

SISTEMI OPERATIVI

Donazioni sempre ben accette ❤
<https://www.paypal.me/mgianinni01>

Introduzione

SISTEMI OPERATIVI

Insieme di programmi di natura generale che mette in grado le app di usare l'hardware a disposizione in modo efficiente → genera efficienza hardware per i programmati che sviluppano applicazioni → SO presenta che permette di rendere programmi indipendenti dall'hardware

↓ Obiettivi fondamentali

creare programmi utente di elaborazione semplice

↳ Non ci si preoccupa degli esodi a ramme

Si cerca inoltre di tenere la CPU sempre occupata e, nel caso di un sistema multi programato, si gestiscono efficientemente le risorse usate concorrentemente in base alla natura e priorità del processo.

↓ Struttura

Indispensabile per sistemi complessi



Più facile usare - rare, costituite, riportate fatti indipendenti

Definiscono funzioni (referto), parametri ma non implementazione → Quanta ista al livello a cui appartiene l'interfaccia

Programmatore di sistema conosce interfaccia hardware (linguaggio assembly)

Il sistema operativo facilita lo sviluppo e la portabilità dei programmi e permette di realizzare politiche di gestione delle risorse hardware (il filesystem rende non necessario sapere la struttura hardware del disco e dove regolare gli accessi concorrenti al disco con politiche di mutua esclusione e di scheduling). Fornisce inoltre meccanismi di protezione e garantisce la sicurezza del sistema e la tolleranza ai guasti

Un processo non può soltanto sulle informazioni che non gli appartengono

API: interfaccia che definisce funzioni macchina astratta → più semplice da usare, efficiente e sicuri di macchina fisica su cui si basa

↓ Esempio

Macchina RISC viene trattata in macchina

Definizioni di SO: Allocatore di risorse → Attore e permette di usare le

CISC

risorse in modo efficiente applicando anche i meccanismi della protezione. Gestisce le risorse hardware e quelle di tipo logico (ad esempio file). Gestisce inoltre accessi conflittuali alle risorse

Programma di controllo → Prende in improprio del calcolatore ed

e semplificare riduzione da usare, utilizzare l'hardware in modo efficiente. degli esodi a ramme nella scrittura del programma. Si cerca inoltre di tenere la CPU sempre occupata e, nel caso di un sistema multi programato, si gestiscono efficientemente le risorse usate concorrentemente in base alla natura e priorità del processo.

→ Programmi. Il programmatore non deve conoscere l'interfaccia operativa, semplificata ulteriormente dalle librerie

→ Livello software di natura generale organizzato in diversi modi. Deve gestire le risorse e quindi la CPU, la memoria e i file. Attivano poi i driver che rappresentano le funzioni delle periferiche

→ CPU, RAM, Video Disco, Tastiera, Stampante, Modem, mouse: collettore per svolgere più task

Single indetto della rete → Progettato e si risolveva moltissimi problemi: eseguire interazioni

↓
ma cosa succede nella CPU ad ogni istante? di unico programma in esecuzione tutte le altre cose sono chiamate

- Ziose sempre è il kernel mentre alle funzioni del kernel

• CENNI STORICI

Si parte da metà degli anni '50 e anche prima nella seconda guerra mondiale in cui si costruiscono calcolatori residucentoli per compiti specifici. Non avendo i transistor le prestazioni erano molto scarse. Il progetto di costruzione di calcolatori prende il via nel 2° dopoguerra. Abbiamo il CT, dedicata alla risoluzione di classici problemi fisici / matematici / ingegneristici, con una struttura molto simile ai computer moderni. Abbiamo poi l'Olivetti Elea 9003, ancora funzionante, costruita con i transistor che l'hanno resa una macchina commerciale venduta molto. Queste due macchine non avevano SO e memoria persistente ed avevano come periferiche solo un lettore di schede perforate ed un registratore. I programmi venivano scritti su schede in Fortran e caricati in memoria centrale del gestore della macchina. L'operatore poi aveva a disposizione un "display" di luci per vedere lo stato della "RAM" e una volta scritto IP in programma cominciava a correre il compilatore. D'output veniva dato tramite schede perforate. Si capisce bene la scarsa efficienza di questo processo.

↓
Si introducono i sistemi mono programmati batch (insieme di schede): viene data più autonomia → JCL (Job Control Language): \$ dice che deve dicono cose fatte dalla macchina → Sistema operativo primordiale che permette funzionamento autonomo della macchina → Primo SO: Monitor + BIOS
↓
interviste JCL Basic I/O system

Si vuole poi che la macchina faccia I/O e output contemporaneamente: si introduce il disco → esecuzione fatta dal processore che ha bisogno di memoria buffer
↳ Spooling:

→ Si introduce poi il DMA che permette correntemente di più batch
↓

Si ricalcola il batch che va associato: di media termina (inizialmente il primo che arriva veniva eseguito e poi shortest job first)
↓

Vengono poi create memorie centrali più grandi: più programmi applicativi e inserito uno scheduler → **sistema multiprogrammato**.
↓

Programmi: sequenza di **CPU-burst** e **I/O-burst** → possono dare la CPU ad altro programma
↓

Puntano da esecuzione sequenziale a multi-tasking: programma eseguito in più volte. Si introducono gli stati di esecuzione del programma: pronto, esecuzione, bloccato
↓

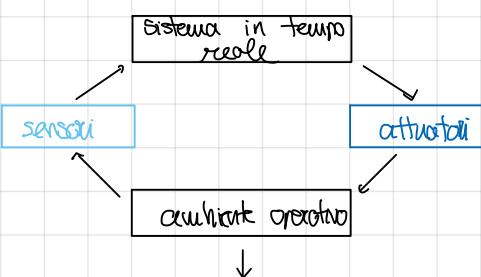
Sistemi preemptive: la CPU può essere raccapata dal sistema operativo indipendentemente dallo stato del programma. In questo periodo si ha anche la transizione dai mainframe a calcolatori più piccoli.
↓

Dovendo garantire la multietenza anche da remoto: linea di comando con linea telefonica

Il passaggio ai mini - calcolatori aumenta sensibilmente la platea degli utenti che li utilizzano. Nasce inoltre l'esigenza di un'interfaccia video più evoluta. Questo comporta la necessità di avere dei device driver per tastiera e monitor e si lasciava in ogni caso con un'interfaccia di tipo texture. Avviene in quegli anni i primi personal computer. I primi hanno solo lettori di floppy e non un hard disk → BIOS: disk-operating system (OS sul floppy caricato in RAM). Fornisce interfaccia texture e BIOS. Si parla poi a PC con disco rigido e l'OS viene messo su tale disco → Il PC diventa comune a tutti. L'interfaccia uomo-macchina viene poco migliorata introducendo le interfacce grafiche. Viene poi introdotta la rete per interconnettere i computer. Da qui inizialmente esce disponibile per pochi ma poi viene resa disponibile per tutti tramite doppino telefonico e modem (da PC a PC). Si apre così il mondo delle applicazioni distribuite con approccio client-server → Si aggiunge nei sistemi operativi un'interfaccia che permette l'accesso alla rete. Da qui succede che la disponibilità dell'accesso alla rete e la riduzione dell'interfaccia del hardware e software: nuove funzioni hardware messe a disposizione.

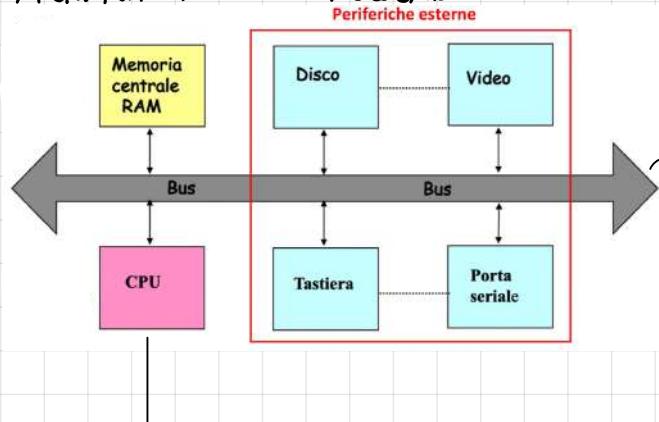
Sistemi time-sharing: introdotto con mini-computer. Il sistema alloca risorse con politiche di condivisione del tempo: preemptive → CPU: tempo da assegnare definendo un quanto di tempo → Allo scadere del tempo la CPU viene sempre revocata: migliore tempo di risposta migliorando la reattività del sistema → Algoritmo rand-robin: dato un insieme di processi pronti, tutti vengono portati avanti indipendentemente della loro completezza. Nei sistemi non-preemptive questo non è applicabile a causa della presenza di processi CPU-bound → Nei sistemi preemptive i processi I/O-bound (interattivi) vengono eseguiti subito e in fretta nonostante la presenza di processi CPU-bound → Sono procedure rapide per effettuare i calcoli di controllo (tempo overhead)

Sistemi in tempo reale: special-purpose → Calcolatori pensati per eseguire applicazioni specifiche che gestiscono sistemi esterni di tipo diverso (automobile, orologio,...)

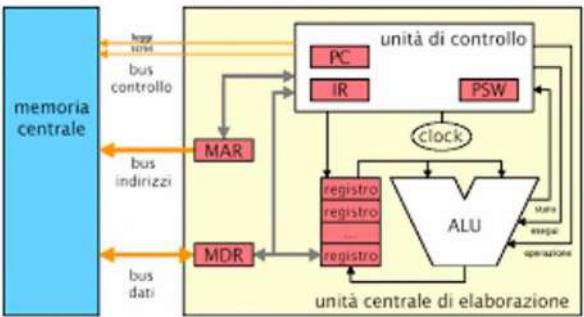


Sistemi embedded (incorporati) → In tempo reale perché devono intervenire entro una certa accadenza di tempo: deadline → propria di ogni task. I task sono tipicamente periodici in quanto i sensori campionano con una certa frequenza di campionamento. Viene costruita una catena di processi pronti in modo da garantire il rispetto di tutte le deadline (rete: istante di tempo nel mondo reale)

• ARCHITETTURA DEL CALCOLATORE



In generale ci sono più livelli: insieme di linee di comunicazione che permettono ai moduli connesi di fare operazioni di lettura e scrittura. Prevede linee di trasferimento e linee di indirizzo. È un sistema di tipo broadcast in quanto



il contenuto alterando la anche i dati in cui il spazio fisso corrisponde più programmi si distinguono in unità di controllo registri di controllo → il contenuto è funzionale PC (program counter) / IP: indirizzo di memoria in cui si trova istruzione del programma. Il registro viene incrementato della dimensione in byte dell'istruzione eseguita e fatta sempre ad istruzione successiva. Questa regola può avere cambiata dalla regola per contenere l'istruzione che deve essere eseguita ed è grande a seconda dalla grandezza delle istruzioni del sistema. Le istruzioni vengono messe in fase di fetch. PSW contiene il livello di privilegio al quale il programma in esecuzione può fare operazioni. È presente anche una "sintesi" dell'esecuzione dell'istruzione precedente sotto forma di flag utilizzati da istruzioni di salto condizionato. SP non compare anche se è necessario per puntare alla pila FIFO per le architetture che ne fanno uso per fare le operazioni di PUSH e POP. Vi è poi un'interfaccia che permette il collegamento con l'unità di elaborazione e il bus. Questo ha delle linee dedicate ai dati (bus dati) e bus indirizzi con capacità congruente a hit necessari per indirizzare tutta la memoria. Si ha poi il bus di controllo: hit per codificare operazione. Nel caso di software l'informazione viene trasferita da RAM a CPU. Viceversa per la scrittura, che modifica lo stato della memoria. Per pilotare il bus dati e indirizzi vi hanno i registri MAR e MDR che contengono rispettivamente l'indirizzo di memoria e i dati (buffer bidirezionali). I bus sono di tipo sincroni e quindi tutte le transazioni hanno durata predefinita: non servono segnali di controllo. MDR si interfaccia con entrambe le unità in quanto i dati trasmessi dalla RAM possono essere sia istruzioni che dati veri e propri. Nell'unità di elaborazione abbiamo la ALU (op. aritmetiche tramite somma) che ha 2 registri di ingresso A e B per presentare i operandi ed un registro C di uscita per il risultato. Vi sono altri registri per registrare il carry flag e definire il tipo di operazione. I registri A, B e C sono comuni con uno stack di registri operativi utili al programmatore ma si basano su ogni operazione. I registri hanno una velocità superiore al clock della CPU (molto veloci).

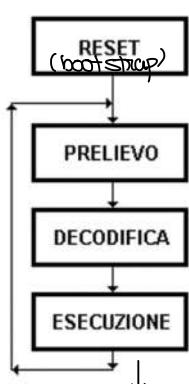
non prevede una comunicazione punto-punto.
I dispositivi master iniziano le transazioni, i dispositivi slave ricevono richieste di accesso di master. Non è possibile che 2 master cedano contemporaneamente al bus (impossibile a livello fisico) → mutua esclusione: bus dove esiste controllo da dispositivo a parte che schedula gli accessi dei master.

→ Modello di von Neumann: esecutore di istruzioni definite in un certo insieme composte in un programma in memoria centrale. La regola di esecuzione è sequenziale partendo dalla prima → tramite IP vengono eseguite le istruzioni. Questo viene incrementato prima di eseguire l'istruzione successiva. Vi sono tuttavia le istruzioni di salto che ne modificano regola generale. In memoria devono esserci programmi vengono eseguiti se lo chiedono e saltare da uno all'altro. La CPU nella prima ottiene del programma.

Il PC (program counter) / IP: indica l'indirizzo di memoria in cui si trova l'istruzione del programma. Il registro viene incrementato della dimensione in byte dell'istruzione eseguita e fatta sempre ad istruzione successiva. Questa regola può avere cambiata dalla regola per contenere l'istruzione che deve essere eseguita ed è grande a seconda dalla grandezza delle istruzioni del sistema. Le istruzioni vengono messe in fase di fetch. PSW contiene il livello di privilegio al quale il programma in esecuzione può fare operazioni. È presente anche una "sintesi" dell'esecuzione dell'istruzione precedente sotto forma di flag utilizzati da istruzioni di salto condizionato. SP non compare anche se è necessario per puntare alla pila FIFO per le architetture che ne fanno uso per fare le operazioni di PUSH e POP. Vi è poi un'interfaccia che permette il collegamento con l'unità di elaborazione e il bus. Questo ha delle linee dedicate ai dati (bus dati) e bus indirizzi con capacità congruente a hit necessari per indirizzare tutta la memoria. Si ha poi il bus di controllo: hit per codificare operazione. Nel caso di software l'informazione viene trasferita da RAM a CPU. Viceversa per la scrittura, che modifica lo stato della memoria. Per pilotare il bus dati e indirizzi vi hanno i registri MAR e MDR che contengono rispettivamente l'indirizzo di memoria e i dati (buffer bidirezionali). I bus sono di tipo sincroni e quindi tutte le transazioni hanno durata predefinita: non servono segnali di controllo. MDR si interfaccia con entrambe le unità in quanto i dati trasmessi dalla RAM possono essere sia istruzioni che dati veri e propri. Nell'unità di elaborazione abbiamo la ALU (op. aritmetiche tramite somma) che ha 2 registri di ingresso A e B per presentare i operandi ed un registro C di uscita per il risultato. Vi sono altri registri per registrare il carry flag e definire il tipo di operazione. I registri A, B e C sono comuni con uno stack di registri operativi utili al programmatore ma si basano su ogni operazione. I registri hanno una velocità superiore al clock della CPU (molto veloci).

Il mio preconcetto ha m'organizzazione logica per poterlo usare con istruzioni Auemhler. Abbiamo uno spazio di memoria che dà una visione funzionale (vettori) alla memoria centrale ed uno spazio di I/O che è la collezione di tutti i registri che definiscono funzionalmente una periferica → Ciascuna di esse è caratterizzata da un device ed un controller che collega con il device con un bus interno specifico e all'esterno con il bus del sistema secondo il suo protocollo. Deve quindi sapere come comandare la periferica e come comunicare con il processo applicativo che la controlla. Affinché la periferica sia visibile tramite il bus si definisce una visione funzionale standard per tutti i dispositivi data da registro di controllo, stato e dati. Il primo è visibile della CPU mentre il secondo è di natura esterna. Questi registri hanno tutti indirizzi presenti nello spazio di I/O organizzato come vettore di indirizzi (= spazio di memoria) ad accesso casuale. A causa della distinzione è necessario discriminare gli indirizzi: nel bus c'è una hit che specifica il tipo. Nella CPU vi sono poi i registri operativi e di stato (IP, F, SF).

↓



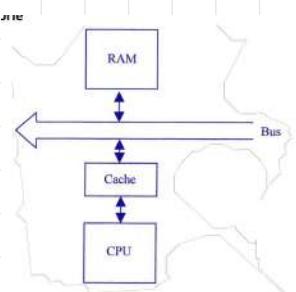
L'unità di controllo ripete d'infinito queste fasi. La lunghezza del ciclo dipende dall'architettura.

getch: trasferisce da RAM a IR istruzione da eseguire. Alla fine di questo si aggiorna il registro IP (contiene istruzione precedente esecuito)

Si legge IR e lo si decodifica in codice operativo (op. da fase) e operandi per comandare la ALU. Non è richiesto accesso al bus.
Si attende il tempo fisso di propagazione dei dati e la produzione del risultato da parte della ALU

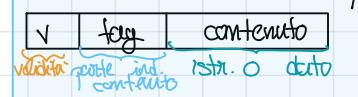
Le istruzioni di rotto modificano il contenuto di IP nella fase di esecuzione con il valore calcolato della ALU. Un altro registro importante è F (Flag) che contiene i flag (CF, ZF, SF, OF). Le istruzioni in genere sono nella forma `ELAB source, destination` ove possono mancare uno dei due operandi e possono essere di trasferimento, arithmetiche, logiche, translation/rotation, salt, gestione di rotogrammi e alt. In genere, le istruzioni che usano la memoria fanno uso del bus e quindi la fase di esecuzione è più lenta.

da memoria

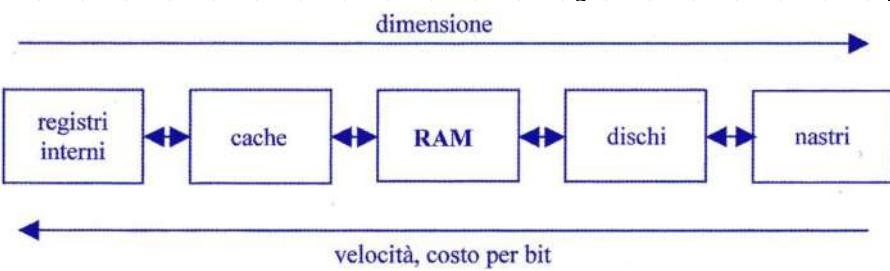


La CPU può leggere e scrivere attraverso il bus una locazione di memoria → array di locazioni grandi 1 byte. Tutte le volte che la CPU fa un accesso in RAM viene quindi trasferito un byte indicato da indirizzi. 1 byte può essere letto o scritto. Ci attendiamo che il bus contenga almeno 8 byte dati (1 byte = 8 bit). È possibile anche averne più di 8 per trasferire più di 1 byte per trasmissione.

Suggeriscono di avere 32 → L'indirizzo rimane riferito ad 1 byte ma ne possono avere trasferiti 4. Vi è, oltre alla memoria principale, la memoria cache. Questo permette di velocizzare le operazioni sfruttando il principio di località dei dati: quando viene generato l'indirizzo della successiva istruzione da eseguire questo sarà vicino (cicli). I notti, che non rispettano questo principio, sono presenti con frequenza molto piccola. Questo principio vale, oltre che per le istruzioni, anche per i dati (vettori, matrici). La CPU genererà quindi indirizzi vicini tra di loro in RAM. Si intende quindi la cache che "avvicina" i dati alla CPU organizzati.

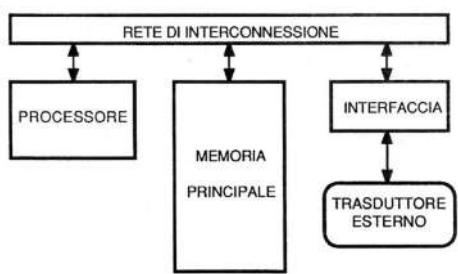


Supponiamo di avere all'inizio dell'esecuzione di un programma. Non abbiamo ovviamente località. In genere abbiamo più livelli di cache. La cache ha una porzione dell'indirizzo e lo confronta con il tag (il quale deve essere 1). Se c'è match, ciò che la CPU sta cercando è in cache e quindi presenta il contenuto della CPU. Tornando alla situazione iniziale, si avranno solo accessi. In questo caso si legge in RAM e vengono portati in cache a byte. Una di questi è quello utile e tutti vengono salvati nel contenuto. La parte più significativa viene messa nel tag. Se nell'esecuzione successiva il tag coincide si ha una hit e quindi il trasferimento si limita alla cache. Nella cache si trovano sia programmi che dati e prima o poi si ricevono di contenuto. Questo può diventare obsoleto perché il programma va ad eseguire istruzioni lontane e perché avendo un sistema multiprogrammato il contesto può essere cambiato. In quanto caso il contenuto viene invalidato. Per quanto riguarda la data cache, possono limitare la scrittura alla cache perché si garantisca che la modifica si propaghi alla RAM prima o poi. Il trasferimento avviene quando il bus non è utilizzato.



Maggiori velocità. Le memorie più veloci e meno costose sono interne alla CPU. Anni fa poi si è scoperto il confine delle memorie volatili. Si parla poi a memoria persistente (tempo ~ 100 ns nelle nostre applicazioni) che sono molto meno veloci. La memoria principale non è tutta volatile in quanto è presente la memoria ROM (Read-Only Memory) che non possono modificare (velocemente). Questa ci serve per il bootstrap ed in particolare per il primo fetch in quanto contiene le istruzioni che devono essere eseguite immediatamente. Per quanto riguarda le memorie non volatili utilizzano il disco, importante anche perché è possibile fare DMA per non impegnare la CPU. Anche nel caso del disco è importante il principio di località. Vi sono infine i nastri, anche essi persistenti ma, a differenza del disco, non ad accesso diretto ma sequenziale (dove leggere le i-1 informazioni precedenti prima di leggere la i-esima).

periferiche



Si hanno due protocolli. Utilizziamo insieme per un accesso più semplice la differenziazione fra spazio di memoria e spazio di I/O.

↓ Visione funzionale

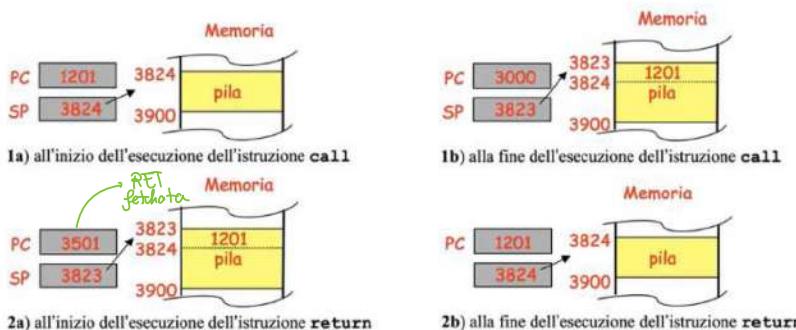
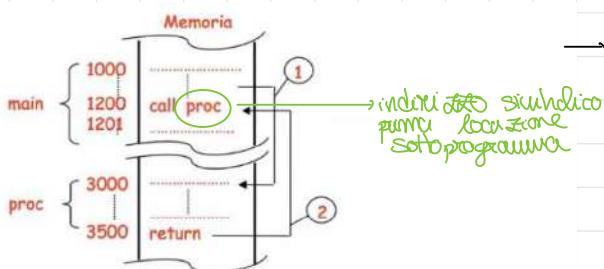
lli	ls	c	→ controllo:
intervento	scritt.		Il cambio di contenuto fa partire processo esterno
je	esito	S	→ CPU invia comandi a dispositivo. S-1: inizia operazione
intervento	scritt.		completo eseguito da CPU
D		D	↓ Cambia a termine operazione

→ Dati: per scaricare contenuto

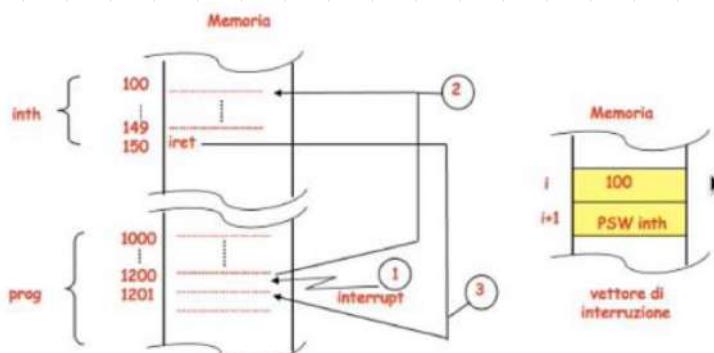
Consideriamo le periferie che sono composte da interfaccia e trasduttore esterno che permette di interagire con il mondo esterno: tramite impulsi elettrici si permette comunicazione. Può funzionare sia in sincronia che in asincrona e funziona grazie a convertitori D/A. C'è bisogno per dell'interfaccia per comunicare con il sistema operativo. Per la comunicazione

→ Il cambio di contenuto fa partire processo esterno
CPU invia comandi a dispositivo. S-1: inizia operazione
completo eseguito da CPU
↓ Cambia a termine operazione

Si mettono nello spazio di 110 utilizzando ogni tripla per costruire vettori degli interruzioni a quelli dello spazio di memoria. c'è però bisogno di sincronizzare CPU e portefinie. Il primo modo è il polling / controllo di programma: è compito dell'applicazione leggere nel registro di stato per vedere se il comando inviato tramite C è terminato (ad esempio). Si avrà quindi una IN del registro di stato di quel dispositivo. Non è applicabile ad un mittente multiprogrammato a causa delle busy-wait (attese lunghe). L'altra alternativa è quella di attivare le interruzioni che eseguono di fatto altro durante le operazioni di I/O. Il segnale di interruzione viene inviato alla fine dell'operazione e ricavato dalla CPU che interrompe il programma corrente per eseguire la routine associata all'interruzione. In generale per la memoria si utilizza il meccanismo della pila.



ma pop e mette ciò che ha estratto in
↓ interruzioni



Questo deve trarre ed eseguire la routine ad cui è associata. Possiamo enumerare i dispositivi ed utilizzare il numero per individuarli in memoria l'indirizzo della prima istruzione della routine. L'enumerazione fatta dal bootstrap viene indicato il livello PSW (V/S) a cui si vuole eseguire l'INT. Il PSW ci serve in quanto la routine è un programma del SO che deve avere eseguito al livello sistema in modo da accogliere anche le interruzioni.

Queste possono essere manderate per un tempo limitato. Alla fine di ogni livello di privilegio precedente le informazioni sono salvate nella memoria. Non ci sono solo le interruzioni esterne ma anche interne generate in presenza di errori. Le modalità di gestione non cambia. Le interruzioni possono essere generate anche tramite software che permettono di realizzare il meccanismo multiprogrammato: permettono di cambiare il programma in esecuzione e cambiare livello di

→ l'esecuzione della CALL è un salto incondizionato come la JUP con la differenza che quando si fa RET si deve tornare all'istruzione successiva alla CALL. Per questo utilizziamo la pila. La CALL fa quindi una push di IP nella pila e poi cambia il contenuto di IP. Il fetch successivo permetterà di eseguire il sottoprogramma. Al momento della RET il contenuto della pila deve avere lo stesso iniziale. La RET fa IP.

Il gestore è simile ad un sottoprogramma ma la RET deve ritornare ad eseguire il programma da dove era stato interrotto. La CPU può accettare o non accettare le interruzioni. È ragionevole accettare le interruzioni in modo da non interrompere il ciclo della CPU. Si rende l'esecuzione di un'interruzione atomica. Quando la CPU accetta l'interruzione

possiamo individuare in memoria l'enumerazione fatta dal bootstrap a cui si vuole eseguire l'INT. Il PSW ci dice di accogliere anche le interruzioni.

Entrambe le informazioni sono salvate nella memoria. Non ci sono solo le interruzioni esterne ma anche interne generate in presenza di errori. Le modalità di gestione non cambia. Le interruzioni possono essere generate anche tramite software che permettono di realizzare il meccanismo multiprogrammato: permettono di cambiare il programma in esecuzione e cambiare livello di

privilegio. → Protezione: modo utente e kernel permettono di fare cose diverse e
dichiarare INTR estore

Classificazione delle architetture

2 punti di vista per classificazione: avere più flussi di esecuzione in parallelo, dove il flusso di esecuzione è detto dell'esecuzione del programma (istanza del programma). Se abbiamo 1 processo c'è un solo flusso di esecuzione. La struttura così si può dire dei dati. Il flusso di esecuzione può elaborare un singolo set di dati oppure più flussi di dati.

