



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

INGEGNERIA INFORMATICA - M

Appunti di
Sistemi Operativi

2018/2019

Prof.ssa Ciampolini

supervised by
Giuseppe Scelzo

Indice

1 Introduzione alla virtualizzazione	7
1.1 Virtualizzazione di Sistema	7
1.2 Emulazione	8
1.2.1 Interpretazione	8
1.2.2 Ricompilazione dinamica	8
1.3 Cenni storici	9
1.4 Vantaggi della virtualizzazione	9
1.5 Realizzazione del VMM	9
1.5.1 VMM di sistema vs VMM ospitati	10
1.5.2 Supporto HW alla virtualizzazione	10
1.5.3 Realizzazione del VMM in architetture non virtualizzabili	11
1.6 Architetture virtualizzabili	11
1.6.1 Protezione nell'architettura x86	11
1.6.2 Funzionamento dei VMM nell'architettura x86 classica	12
1.7 XEN	12
1.8 Gestione di VM	14
1.8.1 Migrazione di VM	15
2 La Protezione nei Sistemi Operativi	17
2.1 Protezione: Modelli, Politiche e Meccanismi	17
2.1.1 Modelli	17
2.1.2 Politiche	17
2.1.3 Meccanismi	18
2.2 Dominio di protezione	18
2.2.1 Associazione tra processo e domino	18
2.3 Matrice degli accessi	19
2.3.1 Realizzazione matrice degli accessi	20
2.3.2 Revoca diritti di accesso	21
2.3.3 ACL vs CL	21
2.4 Sicurezza multilivello	21
2.4.1 Modello Bell-La Padula	22
2.4.2 Modello Biba	23
2.5 Architettura dei sistemi ad elevata sicurezza	23
2.6 Classificazione della sicurezza dei sistemi di calcolo	24
3 Programmazione Concorrente	25
3.1 Sistemi Multiprocessore	25
3.1.1 Distributed-Memory	26
3.2 Classificazione architetture	26
3.3 Tipi di applicazioni	27
3.4 Processi non sequenziali e tipi di iterazione	27
3.4.1 Processo sequenziale	28
3.5 Processo non sequenziale	28

3.5.1	Scomposizione di un processo non sequenziale	28
3.5.2	Interazione tra processi	28
3.6	Architetture e linguaggi per la programmazione concorrente	29
3.7	Architettura di una macchina concorrente	29
3.7.1	Architettura della macchina M	29
3.8	Costrutti linguistici per la specifica della concorrenza	30
3.9	Realizzazione delle primitive di creazione e terminazione dei processi	31
3.10	Programmi	34
3.10.1	Proprietà dei programmi	35
4	Modello a memoria comune	37
4.1	Meccanismo di controllo degli accessi	37
4.1.1	Compiti del gestore di una risorsa	38
4.1.2	Accesso a risorse	38
4.2	Regione critica	39
4.2.1	Esempio: Scambio di informazioni tra processi	39
4.3	Mutua Esclusione	39
4.3.1	Strumenti di sincronizzazione: il Semaforo	40
4.4	Esempi di utilizzo dei semafori	42
4.4.1	Semaforo di mutua esclusione	42
4.4.2	Semaforo evento	44
4.4.3	Semafori binari composti	45
4.4.4	Semafori condizione	46
4.4.5	Esempio di uso di semafori condizione: Gestione di un pool di risorse equivalenti	47
4.4.6	Semafori risorsa	49
4.4.7	Semafori privati: specifica di strategia di allocazione	50
4.5	Realizzazione dei semafori	53
5	Nucleo di un sistema multiprogrammato	55
5.1	Nucleo	55
5.2	Realizzazione del Nucleo (Architettura Monoprocessoore)	56
5.2.1	Descrittore del processo	56
5.2.2	Coda dei processi pronti	57
5.2.3	Descrittori liberi	57
5.2.4	Processo in esecuzione:	58
5.3	Funzioni del Nucleo	58
5.3.1	Funzioni del livello inferiore: Cambio di Contesto	59
5.3.2	Gestione del temporizzatore	59
5.3.3	Semafori (Ambiente monoprocessoore)	59
5.3.4	Meccanismo di passaggio dall'ambiente di nucleo all'ambiente dei processi e viceversa.	61
5.4	Realizzazione nucleo (Architettura Multiprocessoore)	62
5.4.1	Simmetric Multi Processing	62
5.4.2	Realizzazione SMP	63
5.4.3	Nuclei distinti	63
5.4.4	Realizzazione nuclei distinti	63
5.5	SMP vs Nuclei Distinti	65
6	Modello a scambi di messaggi	67
6.1	Canali di comunicazione	67
6.2	Primitive di comunicazione	69
6.2.1	Port	69
6.2.2	Send	69
6.2.3	Receive	69

6.2.4	Comando con guardia	70
6.2.5	Comando con guardia alternativo	70
6.2.6	Comando con guardia ripetitivo	71
6.3	Primitive Asincrone	71
6.3.1	Risorsa condivisa con una sola operazione	72
6.3.2	Risorsa condivisa con più operazioni	73
6.3.3	Risorsa condivisa con più operazioni e condizioni di sincronizzazione	74
6.4	Corrispondenza tra monitor e processi servitori	75
6.5	Esempi di processi servitori	76
6.5.1	Pool di risorse equivalenti	76
6.5.2	Simulazione di un semaforo	77
6.5.3	Pool di risorse con priorità	77
6.5.4	Produttori & Consumatori	78
6.6	Realizzazione delle primitive asincrone	80
6.6.1	Architetture mono e multielaboratore	80
6.7	Architetture distribuite	83
6.8	Primitive di comunicazione sincrone	85
6.8.1	Mailbox di dimensioni finite	85
6.9	Specifica di strategia di priorità	88
7	Linguaggio GO	91
7.1	Costrutti del linguaggio	91
7.1.1	Dichiarazione	91
7.1.2	Assegnazione	92
7.1.3	If	92
7.1.4	For	92
7.1.5	Switch	92
7.1.6	Funzioni	93
7.1.7	Tipi strutturati	93
7.2	Struttura dei programmi go	93
7.3	Concorrenza	94
7.3.1	Interazioni: canali	95
7.3.2	Uso del canale	95
7.3.3	Sincronizzazione padre-figlio	96
7.3.4	Funzioni & canali	96
7.3.5	Chiusura canale: close	97
7.3.6	Send asincrone	97
7.3.7	Comandi con guardia: select	97
7.4	Guardia logica	98
7.4.1	Esempio pool priorità	99
8	Comunicazione con sincronizzazione estesa	101
8.1	Chiamata di procedura remota	102
8.2	Rendez-vous	103
9	Linguaggio ADA	107
9.1	Tipi di dato	107
9.1.1	Tipi scalari	107
9.1.2	Array	108
9.1.3	Record	108
9.1.4	Puntatori: access	108
9.2	Istruzioni di controllo	108
9.2.1	Alternativa: <i>if</i>	108
9.2.2	Ripetizione : <i>loop</i>	109

9.2.3	Ripetizione: <i>for</i>	109
9.3	Task	109
9.4	Rendez-vous	110
9.4.1	Select	111
9.5	Esempio produttore / consumatore	114
9.6	Corrispondenza tra monitor e processi servitori in ADA	115
10	Algoritmi di Sincronizzazione Distribuiti	117
10.1	Proprietà desiderate	117
10.2	Algoritmi di Sincronizzazione	118
10.3	Algoritmi per la gestione del tempo	118
10.3.1	Algoritmo di Lamport	119
10.3.2	Lamport: aggiornamenti degli orologi	119
10.4	Mutua esclusione distribuita	120
10.4.1	Soluzione centralizzata	120
10.4.2	Algoritmo Ricart-Agrawala	121
10.4.3	Algoritmo token ring	122
10.4.4	Algoritmi di mutua esclusione a confronto	123
10.5	Algoritmi di elezione	124
10.5.1	Algoritmo Bully	124
10.5.2	Algoritmo di elezione ad Anello	125
11	Azioni atomiche	127
11.0.1	Consistenza dei dati	127
11.1	Realizzazione di Azioni Atomiche Proprietà Fondamentali	128
11.2	Serializzabilità	128
11.2.1	Proprietà di serializzabilità:	128
11.3	Protocollo per la serializzabilità: two phase lock protocol	129
11.4	Proprietà "tutto o niente"	130
11.4.1	Effetto domino	131
11.4.2	Operazione commit	131
11.4.3	Terminazione	132
11.5	Realizzazione della memoria stabile	133
11.6	Azioni Atomiche: caso monoprocesso	134
11.6.1	Descrittore di azione e primitive	135
11.6.2	Gestione degli oggetti	135
11.7	Azioni atomiche: caso multiprocesso	136
11.7.1	Modello a memoria comune	137
11.7.2	Modello a scambio di messaggi	138
11.7.3	Azioni atomiche multiprocesso e sistemi distribuiti	138
11.8	Azioni atomiche Nidificate	139
11.8.1	Esempio: Chiamata di procedura remota in sistemi distribuiti	140

Capitolo 1

Introduzione alla virtualizzazione

Dato un sistema caratterizzato da un insieme di risorse sia HW che SW, **virtualizzare il sistema** significa presentare all'utilizzatore una visione delle risorse diversa da quella reale. Ciò si ottiene introducendo un livello di in-direzione tra la visione logica (sistema virtuale) e la visione fisica delle risorse (sistema reale). L'obiettivo è quello di **disaccoppiare il comportamento delle risorse di un sistema di elaborazione dalla sua realizzazione fisica**.

Alcuni esempi di virtualizzazione sono:

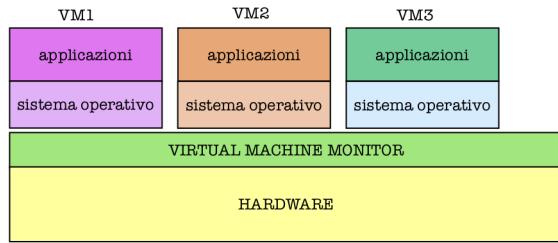
- **Virtualizzazione a livello di processo:** i processi multitasking permettono la contemporanea esecuzione di più processi, ognuno dei quali dispone di una VM dedicata. In questo caso il sistema vede la macchina come se fosse interamente dedicata a loro. Virtualizzazione realizzata dal kernel del sistema operativo.
- **Virtualizzazione della memoria:** in presenza della memoria virtuale, ogni processo vede uno spazio di indirizzamento di dimensioni indipendenti dallo spazio fisico effettivamente a disposizione. La virtualizzazione è realizzata dal kernel del sistema operativo. Questo meccanismo dà l'illusione a chi usa la memoria di avere a disposizione una memoria di grande di quella che c'è realmente sotto.
- **Astrazione:** un oggetto astratto (risorsa virtuale) è la rappresentazione di un oggetto (risorsa fisica). Questo disaccoppiamento è realizzato dalle operazioni (interfaccia) con le quali è possibile utilizzare l'oggetto.
- **Linguaggi di programmazione:** la capacità di portare lo stesso programma su architetture diverse è possibile grazie alla definizione di VM in grado di interpretare ed eseguire ogni istruzione del linguaggio, indipendentemente dall'architettura del sistema.

1.1 Virtualizzazione di Sistema

Una singola piattaforma HW viene condivisa da più elaboratori virtuali (macchine virtuali o VM) ognuno gestito da un proprio sistema operativo.

Il disaccoppiamento è realizzato da un componente chiamato **Virtual Machine Monitor** (VMM o Hypervisor) il cui compito è consentire la condivisione di parte di più VM di una singola piattaforma HW.

Ogni VM è isolata dalle altre poiché esegue all'interno di una sandbox in modo da non avere interferenze con le altre VM che eseguono sul sistema. Inoltre per non causare instabilità del sistema, le impostazioni di sicurezza da una VM non devono andare ad interferire con altre VM virtualizzate. Sulla stessa macchina posso andare a creare e definire più VM e ognuna appare all'utente come una macchina completa e utilizzabile con hardware virtuale e un suo sistema operativo su cui possono girare applicazioni.



Il VMM è il mediatore unico nelle iterazioni tra macchine virtuali e l'hardware sottostante e deve garantire isolamento tra le VM e stabilità del sistema.

1.2 Emulazione

Esecuzione di programmi compilati per una particolare architettura su un sistema di elaborazione dotato di un diverso insieme di istruzioni.

Vengono quindi emulate interamente le singole istruzioni dell'architettura ospitata permettendo a sistemi operativi o applicazioni pensati per determinate architetture di girare anche su architetture diverse.

L'approccio dell'emulazione ha seguito nel tempo due strade: l'interpretazione e la ricompilazione dinamica.

