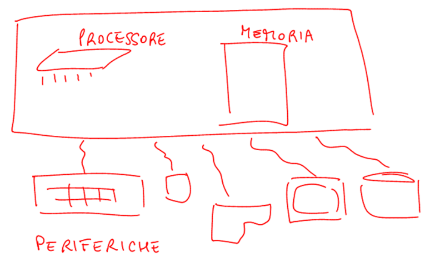
Modulo 1 - Architettura del calcolatore

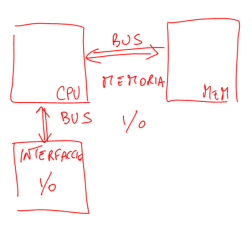
# L1 - Macchina di von Neumann: architettura e funzionamento

## Architettura del calcolatore e macchina von Neumann

L'architettura di un calcolatore è composta dall’unità di elaborazione in grado di eseguire programmi contenuti nella memoria centrale. Quindi l’unità di elaborazione è costituito da un processore e dalla memoria. Le informazioni su quale il processore può operare sono fornite dalle periferiche I/O (es. tastiera, monitor e stampante).



Il modello di von Neumann è composto da processore, memoria centrale e interfacce I/O. Il processore e la memoria centrale si scambiano le informazione attraverso un canale chiamato bus di memoria mentre il processore e le interfacce I/O sono connesse dal bus di I/O.



## Fasi del processore

Il processore è in grado di individuare dalla memoria centrale una istruzione dopo l’altra grazie al Program counter, un registro che individua le parole di memoria desiderate. Questo viene automaticamente incrementato di 1 ad ogni istruzione per individuare l'istruzione successiva da eseguire. Questo meccanismo definisce un supporto hardware per eseguire automaticamente sequenze di istruzioni (o operazioni).

Ad ogni istruzione, il processore esegue tre fasi:

* **Fetch** - Acquisizione operazione

L'operazione individuata viene caricata nel processore durante la fase di Fetch e viene salvata nel registro CIR (Current Instruction Registry).

* **Decode** - Decodifica

L'unità di controllo CU (Control Unit) decodifica e analizza l'ammissibilità dell'operazione.

* **Execute** - Esecuzione

In caso di esito positivo, la CU attiva l'unità aritmetico logica ALU (Arithmetic Logic Unit) in grado di provvedere all'esecuzione dell'istruzione e permette al processore di accedere alle parti di memoria necessarie per prelevare le informazioni necessarie per l'esecuzione dell'istruzione.

Il processore esegue in modo ciclico all’infinito le tre fasi di Fetch, Decode ed Execute in modo sequenziale. In caso di programmi complessi si usano spesso delle frasi condizionali o dei cicli per rieseguire un blocco di operazioni.

## Istruzione jump

Per costituire una struttura così composta è necessario avere la possibilità di saltare da blocchi di memoria ad altri senza perdere informazioni. Avremo quindi bisogno di una istruzione jump che ci permetta di saltare all'interno della memoria in base a delle condizioni poste. Infatti se una condizione è vera (e quindi deve essere effettuato un salto da un blocco all'altro) il PC verrà azzerato e il suo contenuto verrà sostituito con indirizzo della prima istruzione del blocco in cui si vuole saltare. Al termine il processore ripristinerà all'interno del PC il valore dell'istruzione successiva all'istruzione chiamante.

## Interruzioni

Però abbiamo anche bisogno di poter gestire le interruzioni provenienti dal mondo esterno. L'interruzione è il meccanismo hardware che permette di accorgersi dell'avvenire di eventi esterni. Infatti una interruzione bloccherà temporaneamente l'esecuzione del programma attivo per rispondere all'evento esterno che è stato segnalato ("risposta all'interruzione”). Al termine di ciò, il processore dovrà ripristinare lo stato iniziale del programma al momento precedente all'interruzione e quindi ritornerà ad eseguire la prima istruzione successiva al verificarsi dell’evento dell’interruzione.

La verifica del segnale di interruzione dal mondo esterno avviene tra le fasi di Execute e Fetch, quindi prima che inizio un nuovo ciclo. Se è presente, l’esecuzione del programma viene sospesa e il processore passa ad eseguire una serie di operazioni di risposta che gestiscono quel tipo di interruzione. Quando la risposta termina, si ritorna ad eseguire l’istruzione individuata dal PC. Questo permette di trattare eventi esterni in modo asincrono con l’evoluzione della computazione del nostro programma.

# L2 - Chiamate di procedura e risposta alle interruzioni

## Procedura e stack

Una procedura ha un certo numero di parametri formali, e una chiamata di procedura è l'atto di chiamare questa procedura con certi valori dei parametri formali. Per permettere l'esistenza di una procedura, è necessario impostare una struttura dati per gestire la chiamata di una funzione. Questa struttura prende il nome di stack (pila), che è una parte di memoria centrale che è in grado di memorizzare tutte le informazioni necessarie.

La cima dello stack è individuata nel registro Stack Pointer (SP). Le info sullo stack vengono inserite secondo la modalità LIFO, e ciò consente di aprire la procedura, eseguirla , chiamare un'altra procedura senza interrompere la procedura che l'ha chiamata.

## Chiamata di procedura

Quando viene chiamata una procedura, salvo tutto ciò che ho di interessante nella CPU attualmente (PC, valori dei registri e così via) in cima allo stack. Poi, sempre nello stack trovo dello spazio per contenere i valori della procedura: il suo PC, i suoi parametri etc. In questo modo, conoscendo l'indirizzo in memoria della cima dello stack, il codice della procedura saprà con certezza che, in modo relativo alla cima dello stack, il primo parametro si troverà 1 word dopo, il secondo 2 word dopo, l'indirizzo a cui impostare il PC alla fine della procedura in un certo altro posto e così via. In questo modo, creo il contesto di attivazione della procedura, eseguo il suo codice, e sono in grado di tornare poi al codice che ha chiamato la procedura.

Nel contesto di attivazione viene salvato anche l'indirizzo di partenza della porzione di stack che riguarda chi ha chiamato quella procedura. Questo serve per recuperare le variabili. Infatti, le variabili di una procedura vengono cercate prima nella porzione di stack a me allocata, poi se non le trovo vado nella porzione di stack relativa a chiamato me e le cerco lì, se non le trovo ancora vado nella porzione di stack relativa a chi ha chiamato il chiamante etc. etc.

Le variabili locali scompaiono quando la procedura termina: non servono più, e lo spazio che occupavano deve essere reso nuovamente disponibile. Alla fine della procedura, poi, si ripristina il contesto di chi l'aveva chiamata: PC, registri e così via.

## Differenza tra chiamata di procedura e risposta ad interrupt

La differenza tra chiamata di procedura e risposta ad interrupt è la sincronicità. Una procedura viene sempre chiamata con una CALL. La risposta ad un interrupt invece no, non è predicibile quando avverrà (asincrona rispetto al processore).

Quando si risponde ad un'interruzione, si salva tutto il contesto locale e si salta ad un indirizzo che era stato precedentemente associato a quella particolare interruzione. Si esegue il codice di quell'indirizzo, e alla fine si ritorna.

Siccome non c'è nessuna CALL, durante la risposta ad un interrupt in genere si disabilitano gli interrupt, perché il processore non può rispondervi: non saprebbe più tornare al programma originale se un interruzione viene interrotta ancora una volta, proprio perché nessuna istruzione CALL ha salvato il PC del programma originale, e quindi il processore non saprebbe più ripristinare il contesto originale.

# L3 - Memoria

## Registri

All'interno del processore abbiamo dei blocchi di memoria chiamati registri. Il numero e la dimensione dipende dal tipo del processore. L'accesso a questi blocchi è estremamente rapido e diretto, ma la loro capacità di contenere informazioni è molto ridotta.

## Cache

La cache di primo livello è costruita all'interno del processore. Quella di secondo livello appena fuori. Sono leggermente più lente dei registri (ma sempre più veloci rispetto alle altre memorie), ma anche più capienti.

Servono per copiarci la zona di memoria centrale su cui il processore sta lavorando attualmente. In questo modo può leggere e scrivere in modo molto veloce senza passare per i bus. Occorre stare attenti alla coerenza della memoria: se scrivo in cache e non nella porzione di memoria centrale corrispondente, allora quando svuoto la cache devo aggiornare anche la memoria centrale, se no si creano discrepanze. Tutto ciò viene fatto dall'hardware.

## Memoria Centrale

I dati del sistema devono risiedere in memoria centrale per essere eseguiti. Questa è lenta (rispetto alle cache) e il processore può accedere anche in modo diretto e la sua capacità è considerevolmente più grande della cache.

## Memoria di massa

Un’ulteriore memoria è costituita dalle tipologie di memoria di massa: memorie grandi ma lente a causa dei tempi di accesso di tipo meccanico o magnetico.

* i dischi magnetici: dischi metallici con materiale elettromagnetico su cui vanno a scrivere le testine i memorizzazione. Il tempo di accesso è medio, l’accesso agli elementi in esso contenuti può essere sequenziale o sequenziale e la loro capacità è enormemente superiore rispetto alle memorie centrali.
* i dischi ottici: CD, DVD. L’accesso è più lento rispetto ai dischi magnetici, si può avere accesso diretto sequenziale e la capacità può essere grande.
* i nastri magnetici: sono dei dispositivi di backup (dischi magnetici, cassette) caratterizzati da accesso molto lento e di tipo sequenziale, caratterizzati da una capacità molto ampia.

## Caching

Per accelerare l’esecuzione di un programma si può usare il meccanismo di caching che consiste nel copiare porzioni di memoria centrale verso dispositivi più rapidi (memorie cache di primo e secondo livello). Questo meccanismo può essere anche d'aiuto quando il processore deve prelevare con una certa frequenza una parte di memoria centrale: infatti in questo caso, si può evitare di far attendere tutto questo tempo per accedere nella memoria centrale, copiando la parte di memoria interessata dalla memoria di massa alla memoria cache.

Il contenuto della cache deve sempre coincidere con il contenuto della memoria centrale. Se nella cache viene fatta una modifica i dati devono essere aggiornati anche in memoria centrali in modo tale da aggiornare il riferimento della dato che potrebbe essere usato successivamente.

## Protezione della memoria

La protezione della memoria è importante in un ambiente con più programmi caricati in memoria centrale. Infatti si deve confinare ogni programma ad uno spazio determinato in modo da evitare a questo di sovrascrivere dati presenti in memoria che appartengono ad altri programmi.

* registri: la loro protezione è implicita nel cambiamento del programma in esecuzione.
* cache: il contenuto della cache è aggiornato automaticamente da meccanismi hardware e quindi il programmatore non deve preoccuparsi della sua protezione
* memoria centrale: si deve curare il fatto che delle porzioni di memoria in uso dal programma in esecuzione non vengono usate da altri programmi. Ciò viene fatto attraverso un dispositivo chiamato Memory Management Unit.
* memoria di massa: la protezione delle memorie di massa è a cura del file system

Per la memoria centrale si osserva che il processore, quando vuole accedere ad una porzione di memoria centrale, emette un indirizzo desiderato all’interno di tale porzione. Per garantire la correttezza dell’indirizzo (e quindi impedire al programma di accedere a zone di memoria di proprietà di altri programmi) viene in nostro aiuto la MMU. Nel MMU viene inserito un valore di base che corrisponde al primo indirizzo dell’area di memoria riservata al programma in esecuzione.

Esiste anche un secondo registro Limite che individua l’ultimo elemento in memoria centrale occupato dal programma. Quindi prima di accedere direttamente ad un indirizzo, il processore fornisce questo al MMU il quale controlla la correttezza del valore (compreso tra valori di base e valore limite) e in caso positivo restituisce l’indirizzo desiderato alla CPU, altrimenti lancia un’interruzione al processore.

# L4 - Connessione delle periferiche

## Unità centrale

Sappiamo che CPU e memoria sono collegate dal BUS di Memoria e che i dati che una CPU deve elaborare sono prelevati dalla Memoria. I dati da elaborare vengono posti in memoria dalle periferiche di ingresso. Le periferiche comunicano con il resto del calcolatore grazie a delle schede opportune (interfacce) che scambiano dati su un BUS di I/O. L’insieme di CPU, memoria e interfacce prende il nome di Unità centrale.

L’interfaccia di elaborazione è costituita da una parte che si interfaccia con il calcolatore (riceve ed invia comandi e dati) e una parte che si interfaccia con le periferiche. Quest’ultima parte è connessa ad un opportuno cavo che termina sull’elettronica di controllo della periferica.

Questa elettronica di controllo è composta da una parte di gestione di connessione e una parte di controllo della periferica vera e propria (che si occupa della movimentazione di componenti della periferica (mantenimento della rotazione del disco e posizionamento delle testine di lettura e scrittura).

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 quella a bordo dell’interfaccia) costituisce insieme al cavo di connessione della periferica il “canale di comunicazione”, canale attraverso il quale fluiscono le informazioni verso e da il sistema di elaborazione.

## Interfaccia nell'unità centrale

Per gestire tale canale all’interno dell'unità centrale esistono dispositivi di controllo appositi che costituiscono l’interfaccia della periferica. Questa contiene un insieme di registri che permettono il controllo con la periferica e il ritorno dei risultati delle operazioni svolte dalla periferica verso il calcolatore. In particolare abbiamo: un registro di stato, un registro comandi ed un registro dati.

L'unità centrale chiede all’interfaccia di verificare lo stato di comunicazione con la periferica ottenendo il contenuto del registro di stato: se la connessione è ok allora la periferica è libera e il calcolatore potrà quindi proseguire con l’esecuzione di un comando di I/O deponendo all’interno del registro dati l’ordine da fare pervenire alla periferica.

L’unità centrale scriverà nel registro comando l’ordine da inviare e i dati richiesti alla periferica. L’elettronica dell’interfaccia comprenderà che è necessario inviare alla periferica i dati richiesti nel registro comando. Ciò viene effettuato trasferendo il contenuto del registro dati a bordo dell’interfaccia di I/O del calcolatore che controlla la connessione con la periferica, la quale trasferirà l’ordine della richiesta presente nel registro dati all’interno della periferica stessa.

Allora i dati verranno interpretati, verrà eseguito il comando e verrà ritornato eventualmente il risultato all’interno del registro dati, aggiornando il sistema dell’interfaccia della periferica in modo tale che il processore venga informato del risultato della richiesta effettuata.

Il controllo dell’interfaccia è effettuato attraverso due modi:

* attesa attiva: il processore attraverso il bus di i/o parla con l’interfaccia della periferica e va a leggere il contenuto del registro di stato ciclicamente fino a quando scopre che l’operazione è stata effettuata. Solo allora il processore effettuerà le altro operazioni.
* interruzioni: evita le attese del processore. La CPU connessa all’interfaccia provvede ad inviare il comando e procede nello svolgere altre attività. Quando la parte di gestione del canale di comunicazione a bordo dell’interfaccia verso la periferica provvede a modificare il registro di stato segnalando il risultato dell’operazione, porta alla generazione di un interruzione verso il processore in modo tale da notificare a quest’ultimo il risultato avuto dal termine dell’operazione richiesta, in modo tale da procedere all’esecuzine del programma.

## Trasferimento dati DMA

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

* per singola parola di memoria: modo appena visto attraverso il registro dati. Non funziona in maniera efficace ed efficiente quando si devono trasferire tanti dati.
* a blocchi: usando il Direct Memory Access (DMA), consentendo alla CPU di essere più efficiente nel trasferimento di dati, permettendo il trasferimento di un byte dopo l’altro fino al termine del trasferimento.

Un altro modo per velocizzare le operazione di trasferimento di dati consiste nel considerare la periferica mappata in memoria: si ha l'unità centrale, la memoria centrale, l’interfaccia di I/O della periferica in cui una porzione della memoria centrale viene mappata.

In questo caso una porzione di memoria, anziché essere contenuta nella memoria centrale fa riferimento ad una porzione di memoria contenuta nell’interfaccia di I/O . Quindi quando il processore cercherà di accedere a tale porzione di memoria non risponderà la memoria centrale ma l’interfaccia di I/O fornendo l’accesso all’informazione corrispondente all’indirizzo dato.

La porzione di controllo del canale verso la periferica caricherà in questa porzione di memoria posta all’interno della nostra interfaccia tutti i dati che devono essere trasferiti al processore provenienti dalla periferica. Il processore quando vuole inviare dati in blocco alla periferica dovrà scriverli direttamente in questa memoria e quindi sarà l’interfaccia la porzione di controllo del canale che provvederà a trasferire alla periferica il blocco dei dati senza aver bisogno di passare prima dalla memoria centrale.

## Canale di comunicazione

Il canale di comunicazione può' essere gestito da due aspetti:

* controllo nell’interfaccia dell’unita centrale
* controllo nella periferica

All’interno della unità centrale si ha l’interfaccia con una porzione dedicata a gestire il canale. Analogamente nella periferica si ha una porzione di elettronica dedicata a gestire il canale di comunicazione e il cavo di connessione tra i due.

La porzione all’interno del processore provvede a mandare comandi e dati alla periferica e ottenere il risultato delle richieste. Dopo aver gestito il trasferimento delle richieste dal processore all’interfaccia e dopo aver gestito il canale di comunicazione che fa arrivare i comandi desiderati alla periferica è necessario comprendere ed eseguire i comandi ricevuto dal processore.

All’interno dell’elettronica della periferica esiste una elettronica in cui una parte acquisisce comandi e dati provenienti dal processore e un’altra parte in grado di decodificare e comprendere le richieste del comando e restituire il risultato dell’attività svolta.

# L5 - Reti informatiche: architetture e funzionamento

## Reti informatiche

Le reti permettono di supportare la condivisione di risorse informative e fisiche tra calcolatori per ridurre i costi di acquisizione e gestione e al fine di supportare la condivisione di informazioni tra utenti che utilizzano macchine differenti.

Esistono due tipologie di reti:

* reti locali: reti che interconnettono siti relativamente vicini geograficamente (un ufficio o un edificio o edifici vicini). Viene realizzata una connessione con opportuni cablaggi per mettere in comunicazione sistemi in grado di erogare servizi condivisi.
* reti geografiche: interconnettono siti geograficamente molto distanti. Sono costituite da siti nei quali vengono installati degli elaboratori interconnessi da una rete. La connessione è permessa da dispositivi che favoriscono la connessione tra i vari sistemi.

Per realizzare le strutture di rete esistono varie tipologie:

* rete mesh: ogni elaboratore ha una connessione diretta verso ogni altro elaboratore. Molto costosa.
* rete parzialmente connessa: consento no di ottenere ridondanza e quindi capacità di sopravvivere a guasti di linee, abbassando i costi
* rete a stella: una macchina centrale (centro stella) funge da canale di comunicazione verso tutte le altre macchie connesse alla rete
* rete a anello: le macchine sono connesse una all’altra formando un anello
* rete lineare: le macchine sono connessa una all’altra formando una catena aperta

Per la loro realizzazione esistono diverse tecnologie e standard:

* per le reti a bus: Ethernet
* per reti ad anello: Token Ring, FDDI
* reti a stella: cablaggio strutturato
* reti wireless: wifi e bluetooth

## Come viene realizzata l’interconnessione di un calcolatore?

Viene ripercorsa esattamente la struttura vista sopra per la connessione verso le periferiche. In particolare si ha la CPU che è connesso alla memoria centrale (attraverso il BUS di memoria) e l’interfaccia di rete (connessa alla CPU attraverso il BUS di I/O). Quest’ultima contiene un registro di stato e di comando e contiene inoltre una porzione di memoria per effettuare il trasferimento di dati da/verso la rete secondo un meccanismo di periferica mappata in memoria.