	↓ Single
Si :	Single Instruction Stream
Mi :	Multiple Instruction Stream
SISD :	Single Data Stream
MIMD :	Multiple Data Stream

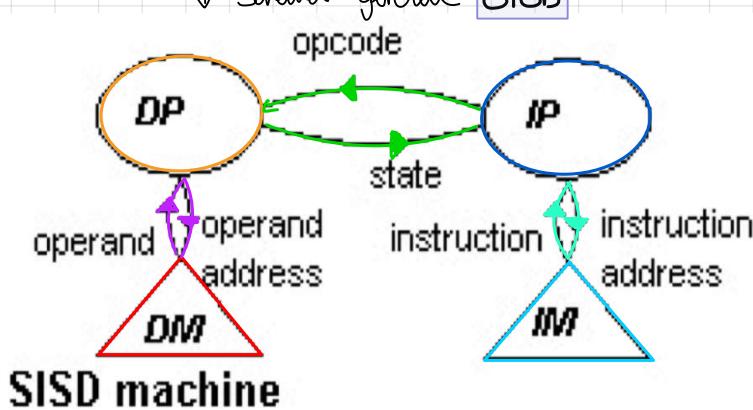
→ possibili → SISD, MISD, SIMD, MIMD

Organizzazione generale: tutte le macchine moderno in architettura

Cerchi: componenti attive sistema di elaborazione

Triangoli: gerarchie di memoria: rappresentano il rapporto tra capacità e velocità

taxonomia di Flynn



→ Instruction Processor: nel processore chiamiamo unità operativa di controllo con alcuni registri anche non visibili al programmatore. Nell'unità operativa chiamiamo la ALU e i vari registri. Lì IP rappresenta l'unità di controllo in quanto IP è l'elaboratore delle istruzioni che si occupa di fare fetch e decodifica e calcola istruzione successiva

→ Data Processor: Unità operativa in grado di eseguire l'istruzione comune nell'elaborare i dati

→ Bus: 2 bus

- opcode: IP invia opcode a DP
- state: IP restituisce ad IP una serie di informazioni (ad esempio flag)

→ Instruction memory: Siamo abituati a vedere alla memoria centrale come unica contenente istruzioni e dati. Qui dividiamo in quanto nella realtà dati e istruzioni rimangono nettamente separati. C'è inoltre una gerarchia per le istruzioni dato che i programmi stanno in memoria permanente

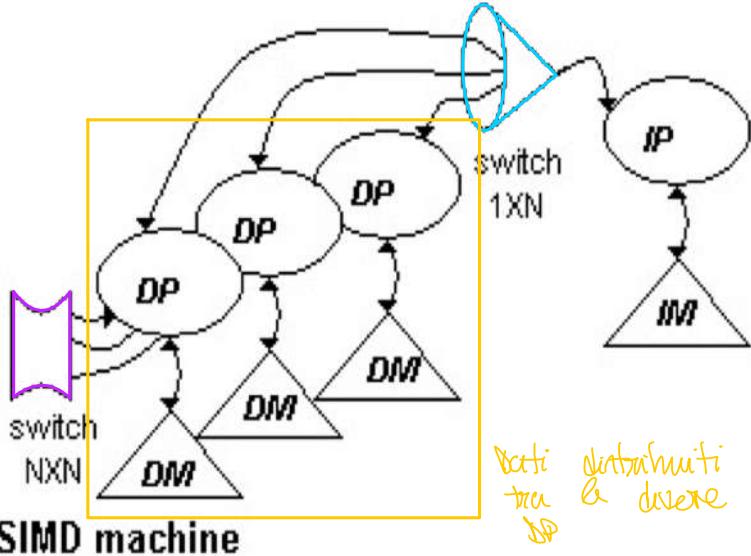
→ instruction address: fatto durante la fetch
instruction: restituita da fetch

→ Trasformazione di operandi (a ux mu dati, a bx mu indirizzi)

L'architettura è tra le più veloci nello schema sulla separazione delle memorie (l'importante separazione currente velocità) e permette di standard. Nelle architetture che usiamo e il mu è uno solo. Le macchine lavorano isolate. In queste macchine si esegue un solo programma che lavora su un solo flusso di dati.

dato quello che avevamo detto per sistemi embedded) → la covere tempi di esecuzione la separazione è solo logica SISD, come si vede nello schema

Machine SIMD



eseguite tenendo vantaggio del parallelismo nei dati (app. scientifiche)

→ **Switch NxN:** Non tutte le operazioni che sono la macchina parallela funzionano se le DP non comunicano fra di loro (vediamo matrici) per comunicare le IP permette di eseguire in modo parallelo anche app penate per l'esecuzione sequenziale.

Topologie di connessione regolari o create ad hoc in quanto il bin non deve mai saltare per non dare problemi di sincronizzazione. Comunicazioni regolari non creano conflitti, sono efficienti e poco costose.

Supercomputer, macchine scalari → Modello computazionale sincrono: una unica unità di controllo con DP che eseguono stessa intrattive. Alcune spingono verso dell'parallelismo temporale in cui le DP eseguono solo una parte ed ogni DP esegue una fine diversa (moltiplicazione). Questo è efficiente se riusciamo a riempire la pipeline. In contrapposizione utilizzano il parallelismo spaziale ove tutte le DP lavorano su dati tutti uguali. Alcune macchine possono usare entrambi i parallelismi → Supercomputer vettoriali, vector processor pipeline (tempo), Array process (matrice), Systolic Array (entrambi).

↓ Esempio



Diversità di funzionamento tra un processore scalare ed un processore vettoriale

Esempio

$c = a + b;$

Processore scalare: gli operandi sono scalari

Processore vettoriale: gli operandi sono vettori



Compilatore vettoriale

Esempio

$\text{int } i = 0;$

$\text{for } (; i < 10; i++)$

$\quad c[i] = a[i] + b[i];$

Riconosce tutti quei cicli sequenziali trasformabili in un'unica operazione vettoriale

Macchine MIMD

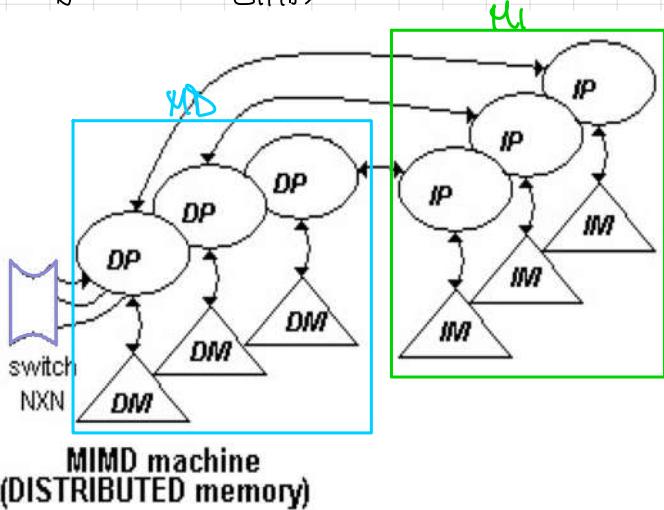
più fluvi di istruzioni paralleli lavorano su unico flujo di dati → Macchine di questo tipo oppure mettiamo macchine che usano CPU pipeline anche se c'è una classificazione troppo forte

→ **Switch:** Binario generico in quanto rappresenta ogni sistema di interconnessione → Ci sono in quanto la macchina ha parallelli sui miei dati. Si ha una rete IP e molte DP concorrenti con una propria IP. Tutte le AW eseguono la stessa intrattiva parallelamente e per questo si ha uno switch e a sua volta propone l'opzione a tutte le DP. Questo è utile ad esempio per lavorare sulle matrici ed in generale per operazioni che possono avere dati (app. scientifiche)

per operazioni che possono avere dati (app. scientifiche)

per operazioni che possono avere dati (app. scientifiche)

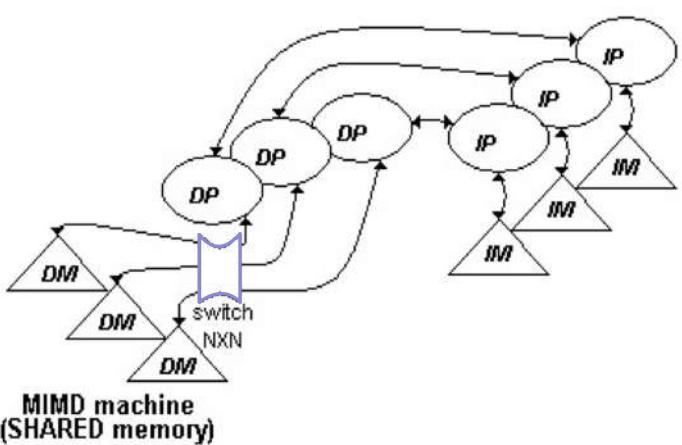
Macchine MM-MIMD: flusso multiplo di istruzioni su flusso multiplo di dati. Si hanno più unità di elaborazione con più IP e DM. È importante che ci sia comunicazione tra le varie unità di elaborazione. Per questo si deve introdurre una rete (+ da macchine SIMD)



ha un suo flusso di esecuzione di istruzioni. Abbiamo scambio di dati e flussi multipli di istruzioni operano su unità memoria condivise e per switch NXN.

Multi computer: computer connessi tra di loro tramite una rete reca, a differenza di internet, uno switch regolare e più performante. Cambia la latenza introdotta ma rispetto ad internet non cambia nulla. Per trasferire i dati su una rete, il servizio peering ed il modello funziona indipendentemente dal modello della rete. I sistemi godono di un'elevata scalabilità per la quale è sufficiente aggiungere nuovi nodi (la connessione alla rete è facile) anche lato server.

Macchine SM-MIMD



ogni unità ha un suo flusso specificamente per queste macchine multi-core succede in questa limitato perché il collegamento è fatto del basso. La capacità della regolarità dello switch NXN.

↓ confronto SIMD

SIMD	MIMD
meno hw	meno hw
meno memoria	meno memoria
l'cpu,	cpu che non vede per

Le SIMD hanno meno hardware mentre le MIMD hanno processori paralleli più elevati per le MIMD.

→ Ogni IP è collegato ad una ed una sola IM (privata). Per permettere la comunicazione su una rete switch NXN che collega tra di loro le IP che possono scambiarsi messaggi tra di loro

Memoria distribuita: privata tra copie IP-IP. → Le singole copie sono macchine SISD. Qualunque rete di calcolatori è una macchina SIMD: ciascuna coppia IP-IP è indipendente in quanto è comandata da sé. Si hanno più flussi di dati. Tra i nodi non hanno wave lo

→ La memoria pur essendo distribuita è comdivisa e lo switch la rende visibile come un'unica memoria. Le porzioni della memoria possono essere usate per scambiare informazioni. Questo introduce problemi di mutua esclusione. Queste macchine sono molto più complicate dei multiclienter multi-processori → Lo switch NXN deve essere molto potente per non perdere i vantaggi. Le applicazioni

per queste macchine → Tutte le categorie anche se in modo limitata

general-purpose.
molto sentita

METRICHE

Speed-up (S): Accelerazione → Confronto il tempo di esecuzione rispetto alla macchina uniprocesso: N : numero di processori. Se l'app può beneficiare dei più processori avremo $T_N < T_U$. Il speedup teorico è N : non lo raggiunge quasi mai perché dipende dall'applicazione. Se più essere diversi i funzioni indipendenti non bisogna di comunicazione e quindi $T_U = \sum T_{indip}$ e di conseguenza $S \approx N$. Mi avvicino di più se le funzioni vengono eseguite nello stesso tempo. Se c'è necessità di comunicazione allora S rimane più lontano da N in quanto questa riduce tempo (vedi spazio I/O) con la quale posso utilizzare la macchina.

Efficienza (ξ): Efficienza S/N

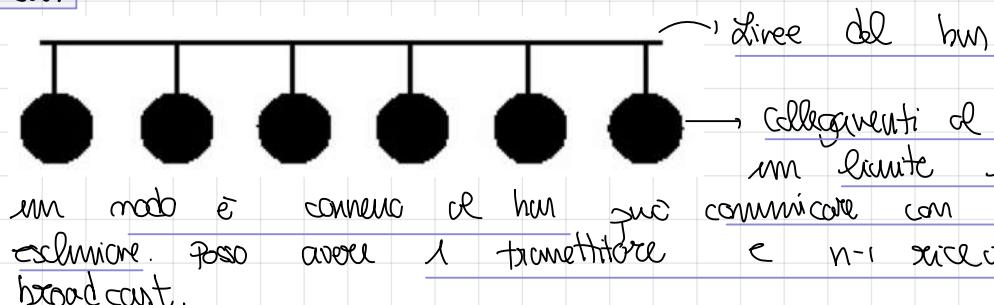
Legge di Amdahl: // perfetto non esiste. Ogni parallelizzabile parte regolare eseguita in tempo T_{seq} . anche il tempo di comunicazione

$$S = \frac{T_U}{T_U + \frac{N-1}{N} T_{seq}} \xrightarrow{N \rightarrow \infty} S = \frac{T_U}{T_{seq}}$$

non parallelizzabile parallelizzabile

TOPOLOGIE DI CONNESSIONE

bus



un modo è connuto del bus può escludere. Posso avere 1 trasmettitore e $n-1$ ricevitori e quindi è broadcast.

Collegamenti del bus: è espandibile fino ad un limite superiore di nodi. Quando comunicare con gli altri una in mutua esclusione. Possiamo avere 1 trasmettitore e $n-1$ ricevitori e quindi è broadcast.

Parametri:

- grado = 1 → Introduce meccanismo per avere connuti del bus
- diametro = 1 → Quanti link sono necessari per raggiungere tutti i nodi: hop = distanza tra trasmettitore e ricevitore. Se c'è un modo che trasmette l'informazione viene trasmessa a tutti → con la stessa velocità → hop = 1 (tutti ricevono)

totale link = 1

⚠ Scavaggio: **waking competition** → Mutua esclusione con meccanismo di arbitraggio per problema elettrico

array lineare



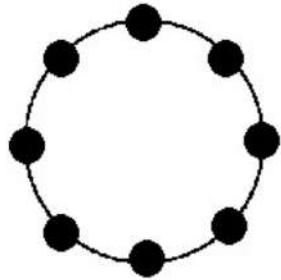
Parametri:

- grado = 1 per il esimo nodo, 2 per gli altri
- Diametro = $N-1$ (\Rightarrow bus)

⚠ Riduce competizione: $\frac{N}{2}$ trasmettitori contemporanei + routing

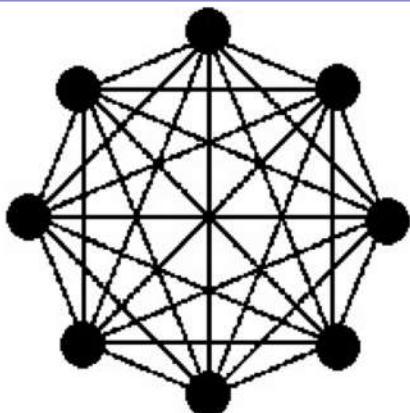
⚠ No tolleranza nei guasti: per comunicare tra tutti i nodi deve rate viere partita e guasto e la comunicazione rimane attiva al suo interno

ring



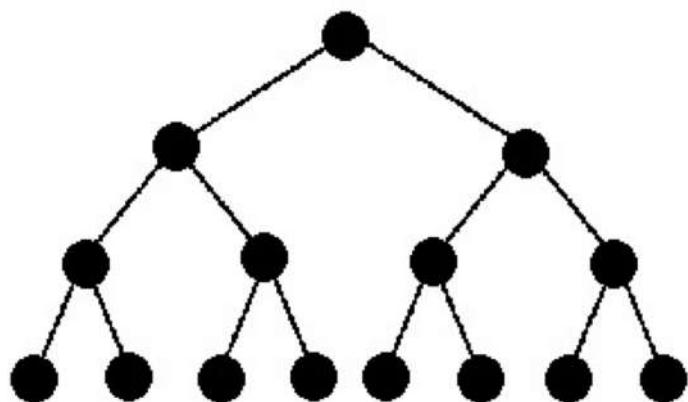
Grado = 2
Diametro = $\lceil \frac{n}{2} \rceil$
totale link = N
Tolleranza a guanti: 1

Connessione completa



Grado = $N-1$
Diametro = 1
totale link = $\frac{N(N-1)}{2}$

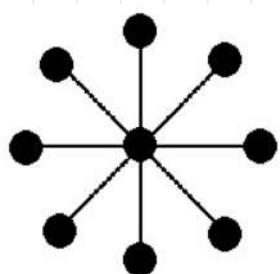
B-Tree



Altezza $h = \lceil \log_2 N \rceil$
Grado = 2 radice, 1 foglie, 3 altri nodi
Diametro = $2 \cdot (h-1)$
totale link = $N-1$
Guanti: radice non può quantarci perché tutto vi ha un posto sulla radice

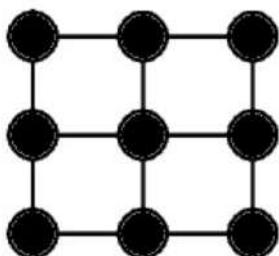
Link asimmetrici: a livello più alto i link sono più congestionati e possono fare da bottleneck: si evita i fat tree

Star



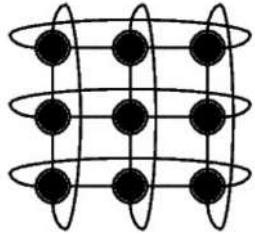
Grado = $N-1$ centrale, 1 altri
Diametro = 2
totale link = $N-1$
Poco tollerante a guanti per node centrale

Mesh 2D



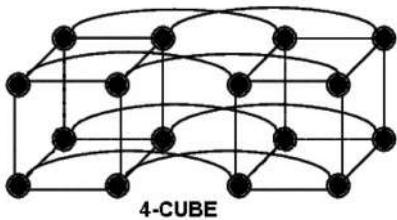
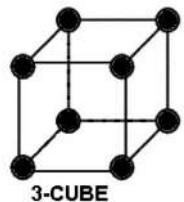
$n = \sqrt{N}$
Grado = 2 vertici, 3 centrali lati, 6 altri
Diametro = $2 * (r-1)$
totale link: $2 * n - 2 * r$
Buona tolleranza a guanti: multipath

Torun



grado = 4
diametro : $2 \cdot \lceil r/2 \rceil$
totale link = $2N$
scala bene

Ipercubo



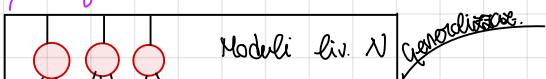
dimensione = d
 $N = 2^d$
grado = d
diametro = $\log_2 N = d$
totale link = $d \cdot N/2$
Scalabile: utilizzata per multi processi

Struttura del SO

(sistema multi programmato
uno processore)

Narcano come i interni monolitici SW → vengono divisi a blocchi

→ interfaccia macchina virtuale liv. N: API



interfaccia macchina virtuale liv. i

Moduli liv. i

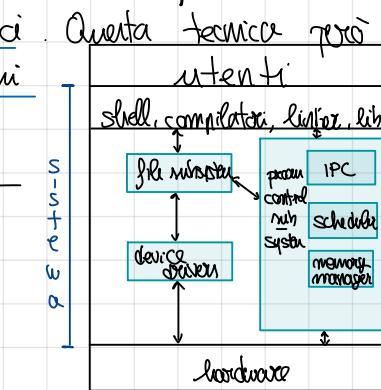
interfaccia macchina virtuale liv. 1

Moduli liv. 1

Macchina fhw

gerarchia

strutture



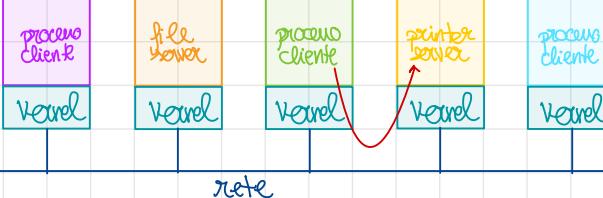
è indicativa della complessità del

interfaccia system - call

connette service provider e service user



client-server



gestione dei processi

processo = sequenza di eventi (sequenza statica). In un sistema multi programmato l'esecuzione di un processo è influenzata anche dagli altri processi → processo = unità di esecuzione in SO multi programmato → più processi vengono eseguiti in modo concorrente

c'è una relazione di ordine e di causalità: se varia lo stato delle variabili del programma posso ridurre allo stato del processo partendo dalla conoscenza dello stato iniziale (variabili di ingresso). L'istante di un processo è dato dall'esecuzione di un programma con determinati dati. Se questi cambiano, cambia anche l'istante.

↓ Rappresentazione processo

codice del programma: in un interno

dati: sotto forma di file in HDD

Program Counter: Indirizzo prox. istruzione

Registri: tutti i registri della CPU → contenuto = stato di progra

Stack: ci serve per CALL e RETURN

+ per esecuzione: RISORSE

Memoria: principale → Assegnazione gestita dal SO e non implicita nel programma

File aperti: uno scritto esplicitamente nel codice → risorse logiche e nei interni UNIX anche (periferiche)

dispositivi di I/O

↓ Memoria

stack

heap

data

text → Codice

→ Allocazione dinamica di memoria

Stati

Sistema monoprogrammato

Esegue istruzioni

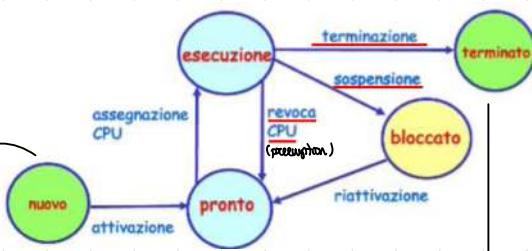


Attenza evento: trasformazione di I/O

ci serve il SO a riconoscere il risveglio

Sistema multiprogrammato

Processo appena creato: al login del SO viene avviato processo per gestore della sessione dell'utente (shell) in attesa di qualche comando. La shell è padre di processi figli creati per eseguire comandi



Processo non ha più istruzioni da eseguire (HIT): le risorse devono essere rilasciate e sync tra padre e figlio

Viene creato descrrittore, riempito ed assegnata memoria: attivazione → finita a pronta

Lo scheduler viene attivato ogni volta che c'è una freccia uscente da executive. Il processo rilascia spontaneamente la CPU in caso di terminazione e sospensione. Il caso di revoca è una revisione che non è invece spontanea. C'è anche un altro caso in cui viene chiamato lo scheduler: entro processo in coda pronta per eventuale preemption

↓ struttura dati

Descrittore: nome del processo → binario da SO (intoco in UNIX: pid)

Stato → Quelli sopra: può anche essere bloccato dalla lista dei descriptori. Modelli di servizio dei processi → Algoritmi che hanno scheduling in priorità. Informazioni sulla gestione della memoria → SO deve sapere dove e dove è allocata memoria virtuale processo

Coda

Contesto →insieme dei registri della CPU

Utilizzo delle risorse → Quelli sono i file aperti (puntatori)

Identificazione processo successivo → Creazione liste

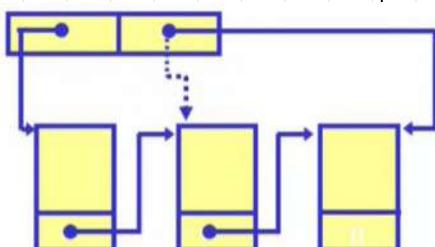
↓ è possibile avere

Tabella dei processi:

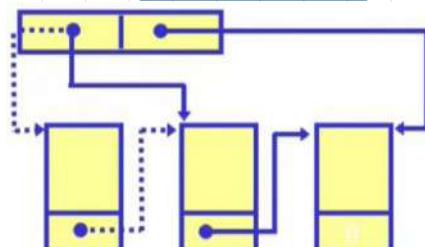
contiene tutti descrittori

perché morti di frequente

→ C'è sicuramente una coda pronta (ce ne fanno essere anche di più organizzate per priorità), una o più code di processi bloccati. È importante mantenere un puntatore al descrittore del processo in executive → Registro



Inserimento di un descrittore

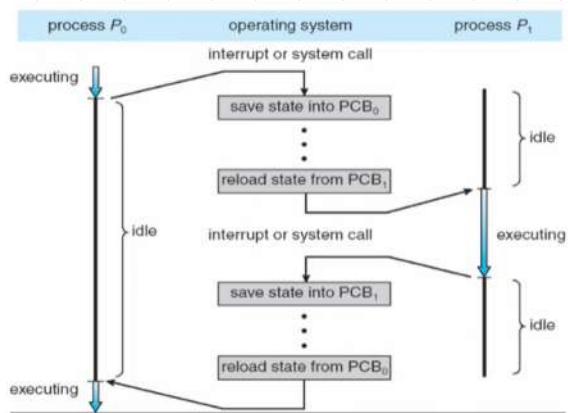


Prelievo di un descrittore

Cambio contesto: uso della CPU viene commutato da un processo ad un altro

↓

1. Si lancia stato processo in exec: copia registri da CPU a memoria
2. Descrittore in executive si mette nella coda giusta
3. Riconfigurazione CPU: salvataggio puntatore a processo in executive e selezione nuovo da coda pronta del pool dello scheduler
4. Cambio stato: operazione dunque a salvataggio stato



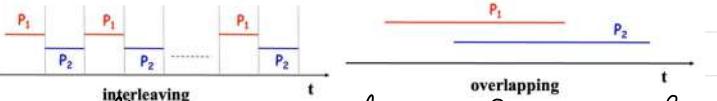
⚠: Il cambio di contesto c'è un'interruzione intorno che chiama una priorità

qui richiedere trascorsi orari anche per la reorganizzazione della memoria anche con l'utilizzo della memoria secondaria

In un sistema operativo abbiamo una struttura gerarchica di processi avendo una radice e vari livelli di fogli: informazione mantenuta in memoria per poter fare sincronizzazione

Interazione fra processi: processi concorrenti → Se avessi tanti processi quanti processi questi verranno eseguiti contemporaneamente / parallellamente. In un sistema multiprogrammato ne viene eseguito uno alla volta cavando come risultato l'equivalente del parallelo.

Dai processi si dicono concorrenti se la prima operazione di uno comincia prima che termini l'ultima dell'altro. →



Saranno due forme di interazione: concorrenza → competizione → implicita, non presente nel programma. Lo stesso vale anche per la memoria

Processi indipendenti: P1 e P2 non si influenzano nella loro esecuzione → proprietà della riproducibilità: non possibile per competizioni viste prime

Processi interagenti: opposto di indipendenti → effetto delle non indipendenze dipende dalla velocità dei processi: comportamento non riproducibile

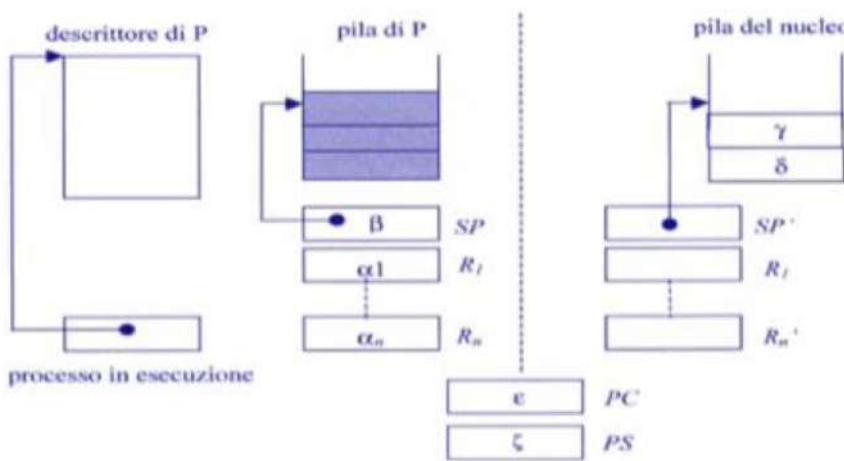
Competizione: uno d'risorse come che non possono avere niente contemporaneamente (mutua esclusione) → code di processi che ne fanno richiesta. Come ad esempio nel caso delle risorse logiche o variabili condivise. In alcuni casi c'è bisogno di atomicità per non portare variabili condivise in stato inconsistente.

Cooperazione: processi vogliono scambiarsi informazioni → sequenze singole oppure interagiscono con contenuto. In ogni caso richiede coordinamento: rapporto di causalità tra i processi.