1.2.1 Interpretazione

Si basa sulla lettura di ogni singola istruzione del codice macchina che deve essere eseguito e sull'esecuzione di più istruzioni dell'host virtualizzante per ottenere lo stesso risultato. Produce però un sovraccarico molto elevato poiché possono essere necessarie molte istruzioni dell'host per interpretare una singola istruzione sorgente. Questo approccio risulta meno performante di un approccio compilato in cui prima si fa una traduzione preliminare e poi risulta molto più veloce a tempo di esecuzione.

1.2.2 Ricompilazione dinamica

Invece di leggere ogni singola istruzione, legge interi blocchi di codice, li analizza e li traduce per la nuova architettura ottimizzandoli e infine li mette in esecuzione. Il codice quindi viene prima tradotto e ottimizzato utilizzando tutte le possibilità offerte dalla nuova.

Tutti i più noti emulatori (es: QEMU , Virtual PC, Mame) utilizzano questa tecnica per implementare l'emulazione.

- **QEMU** è un software che implementa un particolare sistema di emulazione che permette di ottenere un'architettura nuova e disgiunta in un'altra che si occuperà di ospitarla permettendo di eseguire programmi compilati su architetture diverse. Utilizza la tecnica della traduzione dinamica.
- **Virtual PC** è un software di emulazione che consentiva a computer con sistema operativo Windows o Mac OSX l'esecuzione di sistemi operativi diversi come altre versioni Windows o Linux. Permetteva l'interoperatività tra sistemi Windows e Mac quando la differenza tra i due sistemi non lo permetteva naturalmente.
- **MAME** è un software per personal computer in grado di caricare ed eseguire codice binario originale delle ROM dei videogiochi da bar emulando l'hardware tipico di quelle architetture. Poiché gli attuali PC hanno una potenza di calcolo nettamente superiore a quello dei primi processori da giochi questo software utilizza la tecnica dell'interpretazione senza compromettere l'efficienza.

1.3 Cenni storici

La virtualizzazione non è un concetto nuovo:

Anni 60 → IBM crea un sistema CP/CMS diviso in due livelli appunto CP e CMS. Il primo esegue direttamente sull'HW svolgendo il ruolo di VMM, il secondo invece è un sistema operativo interattivo e monoutente replicato su ogni MV.

Anni 70 → nascono i sistemi operativi multitasking

Anni 80 → l'evoluzione della tecnologia porta alla nascita dei microprocessori, quindi si passa da architetture basate su mainframe verso minicomputer e PC.

Anni 80/90 → i produttori di hardware abbandonano l'idea di supportare un concetto di virtualizzazione a livello architetturale seguendo il paradigma: "One application, one server"

Fine anni 90 → nuovi sistemi di virtualizzazione per architetture Intel x86

Anni 2010 → Cloud Computing

1.4 Vantaggi della virtualizzazione

La virtualizzazione comporta numerosi vantaggi:

- **Uso di più sistemi operativi:** permette di utilizzare uno o più sistemi operativi sulla stessa macchina fisica permettendo a un utente di avere più ambienti eterogenei.
- **Isolamento degli ambienti di sicurezza:** ogni VM definisce un ambiente di esecuzione separato (sandbox) da quelli delle altre garantendo isolamento degli ambienti di esecuzione.
- **Consolidamento HW:** possibilità di concentrare più macchine su un'unica architettura HW per un utilizzo più efficiente delle risorse fisiche.
- **Gestione facilitata delle macchine:** le VM possono essere create, configurate e amministrate in maniera molto semplice, permettendo anche di effettuare operazioni di migrazione a caldo tra macchine fisiche.

1.5 Realizzazione del VMM

Ogni VMM deve in generale offrire alle diverse VM le risorse che sono necessarie per il loro funzionamento come la CPU, la memoria e i dispositivi di I/O. Secondo Popek e Goldberg i requisiti per la realizzazione di un VMM sono i seguenti:

- **Ambiente di esecuzione per i programmi sostanzialmente identico a quello della macchina reale.**
- **Garantire un'elevata efficienza nell'esecuzione dei programmi,** quindi il VMM deve permettere l'esecuzione diretta delle istruzioni impartite dalle VM.
- **Garantire la stabilità e la sicurezza dell'intero sistema.** Il VMM deve rimanere sempre nel pieno del controllo hardware, poiché le VM non possono accedervi in modo privilegiato.

I parametri per classificare il VMM sono Livello e Modalità di dialogo.

LIVELLO si intende dove è collocato il VMM, e può essere di due tipi: **VMM di sistema** che esegue direttamente sopra l'HW dell'elaboratore oppure **VMM ospitato** che esegue come una qualunque applicazione sopra al sistema operativo esistente.

Tra il SO e la VMM al posto di una operazione privilegiata ci deve essere una Hypercall che permette di delegare alla VMM l'esecuzione della primitiva.

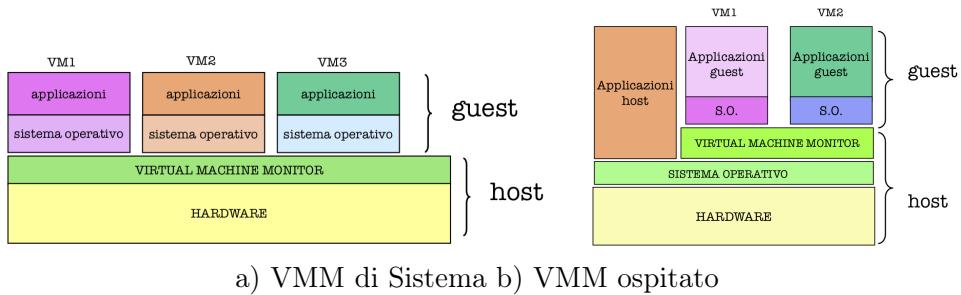
MODALITA DI DIALOGO per l'accesso alle risorse fisiche tra la macchina virtuale e il VMM; **Virtualizzazione pura**, cioè le VM usano la stessa interfaccia dell'architettura fisica o **Paravirtualizzazione**, cioè il VMM presenta un'interfaccia diversa da quella dell'architettura HW.

1.5.1 VMM di sistema vs VMM ospitati

In un VMM di sistema le funzionalità di virtualizzazione sono integrate in un sistema operativo leggero, costituendo un unico sistema posto direttamente sopra l'HW. (Vmware, xen, kvm).

Host: piattaforma di base sulla quale si realizzano le VM e comprende la macchina fisica, l'eventuale sistema operativo (solo nel caso di VMM ospitato) e il VMM.

Guest: la macchina virtuale. Comprende applicazioni e sistema operativo della VM.



Un VMM ospitato viene installato sopra un sistema operativo già esistente come un'applicazione, a questo punto il VMM opera nello spazio utente e accede all'HW tramite le system call del SO su cui è installato.

D'ora in poi faremo riferimento implicitamente riferimento a VMM di sistema!

L'architettura della CPU prevede in generale almeno due livelli di protezione, chiamati **ring**: supervisore e utente. Solo il VMM opera nello stato supervisore mentre il SO e le applicazioni della macchina virtuale operano nello stato utente. Si possono presentare due tipologie di problemi:

- **Ring deprivilegging:** il SO della VM esegue in uno stato che non gli è proprio (problema legato all'esecuzione delle system call). Le istruzioni privilegiate richieste dal SO nell'ambiente guest non possono essere eseguite.

Una possibile soluzione è l'approccio TRAP AND EMULATE: se il guest tenta di eseguire un'istruzione privilegiata, la CPU notifica un'eccezione al VMM (trap) e gli trasferisce il controllo, a quel punto il VMM controlla la correttezza dell'operazione richiesta e ne emula il comportamento (emulate). Un esempio è il tentativo di esecuzione dell'istruzione privilegiata popf per disabilitare le interruzioni da parte del SO guest (sarebbero disabilitati gli interrupt per tutto il sistema).

- **Ring compression:** se i ring utilizzati sono 2, applicazioni e SO della VM eseguono tutte e due allo stesso livello, il che comporta la necessità di protezione tra spazio del SO e delle applicazioni. In questo modo alcune istruzioni non privilegiate eseguite in modalità user permettono di accedere in lettura ad alcuni registri la cui gestione dovrebbe essere riservata al VMM.

1.5.2 Supporto HW alla virtualizzazione

Un'architettura si dice **naturalmente virtualizzabile** (o con supporto nativo alla virtualizzazione) se prevede l'invio di notifica allo stato supervisore per ogni operazione privilegiata eseguita da un livello di protezione diverso da quello supervisore.

In questo caso la realizzazione del VMM è semplificata proprio grazie al supporto nativo della virtualizzazione e sfruttando l'approccio TRAP AND EMULATE. Però non tutte le architetture sono naturalmente virtualizzabili ed in questi casi ci sono alcune istruzioni privilegiate eseguite a livello user che non provocano una trap ma vengono ignorate non consentendo l'intervento del VMM.

Ulteriore problema (**Ring Aliasing**): Alcune istruzioni non privilegiate, eseguite in modo user, permettono di accedere in lettura ad alcuni registri la cui gestione dovrebbe essere riservata al VMM; possibili inconsistenze. Esempio: registro CS, che contiene il livello di privilegio corrente (CPL).

1.5.3 Realizzazione del VMM in architetture non virtualizzabili

Se il processore non fornisce alcun supporto alla virtualizzazione allora è necessario ricorrere a soluzioni SW come ad esempio il FAST BINARY TRANSLATION e la PARAVIRTUALIZZAZIONE.

Con la **fast binary translation** il VMM scansiona dinamicamente il codice del SO guest (un pò come succedeva nella compilazione dinamica) prima dell'esecuzione per sostituire a run-time i blocchi contenenti istruzioni privilegiate. I blocchi tradotti sono poi eseguiti e conservati in una cache per eventuali riutilizzi. Con questo meccanismo ogni VM è una esatta replica della macchina fisica ma la traduzione dinamica è molto costosa.

Con la **Paravirtualizzazione** invece il VMM (Hypervisor) offre al SO guest un'interfaccia virtuale, le hypercall API, alla quale i SO guest devono fare riferimento per avere accesso alle risorse.

Ad esempio per ottenere un servizio che richiede l'esecuzione di istruzioni privilegiate, non vengono generate interruzioni per il VMM ma viene invocata la hypercall corrispondente.

I SO guest quindi devono essere modificati per avere accesso all'interfaccia del particolare VMM, la cui struttura è semplificata perché non deve più preoccuparsi di tradurre dinamicamente i tentativi delle VM.

La Paravirtualizzazione ha prestazioni migliori rispetto alla fast binary translation, ma necessita di porting dei SO guest.

1.6 Architetture virtualizzabili

L'uscita sul mercato di processori con supporto nativo alla virtualizzazione ha dato l'impulso allo sviluppo di VMM semplificati basati su virtualizzazione pure. Questo ha eliminato il ring compression/aliasing poiché il SO guest esegue in un ring separato (livello di protezione intermedio) da quello delle applicazioni. Inoltre ogni istruzione privilegiata richiesta dal SO guest genera un trap gestito dal VMM (in questo modo si gestisce anche il ring deprivilegging).

Come vantaggio non c'è bisogno della binary translation ma di contro le API presentate dall'hypervisor sono le stesse offerte dal processore.

1.6.1 Protezione nell'architettura x86

PRIMA GENERAZIONE → non avevano nessuna capacità di protezione e non facevano distinzione tra SO e applicazioni facendo girare entrambe con i medesimi privilegi. In questo modo le applicazioni potevano accedere direttamente ai sottosistemi di I/O, allocare memoria senza alcun intervento del SO.

SECONDA GENERAZIONE → viene introdotto il concetto di protezione, con la distinzione tra SO che possiede controllo assoluto sulla macchina fisica sottostante e le applicazioni, che possono interagire con le risorse fisiche solo facendone richiesta al SO. (concetto di ring di protezione).

Registro CS: i due bit meno significativi vengono riservati per rappresentare il livello corrente di privilegio **CPL (Livello Protezione Corrente)** utilizzando 4 diversi ring: ring 0 è quello dotato di maggiori privilegi e quindi destinato al kernel del SO mentre il ring 3 è il livello dotato di minor privilegi e quindi destinato alle applicazioni utente.

Nonostante ci siano 4 ring solitamente ne vengono utilizzati solo due: il ring 0 per il SO e il ring 3 per le applicazioni. Non è chiaramente permesso a ring diversi dal 0 di eseguire le istruzioni privilegiate che sono destinate solo al kernel del SO in quanto considerate critiche e potenzialmente pericolose. Segmentazione: ogni segmento è rappresentato da un descrittore in una tabella GDT o LDT; nel descrittore sono indicati il livello di protezione PL e i permessi di accesso R,W,X.