Sull’interfaccia di rete esiste la parte di controllo del canale di comunicazione che consente di gestire la connessione alla rete attraverso il cavo di rete. La rete quindi può essere vista dal punto di vista logico come una periferica particolarmente complessa. Infatti dal punto di vista del calcolatore (M1) con CPU, memoria e interfaccia di rete, la connessione alla rete appare come un cavo che esce dal sistema e va a connettersi al sistema di interconnessione della rete, alla quale saranno connesse altre macchine (es. M2, M3).

# L6 - Classificazione dei sistemi di elaborazione

## Mainframe

Il tipo più vecchio dei sistemi di elaborazione è il Mainframe composto da una unità centrale alla quale vengono connessi dispositivi di ingresso e di uscita. Negli anni 60 si usavano come dispositivi di ingresso le schede perforate e come dispositivi di uscita le stampanti.

In seguito, con l’aumentare dei dati (e quindi con l’aumentare della richiesta dello spazio necessario per memorizzarli), vennero introdotti i nastri magnetici (sia in entrata che in uscita). Questo approccio era tipico dei sistemi monoprogrammati (un solo programma in esecuzione) dove ovviamente la CPU era sottoutilizzata (costi di uso molto alti). Una soluzione a questo è stato l’uso di sistemi multiprogrammati in cui la memoria centrale era ripartita tra job (multiprogrammazione, la macchina tiene in memoria contemporaneamente più programmi) e vi era anche la condivisione della CPU (multiprocessing, in cui la CPU esegue contemporaneamente più programmi).

Una evoluzione dei mainframe si è avuta con la multiutenza: grazie ai terminali diversi utenti potevano operare sullo stesso mainframe in cui i dispositivi di input erano le stampanti mentre i dispositivi di output erano e stampanti e i dischi. L’architettura era orientata a supportare molti utenti operanti contemporaneamente, quindi si aveva ora la possibilità di elaborare contemporaneamente più flussi di attività. Questi sistemi interattivi multiutente erano caratterizzati da:

* ripartizione della memoria tra processi (multiprogrammazione)
* condivisione della CPU (multiprocessing)
* gestione CPU in condivisione di tempo (time sharing, si stabilisce che un processo, qualsiasi cosa faccia, usi la CPU per un certo periodo di tempo e poi passi il testimone al processo dopo): crea l’illusione di un sistema complesso dedicato ad ogni utente

## Minicomputer

L’evoluzione ha favorito la riduzione dei mainframe per soddisfare le esigenze dei “piccoli” gruppi di utenti a costi contenuti, portando alla creazione dei Minicomputer. Da qui in poi si assume che tutti i sistemi siano chi più chi meno multiprogrammati, multiprocessing e time sharing.

## Workstation

La tecnologia ha poi portato alla creazione delle Workstation, riducendo la scala dei minicomputer per soddisfare le esigenze di un utente sofisticato a costi contenuti. Questi forniscono una grande potenza di calcolo e per tale motivo sono utilizzati per l’interazione avanzata e l’elaborazione intensiva.

## Personal Computer e palmare

Dall’altra parte la tecnologia ha portato alla creazione di sistemi più piccoli e intelligenti, portando alla creazione dei personal computer. Hanno capacità di grafica e dispositivi di interazione avanzata.

Questo tipo di tecnologia è stata ulteriormente affinata per creare dei dispositivi portatili di dimensioni estremamente ridotti orientati al supporto di attività personali: sistemi palmari e telefoni cellulari. Sono dei sistemi interattivi multiprocesso con ridotto consumo di potenza e basso numero di processi.

## Sistema di elaborazione in tempo reale

Alcuni sistemi particolari sono stati sviluppati per applicazioni specifiche e sono sistemi orientati ad applicazioni in tempo reale utili nel: controllo dei processi industriali, sistemi di automazione industriale, sistemi automazione della casa, sistemi biomedicali, etc…

Il loro obiettivo è quello di dare una risposta agli eventi in tempi brevi. Esistono infatti sistemi in tempo reale stretto (hard real-time) o lasco (soft real-time). L’architettura deve avere quindi la capacità di scambiare dei segnali con il mondo esterno attraverso delle interfacce pensate appositamente.

## Sistema dedicato, multimediali e multiprocessore

Sono stati sviluppati anche sistemi specifici dedicati supportare una sola applicazione e si trovano ad esempio nelle auto o negli elettrodomestici. Spesso gli utenti hanno cercato di avere una forte interazione con modalità di tipo diverso e da ciò nascono i sistemi multimediali.

Per supportare la necessità computazionale di alcuni ambienti sono stati ideati i sistemi multiprocessori: sistemi con una forte capacità di elaborazione che offrono una maggiore affidabilità in caso di guasto. In questi sistemi interattivi multiutente, i processori sono collegati attraverso un BUS di comunicazione con varie CPU connesse al sistema, con una memoria condivisa tra le varie CPU e dalle varie interfacce delle periferiche, collegate dal bus di I/O.

I processori possono tutti svolgere la stessa tipologia di attività e in quel caso si parla di sistema multiprocessore simmetrico, mentre nel caso in cui i processori sono specializzati nell’esecuzione di particolari task, il sistema prende il nome di sistema multiprocessore asimmetrico.

## Cluster e sistemi distribuiti

Nell’architettura a cluster le interconnessioni con il sistema sono più lasche: infatti si ha un sistema di BUS che connette più calcolatori tra di loro in modo abbastanza stretto, ma non così stretto come nel caso del sistema multiprocessore. Questo serve a migliorare ancora le economie di scala sulle periferiche e l’affidabilità del sistema in caso di guasto.

Il sistema distribuito è il sistema in cui sono connessi in maniera lasca diversi calcolatori utilizzando una rete informatica. Anche in questo si ha una grande capacità di elaborazione, un’economia di scala sulle periferiche, maggiore affidabilità in caso di guasti e si realizza con computer disponibili sul mercato (portando all’abbattimento dei costi).

Modulo 2 - Architettura dei sistemi operativi

# L1 - Funzioni di un sistema operativo

Un calcolatore, come abbiamo visto, è composto da CPU, memoria centrale e da un insieme di periferiche che consentono l’interazione con il mondo esterno: questo è l’aspetto hardware che il Sistema Operativo deve gestire.

Gli obiettivi del Sistema Operativo sono principalmente 2:

* Astrazione dei componenti della macchina
* Virtualizzazione delle risorse

## Astrazione

## L’innalzamento del livello di astrazione dei componenti permette di semplificare l’accesso alle risorse e mostrare ai programmi e agli utenti un comportamento più semplice delle varie risorse, nascondendo dietro l’astrazione i dettagli di gestione delle periferiche.

## Questo consente di individuare i dati individuandone un nome in modo logico all’interno del nostro sistema e non più sapendo dove si trovano fisicamente le informazioni nelle risorse. Oltre a ciò il sistema dispone di un insieme di funzioni in grado di interfacciare utenti e programmi al il sistema stesso, costituendo la gestione dell’interfaccia utente.

## Virtualizzazione

## L'obiettivo è creare un immagine del sistema di elaborazione dedicata a ciascun programma in esecuzione anche se in realtà i vari programmi vengono eseguiti un po uno un po l’altro. Questo ha come obiettivo quello di fornire la possibilità ai programmi di non considerare la presenza di altri programmi e ha anche come obiettivo quello di semplificare la programmazione dei software.

Quindi l’insieme delle funzioni di gestione del processore, gestione della memoria e gestione delle periferiche di I/O fanno credere ad ogni programma in esecuzione di avere a sua completa disposizione una macchina di von Neumann completa, anche se in realtà le risorse della macchina sono condivise.

## Gestione del processore e della memoria centrale

Ogni sistema mette a disposizione delle funzioni per realizzare la gestione dei singoli componenti della macchina di von Neumann e per gestire il livello di astrazione logico per reperire le informazioni, costituito dal file system.

Per la gestione del processore è necessario avere ad esempio delle funzioni per la gestione dei processi: creazione e terminazione dei processi, sospensione e riattivazione dei processi, schedulazione dei processi, sincronizzazione tra processi, gestione di situazioni di stallo (deadlock), comunicazione tra processi.

Un altro insieme di funzioni necessarie sono quelle necessarie per la gestione della memoria centrale: queste funzioni costituiscono il supporto per la multiprogrammazione, devono provvedere alla allocazione e deallocazione della memoria ai processi, devono provvedere al caricamento e allo scaricamento di processi e di loro porzioni in memoria centrale, garantendo la protezione della memoria.

## Gestione delle periferiche

Per la gestione delle periferiche si ha bisogno di fornire una omogeneità di interazione: deve consentire la configurazione e inizializzazione, gestione ottimizzata dei dispositivi di ingresso, uscita, memorizzazione di massa e rete informatica, protezione delle periferiche, bufferizzazione e caching.

## File System

Il livello di astrazione superiore che rappresenta il File System e che permette l’accesso alle informazioni in modo logico e non fisico presenti nel sistema, deve permettere di creare file come componente elementare che favorisce il contenimento di informazioni e la gestione di questi:

il file system deve quindi dare all’utente la possibilità di creare e cancellare file e cartelle, leggere e scrivere, copiare, ricercare, proteggere, gestirne l’accounting e permettere il ripristino.

## Gestione dell'interfaccia utente

Interfacciarsi con il SO vuol dire fornire quelle funzioni che permettono di dare comandi e ricevere i risultati. Esiste quindi un interprete di comandi per gli utenti con il quale questi interagiscono e un interprete di comandi per i programmi applicativi (che permettono di effettuare le chiamate di sistema).

Devono quindi essere presenti le funzioni: interprete comandi a livello utente e a livello applicativo (chiamate di sistema), librerie di sistema (modo con il quale i comandi possono interagire con il resto del sistema), gestione delle autenticazioni e gestione degli errori e dei malfunzionamenti.

# L2 - Architetture dei sistemi operativi

## Sistema monolitico

Il più vecchio tipo di architettura di Sistema Operativo è costituito dal Sistema Monolitico, dove il SO era un “contenitore” in cui venivano inserite funzioni senza una specifica strutturazione ma inserendo le chiamate di funzione come poteva essere utile dal punto di vista della programmazione del sistema. Usando questa struttura, le funzioni di livello più basso (funzioni più semplici) potevano richiamare delle funzioni di livello più alto, portando alla non chiarezza delle funzioni di conseguenza la manutenzione era molto difficile.

Questo genere di architettura è stato usato molto fino agli anni settanta in cui i calcolatori usati come i compiti svolti erano piuttosto elementari. Il vantaggio principale di quest’approccio è che il sistema è compatto, veloce ed efficiente, quindi è un approccio adatto a sistemi semplici.

Con l'evolversi delle esigenze e dell'hardware si è reso necessario uno sforzo organizzativo della struttura del sistema operativo.

## Sistema con struttura gerarchica

Tale sforzo si è concentrato nel organizzare le varie funzioni del sistema operativo in gruppi dipendenti fra di loro. In particolare si sono creati vari livelli funzionali dal più vicino alla struttura hardware al più astratto, più vicino cioè all'interfaccia utente. Si è poi stabilito che ogni livello potesse comunicare esclusivamente con livelli di strato inferiore, anche se non necessariamente con quello immediatamente successivo.

Questa nuova organizzazione ha reso più semplice la manutenzione rispetto ai precedenti sistemi monolitici, ma racchiude in sé ancora dei limiti derivanti in particolare dalla non chiara suddivisione in ruoli delle singole funzioni. Inoltre anche la progettazione risulta un compito difficile, in quanto è necessario stabilire a priori a quale livello deve appartenere la funzione da implementare. Se non viene scelto il livello corretto si rischia di rendere tale funzione inaccessibile ad altre di livello diverso (inferiore) che potrebbero averne bisogno.

## Sistema stratificato

Un'evoluzione della struttura gerarchica la si ha con i sistemi stratificati. Questi introducono una chiara separazione modulare delle funzioni svolte da ciascun componente del sistema operativo. Si parte quindi da un modulo di base per la gestione del microprocessore, su cui poggia il modulo per la gestione della memoria che a sua volta funge da base per il modulo di gestione delle periferiche e così via fino al modulo di interfaccia utente.

In questa maniera le varie componenti sono gerarchicamente suddivise in livelli funzionali, che permettono una più semplice manutenzione del sistema, penalizzando però l'efficienza dell'intero sistema, perdendo tempo nel passaggio e nelle comunicazioni tra un livello e l'altro.

## Sistema a microkernel

Per superare queste problematiche si è sviluppato un approccio diverso, in cui vi fosse la possibilità di aggiornare e modificare il sistema senza mai fermarlo (oltre a migliorare l’organizzazione). Allora si è introdotto il concetto di SO a microkernel dove si separano i meccanismi (serie di operazione di gestione di accesso alle singole risorse) dalle politiche (definizione astratto del diritto/ordine dell’uso della risorsa).

Per meccanismi si intendono tutte quelle operazioni, fisse, raramente modificabili, di accesso alle risorse, mentre le politiche rappresentano quel genere di definizioni più astratte di diritto, piuttosto che priorità o ordine di uso di una risorsa. Nasce così il concetto di microkernel ovvero a tutti gli effetti un kernel ma scremato di quelle componenti non strettamente necessarie alla gestione dei meccanismi. Tutto il resto, ovvero tutto ciò che è politica, viene organizzato in processi al di sopra del microkernel.

Con questa struttura i programmi utente in esecuzione non comunicano direttamente con i servizi messi a disposizione dal sistema operativo, ma comunicano indirettamente mediante lo scambio di messaggi con il microkernel. Questo è il limite principale del sistema, perché ne riduce le prestazioni. In compenso si ha un ottima modificabilità e la possibilità di estendere il sistema senza modificarne il kernel, inoltre vi è una maggiore affidabilità in quanto la maggior parte dei servizi funziona in modalità utente.

## Sistema a moduli funzionali

Un'ultima evoluzione dell'architettura dei sistemi operativi, nell'ottica di una sempre più semplice ed efficiente espansione e manutenzione globale del sistema fa uso delle moderne tecniche di programmazione ad oggetti. Ciò permette di costruire kernel modulari a cui è possibile aggiungere o togliere dinamicamente delle componenti a seconda delle esigenze. Si torna quindi ad avere un kernel limitato alle funzioni di base intorno al quale vengono agganciati moduli diversi in base alle esigenze del momento.

Questo approccio ha diversi punti in comune con le soluzioni a strati (le interfacce di interazione con i diversi moduli) ed a microkernel (un kernel con funzioni basilari), è però più efficiente in quanto ogni funzione non è vincolata a richiamar]ne solo di appartenenti a livelli inferiori ed ognuna di esse può comunicare con le altre direttamente, senza scambiare messaggi con il kernel.

## Sistema a macchine virtuali

Per astrarre ulteriormente questo concetto sono state introdotte le macchine virtuali nelle quali si ha una costruzione gerarchica di macchine astratte caratterizzate dall’hardware, al di sopra si ha un kernel della VM che provvede a replicare l’hardware replicandolo esattamente per ogni insieme di processi che si vogliono eseguire sulla macchina virtuale, dove si potrà installare un sistema operativo. Su ciascuno di questi ambienti diversi si avranno dei processi in esecuzione.

Ciascuno vedrà il proprio SO senza sapere che in realtà ci sono altri ambienti operativi disponibili sul sistema. La virtualizzazione permette di usare contemporaneamente diversi operativi.

Questa tecnica non mette a disposizione nessuna nuova funzionalità, ma fornisce un'interfaccia identica all'hardware sottostante gestita dal kernel di macchina virtuale che è in grado di generare più macchine virtuali su cui installare sistemi operativi diversi. Ottenendo così la convivenza di sistemi operativi eterogenei. Il limite più grande lo si ha nel considerevole calo di prestazioni.

## Programmi di sistema

All’interno del sistema si hanno quindi un insieme di programmi che gestiscono le risorse in maniera ottimale (gestione file, informazione sullo stato, esecuzione programmi etc..) e programmi per lo sviluppo di applicazioni (editor, compilatori, linker, debugger etc...).

# L3 - Generazione e avvio di un sistema operativo

## Generare un Sistema Operativo

Generare il SO vuol dire trovare la configurazione del sistema ed applicarla, mentre avviare il SO vuol dire caricarlo in memoria centrale all’accensione del sistema e fare in modo che il SO prenda il controllo della macchina.

Generare un SO è una attività complessa; si tratta di individuare le caratteristiche dell’ambiente dove si vogliono fare operare i programmi e gli utenti per quella specifica installazione e quindi capire le caratteristiche degli utenti, dei programmi, i carichi di lavoro che essi generano etc.. per garantire una equa ripartizione dell’uso delle risorse tra i vari utenti e per gestire le risorse in maniera efficiente. E’ necessario applicare questi parametri per caricare il codice eseguibile per caricarlo poi in memoria e farlo partire.

Identificare le caratteristiche di un ambiente richiede una accurata analisi dei programmi da usare, del carico di lavoro e l’utilizzo di risorse. Si analizzano anche gli utenti che li useranno per capire meglio i carichi di lavoro. Valutare le caratteristiche del carico di lavoro consiste nel mettere insieme le info raccolte sul modo di uso per creare un modello che descriva il carico di lavoro e di richieste per le varie risorse, e quindi valutare le caratteristiche dell’hardware e del SO in modo che quest’ultimo risponda in maniera efficace ed efficiente. La raccolta di tali informazioni può essere fatta manualmente, in un ambiente operativo simulato o in maniera automatica nell’ambiente operativo reale in funzionamento effettivo monitorando le attività degli utenti. Valutare le caratteristiche dello scenario operativo può essere fatto in base all’esperienza o su base statistica.

Per definire i parametri del SO è necessario analizzare il modello appena costruito e valutare quali parametri possono garantire/rispettare le richieste degli utenti e delle applicazioni: può essere fatto attraverso una valutazione manuale in base all’esperienza, manuale su basi statistiche, automatica in base a casi predefiniti, automatica in base a regole.

Fatto ciò si passa all’applicazione di tali parametri andando a modificare i file di configurazione del sistema operativo, eseguendo poi i programmi che generano il codice eseguibile dei moduli modificati e generando il codice eseguibile complessivo e dei programmi di sistema.

Infine l’aggiornamento del sistema consiste nella memorizzazione del nuova versione del sistema operativo e dei programmi di sistema e il conseguente caricamento di questo in memoria centrale.

## Avvio di un Sistema Operativo

Avviare un sistema operativo (bootstrap) è una operazione che può essere realizzata in vari modi. I metodi di avviamento del sistema sono vari e possono essere svolti in uno o più passi e ciò consente di avere diversi gradi di modificabilità ed efficienza del SO. I metodi di avviamento si dividono in: 1 passo, 2 passi, 3 passi.

E’ possibile pensare ad ulteriori passi per favorire una maggiore possibilità di aggiornamento del SO ma ciò porterebbe ad un rallentamento del caricamento del SO.

### Bootstrap ad 1 passo

In memoria centrale si considerano due porzioni: RAM (in cui è possibile leggere e scrivere) e ROM (memoria a sola lettura che contiene dei valori permanenti).

Il SO è presente in ROM e questo viene letto immediatamente all’accensione del computer. Essendo il SO salvato nella memoria centrale, il suo caricamento è rapidissimo ma essendo la ROM a sola lettura, il SO non è modificabile. E’ utilizzato specialmente nei Sistemi Embedded.

### Bootstrap in 2 passi

Riduce la complessità dell’aggiornamento favorendo l’aggiornamento del SO. In questo caso il SO non è tutto nella ROM. Infatti in essa è presente solo un loader del SO. Esso si occuperà nella prima fase di inizializzare l’hardware e reperire su un’altra memoria di massa facilmente modificabile (in una posizione ben determinata) il loader del SO.

Quindi il SO è modificabile mentre il loader non lo è (ROM). La seconda fase consiste nel caricare in memoria centrale il resto del SO dal disco rigido.

### Bootstrap in 3 passi

L’avviamento in tre passi vuole superare il limite imposto dal fatto che il SO deve stare in un solo settore specifico del disco (quello memorizzato nel loader).

