

DISCLAIMER

Il presente documento è una rielaborazione personale delle slide del corso di Sistemi Operativi del Professore Ferretti. I contenuti sono stati trascritti e organizzati con il solo scopo di facilitare lo studio personale della materia.

Sebbene il documento possa essere liberamente condiviso a scopo di studio, l'autore non si assume alcuna responsabilità riguardo alla completezza e all'accuratezza dei contenuti. Si invitano pertanto gli utilizzatori a verificare sempre le informazioni consultando direttamente il materiale didattico originale fornito dal Prof. Ferretti.

Questo documento non intende sostituire in alcun modo il materiale ufficiale del corso, ma rappresenta esclusivamente uno strumento di supporto allo studio individuale.

Qualora vengano riscontrati errori o imprecisioni, siete liberi di modificare i file .tex e aprire una pull request su GitHub per proporre le vostre correzioni, nell'interesse di tutti gli utilizzatori.

Sistemi Operativi

Università di Bologna sede di Cesena - Corso di Laurea
Triennale in Ingegneria Informatica

Cattolico Giuseppe

8 febbraio 2025

Indice

1	Introduzione ai Sistemi Operativi	4
1.1	Cos'è un sistema operativo?	5
1.1.1	Tipi di sistemi	6
1.2	Concetti base di architetture degli elaboratori	7
1.2.1	Interrupt	7
1.2.2	Comunicazione fra processore e dispositivi di I/O	9
1.2.3	Memoria	10
1.2.4	Protezione Hardware	11
1.3	Struttura dei sistemi operativi (panoramica)	13
1.3.1	Architettura dei sistemi operativi	13
1.3.2	System Call	14
1.3.3	File Speciali (UNIX)	16
2	Scheduling	17
2.1	Scheduler, Processi e Thread	18
2.1.1	Introduzione	18
2.1.2	Descrittori dei processi	18
2.1.3	Scheduler	19
2.1.4	Mode switching e Context switching	20
2.1.5	Processi e Thread	20
2.1.6	Multithreading	22
2.2	Schedule	23
2.2.1	Rappresentazione degli schedule	23
2.3	Algoritmi di scheduling	24
2.3.1	First Come, First Served (FCFS)	24
2.3.2	Shortest Job First (SJF)	25
2.3.3	Round-Robin	26
2.3.4	Scheduling a priorità	26
2.3.5	Scheduling a classi di priorità	27
2.3.6	Scheduling multilivello	27
2.4	Scheduling Real-Time	27
2.4.1	Esempi di Scheduler RT	28
3	Concorrenza	29
3.1	Introduzione alla concorrenza	30
3.1.1	Modello concorrente	30
3.1.2	Processi	30
3.1.3	Multiprogramming e multiprocessing	30
3.2	Interazioni tra processi	31
3.2.1	Tipi di interazione	31
3.2.2	Proprietà fondamentali	32
3.2.3	Mutua esclusione	32
3.2.4	Deadlock (stallo)	33
3.2.5	Starvation (inedia)	33
3.2.6	Azioni atomiche	33
3.3	Sezioni critiche	34
3.3.1	Tecniche software: Algoritmo di Dekker	36
3.3.2	Tecniche software: Algoritmo di Peterson	37
3.3.3	Algoritmo di Peterson – Generalizzazione per N processi	38

3.3.4	Tecniche hardware: disabilitazione interrupt, istruzioni speciali	38
3.4	Semafori	39
3.4.1	Definizione	40
3.4.2	Implementazione	41
3.4.3	Semafori binari	41
3.5	Problemi classici	43
3.5.1	Producer/Consumer	43
3.5.2	Bounded Buffer	43
3.5.3	Cena dei Filosofi	44
3.5.4	Lettori e scrittori	45
3.5.5	Come derivare una soluzione basata su semafori	46
3.6	Conditional Critical Regions (CCR)	47
3.6.1	Introduzione	47
3.6.2	CCR tramite semafori (passing the baton)	49
3.6.3	Implementazione di semafori tramite CCR	49
3.6.4	CCR - Produttore / Consumatore	49
3.6.5	CCR - Filosofi a cena	50
3.6.6	CCR - R/W Priorità ai lettori	50
3.6.7	CCR - R/W Priorità agli scrittori	51
3.6.8	CCR - Bounded buffer (accesso esclusivo)	51
3.7	Monitor	52
3.7.1	Introduzione	52
3.7.2	Implementazione dei semafori	53
3.7.3	Monitor - R/W	54
3.7.4	Monitor - Produttore / consumatore	56
3.7.5	Monitor - Buffer limitato	57
3.7.6	Monitor - Filosofi a cena (no deadlock)	58
3.7.7	Implementazione dei monitor tramite semafori	59
3.8	Message passing	60
3.8.1	Introduzione	60
3.8.2	MP sincrono	60
3.8.3	MP asincrono	61
3.8.4	MP completamente asincrono	61
3.8.5	MP sincrono dato quello asincrono	62
3.8.6	Message Passing - Filosofi a cena	63
3.8.7	Message Passing - Produttori e consumatori	64
3.9	Conclusioni	64
3.9.1	Riassunto	64
3.9.2	Potere espressivo	64
4	Risorse	65
4.1	Introduzione	66
4.1.1	Classi di risorse	66
4.1.2	Risorse non prerilasciabili	66
4.2	Deadlock	67
4.2.1	Condizioni per avere deadlock	67
4.2.2	Grafo di Holt	67
4.3	Metodi di gestione dei deadlock	68
4.3.1	Deadlock detection	68
4.3.2	Deadlock recovery	69
4.3.3	Deadlock prevention e avoidance	70
4.4	Algoritmo del Banchiere	71
4.4.1	Algoritmo del Banchiere Singola valuta	71
4.4.2	Algoritmo del Banchiere Multivaluta	74

5 Memoria	77
5.1 Binding, loading, linking	78
5.1.1 Binding	78
5.1.2 Loading Dinamico	79
5.1.3 Linking statico	79
5.1.4 Linking dinamico	79
5.2 Allocazione	80
5.2.1 Definizioni	80
5.2.2 Allocazione a partizioni fisse	80
5.2.3 Allocazione a partizioni dinamiche	81
5.2.4 Compattazione	81
5.3 Paginazione	84
5.3.1 Introduzione	84
5.3.2 Translation lookaside buffer (TLB)	85
5.4 Segmentazione	86
5.4.1 Introduzione e confronti	86
5.5 Memoria virtuale	88
5.5.1 Introduzione	88
5.5.2 Pager/swapper	88
5.5.3 Gestione dei page fault	89
5.5.4 Algoritmo del meccanismo di demand paging	91
5.5.5 Algoritmi di rimpiazzamento	91
5.5.6 Algoritmo LRU (Least Recently Used)	93
5.5.7 Allocazione	95
6 File System	97
6.1 Introduzione	98
6.1.1 Directory	100
6.1.2 Strutture delle directory	100
6.2 Visione implementazione	101
6.2.1 Organizzazione del disco	101
6.2.2 Allocazione	102
6.2.3 Gestione spazio libero	104

Capitolo 1

Introduzione ai Sistemi Operativi

1.1 Cos'è un sistema operativo?

Definizione: Un sistema operativo è un programma che controlla l'esecuzione di programmi applicativi e agisce come interfaccia tra le applicazioni e l'hardware del calcolatore.

- Efficienza: Un S.O. cerca di utilizzare in modo efficiente le risorse del calcolatore
- Semplicità: Un sistema operativo dovrebbe semplificare l'utilizzazione dell'hardware di un calcolatore

S.O come gestione di risorse: Gestendo le risorse di un calcolatore, un S.O. controlla il funzionamento del calcolatore stesso. Ma questo controllo è esercitato in modo "particolare". Normalmente, il meccanismo di controllo è esterno al sistema controllato.

Esempio: termostato e impianto di riscaldamento

In un elaboratore, il S.O. è un programma, simile all'oggetto del controllo, ovvero le applicazioni controllate. Il S.O. deve lasciare il controllo alle applicazioni e affidarsi al processore per riottenere il controllo.

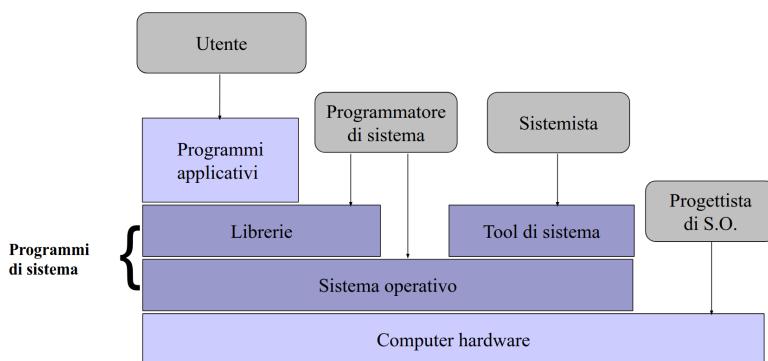


Figura 1.1: Un S.O nasconde ai programmatori i dettagli dell'hardware e fornisce ai programmatori una API conveniente e facile da usare, agisce come intermediario tra programmatore e hardware.

S.O come macchina estesa

```
1 //Esempio senza S.O.  
2 li $t0, 0xDEFF12 // init  
3 sw $t0, 0xB000.0040  
4 li $t0, 0xFFDF // motor  
5 sw $t0, 0xB000.0044  
6 li $t0, 0xFFBB  
7 sw $t0, 0xB000.0048  
8 ...
```

```
1 //Esempio con S.O.  
2 fd = open("/etc/rpc");  
3 read(fd, buffer, size);
```

NB: Questo è esempio serve a dare un'idea, la realtà è molto più complessa ...

Servizi estesi offerti da un S.O.

- esecuzione di programmi
- accesso semplificato ai dispositivi di I/O
- accesso controllato a dispositivi, file system, etc.
- accesso al sistema
- rilevazione e risposta agli errori
- accounting

1.1.1 Tipi di sistemi

Sistemi Paralleli

Definizione: Un sistema parallelo è un singolo elaboratore che possiede più unità di elaborazione. Si dicono anche sistemi tightly coupled. Alcune risorse contenute nell'elaboratore possono essere condivise. *Esempio: Memoria* La comunicazione avviene tramite memoria condivisa o canali di comunicazione dedicati.

Vantaggi dei sistemi paralleli

- incremento delle prestazioni
- incremento dell'affidabilità (graceful degradation)

Tassonomia basata sulla struttura

- **SIMD** - Single Instruction, Multiple Data: Le CPU eseguono all'unisono lo stesso programma su dati diversi
- **MIMD** - Multiple Instruction, Multiple Data: Le CPU eseguono programmi differenti su dati differenti

Tassonomia basata sulla dimensione A seconda del numero (e della potenza) dei processori si suddividono in:

- sistemi a basso parallelismo pochi processori in genere molto potenti
- sistemi massicciamente paralleli gran numero di processori, che possono avere anche potenza non elevata

Sistemi Distribuiti

Definizione: Sono sistemi composti da più elaboratori indipendenti (con proprie risorse e proprio sistema operativo) Si dicono anche sistemi loosely coupled. Ogni processore possiede la propria memoria locale e sono collegati tramite linee di comunicazione (rete, linee telefoniche, linee wireless, etc)

Vantaggi dei sistemi distribuiti

- Condivisione di risorse
- Suddivisione di carico, incremento delle prestazioni
- Affidabilità
- Possibilità di comunicare

I Sistemi operativi di rete forniscono condivisione di file, la possibilità di comunicare e ogni computer opera indipendentemente dagli altri.

I Sistemi operativi distribuiti offrono minore autonomia tra i computer e danno l'impressione che un singolo sistema operativo stia controllando la rete

Sistemi real-time

Definizione: Sono i sistemi per i quali la correttezza del risultato non dipende solamente dal suo valore ma anche dall'istante nel quale il risultato viene prodotto

I sistemi real-time si dividono in:

- hard real-time: se il mancato rispetto dei vincoli temporali può avere effetti catastrofici. *e.g. controllo assetto velivoli, controllo centrali nucleari.*
- soft real-time: se si hanno solamente disagi o disservizi *e.g. programmi interattivi*

1.2 Concetti base di architetture degli elaboratori

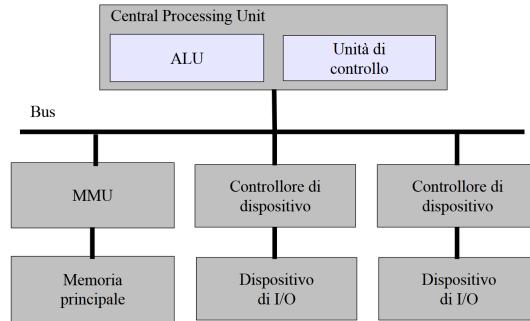


Figura 1.2: Architettura di Von Neumann

1.2.1 Interrupt

Definizione: Un meccanismo che permette l'interruzione del normale ciclo di esecuzione della CPU.

Le caratteristiche: introdotti per aumentare l'efficienza di un sistema di calcolo, permettono ad un S.O. di "intervenire" durante l'esecuzione di un processo utente, allo scopo di gestire efficacemente le risorse del calcolatore. *Esempio:* processore, memoria, dispositivi di I/O

Gli interrupt possono essere sia hardware che software (detti trap) e possono essere mascherati (ritardati) se la CPU sta svolgendo compiti non interrompibili.

Interrupt vs Trap

- Interrupt Hardware: eventi hardware asincroni, non causati dal processo in esecuzione
Esempi: dispositivi di I/O (per notifica di eventi quali il completamento di una operazione di I/O), clock (scadenza del quanto di tempo)
- Interrupt Software (Trap): Causato dal programma
Esempi: eventi eccezionali come divisione per 0 o problemi di indirizzamento, richiesta di servizi di sistema.(system call)

Gestione Interrupt - Panoramica

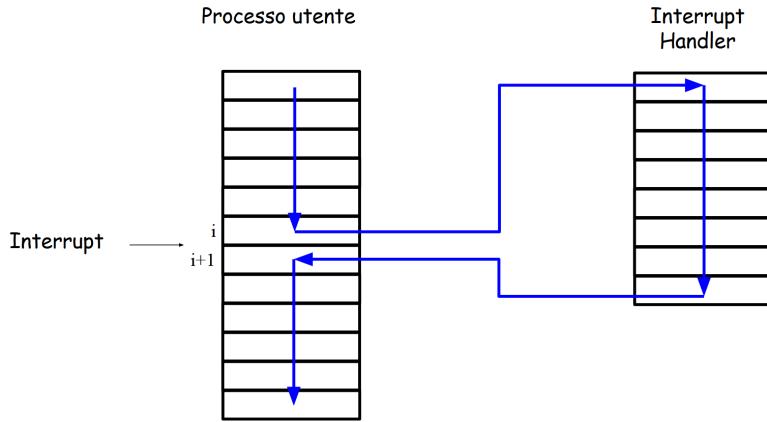
Cosa succede in seguito ad un interrupt.

Dal lato software:

1. Un segnale "interrupt request" viene spedito al processore.
2. Il processore sospende le operazioni del processo corrente e salta ad un particolare indirizzo di memoria contenente la routine di gestione dell'interrupt (interrupt handler).

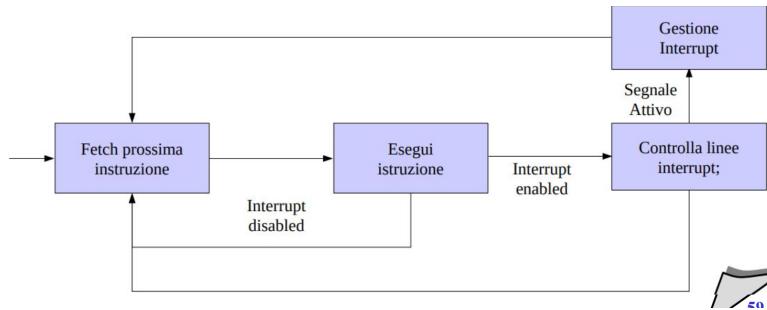
Dal lato hardware

1. L'interrupt handler gestisce nel modo opportuno l'interrupt e ritorna il controllo al processo interrotto (o a un altro processo, nel caso di scheduling).
2. Il processore riprende l'esecuzione del processo interrotto come se nulla fosse successo



Gestione Interrupt - Dettagli

1. Un segnale di interrupt request viene spedito alla CPU.
2. La CPU finisce l'esecuzione dell'istruzione corrente.
3. La CPU verifica la presenza di un segnale di interrupt, e in caso affermativo spedisce un segnale di conferma al device che ha generato l'interrupt.



4. Preparazione al trasferimento di controllo dal programma all'interrupt handler
5. Selezione dell'interrupt handler appropriato a seconda dell'architettura, vi può essere un singolo interrupt handler, uno per ogni tipo di interrupt o uno per dispositivo. La selezione avviene tramite l'interrupt vector.
6. Caricamento del PC con l'indirizzo iniziale dell'interrupt handler assegnato.
Nota: tutte le operazioni compiute fino a qui sono operazioni hardware, la modifica del PC corrisponde ad un salto al codice dell'interrupt handler. A questo punto: il ciclo fetch-execute viene ripreso il controllo è passato in mano all'interrupt handler.
7. Salvataggio dello stato del processore: salvataggio delle informazioni critiche non salvate automaticamente dai meccanismi hardware di gestione interrupt.
8. Gestione dell'interrupt: lettura delle informazioni di controllo proveniente dal dispositivo. Eventualmente, spedizione di ulteriori informazioni al dispositivo stesso.
9. Ripristino dello stato del processore l'operazione inversa della numero 7.
10. Ritorno del controllo al processo in esecuzione (o ad un altro processo, se necessario).

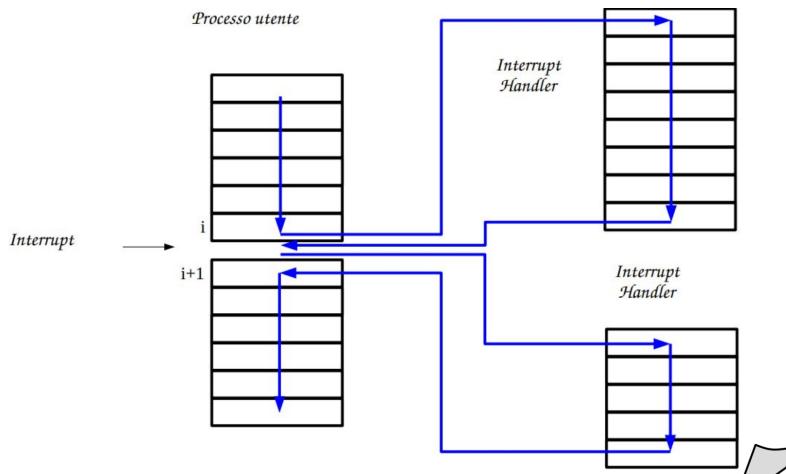
Interrupt Multipli

La discussione precedente prevedeva la presenza di un singolo interrupt. Esiste la possibilità che avvengano interrupt multipli ad esempio, originati da dispositivi diversi, un interrupt può avvenire durante la gestione di un interrupt precedente.

Esistono due approcci possibili: disabilitazione degli interrupt e interrupt annidati.

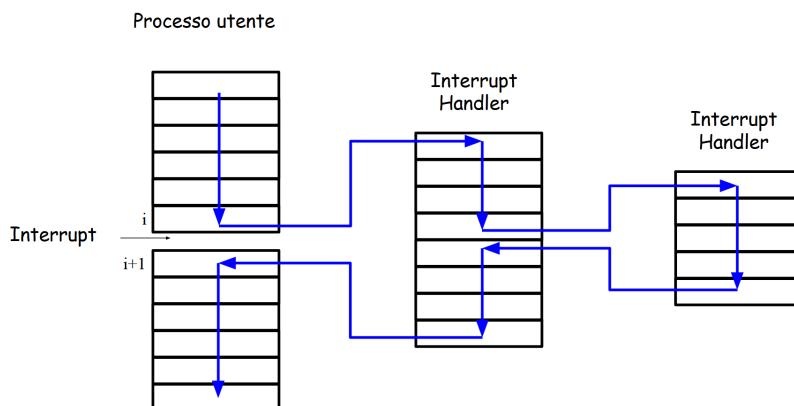
Disabilitazione Interrupt: durante l'esecuzione di un interrupt handler ulteriori segnali di interrupt vengono ignorati, i segnali corrispondenti restano pendenti. Gli interrupt vengono riabilitati prima di riattivare il processo interrotto e il processore verifica quindi se vi sono ulteriori interrupt, in caso attiva l'interrupt handler corrispondente.

Vantaggi e svantaggi:
Approccio semplice, interrupt gestiti in modo sequenziale, non tiene conto di gestioni "time-critical".



Interrupt Annidati: E' possibile definire priorità diverse per gli interrupt. Un interrupt di priorità inferiore può essere interrotto da un interrupt di priorità superiore. Necessario prevedere un meccanismo di salvataggio e ripristino dell'esecuzione adeguato.

Vantaggi e svantaggi:
dispositivi veloci possono essere serviti prima (es. schede di rete), approccio più complesso.



1.2.2 Comunicazione fra processore e dispositivi di I/O

Il controllore governa il dialogo con il dispositivo fisico, il controllo della politica di accesso al dispositivo è a carico del dispositivo stesso

Esempio: il controller di un disco accetta una richiesta per volta, l'accodamento delle richieste in attesa è a carico del S.O.

Esistono 3 modalità di controllo;

1. Programmed I/O
2. Interrupt-Driven I/O
3. Direct Memory Access (DMA)

Programmed I/O (obsoleto)

Operazione di input:

La CPU carica (tramite il bus) i parametri della richiesta di input in appositi registri del controller (registri comando).

Il dispositivo esegue la richiesta, il risultato dell'operazione viene memorizzato in un apposito buffer locale sul controller. Il completamento dell'operazione viene segnalato attraverso appositi registri di status.

Il S.O. attende (busy waiting) che il comando sia completato verificando periodicamente il contenuto del registro di stato.

Infine, la CPU copia i dati dal buffer locale del controller alla memoria.

Interrupt-Driven I/O

Operazione di input:

La CPU carica (tramite il bus) i parametri della richiesta di input in appositi registri del controller (registri comando).

Il S.O. sospende l'esecuzione del processo che ha eseguito l'operazione di input ed esegue un altro processo.

Il dispositivo esegue la richiesta, il risultato dell'operazione viene memorizzato in un apposito buffer locale sul controller, il completamento dell'operazione viene segnalato attraverso interrupt.

Al ricevimento dell'interrupt, la CPU copia i dati dal buffer locale del controller alla memoria.

Programmed I/O vs Interrupt-Driven I/O

Nel caso di operazioni di output il procedimento è similare:

- i dati vengono copiati dalla memoria ai buffer locali.
- questa operazione viene eseguita prima di caricare i parametri della richiesta nei registri di comando dei dispositivi.

Svantaggi dei due approcci:

- il processore spreca parte del suo tempo nella gestione del trasferimento dei dati.
- la velocità di trasferimento è limitata dalla velocità con cui il processore riesce a gestire il servizio.

Direct Memory Access (DMA)

Il S.O. attiva l'operazione di I/O specificando l'indirizzo in memoria di destinazione (Input) o di provenienza (Output) dei dati.

L'interrupt specifica solamente la conclusione dell'operazione di I/O.

Vantaggi e svantaggi:

- c'è contesa nell'accesso al bus device driver più semplici.
- efficace perché la CPU non accede al bus ad ogni ciclo di clock.

1.2.3 Memoria

Memory Mapped I/O

Un dispositivo è completamente indirizzabile tramite bus. I registri di dispositivo vengono mappati su un insieme di indirizzi di memoria, una scrittura su questi indirizzi causa il trasferimento di dati verso il dispositivo. *Esempio: video grafico nei PC.*

Vantaggi e svantaggi:

- gestione molto semplice e lineare.
- necessità di tecniche di polling.

Dischi

Sono dispositivi che consentono la memorizzazione non volatile dei dati permettono accesso diretto. Per individuare un dato sul disco (dal punto di vista fisico) occorre indirizzarlo in termini di cilindro, testina, settore.

Le operazioni gestite dal controller sono: READ (head, sector), WRITE(head, sector), SEEK(cylinder).

L'operazione di seek: Corrisponde allo spostamento fisico del pettine di testine da un cilindro ad un altro ed è normalmente la più costosa

L'operazione di read e write: prevedono l'attesa che il disco ruoti fino a quando il settore richiesto raggiunge la testina.

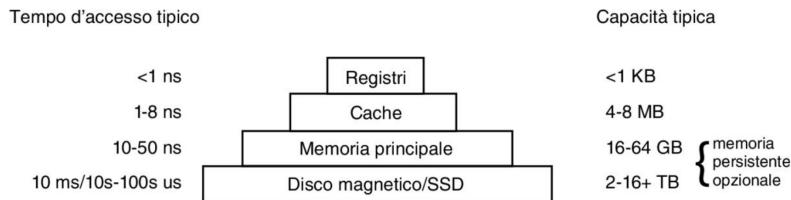
Gerarchia di memoria

Trade off: Quantità, velocità, costo.

Limitazioni:

- tempo di accesso più veloce, costo maggiore.
- maggiore capacità, costo minore (per bit).
- maggiore capacità, tempo di accesso maggiore.

Soluzione: utilizzare una gerarchia di memoria.



Cache

Un meccanismo di caching consiste nel memorizzare parzialmente i dati di una memoria in una seconda più costosa ma più efficiente. Se il numero di occorrenze in cui il dato viene trovato nella cache (memoria veloce) è statisticamente rilevante rispetto al numero totale degli accessi, la cache fornisce un notevole aumento di prestazione.

E' un concetto che si applica a diversi livelli: cache della memoria principale (DRAM), cache di disco in memoria, cache di file system remoti tramite file system locali.

I meccanismi di caching si distinguono in due modi:

- **Hardware** ad esempio cache CPU; politiche non modificabili dal S.O.
- **Software** ad esempio cache disco; politiche sotto controllo del S.O.

I problemi da considerare nel S.O. sono molteplici. Come l' algoritmo di replacement, data la dimensione limitata della cache bisogna scegliere un algoritmo che garantisca il maggior numero di accessi in cache. Oppure la coerenza, gli stessi dati possono apparire a diversi livelli della struttura di memoria.

1.2.4 Protezione Hardware

I sistemi multiprogrammati e multiutente richiedono la presenza di meccanismi di protezione. Bisogna evitare che processi concorrenti generino interferenze non previste, ma soprattutto bisogna evitare che processi utente interferiscano con il sistema operativo.

Quindi i meccanismi di protezione possono essere realizzati totalmente in software, oppure abbiamo bisogno di meccanismi hardware dedicati?

Modalità utente / Modalità kernel

Modalità kernel / supervisore / privilegiata: I processi in questa modalità hanno accesso a tutte le istruzioni, incluse quelle privilegiate, che permettono di gestire totalmente il sistema.

Modalità utente: I processi non hanno accesso alle istruzioni privilegiate. Il mode bit è un bit utilizzato nei processori per distinguere tra due modalità operative: modalità kernel e modalità utente.

Come funziona? Alla partenza, il processore è in modalità kernel. Viene caricato il sistema operativo (bootstrap) e si inizia ad eseguirlo, quando passa il controllo ad un processo utente, il S.O. cambia il valore del mode bit e il processore passa in modalità utente. Tutte le volte che avviene un interrupt, l'hardware passa da modalità utente a modalità kernel.

Protezione I/O

Le istruzioni di I/O devono essere considerate privilegiate, il S.O. dovrà fornire agli utenti primitive e servizi per accedere all'I/O. Tutte le richieste di I/O passano attraverso codice del S.O. e possono essere controllate preventivamente. *Esempio: accesso al dispositivo di memoria secondaria che ospita un file system. Vogliamo evitare che un qualunque processo possa accedere al dispositivo modificando (o corrompendo) il file system stesso.*

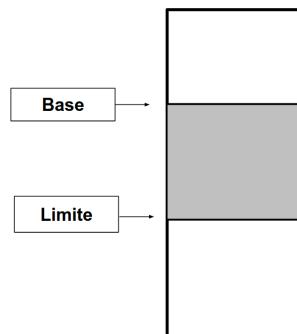
Protezione Memoria

La protezione non è completa se non proteggiamo anche la memoria, altrimenti, i processi utente potrebbero: modificare il codice o i dati di altri processi utenti, modificare il codice o i dati del sistema operativo, modificare l'interrupt vector, inserendo i propri gestori degli interrupt.