Protezione della memoria: una violazione dei vincoli di protezione provoca un'eccezione e questo può accadere se il CPL (Livello Protezione Corrente) è maggiore del PL (Livello di Protezione) del segmento di codice contenente l'istruzione invocata.

1.6.2 Funzionamento dei VMM nell'architettura x86 classica

Anche in questo caso è presente il problema del ring deprivilegging viene dedicato il ring 0 alla VMM e conseguentemente i SO guest vengono collocati in ring a privilegi ridotti. **Vengono comunemente utilizzate 2 tecniche** (seguono lo schema VMM/kernel guest/applicazioni):

- **0/1/3:** il SO viene spostato dal ring 0, dove nativamente dovrebbe trovarsi, al ring 1, lasciando le applicazioni al ring 3, mentre **al ring 0 viene installato il VMM**.

Il livello applicativo non può danneggiare il SO virtuale sul quale è in esecuzione (ring 1) come ad esempio andando a scrivere su porzioni di memoria ad esso dedicate . Le eccezioni generate sono catturate dal VMM sul ring 0 e passate al livello 1 dove è presente il SO.

- **0/3/3:** il SO viene spostato direttamente al livello applicativo, ring 3, dove si trovano anche le applicazioni, mentre sul ring 0 viene installato il VMM.

In questa modalità non è possibile generare eccezioni quindi devono essere intrapresi meccanismi molto sofisticati con un controllo continuo da parte del VMM.

Problemi: Ring aliasing

Alcune istruzioni non privilegiate, eseguite in uno stato utente permettono di accedere in lettura ad alcuni registri del sistema la cui gestione dovrebbe essere riservato solo al VMM. Il S.O. guest non si deve poter rendere conto su quale livello di protezione sta eseguendo perché a quel punto si renderebbe conto di eseguire su un VM, al ring 1 invece che allo 0.

Nell'architettura x86 esistono istruzioni privilegiate che, se eseguite in $\text{ring} > 0$, non provocano un'eccezione, ma vengono ignorate non consentendo quindi l'intervento trasparente del VMM , o in alcuni casi provocano il crash del sistema. (Esempio popf)

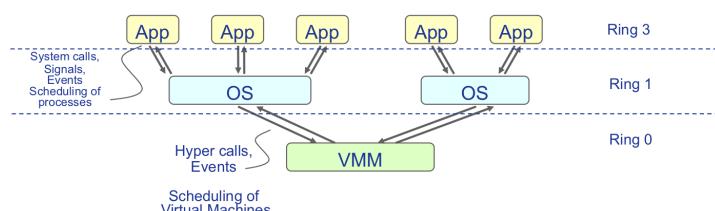
1.7 XEN

VMM open source che opera secondo i principi della Paravirtualizzazione.

Nell'architettura XEN il VMM viene chiamato Hypervisor e le VM vengono definite Domains, di cui una in particolare, chiamata Domain0, controlla tutto il sistema e gode di privilegi diversi rispetto a tutte le altre macchine. Il Domain0 deve essere sempre attivo per permettere il funzionamento di tutto il sistema, inoltre al suo interno può controllare i driver dei dispositivi.

Il VMM (Hypervisor) si occupa della virtualizzazione della CPU e della memoria e dei dispositivi per ogni VM. Dispone di un'interfaccia di controllo in grado di gestire la divisione di queste risorse tra i vari domini, però l'accesso a questa interfaccia è ristretto al solo Domain0.

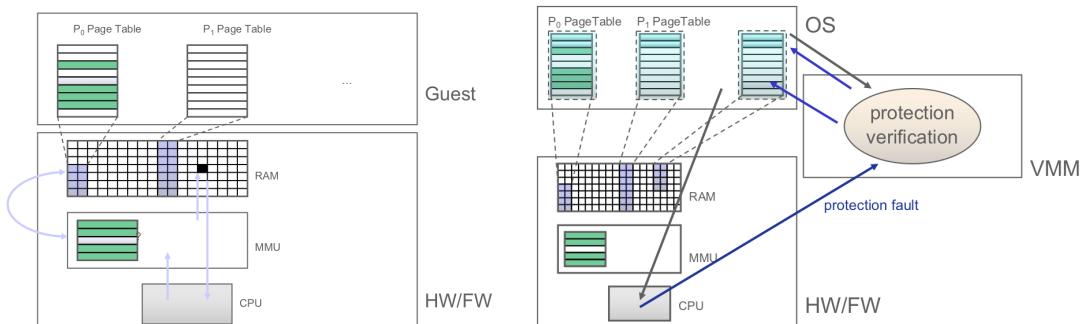
XEN adotta una configurazione dei ring come quella 0/1/3, inoltre le VM eseguono direttamente le istruzioni non privilegiate e l'esecuzione delle istruzioni privilegiate viene delegata al VMM tramite le hypercall.



Gestione della memoria i SO guest gestiscono la memoria virtuale mediante i tradizionali meccanismi di paginazione, adottando una soluzione in cui le tabelle delle pagine delle VM vengono mappate nella memoria fisica dal VMM, non possono essere accedute in scrittura dai kernel guest ma solo dal VMM e sono accessibili in modalità lettura anche dai guest. I SO guest si occupano della paginazione delegando al VMM la scrittura delle page table entries. La tabella delle pagine devono essere create e verificate dal VMM su richiesta dei guest.

Memory split XEN risiede nei primi 64 bit del virtual address space. Lo spazio di indirizzamento virtuale per ogni VM è strutturato in modo da contenere XEN e il kernel in segmenti separati.

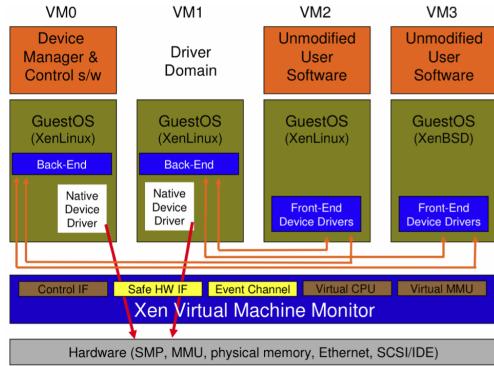
Creazione di un processo Il SO guest richiede una nuova tabella delle pagine al VMM: alla tabella vengono aggiunte le pagine appartenenti al segmento di Xen; Xen registra la nuova tabella delle pagine e acquisisce il diritto di scrittura esclusiva, poi ad ogni successiva update da parte del guest provocherà una **protection-fault** la cui gestione comporterà la verifica e l'effettivo aggiornamento della tabella delle pagine.



Balloon process La paginazione è a carico del guest, occorre quindi un meccanismo efficiente che consenta al VMM di reclamare ed ottenere in caso di necessità dalle altre VM pagine di memoria meno utilizzate. Su ogni macchina virtuale è in esecuzione un processo (Balloon process) che comunica con il VMM, questo processo è sempre attivo e viene chiamato in causa dal VMM in modo tale che richieda al proprio guest altre pagine. Questa richiesta provoca da parte del S.O. guest l'allocazione di nuove pagine al balloon process che una volta ottenute le cede al VMM.

Virtualizzazione della CPU il VMM definisce un'architettura virtuale simile a quella del processore nella quale però le istruzioni privilegiate sono sostituite da opportune hypercall (l'invocazione di una hypercall determina il passaggio da guest a xen, ring 1 → ring 0). Il VMM si occupa dello scheduling delle VM utilizzando il Borrowed Virtual Time algorithm che consente in caso di vincoli temporali stringenti di ottenere schedulazioni efficienti. Vengono quindi utilizzati due clock: real-time (tempo del processore) e virtual-time (associato alla VM e avanza solo quando la VM esegue).

Virtualizzazione I/O per motivi di indipendenza dall'HW delle singole VM si è deciso di condensare nel Domain0 i driver (**back-end driver**), mantenendo però in ogni singola VM un'interfaccia dei driver (**front-end drivers**) che si occupano di trasferire al vero e proprio driver la richiesta.



Nella soluzione adottata da XEN quindi avremo un back-end driver per ogni dispositivo e il suo driver è isolato all'interno della VM Domain0, con accesso diretto all'HW. Mentre ogni guest prevede un driver virtuale semplificato, cioè i front-end driver, che consentano l'accesso al device tramite il back-end driver contenuto nel Domain0. Questa soluzione comporta dei vantaggi come ad esempio la portabilità, l'isolamento e la semplificazione del VMM, ma al contrario necessita di una continua comunicazione con il back-end.

Gestione interruzioni e eccezioni la gestione delle interruzioni viene virtualizzata in modo molto semplice, cioè ogni interruzione viene gestita direttamente dal guest.

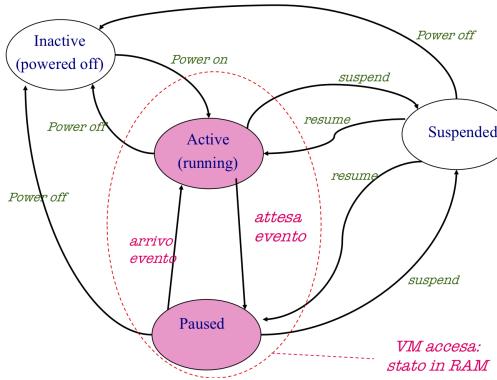
Un caso particolare riguarda il **Page Fault**, eccezione di tipo trap generata quando un processo cerca di accedere ad una pagina che è mappata nello spazio di indirizzamento virtuale, ma che non è presente nella memoria fisica. La gestione non può essere delegata completamente al guest, perché richiede l'accesso al registro CR2 contenente l'indirizzo che l'ha causato, ma è accessibile solo nel ring 0, quindi la gestione del page fault deve coinvolgere il VMM. La routine di gestione eseguita da XEN legge il contenuto di CR2 e lo copia in una variabile dello spazio del guest; successivamente viene trasferito il controllo al guest che andrà finalmente a gestire il page fault.

1.8 Gestione di VM

Il compito della VMM è quello di gestire le VM, compresa la creazione/allocazione, spegnimento/accensione, l'eliminazione e la migrazione live. In particolare quindi il VMM opera sulle VM con le stesse operazioni che un S.O. mette in atto per la gestione dei processi.

Una VM si può trovare nei seguenti stati:

- **Running:** la VM è accesa e occupa memoria nella RAM del server su cui è allocata;
- **Inactive:** la VM è spenta ed è rappresentata nel file system tramite un file immagine;
- **Paused:** la VM è in attesa di un evento;
- **Suspended:** la VM è stata sospesa dal VMM tramite il comando suspend; il suo stato e le risorse utilizzate sono salvate nel file system. L'uscita dallo stato di sospensione avviene tramite l'operazione resume da parte del VMM.



1.8.1 Migrazione di VM

In datacenter di server virtualizzati sempre più sentita la necessità di gestione agile delle VM per far fronte a quartazioni dinamiche del carico, la manutenzione online, la gestione finalizzata al risparmio energetico e la tolleranza ai guasti. Le VM possono essere spostate da un server a un altro senza essere spente e chi utilizza la VM in questione non si accorge minimamente che la macchina sia stata spostata.

Per effettuare una migrazione live basterebbe quindi effettuare una suspend di una VM sul nodo A ed effettuare una resume della stessa VM ma questa volta sul nodo B. In una migrazione live è desiderabile minimizzare il downtime cioè il tempo in cui la macchina non risponde alle richieste degli utenti, il tempo di migrazione e il consumo di banda.

La soluzione adottata da XEN per effettuare una migrazione live è quella della **Precopy** che si svolge in 6 passaggi:

- Pre-migrazione:** individuazione della VM da migrare e dell'host di destinazione
- Reservation:** viene inizializzata una VM sul server di destinazione
- Pre-copia** iterativa delle pagine: viene eseguita una copia di tutte le pagine allocate in memoria sull'host A per la VM da migrare sull'host B; successivamente vengono iterativamente copiate da A a B tutte le pagine modificate (perché la VM sull'host A è ancora attiva e quindi modifica dei file) fino a quando il numero di dirty pages (pagine modificate) è inferiore a una soglia.
- Sospensione:** la VM viene sospesa e il suo stato insieme alle dirty pages viene copiato da A a B
- Commit:** la VM viene eliminata dal server A
- Resume:** la VM viene attivata nel server B

In alternativa alla Precopy si può utilizzare una tecnica chiamata **Postcopy** in cui la VM viene sospesa e vengono copiate pagine e stato. Il tempo di migrazione è più basso ma il downtime è molto più elevato. Posso utilizzare questa tecnica solo se la VM non deve sempre reattiva poiché la macchina viene congelata e poi copiata.

