Appunti di sistemi operativi M

A cura di Marco Moschettini

Protezione e sicurezza

Modelli di protezione

- **OGGETTI**: Passivi, Risorse
- **SOGGETTI**: Processi —> Ad ogni soggetto è associato un dominio.
- **DIRITTI D'ACCESSO**: Protezioni

Politiche

- **DAC [Discretional Access Control]**
  Chi crea ha il diritto di accesso (Decentralizzato)
- **MAC [Mandatory Access Control]**
  Diritti centralizzati, Alta sicurezza.
- **RABC [Role Based Access Control]**
  Ruoli!

Principio del privilegio minimo

Ti è garantito accesso SOLO a quello che ti serve per vivere.
Distinzione tra POLITICA (cosa va fatto) e MECCANISMI (come va fatto)

Es (Unix):
Meccanismo con 3 bit (read, write, execute) per la protezione.

Dominio di Protezione

Definizione: "Un dominio definisce un insieme di coppie, ognuna contenente l'identificatore di un oggetto e l'insieme delle operazione che il soggetto associato al dominio può eseguire su ciascuno oggetto (diritti d'accesso)"

D(S) = {<oggetto, diritti>} —> D1 ={ <File1, (read, write)>, <File2, (execute)>}

Associazione tra processo e dominio

- **STATICA:** Insieme di risorse disponibili ad un processo fisse nel tempo.
  - non adatta se si vuole limitare le risorse ad un processo (Privilegio minimo)
  - non è detto che si posseggano tutte le informazioni sulla vita di un processo prima che parta

  - l'insieme delle risorse necessarie ad un progetto cambia durante la sua vita.
- **DINAMICA:** Associazione tra processo e dominio cambia durante l'esecuzione del processo.
  - cambiando dinamicamente dominio si può mettere in pratica il principio del privilegio minimo. (Occorre un *meccanismo* per passare da un dominio all'altro)
  - Esempio 1: **dual mode** (dominio user e dominio kernel) —> cambio di dominio associato alle **system calls**. (Protezione solo tra kernel ed utente, non tra più utenti)
  - Esempio 2: Dominio associato all'utente. Cambio di **UID =** cambio di dominio —> Se un utente A inizia l'esecuzione di un file P il cui proprietario è B e il file ha dettato UID = on, allora al processo che esegue P viene assegnato lo user-id di B.

Matrice degli accessi

- Ogni riga è un soggetto (es. utente) —> riga = dominio
- Ogni colonna è associata ad un soggetto.

Matrice degli accessi

	Oggetto 1	**Oggetto 2**	**Oggetto 3**
Soggetto 1	*read, write*	*execute*	*write*
Soggetto 2	*execute*	*read, write*	*execute*

Il modello specifica lo *stato di protezione*.
Tramite questo modello si può:
- **verificare** se un azione è **lecita**
- **modificare dinamicamente** *gli oggetti e i soggetti*
- **Cambiare dominio** ad un processo

Modifica dello stato di protezione

- **DAC —> utenti**
- **MAC —> unità centrale**

La modifica dello stato di protezione si può ottenere con 8 primitive (**Graham e Denning)**
- *create, delete* ***object***
- *create, delete* ***subject***
- *read, grant, delete, transfer* ***access right***

La possibilità di copiare un diritto di accesso è specificato tramite un * nella matrice (**copy flag**)

Matrice degli accessi, copy flag

	Oggetto 1	**Oggetto 2**	**Oggetto 3**
Soggetto 1	*read*, write*	*execute*	*write*
Soggetto 2	*execute*	*read, write**	*execute*

La propagazione può avvenire per:
- *trasferimento*
- *copia*

Diritti fondamentali

1. **Diritto owner:** assegnazione di un qualunque diritto di accesso su un oggetto X ad un qualunque soggetto S_j da parte di un soggetto S_i.

Matrice degli accessi, diritto owner

	Oggetto 1	**Oggetto 2**	**Oggetto 3**
Soggetto 1	*read*, write*	*execute*	*write*
Soggetto 2	*execute,* ***owner***	*read, write**	*execute*

Il soggetto 2 può trasferire tutti i diritti che ha sull'oggetto 1 al soggetto 1

2. **Diritto control:** eliminazione di un diritto d'accesso per un oggetto X nel dominio S_j da parte di S_i

Matrice degli accessi, diritto control

	Oggetto 1	**Oggetto 2**	**Oggetto 3**	**Soggetto 1**	**Soggetto 2**
Soggetto 1	*read*, write*	*execute*	*write*		***control***
Soggetto 2	*execute*	*read, write**	*execute*		

Soggetto 1 può revocare a Soggetto 2 il diritto write su O2

3. **Diritto switch**: un processo che esegue nel dominio del soggetto S_i può commutare al dominio di un altro soggetto S_j

Matrice degli accessi, diritto switch

	Oggetto 1	**Oggetto 2**	**Oggetto 3**	**Soggetto 1**	**Soggetto 2**
Soggetto 1	*read*, write*	*execute*	*write*		
Soggetto 2	*execute*	*read, write**	*execute*	***switch***	

Un processo che esegue nel dominio di Soggetto 2 può trasferirsi al dominio di Soggetto 1

Graham e Denning dimostrano che le regole precedenti danno luogo ad un sistema che può risolvere problemi come:
- propagazione controllata dei diritti di accesso (*confinement)*
- prevenire la modifica dei diritti di accesso (*sharing parameters)*
- uso non corretto dei diritti di accesso di un processo da parte di un altro (*trojan horses)*.

Realizzazione della matrice degli accessi

- **Access Contro List (ACL):**
  - Memorizzazione per **colonne**
  - Ogni oggetto è associato ad una lista di soggetti che vi possono accedere.
    Oggetto —> {Soggetti, Diritti}
- ***Capability List:***
  - Memorizzazione per **righe**
  - Ogni soggetto è associato ad una lista di oggetti che può accedere.
    Soggetto —> {Oggetto, Diritti}

ACL - Access Control List

- ACL definita per singoli utenti.
- Possibilità di operare su *gruppi di utenti*
  - in questo caso è necessario specificare anche il gruppo di appartenenza
    UID_1, GID_1 : <diritti>
  - lo stesso utente può appartenere a più gruppi —> **ruoli**
  - possibilità di specificare diritti non legati al gruppo
    *UID_1, * : <diritti>*

Capability List

Quando un processo vuole eseguire un'operazione su un oggetto O_i il sistema controlla che nella lista sia presente un riferimento esistente per il Soggetto in corso

Le liste devono essere protette da **manomissioni**. Ciò si può ottenere:
- La capability list viene gestita solo dal OS e l'utente fa riferimento alla risorsa con un riferimento alla sua posizione nella lista (simile al file descriptor in UNIX)
- Architettura etichettata. Soluzione applicata a livello hardware. (Es IBM)

Capability List

	F1	F2	F3	F4	F5
Soggetto S	-	-	R	RWE	W

Lista di tutti gli oggetti a cui può accedere un singolo soggetto.

Revoca dei diritti di accesso

La revoca dei diritti di accesso per un oggetto può essere:
- **Generale o selettiva**: valere per tutti gli utenti che hanno quel diritto o solo per un gruppo
- **Parziale o totale**: riguardare solo un sottoinsieme di diritti o tutti
- **Temporanea o permanente**: diritto non più disponibile oppure può essere riottenuto

Revoca per un oggetto e ACL

Si fa riferimento ad una ACL associata ad un oggetto e si cancellano i diritti d'accesso che si vogliono revocare.

Revoca per un oggetto e Capability

Operazione più complessa. Bisogna verificare per **ogni** dominio se contiene capability con riferimento all'oggetto indicato.

ACL vs CL

- ACL: informazione sui diritti di un soggetto sparsa nelle varie ACL
- CL: informazione sui diritti su un oggetto sparsa nelle varie CL

Soluzione mista

- ACL memorizzata su supporto permanente (disco). Se un soggetto tenta di accedere ad un oggetto per la prima volta:
  - Si analizza l'ACL
  - Se esiste una antri con il nome del soggetto e tra i diritti di accesso c'è quello richiesto, viene fornita la CL in memoria volatile
  - In questo modo posso accedere più volte all'oggetto senza dover ogni volta interrogare la ACL
  - Dopo l'ultimo accesso la capability viene distrutta.

Esempio (Unix) - Apertura di un file.
fd = open(<nome_file>, <diritti>)
- Si cerca il file nella directory e si verifica se l'accesso è consentito
- In caso affermativo viene creata una entry nella tabella dei file aperti con fd e diritti (CL)
- viene ritornato fd al processo (riferimento al file aperto)
- tutte le successive operazioni vengono fatte su fd (CL)

Sicurezza multilivello

In alcune situazioni la politica DAC (ognuno può leggere e scrivere i propri files) può non bastare (es Militare). In questo caso si utilizza una politica MAC (controllo degli accessi obbligatorio)

Modelli

I modelli di sicurezza multilivello più usati sono 2. In entrambi i modelli i soggetti (utenti) e oggetti (files) sono classificati in livelli (classi di accesso)

1. Livelli per i soggetti (**clearence levels)**
2. Livelli per gli oggetti **(sensitivity levels)**

I modelli sono 2:

- **Bell-La Padula**
  - **obiettivo: confidenzialità**
  - usato in ambito militare.
  - garantisce la confidenzialità
  - associa ad un modello DAC, 2 regole di sicurezza MAC
  - *4 livelli di sensibilità dei documenti*
    - Non classificato
    - Confidenziale
    - Segreto
    - Top secret
  - *4 livelli per i soggetti*
  - Regole di sicurezza: stabiliscono come le informazioni possono circolare:
    - **Proprietà di semplice sicurezza:** un processo in esecuzione ad un livello di sicurezza k può leggere solo oggetti del suo livello o livelli inferiori.
    - **Proprietà *:** un processo in esecuzione ad un livello k può scrivere solo oggetti al suo livello o superiori.
  - I processi possono scrivere verso l'alto e leggere verso il basso… ma non il contrario.
  - *Modello concepito per mantenere i segreti… non per garantire l'integrità dei dati. Infatti è possibile **sovrascrivere** i dati di un livello superiore.*
  - Esempio: difesa da un cavallo di troia.
- **Biba:**
  - **obiettivo: integrità dei dati**
  - Regole di sicurezza: stabiliscono come le informazioni possono circolare:
    - **Proprietà di semplice sicurezza:** un processo in esecuzione ad un livello k può scrivere solo al suo livello o inferiore. (non verso l'alto)
    - **Proprietà di integrità:** un processo può leggere solo verso l'alto.

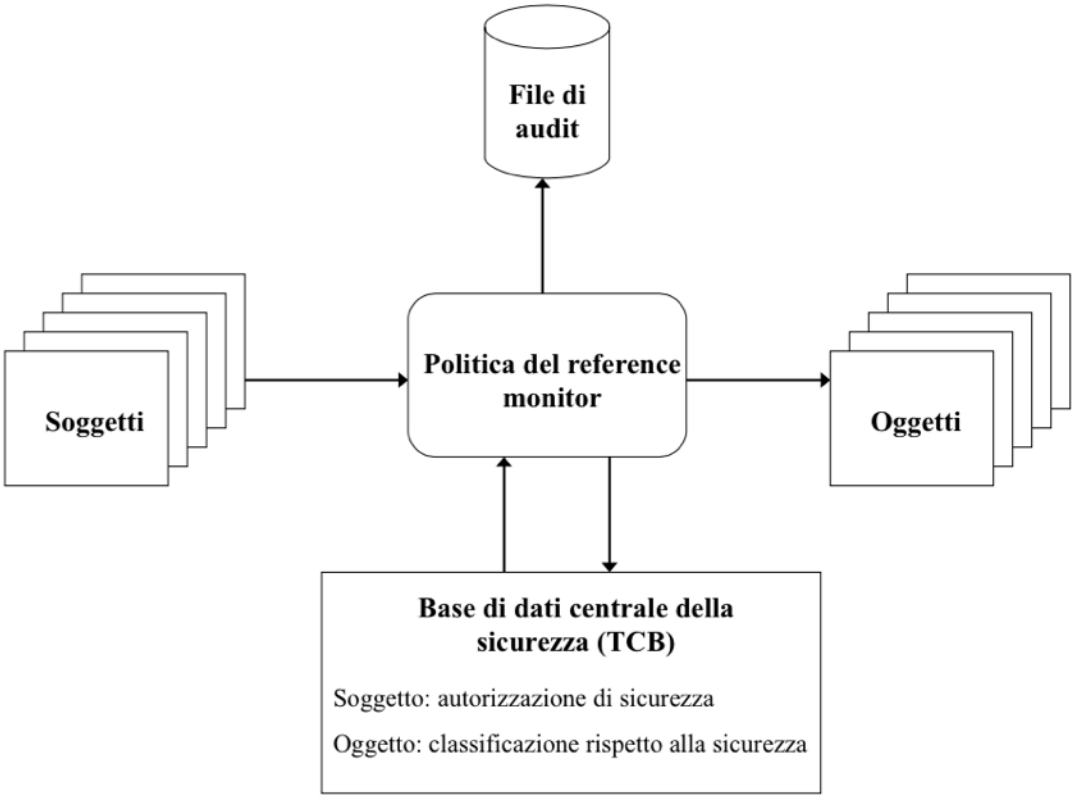
I due modelli sono in conflitto tra loro e **NON** si possono usare contemporaneamente.

Reference Monitor

RM è la realizzazione del modello B-LP. Quindi impone no read-up, no write-down) ed ha le seguenti proprietà:

- **Mediazione completa:** le regole di sicurezza sono applicate ad ogni accesso. (Non solo all'apertura di un file).

- **Isolamento:** Il monitor è protetto (a livello hardware) rispetto alla modifiche non autorizzate.

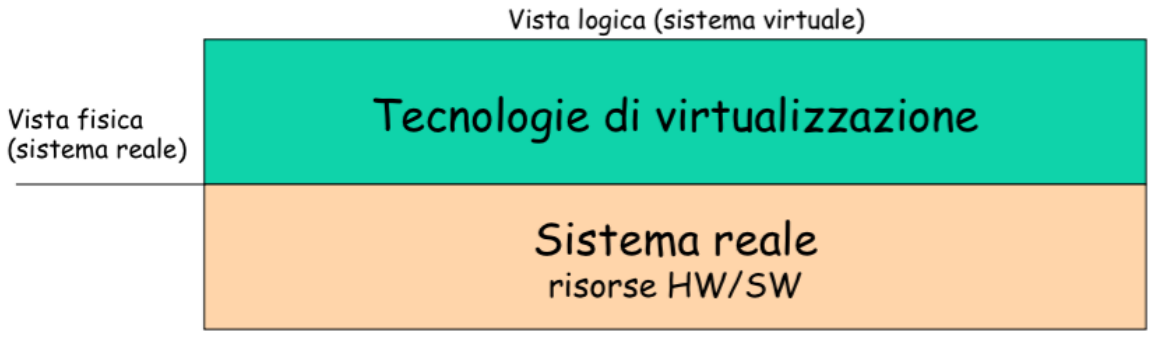


audit file: contiene eventi importanti per la sicurezza (violazioni ecc…)

Virtualizzazione

Definizione

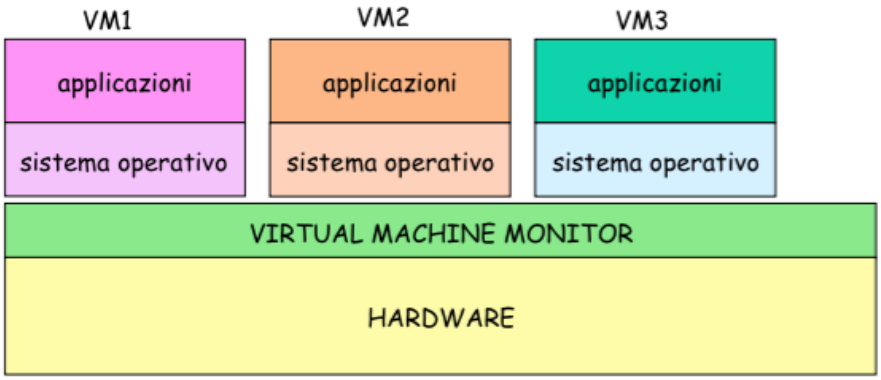
Dato un sistema dotato di Hardware e Software, virtualizzare il sistema significa presentare all'utilizzatore nua visione delle risorse del sistema diversa da quella reale. Ciò si ottiene introducendo un livello di **indirezione** tra la vista logica e quella fisica delle risorse



Tecnologie di virtualizzazione

Obiettivo: disaccoppiare il comportamento delle risorse hardware da quelle software.

Virtualizzazione di sistema



Introduzione del **Virtual Machine Monitor (VMM)** che si occupa di creare un interfaccia tra una singola piattaforma hardware e software differenti tra di loro.

VMM funge da unico mediatore tra hardware e software e garantisce:
- Isolamento tra le VM
- Stabilità del sistema

Emulazione

Esecuzione di programmi compilati per un certo **insieme di istruzioni** su un sistema diverso. Vengono emulate interamente le singole istruzioni dell'architettura host.
- **Vantaggi:** interoperabilità tra ambienti eterogenei
- **Svantaggi:** perdita di performance

Dal punto di vista dell'implementazione ha visto 2 strade.
- **interpretazione:** Grande flessibilità (legge ogni singola istruzione) ma scarsa performance.
- **compilazione dinamica:** legge il codice **a blocchi** ottimizzandoli e riutilizzando le parti usate spesso. Tutti i più noti emulatori (Virtual PC, Power PC, QEMU) usano questa tecnica.

Virtual PC

Virtualizzazione che permette ad un computer con sistema operativo Mac OSX o Windows l'esecuzione di sistemi operativi diversi. Ricrea quasi interamente un ambiente di natura Intel. Destinato all'utilizzo di vecchi applicazioni.

