SISTEMI OPERATIVI M

APPUNTI DI FRANCESCO PENNELLA

ANNO 2016/2017

SOMMARIO

01. VIRTUALIZZAZIONE

Dato un sistema caratterizzato da un insieme di risorse sia HW che SW, virtualizzare il sistema significa presentare all'utilizzatore una visione delle risorse diversa da quella reale. Ciò si ottiene introducendo un livello di indirezione tra la visione logica (sistema virtuale) e la visione fisica delle risorse (sistema reale). L'obiettivo è quello di disaccoppiare il comportamento delle risorse di un sistema di elaborazione dalla sua realizzazione fisica.

Alcuni esempi di virtualizzazione sono:

- **Virtualizzazione a livello di** processo -> i processi multitasking permettono la contemporanea esecuzione di più processi, ognuno dei quali dispone di una MV dedicata. In questo caso il sistema vede la macchina come se fosse interamente dedicata a loro. Virtualizzazione realizzata dal kernel del sistema operativo.
- **Virtualizzazione della memoria** -> in presenza della memoria virtuale, ogni processo vede uno spazio di indirizzamento di dimensioni indipendenti dallo spazio fisco effettivamente a disposizione. La virtualizzazione è realizzata dal kernel del sistema operativo. Questo meccanismo dà l'illusione a chi usa la memoria di avere a disposizione una memoria id grande di quella che c'è realmente sotto.
- **Astrazione** -> un oggetto astratto (risorsa virtuale) è la rappresentazione di un oggetto (risorsa fisica). Questo disaccoppiamento è realizzato dalle operazioni (interfaccia) con le quali è possibile utilizzare l'oggetto.
- **Linguaggi di programmazione** -> la capacità di portare lo stesso programma su architetture diverse è possibile grazie alla definizione di MV in grado di interpretare ed eseguire ogni istruzione del linguaggio, indipendentemente dall'architettura del sistema.

VIRTUALIZZAZIONE DI SISTEMA ----------

Una singola piattaforma HW viene condivisa da più elaboratori virtuali ognuno gestito da un proprio sistema operativo. Il disaccoppiamento è realizzato da un componente chiamato VMM (Virtual Machine Monitor o Hypervisor) il cui compito è consentire la condivisione d parte di più MV di una singola piattaforma HW.

Il VMM è il mediatore unico nelle iterazioni tra macchine virtuali e l'hardware sottostante e deve garantire isolamento tra le MV e stabilità del sistema. Ogni MV è isolata dalle altre poiché esegue all'interno di una sandbox in modo da non avere interferenze con le altre MV che eseguono sul sistema. Inoltre per non causare instabilità del sistema, le impostazioni di sicurezza da una MV non devono andare ad interferire con altre MV virtualizzate.

Sulla stessa macchina posso andare a creare e definire più MV e ognuna appare all'utente come una macchina completa e utilizzabile con hardware virtuale e un suo sistema operativo su cui possono girare applicazioni.

EMULAZIONE ----------

Esecuzione di programmi compilati per una particolare architettura su un sistema di elaborazione dotato di un diverso insieme di istruzioni. Vengono quindi emulate interamente le singole istruzioni dell'architettura ospitata permettendo a sistemi operativi o applicazioni pensati per determinate architetture di girare anche su architetture diverse. Si possono utilizzare due differenti approcci:

- **Interpretazione** -> si basa sulla lettura di ogni singola istruzione del codice macchina che deve essere eseguito e sull'esecuzione di più istruzioni dell'host virtualizzante per ottenere lo stesso risultato. Produce però un sovraccarico molto elevato poiché possono essere necessarie molte istruzioni dell'host per interpretare una singola istruzione sorgente. Questo approccio risulta meno performante di un approccio compilato in cui prima si fa una traduzione preliminare e poi risulta molto più veloce a tempo di esecuzione.
- **Ricompilazione dinamica** -> invece di leggere ogni singola istruzione, legge interi blocchi di codice, li analizza e li traduce per la nuova architettura ottimizzandoli e infine li mette in esecuzione. Il codice quindi viene prima tradotto e ottimizzato utilizzando tutte le possibilità offerte dalla nuova

architettura e poi messo in esecuzione. Parti di codice usate frequentemente possono essere anche bufferizzate per evitare di doverle ricompilare in seguito.

ESEMPI DI EMULATORI ----------

QEMU è un software che implementa un particolare sistema di emulazione che permette di ottenere un'architettura nuova e disgiunta in un'altra che si occuperà di ospitarla permettendo di eseguire programmi compilati su architetture diverse. Utilizza la tecnica della traduzione dinamica.

Virtual PC è un software di emulazione che consentiva a computer con sistema operativo Windows o Mac OSX l'esecuzione di sistemi operativi diversi come altre versioni Windows o Linux. Permetteva l'interoperatività tra sistemi Windows e Mac quando la differenza tra i due sistemi non lo permetteva naturalmente.

MAME è un software per personal computer in grado di caricare ed eseguire codice binario originale delle ROM dei videogiochi da bar emulando l'hardware tipico di quelle architetture. Poiché gli attuali PC hanno una potenza di calcolo nettamente superiore a quello dei primi processori da giochi questo software utilizza la tecnica dell'interpretazione senza compromettere l'efficienza.

CENNI STORICI ----------

Anni 60 -> IBM crea un sistema CP/CMS diviso in due livelli appunto CP e CMS. Il primo esegue direttamente sull'HW svolgendo il ruolo di VMM, il secondo invece è un sistema operativo interattivo e monoutente replicato su ogni MV.
Anni 70 -> nascono i sistemi operativi multitasking
Anni 80 -> l'evoluzione della tecnologia porta alla nascita dei microprocessori, quindi si passa da architetture basate su mainframe verso minicomputer e PC.
Anni 80/90 -> i produttori di hardware abbandonano l'idea di supportare un concetto di virtualizzazione a livello architetturale seguendo il paradigma: "One application, one server"
Fine anni 90 -> nuovi sistemi di virtualizzazione per architetture Intel x86
Anni 2010 -> Cloud Computing

VANTAGGI DELLA VIRTUALIZZAZIONE ----------

La virtualizzazione comporta numerosi vantaggi:

- **Uso di più sistemi operativi** -> permette di utilizzare uno o più sistemi operativi sulla stessa macchina fisica permettendo a un utente di avere più ambienti eterogenei.
- **Isolamento degli ambienti di sicurezza** -> ogni MV definisce un ambiente di esecuzione separato (sandbox) da quelli delle altre garantendo isolamento degli ambienti di esecuzione.
- **Consolidamento HW** -> possibilità di concentrare più macchine su un'unica architettura HW per un utilizzo più efficiente delle risorse fisiche.
- **Gestione facilitata delle macchine** -> le MV possono essere create, configurate e amministrate in maniera molto semplice, permettendo anche di effettuare operazioni di migrazione a caldo tra macchine fisiche.

REALIZZAZIONE DEL VMM ----------

Ogni VMM deve in generale offrire alle diverse MV le risorse che sono necessarie per il loro funzionamento come la CPU, la memoria e i dispositivi di I/O.

Secondo Popek e Goldberg i requisiti per la realizzazione di un VMM sono i seguenti:

- **Ambiente di esecuzione per i programmi sostanzialmente identico a quello della macchina reale**
- **Garantire un'elevata efficienza nell'esecuzione dei programmi**, quindi il VMM deve permetter l'esecuzione diretta della istruzioni impartite dalle MV
- **Garantire la stabilità e la sicurezza dell'intero sistema**. Il VMM deve rimanere sempre nel pieno del controllo hardware, poiché le MV non possono accedervi in modo privilegiato.

LIVELLO -> si intende dove è collocato il VMM, e può essere di due tipi: **VMM di sistema**, quindi esegue direttamente sopra l'HW dell'elaboratore oppure **VMM ospitato** e quindi esegue come una qualunque applicazione sopra al sistema operativo esistente. Tra il SO e la VMM al posto di una operazione privilegiata ci deve essere una Hypercall che permette di delegare alla VMM l'esecuzione della primitiva.

MODALITA DI DIALOGO -> per l'accesso alle risorse fisiche tra la macchina virtuale e il VMM: **Virtualizzazione pura**, cioè le MV usano la stessa interfaccia dell'architettura fisica o **Paravirtualizzazione**, cioè il VMM presenta un'interfaccia diversa da quella dell'architettura HW.

In un **VMM di sistema** le funzionalità di virtualizzazione sono integrate in un sistema operativo leggero, costituendo un unico sistema posto direttamente sopra l'HW. (VMWARE, XEN KVM). Mentre in un VMM ospitato, questo viene installato sopra un sistema operativo già esistente come un'applicazione, a questo punto il VMM opera nello spazio utente e accede all'HW tramite le system call del SO su cui è installlato.

HOST -> piattaforma di base sulla quale si realizzano le MV e comprende la macchina fisica, l'eventuale sistema operativo (solo ne caso di VMM ospitato) e il VMM.

GUEST -> la macchina virtuale. Comprende applicazioni e sistema operativo della MV.

VMM DI SISTEMA ----------

L'architettura della CPU prevede in generale almeno due livelli di protezione, chiamati RING: supervisore e utente. Solo il VMM opera nello stato supervisore mentre il SO e le applicazioni della macchina virtuale operano nello stato utente.

SI possono presentare due tipologie di problemi:

- **Ring deprivileging** -> il SO della MV esegue in uno stato che non gli è proprio (problema legato all' esecuzione delle system call). Le istruzioni privilegiate richieste dal SO nell'ambiente guest non possono essere eseguite. Una possibile soluzione è l'approccio **TRAP AND EMULATE**: se il guest tenta di eseguire un'istruzione privilegiata, la CPU notifica un'eccezione al VMM (trap) e gli trasferisce il controllo, a quel punto il VMM controlla la correttezza dell'operazione richiesta e ne emula il comportamento.
- **Ring compression/aliasing** -> se i ring utilizzati sono 2, applicazioni e So della MV eseguono tutte e due allo stesso livello, il che comporta la necessità di protezione tra spazio del So e delle applicazioni. In questo modo alcune istruzioni non privilegiate eseguite in modalità user permettono di accedere in lettura ad alcuni registri la cui gestione dovrebbe essere riservata al VMM.

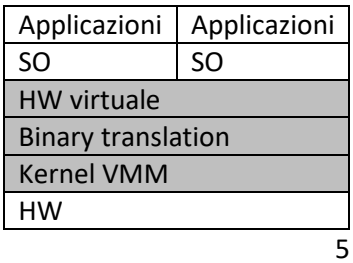
SUPPORTO HW ALLA VIRTULIZZAZIONE ----------

Un'architettura si dice **naturalmente virtualizzabile** se prevede l'invio dei notifica allo stato supervisore per ogni operazione privilegiata eseguita da un livello di protezione diverso da quello supervisore. In questo caso la realizzazione del VMM è semplificata proprio grazie al supporto nativo della virtualizzazione e sfruttando l'approccio TRAP AND EMULATE. Però non tutte le architetture sono naturalmente virtualizzabili ed in questi casi ci sono alcune istruzioni privilegiate eseguite a livello user che non provocano una trap ma vengono ignorate non consentendo l'intervento del VMM.

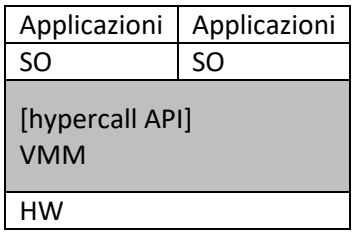
VMM IN ARCHITETTURE NON VIRTUALIZZABILI ----------

Se il processore non fornisce alcun supporto alla virtualizzazione allora è necessario ricorrere a soluzioni SW come ad esempio il **FAST BINARY TRANSLATION** e la **PARAVIRTUALIZZAZIONE**.

Con la **fast binary translation** il VMM scansiona dinamicamente il codice del SO guest (un po' come succedeva nella compilazione dinamica) prima dell'esecuzione per sostituire a run time i blocchi contenenti istruzioni privilegiate. I blocchi tradotti sono poi eseguiti e conservati in una cache per eventuali riutilizzi. Con questo meccanismo ogni MV è una esatta replica della macchina fisica ma la traduzione dinamica è molto costosa.



Con la **Paravirtualizzazione** invece il VMM (Hypervisor) offre al SO guest un'interfaccia virtuale, le hypercall API, alla quale i SO guest devono fare riferimento per avere accesso alle risorse: come ad esempio per ottenere un servizio che richiede l'esecuzione di istruzioni privilegiate, in quel caso non vengono generate interruzioni per il VMM ma viene invocata la hypercall corrispondente. I SO guest quindi devono essere modificati per avere accesso all'interfaccia del particolare VMM, la cui struttura è semplificata perché non deve più preoccuparsi di tradurre dinamicamente i tentativi delle MV. La **Paravirtualizzazione** ha prestazioni migliori rispetto alla **fast binary translation**, ma necessità di porting dei SO guest.



ARCHITETTURE VIRTUALIZZABILI ----------

L'uscita sul mercato di processori con supporto nativo alla virtualizzazione ha dato l'impulso allo sviluppo di VMM semplificati basati su virtualizzazione pure. Questo ha eliminato il **ring compression/aliasing** poiché il SO guest esegue in un ring separato (intermedio) da quello delle applicazioni e ogni istruzione privilegiata richiesta dal SO guest genere un trap gestito dal VMM (in questo modo si gestisce anche il **ring deprivileging**). In questo caso come vantaggi non c'è bisogno della binary translation ma le API presentate dall'hypervisor sono le stesse offerte dal processore.

1° generazione -> non avevano nessuna capacità di protezione e non facevano distinzione tra SO e applicazioni facendo girare entrambe con i medesimi privilegi. In questo modo le applicazioni potevano accedere direttamente ai sottosistemi di Io, allocare memoria senza alcun intervento del SO.

2° generazione -> viene introdotto il concetto di protezione, con la distinzione tra SO che possiede controllo assoluto sulla macchina fisica sottostante e le applicazioni, che possono interagire con le risorse fisiche solo facendone richiesta al SO (concetto di **ring di protezione**).

Registro CS -> i due bit meno significativi vengono riservati per rappresentare il livello corrente di privilegio CPL (Livello Protezione Corrente) utilizzando 4 diversi ring: ring 0 è il ring dotato di maggiori privilegi e quindi destinato al kernel del SO mentre il ring 3 è il livello dotato di minor privilegi e quindi destinato alle applicazioni utente; e nonostante ci siano 4 ring solitamente ne vengono utilizzati solo due: il ring 0 per il SO e il ring 3 per le applicazioni. Non è chiaramente permesso a ring diversi dallo 0 di eseguire le istruzioni privilegiate che sono destinate solo al kernel del SO in quanto considerate critiche e potenzialmente pericolose.

Segmentazione -> ogni segmento è rappresentato da un descrittore in una tabella GDT o LDT; nel descrittore sono indicati il livello di protezione PL e i permessi di accesso R,W,X

Protezione della memoria -> una violazione dei vincoli di protezione provoca un'eccezione e questo può accadere se il CPL (Livello Protezione Corrente) è maggiore del PL (Livello di Protezione) del segmento di codice contenente l'istruzione invocata.

