M1 - UD1 - Lezione 1 - Macchina di Von Neumann: architettura e funzionamento

L'architettura generale di un calcolatore è composta da un processore in grado di eseguire operazioni prelevate dalla memoria centrale.
La coppia processore + memoria costituisce l'unità centrale di elaborazione.
Le informazioni su cui l'unità centrale può lavorare sono fornite da dispositivi di I/O, le cosiddette "periferiche" (ad es. la tastiera (input), video, stampanti (output), unità di memoria di massa (memorizzazione)).
L'architettura di VN è costituita da l'unità di elaborazione (CPU), la memoria centrale (dove vengono memorizzati i programmi da eseguire) e le interfacce verso le periferiche.
CPU e memoria sono connessi tra di loro tramite il bus di memoria, mentre le interfacce di I/O e la CPU sono connesse dal bus di I/O.
Il processore, connesso alla memoria centrale, è in grado di individuare nella memoria centrale un'istruzione dopo l'altra, grazie al program counter, un registro in grado di individuare le parole di memoria desiderate.
Questo registro viene incrementato di uno ad ogni istruzione eseguita per individuare l'istruzione immediatamente successiva a quella in esecuzione.
Questo meccanismo fornisce un supporto hardware per eseguire automaticamente sequenze di istruzioni.
L'operazione individuata viene caricata nel processore durante la fase di "fetch", e viene memorizzata nel registro CIR (current instruction register).
La CU del sistema provvede ad acquisire tale istruzione e a capire se essa è ammissibile o meno (fase di decode dell'operazione).
Se è ammissibile, la CU provvede ad attivare l'opportuna unità aritmetico-logica capace di eseguire l'operazione, e provvede anche ad attivare il trasferimento del contenuto dei registri in cui sono memorizzati gli operandi per l'operazione in modo che essa possa essere compiuta.
Il risultato sarà posto nel registro specificato dall'istruzione stessa (fase di execute).
Il processore esegue in modo ciclico (all'infinto) le fasi di fetch, decode e execute.
Per descrivere la soluzione di problemi applicativi abbiamo bisogno di diverse strutture del linguaggio:

- sequenza
- frase condizionale semplice (esecuzione di un blocco se viene rispettata una condizione)
- frase condizionale doppia (esecuzione di un blocco se viene rispettata una condizione, esecuzione di un altro se non viene rispettata)
- ciclo a condizione iniziale (viene valutata la condizione, se la condizione è vera si esegue un blocco di operazioni e al termine si ritorna valutare la condizione, se la condizione è falsa il blocco non viene eseguito e viene eseguita l'istruzione successiva al ciclo.
- ciclo a condizione finale
- ciclo a conteggio

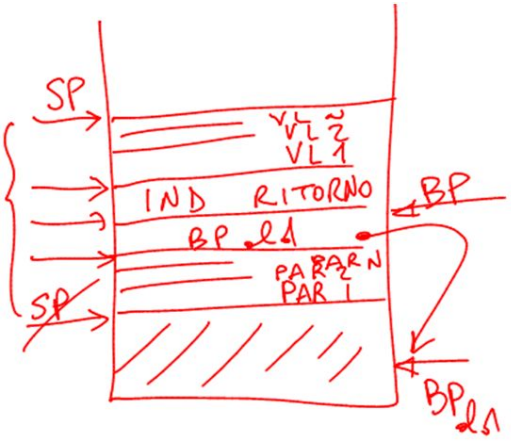
Sorge la necessità di implementare costrutti non sequenziali.
Per esempio, la frase condizionale doppia rende necessario non più solamente eseguire sequenze di operazioni, ma essere in grado di saltare da una parte all'altra della memoria.
Dunque, a livello di memoria, saranno eseguite tutte le operazioni a partire dalla prima istruzione della valutazione (if) fino alla fine della stessa, e al termine di essa saremo in grado di valutare il risultato della condizione, e decidere se eseguire un blocco A di istruzioni piuttosto che un altro B.
Quindi si applicherà una funzione di Jump ad un valore vero o falso: se la condizione è rispettata viene sostituito nel PC l'indirizzo della prima istruzione del blocco B da eseguire (saltando così tutte le operazioni del blocco A da non eseguire) , altrimenti viene eseguita direttamente

l'istruzione dopo fino alla fine del blocco A, e al termine ci sarà un'istruzione di Jump incondizionato che salterà la parte di operazioni del blocco B.
Questo realizza la semantica completa della frase condizionale doppia, linearizzando in memoria centrale una struttura non lineare, e consentendo al processore (in grado di eseguire solo strutture lineari) di eseguire strutture non lineari.
Tutto ciò permette di descrivere una computazione **sincrona** in modo che ad una istruzione succeda un'altra in un modo predeterminato.
Esistono però delle situazioni (ad es. con gli eventi esterni al calcolatore), in cui non è possibile predeterminare l'ordine in cui eseguire le operazioni (es. gestione di ingressi e uscite, o di segnali in un sistema di controllo).
Questo vuol dire che abbiamo bisogno di gestire eventi asincroni con l'evoluzione della computazione.
L'interruzione è il meccanismo hardware che permette di accorgersi di eventi esterni.
Quindi mentre il processore sta eseguendo fetch, decode ed execute delle operazioni caricate in memoria, potrebbe arrivare un'interruzione dal mondo esterno.
Nel momento in cui arriva l'interruzione, si sospende l'esecuzione del programma e si va ad eseguire una sequenza di operazioni prevista per trattare l'evento esterno.
Al termine di questa sequenza si ritorna ad eseguire la prima istruzione successiva al verificarsi dell'evento, in cui si era avuta la sospensione di esecuzione del programma.
La sequenza di operazioni eseguita prende il nome di **Risposta all'interruzione**.
Per accorgersi di un'interruzione, durante il suo ciclo di esecuzione di fetch, decode ed execute, prima di eseguire una fetch il processore va a sentire se è arrivata un'interruzione dal mondo esterno, se è arrivata sospende l'esecuzione del programma e attiva la risposta all'interruzione. Al termine si torna ad eseguire l'istruzione individuata dal PC.
Questo permette di trattare eventi esterni in modo asincrono con l'evoluzione della computazione del programma.

M1 - UD1 - Lezione 2 - Chiamate di procedura e risposta alle interruzioni

In un linguaggio ad alto livello una chiamata di procedura permette di incapsulare sequenze di operazioni con un nome simbolico, in modo da poterla chiamare all'interno del programma.
La procedura può avere un insieme di parametri in modo da operare su insiemi diversi di valori di tali parametri. Per supportare l'esecuzione della chiamata di procedura è necessario creare delle strutture dati adeguate in mem. centrale per seguire la sequenza di attivazione delle chiamate, così come previsto dai linguaggi di programmazione. Questa struttura, e questo modo di gestire le chiamate, ha il nome di Stack (o Pila).
Lo stack è una porzione di memoria centrale in cui viene individuata una base a partire dalla quale vengono accumulate le informazioni necessarie a supportare le varie chiamate.
Quando si vuole richiamare una procedura sullo stack a partire dalla base vengono caricate le informazioni necessarie.
La cima dello stack è individuata nel registro Stack Pointer, in modo da supportare efficacemente, all'interno del processore , l'accesso alla memoria stessa, gestita secondo tale modalità.
Lo stack è una struttura dati in cui le informazioni vengono inserite ed estratte secondo la modalità LIFO. Questo consente di aprire una procedura, eseguire delle operazioni nella procedura, e quindi richiamare un'altra procedura all'interno della procedura chiamata senza interrompere la procedura chiamante.
Nella chiamata di procedura il compilatore traduce la chiamata ad alto livello in una sequenza di operazioni che gestiscono la creazione dell'ambiente in cui dovrà operare la procedura chiamata, ed ad attivare tale procedura.
La cima dello stack individua quindi l'ultimo elemento di tale ambiente.

Il riferimento all'ambiente di lavoro viene individuato tramite il contenuto di un registro di base che punterà ad una posizione opportuna all'interno di questa porzione di memoria riservata alla procedura chiamante.
Si indica col valore vecchio del Base Pointer quello della procedura chiamante.
Quando la procedura chiamante decide di effettuare la chiamata, disporrà sufficiente spazio sufficiente sulla cima dello stack a conservare i valori attuali dei parametri formali, ovvero di quei parametri su cui la procedura chiamata dovrà operare, e la cima dello stack cresce.
Per conservare il riferimento al vecchio ambiente di lavoro in cui operava la procedura chiamante, sulla cima dello stack viene deposto il valore del registro base pointer vecchio, e la nuova cima dello stack diventerà il nuovo valore del base pointer, costituendo il riferimento rispetto al quale verranno valutate le posizioni dei valori attuali dei parametri formali e la posizione delle variabili locali.
Dunque, la procedura richiamata viene attivata, con l'esecuzione dell'istruzione "CALL". Questa operazione mette sulla cima dello stack l'indirizzo di ritorno della procedura chiamante, cioè alla prima istruzione immediatamente successiva all'indirizzo di chiamata. Mette poi nel PC il valore della prima istruzione della procedura chiamata. In questo modo il PC forza il processore a saltare all'esecuzione della prima istruzione della procedura chiamata. Tale prima istruzione completa la creazione dell'ambiente di lavoro riservando lo spazio per le variabili locali sulla cima dello stack. Quest'ultima diventerà dunque la nuova cima dello stack



contesto: l'insieme delle info così memorizzati (valori attuali dei parametri formali, il riferimento alla vecchia base, l'indirizzo di ritorno e il valore delle variabili locali).

La procedura chiamata così procede nelle sue operazioni, andando ad utilizzare le informazioni poste sullo stack nel contesto di attivazione relativo, in particolare i valori attuali dei par. formali, il riferimento al base pointer, il riferimento al nuovo base pointer per calcolare le posizioni dei valori attuali dei par. formali (partendo dal base pointer e andando verso indirizzi crescenti di memoria), l'indirizzo di ritorno quando servirà, e le variabili locali durante le varie operazioni. La pos. di quest'ultime sarà calcolata usando sempre come riferimento il base pointer del contesto corrente.

Eseguite le operazioni della procedura, bisogna terminare la chiamata.
Si rimuovono le variabili locali, si mette in mostra l'ind. di ritorno, e quindi l'esecuzione dell'istruzione END andrà a leggere il valore dalla cima dello stack e lo ripristinerà nel program counter, tornando ad eseguire la prima istruzione successiva alla chiamata di procedura nella procedura chiamante.
Questa istruzione ripristina il contesto di attivazione della procedura chiamante, scaricando nel base pointer il vecchio valore del base pointer.
I valori attuali dei par. formali non servono più, quindi vengono rimossi, tornando così alla posizione dello stack pointer prima della chiamata.

Ritorno di un valore
Possono essere usati i registri (se sufficiente), oppure si può usare una opportuna zona di memoria alla cima dello stack, ad esempio prima del valore dei par. formali verrà lasciato uno spazio per il valore ritornato dalla procedura chiamata, e quindi si riempirà lo stack secondo lo schema.
Il risultato verrà quindi deposto nella zona di memoria preposta.
Allo scarico delle informazioni per il ritorno all'esecuzione della procedura chiamante, il valore ritornato occuperà il primo spazio al di sopra dell'ambiente di attivazione della procedura chiamante, la quale saprà recuperare il valore e porlo nelle proprie variabili.

Risposta all'interruzione e chiamata a procedura
Le due cose differiscono principalmente per la loro **attivazione**.
La risposta all'interruzione è asincrona con l'evoluzione della computazione del programma, mentre la chiamata di procedura è sincrona (la quale viene attivata tutte le volte che viene chiamata in maniera sincrona).
Un'altra differenza sta nel **salvataggio del contesto** di attivazione della procedura in esecuzione per poter creare un contesto per la risposta all'interruzione.
Si deve quindi tener conto che nella risposta di interruzione non vi è un passaggio di parametri, in quanto non esistono parametri da passare ad un'interruzione: si tratta solo di attivare l'interruzione con le informazioni che verranno fornite direttamente dall'hardware per identificare di quale interruzione si tratta.

Quando capita un'interruzione, nello stack avviene che, sul contesto del programma in esecuzione, si andrà a creare il contesto per l'esecuzione della risposta all'interruzione. Verrà posto semplicemente l'indirizzo di ritorno, l'insieme dei flag che definiscono il modo di funzionamento del processore, e quindi si dovrà provvedere a salvare il contesto del programma in esecuzione, e quindi tutti i registri del processore, e quindi si dovranno creare le variabili locali necessarie all'esecuzione della risposta all'esecuzione.
Infine lo stack pointer arriverà in cima allo stack.
Quando la procedura terminerà si dovrà ripristinare lo stato precedente operando al contrario: si rilascerà lo spazio delle var. locali, ripristino del contesto della procedura chiamante (cioè l'insieme di tutti i registri, scaricandoli nei registri del processore), e termina eseguendo l'istruzione di ritorno da interruzione che ripristina i flag e riporta il registro Stack Pointer al livello iniziale.

M1 - UD1 - Lezione 3 - Memoria
All'interno del processore abbiamo un'insieme di elementi di memoria particolarmente veloci chiamati registri,.
Il loro numero e la loro dimensione dipende dallo specifico tipo di processore.
L'accesso a questo tipo di memoria è diretto, ovvero il processore puiòo individuare il registro che contiene l'istruzione desiderata ed accedere ad esso, il problema è che questi elementi hanno una capacità ridotta di contenere informazioni.

Il processore, oltre ad accedere alla memoria centrale, ha la possibilità di accedere ad una memoria più veloce di quest'ultima, chiamata memoria cache.
Questa è una parte del dispositivo fisico del processore, e contiene una copia di una porzione di memoria centrale.
Quando il processore vuole accedere a questa porzione di memoria centrale, invece che utilizzare il tempo necessario ad accedere utilizzando il bus di memoria per accedere alla MC, può accedere direttamente alla copia di questa porzione presente sul dispositivo del processore stesso, nella memoria cache interna al processore.
Questa memoria è molto più veloce della memoria centrale.

I suoi tempi di accesso sono paragonabili ai tempi di esecuzione di operazioni all'interno del processore.
Esiste la possibilità di avere un'ulteriore integrato di memoria cache esterna al processore (più grossa di quella interna), in modo che quando serve si possono copiare le porzioni di memoria centrale all'interno di questa cache esterna, e in un secondo tempo, quando servono alcune porzioni di quest'ultima, queste possono essere copiate nella cache interna.
Il tempo di accesso alla cache esterna è leggermente maggiore rispetto a quello della cache interna, ma è comunque superiore al tempo di accesso alla memoria centrale.

Tutti i programmi e i dati devono essere residenti nella memoria centrale per essere eseguiti.
Questa ha un accesso rapido anche se più lento di cache e registri.
Ha un accesso diretto, cioè il processore può individuare la parola di memoria desiderata e leggere direttamente e solo quella.

La capacità è limitata, anche se considerevolmente più grande della cache.
Un'altra tipologia di memoria è quella di massa. Queste sono memorie più grandi con un accesso più lento (per via dei tempi elettronici e meccanici).
Un esempio sono i dischi magnetici, ossia dei dischi metallici su cui è spalmato del materiale ferromagnetico, in cui la direzione del campo viene polarizzata dal campo elettromagnetico generato dalla testina di scrittura, e la polarizzazione verrà riletta dalla testina di lettura, permettendo così di memorizzare zeri e uni rappresentati dalle polarizzazioni in una direzione o nella direzione opposta del materiale ferromagnetico.
Posizionando le testine sui piatti è possibile scrivere su più tracce l'info desiderata (aumentando la quantità di informazioni memorizzabile su ciascun piatto).
Il tempo di accesso è almeno uno o due ordini superiore a quello della memoria centrale, in quanto sono coinvolti dei tempi elettromagnetici e elettromeccanici di operazione.
In particolare i dischi devono essere ruotati per essere posizionati nella posizione corrispondente all'info da scrivere o leggere.
La capacità dei dischi è enormemente superiore alle memorie centrali.
L'accesso può essere diretto (è possibile andare a scegliere il blocco di dati da leggere o scrivere) o sequenziale (leggendo le info memorizzate dopo l'altra)

Un'altra memoria di massa sono i dischi ottici (CD-ROM, CD-ROM scrivibili)
L'accesso è più lento del disco magnetico, e può anche questo essere diretto o sequenziale.
Vi sono poi i nastri magnetici, nastri di materiale plastico su cui è spalmato il materiale ferromagnetico. Hanno accesso molto lento e strettamente sequenziale. Non è possibile scegliere quindi l'elemento desiderato e leggere solo quello.
D'altra parte hanno una capacità estremamente ampia e quindi sono estremamente utili come dispositivi di backup per salvare le info contenute nel sistema.

Queste memorie ci permettono di costruire un supporto per contenere informazioni e programmi su cui deve lavorare un calcolatore.
Il sistema dovrà provvedere a portare dalle memorie più lente (ma più capaci) alla memoria centrale, l'unica dove è necessario dove devono essere residenti le informazioni per essere elaborate dal calcolatore, ed eventualmente essere portate dalla MC ai registri.

La tecnica di Caching consiste nel copiare porzioni di memoria centrale verso dispositivi più rapidi (ad esempio verso cache di primo e secondo livello).
Lo stesso approccio si può usare per accelerare l'accesso a dispositivi di memoria più lenti (ad esempio scaricando in RAM le info su blocchi posti nel disco, qualora queste siano informazioni a

cui si accede frequentemente (evitando che il processore debba continuamente chiedere continuamente di leggere info dal disco, attendendo il riposizionamento della testina ecc.).
Avendole scaricate e conservate in cache, in memoria centrale oppure in una cache a bordo del disco, si può evitare di attendere questo tempo di tipo elettromeccanico, limitando l'attesa al trasferimento verso la memoria centrale, nelle variabili in cui il programma opera.
Avere un meccanismo di caching significa gestire una coerenza tra le copie della memoria nella cache e le copie di riferimento poste nel livello gerarchico superiore. (ad esempio il contenuto della cache del processore deve sempre coincidere con il contenuto della memoria centrale. Se viene effettuata una modifica ad un valore nella cache, questa deve essere riportata nella memoria centrale in modo da mantenere il valore di riferimento anche se cambia il contenuto nella cache).

La protezione della memoria è importante quando abbiamo un ambiente multiprogrammato (con più programmi caricati in MC).
In questo caso è necessario confinare l'accesso dei vari programmi alla memoria, facendo si che ogni programma possa accedere solamente allo spazio di memoria ad esso riservato, in modo da non alterare il contenuto di memoria degli altri programmi, influenzandone il comportamento e il risultato di computazione.
La protezione dei registri è implicita nel cambio del programma in esecuzione, in quanto al cambio noi dovremo salvare il contenuto dei registri.
Il contenuto della cache è automaticamente aggiornato e salvato dai meccanismi hardware che provvedono alla sua gestione (non è quindi preoccupazione nostra).
Per la memoria centrale è necessario curare che le porzioni di memoria centrale usate da un programma non vengano utilizzate da altri programmi (viene introdotto così un dispositivo dedicato chiamato Memory Management Unit (MMU)).
Per la memoria di massa, la protezione delle info viene realizzata nella gestione del file system.

Quando il processore vuole accedere ad una porzione di memoria centrale, emette un indirizzo desiderato all'interno di questa porzione.
Il MMU permette di verificare che non si effettuino operazioni di lettura o scrittura al di fuori della porzione riservata al programma.
In questo dispositivo viene posto un registro di base il cui valore, quando parte il programma individuato dal sistema, viene posto uguale al primo indirizzo occupato dal programma.
Esiste poi un secondo registro, chiamato "Limite", che individua l'ultimo elemento in MC occupato dal programma.
Quando presentiamo un indirizzo, invece che presentarlo alla memoria centrale, questo viene fornito alla MMU, la quale verifica che questo sia compreso tra il valore iniziale e il valore limite. Se è così, viene consentito l'accesso alla parola desiderata e il valore viene tornato al processore.
Se non fosse così, la MMU lancia una interruzione al processore per segnalare l'errore.
La risposta all'interruzione dovrà comprendere quale problema si è verificato e segnalare l'errore di accesso ad una zona di memoria non consentita.

M1 - UD1 - Lezione 4 - Connessione delle periferiche

In memoria centrale vengono deposti dei dati da elaborare.
Per poter deporre questi dati ed estrarre i risultati da fornire agli utenti è indispensabile usare le periferiche, così come è indispensabile usarle per memorizzare dei risultati (e i programmi) per elaborazioni successive.
Per interfacciare le periferiche esiste un insieme di schede opportune all'interno del calcolatore (le interfacce delle periferiche), connesse all'unità centrale da un opportuno bus di I/O.
L'interfaccia di elaborazione è costituita da due parti: una parte che si interfaccia al calcolatore e una che si interfaccia alle periferiche.

La parte che si interfaccia alle periferiche è connessa ad un opportuno cavo che termina sull'elettronica di controllo a bordo della periferica.
Questa elettronica di controllo sarà composta da una parte di gestione della connessione, e da una parte di controllo della periferica vera e propria. Quet'ultima parte è quella che si occupa della movimentazione delle componenti della periferica (ad esempio per il disco si occupera della rotazione del disco ecc.)
L'insieme dell'elettronica di controllo e degli eventuali dispositivi elettromeccanici costituisce la periferica nel suo complesso.
L'elettronica di gestione della connessione (sia a bordo della periferica, sia a bordo della scheda di interfaccia nella unità di elaborazione) costituisce (assieme al cavo di connessione) quello noto come "canale di comunicazione". È quello attraverso vengono fatte fluire le info dal sistema di elaborazione alla periferica e viceversa.

Per gestire tale canale, nell'unità centrale, vi è un insieme di dispositivi appositi connessi tramite il bus di I/O, che costituiscono l'interfaccia della periferica. Questa contiene un insieme di registri che permette il colloquio con la periferica, l'invio di informazioni e il ritorno di risultati dell'elaborazione effettuata dalla periferica verso il calcolatore.
In particolare, c'è il registro di Stato, il registro di Comando, e il registro Dati.
L'unità centrale chiede all'interfaccia di verificare qual'è lo stato della comunicazione con la periferica, e ottiene quindi il contenuto del registro di stato.
Se la connessione è Ok, la periferica è libera, e il processore è in grado di richiedere un'operazione I O.
Farà ciò andando a deporre nel registro **dati** l'ordine che si vuol far pervenire alla periferica.
L'unità centrale andrà poi a scrivere nel registro di comando l'ordine per l'interfaccia di inviare le info richieste alla periferica.
L'elettronica nell'interfaccia comprenderà quanto specificato nel comando, così da comprendere che è necessario inviare il contenuto del registro dati alla periferica. Questo viene effettuato trasferendo il contenuto del registro dati all'elettronica nell'interfaccia, che controlla la connessione con la periferica. Questa elettronica provvederà a colloquiare con la sua controparte nella periferica e a trasferire l'ordine contenuto nel registro dati nella periferica.
I dati quindi così possono fluire dal processore, attraverso l'interfaccia fino alla periferica
Qui vengono interpretati, eseguiti, ed il risultato verrà tornato eventualmente all'interno del registro dati.
A operazione completata verrà aggiornato il registro di stato dell'interfaccia, in modo che il processore, interrogando questo registro, possa sapere il risultato della computazione effettuata.

Controllo dell'interfaccia da parte della CPU
Esistono due modi:

- Attesa attiva: il processore, tramite il bus di I O, colloquia con l'interfaccia della periferica, e va a leggere qual'è il contenuto del registro di stato in maniera ciclica, finché non scopre che lo stato è cambiato e che l'operazione ha avuto esito (buono o cattivo che sia). A questo punto l'operazione prosegue nelle operazioni corrispondenti al risultato dell'operazione di I O effettuata
- Interruzione: questo evita l'utilizzo di istruzioni non utili (come le istruzioni di attesa riportate sopra). La CPU invia il comando e non sta in attesa attiva, ma procede nello svolgere le proprie attività, eventualmente cambiando il programma in esecuzione. Quando la parte di gestione del canale di comunicazione nell'interfaccia provvede a modificare il registro stato, segnando il risultato dell'interruzione, questo flag porta alla generazione di una interruzione verso il processore, in modo che questo si accorga che l'operazione è terminata, e possa procedere ad acquisirne lo stato. In questo caso tale acquisizione e analisi, e l'attivazione

delle operazioni conseguenti, viene effettuato all'interno della risposta all'interruzione associata a quella specifica interfaccia

Trasferimento dei dati

Il trasferimento dei dati tra CPU e interfaccia può avvenire in diversi modi:

- In *singola parola*, come appena visto, attraverso il registro Dati.
  Ciò non è efficiente quando si devono trasferire grossi blocchi di dati
- *Direct Memory Access*: consente di essere più efficiente nel trasferimento dati, in quanto non si provvede più di trasferire una parola alla volta ma blocchi di parole, consentendo alla CPU di procedere direttamente al trasferimento, un byte dopo l'altro, fino al completamento del trasferimento
- Un altro modo per velocizzare le operazioni di trasferimento dei dati consiste nel considerare la *periferica mappata in memoria*.
  In questo caso abbiamo che una porzione di memoria centrale viene mappata, ossia quando l'unità centrale chiede di accedere ad un indirizzo di un blocco di memoria centrale, non risponde quest'ultima, ma risponde l'interfaccia, fornendo l'accesso all'info corrispondente a quell'indirizzo.
  Questo blocco di memoria centrale è sostituito quindi dalla memoria presente sull'interfaccia (dal punto di vista operativo).La porzione di controllo del canale verso la periferica, provvederà a caricare in quella porzione di memoria mappata i dati che devono essere trasferiti al processore provenienti dalla periferica.Il processore quando vorrà inviare dati in blocco alla periferica, li scriverà in questa memoria.Sarà quindi l'interfaccia che provvederà a trasferire direttamente alla periferica il blocco di dati, senza aver necessità di scrivere prima in MC, e poi col DMA trasferire i dati nell'interfaccia, e ancora dall'interfaccia alla periferica.

Gestione del canale di comunicazione

Il canale di comunicazione può essere gestito da due aspetti:

- Controllo nell'interfaccia dell'unità centrale
- Controllo nella periferica

All'interno dell'unità centrale abbiamo l'interfaccia con una porzione dedicata a gestire il canale. Analogamente, nella periferica c'è una porzione di elettronica destinata a gestire il canale di comunicazione.
La porzione all'interno del processore è quella che deve provvedere ad inviare dati e comandi alla periferica, e ad ottenere il risultato delle richieste desiderate.

Gestione della periferica

All'interno della periferica si ha

- una parte di elettronica che provvede ad acquisire i dati in arrivo dal processore
- una parte in grado di analizzare, comprendere e mandare in esecuzione tali richieste sulla periferica, eseguendo le operazioni desiderate dal programma, e tornando i valori relativi al completamento con successo o meno della richiesta.