La successiva introduzione di processori Intel nei computer Apple, ha, di fatto, reso inutile Virtual PC

QEMU

Software 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. Molto veloce

Vantaggi della virtualizzazione

- Uso di più OS sulla stessa macchina fisica
- Ogni macchina virtuale offre un ambiente di esecuzione separato (sandbox)
  - Possibilità di eseguire test
  - Sicurezza: eventuali attacchi sono confinati alla singola VM
- Consolidamento HW: Server farms! Più server sullo stesso hardware
- Gestione facilitata delle macchine
  - Facile creare nuove macchine virtuali
  - Facile migrare macchine virtuali
- Vantaggi didattici: non far distruggere i computer di laboratorio dagli studenti

Realizzazione del VMM

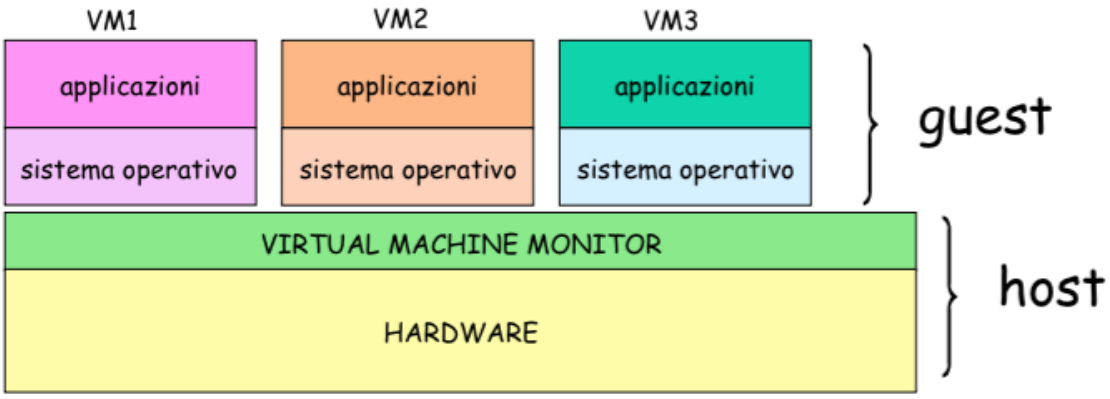
Risorse da offrire:
- CPU
- Memoria
- Dispositivi I/O

Requisiti e parametri

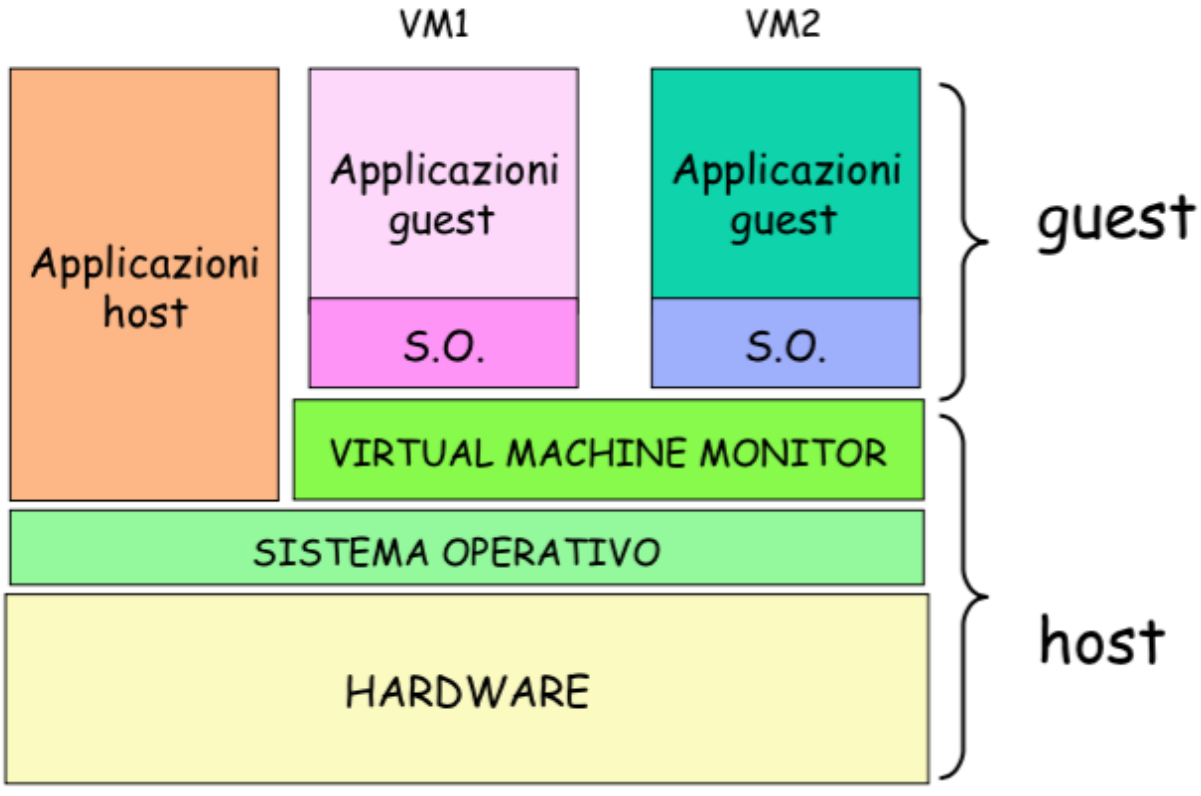
Popek e Goldberg affermano i seguenti **requisiti**:
1. *Ambiente di esecuzione identico alla macchina reale*
2. *Elevata efficienza nell'esecuzione dei programmi*
3. *Garantire la stabilità e la sicurezza del sistema*

Parametri:
- **Livello:**
  - **VMM di sistema:** si appoggia direttamente sopra l'hardware della macchina (Xen)
  - **VMM ospitati:** eseguiti come applicazioni sopra ad un OS esistente (Parallels, ecc…)
- **Modalità di dialogo:**
  - **virtualizzazione pura:** (Es. VMWare) le macchine virtuali usano la stessa interfaccia (istruzioni macchina) dell'hardware
  - **paravirtualizzazione:** (Es: Xen) il VMM ha un'interfaccia diversa da quella dell'architettura hardware

VMM di sistema vs. VMM ospitati



VMM di Sistema



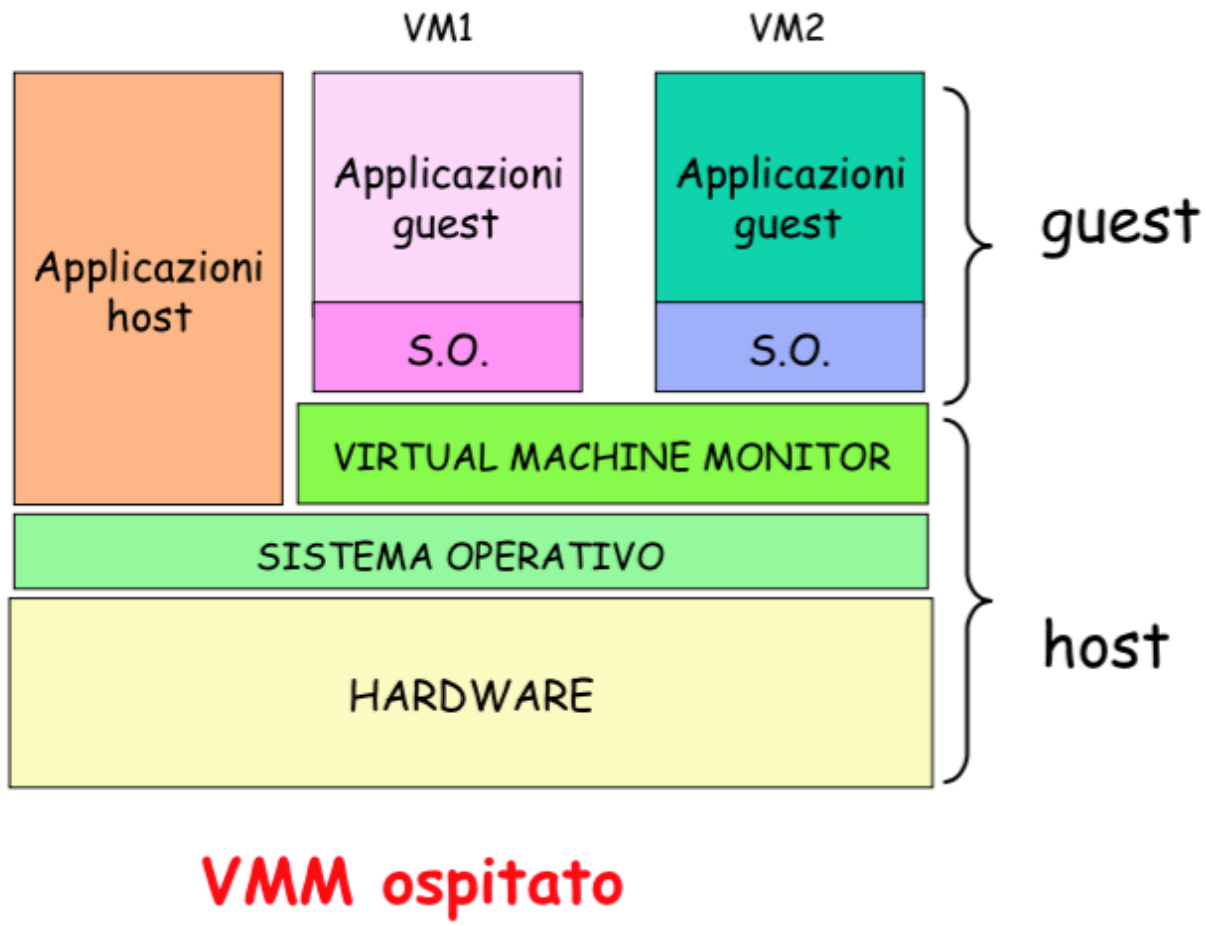
VMM ospitato

VMM di sistema: realizzazione

L'architettura della CPU prevede 2 livelli di protezione (**ring**):
- *Supervisore*
- *Utente*

SOLO il VMM opera nello stato supervisore, mentre l'OS, e le applicazioni operano a livello utente

Problemi:
- **ring deprivileging:** l'OS della macchina esegue in uno stato che non è proprio (*system calls*)
- **ring compression:** se i ring utilizzati sono 2, applicazioni e OS della macchina virtuale eseguono allo stesso livello, è necessario proteggere lo spazio tra OS e applicazioni



VMM ospitato

Ring deprivileging

Le istruzioni privilegiate non possono essere eseguite (richiederebbero lo stato supervisore)
Soluzioni?

- Se il guest tenta di eseguire un'operazione privilegiata, la CPU **notifica un'eccezione alla VMM** e gli trasferisce il controllo (*trap*)**.** A questo punto il **VMM controlla la correttezza dell'operazione richiesta** e ne **emula** il comportamento
- le istruzioni possono essere eseguite direttamente sulla CPU (esecuzione diretta)

Supporto hardware alla virtualizzazione

Un'architettura CPU si dice ***naturalmente virtualizzabile*** se prevede l'invio di notifica allo stato supervisore per ogni istruzione privilegiata eseguita da un diverso livello di protezione.
In questo caso la realizzazione del VMM è **semplificata:** per ogni trap viene eseguita una routine di emulazione (*trap-and-emulate)*
Non tutte le architetture su naturalmente virtualizzabili —> comportamenti indesiderati o crash del sistema

Problema del **Ring Aliasing**
Alcune istruzioni privilegiate possono modificare dei registri dedicati alla VMM —> **instabilità** (Es. registro CS che contiene il livello di privilegio corrente)

Se il processore non supporta nativamente la virtualizzazione è necessario ricorrere a soluzioni software.

Soluzioni:
- **Fast Binary Translation**
- **Paravirtualizzazione**

Fast Binary Translation

Il VMM scansiona **dinamicamente** il codice degli OS guest prima dell'esecuzione —> **sostituirà a run time i blocchi che contengono istruzioni privilegiate** in blocchi equivalenti a chiamate al VMM

- **PRO: macchina virtuale è un esatta replica della macchina fisica**
- **CONTRO: traduzione dinamica è costosa**

Paravirtualizzazione

Il VMM offre al sistema operativo guest una interfaccia virtuale (hypercall API) con:
- istruzioni privilegiate verso VMM che le tradurrà al livello sottostante
- i kernerl degli OS **devono essere modificati** per aver accesso all'interfaccia
- VMM semplificata—> no traduzione dinamica delle system call

- **PRO: prestazioni migliori rispetto a Fast Binary**
- **CONTRO: necessità di modificare gli OS host**

Protezione nell'architettura x86

Fino all'80186 x86 non aveva protezione —> sistema guest poteva allocare memoria per conto suo —> possibili problemi.

Viene introdotto il concetto di protezione —> distinzione tra sistema operativo e applicazioni (possono accedere alla memoria SOLO attraverso il sistema operativo) attraverso il concetto di **ring di protezione**

Registro CS
I due bit meno significativi vengono riservati per rappresentare il **corrente livello di privilegio** (cpl):

- **Ring 0:** maggiori privilegi —> Sistema operativo
- Ring …
- **Ring 3:** minori privilegi —> Applicazioni

NON è permesso a ring diversi dallo 0 di accedere alla CPU

x86 benchè possieda 4 ring di protezione, ne usa solo 2 nella pratica. Gli altri (1 e 2) che sono vie di mezzo erano stati pensati per altre parti del sistema operativo ma non sono stati utilizzati quasi mai.

VMM in x86

- **Ring deprivileging:** viene dedicato il ring 0 al VMM e il sistema operativo viene messo su ring con privilegi ridotti. Due casi:
  - VMM su ring 0, OS su ring 1, applicazioni su ring 3 (**0/1/3**). Tuttavia NON compatibile con 64 bit. Largamente più usata! L'OS può generare eccezioni che vengono catturate dal VMM
  - VMM su ring 0, OS e applicazioni su ring 3 (**0/3/3**). Non potendo generare eccezioni, l'OS deve ricorrere a metodi molto sofisticati per notificare il VMM. —> si avvicina molto all'**emulazione**

Possibili problemi del VMM

Ring Aliasing

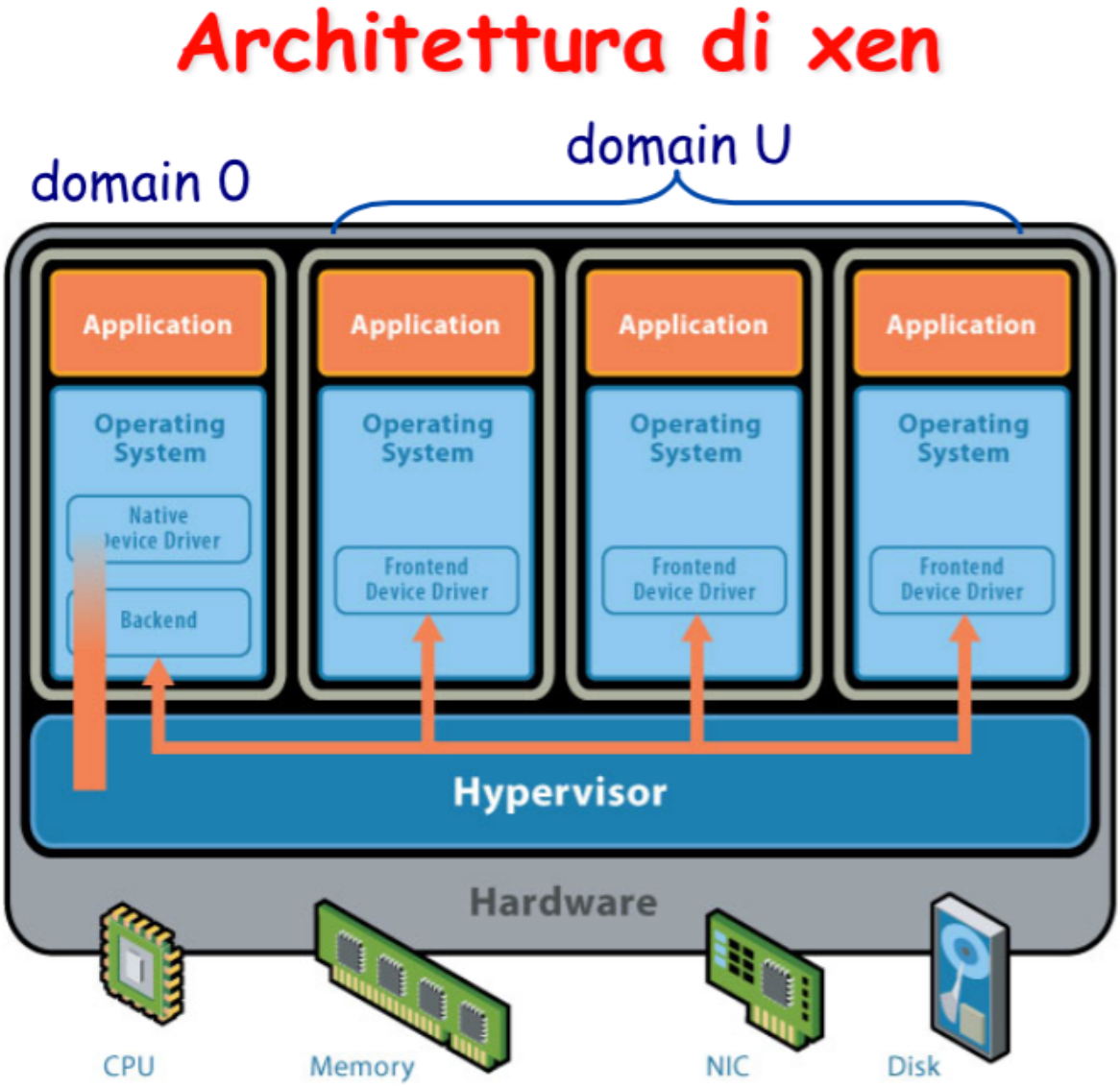
Alcune istruzioni non privilegiate possono modificare registri che non dovrebbero vedere.
Es x86: A qualsiasi livello è possibile effettuare la chiamata **PUSH** che permette di salvare il contenuto del registro CS nello stack. In questo modo è possibile per il sistema operativo guest, sapere su che livello di ring si trova (non lo 0).

Mancate eccezioni

In x86 esistono alcune istruzioni privilegiate che **non lanciano eccezioni** se eseguite a ring > 0. Questo rende impossibile l'intervento trasparente del VMM.

