Indice

Capitolo 1

Virtualizzazione

1.1 Concetti generali sulla virtualizzazione

1.1.1 Definizione generale di virtualizzazione

Il termine virtualizzazione è molto generale, per cui segue una definizione che può essere:

Definizione 1: ***Virtualizzazione***

Dato un sistema costituito da un insieme di risorse hardware e software, ***virtualizzare il sistema*** *significa presentare all'utilizzatore una visione del sistema diversa da quella reale (fisica).*

Dunque, la virtualizzazione è resa possibile grazie all'applicazione di meccanismi soprattutto software (ma anche hardware) che si occupino di introdurre di un **livello di indirezione** che separa la *vista reale* (fisica) da quella dell'utilizzatore (vista logica).

Definizione 2: ***Tecnologie di virtualizzazione***

Insieme di meccanismi volto a disaccoppiare la rappresentazione delle risorse di un sistema dall'effettiva realizzazione fisica. Le risorse, dunque, appaiono all'utente con caratteristiche generalmente diverse da quelle reali.

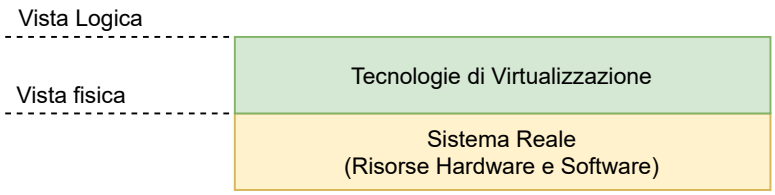


Figura 1.1: Tecnologie di virtualizzazione

L'esempio più pratico di virtualizzazione per un corso di informatica è, senza dubbio, la *macchina virtuale*, schematizzata dalla figura 1.2: un opportuno software (Virtual Machine Monitor o *Hypervisor* permmette al sistema di offrire all'utilizzatore una molteplicità di interfacce, ognuna con una diversa macchina virtuale. In sostanza, su un unico

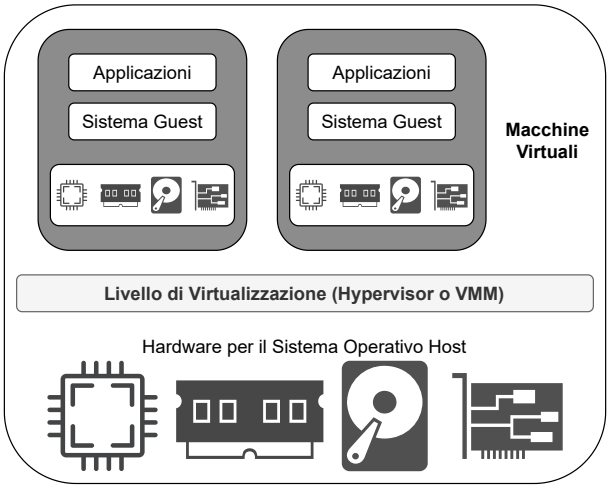


Figura 1.2: Architettura di sistemi virtualizzati

hardware, insistono più macchine virtuali indipendenti, ognuna con un proprio sistema operativo e un proprio ambiente di esecuzione su cui possono girare le applicazioni.

Ad ogni modo, oltre alle macchine virtuali, sono esempi di virtualizzazione:

- **Virtualizzazione a livello di processo**
  tipica dei sistemi *multitasking* che permettono l'esecuzione concorrente di più processi, ognuno dei quali opera come se avesse a disposizione una macchina virtuale dedicata, intesa come l'insieme di CPU, memoria e dispositivi. La virtualizzazione è realizzata dal kernel del sistema operativo.

- **Virtualizzazione della memoria**
  meccanismo molto diffuso nei sistemi operativi in cui è presenta una memoria virtuale: ogni processo ha una visione diversa della memoria rispetto alla situazione reale grazie alla quale esso può eseguire avendo l'illusione di avere a disposizione tutto lo spazio di indirizzamento di cui ha bisogno. Anche in questo caso, la virtualizzazione è resa possibile dal kernel.

- **Astrazione**
  possibilità di rappresentare in modo semplificato un oggetto, in modo tale che vengano esibite le proprietà importanti utili all'utilizzatore ma nascondendo i dettagli che non sono necessari (es: tipo di dato).

- **Linguaggi di programmazione**
  virtualizzazione realizzata da interpreti e compilatori che, dunque, permette i meccanismi di portabilità di un programma nelle diverse architetture.

L'aspetto della virtualizzazione su cui ci si concerterà durante il corso è la **virtualizzazione di un sistema di elaborazione**. Inoltre, si definiscono:

- **Guest**
  è la singola macchina virtuale, compresa di sistema operativo ed applicazioni.

- **Host**
  ambiente sottostante che ospita le macchine virtuali, compreso di macchina fisica e del componente che si occupa della loro gestione (VMM).

1.1.2 Suddivisione a livelli per la virtualizzazione

Come si è anticipato, la virtualizzazione è un concetto molto generale. Tuttavia, è possibile fare una suddivisione a livelli che raggruppa le diverse tipologie di virtualizzazione, alcune introdotte nella sezione precedente. Si procederà dal livello *più basso* a quello *più alto*:

1. **Instruction Set Architecture Level**
   È la classica emulazione che sarà introdotta a breve e che permette di proporre ad un processore diverso l'ISA di un altra cpu per garantire la possibilità ad applicazioni compilate per un sistema differente di eseguire anche sull'host.

2. **Hardware Abstraction Layer (HAL)**
   Questo livello di virtualizzazione si occupa di offrire un'interfaccia alle macchine virtuali: di fatto, viene astratta l'intera architettura hardware, la cui interfaccia viene replicata tante volte quante sono le macchine virtuali che si vogliono installare. Ogni macchina virtuale è equipaggiata con un proprio sistema operativo e appare, dunque, all'utente come una macchina completa a sé stante ed equivalente ad una macchina fisica.

3. **Operating System Level**
   Questo livello di virtualizzazione permette di definire su uno stesso sistema operativo più ambienti di esecuzione diversi e isolati tra loro, detti container, che appaiono simili a delle macchine virtuali. I programmi in esecuzione su un container, ovviamente, possono vedere esclusivamente le risorse che sono state assegnate a quello stesso container, a differenza delle applicazioni che girano su un sistema tradizionale che, invece, possono usufruire della totalità delle risorse disponibili. Rispetto al livello sottostante, dunque, la differenza sostanziale è che tutti i container condividono lo stesso sistema operativo e, inoltre, le prestazioni sono decisamente migliori delle normali macchine virtuali.

4. **Libray Level (User-Level API)**
   Questa tipologia di virtualizzazione si colloca sopra al sistema operativo e permette di mettere a disposizione libreria di sistemi operativi diversi rispetto a quello host.

5. **Application Level**
   Quest'ultimo livello è quello più alto e consente di realizzare la portabilità dei programmi tra i diversi sistemi operativi: tipicamente, un componente software si occupa di eseguire sul sistema host un codice *pseudo-compilato*, come accade ad esempio nel caso di `Java` o `.NET`.

1.1.3 Virtual Machine Monitor

Ogni piattaforma hardware può essere condivisa da più elaboratori virtuali indipendenti e disaccoppiate grazie ad un componente chiamato *Virtual Machine Monitor (VMM)* o *Hypervisor*. L'immagine di riferimento di questa tipologia di virtualizzazione è già stata mostrata ed è la 1.2.

Definizione 3: ***Virtual Machine Monitor***

Componente software, ma in alcuni casi anche hardware, che permette a più macchine virtuali di eseguire su una stessa piattaforma hardware e che, dunque, si occupa di:

- *fornire alle macchine virtuali un'**interfaccia privata** dell'hardware sottostante (spesso ne è una replica);*
- *gestire la **condivisione delle risorse** hardware da parte delle macchine virtuali (sincronizzazione tra le macchine virtuali che competono per l'utilizzo delle stesse risorse).*

Si specifica che ogni macchina virtuale contiene al proprio interno un proprio sistema operativo indipendente ed autonomo, su cui girano le applicazioni e che può essere diverso dai sistemi delle altre macchine che eseguono sullo stesso sistema fisico: ogni VM, allora, ha l'illusione di essere l'unica in esecuzione sul sistema di elaborazione senza avere coscienza della presenza di altre macchine. Il VMM, pertanto, si occupa di realizzare i meccanismi di **isolamento** tra le VM e di **stabilità** generale dell'intero sistema.

1.1.4 Emulazione

Tipicamente, in un sistema non virtualizzato, per l'interazione con la CPU si utilizza il *linguaggio macchina* e, dunque, un set di istruzioni attraverso le quali il kernel del sistema operativo può richiedere all'hardware determinate operazioni. In un sistema virtualizzato, invece, è il VMM a presentare a tutte le macchine virtuali lo stesso set di istruzioni replicato.

Esiste, però, un'ulteriore tipologia di virtualizzazione: l'emulazione.

Definizione 4: ***Emulazione***

Virtualizzazione che fornisce all'utilizzatore la possibilità di eseguire programmi compilati per una particolare architettura su un sistema di elaborazione diverso e dotato di un set di istruzioni differente.

Ciò che viene interamente emulato sono le singole istruzioni dell'architettura ospitata che, dunque, possono girare su piattaforme differenti da quella per cui sono state pensate. Questo, però, comporta il fatto di dover tradurre le istruzioni del sistema ospitato in una o più istruzioni della macchina ospitate con il corrispondente costo in termini di performance ed efficienza dell'esecuzione. Le metodologie per rendere possibile questo meccanismo sono sostanzialmente due:

- **Interpretazione**
  ogni singola istruzione viene opportunamente tradotta nel linguaggio che la piattaforma sottostante è in grado di capire ed eseguire. Il costo, tuttavia, non è trascurabile poichè una singola istruzione da emulare potrebbe richiedere una serie di istruzioni che l'host può comprendere.
- **Complazione dinamica**
  a tempo di esecuzione viene letto il prossimo blocco di codice da eseguire che viene

poi tradotto e, se possibile, vengono applicate delle tecniche di ottimizzazione. Procedendo in questo modo, è possibile avere prestazioni migliori rispetto alla classica interpretazione (la compilazione dinamica interviene meno volte dell'interpretazione e, inoltre, parti di codice potrebbero essere salvate per poter essere riutilizzate).

Sono emulatori *QEMU*, *Virtual PC* e *Mame*.

1.1.5 Cenni storici

La virtualizzazione non è una tecnologia così moderna, benché oggi sia particolarmente diffusa. I primi esempi di sistemi virtualizzati risalgono agli anni '60, quando IBM propose una soluzione per la realizzazione di un sistema multi-utente che chiamò CP/CMS. La sua architettura è strutturata a due livelli: il più basso è chiamato *CP* ed è collocato direttamente sull'hardware con il ruolo di VMM, il secondo, invece, è il *CMS* che svolgeva il ruolo di sistema operativo elementare e monoutente con capacità piuttosto limitate, che veniva replicato per ogni utente. Con il tempo, inoltre, questo sistema si è evoluto concedendo la possibilità di installare sistemi operativi differenziati al di sopra del CP.

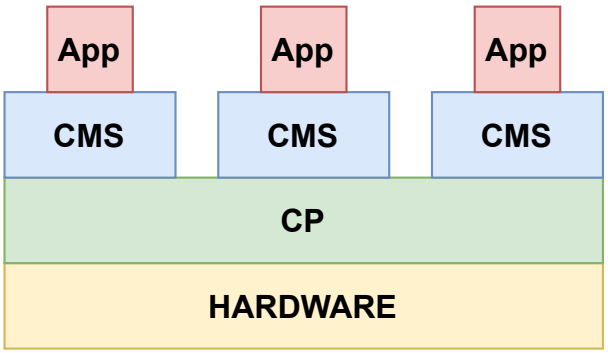


Figura 1.3: Architettura del sistema CP/CMS di IBM

La concorrenza, dunque, era concessa grazie all'impiego di più CMS, uno per ogni utente. Negli anni '70, però, i sistemi operativi sono progrediti a tal punto da diventare *multi-tasking* e, dunque, la multi-utenza era consentita in modo nativo dal sistema.

Nella decade ancora successiva, negli anni '80/'90, lo sviluppo delle architetture ebbe un'accelerazione notevole grazie alla diffusione dei sistemi a microprocessore e a quella dei Personal Computer, con il conseguente crollo del costo dell'hardware e aumento generale delle performance che portano alla migrazione da mainframe a semplici PC.

La soluzione CP/CMS di IBM, allora, venne sempre più accantonata a favore del paradigma *one application, one server*: se un singolo PC aveva raggiunto dei costi bassi per un'azienda negli anni '90, la necessità di isolare una certa applicazione tra i vari utenti, portò alla conseguente scelta di installare tale programma su una macchina scelta e totalmente dedicata ad esso. Dunque, ogni servizio veniva istallato su una singola macchina interamente riservato ad esso. Questo, però, portò ad un'esplosione di server fisici e, dunque, ad un notevole aumento del lavoro di gestione e manutenzione senza trascurare i costi indotti dall'uso contemporaneo di più macchine fisiche; inoltre questa opzione comportava anche un notevole spreco delle risorse hardware che erano inutilizzate per la fruizione del singolo servizio.

La considerazione di questi ultimi aspetti, nei primi anni 2000, ebbe come conseguenza la necessità di razionalizzare le scelte fatte e, quindi, fu riconsiderata l'opzione della

virtualizzazione come scelta valida. Nel 1999, infatti, VMWare propose una delle prime versioni moderne delle tecnologie di virtualizzazione fino ad arrivare agli anni 2010, con l'avvento del *Cloud Computing*[1].

1.1.6 Vantaggi della virtualizzazione moderna

La virtualizzazione, intesa a livello hardware, prevede che sia presente un Virtual Machine Monitor che replica l'hardware sottostante fornendo le interfacce necessarie alle macchine virtuali per la loro esecuzione concorrente.

Questa nuova tecnologia presenta diversi vantaggi:

- **Consolidamento dell'Hardware**
  La concentrazione di più macchine in un'unica architettura hardware porta ad un abbattimento dei costi dell'hardware (ad esempio dello spazio fisico) e dell'amministrazione (meno macchine da amministrare). Il paradigma *one application, one server*, allora, si trasforma in *one application, one VM*.

- **Molteplicità di sistemi operativi su uno stesso hardware**
  Su uno stesso host è possibile utilizzare una molteplicità di sistemi, anche diversi, ognuno isolato. In questo modo, ad esempio, è possibile eseguire applicazioni concepite per sistemi diversi sulla stessa macchina fisica.

- **Effettuazione sicura di test**
  Si possono utilizzare le macchine virtuali per testare le applicazioni sviluppate senza rischio di compromettere il sistema ospitante e in modo sicuro (sandbox).

- **Sicurezza e isolamento**
  Le macchine virtuali sono tutte isolate e, dunque, se ne viene attaccata una il resto del sistema e le altre VMs rimangono integre e non risentono degli effetti.

- **Gestione facilitata delle VMs**
  Amministrare macchine virtuali è molto più semplice rispetto alla gestione di un sistema fisico, in particolare per quanto riguarda le operazioni *time-consuming*, grazie alla possibilità di effettuare in modo agevolato operazioni di creazione, amministrazione e migrazione anche a caldo delle VMs (*workload balancing*, meccanismi di migrazione automatizzati sulla base del carico computazione). Addirittura, molte volte è possibile reperire delle macchine virtuali già pronte e configurate (sotto forma di *virtual appliances*), che sono facilmente importabili nel sistema host.

Uno degli aspetti più potenti tra quelli elencati è la possibilità della **migrazione a caldo**: se c'è la necessità di applicare degli upgrade o delle modifiche ad una macchina fisica (ad esempio, aumenti di RAM o cambio di dischi) è possibile migrare a caldo le macchine virtuali su un altro nodo, in modo tale da non interrompere i servizi in esecuzione, per poi intervenire sul sistema originale e riportare sul nodo originale, nuovamente a caldo, le macchine spostate. Tipicamente, questa tipologia di migrazione è un processo talmente ottimizzato che l'utilizzatore difficilmente percepisce l'istante di malfunzionamento dovuto alla parte centrale del trasferimento delle VMs.

[1]Per *Cloud Computing* si intende la possibilità di erogare da parte di un sistema, detto cloud, delle risorse computazioni remota sotto forma di servizi. L'allocazione reale delle risorse, tuttavia, è ignota all'utilizzatore e, per questo motivo, la tecnologia ha il nome di *cloud*.

La migrazione a caldo, inoltre, è molto utile per far fronte a situazioni di sbilanciamento del carico: trasferendo una macchina in modo periodico o quando si verificano situazioni critiche, è possibile migliorare la situazione complessiva ridistribuendo, dunque, il carico per far fronte allo svantaggio.

Infine, si sottolinea che le macchine virtuali sono utilizzabili per attività di *disaster recovery* che consistono in una serie di strategie che consentano di far fronte a situazioni che minacciano l'operatività di un insieme di computer dedicati all'elaborazione dei servizi di un'azienda o un ente (*datacenter*). Disastri di questo tipo, ovviamente, possono essere terremoti, attentati, incendi e così via... di fronte ad una minaccia, allora, per garantire la continuità dei servizi è possibile sfruttare la migrazione per trasferire i servizi in un ambiente sicuro (ad esempio sale server di riserva in spazi particolarmente protetti) rispetto a quello dove c'è la criticità.

1.2 Il Virtual Machine Monitor

1.2.1 Requisiti per la realizzazione del VMM

Il Virtual Machine Monitor è un componente fondamentale per fini di virtualizzazione in quanto permette l'esecuzione di più macchine virtuali su uno stesso sistema hardware. Il compito principale del VMM, come già detto, è replicare la stessa visione dell'hardware sottostante alle varie macchine virtuali che eseguono sul sistema in modo da garantire le risorse virtuali necessarie al loro funzionamento.

In generale, un sistema virtualizzato deve soddisfare i seguenti requisiti[2]:

- **Equivalenza tra l'ambiente di esecuzione fisico e virtuale**
  In generale, le applicazioni che eseguono sul sistema virtualizzato devono produrre gli stessi effetti che produrrebbero su quello fisico.

- **Efficienza nell'esecuzione**
  La virtualizzazione del sistema deve garantire, il più possibile, efficienza nell'esecuzione: il tempo necessario affinché un programma termini su una macchina virtuale non deve essere tanto più elevato rispetto a quello che si otterrebbe eseguendo su quella fisica. Si ricorda che il set di istruzioni di un processore può essere suddiviso in istruzioni privilegiate e non privilegiate. Per quelle privilegiate, purtroppo, sono presenti diversi problemi di efficienza mentre le seconde vengono eseguite direttamente sull'hardware dal programma che esegue nell'ambiente virtualizzato.

- **Stabilità e sicurezza**
  Ciò che accade all'interno di una macchina virtuale non deve interferire né sull'operatività del sistema, né con le attività che si svolgono all'interno delle altre macchine virtuali che vi lavorano. Questo significa che c'è la necessita di attribuire esclusivamente al **VMM il compito di controllare le risorse hardware in modo completo**.

Dunque, il VMM deve gestire l'hardware e filtrarne, secondo le proprie caratteristiche, l'accesso ai propri utilizzatori (le VMs): segue, di conseguenza, che il Virtual Machine Monitor è l'unica parte del sistema che esegue in modo privilegiato, ovvero a livello kernel.

[2]Popek e Goldberg, *Formal Requirements for Virtualizable Third Generation Architectures*, 1974.

Va da sé che tutto ciò che viene eseguito all'interno di una macchina virtuale non può accedere in modo diretto al livello privilegiato.[3]

1.2.2 Tipologie di VMM

Inoltre, si distinguono:

Definizione 5: ***Virtual Machine Monitor di sistema***

VMM che esegue ***direttamente*** *sull'hardware e che, dunque, sostituisce il tradizionale kernel del sistema operativo.*

Definizione 6: ***Virtual Machine Monitor ospitato***

L'interfaccia delle macchine virtuali (VMM) è rappresentata da un'applicazione che esegue sopra un sistema operativo pre-esistente.

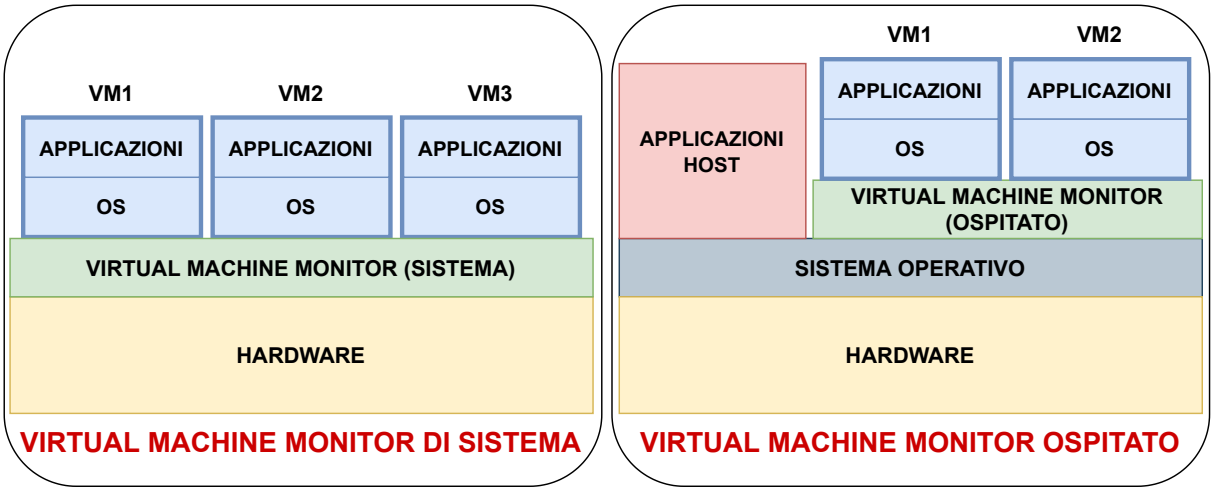


Figura 1.4: Tipologie di Virtual Machine Monitor

A differenza di quelli di sistema, i VMM ospitati eseguono come applicazioni per cui, in questo caso, non è direttamente questo componente che controlla l'hardware bensì sfrutta le funzionali del sistema operativo su cui gira per esercitare il controllo dell'hardware. Tipicamente, questa tipologia di monitor performance peggiori del primo ma è più facilmente installabile e gestibile[4].

[3]Si ricorda che ogni CPU prevede, in generale, almeno due livelli di esecuzione (detti anche *ring di protezione*): quello *supervisor* o *kernel*, il livello 0, in cui è possibile eseguire istruzioni privilegiate, e quello *utente* in cui, invece, è possibile effettuare solo operazioni non privilegiate. Tipicamente il livello 0 esiste sempre ma è possibile che siano presenti più livelli superiori ad esso, non solo quello utente. Si nota, infine, che le istruzioni privilegiate, ovvero quelle che corrispondono ad operazioni più critiche, possono essere eseguite se e solo se il modo corrente di esecuzione impostato sul processore è il livello 0 e questo, ovviamente, introduce un ambiente di protezione.

[4]Infatti, i meccanismi di accesso all'hardware sono demandati al sistema operativo sottostante, rendendo di fatto il VMM ospitato meno complesso rispetto alla sua controparte. D'altro canto, questa tipologia di operazioni comporta chiamate a *system call* che, come dovrebbe essere noto, introducono un costo computazionale non trascurabile.

1.2.3 Virtualizzazione pura e paravirtualizzazione

Oltre a questa prima differenziazione, fatta nella sezione precedente, è doveroso definire:

Definizione 7: ***Virtualizzazione pura***

Tipologia di virtualizzazione secondo cui ogni macchina virtuale ha una visione dell'architettura che è un'esatta replica di quella del sistema fisico. Le VMs, allora, utilizzano la stessa interfaccia (intesa come istruzioni macchina) dell'architettura fisica. Dunque, in questo caso, la tecnologia e tutti i meccanismi che vanno ad implementare la virtualizzazione vengono integrati in un componente software che si appoggia direttamente sopra al sistema fisico, il Virtual Machine Monitor, che fornisce un'interfaccia alle macchine virtuali che è identica a quella dell'hardware reale.

Si potrebbe dire, di fatto, che il Virtual Machine Monitor non è altro che un *sistema operativo leggero*, dato che i meccanismi che esso deve mettere in atto sono molto simili a quelli del kernel di un normale OS. Ad esempio, se il sistema operativo deve gestire i processi, similmente il VMM deve occuparsi dell'esecuzione delle macchine virtuali. Tuttavia, non tutti i compiti sono uguali o trovano una corrispondenza: ad esempio il VMM non deve occuparsi della gestione del file system, mentre quella della memoria è distribuita tra quest'ultimo e le macchine virtuali (esse possono chiedere al VMM di allocare memoria). Si noti che il VMM di sistema **deve contenere tutti i driver necessari al funzionamento delle periferiche**.

Definizione 8: ***Paravirtualizzazione***

Tipologia di virtualizzazione secondo cui il VMM presenta alle macchine virtuali un'interfaccia diversa da quella dell'architettura reale.

Concludendo la suddivisione appena fatta, si potrebbe indicare la paravirtualizzazione come una forma blanda dell'emulazione dato che contiene diversi aspetti di quest'ultima. Il perché sarà chiaro a breve.

1.2.4 Realizzazione del Virtual Machine Monitor di sistema

Problematiche principali: *Ring Deprivileging*, *Ring Compression* e *Ring Aliasing*

Si è detto in precedenza che nel caso di un monitor di sistema, il VMM deve essere l'unica componente autorizzata a controllare completamente l'hardware. Segue, allora, che in un sistema virtualizzato il VMM è l'unico programma ad eseguire a livello kernel (ring 0): tutto il resto deve collocarsi sopra. Questo, allora, significa che **anche i kernel dei sistemi operativi delle macchine virtuali devono eseguire a livello user** e ciò, certamente, porta ad un problema di base: quello del *ring deprivileging*, che deve essere sempre considerato per la realizzazione di un VMM di sistema.

Definizione 9: ***Ring Deprivileging***

Situazione in cui il kernel di un sistema operativo che gira su una macchina virtuale si trova ad eseguire in un ring di protezione che non è privilegiato e che, dunque, non è il livello per cui è stato concepito. Infatti, nei kernel dei sistemi operativi, sono presenti istruzioni privilegiate che, di conseguenza, non possono essere eseguite in ring superiori allo 0: dunque, nelle macchine virtuali, queste chiamate non hanno possibilità di esecuzione in modo diretto.

Come si può capire dalla definizione, la conseguenza diretta di questa problematica è che le istruzioni privilegiate richieste dal sistema guest non possono essere eseguite in quanto richiederebbero lo stato di supervisor. Questo effetto va assolutamente evitato poichè il sistema operativo della macchina virtuale deve funzionare correttamente anche in un contesto virtualizzato, tenendo conto che esegue in un ring per il quale non è stato progettato.

Una prima soluzione è la **trap&emulate**: se il kernel della macchina virtuale chiama un'istruzione privilegiata, la CPU reagisce a questa situazione *scorretta* inviando un'eccezione che viene catturata dal Virtual Machine Monitor. La macchina virtuale, allora, viene interrotta e il controllo viene passato a quest'ultimo che, dunque, verifica l'eccezione e, quindi, la correttezza dell'operazione richiesta, emulandola a favore della VM che l'ha generata. Dunque, è il VMM che, catturando l'eccezione, può eseguire le istruzioni privilegiate richieste dai kernel delle macchine virtuali, di fatto emulandole.

Si specifica che, se una macchina virtuale richiede l'esecuzione dell'istruzione privilegiata *popf*, ovvero quella per la disabilitazione delle interruzioni, il VMM deve verificare tale richiesta e sospendere gli interrupt esclusivamente per la VM che l'ha richiesto e non per tutto il sistema host.

Inoltre, il meccanismo *trap&emulate* è possibile se e solo se si dispone di un processore che offre supporto nativo alla virtualizzazione e questa, ovviamente, è una complicazione poiché non tutti i processori ne sono provvisti. Ovviamente, in assenza di tale supporto, non è possibile risolvere le problematiche del *ring deprivileging* e, in questo, le istruzioni privilegiate vengono ignorate o, in alcuni casi, possono provocare crash del sistema.

Definizione 10: ***Architettura nativamente virtualizzabile***

Tipologia di architettura che prevede in modo nativo l'invio di trap per ogni istruzione privilegiata inviata da un livello di protezione che sia diverso dal ring di supervisor. Un'architettura di questo tipo permette l'applicazione del problema del ring deprivileging mediante l'attuazione della trap&emulate: in questo caso, ogni chiamata di istruzioni privilegiate eseguita da una macchina virtuale genera una trap che viene catturata dal Virtual Machine Monitor e trattata di conseguenza.

Il poblema del *ring deprivileging*, però, pur essendo quello più critico, non è l'unica problematica a cui bisogna far fronte.

Definizione 11: ***Ring Compression***

Situazione che si presenta quando i ring utilizzati in un sistema sono solamente due: in quel caso, il sistema operativo della macchina guest deve eseguire allo stesso livello delle relative applicazioni. Questo, ovviamente, causa una scarsa protezione tra lo spazio dell'OS e quello dedicato alle applicazioni.

Definizione 12: ***Ring Aliasing***

Problematica di inconsistenza che si verifica quanto istruzioni non privilegiate, eseguite in modo user, permettono la lettura di alcuni registri la cui gestione dovrebbe essere riservata al Virtual Machine Monitor. Infatti, il kernel di sistema virtualizzato, assumendo che stia eseguendo a livello 0, può tentare, ad esempio, di accedere a registri che sono accessibili anche a livello user ma che dovrebbero essere gestiti esclusivamente dal VMM (come, ad esempio, il registro CS che contiene il livello di provilegio corrente, il CPL).

Chiaramente, c'è una potenziale inconsistenza che causa questo problema in quanto, come già detto, un sistema operativo virtualizzato deve eseguire come se fosse installato in assenza di virtualizzazione. Pertanto, il kernel di questo OS, assume di eseguire a livello 0 ma, in realtà, pur non eseguendo in quel ring, ha la visibilità del registro CS e, dunque, può vedere il livello di privilegio corrente. Ma, ovviamente, per un sistema operativo di una macchina virtuale questo non può essere 0 e, di conseguenza, c'è la potenziale inconsistenza: il valore atteso dal kernel del guest è diverso da quello che preleverà nel registro. Tale problema si verifica nel caso di sistemi che non hanno supporto nativo alla virtualizzazione.

Fast Binary Translation e *Paravirtualizzazione*: virtualizzazione per sistemi che non dispongono di meccanismi nativi

Non tutti i processori, purtroppo, offrono il supporto nativo alla virtualizzazione e, pertanto, in architetture di questo tipo non è possibile sfruttare il meccanismo di *trap&Emulate*. Dunque, bisogna utilizzare dei meccanismi che permettano di risolvere il *ring deprivileging* a livello software. Sono possibili due soluzioni: *Fast Binary Translation* e *Paravirtualizzazione*.

Definizione 13: ***Fast Binary Translation***

Tecnica di virtualizzazione simile alla compilazione dinamica, in cui il Virtual Machine Monitor pre-scansiona il codice che il kernel del sistema virtualizzato sta per eseguire allo scopo di individuare eventuali chiamate ad istruzioni privilegiate. Se nel blocco appena scansionato compaiono istruzioni di questo tipo, allora esso viene tradotto, a runtime, nel linguaggio che il sistema host è in grado di capire in modo da permettere l'esecuzione delle operazioni richieste. Ovviamente, la traduzione dovrà contenere delle istruzioni che, in qualche modo, trasferiscono il controllo al Virtual Machine Monitor il quale, dunque, interpreta la richiesta e la implementa per la macchina virtuale che generata.

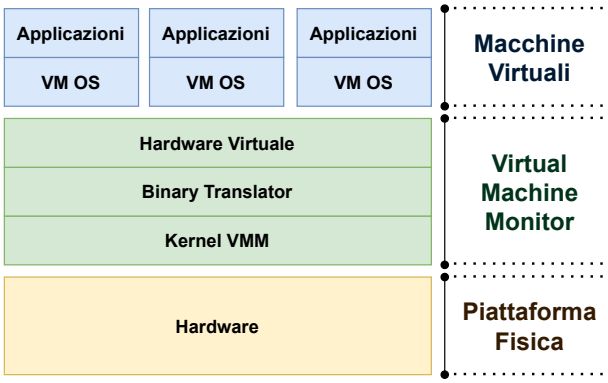


Figura 1.5: Architettura della virtualizzazione con approccio *Fast Binary Translation*

Riassumendo, il VMM, a tempo di esecuzione, interpreta le richieste del kernel della singola macchina virtuale e le traduce, eseguendole opportunamente.

Tale meccanismo, naturalmente, è abbastanza costoso dato che la traduzione è fatta *on-the-fly*, per cui i sistemi che adottano questo approccio devono anche utilizzare delle tecniche che possano diminuire l'impatto sul sistemam tipicamente mediante *cache* in modo da permettere il riutilizzo dei blocchi tradotti per eventuali riusi futuri. Questo approccio, tutt'oggi, è ancora molto utilizzato. La figura 1.5 mostra come è strutturato un sistema che adotta il *Fast Binary Translation*. In particolare, il VMM è suddiviso in tre parti:

- quella più esterna, che espone alle macchine un'interfaccia virtuale che replica in tutto e per tutto quella dell'hardware sottostante;
- una più interna che si occupa della traduzione a tempo di esecuzione;
- il kernel del VMM grazie al quale, una volta eseguita la traduzione dal livello superiore, possono essere processate tali istruzioni.

Dunque, **ogni macchina virtuale è un'esatta replica di quella fisica** e, di conseguenza, è possibile installare in una VM lo stesso sistema operativo che si andrebbe a mettere sul sistema fisico senza virtualizzazione.

Se nella *Fast Binary Translation* il tutto è eseguito in modo dinamico e a tempo di esecuzione, nella *Paravirtualizzazione*, invece, non è così: l'approccio è *compilato*.

Definizione 14: ***Paravirtualizzazione***

Tipologia di virtualizzazione in cui il Virtual Machine Monitor offre alle macchine virtuali un'interfaccia in cui le operazioni privilegiate sono sostituite da funzioni di libreria () che, in realtà, richiamano il VMM stesso. Impropriamente, si potrebbe dire che le hypercall API sono delle system call che, a differenza di quelle normali, richiamano il Virtual Machine Monitor. Le istruzioni non privilegiate, invece, coincidono esattamente con quelle per processore sottostante e possono essere eseguite direttamente dalle macchine virtuali senza l'intervento del monitor.
Nel caso della paravirtualizzazione, il Virtual Machine Monitor prende il nome di ***Hypervisor****.*

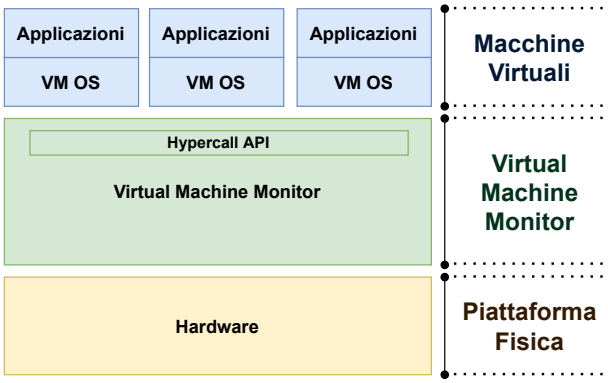


Figura 1.6: Architettura di un sistema paravirtualizzato

Segue, allora, che in tale metodologia di virtualizzazione, il linguaggio *macchina* delle singole macchine virtuali non è quello offerto dal processore: la conseguenza diretta è la **necessità di ricompilare il sistema operativo che deve essere messo nella VM** (*porting*). Infatti, è necessario modificare ad-hoc il kernel del sistema guest in modo tale che il codice delle istruzioni privilegiate richiami le *hypercalls* esibite da un determinato hypervisor.

L'hypervisor, a differenza della *Fast Binary Translation*, non ha bisogno della parte di traduzione delle istruzioni e, dunque, la sua struttura è molto più semplificata in quanto è tutto più diretto. Si potrebbe dire, inoltre, che tale Virtual Machine Monitor è una sorta di emulatore. La semplicità a livello architetturale fa si che le prestazioni della paravirtualizzazione siano migliori rispetto alla *Fast Binary Traslation* ma, di contro, bisogna disporre di sistemi operativi che siano stati esplicitamente adattati a questa architettura virtuale e, quindi, si ha la necessità di *porting* dei sistemi guest. Tuttavia, questo potrebbe essere un problema a livello pratico per molti sistemi operativi proprietari non open source (ad esempio, non esiste una versione di Windows per Xen, un'architettura virtuale che sfrutta la paravirtualizzazione).

Confronto tra le tre soluzioni

Riassumendo, il problema principale per la virtualizzazione dei sistemi operativi è quello del *Ring Deprivileging* e, per risolverlo, sono possibili tre principali soluzioni:

- una soluzione hardware (*Trap&Emulate*);
- due soluzioni software (*Fast Binary Translation* e para-virtualizzazione).

Come già detto, la prima non è applicabile a tutti i sistemi ma solo a quelli che sono nativamente virtualizzabili mentre le seconde, in generale, possono essere adottate da tutte le architetture moderne.

Sperimentalmente, inoltre, i sistemi con supporto nativo vantano prestazioni ottime[5] anche se, in realtà, la soluzione basata su para-virtualizzazione è più performante: infatti, per la *Trap&Emulate*, la richiesta di istruzioni privilegiate solleva eccezioni che, a loro volta, scatenano meccanismi di gestione delle interruzioni che hanno costi complessivamente non trascurabili. Pertanto, la soluzione basata su supporto nativo è sicuramente più semplice e comoda ma, purtroppo, non è la più performante in assoluto; tuttavia, bisogna dire che nei sistemi moderni tutti e tre i metodi elencati funzionano in modo efficiente.

1.2.5 Compiti principali del Virtual Machine Monitor

Il Virtual Machine Monitor, riassumendo, può essere visto come uno speciale sistema operativo che, invece di occuparsi dei tradizionali processi, si trova a dover gestire delle macchine virtuali. Naturalmente, questo componente deve farsi carico di alcuni compiti come la creazione, lo spegnimento, l'accensione, l'eliminazione e la migrazione delle macchine virtuali.

Prima di procedere con un'analisi più approfondita dei compiti principali del VMM, è utile chiarire una macchina virtuale, nel corso della sua esistenza, può trovarsi nei seguenti stati:

- **Running** (attiva)
  la VM è accessa e **occupa della memoria nella RAM** del server sul quale è allocata; essa, dunque, non ha nulla che gli possa impedire l'esecuzione delle proprie applicazioni.

- **Inactive** (spenta)
  la VM esiste ma è inattiva perché è stata spenta; in questo caso essa **non occupa memoria sulla RAM** tuttavia risiede nel file system in un file di immagine che la rappresenta.

- **Paused** (in pausa)
  la VM è in pausa quando **è in attesa di un certo evento**, ovvero quando non può procedere perché è impossibilitata in quanto sta attendendo un evento che gli consentirà la ripresa delle sue attività.

- **Suspended** (sospesa)
  la VM esiste ma è messa in *stand-by* dal Virtual Machine Monitor, ovvero **il suo stato e le risorse allocate a quella macchina sono state congelate** e rappresentate opportunamente in un'immagine salvata sul file system. La macchina rimane, dunque, in attesa e può essere riavviata soltanto per effetto del Virtual Machine Monitor.

[5]Non appena nacquero le prime architetture con supporto nativo, anche Xen, che fino ad allora sfruttava esclusivamente la para-virtualizzazione, fece uscire subito una seconda versione che sfruttasse tale meccanismo e, ad oggi, è ancora presente in entrambe le forme. Anche VMware, nato con l'approccio *Fast Binary Translation*, si è adattato sviluppando una versione per le architetture nativamente virtualizzabili; in ultimo si cita KVM, anch'esso basato su *Trap&Emulate*, la cui principale caratteristica è che integra un modulo di virtualizzazione direttamente nel kernel in modo da poter sfruttare alcuni meccanismi tradizionali del sistema operativo host.

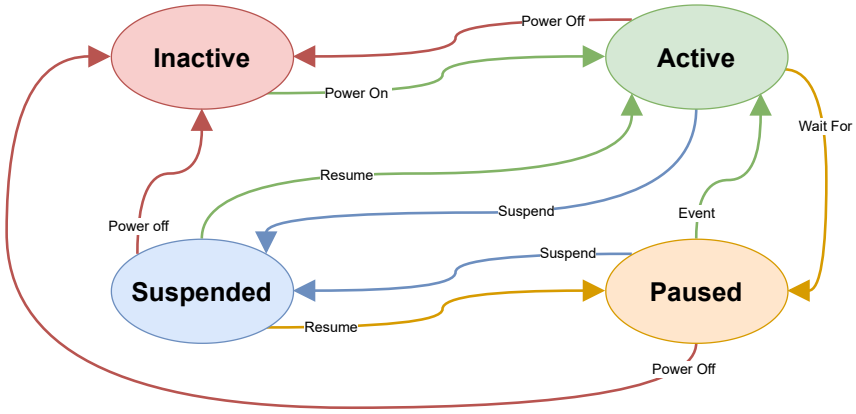


Figura 1.7: Grafo degli stati di una macchina virtuale

Tipicamente, una macchina virtuale viene creata e, dunque, è inizialmente nello stato di inattività ovvero esiste ed è spenta; per renderla operativa, dunque, il Virtual Machine Monitor deve impartire il comando *Power On* per accenderla e farla passare nello stato attivo. Durante la sua attività, può accadere che essa compia un'operazione che la costringa ad attendere un evento e che, dunque, la passare allo stato di pausa: essa, dunque, tornerà attiva quando l'evento che attende si sarà verificato[6]. Tuttavia, mentre la macchina è in attività o in pausa, è possibile che un comando, *suspend*, costringa la macchina ad andare in *stand-by*: da questo stato la VM può uscire o grazie ad una *resume* che la riporta nello stato dov'era prima che fosse sospesa, oppure può essere spenta con una opportuna direttiva, *power off*. Dal momento dello spegnimento, la macchina è rappresentata nello stato di inattività in un'immagine nel file system.

Migrazione delle macchine virtuali

In situazioni in cui bisogna gestire un gran numero di macchine virtuali, facendo uso di un insieme composto da un certo numero di macchine fisiche, è molto sentita la necessità di poter gestire le VM nel modo più agile possibile. L'agilità, in particolare, si traduce con la necessità di spostare una macchina da un nodo all'altro, nel modo migliore possibile. Si parla, allora, di **migrazione** delle macchine virtuali.

La migrazione offre numerosi vantaggi ed è largamente utilizzata per far fronte a:

- situazioni in cui è necessario bilanciare l'insieme delle macchine fisiche che si stanno utilizzando in modo tale da avere la configurazione più efficiente possibile (*load balancing*);
- situazioni in cui è necessario intervenire a livello hardware su una o più macchine fisiche;
- gestione ottimale del risparmio energetico, che si traduce nel consolidamento delle macchine nel minor numero possibile di nodi;
- tolleranza a guasti e disaster recovery.

[6]Si specifica che negli stati *active* e *paused*, la macchina virtuale è accesa e la sua situazione è tracciata da uno stato, occupando, dunque, dello spazio nella memoria centrale.

Definizione 15: ***Migrazione Live***

Operazione di spostamento di una macchina da un nodo fisico ad un altro, senza necessariamente spegnerla. La modalità per effettuare questo tipo di migrazione consiste nel sospendere la macchina, trasferirla insieme con il suo stato e, sul nuovo nodo, farla ripartire grazie ad un'operazione di resume.

Tipicamente, questa operazione non comporta problemi sostanziali in quanto le macchine virtuali sono progettate per essere il più possibile indipendenti dall'architettura fisica e, dunque, è sufficiente trasferirla su un'altra macchina che abbia lo stesso Virtual Machine Monitor.

Generalmente, queste operazioni sono frequenti nei datacenter dei server delle aziende e i parametri che si vogliono minimizzare sono:

- **Downtime**
  noindent tempo in cui la macchina non reagisce alle sollecitazioni esterne (se la macchina è un server, allora è il tempo in cui essa non reagisce alle richieste dei clienti).
- **Tempo di migrazione**
  noindent tempo impiegato per l'effettivo spostamento.
- **Consumo di banda**
  noindent la rete che si utilizza per eseguire la migrazione, tipicamente, è impiegata per altre attività per cui è necessario non consumare troppa banda.

Si specifica che **se il file system è condiviso non c'è bisogno di copiare l'immagine** in quanto essa è vista sia dal nodo di partenza, sia da quello di destinazione. In questo caso, allora, il problema è **trasferire i dati che sono in memoria centrale e lo stato**. Allora, per rendere il più efficiente possibile la migrazione, si utilizza un meccanismo di **copia incrementale** o *Pre-copy*:

0. individuata la necessità di copiare una macchina, si sceglie l'host di destinazione (pre-migrazione);
1. nel nodo di destinazione viene riservato un posto per quella macchina virtuale nel senso che viene creato un contenitore vuoto che avrà come scopo quello di accogliere la macchina che verrà migrata (prenotazione o *reservation*);
2. vengono copiate le pagine allocate in memoria centrale dall'host di partenza a quello di destinazione (pre-copia); la macchine virtuale sul nodo originale continua a girare e, pertanto viene eseguita una copia iterativa anche delle *dirty pages*[7], ovvero quelle il cui contenuto è stato modificato rispetto alla copia originale;
3. quando il numero delle *dirty pages* sono sotto una certa soglia, la macchina sul nodo di partenza viene sospesa, viene trasferito il contenuto delle ultime pagine e lo stato della macchina sul nuovo host; si può assumere che tali informazioni abbiano dimensioni contenute per cui il tempo impiegato per quest'operazione è trascurabile, poco significativo;

[7]Queste pagine, probabilmente, vengono copiate secondo un certo criterio non rilevante ai fini dello studio.

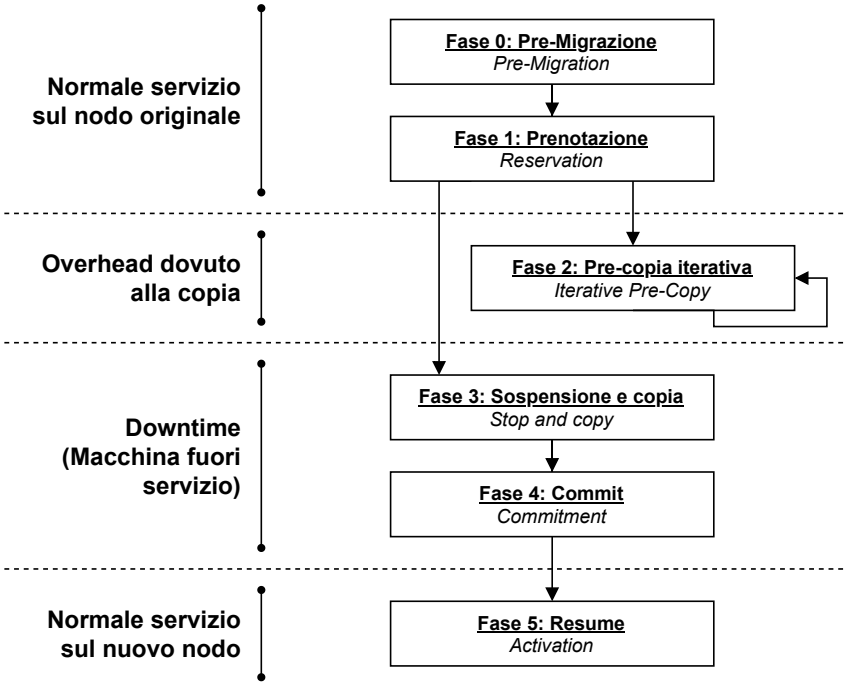


Figura 1.8: Fasi della migrazione live di una macchina

4. ora che tutto è stato trasferito sul nuovo nodo, si elimina la macchina sul vecchio host (commit);

5. dunque, si esegue l'attivazione sul nuovo host per cui la macchina può riprendere esattamente nello stesso punto in cui era stata interrotta (resume).

Ovviamente, il tempo che intercorre dall'inizio della fase 3 alla fine della 5, è quello in cui, effettivamente, il servizio reso disponibile dalla macchina virtuale non è disponibile (*downtime*).

Oltre alla Pre-copy, esiste anche un secondo metodo detto **Post-copy** e, sostanzialmente, consiste nel sospendere la macchina per poi copiare completamente le pagine nella memoria centrale e lo stato e trasferirle sul nuovo sistema fisico.

Per quanto riguarda la Pre-copy, il downtime è trascurabile (nell'ordine dei millisecondi) e, dunque, è decisamente inferiore a quello del Post-copy che, invece, può richiedere anche minuti di inattività seppur con il vantaggio di ridurre, almeno lievemente, il tempo di migrazione. L'unico prezzo per l'abbattimento del *downtime* è un piccolo abbassamento della banda disponibile ai clienti, dovuto al meccanismo di pre-copia.

1.2.6 Xen

Generalità su Xen

Xen[8] nasce come un progetto universitario e open-source. Inizialmente, fu una variante di Linux con lo scopo di supportare l'esecuzione di macchine virtuali approcciando il problema del ring deprivileging con la paravirtualizzazione.

[8]In questa sezione verrà considerata la versione con paravirtualizzazione.

Per questo motivo, dopo la sua nascita di, tantissime versioni di Linux si adattarono per poter essere eseguibili su Xen, ovvero, all'interno dei loro kernel, vennero sostituite le chiamate ad istruzioni privilegiate con porzioni di codice contenenti le *hypercall* di Xen[9].

Nonostante sia nato come prodotto open-source, nel 2007 Xen è diventato anche un prodotto commerciale dopo essere stato acquisito da *Citrix*, un'azienda che fornisce tecnologie di virtualizzazione desktop e server.

Xen è un **hypervisor che si colloca a livello di sistema** e, dunque, è installato direttamente sopra l'hardware per cui, sopra di lui, possono essere eseguite un numero arbitrario di macchine virtuali (*domain*). La sua architettura prevede l'esistenza di una macchina virtuale privilegiata detta *domain 0* sulla quale gira la console di gestione e amministrazione dell'intero sistema[10] virtualizzato (interazione *low-level)*; le altre macchine, invece, sono dette *domain U* (dominio user).

Virtualizzazione dell'I/O

Una grande peculiarità di Xen è che le macchine virtuali possono essere del tutto svincolate dall'hardware sottostante poiché i device driver dei dispositivi fisici disponibili a livello hardware sono tutti collocati e condensati nel domain 0[11] in questo modo anche l'hypervison è molto più leggero e può occuparsi esclusivamente della gestione della CPU, della memoria e dei dispositivi per ogni VM. Per accedere ad un dispositivo, pertanto, ogni macchina virtuale dispone di un device driver virtuale (*front-end device driver*) che, all'occorrenza, comunica con il device driver del domain 0 (*back-end device driver*) che realizza le operazioni richieste. Questo significa che ogni macchina virtuale che gira su Xen è fortemente indipendente dall'hardware sottostante, semplificando notevolmente le operazioni di trasferimento che vengono effettivamente svolte dal domain 0 e non dall'hypervisor, il quale, dunque, è sollevato dalla gestione a basso livello dell'I/O.

Ogni sistema operativo guest, allora, non contiene il device driver del dispositivo, tuttavia ne vede uno virtuale che è rappresentato dal *front-end*.

Questa soluzione, ovviamente, ha come grande vantaggio la portabilità e la semplificazione del Virtual Machine Monitor. Tuttavia, se una macchina virtuale ha bisogno di utilizzare un dispositivo, questa ha bisogno di comunicare la sua richiesta al domain che detiene il device driver di quest'ultimo: questo, ovviamente, viene fatto utilizzando il modello a scambio di messaggi e viene attuato con un meccanismo di coda circolare (*I/O Rings*) che segue la logica FIFO, in cui sono presenti:

- una coda delle richieste per ogni dispositivo;
- una coda delle risposte in cui Xen, una volta completato l'accesso al dispositivo, andrà a scrivere l'esito del trasferimento.

Ogni elemento di questa struttura, allora, rappresenta il *descrittore* di una richiesta. Questo meccanismo, inoltre, è sufficientemente efficiente in termini di prestazioni: grazie alla gestione asincrona delle richieste, il costo di questa coda è abbastanza accettabile.

[9] Si specifica che l'operazione di porting, in realtà, non è così invasiva poiché, mediamente, le percentuale di codice su cui dover apportare modifiche è circa del 2%; tuttavia, non tutti i sistemi operativi sono stati modificati per essere compatibili con Xen (ad esempio Windows).

[10] In questo modo si separano i meccanismi dalle politiche: l'hypervisor fa quello che il domain 0 decide.

[11] In realtà, non è obbligatorio che siano installati necessariamente nel domain 0: è possibile anche affidare ad un'altra macchina virtuale speciale, o a più di una, i driver nativi delle periferiche. Tuttavia, i domain U non vedono comunque i driver nativi ma una loro intefaccia virtuale.

Soluzione al ring compression

Per quanto riguarda il problema del ring compression, invece, tipicamente nelle architetture Intel i bit a disposizione per il livello di privilegio sono 2, motivo per cui Xen colloca l'hypervisor a livello 0 e separa il sistema operativo della VM dalle applicazioni collocando il primo a livello 1 e le seconde al ring 3 (scelta 013) in modo da garantire un discreto livello di protezione per quanto riguarda le aree di memoria su cui insistono le applicazioni e il sistema operativo.