FUNZIONAMENTO DELLA VMM NELL'ARCHITETTURA X86 CLASSICA ----------

Anche in questo caso è presente il problema del **ring deprivileging** viene dedicato il ring 0 alla VMM e conseguentemente i SO guest vengono collocati in ring a privilegi ridotti. Vengono comunemente utilizzate 2 tecniche (seguono lo schema VMM/kernel guest/applicazioni):

- **0/1/3** -> il SO viene spostato dal ring 0 dove nativamente dovrebbe trovarsi al ring 1, lasciando le applicazioni al ring 3, mentre al ring 0 viene installato il VMM. Il livello applicativo non può danneggiare il SO virtuale sul quale è in esecuzione (ring 1) come ad esempio andando a scrivere su porzioni di memoria ad esso dedicate . Le eccezioni generate sono catturate dal VMM sul ring 0 e passate al livello 1 dove e presente il SO.
- **0/3/3** -> il SO viene spostato direttamente al livello applicativo, ring 3, dove si trovano anche le applicazioni, mentre sul ring 0 viene installato il VMM. In questa modalità non è possibile generare eccezioni quindi devono essere intrapresi meccanismi molto sofisticati con un controllo continuo da parte del VMM.

Ring aliasing -> alcune istruzioni non privilegiate, eseguite in uno stato utente permettono di accedere in lettura ad alcuni registri del sistema la cui gestione dovrebbe essere riservato solo al VMM. Il SO guest non si deve poter rendere conto su quale livello di protezione sta eseguendo perché a quel puto si renderebbe conto di eseguire su un MV, al ring 1 invece che allo 0.

XEN ----------

VMM open source che opera secondo i principi della Paravirtualizzazione. Il VMM (Hypervisor) i occupa della virtualizzazione della CPU e della memoria e dei dispositivi per ogni MV. XEN dispone di un'interfaccia di controllo in grado di gestire la divisione di queste risorse tra i vari domini, però l'accesso a questa interfaccia è ristretto e può essere controllata solo utilizzando una MV privilegiata, il Domain 0.

Nell'architettura XEN il VMM iene chiamato **Hypervisor** e le MV vengono definite **Domains**, di cui una in particolare chiamata **Domains 0** controlla tutto il sistema e gode di privilegi diversi rispetto a tutte le altre macchine. Il Domain 0 deve essere sempre attivo per permettere il funzionamento di tutto il sistema, inoltre al suo interno può controllare i driver dei dispositivi.

XEN adotta una configurazione dei ring come quella **0/1/3**, inoltre le MV eseguono direttamente le istruzioni non privilegiate e l'esecuzione delle istruzioni privilegiate viene delegata al VMM tramite le hypercall.

Gestione della memoria -> i SO guest gestiscono la memoria virtuale mediante i tradizionali meccanismi di paginazione, adottando una soluzione in cui le tabelle delle pagine delle VM vengono mappate nella memoria fisica dal VMM, non possono essere accedute in scrittura dai kernel guest ma solo dal VMM e sono accessibili in modalità lettura anche dai guest. I SO guest si occupano della paginazione delegando al VMM la scrittura delle page table entries. La tabella delle pagine devono essere create e verificate dal VMM su richiesta dei guest.

Memory split -> XEN risiede nei primi 64 bit del virtual address space. Lo spazio di indirizzamento virtuale per ogni MV è strutturato in modo da contenere XEN e il kernel in segmenti separati.

Creazione di un processo -> Il SO guest richiede una nuova tabella delle pagine al VMM: alla tabella vengono aggiunte le pagine appartenenti al segmento di XEN, XEN registra la nuova tabella delle pagine e acquisisce il diritto di scrittura esclusiva, poi ad ogni successiva update da parte del guest provocherà una **protection-fault** la cui gestione comporterà la verifica e l'effettivo aggiornamento della tabella delle pagine.

Balloon process -> La paginazione è a carico del guest, occorre quindi un meccanismo efficiente che consenta al VMM di reclamare ed ottenere in caso di necessità dalle altre MV pagine di memoria meno utilizzate. Su ogni machina virtuale è in esecuzione un processo (Ballon process) che comunica con il VMM, questo processo è sempre attivo e viene chiamato in causa dal VMM in modo tale che richieda al proprio guest altre pagine. Questa richiesta provoca da parte del SO guest l'allocazione di nuove pagine al balloon process che una volta ottenute le cede al VMM.

Virtualizzazione della CPU -> il VMM definisce un'architettura virtuale simile a quella del processore nella quale però le istruzioni privilegiate sono sostituite da opportune hypercall (l'invocazione di una hypercall determina il passaggio da guest ring 1 a ring 0). Il VMM si occupa dello scheduling delle MV utilizzando il **Borrowed Virtual Time algorithm** che consente in caso di vincoli temporali stringenti di ottenere schedulazioni efficienti. Vengono quindi utilizzati due clock: **real-time** (tempo del processore) e **virtual-time** (associato alla MV e avanza solo quando la MV esegue).

Virtualizzazione I/O -> per motivi di indipendenza dall'HW delle singole MV si è deciso di condensare nel Domain 0 i driver (back end driver), mantenendo però in ogni singola MV un'interfaccia dei driver (front-end drivers) che si occupano di trasferire al vero e proprio driver la richiesta.

Nella soluzione adottata da XEN quindi avremo un **back-end driver** per ogni dispositivo e il suo driver è isolato all'interno della MV Domain0, con accesso diretto all'HW. Mentre ogni guest prevede un driver virtuale semplificato, cioè i **front-end driver**, che consentano l'accesso al device tramite il back-end driver contenuto

nel Domain 0. Questa soluzione comporta dei vantaggi come ad esempio la portabilità, l'isolamento e la semplificazione del VMM, ma al contrario necessita di una continua comunicazione con il back-end.

Gestione interruzioni e eccezioni -> la gestione delle interruzioni viene virtualizzata in modo molto semplice, cioè ogni interruzione viene gestita direttamente dal guest.

Un caso particolare riguarda il **Page Fault** (eccezione di tipo trap generata quando un processo cerca di accedere ad una pagina che è mappata nello spazio di indirizzamento virtuale, ma che non è presente nella memoria fisica): la gestione non può essere delegata completamente al guest, perché richiede l'accesso al registro CR2 contenente l'indirizzo che l'ha causato, ma è accessibile solo nel ring 0, quindi la gestione del page fault deve coinvolgere il VMM. La routine di gestione eseguita da XEN legge il contenuto di CR2 e lo copia in una variabile dello spazio del guest; successivamente viene trasferito il controllo al guest che andrà finalmente a gestire il page fault.

Gestione MV -> il compito della VMM è quello di gestire le MV, compresa la creazione/allocazione, spegnimento/accensione, l'eliminazione e le migrazione live. In particolare quindi il VMM opera sulle MV con le stesse operazioni che un SO mette in atto per la gestione dei processi.

Una MV si può trovare nei seguenti stati:

- **Running**: la MV è accesa e occupa memoria nella RAM del server su cu è allocata
- **Inactive**: la MV è spenta ed è rappresentata nel file system tramite un file immagine
- **Paused**: la MV è in attesa d un evento
- **Suspended**: la MV è stata sospesa dal VMM tramite il comando suspend; il suo stato e le risorse utilizzate sono salvate nel file system. L'uscita dallo stato di sospensione avviene tramite l'operazione resume da parte del VMM.

Migrazione MV -> in datacenter di server virtualizzati sempre più sentita la necessità di gestione agile delle MV per far fronte a quartazioni dinamiche del carico, la manutenzione online, la gestione finalizzata al risparmio energetico e la tolleranza ai guasti. Le MV possono essere spostate da un server a un altro senza essere spente e chi utilizza la MV in questione non si accorge minimamente che la macchina sia stata spostata.

Per effettuare una **migrazione live** mi basterebbe quindi effettuare una suspend di una MV sul nodo A ed effettuare una resume della stessa MV ma questa volta sul nodo B. In una migrazione live è desiderabile minimizzare il downtime cioè il tempo in cui la macchina non risponde alle richieste degli utenti, il tempo di migrazione e il consumo di banda. La soluzione adottata da XEN per effettuare una migrazione live è quella della **Precopy** che si svolge in 6 passaggi:

1. **Pre-migrazione**: individuazione della MV da migrare e dell'host di destinazione
2. **Reservation**: viene inizializzata una MV sul server di destinazione
3. **Pre-copia** iterativa delle pagine: viene eseguita una copia di tutte le pagine allocate in memoria sull'host A per la MV da migrare sull'host B; successivamente vengono iterativamente copiate da A a B tutte le pagine modificate (perché la MV sull'host A è ancora attiva e quindi modifica dei file) fino a quando il numero di dirty pages (pagine modificate) è inferiore a una soglia.
4. **Sospensione**: la MV viene sospesa e il suo stato insieme alle dirty pages viene copiato da A a B
5. **Commit**: la MV viene eliminata dal server A
6. **Resume**: la MV viene attivata nel server B

In alternativa alla Precopy si può utilizzare una tecnica chiamata **Postcopy** in cui la MV viene sospesa e vengono copiate pagine e stato. Il tempo di migrazione è più basso ma il downtime è moto più elevato. Posso utilizzare questa tecnica solo se la MV non deve sempre essere reattiva poiché la macchina viene congelata e poi copiata.

Il comando di migrazione viene eseguito da un demone di migrazione nel Domain 0 del server di origine della MV da migrare. Inoltre le pagine da copiare iterativamente vengono compresse per ridurre l'occupazione di banda durante il trasferimento.

02. PROTEZIONE

La **protezione** consiste nell'insieme di attività volte a garantire il controllo dell'accesso alle risorse logiche e fisiche da parte degli utenti. La **sicurezza** riguarda l'insieme delle tecniche con le quali regolamentare l'accesso degli utenti al sistema di elaborazione. Perciò mentre con protezione ci si riferisce ai dati interni al sistema, con sicurezza ci si riferisce all'intero sistema.

PROTEZIONE: MODELLI, POLITICHE E MECCANISMI ----------

Il controllo degli accessi è suddividibile in 3 livelli: **modelli**, **politiche** e **meccanismi**. Il livello più alto sono i modelli e si riferisce a quale modello di protezione il sistema operativo deve scegliere e mettere in atto, il livello intermedio è quello delle politiche e infine ci sono i meccanismi.

Modelli -> un modello di protezione definisce i soggetti, gli oggetti ai quali i soggetti hanno accesso ed i diritti di accesso. Gli **oggetti** costituiscono la parte passiva cioè le risorse fisiche e logiche alle quali si può accedere e su cui si può operare . I **soggetti** rappresentano la parte attiva di un sistema, cioè le entità che possono richiedere l'accesso alle risorse (ad esempio i processi che agiscono per conto degli utenti). I **diritti di accesso** sono le operazioni con le quali è possibile operare sugli oggetti. Un soggetto può avere diritti di accesso sia per gli oggetti che per altri soggetti.

A ogni soggetto è associato un dominio che rappresenta l'ambiente di protezione nel quale esegue e specifica i diritti di accesso posseduti dal soggetto nei confronti di ogni risorsa. Un dominio di protezione è unico per ogni soggetto mentre un processo può eventualmente cambiare dominio durante la sua esecuzione. Quindi l'associazione tra soggetto e dominio è statica, mentre l'associazione tra processo e soggetto/dominio è dinamica (ad esempio un processo P può eseguire nel dominio D1 per il soggetto S1 e poi eseguire nel dominio D2 per il soggetto S2).

Politiche -> le politiche di protezione definiscono le regole con le quali i soggetti possono acceder agli oggetti. Mentre il modello è qualcosa di insito nel sistema, le politiche in genere vengono scelte da chi opera su quel sistema. Esistono 3 diversi tipi di politiche:

- **DAC** (Discretional Access Control): il creatore di un oggetto controlla i diritti di accesso per quell'oggetto, quindi la gestione delle politiche è decentralizzata (UNIX)
- **MAC** (Mandatory Access Control): i diritti di accesso vengono definiti in modo centralizzato. Questa soluzione viene utilizzata in sistemi ad ala sicurezza per garantire assoluta confidenzialità e l diritti vengono gestiti da un'entità centrale.
- **RABC** (Role Based Access Control): ad ogni ruolo sono assegnati specifici diritti di accesso alle risorse, quindi gli utenti possono avere diversi ruoli.

Principio del privilegio minimo -> ad ogni soggetto sono garantiti i diritti di accesso solo agli oggetti strettamente necessari per la su esecuzione (caratteristiche desiderabile per tutte le politiche di protezione)

Meccanismi -> i meccanismi di protezione sono gli strumenti messi a disposizione dal sistema di protezione per imporre una determinata politica

Principi di realizzazione:

- **Flessibilità del sistema di protezione**: i meccanismi di protezione devono essere sufficientemente generali pe consentire l'applicazione di diverse politiche di protezione
- **Separazione tra meccanismi e politiche**: la politica definisce "cosa va fatto" ed il meccanismo invece definisce "come va fatto"

DOMINIO DI PROTEZIONE ----------

Un **dominio** definisce un insieme di coppie, ognuna contenente l'identificatore di un oggetto e l'insieme delle operazioni che il soggetto associato al dominio può eseguire su ciascuno oggetto (diritti di accesso). Ogni dominio è associato univocamente ad un soggetto; il soggetto può accedere solo agli oggetti definiti nel suo dominio utilizzando i diritti specificati dal dominio.

D(S) cioè Dominio del soggetto S specifica l'insieme degli oggetti su cui ha dei diritti e in particolare quali diritti sono ad esso associati.

Associazione processo dominio -> l'associazione tra processo e dominio può essere statica o dinamica. Nel caso **statico** l'insieme delle risorse disponibili ad un processo rimane fisso durante tutto il suo tempo di vita, nel caso **dinamico** invece l'associazione tra processo e dominio varia durante l'esecuzione del processo.

L'insieme globale delle risorse che un processo potrà utilizzare non può essere un'informazione disponibile prima dell'esecuzione del processo stesso. Inoltre l'associazione **statica** non è adatta nel caso di voglia limitare per un processo l'uso delle risorse a quello strettamente necessario (principio del privilegio minimo). Utilizzando l'associazione **dinamica** invece si può mettere in pratica il principio del privilegio minimo cambiando dinamicamente il dominio quando necessario.

MATRICE DEGLI ACCESSI ----------

Un sistema di protezione può essere rappresentato a livello astratto utilizzando il modello della **matrice degli accessi**. Ogni riga della matrice è associato a un soggetto (utente), ogni colonna invece è associata a un oggetto (risorsa, file). Il modello mantiene tutta l'informazione che specifica il tipo di accessi che i soggetti hanno per gli oggetti (stato di protezione). La matrice quindi offre ai meccanismi di protezione le informazioni che consentono di verificare il rispetto dei vincoli di accesso.

Il meccanismo associato al modello:

- Ha il compito di verificare se una richiesta di accesso che proviene da un processo che opera in un determinato dominio è consentita o meno
- Consente di modificare dinamicamente il numero degli oggetti e dei soggetti
- Consente ad un processo di cambiare dominio durante l'esecuzione
- Consente di modificare in modo controllato il cambiamento dello stato di protezione

Verifica del rispetto dei vincoli di accesso -> Il meccanismo consente di assicurare che un processo che opera nel dominio Dj possa accedere solo agli oggetti specificati nella riga i e solo con i diritti di accesso indicati. Quando un'operazione M deve essere eseguita nel dominio Di sull'oggetto Oj, il meccanismo consente di controllare che M sia contenuta nella casella access(i,j).

Modifica dello stato di protezione -> secondo la politica DAC gli utenti possono modificare lo stato di protezione, mentre nella politica MAC può essere fatto solo dall'entità centrale. La modifica controllata dello stato di protezione può essere ottenuta tramite un opportuno insieme di comandi:

- Create, delete object (aggiungere o eliminare le colonne della matrice, cioè gli oggetti)
- Create, delete subject (aggiungere o eliminare le righe della matrice, cioè i soggetti)
- Read, grant, delete, transfer access right (gestione diritti di accesso)

Propagazione diritti di accesso (Copy flag) -> la possibilità di copiare un diritto di accesso per un oggetto da un dominio ad un altro della matrice di accesso è indicato con un asterisco (*) che rappresenta il **copy flag**. L'operazione di propagazione può essere realizzata in due modi: **trasferimento del diritto**, quindi il soggetto S1 trasferisce read* e perde il diritto per l'oggetto O1, oppure per **copia del diritto**, quindi viene copiato solo read mentre il soggetto S1 mantiene read*.