Xen

VMM open source che opera in **paravirtualizzazione**

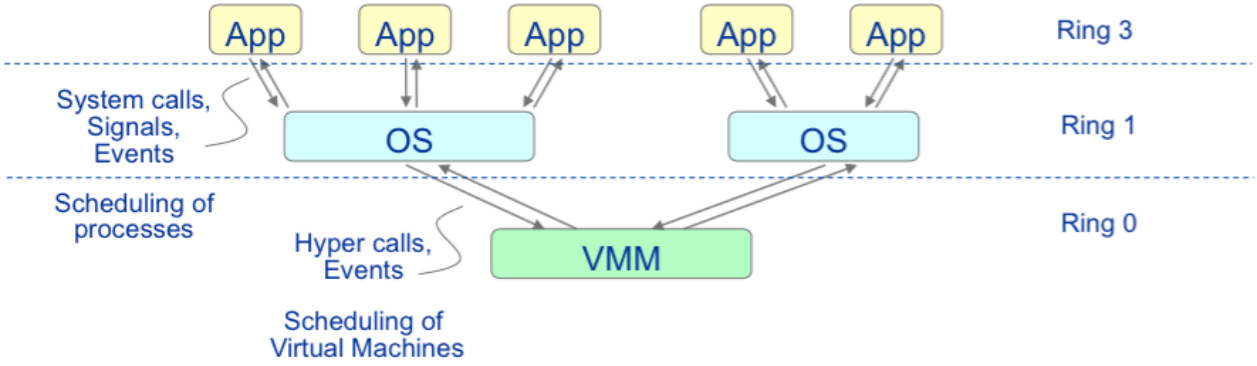


Organizzazione di xen

- VMM (hypervisor) si occupa della virtualizzazione della CPU
- Xen dispone di una **interfaccia di controllo** accessibile solo dal Domain 0 che suddivide le risorse tra i veri sistemi guest.
- Nel dominio 0 viene eseguita l'applicazione che controlla tutta la piattaforma

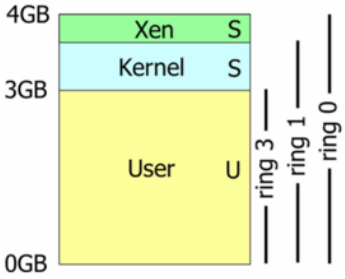
Caratteristiche

- **Paravirtualizzazione:** Sistemi guest eseguono direttamente istruzioni privilegiate, il VMM si occupa di eseguire realmente queste istruzioni
- **Protezione x86:** I sistemi sono collocati al ring 1, il VMM al ring 0 e le applicazioni al ring 3 **(0/1/3)**



Gestione della memoria e paginazione

- **Gestione della memoria:**
  - Memoria virtuale gestita con **meccanismi di paginazione normali**.
  - I guest OS si occupano della paginazione ma delegano la scrittura delle page table al VMM
  - **Page faults** gestiti direttamente al livello hardware (TLB)
- **Soluzione adottata:** tabelle delle pagine delle VM:
  - mappate nella memoria fisica della VMM
  - non possono essere accedute in scrittura dai kernel guest ma sono dalla VMM
  - sono accessibili in modalità read-only anche dai guest
- **Memory split**
  - Xen risiede nei primi 64MB del virtual address space per una maggiore efficienza
  - Lo spazio di indirizzamento virtuale per ogni VM è fatto in modo da contenere Xen e il kernel in segmenti separati



Creazione di un processo

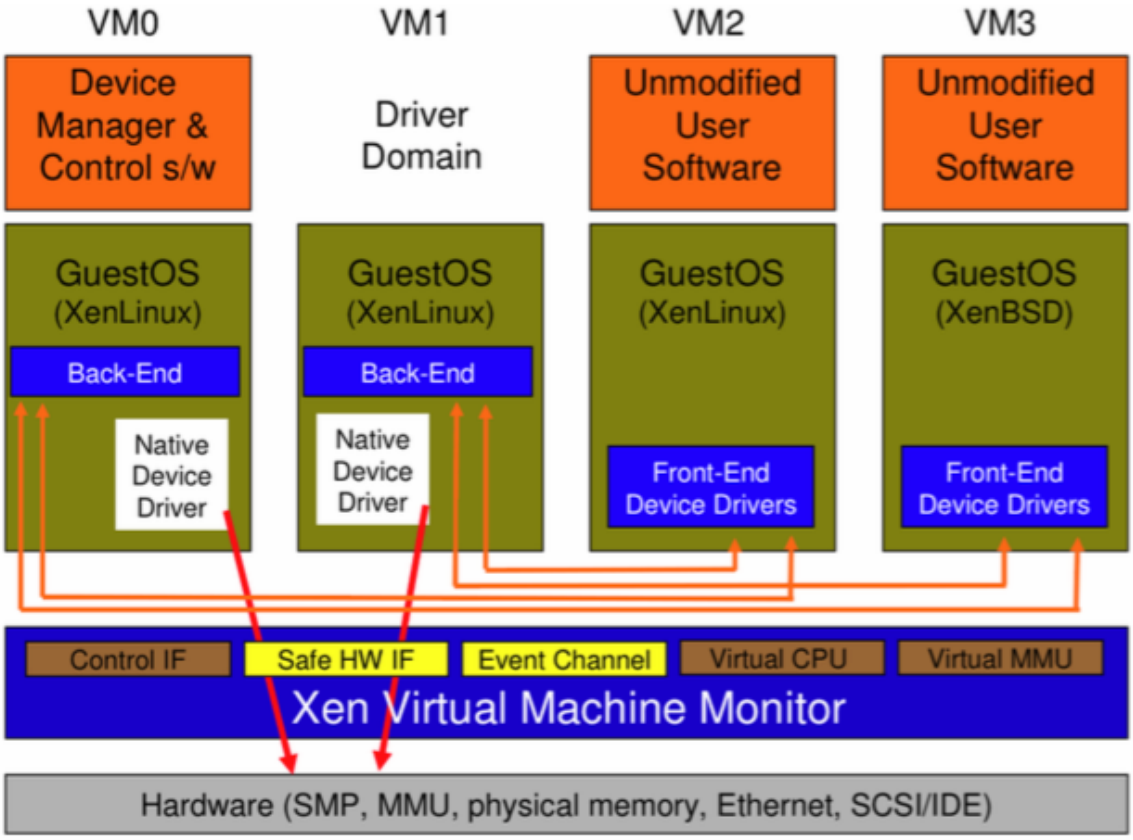
- Guest OS richiede una nuova tabella delle pagine al VMM:
  - Alla tabella vengono aggiunte le pagine relative al segmento Xen
  - **Xen registra la nuova tabella e ne acquisisce il diritto di scrittura esclusiva**
  - Ogni successivo update da parte del guest provocherà un **protection-fault** che verrà gestita dal VMM.
- Gestione della memoria comune ai sistemi guest attraverso un **balloon process:**
  - Su ogni macchina virtuale è in esecuzione un processo (**balloon process**) che *comunica con il VMM*

- In caso di necessità di nuove pagine il VMM chiede ad altre VM di liberare memoria
  - Chiede al balloon process di *gonfiarsi*
  - **La VM guest alloca nuove pagine al balloon process che le passerà in seguito al VMM**

Virtualizzazione della CPU

- Architettura CPU simile a quella fisica con paravirtualizzazione —> hypercalls
- VMM si occupa dello scheduling delle VM con il *Borrowed Virtual Time scheduling algorithm* (algoritmo general purpose che permette schedulazioni efficienti
- Possiede **due clock** (sincronizzati tramite :
  - Real-time (sync con il processore)
  - Virtual-time (associato alla VM)

Virtualizzazione dell'I/O



- **Back-end driver**: per ogni dispositivo, il suo driver è isolato in **Dom 0** (tipicamente) —> Accesso diretto all'HW
- **Front-end driver:** driver semplificato che parla con il back-end driver.

PRO e CONTRO

- **PRO:** portabilità, isolamento VM e VMM
- **CONTRO:** comunicazione asincrona con back-end driver —> produttore e consumatore!

Gestione interruzioni e eccezioni

Il vettore delle interruzioni punta direttamente alla routine del kernel task, **tranne page fault.** Questa non può essere delegata completamente al guest perché **richiede l'accesso al registro CR2** (Contiene l'indirizzo che ha generato il page fault) —> CR2 accessibile solo dal ring 0 —> VMM implicata.
Quindi:

- Handler del vettore punta a xen (ring 0)
- Xen copia il contenuto di CR2 in uno spazio del guest
- Jump al guest
- Gestione dell'eccezione

Gestione VM

Compito del VMM è la gestione delle macchine virtuali

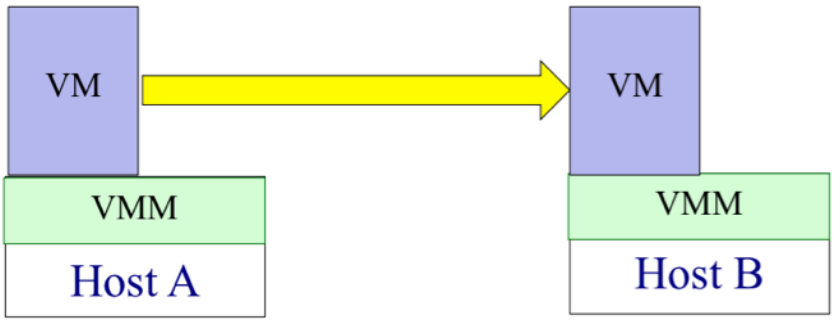
- Creazione
- Spegnimento
- Eliminazione
- Migrazione

Stati di una VM

- **Running** (attiva): macchina accesa e ram occupata
- **Inactive**(powered off): macchina spenta e rappresentata nel filesystem con un file di immagine
- **Paused**: macchina virtuale in attesa di un evento
- **Suspended**: macchina sospesa dal VMM. **Stato e risorse** salvate su *file immagine* (Necessario un resume per ripartire)

Migrazione di VM

Le VM possono essere migrate senza essere spente



Soluzione adottata —> Precopy (6 passi):
- **Pre-migrazione:** Individuazione della VM da migrare e dell'host di destinazione (host B)
- **Reservation:** Viene inizializzata una VM container nello spazio di destinazione
- **Pre-copia:**
  - Vengono copiate tutte le pagine della VM da migrare sull'host A
  - Vengono copiate tutte le pagine modificate (dirty pages) da A a B fino a quando i numero di dirty pages è inferiore ad una soglia
- **Sospensione:** viene sospesa la VM e vengono copiate le dirty pages e lo stato da A a B
- **Commit:** viene cancellata la VM da A
- **Resume:** viene attivata la VM su B

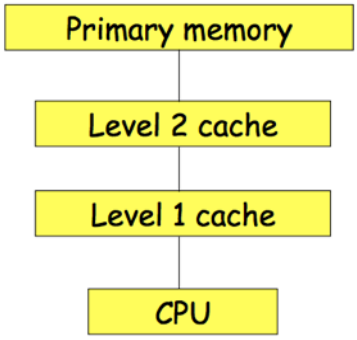
Soluzione alternativa —> Postcopy
- macchina viene sospesa e viene copiato pagine e stato —> tempo di migrazione più breve ma downtime **molto più elevato**

In Xen migrazione basata su pre-copy (pagine compresse per ridurre l'occupazione di banda), con meccanismo **guest-based** (comando di migrazione eseguito da un demone nel **domain 0** nel server A)

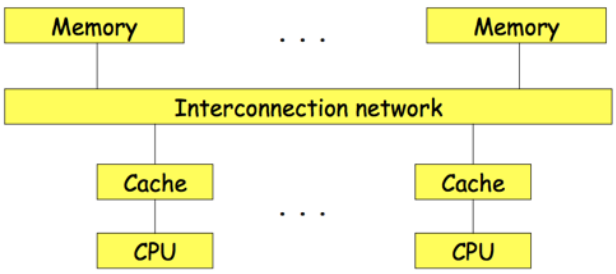
Programmazione concorrente

Tipi di architetture

- ***Single processor***

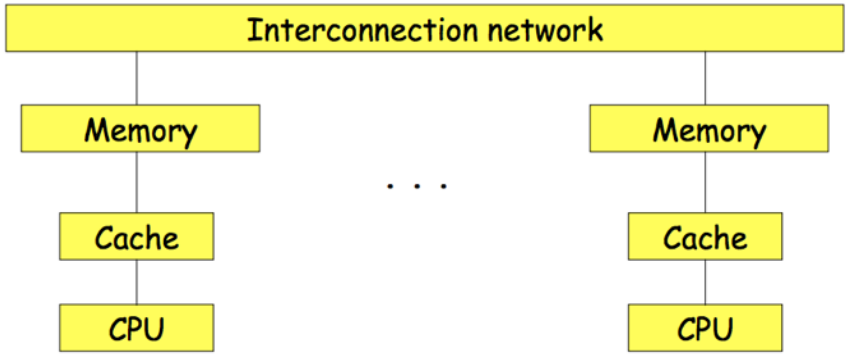


- ***Shared - Memory multiprocessors***



- In sistemi con un numero **ridotto di processori** (*da 2 a 30 circa*)**:**
  - rete di interconnessione realizzata da un memory bus o da crossbar switch
  - **UMA** (*Uniform Memory Access*): tempo di accesso uniforme da ogni processore ad ogni locazione di memoria
- In sistemi con un numero **elevato di processori** (*decine o centinaia*)
  - memoria organizzata in maniera **gerarchica** —> per evitare congestione del bus
  - rete di interconnessione è un insieme di switches e memorie strutturato ad albero.
  - **NUMA** (*Non Uniform Access Time*)

- ***Distributed - Memory Multicomputers and networks***



- **Classificazione*:***
  - **Multicomputer:** Processori e rete *fisicamente* vicini
  - **Network systems:** Nodi sono collegati da una rete locale (Ethernet) o da una geografica (Internet)

Tipi di applicazioni

- **Multithreaded:** Applicazioni strutturate come un insieme di processi. Processi schedulati ed eseguiti indipendentemente
- **Sistemi distribuiti/multitasking:** le componenti dell'applicazione (Task) vengono eseguite su nodi (anche virtuali) collegati tramite canali di comunicazione. I processi comunicano scambiandosi messaggi (Es. Client-Server). Il singolo nodo è spesso *Multithreaded*.
- **Applicazioni parallele:** Obiettivo: risolvere un problema velocemente —> processori paralleli con algoritmi paralleli

Processi non sequenziali

Algoritmo, programma, processo

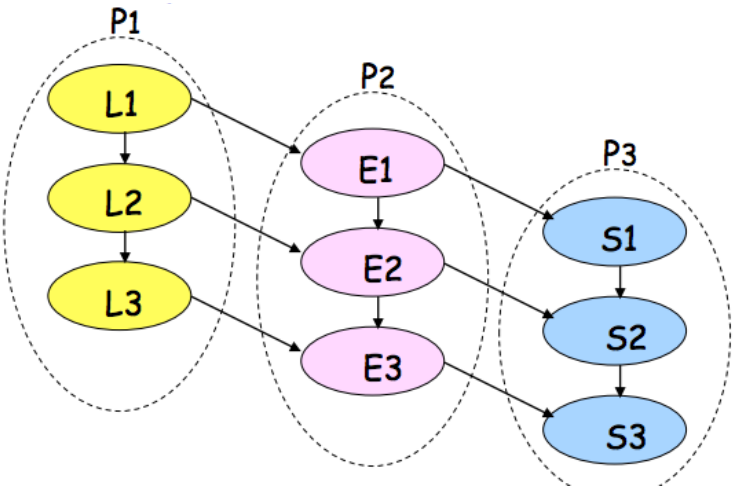
- **Algoritmo:** procedimento logico che viene eseguito per risolvere un problema
- **Programma:** Descrizione di un algoritmo mediante un formalismo (linguaggio) che lo rende comprensibile all'**elaboratore**

- **Processo:** Insieme ordinato di eventi che l'elaboratore genera sotto il controllo del programma

Processi sequenziali e non sequenziali

- **Processo sequenziale**: può essere rappresentato da un grafo di precedenza in cui
  - nodi: singoli eventi
  - archi: precedenze temporali

  e ad **Ordinamento Totale** (Ogni nodo ha esattamente un predecessore e un successore)
- **Processo non sequenziale:**
  - grafo ad **ordinamento parziale**
  - L'esecuzione di un processo non sequenziale richiede:
    - elaboratore non sequenziale (in grado di effettuare più operazione contemporaneamente)
    - linguaggio di programmazione non sequenziale
  - è necessario individuare dei **vincoli di sincronizzazione o di precedenza —> processi iteragenti:**
    - esempio 1: L1, L2, e L3 devono essere eseguiti in sequenza MA loro esecuzione può essere fatta in contemporanea a E1, E2, E3.
    - esempio 2: E1, E2 ed E3 devono essere eseguiti sempre DOPO il termine di L1, L2 e L3
    - i processi devono essere in grado di **scambiarsi informazioni**
    - i processi ad un certo punto **devono sincronizzarsi**



Interazioni tra processi

Possibili interazioni:

- **Cooperazione:** comprende tutte le interazioni *prevedibili* e *desiderate.* Prevede:
  - scambi di informazioni:
    - segnali temporali
    - dati
  - sincronizzazione

- **Competizione:** Coordinamento dei processi nell'accesso alle risorse **condivise** (mutex). Sono comportamenti *prevedibili, non desiderati* **ma *necessari***
  - Esempio: **Mutua esclusione**
  - L'ordine non è importante! Bisogna solo evitare di accedere alla stessa risorsa contemporaneamente
  - **Sezione critica:** sequenza di istruzioni con le quali richiede l'accesso ad un oggetto condiviso con altri processi. ***Sezioni critiche appartenenti alla stessa classe devono escludersi mutuamente nel tempo!***
- **Interferenza:** Interazione provocata da **errori di programmazione.** Situazione *non prevista* e *non desiderata*
  - dipende dalla **velocità relativa** dei processi
  - errori **dipendenti dal tempo**
  - **classi di interferenza:**
    - **primo tipo:** solo P deve operare su una risorsa R ma per errore viene inserita in Q un'istruzione che modifica R. —> la condizione di errore si presenta solo a certe velocità
    - **secondo tipo:** P e Q competono per una stampante comune. (mutex gestito male —> Errore)

Architetture e linguaggi per la programmazione concorrente

Disponendo di un calcolatore concorrente e di linguaggi di programmazione che possono descrivere algoritmi non sequenziali possiamo definire l'elaborazione complessiva come ***Processi sequenziali asincroni interagenti***

Proprietà di un linguaggio per la programmazione concorrente

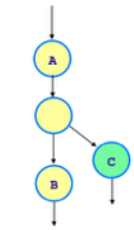
- Un linguaggio deve:
  - contenere costrutti per specificare processi distinti
  - avere costrutti per specificare se un processo deve essere attivato o terminato
  - avere costrutti per specificare le interazioni tra processi

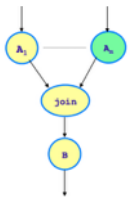
Architettura di una macchina concorrente

- M è una macchina astratta ottenuta con tecniche software basandosi su una macchina fisica M' molto più semplice
- Deve prevedere dei **sistemi di protezione**:
  - utile ad individuare eventuali anomalie e interferenze
  - può essere implementato in hardware o software
  - *Capabilities e liste di controllo degli accessi*

- Nel nucleo (kernel) devono essere presenti:
  - **meccanismo di multiprogrammazione** (*scheduling*)
  - **meccanismo di sincronizzazione e comunicazione**
- A seconda dell'organizzazione logica della macchina (multiprocessore o distribuito) le due macchine definiscono **due modelli di interazione tra i processi:**
  - **Modello a *memoria comune:*** Le interazione tra i processi avviene su oggetti contenuti sulla memoria comune
  - **Modello a *scambio di messaggi:*** Comunicazione tra processi attraverso messaggi scambiati sulla rete che collega gli elaboratori.

Costrutti linguistici per la concorrenza





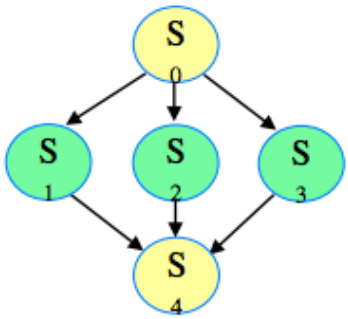
Fork/Join

FORK: Creazione e attivazione di un processo che inizia la propria esecuzione in parallelo con quella del processo chiamante

JOIN: Consente di determinare quando un processo creato tramite **fork** ha terminato il suo compito

Cobegin/Coend

```
S0;
cobegin
   S1;
   S2;
   S3;
coend
S4;
```



Istruzioni S_1, S_2 ed S_3 eseguite in parallelo. Ogni S_i può contenere altre **cobegin/coend** al suo interno.

Processo

Costrutto linguistico per identificare quali moduli di un programma possono essere eseguiti come processi autonomi.