Gestione della memoria e paginazione

Come già detto, l'hypervisor deve occuparsi della gestione dei dispositivi e, dunque, anche e soprattutto della memoria che deve essere amministrata unicamente da lui: una macchina virtuale, allora, deve chiedere al Virtual Machine Monitor per poter allocare spazio in memoria. Nonostante questo, la struttura di un sistema operativo guest è la stessa di quella che si avrebbe in assenza di virtualizzazione, per cui esso deve occuparsi dei classici compiti dei SO (oltre a quelli di base che, come già detto, sono demandati al VMM), come, ad esempio, la gestione della memoria virtuale, componente fondamentale di tutti i sistemi operativi moderni[12]. Ovviamente, da questo segue che le politiche di gestione della memoria virtuale rimangono a carico del sistema operativo, insieme con la gestione del *page fault*. Tuttavia, di fatto, tale gestione può portare a dover compiere operazioni di basso livello (come allocare una nuova pagina o andare ad aggiornare il contenuto di un elemento della tabella delle pagine[13]): dunque, i meccanismi sono demandati al Virtual Machine Monitor[14] mentre le politiche, invece, vengono lasciate al sistema operativo guest.

Riassumendo, il sistema guest, ad ogni istante, deve gestire un insieme di processi e ognuno di essi insiste su una certa tabella delle pagine ognuna delle quali trova una corrispondente allocazione in memoria centrale realizzata dal Virtual Machine Monitor. Una volta create, le tabelle possono essere direttamente lette dal guest ma, per quanto riguarda la scrittura, c'è bisogno dell'intervento dell'hypervisor.

Per quanto riguarda la gestione dello spazio di indirizzamento virtuale, invece, Xen adotta la tecnica del *memory split*: il *virtual address space* deve necessariamente contenere anche le pagine dove è allocato Xen in modo che sia possibile interagire con il Virtual Machine Monitor in ogni momento; dunque, Xen risiede sempre nei primi 64MB del virtual address space. Questo evita la necessità di effettuare delle *flush* dopo l'invocazione delle hypercall, volte a ricaricare le pagine del VMM necessarie alla gestione della chiamata; infatti, se tutti gli spazi di indirizzamento contengono già Xen, non è necessario commutare quello del kernel della macchina virtuale con quello dell'hypervisor in quanto quest'ultimo è già presente. L'adozione di tale meccanismo, pertanto, rende possibile una

[12]Si ricorda che se in un sistema c'è memoria virtuale, allora le pagine possono esserci o meno in memoria centrale e, nel caso in cui una pagina non sia presente, tale situazione viene notificata mediante un *page fault*; la gestione di quest'ultimo, tuttavia, dipende da chi ha progettato il sistema operativo.

[13]Si specifica che le tabelle delle pagine, chiaramente, devono essere mappate nella memoria fisica del Virtual Machine Monitor. La rappresentazione in memoria di queste tabelle, in un sistema virtualizzato, viene detta *Shadow Page Table*.

[14]Quando una macchina virtuale lo richiede, il Virtual Machine Monitor deve andare a scrivere per suo conto in una cella di memoria. Dunque, se c'è bisogno di andare ad aggiornare la tabella delle pagine, ad esempio per la gestione di un *page fault*, il singolo kernel della VM deve andare a chiedere al VMM affinché lo faccia per suo conto; diverso è per la lettura: se un sistema guest deve leggere dalla tabella delle pagine, può farlo direttamente.

maggiore efficienza ma anche maggior protezione: infatti, ogni spazio di indirizzamento è organizzato come segue.

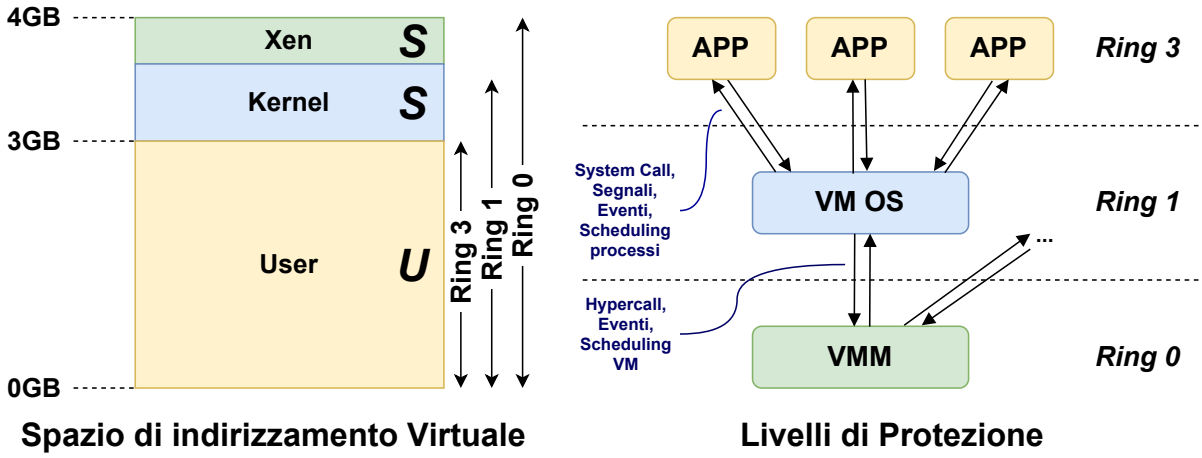


Figura 1.9: Suddivisione dello spazio di indirizzamento e livelli di protezione in Xen

Dunque, per creare un processo, come prima cosa, un sistema operativo guest deve richiedere al Virtual Machine Monitor una nuova tabella delle pagine. L'hypervisor, dunque, crea la tabella dedicata al processo alla quale, però, vengono aggiunte anche le pagine appartenenti al segmento di Xen (Memory Split): Xen, allora, terrà traccia della tabella su cui avrà diritto esclusivo di scrittura mentre il guest avrà solamente quello di lettura. Se il guest tentasse di scrivere sulla tabella delle pagine, allora, genererà un *protection fault* che sarà intercettato dal Virtual Machine Monitor, il quale andrà a fare l'operazione di scrittura per conto del sistema operativo che l'ha richiesta.

La figura 1.10 mostra il meccanismo del *protection fault*.

Un'altra cosa interessante da specificare di Xen a proposito della gestione della memoria virtuale è il *balloon process*.

Anzitutto, si ricorda che non è detto che una macchina virtuale esegua esattamente allo stesso modo e con le stesse condizione che avrebbe in un sistema non virtualizzato[15]. Inoltre, sulla base di quanto appena illustrato, si capisce che la gestione della memoria virtuale è a carico del sistema guest: non è il Virtual Machine Monitor che, di sua iniziativa, crea una pagina bensì è il sistema virtualizzato che ne richiede la creazione. Questo, però, può essere un problema poiché potrebbe accadere che l'hypervisor sia in carenza di memoria e abbia la necessità di allocare nuove pagine ma, a seguito del discorso appena fatto, è chiaro che questo non può reclamare pagine perché questa operazione è possibile solo dai sistemi operativi delle macchine guest. Per far fronte a questa problematica, in Xen, si è deciso di equipaggiare ogni macchina virtuale con un processo che rimane sempre in esecuzione, il *balloon process*, e dialoga costantemente e direttamente con il Virtual Machine Monitor[16]. Tale meccanismo permette al VMM, in caso di necessità, di contattare tutti i ballon process delle macchine virtuali chiedendogli pagine; a loro

[15]Questo, però, è vero nel caso di virtualizzazione pura e *Fast Binary Translation* che, invece, offrono alle macchine virtuali l'illusione di girare in un'architettura in tutto e per tutto coincidente come caratteristiche con quella reale.

[16]Si potrebbe dire, allora, che tale processo è cosciente di essere in un ambiente virtualizzato data la sua comunicazione diretta con il Virtual Machine Monitor. Questo ovviamente, non accade e non può esistere in un ambiente a virtualizzazione pura.

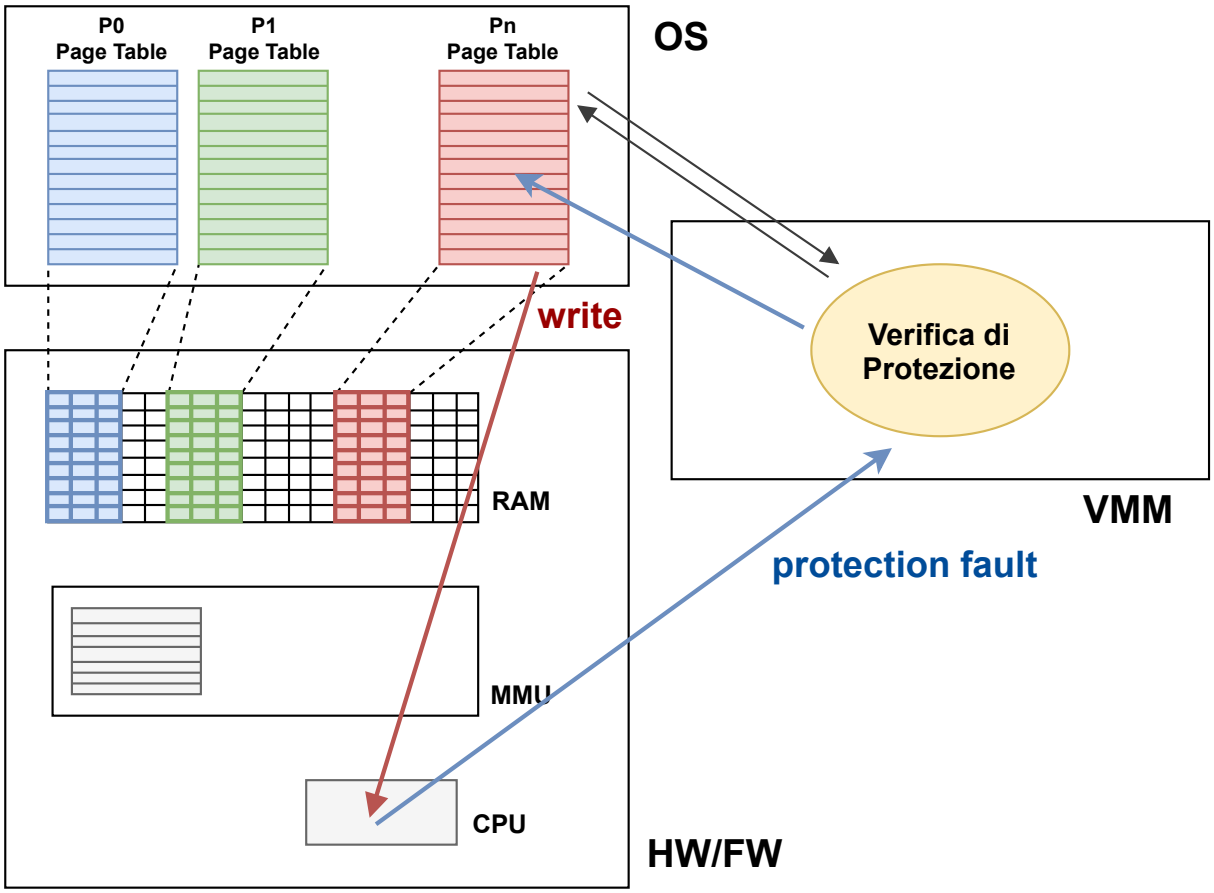


Figura 1.10: Tentativo di scrittura sulla tabella delle pagine da parte di una VM

volta, questi processi, richiedono al proprio sistema operativo di liberare delle pagine che vengono cedute all'hypervisor.

Definizione 16: ***Balloon Process***

Processo tipico di Xen che viene equipaggiato a tutte le macchine virtuali rimanendo sempre in esecuzione su ognuna di esse e che permette al Virtual Machine Monitor una comunicazione diretta che viene sfruttata per eseguire meccanismi di liberazione della memoria. Nello specifico, l'hypervisor, in caso di carenza di memoria, può richiedere pagine a tutti i Balloon Process delle macchine virtuali che, a loro volta, chiedono al sistema operativo di liberarne alcune, in modo tale che possano essere cedute al Virtual Machine Monitor.

Virtualizzazione della CPU

Lo scheduler, ovviamente, è un componente fondamentale di ogni Virtual Machine Monitor. Rispetto ai suoi competitor, Xen offre uno scheduler molto particolare che si basa su un algoritmo che si chiama *Borrowed Virtual Time*.

Innanzitutto, si specifica che in un'architettura virtuale si hanno due tipi di tempi:

- il *tempo reale*, ovvero quello avanza allo scoccare di ogni tick del clock;

- il *tempo virtuale*, ovvero quello della singola macchina virtuale che avanza solamente quando la VM procede con la sua esecuzione.

Chiaramente, i tempi vengono comunicati al guest tramite eventi che vengono gestiti e concorrono a tenere aggiornato il virtual time, tempo su cui si basa l'algoritmo di scheduling di Xen.

Definizione 17: ***Borrowed Virtual Time Scheduling Algorith***

Algoritmo general-purpose per lo scheduling adottato da Xen, basato sulla nozione di tempo virtuale e che consente di tenere conto dei vincoli temporali delle macchine virtuali.

Gestione delle interruzioni

Come i componenti precedenti, anche la gestione delle interruzioni ha bisogno di essere virtualizzata. Innanzitutto, nel momento in cui si verifica un'interruzione, il VMM deve identificare a quale macchina virtuale è interessata.

La successiva gestione, quindi, è molto simile a quella classica: è presente un vettore delle interruzioni che punta direttamente alle routine del kernel guest, il quale gestisce in modo diretto la propria interruzione.

Tuttavia, questo schema non può essere applicato a tutti i casi come, ad esempio, per il page-fault. Si ricorda che il page-fault è un'interruzione che si verifica nel momento in cui un processo, durante la sua esecuzione, ha bisogno di caricare in memoria una pagina che non è presente nell'istante in cui la richiede. In questo caso, allora, è presente un registro particolare, che si chiama *CR2*, in cui viene memorizzato l'indirizzo che provocato il page-fault e che, però, è accessibile solamente nel ring 0. Segue, allora, che il guest non può eseguire in autonomia la routine di gestione dell'interruzione poiché esso esegue nel livello 1: deve essere chiamato in causa l'hypervisor di Xen e, per questo motivo, in questa situazione specifica, nel vettore delle interruzioni si ha un indirizzo che non punta allo spazio di indirizzamento del guest che ha provocato il page-fault, bensì punta ad un indirizzo che fa parte dello spazio del VMM in cui ci sarà la routine da eseguire che, a sua volta, non farà altro che accedere all'informazione contenuta in CR2 in modo da copiarla in una locazione che sia visibile al sistema guest. A questo punto, allora, sarà eseguito un *jump* (trasferimento di controllo) dall'hypervisor al guest che può procedere con la gestione del page-fault.

Live Migration

Tale meccanismo è presente in quasi tutti i prodotti di virtualizzazione, Xen compreso. La migrazione, in Xen, avviene grazie all'intervento di una sorta di demone che esegue all'interno del domain 0 e che, all'occorrenza, può essere attivato per mettere in atto quei meccanismi che realizzano la migrazione di una macchina da un nodo all'altro. A livello pratico, quel che accade è che viene impartito al domain 0 il comando di migrazione di una macchina che esegue sul nodo che si sta considerando, per spostarla ad un'altro nodo; a seguito di questo comando, allora, viene realizzata la migrazione che è basata su pre-copy: l'unica variante, però, è l'attuazione di meccanismi di compressione sulle pagine da migrare in modo da ridurre l'occupazione di banda e rendere più veloce il trasferimento.

Capitolo 2

Protezione

2.1 Generalità sulla protezione

Si inizia la trattazione sulla protezione facendo una prima differenza tra **protezione** e **sicurezza**. In generale, lo scopo di entrambe è **proteggere un sistema da un uso non corretto e non autorizzato**.

Definizione 18: ***Sicurezza***

Insieme di tecniche utilizzate per regolamentare l'accesso degli utenti ad un sistema: dall'esterno sino all'utilizzo interno. L'obbiettivo principale della sicurezza è ***impedire che utenti non autorizzati accedano al sistema*** *evitando, dunque, che essi possano compiere su di esso operazioni dannose sia inconsapevolmente sia in modo doloso. Tali azioni, infatti, potrebbero alterare il funzionamento del sistema, sottrarre dei dati o, peggio ancora, alterarli.*

Dunque, il compito principale della sicurezza è quello di controllare l'accesso degli utenti all'intero sistema, mettendo in campo dei meccanismi che hanno come scopo:

- **l'identificazione** degli utenti che hanno accesso al sistema, ovvero capire chi sia quest'ultimo e se sia realmente conosciuto;
- **l'autenticazione**, perché l'utente può dichiararsi con un certo nome ma bisogna verificare che l'identità della persona che si sta dichiarando con quel nome corrisponda effettivamente all'utente che il sistema conosce;
- **l'autorizzazione**, dato che non è detto che tutti gli utenti possano fare le stesse cose.

Definizione 19: ***Protezione***

Insieme di tecniche impiegate per proteggere il sistema da operazioni non autorizzate ma, a differenza della sicurezza, la protezione si riferisce a tutte quelle attività che hanno come obiettivo quello di ***garantire un certo tipo di controllo sull'accesso alle risorse*** *(logiche e fisiche) che il sistema offre ai propri utenti. Tali utenti, ovviamente, sono quelli che sono autorizzati dal sistema di sicurezza all'utilizzo delle varie applicazioni.*

Pertanto, lo scopo di sicurezza e protezione è praticamente lo stesso, ovvero proteggere il sistema da attività non previste e non autorizzate. Tuttavia, dopo aver passato la barriera della sicurezza, è necessario capire a quali risorse un utente può accedere ma anche quali siano le operazioni con cui può farlo. Tale meccanismo è stabilito dal sistema di protezione ed è applicato grazie a tecniche di **controllo degli accessi**[1].

2.2 Caratterizzazione di un sistema di protezione

In linea generale, un sistema di protezione è caratterizzato da tre livelli concettuali:

- **Modelli**

  Definizione 20: ***Modello di protezione***

  Un modello di protezione stabilisce le entità fondamentali si cui si basa un sistema di protezione, ovvero stabilisce quali sono gli oggetti, i soggetti e i diritti di accesso contemplati dallo stesso sistema di protezione.

  In un sistema di protezione gli **oggetti** sono le entità passive alle quali, in generale, è possibile accedere (le risorse fisiche e logiche); i **soggetti**, invece, rappresentano quelle entità che sono potenzialmente in grado di accedere alle risorse (utenti o anche processi[2]) e, dunque, agli oggetti; infine, ci sono i **diritti di accesso**, ovvero le operazioni contemplate dal sistema attraverso le quali è possibile accedere agli oggetti. Ovviamente, l'insieme dei diritti di accesso non è universale: in Unix sono tre (lettura, scrittura ed esecuzione) ma ad, esempio, in Windows ce ne sono di più (come la cancellazione).

  In generale, in molti sistemi, possono essere considerati oggetti anche i soggetti stessi, nel senso che un soggetto può godere di diritti di accesso su un altro soggetto che, dunque, in questo caso viene trattato come un entità passiva.

  È importante dare anche un'ulteriore definizione, quella di dominio di protezione, che è unico per ogni soggetto:

  Definizione 21: ***Dominio di protezione***

  Ambiente di protezione all'interno del quale un soggetto, o i processi che eseguono per suo conto, esegue. Sostanzialmente, esso è un'astrazione che consente di specificare i diritti di accesso dell'utente al quale il dominio stesso si riferisce, rispetto alle risorse che sono a disposizione: segue, allora, che l'associazione dominio-soggetto è univoca.

  Normalmente, in un sistema multi-programmato, sono i processi ad eseguire gli accessi alle risorse per conto dell'utente e, dunque, segue che ad ogni processo, in ogni istante della sua vita, corrisponde un soggetto e quindi un particolare dominio di protezione. La cosa importante, però, è che un processo, a tempo di esecuzione, ha la possibilità di cambiare dominio.

[1] Un esempio di queste tecniche sono i bit di protezione di Unix che, pur in forma semplificata, stabiliscono cosa può fare o cosa non può fare un utente.

[2] Generalmente, però, ogni processo è associato ad un utente; ad esempio, in Linux, ogni processo dispone di un *user id* (UID) che rappresenta l'utente a cui esso appartiene.

- **Politiche**

  Definizione 22: ***Politiche di protezione***

  Una volta noti quali sono gli oggetti a disposizione, quali sono i soggetti e quali sono le operazioni applicabili agli oggetti, le politiche di protezione fissano in modo puntuale un insieme di regole con le quali i soggetti possono accedere agli oggetti.

  Tali regole, allora, ad esempio stabiliscono che un soggetto possa leggere ma non scrivere un oggetto, eseguirne un altro, e così via...

  Da un punto di vista generale, è possibile introdurre una classificazione delle politiche:

  – **Discretional Access Control (DAC)**, in cui la definizione delle regole è *decentralizzata*, ovvero è il proprietario di un oggetto che ha il potere di stabilire cosa possono fare o non fare i soggetti previsti nel sistema su quel particolare oggetto[3]; la decisione, dunque, non è concentrata su un singolo soggetto bensì è distribuita tra tutti i vari utenti;

  – **Mandatory Access Control (MAC)**, che, contrariamente al punto precedente, stabilisce che i diritti di accesso vengano stabiliti in modo centralizzato per cui c'è un'unica entità centrare che, per ogni oggetto, stabilisce in modo autoritario quali soggetti possano accedervi e in che modo[4];

  – **Role Based Access Control (RBAC)**, che è una categoria intermedia tra le precedenti e che prevede che sia definito un nuovo concetto, quello di *ruolo*; ad ogni ruolo, dunque, in modo centralizzato (MAC), vengono assegnati specifici diritti di accesso alle risorse; gli utenti possono appartenere a ruoli diversi, a seconda di quello che vogliono fare.

  Definizione 23: ***Principio del privilegio minimo***

  Principio secondo cui ad ogni soggetto sono attribuiti solo i diritti di accesso strettamente necessari a quello che il soggetto deve fare. Questo vuol dire che ***ogni dominio deve essere opportunamente ridimensionato****.*

  Dunque, secondo questo principio, sapendo a priori quali sono le operazioni e le attività che un soggetto potrà compiere, allora, nel dominio ad esso associato si andrà a prevedere solo ed esclusivamente i diritti che sono strettamente necessari a quell'utilizzo. Questo principio è molto importante e consente di evitare sprechi di risorse e di limitare in modo efficace anche errori dovuti a meccanismi scorretti.

- **Meccanismi**

[3] In Unix, ad esempio, questo avviene grazie alla specifica dei bit di protezione.

[4] Questa tipologia di politiche sono per sistemi che hanno un elevatissimo grado di sicurezza come, ad esempio, quelli degli enti governativi.

Definizione 24: ***Meccanismi di protezione***

Strumenti messi a disposizione dal sistema operativo che consentono di mettere in pratica le politiche di protezione.

I principi fondamentali per realizzare i meccanismi sono:

- **flessibilità**, perché i meccanismi devono essere sufficientemente generali al fine di consentire l'applicazione di diverse politiche di protezione;
- **serapazione** netta tra meccanismi e politiche, poiché la politica definisce cosa deve essere fatto e il meccanismo come deve essere fatto; se le due cose sono separate, si ottiene un sistema di protezione che sia il più possibile flessibile.

Esempio: ***Protezione in Unix***

- ***Politica***
  politica di tipo DAC: sono gli utenti che, in quanto proprietari dei file, possono stabilire il valore dei bit di protezione dei file di loro proprietà.
- ***Meccanismo***
  il meccanismo è basato sui bit di protezione contenuti nel descrittore dei vari file che consentono di rappresentare la politica valida per quel particolare file. Il meccanismo, però, riguarda anche l'interpretazione a runtime.

2.3 Dominio di protezione

Definizione 25: ***Dominio di protezione (definizione rigorosa)***

Si consideri un sistema e siano S un suo soggetto, con O un suo oggetto e con D un'insieme di operazioni che è possibile compiere. Allora, si definisce il dominio di protezione $D(S)$:

$$D(S) = < O, D >$$

Dunque, il dominio associato al soggetto S non è altro che un insieme di coppie formate da un oggetto e un'insieme di operazioni che tale soggetto può compiere sul relativo oggetto.

In linea di principio, allora, non è detto che la cardinalità di un insieme $D(S)$ coincida con il numero di oggetti effettivamente presenti nel sistema poiché le coppie oggetto-diritto compaiono solamente se il soggetto ha un qualche diritto sull'oggetto considerato. È molto frequente che un utente non abbia accesso a delle risorse per cui, per quanto appena detto, è normale che in questo caso l'oggetto non compaia nel $D(S)$ del soggetto.

In generale, in un sistema in cui sono presenti n utenti, si hanno n domini differenti, $D_1, D_2, \ldots, D_n$ e, ovviamente, chi effettivamente compie le operazioni per conto dei soggetti sono i processi che, però, in ogni istante possono eseguire in uno ed un solo dominio.

Si specifica che domini che non contengono diritti di accesso in comune sono detto **domini disgiunti**.

In merito all'associazione tra processo e dominio, si distingue:

- **associazione statica**, ovvero l'insieme degli elementi che costituiscono il dominio di un processo rimane costante per tutto il suo tempo di vita; segue che, per tutta la sua vita, il processo esegue in un unico dominio di protezione;
- **associazione dinamica**, ovvero durante l'esistenza di un processo, l'associazione processo-dominio può variare; dunque, un processo nasce in un dominio ma durante la sua esecuzione può passare in altri domini.

In un sistema multi-programmato, però, la prerogativa principale è quella del *non determinismo*, ovvero, in generale, non si può prevedere a priori cosa farà un processo durante la sua esecuzione: di conseguenza, non è possibile dire quale sarà l'insieme complessivo delle risorse che saranno accedute da quel processo in quanto anche questa informazione è soggetta al non determinismo. Dunque, questo significa che **l'insieme minimo delle risorse necessarie ad un processo non è noto a priori** bensì cambia dinamicamente in funzione dell'evoluzione della sua esecuzione, per cui, si conclude che l'associazione statica non è la soluzione da adottare per applicare il principio del privilegio minimo.

Infatti, non sapendo a priori di cosa avrà bisogno il processo, se l'associazione è statica non c'è altra scelta che andare a mettere nell'unico dominio nel quale esso svolgerà la sua attività, tutte le risorse che potrebbe utilizzare, ma di cui non c'è certezza dell'effettivo utilizzo andando, quindi, contro il principio del privilegio minimo.

Per quanto riguarda l'associazione dinamica, invece, dando al processo la possibilità di cambiare dinamicamente il dominio di protezione, è possibile concepire ogni singolo dominio in modo minimale sapendo le attività che tipicamente dovranno svolgersi in ognuno di essi e andando, quindi, a mettervi solo ed esclusivamente le risorse necessarie a quelle attività. In seguito, poiché il processo, in modo non deterministico, andrà a scegliere di compiere attività diverse che richiedono risorse di domini diversi, grazie al meccanismo di transizione dinamica, se ha bisogno di nuove risorse che non sono già presenti in quello in cui sta eseguendo, può passare in un altro dominio dove, invece, queste sono presenti.

Dunque, **occorre un meccanismo di cambio del dominio**.

Esempio 1: ***Cambio di dominio per sistemi standard dual mode***

Nei sistemi operativi a standard dual mode, ovvero quelli che posseggono due domini (o ring) di protezione, quello kernel e quello utente, se un processo che esegue in modo utente deve eseguire un'istruzione privilegiata, allora esso deve cambiare dominio protezione e può farlo tramite il meccanismo a ***System Call****. Questo tipo di sistema, però, non offre protezione tra diversi utenti del sistema ma solamente tra kernel e utente.*

Esempio: ***Cambio di dominio in Unix***

In Unix, per eseguire il cambio di dominio, è sfruttato il meccanismo di ***set-uid*** *e* ***set-groupid****, previsto per ogni risorsa del sistema: ogni file, allora, possiede un proprietario, che viene riferito dallo user-id, e un bit di dominio, il set-uid.*

Si consideri il caso in cui un utente metta in esecuzione un processo che, ad un certo punto, sostituisca il codice del programma con una **exec**, *la system call fornita da Unix per questo genere di operazione:*

- *l'utente manda in esecuzione un programma che, dunque, genera un processo a cui viene assegnato di dafult lo user-id di tale utente;*
- *ad un certo punto, viene chiamata una exec che, dunque, cambia a runtime il programma eseguito sostituendo il codice attualmente in esecuzione dal processo con quello contenuto in un nuovo file eseguibile;*
- *il file che contiene il nuovo codice, ovviamente, ha un suo proprietario che può essere diverso dall'utente che ha originariamente lanciato il programma che si stava eseguendo prima della exec;*
- *se il proprietario del nuovo eseguibile aveva impostato il bit set-uid (set user id) del file ad 1, allora il processo, a seguito della exec, salta nel dominio dell'utente proprietario;*
- *dunque, al processo viene assegnato un nuovo user-id, quello del proprietario.*

Essendo saltato nel nuovo dominio di protezione, il processo perde tutti i diritti che aveva quando eseguiva nel dominio dell'utente originale ma, di contro, acquisisce tutti quelli previsti in quello del proprietario dell'eseguibile.

2.4 Matrice degli accessi

In generale, un sistema di protezione può essere rappresentato a livello astratto mediante un formalismo che viene detto **matrice degli accessi**.

Definizione 26: ***Matrice degli accessi***

Struttura astratta che permette di rappresentare in modo completo lo stato di protezione di un sistema.
Essa è una tabella con un certo numero di colonne, tante quante sono gli oggetti nel sistema, e un certo numero di righe, tante quanti sono i soggetti. Ogni colonna, infatti, è associata ad un oggetto presente nel sistema mentre ogni riga è relativa ad un soggetto diverso: nell'incrocio tra ogni riga e colonna, allora, si avranno i diritti concessi al soggetto relativo alla riga sull'oggetto riferito dalla colonna.

	O1	**O2**	**O3**
S1	read, write	execute	write
S2		execute	read, write

Tabella 2.1: Esempio di matrice degli accessi

Ad esempio, nella tabella 2.1, la cella in alto a sinistra, indica che un processo che esegue nel dominio di *S1* ha i diritti di lettura e scrittura sull'oggetto *O1*, mentre quella

subito sotto, sempre nella prima colonna, stabilisce che, invece, se il processo esegue nel dominio di *S2*, allora non ha diritti su *O1*, e così via per tutte le celle della matrice.

Dunque, come si capisce dall'esempio appena fatto, **le celle vuote significano che l'utente relativo alla riga di tale cella non ha diritti sull'oggetto riferito dalla colonna**.

Con questa matrice, allora, è possibile rappresentare in modo intuitivo ed immediato le regole di protezione che sono presenti sul sistema in un certo istante. In particolare, essa **permette di rappresentare sia il modello ma, soprattutto, le politiche**, ovvero le regole di accesso.

In generale, bisogna tener presente che **le informazioni contenute nella matrice degli accessi variano nel tempo**: ad esempio, quando si va a creare un nuovo file nel file system, di fatto si aggiunge una nuova colonna alla matrice oppure, se l'amministratore elimina un utente del sistema, di fatto, rimuove una riga dalla tabella. Ma non solo, anche il contenuto stesso delle celle può cambiare, sia nel caso MAC, sia in quello DAC.

Si parla, allora, di **stato di protezione**: la matrice degli accessi, istante per istante, contiene delle informazioni che rappresentano lo stato di protezione. Con questo termine, appunto, si sottolinea il fatto che soggetti, oggetti e politiche variano nel tempo.

2.5 Meccanismi di protezione

In un dato istante t, la matrice degli accessi rappresenta l'insieme di politiche valide in quell'istante (e cioè lo stato di protezione); dunque, nel momento in cui un processo richiede l'accesso alle risorse, **i meccanismi hanno il compito di verificare se la richiesta sia consentita o no** e, qualora lo fosse, devono autorizzarla per consentire al processo di proseguire la sua esecuzione. In caso contrario, ovviamente, i meccanismi devono impedire al processo di continuare l'esecuzione.

Segue, allora, che i compiti principali dei meccanismi di protezione sono:

- verificare le richieste di accesso alle risorse che provengono dai processi;
- autorizzare o impedire l'esecuzione di richieste di accesso alle risorse;
- eseguire modifiche dello stato di protezione in seguito alle richieste autorizzate.

Ovviamente, quando un processo che esegue nel dominio dell'utente S_i fa richiesta di esecuzione di un'operazione M sull'oggetto O_j, i meccanismi devono verificare che tale operazione M sia presente nella cella (i, j) della matrice degli accessi: se la autorizzano, diversamente la impediscono.

Inoltre, un processo, all'occorrenza, oltre ad esercitare i diritti classici (lettura, scrittura ed esecuzione), può anche esercitare diritti che, una volta eseguiti, porteranno ad una modifica dello stato di protezione e, quindi, al contenuto stesso della matrice degli accessi.

2.5.1 Modifica dello stato di protezione

Come appena detto, un processo può richiedere la modifica dello stato di protezione: ma chi può modificarlo? Sulla base della classificazione precedente, lo stato di protezione può essere modificato:

- esclusivamente dall'**entità centrale** se la politica è di tipo **MAC**;
- dai **soggetti** (gli utenti) se la politica è di tipo **DAC**, ovviamente a patto che l'utente che richiede la modifica sia autorizzato a farlo.

Negli anni '70, ci fu un gruppo di ricercatori che cercò di formalizzare il discorso sulla modifica dello stato di protezione e, ancora oggi, si fa riferimento ad un articolo di *Graham-Denning* che fissa un opportuno insieme minimo di comandi che prevede **8 primitive** che riguardano gli oggetti, i soggetti e il contenuto delle celle della matrice degli accessi:

1. **create object** (crea un oggetto nel sistema);
2. **delete object** (elimina un oggetto nel sistema);
3. **create subject** (crea un soggetto nel sistema);
4. **delete subject** (elimina un soggetto nel sistema);
5. **read access right** (lettura di un diritto di un utente su una risorsa);
6. **grant access right** (concede ad un utente un diritto su una risorsa);
7. **delete access right** (eliminazione di un diritto di un utente su una risorsa);
8. **transfer access right** (concede ad un utente di esercitazione la propagazione di un diritto).

Le prima quattro primitive, dunque, modificano la struttura della tabella degli accessi mentre gli ultimi, invece, si riferiscono al contenuto vero e proprio della matrice.

È importante chiarire bene l'ultimo punto dell'elenco, ovvero il discorso della propagazione dei diritti di accesso. La possibilità di trasferire i diritti di un accesso da un dominio all'altro, è rappresentata da un particolare diritto, il ***copy flag***, indicato spesso con un asterisco (*): se un soggetto S_i ha accesso ad un oggetto O con il diritto α e, quest'ultimo, ha il copy flag, allora S_i può trasferire il diritto α ad un altro soggetto S_j.

Consideriamo la seguente tabella:

	O1	**O2**	**O3**
S1	read*, write	execute	write
S2		execute	read, write

Tabella 2.2: Matrice degli accessi con copy flag

La cella alla prima riga e alla prima colonna, indica il diritto di lettura e di scrittura di $S1$ sull'oggetto $O1$ e, inoltre, la presenza dell'asterisco nel diritto $read^*$ sta ad indicare, per quanto detto poco fa, che è attivo il copy-flag e che, dunque, $S1$ può propagare ad $S2$ il diritto di lettura[5].

L'implementazione del copy-flag, tuttavia, può prevedere due modalità:

[5]Nella tabella è presente solo $S2$ ma, se vi fossero altri soggetti, la propagazione sarebbe estesa anche a loro.

- **trasferimento del diritto**
  il soggetto che esercita il copy-flag trasferisce l'intero diritto (compreso il copy-flag) al nuovo soggetto e lo perde;

- **copia del diritto**
  il soggetto che esercita il copy-flag copia il diritto sull'oggetto al nuovo soggetto, ma non lo stesso copy-flag che, invece, viene mantenuto soltanto dal primo; la copia, pertanto, è una propagazione limitata del diritto in cui il soggetto originale mantiene il controllo sul trasferimento.

Oltre al copy-flag, esiste anche un altro particolare diritto: l'**owner**: **il soggetto che detiene questo diritto** per un certo oggetto è il proprietario di quest'ultimo e, come tale, **ha il controllo completo della colonna associata all'oggetto** di cui è proprietario. Ovviamente, ogni oggetto ha un solo proprietario il quale può concedere o revocare qualunque diritto sull'oggetto di cui detiene la proprietà a qualunque utente presente nel sistema.

Esiste poi, un altro diritto che si chiama **control** e che è presente in alcuni sistemi. Un esempio completo di matrice degli accessi, infatti, è il seguente:

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

Tabella 2.3: Esempio di matrice degli accessi con diritto owner e control

Rispetto agli esempi precedenti, quest'ultimo esempio di matrice degli accessi possiede anche una colonna per soggetto e questo perché è previsto l'impiego del diritto control. Tale diritto **abilita un soggetto a controllare un altro soggetto**. Questo significa che, ad esempio, nella tabella 2.3 il soggetto $S1$ controlla l'altro soggetto $S2$, nel senso che può andare ad apporre qualunque modifica al contenuto della matrice stessa per quanto concerne la riga del soggetto $S2$.

Allora, sulla base degli ultimi due diritti appena presentati, si conclude che un diritto di accesso per un oggetto X nel dominio di un soggetto S_j è eliminabile da parte di un altro soggetto S_i se:

- il diritto control è presente nella cella $A[S_i, S_j]$, ovvero S_i detiene il diritto control su S_j;

- il diritto owner è presente nella cella $A[S_i, X]$, ovvero S_i è il proprietario dell'oggetto X.

Per concludere, alla luce di quanto appena esposto, le 8 regole di *Graham* e *Denning* consentono una **modifica controllata di un sistema di protezione** ma, oltre a ciò, esse risolvono alcuni problemi tipici di tali sistemi, ovvero:

- **confinement**
  controllo e limitazione dei diritti di accesso;

- **sharing parameters**
  prevenzione di modifiche indiscriminate dei diritti di accesso di un processo;

- **trojan horse** uso non corretto dei diritti di accesso di un processo da parte di un altro.

2.5.2 Cambio di dominio

Nel corso delle sezioni precedenti e nell'ottica del discorso appena concluso, abbiamo sottolineato che lo strumento del cambio di dominio potrebbe essere uno strumento molto importante al fine di garantire il rispetto del principio del privilegio minimo.

Dunque, in un sistema di protezione, è molto importante garantire la disponibilità dei processi di cambiare dinamicamente il dominio di protezione: questo meccanismo si può formalizzare con un determinato diritto detto **switch**.

Consideriamo la seguente matrice degli accessi, anch'essa estesa nelle colonne:

	O1	**O2**	**O3**	**S1**	**S2**
S1	read, write	execute	write, owner		
S2		execute,owner	read, write	**switch**	

In una tabella di questo tipo, allora, è consentito ai processi il **cambio controllato nel dominio di protezione** esplicitato dalla presenza del diritto **switch**. Inoltre, in questo matrice il diritto switch di $S2$, quando esercitato, consente ad un processo che esegue in quel dominio di protezione di passare a quello di $S1$ a runtime.

Pertanto, considerando un sistema con una matrice degli accessi A e indicando due utenti con S_i e S_j, **lo switch presente nella cella** $A[S_i, S_j]$ **indica che un processo che esegue nel dominio di** S_i **può esercitare tale diritto e commutare al dominio di** S_j.

2.6 Realizzazione della matrice degli accessi

La matrice degli accessi è una notazione astratta che rappresenta, istante per istante, lo stato di protezione. Chiaramente, volendo implementare lo stato di protezione, è necessario che tutte le informazioni che sono contenute a livello astratto nella matrice degli accessi vengano concretamente rappresentate.

Gli aspetti fondamentali per rappresentare in modo concreto la matrice degli accessi sono due:

- **dimensione**
  Nella realtà la situazione è molto diversa da quella che finora abbiamo utilizzato negli esempi dato che, in generale, in un sistema si hanno decine di migliaia di file che, dunque, sono oggetti che vanno mappati nella matrice come colonne. Il numero di colonne, dunque, è molto considerevole anche se, in generale, anche gli utenti sono tanti di più rispetto agli esempi mostrati finora. Si conclude allora, che la dimensione della matrice è molto considerevole, soprattutto nel numero delle colonne.

- **matrice sparsa**
  Nella realtà, la matrice degli accessi può essere considerata sparsa nel senso che, in generale, sono molte di più le caselle vuote che quelle che contengono informazioni significative.

Sulla base dei due aspetti appena mostrati, ne consegue che è altamente inefficiente rappresentare le informazioni contenute nella matrice in un'unica struttura dati poiché sarebbe un grande spreco di memoria. Nella realtà, infatti, non esiste un sistema operativo

che prende la matrice degli accessi e la concretizza nella memoria esattamente così com'è, bensì vengono adottate delle strategie che rendano efficiente sia l'occupazione di memoria, sia le operazioni sulla tabella stessa.

In linea di principio, gli approcci possibili sono due:

2.6.1 Access Control List

Rappresentazione ottimizzata per colonne

Semplificando, è come se si rappresentassi la matrice degli accessi per colonne, ovvero per ogni oggetto (e, quindi, per ogni colonna), si costruisce una **lista che contiene tutti i soggetti che possono accedere a tale oggetto con i relativi diritti**. Il grande vantaggio è che *ACL* di un oggetto conterrà tanti elementi quanti sono i soggetti autorizzati ad accedere, in uno o più modi, a quello stesso oggetto: i soggetti che, invece, non hanno alcun diritto su di esso non sono presenti nella lista permettendo, quindi, un risparmio in termini di spazio della struttura dati.

Le entry di una ACL sono nella forma:

```
soggetto, insieme di diritti
```

Dunque, nei sistemi che adottano la Access Control List, per ogni oggetto viene costruita questa lista che è rappresentata da elementi in cui le informazioni contenute sono il soggetto e l'insieme dei diritti accordati ad esso per l'oggetto che si sta considerando: ovviamente, ci si limita ai soggetti che hanno un insieme di diritti non vuoto per l'oggetto. Segue, allora, che quando deve essere eseguita una certa operazione M su un oggetto O_j da parte di un soggetto S_i, concettualmente occorre seguire questa sequenza di passaggi:

1. si recupera la ACL dell'oggetto O_j considerato;
2. si va a vedere se in tale lista esiste un elemento relativo ad S_i;
3. se l'elemento esiste, si ricerca nell'elenco dei diritti associati a S_i se compare l'operazione M richiesta.

È possibile fare un'ulteriore ottimizzazione per rendere più efficiente il meccanismo di ricerca e verifica creando una **lista di default** che contiene tutti i diritti applicabili a tutti gli oggetti: se, ad esempio, tutti i soggetti posseggono uno stesso diritto su un oggetto, allora grazie a questa lista è possibile indicare tale diritto una volta per tutte, senza dover scorrere l'ACL dell'oggetto.

In aggiunta, è doveroso sottolineare che in molti sistemi la situazione si complica poiché al concetto di utente è affiancato quello di **gruppo** un'entità con un nome che, in linea di principio, rappresenta un insieme di utenti e che può essere inserito nell'ACL. In questo caso, allora, tale lista ha una forma un po' più complicata per cui ogni elemento ha una coppia gruppo, utente e un insieme di diritti, come segue:

```
UID, GID:<insieme di diritti>
```

Il concetto di gruppo, in alcuni casi, identifica il concetto di **ruolo** che un utente può ricoprire nell'ambito di uno stesso sistema (come, ad esempio, in Unix) e, dunque, uno stesso utente può appartenere a gruppi diversi con diritti diversi. È anche possibile svincolare l'utente dal gruppo concedendogli un diritto su un certo oggetto a prescindere da quale sia il suo gruppo (generalmente si una `*` come GID nella ACL).

2.6.2 Capability List

Rappresentazione ottimizzata per righe

Ogni soggetto ha una struttura dati associata ad esso, la *Capability List*, ovvero una **lista che contiene gli oggetti ad esso accessibili con i diritti di accesso**; in sostanza, tale lista è la rappresentazione di un dominio di protezione. Anche in questo caso, l'ottimizzazione è realizzata grazie all'omissione di elementi che non contengono contenuto informativo.

Ogni soggetto, allora, ha associata una propria Capability List e ogni elemento di tale lista è associato ad un particolare oggetto a cui i processi, che operano nel dominio di quel soggetto, possono accedere. In quell'elemento, ovviamente, oltre ad esserci il riferimento all'oggetto, ci sono anche quelli che sono i diritti di accesso consentiti su di esso.

Gli elementi di questa lista, inoltre, prendono il nome di **capability**, da cui il nome di tale struttura.

Si specifica che il riferimento all'oggetto può essere un identificatore o, più spesso, un indirizzo che consente di riferire l'oggetto mentre la rappresentazione dei diritti concessi, ad esempio, può essere un insieme di bit.

Se un soggetto S_i intende eseguire un'operazione M su un oggetto O_j, allora

1. si reperisce la capability list del soggetto;
2. si va a cercare nella lista se esiste un elemento associato ad O_j;
3. si verifica se l'operazione M richiesta è prevista per quell'oggetto.

In molti sistemi, inoltre, **la capability list è dinamica** e l'uso del puntatore consente, una volta verificati i diritti di accesso, di poter accedere in modo rapido e diretto alla risorsa. Ovviamente, c'è tutto un discorso sulla protezione delle liste di capability che, dunque, sono gestite in modo centralizzato, ovvero solo dal sistema operativo: ogni volta che un utente deve accedere ad una risorsa, farà riferimento ad un puntatore che consente di identificare la sua posizione nella lista che, però, **appartiene ad uno spazio del kernel**. Sarà poi, compito di quest'ultimo, accedere e fare aggiornamenti alla struttura in modo del tutto controllato al fine di evitare manomissioni.

In sistemi obsoleti, però, la soluzione adottata per la protezione delle capability era fortemente basata sull'hardware: esistono delle architetture dette *etichettate* nelle quali la memoria è organizzata in modo tale che ad ogni parola sia associato un bit speciale, il *tag*, che ha lo scopo di esprimere un vincolo di protezione su quella cella di memoria. Se una parola riferisce una capability, allora, il tag viene settato in modo tale che quella parola sia protetta da scritture non autorizzate; ovviamente la gestione del tag e del contenuto della cella può essere fatta solamente in modalità privilegiata. Tale approccio, però, è molto invasivo sulla rappresentazione delle informazioni in memoria per cui è poco utilizzato.

2.6.3 La revoca dei diritti

In un sistema, in generale, è possibile che sia richiesta un'operazione di **revoca** dei diritti di accesso detenuti da uno o più soggetti su di un oggetto.

È possibile introdurre le seguenti categorie di revoca:

generale	la revoca ha valore per tutti gli utenti che detengono il diritto
selettiva	la revoca ha valore solo per un sottoinsieme degli utenti che detengono il diritto
parziale	la revoca si riferisce ad un sottoinsieme di diritti detenuti su un oggetto
totale	la revoca vale per tutti i diritti detenuti su un oggetto
temporanea	la revoca è reversibile
permanente	la revoca è irreversibile

La revoca dei diritti, in realtà è un'**operazione molto frequente**: capita molto spesso di cancellare un file ma, dal punto di vista della protezione, la cancellazione di un file comporta una revoca generale, parziale e permanente su quell'oggetto.

Dunque, è necessario ragionare sul costo di questo genere di operazioni:

- **in un sistema con Access Control List**
  la revoca è poco costosa poiché consiste nella modifica della sola ACL associata a quell'oggetto; se si volesse eliminare un file, allora, sarebbe sufficiente andare ad eliminare la ACL ad esso associata.

- **in un sistema con Capability List** la revoca risulta molto più costosa perché le informazioni relative ai diritti di accesso di un particolare oggetto sono spalmate e distribuite su potenzialmente tutte le capability list: dunque, un'operazione di questo tipo comporta di andare a verificare in ogni singola capability list se esiste un elemento che riferisce a quel particolare oggetto e, nel caso, andare ad aggiornare il contenuto di quell'elemento.

Il discorso è analogo per la revoca dei diritti ad un particolare soggetto ma, statisticamente, è molto meno frequente: ad ogni modo, in questo caso, sarebbero più vantaggiose le capability list. Infatti, volendo fare un'operazione che riguarda un solo utente, sarebbe necessario aggiornare esclusivamente la capability list di quest'ultimo.

2.6.4 Approcci misti

Sulla base di quanto appena detto, per ogni operazione che riguarda un singolo oggetto, le ACL sono molto più efficienti ma, di contro, l'accesso ad ogni oggetto e la verifica dei diritti su di essi da parte di un soggetto, che viene rappresentato da un processo in esecuzione, è molto più efficiente con le capability list.

Di solito, nella maggior parte dei sistemi operativi, si adotta una **combinazione dei due metodi**: essa consiste nell'impiego di una ACL di base e poi, a runtime, man mano che i processi eseguono e richiedono l'accesso alle singole risorse, viene costruita una capability list relativa all'utente.

Definizione 27: ***Soluzione mista (ACL-Capability List)***

Approccio misto in cui il sistema si occupa di rendere persistente lo stato di protezione mediante ACL ma viene creata dinamicamente, a tempo di esecuzione, una capability list.
Nello specifico, quando un processo che opera per conto di un soggetto tenta di accedere ad un certo oggetto per la prima volta, allora:

1. *viene consultata la ACL relativa all'oggetto per verificare se la richiesta di accesso è consentita oppure no;*
2. *se la richiesta ha successo allora viene costruita la capability per quel soggetto, riferita a quell'oggetto;*
3. *la capability viene caricata in memoria centrale;*
4. *per i prossimi accessi a quella stessa risorsa, la capability in memoria centrale consentirà un accesso molto più veloce ed efficiente, senza dover passare ogni volta per l'ACL;*
5. *al completamento della sessione di accesso alla risorsa, la capability viene cancellata.*

Esempio 2: ***Apertura di un file in Unix***

L'approccio misto appena mostrato è adottato in Unix.
Infatti, quando viene aperto un file in Unix:

1. *viene richiesto l'accesso ad un file in una certa modalità (system call* `open`*, a cui si passano i diritti di accesso come, ad esempio, read o write);*
2. *a fronte della richiesta, si innesca un meccanismo per la costruzione di una capability per quel file ovvero, viene creata una nuova entry nella tabella dei file aperti contenente il file descriptor e i diritti di accesso: tale elemento altro non è che la capability;*
3. *viene ritornato il file descriptor;*
4. *da quel momento in poi, il file descriptor è lo strumento che consente di accedere alla capability, e quindi che permette in modo rapido di accedere al file con quei permessi.*

Dunque, le ACL sono presenti nel file system e, di fatto, esse si basano sui valori dei bit di protezione che vengono stabiliti per ogni file; in seguito, a runtime, viene costruita la capability e tramite file descriptor si può accedere nuovamente al file senza dover consultare la ACL.

2.7 Sicurezza

La sicurezza si occupa del **controllo degli accessi al sistema** tramite meccanismi di identificazione, autenticazione e autorizzazione ed ha come compito quello di proteggere il sistema da utenti esterni che potenzialmente possono provocare delle attività dannose.

Generalmente, affinché un sistema operativo sia sicuro **è necessario che sia presente al suo interno un sistema di protezione**; tuttavia, in alcuni ambienti può essere richiesto un controllo più stretto sulle regole di accesso: si parla, allora, di **sicurezza multilivello**.

Questo genere di controllo sugli utenti che sono autorizzati ad accedere al sistema viene tipicamente realizzato mediante politiche di tipo MAC[6]: **esiste un'entità centrale che, in generale, classifica gli utenti e stabilisce cosa possono o non possono fare**. Ovviamente, ciò si integra con il sistema di protezione che è già presente all'interno del sistema operativo e, qualora quanto stabilito dal sistema di sicurezza vada in contraddizione con quanto stabilito da quello di protezione, quello di sicurezza ha la precedenza.

Il sistema di sicurezza, allora, **classifica gli utenti** e, dunque, suddivide quelli autorizzati in categorie diverse secondo il proprio ruolo o la loro funzione all'interno dell'organizzazione che utilizza il sistema ma, oltre a questo, **classifica anche le risorse** e, quindi, stabilisce a priori una classificazione di queste a livelli diversi: una volta stabilite quali risorse appartengono a quali livelli e quali utenti appartengono a quali livelli, il sistema di sicurezza **stabilisce delle regole generali**[7].

In un sistema che necessita di controlli più stringenti sull'accesso su hanno due livelli di protezione: a livello basso c'è la protezione delle risorse mentre a livello più esterno c'è la sicurezza. In particolare, si parla di sicurezza multilivello.

Definizione 28: ***Sicurezza Multilivello***

Tecnica generale per la sicurezza in cui utenti e risorse ***sono classificati in livelli****: si definiscono* ***clearance levels*** *i livelli per i soggetti mentre sono detti* ***sensitivity level*** *quelli per gli oggetti. Una volta eseguita a priori questa classificazione, a questo punto, il modello fissa le* ***regole di sicurezza*** *che controllano il flusso di informazioni tra i livelli diversi della gerarchia.*
Dunque, la sicurezza multilivello si basa su più livelli di protezione e, inoltre, può essere implementata da più protocolli diversi . I due modelli principali di riferimento sono ***Bell-La Padula*** *e* ***Biba****.*

Sistema Multilivello = classificazione in livelli + regole di sicurezza

2.7.1 Modello *Bell-La Padula*: segreti al sicuro

Modello nato in un contesto militare con l'obbiettivo primario di **garantire la confidenzialità** delle informazioni di un'organizzazione e, cioè, per *mantenere i segreti.*

[6]Normalmente, nei sistemi *general purpose* in cui non è richiesto un grado di sicurezza così elevato, è sufficiente utilizzare politiche DAC.

[7]Sulla base di quanto appena esposto, risulta chiaro che potrebbe accadere, ad esempio, che una regola dica che un utente ad un certo livello non possa mai scrivere su un risorse che si trovano a livello più basso mentre le politiche di protezione potrebbero, invece, prevedere che tale utente abbia un diritto su un file a livello sottostante: dunque, in questo caso, c'è un conflitto tra le regole previste dal sistema di protezione e quelle imposte da quello di sicurezza. La priorità, come detto, è del sistema di sicurezza per cui l'utente non potrà accedere a quel file.