L’idea è quella di avere in memoria centrale un piccolo loader posto in ROM che provvede a mantenere un riferimento alla porzione del disco rigido che contiene il loader vero e proprio del sistema operativo. Questo verrà posto in RAM dal caricatore di base e sarà questo a sapere dove si trovano le varie porzioni di SO che devono essere caricate in memoria centrale. Il terzo passo consisterà nel caricare il SO in RAM.

In particolare, i passi eseguiti dal bootstrap sono i seguenti:

1. il caricatore elementare viene caricato in memoria ed eseguito. Si trova da qualche parte in ROM.
2. il caricatore elementare carica in memoria il caricatore complesso, NON il SO. Questo è il cosiddetto loader di 2° livello. Questo loader può cercare SO ovunque nei dischi, non è limitato al primo settore del primo disco o cose simili
3. infine il caricatore del SO si preoccupa di avviare il SO stesso, ed è possibile passargli parametri per caricare moduli opzionali o simili.

Si usa questo sistema per dare l'opportunità di caricare più sistemi operativi, magari da un menu, o anche quando un SO è abbastanza grande da non starci nel settore iniziale di un disco. Ecco perché serve un caricatore secondario. Il caricatore secondario può anche offrire un'interfaccia all'utente, come LILO o GRUB.

Da notare che non è necessario, per avere l'opzione di bootare più sistemi operativi sulla stessa macchina, avere un bootloader in 3 passi. Un bootloader caricato in ROM ed opportunamente programmato può offrire la stessa funzionalità.

# L4 - Interfacce dei sistemi operativi

## Interfacce

L’interfaccia utente è costituita dall’interprete dei comandi fornita all’utente (shell) che rimane in attesa di un comando e rimane in una attesa infinita fintanto che l’utente non dichiara di terminare il comando (fetch). Allora l’interprete analizzerà e verificherà la sintassi del comando (decode) e in caso di esito positivo passerà all’esecuzione dello stesso (execute) attivando un processo ausiliario. Al termine dell’esecuzione del processo il controllo ritorna all’interfaccia utente.

L’interazione con l’utente può essere testuale o grafica:

* testuale: l’utente deve digitare il comando e i parametri in una shell
* interfaccia grafica: si basa su icone e menu che permettono di selezionare i comando e i parametri, minimizzando la possibilità di effettuare errori di sintassi/errori di digitazione.

## Interfacce programmative

L’interfaccia programmativa permette ai programmi di effettuare delle chiamate di sistema (system call) il cui obiettivo è quello di proteggere il sistema operativo, garantire l’integrità dei dati, garantire l’esecuzione completa delle procedure di sistema operativo e garantire l’esecuzione dei controlli nelle procedure di sistema.

Le risorse vanno gestite bene, e a questo scopo è importante cercare di proibire ad un programmatore di bypassare il SO per accedere direttamente all'hardware.

I vantaggi di chiamare il SO invece di accedere direttamente all'hardware sono molteplici:

* il programmatore non deve occuparsi dei dettagli di 1000 tipi di hardware diverso;
* non si devono reimplementare ogni volta le stesse funzioni;
* se le funzioni sono complesse, si può sbagliare a scriverle, se non si è esperti in quel ramo.

Le chiamate di sistema non sono normali chiamate di procedura, anche se concettualmente lo sono. Ogni volta che ne viene fatta una, si controllano ben bene i permessi di chi le sta chiamando. Per evitare che il programmatore possa mettersi a fare Call a indirizzi arbitrari della memoria riservata al sistema operativo (e quindi esporre lo stesso a grandi rischi), bisogna che non si usi una semplice call.

## Meccanismo delle Trap

A questo scopo, si usa il meccanismo delle TRAP. È un meccanismo che genera interruzioni via software, del tutto simili alle interruzioni che arrivano al processore dalle periferiche hardware.

Per scoprire chi ha chiamato la TRAP, si controlla lo stack delle chiamate. Il sistema operativo espone i nomi di funzioni e li associa agli indirizzi delle trap, ma non espone direttamente l'indirizzo del codice delle funzioni che espone.

Trattandosi di interruzioni software sono sincrone con l'evoluzione della computazione, al contrario delle interruzioni hardware che sono asincrone per natura.

### Esempio

Supponiamo di avere un programma P che vuole chiamare una procedura di sistema F. Il processore ha un insieme di registri in cui il programma P andrà a caricare un insieme di informazioni che identificano la system call desiderata e i parametri che dovranno essere usati da tale procedura.

A questo punto il programma chiama una istruzione di TRAP e questa durante la sua esecuzione porta il processore ad attivare una interruzione forzata (via software). Il SO stesso attiva al suo interno una interruzione alla quale assocerà una risposta alle interruzione e per questa interruzione specifica il sistema andrà a definire un modo di identificare la procedura richiesta da P e andrà poi ad eseguirla.

La risposta generata dalla TRAP va in una tabella del SO in cui sono elencate tutte le system call e dove queste sono memorizzate in memoria centrale nello spazio riservato al sistema stesso. Sostanzialmente con la TRAP si va a questo “interprete comandi” che contiene la tabella delle system call e questo interprete provvederà a chiamare la funzione F desiderata.

Terminata la sua esecuzione di ritorna all’interprete comandi, che ritorna il risultato alla TRAP che termina a sua volta restituendo il controllo al programma P che l’aveva chiamata.

Questo meccanismo garantisce che non si possa accedere direttamente alle istruzioni contenute in una system call perché non è il processo a chiamarla ma è il SO stesso ad effettuare la chiamata alla procedura desiderata. Questo disaccoppiamento garantisce che P possa solo dire il nome della funzione e ma accede alla stessa se non attraverso l’interprete comandi della trap previsto dal SO.

### Riepilogo

1. P carica in un registro della CPU le informazioni e i parametri della procedura desiderata
2. P chiama la TRAP
3. la TRAP porta il processore ad attivare una interruzione
4. il sistema attiverà una risposta all’interruzione, portando la trap a chiamare all’interprete comandi I che contiene la tabella con tutte le system call
5. L’interprete I provvederà a leggere dalla sua tabella (in memoria centrale) e ad eseguire la procedura F desiderata
6. l termine di F si ritorna all’interprete I che restituisce il controllo alla TRAP
7. la TRAP quindi termina la sua esecuzione ritornando al programma che l’aveva chiamata.

Modulo 3 - Gestione del processore

UD1 Processi

# L1 - Processi

## Efficienza sistema elaborazione

Un sistema di elaborazione vede sfruttare poco il processore perché le operazione di I/O sono più lente rispetto alle operazioni del processore. Questa lentezza fa si che il processore rimanga in attesa e quindi questo non può effettuare altre computazioni. Una soluzione è quella di eseguire più programmi contemporaneamente (almeno in apparenza): quindi si esegue un altro programma quando si deve attendere per ottenere il risultato di altre operazioni.

La realizzazione di ciò si ottiene con:

* multiprogrammazione. Tecnica che permette di avere più programmi in memoria centrale da eseguire
* multitasking. Tecnica che permette di avere una turnazione dei programmi sul processore quando uno di questi programmi entra in attesa del completamento delle operazioni da parte delle periferiche (esempio, un programma attende l'I/O da qualcosa, viene messo in pausa e viene chiamato un altro dei programmi che sono in ballo in quel momento).

Il processore deve quindi essere in grado di gestire più flussi di esecuzione indipendenti (apparentemente in parallelo).

## Processo

I programmi in esecuzione di cui abbiamo appena parlato sono i processi, l'unità di lavoro nei moderni sistemi a condivisione di tempo (time-sharing). Sono composti dal codice del programma (anche detto sezione di codice) e dai dati del programma, che a loro volta si suddividono in:

* variabili globali, allocate in memoria centrale nell'area dati globali
* variabili locali e non locali delle procedure del programma, memorizzate in uno stack
* variabili temporanee introdotte dal compilatore, tra cui ricordiamo il program counter, caricate nei registri del processore
* variabili allocate dinamicamente durante l'esecuzione, memorizzate in uno heap

Importantissimo sottolineare che i programmi non sono processi. I primi sono infatti entità passive, una sequenza di istruzioni contenute in un file sorgente salvato su disco. I processi sono invece entità attive, con un program counter che specifica l'istruzione successiva da eseguire, un gruppo di istruzioni in uso e una particolare istanza dei dati su cui era stato mandato in esecuzione il programma.

E' per questo che posso benissimo avere due processi associati allo stesso programma, verranno comunque considerati come istanze di esecuzione distinte dello stesso codice. Ad esempio, posso aprire due finestre di Firefox e gestirle separatamente: il loro sorgente è uguale, la sezione dati è differente.

## Processi = flussi di esecuzione della computazione

Un altro modo di pensare i processi è come flussi di esecuzione della computazione. Indipendentemente da cosa fanno o rappresentano, possiamo intuitivamente sostenere che se due flussi sono separati, anche i processi lo sono, e che in questo caso potrebbero evolversi coordinandosi (processi sincronizzati) o in modo assolutamente autonomo (processi indipendenti).

In base ai flussi posso inoltre distinguere due modelli di computazione, il processo monolitico e quello cooperativo. Nel primo vengono eseguite tutte le istruzioni del programma in un' unico flusso dall'inizio alla fine; mentre nel secondo vengono generati una serie di processi concorrenti che lavorano (a livello logico) in parallelo per conseguire lo stesso scopo.

Da notare come i processi monolitici siano tra loro indipendenti, mentre i vari processi che computano a livello cooperativo interagiscono tra loro (ad esempio condividendo i risultati, sincronizzandosi, ecc). La realizzazione di tale modelli di computazione può dunque essere di tre tipi: programma monolitico eseguito come tale, programma monolitico che genera processi cooperanti, programmi separati eseguiti come cooperanti.

## Evoluzione della computazione di un processo

Lo stato di evoluzione della computazione di un processo è l’insieme di tutte le informazioni da cui dipende l’evoluzione della computazione del processo e quindi l’insieme di: variabili globali, variabili locali e non locali (stack), variabili temporanee introdotte dal compilatore, variabili allocate dinamicamente (heap), informazioni di gestione del contesto della chiamata di procedure (indirizzo di ritorno, base pointer, stack pointer) e istruzione corrente.

Si parla di evoluzione perché durante l'esecuzione del processo avviene una trasformazione delle informazioni. Lo si può dunque immaginare come una funzione, in cui su valori iniziali vengono eseguite delle operazioni che produrranno un risultato finale, o come una macchina a stati finiti, dove gli stati sono le informazioni su cui opera e le transizioni le istruzioni che li modificano.

## Stato dei processi

Un processo in esecuzione può assumere diversi stati rispetto all'uso della CPU. Notare bene che non si sta più parlando di stati di computazione, ma stati di uso del processore da parte di un processo, o in breve stati del processo, che rappresentano la modalità di uso corrente del processore e delle risorse da parte del processo.

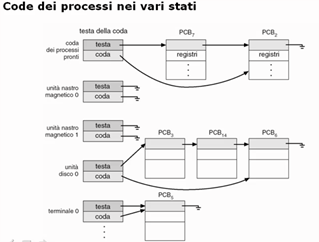
Ciascun processo può trovarsi in uno dei seguenti stati:

* **new** (nuovo), se è stato appena creato e inizializzato
* **running** (in esecuzione), se le istruzioni sono eseguite regolarmente e la computazione evolve effettivamente. In ogni istante, solo un processo per processore può essere in questo stato
* **waiting** (in attesa), se il processo sta aspettando il verificarsi di qualche evento, ad esempio che gli vengano assegnate delle risorse o il completamento di un'operazione di I/O
* **ready-to-run** (pronto all'esecuzione), se ha tutte le risorse necessarie allo svolgimento delle sue attività, eccetto la CPU da cui aspetta di essere chiamato. La computazione potrebbe dunque evolvere, ma non lo fa perché le istruzioni non possono essere eseguite dal processore
* **terminated** (terminato), se ha terminato l'esecuzione e sta aspettando che il sistema operativo rilasci le risorse che utilizzava e lo rimuova dalla memoria (il che non avviene sempre immediatamente).

Per descrivere l’evoluzione degli stati di un processo si introduce il diagramma di stato del processo, che rappresenta l’insieme degli stati del processo e le transizioni tra stati. E’ un grafo orientato in cui i nodi sono gli stati del processo e gli archi sono le transizioni tra gli stati del processo.



## Process Control Block

Per gestire i processi il SO mette a disposizione il Process Control Block che raccoglie tutte le informazioni sul processo a supporto della gestione del processore:

* pid: identificatore del processo
* stato del processo
* program counter
* registri della CPU
* informazioni per la schedulazione della CPU
* informazioni per la gestione della memoria centrale (limiti di memoria)
* informazioni sullo stato dell’I/O

Questi descrittori che raccolgono le informazioni sui processi possono essere accodate nelle code che rappresentano gli stati del processo.

E’ possibile modellare l’evoluzione della computazione di un processo e la transizione tra vari stati con un modello a code dove il processore è un servizio che riceve dei valori da una coda dei processi pronti per l’esecuzione.

## Cambio di contesto

Quando si vuole cambiare il processo in esecuzione è necessario salvare lo stato di evoluzione della computazione per garantire la corretta esecuzione, indipendentemente dagli eventuali cambiamenti dei processi in esecuzione. Questa fase è nota come cambio di contesto (context switch), dove per contesto di un processo si intendono alcune informazioni contenute nel suo specifico PCB, come il valore dei registri, il suo stato, lo stack pointer, ecc.

Per fare ciò è quindi necessario salvare il contenuto dei registri del processore per garantire il ripristino delle condizioni di partenza (tra i quali il program counter) e possono essere salvati sulla cima dello stack o su una struttura dati differente.

Inoltre è necessario salvare lo stack pointer per far sapere dove trovare esattamente la cima dello stack da cui recuperare le variabili da ricaricare nei registri e l’indirizzo dell’istruzione da caricare nel Program Counter. Lo stack pointer non può essere salvato nello stack, perché punterebbe a se stesso. Va salvato quindi in una particolare tabella mantenuta dal SO.

# L2 - Creazione e terminazione dei processi

## Processi monolitici o cooperanti

Un processo è un flusso di esecuzione di computazione. Se si hanno delle attività separate si possono avere flussi separati e si avranno quindi dei processi separati.

Si possono avere dei flussi completamente indipendenti e quindi posso vedere dei processi che operano in modo indipendente l’uno dall’altro (processi che evolvono in modo autonomo indipendentemente l’uno dall’altro). In molti altri casi la risoluzione di un problema complesso può essere visto come un insieme di flussi sincronizzati e coordinati per raggiungere un obiettivo globale (insieme di processi sincronizzati tra di loro). Si possono avere infine un insieme di flussi di esecuzione separati ma che devono comunque essere sincronizzati perché vanno ad utilizzare delle risorse condivise che devono essere usate in modo di mutua esclusiva.

Si può quindi modellare la computazione di una applicazione in vari modi. Dal punto di vista astratto un'applicazione può essere vista come un unico processo (processo monolitico) o come processi cooperanti.

Dal punto di vista della realizzazione del software si può ottenere il modello di esecuzione dell’applicazione in vari modi: infatti si può realizzare un programma monolitico (che può essere realizzato usando un modello computazionale monolitico (monoprocesso) o a processi cooperanti) oppure come un insieme di programmi separati (eseguiti come processi cooperanti) che vengono mandati in esecuzione in modo coordinato per risolvere l’applicazione.

## Generare un processo

La generazione del processo avviene attraverso una chiamata al sistema operativo (chiamata fork) che crea e attiva un nuovo processo. Quindi il processo generante prende il nome di processo padre mentre il processo generato prende il nome di processo figlio.

Le risorse possono essere condivise tra padre e figlio, parzialmente condivise o completamente indipendenti. Tra il padre e i figli e necessario fornire l’insieme dei dati di inizializzazione per definire l’esecuzione dei vari processi.

Lo spazio di indirizzamento del processo figlio è sempre differente da quello del processo padre e ciò permette di garantire la separazione e la sicurezza dei dati.

Lo spazio del processo figlio può essere una copia esatta dello spazio del processo padre in modo tale che il processo figlio e da tutto quello che il processo padre è in grado di vedere.

Quando il processo figlio inizia la sua attività, i dati, lo heap e lo stack del figlio evolveranno in maniera autonoma rispetto a quelli del processo padre, infatti il processo figlio sarà completamente indipendente dal processo padre.

Un’altra tecnica usata è quello di far sì che il processo padre abbia un suo spazio di indirizzamento e il processo figlio non erediti nulla: quindi il figlio avrà uno spazio completamente nuovo caratterizzato da un nuovo codice eseguibile (caricato grazie alla system call exec()). Quindi il programma figlio sarà diverso dal programma padre.

Questo modello di memoria separata può essere difficile da usare se si vuole condividere delle porzioni di memoria centrale e quindi Il SO deve garantire delle funzioni che permettono ciò.

L’esecuzione dei processi può avvenire in due modi:

* il processo padre può eseguire la sua computazione, genera il figlio e continua nella propria computazione mentre il figlio parte con la sua computazione separata (padre e figlio procedono indipendentemente)
* il processo padre può provvedere ad attivare il figlio e sospendersi affinché il figlio termini la propria computazione. Questo è reso possibile da una system call chiamata wait(). All’avvio di un SO si ha un processo radice dell’albero dei processi che provvede ad attivare i processi di sistema che gestiscono l’ambiente operativo per gli utenti: tutti i processi che permettono di caricare/scaricare i programmi dalla memoria di massa, programmi per la gestione delle stampanti etc.. 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 una interruzione al SO segnalando che un utente vuole lavorare e si occuperà di lanciare un processo di shell che prenderà in input i vari comandi dell’utente e li lancerà.

## Terminare un processo

Durante la loro attività i processi svolgono operazioni che realizzano la computazione del programma applicativo per i risolvere i problemi applicativi e possono terminare quando ad esempio viene raggiunta l’ultima istruzione del flusso e in tal caso il processo termina con una istruzione exit() che restituisce al processo chiamante un valore di stato e dealloca le risorse usate dal processo.

Un processo può anche terminare se si verificano comportamenti anomali: eccessivo uso di una risorse o esecuzione di un compito non più necessario.

Può anche avvenire una terminazione a cascata attraverso la funzione abort() che porta alla morte del processo padre e di conseguenza alla morte dei processi figli.

# 

# L3 - Sospensione e riattivazione dei processi

## Tipologie di processo

Sospendere e riattivare i processi è l'anima del multi-tasking, il cui obiettivo è consentirne la turnazione sul processore massimizzandone lo sfruttamento. Ciò viene realizzato secondo una precisa metodologia:

* sospensione del processo in esecuzione, salvando in modo sicuro lo stato di evoluzione della computazione in modo tale da poter tornare ad eseguire il processo dallo stesso punto in cui l'avevo lasciato
* ordinamento dei processi in stato di pronto (scheduling), per stabilire quale deve essere eseguito per primo selezione del processo in stato di pronto da mettere in esecuzione (dispatching), il passo logico successivo allo
* scheduling
* riattivazione del processo selezionato

Tali operazioni devono tener conto del comportamento dei processi rispetto all'uso delle risorse fisiche, in base al quale li distinguiamo in:

* I/O-bound, che effettuano più I/O che computazioni
* CPU-bound, che effettuano principalmente computazioni (in casi limite potrebbero anche non avere alcuna I/O)

Il sistema che implementerà il multi-tasking dovrà dunque tener conto delle classi dei processi, bilanciandoli opportunamente.

Ad esempio, eseguire pochi processi I/O-bound non sfrutterebbe al meglio il processore, dandogli poco e niente da computare; eseguire troppi CPU-bound invece, pur assicurando un uso intensivo della CPU, rallenterà drasticamente il sistema (il processo monopolizza il processore e tutti gli altri non potranno evolvere). Quindi non basta massimizzare lo sfruttamento della CPU, occorre anche che l'evoluzione dei programmi in esecuzioni appaia fluida e parallela all'utente.