M1 - UD1 - Lezione 5 - Reti informatiche: architetture e funzionamento

Reti locali e reti geografiche

Esistono due topologie fondamentali per le reti informatiche: le reti locali (tipicamente usate all'interno di un edificio o tra edifici molto vicini) e le reti geografiche (permettono di interconnerre siti geograficamente distanti).
Nelle reti locali, all'interno dell'edificio, viene realizzata una connessione con opportuni cablaggi per mettere in comunicazione stazioni di lavoro e sistemi in grado di erogare servizi condivisi (ad

esempio capacità computazionale per le applicazioni, o depositi di memoria di massa per contenere le info, o ancora dispositivi di uscita come stampanti di uso comune tra gli utenti).
Le reti geografiche sono costituite da un insieme di siti nei quali vengono installati più elaboratori interconnessi all'unica rete. Questi siti sono connessi tra loro da opportuni dispositivi di collegamento che realizzano il supporto fisico di trasmissione delle informazioni e di accesso ai vari calcolatori della rete.
Per realizzare le strutture di rete esistono varie topologie:

- la più complessa è la rete completamente connessa, in cui ogni calcolatore ha un collegamento diretto verso tutti gli altri calcolatori della rete. È costoso e difficilmente realizzabile, a causa della realizzazione di tutte le interconnessioni
- le reti parzialmente connesse sono strutture più efficienti, che consentono di ottenere un certo grado di ridondanza nelle interconnessioni (e quindi una capacità di sopravvivere a guasti sulle linee di interconnessione) pur riducendo i costi. In questo caso se una macchina desidera parlare con un'altra, può capitare che debba utilizzare una terza macchina come passaggio obbligato.
- Un'altra struttura è quella a stella, in cui un centro stella si preoccupa di realizzare l'interconnessione tra tutte le macchine della rete
- Un'altra struttura spesso usata è la struttura ad anello, in cui le macchine sono connesse una all'altra formando un anello
- Un'altra versione spesso utilizzata è quella lineare, in cui le macchine sono connesse l'una con l'altra formando una catena aperta

Tecnologie e standard per le reti locali
Per le reti a bus vengono utilizzate le varie le tecnologie ethernet, nelle sue varie versioni che supportano diverse velocità.
Per le reti ad anello esistono delle strutture a Token Ring, in cui le connessioni sono realizzate con cavi di rame, oppure si utilizza FDDI in fibra ottica.
Le reti a stella vengono realizzate tipicamente usando un cablaggio strutturato.
Esistono poi le reti wireless, che permettono una facile connettività tra calcolatori portatili. Per questa tipologia si usano le tecnologie bluetooth e wifi.

Interfaccia nell'unità centrale
Se si guardano dispositivi e tecniche per la comunicazione, a livello hardware e software di base, si capisce che viene ripercorsa esattamente la struttura per connettere una qualunque periferica.
In particolare si ha la CPU che viene connessa alla memoria centrale attraverso il bus di memoria, e abbiamo quindi l'interfaccia di rete esattamente strutturata come una qualunque interfaccia delle periferiche, connesse attraverso il bus di I/O alla CPU, con un registro di stato, un registro di comando e una porzione di memoria per effettuare il trasferimento di dati da e verso la rete, secondo un meccanismo di periferica mappata in memoria.
In questo caso, questa zona di buffering dove vengono depositati i dati da inviare o ricevuti dalla rete, vengono visti dalla CPU come deposti direttamente a un indirizzo di memoria centrale, per rendere più veloce la creazione di questi blocchi di dati o la loro analisi e lettura, e il loro uso all'interno dell'applicazione.
Sull'interfaccia di rete esiste la parte di controllo del canale di comunicazione, che provvede a gestire la connessione alla rete, attraverso il cavo di rete.
La struttura è quindi esattamente uguale a quella delle altre periferiche.

Rete come periferica complessa
La rete può essere vista, dal punto di vista logico, solo come una periferica molto complessa, infatti dal punto di vista del calcolatore la connessione alla rete appare come un cavo che esce dal

sistema e va a connettersi alla struttura di interconnessione complessa della rete. A questa struttura staranno appese le altre macchine.
Il canale di comunicazione, in questo caso, è costituito dalla porzione della scheda di interfaccia di rete nel calcolatore che provvede al governo del canale, da tutta la rete, e dai canali di comunicazione verso gli altri calcolatori.
Dal punto di vista astratto, le periferiche sono gli altri calcolatori.
Il nostro calcolatore può inviare richieste di operazioni o di dati agli altri calcolatori attraverso il canale, e riceve le risposte relative, esattamente come avveniva nelle normali periferiche.

M1 - UD1 - Lezione 6 - Classificazione dei sistemi di elaborazione

Mainframe

Il più vecchio sistema di elaborazione è il mainframe: ossia un'unità centrale alla quale vengono connessi dispositivi di ingresso e uscita.
Tra i dispositivi di ingresso vi sono le schede perforate, sulle quali perforazioni o non perforazioni indicavano zeri o uni.
Come dispositivi di output venivano usate le stampanti.
Successivamente, per semplificare l'immissione di grosse quantità di dati (ormai poco maneggevoli con le schede) e estrarre grosse quantità di dati da maneggiare e conservare, vengono introdotti i nastri magnetici, sia come dispositivi di ingresso, che come dispositivi di uscita.
Queste sono architetture tipiche di sistemi orientati all'elaborazione di lavori di tipo non interattivo, specificamente l'uso era quello di elaborazione a lotti.
Questo approccio era tipico dei sistemi monoprogrammati, cioè con un programma solo caricato in memoria centrale in qualsiasi momento.
Chiaramente l'unità centrale era sottoutilizzata, in quanto doveva attendere i tempi elettromeccanici dei dispositivi di ingresso e uscita, senza poter procedere nelle operazioni. Ciò faceva lievitare i costi di utilizzo, in quanto il tempo effettivamente utile dedicato alla computazione del programma era ridotta.

Sistemi multiprogrammati

Una soluzione a questo problema fu l'avvento dei sistemi multiprogrammati, in cui venivano caricati in memoria centrale più programmi contemporaneamente, riducendo i tempi di attesa tra un programma e l'altro.
Questo però non risolveva il problema di attesa delle periferiche per fornire o ricevere informazioni (doveva essere comunque atteso il tempo di completamento dalle periferiche, senza poter svolgere altro lavoro utile).
Una soluzione a quest'ultimo problema fu l'avvento del multiprocessing, ossia l'esecuzione di più programmi contemporaneamente.
Quando un programma lanciava l'esecuzione di un'operazione di ingresso o uscita, il sistema procedeva ad eseguirla e non attendeva la risposta proveniente dalle periferiche, ma eseguiva altri programmi, evitando di lasciare tempi morti nell'esecuzione.

Terminali

Un'evoluzione dell'uso dei Mainframe è stata quella di permettere l'interazione di molti utenti operanti contemporaneamente.
Questi sistemi interattivi nascono quando all'unità centrale vengono collegati dispositivi che consentono l'interazione con gli utenti: i terminali.
Questi dispositivi composti da tastiera e video, permettono di fornire dati e comandi, e di vedere immediatamente i risultati dell'elaborazione da parte dell'utente.
Ancora venivano usate le stampanti, e a seguire dischi per conservare info a lungo termine.

In questo caso l'elaborazione prevede l'elaborazione contemporanea di flussi di attività diversi, chiamati processi. In questo caso si ha un sistema interattivo multiinterattivo con più programmi caricati in memoria centrale, un uso condiviso del processore con tecniche di multiprocessing, in particolare per dare all'utente l'impressione dell'evoluzione contemporanea (e non a singhiozzo a causa dell'utilizzo da parte di altri utenti) viene utilizzato un meccanismo di condivisione omogenea del tempo, basata sul meccanismo di time sharing.

Minicomputer
L'evoluzione della tecnologia ha permesso di ridurre la complessità di questi sistemi "mainframe" di elaborazione per realizzare architetture dedicate a gruppi più piccoli di utenti (minicomputer). Questi sistemi consentono una forte interazione con gli utenti e ancora la presenza di più utenti contemporaneamente nel sistema.

Workstation
Sempre l'evoluzione della tecnologia ha permesso di ridurre la scala dei minicomputer, sviluppando delle workstation che permettono di portare il sistema alla portata del singolo utente. Da un lato queste workstation hanno esigenze più sofisticate, ma permettono di ridurre il costi di uso del sistema interattivo.
Questi sistemi di tipo desktop hanno caratteristiche e dispositivi di interazione avanzata e forniscono una capacità di elaborazione molto elevata. Ancora, c'è una forte interazione in un ambiente multi processo in tempo reale

PC
L'evoluzione delle tecnologie di sviluppo dei terminali ha portato allo sviluppo di terminali interattivi sempre più avanzati, per supportare un'interazione evoluta con sistemi centrali e piccole attività di elaborazione locale.
Questi PC sono diventati sistemi autonomi, con capacità di grafica e dispositivi di interazione avanzata.

Sistemi mobili
Con l'avanzare della tecnologia, le dimensioni si sono ridotte sempre di più, fino ai computer portatili, i sistemi palmari e i telefono cellulari.
In questi sistemi si ha ancora una forte interazione, sono sistemi interattivi multiprocesso che però hanno un ridotto consumo di potenza (e tipicamente un numero limitato di processi).

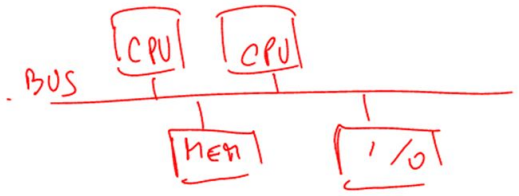
Sistemi real-time
I sistemi di elaborazione in tempo reale sono utilizzati per il controllo e l'automazione industriale, ma anche per i sistemi di automazione della casa, sistemi biomedicale ecc.
Obiettivo di questi sistemi è quello di rispondere in tempi brevi agli eventi in attivo dal mondo esterno, in tempo reale stretto (hard real-time) o lasco (soft real-time).
L'architettura hardware deve essere in grado di scambiare segnali con il mondo esterno per poter supportare queste tipologie di interazione, e il sistema operativo deve adeguarsi per fare in modo che gli eventi esterni siano sentiti e gestiti dal sistema nel tempo richiesto dall'applicazione

Sistemi dedicati
Sono stati anche sviluppati sistemi specifici per una sola applicazione, riducendo fortemente l'architettura generale per selezionare i componenti strettamente indispensabili all'applicazione considerata: i sistemi dedicati (es. motori di automobili, elettrodomestici).
Questi hanno di solito caratteristiche limitate in termini computazionali, di memoria e di periferiche.

Sistemi multimediali

Spesso gli utenti hanno cercato di avere forti interazioni con modalità di tipo diverso: nascono i sistemi multimediali, che hanno delle loro caratteristiche specifiche in termini di architettura, proprio per migliorare la possibilità di interazione attraverso vari media (canali di comunicazione con l'utente) in tempo reale.

Sistemi multiprocessore
Per supportare la necessità computazionale di alcuni ambienti applicativi sono stati sviluppati dei sistemi multiprocessore, in modo da avere forti capacità di elaborazione e effettuare economia di scala sulle periferiche.
In questi sistemi abbiamo un'architettura basata su un bus di connessione, con varie unità di elaborazione connesse al sistema, con una memoria condivisa dai vari sistemi di calcolo, e un insieme di interfacce comuni per la gestione dell'I/O.



I processori possono eseguire tutti la stessa tipologia di attività, e quindi essere sistemi **simmetrici**. Oppure possono essere specializzati: alcuni per eseguire I/O, altri per il sistema operativo e così via.. in questo caso si parla di sistema multiprocessore **asimmetrico**.
Anche in questo caso si supportano più utenti in modo interattivo.

Sistemi cluster
L'architettura a cluster è un'architettura in cui le interconnessioni con il sistema sono più lasche, ossia abbiamo un sistema di bus che collega più calcolatori tra di loro in modo stretto.
Sicuramente non però così stretto come nel multiprocessore, nel quale i processori connessi tra di loro da un solo bus di sistema.
Questo serve a migliorare ulteriormente le economie di scala sulle periferiche, e l'affidabilità del sistema in caso di guasto. Ma soprattutto, l'obiettivo è quello di essere realizzato con componenti di facile reperibilità sul mercato, e quindi riducendo i costi di acquisizione.

Sistemi distribuiti
Il sistema distribuito è il sistema in cui connettiamo più lascamente dei calcolatori in modo da poter condividere risorse in modo più lasco attraverso una rete informatica.
Anche in questo caso si possono supportare sistemi interattivi con più utenti.

M2 - UD1 - Lezione 1 - Funzioni di un sistema operativo

Obiettivi del sistema operativo
Gli obiettivi del sistema operativo sono due:

- Astrazione: creare un alto livello di astrazione dei componenti della macchina su cui il sistema opera. L'innalzamento del livello di astrazione ha come obiettivo quello di rendere più semplice l'utilizzo e l'accesso alle risorse, e mostrare ai programmi e agli utenti un aspetto più semplice di interazione, e inoltre di fornire una gestione ottimizzata delle risorse rendendole astratte, e nascondendo i dettagli di gestione in questa astrazione.

- Virtualizzazione delle risorse: l'obiettivo è quello di creare un'immagine dell'intero sistema di elaborazione dedicata a ciascun programma in esecuzione, anche se in realtà esistono più programmi caricati nel sistema. Questi vengono apparentemente eseguiti in parallelo, anche se in realtà vengono eseguiti un po' uno, un po' l'altro. Questo fornisce la possibilità ai programmi di non considerare la presenza di altri programmi quando devono operare

all'interno del sistema di elaborazione, e inoltre semplifica la programmazione, in quanto se il programmatore immagina di poter avviare una macchina virtuale dedicata per ogni suo programma, può ignorare i problemi di condivisione delle risorse e di eventuali necessità di interagire con altri programmi per ottenere l'accesso alle risorse desiderate (e quindi ogni programma vedere una macchina tutta per sé, mentre in realtà la macchina è ad uso condiviso).

Organizzazione logica

Un sistema di elaborazione è composto da un processore, una memoria centrale e un insieme di periferiche. Questo è l'aspetto hardware che vogliamo gestire in maniera ottimale nel sistema operativo.
Il primo insieme di attività che devono essere svolte dal sistema operativo è quello di rendere astratto e virtuale l'uso del processore. Questo insieme di attività viene svolto all'interno del SO dalla **Gestione del processore**.
Questo consente per un insieme di programmi in esecuzione di mostrare un processore dedicato a ciascuno di questi.
Un secondo insieme di funzioni è la **gestione della memoria centrale**. Questo insieme di funzioni crea un'immagine virtuale della memoria, facendo sì che ciascun programma in esecuzione abbia accessibile una memoria completamente dedicata a sè (anche se in realtà non è così, fisicamente ha solo una porzione).
Un terzo insieme di funzioni è la **gestione delle periferiche di I/O**. Questo consente di creare un'astrazione e una virtualizzazione delle periferiche in modo che l'insieme delle periferiche fisiche venga visto come dedicato a ciascuno dei programmi in esecuzione, anche se in realtà ad uso condiviso.
L'insieme di queste tre funzionalità permette di creare l'illusione per ciascun programma di aver a disposizione un proprio processore, una propria memoria, e le proprie periferiche.

Un secondo aspetto dell'organizzazione logica del sistema operativo mira ad elevare ulteriormente il livello di astrazione, in quanto con le modalità appena indicate abbiamo sì fornito ad ogni programma una sua "macchina di Von Neumann", ma ancora questo programma deve conoscere la struttura della macchina stessa, sapere quindi dove sono poste le varie informazioni.
Il sistema deve reperire le informazioni conoscendo la posizione fisica, e questo è scomodo.
Dunque il sistema può fare un ulteriore salto di qualità, fornendo ai programmi una visione ancora più astratta ed elevata in cui si provvede a nascondere i dettagli di dove sono poste le informazioni permettendo un reperimento di tipo logico e non più in base alla posizione fisica all'interno del sistema. Questo ulteriore insieme di funzioni è la **gestione del File System**.
Questo consente di organizzare le informazioni, e in generale tutte le risorse del sistema in modo che sia possibile reperire tali informazioni individuandone un nome in modo logico all'interno del sistema.
Oltre a questo, il sistema avrà a disposizione un insieme di funzioni che permettono di interfacciare gli utenti e i programmi applicativi al sistema stesso. È quello che costituisce **l'interfaccia del sistema verso gli utenti**, sia verso l'utente fisico che verso i programmi applicativi in modo da supportare la chiamata alle funzioni fornite dal SO.

Funzioni per il processore

Le funzioni messe a disposizione per creare questi ambienti virtuali sono diverse e di diversa natura. In particolare, per la gestione del processore è necessario mettere a disposizione delle funzioni che gestiscano i programmi in esecuzione.
Nei SO moderni, i programmi in esecuzione sono chiamati "*processi*", in quanto sono sequenze di attività svolte dal sistema.

È quindi necessario mettere a disposizione funzioni per la creazione e la terminazione di processi, la sospensione e la riattivazione di essi.
La gestione ottimale del processore, e la sua condivisione nel tempo, viene ottenuta attraverso l'esecuzione di particolari algoritmi di schedulazione dei processi sul processore, che determinano l'ordine con cui questi processi vengono eseguiti per far apparire condiviso l'uso del processore ai singoli processi, e quindi creare la virtualizzazione del processore per i processi.
I processi possono interagire tra di loro sincronizzandosi per l'uso delle risorse condivise, e un insieme di funzioni dovrà quindi supportare tale sincroniccazione.
I meccanismi di sincronizzazione possono portare ad attese infinite, e il sistema operativo deve risolvere queste situazioni, in cui ciascun processo è in attesa di risorse possedute da altri processi che non verranno mai rilasciati, in quanto altri sono in attesa di risorse del primo processo (deadklock).
I processi non solo competono per l'uso di risorse condivise, ma possono anche cooperare, e quindi serve un insieme di funzioni per la comunicazione tra i processi.

Funzioni per la memoria centrale
Un altro insieme di funzioni è quello per la gestione della memoria centrale. I processi sono sono in mc per la loro esecuzione, e quindi deve essere assegnata loro una porzione di memoria affinché il processore possa trovare le istruzioni e i dati necessari all'esecuzione.
Queste funzioni costituiscono il supporto per la multiprogrammazione, per la condivisione della memoria centrale, e devono provvedere all'allocazione e alla deallocazione di porzioni di memoria per i vari processi, nonché al caricamento in mc dei processi necessari per l'esecuzione.
Ovviamente devono anche fornire un'insieme di funzioni di protezione in modo tale che le porzioni in uso da un processo non possano essere toccate da altri processi.

Funzioni per le periferiche
Per la gestione delle periferiche abbiamo bisogno di fornire una omogeneità di interazione. Il SO ha come obiettivo quello di configurare e inizializzare le periferiche, fornire un'interfaccia unica ed omogenea delle periferiche e ottimizzarne la gestione in un ambiente condiviso.
Dunque devono essere organizzati dei meccanismi di bufferizzazione e di caching delle info per renderne più veloce l'accesso da parte dei processi.

Funzioni per il file system
Il livello del file system deve permettere di creare i file come componente elementare che descrive sequenze di informazioni, e un'organizzazione logica di questi file all'interno del sistema (detto "albero dei direttori del file system").
Tali funzioni devono gestire file e direttori in modo da garantire:
- creazione e cancellazione
- leggere, scrivere e copiare
- ricerca
- protezione e sicurezza
- di fare valutazioni di uso in modo da gestirne l'accounting
- salvataggio e ripristino

Funzioni per l'interfaccia utente
Interfacciarsi col SO significa fornire le funzioni per utenti e processi che permettono di dare comandi e ricevere i risultati delle elaborazioni. Esiste quindi un interprete dei comandi per gli utenti con cui gli stessi interagiscono, e un interprete di comandi per i programmi applicativi, in modo tale da poter chiamare le funzioni di sistema. Le librerie di sistema costituiscono quindi il modo con cui i programmi possono interagire con il SO. Ovviamente devono essere fornito anche un insieme di funzioni per l'autenticazione degli utenti, per garantire l'accesso alle risorse in modo

protetto. Un altro insieme di funzioni invece dovrà occuparsi della gestione degli errori e malfunzionamenti nel sistema.

M2 - UD1 - Lezione 2 - Architetture dei sistemi operativi

Sistema Monolitico

Il più vecchio tipo di architettura. In questo tipo il SO era un contenitore in cui venivano inserite funzioni senza una specifica strutturazione, ma semplicemente organizzando le chiamate di funzione interne al SO come poteva essere utile dal punto di vista della programmazione del sistema.
Questo faceva sì che la struttura globale del sistema potesse contenere dei modi di chiamare le varie funzioni per cui le funzioni di livello più astratto potevano essere chiamate da funzioni di livello più basso, più semplici. Ciò poteva portare ad una non chiarezza della dipendenza delle funzioni una dall'altra, per cui la manutenzione poteva diventare molto difficile.
Questo era il tipico approccio dei primi SO in cui la quantità di funzioni era limitata, in quanto le macchine da gestire erano semplici e il tipo di gestione era fondamentalmente orientato alla gestione a lotti dei programmi.

Sistema con struttura gerarchica

Fornisce una strutturazione di tipo gerarchico alle chiamate alle funzioni, facendo sì che le funzioni di alto livello possano chiamare unicamente funzioni di più basso livello, in modo da eliminare le relazioni tra funzioni semplici usate dai bassi livelli e quelle di alto livello.
Sostanzialmente si sono andate ad identificare delle dipendenze gerarchiche tra le funzioni, e ciascuna funzione è stata posta al livello gerarchico corrispondente, chiarendo la struttura (da un grafo completo contenente eventualmente anche dei cicli ad un'organizzazione di tipo strettamente gerarchico).
La manutenzione si è semplificata, seppur è rimasta difficile, in quanto non c'è chiarezza di quali siano i ruoli delle singole componenti all'interno del sistema. La separazione nei livelli gerarchici è puramente legata alla dipendenza delle chiamate.

Sistema stratificato

Si è pensato quindi di introdurre una chiara separazione modulare delle funzioni svolte da ciascun componente nel SO, ed è stato così introdotto il sistema stratificato.
Qui abbiamo una gestione del processore, che costituisce la componente di base, sulla quale si appoggia la gestione della memoria centrale, la gestione delle periferiche, e al di sopra di quest'ultima la gestione del File System, e infine la gestione dell'interfaccia utente.
In questo modo le funzioni sono gerarchicamente separate in livelli, con la conseguenza che l'efficienza del sistema risulta limitata, ridotta rispetto al caso più semplice precedente, in quanto si ha che un livello deve chiamare i livelli inferiori per poter accedere alle operazioni di base del sistema, per la virtualizzazione dei singoli componenti.

Sistema a microkernel

Per superare il problema di comunicazione tra i livello, in cui diventava critico il tempo di risposta agli utenti, si è cominciato a sviluppare un approccio diverso.
Si è considerata anche la necessità di aggiornare e modificare il sistema senza mai fermarlo, e quindi dare una continuità ai servizi.
Ciò introduce il concetto di sistema a microkernel.
Si è pensato di separare rigidamente i meccanismi dalle politiche, ovvero di separare quella serie di operazioni di gestione di accesso alle singole risorse che non possono cambiare mai, anche al cambiare del modo con cui si desidera ordinare od effettuare la gestione (meccanismo), dalla definizione astratta di diritto di uso della risorsa, dell'ordine di uso della risorsa, dalla priorità relativa tra i vari processi per accedere all'uso della risorsa (politica).

Si ha quindi un microkernel in cui si hanno i **meccanismi** per la gestione del processore, della memoria e la gestione dell'ingresso e uscita e così via. Tutto ciò che è meccanismo per la gestione degli aspetti di base nel SO, viene messo nel microkernel.
Tutto ciò che è **politica**, cioè regole di gestione (e non operazioni) vengono posti al di sopra di questo microkernel come politiche, ad esempio per il file system, per lo scheduling dei processi ecc.
Addirittura queste porzioni possono essere realizzate come processi in esecuzione sul sistema.
La modificabilità è molto facile, il problema è che le prestazioni possono diventare limitate, soprattutto in sistemi complessi

Sistema a moduli funzionali
Questo sistema, basato sulle tecnologie di sviluppo software più recenti, ossia lo sviluppo con linguaggi oggetti, garantisce un'ottima modificabilità.
In questo caso si ha un kernel del sistema con i meccanismi di base, e intorno a questo vengono costruiti dei moduli che si combinano in maniera modulare, e che consentono l'inserimento e l'estrazione dei componenti senza dover rimodificare il resto delle parti del sistema.
L'introduzione di oggetti e relativa gestione porta ad un overhead di esecuzione, che riduce le prestazioni globali.

Sistema a macchine virtuali
Per astrarre ulteriormente, e fare in modo che ogni programma in esecuzione veda un suo proprio ambiente, eventualmente un ambiente diverso dagli altri, sono state introdotte le macchine virtuali, ovvero si ha alla base l'hardware, e quindi un kernel di virtual machine che provvede a replicare esattamente l'hw alla base, senza alcuna astrazione o modifica, una copia per ogni insieme di processi che si vogliono eseguire su una VM specifica.
Questo kernel di VM genererà tante macchine virtuali sulle quali si potrà installare per ciascuna uno specifico sistema operativo.
Su ciascuno di questi ambienti diversi ci saranno i vari processi in esecuzione. Ciascuno vedrà il suo proprio sistema operativo senza sapere che in realtà ci sono in contemporanea altri ambienti operativi disponibili nel sistema.
Ciascuno dei sistemi operativi che vengono installati al di sopra del kernel che realizza la VM vede la macchina hardware così com'è, come fosse appoggiata direttamente sull'hardware. Questo consente di avere la convivenza di diversi sistemi operativi.
Ovviamente si paga questa astrazione e modularità con prestazioni ridotte.
Sostanzialmente ciascuna delle VM può essere utilizzata eventualmente da un solo processo, e quindi il VM kernel consente di virtualizzare completamente le risorse, rendendo minima la capacità di gestione a livello superiore.
Tutte le funzioni che si possono mettere a livello superiore sono quelle che permettono di astrarre e semplificare l'uso delle risorse come se fossero completamente dedicate al singolo processo.

Programmi di sistema
All'interno del sistema si ha quindi un insieme di programmi di sistema che permette di gestire in modo ottimale le risorse.
Ad esempio si ha una serie di funzioni per

- la gestione dei file
- reperire informazioni sullo stato
- attivare l'esecuzione di programmi
- eseguire le comunicazioni all'interno del sistema

Si ha poi un insieme di risorse che permette di sviluppare applicazioni:

- editor per scrivere i programmi

- compilatori e assemblatori per generare i codici oggetto
- linker per generare il codice eseguibile mettendo insieme i vari oggetti creati separatamente
- debugger per verificare il corretto funzionamento dei programmi in un ambiente speciale, controllato e guidato

Questo insieme di programmi completa il supporto fornito dal SO in modo tale che l'utente abbia sia le varie funzioni messe a disposizione secondo le strutture viste che dei supporti più complessi che vengono utilizzati come programmi normali da mandare in esecuzione sull'architettura di base, qualsiasi essa sia.

M2 - UD1 - Lezione 3 - Generazione e avvio di un sistema operativo

Generazione del sistema operativo

Generare il sistema operativo consiste nell'identificare le caratteristiche dell'ambiente in cui vogliamo far operare i programmi e gli utenti per quella specifica installazione.
Quindi si dovranno capire le caratteristiche degli utenti, dei programmi applicativi, e i carichi di lavoro che questi generano, le richieste di uso di risorse che essi generano, in modo da definire i parametri ottimali per il SO per gestire in modo efficiente le risorse, e per garantire un'equa ripartizione dell'uso delle risorse tra le varie tipologie di utenti.
È necessario poi applicare questi parametri per generare nuovo codice eseguibile, e infine memorizzarlo e caricarlo per poter far ripartire il SO con la nuova configurazione applicata.

Identificazione delle caratteristiche

Identificare le caratteristiche è un'attività che richiede un'accurata analisi delle applicazioni che si vogliono utilizzare, delle caratteristiche del carico di lavoro sul processore, e in generale di uso delle risorse del sistema, sia risorse informative che fisiche.
Si dovrà analizzare qual è l'ambiente operativo e gli utenti che vogliono usare questa applicazione, per comprendere le loro abitudini, e quindi i carichi di lavoro che vanno a indurre sulle applicazioni.
Valutare le caratteristiche del carico di lavoro prodotto dai vari applicativi consiste nel mettere insieme tutte le info raccolte sul modo di uso e di funzionare degli applicativi per creare uno scenario, un modello, che descriva il carico di lavoro e di richieste per le singole risorse fisiche o informative del sistema, e dunque valutare non solo l'hardware ma le caratteristiche del SO, e come questo deve rispondere in modo più efficiente possibile alle richieste degli utenti.
La raccolta delle info può essere manuale o automatica, monitorando il comportamento degli utenti in un ambiente simulato, oppure ponendosi nell'ambiente reale, magari con la versione corrente del SO, e osservando come gli utenti lavoro e collezionando informazioni sulle caratteristiche di uso da parte degli utenti sulle varie risorse.
Valutare le caratteristiche dello scenario operativo può essere fatto sulla base dell'esperienza o sulla base di statistiche.