La protezione avviene tramite la **Memory Management Unit (MMU)**.

Registro base + registro limite: ogni indirizzo generato dal processore viene confrontato con due registri, detti base e limite. Se non incluso in questo range, l'indirizzo non è valido e genera un'eccezione.

Traduzione indirizzi logici in indirizzi fisici: ogni indirizzo generato dal processore corrisponde ad un indirizzo logico, l'indirizzo logico viene trasformato in un indirizzo fisico a tempo di esecuzione dal meccanismo di MMU. Un indirizzo viene protetto se non può mai essere generato dal meccanismo di traduzione.



1.3 Struttura dei sistemi operativi (panoramica)

1.3.1 Architettura dei sistemi operativi

L'architettura di un sistema operativo descrive quali sono le varie componenti del S.O. e come queste sono collegate fra loro. I vari sistemi operativi sono molto diversi l'uno dall'altro nella loro architettura, la progettazione di essa è un problema fondamentale.

L'architettura di un S.O. da diversi punti di vista: servizi forniti (visione utente), interfaccia di sistema (visione programmatore), componenti del sistema (visione progettista S.O.)

Le componenti di un S.O:

1. Gestione dei processi
2. Gestione della memoria principale
3. Gestione della memoria secondaria
4. Gestione file system
5. Gestione dei dispositivi di I/O
6. Protezione
7. Networking
8. Interprete dei comandi

Gestione dei processi Un processo è un programma in esecuzione che utilizza le risorse fornite dal computer per assolvere i propri compiti.

Il sistema operativo è responsabile delle seguenti attività riguardanti la gestione dei processi: creazione e terminazione dei processi, sospensione e riattivazione dei processi, gestione dei deadlock, comunicazione tra processi e sincronizzazione tra processi.

Gestione della memoria principale La memoria principale è un "array" di byte indirizzabili singolarmente. Consideriamolo un deposito di dati facilmente accessibile e condiviso tra la CPU ed i dispositivi di I/O.

Il sistema operativo è responsabile delle seguenti attività riguardanti la gestione della memoria principale: tenere traccia di quali parti della memoria sono usate e da chi, decidere quali processi caricare quando diventa disponibile spazio in memoria e allocare e deallocare lo spazio di memoria quando necessario.

Gestione della memoria secondaria Poiché la memoria principale è volatile e troppo piccola per contenere tutti i dati e tutti i programmi in modo permanente, un computer è dotato di memoria secondaria. In generale, la memoria secondaria è data da hard disk, dischi ottici, nastri, etc.

Il sistema operativo è responsabile delle seguenti attività riguardanti la gestione della memoria secondaria: Allocazione dello spazio inutilizzato, Gestione dello spazio di memorizzazione e ordinamento efficiente delle richieste (disk scheduling).

Gestione file system Un file è l'astrazione informatica di un archivio di dati. Il concetto di file è indipendente dal media sul quale viene memorizzato (che ha caratteristiche proprie e propria organizzazione fisica). Un file system è composto da un insieme di file.

Il sistema operativo è responsabile delle seguenti attività riguardanti la gestione del file system: creazione e cancellazione di file, creazione e cancellazione di directory, manipolazione di file e directory e codifica del file system sulla memoria secondaria.

Gestione dei dispositivi di I/O La gestione dell'I/O richiede: un'interfaccia comune per la gestione dei device driver, un insieme di driver per dispositivi hardware specifici e un sistema di gestione di buffer per il caching delle informazioni.

Protezione Il termine protezione si riferisce al meccanismo per controllare gli accessi di programmi, processi o utenti alle risorse del sistema e degli utenti.

Il meccanismo di protezione software deve: distinguere tra uso autorizzato o non autorizzato, specificare i controlli che devono essere imposti e fornire un meccanismo di attuazione della protezione.

Networking Consente di far comunicare due o più elaboratori, condividere risorse, utilizzando servizi come: protocolli di comunicazione a basso livello, TCP/IP, UDP, comunicazione ad alto livello, file system distribuiti (NFS, SMB) e print spooler.

Interprete dei comandi Interfaccia utente - S.O. . Può attivare un programma, terminare un programma, interagire con le componenti del sistema operativo (file system). Può essere: grafica (a finestre, icone, etc.), testuale (linea di comando). Può cambiare il "linguaggio" utilizzato, ma il concetto è lo stesso, d'altronde vi sono differenze di espressività

N.B L'interprete dei comandi usa i servizi dei gestori di processi, I/O, memoria principale e secondaria

1.3.2 System Call

Cos'è la system call

Problema: poiché le istruzioni di I/O sono privilegiate, possono essere eseguite unicamente dal S.O. . Com'è possibile per i processi utenti eseguire operazioni di I/O?

Soluzione: i processi utenti devono fare richieste esplicite di I/O al S.O. . Questo introduce il meccanismo delle system call, ovvero trap generate da istruzioni specifiche.

Ogni volta che un processo ha bisogno di un servizio del S.O. richiama una system call. Sono in genere disponibili come istruzioni a livello assembler, esistono librerie che permettono di invocare le system call da diversi linguaggi (*ad es. librerie C*). Vengono normalmente realizzate tramite interrupt software. Le system call sono specifiche dei vari sistemi operativi.

Esempi di istruzioni privilegiate eseguibili in Kernel Mode:

- Gestione della memoria
- Gestione della memoria virtuale (allocazione, etc.)
- Controllo dell'hardware (I/O, controller di interrupt)
- Gestione della CPU
- Istruzioni di halt e riavvio
- Gestione del clock di sistema
- Operazioni sulla cache
- Gestione dell'alimentazione
- Operazioni crittografiche hardware
- Gestione delle eccezioni e degli interrupt
- Accesso a registri di controllo speciali
- ...

Gestione dei file

```
1 pid = fork() //crea un processo figlio identico al padre  
1 pid = waitpid(pid, &statloc, options) //aspetta la terminazione di un processo figlio  
1 s = execve(name, argv, environment) //esegue un programma  
1 exit(status) //termina l'esecuzione del processo corrente
```

```

1 //Un programma che genera un processo figlio:
2 int main(void) {
3     int pid;
4     pid = fork();
5     if (pid > 0) {
6         printf('Padre\n');
7     } else if (pid == 0) {
8         printf('Figlio\n');
9     } else {
10        printf('Errore!\n');
11    }
12 }
```

Gestione del file system e delle directory

```

1 s = mkdir(name, mode) //Crea una nuova directory
1 s = rmdir(name) //Cancella una directory
1 s = link(name1, name2) //Crea un nuovo link ad un file esistente
1 s = unlink(name) //Cancella un file
1 s = mount(special, name, flag) //Monta una partizione nel file system
1 s = umount(special) //Smonta una partizione
```

Varie

```

1 s = chdir(dirname) //Cambia la directory corrente
1 s = chmod(name, mode) //Cambi i bit di protezione di un file
1 s = kill(pid, signal) //Spedisce un segnale ad un processo
1 seconds = time(&seconds) //Restituisce il tempo di sistema
```

1.3.3 File Speciali (UNIX)

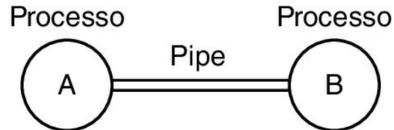
I file speciali sono pensati per far sì che i dispositivi di I/O siano visti come file.

I file speciali a blocchi: usati per modellare dispositivi costituiti da un insieme di blocchi indirizzabili in modo casuale (SSD, dischi). Aprendo un file speciale a blocchi e leggendo il blocco 4, un programma può accedere direttamente al quarto blocco, senza considerare la struttura del file system

I file speciali a caratteri: usati per modellare stampanti, tastiere, mouse, etc. che accettano una sequenza di caratteri in output, contenuti nella directory /dev.

Pipe

Una pipe è una specie di pseudo-file che può essere usato per connettere due processi.



Pipe in Bash: Le abbiamo viste con Ghini. *Esempio: A || B*

Pipe in C: Utilizza la funzione `pipe()` e i descrittori di file, permette la comunicazione tra processi correlati (es. genitore-figlio). Sono unidirezionali (ma possono essere create multiple pipe per bidirezionalità) La creazione è esplicita nel codice del programma e persiste finché non viene chiusa o il processo termina. *Esempio: pipe(pipefd) ... write(pipefd[1], ...) ... read(pipefd[0], ...)*

Capitolo 2

Scheduling

2.1 Scheduler, Processi e Thread

2.1.1 Introduzione

Un sistema operativo è un gestore di risorse come: processore, memoria principale e secondaria, dispositivi. Per svolgere i suoi compiti, un sistema operativo ha bisogno di strutture dati per mantenere informazioni sulle risorse gestite. Queste strutture dati comprendono: tabelle di memoria, tabelle di I/O, tabelle del file system e tabelle dei processi.

Tabelle per la gestione della memoria

Allocazione memoria per il sistema operativo, allocazione memoria principale e secondaria per i processi, informazioni per i meccanismi di protezione.

Tabelle per la gestione dell'I/O

Informazioni sullo stato di assegnazione dei dispositivi utilizzati dalla macchina, gestione di code di richieste.

Tabelle per la gestione del file system

Elenco dei dispositivi utilizzati per mantenere il file system, elenco dei file aperti e loro stato.

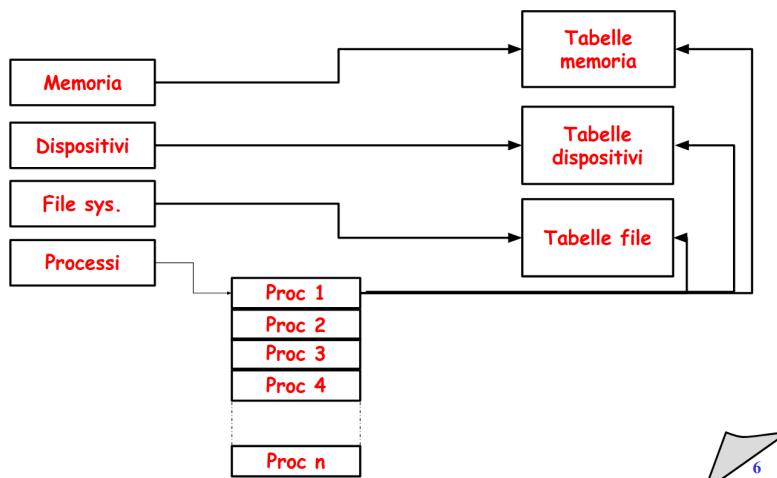


Figura 2.1: Tabelle per la gestione

2.1.2 Descrittori dei processi

Qual é la manifestazione fisica di un processo?

1. il codice da eseguire (segmento codice)
 2. i dati su cui operare (segmenti dati)
 3. uno stack di lavoro per la gestione di chiamate di funzione, passaggio di parametri e variabili locali
 4. un insieme di attributi contenenti tutte le informazioni necessarie per la gestione del processo stesso
- Questo insieme di attributi prende il nome di descrittore del processo (process control block, PCB).

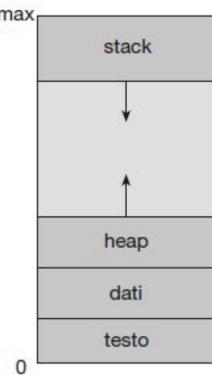


Figura 2.2: Processo in memoria

Tabella per la gestione dei processi: contiene i descrittori dei processi, ogni processo ha un descrittore associato. È possibile suddividere le informazioni contenute nel descrittore in tre aree:

- informazioni di identificazione di processo
- informazioni di stato del processo
- informazioni di controllo del processo

Informazioni di identificazione di un processo

L'identificatore di processo (process id, o pid), può essere semplicemente un indice all'interno di una tabella di processi. Può essere un numero progressivo; in caso, è necessario un mapping tra pid e posizione del relativo descrittore, molte altre tabelle del s.o. utilizzano il process id per identificare un elemento della tabella dei processi.

Possono essere identificatori di altri processi logicamente collegati al processo. *Ad esempio, pid del processo padre*

Id dell'utente che ha richiesto l'esecuzione del processo

Informazioni di stato del processo

Come registri generali del processore o registri speciali, come il program counter e i registri di stato.

Informazioni di controllo del processo

Informazioni di scheduling: stato del processo (in esecuzione, pronto, in attesa), informazioni particolari necessarie dal particolare algoritmo di scheduling utilizzato, identificatore dell'evento per cui il processo è in attesa.

Informazioni di gestione della memoria: valori dei registri base e limite dei segmenti utilizzati, puntatori alle tabelle delle pagine, etc.

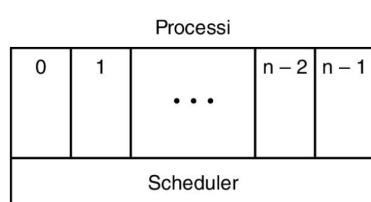
Informazioni di accounting: tempo di esecuzione del processo, tempo trascorso dall'attivazione di un processo.

Informazioni relative alle risorse: risorse controllate dal processo, come file aperti, device allocati al processo.

Informazioni per interprocess communication (IPC): stato di segnali, semafori, etc.

2.1.3 Scheduler

Lo scheduler è la componente più importante del kernel, gestisce l'avvicendamento dei processi decidendo quale processo deve essere in esecuzione ad ogni istante, interviene quando viene richiesta un'operazione di I/O e quando un'operazione di I/O termina, ma anche periodicamente.



Schedule: è la sequenza temporale di assegnazioni delle risorse da gestire ai richiedenti.

Scheduling: è l'azione di calcolare uno schedule.
Scheduler: è la componente software che calcola lo schedule.

2.1.4 Mode switching e Context switching

Tutte le volte che avviene un interrupt (software o hardware) il processore è soggetto ad un **mode switching** (modalità utente → modalità supervisore). Durante la gestione dell'interrupt vengono intraprese le opportune azioni per gestire l'evento. Viene chiamato lo scheduler: se lo scheduler decide di eseguire un altro processo, il sistema è soggetto ad un **context switching**.

Durante un context switching, lo stato del processo attuale viene salvato nel PCB corrispondente mentre lo stato del processo selezionato per l'esecuzione viene caricato dal PCB nel processore.

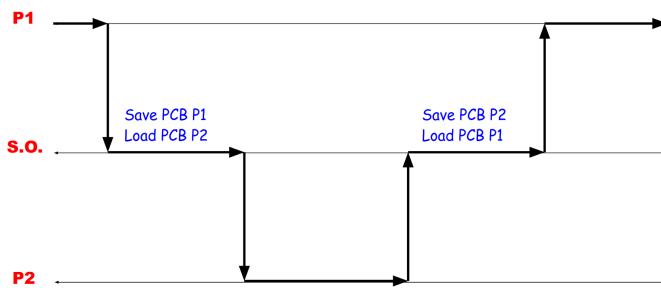


Figura 2.3: Funzionamento del context switch

2.1.5 Processi e Thread

Vita di un processo

Gli stati di un processo possono essere:

Running: il processo è in esecuzione.

Waiting: il processo è in attesa di qualche evento esterno (e.g., completamento, operazione di I/O); non può essere eseguito.

Ready: il processo può essere eseguito, ma attualmente il processore è impegnato in altre attività.

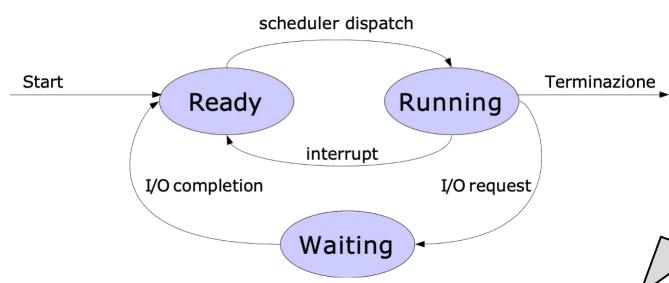


Figura 2.4: Stati dei processi

Tutte le volte che un processo entra nel sistema, viene posto in una delle code gestite dallo scheduler.

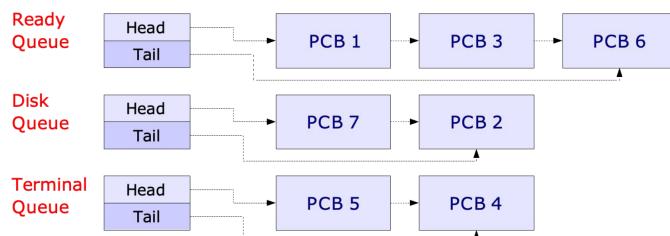


Figura 2.5: Le code gestite dallo scheduler

Gerarchia di processi

Nella maggior parte dei sistemi operativi i processi sono organizzati in forma gerarchica.

Quando un processo crea un nuovo processo, il processo creante viene detto padre e il creato figlio si viene così a creare un albero di processi.

Le motivazioni sono la semplificazione del procedimento di creazione di processi, non occorre specificare esplicitamente tutti i parametri e le caratteristiche e ciò che non viene specificato, viene ereditato dal padre.

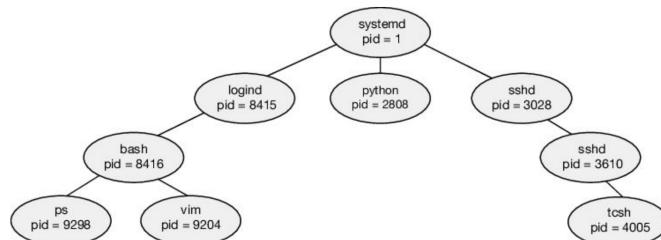


Figura 2.6: Albero dei processi di un sistema Linux

Threads

La nozione di processo discussa in precedenza assume che ogni processo abbia una singola “linea di controllo”. Per ogni processo, viene eseguita una singola sequenza di istruzioni, un singolo processo non può eseguire due differenti attività contemporaneamente.

Esempi: scaricamento di due differenti pagine in un web browser, inserimento di nuovo testo in un wordprocessor mentre viene eseguito il correttore ortografico

Tutti i sistemi operativi moderni supportano l'esistenza di processi **multithreaded**. In un processo multithreaded esistono molte “linee di controllo”, ognuna delle quali può eseguire un diverso insieme di istruzioni.

Esempi: Associando un thread ad ogni finestra aperta in un web browser, è possibile scaricare i dati in modo indipendente.

Cos'è un thread? Un thread è l'unità base di utilizzazione della CPU. Ogni thread possiede la propria copia dello stato del processore, il proprio program counter e uno stack separato. I thread appartenenti allo stesso processo condividono: codice, dati e risorse di I/O.

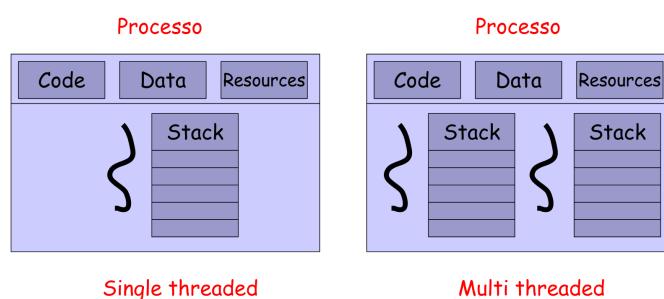


Figura 2.7: Singlethreaded e multithreaded

Benefici dei thread La condivisione di risorse: i thread condividono lo spazio di memoria e le risorse allocate degli altri thread dello stesso processo. Condividere informazioni tra thread logicamente correlati rende più semplice l'implementazione di certe applicazioni.

Esempio: web browser: condivisione dei parametri di configurazione fra i vari thread.

Economia: allocare memoria e risorse per creare nuovi processi è costoso, fare context switching fra diversi processi non è ottimale. Mentre gestire i thread è in generale più economico, creare thread all'interno di un processo permette di fare context switching fra thread che è meno costoso.

Esempio: creare un thread in Solaris richiede 1/30 del tempo richiesto per creare un nuovo processo.

Thread = processi “lightweight” Utilizzare i thread al posto dei processi rende l'implementazione più efficiente. In ogni caso, abbiamo bisogno di processi distinti per applicazioni differenti.

2.1.6 Multithreading

Un sistema operativo può implementare i thread in due modi:

- User thread
- Kernel thread

User thread

Gli user thread vengono supportati sopra il kernel e vengono implementati da una **thread library** a livello utente. La thread library fornisce supporto per la creazione, lo scheduling e la gestione dei thread senza alcun intervento del kernel.

Vantaggi: l'implementazione risultante è molto efficiente.

Svantaggi: Se il kernel è single-threaded, qualsiasi user thread che effettua una chiamata di sistema bloccante (che si pone in attesa di I/O) causa il blocco dell'intero processo.

Kernel thread

I kernel thread vengono supportati direttamente dal sistema operativo. La creazione, lo scheduling e la gestione dei thread sono implementati a livello kernel.

Vantaggi: poiché è il kernel a gestire lo scheduling dei thread, se un thread esegue una operazione di I/O, il kernel può selezionare un altro thread in attesa di essere eseguito.

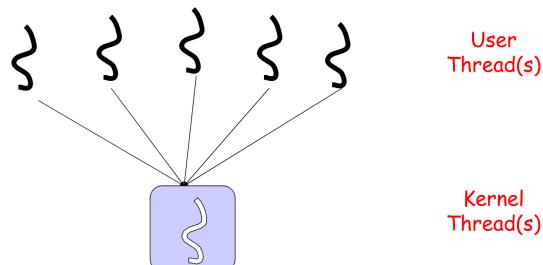
Svantaggi: l'implementazione risultante è più lenta, perché richiede un passaggio da livello utente a livello supervisore.

Modelli di multithreading

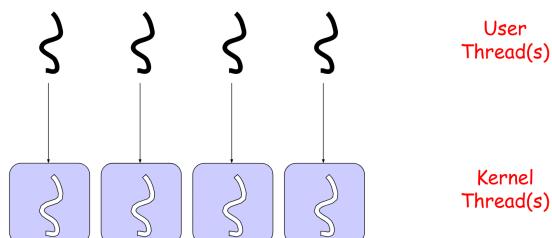
Molti sistemi supportano sia kernel thread che user thread. Si vengono così a creare tre differenti modelli di multithreading:

- Many-to-One
- One-to-One
- Many-to-Many

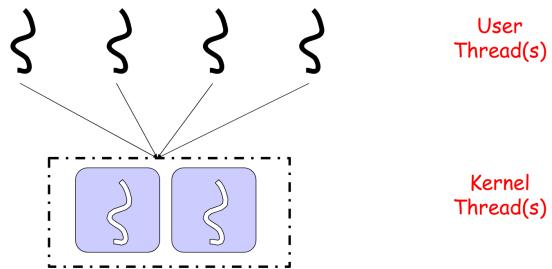
Many-to-One Un certo numero di user thread vengono mappati su un solo kernel thread. Modello generalmente adottato da s.o. che non supportano kernel thread multipli.



One-to-One Ogni user thread viene mappato su un kernel thread. Può creare problemi di scalabilità per il kernel.



Many-to-Many Riassume i benefici di entrambe le architetture. Supportato da Solaris, IRIX, Digital Unix.



2.2 Schedule

2.2.1 Rappresentazione degli schedule

Per rappresentare uno schedule si usano i diagrammi di Gantt

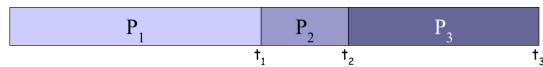


Figura 2.8: In questo esempio, la risorsa (es. CPU) viene utilizzata dal processo P_1 dal tempo 0 a t_1 , viene quindi assegnata a P_2 fino al tempo t_2 e quindi a P_3 fino al tempo t_3 .

Nel caso si debba rappresentare lo schedule di più risorse (*e.g., un sistema multiprocessore*) il diagramma di Gantt risulta composto da più righe parallele.

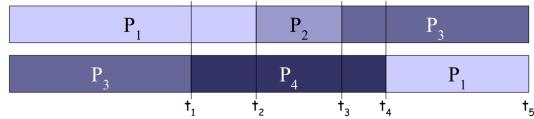


Figura 2.9: Diagramma di Gantt multi-risorsa

Tipi di scheduler

Eventi che possono causare un context switch:

1. quando un processo passa da stato running a stato waiting(system call bloccante, operazione di I/O).
2. quando un processo passa dallo stato running allo stato ready (a causa di un interrupt).
3. quando un processo passa dallo stato waiting allo stato ready.
4. quando un processo termina.

(Nota: nelle condizioni 1 e 4, l'unica scelta possibile è quella di selezionare un altro processo per l'esecuzione. Nelle condizioni 2 e 3, è possibile continuare ad eseguire il processo corrente)

Uno scheduler si dice **non-preemptive** (o cooperativo) se i context switch avvengono solo nelle condizioni 1 e 4. In altre parole: il controllo della risorsa viene trasferito solo se l'assegnatario attuale lo cede volontariamente. Vantaggi: non richiede alcuni meccanismi hardware come ad esempio timer programmabili.

Uno scheduler si dice **preemptive** (tutti gli scheduler moderni) se i context switch possono avvenire in ogni condizione. In altre parole: è possibile che il controllo della risorsa venga tolto all'assegnatario attuale a causa di un evento. Vantaggi: permette di utilizzare al meglio le risorse.

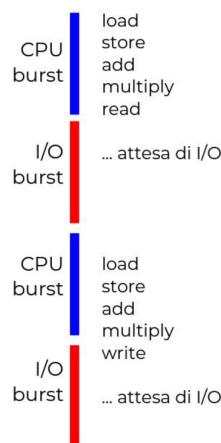
Criteri di scelta di uno scheduler

- **Utilizzo della risorsa (CPU):** lq percentuale di tempo in cui la CPU è occupata ad eseguire processi deve essere massimizzato.
- **Throughput:** numero di processi completati per unità di tempo. Dipende dalla lunghezza dei processi, deve essere massimizzato.
- **Tempo di turnaround:** tempo che intercorre dalla sottomissione di un processo alla sua terminazione, deve essere minimizzato.
- **Tempo di attesa:** il tempo trascorso da un processo nella coda ready, deve essere minimizzato.
- **Tempo di risposta:** il tempo che intercorre fra la sottomissione di un processo e il tempo di prima risposta. Particolarmente significativo nei programmi interattivi, deve essere minimizzato.

Caratteristiche dei processi

Durante l'esecuzione di un processo si alternano periodi di attività svolte dalla CPU (**CPU burst**) e periodi di attività di I/O (**I/O burst**).

I processi caratterizzati da CPU burst molto lunghi si dicono **CPU bound**, mentre quelli caratterizzati da I/O burst molto lunghi si dicono **I/O bound**.



2.3 Algoritmi di scheduling

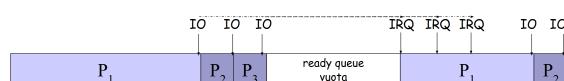
Gli algoritmi di scheduling che vedremo sono:

- First Come, First Served
- Shortest-Job First
 - Shortest-Next-CPU-Burst First
 - Shortest-Remaining-Time-First
- Round-Robin
- Scheduling a priorità
- Scheduling a classi di priorità
- Scheduling multilivello

2.3.1 First Come, First Served (FCFS)

Algoritmo: il processo che arriva per primo, viene servito per primo, politica senza preemption.

Supponiamo di avere: un processo CPU bound, un certo numero di processi I/O bound. I processi I/O bound si "mettono in coda" dietro al processo CPU bound, e in alcuni casi la ready queue si può svuotare (convoy effect).



Implementazione: semplice, tramite una coda (politica FIFO)

Problemi: elevati tempi medi di attesa e di turnaround, processi CPU bound ritardano i processi I/O bound.

Esempio:



Figura 2.10: Ordine di arrivo: P_1, P_2, P_3

Lunghezza dei CPU-burst in ms: 32, 2, 2

Tempo medio di turnaround: $(32+34+36)/3 = 34$ ms

Tempo medio di attesa: $(0+32+34)/3 = 22$ ms

2.3.2 Shortest Job First (SJF)

Algoritmo: la CPU viene assegnata al processo ready che ha la minima durata del CPU burst successivo, politica senza preemption.

Problemi: è ottimale rispetto al tempo di attesa, in quanto è possibile dimostrare che produce il minor tempo di attesa possibile, ma è impossibile da implementare in pratica! E' possibile solo fornire delle approssimazioni.

Negli scheduler long-term possiamo chiedere a chi sottomette un job di predire la durata del job. Negli scheduler short-term non possiamo conoscere la lunghezza del prossimo CPU burst ma conosciamo la lunghezza di quelli precedenti.