Realizzazione delle primitive in un sistema a monoprocessore

Descrittori dei processi

Ogni processo viene identificato con:

- un numero intero (**PID**). A questo scopo il **kernel** definisce una funzione **assegna_nome** che, chiamata all'atto della creazione del processo restituisce il suo **PID.**

PID *assegna_nome();*

- una **modalità di servizio** (per scheduling con priorità) composta da:
  - priorità (intero)
  - delta_t (time_slice)
- **contesto di esecuzione**
- **stato**: running, ready ecc…
- **PID del padre**
- **numero figli**
- **lista dei figlio,** ogni figlio ha:
  - **PID**
  - **terminato:** booleano
- **puntatore al processo successivo**

I descrittori sono organizzati in opportune code del tipo:

typedef struct {
p_des primo,
p_des ultimo
} **coda_des**;

Con code dei **processi pronti** (scheduling) e code **dei descrittori liberi**

Meccanismi realizzati dal Kernel

- **Cambio di contesto:**
  - Si salva lo stato tramite i valori contenuti nei registri della CPU nel contesto di esecuzione del descrittore del processo
  - Si ripristina lo stato dei registri con il valore contenuto nel contesto di esecuzione del descrittore del processo.
- **Scheduling dei processi:**
  - Finché non ci sono processi pronti —> aspetto

  - Appena ho pronto un processo
    - lo prelevo dalla coda dei descrittori
    - imposto il suo stato su running (p -> stato = <running>)
    - lo metto in esecuzione
    - imposto il registro temporizzatore della CPU con il delta_t della modalità di servizio (p -> servizio.delta_t).
- **Attivazione di un processo:**
  - Dopo una fase di sospensione, porta P nello stato di pronto (coda dei processi pronti). Nel caso il sistema preveda ***preemption,*** il processo, se ha **priorità maggiore** di quello in esecuzione, ne prende il posto (quello in esecuzione viene inserito nella coda dei processi pronti); in caso contrario lo stato viene settato a *ready* e viene inserito nella coda dei processi pronti
- **Fork:**
  - Se non ci sono descrittori liberi —> ritorna un eccezione
  - Altrimenti io processo:
    - Prelievo un descrittore libero p*
    - Lo inizializzo gli assegno un PID
    - Metto il mio PID come PID del padre
    - Inserisco p* nei miei figli e metto *terminato* a false
    - incremento numero_figli
    - richiedo l'attivazione del processo (vedi sopra)
- **Join:**
  - prelevo il figlio da joinare dalla mia lista di figli
  - se il figlio ha *terminato = false*:
    - mi sospendo (*stato = <sospeso>*) in attesa che il figlio termini
    - chiedo l'assegnazione della CPU (scheduling)
- **Quit:**
  - prelevo il PID di mio padre
  - setto il mio *stato=terminato* nella lista di descrittori del padre (*padre->prole[k].terminato = true*)
  - inserisco il mio handle nella coda dei descrittori liberi.
  - Se il padre è sospeso (*padre->stato == <sospeso>*)
    - Inserisco mio padre nella lista dei processi pronti
    - Imposto il suo stato a *<ready>*
- **Time Sharing:**
  - Per consentire modalità di servizio a *divisione di tempo* occorre **revocare** ad intervalli fissati di tempo la CPU al processo in esecuzione e di assegnarla ad un nuovo processo pronto (dà l'illusione della contemporaneità tra i processi)

Proprietà dei programmi

Attività più importante per il programmatore è seguire:

- la *traccia dell'esecuzione* ovvero la **sequenza di stati attraversati dal sistema durante l'esecuzione del programma**
- lo *stato di esecuzione ovvero* l'**insieme dei valori delle variabili definite dal programma e delle variabili implicite** (*program counter ecc…*)

In particolare:
- **Programmi sequenziali** —> ogni esecuzione di un programma P su un insieme di dati D **produce sempre la stessa traccia**
- **Programmi concorrenti** —> l'esito dipende dall'effettiva sequenza cronologica di esecuzione. In genere **producono tracce diverse**

Definizione di proprietà

*Una **proprietà** di un programma è un **attributo** che è sempre vero in ogni possibile traccia generata dall'esecuzione di P*

Tipi di proprietà

In generale le proprietà si dividono in 2 categorie:
- **Safety Properties:** garantiscono che durante l'esecuzione di P **non si entrare mai in uno stato errato.** (Ovvero uno stato in cui le variabili assumono valori *non desiderati*)
- **Liveness Properties:** garantiscono che durante l'esecuzione di P, **prima o poi si entrerà in uno stato corretto**

Requisiti

Per ogni programma **sequenziale** devono essere:
- **Correttezza del risultato finale:** Per ogni esecuzione il risultato ottenuto è giusto **(Safety)**
- **Terminazione:** Prima o poi l'esecuzione termina **(Liveness)**

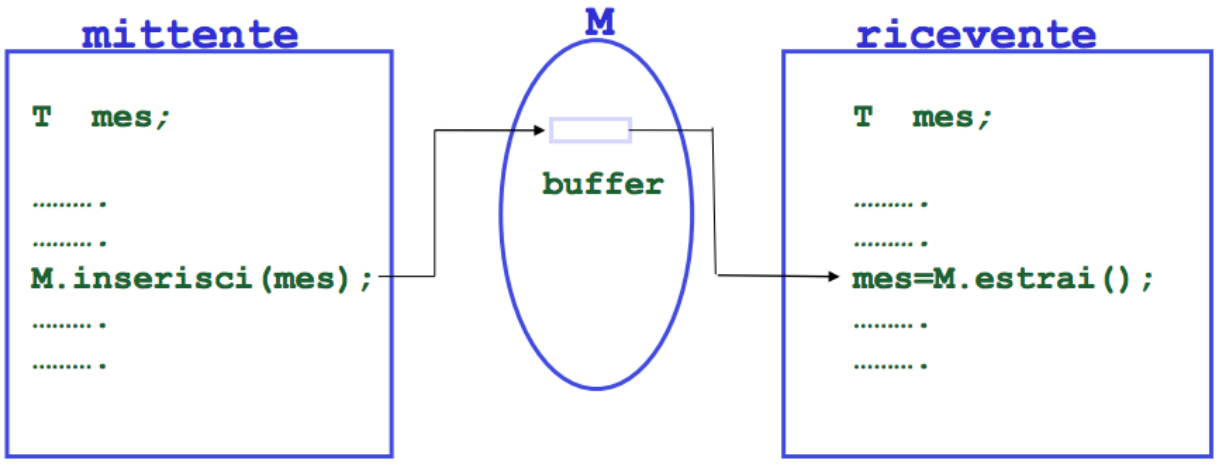
Inoltre per ogni programma **concorrente** devono essere:
- **Mutua esclusione nell'accesso a risorse condivise:** per ogni esecuzione non accadrà mai che più di un processo acceda alla stessa risorsa contemporaneamente **(Safety)**
- **Assenza di deadlock:** per ogni esecuzione non si creeranno mai situazioni di blocco **(Safety)**
- **Assenza di starvation:** prima o poi ogni processo potrà accedere alla risorsa richiesta **(Liveness)**

Modello a memoria comune

Ogni interazione avviene tramite oggetti contenuti nella memoria comune
- **Risorsa:** qualunque oggetto, fisico, logico che un processo necessità per portare a termine il suo compito

- Sono raggruppate in *classi.* Una classe identifica l'insieme di tutte le operazioni che un processo può eseguire su risorse di quella classe
- Struttura dati allocata nella memoria comune.

Necessità di creare un **meccanismo di controllo degli accessi**.



Gestori di risorse

Per ogni risorsa R definiamo un **Gestore(R)** che definisce in ogni istante t l'insieme ∑R(t) di processi che, in tale istante, hanno diritto di operare su R

Classificazione delle risorse

Risorsa R:
- **dedicata**: se ∑R(t) <= 1
- **condivisa:** se ∑R(t) > 1
- **allocata staticamente:** se ∑R(t) è una costante
- **allocata dinamicamente:** se ∑R(t) è in funzione del tempo.

	Risorse dedicate	**Risorse condivise**
Risorse allocate staticamente	Risorse private	Risorse comuni
Risorse allocate dinamicamente	Risorse comuni	Risorse comuni

Il gestore di una risorsa è, in questo caso, **il programmatore.**

Compiti del gestore di una risorsa

1. Mantenere **aggiornato** ∑R(t) ovvero lo stato di allocazione della risorsa
2. Fornire i **meccanismi** che un processo può utilizzare per acquisire il diritto di accedere ad una risorsa ed entrare a far parte di ∑R(t)
3. Implementare la **strategia** di allocazione della risorsa, ovvero definire quando, a chi e per quanto tempo allocare la risorsa

Data una risorsa R il suo Gestore G_R è costituito da:
- una **risorsa condivisa**: nel caso di un sistema a memoria comune
- un **processo:** in un sistema a scambio di messaggi

Accesso alle risorse

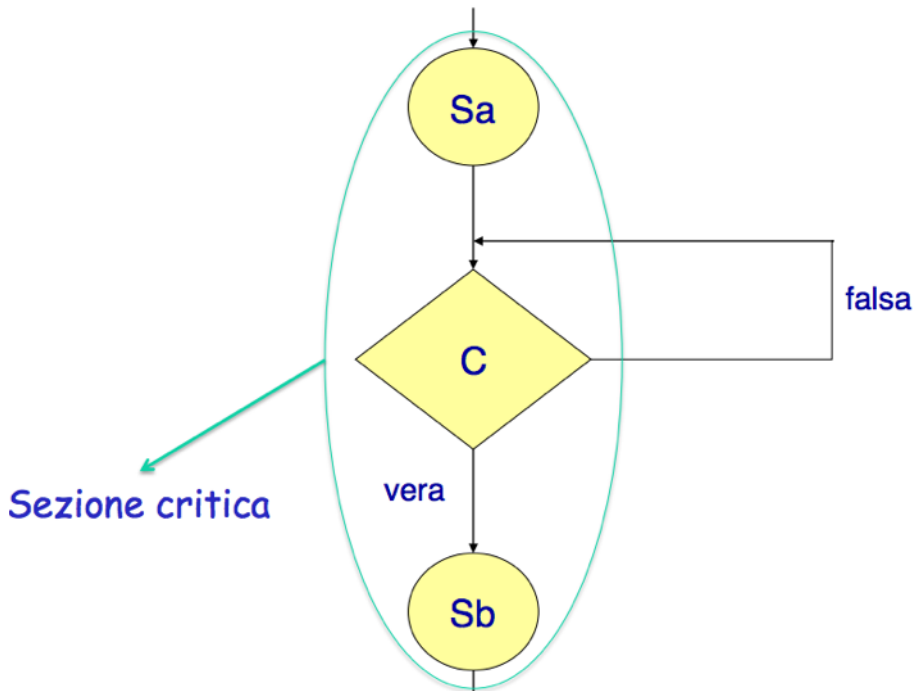
- Se R è allocata **staticamente** a P allora il processo possiede il diritto di operare in **qualunque istante**
- Se R è allocata **dinamicamente** a P allora il gestore G_R dovrà prevedere un meccanismo di richiesta e rilascio della risorsa.

GR.richiesta();
R.operazione()
GR.rilascio();

- Se R è allocata come **risorsa condivisa:** è necessario che gli accessi avvengano in modo atomico.
- Se R è allocata com **risorsa dedicata:** non è necessario prevedere nessun tipo di sincronizzazione (unico processo che può accedere è P)

Sincronizzazione condizionale

region R << S_a; when(C) S_b;>>
- *<<…>>* rappresenta una **sezione critica**
- Al termine dell'esecuzione di S_a se C è vera si procede a valutare S_b altrimenti si attende che C diventi vera.



Casi particolari:

- *region R << S;>>*: specifica la sola **mutua esclusione** senza sincronizzazione diretta
- *region R << when(C)>>:* specifica di un semplice **vincolo di sincronizzazione**
- *region R << when(C) S;>>*: specifica il caso la condizione C debba essere vera **prima** di eseguire l'operazione S (**precondizione**)

Mutua esclusione

Azione atomica: esegue una trasformazione di stato indivisibile. Può esistere uno stato intermedio. Ma da fuori non è rilevante.

Ipotesi:

- I valori dei tipi base sono memorizzati in parole che vengono lette e scritte in modo *atomico*
- I valori sono manipolati caricandoli su registri, operando su dei registri e memorizzandoli in memoria
- Ciascun processo ha il proprio set di registri.

Sezione critica

La sequenza di istruzioni con le quali un processo accede, modifica un insieme di variabili comuni prender il nome di **sezione critica.**

Ad un insieme di variabili comuni possono essere associate *una solo sezione critica* (usata da tutti i processi) o *più sezioni critiche* (classe di sezioni critiche)

La regola di mutua esclusione definisce che:

Sezioni critiche appartenenti alla stessa classe devono escludersi mutuamente nel tempo

oppure

Ad ogni istante può essere "in esecuzione" al più 1 sezione critica di ogni classe

Schema generale di soluzione

Ogni processo deve:

- chiedere **l'autorizzazione per l'uso esclusivo della risorsa,** prima di entrare in una sezione critica (**PROLOGO**)
- **liberare la risorsa,** una volta conclusa la sezione critica (**EPILOGO**)

Soluzione algoritmiche

La soluzione non richiede la disponibilità di meccanismi di sincronizzazione (lock, semafori) ma sfrutta la sola possibilità di condividere delle variabili. Tra i vari algoritmi ricordiamo:

1. **Algoritmo di Dekker**
   - Prima soluzione corretta al problema (vedi algoritmi)
   - 3 variabili:
     - int libero1 = 0

  - int libero2 = 0
  - int turno = 1 dominio {1,2}
2. **Algoritmo di Peterson**
   - Più semplice di quello di Dekker (vedi algoritmi)
   - stesse 3 variabili
3. **Algoritmo del fornaio**
   - Ogni processo preleva un ticket con un numero sempre crescente. Entra nella sezione critica quando il numero in suo possesso è il minimo tra tutti quelli non serviti.
   - 2 variabili:
     - int np = 0
     - int nq = 0
   - Difficile implementazione
   - Ogni thread deve chiedere il numero a tutti gli altri

In generale:
- **Evitare la starvation**
- **Evitare deadlocks**
- **Evitare la busy waiting** (attesa attiva —> lack of performance)

Soluzioni Hardware

- **Disabilitazione delle interruzioni durante le sezioni critiche:**
  - Prologo —> Disattivo
  - Epilogo —> Attivo
  - Soluzione parziale perché è valida solo per sezioni critiche che operino sullo **stesso processore, inoltre *elimina* ogni possibilità di parallelismo.**
- **Lock/Unlock:**
  - Molte macchine possiedono particolari istruzioni che permettono di esaminare e modificare il contenuto di una parola e di un altra in un ciclo di memoria
  - **Ipotesi:**
    - Operazione di lock/unlock sono indivisibili —> l'esecuzione delle due operazioni viene **automaticamente** sequenzializzata dall'hardware.
      - Questa ipotesi viene garantita dall'istruzione test-and-set(x) che permette la lettura e la modifica di una parola in memoria in modo *indivisibile* (in un solo ciclo di memoria)
  - I requisiti sono quasi tutti soddisfatti, tranne quello della starvation che non è implicitamente soddisfatto e il requisito dell'attesa attiva.
  - Si applica in ambiente multiprocessore
  - Comunque **attesa attiva**

Strumenti di sincronizzazione

Il Semaforo

- Strumento linguistico di basso livello per risolvere problemi di sincronizzazione nel modello a memoria comune.
- Realizzato nel kernel della macchina.
- Usato per costruire strumenti più di alto livello

Definizione

Un semaforo è una **variabile intera non negativa** a cui è possibile accedere solo tramite due operazione ***P*** e ***V***

Operazioni sul semaforo

Un oggetto di tipo **semaphore** è condivisibile tra due o più thread, che operano su di esso attraverso le operazioni P e V.

- **P**: *region<<when(val_s>0) val_s- -;>>*
- **V***: region<<val_s++>>*

Il semaforo viene usato per sospensione e risveglio:

- **Sospensione**: P(s), val_s == 0
- **Risveglio:** V(s), se vi è almeno un processo sospeso.

Proprietà del semaforo

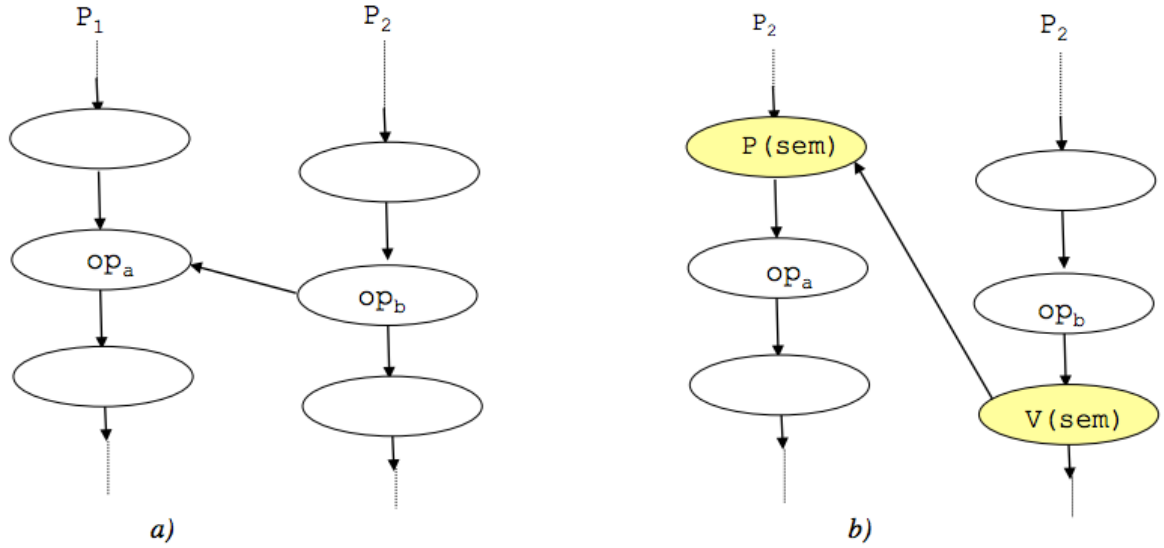
- **Relazione di invarianza.**
  - Ad ogni istante possiamo esprimere il valore del semaforo come: $val_s = l_s + nv_s - np_s$ da cui —> $np_s <= l_s + nv_s$
  - La relazione di invarianza è sempre verificata per ogni semaforo.

Semafori particolari

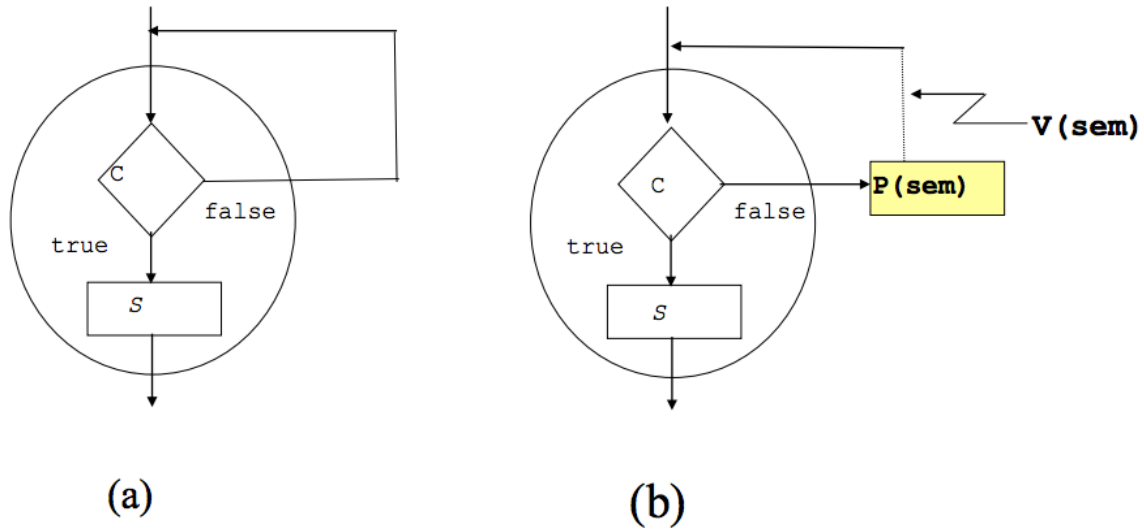
- **Semaforo binario (mutua esclusione)**
  - può assumere solo i valori 0 o 1
  - soddisfa la mutua esclusione —> Dimostrazione
- **Semaforo per mutua esclusione di gruppi di processi**
  - Necessari 2 semafori di mutua esclusione.
  - Blocco il primo.
  - Incremento un contatore (coda)
  - Se il contatore == 1
    - blocco il secondo mutex
    - libero il primo mutex
  - eseguo l'operazione i-esima.