In Xen il comando di migrazione viene eseguito da un demone di migrazione nel Domain0 del server di origine della VM da migrare. La migrazione sfrutta il protocollo di trasporto RDMA (Remote Direct Memory Access), ottimizzato allo scopo. Inoltre le pagine da copiare iterativamente vengono compresse per ridurre l'occupazione di banda durante il trasferimento.

Capitolo 2

La Protezione nei Sistemi Operativi

La **protezione** consiste nell'insieme di attività volte a garantire il controllo dell'accesso alle risorse logiche e fisiche da parte degli utenti.

La **sicurezza** riguarda l'insieme delle tecniche con le quali regolamentare l'accesso degli utenti al sistema di elaborazione. Perciò mentre con protezione ci si riferisce ai dati interni al sistema, con sicurezza ci si riferisce all'intero sistema.

2.1 Protezione: Modelli, Politiche e Meccanismi

Il controllo degli accessi è suddivisibile in 3 livelli: modelli, politiche e meccanismi.

Il livello più alto sono i **modelli** e si riferisce a quale modello di protezione il sistema operativo deve scegliere e mettere in atto, il livello intermedio è quello delle politiche e infine ci sono i meccanismi.

2.1.1 Modelli

Un modello di protezione definisce i **soggetti**, gli **oggetti** ai quali i soggetti hanno accesso ed i **diritti** di accesso.

- Gli oggetti costituiscono la parte passiva cioè le risorse fisiche e logiche alle quali si può accedere e su cui si può operare (ad esempio i file).
- I soggetti rappresentano la parte attiva di un sistema, cioè le entità che possono richiedere l'accesso alle risorse (ad esempio i processi che agiscono per conto degli utenti).
- I diritti di accesso sono le operazioni con le quali è possibile operare sugli oggetti. Un soggetto può avere diritti di accesso sia per gli oggetti che per altri soggetti.

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

2.1.2 Politiche

Le politiche di protezione definiscono le regole con le quali i soggetti possono accedere agli oggetti. Mentre il modello è qualcosa di insito nel sistema, le politiche in genere vengono scelte da chi opera su quel sistema. Esistono 3 diversi tipi di politiche:

- **DAC (Discretionary Access Control)**: il creatore di un oggetto controlla i diritti di accesso per quell'oggetto, quindi la gestione delle politiche è decentralizzata (UNIX fornisce un meccanismo per definire e interpretare per ciascun file i tre bit di *read*, *write* e *execute* per il proprietario del file, il gruppo e gli altri. L'utente definisce il valore dei bit per ogni oggetto di sua proprietà.).
- **MAC (Mandatory Access Control)**: i diritti di accesso vengono definiti in modo centralizzato. Questa soluzione viene utilizzata in sistemi ad alta sicurezza per garantire assoluta confidenzialità e i diritti vengono gestiti da un'entità centrale.

- **RABC (Role Based Access Control):** ad ogni ruolo sono assegnati specifici diritti di accesso alle risorse, quindi gli utenti possono avere diversi ruoli.

Principio del privilegio minimo: ad ogni soggetto sono garantiti i diritti di accesso solo agli oggetti strettamente necessari per la sua esecuzione (caratteristica desiderabile per tutte le politiche di protezione).

2.1.3 Meccanismi

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

- **Flessibilità del sistema di protezione:** i meccanismi di protezione devono essere sufficientemente generali per consentire l'applicazione di diverse politiche di protezione.
- **Separazione tra meccanismi e politiche:** la politica definisce "cosa va fatto" meccanismo invece definisce "come va fatto".

E' desiderabile la massima indipendenza tra le due componenti.

2.2 Dominio di protezione

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

$$D(S) = \{ \langle o, \text{diritti} \rangle \mid o \text{ è un oggetto, } \text{diritti} \text{ è un insieme di operazioni} \}$$

Un dominio di protezione è unico per ogni soggetto mentre un processo può eventualmente cambiare dominio durante la sua esecuzione. Il soggetto può accedere solo agli oggetti definiti nel suo dominio utilizzando i diritti specificati dal dominio.

Domini disgiunti o con diritti di accesso in comune: possibilità per due o più soggetti di effettuare alcune operazioni comuni su un oggetto condiviso; le operazioni vengono svolte da processi che operano per conto di soggetti (a alcuni sono associati i domini).

2.2.1 Associazione tra processo e dominio

L'associazione tra processo e dominio può essere statica o dinamica. Nel caso **statico** l'insieme delle risorse disponibili ad un processo rimane fisso durante tutto il suo tempo di vita, nel caso **dinamico** invece l'associazione tra processo e dominio varia durante l'esecuzione del processo.

L'insieme globale delle risorse che un processo potrà utilizzare non può essere un'informazione disponibile prima dell'esecuzione del processo stesso. Inoltre l'**associazione statica non è adatta nel caso di voglia limitare per un processo l'uso delle risorse a quello strettamente necessario** (principio del privilegio minimo). Utilizzando l'**associazione dinamica** invece si può mettere in pratica il principio del **privilegio minimo** cambiando dinamicamente il dominio quando necessario.

Esempio di cambio di dominio

Standard dual mode: due domini di protezione, quello dell'utente (user mode) e quello del kernel (monitor o kernel mode) → cambio di dominio associato alle **system call**: quando un processo deve eseguire una istruzione privilegiata (accesso ai file, alle funzioni di rete, generazione di thread ...) avviene un **cambio di dominio**.

Ciò non realizza la protezione tra utenti, ma solo tra kernel e utente; insufficiente per la multiprogrammazione.

Unix → dominio associato all'utente: il cambio dominio corrisponde al cambio temporaneo d'identità (UID) del processo.

A ogni file sono associati l'identificazione del proprietario (*user-id*) e un bit di dominio (*set-uid*). Quando un utente A (*user-id=A*) inizia l'esecuzione di un file P il cui proprietario è B (*user-id=B*) ed il file ha *set-uid=on*, al processo che esegue P viene assegnato lo *user-id* di B.

2.3 Matrice degli accessi

Un sistema di protezione può essere rappresentato a livello astratto utilizzando il modello della matrice degli accessi. Ogni riga della matrice è associata a un soggetto (utente), ogni colonna invece è associata a un oggetto (risorsa, file).

Il modello mantiene tutta l'informazione che specifica il tipo di accessi che i soggetti hanno per gli oggetti (stato di protezione). La matrice quindi offre ai meccanismi di protezione le informazioni che consentono di verificare il rispetto dei vincoli di accesso. Il meccanismo associato al modello:

- Ha il compito di verificare se una richiesta di accesso che proviene da un processo che opera in un determinato dominio è consentita o meno;
- Consente di modificare dinamicamente il numero degli oggetti e dei soggetti;
- Consente ad un processo di cambiare dominio durante l'esecuzione;
- Consente di modificare in modo controllato il cambiamento dello stato di protezione.

Verifica del rispetto dei vincoli di accesso

Il meccanismo consente di assicurare che un processo che opera nel dominio D_j possa accedere solo agli oggetti specificati nella riga i e solo con i diritti di accesso indicati. Quando un'operazione M deve essere eseguita nel dominio D_i sull'oggetto O_j , il meccanismo consente di controllare che M sia contenuta nella casella $access(i,j)$.

Modifica dello stato di protezione

Secondo la politica DAC gli utenti possono modificare lo stato di protezione, mentre nella politica MAC può essere fatto solo dall'entità centrale. La modifica controllata dello stato di protezione può essere ottenuta tramite un opportuno insieme di comandi:

- Create, delete object (aggiungere o eliminare le colonne della matrice, cioè gli oggetti)
- Create, delete subject (aggiungere o eliminare le righe della matrice, cioè i soggetti)
- Read, grant, delete, transfer access right (gestione diritti di accesso)

Propagazione diritti di accesso (Copy flag)

La possibilità di copiare un diritto di accesso per un oggetto da un dominio ad un altro della matrice di accesso è indicato con un asterisco (*) che rappresenta il **copy flag**.

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

L'operazione di propagazione può essere realizzata in due modi: **trasferimento** del diritto, quindi il soggetto S_1 trasferisce $read^*$ e perde il diritto per l'oggetto O_1 , oppure per **copia** del diritto, quindi viene copiato solo $read$ mentre il soggetto S_1 mantiene $read^*$.

DIRITTO OWNER chi possiede tale diritto può assegnare/revocare un qualunque diritto di accesso sull'oggetto X di cui è owner, ad un qualunque altro soggetto.

Ad esempio se S_2 ha il diritto owner su O_2 allora può concedere il diritto write su O_2 al soggetto S_1 .

DIRITTO CONTROL chi possiede tale diritto può revocare un qualunque diritto di accesso per l'oggetto X al soggetto Sj su cui ha i diritti di control.

Ad esempio se S1 possiede i diritti di control su S2 e S2 ha il diritto di write su O2 allora S1 può revocare il diritto di write di S2 su O2.

DIRITTO SWITCH un processo che esegue nel dominio del soggetto Si può commutare al dominio di un altro soggetto Sj. Questa operazione è consentita solo se il diritto di switch appartiene a A[Si,Sj].

Ad esempio se S2 ha il diritto di switch S1 allora un processo che esegue nel dominio di S2 può passare nel dominio di S1.

2.3.1 Realizzazione matrice degli accessi

La matrice degli accessi è una notazione astratta che rappresenta il sistema di protezione, ma poiché questa matrice può raggiungere dimensioni molto grandi, visto che ogni colonna rappresenta un possibile oggetto e quindi un file del mio sistema, la matrice sarà una matrice sparsa cioè che rispetto alle dimensioni effettive ci saranno moltissime celle vuote. Per la realizzazione ottimale della matrice degli accessi si fa riferimento ad ACL e CL.

ACL (Access Control List): memorizzazione per colonne. Per ogni oggetto è associata una lista che contiene tutti i soggetti che possono accedere all'oggetto con i relativi diritti di accesso.

La lista degli accessi per ogni oggetto è rappresentata dall'insieme delle coppie:

$\langle \text{soggetto}, \text{insieme dei diritti} \rangle$

limitatamente ai soggetti con un insieme non vuoto di diritti per l'oggetto. Quando deve essere eseguita un'operazione M su un oggetto Oj da parte di Si, si cerca nella lista degli accessi $\langle Si, Rk \rangle$, con M appartenente a Rk.

La ricerca può essere fatta preventivamente in una lista di default contenente i diritti di accesso applicabili a tutti gli oggetti. Se in entrambi i casi la risposta è negativa, l'accesso è negato.

Solitamente ogni soggetto rappresenta un singolo utente; molti sistemi hanno il concetto di gruppo di utenti. I gruppi hanno un nome e possono essere inclusi nella ACL con la seguente forma:

$UID_i, GID_i : \langle \text{insieme di diritti} \rangle$

dove **UID** è lo user identifier e **GID** il group identifier.

In tal modo uno stesso utente può appartenere a gruppi diversi e quindi con diritti diversi.

L'utente può accedere a certi oggetti indipendentemente dal gruppo cui appartiene:

$UID_i, * : \langle \text{diritti di accesso} \rangle$

(CL) Capability List: memorizzazione per righe. Ad ogni soggetto è associata una lista che contiene gli oggetti accessibili dal soggetto ed i relativi diritti di accesso.

Ogni elemento della lista prende il nome di **capability** e garantisce al soggetto certi diritti su un certo oggetto. La capability si compone di un identificatore che identifica l'oggetto e una sequenza di bit che esprime i vari diritti.

Quando S intende eseguire un'operazione M su un oggetto Oj: il meccanismo di protezione controlla se nella lista delle capability associata a S ne esiste una relativa ad Oj che abbia tra i suoi diritti M.

Le CL devono assolutamente essere protette da manomissioni, e questo vengono gestite solo dal SO. L'utente fa riferimento ad un puntatore (capability) che identifica la sua posizione nella lista appartenente allo spazio del kernel.

La CL ha un approccio più efficiente rispetto alle ACL, che risulta più vantaggiosa ogni volta che dobbiamo compiere delle azioni che magari interessano tutti i soggetti per un particolare oggetto, come ad esempio la rimozione di un file.

2.3.2 Revoca diritti di accesso

In un sistema di protezione dinamica può essere necessario revocare i diritti di accesso per un certo oggetto. La revoca può essere:

- Generale o selettiva, cioè vale per tutti gli utenti che hanno quel diritto di accesso o solo per un gruppo;
- Parziale o totale, cioè si riferisce a un sottoinsieme di diritti per l'oggetto o tutti;
- Temporanea o permanente, cioè il diritto di accesso può essere riottenuto successivamente o non sarà più disponibile.

Revoca per un oggetto e ACL con le ACL la revoca risulta semplice. Si fa riferimento alla ACL associata all'oggetto e si cancellano i diritti di accesso che si vogliono revocare.

