Sistemi Operativi M

Federico Andrucci

September 2021

Contents

1 Virtualizzazione

Virtualizzare un sistema (hardware e software) significa presentare all'uilizzatore una visione delle risorse del sistema diversa da quella reale. Ciò è possibile introducendo un **livello di indirezione** tra la vista logica e quella fisica delle risorse.
Quindi l'obiettivo della virtualizzazione è quello di disaccoppiare il comportamento delle risorsedi un calcolatore dalla loro realizzazione fisica. Quindi apparendo diverse da quelle effettive della macchina. Il software che si occupa di virtualizzare in parole semplici divide le risorse reali nel numero di macchine virtuali necessarie. Quindi ogni macchina virtuale avrà la sua CPU, GPU, RAM, ecc...
Esmpi di virtualizzazione:

- **Virtualizzazione a livello di processo:** i sistemi multitasking permettono l'esecuzione contemporanea di più processi, ognuno dei quale dispone di una macchina virtuale dedicata. Questo tipo di virtualizzazione viene realizzata dal kernel del sistema operativo.
- **Virtualizzazione della memoria:** in presenza di memoria virtuale, ogni processo vede uno spazio di indirizzamento di dimensioni indipendenti dallo spazio fisico effettivamente a dispozione. Anche questa virtualizzaione è realizzata dal kernel.
- **Astrazione:** un oggetto astratto (risorsa virtuale) è la rappresentazione semplificata di un oggetto (risortsa fisica), quindi esibendo le proprietà significative per l'utilizzatore e nascondendo i dettagli realizzativi non importanti

1.1 Virtualizzazione di un Sistema di Elaborazione

Tramite la virtualizzazione una singola piattaforma hardware viene condivisa da più elaboratori virtuali, ognuno gestito da un proprio sistema operativo. Il disaccoppiamento viene realizzato dal **Virtual Machine Monitor (VMM)**, il cui compito è quello di consentire la condivisione da parte di più macchine virtuali di una singola piattaforma hardware.
Quindi il **VMM** è il **mediatore unico** nelle interazioni tra le macchine virtuali e l'hardware, il quale garantisce: **isolamento tra le VM** e **stabilità del sistema**.

1.2 Tecniche del VMM

1.2.1 Emulazione

L'emulazione è l'insieme di tutti quei meccanismi che permettono l'esecuzione di un programma compilato su un determiato sistema di girare su un qualsiasi altro sistema differente da quello nel quale è stato compilato. Quindi vengono emulate interamente le singole istruzioni dell'architettura ospitata.

I vantaggi dell'emulazione sono l'interoperabilità tra ambienti eterogenei, mentre gli svantaggi sono le ripercussioni sulle performances.

Esistono principalmente due tecniche di emulazione: **interpretazione** e **ricompilazione dimanica**.

Interpretazione:
L'interpretazione si basa sulla lettura di ogni singola istruzione del codice macchina che deve essere eseguito e sulla esecuzione di più istruzioni sull'host virtualizzante. Produce un sovraccarico elevento in quanto potrebbero essere necessarie molte istruzioni dell'host per interpretare una singola istruzione sorgente.

Compilazione dinamica:
Invece di leggere una singola istruzione del sistema ospitato, legge interi blocchi di codice, vengono analizzati, tradotti per la nuova architettura, ottimizzati e messi in esecuzione. Il vantaggio in termini prestazionali rispetto all'interpretazione è notevolmente maggiore.
Ad esempio parti di codice utilizzati frequentemente vengono bufferizzati nella cache per evitare di doverli ricompilare in seguito.
.. .. ..

1.3 Realizzaione del VMM

Requisiti di Popek e Goldberg del 1974:

- **Ambiente di esecuzione per i programmi sostanzialmente identico a quello della macchina reale**: Gli stessi programmi che eseguono nel sistema non virtualizzato possono essere eseguiti nelle VM senza modifiche e problemi.
- **Garantire un'elevata efficienza nell'esecuzione dei programmi**: Il VMM deve permettere l'esecuzione diretta delle istruzioni impartite dalle macchine virtuali, quindi le istruzioni non privilegiate vengono eseguite direttamente in hardware senza coinvolgere il VMM
- **Garantire la stabilità e la sicurezza dell'intero sistema**: Il VMM deve sempre rimanere sempre nel pieno controllo delle risorse hardware, e i programmi in esecuzione nelle macchine virtuali non possono accedere all'hardware in modo privilegiato

Parametri e classificazione

- **Livello** nel quale è collocato il VMM:
  - **VMM di sistema**: eseguono direttamente sopra l'hardware del elaboratore (vmware, esx, xen, kvm)

  - **VMM ospitati**: eseguiti come applicazioni sopra un S.O. esistente (parallels, virtualbox)

- **Modalità di dialogo**: per l'accesso alle risorse fisiche tra le macchine virtuali ed il VMM:
  - **Virtualizzazione pura** (vmware): le macchine virtuali usano la stessa interfaccia dell'architettura fisica
  - **Paravirtualizzazione** (xen): il VMM presenta un'interfaccia diversa da quella dell'architettura HW

1.3.1 Ring di protezione

La CPU prevede due livelli di protezione: **supervisore o kernel (0)** e **utente (<0)**.
Ogni ring corrisponde a una diversa modalità di funzionamento del processore:

- a livello 0 vengono eseguite le istruzioni privilegiate della CPU
- nei ring di livello superiore a 0 le istruzioni privilegiate non vengono eseguite

Alcuni progrmmi sono progettati per eseguire nel ring 0, ad esempio il Kernel del S.O. infatti è l'unico componente che ha pieno controllo dell'hardware.

VMM (vmm di sistema)
In un sistema virtualizzato il VMM deve essere l'unica componente in grado di mantenere il controllo completo dell'hardware. Infatti solo il VMM opera nello stato supervisore, mentre il S.O. e le applicazioni eseguono in un ring di livello superiore.
Sorgono però due problemi:

- **Ring deprivileging**: il s.o. della macchina virtuale esegue in un ring che non gli è proprio
- **Ring compression**: se i ring utilizzati sono solo 2, applicazioni e s.o. della macchina virtuale eseguono allo stesso livello: scarsa protezione tra spazio del s.o. e delle applicazioni.

1.3.2 Ring Deprivileging

Con Ring Deprivilenging si indica una situazione nel quale l'esecuzione di istruzioni privilegiate richieste dal sistema operativo nell'ambiente guest non possono essere eseguite in quanto richiederebbero un ring 0, ma il kernel della macchina virtuale esegue in un ring di livello superiore (foto telefono 1)
Una possibile prima soluzione è il **Trap & Emulate**: nel quale se il guest tenta di eseguire un'istruzione privilegiata

- la CPU notifica un'eccezione al VMM (**trap**) e gli trasferisce il controllo

- il VMM controlla la correttezza dell'operazione richiesta e ne emula il comportamento (**emulate**)

Quindi in poche parole la CPU notifica e delega al VMM il controllo e l'esecuzione dell'istruzione privilegiata.
Esempio:
Il guest tenta di disabilitare le interruzioni (popf), se la richiesta della macchina virtuale fosse eseguita direttamente sulla CPU sarebbero disabilitati tutti gli interrupt di sistema e quindi il VMM non potrebbe riottenere il controllo. Invece, con Trap&Emulate riceve la notifica di tale richiesta e ne emula il comportamento sospendendo gli interrupt solamente per la macchina virtuale richiedente.

Supporto HW alla virtualizzazione
L'archietettura della CPU si dice **naturalmente virtualizzabile** se e solo se prevede l'invio di trap al VMM per ogni istruzione privilegiata invocata da un livello di protezione differente dal quello del VMM.
Se la CPU è naturalmente virtualizzabile viene implementato il trap&emulate, altrimenti, se non è virtualizzabile vi sono 2 possibilità: **Fast Binary Translation** e **Paravirtualizzazione**.

1.3.3 Fast Binary Translation

Il VMM scansiona dinamicamente il codice dei sistemi operativi guest prima dell'esecuzione per sostituire a run time i blocchi contenenti istuzioni privilegiate in blocchi equivalenti dal punto di vista funzionale e contenenti chiamate al VMM. Inoltre i blocchi tradotti sono eseguiti e conservati in cache per eventuali riusi futuri. (SISTEMARE)
(immagine slide 33)
Il principale limite della Fast Binary Translation è che la traduzione dinamica è molto costosa. Però, con questa tecnica, ogni macchina virtuale è una esatta copia della macchina fisica, con la possiblità di installare gli stessi s.o. di architetture non virtualizzate.

1.3.4 Paravirtualizzazione

Il VMM (hypervison) offre al sistema operativo guest un'interfaccia virtuale (ovviamente differente da quello hardware del processore) chiamata **hypercall API** alla quale i s.o. guest devono rifersi per avere accesso alle risorse (system call).
Queste Hypercall API permettono di:

- richiedere l'esecuzione di istruzioni privilegiate, senza generare un interrupt al VMM
- i kernel dei s.o. guest devono quindi essere modificati per avere accesso all'interfaccia del particolare VMM

- la struttura del VMM è semplificata perchè non deve più preoccuparsi di tradurre dinamicamente i tentativi di operazioni privilegiate dei s.o. guest

Le prestazioni rispetto alla Fast Binary Translation sono notevolmente superiori, però ovviamente c'è una necessità di porting dei dei s.o. guest (non sempre facile).
(aggiungere protezione processore)

2 La protezione nei Sistemi Operativi

Sicurezza: riguarda l'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 di dati.

Protezione: insieme di attività volte a garantire il controllo dell'accesso alle risorse logiche e fisiche da parte degli utenti autorizzati all'uso di un sistema di calcolo.
La sicurezza mette a disposizione meccanismi di **identificazione, autenticazione, ...**
Per rendere un sistema sicuro è necessario stabilire per ogni utente autorizzato:

- quali siano le risore alle quali può accedere
- con quali operazioni può accedervi

Tutto ciò è stabilito dal sistema di protezione attraverso delle tecniche di controllo dell'accesso.
In un sistema il controllo degli accessi si esprime tramite la definizione di tre livelli concettuali:

- modelli
- politiche
- meccanismi

Modelli:
Un modello di protezione definisce i soggetti, gli oggetti e i diritti d'accesso:

- **oggetti**: costituiscono la parte passiva, cioè le risorse fisiche e logiche alle quali si può accedere e su cui si può operare.
- **soggetti**: rappresentano la parte attiva di un sistema, cioè le eintità che possono richiedere l'accesso alle risorse (utenti e processi)
- **diritti d'accesso**: sono le operazioni con le quali è possibile operare sugli oggetti

(Un soggetto può avere diritti d'accesso sia per gli oggetti che per gli altri soggetti)
Ad ogni soggetto è associato un **dominio di protezione**, che rappresenta l'ambiente di protezione nel quale il soggetto esegue. Quindi il dominio indica i diritti d'accesso posseduti dal sogetto nei confronti di ogni risorsa.
Un dominio di protezioen è unico per ogni soggetto, mentre un processo può eventualmente cambiare dominio durante la sua esecuzione.

Politiche:
Le **politiche di protezione** definiscono le regole con le quali i soggetti possono accedere agli oggetti Classificazione delle politiche:

- **discretional access control (DAC)**: il creatore di un oggetto controlla i diritti di accesso per quell'oggetto (unix). La definizione delle politiche è decentralizzata.
- **mandatory access control (MAC)**: i diritti di accesso vengono definiti in modo centralizzato. Ad esempio in installazioni di alta sicurezza
- **role based access control (RBAC)**: ad un ruolo sono assegnati specifici diritti di accesso sulle risorse. Sli utenti possono appartenere a diversi ruoli. I diritti attribuiti ad ogni ruolo vengono assegnati in modo centralizzato

Principio del privilegio minimo: ad ogni soggetto sono garantiti i diritti d'accesso solo agli oggetti strettamente necessari per la sua esecuzione (POLA: principle of least authority). il POLA è una caratteristicha desiderabile in ogni sistema di controllo.

