sem_wait(&sem);
/* critical section */
/* release the semaphore */
sem_post(&sem);
```

VARIABILI CONDITION IN POSIX

- Poiché POSIX è tipicamente utilizzato in C/C++ che non forniscono il tipo monitor, le variabili condition POSIX sono associate ai lock mutex per garantire la mutua esclusione
- Creazione e inizializzazione della variabile condition

```
pthread_mutex_t mutex;
pthread_cond_t cond_var;
```

```
pthread_mutex_init(&mutex,NULL);
pthread_cond_init(&cond_var,NULL);
```

- Thread in attesa che valga a==b:

```
pthread_mutex_lock(&mutex);
while (a != b)
pthread_cond_wait(&cond_var, &mutex);
pthread_mutex_unlock(&mutex);
```

- Thread che "sveglia" un altro thread in attesa sulla variabile condition:

```
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

SINCRONIZZAZIONE JAVA

Java fornisce un ricco set di funzionalità di sincronizzazione:

- Monitor Java
- Serrature rientranti
- Semafori
- Variabili condition

MONITOR JAVA

Ad ogni oggetto Java è associato un singolo blocco

- Se un metodo viene dichiarato **sincronizzato**, un thread chiamante deve possedere il blocco per l'oggetto
- Se il blocco è di proprietà di un altro thread, il thread chiamante deve attendere il blocco finché non viene rilasciato
- I blocchi vengono rilasciati quando il thread proprietario esce dal metodo **sincronizzato**

APPROCI ALTERNATIVI

- Memoria transazionale
- OpenMP
- Linguaggi di programmazione funzionale

MEMORIA TRANSAZIONALE

- Considera una funzione update() che deve essere chiamata atomicamente.

Un'opzione è usare i blocchi mutex:

```
void update () {
    acquire();
    /* modify shared data */
    release();
}
```

- Una transazione di memoria è una sequenza di operazioni di lettura-scrittura nella memoria eseguite in modo atomico. Una transazione può essere completata aggiungendo atomic{S} che assicura che le istruzioni in S vengano eseguite atomicamente:

```
void update () {
    atomic {
    /* modify shared data */
```

```
        }  
    }
```

OpenMP

- OpenMP è un insieme di direttive del compilatore e API che supportano la programmazione parallela.
- Include la direttiva → **#pragma omp critical** che specifica che la successiva regione di codice è una sezione critica in cui solo un thread alla volta può essere attivo.

```
void update(int value)  
{  
    #pragma omp critical  
    {  
        count += value  
    }  
}
```

LINGUAGGI DI PROGRAMMAZIONE FUNZIONALI

- La differenza fondamentale fra linguaggi procedurali e funzionali è che i linguaggi funzionali non mantengono uno stato.
- Una volta che una variabile è stata definita e inizializzata il suo valore è immutabile e non cambia stato → no race condition
- C'è un crescente interesse per i linguaggi funzionali come Erlang e Scala per il loro approccio.

---- CAPITOLO 8 : DEADLOCK ----

OBBIETTIVI

- Illustrare come e quando può verificarsi il deadlock
- Definire le quattro condizioni necessarie che lo caratterizzano
- Identificare il deadlock in un grafo di allocazione delle risorse
- Valutare diversi approcci per prevenire il deadlock
- Applicare l'algoritmo del Banchiere per evitare il deadlock
- Applicare l'algoritmo di rilevamento dei deadlock
- Valutare gli approcci per il recupero da situazioni di deadlock

IL PROBLEMA DEL DEADLOCK

- In un ambiente multiprogrammato più thread possono entrare in competizione per ottenere un numero finito di risorse.
- Deadlock o stallo - insieme di thread bloccati, ciascuno dei quali “possiede” (almeno) una risorsa ed attende di acquisire una o più risorse allocate ad altri thread dell'insieme (che detengono risorse, ma ne attendono altre)
- Nella programmazione multithread, gli stalli si verificano più frequentemente, poiché più thread possono competere per un insieme (limitato) di risorse condivise

MODELLO DI SISTEMA

- Il sistema è costituito da risorse
- Tipi di risorse R1, R2, ..., Rm
 - Cicli di CPU, spazio di memoria, file, dispositivi di I/O, lock, semafori
- Ciascun tipo di risorsa Ri ha Wi istanze
- Ogni processo utilizza una risorsa come segue:
 - **Richiesta**
 - **Utilizzo**
 - **Rilascio**

ESEMPIO DEADLOCK CON DUE SEMAFORI

Semafori A e B, inizializzati a 1:

T1

- 1) wait (A);
- 2) wait(B);
- 3) wait (B);
- 4) wait(A);

T2

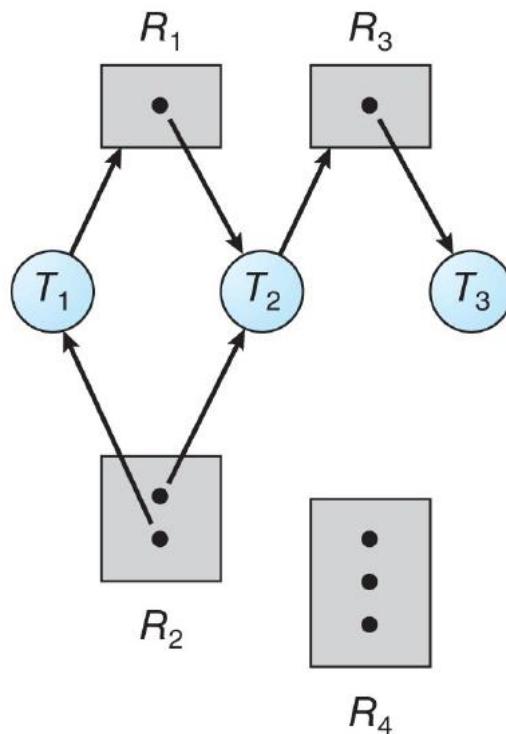
CARATTERIZZAZIONE DEL DEADLOCK

Una situazione di deadlock può verificarsi solo se valgono simultaneamente le seguenti condizioni:

- **Mutua esclusione** nel sistema, esiste almeno una risorsa non condivisibile, cioè utilizzabile da un solo thread alla volta.
- **Possesso ed attesa un thread**, che possiede almeno una risorsa, attende di acquisire ulteriori risorse possedute da altri thread.
- **Impossibilità di prelazione** una risorsa può essere rilasciata dal thread che la detiene solo volontariamente, al termine del suo utilizzo.
- **Attesa circolare** esiste un insieme $\{T_0, T_1, \dots, T_n\}$ di thread in attesa, tali che T_0 è in attesa di una risorsa che è posseduta da T_1 , T_1 è in attesa di una risorsa posseduta da $\{T_2, \dots, T_{(n-1)}\}$ è in attesa di una risorsa posseduta da T_n e T_n è in attesa di una risorsa posseduta da T_0 .

GRAFO DI ASSEGNAZIONE DELLE RISORSE

- Un grafo è costituito da un insieme di vertici (o nodi) V variamente connessi per mezzo di un insieme di archi E .
- Nel grafo di assegnazione delle risorse...
 - L'insieme V è partizionato in due sottoinsiemi:
 - $P = \{P_1, P_2, \dots, P_n\}$ è l'insieme costituito da tutti i thread attivi nel sistema
 - $R = \{R_1, R_2, \dots, R_n\}$, è l'insieme di tutti i tipi di risorse presenti nel sistema
 - **Arco di richiesta** – directed edge $P_i \rightarrow R_j$
 - **Arco di assegnazione** – directed edge $R_j \rightarrow P_i$



- Se il grafo non contiene cicli \rightarrow Non ci sono deadlock
- Se il grafo contiene un ciclo:
 - Se vi è una sola istanza per ogni tipo di risorsa, allora si ha un deadlock
 - Se si hanno più istanze per tipo di risorsa, allora il deadlock è possibile (ma non certo)

METODI DI GESTIONE DEL DEADLOCK

Assicurare che il sistema non entri **mai** in uno stato di deadlock

- Prevenire i deadlock → evitare che si verifichino contemporaneamente mutua esclusione, possesso e attesa, impossibilità di prelazione e attesa circolare.
 - Basso utilizzo delle risorse e throughput ridotto
- Evitare i deadlock evitare gli stati del sistema a partire dai quali si può evolvere verso il deadlock
 - Informazioni aggiuntive sulle richieste dei thread
- Permettere al sistema di entrare in uno stato di deadlock, quindi ripristinare il sistema.
- Ignorare il problema e fingere che i deadlock non si verifichino mai

PREVENIRE I DEADLOCK

- Limitare le modalità di richiesta/accesso delle/alle risorse per invalidare una delle quattro condizioni necessarie:

- **Mutua esclusione** → non è richiesta per risorse condivisibili → non possono essere coinvolte in uno stallo deve valere invece per risorse che non possono essere condivise, quindi non può essere prevenuta.
- **Possesso e attesa** → occorre garantire che, quando un thread richiede una risorsa, non ne possieda altre.
- Esigere dal thread di stabilire ed allocare tutte le risorse necessarie prima che inizi l'esecuzione, o consentire la richiesta di risorse solo quando il thread non ne possiede alcuna
- Basso impiego delle risorse possibile l'attesa indefinita per thread con forti richieste di risorse.
- **Impossibilità di prelazione:**
 - Se un thread, che possiede alcune risorse, richiede un'altra risorsa, che non gli può essere allocata immediatamente, allora rilascia tutte le risorse possedute
 - Le risorse rilasciate (prelazionate al thread) vengono aggiunte alla lista delle risorse che il thread sta attendendo
 - Il thread viene avviato nuovamente solo quando può ottenere sia le risorse precedentemente possedute sia quelle attualmente richieste
 - Alternativamente, le risorse possono essere assegnate al thread che le richiede, qualora siano attualmente possedute da un altro thread in attesa
 - Protocollo adatto per risorse il cui stato si può salvare e recuperare facilmente memoria, non per dispositivi di I/O o primitive di sincronizzazione
- **Attesa circolare** → si impone un ordinamento totale (possibilmente "logico") su tutti i tipi di risorsa, f: R→N e si pretende che ciascun thread richieda le risorse in ordine crescente

ATTESA CIRCOLARE

- L'annullamento della condizione di attesa circolare è più comunemente utilizzato
- Assegnare semplicemente a ciascuna risorsa (ad es. blocchi mutex) un numero univoco
- Le risorse devono essere acquisite in ordine

EVITARE I DEADLOCK

- Prevenire i deadlock può causare scarso utilizzo dei dispositivi e ridotta produttività del sistema.
- Viceversa, evitare i deadlock presuppone che il sistema conosca **a priori** informazioni addizionali sulle richieste future dei thread.
 - Il modello più semplice e utile richiede che ciascun thread dichiari il **numero massimo** di risorse di ciascun tipo di cui potrà usufruire nel corso della propria esecuzione.
 - L'algoritmo di deadlock-avoidance esamina dinamicamente lo stato di allocazione delle risorse per assicurarsi che non si possa verificare una condizione di attesa circolare
 - Lo stato di allocazione delle risorse è definito dal numero di risorse disponibili ed allocate, e dal massimo numero di richieste dei thread.

STATO SICURO

- Quando un thread richiede una risorsa disponibile, il sistema deve decidere se l'allocazione immediata lasci il sistema in stato sicuro.
- Il sistema si trova in uno **stato sicuro** se e solo se esiste una sequenza sicura di esecuzione di tutti i thread

- La sequenza $\{T_1, T_2, \dots, T_n\}$ è sicura se, per ogni T_i le risorse che T_i può ancora richiedere possono essergli allocate sfruttando le risorse disponibili, più le risorse possedute da tutti i T_j con $j < i$
- Cioè...
 - Se le richieste di T_i non possono essere soddisfatte immediatamente, allora T_i può attendere finché tutti i T_j ($j < i$) abbiano terminato.
 - Quando i T_j hanno terminato, T_i può ottenere le risorse richieste, eseguire i suoi compiti, restituire le risorse allocate e terminare
 - Quando T_i termina, $T_{(i+1)}$ può ottenere le risorse richieste, etc...