Revoca per un oggetto e CL l'operazione risulta più complessa poiché è necessario infatti verificare per ogni dominio se contiene la capability con riferimento all'oggetto desiderato.

2.3.3 ACL vs CL

Con le ACL l'informazione di quali diritti di accesso possiede un soggetto S è sparsa nelle varie ACL relative agli oggetti del sistema. Con le CL invece l'informazione relativa a tutti i diritti di accesso applicabili ad un certo oggetto O è sparsa nelle varie CL.

Un sistema di protezione che si basa solo su ACL o CL può presentare alcuni problemi di efficienza, per questo motivo vengono spesso utilizzate delle **soluzioni miste**, cioè utilizzare una combinazione dei due metodi. Le ACL sono memorizzate in forma persistente ad esempio sul disco.

Quindi se un soggetto tenta di accedere ad un oggetto per la prima volta: prima si analizza la ACL; se esiste una entry contenente il nome del soggetto e se tra i diritti di accesso è presente quello richiesto dal soggetto allora viene fornita la capability per l'oggetto. Ciò consente al soggetto di accedere all'oggetto più volte senza che sia necessario analizzare più volte la ACL, e solo in seguito all'ultimo accesso la capability è distrutta.

2.4 Sicurezza multilivello

La maggior parte dei sistemi operativi permette a singoli utenti di determinare chi possa leggere e scrivere i loro file ed i loro oggetti (DAC).

In certi ambiti in cui è necessario un controllo obbligatorio degli accessi al sistema, l'ente definisce delle politiche MAC che stabiliscano regole generali su chi può accedere e a che cosa.

I modelli di sicurezza più utilizzati sono due: il modello **Bell-La Padula** e il **modello Biba**. Entrambi sono modelli multilivello in cui soggetti (utenti) e oggetti (risorse) sono classificati in livelli (classi di accesso):

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

I clearance levels come i sensitivity levels sono chiamati access classes. Una classe di accesso (access class) consiste di due componenti:

1. **security level**: è un elemento basato sull'ordinazione, è costituito da una classificazione gerarchica (Non classificato, Confidenziale, Segreto, Top secret);
2. **category set**: è un insieme di categorie dipendenti dall'applicazione in cui i dati sono usati; non è gerarchico, indica il settore di appartenenza. (ES: posta elettronica, prodotti commerciali, centrali atomiche ecc.).

Ogni soggetto può inoltre accedere ai vari oggetti secondo modi di accesso (access modes) detti:

Read: per la sola lettura;

Append: per l'aggiunta (senza lettura);

Execute: l'esecuzione (per i programmi);

Write: la lettura-scrittura.

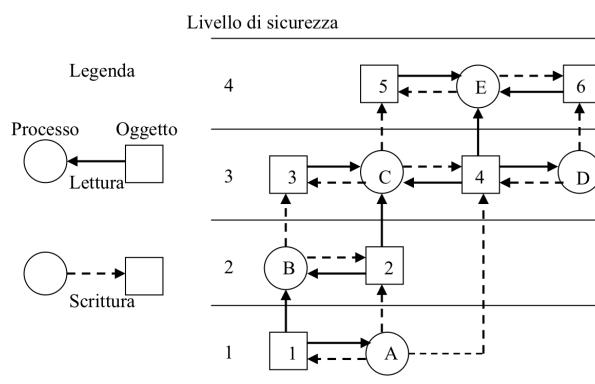
Il modello fissa inoltre le regole di sicurezza, che controllano il flusso delle informazioni tra i livelli.

2.4.1 Modello Bell-La Padula

Progettato per realizzare la sicurezza in ambiente militare, garantendo la confidenzialità delle informazioni, è stato concepito per **mantenere i segreti** e non per garantire l'integrità dei dati.

Questo modello associa a un modello di protezione 2 regole di sicurezza MAC che stabiliscono la direzione di propagazione delle informazioni nel sistema:

- **Proprietà di semplice sicurezza:** un processo in esecuzione al livello di sicurezza k può leggere solo oggetti al suo livello o a quelli inferiori.
- **Proprietà di integrità *:** un processo in esecuzione al livello di sicurezza k può scrivere solo oggetti al suo livello o a quelli superiori.



Esempio: Difesa dal cavallo di Troia

Nell'esempio è utilizzato un cavallo di troia per aggirare un meccanismo di controllo basato su ACL.

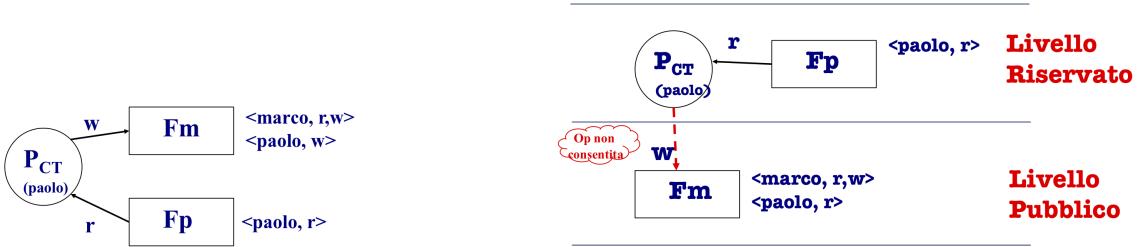
Un utente, Paolo, ha creato il file Fp, contenente la stringa di caratteri riservati "CPE1704TKS", con i permessi lettura/scrittura solo per i processi che appartengono a lui.

Un utente ostile, Marco, ottenuto l'accesso al sistema installa un file eseguibile CT (il cavallo di troia) e copia nel filesystem un file privato Fm che verrà utilizzato come "tasca posteriore".

- Marco ha i permessi di lettura e scrittura per il suo file Fm.
- Marco dà a Paolo il permesso di scrittura su Fm.
- Marco concede a Paolo il diritto di esecuzione su CT.
- Il file Fp (contenente i dati da proteggere) è leggibile solo da Paolo.

Marco induce a Paolo ad eseguire il cavallo di Troia CT (per esempio, spacciandolo come un programma di utilità).

Il programma CT, eseguito da Paolo tramite il processo P_{CT} copia la stringa dei caratteri riservati nel file "tasca posteriore" Fm di Marco: sia l'operazione di lettura che quella di scrittura soddisfano i vincoli imposti da ACL.



Un possibile metodo di difesa nel caso del modello Bell-La Padula è attraverso **due livelli sicurezza**, riservato e pubblico:

- Ai processi e al file Fp di Paolo viene assegnato il livello di sicurezza "riservato";
- A quelli di Marco (Fm e CT) il livello "pubblico".

Quando Paolo esegue CT, il processo P_{CT} creato acquisisce il livello di sicurezza di Paoloo (riservato) e può vedere la stringa di caratteri riservata.

Quando P_{CT} tenta di scrivere la stringa del file Fm pubblico, la proprietà * verrebbe violata ed il tentativo viene negato dal sistema nonostante l'ACL lo consenta (la politica di sicurezza ha precedenza sul meccanismo delle ACL)

2.4.2 Modello Biba

Questo **modello** a differenza di quello Bell-La Padula è stato **concepito per garantire l'integrità dei dati**, e come l'altro modello si basa su 2 regole di sicurezza:

- **Proprietà di semplice sicurezza**: un processo in esecuzione al livello di sicurezza k può scrivere solo oggetti al suo livello o a quelli inferiori.
- **Proprietà di integrità ***: un processo in esecuzione al livello k può leggere solo oggetti al suo livello o a quelli superiori.

Chiaramente i due modelli sono in conflitto tra loro e non possono essere usati contemporaneamente, quindi non possono essere combinati. Il modello Bell-La Padula permette la lettura verso il basso e la scrittura verso l'alto, mentre il modello Biba permette la lettura verso l'alto e la scrittura verso il basso.

2.5 Architettura dei sistemi ad elevata sicurezza

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

Reference monitor: è un **elemento di controllo** realizzato dall'HW e dal S.O. che regola l'accesso dei soggetti agli oggetti sulla base di parametri di sicurezza del soggetto e dell'oggetto.

Il RM ha accesso a una base di calcolo fidata (**Trusted Computing Base, TCB**) che contiene i privilegi di sicurezza di ogni soggetto e gli attributi di protezione di ogni oggetto. Inoltre impone le regole di sicurezza ed ha le seguenti proprietà:

- **Mediazione completa**: le regole di sicurezza vengono applicate ad ogni accesso e non solo, ad esempio quando viene aperto un file (per motivi di efficienza, è preferibile una soluzione parzialmente HW);
- **Isolamento**: il monitor dei riferimenti e la base di dati sono protetti rispetto a modifiche non autorizzate (impossibilità da parte dell'attaccante di modificare la logica);
- **Verificabilità**: la correttezza del monitor dei riferimenti deve essere provata cioè deve essere possibile dimostrare formalmente che il monitor impone le regole di sicurezza e fornisce mediazione completa ed isolamento.

I soggetti nell'accesso agli oggetti devono passare sempre per il RM il quale per capire se di può fare o meno tale accesso deve richiederlo prima al TCB (Trusted Computing Base) cioè la base di dati centrale della sicurezza.

Audit file: vengono mantenuti in questo file eventi importanti per la sicurezza come i tentativi di violazione alla sicurezza e le modifiche autorizzate alla base di dati del nucleo di sicurezza.

2.6 Classificazione della sicurezza dei sistemi di calcolo

Orange Book: Documento pubblicato dal Dipartimento della Difesa americano (D.O.D). Sono specificate quattro categorie di sicurezza:A,B,C,D (in ordine decrescente).

- **Categoria D: Minimal Protection**

Non prevede sicurezza. Esempio MS-DOS, Windows 3.1.

- **Categoria C: Discretionary Protection**

Suddivisa in C1 e C2.

C1. La TCB consente:

- Autenticazione degli utenti (password). I dati di autenticazione sono protetti rendendoli inaccessibili agli utenti non autorizzati.
- Protezione dei dati e programmi propri di ogni utente.
- Controllo degli accessi a oggetti comuni per gruppi di utenti definiti (Unix).

C2. La TCB consente, oltre a quanto definito per la C1, il controllo degli accessi su una base individuale. Esempio Windows NT e 2000.

- **Categoria B: Mandatory Protection**

B1. La TCB consente, oltre a quanto definito in C2, l'introduzione dei livelli di sicurezza (modello Bell-La Padula). Almeno due livelli.

B2. La TCB estende l'uso di etichette di riservatezza ad ogni risorsa del sistema, compresi i canali di comunicazione.

B3. La TCB consente la creazione di liste di controllo degli accessi in cui sono identificati utenti o gruppi cui non è consentito l'accesso ad un oggetto specificato. Es. XTS-400 (STOP kernel), 2003

- **Categoria A: Verified Protection**

Suddivisa in A1 e classi superiori A1. E' equivalente a B3, ma con il vincolo di essere progettato e realizzato utilizzando metodi formali di definizione e verifica (es. SCOMP, Honeywell 1983).

Un sistema appartiene ad una classe superiore ad A1 se è stato progettato e realizzato in un impianto di produzione affidabile da persona affidabile.

Capitolo 3

Programmazione Concorrente

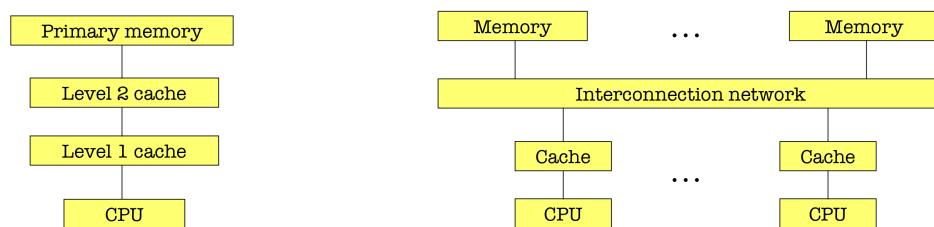
Con il termine programmazione concorrente si intende l'insieme delle tecniche, metodologie e strumenti per il supporto all'esecuzione di sistemi software composti da insiemi di attività svolte simultaneamente.

Nasce negli anni 1960, inizialmente è stata introdotta come una tecnica da utilizzare nel progetto di un sistema operativo, con l'obiettivo di sfruttare la disponibilità dei canali I/O: si trattava di veri e propri processori special purpose, dotati di un linguaggio macchina e in grado di fornire il supporto all'esecuzione di programmi speciali, dedicati a controllare il trasferimento dei dati ed eseguiti concorrentemente al programma in esecuzione sulla CPU.

