INDICE

Prima parte

Seconda parte

PROCESSI

- **processi** = programmi In esecuzione
- sono composti da
    - codice del programma (sezione di codice)
    - dati del programma:
        - variabili globali
        - variabili locali e non locali delle procedure (nello STACK)
        - variabili introdotte dal compilatore es program Counter (nei registri del processore)
        - variabili allocate dinamicamente (nello HEAP)
- **i programmi NON sono processi**
- i processi si possono pensare come flussi di esecuzione della computazione.
- le relazioni tra processi e programmi possono essere di tre tipi principalmente
    - programma monolitico eseguito come tale
    - programma monolitico che genera processi cooperanti
    - programmi separati eseguiti come cooperanti
- **Stato di evoluzione della computazione** = insieme dei valori di tutte le informazioni da cui dipende l'evoluzione della computazione del processo. Indica a che punto dell'esecuzione si trova il processo in questione.
    - si parla di evoluzione perche durante l'esecuzione del processo avviene una trasformazione delle informazioni
    - funziona un po' come una macchina a stati finiti:
        - stati: informazioni su cui opera il processo
        - transizioni: istruzioni che li modificano
- **stati del processo** = stati che rappresentano la modalita di uso corrente del processore e delle risorse da parte del processo. Essi sono:
    - **new** = appena creato e inizializzato
    - **running** = se e' in esecuzione e quindi le istruzioni sono eseguite correttamente
    - **waiting** = se il processo sta aspettando il verificarsi di qualche evento (es I/O)
    - **ready-to-run** = se il processo ha tutte le risorse per partire ma aspetta di esser chiamato dalla CPU.
    - **terminated** = se il processo ha terminato l'esecuzione.

- **PCB** = struttura dati in cui vengono memorizzate le principali informazioni dei processi
    - contiene molte informazioni, anche troppe ma lo scopo principale di questa struttura e' avere una mappa generale di ogni processo.
    - le PCB relative ad ogni singolo processo sono fondamentali nell'implementazione del multi tasking

- **Cambio di contesto** = è una particolare operazione del sistema operativo che conserva lo stato del processo o thread, in modo da poter essere ripreso in un altro momento. Questa attività permette a più processi di condividere la CPU, ed è anche una caratteristica essenziale per i sistemi operativi multitasking.
    - vengono salvati registri tramite una funzione **PUSH ALL**
    - lo stack pointer viene memorizzato nel PCB

A.R.B

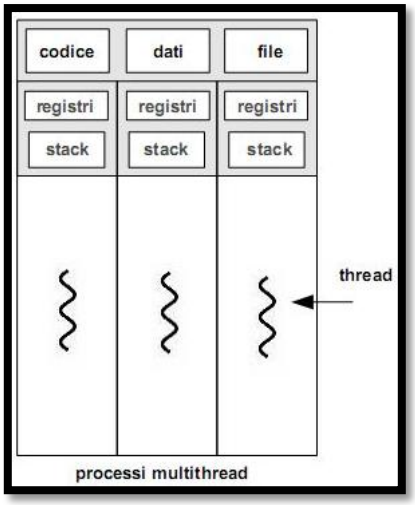
- **Multitasking** = metodologia di esecuzione dei processi avente come obiettivo quello di consentirne la turnazione sul processore massimizzandone lo sfruttamento
- per attuare il multi tasking bisogna:
    - sospendere il processo in esecuzione
    - scheduling (ordinamento dei processi in stato di pronto)
    - dispatching (selezione del processo in stato di pronto da mettere in esecuzione)
    - riattivazione del processo selezionato
- il multitasking deve tenere conto anche della tipologia di processo con cui si ha a che vedere:
    - **IO BOUND** = effettuano piu operazioni IO che computazioni
    - **CPU BOUND** = effettuano piu computazioni che operazioni IO
- bisogna bilanciare bene il numero di processi io e cpu bound eseguiti per massimizzare l'utilizzo
- la sospensione del processo in esecuzione avviene in due passaggi:
    1) **attivazione policy di sospensione**
    2) **salvataggio del contesto di esecuzione**

- **Time sharing** = sistema multi tasking a convidisione di tempo. Ha come obiettivo quello di gestire la turnazione dei processi sul processore in modo da creare l'illusione di evoluzione contemporanea delle computazioni come se ogni processo avesse tutta la CPU per se'. Si parla inoltre di:
    - **Time Slice** = intervallo di tempo massimo di uso consecutivo del processore consentito a ciascun processo
    - **Pre-emption** = rilascio forzato del processore con la sospensione del processo in esecuzione.
    - **Real-Time Clock** = dispositivo che rende possibile la sospensione di un processo ancora in esecuzione allo scadere del time slice

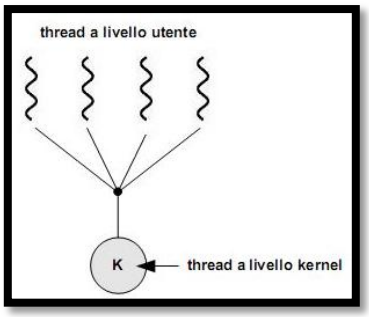
- **Spazio di indirizzamento** = porzione di memoria centrale riservata al processo dal sistema operativo alla quale nessun'altro processo puo accedere. Vengono memorizzati codice e dati.
- **Fork** e' una chiamata a sistema che permette la creazione di un nuovo processo. Un padre puo attendere che i figli terminino oppure no, inoltre puo decidere se condividere tutti/alcuni o nessun dato.
- **Exec** e' una chiamata a sistema che permette di caricare un nuovo programma nello spazio di memoria del processo che la esegue
    - in questo modo e' possibile generare due processi indipendenti.
- **Exit** e' la chiamata a sistema che permette di terminare un processo

THREAD

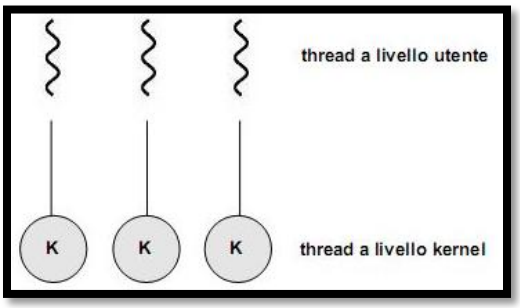
- **Thread** = gruppo di flussi di esecuzione autonomi sullo stesso programma che accedono alla stessa porzione di memoria centrale.
- un processo multithread e' un processo che e' caratterizzato da piu flussi di esecuzione di istruzioni in parallelo operanti in contemporanea e operanti con parte delle informazioni condivise in memoria centrale
    - ogni thread puo svolgere le proprie operazioni
    - i thread devono essere sincronizzati opportunatamente
    - benefici:
        - prontezza di risposta
        - condivisione nativa delle risorse
        - programmi sfruttabili dai sistemi multiprocessori
    - supporto realizzato a due livelli:
        1) **Spazio Utente**: supporto al di sopra del kernel da parte del processo e basta, il sistema operativo sa che esiste ma non ci mette mano: reso possibile dalle librerie thread e quindi nulla a che vedere col kernel.
        2) **Spazio Kernel**: supporto definito direttamente dal sistema operativo implementando ad esempio una libreria livello kernel



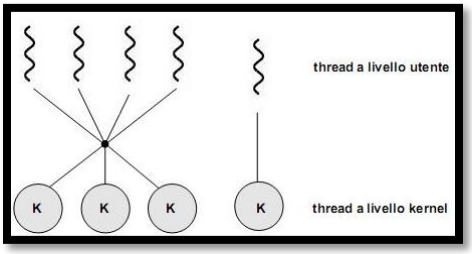
- le due tipologie sopraelencate possono essere messe in relazione tra loro:
    - **MODELLO MOLTI A UNO**
        - **riunisce molti thread utente in un unico flusso di controllo su kernel**
        - la gestione dei thread e' fatta dalla libreria nello spazio utente → il programma deve gestire la sincronizzazione in modo che il kernel thread che li gestisce ne veda solo uno
        - SVANTAGGIO: se uno dei thread si blocca anche gli altri del suo gruppo si bloccano

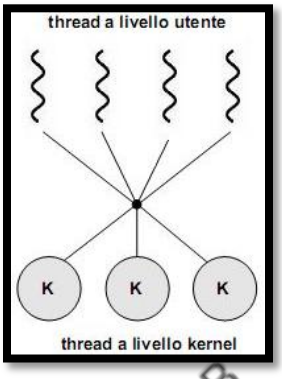


    - **MODELLO UNO A UNO**
        - **mappa ciascun thread utente in un kernel thread**
        - un thread puo essere eseguito nonostante un'eventuale chiamata bloccante da parte di un altro thread
        - sui multiprocessore consente che piu thread siano eseguiti in parallelo su diversi processori
        - SVANTAGGIO: overhead: ad ogni thread utente un thread kernel, e' quasi overkill e si perde efficienza complessiva



    - **MODELLO MOLTI A MOLTI**
        - **raggruppa in vario modo i thread a livello utente verso un numero inferiore o equivalente di kernel thread** (dipende dall'applicazione o la macchina stessa)
        - superati entrambi i limiti definiti prima per gli altri modelli
        - una variante comune e' quella di mappare molti thread di livello utente verso un numero piu piccolo o uguale di kernel thread - - - - >





A.R.B

- la cooperazione tra thread deve puo essere definita tramite i seguenti modelli:
    - **MODELLO A THREAD SIMMETRICI**
        - **i thread sono equipotenti**: e' possibile scegliere di attivarne uno qualunque per servire una richiesta esterna
    - **MODELLO A THREAD GERARCHICI**
        - **divisione in due livelli tra coordinatori e lavoratori:**
            - **Coordinatori**: ricevono richieste esterne e decidono eventualmente a quale thread lavoratore indirizzarle (generalmente mappato nel kernel)
            - **Lavoratori**: eseguono il lavoro attribuito dai coordinatori (generalmente mappato in area utente
    - **MODELLO A THREAD IN PIPELINE**
        - **ogni thread svolge una porzione del lavoro complessivo essendo specializzato in un preciso sottoinsieme delle funzioni dell'elaborazione complessiva**
        - distribuzione dei lavori ed throughput elevato, tutto funziona come una catena di montaggio
        - ogni thread torna in attesa di soddisfare nuove richieste dopo il tmepo minimale che impiega per svolgere la sua piccola sequenza di operazioni (**fare poche operazioni ma spesso**)