## Politiche di sospensione dei processi

Quindi vanno definite delle politiche da utilizzare. In funzione delle regole scelte si avranno dei risultati diversi dal punto di vista dell’evoluzione della computazione dei processi. Oltre alle politiche esistono dei meccanismi per realizzare gli obiettivi di cui prima come:

* la sospensione del processo in esecuzione con salvataggio del suo contesto di esecuzione
* dispatching del processo da mettere in esecuzione
* riattivazione di un processo con ripristino del suo contesto di esecuzione

Le politiche di sospensione dei processi sono differenti; il processo può essere sospeso:

* in modo implicito (ad esempio quando andiamo ad eseguire operazioni di I/O o dopo aver creato un sottoprocesso attendendo la terminazione)
* in modo esplicito (quando il programma stesso chiama una system call che provoca il rilascio del processo per consentire ad altri processi di utilizzarlo)

In questi casi la sospensione è sincrona con l’evoluzione della computazione in procedure del sistema operativo.

## Time sharing

Il time-sharing è un multitasking a condivisione di tempo che ha come obiettivo quello di creare una illusione di una evoluzione contemporanea dei processi agli occhi degli utenti. Il problema che si ha in questo caso è che i processi legati alla CPU tendono a non rilasciare il processore abbastanza frequentemente da permettere tale illusione. La soluzione è quella di forzare il rilascio del processore attraverso di meccanismi di pre-emption.

Si introduce quindi il concetto di quanto di tempo (o time slice) che rappresenta l’intervallo di tempo massimo di uso consecutivo del processore consentito a ciascun processo. Questo permette una equa ripartizione della CPU tra i vari processi creando l’illusione parallela di questi.

Se un processo P1 non riesce a terminare le sue operazione entro il quanto di tempo stabilito, esso subirà un rilascio forzato del processore attraverso la pre-emption.

A questo punto il processore verrà assegnato ad un altro processo P2 pronto e anche in questo caso P2 potrà usare il processore per un tempo Delta pari al quanto di tempo e in seguito verrà forzato il rilascio del processore e così fino al termine delle operazioni del processo.

Per effettuare questo meccanismo di valutazione del time slice, esiste un dispositivo all’interno del calcolatore, il Real Time Clock, che scandisce il tempo (quindi stabilisce la durata del quanto di tempo) e genera periodicamente una interruzione.

Il problema è che usualmente il periodo di RTC è troppo breve portando quindi ad un sovraccarico di gestione delle interruzioni.

La soluzione è quella di trovare un periodo di tempo (un quanto di tempo) che non sia esattamente coincidente con il periodo di generazione del RTC ma sia sufficientemente lungo da dare l’illusione della turnazione frequente dei processi senza sovraccaricare eccessivamente il processore e ciò viene fatto prendendo un multiplo del periodo del RTC.

Per quando riguarda le politiche di sospensione dei processi in un ambiente di time sharing possiamo dire che il processo in esecuzione viene sospeso:

1. dopo aver effettuato una richiesta di I/O
2. dopo aver creato un sottoprocesso attendendo la terminazione
3. quando rilascia volontariamente il processore
4. quando scade il quanto di tempo

Rispetto all’evoluzione della computazione si ha una sospensione sincrona nei casi 1,2,3 e una sospensione asincrona del processo nel caso 4 (non c’è infatti una correlazione tra la sequenza delle operazioni svolte da un processo e il momento in cui si presenta un'interruzione dovuta allo scadere del quanto di tempo a disposizione).

Mentre rispetto alla scrittura del programma si ha una sospensione implicita nel sistema nei punti 1,2,4 e una sospensione esplicita nel caso 3 (infatti se vogliamo che n processo rilasci processore dobbiamo chiamare una procedura in modo esplicito).

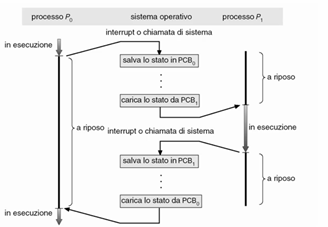
## Meccanismi di sospensione dei processi

Sospendere un processo in esecuzione vuol dire:

* attivare una procedura di sospensione: che è sincrona rispetto all’evoluzione della computazione in stato di supervisore (in procedure di I/O o creazione processi), sincrona rispetto all’evoluzione della computazione in stato utente (nel rilascio volontario delle risorse) o asincrona allo scadere del time-slice nel time-sharing
* salvare il contesto di esecuzione: per garantire che l’esecuzione riparta esattamente da dove era stata interrotta salvando tutti i registri del processore sullo stack e salvare lo Stack Pointer nel Process Control Block

Riattivare un processo vuol dire prendere il contesto di esecuzione salvato e ricaricarlo. Quindi i passi necessari sono:

* Ripristinare il valore del registro Base Pointer prelevandolo dal Process Control Block del processo da riattivare
* Ripristinare il valore dello stack Pointer prelevandolo dal Process Control Block del processo da riattivare
* Ripristinare tutti i registri del processore prendendoli dallo stack

Cambiare il processo di esecuzione (e quindi cambiare il contesto del processo in esecuzione) vuol dire sospendere il processo in esecuzione e riattivare il processo selezionato da mettere in esecuzione. 

Quindi un processo P0 in esecuzione viene interrotto da una chiamata di sistema (o da una interrupt) che salva lo stato del processo, va a prendere lo stato del nuovo processo da mettere in esecuzione (P1), lo si mette in esecuzione e fintanto che questo non è obbligato a rilasciare il processore, questo lavora mentre l’altro è in attesa.

Quando terminerà la sua esecuzione si salverà il suo stato verrà caricato lo stato del processo P0 ricostruendo esattamente lo stato di esecuzione e l’ambiente che si aveva quando il processo era stato sospeso.

Il Dispatching del processo consiste nel prendere il primo processo presente nella coda dei processi in stato di pronto (generata dalla schedulazione dei processi) e porlo in esecuzione

UD2 Thread

# L1 - Thread

## Cosa sono i thread?

Una applicazione esegue tipicamente tre tipi di attività: il controllo del flusso di operazione, le operazioni di I/O, elaborazione all’interno del processore. In applicazioni in cui è necessario avere una alta disponibilità del servizio e un basso tempo di risposta è necessario porre particolare attenzione a queste tre funzioni.

I problemi che nascono in questo caso sono molteplici, infatti:

* è necessario avere più flussi di controllo nello stesso processo per svolgere attività simili
* bisogna stare attenti alla gestione delle operazioni di I/O perché questa attesa può bloccare l’esecuzione del programma
* è utile avere la possibilità di condividere la memoria centrale come mezzo rapido per condividere le informazioni e ridurre il tempo di scambio di tali informazioni.

## Monothread vs Multithread

L’idea sarebbe avere dei flussi di controllo di esecuzione delle istruzioni di un programma e ciò è proprio l’obiettivo dei thread, che rappresentano quindi l’unità di base di utilizzo della CPU.

Il thread può essere anche considerato come l'unità base dell'utilizzo della CPU, e comprende un identificatore, un program counter, un set di registri e uno stack, e condivide con gli altri thread che appartengono allo stesso processo la sezione di codice, quella dei dati e altre risorse del sistema operativo.

Un processo mono-thread è caratterizzato da un solo flusso di controllo mentre oggi sono molto utilizzati i processi multi-thread, caratterizzati da più thread che operano contemporaneamente che condividono le informazioni in memoria centrale.

In un processo multi thread, i vari thread condividono il codice (unico per tutti i thread del processo), i file su cui possono operare e i date globali su cui operare (anche se ogni thread ha un proprio contesto di esecuzione). Ogni thread dispone di una sua copia sia dei registri sia dello stack. In questo modo ogni thread svolgerà le proprie operazioni e potrà eventualmente trasmetterne i risultati a flussi diversi della computazione, proprio in virtù della condivisione dell'accesso allo stesso spazio di indirizzamento. Ovviamente andranno sincronizzati opportunamente, o altrimenti si potrebbero avere dati inconsistenti (perché ad esempio modificati da altri).

Pur avendo in comune lo stesso codice, due thread diversi hanno comunque contesti specifici su cui operare (quindi registri e stack separati), in modo da poter eseguire operazioni diverse prese da parti diverse del codice. Se infatti condividessero anche stack e registri, finirebbero inevitabilmente per eseguire le stesse operazioni nello stesso momento.

## Vantaggi multithread

La programmazione multithread permette di rendere più flessibile ed efficiente la computazione rispetto alla programmazione mono-thread infatti si ha una alta affidabilità del sistema perché esiste una probabilità alta che almeno un thread sia libero per rispondere ad un evento esterno.

L’uso di thread permetto inoltre di condividere risorse e informazioni in modo molto semplice e quindi aumenta la velocità di accesso alle informazioni e di esecuzione della computazione dei thread stessi e porta ad effettuare una economia come occupazione in memoria (infatti c’è un solo codice, dati e file).

I thread condividono lo stesso spazio di indirizzamento poiché arte di uno stesso processo. Il SO può non forzare la separazione dello spazio di memoria indirizzabile da ogni thread perché anche se questo si dessero fastidio tra di loro, questi non provocherebbero problemi al resto dei processi del sistema (infatti nessuna altra applicazione sarebbe toccata da questi errori).Un altro vantaggio dell’uso dei thread è la possibilità di utilizzo dei sistemi multi-processore per velocizzare l’esecuzione dei thread.

L’utilizzo dei thread porta a diversi benefici:

* Maggior disponibilità: è infatti possibile gestire delle richieste mentre altri thread sono occupati;
* Condivisione delle risorse: tutti i thread condividono tra loro codice, dati e risorse, e ciò permette loro di accedervi semplicemente e anche di comunicare (sincronizzando però gli accessi per evitare problemi di consistenza);
* Economia: creare un nuovo thread richiede meno tempo e memoria rispetto a un nuovo processo, poiché non è necessario copiare nulla (ad esempio in Solaris la creazione è 30 volte più veloce);
* Sfruttamento delle architetture multiprocessore: i thread possono essere eseguiti in parallelo su processori differenti, raggiungendo così una condivisione efficiente e diretta delle informazioni.

## Gestione thread

Il supporto ai thread può essere realizzato nello spazio utente, come una libreria, oppure essere implementato direttamente dal SO nello spazio kernel.

* Se è implementato nello spazio utente, è il mio processo che gestisce tutti i suoi thread interni, e il SO non ne sa niente. Il SO vede solo un processo; quello che poi il processo fa al suo interno non lo sa.
* Se invece è il kernel ad implementarli, allora il SO è a conoscenza dei miei thread.

Il SO mette quindi a disposizione un insieme di funzioni che costituiscono la libreria di gestione dei thread:

* spazio utente: sono effettuate chiamate a funzioni locali
* spazio kernel: sono effettuate chiamate di sistema

# L2 - Modelli multi-thread

Per realizzare i sistemi multi-thread è possibile avere:

* un supporto fornito dal SO per i soli processi: in questo caso non esiste nessun supporto diretto all’interno del SO. Per realizzare la gestione diventa indispensabile simulare a livello utente, all’interno di un processo, l’evoluzione parallela dei thread.
* un supporto diretto dei thread all’interno del kernel del SO: in questi è proprio il kernel a gestire la schedulazione dei thread nel sistema. Il SO nel caso di supporto per i thread nel kernel, esegue l’ambiente multi thread direttamente a livello del kernel e ciò viene effettuato con una esecuzione diretta con una esecuzione diretta di un thread utente su un thread di livello kernel oppure può essere un gruppo di thread a livello utente simulata su un thread a livello kernel.

All'interno del SO, posso vedere i thread di livello utente in molti modi:

* molti a uno, più thread del processo sono visti come 1 thread dal processore;
* uno a uno, 1 thread del processo equivale ad 1 thread del processore;
* molti a molti, i thread del processo sono raggruppati in vario modo in diversi thread (di numero inferiore) del processore;
* 2 livelli.

## Modello molti a uno

Uno dei modelli multithread è il modello molti-a-uno in cui si possono avere molti thread a livello utente che sono mappati su un unico thread gestito dal kernel ed è il modo più semplice per il sistema per tenere in conto l’evoluzione dei thread.

Questo porta ad una serializzazione della computazione dei thread: quando computa uno nessun altro deve poter computare e quindi c’è una limitazione del parallelismo dell’evoluzione della computazione dei thread.

Esso comporta che in un ambiente multiprocessore, pur essendo concettualmente separata l’evoluzione del flusso di commutazione dei thread, risulta comunque serializzata perché il thread a livello kernel è uno.

Di contro l’efficienza è diminuita perchè generare un thread di livello kernel genera un overhead e quindi in molti SO viene limitato il numero massimo di thread di livello kernel che possono essere generati.

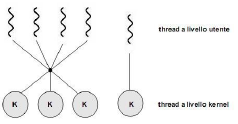
## Modello uno a uno

E’ il più utilizzato, e mappa ogni thread livello utente su un thread a livello kernel. Permette il massimo sfruttamento dei sistemi multiprocessore (più thread vengono eseguiti in parallelo su diversi processori) e la massima concorrenza, permettendo ad un thread di essere eseguito nonostante un'eventuale chiamata bloccante da parte di un altro (se un thread si blocca, si blocca solo quello).

Lo svantaggio di tale modello è che la creazione di un thread utente richiede la creazione del corrispondente kernel thread, con un conseguente overhead di gestione che grava le prestazioni del sistema all’aumentare dei thread.

## Modello molti a molti

Un metodo alternativo è il modello a due livelli in cui i thread a livello di utente possono raggrupparsi in sottoinsiemi separati, ciascuno dei quali può ò essere mappato in un thread a livello di kernel.

Questo modello garantisce una sufficiente ripartizione di capacità di elaborazione parallela specializzando i gruppi di thread a livello utente e dando una diversa opportunità di esecuzione a seconda delle esigenze di ogni singolo thread all’interno di un processo.

## Modello due livelli

Una variante comune è quella del modello a due livelli, che mappa molti thread di livello utente verso un numero più piccolo o equivalente di kernel thread, ma permette anche di associarne alcuni in modalità uno-a-uno.

## Cooperazione tra thread

La cooperazione dei thread di un processo si può organizzare attraverso modelli di:

* thread simmetrici: si hanno un insieme di thread uguali all’interno del processo. Le richieste che pervengono al processo possono essere trattate da uno qualunque dei thread perchè sono tutti equivalenti
* thread asimmetrici (o gerarchici): le richieste pervengono al processo e vengono acquisite e trattate da un thread coordinatore del gruppo, l’unico abilitato a ricevere le richieste e inviarle ai thread “lavoratori” del processo
* thread in pipeline: le richieste che pervengono al processo vengono elaborate una alla volta da un primo thread il quale svolgerà delle attività di elaborazione della richiesta, che produrrà una ulteriore richiesta che verrà passata ad un ulteriore thread e così via, formando una connessione a cascata tra i thread.

# L3 - Gestione dei thread

## Creazione ed esecuzione dei thread

Quando un processo desidera creare un thread esegue una funzione di fork() che porta ad una duplicazione di tutti i thread del processo o del solo processo che chiama la fork() e ciò dipenderà dallo specifico sistema operativo.

Quando un processo vuole rivestirsi del codice eseguibile di un programma può invocare la funzione di exec() e in questo caso il programma andrà a rimpiazzare l’intero processo.

## Esecuzione

La funzione di esecuzione exec() riveste il thread di un nuovo codice, rimpiazzando quello di partenza. In questo caso oltre la sintassi rimane invariata anche la semantica rispetto alla corrispondente chiamata di sistema per i processi.

A questo punto diventa più semplice decidere quale tipo di fork() andare ad eseguire:

* se dovrò chiamare una exec() subito dopo la fork(), allora quest'ultima potrà essere benissimo quella che duplica il solo thread chiamante, dato che la prima operazione che eseguirà sarà cancellare sé stesso e caricare qualcos'altro
* se non viene chiamata alcuna exec() dopo la fork(), allora sarebbe più opportuno duplicare tutti i thread del processo padre.

Quindi, perché due thread appartenenti a uno stesso processo possano eseguire operazioni diverse, posso adottare due strategie: faccio delle call() a porzioni diverse del codice condiviso, oppure utilizzo le exec().

## Cancellazione dei thread

Se si vuole eliminare un thread è necessario individuarlo e terminarlo prima che esso abbia terminato le sue attività e per fare ciò esistono due metodi:

* cancellazione asincrona: comporta la terminazione immediata del thread
* cancellazione differita, che può avvenire in due modi differenti: periodicamente il thread verifica, in specifici punti di cancellazione, può terminare la sua esecuzione in seguito a una richiesta (e in quel caso termina), oppure terminerà al termine delle istruzioni da eseguire (quindi al termine dell’attività), seconda una modalità detta ordinaria.

## Sincronizzazione e comunicazione

Le operazione di sincronizzazione e comunicazione dei thread all’interno del processo vengono realizzate utilizzando la memoria condivisa o utilizzando dei meccanismi sofisticati. La comunicazione viene data a tutti i thread o a un sottoinsieme di thread del processo.

Se si hanno dei processi divisi in thread che vogliono comunicare, il thread del processo mittente che invoca l’operazione di comunicazione o di sincronizzazione può effettuare questo inviando la comunicazione o la sincronizzazione con tutti i thread del processo destinatario , con un sottoinsieme dei thread o con un thread specifico del destinatario.

## Processo leggero

Il processo leggero, altrimento noto come LWP = Light Weight Process, è una via di mezzo tra il processo ed i thread. Del processo ha tutto ciò che lo renderebbe autonomo e passibile di context-switching, al pari degli altri processi. Però gran parte della sua memoria viene condivisa con quella di altri processi.

È un modo per realizzare i thread nello spazio kernel, in quanto è sì schedulato dal kernel come un processo, ma dispone della memoria assegnata ad un altro processo.

Il processo leggero si frappone ai thread gestiti dal kernel fornendo, all’insieme dei processi utente, un processore virtuale che mapperà i thread a livello utente in modo trasparente a livello del kernel, disaccoppiando la gestione del parallelismo dei thread a livello utente dal mappaggio e dall’esecuzione a livello del kernel, facendo si che il processo a livello utente non debba curarsi dei problemi di gestione ed eventuale parallelismo.

Sostanzialmente vengono visti come un unico processore che verrà replicato sui thread del kernel garantendo una attivazione efficiente a seconda delle condizioni operative dell’intero sistema.

UD3 - Schedulazione

# L1 - Schedulazione

Nei sistemi operativi multiprogrammati più processi possono essere in esecuzione e tutti concorrono nell'uso delle risorse.

Poiché solo un processo alla volta può accedere alla cpu, si è reso necessario innalzare il livello di astrazione per analizzare le tipologie di processi e mediante l'adozione di opportune politiche di schedulazione dei processi, sfruttare al meglio l'attività del processore.

I sistemi operativi possono schedulare secondo tre modalità, detti livelli di schedulazione: a breve termine, a lungo termine e a medio termine (i nomi stessi sottolineano la loro dipendenza dalla frequenza con cui sono eseguiti).

Da notare che lo scheduling a lungo termine non riguarda solo i processi in memoria centrale, ma anche quelli ancora da caricare dalla memoria centrale. Si parla quindi di uno spazio di memoria astratto, tutto quello che il gestore della memoria mi può dare in pasto.

I processi pronti sono salvati in memoria centrale e lo schedulatore a breve termine sceglierà il processo che deve andare in esecuzione. Il dispatcher effettuerà il cambio di contesto e assegnerà la cpu.