Mechanismi di sincronizzazione: competizione → mutua esclusione: monotonizzazione indiretta o implicita
cooperazione → le operazioni tra produttori e consumatori devono seguire in ordine specifico: sincronizzazione diretta o esplicita

Il nucleo ha bisogno per realizzare l'estrazione della CPU: funzioni di risposta ad interruzioni, il meccanismo per il cambio di contesto e le system call (creare / terminare e sincronizzazioni)

Riduzione overhead



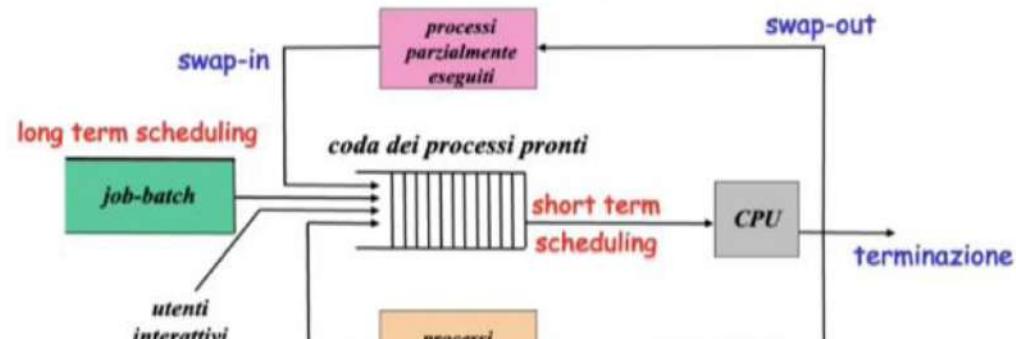
Scheduling (Scelta fra più processi con determinata politica per assegnamento CPU e acciaccamento in memoria centrale)

- ✗ breve termine: Estrae processi da lista pronti ed assegna CPU
- ✗ medio termine: Swapping, trasferisce da mem. principale a memoria secondaria e viceversa mem. virtuale processo
- ✗ lungo termine: definisce i processi attivi. Uno scheduling di questo tipo garantisce una esecuzione equitativa tra processi CPU-bound e I/O-bound. Se avessi troppi processi CPU-bound avrei troppa competizione e inutilizzo di I/O → Determina grado di multiprogrammazione: nr. processi attivi + istante. Se troppo buro la CPU è usata in modo poco efficiente. Aumentando le prestazioni della macchina migliorano

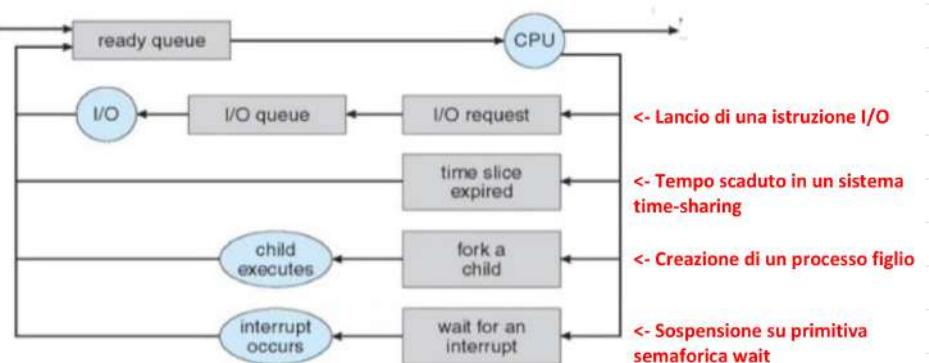
Scheduling di breve termine: Possono avere non-preemptive (senta diritto di revoca) e preemptive. Il primo intavola quando un processo termina o si sospende. Nel secondo può intavolare anche nel caso di revoca nel caso di timeout o di nuovi processi in coda pronti. Avendo invocato molto frequentemente questo deve avere molto veloce per ridurre l'overhead

↓ Recap

medium term scheduling



A
l
g
o
r
i
t
h
m



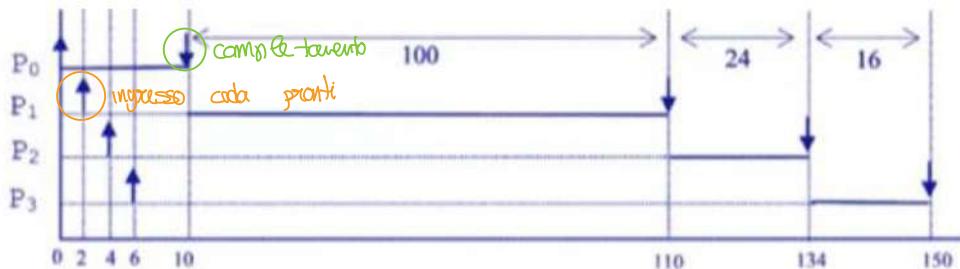
valutati con clare metriche: utilizzazione della CPU, escluso come % nel tempo; tempo medio di completamento (turnaround time), prendendo come t_{inizio} la prima entrata del processo in coda pronti e considerando come t_{fine} l'ultima di fine del processo. In un sistema monoprogrammato, CPU burst = turnaround time perché ciò non è vero nel caso multi-programmato. Si calcola T_{ave} = t_{fine} - t_{inizio} e si prende la media di questi tempi (T_{ave} medio). Se invertiamo il turnaround medio otteniamo il numero di processi completati in un'unità di tempo: throughput rate. Abbiamo poi il tempo di risposta: $\Delta t_{completamento}$, t_{ing. coda pronti} → tolleranza per utenti multi-programmati ($t_r > CPU\text{-BURST}$). Si ha poi il tempo d'attesa che indica quanto il processo rimane in coda pronti: $T_w = \sum t_{w,i}$ dove i dipende dello scheduling che se non preemptive vale 1 mentre può essere > 1 se preemptive. Tutti i fai fanno riferimento al primo ingresso in coda pronti. Si ha infine il rispetto dei vincoli temporali.

Algoritmi non-preemptive:

FIRST-COME-FIRST-SERVED (FCFS)

La coda pronti è gestita con politica FIFO e lo scheduler preleva sempre dalla testa

Processo	Istante di arrivo	Durata del CPU burst
P ₀	0	10
P ₁	2	100
P ₂	4	24
P ₃	6	16



A 10 calchiamo solo P0 in coda pronti e viene eseguito fino a completare il CPU-burst all'istante 10. Nel frattempo nella coda entrano gli altri processi. Questi non vengono eseguiti in quanto P0 non ha finito. Durante l'esecuzione di P0 vengono eseguite le routine per inserire i processi in coda pronti (nel grafico non presenti perché troppo vicino l'overhead). All'istante 10 P0 termina e FCFS preleva il processo in testa (P1). Questo viene ripetuto per tutti i processi in coda.

$\Rightarrow 150$ unità di tempo per overhead (2, 4, 6, 10, 110, 134) con gli ultimi 3 \Rightarrow per la presenza del cambio di contesto. La CPU viene usata al 100%.

Calcoliamo il turnaround: $T_0 = 10 \rightarrow$ Medio = 60

$$T_1 = 110 - 2 = 108$$

$$T_2 = 134 - 4 = 130$$

$$T_3 = 150 - 6 = 144$$

Tempo di attesa: $T_{w,0} = 0 \rightarrow$ Medio = 60.5

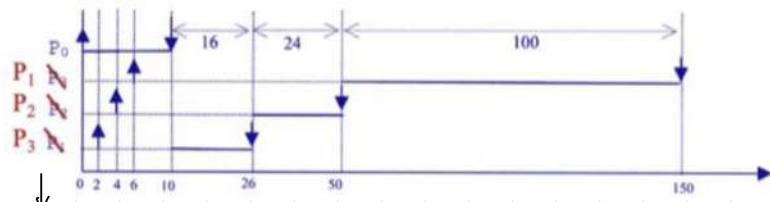
$$T_{w,1} = 8$$

$$T_{w,2} = 106$$

$$T_{w,3} = 121$$

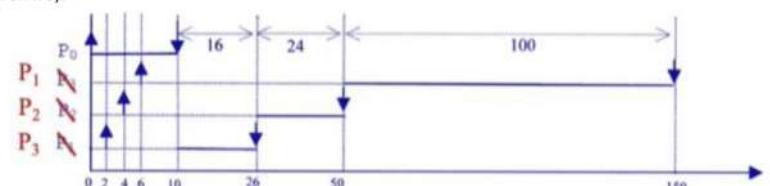
Se confrontiamo T_{ave} con i CPU-burst notiamo una percentuale sufficiente per P1 ed insufficiente per gli altri. Considerando invece

il tempo di attesa si ottengono le stesse considerazioni. FCS è quindi efficiente dato che la bandita delle non è fluttuante. Possiamo rafforzare questa l'ordine di arrivo



P₃ a t = 2
P₂ a t = 6
P₁ a t = 6

è necessario cioè una priorità
SHORTEST-JOB-FIRST (SJF): priorità statica



A t=0 c'è solo P₀ che viene eseguito. A t=10 lo scheduler guarda il CPU-burst che **! deve** cominciare ed assegna una priorità tanto più alta quanto più è breve il CPU-burst. Si ottiene un turnaround = 55 e un taw = 18.5. Il per il calcolo della capacità di decidere indipendentemente dall'ordine (dunque turnaround e taw) con una complicità comunque bassa (poco overhead). Si deve potere sempre stimare il CPU-burst per applicare questo algoritmo (possibile nei sistemi special purpose).

↓ Come stimiamo il CPU-burst?

Si usa la stima esponentiale: si basa su una stima iniziale aggiornata con la storia delle esecuzioni.

t_n = durata CPU-burst n-esimo

s_n = stima n-esima $\rightarrow S_{n+1} = \alpha t_n + (1-\alpha)s_n$

α = numero 0 < α < 1 $\rightarrow \alpha = 0$: tutte le stime uguali a quella iniziale

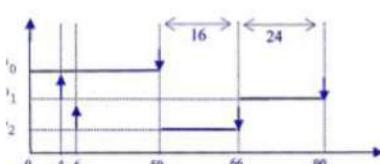
$\alpha = 1$: $S_{n+1} = t_n$ → non tocca conto della storia

tipicamente $\alpha = \frac{1}{2}$

Algoritmi preemptivi: **SHORTEST- REMAINING-TIME-FIRST (SRTF)**: priorità dinamica

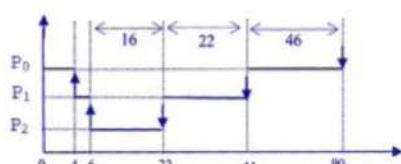
Processo	Istante di arrivo	Durata del CPU burst
P ₀	0	50
P ₁	4	24
P ₂	6	16

Turnaround medio: 65.3
Tempo medio di attesa: 35.3



SJF (Fig. 2.15)

Turnaround medio: 48.6
Tempo medio di attesa: 18.6



SRTF (Fig. 2.16)

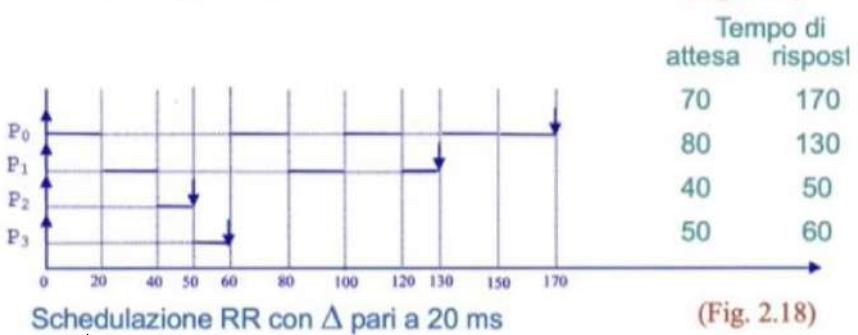
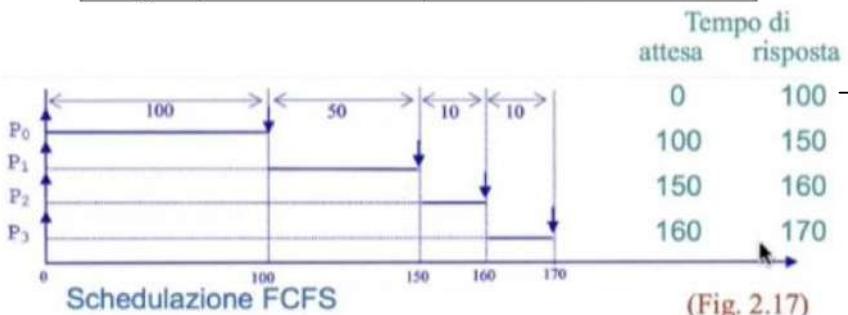
SJF non può fare nulla riguardo l'ordine in coda pronto e quindi funziona se ci sono più processi in coda. Per SRTF abbiamo l'esecuzione di P₀ a t=0 poiché unico presente. Quando compare P₁ calcola il remaining time di P₁ e P₀. Quando < quello di P₁

SRTF fa la preemption di T_0 e viene eseguito T_1 . All'arrivo di T_2 si fa la stessa cosa. Il cambio di contesto non viene fatto se i nuovi arrivati hanno priorità più bassa. L'overhead aumenta per rimanendo piccolo. Confrontando i tempi con SJF avremo un turnaround = 60.6 contro 55.3 ed un taw = 12.6 anziché 35.3. Ci arricchiamo quindi di CPU-burst più piccolo.

ROUND-ROBIN (RR)

Nei sistemi multi-programmati vi sono processi interattivi (multi I/O-bound). Si introduce RR che è un TCFS con preemption. Lo scheduler in questo caso fa un cambio di contesto ad un timeout prefissato interruzione con timer esterno.

Processo	Istante di arrivo	Durata del CPU burst
P_0	0	100
P_1	0	50
P_2	0	10
P_3	0	10



A to i processi sono in coda pronti e RR prende T_0 perché rispetta la coda. C'è un timeout di 20 ms e RR va in esecuzione allo scadere di questo timeout o al termine del processo. Se la coda pronta è vuota non vi fu cambio di contesto. Notiamo una decrescita di entrambi i parametri medi con un piccolo overhead. Al crescere del numero di processi questi devono fare sempre un ciclo di attesa iniziale prima di essere eseguiti con RR a lungo termine privilegia i processi brevi. Se riduciamo il timeout aumenta l'overhead dato dai cambi di contesto.

↓ Diminuendo Δ



giri: $(n-1)\Delta$ attesa. RR è utilizzato nei sistemi interattivi e è particolarmente adatto a quelli real-time.

→ Valori medi decrescono ma andare sotto 10 ms non è una buona idea poiché vengono penalizzati anche i processi brevi che devono fare più

interruzioni.

SCHEDULAZIONE a CODE MULTIPLE

Combiniamo insieme gli algoritmi appena visti.
Abbiano più code pronti (livelli) a cui è associato un livello di priorità (0,1). A ciascuna si associa un algoritmo di schedulazione. Quando si ha la priorità può succedere che il SO non dia mai la CPU ai processi con priorità più bassa (fenomeno di starvation). Per eliminarla si può stabilire una soglia per il tempo di attesa allo scorrere del quale si dirà la priorità: tecniche di aging.

Queste non sono necessarie per RR. Ma come applichiamo la priorità alle code? Nel momento della recita di cambio di processo si controlla che la coda con priorità più alta non sia vuota. Per la coda di livello 0 entro di nuovo FCFS per non bloccare tutto con processi lunghi. Se si usa RR per servire la coda di livello 1 solo quando la coda 0 è vuota. Quando si creano i processi deve avere stiato anche il livello 1a seconda dell'interattività: lv 1 per CPU-hand, lv 0 per I/O-hand). Si eseguono in background processi interattivi e in foreground processi CPU-hand. Ricorre sempre multilevel feedback. Tutte le volte che si crea un processo viene messo nella coda di livello 0 servita con RR per $\Delta = 10\text{ms}$. Se CPU-hand $\leq 10\text{ms}$ (interattivo) allora viene eseguito ed esce dal sistema mentre se $> 10\text{ms}$ nella coda di livello 1 servita con $\Delta = 50\text{ms}$. Se il suo CPU-hand è $>$ allora pone nella coda 2 servita con FCFS.

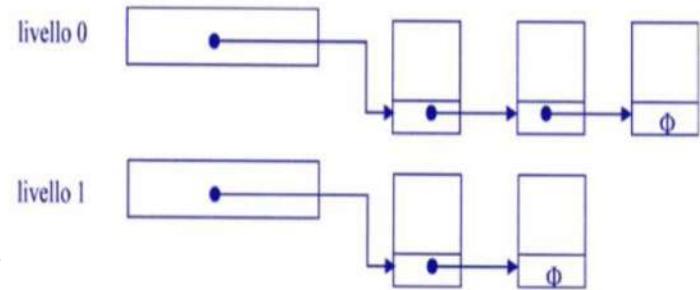
C'è ancora starvation fra le code se Δ è troppo grande \rightarrow lo svuotamento della coda 2 non è garantito. Questa può essere evitata inserendo uno preemptive mentre l'ultima viene quindi fatta preemption solo quando il CPU è libero, il che non tocca se abbiamo un sistema multi programmati. Anche sull'ultima coda viene quindi garantito che il sistema rimanga responsive. Non si ha il timeout perché se la coda pronta finge vuota il timeout sarebbe inutile. Il processo appena sospeso viene mantenuto nella coda iniziale (livello 2) in testa, in accordo con la politica FCFS. Questo permette di completare il processo n per volta. Nei SO possiamo avere anche più di 3 code mantenendo però sempre le funzionalità per cui il sistema è pensato.

SCHEDULAZIONE SISTEMI IN TEMPO REALE

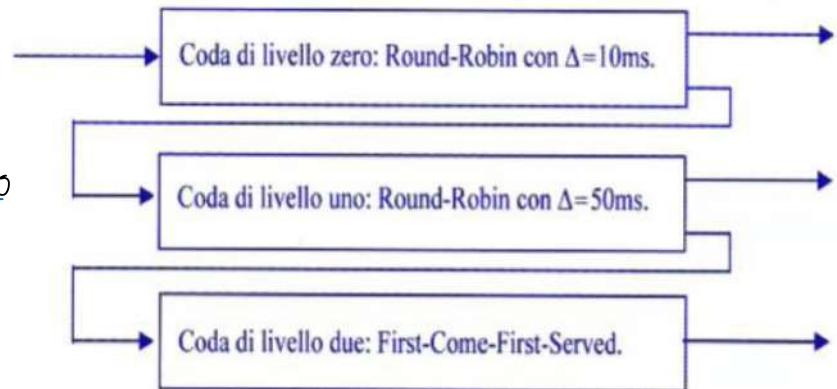
Sistemi special purpose, generalmente embedded \rightarrow interagiscono con mondo esterno tramite i sensori con un certo periodo di campionamento \rightarrow sistemi multiprogrammati (molti sensori) che svolgono task periodici e quindi il sistema rimane periodico. Il periodo di campionamento non è deciso dal programmatore ma è invece un requisito del sistema \rightarrow tempo reale: devono rispettare deadline = terminazione ultimo in cui il sistema deve aver completato tutti i task.

2 tipi:

- hard real time:** Nessuna deadline può essere violata (no overflow) \rightarrow si controlla se il sistema
- soft real time:** Qualche deadline può essere violata senza che il sistema vada in stato distruttivo \rightarrow sistemi multimediali (trasferimento dati con sequenze definite del media da trasmettere). se



può stabilire una soglia per il tempo di attesa allo scorrere del quale si dirà la priorità: tecniche di aging. Come applichiamo la priorità alle code? Nel momento della recita di cambio di processo si controlla che la coda con priorità più alta non sia vuota. Per la coda di livello 0 entro di nuovo FCFS per non bloccare tutto con processi lunghi. Se si usa RR per servire la coda di livello 1 solo quando la coda 0 è vuota. Quando si creano i processi deve avere stiato anche il livello 1a seconda dell'interattività: lv 1 per CPU-hand, lv 0 per I/O-hand). Si eseguono in background processi interattivi e in foreground processi CPU-hand. Ricorre sempre multilevel feedback. Tutte le volte che si crea un processo viene messo nella coda di livello 0 servita con RR per $\Delta = 10\text{ms}$. Se CPU-hand $\leq 10\text{ms}$ (interattivo) allora viene eseguito ed esce dal sistema mentre se $> 10\text{ms}$ nella coda di livello 1 servita con $\Delta = 50\text{ms}$. Se il suo CPU-hand è $>$ allora pone nella coda 2 servita con FCFS.



c'è ancora starvation fra le code se Δ è troppo grande \rightarrow lo svuotamento della coda 2 non è garantito. Questa può essere evitata inserendo uno preemptive mentre l'ultima viene quindi fatta preemption solo quando il CPU è libero, il che non tocca se abbiamo un sistema multi programmati. Anche sull'ultima coda viene quindi garantito che il sistema rimanga responsive. Non si ha il timeout perché se la coda pronta finge vuota il timeout sarebbe inutile. Il processo appena sospeso viene mantenuto nella coda iniziale (livello 2) in testa, in accordo con la politica FCFS. Questo permette di completare il processo n per volta. Nei SO possiamo avere anche più di 3 code mantenendo però sempre le funzionalità per cui il sistema è pensato.

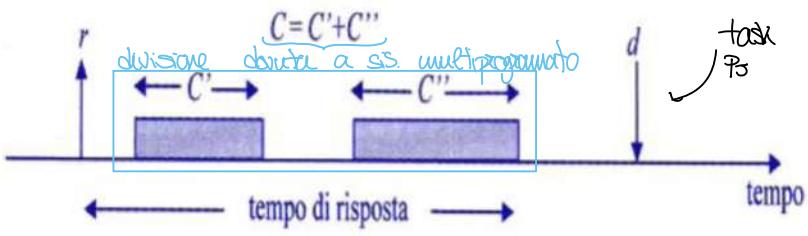
SCHEDULAZIONE SISTEMI IN TEMPO REALE

Sistemi special purpose, generalmente embedded \rightarrow interagiscono con mondo esterno tramite i sensori con un certo periodo di campionamento \rightarrow sistemi multiprogrammati (molti sensori) che svolgono task periodici e quindi il sistema rimane periodico. Il periodo di campionamento non è deciso dal programmatore ma è invece un requisito del sistema \rightarrow tempo reale: devono rispettare deadline = terminazione ultimo in cui il sistema deve aver completato tutti i task.

2 tipi:

- hard real time:** Nessuna deadline può essere violata (no overflow) \rightarrow si controlla se il sistema
- soft real time:** Qualche deadline può essere violata senza che il sistema vada in stato distruttivo \rightarrow sistemi multimediali (trasferimento dati con sequenze definite del media da trasmettere). se

le deadline non sono rispettate si ha un attivamento



r: istante di richiesta (task entra in coda pronto)
d: deadline ($d > r$)
periodo
Tempo CPU richiesto (C) per fornire risposta caratteristiche di P_i

C = tempo CPU di cui ha bisogno il processore: dipende molto da valori di ingresso, non dipende però dalla schedulazione, ma invece dipende dall'architettura

C = C_{max}: indica tempo richiesto in sistema monoprogrammato nel caso peggiore

Vogliamo semplificare il problema avendo più task. Tutti che un task è periodico si possono stabilire relazioni tra gli istanti di richiesta: $r_{i+1} = r_i + t_i$. Rilanciamo adesso i requisiti della deadline: $r_i \quad di \quad r_{i+1}$. Più di è lontano

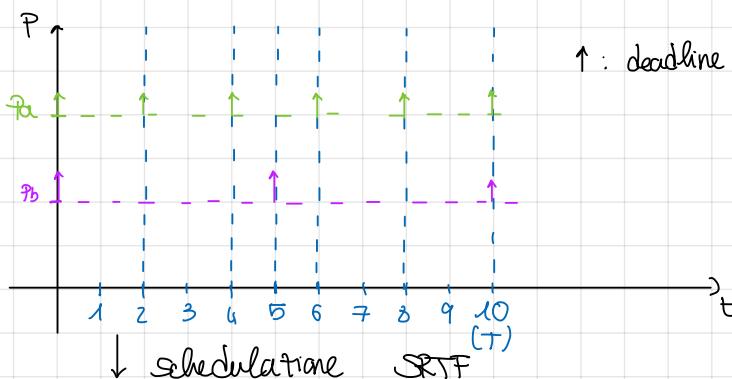
da r_i più è difficile fare lo scheduling. Per semplificare prendo sempre il tempo massimo e quindi pergo di = λt_i . Se ha infine, detta frequenza, che un sistema composto da elementi periodici è periodico con $T = \text{mcu}(t_i)$, $\forall i$

↓ Schedulazione

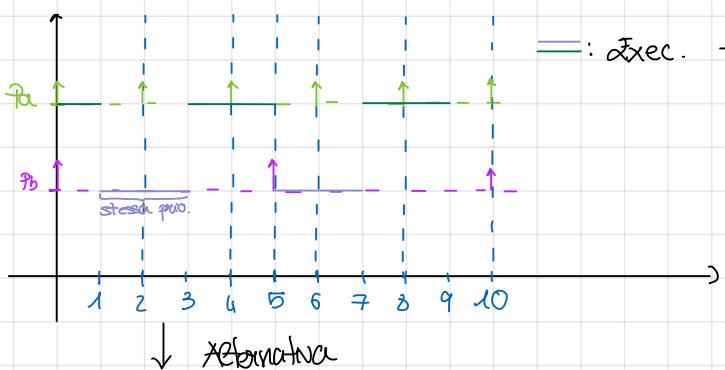
In sistema hard real time \rightarrow è un'esecuzione periodica di tutti i processi in modo che nessuna deadline venga violata. $\rightarrow C_{\max}, \exists i: di - ri \leq t_i$

Esempio: vogliamo schedulare fu: $t = 2$; $C = 1 \rightarrow T = 10$

Ph: $t = 5$; $C = 2$

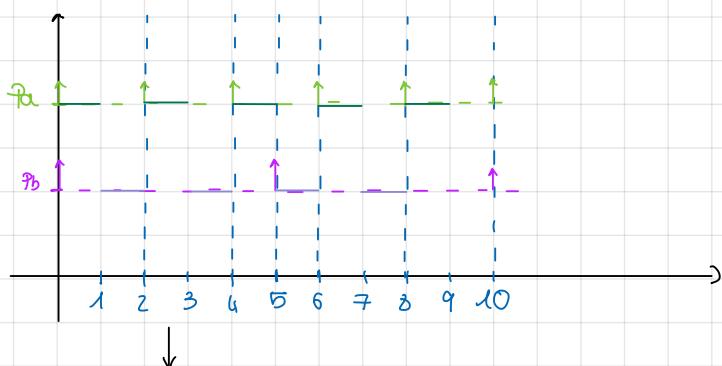


Non possono avere FCFS in quanto non preattive e RR perché vuole. Proviamo a pensare di avere SRTF ma questo richiede molto overhead e quindi è un algoritmo lento e noi vogliamo ottimizzare invece i tempi



Il sistema è schedulabile (no overflow). L'intervallo tra $a < 10$ è idle e potrebbe essere utilizzato per eseguire altro.

Vogliamo trovare un algoritmo che non faccia la schedulazione su C: utilizziamo **RATE MONOTONIC (RM)**: prioritario preattivo dove la priorità in funzione è data dall'inverso del periodo ($f = \text{rate}$). Nove la priorità in funzione del periodo tiene conto anche di C_{\max} in quanto $C_{\max,i} \leq di - ri \leq t_i$



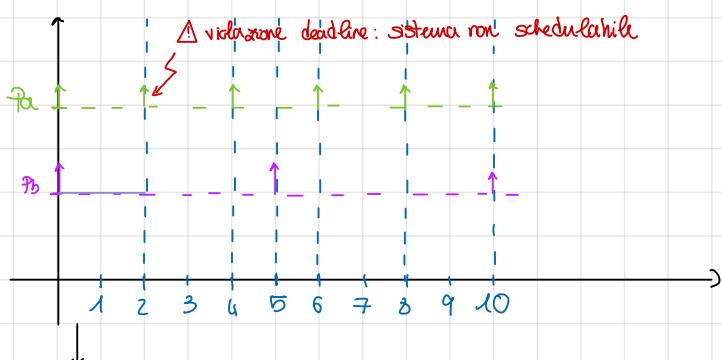
$$pa = (Pa) \rightarrow pa = \frac{T}{ta} = 5$$

$$nb = \frac{T}{tb} = 2$$

Impiego CPU

$$tu = naCa + nbCb = 5ta + 2tb = 5 \cdot 1 + 2 \cdot 5 = 15$$

A differenza di SRTF, RM ha priorità statica anche se in questo esempio semplice non si cogono i vantaggi di RM. Proviamo ad immaginare un algoritmo / pa (Pa) > pb (Pb)



RM è OTTIMO nella sua categoria (preemptive con pa. statica). Se un sistema quindi non è schedulabile con RM questo non è schedulabile con nessun algoritmo della sua stessa categoria. Se però no, è possibile che ci riconosca solo anche con altri algoritmi della stessa categoria.