Successivamente, l'introduzione del meccanismo di interruzione e dei canali ad accesso diretto alla memoria (DMA) ha fornito il supporto HW all'introduzione della tecnica della multiprogrammazione. Con tale tecnica, la programmazione concorrente è stata progressivamente sviluppata in stretto rapporto con lo sviluppo dei S.O. Successivamente, con l'introduzione di architetture multiprocessore, la conoscenza ha usufruito anche del supporto HW, che ha consentito l'esecuzione realmente in parallelo (overlapping) di più processi sequenziali.

3.1 Sistemi Multiprocessore

In un'architettura a singolo processore si ha una sola CPU con due livelli di cache e la memoria, mentre in un'architettura a memoria condivisa e multiprocessore abbiamo le memorie condivise tra le varie CPU presenti tramite una rete di interconnessione.



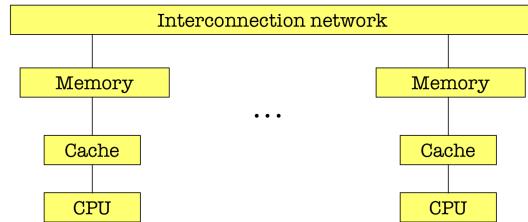
Possiamo distinguere due modelli di sistemi multiprocessore:

UMA (Uniform Memory Access) : sistemi a multiprocessore con un numero ridotto di processori (da 2 a 30 circa). Presentano una rete di interconnessione realizzata da un memory bus o da crossbar switch; il tempo di accesso è uniforme da ogni processore ad ogni locazione di memoria. (chiamati anche Symmetric MultiProcessors).

NUMA (Non Uniform Access Time) : sistemi con un numero elevato di processori (decine o centinaia). Presentano una memoria organizzata gerarchicamente (per evitare la congestione del bus). La rete di interconnessione è un insieme di switches e memorie strutturato ad albero. Ogni processore

ha memorie che sono più vicine ed altre più lontane; il tempo di accesso dipende dalla distanza tra processore e memoria.

3.1.1 Distributed-Memory



Nelle architetture **Distributed-Memory** invece ogni processore accede alla propria memoria che quindi non è condivisa, e il tempo di accesso è veloce ma è limitato alla sola memoria locale alla CPU. Possono essere classificate in *Multicomputer*, in cui i processori e la rete sono fisicamente vicini, quindi si utilizza un collegamento diretto, oppure *Network System*, in cui i nodi sono collegati da una rete locale (ethernet) o da un rete internet.

3.2 Classificazione architetture

La più usata classificazione dei sistemi di calcolo paralleli è la tassonomia di **Flynn** (1972). Si basa su due concetti:

- **Parallelismo a livello di istruzioni:**

- **Single instruction stream:** l'architettura è in grado di eseguire un singolo flusso di istruzioni.
- **Multiple instruction streams:** possono essere eseguiti più flussi di istruzioni in parallelo.

- **Parallelismo a livello di dati:**

- **Single data stream:** l'architettura è in grado di elaborare un singolo flusso sequenziale di dati.
- **Multiple data streams:** l'architettura è in grado di processare più flussi di dati paralleli.

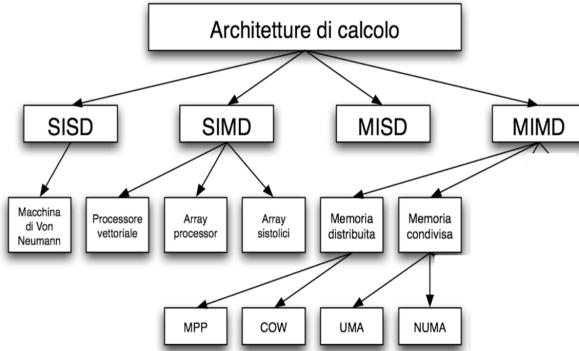
		Instruction Streams	
		one	many
Data Streams	one	SISD traditional von Neumann single CPU computer	MISD May be pipelined Computers
	many	SIMD Vector processors fine grained data Parallel computers	MIMD Multi computers Multiprocessors

- **SISD:** single instruction, single data. Elaboratore monoprocesso (modello Von Neumann)
- **SIMD:** architetture composte da molte unità di elaborazione che eseguono contemporaneamente la stessa istruzione su insiemi di dati diversi.

Esempio: **Array processors**

Architetture composte da più unità di elaborazione gestite da un'unica unità di controllo, in modo che eseguano in modo sincrono la stessa istruzione su data stream separati (sfruttati per il calcolo ad alte prestazioni).

- **MISD**: più istruzioni in parallelo sullo stesso flusso di dati.
- **MIMD**: istruzioni diverse vengono eseguite in parallelo su dati diversi (in questa categoria rientrano sia i multiprocessori, sia i multiciders).



3.3 Tipi di applicazioni

Le applicazioni possono essere classificate in 3 tipologie:

- **Multithreaded**: applicazioni strutturate come un insieme di processi per far fronte alla complessità, aumentare l'efficienza e semplificare la programmazione. Sono caratterizzate dal fatto che esistono più processi che non processori per eseguire i processi, quindi quest'ultimi sono schedulati ed eseguiti indipendentemente.
- **Sistemi distribuiti / sistemi multitasking**: le componenti dell'applicazione (task) vengono eseguite su nodi collegati tramite opportuni mezzi di interconnessione; i processi comunicano scambiandosi dei messaggi (tipica organizzazione Client/Server).
- **Applicazioni parallele**: si vuole risolvere un dato problema più velocemente oppure un problema di dimensioni maggiori sempre nello stesso tempo, e per fare ciò sono eseguite su processori paralleli (ad esempio processori vettoriali) facendo uso di algoritmi appositi.

3.4 Processi non sequenziali e tipi di iterazione

Algoritmo Procedimento logico che deve essere eseguito per risolvere un determinato problema.

Programma Descrizione di un algoritmo mediante un opportuno formalismo (linguaggio di programmazione), che rende possibile l'esecuzione dell'algoritmo da parte di un particolare elaboratore.

Processo Insieme ordinato degli eventi cui dà luogo un elaboratore quando opera sotto il controllo di un programma.

Elaboratore Entità astratta realizzata in hardware e parzialmente in software, in grado di eseguire programmi (descritti in un dato linguaggio).

Evento Esecuzione di un'operazione tra quelle appartenenti all'insieme che l'elaboratore sa riconoscere ed eseguire; ogni evento determina una transizione di stato dell'elaboratore.

→ Un programma descrive non un processo, ma un insieme di processi, ognuno dei quali è relativo all'esecuzione del programma da parte dell'elaboratore per un determinato insieme di dati in ingresso.

3.4.1 Processo sequenziale

Sequenza di stati attraverso i quali passa l'elaboratore durante l'esecuzione di un programma. Se gli stati del processo sono rigidamente ordinati in sequenza allora abbiamo un processo sequenziale (per una coppia di stati sono sempre in grado di dire quale dei due è venuto prima e quale dopo, poiché vi una relazione d'ordine totale tra gli stati).

Grafo di precedenza Un processo può essere rappresentato tramite un grafo di precedenza del processo costituito da nodi ed archi orientati: i nodi del grafo rappresentano i singoli eventi, mentre gli archi orientati rappresentano le precedenze temporali tra tali eventi.

Ogni nodo rappresenta un evento dell'esecuzione di un'operazione tra quelle appartenenti all'insieme che l'elaboratore sa riconoscere e eseguire.

3.5 Processo non sequenziale

Se i processi non sono sequenziali allora gli eventi non avvengono in sequenza e quindi l'insieme degli eventi che lo descrive è ordinato secondo una relazione d'ordine parziale.

L'esecuzione di un processo non sequenziale richiede:

- Un **elaboratore non sequenziale**, cioè un elaboratore in grado di eseguire più operazioni contemporaneamente (mono o multi elaboratori);
- Un **linguaggio di programmazione non sequenziale** (o concorrente) cioè un linguaggio che consenta di descrivere un insieme di attività concorrenti tramite moduli che possono essere eseguiti in parallelo.

3.5.1 Scomposizione di un processo non sequenziale

Se il linguaggio concorrente permette di esprimere il parallelismo a livello di sequenza di istruzioni allora si può scomporre un processo non sequenziale in un insieme di processi sequenziali eseguiti contemporaneamente e far fronte alla complessità di un algoritmo non sequenziale.

Le attività rappresentate dai processi possono essere:

- **indipendenti** quindi l'evoluzione di un processo non condiziona gli stati degli altri processi.
- **interagenti** nel senso che sono assoggettati a vincoli di precedenza tra stati che appartengono a processi diversi (come ad esempio vincoli di precedenza tra le operazioni o vincoli di sincronizzazione).

3.5.2 Interazione tra processi

Le possibili iterazioni tra processi possono essere:

1. **Cooperazione:** comprende tutte le iterazioni prevedibili e desiderate insite nella logica dei programmi. Prevede scambio di informazioni con trasmissione di dati (messaggi) o senza trasferimento di dati (segnali temporali).

E' presente una relazione di causa e effetto tra l'esecuzione dell'operazione di invio da parte del processo mittente e l'esecuzione dell'operazione di ricezione da parte del processo ricevente; ovviamente deve esistere un vincolo di precedenza tra questi due eventi.

2. **Competizione:** rappresenta un'iterazione prevedibile non desiderata, ma necessaria; ha come obiettivo il coordinamento dei processi nell'accesso alle risorse condivise.

La macchina su cui sono eseguiti i processi mette a disposizione un numero limitato di risorse condivise tra i processi e la competizione. Per risorse che non possono essere utilizzate contemporaneamente da più processi bisogna prevedere dei meccanismi di competizione. (Un esempio di competizione è la mutua esclusione).

Sezione critica è la sequenza di istruzioni con le quali un processo accede a un oggetto condiviso con altri processi. La regola di mutua esclusione stabilisce che sezioni critiche appartenenti alla stessa classe devono escludersi mutuamente nel tempo.

3. **Interferenza:** interazione provocata da errori di programmazione, cioè rappresenta un'interazione non prevista e non desiderata: dipende dalla velocità relativa dei processi e gli errori possono manifestarsi nel corso del programma a seconda delle diverse condizioni di velocità di esecuzione dei processi. Uno degli obiettivi della programmazione concorrente è l'eliminazione delle interferenze.

3.6 Architetture e linguaggi per la programmazione concorrente

Avendo a disposizione una macchina concorrente, cioè in grado di eseguire più processi sequenziali insieme, e di un linguaggio di programmazione con il quale descrivere algoritmi non sequenziali, è possibile scrivere e dar eseguire programmi concorrenti. L'elaborazione complessiva può essere descritta come un insieme di processi sequenziali asincroni interagenti.

Le proprietà di un linguaggio di programmazione concorrente:

- Contenere appositi costrutti con i quali sia possibile dichiarare moduli di programma destinati ad essere eseguiti come processi sequenziali distinti.
- Non tutti i processi costituenti un'elaborazione vengono eseguiti contemporaneamente, infatti alcuni processi vengono svolti se dinamicamente si verificano particolari condizioni. Bisogna quindi poter specificare quando un processo deve essere attivo e terminato.
- Devono essere presenti strumenti linguistici per specificare le iterazioni che dinamicamente potranno verificarsi tra i vari processi.

3.7 Architettura di una macchina concorrente

La macchina concorrente M spesso è una macchina astratta creata equipaggiando un macchina fisica con una macchina SW che crea un livello di astrazione e consente di risolvere i problemi di gestione. Al proprio interno M contiene ciò che deve essere messo in atto quando richiediamo l'esecuzione di processi concorrenti e tutto quello che riguarda l'iterazione (sincronizzazione con scambio di informazioni).

Nel nucleo sono sempre presenti due funzionalità base:

- **Meccanismo di multiprogrammazione:** è quello preposto alla gestione delle unità di elaborazione della macchina M' (unità reali) consentendo ai vari processi eseguiti sulla macchina astratta M di condividere l'uso delle unità reali di elaborazione (scheduling) tramite l'allocazione in modo esclusivo ad ogni processo di un'unità virtuale di elaborazione
- **Meccanismo di sincronizzazione e comunicazione:** è quello che estende le potenzialità delle unità reali di elaborazione, rendendo disponibile alle unità virtuali strumenti mediante i quali sincronizzarsi e comunicare.

Oltre a questi due meccanismi è presente anche un **meccanismo di protezione**, realizzato o in HW o in SW, che permette di rilevare eventuali interferenze tra i processi.

3.7.1 Architettura della macchina M

In base all'organizzazione logica di M vengono definiti due modelli di interazione tra i processi:

1. **Modello a memoria comune**, in cui l'interazione tra i processi avviene su oggetti contenuti nella memoria comune (modello ad ambiente globale).

