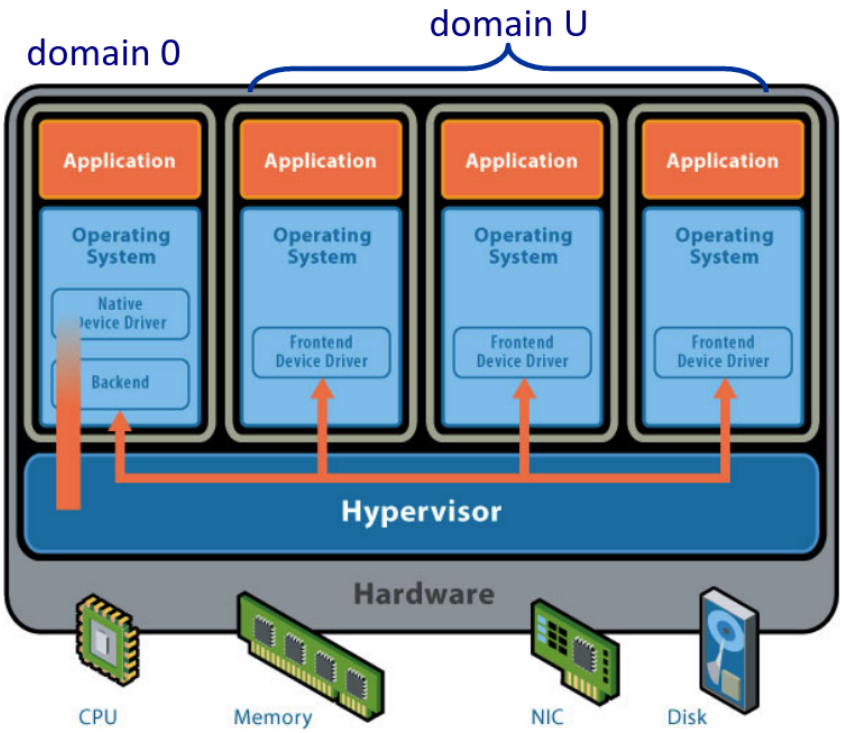
Indice

1 — xen

VMM nato come opensoruce paravirtualizzato.

Figura 1.1: Architettura xen



Xen è un VMM di sistema, ovvero si appoggia direttamente sull'hardware e quindi può eseguire direttamente chiamate system call che necessita del rin 0.

Il vmm si occupa della virtualizzazione della CPU, della memoria e dei dispositivi per ogni macchina virtuale, qui definiti domain.

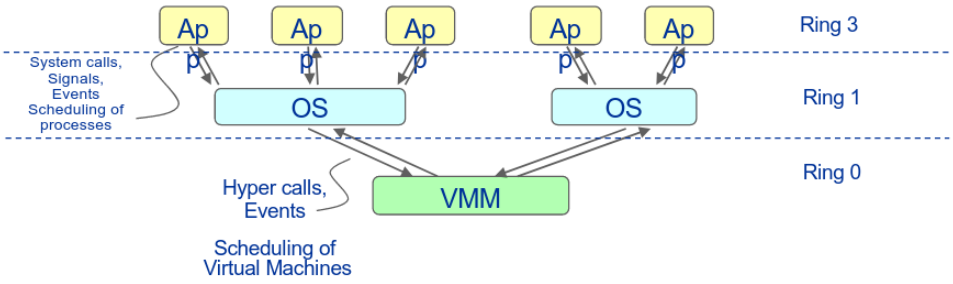
Xen offre una interfaccia di controllo in grado di gestire la divisione delle risorse tra i vari domini. L'accesso a questa interfaccia è consentito soltanto da una speciale VM, la domain 0.

1.1 Caratteristiche

Data la natura **paravirtualizzata** delle VM gestite da xen, le VM possono eseguire direttamente system calls che vengono delegate al VMM tramite hypercalls.

Per quanto riguarda la **protezione**, i guest OS sono collocati nel ring 1.

Figura 1.2: Protezione e hypercalls guest OS



1.2 Gestione memoria e paginazione

Gestione della memoria

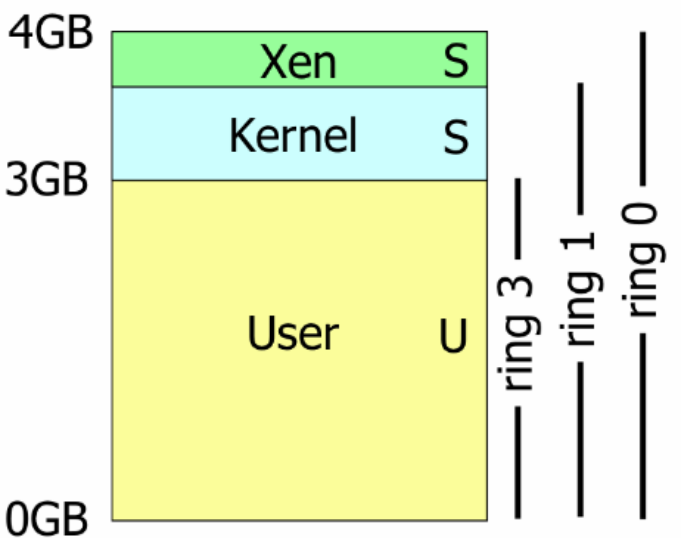
Gli OS guest gestioscono la memoria virtuale mediante i meccanismi e le politiche tradizionali.

La soluzione adottata si basa sulle tabelle delle pagine delle VM:
Vengono mappate nella memoria fisica dal VMM (**shadow page tables**, possono essere accedute in scrittura soltanto dal VMM stesso ma sono disponibili in modalità read-only ai guest.

In caso di necessità di update, il VMM valuta la richiesta e la esegue.

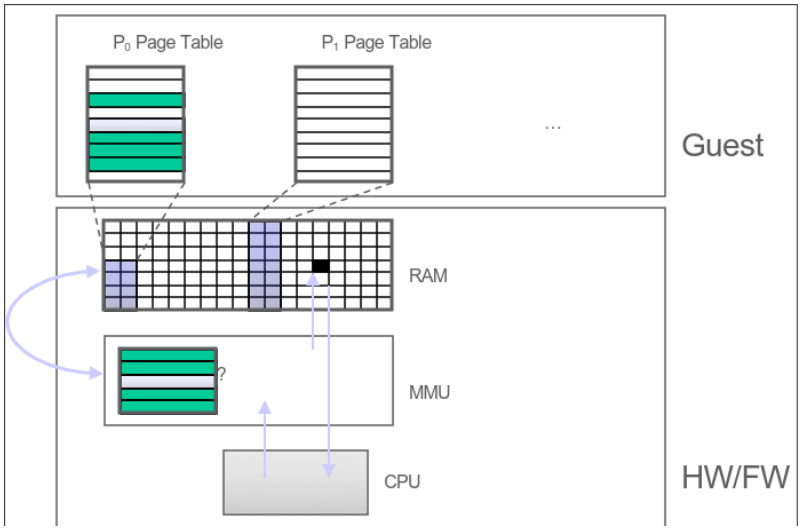
Per alleggerire l'onere del procedimento di update, viene implementato il **memory split**, per permettere una maggiore efficienza delle hypercalls: xen risiede nei primi 64MB del virtual address space.

Figura 1.3: Struttura virtual address space- memory split



I guest OS si occupano della paginazione, delegando al VMM la scrittura delle page table entries, una volta create sono disponibili in read-only per il guest che le ha richieste.

Figura 1.4: Creazione pagine dal VMM



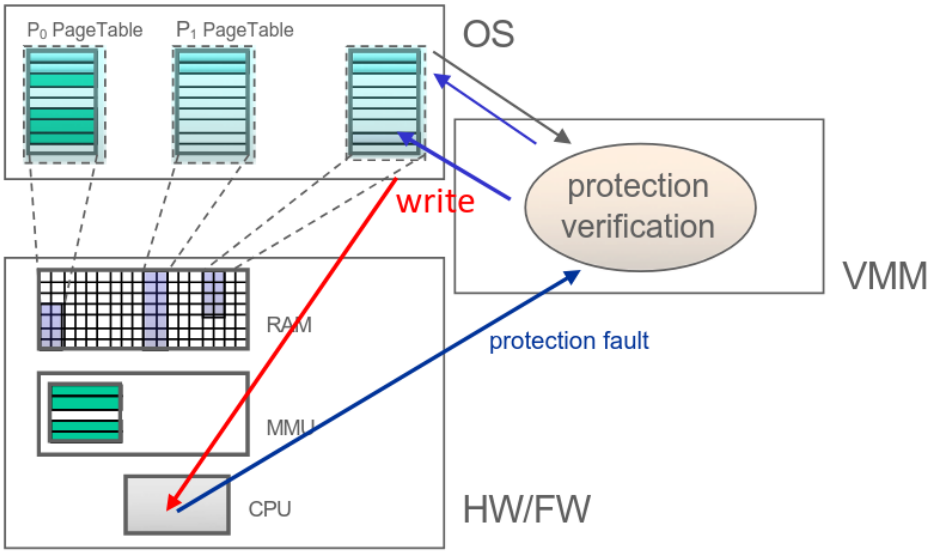
Creazione di un processo

Il SO richiede una nuova tabella delle pagine al VMM:

- aggiunte alla tabella le pagine appartenenti al segmento di xen
- xen registra la nuova tabella e acquisisce il diritto di scrittura esclusiva

- ogni successiva update da parte del guest provoca un protection-fault, comporta la verifica e l'aggiornamento della PT

Figura 1.5: Creazione di un processo



Balloon process

Dato che la paginazione è a carico dei guest, il VMM necessita di avere un meccanismo per reclamare da altre macchine virtuali pagine di memoria meno utilizzate. Su ogni macchina virtuale è in esecuzione un **ballon process** che, in caso di necessità, si "gonfia" per ottenere altre pagine che poi cede al VMM.

1.3 Virtualizzazione della CPU

Il VMM definisce una archiettura simile a quella del processore, con istruzioni privilegiate sostituite da opportune hypercalls.

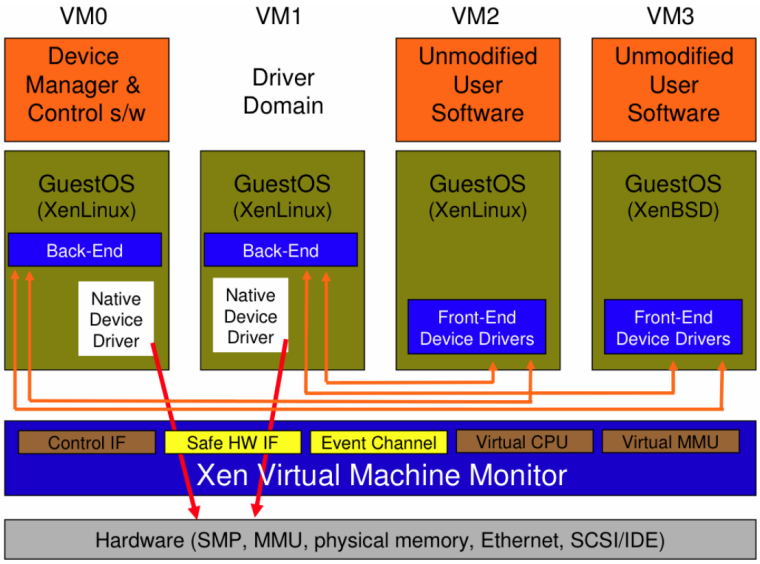
Il VMM si occupa dello scheduling delle macchine virtuali: **Borrowed Virtual Time** scheduling algorithm:

- si basa sulla nozione di virtual-time
- algoritmo general-purpose, consente di ottenre schedulazioni efficienti in caso di vincoli stringenti

Esistono due clock:

- real-time, inizia al boot
- virtual-time, associato a VM, avanza solo quando la VM esegue

Figura 1.6: Virtualizzazione dell'I/O

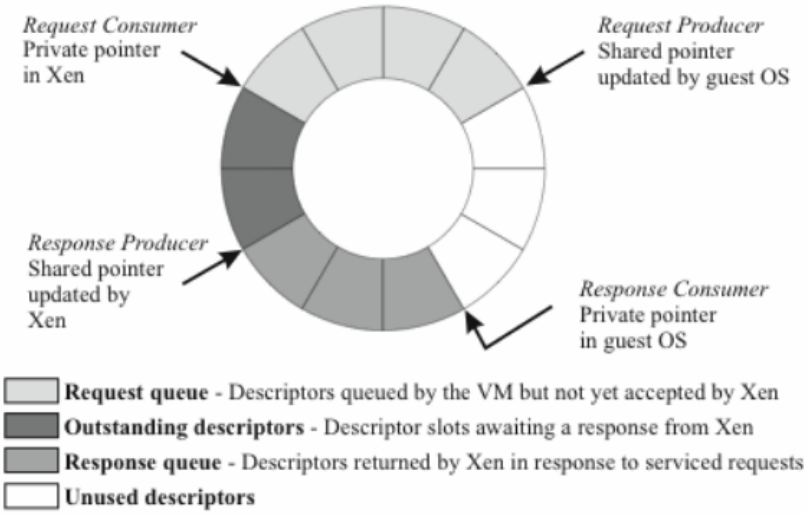


Esiste un **back-end driver** per ogni dispositivo, il suo driver è isolato all'interno di una particolare macchine virtuale (tipicamente dom0), ha accesso diretto all'hardware.

Ogni guest prevede un **front-end driver** virtuale semplificato che consente l'accesso al device tramite il back-end.

Questa tecnica comporta una semplificazione della portabilità a scapito della necessità di comunicaazione don il back-end attraverso degli asynchronous I/O rings.

Figura 1.7: I/O rings



1.4 Gestione interruzioni e eccezioni

La gestione delle interruzione viene virtualizzata in modo da lasciare a ogni guest la gestione delle interruzioni, il vettore punta direttamente alle routine del kernel guest.

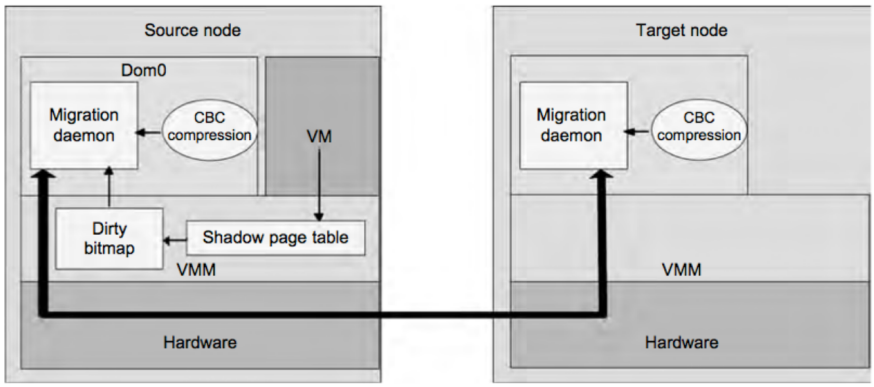
Il page-fault è un caso particolare, nel quale è necessario l'intervento del VMM, in quanto l'indirizzo che ha provocato il page-fault è contenuto nel registro CR2, inaccessibile al guest: il guest punta a codice xen che esegue una copia del CR2 all'interno di una variabile nello spazio guest.

1.5 Live migration in xen

La migrazione è **guest-based**: il comando di migrazione viene eseguite da un demone di migrazione nel domain0 del server di origine della macchine da migrare.

La realizzazione si basa sulla modalità pre-copy, le pagine da migrare vengono compresse per ridurre l'occupazione di banda.

Figura 1.8: Migrazione live



2 — Protezione

Sicurezza

Insieme delle tecniche per regolamentare l'accesso degli utenti al sistema di elaborazione. La sicurezza impedisce accessi non autorizzati al sistema e i conseguenti tentativi dolosi di alterazione e distruzione dei dati.

Meccanismi per **identificazione**, **autenticazione** e **autorizzazione** di utenti "fidati".

Protezione

Insieme di attività volte a garantire il controllo dell'acecsso alle risorse logiche e fisiche da parte degli utenti autorizzati all'uso di un sistema di calcolo.

Definizione, per ogni utente autorizzato, di:

- quali **risorse** sono accessibili
- quali **operazioni** può effettuare

Sono stabilite tramite tecniche di controllo degli accessi.

2.1 Protezione

In un sistema, il controllo degli accessi si esprime tramite la definizione di tre livelli concettuali:

- modelli
- politiche
- meccanismi

2.1.1 Modelli

Un modello di protezione definisce **soggetti**, **oggetti** ai quali i soggetti possono accedere e i **diritti** di accesso:

- oggetti, risorse fisiche e logiche alle quali applicare limitazione di accesso;
- soggetti, entità che possono richiedere l'accesso agli oggetti, utenti e processi;
- diritti di accesso, operazioni con le quali è possibile operare sugli oggetti;

Dominio di protezione

Ad ogni soggetto è associato un **dominio** che rappresenta l'ambiente di protezione nel quale il soggetto esegue, il dominio specifica i diritti di accesso posseduti dal soggetto nei confronti di ogni risorsa.

Un dominio di protezione è **unico per ogni soggetto**, mentre un processo può eventualmente cambiare dominio durante la sua esecuzione.

2.1.2 Politiche

Le **politiche di protezione** definiscono le regole con le quali i soggetti possono accedere agli oggetti.

Classificazione delle politiche:

- **Discetional Access Control** (DAC): il creatore di un oggetto controlla i diritti di accesso per quell'oggett (UNIX), definzione delle politiche decentralizzata.
- **Mandatory Access Control** (MAC): i diritti di accesso vengono definiti in modo centralizzato. Installazione ad alta sicurezza (es. enti gonvernativi)
- **Role Based Access Control** (RABC): un ruolo ha specifici diritti di accesso alle risorse, gli utenti possono appartenere a diversi ruoli, i diritti sono assegnati in modo centralizzato.

Principio del privilegio minimo

Ad ogni soggetto sono garantiti i diritti di accesso solo agli oggetti strettamente necessari per la sua esecuzione, è una caratteristica desiderabile in tutte le politiche di protezione.

2.1.3 Meccanismi

I meccanismi di protezione sono gli strumenti messi a disposizione dal sistema di protezione per imporre una determinata politica.

Principi di realizzazione

- **Flessibilità** del sistema di protezione, i meccanismi di protezione devono essere sufficientemente generali per consentire l'applicazione di diverse politiche;

- **Separazione** tra meccanismi e politiche, la politica definisce il *cosa* va fatto e il meccanismo il *come* va fatto.