Esempio:



Figura 2.11: Tempo medio di turnaround: $(0+2+4+36)/3 = 7$ ms

Tempo medio di attesa: $(0+2+4)/3 = 2$ ms

Calcolo approssimato della durata del CPU burst basata su media esponenziale dei CPU burst precedenti sia t_n il tempo dell'n-esimo CPU burst e T_n la corrispondente previsione; T_{n+1} può essere calcolato come segue: $T_{n+1} = \alpha t_n + (1 - \alpha)T_n$

Media esponenziale svolgendo la formula di ricorrenza, si ottiene:

$$T_{n+1} = \sum_{j=0}^n \alpha(1 - \alpha)^j t_{n-j} + (1 - \alpha)^{n+1} T_0$$

Spiegazione:

- t_n rappresenta la storia recente.
- T_n rappresenta la storia passata.
- α rappresenta il peso relativo di storia passata e recente.

(Nota: SJF può essere soggetto a starvation)

Shortest Job First "approssimato" esiste in due versioni:

Non preemptive: Shortest-Next-CPU-Burst First Il processo corrente esegue fino al completamento del suo CPU burst.

Preemptive: Shortest-Remaining-Time First Il processo corrente può essere messo nella coda ready, se arriva un processo con un CPU burst più breve di quanto rimane da eseguire al processo corrente.

2.3.3 Round-Robin

E' basato sul concetto di **quanto di tempo** (o time slice). Un processo non può rimanere in esecuzione per un tempo superiore alla durata del quanto di tempo.

La durata del quanto di tempo è un parametro critico del sistema, se il quanto di tempo è breve, il sistema è meno efficiente perché deve cambiare il processo attivo più spesso.

Se il quanto è lungo, in presenza di numerosi processi pronti ci sono lunghi periodi di inattività di ogni singolo processo. (In sistemi interattivi, questo può essere fastidioso per gli utenti).

Implementazione: l'insieme dei processi pronti è organizzato come una coda. Due possibilità:

- un processo può lasciare il processore volontariamente, in seguito ad un'operazione di I/O.
- un processo può esaurire il suo quanto di tempo senza completare il suo CPU burst, nel qual caso viene aggiunto in fondo alla coda dei processi pronti.

In entrambi i casi, il prossimo processo da eseguire è il primo della coda dei processi pronti.

Nell'implementazione è necessario che l'hardware fornisca un timer (**interval timer**) che agisca come "sveglia" del processore.

Il timer è un dispositivo che, attivato con un preciso valore di tempo, è in grado di fornire un interrupt allo scadere del tempo prefissato, viene interfacciato come se fosse un'unità di I/O.

Esempio:

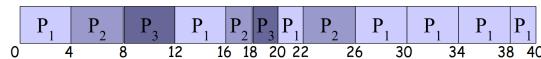


Figura 2.12

Tre processi: P_1, P_2, P_3

Lunghezza dei CPU-burst in ms ($P_1: 10+14; P_2: 6+4; P_3: 6$)

Lunghezza del quanto di tempo: 4

Tempo medio di turnaround $(40+26+20)/3 = 28.66$ ms

Tempo medio di attesa: $(16+16+14)/3 = 15.33$ ms

Tempo medio di risposta: 4 ms

2.3.4 Scheduling a priorità

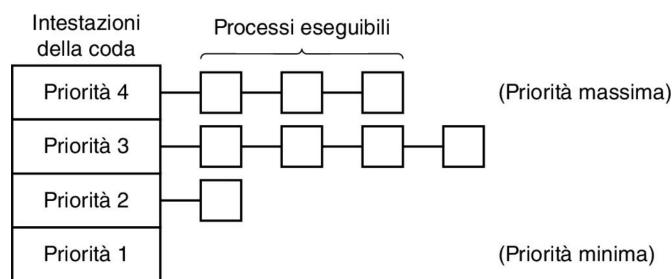
Il round-robin fornisce le stesse possibilità di esecuzione a tutti i processi. Ma i processi non sono tutti uguali, infatti usando round-robin puro la visualizzazione di un video MPEG potrebbe essere ritardata da un processo che sta per esempio smistando la posta, la lettera può aspettare mezzo secondo, il frame video no!

Vediamo quindi come impostare un round-robin con delle priorità...

Descrizione: a ogni processo è associata una specifica priorità, lo scheduler sceglie il processo pronto con priorità più alta.

Le priorità possono essere:

- **Definite dal sistema operativo:** vengono utilizzate una o più variabili per calcolare la priorità di un processo. *Esempio: SJF è un sistema basato su priorità*
- **Definite esternamente:** le priorità non vengono definite dal sistema operativo, ma vengono imposte dal livello utente.



Tecniche di assegnazione delle priorità

Priorità statica: la priorità non cambia durante la vita di un processo. Un problema di questa tecnica è che i processi a bassa priorità possono essere posti in starvation da processi ad alta priorità.

Priorità dinamica: la priorità può variare durante la vita di un processo, è possibile utilizzare metodologie di priorità dinamica per evitare starvation

Priorità basata su aging: tecnica che consiste nell'incrementare gradualmente la priorità dei processi in attesa, posto che il range di variazione delle priorità sia limitato, nessun processo rimarrà in attesa per un tempo indefinito perché prima o poi raggiungerà la priorità massima.

2.3.5 Scheduling a classi di priorità

Descrizione: è possibile creare diverse classi di processi con caratteristiche simili e assegnare ad ogni classe specifiche priorità. La coda ready viene quindi scomposta in molteplici "sottocode", una per ogni classe di processi.

Algoritmo: uno scheduler a classi di priorità seleziona il processo da eseguire fra quelli pronti della classe a priorità massima che contiene processi.

2.3.6 Scheduling multilivello

Descrizione: all'interno di ogni classe di processi, è possibile utilizzare una politica specifica adatta alle caratteristiche della classe.

Uno scheduler multilivello cerca prima la classe di priorità massima che ha almeno un processo ready, sceglie poi il processo da porre in stato running coerentemente con la politica specifica della classe.

Quattro classi di processi (priorità decrescente):

- processi server (*priorità statica*)
- processi utente interattivi (*round-robin*)
- altri processi utente (*FIFO*)
- processo vuoto (*FIFO banale*)

2.4 Scheduling Real-Time

In un sistema real-time la correttezza dell'esecuzione non dipende solamente dal valore del risultato, ma anche dall'istante temporale nel quale il risultato viene emesso.

Hard real-time: le deadline di esecuzione dei programmi non devono essere superate in nessun caso.
Esempi: sistemi di controllo nei velivoli, centrali nucleari o per la cura intensiva dei malati...

Soft real-time: errori occasionali sono tollerabili.
Esempi: ricostruzione di segnali audio-video, transazioni interattive

Esistono due tipi di processi RT (real-time).

Processi periodici: sono periodici i processi che vengono riattivati con una cadenza regolare (periodo).
Esempio: controllo assetto dei velivoli, basato su rilevazione periodica dei parametri di volo.

Processi aperiodici: i processi che vengono scatenati da un evento sporadico.
Esempio: l'allarme di un rilevatore di pericolo

2.4.1 Esempi di Scheduler RT

Rate Monotonic

Descrizione: è una politica di scheduling, dove ogni processo periodico deve completarsi entro il suo periodo, tutti i processi sono indipendenti e la preemption avviene istantaneamente e senza overhead.

Viene assegnata staticamente una priorità a ogni processo, processi con frequenza più alta (*i.e. periodo più corto*) hanno priorità più alta. Ad ogni istante, viene eseguito il processo con priorità più alta (facendo preemption se necessario).

Earliest Deadline First (EDF)

Descrizione: è una politica di scheduling per processi periodici real-time, viene scelto di volta in volta il processo che ha la deadline più prossima.

Viene detto "a priorità dinamica" perchè la priorità relativa di due processi varia in momenti diversi.

Capitolo 3

Concorrenza

3.1 Introduzione alla concorrenza

3.1.1 Modello concorrente

Un sistema operativo consiste in un gran numero di attività che vengono eseguite più o meno contemporaneamente dal processore e dai dispositivi presenti in un elaboratore.

Senza un modello adeguato, la coesistenza delle diverse attività sarebbe difficile da descrivere e realizzare.

Il modello che è stato realizzato a questo scopo prende il nome di modello concorrente ed è basato sul concetto astratto di processo.

In questa serie di lucidi: analizzeremo il problema della gestione di attività multiple da un punto di vista astratto.

Il modello concorrente rappresenta una rappresentazione astratta di un S.O. multiprogrammato.

Negli altri moduli del corso: vedremo i dettagli necessari per la gestione di processi in un S.O. reale.

In particolare, il problema dello scheduling, ovvero come un S.O. seleziona le attività che devono essere eseguite dal processore.

3.1.2 Processi

Definizione: un'attività controllata da un programma che si svolge su un processore. Un processo non è un programma!

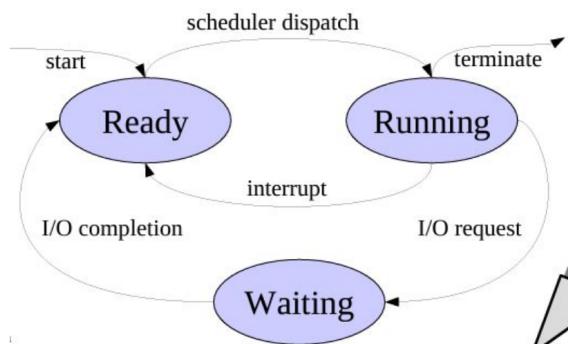
Un programma è un'entità statica, un processo è dinamico.

Un programma specifica un'insieme di istruzioni e la loro sequenza di esecuzione ma non specifica la distribuzione nel tempo dell'esecuzione.

Un processo rappresenta il modo in cui un programma viene eseguito nel tempo. Assioma di finite progress. Ogni processo viene eseguito ad una velocità finita ma sconosciuta.

Stato di un processo

- Running: il processo è in esecuzione.
- Waiting: il processo è in attesa di qualche evento esterno (*ad es. completamento operazione di I/O*) non può essere eseguito.
- Ready: il processo può essere eseguito, ma attualmente il processore è impegnato in altre attività.



3.1.3 Multiprogramming e multiprocessing

Cos'è la concorrenza?

Tema centrale nella progettazione dei S.O. riguarda la gestione di processi multipli.

Due programmi si dicono in esecuzione concorrente se vengono eseguiti in parallelo (con parallelismo reale o apparente). La concorrenza è l'insieme di notazioni per descrivere l'esecuzione concorrente di due o più programmi, sono tecniche per risolvere i problemi associati all'esecuzione concorrente, quali comunicazione e sincronizzazione.

La multiprogrammazione è stata inventata affinché più processi indipendenti condividano il processore. Alcune applicazioni possono essere progettate come un insieme di processi o thread concorrenti.

Inoltre, molte funzioni del sistema operativo possono essere implementate come un insieme di processi o thread.

- **Multiprogramming:** più processi su un solo processore, parallelismo apparente.
- **Multiprocessing:** più processi su una macchina con processori multipli, parallelismo reale.
- **Distributed processing:** più processi su un insieme di computer distribuiti e indipendenti, parallelismo reale.

Differenze tra multiprocessing e multiprogramming

In un singolo processore: processi multipli sono "alternati nel tempo" per dare l'impressione di avere un multiprocessore. Ad ogni istante, al massimo un processo è in esecuzione, si parla di **interleaving**.

In un sistema multiprocessore: più processi vengono eseguiti simultaneamente su processori diversi, i processi sono "alternati nello spazio e si parla di **overlapping**.

A prima vista si potrebbe pensare che queste differenze comportino problemi distinti. In un caso l'esecuzione è simultanea, nell'altro caso la simultaneità è solo simulata.

In realtà presentano gli stessi problemi, ovvero, non è possibile predire la velocità relativa dei processi.

Alcune considerazioni: non vi è sostanziale differenza tra i problemi relativi a multiprogramming e multiprocessing, ai fini del ragionamento sui programmi concorrenti si ipotizza che sia presente un "processore ideale" per ogni processo.

I problemi derivano dal fatto che non è possibile predire gli istanti temporali in cui vengono eseguite le istruzioni i due processi quando accedono ad una o più risorse condivise.

Race condition

Definizione: si dice che un sistema di processi multipli presenta una race condition qualora il risultato finale dell'esecuzione dipenda dalla temporizzazione con cui vengono eseguiti i processi.

Per scrivere un programma concorrente è necessario eliminare le race condition.
In pratica scrivere programmi concorrenti è più difficile che scrivere programmi sequenziali. La correttezza non è solamente determinata dall'esattezza dei passi svolti da ogni singola componente del programma, ma anche dalle interazioni (volute o no) tra essi. Fare debug di applicazioni che presentano race condition non è per niente piacevole...

Notazione per descrivere processi concorrenti

```

1 process nome {
2 ... statement(s) ...
3 }
```

Esempio

```

1 process P1 {
2     totale = totale + valore;
3 }
4
5 process P2 {
6     totale = totale - valore;
7 }
```

3.2 Interazioni tra processi

E' possibile classificare le modalità di interazione tra processi in base a quanto sono "consapevoli" l'uno dell'altro.

3.2.1 Tipi di interazione

Processi ignari

Processi totalmente "ignari" l'uno dell'altro non sono progettati per lavorare insieme, sebbene siano indipendenti, vivono in un ambiente comune.

Come interagiscono? essi competono per le stesse risorse e devono sincronizzarsi nella loro utilizzazione. Il sistema operativo ha il compito di arbitrare questa competizione, fornendo meccanismi di sincronizzazione.

Processi indiretti

Mentre processi "indirettamente" a conoscenza uno dell'altro sono processi che condividono risorse, come ad esempio un buffer, al fine di scambiarsi informazioni, non si conoscono in base ai loro id, ma interagiscono indirettamente tramite le risorse condivise.

Come interagiscono? cooperano per qualche scopo e devono sincronizzarsi nella utilizzazione delle risorse. Il sistema operativo deve facilitare la cooperazione, fornendo meccanismi di sincronizzazione.

Processi diretti

Processi "direttamente" a conoscenza uno dell'altro sono coloro che comunicano uno con l'altro sulla base dei loro id, la comunicazione è diretta, spesso basata sullo scambio di messaggi

Come interagiscono? cooperano per qualche scopo e comunicano informazioni agli altri processi. Il sistema operativo deve facilitare la cooperazione, fornendo meccanismi di comunicazione.

3.2.2 Proprietà fondamentali

Definizione: una proprietà di un programma concorrente è un attributo che rimane vero per ogni possibile storia di esecuzione del programma stesso.

Esistono due tipi di proprietà:

- **Safety ("nothing bad happens"):** mostrano che il programma (se avanza) va "nella direzione voluta", cioè non esegue azioni scorrette.
- **Liveness ("something good eventually happens"):** il programma avanza, non si ferma... insomma è "vitale".

Esempio: **Consensus**

Si consideri un sistema con N processi: all'inizio, ogni processo propone un valore. Alla fine, tutti i processi si devono accordare su uno dei valori proposti (decidono quel valore).

Proprietà di safety: se un processo decide, deve decidere uno dei valori proposti, se due processi decidono, devono decidere lo stesso valore.

Proprietà di liveness: prima o poi ogni processo corretto (*i.e. non in crash*) prenderà una decisione.

Proprietà - programmi sequenziali

Nei programmi sequenziali le **proprietà di safety** esprimono la correttezza dello stato finale (il risultato è quello voluto). La principale **proprietà di liveness** è la terminazione.

Proprietà - programmi concorrenti

Proprietà di safety: i processi non devono "interferire" fra di loro nell'accesso alle risorse condivise, questo vale ovviamente per i processi che condividono risorse (non per processi che cooperano tramite comunicazione). I meccanismi di sincronizzazione servono a garantire la proprietà di safety, devono essere usati propriamente dal programmatore, altrimenti il programma potrà contenere delle race condition.

Proprietà di liveness: i meccanismi di sincronizzazione utilizzati non devono prevenire l'avanzamento del programma. Non è possibile che tutti i processi si "blocchino", in attesa di eventi che non possono verificarsi perché generabili solo da altri processi bloccati. Non è possibile che un processo debba "attendere indefinitivamente" prima di poter accedere ad una risorsa condivisa.

3.2.3 Mutua esclusione

Definizione: l'accesso ad una risorsa si dice mutualmente esclusivo se ad ogni istante, al massimo un processo può accedere a quella risorsa.

Esempi da considerare: due processi che vogliono accedere contemporaneamente a una stampante. Due processi che cooperano scambiandosi informazioni tramite un buffer condiviso.

3.2.4 Deadlock (stallo)

La mutua esclusione permette di risolvere il problema della non interferenza, ma può causare il blocco permanente dei processi.

Esempio: Siano R_1 e R_2 due risorse. Siano P_1 e P_2 due processi che devono accedere a R_1 e R_2 contemporaneamente, prima di poter terminare il programma.

Supponiamo che il S.O. assegni R_1 a P_1 , e R_2 a P_2

I due processi sono bloccati in attesa circolare, si dice che P_1 e P_2 sono in **deadlock**. E' una condizione definitiva da evitare. Nei sistemi reali, se ne può uscire solo con metodi "distruttivi", ovvero uccidendo i processi, riavviando la macchina, etc.

3.2.5 Starvation (inedia)

Il deadlock è un problema che coinvolge tutti i processi che utilizzano un certo insieme di risorse. Esiste anche la possibilità che un processo non possa accedere ad una risorsa perché "sempre occupata".

Esempio: se siete in coda ad uno sportello e continuano ad arrivare "furbi" che passano davanti, non riuscirete mai a parlare con l'impiegato.

Esempio: Sia R una risorsa. Siano P_1 , P_2 e P_3 tre processi che devono accedere periodicamente a R.

Supponiamo che P_1 e P_2 si alternino nell'uso della risorsa P_3 non può accedere alla risorsa, perché utilizzata in modo esclusivo dagli altri due processi.

Si dice che P_3 è in **starvation**. A differenza del deadlock, non è una condizione definitiva è possibile uscirne, basta adottare un'opportuna politica di assegnamento. Rimane comunque una situazione da evitare.

3.2.6 Azioni atomiche

Definizione: le azioni atomiche vengono compiute in modo indivisibile, soddisfano la condizione: o tutto o niente. Nel caso di parallelismo reale si garantisce che l'azione non interferisce con altri processi durante la sua esecuzione. Nel caso di parallelismo apparente l'avvicendamento (context switch) fra i processi avviene prima o dopo l'azione, che quindi non può interferire.

Esempi: Le singole istruzioni del linguaggio macchina sono atomiche

```
1 sw $a0, ($t0)
```

Nel caso di parallelismo apparente: il meccanismo degli interrupt (su cui è basato l'avvicendamento dei processi) garantisce che un interrupt venga eseguito prima o dopo un'istruzione, mai "durante".

Nel caso di parallelismo reale: anche se più istruzioni cercano di accedere alla stessa cella di memoria (quella puntata da $t0$), la politica di arbitraggio del bus garantisce che una delle due venga servita per prima e l'altra successivamente.

Controesempi: In generale, sequenze di istruzioni in linguaggio macchina non sono azioni atomiche

```
1 lw $t0, ($a0)
2 add $t0, $t0, $a1
3 sw $t0, ($a0)
```

(Attenzione però: le singole istruzioni in linguaggio macchina sono atomiche, le singole istruzioni in assembly possono non essere atomiche!)

E nei compiti di concorrenza? Assumiamo che in ogni istante, vi possa essere al massimo un accesso alla memoria alla volta.

Questo significa che operazioni tipo: aggiornamento di una variabile, incremento di una variabile e valutazione di espressioni **non sono atomiche**.

Operazioni tipo: assegnamento di un valore costante ad una variabile, **sono atomiche**.

Annotazione

Nel seguito, utilizzeremo la notazione $\langle S \rangle$ per indicare che lo statement S deve essere eseguito in modo atomico

Esempio:

```
1  < x = x + 1; >
```

3.3 Sezioni critiche

Non interferenza

Se le sequenze di istruzioni non vengono eseguite in modo atomico, come possiamo garantire la non-interferenza?

Dobbiamo trovare il modo di specificare che certe parti dei programmi sono "speciali", ovvero devono essere eseguite in modo atomico (senza interruzioni).

Definizione: La parte di un programma che utilizza una o più risorse condivise viene detta **sezione critica** (critical section, o **CS**).

Esempio:

```
1  process P1 {
2      a1 = rand();
3      totale = totale + a1; //CS
4  }
5
6  process P2 {
7      a2 = rand();
8      totale = totale + a2; //CS
9 }
```

La parte evidenziata è una sezione critica, in quanto accede alla risorsa condivisa totale; mentre a1 e a2 non sono condivise.

Obiettivi: Vogliamo garantire che le sezioni critiche siano eseguite in modo mutualmente esclusivo (atomico) evitando così situazioni di blocco, sia dovute a deadlock sia dovute a starvation.

Sintassi:

- **[enter cs]** indica il punto di inizio di una sezione critica.
- **[exit cs]** indica il punto di fine di una sezione critica.

Perché abbiamo bisogno di costrutti specifici? Perché il S.O. non può capire da solo cosa è una sezione critica e cosa non lo è.

Esempio:

```
1  x=0
2  cobegin
3      [enter cs]; x = x+1; [exit cs];
4
5      [enter cs]; x = x+1; [exit cs];
6  coend
```

Requisiti delle CS

Si tratta di realizzare N processi della forma:

```
1  process Pi { // i=1...N
2      while (true) {
3          [enter cs]
4              critical section
5          [exit cs]
6              non-critical section
7      }
8  }
```

In modo che valgano le seguenti proprietà:

1. **Mutua esclusione:** solo un processo alla volta deve essere all'interno della CS, fra tutti quelli che hanno una CS per la stessa risorsa condivisa.
2. **Assenza di deadlock:** uno scenario in cui tutti i processi restano bloccati definitivamente non è ammissibile.
3. **Assenza di delay non necessari:** un processo fuori dalla CS non deve ritardare l'ingresso della CS da parte di un altro processo.
4. **Eventual entry (assenza di starvation:)** ogni processo che lo richiede, prima o poi entra nella CS.

N.B: Se un processo entra in una critical section, prima o poi ne uscirà, quindi esso può terminare solo fuori dalla sua sezione critica!

Possibili approcci

- **Approcci software:** la responsabilità cade sui processi che vogliono accedere ad un oggetto distribuito. Può essere soggetto ad errori e vedremo che è costoso in termini di esecuzione (busy waiting).
- **Approcci hardware:** utilizza istruzioni speciali del linguaggio macchina, progettate apposta. Sono efficienti ma non sono adatti come soluzioni general-purpose.
- **Approcci basati su supporto nel S.O. o nel linguaggio:** la responsabilità di garantire la mutua esclusione ricade sul S.O. o sul linguaggio. Esempi: *semafori, monitor, message passing.*

3.3.1 Tecniche software: Algoritmo di Dekker

```

1  shared int turn = P;
2  shared boolean needp = false; shared boolean needq = false;
3
4  cobegin P -> Q coend
5
6      process P {
7          while (true) {
8              // entry protocol
9              needp = true;
10             while (needq)
11                 if (turn == Q) {
12                     needp = false;
13                     while (turn == Q);
14                         // do nothing
15                     needp = true;
16                 }
17             //critical section
18             needp = false; turn = Q;
19             //non-critical section
20         }
21     }
22
23     process Q {
24         while (true) {
25             // entry protocol
26             needq = true;
27             while (needp)
28                 if (turn == P) {
29                     needq = false;
30                     while (turn == P);
31                         // do nothing
32                     needq = true;
33                 }
34             //critical section
35             needq = false; turn = P;
36             //non-critical section
37         }
38     }

```

Dimostrazione (mutua esclusione)

Per assurdo supponiamo che P e Q siano in CS contemporaneamente. Poiché gli accessi in memoria sono esclusivi, per entrare devono almeno aggiornare / valutare entrambe le variabili needp e needq.

Uno dei due entra per primo; diciamo sia Q: needq sarà true fino a quando Q non uscirà dal ciclo, poiché P entra nella CS mentre Q è nella CS, significa che esiste un istante temporale in cui needq = false e Q è in CS.

ASSURDO!

Dimostrazione (assenza di deadlock)

Per assurdo supponiamo che né P né Q possano entrare in CS. P e Q devono essere bloccati nel primo while, esiste un istante t dopo di che needp e needq sono sempre true.

Supponiamo che all'istante t, turn = Q. L'unica modifica a turn può avvenire solo quando Q entra in CS, dopo t, turn resterà sempre uguale a Q, P entra nel primo ciclo, e mette needp = false.
ASSURDO!

Dimostrazione (assenza di ritardi non necessari)

Se Q sta eseguendo codice non critico, allora needq = false. Allora P può entrare nella CS.

Dimostrazione (assenza di starvation)

Se Q richiede di accedere alla CS: needq = true. Se P sta eseguendo codice non critico: Q entra Se P sta eseguendo il resto del codice (CS, entrata, uscita): prima o poi ne uscirà e metterà il turno a Q. Q potrà quindi entrare.

3.3.2 Tecniche software: Algoritmo di Peterson

Più semplice e lineare di quello di Dijkstra / Dekker e facilmente generalizzabile al caso di processi multipli.

```
1 shared boolean needp = false;
2 shared boolean needq = false;
3 shared int turn;
4
5 cobegin P -> Q coend
6
7     process P {
8         while (true) {
9             // entry protocol
10            needp = true;
11            turn = Q;
12            while (needq && turn != P);
13                // do nothing
14            //critical section
15            needp = false;
16            //non-critical section
17        }
18    }
19
20    process Q {
21        while (true) {
22            // entry protocol
23            needq = true;
24            turn = P;
25            while (needp && turn != Q);
26                //do nothing
27            //critical section
28            needq = false;
29            //non-critical section
30        }
31    }
}
```

Dimostrazione (mutua esclusione)

Supponiamo che P sia entrato nella sezione critica.

Vogliamo provare che Q non può entrare, sappiamo che needP == true e Q entra solo se turn = Q quando esegue il while. Si consideri lo stato al momento in cui P entra nella critical section.

Ho due possibilità, needq == false or turn == P:

- Se needq == false, Q doveva ancora eseguire needq == true, e quindi lo eseguirà dopo l'ingresso di P e porrà turn=P, precludendosi la possibilità di entrare.
- Se turn==P, come sopra;

Dimostrazione (assenza di deadlock)

Supponiamo che per assurdo che P voglia entrare nella CS e sia bloccato nel suo ciclo while, questo significa che: needp = true, needq = true, turn = Q per sempre.

Possono darsi tre casi:

- Q non vuole entrare in CS: impossibile, visto che needq = true
- Q è bloccato nel suo ciclo while: impossibile, visto che turn = Q
- Q è nella sua CS e ne esce (prima o poi): impossibile, visto che prima o poi needq assumerebbe il valore false

Dimostrazione (assenza di ritardi non necessari)

Se Q sta eseguendo codice non critico, allora needq = false, allora P può entrare nella CS.

Dimostrazione (assenza di starvation)

Simile alla dimostrazione di assenza di deadlock, aggiungiamo un caso in fondo: Q continua ad entrare ed uscire dalla sua CS, prevenendo l'ingresso di P.

Impossibile poiché quando Q prova ad entrare nella CS pone turn = P siccome needp = true e quindi Q deve attendere che P entri nella CS.

3.3.3 Algoritmo di Peterson – Generalizzazione per N processi

```
1 shared int[] stage = new int[N]; /* 0-initialized */
2 shared int[] last = new int[N]; /* 0-initialized */
3
4 cobegin P0 -> P1 -> ... -> PN-1 coend
5
6 process Pi { // i = 0...N-1
7     while (true) {
8         // Entry protocol
9         for (int j=0; j < N; j++) {
10             stage[i] = j; last[j] = i;
11             for (int k=0; k < N; k++) {
12                 if (i != k){
13                     while (stage[k] >= stage[i] && last[j] == i)
14                     ;
15                 }
16             }
17         }
18         //critical section
19         stage[i] = 0;
20         //non-critical section
21     }
22 }
```

3.3.4 Tecniche hardware: disabilitazione interrupt, istruzioni speciali

Le soluzioni di Dekker e Peterson prevedono come uniche istruzioni atomiche le operazioni di Load e Store. Si può pensare di fornire alcune istruzioni hardware speciali per semplificare la realizzazione di sezioni critiche. Nei sistemi uniprocesso, i processi concorrenti vengono "alternati" tramite il meccanismo degli interrupt. Possiamo quindi disabilitarli.