- ci sono diverse funzioni messe a disposizione per la gestione dei thread:
    1) **Creazione:**
        - tramite chiamata **fork()**
        - Se il thread di un programma chiama una fork verra' creato un nuovo processo con la copia di tutti i thread oppure un processo pesante composto da quell'unico thread che ha lanciato la fork (dipende dal SO)
    2) **Esecuzione:**
        - tramite chiamata ad **exec()**
        - riveste il thread di un nuovo codice rimpiazzando quello di partenza
        - permette quindi di attribuire al nuovo thread creato un compito diverso da quello del thread padre
    3) **Cancellazione**
        - cancellazione = terminazione del thread prima che abbia terminato la sua esecuzione.
        - Puo avvenire in due modi:
            - **MOD. DIFFERITA:**
                - inserendo il thread in una specie di lista nera e verificando periodicamente se sono in punti del codice in cui e' possibile terminarlo.
                - attendo semplicemente che il thread finisca la sua computazione ottenendo di fatto una terminazione ordinaria
            - **MOD. ASINCRONA:**
                - con terminazione immediata indipendentemente dall'operazione che sta svolgendo in quel momento
- Se ho due processi che lavorano insieme ed ho un thread che vuole comunicare con l'altro processo, la comunicazione può avvenire con tutti i thread del processo destinatario, con un loro sottoinsieme o con uno specifico.
- e' necessario coordinare bene kernel con libreria dei thread effettuando una schedulazione in modo da aggiustarne dinamicamente il numero nel kernel. Se pero' il kernel non supporta tale schedulazione bisogna "mascherare" i thread da processi, cosi che il sistema operativo possa sfruttare algoritmi che gia' conosce per ottenere multi tasking.
- **lightweight process (LWP)**: processi che contengono tutte le caratteristiche dei processi che lo rendono autonomo e condividono gran parte della memoria con altri thread livello utente

SCHEDULAZIONE

- tecnica con lo scopo di gestire in modo ottimale la **turnazione** dei processi sulla CPU definendo delle politiche di ordinamento.
- si basa sulla proprieta' di ciclicita' del processo: Alternamento continuo di fasi di elaborazione CPU e fasi di attesa di I/O che si concludono con una richiesta di terminazione al sistema.
    - processi I/O Bound avranno molti picchi brevi della CPU
    - processi CPU Bound avranno pochi picchi lunghi della CPU
- esistono due strategie di attivazione della schedulazione:

1) NON PRE-EMPTIVE:

- una volta assegnata la CPU ad un processo questo la tiene finche' :
    - passa in attesa
    - passa in fase di ready
    - rilascia volontariamente il processore
    - termina
- sincrona con evoluzione dello stato della computazione

2) CON PRE - EMPTIVE:

- nei sistemi multi-tasking a time sharing
- asincrona con la computazione che corrisponde allo scadere del quanto di tempo concesso al processo in esecuzione
- questo meccanismo allo stato crudo potrebbe portare anche allo stato di inconsistenza dei dati

- i S.O possono schedulare secondo tre modalita', chiamate **livelli di scheduling:**

1) SCHEDULAZIONE A BREVE TERMINE [CPU SCHEDULER]:

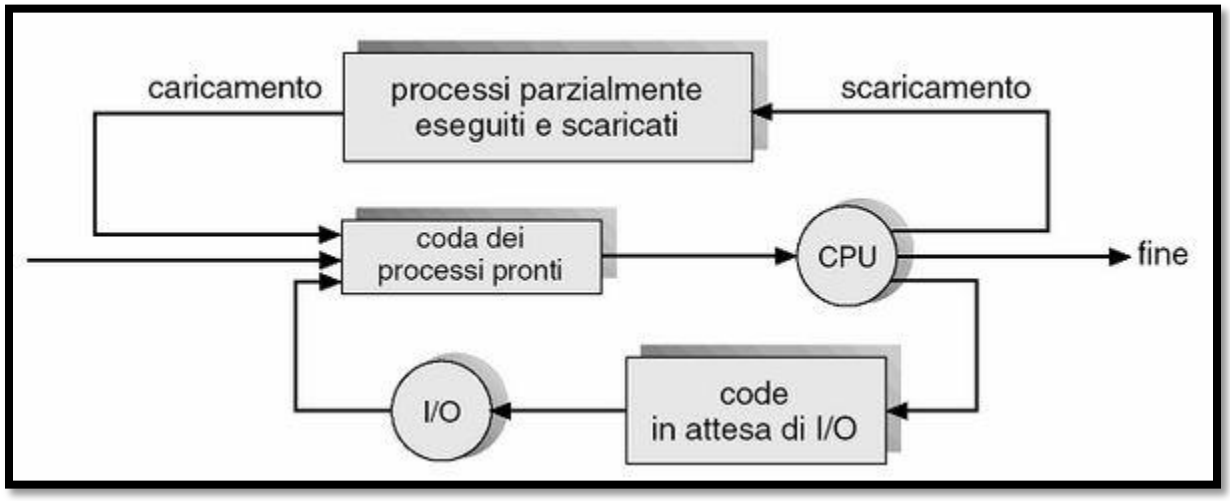
- ordina i processi gia' caricati in memoria centrale e nello stato ready to run
- si considerano solo i processi gia' caricati in memoria perche' e' piu veloce
- garantisce una turnazione rapida -> viene eseguita frequentemente
- deve usare un algoritmo efficiente

2) SCHEDULAZIONE A LUNGO TERMINE [JOB SCHEDULER]:

- ordina i processi attivati nel sistema compresi quelli in memoria di massa
- identifica il gruppo di processi che devono essere caricati in memoria centrale per l'esecuzione e in ready to run. -> gruppo formato da processi CPU bound e IO bound per massimizzare lo sfruttamento della CPU
- si richiede equilibrio per garantire interazione con utente e allo stesso tempo non avere tempi di esecuzione infiniti
- algoritmi generalmente lenti e complessi -> esecuzione non frequente per evitare di sovraccaricare il sistema

3) SCHEDULAZIONE A MEDIO TERMINE [MID TERM SCHEDULER]:

- livello intermedio tra i due precedenti
- implementato in sistemi operativi come quelli a gestione time sharing
- si propone di risolvere molte ciritcita' come la concorrenza per l'utilizzo della CPU, sbilanciamento tra processi I/O e CPU bound, scarsita' o esaurimento della memoria centrale.
- modifica dinamicamente il gruppo di processi caricati in memoria centrale e in ready to run dallo schedulatore a lungo termine al fine di adattarli all'effettiva distribuzione dei lavori che stanno compiendo -> corregge gli errori della schedulazione a lungo termine in base alla situazione attuale
- i processi rimossi dalla memoria centrale vengono spostati in un'area di memoria di massa temporanea -> procedura **swapping out**
- Tale schedulazione viene cosi rappresentata:



- Prima di decidere quale tipo di algoritmo di schedulazione utilizzare bisogna tenere conto di alcuni criteri:
  - **UTILIZZO DEL PROCESSORE**: quanto utilizzo del proc. Riesco a massimizzare
  - **THROUGHPUT**: quanti processi riesco a completare in una unità di tempo
  - **TURNAROUND TIME**: tempo di completamento di un processo
  - **TEMPO DI ATTESA**: tempo di attesa del processo nella coda dei processi pronti
  - **TEMPO DI RISPOSTA**: intervallo di tempo che intercorre tra la formulazione della prima richiesta alla produzione della prima risposta

- Per vagliare se gli algoritmi considerati soddisfino i criteri da raggiungere, si possono usare diversi metodi di valutazione:
  - **VALUTAZIONE ANALITICA**
    - Definisce in modo matematico le prestazioni in base a un carico di lavoro predeterminato
    - **PRO** : Semplice dal punto di vista logico, veloce e precisa
    - **CONTRO**: richiede numeri esatti in ingresso senza approssimazione inoltre i risultati sono validi solo per i casi studiati e non per casi generalizzati.
    - Usata per la descrizione di algoritmi di schedulazione e per produrre esempi

- **VALUTAZIONE STATISTICA**
  - permette di considerare da un punto di vista probabilistico i valori dei termini di riferimento che interessano.
  - Intrinseca dei dati rappresentativi con un certo grado di incertezza (margine di errore)
  - un tipo di valutazione statistica molto buono è la realizzazione di un simultatore che permette di modificare le variabili del sistema tra cui tempo, numero e tipi di processi coinvolti e man mano monitorare le prestazioni così da compilare delle statistiche.
  - **PRO**: Abbastanza accurate le simultazioni e molto elastiche
  - **CONTRO**: Molto costose in termini di risorse di tempo (realizzazione del simulatore)

- **IMPLEMENTAZIONE**
  - Realizzazione effettiva del sistema hardware e software
  - **PRO:** Possibile effettuare una scelta dell'algoritmo di scheduling in base ale esigenze e alle caratteristiche reali racogliendo informazioni sul carico di lavoro e utilizzando sistemi di rilevazione all'interno del S.O
  - **CONTRO:** costo elevato in termini di sforzi di implementazione dell'algoritmo e utenti che se ne sbattono altamente di cooperare (chissene se è piu efficiente, da ignorantoni basta che tutto funzioni e bona)

- Si trattano ora i principali algoritmi di schedulazione:

1) FIRST COME FIRST SERVED

- Facile: il processo che chiedi la CPU per primo la ottiene per primo
- Utilizza strutture dati in stile FIFO
- Quando un processo passa allo stato ready to run, il suo PCB viene collegato alla base della coda
- Quando invece il processore si libera, il processoin testa alla coda viene rimosso da essa e messo in esecuzione
- **PRO**: facilità di implementazione e logica
- **CONTRO**: tempo medio di attesa piuttosto lungo

2) SHORTEST JOB FIRST

- Facile: il processo piu breve viene eseguito per primo.
- Schedulazione ottimale poi chè fornisce sempre il tempo medio minimo
- Può essere implementato in due modi:
  - Con pre-emptive: appena un processo diventa pronto interrompe quello in esecuzione richiedendo una schedulazione
  - Senza pre-emptive: non interrompe.
- tutto una figata, si, ma il problema sta nel capire effettivamente QUALI sono i processi piu veloci da eseguire.
- Tendezialmente il tuto si risolve con una "predizione", basandosi sui processi precedenti, una specie di induzione.
- Problema anche di STARVATION

3) SCHEDULAZIONE A PRIORITA'

- assegna ad ogni processo un grado di priorità
- priorità = proprietà che indica quanto prima dobbiamo eseguire un processo rispetto ad altri
- il valore della priorità è espresso generalmente in numeri memorizzati in un indice associato al processo
- può essere implementato in due modi:
    - con pre-emptive: Quando un processo entra nello stato ready to run andrà a controllare e il processo in esecuzione ha priorità minore, nel caso si sostituirà a lui con una pre-emption. Se invece è running un processo di priorità maggiore, l'ultimo arrivato verrà ordinato secondo i normali criteri di scedulazione
    - senza pre-emptive: anche se il processo che diventa pronto ha priorità maggiore di quello in esecuzione, non interrompe nulla
- **Problema: Starvation**, la soluzione è detta **aging**

4) ROUND ROBIN

- Funziona come un FCFS con aggiunta di pre-emption per alternare i processi allo scadere di un quanto di tempo
- Lo schedulatore tratta la coda dei processi pronti in modo cicolare assegnando la CPU a ciascun processo per un intervallo di tempo della durata massima del time slice
- Si utilizza una coda FIFO
- ad ogni processo lo scheduler assegna un timer che genera un interrupt al termine del quanto di tempo
- ovviamente il context switch avviene sempre in un tempo inferiore al quanto di tempo del timer del processo
- prestazioni meh, dipendono dalla durata del time slice:
    - TROPPO: diventa un FCFS comune e va beh, ha i contro sopra descritti
    - TROPPO POCO: sovraccarico della gestione del sistema dovuto ai frequenti cambi di contesto

5) CODA A PIU LIVELLI

- Se sono presenti nel sistema molti processi di tipo diverso si può provare a raggrupparli per tipologie omogenee cosi da poter applicare ad ognuna di esse l'algoritmo di schedulazione piu performante
- Partiziona le code dei processi pronti in piu code di attesa separate ognuna col suo algoritmo di schedulazione.
- I processi sono inseriti PERMANENTEMENTE in una coda scelta in base a delle proprietà
- Le code a loro volta sono schedulate da un algoritmo dedicato usualmente di itpo pre-emptive a priorità fissa/time slice.