In UNIX si utilizza la politica DAC e il SO offre un meccanismo per definire e interpretare i tre bit dei permessi.

2.1.4 Dominio di protezione

Un dominio definisce una serie di coppie che associano un oggetto all'insieme delle operazioni che il soggetto associato al dominio può eseguire.

Ogni dominio è associato univocamente a un soggetto.

Domini disgiunti o con diritti di accesso comune

Esiste la possibilità per due o più soggetti di effettuare alcune opezione comuni su un oggetto condiviso, le operazioni vengono svolte da processi che operano per conto di soggetti, tuttavia un processo appartiene a un solo dominio in ogni istante.

Associazione tra processo e dominio

Modalità statica L'insieme di risore disponibili a un processo rimane **statico** durante tutto il suo tempo di vita.

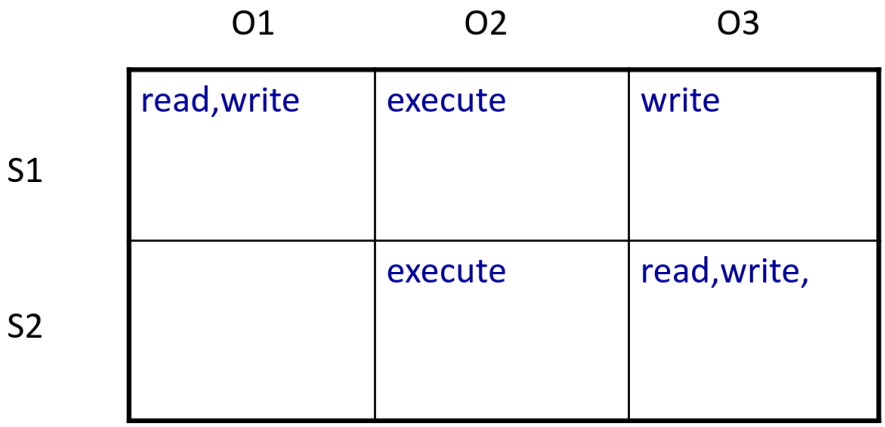
L'associazione statica non è adatta nel caso si voglia limitare per un processo l'uso delle risorse a quello strettamento necessario.

Modalità dinamica L'associazione tra processo e dominio varia durante l'esecuzione del processo.

Matrice degli accessi

Un sistema di protezione può essere rappresentato a livello astratto utilizzando una **matrice degli accessi**.

Figura 2.1: Matrice degli accessi



- Ogni riga è associata a un oggetto
- Ogni colonna è associata a un oggetto

La matrice consente di rappresentare il modello e le politiche valide nel sistema considerato, specificando

- **soggetti**
- **oggetti**
- **diritti** accordati ai soggeti sugli oggetti

Le informazioni contenute nella matrice possono variare nel tempo, per effetto di operazioni che ne consentono la modifica → le informazioni contenute nella matrice all'istante t rappresenta lo **stato di protezione** del sistema in t.

La matrice degli accessi offre ai **meccanismi** di protezione le informazioni che consentono di verificare il rispetto dei vincoli di accesso.

Il meccanismo di protezione:

- verifica se ogni richiesta di accesso che proviene da un processo che opera in un determinato dominio è consentita oppure no
- autorizza l'esecuzione delle richieste se permesse
- esegue la modifica dello stato di protezione in seguite ad ogni richiesta autorizzata da pate di un processo

Quando un operazione M deve essere eseguita nel dominio D_i sull'oggetto O_j, il meccanismo consente di controllare che M sia contenuta nella casella `access(i,j)`.

2.1.5 Modello di Graham-Denning

La modifica controllata dello stato di protezione può essere ottenuta tramite un opportuno insieme di comandi (Graham e Denning, 1972):

- create object
- delete object
- create subject
- delete subject
- read access right
- grant access right
- delete access right
- transfer access right

Propagazione dei diritti di accesso

La possibilità di copiare un diritto di accesso per un oggetto da un dominio ad un altro della matrice di accesso è indicato con il copy flag $*$.

Un soggetto S_i può trasferire un diritto di accesso α per un ogget X ad un altro soggetto S_j ad un altro soggetto S_j solo se S_i ha accesso a X con il diritto α, e α ha il copy flag.

L'operazione di propagazione può essere realizzata in due modi:

- trasferimento del diritto, viene perso dal soggetto originale
- copia del diritto: viene mantenuto dal soggetto originale

Diritto owner

Se un soggetto S_i ha il diritto **owner** su un oggetto X, può assegnare/revocare un qualunque diritto di accesso a un soggetto S_j.

Diritto control

Se un soggetto S_i ha il diritto **control** su un soggetto S_j, può assegnare/revocare un qualunque diritto di accesso a un soggetto S_j per un qualsiasi oggett X.

Figura 2.2: Matrice degli accessi

	O1	O2	O3	S1	S2
S1	read*, write	execute write	write owner		control
S2		owner	read, ~~write~~		

Switch

Un processo che esegue nel dominio del soggetto può commutare al dominio di un altro soggetto S_j. L'operazione è consentita solo se il diritto **switch** appartiene a `access(`S_i`,`S_j`)`.

2.1.6 Realizzazione della matrice degli accessi

La matrice degli accessi è una notazione astratta, realizzare in memoria una struttura dati matriciale $N_s \times N_o$ non sarebbe ottimale, considerando il fatto che è una matrice sparsa.

Esistono due approcci possibili:

- **Access Control List** (ACL): rappresentazione per colonne, ogni oggetto possiede una lista che contiene tutti i soggetti che possono accedervi, con relativi diritti di accesso.
- **Capability List**: Rappresentazione per righe, ogni soggetto possiede una lista che contiene gli oggetti accessibili, con relativi diritti di accesso.

Access control list

La lista degli accessi, per ogni oggetto, è rappresentata da un insieme di coppie:

`<soggetto, insieme dei diritti>`

limitatamente ai soggetti con un insieme non vuoto di diritti per l'oggetto.

Quando si deve eseguire l'operazione M su un oggetto O_j da parte di S_i, si cerca nella lista degli accessi

<S_i,R_k>, con M appartente a R_k

La ricerca può essere fatta in precedenza su una lista di default che contiene i diritti di accesso applicabili a tutti gli oggetti.

Utenti e gruppi Solitamente ogni soggetto rappresenta un singolo utente, molti sistemi tuttavia hanno il concetto di **gruppo di utenti**, liste di utenti identificate da un nome ch e possono essere inclusi nell'ACL.

Se i gruppi sono presenti, l'ACL ha la seguente forma:

UID_1 GID_1: `<insieme di diritti>`

con UID user identifier e GID group identifier.

Capability list

La lista delle capability, per ogni soggetto, è la lista di elementi ognuno dei quali:

- è associato a un oggetto a cui il soggetto può accedere
- contiene i diritti di accessi consentiti su tale oggetto

ogni elemento della lista prende il nome di **capability** .

La capability si compone di un identificatore (indirizzo) che identifica l'oggetto e la rappresentazione dei vari diritti concessi.

Quando S intende eseguire una operazione M su O_j, il meccanismo di protezione controlla se tra le capability di S ne esista una relativa ad O_j che contiene M

Le liste di capability devono essere protette da manomissioni, proprietà ottenibile spostando l'effettiva lista nello spazio del kernerl e esponendo soltato un riferimento a tale lista.

L'utilizzo di una sola delle due soluzioni è inefficiente, in ACL tutti i diritti di un soggetto sono sparsi nelle varie ACL degli oggetti, e nelle CL tutti i diritti di accesso applicabili a un oggeto sono sparsi nelle varie CL dei soggetti.

Revoca dai diritti di accesso

In un sistema di protezione dinamica può essere necessario revocare i diritti di accesso per un oggetto, la revoca può essere:

- **generale** o **selettiva**: valere per tutti gli utenti o solo per un sottoinsieme
- **parziale** o **totale**: tutti i diritti o un sottoinsieme
- **temporanea** o **permanente**: il diritto di accesso non sarà più disponibile, oppure potrà essere successivmente riottenuto

In ACL revocare diritti su un oggetto risulta semplice, in quanto sono raccolte in un unica entri della lista, al contrario di CL nel quale è necessario agire su ogni entry che riguarda anche l'oggetto in esame.

Cancellazione/aggiunta di un utente

In un sistema di multi-user è possibile modificare l'insieme degli utenti autorizzati:

- **cancellazione** utente esistente → eliminazione di ogni traccia dell'utente dal sistema di protezione
- **aggiunta** nuovo utente → inizializzare il sistema di protezione per l'utente e il suo accesso alle risorse

In ACL eliminare un utente è complesso, in quanto è necessario agire su ogni entry che riguarda anche l'utente in esame, al contrario di CL nel quale basta eliminare l'entry associata all'utente.

Si implementa spesso una soluzione mista, con ACL (unita a una cache in RAM per accessi frequenti), tutte le operazioni successiva su una risorsa vengono effettuate tramite il fd e le capability.

2.2 Sicurezza

La sicurezza riguarda il controllo degli accessi al sistema, la protezione di un sistema può essere inefficace se un utente non fidato riesce a fare eseguire programmi che agiscono sulle risorse del sistema.

2.2.1 Sicurezza multilivello

La maggior parte dei sistemi operativi permette ai singoli utenti di gestire accesso ai loro file e oggetti, tuttavia in alcuni ambiti è richiesto un più stretto controllo sulle regole di accesso alle risorse, ottenibile stabilendo regole più generali (MAC).

L'organizzazione che gestisce il sistema definisce le politiche MAC che stabiliscono regole generali su chi può accedere e a che cosa tramite l'adozione di un **modello di sicurezza**.

I modelli di sicurezza più usati sono:

- modello **Bell-La Padula**
- modello **Biba**

Entrambi sono modelli multilivello.

2.2.2 Modelli di sicurezza multilivello

I **soggetti** (utenti) e gli **oggetti** (risorse) sono classificati in **livelli** di accesso:

- clearance levels
- sensititivity levels

Il modello fissa inoltre le **regole di sicurezza** che controllano il flusso delle informazioni tra i livelli.

2.2.3 Modello Bell-La Padula

Progettato principalmente per organizzazioni militari che necessitano di **confidenzialità** delle informazioni. Associa a un sistema di protezione due regole di sicurezza MAC che stabiliscono il flusso di propagazione delle informazioni nel sistema.

Livelli di sensibilità degli oggetti:

- non classificato
- confidenziale
- segreto
- top secret

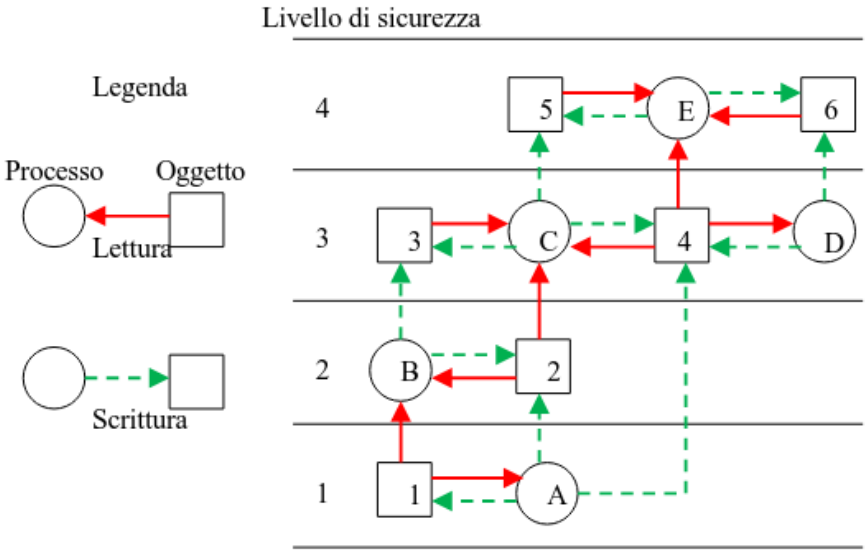
Livelli di autorizzazione (clearance) per i soggetti, assegnati a seconda del ruolo dell'utente nell'organizzazione ovvero dei documenti a quali è consentito accedere.

Regole di sicurezza

- proprietà di semplice sicurezza: un processo in esecuzione al livello di sicurezza k può leggere solo oggetti al suo livello o a livelli inferiori
- proprietà * : un processo in esecuzione allivello di sicurezza k può scrivere solamente oggetti al suol livello o a quelli superiori

Questo significa che i processi possono leggere verso il basso e scrivere verso l'alto, ma non il contrario, quindi il flusso delle informazioni è dal basso verso l'alto.

Figura 2.3: Diagramma sicurezza Bell-La Padula



Questo modello non è concepito per mantenere l'integrità dei dati ma per conservare segreti, è infatti ammesso sovrascrivere informazioni appartenenti a livelli superiori.

Esempio - difesa cavalli di troia

Si immagini un utente, Paolo, creatore del file `Fp` contenente una stringa riservata con permessi `r/w` solo per processi che appartengono a lui. Un utente ostile, Marco, ottenuto accesso al sistema, installa il file eseguibile `CT` e copia nel file system un file privato `Fm` che verrà utilizzato come "tasca posteriore".

Marco induce Paolo a eseguire il processo `CT`, che copia il contenuto di `Fp` in `Fm`, senza violare classiche regole di protezione (es. ACL).

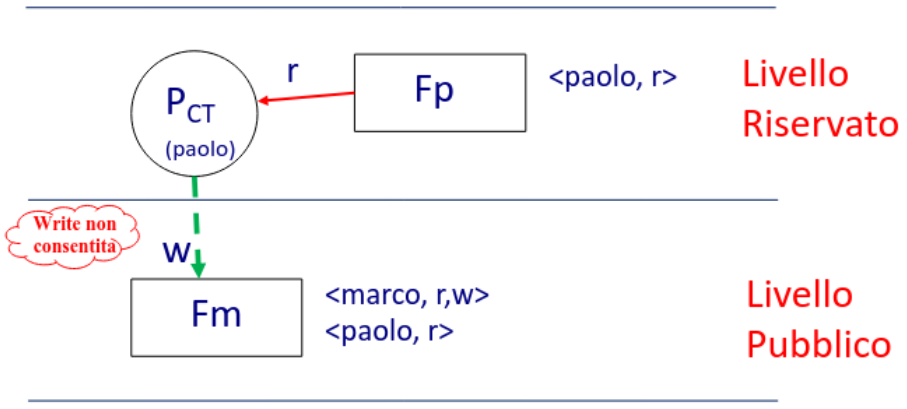
Il modello, per prevenire attacchi di questo tipo, implementa 2 livelli di sicurezza, **privato** e **pubblico**:

- ai processi e file di Paolo viene assegnato il livello *riservato*
- ai processi e file di Marco viene assegnato il livello *pubblico*

Quando il processo, avviato da Paolo con livello riservato, tenta di scrivere su `Fm` (pubblico) la proprietà `*` è violata e il tentativo è negato, nonostante ACL lo consenta.

La politica di sicurezza ha la precedenza sulle regole ACL.

Figura 2.4: Diagramma blocco cavallo di troia



2.2.4 Modello Biba

L'obiettivo di questo modello è l'integrità dei dati, diversamente dal Bell-La Padula.

- proprietà di semplice sicurezza: un processo in esecuzione al livello di sicurezza k può scrivere solo oggetti al suo livello o a livelli inferiori

- proprietà $*$: un processo in esecuzione al livello di sicurezza k può leggere solamente oggetti al suol livello o a quelli superiori

Questo modello è il duale matematico del Bell-La Padula, quindi non sono utilizzabili contemporaneamente.

2.2.5 Architettura dei sistemi ad elevata sicurezza

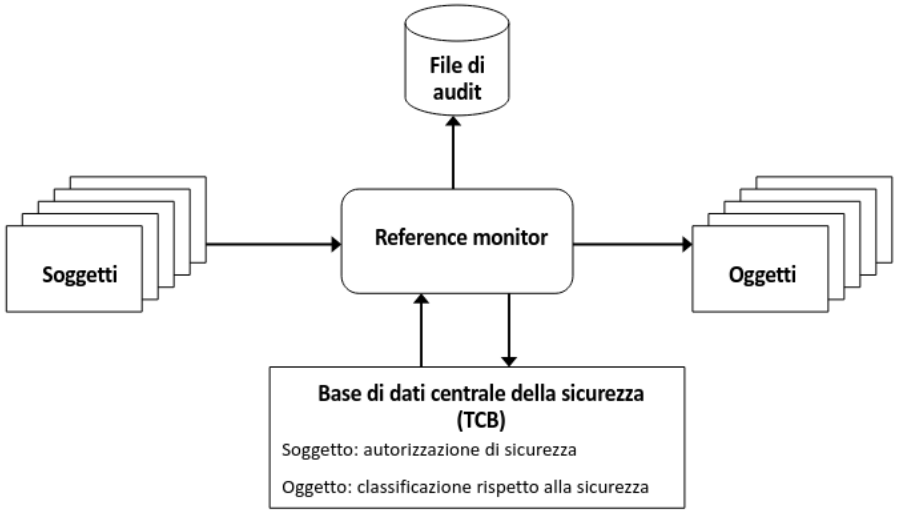
Sistemi operativi sicuri o fidati Sistemi per i quali è possibile definire formalmente dei requisiti di sicurezza.

Reference monitor É un elemento di controllo realizzato a livello hardware che regola l'accesso dei soggetti agli oggetti sulla base di parametri di sicurezza.

Trusted computing base Il RM ha accesso a una base di calcolo fidata (TCB) che contiene:

- privilegi di sicurezza per ogni soggetto
- attributi (classificazione di sicurezza) di ciascun oggetto

Figura 2.5: Architettura sistemi ad elevata sicurezza



2.2.6 Sistemi fidati

Il reference monitor impone le regole di sicurezza (Bell-La Padula) e ha le seguenti proprietà:

Mediazione completa Le regole di sicurezza vengono applicate ad ogni accesso. Per motivi di efficienza, le soluzioni devono essere almeno parzialmente hardware.

Isolamento il reference monitor e la base di dati sono protetti da modifiche non autorizzate.

Verificabilità La correttezza del reference monitor deve essere provata, deve essere possibile dimostrare formalmente che il monitor impone le corrette regole di sicurezza e fornisce mediazione completa e isolamente.