Esempio:

```
1 process P {
2     while (true) {
3         //disable interrupt
4         critical section
5
6         //enable interrupt
7         non-critical section
8     }
9 }
```

Il problema è che il S.O. deve lasciare ai processi la responsabilità di riattivare gli interrupt ciò è altamente pericoloso perché riduce il grado di parallelismo ottenibile dal processore. Inoltre non funziona su sistemi multiprocessore.

Test e Set

Test e Set sono istruzioni che realizzano due azioni in modo atomico. *Esempi: lettura e scrittura, test e scrittura.*

$$TS(x, y) := < y = x; x = 1 >$$

Esso ritorna in y il valore precedente di x e assegna 1 ad x.

```
1     shared lock=0; cobegin P -> Q coend
2
3     process P {
4         int vp;
5         while (true) {
6             do {
7                 TS(lock, vp);
8             } while (vp);
9
10            //critical section
11            lock=0;
12            //non-critical section
13        }
14    }
15
16    process Q {
17        int vp;
18        while (true) {
19            do {
20                TS(lock, vp);
21            } while (vp);
22
23            //critical section
24            lock=0;
25            //non-critical section
26        }
27    }
```

Rispetta tutti i requisiti delle CS:

- Mutua esclusione: entra solo chi riesce a settare per primo il lock.
- No deadlock: il primo che esegue TS entra senza problemi.
- No unnecessary delay: un processo fuori dalla CS non blocca gli altri.
- No starvation: no, se non assumiamo qualcosa di più.

Riassumendo

- **Vantaggi delle istruzioni speciali hardware:**

- sono applicabili a qualsiasi numero di processi, sia su sistemi monoprocessoressi che in sistemi multiprocessori.
- semplice e facile da verificare.
- può essere utilizzato per supportare sezioni critiche multiple; ogni sezione critica può essere definita dalla propria variabile.

- **Svantaggi:**

- si utilizza ancora busy-waiting.
- i problemi di starvation non sono eliminati.
- sono comunque complesse da programmare.

3.4 Semafori

Due o più processi possono cooperare attraverso semplici segnali, in modo tale che un processo possa essere bloccato in specifici punti del suo programma finché non riceve un segnale da un altro processo.

3.4.1 Definizione

E' un tipo di dato astratto per il quale sono definite due operazioni:

- **V** (dall'olandese verhogen): viene invocata per inviare un segnale, quale il verificarsi di un evento o il rilascio di una risorsa.
- **P** (dall'olandese proberen): viene invocata per attendere il segnale (ovvero, per attendere un evento o il rilascio di una risorsa).

Un semaforo può essere visto come una variabile intera, che viene inizializzata ad un valore non negativo. L'operazione P attende che il valore del semaforo sia positivo e decremente il valore del semaforo. L'operazione V incrementa il valore del semaforo.

N.B: le azioni P e V sono atomiche

```
1 class Semaphore {  
2     private int val;  
3     Semaphore(int init) { }  
4         void P() {}  
5         void V() {}  
6 }
```

Semaforo - Invariante

Siano:

- n_P il numero di operazioni P completate
- n_V il numero di operazioni V completate
- init il valore iniziale del semaforo

Vale il seguente invariante: $n_P \leq n_V + init$

Due casi:

- **eventi** ($init = 0$) : il numero di eventi "consegnati" deve essere non superiore al numero di volte che l'evento si è verificato.
- **risorse** ($init \geq 0$) : il numero di richieste soddisfatte non deve essere superiore al numero iniziale di risorse + il numero di risorse restituite.

Implemetazione di CS

```
1 Semaphore s = new Semaphore(1);  
2  
3     process P {  
4         while (true) {  
5             s.P();  
6             //critical section  
7             s.V();  
8             //non-critical section  
9         }  
10    }
```

Politiche di gestione dei processi bloccati

Per ogni semaforo, il S.O. deve mantenere una struttura dati contenente l'insieme dei processi sospesi. Quando un processo deve essere svegliato, è necessario selezionare uno dei processi sospesi.

I **semafori FIFO** adottano la politica first-in, first-out. Il processo che è stato sospeso più a lungo viene svegliato per primo, è una politica fair, che garantisce assenza di starvation, la struttura dati è una coda. Se non viene specificata l'ordine in cui vengono rimossi, i semafori possono dare origine a starvation.

Da ora in poi utilizzeremo solo i semafori FIFO...

3.4.2 Implementazione

La funzione P

```
1 void P() {
2     value--;
3     if (value < 0) {
4         pid = <id del processo che ha invocato P>;
5         queue.add(pid);
6         suspend(pid);
7     }
8 }
```

Riga 5 Il process id del processo bloccato viene messo in un insieme queue.

Riga 6 Con l'operazione suspend, il s.o mette il processo nello stato waiting.

La funzione V

```
1 void V() {
2     value++;
3     if (value <= 0){
4         pid = queue.remove();
5         wakeup(pid);
6     }
7 }
```

Riga 4 Il process id del processo da sbloccare viene selezionato dall'insieme queue.

Riga 5 Con l'operazione wakeup, il S.O. mette il processo nello stato ready.

Quindi è necessario utilizzare una delle tecniche di critical section viste in precedenza: Dekker, Peterson oppure test-set, swap, etc.

```
1 void P() {
2     // [enter CS]
3     value--;
4     if (value < 0) {
5         int pid = <id del processo che ha invocato P>;
6         queue.add(pid);
7         suspend(pid);
8     }
9     // [exit CS]
10 }
11
12
13 void V() {
14     // [enter CS]
15     value++;
16     if (value <= 0){
17         int pid = queue.remove();
18         wakeup(pid);
19     }
20     // [exit CS]
21 }
```

Utilizzando queste tecniche, abbiamo limitato busy-waiting alle sezioni critiche di P e V, e queste sezioni critiche sono molto brevi, in questo modo la sezione critica non è quasi mai occupata e busy waiting avviene raramente.

3.4.3 Semafori binari

Definizione: variante dei semafori in cui il valore può assumere solo i valori 0 e 1.

A cosa servono? servono a garantire mutua esclusione, semplificando il lavoro del programmatore. Hanno lo stesso potere espressivo dei semafori "normali".

Invariante dei semafori binari: $0 \leq n_V + init - n_P \leq 1$ oppure $0 \leq s.value \leq 1$

Implementazione nei sistemi operativi

```
1  class BinarySemaphore {
2      private int value;
3      Queue queue0 = new Queue();
4      Queue queue1 = new Queue();
5      BinarySemaphore() { value = 1; }
6
7      void P() {
8          // [enter CS]
9          int pid = <process id>;
10         if (value == 0) {
11             queue0.add(pid);
12             suspend(pid);
13         }
14         value--;
15
16         if (queue1.size() > 0) {
17             int pid = queue1.remove();
18             wakeup(pid);
19         }
20         // [exit CS]
21     }
22
23     void V() {
24         // [enter CS]
25         int pid = <process id>;
26         if (value == 1) {
27             queue1.add(pid);
28             suspend(pid);
29         }
30         value++;
31
32         if (queue0.size() > 0) {
33             int pid = queue0.remove();
34             wakeup(pid);
35         }
36         // [exit CS]
37     }
}
```

Implementazione tramite semafori generali

```
1  class BinarySemaphore {
2      private Semaphore s0 , s1 ;
3      int value ;
4
5      BinarySemaphore(int v){ // fail if v not in {0,1}
6          s0 = new Semaphore(v)
7          s1 = new Semaphore(1-v)
8      }
9
10     void P(void){
11         s0.P();
12         s1.V();
13     }
14
15     void V(void){
16         s1.P();
17         s0.V();
18     }
19 }
```

3.5 Problemi classici

Esistono un certo numero di problemi "classici" della programmazione concorrente:

- producer/consumer
- bounded buffer
- dining philosophers
- readers/writers

Nella loro semplicità rappresentano le interazioni tipiche dei processi concorrenti.

3.5.1 Producer/Consumer

Definizione: esiste un processo "produttore" **Producer** che genera valori (record, caratteri, oggetti, etc.) e vuole trasferirli a un processo "consumatore" **Consumer** che prende i valori generati e li "consuma". La comunicazione avviene attraverso una singola variabile condivisa.

Proprietà da garantire: Producer non deve scrivere nuovamente l'area di memoria condivisa prima che Consumer abbia effettivamente utilizzato il valore precedente. Consumer non deve leggere due volte lo stesso valore, ma deve attendere che Producer abbia generato il successivo. Assenza di deadlock.

```
1 shared Object buffer;
2 Semaphore empty = new Semaphore(1);
3 Semaphore full = new Semaphore(0);
4
5 //cobegin
6   Producer
7   ....
8   Consumer
9 //coend
10
11 process Producer {
12   while (true) {
13     Object val = produce();
14     empty.P();
15     buffer = val;
16     full.V();
17   }
18 }
19
20 process Consumer {
21   while (true) {
22     full.P();
23     Object val = buffer;
24     empty.V();
25     consume(val);
26   }
27 }
```

3.5.2 Bounded Buffer

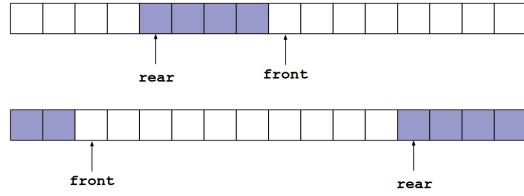
Definizione: è simile al problema del produttore/consumatore. In questo caso, però, lo scambio tra produttore e consumatore non avviene tramite un singolo elemento, ma tramite un buffer di dimensione limitata, i.e. un vettore di elementi.

Proprietà da garantire: Producer non deve sovrascrivere elementi del buffer prima che Consumer abbia effettivamente utilizzato i relativi valori. Consumer non deve leggere due volte lo stesso valore, ma deve attendere che Producer abbia generato il successivo. Assenza di deadlock e assenza di starvation.

Struttura del buffer

Array circolare: si utilizzano due indici front e rear che indicano rispettivamente il prossimo elemento da scrivere e il prossimo elemento da leggere. Gli indici vengono utilizzati in modo ciclico (modulo l'ampiezza del buffer).

Esempi:



```

1  Object buffer[SIZE];
2  int front = 0;
3  int rear = 0;
4  Semaphore empty = new Semaphore(SIZE);
5  Semaphore full = new Semaphore(0);
6
7  process Producer {
8      while (true) {
9          Object val = produce();
10         empty.P();
11         buf[front] = val;
12         front = (front + 1) % SIZE;
13         full.V();
14     }
15 }
16
17 process Consumer {
18     while (true) {
19         full.P();
20         Object val = buf[rear];
21         rear = (rear + 1) % SIZE;
22         empty.V();
23         consume(val);
24     }
25 }
```

3.5.3 Cena dei Filosofi

Descrizione: Cinque filosofi passano la loro vita a pensare e a mangiare (alternativamente). Per mangiare fanno uso di una tavola rotonda con 5 sedie, 5 piatti e 5 posate fra i piatti. Per mangiare, un filosofo ha bisogno di entrambe le posate (destra/sinistra), per pensare, un filosofo lascia le posate dove le ha prese.

I problemi produttore/consumatore e buffer limitato mostrano come risolvere il problema di accesso esclusivo a una o più risorse indipendenti.

Il problema dei filosofi mostra come gestire situazioni in cui i processi entrano in competizione per accedere ad insiemi di risorse a intersezione non nulla.

La vita di un filosofo

```

1  process Philo[i] { /* i = 0...4 */
2      while (true) {
3          //think
4          //acquire chopsticks
5          //eat
6          //release chopsticks
7      }
8  }
```

Le bacchette vengono denominate: **chopstick[i]** con $i=0\dots4$; Il filosofo i accede alle posate **chopstick[i]** e **chopstick[(i+1) \% 5]**;

Invarianti

up_i il numero di volte che la bacchetta i viene preso dal tavolo $down_i$ il numero di volte che la bacchetta i viene rilasciata sul tavolo

Invariante $down_i \leq up_i \leq down_i + 1$

Per comodità si può definire $chopstick[i] = 1 - (up_i - down_i)$ (può essere pensato come un semaforo binario)

Soluzione

```
1 Semaphore chopsticks = { new Semaphore(1), ..., new Semaphore(1)};
2
3     process Philo[0] {
4         while (true) {
5             //think
6             chopstick[1].P();
7             chopstick[0].P();
8             //eat
9             chopstick[1].V();
10            chopstick[0].V();
11        }
12    }
13
14    process Philo[i] /* i = 1...4 */ {
15        while (true) {
16            //think
17            chopstick[i].P();
18            chopstick[(i+1)%5].P();
19            //eat
20            chopstick[i].V();
21            chopstick[(i+1)%5].V();
22        }
23    }
}
```

3.5.4 Lettori e scrittori

Descrizione: Un database è condiviso tra un certo numero di processi, esistono due tipi di processi. I lettori accedono al database per leggerne il contenuto e gli scrittori accedono al database per aggiornarne il contenuto.

Proprietà: Se uno scrittore accede a un database per aggiornarlo, esso opera in mutua esclusione; nessun altro lettore o scrittore può accedere al database. Se nessuno scrittore sta accedendo al database, un numero arbitrario di lettori può accedere al database in lettura.

Motivazioni: La competizione per le risorse avviene a livello di classi di processi e non solo a livello di processi. Mostra che mutua esclusione e condivisione possono anche coesistere.

Invariante: Sia **nr** il numero dei lettori che stanno accedendo al database e sia **nw** il numero di scrittori che stanno accedendo al database.

L'invariante è il seguente: $(nr > 0 \&& nw == 0) \parallel (nr == 0 \&& nw \leq 1)$

NOTA: il controllo può passare dai lettori agli scrittori o viceversa quando: $nr == 0 \&& nw == 0$

```
1 process Reader {
2     while (true) {
3         startRead();
4         //read the database
5         endRead();
6     }
7 }
8
9 process Writer {
10    while (true) {
11        startWrite();
12        //write the database
13        endWrite();
14    }
15 }
```

Note: startRead() e endRead() contengono le operazioni necessarie affinché un lettore ottenga accesso al database.

startWrite() e endWrite() contengono le operazioni necessarie affinché uno scrittore ottenga accesso al database.

Il problema dei lettori e scrittori ha molte varianti che si basano sui diversi concetti di priorità.

Priorità ai lettori Se un lettore vuole accedere al database, lo potrà fare senza attesa a meno che uno scrittore non abbia già acquisito l'accesso al database.

Scrittori hanno possibilità di starvation.

Priorità agli scrittori Uno scrittore attenderà il minimo tempo possibile prima di accedere al db. Lettori hanno possibilità di starvation.

Lettori e scrittori - Soluzione

```
1  /* Variabili condivise */
2  int nr = 0;
3  Semaphore rw = new
4  Semaphore(1);
5  Semaphore mutex = new
6  Semaphore(1);
7
8  void startRead() {
9      mutex.P();
10     if (nr == 0){
11         rw.P();
12         nr++;
13         mutex.V();
14     }
15     void startWrite() {
16         rw.P();
17     }
18
19     void endRead() {
20         mutex.P();
21         nr--;
22         if (nr == 0) {
23             rw.V();
24             mutex.V();
25         }
26
27     void endWrite() {
28         rw.V();
29     }
```

Problemi: limitata a priorità per i lettori, di comprensione non semplice, non è chiaro da dove saltano fuori alcuni punti della soluzione.

3.5.5 Come derivare una soluzione basata su semafori

Alcune definizioni utili - Andrews

Sia **B** una condizione booleana.

Sia **S** uno statement (possibilmente composto)

Allora: $\langle S \rangle$: esegui lo statement S in modo atomico.

$\langle \text{await}(B) \rightarrow S \rangle$: attendi fino a che la condizione B è verificata e quindi esegui S. L'attesa e il comando vengono eseguiti in modo atomico, quindi, quando S viene eseguito, B è verificata.

Passi da seguire:

1. Definire il problema con precisione: identificare i processi, specificare i problemi di sincronizzazione, introdurre le variabili necessarie e definire un'invariante.
2. Abbozzare una soluzione: produrre un primo schema di soluzione, e identificare le regioni che richiedono accesso atomico o mutualmente esclusivo.

3. Garantire l'invariante: verifica che l'invariante sia sempre verificato
4. Implementare le azioni atomiche: esprimere le azioni atomiche e gli statement **await** utilizzando le primitive di sincronizzazione disponibili.

Soluzione Lettori/Scrittori con Andrews

Variabili: nr, nw - numero corrente di lettori/scrittori.
Invariante: $(nr > 0 \&\& nw == 0) \mid\mid (nr == 0 \&\& nw \leq 1)$

Schema della soluzione:

```

1 process Reader {
2     < await (nw == 0) = nr++ >
3     //read the database
4     <nr-->
5 }
6
7 process Writer {
8     < await (nr == 0 && nw == 0) = nw++ >
9     //write the database
10    <nw-->
11 }
```

dalla slide 142 alla 155 guardare sulle slide

3.6 Conditional Critical Regions (CCR)

3.6.1 Introduzione

I linguaggi di programmazione concorrente sono dotati di costrutti ad alto livello, che si propongono di prevenire la possibilità di errori dovuti all'uso scorretto delle primitive.

I costrutti di programmazione concorrente sottraggono al programmatore la responsabilità dell'uso delle primitive limitando così la possibilità di accessi indiscriminati ai dati comuni e favorendo la programmazione strutturata.

Questo è il compito del compilatore del linguaggio concorrente, che traduce i costrutti per la concorrenza in un insieme di primitive per la concorrenza.

Le **regioni critiche condizionali** sono costrutti che specificano operazioni su dati condivisi, da eseguire in mutua esclusione, che possono determinare la sospensione e la riattivazione dei processi.

Forniscono una notazione più strutturata di quella dei semafori per specificare la sincronizzazione.

Sintassi CCR

name (var declarations)

name è un'identificatore per la risorsa condivisa.

var declarations è un'insieme di variabili condivise. Dichiara che le variabili racchiuse tra parentesi sono condivise e devono essere accedute in mutua esclusione.

region name when condition do statement condition è una condizione booleana. **region name do statement statement** è uno statement (potenzialmente composto) da eseguire. È l'istruzione per acquisire la mutua esclusione su **name**.

L'esecuzione consiste in:

- Acquisire la mutua esclusione
- Valutare la condizione:
 - se è falsa, rilasciare la mutua esclusione e ritardare fino a quando la condizione non è vera.
 - se è vera, eseguire statement.

Vantaggi CCR

Il compilatore del linguaggio attraverso il controllo dello scope delle variabili, può rilevare eventuali riferimenti illegittimi ai dati comuni.

Esempio: riferimenti non inclusi in regioni critiche condizionali associate al nome corretto

Inoltre: "Compila" i costrutti region tramite le opportune chiamate a primitive di sincronizzazione (ad. es., semafori)

Implementazione CCR tramite semafori

Il ritardo di un processo che, all'interno della CCR, trovi la condizione falsa è realizzata con la sospensione fuori dalla mutua esclusione.

Una tecnica per rilevare la transizione al valore vero della condizione consiste nel far ripetere la verifica della condizione al processo medesimo, che naturalmente deve ri-accedere alla mutua esclusione.

```

1 resource name (var declarations)
2
3 /* Mutual exclusion semaphore, one for each critical region */
4 Semaphore mutex_name = new Semaphore(1);
5 /* Processes for which the condition is false must be
6 suspended */
7
8 Semaphore suspended_name = new Semaphore(0);
9 /* Number of suspended processes */
10 int nsuspended_name = 0;
11 /* Additional declarations */
12 var declarations

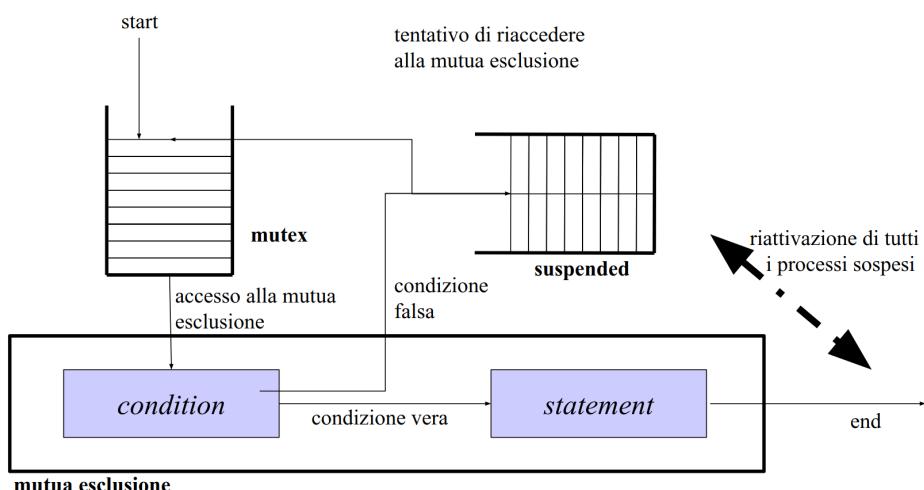
```

```

1 region name when condition do statement
2
3 mutex_name.P();
4 while (!condition) {
5     /* condition is false */
6     nsuspended_name++;
7     mutex_name.V();
8     suspended_name.P();
9 /* when process is reactive, it must re-gain access to the mutual
10   exclusion */
11     mutex_name.P();
12 }
13 /* condition is true */
14 statement;
15 /* after statement, one or more conditions may be true */
16 while (nsuspended_name-- > 0)
17     suspended_name.V();

```

Schema CCR-Semafori



Svantaggi: questa realizzazione può risultare molto inefficiente nei sistemi uniprocesso, infatti, i processi possono essere riattivati ripetutamente prima di trovare la condizione vera. Numerosi context switch del tutto improduttivi.

Vantaggi: è più facile da realizzare qualora le condizioni siano complesse e coinvolgano variabili locali (*non condivise*) dei processi coinvolti.

3.6.2 CCR tramite semafori (passing the baton)

```

1 resource name (var declarations)
2
3     Semaphore mutex_name = new Semaphore(1);
4     Semaphore suspended_name[M_name] = { new Semaphore(0), ..., new
5         Semaphore(0) };
6
7     int nsuspended_name[M_name] = { 0, ..., 0 };
8     //var declarations

```

dove M_{name} è il numero di regioni critiche con condizione associate a **name**.

```

1 region name when condition_i do statement
2
3     mutex_name.P();
4     if (!condition_i) {
5         nsuspended_name[i]++;
6         mutex_name.V();
7         suspended_name[i].P();
8     }
9     statement;
10    SIGNAL();
11
12
13    void SIGNAL() {
14        if (nsuspended_name[0] > 0 && condition_0)
15            { nsuspended_name[0]--; suspended_name[0].V(); }
16
17        else if (nsuspended_name[1] > 0 && condition_1)
18            { nsuspended_name[0]--; suspended_name[1].V(); }
19
20        else if (... )
21            ...
22        else
23            mutex_name.V();
24    }

```

3.6.3 Implementazione di semafori tramite CCR

Semaforo generali; valore iniziale n

```

1 class Semaphore {
2     resource sem ( int value; );
3
4     Semaphore(int n) {
5         /* Initialization; done only once */
6         region sem do value = n;
7     }
8
9     P() {
10         region sem when (value>0) do value--;
11     }
12
13     V() {
14         region sem do value++;
15     }
16 }

```

3.6.4 CCR - Produttore / Consumatore

```

1 resource buffer (Object b; boolean full = false);
2
3 process Producer {
4     while (true) {
5         Object val = produce();
6         region buffer when (!full) do { b = val; full = true; }
7     }
8 }
9
10 process Consumer {
11     while (true) {
12         Object val;
13         region buffer when (full) do { val = b; full = false; }
14         consume(val);
15     }
16 }
```

3.6.5 CCR - Filosofi a cena

```

1 resource table (
2     boolean eating[N] = { false, false, false, false, false }
3 )
4
5 process Philo[i] {
6     while (true) {
7         //think
8         int left = (i+N-1) % N;
9         int right = (i+1) % N;
10        region table when (!eating[left] && !eating[right]) do
11
12            eating[i] = true;
13            //eat
14            region table do eating[i] = false;
15        }
16    }
```

3.6.6 CCR - R/W Priorità ai lettori

```

1 resource db ( int nr = 0; int nw = 0; );
2
3 process Reader {
4     while (true) {
5         region db when (nw==0) do nr++;
6         //read the database
7         region db do nr--;
8     }
9 }
10
11 process Writer {
12     while (true) {
13         region db when (nr==0 && nw==0) do nw++;
14         //write the database
15         region db do nw--;
16     }
17 }
```

3.6.7 CCR - R/W Priorità agli scrittori

```
1 resource db ( int nr = 0; int nw = 0; int ww = 0; );
2
3     process Reader {
4         while (true) {
5             region db when (nw==0 && ww=0) do nr++;
6             //read the database
7             region db do nr--;
8         }
9     }
10
11    process Writer {
12        while (true) {
13            region db do ww++;
14            region db when (nr==0 && nw==0) do { nw++; ww--; }
15            //write the database
16            region db do nw--;
17        }
18    }
}
```

3.6.8 CCR - Bounded buffer (accesso esclusivo)

```
1 resource buffer (
2     Object buf[N];
3     int front=0; int rear=0;
4     int count=0;
5 );
6
7     process Producer {
8         while (true) {
9             Object val = produce();
10            region buffer
11            when (count < N) do {
12                buf[front] = val;
13                front = (front+1) % N;
14                count++;
15            }
16        }
17    }
18
19    process Consumer {
20        Object val;
21        while (true) {
22            region buffer
23            when (count > 0) do {
24                val = buf[rear];
25                rear = (rear+1) % N;
26                count--;
27            }
28            consume(val);
29        }
30    }
}
```

3.7 Monitor

3.7.1 Introduzione

I monitor sono un paradigma di programmazione concorrente che fornisce un approccio più strutturato alla programmazione concorrente.

Un monitor è un modulo software che consiste di: dati locali, una sequenza di inizializzazione e una o più "procedure".

Le caratteristiche principali sono:

- I dati locali sono accessibili solo alle procedure del modulo stesso.
- Un processo entra in un monitor invocando una delle sue procedure.
- Solo un processo alla volta può essere all'interno del monitor; gli altri processi che invocano il monitor sono sospesi, in attesa che il monitor diventi disponibile.

```
1 monitor name {  
2  
3     //private variable declarations...  
4  
5     procedure entry type procedurename1(args...) {  
6         //visible procedures  
7     }  
8  
9     type procedurename2(args...) {  
10        //private procedures  
11    }  
12  
13    name(args...) {  
14        //initialization  
15    }  
16}
```

Assomiglia ad un "oggetto" nella programmazione Object Oriented, il codice di inizializzazione corrisponde al costruttore.

Le procedure entry sono richiamabili dall'esterno e corrispondono ai metodi pubblici di un oggetto.

Le procedure "normali" corrispondono ai metodi privati. Le variabili locali corrispondono alle variabili pubbliche.

Caratteristiche

Solo un processo alla volta può essere all'interno del monitor, esso fornisce un semplice meccanismo di mutua esclusione. Strutture dati condivise possono essere messe all'interno del monitor.

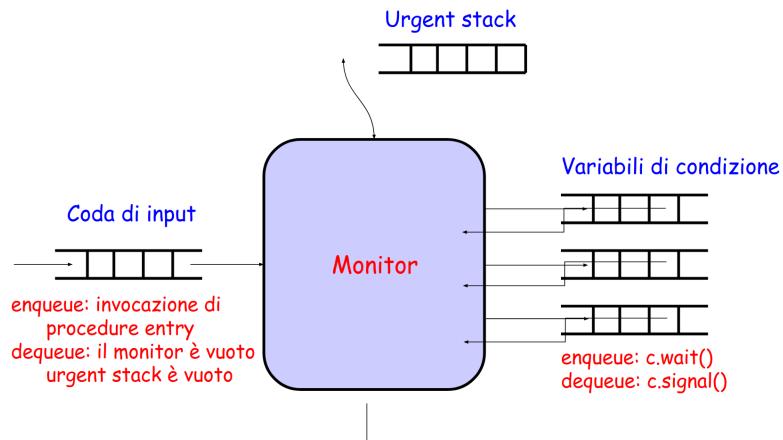
Per essere utile per la programmazione concorrente, è necessario un meccanismo di sincronizzazione. Abbiamo quindi necessità di poter sospendere i processi in attesi di qualche condizione, far uscire i processi dalla mutua esclusione mentre sono in attesa e permettergli di rientrare quando la condizione è verificata.