- blocco il primo
- decremento il contatore (coda)
- se contatore == 0
  - libero il secondo mutex
- libero il primo mutex

- **Semafori evento —> Dimostrazione**
  - Semaforo binario utilizzato per imporre un vincolo di precedenza tra operazioni.
  - Un semaforo *sem*, inizializzato a 0.
  - prima di eseguire l'operazione op_a P_1 esegue P(*sem*)
  - dopo avere eseguito l'operazione op_b P_2 esegue V(*sem*)



- **Semafori condizione**
  - L'esecuzione di un'istruzione S1 su una risorsa R è subordinata al verificarsi di una condizione C.



Lo schema **a** prevede una forma di attesa attiva da parte del processo che non trova soddisfatta la condizione.
Nello schema **b** invece si sospende il processo su semaforo *sem* da associare alla condizione (semaforo condizione)

- **Semafori risorsa**
  - Il valore del semaforo rappresenta il numero di risorse libere.
  - Usato per gestire un pool di risorse equivalenti.
- **Semafori privati**
  - Un semaforo è **privato** per un processo se **solo tale processo può eseguire la primitiva P** su tale semaforo.
  - Viene sempre inizializzato con il valore 0
  - Un processo si sospende sul proprio semaforo privata *sem* che un altro processo risveglierà tramite una V(*sem*)

Realizzazione dei semafori

Il semaforo viene realizzato dal **kernel**
Descrittore di un semaforo:

- int contatore;
- coda queue;

Una P su un semaforo con contatore a 0, **sospende** il processo nella coda queue, altrimenti il contatore viene decrementato. Una V su un semaforo con una queue non vuota, estrae un processo dalla coda, altrimenti incrementa il contatore.

Threads e Processi

Threads vs Processi

- **Processi pesanti:**
  - spazi di indirizzamento distinti (**non** condividono memoria)
  - complesse operazioni di cambio contesto —> salvataggio e ripristino dello spazio di indirizzamento —> overhead
  - Utilizzano modello a scambio di messaggi
  - Possesso delle risorse
  - Esecuzione: schedulato rispetto ad altri processi pesanti
- **Thread:**
  - elemento a cui viene assegnata la CPU
  - flusso di esecuzione all'interno di un processo pesante.
  - tutti i thread definiti in un processo condividono le risorse del processo —> modello a memoria comune
  - ogni thread ha:
    - uno **stato**: *ready, running, blocked*
    - un **contesto:** è salvato quando il thread non è in esecuzione
    - uno **stack di esecuzione**:

  - uno spazio di memoria **privato per le variabili locali**
  - accesso alla memoria e alle risorse del task **condiviso con altri thread**
- **vantaggi:**
  - **maggiore efficienza** (nella creazione e distruzione dei thread)
  - maggiore possibilità di utilizzare sistemi multiprocessore

I threads nelle varie architetture

- MS-DOS: 1 processo utente e un solo thread
- UNIX: più processi utente, ciascuno con un thread
- JAVA: 1 processo utente, più thread
- Linux, Windows, ecc..: più processi utente, ciascuno con più thread

Realizzazione dei thread

A livello utente

Ad esempio **java**

- libreria che opera a livello utente e fornisce il supporto per la creazione, terminazione e sincronizzazione dei thread.
- il sistema operativo **ignora** la presenza dei thread.
- ogni processo parte con un solo thread e ne può creare di nuovi.
- problemi:
  - una chiamata a system call blocca tutto il processo pesante e quindi tutti i threads.
  - non si può sfruttare il parallelismo delle architetture multiprocessore.

A livello kernel

Ad esempio **Linux**

- l'OS si occupa di gestire i thread.
- mantiene tutti i descrittori dei thread
- quando un thread si blocca, l'OS può mettere in esecuzione un altro thread dello stesso processo.
- soluzione **meno efficiente** della precedente.
- Si può inserire su architetture multiprocessore

Il Kernel di un sistema a processi

Definizione

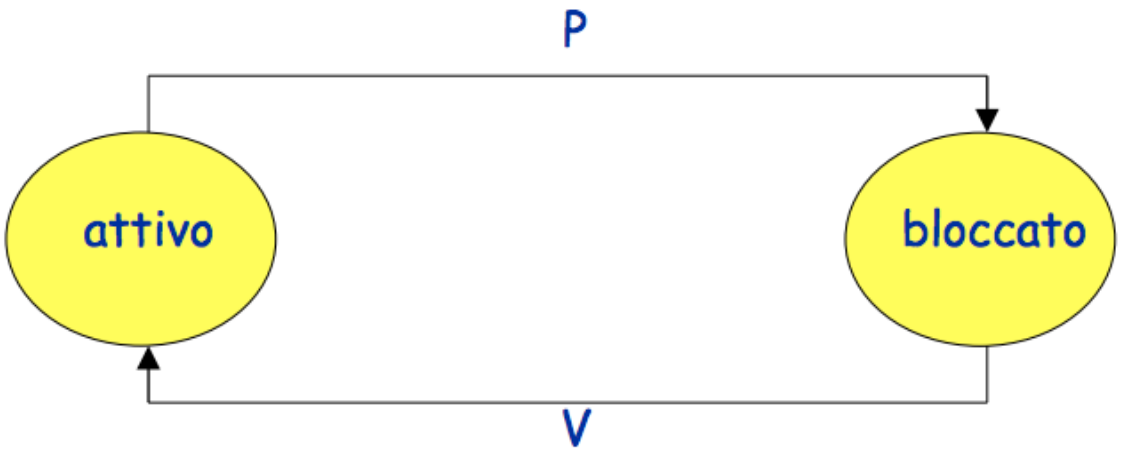
*Si chiama **Nucleo (kernel)** il modulo realizzato in software, hardware o firmware che supporta il concetto di processo e realizza gli strumenti necessari per la gestione dei processi.*

Il nucleo è il *livello più interno* di qualsiasi sistema multiprogrammato. È inoltre l'unico modulo che è **conscio** dell'esistenza delle **interruzioni**.

Caratteristiche

- **Efficienza:** condiziona l'intera struttura
- **Dimensioni:** estremamente limitate
- **Separazioni tra meccanismi e politiche:** dovrebbe contenere solo meccanismi per decidere le politiche a livelli superiori.

Stati di un processo



Quando un processo perde il controllo del processore, il contenuto dei registri del processore viene salvato in un'area di memoria chiamata **descrittore**.
Ciò consente **maggiore flessibilità** nella politica di assegnazione del processore ai processi rispetto a salvare le informazioni nello **stack.**
Contesto di un processo: insieme di informazioni contenuti nei *registri del processore*

Compiti del kernel

Funzione fondamentale del kernel è la gestione delle **transizioni di stato** dei processi.
Il nucleo dovrà:

- **Gestire il *salvataggio e ripristino* dei contesti dei processi:**
  - salvare tutte le informazioni nel descrittore quando un processo termina/si sospende
  - caricare le informazioni nei registri quando un processo parte/riparte
- **Scegliere a quale processo assegnare la CPU (scheduling)**
- **Gestire le interruzioni dei dispositivi esterni**
- **Realizzare le primitive di sincronizzazione dei processi.**

Strutture dati

Descrittore del processo

Contiene le seguenti informazioni:
- **Identificazione del processo (id)**
- **Modalità di servizio dei processi**
  - FIFO
  - Priorità (fissa o variabile)
  - Deadline: tempo massimo entro cui la richiesta deve essere soddisfatta.
- **Contesto del processo:**
  - Program conunter, registro di stato, registri generali, indirizzo dell'area privata al processo
- **Identificazione del processo successivo**
  - per gestire le code

Le code

Coda dei processi pronti
- Ne esistono una o più di una (in caso di processi con priorità). Quando un processo viene riattivato viene inserito in fondo alla coda corrispondente la sua priorità.
- contiene sempre almeno un **processo fittizio (dummy)** che va in esecuzione quando tutte le altre code sono vuote (ha la priorità più bassa ed è sempre nello stato di pronto)
- il processo dummy rimane in esecuzione finchè non c'è un altro processo pronto.

Coda dei descrittori liberi
- Coda nella quale sono concatenati i descrittori disponibili per la creazioni di nuovi processi
- In essa vi sono inseriti i descrittori dei processi terminati

Id del processo in esecuzione

Viene contenuto all'interno di un particolare registro della CPU (Viene inizializzato al bootstrap del sistema)

Funzioni del kernel

Realizzano le funzione di transizione di stato
- **Inserimento e Prelievo** di un descrittore da una coda. Se la coda è vuota si ipotizza che il primo campo (ma anche l'ultimo) abbia valore -1

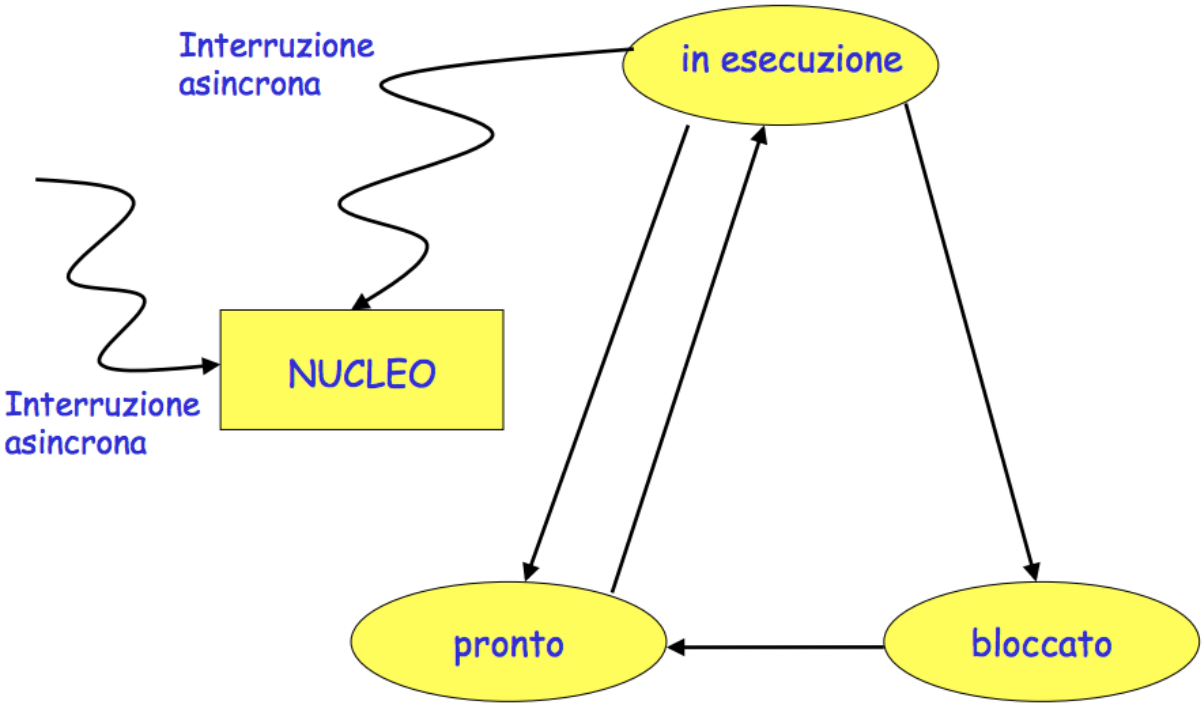
Le funzioni del kernel vengono suddivise in due livelli:
1. **Livello superiore:** contiene tutte le informazioni utilizzabili dai processi sia esterni (I/O) che interni
   1. risposta alle interrupt
   2. primitive per la *creazione, eliminazione e sincronizzazione dei processi*
2. **Livello inferiore:** realizza le funzionalità di:

1. Cambio di contesto
2. Scelta di un nuovo processo da mettere in esecuzione

Le funzioni del nucleo, per motivi di sicurezza sono le sole che
- possono accedere alle strutture dati precedenti
- possono usare istruzioni privilegiate (*abilitazione interrupt, invio di comandi ai dispositivi*)

Inoltre le funzioni devono essere eseguite in maniera mutuamente esclusiva.
I due ambienti di esecuzione (kernel e utente) possono essere switchati con delle interruzioni.
In particolare:
- in caso di processi **esterni:** il passaggio al modo kernel è ottenuto tramite **interrupt**
- nel caso di processi **interni:** il passaggio avviene tramite l'esecuzione di system calls

In entrambi i casi al completamento della funzione si ritorna all'ambiente di provenienza con una **RTI (ritorno da interruzione)**



Funzioni di livello inferiore: Cambio di contesto

- Salvataggio del contesto nel suo descrittore (**Salvataggio stato**)
- Inserimento del descrittore nella coda dei processi bloccati o dei processi pronti
- Rimozione del processo a maggior priorità all coda dei pronti ed esecuzione (**assegnazione CPU**)
- caricamento del contesto del nuovo processo (**Ripristino stato**)

Gestione del temporizzatore

Ad intervalli regolari un dispositivo temporizzatore dovrà sospendere il processo in esecuzione ed assegnare l'unità di elaborazione ad un altro processo.

Funzioni di livello inferiore: Semaforo (monoprocessore)

Viene realizzato tramite:
- Una variabile intera
- Un puntatore ad una lista di descrittori di processi in attesa sul semaforo
- Se non ci sono processi in coda, il puntatore contiene **nil**
- La coda viene gestita a **FIFO**

Passaggio di ambiente

Costituito da un meccanismo di **interruzioni**.
Ad ogni processo viene associata una pila (*stack*) gestita tramite il registro *stack point.* La pila rappresenta l'area di lavoro del processo e contiene variabili temporanee ed i ricordi di attivazione
Registri utilizzati:
- PC e PS(Registri di stato), R1, R2,….Rn, R1', R2',….Rn' SP1, SP1' (registri generali e stack pointer) associati rispettivamente a **nucleo** e **processi**

L'esecuzione di una primitiva da parte di P corrisponde all'esecuzione di un'istruzione di tipo *SVC*
1. **Interruzione** di tipo *sincrono*
2. **Salvataggio di PC e PS** di P in cima alla pila del kernel
3. **Caricamento di PC e PS** dell'indirizzo della procedura di risposta all'interruzione e di PS del nucleo
4. **Esecuzione della procedura di risposta**
5. **P passante**: esecuzione di **ritorno dall'interruzione (RTI)** che ripristina in PC e PS i valori del processo contenuti nella pila del nucleo
6. **P bloccante (non passante): salvataggio stato…**

Kernel in sistemi multi-processore

Vari modelli possibili:
1. **Modello SMP:** una copia del nucleo condivisa tra tutte le CPU
2. **Modello a nuclei distinti**

Simmetric Multi Processing (SMP)

- Una sola copia del nucleo
- Competizione tra CPU nell'esecuzione del nucleo —> **sincronizzazione**

Sincronizzazione —> più possibili soluzioni:
- **Soluzione ad un solo lock:**
  - Accesso esclusivo con un solo lock applicato al kernel

  - Limita il parallelismo —> evita che le due CPU lavorino insieme
- **Soluzione a più lock:**
  - lock distinti su:
    - **code processi pronti**
    - **singoli semafori**
  - In generale conviene che un processo venga eseguito su un solo processore; in particolare sul processore che possiede il codice del processo nella sua memoria privata (accesso veloce)

Realizzazione dei semafori in SMP

Tutte le CPU condividono lo stesso nucleo. Vengono sequenzializzati solo gli accessi alla coda dei processi pronti.

Se si ha uno scheduling pre-emptive basato su priorità, 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. In tal caso:

- Il nucleo deve provvedere a *revocare l'unità di elaborazione al processo con **priorità più bassa*** ed *assegnarla al processo risvegliato dalla V*
- Necessario un meccanismo di **segnalazione inter-processore**

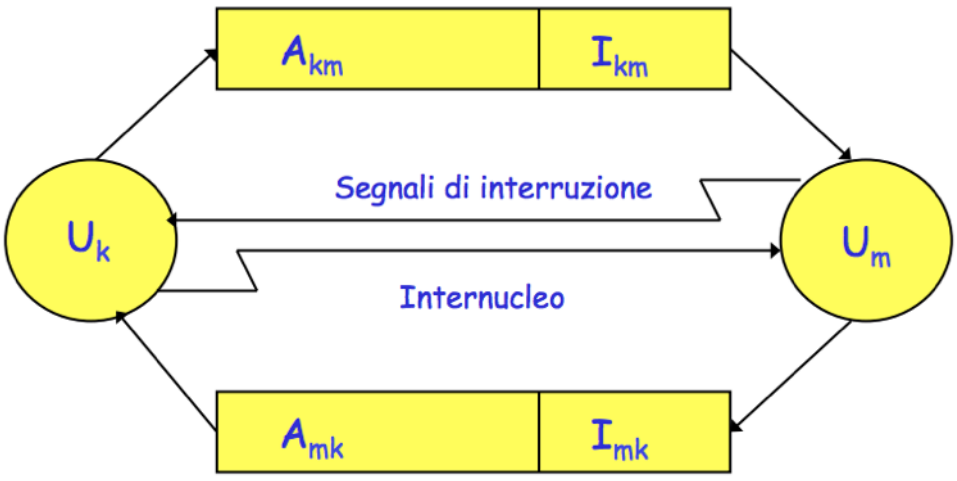
Modello a nuclei distinti

- Ogni CPU gestita da un nucleo distinto
- Massima concorrenza. Maggiore scalabilità.
- Il modello SMP fornisce presupposti per un migliore **bilanciamento del carico** (carico diviso in egual modo tra le CPU)

Realizzazione dei semafori (caso nuclei distinti)

Va fatta distinzione tra:

- **semafori privati** di un nodo singolo nodo U
- **semafori condivisi:** cioè utilizzati da processi appartenenti a nodi differenti
  - ogni semaforo sarà protetto da un **lock x**



- **A_{km}, A_{mk}** sono aree di comunicazione tra i nodi
- **N_k** inserisce in A_{km} l'identità del processo **P_j** risvegliato e lancia un'interruzione a **N_m**
- **N_m** provvede a prelevare le informazioni contenuti in **A_{km}** ed a eseguire e operazioni descritte.
- **I_{km}** e **I_{mk}** sono indicatori per sincronizzare le operazioni tra nuclei.
  - Se I_{km} ha valore 1 significa che l'informazione contenuta in A_{km} non è ancora stata prelevata da N_m . Se ha valore 0 significa che A_{km} è disponibile per un nuovo messaggio.

Il Monitor

Il costrutto monitor

Definizione

Costrutto sintattico che associa un insieme di operazioni pubbliche ad una struttura dati comune a più processi

Le operazioni pubbliche sono le sole operazioni eseguibili e sono **mutuamente esclusive.** Le variabili sono permanenti nel tempo e sono accessibili solo all'interno del monitor (private).

Uso del monitor

L'assegnazione della risorse avviene tramite 2 **livelli di controllo**:
- **Un solo** processo alla volta può accedere alle variabili comuni
- Viene controllato **l'ordine di accesso** al monitor (code su condizioni)

Nel caso in cui le condizioni di accesso non siano verificate il processo si sospende su una **condition**.

Variabili condizione

Ogni variabile di tipo condizione rappresenta una coda nella quale i processi si sospendono.
Operazioni eseguibili su una coda:
- *wait(cond)*: sospende il processo inserendolo nella coda cond
  - specificando wait(cond, priorità) è possibile specificare una priorità al processo.
- *signal(cond)*: risveglia un processo in attesa nella coda cond

Esistono anche la signal_and_wait() e la signal_and_continue() per evitare che il processo che chiami la signal interferisca con l'accesso al monitor del processo risvegliato.