Viene inoltre compilato un **audit file** dove vengono mantenuti gli eventi importanti per la sicurezza, come i tentativi di violazione alla sicurezza e le modifiche non autorizzate al nucleo di sicurezza.

3 — Programmazione concorrente

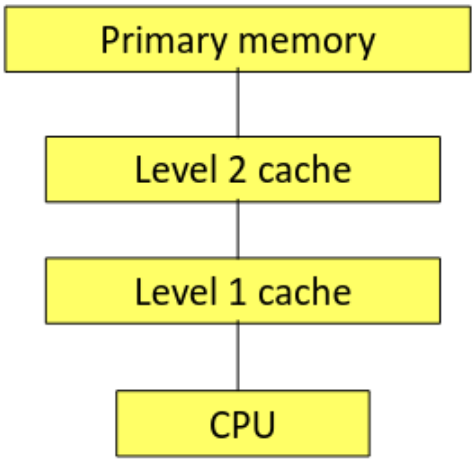
La programmazione concorrente è l'insieme delle tecniche, metodologie e strumenti per il supporto all'esecuzione di sistemi software da insieme di attività svolte simultaneamente.

Inizialmente implementata attraverso interruzioni (problemi con variabili comuni), oggi la programmazione concorrente è resa più facile dal reale parallelismo reso possibile da sistemi multiprocessore sempre più diffusi.

Le decisioni prese in merito di metodi di suddivisione dei processi e corretta sincronizzazione dipendono da tipo di applicazione e di architettura disponibile.

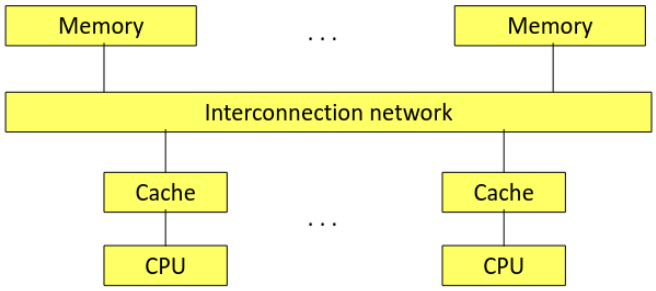
3.1 Tipi di architettura

Figura 3.1: Architettura single processor



3.1.1 Sistemi multiprocessore

Figura 3.2: Architettura memory multiprocessor

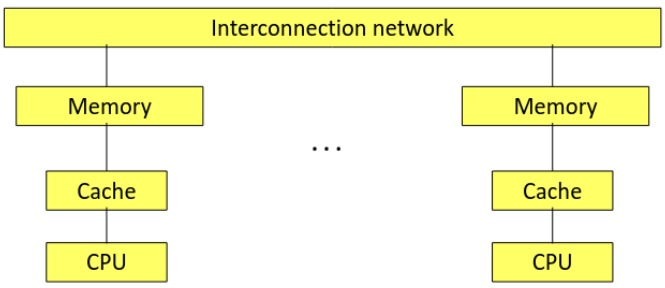


Due modelli:

- **UMA**: sistemi a multiprocessore con un numero ridotto di processori (da 2 a 30 circa)
  - la rete di interconnessione realizzata tramite *memory bus* o *crossbar switch*
  - Uniform Memory Access: tempo di accesso uniforme a da ogni processore a ogni locazione di memoria, chiamati anche **SMP** (symmetric multiprocessors).
- **NUMA**: sistemi con un numero elevato di processori (decine o centinaia)
  - memoria organizzata gerarchicamente per evitare congestioni sui bus
  - rete di interconnessione composta da insieme di *switches* e *memorie* strutturato ad albero, distanze variabili dai processori
  - Non Uniform Memory Access: tempo di accesso non uniforme

3.1.2 Distributed memory

Figura 3.3: Architettura Multicomputers e Network systems



Due modelli:

- **Multcomputer**: processori e rete fisicamente vicini → *tightly coupled machine*, la rete di interconnessione offre un cammine di comunicazione tra i processori ad alta velocità
- **Network systems**: nodi collegati da una rete locale o geografica → *loosely coupled systems*.

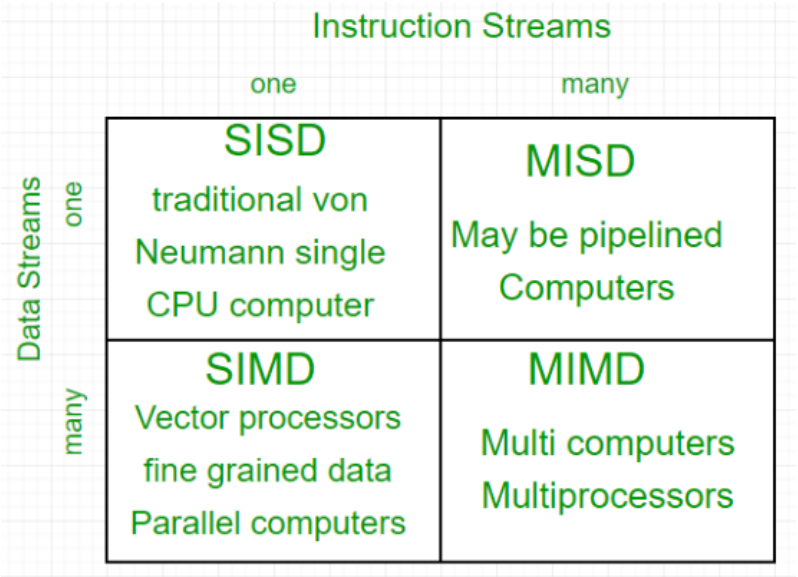
I nodi di un distributed memory system possono essere o singoli processori o shared memory multiprocessor.

3.1.3 Classificazione di Flynn

La tassonomia di Flynn è basata su due concetti:

- parallelismo a livello di istruzioni
  - **single instruction stream**: esecuzione di un singolo flusso di istruzioni
  - **multiple instruction stream**: esecuzione di più flussi in parallelo
- parallelismo a livello di dati:
  - **single data stream**: elaborazione di un singolo flusso sequenziale di dati
  - **multiple data stream**: elaborazione di multipli flussi di dati paralleli

Figura 3.4: Tassonomia di Flynn



3.2 Applicazioni

- multithreaded
  - strutturate come un insieme di processi per far fronte alla **complessità**, aumentare l'**efficienza** e per semplificare la programmazione.
  - i processi possono condividere variabili
  - esistono più processi che processori (generalmente)
  - processi schedulati ed eseguiti indipendentemente
- sistemi multitasking/distribuiti
  - le coponenti dell'applicazione vengono eseguite su nodi collegati tramite opportuni mezzi di interconnessione
  - comunicazione tramite scambio di messaggi
  - tipicamente client server
- applicazioni parallele
  - risolvere un dato problema più velocemente sfruttando il parallelismo disponibile a livello HW
  - a seconda del modello, istruzioni/thread/processi paralleli interagenti tra di loro

3.3 Processi non sequenziali e tipi di interazione

Algoritmo Procedimenti logico che deve essere eseguite per risolvere un determinato problema.

Programma Descrizione di un algoritmo mediante un linguaggio, che rende possibile l'esecuzione da parte di un elaboratore.

Processo Insieme ordinato degli eventi cui da luogo un operatore sotto il controllo di un programma.

Elaboratore Entità astratta realizzata in hardware e parzialmente in software, in grado di eseguire programmi.

Evento Esecuzione di una operazione, ogni evento determina una transizione di stato dell'elaboratore.

3.3.1 Processo sequenziale

Sequenza di stato attraverso i quali passa l'elaboratore durante l'esecuzione di un programma.

Un programma può evere più processi associati ad esso, ognuno di questi rappresenta l'esecuzione dello stesso codice con dati di ingresso (possibilmente) diversi.

Un processo può essere rappresentato tramite un grado orientato, detto grafo di precedenza del processo, composto da nodi e archi orientati. Ogni nodo rappresenta un evento corrispondente all'esecuzione di una operazione.

Il grafo di precedenza è a **ordinamento totale**, ovvero ogni nodo ha un predecessore e un successore.

3.3.2 Processo non sequenziale

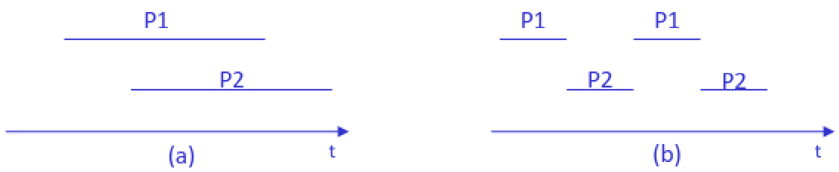
Non sempre un processo possiede la proprietà dell'ordinamento totale, molti problemi possono essere risolti più naturalmente tramite processi non sequenziali.

Un processo non sequenziale è una successione di eventi seconda una relazione d'ordine parziale.

L'esecuzione di un tale processo richiede un elaboratore in grado di supportare questo tipo di esecuzioni e un linguaggio di programmazione adatto, non sequenziale.

Elaboratore non sequenziale

- sistemi multielaboratori (a)
- sistemi monoelaboratori (b)



Linguaggio non sequenziale consente la descrizione di un insieme di attività concorrenti, sfruttando moduli che possono essere eseguiti in parallelo.

Slide 32-63!!!

3.4 Proprietà dei programmi

Traccia dell'esecuzione Sequenza degli stati attraversati dal sistema di elaborazione drante l'esecuzione del programma.

Stato Insieme dei valori delle variabili definite nel programma e di quelle implicite.

Programmi sequenziali Programmi che, su un insieme di dati D si ottiene sempre la stessa traccia.

Pogrammi concorrenti L'esito dell'esecuzione dipende dalla sequenza cronologica di esecuzione delle istruzioni contenute, lo stesso insieme di dati D può dare una traccia diversa, non determinismo.

Verificare che programmi di questo tipo siano corretti non è banale, un semplice debug non garantisce il soddisfacimento di una determinata proprietà.

3.4.1 Proprietà dei programmi

Una proprietà del programma P è un attributo che è sempre vero, data ogni traccia del programma P.

Le proprietà si possono classificare in due categorie:

- **safety properties**
- **liveness properties**

Safety

É una proprietà che garantisce che durante l'esecuzione di P, non si entrerà mai in uno stato "errato".

Liveness

É una proprietà che garantisce che durante l'esecuzione di P, prima o poi si entrerà inuno stato "corretto".

Nel caso di programmi sequenziali, entrambe le proprietà devono essere realizzate, il programma deve restituire un risultato valido per ogni esecuzione e prima o poi terminare.

Per i programmi concorrenti, si aggiungono anche altri fattori alla completezza della safety e liveness:

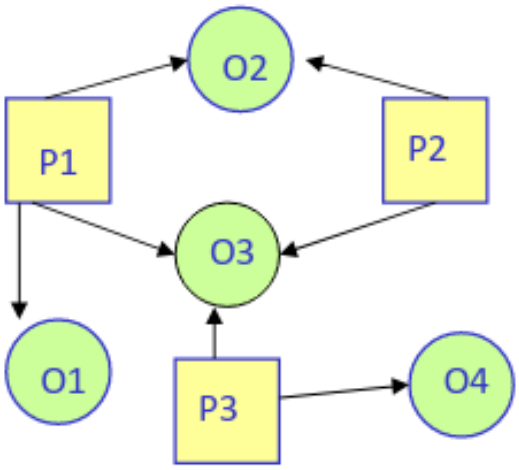
- Mutua esclusione nell'accesso a risorse (safety): nessun processo accede a una risorsa già occupata da un altro processo contemporaneamente
- Assenza di deadlock (safety): per ogni esecuzione non si devono verificare situazioni di blocco
- Assenza di starvation (liveness): prima o poi ogni processo potrà accedere akke risorse richieste

4 — Modello a memoria comune

Esistono due modelli principale di interazione tra i processi:

- memoria comune, ambiente globale con memoria condivisa
- scambio di messaggi, ambiente locale con memoria distribuita

In questo capitolo si analizza il modello a memoria comune.
Il sistema è visto come un insieme di **processi** e **oggetti**.



In questo grafo, O1 e O4 sono risorse private, mentre O2 e O3 sono comuni.
Esistono due tipi di interazione tra processi:

- **competizione**
- **cooperazione**

In questo modello, ogni applicazione viene strutturata come uninsieme di componenti, suddiviso in due sottoinsiemi disgiunti, i processi come componenti attivi e le risorse come componenti passivi.

Risorsa Qualunque oggetto fisico di cui un processo necessita per portare a termine il suo compito.

Le risorse sono raggruppate in classi, categorie che identificano l'insieme delle operazioni che un processo può eseguire.

4.1 Gestore di una risorsa

Per ogni risorsa R, il suo **gestore** definisce, in ogni istante t, l'insieme $SR(t)$ dei processi che hanno il diritto di operare su R.

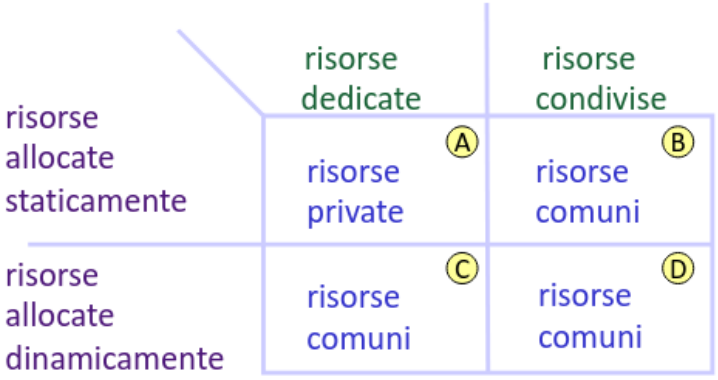
Classificazione delle risorse in base alla condivisione:

- **dedicata** se $SR(t)$ ha una cardinalità sempre ≤ 1
- **condivisa** in caso contrario

Classificazione delle risorse in base al tipo di allocazione:

- **allocata staticamente** se $SR(t)$ è una costante $SR(t) = SR(t_0), \forall t$
- **allocata dinamicamente** se $SR(t)$ è una funzione del tempo

Figura 4.1: Tipologia di allocazione delle risorse



Per ogni risorsa allocata *staticamente*, l'insieme $SR(t)$ è definito prima che il programma inizi la propria esecuzione, il gestore della risorsa è il programmatore che stabilisce quale processo può operare su R.

Per ogni risorsa allocata *dinamicamente*, il gestore G_R definisce l'insieme $SR(t)$ in fase di esecuzione e quindi deve essere un componenete della stessa applicazione, nel quale l'allocazione viene decisa a run-time in base alle politiche date.

4.1.1 Compiti del gestore di una risorsa

Il gestore di una risorsa deve essere in grado di:

- mantenere **aggiornato** l'insieme $SR(t)$ e lo stato di allocazione della risorsa
- fornire i **meccanismi** che un processo può utilizzare per ottenere i permessi per accedere alla risorsa e quindi entrare a far parte dell'insieme $SR(t)$ e per rilasciare questi permessi

- implementare la **strategia** di allocazione della risorsa e cioè definire quando, a chi e per quanto tempo allocare la risorsa

4.1.2 Accesso a risorse

Considerando un processo P che deve operare su una risorsa R di tipo T.

Allocata staticamente

Se R è allocata staticamente a P il processo, se appartiene a $SR(t)$ possiede il diritto di operae su R in qualunque istante.

Allocata dinamicamente

Se R è allocata dinamicamente a P, è necessario prevedere un gestore GR che implementa le funzioni di richiesta e rilascio, il processo deve richiedere accesso, eseguire operazione e rilasciare accesso.

Allocata condivisa

Se R è allocata come *risorsa condivisa* è necessario assicurare che gli accessi avvengano in modo non divisibile: le funzioni di acecsso alla risorsa devono essere programmate come una classe di sezioni critiche utilizzando meccanismi di sincronizzazione).

Allocata dedicata

Se R è allocata come *risorsa dedicata* essendo P l'unico processo che accede alla risorsa, non è necessario prevedere sincronizzazione.

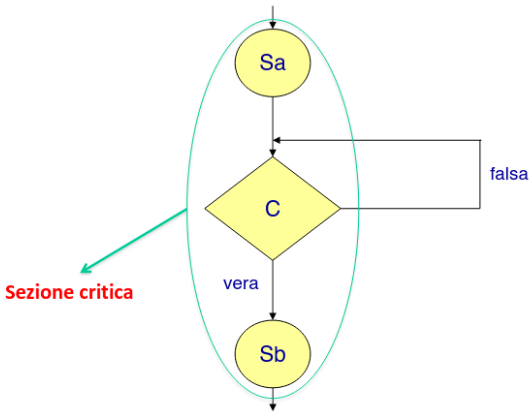
4.1.3 Specifica della sincronizzazione

Regione critica condizionale [Hoare, Brinch-hansen] Formalismo che consente di esprimere la specifica di qualunque **vincolo di sincronizzazione**.

```
region R << Sa; when(C) Sb;>>
```

Il corpo della region rappresenta una operazione da eseguire sulla risorsa condivisa R, è la **sezione critica** che deve essere eseguita in mutua esclusione con le altre operazioni.

Vengono eseguite le istruzioni `Sa` e, non appena C è vera, `Sb`.



4.1.4 Casi particolari

- `region R << S; >>` mutua esclusione senza ulteriori vincoli
- `region R << when(C) >>` specifica di un vincolo di sincronizzazione: P deve attendere che C si verifichi
- `region R << when(C) S; >>` in questo caso C è una precondizione necessaria per eseguire S

4.2 Il problema della mutua esclusione

Il problema della mutua esclusione nasce quando più processi possono avere accesso a variabili comuni, la regola impone che le operazioni non si sovrappongano nel tempo, nessun vincolo è imposto sull'ordine delle operazioni.

Sezione critica Una sequenza di istruzioni che accede e modifica un insieme di variabili comuni prende il nome di sezione critica.

La regola di mutua esclusione stabilisce che sezioni critiche della stessa classe non pososno essere in esecuzione contemporaneamente.

Il protocollo di esecuzione di una sezione critica è il seguente:

```
<prologo>
S;
<epilogo>
```

Nel prologo si richiede e ottiene l'autorizzazione a eseguire la sezione, nell'epilogo si rilascia la risorsa.

4.3 Strumenti linguistici per la programmazione di interazioni

4.3.1 Semaforo

É uno strumento di basso livello, realizzato dal kernel della macchina, utile a risolvere qualsiasi problema di sincronizzazione.