Meccanismi:
I **meccanismi di protezione** sono gli strumenti necessari a mettere in atto una determinata politica. Principi di realizzazione:

- **Flessibilità del sistema di protezione**: i meccanismi devono essere sufficientemente generali per consentire l'applicazione di diverse politiche di protezione
- **Separazione tra meccanismi e politiche**: la politicha definische "cosa va fatto" ed il meccanismo "come va fatto". Ovviamente è desiderata la massima indipendenza tra le due componenti.

2.1 Dominio di protezione

Un dominio definisce un insiem edi coppie, ognuna contenente l'identificatore di un oggetto e l'insieme delle operazioni che il soggetto associato al dominio può eseguire su ciascun oggetto

$D(S) = \{<$o, diritti$> \mid$ o è un oggetto, diritti è un insieme di operazioni$\}$

Modello di Grahmm-Denning
Questo modello forsnisce una serie di comandi che garantiscono la modifica controllata dello stato di protezione:

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

- grant access right
- delete access right
- tranfer access right

2.1.1 Diritti

Diritto Owner:

Il diritto owner permette l'assegnazioen di qualunque diritto di accesso su un oggetto X ad un qualunque soggetto Sj da parte di un soggetto Si. L'operazione è consentita solo se il diritto owner appartiene a A[Si, X]

Diritto Control:

Eliminazione di un diritto di accesso per un oggetto X nel dominio di Sj da parte di Si. L'operazione è consentita solo se il diritto control appartiene a A[Si, Sj], oppure owner appartiene a A[Si, X].

Cambio di dominio: switch

Il cambio di dominio permette che un processo che esegue nel dominio del soggetto si può commutare al dominio di un altro soggetto Sj. L'operazione è consentita solo se il diritto switch appartiene a A[Si, Sj].

2.2 Realizzazione della matrice delgi accessi

La matrice degli accessi è una notazione astratta che rappresenta lo stato di protezione. Nella rappresentazione concreta è necessario considerare: la dimensione della matrice e matrice sparsa.
La rappresentazione concreta della matrice degli accessi deve essere ottimizzata sia riguardo all'occupazione di memoria sia rispetto all'efficienza nell'accesso e nella gestione della informazioni di protezione. Ci sono principalmente di approcci:

- **Access Control List (ACL)**: rappresentazione per colonne, per ogni oggetto è associata una lista che contiene tutti i soggetti che possono accedere all'oggetto, con i relativi diritti d'accesso per l'oggetto
- **Capability List**: rappresentazione per righe, ad ognin soggetto è associata una lista che contiene gli oggetti accessibili dal soggetto ed i relativi diritti d'accesso.

2.2.1 Access Control List

La lista degli accessi per ogni oggetto è rappresentata dall'insieme delle coppie: <**soggetto, insieme dei diritti**> limitatamente ai soggetti con un insieme non vuoto di diritti per l'oggetto.

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

Utenti e Gruppi

Generalmente ogni soggetto rappresenta un singolo utente. Molti sistemi hanno il concetto di **gruppo di utenti**. I gruppi hanno un nome e possono essere inclusi nella ACL.
In questo caso l'entry in ACL ha la forma: **UID-1, GID-1 : <insieme di diritti>**
UID-2, GID-2 : <insieme di diritti>
Dove UID è lo user identifier e GID è il group identifier.

In certi casi il gruppo identifica un ruolo: uno stesso utente può appartenere a gruppi diversi e quindi con diritti diversi. In questo caso, quando un utente accede, specifica il gruppo di appartenenza.

2.2.2 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**. Il quale di compone di un identificatore (o un indirizzo) che indica l'oggetto e la rappresentazione dei vari diritti d'accesso. Quando S intende eseguire un'operazione M su un oggetto Oj: il meccanismo di protezione controlla se nella lisra delle capability associata a S ne esiste una relativa ad Oj che abbia tra i suoi diritti M.

Ovviamente le Capability List devono essere protette da manomissioni, ed è possibile in diversi modi:

- la capability list viene gestita solamente da s.o.; l'utente fa riferimento ad un puntatore (capability) che identifica la sua posizione nella lista appartenete allo spazio del kerner
- Architettura etichettata: a livello HW, ogni singola parola ha bit extra, che esprimono la protezione su quella cella di memoria. Ad esempio, se è una capability, deve essere protetta da scritture non autorizzate.

I bit tag non sono utilizzari dall'aritmetica, dai confronti e da altre istruzioni normali e può essere modificato solo da programmi che agiscono in modo kernel.

2.2.3 Revoca dei diritti di accesso

In un sistema di protezione dinamica può essere necessario revocare i diritti d'accesso per un oggetto. La revoca può essere di tre tipi:

- **generale o selettiva**: cioè valere per tutti gli utenti che hanno quel diritto di accesso o solo per un gruppo
- **parziale o totale**: cipè riguardare un sottoinsieme di diritti per l'oggetto, o tutti
- **temporanea o permanente**: cioè il diritto di accesso non sarà più disponibile, oppure potrà essere successivamente riottenuto

Revoca del diritto per un oggetto con ACL:
Si fa riferimento alla ACL associata all'oggetto e si cancellano i diritti di accesso che si vogliono revocare

Revoca del diritto per un oggetto con Capability List:
Più complicato rispetto ad ACL. È necessario verificare per ogni dominio se contiene la capability con riferimento all'oggetto considerato.

2.3 Protezione e Sicurezza

La protezione riguarda solamente il controllo degli accessi alle risorse interne al sistema. Invece la sicurezza si occupa di controllare gli accessi al sistema stesso. In alcuni casi la sola protezione può non essere efficace, nel caso in cui, ad esempio, un utente autorizzato riesce a far eseguire programmi che agiscono sulle risorse del sistema.

2.3.1 Sicurezza Multilivello

La maggior parte dei sistemi operativi permette ai singoli utenti di determinare chi possa leggere e scrivere i loro file ed oggetti. Invece in acluni ambienti è richiesto e necessario un più stretto controllo sulle regole di accesso alle risorse (ambiente militare, ospedaliero, ecc). Vengono quindi stabilite delle regole **generali** su "chi può accedere e a che cosa", che possono essere modificate solo da un'entità centrale autorizzata.
Quando è necessario un controllo obbligatorio degli accessi al sistema, l'organizzazione a cui il sistema appartiene definisce le politiche **MAC** che stabiliscono le **regole generali** tramite l'adozione di un modello di sicurezza.

I modelli di sicurezza più utilizzati sono:

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

Entrambi sono modelli multilivello.

In un modello di sicurezza multilivello i **soggetti** (utenti) e gli **oggetti** (risorse, file, ecc) sono **classificati** in **livelli** (classi di accesso):

- Livelli per i soggetti (**clearance levels**)
- Livelli per gli oggetti (**sensivity levels**)

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

Modello Bell-La Padula

Modello progettato per realizzare la sicurezza in organizzazioni militari, garantendo la **confidenzialità** delle informazioni.
Associa a un sistema di protezione (matrice degli accesso) due regole di sicurezza MAC, che stabiliscono la direzione di propagazione delle informazioni nel sistema.

Quattro livelli di sensibilità degli oggetti:

- Non classificato
- Confidenziale
- Segreto
- Top secret

Quattro livelli di autorizzazione (clearance) per i soggetti:
Le persone sono assegnate ai livelli a seconda del ruolo nell'organizzazione.

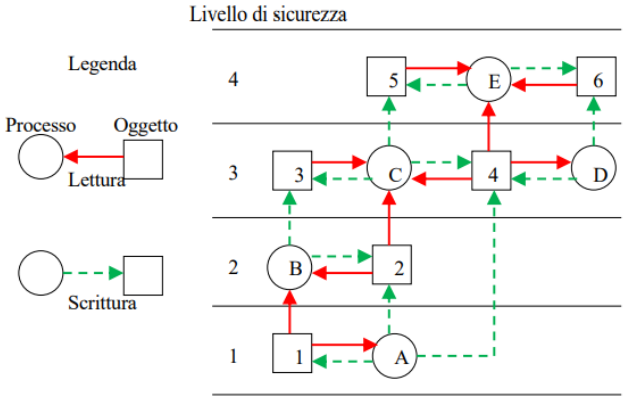
Regole di Sicurezza:

- **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 al livello di sicurezza k può scrivere solamente oggetti al suo livello o a quelli superiori

Quindi i processi possono leggere verso il basso e scrivere verso l'alto, ma non il contrario. (flusso delle informazioni dal basso verso l'alto).

Generalmente a queste regole si aggiungono le regole di protezioen speficicate dalla matrice degli accessi.

Il modello Bell-La Padula è stato concepito per mantenere i segreti, non per garantire l'integrità dei dati. Epossibile infatti sovrascrivere l'informazione appartenente ad un livello superiore.



Esempio cavallo di troia

Modello Biba

Se il modello Bell-La Padula è stato concepito per mantenere i segreti e non per garantire l'integrità dei dati, il Modello Biba ha come obiettivo principale proprio l'integrità dei dati.

- **Proprietà di semplice sicurezza:** un processo in esecuzione al livello di sicurezza k può scrivere solamente oggetti al suo livello o a quelli inferiori (nessuna scrittua verso l'alto)
- **Proprietò di integrità ***: un processo in esecuzione al livello k può leggere solo oggetti al suo livello o quelli superiori (nessuna lettura verso il basso)

Ovviamente il modello Biba è in conflitto con il modello B-LP e quindi non possono essere utilizzati contemporaneamente.

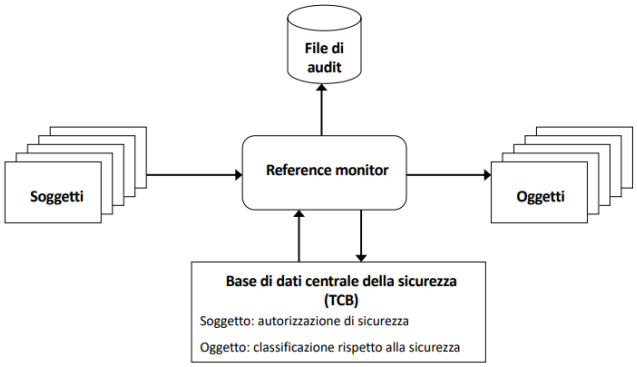
2.3.2 Architettura dei sistemi ad elevata sicurezza

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

Reference Monitor: è un elemento di controllo realizzato dall'hardware e dal S.O. che regola l'accesso dei soggetti agli oggetti sulla base di paramentri di sicurezza (es. modello Bell-La Padula)

Trusted computing base: il RM ha accesso ad una base di calcolo fidata (Trusted computing base, o TBC) che contiene:

- Privilegi dei sicurezza (autorizzazioni di sicurezza) di ogni soggetto
- Attributi (classificazione rispetto alla sicurezza) di ciascun oggetto



Sistemi fidati

Il RM impone le regole di sicurezza (B-LP: no read-up, no write-down) ed ha le seguenti proprietà:

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

Il requisito di **mediazione completa** rende preferibile, per motivi di efficienza, che la soluzione debba essere almeno parziamente hardware.

Il requisito di **isolamento** impone che non sia possibile oer chi porta l'attacco, modificare la logica del RM o il contenuto del DB centrale della sicurezza.

Il requisito della **verificabilità** è difficile da soddisfare per un sistema general-purpose.

3 Programmazione Concorrente

4 Modello a memoria comune

Esistono 2 principali modelli di interazione tra i processi:

- Modello a **memoria comune** (ambiente globale, shared memory)
- Modello a **scambio di messaggi** (ambiente locale, distributed memory)

Il modello a memoria comune rappresenta la più semplice astrazione del funzionamento di un sistema in multiprogrammazione costituito da uno o più processi che hanno accesso ad una memoria comune.
Ogni appliczione viene strutturata come un insieme di componenti, suddiviso in due sottoinsieme disgiunti:

- **Processi** (componenti attivi)
- **Risorse** (componenti passivi)

Le Risorse rappresentatno un qualunque oggettim fisico o logico, di cui un processo necessita per portare a termine il suo compito. Le risorse vengono raggruppate in classi, dove una classe rappresenta l'insieme di tutte e sole le operazioni che un processo può eseguire per operare su risorse di quella classe,
Ovviamente ci deve essere la necessità di specificare quali processi ed in quali istanti possono accedere alla risorsa. Quindi il **meccanismo di controllo degli accessi** si occupa di controllare che gli accessi dei processi alle risorse avvengano correttamente.