Definizione dei parametri

Per definire i parametri della nuova installazione del sistema operativo è necessario analizzare i carichi di lavoro, il modello che così si è costruito, e valutare quali sono i migliori parametri che portano il SO a comportarsi il meglio possibile rispetto alle richieste degli utenti e delle applicazioni che essi lanciano.
Può essere fatto tramite esperienza, analisi statistiche, confronto con casistiche predefinite e con l'individuazione di regole che permettono di guidare il computo dell'insieme di parametri migliori per il sistema.

Applicazione dei parametri

Consiste nella modifica dei file di configurazione del sistema operativo memorizzando i nuovi valori dei parametri, attivare le procedure che generano il codice eseguibile per i moduli che sono stati modificati in questa fase di configurazione, e quindi generare il codice complessivo del SO e di tutti i programmi di sistema relativi

Aggiornamento di sistema
Consiste nella memorizzazione della nuova versione del SO e dei programmi di sistema, nonché nel caricamento in memoria centrale del nuovo SO, cedendo il controllo poi al SO con i nuovi parametri.

Avviamento del sistema operativo
Si tratta di un'operazione che può essere realizzata all'accensione del nostro sistema di elaborazione in vari modi a seconda del supporto hardware del sistema.
I metodi di avviamento del sistema (bootstrap) sono vari, possono essere fatti in uno o più passi, e questo consente, grazie al diverso modo di memorizzare tutto o parte del SO nei vari componenti di memoria, di ottenere diversi gradi di modificabilità del SO stesso o dall'altra parte diversi gradi di efficienza di accesso alle funzioni di sistema in fase di esecuzione o all'avviamento.
Ad esempio, nel caso di sistemi embedded è utile avere un rapido accesso alle funzioni del SO perché si vuole una forte capacità di risposta in tempi brevi alle richieste dell'ambiente e dell'utente.
In un sistema interattivo (PC, mainframe) diventa utile poter modificare il SO per poter aggiungere funzionalità a seconda dell'evoluzione delle esigenze degli utenti e a seconda del cambiare dei carichi di lavoro indotti dagli utenti.

Metodi di avviamento
Singolo passo
In questo caso in memoria centrale vengono considerate due porzioni: una è realizzata tipicamente come RAM in cui è possibile effettuare le usuali operazioni di lettura e scrittura, mentre un'altra è realizzata come ROM, ossia come memoria di sola lettura.
Questa ROM contiene i valori prima memorizzati anche in caso di spegnimento del sistema, e all'accensione continuerà ad avere quindi sempre gli stessi valori.
Ciò permette di configurare l'architettura hardware del processore in modo tale da andare a caricare nel program counter il primo indirizzo del sistema operativo, da cui ne parte l'esecuzione, come uno degli indirizzi posti nella ROM.
Questo consente una rapida accessibilità delle funzioni, dunque i tempi di accesso sono quelli della memoria centrale. Ciò però ha un grosso problema: il SO non ha modificabilità, e infatti per modificare il sistema operativo andrebbe sostituito fisicamente il componente ROM, cosa non sempre facilmente fattibile.

Due passi
Questo è diretto a ridurre la complessità dell'aggiornamento permettendo una modificabilità più semplice il sistema operativo.
L'idea è quella di non mettere più in memoria centrale tutto il sistema operativo in modo fisso, ma di avere nella ROM solamente il caricatore (loader), ovvero quella porzione di SO che verrà attivata all'accensione, la quale si preoccuperà di fare soltanto qualche inizializzazione minima dell'architettura hw, ma soprattutto di andare a reperire su un'altra memoria di massa facilmente modificabile (es. disco magnetico) in una posizione predeterminata (es. disco 0, traccia 0, settore 0) il SO vero e proprio. Il caricatore farà quindi sempre riferimento a questa porzione di memoria, e una volta completata l'inizializzazione minima del processore e delle periferiche, provvederà ad accedere al disco rigido per acquisire il SO. Il caricatore non è modificabile, ma il sistema operativo sarà modificabile, in quanto su un disco.

La seconda fase di questo avviamento (bootstrap secondario), consiste nel caricare dalla posizione fissa predeterminata dell'hd il resto del sistema operativo.
Una volta completato il caricamento da parte del loader in ROM, il SO posto in una porzione della ram, potrà essere abilitato a prendere il controllo del sistema (il loader, terminato il caricamento, cede al SO il controllo del calcolatore).

Tre passi
Questo ha come obiettivo quello di superare il limite imposto dal fatto che il SO deve stare in un solo settore fisso del SO (limitante dal punto di vista di dimensione). L'idea è quella di avere in memoria centrale un piccolo caricatore posto in ROM, un caricatore di base, che provvede a mantenere il riferimento a una porzione dell'hd in cui si trova il caricatore vero e proprio.
La posizione del loader nel disco rigido dovrà essere fissa e nota al loader in ROM.
Effettuata l'identificazione di questa porzione, questa andrà a contenere non solo più il sistema operativo, ma il caricatore completo.
Questo caricatore verrà posto in una porzione di RAM dal caricatore di base e sarà questo caricatore che andrà a sapere dove, all'interno dell'hd, si troveranno le varie porzioni di sistema operativo che devono essere caricate.
Il terzo passo consisterà nel caricare in memoria centrale il sistema operativo vero e proprio.
Il caricatore in RAM può essere abbandonato, e rimarrà presente in RAM il SO vero e proprio.
Il caricamento può essere reso complesso ulteriormente andando a caricare non tutto il sistema operativo in un colpo solo in ram, ma andandolo a caricare solo quando richiesto (caricando i singoli moduli quando servono, per eseguire le specifiche funzioni richieste dagli ambienti applicativi che verranno attivati).

Passi multipli
Si può pensare ad ulteriori passi per garantire una migliore modificabilità del SO, però ciò porta ovviamente ad un caricamento del SO più complesso e lento, e un'accessibilità ridotta alle funzioni non caricate che devono essere caricate su richiesta, prendendole dalla memoria di massa.

M2 - UD1 - Lezione 4 - Interfacce dei sistemi operativi

Interfaccia Utente
L'interfaccia utente è costituita dall'interprete dei comandi forniti dall'utente, detta anche "shell".
L'interprete dei comandi rimane inizialmente in attesa che l'utente digiti il comando desiderato, e rimane in un'attesa infinita di questa sequenza di caratteri proveniente dall'utente, finché l'utente dichiara di terminare il comando digitando il tasto "carriage return" (fase di fetch).
Una volta effettuato questo l'interprete comando provvede ad analizzare e verificare l'esistenza e la correttezza del comando digitato, in modo da essere certi che sia un comando ammissibile per il sistema (fase di decode).
Dunque se il comando è corretto, il comando viene mandato in esecuzione, attivando un processo ausiliario (associato al comando) a cui vengono passati i parametri specificati a seguito del comando (fase di execute). Al termine dell'esecuzione del processo ausiliario, il controllo ritorna all'interfaccia utente, la quale provvede a mettersi ancora in attesa di un comando da parte dell'utente.

Interazione con utente
L'interazione con l'utente può avvenire in due modi:

- Interazione di tipo testuale: prevede che l'utente digiti da tastiera il comando e gli eventuali parametri complessivamente con il simbolo di terminazione del comando (\r)
- Interazione di tipo grafico: si superano i problemi di digitazione del comando e dei parametri (in quanto l'utente può commettere errori, che portano all'esecuzione del comando su parametri sbagliati o in un'inammissibilità del comando). L'interfaccia grafica si

basa su icone e menù che permettono di selezionare graficamente il comando e gli eventuali parametri, limitando la digitazione ai parametri strettamente indispensabili. Attivare il comando significa semplicemente cliccare sull'icona che rappresenta il comando, selezionare i parametri (o una funzione) vuol dire selezionare l'opzione desiderata attraverso il menù all'interno del comando. La digitazione viene quindi riservata alla specifica da parte dell'utente di quali sono le risorse sulle quali deve essere applicata l'attività del comando. Se la risorsa è già presente nel sistema può essere sostituita da una selezione grafica, riducendo ulteriormente la probabilità di errore.

Interfaccia programmativa

L'interfaccia programmativa è un'interfaccia che permette ai programmi di richiedere e ottenere le funzioni fornite dal SO. Queste sono chiamate "chiamate di sistema" (o "chiamata a supervisore" o "chiamata a monitor").
Obiettivo di questa interfaccia e della chiamata di sistema è quello di proteggere il SO garantendone l'integrità dei dati e la completa esecuzione delle procedure che si intende attivare e soprattutto di garantire che gli eventuali controlli previsti dalle procedure vengano effettivamente eseguiti. Ciò vuol dire che la chiamata di sistema è molto simile ad una normale chiamata di procedura, ma deve aver qualcosa in più per garantire quanto detto.

Chiamata a sistema

Una funzione del sistema operativo sarà divisa in due parti: una parte di controllo sull'effettiva ammissibilità della chiamata da parte del processo per quell'utente, mentre la seconda esegue effettivamente il comando specificato.
Se il programma applicativo avesse la possibilità di chiamare la funzione del SO come una normale procedura potrebbe succedere che un programmatore sveglio possa cercare di bypassare i controlli andando direttamente alla fase di esecuzione. Questo potrebbe portare ad attivare la procedura scelta anche nel caso in cui non sia lecito.
Per evitare situazioni di questo tipo è necessario forzare l'accesso e controllarlo attraverso la tecnologia "trap" (interruzione attivata via software).
Il programma non chiamerà quindi direttamente la procedura, ma piuttosto andrà a valorizzare dei registri all'interno del processore con delle informazioni che identificano la procedura di sistema desiderata, e andrà a caricare i parametri che dovranno essere utilizzati da tale procedura.
A questo punto il programma chiama un'istruzione di "trap" (interruzione da software), e questa, durante la sua esecuzione, porta il processore ad attivare un'interruzione. Il SO assocerà a tale interruzione la sua risposta, e per questa interruzione specifica il sistema definirà un modo di verificare quale era la funzione richiesta dal programma applicativo e ad eseguirla.
La risposta all'interruzione generata dalla trap provvede ad andare in una tabella del SO in cui sono elencate tutte le funzioni, e dove queste sono memorizzate nello spazio riservato al sistema stesso. In pratica, la trap va a chiamare questo interprete comandi, che contiene la tabella delle funzioni di SO, e questo interprete comandi provvede, leggendo dalla tabella, a chiamare la funzione desiderata. Quando la funzione desiderata termina, si ritorna all'interprete comandi contenuto nella risposta alla trap prevista per chiamare il SO, la trap a sua volta termina ritornando al programma applicativo che l'aveva chiamata.
Meccanismo complesso, ma che garantisce che non si possa accedere direttamente alle istruzioni contenute nella funzione desiderata, in quanto non è il processo a richiamare la procedura desiderata, ma è il sistema stesso che chiama la procedura durante l'esecuzione della trap.

M3 - Gestione del Processore
UD1 - Processi
Lezione 1 - Processi

Multi-Tasking

Un qualunque sistema di elaborazione vede sfruttare poco il processore, in quanto generalmente le operazioni di ingresso e uscita sono usualmente molto più lente delle operazioni svolte all'interno del processore.
L'esecuzione delle istruzioni nel processore (interne o che coinvolgono l'accesso alla memoria centrale) coinvolgono dei tempi rapidi in quanto vengono utilizzati dei dispositivi elettronici (e quindi con tempi di reazione del sistema elettronici).
Nel caso delle periferiche abbiamo la necessità di interagire con il mondo esterno, e quindi di effettuare operazioni di tipo meccanico, elettronico, ottico con il mondo esterno, attivate spesso da dispositivi che richiedono attività elettromeccaniche, e queste sono attività molto più lente rispetto a quelle del processore.
Questa lentezza relativa fa sì che il processore debba rimanere in attesa, e quindi non può usare computazioni utili per far avanzare il programma in esecuzione.
Una soluzione a questo, che tende a ridurre il costo e i tempi morti di uso del processore è l'idea di eseguire più programmi contemporaneamente (apparentemente), ossia del multitasking.
Questo modello indica che se il processore è in attesa del completamento di una operazione di I/O, questo deve passare all'esecuzione di un altro programma, cercando di mantenere il processore nello stati di esecuzione di programmi il più possibile.

Ciò si realizza utilizzando il concetto di multiprogrammazione (più programmi caricati in memoria centrale) perché devono essere pronti per essere eseguiti (il processore è in grado di eseguire solo istruzioni presenti in memoria centrale).
Un altro aspetto necessario è quello del multi-tasking, che preveda la gestione della turnazione dei programmi quando il programma in esecuzione è in attesa della risposta delle periferiche.
Il processore quindi viene ad essere trattato in modo tale che gestisca più flussi di esecuzione indipendenti tra di loro, ovvero appare eseguire più programmi in parallelo (in realtà comunque il processore eseguirà sempre e solo un'istruzione di un solo programma alla volta).

Processo

Il processo è un programma in esecuzione. È costituito da un insieme di componenti che sono disposti in memoria centrale per poterne supportare l'esecuzione. Vi è il codice del programma (ossia le istruzioni eseguibili del programma), l'insieme dei dati del programma (variabili globali, variabili locali e non locali (nello stack viene creato il contesto di esecuzione che contiene le variabili non locali), variabili temporanee generate dal compilatore per supportare l'esecuzione di istruzioni complesse (poste in genere nei registri), variabili allocate dinamicamente con le funzioni apposite del linguaggio di programmazione (poste nello heap).
Lo stato di evoluzione della computazione è costituito dal program counter e dall'insieme di tutti i valori delle variabili (dato che il codice eseguibile evolverà in funzione del valore di esse).

Il programma NON È un processo. È invece la sequenza di istruzioni generata dalla compilazione del codice sorgente.
Il processo è un'entità attiva, è il programma in esecuzione, l'insieme dei valori delle variabili e delle risorse in uso che vengono utilizzate dal programma.
Quindi l'evoluzione della computazione di un processo sarà l'insieme delle attività che sta svolgendo il processo durante la sua esecuzione.
Quindi il processo è una funzione (dal punto di vista astratto) che trasforma informazioni, eseguendo le istruzioni, partendo dai valori iniziali (costanti e valori acquisiti dall'esterno dalle varie periferiche) fino a produrre i risultati finali che vengono emessi attraverso le periferiche.

Dunque il processo può essere visto come una macchina a stati finiti. Gli stati sono le informazioni su cui opera e le transizioni sono le istruzioni che modificano le informazioni.

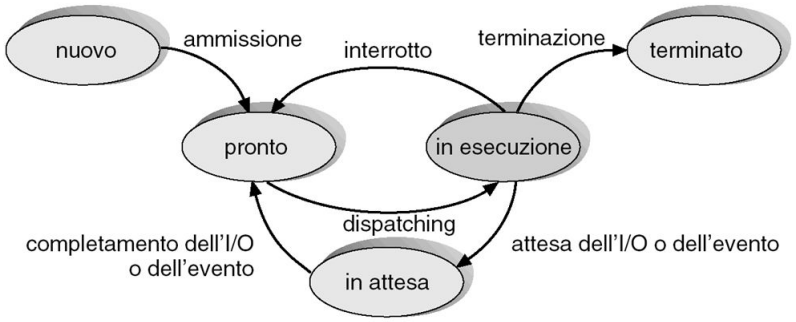
Stato di evoluzione della computazione di un processo
Si tratta dell'insieme dei valori di tutte le informazioni da cui dipende la computazione del processo.
Quindi sarà l'insieme dei dati globali, dati allocati dinamicamente nello heap, dati posti nello stack, più il contenuto dei vari registri nel caso in cui ci siano delle variabili temporanee introdotte dal compilatore o risultati temporanei dell'elaborazione prodotta dalle istruzioni, più il program counter.
Quando vogliamo cambiare il processo in esecuzione è necessario quindi salvare lo stato di evoluzione della computazione.
Se vogliamo sospendere l'esecuzione di un processo per poter andare ad eseguire un'altro mentre il primo sta attendendo la risposta da una periferica, quando torneremo ad eseguire il primo processo dobbiamo tornare esattamente nello stesso stato di evoluzione della computazione, avendo gli stessi valori valori per le variabili, in modo che la computazione possa procedere come se non avessimo mai cambiato il processo.
Alcuni linguaggi di programmazione prevedono un codice automodificante, in questo caso anche il codice diventa parte dello stato di evoluzione della computazione, in quanto dovrebbe essere salvato per sapere qual'era il suo valore quando si torna ad eseguire il programma stesso.
Nel caso il linguaggio non sia di questo tipo, non è necessario salvare il codice, in quanto questo non subirà modifiche in memoria centrale.
Non è necessario nemmeno salvare i dati globali, in quanto questi rimangono in memoria centrale. Se noi garantiamo che la memoria di un processo è accessibile solo al processo in esecuzione (che possiede quella parte di memoria) quella parte di memoria non sarà toccata da altri processi, e quindi non subirà variazioni.
Stesso discorso vale per lo heap, che è ancora nello spazio di indirizzamento del processo.
Ciò che non è nello spazio di indirizzamento e non è salvato in una memoria non toccata dagli altri processi è il contenuto dei registri. Questi devono essere salvati da qualche parte in modo da poter essere ripristinati e garantire l'evoluzione della computazione esattamente dal punto in cui era stata sospesa. Stesso discorso per il program counter. Tutti possono essere salvati in strutture dati del SO, oppure possono essere memorizzati semplicemente sulla cima dello stack, in modo che da lì possano essere recuperati successivamente.
Quando il sistema tornerà in esecuzione, troverà a partire dalla posizione individuata dallo Stack Pointer il valore del program counter e dei registri, e potrà ripristinarli.
Anche il contenuto dello stack rimane salvato, in quanto in memoria centrale, e quindi non deve essere salvato altrove. Il problema sorge con lo stack pointer, che è un registro e deve essere per forza salvato opportunamente per poter garantire di poter ritrovare la cima dello stack da cui recuperare le variabili che devono essere caricate nei registri e il program counter.

Uso del processore da parte di un processo
In questa fase, le attività svolte sono l'uso del processore per eseguire effettivamente la computazione, oppure un processo può attendere di attendere di ottenere l'uso del processore pur avendo tutto ciò che gli serve (risorse informative o fisiche). Teoricamente potrebbe evolvere, ma non riesce a farlo in quanto il processore non sta eseguendo le istruzioni del programma.
Un processo può anche dover attendere che una risorsa informativa o fisica diventi disponibile. In questo caso il processo non può evolvere, in quanto gli manca la risorsa necessaria.
Ciò introduce il concetto di "stato del processo", o più precisamente "stato di uso del processore da parte di un processo" come la modalità con cui il processo sta usando il processore.
Stati del processo:

- NEW: Quando il processo viene creato
- RUNNING: Quando entra in esecuzione ed usa il processore

- READY TO RUN: pronto per essere eseguito e gli manca solo il processore ma ha tutte le altre risorse
- WAITING: quando deve attendere una risorsa o il completamento di un'operazione richiesta ad una periferica
- TERMINATED: Quando il processo termina la sua esecuzione

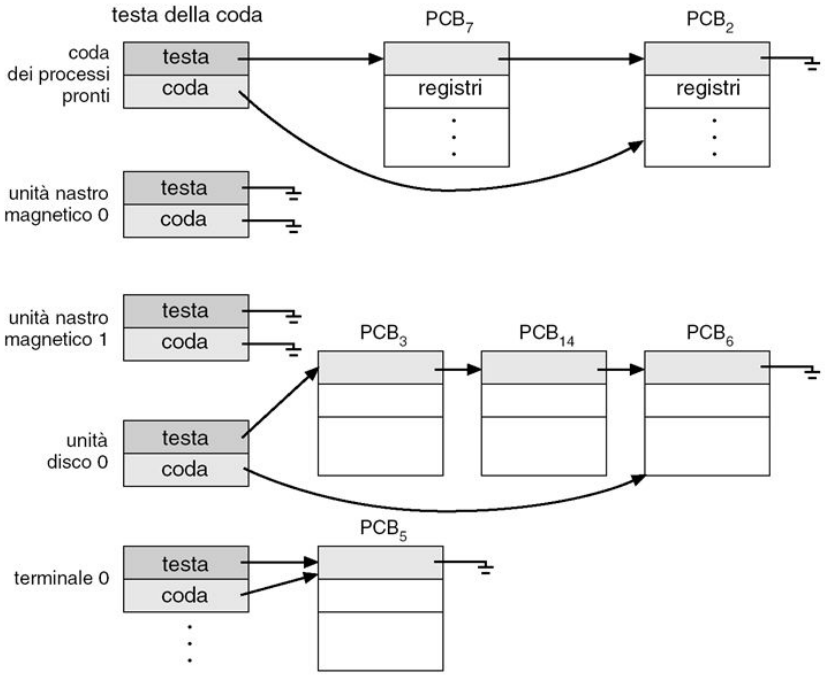
Diagramma degli stati del processo
Questo diagramma rappresenta l'insieme degli stati e le transizioni tra i vari stati.
È un grafo orientato i cui nodi sono gli stati in cui si può trovare un processo, e le transizioni sono gli archi che rappresentano come un processo passa da uno stato ad un altro.



- Durante la creazione vengono configurate tutte le info relative alla sua gestiona, viene riservato lo spazio in memoria centrale, vengono dichiarate quali risorse ha bisogno di utilizzare, vengono fatte delle operazioni di memorizzazione delle richieste di uso di risorse da parte del programma e infine il SO concede al nuovo processo di entrare nella competizione per utilizzare al processore, ponendolo nello stato di "Pronto".
- Dai processi in stato di pronto, il SO preleverà un processo secondo opportuni criteri, e lo porrà in esecuzione tramite il meccanismo del dispatching.
- Il processo in esecuzione userà il processore e potrà terminare (entrare nello stato di terminato, completare l'accounting, e quindi rimosso. In caso di terminazione a causa di errore potrà essere attivato il post-mortem debugging per vedere il motivo della terminazione), oppure il processo potrà mettersi in attesa di una risorsa o del completamento di una op. di I/O rilasciando il processore e quindi entrando nella coda di attesa per la relativa risorsa. Quando l'operazione è completata o la risorsa diventa disponibile il processo passa dallo stato di attesa allo stato di pronto (ha tutto eccetto il processore)
- Dallo stato di esecuzione il processo può essere forzatamente sospeso e tolto dall'esecuzione pur avendo tutte le risorse disponibili per poter operare, per lasciar posto ad altri processi. In questo il processo torna nello stato di "pronto", e tornerà in competizione con gli altri.

Supporti per la gestione dei processi
Per gestire i processi il SO mette a disposizione il Process Control Block (PCB) che raccoglie tutte le info riguardanti il processo:
- l'identificatore
- lo stato del processo
- il program counter
- gli eventuali registri (a meno che non vengano salvati nello stack, un questo caso viene salvato lo stack pointer)
- le info sulla schedulazione (come e quando può essere consegnato l'uso del processore)
- info sulla gestione della gestione della memoria centrale
- info sulle periferiche coinvolte
- info relative alla valutazione dei costi di uso del sistema

Questi descrittori che contengono le informazioni sui singoli processi possono essere accodati nelle varie code che rappresentano gli stati del processo.



Queste strutture sono mantenute dal SO.
Descrivono una coda dei processi pronti all'esecuzione.
Ci sarà un solo processo in "RUNNING", in caso di singolo processore. Nel caso di più processori ci sarà un processo in stato di "RUNNING" per ciascun processore.
Poi per ciascuna delle risorse si avrà un insieme di processi in attesa dell'esecuzione delle operazioni richieste o di ottenimento di disponibilità della risorsa.

Si può modellare l'evoluzione della computazione di un processo, e la transizione tra i vari stati con un modello a code, in cui si può vedere che il processore è un servizio per cui c'è una coda di processi pronti per l'esecuzione, e quando un processo viene creato viene inserito in questa coda di processi pronti. Quando un processo termina la sua computazione esce dal processore e non rimane più nel sistema. Se il processo invece esegue un'operazione di I/O, entra nella coda di I/O finché la periferica non completa l'operazione e il processo può ritornare nella coda dei processi pronti.

Lezione 2 - Creazione e Terminazione Processi

Processi come flussi di operazioni

Un processo è un flusso di operazioni che viene eseguito nel nostro sistema, quindi si può vedere come un flusso di esecuzione della computazione.
Se abbiamo attività separate possiamo vedere flussi separati nel sistema, e possiamo vederli dal punto di vista logico come processi separati.
Nel sistema ci sono flussi completamente indipendenti, che dipendono dal fatto che si possono avere vari utenti che hanno obiettivi indipendenti l'uno dall'altro. Questi flussi possono essere visti come dei processi che evolvono autonomamente.
Quindi si possono avere per un'applicazione più obiettivi che possono essere affrontati in maniera indipendente, e ciò può essere visto come un insieme di processi che evolvono in modo autonomo.
In molti altri casi la risoluzione di un problema complesso in un'applicazione può essere vista come un insieme di flussi sincronizzati, coordinati tra loro per raggiungere l'obiettivo (o l'insieme di obiettivi) globale. In questo caso si può descrivere la computazione mediante un insieme di

processi che evolvono sincornizzandosi tra di loro per il raggiungimento degli obiettivi e per l'uso condiviso delle risorse del sistema.
Questi processi possono anche comunicare tra di loro, cooperando in modo attivo.
Si può avere infine un insieme di processi separati che hanno comunque bisogno di essere sincronizzati, in quanti devono utilizzare delle risorse condivise che devono essere utilizzate in modo mutuamente esclusivo, per garantire la coerenza e la consistenza dell'uso della risorsa.
In questi casi, pur non essendoci una necessità di sincronizzare l'evoluzione della computazione dei processi (in quanto non c'è bisogno di svolgere un'attività prima o dopo un'altra dal punto di vista degli obiettivi delle singole applicazioni) esiste la necessità del sincronismo, in quanto è la risorsa condivisa che impone ciò per un suo uso corretto.

Modellazione della computazione a processi
Possiamo allora modellare la computazione della computazione in diversi modi.
Dal punto di vista astratto abbiamo un'applicazione che ha un certo insieme di obiettivi da raggiungere, e possiamo descriverla quindi come un unico processo, un unico flusso di attività, che pian piano risolvono i vari singoli sottobiettivi per raggiungere l'insieme degli obiettivi dell'intera applicazione; oppure possiamo descrivere l'attività di questa attività dell'applicazione come un'insieme di flussi, ciascuno diretto a risolvere una parte dell'applicazione globale in un modo cooperante (come un insieme di processi cooperanti).
Dal punto di vista del software che si sviluppa si può ottenere questo modello di come si esegue la computazione in vari modi:

- possiamo avere un unico programma che gestisce tutta l'applicazione (monolitico) il quale provvederà ad essere eseguito o come un singolo processo, oppure come un insieme di flussi di esecuzione cooperanti tra di loro.
- Si può modellare l'applicazione direttamente come un insieme di programmi separati che vengono mandati in esecuzione in modo coordinato per risolvere l'applicazione

Si tratta quindi di decidere se all'interno dell'applicazione generare delle porzioni di codice dirette a risolvere i sottoproblemi come flussi separati, oppure se spezzare completamente il codice dell'applicazione in programmi separati.
In tutti i casi tutti i programmi conosceranno l'esistenza degli altri, in quanto sapranno che ci sono vari sottobiettivi, ed ognuno di loro è conscio di qual'è il suo specifico sottobiettivo da raggiungere.