L'eventuale attesa nell'esecuzione può essere realizzata utilizzando i meccanismi di gestione dei thread (sospensione, riattivazione) offerti dal kernel.

Un semaforo è una variabile *interna non negativa* alla quale è possibile accedere solo tramite le due operazioni `P` e `V`.

Specifica delle operazioni di un semaforo:

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


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

Dato un semaforo S, siano:

- $\texttt{val}_s$: valore dell'intero non negativo associato al semaforo
- $\texttt{I}_s$: valore interno maggiore di zero di inizializzazione
- $\texttt{nv}_s$: numero di volte che l'operazione `V(s)` è stata eseguita
- $\texttt{np}_s$: numero di volte che l'operazione `P(s)` è stata eseguita

Relazione di invarianza

Ad ogni istante possiamo esprimere il valore del semaforo come $\texttt{val}_s = \texttt{I}_s + \texttt{nv}_s - \texttt{np}_s$ da cui si ottiene $\texttt{np}_s \leq \texttt{I}_s + \texttt{nv}_s$

La relazione di invarianza è sempre soddisfatta (safety property), si può usare questa proprietà per dimostrare formalmente le proprietà dei programmi concorrenti.

Utilizzo

Il semaforo è uno strumento generale che consente la risoluzione di qualunque problema di sincronizzazione, ne esistono però specializzazioni utili in particolari casi:

- semafori mutua esclusione

- semafori evento
- semafori binari composti
- semafori condizione
- semafori risorsa
- semafori privati

Semaforo mutua esclusione

Viene inizializzato a 1 ed è usato per realizzare le sezioni critiche di una stessa classe.

```
class risorsa {
    semaphore mutex = 1;
    public void op1() {
        P(mutex);
        //<sez. critica>
        V(mutex);
    }
}
```

Le seguenti condizioni sono soddisfatte:

- sezione critiche della stessa classe devono essere eseguite in modo mutualmente esclusivo
- non deve essere possibile il verificarsi di deadlock
- un processo fuori dalla sezione critica non deve bloccare l'entrata ad altri processi

4.3.2 Mutua esclusione tra gruppi e processi

In alcuni casi può essere utile consentire a più processi di eseguire contemporaneamente la stessa operazione su una risorsa, ma non operazioni diverse.

Data la risorsa condivisa `ris` e indicate con $\texttt{op}_1 \dots \texttt{op}_n$ le n operazioni eseguibili su `ris`, si vuole garantire che i processi possano eseguire contemporaneamente $\texttt{op}_i$.

Lo schema è il solito prologo, operazione, epilogo:

- il prologo deve sospendere il processo che ha chiamato l'operazione se sulla risorsa sono in esecuzione operazioni diverse, altrimenti deve procedere
- l'epilogo deve liberare la mutua esclusione solo se il processo che lo esegue è l'unico processo in esecuzione sulla risorsa

Si definisce una *semaforo mutex* per la mutua esclusione tra operazioni e un'altro per le sezioni critiche prologo e epilogo.

```
semaphore mutex=1, m_i=1;

public void op_i() {
    P(m_i);
    cont_i++;
    if (cont_i==1) P(mutex);
    V(m_i);
    //<corpo operazione>
    P(m_i);
    cont_i--;
    if (cont_i==0) V(mutex);
    V(m_i);
}
```

Questo schema è applicabile alla casistica di lettura scrittura su file: più processi possono leggere un file contemporaneamente mentre una sola può modificare.

4.3.3 Semafori evento - scambio di messaggi temporali

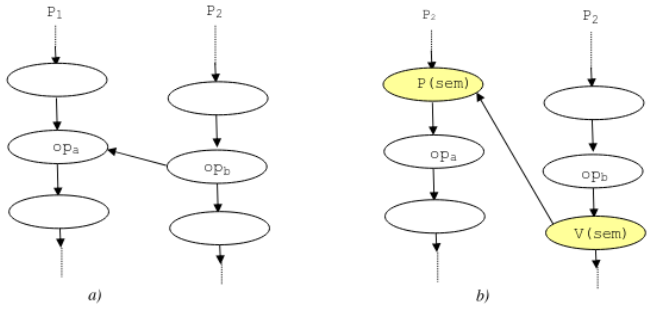
Un semaforo **evento** è un semaforo binario utilizzato per imporre un vincolo di precedenza tra operazioni e processi.

Esempio

$\texttt{op}_a$ deve essere eseguita da $\texttt{P}_1$ solo dopo che $\texttt{P}_2$ ha eseguito $\texttt{op}_b$.

Si introduce un semaforo `sem` inizializzato a zero:

- prima di eseguire $\texttt{op}_a$, $\texttt{P}_1$ esegue `P(sem)`
- dopo aver eseguite $\texttt{op}_b$, $\texttt{P}_2$ esegue `V(sem)`

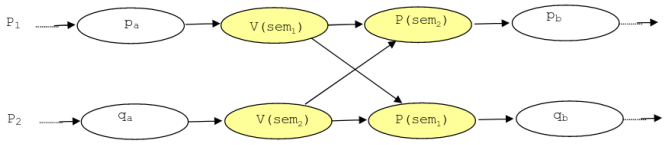


Problema del rendez-vous

Due processi $\texttt{P}_1$ e $\texttt{P}_2$ eseguono ciascuno due operazioni, $\texttt{p}_a$ e $\texttt{p}_b$ il primo e $\texttt{q}_a$ e $\texttt{q}_b$ il secondo.

Vincolo di rendez-vous
L'esecuzione di p_b da parte di P_1 e q_b da parte di P_2 possno inziare solo dopo che entrambi i processi hanno completato la loro prima operazione(p_a e q_a).

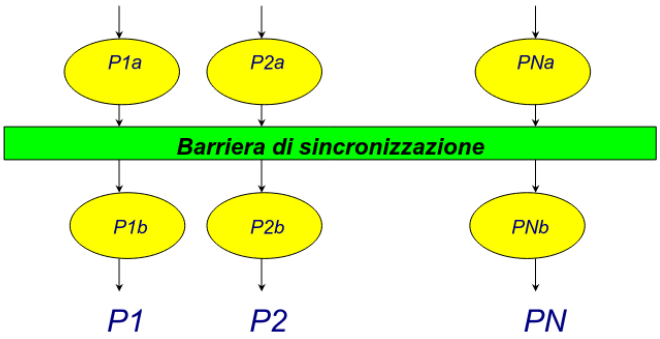
Ogni processo segnala quando arriva al punto di incontro e attende, si utilizzano due semafori `sem1` e `sem2`.



Si può generalizzare il concetto del rendez-vous a `N` processi, introducendo il concetto di barriera di sincronizzazione.

Barriera di sincronizzazione

L'esecuzione di ogni operazione P_{ib} è subordinata al completamento di tutte le istruzioni $P_{ia}(i = 1, \ldots, N)$

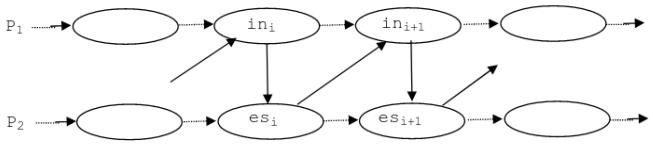



```
// condivise
semaphore mutex = 1;
semaphore barriera = 0;
int completati = 0;
//struttura thread i-esimo P_i
p(mutex);
completati++;
if (completati==N)
    v(barriera);
v(mutex);
p(barriera);
v(barriera);
```

4.3.4 Semafori binari composti - scambio di dati

Due processi P_1 e P_2 si scambiano dati di tipo T utilizzando una memoria condivisa.

Gli accessi al buffer devono essere mutuamente esclusivi, P_2 può *prelevare* un dato solo dopo che P_1 lo ha *inserito*, P_1, *prima* di inserire un dato, deve attendere che P_2 abbia *estratto* il precedente.



Per implementare questa funzionalità si utilizzano due semafori:

- vu, per realizzare l'attesa di P_1 in caso di buffer pieno
- pn, per realizzare l'attesa di P_2 in caso di buffer vuoto

```
void invio(T dato) {
    P(vu);
    inserisci(dato);
    V(pn);
}


//P1
while (true) {
    //prepara messaggio
    invio(msg);
}
```

```
T ricezione() {
    T dato;
    P(pn);
    dato = estrai();
    V(vu);
    return dato;
}
//P2
while (true) {
    M = ricezione();
    //consuma
}
```

> Un semaforo binario composto è un insieme di semafori usato in modo tale che:
>
> - uno solo di essi sia inizializzato a 1 e tutti gli altri a 0
> - ogni processo che usa questi semafori esegue sempre sequenze che iniziano con la P su uno di questi e termina con la V su un altro

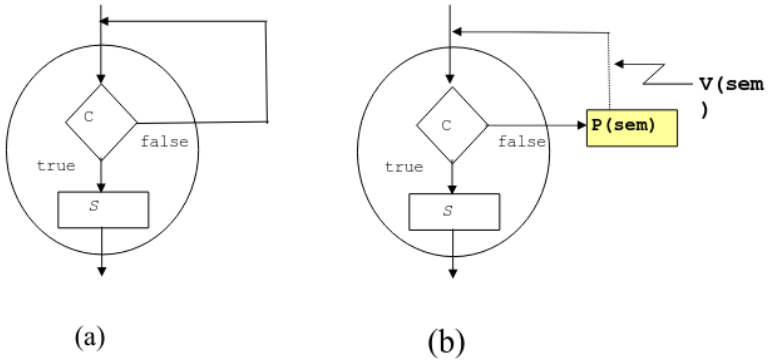
4.3.5 Semafori condizione

In alcuni casi, l'esecuzione di una istruzione S_1 su una risorsa R è subordinata a una condizione C

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

- `op1()` è una regione critica
- `S1` ha come precondizione la validità della condizione logica `C`

Il processo deve sospendersi se la condizione non si verifica, e deve uscire dalla regione per permettere ad altri processi di eseguire operazioni su `R` per rendere vera la condizione `C`.



(a) (b)

Lo schema `(a)` presuppone una forma di attesa attiva da parte del processo che non trova soddisfatta la condizione.

Nello schema `(b)` si realizza la region **sospendendo** il processo sul semaforo `sem` da associare alla condizione, si rende necessaria un'altra operazione `op2` che abbia `C` come postcondizione e che chiami nel suo contesto `V(sem)`.

Schema attesa circolare

```
public void op1() {
    P(mutex);
    while (!C) {
        csem++;
        V(mutex);
        P(sem);
        P(mutex);
    }
    S1;
    V(mutex);
}
```

```
public void op2() {
    P(mutex);
    if (csem > 0) {
        csem--;
        V(sem);
    }
    V(mutex);
}
```

Schema passaggio testimone

```
public void op1() {
    P(mutex);
    if (!C) {
        csem++;
        V(mutex);
        P(sem);
        csem--;
    }
    S1;
    V(mutex);
}
```

```
public void op2() {
    P(mutex);
    S2;
    if (C && csem > 0)
        V(sem);
    else
        V(mutex);
}
```

Il metodo del testimone è il più efficiente, tuttavia è possibile risvegliare *un solo* processo alla volta e la condizione `C` non può dipendere da parametri locali a `op1()`.

Gestione di un pool di risorse equivalenti

Si consideri un **pool** di `N` risorse tutte uguali, ciascun processo può operare su una qualsiasi risorsa del pool, purché *libera*.

É necessario un **gestore** che mantenga aggiornato lo stato delle risorse:

- ogni processo, prima di operare su una risorsa *chiede al gestore* l'allocazione di una di esse
- il gestore **assegna** al processo una risorsa libera, passandogli l'**indice** relativo
- il processo opera sulla risorsa
- al termine il processo **rilascia** la risorsa a gestore

```
class Gestore {
    semaphore mutex = 1;
    semaphore sem = 0;
    int csem = 0;
    boolean libera[N];
    int disponibili = N;
    {for (int i = 0; i < N; i++) libera[i]=true;}

    public int richiesta() {
        int i = 0;
        P(mutex);
        if (disponibili == 0) {
            csem++;
            V(mutex);
            P(sem);
```

```
            csem--;
        }
        while (!libero[i]) i++;
        libero[i] = false;
        V(mutex);
        return i;
    }

    public void rilascio(int r) {
        P(mutex);
        libero[r] = true;
        disponibili++;
        if (csem > 0)
            V(sem);
        else
            V(mutex);
    }
}
```

Schema di esempio di un generico processo che vuole accedere a una risorsa `ris`.

```
process P {
    int ris;
    ...
    ris = G.richiesta();
    //<utilizzo risorsa>
    G.rilascio();
    ...
}
```

4.3.6 Semaforo risorsa

I semafori risorsa sono semafori generali, ovvero possono assumere qualunque valore maggiore di zero.

Sono utilizzato per realizzare allocazione per risorse equivalenti, il valore del semaforo rappresenta il **numero di risorse libere**.

Gestione di un pool di risorse equivalenti

Si può utilizzare un semaforo risorsa per gestire un pool di risorse, si crea un unico semaforo `n_ris` inizializzato con un valore uguale al numero di risorse, si eseguono P(n_ris) in allocazione e V(n_ris) in rilascio.

```
class Gestore {
    semaphore mutex = 1;
    semaphore n_ris = N;
```

```
    boolean libero[N];

    {for (int i = 0; i < N; i++)
        libera[i] = true;}

    public int richiesta() {
        int i = 0;
        P(n_ris);
        P(mutex);
        while (!libero[i]) i++;
        libero[i] = false;
        V(mutex);
        return i;
    }

    public void rilascio (int r) {
        P(mutex);
        libero[r] = true;
        V(mutex);
        V(n_ris);
    }

}
```

Problema dei produttori/consumatori

Si crea un buffer di `n` elementi, strutturato come una coda

```
coda_di_n_T buffer;
semaphore pn = 0;
semaphore vu = 0;
semaphore mutex = 1;
```

```
void invio(T dato) {
    P(vu);
    P(mutex);
    buffer.inserisci(dato);
    V(mutex);
    V(pn);
}
```

```
void ricezione() {
    T dato;
    P(pn);
    P(mutex);
    dato = buffer.estrai();
    V(mutex);
    V(vu);
    return dato;
}
```

4.3.7 Semafori privati - specifiche strategie di allocazione

Qualora si voglia realizzare una politica di gestione delle risorse particolare, la decisione se permettere o no l'esecuzione a un dato processo dipende dal verificarsi duna specifica **condizione di sincronizzazione**.

Queste condizioni vengono espresse in termini di variabili che rappresentano lo **stato della risorsa** e variabili *locali* ai singoli processi.

Sorge il problema di quale processo mettere in esecuzione, risolvibile definendo una **politica di risveglio** dei processi bloccati.

Nei casi precedenti la politica di risveglio era dipendente dalla specifica implementazione dell'algoritmo all'interno della V, solitamente realizzato con risveglio FIFO.

> Un semaforo `s` si dice **privato** per un processo quando solo tale processo può eseguire la primitiva `P` su semaforo `s`. La primitiva `V` può essere eseguita da qualunque processo.

Un semaforo privato viene inizializzato con valore **zero**.

Il processo che acquisisce la risorsa può sospendersi sul suo semaforo privato se la condizione di sincronizzazione non è soddisfatta, chi rilascia tale risorsa, può risvegliare uno dei processi sospesi (in base alla politica definita) mediante una `V` sul semaforo privato del processo scelto.

Schema generale:

```
class Gestore {
    //<struttura dati gestore>;
    semaphore mutex = 1;
    semaphore priv[n] = {0, 0, ..., 0} //semafori privati

    public void acquisizione (int i) {
        P(mutex);
        if (/*<condizione di sincronizzazione>)*/ {
            //<allocazione della risorsa>;
            V(priv[i]);
        }
        else {
            //<registrare la sospensione del processo>;
        }
        V(mutex);
        P(priv[i]);
    }

    public void rilascio() {
        int i;
        P(mutex);
```

```
        //<rilascio risorsa>;
        if (/*<min 1 processo soddisfa sync condition>*/) {
            //<scelta tra sospesi del Pi da riattivare>;
            //<allocazione risorsa a Pi>;
            //<registrare Pi come non sospeso>;
            V(priv[i]);
        }
        V(mutex);
    }
}
```

La sospensione del processo in acqusizione, se la condizione di sincronizzazione non è soddisfatta, deve avvenire *al di fuori dell sezione critica*, altrimenti si impedirebbe a un processo che rilascia la risorsa di accedere a sua volta alla sezione critica.

La soluzione mostrata può presentare inconvenienti:

- la `P` sul privato viene **sempre eseguita**, anche se non necessario
- il codice dell'assegnazione della risorsa è *duplicato* nelle due procedure

Correggendo questi problemi:

```
class Gestore() {
    //<struttura dati gestore>;
    semaphore mutex = 1;
    semaphore priv[n] = {0, 0, |$\dots$|, 0}; //semafori privati

    public void acqusizione(int i) {
        P(mutex);
        if (! //<condizione di sincronizzazione) {
            //<registrare sospensione processo>;
            V(mutex);
            P(priv[i]);
            //<registrare processo come non sospeso>;
        }
        //<allocazione risorsa>;
        V(mutex);
    }

    public void rilascio() {
        int i;
        P(mutex);
        //<rilascio risorsa>;
        if (/*<min 1 processo soddisfa sync condition>*/) {
            //<scelta tra sospesi del Pi da riattivare>;
            V(priv[i]);
        }
```

```
        else
            V(mutex);
    }
}
```

Il risveglio segue lo schema del **passaggio del testimone**.

A differenza della soluzione precedente è più complesso realizzare la riattivazione di più processi che hanno condizione di sincronizzazione verificata: il processo che rilascia la risorsa attiva al massimo un processo, il quale dovrà a sua volta provvedere a riattivare eventuali processi.

Esempio 1

Su un buffer di N celle di memoria, più produttori possono depositare messaggio di dimensione diversa.

Politica di gestione: tra più produttori ha priorità di accesso quello che fornisce il messaggio di dimensione maggiore.

Condizione di sincronizzazione: il deposito può avvenire se c'è abbastanza spazio per memorizzare il messaggio e non ci sono produttori in attesa.

Il **prelievo** di messaggi da parte di un consumatore riattiva il produttore con messaggio con **dimensione maggiore** (se spazio sufficiente nel buffer), se lo *spazio non è sufficiente*, nessun produttore viene riattivato.