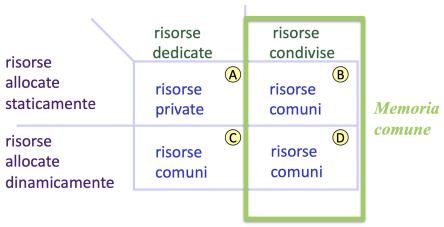
4.1 Gestore delle Risorse

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

Classificazione delle risorse:

- Risorsa R **dedicata**: se SR(t) ha una caardianlità sempre ¡= 1
- Risorsa R **condivisa**: in caso contrario
- Risorsa R **allocata staticamente**: se SR(t) è una costante, quindi se SR(t) = SR(t0) per ogni t
- Risorsa R **allocata dinamicamente**: se SR(t) è funzione del tempo

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, in base alle regole del linguaggio, stabilisce quale processo può vedere e quindi operare su R.
Per ogni risorsa **allocata dinamicamente**, il relativo gestore GR definisce l'insieme SR(t) in fase di esecuzione e quindi deve essere un componente della



stessa applicazione, nel quale l'allocazione viene decisa a run-time in base a politiche date.

Quindi i principali compiti del Gestore delle risorse sono:

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

Accesso a Risorse Consideriamo un processo P che deve operare, ad un certo istante, su una risorsa R di tipo T:
Se R è allocata **staticamente** a P (modalità A e B), il processo, se appartiene a SR, possiede diritto di operare su R in qualunque istante.

```
1 R.op(...);
```

Se R è allocata **dinamicamente** a P (modalità C e D), è necessario prevedere un gestore GR, che implementa le funzioni di Richiesta e Rilascio della risorsa; quindi il processo P deve seguire il seguente protocollo:

```
1 GR.Richiesta(...);  // acquisizione della risorsa
2 R.op(...);          // esecuzione dell'operazione op su R
3 GR.Rilascio(...);   // rilascio della risorsa R
```

Se R è allocata come **risorsa condivisa**, (modalità B e D) è necessario assicurare che gli accessi avvengano in modo non divisibile: nel senso che òe funzioni di accesso alla risorsa devono essere programmate come una **classe di sezioni critiche**, utilizzando meccanismi di sincronizzazione offerti dal linguaggio di programmazione e supportati dalla macchina concorrente.

Se R è allocata come **risorsa dedicata**, (modalità A e C), essendo P l'unico processo che accede alla risorsa, non è necessario prevedere alcuna forma di sincronizzazione.

Regione critica condizionale [Hoare, Brinch-hansen] Formalismo che permette di esprimere la specifica di qualunque vincolo di sincronizzazione. Data una risorsa R condivisa:

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

- tra doppie parentesi angolai il **corpo** della region che rappresenta una operazione da eseguire su una risorsa condivisa R e quindi costituisce una sezione critica che deve essere eseguita in **mutua esclusione** con le altre operazioni definite su R
- il corpo della region è costituito da due istruzioni da eseguire in sequenza: l'istruzione **Sa** e successivamente l'istruzione **Sb**
- in particolare, una volta terminata l'esecizione di Sa viene valutata la condizione **C**:
  - se C è **vera** l'esecuzione continua con Sb
  - se C + **false** il processo che ha invocato l'operazione attende che la condizione C diventi vera. Non appena C sarà vera l'esecuzione della region potrà riprendere ed eseguire Sb

Esistono però dei casi particolari di regioni critiche:

- **region R << S; >>**: specifica della sola mutua esclusione, senza ulteriori vincoli
- **region R << when(C) >>**: specifica di un semplice vincolo di sincronizzazione, nel quale il processo deve attendere che C sia verificata prima di proseguire
- **region R << when(C) S; >>**: specifica il caso dìin cui la condizione C di sincronizzazione caratterizza lo stato in cui la risorsa R deve trovarsi per poter eseguire l'operazione S (C quindi è una precondizione di S)

4.2 Mutua Esclusione

Il probelma della mutua esclusione nasce quando più di un processo alla volta può e deve accedere a variabili comuni. Quindi è di fondamentale importanza che le operazioni con le quali i processi accedono alle variabili comuni non si sovrappongano nel tempo.

Con sezione critica s'intende la sequenza di istruzioni con le quali un processo accede e modifica un insieme di variabili comuni. Ad un insieme di variabili comuni possono essere associate una sola sezione critica (usata da tutti i processi) oppure più sezioni critiche (classe di sezioni critiche).
La regola della mutua esclusione stabilisce che:

Sezioni critiche appartenenti alla atessa classe devono escludersi mutuamente nel tempo.

oppure

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

Per specificare una sezione critica **S** che opera su una risorsa condivisa R:

```
1    <prologo>
2        S;
3    <epilogo>
```

Attraverso il **prologo** si ottiene l'autorizzazione ad eseguire la sezione critica, quindi R viene acquisita in modo esclusivo. Invece attraverso l'**epilogo** la risorsa R viene liberata.

Le principali soluzioni possibili alla mutua esclusine sono:

- **Algoritmiche:** (es. Algoritmi di Dekker, ecc.) la soluzione non necessita di meccanismi di sincronizzazione (es. semafori, lock, ecc.), ma sfrutta solo la possibilità di condivisione di variabili; l'attesa di un processo che trova la variabile condivisa già occupata viene modellata attraverso cicli di attesa attiva
- **Hardware-based:** ad esempio disabilitazione delle interruzioni, lock/unlock. Quindi il supporto è fornito direttamente dall'architettura HW.
- **Strumenti software di sincronizzazione realizzati dal nucleo della macchina concorrente:** prologo ed epilogo sfruttano strumenti di sincronizzazione che consentono l'effettiva sospensione dei processi in attesa ed eseguire sezioni critiche.

4.3 Strumenti linguistici per la programmazione di interazioni

4.3.1 Il Semaforo

Il semaforo è uno strumento linguistico di basso livello che consente di risolvere qualunque problema di sincronizzazione nel modello a memoria comune.
Eŕealizzato dal nucleo della macchina concorrente. L'eventuale attesa nella esecuzione può essere realizzata utilizzando i meccanismi di gestione dei thread

offerti dal nucleo. Inoltre viene utilizzato per realizzaer strumenti di sincronizzazione di può alto livello ad esempio le *condition*.

Definizione: un semaforo è una **variabile intera non negativa**, alla quale è possibile accedere solo **tramite le due operazioni P e V**.

Definizione di un oggetto di tipo **semaphore**:

```
1    semaphore s = i;     // dove i (i >= 0)     il valore
        ↪iniziale
```

Al tipo semaphore sono associati:

$$\text{Insieme di valori} = \{X | X \in N\}$$

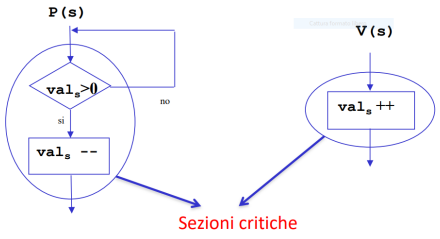
$$\text{Insieme delle operazioni} = \{P, V\}$$

Operazioni sul semaforo

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

```
1    void P(semaphore s):
2        region s << when(val_s > 0) val_s--; >>
3
4    void V(semaphore s):
5        region s << val_s++; >>
6
7    // dove val_s rappresenta il valore del semaforo
```

Essendo **s** l'oggetto condiviso, le due operazioni P e V vengono definite come **sezioni critiche** da eseguire in mutua esclusione e in forma atomica.



Quindi il semaforo viene utilizzato come strumento di sincronizzazione tra processi concorrenti:

- **attesa**: P(s), val-s == 0
- **risveglio**: V(s), se vi è almeno un processo sospeso

Proprietà del Semaforo Dato un semaforo S, siano:

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

Ad ogni istante possiamo esprimere il valore del semafor come:

$$val_s = I_s + nv_s - np_s$$

da cui ($val_s >= 0$):

Relazione di Invarianza

$$np_s <= I_s + nv_s$$

La relazione di invarianza è sempre soddisfatta, per ogni semaforo, qualunque sia il suo valore e comunque sia strutturato il programma concorrente che lo usa.

Il semaforo è quindi uno strumento generale che consente la risoluzione di qualunque problema di sincronizzazione.
Esistono molti casi d'uso del meccanismo semaforico:

- semafori di mutua esclusione
- semafori evento
- semafori binari composti
- semafori condizione
- semafori risorsa
- semafori privati

Semaforo di mutua esclusione

Il semaforo di mutua esclusione viene inizializzato ad 1. Principalmente viene utilizzato per realizzare le sezioni critiche di una stessa classe, secondo il protocollo:

```
1    class tipo_risorsa {
2        <struttura dati di ogni istanza della classe>;
3
4        semaphore mutex = 1;
5
```

```
6           public void op1() {
7               P(mutex);   // prologo
8               <sezione critica: corpo della funzione op1>;
9               V(mutex);   // epilogo
10          }
11
12          public void opN() {
13              P(mutex);   // prologo
14              <sezioen critica: corpo della funzione opN>;
15              V(mutex);   // epilogo
16          }
17      }
18
19      tipo_risorsa ris;
20      ris.opi();
```

(il semaforo di mutua esclusione può assumere solo i valori 0 e 1)

In alcuni casi è consentito a più processi di eseguire contemporaneamente la stessa operazione su una risorsa, ma non operazioni diverse.
Quindi una soluzione protrebbe essere:

- definisco un semaforo mutex per la mutua esclusione tra operazioni
- prologo ed epilogo di op_i sono sezioni critiche quindi introduco un ulteriore semaforo di mutua esclusione m_i

Un esempio potrebbe essere il **problema dei lettori/scrittori**.
Sia data una risorsa condivisa F (ad esmepio un file) che può essere acceduta dai thread concorrenti in due modi:

- **lettura**;
- **scrittura**

Una possibile soluzione di sincronizzazione potrebbe essere:

- la lettura è consentita a più thread contemporaneamente;
- la scrtittura è consentita ad un thread alla volta;
- lettura e scrittura su F non possono avvenire contemporaneamente

```
1       semaphore mutex = 1;
2       semaphore ml = 1;
3       int contl = 0;
4
5       public void lettura(...) {
6           P(ml);
7           contl++;
```

```
8
9           if(contl == 1) {
10              P(mutex);
11          }
12
13          V(ml);
14          <lettura del file>;
15          P(ml);
16          contl--;
17
18          if(contl == 0) {
19              V(mutex);
20          }
21
22          v(ml);
23      }
24
25      public void scrittura(...) {
26          P(mutex);
27          <scrittura del file>;
28          V(mutex);
29      }
```

Semaforo evento

Un semaforo evento è un semaforo binario utilizzato per imporre un **vincolo di precedenza** tra le operazioni dei processi. (ad es. op_a deve essere eseguita da P1 solo dopo che P2 ha eseguito op_b).

Introduciamo quindi un semaforo **sem** inizializzato a **zero**:

- prima di eseguire op_a, P1 esegue P(sem)
- dopo aver eseguito op_b, P2 esegue V(sem)

problema del rendez-vous slide 48

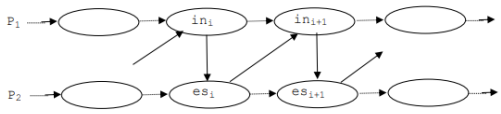
Semafori binari composti

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.