Il modello prevede **4 livelli di sensibilità degli oggetti**, la cui sicurezza aumenta salendo di livello:

1. non classificato;
2. confidenziale;
3. segreto;
4. top secret;

Dunque, ad esempio, le risorse a livello top secret non possono essere mai visibili a quello non classificato. Le risorse di sistema, allora, sono catalogate inserendo ognuna di esse in un solo livello.

Anche per i soggetti, però, sono previsti quattro livelli e sono esattamente gli stessi di quelli degli oggetti: gli utenti, allora, sono classificati secondo i livelli in base a quello che possono fare e, dunque, delle risorse a cui possono avere accesso.

Per quanto riguarda, invece, le **regole di sicurezza** di questo modello, esse sono due:

1. **Proprietà di semplice sicurezza**
   stabilisce che un processo in esecuzione su un dato livello può leggere oggetti collocati esclusivamente al suo livello o in livelli inferiori. Questo significa che un soggetto collocato in un dato livello k non potrà mai leggere in un livello $k + x$ in modo da garantire la segretezza; infatti, se i documenti più segreti sono mantenuti al livello più alto, un impiegato, che sicuramente sarà a livelli più bassi, non potrà mai leggere le informazioni contenute in quello massimo.

2. **Proprietà ***[8]
   stabilisce che un processo in esecuzione su un dato livello può scrivere documenti soltanto al proprio livello o in livelli superiori. Questa proprietà, allora, si potrebbe dire che è antitetica a quella precedente.

Poiché il modello si basa sulle regole appena mostrate, questo viene anche chiamato **No read up, no write down** in quanto i processi possono leggere verso il basso e scrivere verso l'alto ma non il contrario. Secondo questa considerazione, allora, si può concludere che in un'organizzazione basata su questo modello, le informazioni seguono un preciso flusso di propagazione che va dal basso verso l'alto ma non viceversa[9].

Il fatto che si possa scrivere su livelli superiori, però, significa che è possibile aggiornare o anche alterare risorse che sono sopra ma, in un sistema sicuro che adotta questo modello, c'è anche un sistema di protezione e, dunque, ci penseranno le regole di protezione a stabilire che l'utente al livello inferiore non può comunque scrivere su un documento segreto e critico che si trova a livelli superiori.

Il modello Bell-La Padula, oltre a garantire un'assoluta confidenzialità delle informazioni, permette di difendere il sistema dai *cavalli di Troia*, ovvero meccanismi che hanno come obiettivo quello di rubare informazioni sensibili all'interno del sistema attaccato. Per capire meglio il meccanismo sfruttato per il furto, si consideri un sistema con un utente accreditato ed autorizzato ad ad utilizzare il sistema, che chiameremo Paolo e un secondo utente ostile, Marco.

[8] La proprietà ha questo nome poiché, quando è stato definito il modello, ai creatori non è venuto in mente un nome per questa regola e quindi hanno messo un asterisco che poi è rimasto definitivamente.

[9] Ovvero chi sta sotto può scrivere verso sopra ma non viceversa.

- Paolo ha creato un file F_P che contiene delle stringhe di caratteri riservati e assolutamente confidenziali;
- data la riservatezza del contenuto, paolo avrà opportunamente impostato i permessi sul file in modo da vietare l'accesso a tutti gli altri utenti del sistema ad eccezione di sé stesso;
- l'utente ostile, Marco, riesce ad iniettare nel sistema un file eseguibile CT, magari grazie ad un allegato di posta elettronica;
- tale eseguibile è un cavallo di troia e ad esso viene allegato un secondo file privato F_M (oppure lo crea lui quando in esecuzione) che servirà all'attaccante per depositare le informazioni rubate (*back pocket*, ovvero *tasca posteriore*);
- Marco, allora, induce Paolo ad eseguire il cavallo di Troia, ingannandolo, magari spacciandolo per un programma di utilità;
- se Paolo esegue il cavallo di Troia, allora, se, ad esempio, ci troviamo in un sistema Unix, il processo relativo al programma appena lanciato prenderà lo user id di Paolo e, dunque, potrà leggere il contenuto di F_P;
- il cavallo di Troia, allora, legge F_P estraendone le informazioni sensibili e le scriverà nel file F_M su cui Marco si sarà premurato di dare il diritto di scrittura a Paolo (oltre a quelli di lettura e scrittura dello stesso Marco);
- CT, allora, può trasferire le informazioni contenute in F_M a Marco.

Le regole classiche di protezione, purtroppo, non possono fronteggiare un cavallo di Troia e neanche evitare la trasmissione di informazioni all'attaccante, ma se ad esse è stato abbinato un sistema di sicurezza multilivello basato su Bell-La Padula. Supponiamo, per semplicità, che i livelli siano due: uno riservato e uno pubblico. Allora, nell'esempio di prima, poiché Paolo è un utente fidato, allora gli viene assegnato il livello riservato mentre a Marco e ai suoi file, viene assegnato il più basso, quello pubblico. Segue, allora, che quando Paolo andrà ad eseguire il *trojan horse*, il processo che viene generato acquisirà il livello di sicurezza associato a Paolo, ovvero quello massi e, dunque, può leggere le informazioni riservate. Tuttavia, **il meccanismo fallisce quando il processo tenta di andare a scrivere le informazioni nella *back pocket*, poiché quel file si trova inevitabilmente al livello più basso** e, quindi, il tentativo di scrittura in F_M fallisce in linea con le regole del modello di Bell-La Padula e indipendentemente da quanto stabilito dal sistema di protezione, dato che quello di sicurezza ha la priorità.

2.7.2 Modello *Biba*: dati integri

Altro modello molto noto che, a differenza di quello precedente, ha l'obbiettivo di **garantire l'integrità dei dati**. Si presta molto a supportare il lavoro cooperativo di soggetti inquadrati all'interno di una stessa organizzazione; ad esempio, potrebbe essere impiegato in un'azienda che sviluppa software.

Le regole che impone sono antitetiche rispetto a Bell-La Padula, anche se i livelli sono sempre quattro (sia per gli oggetti e sia per i soggetti) e sono regolati dalle seguenti proprietà:

1. **Proprietà di semplice sicurezza**

   noindent un processo in esecuzione su un certo livello può scrivere solamente al suo livello o a quelli più bassi.

2. **Proprietà di integrità ***
   un processo in esecuzione su un certo livello può leggere solamente al suo livello o a quelli superiori.

Come si può notare, queste regole sono esattamente il contrario di quelle stabilite da Bell-La Padula e, dunque, anche verso di propagazione delle informazioni è il contrario: se per il primo modello andava dal basso verso l'alto, in Biba va dall'alto verso il basso.

Dunque, i documenti che si trovano al livello più basso possono essere scritti non soltanto dai soggetti che si trovano su quel livello ma anche da tutti quelli che si trovano sopra nella gerarchia e questo si presta molto bene, ad esempio, quando si deve affrontare il progetto di un certo prodotto o di un software articolato in cui sono coinvolti più sviluppatori di una stessa azienda.

I modelli Bell-La Padula e Biba sono in conflitto tra loro e non potranno mai essere combinati in quanto sono esattamente l'uno il contrario dell'altro.

2.7.3 Architettura dei sistemi ad elevata sicurezza

Definizione 29: ***Sistema sicuro (o Trusted System)***

Un sistema è detto sicuro quando è possibile riconoscere e definire formalmente dei requisiti di sicurezza.

Sistemi di questo tipo, allora, hanno un *cuore* che viene definito **Reference Monitor**:

Definizione 30: ***Reference Monitor***

Componente tipico di un sistema sicuro che è in grado di attuare tutte le direttive specifiche di quel sistema per quanto riguarda la sicurezza. Dunque, è un elemento di controllo che generalmente è realizzato sia a livello hardware, sia software poiché ha il compito di regolare l'accesso dei soggetti agli oggetti mettendo in pratica il modello di sicurezza adottato dal sistema.

Il Reference Monitor, per svolgere il suo compito, fa riferimento ad un altro componente, il **Trusted Computing Base**:

Definizione 31: ***Trusted Computing Base (TCB)***

Base di calcolo fidata che contiene le esplicitazioni delle caratteristiche di sicurezza di un sistema, ovvero:

- *le autorizzazioni di sicurezza di ogni soggetto;*
- *la classificazione degli oggetti per quanto riguarda i livelli definiti all'interno di quel sistema.*

Il Reference Monitor, allora, conosce le regole ed è in grado di verifica a runtime che ogni operazione richiesta da un certo soggetto su un certo oggetto sia conforme a quanto

specificato dalle regole e dalla Trusted Computing Base. Il Reference Monitor, inoltre, è l'unico abilitato ad accedere alla TCB.

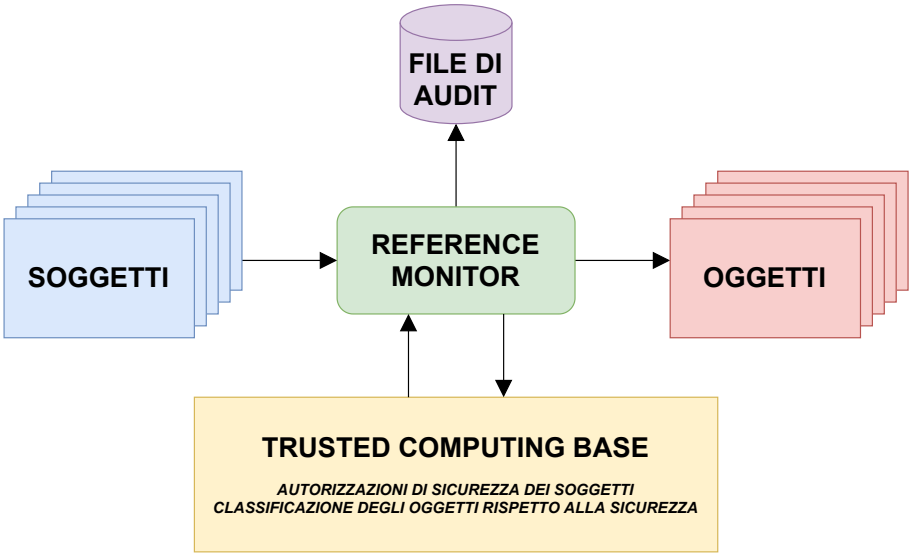


Figura 2.1: Architettura dei sistemi ad elevata sicurezza

La figura 2.1 schematizza l'architettura di un sistema ad elevata sicurezza. In particolare, al momento in un cui un soggetto richiede di accedere ad un oggetto, il Reference Monitor verifica se la richiesta possa essere soddisfatta , ovvero se è conforme ai dati presenti nella TCB, e consente l'accesso qualora la verifica vada a buon fine. Di solito, è presente anche il **file di audit**, un file di log in cui vengono tracciati, man mano, gli eventi che accadono nel sistema e che riguardano la sicurezza.

Proprietà fondamentali del Reference Monitor

Sulla base di quanto detto, allora, è chiaro che il Reference Monitor ha il compito di far rispettare le regole di sicurezza. Inoltre, esso, idealmente, deve offrire le seguenti proprietà:

- **Mediazione completa**
  Le regole di sicurezza vengono applicate ad ogni accesso, non solo al primo. In Unix, ad esempio, questa mediazione non c'è, dato che la richiesta di apertura di un file comporta una verifica *una tantum* e, da quell'apertura in poi, questa verifica non verrà più fatta. In un sistema fidato, invece, ad ogni singolo accesso viene ri-testato il soddisfacimento della regola di sicurezza, anche se questo comporta un certo *overhead*: è proprio per questo motivo che il Reference Monitor non è puro software, ma deve essere implementato il più possibile a livello hardware per questioni di efficienza.

- **Isolamento**
  Poichè il Reference Monitor e la TCB sono vitali per garantire la sicurezza del sistema, queste due componenti devono essere a loro volta protetti rispetto a modifiche non autorizzate e questo, tipicamente, viene fatto concedendone l'accesso soltanto a chi opera in modalità privilegiata, ovvero al kernel. Lo scopo, dunque, è quello di evitare ogni tipo di alterazione della TCB e del Reference Monitor da parte ogni eventuale utente o persona che attacca il sistema.

- **Verificabilità**
  Deve essere possibile dimostrare formalmente, certificare, che il Reference Monitor opera correttamente, ovvero che effettivamente questo imponga il rispetto delle regole di sicurezza e che fornisca mediazione completa e isolamento. Tale proprietà, sicuramente, è molto difficile da soddisfare e la sua complessità cresce all'aumentare di quella della struttura del Reference Monitor e, più in generale, del sistema. Inoltre, la verificabilità contraddistingue i sistemi che hanno i livelli di sicurezza più elevati di tutti.

2.7.4 Classificazione della sicurezza

Per classificare la sicurezza, il Dipartimento della Difesa americano (D.O.D.) ha rilasciato un documento, l'**Orange Book**, che, sostanzialmente, stabilisce le classifiche dei sistemi in funzione delle caratteristiche di sicurezza. In particolare, questo documento abbastanza corposo specifica quattro categorie corrispondenti a quattro livelli di sicurezza: A, B, C, D (ordine decrescente, con D che offre meno protezione di tutti)

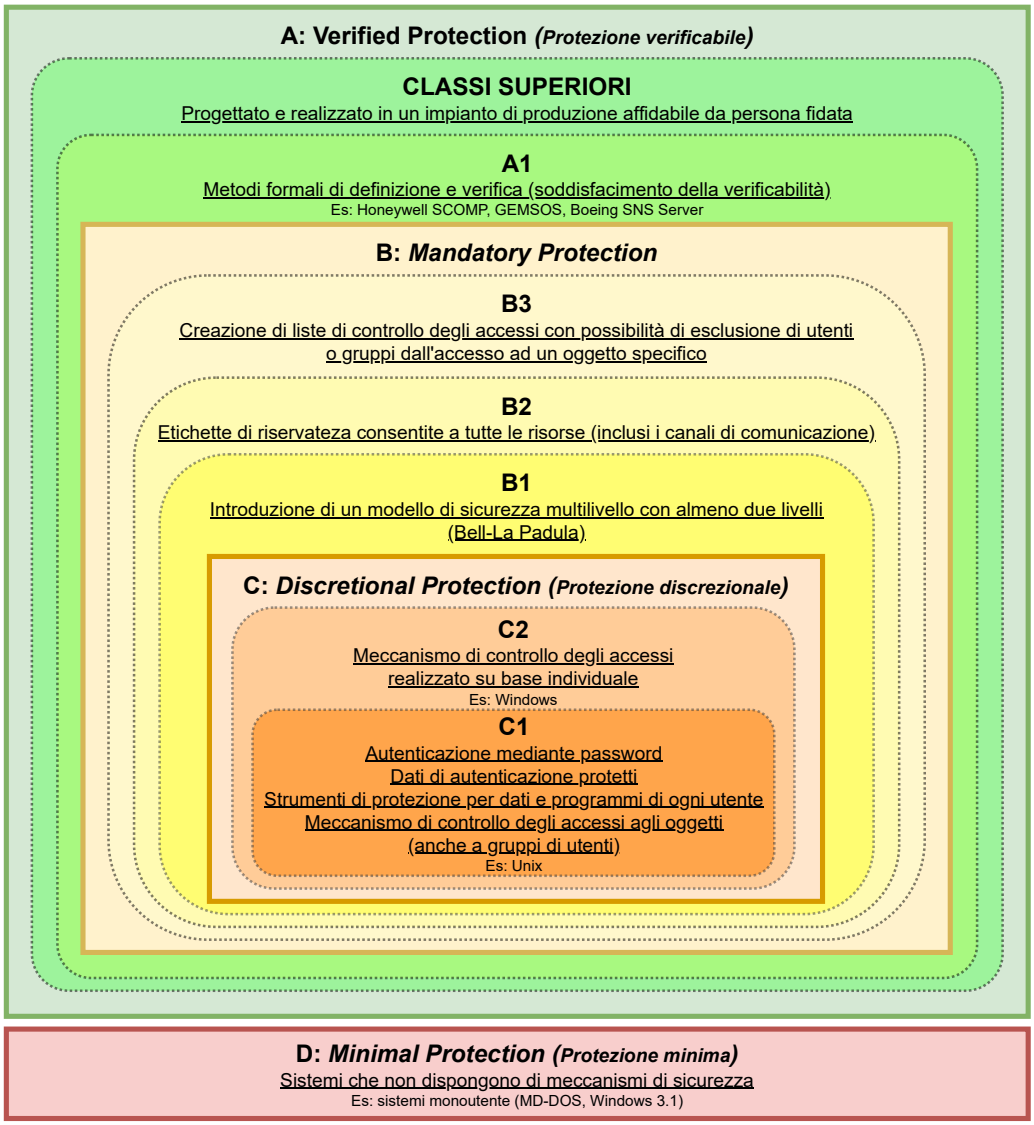


Figura 2.2: Livelli di sicurezza stabiliti dal documento *Orange Book*

Capitolo 3

Programmazione Concorrente

3.1 Richiami ed introduzione sulla programmazione concorrente

Una delle cose principali che la programmazione concorrente è in grado di fare è quella di fornire i meccanismi per la creazione di applicazioni che sfruttino l'eventuale parallelismo presente a livello hardware.

Definizione 32: ***Programmazione Concorrente***

Insieme di tecniche, metodologie e strumenti che servono per sviluppare e supportare l'esecuzione di sistemi software in grado di portare avanti più attività ***simultaneamente****.*

In linea generali, si hanno attività simultanee quando, ad un certo istante, si hanno più di un'attività - che potrebbero essere processi - iniziate ma non ancora terminate nell'ambito dell'esecuzione dello stesso programma.

Si ricorda che esistono due modelli principali:

- **memoria comune**
  condivisione di una o più variabili
- **memoria distribuita**
  scambio di messaggi

La programmazione concorrente nasce negli anni '60, quando l'evoluzione delle architetture e dei sistemi operativi ha reso concreta la possibilità per il programmatore di pensare a programmi la cui esecuzione potesse procedere grazie a diverse attività simultanee che, a seconda del contesto, prendono il nome di **processi** o **thread**.

Tutto questo è stato inizialmente provocato dall'introduzione, a livello architetturale, dall'**introduzione dei canali** o controllori di dispositivi che hanno permesso ai sistemi di eseguire e gestire attività diverse (esecuzione concorrente di istruzioni di I/O e delle istruzioni dei programmi).

Lo sviluppo dei sistemi moderni, dunque, si basa sul meccanismo di **interruzione** e **time-sharing**, grazie alle quali, periodicamente, è possibile realizzare la commutazione di contesto, ovvero l'avvicendamento di due o più programmi nell'uso di un'unica CPU a disposizione a livello architetturale. La possibilità di un sistema operativo di gestire

più programmi contemporaneamente in esecuzione, tuttavia, ha posto il problema del non determinismo: parti diverse dello stesso programma in esecuzione possono essere eseguite in un ordine assolutamente non predicibile causando, di conseguenza, interferenze nell'accesso a risorse comuni.

In seguito, però, tutti i sistemi iniziarono ad offrire la possibilità di disporre di un certo numero di processori o core, in grado di portare avanti in modo parallelo attività diverse. Essi prendono il nome di **sistemi multiprocessore** e, dunque, offrono la possibilità a differenti processi di una stessa applicazione di eseguire in **reale parallelismo**, aprendo nuovi scenari in termini di efficienza.

Il concetto base su cui si fonda la programmazione concorrente, dunque, è quello di attività simultanea.

Nel progetto di un'applicazione concorrente, allora, è necessario stabilire come questa debba essere suddivisa in processi, e quindi attività, simultanee e quante queste debbano essere. Inoltre, si ha sicuramente la necessità di mettere in interazione queste ultime e, quindi, bisogna garantirne la corretta sincronizzazione. Tali decisioni dipendono **dal tipo di applicazione** ma anche dal **tipo di architettura** di cui si dispone.

3.1.1 Tipologie di architetture

La figura 3.1 mostra le tre principali tipologie di architetture dei sistemi di elaborazione.

Shared-Memory Multiprocessors (Sistemi multiprocessore)

Questa tipologia di sistemi, in realtà, può essere di vari tipi anche a seconda della dimensione. In particolare, i modelli principali sono due:

1. **UMA (Uniform Memory Access)**
   Sistemi con un numero ridotto di processori, indicativamente da 2 a 30, caratterizzati dalla presenza di una rete di interconnessione che di fatto è realizzata da un memory bus[1]: per questo motivo, il tempo di accesso alla memoria è costante e non dipende dalla cella di memoria alla quale si vuole accedere. Generalmente, questo tipo di sistemi è chiamato anche *Symmetric Multiprocessors* (SMP) ma questo nome è utilizzato anche per indicare il modo con cui il sistema operativo gestisce questo tipo di architettura.

2. **NUMA (Non Uniform Access Time)**
   Sistemi con un numero molto elevato di processori, che possono essere anche nell'ordine delle centinaia ma sempre con un'unica memoria condivisa e accessibile da qualunque processore. Questa volta, però, a causa del grande numero di processori la memoria viene organizzata in modo gerarchico in modo da evitare la congestione del bus. Inoltre, ogni processore può accedere più velocemente alle locazioni di memoria che gli sono più vicine e, dunque, il tempo di accesso dipende dalla distanza tra processore e memoria, caratteristica principale che da il nome al modello.

[1] Per rendere questo tipo di sistema più veloce, si può anche utilizzare un *crossbar switch* ma, normalmente, c'è un bus.

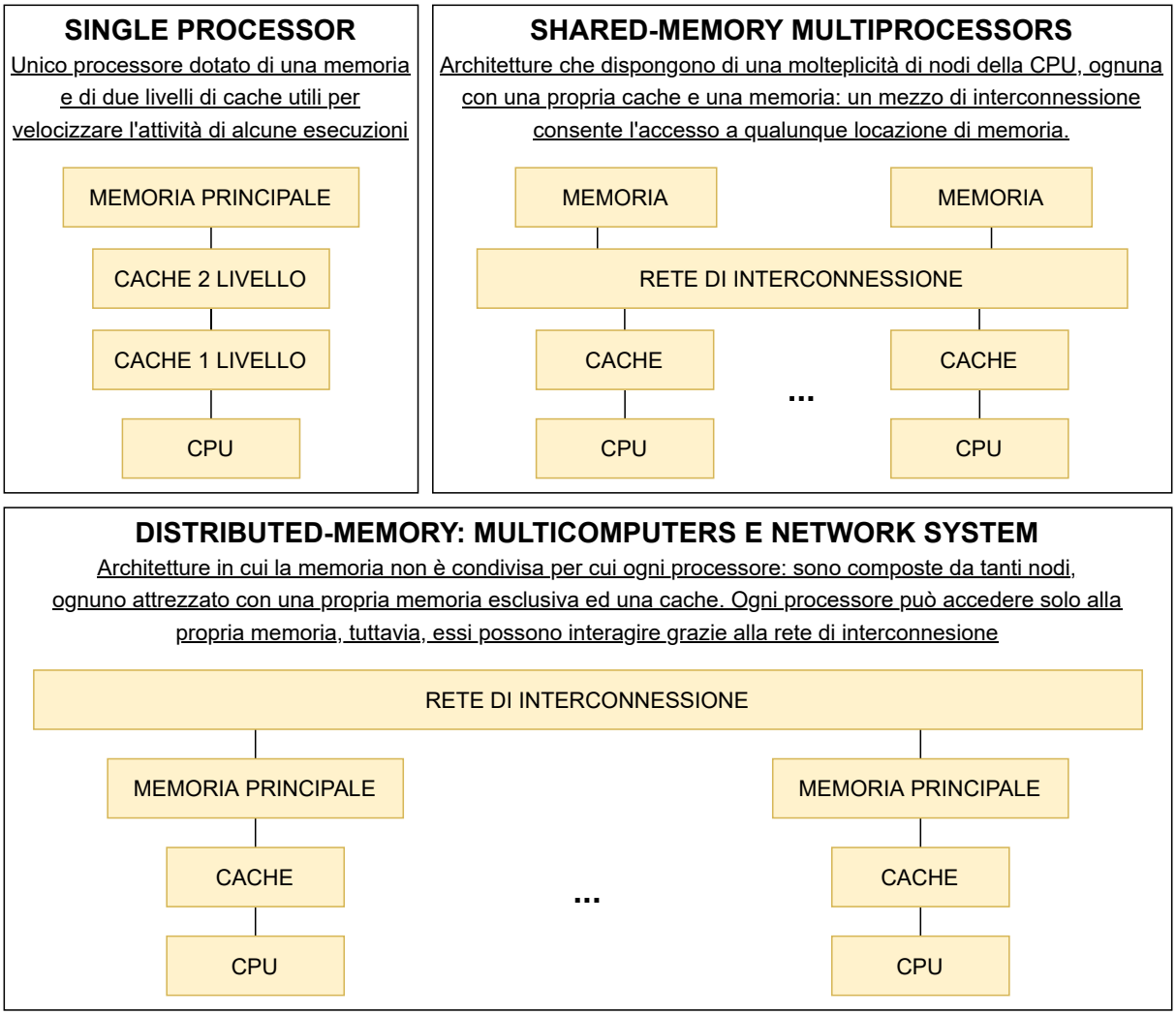


Figura 3.1: Principali tipologie di architetture dei sistemi di elaborazione

Distributed-Memory (Sistemi a memoria distribuita)

I sistemi di questo tipo sono composti da vari nodi che interagiscono tra di loro grazie ad una rete di interconnessione; ogni nodo è composto da una memoria, da un processore e da una cache. Si hanno, inoltre, due modelli:

1. **Multicomputer**
   Sistemi in cui i processori e la rete sono fisicamente vicini, ovvero appartengono alla stessa macchina o struttura fisica (ad esempio i *cluster*, un insieme di computer collegati fisicamente all'interno dello stesso cabinet). L'interconnessione è sicuramente più performante rispetto al secondo modello. In questa categoria rientrano anche alcuni sistemi di *high performance*, quindi architetture parallele, che sono composti da una molteplicità di nodi, ognuno dei quali ha una o più cpu, magari con un certo numero di core, e che sono collegati tra loro da una rete particolarmente performante che consente di rendere più veloci possibili gli scambi di informazione.

2. **Network Systems**
   Sistema di computer collegati tra di loro, ad esempio mediante una rete locale o anche grazie ad una rete geograficamente più ampia, in cui i vari nodi eseguono per attività che fanno parte, almeno concettualmente, di una stessa applicazione

concorrente. Dunque, vengono sfruttate le memorie locali dei nodi e la rete in modo tale che queste attività possano sincronizzarsi e portare a termine il compito assegnato all'applicazione.

Spesso, nel caso dei Distributed System, si può parlare di sistemi ibridi dato che, soprattutto oggi, i singoli nodi sono, in realtà, dei multiprocessori.

3.1.2 Classificazione di Flynn

La tassonomia (o classificazione) di Flynn ha come obiettivo quello di classificare i sistemi di calcolo soprattutto dal punto di vista della gestione della memoria e della capacità di parallelismo delle architetture. In particolare, si basa su due parametri principali di classificazione:

- **Parallelismo a livello di istruzioni**
  Se questa tipologia di parallelismo è presente, allora il sistema, in ogni istante, è in grado di eseguire parallelamente istruzioni diverse, ovvero può eseguire in modo parallelo più flussi di istruzioni. Allora, fotografando l'attività di un sistema di questo tipo in un dato istante, sarà possibile che esso stia eseguendo in quell'istante operazioni diverse sfruttando, magari, CPU diverse.

  Si distinguono, allora:

  - Single istruction set, le architetture in grado di eseguire, istante per istante, un singolo flusso di istruzioni (es: monoprocessore);
  - Multiple istruction set, quelle in grado di eseguire più flussi in parallelo.

- **Parallelismo a livello di dati**
  Il concetto è analogo a quello precedente ma, questa volta, il parallelismo riguarda i dati. In particolare, si ha:

  - Single data stream, l'architettura è un grado di elaborare un singolo flusso di dati, in altre parole, il processore o i processori, in ogni istante, potranno fare riferimento allo stesso, unico, dato indipendentemente da quel che stanno facendo;
  - Multiple data stream, architetture in cui le attività che si stanno svolgendo sui diversi processori, possono riferire anche dati diversi.

La figura 3.2 mostra una schematizzazione della tassonomia di Flynn.

Per quanto riguarda le architetture **MISD**, fino a qualche hanno fa, esse erano realizzate mediante computer messi in pipeline ed operanti sullo stesso stream di dati; oggi, in realtà, è una categoria in cui non si ritrovano sistemi e l'unico caso d'uso potrebbe essere la *fault tolerance*.

Invece, per ciò che concerne le architetture di tipo **SIMD**, esse sono composte da più unità di elaborazione tutte controllate dalla stessa unità di controllo e, dunque, ognuna di esse eseguirà la stessa istruzione delle altre però su dati diversi. Questo tipo di architetture si presta molto per tutti quei problemi in cui i dati sono forniti in forma vettoriale o matriciale e l'algoritmo da applicare è lo stesso (es: problemi di simulazione di fenomeni fisici).

Flusso di istruzioni	Flusso di dati	Nome	Descrizione
Singolo	*Singolo*	SISD	Il sistema può eseguire, ad ogni istante, **una singola istruzione su un solo dato**. *Computer singoli (Von Neumann)*
Singolo	*Multiplo*	SIMD	Il sistem può esegure, ad ogni istante, **una singola istruzione ma su più flussi di dati**. *Array processors, GPU (es: stessa operazione su tutti i pixel di un'immagine)*
Multiplo	*Singolo*	MISD	Il sistema, in ogni istante, può eseguire **istruzioni multiple su uno solo dato**. *Computer in pipeline*
Multiplo	*Multiplo*	MIMD	Il sistema dispone di più unità di elaborazione che, in ogni istante, **sono in grado** ognuna **di eseguire operazioni diverse su dati diversi**. *Multicomputer e Multiprocessori*

Figura 3.2: Tabella di verità della tassonomia di Flynn

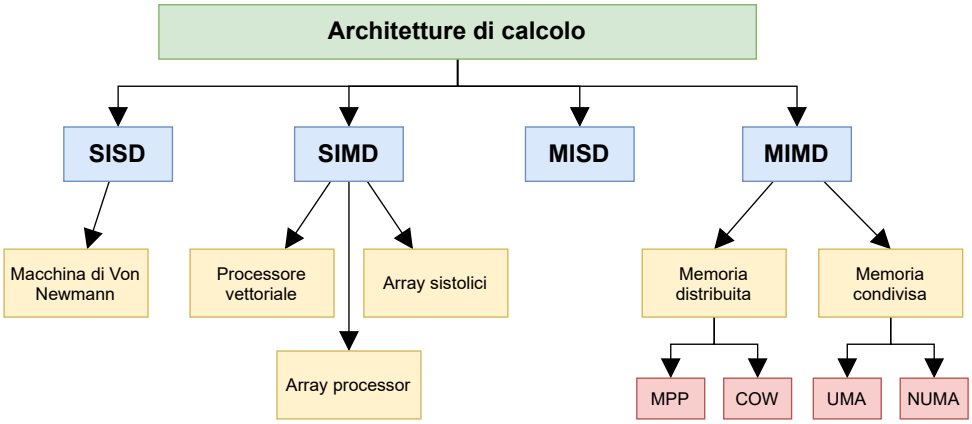


Figura 3.3: Grafico riassuntivo della tassonomia di Flynn

3.1.3 Livello di applicazione

In generale, il piano dell'applicazione e quello dell'architettura possono essere considerati indipendenti: ad esempio, è possibile avere come architettura un modello multiprocessore, in cui è prevista la memoria condivisa tra le varie unità di elaborazione, però a livello applicativo si può avere un sistema in cui i processi non possono condividere memoria, nonostante sotto questa sia comune.

Per questo motivo, **d'ora in avanti considereremo i due piani, architettura e applicazione, in modo distinto** anche se, nello sviluppo di un'applicazione, è chiaro che il modo più efficace è quello di tener conto di quelle che sono le reali caratteristiche dell'architettura per trarne il massimo vantaggio possibile.

Tipologie di applicazioni

Le principali tipologie di applicazioni in ambito concorrente sono le seguenti:

- **Sistemi Multithreaded**
  Sono le applicazioni strutturate con un insieme di processi leggeri, i thread che, a

seconda del tipo di applicazione e del suo obiettivo, hanno lo scopo di far fronte alle necessità, aumentare l'efficienza e semplificare la programmazione (ad esempio applicando il *Divide et Impera*). In generale, le applicazioni multithreaded sono caratterizzate dal fatto che esistono più processi che processori e, dunque, i processi sono schedulati secondo le politiche del sistema operativo ed eseguiti indipendentemente.

Il primo esempio di applicazione di questo tipo è il sistema operativo.

- **Sistemi Multitasking o Distribuiti**
  Le attività che si svolgono nell'applicazione sono eseguite su nodi, eventualmente virtuali, collegati tra loro mediante qualche infrastruttura[2] che consente di realizzare l'interazione. Il modello adottato è a scambio di messaggi e non è prevista occupazione di memoria.

  Vari esempi possono essere: sistemi web, client/server, database system, sistemi distribuiti pervasivi, ecc... Se l'architettura è di tipo distribuito, allora è chiaro che questa tipologia di sistemi è quella più efficace anche se, in linea di principio, non è una scelta obbligata. Infatti, è possibile realizzare un'astrazione di memoria condivisa anche se l'architettura hardware prevede memoria distribuita.

- **Applicazioni parallele**
  Questa tipologia di applicazioni, pur essendo, come nei casi precedenti, dei programmi le cui attività vengono svolte contemporaneamente, sono state concepite per trarre massimo vantaggio dall'architettura sottostante: in questo caso, allora, l'assunzione di indipendenza tra il livello di applicazione e quello di architettura non è valida. L'obiettivo primario di chi sviluppa questo tipo di software, infatti, è trarre massimo vantaggio dal parallelismo presente a livello hardware per cui, non è possibile prescindere da com'è stato sviluppato l'hardware in modo tale da massimizzare le performance.

3.1.4 Sequenzialità dei processi

Richiami introduttivi

Prima di parlare di processo non sequenziale, vale la pena fare alcuni richiami su concetti che potrebbero risultare scontati, ma che, in realtà, hanno rilevante importanza:

- **Algoritmo**
  Formalizzazione del procedimento logico che deve essere applicato ed eseguito per risolvere un particolare problema.

- **Programma**
  Descrizione di un algoritmo mediante un opportuno formalismo, il linguaggio di programmazione, che rende possibile l'esecuzione dell'algoritmo da parte di un particolare esecutore, l'elaboratore.

- **Processo**
  Insieme ordinato degli eventi che avvengono all'interno dell'elaboratore quando opera sotto il controllo di un determinato programma.

[2] Esempi di infrastruttura potrebbero essere i classici canali di comunicazione.

Dunque, l'**elaboratore** è l'entità astratta, tipicamente realizzata in hardware ma anche, parzialmente, in software, che è in grado di eseguire le operazioni specificate dai programmi. L'**evento**, invece, rappresenta l'esecuzione di un'operazione, tra quelle che l'elaboratore è in grado di riconoscere ed eseguire: ovvero, è l'entità che rappresenta l'esecuzione di una singola operazione. Dunque, quando un elaboratore è in attività, quest'ultima può essere descritta come un insieme ordinato di eventi che rappresentano l'attività di chi esegue; allora, ogni evento avviene in un certo istante e, in linea di principio, determina una **transizione di stato** dell'elaboratore.

Processo sequenziale

Abbiamo detto che un processo è un insieme ordinato di eventi che avvengono nell'elaboratore quando esegue un certo programma: dunque, tali eventi, in linea di principio, possono essere ordinati secondo una **relazione d'ordine totale**. In questo caso, allora, si parla di **processo sequenziale**, ma questa non è l'unica totalità.

Definizione 33: ***Relazione d'ordine totale***

In relazione ad un processo, una relazione d'ordine è ***totale*** *se gli stati che lo descrivono sono ordinati secondo un criterio rigidamente sequenziale ovvero se esaminandoli, per ogni coppia di stati, è sempre possibile dire quale dei due precede l'altro.*

Esempio: ***Massimo Comun Divisore tra due numeri***

Si supponga che i dati siano due numeri naturali x e y; allora è possibile calcolare il loro M.C.D. grazie al metodo di Euclide. L'algoritmo, in C, potrebbe essere il seguente:

```
int mcd(int x, int y) { /*****************************************/

     int a = x; int b = y;

     while(a != b) {
          if(a > b)
               a = a - b;
          else
               b = b - a;
     }

     return a;

} /************************************************************/
```

L'algoritmo valuta se i due numeri dati in ingresso alla funzione sono uguali e, qualora lo sono, ritorno il loro valore. Se sono diversi, riesegue la valutazione utilizzando la loro differenza e il più piccolo dei due.

L'esempio appena illustrato permette di mostrare l'evoluzione dello stato.

La figura 3.4, infatti, mostra l'evoluzione dello stato dopo una chiamata `mcd(18, 24)`: ogni colonna rappresenta uno stato ed ogni transizione è causata da un evento. Dunque, l'**evoluzione dell'esecuzione**, portata avanti dal processo che esegue la funzione, **è**

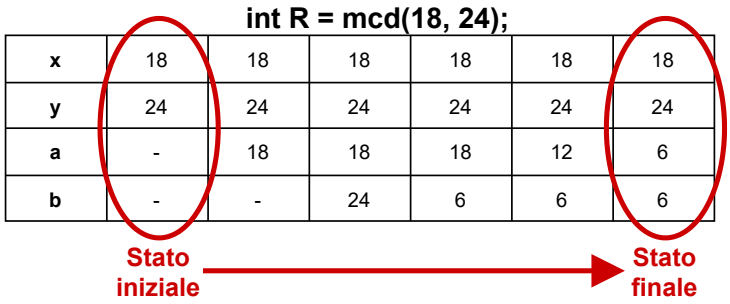


Figura 3.4: Transizioni di stato di una esecuzione dell'algoritmo di Euclide

descritta da una sequenza di stati attraverso i quali l'elaboratore passa per poter arrivare allo stato finale che è quello corrispondente al ritorno del controllo al chiamante.

Definizione 34: ***Traccia dell'esecuzione***

Si definisce Traccia dell'esecuzione la sequenza di stati attraversati dall'esecutore durante l'esecuzione di un programma

La traccia dell'esecuzione, ovviamente, cambia nel caso di dati di ingresso differenti. In generale, inoltre, è possibile **rappresentare un processo mediante un grafo orientato**, detto **grafo di precedenza** del progesso.

Definizione 35: ***Grafo di precedenza***

Grafo associato ad un processo e costituito da una serie di nodi collegati tra di loro da archi orientati, ovvero archi ai quali è stato attribuito un verso. Ogni nodo rappresenta uno stato che viene attraversato dall'esecutore nell'arco del programma mentre gli archi orientati, invece, sono le dipendenze e le precedenze temporali tra gli eventi.

In figura 3.5 è riportato il grafo di precedenza dell'esempio dell'algoritmo di Euclide. Si può osservare che dal primo stato incontrato (`a=18`) al secondo (`b=24`) c'è una relazione di rigida sequenzialità per come è fatto il programma: questo, infatti, stabilisce che prima si assegna ad `a` il valore di `x` e poi a `b` quello di `y`.

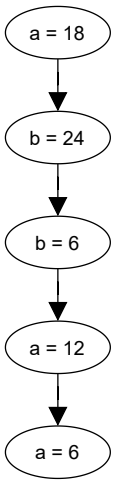


Figura 3.5: Esempio di Grafo di precedenza semplificato

Ciò che contraddistingue un processo strettamente sequenziale è che il grafo di precedenza è ad ordinamento totale, cioè se si focalizza l'attenzione si un particolare nodo intermedio, è possibile osservare che esso ha esattamente uno e un solo predecessore e uno e un solo successore.

Processi non sequenziali

I processi sequenziali appena illustrati non sono l'unica possibilità: si possono avere anche **non sequenziali**. A differenza di quanto visto finora, in quest'ultimo caso, l'insieme degli eventi che lo descrive è ordinato non più secondo una relazione d'ordine totale, bensì secondo una **relazione**

d'ordine parziale. Dunque, è possibile che il processo sia composto da eventi che non sono ordinati tra di loro.

Esempio: ***Valutazione di un espressione***

Si consideri un programma che abbia come scopo la valutazione di un espressione composta. L'espressione da valutare è la seguente:

$$(3 * 4) + (2 + 3) * (6 - 2)$$

In un ambiente sequenziale, in cui vige l'ordinamento totale, si potrebbe procedere valutando l'espressione da sinistra a destra, tenendo conto delle precedenze degli operatori. Questa, però, non è l'unica possibilità poiché il calcolo degli operandi che compaiono nell'espressione, in realtà, l'ordine non è fissato a priori: perché, allora, non tenere conto di questo e non adottare un ***ordinamento parziale****?*

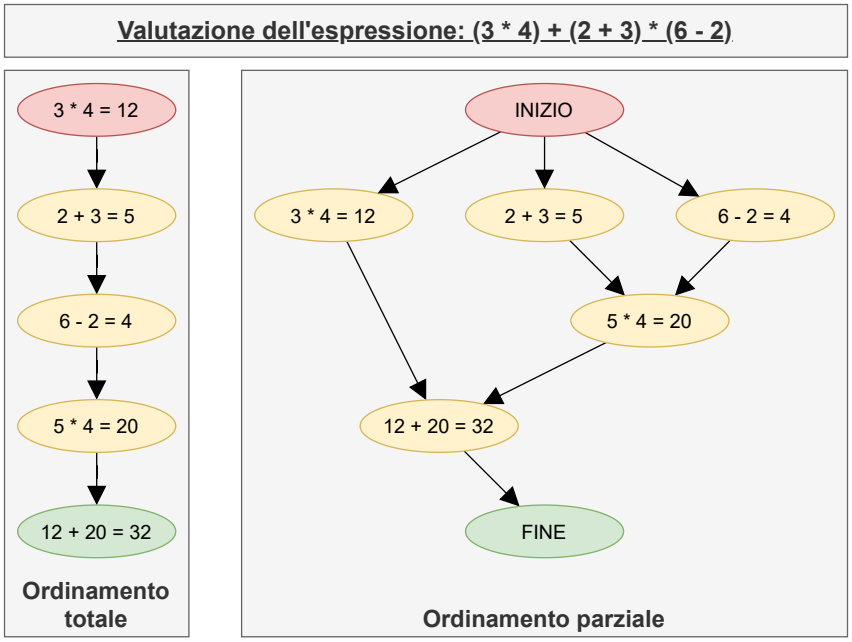


Figura 3.6: Grafi di precedenza della valutazione di un'espressione secondo l'ordinamento parziale e totale

Definizione 36: ***Ordinamento parziale***

A livello pratico, l'ordinamento degli eventi di un processo è parziale quando tra alcuni di essi non è stabilita una relazione d'ordine.

Ad esempio, nella figura 3.6, i tre eventi che seguono *INIZIO* nel grafo a destra non hanno alcuno vincolo temporale tra di loro.

Esempio: ***Elaborazione dei dati su un file***

Si supponga di dover processare un file secondo un ciclo in cui si eseguono, byte per byte, operazioni di lettura, elaborazione e scrittura come segue:

```
buffer B;

//N = numero dei byte del file
for(int i=0; i<=N; i++) {

     //Lettura del byte e inserimento nel buffer *****************
     lettura(B);              /* L */

     //Elaborazione del byte e sovrascrittura nel buffer *********
     elaborazione(B);         /* E */

     //Scrittura del byte elaborato sul file *********************
     scrittura(B);       /* S */
}
```

In questo caso, allora, lettura, elaborazione e scrittura si susseguono l'una dopo l'altra formando delle triplette sequenziali. Ma questa non è l'unica via perché, in realtà, i vincoli del problema sono solo due:

- *le operazioni di lettura sul file di input devono essere eseguite in rigida sequenza dato il metodo di accesso sequenziale del sistema operativo (e lo stesso vale per la scrittura e l'elaborazione;*
- *dopo l'acquisizione di un record, le operazioni da compiere su di esso sono rigidamente ordinate ovvero, dopo la sua acquisizione, dovrà essere elaborato e successivamente scritto.*

Non essendoci altre relazioni di precedenza, ad esempio, allora non esiste alcuna relazione logica di precedenza tra la lettura dell'i-esimo blocco e la scrittura dell'$(i-1)$-esimo. Questo significa che ciò che avviene nell'elaboratore mentre il processo esegue può essere descritto da un grafo ad ordinamento parziale.

La figura 3.7 mostra i due possibili grafi per l'esempio dell'elaborazione di un file. Ovviamente, il grafo ad ordinamento parziale consente il **parallelismo**: guardando la figura, è possibile trovare degli elementi che non sono vincolati da una relazione di precedenza temporale. Consideriamo, ad esempio, S_2 ed E_3: essi non sono legati da nessuna relazione di precedenza per cui, nella pratica, tutto funziona correttamente sia nel caso che E_3 venga eseguito prima di S_2 ma anche nel caso in cui S_2 venga eseguito prima di E_3 o, in ultimo, se l'ambiante di esecuzione lo consente, anche se S_2 ed E_3 avvenissero contemporaneamente.

Per poter sfruttare la possibilità di descrivere il procedimento risolutivo di un dato problema in modo non sequenziale, ovvero tramite un processo non sequenziale, è necessario disporre di un ***elaboratore non sequenziale*** *e di un* ***linguaggio di programmazione non sequenziale****.*

- **Elaboratore non sequenziale**
  Elaboratore in grado di mettere in pratica l'indipendenza temporale di due o più eventi che, quindi, possono essere eseguiti contemporaneamente. Questo si riscontra in tutti i sistemi multielaboratori e multi-paralleli, dove la non sequenzialità è

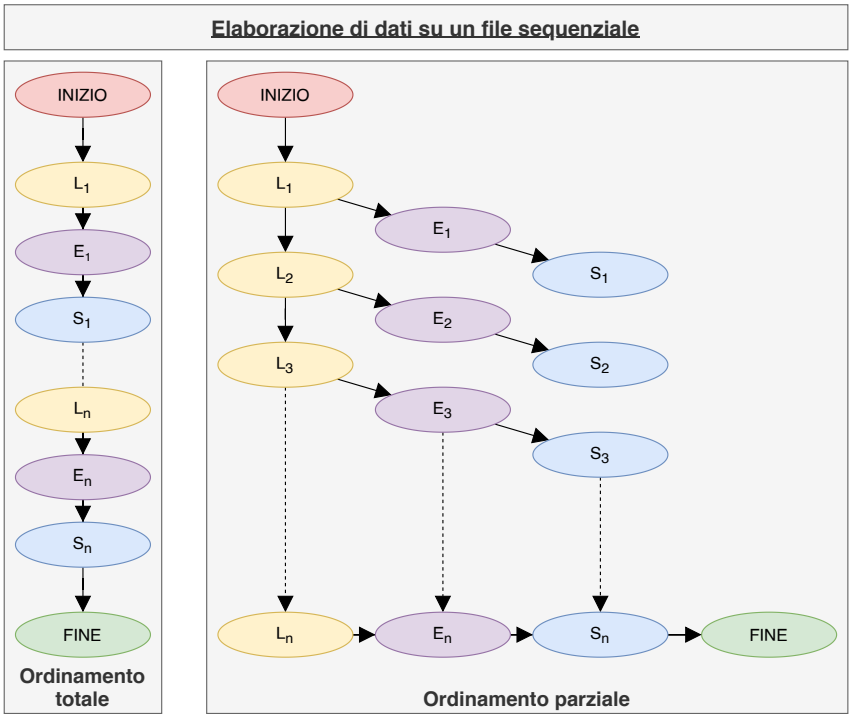


Figura 3.7: Grafi dell'esempio dell'elaborazione di un file

realizzata già a livello hardware; tuttavia, è possibile realizzare la non sequenzialità anche in sistemi che non dispongono di questa predisposizione hardware: la non sequenzialità, allora, viene implementata a livello software come nel caso dei sistemi mono-elaboratori ma multi-programmati. In questo caso, allora, il sistema operativo, attraverso i suoi meccanismi, costruisce, per chi lo usa, l'illusione di portare avanti più attività simultaneamente.

- **Linguaggio di programmazione non sequenziale (o concorrente)**
  Linguaggio di programmazione che sia in grado di descrivere grafi ad ordinamento parziale. In questo tipo di linguaggio deve esserci la possibilità di specificare, all'occorrenza, che due o più operazioni o insiemi di operazioni possano eventualmente essere eseguite contemporaneamente o nello stesso arco di tempo.

  Tali linguaggi, allora, mettono a disposizione del programmatore dei moduli che possono essere eseguiti in modo concorrente o parallelo e che, quindi, sono scorrelati tra di loro per quanto riguarda l'ordine temporale di esecuzione. Ogni singolo modulo ha una dimensione minima che dipende dal linguaggio (granularità):

  – singola istruzione (CSP, Occam);
  – sequenza di istruzioni (Java, Ada, Go, ecc. . .).

Scomposizione di un processo non sequenziale

Abbiamo visto che i linguaggi di programmazione non sequenziale mettono a disposizione dei moduli la cui granularità può essere la singola istruzione oppure sequenze di istruzioni. La situazione più comune, però, è il caso di linguaggi in cui la concorrenza può essere espressa a livello di sequenze di istruzioni e, dunque, per questo motivo è importante **scomporre un processo non sequenziale**.

Nella pratica, data la descrizione astratta di un processo non sequenziale (tipicamente un grafo ad ordinamento parziale), il programmatore allora deve scomporre il processo che viene descritto in un insieme di processi sequenziali, ognuno dei quali avrà la caratteristica di essere eseguito senza alcun vincolo di precedenza rispetto agli altri e contemporaneamente o concorrentemente ad esso, in base alle risorse hardware a disposizione.

Tale tecnica viene comunemente adottata per dominare la complessità di problemi molto complicati. Per quanto riguarda i processi derivanti dalla scomposizione, essi possono essere:

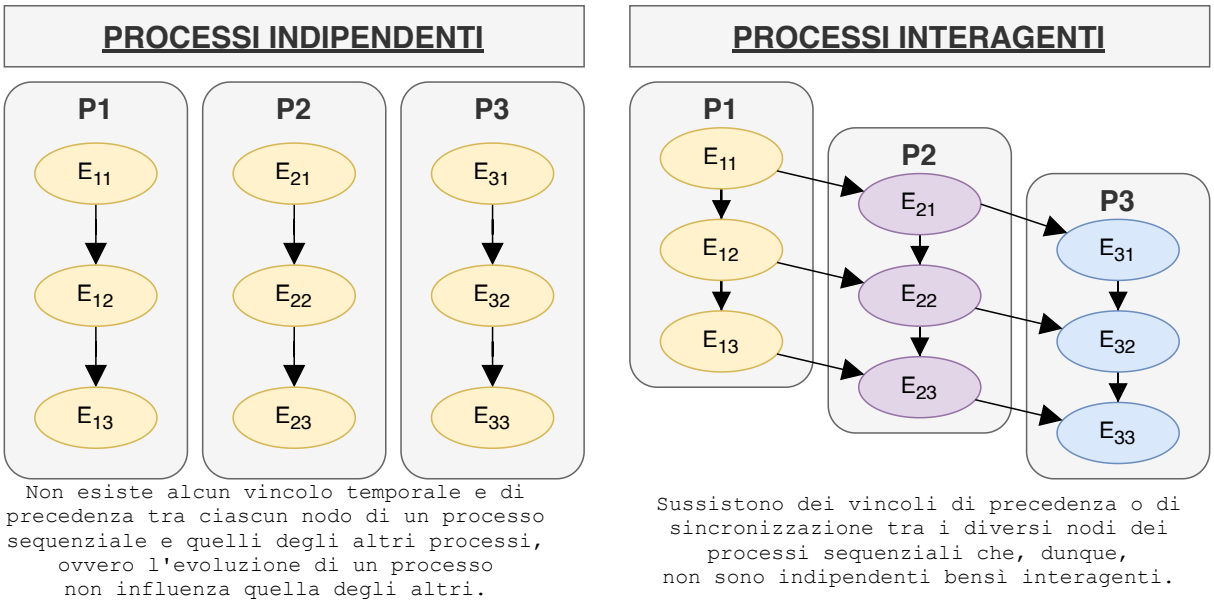


Figura 3.8: Differenza tra processi indipendenti e interagenti

Formalmente:

Definizione 37: ***Processi indipendenti***

Due o più processi sequenziali si dicono ***indipendenti*** *se l'evoluzione di uno dei due non influenza in nessun modo quella degli altri, ovvero se non sussistono vincoli di precedenza tra gli eventi dei processi considerati.*

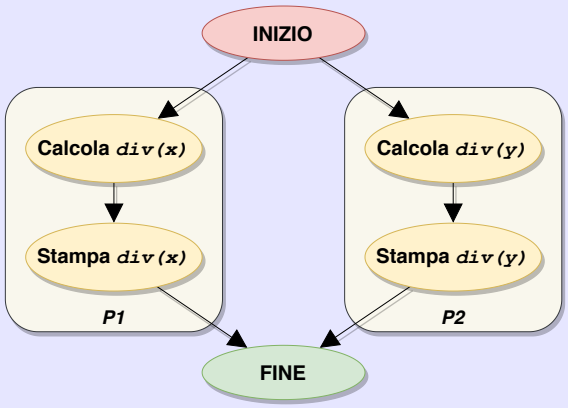
Definizione 38: ***Processi interagenti***

Due o più processi sequenziali di dicono ***interagenti*** *se sussistono dei vincoli di precedenza o di sincronizzazione tra le operazioni o gli eventi dei diversi processi considerati. Ogni vincolo di precedenza tra gli eventi dei processi, dunque, rappresenta un'interazione che dovrà essere espressa in modo idoneo a livello di linguaggio. Tra i processi che interagiscono, allora, deve esserci comunicazione e, dunque, uno scambio di informazioni.*

Esempio: ***Processi indipendenti: calcolo dei divisori di due interi***

Si supponga di voler calcolare l'insieme dei divisori (div) di due numeri interi x e y: l'obiettivo, allora, è stampare $div(x)$ e $div(y)$. Senza entrare nello specifico del-

l'algoritmo di calcolo, il problema si presta ad essere risolto utilizzando un processo non sequenziale, a sua volta scomposto in due processi sequenziali, $P1$ e $P2$, come descritto dal grafo che segue:



I due processi sequenziali sono ***indipendenti*** *tra di loro poiché ogni evento di $P1$ non è in alcun modo soggetto a vincoli di precedenza con quelli di $P2$ e viceversa.*

Esempio: ***Processi interagenti: elaborazione di un file***

La versione ad ordinamento parziale del grafo mostrato dalla figura 3.7 rappresenta un esempio di processi interagenti. Per un file che deve essere elaborato secondo le specifiche di quell'esempio, infatti, sono stati utilizzati tre processi, ognuno dei quali deve rispettare i vincoli di sincronizzazione con gli altri (es: non è possibile elaborare un byte prima che sia stato letto, ecc...).