Soluzione:

```
class Buffer {
    int richiesta[num_proc] = 0; //numero di celle richieste da Pi
    int sospesi = 0;
    int vuote = n; //numero celle vuote del buffer

    semaphore mutex = 1;
    semaphore priv[num_proc] = {0, 0, ..., 0};

    public void acqusizione(int m, int i) {
        // m dim mess, i id processo
        P(mutex);
        if (sospesi == 0 && vuote >= m) {//assegna m celle a Pi
            vuote = vuote - m;
            V(priv[i]);
        }
        else {
            sospesi++;
            richiesta[i] = m;
        }
        V(mutex);
        P(priv[i]);
```

```
    }

    public void rilascio(int m) {
        int k;
        P(mutex);
        vuote += m;
        while (sospesi != 0) {
            //<individuazione del Pk con max richiesta>;
            if (richiesta[k] <= vuote) { //assegno a Pk
                vuote = vuote - richiesta[k];
                richiesta[k] = 0;
                sospesi--;
                V(priva[k]);
            }
            else {
                break;
            }
        }
        V(mutex);
    }
}
```

Esempio 2

Un insieme di processi utilizza un insieme di risorse comuni R1, R2, . . . , Rn. Ogni processo può utilizzare una qualunque delle risorse.

Politica di gestione: a ogni processo è assegnata una **priorità**, in fase di riattivazione dei processi sospesi, viene scelto quello con la *massima priorità*.

Condizione sincronizzazione: accesso consentito solo se esiste una *risorsa libera*.

Soluzione:

Variabili introdotte

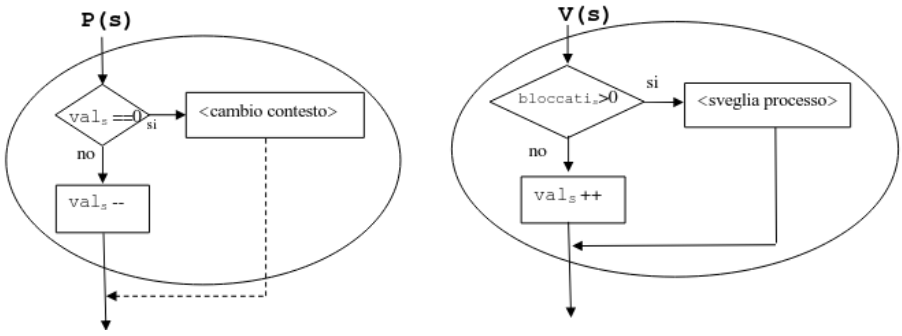
- `PS[i]`: vero se il processo `Pi` è sospeso, falso altrimenti
- `libera[j]`: falso se la risorsa j-esima è occupata, vero altrimenti
- `disponibili`: numero delle risorse non occupate
- `sospesi`: numero di processi sospesi
- `mutex`: semaforo di mutua esclusione
- `priv[i]`:semafori privati del processo `Pi`

```
class Gestore {
    semaphore mutex = 1;
    semaphore priv[num_proc] = {0, 0, ..., 0};
    int sospesi = 0;
    boolean PS[num_proc] = {false, false, ..., false};
    int disponibili = num_ris;
    boolean libera[num_ris] = {true, true, ..., false};

    public int richiesta(int proc) {
        int i = 0;
        P(mutex);
        if (disponibili == 0) {
            sospesi++;
            PS[proc] = true;
            V(mutex);
            P(priv[proc]);
            PS[proc] = false;
            sospesi--;
        }
        while (! libera[i]) i++;
        libera[i] = false;
        disponibili--;
        V(mutex);
        return i;
    }

    public void rilascio(int r) {
        P(mutex);
        libera[r] = true;
        disponibili++;
        if (sospesi > 0) {
            //<seleziona il processo Pj a massima priorità tra i sospesi>
            V(priv[j]);
        }
        else V(mutex);
    }
}
```

4.4 Realizzazione dei semafori

Nei sistemi multiprogrammati, il semaforo è realizzato dal kernel, che usa i meccanismi di gestione dei processi per evitare attesa attiva.

Si può definire le funzioni `P` e `V` in questo modo, garantendo le proprietà del semaforo.



Un semaforo è identificato dal suo descrittore, definito come segue:

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

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 *coda non vuota*, *estrae* un processo dalla coda, altrimenti *incrementa il contatore*.

4.4.1 Architettura monoprocessore

Considerando interruzioni disabilitate (per garantire l'atomicità), l'implementazione di `P` e `V` sono:

```
void P(semaphore s) {
    if (s.contatore == 0)
        //<sospensione del processo nella coda associata a s>;
    else
        s.contatore++;
}

void V(semaphore s) {
    if (s.queue != NULL)
        //<estrazione del primo processo dalla coda a s.queue e impostazione a stato di ready>
    else
        s.contatore--;
}
```

5 — Il nucleo di un sistema multiprogrammato (modello a memoria comune)

5.0.1 Nucleo di un sistema a processi

In un sistema multiprogrammato, vengono offerte tante unità di elaborazione astratte quanti sono i processi. Ogni macchina possiede tutte le istruzioni elementari dell'unità centrale, più alcune irate alla gestione dei processi, alla comunicazione e sincronizzazione.

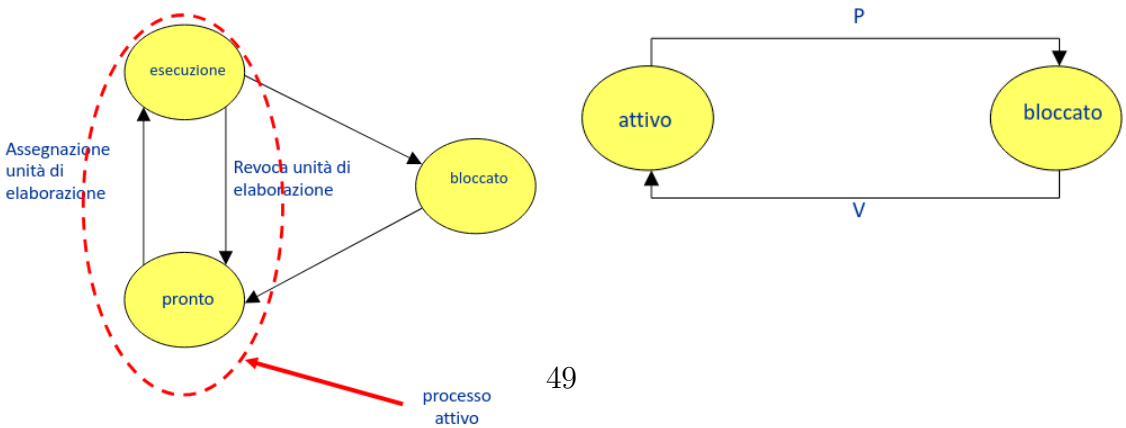
Questo modello mette in evidenza le proprietà logiche di comunicazione e sincronizzazione senza pensare agli aspetti implementativi.

> Si chiama **kernel** (nucleo) il modulo realizzato in software o firmware che supporta il concetto di processo e realizza gli strumenti necessari per la gestione di questi ultimi.

Il nucleo è l'unico componente del sistema che è "conscio" dell'esistenza delle **interruzioni**: ogni processo che richiede l'esecuzione di una operazione a un dispositivo effettua una chiamata primitiva al nucleo che lo *sospende*, quest ultimo riceve un *segnale di interruzione* dal dispositivo e risveglia il processo sospeso.

La gestione delle interruzioni è invisibile ai processi.

5.0.2 Stati di un processo



Contesto di un processo
Insieme delle informazioni contenute nei registri del processore, quando esso opera sotto il controllo del processo.

Salvataggio di contesto
Quando un processo perde il controllo del processore, il contenuto dei registri del processore (contesto) viene salvato in una struttura dati associata al processo, chiamata **descrittore**.

Ripristino del contesto
Quando un processo viene schedulato, i valori salvati nel suo descrittore vengono caricati nei registri del processore.

5.0.3 Funzioni del nucleo

Il compito fondamentale del nucleo è quello di **gestire le transizione di stato** dei processi, in particolare:

- gestione del **salvataggio** e **ripristino** dei *contesti* dei processi
- scelta, tra i processi pronti, del processo al quale assegnare l'unità di elaborazione (**scheduling**, secondo una particolare poitica (FIFO, SJF, Priorità etc.)
- gestione delle interruzioni dei dispositivi esterni
- realizzazione dei meccanismi di sincronizzazione dei processi

5.0.4 Caratteristiche del nucleo

Efficienza
Dato che condiziona l'intera struttura a processi, in certi sistemi alcune funzionalità sono realizzate in hardware o microprogrammi.

Dimensioni
Necessità di avere un nucleo semplice e di dimensione limitata.

Separazione meccanismi e politiche
Il nucleo deve contenere, possibilmente, soltanto meccanismi, consentendo così a livello di processo di utilizzare tali meccanismi ad hoc.

5.1 Realizzazione del nucleo (monoprocessore)

5.1.1 Strutture dati del nucleo

Descrittore del processo

Contiene le seguenti informazioni

- **identificatore** del processo, univoco
- **stato** del processo: pronto, esecuzione, bloccato etc.
- modalità di **servizio** dei processi: parametri di scheduling, ad esempio
  - FIFO
  - Priorità
  - Deadline (completamento richiesta processo)
  - quanto di tempo
- **contesto** del processo: *contatore* di programma, *registro di stato*, registri generali, indirizzo memoria privata del processo
- riferimenti a **code**: riferimento elemento successivo nella coda (di processi bloccati, di processi pronti...)

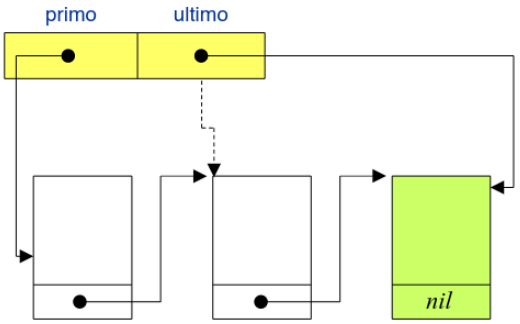
```
typedef struct {
    int indice_priorità;
    int delta_t;
} modalita_di_servizio;

typedef struct {
    int nome;
    ...
    modalita_di_servizio servizio;
    tipo_contesto contesto;
    tipo_stato stato;
    int successivo;
} descrittore_processo;
```

Coda dei processi pronti

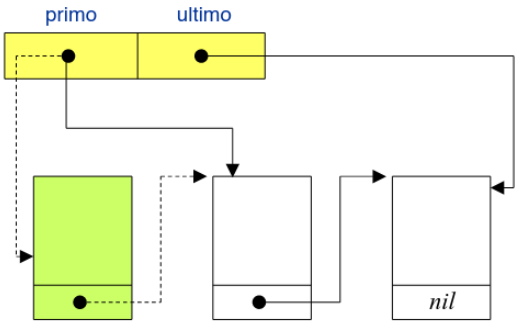
Esistono diverse code di *processi pronti*, quando un processo è riattivato da un V, viene inserito in fondo alla coda corrispondente alla sua priorità.

Non sempre è presente un processo pronto, però esiste un **dummy processo** che va in esecuzione quanto tutte le altre code sono vuote, rimanendoci fino a un altro processo diventa pronto, ha la priorità più bassa ed è sempre in stato pronto.

```
typedef struct {
    int primo, ultimo;
} descrittore_coda;

typedef descrittore_coda coda_a_livelli[Npriorità];
```




```
void Inserimento(int P,
                 descrittore_coda C) {
    //inserimento processo di indice P
    //nella coda C
}
```




```
int Prelievo(descrittore_coda C) {
    //estrae il primo processo da coda
    //C e restituisce l'indice
}
```

Coda dei descrittori liberi

In questa coda sono contenuti i descrittori *disponibili* per la creazione di nuovi processi, vengono reinseriti i descrittori dei processi terminati.

```
descrittore_coda descrittori_liberi;
```

Processo in esecuzione

Il nucleo necessita di tenere traccia di quale processo è in esecuzione, informazione rappresentata dall'indice del descrittore del processo, viene salvato in una variabile del nucleo (spesso un registro del processore)

```
int processo_in_esecuzione;
```

5.1.2 Funzioni del nucleo

Le funzioni del nucleo implementano le operazioni di **transizione di stato** per i singoli processi, ogni transizione prevede il *prelievo* da una coda e l'*inserimento* in un'altra.

Se la coda è vuota, essa contiene soltanto il valore -1 (NIL), valore inserito anche in code non vuote come ultimo elemento.

Struttura del nucleo

La struttura del nucleo si articola in due livelli:

- **livello superiore**: contiene tutte le funzioni direttamente **utilizzate dai processi**, sia interni che esterni; in particolare le primitive per la creazione, eliminazione e sincronizzazione dei processi e le funzioni di risposta ai segnali di interruzione
- **livello inferiore**: realizza le funzionalità di **cambio di contesto**: salvataggio del contesto del processo deschedulato, scelta di un nuovo processo da mettere in esecuzione tra quelli pronti e ripristino del suo contesto

Esecuzione del kernel

Le funzioni del nucleo, per motivi di protezione, sono le sole che possono operare su strutture dati dello stato del sistema e utilizzare istruzioni privilegate.

Grazie al meccanismo dei **ring**, nucleo e processi eseguono in due **ambienti separati**:

- *nucleo*, massimo privilegio: **modo kernel** o ring 0
- *processi*, minore privilegio: **modo user** o ring ¿ 0

Il passaggio da un "modo" all'altro è basato sul *meccanismo delle interuzioni*:

- funzioni da processi **esterni**: passaggio ad ambiente nucleo tramite risposta al segnale di **interruzione**
- funzioni da processi **interni**: passaggio ad ambiente nucleo tramite **system calls** (SVC, interruzioni interne)

In entrambi i casi al completamento della funzione, il trasferimento avviene tramite il meccanismo di **ritorno da interruzione** (RTI).

Funzioni del livello inferiore

Le funzioni principali di **cambio di contesto** sono

- `salvataggio_stato`: salvataggio del contesto del processo in esecuzione e inserimento del descrittore nella coda dei processi bloccati o pronti
- `assegnazione_cpu`: rimozione del processo a maggior priorità dalla coda e caricamento dell'identificatore nel registro del processo in esecuzione
- `ripristino_stato`: caricamento del contesto del nuovo processo nei registri di macchina

```
void salvataggio_stato() {
    int j;
    j = process_in_esecuzione;
    descrittori[j].contesto = <valori dei registri CPU>;
}

void ripristino_stato() {
    int j;
    j = processo_in_esecuzione;
    <registro-temp> = descrittori[j].servizio.delta_t;
    <registri-CPU> = descrittori[j].contesto;
}

void assegnazione_CPU() { //scheduling: hp algoritmo con priorità
    int k = 0, j;
    while ((coda_processi_pronti[k].primo) == -1) {
        k++;
    }
    j = prelievo(coda_processi_pronti[k]);
    processo_in_esecuzione = j;
}
```

Gestione del temporizzatore

Per consentire la modalità di servizio a divisione del tempo è necessario che il nucleo gestisca un **dispositivo temporizzatore** tramite una apposita procedura che ad intervalli di tempo fissati, provveda a sospendere il processo in esecuzione ed assegnare l'unità di elaborazione ad un altro processo.

```
void cambio_contesto() {
    int j, k;
    salvataggio_stato();
    j = processo_in_esecuzione;
```

```
    k = descrittori[j].servizio.priorità;
    inserimento(j, coda_processi_pronti[k]);
    assegnazione_CPU();
    ripristino_stato();
}
```

5.2 Realizzazione del semaforo (monoprocessore)

5.2.1 Semafori

Nel nucleo di un sistema monoprocessore il semaforo può essere implementato tramite:

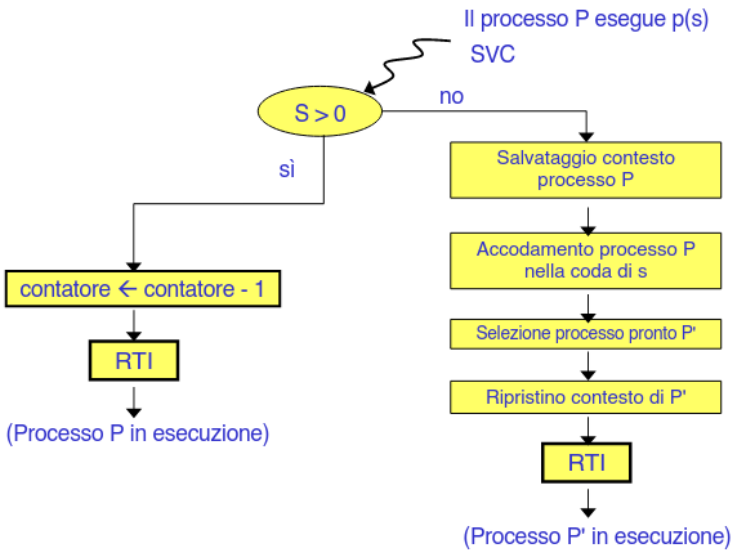
- una *variabile intera* che rappresenta il suo valore (`>= 0`)
- una *coda di decrittori* dei processi in attesa sul semaforo

Se non ci sono processi in coda, il puntatore contiene costante `NIIL`.

La coda viene gestita con politica FIFO, un descrittore viene inserito nella coda come risultato dei una `P` bloccante e viene prelevato per effetto di una `V`.