Meccanismi di sincronizzazione

Dichiarazione di variabili di condizione (CV) `condition c`; Le operazioni definite sulle CV sono:

- `c.wait()` - attende il verificarsi della condizione. Viene rilasciata la mutua esclusione, il processo che chiama `c.wait()` viene sospeso in una coda di attesa della condizione `c`.
- `c.signal()` - segnala che la condizione è vera. Causa la riattivazione immediata di un processo (secondo una politica FIFO), il chiamante viene posto in attesa e verrà riattivato quando il processo risvegliato avrà rilasciato la mutua esclusione (urgent stack). Se nessun processo sta attendendo `c` la chiamata non avrà nessun effetto.

Rappresentazione grafica



wait/signal vs P/V

A prima vista **wait** e **signal** potrebbero sembrare simili alle operazioni sui semafori **P** e **V**, ma in verità ci sono differenze sostanziali.

- **signal** non ha alcun effetto se nessun processo sta attendendo la condizione. Mentre **V** "memorizza" il verificarsi degli eventi.
- **wait** è sempre bloccante, **P** (se il semaforo ha valore positivo) no.
- Il processo risvegliato dalla **signal** viene eseguito per primo.

Politiche di signaling

Signal urgent è la politica "classica" di signaling. SU - signal urgent.
Ne esistono altre..

3.7.2 Implementazione dei semafori

```

1 monitor Semaphore {
2     int value;
3     condition c; /* value > 0 */
4
5     procedure entry void P() {
6         value--;
7         if (value < 0)
8             c.wait();
9     }
10
11    procedure entry void V() {
12        value++;
13        c.signal();
14    }
15
16    Semaphore(int init) {
17        value = init;
18    }
19}
```

3.7.3 Monitor - R/W

```
1 process Reader {
2     while (true) {
3         rwController.startRead();
4         //read the database
5         rwController.endRead();
6     }
7 }
8
9 process Writer {
10    while (true) {
11        rwController.startWrite();
12        //write the database
13        rwController.endWrite();
14    }
15 }
```

```

1 monitor RWController
2     int nr; /* number of readers */
3     int nw; /* number of writers */
4     condition okToRead; /* nw == 0 */
5     condition okToWrite; /* nr == 0 && nw == 0 */
6
7     procedure entry void startRead() {
8         if (nw != 0)
9             okToRead.wait();
10        nr = nr + 1;
11
12        if (nw == 0) /* always true */
13            okToRead.signal();
14
15        if (nw == 0 && nr == 0) /* always false */
16            okToWrite.signal();
17    }
18
19    procedure entry void endRead() {
20        nr = nr - 1;
21        if (nw == 0) /* true but useless */
22            okToRead.signal();
23
24        if (nw == 0 && nr == 0)
25            okToWrite.signal();
26    }
27
28    procedure entry void startWrite() {
29        if (!(nr==0 && nw==0))
30            okToWrite.wait();
31        nw = nw + 1;
32
33        if (nw == 0) /* always true */
34            okToRead.signal();
35
36        if (nw == 0 && nr == 0) /* always false */
37            okToWrite.signal();
38    }
39
40    procedure entry void endWrite() {
41        nw = nw - 1;
42        if (nw == 0) /* Always true */
43            okToRead.signal();
44
45        if (nw == 0 && nr == 0)
46            okToWrite.signal();
47    }
48
49    RWController() { /* Constructor */
50        nr = nw = 0;
51    }

```

E' possibile semplificare il codice eliminando le righe if quando sempre vere, eliminando le righe if e il ramo opportuno quando sempre falso.

Monitor - R/W semplificato

```
1 procedure entry void startRead() {
2     if (nw != 0) okToRead.wait();
3     nr = nr + 1;
4     okToRead.signal();
5 }
6
7 procedure entry void endRead() {
8     nr = nr - 1;
9     if (nr == 0)
10        okToWrite.signal();
11 }
12
13 procedure entry void startWrite() {
14     if (!(nr==0 && nw ==0))
15        okToWrite.wait();
16     nw = nw + 1;
17 }
18
19 procedure entry void endWrite() {
20     nw = nw - 1;
21     okToRead.signal();
22     if (nw == 0 && nr == 0)
23        okToWrite.signal();
24 }
```

3.7.4 Monitor - Produttore / consumatore

```
1 process Producer {
2     Object x;
3     while (true) {
4         x = produce();
5         pcController.write(x);
6     }
7 }
8
9 process Consumer {
10    Object x;
11    while (true) {
12        x = pcController.read();
13        consume(x);
14    }
15 }
```

```

1 monitor PCController {
2     Object buffer;
3     condition empty;
4     condition full;
5     boolean isFull;
6
7     PCController() {
8         isFull=false;
9     }
10
11
12     procedure entry Object read() {
13         if (!isFull)
14             full.wait();
15
16         int retval = buffer;
17         isFull = false;
18         empty.signal();
19         return retval;
20     }
21
22     procedure entry void write(int val) {
23         if (isFull)
24             empty.wait();
25
26         buffer = val;
27         isFull = true;
28         full.signal();
29     }
30 }
```

3.7.5 Monitor - Buffer limitato

```

1 monitor PCController {
2     Object[] buffer;
3     condition okRead, okWrite;
4     int count, rear, front;
5
6     PCController(int size) {
7         buffer = new Object[size];
8         count = rear = front = 0;
9     }
10
11    procedure entry Object read() {
12        if (count == 0)
13            okRead.wait();
14
15        int retval = buffer[rear];
16        cont--;
17        rear = (rear+1) % buffer.length;
18        okWrite.signal();
19        return retval;
20    }
21
22    procedure entry void write(int val) {
23        if (count == buffer.length)
24            okWrite.wait();
25        buffer[front] = val;
26        count++;
27        front = (front+1) %
28        buffer.length;
29        okRead.signal();
30    }
```

3.7.6 Monitor - Filosofi a cena (no deadlock)

```
1 process Philo[i] {
2     while (true) {
3         //think
4         chopstick[MIN(i,i+1)].pickup();
5         chopstick[MAX(i,i+1)].pickup();
6         //eat
7         chopstick[MIN(i,i+1)].putdown();
8         chopstick[MAX(i,i+1)].putdown();
9     }
10 }
```

```
1 monitor DPController {
2     condition[] unusedchopstick = new condition[5];
3     boolean[] chopstick = new boolean[5];
4
5     procedure entry void startEating(int i) {
6
7         if (chopstick[i])
8             unusedchopstick[i].wait();
9             chopstick[i] = true;
10
11         if (chopstick[i+1])
12             unusedchopstick[i+1].wait();
13             chopstick[i+1] = true;
14     }
15
16     procedure entry void finishEating(int i) {
17         chopstick[i] = false;
18         chopstick[i+1] = false;
19         unusedchopstick[i].signal();
20         unusedchopstick[i+1].signal();
21     }
22 }
23
24 monitor chopstick[i] {
25     boolean inuse = false;
26     condition free;
27
28     procedure entry void pickup() {
29         if (inuse)
30             free.wait();
31             inuse = true;
32     }
33
34     procedure entry void putdown() {
35         inuse = false;
36         free.signal();
37     }
}
```

3.7.7 Implementazione dei monitor tramite semafori

Ingredienti:

- un modulo di gestione stack (per urgent)

```
1     interface Stack {
2         void push(Object x);
3         Object pop();
4         boolean empty();
5     }
```

- un semaforo di mutua esclusione e
- per ogni variabile di condizione $cond_i$, una coppia (c_i, nc_i) . c_i è un semaforo correlato alla condizione, inizializzato a 0. nc_i è il numero di processi che sono in attesa del verificarsi della condizione.
- un "allocatore" di semafori (*o alternativamente un semaforo per ogni processo*).

```
1 //Inizializzazione
2     Semaphore e = new Semaphore(1);
3     Stack stack = new Stack();
4
5 //Entrata nel monitor
6     e.P();
7
8 //Wait su cond_i
9     nc_i++;
10    if (!stack.empty()) {
11        Semaphore s = stack.pop();
12        s.V();
13    } else {
14        e.V();
15    }
16    c_i.P();
17
18 //Signal su cond_i
19    if (nci > 0) {
20        nc_i--;
21        ci.V();
22        Semaphore s = new Semaphore(0);
23        stack.push(s);
24        s.P();
25        /* free(s) / garbage coll. */
26    }
27
28 //Uscita dal monitor
29    if (!stack.empty()) {
30        Semaphore s = stack.pop();
31        s.V();
32    } else {
33        e.V();
34    }
```

3.8 Message passing

3.8.1 Introduzione

Abbiamo già visto paradigmi di sincronizzazione come: semafori, monitor. In questi paradigmi, la comunicazione avviene tramite memoria condivisa.

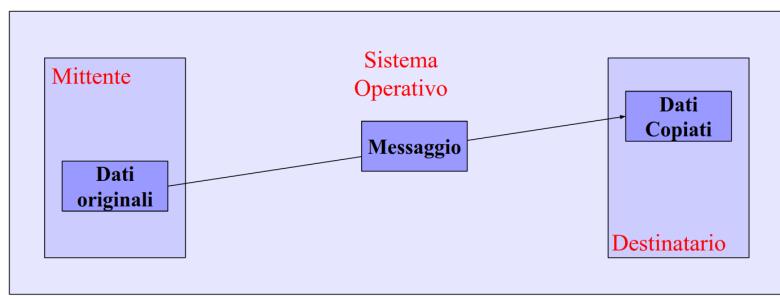
I paradigmi di comunicazione come il **message passing** permettono la comunicazione tra processi tramite messaggi.

La sincronizzazione avviene tramite lo scambio di messaggi, e non più semplici segnali.

Definizioni

Un **messaggio** è un insieme di informazioni formattate da un processo mittente e interpretate da un processo destinatario.

Un meccanismo di **"scambio di messaggi"** copia le informazioni di un messaggio da uno spazio di indirizzamento di un processo allo spazio di indirizzamento di un altro processo.



Operazioni

send: utilizzata dal processo mittente per "spedire" un messaggio ad un processo destinatario specificato.

receive: utilizzata dal processo destinatario per "ricevere" un messaggio da un processo mittente. Il processo mittente può essere specificato, o no.

Il passaggio dallo spazio di indirizzamento del mittente a quello del destinatario è mediato dal sistema operativo (protezione memoria), il processo destinatario deve eseguire un'operazione receive per ricevere qualcosa.

Tassonomia

- MP sincrono
 - Send sincrono
 - Receive bloccante
- MP asincrono
 - Send asincrono
 - Receive bloccante
- MP completamente asincrono
 - Send asincrono
 - Receive non bloccante

3.8.2 MP sincrono

Operazione send sincrona

Sintassi: `ssend(m, q)`.

Il mittente p spedisce il messaggio m al processo q, restando bloccato fino a quando q non esegue l'operazione `sreceive(m, p)`.

Operazione receive bloccante

Sintassi: `m = sreceive(p)`

Il destinatario q riceve il messaggio m dal processo p; se il mittente non ha ancora spedito alcun messaggio, il destinatario si blocca in attesa di ricevere un messaggio, è possibile lasciare il mittente non specificato (utilizzando *).

3.8.3 MP asincrono

Operazione send asincrona

Sintassi: `asend(m, q)`.

il mittente p spedisce il messaggio m al processo q, senza bloccarsi in attesa che il destinatario esegua l'operazione `areceive(m, p)`.

Operazione receive bloccante

Sintassi: `m = areceive(p)`

Il destinatario q riceve il messaggio m dal processo p; se il mittente non ha ancora spedito alcun messaggio, il destinatario si blocca in attesa di ricevere un messaggio, è possibile lasciare il mittente non specificato (utilizzando *).

3.8.4 MP completamente asincrono

Operazione send asincrona

Sintassi: `asend(m, q)`.

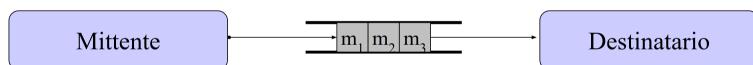
il mittente p spedisce il messaggio m al processo q, senza bloccarsi in attesa che il destinatario esegua l'operazione `nb-receive(m, p)`.

Operazione receive non bloccante

Sintassi: `m = nb-receive(p)`

Il destinatario q riceve il messaggio m dal processo p; se il mittente non ha ancora spedito alcun messaggio, la nb-receive termina ritornando un messaggio "nullo", è possibile lasciare il mittente non specificato (utilizzando *).

Message passing asincrono



Message passing sincrono



In letteratura ci sono numerose diverse sintassi per descrivere message passing. Ad esempio, invece che indicare il processo destinazione/mittente, si indica il nome di un canale.

Message passing asincrono con 3 primitive principali: send, receive, reply (Thoth). Non la receive, ma solamente la reply sblocca il mittente, utile per rendere MP simile alle chiamate di procedura remota.

3.8.5 MP sincrono dato quello asincrono

```

1 void ssend(Object msg, Process q) {
2     asend(msg, q);
3     ack = areceive(q);
4 }
5
6 Object sreceive(p) {
7     Object msg = areceive(p);
8     asend(ack, p);
9     return msg;
10}

```

```

/* p identifies the calling process */
1
2
3 void asend(Object m, Process q) {
4     ssend(SND(m,p,q), server);
5 }
6
7 void areceive(Process q) {
8     ssend(RCV(p,q), server);
9     Object m = sreceive(server);
10    return m;
11 }
12
13 process server {
14 /* One element x process pair */
15     int [][] waiting;
16     Queue [][] queue;
17     while (true) {
18         handleMessage();
19     }
20 }
21
22 void handleMessage() {
23     msg = sreceive(*);
24     if (msg == <SND(m,p,q)>) {
25         if (waiting[p,q]>0) {
26             ssend(m, p);
27             waiting[p,q]--;
28         } else {
29             queue[p,q].add(m);
30         }
31     } else if (msg == <RCV(q,p)>) {
32         if (queue[p,q].isEmpty()) {
33             waiting[p,q]++;
34         } else {
35             m = queue[p,q].remove();
36             ssend(m, dest);
37         }
38     }
39 }

```

3.8.6 Message Passing - Filosofi a cena

```
1 process Philo[i] {
2     while (true) {
3         //think
4         asend(<PICKUP,i>, chopstick[MIN(i, (i+1)%5)]);
5         msg = areceive(chopstick[MIN(i, (i+1)%5)]);
6         asend(<PICKUP,i>, chopstick[MAX(i, (i+1)%5)]);
7         msg = areceive(chopstick[MAX(i, (i+1)%5)]);
8         //eat
9         asend(<PUTDOWN,i>, chopstick[MIN(i, (i+1)%5)]);
10        asend(<PUTDOWN,i>, chopstick[MAX(i, (i+1)%5)]);
11    }
12 }
```

```
1 process chopstick[i] {
2     boolean free = true;
3     Queue queue = new Queue();
4     while (true) {
5         handleRequests();
6     }
7 }
8
9 void handleRequests() {
10    msg = areceive(*);
11    if (msg == <PICKUP,j>) {
12        if (free) {
13            free = false;
14            asend(ACK, philo[j]);
15        } else {
16            queue.add(j);
17        }
18    } else {
19        if (msg == <PUTDOWN, j>) {
20            if (queue.isEmpty()) {
21                free = true;
22            } else {
23                k = queue.remove();
24                asend(ACK, philo[k]);
25            }
26        }
27    }
}
```

3.8.7 Message Passing - Produttori e consumatori

```
1 process Producer {
2     Object x;
3     while (true) {
4         x = produce();
5         ssend(x, PCmanager);
6     }
7 }
8
9 process Consumer{
10    Object x;
11    while (true) {
12        x = sreceive(PCmanager);
13        consume(x);
14    }
15 }
16
17 process PCmanager {
18    Object x;
19    while (true) {
20        x = sreceive(Producer);
21        ssend(x, Consumer);
22    }
23 }
```

3.9 Conclusioni

3.9.1 Riassunto

- **Semafori:** fondamentale primitiva di sincronizzazione, effettivamente offerta dai S.O. . Di livello troppo basso; facile commettere errori.
- **Monitor:** meccanismi integrati nei linguaggi di programmazione, pochi linguaggi di larga diffusione sono dotati di monitor; unica eccezione Java, con qualche distinguo.
- **Message passing:** da un certo punto di vista, il meccanismo più diffuso. Può essere poco efficiente (copia dati tra spazi di indirizzamento).

3.9.2 Potere espressivo

Definizione: Si dice che il paradigma di programmazione A è espressivo almeno quanto il paradigma di programmazione B (e si scrive $A \geq B$) quando è possibile esprimere ogni programma scritto con B mediante A.

Ovvero, quando è possibile scrivere una libreria che consenta di implementare le chiamate di un paradigma B esprimendole in termini di A si avrà $A \geq B$.

Si dice che due paradigmi hanno lo stesso potere espressivo se $A \geq B$ e $B \geq A$.

In vari punti di questi lucidi si mostrano delle relazioni tra i vari paradigmi di programmazione mediante funzioni di implementazione.

Si possono tracciare le seguenti classi di paradigmi:

- **Metodi a memoria condivisa:**
 - semafori, semafori binari, monitor hanno tutti lo stesso potere espressivo.
 - dekker e peterson, Test&Set necessitano di busy waiting.
- **Metodi a memoria privata:** message passing asincrono (ha maggiore potere espressivo) e message passing sincrono.

Capitolo 4

Risorse

4.1 Introduzione

Un sistema di elaborazione è composto da un insieme di risorse da assegnare ai processi presenti. I processi competono nell'accesso alle risorse.

Esempi di risorse: memoria, stampanti, processore, dischi interfaccia di rete, descrittori di processo...

4.1.1 Classi di risorse

Le risorse possono essere suddivise in classi, le risorse appartenenti alla stessa classe sono equivalenti.
Esempi: byte della memoria, stampanti dello stesso tipo, etc.

Le risorse di una classe vengono dette **istanze della classe**, il numero di risorse in una classe viene detto **molteplicità** del tipo di risorsa.

Un processo non può richiedere una specifica risorsa, ma solo una risorsa di una specifica classe. Una richiesta per una classe di risorse può essere soddisfatta da qualsiasi istanza di quel tipo.

Assegnazione delle risorse

Risorse ad assegnazione statica Avviene al momento della creazione del processo e rimane valida fino alla terminazione.

Esempi: descrittori di processi, aree di memoria (in alcuni casi)

Risorse ad assegnazione dinamica I processi richiedono le risorse durante la loro esistenza, le utilizzano una volta ottenute e le rilasciano quando non più necessarie.

Esempi: periferiche di I/O, aree di memoria (in alcuni casi)

Tipi di richieste

Richiesta singola Si riferisce a una singola risorsa di una classe definita, è il caso normale.

Richiesta multipla Si riferisce a una o più classi, e per ogni classe, ad una o più risorse, deve essere soddisfatta integralmente.

Richiesta bloccante Il processo richiedente si sospende se non ottiene immediatamente l'assegnazione, la richiesta rimane pendente e viene riconsiderata dalla funzione di gestione ad ogni rilascio.

Richiesta non bloccante La mancata assegnazione viene notificata al processo richiedente, senza provocare la sospensione.

Tipi di risorse

Risorse seriali (o con accesso mutuamente esclusivo) Una singola risorsa non può essere assegnata a più processi contemporaneamente.

Esempi: i processori, le sezioni critiche, le stampanti

Risorse non seriali *Esempio: file di sola lettura*

Definizione: Una risorsa si dice prerilasciabile se la funzione di gestione può sottrarla ad un processo prima che questo l'abbia effettivamente rilasciata.

Meccanismo di gestione: Il processo che subisce il prerilascio deve sospondersi, la risorsa prerilasciata sarà successivamente restituita al processo.

Una risorsa è prerilasciabile: se il suo stato non si modifica durante l'utilizzo oppure il suo stato può essere facilmente salvato e ripristinato.

Esempi: processore, blocchi o partizioni di memoria (nel caso di assegnazione dinamica)

4.1.2 Risorse non prerilasciabili

Definizione: La funzione di gestione non può sottrarre al processo al quale sono assegnate, sono non prerilasciabili le risorse il cui stato non può essere salvato e ripristinato.

Esempi: stampanti, classi di sezioni critiche, partizioni di memoria (nel caso di gestione statica)

4.2 Deadlock

Come abbiamo visto i deadlock impediscono ai processi di terminare correttamente, inoltre le risorse bloccate in deadlock non possono essere utilizzati da altri processi.

Vedremo ora le condizioni che necessarie affinché un deadlock si presenti e le tecniche che possono essere utilizzate per gestire questo problema.

4.2.1 Condizioni per avere deadlock

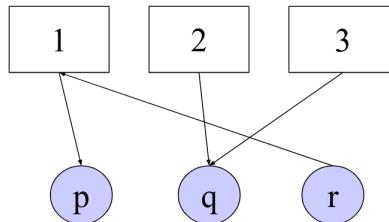
- **Mutua esclusione:** le risorse coinvolte devono essere seriali
- **Assenza di prerilascio:** le risorse coinvolte non possono essere prerilasciate, ovvero devono essere rilasciate volontariamente dai processi che le controllano
- **Richieste bloccanti (detta anche "hold and wait"):** le richieste devono essere bloccanti, e un processo che ha già ottenuto risorse può chiederne ancora
- **Attesa circolare:** esiste una sequenza di processi P_0, P_1, \dots, P_n , tali per cui P_0 attende una risorsa controllata da P_1 e P_1 attende una risorsa controllata da P_2, \dots , e P_n attende una risorsa controllata da P_0 .

L'insieme di queste condizioni è necessario e sufficiente. Devono valere tutte contemporaneamente affinché un deadlock si presenti nel sistema.

4.2.2 Grafo di Holt

Caratteristiche:

- è un grafo diretto: gli archi hanno una direzione.
- è un grafo bipartito: i nodi sono suddivisi in due sottoinsiemi e non esistono archi che collegano nodi dello stesso sottoinsieme. I sottoinsiemi sono risorse e processi.
- gli archi risorsa → processo indicano che la risorsa è assegnata al processo.
- gli archi processo → risorsa indicano che il processo ha richiesto la risorsa.

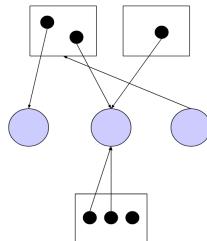


Grafo di Holt generale

Nel caso di classi contenenti più istanze di una risorsa, l'insieme delle risorse è partizionato in classi e gli archi di richiesta sono diretti alla classe e non alla singola risorsa.

Rappresentazione: i processi sono rappresentati da cerchi, le classi sono rappresentate come contenitori rettangolari e le risorse come punti all'interno delle classi.

Nota: non si rappresentano grafi di Holt con archi relativi a richieste che possono essere soddisfatte, se esiste almeno un'istanza libera della risorsa richiesta, la risorsa viene assegnata.



Alcuni autori preferiscono indicare numericamente: Sugli archi la molteplicità della richiesta (archi processo → classe) e la molteplicità dell'assegnazione (archi classe → processo). All'interno delle classi il numero di risorse non ancora assegnate.

4.3 Metodi di gestione dei deadlock

- **Deadlock detection and recovery:** permettere al sistema di entrare in stati di deadlock; utilizzare un algoritmo per rilevare questo stato ed eventualmente eseguire un'azione di recovery.
- **Deadlock prevention / avoidance:** impedire al sistema di entrare in uno stato di deadlock
- **Ostrich algorithm:** ignorare il problema del tutto.

4.3.1 Deadlock detection

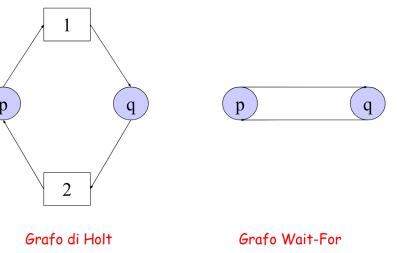
Descrizione mantenere aggiornato il grafo di Holt, registrando su di esso tutte le assegnazioni e le richieste di risorse. Utilizzare il grafo di Holt al fine di riconoscere gli stati di deadlock.

Problema: come riconoscere uno stato di deadlock?

Caso 1 - Una sola risorsa per classe

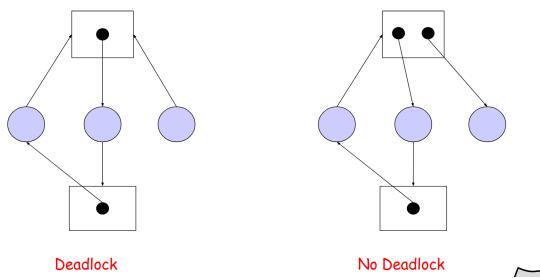
Teorema: se le risorse sono ad accesso mutualmente esclusivo, seriali e non prerilasciabili, lo stato è di deadlock se e solo se il grafo di Holt contiene un ciclo.

Dimostrazione: si utilizza una variante del grafo di Holt, detto grafo Wait-For, si ottiene un grafo wait-for eliminando i nodi di tipo risorsa e collassando gli archi appropriati, il grafo di Holt contiene un ciclo se e solo se il grafo Wait-for contiene un ciclo. Se il grafo Wait-for contiene un ciclo, abbiamo attesa circolare



Caso 2 - Più risorse per classe

La presenza di un ciclo nel caso di Holt non è condizione sufficiente per avere deadlock.



Riducibilità di un grafo di Holt

Definizione: un grafo di Holt si dice riducibile se esiste almeno un nodo processo con solo archi entranti

Riduzione: consiste nell'eliminare tutti gli archi di tale nodo e riassegnare le risorse ad altri processi.

Qual è la logica? eventualmente, un nodo che utilizza una risorsa prima o poi la rilascerà; a quel punto, la risorsa può essere riassegnata.

Deadlock detection con grafo di Holt

Teorema: se le risorse sono ad accesso mutualmente esclusivo, seriali e non prerilasciabili, lo stato non è di deadlock se e solo se il grafo di Holt è completamente riducibile. (*esiste una sequenza di passi di riduzione che elimina tutti gli archi del grafo*)

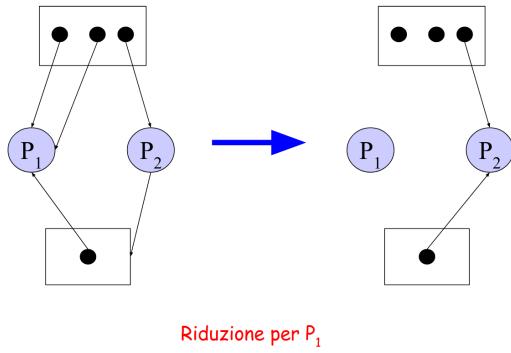


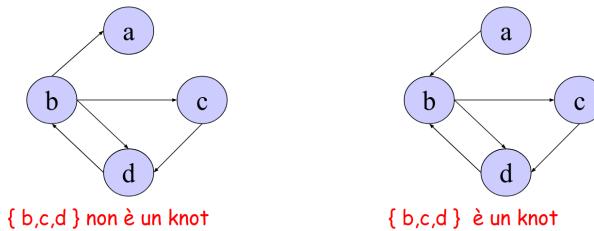
Figura 4.1: Esempio di riduzione

Deadlock detection - Knot

Definizione: dato un nodo n , l'insieme dei nodi raggiungibili da n viene detto insieme di raggiungibilità di n (scritto $R(n)$)

Un knot del grafo G è il massimo sottoinsieme (non banale) di nodi M tale che per ogni n in M , $R(n)=M$

In altre parole: partendo da un qualunque nodo di M , si possono raggiungere tutti i nodi di M e nessun nodo all'infuori di esso.



Teorema: dato un grafo di Holt con una sola richiesta sospesa per processo se le risorse sono ad accesso mutualmente esclusivo, seriali e non prerilasciabili, allora il grafo rappresenta uno stato di deadlock se e solo se esiste un knot.

4.3.2 Deadlock recovery

Dopo aver rilevato un deadlock bisogna risolvere la situazione.

La soluzione può essere:

- **Manuale:** l'operatore viene informato e eseguirà alcune azioni che permettano al sistema di proseguire.
- **Automatica:** il sistema operativo è dotato di meccanismi che permettono di risolvere in modo automatico la situazione, in base ad alcune politiche

Meccanismi per il deadlock recovery

- Terminazione totale: tutti i processi coinvolti vengono terminati.
- Terminazione parziale: viene eliminato un processo alla volta, fino a quando il deadlock non scompare.
- Preemption: una risorsa (o più di una, se necessario) viene sottratta ad uno dei processi coinvolti nel deadlock.
- Checkpoint/rollback: lo stato dei processi viene periodicamente salvato su disco (checkpoint). In caso di deadlock, si ripristina (rollback) uno o più processi ad uno stato precedente, fino a quando il deadlock non scompare.