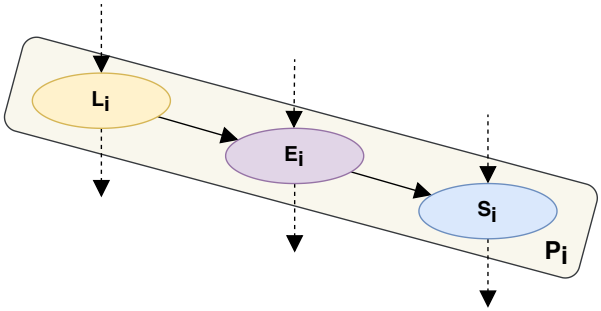


Figura 3.9: Decomposizione alternativa per l'esempio dell'elaborazione di un file

L'esempio relativo alla figura 3.7 mostra come la decomposizione di un processo non sequenziale possa essere fatta in modi diversi: sarebbe possibile utilizzare tre processi, uno per ogni tipo di operazione (lettura, elaborazione e scrittura, grafo a destra nella figura 3.7) oppure un processo per ogni record che, dunque, si occupi di leggerlo, elaborarlo e scriverlo (figura 3.9). Non c'è una ricetta già pronta per la scelta del modo in cui scomporre un problema, ma quella più idonea deve tenere conto del costo computazionale dovuto all'esecuzione di ogni singolo processo (costi di creazione e gestione dei processi) ma anche dei costi dell'interazione e del suo impatto complessivo sull'efficienza. Dunque, **nella scelta della decomposizione bisogna trovare una soluzione che sia un**

compromesso la minimizzazione dei costi di gestione dei processi e quelli della comunicazione.

3.1.5 Richiami sull'interazione tra processi

Stando a quanto detto sinora, abbiamo visto che è possibile ottimizzare la risoluzione di alcuni problemi scomponendo un processo non sequenziale in *sotto-processi* sequenziali che possono essere indipendenti tra loro o, come nel caso più comune, interagenti.

L'interazione, però, può avere tre possibili forme:

- **Cooperazione**
  Tipologia di interazione che fa parte della logica del programma e dell'algoritmo che si sta rappresentando e che permette di esprimere i vincoli di precedenza. Allora, dato il grafo che rappresenta la scomposizione del processo non sequenziale relativo al programma, tutti gli archi che collegano i distinti sotto-processi sequenziali sono previsti e insiti nella logica stessa degli algoritmi che compongono il programma.

  In generale, può esserci scambio di informazioni (sincronizzazione pura o comunicazione[3]).

  Inoltre, in interazioni di questo tipo, sussiste sempre una relazione di causa ed effetto tra l'esecuzione dell'operazione di invio da parte del processo mittente e l'esecuzione di quella di ricezione da parte del processo ricevente: se, ad esempio, ci si trova ad avere a che fare con due processi che interagiscono secondo *sincronizzazione pura*, allora l'invio di un segnale da parte del mittente permetterà al secondo di proseguire l'esecuzione; dunque, in questo caso, l'invio del segnale è stata la causa che ha avuto come effetto che il secondo processo, in attesa di sincronizzazione, potesse proseguire correttamente rispettando il vincolo di precedenza.

  Infine, affinché sia possibile realizzare interazioni cooperative, è necessario che il linguaggio di programmazione fornisca sia i costrutti di base per la realizzazione della concorrenza, sia quelli atti a specificare la sincronizzazione e la comunicazione tra i processi interagenti.

- **Competizione**

  Tipo di interazione che interviene quando, nell'ambiente all'interno del quale più processi eseguono, esistono delle risorse condivise sulle quali possono esserci dei vincoli[4]. Dunque, se esistono dei vincoli di accesso a tali risorse condivise, allora, ogni volta che un processo tenta l'accesso ad una di queste risorse, sarà necessario un meccanismo di sincronizzazione[5].

  La competizione, allora, è **prevedibile ma non è desiderata**, ovvero non fa parte dell'algoritmo che il processo sta eseguendo: è resa semplicemente **necessaria** da

[3] Si ricorda che la sincronizzazione prevede uno scambio di segnali temporali privi di dati mentre, invece, la comunicazione è realizzata mediante scambio di messaggi e, dunque, di dati.

[4] Il vincolo più presente, come si ricorderà, è quello di mutua esclusione il quale impedisce che la risorsa soggetta a quest'ultima venga acceduta contemporaneamente da più di un processo.

[5] Ad esempio, se un processo ha bisogno di accedere ad una risorsa mutuamente esclusiva che, però, è già in uso da qualcun altro, allora esso sarà costretto ad attendere indipendentemente anche da ciò che che l'algoritmo specifica.

dei vincoli che non dipendono dall'algoritmo e dal programma bensì dalla natura delle risorse alle quali i processi possono voler accedere durante l'esecuzione.

Si specifica, inoltre, che tra due o più processi che possono accedere ad una risorsa su cui insistono dei vincoli di accesso come la mutua esclusione, non è detto che si verifichi interazione: infatti, questo dipende soprattutto dagli istanti temporali in cui essi richiedono l'accesso alla risorsa condivisa[6].

Il concetto base della competizione è quello di **sezione critica**: sequenza di istruzioni con le quali un processo accede ad una risorsa condivisa con altri. Ovviamente, se vale il vincolo di mutua esclusione, è necessario garantire che sezioni critiche riferite alla stessa risorsa condivisa devono escludersi mutuamente nel tempo. Inoltre, il concetto di sezione critica è accompagnato da quello di **classi di sezioni critiche**: insieme di tutte le sezioni critiche riferite alla stessa risorsa condivisa.

In un ambiente mutuamente esclusivo, sezioni critiche appartenenti alla stessa classe devono necessariamente escludersi mutuamente nel tempo.

- **Interferenza**
  Come intuibile dal nome, questa forma di interazione ha un'accezione negativa e, infatti, **essa non è né desiderata né prevista**. Tipicamente, l'interferenza si verifica quando si commettono degli errori nella progettazione dell'applicazione concorrente; un esempio classico di interferenza è il *deadlock*[7].

3.1.6 Macchina concorrente

Definizione 39: ***Macchina concorrente***

Definiamo ***macchina concorrente*** *quella a cui fa riferimento un linguaggio di programmazione concorrente, ovvero una macchina in grado di eseguire più processi sequenziali contemporaneamente. Si specifica che non è detto che tale macchina sia completamente hardware bensì potrebbe essere realizzata parzialmente in hardware e parzialmente in software.*

Si supponga di disporre di una macchina concorrente e di un linguaggio di programmazione mediante il quale è possibile descrivere applicazioni concorrenti, e quindi algoritmi non sequenziali, che possono poi essere eseguiti su quella macchina. Allora, l'elaborazione complessiva di un programma scritto con tale linguaggio e che esegue su quella macchina può essere descritta come un insieme di processi sequenziali interagenti.

Un linguaggio di programmazione concorrente, inoltre, deve avere delle caratteristiche specifiche:

[6] Si considerino due processi P e Q che utilizzano una stampante condivisa per stampare un messaggio. Si supponga che P richieda l'uso della stampante per prima:

- se, durante l'uso della stampante da parte di P, Q ne fa richiesta, allora necessariamente ci sarà competizione e Q dovrà attendere il completamente dell'utilizzo della stampante da parte di P;
- se P termina di usare la stampante e Q la richiede in un istante di tempo successivo al rilascio da parte di P, allora non si crea nessuna situazione di interazione e non si verifica competizione.

La situazione è la stessa ma a parti opposte se il primo ad arrivare è Q.

[7] Si ricorda che il *deadlock* è una situazione di stallo in cui i processi di uno stesso gruppo sono tutti in attesa di un evento che può essere causato soltanto da uno di loro stessi.

- deve contenere apposite istruzioni e costrutti linguistici con i quali sia possibile denotare le parti di programma che possono essere eseguiti come processi sequenziali distinti (ad esempio la classe `Thread` di Java e il relativo metodo `run()` in cui viene specificato il codice del processo sequenziale che sarà istanziato);
- deve fornire opportuni strumenti che permettano di specificare quando i processi devono essere creati, messi in esecuzione e, dunque, terminati; tali strumenti devono poter essere invocati in modo dinamico (ad esempio, in Java si utilizza la `new` per istanziare un oggetto della classe `Thread` e il metodo `start` per attivare un processo);
- deve offrire degli strumenti linguistici che consente di specificare le interazioni che potrebbero verificarsi durante l'esecuzione tra i vari processi concorrenti.

Per quanto riguarda, invece, la macchina concorrente, quella a cui si fa riferimento è una macchina astratta che offre il linguaggio di programmazione e l'ambiente di esecuzione per i programmi concorrenti. Sostanzialmente, il programmatore specificherà i propri programmi nel linguaggio concorrente L e poi, tramite l'apposito compilatore, si avranno dei programmi che saranno in grado di sfruttare opportunamente le caratteristiche della macchina concorrente. Tale macchina astratta, che chiameremo M, allora:

- fornisce un certo numero di unità di elaborazione (anche virtuali), una per ogni processo che è in grado di supportare contemporaneamente;
- si basa su una macchina fisica, M', che generalmente è più semplice in quanto offre un numero di unità di elaborazione generalmente minore del numero dei processi per i quali M offre possibilità di concorrenza;
- è virtuale nel senso che offre una vista del sistema che è diversa da quelle che sono le caratteristiche di M'.

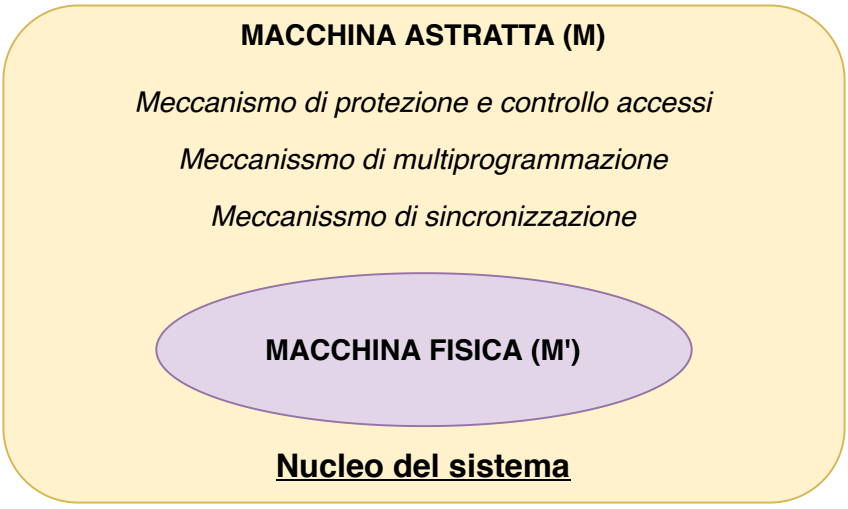


Figura 3.10: Architettura generale di una macchina astratta

La figura 3.10 mostra l'architettura generale della macchina concorrente astratta a cui ci si sta riferendo. Dunque, il nucleo (la parte gialla) corrisponde al supporto a runtime del compilatore di un linguaggio concorrente e, in esso, sono presenti due funzionalità base: il meccanismo di multiprogrammazione e quello di sincronizzazione e comunicazione.

Nello specifico:

- il **nucleo** più interno è rappresentato dalla macchina fisica, M', che ha le sue caratteristiche ed offre tutti i meccanismi presenti;
- la **macchina astratta** M, che si interfaccia con il linguaggio di programmazione concorrente, si colloca sopra a M';
- il **meccanismo di multiprogrammazione**, di cui è dotata M, da l'illusione ai programmi che si appoggiano alla macchina di avere a disposizione un numero di unità di elaborazione sufficientemente grande e indipendente da quelli fisici effettivamente presenti in M'; se CPU disponibili a livello fisico sono in numero inferiore al grado di concorrenza offerto dalla macchina astratta, allora questo meccanismo si basa fortemente sull'attività di scheduling[8];
- il **meccanismo di sincronizzazione** permette di realizzare l'interazione tra i processi indipendenti ed eventualmente è esteso con meccanismi di comunicazione;
- il **meccanismo di protezione** consente di regolare gli accessi nei confronti delle risorse condivise e può essere importante per rilevare ed evitare interferenze tra i processi.

Architettura di M

La macchina astratta M, come già detto, ricopre quella fisica nascondendone ai processi le reali caratteristiche; essa è caratterizzata da due diverse organizzazioni logiche:

1. gli elaboratori virtuali di M sono collegati ad un'**unica memoria principale**, ovvero c'è una condivisione di memoria per cui, di fatto, l'architettura della macchina ha caratteristiche simili a quelle dei sistemi multiprocessore per quanto riguarda la gestione della memoria;
2. gli elaboratori virtuali di M non condividono memoria bensì ognuno è attrezzato con una **propria memoria privata** e sono collegati tra loro mediante una rete di comunicazione similmente a quanto accade nei sistemi multicomputer; poiché ogni elaboratore ha una memoria privata, i processi, per interagire, non possono fare altro che utilizzare la rete di comunicazione;

Le organizzazioni logiche appena esposte definisco i due principali **modelli di interazione** tra i processi:

1. modello **a memoria comune**, in cui l'interazione tra processi avviene grazie ad oggetti contenuti nella memoria comune;
2. modello **a scambio di messaggi**, in cui la comunicazione e la sincronizzazione si basa sulla rete che collega i vari elaboratori.

Risulta chiaro che, d'ora in avanti, si darà per scontata l'indipendenza tra l'architettura della macchina che vedono i processi tramite il linguaggio di programmazione ed il suo supporto a runtime e quella della macchina fisica (M').

[8]Si ricorda che lo scheduling è l'attività di un sistema operativo che si occupa di allocare le risorse a disposizione e, dunque, le CPU ai processi in modo mutuamente esclusivo.

3.1.7 Costrutti linguistici per la specifica della concorrenza

Fork

Molto simile a quella di Unix, la *Fork* è un meccanismo che consente la creazione e l'attivazione di un processo. L'invocazione di una `fork` causa l'inizio dell'esecuzione di un nuovo processo che esegue concorrentemente a quello chiamante, in virtuale parallelismo.

```
/* PROCESSO P ************************ */

     //Istruzioni precedenti ...
A:   .......... ;
     p = fork fun;
B:   .......... ;
     //Istruzioni successive ...


/* NUOVO PROCESSO ******************** */

void fun() {
C:   .......... ;
     //Istruzioni successive ...
}
```

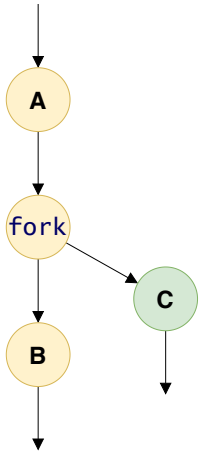


Figura 3.11: Modello a fork: esempio di codice e grafo di precedenza

Dunque, nella figura 3.11, a seguito dell'istruzione `p = fork fun;`, è stato creato un nuovo flusso di esecuzione, potenzialmente parallelo, che andrà ad eseguire la funzione `fun` che avrà tutta una serie di istruzioni che inizia da `C` e va avanti.

Join

La *Join*, similmente alla *wait* di Unix, è un'istruzione che consente di sincronizzare un processo che ha creato un altro processo, tramite una fork, con la terminazione di quest'ultimo.

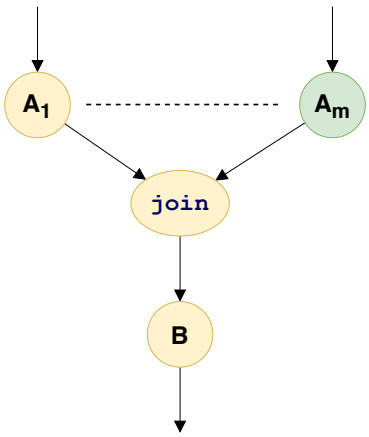


Figura 3.12: Esempio di grafo di precedenza del modello Join

Una volta creato un processo mediante `fork`, il chiamante può attendere la terminazione del figlio chiamando una `join`: risulta chiaro la join ha senso solo a seguito di una fork, per cui l'intero modello prende il nome di *Fork/Join*.

```
/* PROCESSO P ********************** */

      //Istruzioni precedenti ...
A:    .......... ;
      p = fork fun;
B:    .......... ;
      join p;
D:    .......... ;
      //Istruzioni successive ...



/* NUOVO PROCESSO ****************** */

void fun() {
C:    .......... ;
      //Istruzioni successive ...
}
```

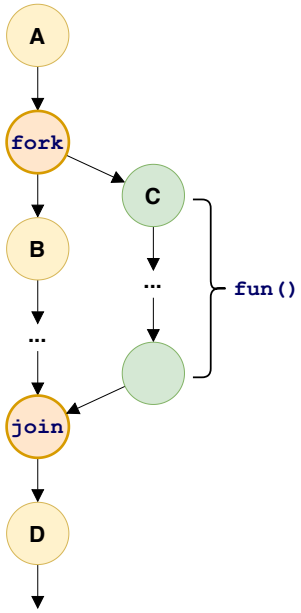


Figura 3.13: Modello Fork/Join: esempio di codice e grafo di precedenza

Il processo che ha chiamato la `fork`, come in figura 3.13, con la chiamata `join p`, si metterà in attesa della terminazione del processo precedentemente creato mediante l'istruzione `p = fork fun`, se necessario. Il *se necessario* è dovuto al fatto che è anche possibile che il nuovo processo termini prima che il padre arrivi alla chiamata `join`: sostanzialmente, però, con quest'ultima si ha comunque un punto di sincronizzazione in quanto, qualsiasi cosa accada, prima dell'esecuzione di `D` il processo originale ha verificato la terminazione del figlio. Nel grafo di precedenza in figura 3.13, allora, lo stato relativo alla `join` rappresenta un punto di sincronizzazione.

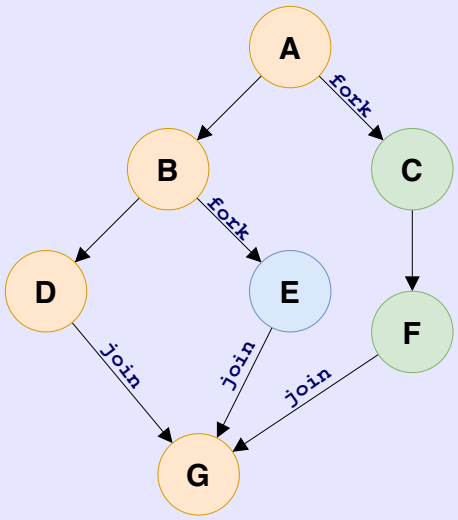
Esempio: ***Esempio di codice con uso di Fork/Join***

```
begin
      A;
      fork(LC);
      B;
      fork(LE);
      D;
      goto LG;

LC:   C;
      F;
      goto LG;

LE:   E;
      goto LG;

LG:   join(3);
      G;
end
```



In questo esempio, la `join` *utilizzata ha una sintassi diversa da quella usata precedentemente: in questo caso si impone sono che il punto di sincronizzazione riguarda tre processi, ovvero quello iniziale e quelli successivamente creati. Questa* `join`*, allora, permette di sincronizzare non solo il primo processo creato bensì anche il secondo.*

Cobegin-Coend

Il modello Fork/Join non è l'unico modello in grado di esprimere la concorrenza. Esiste, infatti, un modello più rigido che trae ispirazione dal modello che sta alla base dei linguaggi di programmazione strutturati e che, sostanzialmente, da la possibilità di istanziare, all'interno di un blocco particolare di istruzioni, un numero arbitrario di processi che procedono in parallelo: il modello Cobegin-Coend.

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

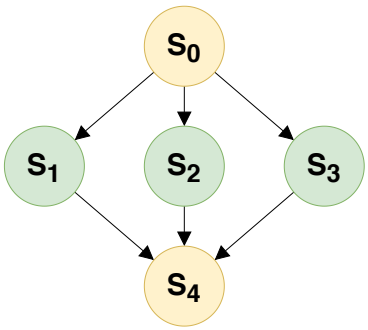


Figura 3.14: Modello Cobegin-Coend: esempio di codice e grafo di precedenza

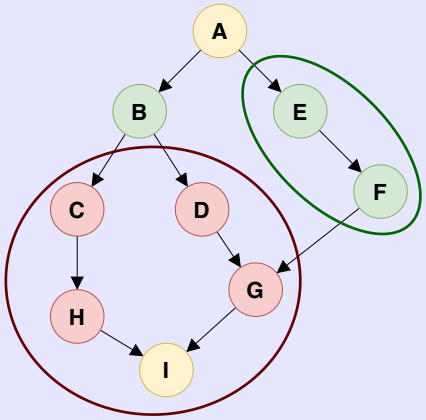
Le istruzioni comprese nel blocco `cobegin-coend`, e quindi S_1, S_2 ed S_3 vengono eseguite tutte e tre in modo potenzialmente parallelo: questo nuovo modello, a differenza della `fork`, può mettere in esecuzione un numero arbitrario di processi, come visibile nel

grafo di precedenza in figura 3.14. Nella keyword `coend`, inoltre, è insito un punto di sincronizzazione per cui prima di procedere con l'istruzione successiva, è necessario che tutti i processi concorrenti messi in esecuzione dalla `cobegin` siano effettivamente terminati. Inoltre, è possibile che dei processi messi in concorrenza dalla `cobegin`, contengano al loro interno altri blocchi Cobegin-Coend permettendo la costruzione di processi non sequenziali con una struttura molto più complessa.

Il modello Fork/Join è molto più generale rispetto a quello Cobegin-Coend nel senso che tutto quello che si può esprimere con quest'ultimo è sicuramente esprimile utilizzando il modello Fork/Join ma non è vero il viceversa. Ad esempio, con la Cobegin-Coend non sarebbe possibile imporre una sincronizzazione tra un processo interno al blocco ed un altro processo esterno.

Esempio: ***Grafo di precedenza non esprimibile con Cobegin-Coend***

La figura mostra un esempio di grafo non esprimibile mediante Cobegin-Coend ma solo tramite Fork/Join: infatti, non sarebbe possibile sincronizzare il processo interno al blocco `cobegin-coend` (cerchiato in rosso) con quello esterno (cerchiato in verde).



3.1.8 Proprietà dei programmi

Programmi concorrenti vs sequenziali

In generale, quando si sviluppa un programma, è fondamentale essere sufficientemente convinti questo sia corretto prima di rilasciare la versione finale. Dunque, la **verifica della correttezza** è molto importante specialmente se si sviluppa un software per la produzione.

Prima di tutto, bisogna ragionare sulla **traccia dell'esecuzione**, ovvero la sequenza di stati che saranno attraversati dal sistema durante l'esecuzione del programma. Ogni stato è caratterizzato da un insieme di valori delle variabili sia interne al programma ma anche implicite, che rappresentano la fase di esecuzione del programma come, ad esempio, il program counter.

È possibile fare due distinzioni principali tra i programmi in base a quello che accade a tempo di esecuzione:

Definizione 40: ***Programma sequenziale***

*Programma descritto da un grafo di precedenza con una struttura rigidamente lineare i cui stati legati tra loro da una relazione d'ordine totale. Dunque, ogni esecuzione di un programma sequenziale P_s su uno stesso insieme di dati D genererà sempre e comunque la stessa traccia (**determinismo**).*

Definizione 41: ***Programma concorrente***

*Programma in cui l'esito dell'esecuzione dipende da quale sia l'effettiva sequenza cronologica di esecuzione delle istruzioni in esso contenute: essi sono la descrizione di processi non sequenziali. Ogni esecuzione di un programma concorrente P_c su uno stesso insieme di dati D può dare origine ad una traccia diversa che dipende dai tempi in cui le diverse fasi del programma vengono eseguite (**non determinismo**).*

In linea di principio, il debugging, per un programma sequenziale, è più che sufficiente a verificare l'effettiva correttezza del programma. La stessa cosa, purtroppo, non è valida per i programmi concorrenti nei quali, invece, in linea di principio, l'esito dell'esecuzione dipende da quale sia stata l'effettiva sequenza cronologica delle sue istruzioni e, a causa di questo, il semplice debug non è sufficiente per la verifica della correttezza. Infatti, per la verificare completamente tale proprietà, sarebbe necessario fare un debug che tenga conto di tutti i possibili ordini di esecuzione delle varie parti concorrenti, il che è un lavoro troppo complicato.

Dunque, nel caso di programmi concorrenti, può essere molto utile verificare a priori, in modo statico, la validità di certe proprietà, e non a runtime come nel caso del debugging.

Definizione e classificazione delle proprietà

Definizione 42: ***Proprietà di un programma***

In generale, per un qualsiasi programma, una proprietà è un attributo che è sempre vero, ovvero una condizione che è sempre soddisfatta in ogni possibile traccia generata dal programma considerato.

Molto in generale, è possibile classificare le proprietà dei programma in due categorie principali:

- **Proprietà Safety**
  Tipo di proprietà che garantisce che durante qualunque esecuzione di un programma non si entrerà mai in uno stato errato, ovvero in uno stato in cui le variabili assumono un valore non desiderato. Per variabili, però, non si intende solo quelle proprie del programma bensì anche quelle di contesto che contribuiscono a definire lo stato di esecuzione (come il valore dei registri, ad esempio il program counter).

- **Proprietà Liveness**
  Tipo di proprietà che garantisce che durante l'esecuzione di un programma, prima o poi si entrerà in uno stato corretto: infatti, è possibile che durante l'esecuzione si transiti in stati in cui le variabili non hanno valori che sono desiderati ma, se vale una proprietà di questo tipo, c'è certezza che, prima o poi, si arriverà in uno stato in cui le variabili hanno i valori previsti.

Esempio: ***Proprietà fondamentali di un programma sequenziale***

Le proprietà fondamentali che deve avere ogni programma sequenziale sono:

- ***Correttezza del risultato finale (proprietà SAFETY)**: per qualunque esecuzione di quel programma, il risultato ottenuto è sempre giusto, ovvero soddisfa le specifiche del problema che il programma vuole risolvere;*

- ***Terminazione (proprietà LIVENESS)**: un programma gode di questa proprietà se c'è certezza che prima o poi termina.*

Esempio: ***Proprietà fondamentali di un programma concorrente***

Per quanto riguarda i programmi concorrenti, ricorrono anche in questo caso le proprietà di correttezza del risultato finale e di terminazione, che sono indispensabili, a cui, però, vengono aggiunte le seguenti tre proprietà che, appunto, dipendono dal non determinismo:

- ***Mutua esclusione nell'accesso a risorse condivise (proprietà SAFETY)**: se è previsto che i processi concorrenti accedano a risorse condivise, allora bisogna garantire che per ogni esecuzione non accadrà mai che più di un processo acceda contemporaneamente ad una stessa risorsa condivisa;*
- ***Assenza di deadlock (proprietà SAFETY)**: per ogni esecuzione del programma non si dovranno verificare mai situazioni di blocco critico;*
- ***Assenza di starvation**[a] **(proprietà LIVELESS)**: prima o poi, ogni processo può accedere alle risorse di cui ha bisogno; chiaramente potrebbe accadere che un processo attraversi degli stadi per cui non può accedere ad una risorsa di cui necessita, tuttavia questa proprietà garantisce che prima o poi questo vi avrà accesso.*

[a]Si ricorda che la starvation è un fenomeno per cui un processo, magari in una situazione in cui sono previste delle priorità, non può accedere ad una risorsa in un tempo finito poiché la politica di accesso a quella risorsa mette sempre avanti processi che hanno priorità su di lui.

Gli esempi appena mostrati saranno utili in futuro a capire alcuni discorsi a proposito delle proprietà.

Per concludere questo discorso sulle proprietà, è già stato evidenziato il fatto che la ripetizione sperimentale dell'esecuzione di un programma su vari insiemi di dati, in tempi diversi, in generale, non è sufficiente a dimostrare rigorosamente il soddisfacimento di proprietà per i programmi concorrenti. Un possibile approccio alternativo ma valido, che verrà approfondito in futuro, potrebbe essere specificare in modo formale la presenza di determinate proprietà per arrivare a dare delle vere e proprie **dimostrazioni di proprietà** ragionate a priori solamente sul sorgente.

3.2 Il modello a memoria comune

Il modello a scambio di messaggi, come accennato in precedenza, è uno dei modelli per l'interazione dei processi. Nell'ottica di questo modello, il sistema è visto come un insieme di:

- **processi** che portano avanti attivamente l'esecuzione;
- **oggetti**, che chiameremo risorse, e che possono essere utilizzati dai processi per portare avanti l'esecuzione, seguendo il vincolo di mutua esclusione.

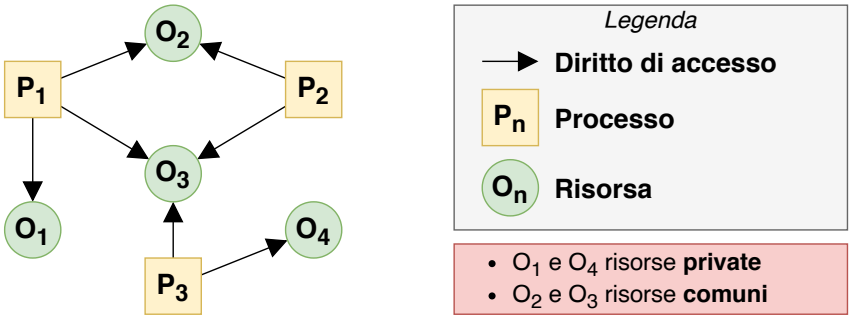


Figura 3.15: Esempio di grafo di allocazione delle risorse

Il grafico in figura 3.15 è un esempio di *grafo di allocazione delle risorse*, ovvero un grafo che mostra le risorse disponibili, i processi, e i loro diritti di accesso sulle prime.

In generale, nel modello a memoria comune, ogni interazione avviene mediante gli oggetti che sono collocati all'interno nella memoria comune che è condivisa tra tutti i processi.

Definizione 43: ***Risorsa***

Una risorsa è qualunque oggetto, logico[a] o fisico[b], di cui un processo necessita per portare a termine il suo compito.
Le risorse possono essere raggruppate in ***classi****, ognuna delle quali identifica l'insieme di tutte e sole le operazioni che un processo può eseguire su un'istanza, e dunque su una risorsa, di quella classe. Qualunque risorsa ha un corrispettivo che la rappresenta nella memoria comune.*

[a] Un esempio oggetto logico potrebbe essere una variabile oppure un file.
[b] Un oggetto fisico potrebbe essere, per esempio, un dispositivo.

3.2.1 Gestore di una risorsa

In un sistema concorrente, bisogna garantire che gli accessi dei processi nei confronti delle risorse avvengano in modo corretto, ovvero coerentemente con le politiche che sono state fissate nel contesto per l'accesso alle singole risorse. Tale compito è affidato ad un componente che prende il nome di **gestore di una risorsa** e che stabilisce, istante per istante, quali processi possano operare sulla risorsa riferita da quel gestore.

Definizione 44: ***Gestore di una risorsa***

Entità di cui è dotata ogni risorsa presente in un sistema concorrente che ha il compito di gestire l'insieme dei processi che operano sulla risorsa riferita, istante per istante.

Consideriamo, allora, una risorsa R e l'insieme $SR(t)$ dei processi che hanno il diritto di operare su R all'istante t. Tale risorsa R è:

- **dedicata** se $SR(t)$ ha una cardinalità sempre minore o uguale ad 1, ovvero se, ad ogni istante, è autorizzato uno e un solo processo ad accedere a quella risorsa;

- **condivisa** se non è dedicata, ovvero se la cardinalità d $SR(t)$, in ogni istante, può essere maggiore di 1 ovvero se, ad ogni istante, è possibile avere più processi che contemporaneamente necessitano di accedere a quella risorsa;
- **allocata staticamente** se il contenuto dell'insieme $SR(t)$ è costante, ovvero se non variano nel tempo gli elementi di quell'insieme, dunque $SR(t) = SR(t_0) \quad \forall t$;
- **allocata dinamicamente** se $SR(t)$ ha un contenuto che varia nel tempo, ovvero se per ogni t l'insieme dei processi contenuti in $SR(t)$ è, in generale, diverso.

Sulla base delle casistiche appena elencate, è possibile costruire la seguente tabella che mostra le tipologie di allocazione delle risorse:

	risorse dedicate	risorse condivise
risorse allocate staticamente	A *risorse private*	B *risorse comuni*
risorse allocate dinamicamente	C *risorse comuni*	D *risorse comuni*

Le colonne della tabella distinguono il caso di cardinalità 1 (risorsa dedicata, prima colonna) da quello di cardinalità maggiore di 1 (risorsa condivisa, seconda colonna); le due righe, invece, distinguono il caso di risorsa allocata staticamente da quello di risorsa allocata staticamente. Analizziamo, ora, i vari casi, ovvero le varie celle della tabella:

- (A)
  la risorsa è dedicata e allocata staticamente, dunque essa è privata, di uno e un solo processo: per tutta la durata dell'esecuzione, quella risorsa potrà essere accessibile ad un solo processo, per cui non sono necessari meccanismi di sincronizzazione.
- (C)
  la risorsa può essere acceduta da più processi diversi ma, ad ogni istante, potrà esserci uno e un solo processo che ha accesso alla risorsa senza possibilità di competizione.
- (B) e (D)
  in entrambi i casi, ad ogni istante, c'è la possibilità che più processi richiedano contemporaneamente l'accesso alla risorsa per cui, in questi casi, c'è la necessità di controllare opportunamente e regolare tali accessi in base alle politiche del programma. Dunque, questo è il caso della **memoria comune**.

Sulla base di quanto appena detto, è possibile fare le seguenti considerazioni:

- per ogni risorsa allocata staticamente, abbiamo detto che l'insieme $SR(t)$ è costante ed è noto a priori, quindi definito a tempo di compilazione, per cui il gestore della risorsa coincide con il programmatore che, in base alle regole e gli strumenti che il linguaggio mette a disposizione, stabilisce a priori quali processi possono vedere, e quindi operare, sulla risorsa data.

- per ogni risorsa allocata dinamicamente, l'insieme $SR(t)$ non è noto a tempo di compilazione e, quindi, il gestore è un entità che, a runtime, definisce quale sia, istante per istante, l'insieme dei processi che sono abilitati ad accedere alla risorsa. Il gestore, dunque, è mappato su un componente del programma.

Come si può capire, **il gestore è un componente fondamentale** in ambienti ad allocazione dinamica delle risorse, in quanto i suoi compiti sono:

- mantenere aggiornato l'insieme $SR(t)$;
- fornire i meccanismi che un processo può sfruttare per acquisire i diritti per operare su una risorsa entrando, quindi, a far parte dell'insieme $SR(t)$; analogamente, il gestore deve fornire anche il meccanismo di rilascio che permette di rilasciare il diritto quando non più necessario;
- implementare la strategia di allocazione della risorsa, a seconda dell'applicazione concorrente, definendo quando, a chi e per quanto tempo allocare una data risorsa.

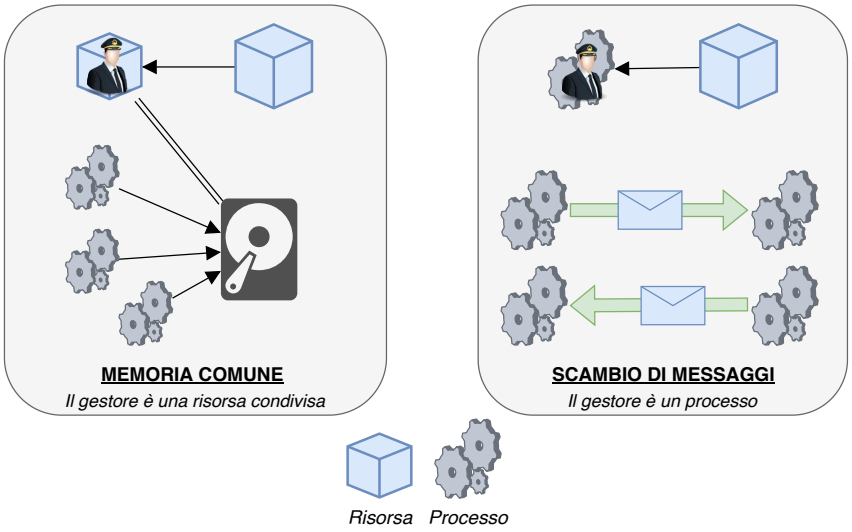


Figura 3.16: Gestore di una risorsa nei due modelli (memoria comune e scambio di messaggi)

Si specifica, inoltre, che data una risorsa R, il suo gestore G_R è costituito da una risorsa condivisa nel caso che il sistema sia organizzato secondo il modello a memoria comune[9] oppure da un processo[10], se invece vige il modello a scambio di messaggi come mostrato dalla figura 3.16.

Per questa sezione, si farà riferimento soltanto al modello a memoria comune.

Accesso a risorse

Consideriamo un processo P che, ad un certo istante, deve operare su una risorsa R in un sistema a memoria comune:

- se la risorsa è allocata **staticamente** a P, allora esso può accedervi solo se appartiene a $SR(t)$ e, in questo caso, può operare su R in qualsiasi istante. In particolare, se P vuole eseguire un operazione $\texttt{op}_i$ su R, il codice sarà il seguente:

[9] Si ricorda, ad esempio, il Monitor, ovvero la risorsa che gestisce gli accessi ad un'altra risorsa.

[10] I processi, per utilizzare una risorsa, devono rivolgersi ad un altro processo e, dunque, ad un'entità attiva.

```
//Esecuzione dell'operazione sulla risorsa
R.opi(...);
```

- se la risorsa è allocata **dinamicamente** a P, bisogna necessariamente fare riferimento al gestore G_R, che nel caso di memoria comune è un oggetto e, dunque, un'entità passiva, sul quale invocare in modo preliminare una richiesta di accesso e successivamente di rilascio di R. Il protocollo è il seguente:

```
//Richiesta di accesso alla risorsa
GR.richiesta(...);

//Esecuzione dell'operazione sulla risorsa
R.opi(...);

//Richiesta di rilascio della risorsa
GR.rilascio(...);
```

  Si specifica che la richiesta di accesso alla risorsa potrebbe comportare delle attese se, ad esempio, la risorsa è accessibile in modo mutuamente esclusivo e c'è già un altro processo che la sta utilizzando. Tuttavia, una volta che la chiamata `GR.richiesta(...)` ritorna, è implicito che la risorsa è allocata al processo che ne ha effettuato richiesta che, dunque, può eseguire una o più operazioni su R. Al termine della fase di utilizzo, il processo che ha utilizzato la risorsa si rivolgerà nuovamente al gestore per l'operazione di rilascio.

- se R è allocata come risorsa **condivisa**, è necessario assicurare che gli accessi avvengano in modo non divisibile e, dunque, le funzioni di accesso alla risorsa devono essere programmate come una classe di sezioni critiche, che la risorsa sia allocata dinamicamente o staticamente.

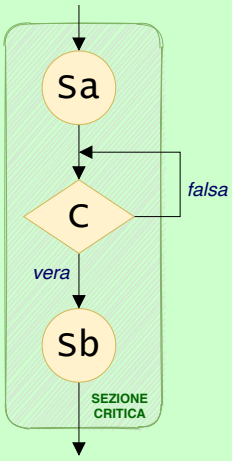
- se R p allocata come risorsa **dedicata**, non è necessaria alcuna forma di sincronizzazione o controllo degli accessi dato che P è l'unico processo a poter utilizzare la risorsa.

Per la specifica della sincronizzazione, dunque, utilizzeremo un formalismo molto noto detto **Regione critica condizionale**.

Definizione 45: ***Regione Critica Condizionale***

Formalismo per la specifica della sincronizzazione che consente di esprimere un qualunque vincolo di sincronizzazione nell'accesso ad una risorsa R condivisa. Dal punto di vista sintattico, il formalismo ha la seguente struttura:

```
region R « Sa; when(C) Sb; »
```



Si specifica che tale struttura non è da confondersi con una istruzione di un linguaggio di programmazione, bensì è un semplice formalismo per la descrizione di vincoli di sincronizzazione che possono essere applicati a determinate risorse.

In particolare, il vincolo che vale sulla risorsa R è espresso dal contenuto delle parentesi angolari (il corpo della region) che è da interpretarsi come una sezione critica:

- `Sa`, *istruzione (o insieme di istruzioni) che viene eseguita in modo incondizionato una volta ottenuto l'accesso alla risorsa;*
- `Sb`, *un'altra istruzione (o insieme di istruzioni) la cui esecuzione dipende dalla clausola* `when` *che la precede;*
- `when(C)`, *clausola che fa si che:*
  - *se* `C` *è vera viene eseguita* `Sb`*;*
  - *se* `C` *è falsa, il processo che sta eseguendo la* `region` *attende.*

  Generalmente, `C` *rappresenta lo stato, o una parte dello stato, in cui la risorsa R deve trovarsi affinché* `Sb` *possa essere eseguita.*

In sostanza, un formalismo di questo tipo da la possibilità al programmatore di esprimere alcun vincolo di sincronizzazione valido su una risorsa per la quale l'accesso deve avvenire in modo mutuamente esclusivo. Inoltre, se la condizione `C` è falsa, il processo attende sospendendo la regione critica in modo tale che la risorsa sia temporaneamente rilasciata per essere acceduta da qualcun altro che, magari, modificherà la condizione che era precedentemente falsa.

Questo, però, è uno strumento molto generale e, infatti, esistono diversi casi partiolari:

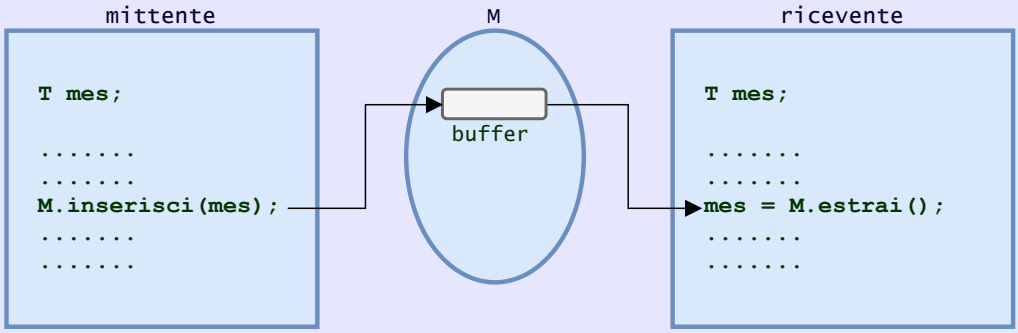
- `region R « S; »`
  modalità per esprimere una parte di codice in cui non si è prevista alcuna condizione di sincronizzazione ma che deve comunque essere eseguita in modo mutuamente esclusivo. In sostanza, `S` viene eseguita in mutua esclusione sulla risorsa R: se la risorsa è occupata, il processo che vuole eseguire la region attende, se, invece, è libera, e quindi non è attualmente utilizzata da nessun altro, allora il processo che esegue lo statement entra in modo mutuamente esclusivo nella risorsa per eseguire `R`.

- `region R « when(C); »`
  rispetto al caso generale mancano le due istruzioni, per cui questo caso particolare deve essere interpretato come un puro vincolo di sincronizzazione. Sostanzialmente, questo è un modo per imporre che un processo attenda il verificarsi di una certa condizione prima di proseguire la sua esecuzione; chiaramente, il fatto che la `when(C)` sia nelle parentesi angolari, impone che il controllo della condizione sia fatto rigorosamente in modo mutuamente esclusivo.

- `region R « when(C) S; »`
  molto probabilmente, questo è il caso più frequente che incontreremo nel corso; rispetto alla struttura originale manca solamente il primo statement. Questa modalità permette di specificare che l'istruzione (o la sequenza di istruzioni) `S` è condizionata al verificarsi di `C` che, dunque, è una precondizione per `S`.

Esempio: ***Esempio di sincronizzazione grazie alla Regione Critica Condizionale***

Si considerino un processo mittente, uno ricevente e un buffer sulla memoria condivisa come in figura:



In questo caso, il processo mittente deposita il messaggio da inviare in una variabile condivisa nella memoria comune, il buffer. Nella fase successiva, tale messaggio potrà essere estratto dal destinatario.
Ovviamente, uno schema di questo tipo deve essere necessariamente sincronizzato in modo che l'accesso al buffer avvenga nell'ordine corretto (il ricevente, prima di estrarre il messaggio, deve essere sicuro che il mittente lo abbia precedente depositato nel buffer).
Allora, `M` avrà la seguente struttura:

```
T buffer;
boolean pieno;

void inserisci(T dato) :
     region M << when(pieno == false)
                     buffer = dato;
                     pieno = true; >>

T estrai():
     region M << when(pieno == true)
                     pieno = false;
                     return buffer; >>
```

Dunque, `inserisci` *ed* `estrai` *sono due sezioni critiche che si riferiscono alla struttura costituita dalle due variabili condivise* `buffer` *e* `pieno`*: la prima funzione è eseguita dal mittente ogni volta che andrà a depositare un messaggio nel buffer, mentre la seconda sarà quella utilizzata dal destinatario per estrarre i messaggi. Inoltre, la possibilità di inserire un messaggio nel buffer è dettata dalla variabile* `pieno`*, che è una condizione di sincronizzazione*[a]*: la clausa* `when`*, infatti, è bloccante nel caso in cui la condizione sia falsa.*
Il discorso è speculare per l'operazione di estrazione. Ovviamente, le operazioni contenute nelle region, come da specifica stessa della Regione Critica Condizionale, sono tutte eseguite in modo mutuamente esclusivo.

[a]Se `pieno` è vera, allora il buffer non è vuoto e bisogna attendere che il messaggio già presente venga consumato prima di poterne inserire uno nuovo.

3.2.2 Richiami sulla mutua esclusione

La Regione Critica Condizionale appena mostrata è uno strumento utile a descrivere alcuni strumenti di sincronizzazione che saranno illustrati in questo corso. Prima di passare a mostrare tali strumenti, è utile fare alcuni richiami su uno dei problemi cardine quando di parla di interazioni tra processi, quello della **mutua esclusione**.

Senza scendere nuovamente nei dettagli di questo problema, già analizzato nel corso di Sistemi Operativi T, ricordiamo che, la mutua esclusione:

- è un problema tipico del modello a memoria comune, ovvero di quando c'è possibilità che più processi possano accedere alla stessa variabile;
- fa si che le operazioni con le quali i processi accedono alle variabili comuni non si sovrappongano mai nel tempo, per cui un solo processo alla volta può accedere ad una stessa risorsa condivisa, che sia una variabile o una struttura dati più complessa;
- non impone vincoli di ordine con il quale le operazioni devono essere eseguite bensì, come già detto, l'unica condizione che impone è che mai più di un processo alla volta possa eseguire una sezione critica su una risorsa condivisa, ma senza specificare l'ordine di accesso dei processi.

Dunque, la regola di mutua esclusione stabilisce che:

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

oppure

Ad ogni istante può essere "in esecuzione"[11]
al più una sezione critica di ogni classe.

In generale, per specificare una sezione critica S che opera su una risorsa R, ci si attiene al protocollo che segue:

[11]Con l'espressione *"in esecuzione"* si intende una sezione critica iniziata ma non ancora terminata.

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

dove:

- il `prologo` è una sequenza di istruzioni che serve ad ottenere l'autorizzazione ad eseguire in modo esclusivo la sezione critica e, quindi, poiché si sta ipotizzando che S operi su R, un effetto del prologo è l'acquisizione in modo esclusivo della risorsa R, che dal termine del prologo è allocata esclusivamente al processo che deve eseguire S;
- l'`epilogo` è una sequenza di istruzioni con la quale si notifica il fatto che la sezione critica è terminata, con l'effetto di rilasciare la risorsa R che, dunque, sarà nuovamente disponibile per essere acceduta da nuovi processi concorrenti.

Inoltre, per imporre la regola di mutua esclusione, come si ricorderà dal corso della triennale, è possibile ricorrere a tre principali tipi di soluzione:

- **Algoritmiche**
  soluzioni che sono tipicamente realizzate da opportuni algoritmi che sfruttano solo e unicamente la possibilità di condivisione di variabili (ad esempio algoritmi di Dekker, Peterson, ecc...). Tutta queste soluzioni sono accomunate dal fatto che non assumono nessun tipo di supporto da parte del nucleo della macchina concorrente, se non la possibilità di avere processi concorrenti e variabili condivise. Lo svantaggio principale di questo tipo di soluzione è che l'attesa di un processo che trova una variabile condivisa già occupata da qualcun altro, viene realizzata attraverso dei cicli di **attesa attiva**[12] non essendo impiegato alcuno strumento del nucleo, l'unica soluzione è quella di creare dei loop, ovvero dei *busy-waiting* o *cicli di attesa attiva.*
- **Hardware-based**
  soluzioni che sfruttano opportuni meccanismi messi a disposizione dal processore (come la disabilitazione delle interruzioni o il *lock/unlock*). Dunque, in questi casi il supporto è realizzato direttamente in hardware; lo svantaggio principale, però, è che anche in questo caso, per modellare l'attesa da parte di un processo, non c'è altra scelta se non l'attesa attiva.
- **Strumenti Software**
  sono le soluzioni più importanti dal punto di vista di gestione delle risorse del sistema, e sono strumenti realizzati in software da parte del nucleo della macchina concorrente. Sfruttando questi strumenti, infatti, è possibile sospendere i processi in attesa di eseguire sezioni critiche in modo effettivo, senza attesa attiva e cambiando a tutti gli effetti lo stato del processo che trova la condizione non soddisfatta da *running* a *waiting.* In questo modo, il processo abbandona l'uso dello CPU mettendosi in attesa in code realizzate dal nucleo, fintanto che non si verifica la condizione che gli consente di proseguire la sua esecuzione.

[12] Si ricorda che l'attesa attiva è un grande svantaggio in quanto fa consumare tempo di CPU ad un processo che, da un punto di vista logico, non può proseguire la sua esecuzione in quanto la risorsa di cui questo necessita è occupata. Mentre attende, infatti, il processo rimane comunque in esecuzione a consumare risorse che potrebbero essere utilmente utilizzate da altri processi.

3.2.3 Il semaforo

Definizione di semaforo

Definizione 46: ***Semaforo: definizione semplificata***

Strumento di sincronizzazione realizzato dal nucleo di una macchina concorrente che ***consente di risolvere qualunque problema di sincronizzazione*** *nel modello a memoria comune. Esso è disponibile nelle librerie standard per la realizzazione dei programmi concorrenti con linguaggi sequenziali come, ad esempio,* `C` *e Java.*

Dato che il semaforo è realizzato dal nucleo della macchina concorrente, l'eventuale attesa dei processi nell'esecuzione può essere realizzata utilizzando i meccanismi di gestione dei thread, in particolare quelli di sospensione e riattivazione, evitando l'attesa attiva. Il semaforo, inoltre, **è uno strumento di base che consente di realizzarne altri di più alto livello**, come, ad esempio, *Monitor* e *Condition*.

Definizione 47: ***Semaforo: definizione rigorosa***

Un semaforo è una ***variabile intera non negativa*** *alla quale è possibile accedere solamente* ***tramite due operazioni*** *che chiameremo* `P` *e* `V`.

Tipicamente, se in un linguaggio è disponibile il semaforo, allora lo è anche un tipo di dato astratto attraverso il quale è possibile definire variabili che aderiscono alla definizione appena fornita. Generalmente, un oggetto di questo tipo è definito in modi analoghi a quello che segue:

```
    semaphore s = i;
    //con i valore intero e tale che i>=0.
```

Al tipo di dato `semaphore` sono, inoltre, associati:

- un **insieme di valori**: $\{X \mid X \in N\}$;
- un **insieme delle operazioni**: $\{\texttt{P}, \texttt{V}\}$.