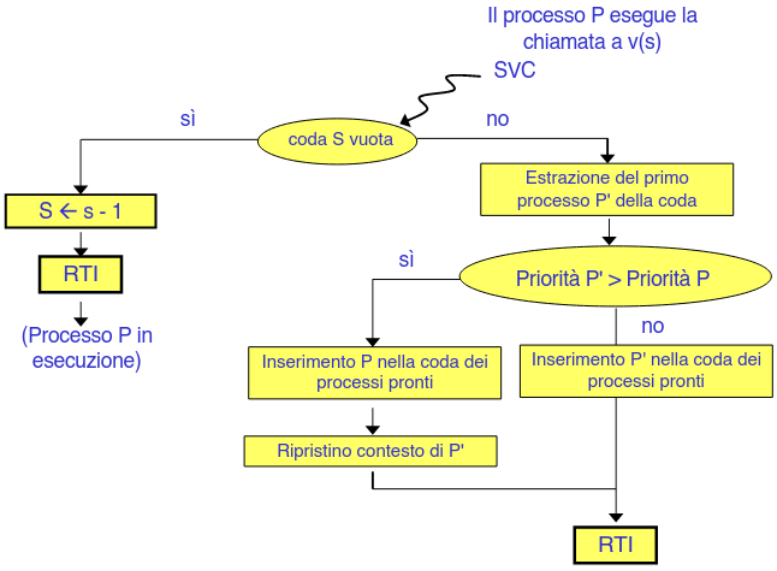
```
typedef struct {
    int contatore;
    descrittore_coda coda;
} descr_semaforo;
```

Operazione P



```
// insieme di tutti i semafori
descr_semaforo semafori[num_max_sem];
// ogni semaforo è rappresentato dall'indice che lo individua nel vettore

typedef int semaforo;

void P(semaforo s) {
    int j, k;
    if (semafori[s].contatore == 0) {
        salvataggio_stato();
        j = processo_in_esecuzione;
        inserimento(j, semafori[s].coda);
        assegnazione_CPU(); // scheduling
        ripristino_stato();
    }
    else
        contatore--;
}
```

Operazione V

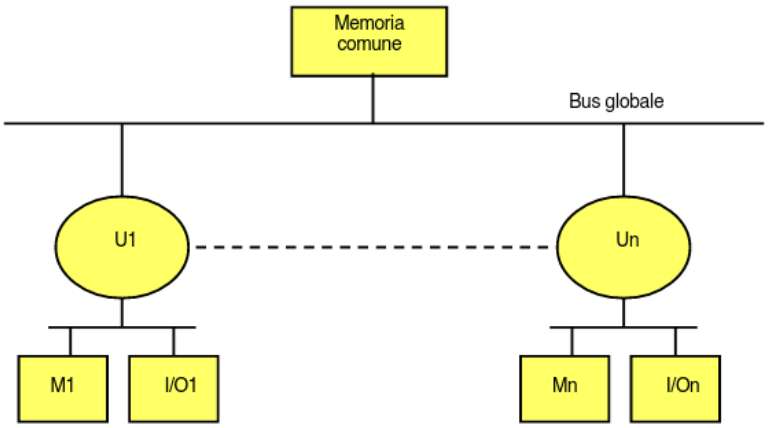



```
void V(semaforo s) {
    // j, k processi -- p, q indici priorità
    int j, k, p, q;

    if (semafori[s].coda.primo != -1) { // la coda non è vuota
        k = prelievo(semafori[s].coda);
        j = processo_in_esecuzione;
        p = descrittori[j].servizio.priorità; // priorità proc running
        q = descrittori[k].servizio.priorità; // priorità proc risvegliato
```

```
        if (p > q) // processo running ha priorità
            inserimento(k, coda_processi_pronti[q]);
        else {
            salvataggio_stato();
            inserimento(j, coda_processi_pronti[p]);
            processo_in_esecuzione = k;
            ripristino_stato();
        }
    }
    else
        semafori[s].contatore++; // se non ci sono processi in coda
}
```

5.3 Realizzazione del nucleo (multiprocessore)

Architettura



Modelli principali

Un OS che esegue su una architettura multiprocessore deve gestire **molteplici CPU**, le quali possono tutte accedere a una **memoria condivisa**.

Esistono due modelli:

- modello **SMP**: unica copia del nucleo, condivisa tra tutte le CPU
- modello **nuclei distinti**: più istanze concorrente del nucleo

5.4 Modello SMP - Symmetric Multi Processing

Nel modello SMP, esiste una *unica copia del nucleo* del sistema operativo, allocata nella memoria comune e si occupa della gestione di tutte le risorse disponibili, comprese le CPU.

Caratteristiche:

- ogni processo può essere allocato su *una qualunque* delle CPU
- la system call richieste dai processi possono essere contemporanee, è necessario che gli accessi al nucleo avvengano in modo **sincronizzato**, in quanto in generale è richiesto l'accesso a strutture dati interne

5.4.1 Sincronizzazione nell'accesso al nucleo

Soluzione singolo lock

Viene associato un lock `L` al nucleo, per garantire la mutua esclusione dell'esecuzione di funzioni del nucleo: si sincronizza l'accesso alle strutture dati eseguendo `lock` e `unlock` sull'unico lock `L`.

Il problema fondamentale di questa soluzione è la vasta *limitazione del grado di parallelismo*, escludendo operazioni che accedono a strutture dati distinte.

Soluzione molteplici lock

Un grado maggiore di parallelismo si può ottenere individuando *classi* di sezioni critiche, ognuna associata a una struttura sufficientemente indipendente dalle altre, a ogni struttura dati viene associato un **lock distinto**.

5.4.2 Scheduling dei processi

Il modello SMP consente la schedulazione di ogni processo su uno qualunque dei processori, con la possibilità di attuare politiche **load balancing** sul carico dei diversi processori.

Esistono casi in cui conviene assegnare un processo a un *determinato processore*:

- schedulare il processo a un processore che contiene il codice nella sua memoria privata (accesso più veloce)
- sistemi NUMA accedono alla memoria più "vicina" più velocemente, quindi conviene schedulare il processo a un processore più vicino alla memoria contenente lo spazio di indirizzamento

- in caso di presenza di memoria cache (es. TLB memoria virtuale) può essere utile schedulare su un processore precedentemente usato per un particolare processo

5.5 Modello a nuclei distinti

In questo modello, la struttura interna del sistema operativo è articolata su più nuclei, ognuno dedicato alla gestione di una CPU diversa.

Si assume che i processi presenti nel sistema siano divisibili in sottoinsiemi, **nodi virtuali**, *lascamente* connessi, ovvero con un poche interazioni.

Ogni nodo virtuale è assegnato a un **nodo fisico** gestito da un nucleo distinto, nella *memoria privata* del nodo fisico vengono istanziate le strutture dati del nucleo relative al nodo virtuale assegnato.

Con questa tecnica, tutte le interazioni *locali* al nodo virtuale, possono avvenire in modo **indipendente** e **concorrente** a quelle di altri nodi virtuali.

Solo le interazioni tra processi appartenenti a nodi virtuali diversi utilizzano la *memoria comune*.

5.5.1 Nuclei distinti vs. SMP

Principali differenze:

- **maggiore parallelismo** delle CPU: ND meno accoppiate e più scalabile di SMP
- **gestione NON ottimale** delle risorse computazionali: ND obbliga a schedulare un procesos sullo stesso nodo, SMP permette un load balancing migliore

5.6 Realizzazione semafori

5.6.1 Modello SMP

Le CPU condividono lo stesso nucleo, quindi per sincronizzare gli accessi al nucleo le strutture dati sono protette tramite lock.

In particolare:

- singoli **semafori**
- la **coda dei processi pronti**

vengono protetti da **lock distinti**

In questo modo, due operazioni `P` su semafori diversi possono operare in modo **contemporaneo** se non risultano sospensive, altrimenti vengono **sequenzializzati** solo gli accessi alla coda dei processi pronti.

Esempio

Scheduling pre-emptive basato su priorità.

L'esecuzione della `V` può portare al cambio di contesto tra il processo risvegliato P_r e uno tra i processi in esecuzione P_i, se la priorità di P_r è più alta di P_i il nucleo deve provvedere a revocare l'unità di elaborazione al processo P_i e assegnarla a P_r riattivato dalla `V`.

É necessario che il nucleo mantenga l'informazione del processo in esecuzione e del nodo fisico su cui opera.

Occorre inoltre un meccanismo di **segnalazione** tra le unità di elaborazione.

SMP - segnalazione inter-processore

Siano:

- `sem`, sul quale è inizialmente sospeso un processo P_j
- P_i processo che esegue `V(sem)` dal nodo U_i
- P_j processo riattivato per effetto di `V(sem)`
- P_k processo a più bassa priorità sul nodo U_k

accade:

1. P_i chiama la `V(sem)`: il nucleo, attualmente eseguito da U_i con la `V(sem)`, invia una interruzione a U_k
2. U_k gestice l'interruzione: inserisce P_k nella coda dei processi pronti e mette in esecuzione P_j

5.6.2 Modello nuclei distinti

In questo modello solo le interazioni tra processi su nodi virtuali diversi utilizzano la memoria comune.

Si distinguono:

- semafori **privati**: utilizzati dai processi appartenenti a un nodo `U`, realizzati come nel caso monoprocessore
- semafori **condivisi**: utilizzati da processi appartenenti a nodi virtuali diversi

La memoria comune dovrà contenere tutte le informazioni relative ai semafori condivisi.

5.6.3 Realizzazione semafori condivisi

Ogni semaforo condiviso è rappresentato, in memoria comune, da un intero non negativo e sarà protetto da un lock, anch'esso in memoria comune.

Rappresentante del processo
Insieme minimo di informazione sufficienti per identificare sia il *nodo fisico* su cui il processo opera, sia il *descrittore* contenuto nella memoria privata del processore.

Per ogni semaforo condivis `S`, vengono mantenute *diverse code*:

- per nodo U_i: una coda locale a U_i contenente i descrittore dei processi locali sospesi su `S`, la coda risiede nella memoria privata del nodo e viene gestita esclusivamente dal nucleo associato
- **globale**: tutti i processi sospesi su `S`, accessibile da ogni nucleo

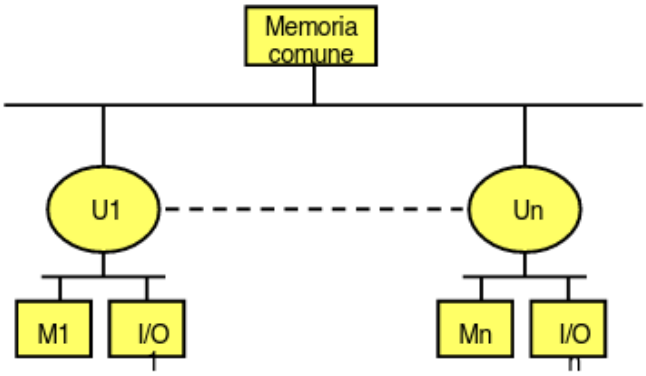
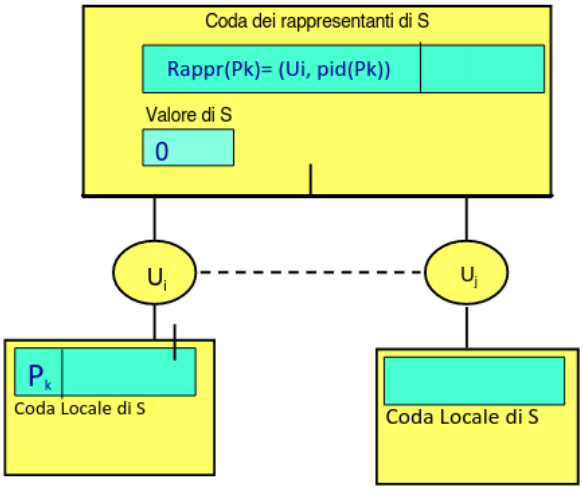


Figura 5.1: Semaforo condiviso con singolo processo in attesa



Esecuzione di una `P` sospensiva su semaforo `S` condiviso

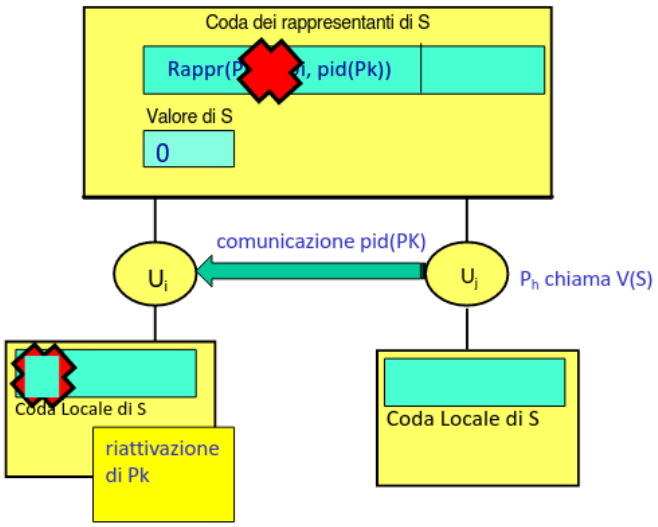
Il nucleo del nodo U_i sul quale opera il processo P_k che ha chiamato `P(S)` provvede a:

- inserire il rappresentante di P_k nella coda globale dei rappresentanti associata ad `S`
- inserire il descrittore di P_i nella coda locale a U_i associata a `S`

Esecuzione di una `V` su un semaforo `S`

Assumendo che la `V(S)` sia chiamata dal processo P_h appartenente al nodo U_j:

- viene eliminato dalla coda dei rappresentanti il primo elemento, processo P_k nodo U_i
- U_j comunica *tempestivamente* a U_i l'identità del processo P_k
- U_i provvede a risvegliare P_k, estraendolo dalla coda locale



5.7 Comunicazione tra nuclei

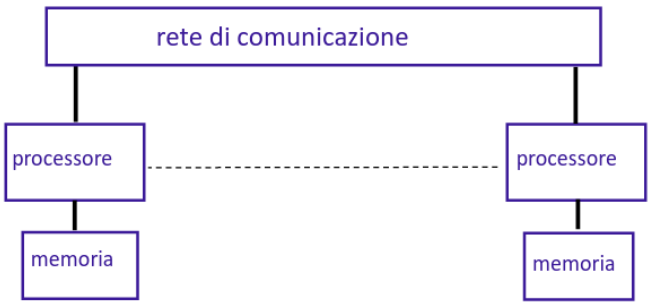
La comunicazione tra due nuclei U_j e U_i deve essere **tempestiva**, occorre interrompere qualsiasi attività su U_i per notificargli della presenza di un processo locale da riattivare: **segnale di interruzione** da U_j a U_i.

5.8 Implementazione semafori - nuclei distinti

```
void P(semaforo sem)  {
    if (/*<sem appartiene a memoria privata>*/)
        //<come in monoprocessore>
    else {
        lock(x);
        //<esecuzione P con eventuale sospensione del rappresentante del processo nella coda di sem>
        unlock(x);
    }
}

void V(semaforo sem) {
    if (/*<sem appartiene a memoria privata>*/)
        //<come in monoprocessore>
    else {
        lock(x);
        if (/*<coda non vuota>*/)
        else {

        }
    }



}
```

6 — Modello a scambio di messaggi

6.1 Caratteristiche del modello



Ogni processo può accedere *esclusivamente* alle risorse allocate nella *propria memoria locale*, ogni risorsa è quindi accessibile da un solo processo, il **gestore della risorsa**.

Se una risorsa è necessaria a più processi, ognuno di questi **processi clienti** deve delegare le operazioni all'unico processo gestore, il quale restituirà gli eventuali risultati.

Ognuno dei processi che necessita di usufruire dei servizi offerti da una risorsa, dovrà **comunicare** con il gestore, il meccanismo base per permettere questa comunicazione è lo **scambio di messaggi**.

6.2 Canali di comunicazione

> Un **canale** è un collegamento logico mediante il quale due o più processi comunicano.

L'astrazione di canale è realizzata dal nucleo della macchina concorrente, come meccanismo primitivo per lo scambio di informazioni.

Il compito del linguaggio di programmazione è quello di fornire gli strumenti linguistici di alto livello per:

- *specificare i canali* di comunicazione
- utilizzarli per esprimere le interazioni tra i processi

6.2.1 Caratteristiche

I parametri che caratterizzano l'astrazione di canale sono:

- **direzione del flusso di dati** che un canale può trasferire
- designazione dei processi **origine** e **destinatario** di ogni comunicazione
- il **tipo di sincronizzazione** tra i processi comunicanti

6.3 Tipi di canale

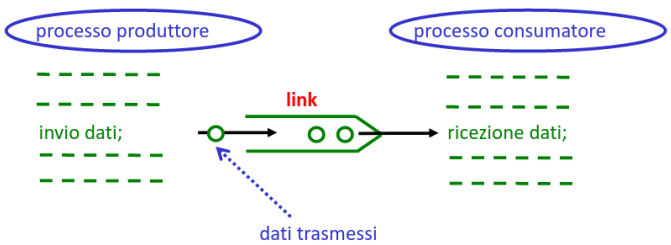
Distinguendo la direzione del *flusso di dati*, si denotano due tipi:

- canale **monodirezionale**, consente il flusso di messaggi in una sola direzione
- canale **bidirezionale**, consente di inviare e ricevere messaggi

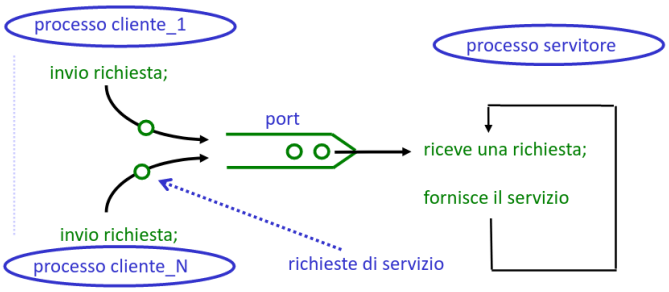
Distinguendo la *designazione dei processi*, si definiscono tre tipi:

- **link**: uno a uno (canale **simmetrico**)
- **port**: molti a uno (canale **asimmetrico**)
- **mailbox**: molti a molti (canale **asimmetrico**

Link



Port



Distinguendo la modalità di **sincronizzazione** tra i processi comunicanti, si individuano tre tipi:

- comunicazione **asincronia**
- comunicazione **sincrona**
- comunicazione con **sincronizzazione estesa**

6.3.1 Comunicazione asincrona

Il processo *continua la sua esecuzione* immediatamente dopo l'invio del messaggio.

Il messaggio viene ricevuto in un istante successivo all'invio, le informazioni contenute nel messaggio *non possono essere attribuite allo stato attuale* del mittente.

L'invio di un messaggio *non è un punto di sincronizzazione* tra mittente e destinatario.

Proprietà

- **carenza espressiva**
- **assenza di vincoli di sincronizzazione** favorisce il grado di concorrenza
- sarebbe necessario un buffer di *capacità illimitata*

Ogni implementazione prevede un limite alla capacità del buffer, in caso di *buffer pieno*, è necessario sospendere il processo che invia il messaggio.