⚠️ I sistemi multitasking possono ospitare processi asincroni (e devono poterli ospitare) solo se l'idle time è > 0. Questi devono soddisfare requisiti: % idle & presenti & T

Condizioni per la schedulabilità: Necessaria → Devo avere abbastanza tempo per finire le cose se vera posso RM ottimale (T). Devo quindi eseguire tutti i processi il numero richiesto di volte con tempo C per ogni esecuzione. → $T \geq \sum_{j=0}^{N-1} N_j C_j = \sum_j \frac{T}{f_j} C_j$

$$\text{U} = \text{fattore di utilizzazione} \quad \frac{\sum_j C_j}{\sum_j f_j} \leq 1$$

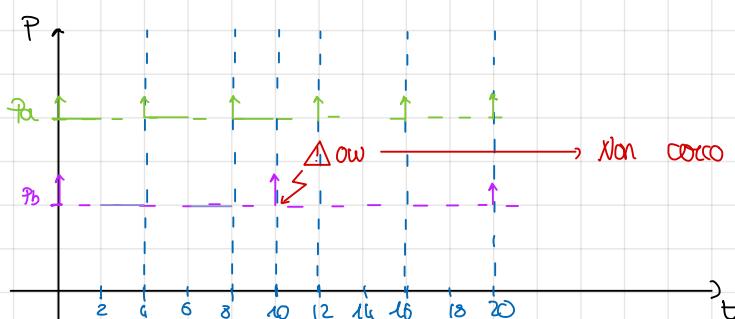
Sufficiente → $U \leq N (2^N - 1)$ dove $N = \text{numero dei processi}$
 ↓ limit → ∞
 70%

$$Pa: t = u; C_{MAX} = 2$$

$$Pb: t = 10; C_{MAX} = 5$$

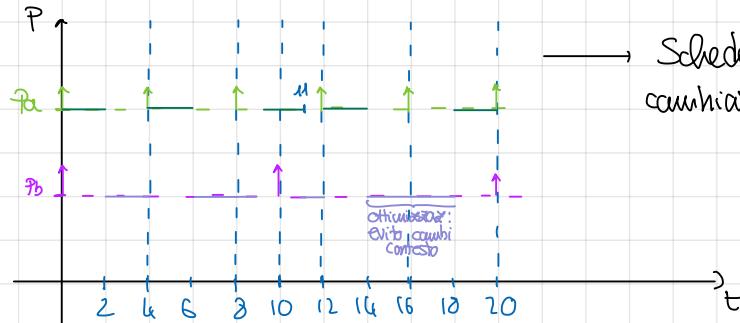
$$T = 20$$

$$U = \frac{2}{4} + \frac{5}{10} = 1 \rightarrow \text{CJ rispettata "a pelo"} \\ \text{CS non rispettata}$$



Priorità dinamica
preemptive

EARLIEST DEADLINE FIRST (EDF): Beicati \rightarrow a deadline più vicina (priorità dinamica)



→ Schedulabile con EDF: non ha quindi cambiare categoria

• CATEGORIE di PROCESSI

Processo = unità di elaborazione \rightarrow Stato CPU che esegue un programma.

Un processo è perciò anche caratterizzato da una memoria allocata. Il processo ha inoltre bisogno di risorse logiche e fisiche

→ processi pesanti (task): possiedono risorse e le gestiscono ad un determinato istante \rightarrow Se B vuole accedere alle risorse di A deve chiedere il permesso

→ processi leggeri (thread): assegnata la CPU \rightarrow condividono risorse

↓

Divisione porta vantaggi al livello di overhead: cambio contesto per task richiede svantaggio stato e risorse allocate. Per i thread è invece sufficiente salvare i registri: meno overhead. I thread condividono uno spazio di indirizzamento e quindi possono comunicare tramite strutture dati. I sistemi multithread possono prestarvi in diverse forme: presenti cerche intercorrenti a processi pesanti. Ogni thread deve avere il proprio task ed il proprio compilatore (analogamente a processi).

↓ Utilizzo

Utente: librerie permettono di scrivere app multi-thread. Da loro schedulazione non richiede il supporto del SO \rightarrow se non è multi-thread vede solo il processo pesante e schedula solo a livello di processo: con la preemption i thread vengono stoppati.

livello Kernel: Gestione supportata dal SO \rightarrow Schedulazione a livello di thread: più core - più thread di processi diversi o stesso processo.

Sincronizzazione di processi

3 tipi di interazione:

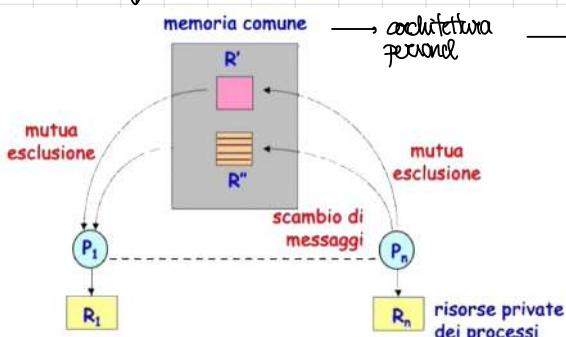
cooperazione = sincronizzazione

diretta / esplicita

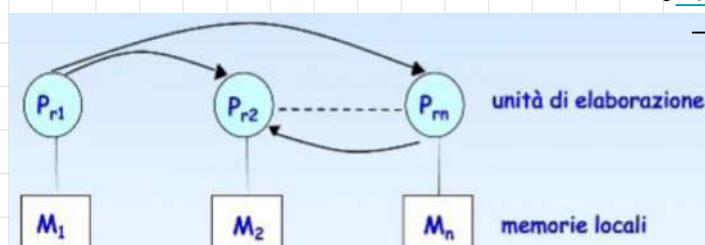
competizione = sincronizzazione

indiretta

interpenetrazione = eseguire su macchine diverse in momento diverso durante l'esecuzione → eseguire time-dependent



zona d'area 1: Memoria comune → Insieme di processi indipendenti con risorse private una che hanno area di memoria comune per fare sincronizzazione diretta. Può esistere una risorsa R' che deve essere mantenuta la mutua esclusione perché più processi vogliono farci operazioni (non importa ordine). Abbiamo un buffer R'' in cui un processo è produttore di messaggi ed un altro consumatore. In questo caso l'ordine delle operazioni è fondamentale.



→ Sistemi MIMD: Separazione volta tra memorie locali → Sincronizzazione: scambio di messaggi perché c'è buffer in quanto c'è memoria comune

PROBLEMA della MUTUA ESCLUSIone

T stack[n];
int top=-1; Vogliamo implementare $\frac{1}{2}$

```
void Inserimento(T y) {
    top++;
    stack[top]=y;
}

T Prelievo() {
    T temp;
    temp=stack[top];
    top--;
    return temp;
}
```

Possibile sequenza di esecuzione:

t0: top++;	(P1)
T1: temp=stack[top];	(P2)
T3: top--;	(P2)
T4: stack[top]=y;	(P1)

Inserimento e prelievo sono sezioni critiche: pezzi di codice che definiscono metodi in una struttura dati condivisa

Vogliamo capire cosa succede in base all'ordine di esecuzione in quanto gli errori si evitano stanno dipendentemente dall'ordine

Vogliamo rimuovere gli errori in ogni situazione: le sezioni critiche devono avere atomicità → Non lo sono a causa del meccanismo delle interruzioni. Il problema di mutua esclusione prevede l'atomicità ed il mantenimento di un ordine di esecuzione delle sezioni critiche (insert → extract ≠ extract → insert, non si generano errori ma cambiano programma) → Non ci occupiamo dell'ordine in quanto specificato dal programma, noi garantiamo solo atomicità. → Operazioni di inserimento e prelievo sono atomiche.

↓ Soluzione (esatta)

P1
acquisire la risorsa (A sezione critica)
prologo: while (occupato==1);
occupato=1; **<sezione critica A>**
epilogo: occupato=0;
Rilascio risorsa

P2
protegge la risorsa, non la sezione critica
prologo: while (occupato==1);
occupato=1;
<sezione critica B>
epilogo: occupato=0;

⚠️ Non funziona: P1 e P2 prendono bolzoni ad eseguirsi in momenti diversi prologo ed epilogo a causa dello scheduler.
Non c'è atomicità su occupato (condizionale).

Soluzione

Il prologo

NON

è atomico

lock(x):

```

TSL registro, x (copia x nel registro e pone x =1)
CMP registro, 0 (il valore di x era 0 ?)
JNE lock (se no, ricomincia il ciclo)
RET (ritorna al chiamante; accesso alla sezione critica)

```

unlock(x):

```

MOVE x, 0 (inserisce 0 in x)
RET (ritorna al chiamante)

```

Un hw su cui posso fare lock per garantire atomicità. La TSL non può essere interrutta e dopo l'esecuzione siamo certi che X=1: la usiamo per proteggere la risorsa. Se ora 0 la variabile è libera e è stata più volte fin quando non è 0. La lock non è atomicca ma il primo thread che la esegue e trova x=0 blocca tutti gli altri thread nella lock stessa.

↓ Nuova formulazione

P₁

prologo: lock(x);
<sez. critica A>;
epilogo: unlock(x);

P₂

prologo: lock(x);
<sez. critica B>;
epilogo: unlock(x);

test-and-set:

Istruzione atomica che accede alla memoria 2 volte: TSL registro, x

copia x nel registro e mette x=1.
 Fa quindi emergerà letteralmente il bus ed infine fa emergerà sul bus in notifica: Devo avere

⚠ Nella lock più avrai busy wait:
 Attendo che x diventi 0 → Ma per sistema multiprogrammato: accettabile solo per sezioni critiche brevi; altrimenti sistema inefficiente. Significa di avere memoria condivisa e

sistema multiprogrammato. Questi accadono alla risorsa x in memoria condivisa. La soluzione funziona poiché la lock richiede il blocco del bus che è unico per tutti i processi. Sol. **Semafori**: strutture dati definite da SO in cui sono definite 2 sezioni critiche. I processi possono utilizzarli solo tramite le primitive wait e signal (atomiche).

```

void wait(s) {
    if(s.value == 0)
        <il processo viene sospeso e il suo descrittore inserito in s.queue>
    else → processo non si sospende
        s.value = s.value - 1;
}

```

↓ processo sospeso
in coda semplice

```

void signal(s) {
    if(<esiste almeno un processo nella coda s.queue>)
        <descrittore del primo processo estratto e inserito in coda pronti>
    else
        s.value = s.value + 1;
}

```

→ S: struttura com
interv > 0 e
coda di processi

Semaforo bloccante: value = 0 inizialmente → processi bloccati e vengono inseriti in coda in ordine.

Vengono sbloccati solo dalla signal

Semaforo non bloccante: signal prima di wait

Value = nr processi max che possono accedere ad una risorsa. Se aumentiamo a value il valore n, i primi n processi che eseguono la wait non si bloccano, mentre si bloccano l'anti-ultimo. Questo verrà sbloccato solo con una signal. Se value = 1 non si sospende solo il primo thread che esegue wait.

⚠ wait e signal devono essere atomiche

↓ Semaforo specifico

mutex: inizializzato ad 1 → **prologo:** wait(mutex);
<sezione critica>;
epilogo: signal(mutex);

Risolve problema mutua esclusione: non c'è busy wait perché il processo viene sospeso nella coda del semplice

• PROBLEMA della CONDIVISIONE (sincronizzazione CES)

Coopera zione tra processi: produttore e consumatore. Del comunicazione avviene attraverso il buffer in memoria condivisa con 2 sezioni critiche per inserimento e prelievo messaggio. In questo caso l'ordine è fondamentale (messaggio non inserito in buffer prima, messaggio non prelevato da buffer vuoto).

↓ Possibile soluzione

2 segnalibri: spazio disponibile inizializzato a 1 per risolvere problema 1. L'inizializzatore ne dipende dal numero di puntatori presenti nel buffer iniziale. messaggio disponibile inizializzato a 0 per problema 2. bloccante per consumatore

```
// Processo produttore
do {
    <produzione del nuovo messaggio>; -> Producio il messaggio
    wait(spazio_disponibile); -> Verifico se lo spazio è disponibile (non lo è se il consumatore deve ancora leggere)
    <deposito del messaggio nel buffer>; -> Manipolo il buffer
    signal(messaggio_disponibile); -> Segnalo al consumatore che è disponibile nuovo contenuto
} while(!fine);
```

```
// Processo consumatore
do {
    wait(messaggio_disponibile); -> Verifico se ci sono cose nuove da leggere (se non ci sono significa che il produttore non ha ancora eseguito la signal)
    <prelievo del messaggio dal buffer>; -> Usufruisco del nuovo contenuto
    signal(spazio_disponibile); -> Segnalo al produttore che ho consumato il messaggio
    <consumo messaggio>;
} while(!fine);
```

↓ Sistemi multiprocesso

mutex = 1

```
wait (mutex);
{ <sezione critica>
    signal (mutex); }
```

⚠ mutex deve essere in memoria condivisa per poter essere visto da tutti i processi

Le primitive sono però atomiche solo a livello del programma su cui vengono eseguite. NO sincronizzazone tra processori. Se vengono eseguite più volte su processori diversi questa perde la proprietà di atomicità. Le wait fai infatti operazioni di lettura e scrittura che possono essere eseguite in contemporanea sul bus altrui che non si chiude un meccanismo di lock sul bus. Utilizziamo le primitive lock e unlock già viste per rendere atomiche wait e signal in un contesto multiprocesso. Possono tuttavia bussi wait in quanto wait e signal sono sezioni critiche molto brevi

↓ Modifica mutex

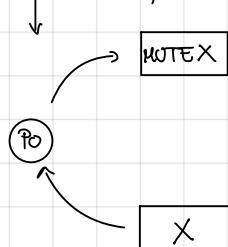
mutex = 1 Memoria condivisa

lock (x); → Atomicità garantita da istruzione TSL

wait (mutex); → se il programma si blocca sul mutex non può essere fatto unlock e quindi il bus rimane bloccato

```
<sezione critica>
lock (x);
signal (mutex);
unlock (x);
```

deadlock: in altro processo che vuole fare la signal non può farla a causa della presenza di lock(x) prima di signal (mutex).



: archi = possono rintracciare

⚠ Attesa ciclica

↓
Possiamo provare a togliere la unlock e la lock nel metodo. In questo caso la coda dei mutex rimane sempre vuota perché la wait può essere fatta solo dopo la lock e per farla c'è bisogno che il processo precedente abbia fatto unlock: blocco sulla lock. → **⚠ Problema dato della lentezza**, hung wait inaccettabile se la sezione critica è lunga.

↓ Risoluzione: cambiamo wait

```
wait (X)
if (S.value == 0) {
    unlock (X);
    <P si risponde>
}
else
    S.value --;
```

→ del signal non deve essere modificata in quanto non è bloccante

$$\text{Val}(S) = S_0 + n_{\text{signal}}(S) - n_{\text{wait}}(S) \geq 0 \implies n_{\text{W}}(S) \leq S_0 + n_{\text{S}}(S)$$

$$N = n_{\text{W}}(S) - n_{\text{signal}} : \text{Processi in sezione critica su una variabile condivisa}$$

$$\text{mentre } N = n_{\text{W}}(S) - n_{\text{S}}(S) \leq n_{\text{S}}(S) + S_0 - n_{\text{S}}(S) = S_0 \rightarrow \text{limite superiore}$$

Problema comunicazione su buffer grande N

spazio disponibile: valore iniziale = N

memaggio: valore iniziale = 0

```
// produttore
wait(spazio disponibile); -> Verifico se lo spazio è disponibile (semaforo inizializzato a n e non 1)
wait(mutex); -> MUTUA ESCLUSIONE
<ins>;                                buffer [Coda] = msg;
coda = (coda+1) % N;
signal(mutex); -> MUTUA ESCLUSIONE
signal(msg disponibile); -> Segnalo al consumatore che è disponibile nuovo contenuto
```



```
// consumatore
wait(msg disponibile); -> Verifico se ci sono cose nuove da leggere
wait(mutex); -> MUTUA ESCLUSIONE
<prel>;                                msg = buffer [testa];
testa = (testa+1) % N;
signal(mutex); -> MUTUA ESCLUSIONE
signal(spazio disponibile); -> Segnalo al produttore che ho consumato il messaggio (semaforo inizializzato a n e non 1)
```

testa e coda non
si sovrappongono grazie
ai invarianti

• ALTRI PROBLEMI di SINCRONIZZAZIONE

banded-buffer problem: problema di produttori e consumatori già visto

reader-writer problem

Dato che la lettura non modifica il documento più lettori possono agire in contemporanea: NO mutua esclusione. Poiché la scrittura modifica, se ne può avere uno per volta e, quando W è in sezione critica il documento non può essere aperto in lettura.

↓ semafori

mutex inizializzato ad 1

wrt (writer) inizializzato ad 1: controlla gli scrittori

Problema di mutua esclusione: non devo stabilire un ordine per quanto riguarda la scrittura →

```
do {
    wait (wrt);
    // writing is performed
    signal (wrt);
} while (TRUE);
```

writer

→ lettori devono ringraziare un wrt per vedere se la scrittura è occupata. Si possono avere più lettori contatti del semaforo readcount inizializzato a 0.

do {

```

    wait (mutex) ;
    readcount ++ ;
    if (readcount == 1)
        wait (wrt) ;
    signal (mutex)
        // reading is performed
    wait (mutex) ;
    readcount -- ;
    if (readcount == 0)
        signal (wrt) ;
    signal (mutex) ;
} while (TRUE);

```

leggere

Prima di prendere scrivere vediamo se c'è scrittura

se non è ultimo lettore li hero la scrittura in modo da permettere la scrittura

Crea sezione critica su readcount in quanto condivide fra tutti i writer.

Si può avere starvation? da starvation si parla quando si gestisce la coda con una politica prioritaria.

Sostiene: il primo viene e quelli dopo vanno in coda → processi reattivati con politica FCFS. Nessuna starvation.

Potrei priorizzare però anche in base al tempo impiegato. In questo caso avrei rischio di starvation a causa dell'arrivo di processi veloci.

Variazioni: 1. scrittori non possono essere accreditati finché wrt = 0
2. No priority a scrittori

dining - philosophers problem



→ Affamato: filosofo ha a disposizione 2 bacchette per prendere il riso ed i filosofi sono 5. Prima deve prendere la destra e poi la sinistra e poi può mangiare. Alla fine le deposita nel controllo. Le 2 bacchette sono però condivise tra tutti in quanto sono 5: mutua esclusione.

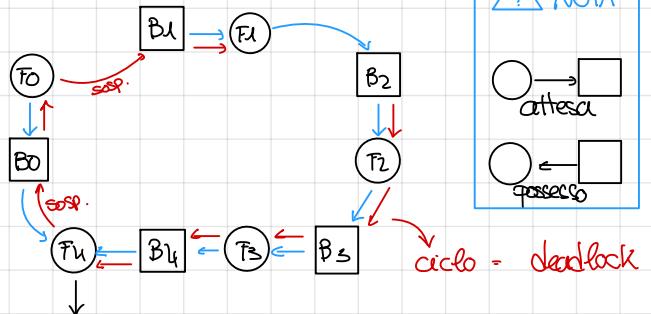
vettore di rendere chopstick [5] iniziali settati ad 1: si fa la mutua esclusione

```

do {
    wait ( chopstick[i] );
    wait ( chopstick[ (i + 1) % 5] );
    // eat
    signal ( chopstick[i] );
    signal ( chopstick[ (i + 1) % 5] );
    // think
} while (TRUE);

```

→ **⚠ deadlock**



Arco vuente = processo bloccato.

2 archi entranti = processo può proseguire.

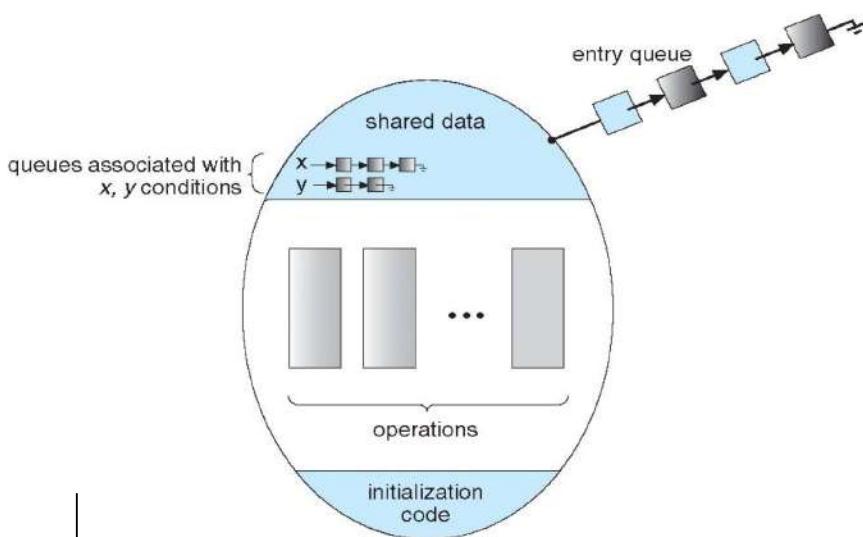
Stato non preoccupante = non c'è ciclo

Signal: si toglie arco e si invierte arco attira

Monitor: Cassetti = struttore dati definiti sul monitor su cui sono definiti metodi → tutte le procedure gestiscono la risorsa in mutua esclusione

→ Non è necessario usare semafori in quanto il codice viene implementato tramite l'uso di semafori in automatico.

Tra le strutture condizionate ci sono le variabili condizione su cui sono definite wait e signal. La wait è sempre bloccante mentre la signal ricattiva un processo o non fa nulla ↳ semafori senza valore



Alla coda esterna si aggiungono tutte le code interne quante sono le variabili condizione.

⚠ Potrebbe nascere deadlock: wait e signal su variabili condizione possono essere eseguite con monitor in mutua esclusione. Se è stata fatta una wait e il processo si è sospeso non può riattivarlo dato che il monitor è in mutua esclusione

Quando P invoca una signal su una VC e Q aveva eseguito una wait prima Q deve avere risvegliato

↓ politiche

Signal and wait: Q viene eseguito e P viene sospeso

Signal and continue: P prosegue e Q viene messo in coda pronta

↳ "princípio di testimone": diritto di uscire

↓ implementazione

Variabili:

Procedere: (automaticamente)

wait(mutex);
...
body of F;

...
if (next_count > 0)
signal(next);
else
signal(mutex);

Bloccante: implementa politica signal and wait

Variabili condizione: semaphore x_sem ; // (initially = 0)

int $x_count = 0$; → Contatore processi sospesi nella VC

$x_count++$; → OK: viene eseguito in mutex

if (next_count > 0) } → controllo processi pronti per procedere ad esecuzione nel monitor

signal(next); } → processi che possono essere eseguiti sono stop e cerca di mutex
else { → processi che possono essere eseguiti sono stop e cerca di mutex
signal(mutex); } → Rilascio il monitor prima di bloccarsi

wait(x_sem); } → finita il deadlock: rilascio il monitor prima di bloccarsi
 $x_count--$; } → finita della sospensione viene fatto dunque una signal

Attenzione: eseguito in mutex

X. wait: if ($x_count > 0$) { → se $x_count = 0$ non fa nulla

next_count++; → processo pronto nel monitor

signal(x_sem); → processi da eseguire

wait(next); } → mi sospendo per next bloccante

next_count--; } → processo eseguito

Risoluzione problema dei 5 filosofi

```
monitor DiningPhilosophers
```

```
{
```

```
    enum { THINKING, HUNGRY, EATING } state [5]; → Al più 2 filosofi eating non  
    condition self [5]; → adiacenti (ore 6, ore 10:30)
```

```
    void pickup (int i) { → Voglio prendere ma ho cuchia  
        state[i] = HUNGRY;  
        test(i);  
        if (state[i] != EATING) self [i].wait; → Mangio se posso farlo, mi prendo  
    } → altrimenti se il test è falso.
```

```
    void putdown (int i) { → Finito di mangiare, ritorno a stato thinking  
        state[i] = THINKING;  
        // test left and right neighbors  
        test((i + 4) % 5); → Controlla se i vicini possono mangiare  
        test((i + 1) % 5); → Si rispetta politica signal-and-wait  
    } → △ viene data priorità a filosofo di destra: starvation
```

```
    void test (int i) { → testa se i vicini non stanno  
        if ( (state[(i+4) % 5] != EATING) &&  
            (state[i] == HUNGRY) &&  
            (state[(i+1) % 5] != EATING) ) {  
                state[i] = EATING;  
                self[i].signal(); → permette di far mangiare qualcuno se  
            } → c'è qualcuno che attende  
        }
```

```
initialization_code() {  
    for (int i = 0; i < 5; i++)  
        state[i] = THINKING;
```

```
}
```

```
↓
```

In alcune implementazioni la VC possono avere
il monitor può avere un solo per bloccare qualcuno:

ma con priorità

```
monitor ResourceAllocator
```

```
{
```

```
    boolean busy;
```

```
    condition x;
```

```
    void acquire(int time) {
```

```
        if (busy)
```

```
            x.wait(time);
```

```
        busy = TRUE;
```

```
}
```

```
    void release() {
```

```
        busy = FALSE;
```

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

```
}
```

```
    initialization_code() {
```

```
        busy = FALSE;
```

```
}
```

```
}
```

• PRIMITIVE SEND e RECEIVE

Modello a memoria locale: utilizzato dai processi (a differenza di thread che hanno memoria condivisa). → Comunicazione: processo genera evento e lo comunica ad un altro. A questo evento può essere associato un contenuto (corpo del messaggio) → Canale di comunicazione sostituisce il buffer: P₁ mittente, P₂ destinatario. Si usano primitive send e receive. In questo modello è sufficiente specificare il messaggio in quanto il canale di comunicazione è dedicato



⚠ Non è detto che esista un canale punto-punto: dest può ricevere messaggi da molteplici mittenti senza doverli specificare → modello client-server. Quando il server è destinatario il canale di comunicazione diventa una coda di messaggi dei client ed il mittente è implicito nel messaggio.

↓ Cambiano canale di comunicazione

Cambia percorso dest/origine ed eventuali sospensioni processi. Nel caso restando, se P₁ fa una send la primitiva può essere bloccante o non bloccante.

Send bloccante (sincrona): Invia se dest può ricevere o bloccata se dest non può ricevere

Send non bloccante (no synchronization): Messaggio viene inviato subito su non si riceve a capire se è arrivato al destinatario.

↓ App. web

Client = browser, protocollo HTTP. Da richiesta HTTP viaggia sul canale di comunicazione TCP: definisce canale virtuale in figura con Stream bidirezionale di lunghezza non definita e se il mittente invia un messaggio può avere sicuro che questo arriverà. HTTP usa send sincrono ed eventuali re-send sono a carico dell'applicazione → timeout: client invia di nuovo messaggio. Send asincrono semplificano send ma richiedono feedback

↓ Vice versa

Receive sincrona: P₂ che la esegue si risponde finché il messaggio non è disponibile utilizzando semififo su coda di msg. in ingresso

receive asincrona: P₂ che la esegue non si sospende: se il msg è presente tutto ok, viceversa il processo prosegue ed a livello di programma va gestito il fallimento. Capiamo che le primitive sincrone facilitano la sincronizzazione.

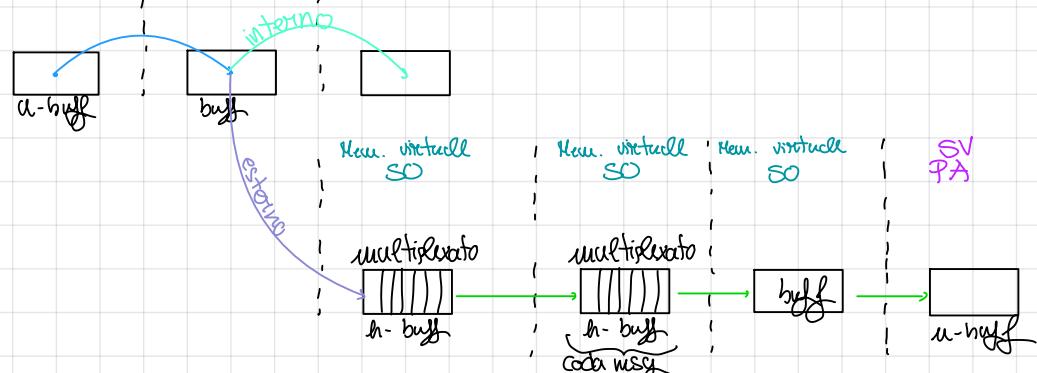
↓ Regola di sinc.

Send bloccante: P₁ si risponde fino ad avendo evento ^{2 possibilità} messaggio partito: prima di send viene istanziato messaggio che viene inviato come parametro della send. Questo è posto in una struttura dati nello spazio virtuale del processo. Il contenuto deve poter essere copiato per poi trasmetterlo. Viene quindi copiato in un buffer residente nello spazio del SO. Il processo rimane bloccato fino a transizione completa. Per riattivarlo non si può more copia da spazio virtuale a spazio SO perché non ci sia sicurezza di invio. Si pensa quindi di more momento di fine invio. (analogia a op I/O). Non lo perci se ne è arrivato e se il processo dest. lo ha cominciato. La maggioranza delle send sincrone attendono quindi il momento della ricezione completa del messaggio. → buffer in coda

I sistema

multi programmato

2 livelli di buffer sistema: buffer dedicato a processi per trasferimenti da mem virtuale processi a sistema. Sotto il buffer si gestisce del canale: buffer unico che gestisce più dest se canale è fermo, buffer non multiplexato se comunicazione tra processi residenti.