Ad esempio se S1 possiede il diritto read sull'oggetto O1 allora S1 può propagare il diritto read anche ad S2 ma sempre sull'oggetto O1.*

Diritto owner -> chi possiede il diritto **owner** può assegnare/revocare un qualunque diritto di accesso sull'oggetto X di cui è owner, ad un qualunque altro soggetto.

Ad esempio se S2 ha il diritto owner su O2 allora può concedere il diritto write su O2 al soggetto S1.

Diritto control -> chi possiede il diritto **control** può revocare un qualunque diritto di accesso per l'oggetto X al soggetto Sj su cui ha i diritti di control.

Ad esempio se S1 possiede i diritti di control su S2 e S2 ha il diritto di write su O2 allora S1 può revocare il diritto di write di S2 su O2.

Cambio di dominio (diritto switch) -> un processo che esegue nel dominio del soggetto Si può commutare al dominio di un altro soggetto Sj. Questa operazione è consentita solo se il diritto di **switch** appartiene a A[Si,Sj].

Ad esempio se S2 ha il diritto di switch S1 allora un processo che esegue nel dominio di S2 può passare nel dominio di S1.

REALIZZAZIONE MATRICE EGLI ACCESSI ----------

La matrice degli accessi è una notazione astratta che rappresenta il sistema di protezione, ma poiché questa matrice può raggiungere dimensioni molto grandi, visto che ogni colonna rappresenta un possibile oggetto e quindi un file del mio sistema, la matrice sarà una **matrice sparsa** cioè che rispetto alle dimensioni effettive ci saranno moltissime celle vuote.

ACL (Access Control List) -> memorizzazione per colonne. Per ogni oggetto è associata una lista che contiene tutti i soggetti che possono accedere all'oggetto con i relativi diritti di accesso.

(CL) Capability List -> memorizzazione per righe. Ad ogni soggetto associata una lista che contiene gli oggetti accessibili dal soggetto ed i relativi diritti di accesso.

ACL (LISTA DEGLI ACCESSI) ----------

L'ACL per ogni oggetto è rappresentata dall'insieme delle coppie: **<soggetto, insieme dei diritti>**

Quando deve essere eseguita un'operazione M su un oggetto Oj sa parte di Si, si cerca nella ACL **<Si, Rk>** con M appartenente ad Rk. La ricerca può essere fatta preventivamente in una lista di default contenente i diritti di accesso applicabili a tutti gli oggetti. Se in entrambi i casi la risposta è negativa l'accesso è negato.

L'ACL è definita per utenti singoli, ma in molti sistemi esiste il concetto di gruppo di utenti, per cui ogni gruppo ha un nome e possono essere inclusi nella ACL. In questo caso avrà la forma **UID, GID: <insieme dei diritti>**.

Ruolo -> ogni utente può appartenere a gruppi diversi e quindi con diritti diversi.

CL (CAPABILITY LIST) ----------

La CL ha un approccio più efficiente rispetto alle ACL, che risulta più vantaggiosa ogni volta che dobbiamo compiere delle azioni che magari interessano tutti i soggetti per un particolare oggetto, come ad esempio la rimozione di un file. La CL per ogni soggetto è la lista di oggetti insieme agli accessi consentiti su di essi, quindi a cui il soggetto può accedere.

Ogni elemento della lista prende il nome di **capability** e garantisce al soggetto certi diritti su un certo oggetto. La capability si compone di un identificatore che identifica l'oggetto e una sequenza di bit che esprime i vari diritti.

Le CL devono assolutamente essere protette da manomissioni, e questo vengono gestite solo dal SO; l'utente quindi fa riferimento ad un puntatore che identifica la sua posizione nella lista appartenente allo spazio del kernel.

REVOCA DEI DIRITTI DI ACCESSO ----------

In un sistema di protezione dinamica può essere necessario revocare i diritti di accesso per un certo oggetto. La revoca può essere: **generale o selettiva**, cioè vale per tutti gli utenti che hanno quel diritto di accesso o solo per un gruppo, **parziale o totale** cioè si riferisce a un sottoinsieme di diritti per l'oggetto o tutti, oppure **temporanea o permanente**, cioè il diritto di accesso può essere riottenuto successivamente o non sarà più disponibile.

Revoca per un oggetto e ACL -> con le ACL la revoca risulta semplice. Si fa riferimento alla ACL associata all'oggetto e si cancellano i diritti di acceso che si vogliono revocare.

Revoca per un oggetto e CL -> l'operazione risulta più complessa poiché è necessario infatti verificare per ogni dominio se contiene la capability con riferimento all'oggetto desiderato.

SOLUZIONE MISTA ----------

Con le **ACL** l'informazione di quali diritti di accesso possieda un soggetto S è sparsa nelle varie ACL relative agli oggetti del sistema. Con le **CL** invece l'informazione relativa a tutti i diritti di accesso applicabili ad un certo oggetto O è sparsa nelle varie CL.

Un sistema di protezione che si basa solo su ACL o CL può presentare alcuni problemi di efficienza, per questo motivo vengono spesso utilizzate delle **soluzioni miste**, cioè utilizzare una combinazione dei due metodi.

Le ACL sono memorizzate in forma persistente ad esempio sul disco. Quindi se un soggetto tenta di accedere ad un oggetto per la prima volta: prima si analizza la ACL; se esiste una entry contenente il nome del soggetto e se tra i diritti di accesso è presente quello richiesto dal soggetto allora viene fornita la capability per l'oggetto. Ciò consente al soggetto di accedere all'oggetto più volte senza che sia necessario analizzare più volte la ACL, e solo in seguito all'ultimo accesso la capability è distrutta.

SICUREZZA MULTILIVELLO ----------

In certi ambiti in cui è necessario un controllo obbligatorio degli accessi al sistema, l'ente definisce delle politiche MAC che stabiliscano regole generali su chi può accedere e a che cosa. I modelli di sicurezza più utilizzati sono due: il **modello Bell-La Padula** e il **modello Biba**. Entrambi sono modelli **multilivello** in cui soggetti e oggetti sono classificati in livelli: livelli per i soggetti e livelli per gli oggetti.

I documenti sono classificati in 4 livelli: Non classificato, Confidenziale, Segreto, Top secret.

I soggetti sono classificati in 4 livelli: ogni persona è assegnata al livello a seconda dei documenti che è autorizzato ad esaminare.

Modello Bell-La Padula -> progettato per realizzare la sicurezza in ambiente militare, garantendo la confidenzialità delle informazioni, infatti è stato concepito per mantenere i segreti e non per garantire l'integrità dei dati. Questo modello associa a un modello di protezione 2 regole di sicurezza MAC che stabiliscono la direzione di propagazione delle informazioni nel sistema:

- **Proprietà di semplice sicurezza** -> un processo in esecuzione al livello di sicurezza k può leggere solo oggetti al suo livello o a quelli inferiori.
- **Proprietà di integrità *** -> un processo in esecuzione al livello di sicurezza k può scrivere solo oggetti al suo livello o a quelli superiori.

Modello Biba -> Questo modello a differenza di quello Bell-La Padula è stato concepito per garantire l'integrità dei dati, e come l'altro modello si basa su 2 regole di sicurezza:

- **Proprietà di semplice sicurezza** -> un processo in esecuzione al livello di sicurezza k può scrivere solo oggetti al suo livello o a quelli inferiori.
- **Proprietà di integrità *** -> un processo in esecuzione al livello k può leggere solo oggetti al suo livello o a quelli superiori.

Chiaramente i due modelli sono in conflitto tra loro e non possono essere usati contemporaneamente, quindi non possono essere combinati. Il **modello Bell-La Padula** permette la lettura verso il basso e la scrittura verso l'alto, mentre il **modello Biba** permette la lettura verso l'alto e la scrittura verso il basso.

REFERENCE MONITOR ----------

Sistemi fidati -> sistemi per i quali è possibile definire formalmente dei requisiti di sicurezza.

Reference monitor -> è un elemento di controllo realizzato dall'HW e dal SO che regola l'accesso dei soggetti agli oggetti sulla base di parametri di sicurezza del soggetto e dell'oggetto. Il RM ha accesso a una base di calcolo fidata (Trusted Computing Base, TCB) che contiene i **privilegi i sicurezza** di ogni soggetto e gli **attributi di protezione** di ogni oggetto.

Il RM impone le regole di sicurezza ed ha le seguenti proprietà:

- **Mediazione completa** -> le regole di sicurezza vengono applicate ad ogni accesso e non solo, ad esempio quando viene aperto un file
- **Isolamento** -> il monitor dei riferimenti e la base di dati sono protetti rispetto a modifiche non autorizzate
- **Verificabilità** -> la correttezza del monitor dei riferimenti deve essere provata cioè deve essere possibile dimostrare formalmente che il monitor impone le regole di sicurezza e fornisce mediazione completa ed isolamento.

I soggetti nell'accesso agli oggetti devono passare sempre per il RM il quale per capire se di può fare o meno tale accesso deve richiederlo prima al **TCB** (Trusted Computing Base) cioè la base di dati centrale della sicurezza.

Audit file -> vengono mantenuti in questo file eventi importanti per la sicurezza come i tentativi di violazione alla sicurezza e le modifiche autorizzate alla base di dati del nucleo di sicurezza.

03. PROGRAMMAZIONE CONCORRENTE

Con il termine **programmazione concorrente** si intende l'insieme delle tecniche, metodologie e strumenti per il supporto all'esecuzione di sistemi software composti da insiemi di attività svolte simultaneamente.

Primary memory
Level 2 cache
Level 1 cache
CPU

Memory		Memory
Interconnection network		
Cache		Cache
CPU		CPU

In un'architettura a **singolo processore** si ha una sola CPU con due livelli di cache e la memoria, mentre in un'architettura a **memoria condivisa e multiprocessore** abbiamo le memorie condivise te le varie CPU presenti tramite una rete di interconnessione.

In sistemi multiprocessore con un numero ridotto di processori la rete di interconnessione è realizzata da un memory bus. Inoltre il sistema che garantisce la connessione tra le CPU e la memoria è fatto in modo tale da garantire lo stesso tempo di accesso per ogni CPU (**UMA**, Uniform Memory Access). In sistemi invece con un numero elevato di processori la memoria è organizzata gerarchicamente, quindi ogni CPU ha memoria più vicine e più lontane, quindi il tempo di accesso alle memoria non sarà sicuramente uniforme (**NUMA**, Non Uniform Memory Access)

Interconnection Network		
Memory		Memory
Cache		Cache
CPU		CPU

Nell architettura **Distributed-Memory multicomputers and networks** invece ogni processore accede alla propria memoria che quindi non è condivisa, e il tempo di accesso è veloce ma è limitato alla sola memoria locale alla CPU.

Le architetture **Distributed-Memory** possono essere classificate in **Multicomputer**, in cui i processori e la rete sono fisicamente vicini, quindi si utilizza un collegamento diretto, oppure **Network System**, in cui I nodi sono collegati da una rete locale (ethernet) o da un rete internet.

Le applicazioni possono essere classificate in 3 tipologie:

- **Multithreaded** -> applicazioni strutturate come un insieme di processi per semplificare la loro programmazione e sono caratterizzate dal fatto che esistono più processi che non processori per eseguire i processi, quindi quest'ultimi sono schedulati ed eseguiti indipendentemente.
- **Sistemi distribuiti/sistemi multitasking** -> le componenti dell'applicazione (task) vengono eseguite su nodi collegati tramite opportuni mezzi di interconnessione e i processi comunicano scambiandosi dei messaggi. Tipica organizzazione Client/Server.
- **Applicazioni parallele** -> si vuole risolvere un dato problema più velocemente oppure un problema di dimensioni maggiori sempre nello stesso tempo, e per fare ciò sono eseguite su processori paralleli facendo uso di algoritmi appositi.

PROCESSI NON SEQUENZIALI E TIPI DI ITERAZIONE ----------

Algoritmo -> procedimento logico che deve essere eseguito per risolvere un determinato problema

Programma -> descrizione di un algoritmo mediante un opportuno linguaggio di programmazione che rende possibile l'esecuzione dell'algoritmo da parte di un particolare elaboratore.

Processo -> insieme ordinato degli eventi da cui dà luogo un elaboratore quando opera sotto il controllo di un programma.

Elaboratore -> entità astratta realizzata in HW e parzialmente in SW in grado di eseguire dei programmi

Evento -> esecuzione di un'operazione tra quelle appartenente all'insieme che l'elaboratore sa riconoscere ed eseguire; ogni evento determina una transizione di stato dell'elaboratore.

Processo sequenziale -> sequenza di stati attraverso i quali passa l'elaboratore durante l'esecuzione di un programma. Inoltre un processo è caratterizzato da stati, se gli stati sono rigidamente ordinati in sequenza allora abbiamo un processo sequenziale. (per una coppia di stati sono sempre in grado di dire quale dei due è venuto prima e quale dopo, poiché vi una **relazione d'ordine totale** tra gli stati)

Inoltre più processo possono essere associati dallo stesso programma, le **istanze**, ciascuna delle quali rappresenta l'esecuzione dello stesso codice con dati di ingresso diversi.

Grafo di precedenza -> un processo può essere rappresentato tramite un grafo di precedenza del processo costituito da nodi ed archi orientati: i nodi del grafo rappresentano i singoli **eventi**, mentre gli archi orientati rappresentano le **precedenze temporali** tra tali eventi. In parole semplici ogni nodo rappresenta un evento dell'esecuzione di un'operazione tra quelle appartenenti all'insieme che l'elaboratore sa riconoscere e eseguire.

Processo non sequenziale -> se i processo non sono sequenziali allora gli eventi non avvengono in sequenza e quindi l'insieme degli eventi che lo descrive è ordinato secondo una **relazione d'ordine parziale**.

L'esecuzione di un processo non sequenziale richiede: un **elaboratore non sequenziale, cioè un elaboratore in grado di eseguire più operazioni contemporaneamente,** e un **linguaggio di programmazione non sequenziale (o concorrente)** cioè un linguaggio che consenta di descrivere un insieme di attività concorrenti tramite moduli che possono essere eseguiti in parallelo.

Scomposizione di un processo non sequenziale -> se il linguaggio concorrente permette di esprimere il parallelismo a livello si sequenza di istruzioni allora si può scomporre un processo non sequenziale in un insieme di processi sequenziali eseguiti contemporaneamente. Le attività dei processi sono **indipendenti** quindi l'evoluzione di un processo non condiziona gli stati degli altri processi, però sono **interagenti** nel senso che sono assoggettati a vincoli di precedenza tra stati che appartengono a processi diversi (come ad esempio vincoli di precedenza tra le operazioni o vincoli di sincronizzazione).

Processi interagenti -> le iterazioni tra processi di lettura, elaborazione e scrittura sono relative ad uno scambio d informazioni. I dati su cui opera il processo di elaborazione sono forniti dal processo di lettura. I due processi quando arrivano ad un punto di iterazione corrispondente a d uno scambio di informazioni devono sincronizzarsi, mediante un **vincolo di sincronizzazione** cioè un vincolo imposto da ogni arco del grafo di precedenza che collega nodi di processi diversi.

ITERAZIONE TRA PROCESSI ----------

Le possibili iterazioni tra processi possono essere: cooperazione, competizione e interferenza.