FATTI BASILARI

- Se un sistema è in stato sicuro \rightarrow non si evolve verso il deadlock
- Se un sistema è in stato non sicuro \rightarrow si può evolvere in deadlock
- Algoritmo di evitamento:
 - Assicurarsi che un sistema non entri mai in uno stato non sicuro

ALGORITMI PER EVITARE IL DEADLOCK

- Risorse con istanza singola \rightarrow utilizzo del grafo di assegnazione delle risorse (versione modificata)
- Risorse con istanza multipla \rightarrow algoritmo del banchiere

ALGORITMO CON GRAFO DI ASSEGNAZIONE DELLE RISORSE

- Un **arco di reclamo** $T_i \rightarrow R_j$ (rappresentato da una linea tratteggiata) indica che il thread T_i può richiedere la risorsa R_j in futuro.
- Un **arco di reclamo** viene convertito in un arco di richiesta quando il thread T_i richiede effettivamente la risorsa R_j
- Un **arco di richiesta** viene convertito in un arco di assegnazione quando la risorsa viene allocata al thread
- Quando una risorsa viene rilasciata da un thread, l'**arco di assegnazione** viene riconvertito in un arco di reclamo.
- Le risorse devono venir reclamate a priori nel sistema
- Supponiamo che il thread T_i richieda una risorsa R_j
- La richiesta può essere accordata solo se la conversione dell'arco di reclamo in arco di assegnazione non provoca il formarsi di un ciclo nel grafo di assegnazione delle risorse

ALGORITMO DEL BANCHIERE

- Deve il suo nome al fatto che le banche non assegnano mai tutto il denaro di cui dispongono, per non restare sprovviste di fronte a nuove richieste dei clienti.
- Permette di gestire istanze multiple di una risorsa (a differenza dell'algoritmo con grafo di assegnazione risorse)
- Ciascun thread deve dichiarare a priori il massimo impiego di risorse
- Quando un thread richiede una risorsa può non venir servito istantaneamente, per mantenere lo stato sicuro
- Quando ad un thread vengono allocate tutte le risorse deve restituirlle in tempo finito

STRUTTURE DATI DELL'ALGORITMO DEL BANCHIERE

- Sia n il numero dei thread presenti nel sistema ed m il numero dei tipi di risorse.
 - **Available:** vettore di lunghezza m ; se $Available[j]=k$, vi sono k istanze disponibili del tipo di risorsa R_j .
 - **Max:** matrice $n \times m$ se $Max[i,j]=k$; il thread T_i può richiedere al più k istanze del tipo di risorsa R_j
 - **Allocation:** matrice $n \times m$ se $Allocation[i,j]=k$, a T_i sono attualmente allocate k istanze di R_j
 - **Need:** matrice $n \times m$ se $Need[i,j]=k$; T_i può richiedere k ulteriori istanze di R_j per completare il proprio task
$$Need[i,j] = Max[i,j] - Allocation[i,j]$$
- Al trascorrere del tempo, le strutture variano sia nelle dimensioni che nei valori.

ALGORITMO DI VERIFICA DELLA SICUREZZA

1. Siano **Work** e **Finish** vettori di lunghezza m ed n rispettivamente; si inizializzi:
 - a) **Work = Available**
 - b) **Finish[i]=false per i=0,1,...,n**
2. Si cerchi i tale che valgano contemporaneamente:
 - (a) **Finish[i]= false**
 - (b) **Need(i) <= Work**

Se tale **i** non esiste, si esegua il passo 4
3. Si assegni **Work=Work+Allocation(i)**
Finish[i]= true
si torni al passo 2.
4. Se **Finish[i]=true** per ogni i il sistema è in stato sicuro altrimenti lo stato è non sicuro.

ALGORITMO DI RICHIESTA DELLE RISORSE PER IL THREAD Pi

- Sia **Request(i)** il vettore delle richieste per il thread **Pi**. Se **Request(i)[j]=k** il thread **Pi** richiede ulteriori k istanze del tipo di risorsa **Rj**.
 1. Se **Request(i)<=Need(i)**, si vada al passo 2 altrimenti, si riporti una condizione di errore, poiché il thread ha ecceduto il massimo numero di richieste
 2. Se **Request(i)<=Available**, si vada al passo 3; altrimenti **Pi** deve attendere, poiché le risorse non sono disponibili.
 3. Il sistema simula l'allocazione a **Pi** delle risorse richieste, modificando lo stato di allocazione come segue:

$$\begin{aligned} \text{Available} &= \text{Available}-\text{Request}(i) \\ \text{Allocation}(i) &= \text{Allocation}(i)-\text{Request}(i) \\ \text{Need}(i) &= \text{Need}(i)-\text{Request}(i) \end{aligned}$$

- Se lo stato è sicuro, le risorse vengono definitivamente allocate a **Pi**.
- Se lo stato è non sicuro, **Pi** deve attendere, e viene ripristinato il vecchio stato di allocazione delle risorse.
 - 5 processes P_0 through P_4 and 3 resource types:
A (10 instances), B (5 instances), and C (7 instances)
 - Snapshot at time T_0 :

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

RECAP

Allocation: P_i is currently allocated k instances of R_j
Max: P_i may request at most k instances of resource type R_j
Available: there are k instances of resource type R_j available

RILEVAMENTO DEADLOCK

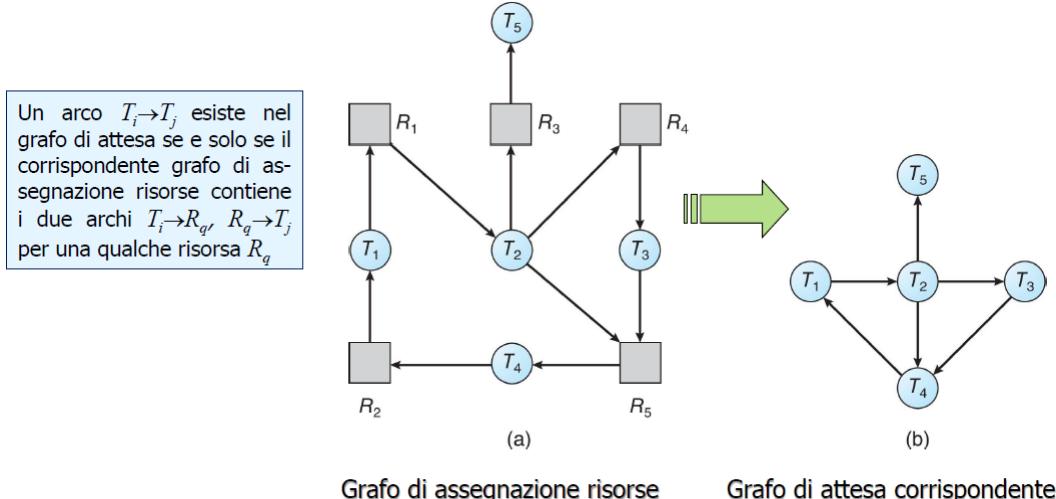
- Per ripristinarne le funzionalità, sono necessari:
 - un algoritmo di rilevamento del deadlock.
 - e, successivamente, un algoritmo di ripristino dal deadlock.

ISTANZA SINGOLA SU CIASCUN TIPO DI RISORSA

- Si impiega un **grafo di attesa**

- I nodi rappresentano i thread
- L'arco $T_i \rightarrow T_j$ esiste se T_i è in attesa che T_j rilasci una risorsa che gli occorre
- Periodicamente viene richiamato un algoritmo che verifica la presenza di cicli nel grafo:
 - La presenza di un ciclo attesta una situazione di deadlock
- Un algoritmo per rilevare cicli in un grafo richiede un numero di operazioni dell'ordine di n^2 , dove n è il numero di vertici del grafo.

GRAFO DI ASSEGNAZIONE RISORSE E GRAFO DI ATTESA



PIU' ISTANZE PER CIASCUNA RISORSA

Occorrono strutture dati variabili, simili a quelle utilizzate per l'algoritmo del Banchiere:

- **Available** → vettore di lunghezza m indica il numero di risorse disponibili di ciascun tipo
- **Allocation** → matrice $n \times m$ definisce il numero di risorse di ciascun tipo attualmente allocate a ciascun thread
- **Request** → matrice $n \times m$ indica la richiesta corrente di ciascun thread se $\text{Request}[i,j]=k$, il thread T_i richiede k istanze supplementari della risorsa R_j .

L'algoritmo di rilevamento indaga su ogni possibile sequenza di assegnazione per i thread da completare.

ALGORITMO DI RILEVAMENTO

1. Siano **Work** e **Finish** due vettori di lunghezza m (risorse) e n (processi) rispettivamente; inizialmente sia:

- Work = Available**
- Per $i = 1, 2, \dots, n$, se $\text{Allocation}(i) \neq 0$, allora $\text{Finish}[i] = \text{false}$; altrimenti, $\text{Finish}[i] = \text{true}$

2. Si trovi un indice i tale che valgano entrambe le condizioni:

- $\text{Finish}[i] == \text{false}$
- $\text{Request}(i) \leq \text{Work}$

Se tale i non esiste, si vada al passo 4.

3. Si assegna: $\text{Work} = \text{Work} + \text{Allocation}(i)$

$$\text{Finish}[i] = \text{true}$$

Si vada al passo 2.

4. Se $\text{Finish}[i] == \text{false}$, per qualche i , $1 \leq i \leq n$, il sistema è in uno stato di deadlock; inoltre, se $\text{Finish}[i] == \text{false}$, allora P_i è in deadlock.

ESEMPIO ALGORITMO DI RILEVAMENTO

- ❖ 5 thread, da T_0 a T_4
- ❖ 3 tipi di risorse:
A (7 istanze), B (2 istanze) e C (6 istanze)
- ❖ Istantanea al tempo t_0 :

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
T_0	0	1	0	0	0	0	0	0	0
T_1	2	0	0	2	0	2			
T_2	3	0	3	0	0	0			
T_3	2	1	1	1	0	0			
T_4	0	0	2	0	0	2			

- ❖ La sequenza $\langle T_0, T_2, T_1, T_3, T_4 \rangle$ conduce a $\text{Finish}[i] = \text{true}$ per ogni i
- ❖ T_2 richiede un'istanza supplementare della risorsa C

	Request		
	A	B	C
T_0	0	0	0
T_1	2	0	2
T_2	0	0	1
T_3	1	0	0
T_4	0	0	2

- ❖ Qual è lo stato del sistema?
 - Può reclamare le risorse possedute dal thread T_0 , ma il numero di risorse disponibili non è sufficiente a soddisfare gli altri thread
 - ⇒ deadlock: coinvolge i thread T_1, T_2, T_3 e T_4

IMPIEGO DELL'ALGORITMO DI RILEVAMENTO

- Quando e quanto spesso richiamare l'algoritmo di rilevamento dipende da:
 - Frequenza (con la quale si verificano i deadlock)
 - Numero di thread che vengono eventualmente influenzati dal deadlock (e sui quali occorre effettuare un rollback)
 - Almeno un thread per ciascun ciclo
- Se l'algoritmo viene richiamato con frequenza casuale, possono essere presenti molti cicli nel grafo delle risorse → non si può stabilire quale dei thread coinvolti nel ciclo abbia "causato" il deadlock

RIPRISTINO DAL DEADLOCK: TERMINAZIONE DEI THREAD

- Terminazione in abort di tutti i thread in deadlock
- Terminazione in abort di un thread alla volta fino all'eliminazione del ciclo di deadlock
- In quale ordine si decide di terminare i thread? I fattori significativi sono:
 - La priorità del thread
 - Il tempo di computazione trascorso e il tempo ancora necessario al completamento del thread
 - Quantità e tipo di risorse impiegate (non/facilmente)
 - Risorse ulteriori necessarie al thread per terminare il proprio compito
 - Numero di thread che devono essere terminati

RIPRISTINO DAL DEADLOCK: PRELAZIONE DI RISORSE

- **Selezione di una vittima:** È necessario stabilire l'ordine di prelazione per minimizzare i costi
- **Rollback:** Un thread a cui sia stata prelazionata una risorsa deve ritornare ad uno stato sicuro, da cui ripartire.
- **Starvation:** Alcuni thread possono essere sempre selezionati come vittime della prelazione includere il numero di rollback nel fattore di costo.

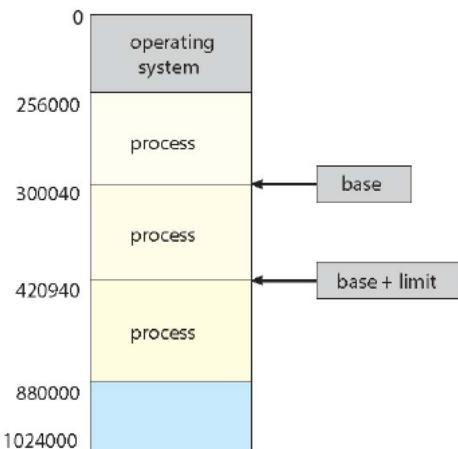
--- CAPITOLO 9 - MEMORIA CENTRALE ---

BACKGROUND

- Il programma è conservato permanentemente nel **backing store** (dischi)
- Per poter essere eseguiti, i programmi devono essere copiati (almeno parzialmente) in memoria centrale, per "trasformarsi" in processi.
- La memoria "vede" soltanto un flusso di indirizzi e non conosce la modalità con cui sono stati generati o a cosa si riferiscano (istruzioni o dati):
 - indirizzo + richiesta di lettura
oppure
 - indirizzo + dati + richiesta di scrittura
- La memoria principale e i registri sono gli unici dispositivi di memoria cui la CPU può accedere direttamente.
- I registri vengono acceduti in un ciclo di clock (o meno)
- Un accesso alla memoria centrale può invece richiedere diversi cicli di clock (occorre "passare" dal bus)
 - Possibile stallo del processore in attesa dei dati
- La memoria cache (on-chip) che si situa, nella gerarchia delle memorie, tra la memoria principale ed i registri della CPU.
- La protezione della memoria (a livello hardware - il SO non interviene negli accessi della CPU alla RAM) è fondamentale per la corretta operatività del sistema.