⚠ Se la send è sincrona non per forza deve avere anche la receive
formato dei messaggi



Mittente e destinatario: a processi viene assegnato pid univoco. Se però mitt e dest sono su macchine diverse si aggiunge informazione sulla locazione destinataria: IP e porta per sapere su che buffer copiare.
Possono avere anche ACK
lunghezza stringhe

Comunicazione diretta simmetrica

? prende una send c, u e receive specifica quale produttore vuol

```
// Processo produttore P
pid c = ...;
main() {
    msg M;
    do {
        produci (&M);
        ...
        send(c, M);
    } while(!fine);
}
```

```
// Processo consumatore C
pid p = ...;
main() {
    msg M;
    do {
        receive(p, &M);
        ...
        consuma(M);
    } while(!fine);
}
```

Primitive possono avere sincrone o asincrone. Non si fa uso dei semafori in quanto la sincronizzazione avviene tramite send e receive

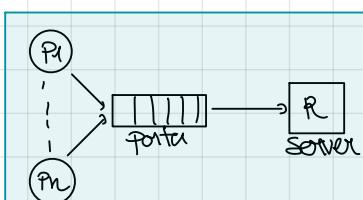
Comunicazione diretta asimmetrica

Potremo sospendere finché non arriverà il messaggio. C non specifica produttore

```
// Processo produttore p
pid c = ...;
main() {
    msg M;
    do {
        produci (&M);
        ...
        send(c, M);
    } while(!fine);
}
```

```
// Processo consumatore c
...
main() {
    msg M; pid id;
    do {
        receive(&id, &M);
        ...
        consuma(M);
    } while(!fine);
}
```

modello client-server

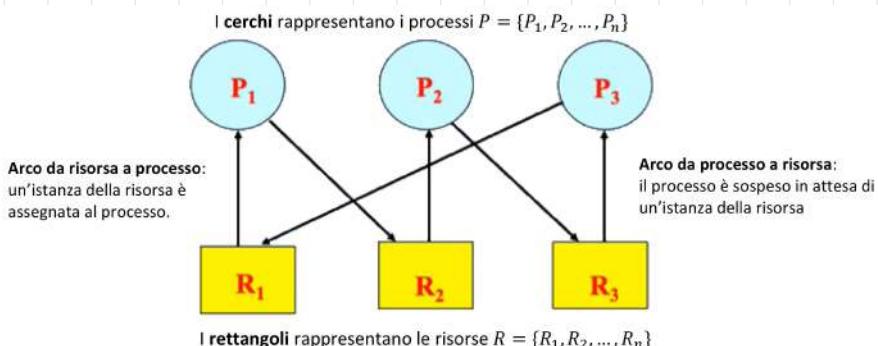


send di C devono essere bloccanti: eseguire procedure call → invio msg (implica esecuzione servizio: procedura remota su computer esterno)

Utilizzo multi-threading → in genere molti da primitive sincrono

blocco critico

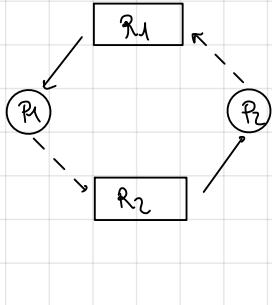
Ricorda risorse disponibili in una singola istanza



- ⚠ Processo bloccato se ha due o più arco uscenti
- ⚠ Processo possiede una risorsa se ha due o più arco entranti

⚠ Blocco critico se attivano percorso circolare: attesa circolare irrinviabile

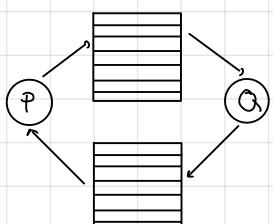
Arco tratteggiato = processo non ha ancora fatto richiesta sulla risorsa sua prima o poi la farà → possono trasformarsi in archi di richiesta bloccanti o invertire il verso (anagnosazione)



Se P_1 possiede R_1 ed P_2 blocca è in grado di terminare e rilasciare la risorsa. Posso dirlo in quanto vengono usate le wait e le signal. Lo stesso non si può però dire se non ci sono archi tratteggiati. Nel caso degli archi tratteggiati infatti le frecce possono invertirsi se vi sono più istanze delle risorse

Algoritmo del banchiere: provare dinamicamente lo stato. Quale perci' comandare gli archi tratteggiati e quindi in un certo senso il futuro.

Dobbiamo distinguere tra risorse commutabili e non commutabili. Si possono avere deadlock anche nel 1° caso



P e Q possono comportarsi sia come produttore che come consumatore. Sui buffer sono definiti renofori. P vuole produrre msg e si suppone che P abbia cercato di produrre msg su buffer pieno: sospendo. Si ha effetto ciclico: blocco critico con risorse commutabili.

condizioni necessarie per lo STATO

1. Le risorse così date devono richiedere accesso in mutua esclusione
2. La riduzione per la mutua esclusione deve prevedere posso e attesa: P ha risorse (archi entranti) ed è bloccato → semafori
 - ↳ protago: wait (mutex)
 - ↳ seg. critica
 - epilogo: signal (mutex)
3. Avanza di diritto di risorsa delle risorse da parte del

SO

4. Esistenza di almeno un'attesa circolare nel graph \rightarrow Diventa sufficiente se tutte le risorse nel graph sono a singola istanza

\downarrow Negoziazione

2. Prevenzione statica (deadlock prevention) \rightarrow Possono essere applicate anche a 3.
4. Prevenzione dinamica (deadlock avoidance) \rightarrow Funzione a graticcio: SO non impone nulla sul codice ma quando blocca una risorsa controlla se il rimanente si incontra su uno stato sicuro: deve poter soddisfare le wait future \rightarrow se i processi possono completare e rilanciare risorse

+

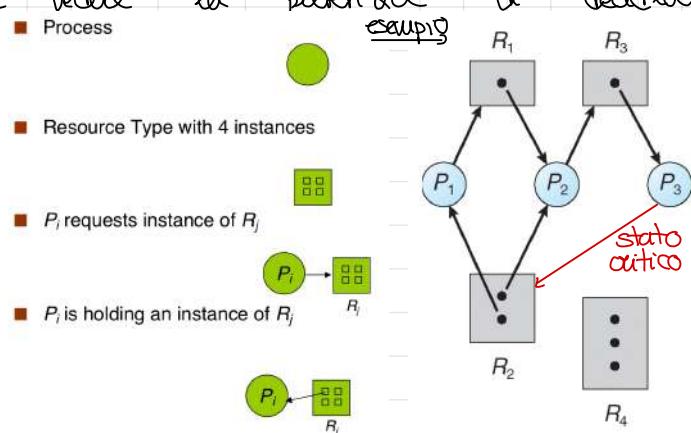
Deadlock detection: preventi probabile deadlock e comportamento di conseguenza

\downarrow

A aiuta a ridurre problemi dei 5 filosofi

\downarrow Modello problema

Processi che hanno risorse implicite (CPU, memoria) ed esplicite (fisiche e logiche). A memoria si indica con w il numero delle sue istanze (=1 per CPU, >1 per memoria, tipicamente 1 per I/O). Quando un oggetto vuole mare una risorsa segue sempre il protocollo richiesta, no, rilascio. Per avere deadlock devono verificarsi tutte e 4 le CN già viste e per vedere la presenza di deadlock si usa il grafico **possesso-attesa**



Gestione del deadlock

3 approcci: Sistema mai in deadlock: deadlock avoidance / prevention

\downarrow Controllo periodicamente deadlock e risolvo: deadlock detection
Ignoro deadlock nel caso in cui siano poco frequenti

deadlock prevention: tecniche statiche (basano su codice o SO) per negare una delle protocolli visto prima

Hold and Wait: P non può richiedere risorse se ne possiede altre (non può mai eseguire un arco entrante se ce n'è almeno uno waiting) \rightarrow Vincoli a programmatore ma controllabili a tempo di compilazione. Inoltre, un vincolo più forte può essere quello di dover dare tutte le risorse al processo prima di poter eseguire. Molto inefficiente \rightarrow Risorse troppo

non utilizzate sempre ed è necessario conoscere prima

le risorse utilizzate

No preemption: SO si rende conto che P non può proseguire perché gli manca una risorsa \rightarrow preemption (solva/carta stop). Questo produce poco overhead nel caso della CPU ma ne produce molto per le altre risorse

Circular wait: Relazione di ordine tra le risorse (in base al loro indice ad esempio). P suo acquirente risorse restante in ordine → uno non efficiente risorse e vincolanti per ordine

deadlock avoidance: Prevenzione dinamica: funziona ce runtime → SO controlla di mantenere in stato sicuro

→ A gestire da questo non si può evitare deadlock

SO rimane in grado di accettare nuove richieste da P. La risorsa viene condata se poi viene restituita: dà potere per sempre terminare i processi fornendogli le risorse necessarie

↓ Condizioni

1. Sistema deve avere qualche concordanza a priori (bandiere: quali e quanti istanze): struttura di tipo conservativo (struttura più risorse di runtime)

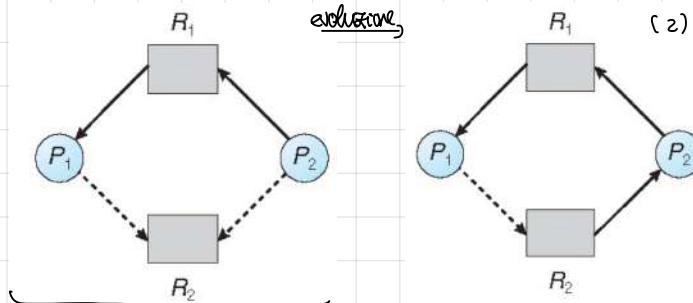
2. Algoritmi lavorano a runtime: intervengono quando P richiede una risorsa

↓ Algoritmo

Un processo che può avere conflitto con le risorse disponibili? Se sì, in quale stato si trova? Il processo può terminare? Se sì, prima o poi rilancerà le risorse → se riesce a trovare una sequenza che include tutti i processi allora lo stato è sicuro. Se le risorse richieste non sono disponibili il processo viene messo in cattiva. Altrimenti, l'algoritmo "fa finta" di concedere la risorsa e vede se lo stato è safe.

↓ 2 tipi di algoritmi

Risorse in singola istanza: Ciclo nel grafo è un deadlock. Usiamo il grafo a cui assegniamo un terzo tipo di arco: claim (richiesta). Questo indica che in futuro il processo potrebbe richiedere la risorsa. Si ha la concordanza a priori. Quando il processo richiede la risorsa l'arco può mantenere il verso o girarsi.



Possiamo vedere se stato destinazione è sicuro. Se P1 richiede R2 e questa è disponibile l'arco si gira e P1 ottiene risorse. Non avendo cicli lo stato è safe. Se P2 richiede R2 prima di P1 → ha lo stato (2): in questo caso però può crearsi un ciclo se poi P1 richiede R2: ciclo = CNS deadlock

↓ Algoritmo permette di riordinare le richieste

Il processo può correggere anche se la risorsa è libera e deve restituirla tutte alla fine

↓ Algoritmo del bandiere

N = numero processi, m = numero risorse

Risorse in istanze multiple:

- Available [m] → Available [j] = K: K istanze disponibili risulta R_j
Max [nxn] → A process, i risulta mi dice di quante istanze ha bisogno el utentino: Max[i,j] = K
Allocation [nxn] → Allocation [i,j] · K: Pi ha assegnato K istanze R_j
Need [nxn] → Need [i,j] · K: Pi ha bisogno di K più istanze di R_j
 ↓ Verifica stato sicuro

1. Let Work and Finish be vectors of length m and n, respectively. Initialize:

Work = Available

Finish [i] = false for i = 0, 1, ..., n-1

dimensione n

2. Find an i such that both:

(a) Finish [i] = false

(b) Need_i ≤ Work → Possò soddisfare necessità

If no such i exists, go to step 4

3. Work = Work + Allocation_i → Restituisce risorse che gli sono state + quelle che aveva prima (ne può terminare)
Finish[i] = true
 go to step 2

4. If Finish [i] == true for all i, then the system is in a safe state

↓ Algoritmo del banchiere

Request = request vector for process P_i . If Request_i [j] = k then process P_i wants k instances of resource type R_j

1. If Request_i ≤ Need_i go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim ↗ processo abortito
2. If Request_i ≤ Available, go to step 3. Otherwise P_i must wait, since resources are not available
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

Available = Available - Request_i;

Allocation_i = Allocation_i + Request_i;

Need_i = Need_i - Request_i;

- If safe ⇒ the resources are allocated to P_i
- If unsafe ⇒ P_i must wait, and the old resource-allocation state is restored

↓ Esempio

$$AV = (10, 5, 7)$$

	Max	All
P_0	7 5 3	0 1 0
P_1	3 2 2	2 0 0
P_2	9 0 2	3 0 2
P_3	2 2 2	2 1 1
P_4	6 3 3	0 0 2

	Need
	7 4 3
	1 2 2
	6 0 0
	0 1 1
	6 3 1

	AV
	3 3 2 → 5 3 2 → 7 4 3 → 7 4 5
	→ 7 5 5

Need₀ ≤ AV? NO: Non può terminare, rimane falso

Need₁ ≤ AV? SÌ → AV = AV + ALL1

Need₃ ≤ AV → Aggirovo AV

Need₄ ≤ AV

Need₀ ≤ AV

Need₂ ≤ AV

seg
gen
za

Portando allo stato iniziale: P_1 : Request₁ = (1, 0, 2) ≤ AV → Nuovo stato ipotetico

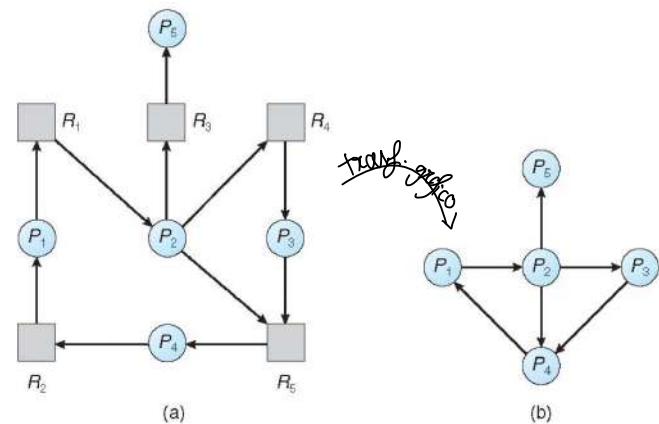
	NEED	AV
0 1 0	7 4 3	2 3 0 → 5 3 2 → 7 4 3
3 0 2	0 2 0	→ 7 5 5

3	0	2		6	0	0
2	1	1		0	1	1
0	0	2		6	3	1

È sicuro? P_1 termina, P_3 termina, P_4 termina, P_0 termina, P_2 termina. Lo stato è sicuro e quindi può avere confidenza.

deadlock detection: Faccio evolvere il sistema e poi eventualmente trovo il deadlock
 ↓ Algoritmo

Risorse in singola istanza: wait-for graph: i nodi sono processi e se si ha $P_i \rightarrow P_j$, P_j deve aspettare che P_i termini \rightarrow Semplificazione resource allocation graph
 ↓
 Ogni tanto si guarda se c'è un ciclo e se c'è si ha deadlock: $O(n^2)$



Se si trova un ciclo, i processi coinvolti nel ciclo sono in deadlock, gli altri non si sa

Risorse in istanze multiple: Abbiamo i vettori available, Allocation e Request \rightarrow risorse richieste dal processo, Δ includendo anche quelle bloccanti. Request è mantenuto a settanta
 ↓ Algoritmo

1. Let Work and Finish be vectors of length m and n, respectively
 initialize:
 - (a) Work = Available
 - (b) For $i = 1, 2, \dots, n$, if Allocation $_i \neq 0$, then
 $\text{Finish}[i] = \text{false}$; otherwise, $\text{Finish}[i] = \text{true}$
2. Find an index i such that both:
 - (a) Finish $[i] == \text{false}$
 - (b) Request $_i \leq \text{Work}$
 If no such i exists, go to step 4
3. Work = Work + Allocation $_i$
 $\text{Finish}[i] = \text{true}$
 go to step 2
4. If Finish $[i] == \text{false}$, for some i , $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if Finish $[i] == \text{false}$, then P_i is deadlocked

Adatto a sistemi general purpose \rightarrow Diverse heuristiche per abortire processi in DL: priorità, tempo esecuzione, rimane wate, rimane per completare, processo interattivo / batch

Scheduling in UNIX

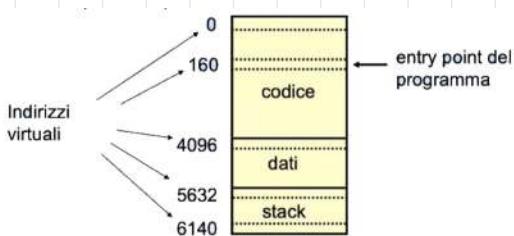
Vengono privilegiati i processi interattivi utilizzando lo scheduling RR con più code ciclamica con il proprio livello di priorità gestite con RR. Ai processi viene assegnato un livello di priorità: priorità > 0: processi esterne ordinari
↓
priorità < 0: processi sistema.
Con il comando nice si può cambiare la priorità a runtime.

Gestione della memoria

Facciamo sempre riferimento al modello di Von Neumann: uno RAM per memorizzare dati e codice programmi in esecuzione → non sufficiente perché molti processi e non vogliono impostare limitazioni su spazio memoria per un processo. Quando si parla di processo si intende un processo per sé.

Analogie tra gestione CPU e memoria: virtuale e reale → un processo non può esistere se non c'è CPU, Mem. virtuale. Sono ancora in dubbio per permettere la virtualizzazione e sono ancora individuare un gestore, ovvero un algoritmo che permette la virtualizzazione della memoria. Vengono poi questi metti implementare lo swapping che permette di trasferire più efficienti. È necessario poi disco alla RAM e viceversa → implementa algoritmi con diritto di revoca. Per quanto riguarda le differenze, la CPU è atomica mentre invece la memoria può avere vista come un array di locazioni vicina con un indirizzo. Queste locazioni possono essere gestite indipendentemente e questo permette di allocare posizioni diverse a processi diversi → **⚠ Le posizioni vanno protette dagli altri processi.** E' però anche possibile condividere protezioni di memoria fra più processi.

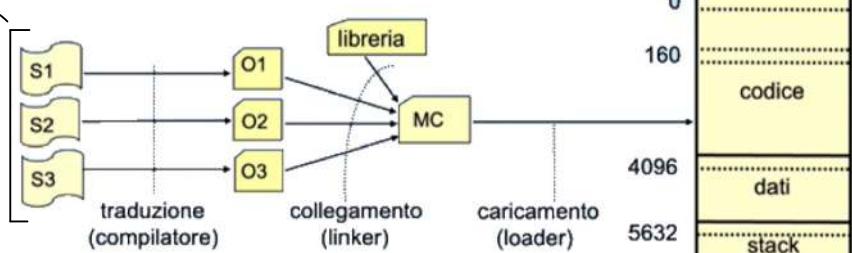
• MEMORIA VIRTUALE



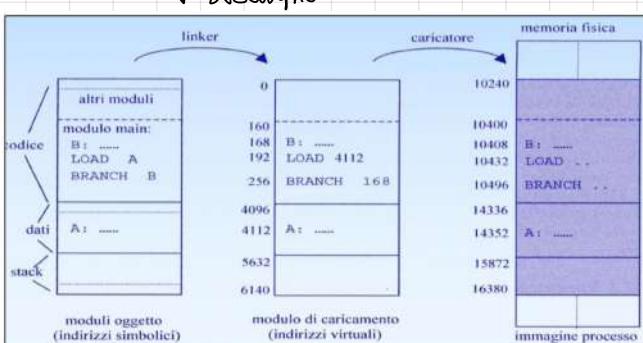
Quando un programma viene compilato vengono gli indirizzi virtuali delle istruzioni hanno indirizzi virtuali nella parte codice e quelle operative fanno riferimento a indirizzi virtuali nella parte dati mentre quelle di sotto si riferiscono alla parte codice.

⚠ Programma sviluppato in moduli distinti

⚠ Posso avere istruzioni che fanno riferimento a indirizzi: se sono nello stesso modulo eseguito non ci saranno problemi. E' però utile permettere il riferimento ad altri moduli: collegamento utilizzando librerie → unico codice e unico spazio di memoria. La sua dimensione è data da dim codice + dim dati + dim stack. Una volta fatto il linking ottieniamo il modulo processo (spazio virtuale) che garantisce il



indirizzamento a cui si aggiunge la pila. Una volta di caricamento che contiene l'immagine del processo nella memoria RAM.



Tutti gli indirizzi simbolici vengono tradotti in indirizzi numerici: **⚠ indirizzi virtuali.**

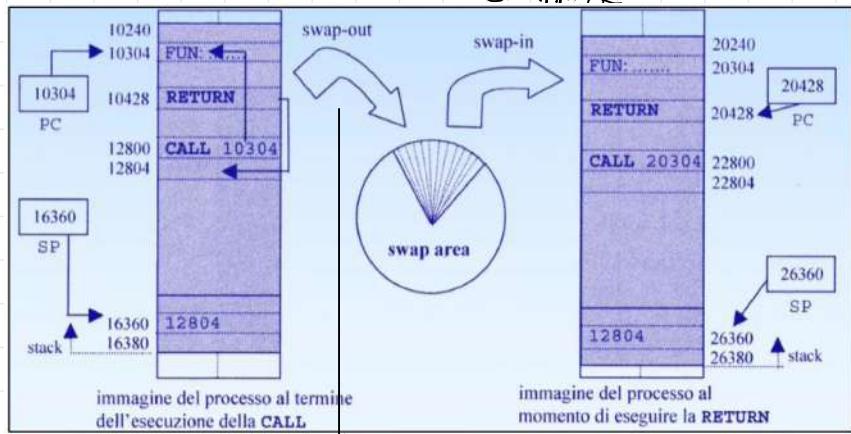
Quando vengono caricati in memoria vengono trasformati in fisici: dato che nel caso precedente ho mantenuto la contiguità è sufficiente una traduzione.

Il processo: spazio virtuale, spazio fisico e rilocazione: SV → SF

Rilocazione statica

Quando un processo viene creato, la sua immagine viene trasferita in memoria virtuale e gli indirizzi virtuali vengono transformati in fisici → fatto durante il caricamento del caricatore rilocante. Alla fine del caricamento in memoria fisica troviamo indirizzi fisici. La CPU genera quindi sempre indirizzi fisici quando viene usata la rilocazione statica.

↓ Rilocazione statica + swapping



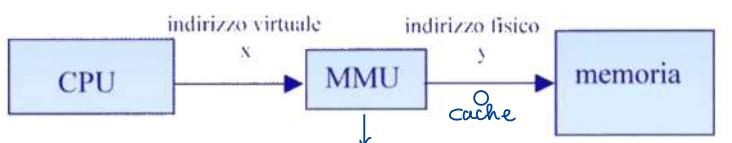
Trasferimento di tutto lo spazio virtuale della RAM alla swap area: richiede tempo ma non è un problema in quanto la CPU rimane libera grazie al WIA e quindi viene riamministrata. Il processo pronto ma rimane invece bloccato. Per esempio se è pronto è necessario che la sua immagine sia in memoria fisica: swap in → Non è detto che dopo lo swap in abbia gli stessi indirizzi di prima dello swap out e quindi gli indirizzi fisici non hanno più validità.

⚠ Problema rilocazione statica: Quando si fa swap in si ricalcolano degli indirizzi virtuali → **⚠ Nello stack rimane vecchio ind. fisico ritorno**. E quindi necessario calcolare gli indirizzi a runtime (non necessario per rilocazione statica ma poco efficiente se usato molto swap).

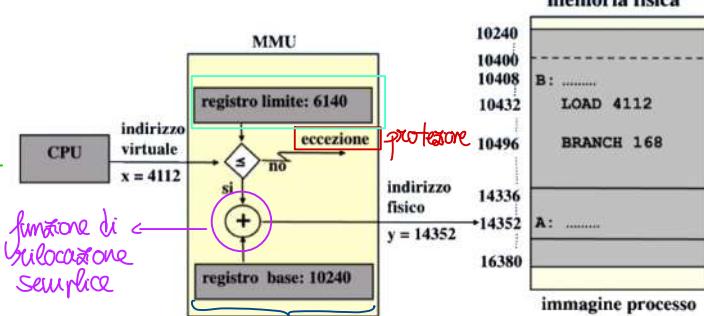
Il singolo spazio virtuale non può essere più grande della memoria fisica (non frammentato). Si implementa una swap area che ci permette di implementare il meccanismo di swapping.

Rilocazione dinamica

Come rilocazione statica una rimangono virtuali → A runtime la CPU genera indirizzi virtuali: ad ogni fine di ciclo della CPU posso trasformi a dover calcolare indirizzi fisici. MMU implementa in low la funzione di rilocazione e quindi l'overhead viene ridotto → Può utilizzare lo swapping liberamente. Ma problema stack e ok swap in a indirizzi diversi.



Dove contenere memoria per implementare le funzioni di rilocazione (costanti, tabella)

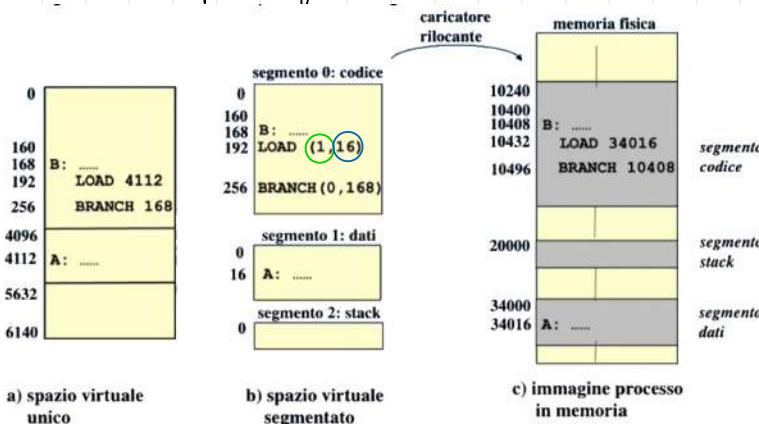


Il contenuto dei registri viene cambiato al cambio di processo: cambio fatto al cambio di contesto e dovrà dunque contenere questi campi.

Inizializzato quando immagine processo è caricata in memoria fisica
→ Indirizzo virtuale deve essere [base, limite] altrimenti eccezione

Organizzazioni della memoria virtuale

Può essere unica o segmentata.
 ↓ esempio segmentazione



→ Segmenti di dimensione diversa /

↓ dimensione diversa

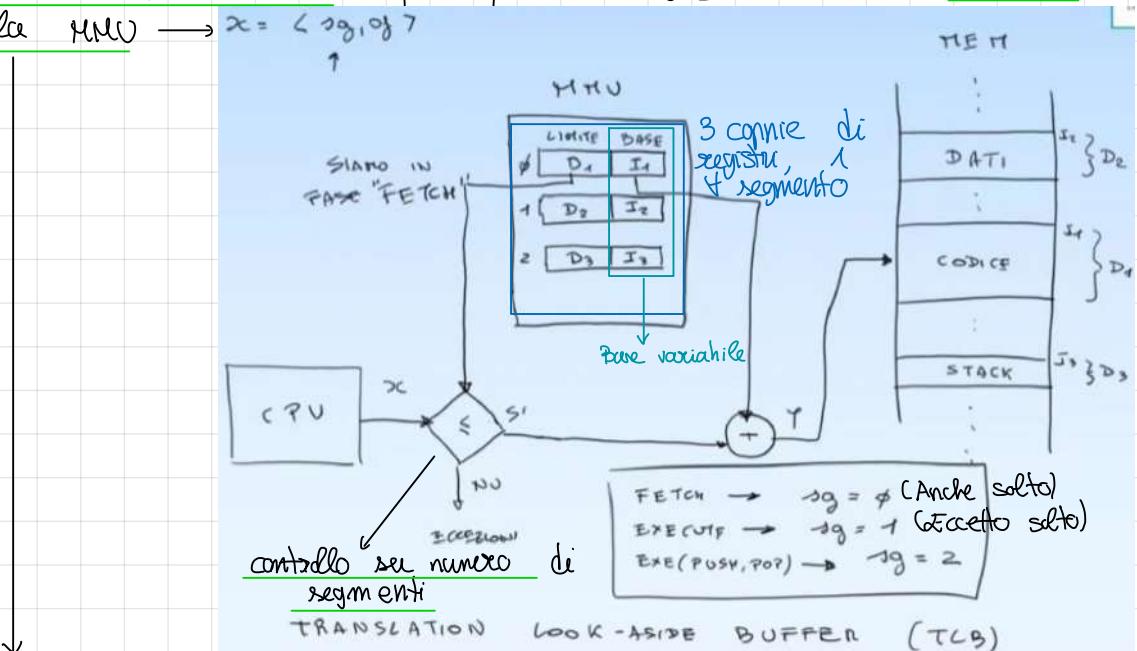
↓ Vantaggi

Semplificare swap-in in quanto sono richieste partition più piccole per ricaricare i segmenti. Gli indirizzi vengono "aggiustati" tramite la rilocazione statica.