Cooperazione -> comprende tutte le iterazioni prevedibili e desiderate insite nella logica dei programmi, prevede scambio di informazioni con trasmissione di dati (messaggi -> comunicazione) o senza trasferimento di dati (segnali temporali). È presente una relazione di causa e effetto tra l'esecuzione dell'operazione di invio da parte del processo mittente e l'esecuzione dell'operazione di ricezione da parte del processo ricevente; ovviamente deve esistere un vincolo di precedenza tra questi due eventi.

Competizione -> rappresenta un'iterazione prevedibile non desiderata, ma necessaria. La macchina su cui sono eseguiti i processi mette a disposizione un numero limitato di risorse condivise tra i processi e la competizione ha come obiettivo il coordinamento dei processi nell'accesso alle risorse condivise. Per risorse

che non possono essere utilizzate contemporaneamente da più processi bisogna prevedere dei meccanismi di competizione. (Un esempio di competizione è la **mutua esclusione**)

Sezione critica -> la sequenza di istruzioni con le quali un processo accede a un oggetto condiviso con altri processi. La regola di mutua esclusione stabilisce che sezioni critiche appartenenti alla stessa classe devono escludersi mutuamente nel tempo.

Interferenza -> interazione provocata da errori di programmazione, cioè rappresenta un'interazione non prevista e non desiderata: dipende dalla velocità relativa dei processi e gli errori possono manifestarsi nel corso del programma a seconda delle diverse condizioni di velocità di esecuzione dei processi. Uno degli obiettivi della **programmazione concorrente** è l'eliminazione delle interferenze.

ARCHITETTURE E LINGUAGGI PER LA PROGRAMMAZIONE CONCORRENTE ----------

Avendo a disposizione una macchina concorrente (cioè in grado di eseguire più processi sequenziali insieme) e di un linguaggio di programmazione con il quale descrivere algoritmi non sequenziali p possibile scrivere e dar eseguire programmi concorrenti. L'elaborazione complessiva può essere descritta come un insieme di **processi sequenziali asincroni interagenti**.

Le proprietà di un **linguaggio di programmazione concorrente**:

- Deve avere costrutti con i quali sia possibile dichiarare moduli di programma destinati ad essere eseguiti come processi sequenziali distinti
- Non tutti i processi vengono eseguiti contemporaneamente, infatti alcuni processi vengono svolti dinamicamente quindi solo se si verificano particolari condizioni. Bisogna quindi poter specificare quando un processo deve essere attivo e terminato.
- Devono essere presenti strumenti linguistici per specificare le iterazioni che dinamicamente potranno verificarsi tra i vari processi.

Architettura di una macchina concorrente -> la macchina concorrente M spesso è una macchina astratta creata equipaggiando un macchina fisica con una macchina SW che crea un livello di astrazione e consente di risolvere i problemi di gestione. Al proprio interno M contiene ciò che deve essere messo in atto quando richiediamo l'esecuzione di processi concorrenti e tutto quello che riguarda l'iterazione (sincronizzazione con scambio di informazioni).

Nel nucleo della macchina astratta oltre ai **meccanismi di multiprogrammazione e sincronizzazione** è presente anche un **meccanismo di protezione** che permette di rilevare eventuali interferenze tra i processi e può essere realizzato sia in HW che in SW.

Meccanismo di multiprogrammazione -> è quello preposto alla gestione delle unità di elaborazione della macchina reale consentendo ai vari processi eseguiti sulla macchina astratta M di condividere l'uso delle unità reali di elaborazione.

Meccanismo di sincronizzazione e comunicazione -> è quello che estende le potenzialità delle unità reali di elaborazione rendendo disponibile alle unità virtuali strumenti con cui sincronizzarsi e comunicare.

Architettura della macchina M può avere due diverse organizzazioni:

- **Modello a memoria comune** -> in cui l'iterazione tra i processi avviene su oggetti contenuti nella memoria comune (ambiente globale)
- **Modello a scambio di messaggi** -> la comunicazione e la sincronizzazione tra i processi si basa sullo scambio di messaggi sulla rete che collega gli elaboratori.

COSTRUTTI LINGUISTICI PER LA SPECIFICA DELLA CONCORRENZA ----------

Fork/Join -> l'esecuzione di una **fork** coincide con la creazione e l'attivazione di un processo che inizia la propria esecuzione in parallelo con quella del processo chiamante. La **join** consente di determinare quando un processo creato tramite la fork ha terminato il suo compito sincronizzandosi con tale evento. Mette in pausa chi la esegue nell'attesa della terminazione dell'altro.

Cobeign/Coend -> La concorrenza viene espresso tramite in costrutto **cobegin/coend** al cui interno vengono eseguite delle istruzioni S1, S2, … Sn che sono eseguite in parallelo.

Processo -> costrutto linguistico per individuare in modo sintatticamente preciso quali moduli di un programma possono essere eseguiti come processi autonomi. La keyword **process** o qualunque altra keyword messa a disposizione per definire un processo permette di indicare quali moduli possono essere eseguiti come processi autonomi: ad esempio **process** <identificatore> (<parametri formali>) { … }

PROPRIETA DEI PROGRAMMI ----------

Una delle attività più importanti per chi sviluppa programmi è la **verifica di correttezza** dei programmi realizzati.

Traccia dell'esecuzione (Storia) -> sequenza degli stati attraversati dal sistema di elaborazione durante l'esecuzione del programma.

Stato -> insieme dei valori delle variabili definite nel programma insieme alle variabili implicite.

Programmi sequenziali -> ogni esecuzione di un programma P su un particolare insieme di dati D genera la stessa traccia. La verifica può essere agevolmente svolta tramite il **debugging**.

Programmi concorrenti -> l'esito dell'esecuzione dipenda da quale sia l'effettiva sequenza cronologica di esecuzione delle istruzioni contenute (poiché non abbiamo vincoli su cosa esegue prima e cosa dopo), quindi ogni esecuzione di un programma P su un particolare insieme di dati D dà origine a una **traccia dell'esecuzione** diversa. La verifica è molto complessa in un programma concorrente poiché il semplice debug su una esecuzione non dà la garanzia sul soddisfacimento di una proprietà data.

Proprietà dei programmi -> una proprietà di un programma P è un attributo che p sempre vero in ogni possibile traccia generata dall'esecuzione di P, in generale vengono classificate in due categorie: **safety properti**es e **liveness properties**.

Proprietà SAFETY -> proprietà che garantisce che durante l'esecuzione di P non si entrerà mai in uno stato errato (stato in cui le variabili assumono valori non desiderati).

Proprietà LIVENESS -> proprietà che garantisce che durante l'esecuzione di P prima o poi si entrerà in uno stato corretto (stato in cui le variabili assumono valori desiderati).

Un **programma sequenziale** deve avere fondamentalmente due proprietà:

- **Correttezza del risultato finale** -> quindi per ogni esecuzione il risultato ottenuto è giusto (SAFETY)
- **Terminazione** -> quindi prima o poi l'esecuzione termina (LIVENESS).

Un **programma concorrente** invece deve avere fondamentalmente le stesse proprietà di un programma sequenziale con in aggiunta queste altre proprietà:

- **Mutua esclusione nell'accesso a risorse condivise** -> quindi per ogni esecuzione non accadrà mai che più di un processo acceda contemporaneamente alla stesa risorsa (SAFETY)
- **Assenza di deadlock** -> quindi per ogni esecuzione non si verificheranno mai situazioni di blocco critico (SAFETY)
- **Assenza di starvation** (blocco critico in attesa di acquisire una risorsa) -> cioè prima o poi ogni processo potrà accedere alle risorse richieste (LIVENESS).

04. MODELLO A MEMORIA COMUNE

I modelli di iterazione tra processi sono due: modello a **memoria comune** e modello a **scambio di messaggi**.

MODELLO A MEMORIA COMUNE ----------

Il sistema è visto come un insieme di **processi** e **oggetti**, con i processi che hanno diritto i accesso sugli oggetti. Il tipo di iterazione tra i processi è di **competizione** o **cooperazione**.

Il modello a memoria comune rappresenta la naturale astrazione del funzionamento di un sistema in multiprogrammazione costituito da uno o più processori che hanno accesso ad una memoria comune. Ad ogni processore può essere associata una memoria privata, ma ogni iterazione avviene tramite oggetti contenuti nella **memoria comune** (ambiente globale, accessibile a tutti).

Ogni applicazione viene strutturata come un insieme di componenti suddiviso in due sottoinsiemi disgiunti: i **processi** (componenti attivi) e le **risorse** (componenti passivi) che a loro volta sono suddivise in risorse di tipo **primitivo** e di tipo **astratto**.

Risorsa -> qualunque oggetto fisico o logico di cui un processo necessita per portare a termine il suo compito. Le risorse sono raggruppate in classi e una classe identifica l'insieme di tutte le operazioni che un processo può eseguir per operare su risorse di quella classe. Il termine risorsa si identifica con quello di **struttura dati** allocata nella memoria comune.

MECCANISMO DI CONTROLLO DEGLI ACCESSI ----------

Necessità di specificare quali processi ed in quali istanti possono accedere alla risorsa, in modo tale da controllare che gli accessi avvengano in maniera corretta.

Gestore di una risorsa -> per ogni risorsa R il suo gestore definisce in ogni istante t l'insieme SR(t) dei processi che in quell'istante hanno il diritto di operare su R.

Una risorsa può essere classificata come:

- **Dedicata** -> se SR(t) ha una cardinalità sempre <= 1
- **Condivisa** -> se la risorsa non è dedicata
- **Allocata staticamente** -> se SR(t) è una costante. SR(t) viene definito prima dell'esecuzione.
- **Allocata dinamicamente** -> se SR(t) è funzione del tempo. Il gestore definire SR(t) in fase di esecuzione e quindi dinamicamente.

Il caso A è l'unico in cui la risorsa è privata, mentre in tutti gli altri casi B, C e D le risorse sono comuni. Nei casi B e D però la competizione si verifica nel momento in cui due o più processi tentano di accedere alla stessa risorsa. Nel caso C invece è il gestore a decidere chi può accedere a una particolare risorsa. Ma lo stesso gestore è una risorsa quindi ci sarà competizione nell'accedere al gestore.

	Dedicata	Condivisa
Alloc. Static.	A) privata	B) comune
Alloc. Dinam.	C) comune	D) comune

Compiti del gestore di una risorsa -> Il gestore di una risorsa deve svolgere alcuni compiti:

- Mantenere aggiornato l'insieme SR(t) e cioè lo stato di allocazione della risorse di cui è gestore
- Fornire i meccanismi che un processo può utilizzare per acquisire il diritto di operare sulla risorsa entrando a far parte dell'insieme SR(t) e per rilasciare tale diritto quando non più necessario
- Implementare la strategia di allocazione della risorsa e definire quando, a chi e per quanto tempo allocare la risorsa.

Il gestore Gr di una risorsa R è costituito da una **risorsa condivisa** in un sistema organizzato secondo il modello a **memoria comune**, un **processo** in un sistema organizzato secondo il modello a **scambio di messaggi**.

Se R è **allocata staticamente** a P il processo possiede il diritto di operare in qualunque istante sulla risorsa R. Se R è **allocata dinamicamente** a P è necessario prevedere il gestore Gr che implementa le funzioni di richiesta e rilascio, il processo P deve quindi eseguire il protocollo: richiesta – esecuzione – rilascio.

Se R p una risorsa **condivisa** è necessario assicurare che gli accessi avvengano in modo non divisibile, quindi l'accesso alla risorsa deve essere programmato come **sezione critica**. Se R è una risorsa **dedita**, essendo P l'unico processo che può accedervi non è necessario prevedere nessun meccanismo di sincronizzazione.

TIPO DI ITERAZIONE ----------

Competizione -> con la **modalità B** la competizione tra processi avviene al momento dell'accesso alla risorsa e l'accesso esclusivo è garantito dal meccanismo di mutua esclusione utilizzato nel programmare le funzioni di accesso. Con la **modalità C** la competizione avviene al momento dell'accesso alle operazioni del gestore invece.

Cooperazione -> con la **modalità B** si ha cooperazione se uno dei processi memorizza in R informazioni che possono essere lette da un altro processo. Con la **modalità C** si ha cooperazione se un processo acquista dal gestore il diritto di operare su R vi memorizza informazioni e se successivamente il diritto di accesso viene acquisito da un altro processo che legge quelle informazioni.

SEZIONE CRITICA ----------

Specifica della sincronizzazione: **region** R << Sa; when(C) Sb; >>

Il corpo della **region** rappresenta un'operazione da eseguire sulla risorsa condivisa R e quindi costituisce una sezione critica che deve essere eseguita n mutua esclusione con le altre operazioni definite su R. il corpo della region è costituito da due istruzioni da eseguire in sequenza: l'istruzione Sa e successivamente dopo aver valutato la condizione C l'istruzione Sb (se C è falsa si attende).

Casi particolari per le regioni critiche:

- Region R << S; >> : specifica della sola mutua esclusione senza che sia prevista alcuna forma di sincronizzazione diretta
- Region R << when(C) >> : specifica di un semplice vincolo di sincronizzazione, quindi il processo attende che C sia verificata prima di eseguire
- Region R << when(C) S; >> : specifica il caso in cui la condizione C di sincronizzazione caratterizza lo stato in cui la risorsa R deve trovarsi al fine di poter eseguire l'operazione S.

PROBLEMA DELLA MUTUA ESCLUSIONE (RIPASSO) ----------

Il problema della **mutua esclusione** nasce quando più di un processo alla volta può aver accesso a variabili comuni. La regola di mutua esclusione impone che le operazioni con le quali i processi accedono alle variabili comuni non si sovrappongano nel tempo.

Soluzioni possibili:

- **Algoritmiche** -> algoritmo di Dekker, Algoritmo di Paterson, Algoritmo del fornaio
- **HW** -> disabilitazione delle interruzioni, lock/unlock
- **Strumenti sincronizzazione realizzati dal nucleo della macchina concorrente** -> semaforo e monitor

SEMAFORO ----------

Semaforo -> strumento linguistico di basso livello che consente di risolvere problemi di sincronizzazione nel modello a memoria comune. Il meccanismo semaforico è realizzato dal nucleo della macchina concorrente.

Definizione semaforo -> un semaforo è una variabile intera non negativa cui è possibile accedere solo tramite le due operazioni P e V. essendo il semaforo l'oggetto condiviso le due operazioni P e V vengono definite come **sezioni critiche** da eseguire in mutua esclusione.

```
void P (semaphore s):
      region s << when (vals>0) vals--; >>

void V(semaphore s):
      region s << vals++; >>
```

Il semaforo viene utilizzato come strumento di sincronizzazione tra processi concorrenti: sospensione (P(s), s==0) e risveglio (V(s), se vi è almeno un processo sospeso). Il semaforo è uno strumento generale che consente la risoluzione di qualunque problema di sincronizzazione.

Proprietà del semaforo -> dato un semaforo s siano:

- val_s : valore dell'intero non negativo associato al semaforo
- I_s: valore intero >= 0 con cui il semaforo s viene inizializzato
- nv_s: numero di volte che l'operazione V(s) è stata eseguita
- np_s: numero di volte che l'operazione P(s) è stata eseguita

Relazione di invarianza -> ad ogni istante possiamo esprimere il valore del semaforo come $val_s = I_s + nv_s - np_s$. La relazione invarianza è sempre soddisfatta per ogni semaforo qualunque sia il suo valore e comunque sia il programma concorrente che lo sta utilizzando.