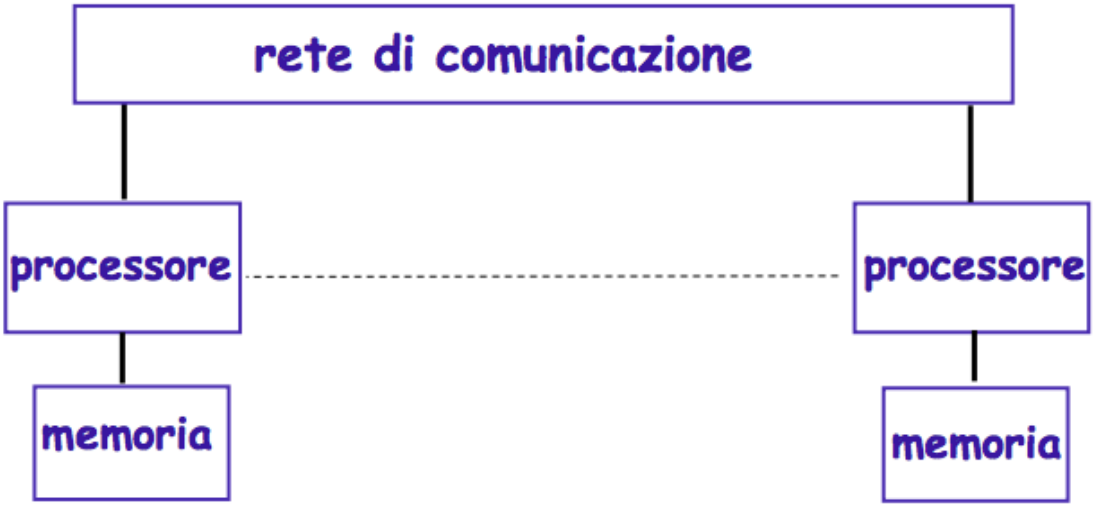
signal_and_urgent_wait: processo Q manda una signal a P, si sospende in una coda urgent interna al monitor. Quando P ha finito passa il controllo a Q **senza liberare il monitor.** In questo modo Q ha la priorità su tutti gli altri processi in attesa.
signal_and_return: Processo Q cede la sua esecuzione al Processo P **senza liberare il monitor**.

È possibile risvegliare tutti i processi con la signal_all().
È possibile specificare variabili condizione **monoprocesso** —> Array di variabile condizione (una per ogni processo) —> permette di risvegliare selettivamente un processo preciso.

Monitor tramite semafori

Viene assegnato al monitor
- un semaforo **mutex** inizializzato a 1 per l'accesso in mutua esclusione alle risorse
- un semaforo **condsem** inizializzato a 0 sul quale il processo si può sospendere tramite una wait(consem)

Modello a scambio di messaggi



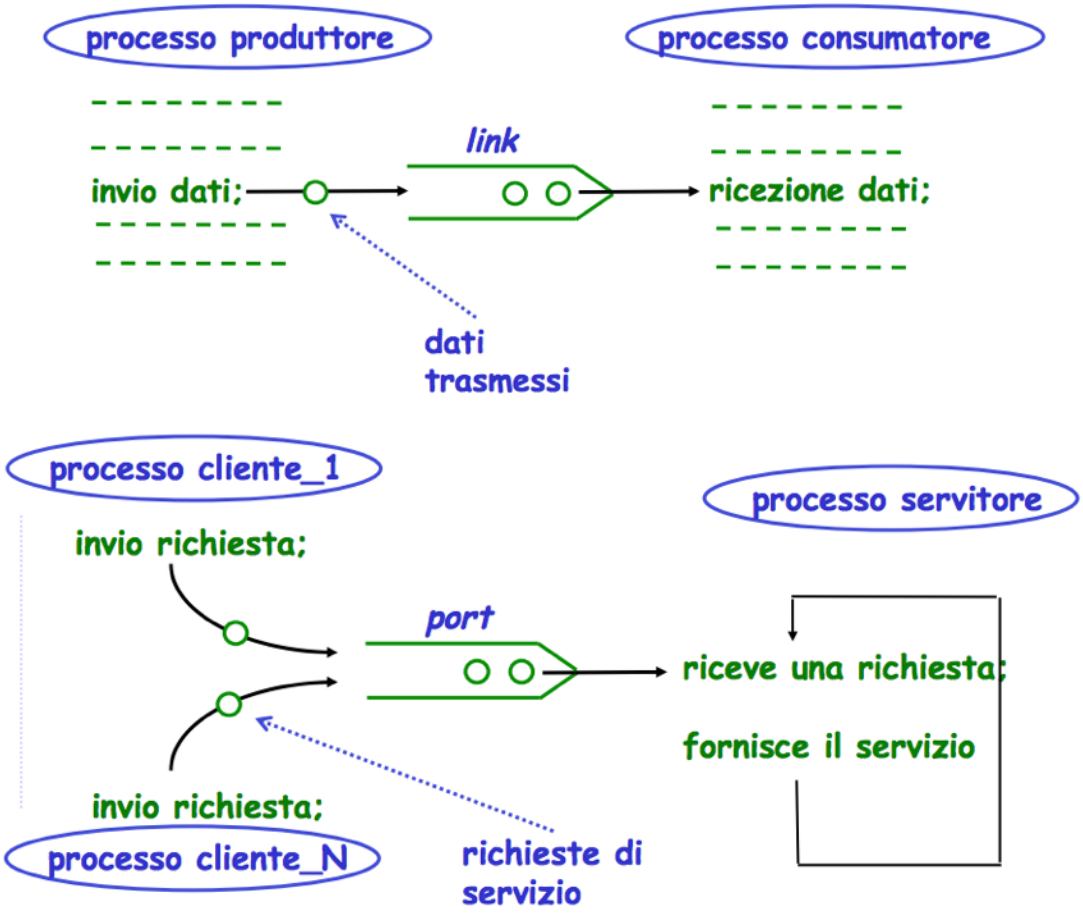
- Ogni processo può accedere esclusivamente alle risorse allocate nella propria memoria locale
- Ogni risorsa del sistema è accessibile ad un solo processo
- Se una risorsa è necessaria a più processi. Il processo interessato dovrà chiedere all'unico processo che vi può accedere di effettuare le operazioni richieste.
- Coincide col il concetto di **client-server**

Il Canale

È il concetto fondamentale nel modello a scambio di messaggi. Inteso come collegamento logico tra due processi. È compito del kernel del sistema operativo fornire l'astrazione del canale.

Caratteristiche

- **Tipologia:** intesa come **direzione del flusso**
  - Canali **monodirezionali**
  - Canali **bidirezionali**
- **Designazione:** sorgente e destinatario di ogni comunicazione
  - **link:** da uno a uno
  - **port:** da molti a uno
  - **mailbox:** da molti a molti
- Tipo di **sincronizzazione** tra i processi comunicanti
  - Comunicazione **sincrona**
  - Comunicazione **asincrona:**
  - Comunicazione con **sincronizzazione estesa**



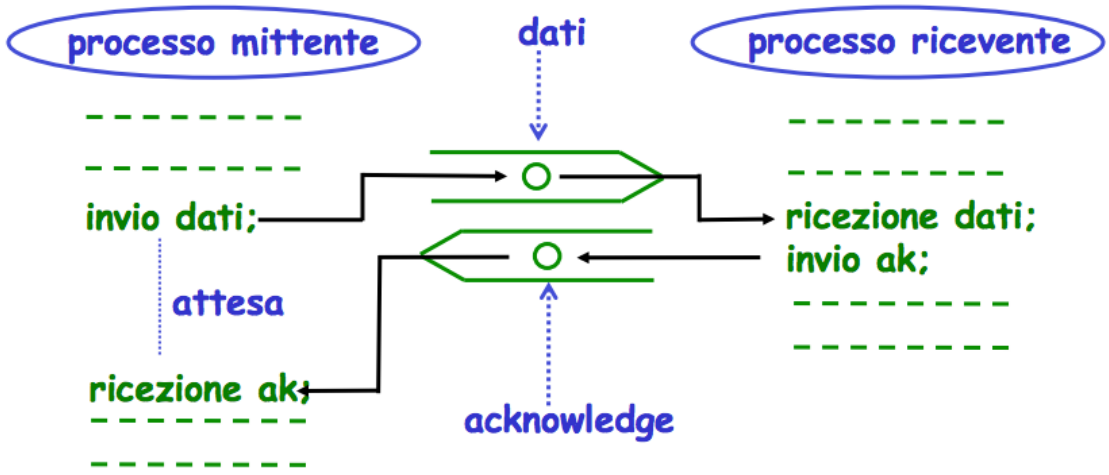
Comunicazione asincrona

- Il processo **continua la sua esecuzione** dopo che il messaggio è stato inviato.
- Comunicazione stateless (il ricevente non può avere informazioni sullo stato attuale del mittente)
- La **send non è un punto di sincronizzazione**
- Poco espressiva
- Molta più concorrenza
- Richiede un buffer di capacità illimitata

Comunicazione sincrona

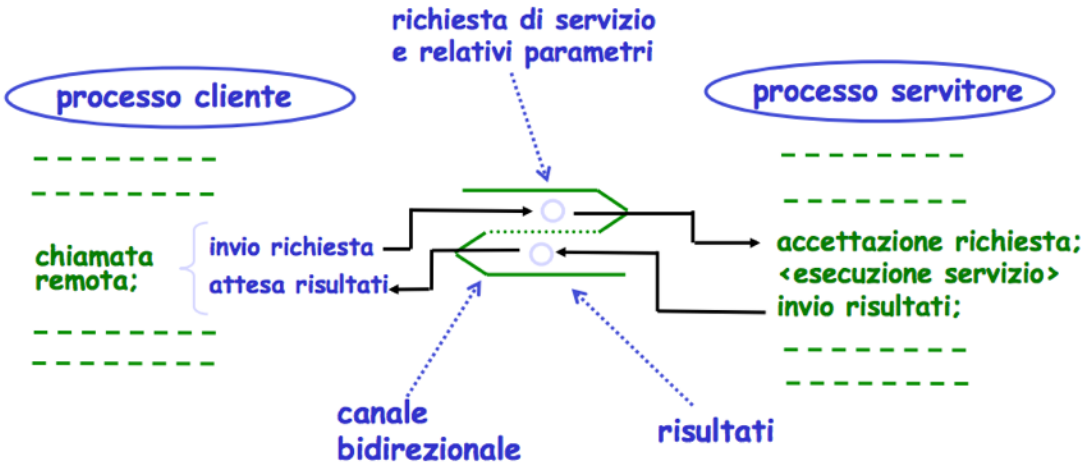
Detta ance **rendez-vous semplice**

- Il primo dei due processi comunicanti che esegue l'invio o la ricezione si **sospende** in attesa che l'altro sia pronto
- Non esiste la necessità di introdurre un buffer
- Un messaggio ricevuto contiene informazioni corrispondenti all **stato attuale** del processo mittente.



Comunicazione con sincronizzazione estesa

- Il processo mittente rimane in attesa finché il ricevente non ha terminato l'azione richiesta
- Analogia con procedura
- Modello client-server
- Riduzione del parallelismo



Primitive di comunicazione (sincrone)

Dichiarazione di canale:

<tipo_canale> <tipo> <id>

ch1 denota un canale di tipo ***port*** (asimmetrico da molti a uno) che trasporta interi.
Viene dichiarato nel processo ricevente ed è visibile agli altri processi tramite la **dot notation** (processo.canale)

Send

Primitiva di invio:

send(<valore>) to <porta>

- ***<porta>*** identifica il canale a cui inviare il messaggio
- ***<valore>*** espressione dello stipo di porta

Receive

Primitiva di ricezione:

receive(<variabile>) from <porta>

La primitiva **sospende il processo** se non ci sono messaggi sul canale e restituisce un valore del tipo predefinito **process** che identifica il nome del processo mittente.

proc = receive(m) from ch1

Il processo si sospende se non ci sono messaggi sul canale. Altrimenti
- estrae il messaggio e lo mette in **m**
- assegna il nome del mittente a **proc**

Si introduce il problema della scelta del canale su cui eseguire la receive. Possibilità di bloccarsi su un canale in presenza di messaggi su un altro —> priorità

Comando con guardia

<guardia> —> <istruzione>

<guardia> è costituita da una coppia: (***<espressione booleana>, <primitiva receive>)***
Una guardia viene valutata e può fornire 3 diversi valori:
- Guardia **fallita**: se l'espressione booleana è false
- Guarda **ritardata:** se l'espressione è true ma la receive è bloccante poichè non ci sono messaggi pronti sul canale
  - Il processo viene sospeso. Non appena arriva un messaggio viene riattivato e viene eseguita l'istruzione.
- Guarda **valida:** espressione vera e messaggi in attesa.

Espressione Vera e Recieve non bloccante (che vuol dire che c'è un messaggio in attesa)

Esistono delle varianti:
- **Comando con guardia alternativo:**)duce un if
- **Comando con guardia ripetitivo:** dentro ad un ciclo

Vengono valutate tutte le guardie di tutti i rami.
Si possono verificare 3 casi:
1. Se una o più guardie sono valide, viene selezionato, in modo non deterministico, uno dei rami con guardia valida.
2. se tutte le guardie non fallite sono ritardate, il comando con guardia alternativo attende che arrivi un messaggio che abilita la transizione

Primitive di comunicazione (asincro

Processi servitori

Simili ai **monitor** del modello a memoria comune. Possono mettere a disposizione una sola operazione con il solo vincolo della mutua esclusione.

Modello a memoria comune	Corrisponde	Modello a scambio di messaggi
risorsa condivisa: istanza di un monitor	—>	risorsa condivisa: struttura dati locale ad un processo server
identificatore di funzione di accesso monitor: entry	—>	porta del processo server
tipo dei parametri della funzione	—>	tipo della porta
tipo del valore restituito dalla funzione	—>	tipo della porta da cui il processo cliente riceve il risultato
per ogni funzione del monitor	—>	un ramo (**comando con guardia**) dell'istruzione ripetitiva che costituisce il corpo del server
condizione di sincronizzazione di una funzione	—>	espressione logica componente la guardia del ramo
chiamata di funzione	—>	invio della richiesta sulla corrispondente porta del server seguito da attesa dei risultati sulla propria porta
esecuzione di mutua esclusione fra le chiamate alle funzioni del monitor	—>	scelta di uno dei rami con guardia valida del comando ripetitivo del server
corpo della funzione	—>	istruzione del ramo corrispondente alla funzione

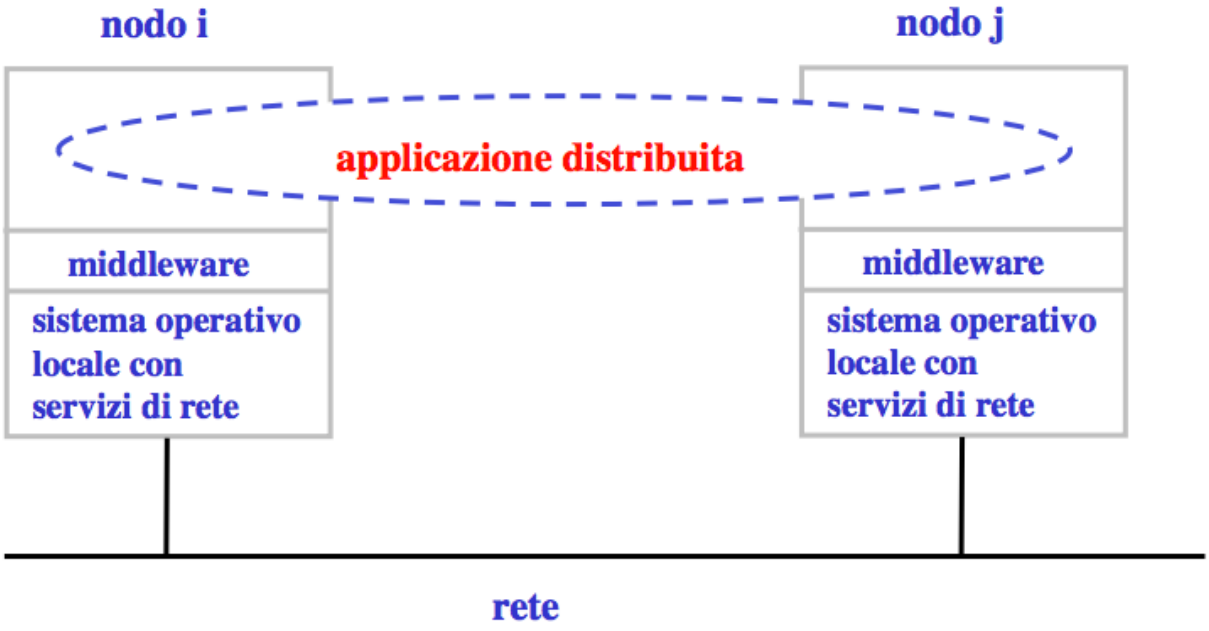
Realizzazione delle primitive asincrone

Le primitive sincrone possono essere tradotte dal compilatore del linguaggio in termini di primitive asincrone.