↓ Nuovo formato ind. virtuale
Segmento + Displacement

↓

Dove intervenire già al livello di linker: comunque di generare SV ma al livello di linker.
 La rilocazione è facile e può essere fatta quando introduce però i problemi di swap-in già visti e quindi uniamo la rilocazione dinamica con HMU → **⚠️ va complicata**: quella di prima fa riferimento a SV unico.
 Voglio quindi usare HMU con n copie di registri, ove n è il numero di segmenti. Possono uniformare la struttura a 3 segmenti (codice, dati, stack) a tutti i prezzi → segmentazione semplice: linker non deve più avere comunque dei 3 segmenti in quanto si differenziano gli indirizzi in base alle istruzioni (LOAD X, X = dati). Il problema dell'indirizzo può quindi essere risolto a runtime. Dovendo modificare la HMU → $x = \langle sg, off \rangle$



della contiguità nello spazio fisico viene mantenuta? Con le tecniche visto crea frammentazione della memoria. Comunque ho adhesione memoria per tutti gli spazi virtuali ma le singole sezioni sono troppo piccole. Possono aver fatto operazioni di compattamento ma queste operazioni richiedono molto arché e coinvolgono tutta la memoria → possibile solo con rilocazione dinamica

↓ Eliminano contiguità

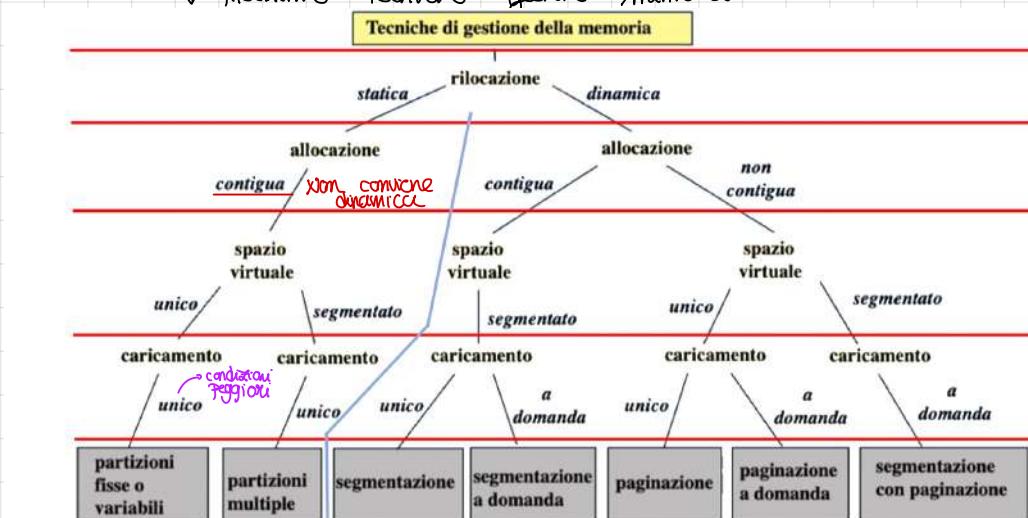
Spazio diviso in pagine della stessa dimensione: **Si** **tranne** **sì**. Questo introduce alcuni problemi in quanto la funzione di rilocazione diventa comulta: tabella. Si parla di pagine logiche quando ci troviamo in memoria virtuale mentre si parla di frame memoria fisica: **⚠️ stessa dimensione**. Queste vengono poste la contiguità ma non ha più frammenta-

- ZONE esterne .

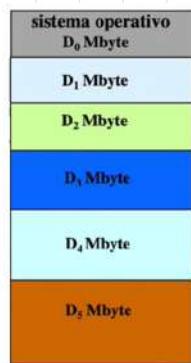
↓
Abbiamo foci delle considerazioni sulla dimensione dello spazio virtuale:
 ≤ fisico: caricamento unico
 > fisico: caricamento a domanda

Per eseguire il processo devono essere presenti in memoria due sue istruzioni: previsto da fare di fetch → operazione di prelevazione?
 ↓ Cosa dicono fare?

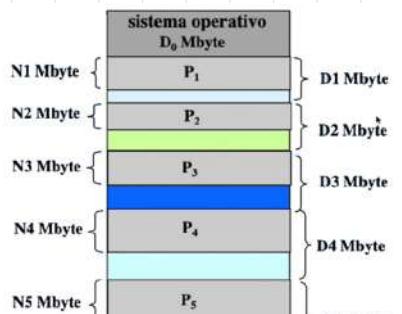
È necessario che sia definito sulla piattaforma la posizione di spazio virtuale che deve essere caricata. Per fare questo bisogna fissare in fase di bootstrapping che lo spazio sia in memoria: descrivere nello spazio virtuale che viene visto quando servono dati e produrre eccezione se non presenti in modo da poterli caricare.
 ↓ Rieccunto tecniche gestire memoria



partizioni fisse: Abbiamo partizioni di dimensioni f. Quando il processo viene creato viene calcolata la dimensione del suo SV e si cerca una posizione libera anch'essa grande per contenervolo. Si sceglie poi la partizione più piccola.

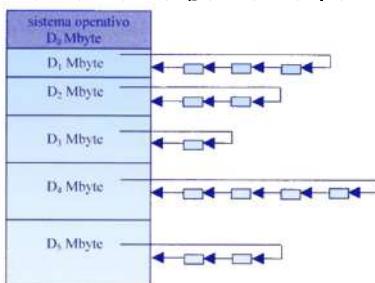


a) partizioni libere



b) frammentazione interna = $(D_1 \cdot N_1) + (D_2 \cdot N_2) + (D_3 \cdot N_3) + (D_4 \cdot N_4) + (D_5 \cdot N_5)$

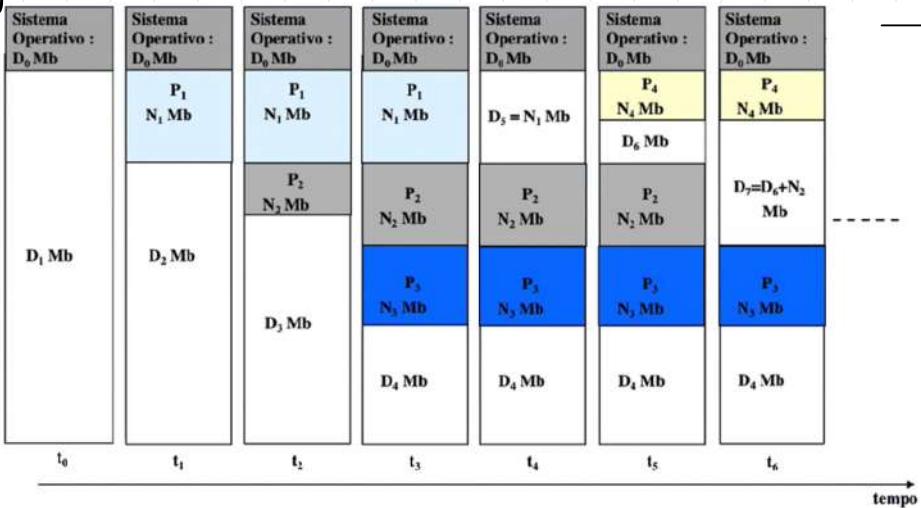
Frammentazione interna. $\frac{1}{(p_{\text{util}})} = \frac{1}{(s_{\text{f}})}$ ha: numero di byte grande rimane inviolato → partizioni **FIXE**. La tecnica è poco flessibile ma permette di fare swap-in e swap-out con rilocazione statica →



Le partizioni possono essere sempre rallocati dopo lo swap out in quanto tutte le partizioni hanno la stessa dimensione. La tecnica richiede inoltre zero overhead.

partizioni variabili

Per ridurre le frammentazioni infine dovrà essere partizioni variabili.



→ La memoria viene organizzata nel tempo in base ai processi creati e terminati. Con la terminazione si creano partizioni sempre più piccole (frammentazione esterna) in modo sempre maggiore. È necessario fare merge di eventuali partizioni libere adiacenti. Se usiamo la tecnica best fit come prima la frammentazione esterna aumenta in quanto le partizioni libere.

sono piccole. Scegliamo però la strategia opposta, il sistema si frammenta più lentamente avendo il costo di allocazione e di rilascio è lo stesso. first fit. In questo caso ordino le partizioni libere. Viene quindi scelta la prima (in termini di indirizzo) partizione libera sufficiente per dimensione crescente. Quando next-fit o worst-fit la fusione è complicata in quanto è necessario scorrere tutta la lista. Quando first-fit considerando le partizioni ordinate per indirizzo non dovrà accadere tutta la lista per fare fusioni.

Segmentazione

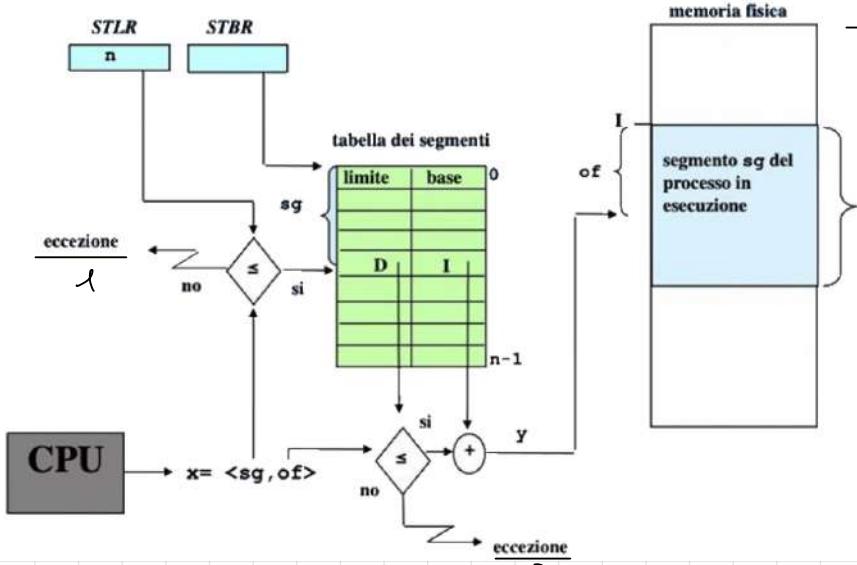
Riallocazione dinamica, allocazione contigua, spazio virtuale segmentato. I sistemi non pongono limiti al numero di segmenti e danno libertà anche per la dimensione. Se non ci sono limiti di numero e dimensione, all'interno di un segmento ci sono informazioni tra loro coerenti (codice, dati, stack,...). Nei programmi più complessi si possono avere più segmenti codice e dati. → Lo è ben organizzato e quindi si ha > efficienza: SO può caricare in RAM solo porzione codice interessante ad esempio. Inoltre, poiché i segmenti sono coerenti con il contenuto, è possibile definire meglio i diritti di accesso ai segmenti. È possibile anche così vedere segmenti utili a processi distinti.

↓ Implementazione

Vinco: Gli indirizzi virtuali devono avere nella forma $x = \langle$ segmento, offset \rangle . È possibile delegare questa operazione al linker solo se i segmenti sono 3
↓

Il processo deve 3 tabella dei segmenti: contiene descrittori di segmento → indirizzo fisico a partire dal quale è allocato (base) e le dimensioni in byte (limite). La grandezza è nell'ordine della decina di segmenti. Si considera sufficiente 2 byte per elemento per contenere base e limite. Le tabelle devono stare in RAM in quanto vengono utilizzate molto frequentemente. Se ho un processo anche il descruttore di processo: diverso 2 campi corrispondenti a 2 registri macchina. STIR (segment table base register) contiene l'indirizzo della tabella dei segmenti del processo. STLR contiene le dimensioni della tabella dei segmenti

f
i
s
e
m
e
n



Protezione: SO deve controllare che segmento appartenga al processo. Se $rg \subseteq n$ è OK, altrimenti eccitare. sg viene usato come indice nella tabella. Questa contiene base e limite. Si deve controllare che l'offset sia dentro il segmento altrimenti eccitare. La traduzione per la memoria fisica è data dalla somma di offset ed I

- ↳ Eccezioni:
 1. Numero segmenti: grave → processo abortito
 2. Sistema non permette di estendere vettore: processo abortito

Sistema permette allocazione dinamica memoria: gestita come richiesta di estensione segmento contenente il dato.

Descrizione di segmento:

↓ Stati processo

base	limite	controllo
Indirizzo della partizione o della swap area	Dimensione della partizione	R W U M P

per adesso NO

Replirazione permessa segmento

→ Nuovi stati: gestire memoria → processo può non avere tutti i segmenti in memoria. In questo caso, se non c'è paginazione o demanda il processo non può eseguire

↓
bloccato swapped: Faccio swap-out processi bloccati

pronto swapped: riattivazione bloccato swapped. Per diventare pronto SO deve fare swap in. Questo stato è raggiungibile anche con lo swap-out di un processo in coda pronti

Scheduling a medio termine:
politica gestione swap in e swap out. Le "vittime" sono tipicamente i processi bloccati in quanto i pronti sono tipicamente scelti da sole delle a breve termine

↓
Politica: apriro processo in bloccato swap-out in riattivo in file swap-in
(possibile politica semplice)

Segmentazione e domanda

Non ha più senso parlare di stato swapped in quanto segmenti vengono caricati quando richiesti dal processo.
↓ Segmento

base	limite	controllo
Indirizzo della partizione o della swap area	Dimensione della partizione	R W U M P

Bit di presenza (in RAM): quando = 1 il campo base contiene l'indirizzo base del segmento. Il campo base contiene questi campi. P invece è definita solo dopo l'inizializzazione in valore 0

All'inizializzazione il segmento è in memoria secondaria: base contiene indirizzo secondaria (file o swap area e viene fatto swap out).
↳ utilizzo → numero in riferimento a tale segmento

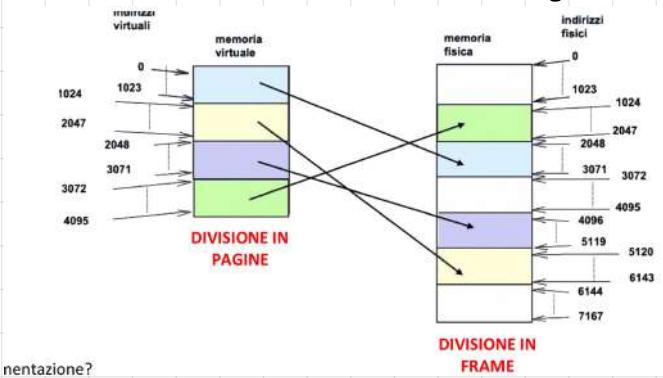
→ M: modificare → segmento è stato modificato (segmenti dati)
 ↓ Utilizzati

Controllo P : P=1 → segmentazione come prima

P=0 → eccedere segment fault: la gestione prevede di trovare una partitione abbastanza grande per contenere il segmento. Per questo si cerca il limite e la lista delle partitioni libere. Considerando quindi il destinatario. Ha anche il sorgente in quanto in base abbiamo un indirizzo di memoria secondaria. In questo momento il processo è bloccato. Una volta conclusa la copia aggiorniamo le strutture dati: P=1, U=0, U=0, have=ind. frame partitione in cui si è copiato. A questo punto il processo viene messo in coda pranti. Quando torna in esecuzione il processo esegue l'istruzione che ha generato segment fault (indirizzo scritto nel Program Counter). Se non c'è spazio abbiamo introducere tecniche di rimpiazzamento
 ↓ Idea

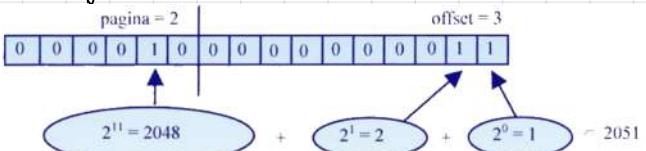
Mantengo la località o permetto interferenza? Generalmente prende la prima quincialazione.

Riallocazione dinamica, allocazione non contigua, spazio virtuale unico. Lo sv è diviso in pagine di dimensione fissa. Lo stesso è fatto per lo sf (stessa dim.). Politica: pug → free. Per allocare uno sv di n pagine devo trovare n frame liberi in sf senza però considerare alcuna relazione di ordine. → Il processo deve avere una tabella delle pagine con deviuttori di pagine (+ grande segmenti perché abbiamo + pagine).

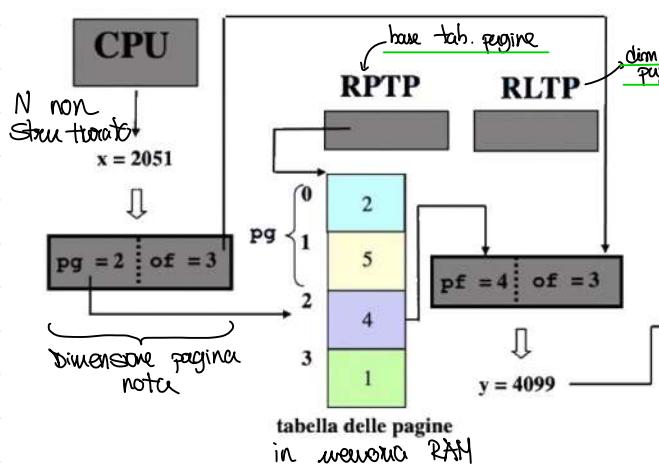


nentazione?

A livello di indirizzo virtuale, posso la dimensione fissa delle pagine, il linker non deve costruire un indirizzo del tipo cpg, offset.
 ↓



Per gestire la divisione numero-offset consideriamo i hit più e meno significativi dell'indirizzo virtuale Δ A runtime



Nello sf l'indirizzo di inizio di ciascun frame ha una esigenza dimensione costante. Avremo quindi che per indirizzare faremo ind. fisico / dim. frame. Questo si chiama indice e la sua dimensione è < indirizzo, il che snellisce il deviuttore.

Paginazione a domanda

Come segmentazione a domanda

descritto

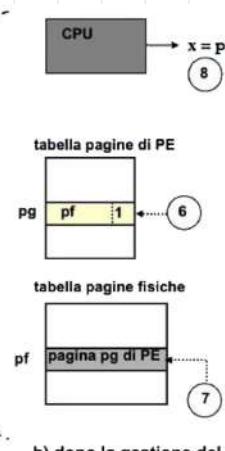
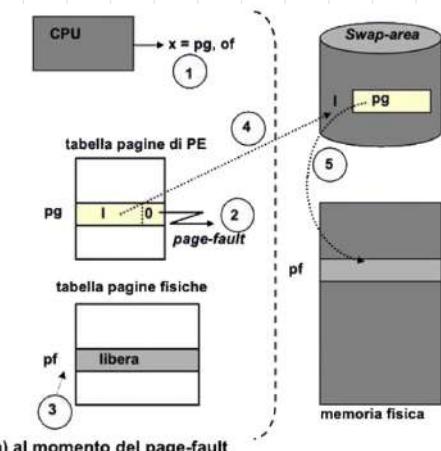
indice frame $i=1$
indirizzo interno $j=0$

R W V M P

Page fault se $P=0$

Sono più efficaci se viene mossa anche la segmentazione

Gestione PF

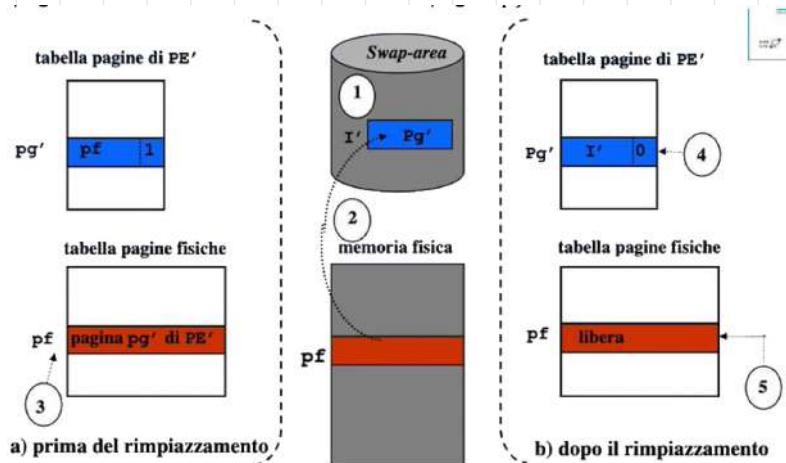


Come prima, utilizzati per rimpiazzamento

1. CPU genera indirizzi
2. Quando schema di traduzione già visto si vede $P=0$ quindi FF e I=ind. cerca Swap → handler
3. SO cerca una pagina libera: tabella pagine fisiche
4. Se I, porta dati da I (cerca swap) a frame libero quando il DMA. PE viene scritto con istruzione che ha generato FF in PC
5. trasferimento concluso: dati in pf

6. Aggiornamento descriptor pagina: $P=1$, $O=0$, $M=0$, pf in indirizzo
 7. Aggiornamento tab. frame liberi: pagina di PE in reg. pf → tabella con dim. > 1 hit
 8. PE torna in coda pronti e quando viene eseguito esegue istruzione che ha causato PF (se tutto va bene)
- ↓ 3. possono non avere frame liberi

Tecniche di rimpiazzamento: ipotizziamo di aver trovato la vittima con qualche funzione. Si fa un trasferimento da pf a ind. swap area che chiamiamo perché ci servirà. Si aggiornano poi le strutture dati: parto da tab. pagina vittima mettendo $P=0$ e ind. swap area. Dichiaro poi pf libero. Per aggiornare la tabella di PE mi serve l'indice della pagina che trovo nella tabella di frame della vittima.



ALGORITMI DI RIMPIAZZAMENTO

ci sono più logico trovare la vittima nel processo che genera PF per non creare intolleranze. Si nota che il working-set dei processi si espanderà fino a sovraccaricarlo. Questo ci permette di lavorare sul working set. L'algoritmo ottimo (teorico) è quello che prende di rimpiazzare pagine che non vedranno più riferimenti più tardi nel tempo: in contrapposizione al principio di località con cui viene costruito il working set.

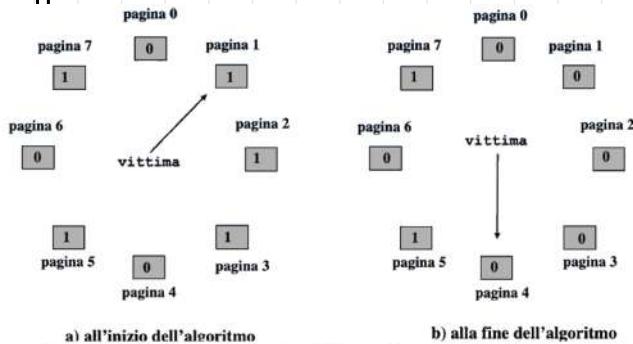
Algoritmo FIFO

Prendo tabella pagine fisiche e rimpiazzo la più vecchia: ne è di lì da tanto tempo ipoteticamente e poco utile → frame gestiti con coda circolare. Produce poco overhead ma può eliminare pagine con alta probabilità di riferimento. È molto probabile che faccia rimpiazzamenti di pagine uscite a vuoto.

Algoritmo LRU (Least Recently Used)

Vittima scelta tra pagine utilizzate meno recentemente. Le pagine devono avere timestamp al posto di U: overhead di swap. Questo deve inoltre avere assegnato ad ogni utile della pagina

Algoritmo second-chance (clock)



Algoritmo FIFO: inizialmente si punta a pag. più vecchia. Si mette in evidenza bit di uso. Quando viene chiamato, si ha come vittima quella puntata dal puntatore. Se U=1, si mette a 0 e si va avanti neutral se U=0 si fa swap-out. → migliore dei 2 casi precedenti. Quando tutti U=1 si comporta come TTO: garantisce di assenza di starvation.

Segmentazione paginata

"propria traduzione" ind. virtuale espressi come c seg. off. Si muo per questo la tabella dei segmenti. I segmenti possono non essere presenti in memoria a causa della pagina erronea → tabella delle pagine. Nei segmenti l'indirizzo base viene sostituito dall'indirizzo della tabella delle pagine di quel segmento.

L'indice della tabella delle pagine viene costituito a partire dallo scostamento dividendo in pagina e offset.

La tecnica è adottata da tutti i processori moderni. La MMU viene affiancata dalla cache TLB (come visto a calcolatori).

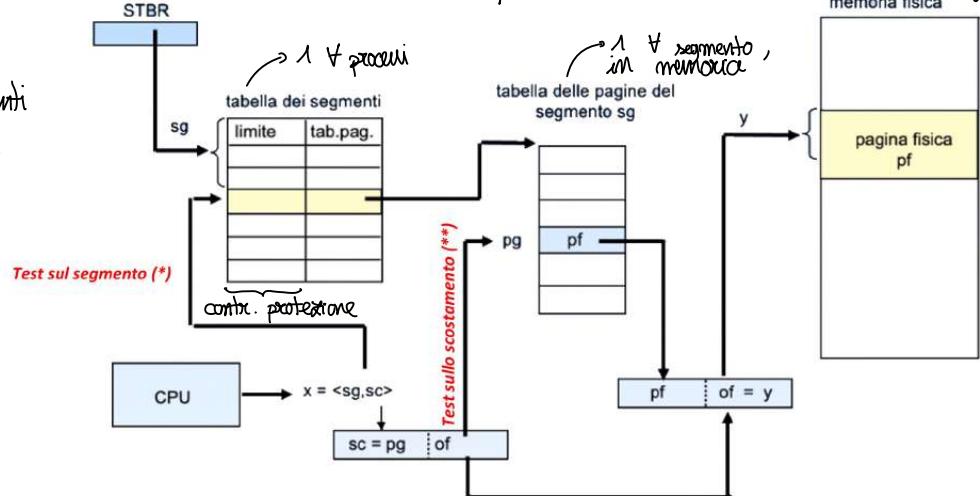
Esercizio

segue reference string: traccia accesi in memoria del processo. Posso può avere al massimo 5 frame. Il processo genera 12 indirizzi virtuali che fanno riferimento a 5 pagine diverse. Da P-r-s è quindi lunga 12 ed al suo interno troviamo gli indirizzi di pagine f. Indicare il numero massimo minimo di page faults avendo che inizialmente non vi sono pagine caricate.

↓

P-r-s: 1, 0, 2, 2, 4, 3, 0, 2, 1, 0, 2, 3
 ↑↑↑↑↑↑↑↑↑↑↑↑

per ottenere ind. fisico. Gli ind. virtuali sono infatti



da MMU viene affiancata dalla cache TLB (come visto a calcolatori).

segue reference string: traccia accesi in memoria del processo

Posso può avere al massimo 5 frame. Il processo genera 12 indirizzi virtuali che fanno riferimento a 5 pagine diverse. Da P-r-s è quindi lunga 12 ed al suo interno troviamo gli indirizzi di pagine f. Indicare il numero massimo minimo di page faults avendo che inizialmente non vi sono pagine caricate.

(↑: page fault)

Numero massimo: numero di valori distinti nella P-r-s

Numero minimo: equivalente al numero massimo

Se il working set fosse < dei valori distinti? Facciamo un esempio con WS=4.