Dal punto di vista della realizzazione avremo le casistiche:

- Programma monolitico eseguito come processo monolitico
- Programma monolitico che genera processi cooperanti
- programmi separati eseguiti come processi cooperanti (che eventualmente generano altri processi cooperanti).

Ciò dà una maggiore modularità al sistema, perché permette di progettare l'applicazione focalizzando l'attenzione del progettista e del programmatore su un sottoaspetto alla volta, e ottenere la risoluzione del problema applicativo mediante il coordinamento globale delle varie parti realizzate con i vari programmi.

Generazione di un processo
Il processo in esecuzione può chiamare un'apposita funzione di sistema operativo (chiamata "fork") che produce una copia del programma, e attiva un nuovo processo.
Il processo in esecuzione svolgerà la sua computazione, e quando raggiungerà l'esecuzione della funzione fork, esso genera un processo. Questo processo generato prende il nome di "*processo figlio*". Il *processo padre* continuerà la sua computazione maniera indipendente o coordinata con il processo figlio generato. Nel sistema sono stati quindi attivati due flussi di computazione.

Le risorse per questi flussi possono essere:

- condivise tra padre e figlio
- parzialmente condivise (una porzione viene trasmessa ai figli, mentre una porzione rimane al padre)
- indipendenti dal padre, cioè ogni processo figlio acquisisce le info e le risorse che gli servono direttamente dal SO, senza ereditare nulla dal padre

Tra il padre e i figli è necessario trasmettere anche come informazioni un insieme di dati di inizializzazione per definire esattamente l'esecuzione dei vari processi. Esiste quindi la possibilità, quando si attivano i processi, di trasmettere ad essi delle sequenze di dati come informazioni di partenza di configurazione sulle quali operare.

Spazio di indirizzamento

Quando si ha un processo figlio, per garantire la corretta evoluzione della computazione dei processi, si ha sempre che codice, dati, heap e stack del processo padre sono distinti dagli stessi spazi per il processo figlio.
Questo permette di garantire la separazione degli spazi di memoria centrale, e permette di garantirne la protezione, e quindi l'evoluzione corretta e consistente della computazione senza che un processo possa accidentalmente o malevolmente modificare le informazioni su cui l'altro processo evolve.

In alcuni casi si usa avere una copia esatta dello spazio del processo padre, con codice, dati, heap e stack, in modo tale che il processo figlio veda, almeno al momento dell'attivazione, esattamente tutto quello che vedeva il padre all'atto della creazione.
Chiaramente Dati, Heap e Stack, all'inizio della computazione del processo figlio, evolveranno in modo autonomo e separato rispetto a quello del padre.
In altri casi invece si avrà un nuovo contenuto nello spazio di indirizzamento del figlio (nuovo codice, heap, stack e dati), senza ereditare nulla dal padre. Quindi, verrà caricato un nuovo programma (per questo è la funzione del sistema operativo "**exec**" che provvede al caricamento).

Questo modello della memoria separata può diventare di difficile utilizzo se si vogliono condividere delle porzioni di memoria centrale. In questi casi il SO dovrà provvedere a mettere a disposizione delle funzioni specifiche per garantire la possibilità di condividere porzioni di memoria centrale.

L'esecuzione dei processi può avvenire in vari modi:

- Il padre può eseguire la sua computazione, raggiungere il punto in cui genera il processo figlio e quindi continuare nella sua propria computazione, mentre il figlio parte con la sua computazione separata. Queste due computazioni possono avvenire parallelamente, in maniera indipendente
- Il padre può eseguire la computazione fino al punto in cui genera il processo figlio, provvedere ad attivare il figlio e sospendersi, attendendo che il figlio termini la sua computazione, e soltanto quando il figlio termina riprendere con la sua computazione. Ciò si ottiene con un'operazione esplicita di chiama al SO di **wait** del processo figlio

Albero dei processi

Questo può portare a generare tutto l'insieme di processi che si hanno attivi all'interno del sistema di elaborazione. usualmente quando il SO viene caricato, dopo una fase di init, viene generato un processo "padre" di tutti i processi del sistema, una radice dell'albero dei processi, che effettua un insieme di operazioni di inizializzazione del sistema e provvede in particolare ad attivare tutti i processi di sistema che gestiscono l'ambiente a livello di SO per gli utenti (es. i processi per scaricare e caricare i programmi da memoria di massa, ecc.).

In particolare, viene generato un processo “**init**” che è il processo padre di tutti i processi di interfaccia utente. Quando un terminale si accende lancia un’interruzione al SO, segnalando che un utente vuole lavorare, il processo init provvede ad attivare un processo di interfaccia utente che comincia ad acquisire un comando dopo l’altro per quell’utente e a mandarlo in esecuzione.

Terminazione di un processo
Durante tutta la loro attività i processi svolgono operazioni che realizzano la computazione per risolvere i problemi applicativi, e possono terminare quando viene raggiunta l’ultima istruzione del flusso previsto dall’applicazione. In tal caso, il processo termina con l’esecuzione di una funzione “**exit**”, la quale provvede a restituire un’informazione sullo stato di esecuzione del processo all’interfaccia utente (o in generale al processo chiamante) e a deallocare le risorse utilizzate dal processo.
Un processo può anche terminare in caso si verifichino dei comportamenti anomali: eccessivo uso di risorse, compito non più necessario, oppure perché si è verificato un errore nell’uso di una risorsa.
Ciò può anche avvenire con una serie di terminazioni forzate in cascata quando termina un processo padre, ed è previsto che i suoi processi figli muoiano forzatamente.
Ciò avviene con la funzione di “**abort**”, che porta alla gestione della terminazione con anomalie del comportamento e delle attività dei processi.

Lezione 3 - Sospensione e riattivazione dei processi

Classificazione dei processi
Un processo può usare le varie risorse fisiche del sistema in vari modi.
Si possono avere dei processi legati alle op. di I/O quando questi effettuano tante operazioni di I/O rispetto alle op. di manipolazione di dati all’interno del processore.
Al contrario, i processi saranno legati al processore se eseguono molte operazioni aritmetico-logiche e di spostamento di dati da/verso memoria centrale rispetto alle op. di I/O.
Queste due tipologie di processi descrivono bene le applicazioni che si hanno in esecuzione sul processore.
In alcuni casi le app. possono avere porzioni che tendono a comportarsi come legate all’I/O, e altre porzioni che tendono a comportarsi come legati come processori.
Molto spesso le app. sono ben separate. Eseguono o fortemente operazioni di I/O durante il loro ciclo di vita, o operazioni aritmetico-logiche (eccetto il caricamento dei dati all’inizio e il salvataggio dei dati contenenti i risultati alla fine).
Su queste due tipologie vengono basati gli aspetti di gestione (virtualizzazione) del sistema.
Il multitasking ha l’obiettivo di eseguire più processi sul processore in modo da massimizzarne lo sfruttamento, quindi è per realizzare ciò è necessario gestire la turnazione dei processi sul processore.
Il multitasking funziona nel seguente modo:

- Si sospende il processo in esecuzione
- Si ordinano i processi pronti all’esecuzione (scheduling dei processi)
- Si seleziona tra i processi ordinati quello da mettere in esecuzione (dispatching)
- Scelto e caricato il processo, questo viene riattivato.

Queste operazioni creano l’illusione di uso contemporaneo mediante il multitasking.

Si può definire quando sospendere un processo in esecuzione, o si può definire come ordinare i processi. Per questi aspetti vanno definite delle politiche, delle regole da utilizzare.
In funzione di queste regole si avranno risultati diversi dal punto di vista dell’evoluzione di computazione dei processi sul sistema. Chiaramente il risultato prodotto dai processi non cambierà, piuttosto potrà cambiare l’ordine con cui vengono eseguite le operazioni, cambiando così anche la percezione dell’utente.

Esistono poi dei meccanismi per realizzare ciò:

- Sospensione del processo con il salvataggio del contesto
- Dispatching del processo da mettere in esecuzione (individuato dalla politica di schedulazione)
- Riattivazione del processo

Questi aspetti dovranno essere eseguiti qualsiasi politica venga adottata, valgono sempre.

Politiche di sospensione dei processi nel multi-tasking

Il processo in exec può essere sospeso in modo implicito:
ad esempio quando si eseguono operazioni di I/O, che prevedono tempi lunghi e che rendono inutile tenere il processo in esecuzione, oppure quando viene creato un sottoprocesso di cui bisogna attendere la terminazione, nel cui caso è inutile tenere il padre in esecuzione.
Si può sospendere il processo anche in modo esplicito, quando il programma stesso chiama un'apposita funzione del SO che provca il rilascio del processore, in modo che altri processi possano utilizzarlo.

In tutti i casi la sospensione è sincrona con l'evoluzione della computazione, quindi tutte le volte che si arriva a questi punti il processore viene rilasciato (ad esempio tutte le volte che viene chiamata l'operazione di I/O il processo rilascerà il processore, così se dovrà attendere la terminazione del processo figlio ecc.)

Time sharing

Multitasking a condivisione di tempo, che vuole creare l'illusione dell'evoluzione contemporanea dei processi agli occhi l'utente.
Un processore può eseguire una sola istruzione di un solo processo alla volta, ma ciò può funzionare perché i tempi elettronici del processore sono molto rapidi.
I processi legati al processore CPU-bound tendono a non rilasciare frequentemente il processore, in modo da permettere l'illusione.

La soluzione è quella di forzare il rilascio del processore tramite meccanismi di *pre-emption*.
Per far ciò si introduce il concetto di **time slice** (quanto di tempo), come intervallo max consentito ad un processo di uso del processore. Utilizzato il processore per un tempo "delta" (time slice) il processo dovrà rilasciare anticipatamente il processore (pre-emption). A questo punto il controllo viene ceduto ad un altro processo pronto. Il processo a seguire potrà anche lui tenere il processore solo per il tempo "delta" prima di subire la pre-emption. Un processo che subisce la pre-emption, se necessita potrà rimettersi in coda per lavorare ancora col processore.
Ciò permette di avere un'equa ripartizione della CPU a prescindere da ciò che i processi fanno (I processi CPU-bound verranno interrotti più frequentemente).

La realizzazione di questo meccanismo avviene attraverso un dispositivo nel sistema che permette di generare un'interruzione di real-time clock (RTC) in modo periodico.
Quando scade il tempo previsto all'interno del RTC, ciò genera un'interruzione. Nella risposta all'interruzione associata si potrà provvedere ad accorgersi che il periodo di tempo è scaduto e si potrà decidere se attivare il meccanismo di cambiamento di processo in esecuzione.
Il periodo di RTC è però in genere troppo breve, quindi la routine di risposta all'interruzione verrebbe chiamata troppo frequentemente con un sovraccarico di gestione per il processo.
La soluzione è quella di trovare un periodo di tempo che non sia coincidente al RTC ma sia sufficientemente lungo per dare l'illusione di turnazione frequente con conseguente evoluzione contemporanea della computazione dei processi, pur non sovraccaricando il processore.

Viene quindi preso come quanto di tempo un multiplo del periodo di RTC. La routine di risposta all'interruzione associata a RTC conterà K periodi di tempo di RTC, e al raggiungimento dei K periodi di tempo dichiarerà terminato il quanto di tempo per il processo in esecuzione.

Politiche di sospensione dei processi nel time-sharing
Come nel multitasking si ha la sospensione del processo:
- dopo aver effettuato una richiesta di I/O
- dopo aver creato un sottoprocesso attendendone la terminazione
- quando rilascia volontariamente il processore
- quando scade il quanto di tempo (sopensione asincrona, non c'è relazione tra le operazioni svolte da un processo e il momento in cui casca l'interruzione)

La sospensione sarà implicita nei casi 1, 2 e 4. Nel caso 3 sarà esplicita.

Sospensione del processo in esecuzione
Ciò significa svolgere due operazioni:
- attivare la procedura di sospensione
    - sincrona rispetto alla computazione, in caso di attivazione all'interno dello stato supervisore, ossia si esegue una op. di I/O o creazione di processi.
    - sincrona rispetto alla computazione, in caso di attivazione all'interno dello stato utente (in rilascio volontario)
    - asincrona rispetto alla computazione, allo scadere del time slice nel time sharing

- salvare il contesto di esecuzione (in modo che riparta dove era stata interrotta)
    - salvare tutti i registri del processore sullo stack
    - salvare lo stack pointer nel Process Control Block

Riattivazione del processo
- Ripristino del contesto di esecuzione
    - Ripristinare il valore del registro che punta alla base dello stack dal Process Control Block del processo da riattivare
    - Ripristinare il valore dello stack pointer prendendolo sempre dal Process Control Block
    - Ripristinare tutti i registri del processore prendendoli dallo stack

Cambiamento del processo in esecuzione
- Sospendere il processo in esecuzione
- Riattivare il processo da mettere in esecuzione

Dispatching del processo in esecuzione
- Si tratta di prendere il primo processo in stato di pronto nell'elenco ordinato generato dalla schedulazione dei processi e porlo in esecuzione

UD2 - Thread
Lezione 1 - Thread
Nelle applicazioni che necessitano di alta disponibilità di servizio e basso tempo di risposta alle richieste dall'ambiente esterno è necessario porre cura nella realizzazione, nel bilanciamento e nell'esecuzione delle op. tipiche.
Ad esempio nei sistemi di wordprocessing interattivi avanzati è necessario vedere subito l'effetto completo delle azioni nel sistema, e ciò richiede forte interazione e tempo di risposta molto basso.

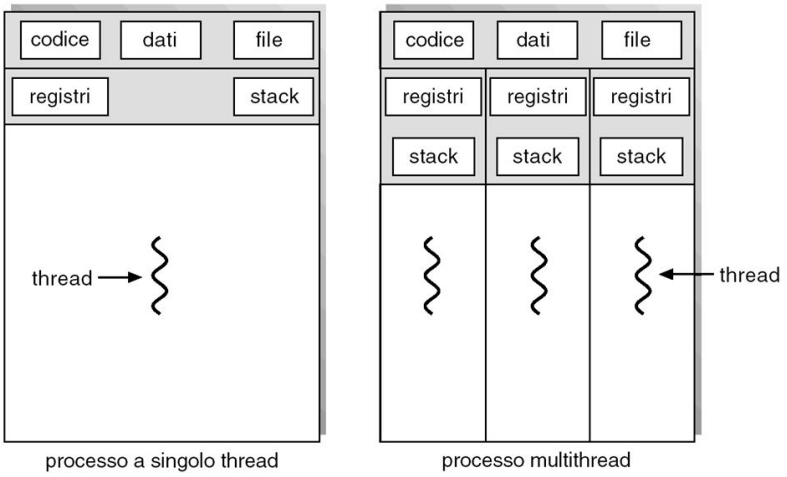
Ancora, se abbiamo un ambiente web, è necessario che il sistema che eroga le pagine del sito web di interesse deve essere capace di rispondere in tempi brevi alle richieste e deve essere anche disponibile (anche con tanti processi che richiedono l'accesso, il sistema deve soddisfare rapidamente ognuno di essi)
Un altro esempio sono i sistemi informativi complessi (magari che devono gestire grosse moli di dati): questi richiedono un'architettura software altamente disponibile e che risponda rapidamente alle richieste degli utenti, e inoltre deve essere scalabile rispetto alla dimensione dell'insieme degli utenti.

Ciò fa nascere alcuni problemi:

- è necessario eseguire più flussi di controllo nello stesso processo per attività simili (ciò renderebbe un server web disponibile a rispondere rapidamente alle richieste degli utenti).
- Bisogna prestare attenzione a come gestire l'attesa delle operazioni di I/O, in quanto ciò potrebbe bloccare delle richieste simili (se il processo sta attendendo il completamento di una operazione potrebbe di I/O non riesce ad ascoltare richieste simili).
- spesso è necessario condividere molte informazioni tra processi, e quindi diventa utile avere la possibilità di condividere memoria centrale come mezzo rapido di scambio di informazioni tra processi cooperanti

In un ambiente a processi si risolverebbe il problema con un processo di servizio (server) + vari processi client che accedono al servizio o cooperano per l'esecuzione dell'applicazione, ma ciò non è efficiente.

Thread
I thread sono flussi di controllo dell'esecuzione di istruzioni di un programma. Sono paralleli e indipendenti dal processo.
Un processo tradizionale (processo pesante) tradizionalmente ha solo un thread, e ha quindi un solo flusso di controllo.
L'introduzione di più flussi di controllo all'interno di un solo processo permette di risolvere i problemi di cui prima.
L'idea è di avere un processo multi-thread con più thread operanti contemporaneamente e condividenti informazioni in memoria centrale.



Quando abbiamo un processo diviso in più thread, ciascuno di essi deve avere un proprio ambiente, un contesto, che deve garantire l'evoluzione indipendente della computazione all'interno di ciascuno di essi, ma essendo nell'ambito di un solo processo i vari thread condivideranno codice, file su cui operare e i dati globali e allocati dinamicamente.
Ogni thread avrà quindi una sua copia dello stack e dei registri.

Benefici
Questi modelli permettono di rendere più flessibile ed efficiente la computazione rispetto al modello del processo pesante.
Si ha una maggiore **prontezza di risposta** alle richieste che pervengono al processo, in quanto ciascun thread, se non occupato, può farsi carico di rispondere alla nuova richiesta.
Ciò permette anche di avere un'alta affidabilità e disponibilità del sistema, in quanto esiste la probabilità che almeno un thread sia libero per rispondere ad una richiesta esterna (solo se non ci sono thread disponibili, le richieste rimangono pendenti per un tempo anche non brevissimo)
L'uso dei thread **permette di condividere risorse e informazioni** in modo semplice, e quindi di aumentare la velocità di accesso alle informazioni e di esecuzione della computazione dei thread stessi
Porta anche ad effettuare una **economia come occupazione in memoria**, in quanto c'è un solo codice, un solo insieme di dati globali, un solo insieme di informazioni relativo alla gestione dei file, e anche un'**economia nell'esecuzione di operazioni d'accesso** alle info condivise, in quanto non è necessario attivare meccanismi complessi per superare la barriera imposta per una corretta evoluzione della computazione sull'accessibilità della memoria centrale da parte di un processo. Ogni processo può accedere solo alla sua porzione di memoria, e ciò rende difficile la condivisione di informazioni e risorse tra processi, ma non tra thread.
Per quanto riguarda i thread, il SO può tollerare di non forzare la separazione degli spazi di memoria indirizzabile dai thread in quanto è sicuro che se si dessero fastidio tra di loro, i vari thread non darebbero mai fastidio ad altri processi. Al più sarebbe l'applicazione stessa ad avere problemi di esecuzione, in quanto potrebbe bloccarsi e non comportarsi correttamente.
Si possono **usare così le architetture multiprocessore in modo efficiente** distribuendo i thread dinamicamente sui vari processori, e migliorando globalmente il tempo di risposta dell'applicazione

Supporto alla gestione dei thread
Il SO mette a disposizione due approcci di supporto:

- Gestione a livello di Kernel: il SO mette a disposizione un insieme di funzioni direttamente gestite ed eseguite a livello di kernel
- Gestione a livello Utente: il SO si fa carico dell'insieme dei thread di un processo globalmente, senza vedere i singoli thread, in quanto il SO si farebbe carico solo del singolo processo. La gestione dei thread diventa responsabilità e carico di un insieme di funzioni apposite eseguite all'interno del processo (quindi gestita in una modalità utente).

Il SO mette a disposizione un insieme di funzioni (sia nello spazio utente che nello spazio kernel) che costituiscono la libreria di gestione dei thread. Nel caso dello spazio utente si tratta di usuali chiamate a sistema, mentre nella gestione a liv. kernel la libreria è un insieme di funzioni, eseguite come chiamate a funzioni locali del processo applicativo.

Lezione 2 - Modelli Multi-Thread

Realizzazione di sistemi multi-thread
Nel caso in cui il sistema operativo abbia il supporto per i soli processi (quindi non per i thread), per realizzare la gestione diventa indispensabile simulare a livello utente, in un processo, l'evoluzione parallela dei vari thread di quel processo.
In altri sistemi esiste il supporto diretto dei thread all'interno del Kernel, in questo caso è il SO che si fa carico di gestire la schedulazione dei thread nel sistema, senza alcuna necessità di simularlo, effettuando solo la gestione dell'intero processo. In questo caso, il SO esegue l'ambiente multi-thread direttamente all'interno del kernel.
Questo può essere effettuato con un'esecuzione diretta di un thread utente su un thread del livello kernel, oppure può essere un gruppo di thread del livello utente simulata su un thread a livello del kernel.

Modello molti a uno
Un modello può essere quello in cui si possono avere molti thread visti a livello utente mappati su un unico thread a livello kernel.
In questo modo il sistema può tenere conto in maniera semplice dell'evoluzione dei thread in cui è suddiviso il processo, semplicemente vedendo l'insieme come un'unica attività suddivisa al suo interno nei vari thread.
Questo porta ad una serializzazione dei vari thread, in quanto quando computa uno bisogna non far computare gli altri. Ciò porta ad una riduzione del parallelismo della evoluzione della computazione dei thread.
Questo porta anche al fatto che in un ambiente multi processore, pur essendo concettualmente separata l'evoluzione dei flussi di computazione dei singoli thread, di fatto risulta serializzata, in quanto il thread di liv. kernel è uno.

Modello uno a uno
Si possono superare questi limiti introducendo a liv. kernel un thread per ognuno dei thread a livello utente. Ciò massimizza la parallelizzazione effettiva di gestione a livello di gestione del SO, permettendo anche un parallelismo fisico in caso di disponibilità di più processori.
Evita inoltre il problema di sospensione dell'esecuzione della computazione dell'applicazione nel caso della gestione di ingressi e uscite.
Il problema però è che l'efficienza può essere diminuita, in quanto la generazione dei thread a livello di kernel comporta un certo overhead (da ciò deriva che in molti SO il numero dei thread di sistema è limitato).

Modello molti a molti
In questo modello un numero N di thread a livello utente viene mappato su un gruppo di M thread a livello kernel con M < N. Questo risolve il problema di parallelismo da una parte, e risolve anche la necessità di tener limitato il numero di thread attivabili in parallelo per consentire una efficiente gestione

Modello a due livelli
In questo modello i thread a livello utente possono raggrupparsi in sottoinsiemi separati e ciascun sottoinsieme potrà essere mappato su un gruppo di thread a livello di SO. (es. un processo di 5 thread può essere suddiviso in un insieme di 4 thread + un altro thread. L'insieme di 4 thread può essere mappato su un sottoinsieme di kernel thread, mentre l'altro singolo thread può essere mappato su un singolo thread a livello kernel). Ciò garantisce una sufficiente ripartizione della capacità di elaborazione parallela, specializzando i gruppi dei thread a livello utente e dando una diversa opportunità di esecuzione, di tempi di risposta, prontezza e efficienza globale, a seconda delle necessità dei singoli thread.

Organizzazione della cooperazione
- thread simmetrici
- thread gerarchici
- thread a pipeline

Thread simmetrici
Si ha un insieme di thread nel sistema, tutti uguali e capaci di risolvere l'applicazione, all'interno del processo. Le richieste che pervengono al processo possono essere trattate da uno qualunque di questi thread, in quanto sono simmetrici ed equivalenti tra loro

Thread gerarchici
Le richieste dal mondo esterno pervengono al processo e vengono acquisite e trattate da un thread coordinatore del gruppo. Questo è l'unico thread abilitato a ricevere richieste dal mondo esterno. Una volta presa in carico, provvederà a comprenderla ed effettuare delle manipolazioni, e quindi ad inviarla ai thread lavoratori del processo. Provvederà ad assegnare ad un thread specifico il lavoro da svolgere in soddisfacimento della richiesta pervenuta.

Thread in pipeline
Le richieste vengono elaborate con il primo thread che prende una richiesta alla volta, svolgendo un'attività di elaborazione di questa richiesta, producendo una risposta parziale che viene passata ad un ulteriore thread del gruppo che farà la stessa cosa e propagherà la richiesta al terzo e così via. I thread sono connessi in cascata, finché l'ultimo non completerà l'elaborazione rilasciando i risultati ai richiedenti.

Lezione 3 - Gestione dei Thread

Creazione dei thread
Quando un processo desidera creare un thread esegue una funzione di fork() analoga a quella per i processi. Questa porta alla duplicazione di tutti i thread del processo, oppure alla duplicazione del solo thread chiamante (dipende dallo specifico SO).

Esecuzione in thread
Quando un thread vuole rivestirsi del codice eseguibile di un programma può invocare la funzione di exec(), in questo caso il programma va a rivestire l'intero processo (il codice è condiviso da tutti i thread, quindi anche gli altri thread vedranno cambiare il codice del programma in esecuzione, quindi cambieranno l'esecuzione).

Eliminazione di un thread
Bisognerà individuare ed eliminare il thread prima che abbia completato le sue attività.
Esistono due modalità:

- cancellazione asincrona: terminazione immediata
- cancellazione differita: il thread verifica in opportuni momenti della sua computazione se può terminare o se deve procedere nella sua attività. Appartiene a questa categoria di cancellazione anche la terminazione ordinaria (al termine della computazione il thread termina e viene rimosso dal sistema)

Sincronizzazione e comunicazione
Queste vengono realizzate secondo le modalità per i processi, utilizzando la memoria condivisa, meccanismi sofisticati ecc.
La comunicazione viene data a tutti i thread di un processo o ad un sottoinsieme. Se ci sono dei processi suddivisi in thread che vogliono comunicare tra di loro, il thread del processo mittente che invoca l'operazione di comunicazione o di sincronizzazione può farlo inviando la comunicazione o la sincronizzazione con tutti i thread del processo destinatario, con un loro sottoinsieme o con un singolo specifico thread.

Processi leggeri
Per gestire meglio le operazioni dei thread nel sistema viene spesso introdotto il "processo leggero" (LWP). Questi sono processori virtuali utili per disaccoppiare la visione dei thread a livello utente dalla gestione dei thread a livello di kernel.
Il LWP si frappone fornendo all'insieme di processi utenti un processore virtuale, i quali verranno mappati a livello del kernel in un modo trasparente (il processo leggero disaccoppia la gestione del parallelismo dei thread a livello utente dal mappaggio e relativa esecuzione a livello di kernel,

facendo sì che il processo a liv. utente non debba curarsi di problemi di gestione di parallelismo, in quanto vengono visti come un unico processore il quale verrà poi replicato opportunamente sui thread del kernel garantendo un'attivazione efficiente a seconda delle condizioni operative dell'intero sistema.

UD3 - Schedulazione

Lezione 1 - Schedulazione

Obiettivo

L'obiettivo della schedulazione è la gestione della turnazione dei processi sul processore.
Questo significa definire le politiche di ordinamento dei processi per gestire tale turnazione.
La schedulazione può essere attivata a:

- breve termine: un tempo al di sotto del secondo
- medio termine
- lungo termine: un tempo dell'ordine di minuti

Intendendo con questi delle durate rispetto all'evoluzione della computazione dei processi.

Schedulazione a breve termine (short term scheduling, o CPU scheduling)

Ha come obiettivo quello di ordinare i processi già presenti in memoria centrale e pronti all'esecuzione (quindi guarda cosa c'è già in MC e che è in stato di pronto).
L'obiettivo è quello di ordinare questi processi in modo che il dispatcher prenderà il primo processo della lista ordinata e lo porrà running, attuando il cambiamento di contesto dal processo attualmente in esecuzione al suddetto processo.
Il processo tolto viene messo tra i processi che verranno successivamente rischedulati per ottenere il processore quando disporrà di tutte le risorse informative e fisiche che gli servono per l'evoluzione della computazione.