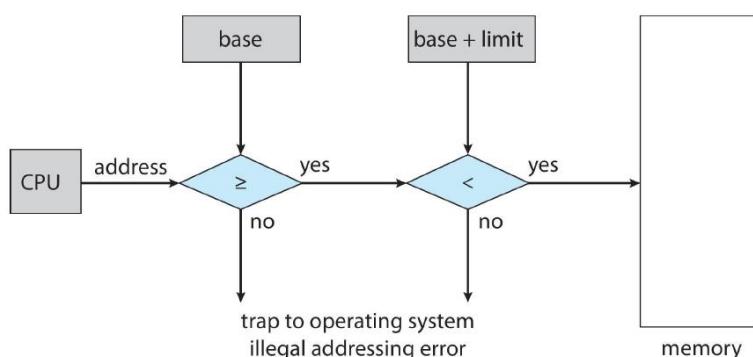
PROTEZIONE DELLA MEMORIA

- Ogni processo deve avere uno spazio di memoria separato
- Possibile soluzione: i registri **base** e **limite** definiscono lo spazio degli indirizzi fisici di ciascun processo.



PROTEZIONE DELL'INDIRIZZO HARDWARE

- Per mettere in atto il meccanismo di protezione, la CPU confronta ogni indirizzo generato dal processo utente con i valori contenuti nei due registri
 - indirizzo \geq base
 - indirizzo $<$ base+limite



Supporto hardware alla protezione della memoria mediante registri base e limite

- Solo il SO può caricare i registri base e limite tramite istruzioni privilegiate.
- I programmi su disco, pronti per essere portati in memoria per l'esecuzione, vengono inseriti in una coda di input
 - Senza supporto, deve essere caricato all'indirizzo 0000
- Non conviene avere l'indirizzo fisico del processo del primo utente sempre a 0000.
 - Come non può essere?
- Prima di essere eseguiti, i programmi utente passano attraverso stadi diversi, con diverse rappresentazioni degli indirizzi
 - Gli indirizzi nel programma sorgente sono simbolici
 - e.g. "indice", un nome di variabile
 - Il compilatore associa gli indirizzi simbolici a indirizzi rilocabili (per esempio, calcolati a partire dalla prima istruzione del programma)
 - e.g. "14 byte dall'inizio del modulo corrente"
 - Il linker o il loader fa corrispondere gli indirizzi rilocabili a indirizzi assoluti (nella memoria fisica)
 - e.g. 74014
- Ogni associazione stabilisce una corrispondenza fra spazi di indirizzi (mappa uno spazio di indirizzi in un altro).

BINDING (ASSOCIAZIONE) DI ISTRUZIONI E DATI ALLA MEMORIA

L'associazione - binding - di istruzioni e dati a indirizzi di memoria può avvenire durante la fase di...

- **Compilazione:** se la posizione in memoria del processo è nota a priori, può essere generato **codice assoluto** se la locazione iniziale cambia, è necessario ricompilare il codice (esempio: programmi MS-DOS nel formato COM).
- **Caricamento:** se la posizione in memoria non è nota in fase di compilazione, è necessario generare **codice rilocabile** il compilatore genera indirizzi relativi che vengono convertiti in indirizzi assoluti dal loader se l'indirizzo iniziale cambia, il codice deve essere ricaricato in memoria.
- **Esecuzione:** se il processo può essere spostato a run time da un segmento di memoria all'altro, il binding viene rimandato fino al momento dell'esecuzione - codice dinamicamente rilocabile
 - Necessita di un opportuno supporto hardware per il mapping degli indirizzi.

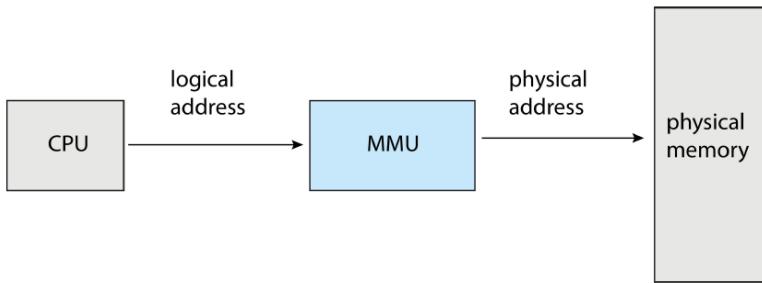
SPAZI DI INDIRIZZI LOGICI E FISICI

Il concetto di uno spazio di indirizzi logici nettamente distinto dallo spazio degli indirizzi fisici è centrale per la gestione della memoria

- **Indirizzi logici** → generati dalla CPU, altrimenti chiamati **indirizzi virtuali**
- **Indirizzi fisici** → indirizzi contenuti nel memory address register MAR e utilizzati dall'unità di memoria
- Gli indirizzi logici corrispondono agli indirizzi fisici negli schemi di binding in fase di compilazione e di caricamento, mentre differiscono per il binding in fase di esecuzione:
 - Lo **spazio degli indirizzi logici** è costituito dall'insieme di tutti gli indirizzi logici generati da un programma
 - Lo **spazio degli indirizzi fisici** è l'insieme di tutti gli indirizzi fisici "visti dalla memoria" per quel programma

MEMORY MANAGEMENT UNIT (MMU)

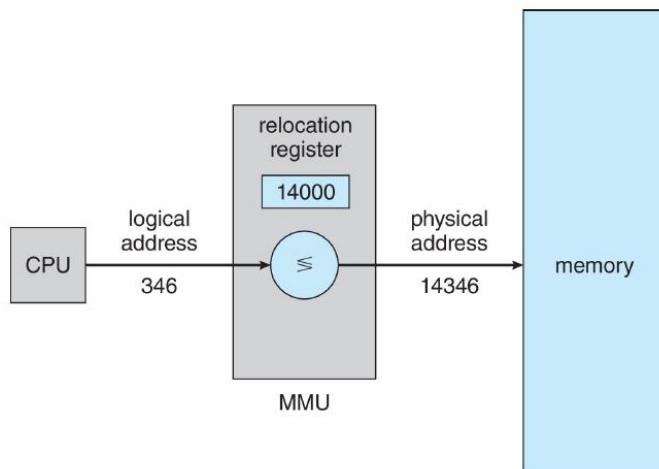
- Dispositivo hardware che mappa indirizzi virtuali su indirizzi fisici a run-time



- Molti metodi possibili, trattati nel resto di questo capitolo

MMU - RICOLOCAZIONE DEL REGISTRO

- La più semplice il valore contenuto nel **registro di rilocazione** (il registro base) viene sommato ad ogni indirizzo generato dal processo utente nel momento stesso in cui l'indirizzo viene inviato alla memoria.
- Il programma utente "ragiona" in termini di indirizzi virtuali, né mai è (deve essere) consci del loro mapping fisico.



CARICAMENTO DINAMICO

- L'intero programma non deve risiedere in memoria per essere eseguito
- Le routine risiedono su disco in un formato rilocabile e non vengono caricate fino a quando non vengono richiamate.
- Miglior impiego della memoria routine non utilizzate non vengono mai caricate
- Utile quando si richiedono grandi quantità di codice per gestire situazioni che avvengono raramente (condizioni di eccezione, errori) :
 - Il caricamento dinamico viene implementato mediante software opportunamente codificato (modulare)
 - Tuttavia, il SO fornisce librerie di procedure che realizzano il caricamento dinamico

LINK DINAMICO DELLE LIBRERIE

- **Link statico** → librerie di sistema e codice del programma combinati dal caricatore nell'immagine binaria
- Link dinamico → il link viene rinviato fino al momento dell'esecuzione
- Piccole porzioni di codice, dette stub vengono impiegate per localizzare la routine appropriata nella libreria residente in memoria
- Lo stub rimpiazza se stesso con l'indirizzo della routine e la esegue.
- Il SO deve verificare se la routine si trova già nello spazio degli indirizzi del processo o, eventualmente, deve renderla accessibile se è presente in memoria nello spazio di indirizzi di un altro processo.
 - Se non si trova nello spazio degli indirizzi, aggiungere allo spazio degli indirizzi
- Il è particolarmente utile per le librerie
- Modalità operativa a librerie condivise

ALLOCAZIONE CONTIGUA (ALLOCAZIONE MEMORIA NELLA SLIDE INGLESE)

- La memoria principale è una risorsa limitata, che deve essere allocata in modo efficiente, l'**allocazione contigua**
 - è uno dei metodi che lo consente.
- La memoria principale deve contenere sia i processi utente che di sistema viene pertanto suddivisa in **due parti**:
 - La parte residente del SO è generalmente memorizzata nella memoria alta (nelle architetture moderne), insieme al vettore degli interrupt.
 - I processi utente sono memorizzati nella memoria bassa.
- Ciascun processo è contenuto in un'unica sezione contigua di memoria.

PROTEZIONE DELLA MEMORIA

- I registri di rilocazione e limite vengono utilizzati per proteggere reciprocamente i processi utente e per prevenirne eventuali accessi a codice e dati di sistema:
 - Il registro di rilocazione contiene il valore del più piccolo indirizzo fisico dell'area di memoria allocata ad un processo.
 - Il registro limite definisce l'intervallo di variabilità degli indirizzi logici
 - La MMU mappa dinamicamente gli indirizzi logici negli indirizzi fisici
 - Il codice del kernel può essere in parte transiente ed il SO può cambiare dinamicamente le proprie dimensioni

ALLOCAZIONE PARTIZIONI VARIABILI

- **Partizione variabile**: ogni partizione è allocata dinamicamente in base alla dimensione del processo da allocare.
- Un buco - **hole** - è un blocco di memoria disponibile nella memoria sono sparsi buchi di varie dimensioni.
- Quando viene caricato un nuovo processo, gli viene allocato un buco grande abbastanza da contenerlo
- Quando un processo termina, libera la memoria alloca ta partizioni libere contigue vengono "riasseminate".
- Il SO conserva informazioni su:
 - Partizioni allocate
 - Partizioni libere

PROBLEMA DI ALLOCAZIONE DINAMICA

- Come soddisfare una richiesta di lunghezza n da una lista di buchi (hole) liberi?
 - **First-fit**: Viene allocato il **primo buco** grande abbastanza
 - **Best-fit**: Viene allocato il buco **più piccolo** capace di contenere il processo è necessario scandire tutta la lista dei buchi (se non è ordinata) → Si produce il più piccolo buco residuo
 - **Worst-fit**: Viene allocato il buco **più grande** è ancora necessario ricercare in tutta la lista
- First-fit e Best-fit sono migliori di Worst-fit in termini di velocità e di impiego di memoria, rispettivamente.

FRAMMENTAZIONE

- **Frammentazione interna** → La memoria allocata può essere leggermente maggiore della memoria effettivamente richiesta (pochi byte di differenza) la differenza di dimensioni è memoria interna ad una partizione che non viene impiegata completamente.
- **Frammentazione esterna** → È disponibile lo spazio necessario a soddisfare una richiesta, ma non è contiguo
- **Regola del 50%**: con First-fit, per n blocchi assegnati, 0.5n blocchi possono andare "persi" per frammentazione.
 - Un terzo della memoria diventa inutilizzabile!
- Si può ridurre la frammentazione esterna con la **compattazione**

COMPATTAZIONE

- Mescola i contenuti della memoria per mettere insieme tutta la memoria libera in un blocco di grandi dimensioni
- La compattazione è possibile solo se il riposizionamento è dinamico e viene effettuato in fase di esecuzione
- Problema di I/O: I/O eseguito durante la compattazione → I dati arrivano nel posto sbagliato.