## Short term scheduler (CPU scheduler)

Si intende un tempo al di sotto del secondo (esempio, 100 ms). Ordina i processi già presenti in memoria centrale che sono pronti per l’esecuzione. Il suo obiettivo è quello di ordinarli in modo tale che dopo il dispatcher possa prendere il processo presente in memoria per eseguirlo. Il processo posto in prima posizione dell’ordinamento è quello che il dispatcher metterà in esecuzione quando avverrà il cambiamento di contesto successivo.

Deve essere eseguita in modo frequente per garantire una turnazione rapida dei processi (almeno una volta ogni 100 millisecondi). Deve essere veloce per minimizzare il sovraccarico di gestione, bisogna infatti usare il tempo di lavoro del processore per effettuare attività computazionali utili (quindi non per programmare la schedulazione) e per permettere ciò è meglio usare algoritmi semplici per la schedulazione.

## Long term scheduler

Si intende un tempo nell’ordine dei minuti. Il sistema può contenere molti più processi di quelli che possono stare in memoria centrale e qui viene in aiuto il long-term scheduler, che seleziona tra tutti i processi attivi quelli che devono essere caricati in memoria centrale ed essere posti nello stato di pronto all’esecuzione.

Allora nel sistema vengono attivati i processi ma non vengono messi in memoria centrale ma nella memoria di massa e lo scheduler sceglierà tra questi quale mettere in “ready to run” in memoria centrale. Lo short scheduler provvederà ad ordinare solo questi processi tra tutti quelli attivati sul sistema, per l’esecuzione.

Quindi lo short-term scheduler opererà su un sottoinsieme dei processi attivati nel sistemi, cioè quelli posti dal long-term scheduler in memoria centrale. Il long-term scheduler costruisce quindi il gruppo di processi mescolando processi legati al processore e i processi legati all’I/O per garantire un elevato sfruttamento del processore.

Il long-term scheduler può essere ridotto al minimo o completamente assente in alcuni sistemi. Siccome questo scheduler viene eseguito molto meno frequentemente, ci si può permettere di usare algoritmi più lenti e complessi, anche perché così in genere si ottengono migliori risultati. Poi, siccome devo caricare anche processi dai dischi, non posso certo permettermi di farlo ogni 100 ms, perché il tempo di caricamento in genere è di qualche ordine di grandezza in più.

## Mid term scheduler

Ha come obiettivo quello di mediare le caratteristiche dei due scheduler precedenti. Il suo obiettivo è quello di limitare la concorrenza tra i processi, ottimizzare la distribuzione dei processi legati alla CPU e quelli legati all’I/O e quindi ottimizzare lo spazio di memoria centrale concessa ai vari processi gestiti dallo scheduler a breve termine per poterne migliorare l’evoluzione e ridurne i tempi di esecuzione.

La soluzione è quello di modificare dinamicamente il gruppo di processi caricati in memoria centrale e post nello stato di pronto all’esecuzione dallo schedulatore a lungo termine al fine di adattare il gruppo di processi caricato in memoria centrale all’effettivo carico.

Allora lo schedulatore a medio termine ordina i processi che erano stati selezionati dallo schedulatore a lungo termine per il caricamento in memoria centrale e l’ammissione allo stato di pronto all’esecuzione, ponendo realmente in memoria centrale solo alcuni processi pronti all’esecuzione tra questo in modo da garantire un bilanciamento ottimale effettivo del carico di lavoro tra processi legati alla CPU e processi legati all’I/O, garantendo inoltre un sovraccarico minimo della memoria centrale.

Gli altri processi vengono lasciati in una area temporanea della memoria di massa in modo tale che possano essere presi successivamente dallo scheduler per una esecuzione futura. Il caricamento dei processi dalla memoria centrale alla memoria di massa temporanea (in cui vengono conservati i processi parzialmente eseguiti) prende il nome di swapping out mentre la reimmissione dei processi dalla memoria di massa alla memoria centrale prende il nome di swapping in.

## Attivazione schedule

La schedulazione può essere attivata in diversi modi:

* senza rilascio anticipato (non-preemptive scheduling, senza rilascio forzato): il processo in esecuzione viene cambiato dopo aver richiesto una operazione di I/O, dopo aver creato un processo e ne attende la terminazione, quando rilascia volontariamente il processore o quando termina. La schedulazione viene attivata sempre in modo sincrono con l’evoluzione della computazione del processo.
* con rilascio anticipato (pre-emptive): la computazione viene eseguita in modo asincrono con l’evoluzione della computazione del processo perché il contesto del processo può cambiare quando scade il quanto di tempo concesso per l’esecuzione. E’ tipico dei sistemi time-sharing.

# L2 - Criteri di valutazione

Per valutare la bontà di una politica di schedulazione si possono adottare diversi criteri che mettono in risalto aspetti diversi di uso del processo e di evoluzione della computazione dei processi.

* Utilizzo del processore, rappresentato dalla percentuale di tempo per cui il processore esegue computazione utile per i processi applicativi. Maggiore è l’utilizzo del processo e minore sarà il costo del sistema per unità di tempo effettivamente lavorato.
* Throughput, frequenza di completamento. E’ utile vedere con che capacità il sistema è in grado di completare processi. Quindi la frequenza di completamento, o throughput, è il numero di processi completati nell’unità di tempo.
* Tempo di completamento. Quando si hanno dei processi che devono essere completati entro un tempo massimo, il tempo di completamento (o turnaround time) diventa importante come criterio di valutazione del sistema.
* Tempo di attesa. A volte è importante capire quanto tempo un processo rimane in attesa (tempo di attesa) per valutare per vedere quanto usa il processore
* Tempo di risposta. Misura entro quando il processo riesce ad iniziare l’elaborazione richiesta dall’utente

Ottimizzare la schedulazione vuol dire quindi cercare di massimizzare o minimizzare alcuni o tutti gli aspetti analizzati sopra. Es. massimizzare l’utilizzo del processore, minimizzare il throughput, minimizzare il tempo di completamento etc…

Un obiettivo della tecniche di schedulazione è quello di garantire la varianza degli aspetti descritti per avere una predittività buona del comportamento del sistema.

## Approcci di valutazione criteri

I dati da utilizzare per la definizione e la scelta dell'algoritmo costituiscono un modello e tale modellazione può essere:

* Di tipo deterministico. Si basa su dati in ingresso predeterminati ma essi devono essere precisi e le caratteristiche dei processi devono essere stabili, altrimenti il modello non è reale e quindi le stime sono molto approssimative.
* A rete di code. Si basa sui calcoli della media esponenziale dei picchi di CPU e I/O dell'esecuzione dei processi. Questi valori permettono di stimare i tempi di utilizzo e di attesa dei processi. Non rispecchiano la realtà e sono molto approssimati.
* Tramite simulazione. Vengono raccolti e analizzati i dati prodotti da un modello del sistema del computer che simula l'ambiente reale. La sua realizzazione è molto costosa ed è tanto più efficace quanto più lo si utilizza.
* Tramite implementazione. L'algoritmo è eseguito nel suo ambiente di destinazione, in un contesto di utilizzo reale. Richiede costose modifiche al sistema operativo e può avere un impatto negativo sull'utenza.

## 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 sostituendo la distribuzione dei carichi di lavoro come ingressi della formula ottenuta. L’approccio è semplice, veloce e preciso ma si devono avere i valori di carico di lavoro esatti. I risultati possono però essere generalizzati.

## Modellazione statistica

Nella modellazione di tipo statistico si possono effettuare della valutazioni attraverso un modello a rete di code in cui il sistema è descritto come un insieme di servizi, ogni servizio ha una coda di attesa, le richieste di uso del servizio vengono inserite nella coda e quando il servizio è libero viene presa una richiesta , viene servita e poi il sistema la rilascia affinché il processo che la rappresenta possa continuare le attività. Nella sua vita un processo richiede una sequenza di servizi come l’uso di procedure, di periferiche etc..

### Modello a reti di code

Allora il sistema di elaborazione può essere visto come una rete di code, in cui ogni coda rappresenta un servizio del sistema (composto dal servitore e dalla coda di attesa di tale servizio). Le transizioni tra le code rappresentano i flussi delle richieste del servizio.

Analizzare le reti di code vuol dire attuare delle operazioni tipiche del metodo e in particolare vuol dire specificare la topologia del grafo della rete, specificare le caratteristiche di ogni servizio (come frequenza di arrivo delle richieste e il tempo di servizio), analizzare la rete basata sulla Teoria delle Code per effettuare l’analisi delle caratteristiche del grafo. Tale teoria porta a valutare l’utilizzo delle code, lunghezza media della coda e tempo medio di attesa (alcune semplificazioni sono necessarie!)

### Simulazione

Per modellare sempre in modo statistico il comportamento del sistema è simulare la realtà, realizzando un software del modello del sistema (hardware e software), incluso lo schedulatore e i processi applicativi in modo tale da simularne l’esecuzione. Ciò vuol dire identificare gli insiemi di dati significativi che caratterizzano le attività dei processi e poterle applicare all’interno della nostra simulazione. In questo ambiente sintetico si possono misurare le caratteristiche della schedulazione assumendo un comportamento statistico del nostro ambiente.

## Implementazione

Implementazione vera e propria del sistema hardware e software con ingressi reali. Si misurano veramente ed effettivamente le caratteristiche desiderate della schedulazione direttamente nel sistema reale. Ciò è molto oneroso e richiede la cooperazione degli utenti che sono coinvolti nella misurazione.

La schedulazione viene allora scelta in base alle caratteristiche reali che possono essere raccolte in modo automatico da parte di componenti del sistema operativo. Ciò è importante perché le informazioni raccolte possono essere utili per effettuare delle analisi successivi e quindi degli affinamenti dinamici dello schedulatore al variare delle caratteristiche del carico di lavoro del sistema di elaborazione.

Infine, teniamo in conto che magari il mio sistema riceve carichi diversi a seconda del momento della giornata, e quindi potrebbe aver bisogno di scheduling diversi, e a questo punto occorre valutare se usare il trend medio oppure adattarsi ogni volta alla nuova situazione.

# L3 - Politiche di schedulazione

## First Come, First Served

First-come, first-served (il primo arrivato è il primo servito) è il più semplice modello di schedulazione. Si ha una coda in cui i processi vengono inseriti e dalla quale vengono estratti quando il processore si libera. Questo algoritmo è non pre-emptive, infatti non porta alla sospensione dei processi in esecuzione per l’attivazione dell’algoritmo di schedulazione.

Il problema è che i processi legati alla CPU possono monopolizzare il processore, mentre l’altro problema è il tempo di attesa che può diventare alto a seconda del momento in cui i vari processi entrano nella coda.

## Shortest Job First

L’obiettivo quello di eseguire prima i processi più brevi per abbassare il tempo di attesa. Nella coda dei processi pronti, i processi saranno ordinati in ordine crescente di unità di tempo di elaborazione necessaria. Questo algoritmo può essere realizzato sia in modo pre-emptive sia in modo non preemptive.

Nel primo caso il processo che diventa pronto interrompe il processo in esecuzione richiedendo la schedulazione, mentre nel secondo caso il processo che diventa pronto non interrompe il processo in esecuzione ma lo lascia completare e solo successivamente viene eseguita la schedulazione. Globalmente, usando il modello implementato in maniera pre-emptive, il tempo di attesa medio può scendere rispetto allo stesso modello implementato in maniera non preemptive.

Questo algoritmo garantisce il tempo minimo di attesa se si conosce a priori il tempo di processore richiesto dai processi. Ciò è difficile da sapere a priori ma è possibile utilizzare delle tecniche di predizione per effettuare delle stime statistiche.

## Modello delle priorità

Cerca di mettere in evidenza l’importanza relativa tra i vari processi. Nella coda dei processi pronti contiene dei processi ordinata che hanno una rilevanza relativa crescente o decrescente in base alla logica utilizzata. La logica diretta è caratterizzata da processi in cui l’indice di priorità più alto indica una priorità alta del processo, mentre la logica indiretta è caratterizzata da processi in cui l’indice più basso indica la priorità più alta.

Questo algoritmo può essere implementato in modo preemptive o non preemptive. Il problema che si verifica con questo tipo di gestione è che alcuni processi a bassa priorità potrebbero subire un blocco indefinito (chiamato starvation). La soluzione a tale problema è rappresentata da un progressivo invecchiamento della priorità (attraverso il meccanismo di aging) con un periodico ripristino al valore iniziale oppure ringiovanimento fino al valore iniziale.

## Modello Round Robin

Basato sulla rotazione dei processi sul processore. I vari processi “ready to run” presenti sul sistema otterranno a turno l’utilizzo del processore ed è molto simile l modello First Come, First Served con l’aggiunta della pre-emption.

L’algoritmo di RR da una distribuzione uniforme del tempo di elaborazione tra i processi pronti. La velocità di esecuzione dei processi dipende dal numero di processi pronti e l’apparenza di velocità che questi processi hanno dipenderà dal numero dei processi pronti che sono presenti nel sistema: infatti più sono i processi pronti minore sarà la velocit di esecuzione. Il tempo di turn-around dipenderà dal quanto di tempo.

Il comportamento del RR dipende dalla durata del quanto di tempo: se il quanto di tempo è molto lungo il RR tende a diventare FCFS, se molto breve tende a diventare una condivisione del processore con ogni processo che vede di fatto un N-esimo della capacità computazionale ad esso associato. Questo porta a frequenti cambi di contesto con conseguente carico di gestione. L’ideale sarebbe avere circa l’80% di richieste di elaborazione completate in un quanto di tempo (è un ideale di tipo empirico).

## Coda a più livelli (C+L)

I vari processi sono raggruppati per tipologie omogenee e ogni tipologia p associata in modo permanente a un livello della coda di schedulazione, rappresentato da una coda di attesa specifica. Ogni coda di attesa ha un suo algoritmo di schedulazione e l’insieme delle code di attesa viene schedulato da un algoritmo dedicato (solitamente vengono usate delle schedulazioni a priorità fisse).

## Coda a più livelli con retroazione (C+LR)

I vari processi possono migrare da un livello all’altro grazie a meccanismi di promozione e/o degradazione. Si può avere una coda a priorità più elevata in cui il quanto di tempo è minore per garantire una maggiore rotazione dei processi e una coda a priorità più bassa per garantire una rotazione più lente, un minor uso di tempo di gestione per processi meno rilevanti (che possono evolvere più lentamente). Quindi i processi possono migrare da una coda all’altra in base alle necessità.

Si hanno quindi code di attese separate in funzione dell’uso dinamico del processore da parte dei processi. Anche in questo caso si ha un algoritmo di schedulazione specifico per ciascuna coda di attesa e si hanno delle politiche di allocazione, di promozione e degradazione dei processi (usati per inserire i processi nelle code).

## Sistemi multiprocessore

Gli algoritmi di schedulazione ricoprono grande importanza nei sistemi multiprocessore. In particolare la schedulazione deve tener conto delle caratteristiche dell’architettura del sistema di elaborazione, come:

* processori: possono essere omogenei o diversi
* memoria: solo condivisa o memoria locale dei processori
* periferiche: accessibili da un singolo processore o accessibili da tutti i processori

In questo caso si possono pensare delle politiche di schedulazioni specifiche a seconda di queste politiche hardware. Ad esempio:

* Su sistemi con processori omogenei, memoria solo condivisa e periferiche accessibili da tutti i processori si può usare una coda unica dei processi oppure una coda per ogni processore, lasciando questa coda nella memoria centrale condivisa
* Su sistemi con processori omogenei, memoria anche locale e periferiche accessibili da tutti i processori si può usare una coda unica o una coda per processore, ponendo questa coda nella memoria condivisa o nella memoria locale per velocizzare le operazioni.
* Su sistemi con processori omogenei, memoria solo condivisa e periferiche accessibili da alcuni processori si potrebbe avere una coda per processori omogenei in memoria condivisa e una coda per ogni processore che gestisce una specifica periferica.
* Su sistemi con processori omogenei, memoria anche locale e periferiche accessibili da alcuni processori si può avere una coda per ogni processore omogeneo salvata in memoria condivisa e una coda per ogni processore che gestisce una specifica periferica oppure una coda per ognuno dei processori omogenei in memoria locale e una coda per ogni processore che gestisce una specifica periferica
* Su sistemi con processori eterogenei si può avere una coda per ogni processore oppure una coda per i gruppi di processori omogenei

Si possono pensare, nel caso di sistemi multiprocessori, due diversi tipi di multi processamento:

* multi processamento asimmetrico: in cui è presente un processore master che esegue il SO (e schedula tutti i processi per tutti i processori), mentre i processori slave eseguono solo i processi applicativi
* multi processamento simmetrico: in cui ogni processore esegue il sistema operativo e provvede a schedulare i vari processi applicativi

# L4 - Schedulazione per sistemi in tempo reale

## Hard Real Time

I sistemi in tempo reale stretto (Hard real-time System) sono quelli in cui è obbligatorio che un processo termini la computazione entro un tempo massimo garantito. Il problema principale è che un processo T non riesca a completare la computazione nel tempo stabilito.

I sistemi di schedulazione nei sistemi a tempo reale stretto hanno degli approcci differenti che evidenziano l’importanza di avere una risposta a segnali esterni in tempo reale. In particolare gli approcci sono:

* schedulazione con un tempo massimo di completamento dei processi garantito. Prevede che lo schedulatore accetti un processo se riesce a garantirne l’esecuzione nel tempo massimo garantito altrimenti rifiuta il processo. L’accettazione del processo è basata su una stima del tempo di completamento del processo e sulla prenotazione delle risorse necessarie al processo. Il problema di questo algoritmo risiede nella predicibilità del tempo di completamento.
* schedulazione dei processi periodici: i processi periodici sono caratterizzati da un tempo di di elaborazione, da una scadenza (tempo massimo) e da un periodo. Tipicamente i tempi sono ordinati nel modo seguente: 0 <= tempo di elaborazione <= scadenza <= periodo.
* schedulazione a frequenza monotona: è un algoritmo per processi periodici con priorità pre-emption. La priorità può essere assegnata usando tecniche statiche o proporzionale alla frequenza 1/periodo del processo. Questa tecnica di schedulazione prevede la pre-emption, quindi se un processo a più bassa priorità è in esecuzione e arriva un processo con priorità più alta, allora i l primo processo viene interrotto per eseguire il secondo processo (a più alta priorità).
* schedulazione a scadenza più urgente: conosciuta come EDF (Earliest-Deadline First) prevede che sia assegnino le priorità ai processi in modo inversamente proporzionale alla loro scadenza e la priorità viene ricalcolata in maniera dinamica, in funzione dei processi che diventano pronti.

## Soft Real Time

I Sistemi a tempo reale lasco, “Soft Real Time System”, sono quei sistemi in cui i processi non sono tutti critici e priorità. Nei processi critici in cui si ha ovviamente l’esigenza di un rapida gestione. Quindi la priorità è statica per i processi critici ed eventualmente dinamica per i processi non critici.

Il problema di questi processi è quello di ottenere una bassa latenza nel dispatching, quindi si ha spesso bisogno di poter introdurre una “interrompibilità” delle chiamate di sistema lunghe (attraverso i pre-emption point) oppure che tutto il kernel sia interrompibile., garantendo il rapido caricamento di tutti i processi critici.

# L5 - Schedulazione dei thread

La schedulazione può essere fatta a due livelli: il livello di processo (se il sistema operativo non dispone di metodi propri per la gestione della schedulazione dei thread) e il livello di sistema (se il sistema operativo riesce a gestire la schedulazione dei thread direttamente).

La schedulazione a livello di processo prevede una visione a livello di “contesa delle risorse” all’interno del processo e la schedulazione è parte della libreria gestita dal sistema operativo adibita alla gestione dei thread. Quindi si avrà una schedulazione dei thread all’interno dei singoli processi, differente dallo scheduler dei processi del sistema operativo.

Le schedulazioni tipiche sono: a priorità fisse o modificabili al programmatore (con pre-emption), First Come - First Served, Round Robin.