P-r-s: 1, 0, 2, 2, 4, 3, 0, 2, 1, 0, 2, 3
 ↑↑↑↑↑↑↑↑↑↑↑↑

gf con riempimento (supponiamo LRU) → il numero di gf dipende quindi dall'algoritmo scelto
 ↳ introduce overhead

Esercizio 2

P-r-s: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

FIFO, No pre-filling

caso 1: |WS|=1 → 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

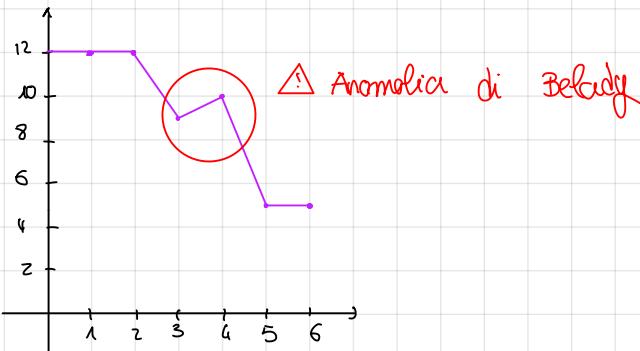
↑↑↑↑↑↑↑↑↑↑↑↑ : 12 ff, 11 cm riempimento

caso 2: $|WS| = 2 \rightarrow 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5$
 $\uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow$: 12 PF, 10 con rimpiazzamento

caso 3: $|WS| = 3 \rightarrow 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5$
 $\uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow$: 9 PF, 6 con rimpiazzamento

caso 4: $|WS| = 4 \rightarrow 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5$
 $\uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow$: 10 PF, 6 con rimpiazzamento

caso 5: $|WS| = 5 \rightarrow 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5$
 $\uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow$: 5 PF, 0 con rimpiazzamento



Algoritmo LRU

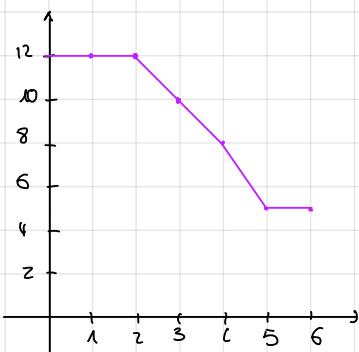
$|WS| = 1 \rightarrow 12 \text{ PF} \text{ (come prima)}$

$|WS| = 2 \rightarrow 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5$
 $\uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow$ → 12 PF, 10 rimpiazzamenti

$|WS| = 3 \rightarrow 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5$
 $\uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow$ → 10 PF, 7 rimpiazzamenti

$|WS| = 4 \rightarrow 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5$
 $\uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow$ → 8 PF, 4 rimpiazzamenti

$|WS| = 5 \rightarrow 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5$
 $\uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow \uparrow\uparrow$ → 5 PF, 0 rimpiazzamenti

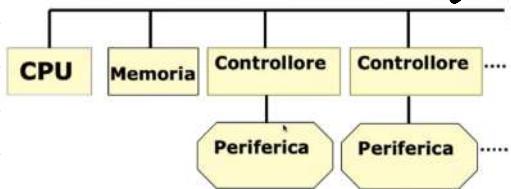


Gestione in VMS

Dal BSD 3 si inverte la segmentazione paginata ed in più viene mossa la paginazione a richiesta. Per quanto riguarda l'eccezione page fault gestita come richiamo già visto. La sostituzione delle pagine viene fatta dal processo centrale del page fault in quanto nel tempo di sostituzione non si considera → core map dovrebbe stato di elaborare dei frame. È sufficiente mantenere sempre qualche frame libero per non fare verificare sostituzione: processo di sistema gestire frame liberi (page daemon) facendo swap-out addendo algoritmo second chance. Vi sono 3 regole: latsfree = nr minimo frame liberi per evitare

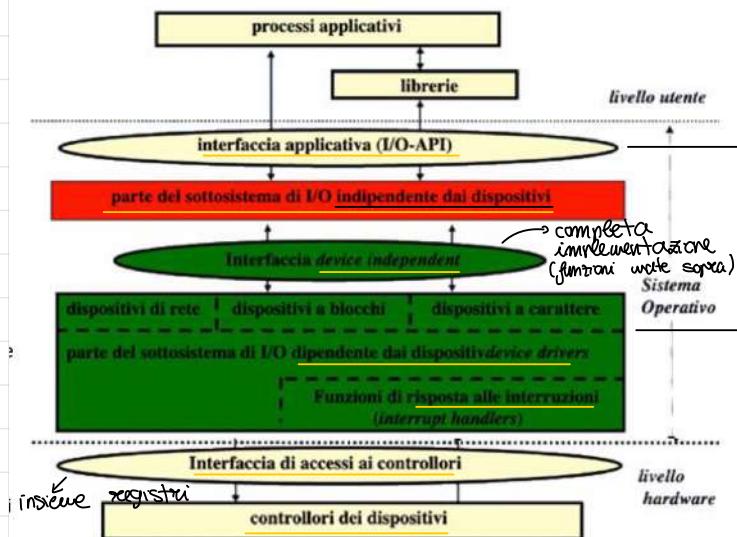
Sostituzione, minfree = nr minimo frame liberi per evitare swapping, desfree = numero minimo frame desiderabili ($lotfree \rightarrow desfree \rightarrow minfree$). Processo swapper fa swap att di tutte le pagine di un processo quando invocato se c'è un processo che richiede molte pagine. Questo ha senso in quanto tipicamente questi processi richiedono velocemente le stesse richieste. Swapper viene eseguito quando nliberi < minfree AND nmodelli < desfree per un certo tempo.

gestione delle periferiche



e' la standardizzazione: nascono il nottosistema di I/O definisce molti funzionamenti mantenendoli all'interno tra le attività di un dispositivo. ↓ Orazioni gestione modulare

Numerose varieggie e molte tipologie e versioni delle periferiche (eterogeneità). Ogni periferica si interfaccia con il bus tramite il controllore. Questo è una CPU dedicata a ciascuna periferica e quindi non servono algoritmi di scheduling. L'obiettivo del nottosistema di I/O sono tutti i dettagli hardware dei controlleri. In particolare uno schema di naming per i dispositivi, gestire i sincronizzazioni tra le attività del processo che l'ha attivato.



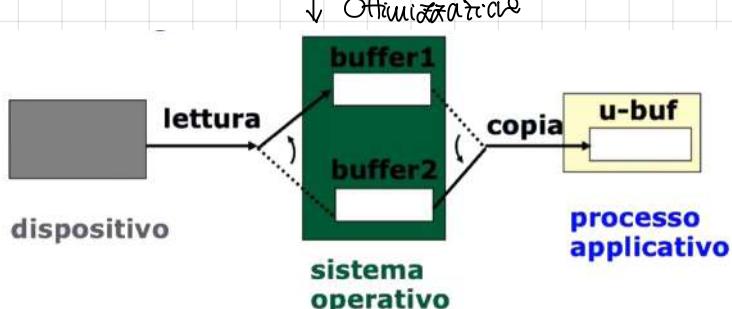
Utilizzata per scrivere programmi che utilizzano I/O. Le funzioni sono implementate dal SO.

3 classi di dispositivi: file, a blocchi, a controllore. Vi è una parte del nottosistema I/O device dependent: collezione di device drivers. Si ha infine l'implementazione degli interrupt handler per gestire la sincronizzazione.



indipendente da device

→ buffer SO device dependent mentre u-buf è indipendente dal dispositivo. → Segue la logica della primitiva scrittura a dispositivo dell'intervento di I/O.



→ Mi permette di riempire un buffer copiando contemporaneamente in u-buf il contenuto dell'utero grazie al DMA

livello indipendente dei dispositivi

Funzioni: naming, gestione malfunzionamenti, allocazione dispositivi a processi applicativi:

- **Naming:** Nome univoco assegnato a dispositivo per riferirsi al gestore del dispositivo. → In Unix gestiti come file allocati in cartella dev.
- **Gestore malfunzionamenti:** Evidenzia alcuni malfunzionamenti non rinolvibili riferendo il comando
- **Allocazione ad ayy:** scheduling e gestire accessi concorrenti.

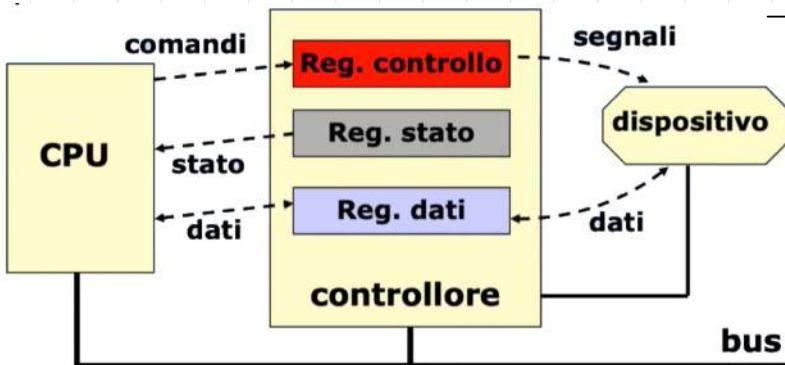
livello dipendente dei dispositivi

Funzioni di accesso ai dispositivi →

N = record (disp, buffer, nbytes)
↓
non disp. Buffer sistema
byte effettivamente transferiti

Numerose liste da trasferire

GESTIONE DISPOSITIVI



→ 3 registri per dispositivo. Quando un programma deve interagire con un dispositivo è sufficiente leggere e scrivere dai registri che hanno indirizzo nello spazio di I/O (device independent). Per far ciò si usano le istruzioni IN e OUT.



bit Start: quando passa da 0 ad 1 il dispositivo si attiva

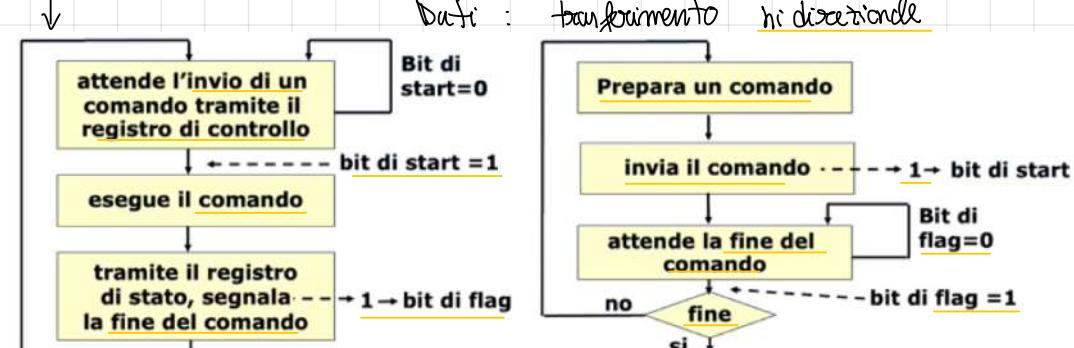
bit abilitazione interruzioni: 1 = sync con INTR, 0 = sync al controllo di programma

bit di lettura: usato dal controllore per informare programma

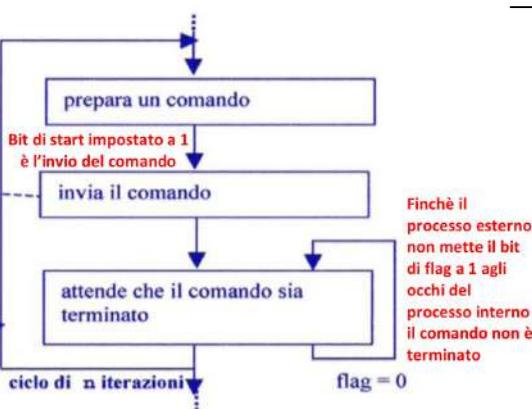
bit e: verifica presenza di errori

bit flag: come finiscono comandi

+
Dati: transitorio



Il processo esterno è libero finché il processo interno non mette a 1 il bit di start

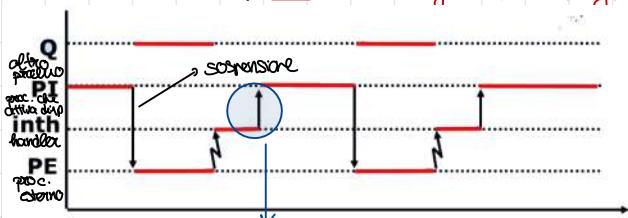


→ Comunicano tramite memoria condivisa (registro nello spazio di I/O)

processo esterno processo interno

↓ △ Non vogliamo bus wait quindi usiamo la INTR.

Quando il dispositivo scrive nel registro di stato manda anche un'INTR. Questa viene poi gestita tramite handler (intr) eseguito dalla CPU. Il codice di intr viene eseguito dal processo in esecuzione (lo vedremo)

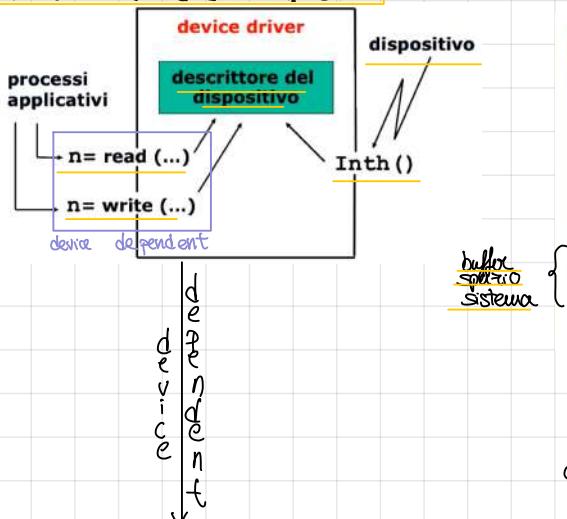


intr dove riattivare Pi: risponde CPU il registro di stato, gestisce gli errori

quando scheduler gialla consegna alegge inoltre i dati se non ci sono erari

L'evoluzione deve ripetersi tante volte quanti sono i hyk che devono avere targetisti. Nel modello precedente si ha un caosico di contexto per ogni hyk: trovino overhead → si eviterà di guadagnare incaricando enti di gestire trasformanti di più hyk.

Astrazione del dominio



- ↳ Informazione dependente dal dispositivo
- risolve problemi di sincronizzazione
- ↳ (inizializzato a 0)
- ↳ Informazioni da fornire a int.
 - il valore di contatore è scritto nelle primitività come nbytes.

↓ Con DNA
contatore e quantità sono direttamente in DNA. Il trasferimento
avviene per via dello stesso modo.

dere (utilizzato da primitiva): int _read (*drex, buffer, int cont)

→ costituto da: primitiva - read + init + descrittore

↓ definizione della primitiva

int -read (int dup, char * phuf, int 'cont') {

dove [dirig]. cont = cont; 7 Iniziali fiorano paraventosi direttore

dove [dup]. punt - phuf; → Sicuro punti per attivare il dispositivo

2 attivazione del dispositivo ! R processo P va sospeso: utilizziamo semaphore

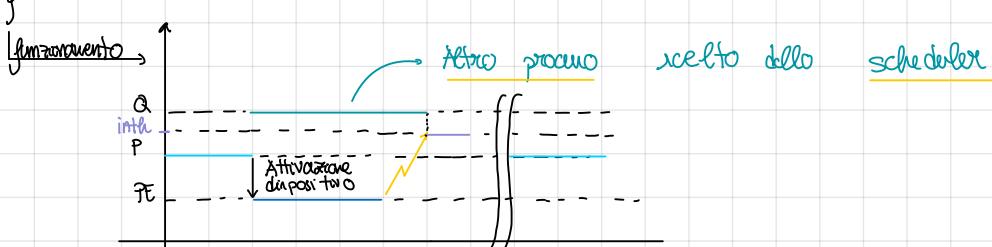
→ Scriviamo nel registro di controllo : istruzione OUT .

denoe [dɛnɔ̃]. dato - denoniale . wait () ;

if (Codice Esito . esito = = < codice di errore >)

return (-1);

return (cont - dece [dup].cont); —> Restituisco hypothesis trasferiti effettivamente



così dove farce inter? Transferire tutti i dati e poi riutilizzare il pacchetto una volta transferiti tutti

void int h () { → Nombrar parámetro, especificar del dispositivo

char b; // variabile di connoggio

<lettura registro di stato del dispositivo>

if (hit_error == 0) { // Non ci sono stati errori che trasferiti correttamente da periferica
// a buffer }

b - / effusa del registro dati >

* deroc [dɪərɒk]. went = b.

derex [dɛrp]. punt ++;

derex [dɛrɛks]. cont --;

if (decor [dup].cont != 0) { // Non abbiamo ancora finito tutto

< riattivazione del dispositivo >

else {

dove [dip]. esito = <terminazione corretta>;

derc [disp]. auto_ devoni hole . signal C); // Riattivazione del processo

```

else { // si è verificato un errore : continuiamo di maneggiarla
    c routine gestire errore>
    if (<errore non recuperabile>) devor [dup]. esito = <codice errore>;
    devor [dup]. defo. dependent. signal();
}
return;

```

↓ Funzionamento

Interfaccia applicativa

Processo P_i

...
...
...

Nel momento in cui il processo P_i vuole accedere allo spazio di I/O provvederà a istanziare un buffer (u-buf) nello spazio virtuale del processo applicativo.

```
int n;
int ubufsize;
char ubuf[ubufsize];
```

A un certo punto abbiamo l'invocazione della primitiva, che appartiene all'interfaccia di I/O (API) messa a disposizione dal sistema operativo al "sistema di programmazione".

Sia IN il nome simbolico del dispositivo:

```
n = read(IN, ubuf, ubufsize);
```

A seguito della chiamata si ha la system call, con cui avviene cambio di privilegio da utente a kernel e disattivazione delle interruzioni (salvataggio in pila e aggiornamento opportuno del registro dei flag).

UNIX. In UNIX il soggetto è una terna dei valori *pid-uid-gid*. Quello che facciamo è cambiare il soggetto, alterando solo uid-gid.

...
...
...

System call

Sistema operativo *device-independent*

Siamo passati da lato applicativo a lato sistema operativo *device-independent*, attraversando l'interfaccia applicativa. Eseguiamo la primitiva read invocata dal processo: al suo interno definiamo il buffer di sistema, controlliamo i permessi, facciamo naming e invochiamo il device driver (quindi della read che abbiamo definito poco indietro)

```
int read(device dp, char* punt, int cont) {
    int n, D;
    // N dimensione ideale per ottimizzare il trasferimento con la periferica (dimensione del blocco logico periferica)
    char buffer[N];
    < controllo accessi >
    < naming per la traduzione dell'identificatore simbolico nell'identificatore univoco dp <-> D >;
    n = _read(D, buffer, cont);
    <trasferimento dati da buffer a ubuf>;
    return <codice di errore, in generale numero di byte letti>;
}
```

Interfaccia *device-independent*

Sistema operativo *device-dependent*

Abbiamo attraversato l'interfaccia device-independent (interna, non visibile al programmatore ma necessaria per strutturare il codice del gestore I/O all'interno del sistema operativo) ponendoci nella parte device-dependent:

```
int _read(int disp, char* pbuf, int cont) {
    <inizializza il descrittore del dispositivo>;
    <attiva il dispositivo>;
    <sospende il processo>;
    return byte_letti;
}
```

L'attivazione del dispositivo avviene lanciando istruzioni dell'interfaccia hardware. L'attivazione del dispositivo da inizio all'attività del processore esterno (controllore della periferica). Il device driver, dopo aver attivato il dispositivo, sospende il processo (e viene posto in coda, nel semaforo dedicato alla periferica).

A un certo punto il dispositivo lancia un'interruzione esterna, che gestiamo attraverso l'interrupt handler *inth*

```
void inth() {
    <trasferimento dati in buffer>;
    <riattivazione del processo>;
}
```

Interruzione esterna

Il lancio dell'interruzione esterna con esecuzione di *inth* può ripetersi diverse volte, al più *cont* volte.

Interfaccia Hardware

HARDWARE

Esempio: device driver di un timer
 Orologio che oscilla secondo una certa frequenza, reprogrammabile per lanciare esecuzioni esterne

↓
 Primitiva delay: processo può sospendersi per un certo intervallo di tempo (esempio in ticks) specificato

IND. REG. CONTROLLO
IND. REG. STATO
IND. REG. CONTATORE
ARRAY SEMAFORE:
fine attesa [N]
ARRAY di INTERI:
ritardo [N]

} permette di gestire delay da più processi

void delay (int n){

int proc;

proc = < indica proc in esecuzione (chiamante) >;

descx. ritardo [proc] = n;

descx. fine - attesa [proc]. wait();

} ↓ Gestore interruzioni

void intth () {

for (int i=0; i < N; i++) {

if (descx. ritardo [i] != 0) {

descx. ritardo [i]--;

if (descx. ritardo [i] == 0)

descx. fine - attesa [i]. signal();

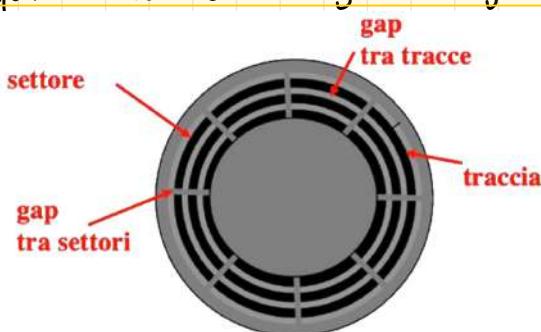
}

}

↓

⚠ Se N è grande il ritardo dell'esecuzione del codice potrebbe non essere troncabile

Gestione del disco rigido (magnetico)



entrambe le facce. I dischi possono in questo caso riportare il concetto di raggio. Il riconoscimento di tracce e settori è gerarchico (inter-settore). I settori hanno lunghezza e distanza dal centro stessa numero di hit. Quelli più centrali sono più densi e quindi è più probabile che si verifichino errori.

↓ Tempi

Seek: Tempo impiegato per testina mobile per spostarsi

- minimo: da cilindro ad adiacente
- medio

Rotazione: Tempo costante (ω è costante) → Tempo impiegato da testina per "vedere" traccia intera

Trasferimento settore

↓

I settori vengono organizzati come un vettore. Il disco virtuale è un vettore indicizzato da una tripla: < faccia, indice traccia in cilindro e indice

settere in traccia).

Scheduling

Nel caso di testina fissa perde di significato il tempo di seek mentre rimane significativo il tempo di rotazione. Il tempo di accesa è dato dal trot/sector. Nel caso di testina mobile il seek è invece significativo.

Si avrà $TF = \underline{TT} + \underline{TA} + \underline{ST}$ tempo trasferimento "effettivo"

$$TA = \underline{\text{tempo medio accesa}} = ST + RL$$

Seek true rotation latency

Il tempo dipende anche da come il file è organizzato sul disco: file = array di byte. Si prende la dura in byte del file e del settore in modo da sapere quanti settori ci sono. Dobbiamo adesso decidere come posizionare i settori. Possiamo scegliere di allocarli in modo contiguo e quanto ha senso in quanto si minimizza il seek time. Potrei invece scegliere di allocare il file in modo non contiguo per non avere fragmentazione esterna.

↳ Differenze tra allocazione contigua e non contigua (in termini di tempo)

$$TF = ST + RL + TT$$

$$TF_{\text{file}} = \underbrace{5.2}_{\text{medio}} + \underbrace{3}_{\text{medio}} + \underbrace{0.019}_{\text{trasferimento}}$$

Trasferiamo un file da 320 KB su 60 settori

sett. contigui

$$TF_{\text{contigua}} = 5.2 + 3 + 0.019 + \frac{320}{60} \cdot 0.019$$

↓ 1° settore altri settori

$$TF_{\text{contigua}} = 5.2 + 3 + 6 + 0.6 = 23.8 \text{ ms}$$

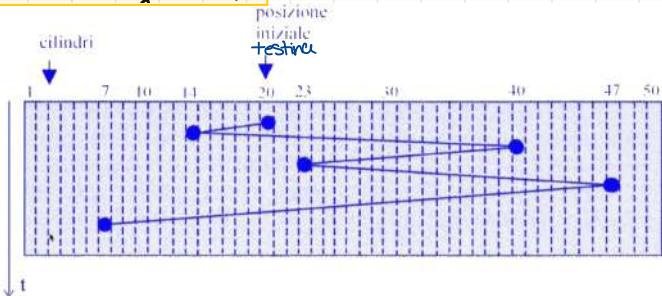
lego tutta traccia tempo minimo (traccia adiacente)

$$TF_{\text{casuale}} = 60 \cdot T_{\text{tracce}} = 5260 \cdot 16 \text{ ms} (\gg TF_{\text{contigua}})$$

Non sappiamo dove sono

Vengono formate code di accesa al disco per trasferire i file. Si utilizzera un buffer di sistema di dimensione pari alla dimensione del settore. Avendo che ST è più grande conviene organizzare uno schema che lo minimizza.

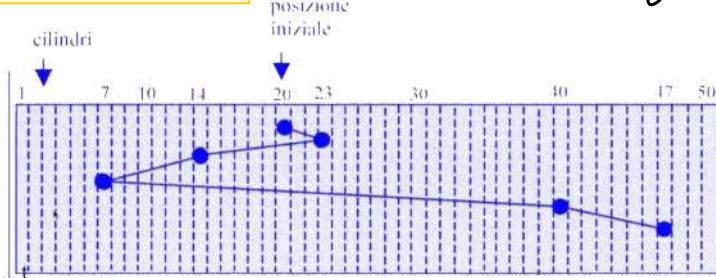
Scheduling FCFS



Rappresentiamo le richieste in coda indicando -ne il cilindro in questo caso abbiamo 14, 20, 23, 47, 7

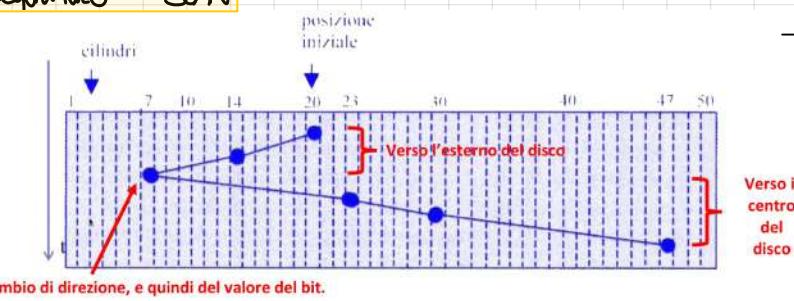
L'algoritmo FCFS prevede di servire le richieste in ordine, quando però numerosi spostamenti sul disco → tempo attesa grande: 113 cilindri attraversati · 0.6

Scheduling SSFS (shortest seek time first)



Algoritmo con priorità: sceglie richiesta che richiede lo spostamento più piccolo sul disco. → Attraversiamo 52 cilindri anziché 113 ma abbiamo più overhea d per la solitazione e le probabilità formazione di starvation

Algoritmo SCAN



poi inverti il verso (versione più semplice dell'algoritmo) — riconosci derivano da richieste fatte nel momento scagliato, che è comunque \leq al giro completo.

→ Inseriamo nuovo hit: direzione di spostamento attuale delle testine. Vengono prima quindi gestite le richieste "fuori dal'attuale" verso di spostamento della testina. D'algoritmo inverte quando non vi sono più richieste in quel verso (svolare ampiamente oppure arriva fino al cilindro 1 e

File System

Fornisce l'interfaccia di accesso all'archiviazione (file). Facciamo riferimento ad un disco virtuale: ovvero di settori. Il file system ha il concetto di file: non dei limiti in termini di dimensioni, il contenuto ha una semanticà comune (interpretazione bit che comprendono il file: estensione di tipo). Si hanno anche le directory (insieme di file) e le partizioni (raggruppamento di settori nel disco). Quando si parla di file system si intende ovviamente memoria persistente.

↓ Organizzazione logica



STRUTTURA LOGICA

Facciamo riferimento al file: costituito da record logici → elemento del file quando il file è rappresentato come un vettore di settori. Al file vengono associati alcuni attributi: tipo, che può essere esplicito o implicito scritto nel descrittore di file (-note in Unix). Si ha poi l'indirizzo per individuare il file (dipende dalla tecnica di allocazione). Se un file è allocato in modo contiguo nel descrittore si riceve l'indirizzo del primo blocco dove è memorizzato il file. Se è memorizzato in modo casuale dovrei riportare tutti gli indici su cui questo è troppo pesante, quindi si usa una lista la cui testa viene memorizzata nel descrittore. Si hanno poi il nome ed altri attributi. Per quanto riguarda le directory, queste possono contenere file o altre directory costituendo una struttura ad albero con relazioni padre - figlio. L'intento è di poter modificare se ha i diritti giusti. Il fs è quindi un'aggregazione di file e directory con topologia ad albero esponenziale. I file rappresentano le foglie. La topologia del fs influenza il formato dei nomi che vengono attribuiti ai file → Si specifica il path: in path + possono esserci nomi file = . Il fs Unix ha una topologia a grafo diretto aciclico, ovvero posso raggiungere lo stesso nodo con più percorsi (percorso diversi). Questo viene utilizzato nel linking.

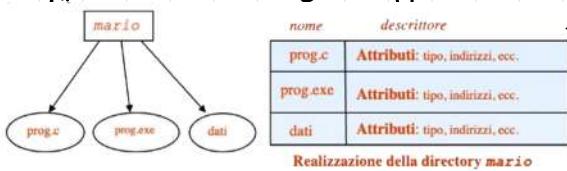
↓ gestire fs

Ahhiamo bisogno di primitive: creazione / cancellazione directory se si hanno i diritti per fare, apertura / cancellazione file. Vi sono diversi modi per indicare il contenuto di un directory: possiamo indicare i descrittori nella directory oppure metterli tutti in una tabella. Si hanno quindi primitive di listing, che permette di indicare il contenuto della directory e di navigazione, utilizzabili con i diritti giusti (lettura necessario).

ACCESSO

Dirende dell'organizzazione del fs: operazioni di scrittura, lettura e append. A livello accesso viene definito il descrittore di file: memorizza gli attributi e come i file anche i descrittori devono essere memorizzati in memoria persistente. Quando il file viene usato il descrittore viene copiato in RAM

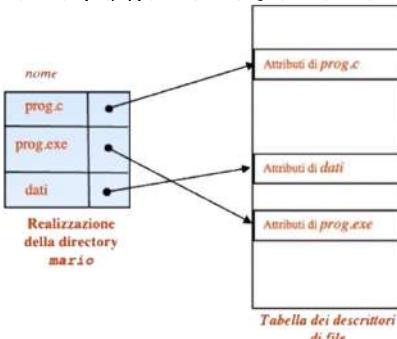
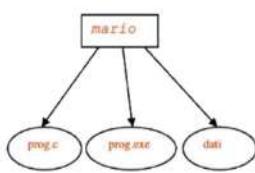
representazione directory



→ Si ha un elenco dei contenuti (nomi file / directory) all'interno. In windows la directory è rappresentata da una tabella con 2 colonne: nome e descrittore. Si ha quindi un approccio distribuito in cui

i descrittori sono contenuti nelle varie directory. → il nome (path) del file viene utilizzato per leggere tutti i descrittori riguardanti il path nome: si leggono prima i descrittori più in alto e poi si accede questo richiede però molti accessi al disco

↓ Approccio Unix



→ La tabella contiene i puntatori ai file contenuti nello directory. Questo non ci porta vantaggi in termini di accessi ma anzi ci serve fare un accesso in più in quanto si deve prima leggere l'indirizzo puntato e poi andare a quella locazione di memoria

Operazioni → lettura di record logici del file

Scrittura: inserimento di nuovi record logici nel file

↓ ▲

→ file sono visti come array di record logici a cui si accede tramite indice (semplifica operazioni)

↓ Allocazione in HDD

Più avere contiguo oppure quando (come memoria già vista). Per accedere al file il programma deve avere: diritti giusti. Nota che i record sono sul disco, la localizzazione delle informazioni può essere molto costosa

↓ Selezione

SO mantiene in memoria una struttura che mantiene informazioni sui file aperti: puntatore a file, posizone sul disco,... Inoltre, i file aperti vengono copiati in memoria centrale: memoria mapping → Devo garantire consistenza tra copia in RAM e copia sul disco. Introduciamo quindi l'operazione di chiusura: salvo le modifiche fatte in RAM sulla copia del file in memoria secondaria

↓ Metodi di accesso

1. Accesso sequenziale

File = sequenza di record logici di dimensione fissa. Il metodo di accesso è indipendente dalla tipologia di disco e della tecnica di allocazione in memoria secondaria (block: esemplificati da file in RAM).