Deve essere eseguita frequentemente in modo che i processi vedano evolvere la loro computazione in modo parallelo. Per far ciò è necessario concedere un po' di processore a ciascun processo molto frequentemente nell'unità di tempo.
È quindi essenziale avere una turnazione rapida dei processi.
È tipico avere questa schedulazione ogni 100ms. Ciò significa che per poter verificare e generare l'ordine dei processi l'esecuzione dell'algoritmo di scheduling genera un carico di lavoro minimo, in modo da non sprecare troppo tempo in gestione (per l'exec degli algoritmi di scheduling) anziché in esecuzione dei processi.
Quindi gli algoritmi per la schedulazione a breve termine devono essere molto semplice.

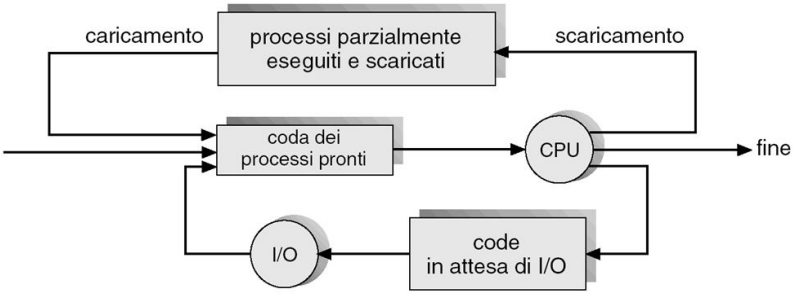
Schedulazione a lungo termine (long-term scheduler o job scheduler)

Il sistema può contenere molti più processi di quelli che possono stare efficientemente in memoria centrale.
Quando viene creato un processo, viene posto nello stato di attivazione, e se abbiamo un numero elevato di processi fare in modo che questi stiano nella MC può portare a dover concedere ad ogni processo uno spazio molto ridotto in MC, e quindi a indurre una difficoltà nell'evoluzione della computazione dei vari processi.
Se questo spazio è piccolo, il tempo che il processo passa a gestire la memoria è virtuale e fa poco in computazione, e questo un problema.
Il job scheduler considera processi che devono essere posti in memoria centrale e nello stato di pronto all'esecuzione.
Quindi quando vengono attivati i processi questi non vengono posti in memoria centrale, ma in un'apposita zona della memoria di massa e lo schedulatore sceglierà tra questi quelli da porre nello stato di ready-to-run nella memoria centrale.

Lo short-scheduler provvederà quindi ad ordinare solo i processi che si troverà posti in MC dal long-term scheduler (dunque opererà su un sottoinsieme dei processi).
Il long-term scheduler creerà il gruppo di processi in memoria centrale mescolando in modo adeguato i processi CPU-bound e i processi I/O bound, in modo da garantire un elevato sfruttamento del processore.

Il job scheduler può essere ridotto al minimo o addirittura essere assente in alcuni sistemi. In ogni caso l'algoritmo utilizzato per l'ordinamento dei processi è complesso, in quanto deve tener conto della predizione del comportamento dei processi e quindi eseguire un bilanciamento ottimale tra le varie esigenze.
Siccome non si deve sovraccaricare troppo il sistema, non è possibile eseguire questo processo troppo frequentemente, tipicamente viene svolto ogni qualche minuto.

Schedulazione a medio termine (medium-term scheduler)
Spesso viene introdotta questa schedulazione per mediare le caratteristiche dei due tipi di schedulazione esaminati (la velocità dello short-term scheduler e la buona capacità di previsione di comportamento dei processi da parte del long-term scheduler).
Al caricamento dei processi scelti dal long term scheduler, le prestazioni possono non essere ottimali (la previsione può non essere eccellente, i processi possono cambiare il loro comportamento, possono essere via via caricati in memoria un numero eccessivo di processi per cui la concorrenza riduce le prestazioni del processore).
L'obiettivo è quello di mantenere limitata la concorrenza dei processi, cercando di ottimizzarne la distribuzione dei processi tra CPU-bound e I/O-bound, in modo da mantenere bilanciato nel tempo tale tipo di distribuzione e massimizzare lo sfruttamento del processore.
Quindi si cerca di massimizzare lo spazio di memoria concesso ai processi che si vogliono gestire con lo scheduler a breve termine per poterne migliorare l'evoluzione e ridurre i tempi di gestione.
La soluzione per il raggiungimento di questi obiettivi è la modifica dinamica del gruppo dei processi caricati in MC.
Quindi lo scopo sarà quello di adattare l'insieme selezionato dal long-term scheduler alle effettive caratteristiche del carico di lavoro rilevate durante l'esecuzione, e quindi a predisporre in MC la distribuzione ottimale di attività di I/O e di CPU.
Il medium-term scheduler quindi ordina i processi selezionati dallo schedulatore a lungo termine per il caricamento in memoria centrale e l'ammissione allo stato di pronto all'esecuzione, mettendo effettivamente in MC solo alcuni dei processi pronti all'esecuzione in modo da garantire l'equilibrio.
Gli altri processi vengono tenuti in un'area di memoria di massa temporanea, in modo che possano essere successivamente presi dallo scheduling a medio termine per l'esecuzione effettiva.



Quando opportuno, i processi possono essere o rimessi nella coda dei processi in MC, oppure scaricati in memoria di massa parzialmente eseguiti, e il medium-term scheduler provvederà successivamente a ricaricarli mettendoli nella coda dei processi pronti.
Il caricamento dalla memoria centrale alla memoria di massa temporanea prende il nome di **swapping-out**.
La reimmissione in memoria centrale dalla memoria di massa prende il nome di **swapping-in**.

Attivazione della schedulazione
La schedulazione può essere attivata in diversi modi:

- Senza rilascio anticipato (pre-rilascio, rilascio forzato), schema del “non pre-emptive scheduling”: In questo caso il processo viene cambiato quando richiede una operazione di IO, quando crea un processo e ne attende la terminazione, quando rilascia volontariamente il processore e quando termina.
  In questo caso la schedulazione viene attivata sempre in modo sincrono con la computazione del processo.

- Con rilascio anticipato, schema del “pre-emptive scheduling”: in questo caso la schedulazione viene eseguita in modo asincrono con l’evoluzione della computazione del processo, in quanto il processo in esecuzione può essere cambiato anche quando scade il quanto di tempo (sistemi time-sharing).

Lezione 2 - Criteri di schedulazione

Criteri di schedulazione
Per valutare la bontà di una politica di schedulazione si possono adottare diversi criteri, che mettono in evidenza aspetti diversi di uso del processore e di evoluzione della computazione dei processi per sé e dal punto di vista degli utenti che osservano l’evoluzione

I criteri sono:

- uso del processore: la percentuale di tempo per cui il processore esegue computazione utile per i processi anziché dedicare tempo ad attività di gestione o rimanere in attesa di svolgere una qualunque computazione.
- Capacità o frequenza di completamento (throughput): il numero di processi completati nell’unità di tempo. Ci dice quanto il sistema è in grado di completare le singole attività richieste dagli utenti.
- Tempo di completamento (turnaround time): ci dice entro quanto tempo le operazioni vengono completate, tenendo conto del tempo in cui il processore non è assegnato al processo in questione perché assegnato ad un altro processo o alla gestione del sistema
- Tempo d’attesa: ci dice quanto tempo un processo rimane in attesa, e specularmente quanto effettivamente utilizza il processore.
- Tempo di risposta: capire quanto velocemente il processo riesce ad ottenere il processore per rispondere alle richieste dell’utente. Cioè misura entro quanto tempo il processore riesce ad iniziare a elaborare le richieste dell’utente

Ottimizzazione nella schedulazione
Ottimizzare il processo di schedulazione significa cercare di perseguire i seguenti obiettivi:

- Massimizzare l’utilizzo del processore
- Massimizzare il throughput
- Minimizzare il tempo di completamento (in modo che terminino rapidamente)
- Minimizzare il tempo d’attesa (in modo che non stiano ad aspettare troppo)
- Minimizzare il tempo di risposta

Comunque, quello che si va a massimizzare tramite queste cifre di merito è il valor medio (ad esempio l’utilizzo del processore può essere al 75%, ma questo potrebbe in realtà oscillare tra un valore di 50% e un valore di 100%, dunque la varianza è molto elevata rispetto al valor medio statistico, il che rende il dato poco significativo).

Dunque il miglioramento dovrebbe mirare a minimizzare la varianza di questi parametri, in modo da dare un buon significato al valor medio per l'ambito di ottimizzazione, e ottenere quindi una buona prevedibilità del comportamento del sistema.

Valutazione delle cifre di merito
Vi sono diversi modi:

- Modo analitico: andando a creare una descrizione formale e matematica con delle formule analitiche del comportamento dell'algoritmo, e quindi calcolare il valore desiderato in funzione della distribuzione dei carichi di lavoro correnti (modellazione deterministica)
- Modo statistico
- Realizzazione del sistema e valutazione effettiva dei parametri

Modellazione deterministica
Nella modellazione deterministica vengono costruite le formule che descrivono il comportamento dell'algoritmo di schedulazione e quindi si calcola il valore del parametro caratteristico semplicemente sostituendo la distribuzione dei carichi di lavoro come ingressi della formula ottenuta.
L'approccio è semplice, veloce una volta costruita la formula ed è precisio.
Però richiede dei valori esatti per i carichi di lavoro, e questo non è sempre facile o possibile.
Inoltre i risultati possono essere difficilmente generalizzati, e questo comporta una scarsa capacità di astrazione e di comprensione di come il sistema potrà funzionare a partire solo da una modellazione di tipo analitico.

Modellazione statistica
La modellazione statistica permette di rimediare ai problemi della modellazione deterministica.
Si può attuare in diversi modi:

Modellazione a reti di code
In questo modello il sistema è descritto come un insieme di servizi, i quali dispongono ciascuno di una sua coda d'attesa, le richieste di uso del servizio vengono inserite in questa coda.
Quando il servizio diventa libero viene presa una richiesta dalla coda, viene lavorata, viene servita, e il sistema la rilascia successivamente affinché il processo che la rappresenta possa procedere nella sua attività.
Quindi, nella sua vita, un processo effettua una sequenza di richieste di servizio.
Questo per esempio passa attraverso l'uso del processore, l'uso delle periferiche, e quindi andrà ad attendere nella coda del processore che questo diventi libero. Quando lo ottiene e ha bisogno di acquisire dei dati dal disco, terminata la sua elaborazione, entra nella coda di attesa del disco. Quando i dati dal disco sono stati prelevati e resi disponibili in memoria centrale per l'elaborazione, il processo uscirà e tornerà in uno stato di attesa del processore per completare le operazioni sui dati ottenuti. Alternativamente potrebbe dover mettersi in attesa per i dati in arrivo da tastiera, e in tal caso si immetterà nella coda relativa e il procedimento è uguale. Quando il processo terminerà, uscirà dal sistema e non utilizzerà più le risorse dello stesso.
Quindi il sistema di elaborazione può essere rappresentato come una rete di code. Ogni coda rappresenta un servizio del sistema (composto da servitore e coda di attesa dell'utilizzo del servizio. Le transizioni rappresentano i flussi delle richieste di servizio).

Analizzare le reti di code vuol dire, in particolare, specificare la topologia del grafo della rete, specificare le caratteristiche di ogni servizio con la frequenza di arrivo delle richieste, il tempo di servizio (in modo statistico, con media e varianza, o media e distribuzione). Vuol dire poi utilizzare delle tecniche matematiche della teoria delle code per effettuare l'analisi delle caratteristiche di questo grafo.

Tale teoria porta a valutare qual è l'utilizzo delle risorse, la lunghezza media della coda e il tempo medio d'attesa, ed eventualmente altri parametri caratteristici dei singoli componenti del modello Serve chiaramente attuare delle semplificazioni per realizzare questo modello a code e a risolvere con le tecniche matematiche della teoria delle code il modello così costruito.

Simulazione
La simulazione della realtà è un altro modo per modellare in modo statistico il comportamento del sistema. Si può realizzare un modello software del sistema reale che si vuole realizzare, includendo lo schedulatore, i processi applicativi e tutto, in modo da simularne l'esecuzione.
Questo vuol dire andare ad identificare un insieme di dati significativi che caratterizzano le attività dei processi, e quindi applicarli all'interno della simulazione. Vuol dire poter realizzare la computazione dei processi all'interno del simulatore.
In questo ambiente sintetico si potranno misurare le caratteristiche della schedulazione e lo si può fare assumendo un comportamento statistico del sistema, variando la distribuzione dei dati su cui i vari processi operano.

Implementazione reale
Se si vuole avere una valutazione accurata spesso un'analisi di tipo statistico non basta, e quindi si deve andare a realizzare il sistema.
Quindi ciò che si fa in questa metodologia consiste proprio nel realizzare il sistema, lato hw e sw, SO e algoritmo di schedulazione incluso, con gli ingressi reali.
Quindi si misura veramente le caratteristiche desiderate nel sistema di schedulazione. Questo è oneroso, e richiede la cooperazione degli utenti che sono coinvolti in questa misura.
La schedulazione viene quindi scelta in base alle caratteristiche reali, e queste caratteristiche per i carichi di lavoro possono essere poi raccolte in modo automatico da parte dei componenti del SO.

Questo è importante perché permette di avere un meccanismo di raffinamento della schedulazione durante la vita operativa del sistema. Durante tale vita si possono raccogliere info sui carichi di lavoro in maniera automatica, e quindi si può fare un'analisi successiva e quindi una definizione più precisa di alcuni parametri costitutivi dell'algoritmo di schedulazione per avere quindi un adattamento dinamico dello schedulatore al cambiare delle caratteristiche dei carichi di lavoro del sistema di elaborazione.

Lezione 3 - Politiche di Schedulazione

Regole di schedulazione
Le regole che possono essere usate per definire l'ordine con cui i processi pronti all'eccezione vengono ordinati in modo da essere eseguiti sul processore quando questo si libera.

FCFS (First Come, First Served)
Si tratta della politica più semplice. In questo caso si ha una coda in cui i processi pronti all'esecuzione vengono inseriti e dalla quale vengono estratti man mano che il processore si libera.
Questa politica è non-preemptive, cioè non porta alla sospensione del processo in esecuzione per l'attivazione dell'algoritmo di schedulazione. Solo quando il processo in esecuzione rilascia il processore, il primo processo nella coda di attesa entra per usare il processore.
Il problema è che, in questo caso, i processi CPU-bound possono utilizzare il processore per molto tempo, e quindi tendere a monopolizzarlo. Un altro aspetto critico è il tempo di attesa, che può diventare alto a seconda del momento in cui entrano nella coda (se viene eseguito per primo un processo che occupa il processore a lungo, e a seguire altri processi, il fatto che gli altri processi debbano attendere a lungo per utilizzare il processore fa aumentare di molto il tempo di attesa medio).

SJS (Shortest Job First)
L'obiettivo di questa politica è di far eseguire per primo il processo più breve, in modo da ridurre globalmente in tempo di attesa.
Questo algoritmo può essere realizzato in due modi:

- pre-emptive: quando un processo diventa pronto viene interrotto il processo in esecuzione e l'attivazione dell'algoritmo di schedulazione
- non pre-emptive: il processo che diventa pronto non interrompe il processo in esecuzione. Invece lo lascia completare e solo successivamente avviene la schedulazione

Globalmente, in tempo medio d'attesa può decrescere con la modalità pre-emptive.
Questo algoritmo garantisce il minimo tempo di attesa se si conosce a priori il tempo richiesto da ciascuno dei processi. Questo è difficile da sapere esattamente, e quindi ci si basa su tecniche di predizione, come stime di tipo statistico basata sul tipo di uso che ha fatto il processo in precedenza.
Una prima soluzione è quella di utilizzare come tempo di uso del processore un tempo medio rispetto agli usi precedenti. Altrimenti si può effettuare un'analisi tramite media esponenziale, in cui il tempo predetto all'istante corrente è uguale ai tempi che si avevano negli istanti precedenti opportunamente pesati + il tempo corrente.

Priorità
Questo approccio ha l'obiettivo di mettere in evidenza l'importanza relativa tra i vari processi, piuttosto che il loro tempo di arrivo.
Esistono due rappresentazioni dell'indice di priorità legato ai processi:

- Logica diretta: l'indice di priorità più alto identifica una priorità più alta
- Logica inversa: l'indice di priorità più basso identifica una priorità più alta

Questo algoritmo può essere utilizzato in due modi:

- Pre-emptive: il processo che diventa pronto interrompe il processo in esecuzione richiedendo la schedulazione.
- Non pre-emptive

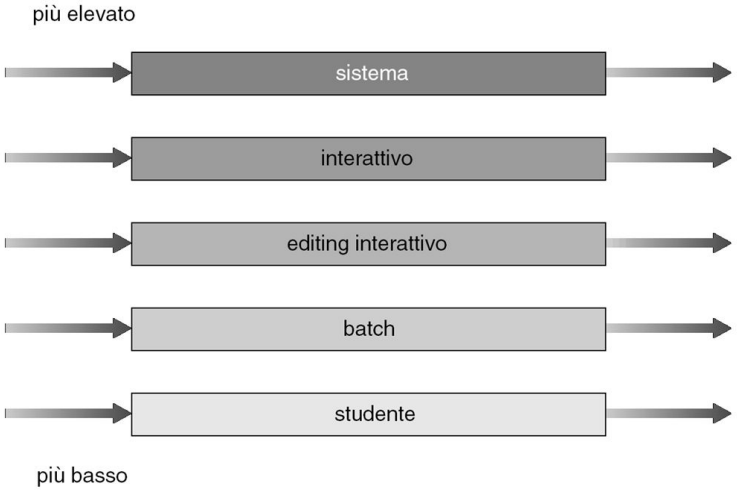
Il problema che si verifica con questa gestione è che i processi a bassa priorità potrebbero rimanere bloccati indefinitamente (starvation), in quanto sperano di ottenere la CPU ma processi di priorità più alta continuano ad impossessarsene.
La soluzione può essere di avere un adattamento dinamico delle priorità (aging). Man mano che il processo a priorità bassa rimane nella coda di attesa per ottenere il processore, i processi di priorità più alta vedranno ridursi la loro priorità, finché il processo di bassa priorità in stato di starvation arriva allo stesso livello di priorità degli altri, riuscendo così a competere ad armi pari per l'utilizzo del processore. Alternativamente si può far sì che la priorità di questo processo aumenti fino a diventare competitiva con i processi di priorità elevata.
All'ottenimento del processore, il processo deve tornare a competere con i rapporti di rilevanza iniziali, e quindi le priorità modificate dal meccanismo di aging vengono riportate ai valori iniziali (progressivamente o immediatamente).

RR (Round Robin)
Il round robin consiste nella rotazione dei processi sul processore.
Al rilascio del processore da parte del processo che lo teneva occupato (sia per la pre-emption causata dalla scadenza del tempo per il time sharing, sia perché deve attendere una risorsa da io, ecc.) la politica andrà a cercare il primo processo ready to run a rotazione.
Ogni processo saltato dovrà attendere che finisca la rotazione dei processi.

Questo algoritmo è tipico dei sistemi time-sharing, ed è molto simile a FCFS con l'aggiunta però della pre-emption.
L'algoritmo di RR da una distribuzione uniforme del tempo di elaborazione tra i processi pronti, e l'apparenza di velocità che questi processi hanno dipenderà dal numero di processi pronti (più sono, e minore sarà la velocità percepita).
Il tempo di turnaround dipenderà dalla durata del quanto di tempo.
Il comportamento dell'algoritmo RR dipende dal quanto tempo:

- se il quanto è molto lungo RR tende a diventare FCFS
- se il quanto è molto breve tendere a generare una forte condivisione del processore per N processi, con ogni processo che vede di fatto 1/N di capacità computazionale ad esso assegnata. Ciò porta a frequenti cambiamenti di contesto, e quindi a sovraccarico computazionale.
- L'ideale empirico sarebbe che l'80% delle richieste di elaborazione sia completata in quanto di tempo

C+L (Coda a più livelli)
In questo caso i processi sono raggruppati per tipologie omogenee, e ogni tipologia è assegnata in a uno specifico livello di schedulazione, rappresentata da una coda di attesa specifica per l'uso del processore.
Per ogni coda è possibile introdurre un algoritmo specifico di schedulazione. L'insieme delle code verrà schedulate da un algoritmo dedicato. Genericamente vengono utilizzate delle schedulazioni a priorità fisse.



Così vengono categorizzati i processi in base alle necessità (ad esempio un sistema processo interattivo avrà bisogno di risposte rapide, e quindi priorità più alta rispetto a processi batch).

C + LR (coda a più livelli con retroazione)
In questo caso i vari processi possono migrare da un livello all'altro attivando un meccanismo di promozione o degradazione.
Si potrà avere una coda di priorità più elevata in cui il quanto di tempo sarà molto breve per garantire una rapida turnazione dei processi, e si potrà man mano allungare il tempo tra le varie code per garantire una rotazione più lenta, un minor uso di tempo per la gestione, per processi meno rilevanti, i quali evolveranno più lentamente.
Si hanno quindi code di attesa separate in funzione dell'uso dinamico che si vuol far fare ai processi del processore. Si ha ancora un algoritmo specifico di schedulazione per ogni coda di attesa, e in più si avranno delle politiche di allocazione dei processi nella coda opportuna, e di promozione o degradazione dei processi per spostarli tra i vari livelli.

Schedulazione in sistemi multiprocessore

La schedulazione, in questo caso, deve considerare le caratteristiche specifiche dell'architettura del sistema di elaborazione:

- I processori potrebbero essere omogenei o eterogenei (uguali, o diversi)
- Il fatto che la memoria possa essere solo condivisa, oppure locale dei processi.
- Il fatto che le periferiche possano essere ugualmente accessibili da tutti i processori oppure che ci siano singoli processori di ingresso e uscita dedicati a gestirle (o a gestirne dei sottoinsiemi)

In questo caso si possono pensare delle politiche di schedulazione specifiche a seconda della combinazione di queste caratteristiche hw
Ad esempio:

- se si ha un sistema con processori omogenei, memoria solo condivisa e periferiche accessibili a tutti i processori si potrà avere una coda unica dei processi oppure una coda per ogni processore lasciando questa coda nella MC condivisa.
- Se si ha anche una memoria locale, la coda specifica può stare sia nella MC condivisa oppure può essere messa, per velocizzare le operazioni, nella memoria locale
- Se si hanno ancora processori omogenei, MC condivisa e periferiche accessibili solo da processori specializzati, ci sarà la necessità di introdurre una coda per ogni processore che gestisce la specifica periferica di interesse e una coda per processori omogenei in MC.
- Se si hanno processori omogenei, memoria anche locali e periferiche accessibili da alcuni processori la coda per la gestione dei processori (che seppur omogenei gestiscono una specifica periferica) potrà essere messa nella memoria locale
- Se si hanno processori eterogenei si avrà una coda per ogni processore oppure una coda per gruppi di processori omogenei nell'insieme

Si possono pensare, nel caso di sistemi multiprocessore diversi tipi di processamento:

- Multiprocessamento asimmetrico: quando si ha un processore master che esegue il sistema operativo, e quindi provvede a gestire la schedulazione di tutti i processi, mentre gli altri processori si occupano solamente di eseguire i processi applicativi, e si vedono assegnati i processi
- Multiprocessamento simmetrico: quando ogni processore esegue il sistema operativo, e quindi ogni processore provvede a gestire la schedulazione.

Lezione 4 - Schedulazione per sistemi in tempo reale

Sistemi in tempo reale stretto (hard real-time system)
Sono i sistemi in cui non solo è necessario garantire la turnazione dei processi sul processore, ma è obbligatorio che un processo termini la computazione entro un tempo massimo garantito dalla sua attivazione.
Il problema di fondo è che, nel momento in cui si verifica un evento in un certo istante, il sistema deve attivare una computazione relativa a questo evento in modo che questa sia completata entro una scadenza massima dal sollevamento dell'evento. Se la computazione si prorogasse oltre il termine previsto, il sistema potrebbe avere seri problemi e non essere più rispettata la corretta reazione agli eventi esterni.
Per gestire la schedulazione vi sono diversi approcci:

Tempo massimo di completamento garantito
Questo approccio prevede che lo schedulatore accetti un processo garantendone il completamento entro il tempo massimo consentito, oppure rifiuti il processo.
L'accettazione del processo è basata sul fatto che si riescano a stimare i tempi di completamento del processo e si riesca a garantire che, dato il carico di lavoro attualmente già accettato dal

sistema, l'aggiunta di questo processo non comporti problemi per l'esecuzione dei processi già accettati e del nuovo processo.
L'accettazione si basa quindi sul fatto che si riescano ad ottenere tutte le risorse necessarie per le operazioni del processo entro il tempo massimo consentito.
Le politiche di schedulazione che si possono adottare sono quelle tradizionali, non esiste nulla di specifico.
L'unico aspetto essenziale è che si effettui un'analisi a priori che porti all'accettazione o al rifiuto del nuovo processo che si cerca di attivare nel sistema.
Il problema è la predicibilità del tempo di completamento: se è effettuabile in modo sicuro allora si può applicare la valutazione di accettazione, se invece è incerta non abbiamo la certezza di poter effettuare una valutazione a priori corretta.

Processi periodici
Molti processi sono di tipo periodico, e cioè eseguiti periodicamente, a intervalli regolari.
Il processo periodico ha:

- un tempo fisso di elaborazione t_i
- una scadenza d_i
- una perioditicà p_i

Tipicamente questi tempi vengono ordinati in maniera crescente, ovvero all'interno del periodo si fissa la scadenza massima, e il tempo di elaborazione dovrà essere minore della scadenza:
$0 < t_i < d_i < p_i$

La scadenza viene messa minore del periodo per garantire che ci sia una facile gestibilità del processo e non si rischi di avere in alcuni casi critici una non capacità di gestione perché nello stesso periodo sono eseguite due istanze del processo periodico.
Nel caso di processi periodici si possono utilizzare politiche di schedulazione un RR o una priorità assegnata in base alla scadenza o alla frequenza $1/p_i$ di attivazione del processo.
Anche nel caso dei processi periodici, si effettua un controllo sull'ammissibilità del processo, e se il processo viene visto in grado di completare la sua computazione entro la scadenza dichiarata utilizzando la politica di schedulazione adottata dal sistema allora viene ammesso, altrimenti viene rifiutato.

Schedulazione a frequenza monotona
Si tratta di un algoritmo per processi periodici con priorità e pre-emption. In questo sistema il tempo di elaborazione è omogeneo per ogni iterazione del processo.
La priorità viene assegnata usando tecniche di tipo statico o proporzionale alla frequenza $1/p_i$ con cui il processo viene attivato.
Questa tecnica di schedulazione prevede la pre-emption: se arriva un processo di priorità più alta di quello attualmente in esecuzione, questo viene interrotto, tolto dal processore, e il processo di priorità più alta potrà utilizzare il processore.

Schedulazione a scadenza più urgente (EDF)
Si tratta di un'altra tecnica di schedulazione attuabile per i sistemi in tempo reale sia per processi periodici che non periodici, in cui il tempo di elaborazione dei processi non deve essere più necessariamente omogeneo.
Questa tecnica prevede che si assegnino delle priorità ai processi in modo inversamente proporzionale alla loro scadenza.
La priorità viene assegnata in modo dinamico, cioè man mano che arrivano dei processi pronti si rivalutano le priorità, in modo da far galleggiare a priorità più elevata i processi per cui diventa più urgente la scadenza.

Sostanzialmente la tecnica cerca di completare prima la computazione dei processi che vedono avvicinarsi la scadenza più rapidamente, in modo da garantirne il rispetto.