Architetture distribuite

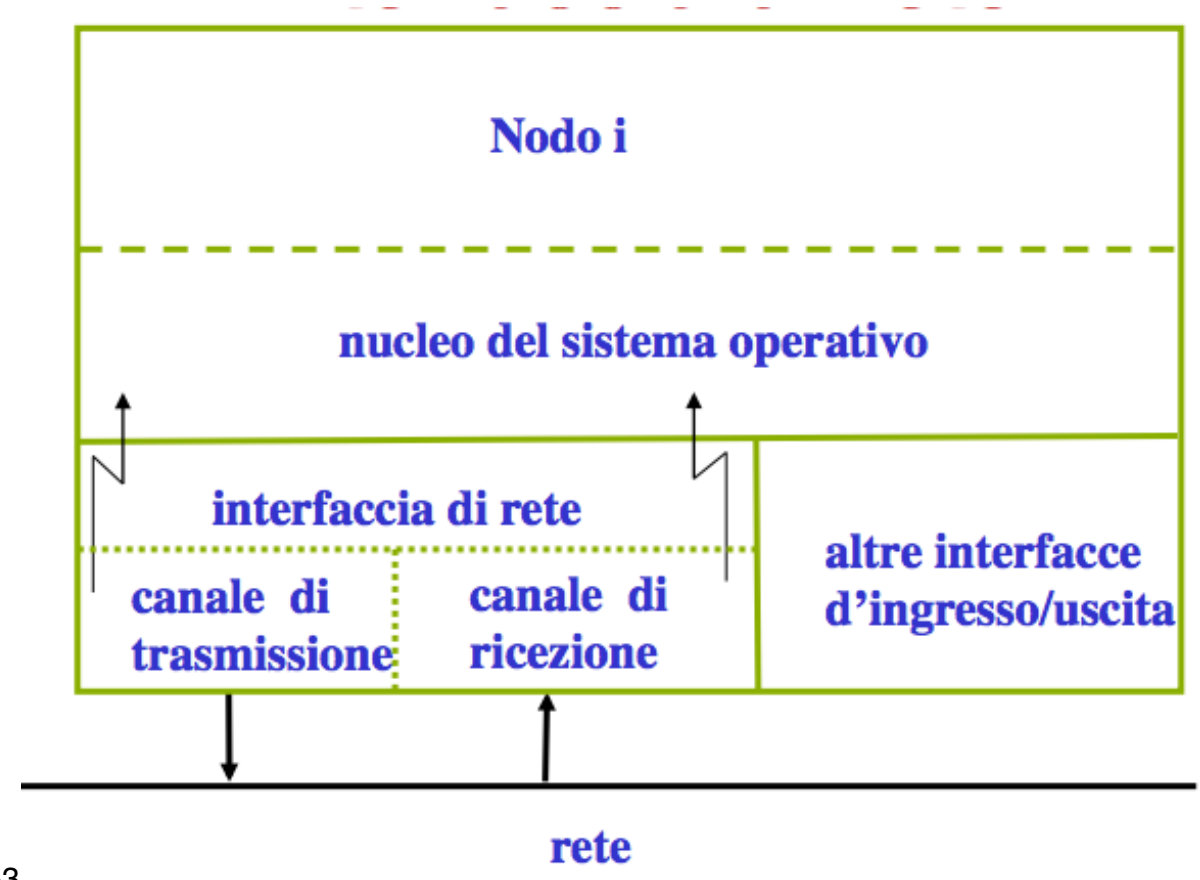
- Sistemi operativi distribuiti (**DOS - Distributed Operating Systems**)
  - Insieme di nodi tra loro omogenei e tutti dotati dello stesso sistema operativo
  - Scopo: gestire tutte le risorse nascondendo all'utente la loro distribuzione nella rete.
- Sistemi operativi di Rete (**NOS - Network Operating Systems**)
  - Insieme di nodi **eterogenei** con sistemi operativi **diversi** ed autonomi nodo per nodo
  - Tramite un **middleware** si omogeneizza i sistemi distribuiti

Sistemi NOS



Sistemi DOS

Interfaccia di rete



Pacchetti, interfacce, canali

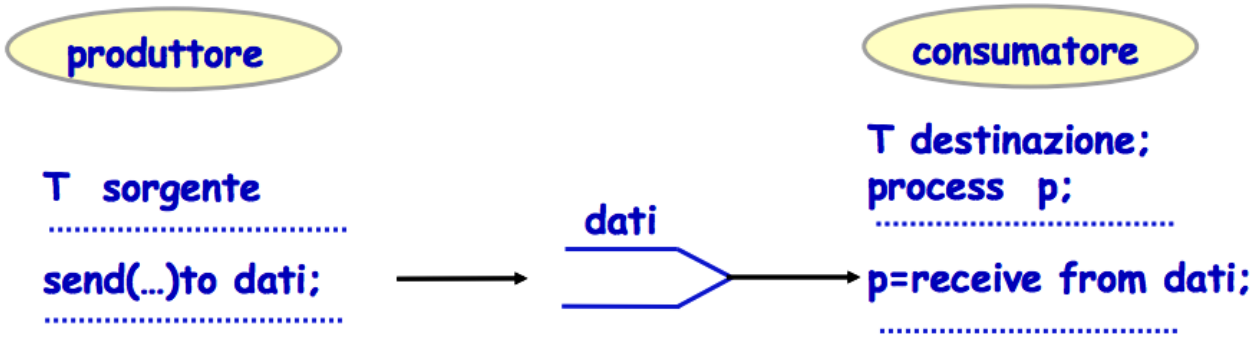
L'unità di trasmissione tra i nodi è il **pacchetto.** La struttura del pacchetto **dipende dalla realizzazione (protocolli di comunicazione)**. L'interfaccia di rete è strutturata in 2 parti (**canali**)

- uno per la **trasmissione dei messaggi:** viene acceduto per realizzare l'invio dei pacchetti. Vi sono associati
  - una **coda di pacchetti**, nella qual il sender deposita il proprio pacchetto se il canale è occupato da un altro processo
  - un **buffer** di pacchetti nel quale vengono depositati i pacchetti.
- uno per **ricezione dei messaggi** che conterrà
  - un **buffer** di pacchetti nel quale vengono depositati i pacchetti ricevuti

Partenza ed arrivo di pacchetti vengono notificati tramite **interruzioni**.

Primitive di comunicazione sincrone

- Minore grado di concorrenza rispetto alle primitive asincrone
- Non è più necessario un buffer illimitato.
  - Necessità di inserire una struttura interna al processo per bufferizzare i messaggi inviati



Comunicazione con sincronizzazione estesa

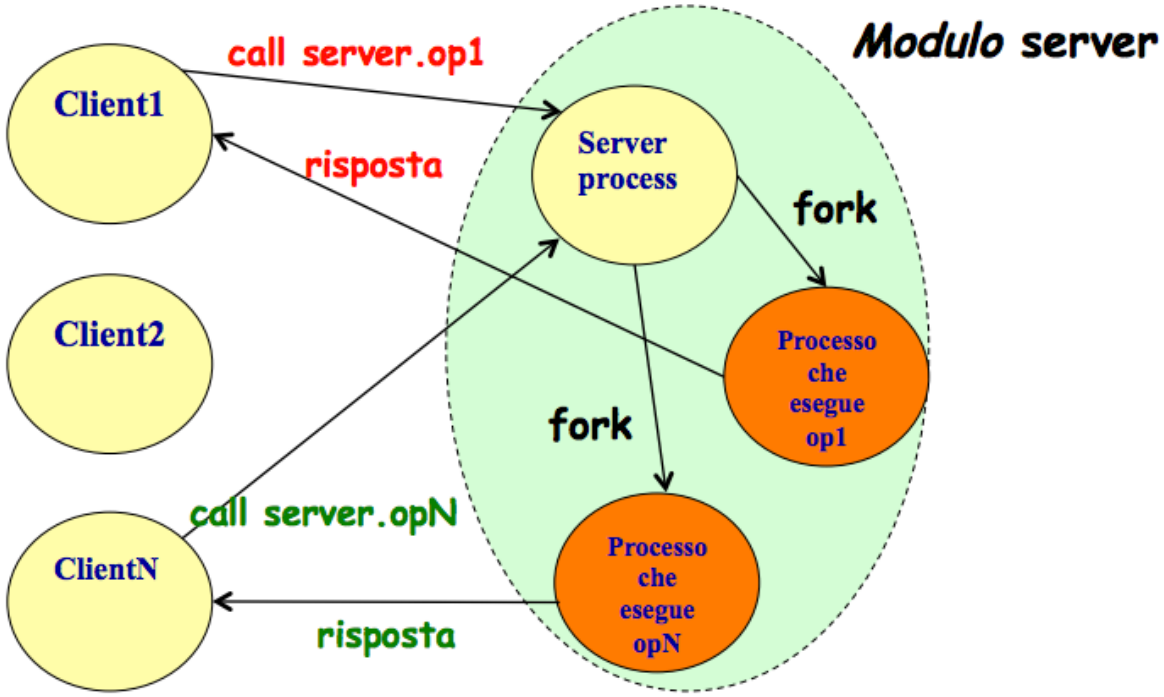
Meccanismo di **comunicazione** e **sincronizzazione** tra processi. Il processo che **richiede un servizio** rimane **sospeso** fino al completamento del servizio richiesto (sincrono)
I processi rimangono sincronizzati durante l'esecuzione del servizio da parte del ricevente fino ala ricezione dei risultati da parte del mittente (**rendez-vous esteso**)

Analogia semantica con una normale chiamata a funzione. Realizzabile in 2 modi:

- **RPC (Remote Procedure Call):**
- **Rendez-Vous**

Chiamate di procedura remota

Per ogni operazione che un processo client può richiedere viene dichiarata, lato server, una **procedura** e per ogni richiesta di operazione viene creato un nuovo **processo**(thread) servitore con il compito di eseguire la procedura corrispondente.

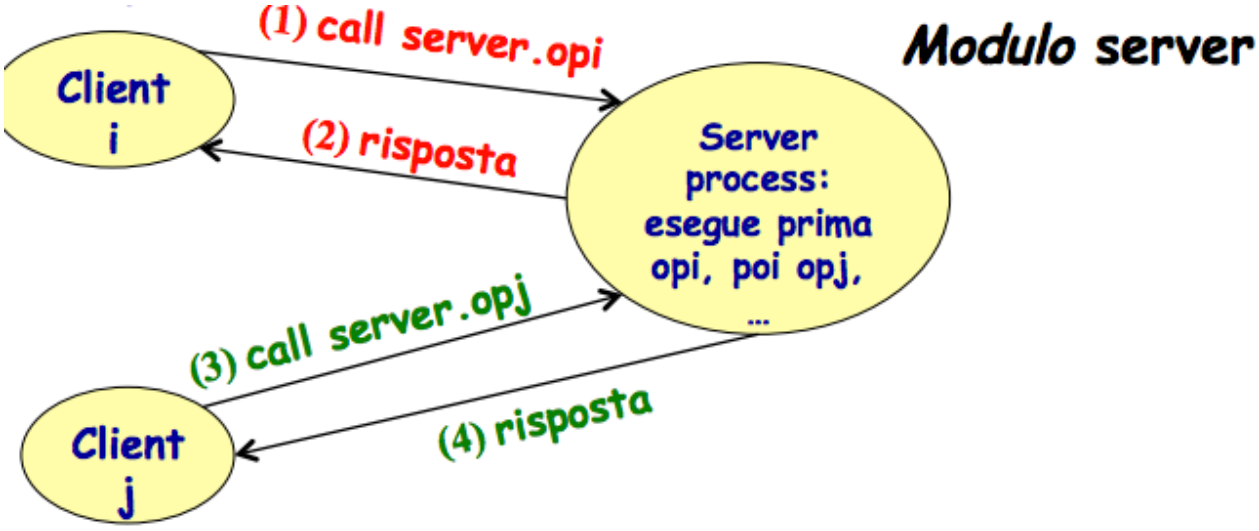


Rappresenta solo un **meccanismo di comunicazione tra processi.** Bisogna stare attenti che i vari processi servitori del server non operino concorrentemente in mondo errato —> **sincronizzazione**.

Rendez vous

L'operazione richiesta viene specificata come un **insieme di istruzioni** che può comparire in un punto qualunque del processo servitore (vedi ADA). Il processo servitore utilizza un istruzione di input (*accept*) che lo sospende in attesa di una richiesta. All'arrivo della richiesta il server esegue l'operazione e comunica i risultati al client.

Combina comunicazione con sincronizzazione. Esiste infatti un solo processo servitore!



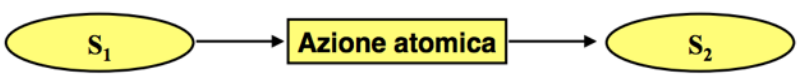
Linguaggio ADA

Sviluppato per conto del **DOD (Department of Defense)** degli Stati Uniti.

- Permette di definire applicazioni tradizionali ed in tempo reale.
- Adotta come **metodo d'interazione tra *task* (processi) di tipo simmetrico a rendezvous**
- Successiva introduzione dei ***protected type*** **(simili al monitor)** e l'istruzioni ***requeue*** per dare al programmatore controllo sulla sincronizzazione e sullo scheduling.
- Ogni ***task*** può definire delle operazioni pubbliche ***entry*** visibili da altri task.
- Il rendezvous viene stabilito quando un processo Q chiama un'operazione ***entry*** su P
- è un linguaggio **fortemente e staticamente tipato.**

Azioni Atomiche

- **Strumento di alto livello per la strutturazione di programmi concorrenti.**
- È un'astrazione realizzata con meccanismi linguistici
- Ogni oggetto astratto può trovarsi in stati **consistenti** o **inconsistenti** a seconda che si sia verificata o meno una particolare relazione (**invariante del tipo**) fra i valori delle variabili componenti l'oggetto
- Ogni tipo ha la sua relazione invariante che lo caratterizza dal punto di vista semantico
- Ogni azione deve essere programmata in modo da lasciare l'oggetto in uno stato **consistente** (cioè uno stato in cui l'invariante è soddisfatta)



Definizione

Operazione che porta un insieme di oggetti O = {o_1,o_2… o_n} da uno stato consistente S_1 ad uno stato consistente S_2.

Consistenza dei dati

Durante l'esecuzione di un'operazione atomica l'insieme O può passare attraverso stati inconsistenti, che, tuttavia, **non devono essere visibili** ad altre operazioni.

Esempio (Banca)
- **Oggetti**: conti correnti
- **Programmi**: operazioni di lettura, modifica che riguardano più oggetti

Ad esempio: **Operazione di trasferimento**: dati gli oggetti O1 e O2, spostare x da O1 a O2
L'invariante è **Valore(O1) + Valore(O2) = costante** (I soldi totali non devono essere cambiati)
Durante l'operazione si ha un momento in cui **x** è stata tolta da O1 ma non ancora sommata a O2; si ha quindi uno stato complessivo **inconsistente.** La mutua esclusione sui singoli oggetti garantisce l'atomicità delle operazioni su di essi, ma non dell'**intera operazione nel suo insieme.** È necessario che l'intero programma sia tratta atomicamente!

Possibili soluzioni al problema dell'inconsistenza dei dati

1. **nuova astrazione**: racchiude tutti gli oggetti in un monitor
   - **Limiti:**
     - Le operazioni possono **non essere note** al momento della definizione dell'astrazione
     - Non è possibile operare direttamente sugli oggetti.
2. Associare ad ogni oggetto in **gestore** ed obbligare i processi applicativi a **richiedere** l'uso dedicato di tutti gli oggetti e rilasciati solo che il nuovo stato **consistente** è stato raggiunto.

Proprietà Fondamentali

Per garantire la **consistenza dei dati** l'azione atomica deve possedere 2 proprietà fondamentali:
1. **Serializzabilità** (o atomicità nei confronti della concorrenza)
2. **Tutto o niente** (o atomicità nei confronti di eventi anomali)

Malfunzionamenti

È possibile che, se l'azione atomica, fallisce in uno dei suoi stati inconsistenti, ci si ritrovi a metà della sua esecuzione ed in uno stato inconsistente. È quindi necessario prevedere una forma di **fallback** (recupero) che riporti il sistema in uno stato **consistente.**

Tipi di malfunzionamento

- **disturbi temporanei**
  - **errori in lettura:** possono essere eliminati rileggendo le informazioni desiderate —> uso di **codici a ridondanza ciclica (CRC)**.
  - **errori in scrittura:** sono rilevabili leggendo le informazioni appena scritte e confrontandole con le originali —> se errato —> riscrivere informazioni.
- **guasti hardware**
  - **alterazioni delle informazioni:** a causa di disturbi o guasti hardware —> tecnica delle **copie multiple**. (spesso a livello 2)

Disco permamente: *astrazione* ottenuta eliminando ogni errore temporaneo. (utilizzando una **coppia di dischi permamenti**)

Proprietà "Tutto o niente"

Indicando con S' lo stato risultante dall'esecuzione dell'azione atomica, si **deve avere.**

S' = S1 (niente) oppure **S' = S2 (tutto)**

dove S1 e S2 sono rispettivamente lo stato iniziale e quello finale dell'azione.
La proprietà prende il nome di **tutto o niente.**

Cause

1. **Eccezione software** sollevata durante una delle operazioni sugli oggetti. Il processo esegue la primitiva **abort.**
2. **Malfunzionamento hardware** o di una condizione di **blocco critico** prima che le operazioni siano terminate. Il sistema di recupero da malfunzionamento **forza l'aborto dell'azione atomica.**

In entrambi i casi è necessario un **meccanismo di ripristino**

Meccanismo di ripristino

- necessita di informazioni relative allo stato corrente delle operazioni.
  - durante l'esecuzione di programmi (eccezioni) sono facilmente disponibili
  - in caso di malfunzionamenti hardware, si ha la totale perdita delle informazioni che risiedono in RAM —> le informazioni per il recupero vengono mantenute in **memoria di massa.**

Esiste la possibilità che le informazioni contenute nella memoria di massa risultino alterate a seguito di un malfunzionamento —> introdotto il concetto di **memoria stabile.**

Memoria stabile

Astrazione ottenuta attraverso l'uso della **ridondanza** delle informazioni.

- Le operazioni per leggere e scrivere su questa memoria sono **atomiche.**
  - **stable-read, stable-write**
  - un malfunzionamento durante la loro esecuzioni deve avere gli stessi effetti di un malfunzionamento avvenuto **prima** o **dopo** l'azione.
- Per specificare il tipo di azione che il Meccanismo di ripristino deve eseguire dopo un crash si introducono gli **stati di un'azione atomica.**

Realizzazione della memoria stabile

La memoria ha stabile ha due proprietà fondamentali:

- **non** è soggetta a malfunzionamenti
  - Ottenuto tramite **ridondanza delle informazioni**
- Le informazioni in essa residenti non vengono perdute a causa della caduta dell'elaboratore
  - Operazione di lettura e scrittura **atomiche —> stable-read, stable-write (**godono delle proprietà del "tutto o niente")

***Per ipotesi si suppone che le due copie non vengano alterate contemporaneamente dallo stesso malfunzionamento* —> in questo modo una delle due copie è sempre valida**

È realizzata come un **monitor**.

Stati di un'azione atomica

- **Working:** Quando il processo è in questo stato gli oggetti sono *inconsistenti*. —> **abortire** azione atomica
- **Committing:** Durante la terminazione corretta dell'azione. Gli oggetti sono al loro *stato finale.* —> **completare** l'azione
- **Aborting:** Durante la terminazione anomala dell'azione atomica. Gli oggetti devono essere **ripristinati**.

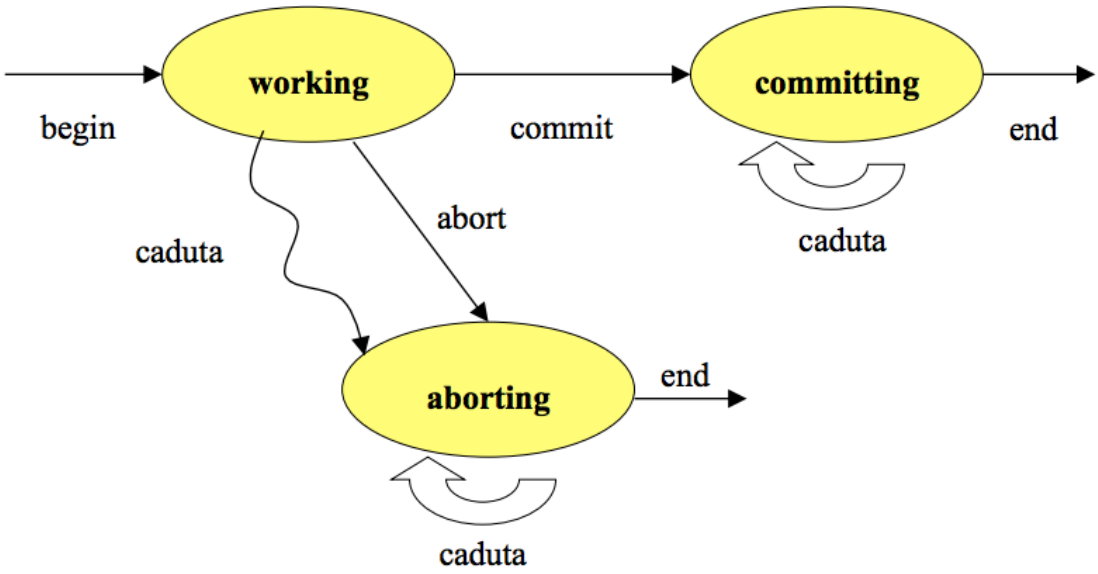
caso 1) Working —> Aborting —> Stato iniziale
caso 2) Committing —> Stato finale

Le informazioni sullo stato in cui si trova il processo sono contenute in una struttura dati (**descrittore dell'azione**) allocata in memoria stabile.