La schedulazione a livello di sistema si hanno dei thread a livello del kernel o a livello user ma mappati su thread a livello kernel (eventualmente con LWP ) si ha direttamente nel Sistema Operativo la possibilità di vedere i processi divisi al loro interno in thread, ma il SO gestisce direttamente la schedulazione di ciascuno dei thread, avendo uno schedulatore capace di gestire i singoli processi ma anche i relativi thread.

UD4 - Comunicazione tra processi

# L1 - Processi cooperanti

I processi non sono sempre in lotta tra di loro, a volte un programma è stato scritto in modo tale da prevedere che processi diversi collaborino per lo stesso scopo. Nasce quindi l'esigenza di coordinare adeguatamente i vari processi (o i vari thread, al solito).

Se i processi hanno accesso a risorse ben separate, non c'è nessun problema perché non ci sono conflitti (tranne ovviamente quelli per l'accesso al processore). In caso contrario, occorre sincronizzarli.

Ad esempio, se ho processi che devono solo leggere un file, non c'è problema, possono leggere anche a pezzi. Ma se uno legge e l'altro scrive, allora nasce il problema della consistenza dei dati: può accadere infatti che il processo stia leggendo qualcosa, venga interrotto, il processo che scrive modifica, e poi il processo che legge torna a leggere i dati modificati interamente o parzialmente dal processo scrivente. Alla fine si troverà in mano dei dati inconsistenti e spesso inutilizzabili. Deve essere il SO a garantire che queste anomalie non si verifichino, deve prevenire il loro accadere in tutti i casi, per quanto rari possano essere.

Se i processi accedono a risorse condivise, vanno coordinati **implicitamente**. Altrimenti devo pensare ad un sistema **esplicito** per coordinarli, nel caso cooperino. I processi indipendenti non hanno niente in comune, quindi la loro coordinazione riguarda solamente le risorse comuni a tutti, eg il disco fisso, la tastiera, lo schermo. È quando i processi sono cooperanti che nascono i problemi, perché in qualche modo devono condividere informazioni, e influenzarsi l'un l'altro.

Affinché ciò sia possibile, il SO deve fornire politiche e meccanismi di comunicazione (IPC) e sincronizzazione. Le entità coinvolte nella comunicazione sono il processo mittente (da ora P), il ricevente (da ora Q) e il canale di comunicazione.

Ogni metodo di IPC ha delle caratteristiche proprie. La scelta di un metodo da utilizzare in un’applicazione va fatta in base a:

* Quantità di informazioni da trasmettere
* Velocità di esecuzione
* Scalabilità, ossia se il metodo scelto permette anche a un numero variabile di processi di comunicare
* Semplicità d’uso nelle applicazioni
* Omogeneità delle comunicazioni: è preferibile evitare di utilizzare più metodi di IPC nella stessa applicazione, così da evitare errori comuni
* Integrazione nel linguaggio di programmazione, che garantisce la portabilità
* Affidabilità
* Sicurezza
* Protezione

## Vantaggi

Ci sono dei vantaggi nello scrivere applicazioni fatte da processi cooperanti.

* **modularità**: mi occupo di un aspetto alla volta, e poi metto insieme i moduli che riguardano i diversi aspetti. Certo, il "mettere insieme" va studiato, però in questo modo posso dividermi il lavoro, e non pensare a cose complicate come lo scheduling, perché ci pensa il SO o la libreria di thread che uso.
* **parallelizzazione**: più processi possono essere fatti girare su più processori, nel caso ci siano, con vantaggi per tutti. Se ho un singolo processo monolitico ciò non è possibile.
* **scalabilità**: se aggiungo dei processi al mio programma, ne amplio le capacità in modo semplice.
* **specializzazione**: un processo fa una cosa, e posso chiamare uno specialista che mi scriva quel processo: non devo fare tutto io.
* **qualità del progetto e della realizzazione**: sviluppando ogni aspetto in modo separato posso raggiungere qualità elevata.

# L2 - Comunicazione tra processi

La comunicazione tra processi è l’insieme dei meccanismi e delle politiche che permettono ai processi di scambiarsi informazioni per operare in modo cooperativo.

La necessità di realizzare la comunicazione tra processi quando si vuole effettuare il trasferimento le informazioni da un processo mittente ad un ricevente o quando si ha la necessità di condividere le informazioni.

La comunicazione vede coinvolte quindi diverse entità: un processo produttore dell’informazione, l’informazione che viene trasferita, un processo ricevente (utilizzatore dell’informazione) e il canale di comunicazione a livello delle procedure di sistema . Il canale di comunicazione monodirezionale o bidirezionale.

Le comunicazioni sono caratterizzate da vari aspetti: dalla quantità di informazioni da trasmettere, dalla velocità di esecuzione, dalla scalabilità (quando si vuole aumentare il numero di processi che si vogliono gestire in comunicazione), la semplicità di uso nelle applicazioni, omogeneità delle comunicazioni, integrazione nel linguaggio di programmazione, affidabilità, sicurezza e protezione.

## Implementazione

Per realizzare la comunicazione tra processi esistono diverse tecniche:

* memoria condivisa: i processi si scambiano le informazioni attraverso la lettura/scrittura di un’area di memoria condivisa.
* messaggi: i processi si scambiano informazioni usando delle funzioni di sistema attraverso la lettura/scrittura delle informazioni che vengono poste in aree di memoria del SO chiamate Buffer.
* mailbox: il processo mittente e il ricevente si scambiano le informazioni, attraverso funzioni di SO, che vengono poste in una area di memoria chiamata Mailbox. Il processo ricevente acquisisce il messaggio prelevandolo dalla struttura dati.
* file e pipe: l processo mittente e il processo ricevente si scambiano le informazioni ponendole in un file sul disco. Il processo ricevente prenderà le info prelevando dal file. Una variante di questa tecnica è che i processi si scambiano le informazioni attraverso una pipe (struttura dati che risiede in memoria centrale) che utilizza la tecnica FIFO per inserire e prelevare i messaggi dalla coda.
* socket: è una tecnica usate nei sistemi distribuiti, infatti i processi comunicanti possono risiedere in due macchine diverse. In questo caso il processo mittente pone l’informazione da scambiare ad un estremo della pipe e il processo ricevente lo estrae dall’altro estremo opposto della pipe. I messaggi “viaggiano” da una macchina ad un’altra utilizzando la rete.

# L3 - Comunicazione con memoria condivisa

## Condividere dati in memoria centrale

La condivisione di variabili globali tra due processi consiste nell’avere un processo P con un suo spazio di indirizzamento in cui viene messo il codice, i dati globali che non si vogliono condividere , le variabili globali che si vogliono condividere, lo heap e lo stack e avremo un processo Q con le stesse aree aree di memoria e il tutto è supportato dal SO. Ovviamente l’area di memoria condivisa del processo P e vista dal processo Q come facente parte del suo spazio di indirizzamento.

Il processo P che vuole inviare messaggi al processo Q scrive le informazioni da inviare all’interno dell’area di memoria condivisa tra i due processi. I problemi della condivisione di variabili globali sono molteplici, tra i quali vi è l’identificazione del processi comunicanti (i processi devono essere a conoscenza di chi è il mittente e chi è il ricevente), la consistenza delle operazioni (lettura e scrittura sono incompatibili tra loro) e la sincronizzazione per l’accesso in mutua esclusione all’area dati comune ai due processi.

## Area comune copiata dal SO

Il processo P vedrà uno spazio di indirizzamento contenente il codice, una parte con le variabili globali da non condividere, una parte con le variabili globali da condividere, lo heap e lo stack. Il processo Q analogamente vedrà uno spazio di indirizzamento uguale a quello descritto precedentemente.

Il SO utilizzerà una ulteriore porzione di memoria che conterrà una copia del contenuto della memoria condivisa tra P e Q. Quando il processo P vuole condividere una variabile con il processo Q va a scrivere l’informazione nel suo spazio locale di memoria adibito alla condivisione dei dati.

Fatto questo sarà il sistema operativo a prendersi i valori scritti e li copierà nel suo spazio riservato alla condivisione. Solo quando il processo Q andrà a richiedere le informazioni condivisi il SO andrà a copiare le informazioni aggiornate nell’area di memoria contenente le variabili da condividere del processo P.

Questa tecnica è poco utilizzata perché è lenta a causa della copia di informazioni anche non modificate (quindi se il processo P modifica solo una variabile, al processo Q verranno passate tutte le variabili, comprese quelle che il processo P non ha aggiornato, portando ad un rallentamento delle operazioni computazionali).

## Area comune fisicamente condivisa

Si hanno due processi P e Q con le solite aree di memoria tranne l’area di memoria condivisa. Infatti i due processi avranno adesso uno spazio per il codice, i dati globali che non si vogliono condividere , lo heap e lo stack.

Per mettere in comunicazione i due processi si ha però a disposizione un’area fisica di memoria (al di sopra del SO) condivisa dai due processi. Quindi i due processi vedono la porzione di memoria condivisa come propria e questa area di memoria garantirà la comunicazione tra i due processi.

## Condivisione di buffer

I processi P e Q dispongono in memoria di uno spazio per il codice, uno per i dati, uno per lo heap e uno per lo stack. Lo spazio di indirizzamento dei due processi sono separati. Il processo P invierà una serie di informazioni al processo Q ponendole in una porzione di memoria condivisa dai due processi, ma ciò che viene trasferito è solo la serie di informazioni che viene appositamente costruita dal processo mittente al processo ricevente.

I problemi sono l’identificazione dei processi comunicanti (perché la comunicazione è diretta), deve esserci consistenza degli accessi alla zona di buffer condivisa e i due processi devono sincronizzati per garantire l’accesso in mutua esclusione. Per realizzare ciò è possibile utilizzare un buffer con copiatura gestita dal SO o utilizzare un buffer in memoria fisicamente condivisa.

# L4 - Comunicazione con scambio di messaggi

Si ha un processo mittente P che vuole inviare delle informazioni rappresentandole in messaggi per un processo Q che le deve ricevere. Il processo P gestisce l’invio al processo Q con il supporto di opportune primitive di SO . I messaggi conterranno le informazioni atte a trasferire i dati dal destinatario. In particolare i messaggi contengono: l’identificativo del processo mittente, l’identificativo del processo destinatario, informazioni da trasmettere, eventuali altre informazioni di gestione dello scambio messaggi. I messaggi possono avere una dimensione fissa o variabile.

I messaggi vengono inviati da un processo attraverso il SO al destinatario. Il SO memorizza le informazioni da trasferire in spazi di memoria interni che prendono il nome di Buffer. I Buffer possono essere assegnati a ciascuna coppia di processi o possono essere assegnati di volta in volta ad ogni coppia di processo che ne fa richiesta. La quantità dei buffer può essere illimitata (quindi possono essere inviati messaggi illimitati), limitati (c’è un limite al numero di informazioni che si possono trasmettere) o nulla (il processo mittente dovrà sempre aspettare che vi sia un ricevente pronto a ricevere i dati).

Per gestire le comunicazioni il SO mette a disposizione una serie di primitive. Per l’invio mette a disposizione la primitiva send(Q, messaggio) che depositerà il messaggio in un buffer libero. Questa primitiva può essere bloccante, nel senso che bloccherà il mittente, fintanto che non vi siano dei buffer liberi per completare la comunicazione con il destinatario.

Il messaggio può essere ricevuto dalla primitiva receive(P, messaggio) che permetterà la ricezione del messaggio in un buffer. La primitiva può essere bloccante per il destinatario fintanto che non vi sia un messaggio ricevibile (quindi fino a quando non vi sia un buffer libero).

In alcuni casi si ha la necessità di avere delle primitive per l’invio condizionale: quindi se vi sono buffer liberi nel processo destinatario allora il processo P invia i dati, se invece non vi sono buffer liberi in cui scrivere viene ritornata una condizione di errore non bloccando il mittente. Questa primitiva è cond\_send(Q, messaggio): error\_status.

Esiste una primitiva simile per la ricezione condizionale che prevede una ricezione non bloccante di messaggi: se vi sono messaggi ricevibili li riceve altrimenti segnala una condizione di errore senza bloccare il destinatario. Questa primitiva è cond\_receive(P, messaggio): error\_status.

Quando i processi accedono alle operazione di comunicazione mediante scambio di messaggi, le comunicazioni possono essere sincronizzate in vari modi: si possono avere comunicazioni asincrone che diventano bloccanti se l’operazione non può essere completata (se il processo mittente non trova un buffer libero nel ricevente è costretto a fermarsi e quindi deve sincronizzarsi con il ricevente) e comunicazioni sincrone se la comunicazione avviene solo con la presenza contemporanea dei dei processi (quindi il processo mittente aspetta sempre che il processo destinatario esegua la funzione di ricezione).

I processi comunicanti devono identificarsi: nella comunicazione con messaggi se la comunicazione è simmetrica mittente e destinatario sono sempre univocamente identificati (perché la comunicazione è diretta) mentre se la comunicazione è asimmetrica mittente o destinatario possono non essere identificati univocamente. Se il destinatario non è univocamente specificato, allora questo può essere o un processo appartenente ad un gruppo specificato o appartenente ad un gruppo qualunque. Per quanto riguarda invece l’invio di messaggi, un processo può inviare messaggi ad un processo di un gruppo specificato o ad un processo qualunque.

Quindi un sistema di comunicazione tra processi basato sullo scambio di messaggi è caratterizzato dall’identificazione dei processi comunicanti, dall’assenza di una memoria condivisa dai processi comunicanti e dalla sincronizzazione per l’accesso ai messaggi gestita implicitamente dal sistema operativo.

L’implementazione di tale tecnica è descritta di seguito. Si hanno due processi P e Q, il cui spazio di indirizzamento è separato, e il SO che mette a disposizione un insieme di spazi di memoria, i buffer, strutturati con una parte mittente, un destinatario e una messaggio. Il SO potrà mettere a disposizione dei processi un insieme di buffer per le comunicazione tra processi al fine di realizzare le operazioni bloccanti o non bloccanti.

# L5 - Comunicazione con mailbox

Il modello di comunicazione con mailbox prevede che un processo mittente P vuole trasmettere un messaggio utilizzando una struttura del SO chiamata mailbox o porta. Questa struttura contiene dei buffer in cui i messaggi possono essere accumulati. Quando un proceso Q vule ricevere un messaggio, accede alla mailbox attraverso delle funzioni di SO prelevando il messaggio. Il contenuto dei messaggi presenti nella mailbox è formato dall’identificativo del processo mittente, dalla mailbox destinataria, dalle informazioni da trasmettere ed eventuali altre informazioni a supporto della gestione dei messaggi nella mailbox. La dimensione dei messaggi può inoltre essere fissa o variabile. La mailbox po' avere diversa capacita: se illimitata potrà contenere un numero illimitato di messaggi (quindi il processo mittente non verrà mai bloccato), se limitata potrà contenere soltanto un determinato numero di messaggi (quindi il processo mittente può essere bloccato se non vi è più spazio a sufficienza per depositare il messaggio), se invece la capacità è nulla nessun messaggio potrà essere depositato e quindi il mittente potrà inviare il messaggio solo se vi è in attesa un processo ricevente.

La funzione di creazione della mailbox è la create(M) mentre la funzione per la sua cancellazione è la delete(M). La procedura di invio dei messaggi è la send(M, messaggio) la cui funzione è quella di depositare un messaggio nella mailbox. Questa funzionè è non bloccante se la capacità della Mailbox è illimitata, bloccante se la mailbox è limitata e piena, bloccante se la capacità e nulla e non c’è un processo in ricezione. La funzione di ricezione è la receive(M, messaggio) la cui funzione è ricevere un messaggio dalla mailbox e può essere bloccante se non c’è almeno un messaggio ricevere.

Possono essere messe a disposizione dal SO le funzioni condizionali come la cond\_send(M, messagio): err\_status la quale deposita un messaggio nella mailbox se la comunicazione può' essere completata, mentre se ‘invio bloccasse il mittente verrebbe ritornato un messaggio di errore non bloccando il mittente. Esiste anche la cond\_receive(M, messaggio): err\_status riceve un messaggio dalla mailbox se c’è almeno un messaggio e se non ci sono messaggi ricevibili ritorna una condizione di errore non bloccando il destinatario.

I processi che stanno comunicando vengono sincronizzati in base alla capacità della mailbox: se la capacità è illimitata la comunicazione è asincrona, se la mailbox è limitata la comunicazione è bufferizzata, mentre se la mailbox è nulla allora la comunicazione è sincrona.

Nel caso di una comunicazione basata sulla mailbox non si ha più a necessità di conoscere gli identificativi dei processi comunicanti (perché la comunicazione è indiretta), non si ha memoria condivisa tra i processi e si ha sincronizzazione tra i processi solo per l’accesso ai messaggi, ma è gestita implicitamente dal SO.

I messaggi in attesa nella coda della mailbox possono essere ordinati attraverso delle opportune politiche di ordinamento: FIFO, per priorità, per scadenza. La proprietà della mailbox può essere del SO e quindi non essere legata ad alcun processo o essere specifica di un processo e quindi solo il processo proprietario riceve da questa mailbox e gli altri processi possono solo inviare. Se termina il processo proprietario scompare la mailbox.

Le mailbox permettono di realizzare comunicazioni con più processi mittenti o più processi riceventi. In ogni caso la comunicazione coinvolge comunque sempre solo due processi (un mittente e un ricevente).

## Comunicazione molti a uno, uno a molti, molti a molti

Nelle comunicazioni molti a uno la mailbox viene vista da più processi mittenti che vogliono inviare un messaggio alla mailbox. Ogni processo invierà il messaggio e la mailbox, se ha spazio a sufficienza potrà procedere con la computazione. La mailbox vede solo un processo ricevente quindi quando un messaggio arriva nella mailbox questo viene estratto e viene computato dal processo ricevente.

Nella comunicazione uno a molti si ha una maibox a cui fanno riferimento più processi riceventi. Questi si porranno in attesa che arrivino dei messaggi. Quando P invia un messaggio alla mailbox, il messaggio verrà dato al primo processo presente nella coda di attesa (secondo la politica scelta) mentre gli altri processi rimarranno in coda, in attesa di ulteriori messaggi.

Nella comunicazione molti a molti si ha una mailbox e che a questa fanno riferimento una serie di processi in attesa e a questa fanno riferimenti molti processi che desiderano inviare messaggi. In questo caso un processo P depone un messaggio, un processo in attesa riceverà il messaggio e verrà estratto dalla coda e verrà fatto continuare nella sua computazione e anche il processo P continuerà nella sua computazione. Quindi appena un altro processo invierà un messaggio alla mailbox vi sarà un altro processo in attesa che riceverà tale messaggio che permetterà ai due processi (mittente e ricevente) di continuare la loro computazione.

# L6 - Comunicazione con files

## File condivisi e pipe

Le comunicazione attraverso file condiviso vengo realizzate mediante un processo P che vuole inviare un messaggio ad un altro processo utilizzando delle strutture di memoria di massa, in particolare un file sul disco. In questo modo il processo mittente P scrive sul file sul disco e il processo ricevente Q legge dal file.

Invece le pipe sono dei file presenti in un file locati in memoria centrale. Inoltre il meccanismo basato sulla pipe prevede che le informazioni vengano inviate in un modo ordinato, in particolare utilizzando la politica FIFO. Il processo P che vuole deporre un messaggio, salva questo in maniera ordinata in una struttura pipe (tubo) e il processo Q estrae i messaggi dall’altro estremo della pipe.

I messaggi scambiati in questo modo hanno il seguente contenuto: l’identificativo del processo mittente, le informazioni da trasmettere ed eventuali altre informazioni a supporto della gestione dei messaggi. La dimensione dei messaggi può essere fissa o variabile.

## Funzioni e politiche

Le funzioni messe a disposizione dal SO sono quelle di creazione, cancellazione, lettura e scrittura dei file e delle pipe e sono funzioni gestite attraverso il modulo di gestione del filesystem.