6) CODA A PIU LIVELLI CON RETROAZIONE

- è come la coda a piu livelli ma permette ad un processo di muoversi tra le code di schedulazione
- ad esempio, se un processo utilizza troppo tempo la CPU c'è il rischio che mandi in starvation tutti i processi delle code a priorità piu bassa. Si attua quindi una **POLITICA DI DEGRADAZIONE** ovvero lo si sposta nella coda di schedulazione di livello inferiore. Se un processo invece attende troppo a lungo di essere eseguito, si attuerà una **POLITICA DI PROMOZIONE**
- **PRO**: Dinamico, flessibile e completo
- **CONTRO**: Complesso as fu**

- La schedulazione è un'operazione strettamente vincolata al sistema di elaborazione, per questo la sua progettazione è definita dal tipo di sistema su cui verrà applicata:

  - **SISTEMI MULTIPROCESSORE**
    - Bisogna considerare le caratteristiche dell'architettura del sistema:
      - Omogeneità = se i processori sono equipollenti
      - Eterogeneità = se ogni processore è specializzato per una funzione diversa
      - Memoria condivisa o meno
      - Periferiche accessibili indistintamente da tutti i processori o solo da alcuni

    - In sistemi con OMOGENEITA, MEMORIA E PERIFERICHE CONDIVISE:
      - Possiamo attuare una policy di suddivisione del carico = **load sharing**
      - Si fornisce una coda separata per ciascun processore.
      - Una coda potrebbe rimanere vuota e quindi lasciare il processore inutilizzato, si previene il fenomeno utilizzando un'unica coda dei processi pronti per tutte le CPU che vengono schedulati sulla prima che si libera

    - in sistemi con CONDIVISIONE DI TUTTO TRANNE PERIFERICHE:
      - va creata una coda relativa alla periferica a cui solo quella CPU ha accesso.
      - se anche le memorie sono locali allora ci sarà una coda per ogni entità non condivisa

    - in sistemi con ETEROGENEITA
      - esistono code per ogni processore e sopra di esse una coda per ogni gruppo di processori omogenei.

    - Il **multiprocessamento** può esser:
      - **ASIMMETRICO:** tutte le decisioni di schedulazione, di attività I/O e le alre attività si sistema sono eseguite da un singolo processore definito master e tutti gli altri pensano solo ad eseguire processi applicativi
      - **SIMMETRICO:** dove ogni processore esegue il sistema operativo e i processi applicativi che in questo caso saranno schedulati da lui stesso

SCHEDULAZIONE PER SISTEMI IN TEMPO REALE

- **real-time** =sistemi in cui il tempo è un fattore critico, dove la correttezzza del risultato della computazione dipende dal tempo di risposta
- ci son due tipi di real time: **hard real time, e soft real time**

HARD REAL TIME

- deve essere garantito il completamento di un'operazione critica entro un certo intervallo di tempo. Ovvero bisogna fare in modo che un processo termini la sua computazione entro un tempo massimo garantito dalla sua attivazione
- Esistono divere tecniche di scheduling:
  - **Resource reservation** = quando un processo viene avviato, gli viene asociata un'informazione sulla quantità di tempo entro il quale deve essere terminato o deve eseguire un I/O. Lo schedulatore dovrà fare una stima del tempo di elaborazione e solo se riuscirà a garantire la terminazione del processo lo accetterà e gli prenoterà le risorse necessarie. Si presuppone ovviamente che lo schedulatore conosca con previsione i tempi di esecuzione di ogni funzione del sistema operativo

  - **Controllo di ammissione** = se i processi sono periodicim posso associare ad ognuno di essi tre informazioni che sono **il tempo fisso di elaborazione t**, **una scadenza d entro cui deve essere servito e il suo periodo p**. Posso sfruttare queste proprietà facendo sì che il processo dichiari allo schedulatore la frequenza 1/p e la propria scadenza, imponendo che uno dei due valori sia considerato come priorità. L'algoritmo di schedulazione applica poi una tecnica di controllo dell'ammissione, che verifica la possibilità di completamento entro la scadenza dichiarata con la politica di schedulazione descritta.

  - **Scheduling a frequenza monotona:** si applica ai processi periodici ed è un'estensione della politica precedente con aggiunta di pre emption. La priorità viene stabilita in modo inversamente proporzionale al periodo quindi è maggiore nei processi piu frequenti e si assume che il tempo di elaborazione di un dato processorimanga uguale ad ogni accesso alla CPU

  - **Earlies-Deadline First:** Assegna dinamicamente le priorità a seconda delle scadenza dei processi. Prima è la scadenza, piu alta è la priorità. L'assegnamento è **dinamico** poiché quando un processo diventa ready to run le priorità di tutti lgi altri processi nel sistema possono essere modificate per riflettere la scadenza del nuovo rispetto a quelli già presenti.

- **SOFT REAL TIME**
    - La computaione richiede solo che i processi critici ricevano maggiore priorità di quelli meno importanti.
    - È necessario quindi trovare un modo per definire quali sono i processi critici e quali non
    - È necessario ridurre quanto possibile la **latenza del dispatch** ovvero il tempo che intercorre tra lo stato ready to run e il running.
    - Ausilio di **pre-remption point** = punti nel quale verificare se ci sono processi critici che richiedono il processore. Nel caso affermativo si effettua pre-emption. Cio però aumenterebbe la latenza del dispatch perché bigognerebbe metterne molti e soprattutto in punti sicuri (per evitare perdite di dati)
    - Ausilio di **completa interrompibilità del kernel** ovvero si rende il kernel interrompibile proteggendolo con meccanismi di sincronizzazione
    - **Inversione di probabilità** = quando un processo che deve leggere o modificare roba dal kernel si ritrova bloccato da un processo a priorità piu bassa che sta ancora modificando la stessa roba che il processo piu importante vuole. Cio si risolve con:
        - **protocolli di ereditarietà** = viene stabilito che tutti i processi che stanno accedendo alle risose necessarie per un processo a piu alta priorità ereditano momentaneamente tale priorità finchè non terminano la computazione

COMUNICAZIONE TRA PROCESSI

- **Processo Indipendente =** processo che non convidide dati con altri processi e cui evoluzione non è influenzata da nessuno e non può influenzare nessuno.
- **Processi cooperanti =**processi cui cooperazione concorono all'adempimento di uno scopo applicativo congiunto,
- la cooperazione porta ai seguenti vantaggi:
    - condivisione delle informazioni
    - parallelizzazione
    - modularità
    - scalabilità
    - specializzazione
- per far si che le interazioni tra processi siano effettivamente funzionali, è necessario che possano sincronizzarsi.
- **Il processo di comunicazione comporta la descrizione di politiche che permettano ai processi di scambiarsi informazioni per operare in modo cooperative**
- Bisogna definire le entità coinvolte – processo mittente (P), processo ricevente (Q), canale di comunicazione
- per decidere quale tipo di comunicazione è meglio adottare bisogna tener conto di alcune caratteristiche:
    - quantita informazioni da trasmettere
    - scalabilità
    - omogeneità
    - integrazione nel linguaggio di programmazione
    - affidabilità
    - sicurezza
    - protezione

- ci sono due tipi di comunicazione:
    1) **COMUNICAZIONE DIRETTA**
        - comunicazione in cui ogni processo che voglia comunicazione deve conoscere esplicitamente il nome del destinatario o del mittente della comunicazione.
        - Tra ogni coppia di processi puo sussistere un'unica connessione

    2) **COMUNICAZIONE INDIRETTA**
        - comunicazione in cui un mittente e ricevente non si conoscono e la comunicazione avviene su punti noti ad entrambi sfruttando una struttura dati passive per contenere l'informazione.

A.R.B

- Si definiscono ora le possibili implementazioni di questa comunicazione:
    - **MEMORIA CONDIVISA**
        - Modalità di comunicazione DIRETTA e viene realizzata attraverso due metodi:
            - **VARIABILI GLOBALI**
                - Si hanno due processi con una porzione del loro spazio di indirizzamento che si sovrappone
                - Necessario un modo per sincronizzare (evitare inconsistenza dati)
                    - **Virtualizzazione area di memoria:** condivisa, ovvero il SO copia l'area commune e crea illusion della condivisione spostando l'informazione in due tempi. Bisogna aggiornare ambi spazi e quindi è abbastanza lenta
                    - **Area commune fisicamente condivisa:** SO garantisce che l'area commune appartenga contemporaneamente agli spazi di indirizzamento di entrambi I processi. Lo spazio rimane logicamente separato e protetto dal SO ma alcune porzioni sono residenti fisicamente negli stessi indirizzi. Necessarie ovviamente policy di sync.
            - **BUFFER**
                - si utilizza un buffer nel modo piu logico possible, un processo scrive qualcosa sul buffer e un altro processo lo legge.
                - Con la sincronizzazione è possible imporre accessi in mutual esclusione.
                - Come prima è possible virtualizzare il buffer oppure condividerlo fisicamente

    - **SCAMBIO DI MESSAGGI**
        - Modalità di comuniczione DIRETTA
        - Prevede che l'informazione viaggi incapsulata all'interno di messaggi
        - Sti messaggi sono un po' come una PDU
        - I msg vengono memorizzati in un buffer che il sistema operative può assegnare esplicitamente ad ogni coppia di processi o predisporne un certo numero di uso generale che mette a disposizione di chiunque ne abbia bisogno. La quantità può essere:
            - Illimitata
            - Limitata
            - Nulla : non c'è buffer disponibile per depositare il messaggio.