Operazioni sui semafori: `P` e `V`

Dunque, bisogna definire quali sono le due operazioni `P` e `V`. Seguendo il modello a Regione Critica Condizionale, una possibile definizione delle due operazioni potrebbe essere la seguente:

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

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

/* vals rappresenta il valore del semaforo */
```

Analizzando la specifica appena fornita, ricordando che i contenuti delle parentesi angolari sono delle sezioni critiche, si può dire che:

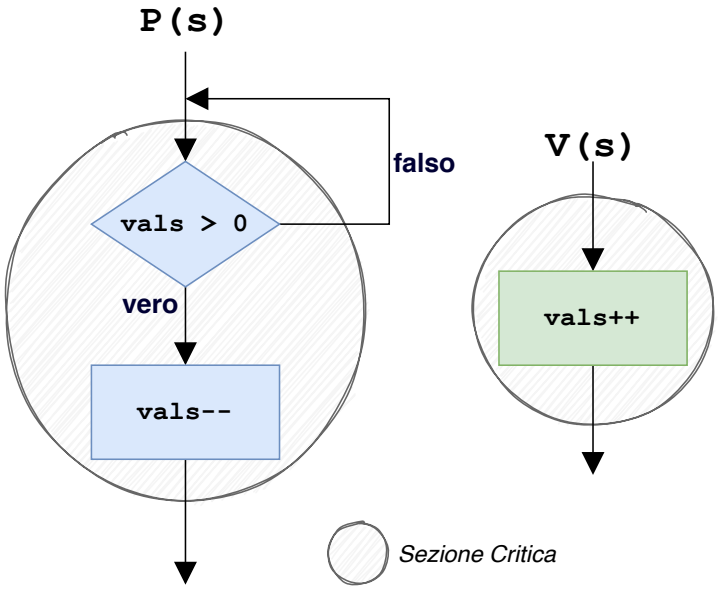


Figura 3.17: Diagramma di flusso delle operazioni `P` e `V`

- **Operazione** `P`
  in modo mutamente esclusivo, questa operazione accede al semaforo `s` valutandone il valore: se è strettamente positivo allora si può procedere e si decrementerà il valore di `s`. Se, però, il valore del semaforo è minore o uguale a zero, allora la clausola `when` mette il processo in attesa, liberando, ovviamente, il semaforo. Dunque, la `P` è bloccante quando **il semaforo è rosso**, ovvero se il valore del semaforo è 0.

- **Operazione** `V`
  in modo mutuamente esclusivo, questa operazione accede al semaforo `s` incrementandone di 1 il valore.

Si ricorda, inoltre, che i semafori sono variabili condivise utilizzate da più processi per sincronizzarsi: da qui il motivo per cui la `P` e la `V` sono implementate come **sezioni critiche eseguite in forma atomica**, in modo da evitare inconsistenze.

Relazione di invarianza

La relazione di invarianza è una delle proprietà fondamentali del semaforo. È una proprietà di tipo **safety** in quanto è sempre valida per qualsiasi semaforo, qualunque sia il suo valore e comunque sia strutturato il programma concorrente che lo utilizza.

Definizione 48: ***Relazione di invarianza***

Dato un semaforo `s`*, siano:*

- val_s*: valore dell'intero non negativo associato al semaforo;*
- i_s*: valore intero strettamente positivo con cui il semaforo* `s` *viene inizializzato;*
- nv_s*: numero di volte che l'operazione* `V(s)` *è stata eseguita;*
- np_s*: numero di volte che l'operazione* `P(s)` *è stata eseguita.*

Allora, è possibile esprimere il valore del semaforo come:

$$val_s = i_s + nv_s - np_s$$

Essendo, però, $val_s \geq 0$*, è possibile concludere che:*

$$np_s \leq i_s + nv_s \tag{3.1}$$

***L'equazione 3.1 esprime la relazione di invarianza**.*

Questa proprietà è particolarmente utile per dimostrare formalmente le proprietà concorrenti dei semafori.

Casi d'uso del semaforo

Come già anticipato, **il semaforo è uno strumento assolutamente generale** che consente di risolvere qualunque problema di sincronizzazione: esistono, dunque, una serie di casi d'uso più o meno complicati. Verrano, dunque, presentati tutti i casi d'uso del semaforo.

3.2.4 Il semaforo di mutua esclusione

Questo tipo di semaforo è il caso d'uso che dovrebbe essere più familiare e che **serve a garantire che sezioni critiche di una stessa classe vengano eseguite in modo mutuamente esclusivo**. Per capire il funzionamento di questo tipo di semaforo, analizziamo il protocollo da seguire per realizzare le sezioni critiche di una classe, che chiameremo `tipo_risorsa`, e che definisce una risorsa condivisa:

```
1  class tipo_risorsa {
2
3       <struttura dati di ogni istanza della classe>
4       semaphore mutex = 1;
5
6       public void op1() {
7            P(mutex); //Prologo
8            <sezione critica: corpo del metodo op1>
9            V(mutex); //Epilogo
10      }
11
12      ...
13
14      public void opN() {
15           P(mutex);
16           <sezione critica: corpo del metodo opN>
```

```
17            V(mutex);
18        }
19    }
```

Si specifica subito che **un semaforo di mutua esclusione è sempre inizializzato ad 1** ed è utilizzato per tutte le sezioni critiche contenute nella classe. Inoltre, la `P` del semaforo è sempre il prologo della sezione critica mentre la `V` è l'epilogo. Per utilizzare le operazioni della classe appena creata, invece, si può utilizzare il protocollo appena seguente:

```
tipo_risorsa ris;
ris.opi();
```

dove `opi` è un formalismo generico per indicare un metodo della classe `tipo_risorsa`. Risulta chiaro, per la definizione rigorosa di semaforo e per quanto appena detto sull'inizializzazione di quello di mutua esclusione, che quest'ultimo **può assumere solo i valori 0 e 1**: inoltre, quando il primo processo esegue la prima `P`, entrando in una sezione critica, il valore del semaforo scenderà a 0, per cui nessun altro processo potrà portare a termine un'altra `P` fintanto che il primo non abbia concluso la `V`.

Dimostrazione di correttezza

La correttezza della soluzione appena proposta è verificabile sperimentalmente, tuttavia è possibile dimostrarlo anche formalmente. Riducendo il protocollo all'osso, si ottiene:

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

Dunque, si vuole dimostrare che **un programma che rispetta questo protocollo soddisfi tutte le condizioni necessarie per la mutua esclusione**, ovvero che:

1. sezioni critiche della stessa classe devono essere eseguite in modo mutuamente esclusivo (definizione di mutua esclusione);
2. non siano possibili situazioni di stallo (deadlock);
3. quando un processo si trova all'esterno di una sezione critica, non può impedire l'accesso alla stessa sezione (o a sezioni della stessa classe) ad altri processi.

Verifichiamo, allora, le tre proprietà appena elencate:

- **Dimostrazione proprietà 1:**

Tesi:
Il numero dei processi nella sezione critica è sempre ≤ 1.

Si indichi con $N_{sez}(t)$ il numero di processi che si trovano, all'istante t, all'interno della sezione critica: allora, la tesi è dimostrata se vale sempre $0 \leq N_{sez}(t) \leq 1, \forall t$.

Chiamiamo $n_P(t)$ e $n_V(t)$ rispettivamente il numero di `P` e il numero di `V` completate all'istante t. Allora, dato anche il protocollo, si avrà che $N_{sez}(t) = n_P(t) - n_V(t)$, numero che coincide con quelli dei processi che hanno completato la `P`, entrando nella sezione critica, ma non la `V`.

Ricordando la relazione invariante, per un semaforo di mutua esclusione, essendo $i_s = 1$, si ha che:

$$1 + n_V(t) - n_P(t) \geq 0 \quad \Rightarrow \quad n_P(t) - n_V(t) \leq 1 \quad \Rightarrow \quad N_{sez}(t) \leq 1 \tag{3.2}$$

Inoltre, poiché il protocollo impone sempre che un processo, prima di entrare nella `V`, sia necessariamente passato per la `P`, seguirà che

$$n_P(t) \geq n_V(t) \quad \Rightarrow \quad n_P(t) - n_V(t) \geq 0 \quad \Rightarrow \quad N_{sez}(t) \geq 0 \tag{3.3}$$

Unendo la 3.3 e la 3.2, si conclude che

$$0 \leq N_{sez}(t) \leq 1, \quad \forall t \tag{3.4}$$

Corollario:
Il semaforo può assumere solo i due valori $\{0, 1\}$.

Per la definizione di semaforo, il suo valore è sempre un intero non negativo. Dunque, per dimostrare il corollario è sufficiente dimostrare che

$$val_s(t) = 1 + n_V(t) - n_P(t) \leq 1, \quad \forall t$$

La dimostrazione è immediata in quanto, essendo sempre $n_P(t) \geq n_V(t)$, si conclude:

$$n_P(t) - n_V(t) \geq 0 \quad \Rightarrow \quad n_V(t) - n_P(t) \leq 0 \quad \Rightarrow \quad 1 + n_V(t) - n_P(t) \leq 1, \quad \forall t \tag{3.5}$$

- **Dimostrazione proprietà 2:**

Tesi:
assenza di deadlock

Quello che bisogna dimostrare è che, in qualunque esecuzione e qualunque sia il numero di processi, supponendo che, però, ognuno di essi sia coerente con il protocollo specificato, non è possibile che si verifichino situazioni di deadlock. Dunque, supponiamo che si verifichi una situazione di deadlock; allora:

1. tutti i processi sarebbero in attesa sulla `P`
$$\Rightarrow \quad val_s(t) = 0 \quad t \geq t_0; \tag{3.6}$$

2. nessun processo potrebbe eseguire la `V` per cui, nessuno di essi sarebbe nella sezione critica
$$\Rightarrow \quad N_{sez}(t) = n_P(t) - n_V(t) = 0, \quad t \geq t_0 \tag{3.7}$$
con t_0 istante in cui si verifica il deadlock.

Dalla relazione invariante, ricordando che per il semaforo di mutua esclusione $i_s = 1$, si ottiene:

$$val_s(t) = i_s - n_P(t) + n_V(t) = 1 - N_{sez}(t) \tag{3.8}$$

Segue, allora, che sostituendo la 3.6 e la 3.7 nella 3.8, si conclude che

$$val_s(t) = 1 - N_{sez}(t) \quad \Rightarrow \quad 0 = 1 - 0 \quad \Rightarrow \quad 0 = 1 \quad t \geq t_0$$

che è un assurdo!

- **Dimostrazione proprietà 3:**

Tesi:
un processo all'esterno della sezione critica non può impedire ad altri processi di entrare nella sezione critica

Considerando un certo istante $\widetilde{t}$, ragioniamo sul fatto che non ci sia nessuno nella sezione critica: dunque, ad un qualunque processo, deve essere possibile entrarvi senza ritardi. Ovviamente, se la sezione critica è libera, questo significa che

$$N_{sez}(\widetilde{t}) = n_P(\widetilde{t}) - n_V(\widetilde{t}) = 0 \tag{3.9}$$

Utilizzando la relazione invariante per un semaforo di mutua esclusione e sostituendo la 3.9, si ottiene

$$val_s(\widetilde{t}) = i_s - n_P(\widetilde{t}) + n_V(\widetilde{t}) \quad \Rightarrow \quad val_s(\widetilde{t}) = 1 \tag{3.10}$$

Da questo risultato ne consegue che la `P`, prologo di qualunque processo, non è bloccante nel caso in cui non ci siano altri processi nella sezione critica, per cui la proprietà è dimostrata.

Mutua esclusione tra gruppi di processi

Un **variante del problema della classica mutua esclusione** è quella della mutua esclusione tra gruppi di processi: in alcuni casi, dipendentemente dal tipo di risorsa e dal tipo di accesso che si vuole realizzare su di essa, può essere consentito a più processi di **eseguire contemporaneamente una data operazione sulla risorsa considerata garantendo, però, la mutua esclusione tra operazioni diverse**.

Dunque, data una certa risorsa `ris` e indicate con `op1`, `op2`, `...`, `opn` le operazioni che sono state previste per operare su `ris`, **si vuole garantire che più processi possano eseguire concorrentemente la stessa operazione** `opi` **ma non devono essere consentite esecuzioni contemporanee di operazioni diverse**.

Anche in questo caso, lo schema che sarà adottato sarà basato su **prologo** ed **epilogo**:

```
public void opi() {
     <prologo_i>
     <corpo della funzione opi>;
     <epilogo_i>
}
```

dove:

- `prologo_i` ha il ruolo di sospendere il processo che ha chiamato `opi` se su quella risorsa sono in esecuzione operazioni diverse da quella, ovvero se c'è già almeno un processo che sta eseguendo un'altra operazione `opj` con `j` $\neq$ `i`;
- `epilogo_i` ha il ruolo di liberare la mutua esclusione solo se il processo che lo esegue è l'unico ad utilizzare la risorsa e, dunque, è l'ultimo processo di un gruppo che hanno eseguito la stessa operazione `opi`

Per `opi` il vincolo di mutua esclusione è rilassato in quanto si è data la possibilità a più processi di eseguirla in contemporanea impedendo, però, l'esecuzione di altre operazioni sulla risorsa fintanto che le `opi` sono ancora in corso.

La soluzione, per garantire tutto ciò, si basa sull'**impiego di due semafori**:

- un semaforo `mutex` di mutua esclusione che serve a garantire la mutua esclusione tra operazioni diverse
- un semaforo `m_i` di mutua esclusione che serve solo per garantire che il prologo e l'epilogo di `opi` vengano eseguite in modo mutuamente esclusivo

A livello di codice, quanto detto si traduce nelle seguenti righe:

```
1  public void opi() {
2
3       /* PROLOGO ******************************/
4       P(m_i);
5       cont++;
6       if(cont==1) P(mutex);
7       V(m_i);
8
9       <corpo di opi>
10
11      /* EPILOGO ******************************/
12      P(m_i);
13      cont--;
14      if(cont==0) V(mutex);
15      V(m_i);
16
17 }
18
19 public void opj() {
20      P(mutex);
21      <corpo di opj>
22      V(mutex);
23 }
```

Esempio: ***Problema dei lettori/scrittori***

Si supponga di avere una risorsa condivisa F (ad esempio un file, ma potrebbe essere anche una struttura dati) che può essere acceduta da più thread concorrenti tramite due operazioni: una di ***lettura****, per ispezionare il contenuto della risorsa senza modificarlo, e l'altra di* ***scrittura****, per modificare ed aggiornare il contenuto*

di F.
Ovviamente, è esagerato imporre il vincolo di mutua esclusione su più processi che vogliono solamente leggere quella risorsa: ***la lettura, quindi, potrebbe essere implementata in modo tale da consentire a più processi di accedere****, tramite quest'operazione, contemporaneamente alla risorsa dato che sappiamo a priori che essa non ne modificherà il suo stato. Lo stesso, però, non si può dire per* ***la scrittura che, invece, deve essere consentita solo ad un processo alla volta****. Inoltre,* ***lettura e scrittura su F non possono avvenire contemporaneamente****.*
La soluzione, allora, può essere modellata utilizzando la mutua esclusione tra gruppi di processi. Traducendo in codice:

```
semaphore mutex = 1; semaphore ml = 1; int countl = 0;

public void lettura(...) {
      /* PROLOGO ******************************/
      P(ml); countl++;
      if(countl==1) P(mutex);
      V(ml);

      <lettura della risorsa>

      /* EPILOGO ******************************/
      P(ml); countl--;
      if(countl==0) V(mutex);
      V(ml);
}

public void scrittura(...) {
      P(mutex);
      <scrittura del file>
      V(mutex);
}
```

3.2.5 Il semaforo evento

Abbiamo visto, parlando degli algoritmi non sequenziali e della loro traduzione in una collezione di algoritmi sequenziali interagenti, che uno dei problemi è **rappresentare, nel modo corretto, i vincoli di precedenza**, ovvero gli archi che collegano un processo sequenziale ad un altro imponendo, appunto, tale vincolo. Questo, in generale, si può fare con i **semafori evento**.

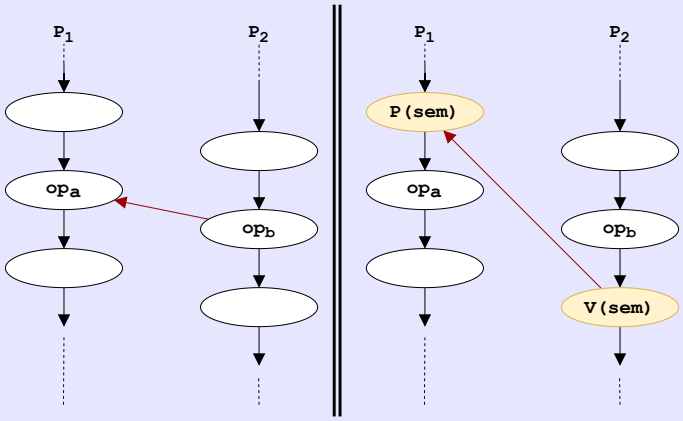
Definizione 49: ***Semaforo evento***

Il semaforo evento è un ***tipo di semaforo binario*** *, ovvero che può assumere solo i due valori 0 e 1,* ***che serve per imporre un vincolo di precedenza tra due operazioni*** *che si svolgono in processi concorrenti.*

Esempio: ***Vincolo di precedenza tra due operazioni di due processi concorrenti***

Supponiamo di avere due processi concorrenti, P_1 e P_2, che devono eseguire due operazioni op_a e op_b. Inoltre, P_1 deve eseguire op_a solo dopo che P_2 abbia eseguito op_b: dunque, si vuole stabilire un vincolo di precedenza che faccia in modo che op_a possa avvenire soltanto dopo che si è compiuta op_b.
Per risolvere il problema, allora, si introduce un semaforo, `sem`*, necessariamente inizializzato a 0 che verrà utilizzato come segue:*

- *il processo P_1 esegue una* `P(sem)` *prima di eseguire op_a;*
- *il processo P_2 esegue una* `V(sem)` *dopo aver completato op_b.*



Grazie al semaforo, infatti:

- *se P_1 arriva per primo in prossimità di op_a, prima che P_2 abbia eseguito op_b, allora P_1 farà la* `P` *sul semaforo ma, siccome questo è stato inizializzato a 0, la* `P` *sarà sospensiva e, dunque, P_1 sarà sospeso fintanto che P_2 non arrivi ad eseguire op_b e, subito dopo, la* `V` *che, dunque, avrà come effetto lo sblocco del processo che si era sospeso sulla* `P`*;*
- *se, prima che l'altro esegua la* `P`*, P_2 esegue op_b e quindi la* `V` *del semaforo allora, pur non sbloccando nessuno (l'altro processo non è ancora arrivato alla* `P`*), la* `V` *incrementa il valore del semaforo in modo tale che quando, in un secondo momento, P_1 eseguirà la* `P`*, allora quest'ultima sarà non sospensiva.*

Considerando l'esempio appena mostrato, possiamo provare a dimostrare formalmente che la soluzione adottata, basata sul semaforo evento, garantisce il rispetto del vincolo di precedenza. Il protocollo seguito è mostrato formalmente in figura 3.18.

- **Dimostrazione:**

Tesi:
nel caso di due processi interagenti P_1 e P_2, strutturati secondo il protocollo appena visto, op_a viene sempre eseguita dopo op_b

Procediamo con una dimostrazione per assurdo: supponiamo che sia possibile che op_a venga eseguita prima di op_b. Allora, esisterà un istante $\tilde{t}$ in cui è già stata

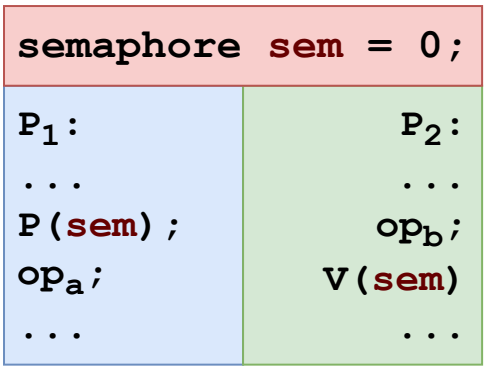


Figura 3.18: Protocollo adottato per il semaforo evento

eseguita la `P(sem)` ma non ancora la `V(sem)`, ovvero un istante in cui è stata eseguita op_a ma non op_b.

In tale istante, allora, si avrà:

$$n_{P(sem)}(\tilde{t}) > n_{V(sem)}(\tilde{t}) \quad \Rightarrow \quad n_{P(sem)}(\tilde{t}) - n_{V(sem)}(\tilde{t}) > 0 \tag{3.11}$$

Tuttavia, utilizzando la relazione invariante, si ottiene anche:

$$0 \geq n_{P(sem)}(\tilde{t}) - n_{V(sem)}(\tilde{t}) \tag{3.12}$$

Unendo la 3.11 e la 3.12, si conclude che:

$$0 \geq n_{P(sem)}(\tilde{t}) - n_{V(sem)}(\tilde{t}) > 0 \tag{3.13}$$

che è assurdo!

Il problema del Rendez-vous

Supponiamo di avere due processi P_1 e P_2: P_1 esegue e operazioni p_a e p_b mentre P_2 esegue q_a e q_b.
Allora, **il vincolo di rendez-vous** impone che l'esecuzione di p_b e quella di q_b, eseguite rispettivamente da P_1 e P_2, possano iniziare solamente dopo che entrambi i processi abbiano completato la loro prima operazione (p_a e q_a).

Dunque, il vincolo di rendez-vous impone che il processo P_2, prima di eseguire q_b debba attendere non solo che sia stata eseguita q_a, ma anche che sia stata eseguita p_a nell'altro processo. Ugualmente, il vincolo impone anche che P_1 possa eseguire p_b non solo dopo che sia stata eseguita p_a, ma deve attendere che sia completata anche q_a del processo P_2.

La soluzione prevede l'impiego di due semafori evento inizializzati a 0, come nella figura 3.19 che mostra il protocollo di soluzione al problema. Dunque, i semafori evento `sem1` e `sem2` sono utilizzati per realizzare una **coppia di vincoli di precedenza** che riguardano mutuamente i due processi. Dunque, nella soluzione appena proposta:

- p_b sicuramente avverrà dopo p_a, così come q_b succederà q_a a causa della natura sequenziale insita nei due processi a cui appartengono le operazioni;
- `sem1` permette di sincronizzare P_1 con P_2 in modo tale che l'esecuzione di p_b avvenga dopo che sia stata completata q_a;
- `sem2` permette di sincronizzare P_2 con P_1 in modo tale che l'esecuzione di q_b avvenga dopo che sia stata portata a termine p_a.

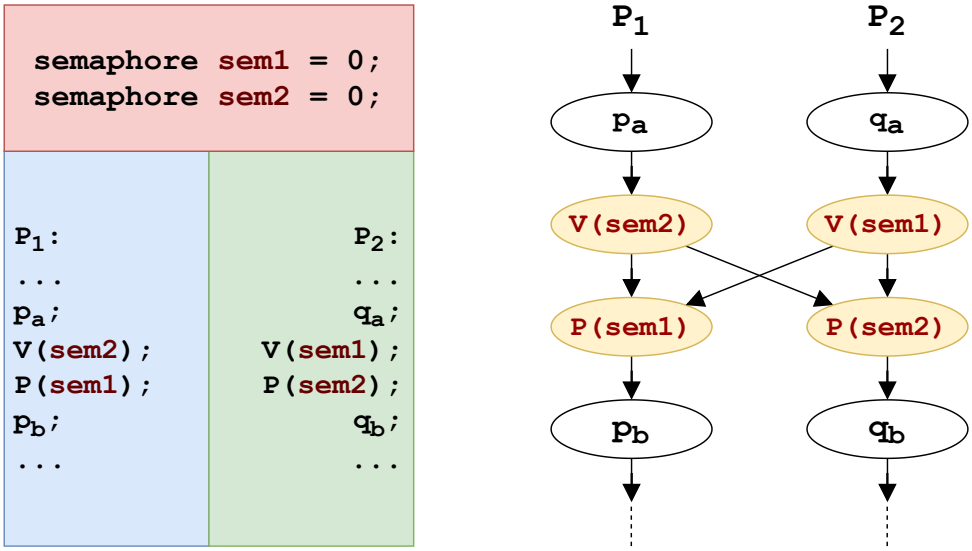


Figura 3.19: Protocollo e grafo della soluzione al problema del Rendez-Vous

Estensione del problema del Rendez-Vous a più processi

Consideriamo il problema del Rendez-Vous appena mostrato: cosa accadrebbe se i processi fossero N?

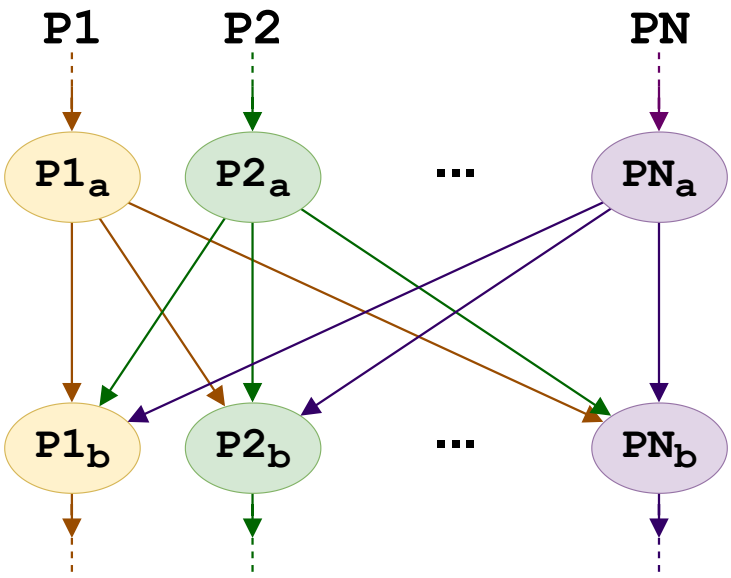


Figura 3.20: Grafo di precedenza del problema del Rendez-Vous esteso a N processi

La figura 3.20 mostra il grafo di precedenza del problema del Rendez-Vous esteso ad un numero N di processi. Infatti, in un ambiente concorrente, potrebbero esserci delle situazioni in cui è necessaria una *sincronizzazione collettiva*, che imponga che, ad un certo punto dell'esecuzione, **un'operazione di un processo possa essere eseguita soltanto dopo che tutti gli altri processi abbiano completato una certa fase**; nel problema del Rendez-Vous esteso, **questo vincolo vale per tutti i processi** presenti.

Il problema del Rendez-Vous, esteso a N processi, consiste nell'**imporre che una certa fase di esecuzione, in tutti i processi, possa partire soltanto dopo che quella precedente, sempre in tutti i processi, sia stata completata**.

Considerando sempre la figura 3.20, quanto detto si traduce nel fatto che, prima che un processo $Pi, i \in \{1 \dots N\}$ possa eseguire l'operazione Pi_b, devono essere completate tutte le operazioni $Pj_a, \forall j \in \{1 \dots N\}$.

La **prima idea** che potrebbe venire in mente è quella di **utilizzare** N **semafori temporali** da utilizzare similmente a quanto fatto nel caso a due processi: questo però, **impone un discreto carico** poiché ognuno di questi processi, a valle della sua operazione Pi_a, dovrà invocare $N-1$ `V`, una per ogni altro compagno e, analogamente, prima della sua Pi_b, dovrà fare altre $N-1$ `P` su tutti i semafori evento che lo riguardano. Ovviamente, **questa soluzione non è scalabile** poiché al crescere del numero dei processi aumenta linearmente il numero dei semafori e questo non è sostenibile, soprattutto in un contesto con un numero considerevole di processi.

La soluzione migliore da adottare è la **barriera di sincronizzazione**.

Definizione 50: ***Barriera di sincronizzazione***

Strumento (o schema) di sincronizzazione utile in ambienti concorrenti in cui deve valere il vincolo di Rendez-vous esteso a tutti i processi che lo popolano. Tale schema si basa sull'impiego di due semafori e di un contatore:

- *un semaforo di mutua esclusione inizializzato ad 1 (`mutex`);*
- *un semaforo inizializzato, però, a 0 (`barriera`);*
- *il contatore che tiene traccia del numero dei processi che hanno completato la prima fase (`completati`).*

In pseudocodice, la definizione delle variabili è la seguente:

```
semaphore mutex = 1;
semaphore barriera = 0;
int completati = 0;
```

Ogni processo, allora, sarà strutturato come segue:

```
<Operazione a di Pi>

P(mutex);
completati++;
if(completati==N)
     V(barriera);
V(mutex);
P(barriera);
V(barriera);

<Operazione b di Pi>
```

Dunque, il generico processo che ha terminato la propria operazione a, aggiorna il valore del contatore e ne testa il valore: se questo ha raggiunto il valore finale N, con N che è il numero totale dei processi concorrenti, allora quello è l'ultimo processo che ha compiuto la sua prima fase e, quindi esegue una `V` sul semaforo della barriera. Poiché il semaforo barriera era stato inizializzato a 0, i processi che hanno già terminato la prima fase sono tutti in attesa sulla `P(barriera)` per cui, quando l'ultimo eseguirà la `V(barriera)`,

permetterà agli altri, uno ad uno, di uscire dalla barriera. Questo meccanismo prende il nome di **schema a tornello** in quanto, dopo che il primo processo avrà eseguito la `V` su barriera, ne sbloccherà uno che era sospeso sulla `P` che, a sua volta, ne sveglierà un altro con la `V` subito successiva: dunque, si avrà una catena per cui, uno ad uno, tutti i processi saranno svegliati, similmente a quanto avviene con i tornelli.

3.2.6 Il semaforo binario composto

Consideriamo due processi P_1 e P_2 che si scambiano dati di tipo `T` utilizzando una memoria condivisa (*buffer*); supponiamo che P_1 assuma il ruolo di mittente mentre P_2 quello di destinatario. Chiaramente, bisogna garantire che:

- l'**accesso al buffer** avvenga in modo **mutuamente esclusivo**;
- il processo destinatario P_2 **può prelevare un messaggio soltanto dopo che** P_1 **lo abbia inserito**;
- il processo mittente P_1 **può inserire il proprio messaggio soltanto quando il buffer è vuoto**, ovvero quando il messaggio precedente è stato estratto.

Questo problema è quello che, in altri contesti, abbiamo chiamato *problema del produttore-consumatore*. Il modo migliore per risolvere problemi di questo tipo è utilizzare dei **Semafori binari composti**.

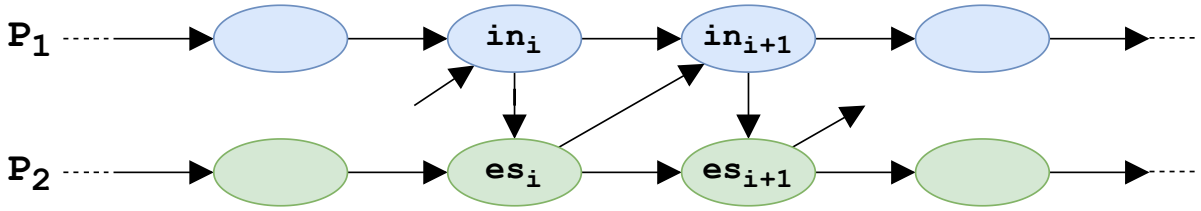


Figura 3.21: Grafo di precedenza di esempio del problema del produttore-consumatore

Dunque, considerando la figura 3.21, i vincoli di precedenza appena espressi, considerando il messaggio i-esimo, possono essere espressi come segue:

- es_i, ovvero l'estrazione del messaggio i-esimo da parte di P_2 può avvenire solo dopo che è stata completata in_i che è l'operazione di inserimento del messaggio eseguita parte di P_1;
- in_{i+1} può partire solamente se P_2 ha completato es_i.

Per imporre tali vincoli, allora, **necessari due semafori**:

- `vu`, per realizzare l'attesa di P_1 nel caso il buffer sia pieno;
- `pn`, per realizzare l'attesa di P_2 nel caso il buffer sia vuoto.

La figura 3.22 mostra il protocollo per la soluzione del problema con l'impiego dei due semafori appena illustrati. **L'ipotesi iniziale è che il buffer sia vuoto**, motivo per cui `vu` sarà inizializzato ad 1 mentre `pn` a 0. Dunque:

```
semaphore vu = 1;
semaphore pn = 0;
```

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


P1:

while(true) {
       <prepara msg>
       invio(msg);
}
```

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

P2:

while(true) {
       M = ricezione();
       <consuma M>
}
```

Figura 3.22: Protocollo di soluzione per il problema del produttore-consumatore

- ogni inserimento eseguito da P_1 dovrà essere preceduto da una `P(vu)` che, dunque, **costringerà il processo ad attendere che il buffer sia vuoto** prima di inserire il nuovo dato;
- dopo aver inserito il dato, il mittente farà una `V(pn)` con la quale **notificherà al destinatario che è presente un messaggio nel buffer**;
- il destinatario, prima di poter estrarre il messaggio, deve sincronizzarsi con l'altro processo **in attesa che sia stato inserito un nuovo messaggio** all'interno del buffer, compito della `P(pn)`;
- dopo aver prelevato il messaggio, il destinatario farà una `V(vu)` che **servirà a notificare a P_1 che il buffer è vuoto**, con l'effetto di risvegliare quest'ultimo nel caso in cui sia in attesa sulla `P(vu)`.

La coppia di semafori utilizzata è detta **semafori binari composti** e, inoltre, **impedisce ai due processi di accedere contemporaneamente al buffer**, comportandosi, dunque, come un **semaforo di mutua esclusione**.

Definizione 51: ***Semaforo binario composto***

Insieme di due o più semafori utilizzato in modo tale che:

- ***uno solo di essi è inizializzato ad*** 1 *e tutti gli altri a* 0*;*
- *ogni processo che li utilizza esegue sempre sequenze di istruzioni che* ***iniziano con la*** `P` ***su uno di questi e terminano con la*** `V` ***su un altro****.*

3.2.7 Il semaforo condizione

In molti casi, c'è la necessità di condizionare l'esecuzione di una data istruzione al verificarsi di una condizione che, ad un certo momento può essere falsa, ma, magari, per effetto

dell'esecuzione di altri processi concorrenti potrebbe diventare vera. Questo caso è molto comune nel caso di programmi costituiti da una collezione di processi concorrenti e si può esprimere formalmente in questo modo:

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

Dunque, `op1()` **è una regione critica** che opera sulla risorsa condivisa `R` e, inoltre, `S1` **viene eseguita se e solo se è vera la condizione** `C` che, in questo caso, è una **pre-condizione**.

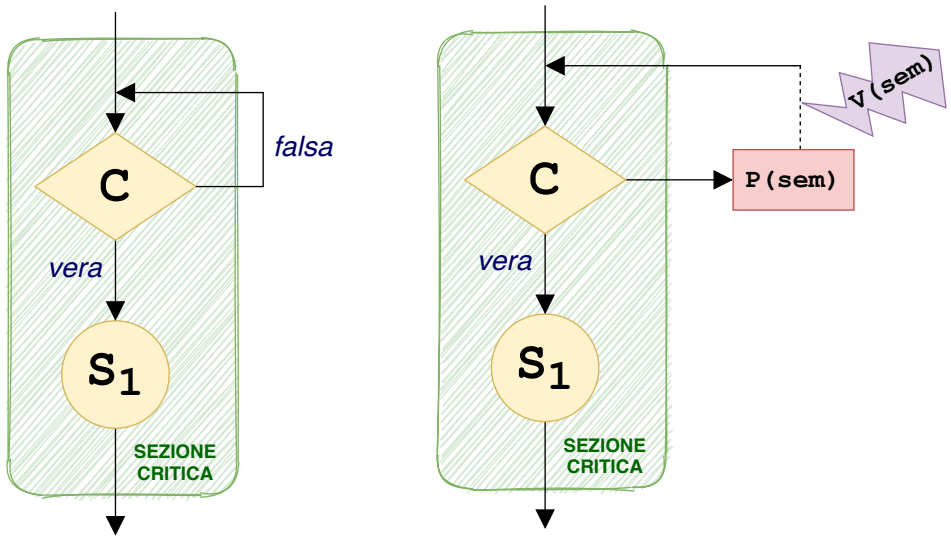


Figura 3.23: Pre-condizione: diagramma di flusso senza (sinistra) e con (destra) l'impiego di un semaforo

La figura 3.23 mostra due possibili soluzioni al problema della pre-condizione: a sinistra la soluzione senza l'impiego dei semafori, soggetta, quindi, ad attesa attiva, a destra quella con il semaforo e che, dunque, prevede la sospensione del processo.

Segue che `sem`, allora, è un **semaforo condizione** inizializzato a 0: se la condizione non è verificata, viene eseguita la `P(sem)` che, data l'inizializzazione del semaforo, è sospensiva per il processo che la esegue fintanto che qualcuno non avrà eseguito una `V(sem)`.

Chiaramente, tutto questo funziona se prima o poi qualcuno chiamerà questa `V(sem)`, per cui si ipotizza che **il programma concorrente preveda anche un'altra operazione**, che chiameremo `op2`, che, ad un certo punto, **aggiorni lo stato interno della risorsa**, provocando quindi il cambio della condizione `C`, che da falsa diventerà vera, **eseguendo, contestualmente, anche la** `V(sem)` che avrà come effetto quello di risvegliare il processo precedentemente sospeso.

Soluzione con attesa circolare

La figura 3.24 mostra un possibile schema di soluzione con **attesa circolare** in cui sono utilizzati due semafori: `mutex`, di mutua esclusione, per rendere mutuamente esclusive le fasi di accesso alle risorse condivise e `sem`, il semaforo condizione iniziato a 0 di default.

Inoltre, il processo che esegue l'operazione sospensiva di test della condizione, **esegue la verifica all'interno di una sezione critica e con un** `while` al posto di un `if`: se è falsa, non c'è dubbio che il processo debba attendere, andando dapprima ad incrementare

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

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

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

Figura 3.24: Schema di soluzione alla pre-condizione con attesa circolare

il contatore `csem` che, semplicemente, tiene traccia di quanti processi sono in attesa che la condizione sia verficata, e poi, **uscendo dalla sezione critica**[13], **si sospende sul semaforo condizione** con la `P(sem)`. Ovviamente, questo rimarrà sospeso fino a quando un altro processo, a ragion veduta, non chiamerà la `V(sem)` dopo aver cambiato il valore della condizione. Una volta che è stato svegliato, il processo deve nuovamente entrare in sezione critica prima di poter eseguire `S1`.

Si nota che, dopo aver eseguito `S2`, blocco di istruzioni che modificherà, in qualche modo, la risorsa condivisa, **il secondo processo non verifica se la condizione sia verificata o no**, ma comunque risveglia i processi in attesa su `sem`, se ce ne sono. Infatti, per com'è strutturata la soluzione, **sarà lo stesso processo risvegliato a ri-testare la condizione** nello stesso `while`: se la condizione è vera allora esce dal ciclo o, nel caso più sfortunato, dovrà rimettersi in attesa.

Infine, si sottolinea che **è molto importante che la `V` su `sem` venga fatta solo se ci sono processi in attesa**; infatti, se venisse ugualmente fatta, la `P` presente in `op1` non sarebbe sospensiva e questo, ovviamente, è un problema.

Soluzione con passaggio di testimone

La figura 3.25 mostra lo schema di soluzione con **passaggio di testimone**. Il tutto si basa ancora sulle stesse variabili condivise della soluzione con attesa circolare ma, questa volta, il processo che esegue `op1` testa la condizione con un `if`: se la condizione non è soddisfatta, il processo si sospende sulla `P(sem)` ma, in questo caso, subito dopo il risveglio, non c'è la `V(mutex)` e questo è dovuto alla nuova struttura di `op2`.

Nella soluzione corrente, infatti, in `op2` si entra nella sezione critica con la `P(mutex)`, viene eseguito lo statement `S2` che, potenzialmente, muta il contenuto della risorsa condivisa, dopodiché viene testato il valore della condizione `C` insieme con il valore del contatore `csem`: se `C` è diventata vera e c'è almeno un processo in attesa sul semaforo condizione, allora il processo che sta eseguendo `op2` chiama la `V(sem)` che sicuramente risveglierà il primo processo in attesa su quel semaforo per poi terminare **trasferendo il diritto di eseguire in modo mutuamente esclusivo sulla risorsa al processo risvegliato**. Allora, quest'ultimo non dovrà testare nuovamente la condizione, dato che questa è stata

[13] L'uscita dalla sezione critica è fondamentale per evitare situazioni di deadlock

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

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

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

Figura 3.25: Schema di soluzione alla pre-condizione con passaggio di testimone

già verificata da chi l'ha risvegliato, e non dovrà neanche riacquisire esplicitamente il diritto di accedere alla sezione critica in quanto questo gli è già stato ceduto dal risvegliante. Dunque, il processo appena svegliato non dovrà far altro che decrementare il contatore dei processi e proseguire con la sua esecuzione.

Attesa circolare vs Passaggio del testimone

Per quanto riguarda la scelta della soluzione, di sottolinea che **non è detto che il processo che esegue `op2` abbia la possibilità di testare la condizione** e questo perchè **`C` potrebbe dipendere non solo da variabili globali, ma anche da variabili locali interne ad `op1`**: in questo caso lo schema da impiegare è quello dell'attesa circolare. Inoltre, rimanendo sempre **nell'attesa circolare, anche se `op2` verificasse la condizione prima di risvegliare l'altro processo, non è detto che il risvegliato possa effettivamente riprendere il controllo** poiché la `P(mutex)` subito dopo potrebbe rimetterlo in attesa a potrebbe esserci qualcun altro che prima di lui potrebbe essere entrato nella sezione critica: dunque, non è possibile sapere a priori cosa possa aver fatto questo processo che, addirittura, potrebbe aver nuovamente cambiato il valore della condizione obbligando, dunque, il risvegliato a fare un nuovo controllo.

Per quanto riguarda **il passaggio del testimone**, sicuramente **è più efficiente** dell'attesa circolare perché nel primo schema il testing della condizione viene eseguito una sola volta dal processo che esegue `op1` mentre nel secondo è possibile che si debba reiterare più volte, generando cicli inutili.

Tuttavia, ci sono delle limitazioni rendono il passaggio del testimone adattabile solo a determinati casi:

- consente di **risvegliare solo un processo alla volta** per cui, se il problema da risolvere prevede la possibilità che, a fronte di aggiornamento, sia necessario risvegliarne più di uno, con il passaggio del testimone la cosa non è realizzabile mentre con l'attesa circolare è possibile impiegando un ciclo di `V(sem)`[14];

[14]Si consideri il caso di un problema in cui un processo può chiedere l'allocazione in memoria di un numero arbitrario di risorse. In `op1`, innanzitutto, la condizione verifica che ci siano le risorse di cui il processo ha bisogno: se non ci sono, allora questo si sospende. Supponiamo, allora, che ci siano due

- la condizione `C`, che deve necessariamente poter essere testata da chi risveglia nel caso del passaggio di testimone, non è detto che lo sia poiché potrebbe dipendere anche da variabili locali che sono fuori dalla visibilità del processo che esegue `op2`.

Esempio: ***Gestione di un pool di risorse equivalenti***

Consideriamo un pool di N risorse equivalenti, ovvero un insieme di risorse tutte funzionalmente identiche[a]*. Il compito di un* ***gestore di un pool di risorse equivalenti****, allora, è allocare, nel modo più efficiente possibile, le risorse dell'insieme ai processi che lo richiedono secondo il protocollo che segue:*

1. *quando ha la necessità di operare su una risorsa,* ***il processo richiede al gestore una qualunque delle risorse del pool****;*
2. *se non è disponibile nessuna risorsa, allora il gestore mette in attesa il processo; se, invece, ci sono una o più risorse libere allora* ***il gestore ne assegnerà una al processo****, trasferendogli le informazioni necessarie a poterla utilizzare (ad esempio un indice o un puntatore);*
3. ***il processo può operare liberamente sulla risorsa****, senza doversi preoccupare della mutua esclusione che è stata garantita a monte dal gestore;*
4. *al termine dell'utilizzo,* ***il processo si rivolge al gestore per il rilascio della risorsa****.*

Il gestore, allora, deve esibire due operazioni accessibili in modo mutuamente esclusivo[b]*:*

```
int richiesta():
     region G <<
          when( <test risorse disponibili> )
               <scelta di una risorsa disponibile>;
          int i = <indice della risorsa scelta>;
          <registrazione risorsa di indice i non disponibile>

          return i;
     >>

void rilascio(int r):
     region G <<
          <registrazione risorsa r-esima nuovamente disponibile>
     >>
```

Quello appena espresso è il protocollo di realizzazione del gestore. Bisogna, dunque, scendere nei dettagli del codice:

```
1 class tipo_gestore {
2
```

processi sospesi che richiedono l'allocazione di una risorsa che, in quel momento, non è disponibile e supponiamo anche che, dall'altra parte, arrivi un processo che ne rilascia tre con `op2`. A fronte del rilascio delle tre risorse, allora, quest'ultimo deve poter risvegliare un solo processo, ma anche tre processi che chiedono ognuno una risorsa e quindi, nel caso di passaggio di testimone, poiché si può risvegliare un solo processo, bisognerebbe aggiungere schemi più complessi basati, ad esempio, su tornelli.

```
3       /* SEMAFORI E CONTATORI ****************************** */
4       semaphore mutex = 1; /* semaforo di mutua esclusione */
5       semaphore sem = 0; /* semaforo condizione */
6       int csem = 0; /* contatore processi sospesi su sem */
7       boolean libera[N] /* indicatori risorsa libera */
8       int disponibili = N; /* contatore risorse libere */
9
10      /* INIZIALIZZAZIONE ********************************** */
11      {
12            for(int i=0; i < N; i++)
13                  libera[i] = true;
14      }
15
16      /* RICHIESTA ***************************************** */
17      public int richiesta() {
18            int i = 0;
19            P(mutex); /* ingresso in sezione critica */
20
21            //Controllo la disponibilita di risorse:
22            //se non ce ne sono, il processo si sospende
23            //in attesa (con passaggio del testimone).
24            if(disponibili == 0) {
25                  csem++;
26                  V(mutex);
27                  P(sem);
28                  csem--;
29            }
30
31            //Individuo la risorsa libera
32            while(!libero[i]) i++;
33
34            libero[i] = false;
35            disponibili--;
36            V(mutex); /* uscita dalla sezione critica */
37
38            return i;
39      }
40
41      /* RILASCIO ****************************************** */
42      public void rilascio(int r) {
43            P(mutex) /* ingresso in sezione critica */
44
45            //Ripristino della stato di libero della risorsa
46            libero[r] = true;
47            disponibili++;
48
49            //Passaggio del testimone
50            if(csem > 0)
51                  V(sem);
52            else
53                  V(mutex);
54      }
55  }
56
57  /* DEFINIZIONE DEL GESTORE ***************************** */
58  tipo_gestore G;
```

```
59
60 /* STRUTTURA DEL GENERICO PROCESSO ********************** */
61 process P {
62       int ris;
63
64       ...
65
66       ris = G.richiesta();
67       <utilizzo della risorsa>
68       G.rilascio(ris);
69
70       ...
71 }
```

Lo schema adottato è quello del ***passaggio del testimone*** *perchè:*

1. *a fronte di un rilascio, è possibile risvegliare al più un processo;*
2. *poiché lo stato del pool che viene gestito è rappresentato da un insieme di variabili condivise tra i processi che, quindi, sono visibili a tutti i processi, chi andrà a fare il risveglio potrà farlo in modo puntuale avendo prima verificato che la condizione sia vera.*

Dal codice, si nota subito la presenza di ***due semafori****:* `mutex` *per garantire la mutua esclusione dei metodi del gestore e* `sem` *che, invece serve a notificare il fatto che ci sono risorse libere. Per quanto riguarda il pool di risorse, invece, è stato impiegato un* ***array di booleani****,* `libera`*, per tenere traccia delle risorse libere, e, inoltre, anche un* ***contatore****,* `disponibili` *che mantiene il numero di risorse libere. Il resto del codice è spiegato nei commenti.*

[a] Due o più risorse sono equivalenti o funzionalmente identiche se, considerando un processo che necessita di una risorsa di quel tipo, non è strettamente necessario che questo ne richieda una in particolare ma è sufficiente che ne ottenga una qualunque dell'insieme. Come esempio di si consideri un utente di un ufficio che deve stampare: se le stampanti sono tutte nella stessa stanza, allora all'utente non interessa quale stampante utilizzare, ma basta che ne sia una della stanza.

[b] Si ricorda, infatti, che il gestore è esso stesso una risorsa condivisa.

3.2.8 Il semaforo risorsa

L'esempio appena visto, in realtà, si presta molto ad essere risolto con un altro schema di sincronizzazione: quello basato sull'impiego di **semafori risorsa**.

Definizione 52: ***Semaforo risorsa***

Tipo di semaforo che viene utilizzato nei problemi ***di allocazione delle risorse****. Nello specifico, è un* ***semaforo generale non limitato superiormente****, con un valore massimo arbitrario che, però, se utilizzato per gestire pool di risorse equivalenti, avrà come limite superiore la cardinalità dell'insieme che gestisce.*

Dunque, il caso classico d'uso del semaforo risorsa è proprio quello dell'esempio appena mostrato, quello della gestione di un insieme di risorse equivalenti. Anche in questo caso,

è previsto una gestore delle risorse ma, al posto di utilizzare un semaforo condizione, verrà adottato un semaforo risorsa `n_ris`, inizializzato con un valore uguale al numero di risorse da allocare.

Segue che, ogni volta che verrà richiesta una risorsa invocando il metodo opportuno dal gestore, verrà chiamata una `P(n_ris)` che, ovviamente, sarà passante se nel pool ci sono ancora risorse disponibili, altrimenti dovrà sospendere il processo che esegue la richiesta; analogamente, in fase di rilascio saranno opportunamente eseguite delle `V(n_ris`.

Esempio: ***Gestione di un pool di risorse equivalenti con semaforo risorsa***

```
class tipo_gestore {

     /* SEMAFORI E CONTATORI ****************************** */
     semaphore mutex = 1; /* semaforo di mutua esclusione */
     semaphore n_ris = N; /* semaforo risorsa per il pool */
     boolean libera[N] /* indicatori risorsa libera */

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

     /* RICHIESTA ***************************************** */
     public int richiesta() {
          int i = 0;
          P(n_ris);
          //Questa P sospende il processo se n_ris = 0.
          //Altrimenti non fa altro che aggiornare il valore
          //del semaforo con il numero di risorse disponibili
          //decrementando l'intero associato al semaforo.

          P(mutex); /* ingresso in sezione critica */

          //Individuo la risorsa libera
          while(!libero[i]) i++;

          libero[i] = false;
          V(mutex); /* uscita dalla sezione critica */

          return i;
     }

     /* RILASCIO ****************************************** */
     public void rilascio(int r) {
          P(mutex); /* ingresso in sezione critica */
          libero[r] = true;
          V(mutex); /* uscita dalla sezione critica */

          V(n_ris);
          //Questa V risveglia eventuali processi in attesa
          //di risorse disponibili. Se non ve ne sono, non fa
          //altro che incrementare il valore del semaforo n_ris
     }
}
```

Come si può notare, la struttura è molto più semplice. Utilizzando la `P(n_ris)` *all'inizio di ogni operazione di richiesta e la* `V(n_ris)` *alla fine di ogni rilascio, si avrà l'effetto che il valore associato al semaforo* `n_ris` ***rappresenterà ad ogni istante il numero delle risorse effettivamente disponibili****; per questo motivo, ovviamente, non c'è necessità di prevedere un contatore delle risorse disponibili come, invece, è stato fatto prima.*
Si specifica, infine, che la `V(n_ris)` *nel metodo* `rilascio` *è fuori dalla sezione critica per un motivo di efficienza: potrebbe anche stare dentro la sezione critica, ma in questo caso si ritarderebbe la liberazione del gestore*[a].

[a] Inoltre, la `P` e la `V`, come si vedrà, devono essere loro stesse implementate come *operazioni atomiche* per cui, a livello di correttezza, l'ordine delle `V`, in questo caso, è indifferente se non per un discorso di efficienza, come già detto.

Esempio: ***Problema dei produttori-consumatori con semaforo risorsa***

In figura 3.22 avevamo visto come risolvere il problema del produttore-consumatore con un semaforo binario composto nel caso in cui il buffer potesse contenere al più un elemento. Se il buffer avesse avuto capacità n, allora la soluzione sarebbe la seguente:

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

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

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

In questo caso, abbiamo un buffer che, in realtà, è una ***coda di messaggi****. Inoltre, i vincoli di precedenza sono stati estesi rispetto al caso precedente:*

- ***un produttore deve fermarsi se in quella coda del buffer non ci sono posti liberi*** *in cui inserire il messaggio;*
- ***un consumatore dovrà attendere se nella coda non c'è nessun messaggio disponibile****, quindi se questa è completamente vuota.*

Tutto questo, è stato espresso con l'impiego di due semafori risorsa: `pn`*, inizializzato a 0 e associato al numero di elementi pieni del buffer, e* `vu`*, inizializzato ad n,*

associato, invece, al numero di elementi vuoti. Le inizializzazioni di questi semafori sono dovute al fatto che il buffer, all'inizio, è vuoto. L'impiego di questi due semafori, allora, ***equivale alla gestione di due pool di risorse equivalenti****: uno che rappresenta gli elementi vuoti e l'altro che, invece, rappresenta i messaggi disponibili nel buffer.*
Dunque:

- `invio` ***è un'operazione di richiesta per gli elementi vuoti*** *mentre è di rilascio per quelli pieni (il produttore richiede l'allocazione di una risorsa elemento vuoto);*
- `ricezione` ***è un'operazione di richiesta per gli elementi pieni*** *mentre è di rilascio per gli elementi vuoti (il consumatore richiede l'allocazione di una risorsa elemento pieno);*

Il semaforo privato

È il caso d'uso più generale del semaforo. Finora, infatti, abbiamo sempre considerato schemi di sincronizzazione in cui l'ordine con cui i processi accedevano alle risorse contese era affidato alla realizzazione della `V`: dunque, il programmatore non aveva modo di influire sull'ordine con cui i processi possono accedere ad una data risorsa. Questa, ovviamente, è una limitazione poiché, spesso, viene fornita una **politica da rispettare che prevede che alcune categorie di processi vengano privilegiate** rispetto ad altri.

In generale, quando c'è una politica da seguire per la gestione di una risorsa, bisogna avere la possibilità di:

- **esprimere il criterio di scelta in base ad una condizione**, che chiameremo condizione di sincronizzazione
- **risvegliare arbitrariamente un particolare processo** poiché, magari, nella politica è stabilito che questo abbia la priorita.

Tutto questo si può realizzare utilizzando il **semaforo privato**.

Definizione 53: ***Semaforo Privato***

Un semaforo `s` *si dice privato per un processo (o per un insieme di processi di uguale priorità) se* ***solamente tale processo (o tale insieme di processi) può chiamare la primitiva*** `P` *sul semaforo* `s`*,* ***inizializzato sul valore*** 0*. Per quanto riguarda la* `V`*, essa può essere eseguita anche da altri processi.*

Per semplicità, consideriamo un semaforo privato per un solo processo: allora, questo è ***privato*** nel senso che solamente quel processo potrà sospendersi su quel semaforo, pur non essendoci vincoli sulla `V`, eseguibile da qualsiasi processo dell'applicazione concorrente. L'impiego di questi semafori è molto vicino all'uso dei semafori condizione, nel senso che un processo che deve accedere ad una risorsa sulla quale è stata specificata una politica di allocazione arbitraria, dovrà testare la sua condizione di sincronizzazione e, se questa non è verificata, invece di sospendersi sull'unico semaforo che si aveva nel caso del semaforo condizione, ora si sospenderà sul suo semaforo privato. A questo punto, se ogni processo, caratterizzato su un certo livello di priorità, si sospende su un suo semaforo proprio, per

il risveglio si andrà a vedere dapprima se c'è qualcuno sul semaforo di più alta priorità, poi si scenderà al livello di priorità sottostante e poi così via fintanto che non si trova il semaforo con la priorità più alta su cui è sospeso un processo.

Esempio: ***Pool di risorse equivalenti con priorità***

Supponiamo di avere un pool di risorse equivalenti e un insieme di processi che possono richiedere una risorsa dell'insieme. Inoltre, ogni processo avrà un attributo che lo caratterizza dal punto di vista della priorità nell'accesso alle risorse.
Dunque, ***quando un processo chiede una risorsa, se è libera la può acquisire*** *ma, al momento del rilascio, se vi sono processi sospesi,* ***chi esegue il rilascio dovrà tenere conto della priorità****: tra tutti quelli sospesi* ***andrà selezionato, per il risveglio, quello con priorità massima****.*
Si ricorda che ***il semaforo non da la possibilità di scegliere quale processo risvegliare*** *bensì la scelta è implicita nel semaforo ed è fatta nella fase della sua implementazione ma, generalmente, è FIFO: il primo processo che si è addormentato è quello risvegliato per prima. Allora, bisogna basarsi sul semaforo privato:*

- *il processo che tenta di acquisire la risorsa, se la condizione di sincronizzazione non è soddisfatta,* ***si sospende sul suo semaforo privato****;*
- *chi rilascia una risorsa, in base alla politica vigente, risveglierà uno dei processi sospesi* ***mediante una*** `V` ***sul semaforo privato*** *del prescelto.*

Gli schemi da adottare per la soluzione sono molteplici, se ne propongono due. Si supponga il programma abbia n processi concorrenti.