- Soluzioni al problema I/O
 - Aggancia il processo in memoria mentre è coinvolto nell'I/O
 - Eseguire l'I/O solo nei buffer del sistema operativo
- Considera ora che il backup store ha gli stessi problemi di frammentazione ma su dischi

ALLOCAZIONE NON CONTIGUA

- Consente a un processo di risiedere in posizioni diverse della memoria:
 - Segmentazione
 - Paging
 - Paged segmentation

SEGMENTAZIONE

- Schema di gestione della memoria che supporta la visualizzazione della memoria da parte dell'utente
- Un programma è una raccolta di segmenti
- Un segmento è un'unità logica come:
 - programma principale
 - procedura
 - funzione
 - metodo
 - oggetto
 - variabili locali, variabili globali
 - blocco comune
 - pila
 - tabella dei simboli
 - matrici
- Lo spazio degli indirizzi fisici di un processo può essere non contiguo;
- Blocchi di memoria di dimensioni variabili
- Frammentazione esterna

ARCHITETTURA DELLA SEGMENTAZIONE

- L'indirizzo logico è costituito una tupla con due elementi:
 $\langle \text{numero-segmento}, \text{offset} \rangle$
- **Tabella dei segmenti** - mappa gli indirizzi fisici bidimensionali; ogni voce della tabella ha:
 - **base** - contiene l'indirizzo fisico iniziale in cui si trovano i segmenti risiedono nella memoria
 - **limit** - specifica la lunghezza del segmento
- **Il registro di base della tabella dei segmenti (STBR)** punta alla posizione della tabella dei segmenti in memoria.
- **Il registro della lunghezza della tabella dei segmenti (STLR)** indica il numero di segmenti utilizzati da un programma.
 - Il numero di segmento s è legale se $s < \text{STLR}$

PAGINAZIONE (PAGING)

- Con la paginazione, lo spazio degli indirizzi fisici di un processo può essere non contiguo
 - Al processo è allocata memoria fisica ogni volta che quest'ultima si rende disponibile.
 - Evita la frammentazione esterna.
 - Evita il problema di blocchi di memoria di dimensioni variabili.
- I blocchi di memoria fisica, chiamati **frame** o pagine fisiche hanno dimensione pari a una potenza del 2, valori tipici negli elaboratori attuali sono compresi nell'intervallo 4 KB - 1 GB
- La memoria logica viene suddivisa in blocchi della stessa dimensione, chiamati **pagine** logiche
- Occorre tenere traccia di tutti i frame liberi
- Per eseguire un programma di dimensione N pagine, è necessario trovare N frame liberi prima di caricare il programma
- Si impiega una **tabella delle pagine** per tradurre gli indirizzi logici negli indirizzi fisici corrispondenti
- Si ha solo frammentazione interna (relativa all'ultimo frame).

SCHEMA DI TRADUZIONE DEGLI INDIRIZZI

L'indirizzo logico generato dalla CPU viene suddiviso in:

- **Numero di pagina (p)** — impiegato come indice in una **tabella delle pagine** che contiene l'indirizzo base di ciascun frame nella memoria fisica (o il numero del frame)
- **Offset nella pagina (d)** — combinato con l'indirizzo base per definire l'indirizzo fisico che viene inviato all'unità di memoria

page number	page offset
p	d
$m-n$	n

Spazio logico di 2^m indirizzi con 2^{m-n} pagine di dimensione 2^n

PAGING - CALCOLO DELLA FRAMMENTAZIONE INTERNA

- ❖ **Esempio**
 - Dimensione della pagina: 2048 byte
 - Dimensione del processo: 72766 byte
 - 35 pagine + 1086 byte \Rightarrow 36 frame
 - Frammentazione interna: $2048-1086=962$ byte
- ❖ Frammentazione nel caso peggiore: 1 frame – 1 byte; nel caso medio pari a mezzo frame
- ❖ Dunque frame piccoli sono preferibili?
 - Non necessariamente, dato che occorre memoria per tener traccia di ogni elemento della tabella delle pagine; inoltre, l'I/O è più efficiente per pagine grandi
 - La dimensione delle pagine cresce nel tempo
 - Windows 10 supporta pagine da 4 KB e 2 MB
 - Linux supporta pagine da 4 KB e *huge page* più grandi, di dimensioni dipendenti dall'architettura (`getpagesize()` o, da linea di comando, `getconf PAGESIZE`)
- ❖ La memoria "vista" dai processi e la memoria fisica differiscono significativamente
- ❖ I processi sono "relegati" nel loro spazio di memoria grazie all'implementazione del paging

IMPLEMENTAZIONE DELLA TABELLA DELLE PAGINE

- Attualmente, la tabella delle pagine risiede in memoria centrale in generale, si ha una tabella per ogni processo:
 - Il **registro Page-Table Base Register PTBR** punta all'inizio della tabella
 - Il **registro Page-Table Length Register PTLR** indica la dimensione della tabella
- Con questo schema, ogni accesso a dati o istruzioni richiede di fatto due accessi alla memoria uno per la tabella e uno per le istruzioni/dati.
- Il doppio accesso alla memoria può essere evitato con l'uso di registri associativi (altrimenti detti **translation-look-aside buffer, TLB**) attraverso i quali si effettua una ricerca parallela veloce su una piccola tabella 64-1024 elementi)

HARDWARE MEMORIA ASSOCIAUTIVA (DETTA TLB)

- Memoria associativa-ricerca parallela



- Traduzione dell'indirizzo (p, d):

- Se p è nel registro associativo, si reperisce il numero del frame
- Altrimenti, si recupera il numero di frame dalla tabella delle pagine in memoria

BUFFER DI TRADUZIONE LOOK-ASIDE

- TLB tipicamente piccoli (da 64 a 1.024 voci)
- In caso di mancato TLB, il valore viene caricato nel TLB per un accesso più rapido la prossima volta
 - È necessario considerare le politiche di sostituzione
 - Alcune voci possono essere **cachable** per un accesso rapido permanente
- Alcuni TLB memorizzano gli **ASID (Address Space Identifier)** in ogni voce TLB - identificano in modo univoco ogni processo per fornire la protezione dello spazio degli indirizzi per quel processo
 - In caso contrario, è necessario svuotare il TLB a ogni cambio di contesto

TEMPO DI ACCESSO EFFETTIVO

- Hit ratio ALFA - percentuale di volte che un numero di pagina viene reperito nei registri associativi è correlata al numero di registri associativi.
- Un Hit ratio dell'80% significa che troviamo il numero di pagina desiderato nel TLB l'80% delle volte.
- Supponiamo che occorrono 10 nanosecondi per accedere alla memoria:
 - Se troviamo la pagina desiderata in TLB, un accesso alla memoria mappata richiede 10 nanosecondi
 - Altrimenti, abbiamo bisogno di due accessi alla memoria, quindi sono 20 nanosecondi
- Tempo di accesso effettivo **EAT (Effective Access Time)**
 - $EAT = 0.80 \times 10 + 0.20 \times 20 = 12$ nanosecondi

implicando un rallentamento del 20% nel tempo di accesso
- Consideriamo un rapporto di successo più realistico del 99%
 - $EAT = 0.99 \times 10 + 0.01 \times 20 = 10.1$ nanosecondi

implicando solo un rallentamento dell'1% nel tempo di accesso

PROTEZIONE DELLA MEMORIA

- La protezione della memoria è implementata associando uno o più bit di protezione a ciascun frame nella tabella delle pagine.
 - I bit determinano se una certa pagina può essere acceduta in sola lettura, lettura/scrittura e/o esecuzione.
- Nelle architetture più datate, in alternativa all'utilizzo del **Page-Table-Length-Register**, un ulteriore bit di **validità** veniva associato ad ogni elemento della tabella delle pagine:
 - Un valore "valido" indicava che la pagina era nello spazio degli indirizzi logici del processo, e quindi era legale.
 - Un valore "non valido" indicava che la pagina non si trovava nello spazio degli indirizzi logici del processo.
 - Il bit di validità consentiva di riconoscere gli indirizzi illegali e di notificare l'anomalia con un segnale di eccezione
- Qualsiasi violazione provoca una trappola per il kernel
- Può anche aggiungere più bit per indicare se è consentita la sola lettura, lettura-scrittura, sola esecuzione

PAGINE CONDIVISE

- **Codice condiviso:**
 - Una copia di codice di sola lettura rientrante viene condivisa fra processi (ad es text editor, compilatori, sistemi a finestre).
 - Simile ai thread che condividono lo stesso "spazio di indirizzi del task" di appartenenza
 - Il codice condiviso deve apparire nella stessa locazione nello spazio degli indirizzi logici di tutti i processi.
- **Codice e dati privati:**
 - Ciascun processo mantiene una copia separata dei dati e del codice.
 - Le pagine di codice e dati privati possono apparire ovunque nello spazio degli indirizzi logici.

STRUTTURA DELLA TABELLA DELLE PAGINE

L'occupazione di memoria dovuta alle strutture dati necessarie per gestire la paginazione può divenire eccessivamente grande; si consideri:

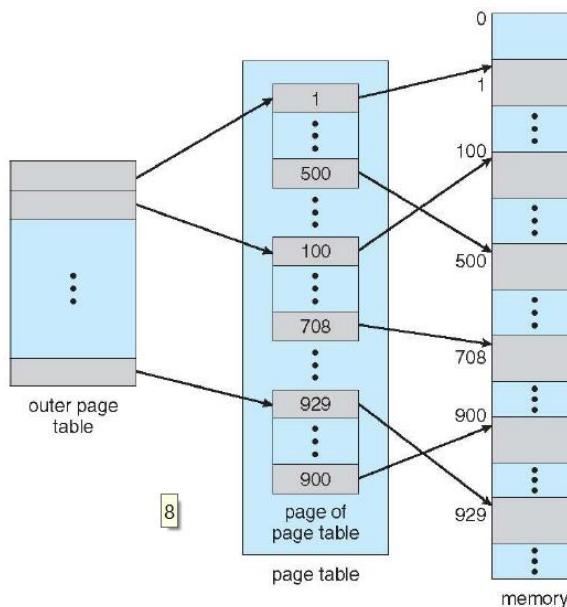
- Uno spazio di indirizzi logici a 32 bit, con pagine di dimensione pari a 4 KB, 2^{12} byte
 - La tabella delle pagine potrebbe essere costituita da più di un milione di elementi ($2^{32}/2^{12}$)
- Se ciascun elemento occupasse 4 byte
 - 4 MB di occupazione di memoria per la sola tabella delle pagine
 - Meglio evitare di collocare la tabella delle pagine in modo contiguo in memoria centrale.

SOLUZIONE: ottimizzare l'occupazione di memoria e/o l'accesso all'informazione contenuta

- Paginazione gerarchica
- Tabella delle pagine hash
- Tabella delle pagine invertita

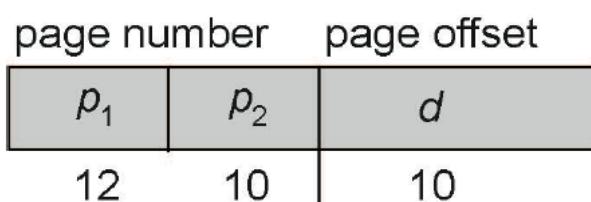
PAGINAZIONE GERARCHICA

- Dividere lo spazio degli indirizzi logici in più tabelle delle pagine.
- Un metodo semplice consiste nell'adottare un algoritmo di paginazione a due livelli.
- La tabella delle pagine viene essa stessa paginata.



ESEMPIO DI PAGINA GERARCHICA (a due livelli)

- Un indirizzo logico, in architetture a 32 bit con dimensione della pagina di 4 KB, viene suddiviso in
 - un numero di pagina a 20 bit
 - un offset all'interno della pagina di 12 bit
- Dato che la tabella delle pagine è paginata, il numero di pagina viene ulteriormente suddiviso in
 - un numero di pagina di 10 bit (tabella esterna)
 - un offset di 10 bit (tabella delle pagine)
- Pertanto, un indirizzo logico si rappresenta come



dove p_1 è un indice nella tabella delle pagine esterna e p_2 rappresenta lo spostamento all'interno della pagina indicata dalla tabella esterna (4 byte per ogni elemento)

- Metodo noto anche come tabella delle pagine ad **associazione diretta (forward-mapped-page-table)**

SPAZIO DI INDIRIZZI LOGICI ARCHITETTURA A 64 BIT

- ❖ Nell'esempio seguente, la tabella delle pagine esterna di secondo livello ha ancora una dimensione pari a 2^{34} byte

outer page	inner page	offset	
p_1	p_2	d	
42	10	12	
2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

⇒ Necessari 4 accessi alla memoria per reperire un dato

- ⇒ Per le architetture a 64 bit la paginazione gerarchica è da considerarsi inadeguata a causa dei costi proibitivi di accesso alla memoria in caso di TLB miss

TABELLA DELLE PAGINE HASH

- Comune per spazi di indirizzi a più di 32 bit
- L'argomento della funzione hash è il numero della pagina virtuale
 - Per la gestione delle collisioni, ogni elemento della tabella hash contiene una lista concatenata di elementi che la funzione hash fa corrispondere alla stessa posizione nella tabella
- Ciascun elemento della tabella è composto da tre campi
 - Numero della pagina virtuale
 - Indirizzo del frame corrispondente
 - Puntatore al successivo elemento nella lista
- I numeri di pagina virtuale vengono confrontati con tutti gli elementi della catena
 - A fronte di una ricerca positiva, si estrae il corrispondente numero di frame

TABELLA DELLE PAGINE HASH RAGGRUPPATE

- Variazione per indirizzi a 64 bit
- Simile all'hashing ma ogni voce fa riferimento a più pagine (come 16) anziché a 1
- Particolarmente utile per spazi di indirizzi **sparsi** (dove i riferimenti di memoria sono non contigui e sparsi)

TABELLA DELLE PAGINE INVERTITA

- Generalmente, si associa una tabella delle pagine ad ogni processo, che contiene un elemento per ogni pagina virtuale che il processo sta utilizzando (o un elemento per ogni possibile indirizzo virtuale, a prescindere dalla validità)
- Rappresentazione naturale, dato che i processi si riferiscono alle pagine per mezzo di indirizzi virtuali ed il SO si occupa della traduzione in indirizzi fisici.