        - Vengono messe a disposizione alcune funzioni:
            - Invio = deposit ail messaggio in un buffer libero. Se il buffer non è libero, la funzione blocca il mittente e quando si libera
            - Invio condizionale = funzione non bloccante. Se al momento di depositare il msg non ci sono buffer, ritornerà un errore e non verrà consegnato il msg.
            - Ricezione = riceve il messagio presente nel buffer. Blocca il destinatario finchè non c'è qualcosa da leggere nel buffer
            - Ricezione condizionale = il processo preleva il msg dal buffer, se non ci sono cose da leggere ritorna un msg di errore senza bloccare il destinatario.

- **MAILBOX**
  - Sistema **INDIRETTO** in cui non c'è conoscenza esplicita tra I processi che comunicano
  - Sistema completamente ANONIMO
  - Il msg viene depositato in una MAILBOX, una struttura dati presente nel sistema operativo caratterizzata da un nome.
  - I msg contengono:
    - Dati da trasmettere
    - ID mittente
    - Nome mailbox
  - Si possono stabilire policy di accesso per evitare che tutti I processor accedano a tutti I messaggi.
  - Effettivamente è possibile creare una mailbox dedicate ad una coppia di processi, dopo che viene utilizzata viene distrutta, ridotta ad atomi.
  - Può avere capienza limitata, illimitata o nulla
  - Contiene le **stesse funzioni sopradefinite con lo stesso comportamento** con aggiunta di:
    - **Create** = crea la mailbox
    - **Delete** = distrugge la mailbox
  - Come per I buffer le polixy di sync delle mailbox dipendono dalla loro capacità
    - **ILLIMITATA**: comunicazione async
    - **NULLA**: comunicazione sincrona (chiamata anche **rendez-vou**)
    - **LIMITATA:** comunicazione bufferizzata
  - Molto utile per le comunicazioni di tipo:
    - **MOLTI A UNO:** un processo server e molti processi client. Nel caso in cui il server abortisca il SO ne esegue immediatamente una copia diversa solo per l'ID. Nessun client si accorge della situazione tranne quello che è in quell preciso istante in comunicazione col server. Tutte le richieste pendenti vengono "dimenticate"
    - **UNO A MOLTI:** molti processi server e un processo client. Utile quando un processo ha molte richieste che vuole vengano soddisfatte subito.
    - **MOLTI A MOLTI:** I diversi processi di servizion comunicano con diversi processi client.
- **FILE**
  - Ci sono due implementazioni:
    - **FILE CONDIVISI:**
      - diretta estensione della comunicazione con le mailbox
      - le mailbox sono memorizzate in memoria centrale,I file in memoria di massa.
      - Gestione del SO che obv gestisce anche problem di sync
    - **MEDIANTE PIPE:**
      - Impiega invece utilizzo dei pipe come strutture di appoggio
      - Pipe = strutture dati di tipo FIFO residenti in RAM che condivide coi file molte delle loro funzioni
  - Entrambi utilizzano le stesse funzioni delle mailbox delle quail ereditano stessi problem e vantaggi.

- **SOCKET**
  - Generalizzazione in rete delle pipe.
  - In sostanza: il SO virtualizza la comunicazione tra macchine diverse utilizzando una pipe spezzata in due porzioni ognuna residente sulla memoria central delle machine coinvolte.
  - I messaggi possono avere dimensione fissa o variabile a seconda delle applicazioni che le utilizzano, e sono ordinate in mod. FIFO.
  - I socket utilizzano una porta per leggere e una per scrivere cosi da non avere un solo canale intasato tra letture e scritture.
  - La connessione può avvenire tramite gestione del SO oppure senza.
  - Esiste infine una bellissima tecnica che può andare a fare in cu. Chiamata MuLtICaST

CONCORRENZA

- **Concorrenza** = situazione che sussiste quando piu processi richiedono accesso a risorse condivise usabili solo in mutual esclusione. I processi in considerazione vengono chiamati concorrenti.
- **Mutua esclusione** = situazione per la quale non si possono effettuare operazioni contemporaneamente su una determinate risorsa
- **Sincronizzazione** = insieme di policy e meccanismi che si occupano di garantire accesso in mutual esclusione nei confronti dell'accesso a risorse fisiche o informative. E' un problema del multitasking
- Problema **produttore consumatore**
- **Sezione critica** = porzione di codice che può generare corse critiche se eseguita in modo concorrente.
- è necesario individuare la sezione critica per poter creare un protocollo che possa essere utilizzato per bypassare I problem di inconsistenza. Deve soddisfare:
    - **MUTUA ESCLUSIONE** = se un processo sta eseguendo la sua sezione critica, nessun altro può farlo
    - **PROGRESSO** = stabilisce che solo chi non sta eseguendo la propria sezione critica può concorrere con gli altri per accedervi.
    - **ATTESA LIMITATA** = bisogna fare in modo che nessun processo attenda troppo a lungo di evolversi
- SINTESI: le sezioni critiche di codice devono avere acceso esclusivo alle variabili condivise e devono sruttarle in modo rapido perchè altri processi che usano la stessa risorsa non debbano attendere indefinitamente. Ciò si ottiene realizzando processi cooperanti che superino le criticità con un'opportuna sincronizzazione dell'evoluzione delle loro computazioni
- ci sono diversi approcci alla sincronizzazione:

1) VARIABILI DI TURNO

- Variabili condivise tra processi che interagiscono per acedere in modo concorrente ad una risorsa stabilendone il turno di uso.
- Ci sono diversi algoritmi che però non inserisco qua

2) VARIABILI DI LOCK

- Variabile condivisa che definisce lo stato di uso di una risorsa ovvero quando è in uso da parte di un processo che è nella sua sezione critica.
- Non sono piu I processi ad alternarsi ma è la risorsa stessa a dire se è disponibile o no.
- Le variabili di lock sono scalabili, non importa quanti processi abbiamo, la risorsa si gestisce da sola
- Funzionano in modo abbastanza semplice:
    - Disabilito le interruzioni
    - Leggo la variabile di lock
    - Se lock==0 , metto lock=1 e riabilito le interruzioni
    - Se lock==1, riabilito le interruzioni e metto il processo in attesa che la risorsa si liberi
    - Per rilasciare la risorsa pongo lock=0
- Si disabilitano le interruzioni perchè le operazioni di lettura ed eventuale lettura di un lock sono realizzate con una sequenza di istruzioni macchina e quindi interrompibili. Se non si disattivano si finirebbe per avere una corsa critica nello stesso sistema che si usano per prevenirle.
- Esiste l'istruzione atomic **TEST-AND-SET** che traduce la sequenza di istruzioni precedent in una sola
- Questa soluzione ha il limite che essendo TEST-AND-SET una istruzione atomica, deve essere il progettista del processore ad implementare e non al programmatore.

3) SEMAFORI

- Sono una struttura dati gestita dal sistema operative che rappresenta lo stato di uso della risorsa condivisa: ha due valori:
  - **1 se è libera**
  - **0 se è in usa**
- sono ad un'astrazione superiore dei lock, e quindi offrono direttamente delle funzioni:
  - **acquire(S)** = richiede utilizzo della risorsa
  - **release(S)** = rilascia la risorsa
- l'acquire crea attese attive sul semafoto S ovvero processi che pur essendo in stato di attesa continuano a effettuare computazioni per verificare se la risorsa si è liberate. Ciò implica uno spreco di CPU e questo evento si chiama **SPINLOCK**
- lo spinlock per lunghe attese NON va bene, bislogna gestirlo:
  - quando un processo esegue un **acquire(S),** se la risorsa non è disponibile passa in stato di wait e viene accodato nella coda di attesa del semafoto
  - quando un processo esegue una **release(S)**, la risorsa viene rilasciata e viene attivato il primo processo della coda di attesa di S a cui viene consentito l'accesso
- bisogna stare attenti comunque con la lista di attesa per non creare DEADLOCK
- bisogna prestare attenzione all'odine in cui avvengono le chiamate di sistema di prenotazione e rilascio delle risorse o potrei avere attese circolari senza rilascio
- pericolo di **starvation**
- I semafori ovviamente sono generalizzati, se metto che 4 processi possono accedere ad una risorsa allora N non sarà piu' 1 ma 4.

4) MONITOR

- Costrutto che incapsula un insieme di operazioni definite dall'utente su cui garantisce la mutual esclusione.
- Formulato a livello di linguaggio di programmazione e al suo interno vengono formulate una serie di procedure usate per entrare/uscire dalla sezione critica.
- Perchè un processo possa accedere alle risorse deve soddisfare alcune condizioni, rappresentate da variabili di tipo condition definite dal programmatore. Se il processo associato ad una di esse la verifica, prosegue. Altrimenti sta in wait fino a quando un processo con le stesse condizioni lo risveglia con un signal.
- Supponiamo di avere P e Q associati ad una condizione X con Q in attesa. Se P invoca una x.signal e viene concesso al processo Q di riprendere, ci si può comportare in due modi:
  - **Segnala e aspetta**, P attende finchè Q lascia il monitor o aspetta un'altra condizione
  - **Segnala e continua**, il contrario

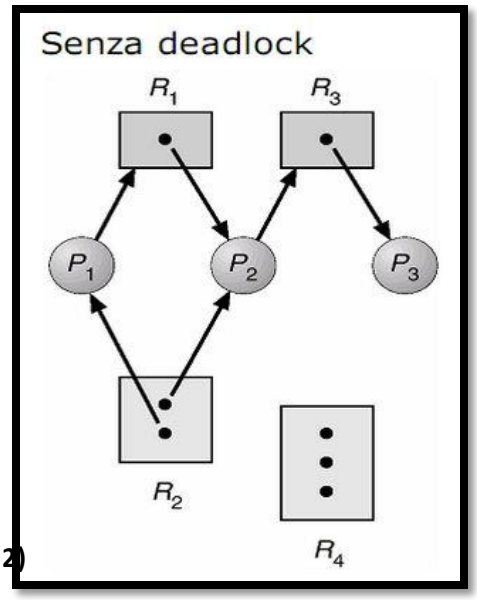
DEADLOCK

- un gruppo di processi si dice in stato di **Deadlock** quando ogni processo del gruppo sta aspettando un evento che può essere generato soltanto da un altro processo del gruppo.
- I processi non terminano mai la loro esecuzione e le risorse di sistema a loro assegnate rimangono bloccate
- Si ha un deadlock solo se si verificano queste 4 condizioni:
    - **Mutua Esclusione**
        - Almeno una risorsa deve essere usata in un modo non condivisibile.
        - Se uno degli N processi chiede tale risorsa ma già uno la sta usando, esso deve andare in attesa.
    - **Possesso e Attesa**
        - Un processo che possiede già almeno una risorsa per terminare la computazione deve attendere che se ne liberino alter occupate
    - **No Pre-Emption**
        - se si consentisse pre-emption allora non si potrebbe mai bloccare l'accesso a una risorsa dato che chi la utilizza verrebbe forzatamente terminato
    - **Attesa Circolare**
        - Molto simile allo stallo alla messicana ma con risorse, uno ha bisogno di rob ache ha l'altro e l'altro ha bisogno di roba che ha un altro ancora etc etc

- Per verificare con precision la sussistenza di un deadlock si utilizza il **grafo di allocazione delle risorse** ovvero un grafo costituito da :
    - **Nodi** : rappresentano sia I processi P che le risorse R
    - **Archi di richiesta** : vanno da P ad R
    - **Archi di assegnazione** : vanno da R a P

- Se I grafi contengono dei cicli allora POTREBBERO esserci stalli
- Tre situazioni di seguito:

1) **SENZA DEADLOCK**



il processo P1 ha richiesto R1 che però è stato assegnato a P2 quindi rimane in attesa.