Terminare processi può essere costoso, questi processi possono essere stati eseguiti per molto tempo; se terminati, dovranno ripartire da capo. Terminare processi può lasciare le risorse in uno stato inconsistente, se un viene terminato nel mezzo di una sezione critica.

Fare preemption non sempre è possibile, può richiedere interventi manuali.

4.3.3 Deadlock prevention e avoidance

Prevention: per evitare il deadlock si elimina una delle quattro condizioni del deadlock così il deadlock viene eliminato strutturalmente.

Avoidance: prima di assegnare una risorsa ad un processo, si controlla se l'operazione può portare al pericolo di deadlock e in caso, l'operazione viene ritardata.

Prevention

Attaccare la condizione di "Mutua esclusione", permettere la condivisione di risorse.

Problemi dello spooling: in generale, lo spooling non sempre è applicabile perché sposta il problema verso altre risorse.

Attaccare la condizione di "Richiesta bloccante" *Allocazione totale*, è possibile richiedere che un processo richiede tutte le risorse all'inizio della computazione. I problemi possono essere: non sempre l'insieme di richieste è noto fin dall'inizio e si riduce il parallelismo.

Attaccare la condizione di "Assenza di prerilascio" non sempre è possibile e può richiedere interventi manuali.

Attaccare la condizione di "Attesa Circolare" *Allocazione gerarchica*, quando alle classi di risorse vengono associati valori di priorità, ogni processo in ogni istante può allocare solamente risorse di priorità superiore a quelle che già possiede. Se un processo vuole allocare una risorsa a priorità inferiore, deve prima rilasciare tutte le risorse con priorità uguale o superiore a quella desiderata.

L'allocazione gerarchica e allocazione totale danno problemi. Prevengono il verificarsi di deadlock, ma sono altamente inefficienti perché nell'allocazione gerarchica: l'indisponibilità di una risorsa ad alta priorità ritarda processi che già detengono risorse ad alta priorità.

Nell'allocazione totale: anche se un processo ha necessità di risorse per poco tempo deve allocarla per tutta la propria esistenza.

Avoidance - Algoritmo del Banchiere

Avoidance è l'azione per cui prima di far qualcosa si verifica se porterà a deadlock.

L'algoritmo del banchiere è un algoritmo per evitare lo stallo sviluppato da Dijkstra (1965).

Il nome deriva dal metodo utilizzato da un ipotetico banchiere di provincia che gestisce un gruppo di clienti a cui ha concesso del credito; non tutti i clienti avranno bisogno dello stesso credito simultaneamente.

4.4 Algoritmo del Banchiere

4.4.1 Algoritmo del Banchiere Singola valuta

Descrizione: Un banchiere desidera condividere un capitale (fisso) con un numero (prefissato) di clienti.

Ogni cliente specifica in anticipo la sua necessità massima di denaro che ovviamente non deve superare il capitale del banchiere. I clienti fanno due tipi di transazioni: richieste di prestito e restituzioni.

Il denaro prestato ad ogni cliente non può mai eccedere la necessità massima specificata a priori.

Ogni cliente può fare richieste multiple, fino al massimo importo specificato, una volta che le richieste sono state accolte e il denaro è stato ottenuto deve garantire la restituzione in un tempo finito.

Metodo di funzionamento: Il banchiere deve essere in ogni istante in grado di soddisfare tutte le richieste dei clienti, o concedendo immediatamente il prestito oppure comunque facendo loro aspettare la disponibilità del denaro in un tempo finito.

Teniamo conto delle seguenti variabili:

- N - il numero dei clienti
- IC - capitale iniziale
- c_i - limite di credito del cliente i ($c_i \leq IC$)
- p_i : denaro prestato al cliente i ($p_i \leq c_i$)
- $n_i = c_i - p_i$ - credito residuo del cliente i
- $SC = IC - \sum_{i=1}^N p_i$ - saldo di cassa

Stato SAFE Le richieste che ogni cliente i può fare possono essere soddisfatte dalle risorse attualmente disponibili più tutte le risorse detenute dai processi j soddisfatti precedentemente ($j < i$).

Sia s una permutazione dei valori $1\dots N$. *Esempio, con $N=4$ e $s=(1, 3, 4, 2)$*

Indichiamo con $s(i)$ l' i -esima posizione della sequenza.

Si calcoli il vettore avail come segue:

- $avail[1] = SC$
- $avail[j + 1] = avail[j] + p_{s(j)}$, con $j=1\dots N-1$

Uno stato del sistema si dice safe se vale la seguente condizione: $n_{s(j)} \leq avail[j]$, con $j=1\dots N$

Lo stato UNSAFE è condizione necessaria ma non sufficiente per non avere deadlock. Un sistema in uno stato UNSAFE può evolvere senza procurare alcun deadlock.

Esempio - stato SAFE

Capitale Iniziale	IC	150
Saldo di cassa	SC	150

Cliente	MAX	Prestito Attuale	Credito residuo
i	c_i	p_i	n_i
1	100	0	100
2	20	0	20
3	30	0	30
4	50	0	50
5	70	0	70

Figura 4.2: Situazione iniziale

Regola pratica (per il banchiere a singola valuta): lo stato SAFE può essere verificato usando la sequenza che ordina in modo crescente i valori di n_i (credito residuo di i).

Sono sicuri che concedo ad ogni passo il minimo possibile per poi recuperarlo e aumentare la disponibilità di prestito per il prossimo cliente.

Capitale Iniziale	IC	150
Saldo di cassa	SC	0

Cliente	MAX	Prestito Attuale	Credito residuo
i	c _i	p _i	n _i
1	100	70	30
2	20	10	10
3	30	30	0
4	50	10	40
5	70	30	40

Immaginiamo di essere arrivati a questo stato di prestiti.

Per valutare se siamo in uno stato safe: dobbiamo considerare il caso pessimo delle richieste, i.e., massime richieste possibili (i.e., residui n_i)

Ordiniamo i processi sulla base dei loro n_i

Capitale Iniziale	IC	150
Saldo di cassa	SC	0

Cliente	MAX	Prestito Attuale	Credito residuo
i	c _i	p _i	n _i
1	100	70	30
2	20	10	10
3	30	30	0
4	50	10	40
5	70	30	40

la sequenza 3,2,1,4,5 consente il soddisfacimento di tutte le richieste

Capitale Iniziale	IC	150
Saldo di cassa	SC	0

Cliente	MAX	Prestito Attuale	Credito residuo
i	c _i	p _i	n _i

avail[i]

la sequenza 3,2,1,4,5 consente il soddisfacimento di tutte le richieste

3	30	30	0
2	20	10	10
1	100	70	30
4	50	10	40
5	70	30	40

0
30
40
110
120

Esempio - stato UNSAFE

Capitale Iniziale	IC	150
Saldo di cassa	SC	10

Cliente	MAX	Prestito Attuale	Credito residuo
i	c _i	p _i	n _i
1	100	65	35
2	20	10	10
3	30	5	25
4	50	15	35
5	70	35	35

Capitale Iniziale	IC	150
Saldo di cassa	SC	10

Cliente	MAX	Prestito Attuale	Credito residuo	avail[i]
i	c _i	p _i	n _i	avail[i]

Capitale Iniziale	IC	150
Saldo di cassa	SC	10

se esistesse una sequenza,
questa sarebbe 2,3,1,5,4
in evidenza e corsivo sono indicati i
casi nei quali la condizione di
safety fallisce

2	20	10	10	10
3	30	5	25	20
1	100	65	35	25
5	70	35	35	100
4	50	15	35	135

Capitale Iniziale	IC	150
Saldo di cassa	SC	0

Cliente	MAX	Prestito Attuale	Credito residuo	avail[i]
i	c _i	p _i	n _i	avail[i]

Capitale Iniziale	IC	150
Saldo di cassa	SC	0

UNSAFE non implica deadlock

se il cliente 5 restituisce il suo
prestito di 35 euro la
situazione ritorna SAFE

2	20	10	10	45
3	30	5	25	55
1	100	75	35	60
4	50	15	35	135
5	70	0	70	150

Conclusion Algoritmo Banchiere singola valuta

La similitudine fra banchieri e sistemi operativi ora è chiara: il denaro sono le risorse, il sistema le deve allocare ai processi senza che si possa verificare deadlock.

Le definizioni viste fino a questo punto riguardano il caso teorico elementare di un sistema avente un'unica classe di risorse.

4.4.2 Algoritmo del Banchiere Multivaluta

L'estensione del problema del banchiere singola valuta dove si ipotizza che il banchiere debba fare prestiti usando valute diverse (euro, dollari, yen, etc.) Le diverse valute rappresentano diverse classi di risorse.

Teniamo conto le seguenti variabili:

- N - il numero dei clienti
- IC_k - capitale iniziale della valuta k
- $c_{i,k}$ - limite di credito in valuta k del cliente i ($c_{i,k} < IC_k$)
- $p_{i,k}$ - denaro prestato in valuta k al cliente i ($p_{i,k} \leq c_{i,k}$)
- $n_{i,k} = c_{i,k} - p_{i,k}$ - credito residuo in valuta k del cliente i
- $SC_k = IC_k - \sum_{i=1}^N p_{i,k}$ - saldo di cassa in valuta k

Stato SAFE Sia s una permutazione dei valori $1\dots N$. Esempio, con $N=4$ e $s=(1, 3, 4, 2)$. Indichiamo con $s(i)$ l' i -esima posizione della sequenza.

Si calcoli il vettore $avail_k$ come segue:

- $avail_k[1] = SC_k$
- $avail_k[j + 1] = avail_k[j] + p_{s(j)}$, con $j=1\dots N-1$

Uno stato del sistema si dice safe se vale la seguente condizione: $n_{s(j),k} \leq avail_k[j]$, con $j=1\dots N$

Problema La regola di ordinare i processi secondo i valori di n non è applicabile, l'ordine può essere in generale diverso fra le diverse valute gestite dal banchiere.

Soluzione Si può creare la sequenza procedendo passo passo aggiungendo un processo a caso fra quelli completamente soddisfacibili. Ovvero, al passo j si sceglie quelli per cui: $n_{s(j),k} \leq avail_k[j]$

Esempio - Banchiere Multivaluta

Prestiti Risorse $p_{i,k}$				Massimo Richieste $C_{i,k}$				Saldo in Cassa $SC_{i,k}$			
Cliente	A	B	C	Cliente	A	B	C	Cliente	A	B	C
C1	0	1	0	C1	7	5	3				
C2	2	0	0	C2	3	2	2				
C3	3	0	2	C3	9	0	2				
C4	2	1	1	C4	2	2	2				
C5	0	0	2	C5	4	3	3				

Massimo Richieste $C_{i,k}$				Prestiti Risorse $p_{i,k}$				Credito Residuo $n_{i,k}$			
Cliente	A	B	C	Cliente	A	B	C	Cliente	A	B	C
C1	7	5	3	C1	0	1	0	C1	7	4	3
C2	3	2	2	C2	2	0	0	C2	1	2	2
C3	9	0	2	C3	3	0	2	C3	6	0	0
C4	2	2	2	C4	2	1	1	C4	0	1	1
C5	4	3	3	C5	0	0	2	C5	4	3	1

Prestiti Risorse $p_{i,k}$				Credito Residuo $n_{i,k}$				Disponibilità $avail = SC_{i,k}$			
Cliente	A	B	C	Cliente	A	B	C	Cliente	A	B	C
C1	0	1	0	C1	7	4	3	C1	3	3	2
C2	2	0	0	C2	1	2	2				
C3	3	0	2	C3	6	0	0				
C4	2	1	1	C4	0	1	1				
C5	0	0	2	C5	4	3	1				

Prestiti Risorse $p_{i,k}$			Credito Residuo $n_{i,k}$			Disponibilità $avail = SC_{i,k}$		
Cliente	A	B	C	A	B	C	A	B
C1	0	1	0	7	4	3	≤	3 3 2
C2	2	0	0	1	2	2		
C3	3	0	2	6	0	0		
C4	2	1	1	0	1	1		
C5	0	0	2	4	3	1		

Sequenza Clienti = < C2 >

Prestiti Risorse $p_{i,k}$			Credito Residuo $n_{i,k}$			Disponibilità $avail$		
Cliente	A	B	C	A	B	C	A	B
C1	0	1	0	7	4	3	>	5 3 2
C2	0	0	0	1	2	2		
C3	3	0	2	6	0	0		
C4	2	1	1	0	1	1		
C5	0	0	2	4	3	1		

Sequenza Clienti = < C2 >

Prestiti Risorse $p_{i,k}$			Credito Residuo $n_{i,k}$			Disponibilità $avail$		
Cliente	A	B	C	A	B	C	A	B
C1	0	1	0	7	4	3	≤	5 3 2
C2	0	0	0	1	2	2		
C3	3	0	2	6	0	0		
C4	2	1	1	0	1	1		
C5	0	0	2	4	3	1		

Sequenza Clienti = < C2, C4 >

Prestiti Risorse $p_{i,k}$			Credito Residuo $n_{i,k}$			Disponibilità $avail$		
Cliente	A	B	C	A	B	C	A	B
C1	0	1	0	7	4	3	≤	7 4 3
C2	0	0	0	1	2	2		
C3	3	0	2	6	0	0		
C4	0	0	0	0	1	1		
C5	0	0	2	4	3	1		

Sequenza Clienti = < C2, C4, C5 >

Prestiti Risorse $p_{i,k}$				Credito Residuo $n_{i,k}$				Disponibilità avail		
Cliente	A	B	C	Cliente	A	B	C	A	B	C
C1	0	1	0	C1	7	4	3	\leq		
C2	0	0	0	C2	1	2	2			
C3	3	0	2	C3	6	0	0			
C4	0	0	0	C4	0	1	1			
C5	0	0	0	C5	4	3	1			

Sequenza Clienti = < C2, C4, C5, C1 >

Prestiti Risorse $p_{i,k}$				Credito Residuo $n_{i,k}$				Disponibilità avail		
Cliente	A	B	C	Cliente	A	B	C	A	B	C
C1	0	0	0	C1	7	4	3	\leq		
C2	0	0	0	C2	1	2	2			
C3	3	0	2	C3	6	0	0			
C4	0	0	0	C4	0	1	1			
C5	0	0	0	C5	4	3	1			

Sequenza Clienti = < C2, C4, C5, C1, C3 >

Prestiti Risorse $p_{i,k}$				Credito Residuo $n_{i,k}$				Disponibilità avail		
Cliente	A	B	C	Cliente	A	B	C	A	B	C
C1	0	0	0	C1	7	4	3	\leq		
C2	0	0	0	C2	1	2	2			
C3	0	0	0	C3	6	0	0			
C4	0	0	0	C4	0	1	1			
C5	0	0	0	C5	4	3	1			

Sequenza Clienti = < C2, C4, C5, C1, C3 >

Stato SAFE

Capitolo 5

Memoria

5.1 Binding, loading, linking

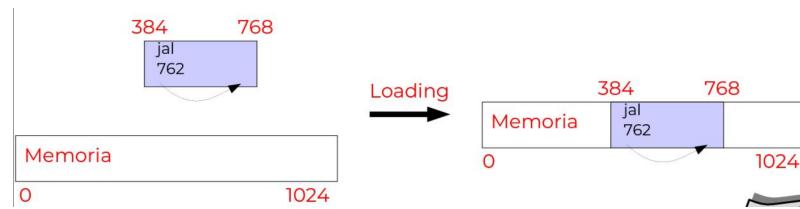
La parte del sistema operativo che gestisce la memoria principale si chiama **memory manager** che in alcuni casi può gestire anche parte della memoria secondaria, al fine di emulare memoria principale. I compiti di un memory manager sono: tenere traccia della memoria libera e occupata e allocare memoria ai processi e deallokarla quando non più necessaria.

5.1.1 Binding

Definizione: con il termine binding si indica l'associazione di indirizzi di memoria ai dati e alle istruzioni di un programma. Il binding può avvenire durante la compilazione, il caricamento o l'esecuzione.

Binding durante la compilazione

Gli indirizzi vengono calcolati al momento della compilazione e resteranno gli stessi ad ogni esecuzione del programma, il codice generato viene detto codice **assoluto**. *Esempi: codice per microcontrollori, per il kernel, file COM in MS-DOS*

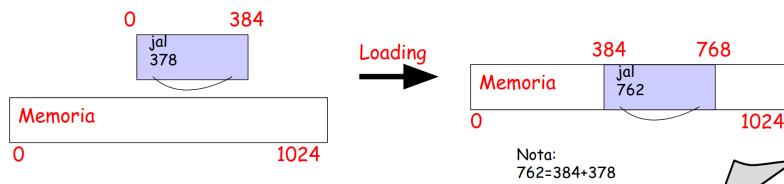


Vantaggi: non richiede hardware speciale, è semplice, molto veloce.

Svantaggi: non funziona con la multiprogrammazione.

Binding durante il caricamento

Il codice generato dal compilatore non contiene indirizzi assoluti ma relativi (al PC oppure ad un indirizzo base), questo tipo di codice viene detto **rilocabile**. Durante il caricamento il loader si preoccupa di aggiornare tutti i riferimenti agli indirizzi di memoria coerentemente al punto iniziale di caricamento.

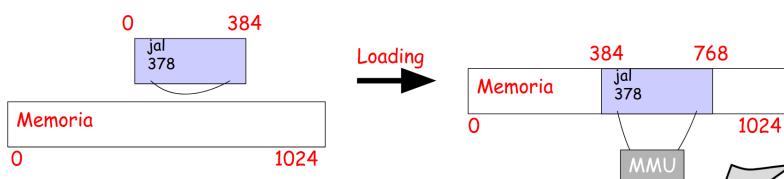


Vantaggi: permette di gestire multiprogrammazione, non richiede uso di hardware particolare.

Svantaggi: richiede una traduzione degli indirizzi da parte del loader, e quindi formati particolari dei file eseguibili.

Binding durante l'esecuzione

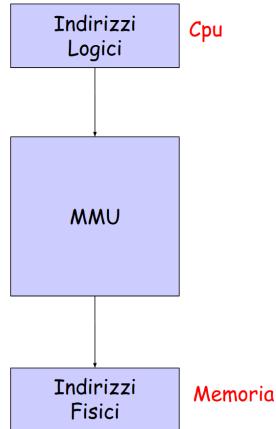
L'individuazione dell'indirizzo di memoria effettivo viene effettuata durante l'esecuzione da un componente hardware apposito: la memory management unit (**MMU**).



Indirizzi logici e indirizzi fisici

Spazio di indirizzamento logico ogni processo è associato ad uno spazio di indirizzamento logico, gli indirizzi usati in un processo sono indirizzi logici, ovvero riferimenti a questo spazio di indirizzamento.

Spazio di indirizzamento fisico ad ogni indirizzo logico corrisponde un indirizzo fisico, la MMU opera come una funzione di traduzione da indirizzi logici a indirizzi fisici.



5.1.2 Loading Dinamico

Il loading dinamico consente di poter caricare alcune routine di libreria solo quando vengono richiamate. Tutte le routine a caricamento dinamico risiedono su un disco (codice rilocabile), quando servono vengono caricate, le routine poco utili (*e.g., casi di errore rari...*) non vengono caricate in memoria al caricamento dell'applicazione.

Nota: spetta al programmatore sfruttare questa possibilità, dato che il sistema operativo fornisce semplicemente una libreria che implementa le funzioni di caricamento dinamico.

5.1.3 Linking statico

Se il linker collega e risolve tutti i riferimenti dei programmi, le routine di libreria vengono copiate in ogni programma che le usa (*e.g. printf in tutti i programmi C*).

5.1.4 Linking dinamico

E' possibile posticipare il linking delle routine di libreria al momento del primo riferimento durante l'esecuzione. Consente di avere eseguibili più compatti e le librerie vengono implementate come codice reentrant, ovvero esiste una sola istanza della libreria in memoria e tutti i programmi eseguono il codice di questa istanza.

Vantaggi: risparmio di memoria, consente l'aggiornamento automatico delle versioni delle librerie che vengono caricate alla successiva attivazione dei programmi.

Svantaggi: può causare problemi di 'versioning', ovvero conflitti che si verificano quando diverse applicazioni richiedono versioni diverse della stessa libreria condivisa o quando l'aggiornamento di una libreria la rende incompatibile con alcune applicazioni che la utilizzano.

5.2 Allocazione

5.2.1 Definizioni

E' una delle funzioni principali del gestore di memoria. Consiste nel reperire ed assegnare uno spazio di memoria fisica a un programma che viene attivato, oppure per soddisfare ulteriori richieste effettuate dai programmi durante la loro esecuzione.

Allocazione contigua tutto lo spazio assegnato ad un programma deve essere formato da celle consecutive.

Allocazione non contigua è possibile assegnare a un programma aree di memorie separate.

Nota: la MMU deve essere in grado di gestire la conversione degli indirizzi in modo coerente. Esempio: la MMU basata su rilocazione gestisce solo allocazione contigua.

Allocazione statica: un programma deve mantenere la propria area di memoria dal caricamento alla terminazione. Non è possibile rilocare il programma durante l'esecuzione.

Allocazione dinamica: durante l'esecuzione, un programma può essere spostato all'interno della memoria.

5.2.2 Allocazione a partizioni fisse

Descrizione: la memoria disponibile (quella non occupata dal s.o.) viene suddivisa in partizioni, ogni processo viene caricato in una delle partizioni libere che ha dimensione sufficiente a contenerlo.

Caratteristiche: statica e contigua. Molto semplice. Ma produce spreco di memoria e ha un grado di parallelismo limitato dal numero di partizioni.



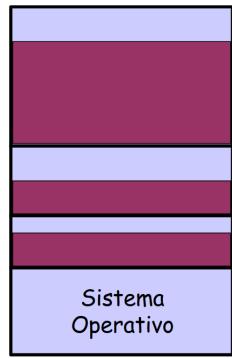
E' possibile utilizzare una coda per partizione, oppure una coda comune per tutte le partizioni. Esiste una sola partizione, dove viene caricato un unico programma utente.

Frammentazione interna

Nell'allocazione a partizione fisso se un processo occupa una dimensione inferiore a quella della partizione che lo contiene, lo spazio non utilizzato è sprecato.

La presenza di spazio inutilizzato all'interno di un'unità di allocazione si chiama frammentazione interna.

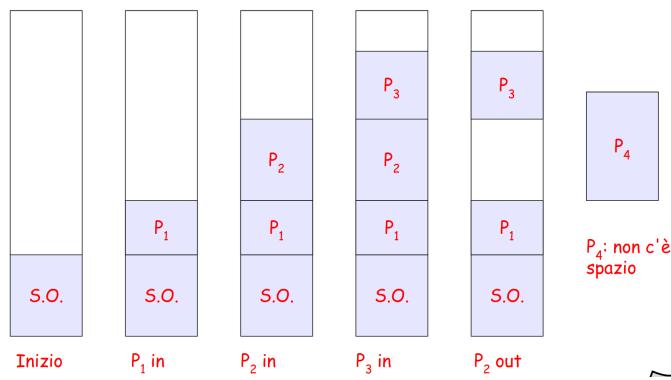
Nota: il fenomeno della frammentazione interna non è limitata all'allocazione a partizioni fisse, ma è generale a tutti gli approcci in cui è possibile allocare più memoria di quanto richiesto (per motivi di organizzazione).



5.2.3 Allocazione a partizioni dinamiche

Descrizione: la memoria disponibile viene assegnata ai processi che ne fanno richiesta. Nella memoria possono essere presenti diverse zone inutilizzate per effetto della terminazione di processi, oppure per non completo utilizzo dell'area disponibile da parte dei processi attivi.

Caratteristiche: statica e contigua. Esistono diverse politiche per la scelta dell'area da utilizzare.



Frammentazione esterna

Dopo un certo numero di allocazioni e deallocazioni di memoria dovute all'attivazione e alla terminazione dei processi lo spazio libero appare suddiviso in piccole aree questo fenomeno prende il nome di frammentazione esterna.

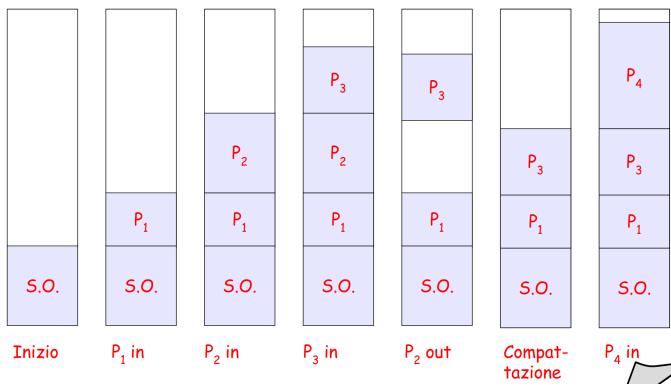
Nota: la frammentazione interna dipende dall'uso di unità di allocazione di dimensione diversa da quella richiesta.

5.2.4 Compattazione

Se è possibile rilocare i programmi durante la loro esecuzione, è allora possibile procedere alla compattazione della memoria. Compattare la memoria significa spostare in memoria tutti i programmi in modo da riumare tutte le aree inutilizzate, è un'operazione volta a risolvere il problema della frammentazione esterna.

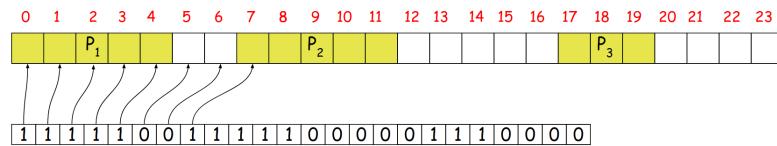
Purtroppo è un'operazione molto onerosa in quanto occorre copiare (fisicamente) in memoria grandi quantità di dati. Non può essere utilizzata in sistemi interattivi dato che i processi devono essere fermi durante la compattazione.

Quando la memoria è assegnata dinamicamente abbiamo bisogno di una struttura dati per mantenere informazioni sulle zone libere e sulle zone occupate, le strutture dati possibili: mappe di bit, liste con puntatori, ...



Mappa di bit

La memoria viene suddivisa in unità di allocazione dove ad ogni unità di allocazione corrisponde un bit in una bitmap. Le unità libere sono associate ad un bit di valore 0, le unità occupate sono associate ad un bit di valore 1.



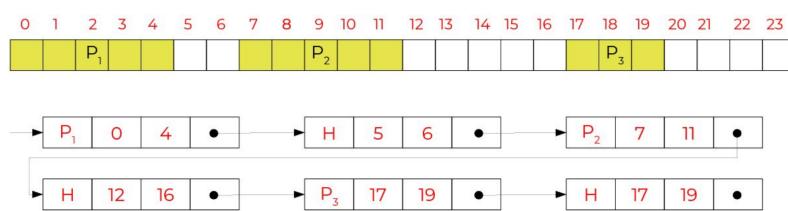
La dimensione dell'unità di allocazione è un parametro importante dell'algoritmo, esiste un trade-off fra dimensione della bitmap e frammentazione interna.

Vantaggi: la struttura dati ha una dimensione fissa e calcolabile a priori.

Svantaggi: per individuare uno spazio di memoria di dimensione k unità, è necessario cercare una sequenza di k bit 0 consecutivi, in generale, tale operazione è $O(m)$, dove m rappresenta il numero di unità di allocazione.

Liste di puntatori

Si mantiene una lista dei blocchi allocati e liberi di memoria, ogni elemento della lista specifica, se si tratta di un processo (P) o di un blocco libero (hole, H), la dimensione (inizio/fine) del segmento.

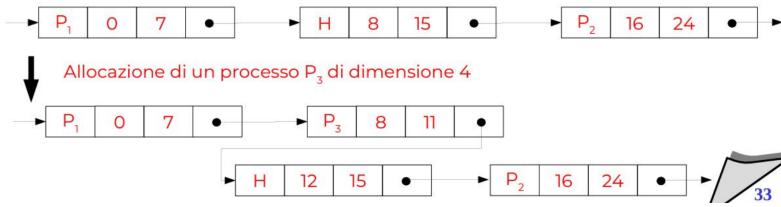


Un blocco libero viene selezionato e suddiviso in due parti:

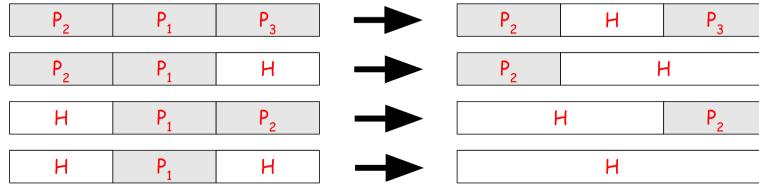
1. un blocco processo della dimensione desiderata
2. un blocco libero con quanto rimane del blocco iniziale

Allocazione di memoria Se la dimensione del processo è uguale a quella del blocco scelto, si crea solo un nuovo blocco processo.