ESEMPI NOTEVOLI DI UTILIZZO DEI SEMAFORI ----------

Semafori mutua esclusione -> semafori inizializzato a 1 viene utilizzato per realizzare le sezioni critiche di una stessa classe. Il semaforo mutua esclusione può assumere solo i valori 0 e 1 (**semaforo binario**). Le condizioni necessarie sono le seguenti:

- sezioni critiche della stessa classe devono essere eseguite in modo **mutuamente esclusivo**
- non deve essere possibile il verificarsi di situazioni in cui i processi impediscono mutuamente la prosecuzione della loro esecuzione (**deadlock**)
- infine quando un processo si trova all'esterno di una sezione critica non può rendere impossibile l'accesso alla stessa sezione ad altri processi.

```
Semaphore mutex = 1;
...
P(mutex);
<sezione critica>
V(mutex);
```

Semafori evento -> (scambio di segnali temporali) un semaforo evento è un semaforo binario utilizzato per imporre un vincolo di precedenza tra le operazioni dei processi.

```
Semaphore sem = 0;
```

```
P1:                          P2
...                          ...
P(sem);                      operazioneB;
operazioneA;                 V(sem);
...                          ...
```

Due processi P1 e P2 eseguono due operazioni Pa e Pb il primo mentre il secondo esegue le operazioni Qa e Qb. Secondo il **vincolo di rendez-vous** l'esecuzione di Pb da parte di P1 e Qb da parte di P2 possono iniziare solo dopo che entrambi i processi hanno completato la loro prima operazione Pa e Qa.

```
Semaphore sem1 = 0;
Semaphore sem2 = 0;

P1:                                 P2
Pa                                  Qa
V(sem1);                            V(sem2);
P(sem2);                            P(sem1);
Pb                                  Qb
```

Semafori binari composti -> (scambio di dati) due processi P1 e P2 si scambiano dati di tipo T utilizzando una memoria condivisa e l'acceso al buffer deve essere mutuamente esclusivo. P2 può prelevare und dato solo dopo che P1 'ha inserito e P1 prima di poter inserire un nuovo dato deve aspettare che P2 l'abbia prelevato.

Utilizziamo due semafori vu per realizzare l'attesa di P1 in caso di buffer pieno e pn per realizzare l'attesa di P2 in caso di buffer vuoto. Poiché il buffer inizialmente è vuoto avremo vu=1 e pn=0;

```
Semaphore vu = 1; //il buffer inizialmente è vuoto
Semaphore pn = 0;

void invio(T dato) {
      P(vu);    //val_vu--
      Inserisci(dato);
      V(pn);    //val_pn++
}
```

```
void ricezione() {
      P(pn);    //val_pn--
      T dato = estrai();
      V(vu);    //val_vu++
      Return dato;
}
```

pn e vu garantiscono da soli la mutua esclusione delle operazioni estrai e inserisci, inoltre la coppia di semafori si comporta come se fosse un unico semaforo binario di mutua esclusione, poiché il valore di un semaforo è sempre l'inverso dell'altro e quindi prende il nome di **semaforo binario composto**.

Semafori condizione -> l'esecuzione di un'istruzione S1 su una risorsa R è subordinata al verificarsi di una condizione C, inoltre l'attesa della condizione C deve comportare l'uscita della sezione critica permettendo quindi ad altri processi di accedere a quella risorsa R e poter eseguire altre operazioni.

```
Void op1( ): region R << when(C) S1; >>
```

Si possono effettuare due tipi di schemi per l'accesso alla risorsa R: uno prevede l'**attesa circolare** quindi viene utilizzato un ciclo while per verificare la condizione e il ciclo continuerà fino a che la condizione non è verificata; l'altra possibilità è quella di utilizzare uno schema con **passaggio di testimone** in cui viene utilizzato un if per verificare la condizione e nel caso di condizione non verificata deve abbandonare la struttura ciclica.

Il secondo schema è più efficiente del primo ma consente di svegliare un solo processo alla volta poiché ad uno solo può passare il diritto di operare in mutua esclusione.

Semafori risorsa -> semafori generali, che possono assumere qualunque valore maggiore o uguale a zero e vengono impiegati per realizzare l'allocazione di risorse equivalenti: il valore del semafori rappresenta il numero di risorse libere.

Ad esempio il problema dei Produttori e Consumatori, in cui il Produttore richiede l'allocazione di una risorsa "elemento vuoto" mentre il Consumatore l'allocazione di una risorsa "elemento pieno"

Semafori privati -> un semaforo si dice privato per un processo quando su tal processo può eseguire la primitiva P sul semaforo s, mentre la primitiva V sul semaforo s può essere eseguita anche da altri processi. Un semaforo privato viene sempre inizializzato con valore zero. Possono essere utilizzati per realizzare particolari politiche di allocazione di risorse.

REALIZZAZIONE DEI SEMAFORI ----------

In sistemi operativi multi programmati il **semaforo** vinee realizzato dal kernel che sfruttando i meccanismi di gestione dei processi 8riattivazione e sospensione) elimina la possibilità di attesa.

```
typedef struct {
      int contatore;
      coda queue;
} semaforo;
```

Una **P** su un semaforo con contatore a 0, sospende il processo nella coda queue altrimenti se il contatore è maggiore di zero allora viene decrementato. Una **V** su un semaforo la cui coda non è vuota estrae un processo dalla cosa, nel caso in cui la coda sia vuota invece incrementa il contatore.

L'**implementazione di P e V** viene svolta dal nucleo della macchina concorrente e dipende quindi dal tipo di architettura HW (monoprocessore o multiprocessore) e da come il nucleo rappresenta e gestisce i processi concorrenti.

05. NUCLEO DI UN SISTEMA MULTIPROGRAMMATO (MODELLO A MEMORIA COMUNE)

Si chiama **nucleo** (kernel) il modulo realizzato in SW, HW e firmware che supporta il concetto di processo e realizza gli strumenti necessari per la gestione dei processi. Il nucleo costituisce il livello più interno di un qualunque sistema basato su processi.

modello a processi -> modello che prevede l'esistenza di tante unità di elaborazione (MV) quanti sono i processi. Ogni macchina possiede come set di istruzioni elementari quelle corrispondenti all'unità centrale reale più le istruzioni relative alla creazione ed eliminazione dei processi, al meccanismo di comunicazione e sincronizzazione

Le caratteristiche fondamentali del nucleo sono:

- **Efficienza** -> condiziona l'intera struttura a processi. Per questo motivo esistono sistemi in cui alcune o tutte le operazioni del nucleo sono realizzate in HW o tramite microprogrammi.
- **Dimensione** -> la semplicità delle funzioni richieste al nucleo fa sì che la sua dimensione risulti estremamente limitata.
- **Separazione tra meccanismi e** politiche -> il nucleo deve contenere solo meccanismi consentendo così a livello di processi di utilizzare tali meccanismi per la realizzazione di diverse politiche di gestione.

Stati di un processo -> le transizioni tra i due stati (ATTIVO/BLOCCATO) sono implementate dai meccanismi di sincronizzazione realizzati dal nucleo. Ad esempio tramite un semaforo: P per sospensione e V per risveglio.

Quando un processo perde il controllo del processore il contenuto dei registri viene salvato in un'area di memoria associata al processo chiamata **descrittore**. Ciò consente una maggiore flessibilità nella politica di assegnazione del processore ai processi, rispetto a salvare le informazioni nello stack; infatti in un qualunque istante uno qualsiasi dei processi pronti può ottenere l'uso del processore.

Contesto di un processo -> è l'insieme delle informazioni contenute nei registri del processore e relative al processo.

Il nucleo deve svolgere determinate operazioni:

- **Gestire il salvataggio ed il ripristino dei contesti dei processi** -> quando un processo abbandona il controllo della CPU tutte le informazioni contenute nei registri devono essere trasferite nel descrittore. Analogamente quando un processo riprende l'esecuzione tutte le informazioni contenute nel descrittore devono essere trasferite nei registri.
- **Scegliere a quale tra i processi assegnare la CPU** -> quando un processo abbandona il controllo della CPU il nucleo deve scegliere a quale tra tutti i processi pronti assegnarla, magari secondo alcune priorità o semplicemente in modalità FIFO.
- **Gestire le interruzioni dei dispositivi esterni** -> tradurre le interruzioni in attivazioni di processi da bloccato a pronto o viceversa.
- **Realizzare meccanismi di sincronizzazione dei processi** -> realizzare meccanismi di sincronizzazione gestendo il passaggio dei processi dallo stato di esecuzione allo stato di bloccato e viceversa oppure per la preparazione di un descrittore per un processo ed il suo inserimento nella coda dei processi pronti e viceversa.

REALIZZAZIONE DEL NUCLEO (ARCHITETTURA MONOPROCESSORE) – STRUTTURE DATI ----------

Descrittore del processo -> contiene **l'identificatore del processo**, la **modalità di servizio dei processi** (FIFO, priorità, deadline e quanto di tempo), **contesto del processo** e **identificatore del processo successivo**.

Coda dei processi pronti -> esistono una o più code di processi pronti. Quando un processo è riattivato per effetto di una V, viene inserito al fondo della coda corrispondente alla sua priorità. La coda dei processi contiene sempre almeno un processo fittizio (dummy process) che va in esecuzione quando tutte le altre code sono vuote, e rimane in esecuzione fino a quando qualche altro processo diventa pronto.

Descrittori liberi -> coda nella quale sono concatenati i descrittori disponibili per la creazione di nuovi processi e nella quale sono ritornati i descrittori dei processi eliminati.

Processo in esecuzione -> il nucleo necessita conoscere quale processo è in esecuzione. Questa informazione rappresentata dall'indice del descrittore del processo viene contenuta in un particolare registro del processore.

REALIZZAZIONE DEL NUCLEO (ARCHITETTURA MONOPROCESSORE) – FUNZIONI DEL NUCLEO ----------

Le funzioni del nucleo realizzano le operazioni di transizione di stato per i singoli processi; si utilizzano quindi a questo scopo due procedure: **Inserimento** e **Prelievo** di un descrittore da una coda.

Le funzioni del nucleo possono essere divise in due livelli:

- **Livello superiore** -> contiene tutte le funzioni direttamente utilizzabili dai processi sia esterni che interni come: **risposta** ai segnali di interruzione, **creazione**, **eliminazione** e **sincronizzazione** dei processi.
- **Livello inferiore** -> realizza le funzionalità di cambio di contesto: **salvataggio** del contesto del processo che si sospende nel suo descrittore, **scelta** di un nuovo processo da mettere in esecuzione tra quelli pronti e **ripristino** del contesto.

L'ambiente di esecuzione delle funzioni del nucleo ha caratteristiche diverse da quelle dei processi, infatti i due ambienti corrispondono a stati diversi di operazione dell'elaboratore (kernel o user). Le funzioni del nucleo devono essere eseguite in modo mutuamente esclusivo.

Passaggio di ambiente -> nel caso di funzioni chiamate da processi esterni il passaggio all'ambiente del nucleo è ottenuto mediante il meccanismo di **risposta al segnale di interruzioni**. Nel caso di funzioni chiamate da processi interni invece il passaggio ottenuto mediante l'esecuzione di system call. In entrambi i casi comunque il trasferimento all'ambiente user avviene tramite il meccanismo di **ritorno da interruzioni**.

Cambio di contesto -> salvataggio del contesto del processo in esecuzioni nel suo descrittore, **inserimento del descrittore** nella coda dei processi bloccati o dei processi pronti, **rimozione del processo a maggio priorità** dalla coda dei processi pronti e caricamento dell'identificatore di tale processo nel registro della CPU, infine **caricamento del contesto** del nuovo processo nei registri della macchina.

Gestione del temporizzatore -> per consentire la modalità di servizio a divisione di tempo è necessario che il nucleo gestisca un dispositivo temporizzatore tramite un'apposita procedura che ad intervalli di tempo fissati provveda a sospendere il processo in esecuzione ad assegnare la CPU ad un altro processo.

Semafori -> nel nucleo del sistema monoprocessore il semaforo può essere implementato tramite una *variabile intera* che rappresenta il suo valore e un *puntatore ad una lista di descrittori* di processi in attesa sul semaforo (se non ci sono processi il puntatore vale null). Il descrittore viene inserito nella coda del semaforo tramite la primitiva P e viene prelevato per effetto della primitiva V.

Meccanismo di passaggio dall'ambiente del nucleo all'ambiente dei processi e viceversa -> è costituito dal meccanismo di interruzioni. Al completamento della funzione richiesta il trasferimento all'ambiente di utente avviene utilizzando il **meccanismo di ritorno da interruzione**.

REALIZZAZIONE DEL NUCLEO (ARCHITETTURA MULTIPROCESSORE) ----------

Per quanto riguarda le architetture multiprocessore sono possibili vari modelli:

- **Modello SMP** -> unica copia del nucleo condivisa tra tutte le CPU (l'unico nucleo si occupa di tutto)
- **Modello a nuclei distinti** -> più istanze di nucleo concorrenti (esiste una collezioni di nuclei che esegue in modo concorrente)

1. Modello SMP (Simmetric Multi Processing) -> esiste un'unica copia del nucleo nella memoria comune che si occupa della gestione di tutte le risorse disponibili (come ad esempio di tutte le CPU). Ogni processo può operare su ogni CPU. La **competizione tra le CPU** nell'esecuzione del nucleo necessita di un meccanismo di sincronizzazione.

Sono possibili due soluzioni pe quanto riguarda l'accesso al nucleo SMP:

- **Soluzione ad un solo lock** -> assegno al nucleo un lock e ogni accesso esclusivo al nucleo può essere ottenuto solo tramite le primitive **lock** e **unlock**, in modo tale da bloccare gli accessi agli altri processi.
- **Limitazione del grado di parallelismo** -> si esclude a priori la possibilità di eseguire in modo contemporaneo più funzioni del nucleo, ad esempio utilizzando due P su due semafori.
- **Soluzione a più lock** -> per ottenere un maggior grado di concorrenza si può suddividere il nucleo in molteplici sezioni critiche indipendenti ognuna con un proprio semaforo e una coda di processi. È possibile accedere alle sezioni critiche tramite **lock** e **unlock**.

Realizzazione semaforo (Modello SMP) -> Tutte le CPU condividono lo stesso nucleo, quindi per sincronizzare gli accesi al nucleo, le strutture dati del nucleo vengono protette tramite lock in particolare utilizzando singoli **semafori** e **code dei processi** distinti. In questo caso due operazioni P su semafori diversi possono operare in modo contemporaneo, in caso contrario vengono sequenzializzati solo gli accessi alla coda dei processi pronti.

Se si ha uno scheduling pre-emptive basato su priorità allora si rischia che l'esecuzione di una V possa portare l'attivazione di un processo con priorità superiore a quella di almeno uno dei processi in esecuzione. Quindi il nucleo deve provvedere a revocare la CPU al processo con priorità più bassa ed assegnarla al processo risvegliato dalla V. Inoltre sarà necessario un **meccanismo di segnalazione** tra le unità di elaborazione.

2. Modello a nuclei distinti -> ogni singola CPU viene gestita da un nucleo distinto. Questo modello si basa sull'ipotesi che l'insieme di processi sia partizionabile in tanti **nodi virtuali**, ciascuno dei quali è assegnato ad un **nodo fisico**. Tutte le informazioni relative al nodo virtuale vengono allocate sulla memoria privata del nodo fisico. Se nella memoria privata vengono allocate anche le funzioni del nucleo allora tutte le iterazioni locali al nodo virtuale possono avvenire indipendentemente e in modo concorrente a quelle degli altri nodi virtuali. Solo le iterazioni tra processi appartenenti a nodi virtuali diversi utilizzano la memoria comune.

Realizzazione semaforo (Modello a nuclei distinti) -> solo le iterazioni tra processi appartenenti a nodi virtuali diversi utilizzano la memoria comune. Bisogna distinguere tra **semafori privati** cioè utilizzati da processi appartenenti al nodo virtuale in questione e vengono realizzati e gestiti come nel caso monoprocessore, e i **semafori condivisi** tra nodi cioè utilizzati da processi appartenenti a nodi virtuali diversi. La memoria comune dovrà tenente tutte le informazioni relative ai semafori condivisi.