Problema: ogni tabella delle pagine può contenere milioni di elementi ed occupare molta memoria

Soluzione: tabella delle pagine invertita

- Un elemento della tabella per ogni frame
- Gli elementi della tabella contengono il numero della pagina virtuale memorizzata nel dato frame e l'identificativo del processo che possiede tale pagina
- Riduzione della memoria richiesta per realizzare il mapping indirizzi logici/fisici
- Incremento del tempo necessario per ricercare nella tabella quando si fa un riferimento a pagina
 - Notare che è necessario ricercare su tutta la tabella!

SWAPPING

- Un processo può venire temporaneamente riversato - **swapped out** - dalla memoria centrale alla backing store dalla quale, in seguito, viene portato nuovamente in memoria per proseguire l'esecuzione.
 - Lo spazio fisico totale di memoria allocata ai processi può eccedere la memoria fisica.
- **Backing store** - È una partizione del disco ad accesso rapido, sufficientemente capiente da accogliere copie di tutte le immagini di memoria per tutti i processi utente deve garantire accesso diretto a tali immagini.
- **Roll out, roll in** - È una variante dello swapping impiegata per algoritmi di scheduling basati su priorità processi a bassa priorità vengono riversati sulla memoria di massa, in modo tale da permettere che processi a priorità maggiore vengano caricati ed eseguiti.
- La maggior parte del tempo di swap è dovuta al trasferimento di dati il tempo totale di trasferimento è direttamente proporzionale alla quantità di memoria riversata
- Il sistema operativo mantiene una ready queue dei processi la cui immagine è stata riversata su disco e dei processi attualmente presenti in memoria.

SWAPPING E TEMPO DI CONTEXT SWITCH

- Se il prossimo processo da eseguire non è in memoria, e la memoria è piena, occorre scambiare un processo attualmente in memoria con il processo target.
- Il tempo di context-switch può essere molto elevato.
- Esempio: si consideri un processo da 100 MB che viene "swappato" sul disco fisso ad un tasso di 50 MB/sec
 - Tempo di swap-out pari a 2 sec
 - Uguale tempo di swap-in del processo target
 - Tempo totale dell'operazione di swap all'interno del context-switch pari a 4 sec
 - Nelle architetture attuali, tempo di context-switch < 10 nano sec

SWAPPING NEI DISPOSITIVI MOBILI

- Lo swapping non è solitamente supportato nei dispositivi mobili:
 - La memoria di massa è realizzata tramite flash
 - Poco capienti
 - Accesso lento (bassa velocità di trasferimento fra memoria centrale e memoria flash) e rischio di "memory leak".
- Metodi diversi per liberare la memoria a fronte di sovraccarichi:
 - iOS "**chiede**" alle applicazioni di rinunciare volontariamente alla memoria allocata
 - I dati di sola lettura vengono rimossi dal sistema e, se necessario, successivamente ricaricati dalla memoria flash.
 - Terminazione forzata delle app che non riescono a liberare memoria
 - Android termina le app qualora la memoria libera disponibile non sia sufficiente, ma scrive **lo stato dell'applicazione** nella memoria flash, per riavviarle rapidamente.
 - Entrambi i SO supportano invece la paginazione.

--- CAPITOLO 11 - MEMORIA DI MASSA (MEMORIA SECONDARIA) ---

MEMORIA DI MASSA E DISPOSITIVI I/O

- I dispositivi collegati ad un calcolatore possono avere caratteristiche altamente variabili
- Rappresentano, nel loro insieme, la componente più lenta, voluminosa e variegata dell'elaboratore
- Il sistema operativo deve offrire alle applicazioni funzionalità che consentano l'accesso ai dispositivi mediante interfacce semplici e uniformi
- Deve inoltre garantire l'ottimizzazione dell'I/O, che costituisce il collo di bottiglia delle prestazioni del sistema di calcolo.

PANORAMICA DELLA MEMORIA DI MASSA

- La maggior parte della memoria secondaria per i computer moderni è costituita da **unità disco rigido (HDD)** e dispositivi di **memoria non volatile (NVM)**.
- Gli **HDD** (dischi magnetici) rappresentano ancora oggi un mezzo diffuso per la memorizzazione di massa, sono costituiti da piatti, rivestiti con materiale magnetico (ossido di ferro)

- I dischi ruotano ad una velocità compresa tra i 60 e i 250 giri al secondo
- La **velocità di trasferimento** è la velocità con cui i dati fluiscono dall'unità a disco alla RAM.
- Il **tempo di posizionamento** è:
 - il tempo necessario a spostare il braccio del disco in corrispondenza del cilindro desiderato (**seek time**).
 - più il tempo necessario affinché il settore desiderato si porti sotto la testina **latenza di rotazione**.
 - Il **crollo della testina**, corrisponde all'impatto della stessa sulla superficie del disco.
- I dischi possono essere rimovibili

GLI HARD DISK

- Il raggio dei piatti variava, storicamente fra 14 e 85 pollici
- I formati attualmente più comuni sono 3.5", 2.5" e 1.8"
- La capacità standard dei dischi (si attesta fra 500 GB e 16 TB)
- Performance:
 - Velocità di trasferimento (teorica): 6 Gb/sec
 - Velocità di trasferimento (effettiva): 1 Gb/sec
 - Seek time compreso fra 3 msec e 12 msec - (9 msec in media per i dischi presenti nei PC)
 - Tempo di latenza calcolato in base alla velocità di rotazione
 - $1 / (\text{RPM} / 60) = 60 / \text{RPM}$
 - Latenza media $\frac{1}{2}$ giro

PRESTAZIONI DEGLI HARD DISK

- **Tempo di accesso medio** = seek time medio + latenza media
 - Per i dischi più veloci $\rightarrow 3 \text{ msec} + 2 \text{ msec} = 5 \text{ msec}$
 - Per dischi lenti $\rightarrow 9 \text{ msec} + 5.55 \text{ msec} = 14.55 \text{ msec}$
- **Tempo medio di I/O** = tempo medio di accesso + quantità di dati da trasferire/velocità di trasferimento + overhead

DISPOSITIVI DI MEMORIA NON VOLATILE

- Spesso inseriti in chassis simili agli HDD, e perciò denominati **dischi a stato solido (SSD)** sono costituiti da un controllore e da diversi chip di memoria NAND flash
 - Altre forme includono **unità USB (chiavetta USB, unità flash)**, sostituzioni di dischi DRAM, montaggio su superficie su schede madri e memoria principale in dispositivi come smartphone.
- Può essere più affidabile degli HDD.
- Più costoso per MB.
- Forse hanno una durata di vita più breve - necessitano di un'attenta gestione.
- Meno capacità.
- Ma molto più veloce.
- I bus possono essere troppo lenti e ad esempio connettersi direttamente a PCI.
- Nessuna parte mobile, quindi nessun tempo di ricerca o latenza di rotazione.
- Hanno caratteristiche che aprono a nuove sfide per l'affidabilità:
- Letture e scritture con granularità di "pagina" (analogo del settore):
 - Impossibilità di cancellazione per "sovrascrittura"
 - Il contenuto della pagina deve prima essere cancellato e le cancellazioni avvengono per "blocchi" (della dimensione di diverse pagine) \rightarrow operazione costosa
 - Può essere cancellato solo un numero limitato di volte prima che si esaurisca ~ 100.000
 - Durata misurata in **numero di scritture al giorno (Drive Writes per Day)**
 - Su una NAND da 1 TB di classe 5 DWPD si possono scrivere 5 TB al giorno senza errori per il periodo di garanzia.

NAND flash

- Senza sovrascrittura, i blocchi sono costituiti da un mix di pagine valide e non valide.
- Per tenere traccia dei blocchi logici validi, il controllore mantiene la tabella **FTL (Flash-Translation-Layer)**.
- Implementa anche la **garbage collection** per liberare spazio per cancellare dati non validi.

- Normalmente, si mantiene un **overprovisioning** per fornire spazio di lavoro per la garbage collection.
- Ogni cella ha durata di vita limitata, quindi il **livello di usura** deve essere mantenuto uniforme

valid page	valid page	invalid page	invalid page
invalid page	valid page	invalid page	valid page

Blocco NAND con pagine valide e non valide

MEMORIA VOLATILE

- DRAM usata frequentemente come dispositivo di archiviazione di massa
 - Tecnicamente, non si può definire archiviazione secondaria perché volatile, ma può contenere file system, da utilizzare come storage secondario molto veloce.
- Infatti, le **unità RAM** possono essere utilizzate come dispositivi a blocchi non formattati, ma più spesso contengono un file system.
- I computer hanno il buffering, la memorizzazione nella cache tramite RAM, quindi perché le unità RAM?
 - Cache / buffer allocati / gestiti da programmatore, sistema operativo, hardware
 - Unità RAM sotto il controllo dell'utente
 - Trovato in tutti i principali sistemi operativi
- Usato come memoria temporanea ad alta velocità
 - I programmi possono condividere la data in blocco, rapidamente, leggendo/scrivendo sull'unità RAM

NASTRI MAGNETICI

- Il nastro magnetico è stato utilizzato come primo supporto di archiviazione secondaria. Sebbene non sia volatile e possa contenere grandi quantità di dati, il suo tempo di accesso è lento rispetto a quello della memoria principale e delle unità.

MAPPATURA DEGLI INDIRIZZI

- Le unità a disco vengono indirizzate come giganteschi vettori monodimensionali di **blocchi logici** dove il blocco logico rappresenta la minima unità di trasferimento
 - I blocchi logici sono creati all'atto della formattazione di basso livello.
- L'array di blocchi logici viene mappato sequenzialmente nei settori del disco o sulle pagine di un blocco di una NVM.
- Per esempio:
 - Il settore 0 è il primo settore della prima traccia del cilindro più esterno
 - La corrispondenza prosegue ordinatamente lungo la prima traccia, quindi lungo le rimanenti tracce del primo cilindro, e così via, di cilindro in cilindro, dall'esterno verso l'interno
 - I settori danneggiati vengono esclusi dall'operazione di mappatura e sostituiti con settori di riserva collocati in altre parti della stessa unità.

CONNESSIONE DISCO

- L'unità a disco è connessa al calcolatore per mezzo del **bus di I/O**
- Ci sono diversi tipi di bus, **ATA (Advanced Technology Attachment)**, **SATA (Serial ATA)**, **USB (Universal Serial Bus)**, **SAS (Serial Attached SCSI)**, **FC (Fiber Channel)**.
 - Il più comune è SATA.
- Poiché i bus standard possono essere troppo lenti per gli SSD
 - ➔ Vengono collegati al bus PCI di sistema con tecnologia **NVM express**.
- Il trasferimento di dati in un bus è eseguito da speciali unità di elaborazione, dette **controllori (controllers)**.
 - Controller host sul lato computer del bus, controller dispositivo sul lato dispositivo
 - Il computer esegue il comando sul controller host, utilizzando le porte I/O mappate in memoria

SCHEDULING HDD

- Il SO è responsabile dell'uso efficiente dell'hardware per i dischi ciò significa garantire tempi di accesso contenuti e ampiezze di banda elevate.
- Il tempo di accesso al disco si può scindere in due componenti principali:
 - Tempo di ricerca (seek time) → è il tempo impiegato per spostare la testina sul cilindro che contiene il settore desiderato
 - Latenza di rotazione (rotational latency) → è il tempo necessario perché il disco ruoti fino a portare il settore desiderato sotto la testina.
- L' **ampiezza di banda** del disco è il numero totale di byte trasferiti, diviso per il tempo trascorso fra la prima richiesta e il completamento dell'ultimo trasferimento.