Nella comunicazione tra processi utilizzando file e pipe si ha la sincronizzazione dei processi in lettura e scrittura effettuata secondo politiche e meccanismi del file system, si ha l’ordinamento dei messaggi (nel file dipende dal processo scrivente, mentre nella pipe è sempre FIFO) e si ha un ordine dei processi in attesa (nel file dipende dalla gestione del file system).

# L7 - Comunicazione con socket

Nel modello della comunicazione basata su socket si hanno due macchine con due SO che supportano le operazioni dei processi applicativi. Si ha un processo mittente P che desidera mandare un messaggio e abbiamo sull’altra macchina un processo ricevente Q. In questo caso si può generalizzare il modello di comunicazione tramite la pipe, in cui in questo caso la pipe sarà “spezzata” (o condivisa) tra le sue macchine. Quindi la macchina1 e la macchina2 hanno una pipe condivisa grazie all’utilizzo della rete che permette ai processi P e Q di scambiarsi i messaggi pur non risiedendo nella stessa macchina.

La comunicazione quindi ha una architettura client-server in cui il client invia delle richieste a una porta specifica e il server, che è in ascolto su una porta specifica, riceve ed elabora le richieste dei client e invia la risposta al client (che diventa quindi ricevente). Il canale di comunicazione prende il nome di porta ed è individuato da un indirizzo della macchina ospite (indirizzo IP) e da un numero progressivo che univocamente specifica la porta (http:80, ssh:22 etc…).

I messaggi possono contenere le informazioni da trasmettere e non esiste alcun altra informazione di identificazione perché la comunicazione avviene tra mittente e ricevente. La dimensione del messaggio può essere fissa o variabile.

## Funzioni e politiche

Le funzioni di creazione, cancellazione, lettura e scrittura dei socket sono specifiche del SO che si considera.

Il canale di comunicazione che permette il passaggio delle informazioni tra client e server è rappresentato dalla rete e può essere monodirezionale o bidirezionale a seconda della direzione del flusso dei dati.

I dati così inviati sono ordinati attraverso la politica FIFO e lo stesso vale per l’ordinamento dei processi in attesa di ricevere un messaggio (anch’essi utilizzano la politica FIFO). La connessione può essere gestita con gestione della connessione (ogni sequenza di messaggi è controllata e garantita anche in caso di problemi di trasmissione), senza gestione della connessione e con multicast (un messaggio viene consegnato a più processi).

UD5 - Sincronizzazione dei processi

# L1 - Processi concorrenti

La compresenza in memoria di processi cooperanti implica la loro esecuzione (sincrona o asincrona) con conseguente condivisione di dati. La consistenza del dato è fondamentale per la sua corretta esecuzione.

Quando più processi accedono contemporaneamente alla stessa risorsa fisica o logica, si ha la concorrenza. Se l’esito delle operazioni dipende dall’ordine in cui sono eseguite, si ha una corsa critica, e ciò può causare inconsistenza nella risorsa.

Una porzione di codice che causa corse critiche se eseguita in modo concorrente è detta sezione critica. Ciò va evitato con meccanismi e politiche di sincronizzazione (necessaria e importantissima nei sistemi operativi).

Lo scopo dell'individuazione della sezione critica è quello di creare un protocollo che possa essere utilizzato per bypassare i problemi di inconsistenza; la soluzione deve dunque soddisfare tre condizioni:

* mutua esclusione: se un processo sta eseguendo la sua sezione critica nessun altro può farlo
* progresso: la competizione per entrare nella sezione critica deve avvenire tra i processi che non sono nella loro sezione critica, e la competizione non deve durare indefinitamente
* attesa limitata: per prevenire la starvation di un processo viene prefissato un numero di volte massimo secondo cui un determinato processo entra nella sezione critica dopo che un secondo ha fatto richiesta di entrare nella propria; in altre parole, un processo non deve essere costretto ad attendere indefinitamente per entrare nella sua sezione critica

La sincronizzazione deve garantire gli ultimi tre punti elencati sopra attivando sistemi misti hardware/software. Ci sono diversi metodi per garantire la corretta esecuzione della sezione critica.

# L2 - Variabili

Un tipo di approccio a livello di istruzioni per la sincronizzazione di due processi concorrenti, sono le variabili di turno. Esse sono variabili condivise tra i processi che interagiscono per accedere in modo concorrente a una risorsa, stabilendone il turno d'uso. In altre parole dicono quale processo ha il diritto di usarla in un certo istante.

Un altro approccio a livello di istruzioni sono le variabili di lock. E' una variabile condivisa che definisce lo stato di uso di una risorsa, cioè quando è in uso da parte di un processo (che è evidentemente nella sua sezione critica). Cambia così il punto di vista rispetto alla variabile di turno: non sono più i processi ad alternarsi ma è la risorsa stessa a dire se è disponibile o no. Può assumere due valori: 0 se la risorsa è libera, 1 se è in uso.

## Stato di uso di una risorsa

Sostanzialmente questa soluzione consiste nell'allocare all'interno della memoria condivisa (in maniera tale cioè da essere visibile ad ogni processo) una certa cella (o parola o bit) che assume la funzione di "lucchetto" (lock) per l'accesso ad una certa risorsa condivisa. Ogni processo può accedere al suo valore e controllare se essa è "chiusa" o "aperta", cioè se tale flag è settato a 0 o a 1.

Quando il valore della variabile di Lock è 0, allora il processo può settarla ad 1 (cioè "chiuderla"), accedere alla risorsa condivisa ed entrare così nella sua sezione critica; quando ha finito, rimette il suo valore a 0 (per mezzo di una normale MOVE).

Gli altri processi che avessero voluto accedere alla risorsa condivisa avrebbero testato il valore del flag non nullo e avrebbero aspettato l'evento costituito dal ritorno 0 della variabile di lock.

In poche parole, quando un processo deve entrare nella sua sezione critica, controlla il valore di una certa variabile per vedere se l'entrata è possibile. Nel caso non lo sia, il processo in stato di Running che vuole accedere a quella risorsa continua a ciclare testando il valore del lock finché questo non cambia.

## Problema interrupt

Essendo questa operazione costituita da più righe di codice assembly, c’è il rischio che durante questo procedimento arrivi un interrupt ad interrompere questa procedura. Per risolvere questo problema il controllo e il test sul valore di questa variabile viene solitamente effettuata con una istruzione particolare che si trova presente in molti sistemi (soprattutto quelli multiprocessori) detta Test And Set Lock (TSL).

Tale istruzione garantisce l'indivisibilità di esecuzione in quanto tale istruzione, essendo una system call, non può essere interrotta. Questa soluzione è forse la più semplice ed intuitiva ma presta il fianco a molte critiche: essa richiede un considerevole spreco di utilizzo della CPU (spesso il sistema esegue semplicemente un loop di test); inoltre è un metodo comunque poco efficiente in caso che uno dei processi sia molto più lento degli altri.

Una soluzione che fa a meno di sprecare l'utilizzo della CPU è quella proposta da Dijkstra nel 1965 e che, con qualche modifica, si è imposta nei vari sistemi informatici quasi universalmente (per esempio, è una delle soluzioni adottata sia da Unix che da Windows NT): i semafori.

# L3 - Semafori

L’obiettivo dell’uso dei semafori è quello di elevare i l livello di astrazione della gestione della sincronizzazione portandolo dal livello delle istruzioni al livello di funzioni del sistema operativo. Obiettivo di questo è garantire la corretta gestione della sincronizzazione e dell’accesso alle variabili di supporto alla mutua esclusione, quindi si vuole così evitare un uso errato delle operazioni di abilitazione e disabilitazione delle interruzioni o degli assegnamenti alle variabili di turno o lock.

## Semaforo binario

Un semaforo binario è una variabile binario che rappresenta lo stato di uso di una risorsa condivisa. Solitamente si una 1 per la risorsa libera e 0 se la risorsa è in uso (al contrario rispetto alle variabili di lock). Il semaforo S è manipolato da due unzioni:

- acquire(S) → acquisisce l’uso della risorsa

- release(S) → rilascia la risorsa

Queste operazioni sono atomiche poiché sono funzioni di sistema e ciò garantisce che tali funzioni vengano valutate in modo mutuamente esclusivo.

Il semaforo è essenzialmente una variabile binaria che inizialmente conterrà 1. Quando arriva n processo P che vuole acquisire la risorsa, tale processo P, eseguirà una funzione di acquire(S) e porrà a 0 il valore del semaforo, prendendosi la risorsa e procedendo nella sua computazione. Se un processo Q tenterà di effettuare un acquire(S) della risorsa, tale operazione non sarà permessa, quindi il processo verrà messo in coda di attesa. La coda dei processi di attesa è ordinata in modo specifico secondo una politica definita sullo specifico semaforo (es. FIFO, per priorità etc…). Quando il processo P decide di lasciare la risorsa, invierà attraverso la funzione di release la segnalazione che la risorsa è tornata disponibile, ponendo a 1 la variabile presente nel semaforo. Poiché esistono dei processi in attesa la risorsa viene ceduta al primo processo presente nella coda di attesa, il quale procederà nella propria computazione.

## Implementazione semaforo binario

Il semaforo binario può essere implementato in due modi:

* attesa attiva in caso di risorsa non disponibile: in questo caso il processo guarda continuamente il valore della variabile binaria. Quindi basta avere come dati una variabile binaria S e un funzione acquire(S) che cicla continuamente sulla variabile S fintanto che la risorsa non diventa disponibile. Questo metodo comporta uno spreco di risorse di calcolo.
* sospensione e rischedulazione in caso di risorsa non disponibile: in questo caso si usa una variabile binaria S e una coda dei processi in attesa di acquisire la risorsa. La procedura di acquireS) sospende il processo in esecuzione in caso la risorsa sia non disponibile e lo inserisce nella coda di attesa del semaforo, mentre la release(S) rilascia la risorsa e riattiva il primo processo della coda di attesa cedendogli la risorsa; lo schedulatore dei processi in attesa della della risorsa definisce l’ordine di ottenimento della risorsa in base alla politica adottata per il semaforo.

## Semaforo generalizzato

Il semaforo generalizzato S è una variabile intera che rappresenta lo stato di uso di un insieme di risorse omogenee condivise. Il semaforo verrà posto a n, con n uguale al numero delle risorse libere, e diventerà 0 quando tutte le risorse sono in uso. Il semaforo S è manipolato dalle funzioni:

- acquire(S) → acquisisce l’uso di una risorsa

- release(S) → rilascia la risorsa in uso

In questo caso il semaforo S è inizializzato al numero delle risorse. Un processo P farà una operazione di acquire(S) (che porterà ad un decremento della variabile del numero delle risorse) e lo stesso varrà per tutti gli altri processi che effettueranno altre acquire(S). Se le risorse a disposizione sono pari a 0 il processo rimarrà in attesa finché una risorsa non verrà liberata. Appena liberata la risorsa, il processo in attesa uscirà dalla coda di attesa e continuerà nella sua computazione.

# L4 - Monitor

## Problemi legati all’uso dei semafori

Quando si utilizzano i semafori per gestire la sincronizzazione si possono avere degli errori di programmazione che non possono essere gestite dal SO, come le violazioni della mutua esclusione le attese infinite a causa di un mancato rilascio delle risorse. La responsabilità e la correttezza dell’uso dei semafori è lasciata quindi al programmatore, di conseguenza il SO non ha il potere di controllo e di gestione su di essi.

Per risolvere questo problema si può innalzare il livello di astrazione per la gestione della sincronizzazione forzando l’uso corretto. La soluzione è quindi l’uso del Monitor, un costrutto linguistico di sincronizzazione formulato a livello di linguaggio di programmazione, che il compilatore trasforma nelle corrette chiamate di sistema per garantire la correttezza delle chiamate.

## Definizione di monitor

Un processo alla volta può trovarsi all’interno di un monitor e può essere ivi attivo. Quindi un monitor è una collezione di attività delle varie sezioni critiche dei processi e l’insieme di procedure da eseguire in mutua esclusione. Quando si entra in un monitor una sola di queste procedura può andare in esecuzione.

Un processo chiamerà la procedura che realizza la sua sezione critica e il fatto che queste siano incluse in un monitor garantisce che all’inizio venga attivato il meccanismo dell’individuazione dell’utilizzo mutuamente esclusivo e alla fine venga attivato il meccanismo di rilascio della sezione critica e della relativa risorsa utilizzata.

## Realizzazione e uso del monitor

Per realizzare un monitor vengono usate una serie di informazioni che vengono condivise per rappresentare la risorsa comune, le operazioni da effettuare sulla risorsa comune e il codice di inizializzazione dell’ambiente operativo. Quando un processo non ottiene immediatamente l’uso della risorsa contenuta nel monitor verrà accodata in una coda di attesa per l’accesso alla risorsa stessa.

Il monito è quindi costituito da un insieme di risorse che deve essere gestito in mutua esclusione e da un insieme di operazioni che devono essere effettuate su tali risorse. Quando si vuole gestire la mutua esclusione, automaticamente il compilatore associa al monitor un variabile condizione che rappresenta lo stato di uso del monitor (se è libero o se è utilizzato, quindi coincide con un semaforo).

Ponendo a 1 (libero) la variabile condizione, quando arriva un processo p che vuole utilizzare la risorsa e quindi chiamare una operazione contenuta nel monitor che vengono gestite n modo mutualmente esclusivo, se la condizione dice che il monito e libero il processo ottiene l’uso della primitiva e procede, entra nella sezione critica e la variabile condizione viene impostata a 0 (occupato) in modo tale che se un altro processo Q richiedesse l’accesso a delle operazioni gestite nel monitor, il monitor non lo permetterebbe e metterebbe tale richiesta nella coda di attesa dei processi che non riescono a completare le operazioni e rimarrebbe bloccato fino a quando il processo P non terminerebbe le sue operazioni e la variabile condizione venga posta a 1 (libera).

# L5 - Problemi della starvation e del deadlock

Un processo entra in uno stato di starvation quando attende per un tempo indefinito una risorsa poiché altri processi ottengono sempre prima tale risorsa. La causa della starvation 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.

Si ha invece un deadlock quando in un gruppo di due o più processi, ciascuno di essi aspetta una risorsa che è detenuta in modo mutuamente esclusivo da un altro processo e la causa di tale problematica è l’attesa circolare delle risorse senza alcun rilascio di queste.

Per risolvere la situazione di stallo è necessario adottare delle opportune tecniche per impedire che si verifichi una opportunità del genere, prevenire uno stallo, lasciare che il deadlock avvenga per poi risolverlo oppure ignorarlo (quando la possibilità di avere una tale situazione è davvero molto bassa).

# L6 - Transazioni atomiche

Una transazione atomica è un’insieme di istruzioni che eseguono un’unica funzione logica in un'applicazione (es: la sequenza di lettura e manipolazione dei dati). Si dice atomicità delle transazioni il fatto che l’effetto della transazione sulle informazioni memorizzate deve essere permanente solo e soltanto se tutte le operazioni coinvolte nella transazione sono state completate correttamente senza interferenze da parte di altri processi. Ovvero la sequenza di operazione di una transazione deve essere atomica, come un’unica operazione indivisibile.

Una transazione termina correttamente ed esegue il commit quando gli effetti permanenti sono dovuti a tutte le operazioni completate correttamente, se invece la transazione non riesce a completarsi correttamente allora termina in modo errato con una segnalazione di abort, quindi si deve eseguire un rollback alla condizione precedente all’applicazione della transazione.

## Tipologie di archivi

Si possono avere diverse tipologie di archivi in un sistema di elaborazione:

* archivi volatili: le informazioni non sopravvivono allo spegnimento del sistema. É il caso delle memoria cache e della memoria centrale
* archivi non volatili: le informazioni sopravvivono allo spegnimento del sistema: dischi magnetici e ottici, nastri magnetici
* archivi stabili: le informazioni non vengono mai perse perché viene effettuata la replicazione in molti archivi non volatili

## Transazioni atomiche in archivi volatili

Le transazioni atomiche individuali che non hanno concorrenza con altre transazioni su risorse condivise, possono essere gestite mediante le tecniche di logging (write-ahead logging) e di checkpointing.

Il log delle transazioni (registro delle transazioni) memorizza in un archivio stabile le transazioni e il loro stato di esecuzione: nome della transazione, nome dell’oggetto dei dati, il vecchio valore dei dati e il nuovo valore.

Il meccanismo di write-ahead di logging consiste nel memorizzare all’inizio della transazione una informazione all’interno del registro che segnali l’inizio della transazione stesso. Al termine della transazione viene scritto nel log un record che contiene il nome della transazione e lo stato di completamento e la parola chiave commit. Se la transazione non va a buon fine non viene scritto il commit.

Per effettuare il recupero mediante il file di log si utilizzano due operazioni:

* undo (Ti): riporta i dati modificati dalla transazione Ti ai vecchi valori
* redo (Ti): assegna ai dati modificati dalla transazione Ti il nuovo valore

Queste due operazioni sono idempotenti ovvero la ripetuta applicazioni ai dati di un sistema porta sempre ad ottenere lo stesso risultato.

## Logging

Usare il log per effettuare il ripristino vuol dire analizzare la sequenza di informazioni scritte nel log di decidere quale delle due operazioni prima definite chiamare. Se la transazione viene abortita a causa di un errore all’interno della stessa, il log non contiene il commit finale e il sistema deve applicare la operazione di undo per ripristinare la situazione iniziale, prima dell’inizio della transazione. Se invece il sistema stesso di elaborazione fallisce (es. guasto di un componente hardware), nel log si avrà per ogni transazione una informazione che attesta lo stato di esecuzione della transazione. Per ogni transazione del log:

* se il log contiene <T i starts> ma non <T i commits>, esegue undo(T i )
* se il log contiene sia <T i starts> sia <T i commits>, esegue redo(T i )

L’uso del logging può portare ad un tempo molto lungo di ripristino perché si deve ripartire dall’inizio e riscandire tutte le transazioni effettuate. La soluzione è quella di inserire dei check point. La tecnica di Check Pointing prevede che periodicamente si scrivano in un archivio stabile dei record del log memorizzati sul archivio volatile. Dopo di che si marca sul checkpoint di sistema sull’archivio stabile dei log di sistema il fatto che è stato raggiunto un check point. Ripristinare una situazione usando un checkpoint consiste allora nell’eseguire le modifiche alle informazioni volatili che si hanno nel sistema a partire dal checkpoint più recente del log. Quindi in caso di fallimento del sistema di elaborazione, per ogni transazione del log a partire dal check point più recente si va a vedere:

* se il log contiene <T i starts> ma non <T i commits>, e nel caso si esegue un undo(T i )
* se il log contiene sia <T i starts> sia <T i commits>, e nel caso si esegue il redo(T i )

## Esecuzione concorrente di transazioni atomiche

Le transazioni possono essere non solo individuali ma anche concorrenti. L’esecuzione concorrente di transazioni atomiche deve avvenire in modo seriale e in un ordine arbitrario in quanto si deve garantire l’atomicità delle azioni svolte all’interno di ciascuna transazione, quindi si deve creare all’interno di ognuna delle transazioni concorrenti una condizione di serializzabilità. Per serializzare le transazioni si può operare a due livelli:

* a livello di transazione : garantire che le transazioni intere vengano eseguite in sezioni critiche e quindi garantire la serializzazione ponendo un semaforo mutex comune tra le transazioni. Ciò comporta una lentezza nell’esecuzione delle transazioni e riduce le possibilità di parallelismo perché di fatto solo un processo alla volta può accedere alle risorse informative condivise.
* a livello di operazioni svolte nelle transazioni: garantire la serializzazione delle singole operazioni strettamente necessarie (non più delle intere transazioni) e ciò viene fatto grazie ad algoritmi di controllo della concorrenza delle operazioni attraverso una schedulazione concorrente seriale o attraverso una schedulazione concorrente serializzabile (protocollo di lock, protocolli basati su timestamp)

Una schedulazione concorrente seriale prevede che se una transazione T0 arriva prima, questa deve eseguire tutte quante le sue operazioni su tutti i record e solo al termine vengano eseguite le altre operazioni di un’altra transazione T1. Ciò garantisce la consistenza delle informazioni ma impedisce la parallelizzazione delle operazioni.