Ogni semaforo condiviso sarà protetto da un lock e per utilizzarlo bisognerà usare le funzioni **lock** e **unlock**.

Rappresentante del processo -> insieme minimo di informazioni sufficienti per identificare sia il nodo fisico su cui il processo opera, sia il descrittore contenuto nella memoria privata del processo.

Differenza tra i due modelli -> il secondo modello è più vantaggioso in quanto è maggiormente scalabile e permette un alto grado di parallelismo tra le CPU. Il primo modello invece fornisce i presupposti per un

miglior bilanciamento del carico poiché è lo scheduler a decidere di allocare i processi sulle CPU da lui scelte, mentre il secondo modello vincola ogni processo alla CPU del proprio nodo.

06. MONITOR

Monitor -> costrutto sintattico che associa un insieme di operazioni (public o entry) ad una struttura dati comune a più processi. Le operazioni **entry** sono le sole operazioni permesse su quella struttura e sono **mutuamente esclusiv**e quindi un solo processo alla volta può essere attivo nel monitor.

```
monitor tipo_risorsa {
      <dichiarazioni variabili locali>;
      <inizializzazione variabili locali>;

      public void op1 ( ) {
            <corpo della operazione op1 >;
      }
      public void opn ( ) {
            <corpo della operazione opn>;
      }
      <eventuali operazioni non public>
}
```

Le **operazioni public** (o entry) sono le sole operazioni che possono essere utilizzate dai processi per accedere alle **variabili locali**, le quali mantengono il loro valore tra successive esecuzioni delle operazioni del monitor. Le variabili locali sono accessibili sono entro il monitor. Le operazioni **non public** invece non saranno accessibili dall'esterno e sono utilizzabili solo all'interno del monitor.

USO DEL MONITOR ----------

Scopo del monitor è controllare l'assegnazione di una risorsa tra processi concorrenti in accordo a determinate politiche di gestione. Le variabili locali definiscono lo stato della risorsa associata al monitor.

L'assegnazione delle risorse avviene secondo due livelli di controllo:

- Il primo garantisce che solo un processo alla volta possa aver accesso alle variabili comuni del monitor e ciò è ottenuto garantendo che le operazioni public siano eseguite in modo mutuamente esclusivo (eventuale sospensione dei processi nella coda della entry)
- Il secondo controlla l'ordine con il quale i processi hanno accesso alla risorsa. La procedura chiamata verifica il soddisfacimento di una condizione logica che assicura l'ordinamento (eventuale sospensione del processo in una coda associata alla condizione)

Nel caso in cui la condizione non sia verificata la sospensione del processo avviene utilizzando variabili di un tipo detto **condition**. La condizione di sincronizzazione è costituita da variabili locali al monitor e da variabili proprie del processo passate come parametri.

Variabili tipo condizione -> ogni variabile di tipo condizione (dichiarazione: **condition** cond) rappresenta una coda nella quale i processi si sospendono ed è possibile svolgere due operazioni su tale variabile: **wait(cond)** e **signal(cond)**

Wait -> l'esecuzione dell'operazione **wait(cond)** sospende il processo introducendolo nella coda individuata dalla variabile cond e il monitor viene liberato.

Signal -> l'esecuzione dell'operazione **signal(cond)** rende attivo un processo in attesa nella coda individuata dalla variabile cond.

Come conseguenza della signal entrambi i processi quello segnalante Q e quello segnato P possono concettualmente proseguire la loro esecuzione. Si possono utilizzare due possibili strategie: **signal_and_wait** e **signal_and_continue**.

Signal_and_wait -> P riprende immediatamente l'esecuzione ed il processo Q viene invece sospeso, si evita quindi la possibilità che Q proseguendo possa modificare la condizione di sincronizzazione rendendola non più vera per P. Quindi Q si sospende nella coda dei processi che attendono di usare il monitor

Signal_and_continue -> Q prosegue la sua esecuzione mantenendo l'accesso esclusivo al monitor dopo aver risvegliato i processo P. Il processo P viene trasferito dalla coda associata alla variabile condizione alla *entry_queue* e potrà rientrare nel monitor solo dopo che Q l'abbia rilasciato. Poiché alcuni processi potrebbero entrare ne monitor prima di P e potrebbero modificare alcune condizioni di sincronizzazione è fondamentale che P verifichi la condizione quando rientra nel monitor.

Signal_and_urgent_wait -> è una variante della **signal_and_wait**, in questo caso Q ha la priorità rispetto agli altri processi che aspettano di entrare nel monitor. Viene quindi sospeso in una coda interna al monitor. Quando P poi ha terminato la sua esecuzioni trasferisce il controllo a Q senza liberare il monitor.

Un caso particolare della **signal_and_urgent_wait** si ha quando essa corrisponde a una istruzione return diventando quindi **signal_and_return**.

Risvegliare tutti i processi -> con il metodo **signal_all** è possibile svegliare tutti i processi sospesi sulla variabile condizione; in questo caso tutti i processi risvegliati vengono messi nella *entry_queue* dalla quale uno alla volta potranno rientrare nel monitor.

Wait(cond, p) -> sospensione con indicazione della priorità. I processi sono accodati rispettando il valore di p e vengono risvegliati nello stesso ordine.

Empty(cond) -> verifica dello stato della coda. Fornisce il valore false se esistono processi sospesi nella coda associata a cond, altrimenti restituisce true.

Variabili condizione monoprocesso -> è possibile prevedere una array di variabili condizione una per ogni processo che prendono il nome di **variabili condizione monoprocesso**. Ogni processo può sospendersi sulla propria variabile condizione e questo consente di risvegliare un ben determinato processo.

COSTRUTTO MONITOR TRAMITE SEMAFORI ----------

Il compilatore assegna ad ogni istanza di un monitor un **semaforo** mutex inizializzato a q per la mutua esclusione delle operazioni del monitor. La richiesta di un processo di eseguire un'operazione public equivale quindi all'esecuzione di una P(mutex).

Il compilatore assegna ad ogni variabile cond di tipo **condition** un **semaforo** condsem inizializzato a 0 sul quale il processo si può sospendere tramite una wait(condsem) e un **contatore** condcount inizializzato a 0 per tenere conto dei processi sospesi su condsem.

Signal_and_continue

```
wait(cond): {
      condcount++;
      V(mutex);
      P(condsem);
      P(mutex);
}

signal(cond): {
      if (condcount>0){
            condcount--;
            V(condsem);
      }
}
```

Signal_and_wait

```
wait(cond): {
      condcount++;
      V(mutex);
      P(condsem);
}

signal(cond): {
      if (condcount>0) {
            condcount-- ;
            V(condsem);
            P(mutex);
      }
}
```

Signal_and_urgent_wait

```
wait(cond): {
      condcount++;
      if (urgentcount>0)
            V(urgent);
      else V(mutex);
            P(condsem);
      condcount-- ;
}

signal(cond): {
      if (condcount>0) {
            urgentcount++;
            V(condsem);
            P(urgent);
            urgentcount-- ;
      }
}
```

Signal_and_return

```
wait(cond): {
      condcount++;
      V(mutex);
      P(condsem);
      condcount-- ;
}

signal(cond): {
      if (condcount>0)
            V(condsem);
      else V(mutex);
}
```

`urgent`: semaforo per la sospensione del processo segnalante con valore inziale 0
`urgentcount`: contatore dei processi sospesi sul semaforo urgent

REALIZZAZIONE POLITICHE GESTIONE DELLE RISORSE ----------

Allocazione di una risorsa con la strategia Shortest-job-next -> più processi competono per l'uso di una risorsa. Quando questa viene rilasciata essa viene assegnata tra tutti i processi sospesi a quello che la userà per il periodo di tempo minore.

MONITOR NELLA LIBRERIA PTHREAD ----------

La libreria **pthread** non prevede il costrutto monitor ma offre comunque gli strumenti di sincronizzazione necessari ad implementare il monitor. Come il mutex (semaforo di mutua esclusione) e le variabili condizione.

Variabile condizione -> consente la sospensione dei thread in attesa che sia soddisfatta una condizione logica. Una variabile condizione è definita dal tipo **pthread_cond_t** che rappresenta una coda per la sospensione dei thread. Ogni variabile condizione viene acceduta all'interno del monitor che la definisce quindi in questo cado all'interno di una sezione critica protetta dal mutex.

Le operazione fondamentali sulle variabili condizioni sono:

- **Inizializzazione** -> `pthread_cond_init(&C, attr);`
  C individua la condizione da inizializzare, mentre attr punta a una struttura dati che contiene gli attributi della condizione, se NULL viene inizializzata a default.
- **Sospensione** -> `pthread_cond_wait(pthread_cond_t *C, pthread_mutex_t *M);`
  Il thread chiamante viene sospeso nella coda associata a C, M viene liberato e al risveglio del thread M viene rioccupato automaticamente.
- **Risveglio** -> `pthread_cond_signal(pthread_cond_t *C);`
  Se esistono thread sospesi nella coda associate a C viene risvegliato il primo, se invece non ci sono thread sospesi la signal non produce alcun effetto. Si utilizza la politica **signal_and_continue**.

Per ogni sezione critica deve essere associato un mutex per la mutua esclusione e verranno utilizzate le operazioni di **lock**, `pthread_mutex_lock(&M)`, per il prologo dell'operazione e di **unlock**, `pthread_mutex_unlock(&M)`, come epilogo dell'operazione.

Riepilogo **simulazione del comportamento del monitor**:

- Mutua esclusione delle operazioni del monitor: uso di **lock** e **unlock** su un **mutex** all'inizio ed al termine di ogni operazione public.
- Sospensione su una variabile condizione utilizzando la **pthread_ cond_ wait**.
- Riattivazione con **pthread_ cond_ signal**.
- Viene adottata la politica **signal_and_ continue**.
- Non c'è controllo da parte del compilatore.

07. MODELLO A SCAMBIO DI MESSAGGI

Aspetti caratterizzanti il modello -> ogni processo può accedere alla ricorse allocate nella propria memoria (virtuale) locale poiché si utilizza un'architettura del tipo **distributed-memory multicomputers and networks**. Inoltre ogni risorsa del sistema è accessibile ad un solo processo, quindi se una risorsa è necessaria a più processi ciascuno di questi (processi Client) dovrà richiedere all'unico processo che può operare sulla risorsa (processo Server) di eseguire l'operazione richiesta e restituirgli i risultati.

Il processo Server rappresenta anche il gestore della risorsa in questo modello architetturale. Il meccanismo base utilizzato dai processi per qualunque tipo di iterazione è costituito dal meccanismo di **scambio di messaggi**.

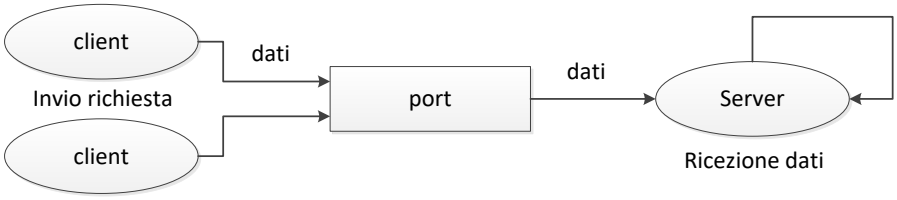
Canali di comunicazione -> un **canale** è un collegamento logico mediante il quale due o più processi comunicano. È compito del linguaggio di programmazione offrire gli strumenti di alto livello per specificare i canali di comunicazione e utilizzarli per le iterazioni tra processi.

I parametri che caratterizzano il concetto di canale sono:

- La **tipologia del canale**, intesa come direzione del flusso di dati che un canale può trasferire. Può essere **monodirezionale** o **bidirezionale** (sia per inviare che per ricevere).
- La designazione del canale e dei processi **sorgente** e **destinatario** di ogni comunicazione
- Il **tipo di sincronizzazione** tra i processi.

I canali possono essere di 3 tipi in relazione al numero di processi comunicanti:

- **Link** canale simmetrico -> da uno a uno (tipo Produttore/Consumatore)
- **Port** canale asimmetrico -> da molti a uno (tipo Client/Server)
- **Mailbox** canale asimmetrico -> da molti a molti



I canali possono avere 3 differenti tipo di sincronizzazione:

- **Comunicazione asincrona** -> il processo mittente continua la sua esecuzione immediatamente dopo che il messaggio è stato inviato. La **send** quindi non è un punto di sincronizzazione, e la comunicazione asincrona crea un disaccoppiamento tra mittente e destinatario.
  A livello implementativo richiede la presenza di un **buffer** ci capacità illimitata per poter memorizzare i messaggi inviati e non ancora ricevuti. In caso di coda dei messaggi pieni c'è bisogno di sospendere il processo che invia i messaggi.
- **Comunicazione sincrona** -> (rendez-vous semplice) il primo dei due processi comunicanti che esegue o l'invio o la ricezione si sospende in attesa che l'altro sia pronto ad eseguire l'operazione duale. Non è necessario un **buffer** perché un messaggio viene inviato solo se l'altro è pronto a riceverlo

- **Comunicazione con sincronizzazione estesa** -> (rendez-vous esteso) il processo mittente rimane in attesa fino a che il ricevente non ha terminato di svolgere l'azione richiesta. Modello Client/Server in cui il Client rimane in attesa fino a che il server non ha terminato l'operazione richiesta.

PRIMITIVE DI COMUNICAZIONE ----------

Port -> un canale di comunicazione **port** identifica un canale asimmetrico molti a molti. L'identificatore denota un canale utilizzato per trasferire messaggi di tipo specificato. Port viene dichiarato locale al processo ricevente e viene utilizzato dal mittente tramite la dot notation: <nome processo>.<identificatore port>

```
port <tipo> <identificatore>        es: port int ch1
```

Send -> primitiva di invio: `send(<valore>) to <porta>`
<porta> identifica in modo univoco il canale a cui inviare il messaggio, mentre <valore> identifica una espressione dello stesso tipo di <porta> e il cui valore rappresenta il contenuto del messaggio inviato.

Receive -> primitiva di ricezione: `receive(<variabile>) from <porta>`
<porta> identifica il canale, locale al processo ricevente dal quale ricevere il messaggio, <variabile> invece è l'identificatore di una variabile dello stesso tipo di <porta> a cui assegnare il valore del messaggio ricevuto. Il processo che la esegue si sospende se non sono presenti messaggi sul canale.

Comando con guardia -> una guardia può risultate **fallita** se l'espressione booleana ha il valore falso, **ritardata** se l'espressione booleana ha valore true ma la receive è bloccante poiché sul canale con ci sono messaggi pronti quindi il processo che la esegue viene bloccato (all'arrivo del messaggio viene riattivato ed esegue <istruzione>), oppure **valida** se l'espressione booleana ha valore true e la receive esegue senza ritardi.

```
<guardia> -> <istruzione>
<guardia> := (<espressione booleana>; <primitiva receive>)
```

Comando con guardia alternativo -> un comando con guardia alternativo è un'istruzione complessa formata da più comandi con guardia semplice, le quali si trovano all'interno di un blocco if e vengono definiti rami.

```
if
      [] <guardia_1> -> <istruzione_1>;
      ...
      [] <guardia_n> -> <istruzione_n>;
fi
```

vengono valutate le guardie di tutti i rami; se una o più guardie sono valide viene scelto in maniera casuale uno dei rami con guardia valida e viene quindi eseguita l'istruzione relativa a quel ramo; se tutte le guardie non fallite sono ritardate, il processo si sospende in attesa che arrivi un messaggio che abilita una guardia, rendendola valida e a quel punto si procede eseguendo l'istruzione di quel ramo; se tutte le guardie sono fallite il comando termina.