```
class tipo_gestore_risorsa {

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

     public void acquisizione(int i) {
          P(mutex); /* ingresso in sezione critica */

          if(<condizione di sincronizzazione) {
               <allocazione risorsa>
               V(priv[i]);
          } else
               <registro che il processo sta per sospendersi>

          V(mutex); /* uscita dalla sezione critica */

          P(priv[i]);
          //Se la condizione di sincronizzazione era soddisfata,
          //allora nell'if precedente era stata eseguita la V
          //per cui ora la P non sospende il processo bensi
          //risulta passante. Se la condizione, invece, non era
          //stata soddisfatta allora il processo si sospende sul
          //suo semaforo privato.
     }

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

```
29            P(mutex); /* ingresso in sezione critica */
30            <rilascio della risorsa>;
31
32            if(<esiste almeno un processo sospeso per il quale
33                la condizione di sincronizzazione risulta
34                soddisfatta>) {
35                      <scelta del processo Pi da riattivare>
36                      <allocazione risorsa a Pi>
37                      <registrazione Pi non piu sospeso>
38                      V(priv[i]);
39            }
40
41            V(mutex); /* uscita dalla sezione critica */
42      }
43 }
```

Questo primo schema prevede un array di n semafori privati, uno per ogni processo, tutti inizializzati a 0 e un altro semaforo di mutua esclusione, `mutex`*. Il gestore, come visibile dal codice, espone due operazioni: una di acquisizione e l'altra di rilascio. Si rimanda ai commenti nel codice per la spiegazione.*
*Inoltre, tale schema è abbastanza semplice: n**ell'acquisizione, la* `P` *sul semaforo privato è fatta all'esterno della sezione critica**, e questo evita alcuni problemi con l'acquisizione e il rilascio del semaforo di mutua esclusione. Tuttavia, ci sono alcuni problemi:*

- *viene **replicato il codice della fase di allocazione** delle risorse sia nell'acquisizione, sia nel rilascio;*
- ***la** `P` **sul semaforo privato viene sempre eseguita**, anche qualora il processo non debba essere sospeso, ed essendo questa una system call, questa soluzione potrebbe risultare inefficiente;*

Dunque, si potrebbe scegliere uno schema diverso, basato su passaggio di testimone:

```
1  class tipo_gestore_risorsa {
2
3       <struttura dati del gestore>
4       semaphore mutex;
5       semaphore priv[n] = {0, 0, ..., 0} /* semafori privati */
6
7       public void acquisizione(int i) {
8            P(mutex); /* ingresso in sezione critica */
9
10           if(!<condizione di sincronizzazione) {
11                <registrazione della sospensione del processo>
12                V(mutex);
13                P(priv[i]);
14                //In questo caso la P viene chiamata solo e soltanto
15                //nel caso in cui la condizione di sincronizzazione
16                //non risulta soddisfatta per cui, sicuramente, essa
17                //sara sospensiva. Dato il passaggio del
18                //testimone, non bisogna preoccuparsi di rieseguire
19                //l'accesso mutuamente esclusivo che viene
                      ereditato.
20
```

```
21                  <registrazione risveglio del processo>
22            }
23
24            <allocazione della risorsa>
25            V(mutex); /* uscita dalla sezione critica */
26      }
27
28      public void rilascio() {
29            int i;
30            P(mutex); /* ingresso in sezione critica */
31            <rilascio della risorsa>;
32
33            if(<esiste almeno un processo sospeso per il quale
34                la condizione di sincronizzazione risulta
35                soddisfatta>) {
36                      <scelta del processo Pi da riattivare>
37                      V(priv[i]);
38                      //La V risveglia il processo che riprende
39                      //l'esecuzione con diritto di mutua esclusione
40                      //sul gestore.
41            }
42            else
43                  V(mutex); /* uscita dalla sezione critica */
44      }
45  }
```

A differenza dello schema precedente, come anticipato, è stato impiegato il ***passaggio del testimone****. In questo caso,* ***il risveglio dei processi è leggermente più complesso*** *ma, come tipico del passaggio del testimone, a fronte di un rilascio, è possibile risvegliare un solo processo alla volta, limitazione che nel primo schema non era presente. Per ovviare a questo si potrebbe complicare lo schema cambiando l'*`ip` *presente dentro il metodo* `rilascio` *in un* `while`*.*

Esempio: ***Produttori-Consumatori con messaggi di dimensione diversa***

Si consideri un buffer di N celle di memoria in cui più produttori possono depositare messaggi di dimensione diversa: rispetto al problema classico del produttori-consumatori, qui non è detto che il messaggio occupi solo una cella, ma può occuparne diverse.
Supponiamo che ***la politica di gestione dia la priorità al produttore che fornisce il messaggio di dimensione maggiore****. Dunque, se più produttori sono in attesa sul buffer pieno, finché quello con il messaggio con dimensioni maggiori (anche dello spazio del buffer) rimane sospeso, nessun altro può depositare il proprio messaggio anche se la sua dimensione potrebbe essere contenuta nel buffer.*
La ***condizione di sincronizzazione****, allora, impone che il deposito possa avvenire solamente se c'è spazio sufficiente nel buffer per la memorizzazione del messaggio e se non ci sono altri produttori in attesa poiché, se ce ne fosse un altro, vorrebbe dire che quest'ultimo avrebbe una richiesta prioritaria rispetto alla sua.*
Per quanto riguarda ***il prelievo*** *ad parte di un consumatore, invece, ovviamente* ***dovrà liberare le celle occupate dal messaggio prelevato*** *e, contestualmente,*

dovrà anche prevedere la riattivazione, tra tutti i produttori sospesi, di quello il cui messaggio ha dimensioni maggiori, *sempre se esiste spazio sufficiente nel buffer; se lo spazio disponibile non è sufficiente allora non viene risvegliato nessun produttore.*
Uno schema di soluzione potrebbe essere il seguente:

```
class buffer {

     int richiesta[num_proc] = 0;
     //con richiesta[i] = numero di celle richieste da Pi,
     //    num_proc = numero totale di processi nell'applicazione;
     //dunque, se richiesta[i] >= 0, vuole dire che il
     //processo i-esimo sta aspettando di avere una
     //quantita di celle pari al valore di richiesta[i]

     int sospesi = 0; /* numero dei processi sospesi */
     int vuote = n; /* numero di celle vuote nel buffer */
     //n = numero totale di celle nel buffer

     semaphore mutex = 1;
     semaphore priv[num_proc] = {0, 0, ..., 0} /* semafori privati
        */

     public void inserimento(int m, int i) {
          //m = dimensione del messaggio,
          //i = id del processo chiamante;

          P(mutex); /* ingresso in sezione critica */

          if(sospesi==0 && vuote>=m) {
               //Assegnazione di m celle ad i
               vuote = vuote - m;
               V(priv[i]);

          } else {
               sospesi++;
               richiesta[i] = m;
          }

          V(mutex); /* uscita dalla sezione critica */
          P(priv[i]);
          //Schema analogo all'esempio precedente
     }

     public void estrazione(int m) {
          //m = numero di celle vuote rilasciate

          int k;
          P(mutex); /* ingresso in sezione critica */
          vuote += m;

          while(sospesi != 0) {
               <individuazione processo Pk con la max richiesta>

               if(richiesta[k] <= vuote) {
                    //Assegnazione celle a Pk
```

```
50                      vuote = vuote - richiesta[k];
51                      richiesta[k] = 0;
52                      sospesi--;
53                      V(priv[k]);
54                  }
55                  else break;
56              }
57
58              V(mutex); /* uscita dalla sezione critica */
59          }
60  }
```

Si noti la condizione di inserimento del messaggio, presente nell'`if`: un processo si sospende non solo se non c'è posto nel buffer, ma anche se ce n'è qualcun altro sospeso e questo è dovuto alla politica di priorità. In questo caso, inoltre, è stato applicato il primo schema dell'esempio precedente ma è stata aggiunta la possibilità di risvegliare più processi grazie al ciclo `while` inserito in `estrazione`.

3.2.9 Realizzazione dei semafori

Il semaforo è uno strumento **realizzato dal nucleo del sistema operativo** che sfrutta le funzionalità per la gestione dei processi: se, ad esempio, un processo invoca la `P` su un semaforo che ha valore nullo, il processo viene effettivamente sospeso ma senza attesa attiva.

Questo, ovviamente, può essere fatto solamente dal nucleo della macchina che ha la completa gestione dei processi, dunque, la `P` e la `V` devono essere realizzate con i due diagrammi di flusso che seguono, sfruttando i meccanismi di sospensione e riattivazione messi a disposizione del kernel:

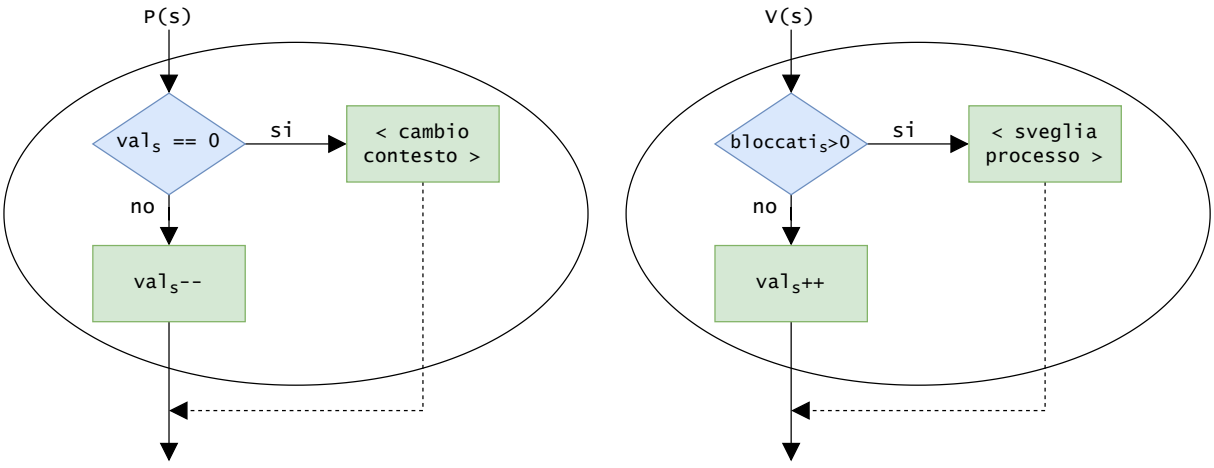


Figura 3.26: Algoritmi di realizzazione della `P` e della `V`

In questo caso, come si diceva, è prevista l'effettiva sospensione del processo nel caso di `P` su un semaforo con valore 0 (`< cambio contesto >`) che, dunque, evita l'attesa attiva evitando un `loop` di attesa. Il processo che viene sospeso, allora, finisce su una **coda specifica che contiene tutti i processi in attesa sul semaforo** su cui ha eseguito

la P; inoltre, poiché il processo è stato sospeso, occorrerà schedulare un nuovo processo a cui assegnare l'uso della CPU.

Analogamente, anche la V non può più avere la struttura della figura 3.17 poiché, avendo dato alla P la possibilità di sospendere dei processi, bisognerà prevedere un meccanismo di risveglio per tali processi (<sveglia processo>: naturalmente, quando un processo esegue una V, se quelli sospesi sono più di uno, si risveglierà un solo processo bloccato.

Per il risveglio dei processi non c'è uno standard ma, tipicamente, viene prevista una coda per cui il risveglio dovrebbe seguire una politica di tipo FIFO.

Tutte le proprietà dei semafori viste precedentemente, ovviamente, rimangono valide anche in questa implementazione.

Se l'implementazione del semaforo fa affidamento sui meccanismi di gestione dei processi propri del kernel della macchina, allora esso non può essere più rappresentato solamente da un intero non negativo bensì, affianco a questo intero, **bisogna prevedere una struttura dati che consenta l'accodamento dei processi** da sospendere:

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

La rappresentazione di un semaforo, dunque, è stata estesa a questa struttura dati che sarà chiamata **descrittore del semaforo** e che, di minima, è costituita dall'intero `contatore` che rappresenta, istante per istante, il valore del semaforo e dalla struttura `queue` di tipo `coda` che contiene i descrittori dei processi che si sospendono su quel particolare semaforo.

Allora:

- **una `P` su un semaforo con contatore** 0 **sospende il processo nella coda `queue`, altrimenti decrementa `contatore`**;
- **una `V` su un semaforo la cui coda `queue` non è vuota, estrae un processo dalla coda, altrimenti incrementa `contatore`**.

Inoltre, nel caso di un'architettura monoprocessore, supponendo che sia attivo il meccanismo di disabilitazione delle interruzioni per garantire l'atomicità della P e della V, una possibile implementazione di queste due operazioni è la seguente:

```
1  void P(semaphore s) {
2       if(s.contatore == 0)
3            <sospensione del processo nella coda associata a s>;
4       else
5            s.contatore--;
6  }
7
8  void V(semaphore s) {
9       if(s.queue != NULL)
10           <estrazione del primo processo dalla cosa s.queue,
11            che viene riportato nello stato di ready>
12      else
13           s.contatore++;
14 }
```

Ovviamente, anche in un'architettura monoprocessore c'è competizione: il semaforo, di fatto, è una risorsa condivisa tra i processi concorrenti dunque tutto funziona purché il corpo della `P` e della `V` siano atomici, ovvero non interrompibili.

Dunque, l'implementazione dei semafori viene effettivamente realizzata dal nucleo della macchina concorrente e dipende da una serie di parametri, tra cui:

- **il tipo di architettura della macchina** (monoprocessore o multiprocessore);
- **le modalità con cui il nucleo gestisce i processi concorrenti**.

L'argomento verrà approfondito nel prossimo capitolo.

3.3 Nucleo di un sistema multiprogrammato nel modello a memoria comune

3.3.1 Caratteristiche e compiti pricipali del nucleo

Nel modello multiprogrammato, detto anche *a processi*, è prevista l'esistenza di tante unità di elaborazione (macchine virtuali) quanti sono i processi: in generale, **ogni processo, ha l'illusione di avere a disposizione un insieme di risorse (memoria e cpu, ma non solo) esclusivamente dedicato a lui**. Ognuna di queste macchine, inoltre, possiede un **set di istruzioni elementari** che sono corrispondenti a quelle dell'unità centrale del sistema sottostante, più i **meccanismi necessari alla gestione dei processi** (creazione ed eliminazione) **e alla comunicazione e sincronizzazione** (inclusi quelli con l'I/O, visti come processi esterni).

Dunque, si procederà analizzando le caratteristiche del nucleo nel vari casi, partendo dai sistemi monoprocessori a quelli a più processori.

Definizione 54: ***Kernel di una macchina concorrente***

Modulo (o insieme di funzioni) realizzato in software ma anche in hardware, che supporta concretamente il concetto di processo e realizza tutti gli strumenti necessari alla loro gestione.

Come sappiamo già, allora, **il nucleo è il livello più interno** di un qualunque sistema basato su più processi ed è anche quello più vicino all'hardware. Inoltre:

- nei sistemi operativi multiprogrammati è il livello più elementare;
- nei linguaggi per la programmazione concorrente è il livello che fornisce il supporto a tempo di esecuzione.

Il nucleo, in oltre, **è l'unico che rileva e gestisce le interruzioni**: ad esempio, i dispositivi che hanno bisogno di colloquiare con periferiche e dispositivi, per farlo, hanno bisogno di utilizzare opportune primitive messe a disposizione dal nucleo stesso che, una volta messe in esecuzione, realizzano, ad esempio, attività di sospensione e riattivazione per l'attesa del completamento di un'operazione, sfruttando il meccanismo delle interruzioni.

I processi, inoltre, non percepiscono il fatto di essere stati sospesi o riattivati bensì hanno solamente l'illusione di disporre della risorsa in modo mutuamente esclusivo senza essere consci del meccanismo *low-level* di interruzioni che c'è a basso livello.

Caratteristiche fondamentali del nucleo

Qualsiasi nucleo deve essere progettato e realizzato secondo opportune linee guida che possono essere sintetizzate come segue:

- **efficienza**: bisogna fare in modo che il carico computazione indotto dall'esecuzione delle funzioni del nucleo sia il più possibile ridotto e, per questo motivo, si sfruttano il più possibile tutti i meccanismi messi a disposizione dall'hardware;
- **dimensioni**: più il nucleo è ridotto, più è semplice e sintetico e più occupa meno memoria (robustezza, tolleranza ai guasti ma possibilità di contrasto con l'efficienza);
- **separazione tra meccanismi e politiche**: aspetto molto importante, che impone che il nucleo offra dei meccanismi sufficientemente generali per l'implementazione e la gestione dei processi e che possano poi dar luogo alla realizzazione di politiche diverse, a seconda anche dell'ambito d'utilizzo della macchina concorrente.

Gestione dei processi

In ogni istante della sua vita, **ogni processo** si trova in un determinato stato: sostanzialmente, ognuno di essi può essere **attivo** o **bloccato**

- **attivo**: un processo in questo stato non ha nessun ostacolo per poter proseguire la sua esecuzione tuttavia non è detto che stia effettivamente utilizzando la CPU;
- **bloccato**: il processo, per un qualche motivo, è stato sospeso all'interno di una coda per cui, durante la fase di utilizzo della CPU, per effetto di un qualche evento o situazione in cui è giunti, questo è stato de-schedulato ed inserito all'interno della coda specifica per quella particolare situazione.

La transizione da attivo a bloccato e viceversa, qualora il nucleo lo preveda, può anche essere realizzato proprio dal semaforo: una `P` bloccante porta il processo dallo stato di attivo a bloccato, la `V` viceversa.

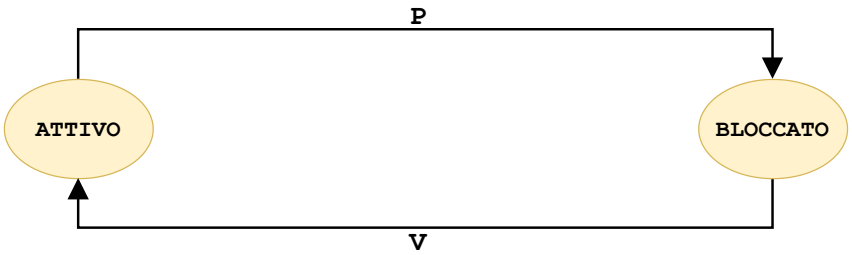


Figura 3.27: Transizione di un processo per effetto della `P` e della `V`

Lo stato di attivo, però, è un *macro-stato* nel senso che può essere espanso come segue: Dunque, un processo attivo può essere in:

- **esecuzione**: il processo ha il controllo della CPU e prosegue l'esecuzione del proprio codice;

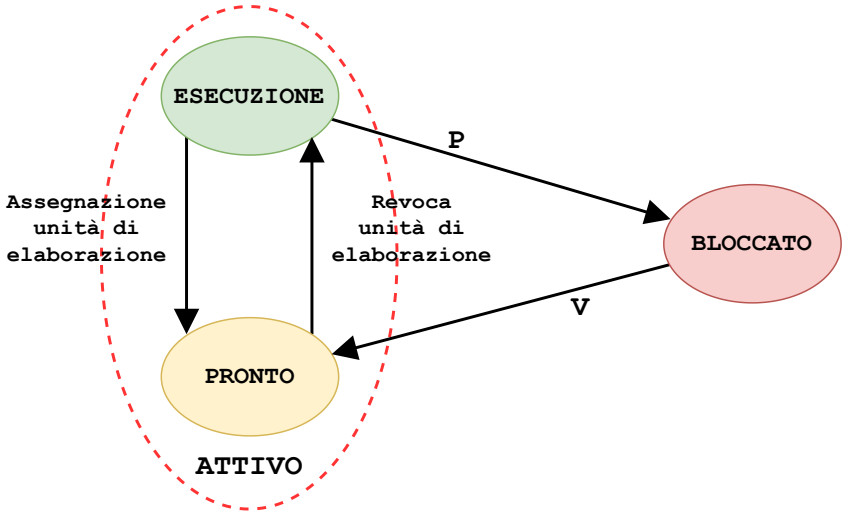


Figura 3.28: Transizione di un processo per effetto della `P` e della `V` con espansione dello stato *attivo*

- **pronto** (o *ready*): il processo ha tutte le carte in regola per poter eseguire ma attende che lo scheduler del sistema lo scelga per poter prendere il controllo della CPU; in questo stato il processo non è sospeso ed è comunque attivo.

Ovviamente, la transazione tra questi due stati è realizzata dallo scheduler che può assegnare o revocare, nei sistemi in cui questo è previsto[15] l'uso del processore.

Per quanto riguarda la gestione dei processi, il nucleo deve:

- **Gestire il salvataggio e il ripristino del contesto:**
  Sia se il processo passa dallo stato di esecuzione a quello di pronto, sia se transita da quello di esecuzione a quello di bloccato, è molto importante che il nucleo si occupi del salvataggio del contesto del processo, ovvero dell'insieme delle informazioni contenute nei registri del processore che *fotografano* lo stato di tale processo che, quindi, deve essere interrotto; queste informazioni vengono salvate nel descrittore[16] e consentiranno, poi, la ripartenza del processo dallo stesso stato in cui si trovata quando è stato interrotto, sempre a carico del nucleo che, dunque, dovrà ripristinare il contesto del processo al momento in cui questo dovrà riprendere l'esecuzione.

- **Effettuare le operazione di scheduling**
  Ricordiamo che lo scheduling è l'attività eseguita dallo *scheduler*, un componente dello stesso nucleo, che si occupa di scegliere il processo a quale, tra i processi pronti, assegnare la CPU non appena quello che la stava utilizzando ne abbandona il controllo (o è costretto a farlo). La scelta, ovviamente, dipende dalla politica: ogni sistema sceglie un algoritmo di scheduling in base a quelle che sono le proprie peculiarità e i propri aspetti.

- **Gestione delle interruzioni**
  Ogni volta che viene sollevata e rilevata un'interruzione, il nucleo deve occuparsi di

[15] Sistemi in cui è prevista la revoca della CPU sono, ad esempio, quelli *pre-emptive*: in questi sistemi è possibile che lo scheduler sottragga la CPU ad un processo in esecuzione riportandolo dallo stato di esecuzione a quello di pronto.

[16] Si ricorda che il descrittore è una struttura dati allocata in memoria centrale che serve a rappresentare in modo corretto e completo il processo che riferisce durante tutto l'arco della sua vita.

mascherare questo evento ai processi gestendolo al proprio interno e traducendolo a seconda del tipo di interruzione e delle operazioni che interessano lo stato dei processi: ad esempio,se un processo è in attesa del completamento di un'operazione di I/O da parte di un dispositivo, quando quest'ultimo l'avrà completata allora invierà alla CPU un'interruzione specifica che verrà rilevata e gestita dal nucleo il quale, interpretando l'evento, cambierà lo stato del processo (o dei processi) che erano in attesa da bloccato a pronto.

- **Realizzare i meccanismi di sincronizzazione**
  Il nucleo deve gestire il passaggio dei processi dallo stato di esecuzione a bloccato implementando opportuni meccanismi di sincronizzazione (come, ad esempio, la `P` e la `V`.

3.3.2 Realizzazione del nucleo in sistemi monoprocessore

Date le premesse generali, ora si analizzerà come sono tradotte in pseudocodice quanto abbiamo appena detto, considerando il modello a memoria comune in un sistema monoprocessore.

Strutture dati del nucleo

- **Descrittore di un processo**
  Il compito principale del nucleo è proprio la gestione dei processi per cui, ognuno di essi, deve essere adeguatamente rappresentato nel nucleo. La struttura dati che assolve questo compito è opportunamente complessa, ma le informazioni di base sono le seguenti:

  - **identificazione del processo**, ovvero l'identificatore che, all'interno del sistema, consente di riferire univocamente il processo in modo puntuale e diretto;
  - **stato del processo**;
  - **modalità di servizio** che rappresenta l'insieme di dati necessari allo scheduler (FIFO, priorità, deadline, quanto di tempo, ecc...);
  - **contesto del processo**, e cioè tutta una serie di informazioni che di solito sono situate nei registri della CPU come il program counter, i registri di stato, altri registri generali e indirizzi delle varie aree di memoria private del processo;
  - **code di processo**, ovvero l'identificatore del processo successivo nella stessa coda in cui il processo è inserito in base al proprio stato.

  Nell'ipotesi di un sistema pre-emptive con priorità, la realizzazione concreta potrebbe essere la seguente:

```
1 /* Definizione del tipo modalita_servizio ***************** */
2 typedef struct {
3      int indice_priorita;
4      int delta_t;
5 } modalita_di_servizio;
6
7 /* Definizione del descrittore ************************** */
8 typedef struct {
9      int nome;
```

```
10      ...
11      modalita_di_servizio servizio;
12      tipo_contesto contesto;
13      tipo_stato stato; //running, ready, waiting, ecc...
14      int successivo;
15  } descrittore processo;
16
17  /* Insieme di tutti i descrittori ************************ */
18  descrittore_processo descrittori[NUM_MAX_PROC];
```

- **Coda dei processi pronti**
  In generale, a seconda del tipo di scheduling che si sta adottando, ci possono essere una o più di una coda di processi pronti.

  Nel caso di scheduling con priorità, come nel listato appena sopra, è necessario prevedere più code per i processi pronti e, dunque, quando un processo viene riattivato è necessario che venga inserito in fondo alla coda corrispondente alla sua priorità.

  In linea di principio, non è detto che in ogni istante ci sia almeno un processo pronto da schedulare e potrebbero anche esserci dei periodi temporali nei quali non c'è nessun processo nello stato di pronto: per far fronte a questa situazione si prevede un processo fittizio sempre nello stato di pronto, chiamato *dummy process*, che ha la priorità più bassa di tutti e che va in esecuzione quando tutte le code sono vuote. Se non c'è nessun processo in esecuzione e non c'è neanche nessun altro processo nello stato di pronto, il *dummy process* va in esecuzione eseguendo un ciclo senza fine o una funzione di servizio, fintanto che qualcun altro, che avrà certamente una priorità più alta della sua, non diventa pronto.

```
1   typedef struct {
2       int primo;
3       int ultimo;
4   } descrittore_coda;
5
6   typedef descrittore_coda coda_a_livelli[NPriorita];
7
8   coda_a_livelli coda_processi_pronti;
9   //Questa definizione istanzia una struttura dati che, in tutto
10  //e per tutto, rappresenta quella che verra utilizzata per
11  //tracciare i processi che si trovano nello stato di pronto
12
13  void inserimento(int P, descrittore_coda C) {
14      //inserimento del processo P in fondo alla coda
15  }
16
17  int prelievo(descrittore_coda C) {
18      //estrazione del primo processo dalla coda
19      //e restituzione del suo indice
20  }
```

  Si specifica che gli interi utilizzati nel codice riferiscono processi che esistono già e che sono mantenuti nel nucleo dalla struttura `descrittori`, mostrata nel punto precedente. Dunque, in realtà, quel che accade è che i processi, mantenuti dal nucleo, vengono collegati alla coda e non che se ne creano di nuovi.

- **Coda dei descrittori liberi**
  Il nucleo, oltre a dover gestire i descrittori che sono inseriti all'interno delle varie

code, dovrà avere anche la possibilità di **reperire un nuovo descrittore** da utilizzare quando è necessario inserire un elemento all'interno della coda: a questo scopo, per ogni coda, viene mantenuta una coda dei descrittori liberi.

```
descrittore_coda descrittori_liberi;
//strutture dati vuote utilizzate per la creazione
//di nuovi processi o per l'inserimento
```

Ovviamente, quando creo un processo ho bisogno di un descrittore che, quando il processo termina, torna libero e, dunque, viene reinserito nella coda dei descrittori liberi.

- **Riferimento al processo in esecuzione**
  Avendo assunto di trovarci in un sistema monoprocessore, è chiaro che questo, in ogni istante, avrà al più un processo in esecuzione dunque, è necessario **tenere traccia di quale sia il processo in esecuzione**. A questo scopo, all'interno del nucleo, è tipicamente mantenuta una variabile che riferisce l'unico processo che in quel momento ha il controllo della CPU:

```
int processo_in_esecuzione;
//indice del processo che sta eseguendo
//(in stato di running)
```

Al momento del boot, inoltre, quando viene *lanciato* il nucleo della macchina concorrente, viene creato il primo processo (ad esempio *init* per Unix) al quale viene subito assegnato l'uso della CPU: dunque, l'indice del processo viene assegnato alla variabile `processo in esecuzione` del nucleo.

Funzioni del nucleo

Le funzioni del nucleo, sostanzialmente, **realizzano le transazioni di stato dei processi**, cosa che richiede una gestione opportuna delle code che verranno predisposte per i processi pronti e quelli sospesi in attesa di determinati eventi.

A livello generale, le funzioni del nucleo possono essere classificati secondo livelli:

- il **livello inferiore**, quello più vicino all'hardware, che realizza le funzionalità che riguardano il cambio del contesto (salvataggio e ripristino); contestualmente a queste operazioni, ovviamente, deve avvenire anche l'azione dello scheduler;
- il **livello superiore**, che si appoggia a quello più basso, che contiene funzioni invocabili dai processi che operano in quel sistema (inclusi quelli esterni che riguardano l'I/O da dispositivi).

Il nucleo, per svolgere i propri compiti, deve poter operare sulle strutture dati che rappresentano lo stato del sistema che sono di sua esclusiva pertinenza e, dunque, sono accessibili dalle sue funzioni. Questo, però, non preclude ai processi la possibilità di accedere ad alcune di queste strutture (ad esempio, non impedisce ai processi di utilizzare i semafori): un processo le può richiederle tramite opportune chiamate di sistema (`P` e `V` sono realizzate tramite system call).

Tutto questo, ovviamente, deve essere eseguito in modo corretto dato che ci si trova in un ambiente concorrente: **le funzioni del nucleo**, allora, **devono essere eseguite in modo mutuamente esclusivo**; inoltre, ogni volta che un processo ha bisogno del

nucleo, lo chiama in causa con il meccanismo delle system call, attraverso cui si ha una commutazione del modo di esecuzione, dal modo user a quello kernel. Alla system call, infatti, corrisponde una trap che viene inviata e gestita con la routine associata a quella chiamata di sistema.

Il discorso vale sia per i *processi interni* al sistema, sia per quelli *esterni* che, invece, rappresentano l'attività dei dispositivi[17]; la differenza, però, sta nel fatto che per quest'ultimi, l'interruzione è asincrona mentre per i primi è sincrona.

Funzioni di livello inferiore

Queste funzioni si occupano di **realizzare il cambio di contesto**, ovvero quel meccanismo che va innescato ogni volta che si avvicendano due processi nell'uso della risorsa CPU. La sequenza delle operazioni è la seguente:

1. **Salvataggio dello stato**
   Quando il processo deve essere de-schedulato vengono salvati i registri della CPU relativi al suo contesto nel descrittore:

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

2. **Assegnazione della CPU ad un nuovo processo**
   La CPU viene assegnata al nuovo processo che la utilizza fintanto che non termina o che lo scheduler non gli revochi l'uso della CPU:

```
void assegnazione_CPU() {
     int k = 0;

     while(coda_processi_pronti[k].primo == -1)
          k++;
     //Si sta assumendo che l'indice cresca in modo
     //opposto alla priorita (l'indice nullo corrisponde
     //alla massima priorita) .
     //Si esce dal ciclo quando si trova la coda non
     //vuota di priorita massima tra tutte.

     int j = prelievo(coda_processi_pronti[k];
     processo_in_esecuzione = j;
     //Con questo assegnamento si stabilisce che il
     //processo che puo utilizzare la CPU ha
     //indice j.
}
```

3. **Ripristino del contesto scelto nella fase precedene**
   Quando il primo processo viene nuovamente schedulato, bisogna ripristinare il suo stato in modo da farlo riprendere esattamente da dov'era quando gli è stato revocato l'uso della CPU:

[17] Ad esempio si potrebbe volere che un dispositivo voglia salvare il contenuto di un buffer, mantenuto in memoria, su un file.

```
void ripristino_stato() {
     int j = processo in esecuzione;

     <registro-temp> = descrittori[j].servizio.delta_t;
     //Questo valore rappresenta la dead-line oltre la quale
     //bisogna revocare la CPU al processo (quanto di tempo).

     <registri-CPU> = descrittori[j].contesto;
}
```

A queste tre funzioni, se ci si trova in un ambiente in cui il cambio di contesto è basato sul quanto di tempo, bisogna prevedere anche un **dispositivo di temporizzazione** che, sostanzialmente, dia la possibilità al nucleo di rilevare quando il quanto di tempo di un processo è terminato per eseguire la revoca della CPU a quel processo ed assegnarla a qualcun altro. Dunque, bisogna anche esprimere la funzione che realizza il cambio di contesto utilizzando le tre funzioni appena realizzate. Il cambio di contesto, dunque, si verifica quando il temporizzatore segnala che il quanto di tempo del processo attualmente in esecuzione è terminato: a quel punto, il nucleo prende il controllo della situazione e, sostanzialmente, esegue la funzione di cambio di contesto:

```
void cambio_di_contesto() {
     salvataggio_stato();
     int j = processo in esecuzione;
     int k = descrittori[j].servizio.priorita;
     inserimento(j, coda_processi_pronti[k]);
     //Il processo j viene messo nella coda dei
     //processi pronti relativa alla priorita
     //del processo (k).

     assegnazione_CPU();
     //Da questo funzione si uscira con un valore
     //di processo_in_esecuzione diverso da quello
     //precedente alla chiamata.

     ripristino_stato();
}
```

3.3.3 Realizzazione del semaforo (caso monoprocessore)

I meccanismi ai quali si fa riferimento durante lo sviluppo di applicazioni concorrenti vengono effettivamente implementati e realizzati all'interno del nucleo della macchina concorrente. Dunque, si analizzerà come il semaforo, lo strumento di più basso livello per la gestione di processi concorrenti, è implementato all'interno del nucleo.

Per adesso, di farà riferimento al solo caso monoprocessore. L'implementazione in un sistema di questo tipo necessiterà di:

- una **variabile intera non negativa** che rappresenta il valore del semaforo;
- un **puntatore ad una lista di descrittori di processi in attesa** sul semaforo per poter realizzare l'attesa come effettiva sospensione dei processi ed evitare l'attesa attiva.

La gestione dei processi concorrenti, come spiegato poco fa, è a carico del nucleo che, dunque, ha la possibilità di cambiare lo stato di un processo.

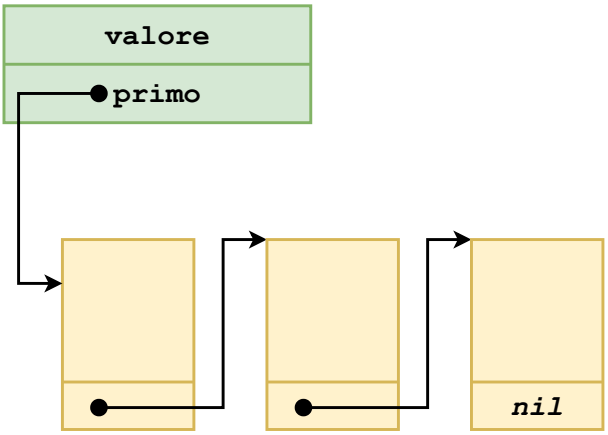


Figura 3.29: Realizzazione del semaforo nel nucleo (valore + puntatore)

La coda, inoltre, viene gestita con una politica in linea di principio arbitraria ma, nella maggior parte dei casi, è di tipo FIFO. Allora, in corrispondenza di una chiamata `P` su un semaforo con valore nullo, il descrittore del processo che ha lanciato l'istruzione viene inserito nella coda del semaforo; specularmente, l'effetto di una `V` è quello di prelevare il descrittore in testa alla coda.

Rimanendo nell'ipotesi di processi gestiti con priorità, allora al semaforo viene effettivamente associato un insieme di code, una per priorità:

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

A questo punto, non rimane che scendere nei dettagli della `P` e della `V` che, ovviamente, come tutte le funzioni del nucleo saranno chiamate di sistema.

Implementazione della `P`

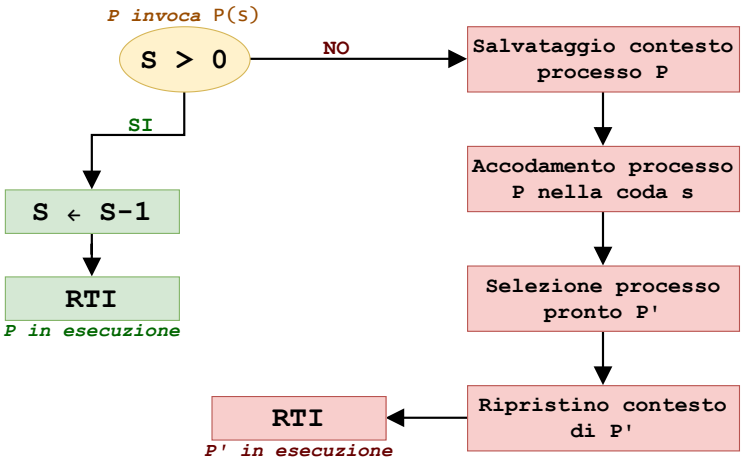


Figura 3.30: Diagramma di flusso dell'operazione `P`

La figura 3.30 mostra il diagramma che realizza la `P` all'interno del nucleo di un sistema monoprocessore concorrente. Si ricorda che la chiamata di una `P` da parte di un processo

P, in realtà, è una chiamata di sistema che, dunque, comporta la commutazione dal modo user a quello kernel. Passando al codice:

```
descr_semaforo semafori[max_num_sem];
//Insieme di tutti i semafori del sistema

typedef int semaforo;
//Un semaforo viene individuato mediante
//il relativo indice nel vettore dei semafori

void P(semaforo s) {
     int j, k;

     if(semafori[s].contatore == 0) {
          salvataggio_stato();

          /******* Processo in esecuzione ***************/
          /***/ j = processo_in_esecuzione; /************/
          /**********************************************/

          /******* Priorita processo in esecuzione ******/
          /***/ k = descrittori[j].servizio.priorita /***/
          /**********************************************/

          inserimento(j, semafori[s].coda[k]);
          assegnazione_CPU();
          ripristino_stato();
     } else
          semafori[s].contatore--;
}
```

Implementazione della V

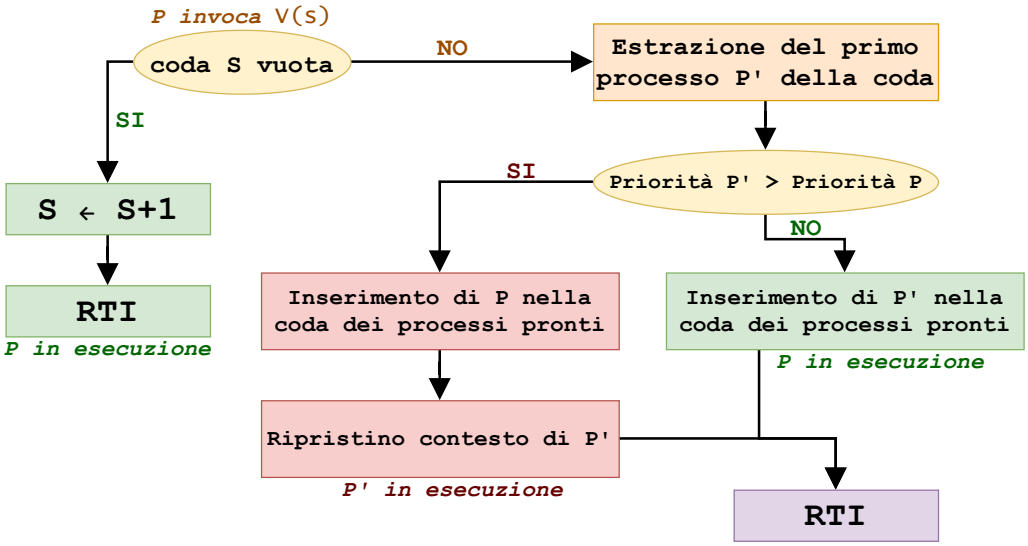


Figura 3.31: Diagramma di flusso dell'operazione V

La figura 3.31 mostra il diagramma dell'operazione V. Segue il codice:

```
1  void V(semaforo s) {
2       int j, p, k;
3       int q=0;
4
5       while(semafori[s].coda[q].primo == -1 && q < min_priorita)
6            q++;
7
8       if(semafori[s].coda[q].primo != -1) {
9
10           /******* Processo in attesa su s **************/
11           /***/ k = prelievo(semafori[s].coda[q]); /*****/
12           /**********************************************/
13
14           /******* Processo in esecuzione ***************/
15           /***/ j = processo_in_esecuzione; /************/
16           /**********************************************/
17
18           /******* Priorita processo in esecuzione ******/
19           /***/ p = descrittori[j].servizio.priorita /***/
20           /**********************************************/
21
22           if(p < q) {
23           //Si ricorda che il valore piu piccolo di priorita,
24           //per convenzione, indica quella piu alta.
25
26                inserimento(k, coda_processi_pronti[q]);
27           } else {
28                salvataggio_stato();
29                inserimento(j, coda_processi_pronti[p]);
30                processo_in_esecuzione = k;
31                ripristino_stato();
32           }
33      } else
34           semafori[s].contatore++;
35
36 }
```

Meccanismo di chiamata a *supervisor call*

Sia la `P`, sia la `V` sono delle *supervisor call* (*SVC*) e questo, quindi, comporta il passaggio dall'ambiente dell'utente a quello del nucleo e viceversa. **Tale meccanismo, essenzialmente, si basa sulle interruzioni** (*trap*): l'attività viene che si stava eseguendo viene interrotta per dare il controllo al nucleo il quale, eseguendo in modalità privilegiata, applicherà la routine di gestione di risposta a quel particolare evento (la chiamata di sistema) eseguendo l'azione richiesta.

Tutto questo, però, si basa su:

- una **pila di sincronizzazione** tra il nucleo e il processo che effettua la chiamata;
- alcuni **registri della CPU**, come il program counter, il registro di stato e altri registri generali associati agli ambienti del nucleo e dei processi.

Nel momento in cui un processo P effettua una chiamata a supervisor call:

1. viene lasciata una **trap di tipo sincrono**;
2. vengono **salvati i registri PC e PS** relativi a P in cima alla pila del nucleo;
3. viene **caricato in PC e PS quanto necessario per l'esecuzione dell'operazione di risposta**;
4. la CPU può eseguire l'operazione richiesta che, ovviamente, è codificata all'interno della routine della procedura di gestione;
5. in base a com'è la chiamata (bloccante o non bloccante) la CPU ri-schedula il processo bloccato.

3.3.4 Realizzazione del nucleo in sistemi multiprocessore

Finora abbiamo dato alcuni esempi di realizzazione del nucleo in ambienti concorrenti monoprocessore. Adesso è possibile estendere la trattazione considerando sistemi multiprocessore in cui, dunque, la concorrenza è effettivamente permessa anche a livello hardware.

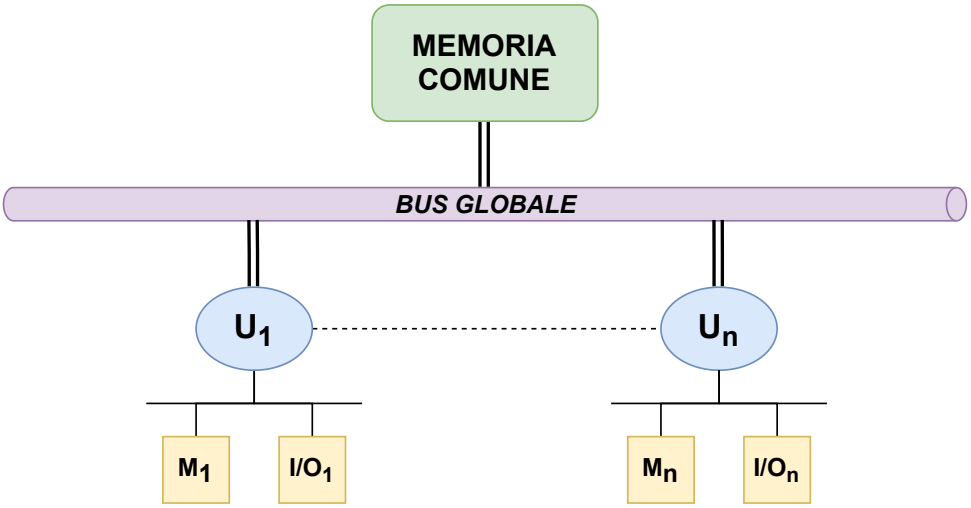


Figura 3.32: Schema di un'architettura multiprocessore

In un sistema multiprocessore, dunque, **sono disponibili più unità di elaborazione**, collegate tutte ad uno stesso **bus comune**, a sua volta collegato ad una **memoria condivisa**, accessibile da tutte le CPU; in generale, però, ogni unità di elaborazione può disporre di una sua area di memoria privata (M_1, M_2, ecc...).

Per quanto riguarda il sistema operativo dedicato alla gestione di un'architettura multiprocessore, la complessità è sicuramente superiore rispetto al caso monoprocessore date le diverse CPU da gestire e l'accesso alla memoria condivisa: in un contesto di questo tipo, allora, non si ha più solamente concorrenza ma **anche parallelismo**.

Il modello organizzativo di questi sistemi non è unico ma i due più diffusi sono i seguenti:

- **MODELLO SMP** (*Symmetric Multi Processing*): c'è un unico nucleo nel sistema multiprocessore, allocato nell'unica memoria condivisa;
- **MODELLO A NUCLEI DISTINTI**: c'è una molteplicità di nuclei, uno per ogni diversa CPU.

Il sistema uscente da questi due modelli è comunque **unico** e da la possibilità a più processi di eseguire in modo parallelo e concorrente che, all'occorrenza, indipendentemente da dove stiano eseguendo, possono interagire e sincronizzarsi.

Il modello SMP

Questo modello è quello che troviamo in quasi tutti i sistemi più comuni.

Definizione 55: ***Modello SMP***

Modello organizzativo di un sistema operativo che gestisce un'architettura multiprocessore che prevede la presenza di un'unica copia del nucleo, allocata nella memoria condivisa, che gestisce tutte le unità di elaborazione e le risorse a disposizione.

Ovviamente, avendo un **unico nucleo** che gestisce tutto, quest'ultimo **vede le unità di elaborazione a disposizione come fossero un pool di risorse equivalenti**: dunque, un processo, nell'arco della sua vita, può essere allocato in una qualunque delle CPU disponibili a livello hardware.

Ciò significa che se un processo nasce su un nodo e se, per un qualche motivo, questo abbandona l'uso della CPU di cui ha usufruito inizialmente, al momento in cui verrà ripristinato, in un sistema SMP **c'è la possibilità che possa essere schedulato su un nodo diverso rispetto a quello a cui era stato assegnato inizialmente**.

Inoltre, è possibile che più processi in esecuzione su CPU diverse, possano aver bisogno, nel medesimo istante, dell'intervento del nucleo (mediante system call): **il nucleo è codice, ma soprattutto dati**, in quanto per gestire tutto deve anche gestire in modo opportuno tutta una serie di strutture dati. **Non è possibile, allora, che due processi accedano in uno stesso istante alle stesse strutture dati** del nucleo a causa del problema della mutua esclusione.

Sulla base di quanto appena detto, si verifica competizione tra le CPU nell'esecuzione delle funzioni del nucleo e, dunque, c'è una forte necessità di sincronizzazione mediante opportuni strumenti e politiche.

Soluzioni per la **sincronizzazione**:

- **Soluzione ad un solo lock**:
  Tipologia di soluzione molto elementare che prevede l'associazione al nucleo di un unico lock, che serve per rendere l'intero nucleo un'unica grande sezione critica. Questo significa che se due processi avessero bisogno contemporaneamente di richiamare funzioni del nucleo, il loro accesso a quest'ultimo sarebbe rigorosamente sequenzializzato, indipendentemente da quali siano le funzioni di cui hanno bisogno i due processi in competizione.

  Questa soluzione funziona ma è molto elementare e limitante: il grado di parallelismo è molto limitato in quanto essa esclude l'esecuzione contemporanea di parti diverse del nucleo.

- **Soluzione a più lock**:
  Soluzione che associa al nucleo un certo numero di lock. Infatti, data la complessità del nucleo, che contiene un numero elevato di operazioni le quali assolvono funzioni diverse, è possibile individuare dei blocchi di codice sufficientemente indipendenti

tra loro. Dunque, se si riesce a tradurre il nucleo in un'insieme di sezioni critiche, ognuna riferita anche a risorse diverse, allora si può associare ad ogni classe un diverso lock: in questo modo, ogni operazione che accede ad una risorsa eseguirà un prologo con una `lock` specifica su quella risorsa e terminerà con un epilogo che farà la relativa `unlock`.

Questa soluzione prevede che il vincolo della mutua esclusione valga solo sulle singole parti del nucleo, per cui, chiaramente, permette un grado molto maggiore di parallelismo

Un altro problema, oltre a quello della sincronizzazione, è quello dello **scheduling dei processi**: essendoci un unico nucleo, questo ha la totale libertà nella scelta di quali CPU assegnare ai processi nel senso che, in linea di principio, non c'è nulla che gli impedisca di schedulare un processo su una qualsiasi CPU a disposizione. Questo da la possibilità allo scheduler di attuare delle politiche ottimizzate nella gestione delle risorse di elaborazione: dunque, è possibile attuare ottimizzazioni di ***load banacing***.

Nel momento in cui un processo deve essere schedulato, nel modello SMP il nucleo ha la facoltà di decidere su quale delle CPU a disposizione allocare quel processo.

Ci sono, però, da fare alcune considerazioni:

- ogni processore ha una sua memoria privata per cui, se un processo p, prima di essere schedulato, eseguiva sulla CPU k, allora nella memoria di quest'ultimo sono rimaste tracce di p (come, ad esempio, il codice). Tralasciando il load balancing, se fosse richiesto di privilegiare l'esecuzione di quel processo sul nodo k, dove ci sono già tracce dell'esecuzione di p (magari c'è già anche il codice), allora si risparmierebbe sui costi di ricaricamento dei dati necessari al processo per eseguire;
- ogni CPU ha una cache usata, ad esempio, nella TLB in memoria virtuale, per cui, nel caso in cui si dovesse ri-schedulare un processo già eseguito su una particolare unità di elaborazione, anche in questo caso potrebbe convenire riutilizzare lo stesso processore per motivi analoghi a quelli del punto precedente.

Normalmente, quindi, in un sistema SMP l'obiettivo è quello di coniugare l'esigenza di distribuzione del carico con le considerazioni appena mostrate. La scelta della politica, ovviamente, avrà un impatto anche sulle strutture dati che il nucleo deve gestire (una singola coda dei processi pronti nel caso di load balancing oppure una coda per ogni nodo in altri casi).