Due processi P1 e P2 si scambiano dati di tipo T utilizzando una memoria condivisa (buffer).
Quindi devono esserci dei vincoli di sincronizzazione:

- accessi al buffer mutuamente esclusivi
- P2 può prelevare un dato solo dopo che P1 lo abbia inserito
- P1, prima di inserire un dato, deve attendere che P2 abbia estratto il precedente



Utilizziamo quindi due semafori:

- **vu**, per realizzare l'attesa di P1, in caso di buffer pieno
- **pn**, per realizzare l'attesa di P2, in caso di buffer vuoto

Buffer inizialmente vuoto, vu = 1, pn = 0

```
1  void invio(T dato)
       ↪{
2      P(vu);
3      inserisci(dato)
           ↪;
4      V(pn);
5  }
```

```
1  T ricezione() {
2      T dato;
3      P(pn);
4      dato = estrai()
           ↪;
5      V(vu);
6      return dato;
7  }
```

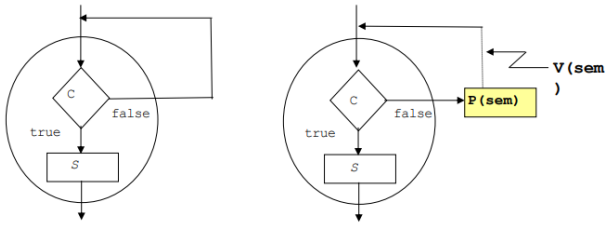
Semaforo condizione

In alcuni casi l'esecuzione di un'istruzione S1 su una risorsa R è subordinata al verificarsi di una condizione C:

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

Il processo deve sospendersi se la condizione non è verificata e deve sucire dalla regione per consentire ad altri processi di eseguire altre operazioni su R per rendere vera la condzione C.

- 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.



(a) (b)

– è necessaria un'altra operazione op2 che, chiamata da un'altro processo, modifichi lo stato interno di R in modo che C diventi vera
– nell'ambito di op2 viene eseguita la V(sem) per risvegliare il processo

Schema con **attesa circolare**

```
semaphore mutex = 1;
semaphore sem = 0;
int csem = 0;
```

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

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

Schema con **passaggio di testimone**:

```
semaphore mutex = 1;
semaphore sem = 0;
int csem = 0;
```

```
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);
    }
}
```

Questo secondo schema è più efficientre del primo ma ha comunque delle limitazioni. Permette di risvegliare un solo processo alla volta poichè ad uno solo può passare il diritto di operare in mutua esclusione. Inoltre la condizione C (precondizione di S1) deve essere verificabile anche all'interno di op2. Ciò significa che non deve contenere variabili locali o parametri della funzione op1.

Semaforo condizione

I semafori risorsa sono semafori generali, quindi possono assumere qualunque valore ¿= 0. Vengono generalmente impiegati per realizzare l'allocazione di risorse equivalenti, nel quale il valore del semaforo rappresenta il numero di risorse libere.

```
class tipo_gestore {
    semaphore mutex = 1;     // semaforo di mutua esclusione
    semaphore n_ris = N;     // semaforo risorsa
    boolean libera[N];       // indicatori di risorsa libera

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

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

```
23
24      public ovid rilascio(int r) {
25          P(mutex);
26          libera[r] = true;
27          V(mutex);
28          V(n_ris);
29      }
30  }
```

Leggere esempi sulle condizioni di sincronizzazione.

Semaforo privato

Un semaforo S si deive privato per un processo quando solo tale processo può eseguire la primitiva P sul semaforo S. La primitiva V sul semaforo può essere invece eseguita da qualunque processo. Generalmente un semaforo privato viene inizalizzato con il valore zero.

I semafori privati vengono utilizzati per realizzare particolari politiche di allocazione di risorse:

- il processo che acquisisce la risorsa può eventualmente sospendersi sul suo semaforo privato (se la condizione di sincronizzazione non è rispettata)
- chi rilascia la risorsa, risveglierà uno tra i processi sospesi, in base alla politica scelta, mediante una V sul semaforo privato del processo scelto.

Allocazione di risorse, **primo schema**:

```
1   class tipo_gestore_risorsa {
2       <struttura dati del gestore>;
3       semaphore mutex = 1;
4       semaphore priv[n] = {0, 0, ..., 0}; // semafori privati
5
6       public void acquisizione(int i) {
7           P(mmutex);
8           if(<condizione di sincronizzazione>) {
9               <allocazione della risorsa>;
10              V(priv[i]);
11          } else {
12              <registra la sospensione del processo>;
13          }
14          V(mutex);
15          P(priv(i));
16      }
17
18      public void rilascio() {
19          int i;
20          P(mutex);
```

```
        <rilascio della risorsa>;
        if(<esiste almeno un processo sospeso per il quale
            ↪la condizione risulta true>) {
            <scelta fra i processi sospesi quello destinato
                ↪alla riattivazione (Pi)>;
            <allocazione della risorsa a Pi>;
            <registrare che Pi non e piu sospeso>;
            V(priv[i]);
        }
        V(mutex);
    }
}
```

Proprietà del primo schema:

- la sospensione del processo, nel caso in cui la condizione di sincronizzazione non sia soddisfatta, non può avvenire entro la sezione critica in quanto ciò impedirebbe ad un processo che rilascia la risorsa di accedere a sua volta alla sezione critica e di riattivare il processo sospeso. E quindi la sospensione avviene fuori dalla sezione critica.
- la specifica del particolare algoritmo di assegnazione della risorsa non è opportuno che sia realizzata nella primitiva V.
- questo schema può presentare i seguanti problemi:
  - l'operazione P sul semaforo privato viene sempre eseguita anche quando il processo richiedente non deve essere bloccato
  - il codice relativo all'assegnazione della risorsa viene duplicato nelle procedure acquisizione e rilascio

Allocazione di risorse, **secondo schema** (supera i limiti del primo):

```
class tipo_gestore_risorsa {
    <struttura dati del gestore>;
    semaphore mutex = 1;
    semaphore priv[n] = {0, 0, ..., 0};

    public void acquisizione(int i) {
        P(mutex);
        if(! <condizione di sincronizzazione>) {
            <registrare la sospensione del processo>;
            V(mutex);
            P(priv[i]);
            <resitrare che il processo non e piu sospeso>;
        }
        <allocazione della risorsa>;
        V(mutex);
```

```
16      }
17
18      public void rilascio() {
19          int i;
20          P(mutex);
21          <rilascio della risorsa>;
22          if(<esiste almeno un processo sospeso per il quale
                ↪la condizione risulta true>) {
23              <scelta del processo Pi da riattivare>;
24              V(priv[i]);
25          } else {
26              V(mutex);
27          }
28      }
29  }
```

Rispetto al primo schema, in questa soluzione risulta più complesso realizzare la riattivazione di più processi per i quali risulti vera contemporaneamente la condizione di sincronizzazione. Infatti il processo che rilascia la risorsa attiva al più un processo sospeso, il quale dovrà a sua volta provvedere alla riattivazione di eventuali altri processi.

5 Realizzazione del nucleo

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

Il nucleo costituisce il livello più interno di un qualunque sistema basato su processi, ad esempio:

- il livello più elementare di un sistema operativo multiprogrammato
- il supporto a tempo di esecuzione di un linguaggio per la programmazione concorrente

Il nucleo è il solo modulo che è consio dell'esistenza delle interruzioni. I processi che colloquiano con i dispositivi utilizzano opportune primitive del nucleo che provvedono a sospenderli in attesa del completamento dell'azione richiesta. Non appena l'azione viene completata, il relativo segnale di interruzione iviato dal dispositivo alla CPU viene catturato dal nucleo che provvede a risvegliare il processo sospeso.
La gestione delle interruzioni è quindi invisibile ai processi ed ha come unico effetto rilevabile di rallentare la loro esecuzione sulle rispettive macchine virtuali.

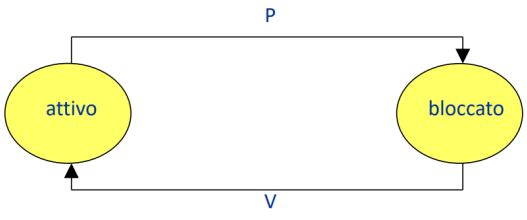
5.1 Caratteristiche fondamentali del nucleo

Efficienza: condiziona l'intera struttura a processi. Infatti esistono dei sistemi in cui alcune o tutte le operazioni del nucleo sono realizzate in hardware o attraverso microprogrammi.

Dimensioni: la semplicità delle funzioni richieste al nucleo fa si che la sua dimensinoe risulti estremamente limitata.

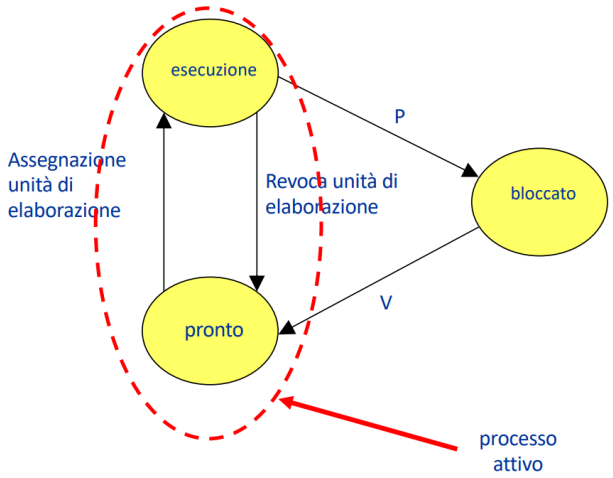
Separazioen tra meccanismi e politiche: il nucleo deve contenere solo meccanismi consentendo così, a livello di processi, di utilizzare tali meccanismi per la realizzazione di diverse politiche di gestione a seconda del tipo di applicazione.

Stati di un processo:



Le transizioni tra i due stati sono implementate dai meccanismi di sincronizzazione realizzati dal nucleo. **P** per sospensione e **V** per risveglio.

Stati di un processo in un sistema in cui il numero di processi supera il numero delle unità di elaborazione:



Quando un processo perde il controllo del processore, il contenuto dei registri del processo viene salvato in un'area di memoria associata al processo, chiamata descrittore.
Ciò consente una maggiore flessibilità nella politica di assegnazione del processore ai processi, rispetto alla soluzione di salvare le informazioni nello stack.

Quindi la funzione fondamentale del nucleo di un sistema a processi è la gestione delle transizioni di stato dei processi. I principali compiti del nucleo sono:

- **Gestire il salvataggio e il ripristino dei contesti dei processi**: quando un processo abbandona il controllo dell'unità di elaborazione fisica, tutte le informazioni contenute nei registri di tale unità devono essere trasferite nel descrittore. Allo stesso modo, quando un processo riprende l'esecuzione tutte le informazioni contenute nel suo descrittore devono essere trasferite nei registri di macchina
- **Scegliere a quale tra i processi pronti assegnare l'unità di elaborazione (scheduling della CPU)**: quando un processo abbandona il controllo dell'unità di elaborazione, il nucleo deve scegliere tra tutti i processi pronti quello da mettere in esecuzione. La scelta può essere o di tipo FIFO, oppure può utilizzare la priorità dei processi.
- **Gestire le interruzioni dei dispositivi esterni**: traducendole in attivazione di processi da bloccato a pronto.
- **Realizzare i meccanismi di sincronizzazione dei processi**: gestendo il passaggio dei processi dallo stato di esecuzioen allo stato di bloccato e da bloccato a pronto.

5.2 Realizzazione del Nucleo: Architettura monoprocessore

5.2.1 Strutture Dati del Nucleo

Il **Descrittore del processo** è la principale struttura dati del nucleo, e contiene le seguenti informazioni:

- **Identificatore del processo**: nome che identifica univocamente il processo durante il suo tempo di vita
- **Stato del processo**
- **Modalità di Servizio**: contiene parametri di scheduling
  - FIFO
  - Priorità (fissa o variabile)
  - Deadline (tempo massimo entro il quale la richiesta può essere soddisfatta)
  - Quanto di temo (sistemi time sharing)
- **Contesto del processo**: program counter, registro di stato, registri generali, indirizzo dell'area di memoria privata del processo.
- **Code di processo**: a seconda del loro stato i processi vengono inseriti in apposite code. Ogni descrittore contiene l'identificatore del processo successivo nella stessa coda.