Comando con guardia ripetitivo -> assomiglia a un comando con guardia alternativo ma i rami sono contenuti all'interno di un ciclo

```
do
      [] <guardia_1> -> <istruzione_1>;
      ...
      [] <guardia_n> -> <istruzione_n>;
od
```

il funzionamento del **comando con guardia ripetitivo** è lo stesso del comando con guardia alternativo solo che la valutazione ed esecuzione dei rami viene svolta in maniera ciclica. Se tutte le guardie sono fallite solo in quel caso si esce dal ciclo.

PRIMITIVE DI COMUNICAZIONE ASINCRONE ----------

Nel modello a scambio di messaggi lo strumento di comunicazione più utilizzato a basso livello è la **send asincrona**. Prendiamo in considerazione alcuni tipici problemi di iterazione nel modello a scambio di messaggi del tipo **accesso a risorse condivise -> processi servitori**:

- Una sola operazione
- Più operazioni mutuamente esclusive
- Più operazioni con condizioni di sincronizzazione

RISORSA CONDIVISA CON UNA SOLA OPERAZIONE

Memoria comune	**Monitor** con una operazione entry
Scambio di messaggi	**Processo servitore** che offre un **unico servizio** senza condiz. di sincronizzazione

RISORSA CONDIVISA CON PIU OPERAZIONI

Memoria comune	**Monitor** con più operazioni entry

Scambio di messaggi	**Processo servitore** che offre 2 servizi senza condizioni si sincronizzazione. • **Soluzione senza comandi con guardi**a -> un solo canale per entrambe le richieste • **Soluzione con comando con guardia** -> due diversi canali per i due tipi di risorse differenti.

RISORSA CONDIVISA CON PIU OPERAZIONI E CONDIZIONI DI SINCRONIZZAZIONE

Memoria comune	**Monitor** con 2 entry e 2 variabili condizione
Scambio di messaggi	**Processo servitore** con più servizi con la specifica di condizioni di sincronizzazione

REALIZZAZIONE DELLE PRIMITIVE ASINCRONE ----------

Realizzazione delle primitive di comunicazione e costrutto port utilizzando gli strumenti di comunicazione offerti dal nucleo del sistema operativo. Riferimento alle primitive asincrone in quanto più primitive (le primitive sincrone possono essere tradotte dal compilatore in termini di primitive asincrone).

Architetture mono e multielaboratore e **architetture distribuite**.

ARCHITETTURE MONO E MULTIELABORATORE ----------

1. Tutti i messaggi scambiati tra i processi sono di u unico tipo T predefinito a livello del nucleo.
2. Tutti i canali sono da molti ad uno (port) e quindi associati al processo ricevente
3. Essendo le primitive asincrone ogni porta deve contenere buffer (coda dei messaggi) di lunghezza indefinita

ARCHITETTURE DISTRIBUITE ----------

Sistemi operativi distribuiti (DOS) -> Insieme di nodi tra loro omogenei e tutti dotati dello stesso sistema operativo (quindi con lo stesso nucleo), con lo scopo di gestire tutte le risorse nascondendo all'utente la loro distribuzione sulla rete.

Sistemi operativi di Rete (NOS) -> Insieme di nodi eterogenei con sistemi operativi diversi e autonomi. Ogni nodo della rete è in grado di offrire servizi a clienti remoti presenti su altri nodi della rete. Trasparenza e distribuzione delle risorse viene ottenuta mediante il middleware (tra SO e applicazioni).

L'unità di trasmissione tra nodi è il **pacchetto,** la cui struttura dipende dalla realizzazione cioè dai protocolli di comunicazione adottati. L'**interfaccia di rete** è strutturata in 2 canali: uno per la trasmissione dei messaggi e l'altro per l'invio.

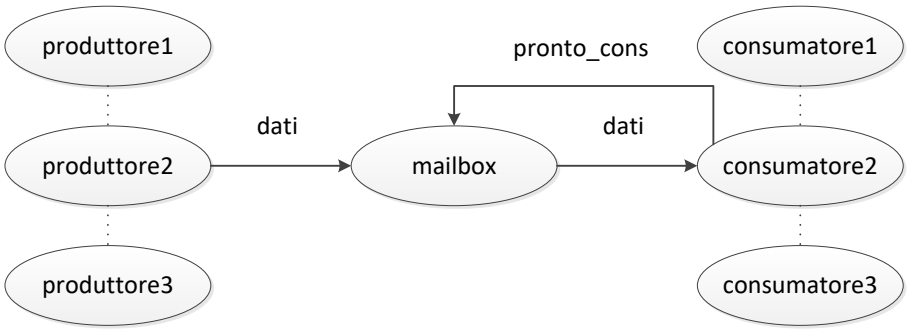
Il **canale di trasmissione** viene acceduto per realizzare l'invio di pacchetti, e ad esso sono associati: una **coda di pacchetti** nella quale ogni processo deposita il proprio pacchetto se il canale è occupato da un altro processo e un **registro buffer** della dimensione di un pacchetto. Il canale di ricezione invece viene acceduto per realizzare la ricezione dei pacchetti, ad esso è associato un registro buffer nel quale viene depositato il pacchetto inviato al nodo.

Il pacchetto è l'informazione che viene trasmessa attraverso i canali e oltre al messaggio contiene anche le informazioni relative al processo destinatario e alla porta di destinazione.

PRIMITIVE DI COMUNICAZIONE SINCRONE ----------

Confronto tra le primitive sincrone e le asincrone -> minor grado di concorrenza delle primitive sincrone rispetto a quelle asincrone. Con le primitive sincrone non è più necessaria la presenza di buffer.

Mailbox -> i dati vengono inseriti nella mailbox che rappresenta il gestore con il compito di gestire i dati e quindi bufferizzare i dati e che sfrutteremo come una coda di messaggi. Rispetto al caso della send asincrona il canale non è in grado di bufferizzare i messaggi inviati quindi c'è la necessità di una struttura dati interna al processo mailbox che funga da **coda di messaggi**.



Mailbox concorrente -> si può tradurre la mailbox come una pipeline di processi quindi viene inserita tra produttore e consumatori questa mailbox costituita da più processi in cascata, che saranno tanti quanti gli elementi del buffer. Ogni processo della mailbox si trova in pipeline con il processo precedente, che gli invia il messaggio, e quello successivo, a cui invece trasferiscono il messaggio. L'ultimo processo comunica direttamente con il consumatore mentre il primo processo preleva il messaggio direttamente dal produttore.

08. RPC (Comunicazione con sincronizzazione estesa)

Chiamata di operazione remota -> meccanismo di comunicazione e sincronizzazione tra processi in cui un processo richiede un servizio ad un altro processo e rimane in sospeso fino al completamento del servizio richiesto. I due processi rimangono sincronizzati durante l'esecuzione del servizio da parte del ricevente fino alla ricezione dei risultati da parte del mittente (**rendez-vous esteso**).

La differenza con una normale **chiamata a funzione** è che il servizio viene eseguito remotamente da un processo diverso dal chiamante.

Due diverse modalità di esecuzione del servizio richiesto:

- **Chiamata di procedura remota (RPC)** -> per ogni operazione che un processo client può richiedere, viene dichiarata lato server una **procedura** e per ogni richiesta di operazione viene creato un uovo processo servitore con il compito di eseguire tale procedura.
- **Rendez-vous** -> l'operazione richiesta viene specificata come un **insieme di istruzioni** che può comparire in un punto qualunque del processo servitore (Ad esempio ADA). Il processo servitore utilizza un'istruzione di input (**accept**) che lo sospende in attesa di una richiesta dell'operazione. All'arrivo di una richiesta il processo server esegue il relativo insieme di istruzioni e i risultati sono inviati al richiedente.

RPC rappresenta solo un meccanismo di comunicazione tra processi. Il rendez-vous invece combina comunicazione con sincronizzazione; esiste infatti un solo processo servitore al cui interno sono definite le istruzioni che consentono di realizzare i vari servizi richiesti. Tale processo di sincronizza con il processo client quando esegue l'operazione **accept**.

RENDEZ-VOUS ----------

```
accept<servizio>(in <par-ingresso>, out<par-uscita>); -> {S1,..,Sn};
```

Accept -> se non sono presenti richieste di servizio l'esecuzione di accept provoca la sospensione del processo servitore. Se lo stesso servizio è richiesto da più processi prima che il servitore esegua la accept, le richieste vengono inserite in una coda associata al servizio e gestita in maniera FIFO.

Lo schema di comunicazione realizzato dal meccanismo di rendez-vous è del tipo asimmetrico **da molti a uno**.

Selezione delle richieste -> nel modello rendez-vous il server può selezionare le richieste da servire in base al suo stato interno utilizzando i **comandi con guardia**.

```
if
      []<stato1>; accept<servizio1>(in <par-ingresso>, out<par-uscita>);
            -> {S11,..,S1n}; ...
      []<stato2>; accept<servizio2>(in <par-ingresso>, out<par-uscita>);
            -> {S21,..,S2n}; ...
      ...
end;
```

Selezione delle richieste in base ai parametri di ingresso -> la decisione se servire o no una richiesta può dipendere oltre che dallo stato della risorsa anche dai parametri della richiesta. Infatti la **guardia logica** che condiziona l'esecuzione dell'azione richiesta deve essere espressa anche in termini dei parametri di ingresso. È necessari auna **doppia iterazione** tra processo client e server: la prima per trasmettere i parametri della richiesta e la seconda per richiedere il servizio.

Vettore di operazioni di servizio -> nell'ipotesi di un numero limitato di differenti richieste si può ottenere una semplice soluzione al problema associando ad ogni richiesta una differente operazione di servizio cioè un **vettore di operazioni di servizio**.

LINGUAGGIO ADA ----------

Linguaggio che utilizza come metodo d'iterazione tra processi (task) il **rendez-vous** con comunicazione ti tipo asimmetrico. Ogni task può definire delle operazioni pubbliche (entry) visibili da altri task, e il rendez-vous tra i due task viene stabilito proprio quando un processo Q chiama una entry di P.

Lo schema base di un task contiene una **parte di specifica** che definisce le operazioni entry ed una **parte body** che consente la realizzazione di tali operazioni.

Rendez-vous -> una entry dichiarata in un task P e resa visibile all'esterno di P può essere chiamata da un altro task Q. La comunicazione tra P e Q avviene quando P esprime la volontà di eseguire la entryname mediante la **accept**.

```
Q:    call P.entryname (<parametri effettivi>);
P:    accept entryname (in <par-ingresso>, out<par-uscita>);
      do S1; S2; ... Sn; end;
```

L'esecuzione di accept entryname da parte di P sospende il task fino a quando non vi è una chiamata di entryname. In quel momento i parametri effettivi sono copiati nei parametri formali e viene eseguita la lista di istruzioni. Al completamento i risultati sono copiati nei parametri di uscita e termina la sincronizzazione tra P e Q.

COMANDO CON GUARDIA FORMA SEMPLICE

```
select
         accept entry1 do..end;
      or accept entry2 do..end;
      ...
      or accept entryn do..end;
end select;
```

COMANDO CON GUARDIA FORMA COMPLETA

```
select
         when cond1 -> accept entry1 do..end;
      or when cond2 -> accept entry2 do..end;
      ...
      or when condn -> accept entryn do..end;
end select;
```

PROGRAMMAZIONE CONCORRENTE IN ADA ----------

Obiettivi del linguaggio Ada:

- Ridurre i costi di sviluppo e manutenzione
- Prevenire bugs
- Rilevare bugs il prima possibile
- Favorire riutilizzo e sviluppo in team
- Semplificare la manutenzione
- Adatto per lo sviluppo in the small e in the large

Ada è un linguaggio **fortemente e staticamente tipato**. Inoltre il programmatore può definire nuovi tipi per esprimere pienamente le caratteristiche del dominio applicativo.

TIPI DI DATO

Tipi scalari	Array	Record	Access (Puntatori)

ISTRUZIONI DI CONTROLLO

if … end if;	loop … end loop;	for … end loop;

Concorrenza -> l'esecuzione concorrente è basata sul concetto di **task** (Task -> Processo). Un task descrive un'attività che può essere eseguita in concorrenza con altre.

```
task <nome_task> is ... end <nome_task>;            --dichiarazione
task body <nome_task> is ...  end <nome_task>;      --definizione
```

Iterazione tra task -> l'iterazione tra task avviene attraverso il meccanismo del rendez-vous. Un task chiama una entry di un altro task cioè un'operazione che un task rende disponibile agli altri task.

Select -> Un task server può esporre più operazioni e accettare le richieste attraverso il comando **select**. Inoltre ad ogni **accept** è associata una coda: se entrambe sono vuote la select sospende il task in attesa di richieste, se invece almeno una contiene una richiesta allora viene fatta una selezione in modo non deterministico (come comando con guardia).

```
select
      accept E(...) do
      ...
      end
or
      accept A(...) do
      ...
      end
end select;
```

Select con guardie logiche -> è possibile utilizzare anche il comando **select** con le **guardie logiche**, e si ottiene un costrutto con la stessa semantica del comando con guardia alternativo.

```
select
      when cond1 => accept E(...)
            do ...
            end E;
or
      when cond2 => accept A(...)
            do ...
            end A;
end select;
```

09. LE AZIONI ATOMICHE

Azione atomica -> strumento di alto livello per la strutturazione di programmi concorrenti e/distribuiti tolleranti ai malfunzionamenti. Utilizzati nella costruzione di programmi tolleranti vari tipi di malfunzionamenti.

L'azione atomica è un'operazione che porta un insieme di oggetti O={ O1, O2, … On } da uno stato consistente S1 a uno stato consistente S2.

Ogni oggetto astratto può trovarsi in stati consistenti o inconsistenti a seconda che si sia verificata o meno una particolare relazione fra i valori delle variabili componenti l'oggetto. Ogni tipo T ha una sua **relazione invariante** che lo caratterizza dal punto di vista semantico. Ogni operazione primitiva su dati di tipo T deve essere programmata in modo da lasciare l'oggetto in uno stato **consistente**.

Necessità di **mantenere la consistenza** dell'insieme degli oggetti al termine dell'esecuzione del programma; quindi se gli oggetti inizialmente si trovano in uno stato consistente anche alla fine devono necessariamente trovarsi in uno stato consistente. Durante l'esecuzione dell'operazione l'insieme O può passare per **stati inconsistenti** ma non devono essere visibili ad altre operazioni.

> *Un esempio di consistenza dei dati e parallelismo possono essere delle operazioni di trasferimento: dati gli oggetti O1 e O2 sostare X sa O1 a O2. Valore O1 + Valore O2 = Costante.*
>
> *Durante l'esecuzione del programma quando x è stato tolto da O1 ma non ancora sommata ad O2 si ha uno **stato inconsistente**. La **mutua esclusione** sui singoli oggetti garantisce l'atomicità delle operazione ma non dell'intera operazione nel complesso. È necessario che l'intera operazione che interessa gli oggetti O1 e O2 sia considerata **atomica** e quindi non divisibile.*

Consistenza dei dati e malfunzionamento -> fino ad ora abbiamo considerato l'azione atomica come un'operazione che in forma non divisibile fa transitare un insieme di oggetti da uno stato consistente S1 ad uno stato finale S2 anch'esso consistente. Affinché gli oggetti non rimangano in uno stato inconsistente è necessario un meccanismo di recupero che in seguito ad una condizione anomala porti gli oggetti in uno stato consistente.

SOLUZIONI ----------

Tutto o niente -> indicando con S' lo stato risultante dell'esecuzione di una azione atomica di deve avere che **S'=S1** oppure **S'=S2** dove S1 e S2 sono stato iniziale e stato finale. Con "tutto" si intende il completamento dell'operazione e con "niente" l'interruzione e il ripristino dello stato iniziale.

