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.

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

$$D(S) = < o, diritti > |oèunoggetto, dirittièuninsiemedioperazioni$$

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
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].
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.
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$_1, GID_1 :< insiemedidiritti > ****$UID_2, GID_2 :< insiemedidiritti >$ $** DoveUIDèlouseridentifiereGIDèilgroupidentifier.$
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.
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.
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.
.. ..

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 applicazione viene strutturata come un insieme di componenti, suddiviso in due sottoinsieme disgiunti:

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

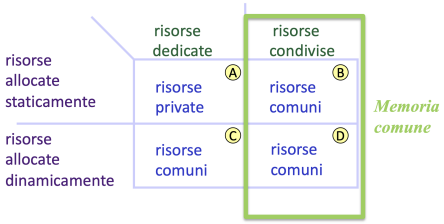
Le Risorse rappresentatno un qualunque oggettim fisico o logico, di cui un processo necessita per portare a termine il suo compito. Le risorse vengono raggruppate in classi, dove una classe rappresenta l'insieme di tutte e sole le operazioni che un processo può eseguire per operare su risorse di quella 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:

```
<prologo>
    S;
<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**:

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

Al tipo semaphore sono associati:

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

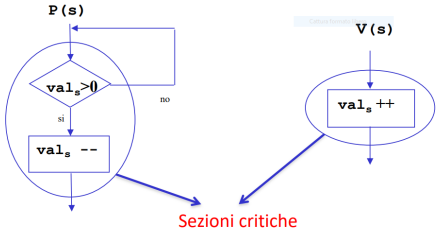
$$\text{Insieme delle operazioni} = \{P, V\}$$

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:

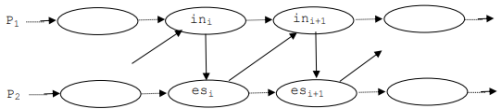
- uno solo di essi sia inizializzato a 1 e tutti gli altri a 0
- ogni processo che usa questi semafori esegue sempre sequenze che iniziano con la P su uno di questi e termina con la V su un altro.

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  }
```

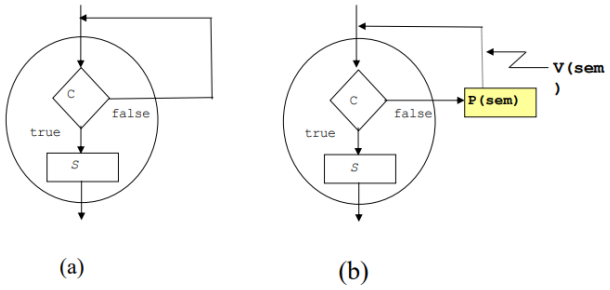
```
1  T ricezione() {
2      T dato;
3      P(pn);
4      dato = estrai()
           ↪;
5      V(vu);
6      return dato;
7  }
```

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.



(a) (b)

- Lo schema (a) presuppone una forma di attesa attiva da parte del processo che non trova soddisfatta la condizione
- Nello schema (b) si realizza la region sospendendo il processo sul semaforo sem da associare alla condizione.

– 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;
    }

    public ovid rilascio(int r) {
        P(mutex);
        libera[r] = true;
        V(mutex);
        V(n_ris);
    }
}
```

Leggere esempi sulle condizioni di sincronizzazione.

Semaforo condizione

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 allo-

cazione 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**:

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

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

    public void rilascio() {
        int i;
        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);
    }

    public void rilascio() {
        int i;
        P(mutex);
        <rilascio della risorsa>;
        if(<esiste almeno un processo sospeso per il quale
            ↪la condizione risulta true>) {
            <scelta del processo Pi da riattivare>;
            V(priv[i]);
        } else {
            V(mutex);
        }
    }
}
```

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.