→ Nell'accesso sequenziale il file è un "mastro" di record logici quindi non abbiamo accesso diretto tramite indice ma il file va letto dall'inizio: se devo accedere a R_i devo accedere prima agli i-1 record precedenti

readnext (f, &V): lettura prossimo record logico della sequenza → leggere n-esimo record va eseguita n volte in un ciclo
n. file ind. buffer dove copiare record letto

writenext (f, V): scrivere prossimo record logico

Per implementare l'accesso sequenziale occorre memorizzare l'indice dell'ultimo record acceduto (I/O pointer) → Non controllato dal programmatore

2. Accesso diretto

File = array in cui ogni record ha un indice su cui il programmatore può fare operazioni

↓ Funzioni

read (i, f, &V): lettura di R_i da f e copia in V

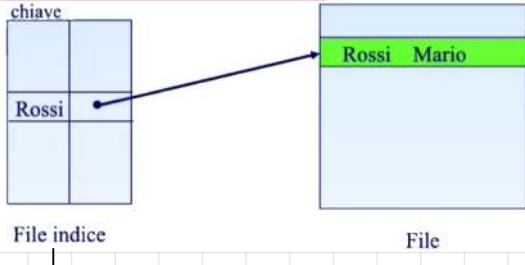
writed (i, f, V): copia di V in R_i di f

differenza da prima

Accesso sequenziale
Prima leggo i record logici precedenti e poi il record che ci interessa
for(i = 1; i < k; i++)
 readnext(f, &V);
 readnext(f, &V);

Accesso diretto
Indico direttamente il record logico a cui voglio accedere
readd(f, k, &V);

3. Accesso a indice



→ A file viene associata una tabella con 2 colonne:
 1. chiave
 2. puntatore a tale file

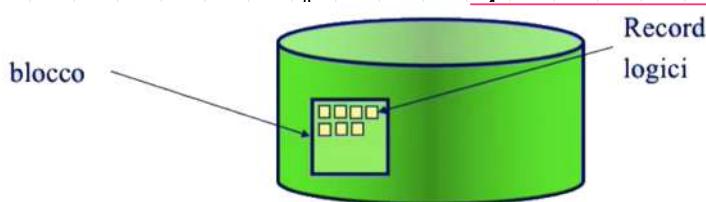
readN(f, key, &v): lettura record con chiave key dal file f

written(f, key, v)

↳ Scrittaccio: 2 accesi per 2 file. Posso eliminarne uno portando il file indice in memoria centrale.

ORGANIZZAZIONE FISICA

MJS visto come disco virtuale: array di blocchi → dim. record indipendente da dim. blocco
 ma la dim. dei blocchi può dipendere dall'hd stesso. Dobbiamo decidere come allocare i record nei blocchi (mapping). Per questo dobbiamo stabilire quanti record mettere in un blocco. Quando si fa un trasferimento del dato alla RAM si prende come riferimento il blocco.



D_b = dimensione blocco

D_r = dimensione record

$$\begin{array}{c} \text{?} \\ \downarrow \end{array} \xrightarrow{\quad} \begin{array}{c} \text{D}_b \\ \downarrow \end{array}$$

$$Xb = D_b/D_r$$

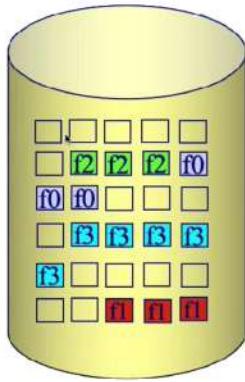
↓ metodi di allocazione dei file

1. Allocazione contigua

Configurati array di record mantenuti anche in memoria secondaria.

La tecnica di allocazione in generale ha implicazioni in termini di informazioni all'interno del dispositivo e di efficienza. Il numero di blocchi è dato da dim file (record) / Xb. Il numero è anche la dimensione della partizione da trovare nel disco virtuale (configurata).

⚠ Stesso problema contiguità RAM



Necessaria struttura che mantenga stato allocazione del disco e sono di notevole importanza le tecniche best fit e first fit. La best fit crea partitione sempre più piccole: frammentazione esterna → Necessarie operazioni di deframmentazione molto lente dato che muovo solo il disco. Nel dispositivo si scrive l'indice del primo blocco dove è allocato il file. Quando un file cresce può rimanere intatto all'interno della partizione grazie alla frammentazione ma può poi rendersi necessaria la riallocazione se la dimensione cresce molto.

↓

Vantaggi: Abbiano un I/O pointer con valore i per trovare il record nel disco basta quindi fare $B+i/N_b$. Per calcolare B non si fanno accesi al disco in quanto è in memoria (descritto), e lo stesso si può dire per gli altri termini. Si fa quindi solo un acceso per prelevare il record se utilizziamo un acceso sequenziale abbiamo I/O pointer in memoria incalcolabile al programmatore. I è però in memoria quindi è tutto cose nere. Nel caso di acceso diretto abbiamo i che è gestito dal programmatore e quindi è in memoria. Si ha quindi sempre un solo acceso al disco.

disinti: Poco scalabilità, frammentazione esterna → se usiamo first fit anziché best fit questa viene ridotta. Utilizzando worst fit la frammentazione esterna viene diminuita. Com first fit è però più semplice fare fusioni.

2. Allocazione a lista concatenata

Il file non viene allocato in modo contiguo. Si trovano i blocchi necessari (non contigui) e vi si alloca il file. In ogni blocco si mette il puntatore al successivo. Nel dispositivo si mette il primo blocco.

Vantaggi: è eliminata frammentazione esterna, allocazione facile
 ↓ Quanti accesi al disco però?

Nel dispositivo ho solo il primo blocco. Dovrò però accedere al disco tante volte quanti sono i blocchi occupati dal file. I/nb ci dà in che posizione si trova il record nella lista: $i = 1 \rightarrow 1^{\text{e}} \text{ blocco} : 1 \text{ acceso}$

$i = 1 \text{ n } i < 2 \rightarrow 2^{\text{e}} \text{ blocco} : \text{dovrò accedere al primo blocco e poi al secondo} : 2 \text{ accesi}$

Accesso sequenziale: $i = 1 / 0 \text{ pointer} = \text{memoria}$. Per leggere / scrivere un record si deve però aver fatto l'accesso ai record precedenti e quindi dovrò calcolare gli indirizzi → Dovrò tenerli in memoria per non recalcolarli. Quando l'indirizzo è n/a non si hanno accesi (memoria) e leggo puntatore al successivo (in RAM). Il caching permette di avere costo costante → Usato in UNIX

Accesso diretto: Perdo località riferimenti: i netto del programmatore e quindi si perde il caching, ogni volta dovrò fare i/nb lettura sul disco

Svantaggi. Oltre al costo, se si "zampa" un blocco non posso indirizzi successivi, al che rende difficile trovare un file corretto. Con la cancellazione "soft" viene cancellato solo il primo blocco. Si ha un minimo di overhead spaziale per memorizzare i puntatori

Lista doppiamente concatenata: puntatore a b. precedente e successivo. Nel dispositivo si ha puntatore a 1^e e ultimo blocco. Aumenta la tolleranza ai guasti interrompendo poco overhead spaziale.

FAT: Tabella centralizzata (nel disco) che contiene i puntatori ai blocchi di indirizzata per blocco e contiene puntatore a blocco successivo se occupato. Si ha > bassa tolleranza in quanto gli indirizzi sono riportati nella FAT. Inoltre, la utilizziamo per ridurre il costo dell'accesso diretto in quanto i valori dei puntatori sono scritti nella FAT. Dovrò fare i/nb accesi alla FAT che, grazie al caching, metto in RAM rendendo il costo di accesso nullo

3. Allocazione a indice

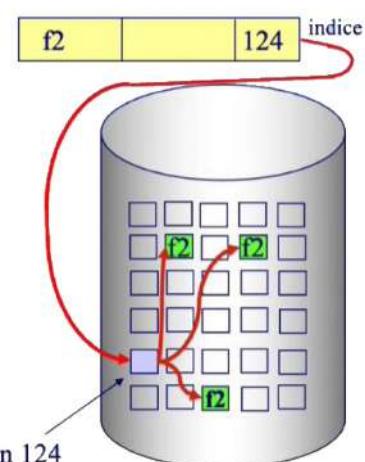
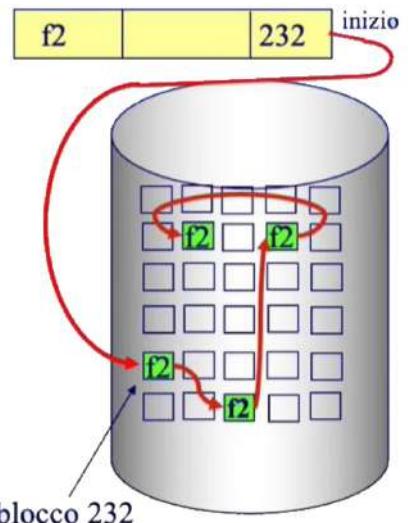
Il 1^e blocco del file contiene gli indirizzi degli altri blocchi del file. Nel dispositivo viene indicato il blocco indice di capacità costante in quanto la dimensione dei blocchi è costante. Sce soluzione non scalata in quanto se si sovraccarica l'indice non si può più espanderne il file.

Vantaggi: Non si ha frammentazione esterna ed il costo è sempre 1: basta trovare l'indice del blocco. I/nb ci serve per accedere alla tabella degli indici. Questo vale sia per accesso diretto che sequenziali. Per rendere il costo O si mette la tabella indice in memoria.

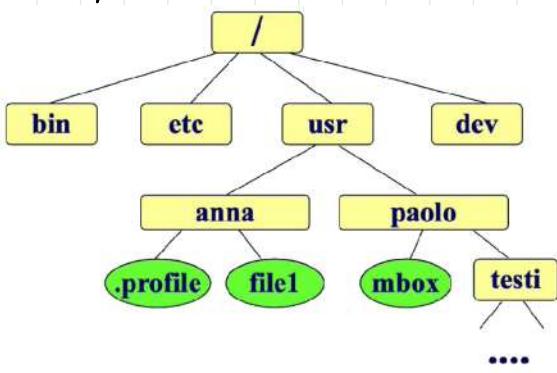
Svantaggi: non scalabilità e tabella indice costituisce single point of failure
 ↓ Utilizzo 3 tecniche

file piccoli: utilizziamo allocazione contigua

file grandi: utilizziamo allocazione ad indice



FILE SYSTEM in UNIX



contenuti nel vettore i-list allocato all'atto della formattazione la superficie huc e il blocco (512 - 1.096 Bytes):

Caso acceso diretto o il nome del file è dato dell path. (possiamo avere sicurezza assoluta che relativa). In UNIX si ha la proprietà di omogeneità: tutto è un file. Avendo quindi i file ordinari, i file che indicano il directory e quelli che indicano i dispositivi (file speciali). Ad un file possono essere associati più nomi e un tipo (non sempre esplicito) ed un solo descrittore i-node. Tutti i descrittori sono sull'HDD e sono identificati univocamente dall'i-number.

Viene divisa in 4 regioni, la cui struttura è:

- boot block: procedure initializzatore sistema (bootstrap)
- super block: fornisce limiti regione, puntatore a blochi liberi e i-node libri
- i-list

data blocks: blochi liberi per allocare file

- ordinario
- directory
- file speciali

proprietario, gruppo (user-id, group-id)
dimensione (record)

3 gradi per semplificare

data → distribuiti nell'i-node

12 bit protettive: deince dell'utente (come assegnare diritti)
numero di links

13-15 indirizzi di blochi.

INDIRIZZAMENTO: Accesso diretto ai primi 10 blochi e ad indice gerarchico per gli altri

Primi 10 blochi (non contigui) acceduti direttamente: costo = 0 accessi al disco. Se il blocco è 512 byte possiamo accedere direttamente 5KB. D'indirizzo in i-node è quello di un blocco indice. Supponiamo che l'indirizzo di un blocco sia 32 bit. Un indice può quindi contenere 512 byte / 32 bit = 16 blochi. Avendo un indirizzo di 32 bit si possono avere 2^16 blochi per file e quindi la dimensione massima del file è 2 GB. Aggiungendo l'el-essivo blocco il file si espanderà di 128.512 byte = 64KB (index). Singola = mi deisco. Se anche il puntatore 12 è un indice (doppia indirezione) il file potra essere grande 128.128.512 byte = 8MB. Se introduciamo indirezione triplice la dimensione massima sarebbe 128.128.128 = 512 byte = 1GB → Dimensione = 1GB + 8MB + 64KB + 5KB da complementi dell'accesso utilizzando gli indici come in modo logaritmico con le diverse directory.

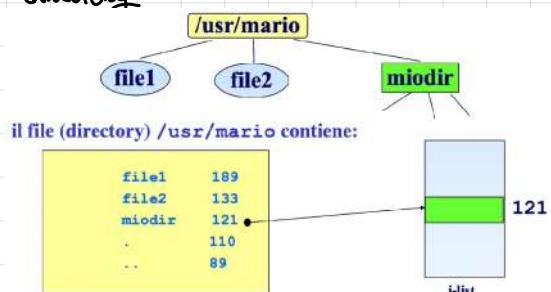
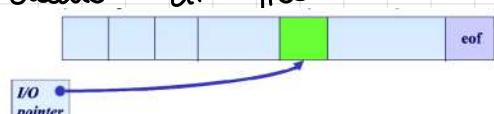


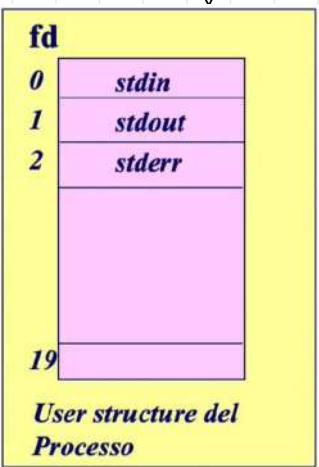
tabelle che contiene, a partire da sinistra, i nomi dei file contenuti nel suo interno e gli i-node relativi ai file.

Accesso ai file



Accesso sequenziale: file = sequenza di byte terminanti da eof. Un file può essere aperto in modalità lettura, scrittura o creazione e l'accesso è subordinato all'apertura.

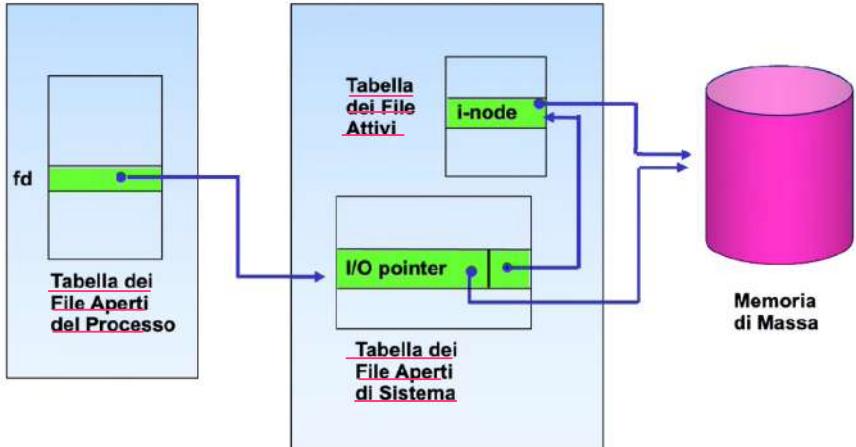
Struttura per l'accesso



Ad ogni processo viene connessa una tabella di file aperti che di default contiene stdin, stdout, stderr. Ad ogni file aperto viene associato un file descriptor e durante la vita del processo possono avere inserite altre entry. Ogni volta che vengono aperti nuovi file durante anche un'altra tabella che riporta tutti i file aperti (file = process + entry differenti). Ogni entriata contiene l'ID pointer ed il puntatore all'i-node del file nella tabella dei file attivi.

User Area del Processo

Area Dati del kernel



L'operazione di close rimuove le entry nelle tabelle dei file aperti del processo e di sistema. Nella tabella dei file attivi c'è un contatore che conta le istanze di quel file e la entry viene eliminata (l'i-node) quando il contatore è 0. Quando il file è aperto l'i-node può contenere e quindi l'i-node deve essere copiato nell'HD per aggiornarlo. Lo stesso si svolge per writing: ha senso farlo per blochi con costo di accesso più alto system call per l'accesso ai file (I/O API).

1. APERTURA

Inserendo di elemento nella 1^a posizione della tabella dei file aperti del processo e nella tabella dei file aperti del sistema. Quindi troviamo il puntatore all'i-node nella tabella dei file aperti che viene copiato se non presente.

`int open(char *namefile[], int flag[], int mode);`

La primitiva restituisce un intero e prende in ingresso:

- un vettore di caratteri contenente il nome simbolico del file (incluso il path, relativo o assoluto);
- un flag con cui specifico la modalità di accesso (Esempi di valori: O_RDONLY, O_WRONLY...), si consideri che la primitiva verificherà che il soggetto (!!!!!) possa porre effettivamente il flag indicato;
- un parametro opzionale che utilizziamo solo quando l'opzione di apertura determina la creazione del file (O_CREAT), per indicare i bit di protezione.

Nel caso in cui il parametro non venga posto quando necessario si utilizzerà un valore di default detto user mask. Il valore restituito è l'identificativo numerico del file descriptor: nel caso in cui l'apertura del file non abbia successo (per esempio nel caso in cui non abbiamo i permessi indicati) si restituisce -1.

2. CHIUSURA

`int close(int fd);`

L'interfaccia è molto più semplice. L'unico parametro di ingresso richiesto è l'identificativo del file descriptor.

Memorizza in memoria persistente il file, quando questo ha avuto delle modifiche. Il valore restituito è un intero che:

- è uguale a zero nel caso in cui l'operazione di chiusura abbia successo;
- è uguale a un valore minore di zero in caso di insuccesso.

3. LETTURA

`int read(int fd, char *buf, int n);`

I parametri richiesti sono:

- il file descriptor del file;
- l'area in cui trasferire i byte letti (puntatore all'area);
- il numero di caratteri da leggere

In caso di successo (lettura di almeno un byte richiesto) viene restituito un intero positivo che rappresenta il numero di caratteri effettivamente letti. Il valore è minore di n nel caso in cui si sia manifestato un errore da gestire a livello applicativo o quando il file contiene un numero di byte minore rispetto al numero indicato in n.

4. SCRITTURA

`int write(int fd, char *buf, int n);`

I parametri richiesti sono:

- il file descriptor del file;
- l'area in cui trasferire da cui trasferire i byte scritti (puntatore all'area);
- il numero di caratteri da scrivere

In caso di successo (scrittura di almeno un byte richiesto) viene restituito un intero positivo che rappresenta il numero di caratteri effettivamente scritti.

Entrambe le operazioni hanno come I/O pointer per accedere sequenzialmente.

Protezione e Sicurezza

PROTEZIONE: Attività che devono effettuare controllo degli accessi alle risorse logiche e fisiche da parte degli utenti

SICUREZZA: Garantisce l'autenticazione degli utenti tipicamente utilizzando credenziali

La protezione si può dividere in 3 livelli concettuali: modelli, politiche e meccanismi.

Modello di protezione

Definisce i soggetti (entità attiva) e gli oggetti (entità passiva): cosa può fare in termini di operazioni il soggetto sugli oggetti. Ad ogni oggetto vengono assegnati dinamicamente diritti soggetto per oggetto. Gli oggetti sono risorse fisiche e logiche mentre un soggetto è dato da un processo + un dominio di protezione: lista di oggetti sui quali il processo ha diverse um diritti.

$$S = \langle PIDs, (VIB, GIB) \rangle$$

processo
dominio di protezione /
ambiente di protezione

Un soggetto può accedere anche ad altri soggetti (interazione) se ne ha il diritto. Il dominio di protezione è unico per un soggetto: a parità di processo se cambia il dominio di protezione cambia anche il soggetto.

Politiche di protezione

Insieme di regole che dicono in che modo i soggetti possono accedere agli oggetti: utilizzati per le politiche di protezione

↓ 3 tipi:

DAC (Discretionary Access Control): Il controllo degli accessi è fatto a discrezione del proprietario del file: default UNIX, presente ma mancante dei diritti di default.

↓ protezione

MAC (Mandatory Access Control): Insieme di regole definite a livello centralizzato.

RBAC (Role-Based Access Control): I regole tra MAC e DAC. I diritti non vengono definiti per utente ma per ruolo che ha nel sistema. In UNIX si può implementare con i gruppi.

↓

Adottiamo il principio del privilegio minimo: Si danno soltanto i privilegi minimi fondamentali e si fornisce un meccanismo di innanzitutto dinamico.

Meccanismi di protezione

Se richiesta di accesso deve controllare se la richiesta è legittima: introduce overheard e quindi si ottimizza. In UNIX il meccanismo di protezione entra in gioco solo allo open. I meccanismi in genere non devono cambiare mentre le politiche possono cambiare più liberamente. In genere, un soggetto può accedere solo agli oggetti presenti all'interno del dominio di protezione. Vogliamo evitare di avere troppi domini disegnati per non avere troppe liste

↓

Se avessi un'associazione statica tra processo e dominio avrei domini invarianti: inadatto per implementare il privilegio minimo. Per poter definire un dominio in modo statico avrei infatti bisogno di sapere di quali oggetti ha bisogno il processo il che avrei non è possibile se non in fase di esecuzione. Utilizziamo quindi un modello dinamico, ovvero la matrice degli accessi.

	X ₁	X ₂	X ₃	S ₁	S ₂	S ₃
S ₁	read*	read	execute		terminate	receive
S ₂		owner write		control receive		terminate
S ₃	write execute		read	send	send receive	

→ Righe: soggetti

Colonne: oggetti

↓

M_{i,j} = diritti del soggetto i sull'oggetto j

⚠ Sulle colonne sono presenti anche soggetti: un soggetto può interagire con altri soggetti secondo certi diritti.

La matrice è grande, abbastanza sparsa (molti vuoti) e modificata dinamicamente

↓ Meccanismo

Ogni volta che si vuole fare un'operazione a su un oggetto X si genera la tripla (S, α, X) e viene vista la matrice. Se $\alpha \in A[S, X]$ allora l'accesso è valido, altrimenti si ha un errore di protezione. Ogni volta che viene creato un file la matrice si espande. La matrice può subire essere modificata. Un soggetto può propagare un diritto ad altri soggetti se e solo se il copy flag (*). È possibile trasferire semplicemente il diritto oppure è possibile trasferire il diritto ed il copy flag. Se si vuole controllare la propagazione si usa il 1° modo. L'ultimo caso di trasferimento è quello in cui il soggetto che trasferisce perde il diritto trasferito. Se un soggetto è owner infine può trasferire qualsiasi diritto agli altri soggetti (eccetto owner). L'owner può indicare togliere i diritti, e questo si può fare anche con il diritto control.

↓ Le regole controllano

Proprietà limitata e controllata (concurrent)

Modifica indirizzata: diritti di accesso (sharing protection)

Uno non corretto diritti di accesso da parte di un altro (trojan horse)

Implementazione della matrice degli accessi: Access Control List

Viene implementata per colonne: sono comunque tante. Ogni colonna fa riferimento ad un oggetto ed è una lista di soggetti che hanno diritto di accesso sull'oggetto. Per fare un'operazione su un oggetto il soggetto deve scorrere tutta la lista: O(n) ore n-number soggetti. Le liste devono stare in memoria e ci copia dell'ram solo quelle degli oggetti sui cui è stata fatta open. Il vantaggio della lista degli accessi è che quando vogliamo modificare i diritti su un oggetto in quanto è sufficiente modificare una sola lista. Questo non è valido nel caso di modifiche su più oggetti. Può essere comodamente avere una lista di default che contiene i diritti riguardanti tutti gli oggetti + soggetto. Questa viene cercata per prima. È una buona idea avere elementi del tipo (Vid, Gid): ciascuna di diritti specifica di specificare i diritti di un gruppo ponendo Vid = * oppure è possibile specificare politiche hanno dei ruoli ponendo Gid = * oppure di default ponendo Vid = Gid = *. Per bloccare tutto ponendo vid, gid = null. Vd, * : <insieme vuoto>

Capacità list

La lista è fatta per regole (soggetti) e ciascun elemento è un oggetto con dei diritti. Per copiare se il soggetto scorre la lista ed individuare l'oggetto qui fare un'operazione dove di nuovo per vedere se si hanno i poteri. La lunghezza è > della ACL in quanto il numero di soggetti è > di quelli degli oggetti (nr Vid - nr Gid). La lista è indirettamente più costosa dal punto di vista della modifica.

	F ₁	F ₂	F ₃	F ₄	F ₅	F ₆	PR ₁	PR ₂
soggetto	-	-	R	RWE	RW	-	W	-

→ Approccio usato in UNIX: Nei hit di proteggere i primi 3 hit (owner) sono un elemento della lista, i secondi 3 si usano per rappresentare i diritti del gruppo e gli ultimi 3 sono per rappresentare i diritti di tutti gli altri → a hit rappresentare la lista degli accessi di dimensione costante.

Rappresentate come vettori (9/12 hit) e sono posti in memoria. Secondo vengono inseriti dentro gli i-node. Questi vengono caricati in RAM come avevamo già visto in precedenza. La tabella dei file aperti di processo di fatto è una compatibility list: ho un soggetto dato da Vid + Gid ed il potere di fd è una compatibility ready rispetto di fare un'operazione. Infine, un elenco viene aggiunto tramite open che fa tutti i controlli di protezione necessari.

Quanto detto fa riferimento alla politica DAC: avere dell'oggetto decide liberamente di essere utile in base a specificazioni un controllo centralizzato per le quali usiamo un approccio MAC (Mandatory Access Control). Anche in questo caso usiamo la matrice degli accessi dando i permessi di modificare a root. Il risultato risulta però insufficiente in termini di protezione: l'accesso alle risorse dipende da un certo livello di sicurezza. Significano di avere 4: pubblico, confidenziale, secret, top secret → Diritti connessi in base al gruppo di utenti che vogliono accedere.

Modello Bell- da Padula

Abbiamo un certo numero di livelli di sicurezza, prendiamo ad esempio quelli di privacy. Questi livelli vengono attribuiti sia agli oggetti che ai soggetti.

Regole: 1. **Proprietà di semplice sicurezza**: Quando un soggetto può leggere → $SC(S) \geq SC(O)$ livello di sicurezza

2. **Proprietà STAR (*)**: Un soggetto in esecuzione al livello di sicurezza K può ricevere dolcemente oggetti dal suo livello o a quelli superiori
→ $SC(S) \leq SC(O)$

Lettura e scrittura (insieme) possono essere fatte quando $SC(S) = SC(O)$.

Non vogliamo quindi che l'informazione si propaghi da un livello K ad un livello J > K. Esempio: canale di Troia → programma installato con codice eseguibile da due nuovi di oggetti multiuso

$O_1 \quad O_2 \quad O_3$
Si R_iW, O e w

S_2 $e, o \quad R_iW, O$
↓

Si esegue O_1 : Si ha il diritto di farlo ed il codice di O_2 può prevedere di leggere O_1 . Di fatto si sta chiedendo di leggere O_1 e quindi può farlo. Se di nuovo O_2 prevede di ricevere da O_3 può di nuovo farlo → **l'informazione privata in O_1 si è trasferita nel file O_3** .

↓ Aggiungiamo Bell- da Padula

Supponiamo di avere: livelli {Ris, Pub}: Ris > Pub

$SC(S_1) = Ris, SC(S_2) = Pub$

$SC(O_1) = Ris, SC(O_2) = Pub$

Quando eseguiamo O_1 : Si può leggere O_1

Si non può leggere O_3 perché Ris > Pub

Possiamo integrare matrice degli accessi e sicurezza logico tra le condizioni di Bell- da Padula e quelle della matrice degli accessi e ricevere il risultato nella matrice degli accessi

Difetto: regola STAR. L'integrità degli oggetti può essere compromessa in quanto da la possibilità di ricevere su oggetti di livello di sicurezza superiore. Il controllo può essere fatto utilizzando la matrice degli accessi

↓ Soluzione

Sistema BiBA

→ 1. **Proprietà di semplice sicurezza**: $SC(S) \leq SC(O)$

2. **Proprietà di integrità STAR (*)**: $SC(S) \leq SC(O)$

Mutuamente esclusivo con Bell - da Padula