Per garantire consistenza dei dati lazione atomica deve possedere due **proprietà fondamentali**:

- **Serializzabilità** -> assicura che ogni azione operi sempre su un insieme di oggetto il cui stato iniziale è consistente ed i cui stati parziali non sono visibili ad altri processi concorrenti.
- **Tutto o niente**

Two phase lock protocol -> Sia A una azione atomica che opera su un insieme di oggetti O={O1, O2, … On} e siano Richiesta(Oi) e Rilascio(Oj) le operazioni per richiedere al gestore di ogni oggetto l'uso esclusivo dell'oggetto specificato. La serializzabilità viene garantita allocando dinamicamente singoli oggetti in modo dedicato alle azioni atomiche secondo il seguente protocollo:

1. Ogni oggetto deve essere acquisito da una azione atomica A in modo esclusivo prima di qualunque azione su di esso. Richiesta(Oi) è bloccante se l'oggetto Oi non è disponibile.
2. Nessun oggetto deve essere rilasciato prima che siano eseguite tutte le operazioni su di esso
3. Nessun oggetto può essere richiesto dopo che è stato effettuato un rilascio di un altro oggetto.

Questo protocollo è sufficiente a garantire la proprietà di **serializzabilità** e si chiama **two phase lock protocol** perché è suddiviso in due fasi:

- **Fase I (fase crescente)** -> l'azione atomica acquisisce in modo esclusivo tutti gli oggetti ed opera su di essi
- **Fase II (fase calante)** -> inizia non appena viene eseguito il primo rilascio e durante essa non possono essere acquisiti ulteriori oggetti.

Effetto domino -> l'aborto di un'azione atomica genera come effetto collaterale l'aborto di una diversa azione atomica e così via. L'azione atomica rilascia un oggetto prima di ver completato la sequenza di operazioni su tutti gli oggetti e di aver raggiunto uno stadio di avanzamento tale che da quel punto in poi l'azione non sarà più abortita.

Per soddisfare la proprietà del **tutto o niente** occorre definire un'ulteriore requisito:

4. Nessun oggetto può essere rilasciato prima che l'azione atomica abbia completato la sua esecuzione, quindi i rilasci devono costituire le ultime operazioni dell'azione atomica.

Commit -> per garantire la proprietà tutto o niente è necessario che il meccanismo di recupero si comporti in maniera diversa a seconda dell'istante in cui l'evento anomalo si verifica. Dovrà **abortire** se il malfunzionamento avviene quando gli oggetti sono in uno stato inconsistente oppure dovrà **garantire il completamento dell'azione** quando gli oggetti sono nello stato finale. A questo scopo viene introdotta l'operazione **commit** che viene eseguita quando tutti gli oggetti sono giunti al valore finale e produce come effetto l'impossibilità dell'aborto.

Terminazione -> l'azione atomica termina in uno dei due modi: **terminazione normale** quando l'azione atomica completa la sequenza di operazioni e raggiunge lo stato finale, oppure **terminazione anomale** quando l'azione atomica non completa l'intera sequenza di operazioni e gli oggetti tornano al valore iniziale.

Cause della terminazione:

- Il verificarsi di un'eccezione sollevata durante una delle operazioni sugli oggetto, quindi il processo esegue la primitiva **abort**
- Il verificarsi di un malfunzionamento o di una condizione di blocco critico prima che le operazioni siano terminate. Il sistema di recupero da malfunzionamenti forza l'aborto dell'azione atomica

Meccanismo di ripristino -> il meccanismo di ripristino necessita di informazioni relative allo stato corrente delle operazioni ed allo stato iniziale degli oggetti. Le informazioni necessarie per il recupero vengono mantenute in memoria di massa.

Memoria stabile -> memoria con la proprietà di contenere le informazioni necessarie al recupero e di non essere soggetta ad alcun tipo di malfunzionamento. (questo perché esiste la possibilità di perdere le informazioni nella memoria di massa a seguito di un malfunzionamento HW)

AZIONI ATOMICHE MONO-PROCESSO ----------

Il processo che esegue un'azione atomiche può attraversare i seguenti stati:

- **Working** -> durante l'esecuzione del corpo dell'azione atomica, quindi gli oggetti sono inconsistenti. Se l'elaboratore cade il meccanismo di recupero deve abortire l'azione atomica.
- **Committing (commit)** -> durante la terminazione corretta dell'azione. Gli oggetti sono nel loro stato finale. Se l'elaboratore cade il meccanismo di recupero deve completare l'azione.
- **Aborting (abort)** -> durante la terminazione dell'azione atomica. Gli oggetti devono essere ripristinati al loro valore iniziale. Se l'elaboratore cade durante questo processo il meccanismo di recupero deve garantire il ripristino dei valori iniziali degli oggetti.

Le informazioni sullo stato in cui si trova il processo che esegue un'azione atomica vengono tenute all'interno di una struttura dati, il **descrittore di azione**, allocata in memoria stabile. La primitiva **begin action** crea in memoria stabile un descrittore di azione, inizializzando lo stato del processo a working.

All'inizio dell'azione atomica viene creata una copia degli oggetti in memoria volatile sulla quale eseguire le operazioni. L'aborto coincide con la distruzione delle copie in memoria volatile. Per quanto riguarda la terminazione invece solo se l'azione termina correttamente allora i valori finali delle copie di lavoro vengono salvati nella copia valida in memoria stabile.

Copia delle intenzioni -> La copia dei valori finali creata in memoria stabile prima della commit (si è ancora in stato working) senza modificare la copia stabile originale. Solo se la copia delle intenzioni avviene in modo corretto allora successivamente viene trasferito il suo valore nella copia originale.

AZIONI ATOMICHE MULTI-PROCESSO ----------

In alcune applicazioni può risultare convenite che le operazioni sugli oggetti di una azione atomica siano eseguite da più processi, e sono tipiche di sistemi distribuiti. L'elaborazione complessiva rimane atomica se gode delle due proprietà di **serializzabilità** e del **tutto o niente**.

Se ogni processo esegue il **two phase lock protocol** rilasciando gli oggetti su cui opera dopo la terminazione dell'azione è assicurata la proprietà di serializzabilità

Per soddisfare la proprietà del **tutto o niente** è necessario che tutti i processi completino le loro operazioni con lo stesso risultato o successo o abort.

Necessità di introdurre un nuovo stato: lo stato **ready** che caratterizza un processo quando ha completato la propria sequenza di azioni ed è pronto a terminare con successo ma deve attendere gli altri processi. Il passaggio da **working** a **ready** si ottiene tramite la primitiva **prepare**.

Two phase commit protocol -> il protocollo che viene eseguito da ciascun processo per negoziare il tipo di completamento cioè o successo o aborto. Anche in questo caso ci sono due fasi distinte:

- **FASE I** -> ciascun processo specifica il proprio completamento (successo o aborto)
- **FASE II** -> iene verificato il completamento degli altri processi: se tutti hanno terminato con successo tutti i processi passeranno allo stato **committing** altrimenti andranno tutti in **aborting**

Il **descrittore di azione** atomica in memoria stabile deve tenere aggiornato lo stato di tutti i processi partecipanti. La realizzazione dell'azione multi processo cambia a seconda del modello di iterazione tra processi: nel modello a **memoria comune** il descrittore di azione è un **monitor,** mentre nel modello a **scambio di messaggi** è un **oggetto privato** chiamato **coordinatore dell'azione atomica**.

AZIONI ATOMICHE MULTIPROCESSO - MODELLO A MEMORIA COMUNE ----------

L'**azione atomica** viene iniziata da un processo che ne crea in memoria stabile il descrittore e attiva in parallelo un certo numero di processi che la eseguono. Quando tutti i processi hanno completato le loro operazioni, con successo o terminando con aborto, il processo iniziale riprende il controllo e termina l'azione cancellando dalla memoria stabile il descrittore.

Schema del **funzionamento del processo i-esimo**:

1. Richiesta esclusiva degli oggetti
2. Creazione delle copie volatili degli oggetti
3. Sequenza di operazioni sulle copie volatili
4. Terminazione
   4.1 Con aborto -> entra nello stato **aborting** e vengono risvegliati i processi nello stato **ready**

4.2 Con successo -> crea la copia delle intenzioni in memoria stabile. Esegue la **prepare** per transitare nello stato di **ready**, e infine verifica se almeno un processo ha abortito.
    4.2.1 Caso affermativo -> distrugge la copia delle intenzioni e esegue **abort**
    4.2.2 Caso negativo -> tutti i processi sono nello stato di **ready** quindi il processo entra nello stato di **committing** e sovrascrive i dati originali con la copia delle intenzioni. Vengono poi svegliati tutti i processi in catena, uno dopo l'altro.

AZIONI ATOMICHE MULTIPROCESSO - MODELLO A SCAMBIO DI MESSAGGI ----------

Processo coordinatore -> gestisce il descrittore dell'azione atomica. Può essere uno dei processi che realizzano l'azione atomica. Crea il descrittore dell'azione atomica ed attiva in parallelo tutti i processi partecipanti.

Schema del protocollo two **phase commit protocol (lato coordinatore**):

1. Il processore coordinatore qualora sia disponibile a terminare con successo l'azione atomica, crea la copia delle intenzioni in memoria stabile ed esegue la **prepare** per passare allo stato **ready**. Invia poi ad ogni partecipante un messaggio di richiesta sito e rimane in attesa delle risposte.
2. Se arriva almeno una risposta con esito negativo l'azione deve essere abortita (passo 5). Se tutti i partecipanti rispondono con esito positivo l'azione deve terminare con successo (passo 3).
3. Viene eseguita la primitiva **commit** e viene inviato a ciascun partecipante un messaggio con l'indicazione di terminare con successo. Il coordinatore rimane in attesa del messaggio di **avvenuto completamento** da parte di tutti.
4. All'arrivo di tutti i messaggi di avvenuto completamento il coordinatore termina eliminando il descrittore dell'azione atomica.
5. In caso di almeno un esito negativo, vinee eseguita la primitiva **abort** e viene inviato a tutti un messaggio di **completamento con aborto**. Il coordinatore termina abortendo a sua volta

Algoritmo seguito da ogni partecipante:

1. Terminate le operazioni sugli oggetti, il processo può aver abortito (stato aborting) o aver creato la copia delle intenzioni (stato ready). In entrambi i casi attende il messaggio **richiesta-esito** da parte del coordinatore.
3. Risponde con **esito positivo** o **esito negativo** a seconda dello stato in cui si trova. Se è in stato aborting termina abortendo, altrimenti resta in attesa del messaggio di completamento da parte del coordinatore.
5. Se viene ricevuto il messaggio **completare con aborto**, viene eliminata dalla memoria stabile la copia delle intenzioni ed il processo termina abortendo.
7. Se viene ricevuto il messaggio **completare con successo**, il processo termina correttamente trasferendo i valori della copia delle intenzioni nella copia originale degli oggetti ed elimina la copia delle intenzioni. Invia al coordinatore il messaggio di **avvenuto completamento**.

AZIONI ATOMICHE MULTIPROCESSO E SISTEMI DISTRIBUITI ----------

Nei sistemi distribuiti possono verificarsi due tipi particolari di malfunzionamento:

- **Caduta dei singoli nodi della rete** -> ogni processo partecipante deve mantenere in memoria stabile delle variabili di stato da usare in fase di riattivazione.
- **Perdita dei messaggi in rete** -> utilizzo di un **temporizzatore** per limitare il tempo d attesa di un messaggio. Al termine del tempo il processo viene riattivato e viene eseguita una procedura per richiedere nuovamente l'informazione oppure abortire.

AZIONI ATOMICHE NIDIFICATE ----------

In base alle proprietà di serializzabilità e tutto o niente un'azione atomica rappresenta un'unità di programma che non può essere nidificata entro altre azioni atomiche. *Ad esempio A1 e A2 sono azioni atomiche con A2 nidificato dentro A1 (costituisce una della operazioni eseguite da A1)*. Ma questo schema presenta problemi che non ne consentono la realizzazione.

Quando A2 termina il processo che la esegue rilascia tutti gli oggetti su cui A ha aperto e quindi sono nuovamente disponibili per essere acquisiti da altri processo. Si ha quindi **violazione serializzabilità** e violazione della proprietà **tutto o niente**.

Possibile soluzione è quella di modificare i due protocolli **two phase lock** e **two phase commit protocol**.

CHIAMATA DI PROCEDURA REMOTA IN SISTEMI DISTRIBUITI ----------

Le azioni atomiche sono particolarmente utili per strutturare **sistemi transazionali**. Il meccanismo RPC è idoneo per le comunicazioni tra processo transazionale e i processi che gestiscono la base di dati secondo un modello Client/Server.

La realizzazione del meccanismo RPC in un sistema distribuito comporta una serie di problemi legati ai malfunzionamenti proprio dei sistemi distribuiti: come la caduta dei singoli nodi e i guasti alla sottorete di comunicazione. Occorre quindi utilizzare **meccanismi di temporizzazione** nella RPC (time-out).

È possibile utilizzare una delle due possibili semantiche :

- **At most once** -> in presenza di malfunzionamenti durante PC questa viene abortita senza produrre effetti. Per fare ciò la RPC viene gestita come se fosse un'azione atomica nidificata all'interno di un'azione di alto livello che rappresenta l'azione eseguita dal processo cliente. I due processi cliente e servitore devono eseguire il **two phase commit protocol** al termine dell'intera operazione del processo cliente
- **At least once** -> se il processo è interrotto dal meccanismo di temporizzazione prima della ricezione dei risultati si invia di nuovo il messaggio di richiesta. Occorre che le procedure siano **idempotenti** questo perché c'è il rischio che vengano eseguite più volte le stesse procedure.

REALIZZAZIONE MEMORIA STABILE ----------

La memoria stabile è un tipo di memoria che gode delle seguenti proprietà:

- **non è soggetta a malfunzionamenti** -> ciò si ottiene usando tecniche di ridondanza come ad esempio le tecnologie RAID
- le informazioni in essa residenti **non vengono perdute o alterate a causa della caduta dell'elaboratore** -> le operazioni di lettura e scrittura siano operazioni atomiche. Vengono quindi utilizzate le operazioni **Stable read** e **Stable write**.

I tipi di malfunzionamento possono essere tre:

- **errore in lettura** -> dovuti a disturbi temporanei. Possono essere eliminati rileggendo le informazioni desiderate in modo da poterle controllare.
- **errore in scrittura** -> dovuti a disturbi temporanei. Possono essere rilevati rileggendo le informazioni scritte e confrontandole con quelle originali.
- **alterazioni delle informazioni** -> dovuti a guasti HW. Per la soluzione di questo problema si ricorre alle **copie multiple** cioè la duplicazione di tutte le informazioni su due dischi distinti.

Blocco stabile -> un blocco stabile è costituito da una coppia di blocchi uno per ogni disco permanente.

La memoria stabile permette di avere le proprietà di **serializzabilità**, grazie al fatto che viene realizzata come un **monitor** e questo garantisce l'indivisibilità delle operazioni, e la proprietà del **tutto o niente** grazie al fatto che **Stable read** e **Stable write** godono di questa proprietà.

Stable read -> gode della proprietà del tutto o niente perché non effettua alcuna modifica

Stable write -> se si verifica un guasto durante le operazioni di scrittura sul primo blocco, questo viene alterato ma rimane intatto il secondo, se avviene sul secondo quello, allora questo viene alterato ma il primo rimane valido.

Transazione -> sequenza di operazioni effettuate su database che fanno passare il sistema da uno stato all'altro, e tutti e due devono essere consistenti. Devono essere rispettate le **proprietà ACID** cioè Atomicità, Consistenza, Isolamento e Durata.