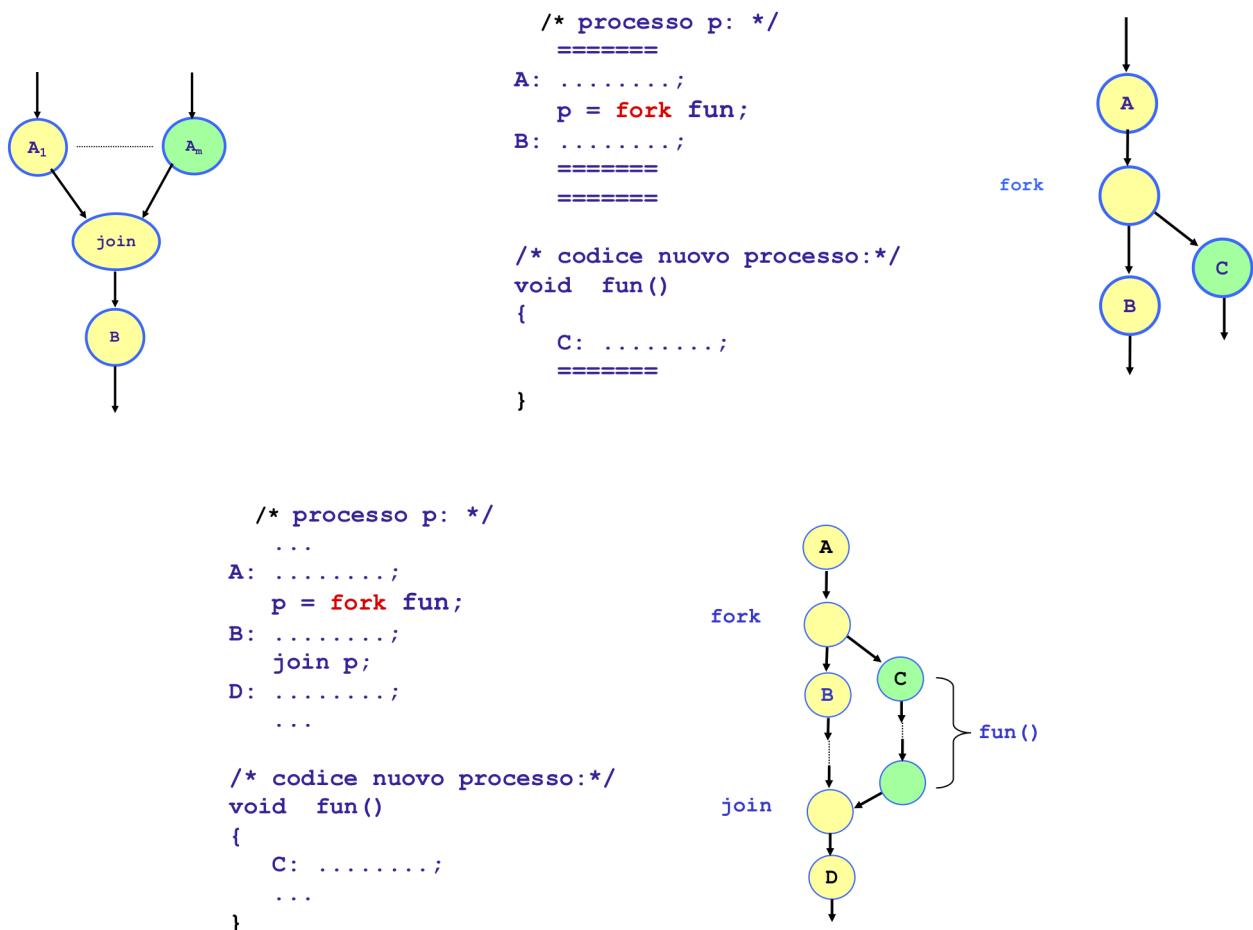
2. **Modello a scambio di messaggi**, in cui la comunicazione e la sincronizzazione tra processi si basa sullo scambio di messaggi sulla rete che collega i vari elaboratori (modello ad ambiente locale).

3.8 Costrutti linguistici per la specifica della concorrenza

Fork L'esecuzione di una fork coincide con la creazione e l'attivazione di un processo che inizia la propria esecuzione in parallelo con quella del processo chiamante.

Ha un comportamento simile a quello di una chiamata di procedura `call`; mentre quest'ultima implica l'attivazione del programma chiamato e la sospensione del programma chiamante, la fork prevede che il programma chiamante prosegua contemporaneamente con l'esecuzione della funzione chiamata. Coincide quindi con una biforcazione del flusso di controllo del programma.

Join Consente di determinare quando un processo, creato tramite la fork, ha terminato il suo compito, sincronizzandosi con tale evento; concettualmente coincide con la congiunzione di più flussi di controllo indipendenti.



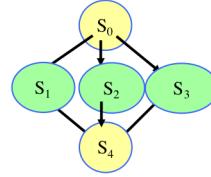
E' facilmente intuibile che un programma concorrente, scritto con `fork` e `join`, non risulti di facile lettura, poiché non è agevole ricavare dal testo quali siano i processi attivi in ogni punto del programma durante la sua esecuzione.

Cobegin/Coend La concorrenza viene espresso tramite in costrutto cobegin - coend al cui interno vengono eseguite delle istruzioni S1, S2, ... Sn che sono eseguite in parallelo.

```

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

```



Fork-Join vs Cobegin-Coend *Fork-Join* è un formalismo più generale di *Cobegin-Coend*: tutti i grafi di precedenza possono essere espressi tramite *fork-join*, ciò non è vero per i costrutti *cobegin-end*.

Nel caso di costrutti *fork-join*, un processo può creare altri processi mediante la **fork**. Il processo padre, quello che ha eseguito la fork, e il processo figlio, quello creato, continuano la loro esecuzione in parallelo. Il processo padre si sospende soltanto se esegue la **join** quando il processo figlio non ha ancora terminato la sua esecuzione (tramite apposita primitiva, esempio *quit*).

Nel caso del costrutto *Cobegin-Coend* il processo che esegue una *cobegin* crea tanti processi quante sono le istruzioni componenti il blocco parallelo, quindi sospende la sua esecuzione in attesa che tutti i processi figli abbiano terminato.

Processo Costrutto linguistico per individuare, in modo sintatticamente preciso, quali moduli di un programma possono essere eseguiti come processi autonomi:

```

|| process <identificatore> ( <parametri formali> )
||   {           <dichiarazione di variabili locali>;
||     <corpo del processo>
||   }

```

3.9 Realizzazione delle primitive di creazione e terminazione dei processi

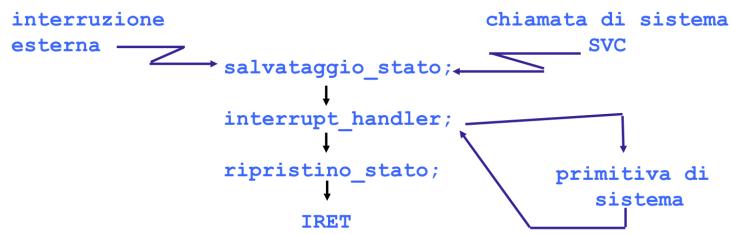
Il compito della realizzazione delle unità centrali virtuali, sulle quali i processi vengono eseguiti, e dei meccanismi di comunicazione e sincronizzazione, necessari per le interazioni tra processi, è affidato a un particolare componente di sistema, chiamato nucleo o kernel.

Il nucleo, oltre a rappresentare il supporto a tempo di esecuzione di un linguaggio di programmazione che utilizza il concetto di processo, rappresenta in generale la parte più interna di un sistema operativo.

In questo paragrafo, ci concentreremo sulla realizzazione, ambiente sia monoprocesso che multiprocessore, delle primitive **fork**, usate da un processo per creare un altro processo (figlio), di quelle **join**, usate per consentire a un processo di attendere la terminazione di un processo figlio (equivalente alla primitiva **wait** di UNIX), e di quelle **quit**, usate per eliminare un processo (equivalente alla primitiva **exit** di UNIX).

Tutte le funzioni del nucleo vengono implementate in modo da comportarsi come operazioni primitive e quindi atomiche (indivisibile da punto di vista logico). La tecnica utilizzata per garantire l'atomicità delle funzioni del nucleo, consiste nell'eseguirle a interruzioni disabilitate e, nel caso dei sistemi multielaboratori, facendo riferimento anche ai meccanismi di *lock* e *unlock*.

Ciò implica che il meccanismo di **passaggio tra l'ambiente dei processi e l'ambiente del nucleo** è costituito dal meccanismo di interruzione:



Un processo invoca una primitiva di sistema eseguendo un'opportuna istruzione del tipo chiamata a supervisore (SVC), che genera un'interruzione producendo un effetto del tutto analogo all'arrivo di un segnale esterno da una periferica (si distinguono infatti interruzioni interne o sincrone da quelle esterne o asincrone).

L'esecuzione di una primitiva da parte di un processo corrisponde all'inserimento, nello stack stesso, dei parametri richiesti dalla primitiva stessa e all'esecuzione di una istruzione di tipo SVC che genera l'interruzione di un tipo sincrono. La gestione dell'interruzione prevede il salvataggio del contesto del processo, cioè dei valori contenuti nei registri di macchina (**salvataggio_stato**). Viene quindi eseguita la procedura di risposta all'interruzione (**interrupt_handler**) e, successivamente si torna al processo ripristinando nei registri i valori prima salvati (**rispristino_stato**) ed eseguendo l'istruzione di ritorno da interruzione (IRET).

Ciascun processo è rappresentato nel nucleo da una struttura dati detta **des_processo** (descrittore del processo):

```

typedef struct {
    PID nome;
    modalità_di_servizio servizio;
    tipo_contesto contesto;
    tipo_stato stato; //running, ready, waiting, ecc.
    PID padre;
    int N_figli;
    des_figlio prole[max_figli];
    p_des successivo;
} des_processo;
  
```

Ogni processo viene identificato univocamente con un identificatore intero:

```

|| typedef int PID;
  
```

A questo scopo, il kernel definisce la funzione **assegna_nome** che, chiamata all'atto della creazione di un processo, restituisce il suo PID:

```

|| PID assegna_nome();
  
```

Modalità di servizio (es. scheduling time sharing con priorità):

```

typedef struct {
    int priorità;
    int delta_t; // time slice
} modalità_di_servizio;
  
```

Descrittore del figlio (**des_figlio**):

```

typedef struct{
    PID figlio;
    boolean terminato; //true = terminato
} des_figlio;
  
```

Insieme di tutti i descrittori (descrittori): **des_processo**

```

|| descrittori[num_max_proc];
  
```

Funzione **descrittore** per selezionare il descrittore del processo di PID=x :

```

|| p_des descrittore(PID x);
  
```

CODE DI DESCRITTORI I descrittori dei processi sono organizzati in Code:.

Rappresentazione della coda di descrittori (`des_coda`):

```
|| typedef struct {
||     p_des primo, ultimo;
|| } des_coda;
```

Inserimento e prelievo in/da coda:

```
|| void Inserimento (p_des proc, des_coda coda);
|| p_des Prelievo (des_coda coda);
```

Code dei processi pronti (es: scheduling multilevel, con priorità):

```
|| des_coda coda_processi_pronti[num_min_priorità];
```

Coda dei descrittori liberi:

```
|| des_coda descrittori_liberi;
```

Meccanismi realizzati dal kernel: cambio di contesto

```
|| p_des processo_in_esecuzione;

|| void salvataggio_stato() {
||     p_des esec = processo_in_esecuzione;
||     esec -> contesto = <valori dei registri della CPU>;
|| }

|| void ripristino_stato( ) {
||     p_des esec = processo_in_esecuzione;
||     <registri della CPU> = esec -> contesto;
|| }
```

Scheduling dei processi

```
|| void assegnazione_CPU {
||     int k=0;
||     p_des p;
||     while ((coda_processi_pronti[k].primo)==null)
||         k++;
||     p = prelievo(coda_processi_pronti [k]);
||     p -> stato = < running>;
||     processo_in_esecuzione = p;
||     <registro-temporizzatore>= p -> servizio.delta_t;
|| }
```

Invocata ogni volta che il processo in esecuzione esce dallo stato di “running”.