```
typedef struct {
    int indice_priorit ;
    int delta_t;
} modalit _di_servizio;

typedef struct {
    int nome;
    modalit _di_servizio servizio;
    tipo_contesto contesto;
    tipo_stato stato;   // running, ready, waiting, ecc.
    int successivo;
} descrittore_processo;

descrittore_processo descrittori[num_max_proc];
```

Listing 1: Realizzazione desctittore del processo

5.2.2 Coda dei processi pronti

Esistono sempre una o più **code di processi pronti**. Non appena un processo viene riattivato tramite una **v** viene inserito in fondo alla coda corrispondente alla sua priorità.
La coda dei processi pronti contiene sempre almeno un **dummy process** il quale viene messo in esecuzione solamente quando tutte le altre code sono vuote e rimane in esecuzione fino a quando qualche altro processo diventa pronto. Inoltre il dummy process ha la priorità più bassa ed è sempre nello stato di pronto.

Listing 2: Esempio di realizzazione coda dei processi pronti

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

typedef descrittore_coda coda_a_livelli[Npriorita];

coda_a_livelli coda_processi_pronti;

// Inserimento => inserisce il processo di indice P nella
    ↪coda C
void Inserimento(int P, descrittore_coda C) {

}

// Prelievo => estrae il primo processo dalla coda C e
    ↪ristituisce il suo indice
int Prelievo(descrittore_coda C) {

}
```

Coda dei descrittori liberi:
Coda nella quale sono concatenati i descrittori disponibili per la creazione di nuovi processi e nella quale sono reinseriti i descrittori dei processi terminati

Processo in esecuzione:
Il nucleo necessita di conoscere quale processo è in esecuzione. Questa informazione, rappresentata dall'indice del descrittore del processo, viene contenuta in una particolare variabile del nucleo. Quando il nucleo viene inizializzato, viene creato un processo e l'indice del processo viene assegnato a $processo_in_esecuzione$.

5.3 Funzioni del Nucleo

Le funzioni del nucleo realizzano le operazioni di **transizione di stato** per i singoli processi. Ogni transizione prevede il prelievo da una coda del descrittore

del processo coinvolto ed il suo inserimento in un'altra coda.
A tale scopo vengono utilizzate due procedure: **Inserimento** e **Prelievo** di un descrittore da una coda. Se la coda è vuota si adotta l'ipotesi che il campo primo assuma il valore -1.

Le funzioni del nucleo vengono suddivise in due livelli:

- **Livello Superiore**: il quale contiene tutte le funzioni direttamente utilizzabili dai processi sia esterni che interni, quali risposta ai segnali di interruzione e primitive per la creazione, eliminazione e sincronizzazione dei processi.
- **Livello Inferiore**: il quale realizza tutte le funzionalità di cambio di contesto, ad esmepio salvataggio del contesto del processo che si sospende nel suo descrittore, scelta di un nuovo processo da mettere in esecuzione tra quelli pronti e ripristino del suo contesto.

L'ambiente di esecuzione delle funzioni del nucleo ha caratteristiche distinte dal quello dei processi. Infatti, per motivi di protezione, le funzioni del nucleo sono le uniche che:

- possono operare sulle strutture dati che rappresentano lo stato del sistema (descrittori, code di descrittori, semafori, ecc..)
- possono utilizzare istruzioni privilegiate (abilitazione e disabilitazione delle interruzioni, ecc)

Le funzioni del nucleo devono essere eseguite in modo mutuamente esclusivo. Inoltre i due ambienti di esecuzioen (nucleo e processi utente) corrispondono a stati diversi di operazione dell'elaboratore (kernel e user). E il meccanismo di passaggio da uno all'altro è basato sul meccanismo delle interruzioni.
In particolare:

- Nel caso di funzioni chiamate da **processi esterni**, il passaggio all'ambiente del nucleo è ottenuto tramite il meccanismo di risposta al segnale di **interruzione**
- Nel caso di funzioni chiamate da **processi interni**, il passaggio è ottenuto tramite l'esecuzione di **system calls**
- In entrambi i casi, al completamento della funzione richiesta, il trasferimento all'ambiente user avviene tramite il meccanismo di **ritorno da interruzione (RTI)**

5.3.1 Funzione di livello inferiore: Cambio di Contesto

La funzione di cambio di contesto è costituita da 3 principali fasi:

Salvataggio dello stato:
Salvataggio del contesto del processo in esecuzione nel suo descrittore e inserimento del descrittore nella coda dei processi bloccati o dei processi pronti.

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

Assegnazione della CPU
Rimozione del processo a maggiore priorità dalla coda dei pronti e caricamento dell'identificatori di tale processo nel registro processo in esecuzione.

```
// scheduling: algoritmo con priorit
void Assegnazione_CPU() {
    int k = 0, j;
    while(coda_processi_pronti[k].primo == -1) {
        k++;
    }

    j = Prelievo(coda_processi_pronti[k]);
    processo_in_esecuzione = j;
}
```

Ripristino dello stato:
Caricamento del processo del nuovo processo nei registri di macchina.

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

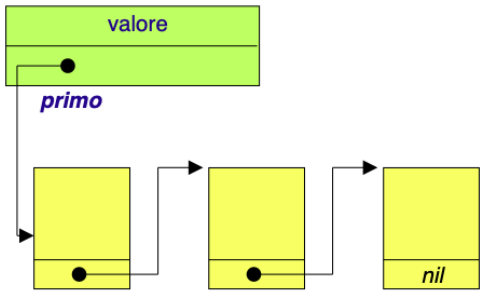
Gestione del temporizzatore:
Per consetire la modalità di servizio a divisione di tempo è necessario che il nucleo gestica un **dispositivo temporizzatore** tramite un'apposita procedura che ad intervalli di tempo fussati, provveda a sospendere il processo in esecuzione ed assegnare l'unità di elaborazione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

```
void Cambio_di_Contesto() {
    int j, k;
    Salvataggio_stato();
    j = processo_in_esecuzione;
    k = descrittori[j].servizio.priorit ;
    Inserimento(j, coda_processi_pronti[k]);
    Assegnazione_CPU();
    Ripristino_stato();
}
```

5.4 Realizzazione Semaforo (caso monoprocessore)

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

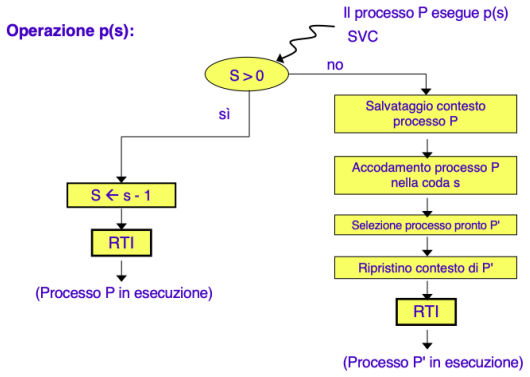
- una variabile intera che rappresenta il suo valore (¿=0)
- un puntatore ad una lista di descrittori di processi in attesa sul semaforo (bloccati)



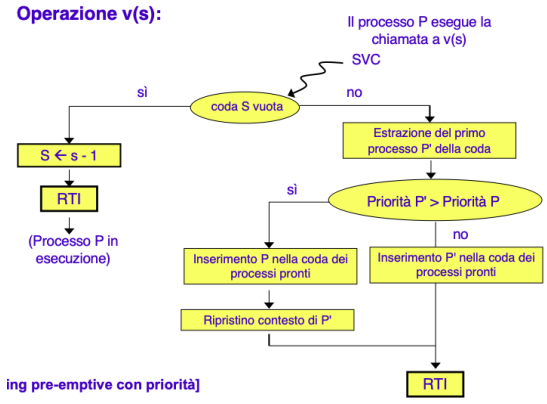
Se non sono presenti semafori in coda, il puntatore alla lista contiene la costante **nil**.
La coda viene gestita con politica FIFO, e quindi i processi risultano ordinati secondo il loro tempo di arrivo nella coda associata al semaforo.
Il descrittore di un processo viene inserito nella coda del semaforo come conseguenza di una primitiva p non passsante, e viene prelevato per effetto di una v.

```
1  // gestione dei processi basata su priorit  => associamo al
       ↪ semaforo un insieme di code (una per priorit )
2
3  typedef struct {
4      int contatore;
5      coda_a_livelli coda;
6  } descr_semaforo;
```

```
1  // insieme di tutti i semafori
2  descr_semaforo semafori[num_max_sem];
3
4  //ogni semaforo    rappresentato dall'indice che lo
       ↪individua nel vettore semafori
5  typedef int semaforo;
6
7  void P(semaforo s) {
8      int j, k;
```




```
9       if(semafori[s].contatore == 0) {
10          Salvataggio_stato();
11          j = processo_in_esecuzione;
12          k = descrittori[j].servizio.priorita;
13          Inserimento(j, semafori[s].coda[k]);
14          Assegnazione_CPU();
15          Ripristino_stato();
16      } else {
17          contatore--;
18      }
19  }
```




```
1   void V(semaforo s) {
2       int j, k, p, q; // j,k: processi; p,q: indici priorita
3       q = 0;
4
5       while(semafori[s].coda[q].primo == -1 && q <
            ↪min_priorita) {
```

```
6           q++;
7       }
8
9       if(semafori[s].coda[q].primo != -1) {
10          k = Prelievo(semafori[s].coda[q]);
11          j = processo_in_esecuzione;
12          p = descrittori[j].servizio.priorita;
13          if(p < q) {
14              // il processo in esecuzione    prioritario
15              Inserimento(k, coda_processi_pronti[q]);
16          } else {
17              // preemption
18              Salvataggio_stato();
19              Inserimento(j, coda_processi_pronti[p]);
20              processo_in_esecuzione = k;
21              Ripristino_stato();
22          }
23      } else {
24          semafori[s].contatore++;
25      }
26  }
```

5.5 Passaggio da ambiente di nucleo all'ambiente processi e viceversa

Il passaggio da ambiente di nucleo all'ambiente di processi e viceversa è basato sul meccanismo di **interruzioni** (interne o asincrone, interne o sincrone). In entrambi i casi, al completamento della funzione richiesta, il passaggio avviene sfruttando il meccanismo di ritorno da interruzione.

Ad ogni processo è associata una pila (stack) gestita tramite il registro stack pointer. La pila rappresenta l'area di lavoro del processo e contiene variabili temporanee ed i record di attivazione delle procedure chiamate.

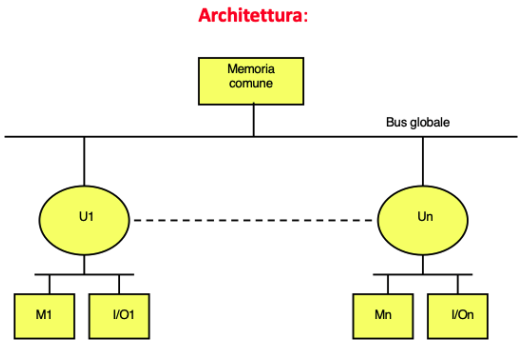
I registri presenti sono: PC e PS (resgistro di stato), R1,...,Rn, R'1,...,R'n, SP1, SP1' (registri generali e stack pointer) associati rispettivamente agli ambienti di nucleo e dei processi.