La Risorsa R2 mette a disposizione due istanze che pur essendo identiche possono essere usate in parallel, non ci sono conflitti dato che il principio di mutual esclusione si applica solo sulla stessa istanza.

Il processo P3 spezza la possibilità di formare una attesa circolare non avendo richieste.

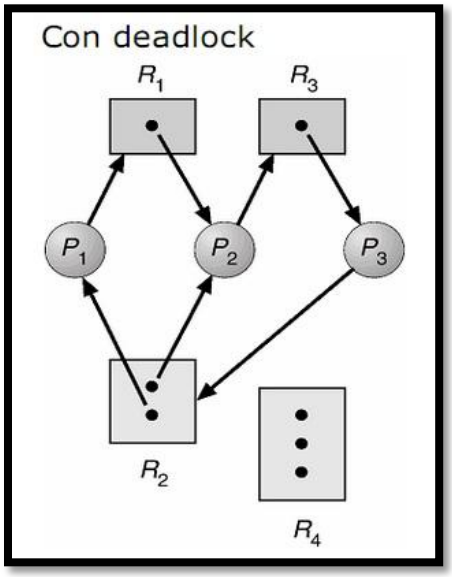
Una volt ache P3 termina l'utilizzo di R3 lo rilascia sbloccando P2 e quindi P1

h

2)

A.R.B

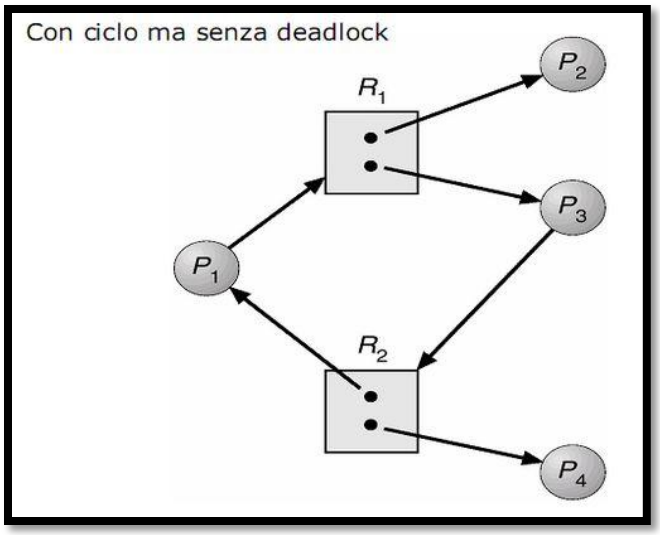
3) CON DEADLOCK



Il processo P1 richiede R1 che è assegnato a P2 quindi rimane in attesa.
La risorsa R2 mette a disposizione due istanze che pur essendo identiche possono essere usate in parallel. Non ci sono conflitti per ora.
Il processo P3 richiede R2 che non ha istanze, si mette in attesa.
Si formano dei cicli:
P1 -> R1 -> P2 -> R3 -> P3 -> R2 ->P1
P2 -> R3 -> P3 -> R2 -> P2
P1 e P2 e P3 sono in deadlock.
P2 sta aspettando R3 che è tenuta da P3 il quale sta aspettando che P1 o P2 rilascino una istanza di R2, inoltre P1 sta apsettando che P2 liberi R1.

a

4) CICLO MA SENZA DEADLOCK



Esiste un ciclo tra P1 -> R1 -> P3 -> R2 ->P1
Dimostrazione che se una risorsa ha piu istanze posso avere cicli ma non deadlock
P2 e P4 prima o poi termineranno la loro computazione e rilasceranno un'istanza che potrà essere utilizzata da P1 e P3 per superare lo stallo.

- Ci sono diversi modi di gestire i deadlock:
  - **IGNORARE** = si ignora il problema dei deadlock. Sta roba ignorante as fuck funziona ovviamente solo in sistemi in cui gli stalli sono MOLTO rari. Nel caso si verificassero bisognerebbe abortire o riavviare tutto il sistema.
  - **PREVENZIONE =** garantisce a priori che le richieste non generino situazioni di stallo, Consiste in un insieme di metodi atti ad accertarsi che almeno una delle condizoini necessarie viste prima non possano verifricarsi
  - **EVITARE =**non impedisce ai processi di richiedere risorse ma se queste potrebbero causare stalli le blocca.
  - **RILEVAZIONE E RECUPERO =**usa algoritmi per rilevare deadlock ed eliminarli

A.R.B

- Si osservano ora nel dettaglio:

1) PREVENZIONE DEL DEADLOCK

- In sintesi la logica è quella di trovare un modo di soddisfare una delle 4 condizioni. Essendo che tutte sono in AND, se una viene "risolta" il deadlock cessa di esistere.

    - **Mutua esclusione** = verificare che la mutual esclusione sia applicata solo su quelle risorse intrinsicamente non condividibili. Estendere la mutual esclusione a tutte le risorse aumenta il rishcio di stallo

    - **Possesso e attesa =** bisogna fare che ogni volt ache un processo chiede risorse non ne possegga già qualcuna. Ci sono due tecniche:

        - <u>un processo chiede e ottiene tutte le risorse prima di iniziare l'esecuzione</u>. Se il SO non può dargliele semplicemente non lo avvia.
        - <u>Un processo che possiede alcune risorse e vuole chiederne altre deve prima rilasciare tutte quelle che già detiene</u>

    - **Pre-eption =** ci sono due tecniche per evitare la pre-emption:

        - Se un processo detiene delle risorse e ne chiede altre per le qualii deve attendere allora deve rilasciare anticipatamente tutte le risorse possedute che vengono aggiunte in una lista a tutte quelle che il processo sta ancora attendendo.
        - Se un processo chiede risorse si verifica che esse siano disponibili, se sono già associate ad un altro processo che non le usa , viene imposto a quest'ultimo di rilasciarle anticipatamente e evengono aggiunte alla lista delle risorse per cui il processo richiedente è in attesa.

    - **Attesa circolare** = bisogna dare un ordinamento globale su tutte le risorse Ri, in modo che si può sapere se una è maggiore o minore di un'altra in base al suo indice. Ad esempio l'indice di un file da stampare è minore di quello della periferica (piu importante la stampante del file da stampare...). Non basta fare così, bisogna attuare dele policy:
        - se un processo chiede un certo numero di istanze della risorsa Rj e detiene solo risorse Ri con i<j, se le istanze sono disponibili gli vengono assegnate altrimenti deve attendere.
        - Se invece richiede istanze della risorsa Rj e detiene risorse Ri con i > j allora il processo dovrà prima o poi rilasciare tutte le sue risorse Ri quindi richiederle tutte, vecchie e nuove comprese.

            "Ad esempio se il processo P detiene la risorsa R25, potrà chiedere solo risorse indicizzate da R26 in su, e non da R25 in giù, a meno che non rilasci tutte quelle che ha già. Questa tecnica impedisce che più processi possano attendersi l'un l'altro: gli indici lo impediscono."

2) EVITARE IL DEADLOCK

- applica alcune regole su come devono essere fatte le richieste di accesso alle risorse
- come prima bisognerebbe assicurare la non-presenza di almeno una delle condizioni prima descritte.
- Queste condizioni sono molto restrittive e quindi portano ad un basso rendimento.
- Una della alternative possibili è **verificare a priori se la sequenza di richieste e rilasci di risorse effettuate da un processo porta a stalli** tenendo anche conto delle sequenze dei processi già accettati nel sistema. Ciò significa evitare il deadlock e richiede che i processi forniscano al S.O alcune informazioni sul loro comportamento future cosi che questo posa sapere in qualsiasi momento:
    - **Il numero Massimo di risorse per ogni processo**
    - **Numero di risorse assegnate e numero risorse disponibili**
    - **Sequenza di richieste e rilasci da parte di un processo**

- **Stato sicuro =** se il sistema operative può allocare le risorse richieste da ogni processo in un certo ordine **garantendo** che non si verifichi deadlock
- **Sequenza sicura =** sequenza di processi tale che le richieste di ognuno possono essere soddisfatte con le risorse già disponibili o in via di rilascio dai processi precedent ad ogni singolo processo i-esimo.
- **stato sicuro = solo se esiste una sequenza sicuro.**
- Ci sono due possibili soluzioni:
    - **VARIANTE DEL GRAFO DI ALLOCAZIONE DELLE RISORSE**
        - Usato per istanze **SINGOLE**
        - Se abbiamo un sistema di allocazione delle risorse con un'unica istanza, si può utilizzare una versione special di grafo di allocazione delle risorse per INDIVIDUARE i deadlock.
        - Si inserisce semplicemente un nuovo arco, l'arco di PRENOTAZIONE
        - **Arco di prenotazione** = indica quail risorse saranno prima o poi necessarie ad un processo perchè possa portare Avanti la propria computazione

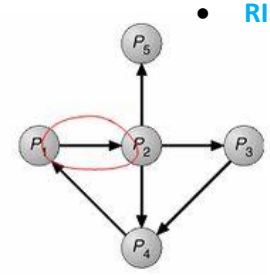
    - **ALGORITMO DEL BANCHIERE**
        - usato per istanze **MULTIPLE**
        - fa uso di alcune strutture dati per mantenere le informazioni sui processi (n=numero processi, m=numero risorse):
            - **available =** array di m elementi contenente il numero di istanze disponibili per ogni risorsa
            - **allocation =** matrice n*m contenente le allocazioni per ogni processi
            - **max =** matrice n*m contenente per ogni processo le risorse massime che può utilizzare
            - **need =** matrice n*m contenente per ogni processo le risorse massime che può ancora richiedere
        - si analizza ora l'algoritmo:
            - un processo fa richiesta, si controlla se è valida e se è soddisfacibile
            - si simula l'assegnazione della risorsa con le strutture dati

- si esegue **l'algoritmo dello stato sicuro**:
  - si cercano tutti i processi il cui fabbisogno può essere soddisfatto con le attuali disponibilità piu le risorse assegnate ai processi analizzati precedentemente che le rilasciano al termine
  - se alcuni processi rimangono scoperti lo stato non è sicuro, si ripristaninano i vecchi valori delle strutture dati e il processo deve attendere.
  - In caso contrario il processo ottiene le risorse richieste

3) RILEVAMENTO E RISOLUZIONE DEL DEADLOCK

- Consente di aumentare l'utilizzo delle risorse
- Si analizzano ora le due "fasi"

- **RILEVAZIONE**



  - in caso di di istanze single delle risorse, si può usare il **grafo di attesa**: variante del grafo di allocazione con tutti i nodi di risorsa collassati per mostrare le attese tra i processi. Se nel grafo ci sono cicli, quei processi sono coinvolti in un deadlock.
  - In caso di istanze multiple si usa un algoritmo simile a quello del banchiere. Utilizza le stesse strutture dati piu matrice **request (n*m)** che contienele attese di ogni processo. Si cercano tutti I processi le cui richieste possono essere soddisfatte con le attuali disponibilità piu le risorse detenute dai processi analizzati precedentemente. Se alcuni processi sono scoperti, essi sono coinvolti in un deadlock.