RICHIESTE DI I/O DEL DISCO

- Esistono molte fonti di richiesta di I/O del disco
 - Sistema operativo
 - Processi di sistema
 - Processi degli utenti
- La richiesta di I/O include:
 - modalità di ingresso o uscita
 - indirizzo del disco
 - indirizzo di memoria
 - numero di settori da trasferire
- Il sistema operativo mantiene la coda delle richieste, per disco o dispositivo:
 - Il disco inattivo può funzionare immediatamente su richiesta di I/O
 - Disco occupato significa che il lavoro deve essere messo in coda

➔ Gli algoritmi di ottimizzazione hanno senso solo quando esiste una coda

GESTIONE DELLE CODE

- In passato, sistema operativo era responsabile della gestione delle code e della pianificazione della testina dell'unità disco:
 - Ora, è integrato nei dispositivi di archiviazione, controller
 - Basta fornire Logical Block Addressing (LBA), gestire l'ordinamento delle richieste
- Di seguito vengono descritti alcuni degli algoritmi utilizzati

ALGORITMI PER LO SCHEDULING DEL DISCO

- Esistono diversi algoritmi per pianificare la manutenzione delle richieste di I/O del disco
- Gli algoritmi di scheduling del disco verranno testati sulla coda di richieste per i cilindri (0-199):

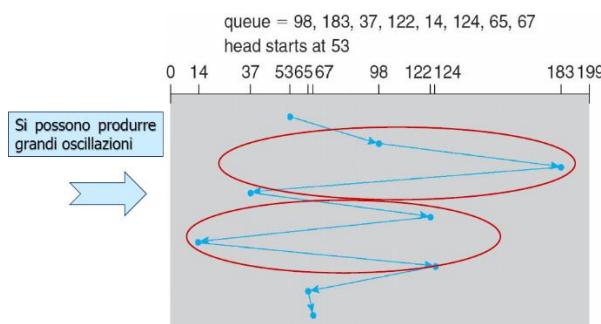
98, 183, 37, 122, 14, 124, 65, 67

La testina dell'unità a disco è inizialmente posizionata sul cilindro 53

- Confronta con algoritmi per spostare un ascensore in un edificio

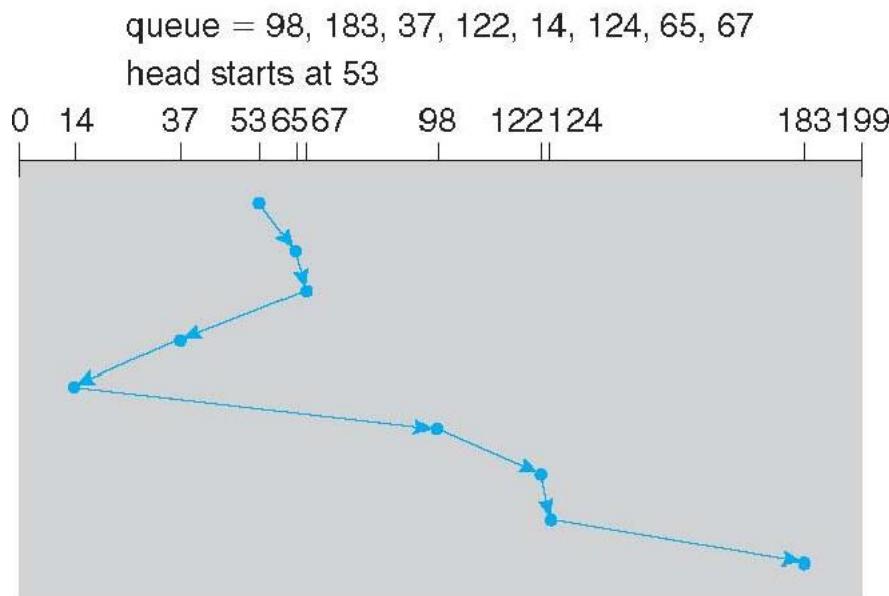
First-Come-First-Served (FCFS)

- FCFS - First Come First Served - è un algoritmo intrinsecamente equo.
 - Si produce un movimento totale della testina pari a 640 cilindri.



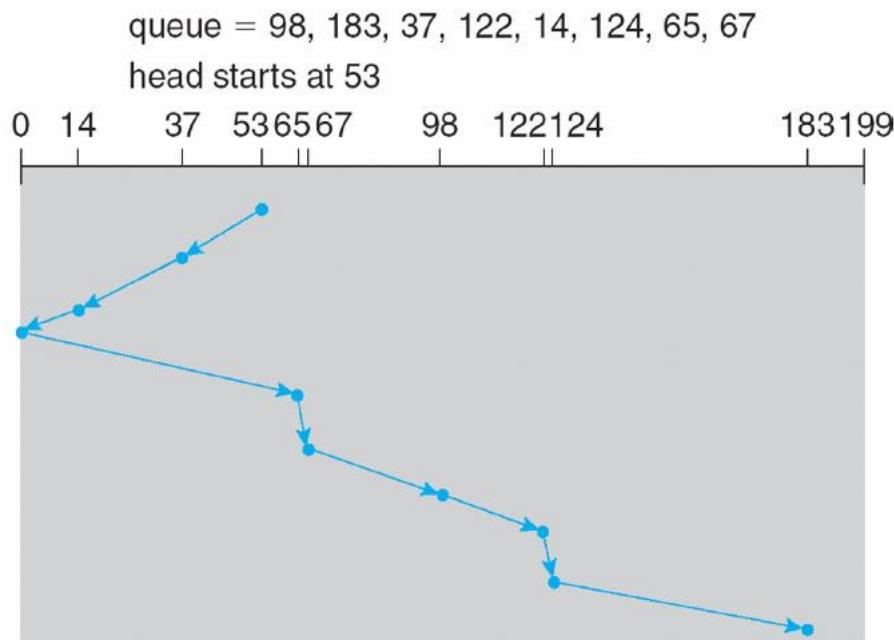
Shortest Seek Time First (SSTF)

- Seleziona la richiesta con il tempo di ricerca minimo dalla posizione attuale della testa
- Lo scheduling SSTF è una forma di scheduling SJF; può causare lo starvation di alcune richieste
- Esempio - movimento totale della testata di 236 cilindri



ALGORITMO SCAN

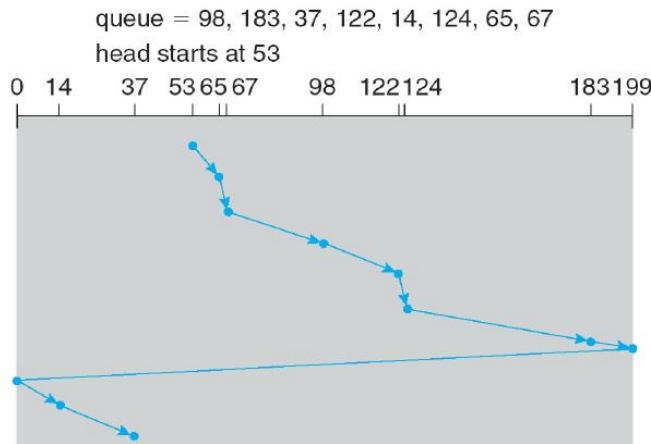
- Il braccio della testina si muove da un estremo all'altro del disco, servendo sequenzialmente le richieste giunto ad un estremo inverte la direzione di marcia e, conseguentemente, l'ordine di servizio
- È chiamato anche algoritmo dell'ascensore
- Se gli accessi sono distribuiti uniformemente, quando la testina inverte il proprio movimento, la maggior densità di richieste si ha all'estremo opposto del disco.
 - ➔ Tali richieste avranno anche i tempi più lunghi di attesa di servizio.
- L'illustrazione mostra il movimento totale della testa di 208 cilindri



SCAN CIRCOLARE (C-SCAN)

- Garantisce un tempo di attesa più uniforme rispetto a SCAN.
- La testina si muove da un estremo all'altro del disco servendo sequenzialmente le richieste
- Quando raggiunge l'ultimo cilindro ritorna immediatamente all'inizio del disco, senza servire richieste durante il viaggio di ritorno.

- Considera i cilindri come organizzati secondo una lista circolare, con l'ultimo cilindro adiacente al primo.
- Si ha un movimento totale pari a 383 cilindri



SCELTA DI UN ALGORITMO DI SCHEDULING

- SSTF è comune e ha un fascino naturale
- SCAN e C-SCAN forniscono buone prestazioni in sistemi che utilizzano intensamente le unità a disco
 - Meno starvation, ma ancora possibile
- Le prestazioni dipendono comunque dal numero e dal tipo di richieste.

ALGORITMO:

FCFS	640
SSTF	236
SCAN	208
C-SCAN	382

MOVIMENTI IN CILINDRI:

FCFS	640
SSTF	236
SCAN	208
C-SCAN	382

SCHEDULING IN UNIX

- Per evitare la starvation, Linux implementa lo scheduler **deadline** (scadenza).
 - Mantiene le code di lettura e scrittura separate, assegna la priorità di lettura
 - ➔ Perché è più probabile che i processi si blocchino in lettura che in scrittura
 - Implementa quattro code: 2 in lettura e 2 in scrittura:
 - ➔ 1 coda di lettura e 1 di scrittura ordinate in ordine LBA, implementando essenzialmente C-SCAN
 - ➔ 1 coda di lettura e 1 di scrittura ordinate in ordine FCFS
 - ➔ Tutte le richieste di I/O inviate al batch sono ordinate nell'ordine di quella coda
 - ➔ Dopo ogni batch, controlla se ci sono richieste in FCFS precedenti al momento configurato(predefinito 500 ms).
- In tal caso, la coda Logical Block Addressing (LBA) contenente tale richiesta viene selezionata per il batch successivo di I/O
- In Red Hat (RHEL 7) sono disponibili anche **NOOP** e **completamente fair queuing scheduler (CFQ)**, le impostazioni predefinite variano in base al dispositivo di archiviazione.

SCHEDULING SU NVM

- Nelle unità NVM, dove non esistono parti mobili, si utilizza di solito una politica FCFS
 - L'unica ottimizzazione possibile riguarda il servizio combinato di richieste di scrittura relative a indirizzi logici adiacenti.
- Tuttavia, il vantaggio dei dispositivi NVM è meno sensibile in caso di accesso sequenziale – contrariamente al tempo di ricerca sugli HDD che, in questo caso, è ridotto al minimo
 - Prestazioni equivalenti o anche peggiori in caso di elevata usura del dispositivo

- Problema dell'amplificazione di scrittura quando si attivano operazioni aggiuntive per la garbage collection.

RILEVAMENTO E CORREZIONE DEGLI ERRORI

- Il **rilevamento degli errori** determina se si è verificato un problema (ad esempio un bit flipping)

- A fronte del verificarsi di un errore, il sistema può interrompere l'operazione prima che l'errore venga propagato.
- Rilevazione eseguita frequentemente tramite bit di parità
- La parità è una forma di **checksum** che utilizza l'aritmetica modulare per calcolare, archiviare, confrontare valori su parole a lunghezza fissa
- Un altro metodo di rilevamento degli errori comune nelle reti è il **controllo di ridondanza ciclica (CRC)** che utilizza una funzione hash per rilevare errori su più bit.
- Il codice di **correzione degli errori (ECC)** non solo rileva, ma può correggere alcuni errori.
 - Errori soft correggibili, errori hard rilevati ma non corretti

GESTIONE DELL'UNITA' A DISCO

- **Formattazione di basso livello o fisica:**
 - Solitamente, 512 byte o 4Kb di dati ma possono essere selezionabili.
 - Dimensione standard pari a 4 KB
 - Si suddivide il disco in settori, che possono essere letti e scritti dal controllore del disco.
 - In entrambi i casi, la formattazione di basso livello inserisce nel dispositivo una speciale struttura dati per ogni "blocco" di memoria:
 - Intestazione, dati, coda
 - L'intestazione e la coda contengono informazioni (numero settore/pagina, codice **ECC**) ad uso del controllore.
- Per poter impiegare un dispositivo per memorizzare i file, il SO deve mantenere le proprie strutture dati sul disco (HDD/SSD):
 - Si **partiziona** il dispositivo in uno o più gruppi di cilindri/pagine, ognuno dei quali rappresenta un "disco logico".
 - **Formattazione logica** o "creazione di un file system"
 - ➔ Strutture dati del SO per la descrizione dello spazio libero/occupato e creazione di una directory iniziale vuota.
 - Per migliorare le prestazioni, la maggior parte dei file system accoppia i blocchi in gruppi, detti **cluster**.
 - ➔ I/O su disco fatto per blocchi
 - ➔ I/O via file system fatto per cluster (accesso sequenziale)
 - ➔ File e metadati vicini su HDD per diminuire i movimenti della testina.
- La **partizione di root** contiene il SO altre partizioni possono contenere altri SO, altri file system o essere partizioni raw
 - Viene **montata** all'avvio del sistema
 - Altre partizioni possono essere montate automaticamente o manualmente (al boot o successivamente)
- Al momento del montaggio di ogni partizione, si verifica la coerenza del file system (controllando la correttezza dei metadati) :
 - Se no, si corregge.
 - Se sì, si aggiorna la tabella di montaggio
- Nel boot block sono contenute le informazioni necessarie all'inizializzazione del sistema.
- In Windows si chiama MBR (Master Boot Record):
 - Esecuzione del codice del **bootstrap loader** contenuto nel firmware.
 - Lettura/esecuzione del codice contenuto nell'MBR, che contiene anche una tabella delle partizioni, con un flag che identifica la partizione di boot
 - Caricamento, dalla partizione di boot, del kernel e dei sottosistemi del SO.

GESTIONE DELL'AREA DI SWAP

- Utilizzato per spostare interi processi (swapping) o pagine (pagine) dalla DRAM allo storage secondario quando la DRAM non è abbastanza grande per tutti i processi
- Il sistema operativo fornisce la **gestione dello spazio di scambio**
 - Storage secondario più lento della DRAM, quindi importante per ottimizzare le prestazioni
 - Solitamente sono possibili più spazi di swap - diminuendo il carico di I/O su un dato dispositivo
 - Meglio avere dispositivi dedicati
 - Può essere in una partizione grezza o un file all'interno di un file system (per comodità di aggiunta)
 - Strutture dati per lo scambio su sistemi Linux:

MEMORIA SECONDARIA CONNESSA ALLA MACCHINA

- I computer accedono alla memoria in tre modi
 - Collegamento all'host
 - Collegamento alla rete
 - Cloud
- Accesso all'host tramite porte I/O locali, utilizzando una delle numerose tecnologie
 - Per accedere ad un maggiore spazio di archiviazione, si utilizza bus di archiviazione come USB, firewire, thunderbolt.
 - I sistemi di fascia alta utilizzano **Fibre Channel (FC)**
 - ➔ Architettura seriale ad alta velocità che utilizza cavi in fibra o in rame.
 - ➔ Più host e dispositivi di archiviazione possono connettersi alla struttura FC.