Deallocazione di memoria A seconda dei blocchi vicini, lo spazio liberato può creare un nuovo blocco libero, oppure essere accorpato ai blocchi vicini. L'operazione può essere fatta in tempo $O(1)$.



Rimozione P₁, quattro casi possibili:



Selezione blocco libero L'operazione di selezione di un blocco libero è concettualmente indipendente dalla struttura dati.

- **First Fit:** scorre la lista dei blocchi liberi fino a quando non trova il primo segmento vuoto grande abbastanza da contenere il processo.
- **Next Fit:** come First Fit, ma invece di ripartire sempre dall'inizio, parte dal punto dove si era fermato all'ultima allocazione (è stato progettato per evitare di frammentare continuamente l'inizio della memoria ma sorprendentemente, ha performance peggiori di First Fit).
- **Best Fit:** seleziona il più piccolo fra i blocchi liberi presenti in memoria. Più lento di First Fit, in quanto richiede di esaminare tutti i blocchi liberi presenti in memoria, genera più frammentazione di First Fit, in quanto tende a riempire la memoria di blocchi liberi troppo piccoli.
- **Worst fit:** seleziona il più grande fra i blocchi liberi presenti in memoria. Proposto per evitare i problemi di frammentazione di First/Best Fit, rende difficile l'allocazione di processi di grosse dimensioni.

Nella struttura proposta liste di puntatori, il costo della deallocazione è O(1). Possibile ottimizzare il costo di allocazione mantenendo una lista di blocchi liberi separata o eventualmente, ordinando tale lista per dimensione.

5.3 Paginazione

5.3.1 Introduzione

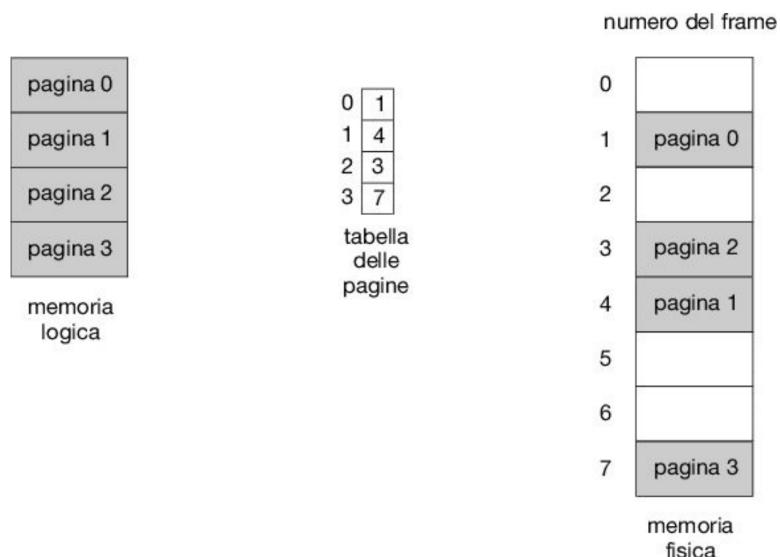
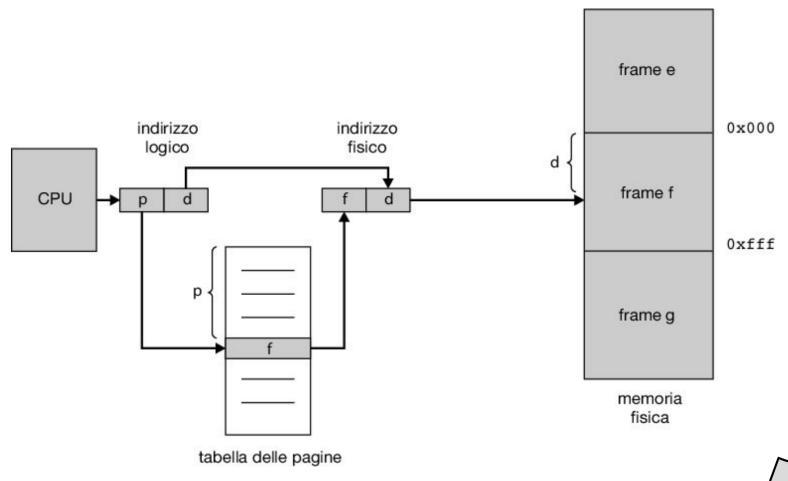
I meccanismi visti (partizioni fisse, partizioni dinamiche) non sono efficienti nell'uso della memoria perché causano le frammentazioni.

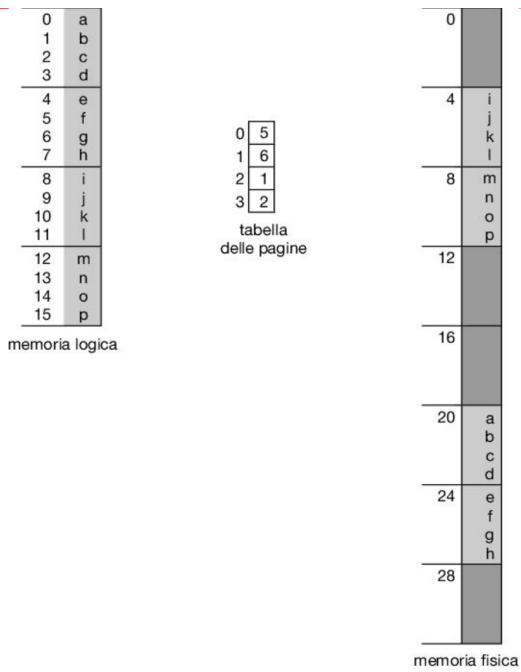
La paginazione è l'approccio contemporaneo che riduce il fenomeno di frammentazione interna ed elimina il fenomeno della frammentazione esterna.

Lo spazio di indirizzamento logico di un processo viene suddiviso in un insieme di blocchi di dimensione fissa chiamati **pagine**.

La memoria fisica viene suddivisa in un insieme di blocchi della stessa dimensione delle pagine, chiamati **frame**.

Quando un processo viene allocato in memoria: vengono reperiti ovunque in memoria un numero sufficiente di frame per contenere le pagine del processo.





Dimensione delle pagine La dimensione delle pagine deve essere una potenza di due, per semplificare la trasformazione da indirizzi logici a indirizzi fisici, la scelta della dimensione deriva da un trade-off poichè con pagine troppo piccole, la tabella delle pagine cresce di dimensioni mentre con pagine troppo grandi, lo spazio di memoria perso per frammentazione interna può essere considerevole. (*valori tipici: 1KB, 2KB, 4KB*)

Implementazione della page table

Dove mettere la tabella delle pagine?

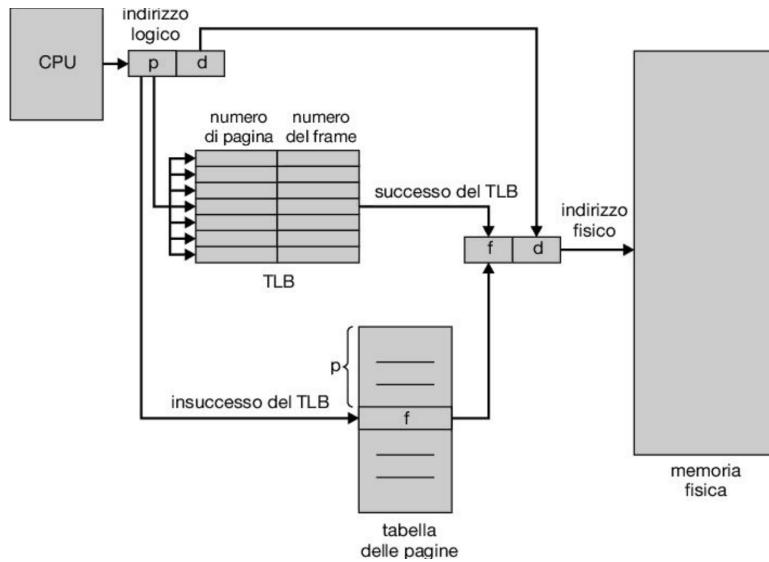
Soluzione 1: registri dedicati. La tabella può essere contenuta in un insieme di registri ad alta velocità all'interno del modulo MMU (o della CPU), però è una soluzione troppo costosa.

Soluzione 2: totalmente in memoria. Il problema è che il numero di accessi in memoria verrebbe raddoppiato; ad ogni riferimento, bisognerebbe prima accedere alla tabella delle pagine, poi al dato.

5.3.2 Translation lookaside buffer (TLB)

Descrizione: un **TLB** è costituito da un insieme di registri associativi ad alta velocità. Ogni registro è suddiviso in due parti, una chiave e un valore. Quando avviene l'operazione di **lookup**: viene richiesta la ricerca di una chiave che viene confrontata simultaneamente con tutte le chiavi presenti nel buffer.

- se la chiave è presente (**TLB hit**), si ritorna il valore corrispondente.
- se la chiave non è presente (**TLB miss**), si utilizza la tabella in memoria.



5.4 Segmentazione

5.4.1 Introduzione e confronti

In un sistema basato su segmentazione, uno spazio di indirizzamento logico è dato da un insieme di segmenti.

Un **segmento** è un'area di memoria (logicamente continua) contenente elementi tra loro affini dove ogni segmento è caratterizzato da un nome (normalmente un indice) e da una lunghezza.

Ogni riferimento di memoria è dato da una coppia (*nome segmento, offset*).

Segmentazione vs Paginazione

Paginazione:

- la divisione in pagine è automatica.
- le pagine hanno dimensione fissa.
- le pagine possono contenere informazioni disomogenee (*ad es. sia codice sia dati*).
- una pagina ha un indirizzo.
- dimensione tipica della pagina: 1-4 KB.

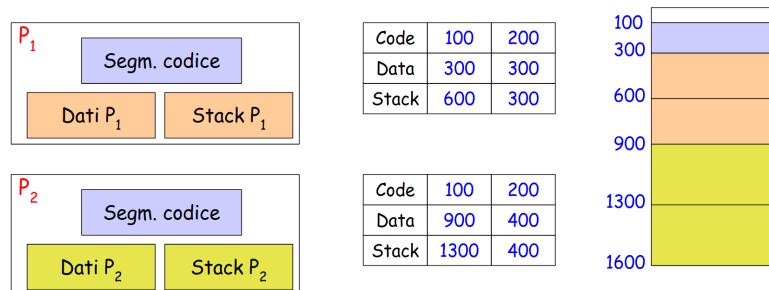
Segmentazione:

- la divisione in segmenti spetta al programmatore.
- i segmenti hanno dimensione variabile.
- un segmento contiene informazioni omogenee per tipo di accesso e permessi di condivisione.
- un segmento ha un nome.
- dimensione tipica di un segmento: 64KB - 1MB.

Segmentazione - condivisione

La segmentazione consente la condivisione di codice e dati.

Esempio: editor condiviso



Segmentazione - frammentazione

Allocare segmenti di dimensione variabile è del tutto equivalente al problema di allocare in modo contiguo la memoria dei processi. E' possibile utilizzare tecniche di allocazione dinamica (e.g., First Fit), compattazione, ma così torniamo ai problemi precedenti.

Segmentazione - paginazione

E' possibile utilizzare il metodo della paginazione combinato al metodo della segmentazione, ogni segmento viene suddiviso in pagine che vengono allocate in frame liberi della memoria (non necessariamente contigui).

Requisiti hardware: la MMU deve avere sia il supporto per la segmentazione sia il supporto per la paginazione.

Benefici: sia quelli della segmentazione (condivisione, protezione) e sia quelli della paginazione (no frammentazione esterna).

5.5 Memoria virtuale

5.5.1 Introduzione

Definizione: è la tecnica che permette l'esecuzione di processi che non sono completamente in memoria.

Permette di eseguire in concorrenza processi che nel loro complesso (o anche singolarmente) hanno necessità di memoria maggiore di quella disponibile. La memoria virtuale può diminuire le prestazioni di un sistema se implementata (e usata) nel modo sbagliato.

Le istruzioni da eseguire e i dati su cui operano devono essere in memoria, ma non è necessario che l'intero spazio di indirizzamento logico di un processo sia in memoria. I processi non utilizzano tutto il loro spazio di indirizzamento contemporaneamente.

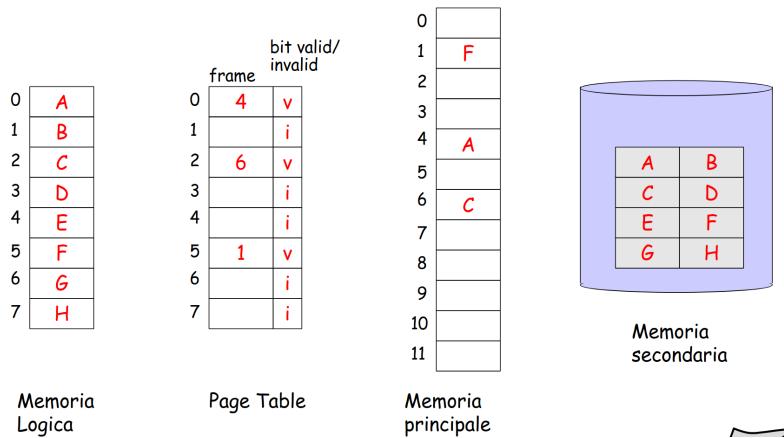
Implementazione: ogni processo ha accesso ad uno **spazio di indirizzamento virtuale** che può essere più grande di quello fisico.

Gli indirizzi virtuali possono essere mappati su indirizzi fisici della memoria principale oppure, possono essere mappati su memoria secondaria, in questo caso: i dati associati vengono trasferiti in memoria principale, se la memoria è piena, si sposta in memoria secondaria i dati contenuti in memoria principale che sono considerati meno utili.

Si utilizza la tecnica della paginazione a richiesta (**demand paging**), ammettendo però che alcune pagine possano essere in memoria secondaria.

Nella tabella delle pagine si utilizza un bit (v, per valid) che indica se la pagina è presente in memoria centrale oppure no.

Quando un processo tenta di accedere ad un pagina non in memoria il processore genera un trap (**page fault**) e un componente del s.o. (**pager**) si occupa di caricare la pagina mancante in memoria, e di aggiornare di conseguenza la tabella delle pagine.



5.5.2 Pager/swapper

Swap: con questo termine si intende l'azione di copiare l'intera area di memoria usata da un processo. Era una tecnica utilizzata nel passato quando demand paging non esisteva.

- dalla memoria secondaria alla memoria principale (**swap-in**)
- dalla memoria principale alla memoria secondaria (**swap-out**).

La paginazione su richiesta (demand paging) può essere vista come una tecnica di swap di tipo lazy, viene caricato solo ciò che serve.

Per questo motivo alcuni sistemi operativi indicano il pager con il nome di **swapper** ma è da considerarsi una terminologia obsoleta.

Noi utilizziamo il termine **swap area** per indicare l'area del disco utilizzata per ospitare le pagine in memoria secondaria.

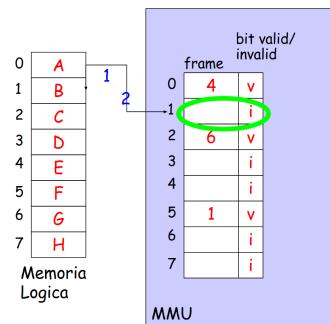
5.5.3 Gestione dei page fault

Vediamo passo a passo come viene gestito il page fault:

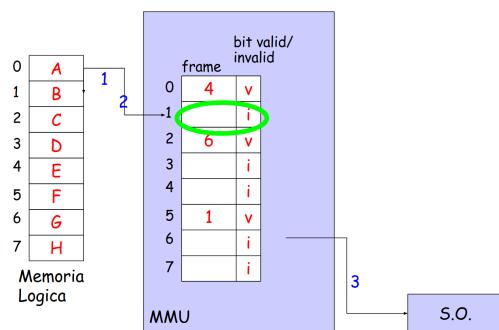
- Supponiamo che il codice in pagina 0 faccia riferimento alla pagina 1.



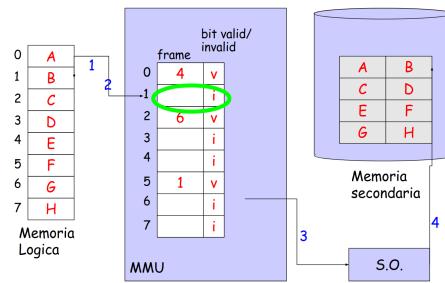
- La MMU scopre che la pagina 1 non è in memoria principale.



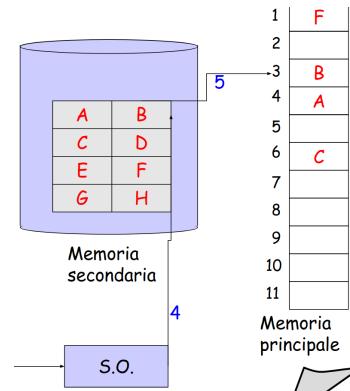
- Viene generato un trap "page fault", che viene catturato dal s.o.



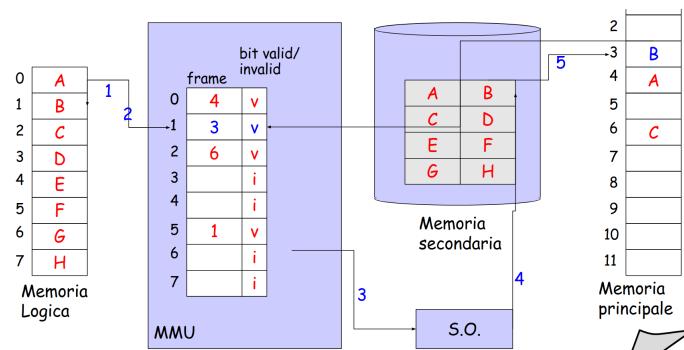
4. Il s.o. cerca in memoria secondaria la pagina da caricare.



5. Il s.o. carica la memoria principale con il contenuto della pagina.



6. Il s.o. aggiorna la page table in modo opportuno e riavvia l'esecuzione.



In mancanza di frame liberi occorre "liberarne" uno, la pagina da rimpiazzare deve essere la meno "utile". Esistono **algoritmi di sostituzione o rimpiazzamento** per questo compito.

5.5.4 Algoritmo del meccanismo di demand paging

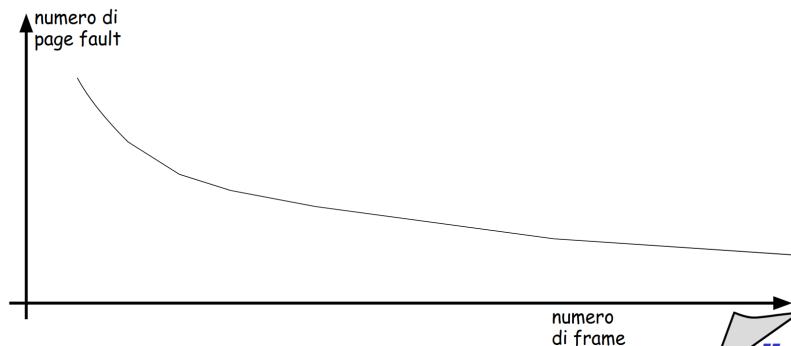
- Individua la pagina in memoria secondaria
- Individua un frame libero
- Se non esiste un frame libero
 - richiama algoritmo di rimpiazzamento
 - aggiorna la tabella delle pagine (invalida pagina "vittima")
 - se la pagina "vittima" è stata variata, scrive la pagina sul disco
 - aggiorna la tabella dei frame (frame libero)
- Aggiorna la tabella dei frame (frame occupato)
- Leggi la pagina da disco (quella che ha provocato il fault)
- Aggiorna la tabella delle pagine
- Riattiva il processo

5.5.5 Algoritmi di rimpiazzamento

L'algoritmo di rimpiazzamento ha come obiettivo minimizzare il numero di page fault. Gli algoritmi vengono valutati esaminando come si comportano quando applicati ad una stringa di riferimenti in memoria.

Stringhe di riferimenti possono essere generate esaminando il funzionamento di programmi reali o con un generatore di numeri random. La stringa di riferimenti può essere limitata ai numeri di pagina, in quanto non siamo interessati agli offset.

Andamento dei page fault in funzione del numero di frame. Ci si aspetta un grafico monotono decrescente, ma non sempre è così.



Algoritmo FIFO

Descrizione: Quando c'è necessità di liberare un frame viene individuato come "vittima" il frame che per primo fu caricato in memoria.

Vantaggi: semplice, non richiede particolari supporti hardware.

Svantaggi: vengono talvolta scaricate pagine che sono sempre utilizzate.

Esempio 1 numero di frame in memoria: 3 , numero di page fault: 15 (su 20 accessi in memoria)

tempo	1	20
stringa riferim.	7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1	
pagine in memoria.	7 7 7 2 2 2 2 4 4 4 0 0 0 0 0 0 7 7 7 0 0 0 0 3 3 3 2 2 2 2 1 1 1 1 1 0 0 1 1 1 1 0 0 0 3 3 3 3 2 2 2 2 2 1	

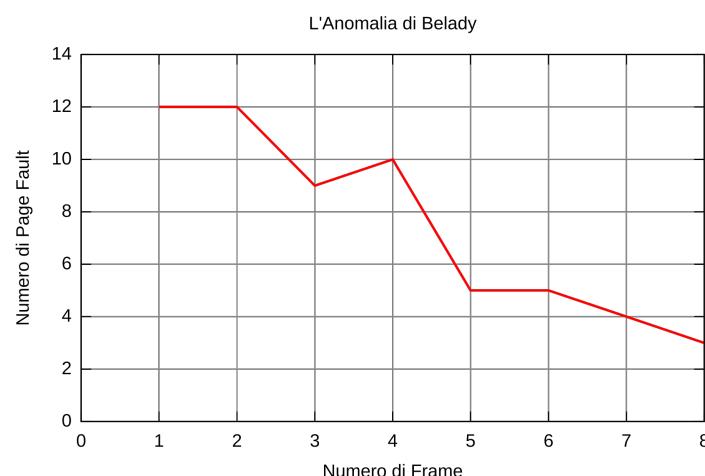
Esempio 2 numero di frame in memoria: 3, numero di page fault: 9 (su 12 accessi in memoria)

tempo	1	12
stringa riferim.	1 2 3 4 1 2 5 1 2 3 4 5	
pagine in memoria.	1 1 1 4 4 4 5 5 5 5 5 5 2 2 2 1 1 1 1 1 3 3 3 3 3 3 3 2 2 2 2 2 4 4	

Esempio 3 numero di frame in memoria: 4, numero di page fault: 10 (su 12 accessi in memoria)

tempo	1	12
stringa riferim.	1 2 3 4 1 2 5 1 2 3 4 5	
pagine in memoria.	1 1 1 1 1 1 5 5 5 5 4 4 2 2 2 2 2 2 1 1 1 1 1 5 3 3 3 3 3 3 3 2 2 2 2 2 4 4 4 4 4 4 3 3 3 3	

In alcuni algoritmi di rimpiazzamento non è detto che aumentando il numero di frame allora il numero di page fault diminuisca (e.g., FIFO). Questo fenomeno indesiderato si chiama **Anomalia di Belady**.

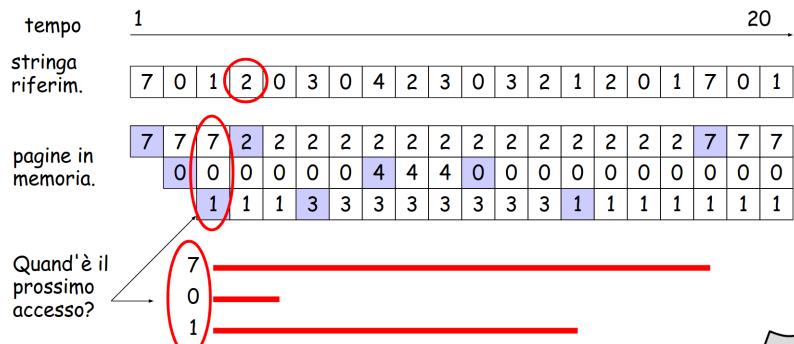


Algoritmo MIN - Ideale

Descrizione: seleziona come pagina vittima una pagina che non sarà più acceduta o la pagina che verrà acceduta nel futuro più lontano.

Ottimale perché fornisce il minimo numero di page fault, è un **algoritmo teorico** perché richiederebbe la conoscenza a priori della stringa dei riferimenti futuri del programma, viene utilizzato a posteriori come paragone per verificare le performance degli algoritmi di rimpiazzamento reali.

Esempio numero di frame in memoria: 3, numero di page fault: 9 (su 20 accessi in memoria)

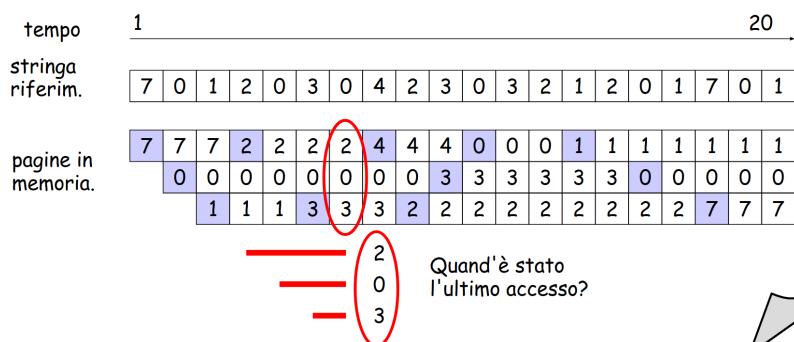


5.5.6 Algoritmo LRU (Least Recently Used)

Descrizione: seleziona come pagina vittima la pagina che è stata usata meno recentemente nel passato.

Basato sul presupposto che la distanza tra due riferimenti successivi alla stessa pagina non varia eccessivamente, stima la distanza nel futuro utilizzando la distanza nel passato.

Esempio numero di frame in memoria: 3, numero di page fault: 12 (su 20 accessi in memoria)



Implementazione: E' necessario uno specifico supporto hardware. La MMU deve registrare nella tabella delle pagine un **time-stamp** quando accede ad una pagina. Il time-stamp può essere implementato come un contatore che viene incrementato ad ogni accesso in memoria.

Bisogna gestire l'overflow dei contatori, essi devono essere memorizzati in memoria e questo richiede accessi addizionali alla memoria. La tabella deve essere scandita totalmente per trovare la pagina LRU.

Implementazione basata su stack: Si mantiene uno stack di pagine, tutte le volte che una pagina viene acceduta, viene rimossa dallo stack (se presente) e posta in cima allo stack stesso. In questo modo:

- in cima si trova la pagina utilizzata più di recente.
- in fondo si trova la pagina utilizzata meno di recente.

L'aggiornamento di uno stack organizzato come double-linked list richiede l'aggiornamento di 6 puntatori!

La pagina LRU viene individuata con un accesso alla memoria.

Algoritmi a stack

Definizione: si indichi con $S_t(A, m)$ l'insieme delle pagine mantenute in memoria centrale al tempo t dell'algoritmo A, data una memoria di m frame.

Un algoritmo a stack non genera casi di Anomalia di Belady. L'algoritmo di LRU è a stack.

Un algoritmo di rimpiazzamento viene detto **stack algorithm** se per ogni istante t si ha:
 $S_t(A, m) \subseteq S_t(A, m + 1)$

In altre parole: se l'insieme delle pagine in memoria con m frame è sempre un sottoinsieme delle pagine in memoria con $m + 1$ frame.

LRU - Implementazione approssimata

In entrambi i casi (contatori, stack), mantenere le informazioni per LRU è troppo costoso.

In realtà poche MMU forniscono il supporto hardware per l'algoritmo LRU, alcuni sistemi non forniscono alcun tipo di supporto, e in tal caso l'algoritmo FIFO deve essere utilizzato.