Sistemi in tempo reale lasco
Si tratta di sistemi i cui processi non sono tutti critici o prioritari. Ci sono alcuni processi critici per cui diventa importante avere una rapida gestione, una rapida risposta da questi, e un'insieme di processi non critici.
Si può quindi utilizzare una schedulazione a priorità separando i processi critici dai processi non critici.
Si può utilizzare una priorità statica per i processi critici, più elevata di quella dei processi non critici, e per questi ultimi si può considerare eventualmente delle tecniche dinamiche che consentano, attraverso dei meccanismi di aging, di far ottenere (prima o poi) l'uso del processore senza lasciare questi processi non critici in una situazione di starvation.
La cosa importante in questi sistemi è di avere una bassa latenza nelle operazioni di dispatching, e quindi si avrà bisogno di introdurre una interromperrombilità con dei punti specifici in cui effettuare la pre-emption, o addirittura garantire che l'intero Kernel sia interrompibile, in modo che il così il caricamento dei processi sia rapido.

Lezione 5 - Schedulazione dei thread

Livelli di schedulazione
Può avvenire su due livelli diversi:

- livello di processo: se il sistema non supporta in modo diretto la gestione dei thread, ma mette a disposizione delle librerie per gestire i thread all'interno del processo, la gestione sarà affidata al processo, mentre il sistema operativo vedrà schedulato il processo e non i singoli thread
- livello di sistema: se il sistema supporta in modo nativo i thread allora potremo avere la gestione della schedulazione direttamente effettuata a livello di SO

Schedulazione a livello di processo (thread user-level scheduling)
Questo tipo di schedulazione prevede una visione a livello di contesa delle risorse all'interno del processo. Questo è gestito dalla libreria di schedulazione, fornita all'interno della libreria dei thread. Lo scheduler dei thread è quindi interno a ciascuno dei processi. Il SO schedulerà globalmente in processi.
Tipicamente le schedulazioni effettuate nelle librerie dei thread sono:

- a priorità fisse o modificabili dal programmatore, e usualmente hanno un meccanismo di pre-emption
- FCFS
- Round Robin

Schedulazione a livello di sistema (thread kernel-level scheduling, oppure thread user-level mappati su thread kernel-level (eventualmente con Light Weight Processes))
Si ha direttamente nel SO la possibilità di vedere i processi divisi in thread secondo quanto stabilito dal programmatore, ma con il SO che gestirà direttamente la schedulazione di ciascun thread, avendo a bordo uno scheduler in grado di gestire sia i processi che i thread nei processi.
Si ha quindi una visione della contesa del processore a livello di sistema, gestita direttamente dallo scheduler del SO.

UD4 - Comunicazione tra processi
Lezione 1 - Processi Cooperanti

Coordinamento

Un insieme di processi all'interno del sistema di elaborazione può operare in modo coordinato, in modo che le varie attività avvengano rispettando certi ordinamenti nel tempo.
A tal fine deve essere attivata una sincronizzazione della computazione dei processi, che garantisca l'accesso a risorse condivise, e/o che assicuri l'evoluzione congiunta dei processi per ottenere uno scopo applicativo comune.
La collaborazione tra processi può arrivare ad essere addirittura una cooperazione. In questo caso il gruppo di processi opera coordinatamente insieme per il raggiungimento di uno scopo applicativo comune condividendo e scambiandosi info.

Processi Indipendenti

Si tratta di processi che non hanno scopi comuni con gli altri processi, non influenzano e non sono influenzati da altri processi, ma in modo mirato tendono a risolvere una specifica applicazione.
Non condividono alcuna risorsa con gli altri processi. L'unica è la competizione per l'uso della CPU ed eventualmente per l'accesso a periferiche condivise.
In questo caso il coordinamento della computazione avrà come unico obiettivo quello di sincronizzare questi processi perché usino le risorse condivise in modo consistente, in particolare quando le periferiche possono essere usate coerentemente se usate in mutua esclusione.

Processi Cooperanti

Si tratta di gruppi di processi che operano congiuntamente. Questi processi hanno un unico scopo comune. Possono condividere informazioni e possono influenzarsi l'un l'altro. In questo caso si avrà uno scambio di info tra processi cooperanti, e questo realizza la **comunicazione** tra processi.
Avremo anche eventualmente la coordinazione della computazione di questi processi secondo i modelli della **sincronizzazione** della computazione.

Vantaggi della cooperazione

La cooperazione di processi ha dei vantaggi rispetto alla computazione con un singolo processo: permette di scomporre il problema applicativo in parti e affrontare ciascuna parte separatamente con un processo mirato.
Questo consente di avere una buona modularità del sistema e quindi di poter aggiungere e togliere componenti man mano che diventa necessario.

Consente anche di avere una parallelizzazione dell'evoluzione della computazione, in quanto avendo processi separati, in un sistema multitasking, è possibile vedere l'evoluzione dell'attività in modo coordinato e concorrente sul processore.
Addirittura nei sistemi distribuiti o a multi-processore, l'allocazione dei processi sui vari processori permette di avere fisicamente una evoluzione parallela di ogni processo sui vari processori su cui sono allocati, e quindi una forte accelerazione della computazione, e tempi di risposta brevi.

Vedere la cooperazione come un insieme di processi interagenti permette di avere una scalabilità nella risoluzione del problema: se un processo deve gestire una applicazione che deve servire richieste dagli utenti, in una gestione con processo singolo si ha una serializzazione delle gestioni delle applicazioni, e questo aumenta i tempi di risposta, avere un processo pensato in modo modulare permette di scalare facilmente la soluzione del problema attivando tanti processi cooperanti quanti ne servono per risolvere l'intero insieme di richieste pervenute all'applicazione.
D'altra parte, dal punto di vista dell'ingegnerizzazione del sistema, avere un sistema pensato come un gruppo di processi cooperanti permette di specializzare la realizzazione di ciascun modulo, di

assegnare la realizzazione a gruppi specializzati, e di ottenere quindi una migliore qualità del progetto e della sua realizzazione.
Esempi di processi cooperanti sono i classici processi produttore-consumatore, client-server, catena compilatore-linker-caricatore.

Lezione 2 - Comunicazione tra processi

Comunicazione

La comunicazione tra processi è l'insieme delle politiche e dei meccanismi che permettono a due processi di scambiare informazioni in modo da poter realizzare la cooperazione.
L'invio e la ricezione di informazioni tra i processi consente di coordinare l'evoluzione della computazione, e di fornire quelle informazioni sui risultati della computazione sui quali si baserà l'evoluzione della computazione successiva del processo ricevente.
Si ha la necessità di effettuare la comunicazione quando si vogliono trasferire informazioni tra due processi o quando si vuole condividere delle informazioni con due o più processi.

Entità coinvolte nella comunicazione

Le entità coinvolte saranno:

- Processo produttore dell'informazione
- L'informazione che viene trasferita
- Processo ricevente e utilizzatore dell'informazione
- Canale di comunicazione (insieme delle attività di trasferimento delle informazioni dal mittente al ricevente e dei meccanismi coinvolti)
  Questo può essere monodirezionale o bidirezionale (se il sistema supporta solo canali monodirezionali si useranno due di questi canali per effettuare scambi bidirezionali)

Caratteristiche delle comunicazioni

- *Quantità di info da trasmettere*: questo definisce quale tipo di canale di comunicazione è meglio utilizzare
- *Velocità di esecuzione*: definisce come devono essere complesse le operazioni di comunicazione in modo da rendere rapida l'esecuzione e il trasferimento di info. Anche questo coinvolge il tipo di canale di comunicazione da utilizzare
- *Scalabilità*: utile per determinare il numero di comunicazioni (e quindi di processi) che si vogliono gestire in comunicazione in modo da garantire che il sistema mantenga delle prestazioni sufficientemente elevate.
- *Usabilità delle procedure di comunicazione*: caratteristica importante, in quanto garantisce la scrittura di programmi di qualità in modo rapido
- *Omogeneità delle comunicazioni*: possibilità di cui tener conto, in modo che sia semplice per il programmatore capire come ottenerle e realizzarle correttamente.
- *Integrazione delle primitive di comunicazione nel linguaggio di programmazione*: ciò permette di rendere nativo l'utilizzo di tali primitive quando le comunicazioni sono un aspetto fondamentale delle attività dei processi cooperanti
- *Affidabilità delle comunicazioni*: garantire che le comunicazioni vengano completate correttamente e che venga segnalato quando non avviene ciò
- *Sicurezza e protezione*: assicurazione il livello di privatezza delle informazioni scambiate

Implementazione

Esistono varie tecniche:

- ***Condivisione della memoria tra processi*** (il processo che vuole condividere l'info può accedere in scrittura, mentre il processo che ne deve usufruire può leggere, possibilmente nello spazio di indirizzamento)

- ***Messaggi***: i processi comunicano tra di loro tramite funzioni di sistema operativo e spazi di memoria posti all'interno del sistema operativo (i buffer). In questo caso, il processo mittente depone il messaggio nel buffer e il processo ricevente lo estrae dallo stesso buffer.

Queste due tecniche richiedono l'identificazione univoca e completa del processo che produce le info e del processo che le riceve. Sono quindi tecniche di comunicazione diretta.
A volte può essere utile però avere un disaccoppiamento tra mittente e ricevente, in modo da poter gestire multipli mittenti per lo stesso ricevente e più riceventi per uno stesso messaggio inviato da un mittente, o in altri casi, la possibilità di riattivare un processo coinvolto nella comunicazione a causa del fatto che un errore ha portato alla sua terminazione anticipata.
In questi casi è più utile avere dei meccanismi di comunicazione indiretta in modo da garantire il fatto che non si conosce a priori il processo destinatario, ma si ha una struttura dati dove vengono depositate le informazioni da scambiare:

- ***Mailbox***: processo mittente e ricevente si scambiano le informazioni attraverso il SO depositando i messaggi in una struttura apposita chiamata mailbox. Il processo ricevente acquisisce il messaggio desiderato prendendolo da questa struttura, così che il mittente non sia più obbligato a conosce il ricevente né viceversa, ma che sia sufficiente conoscere questa struttura dati nel SO.
- ***File e pipe***: permettono di estendere il concetto di memoria condivisa da MC a memoria di massa, e quindi avere la realizzazione della comunicazione di due processi attraverso i supporti del SO ponendo le info in un file su disco. Il destinatario estrarrà le info dal file.
  La variante pipe prevede che la comunicazione avvenga buffer ordinati sequenzialmente posti in MC del SO. Il processo mittente accoda nella pipe i messaggi secondo schema FIFO e il destinatario li estrae dalla testa della pipe
- ***Socket***: generalizzazione della tecnica delle pipe per funzionare sui sistemi distribuiti. I processi potranno risiedere su due macchine diverse (anche con SO diversi). Il mittente depositerà il messaggio in un estremo della pipe e il processo destinatario lo estrae dall'altro estremo posto sull'altra macchina. La comunicazione tra i due estremi viene garantita dalla rete, che prende in modo trasparente i messaggi di un estremo e li pone nell'altro. Questo sistema funziona sia su sistemi distribuiti che sulla stessa macchina, in questo caso i due tronconi risiederanno sulla stessa macchina.

Lezione 3 - Comunicazione con memoria condivisa

Condivisione di variabili globali

Un processo che vorrà inviare dei dati ad un altro disporrà di una zona nel suo spazio di indirizzamento che identificherà i dati condivisi con il processo destinatario (diversa da Heap, Stack, Dati Globali non condivisi e codice).
Dunque, quando il processo mittente effettuerà una scrittura in questa zona di memoria, questa, essendo condivisa, è come se fosse una memoria nello spazio di indirizzamento del processo destinatario.
Quindi, il processo destinatario vedrà questa condivisione come se fosse la sua memoria locale, quindi leggerà direttamente le informazioni da tale memoria.
È quindi come se si avesse una zona di memoria in comune tra i due processi, nella quale scrivere le variabili che si desidera condividere.

Problemi della condivisione di variabili globali

- Innanzitutto bisogna identificare i processi comunicanti, in quanto la comunicazione è diretta (il processo mittente deve sapere chi è il ricevente e viceversa, in quanto ogni processo dovrà trovare la porzione di memoria dove andare a scrivere o leggere)

- Le operazioni devono essere eseguite in modo consistente: lettura e scrittura sono incompatibili tra di loro, e quindi vanno eseguite in mutua esclusione. Questo serve per garantire la consistenza delle informazioni che vengono scritte nella memoria.
- Quindi è necessario introdurre delle tecniche di sincronizzazione per l'accesso mutuamente esclusivo alla memoria

Realizzazione con area comune copiata dal SO
Il SO garantirà le copiature dei valori tra le due zone dei relativi spazi di indirizzamento dei processi. Il SO utilizzerà un'altra zona di memoria di grandezza sufficiente per contenere una copia della memoria da condividere tra i processi.
Quando il processo mittente deve scrivere nella memoria condivisa, compete per ottenere l'accesso mutuamente esclusivo ed esegue una scrittura nel suo spazio locale di memoria.
Al termine, rilascia la memoria in modo che il SO possa prendere i valori nella zona di memoria e copiarli nello spazio riservato al suo interno.
Quando il processo destinatario richiederà di accedere alla memoria condivisa il SO, previa decisione di concedergli l'accesso, provvederà a copiare i valori modificati nella zona di memoria del processo che deve contenere le var condivise. Viene concesso l'accesso in modo mutuamente esclusivo e così il processo può leggere il contenuto della memoria.
Questo richiede una doppia copiatura dei dati, e se la quantità di dati è molto grossa, ciò può portare ad una notevole lentezza nell'esecuzione delle operazioni di condivisione.
Questo è ancora più inaccettabile nel caso in cui si modifichino pochi dati nella zona di memoria in comune, in quanto molte delle info copiate siano info che non sono state modificate.

Realizzazione con area comune fisicamente condivisa
Se l'hw del sistema supporta una condivisione di memoria dei processi, e permette quindi di rilasciare i vincoli forti sul possesso univoco delle porzioni di memoria da parte di un processo si può creare una condivisione semplice: esisterà una porzione di memoria al di fuori dello spazio del sistema operativo, una porzione usata in modo mutuamente esclusivivo per condividere le info comuni.
In questo caso entrambi i processi vedono questa porzione di memoria condivisa come parte del loro spazio di indirizzamento memoria.
Questo evita la doppia copiatura, però prevede che esista la gestione hw e di SO che supporti la condivisione fisica di queste porzioni di memoria.
Questo è contro il principio generale di condivisione della memoria, ma è accettabile dal punto di vista dei processi specifici che cooperano tra di loro, in quanto ognuno mirerà al raggiungimento dello stesso obiettivo, senza cercare di arrecare danno all'altro.

Condivisione di buffer
In questo caso il processo mittente e destinatario si scambiano solo le info significative che si vuole trasferire tra i due.
Entrambi i processi avranno il loro spazio di indirizzamento, separati, e il processo mittente invierà una serie di informazioni al processo destinatario depositandole in una apposita area di memoria condivisa tra i due processi, ma ciò che viene trasferito è solo la porzione di informazione costruita appositamente dal mittente per il ricevente.

Sorgono comunque alcuni problemi:
- È necessario comunque identificare i due processi comunicanti (comunicazione diretta), in quanto lo scambio avviene in modo nominale tra i processi
- Si deve garantire la consistenza degli accessi alla zona condivisa
- Si deve sincronizzare l'accesso in mutua esclusione, proprio per garantire la coerenza e consistenza.

Questo sistema si può realizzare con:

- Buffer con copiatura gestita dal SO (come per la memoria condivisa)
- Buffer in memoria fisicamente condiviso

Processi produttore-consumatore.
In questo caso le comunicazioni avvengono utilizzando un buffer condiviso di capacità limitata o illimitata.
Se è illimitata il processo mittente invierà messaggi finché vuole senza mai fermarsi, mentre il processo consumatore acquisisce messaggi se presenti e si ferma se non ci sono messaggi disponibili.
Se invece la capacità è limitata quando il buffer sarà pieno il processo produttore dovrà attendere che si liberi dello spazio (facendo leggere le informazioni al consumatore).
La comunicazione avviene quindi con:

- il produttore genera le info elementari per il consumatore
- il produttore memorizza ciascuna info elementare nel buffer di comunicazione condiviso
- il consumatore acquisisce ciascuna info prendendola dal buffer condiviso
- il consumatore legge e usa ciascuna informazione generata dal produttore

Lezione 4 - Scambio di messaggi

Modello della comunicazione a messaggi
In questo modello, il processo mittente vuole inviare delle info (rappresentate in msg) per un processo destinatario. Il processo mittente gestisce l'invio con il supporto di opportune primitive di sistema operativo.
I msg conterranno un insieme di info atte a trasferire i dati desiderati dal mittente al destinatario:

- identificativo processo mittente
- identificativo processo destinatario
- informazioni da trasmettere
- eventuali altre info di gestione dello scambio di messaggi

I messaggi possono avere dimensione fissa o variabile.

Buffer
I messaggi vengono inviati da un processo tramite SO ad un destinatario
Il SO è responsabile di memorizzare le info da trasferire in spazi di memoria temporanea al suo interno, in buffer dedicati alla comunicazione.
I buffer possono essere assegnati a ciascuna coppia di processi, oppure di uso generale ed assegnati di volta in volta alla coppia di processi che desidera comunicare.
Il numero di buffer assegnati ad un processo può essere illimitato (in tal caso il processo potrà inviare quanti msg vuole) oppure limitato (in tal caso dovrà attendere in caso ecceda il numero di limite).
Se la quantità è nulla il processo dovrà sempre aspettare che ci sia un ricevente pronto in ricezione in modo da completare il trasferimento dei dati.

Funzioni
La funzione per l'invio dei messaggi offerta dal sistema operativo è la funzione
send(Q, messaggio)

Questa primitiva deposita il messaggio in un buffer libero del SO memorizzando che tale messaggio deve essere ricevuto dal processo destinatario Q. Automaticamente il processo memorizzerà chi è il processo mittente (ossia il processo attualmente in esecuzione).

La primitiva di invio può essere bloccante se non ci sono buffer liberi. In tal caso la procedura provoca il blocco della procedura mittente, in caso contrario la procedura consente al processo mittente di depositare il messaggio e di procedere senza attendere l'arrivo di un ricevente.
Se il numero dei buffer disponibili per l'utente è zero sarà sempre bloccante.

Il messaggio può essere ricevuto tramite un'operazione
receive(P, messaggio).
In questo caso il messaggio inviato dal processo P viene ricevuto se il destinatario era il processo Q specificato durante la send chiamata da P e il dato verrà deposto nella variabile "messaggio" nello spazio di indirizzamento del processo ricevente.
La primitiva è bloccante nel momenti in cui non c'è un messaggio ricevibile (mandato da P per il processo Q).

Invio condizionale
L'invio condizionale è utile per impedire che il processo in esecuzione venga bloccato. Ad esempio l'operazione di
cond_send(Q, messaggio) : error_status
provvede a depositare il messaggio se ci sono dei buffer liberi, altrimenti comunque non è bloccante e torna uno stato di errore, e il mittente non depositerà più il suo messaggio nel buffer.
Analogamente si può utilizzare una primitiva di ricezione condizionale
cond_receive(P, messaggio) : error_status
per cui valgono le stesse considerazioni.

Sincronizzazione dei processi comunicanti
Quando i processi accedono alle op. di comunicazione mediante scambio di msg, le comunicazioni possono essere sincronizzate in vari modi:

- Comunicazioni asincrone, che diventano bloccanti solo se l'operazione non può essere completata. Quindi il processo se trova spazio nei buffer depone il messaggio e procede senza essere sincronizzato col ricevente, se non trova spazio nei buffer il processo mittente si ferma e quindi è obbligato a sincronizzarsi con il processo ricevente.
- Comunicazioni sincrone: la comunicazione avviene solo e soltanto con la presenza contemporanea dei due processi. Nel caso in cui la coda di lunghezza sia nulla, il mittente sarà sempre obbligato ad aspettare che il destinatario arrivi ed esegua l'operazione di ricezione.

Identificazione dei processi comunicanti
Nella comunicazione con messaggi, se la com. è di tipo simmetrico, mittente e destinatario sono obbligati ad identificarsi, in quanto si ricade in un caso di comunicazione diretta.
Nel caso di comunicazione asimmetrica, mittente e destinatario possono non essere identificati univocamente. Se il destinatario non è specificato univocamente il destinatario può essere un processo di un gruppo specificato o un processo qualunque che richiede di ricevere dal processo mittente.
Specularmente nella ricezione, questa può avvenire da un processo di un gruppo specificato o da un processo qualunque che invia a lui.

Caratteristiche e problemi

- Vi sono quindi varianti nell'identificazione dei processi comunicanti
- Non è presente memoria condivisa tra processi
- Per accedere ai messaggi è necessaria una sincronizzazione, per garantire la consistenza del messaggio letto, e questa viene operata direttamente dal sistema operativo

Implementazione
Il SO mette a disposizione un insieme di spazi di memoria, i buffer, ai processi che vogliono comunicare, strutturati in
| mittente | destinatario | messaggio |

Quindi il processo mittente P, per inviare un messaggio al processo Q, genererà nel buffer una struttura simile a
| P | Q | mersaggio |

Il SO potrà mettere a disposizione dei due processi un insieme di buffer per trasmettere messaggi al fine di realizzare le operazioni bloccanti o non bloccanti come visto precedentemente.

Lezione 5 - Comunicazione con Mailbox

Modello della comunicazione
Il modello mailbox prevede che un processo mittente voglia scambiare un messaggio contenente informazioni attraverso non più una comunicazione diretta al processo ma utilizzando una struttura del SO chiamata “mailbox”, o “porta”.
Questa struttura contiene dei buffer dove i messaggi possono essere accumulati.
Quando il processo destinatario vuole ricevere un messaggio dalla mailbox accede a questa tramite una funzione di SO ed estrae il msg.

Messaggi
I messaggi conterranno delle informazioni di gestione (processo mittente, mailbox desiderata, info da trasmettere ed eventuali altre info a supporto della gestione dei messaggi nella mailbox).
Potranno avere lunghezza fissa o variabile.

Mailbox
- Illimitata: potrà contenere un numero illimitato di messaggi, e quindi il processo mittente non si vedrà mai bloccato in quanto troverà sempre spazio per depositare il messaggio
- limitata: potrà contenere un numero finito di messaggi, e il processo mittente se non trova spazio dovrà bloccarsi
- nulla: nessun messaggio può essere depositato, quindi il mittente potrà completare la comunicazione solo se c’è un ricevente in attesa

Funzioni
Il SO mette a disposizione funzioni di creazione e cancellazione delle mailbox:
`create(M)` e `delete(M)`
Per realizzare le comunicazioni vi sono funzioni di invio:
`send(M, messaggio)`
e questa depositerà il msg nella mailbox desiderata.
Se la mailbox avrà capacità illimitata non sarà mai bloccante, se la mailbox ha capacità limitata sarà bloccante solo qualora la mailbox desiderata è piena, se la mailbox ha capacità nulla sarà sempre bloccante a meno che non ci sia un processo in ricezione.
Il messaggio può essere ricevuto con la funzione:
`receive(M, messaggio)`
che sarà sempre bloccante se non ci sono messaggi nella coda.
Sussistono anche qui le funzioni di invio condizionale
`cond_send(M, messaggio): error_status`
Questa deposita il messaggio secondo le stesse condizioni della “send”, però, se la chiamata diventasse bloccante, questa tornerebbe uno stato di errore, il processo non si blocca e il msg non viene depositato.

Analogamente funziona la ricezione condizione
`cond_receive(M, messaggio) : error_status`

Sincronizzazione
Se la capacità è illimitata la comunicazione è asincrona (il mittente non si curerà del processo ricevente)
Se la capacità nulla la comunicazione sincrona (solo quanto i due processi sono contemporaneamente presenti nella mailbox possono proseguire la comunicazione)
Se la capacità è limitata si ha una comunicazione bufferizzata (asincrona finché c'è spazio, sincrona quando non c'è spazio).
Il processo che vuole deporre viene bloccato quando non c'è più spazio nella mailbox, e dovrà attendere che un processo liberi un buffer nella mailbox per proseguire (non dovrà attendere per forza lo specifico processo con cui vuole comunicare).

Caratteristiche e problemi
Non serve più identificare i processi comunicanti (si ha una comunicazione indiretta).
Non esiste memoria condivisa tra processi, e quindi nemmeno tutti i problemi di sincronizzazione che ne derivano.
La sincronizzazione per l'accesso ai messaggi è gestita implicitamente dal SO

Ordinamento delle code dei messaggi e dei processi in attesa
Le politiche di ordinamento delle code dei messaggi nella mailbox e dei processi in attesa possono essere diverse:
- FIFO
- Priorità
- Scadenza

Proprietà della mailbox
La mailbox può essere del SO, e quindi non è correlata ad alcun processo, oppure specifica di un processo (in questo caso solo il processo proprietario può ricevere dalla mailbox, a meno che questo non conceda questo diritto ad altri processi, mentre questi ultimi possono solo inviare. Quando il proprietario termina la mailbox scompare).

Comunicazioni con molti possibili mittenti o riceventi
Le mailbox permettono di realizzare comunicazioni con più processi mittenti o più processi riceventi. In ogni caso la comunicazione coinvolge sempre e solo due processi (un mittente e un ricevente), anche se questi non sono meglio identificati.

Comunicazioni molti a uno
La mailbox è vista da più processi mittenti che vogliono inviare un messaggio alla stessa mailbox. La mailbox vedrà un solo processo ricevente nella comunicazione. Questo processo, prima che vengano deposti i messaggi, se esegue la receive si metterà in attesa. Quando il messaggio arriva il processo potrà procedere nella sua computazione.
Questo è il caso tipico di un processo che serve una coda di richieste che possono essere avanzate da più processi client.

Comunicazione uno a molti
In questo caso alla mailbox faranno riferimento più processi riceventi. Questi si porranno in attesa che arrivino dei messaggi, e saranno ordinati con un'opportuna politica.

Quando arriva un processo mittente che effettua l'invio di un msg alla mailbox, questo depone, procederà, e il messaggio verrà dato al primo processo in attesa secondo la politica desiderata, che viene estratto dalla coda dei processi e procede, mentre gli altri processi rimarranno in attesa. Questo è il caso tipico di processi di servizio multipli, a cui un processo può chiedere l'esecuzione del servizio.

Comunicazione molti a molti
In questo caso alla mailbox faranno riferimento un insieme di processi che si pongono in attesa dell'arrivo dei msg, e un insieme di processi che desiderano inviare messaggi.
Alla deposizione di un messaggio, questo verrà dato immediatamente al primo processo in attesa, il quale verrà estratto dalla coda e procederà la computazione. Così vale per gli altri processi.
Questo è tipico quando si hanno più processi che servono una coda e più processi client che richiedono servizi (omogenei tra loro).

Lezione 6 - Comunicazione con File

Comunicazione mediante file condivisi
Queste vengono realizzate da un processo che desidera inviare un messaggio ad un altro processo utilizzando non buffer o memoria condivisa ma bensì strutture di memoria di massa.
In particolare utilizzare un file sul disco. Il processo mittente scrive sul file nel disco, e il ricevente legge dallo stesso file.

Comunicazione mediante pipe
Le pipe sono file posti in memoria centrale per migliorare la rapidità di accesso alle info.
Inoltre il meccanismo basato sulle pipe prevede che le info siano inviate in maniera strettamente sequenziale secondo la politica FIFO.
Il processo mittente che vuole inviare un messaggio lo farà tramite il SO, andando a scrivere in uno spazio di MC gestito dal SO i messaggi ordinatamente.
Il processo ricevente, mediante funzioni di SO, estrae i messaggi dalla pipe secondo l'ordine con cui sono arrivati. Questo è un meccanismo che permette la comunicazione tra due processi in modo efficiente in quanto le operazioni vengono svolte in MC, anche se le funzioni utilizzate sono le funzioni tipiche dei file.

Caratteristiche dei messaggi
I messaggi conterranno le info su processo mittente, le info da trasmettere ed eventuali altre informazioni a supporto della gestione dei messaggi.
La dimensione dei messaggi può essere fissa o variabile.