Una schedulazione concorrente serializzabile invece prevede che in due transazioni T0 e T1 che operano sugli stessi record (es A e B), vengano prima eseguite le operazioni della transazione T0 sul un record A, poi le operazioni della transazione T1 sul record A, poi ancora le operazioni della transazione T0 sul record B e infine le operazioni della transazione T1 sul record B. In tal modo viene favorita la serializzazione e il parallelismo.

Per realizzare una schedulazione concorrente serializzabile possiamo usare delle tecniche differenti:

* il Lock (blocco) è una variabile associata ad un dato che definisce l’accessibilità al dato stesso. Il lock viene posto come “libero” se l’accesso al dato è consentito, mentre viene posto come “in uso” se la transazione è sospesa in attesa di un blocco libero.

Esistono due tipi di lock:

* il lock condiviso: consente a più processi di accedere allo stesso dato nel caso in cui le operazioni che devono svolgere non siano conflittuali (es. lettura di un dato)
* il lock esclusivo: garantisce che solo un processo veda la risorsa come usabile e possa quindi modificarlo garantendo la consistenza delle operazioni per gli altri processi.

Il protocollo di lock di base prevede che una transazione applichi il lock sul dato Q che desidera leggere, se il lock è disponibile Ti accede al dato, se il lock non è disponibile si possono avere diverse situazioni:

* Se il lock richiesto è esclusivo, Ti attende finché il dato viene rilasciato
* Se il lock richiesto è condiviso:
  + Ti accede al dato se esso è correttamente bloccato con lock condiviso
  + Ti attende se il dato è correttamente bloccato con lock esclusivo

## Protocollo di lock a due fasi

Una situazione di questo genere non garantisce la serializzabilità. Per risolvere si può introdurre il protocollo di lock a due fasi. Nella prima fase (chiamata fase di crescita) una transazione può ottenere dei lock, ma non li può rilasciare, mentre nella seconda fase (fase di contrazione), una transazione può rilasciare i lock, ma non ne può ottenere di nuovi. Questo protocollo assicura la serializzabilità ma può portare a situazioni di stallo.

Per garantire la serializzazione nei protocolli di lock si può definire un ordine di serializzazione di ogni coppia di transazioni in conflitto in modo che la loro esecuzione venga determinata dal primo lock che viene richiesto e che definisce le incompatibilità. Una tecnica per ottenere questo è il timestamp. Il timestamp (marca i tempo) è un attributo che rappresenta quando la transazione Ti è entrata nel sistema. Questo è univocamente associato alle transazioni dal sistema operativo e usualmente viene generato basandosi sul clock di sistema o su un contatore.

Il protocollo basato sul timestamp per accedere alla risorsa condivisa prevede l’uso di due tipologie di timestamp:

- il Write-TimeStamp (Q)

- il Read-TimeStamp(Q)

Ogni operazione read o write in conflitto è eseguita nell’ordine della marca di tempo.

## Esempi

Vediamo due esempi di lettura e scrittura di un dato, da parte di una transazione, usando il protocollo basato sul timestamp:

read(Q)

* Se TS(T i ) < W-timestamp(Q), la lettura è negata e T i esegue roll-back
* Se TS(T i ) ≥ W-timestamp(Q),la lettura è eseguita e R-timestamp(Q)=max{R-timestamp(Q), TS(T i )}

write(Q)

* Se TS(T i ) < R-timestamp(Q), la scrittura è negata e T i esegue roll-back
* Se TS(T i ) < W-timestamp(Q), la scrittura è negata e T i esegue roll-back
* Altrimenti, la scrittura è eseguita

L’ordine di serializzazione di ogni coppia di transizioni in conflitto è determinato da timestamp associato a ciascuna transazione alla sua attivazione.

UD6 - Deadlock

# L1 - Caratterizzazione del deadlock

Quando si hanno delle risorse condivise in un sistema è necessario fare un uso corretto e consistente delle risorse. Esistono delle risorse condivise che possono essere usate in modo non esclusivo (come un file in sola lettura) o in modo mutuamente esclusivo (stampante) e quindi è necessario sincronizzare i processi per l’accesso all’uso delle risorse condivise usabili solo in modo mutuamente esclusivo, per poter garantire la consistenza delle risorse. A tal fine è necessario effettuare una serie di passi come:

* la richiesta di uso della risorsa
* l’uso della risorsa
* rilascio della risorsa

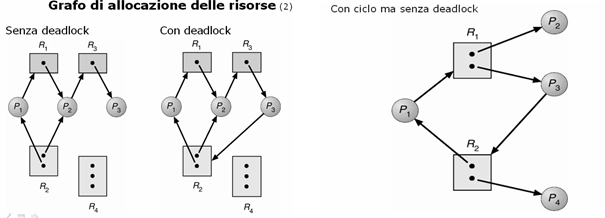
Il deadlock (o stallo) è una condizione in cui ogni processo in un gruppo è bloccato in attesa di un evento che può essere generato solo da un altro processo del gruppo, tipicamente il rilascio di risorse fisiche o logiche. Nella condizione di deadlock i processi non terminano mai la loro esecuzione e le risorse di sistema a loro assegnate rimangono bloccate, impedendone l'accesso agli altri processi.

## Condizioni

Affinchè il deadlock si possa verificare devono essere soddisfatte contemporaneamente 4 condizioni:

1. **mutua esclusione**: indica il fatto che la risorsa deve essere usata in modo mutuamente esclusivo
2. **Possesso e attesa**: il processo ottiene la risorsa ma non la rilascia entrando in uno stato di attesa di altre risorse
3. **nessun rilascio anticipato**: indica che la risorsa non è sottoponibile a preemption, quindi non può essere tolta al processo che la sta usando prima del termine del processo stesso per garantire la consistenza del dato
4. **attesa circolare**: prevede che un insieme di processi siano uno in attesa di una risorsa posseduta da un altro in modo circolare.

Per verificare la presenza di tali condizioni si può utilizzare il grafo di allocazione delle risorse. E’ un grafo con un insieme di nodi V e Archi E, definito come G(V, E). I nodi possono essere processi del sistema o risorse del sistema (le risorse possono avere più instanza identiche), mentre gli archi possono essere archi di richiesta (da un processo ad una risorsa) o achi di assegnazione di una risorsa (vanno da una risorsa a un processo).



## Gestione deadlock

Per gestire un deadlock esistono diversi metodi:

* Ignorare il deadlock.
  + Nella maggior parte dei sistemi, il deadlock è una condizione rara e quindi si può risparmiare tempo e aumentare l’uso della risorse ignorandolo e resettando il sistema se si verifica.
* Prevenire il deadlock.
  + Si agisce invalidando una delle 4 condizioni che causano il deadlock imponendo dei vincoli sulle richieste, che però comportano un potenziale sottoutilizzo delle risorse e sono scomodi per il programmatore.
* Mutua esclusione.
  + Alcune risorse sono intrinsecamente condivisibili (ad esempio file in lettura), e per queste si può invalidare; tuttavia, in generale non è possibile farlo poiché non tutte lo sono.
* Possesso e attesa.
  + Si può invalidare imponendo che i processi richiedano tutte le risorse all’avvio, oppure imponendo che un processo, prima di fare una richiesta, rilasci tutte le risorse che possiede e le richieda tutte in blocco. In entrambi i casi si ha un sottoutilizzo delle risorse e possibile starvation; inoltre, nel secondo caso, si richiede ai programmatori di rilasciare le risorse in stato consistente poiché potrebbero essere assegnate ad altri processi.
* Nessun rilascio anticipato.
  + Si può invalidarla in 2 modi: quando un processo viene messo in attesa per una richiesta, tutte le sue risorse sono rilasciate anticipatamente, e quindi attende per la richiesta più le risorse che già possedeva. In alternativa, si possono rilasciare anticipatamente solo le risorse di processi in attesa che sono richieste da un processo in esecuzione. Il problema della consistenza descritto prima rimane, tuttavia è il sistema operativo che rilascia le risorse, e non il programmatore. Questo metodo funziona bene con risorse il cui stato può essere salvato e ripristinato.

# L2 - Tecniche di prevenzione del deadlock

La prevenzione di un deadlock si può avere facendo si che almeno una delle condizioni di esistenza dei deadlock non sia soddisfatta.

La mutua esclusione deve essere soddisfatta per tutte le risorse che non sono usabili in modo condivise mentre non è necessaria per le risorse usabili in modo condiviso. La condizione può essere invalidata rimuovendola per le risorse intrinsecamente condivisibili, mentre non può essere mai invalidata per le risorse intrinsecamente non condivisibili.

La condizione di possesso e attesa può essere invalidata garantendo che ogni volta che un processo chiede risorse, non possegga già qualche altra risorsa. Le tecniche per realizzare tale approccio per la prevenzione del deadlock consistono nel:

* far richiedere a un processo tutte le risorse necessarie prima di iniziare la sua esecuzione.
* Un processo che possiede alcune risorse e vuole ottenere altre deve:
  + rilasciare tutte le risorse che possiede
  + chiedere tutte quelle che servono, incluse eventualmente anche alcune di quelle che già possedeva

I problemi derivanti da questo approccio sono lo scarso utilizzo delle risorse e una possibile starvation.

Eliminare la condizione di rilascio anticipato può essere invalidato mediante il rilascio anticipato (pre-emption) per risorse il cui stato di uso all’atto del rilascio anticipato è ripristinabile.

Le tecniche per gestire i rilascio anticipato sono spiegate di seguito.

* Se un processo detiene alcune risorse e ne chiede altre che non possono essere assegnate immediatamente:
  + tutte le risorse possedute sono rilasciate anticipatamente
  + le risorse rilasciate anticipatamente sono aggiunte alla lista delle risorse per cui il processo sta aspettando
  + il processo sarà fatto ripartire soltanto quando potrà ottenere le vecchie e le nuove risorse
* 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
    - 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
    - ripartire quando ottiene tutte le risorse necessarie

## Attesa circolare

Si può invalidare imponendo un ordinamento sulle risorse, ossia una definendo una funzione che assegni ad ogni risorsa un numero intero, e imponendo questo protocollo: quando un processo richiede risorse di ordine i, questi può ottenerle solo se sono disponibili e non possiede già risorse di ordine >=i.

Se ne possiede, le deve rilasciare e richiedere in ordine corretto (con tutti i problemi relativi). In questo modo, l’attesa circolare non può verificarsi, ma la funzione va definita in base ai consueti ordini di utilizzo delle risorse, e può non essere facile.

Esistono alcune tecniche, nelle quali si utilizza un ordinamento globale univoco imposto su tutti i tipi di risorsa R. Allora:

* Se un processo chiede k istanze della risorsa R j e detiene solo risorse R i con i<j, allora se le k istanze della risorsa R j sono disponibili vengono assegnate altrimenti il processo deve attendere (in questo caso un processo non potrà mai chiedere istanze della risorsa R j se detiene risorse R i con i>j).
* 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

# L3 - Tecniche per 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 sequenza dei processi già accettati nel sistema. Per supportare tali analisi è necessario avere a disposizione delle informazioni sul comportamento dei processi:

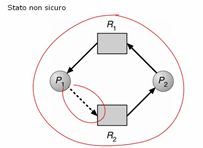
* il numero massimo di risorse per ogni processo
* le risorse assegnate
* le risorse disponibili
* le richieste e i rilasci futuri di risorse

Usando queste informazioni è possibile verificare se il sistema si può trovare in uno stato in cui è garantita l’assenza di deadlock o no. Si dice allora che uno stato è sicuro se il sistema può allocare le risorse richieste da ogni processo in un certo ordine garantendo che non si verifichi deadlock.

Una sequenza di processi <P 1 , P 2 , P 3 , ..., P n > è una sequenza 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 con j<i, allora uno stato è sicuro se esiste una sequenza sicura.

Per evitare un deadlock è quindi necessario garantire che il sistema passi da uno stato sicuro ad un altro stato sicuro quando un processo chiede una nuova risorsa. Infatti si parte da uno stato iniziale sicuro, una richiesta di risorsa viene soddisfatta se la risorsa è disponibile e se il sistema va in uno stato sicuro; se la risorsa non è disponibile, il processo deve attendere. Per supportare tale approccio si può usare l’algoritmo del grafo di allocazione delle risorse, in cui viene aggiunta una nuova tipologia di archi: gli archi di prenotazione. Questa tipologia di archi sono delle richieste di risorse non ancora diventate effettive e vanno dal processo alla risorsa.

Uno stato sarà non sicuro quando la presenza di un arco di prenotazione evidenzia un ciclo tra processi e risorse (vedi immagine sotto).

L’algoritmo di gestione basato sul grafo di allocazione delle risorse prevede la costruzione del grafo con gli archi di prenotazione e nel caso in cui si evidenziano dei cicli, lo stato non è sicuro e quindi non si può accettare la richiesta di risorse dell’ultimo processo inserito. Questo però vale solo per le istanze singole delle risorse.

Per la gestione di istanze multiple delle risorse viene utilizzato l’algoritmo del banchiere. Questo algoritmo è meno efficiente dell’algoritmo del grafo di allocazione delle risorse, il numero massimo di istanze deve essere dichiarato a priori e un processo deve restituire in un tempo finito le risorse utilizzate. Per supportare tale algoritmo devono essere introdotte alcune strutture dati: si hanno m risorse, n processi, un vettore delle risorse disponibili Available[1...m], la matrice Max[1...n,1...m] che rappresenta la massima richiesta di ogni processo per ciascuna risorsa (il primo indice indica il processo, il secondo la risorsa), la matrice Allocation[1...n,1...m] indica le risorse che sono già state assegnate ad ogni processo (quindi quante istanze di ogni risorsa sono state assegnate a un processo), la matrice Need[1...n, 1...m] indica quante risorse un processo può ancora chiedere.

L’algoritmo del banchiere utilizza una procedura di verifica della sicurezza dello stato. In tale procedura vengono dichiarati due vettori Work e Finish. Il vettore Work è un vettore di risorse disponibili nel Sistema e viene posto uguale al vettore Available all’inizio della procedura, mentre il vettore Finish è un vettore di booleani che indica se un processo è stato completamente gestito e viene inizializzato a falso. Si va a cercare un indice i tale per cui il processo non è ancora gestito (e quindi non è ancora stata assegnata l’allocazione delle risorse) e contemporaneamente la richiesta di risorse che esso fa è minore o uguale al numero delle risorse disponibili in quel momento. Il vettore ausiliario Work viene considerato in modo tale da modificare le allocazioni in modo temporaneo fintanto che non viene verificata la sicurezza dello stato che si deve raggiungere. Se non esiste un indice i che soddisfa entrambe le condizioni allora se per ogni i il vettore Finish è uguale a true (quindi tutti i processi sono stati verificati) si deduce che lo stato è sicuro, altrimenti se esiste un indice i che soddisfa le suddette condizioni si allocano fittiziamente le risorse richieste dal processo in modo tale da verificare la sua compatibilità e viene marcato che il processo è stato trattato.

(Di seguito vengono riportati i passi dell’algoritmo)

Work[1..m]

Finish[1..n]

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

Vediamo adesso come funziona l’algoritmo di richiesta delle risorse (quindi l’algoritmo del banchiere vero e proprio).

Viene introdotto il vettore di richieste effettuate dal processo i. Se il vettore di richieste è minore o uguale delle necessità del processo i allora si procede al passo successivo, altrimenti c’è una condizione di errore (il processo ha chiesto più risorse rispetto a quelle che sono dichiarate nel vettore Need[i]). Se non ci si trova in quest’ultimo caso (e quindi siamo al passo 2) si va a verificare se le richieste avanzate dal processi i-esimo sono soddisfacibili e cioè se esiste la disponibilità delle istanze di risorse in grado di soddisfare la richiesta altrimenti il processo deve attendere. Se le risorse sono disponibili si devono instanziare le richieste e quindi si tolgono dal vettore di risorse disponibili Available[..] le richieste del processo i-esimo, si allocano tali richieste nella matrice delle risorse allocate e si tolgono le richieste cosi soddisfatte dal vettore delle necessità dichiarato dal processo i-esimo (Need[i]). Se lo stato risultante è sicuro, al processo Pi vengono confermate le risorse assegnate altrimenti Pi deve aspettare per richieste Request[i] e viene ristabilito il vecchio stato di allocazione delle risorse.

(Di seguito vengono riportati i passi dell’algoritmo)

Request[i] – richiesta del processo P i

1. Se Request[i] <= Need[i], vai al passo 2
   1. Altrimenti, solleva errore: processo ha ecceduto numero massimo di richieste
2. Se Request[i] <= Available, vai al passo 3 altrimenti P i deve attendere: risorse non disponibili
3. Si ipotizzi di stanziare le risorse richieste:
   1. Available=Available-Request[i]
   2. Allocation[i]=Allocation[i]+Request[i]
   3. Need[i]=Need[i]-Request[i]

Questo algoritmo garantisce che se un processo è in grado di ottenere delle risorse senza causare una situazione di deadlock allora le ottiene e quindi può procedere nella sua esecuzione.

# L4 - Tecniche di rilevazione e ripristino del deadlock

Senza algoritmi di prevenzione il deadlock può verificarsi quindi il sistema deve essere in gradi rilevarli dopo che questi sono avvenuti e deve essere in grado di ripristinare una situazione di corretto funzionamento del sistema eliminando il deadlock.

Per istanze singole delle risorse si disegna il grafo di attesa dove gli archi identificano le dipendenze dei processi. La rilevazione può essere fatta nel seguente modo: si analizza il grafo di attesa e se sono presente dei cicli di richieste per una risorsa, allora si ha un deadlock. I processi in deadlock sono quelli coinvolti nei cicli del grafo.

Nel caso di istanze multiple delle risorse bisogna creare un algoritmo più complesso che tenga conto della molteplicità delle istanze e di conseguenza è necessario definire delle strutture dati di supporto alla realizzazione di tale approccio:

* m risorse
* n processi
* Available [1...m] risorse disponibilità
* Allocation [1...n, 1...m]: risorse attualmente assegnate
* Request [1...n, 1...m]: risorse della richiesta corrente

Si ha un vettore Work di risorse disponibili, un vettore Finish che indica se i processi sono stati esaminati o meno. Allora:

1. Work = Available

Per i=0,…, n-1

se Allocation[i]!= 0 allora Finish[i]=false altrimenti Finish[i]= true

1. Si cerca un indice i tale che:
   1. Finish[i] ==false (il processo non è stato ancora verificato se è in deadlock)
   2. Request[i] <= Work (verifica se le richieste del processo sono minori rispetto risorse disponibili)

Se non esiste tale indice i, si va al passo 4, altrimenti si procede al passo 3

1. Work = Work + Allocation[i]; Finish[i]=true (si effettua l’aggiornamento delle allocazioni e si dichiara il processo soddisfatto)
2. Finish[i]==false per qualche i, quindi si ha un processo per il quale non si è riuscito ad allocare una risorsa allora si ha il deadlock. Se finish [i] == false, allora il processo Pi è in deadlock

Si invoca il suddetto algoritmo per la rilevazione del deadlock:

* ogni volta che una richiesta di allocazione non può essere soddisfatta immediatamente: rilevazione immediata, poche risorse e processi bloccati e considerevole sovraccarico computazionale
* a intervalli di tempo prestabiliti: rilevazione più complessa, molte risorse e processi possono essere bloccati e vi è un minor sovraccarico computazionale