L'esecuzione di una primitiva da parte di P corrisponde all'esecuzione di una istruzione di tipo SVC:

1. interruzione
2. Salvataggio di PC e PS relativi a P in cima alla pila del nucleo
3. Caricamento in PC e PS dell'indirizzo della procedura di risposta all'interruzione e di PS del nucleo

4. Esecuzione della procedura di risposta all'interruzione con chiamata alla primitiva di nucleo richiesta (es P)
5. P passante: esecuzzioen di ritorno dall'interruzione che ripristina in PC e PS i valori del processo contenuti nella pila del nucleo
6. P bloccante (non passante): salvataggio stato

5.6 Architettura Multiprocessore: realizzazione del nucleo



Sistemi operativi multiprocessore: organizzazione interna

Un sistema operativo che esegue su un'architettura multiprocessore deve gestire una molteplicità di CPU, ugnuna delle quali può accedere alla stessa memoria condivisa.
Esistono principalmente due modelli:

- **Modello SMP**: unica copia del nucleo condivisa tra tutte le CPU
- **Modello a nuclei distinti**: più istanze del nucleo concorrenti

5.6.1 Modello SMP: Simmetric Multi Processing

In questo modello è presente un'unica copia del nucleo del sistema operativo allocata nella memoria comune che si occupa della gestione di tutte le risorse disponibili, comprese le CPU. Ogni processi può essere allocato su una qualunque CPU. Inoltre è possibile che processi che eseguono su CPU diverse richiedano contemporaneamente funzioni del nucleo ad esempio le System Call. E dato che ogni funzione del nucleo comporta un accesso alle strutture dati interne al nucleo, occore regolare gli accessi al nucleo in modo che avvengano in modo sincronizzato.

Sincronizzazione nell'accesso al nucleo:

Soluzione ad un solo lock:
Viene associato al nucleo **un Lock L**, per garantire la mutua esclusione nell'esecuzione di funzioni del nucleo da parte di processi diversi: l'accesso esclusivo alle sue strutture dati può essere ottenuto delimitando il corpo di ogni richiesta per il nucleo con le primitive **lock** e **unlock** applicate all'**unico Lock l**.
Il problema fondamentale di questa soluzione è la **limitazione del grado di parallelismo**, escludendo a priori ogni possibilità di esecuzione contemporanea di funzioni del nucleo che operano su strutture dai distinte.

Soluzione a più lock:
Un maggiore parallelismo può essere ottenuto individuando all'interno del nucleo diverse classi di sezioni critiche, ognuna associata a una struttura dati separata e sufficientemente indipendente dalle altre. Ad ogni struttura dati viene associato un lock distinto.
Ad esempio la coda dei processi pronti, i singoli semafori, ecc, vengono protetti tramite lock distinti.
Ogni operazionie del nucleo che richiederà l'accesso a una particolare struttura

$$S_i$$

protetta dal locl

$$L_i$$

, conterrà una sezione critica il cui prologo ed epilogo saranno rispettivamente

$$lock(L_i)$$

e

$$unlock(L_i)$$

.

Scheduling dei processi:
Il modello SMP consente la schedulazione di ogni processo su uno qualunque dei processori attraverso la **load balancing**, ovvero la possibilità di attuare politiche di distribuzione equa del carico sui processori.
Tuttavia, in alcuni casi può risultare più conveniente assegnare un processo ad un determinato processore:

- i processori possono accedere più rapidaemnte alla loro memoria privata piuttosto che a quella remota, quindi potrebbe convenire schedulare il processo sul processore la cui memoria privata già contiene il suo codice.
- in sistemi NUMA l'accesso alla memoria "più vicina" è più rapido: conviene schedulare il processo sul processore più vicino alla memoria ove è allocato il suo spazio di indirizzamento
- i processori hanno memoria cache. Il processo dovrebbe essere assegnato al processore sul quale era stato precedentemente eseguito.

5.6.2 Modello a nuclei distinti

In questo modello la struttura interna del sistema operativo è articolata su più nucleo, ognuno dedicato alla gestione di una diversa CPU.
L'assunzione di base è che l'insieme dei processi che eseguiranno nel sistema sua partizionabile in tanti sottoinsiemi (nodi virtuali) lascamente connessi, cioè con un ridotto numero di interazioni reciproche:

- Ciascun nodo virtuale è associato ad un nodo fisico: esso è gestito da un nucleo distinto e pertanto tutte le strutture dati del nucleo relative al nodo virtuale vengono allocate sulla memoria privata del nodo fisico
- In questo modo tutte le interazioni locali al nodo virtuale possono avvenire indipendentemente e concorrentemente a quelle di altri nodi virtuali, facendo riferimento al nucleo del nodo.

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

5.6.3 Organizzazione SMP vs nuclei distinti

- **Grado di parallelismo tra CPU**: il modello a nuclei distinti è più vantaggioso, in quanto il grado di accoppiamento tra CPU è più basso. Maggiore scalabilità.
- **Gestione ottimale delle risorse computazionali**:
  - il modello SMP fornisce i presupposti per un migliore bilanciamento del carico tra le CPU perchè lo scheduler può decidere di allocare ogni processo su qualunque CPU.
  - il secondo modello voncola ogni processo ad essere schedulato sempre sullo stesso nodo.

Realizzazione dei semafori nel modello SMP
Tutte le CPU condividono lo stesso nucleo: per sincronizzare gli accessi al nucleo, le struttura dati del nucleo vengono protette tramite lock.
In particolare, i sinfoli semafori e la coda dei processi pronti vengono protetti tramite **lock distinti**.
In questo caso due operazioni P su semafori diversi:

- possono operare in modo **contemporaneo** se non risultano sospensive
- in caso contrario, vengono **sequenzializzati** solo gli accessi alla coda dei processi pronti.

6 Modello a scambio di messaggi

Il modello architetturale di una macchina concorrente è caratterizzato da:

- Ogni processo può accedere esclusivamente alle risorse allocate nella propria memoria locale.
- Ogni risorsa del sistema è accessibile direttamente ad un **solo processo (il Gestore della risorsa)**
- Se una risorsa è necessaria a più processi applicativi, ciascuno di questi (processi clienti) dovrà delegare l'unico processo che può operare sulla risorsa (processo server) all'esecuzione delle operazioni richieste; al termine di ogni operazione il server restituirà al cliente gli eventuali risultati
- In questo modello il concetto di **gestore** di una risorsa coincide con quello di processo server
- Ogni processo, per usifruire dei servizi offerti da una risorsa, dovrà comunicare con il gestore
- Il meccanismo di base utilizzato dai processi per qualunque tipo di interazione è costituito dal **meccanismo di scambio di messaggi**

6.1 Canale di comunicazione

Il concetto fondamentale del modello a scambio di messaggi è il **canale**, ovvero un **collegamento logico mediante il quale due o più processi comunicano**.
È il nucleo della macchina concorrente a realizzare l'astrazione "canale" come meccanismo primitivo per lo scambio di informazioni. Ed è compito del linguaggio di programmazione offrire gli strumenti linguistici di alto livello per:

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

Caratteristiche del canale di comunicazione

I parametri che caratterizzano il concetto di canale sono:

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

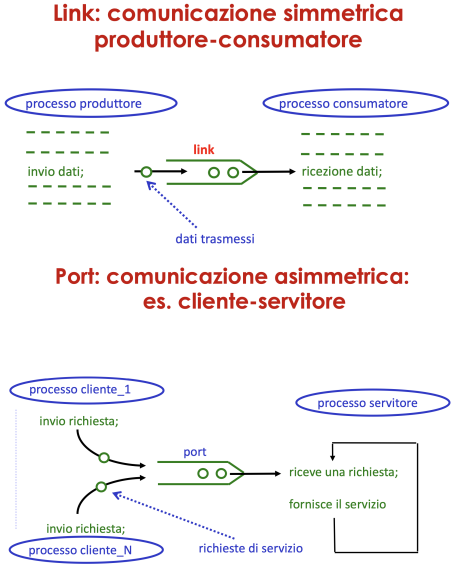
6.2 Tipi di Canali

I canali possono essere distinti in più gruppi. Facendo riferimento alla **direzione del flusso dei dati**, possiamo distinguere due tipi di canali:

- Canale **monodirezionale**: consente il flusso dei messaggi in una sola direzione (mittente verso ricevente)
- Canale **bidirezionale**: può essere usato sia per inviare che per ricevere informazioni

Facendo invece riferimento alla **designazione dei processi comunicanti**, è possibile definire tre tipi di canale:

- **Link**: da uno-a-uno (canale simmetrico)
- **Port**: da molti-a-uno (canale asimmetrico)
- **Mailbox**: da molti-a-molti (canale asimmetrico)



Con riferimento alla modalità di **sincronizzazione** tra i processi comunicanti, possiamo individuare tre tipi di canale:

- Comunicazione **asincrona**
- Comunicazione **sincrona**
- Comunicazione con **sincronizzazione estesa**

6.2.1 Comunicazione asincrona

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

Effetti:

- La ricezione del messaggio può avvenire in un istante successivo all'invio: il messaggio ricevuto contiene informazioni che non possono essere attribuite allo stato attuale del mittente e quindi c'è una difficoltà nella verifica dei programmi.
- L'invio di un messaggio non è un punto di sincronizzazione per mittente e destinatario.

Proprietà:

- **Carenza espressiva**
- **L'assensa di vincoli di sincronizzazione** tra mittente e destinatario favorisce il **grado di concorrenza**
- Da un punto di vista realizzativo, sarebbe necessario un **buffer di capacità illimitata**

Realizzazione:
Ogni implementazione prevede inevitabilmente un limite alla capacità del buffer. Nel caso di un buffer pieno, se si vuole mantenere immutata la semantica, occorre che il supporto a tempo di esecuzione provveda a sospendere il processo che invia il messaggio.

6.2.2 Comunicazione Sincrona: rendez-vous semplice

Semantica:
Il primo dei due processi comunicanti che esegue l'invio (mittente) o la ricezione (destinatario) si sospende in attesa che l'altro sia pronto ad eseguire l'operazione corrispondente.

Effetti:
L'invio di un messaggio è un **punto di sincronizzazione**: ogni messaggio rivcevuto contiene informazioni attribuibili allo stato attuale del processo mittente. Ciò semplifica la scrittura e la verifica dei programmi.

Inoltre **non è necessaria l'introduzione di un buffer**: un messaggio può essere inviato solamente se il ricevente è pronto a riceverlo.

Comunicazione asincrona vs sincrona

- L'invio con semantica sincrona è più espressivo
- La semantica asincrona consente di raggiungere maggiore concorrenza/-parallelismo
- La realizzazione del canale nella comunicazione asincrona richiede opportuna struttura per l'accodamento dei messaggi.

Comunicazione con sincronizzazione estesa: rendez-vous esteso
Assunzione: ogni messaggio inviato rappresenta una richiesta al destinatario dell'esecuzione di una certa azione
Semantica: il processo mittente rimane in attesa fino a che il ricevente non ha terminato di svolgere l'azione richiesta

6.3 Costrutti linguistici e primitice per esprimere la comunicazione

6.3.1 Definizione del canale (port)

Un canale **port** identifica un canale asimmetrico **molti-a-uni**
port ¡tipo¿ ¡identificatore¿;

```
1    port int ch1;
```

L'identificatore ch1 denota un canale utilizzato per trasferire messaggi di tipo intero. **Port** viene dichiarato locale ad un processo (il ricevente) ed è visibile ai processi mittenti mediante la dot notation -¿ ¡nome$_processo$ > . < $identificatore_canale$ >

6.3.2 Primitive di comunicazione sincrone

Send

Send è la primitica che esprime l'invio di un messaggio:

send(¡valore¿) to ¡porta¿;

- **¡porta¿**: identifica in modo univoco il canale a cui inviare il messaggio;
- **¡valore¿**: identifica una espressione dello stesso tipo di ¡porta¿ e rappresenta il contenuto del messaggio inviato.

Semantica: può essere

- **asincrona** -¿ canale bufferizzato;
- **sincrona** -¿ canale a capacità nulla;

A seconda della semantica e delle caratteristiche del canale la send può sospendere o meno il processo che la esegue. Ad esempio:

- **Send sincrona**: il processo attende che il destinatario esegua la primitica di ricezione (receive) corrispondente;
- **Send asincrona**: il processo attende solo se il canale è pieno.

Receive

$$P = \text{receive}(\text{¡variabile¿}) \text{ from } \text{¡porta¿};$$

- **¡porta¿**: identifica il **canale**, locale al processo ricevente, dal quale ricevere il messaggio;
- **¡variabile¿**: è l'identificativo della variabile, dello stesso tipo di ¡porta¿, a cui assegnare il valore del messaggio ricevuto.