- **RISOLUZIONE**
  - O si terminano tutti i procesi ocinvolti
  - Si possono terminare uno per volta i processi in base a qualche criterio finchè non si resolve il deadlock.

ARCHITETTURE DEI SISTEMI OPERATIVI

- Un Sistema Operativo può essere realizzato in molti modi diversi:

1) **MONOLITICO**
   - Esegue **tutte** le funzioni del SO
   - Tutte le funzioni e le strutture sono accessibili da qualsiasi punto del kernel.
   - Poco scalabile e manutenzione difficile, un semplice bug potrebbe creare un crash generale
   - Sistema compatto veloce ed efficient, quindi adatto a sistemi **semplici**

2) **GERARCHICO**
   - Organizza il sistema su livelloi funzionali, una funzione di un certo livello può chiamare solo funzioni di livello inferior
   - Come prima comunque non c'è separazione tra le component del SO.
   - Poco scalabile e manutenzione difficile, ma leggermente piu facile del monolitico

3) **STRATIFICATO**
   - Scomposto in un certo numero di livelli.
   - Ogni livello è un modulo che implementa un component del S.O, nasconde i propri dettagli implementativi ad altri e può comunicare con il livello sottostante attraverso un'interfaccia ben definite.
   - Scalabile e debug semplificato
   - Concetto fondamentale: **Modularità**
   - Una chiamata ad una determinate funzione di un certo livello deve passare per tutti i livelli soprastanti

4) **MICROKERNEL**
   - Rimuove dal kernel tutte le funzioni non indispensabili, implementandole come servizi a livello utente.
   - **Kernel molto piccolo con funzionalità minime:** gestione processi, memoria e comunicazione
   - **Scopo:** permettere comunicazione tra programme client e servizi oltre che tra servizi stessi
   - Comunicazione tramite messaggi scambiati dal kernel
   - **Grande sovraccarico di gestione**
   - Separazione tra Meccanica e Politica (si separa il come si fa da cosa si fa)
   - Facile **modificabilità** e massima **portabilità** e **affidabilità**

5) **MODULARE**
   - il **kernel** possiese solo un insieme minimo di funzioni che si collegano dinamicamente ai moduli che implementano le varie funzioni del sistema
   - le implementazioni di ogni modulo sono nascoste ma gli altri hanno interface ben definite per comunicare.
   - Comunicazione **diretta**, **modificabilità**, massima **portabilità**,**affidabilità**

A.R.B

6) MACCHINE VIRTUALI

- Copia esatta dell'hardware sottostante ad ogni macchina virtuale eseguita in modalità utente, ognuna della quail esegue un S.O
- **Illusion** di fornire ad ogni utente una macchina dedicate
- **Schedulazione e memoria virtual** si occupano di suddividere tempo diCPU e memoria tra le VM.
- **Isolamento di ogni macchina virtual dalla macchina reale.**
- Impossibile condividere risorse dirette tra le VM, ma è possible condividere un disco virtuale tra le machine.
- Possibile inoltre creare una **rete virtual** tra le machine.
- **Calo di prestazioni dovuto allo strato di virtualizzazione**

A.R.B

PAGINAZIONE

- la memoria paginata è una tecnica di gestione della memoria centrale usata da molti SO che permette
    - **multiprogrammazione**
    - utilizzo di **porzioni di memoria non contigue** per un processo.
    - non tenere l'intero processo in memoria ma di scambiare pagine di memoria tra mem centrale e area di swap quando necessario.
    - separa la visione dello spazio logico(illimitato) dei processi da quello fisico (limitato)
- la memoria fisica è divisa in **FRAME** di dimensione fissa
- la memoria logica è divisa in **PAGINE** di dimensioni pari a quelle dei frame
- **tabella delle pagine** = tabella dedicata al singolo processo che contiene le traduzioni pagina-frame
- gli indirizzi logici generati dalla CPU sono formati da **numero di pagina (p) + offset nella pagina (d).**
    - **p** viene usato come indice nella tabella delle pagine
    - **d** viene usato per cercare all'interno della pagina il dato richiesto
- il S.O decide dove allocare le pagine e di conseguenza ha bisogno di mantenere una tabella dei frame per sapere quali sono vuoti e quali no.
- la paginazione permette inoltre di
    - **eliminare la frammentazione esterna** (poiche i frame sono di dimensione fissa)
    - **ridurre notevolmente la frammentazione interna**
- la traduzione di indirizzi logici in fisici è un'operazione svolta dalla **MMU** programmata dal S.O ad ogni cambio di contesto
- con page tables piccole, le pagine possono essere caricate in registri ad alta velocità, visto che però spesso sono grandi, il SO carica solamente l'indirizzo → ogni accesso in memoria ne richiede un altro per recuperare la traduzione. Per ridurre il calo di prestazioni si usa un TLB, cache che mantiene le traduzioni recenti evitando maggior parte dei doppi accessi.
- nella tabella delle pagine ogni riga mantiene
    - bit di protezione = indicano se la pagina è in sola lettura, lettura/scrittura o sola esecuzione
    - bit di validità = marca pagine caricate e rileva accessi illegali
- **PAGE FAULT** = errore generato da accessi illegali alla page table
- alcuni sistemi memorizzano anche la lunghezza della tabella delle pagine di un processo poiche' raramente questi usano tutto lo spazio di indirizzamento.
- alcune pagine possono essere **condivise tra processi**: tramite la paginazione basta mappare lo stesso frame nella tabella delle pagine di piu' processi. Cio' porta a due limitazioni:
    - codice non deve essere **automodificante**
    - codice deve trovarsi nella **stessa locazione logica** in tutti i processi

- le page table hanno **strutture ben definita**:

1) **paginazione gerarchica**
    - la tabella delle pagine e' a sua volta paginata anche piu volte.
    - Con 2 livelli si ha un indirizzo logico formato da p1 (indice tabella esterna delle pagine), p2 (offset nella tabella puntata da p1) e d (offset).
    - Aumenta il numero di accessi necessari per la traduzione, riducendo le performance.

2) **tabella delle pagine con hashing**
    - si applica una funzione di hash al numero della pagina contenuto nell'indirizzo virtuale identificando un elemento nella tabella di hashing
    - l'entry puntata è una lista concatenata di elementi formati da traduzione pagina-frame. Si scandisce la lista finchè non si trova la traduzione cercata

A.R.B

3) **tabella delle pagine invertita**
    - si tiene solo una tabella globale che tiene per ogni frame
        1. identificatore dello spazio di indirizzamento (ASID) cui appartiene.
        2. numero di pagina logica in quello spazio
        3. bit di protezione.
    - si risparmia memoria ma vi è una complicazione nella condivisione e può esser necessario scansionare tutta la lista per una sola traduzione.

- **non sempre è necessario tenere l'intero processo in memoria centrale per l'esecuzione**. Si possono caricare solo alcune pagine ed avviare l'esecuzione:
    - se tenta di accedere a pagina non caricata genera un **page fault.**
    - Il sistema operativo recupera quella pagina richiesta dal disco.
    - la nuova pagina viene inserita in un frame libero oppure al posto di una pagina del processo che verra' messa in area di swap.

SEGMENTAZIONE

- tecnica di gestione della memoria centrale che consente
    - **multiprogrammazione**
    - utilizzo di **porzioni di memoria non contigue** per un processo.
    - non tenere l'intero processo in memoria ma solo i segmenti usati nell'immediato futuro scambiandoli tra mem e area di swap quando necessario.

- la memoria fisica e' divisa in **frame** di dimensione variabile identificati da un **numero**
- la memoria logica viene divisa in **segmenti** ognuno inserito in un frame di egual dimensione.
- i segmenti possiedono **informazioni di tipo** → permette di dare consistenza logica alle porzioni di spazio di indirizzamento dei processi consentendo la **tipizzazione**.
- gli indirizzi logici prodotti dalla CPU sono composti da: numero_segmento(s) + offset (d):
    - **s** viene usato come indice nella tabella dei segmenti
    - **d** viene usato per cercare all'interno del segmento il dato richiesto
- la traduzione e' fatta in hardware dalla MMU che ad ogni traduzione controlla anche i **bit di protezione** nella tabella dei segmenti per impedire accessi a segmenti inesistenti o altri accessi illegali che genererebbero una **segmentation fault**
- la MMU viene programmata dal SO ad ogni cambio di contesto, con tabelle piccole esse possono essere caricate in registri ad alta velocita' ma spesso sono grandi quindi il SO carica solo l'indirizzo in cui si trova la tabella dei segmenti.
- Per ogni accesso bisogna farne un altro per recuperare la traduzione → utilizzo di un TLB per evitare calo di prestazioni.
- **ASID** = identificatore dello spazio di indirizzamento
- ogni entry mantiene un ASID cosi che possa fare caching di tutte le traduzioni per piu processi.
- Poiché ogni segmento contiene una porzione diversa del programma, è probabile che i suoi elementi siano usati allo stesso modo quindi ogni segmento ha associato anche dei **bit di protezione**
- la condivisione di codice o dati e' abbastanza semplice, si **mappa lo stesso frame nelle tabelle dei segmenti di piu processi** → utile per condividere librerie tra procesi. Ci sono due vincoli:
    - codice non automodificante
    - segmento condiviso deve avere lo stesso numero di segmento logico nei vari processi altrimenti non potra' autoreferenziarsi.
- Quando si inserisce un nuovo segmento in memoria si deve trovare uno spazio sufficientemente grande per contenerlo. Esistono varie strategie:

    1) **BEST FIT** = piu piccolo sufficiente
    2) **FIRST FIT** = primo sufficiente
    3) **WORST FIT** = piup grande disponibile.

- le prime due tecniche sono molto utilizzate, nonostante cio' si crea **molta frammentazione esterna** che si puo' ridurre compattando periodicamente la memoria. Non c'e' frammentazione interna dato che i segmenti sono di dimensione decisa in base alle necessita' del programmatore.