MEMORIA SECONDARIA CONNESSA ALLA RETE

- Un dispositivo di memoria secondaria connessa alla rete (Network Attached Storage) **NAS** è un sistema di memoria specializzato al quale si accede in modo remoto attraverso la rete di trasmissione dati.
- I client accedono alla memoria connessa alla rete tramite un'interfaccia RPC, supportata da protocolli quali NFS (UNIX) e CIFS (Windows).
- Le chiamate di procedura remota sono implementate per mezzo di TCP o UDP, sopra una rete IP, di solito la stessa rete che supporta tutto il traffico dei dati fra client e server.
- Il protocollo **iSCSI** utilizza la rete IP per trasportare il protocollo SCSI
 - Collegamento remoto ai dispositivi (blocchi)

MEMORIA SECONDARIA IN CLOUD

- Similmente al NAS, fornisce l'accesso allo storage tramite rete
 - A differenza del NAS, l'accesso al data center remoto avviene tramite Internet o WAN
- NAS si presenta come un altro file system, mentre lo storage cloud è basato su API, con programmi che utilizzano le API per fornire l'accesso.
 - Esempi di cloud storage includono Dropbox, Amazon S3, Microsoft OneDrive, Apple iCloud e Google drive
 - Si impiegano le API a causa delle lunghe latenze e per i numerosi scenari di errore che sono comuni sulle WAN.

STORAGE ARRAY

- Può semplicemente allegare dischi o array di dischi
- Evita lo svantaggio NAS dell'utilizzo della larghezza di banda di rete
- L'array di archiviazione ha controller, fornisce funzionalità agli host collegati
 - Porte per connettere gli host all'array
 - Memoria, software di controllo (a volte NVRAM, ecc.)
 - Da pochi a migliaia di dischi
 - RAID, hot spare, hot swap.
 - Archiviazione condivisa → maggiore efficienza
 - Funzionalità presenti in alcuni file system

STORAGE AREA NETWORK

- Reti private (che impiegano protocolli specifici per la memorizzazione) tra server e unità di memoria secondaria.
- Comune negli ambienti di archiviazione di grandi dimensioni.
- Flessibilità: si possono connettere alla stessa SAN molti calcolatori e molti storage array
- SAN è uno o più array di archiviazione
 - Connesso a uno o più switch Fibre Channel o rete **InfiniBand (IB)**.
- Gli host si collegano anche agli switch
- Storage reso disponibile tramite **Masking Logical Unit Number (LUN)** da array specifici a server specifici.
- Facile da aggiungere o rimuovere spazio di archiviazione, aggiungere nuovo host e allocare spazio di archiviazione.
- Perché avere reti di archiviazione e reti di comunicazione separate?
 - Considera iSCSI, Fibre Channel over Ethernet (FCOE)

STRUTTURE RAID

- **RAID, Redundant Array Of Inexpensive Disks** - l'affidabilità del sistema di memorizzazione viene garantita tramite la **ridondanza**
- Aumento del **tempo medio di guasto**
- Spesso affiancati dalla presenza di **NVRAM** per garantire la consistenza dei dati scritti "contemporaneamente" su dischi multipli e per migliorare le performance
- Inoltre le tecniche per aumentare la velocità di accesso al disco implicano l'uso di più dischi cooperanti.
- Il sezionamento del disco o data striping RAID 0 tratta un gruppo di dischi come un'unica unità di memorizzazione.
 - Ogni "blocco" di dati è suddiviso in "sottoblochi" memorizzati su dischi distinti (es i bit di ciascun byte possono essere letti "in parallelo" su 8 dischi).
 - Il tempo di trasferimento per rotazioni sincronizzate diminuisce proporzionalmente al numero dei dischi nella batteria.
- Gli schemi RAID migliorano prestazioni ed affidabilità del sistema memorizzando dati ridondanti
 - Il mirroring o shadowing RAID 1 conserva duplicati di ciascun disco

ALTRE CARATTERISTICHE

Indipendentemente da dove RAID viene implementato, è possibile aggiungere altre utili funzionalità:

- **L'istantanea** è una vista del file system prima che avvenga una serie di modifiche (ad esempio, in un determinato momento)
- La **replica** è la duplicazione automatica delle scritture tra siti separati
 - Per ridondanza e ripristino di emergenza
 - Può essere sincrono o asincrono
- Il **disco hot spare** non è utilizzato, viene utilizzato automaticamente dalla produzione RAID se un disco non riesce a sostituire il disco guasto e ricostruire il set RAID se possibile.
 - Diminuisce il tempo medio di riparazione

ESTENSIONI

- RAID da solo non previene o rileva il danneggiamento dei dati o altri errori, ma solo guasti del disco
- Solaris ZFS aggiunge i checksum di tutti i dati e metadati
 - Checksum conservati con puntatore all'oggetto, per rilevare se l'oggetto è quello giusto e se è cambiato
 - Può rilevare e correggere il danneggiamento di dati e metadati
- ZFS rimuove anche volumi, partizioni
 - Dischi allocati nei pool
 - I filesystem con un pool condividono quel pool, usano e rilasciano spazio come malloc() e free()
allocare/rilasciare chiamate di memoria

OBJECT STORAGE

- Elaborazione generica, file system non sufficienti per una scala molto ampia
- Un altro approccio: iniziare con un pool di archiviazione e posizionare gli oggetti al suo interno
 - Oggetto solo un contenitore di **dati**
 - Non c'è modo di navigare nel pool per trovare oggetti (nessuna struttura di directory) pochi servizi
 - Orientato al computer, non orientato all'utente
- Sequenza tipica
 - Creare un oggetto all'interno del pool, ricevere un ID oggetto
 - Accedere all'oggetto tramite quell'ID
 - Elimina oggetto tramite quell'ID
- Il software di gestione dell'archiviazione degli oggetti come il **file system Hadoop (HDFS)** e **Ceph** determina dove archiviare gli oggetti, gestisce la protezione.
 - In genere, memorizzando N copie, su N sistemi, nel cluster di archiviazione oggetti
 - **Scalabile orizzontalmente**
 - **Contenuto indirizzabile, non strutturato**

--- CAPITOLO 14 - IMPLEMENTAZIONE DEI FILE SYSTEM ---

STRUTTURA DEL FILE SYSTEM

- Struttura del file:
 - Unità di memorizzazione logica
 - Collezione di informazioni correlate
 - File control block struttura dati del kernel che "descrive" il file
 - Il **file system** risiede nella memoria secondaria
 - Fornisce una semplice interfaccia per la memorizzazione non volatile, realizzando il mapping fra indirizzi logici e fisici
 - Fornisce la possibilità di accedere alla memoria in maniera efficiente, per memorizzare e recuperare rapidamente le informazioni.
 - I dischi sono un mezzo conveniente per la memorizzazione di file perché:
 - Si possono riscrivere localmente si può leggere un blocco dal disco, modificarlo e quindi scriverlo nella stessa posizione.
 - È possibile accedere direttamente a qualsiasi **blocco** di informazioni del disco, quindi a qualsiasi file, sia in modo sequenziale che diretto (spostando le testine di lettura/scrittura ed attendendo la rotazione del disco).
 - Le operazioni di I/O su disco avvengono con "granularità" di blocco:
 - Ciascun blocco è composto da uno o più settori (solitamente 512 byte per settore)
 - **File control block (FCB):** struttura dati del kernel che "descrive" il file
 - **I dispositivi fisici** vengono controllati da device driver
- Il file system è stratificato cioè organizzato in livelli
- Ogni livello si serve delle funzioni dei livelli inferiori per creare nuove, impiegate dai livelli superiori: (collegarli con le frecce)

Application programs
Logical file system
File-organization module
Basic file system
I/O control
Device

LIVELLI DEL FILE SYSTEM

- **Controllo dell'I/O - Driver dei dispositivi**
 - Traducono comandi di alto livello ("recupera il blocco fisico 123") in sequenze di bit che guidano l'hardware di I/O a compiere una specifica operazione in una data locazione.

- In altre parole scrivono specifiche configurazioni di bit in specifiche locazioni della memoria del controllore (microcodice del controllore) per indicare al dispositivo che operazioni compiere e dove.
- **File system di base**
 - Invia comandi generici al driver di dispositivo (“leggi dal l’unità 1 cilindro 73 traccia 2 settore 10 nella locazione di memoria 1060”) per leggere/scrivere blocchi fisici su disco
 - Gestisce il buffer del dispositivo e la cache che conserva i blocchi del file e i metadati (in RAM).
- **Modulo di organizzazione dei file**
 - Traduce gli indirizzi logici di blocco in indirizzi fisici
 - Contiene il modulo per la gestione dello spazio libero
- **File system logico**
 - Gestisce i metadati cioè tutte le strutture del file system eccetto i dati veri e propri memorizzati nei file.
 - ➔ Mantiene le strutture di file tramite i file control block FCB (**inode** in UNIX) che contengono le informazioni sui file, quali proprietario, permessi, posizione del contenuto
 - ➔ Gestisce le directory
 - ➔ Gestisce protezione e sicurezza
- La struttura a strati è utile per ridurre la complessità e la ridondanza, ma aggiunge overhead e può diminuire le performance.
- I livelli logici possono essere implementati con qualsiasi metodo di codifica in base al progettista del sistema operativo.
- Molti file system, a volte molti all'interno di un sistema operativo, ognuno con il proprio formato:
 - CD-ROM è ISO 9660;
 - Unix ha **UFS**, **FFS**;
 - Windows ha **FAT**, **FAT32**, **NTFS** così come floppy, CD, DVD Blu-ray
 - Linux ha più di 130 tipi, con file system estesi **ext3** ed **ext4** in testa; **più file system distribuiti**, ecc.
 - Nuovi ancora in arrivo: **ZFS**, **GoogleFS**, **Oracle ASM**, **FUSE**

OPERAZIONI SU FILE SYSTEM

- Le funzioni del file system vengono realizzate tramite chiamate di sistema invocate attraverso la API, che utilizzano dati, gestiti dal kernel, residenti sia su disco che in memoria.
- Strutture dati del file system residenti su disco:
 - **Blocco di controllo di avviamento** contiene le informazioni per l'avviamento di un SO da quel volume ➡ boot block nell'UFS, nei sistemi Windows è il partition boot sector.
 - Necessario se il volume contiene un SO (vuoto altrimenti)
 - Normalmente è il primo blocco del volume
 - **Blocchi di controllo dei volumi** contengono dettagli riguardanti la partizione, quali numero totale dei blocchi e loro dimensione, contatore dei blocchi liberi e relativi puntatori, contatore degli FCB liberi e relativi puntatori - superblocco in UFS, master file table MFT nell'NTFS.
 - **Strutture delle directory** usate per organizzare i file (in UFS comprendono i nomi dei file ed i numeri di inode associati).

FILE CONTROL BLOCK

- Blocchi di controllo dei file **FCB** (inode nell'UFS), contengono dettagli sul file.
- Identificativo del file, permessi, dimensione, date di creazione/ultimo accesso/ultima modifica, puntatori ai blocchi di dati.

File permission
 File dates (create, access, write)
 File owner, group, ACL
 File size
 File data blocks or pointer to file data blocks

STRUTTURE DATI DEL FILE SYSTEM RESIDENTI IN MEMORIA

- **Tabella di montaggio:** contiene le informazioni relative a ciascun volume montato

- **Struttura delle directory:** contiene informazioni relative a tutte le directory cui i processi hanno avuto accesso di recente (cache)
- **Tabella dei file aperti:** contiene una copia dell'FCB per ciascun file aperto nel sistema
- **Tabella dei file aperti per ciascun processo:** contiene un puntatore all'elemento corrispondente nella tabella generale, più informazioni di accesso specifiche del processo.