Semantica:

- **Default: Semantica Bloccante**. La primirica sospende il processo se non ci sono messaggi sul canale; quando c'è almeno un messaggio nel canale, ne estrae il primo e ne assegna a ¡variabile¿ il valore. La receive restituisce un valore del tipo predefinito **process** che identifica il nome del processo mittente.
- Alcuni linguaggi offrono anche una semantica non bloccante: se il canale è vuoto, il processo continua; se contiene almeno un messaggio, estrae il primo e lo assegna a ¡variabile¿.

6.3.3 Receive Bloccante e Modello Client-Server

Il **Server** è un processo che si occupa di servire le richieste di altri processi, detti **Client**. Il server ha la possibilità di eseguire diversi servizi, ognuno attivato in seguito alla ricezione di un messaggio di tipo diverso. A questo scopo il server gestisce più **canali d'ingresso** (ad esempio porte), ognuno dedicato alla ricezione delle richieste di un particolare servizio.

Il problema principale della recieve bloccante su un modello client-server sta nel fatto che il server specifica il canale sul quale eseguire ogni receive. Se i canali sono più di uno, il server deve eseguire in sequenza le receie su ogni canale: poichè la receive ha semantica bloccante, c'è la possibilità di blocco su un canale mentre contemporaneamente ci sono messaggi in attesa di essere ricevuti su altri canali.

Una possibile soluzione potrebbe essere una **receive con semantica non bloccante**: verifica lo stato del canale, restituisce un messaggio (se presente) o

un'indicazione del canale vuoto (non bloccante). Non sospende mai il processo che la esegue. Il server può eseguire un ciclo di ispezione/ricezione di/da tutti i canali.
In caso di presenza contemporanea di messaggi su più canali la scelta può essere fatta secondo diversi criteri oppure in modo non deterministico.
Problema dell'attesa attiva: se tutti i canali sono vuoti, il server continua a iterare.

Meccanismo di ricezione ideale:

- consente al processo server di verificare contemporaneamente la disponibilità di messaggi su più canali;
- abilità di ricezione di un messaggio da un qualunque canale contenente messaggi;
- quando tutti i canali sono vuoti, blocca il processo in attesa che arrivi un messaggio, qualunque sia il canale su cui arriva.

Questo meccanismo è realizzabile tramite i **comandi con guardia**.

6.3.4 Comandi con Guardia

<**gaurdia**> -> <**istruzione**>

dove **guardia** è costituita dalla coppia: **(espressione booleana; receive)**

- **espressione booleana** viene detta guardia logica;
- **receive** con semantica bloccante e viene detta guardia d'ingresso;

Valutazione della guardia

<guardia> := (<espressione booleana>; <receive>)

La valutazione della guardia può fornire tre diversi valori:

- **guardia fallita**: se l'espressione ha valore false e il comando eseguito con guardia fallisce;
- **guardia ritardata**: se l'espressione booleana ha valore true e nel canale su cui viene eseguita non ci sono messaggi, il processo che esegue il comando viene sospenso e quando arriverà il primo messaggio nel canale riferito dalla guardia d'ingressom il processo verrà riattivato, eseguirà la receive e successivamente l'istruzione;
- **guardia valida**: se l'espressione booleana ha valore true e nel canale c'è almeno un messaggio, quindi il processo esegue la receive e successivamente l'istruzione.

Comando con guardia alternativo

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

Il comando con guardia alternativo (select) racchiude un numero arbitrario di comandi con guardia semplici.
Vengono valutate le guardie di tutti i rami e si possono verificare 3 casi:

- **se una o più guardie sono valide**: viene scelto, in maniera non deterministica, uno dei rami con guardia valida e la relativa guardia viene eseguita, viene quindi eseguita l'istruzione relativa al ramo scelto, e con ciò termina l'esecuzione dell'intero comando alternativo;
- **se tutte le guardie non fallite sono ritardate**: il processo in esecuzione si sospende in attesa che arrivi un messaggio che abilita la transizione di una guardia da ritardata a valida e a quel punto procede come nel caso precedente;
- **se tutte le guardie sono fallite**: il comando termina.

Comando con guardia ripetitivo

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

Vengono valutate le guardie di tutti i rami e si possono verificare 3 casi:

- **se una o più guardie sono valide**: viene scelto in maniera non deterministica uno dei rami con guardia valida e la relativa guardia viene eseguita, viene quindi eseguita l'istruzione relativa al ramo scelto, e poi si passa al ciclo successivo (ripetizione);
- **se tutte le guardie non fallite sono ritardate**: il processo in esecuzione si sospende in attesa che arrivi un messaggio che abilita la transizione da una guardia ritardata a valida; e a quel punto procede come nel caso precedente (si itera);
- **se tutte le guardie sono fallite**: l'esecizione del comando termina.

6.3.5 Primitive di comunicazione asincrone

Nel modello a scambio di messaggi, lo strumento di comunicazione di più basso livello è la **send asincrona**. Nel quale ad ogni operazione associata alla risorsa corrisponde un diverso servizio.

Accesso a risorse "condivise" -¿ processi servitori

Verranno ora analizzate delle **soluzioni di problemi di sicronizzazioen con send asicrona**.
Prendendo in considerazioen alcuni tipici problemi di interazione nel modello a scambio di messaggi. Quindi ci si chiede: data una risorsa, come implementare nel modello a scambio di messaggi (con send asincrona) il gestore della risorsa nel caso in cui sia previsto:

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

1. Risorsa condivisa con una sola operazione

Risorsa condivisa che mette a disposizioen di un insieme di processi "clienti" una sola operazione con il solo vincolo della mutua esclusione.

Soluzione nel modello a memoria comune: il gestore è un **monitor** con **una operazioen entry**

```
monitor gestore {
    tipo_var var;
    <eventuale
        ↪inizializzazione>;
    entry tipo_out fun(
        ↪tipo_in x) {
        <corpo della
            ↪funzione fun>;
    }
}

gestore ris; // istanza del
    ↪ gestore
```

```
thread client {
    tipo_in a;
    tipo_out b;

    b = ris.fun(a);
}
```

Soluzione nel modello a scambio di messaggi:
La risorsa viene gestita da un **processo server** che offre un unico servizio senza condizioni di sincronizzazione.

```
// cliente: un solo canale risposta di tipo tipo_out

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

```
// server: un solo canale input di tipo tipo_in

tipo_out fun(tipo_in x);

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

2. Risorsa condivisa con più operazioni

Risorsa condivisa che mette a disposizione di un insieme di processi clienti due operazioni con il solo vincolo della mutua esclusione.

Soluzione nel modello a memoria comune: il gestore è un monitor con **due operazioni entry**:

```
monitor gestore {
    tipo_var var;
    ...
    entry tipo_out1 fun1(tipo_in1 x1) {
        <corpo della funzione fun1>;
    }
    entry tipo_out2 fun2(tipo_in2 x2) {
        <corpo della funzione fun2>;
    }
}

gestore ris;
```

```
thread client {
    tipo_in1 a1;
    tipo_in2 a2;
    tipo_out1 b1;
    tipo_out2 b2;
    ...
    b1 = ris.fun1(a1);
    ...
    b2 = ris.fun2(a2);
}
```

Soluzione nel modello a scambio di messaggi:
La risorsa è gestita da un processo servitore che offre 2 servizi senza condizioni di sincronizzazione.

- soluzione senza comandi con guardia: un solo canale per entrambi i tipi di richiesta. [codice]
- Soluzione con comandi con guardia: per ogni servizio il server apre un canale distinto. [codice]

3. Risorsa condivisa con più operazioni e condizioni di sincronizzazione

La risorsa è gestita da un processo servitore che offre **due operazioni di sincronizzazione**

Soluzione nel modello a memoria comune: la risorsa è gestita da un monitor con 2 entry e 2 variabili condizione

```
condition c1, c2;
```

```
entry tipo_out1 op1(
    ↪tipo_in1 x1) {
    ...
    if(!cond1)
        wait(c1);
    ...
    signal(c2);
    ...
}
```

```
entry tipo_out2 op2(
    ↪tipo_in2 x2) {
    ...
    if(!cond2)
        wait(c2);
    ...
    signal(c1);
    ...
}
```

Soluzione nel modello a scambio di messaggi:
La risorsa è gestita da **un processo servitore con due servizi**:

- viene associato ad ogni servizio un canale distinto;
- la ricezione delle richieste in arrivo viene realizzata con un **comando con guardia ripetitivo**, con due rami (uno per servizio); in ogni ramo la **guardia logica** esprime la **condizione di sincronizzazione**

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

process server {
    port tipo_in1 input1;
    port tipo_in2 input2;
    tipo_var var;
```

```
8       tipo_in1 x1;
9       tipo_in2 x2;
10      tipo_out1 y1;
11      tipo_out2 y2;
12      ...
13      do {
14          [] (cond1);
15              p = receive (x1) from input1; -> {
16                  y1 = fun1(x1);
17                  send(y1) to p.risposta1;
18              }
19          [] (cond2);
20              p = receive (x2) from input2; -> {
21                  y2 = fun2(x2);
22                  send (y2) to p.risposta2;
23              }
24      }
25  }
```

Corrispondenza tra monitor e processi servitori

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

6.3.6 Esempi

6.3.7 Realizzazione dei meccanismi di comunicazione

Realizzazione delle primitive asincrone

Realizzazione delle primitive di comunicazione de del costrutto port utilizzando gli strumenti di comunicazione offerti dal nucleo del sistema operativo.
Realizzazioen in Architetture **mono/multielaboratore e architetture distribuite**.

Architetture mono e multielaboratore
Ipotesi:

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

```
typedef struct {
    T informazione;
    PID mittente;
    messaggio *successivo;
} messaggio;
```

```
typedef struct {
    messaggio *primo;
    mesasggio *ultimo;
} coda_di_messaggi;
```

```
void inserisci(messaggio *m, coda_di_messaggi c) {
    if (c.primo == null)
        c.primo = m;
    else
        c.ultimo -> successivo = m;

    m->successivo = null;
}
```

```
messaggio *estrai(coda_di_essaggi c) {
    messaggio *pun;
    pun = c.primo;
    c.primo = c.primo->successivo;
    if (c.primo == null)
        c.ultimo = null;
    return pun;
}
```

```
boolena coda_vuota(coda_di_messaggi c) {
    if(c.primo == null)
        return true;
    return false;
}

typedef struct {
    coda_di_messaggi coda;
    p_porta puntatore;
} des_porta;

typedef des_porta *p_porta;

// descrittore del processo
typedef struct {
    p_porta porte_processo[M];
    PID nume;
    modalita_di_servizio servizio;
    tipo_contesto contesto;
    tipo_stato stato;
    PID padre;
    int N_figli;
    des_figlio prole[max_figli];
    p_des successivo;
} des_processo;
```

```
boolean bloccato_su(p_des p, int ip) {
    <testa il campo stato nel descrittore del processo di
        ↪cui p    il puntatore e restituisce il valore true
        ↪se il processo risulta bloccato in attesa di
        ↪ricevere messaggi
    dalla porta il cui indice nel campo porte_processo    ip
        ↪>;
}
```

```
void blocca_su(int ip) {
    <modifica il campo stato del descrittore del
        ↪processo_in_esecuzione per indicare che lo stesso
        ↪si blocca in attesa di messaggi dalla porta il cui
        ↪indice nel campo porte_processo    ip>;
}
```

```
// verifica la presenza di messaggi
void testa_porta(int ip) {
    p_des esec = processo_in_esecuzione;
    p_porta pr = esec->porte_processo[ip];
    if(coda_vuota(pr->coda)) {
        // sospensione
```