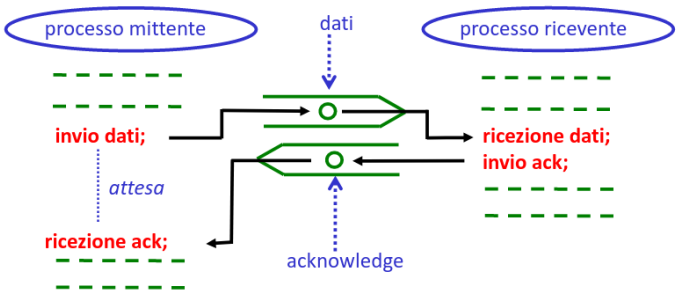
6.3.2 Comunicazione sincrona - randez-vous semplice

Il primo dei due processi comunicanti che esegue l'invio o la ricezione, si **sospende**, in attesa che l'altro esegua l'operazione corrispondente.

L'invio di un messaggio è un *punto di sincronizzazione*, ogni messaggio può essere attribuito allo stato attuale del mittente.

Non è necessaria l'introduzione di un buffer, in quanto un messaggio può essere inviato solo se ildestinatario è pronto a riceverlo.

Realizzazione canale sincrono tramite comunicazioni asincrone

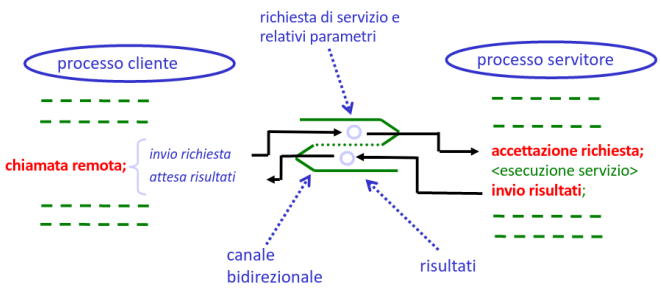


6.3.3 Comunicazione con sincronizzazione estesa - randez-vous esteso

Si assume che ogni messaggio inviato rappresenta una richiesta al destinatario di una esecuzione di *una certa azione*.

Il mittente rimane in attesa fino a che il ricevente non ha terminato di svolgere l'azione richiesta:

- vero e proprio modello **cliente-servitore**
- analogia semantica con **chiamata di procedura**
- riduzione del parallelismo



6.4 Costrutti linguistici e primitive per esprimere la comunicazione

Definizione del canale

```
port <tipo> <identificatore>;
```

`port` viene dichiarato locale a un processo, ed è visibile ai processi mittenti tramite dot notation.

```
<nome processo>.<identificatore canale>
```

6.4.1 Primitive di comunicazione

Send

`send` esprime l'invio di un messaggio

```
send(<valore>) to <porta>;
```

`<porta>` individua il canale a cui inviare il messaggio, `<valore>` è il contenuto del messaggio.

La semantica dipende dal tipo di canale: sincrono o asincrono.

Receive

```
P = receive(<variabile>) from <porta>;
```

- `<porta>` identifica il canale, locale al ricevente, dal quale ricevere il messaggio
- `<variabile>` è l'identificatore della variabile a cui assegnare il valore del messaggio ricevuto

Di default la semantica è **bloccante** in caso di buffer vuoto, con attesa di un messaggio, alcuni linguaggi offrono anche una semantica non bloccante.

6.4.2 Receive bloccante e modello C/S

Il **server** è un processo dedicato a servire le richieste di altri processi (clienti), ognuna rappresentata da un diverso messaggio.

Il server può offrire diversi servizi, ognuno attivato dalla ricezione di un messaggio su un particolare canale, quindi il server gestisce tanti **canali di ingresso** quanti sono i servizi che offre

Nasce il problema della `receive` bloccante, si può infatti ottenere un server bloccato su una `receive` su un canale vuoto, mentre ci sono altri canali non vuoti in attesa di essere serviti.

É necessario implementare una receive con semantica non bloccante: si verifica lo stato del canale e si restituisce un messaggio se presente, altrimenti una segnalazione di canale vuoto.

Sorge tuttavia il problema dell'**attesa attiva** nel caso in cui tutti i canali siano vuoti.

Il meccanismo di ricezione ideale:

- permette al processo server di verificare **contemporaneamente** la disponibilità di messaggi su più canali
- abilita la ricezione di un messaggio da *qualunque canale contenente messaggi*
- quando tutti i canali sono vuoti, blocca il processo in attesa che arrivi un messaggio (su qualunque canale)

6.4.3 Comando con guardia

```
<guardia> -> <istruzione>;
```

dove guardia è costituito dalla coppia

```
(<espressione booleana>; <receive>)
```

L'espressione booleana viene detta **guardia logica**, la receive ha semantica bloccante e viene detta **guardia d'ingresso**.

La valutazione di una guardia ha tre possibili risultati:

- **guardia fallita**: espressione booleana è `false`
- **guardia ritardata**: espressione booleana è `true` ma non ci sono messaggi
- **guardia valida**: espressione booleana `true` ed è presente almeno un messaggio

In caso di

- **guardia fallita**, l'istruzione non viene eseguita

- **guardia ritardata**, il processo viene sospeso e riattivato una volta arrivato un messaggio sulla guardia d'ingresso, si esegue la receive e successivamente l'istruzione
- **guardia valida**, il processo esegue la receive e successivamente l'istruzione

6.4.4 Comando con guardia alternativo

```
select
{
    [] <guardia_1> -> <istruzione_1>;
    [] <guardia_2> -> <istruzione_2>;
    ...
    [] <guardia_n> -> <istruzione_n>;
}
```

Il comando di guardia alternativo racchiude un numero arbitrario di comandi con guardia semplici.

La semantica si basa sulla valutazione delle guardie di tutti i rami, tre casi:

- **una o più guardie valide**: eseguito in maniera **non deterministica** uno dei rami
- **tutte le guardie non fallite sono ritardate**: il processo attende un messaggio che abiliti l'esecuzione di una guardia ritardata
- **tutte le guardie sono fallite**: il comando termina

6.4.5 Comando con guardia ripetitivo

```
do
{
    [] <guardia_1> -> <istruzione_1>;
    [] <guardia_2> -> <istruzione_2>;
    ...
    [] <guardia_n> -> <istruzione_n>;
}
```

Stessa semantica del comando con guardia alternativo ma viene iterato una volta terminato il comando.

Esempio processo servitore

```
processo server {
    port int canale1;
    port real canale2;
    Tipo_di_R R;
    int x;
    real y;
    do {
        [] (cond1); receive (x) from canale1; ->
        {
            R.S1;
            <eventuale restituzione al cliente>;
        }
        [] (cond2); receive (x) from canale2; ->
        {
            R.S2;
            <eventuale restituzione al cliente>;
        }
    }
}
```

6.5 Primitive di comunicazione asincrone

Nel modello a scambio di messaggi, lo strumento di comunicazione di più basso livello è la **send asincrona**, che abilita l'accesso a risorse "condivise" tramite processi servitori.

6.5.1 Problemi di sincronizzazione

Si prendono in considerazione alcuni tipici problemi di interazone nel modello a scambio di messaggi con send asincrona per realizzare un gestore di risorse dei tipi:

- una sola operazione
- più operazioni mutuamente esclusive
- più operazioni con condizioni di sincronizzazione

Si individuano ora schemi di soluzioni alle diverse problematiche.

Esempio - risorsa condivisa con una sola operazione

La risorsa viene gestita da un processo server, che offre un unico servizio senza condizioni di sincronizzazione:

```
process cliente {
    port tipo_out risposta;
    tipo_in a;
    tipo_out b;
    process p;
    ...
    send(a) to server.input;
    p = receive(b) from risposta;
    ...
}
```

```
tipo_out fun(tipo_in x);
process server {
    port tipo_in input;
    tipo_var var;
    process p;
    tipo_in x;
    tipo_out y;
    <inizializzazione>;
    while (true) {
        p = receive (x) from input;
        y = fun(x);
        send (y) to p.risposta;
    }
}
```

Esempio - risorsa condivisa con più operazioni

La risorsa è gestita da un processo server che offre due servizi senza condizioni di sincronizzazione.

Soluzione senza comandi guardia

```
typedef struct {
    enum (fun1, fun2) servizio;
    union {
        tipo_in1 x1; //parametri fun1
        tipo_in2 x2; //parametri fun2
    } parametri
} in_mess;
```

```
tipo_out1 fun1 (tipo_in1 x1);
tipo_out2 fun2 tipo_in2 x2);

process server {
    port in_mess input;
    tipo_var var;
    process p;
    in_mes richiesta;
    tipo_out1 y1;
    tipo_out2 y2;

    while (true) {
        p = receive(richiesta) from input;
        switch (richiesta.servizio) {
            case fun1: {
                y1 = fun1(richiesta.parametri.x1);
                send (y1) to p.risposta1;
                break;
```

```
            }
            case fun2: {
                y2 = fun2(richiesta.parametri.x2);
                send (y2) to p.risposta2;
                break;
            }
        }
    }
}
```

```
process cliente {
    port tipo_out1 risposta1;
    port tipo_out2 risposta2;
    tipo_in1 a1;
    tipo_in2 a2;
    in_mes M;
    tipo_out1 b1;
    tipo_out2 b2;
    process p;
    <inizializzazione M, in base al servizio scelto>;
    if (<servizio1>) then {
        send(M) to server.input;
        p = receive (b1) from risposta1;
    }
    else {
        send(M) to server.input;
        p = receive (b2) from risposta2;
    }
    ...
}
```

Soluzione con comandi guardia, un canale per ogni servizio.

```
tipo_out1 fun1(tipo_in1 x1);
tipo_out2 fun2(tipo_in2 x2);
process server {
    port tipo_in1 input1; //canale per le richieste relative a fun1
    port tipo_in2 input2; //canale per le richieste relative a fun2
    tipo_var var;
    process p;
    tipo_in1 x1;
    tipo_in2 x2;
    tipo_out1 y1;
    tipo_out2 y2;
    <eventuale inizializzazione>;
    do {
        [] p = receive (x1) from input1; ->
            y1=fun1(x1);
```

```
            send (y1) to p.risposta1;

        [] p = receive (x2) from input2; ->
            y2=fun2(x2);
            send (y2) to p.risposta2;
    }
}
```

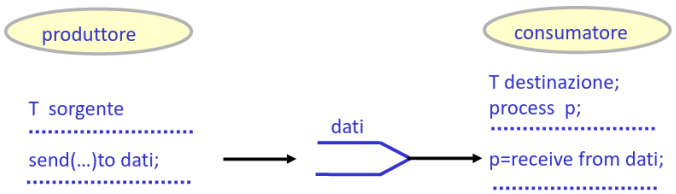
```
process cliente {
    port tipo_out1 risposta1;
    port tipo_out1 risposta1;
    tipo_in1 a1;
    tipo_in2 a2;
    tipo_out1 b1;
    tipo_out2 b2;
    process p;
    <inizializzazione a1 o a2>
    if (<servizio1>) then {
        send(a1)to server.input1;
        p=receive(b1)from risposta1;
    }
    else {
        send(a2)to server.input2;
        p=receive(b2)from risposta2;
    }
    ...
}
```

GUARDA ESEMPI 50-67

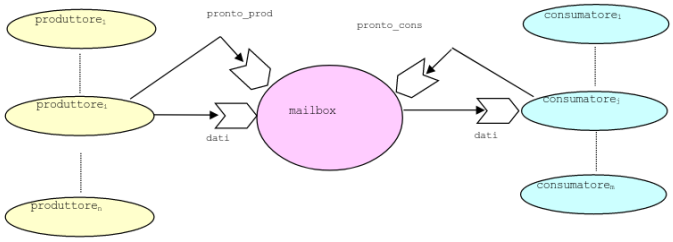
6.6 Primitive di comunicazione sincrone

Con primitive sincrone si ottiene un minore grado di concorrenza rispetto alle asincrone, tuttavia non è necessaria la presenza di buffer nei canali di comunicazione.

Scambio dato - singolo produttore, singolo consumatore



6.6.1 Mailbox di dimensioni finite (protocolli simmetrici)



É necessaria una struttura interna al processo mailbox, una **coda di messaggi**.

Non verrà dettagliata la realizzazione del tipo astratto `coda_messaggi`, si suppone di poter operare su questa struttura dati con le seguenti operazioni.

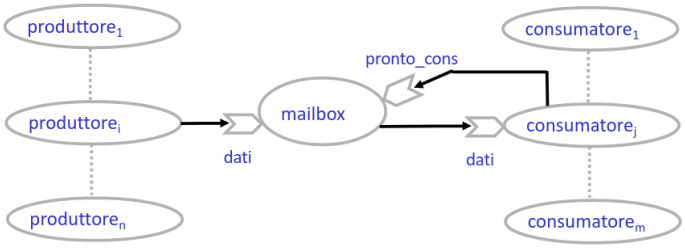
```
void inserimento(T mes);

T estrazione();

boolean piena;

boolean vuota();
```

```
process mailbox {
    port T dati;
    port signal pronto_prod, pronto_cons;
    T messaggio;
    process P;
    signal s;
    coda_messaggi coda;
    <inizializzazione>;

    do {
        [] (!coda.piena()); p = receive(s) from pronto_prod;
            ->  p = receive(messaggio) from dati;
                coda.inserimento(messaggio);
        [] (!coda.vuota()); p = receive(s) from pronto_cons;
            ->  messaggio = coda.estrazione;
                send(messaggio) to p.dati;
    }

    process produttore_i {
        T messaggio;
        signal s;
        //...
        <produci messaggio>;
        send(s) ot mailbox.pronto_prod;
        send(messaggio) to malbox.dati;
```

```
        //...
    }

    process consumatore_i {
        port T dati;
        T messaggio;
        process p;
        signal s;
        //...
        send(s) to mailbox.pronto_cons;
        p = receive(messaggio) from dati;
        <consuma messaggio>;
        //...
    }
```

6.6.2 Mailbox di dimensioni finite

É possibile ottimizzare il protocollo (lato produttore)



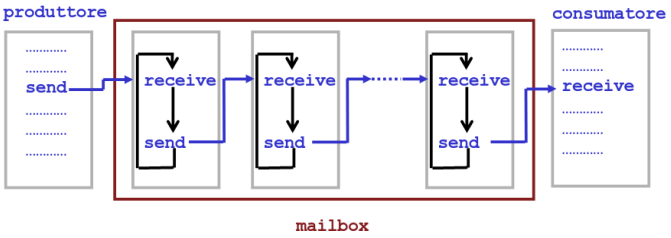

```
process mailbox {
    port T dati;
    port signal pronto_cons;
    T messaggio;
    process p;
    signal s;
    coda_messaggi coda;
    <inizializzazione>;
    do {
        [] (!coda.piena()); p = receive(messaggio) from dati;
            ->  coda.inserimento(messaggio);
        [] (!coda.vuota()); p = receive(s) from pronto_cons;
            ->  messaggio = coda.estrazione;
                send(messaggio) to p.dati;
    }
}

process produttore_i {
    T messaggio;
```

```
    process p;
    signal s;
    //...
    <produci il messaggio>;
    send(messaggio) to mailbox.dati;
    //...
```

6.6.3 Mailbox concorrente

Un buffer di dimensione N può essere implementato da un insieme di processi concorrenti collegati secondo uno schema a cascata (*pipeline*).

Ogni process ha un struttura ciclica e, sfruttando la semantica sincrona della send, può trattenere un singolo messaggio inserito, ma non ancora estratto.



Struttura del processo i-esimo nella pipeline:

```
process m_i { //(0 <= i <= N-2)
    port T dati;
    T buffer;
    process p;
    while (true) {
        p = receive(buffer) from dati;
        send(buffer) to m_i+1.dati;
    }
```

Specifica di strategie di priorità (pool con una risorsa)

```
process server {
    port signal richiesta;
    port int rilascio;
    boolean libera;
    process P;
    signal s;
    int sospesi = 0; boolean bloccato[M];
    process client[M];

    { //inizializzazione
    libera ? true;
    for (int j = 0; j<M; j++)
```

```
        bloccato[j] = false;
    client[0] = "P_0";
    //...
    client[M-1] = "P_M-1";
    }

    do {
        [] p = receive(s) from richiesta; ->
            if (libera) {
                libera = false;
                send (s) to p.risorsa;
            }
            else {
                sospesi ++;
                int j = 0;
                while (client[j] != p) j++;
                bloccato[j] = true;
            }
        [] p = receive (s) from rilascio; ->
            if (sospesi == 0)
                libera = true;
            else {
                int i = 0;
                while (!bloccato[i]) i++;
                sospesi--;
                bloccato[i] = false;
                send (s) to client[i].risorsa;
            }
    }
```

```
process cliente {
    port int risorsa;
    signal s;
    //...
    send(s) to server.richiesta;
    p = receive (s) from risorsa;
    <uso della risorsa r-esima>;
    send(s) to server.rilascio;
    //...
}
```

6.7 Realizzazione meccanismi di comunicazione

6.7.1 Realizzazione delle primitive asincrone

Le *primitive di comunicazione* e del *costrutto port* sono realizzati utilizzando gli strumenti di comunicazione offerti dal nucleo del sistema operativo.

Le **primite asincrone** sono più "primitive", in quanto le primitive sincrone sono realizzate utilizzando combinazioni appropriate di primitive asincrone.

6.7.2 Architetture mono e multielaboratore

Ipotesi (semplificate):

- tutti i messaggi scambiati tra i processi sono di un *unico tipo* T predefinito a livello di nucleo
- tutti i canali sono **da molti a uno (port)** e quindi associati al processo ricevente
- essendo le primitive asincrone, ogni porta deve contenere un **buffer** (coda di messaggi) di *lunghezza indefinita*

CODICE 85-91

FINISCI 92-106

7 — Comunicazione con sincronizzazione estesa

7.1 Semantica

La comunicazione con sincronizzazione estesa è un modello di **comunicazione/estesa** in cui:

- il processo mittente richiede l'esecuzione di un **servizio** al processo destinatario
- il processo mittente rimane **sospeso** fino al competamente del servizio richiesto

I processi rimangono *sincronizzati* durante l'esecuzione del servizio da parte del ricevente fino alla *ricezione dei risultati* da parte del mittente.

Il meccanismo è noto anche con il nome di **chiamata di operazione remota**.

Esiste una forte analogia con una classica chiamata a funzione, dove il chiamante attende il completamento dell'operazione prima di proseguire.

La differenza sostanziale è che l'operazione viene eseguita **remotamente** da un processo diverso dal chiamante.