Funzioni
- Creazione: crea un file (o una pipe)
- Cancellazione: cancella un file (o una pipe)
- Lettura: legge da un file (o da una pipe)
- Scrittura: scrive in un file (o in una pipe)

Si tratta quindi delle funzioni standard di operazione sui file fornite dal SO.

Caratteristiche e problemi
Le funzioni per lettura e scrittura su file o pipe permettono di sincronizzare dei processi in lettura o in scrittura secondo le politiche e i meccanismi previste dal File System.
Se si usa il file l'ordinamento sarà deciso dal processo scrivente, mentre se si usano le pipe l'ordine sarà sempre FIFO.
I processi in attesa potranno essere ordinati a loro volta, nel caso dei file dipenderà dalla gestione del file system, mentre per le pipe la comunicazione diretta specifica con un processo soltanto

Lezione 7 - Comunicazione con socket

Modello della comunicazione mediante socket

In questo modello si avranno due macchine in un ambiente distribuito con due SO (eventualmente diversi) che supportano le operazioni dei processi applicativi.
Il modello della pipe permetteva di mettere in comunicazione i due processi mediante un canale di comunicazione in memoria centrale gestito come file, in cui i msg venivano inviati in ordine strettamente sequenziale.
Nell'ambiente distribuito si potrà generalizzare questo meccanismo, spezzando la pipe in due tronconi e porli uno nella macchina in cui si trova il processo mittente e uno nella macchina del processo destinatario.
Il processo mittente inserirà i msg nel troncone di invio, mentre il processo ricevente riceverà i messaggi prendendoli ordinatamente dal troncone di uscita.
Il trasferimento dei msg dalla macchina mittente alla macchina destinataria è assicurata dalla rete informatica.

Architettura

La comunicazione ha un'architettura in cui si ha una struttura client-server, e ha come obiettivo quello di supportare il trasferimento monodirezionale da un processo all'altro.
Il client invia la richiesta ad un server in ascolto su una porta, il quale riceve le richieste e le elabora, e risponde su un'altra porta al mittente, che ora è il ricevente del messaggio.
Il canale di comunicazione così creato si chiama "**Porta**" ed è individuato dall'indirizzo della macchina che lo ospita e da un numero progressivo che univocamente specifica la porta, ossia la struttura di comunicazione.

Caratteristiche dei messaggi

I messaggi possono contenere info da trasmettere (non servono altre info, in quanto gli attori della comunicazione sono identificati da indirizzo e porta).
La dimensione può essere fissa o variabile

Funzioni

Le funzioni di **creazione** e **cancellazione**, come **lettura** e **scrittura**, sono specifiche del SO che si considera, e in particolare dei protocolli di comunicazione in rete considerati.

Canale di comunicazione

Nella realizzazione, la pipe viene spezzata in due per creare il socket. La rete garantisce il trasferimento delle info da mittente a destinatario.
Sempre la rete potrà ricreare lo stesso canale di comunicazione al contrario in modo che il processo destinatario possa rispondere al processo inizialmente mittente.
Ciascun canale di comunicazione è monodirezionale, e la coppia tra i due canali crea un canale di comunicazione bidirezionale.

Caratteristiche e problemi

L'ordinamento dei messaggi è FIFO
L'ordinamento dei processi in attesa è FIFO.
La connessione può essere gestita in vari modi, con gestione della connessione per cui ogni sequenza di messaggi permane garantita e controllata anche nel caso di problemi di comunicazione in rete, oppure nel caso può essere scambiato separatamente dagli altri, e quindi è realizzabile un meccanismo di multicast che permette di recapitare un messaggio a più processi.

UD5 - Sincronizzazione tra processi
Lezione 1 - Processi concorrenti

Concorrenza

Si ha la concorrenza tra processi quando essi cercando di accedere a risorse condivisi usabili solo in mutua esclusione.
Ciò rende necessario sincronizzare i due processi in modo che possano usare la risorsa condivisa in maniera opportuna.

Sincronizzazione per l'uso di risorse condivise

La sincronizzazione prevede che i processi possano utilizzare le risorse fisiche e informative in modo coordinato, facendo in modo che la risorsa rimanga consistente e congruente e possa essere utilizzata in maniera corretta.

Sezione critica

Ci possono essere delle porzioni di codice che, se eseguite in maniera concorrente, possono generare errori.
Si pensi per esempio se due sezioni effettuano una prima un incremento di una variabile temporanea e poi un'assegnazione di una variabile condivisa, mentre l'altra esegue un decremento su una var. temporanea e poi un'assegnazione alla stessa variabile condivisa: se la prima sezione venisse interrotta tra l'incremento e Il'assegnamento, l'esecuzione della seconda sezione porterebbe ad avere un valore errato nella variabile condivisa.
Il problema è che bisogna garantire l'esecuzione atomica di queste porzioni, cioè bisogna garantire la mutua esclusione.
Quindi ciascuna delle due sezioni deve essere eseguita come *sezione critica*.
Quindi la sezione critica è una sequenza di operazioni che, se eseguire in maniera concorrente, possono generare errori, e quindi devono essere eseguite secondo delle condizioni:

- mutua esclusione con gli altri processi che eseguono le rispettive sezioni critiche
- bisogna garantire che soltanto i processi che stanno cercando di entrare nella sezione critica possano partecipare alla decisione su quale processo effettivamente entra.
- con la sincronizzazione per l'accesso alla sezione critica, ci potrebbero essere dei processi che attendono un tempo indefinito (quindi in condizione di starvation). Quindi la gestione della sez. critica deve garantire che l'attesa di accesso alla sez. sia limitato, in modo che tutti i processi riescano operare

Sincronizzazione di processi cooperanti

Si può avere la sincronizzazione non solo per l'accesso concorrente a risorse fisiche o informative condivise, ma anche per sincronizzare processi cooperanti.
La concorrenza prevede che i processi cerchino di competere per l'uso di processi, ma generalmente non che ci sia una collaborazione per il raggiungimento di un obiettivo comune.
La sincronizzazione prevede che le risorse condivise vengano usate in modo corretto, e quindi garantire la consistenza delle info condivise.
In caso di processi cooperanti si può pensare ad una sincronizzazione, in quanto nello svolgimento delle attività può essere utile essere certi che l'evoluzione della computazione dei processi avvenga in modo coordinato.
Può infatti sorgere la necessità per un processo di voler verificare con certezza che un processo cooperante abbia raggiunto un certo stato della sua computazione.
Ciò si può ottenere facendo sì che il due processi, arrivati ad un certo punto della computazione, si sincronizzino, cioè che un processo segnali all'altro di essere arrivato al punto della computazione interessato, in modo che così il processo che ha ricevuto la segnalazione sia sicuro di quanto eseguito dall'altro processo.

Una volta inviata la comunicazione si sincronizzazione, i processi proseguiranno nella loro computazione (una volta ricevuta la segnalazione).

Lezione 2 - Variabili di lock

Variabile di turno

Una variabile di turno indica il turno di uso di una risorsa tra un insieme di processi, cioè quale processo ha il diritto di utilizzare la risorsa in un certo istante.

Sincronizzazione tra due processi concorrenti mediante variabile di turno

- questa tecnica può essere generalizzata per gestire anche più di due processi, usando la variabile per identificare il processo -

Algoritmo 1

```
public class Algorithm_1 implements MutualExclusion
{
	private volatile int turn;

	public Algorithm 1() {
		turn = TURN 0;
	}
	public void enteringCriticalSection(int t) {
		while (turn != t)
			Thread.yield();
	}
	public void leavingCriticalSection(int t) {
		turn = 1 - t;
	}
}
```

La variabile di turno identifica il processo che può utilizzare la risorsa.
Viene inizializzata al valore del primo processo che può utilizzare la risorsa (0).
Quando il processo vuole utilizzare una risorsa deve chiamare la funzione di entrata nella sezione critica indicando qual è il processo che sta operando.
Il processo andrà a verificare se il turno è il suo o meno.
Quando il processo vorrà lasciare la sezione critica chiamerà la relativa funzione e cambia il valore del turno, ponendolo uguale all'id dell'altro processo. Se l'altro processo era in attesa di utilizzare la risorsa può finalmente entrare.

Questo algoritmo garantisce una mutua esclusione ma impone una stretta alternanza dei processi (un processo non potrà entrare due volte consecutivamente nella sezione critica, dovrà attendere che vi entri l'altro processo.
Non garantisce la condizione di progresso, in quanto non garantisce che siano i processi che non stanno eseguendo la loro sezione critica che possano decidere quale processo entra nella sezione.

Algoritmo 2

```
public class Algorithm_2 implements MutualExclusion
{
	private volatile boolean flag0, flag1;
	public Algorithm 2() {
		flag0 = false; flag1 = false;
	}
	public void enteringCriticalSection(int t) {
		if (t == 0) {
			flag0 = true;
```

```
                while(flag1 == true)
                        Thread.yield();
        }
        else {
                flag1 = true;
                while (flag0 == true)
                        Thread.yield();
        }
    }

    public void leavingCriticalSection(int t) {
        if (t == 0) flag0 = false; else flag1 = false;
    }
}
```

Questo algoritmo risolve il problema della stretta alternanza dei processi, introducendo due var: la var flag0 e flag1, che indicando per i processi l'utilizzo corrispondente delle risorse.
Sono inizializzate a "falso" per indicare che nessun processo sta utilizzando la risorsa.
All'entrata nella sezione critica il processo imposta il flag corrispettivo a "true" per indicare che il processo è in richiesta della sezione critica, e quindi il processo verifica quali sono i processi che non stanno eseguendo la loro sezione critica (verificando che il flag sia falso) , e in caso entra nella sezione critica, altrimenti rimarrà in attesa che il gli altri processi liberino la risorsa. Al rilascio della sezione critica, il processo ripristinerà il flag allo stato precedente, permettendo all'altro processo di usare il processo.

Questo non impone la stretta alternanza di processi.
Non impone però la regola del progresso.
Si può inoltre creare una possibile attesa infinita.

Algoritmo 3

```
public class Algorithm_3 implements MutualExclusion
{
    private volatile boolean flag0;
    private volatile boolean flag1;
    private volatile int turn;
    public Algorithm_3() {
        flag0 = false;
        flag1 = false;
        turn = TURN_0;
    }
    public void enteringCriticalSection(int t) {
        int other = 1 - t;
        turn = other;
        if (t == 0) {
                flag0 = true;
                while(flag1 == true && turn == other)
                        Thread.yield();
        }
        else {
                flag1 = true;
                while (flag0 == true && turn == other)
                Thread.yield();
        }
    }

    public void leavingCriticalSection(int t) {
        if (t == 0) flag0 = false; else flag1 = false;
    }
```

}

Questo algoritmo introduce, oltre ai flag di stato di uso della risorsa, una variabile di turno che identifica quale processo ha diritto di utilizzare la risorsa.
Le var. verranno inizializzate a false e 0.
Quando un processo entrerà nella sez. critica corrispondente andrà ad assegnare al valore della var. di turno il valore dell'altro processo, e andrà ad assegnare al flag il valore di "true".
Dunque verificherà se l'altro processo aveva dichiarato di voler utilizzare la risorsa e andrà a vedere se è il proprio turno o meno.
Nel caso in cui l'altro processo non abbia fatto nulla, si potrà procedere all'utilizzo della risorsa. Al termine dell'utilizzo ripristinerà il valore "false" nel proprio flag per far sì che l'altro processo possa procedere ad acquisirla.
Importante è che se, dopo l'esecuzione di assegnamento alla var turn, un altro processo la esegue viene cambiato nuovamente il turno, e quindi il primo processo arrivato sarà quello che prende la risorsa.

Variabili di lock
Si tratta di una generalizzazione dell'utilizzo delle variabili di turno.
Mentre queste ultime dichiarava lo stato di uso della risorsa da parte dei processi mettendo in evidenza il turno dei processi, la variabile di lock indica in modo assoluto lo stato di utilizzo della risorsa, e eventualmente da parte di quale processo. (0 → libera, 1 → in uso).

Uso ad interruzioni disabilitate
Un modo di usare le variabili di lock che garantisca la mutua esclusione e l'uso corretto delle informazioni è quello di disabilitare le interruzioni.
In questo caso saranno le operazioni sulla variabile di lock a dover essere eseguite in maniera atomica, per garantire che nessun altro si possa infilare in mezzo a causa delle interruzioni o della turnazione dei processi, andando a cambiare il risultato corretto di analisi della var. di lock.
Dunque si seguirà il percorso:

- disabilito le interruzioni
- leggo la variabile di lock
- se la risorsa libera (lock=0), la marco in uso ponendo lock=1 e riabilito le interruzioni
- se la risorsa è in uso (lock=1), riabilito le interruzioni e pongo il processo in attesa che la risorsa si liberi

Questo garantisce che la decisione effettuata sia effettivamente basata con quanto letto dalla var di lock. Se la risorsa è occupata, il processo sarà messo in attesa.
Nel momento in cui la risorsa viene rilasciata si porrà la variabile di lock = 0.
Funziona bene su sistemi monoprocessore, ma non è molto ben scalabile su sistemi multiprocessore.

Hardware per la sincronizzazione
Per ovviare al problema di atomicità, e quindi di abilitazione e disabilitazione delle istruzioni, su alcuni processori è stata implementata l'istruzione macchina TEST-AND-SET, la quale è per definizione atomica.
Questa istruzione legge il valore della variabile di lock, e pone tale valore in un flag del processore, e poi scrive nella var di lock il valore 1.
Se il vecchio valore era 0 la risorsa era libera, e quindi viene presa con l'esecuzione dell'operazione di TEST-AND-SET.
Per decidere se la risorsa era stata presa oppure no è possibile leggere il valore nel flag del processore.

Lezione 3 - Semafori

Obiettivo

L'obiettivo dell'uso dei semafori è quello di elevare il livello di astrazione portando la gestione della sincronizzazione in *funzioni del sistema operativo*.
Obiettivo di questo è garantire la corretta gestione della sincronizzazione mediante l'accesso alle variabili di supporto alla mutua esclusione, e quindi evitare tutti i problemi generati dall'utilizzo di variabili di turno o di lock.

Semaforo Binario

Si tratta di una variabile binaria che rappresenta lo stato di uso della risorsa condivisa (1 → libera, 0 → in uso).
Il semaforo è manipolato dalle funzioni:
`acquire(S)` → acquisisce l'uso della risorsa
`release(S)` → rilascia la risorsa
Queste funzioni sono anche note come P() e V().
Queste operazioni sono atomiche, in quanto sono procedure di sistema (e quindi valutate in maniera mutuamente esclusiva).

Uso del semaforo binario

Il semaforo è essenzialmente una variabile inizializzata 1 per indicare che la risorsa rappresentata dal semaforo è libera.
Quando arriva un processo che vuole acquisire la risorsa eseguirà una funzione di acquire(), e quindi porrà il valore del semaforo a 0, prendendosi la risorsa e procedendo nella sua computazione.
Se arriva poi un altro processo che esegue l'acquire, tale operazione non verrà completata in quanto il semaforo è 0, e quindi il processo verrà accodato in attesa della liberazione della risorsa.
La coda di attesa dei processi in attesa verranno ordinati secondo una politica stabilita sul singolo semaforo (es. FIFO, priorità, scadenza nei sistemi a real time).
Quando il processo vorrà rilasciare la risorsa richiamerà la funzione di release, segnalerà quindi che la risorsa è tornata disponibile, e quindi dovrebbe riporre a 1 il semaforo per permettere agli altri processi di prendere la risorsa (se ci sono processi in attesa, il primo processo in lista otterrà l'utilizzo della risorsa).

Implementazione del semaforo binario

Può sussistere un caso di attesa attiva in caso di risorsa non disponibile (dove il processo in attesa guarda continuamente il valore del semaforo binario). In questo caso basta avere la struttura dati del semaforo binario e la funzione acquire rimarrà in ciclo a leggere il valore della variabile semaforo finché questo non diventa disponibile. Questo comporta uno spreco di risorse.

La soluzione sta nell'avere una sospensione in caso di risorsa non disponibile, e quindi una riattivazione automatica quando la risorsa torna disponibile per il primo processo in coda.
In questo caso si avrà come struttura dati una variabile binaria per il semaforo e in più serve una coda di processi in attesa di acquisire la risorsa.
La procedura di acquire sospende il processo quando la risorsa non è disponibile e lo mette nella coda di attesa del semaforo.
La procedura di release rilascerà la risorsa e riattiverà il processo, andando a prenderlo da tale coda.
Vi sarà quindi uno schedulatore che ordinerà i processi in attesa in modo opportuno per la politica adottata.

Semaforo generalizzato
Si tratta di una generalizzazione del semaforo binario nel caso si abbia un insieme di risorse omogenee da rappresentare. In questo caso S è una variabile intera che rappresenta lo stato di uso di un insieme di risorse omogenee condivise.
Il semaforo verrà posto uguale a N con N = numero di risorse libere. Assumerà valore 0 quando tutte le risorse saranno in uso. Assumerà n-1, n-2 ecc. ogni volta che verrà richiesto l'uso per una risorsa.
Il semaforo è sempre manipolato dalle funzioni di acquire e release.

Uso del semaforo generalizzato
Il semaforo è inizializzato al numero di risorse omogenee.
L'acquire decrementerà il semaforo, e quindi il numero di risorse disponibili.
Se il semaforo è 0, il processo sarà messo in attesa che un processo rilasci una delle risorse.

Lezione 4 - Monitor

Problemi legati all'uso dei semafori
Quando si utilizzano i semafori vi possono essere diversi problemi, ad esempio errori di programmazione non trattabili dal SO.
In particolare si può avere la violazione della mutua esclusione (il programmatore si può dimenticare di usare l'acquire) o di attese infinite (il programmatore si può dimenticare di usare la release). Essendo responsabilità del programmatore, il SO non può fare alcun tipo di controllo e gestione delle primitive legate all'uso dei semafori
Il motivo è che le primitive relative ai semafori sono chiamate di sistema operativo, e come tali operano solo se vengono chiamate (e grazie ar ca…. ☺).

Obiettivo del monitor
La soluzione è cercare di non lasciare al programmatore questa responsabilità, innalzando il livello di astrazione per la gestione della sincronizzazione forzandone l'uso corretto.
La soluzione è quindi il MONITOR, ossia un costrutto linguistico che il compilatore provvede a trasformare nelle chiamate di sistema opportune effettuate nei momenti giusti, e a garantire che tali chiamate vengano eseguite.

Definizione di monitor
Un monitor è un costrutto di sincronizzazione formulato a livello di linguaggi di programmazione.
Un processo alla volta soltanto potrà trovarsi all'interno di un monitor e quindi essere attivo.
Il monitor è sostanzialmente la collezione delle sezioni critiche dei processi, ossia delle procedure, che devono essere eseguite in mutua esclusione.
Quindi quando si entra nel monitor, una sola procedura può diventare attiva. Il processo chiamerà quindi il processo che realizzerà la sua sezione critica, e il fatto che queste siano nell'ambito di un monitor garantisce che vengano richiamati i meccanismi di acquisizione (e controllo) per le risorse da utilizzare e quelli di rilascio delle risorse.

Realizzazione del monitor
Per la realizzazione viene utilizzata una serie di informazioni condivise per rappresentare la risorsa comune, le operazioni da effettuare sulla risorsa comune, e del codice di inizializzazione dell'ambiente operativo.
Quando un processo non ottiene immediatamente l'uso della risorsa contenuta nel monitor verrà accodato in una coda di attesa per l'accesso alla risorsa.

Uso del monitor
Il monitor sarà costituito da un insieme di risorse da gestire in mutua esclusione e di operazioni da attuare su tali risorse.
Quando si vuole gestire la mutua esclusione, il compilatore associa automaticamente al monitor una variabile condizione che rappresenta lo stato di uso del monitor (libero o utilizzato, come un semaforo).
Questa condizione viene inizializzata a “libero”. Quando arriva un processo che vuole chiamare una delle operazioni contenute nel monitor, se la condizione dice che il monitor è libero, allora il processo ottiene l’uso della primitiva e procede. La condizione verrà quindi posta a 0, per indicare che il monitor è occupato.
All’arrivo di un altro processo, questo troverà il monitor in uso e dovrà quindi attenere nella coda di attesa dei processi che non riescono a completare le operazioni, e rimarrà nella coda fino alla liberazione della risorsa.
Al termine della risorsa, il monitor viene rilasciato automaticamente, ed essendoci un processo pendente il sistema provvederà ad attivare una primitiva di signal per risvegliare il processo e dargli l’uso del monitor.

Lezione 5 - Problemi della starvation e del deadlock

Starvation - Blocco indefinito
Si ha starvation quando un processo rimane bloccato in attesa indefinita, in quanto gli altri processi ottengono sempre prima la risorsa.
La causa di ciò è l’uso di una politica di schedulazione della coda di attesa che non garantisce a tutti i processi di ottenere in un tempo finito la risorsa.
La soluzione a questo problema è la scelta accurata dell’algoritmo di schedulazione dei processi in attesa (ad esempio nella politica FCFS questo blocco non avverrebbe).

Deadlock - Stallo
Si ha il deadlock quando in un gruppo di e due o più processi ciascuno aspetta una risorsa che è detenuta in modo mutuamente esclusivo da un altro processo del gruppo.
La causa di questo blocco è l’attesa circolare di risorse senza rilascio.
Questa situazione si verifica quando, ad esempio, abbiamo due risorse che vengono utilizzate in maniera mutuamente esclusiva da due processi, e ad un certo punto entrambi i processi, per procedere nella computazione, devono fare uso della risorsa usata dall’altro processo.
La soluzione per questi problemi è utilizzare alcune tecniche per impedire, prevenire, risolvere o ignorare (solo se la probabilità di stallo è molto bassa e lo stallo non riguarda una zona critica del sistema) il problema.

Lezione 6 - Transazioni atomiche

Definizione di transazione
Una transazione atomica è un insieme di istruzioni che eseguono un’unica funzione logica nell’applicazione, ad esempio una sequenza di letture, scritture e manipolazioni di dati da eseguire come unica entità, e quindi ne deve essere garantita l’atomicità.

Atomicità della transazione
Si definisce “atomicità della transazione” per indicare che l’effetto della transazione sulle informazioni memorizzate deve essere permanente solo se tutte le operazioni sono state completate correttamente senza interferenze da parte di altri processi.
Ovvero la sequenza delle operazioni di una transazione deve essere atomica, come un’unica operazione indivisibile.
Una transazione

- viene vista terminare correttamente ed esegue la “commit” quando tutte le operazioni sono state terminate correttamente senza interferenza da altri processi (effetti resi permanenti)
- viene terminata in maniera errato con un segnalazione di “abort”, quindi bisogna annullare gli effetti sulle info memorizzate dal sistema, e bisogna effettuare un rollback alla situazione precedente.

Tipologie di archivi
In un sistema di elaborazione si possono avere diversi tipi di archivi:
- Archivio volatile: le informazioni non sopravvivono allo spegnimento del sistema (tipicamente in cache e in memoria centrale)
- Archivi non volatile: le informazioni sopravvivono allo spegnimento del sistema (tipicamente dischi magnetici e ottici, nastri magnetici o altri dispositivi di mem. permanente)
- Archivio stabile: archivio in cui le info non vengono mai perse (replicazione in molti archivi non volatili, in modo da far sì che sopravviva almeno una copia)

Transazioni atomiche individuali
Le transazioni atomiche prese individualmente, che non condividono informazioni, quindi che non hanno una concorrenza di altre transazioni sulle stesse risorse informative condivise, possono essere gestite mediante le tecniche di “logging” e “check pointing”

Write-ahead logging
Il log (registro) delle transazioni memorizza in un archivio stabile le transazioni e il loro stato di esecuzione:
- Nome della transazione
- Nome dell’oggetto dei dati
- Vecchio valore dei dati
- Nuovo valore dei dati

Il meccanismo di write-ahead logging consiste nel memorizzare all’inizio della transazione una informazione all’inizio del registro che segnali l’inizio della transazione stessa (quindi si scrive un record con il nome della transazione T_i e si indica che sta partendo).
Quando la transazione termina si scrive nel log un record in cui si memorizza il nome della transazione T_i e si indica che la transazione è stata completata correttamente (“commits”).
Se la transazione abortisce o non viene terminata correttamente, non si scrive il commit.

Recupero basato sul log
Il recupero tramite file di log avviene tramite due operazioni:
`undo(`T_i`)`**:** riporta i dati modificati dalla transazione ai vecchi valori andando a leggere cosa era scritto nel valore vecchio prima dell’operazione registrata del log
`redo(`T_i`)`**:** riesegue le operazioni della transazione rimodificando i dati toccati dalla transazione e riportandoli al nuovo valore
Queste operazioni sono idempotenti, cioè la ripetuta applicazione ai dati di un sistema porta sempre ad ottenere lo stesso risultato.

Usare il log per effettuare il ripristino vuol dire analizzare la sequenza di informazioni registrate nel log e decidere quale delle due operazioni chiamare.
Se la transazione viene abortita a causa di un errore in essa, il log non contiene la frase di commit, e quindi il sistema deve applicare l’operazione di “undo”.
Se il sistema di elaborazione fallisce (a causa di un guasto nell’hardware o nella rete) per ogni transazione del log si avrà uno stato di esecuzione della transazione:

- se il log contiene la dichiarazione di inizio della transazione (starts) ma non quella di fine (commits) la transazione deve considerarsi fallita si deve eseguire "undo(T_i)".
- se il log contiene sia la dichiarazione di inizio che di fine (starts e commits) vuol dire che il sistema assumeva di aver completato la transazione, e quindi deve essere applicata "redo(T_i)".

Check pointing
L'utilizzo del logging può portare ad un lungo tempo di ripristino per i log, in quanto se si sono verificate molte transazioni, bisognerà partire ad esaminarle tutte dall'inizio e il tempo richiesto per la riapplicazione delle azioni svolti può diventare lungo.
La soluzione è quella di utilizzare dei punti di verifica (check point).
Questa soluzione prevede che periodicamente si scrivano in archivio stabile dei record del log memorizzati sull'archivio volatile, e quindi si scrivono i dati modificati sull'archivio stabile. Quindi si marca sul checkpoint di sistema (su archivio stabile dei log di sistema) il fatto che è stato raggiunto un punto di checkpoint.

Ripristino basato su check pointing
Il ripristino in caso di fallimento del sistema di elaborazione consiste nell'eseguire le modifiche alle informazioni volatili che si hanno nel sistema non a partire dall'inizio del log ma dal checkpoint più recente del log.
A questo punto si verificherà se esite solo l'inizio della transazione (undo) o sia inizio che fine (redo).

Transazioni atomiche concorrenti
Le transazioni possono essere non solo individuali (cioè eseguite su archivi separati, per cui ogni sequenza di transazioni opera su un singolo insieme di archivi) ma possono essere eseguiti da processi diversi in modo concorrente.
L'esecuzione concorrente di transazioni atomiche diventa più complessa, in quanto bisogna garantire che la sequenza delle transazioni venga eseguita in un modo seriale (secondo un ordine arbitrario) in modo da garantire l'atomicità delle operazioni svolte nella transazioni.
Quindi si dovrà creare per le transazioni concorrenti una condizione di serializzabilità (quindi andare a porre le azioni delle transazioni in un ordine seriale tale per cui il risultato risulti corretto dal punto di vista delle interazioni delle info memorizzate.

Tecniche per la serializzabilità
Per serializzare le transazioni si pul operare a due livelli:

- a livello di intera transazione: e quindi garantire che le intere transazioni siano svolte in sezioni critiche, e quindi garantire la serializzazione ponendo un semaforo di mutua esclusione tra le varie transazioni.
  Questo comporta una lentezza tra l'esecuzione delle transazioni e riduce le possibilità di parallelismo, in quanto le transazioni, seppur avviate da processi diversi, potranno essere eseguite una sola alla volta (in quanto l'accesso alla sez. critica è mutuamente esclusivo).