- **Paginazione** e **Segmentazione** hanno entrambi vantaggi e svantaggi, Si possono combinare le due tecniche in una tecnica chiamata **segmentazione con paginazione**:
    - memoria fisica divisa in frame di dimensione fissa (evita frammentazione esterna)
    - identificazione dei frame liberi
    - scelta del frame in cui caricare una pagina
    - gestione semplice dell'area di swap
    - memoria logica divisa in segmenti a loro volta divisi in pagine inserite nei frame
    - indirizzi logici formati da
        - numero segmento (s) + pagina nel segmento (p) + offset nella pagina (d)
    - indirizzi fisici formati da
        - numero di frame (f) e offset (d)

    → vantaggi della segmentaizone in merito alla protezione, condivisione e tiupizzazione

A.R.B

VIRTUAL MEMORY

- la memoria virtuale permette di
    - eseguire processi **non completamente in memoria centrale**
    - eseguire processi **piu grandi della memoria fisica**
    - **condividere memoria facilmente**
- un processo viene avviato caricando 0 o piu pagine in altretanti frame, quando esso cercherà di accedere ad una pagina che non è caricata, verrà generato un **page fault** poiché il bit di invalidità è settato e il SO dopo aver verificato che non si tratta di un accesso illegale, caricherà la pagine richiesta inserendola in un frame e poi riavvierà la richiesta interrotta. Queste attività sono svolte dal **paginature**
- **paginatore** = permette di **caricare solo il minimo necessario** e funziona anche per i file mappati nella memoria.
- **Il numero massimo di frame di un processo** = dipende dall'architettura, è il numero massimo di frame cui un'istruzione può accedere.
- il **tempo di accesso** effettivo **dipende dal tasso di page fault**: un accesso ad una pagina caricata richiede pochi nanosecondi ma servono decine di millisecondi per gestire il page fault quindi deve essere un **fenomeno raro per non rallentare il sistema**
- la chiamata fork() spesso è subito seguita da una chiamata exec() che sostituisce il processo che ha eseguito fork con il nuovo programma. Quindi è bene assegnare ad entrambi lo stesso spazio di indirizzamento. Se uno dei due modifica una pagina essa sarà copiata nel suo spazio logico risparmiando memoria e velocizzando la chiamata (**copy on write**). I SO tengono un pool di pagine libere per tali richieste, il cui contenuto è azzerato prima di assegnarle ai processi.
- Molte pagine **non sono modificabili** il che permette **moltlo risparmio**
- La funzione **vfork()** funziona analogamente ma senza copy on write
- Le applicazioni in grado di gestire la memoria autonomamente hanno prestazioni scarse con la memoria virtuale. Alcuni SO lo permettono e forniscono aree di disco da usare con chiamate di I/O grezzo
- Processi **realtime** possono chiderere al SO di tenere residenti alcune pagine per evitare ritardi
- Se un processo richiede una pagina che però non ci sta per via di assenza di frame liberi, la nuova pagina si può sostituire con una già presente, trasferendo quest'ultima in area di swap assegnando nuovi frame solo se necessario. Per ridurre il tempo prima che il processo riprenda, il SO può tenere dei frame liberi in cui caricare la pagina richiesta dandola al processo mentre quella da sostituire è in attesa di I/O
- Ci sono diversi algoritmi di sostituzione della pagina:
    - **SOSTITUZIONE FIFO**
        - Selezionata la pagina piu vecchia
        - Principio che le pagine caricate tempo fa non saranno piu usate
        - Tecnica che soffre dell'anomalia di Belady: piu frame ha un processo piu alto può essere il tasso di page fault
    - **LRU**
        - si sostituisce la pagina che non si usa da piu tempo
        - Richiede grande supporto HW per non rallentare il sistema
        - Non si soffre dell'anomalia di Belady
        - Ci sono diversi modi di implementarlo:
            - Tramite timestamp per ogni entry della tabella delle pagine
            - Usando uno stack che ad ogni accesso pone in cima la pagina acceduta
            - Tramite bit di riferimento ad ogni entry della tabella delle pagine, settato dall'HW e periodicamente azzerato dal SO
            - Si tiene un byte per ogni pagina il cui MSB è il bit di riferimento. Il SO periodicamente shifta a destra il contenuto del byte. La pagina col valore piu basso è quella da sostituire.

- **ALGORITMO DELLA SECONDA POSSIBILITA**
  - le pagine sono scandite in ordine FIFO, se quella selezionata ha bit di riferimento 0 allora è quella che verrà sostituita altrimenti lo pone a zero dandole una seconda possibilità.
  - La versione migliorata controlla bit di riferimento e di modifica, la pagina migliore da sostituire è quella che ha entrambi a zero poiché riduce le chiamate IO

- Per poter suddividere m frame su n processi ci sono due tecniche principalmente:
  - Allocazione **OMOGENEA**
    - Ogni processo ottiene m/n frame
  - Allocazione **PROPORZIONALE**
    - I frame sono divisi in modo direttamente proporzionale a dimensione, priorità etc

- Se i frame allocati ad un processo sono insufficienti a contenere la sua località, questi passerà piu tempo a paginare che ad elaborare . Questo effetto è chiamato Trashing
- Il **trashing** si risolve spostando dei processi in area di swap per assegnare i loro frame ad altri.
- Non è possibile prevedere quanti frame servono ad un processo ma esistono varie approssimazioni:
  - Il modello **WorkingSet** definisce la località di un processo come le pagine accedute negli ultimi N accessi. Se il totale delle pagine dei WS è maggiore del numero di frame, il sistema si dice **in trashing**
  - Working set utile anche nella **prepagainzione**, quando si riprende in mano un processo sospeso, si carica in memoria il uso WS cosi da ridurre i page fault all'avvio
- Analizzare la frequenza dei page fault aiuta anche a prevenirlo: Se il numero di page fault sale sopra una certa soglia, si assegna al processo un nuovo frame, se scende troppo sotto una certa soglila, ne perde uno. Se non ci sono frame liberi, il processo viene sospeso.

PERIFERICHE

- il sottosistema di I/O ha il compito di
    - gestire le periferiche efficientemente
    - permettere alle applicazioni di usarle tramite chiamate di sistema
    - nascondere dettagli dell'HW alle applicazioni
- L'implementazione è stratificata:
    - LIVELLO 0 : hardware
    - LIVELLO 1 : drivers : scritti dal produttore del device, interagiscono da un lato col controller del device e dall'altro con un'interfaccia standard
    - LIVELLO 2 : Resto del sottosistema IO che esegue operazioni vche non dipendono dal dispositivo
- il **kernel** si occupa di mantenere informazioni sull'uso dei device in strutture dati simili alla tabella dei file aperti
- i device differiscono per:
    - Trasferimento a caratteri o blocchi
    - Accesso sequenziale o diretto
    - Trasferimento sincrono o asincrono
    - Condivisibile o dedicato
    - Velocità e latenza
    - Lettura-scrittura, sola lettura, sola scrittura
- Le applicazioni ovviamente non vedono questi dettagli, servono solamente al Sistema Operativo che raggruppa le periferiche in poche categorie convenzionali.
- Molti sistemi operativi permettono alle applicazioni di interagire direttamente coi driver (AKA: backdoor o escape)
- Si analizzano ora alcuni tipi di device:

1) DISPOSITIVI A BLOCCHI

- l'interfaccia di questi device si occupa di **dischi e altre periferiche a blocchi**
- comprendono i comandi **read write seek** se sono ad accesso diretto
- le applicazioni vi interagiscono attraverso il file system ma il SO e alcune applicazioni possono utilizzare alcune chiamate IO grezzo
- l'accesso a file memory mapped è spesso stratificato sopra i driver di device a blocchi e usa tecniche di memoria virtuale

2) DISPOSITIVI A CARATTERE

- L'interfaccia di questi device consente id trasferire i singoli byte
- Comprendono i comandi **GET** e **PUT**
- A livello superiore possono essere implementati diversi servizi ad esempio buffe ring per poter leggere sequenze di caratteri

3) PERIFERICHE DI RETE

- Molti SO forniscono interfacce per le reti chiamate **SOCKET**
- Permettono ad una applicazone di aprire una connessione ad un host in attesa o di attendere connessioni
- Forniscono i comandi **read** e **write** su un flusso full duplex

4) OROLOGI E TIMER

- si parla di periferiche composte da hardware che è un **temporizzatore programmabile**
- vengono fornite chiamate che permettono di ottenere ora, uptime, o di far eseguire un'operazione ad una determinata ora
- il SO tiene una lista di tutti gli eventi e programma continuamente il timer in ordine cronologico.

- Ci sono due tipi di chiamate IO:
  - **BLOCCANTI:** bloccano il processo finchè non sono completate poi lo rimettono in ready
  - **NON BLOCCANTI:** non bloccano il processo e sono utili per le interfacce grafiche.

- il sottosistema IO fornisce:

1) SCHEDULAZIONE

- Per alcune periferiche è utile schedulare la coda di richste per ridurre i tempi di attesa e migliorare le prestazioni globali

2) BUFFERING

- Un buffer è una zona di memoria temporanea con i dati in transito da device a device o da processo a device.
- Ha 3 funzioni:
  - ovviare alla differenza di velocità tra due device (ed da modem a disco)
  - adattatore tra periferiche con blocchi di dimensione diversa
  - evitare che un processo alteri i dati che deve mandare ad un device mentre procede con la sua richiesta in coda. Vengono copiati i dati in un buffer e questo sarà scritto sul device

3) CACHING

- Una cache è una memoria veloce che conserva copie di dati per evitare accessi.
- Diversa dal buffer poiché essa contiene un sottoinsieme di dati e non contiene l'unica copia del dato
- A volte buffer e cache si sovrappongono ovvero il buffer del disco può essere usato anche come cache

4) SPOOLING

- Alcuni device non possono essere usati da piu processi contemporaneamente
- Lo spooler conserva i dati per questi device in buffer su disco e li invia in ordine
- Alcuni SO permettono ai processi di prenotare l'accesso esclusivo ad un device (rischio deadlock)

5) GESTIONE ERRORI

- Il sistema operativo può far fronte ad errori momentanei ma per i guasti permanenti deve per forza comunicare al processo che la richiesta è fallita
- Ritorna quindi un codice di errore alla chiamata

- le richieste al SO devono essere trasformate in richieste hardware, per farlo avvengono una serie di operazioni:
  - il processo effettua una chiamata IO
  - il SO ne controlla la correttezza e se è consentita
  - controlla se può soddisfare subito la richiesta ed eventualmente la soddisfa
  - altrimenti manda la richiesta al driver del device ed eventualmente blocca il processo
  - il driver elabora la richiesta e da comandi al controller che esegue l'operazione ed avverte il SO con un interrupt
  - il driver viene avvertito e segnala al sottosistema IO che la richiesta è stata completata