Modello a nuclei distinti

A prosito del modello SMP, abbiamo appena visto che è necessario predisporre dei meccanismi di sincronizzazione ogni volta che un processo richiede una funzione al nucleo e questo indipendentemente dal numero di processori a disposizione nell'hardware: questo, però, può rappresentare un **collo di bottiglia** poiché, se i processori sono molti, allora potrebbero essere in numero considerevole anche i processi in esecuzione che obbligatoriamente si sincronizzano gli uni con gli altri in base a quanto detto poco fa. Questo, ovviamente, **potrebbe incidere sulle prestazioni qualora l'architettura conti su un numero elevato di CPU**; dunque, in questo caso, conviene adottare una soluzione

più scalabile che, dunque, che svincoli il sistema dalla necessità di sincronizzare i processi che richiedono l'intervento del nucleo.

Definizione 56: ***Modello a nuclei distinti***

Modello organizzativo di un sistema operativo che gestisce un'architettura multiprocessore che impone che, ad ogni nodo di elaborazione, sia dedicato un proprio nucleo privato.

Nonostante questo, però, l'intero sistema viene visto come un unico ambiente di esecuzione, esattamente come nel caso di SMP: processi in esecuzione su nodi diversi potranno comunque sincronizzarsi ed interagire, come accade nel caso precedente.

Definizione 57: ***Nodo virtuale***

In un sistema multiprocessore a nuclei distinti, si dirà nodo virtuale l'insieme dei processi che fanno riferimento ad uno stesso nucleo. Ogni nodo virtuale, ovviamente, dispone di un proprio nucleo le cui strutture dati sono allocate sulla memoria privata del nodo fisico a cui quello virtuale viene associato.

Secondo la visione dei processi, allora, **il sistema è scomposto in una serie di nodi virtuali**, ognuno dei quali racchiude i processi che sono stati allocati ad un certo nodo. Dunque, un sistema di questo tipo può essere visto anche come un'aggregazione di sistemi monoprocessori in cui ogni singolo nodo è gestito da un proprio nucleo in modo locale (il codice e le strutture dati del nucleo sono allocati sulla memoria privata del relativo nodo).

Bisogna, però, necessariamente permettere un meccanismo che consenta l'interazione tra processi di nodi virtuali diversi e questo è possibile grazie alla memoria comune. Se questo non fosse possibile, ovviamente, non si potrebbe parlare di un singolo sistema.

Poiché ogni nodo è gestito da un proprio nucleo, allora un processo che viene creato in determinato un nodo, rimarrà su di esso per tutta la sua esecuzione e non avrà possibilità di transitare su altri nodi.

Questo, ovviamente, preclude la possibilità di attuare politiche di load balancing.

SMP vs nuclei distinti

La prima differenza già evidenziata tra i due modelli, risiede nel fatto che in SMP un processo può transitare su nodi differenti mentre nel modello a nuclei distinti questo non è possibile. Inoltre:

- **il modello al nuclei distinti è molto più scalabile** rispetto al primo in quanto non c'è più quel collo di bottiglia dovuto alla presenza di un unico nucleo: ogni processo, quando ne ha bisogno, richiede le operazioni che gli occorrono al nucleo locale;
- **nel modello SMP la gestione delle risorse è più ottimale**, infatti, nel modello a nuclei distinti non c'è possibilità di far transitare i processi nei diversi nodi e questo impedisce, nel complesso, l'attuazione di politiche di ridistribuzione del carico computazionale;

Realizzazione dei semafori in SMP

Consideriamo una situazione in un cui abbiamo più lock per la protezione delle strutture dati gestite dal nucleo. Nel caso dei semafori, le strutture dati rilevanti sono quelle che rappresentano i singoli semafori, per cui ognuno di essi avrà un suo lock specifico[18], e la coda dei processi pronti, anch'esse ognuna con un lock distinto.

Dunque, in questa situazione, due `P` su due semafori diversi potranno eseguire in modo completamente parallelo se non risultano sospensive[19].

Consideriamo, allora, un semaforo in un sistema SMP su cui c'è almeno un processo sospeso il quale ha priorità maggiore di almeno uno dei processi che stanno utilizzando la CPU:

1. un processo esegue la `V` su quel semaforo;
2. la `V` estrae il descrittore del processo da risvegliare dalla propria coda, esattamente come nel caso monoprocessore;
3. la `V`, data l'ipotesi, dovrà attivare le politiche di preemption.

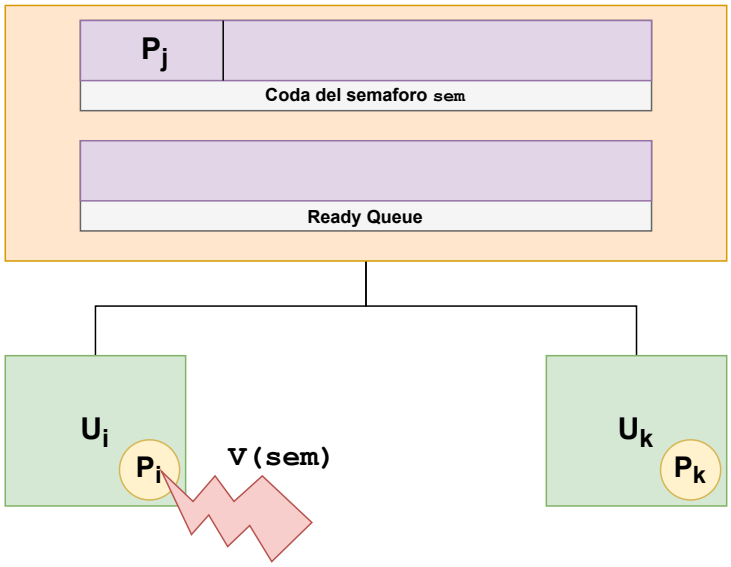


Figura 3.33: Esecuzione di una `V` su un sistema SMP con pre-emption e risveglio di un processo a priorità maggiore

La figura 3.33 mostra quanto si sta descrivendo. Il processo P_i, infatti, eseguendo una `V` sul semaforo `sem`, risveglia un processo P_j che, però, ha priorità su P_k, un processo in esecuzione sull'unità k: data la politica, bisognerebbe deschedulare quest'ultimo, il che significa che P_i dovrebbe indurre un cambio di contesto in un'unità diversa da quella su cui sta eseguendo. Tutto questo **deve presumere la presenza di un meccanismo di segnalazione tra processori**.

[18] Due processi che utilizzano due semafori diversi potranno farlo in reale parallelismo in quanto i due semafori, essendo strutture dati differenti, andranno ad impattare lock differenti. Due `P` su semafori diversi potranno operare in modo contemporaneo nel caso in cui non siano sospensive, mentre in caso contrario saranno sequenzializzati solamente gli accessi alla coda dei processi pronti.

[19] Sempre per il discorso che non c'è accesso alla coda dei processi pronti in quanto le due `P` avranno il solo effetto di decrementare il valore dei semafori.

Per scalzare il processo P_k bisogna innanzitutto interromperlo e, per farlo, P_i invierà un segnale di interruzione dal processore U_i, su cui è in esecuzione la `V`, a U_k. L'interruzione sarà rilevata immediatamente dal nucleo che, dunque, individua la causa dell'interruzione avviando il cambio di contesto richiesto.

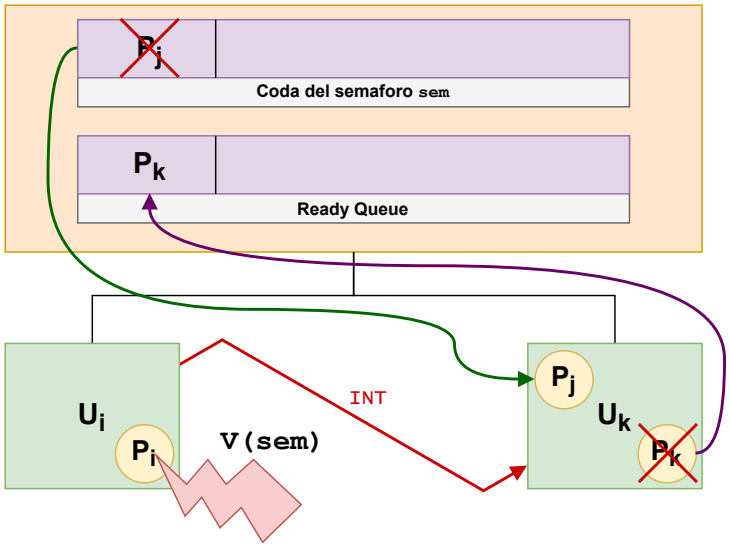


Figura 3.34: Deschedulazione di un processo per effetto di una `V` eseguita da un processo su un'unità di elaborazione diversa

Quanto appena descritto è sintetizzato nella figura 3.34 e prende il nome di **segnalazione inter-processore**.

Realizzazione dei semafori nel modello a nuclei distinti

Nel caso a nuclei distinti, la soluzione è ovviamente diversa poiché ogni singolo processore è associato al proprio nucleo il quale, dunque, gestisce i processi e le strutture dati (semafori inclusi) che riguardano la propria unità di elaborazione. Pertanto, il singolo processo che esegue su un certo nodo ha un certo *scope*: uno privato, che riguarda la propria unità di elaborazione, e uno condiviso per la comunicazione e la sincronizzazione con gli altri processi che eseguono su altri nodi. La distinzione, ovviamente, si ripercuote anche sui semafori, per cui bisogna distinguere tra:

- **semafori privati**, cioè quelli utilizzati solo ed esclusivamente da processi appartenenti allo stesso nodo virtuale;
- **semafori condivisi**, cioè quelli utilizzati da processi appartenenti a nodi virtuali differenti.

L'implementazione dei semafori privati è esattamente la stessa del caso monoprocessore e, quindi, ciò che è interessante è capire come realizzare il semaforo condiviso, la cui visibilità è, invece, estesa a tutti i nodi i quali devono poter accedere a questo strumento. Si ricorda che tutti i nodi, essendo in un sistema multiprocessore, vedono la memoria condivisa che può essere sfruttata come strumento per rendere visibile i semafori condivisi a tutti i processori del sistema.

I semafori condivisi, allora, sono **allocati in memoria condisiva** e dispongono di un proprio lock, anch'esso memorizzato in memoria comune. Si ricorda, però, che il descrittore di ogni processo è di pertinenza del nucleo del nodo a cui questo appartiene, il quale

è l'unico a poter vedere e gestire i descrittori dei processi di quel nodo: in altre parole, **il descrittore di un processo non potrà mai uscire fuori dal nucleo del nodo a cui appartiene**. Questo comporta il fatto che, per realizzare il semaforo condiviso, è necessario prevedere che in memoria condivisa vengano mantenute tutte le informazioni che servono alla sincronizzazione di processi su nodi diversi ma, è anche necessario prevedere, all'interno del nucleo di ogni singolo nodo, una coda che contiene i descrittori dei processi sospesi su quel semaforo.

Per consentire la visibilità globale, e quindi l'interazione di processi appartenenti a nodi diversi, oltre a predisporre una coda per ogni singolo nodo, gestita dal nucleo relativo a quest'ultimo, occorre anche un'opportuna struttura dati, la **coda dei rappresentanti**, all'interno della memoria comune, che, quindi, è accessibile da ogni nucleo.

Definizione 58: ***Rappresentante di un processo***

In un sistema multiprocessore a nuclei distinti, il rappresentante del processo è una struttura dati che mantiene le informazioni di alto livello che, a livello globale, consentono di individuare:

- ***il nodo di appartenenza*** *del processo;*
- ***l'identità del processo*** *all'interno del suo nodo di appartenenza (il pid).*

Riassumendo, considerando un semaforo condiviso s, le strutture dati che devono essere mantenute sono le seguenti:

- **una coda su ogni nodo** n_i che contiene i processi sospesi su s di pertinenza di n_i
  $\rightarrow$ risiede nella memoria privata di n_i e viene gestita solo dal nucleo di n_i
- **una coda globale in memoria condivida** che contiene i rappresentanti
  $\rightarrow$ risiede in memoria condivisa ed è accessibile da tutti i nuclei dei nodi del sistema

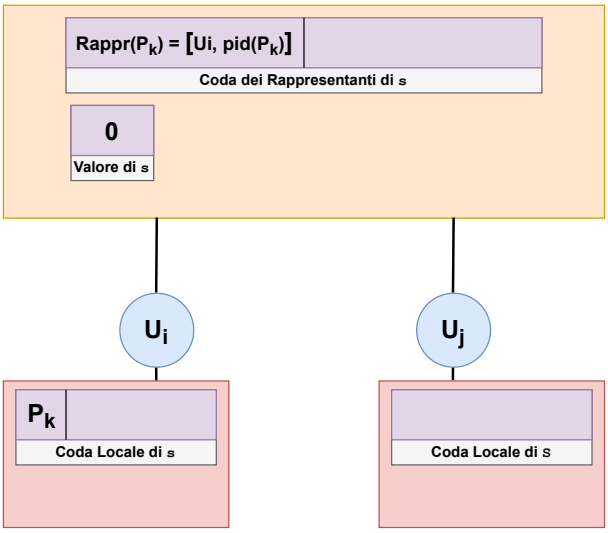


Figura 3.35: Rappresentazione di un semaforo condiviso nel modello a nuclei distinti nel caso di un processo sospeso

Nel caso di una `P` bloccante eseguita da un processo P_k su un semaforo s, allora, gli effetti saranno due:

- **uno locale**, ovvero il nucleo del nodo u_i dove sta eseguendo il processo P_k, dovrà inserirne il descrittore alla coda locale associata a al semaforo s;
- **uno globale**, ovvero il rappresentante di P_k viene inserito all'interno della coda relativa ad s in memoria condivisa.

Grazie all'inserimento del rappresentante del processo nella coda condivisa, sarà poi possibile identificare correttamente il processo quando sarà invocata la `V` sul semaforo s. Consideriamo, allora, un processo P_h appartenente al nodo U_j, diverso da quello su cui giace il processo sospeso P_k, e supponiamo che P_h esegua una chiamata `V(s)`. Allora:

1. **viene verificata la coda dei rappresentanti** per controllare se ci sono processi sospesi e, nel caso, **si estrae il rappresentate del processo da risvegliare**; dunque, per il nostro caso, si estrae il rappresentante di P_k e si analizza il contenuto;
2. **il nucleo del processore** U_j, su cui sta eseguendo la `V(sem)`, **comunicherà tempestivamente ad** U_i, a cui, invece, appartiene il processo P_k, **il fatto che** P_k **dovrà essere risvegliato** e, quindi, riattivato;
3. il nucleo del nodo U_i, essendo l'unico a poterlo fare, estrae il descrittore di P_k dalla coda locale e provvede a risvegliarlo (mettendolo nella coda dei processi pronti o adottando la preemption).

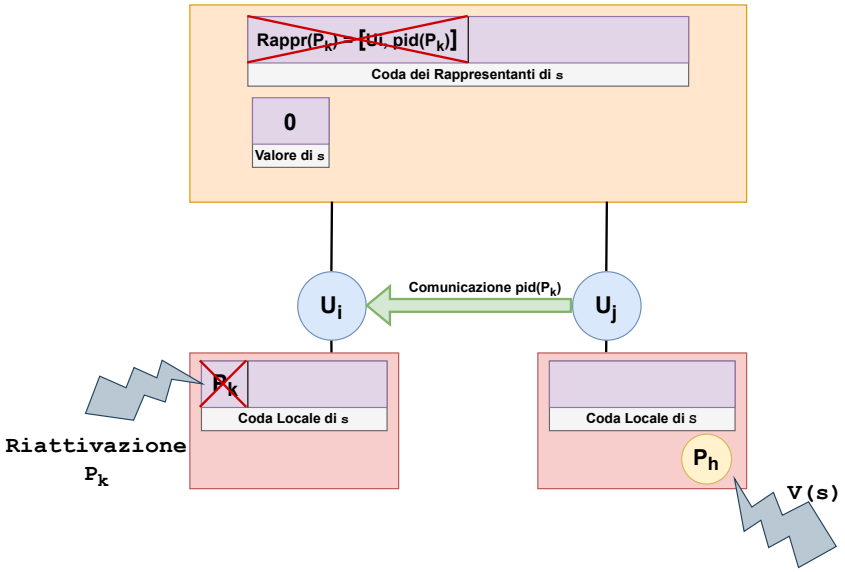


Figura 3.36: Esecuzione di una `V` nel modello a nuclei distinti con risveglio di un processo

Dunque, si potrebbe dire che la struttura di un singolo nucleo del singolo nodo, è analoga a quella del nucleo monoprocessore con la differenza che sono presenti meccanismi di comunicazione con gli altri nuclei del sistema. Non rimane, allora, che definire qual è il meccanismo di comunicazione che permette ad U_j di comunicare tempestivamente ad U_i per il risveglio di P_k.

Anche in questo caso, il meccanismo è basato sulle interruzioni, ovvero lo strumento pi tempestivo possibile a disposizione. Tale meccanismo segue il modello a scambio di messaggi: nella memoria comune viene predisposto un buffer che funge proprio da canale di comunicazione tra i nuclei U_i e U_j.

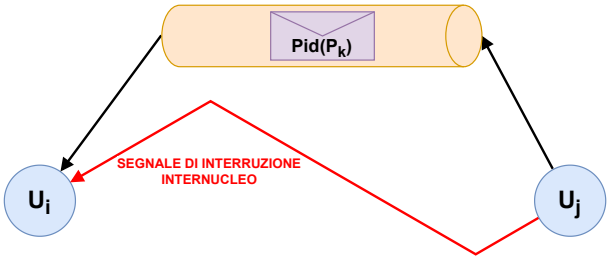


Figura 3.37: Meccanismo di comunicazione internucleo nel modello a nuclei distinti

Tale canale, nel nostro esempio, viene utilizzato dal nucleo U_j per comunicare ad U_i il pid del processo da risvegliare. U_j, quindi, scriverà nel canale l'informazione sul processo da risvegliare e, successivamente, invierà un segnale di interruzione internucleo ad U_i che, di conseguenza, leggerà l'informazione contenuta nel canale attivando il procedimento necessario per portare U_k allo stato di attivo.

A livello di codice, il tutto è tradotto come segue:

```
void P(semaforo sem) {
     if(<sem appartiene alla memoria privata>)
          <uguale al caso monoprocessore>
     else {
          lock(x);
          <esecuzione della P con eventuale sospensione
           del rappresentante del processo nella coda
           di sem>;
          unlock(x);
     }
}

void V(semaforo sem) {
     if(<sem appartiene alla memoria privata>)
          <uguale al caso monoprocessore>
     else {
          lock(x);
          if(<coda non vuota) {
               if(<il processo appartiene al nodo del segnalante>)
                    <caso monoprocessore>;
               else {
                    //Il semaforo risulta condiviso
                    <estrazione processo dalla coda dei rappresentati>;
                    <determinazione del buffer di comunicazione>;
                    if(<area occupata>)
                         <attesa area libera>;
                    <inserimento identificatore del
                     processo riattivato nel buffer>;
                    <invio interrupt al nodo a cui appartiene
                     il processo>;
               }
          }

          <si incrementa il valore del semaforo>;
          unlock(x);
     }
}
```

3.4 Il modello a scambio di messaggi

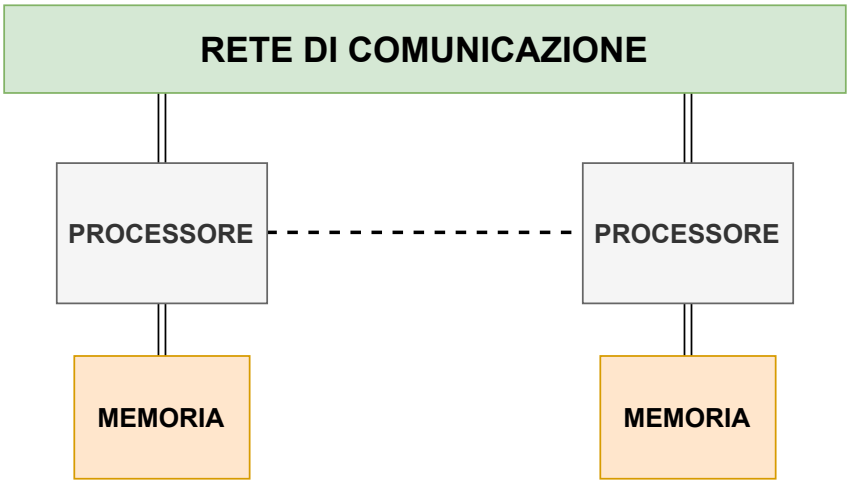


Figura 3.38: Architettura generale del modello a scambio di messaggi

Nel modello a scambio di messaggi, i processi **non hanno in alcun modo la possibilità di condividere memoria**, per cui l'architettura è quella mostrata in figura 3.38: ogni processo concorrente fa riferimento ad una propria memoria privata e, per comunicare con gli altri processi, ha a disposizione una **rete di comunicazione**.

Le caratteristiche principali di questo modello sono le seguenti:

- **ogni processo può accedere esclusivamente alle risorse che sono allocate nella propria memoria locale**;
- **ogni risorsa, allora, è accessibile esclusivamente ad un processo** che, dunque, assume il ruolo di gestore della risorsa;
- pur non essendoci memoria comune, può esserci la necessità da parte di un processo di sfruttare una risorsa che è direttamente accessibile ad un altro processo: allora, in un contesto di questo tipo, il modello di programmazione da adottare è quello client/server in cui **il processo gestore della risorsa, ovvero l'unico che ha la possibilità di accedervi direttamente, funge da servitore** mentre quelli che necessitano, in modo indiretto, della risorsa faranno riferimento a quel processo e, sfruttando un meccanismo di scambio dei messaggi, potranno richiedere l'esecuzione di operazioni su di essa ed ottenere l'eventuale risultato.

Risulta chiaro, allora, che **il concetto di gestore della risorsa si basa sull'idea di un processo servitore** e quindi, ogni processo, per poter usufruire dei servizi disponibili su di una risorsa, dovrà mettersi in contatto con il gestore grazie al meccanismo di scambio di messaggi.

3.4.1 Canali di comunicazione

Per consentire lo scambio di messaggi, assumendo che non ci sia memoria condivisa, occorrono degli strumenti in grado di mettere in comunicazione i processi: i canali, ovvero astrazioni realizzate dal sistema operativo che permettono a due o più processi di mettersi in contatto.

Definizione 59: ***Canale di comunicazione***

Nel modello a scambio di messaggi, un canale è il ***collegamento logico attraverso il quale due o più processi possono comunicare*** *scambiandosi messaggi.*

Mentre il nucleo della macchina si occupa di offrire l'astrazione che realizza il canale, **i linguaggi di programmazione offrono strumenti linguistici di alto livello per**:

- **specificare i canali**, è questo, in molti casi, viene fatto attraverso la definizione di opportune variabili che, in realtà, rappresentano dei canali di comunicazione ;
- **mettere a disposizione delle primitive** che permettono di esprimere concretamente le interazioni tra i processi attraverso un dato canale (`send` e `receive`).

Caratteristiche dei canali

Non tutti i canali sono uguali e, a seconda del sistema operativo, possono avere determinate caratteristiche che, sostanzialmente, possono essere derivate da tre aspetti:

1. **direzione del flusso dei dati** che attraversano il canale

   → **monodirezionale**:
   il flusso dei messaggi va solo dal mittente verso il destinatario;

   → **bidirezionale**:
   il canale può essere utilizzato sia per inviare sia per ricevere informazioni (es: procedure remote).

2. **designazione del canale**, ovvero cosa e chi sono i processi che possono comunicare attraverso al canale (origine e destinazione, figura 3.39)

   → **link**:
   esprime un collegamento uno-a-uno, ovvero mette in comunicazione un mittente con un destinatario (canale simmetrico);

   → **port**:
   esprime un collegamento molti-a-uno, ovvero il canale può essere utilizzato da una molteplicità di mittenti per inviare dati ad un unico destinatario come accade, ad esempio, nel modello client/server (canale asimmetrico);

   → **mailbox**:
   esprime un collegamento molti-a-molti, ovvero il canale consente la comunicazione tra una molteplicità di mittenti e una molteplicità di destinatari, implementando, quindi, il concetto di casella dalla quale possono attingere più destinatari come, ad esempio, le pipe di Unix (canale asimmetrico).

3. **tipo di sincronizzazione** che il canale impone sui processi che comunicano attraverso di esso

   → **comuncazione asincrona**:
   tipo di comunicazione in cui il mittente, all'atto dell'invio del messaggio, non è soggetto ad alcun vincolo di sincronizzazione nei confronti del destinatario, ovvero il primo invia il messaggio, depositandolo del canale, e riprende la sua

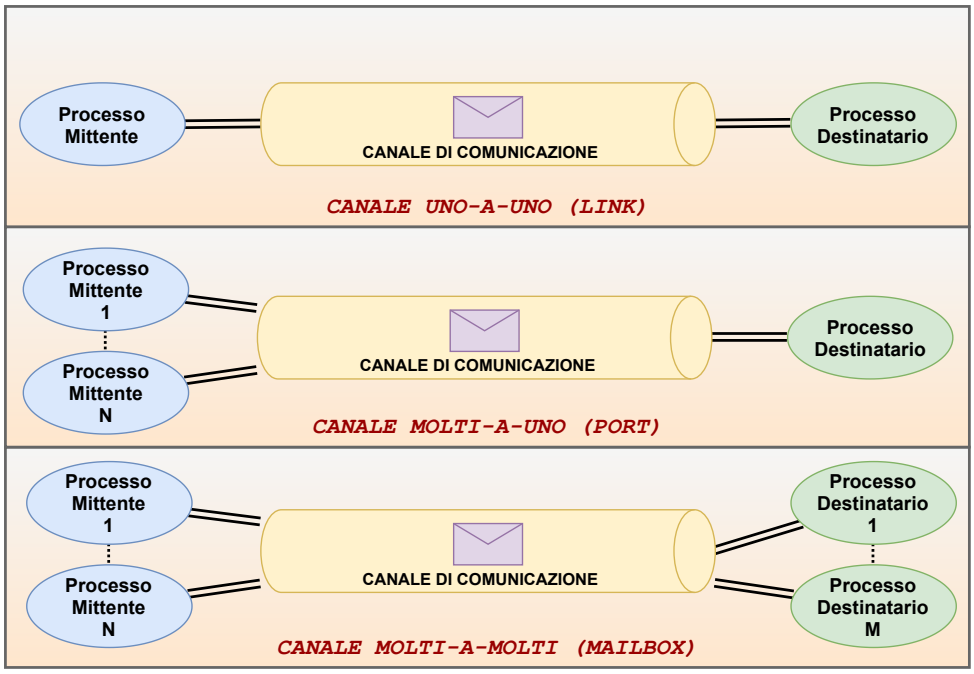


Figura 3.39: Designazione dei canali: tipologie

esecuzione *incurante* se il destinatario ha ricevuto o meno il messaggio inviato. È chiaro, allora, che il messaggio conterrà informazioni che il destinatario non potrà mai attribuire allo stato attuale del mittente, in quanto potrebbe leggerlo anche molto tempo dopo che quest'ultimo l'ha depositato nel canale: l'invio del messaggio, infatti, non è un punto di sincronizzazione tra mittente e destinatario. Una comunicazione di questo tipo gode delle seguenti proprietà:

- **carenza espressiva**, poichè il mittente non può trasferire al destinatario informazioni sullo stato attuale (in quanto potrebbe essere cambiato quando il destinatario legge il messaggio);
- **assenza di vincoli di sincronizzazione**, per cui il grado di concorrenza (o parallelismo) è favorito in quanto il mittente, non essendo costretto ad attendere, può fare altre cose;
- **necessità di buffer a capacità illimitata**, poiché c'è bisogno di mantenere e accodare i messaggi inviati ma non ricevuti[20].

→ **comunicazione sincrona** (o rendez-vous semplice):
si introduce un vincolo stringente di sincronizzazione che impone che chi manda un messaggio debba attendere che il destinatario esegua la corrispondente primitiva di ricezione (`receive`) e, analogamente, chi riceve, se non sono ancora presenti messaggi nel canale, deve attendere che il mittente esegua l'invio. In questo caso **la comunicazione diventa un punto di sincronizzazione importante** dato che il mittente non può procedere con la sua esecuzione fintanto che il mittente non riceve il messaggio: a questo punto, il mittente può comunicare al destinatario informazioni sul suo stato attuale. A livello di realizzazione, allora, non è più necessario un buffer. Segue, quindi che:

[20] Ovviamente, questo non è realizzabile: è necessario **prevedere un limite alla capacità del buffer**, per cui in caso di buffer pieno il mittente si sospende il mittente in attesa che ci sia spazio (ma non è sempre così in quanto potrebbero esserci situazioni in cui, al posto che bloccare il processo, si sollevano delle eccezioni).

- **la comunicazione sincrona è più espressiva di quella asincrona**, in quanto il contenuto dei messaggi è riferibile allo stato attuale del mittente;
- **la semantica sincrona permette un grado di parallelismo inferiore a quella asincrona**;
- **la semantica sincrona è più facile da realizzare** in quanto non è necessario prevedere un buffer per l'accodamento dei messaggi come in quella asincrona.

Tuttavia, i sistemi di base offrono una `send` di tipo asincrono e questo perché è il tipo di comunicazione più generale: infatti, la `send` sincrona è realizzabile mediante una comunicazione di tipo asincrona, come in figura 3.40, in cui sono stati utilizzari due canali monodirezionali.

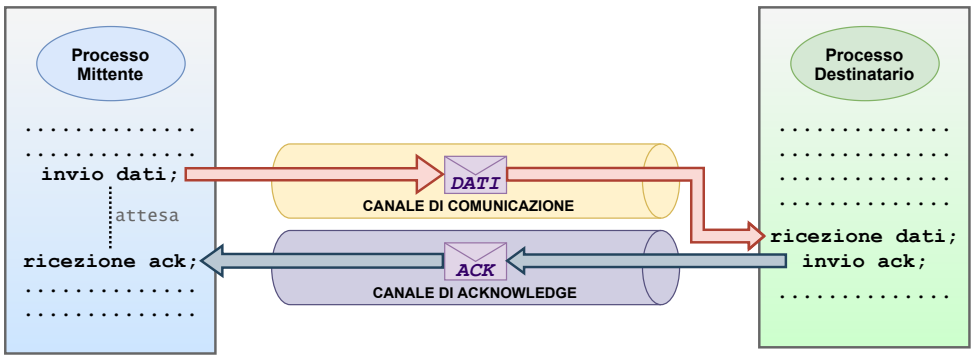


Figura 3.40: Realizzazione della `send` sincrona mediante comunicazioni asincrone

Il messaggio `ACK` ha contenuto privo di significato e il suo unico scopo è quello di notificare l'avvenuta ricezione del messaggio da parte del destinatario.

→ **sincronizzazione estesa**:
questa tipologia di comunicazione parte dall'assunzione che ogni messaggio inviato rappresenti una richiesta al destinatario dell'esecuzione di una certa azione che soltanto quest'ultimo può compiere (modello client/server). Tale semantica, allora, prevede che il mittente rimanga in attesa non solo quando il destinatario ha ricevuto il messaggio, bensì fino a quando questo non avrà completato l'azione che il messaggio gli richiedeva[21]. Rispetto alle altre due forme precedenti:

- **la sincronizzazione estesa introduce un punto di sincronizzazione più vincolante** rispetto a quella di tipo sincrono poiché costringe il mittente ad attendere il completamento di una certa serie di istruzione che viene eseguita dal destinatario e, dunque, ogni dato inviato potrà essere definito attuale.
- **il parallelismo viene ridotto** ancor più rispetto ai casi precedenti, data la rigidità della sincronizzazione.

Nella figura 3.41 è stato utilizzato un canale bidirezionale ma nulla vieta che vengano utilizzati una coppia di canali unidirezionali.

[21] Questo meccanismo ha una forte analogia con quello di procedura remota al punto che, con il tempo, ha preso il nome di *RPC* (Remote Procedure Call).

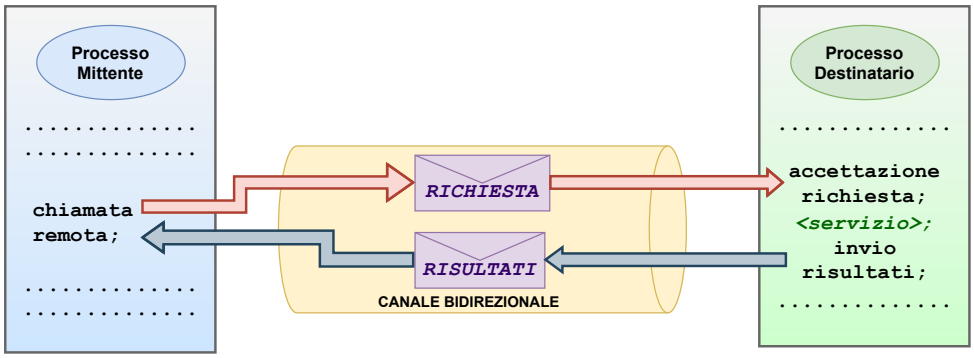


Figura 3.41: Protocollo per la comunicazione con sincronizzazione estesa (procedura remota)

3.4.2 Costrutti linguistici e primitive per la comunicazione

Nel corso di questa sotto-sezione verranno analizzati i costrutti linguistici e le primitive per la comunicazione nel modello a scambio di messaggi in un linguaggio rappresentativo (pseudocodice) utilizzato solo per spiegarne le caratteristiche.

Definizione del canale: `port`

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

- **port** identifica un canale asimmetrico **molti-a-molti**;
- **tipo** tipo di dato trasferibile nel canale (canale tipato);
- **identificatore** denota il canale.

Chiaramente, non ci si riferisce ad uno specifico linguaggio: `port` è semplicemente un tipo che identifica un canale ma non è detto che la primitiva mantenga lo stesso nome nei diversi linguaggi. Ovviamente, il canale viene dichiarato localmente ad un processo (il ricevente) ed è visibile agli altri processi mediante la dot notation:

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

Primitiva per l'invio dei messaggi: `send`

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

- **porta** identifica in modo univoco il canale a cui inviare il messaggio ed è espressa secondo la dot notation vista poco fa;
- **valore** espressione che rappresenta il contenuto del messaggio inviato il cui contenuto deve avere lo stesso tipo di `<porta>`

Esempio: ***Esempio di utilizzo della send***

```
1  port int ch1;
2  send(125) to P.ch1;
```

Il processo che esegue queste righe di codice invia il valore 125 al processo P tramite il canale `ch1` da cui solo P può ricevere. Il fatto che la `send` sia sincrona o no dipende dalla semantica e dalle caratteristiche del canale e, dunque, del linguaggio.

Primitiva per la ricezione dei messaggi: `receive`

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

L'obiettivo della `receive` è quello di estrarre dati dal canale. La sintassi prefeve:

- **varibile** è il riferimento ad una variabile locale al processo ricevente alla quale verrà assegnato il messaggio ricevuto;
- **porta** è una porta aperta localmente dal processo ricevente.

In generale, la receive ha una semantica **bloccante**, ovvero il processo che la esegue, nel caso in cui il messaggio atteso non sia ancora arrivato, si sospende in attesa che il messaggio arrivi. Se, invece, c'è almeno un messaggio nel canale allora la receive non è sospensiva e il suo effetto sarà estrarre il messaggio per assegnare il valore alla variabile specificata come argomento. Infine, normalmente, la receive fornisce l'informazione su chi sia il mittente del messaggio (valore di ritorno) in quanto chi riceve può aver bisogno di sapere chi ha inviato il messaggio.

Esempio: ***Esempio di utilizzo della receive***

```
1     processo proc;
2     proc = receive(m) from ch1;
```

Il processo che esegue queste righe di codice si sospende fintanto che nel canale `ch1` non c'è un messaggio: quando arriva, lo estrae e ne assegna il valore ad `m` assegnando a `proc` l'identificatore del processo mittente. Si nota che se nel canale c'era già un messaggio allora la receive non è bloccante e quello estratto è il primo messaggio in coda al canale (FIFO).

È chiaro che la `receive` deve essere bloccante: sarebbe un guaio se si desse la possibilità al processo destinatario di andare avanti anche se nel canale non ci sono messaggi. Tuttavia, ci sono delle situazioni molto frequenti in cui il meccanismo della receive bloccante può essere un ostacolo: si supponga, infatti, di avere un server che fornisce una molteplicità di servizi e che prevede, per ognuno di essi, un diverso canale, come in figura 3.42.

Il server, almeno idealmente, essere sempre pronto a reagire ad ogni nuova richiesta di servizio, interpretandola ed eseguendo il servizio richiesto: se, dunque, la receive è bloccante, l'unica possibilità che ha il server è quella di eseguire in seguenza le receive su ogni canale (polling) come nel listato che segue:

```
1 //Codice Server:
2 //...
3
4 while(true) {
5     p = receive(m) from canale1;
6     <esecuzione di servizio1>
7
```

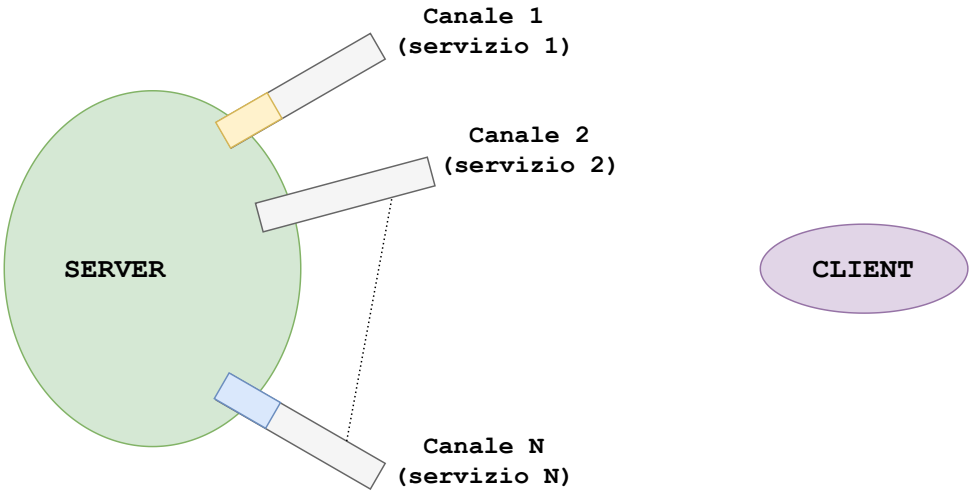


Figura 3.42: Server che fornisce una molteplicità di servizi ad ognuno dei quali è dedicato un canale specifico

```
8       p = receive(m) from canale2;
9       <esecuzione di servizio2>
10
11      //...
12
13      p = receive(m) from canaleN;
14      <esecuzione di servizioN>
15  }
```

Ovviamente, il problema è che si possono verificare situazioni del in cui il server rimane in attesa su di una receive mentre in altri canali si accodano richieste che non possono essere soddisfate fintanto che il server è bloccato su un altro canale, anche se quest'ultimo è vuoto. Inoltre, il server potrebbe bloccarsi in attesa di richieste su un canale che, magari, potrebbe riceverne tra molto tempo, se non mai. **La sola `reiceive` bloccante non è sufficiente a esprimere in modo efficiente il comportamento di un server** che, invece, dovrebbe essere pronto a servire ogni richiesta, nel minor tempo possibile, qualsiasi sia il servizio (e, quindi, il canale) a cui la richiesta riferisce.

Una prima scelta, allora, potrebbe essere **prevedere una receive con una semantica non bloccante** che, quindi, non sarebbe altro che un'istruzione per ispezionare una porta: se c'è un messaggio lo estrae altrimenti notifica semplicemente al chiamante che il canale è vuoto. Questo permette al server di ispezionare sequenzialmente lo stato di tutti i canali finché non ne trova uno in cui c'è una richiesta da soddisfare. **Questo permette di risolvere, a livello logico, il problema ma crea loop di attesa attivi** e ciò non è efficiente in quanto il server consuma CPU anche se tutti i canali sono vuoti. La soluzione è un meccanismo basato su guardia.

Comando con guardia

Il meccanismo ideale per la soluzione del problema del server, che deve attendere *sequenzialmente* su più canali, deve:

- dare a server la **possibilità di monitorare lo stato dei canali** contemporaneamente

- nel caso in cui in un canale fosse presente una richiesta, deve **abilitare la ricezione dei messaggi da quello specifico canale**, svincolandosi dal meccanismo di ispezione;
- se tutti i canali sono vuoti perché non ci sono richieste, ci vuole un **meccanismo che mette il processo in uno stato di sospensione** in cui permanere fintanto che non arriva un messaggio, qualsiasi sia il canale su cui arriva.

Tale meccanismo è realizzato mediante **comandi con guardia**, un costrutto messo a disposizione da diversi linguaggi di programmazione che operano sul modello a scambio di messaggi.

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

dove la guardia è costituita dalla coppia

```
<espressione booleana> -> <receive>;
```

- **l'espressione booleana** viene detta **guardia logica**;
- **la receive** con semantica bloccante che viene detta **guardia di ingresso**,

La valutazione di una guardia può dare i seguenti esiti:

GUARDIA LOGICA	MESSAGGI NEL CANALE	GUARDIA
false	indifferente	**fallita**
true	$= 0$	**ritardata**
true	> 0	**valida**

Allora, l'esecuzione di un comando con guardia determina i seguenti effetti in base alla valutazione della guardia:

- **guardia fallita**
  il comando fallisce, indipendentemente dal numero di messaggi nel canale; questo fallimento è da intendersi come la mancata produzione di effetti: semplicemente, si passa all'istruzione successiva;
- **guardia ritardata**
  il processo che esegue il comando con guardia viene sospeso: quando arriverà un messaggio sul canale, il processo verrà riattivato, eseguirà la receive prelevando il messaggio e, successivamente, eseguirà `istruzione`;
- **guardia valida**
  il processo esegue la receive estraendo il messaggio e, immediatamente dopo, esegue `istruzione`.

Comando con guardia alternativo: `select`

Preso da solo, il comando con guardia è un buon costrutto ma non permette di risolvere il problema del server che ci siamo posti poco fa: esso, infatti, diventa significativo nei **comando con guardia composti**.

```
select {
     [] <guardia_1> -> <istruzione_1>;
     [] <guardia_2> -> <istruzione_2>;
     //...
     [] <guardia_N> -> <istruzione_N>;
}
```

La `select` (comando con guardia alternativo), allora, è un'istruzione strutturata che ha dei rami, ognuno dei quali è un'istruzione con guardia semplice. Quando un processo esegue una select, **vengono valutate tutte le guardie di tutti i rami**:

- **se una o più guardie sono valide**, ovvero se ci sono guardie in cui l'espressione booleana è vera e nel canale c'è almeno un messaggio, la select seleziona in modo non deterministico uno solo dei rami e lo esegue;
- **se non ci sono guardie valide ma c'è almeno una guardia ritardata**, il processo in esecuzione si sospende mettendosi in attesa che arrivi il primo messaggio su uno dei rami con guardia ritarda: quando il primo messagio arriva, viene selezionato il relativo ramo, estratto il messaggio ed eseguita l'istruzione corrispondente;
- **se tutte le guardie sono fallite**, ovvero tutti i rami hanno la condizione falsa, indipendentemente dal fatto che ci siano messaggi in attesa sul canale, la select fallisce e non produce effetti.

Questa istruzione, per quanto appena detto, risolve il problema del server multi-servizio tuttavia, utilizzando un altro tipo di comando con guardia, è possibile risolvere il problema in modo ancor più efficiente.

Comando con guardia ripetitivo: `do []`

```
do {
     [] <guardia_1> -> <istruzione_1>;
     [] <guardia_2> -> <istruzione_2>;
     //...
     [] <guardia_N> -> <istruzione_N>;
}
```

Anche in questo caso viene specificato una serie di rami in cui l'ordine è ininfluente (selezione non deterministica) con la differenza che la valutazione è eseguita in modo ciclico, ripetitivo (guardia ripetitiva). Ad ogni iterazione, allora, vengono valutate le guardie di tutti e tre i rami:

- **se una o più guardie sono valide**, viene scelto in modo non deterministico uno dei rami con guardia rapida (come nel caso precedente) e viene eseguito passando, poi, al ciclo successivo;

- **se non ci sono guardie valide ma c'è almeno una guardia ritardata**, il processo in esecuzione si sospende in attesa che arrivi un messaggio su uno dei rami con guardia ritardata: una volta ricevuto il messaggio, il ciclo riparte;
- **se tutte le guardie sono fallite**, non accade nulla e, essendo in un comando iterativo, questa condizione determina l'uscita dal ciclo.

A questo punto, il problema del server-multiservizio è facilmente risolvibile con un comando con guardia ripetitiva.

Esempio: ***Soluzione al problema del server multi-servizio***

```
process server {
     port int canale1;  //esecuzione di servizio1() su R
     port real canale2; //esecuzione di servizio2() su R
     tipo_di_R R;       //risorsa gestita dal server
     int x;

     do {
          [] (cond1); receive(x) from canale1 -> {
               R.S1(x);
               <eventuale restituzione dei risultati al cliente>;
          }

          [] (cond2); receive(x) from canale2 -> {
               R.S2(x);
               <eventuale restituzione dei risultati al cliente>;
          }
     }

}
```

Ovviamente, assumiamo che `cond1` *e* `cond2` *sono sempre vere: le guardie, allora, non possono essere fallite ma solamente valide o ritardate. Se entrambe sono ritardate allora significa che non c'è nessun messaggio su nessuno dei due canali, per cui il processo si sospende in attesa che arrivi un messaggio su uno dei due canali, evitando, quindi, l'attesa attiva. Quando arriverà la prima richiesta, si sveglierà, eseguirà la receive corrispondente ed eseguirà il servizio richiesto.*
Nel caso in cui una guardia sia ritardata ma una sia valida, in questo caso viene selezionato il ramo relativo a quella valida.
Un ultimo caso da valutare è quello in cui entrambe le guardie siano valide: se questo accadesse, avviene una selezione non deterministica di un ramo, che quindi viene scelto in modo del tutto casuale.

3.4.3 Primitive di comunicazione asincrone

Questo tipo di primitive prevedono una semantica in cui **è assente il vincolo di sincronizzazione** tra chi invia il messaggio e chi lo riceve.

Essendo nel modello a scambio di messaggi, ovviamente, le risorse sono locali ad ogni processo che, per quanto già detto, assume il ruolo di gestore della risorsa: dunque, nel caso di send asincrona, è necessario capire come implementare il modello nel caso in cui, su una risorsa, siano previsti:

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

Risorsa condivisa con una sola operazione

Consideriamo il caso di una risorsa condivisa che mette a disposizione di una serie di processi *clienti* una sola operazione con il solo vincolo della mutua esclusione.

- **Modello a memoria comune**
  Nel modello a memoria comune il gestore è un **moniton con una sola entry**:

```
/* LATO SERVER ********************************** */
monitor gestore {
     tipo_var var;
     <eventuale inizializzazione>

     entry tipo_out fun(tipo_in x) {
          <corpo della funzione fun>;
     }
}

gestore ris; //istanza del gestore

/* LATO CLIENT ********************************** */
thread client {
     tipo_in a;
     tipo_out b;

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

  Il monitor, ovviamente, incapsula al suo interno la risorsa e prevede la sola operazione di tipo `entry`[22].

- **Modello a scambio di messaggi**

```
/* LATO SERVER ********************************** */
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>

```

[22] Una entry, per definizione, è un'operazione mutuamente esclusiva che, quindi, può essere eseguita da al più un thread alla volta.

```
13      while(true) {
14           p = receive(x) from input;
15           y = fun(x);
16           send(y) to p.risposta;
17      }
18 }
19
20 /* LATO CLIENT ********************************** */
21 process client {
22      port tipo_out risposta;
23      tipo_in a;
24      tipo_out b;
25      process p;
26
27      //...
28      send(a) to server.input;
29      p = receive(b) from risposta;
30      //...
31 }
```

Essendo erogato un solo servizio, non è limitante il fatto che la *receive* sia bloccante; inoltre, la `send` è di tipo asincrono ma, per come è stato progettato il codice, il mittente si mette subito in attesa di una risposta, per cui questo attende che il server gli invii il risultato del servizio eseguito.

Risorsa condivisa con una più operazione

Consideriamo il caso di una risorsa condivisa che mette a disposizione di una serie di processi *clienti* due operazioni con il solo vincolo della mutua esclusione.

- **Modello a memoria comune**
  Nel modello a memoria comune il gestore è un **moniton con due entry**:

```
1  /* LATO SERVER ********************************** */
2  monitor gestore {
3       tipo_var var;
4       <eventuale inizializzazione>
5
6       entry tipo_out1 fun1(tipo_in1 x) {
7            <corpo della funzione fun1>;
8       }
9
10      entry tipo_out2 fun2(tipo_in2 x) {
11           <corpo della funzione fun2>;
12      }
13 }
14
15 gestore ris; //istanza del gestore
16
17 /* LATO CLIENT ********************************** */
18 thread client {
19      tipo_in1 a1;
20      tipo_out1 b1;
21      tipo_in2 a2;
22      tipo_out2 b2;
```

```
23
24      //...
25      b1 = ris.fun(a1);
26      //...
27      b2 = ris.fun(a2);
28      //...
29  }
```

Il vincolo di mutua esclusione, ovviamente, è soddisfatto date le `entry` del monitor.

- **Modello a scambio di messaggi** (senza comandi con guardia)

```
1  /* TIPO MESSAGGIO ****************************** */
2  typedef struct {
3       enum (fun1, fun2) servizio;
4       //Tag che dice quale servizio
5
6       union {
7            tipo_in1 x1; //parametri fun1
8            tipo_in2 x2; //parametri fun2
9       } parametri;
10 } in_mess;
11
12 /* LATO SERVER ********************************** */
13 tipo_out1 fun1(tipo_in1 x1);
14 tipo_out2 fun2(tipo_in2 x2);
15
16 process server {
17      port in_mess input;
18      process p;
19      in_mess richiesta;
20      tipo_out1 y1;
21      tipo_out2 y2;
22
23      <eventuale inizializzazione>
24
25      while(true) {
26           p = receive(richiesta) from input;
27           switch(richiesta.servizio) {
28                case fun1: {
29                     y1 = fun1(richiesa.parametri.x1);
30                     send(y1) to p.risposta1;
31                     break;
32                }
33
34                case fun2: {
35                     y2 = fun2(richiesa.parametri.x2);
36                     send(y2) to p.risposta2;
37                     break;
38                }
39           }
40      }
41 }
42
43 /* LATO CLIENT ********************************** */
44 process client {
45      port tipo_out1 risposta1;
```

```
46      port tipo_out2 risposta2;
47      tipo_in1 a1;
48      tipo_in2 a2;
49      in_mess M;
50      tipo_out1 y1;
51      tipo_out2 y2M
52      process p;
53
54      <inizializzazione M in base al servizio scelto>
55
56      if(<servizio1> {
57           send(M) to server.input;
58           p = receive(b1) from risposta1;
59      } else {
60           send(M) to server.input;
61           p = receive(b2) from risposta2;
62      }
63      //...
64  }
```

Non essendoci possibilità di guardia, è previsto un solo canale per ricevere i comandi, indipendentemente da quale sia il servizio al quale si riferisce il generico messaggio: dunque, il servizio deve necessariamente essere codificato nel messaggio. Il vantaggio, ovviamente, è che **il server apre una sola porta** sulla quale ricevere messaggi di entrambi i tipi di servizi e, successivamente, analizza il contenuto dei messaggi ricevuti per poter eseguire il primo o il secondo servizio.

Questa soluzione è un po' complicata, un po' *naive* e poco scalabile dato che, per aggiungere un altro servizio, bisognerebbe complicarne di molto la struttura.

- **Modello a scambio di messaggi** (comandi con guardia)

```
1  /* LATO SERVER ********************************** */
2  tipo_out1 fun1(tipo_in1 x1);
3  tipo_out2 fun2(tipo_in2 x2);
4
5  process server {
6       port tipo_in1 input1;
7       pott tipo_in2 input2;
8       process p;
9       tipo_in1 x1;
10      tipo_in2 x2;
11      tipo_out1 y1;
12      tipo_out2 y2;
13
14      <eventuale inizializzazione>
15
16      do {
17           [] p = receive(x1) from input1; ->
18                y1 = fun1(x1);
19                send(y1) to p.risposta1;
20
21           [] p = receive(x2) from input2; ->
22                y2 = fun2(x2);
23                send(y2) to p.risposta2;
24
```

```
25      }
26  }
27
28  /* LATO CLIENT ********************************** */
29  process client {
30      port tipo_out1 risposta1;
31      port tipo_out2 risposta2;
32      tipo_in1 a1;
33      tipo_in2 a2;
34      tipo_out1 y1;
35      tipo_out2 y2M
36      process p;
37
38      <inizializzazione a1 o a2>
39
40      if(<servizio1> {
41          send(a1) to server.input;
42          p = receive(b1) from risposta1;
43      } else {
44          send(a2) to server.input;
45          p = receive(b2) from risposta2;
46      }
47      //...
48  }
```

In questo caso il server è realizzato con una guardia ripetitiva e due canali, uno per servizio: in questo caso, però, non si sono condizioni da verificare e, quindi, si può assumere che la guardia logica sia sempre vera e questo è esprimibile omettendola.

Risorsa condivisa con più operazioni e condizione di sincronizzazione

Il problema è complicato aggiungendo al caso precedente delle **condizioni di sincronizzazione**: un'operazione potrebbe non essere eseguita perché una condizione, ritenuta necessarie per la sue esecuzione, non è soddisfatta. Dunque, non è più sufficiente il solo vincolo di mutua esclusione, ma bisogna imporre un'ulteriore condizione.

- **Modello a memoria comune**
  Nel monitor, lo strumento per forzare un processo ad attendere nel caso in cui la condizione non sia verificata è la `condition`. Supponiamo, inoltre, che la politica sia di tipo *signal and wait*.