7.2 Implementazione

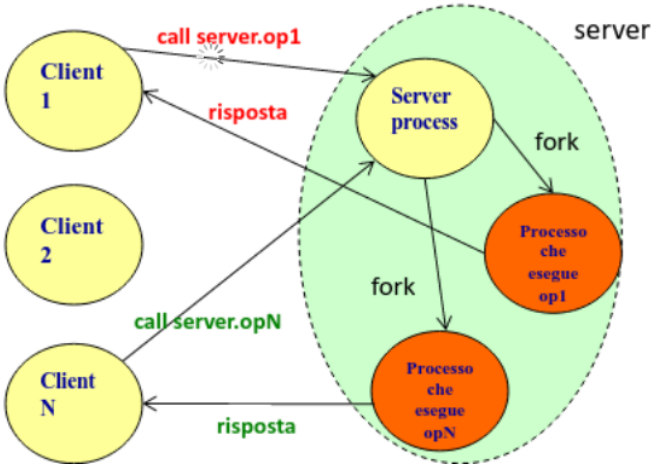
Esistono due modalità diverse di implementazione lato server:

- chiamata di procedura remota **RPC**
- **rendez-vous esteso**

7.2.1 Chiamata di procedura remota

Per ogni operazione che un processo client può ricevere viene dichiarata, lato server, una **procedura**.

Per ogni richiesta di operazione al server, viene creato un **nuovo processo** (thread) che ha il compito di eseguire la procedura corrispondente.

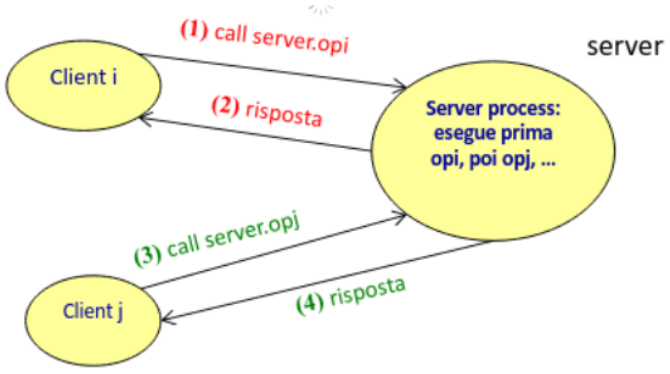


7.2.2 Rendez vous

L'operazione richiesta viene specificata come un **insieme di istruzioni** che può comparire in un punto qualunque del processo servitore (linguaggio ADA).

Il processo servitore utilizza un'istruzione di input (`accept`) che lo sospende in attesa di una richiesta dell'operazione.

All'arrivo della richiesta il servitore esegue il corretto insieme di istruzioni e invia i risultati al chiamante.



7.2.3 RPC vs rendez-vous

RPC

Rappresenta solo un *meccanismo di* ***comunicazione*** *tra processi*: la possibilità che più operazioni siano eseguite concorrentemente implica la necessità di sincronizzazione tra i vari processi servitori (a carico del programmatore).

Rendez-vous esteso

Combina **combinazione** con **sincronizzazione**, esiste un solo processo servitore al cui interno sono definite le istruzioni che consentono di realizzare il servizio richiesto. Il processo servitore si sincronizza con il processo cliente quando esegue l'operazione di `accept`.

7.2.4 Chiamata di procedura remota

L'insieme delle procedure remote è definito all'interno di un componente software, **modulo**, che contiene anche le variabili locali al server ed eventuali procedure e processi locali.

```
//server
module nome_del_modulo {
    <dichiarazione delle variabili locali>;
    <inizializzazione delle variabili locali>;
    public void op1 (<parametri formali>){
    <corpo della procedura op1>;}
    ...
    public void opn (<parametri formali>){
    <corpo della procedura opn >;}
    <dichiarazione di procedure locali>;
    <dichiarazione di processi locali>;
}
```

I singoli moduli operano su *spazi di indirizzamento diversi* e possono quindi essere *allocati su nodi distinti della rete*.

La chiamata di una procedura remota verrà specificata dal client con uno statement del tipo:

```
module client {
    //...
    call nome_del_modulo.op_i (<parametri attuali>);
    //...
```

A ogni istante è possibile che più *thread concorrenti* all'interno del modulo server accedano a variabili interne, è quindi necessaria **sincronizzazione** (monitor, semafori...).

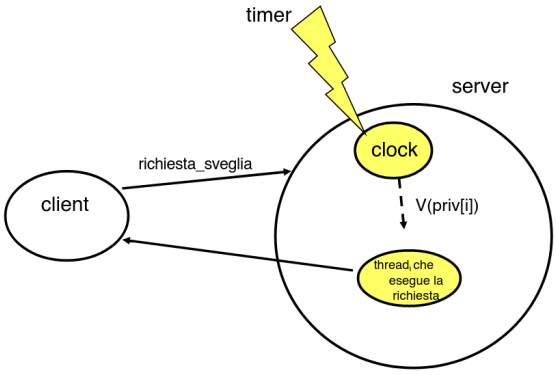
Esempio - servizio di sveglia

Server

```
module allarme {
    int time;
    semaphore mutex = 1;
    semaphore priv[N] = 0; //sem privati sospensione processi
    coda_richieste coda; //struttura richieste di sveglia

    public void richiesta_sveglia(int timeout, int id) {
        int sveglia = time + timeout;
        P(mutex);
        <inserimento sveglia e id nella coda di risveglio
        matenendo ordinata con id non decrescenti>
        V(mutex);
        P(priv[id]); //attesa sveglia
    }

    process clock { //demone
        int tempo_di_sveglia;
        <avvia il clock>;
        while (true) {
            <attende interruzione, quindi riavvia il clock>;
            time++;
            P(mutex);
            tempo_di_sveglia = <min valore di sveglia in coda>;
            while (time >= tempo_di_sveglia) {
                <rimozione di tempo_di_sveglia e id da coda>;
                V(priv[id]);
            }
            V(mutex);
        }
    }
}
```

Client

```
call allarme.richiesta_sveglia(60); //crea thread dedicato nel server
```



7.2.5 Rendez vous esteso

Il servizio richiesto viene specificato dal server come un insieme di istruzioni che può comparire in un qualsiasi punto del processo del server tremite una `accept`.

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

Il client richiede il servizio con l'istruzione

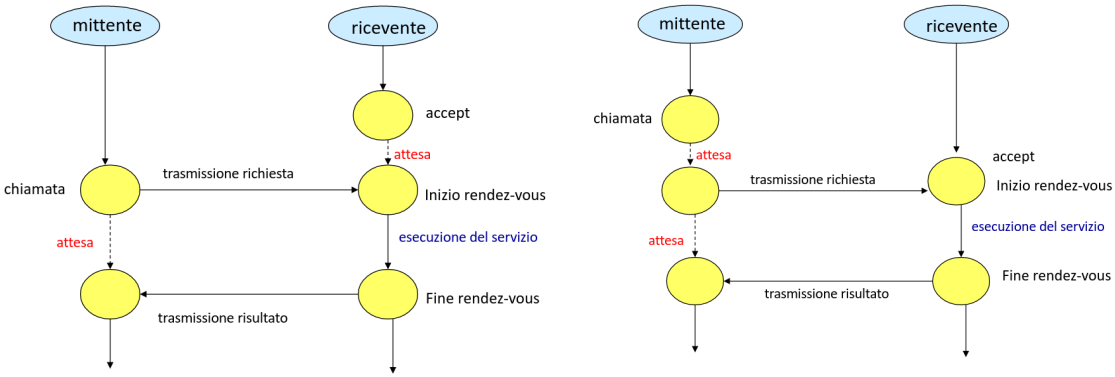
```
call <server>.<servizio>(<parametri attuali>);
```

Accept

Semantica dell'operazione accept:

- se non sono presenti richieste del servizio S è sospensiva
- se S viene richiesto da più processi, vengono evasi in politica FIFO
- a uno stesso servizio possono essere associate più accept nel codice eseguito dal server: a un richiesta possono corrispondere azioni diverse
- lo schema di comunicazione è di tipo **asimmetrico molti a uno**

Possibili sequenze di comunicazione



Selezione delle richieste:

Nel modello rendez-vous le richieste vengono scelte a seconda del suo stato interno, utilizzando comandi con guardia:

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

Esempio - produttore e consumatore, buffer capacità N

```
process buffer { //server
    messaggio buff[N];
    int testa = 0, coda = 0;
    int cont = 0;
    do {
        [] (cont < N); accept inserisci(in dato:messaggio) ->
            { buff[coda] = dato; } //fine rendez-vous
            cont++;
            coda = (coda + 1) % N;
        [] (cont > 0); accept preleva(out dato:messaggio) ->
            { dato = buff[testa]; } //fine rendez-vous
            cont--;
            testa = (testa + 1) % N;
    }
}
```

La sincronizzazione tra client e server è limitata alle sole *istruzioni comprese nel blocco di accept* (dentro {...}).

```
process produttore_i {
    messaggio dati;
    for (;;) {
        <produci dati>;
        call buffer.inserisci(dati);
    }
}

process consumatore_j {
    messaggio dati;
    for (;;) {
        call buffer.preleva(dati);
        <consuma dati>;
    }
}
```

Selezione delle richieste in base ai parametri di ingresso

La decisione di servizio può dipendere anche da **parametri** della richiesta stessa, oltre che allo stato della risorsa.

Ciò non è possibile tramite la guardia di ingresso: è necessaria una **doppia interazione** tra client e server, prima per i parametri e poi per il servizio.

Nell'ipotesi di un *numero limitato di differenti richieste* si può ottenere una semplice soluzione al problema associando ad ogni possibile richiesta una differente operazione.

In alcuni linguaggi è possibile aggregare richieste possibili in strutture di tipo vettore: **vettore delle operazioni di servizio**.

Esempio - sveglia

Si consideri il caso del server allarme con il compito di inviare una segnalazione di sveglia ad un insieme di processi che richiedono il servizio con un tempo da essi stabilito.

Server, tre tipi di richiesta:

- **tick**: aggiornamento del tempo
- **richiesta_di_sveglia(T)**: impostazione sveglia per il mittente
- **svegliami[T]**: attesa del segnale di allarma al tempo specificato

L'ordine con cui il processo allarme risponde alle richieste del tipo `svegliami` dipende solo dal parametro T comunicato con la richiesta.

Struttura generica cliente:

```
process client_i {
    //...
    allarme.richiesta_di_sveglia(T);
    //...
    allarme.svegliami[T];
    //...
}
```

Vettore di operazioni di servizio:

```
typedef struct {
    int risveglio;
    int intervallo;
} dati_di_risveglio;

dati_di_risveglio tempo_di_sveglia[N];
```

Server:

```
process allarme {
    entry tick;
    entry richiesta_di_sveglia(in int intervallo);
    entry svegliami[first..last]; //vettore operazioni

    int tempo;
    typedef struct {
        int risveglio;
```

```
        int intervallo;
    } dati_di_risveglio;
    dati_di_risveglio tempo_di_sveglia[N];

    do {
        [] accept tick; -> {tempo++;}
        [] accept richiesta_di_sveglia(in int intervallo)
            ->
```

8 — Algoritmi di sincronizzazione distribuiti

Il modello a scambio di messaggi è la naturale astrazione di un sistema distribuito, nel quale processi distinti eseguono su nodi fisicamente separati e collegati tra loro attraverso una rete.

Caratteristiche sistemi distribuiti:

- concorrenza/parallelismo delle attività dai nodi
- assenza di risorse condivise tra nodi
- assenza di un clock globale
- possibilità di malfunzionamenti indipendenti
  - nei nodi (crash, attacchi...)
  - nella rete di comunicazione

Le proprietà desiderabili in sistemi di questo tipo:

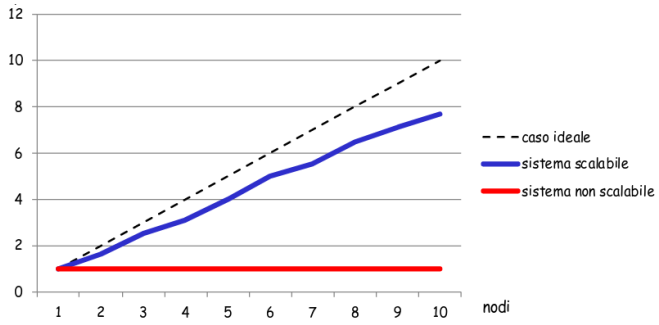
- **scalabilità**: le prestazioni dovrebbero aumentare al crescere del numero di nodi utilizzati
- **tolleranza ai guasti**: capacità di funzionare anche in presenza di guasti

8.0.1 Prestazioni

Un indicatore per misurare le prestazioni di un sistema parallelo è dato dallo **speedup**.

Ipotizzando che un'applicazione possa essere eseguita su un numero variabile di nodi, lo speedup è una funzione del numero di nodi.

```
speedup(n) = tempo(1) / tempo(n)
```



Il grafico traccia la relazione tra speedup e numero di nodi n, lo speedup ideale è

```
speedup(n) = tempo(1) / tempo(n) = n
```

8.0.2 Tolleranza ai guasti

Un sistema distribuito tollerante ai guasti riesce a erogare i propri servizi anche in presenza di guasti su uno o più nodi.

Tipi di guasti:

- transiente
- intermittente
- persistente

La tolleranza ai guasti può essere conseguita con tecniche di **ridondanza**, vengono mantenute più istanze degli stessi componenti, in modo da poter rimpiazzare l'elemento guasto con un elemento equivalente.

É necessario prevedere meccanismi per

- **rilevazione** dei guasti (*fault detection*)
- **ripristino** del sistema dopo ogni guasto rilevato (*recovery*)

8.0.3 Algoritmi di sincronizzazione

Come nel modello a memoria comune, anche nel modello a scambio di messaggi è necessario prevedere algoritmi che permettano la sincronizzazione tra processi concorrenti, per risolvere eventuali problematiche.

Ad esempio:

- **timing**: sincronizzazione dei clock e tempo logico
- **mutua esclusione** distribuita
- elezione di **coordinatori** in gruppi di processi

É desiderabile che questi algoritmi godano delle proprietà di scalabilità e tolleranza ai guasti.

8.1 Algoritmi per la gestione del tempo

8.1.1 Tempo nei sistemi distribuiti

In un sistema distribuito, ogni nodo è dotato di un proprio orologio, se due orologi locali a due nodi non sono sincronizzati, è possibile che se un evento `e2` accade nel nodo `N2` dopo un altro evento `e1` nel nodo `N1`, ad `e2` sia associato un istante temporale precedente quello di `e1`.

In applicazioni distribuite può essere necessario avere un **unico riferimento temporale**, condiviso da tutti i partecipanti.

Realizzabile con:

- un **orologio fisico universale**: a questo scopo vengono utilizzati degli algoritmi di sincronizzazione dei nodi del sistema (algoritmo di Berkley, algoritmo di Cristian) che garantiscono che il tempo misurato da ogni clock su ogni nodo sia lo stesso
- un **orologio logico**, che permetta di associare a ogni evento un istante logico coerente con l'ordine in cui essi si verificano

8.1.2 Orologi logici

Relazione di precedenza tra eventi, Happened Before -¿

- se `a` e `b` sono eventi in uno stesso processo e `a` si verifica prima di `b`: `a->b`
- se `a` è l'evento di **invio di un messaggio** e `b` è l'evento di **ricezione** dello stesso messaggio: `a->b`
- se `a->b` e `b->c` allora `a->c`

Data una coppia di eventi `(a,b)` sono possibili tre casi

- `a->b`
- `b->a`
- `a` e `b` non sono legati dalla relazione **HB**, ovvero sono concorrenti

Assumendo che ad ogni evento venga associato un **timestamp** `C(e)`, si vuole definire un modo per misurare il concetto di tempo tale per cui ad ogni evento `a` possiamo associare un timetamp `C(a)` sul quale tutti i processi sono d'accordo.

I timestamp devono soddisfare la seguente proprietà

se `a->b` allora `C(a) < C(b)`

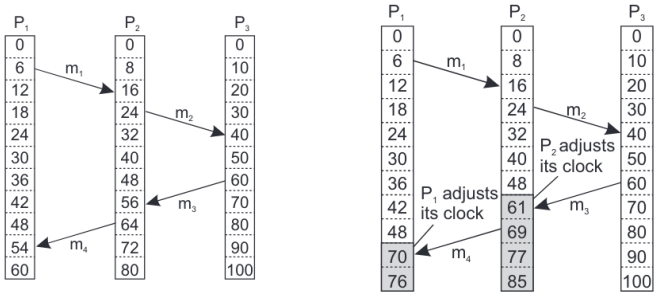
Algoritmo di Lamport

Per garantire il rispetto della proprietà, ogni processo P_i gestisce localmente un **contatore** del tempo logico C_i, che viene gestito nel modo seguente:

- ogni nuovo evento all'interno di P_i provoca un incremento del valore di C_i: $C_i = C_i + 1$
- ogni volta che P_i invia un messaggio `m`, il contatore C_i viene incrementato: $C_i = C_i + 1$ e successivamente al messaggio viene associato il timestamp C_i: `ts(m)` $= C_i$
- quando un processo P_j riceve un messaggio `m`, assegna al proprio contatore C_j un valore uguale al massimo tra C_j e `ts(m)`: $C_j = \max\{C_j, \texttt{ts(m)}\}$ e successivamente lo incrementa di 1: $C_j = C_j + 1$

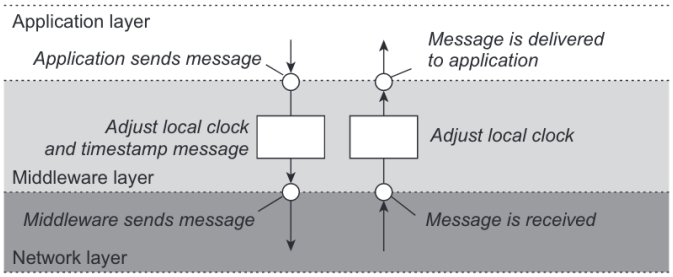
Esempio

Consideriamo tre processi in esecuzione su tre nodi, ognuno con il proprio clock (frequenze diverse):



Lamport: aggiornamenti degli orologi

Nei sistemi distribuiti, l'algoritmo di Lamport viene generalmente eseguito da uno strato di software (*middleware)* che interfaccia i processi alla rete: i processi vedono solo il tempo logico.



8.2 Mutua esclusione in sistemi distribuiti