- a livello di operazioni nelle transazioni: si va a garantire che la serializzazione ci sia solo sulle operazioni strettamente necessarie, e quindi con l'introduzione di algoritmi per schedulare le informazioni in modo concorrente e seriale, oppure serializzare le operazioni e garantire che siano assicurati gli ordini corretti di accesso alle informazioni.

Schedulazione concorrente seriale

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Questa schedulazioni assicura che le operazioni svolte su un determinato record (A o B) per la transazione 0 avvengano prima delle operazioni della transazione 1.
In questo caso è limitata la possibilità di uso delle informazioni e di evoluzione parallela dei due processi.

Schedulazione concorrente serializzabile

T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

In questo caso si valutano le dipendenze che strettamente devono essere serializzate, non andando ad imporre le serializzazioni non necessarie.
Per realizzare ciò si possono utilizzare diverse tecniche.

Lock
Il lock, o blocco, è una variabile associata ad un dato e che definisce l'accessibilità al dato stesso.
Il lock viene posto a "libero" se l'accesso è consentito o "in uso" se la transazione deve essere sospesa in attesa che il dato si liberi.
Tipi di lock:

- Lock condiviso: consente a più processi di accedere allo stesso dato nel caso in cui le operaizoni da svolgere non siano conflittuali (es. più transazioni vogliono leggere lo stesso dato)
- Lock esclusivo: fa in modo che solo un processo veda usabile la risorsa, escludendo gli altri processi (es. un processo vuole effettuare una modifica sul dato)

Protocollo di lock di base
Il protocollo di lock di base prevede che una transazione applichi il lock sul dato da utilizzare.
Se il lock è disponibile, la transazione accede al dato.
Se il lock non è disponibile:

- se il lock richiesto è esclusivo, la transazione attende finché il dato non viene rilasciato
- se il lock è condiviso la transazione può accedere al dato se esso è correntemente bloccato con lock condiviso (la transazione che aveva lockato il dato l'ha fatto magari solo per leggerlo), altrimenti, se il lock con cui era stato bloccato era esclusivo, attende il rilascio.

Una situazione di questo genere non garantisce la serializzabilità.

Protocollo di lock a due fasi
In questo protocollo si avranno due fasi:

- Fase di crescita: una transazione può ottenere dei lock ma non li può rilasciare (li chiede tutti all'inizio o li richiede comunque senza mai rilasciarli)
- Fase di contrazione: dopo aver eseguito le sue elaborazioni una transazione può rilasciare i locks ma non ne può ottenere di nuovi

Questo garantisce la serializzabilità ma può portare a situazioni di stallo.

Serializzazione nei protocolli di lock
L'ordine di serializzazione di ogni coppia di transazioni in conflitto è determinato dal primo lock richiesto e che definisce le incompatibilità.

Timestamp
Una tecnica per ottenere la serializzazione è l'utilizzo del timestamp, ossia l'utilizzo di una marca di tempo $TS(T_i)$ che viene utilizzata come attributo della transazione e indica quando la transazione entra nel sistema.
Tipicamente viene associato alla transazione dal SO, e usualmente è valore del clock di sistema nel momento in cui si attiva la transazione, o un contatore che numera progressivamente e in modo univoco le transazioni.

Protocollo basato su timestamp
Il protocollo basato su Timestamp prevede l'uso di due tipi di timestamp:

- W-timestamp(Q)
- R-timestamp(Q)

Dunque, *ogni operazione di read o di write in conflitto viene eseguita nell'ordine del timestamp*.

Se si vuole eseguire un'op. di lettura "**read(Q)**":

- *se il timestamp della transazione è < del W-timestamp* associata al dato Q, la lettura è negata e T_i deve eseguire il rollback, cioè cancellare tutte le azioni effettuate, in quando sta andando in lettura di un dato Q che è stato sovrascritto
- *se il timestamp della transazione è >= del W-timestamp(Q)*, la lettura è eseguita e il R-timestamp(Q) viene posto uguale al massimo tra R-timestamp(Q) e il timestamp della transazione.

Se si vuole eseguire un'op. di scrittura "**write(Q)**":

- se *il timestamp della transazione è < del R-timestamp(Q) o è < del W-timestamp(Q)* la scrittura è negata e T_i esegue il rollback (rispettivamente, viene negato perché si sta producendo un dato che sarebbe stato necessario precedentemente, oppure si sta scrivendo su un valore ormai obsoleto)
- altrimenti, la scrittura è eseguita

Serializzazione nei protocolli di timestamp
L'ordine di serializzazione di ogni coppia di transizioni di in conflitto è determinato dal timestamp associato a ciascuna transazione alla sua attivazione. Questo garantisce una serializzazione delle operazioni elementari e quindi un elevato parallelismo e concorrenza nel sistema.

UD6 - Deadlock

Lezione 1 - Caratterizzazione del deadlock

Uso di risorse condivise

Quando si hanno risorse condivise nel sistema è necessario prestare attenzione al loro uso, in modo da garantire consistenza e coerenza.
Ci sono risorse che possono essere usate in modo non esclusivo (come un file in sola lettura) o solo in modo mutuamente esclusivo (come una stampante).
Diventa quindi indispensabile sincronizzare l'accesso dei processi all'uso di risorse condivise usabili solo in modo mutuamente esclusivo:

- Effettuano la richiesta d'uso della risorsa (risolve la contesa per l'utilizzo mutuamente esclusivo della risorsa)
- Usando la risorsa
- Rilasciando la risorsa (in modo che gli altri processi possano utilizzare la risorsa)

Problema del deadlock

Si ha il problema del deadlock quando si ha un insieme di processi che rimangono in attesa indefinita quando le risorse che richiedono sono in possesso di altri processi a loro volta in attesa.

Condizioni per il verificarsi del deadlock

- Mutua esclusione: indica che il fatto che la risorsa deve essere utilizzata solo in modo mutuamente esclusivo per garantirne la consistenza
- Possesso e attesa: indica il fatto che un processo potrebbe aver ottenuto l'uso della risorsa e non la rilascia, entrando però in attesa di altre risorse
- no rilascio anticipato (no pre-emption): la risorsa non è sottoponibile a pre-emption, non può essere tolta al processo che la sta usando prima che esso la rilasci, per garantire la corretta consistenza della risorsa
- attesa circolare: prevede il fatto che un insieme di processi siano uno in attesa di una risorsa posseduta da un altro in una maniera circolare (es. P1 detiene una risorsa e si mette in attesa di una risorsa detenuta da P2. P2 detiene quindi la risorsa e si mette in attesa di una risorsa del processo P3 e così via finché il processo Pn non rimane in attesa di una risorsa del processo P1.

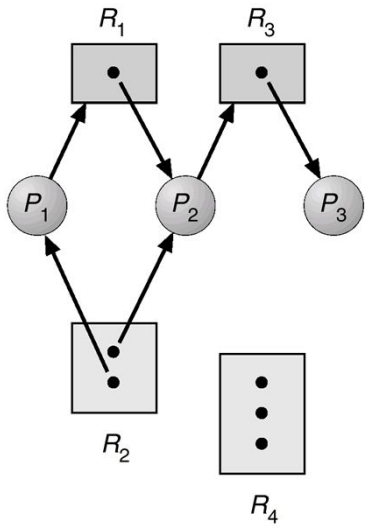
La condizione di deadlock si verifica se tutte queste condizioni avvengono contemporaneamente.

Grafo di allocazione delle risorse

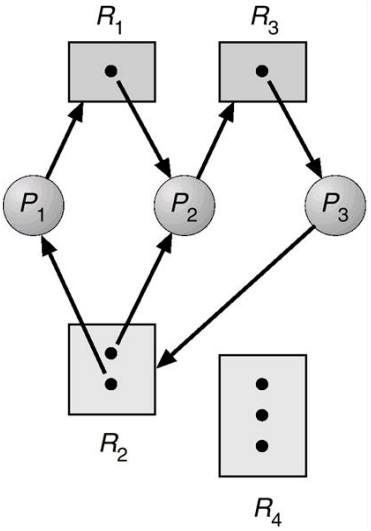
Un modo per verificare la presenza di deadlock è l'utilizzo del grafo di allocazione ecc.
Si tratta di un grafo che dispone di un insieme di nodi V e di un insieme di archi E.
I nodi possono essere processi del sistema P $\{P_1, P_2, \ldots, P_n\}$ oppure risorse del sistema R $\{R_1, R_2, \ldots, R_n\}$, eventualmente con più istanze identiche, rappresentate comunque da un unico nodo R.
Gli archi possono essere:

- archi di richiesta (di un processo di una risorsa), che escono da un processo ed entrano in una risorsa
- archi di assegnazione (di una risorsa ad un processo), che escono da una risorsa ed entrano in un processo

Senza deadlock

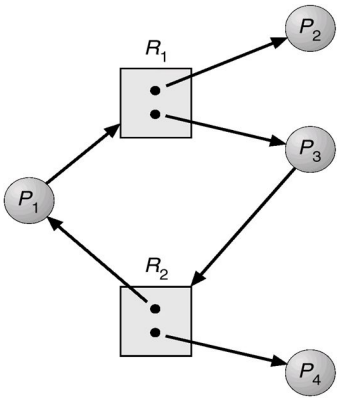


Con deadlock



Nel grafo senza deadlock non ci sono attese circolari, in quanto non vi sono cicli.
Nel grafo con deadlock vi sono attese circolari, in quanto il processo P_3 ha richiesto una risorsa in possesso di P_2, e quindi il processo P_2 detiene la risorsa R_2 e si mette in attesa di R_3, che in unica copia è posseduta da P_3 che è in attesa di della risorsa R_2 (che non gli sarà mai data, in quanto P_1 è in attesa del rilascio di R_1 da parte di P_2).

Ciclo senza deadlock



In questo caso, pur essendoci un ciclo, non vi è deadlock, in quanto al rilascio della risorsa da parte di P_4, P_3 potrà prendere possesso della seconda copia della risorsa R_2.

Metodi di gestione del deadlock

- Ignorare il deadlock (se non è critico, si fa finta che non ci sia)
- Prevenzione del deadlock
- Evitare il deadlock (lavora sul modo in cui vengono allocate le risorse)
- Rilevazione e recupero del deadlock (permette di lasciar verificare il deadlock, di accorgersi di tale condizione e recuperare la situazione riportando la situazione alla normalità)

Lezione 2 - Tecniche di prevenzione del deadlock

Principio della prevenzione

La prevenzione del deadlock può essere ottenuta impedendo si verifichino contemporaneamente le quattro condizioni per cui si verifica.
L'obiettivo della prevenzione del deadlock è quello di far sì che almeno una delle quattro condizioni non sia soddisfatta.

Mutua esclusione

La mutua esclusione deve essere soddisfatta per tutte le risorse non usabili in modo condiviso.
Non è però necessaria se le risorse possono essere usate in modo condiviso.
Si può garantire che non si verifichi il deadlock andando a rimuovere la condizione di mutua esclusione per quelle risorse che intrinsecamente possono essere usabili in modo condiviso.
Tale esclusione non può essere invalidata per le risorse intrinsecamente non condivisibili (es. stampante).

Possesso a attesa
Può essere invalidata questa condivisione facendo in modo che ogni volta che un processo richiede risorse, questo non possegga già qualche altra risorsa.
In questo modo non potrà mai trovarsi a detenere risorse e a chiederne altre.
Tecniche per realizzare tale approccio consistono in:

- obbligare un processo a chiedere tutte le risorse prima di partire, in modo che si possa esser sicuri che disponga già di tutto ciò di cui necessita e non debba chiedere altre risorse. Al termine della sua esecuzione esso rilascerà tutte le sue risorse.
- obbligare un processo che già possiede delle risorse e necessita di altre risorse a:
    - rilasciare tutte le risorse di cui dispone
    - richiedere tutte le risorse che gli servono, eventualmente anche quelle che possedeva in precedenza

Una soluzione di questo genere porta ad uno scarso utilizzo delle risorse e inoltre può portare a delle situazioni di starvation.

Nessun rilascio anticipato
La condizione può essere invalidata mediante rilascio anticipato (pre-emption) per risorse il cui stato di uso all'atto di rilascio anticipato sia ripristinabile, in modo che il processo non veda il fatto che la risorsa è stata tolta e concessa ad un altro processo.
Le tecniche per gestire il rilascio anticipato sono:

- se un processo detiene alcune risorse e ne chiede altre che non possono essere assegnate immediatamente:
    - tutte le risorse possedute vengono rilasciate anticipatamente
    - le risorse rilasciate in anticipo sono aggiunte alla lista delle risorse per cui il processo sta aspettando
    - il processo sarà fatto ripartire quando potrà ottenere sia le vecchie che le nuove risorse (chiaramente tutte le risorse devono essere ripristinabili)

- se un processo detiene alcune risorse e ne chiede altre:
    - se tutte le risorse richieste sono disponibili, vengono assegnate
    - se alcune delle risorse richieste non sono disponibili
        - se sono assegnate ad un processo che sta aspettando ulteriori risorse, le risorse richieste e detenute dal processo in attesa vengono rilasciate anticipatamente e assegnate al processo richiedente e inserite tra quelle per cui il processo è in attesa.
    - se alcune risorse richieste non sono disponibili e non sono possedute da processi in attesa di altre risorse, il processo richiedente deve attendere che si liberino e ripartire quando ottiene tutte le risorse necessarie

Attesa circolare
La condizione può essere invalidata impedendo che si creino attese circolari
Le tecniche per gestire ciò sono:

- Un ordinamento globale univoco che viene imposto su tutti i tipi di risorsa R_i

- Se un processo chiede k istanze della risorsa R_j e detiene solo risorse R_i con i < j
    - se le k istanze della risorsa R_j sono disponibili vengono assegnati
    - altrimenti il processo deve attendere

- Se un processo chiede k istanze della risorsa R_j e detiene risorse R_i con i≥j, il processo deve
    - rilasciare tutte le istanze delle risorse R_i
    - chiedere tutte le istanze della risorsa R_j (quelle detenute precedentemente e le nuove k)
    - chiedere le istanze delle risorse R_i (i>j) che deteneva precedentemente

In questo modo un processo non potrà mai chiedere istanze della risorsa R_j se detiene risorse R_i con i > j. Questo garantisce l'assenza di attese circolari

Lezione 3 - Tecniche per evitare il deadlock

Obiettivi

L'obiettivo del deadlock-avoidance è il cercare di aumentare lo sfruttamento delle risorse rispetto al caso della prevenzione, quindi ottenere un alto sfruttamento delle risorse ed efficienza del sistema con una semplicità di gestione (in modo che il sistema non sia sovraccaricato dalla gestione piuttosto che da attività dai processi).

Principio di evitare il deadlock

Verificare a priori se la sequenza di richieste e rilasci di risorse effettuate da un processo porta al deadlock tenendo conto delle sequenze dei processi già accettati nel sistema.
Sostanzialmente si va a vedere se un processo che deve essere accettato nel sistema può avanzare delle richieste che il sistema non è in grado di soddisfare tenendo conto dell'uso delle risorse attualmente in uso dai processi accettati, così da verificare le richieste del processo sono compatibili con richieste e rilascio di risorse per i processi già accettati.

Informazioni per evitare il deadlock

Per effettuare tale analisi è necessario conoscere a priori delle informazioni sul comportamento dei processi:

- numero massimo di risorse per ogni processo
- risorse assegnate
- risorse disponibili
- richieste e rilasci futuri di risorse

Usando queste informazioni si può decidere se il sistema si può trovare in uno stato garantito no-deadlock oppure rischiare di andare in deadlock.
Quindi uno stato si dice **stato sicuro** se il sistema può allocare le risorse richieste da ogni processo in un certo ordine opportuno garantendo che non si verifichi deadlock.
I processi accettati nel sistema verranno ordinati opportunamente in modo che le richieste non comportino mai l'entrata in uno stato di deadlock.
Se lo stato è sicuro siamo certi che non andremo mai in deadlock, altrimenti sarà possibile ricadere in uno stato di deadlock (ma non certo).

Una sequenza di processi $P_1,...,P_n$ è sicura per l'allocazione corrente se le richieste che ogni processo P_i può fare possono essere soddisfatte dalle risorse attualmente disponibili più tutte le risorse detenute dai processi P_j precedenti a i nell'ordinamento.
Lo stato è quindi detto sicuro se esiste una sequenza sicura, quindi se esiste un ordinamento di attivazione dei processi e delle relative richieste per cui si riesce a garantire di evitare il deadlock.

Come evitare il deadlock
Si può essere certi che il sistema non generi deadlock se si garantisce che il sistema passi da uno stato sicuro ad un altro stato sicuro quando un processo chiede una nuova risorsa.

- Si parte da uno stato iniziale sicuro
- Una richiesta di una risorsa viene soddisfatta se la risorsa è disponibile e il sistema va a finire in uno stato sicuro, altrimento se la risorsa non è disponibile il processo attende.

Algoritmo del grafo di allocazione delle risorse
Nel grafo di allocazione delle risorse si aggiunge una nuova tipologia di arco: gli archi di prenotazione. Questi sono come gli archi di richiesta della risorsa, ma sono richieste non diventate effettive.
Lo stato sarà non sicuro quando la presenza di un arco di prenotazione evidenzierà un ciclo tra processi e risorse.
Questo algoritmo prevede che si costruisca il grafo di allocazione delle risorse con gli archi di prenotazione. Se si evidenziano cicli nel grafo lo stato non è sicuro e quindi non può accettare la richiesta di risorse dell'ultimo processo inserito.
Purtroppo questo vale solo per le singole istanze delle risorse.

Algoritmo del banchiere

- Gestisce istanze multiple delle risorse
- È meno efficiente dell'algoritmo del grafo di allocazione delle risorse (dovrà verificare la possibilità di deadlock con più istanze delle risorse)
- Il numero massimo di istanze deve essere dichiarato a priori
- Un processo deve restituire in un tempo finito le risorse utilizzate

Per supportare l'algoritmo del banchiere devono essere introdotte delle strutture dati:

m	risorse
n	processi
Available[1..m]	risorse disponibili (per ogni elemento viene indicato il n. di ist. disp.)
Max[1..n,1..m]	massima richiesta di ogni processo (per ciascuna risorsa, righe: processi, colonne: risorse)
Allocation[1..n,1..m]	risorse attualmente assegnate (indica quante istanze di ciascuna risorsa sono assegnate al processo)
Need[1..n,1..m]	risorse da richiedere (indica quante risorse ciascun processo può richiedere)

Algoritmo di verifica dello stato sicuro

```
Work[1..m]
Finish[1..n]
1. Work=Available; Finish[i]=false per i=0,1,..,n-1
2. Si cerca i tale che:
     – Finish[i]==false
     – NeediWork
     Se non esiste tale i, vai al passo 4
3. Work=Work+Allocation[i]; Finish[i]=true
Vai al passo 2
4. Se, per ogni i, Finish[i]==true, allora lo stato è sicuro
```

Il vettore work è un vettore di risorse disponibili nel sistema, e viene posto = ad "available".

Il vettore finish è un vettore di booleani che dice se il processo che si è considerato è stato completamente gestito, e viene inizializzato a falso.
Si va a cercare un indice “i” tale per cui il processo non è stato gestito e non è stata completata l’allocazione delle risorse, non è stato ancora considerato nell’ordine del sistema, e contemporaneamente la richiesta delle risorse che fa è minore o al più uguale alle richieste disponibili in quel momento.
Il vettore Work viene considerato in modo da modificare le allocazioni in modo temporaneo finché non è stata verificata la sicurezza dello stato che si raggiunge.
Se non esiste un indice “i” che soddisfa queste condizioni allora si va al passo 4, ossia, se per ogni “i” il vettore finish è = a true, si deduce che lo stato è sicuro, altrimenti se esiste un indice “i” che soddisfa le condizioni del passo 2, allora si allocano fittiziamente le risorse richieste dal processo in modo da verificare la sua compatibilità, e così marcare che il processo è stato trattato.
A questo punto si torna al passo 2 per trattare un altro processo, e si procede finché tutti i processi non sono stati trattati.
Se si riesce ad allocare le risorse per tutti i processi allora lo stato è sicuro.

Algoritmo di richiesta delle risorse

1. Se Request[i] <= Need[i], vai al passo 2: ossia la verifica che le risorse richieste siano quelle dichiarate come necessarie per il processore.
   Altrimenti, solleva errore: processo ha ecceduto numero massimo di richieste (oltre il numero dichiarato inizialmente)

2. Se Request[i] <= Available, vai al passo 3: ossia la verifica che le risorse richieste siano minori di quelle disponibili.
   Altrimenti, P_i deve attendere: risorse non disponibili

3. Si ipotizzi di stanziare le risorse richieste:
   Available=Available-Request[i]
   Allocation[i]=Allocation[i]+Request[i]
   Need[i]=Need[i]-Request[i]

Se lo stato risultante è sicuro (viene eseguita la procedura di verifica dello stato sicuro vista sopra), al processo P_i vengono confermate le risorse assegnate
Altrimenti, P_i deve aspettare per le richieste Request[i] e viene ristabilito il vecchio stato di allocazione delle risorse
Questo permette di evitare di far partire un processo senza che questo possa ottenere tutte le risorse di cui necessita.

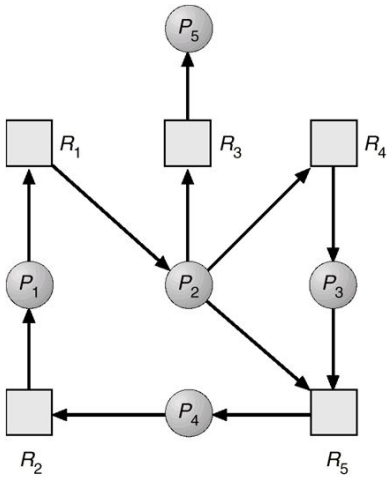
Lezione 4 - Tecniche per rilevazione e ripristino del deadlock

Principio di rilevazione e ripristino

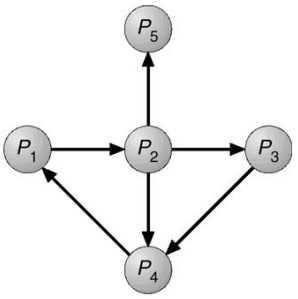
Senza algoritmi di prevenzione o per evitare il deadlock, tale situazione può verificarsi.
Allora Il sistema deve essere in grado di rilevare la presenza di situazioni di deadlock **dopo** che sono avvenute, e quindi ripristinare una situazione di corretto funzionamento eliminando il deadlock.
Con la prevenzione o le tecniche per evitare si agiva **prima**.
Questo approccio vale sia per istanze singole che per istanze multiple delle risorse, anche se poi le specifiche tecniche potranno essere diverse.

Rilevazione per sistemi con solo istanze singole delle risorse

In questo caso dal grafo di allocazione di risorse si può dedurre il *grafo di attesa* (wait-for).



Grado di allocazione delle risorse



Grafo di attesa

Questo grafo ha come nodi i processi attivi nel sistema, mentre gli archi indicano le dipendenze tra i processi dovute al fatto che un processo possiede una risorsa che è richiesta da un altro processo, quindi ad esempio tra P_1 e P_2 si avrà un arco nel grafo di attesa perché la risorsa R_1 sarà posseduta da un processo e richiesta dall'altro.

La rilevazione può essere fatta analizzando il grafo di attesa e se questo contiene cicli si ha deadlock.
I processi in deadlock sono quelli coinvolti in ciascun ciclo presente nel grafo.

Rilevazione per sistemi con istanze multiple delle risorse
In questo caso bisogna creare un algoritmo più complesso che vada a vedere l'effettiva allocazione delle risorse, e che quindi verifichi l'effettiva presenza di deadlock tenendo conto della molteplicità delle istanze.
Si dovranno introdurre le seguenti strutture dati:

`m`	risorse
`n`	processi
`Available[1..m]`	risorse disponibili
`Allocation[1..n,1..m]`	risorse attualmente assegnate per ciascun processo
`Request[1..n,1..m]`	risorse da assegnare per ciascun processo

Algoritmo di rilevazione del deadlock

```
Work[1..m]
Finish[1..n]
1. Work=Available
     Per i=0,1,...,n-1,
          se Allocation[i]≠0, allora Finish[i]=false
          altrimenti Finish[i]=true
2. Si cerca i tale che:
     – Finish[i]==false
     – Request[i]≤Work
     Se non esiste tale i, vai al passo 4
3. Work=Work+Allocation[i];
   Finish[i]=true
     Vai al passo 2
4. Se Finish[i]==false per qualche i, con 0≤i<n,
     allora si ha deadlock
     Se Finish[i]==false, allora il processo Pi è in deadlock
```

Se per il processo non è allocata alcuna risorsa, si segna come “esaminato” (Finish[i] = true).
Allora si valuta se c’è un processo non esaminato le cui richieste sono minori delle risorse disponibili.
Se si ha qualche processo per cui non è stato possibile terminare l’allocazione delle risorse (Finish[i] == false), allora si ha una condizione di deadlock, e il processo i-esimo è quello nella condizione di deadlock.
Se le risorse, invece, sono attribuibili al processo, si aggiorna l’elenco delle risorse disponibili (Work) e si setta il processo come gestito.

Applicazione della rilevazione
Questo algoritmo si deve invocare:

- ogni volta che non si riesce a soddisfare una richiesta di allocazione (e quindi quando si sta per creare un arco d’attesa nel grafo delle richieste).
  In questo modo si può evitare di attivare la situazione di deadlock.
  La rilevazione del deadlock in questo caso è immediata, pochi risorse e processi rimangono coinvolti in situazioni di deadlock e quindi si deve intervenire su poche entità, creando pochi problemi nella prosecuzione della computazione.
  Tuttavia con questa tecnica il carico computazionale per la gestione rischia di diventare notevole, in quanto l’algoritmo è attivato molto frequentemente.

- ad intervalli di tempo definiti. In questo caso la rilevazione è più complessa, le risorse e i processi che possono trovarsi bloccati in deadlock possono essere di più, anche se si ha un sovraccarico computazionale minore.

Ripristino del funzionamento del sistema
Un modo è terminare i processi coinvolti del deadlock, e questo rompe la condizione di deadlock.
Esistono due approcci:

- abortire tutti i processi in deadlock: questo garantisce di tornare privi di condizioni di deadlock. Questo comporta però l’uccisione di molti processi, con conseguente spreco di risorse computazionali (se i processi avevano svolto delle computazioni questa viene resa inutile).
- abortire un processo alla volta fino all’eliminazione del deadlock: si lavora in modo mirato, terminando pochi processi, questo però comporta di attivare più volte l’algoritmo di rilevazione per verificare se avendo eliminato un processo la condizione di deadlock permane o meno.

Terminazione dei processi
Per terminare i processi diventa importante trovare un ordine intelligente di eliminazione dei processi, basato sui criteri di:

- priorità dei processi (vengono eliminati prima i processi di priorità minore)
- tempo di elaborazione (vengono eliminati prima i processi che hanno elaborato per meno tempo)
- uso di risorse (vengono eliminati i processi che usano meno risorse)
- numero di risorse richieste per terminare l’elaborazione (vengono eliminati prima i processi che dovranno richiedere più risorse)
- numero di processi da terminare
- tipologia dei processi (interattivo o batch, il batch da meno impatto sugli utenti)

Rilascio anticipato delle risorse
Un’altra tecnica per ripristinare il funzionamento del sistema potrebbe essere quella di rilasciare anticipatamente le risorse.
Si selezionerà una vittima, ossia un processo il cui costo sarà minimo.
Si effettuerà una rollback dello stato all’ultimo stato sicuro (o un rollback complessivo)
Infine verrà rimossa la risorsa.

Questo approccio può far nascere la starvation, in quanto si potrebbe sempre selezionare la stessa vittima, e potrebbe portare quel processo a non completare mai la sua computazione.