REALIZZAZIONE DELLE DIRECTORY

- **Lista lineare** di nomi di file con puntatori ai blocchi di dati
 - Semplice da implementare
 - Esecuzione onerosa dal punto di vista del tempo di ricerca (complessità lineare nel numero di elementi contenuti nella directory)
 - Lista ordinata (o B- \ominus albero) migliora il tempo di ricerca, ma l'ordinamento deve essere mantenuto a fronte di ogni inserimento/cancellazione
- **Tabella hash** lista lineare con struttura hash
 - Migliora il tempo di ricerca nella directory
 - Inserimento e cancellazione costano $O(1)$ se non si verificano collisioni
 - Collisione situazione in cui due nomi di file generano lo stesso indirizzo hash nella tabella
 - Dimensione fissa e necessità di rehash o liste di trabocco

METODI DI ALLOCAZIONE

- Il metodo di allocazione dello spazio su disco descrive come i blocchi fisici del disco vengono allocati ai file:
 - Allocazione contigua
 - Allocazione concatenata
 - Allocazione indicizzata

ALLOCAZIONE CONTIGUA

- Ciascun file occupa un insieme di blocchi contigui sul disco.
 - Le migliori prestazioni nella maggior parte dei casi
 - Per reperire il file occorrono solo la locazione iniziale (blocco iniziale) e la lunghezza (numero di blocchi)
 - Problemi:
 - Allocazione dinamica dello spazio disco
 - Frammentazione esterna
 - Necessità di conoscere a priori la dimensione dei file (i file non possono crescere, soprattutto per best-fit)
 - Compattazione dello spazio disco

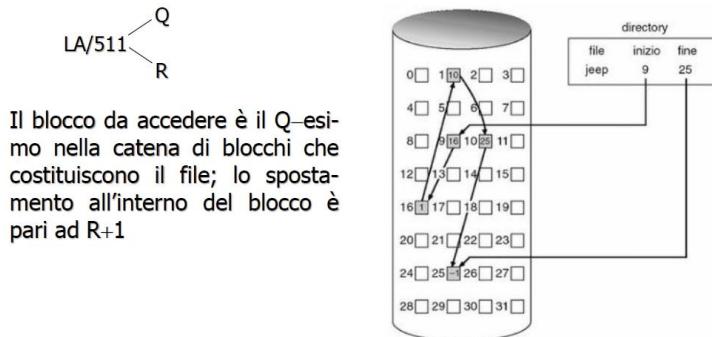
SISTEMI BASATI SULL'EXTENT

- Ciascun file consiste di uno o più extent (di dim. variabile ed eventualmente definita dall'utente)
- Molti file system più recenti (ad es. Veritas File System) utilizzano uno schema di allocazione contiguo modificato
- I file system basati sull'extent allocano i blocchi del disco nelle estensioni.
- Un **extent** è una “porzione di spazio contiguo”
 - Le extent vengono allocate per l'allocazione dei file
 - Un file è costituito da una o più extent

ALLOCAZIONE CONCATENATA - LINKED ALLOCATION

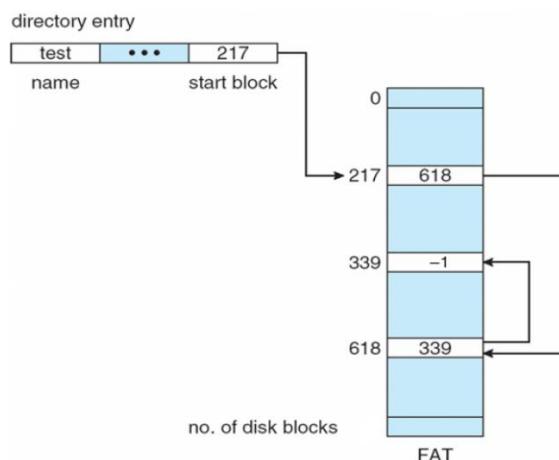
- Ciascun file è una lista concatenata di blocchi i blocchi possono essere sparsi ovunque nel disco:
 - Ciascun blocco contiene un puntatore al blocco successivo
 - Il file termina quando si incontra un blocco con puntatore vuoto
 - Non si ha spreco di spazio (no frammentazione esterna)
 - Quando necessita di un nuovo blocco da allocare ad un file (il file può crescere), il SO invoca il sottosistema per la gestione dello spazio libero.
 - L'efficienza può essere migliorata raccogliendo i blocchi in cluster (però, la frammentazione interna).

- Problemi:
 - Accesso casuale impossibile reperire un blocco può richiedere molte operazioni di I/O ed operazioni di seek.
 - Affidabilità legata ai puntatori
- Nella directory, si mantiene l'indirizzo dei blocchi iniziale e finale
- Mappatura da indirizzi logici a indirizzi fisici



File Allocation Table (FAT)

- L'inizio del volume ha una tabella, indicizzata per numero di blocco
- Molto simile a un elenco collegato, ma più veloce su disco e memorizzabile nella cache
- Nuova allocazione dei blocchi semplice



La FAT consente la memorizzazione "localizzata" dei puntatori

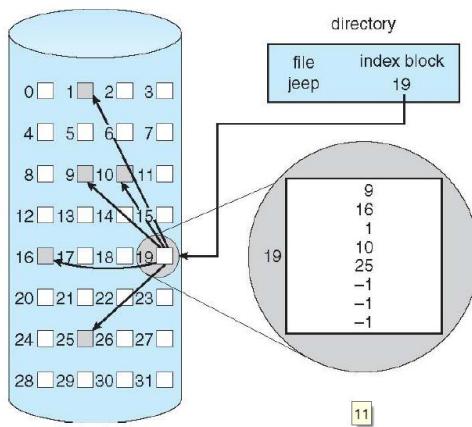
ALLOCAZIONE INDICIZZATA - Indexed Allocation Method

- Per ogni file, si collezionano tutti i puntatori in un **unico blocco indice**

- ❖ Per ogni file, si collezionano tutti i puntatori in un unico **blocco indice**
- ❖ Richiede una tabella indice
- ❖ Accesso casuale
- ❖ Permette l'accesso dinamico senza frammentazione esterna; tuttavia c'è il sovraccarico temporale di accesso al blocco indice
- ❖ Nella directory si memorizza l'indirizzo del blocco indice
- ❖ Mappatura da indirizzi logici a indirizzi fisici per file di dim. max 256K parole e con dimensione di blocco di 512 parole: occorre un solo blocco indice



LA/512 Q spostamento nella tabella indice
R spostamento all'interno del blocco



ALLOCAZIONE INDICIZZATA - Indexed Allocation Method - COMMENTI

- Richiede una tabella indice
- Accesso casuale
- Permette l'accesso dinamico senza frammentazione esterna tuttavia c'è il sovraccarico temporale di accesso al blocco indice
- Mappatura da indirizzi logici a indirizzi fisici per file di dim. max 256K parole e con dimensione di blocco di 512 parole occorre un solo blocco indice.
- Mapping fra indirizzi logici e fisici per un file di lunghezza qualunque (dim blocco 512 byte/word)
- Schema concatenato - Si collegano blocchi della tabella indice
- Indicizzazione multilivello

ALLOCAZIONE INDICIZZATA - Indexed Allocation Method - SCHEMA

- ❖ Mapping fra indirizzi logici e fisici per un file di lunghezza qualunque (dim. blocco 512 byte/word)
- ❖ **Schema concatenato** — Si collegano blocchi della tabella indice
 - Il primo blocco indice contiene l'insieme degli indirizzi dei primi 511 blocchi del file, più un puntatore al blocco indice successivo

$LA/(512 \times 511)$ Q_1 spostamento nella tabella indice
 R_1 R_1 si utilizza come segue...

$R_1/512$ Q_2 spostamento nel blocco della tabella indice
 R_2 R_2 spostamento all'interno del blocco del file

ALLOCAZIONE INDICIZZATA - Indexed Allocation Method - SCHEMA A DUE LIVELLI

- ❖ **Indice a più livelli**
 - Indice a due livelli

$LA / (512 \times 512)$ Q_1 Q_1 spostamento nell'indice esterno
 R_1 R_1 utilizzato come segue...

$R_1 / 512$ Q_2 Q_2 spostamento nel blocco della tabella indice
 R_2 R_2 spostamento nel blocco del file

- Blocchi da 4K possono contenere 1024 puntatori da 4 byte nell'indice esterno → per un totale di 1048567 blocchi di dati e file di dimensione massima pari a 4GB

PERFORMANCE

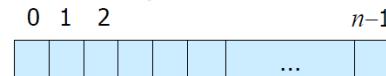
- Il miglior metodo per l'allocazione di file dipende dal tipo di accesso
 - L'allocazione contigua ha ottime prestazioni sia per accesso sequenziale che casuale
 - L'allocazione concatenata si presta naturalmente all'accesso sequenziale

- ➔ Dichiарando il tipo di accesso all'atto della creazione del file, si può selezionare il metodo di allocazione più adatto
- L'allocazione indicizzata è “più complessa”
 - L'accesso ai dati del file può richiedere più accessi a disco (nel caso di un indice a due livelli)
 - Tecniche di clustering possono migliorare il throughput, riducendo l'overhead di CPU.

GESTIONE DELLO SPAZIO LIBERO

- Per tenere traccia dello spazio libero in un disco, il sistema conserva una **lista dello spazio libero**
- **Vettore di bit o bitmap** (n blocchi)

❖ **Vettore di bit o bitmap** (n blocchi)



$$\text{bit}[i] = \begin{cases} 1 & \Rightarrow \text{blocco}[i] \text{ libero} \\ 0 & \Rightarrow \text{blocco}[i] \text{ occupato} \end{cases}$$

Lista concatenata:

- Si collegano tutti i blocchi liberi mediante puntatori e si mantiene un puntatore alla testa della lista in memoria centrale
- Ogni blocco contiene un puntatore al successivo blocco libero
- Non si spreca spazio (solo un puntatore)
- Non è necessario attraversare tutta la lista, poiché di solito la richiesta è relativa ad un singolo blocco
- Non facile da usare per ottenere spazio contiguo
- Grouping:
 - Realizzazione di una lista di blocchi
 - Sul primo blocco memorizzazione degli indirizzi di n blocchi liberi; $n-1$ blocchi sono effettivamente liberi, l' n -esimo contiene gli indirizzi di altri n blocchi, etc...
- Conteggio:
 - Poiché lo spazio viene spesso allocato e liberato in modo contiguo (nell'allocazione contigua, per gli extent o nel caso di cluster)
 - si mantiene una lista contenente un indirizzo del disco ed un contatore, che indica il numero di blocchi liberi contigui (più informazione su ogni elemento della lista, ma lista più breve).

TRIM DEI BLOCCHI NON UTILIZZATI

- TRIM è un comando ATA (Advanced Technology Attachment) che consente al SO di informare il controller di un SSD su quali blocchi di dati può cancellare, perché non sono più in uso
- TRIM è complementare alla garbage collection
 - Elimina la copiatura di pagine di dati scartate o non valide durante il processo di garbage collection per risparmiare tempo e migliorare le prestazioni dell'unità SSD
 - L'SSD ha un minor numero di pagine da spostare durante la garbage collection, il che riduce il numero totale di cicli di programmazione/cancellazione sul supporto flash NAND e prolunga la vita dell'SSD.
 - Inoltre la garbage collection può essere effettuata prima che il dispositivo sia completamente pieno (o quasi).

EFFICIENZA E PRESTAZIONI

- L'efficienza del file system dipende da
 - Tecniche di allocazione del disco e algoritmi di realizzazione/gestione delle directory
 - Preallocazione delle strutture necessarie a mantenere i metadati
 - Tipi di dati conservati nell'elemento della directory corrispondente al file
 - Strutture dati a lunghezza fissa o variabile
- Le prestazioni dipendono da

- Disporre di **buffer cache** cioè sezioni dedicate della memoria in cui si conservano i blocchi usati di frequente.
- Scritture **sincrone** talvolta richieste dalle applicazioni o necessarie al sistema operativo
- Impossibilità di buffering/caching l'operazione di scrittura su disco deve essere completata prima di proseguire l'esecuzione
- Le scritture **asincrone** che sono le più comuni, sono invece bufferizzabili (e più veloci)
- Le operazioni di lettura risultano talvolta più lente delle scritture
- Utilizzo di tecniche di **svuotamento/riempimento delle cache** per ottimizzare l'accesso sequenziale

RIPRISTINO (RECOVERY)

- **Verifica della coerenza:** confronta i dati nella struttura della directory con i blocchi di dati sul disco e tenta di correggere le incoerenze
 - Può essere lento e talvolta fallisce
- Utilizzare i programmi di sistema per eseguire il **backup** dei dati dal disco a un altro dispositivo di archiviazione (nastro magnetico, altro disco magnetico, ottico)
- Recupera file o disco persi **ripristinando** i dati dal backup

LOG STRUTTURATO DEI FILE SYSTEM

- I file system **strutturati di log** (o journaling) registrano ogni aggiornamento dei metadati nel file system come **transazione**
- Tutte le transazioni vengono scritte in un registro
 - Una transazione è considerata confermata una volta che è stata scritta nel log (in sequenza)
 - A volte su un dispositivo separato o una sezione del disco
 - Tuttavia, il file system potrebbe non essere ancora aggiornato
- Le transazioni nel registro vengono scritte in modo asincrono nelle strutture del file system
 - Quando le strutture del file system vengono modificate, la transazione viene rimossa dal registro
- Se il file system si arresta in modo anomalo, tutte le restanti transazioni nel registro devono comunque essere eseguite
- Recupero più rapido da crash, elimina la possibilità di incoerenza dei metadati