- l'IO è un fatore **critico per le prestazioni**
- per migliorarle bisogna ridurre i cambi di contesto, il numero di copie dei dati la frequenza degli interrupt usando grandi trasferimenti, controller intelligenti, polling, spostare le primitive di gestione nell'HW cosi da eseguire IO ed elaborazioni contemporaneamente ed evitare di sovraccaricare un componente coinvolto (CPU, BUS, Memoria …) poiché questo **rallenterebbe l'intero sistema**

PROTEZIONE

- per protezione si intende la sicurezza delle risorse fisiche e logiche contro accessi non autorizzati.
- Bisogna far distinzione tra
  - **Policy** = regole
  - **Meccanismi** = strumenti che applicano le policy
- La protezione è fornita da da un meccanismo che controlla l'accesso alle risorse, che deve fornire i mezzi per stabilire le regole e applicarle. Riceve richieste dai processi **e decide se autorizzarle o meno**
- Ogni risorsa ha un identificativo e un insieme di operazioni.
- Ogni processo opera in un **dominio di protezione** che **definisce le risorse a cui può accedere e le operazioni lecite**
- L'associazione tra **processo** e **dominio** può essere:
  - **STATICA** = processo fisso in un dominio, in alcuni momenti può quindi avere piu autorizzazioni del necessario.
  - **DINAMICA** = Può cambiare dominio, applica meglio il **principio di minima conoscenza**

- Un **dominio** può essere realizzato in diversi modi: un utente, un processo o anche una procedura
- Tipicamente un processo è assegnato ad un dominio dal SO i cui permessi sono decisi dagli utenti o amministratori
- Per determinare le policy si può utilizzare una **matrice di accesso**:
  - **RIGHE** = Domini
  - **COLONNE** = Risorse
  - nella cella (i,j) sono presenti le operazioni che un processo del dominio i può fare sulla risorsa j. I domini possono essere visti anche come **risorse**: se nella cella (i,j) è presente switch allora un processo del dominio i può passare al dominio j
  - la modifica del contenuto della matrice è controllato da **3 operazioni**:
    - **COPIA**
      - Copia l'autorizzazione che un odminio ha su un oggetto ad un altro dominio, con o senza dirito di propagarlo.
    - **PROPRIETA**
      - Consente ad un processo autorizzato di agiungere o rimuovere diritti
    - **CONTROLLO**
      - Consente ad un processo di cambiare diritti di un dominio di cui il suo dominio ha il controllo
  - I diritti copia e proprietà permettono di evitare il propagarsi dei diritti di accesso ma non garantiscono il contenimento delle informazioni che è considerato un problema irrisolvibile
- La **matrice di accesso** va memorizzata in un modo efficiente, ci sono diverse implementazioni:
  1) **TABELLA GLOBALE**
     - Memorizza una lista di terne [dominio,oggetto,diritti]
     - Un processo in un dominio i è autorizzato a eseguire un'operazione m su un oggetto j solo se esiste una terna [i,j,d] con m che appartiene a d
     - Lista molto grande e non permette di raggruppare oggetti o domini

2) **ACCESS CONTROL LIST**
    - Per ogni oggetto si memorizza una lista di coppie [dominio, diritti]
    - Si possono definire diritti predefiniti
    - Memorizza informazioni globali ma è inefficiente su grandi sistemi poiché le ACL solo scandite spesso

3) **CAPABILITY LIST**
    - per ogni dominio si memorizza una lista di coppie [oggetto,diritti] dette **capability**
    - rende semplice l'accessoa informazioni su processi e memorizza informazioni locali ma la revoca è poco efficiente.
    - Questa struttura **NON è direttamente accessibile dal processo ma solo dal SO, per una questione di sicurezza**

4) **MECCANISMO LOCK-KEY**
    - È un compromesso tra ACL e CL
    - Ogni oggetto ha una lista di stringhe di bit uniche dette **lock**
    - Ogni dominio ha una lista di stringhe di bit uniche detta key
    - Un processo in un dominio può accedere ad un oggetto solo se una sua key apre uno dei lock della risorsa

- In un sistema di protezione dinamico può essere necessario **revocare dei permessi**.
- Le revoche possono essere
    - **IMMEDIATE O RITARDATE**
    - **SELETTIVE O GENERALI**
    - **PARZIALI O TOTALI**
    - **TEMPORANEE O PERMANENTI**

- Con le ACL la revoca è semplice
- Con le CL la revoca è piu complessa in quanto le informazioni sono sparse per il sistema
- Esistono varie soluzioni:
    - **RIACQUISIZIONE**
        - Le CL sono periodicamente svuotate, se un processo tenta di usare una capability cancellata tenterà di riacquisirla, se è stata revocata allora gli sarà negata

    - **PUNTATORI ALLE CAPACITA**
        - per ogni oggetto si tiene una lista di puntatori a tutte le capability ad esso associate.
        - Per cancellare una capability basta seguire il puntatore.
        - Costoso ma efficace

    - **INDIREZIONE**
        - le capacità puntano ad una cella di una tabella globale con tutte le capability, che a sua volta punta all'oggetto associato.
        - La revoca cancella il valore in questa tabella
        - Non consente la revoca selettiva

- **CHIAVI**
    - Ogni capability ha una key
    - Ogni oggetto possiede una master key in base alla quale la capacità è ricercata e verficata
    - Alla creazione di una capability le viene associata una key uguale alla master key in quel momento.
    - La revoca cambia la master key
    - Non consente la revoca selettiva

- Alcuni SO scaricano la responsabilità della protezione sul compilatore. Cio permette maggiore flessibilità rispetto all'implementazione nel SO ma meno sicurezza.

FILE SYSTEM

- il file system fornisce supporto per la memorizzazione e l'accesso di file e programmi
- i Filesystem possono essere vasti quindi vanno organizzati al meglio: i dischi sono divisi in partizioni (**volumi**)
- **volumi** = strutture a basso livello in cui sono memorizzati file e directory, può essere parte di un disco oppure occupare piu dischi
- **Directory** = file speciale con una tabella che associa nome di file e puntatore a descrittore. Permette di organizzare i file, ogni partizione ha una dir radice
- per rendere disponibile un file system ai processi, questo va montato. Si specifica nome del device e punto di mount da cui sarà accessibile e il SO controlla che esso non sia corrotto.
- In SO multiutente i file sono condivisibili ed è necessario memorizzare permessi di file/directory dei vari utenti
- Gli utenti possono essere di 3 categorie:
    - **Owner** = stabilisce i permessi
    - **Group** = utenti con cui condividere file
    - **Altri**
- il FS viene implementato a strati:
    - Livello 0 : dischi
    - Livello 1 : driver IO
    - Livello 2 : FS di base
    - Livello 3 : modulo di organizzazione dei file
    - Livello 4 : FS logico
- Servono diverse srutture dati, ogni partizione possiede:
    - un **blocco di controllo del boot** tipicamente il primo settore, con le informazioni su come avviare il SO in quella partizione
    - un **blocco di controllo della partizione** con informazioni su dimensione e numero di blocchi, informazioni sui blocchi liberi, vari dati che variano col FS FD e directory
- Inoltre il SO mantiene sempre
    - Una **tabella delle partizioni** con informazioni sui FS montati
    - Alcuni **descrittori di directory accedute di recente**
    - **Tabelle dei file aperti**
- le directory possono essere organizzate in due modi:
    - **LISTA =** scansione lineare e ausilio di eventuale cachee cancellazione tramite flag
    - **HASH TABLE =** si ricava la posizione nella tabella con una funzione di hash sul nome velocizzando la ricerca
- Il file system deve anche gestire l'allocazione dei file, essa può avvenire in diversi modi:
    - **ALLOCAZIONE CONTIGUA**
        - Ogni file occupa un certo numero di blocchi contigui su disco
        - Ogni file ha memorizzato indirizzo di inizio e lunghezza in blocchi
        - Accessi sequenziali e diretti molto rapidi ma molta frammentazione esterna
        - Se i file aumentano di dimensione può essere necessario spostarli, cosa molto costosa
        - **Extent =** si trova un'altra posizione per i blocchi nuovi del file e si memorizzano le posizioni dei frammenti. Può generare eccessiva frammentazione

- **ALLOCAZIONE COLLEGATA**
  - si memorizza per ogni file puntatore al blocco iniziale e finale
  - ogni blocco ha puntatore al blocco successivo
  - i blocchi possono trovarsi ovunque su disco
  - rapidi accessi sequenziali ma accessi diretti molto lenti
  - i puntatori occupano spazio e la perdita di un puntatore implica la perdita del file
  - variante comune: **File Allocation Table (FAT)** dove per ogni file si memorizza blocco iniziale e in una loczione speciale del disco si mantiene una tabella che per ogni blocco del disco contiene o il bloco successivo o fine file
  - unico svantaggio della FAT è che va tenuta in memoria principale perché essa sia efficiente

- **ALLOCAZIONE INDICIZZATA**
  - per ogni file si mantiene un array di puntatori ognuno dei quali punta ad un blocco del file
  - Critica la dimensione del blocco indice, troppo piccolo non permette costruzione di file grandi, troppo grande e spreca spazio per file piccoli.

- È fondamentale gestire lo spazio libero dato dai file cancellati per evitare che dello spazio venga sprecato, per farlo vi sono diverse tecniche:
  - **BITMAP**
    - Si memorizza un bit per blocco, rende semplice trovare il primo blocco libero o n blocchi liberi consecutivi
    - Deve esser mantenuta in memoria centrale per le prestazioni
    - Periodicamente aggiornata su disco

  - **LISTA COLLEGATA**
    - Ogni blocco libero contiene un puntatore al successivo e si memorizza il puntatore al primo.
    - Facile trovare primo blocco libero ma non n blocchi liberi rapidamente

  - **RAGGRUPPAMENTO**
    - Nel primo blocco libero si memorizzano gli indirizzi di n blocchi di cui n-1 sono liberi e l'ultimo contiene n puntatori
    - Facile trovare blocchi liberi rapidamente

  - **CONTEGGIO**
    - si tiene una lista con indirizzo blocco libero e quanti lo seguono

- Il montaggio permette ai processi di accedere ad un FS, si specifica il device e il punto di mount da cui sarà accessibile. Il SO controlla la coerenza del FS, si trova in stato incoerente quando perde dei dati che magari erano in cache prima che il computer perdesse l'alimentazione
- Il **controllore della coerenza** corregge errori di questo genere ma non riesce sempre a farlo, se si perde per esempio un blocco indice, non c'è nulla da fare.
- Alcuni file system per questo tengono un file di log, il montaggio controlla se c'erano operazioni in corso annullandole per evitare danni
- Altra operazione che si può effettuare è il backup, che serve ad avere una copia dello stato del file system in un determinato momento e nel caso di corruzione permanente del file system può essere usato come punto di ripristino.