- Primitiva **begin action**: crea in memoria stabile un descrittore d'azione inizilizzato a **working**
- Primitive **abort** e **commit:** commutano lo stato ad **aborting** o **committing.**
- Durante la riattivazione dell'elaboratore, il **meccanismo di recupero** può desumere lo stato dei processi dalla memoria stabile e aggiustare le cose.

Ovviamente le azioni sugli oggetti (che risiedono in memoria stabile) sono fatte su **copie** fatte su **memoria volatile.** In questa situazione:

- **Aborto dell'azione —> eliminazione delle copie**
- **Terminazione dell'azione —> salvataggio delle copie in memoria stabile**



Tuttavia la copia viene fatta con una sequenza di azioni *stable-write.* se l'elaboratore cade a metà sequenza si ha uno stato **inconsistente —>** Si copiano le **intenzioni** in memoria stabile **prima del commit (senza modificare le copie originali).** In seguito, **solo se la copia delle intenzioni** è stata correttamente memorizzata, viene sovrascritta la copia originale.
A questo punto:

- se l'elaboratore cade prima del commit —> azione abortita, copia delle intenzioni distrutta, copia originale salva
- se l'elaboratore cade nella seconda fase —> fase finale corrotta, copia finale finale corrotta ma valida la copia delle intenzione in memoria stabile. Si recupera da lì. (fase committing)

Realizzazione di azioni atomiche

Serializzabilità

Assicura che ogni azione atomica operi sempre su un insieme di oggetti il cui stato iniziale è consistente ed i cui stati parziali, durante l'esecuzione, non sono visibili ad altre azioni concorrenti.

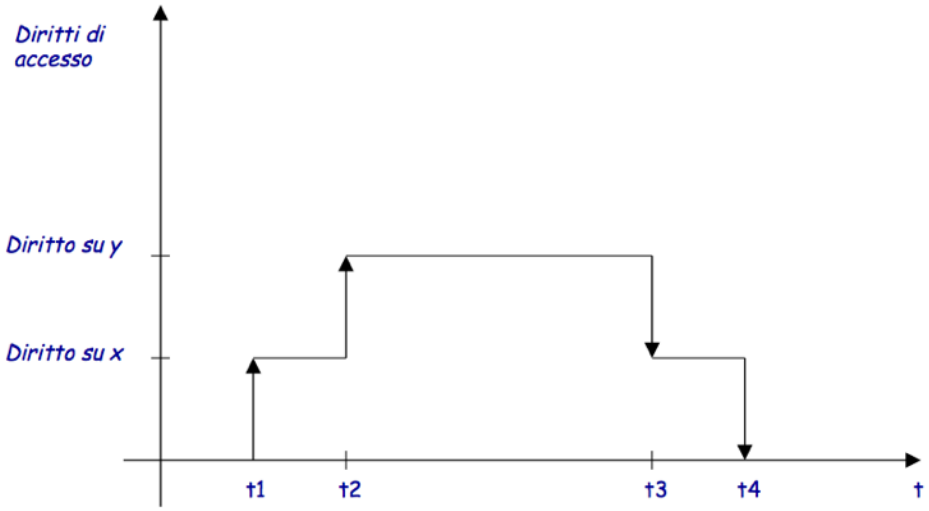
Implementata tramite **two phase protocol:**

- sia A un'azione atomica che opera su un insieme di oggetti O = {O1, O2,…On}
- Siano **Richiesta(Oi)** e **Rilascio(Oi)** le operazioni per richiedere al gestore di ogni oggetto l'uso esclusivo dell'oggetto.
- La serializzabilità viene garantita allocando dinamicamente singoli oggetti in **modo dedicato** alle azioni atomiche con il **seguente protocollo**:
  A. Ogni oggetto deve essere **acquisito** da un'azione atomica A in modo esclusivo prima di qualunque azione su di esso: —> **Richiesta(Oi) è bloccante se l'oggetto Oi non è disponibile.**
  B. Nessun oggetto deve essere **rilasciato** prima che siano eseguite tutte le operazioni su di esso

C. Nessun oggetto può essere **richiesto** dopo che è stato effettuato un **rilascio** di un altro oggetto.

Two Phase Lock Protocol

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



Tutto o niente

Va aggiunto un ulteriore requisito:
D. Nessun oggetto può essere rilasciato prima che l'azione atomica abbia completato la sua esecuzione. In altre parole, i rilasci devono costituire l'ultima parte dell'azione atomica.

Viene quindi introdotta l'operazione primitiva ***commit.***
Il meccanismo di recupero si dovrà comportare in maniera **diversa** a seconda dell'istante in cui l'evento anomalo si verifica. In particolare dovrà:

- **abortire l'azione** se il malfunzionamento avviene quando gli oggetti sono in stato inconsistente (effetto *niente*)
- **garantire il completamento** dell'azione quando gli oggetti sono nel nuovo stato S2 (effetto *tutto*)

Tutto o niente nella stable-write:
Se si verifica un guasto durante la scrittura su memoria stabile

- Sul primo blocco —> viene alterato ma il secondo è intatto (effetto *niente*)
- Sul secondo blocco —> rimane valido il primo (effetto *tutto*)
- Tra le due operazioni: risultato **non consistente.** Il meccanismo di recupero legge i due blocchi:
  - Se uno è alterato viene sostituito con l'altro

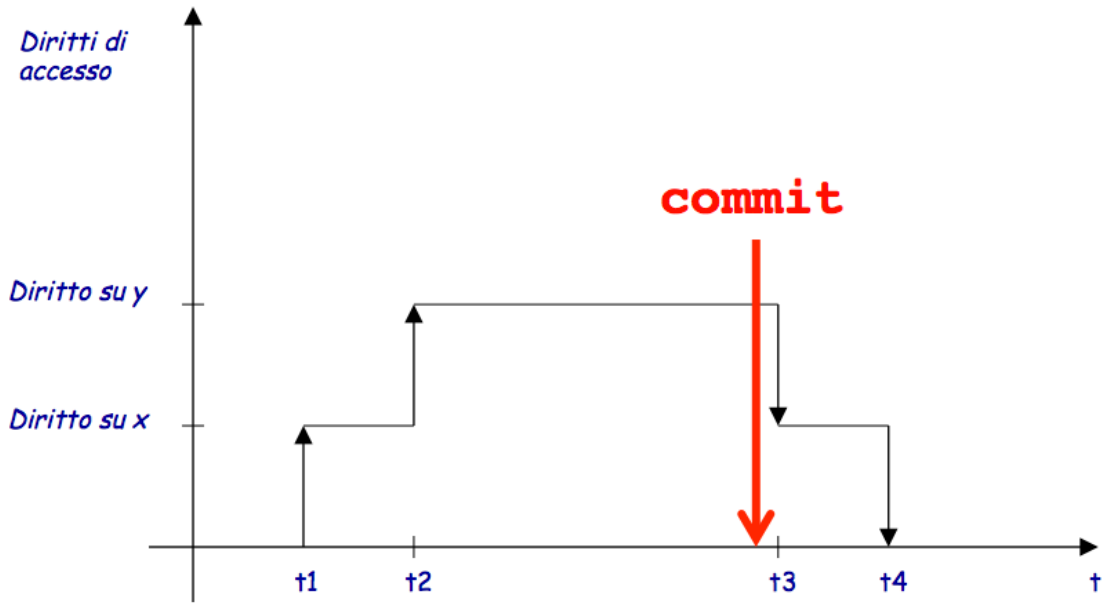
- Se entrambi sono corretti, ma con valori diversi, vuol dire che c'è stato un guasto tra le due scritture. —> Si copia il primo blocco nel secondo

Commit

Riformulando il requisito D in "*Ogni azione atomica deve rilasciare i propri oggetti* ***dopo*** *l'operazione di completamento con successo"* ecco un esempio:

```
A1 :   {
         Richiesta(X);
         Richiesta(Y);
         X:= X+20;
         Y:= Y+20;
         commit;
         Rilascio(X);
         Rilascio(Y);
       }
```

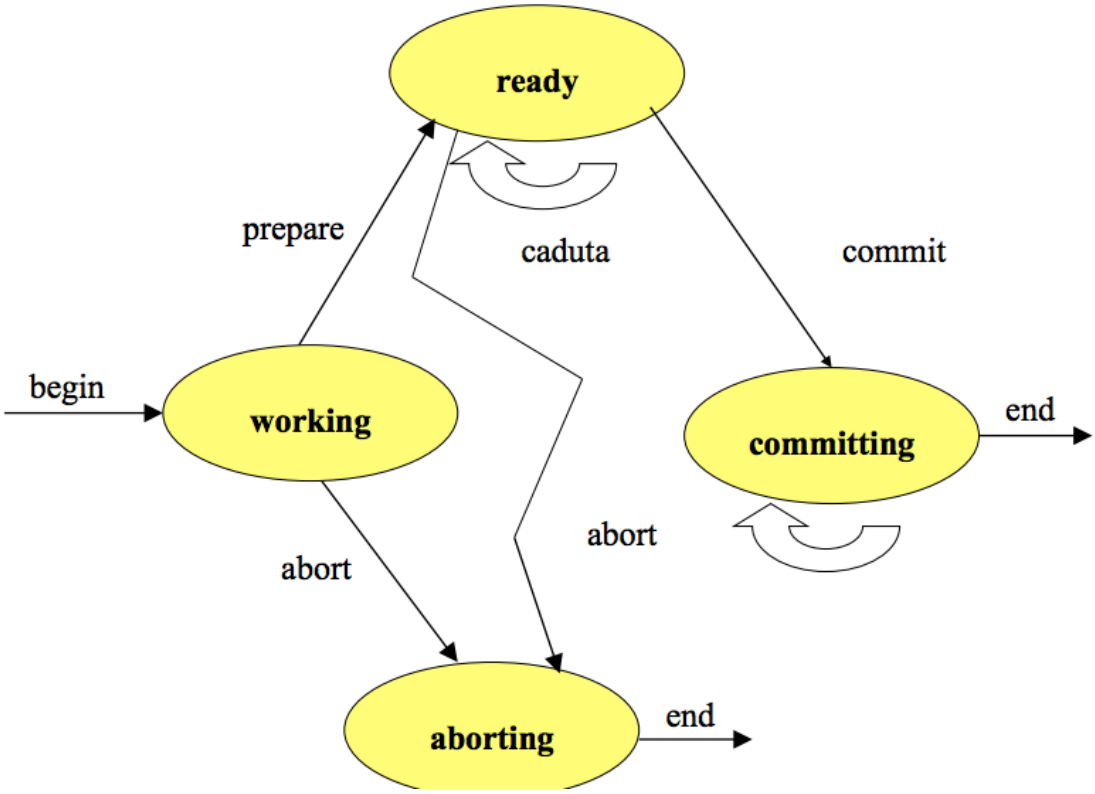
```
A2 :   {
         Richiesta(X);
         Richiesta(Y);
         X:= X*10;
         Y:= Y*10;
         commit;
         Rilascio(X);
         Rilascio(Y);
       }
```



Azioni atomiche multiprocesso

In molti casi le azioni atomiche sono eseguite da più processi anche su nodi diversi(vedi sistemi distribuiti)

In questo modo per ipotesi possiamo considerare che i due processi operino su oggetti diversi —> no blocchi critici.
Se ogni processo esegue il **two phase lock protocol,** è assicurata la **serializzabilità.** Per soddisfare la proprietà del **tutto o niente** è necessario che tutti i processi completino le loro operazioni con lo stesso risultato: ***successo o abort.***
Un comportamento difforme dei processi porterebbe ad uno stato **inconsistente.**
È necessario introdurre un nuovo stato: **ready.** Caratterizza un processo che ha terminato le sue operazioni ed è pronto per il **commit**



La transizione da working a ready si ha quando il processo esegue la primitiva **prepare** (con essa il processo **perde il diritto di abortire per conto suo**). Il protocollo che ciascun processo esegue per negoziare il tipo di completamento prende il nome di **two phase commit protocol:**

Two phase commit protocol

- **Fase 1:** ciascun processo specifica la propria opzione
- **Fase 2:** viene verificata l'opzione degli altri:
  - se tutti hanno optato per la terminazione con successo —> tutti transitano allo stato ***committing***.
  - se almeno uno ha optato per abortire —> tutti transitano allo stato ***aborting.***

Se cade l'elaboratore mentre un processo è in stato *ready,* il meccanismo di recupero deve ripristinare per tale processo lo stato *ready.* Infatti il processo non può essere riattivato in fase *aborting,* poiché con *prepare,* **ha perduto il diritto di abortire unilateralmente.**
Il **descrittore di azione atomica** in memoria stabile, dovrà tenere aggiornato lo stato di tutti i processi partecipanti.

Realizzazione

Cambia a seconda del ***modello di interazione*** tra i processi:

- **Modello a memoria comune:**
  - Descrittore azione = **monitor**
  - L'azione atomica viene iniziata da un processo che ne crea in memoria stabile il descrittore e attiva in **parallelo** i processi che la eseguono
  - Quanto tutti i processi hanno terminato, il processo iniziale riprende il controllo e termina l'azione cancellando il descrittore dalla memoria stabile.
  - **Il processo i-esimo**:
    - Richiede esclusivamente l'accesso agli oggetti
    - Crea le copie volatili degli oggetti
    - Esegue la sequenza di operazioni sulle copie volatili
    - *Terminazione***:**
      - Con **aborto:**
        - entra nello stato *aborting*
        - lo stato viene riportato sul descrittore
        - vengono risvegliati i processi nello stato *ready.*
      - Con **successo:**
        - Crea la copia delle intenzioni in memoria stabile
        - transita allo stato *ready* attraverso l'esecuzione di *prepare;* viene modificato il suo stato da *working,* a *ready* nel descrittore dell'azione atomica
        - Verifica se almeno un processo ha abortito
          - *in caso affermativo —>* ***distrugge la copia delle intenzione ed esegue abort.***
          - *in caso negativo —>* due casi:
            - Qualche altro processo è ancora in stato di working:
              - il processo resta in stato di *ready* e si blocca.
            - Tutti i processi sono in stato di *ready:*
              - il processo entra nello stato *committing,* trasferisce gli oggetti dalla copia delle intenzioni a quella originale e distrugge la copia delle intenzioni.
              - Risveglia un altro processo che si era bloccato che risveglierà a catena gli altri processi dopo aver effettuato il commit.
- **Modello a scambio di messaggi:**
  - Descrittore azione = **oggetto privato di un processo detto *coordinatore* dell'azione atomica.**
    - 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.

- **Two phase commit protocol:**
  - Quando è pronto a terminare con successo l'azione atomica
    - copia le intenzioni in memoria stabile
    - esegue la *prepare* per entrare nello stato di *ready*
    - Invia ad ogni partecipante un messaggio di richiesta esito e rimane in attesa delle risposte
    - Se arrivano una o più risposte di **esito negativo:**
      - L'azione deve essere abortita
      - viene eseguita la primitiva ***abort***
      - viene inviato un messaggio a tutti partecipanti specificando di **completare con aborto**.
      - il coordinatore termina con aborto
    - Se tutti i partecipanti danno **esito positivo**:
      - 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** a parte di tutti i partecipanti
      - All'arrivo di tutti i messaggi, il coordinatore termina distruggendo il descrittore
- **Il partecipante i-esimo**:
  - Terminate le operazioni sugli oggetti, il processo può
    - aver abortito (*stato aborting*)
    - aver creato la copia delle intenzioni (*stato ready*)
  - in entrambi i casi attende il messaggio ***richiesta-esito*** da parte del coordinatore
  - Risponde con esito **positivo** o **negativo** a seconda dello stato in cui si trova:
    - Se è in stato di *aborting* termina abortendo, altrimenti resta in attesa del messaggio di completamento.
  - Se viene ricevuto il messaggio di ***completare con aborto***
    - viene eliminata dalla memoria stabile la copia delle intenzioni ed il processo termina abortendo
  - Se viene ricevuto il messaggio di ***completare con successo***
    - il processo termina correttamente trasferendo i valore della copia delle intenzioni in memoria stabile.
    - Invia al coordinatore il il messaggio: ***avvenuto completamento***

Azioni atomiche multiprocesso e sistemi distribuiti

Nei sistemi **distribuiti** si possono avere due tipi di malfunzionamenti:

- Caduta dei singoli nodi della rete
  - ogni processo partecipante deve mantenere in memoria stabile delle **variabili di stato** da utilizzare in fase di riattivazione
- Perdita dei messaggi in rete
  - Utilizzo di un **temporizzatore** per limitare il tempo di attesa di un messaggio. Al termine del tempo previsto, il processo viene riattivo e viene eseguita una procedere per rispedire il messaggio

Essi comportano il **non arrivo** a destinazione di alcuni messaggi.

Azioni atomiche nidificate

Per la loro natura le azioni atomiche **non possono essere nidificate**

Problemi

Si avrebbe infatti una **violazione di:**

- **serializzabilità:** una volta terminata l'azione interna i suoi oggetti vengono rilasciati e possono essere richiesti da altri processi (azione atomica principale non ancora terminata!!)
- **tutto o niente:** se l'azione atomica principale abortisce subito dopo la terminazione dell'azione interna, non può ripristinare lo stato degli oggetti modificati dall'azione interna in quanto i loro valori originali sono già stati sostituiti.

Soluzione

Modifica dei protocolli **two phase lock protocol e two phase commit protocol:**

- Una sottoazione che termina con successo rende disponibili gli oggetti **solo** all'azione atomica superiore. In questo modo si nasconde la visibilità degli stati intermedi.
- Quando una sottoazione termina con successo, la seconda fase di commit viene ritardata ed eseguita solo se l'azione più esterna termina con successo. —> modifiche in memoria stabile effettuate solo alla fine.

Per garantire questo comportamento per ogni oggetto viene mantenuta aggiornata una **pila di copie di lavoro**.

- All'inizio di una sottrazione viene generata una **nuova copia** in testa alla pila.
- Se l'azione interna termina con **successo**, la copia in testa alla pila diventa la nuova copia di lavoro per l'azione più esterna; diversamente viene **scartata la copia in testa alla pila**

Chiamata di procedura remota in sistemi distribuiti

RPC rappresenta il miglior sistema per la strutturazione di **sistemi transazionali distribuiti** (client-server)

- RPC temporizzate.
- La **semantica** delle RPC dipende dalle azioni intraprese allo scadere dell'intervallo di tempo. Due possibilità:
  - **at most once:** In presenza di malfunzionamenti, la RPC viene **abortita senza produrre effetti.** (gestita come un'azione atomica)
  - **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.**
    - comporta la possibilità di **esecuzioni multiple** della stessa procedura
      - Occorre che le procedure siano **idempotenti** (Ogni loro ripetuta esecuzione producevo stesso risultato); oppure si può ricorrere a particolari algoritmi di riconoscimento

Azioni atomiche e Transazioni

In un sistema distribuito le azioni atomiche vengono utilizzate per strutturare applicazioni **transazionali**.

Transazione

Sequenza di operazioni effettuate su database che fanno passare il sistema da uno stato all'altro, ambedue consistenti.

Deve rispettare 4 proprietà (**ACID**):

1. **Atomicità**: transazione come **entità indivisibile**
2. **Consistenza:** evoluzione del database da uno stato corretto all'altro
3. **Isolamento:** informazioni **protette** durante l'esecuzione delle transazioni (serializzabilità)
4. **Durata:** una volta terminata la transazione, i suoi effetti sul database devono avere effetti **duraturi** (possono essere alterati soltanto con **altre transazioni e non da malfunzionamenti** del sistema)