```
1   condition c1, c2;
2
3   entry tipo_out1 op1(tipo_in1 x1) {
4       //...
5       if(!cond1) wait(c1);
6       //...
7       signal(c2);
8       //...
9   }
10
11  entry tipo_out2 op2(tipo_in2 x2) {
12      //...
13      if(!cond2) wait(c2);
```

```
14      //...
15      signal(c1);
16      //...
17  }
```

Le `signal` sono state messe solo a significare che prima o poi qualcuno dovrà risvegliare i processi che sono sospesi sulle condition, ma non è detto che siano in quella posizione.

- **Modello a scambio di messaggi**

  La risorsa è gestita dal servitore che offre due servizi, una per ogni operazione. In questo caso, per imporre la condizione di sincronizzazione, la scelta migliore è l'impiego di un comando con guardia ripetitivo che, però, fa uso di **guardia logica**. Il codice è lo stesso del caso precedente a cui, però, è aggiunta l'espressione booleana nella guardia.

```
1  /* LATO SERVER ********************************** */
2  tipo_out1 fun1(tipo_in1 x1);
3  tipo_out2 fun2(tipo_in2 x2);
4
5  process server {
6       port tipo_in1 input1;
7       pott tipo_in2 input2;
8       process p;
9       tipo_in1 x1;
10      tipo_in2 x2;
11      tipo_out1 y1;
12      tipo_out2 y2;
13
14      <eventuale inizializzazione>
15
16      do {
17           [] (cond1); p = receive(x1) from input1; ->
18                y1 = fun1(x1);
19                send(y1) to p.risposta1;
20
21           [] (cond2); p = receive(x2) from input2; ->
22                y2 = fun2(x2);
23                send(y2) to p.risposta2;
24
25      }
26 }
27
28 /* LATO CLIENT ********************************** */
29 process client {
30      port tipo_out1 risposta1;
31      port tipo_out2 risposta2;
32      tipo_in1 a1;
33      tipo_in2 a2;
34      tipo_out1 y1;
35      tipo_out2 y2M
36      process p;
37
38      <inizializzazione a1 o a2>
39
```

```
40      if(<servizio1> {
41          send(a1) to server.input;
42          p = receive(b1) from risposta1;
43      } else {
44          send(a2) to server.input;
45          p = receive(b2) from risposta2;
46      }
47      //...
48  }
```

A tempo di esecuzione, quando il server esegue una generica iterazione del comando con guardia, succede quello che abbiamo visto nel caso generale (valutazione delle guardie, ecc...). È auspicabile prevedere dei rami incondizionati.

Corrispondenza tra monitor e processi servitori nella gestione delle risorse

La figura 3.43 mostra la corrispondenza tra il monitor nel modello a memoria comune e i processi servitori nel modello a scambio di messaggi per la condivisione di risorse soggette a vincolo di mutua esclusione.

MONITOR <=> PROCESSI SERVITORI

MEMORIA COMUNE		SCAMBIO DI MESSAGGI
Risorsa condivisa: monitor	<=>	Risorsa condivisa: struttura dati locale al server
Identificatore operazione di accesso al monitor	<=>	Porta di accesso del server
Tipi dei parametri della funzione	<=>	Tipo della porta
Tipo del valore restituito della funzione	<=>	Tipo della porta da cui il cliente riceve il risultato
Per ogni operazione del monitor	<=>	Un ramo (comando con guardia) dell'istruzione ripetitiva che costituisce il corpo del server

Figura 3.43: Tabella di corrispondenza monitor <=> processi servitori

Nel caso di una condizione di sincronizzazione, la tabella deve essere espansa con quella in figura 3.44.

Esempi

Esempio: ***Gestione di un pool di risorse equivalenti***

L'esempio è lo stesso fatto nel caso del modello a memoria ma, in questo caso, ogni risorsa è associata ad un processo gestore che, dunque, ne autorizza o meno l'accesso da parte di eventuali richiedenti, i clienti. L'ipotesi, quindi, è quello di avere n risorse identiche che devono essere gestite:

$R_1, R_2, \ldots, R_n \rightarrow P_1, P_2, \ldots, P_n$ *Un server gestisce il pool e ha il compito di allocare le risorse: al momento in cui un cliente richiede una risorsa, il server la*

MONITOR <=> PROCESSI SERVITORI

MEMORIA COMUNE		SCAMBIO DI MESSAGGI
Condizione di sincronizzazione di una entry	<=>	Guardia logica del ramo corrispondente
Chiamata di una funzione entry	<=>	Invio della richiesta (send) sulla porta del server e attesa del risultato (receive) sulla propria porta
Mutua Esclusione	<=>	Scelta di uno dei rami con guardia valida
Corpo della funzione	<=>	Istruzione del ramo corrispondente alla funzione

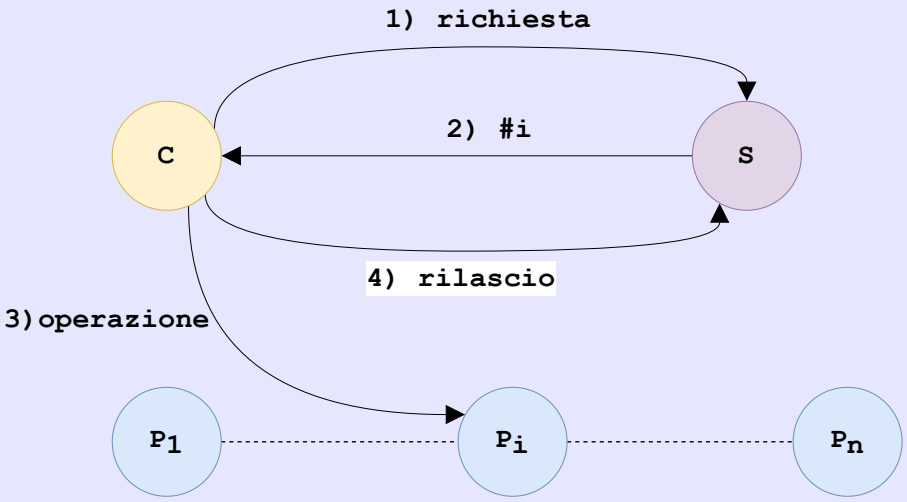
Figura 3.44: Tabella di corrispondenza monitor <=> processi servitori con condizioni di sincronizzazione

individua e comunica al processo che ne sta facendo richiesta il riferimento al processo specifico gestore di quella particolare risorsa.

Dunque, bisogna prevedere:

- ***un processo gestore per ogni risorsa****;*
- ***un processo server*** *nel quale è implementata la logica di allocazione delle risorse;*
- ***un certo numero di clienti*** *che si rivolgono inizialmente al server per richiedere l'allocazione e, con la successiva indicazione di quest'ultimo, potranno comunicare al relativo processo gestore l'operazione da compiere, per poi rilasciare la risorsa.*

Il protocollo da seguire, allora, è il seguente:




```
/* CLIENTE ************************************************** */
process cliente {
     port int risorsa;
     signal s;
     //signal = tipo predisposto per messaggi a contenuto
     //         informativo nullo
```

```
7       int r;
8       process p, server;
9
10      //...
11      send(s) to server.richiesta; //richiesta di una risorsa
12      //N.B.: le send sono di tipo asincrono
13
14      p = receive(r) from risorsa;//attesa della risposta
15      //A questo punto il cliente ha ricevuto dal server la
16      //autorizzazione per utilizzare la risorsa
17
18      <uso della risorsa r-esima>
19      send(r) to server.rilascio;//rilascio della risorsa
20      //...
21  }
22
23  /* SERVER ************************************************** */
24  process server {
25      port signal richiesta;
26      port int rilascio;
27      int disponibili = N;
28      boolean libera[N];
29      process p;
30      signal s;
31      int r;
32
33      //Inizializzazione
34      for(int i=0; i<N; i++)
35          libera[i] = true;
36
37      do {
38          [] (disponibili>0); p = receive(s) from richiesta ->
39              int i = 0;
40              while(!libera[i])
41                  i++;
42              libera[i] = false;
43              diponibili--;
44              send(i) to p.risorsa;
45          [] p = receive(r) from rilascio ->
46              disponibili++;
47              libera[r] = true;
48      }
49  }
```

Si specifica che, di fatto, le richieste sono tutte equiprioritarie, servite nell'ordine in cui arrivano.

Esempio: ***Gestione di un pool di risorse equivalenti con priorità***

Supponiamo che, al caso precedente, venga applicata la seguente variazione che, sostanzialmente, consiste nell'adozione di una politica basata su priorità nell'allocazione delle risorse ai clienti: ogni processo, quindi, porta con se un valore di priorità e il server dovrà privilegiare quelli con la priorità massima.

Ai fini dell'esempio, la priorità sarà un identificatore in cui lo 0 ha priorità massime a, più l'identificatore aumenta, più la priorità decresce. Allora, al momento del rilascio di una risorsa, il server dovrà

- *capire se ci sono processi sospesi;*
- *se si, capire qual è quello di maggior priorità;*
- *assegnare la risorsa a quello di priorità massima.*

Il cliente rimane uguale al caso precedente: la logica di allocazione, infatti, è interna al servitore.

```
/* CLIENTE ************************************************** */
process cliente {
     port int risorsa;
     signal s;
     //signal = tipo predisposto per messaggi a contenuto
     //         informativo nullo
     int r;
     process p, server;

     //...
     send(s) to server.richiesta; //richiesta di una risorsa
     //N.B.: le send sono di tipo asincrono

     p = receive(r) from risorsa;//attesa della risposta
     //A questo punto il cliente ha ricevuto dal server la
     //autorizzazione per utilizzare la risorsa

     <uso della risorsa r-esima>
     send(r) to server.rilascio;//rilascio della risorsa
     //...
}

/* SERVER ************************************************** */
process server {
     port signal richiesta;
     port int rilascio;
     int disponibili = N;
     boolean libera[N];
     process p;
     signal s;
     int r;
     int sospesi = 0;
     boolean bloccato[M];
     process client[M];

     //Inizializzazione
     for(int i=0; i<N; i++)
          libera[i] = true;
     for(int j=0; i<M; i++)
          bloccato[j] = false;
     client[0] = "P0"; ... client[M-1] = "PM-1"

     do {
          [] p = receive(s) from richiesta ->
```

```
45                  if(disponibili > 0) {
46                        int i = 0;
47                        while(!libera[i])
48                              i++;
49                        libera[i] = false;
50                        diponibili--;
51                        send(i) to p.risorsa;
52                  } else {
53                        int j=0;
54                        sospesi++;
55                        while(client[j]!=p)
56                              j++;
57                        bloccato[j] = true;
58                  }
59
60            [] p = receive(r) from rilascio ->
61                  if(sospesi == 0) {
62                        disponibili++;
63                        libera[r] = true;
64                  } else {
65                        int i = 0;
66                        while(!bloccato[i])
67                              i++;
68                        sospesi--;
69                        bloccato[i] = false;
70                        send(r) to client[i].risorsa;
71                  }
72      }
73  }
```

Il protocollo della soluzione appena proposta prevede un server che riceva le richieste di allocazione delle risorse tuttavia non è detto che la richiesta possa essere soddisfatta subito: bisogna verificare che ci siano risorse libere. Se c'è almeno una risorsa disponibile, allora si procede come nel caso precedente, altrimenti viene "sospeso"[a] il processo, registrando questo evento. Di conseguenza, quando viene rilasciata una risorsa, si verifica se c'è un processo sospeso e, se così fosse, si estrae quello più prioritario e gli si alloca la risorsa appena rilasciata.

[a] Si ricorda che il processo rimane in attesa sulla propria `receive` in quanto, se non ci sono risorse, il server non risponde.

Esempio: ***Scambio di dati molti-a-uno***

Il problema in questione è quello dei produttori e consumatori nel caso di send asincrone. In questo caso, si assume in partenza di avere ***uno e un solo*** *consumatore: in questo, la soluzione è banale poiché è sufficiente che il consumatore apra una porta bufferizzata sula quale esegue una* `receive` *ogni volta che vuole leggere dal canale. La presenza del buffer, inoltre, permette ai consumatori di inviare un messaggio semplicemente facendo una* `send` *asincrona sul canale, che, essendo bufferizzato, ha capacità di accodamento del messaggio che, dunque, viene sfruttata.*
Il problema, però, viene complicato nel caso in cui vengano aggiunti più processi

consumatori: poiché la porta non può essere condivisa tra più consumatori è necessario un processo server ***smistatori*** *al quale i produttori si rivolgono ogni volta che vogliono inviare un messaggio e, specularmente, viene interpellato dai consumatori quando vogliono ricevere un messaggio.*
Lo smistatore avrà due porte:

- `dati`, *per ricevere i dati dai produttori, di tipo* `T` *(generico);*
- `pronto`, *per ricevere la notifica dei consumatori che sono pronti a ricevere il messaggio (i messaggi su queste porte sono di tipo* `signal`*).*

```
1  /* SMISTATORE ************************************************ */
2  process smistatore {
3       port T dati;
4       port signal pronto;
5       T messaggio;
6       processo prod, cons;
7       signal s;
8
9       while(true) {
10           cons = receive(s) from pronto;
11           //Sospensiva nel caso in cui non ci siano
12           //consumatori pronti
13
14           prod = receive(messaggio) from dati;
15           //Sospensiva se nessun produttore ha
16           //inviato messaggi
17
18           send(messaggio) to cons.dati;
19      }
20 }
```

In questo caso il programmatore non ha la possibilità di fissare in modo arbitrario la capacità del buffer, bensì questa è determinata implicitamente calla capacità del canale `dati`*.*

Esempio: ***Scambio di dati molti-a-uno con buffer limitato***

Consideriamo l'esempio di prime e supponiamo che si voglia imporre un ***limite superiore alla dimensione del buffer*** *dei messaggi: ovvero, i messaggi inviati, ma non ancora ricevuti, non possono essere più di N. Allora, questo significa introdurre due ulteriori vincoli:*

- *il produttore non può inviare messaggi se il buffer è pieno (nella mailbox ci sono N messaggi inviati ma non ricevuti); a tal scopo è prevista una porta* `pronto_prod`*;*
- *il consumatore non può estrarre un messaggio se il buffer è vuoto e, in quest'ultimo caso, deve attendere; a tal scopo è stata prevista una porta* `pronto_cons`*.*

```
/* PRODUTTORE ************************************************ */
process produttore {
     port signal ok_to_send;
     T messaggio;
     process p;
     signal s;

     //...
     <produzione del messaggio>;
     send(s) to mailboc.pronto_prod;
     p = receive(s) from ok_to_send;
     send(messaggio) to mailbox.dati;
}

/* CONSUMATORE ************************************************ */
process consumatore {
     port T dati;
     T messaggio;
     process p;
     signal s;

     //...
     send(s) to mailbox.pronto_cons;
     p = receive(messaggio) from dati;
     <consuma il messaggio>;
}

/* MAILBOX ************************************************ */
process mailbox {
     port T dati;
     port signal pronto_prod, pronto_cons;
     T messaggio;
     process prod, cons;
     signal s;
     int contatore = 0;

     do {
          [] (contatore<N); prod = receive(s) from pronto_prod -> {
               contatore++;
               send(s) to prod.ok_to_send;
          }

          [] (contatore>0); cons = receive(s) from pronto_cons -> {
               prod = receive(messaggio) from dati;
               //Questa receive sicuramente NON E BLOCCANTE
               //in quanto ho controllato che contatore sia
               //maggiore di zero

               contatore--;
               send(s) to prod.ok_to_send;
          }
     }
}
```

In questo caso il programmatore non ha la possibilità di fissare in modo arbitrario

la capacità del buffer, bensì questa è determinata implicitamente calla capacità del canale `dati`.

3.4.4 Primitive di comunicazione sincrone

La `send` utilizzata finora era di tipo asincrono: considereremo, adesso, `send` di tipo sincrono in cui il processo che chiama la primitiva di invio si mette in attesa che il destinatario esegue la corrispondente `receive`.

In questo caso, allora:

- **il grado di concorrenza è ridotto** in quanto ogni `send` può causare la sospensione del processo per un tempo più o meno lungo;
- **hanno maggior espressività**, in quanto prevedono un punto di sincronizzazione per cui i messaggi inviati con questo tipo di primitiva possono essere associati allo stato attuale del processo;
- **non è più necessaria la presenza di un buffer** associato ad ogni canale (semplificazione di tipo realizzativo).

Esempio: ***Scambio di dati molti-a-uno con buffer limitato***

Nella semantica di tipo sincrono, la soluzione al problema della mailbox è simile al caso precedente tuttavia, essendo la `send` *sincrona, non c'è bisogno che lo smistatore invii una conferma di ricezione del messaggio: il consumatore, infatti, si sospende comunque in attesa che il messaggio sia stato depositato nella mailbox. Dunque, l'accettazione di una richiesta di invio sarà implicito nella* `receive`*; tuttavia, in questo caso non è possibile utilizzare il canale come buffer per cui è necessario prevedere una struttura dati, implementata come una coda di messaggi, che funzioni da buffer interno alla mailbox.*

```
1  /* CODA MESSAGGI ********************************************** */
2  interface coda_messaggi coda {
3       void inserimento(T mes); //inserisce un messaggio in coda
4       T estrazione(); //estrae il primo messaggio nella coda
5       boolean piena(); //coda piena => true
6       boolean vuota(); //coda vuota => true
7  }
8
9  /* PRODUTTORE ************************************************* */
10 process produttore {
11      T messaggio;
12      signal s;
13
14      //...
15      <produzione del messaggio>;
16      send(s) to mailbox.pronto_prod;
17      send(messaggio) to mailbox.dati;
18 }
19
20 /* CONSUMATORE ************************************************ */
21 process consumatore {
22      port T dati;
23      T messaggio;
```

```
24      process p;
25      signal s;
26
27      //...
28      send(s) to mailbox.pronto_cons;
29      p = receive(messaggio) from dati;
30      <consuma il messaggio>;
31  }
32
33  /* MAILBOX ************************************************** */
34  process mailbox {
35      port T dati;
36      port signal pronto_prod, pronto_cons;
37      T messaggio;
38      process p;
39      signal s;
40      coda_messaggi coda;
41      <inizializzazione>;
42
43      do {
44          [] (!coda.piena()); p = receive(s) from pronto_prod -> {
45              //La receive ha gia sbloccato il produttore
46              //dalla attesa
47              p = receive(messaggio) from dati;
48              coda.inserimento(messaggio);
49          }
50
51          [] (!coda.vuota()); p = receive(s) from pronto_cons -> {
52              //La receive ha gia sbloccato il consumatore
53              //dalla attesa...
54              messaggio = coda.estrazione();
55              send(messaggio) to p.dati;
56          }
57      }
58  }
```

Guardando il codice, si può osservare che, in quello del consumatore, sono presenti due send che potrebbero essere collassate in un'unica send. A questo punto, però, ***il protocollo diventa asimmetrico*** *nel senso che la mailbox apre una sola porta per il produttore, unicamente dedicata alla ricezione dei messaggi. Il codice, allora, diventa il seguente:*

```
1   /* PRODUTTORE ************************************************ */
2   process produttore {
3       T messaggio;
4
5       //...
6       <produzione del messaggio>;
7       send(messaggio) to mailbox.dati;
8   }
9
10  /* CONSUMATORE ************************************************ */
11  //Come prima...
12
13  /* MAILBOX ************************************************** */
```

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

      do {
            [] (!coda.piena()); p = receive(messaggio) from dati -> {
                  coda.inserimento(messaggio);
            }

            [] (!coda.vuota()); p = receive(s) from pronto_cons -> {
                  //La receive ha gia sbloccato il consumatore
                  //dalla attesa...
                  messaggio = coda.estrazione();
                  send(messaggio) to p.dati;
            }
      }
}
```

Esempio: ***Mailbox concorrenti***

Il problema dell'esempio precedente può anche essere risolto in un altro modo: al posto che utilizzare un buffer di dimensione N, si possono impiegare un insieme di N processi concorrenti, ognuno dei quali è dedicato alla gestione di un singolo elemento del buffer. A questo punto, la mailbox diventa un insieme di processi concorrenti collegati secondo uno schema a cascata (o a pipeline): ogni processo ha una struttura ciclica in cui, prima fa una `receive` *su una porta di ingresso, dopodiché prenderà il messaggio e lo invierà al canale di ingresso del processo successivo e si rimetterà in attesa sulla* `receive`*.*
I produttori, allora, invieranno i propri dati al canale di input del primo processo mentre i consumatori preleveranno i messaggi dall'ultimo canale dove l'ultimo processo li invia. Dunque, se l'ultimo processo della pipeline si blocca sulla send, allora anche gli altri processi si bloccheranno se i produttori continuano ad inserire messaggi: se sono tutti bloccati, allora `la pipeline sarà piena` *perché, infatti, ogni processo avrà in mano un messaggio che non può recapitare al successivo. Quando un consumatore eseguirà la* `receive`*, allora la situazione sarà sbloccata.*

3.4.5 Realizzazione dei meccanismi di comunicazione

In questa sotto-sezione parleremo di come il nucleo della macchina concorrente andrà ad implementare le primitive di sincronizzazione asincrona ma, soprattutto, i canali.

D'ora in avanti, si faranno le seguenti ipotesi semplificative:

- **tutti i messaggi scambiati tra i processi sono di un unico tipo** `T` predefinito a livello di nucleo (ad esempio, stringhe di byte di dimensione fissa);

- **tutti i canali sono di tipo porta** (molti-a-uno), associati univocamente al processo ricevente;
- **ogni porta deve disporre di un buffer** di lunghezza indefinita, fissata dal sistema (per le primitive asincrone).

Realizzazione delle primitive asincrone nel caso monoprocessore

La `send` di tipo asincrono può essere considerata una primitiva di più basso livello: infatti, con questo tipo di semantica è possibile implementarne altre di qualunque tipo; diversamente, con le send sincrone non è possibile fare l'inverso (o, se è possibile, è molto inefficiente).

Per questo motivo, **di base, tutti i sistemi concorrenti implementano le primitive con semantica di tipo asincrono** e, sulla base di questa, è possibile definire tutte le semantiche.

I tipi di dato e le operazioni necessarie a definire le astrazioni di alto livello per esprimere la comunicazione sono:

- **il messaggio**, realizzato come una struttura dati che incorpora il contenuto informativo e l'informazione sul mittente; poiché i messaggi saranno inseriti in code è necessario inserire anche il riferimento al messaggio successivo all'interno della coda:

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

- **la coda dei messaggi** che, banalmente, è una coppia di puntatori:

```
typedef struct {
     messaggio* primo;
     messaggio* utimo;
} coda_di_messaggi;
```

- **l'operazione di inserimento in coda**, che agisce su una coda di messaggi:

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

- **l'operazione di estrazione dalla coda**, che agisce sempre su una coda di messaggi:

```
messaggio* estrai(coda_di_messaggi c) {
     messaggio* pun;
     pun = c.primo;
     c.primo = c.primo -> successivo;
     if(c.primo == null);
```

```
            c.ultimo = null;

      return pun;
}
```

- **l'operazione che verifica se la coda è vuota**, che agisce sempre su una coda di messaggi:

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

- **il descrittore di una porta**:

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

typedef des_porta* p_porta;
```

- **il descrittore del processo**, che viene rivisto per il modello a scambio di messaggi:

```
typedef struct {
      p_porta porte_processo[M];
      //Questo campo consente di individuare
      //le porte p aperte dal processo

      PID nome;
      modalita_di_servizio servizio;
      tipo_contesto contesto;
      tipo_stato stato;
      //Registra se il processo risulta bloccato
      //e tiene traccia delle porte su cui il processo
      //sta attendendo (estensione rispetto al modello
      //a memoria comune

      PID padre;
      int N_figli;
      des_figlio prole[MAX_FIGLI];
      p_des successivo;
} des_processo
```

- **funzioni per il blocco del processo su una porta**:

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

void blocca_su(int ip) {
      <blocca il processo in esecuzione sulla porta di indice ip>
      //Si ricorda che il processo di esecuzione viene mantenuto
```

```
     //nella variabile processo_in_esecuzione del nucleo
}
```

- **funzioni la costruzione delle primitive di comunicazione**:

```
/* Funzione che verifica la presenza di messaggi su una porta
 * data (ip) del processo in esecuzione
 * => servira a costruire la receive */
void testa_porta(int ip) {
     p_des = esec = processo_in_esecuzione;
     p_porta pr = esec->porte_processo[ip];

     if(coda_vuota(pr->coda)) {
          //Non ci sono messaggi sulla porta, per cui il
          //processo viene bloccato in attesa sulla porta
          blocca_su(ip);
          assegnazione_CPU; //Cambio di contesto
          //Stessa funzione dello scheduler del caso a
          //memoria comune
     }
}

/* Funzione che estrae un messaggio dalla porta ip;
 * si suppone che prima che venga eseguita sia stata
 * chiamata testa_porta per verificare la presenza di
 * messaggi in coda sulla porta */
messaggio* estai_da_porta(int ip) {
     messaggio* m;
     p_des esec = processo_in_esecuzione;
     p_porta pr = esec->porte_processo[ip];
     m = estrai(pr->coda);
     return m;
}

/* Funzione che inserisce un messaggio m nella porta
 * ip del processo proc
 * => servira per la send */
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))
          //Controllo se il destinatario era rimasto
          //precedentemente in attesa sulla porta
          attiva(destinatario);
          //Porta il processo destinatario nello stato
          //di ready
}
```

Queste strutture dati e queste funzioni, allora, permettono di costruire `send` e `receive` asincrone. Segue il codice:

```
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);
     //Questa funzione blocca il processo sulla porta nel
     //caso in cui non ci siano messaggi. Il processo viene
     //riattivato quando qualcuno inserisce un messaggio
     //nella porta.

     m = estrai_da_porta(ip);
     *proc = m->mittente;
     *inf = m->informazione
}
```

Nel corso della trattazione sul modello a scambio di messaggi, più volte ci siamo soffermati sul fatto che è molto utile poter verificare se non presenti dei messaggi su più porte: a questo scopo avevamo introdotto i comandi con guardia. Vediamo, allora, come dovrebbe essere fatta la funzione `testa_porta` nel caso in cui si voglia ispezionare su più porte:

```
int testa_porte(int ip[], int n) {
     p_porta pr;
     int ris = -1;
     int indice_porta;
     p_des esec = processo_in_esecuzione;

     for(int i=0; i<n; i++) {
          indice_porta = ip[i];
          pr = esec->porte_porcesso[indice_porta];
          if(coda_vuota(pr->coda))
               blocca_su(indice_porta);
               //blocca_su NON SOSPENDE IL PROCESSO ma aggiorna
               //solo il campo stato del processo in esecuzione
               // => NON  viene eseguito un cambio di contesto
               // => Un processo puo essere bloccato su piu porte
               //    ma nonostante questo rimane in esecuzione

          else {
               ris = indice_porta;
               esec->stato = <processo attivo>;
               break;
          }
     }
     if(ris == -1);
          assegnazione_CPU();
          //In questo caso tutte le porte sono vuote per cui
          //bisogna indurre un cambio di contesto

     return ris;
     //Indica una delle porte monitorate in cui risulta presente
     //un messaggio
}
```

In realtà, se pensiamo alla semantica classica della `select` o della `do`, abbiamo detto più volte che la selezione avviene in modo non deterministico: qui, in realtà, non è proprio così in quanto si segue l'ordine dell'indice e, quindi, si sceglie la porta con indice minore in cui è presente un messaggio. Per realizzare il non determinismo, allora, **bisognerebbe, allora, scegliere un indice casuale** ma, per semplicità, si è scelto di prendere il primo.

La funzione simile alla `select`, allora, è la seguente:

```
int receive_any(T* inf, PID* proc, in ip[], int n) {
     messaggio* mes;
     int indice_porta;

     do
          indice_porta = testa_porte(ip, n);
     while(indice_porta == -1);
     mes = estrai_da_porta(indice_porta);
     proc = &(mes -> mittente);
     int = &(mes -> informazione);

     return indice_porta;
}
```

Realizzazione della comunicazione nel caso di architettura distribuite

Se i nodi di elaborazioni sono molteplici, allora la situazione si complica. Consideriamo un certo numero autonomo di nodi, con sistemi operativi diversi, connessi tra loro da una rete di comunicazione (*NOS*, Network Operating System): allora, dato che ogni sistema operativo opera in modo diverso dagli altri, non si può avere una gestione trasparente delle risorse **a meno di non collocare al di sopra dei sistemi operativi esistenti uno strato di middleware**.

L'alternativa a questo sono i **sistemi operativi distribuiti** (DOS): si ha una molteplicità di nodi., ognuno con una propria unità di elaborazione e un proprio nucleo, omogenei tra loro e tutti dotati dello stesso sistema operativo. A livello hardware, ogni nodo ha un'**interfaccia di rete** dotata di:

- un **canale di trasmissione**, ovvero il dispositivo che viene utilizzato per l'invio dei messaggi sulla rete;
- un **canale di ricezione**, ovvero il dispositivo che viene utilizzato per la ricezione dei messaggi provenienti da altri nodi all'interno della rete.

Lo scopo di questi sistemi è gestire tutte le risorse e processi nascondendo all'utente la loro distribuzione sulla rete.

Per fare questo, il modello implementativo appena analizzato va esteso e, in particolar modo, bisogna **estendere l'unità di trasferimento delle informazioni tra nodi diversi** in modo che essa contenga, oltre il messaggio, anche altre informazioni utili alla comunicazione.

Definizione 60: ***Pacchetto***

Nelle architetture distribuite, il pacchetto è l'unità di trasferimento delle informazioni trasmesse attraverso il canale, che contiene il messaggio da inviare e altre informazioni per il corretto recapito al destinatario.

La struttura del pacchetto dipende anche dal protocollo di rete che si sta utilizzando. Inoltre, nel canale di trasmissione è presente una coda dove vengono accodati i pacchetti che stanno per essere inviati nella rete: infatti, è probabile che più processi stiano contemporaneamente effettuando una `send`, ma il canale di trasmissione può gestire un pacchetto alla volta per cui, i pacchetti vengono accodati in modo tale che, man mano, possano essere gestiti per l'invio.

Oltre alla coda, nel canale di trasmissione è presente un buffer che contiene, volta per volta, il pacchetto oggetto del trasferimento. Dunque, se un processo vuole inviare un pacchetto:

1. il pacchetto viene inserito nella coda del canale di trasmissione;
2. quando sarà pronto, il gestore della coda metterà il pacchetto nel registro buffer;
3. il gestore attiva l'interfaccia di trasmissione per poter mandare il pacchetto sulla rete.

Il canale di trasmissione, invece, può essere visto come un dispositivo di input per l'acquisizione dei pacchetti che vengono ricevuti e, anche lui, avrà un registro buffer all'interno del quale, di volta in volta, viene depositato il pacchetto ricevuto dal canale.

I canali di trasmissione e ricezione interagiscono con il sistema operativo tramite il meccanismo delle interruzioni: ad esempio, ogni volta che si è compiuta una trasmissione, il canale di trasmissione notifica al sistema, tramite un'interruzione, questo evento. Il sistema, allora, interrompe quello che stava facendo e tramite un'apposita routine, andrà a gestire l'evento (analogamente per il canale di ricezione).

Come si è già detto, la struttura del canale dipende dal protocollo di rete, ma, in generale, contiene:

- **il messaggio** da trasmettere;
- **le informazioni relative al processo destinatario**;
- **la porta** di destinazione.

Per quanto riguarda le primitive, invece, supponendo di aver già costruito il pacchetto, per l'invio occorrono le seguenti funzioni:

- una funzione per l'invio, attraverso la quale si chiama in causa il canale di trasmissione:

```
void invia_pacchetto(packet p) {
     if(<canale di comunicazione occupato>
          packet_queue.inserisci(p);
          //packet_queue: unica coda del nodo per
          //              accodare i pacchetti

     else {
          <inserimento p nel registro buffer del canale>
          <attivazione trasmissione>
          //Al termine della trasmissione bisogna inviate
          //una interruzione
     }
}
```

Si ricorda che il controller ogni periferica dispone di un registro dati, un registro di controllo e un registro di stato: a proposito del discorso che si sta facendo, i registri interessati sono quello dati, in cui viene inserito il pacchetto, e quello di controllo, nel quale viene, di volta in volta, scritto il comando che la periferica dovrà eseguire. Nel registro di controllo, dunque, c'è un bit di start che attiva la trasmissione nel caso di invio del pacchetto, ovvero innesca un meccanismo puramente hardware che fa si che quello che è nel buffer venga inviato attraverso la rete.

- un funzione di routine, che viene messa in esecuzione ogni volta che si verifica l'interruzione del canale di trasmissione:

```
void tx_interrupt_handler() {
     packet p;
     salvataggio_stato();

     //Questa funzione viene attivata ogni volta che viene
     //completato un invio: allora bisogna verificare che
     //non ci siano altri pacchetti da inviare e, se ci sono,
     //bisogna procedere all invio...
     if(!packet_queue.vuota()) {
          p = packet_queue.estrai();
          <inserimento p nel registro buffer del canale>
          <attivazione trasmissione>
     }
     ripristino_stato();
}
```

Grazie a queste funzioni, dunque, è possibile realizzare le **primitive per l'invio**:

- una struttura per l'identificazione globale di un processo nella rete, che estende la definizione di PID data in precedenza:

```
typedef struct {
     int indice-nodo;
     int PID_locale;
} PID;
```

- una `send` remota per l'invio di pacchetti ad altri nodi della rete:

```
void remote_send(T inf, PID proc, int ip) {
     packet p;
     PID mit = processo_in_esecuzione;
     int indice_nodo_destinatario = proc.indice_nodo;

     <preparazione del pacchetto p>
     //In particolare, viene inserito nel campo del nodo del
     //destinatario il valore indice_nodo_destinatario
     //(ad es: IP per internet).
     //In aggiunta, vengono inseriti anche il nome del
     //mittente e i campi inf, proc e ip.
     //I campi dipendono dal protocollo!

     invia_pacchetto(p);
}
```

- una `send` unica che distingua il caso di invio messaggio ad un processo locale da quello di invio remoto:

```
void send(T inf, PID proc, int ip) {

     //nome_nodo contiene il nome del nodo locale
     if(proc.infice_nodo == nome_nodo)
          local_send(inf, proc, ip);
          //Stessa send vista nella sezione precedente
     else
          remote_send(inf, proc, ip);
}
```

Analizziamo, allora, il comportamento di un nodo destinatario alla ricezione di un pacchetto. A tal proposito, si costruiscono le **primitive per la ricezione**:

- la routine di gestione che viene innescata a seguito dell'interruzione da parte del controller del canale di ricezione, il quale, tramite l'interruzione, notifica al sistema operativo il fatto di aver completato un'operazione di ricezione:

```
void rx_interrupt_handler() {
     packet p;
     PID mit;
     T inf;
     PID proc;
     int ip;

     salvataggio_stato();

     <assegnamento a p del pacchetto ricevuto presente
      nel buffer>;
     <attivazione ricezione>;
     <estrazione dei campi del pacchetto p:
      - informazioni ricevute
      - nome del mittente (mit)
      - parametri della send (inf, proc, ip)>;

     messaggio* m = new messaggio;
     m->informazione = inf;
     m->mittente = mit;
     inserisci_porta(m, proc, ip);
     ripristino_stato();
}
```

Sostanzialmente, allora, questa routine riceve il pacchetto e poi lo spacchetta, ricostruendo localmente il messaggio e inserendolo, con la stessa funzione locale `inserisci_porta`, progettata nelle sezioni precedenti, nella porta del processo locale. **L'implementazione della `receive` non cambia** dato che, come descritto da questa funzione, il messaggio viene comunque prelevato dalle porte locali del processo.

Realizzazione delle primitive sincrone mediante primitive asincrone

Abbiamo già anticipato che con la semantica di tipo asincrona è possibile rappresentare tutte le altre semantiche, per cui anche quella sincrona. Vediamo, ora, come implementare

`send` e `receive` sincrone in un sistema che implementa le primitive asincrone (chiamate, ora, `a_send` e `a_receive`) viste nella sezione precedente:

```
void send(T inf, PID proc, int ip) {
     signal s;
     a_send(inf, proc, ip);
     a_receive(s, proc, ack);
}
```

La `send` sincrona, allora, è implementata come una normale send asincrona seguita subito dopo da una receive, che costringe il processo ad attendere la ricezione di un messaggio `ACK` con il quale il destinatario notifica la ricezione del messaggio precedente. Dunque, per quanto riguarda la `receive`:

```
void receive(T &inf, PID &proc, int ip) {
     signal s;
     a_receive(inf, proc, ip);
     a_send(s, proc, ack);
}
```

3.4.6 Sincronizzazione estesa

La sincronizzazione estesa è il terzo tipo di semantica della `send`.

Definizione 61: ***Sincronizzazione estesa***

Meccanismo di comunicazione (semantica) che prevede che il processo mittente, nell'inviare un messaggio al destinatario, si sospenda non solo fino a quando il destinatario ha ricevuto il messaggio ma anche finché quest'ultimo non ha terminato di eseguire l'operazione relativa a quel messaggio che, dunque, nel caso della sincronizzazione estesa, è una richiesta di servizio a chi lo riceve.

Dunque, **chi riceve il messaggio lo interpreta come la richiesta di esecuzione di un'operazione eseguita localmente**. Per cui, il mittente è anche un cliente e il destinatario è un server: il cliente, allora, si sospende fintanto che il server non avrà completato l'esecuzione del servizio che gli è stato richiesto.

In generale, c'è una **forte analogia con una normale chiamata di funzione**, com'è facilmente intuibile, con la differenza che chiamante e chiamato sono due processi concorrenti e la funzione viene eseguita *remotamente* rispetto al processo chiamante. Questo meccanismo, spesso, è detto **chiamata di procedura remota** anche se verranno distinti due modelli di implementazioni diversi per questa semantica, tra loro profondamente diversi:

- **chiamata di procedura remota** (RPC):
  Meccanismo che prevede un processo cliente che, ad un certo punto della sua esecuzione, richiede da un server l'esecuzione di una determinata operazione attraverso un'opportuna sintassi. **Il server**, allora, è un processo la cui funzione è quella di **istanziare processi locali ad esso, ognuno dedicato al servizio di una particolare richiesta** (figura 3.45).

  Allora, localmente al server, ci possono contemporaneamente essere più thread, ognuno dedicato all'esecuzione di una specifica richiesta.

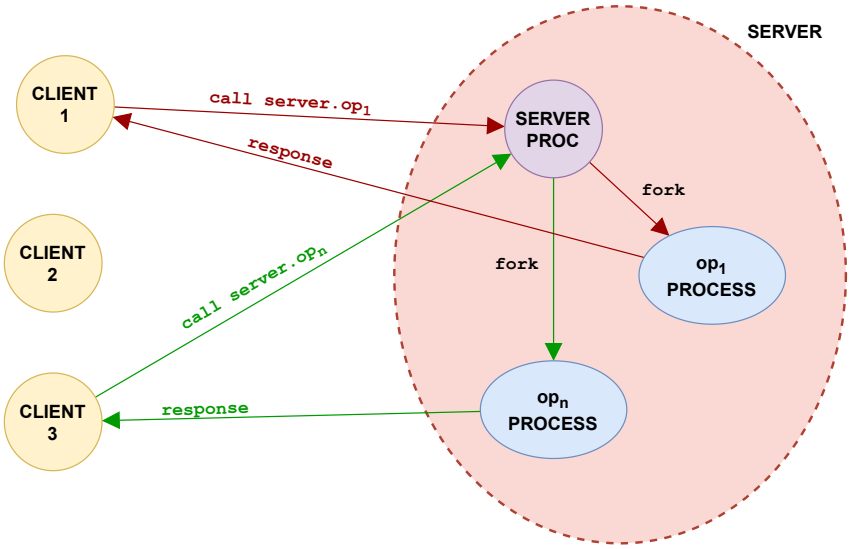


Figura 3.45: Esempio di interazioni RPC

Riassumendo, **RPC rappresenta solo un meccanismo di comunicazione** tra processi ed **implica necessariamente che il programmatore si occupi di sincronizzare** i processi servitori tra loro (politiche di gestione delle risorse condivise, mutua esclusione).

- **rendez-vous esteso**
  Meccanismo che prevede che, localmente al **servitore, strutturato in modo rigidamente sequenziale**, sia incapsulata una serie specifica di istruzioni che realizza l'esecuzione di un dato servizio. Dunque, tutti i servizi offerti dal servitore sono inseriti nel suo codice e, **a seguito di una richiesta, questo potrà *accettarla* grazie alla primitiva** `accept`: una particolare *receive*, anch'essa bloccante nel caso in cui venga chiamata su un servizio su cui, in quel momento, non ci sono richieste. In questo caso, a differenza degli altri modelli, il concetto di *canale* è rimpiazzato da quello di `servizio`: **il server**, infatti, non **si mette in attesa** su di un canale (o una serie di canali), bensì **sulle `accept`, che testano la presenza di richieste relative ad un dato servizio** (figura 3.46).

  I vincoli di sincronizzazione, ovviamente, sono gli stessi di RPC: il cliente rimane in attesa non solo fino a quando il serve farà la `accept`, ma anche finché questo non avrà completato l'operazione richiesta. A differenza di RPC, però, le richieste sono eseguite in modo rigidamente sequenziale, rendendo minimo il grado di concorrenza ma evitando, in particolar modo, la necessità di risolvere problemi di sincronizzazione lato server (non c'è competizione delle risorse). Pertanto **rendez-vous esteso è un meccanismo che combina la comunicazione con la sincronizzazione**.

RPC: semantica ed esempi

Per quanto riguarda RPC, ogni processo esegue un particolare componente software detto **modulo** che definisce uno spazio di indirizzamento con variabili e funzioni che il programmatore associa ad ogni diversa richiesta di servizio. I moduli, dunque, possono essere allocati su nodi diversi della rete.

```
module nome_del_modulo {
    <dichiarazione variabili locali>;
```

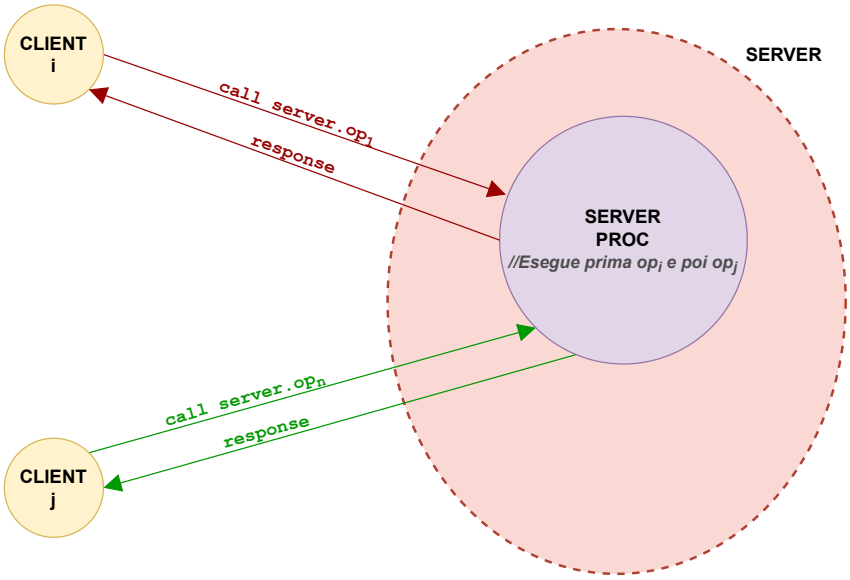


Figura 3.46: Esempio di interazioni rendez-vous esteso

```
     <inizializzazione variabili locali>;

     public void op1(<parametri formali>) {
          <corpo di op1>;
     }

     public void opn(<parametri formali>) {
          <corpo di opn>;
     }

     <dichiarazione procedure locali>
     <dichiarazione processi locali>
}
```

L'esecuzione di un particolare servizio `op1` di un modulo `nome_del_modulo` da parte del client, allora, verrà richiesta con uno statement del tipo:

```
call nome_del_modulo.opi(<parametri attuali>);
```

Il server, a seguito della chiamata, crea un thred che esegue `opi` nello spazio di indirizzamento definito da quel modulo.

Esempio: ***Servizio di sveglia***

Questo esempio mostra il perché sia necessario sincronizzare le risorse condivise nel server. Nello specifico, l'esempio riguarda un server che offre servizi di temporizzazione ai propri clienti.
Il server è definito da un modulo `allarme` *in cui vengono definite tutte le variabili che servono a realizzare la temporizzazione e un servizio,* `richiesta_sveglia`*, invocabile da un cliente per sospendersi volontariamente per un arco di tempo specificato da un parametro* `timeout` *(una sorta di* `sleep`*).*

```
/* SERVER ************************************************** */
```

```
module allarme {
     int time; //Tempo corrente
     semaphore mutex = 1; //Mutua esclusione
     semaphore priv[N] = 0; //Semafori privati per la sospensione
                            // dei processi
     coda_richieste coda; //Struttura contenete le richieste di
                          // di sveglia (sveglia, id) pervenute

     public void richiesta_sveglia(int timeout, int id) {
          int sveglia = time + timeout;
          P(mutex)
          <inserimento sveglia e id nella coda di risveglio in
           modo da mantenere tale coda ordinata secondo valori
           non decrescenti di sveglia>;
          //Tiene traccia che l'utente id vuole essere svegliato
          //al tempo richiesto.
          V(mutex)
          P(priv[id]);
     }

     process clock {
          //Questo processo e' un demone che esegue sempre e che
          //si occupa di gestire il dispositivo temporizzatore
          //aggiornando di conseguenza il tempo corrente.
          //Tale processo, andando ad aggiornare il valore del
          //contatore, verifica ogni volta se nella coda c'e'
          //qualcuno da riattivare

          int tempo_di_sveglia;
          <avvia il clock>;

          while(true) {
               <attende interruzione, quindi riavvia il clock>;
               //Quando arriva la nuova interruzione allora
               //significa che e' passata un'unita' di tempo
               time++;

               P(mutex);
               tempo di sveglia =
                    <tempo minore di sveglia in coda>;

               while(time >= tempo_di_sveglia) {
                    <rimozione di tempo_di_sveglia e id
                     corrispondente nella coda>;
                     V(priv[id]);
               }

               V(mutex);
          }//Fine ciclo processo demone

     }
}

/* CLIENT ************************************************** */
//...
```

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

Rendez-vous esteso: semantica ed esempi

Abbiamo detto che, nel caso de rendez-vous esteso, i servizi vengono specificati nel codice del server come un insieme di istruzioni che può comparire in qualunque punto del codice del servitore tramite l'istruzione `accept`. Tramite la `accept`, il server accetta la richiesta di esecuzione di un particolare servizio; inoltre, **nulla vieta che siano più `accept` per uno stesso servizio**.

La sintassi della `accept` è la seguente:

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

con

- `in`: keyword per l'inizio dei parametri di input;
- `out`: keyword per l'inizio dei parametri di uscita;
- `{S1, ..., Sn}`: sequenza di istruzioni.

Allora, la `accept` verifica se c'è almeno una richiesta del serivizio specificato: se si, la richiesta viene accettata, insieme con i suoi parametri di ingresso ed uscita e, immediatamente dopo, viene eseguita la sequenza di istruzioni specificata dopo la freccia. Se, invece, non sono presenti richieste, la `accept` è sospensiva; specularmente, lato cliente, se questo effettua la richiesta prima che il servitore chiami la `accept`, allora si sospenderà fino a quando il server non sarà pronto a riceverla. Inoltre, **le richieste vengono inserite in una coda associata al servizio lato servitore**, gestita con politica FIFO, in modo tale da non perdere nessuna delle richieste di servizio. Per quanto detto prima, poi, ad uno stesso servizio possono essere associate più `accept` e **nulla vieta che queste abbiano sequenze di istruzioni diverse** (similmente al polimorfismo).

Il cliente, per eseguire la richiesta, può usare una sintassi analoga a RPC:

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

Come si può intuire, lo schema di comunicazione è di tipo **asimmetrico, molti-a-uno**, pur non avendo i canali in modo esplicito.

Considerando, infine, la natura sequenziale del server, è chiaro che ci potrebbero essere dei problemi nell'accettazione delle richieste, facilmente risolti con i comandi con guardia:

```
if
     []<stato1>; accept<servizio1>(in <par-ingresso>, out <par-uscita>)
          -> {S1_1, ..., S1_n}; ...

     []<stato2>; accept<servizio2>(in <par-ingresso>, out <par-uscita>)
          -> {S2_1, ..., S2_n}; ...
end
```

L'istruzione è chiamata `if` ma, in molti casi (come i ADA), è chiamata `select` e, ovviamente, è molto simile ai comandi con guardia alternativi. Ogni ramo, allora, ha la struttura simile a quella della `accept` appena descritta, con l'aggiunta di una condizione che corrisponde alla guardia logica e che stabilisce il tipo di guardia di un determinato ramo. **La differenza rispetto al comando con guardia alternativo è la presenza della `accept` al posto della `receive`** che, come abbiamo detto, implementa la semantica a sincronizzazione estesa.

Esempio: ***Produttore e consumatore con buffer a capacità limitata***

```
/* SERVER ************************************************** */
process buffer {
     messaggio buff[N];
     int testa = 0;
     int coda = 0;
     int cont = 0;

     do {
          //Comando con guardia ripetitiva

          [](cont<N); accept inserisci(in dato:messaggio) -> {
               buff[coda] = dato;
          }
          //La graffa significa che termina il punto di
          //sincronizzazione col cliente. Il client, allora,
          //puo' essere riattivato. Questo riduce i tempi
          //di attesa del cliente a cui non interessa
          //aspettare gli aggiornamenti dei contatori del
          //server. Le istruzioni che seguono NON sono
          //sincronizzate con il cliente.
          cont++;
          coda = (coda+1)%N;

          [](cont>0); accept preleva(out dato:messaggio) -> {
               dato = buff[testa];
          }
               cont--;
               testa = (testa+1)%N;
     }
}

/* PRODUTTORE ********************************************** */
process produttore-i {
     messaggio dati;
     for(;;) {
          <produci dati>;
          call buffer.inserisci(dati);
     }
}

/* CONSUMATORE ********************************************** */
process consumatore-j {
     messaggio dati;
     for(;;) {
          call buffer.preleva(dati);
          <consuma dati>;
```

```
        }
}
```

Rendez-Vous: selezione delle richieste

Nel modello a rendez-vous nella sincronizzazione estesa, può essere importante **avere la possibilità di selezionare le richieste non solo in base alla condizione logica ma anche in base al valore dei parametri della richiesta stessa**: questo è difficile da realizzare in modo efficiente poiché il parametro viene acquisito soltanto dopo che il server ha accettato la richiesta per cui, con gli strumenti attualmente in nostro posso, questa selezione è attuabile solo tramite due interazioni (prima il server riceve le informazioni e dopo, sulla base dei parametri, con un'ulteriore richiesta decide se servirla o meno).

Data l'inefficienza di un meccanismo a due interazioni, molti linguaggi di programmazione mettono a disposizione alcuni strumenti che consentono di selezionare le richieste sulla base dei parametri di ingresso, senza bisogno di una doppia `accept`.

Uno di questi strumenti è il **vettore di operazioni di servizio**, la cui limitazione, però, è quella di dover conoscere a priori quali sono i possibili valori del parametro.

Esempio: ***Sveglia***

Supponiamo di voler creare un server che fornisca la possibilità ai clienti di impostare la sveglia: l'allarme, in una prima fase viene impostato e, in una fase successiva, il cliente si mette in attesa che suoni la sveglia (similmente alla `alarm` *di Unix).*

```
/* SERVER ************************************************** */
process allarme {
     entry tick;
     entry richiesta_di_sveglia(in int intervallo);

     entry svegliami[first...last];
     //Vettore delle operazioni di servizio

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

     dati_di_risveglio tempo_di_sveglia[N];
     //Vettore delle richieste di servizio
     // = famiglia di entry (ADA)

     do {
          [] accept tick; -> {tempo++;}

          [] accept richiesta_di_sveglia(in int intervallo); -> {
               //Chiede al server di registrare le impostazioni di
                  un
               //timer che deve scadere dopo un numero di
                  intervalli
               //di tempo pari ad intervallo
```

```
                <inserimento tempo + intervallo ed intervallo in
                 tempo di sveglia in modo da mantenere tale vettore
                 ordinato secondo valori non decrescenti di
                    risveglio>;
          }

          [] (tempo == tempo_di_sveglia[1].risveglio);
                    accept
                       svegliami[tempo_di_sveglia[1].intervallo];
                       -> {
                //La richiesta puo' essere accettata SOLO se il
                   valore
                //specificato come indice corrisponde al valore di
                //intervallo specificato nel primo elemento di quel
                //tempo di sveglia

                <riordinamento del vettore tempo_di_sveglia>;
          }
     }
}

/* CLIENTE ************************************************** */
process cliente_i {
     //...
     allarme.richiesta_di_sveglia(T);
     //Imposta in modo preliminare l'allarme
     //senza sospendere il processo

     //...
     allarme.svegliami[T];
     //Attende il timeout. Le quadre fanno riferimento al fatto
     //che si utilizza un vettore di servizio.
     //La richiesta viene servita se e solo il valore T corrisponde
     //a quello corrente del tempo, altrimenti la richiesta viene
     //lasciata pendente
}
```

La sintassi utilizzata nel codice è simile a quella di ADA, per cui è stata utilizzata la keyword `entry` *che definisce un servizio esibito dal server nei confronti dei clienti. Per ogni entry sarà presente poi una relativa* `accept`.

Indice analitico

J

K

L

M

N

O

P

R

S

T

U

V