```
        blocca_su(ip);
        assegnazione_CPU; // context switch
    }
}

messaggio *estrai_da_porta(int ip) {
    messaggio *m;
    p_des = processo_in_esecuzione;
    p_porta pr = esec->porte_processo[ip];
    m = estrai(pr->coda);
    return m;
}

void inserisci_porta(messaggio *m, PID proc, int ip) {
    p_des destinatario = descrittore(proc);
    p_porta pr = destinatario->porte_processo[ip];
    inserisci(m, pr->coda);
    if(bloccato_su(destinatario, ip))
        attiva(destinatario);
}
```

```
void send(T inf, PID proc, int ip) {
    messaggio *m = new messaggio;
    m->informazione = inf;
    m->mittente = processo_in_esecuzione;
    inserisci_porta(m, proc, ip);
}

void receive(T *inf, PID *proc, int ip) {
    messaggio *m;
    testa_porta(ip);
    m = estrai_da_porta(ip);
    *proc = m->mittente;
    *inf = m->informazione;
}
```

Nelle slide è presente anche l'esempio di ricezione su più canali.

Architetture distribuite
Insieme di nodi tra loro omogenei e tutti dotati dello stesso sistema operativo (stesso nucleo).

Scopo del sistema: geestire tutte le risorse nascondendo all'utente la loro distribuzione sulla rete.

Sistemi operativi di rete (nos): insieme di nodi eterogenei, con sistemi operativi diversi e autonomi, nodo per nodo.
Ogni nodo della rete è in grado di offrire servizi a clienti remoti presenti su altri nodi della rete.

Trasparenza della distribuzione delle risorse viene ottenuta mediante il middleware.
immagini
codice

7 Algoritmi di Sincronizzazione Dstribuiti

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

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

Nelle applicazioni distribuite è importante poter soddisfare alcune proprietà:

- **scalabilità**: nell'applicazione distribuita le prestazioni dovrebbero aumentare al crescere del numero di nodi utilizzati;
- **tolleranza ai guasti**: l'applicazione è in grado di funzionare anche in presenza di guasti.

Prestazioni: speedup

Oltre al tempo di calcolo, un indicatore per misurare le prestazioni di un sistema parallelo/distribuito è dato dallo speedup.
Ipotizzando che la stessa applicazione possa essere eseguita su un numero variabili di nodi:
lo speedup per n nodi è il rapporto tra il tempo di esecuzione dell'applicazione ottenuto con un solo nodo e quello ottenuto con n nodi:

$$Speedup(1) = \frac{Tempo(1)}{Tempo(n)}$$

Tolleranza ai guasti

Il sistema distribuito tollerante ai guasti riesce a erogare i propri servizi anche in presenza di guasti in uno o più nodi.
Tipi di guasto:

- Transienti;
- Intermittenti;
- Persistenti.

Un sistema tollerante ai guanti deve nascondere i guasti agli altri processi. Questo obiettivo può essere raggiunto, ad esempio, con tecniche di **ridondanza**: vengono mantenute più istanze degli stessi componenti, in modo da poter rimpiazzare l'elemento guasto con un elemento equivalente.

Algoritmi di Sincronizzazione

Come nel modello a memoria comune, anche nel modello a scambio di messaggi è importante poter disporre di algoritmi di sincronizzazione tra i processi concorrenti, che consentano di risolvere alcune problematiche comuni, coordinando i vari processi.
Eímportante che gli algoritmi distribuiti godano delle proprietà di scalabilità e di tolleranza ai guasti.

7.1 Algoritmi per la gestione del tempo

Tempo nei sistemi distribuiti

In un sistema distribuito, ogni nodo è dotato di un proprio orologio. Se gli orologi locali di due nodi non solo sicronizzati, 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 temporali predente quello di e1.

Quindi in applicazioni distribuite può essere necessario:

- disporre di un orologio fisico universale, se c'è bisogno di utilizzare l'ora esatta: a questo scopo vengono utilizzati degli algoritmi di sincronizzazione dei nodi del sistema (es. algoritmo di Berkeley, Christian)
- disporre di un orologio logico, che permetta di associare ad ogni evento un istante logico (timestamp) coerente con l'ordine in cui essi si verificano.

7.1.1 Orologi Logici (Lamport, 1979)

In un'applicazione distribuita, gli eventi sono legati da vincoli di precedenza che danno origine ad una relazione d'ordine parziale (v. algoritmi non sequenziali)

Relazione di precedenza tra eventi (happened-before, $\rightarrow$):

- se a e b sono eventi in uno stesso processo ed a si verifica prima di b: a$\rightarrow$b
- se a e l'evento di invio di un messaggio e b è l'evento di ricezione dello stesso messaggio: a$\rightarrow$b

- se a→b e b→c allora a→c

Data una coppia di eventi (a,b) sono possibili 3 casi:

- a→b, cioè a avviene prima di b;
- b→a, b avviene prima di a;

Assumiamo che ad ogni evento **e** 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 definire un timestamp **C(a)** sul quale tutti i processi siano d'accordo.
Il tempo C(a) deve soddisfare la seguente proprietà:

$$\textbf{se a}\rightarrow\textbf{b allora C(a)} < \textbf{C(b)}\ [*]$$

Quindi:

- se all'interno di un processo **a precede b** allora **C(a)** $<$ **C(b)**
- se **a** è l'evento di **invio** (in un processo Ps) e **b** l'evento di **ricezione** (in un processo Pr) dello stesso messaggio m allora **C(a)** $<$ **C(b)**

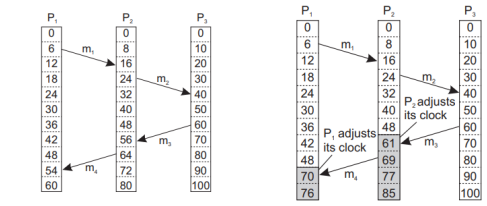
7.1.2 Algoritmo di Lamport

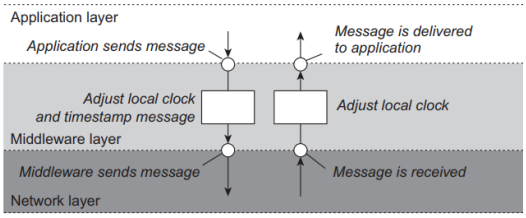
Per garantire il rispetto della proprietà [*] ogni processo Pi gestisce localmente un contatire del tempo logico Ci, che viene gestito nel modo seguente:

- ogni nuovo evento all'interno di Pi provoca un incremento del calore di Ci: Ci=Ci+1
- ogni volta che Pi invia un messaggio m, il contatore Ci viene incrementato: Ci=Ci+1 e successivamente al messaggio viene associato il timestamp Ci: ts(m)=Ci
- quando un processo Pj riceve un messaggio m, assegna al proprio contatore Cj un valore uguale al massimo tra Cj e ts(m): Cj=max{Cj, ts(m)} e successivamente lo incrementa di 1: Cj=Cj+1

Orologio logioco di Lamport: esempio
Considerando 3 processi in esecuzione du 3 nodi, ognuno con il proprio clock:

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





7.2 Mutua Esclusione Distribuita

L'obiettivo della mutua esclusione distribuita è quello di garantire che due o più processi non possano eseguire contemporaneamente certe attività. Ad esempio accesso a risorse condivise come file, stampanti, ecc..

Esistono 2 principali soluzioni:

- **centralizzata**: la risorsa è gestita da un processo dedicato (coordinatore), al quale tutti i processi si rivolgono per accedervi;
- **decentralizzata**: non è previsto un coordinatore, quindi i processi in competizione si sincronizzano tra loro tramite opportuni algoritmi, la cui logica è distribuita tra tutti i processi.

Una soluzione decentralizzata è, in generale, più scalabile di soluzioni centralizzate, nelle quali il processo gestore della risorsa rappresenta un collo di bottiglia.

Soluzioni al problema della mutua esclusione distribuito

In generale, possiamo suddividere le soluzioni in due categorie:

- **Permission-based**: ogni processo che vuole eseguire la sua sezione critica, richiede un permesso ad uno o più processi;
- **Token-based**: un "testimone" (token) viene passato tra i vari processi in competizione: il processo che possiede il token può:
  - eseguire la sua sezione critica;
  - passare il token ad un altro processo, se non intenzionato ad entrare nella sezione critica.

Gli algoritmi token-based sono sempre decentralizzati, mentre gli algoritmi permission-based possono essere sia centralizzati che decentralizzati.

7.2.1 Mutua esclusione: soluzione centralizzata

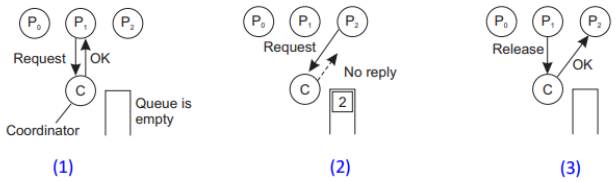
L'algoritmo si basa su **permessi** (permission-based):

- la risorsa viene gestita da un processo coordinatore al qiale ogni processo che vuole eseguire la sua sezione critica si rivolge per ottenere il permesso.
- per eseguire la propria sezione critica ogni proccesso Pi:
  - **Richiesta**: Pi invia una richiesta di autorizzazione al coordinatore; quando verrà accolta, il processo Pi otterrà il permesso.
  - viene eseguita la sezione critica
  - **Rilascio**: Pi comunica al coordinatore il termine della sezione critica

Il coordinatore **concede un permesso alla volta**: ogni Richiesta ricevuta da un processo Pi mentre l'autorizzazione è concessa al processo Pj viene messa in attesa. Inoltre il coordinatore mantiene una coda delle Richieste in attesa.

Esempio:

- P1 chiede il permesso al coordinatore, che autorizza l'accesso;
- P2 chiede il permesso al coordinatore, che non può concederlo perchè P1 sta eseguendo la sua sezione critica; il coordinatore non risponde e inserisce la richiesta di P2 nella coda;
- quando P1 termina la sua sezione critica, invia un messaggio di Rilascio al coordinatore, che quindi può successivamente autorizzare P2.



Vantaggi:
L'algoritmo è equo, e quindi non c'è starvation. Inoltre prevede solo 3 messaggi per ogni sezione critica: richiesta, autorizzazione e rilascio.

Svantaggi:

Scalabilità: la soluzione è poco scalabile perchè al crescere del numero dei processi il coordinatore piò diventare un collo di bottiglia.
Tolleranza ai guasti:

- il coordinatore può essere soggetto a guasto e, in questo caso, l'intero sistema si blocca (single point of failure)
- inoltre, se un processo P che ha fatto una richiesta non ottiene una risposta, P non può distinguere le due possibili cause:
  - autorizzazione non concessa;
  - coordinatore guasto

7.2.2 Mutua esclusione: Algoritmo Ricart-Agrawala

Algoritmo decentralizzato basato su permessi (permission-based). Il sistema è costituito da un insieme di processi in competizione. Ad ogni processo sono associate 2 attività concorrenti (thread):

- **main**: il thread che esegue la sezione critica
- **receiver**: thread dedicato alla ricezione delle autorizzazioni

Questo algoritmo ha un requisito. Ovvero è richiesta la presenza di un temporizzatore globale per tutti i nodi (orologio logico): ogni messaggio è corredato da un timestamp che ne indica l'istante di invio.

Struttura del main:
Quando um processo vuole entrare nella sezioen critica:

1. invia (n-1) richieste di autorizzazioen ai receiver degli altri (n-1) nodi: **Request(Pid, timestamp)**;
2. attende le **(n-1) autorizzazioni**;
3. esegue la **sezione critica**;
4. invia **ok** a tutte le richieste in attesa.

Struttura del receiver:
Quando riceve una richiesta, il receiver può trovarsi in uno di tre possibili stati:

1. **Released**: il processo non è interessato ad entrare nella sezione critica: risponde OK
2. **Wanted**: il processo è in procinto di entrare nella sezione critica (attende autorizzazione): confronta il timestamp Tr della richiesta ricevuta con quello della richiesta inviata Ts:
   - se Tr<Ts risponde OK

- altrimenti non risponde e mette la richiesta ricevuta in coda

3. **Help**: il processo sta eseguendo la sezione critica: la richiesta viene messa in coda.