Reference bit: alcuni sistemi forniscono supporto sotto forma di reference bit, tutte le volte che una pagina è acceduta, il bit associato alla pagina viene aggiornato a 1.

Inizialmente, tutti i bit sono posti a zero dal s.o. , durante l'esecuzione dei processi, le pagine in memoria vengono accedute e i reference bit vengono posti a 1.

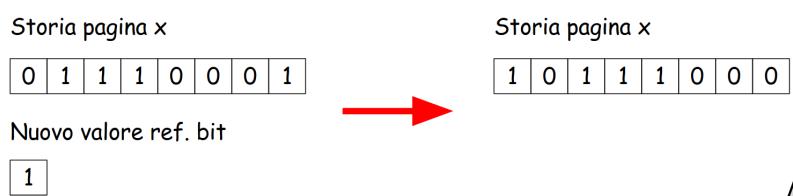
Periodicamente, è possibile osservare quali pagine sono state accedute e quali non osservando i reference bit, ma non conosciamo l'ordine in cui sono state usate.

Additional-Reference-Bit-Algorithm: Possiamo aumentare le informazioni di ordine "salvando" i reference bit ad intervalli regolari (ogni 100 ms)

Esempio: manteniamo 8 bit di "storia" per ogni pagina.

Il nuovo valore del reference bit viene salvato tramite shift a destra della storia ed inserimento del bit come most signif. bit

La pagina vittima è quella con valore minore; in caso di parità, si utilizza una disciplina FIFO.



Second-chance algorithm

Conosciuto anche come algoritmo dell'orologio, corrisponde ad un caso particolare dell'algoritmo precedente, dove la dimensione della storia è uguale a 1.

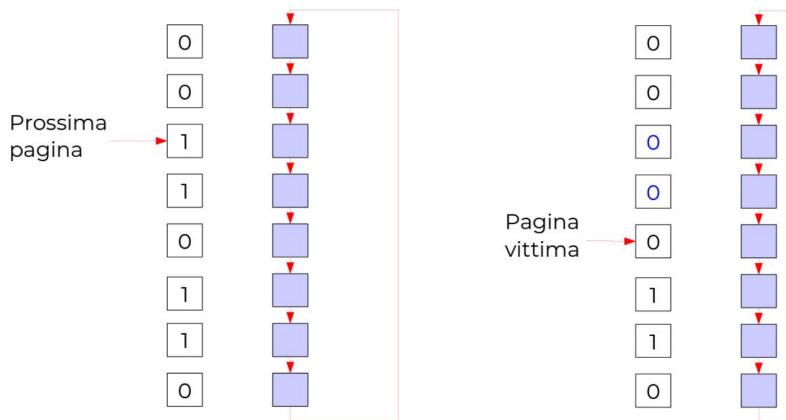
Descrizione: le pagine in memoria vengono gestite come una lista circolare, a partire dalla posizione successiva all'ultima pagina caricata, si scandisce la lista con la seguente regola:

- se la pagina è stata acceduta (reference bit a 1) allora il reference bit viene messo a 0.
- se la pagina non è stata acceduta (reference bit a 0) allora la pagina selezionata è la vittima.

L'idea è semplice: l'algoritmo seleziona le pagine in modo FIFO, se però la pagina è stata acceduta, gli si dà una "seconda possibilità" (second chance);

Si cercano pagine successive che non sono state accedute, se tutte le pagine sono state accedute, degenera nel meccanismo FIFO.

L'implementazione è semplice e non richiede capacità complesse da parte della MMU.



Altri algoritmi di rimpiazzamento

Least frequently used (LFU) si mantiene un contatore del numero di accessi ad una pagina, la pagina con il valore minore viene scelta come vittima.

Una pagina utilizzata spesso dovrebbe avere un contatore molto alto.

Può essere approssimato tramite reference bit.

Problemi: se una pagina viene utilizzata frequentemente all'inizio, e poi non viene più usata, non viene rimossa per lunghi periodi.

Most frequently used (MFU) si mantiene un contatore del numero di accessi ad una pagina, la pagina con il valore maggiore viene scelta come vittima

Pagine appena caricate hanno un valore molto basso, e non dovrebbero essere rimosse.

Può essere approssimato tramite reference bit

Problemi: problemi di performance.

5.5.7 Allocazione

Con algoritmo di allocazione (per memoria virtuale) si intende l'algoritmo utilizzato per scegliere quanti frame assegnare ad ogni singolo processo.

Allocazione locale ogni processo ha un insieme proprio di frame, poco flessibile.

Allocazione globale tutti i processi possono allocare tutti i frame presenti nel sistema (sono in competizione), può portare a **trashing**.

Trashing

Definizione: un processo (o un sistema) si dice che è in trashing quando spende più tempo per la paginazione che per l'esecuzione.

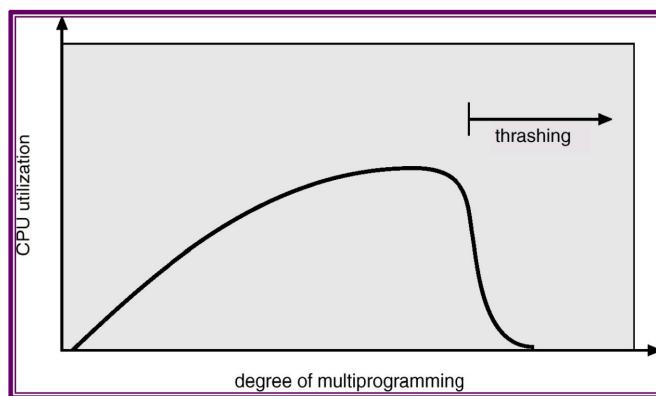
Le possibili cause in un sistema con allocazione globale: si ha trashing se i processi tendono a "rubarsi i frame a vicenda", ovvero non riescono a tenere in memoria i frame utili a breve termine, perché altri processi chiedono frame liberi e quindi generano page fault ogni pochi passi di avanzamento.

Esempio esaminiamo un sistema che accetti nuovi processi quando il grado di utilizzazione della CPU è basso. Se per qualche motivo gran parte dei processi entrano in page fault:

- la ready queue si riduce
- il sistema sarebbe indotto ad accettare nuovi processi

E' UN ERRORE!

Statisticamente, il sistema: genererà un maggior numero di page fault e di conseguenza diminuirà il livello della multiprogrammazione.



Working Set

Definizione: si definisce **working set** di finestra Δ l'insieme delle pagine accedute nei più recenti Δ riferimenti.

E' una rappresentazione approssimata del concetto di località, se una pagina non compare in Δ riferimenti successivi in memoria, allora esce dal working set; non è più una pagina su cui si lavora attivamente.

Esempio con $\Delta = 5$

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
<hr/>										<hr/>									
{ 0,1,2,7 }										{ 0,1,2 }									

Se si sceglie Δ troppo piccolo: si considera non più utile ciò che in realtà serve.

Minore inerzia nel "buttare via".

Se si sceglie Δ troppo grande: si considera utile anche ciò che non serve più.

Sistema più "conservatore".

Cosa serve il Working Set? Se l'ampiezza della finestra è ben calcolata, il working set è una buona approssimazione dell'insieme delle pagine "utili".

Sommendo quindi l'ampiezza di tutti i working set dei processi attivi, questo valore deve essere sempre minore del numero di frame disponibili, altrimenti il sistema è in trashing.

Come si usa il Working Set? Serve per controllare l'allocazione dei frame ai singoli processi.

Quando ci sono sufficienti frame disponibili non occupati dai working set dei processi attivi, allora si può attivare un nuovo processo.

Se al contrario la somma totale dei working set supera il numero totale dei frame, si può decidere di sospendere l'esecuzione di un processo.

Capitolo 6

File System

6.1 Introduzione

I computer possono utilizzare diversi media per registrare in modo permanente le informazioni *esempi: dischi rigidi, floppy, nastri, dischi ottici*

Ognuno di questi media ha caratteristiche fisiche diverse Compito del **file system** è quello di astrarre la complessità di utilizzo dei diversi media proponendo una interfaccia per i sistemi di memorizzazione: comune, efficiente conveniente da usare.

Dal punto di vista dell'utente, un file system è composto da due elementi:

- **file**: unità logica di memorizzazione.
- **directory**: servono per organizzare e fornire informazioni sui file che compongono un file system.

Il concetto di file è l'entità atomica di assegnazione/gestione della memoria secondaria, è una collezione di informazioni correlate che fornisce una vista logica uniforme ad informazioni correlate.

Attributi dei file

Nome: stringa di caratteri che permette agli utenti ed al sistema operativo di identificare un particolare file nel file system, alcuni sistemi differenziano fra caratteri maiusc./minusc., altri no.

Tipo: necessario in alcuni sistemi per identificare il tipo di file

Locazione e dimensione: informazioni sul posizionamento del file in memoria secondaria.

Data e ora: informazioni relative al tempo di creazione ed ultima modifica del file.

Informazioni sulla proprietà: utenti, gruppi, etc. utilizzato per accounting e autorizzazione.

Attributi di protezione: informazioni di accesso per verificare chi è autorizzato a eseguire operazioni sui file.

Altri attributi: flag (sistema, archivio, hidden, etc.), informazioni di locking, etc.

Tipi di file

A seconda della struttura interna possono essere senza formato (stringa di byte): file testo. Oppure con formato: file di record, file di database, a.out,...

A seconda del contenuto: ASCII/binario, sorgente, oggetto o eseguibile (oggetto attivo).

Alcuni S.O. supportano e riconoscono diversi tipi di file così che il s.o. può evitare alcuni errori comuni, quali ad esempio stampare un file eseguibile.

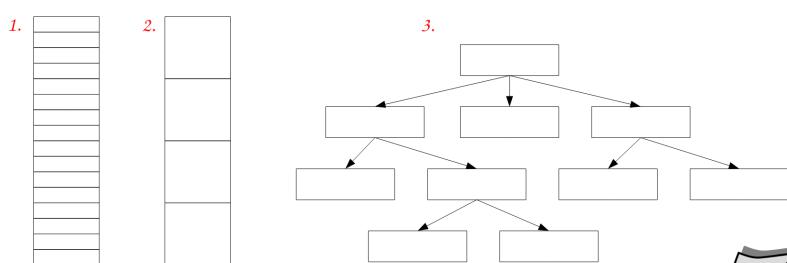
Esistono tre tecniche principali per identificare il tipo di un file:

- meccanismo delle estensioni
- utilizzo di un attributo "tipo" associato al file nella directory
- magic number

Struttura dei file

I file possono essere strutturati in molti modi:

- sequenze di byte
- sequenze di record logici
- file indicizzati (struttura ad albero)



I sistemi operativi possono attuare diverse scelte nella gestione della struttura dei file.

Scelta minimale: i file sono considerati semplici stringhe di byte, a parte i file eseguibili il cui formato è dettato dal s.o. .

Parte strutturata/parte a scelta dell'utente.

Diversi tipi di file predefiniti.

Dunque è un trade-off. Più formati rendono il codice di sistema più ingombrante, causano incompatibilità di programmi (accesso a file di formato differente), ma permettono una gestione efficiente e non duplicata per i formati speciali.

Invece meno formati rendono il codice di sistema più snello.

Metodi di accesso

- Sequenziale: read, write
- Accesso diretto: read pos, write pos (*oppure operazione seek*).
- Indicizzato: read key, write key (*tipico dei database*).
- A indice: è una tabella di corrispondenza chiave-posizione.

Operazioni sui file

Operazioni fondamentali sui file:

- creazione
- apertura/chiusura
- lettura/scrittura/append
- posizionamento
- cancellazione
- troncamento
- lettura/scrittura attributi

L'API (interfaccia per la programmazione) relativa alle operazioni su file è basata sulle operazioni open/close.

I file devono essere "aperti" prima di effettuare operazioni e "chiusi" al termine.

L'astrazione relativa all'apertura/chiusura dei file è utile per mantenere le strutture dati di accesso ai file, controllare le modalità di accesso/gestire gli accessi concorrenti e definire un descrittore per le operazioni di accesso ai dati.

6.1.1 Directory

L'organizzazione dei file system è basata sul concetto di directory, che fornisce un'astrazione per un'insieme di file. In molti sistemi, le directory sono file speciali.

Operazioni sulle directory

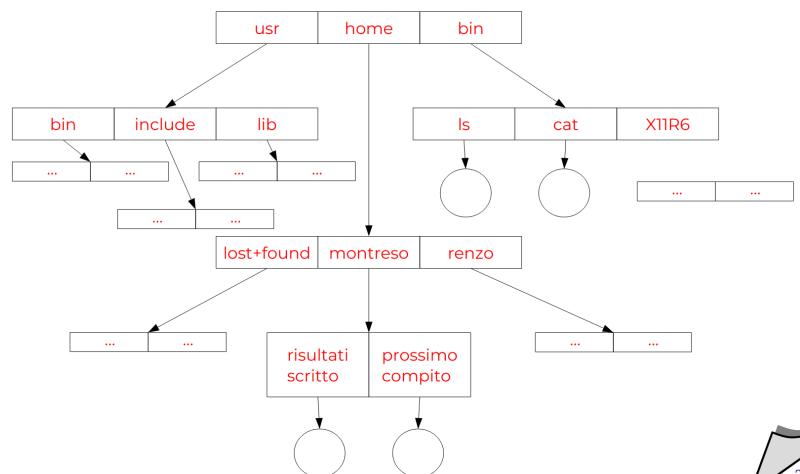
Operazioni definite sulle directory:

- creazione
- cancellazione
- apertura di una directory
- chiusura di una directory
- lettura di una directory
- rinominazione
- link/unlink

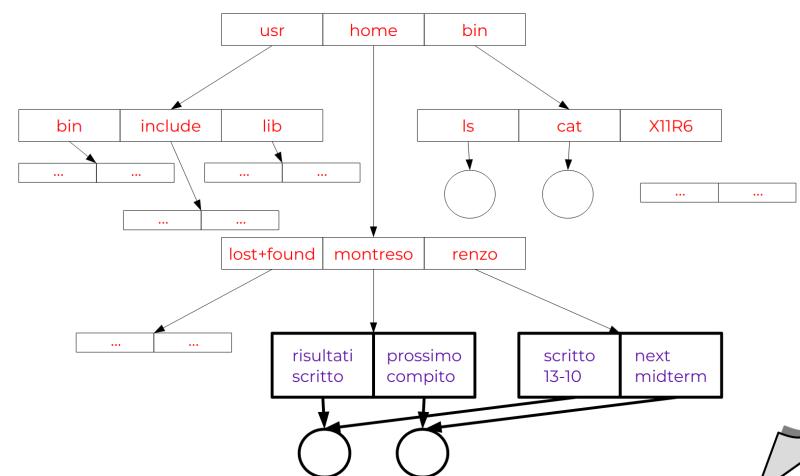
6.1.2 Strutture delle directory

La struttura di una directory può essere: a livello singolo, a due livelli, ad albero, a grafo aciclico, a grafo.

Directory strutturate ad albero



Directory strutturate a grafo aciclico



In un sistema operativo multitasking, i processi accedono ai file indipendentemente.

Come vengono viste le modifiche ai file da parte dei vari processi? In UNIX, le modifiche al contenuto di un file aperto vengono rese visibili agli altri processi immediatamente. Esistono due tipi di condivisione del file: condivisione del puntatore alla posizione corrente nel file, oppure condivisione con distinti puntatori alla posizione corrente.

6.2 Visione implementazione

Implementazione del file system I problemi da tenere in considerazione: organizzazione di un disco, allocazione dello spazio in blocchi, gestione spazio libero, implementazione delle directory, tecniche per ottimizzare le prestazioni, tecniche per garantire la coerenza.

6.2.1 Organizzazione del disco

Struttura del disco: un disco può essere diviso in una o più partizioni, porzioni indipendenti del disco che possono ospitare file system distinti.

Il primo settore dei dischi è il cosiddetto **master boot record** (MBR) è utilizzato per fare il boot del sistema. Esso contiene la **partition table**, l'indicazione della partizione attiva e al boot, il MBR viene letto ed eseguito.



Struttura di una partizione Ogni partizione inizia con un boot block, il MBR carica il boot block della partizione attiva e lo esegue.

Il boot block carica il sistema operativo e lo esegue.

L'organizzazione del resto della partizione dipende dal file system.



6.2.2 Allocazione

L'hardware e il driver del disco forniscono accesso al disco visto come un insieme di blocchi dati di dimensione fissa.

Come vengono scelti i blocchi dati da utilizzare per un file e come questi blocchi dati vengono collegati assieme a formare una struttura unica?

Questo è il problema dell'allocazione.

Allocazione contigua

Descrizione: i file sono memorizzati in sequenze contigue di blocchi di dischi.

Vantaggi: non è necessario utilizzare strutture dati per collegare i blocchi. L'accesso sequenziale è efficiente poiché i blocchi contigui non necessitano operazioni di seek.

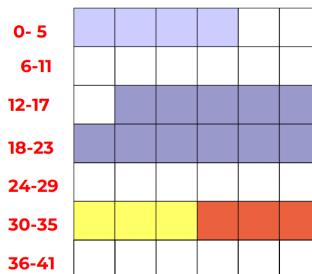
L'accesso diretto è efficiente:

block= offset / blocksize

pos= offset % blocksize

Svantaggi: si ripropongono tutte le problematiche dell'allocazione contigua in memoria centrale (*fragmentazione esterna, politica di scelta dell'area di blocchi liberi da usare per allocare spazio per un file*).

Inoltre i file non possono crescere di dimensione.



directory		
Name	Start	Size
a	0	4
b	13	11
c	30	3
d	33	3

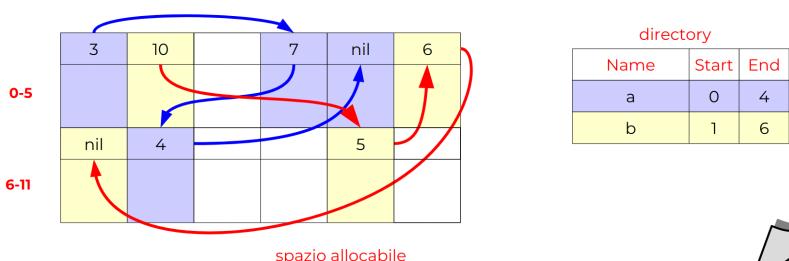
Allocazione concatenata

Descrizione: ogni file è costituito da una lista concatenata di blocchi dove ogni blocco contiene un puntatore al blocco successivo. Il descrittore del file contiene i puntatori al primo e all'ultimo elemento della lista.

Vantaggi: risolve il problema della frammentazione esterna. L'accesso sequenziale o in "append mode" è efficiente.

Svantaggi: L'accesso diretto è inefficiente, progressivamente l'efficienza globale del file system degrada (*i blocchi sono disseminati nel disco, aumenta il n. di seek*).

La dimensione utile di un blocco non è una potenza di due. Se il blocco è piccolo (512 byte) l'overhead per i puntatori può essere rilevante.

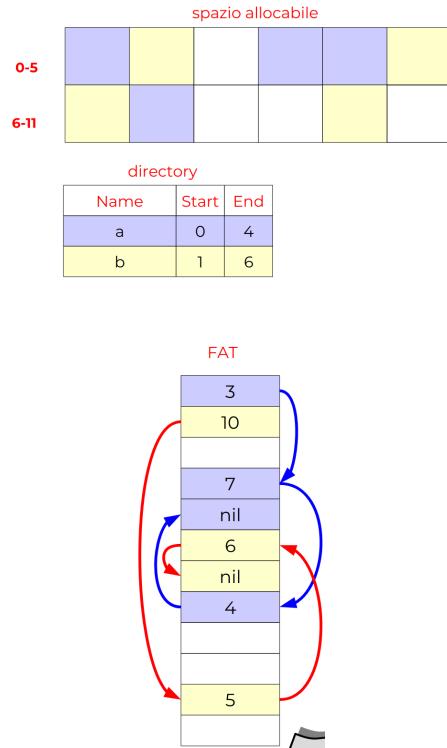


Allocazione basata su FAT

Descrizione: invece di utilizzare parte del blocco dati per contenere il puntatore al blocco successivo si crea una tabella unica con un elemento per blocco (o per cluster).

Vantaggi: i blocchi dati sono interamente dedicati ai dati, possibile fare caching in memoria dei blocchi FAT. L'accesso diretto diventa così più efficiente, in quanto la lista di puntatori può essere seguita in memoria. (*metodo usato da DOS, macchine fotografiche, chiavette USB*)

Svantaggi la scansione richiede anche la lettura della FAT, aumentando così il numero di accessi al disco.

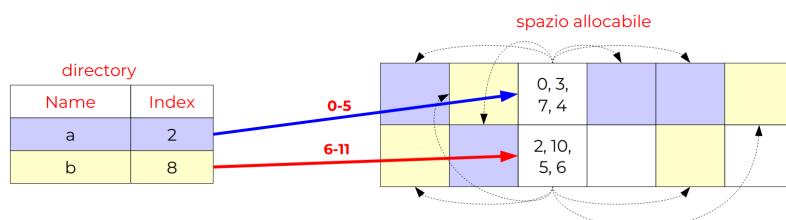


Allocazione indicizzata

Descrizione: l'elenco dei blocchi che compongono un file viene memorizzato in un blocco indice. Per accedere ad un file, si carica in memoria la sua area indice e si utilizzano i puntatori contenuti.

Vantaggi: risolve il problema della frammentazione esterna, è efficiente per l'accesso diretto, il blocco indice deve essere caricato in memoria solo quando il file è aperto.

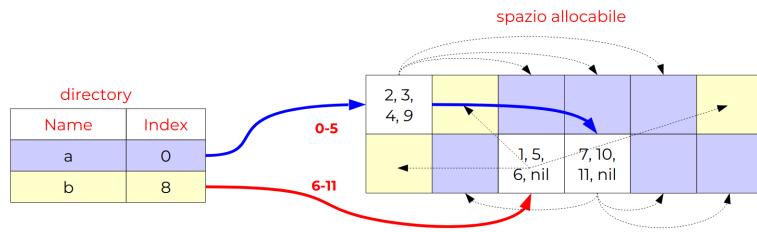
Svantaggi: la dimensione del blocco indice determina l'ampiezza massima del file, utilizzare blocchi indici troppo grandi comporta un notevole spreco di spazio.



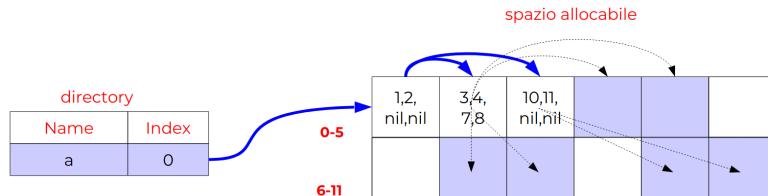
Come risolvere il trade-off?

Allocazione indicizzata - Possibili soluzioni

Concatenazione di blocchi indice l'ultimo elemento del blocco indice non punta al blocco dati ma al blocco indice successivo. Si ripropone il problema per l'accesso diretto a file di grandi dimensioni.

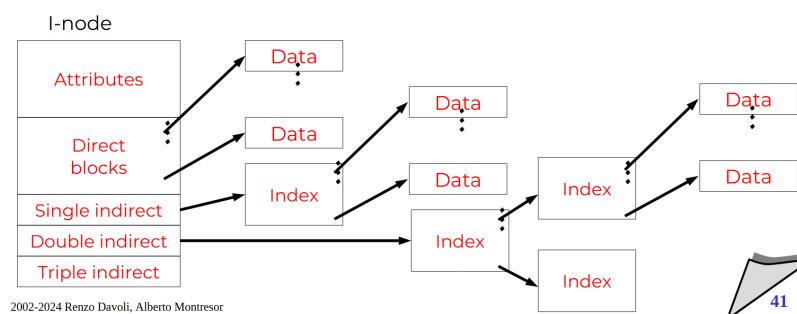


Indice multilivello si utilizza un blocco indice dei blocchi indice. Degradano le prestazioni, in quanto richiede un maggior numero di accessi.



Allocazione indicizzata e UNIX

In UNIX ogni file è associato ad un **i-node** (index node). Un i-node è una struttura dati contenente gli attributi del file, e un indice di blocchi diretti e indiretti, secondo uno schema misto.



Allocazione e performance Lo schema UNIX garantisce buone performance nel caso di accesso sequenziale, file brevi sono acceduti più velocemente e occupano meno memoria.

Ci sono anche ulteriori miglioramenti, il pre-caricamento (*per esempio nell'allocazione concatenata fornisce buone prestazioni per l'accesso sequenziale*), combinazione dell'allocazione contigua e indicizzata, contigua per piccoli file ove possibile, indicizzata per grandi file.

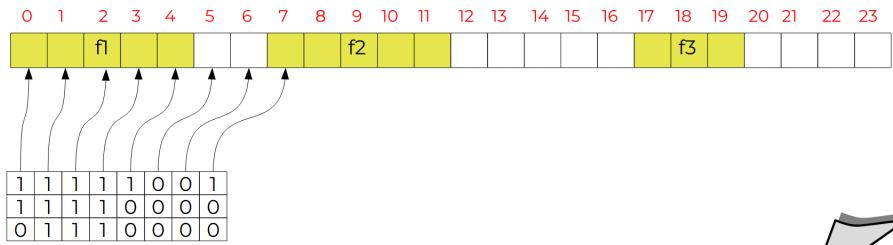
6.2.3 Gestione spazio libero

Mappa di bit

Descrizione: ad ogni blocco corrisponde un bit in una bitmap. I blocchi liberi sono associati ad un bit di valore 0, i blocchi occupati sono associati ad un bit di valore 1.

Vantaggi: semplice, è possibile selezionare aree contigue.

Svantaggi: la memorizzazione del vettore può richiedere molto spazio.

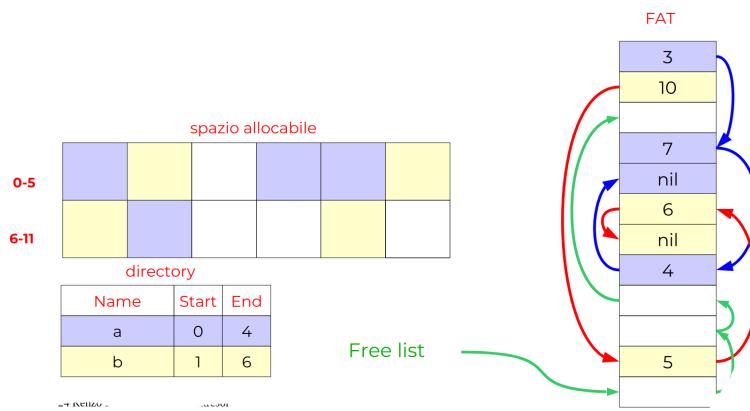


Lista concatenata

Descrizione: blocchi liberi vengono mantenuti in una lista concatenata, si integra perfettamente con il metodo FAT per l'allocazione delle aree libere.

Vantaggi: richiede poco spazio in memoria centrale.

Svantaggi: l'allocazione di un'area di ampie dimensioni è costosa e l'allocazione di aree libere contigue è molto difficoltosa.

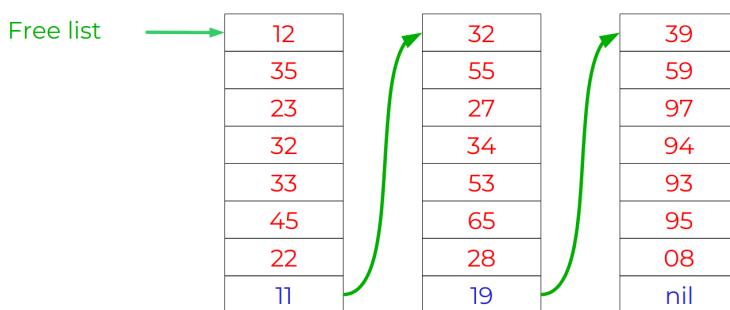


Lista concatenata (blocchi)

Descrizione: è costituita da una lista concatenata di blocchi contenenti puntatori a blocchi liberi.

Vantaggi: ad ogni istante, è sufficiente mantenere in memoria semplicemente un blocco contenente elementi liberi. Non è necessario utilizzare una struttura a dati a parte; i blocchi contenenti elenchi di blocchi liberi possono essere mantenuti all'interno dei blocchi liberi stessi.

Svantaggi: l'allocazione di un'area di ampie dimensioni è costosa e l'allocazione di aree libere contigue è molto difficoltosa



(fine slide 48)