Attivazione di un processo L’attivazione di un processo P, dopo una fase di sospensione, porta P nello stato di pronto (code dei processi pronti). Se lo scheduling prevede preemption, il processo attivato P può scalzare il processo in esecuzione nell’uso della CPU.

```
|| void attiva (p_des proc) {
||     p_des esec = processo_in_esecuzione;
||     int pri_esec = esec -> servizio.priorità;
||     int pri_proc = proc -> servizio.priorità;
||     if (pri_esec > pri_proc) { /* pre-emption */
||         proc -> stato = < running>;
||         inserimento(esec,coda_processi_pronti[pri_esec]);
||         processo_in_esecuzione = proc;
||     } else {
||         proc -> stato = < ready>;
||         inserimento(proc,coda_processi_pronti[pri_proc]);
||     }
|| }
```

fork

```
typedef enum {OK,eccezione} risultato;

risultato fork (des_processo inizializzazione) {
    p_des p;
    int NF;
    p_des esec = processo_in_esecuzione;
    if (descrittori_liberi.primo == NULL )
        return eccezione; /* non ci sono descritt. liberi*/
    else {
        p = prelievo(descrittori_liberi);
        *p = inizializzazione;
        p -> nome = assegna_nome();
        p -> padre = esec -> nome;
        NF = esec->N_figli;
        esec -> prole[NF].figlio = p -> nome;
        esec -> prole[NF].terminato = false; esec -> N_figli++;
        attiva(p);
        return ok;
    }
}
```

join

```
void join (PID nome_figlio) {
    p_des esec = processo_in_esecuzione;
    int k = indice_figlio(esec, nome_figlio);
    if (esec -> prole[k].terminato == false) { /* figlio non terminato*/
        esec -> stato = < 'sospeso in attesa che il figlio termimi' >;
        assegnazione_CPU();
    }
}
```

quit

```
void quit() {
    p_des esec = processo_in_esecuzione;
    PID nome_padre = esec -> padre;
    p_des p_padre = descrittore(nome_padre);
    int k=indice_figlio(p_padre, esec -> nome);
    p_padre -> prole[k].terminato=true;
    inserimento(esec, descrittori_liberi);
    if (p_padre -> stato==< 'in attesa che questo figlio termini' >) {
        int pri = p_padre -> servizio.priorità;
        inserimento(p_padre, coda_processi_pronti[pri]);
        p_padre -> stato = < ready>;
    }
    assegnazione_CPU( );
}
```

3.10 Programmi

Una delle attività più importanti per chi sviluppa programmi è la verifica di correttezza dei programmi realizzati.

Quando viene eseguito un programma si tiene **traccia dell'esecuzione** (Storia) ovvero la sequenza degli stati attraversati dal sistema di elaborazione durante l'esecuzione del programma.

Gli **stati** posso essere definiti come l'insieme dei valori delle variabili definite nel programma più le variabili “implicite” (es. valore del Program counter).

Programmi sequenziali ogni esecuzione di un programma P su un particolare insieme di dati D genera sempre la stessa traccia.

La verifica può agevolmente essere svolta tramite debugging.

Programmi concorrenti l'esito dell'esecuzione dipende da quale sia l'effettiva sequenza cronologica di esecuzione delle istruzioni contenute: ogni esecuzione di un programma P su un particolare insieme di dati D dà origine ad una traccia diversa.

La verifica di proprietà di programmi concorrenti è molto più difficile: il semplice debug su una (o su un numero limitato di esecuzioni) non dà alcuna garanzia sul soddisfacimento di una data proprietà.

3.10.1 Proprietà dei programmi

Una proprietà di un programma P è un attributo che è sempre vero in ogni possibile traccia generata dall'esecuzione di P, in generale vengono classificate in due categorie:

- **SAFETY**: proprietà che garantisce che durante l'esecuzione di P non si entrerà mai in uno stato errato (stato in cui le variabili assumono valori non desiderati).
- **LIVENESS**: proprietà che garantisce che durante l'esecuzione di P prima o poi si entrerà in uno stato corretto (stato in cui le variabili assumono valori desiderati).

Un programma sequenziale deve avere fondamentalmente due proprietà:

1. **Correttezza del risultato finale**: quindi per ogni esecuzione il risultato ottenuto è giusto (SAFETY).
2. **Terminazione**: quindi prima o poi l'esecuzione termina (LIVENESS).

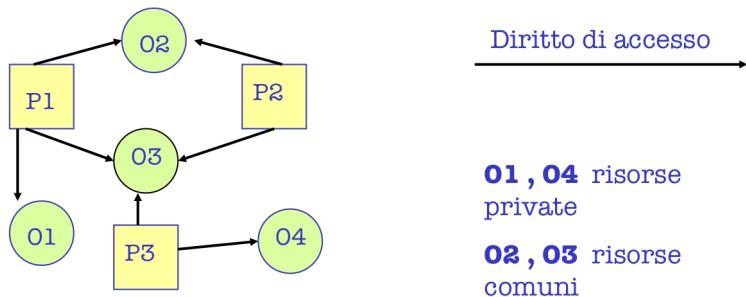
Un programma concorrente invece deve avere fondamentalmente le stesse proprietà di un programma sequenziale con in aggiunta queste altre proprietà:

3. **Mutua esclusione nell'accesso a risorse condivise**: quindi per ogni esecuzione non accadrà mai che più di un processo acceda contemporaneamente alla stessa risorsa (SAFETY).
4. **Assenza di deadlock**: quindi per ogni esecuzione non si verificheranno mai situazioni di blocco critico (SAFETY).
5. **Assenza di starvation** (blocco critico in attesa di acquisire una risorsa): cioè prima o poi ogni processo potrà accedere alle risorse richieste (LIVENESS).

Capitolo 4

Modello a memoria comune

Il sistema è visto come un insieme di processi e oggetti, con i processi che hanno diritto di accesso sugli oggetti. Il tipo di iterazione tra i processi è di competizione o cooperazione.



Il modello a memoria comune rappresenta la naturale astrazione del funzionamento di un sistema in multiprogrammazione costituito da uno o più processori che hanno accesso ad una memoria comune. Ad ogni processore può essere associata una memoria privata, ma ogni iterazione avviene tramite oggetti contenuti nella memoria comune (ambiente globale, accessibile a tutti).

Ogni applicazione viene strutturata come un insieme di componenti suddiviso in due sottoinsiemi disgiunti: i **processi** (componenti attivi) e le **risorse** (componenti passivi) che a loro volta sono suddivise in risorse di tipo primitivo e di tipo astratto.

Risorsa : qualunque oggetto fisico o logico di cui un processo necessita per portare a termine il suo compito. Le risorse sono raggruppate in classi e una **classe** identifica l'insieme di tutte le **operazioni** che un processo può eseguire per operare su risorse di quella classe. Il termine risorsa si identifica con quello di struttura dati allocata nella memoria comune.

4.1 Meccanismo di controllo degli accessi

Il meccanismo di controllo degli accessi ha il compito di controllare che gli accessi dei processi alle risorse avvengano correttamente.

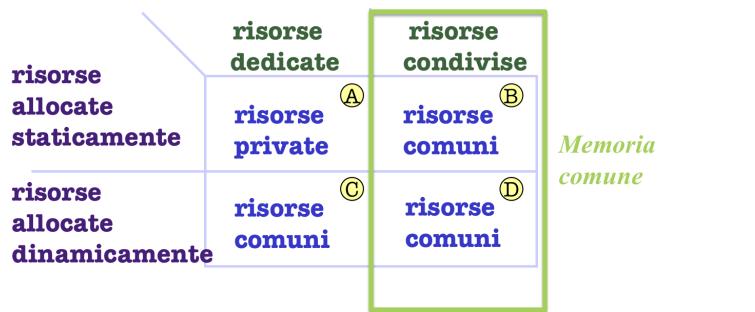
Al fine di distribuire i diritti di accesso alle risorse tra i vari processi di un'applicazione concorrente, per ogni risorsa R viene introdotto il concetto di *gestore* (o *allocatore*) della risorsa, il cui compito è quello di indicare, istante per istante, quali processi hanno il diritto ad operare su R e quali no.

Gestore di una risorsa : Per ogni risorsa R, il suo gestore definisce, in ogni istante t, l'insieme $S_R(t)$ dei processi che, in tale istante, hanno il diritto di operare su R.

Una risorsa può essere classificata come:

- **Dedicata**: se $S_R(t)$ ha una cardinalità sempre ≤ 1 ;

- **Condivisa**: se la risorsa non è dedicata;
- **Allocata staticamente**: se $S_R(t)$ è una costante. $S_R(t)$ viene definito prima dell'esecuzione (il gestore della risorsa è il programmatore che, in base alle regole del linguaggio, stabilisce quale processo può vedere e quindi operare su R);
- **Allocata dinamicamente**: se $S_R(t)$ è funzione del tempo (il gestore GR definisce l'insieme $S_R(t)$ in fase di esecuzione e quindi deve essere un componente della stessa applicazione, nel quale l'allocazione viene decisa a run-time in base a politiche date).



Il caso A è l'unico in cui la risorsa è **privata**, mentre in tutti gli altri casi B, C e D le risorse sono **comuni**. Nei casi B e D però la competizione si verifica nel momento in cui due o più processi tentano di accedere alla stessa risorsa. Nel caso C invece è il gestore a decidere chi può accedere a una particolare risorsa. Ma lo stesso gestore è una risorsa quindi c'è competizione nell'accedere al gestore.

4.1.1 Compiti del gestore di una risorsa

1. Mantenere **aggiornato** l'insieme $S_R(t)$ e cioè lo stato di allocazione della risorsa;
2. Fornire i **meccanismi** che un processo può utilizzare per acquisire il diritto di operare sulla **risorsa**, entrando a far parte dell'insieme $S_R(t)$, e per rilasciare tale diritto quando non è più necessario;
3. Implementare la **strategia** di allocazione della risorsa e cioè definire quando, a chi e per quanto tempo allocare la risorsa.

Data una risorsa R, il suo gestore G_R è costituito da una **risorsa** in un sistema organizzato secondo il modello a memoria comune; un **processo** in un sistema organizzato secondo il modello a scambio di messaggi.

4.1.2 Accesso a risorse

Consideriamo un processo P che deve operare, ad un certo istante, su R di tipo T (op_1, op_2, \dots, op_n):

- Se R è allocata **staticamente** a P (modalità A e B), il processo, se appartiene a S_R , possiede il diritto di operare su R in qualunque istante:

```
|| R.opi(...); /* esecuzione dell'operazione opi su R */
```

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

```
|| GR.Richiesta(...); /* acquisizione della risorsa R */
  R.opi(...); /* esecuzione dell'operazione opi su R */
  GR.Rilascio(...); /* rilascio della risorsa R */
```

- Se R è allocata come **risorsa condivisa**, (modalità B e D) è necessario assicurare che gli accessi avvengano in modo non divisibile:

Le funzioni di accesso alla risorsa devono essere programmate come una classe di sezioni critiche (utilizzando i meccanismi di sincronizzazione offerti dal linguaggio di programmazione e supportati dalla macchina concorrente).

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

4.2 Regione critica

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

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

Il corpo della regione rappresenta un'operazione da eseguire sulla risorsa condivisa R e quindi costituisce una sezione critica che deve essere eseguita in mutua esclusione con le altre operazioni definite su R.

E' costituito da due istruzioni da eseguire in sequenza: l'istruzione Sa e, successivamente, l'istruzione Sb. Una volta terminata l'esecuzione di Sa viene valutata la condizione C: se è vera l'esecuzione continua con Sb, altrimenti il processo che ha invocato l'operazione attende che la condizione C diventi vera.

Casi particolari per le regioni critiche:

- **region R « S; »** : specifica della sola mutua esclusione senza che sia prevista alcuna forma di sincronizzazione diretta.
- **region R « when(C) »** : specifica di un semplice vincolo di sincronizzazione, quindi il processo attende che C sia verificata prima di eseguire.
- **region R « when(C) S; »** : specifica il caso in cui la condizione C di sincronizzazione caratterizza lo stato in cui la risorsa R deve trovarsi al fine di poter eseguire l'operazione S.

4.2.1 Esempio: Scambio di informazioni tra processi

Supponendo che la risorsa M sia una struttura con i seguenti campi:

```
||     T buffer;
||     boolean pieno;
```

si ha:

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

4.3 Mutua Esclusione

Il problema della mutua esclusione nasce quando più di un processo alla volta può aver accesso a variabili comuni.

La regola di mutua esclusione impone che le operazioni con le quali i processi accedono alle variabili comuni **non si sovrappongano nel tempo**. Nessun vincolo è imposto sull'ordine con il quale le operazioni sulle variabili vengono eseguite.

Sezione Critica La sequenza di istruzioni con le quali un processo accede e modifica un insieme di variabili comuni prende il nome di sezione critica.

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

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" al più 1 sezione critica di ogni classe.

Soluzioni possibili

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

4.3.1 Strumenti di sincronizzazione: il Semaforo

Strumento linguistico di basso livello utilizzato a livello macchina concorrente per implementare strumenti di sincronizzazione di più alto livello. Disponibile all'interno di librerie standard, che consentono di implementare programmi concorrenti anche utilizzando linguaggi sequenziali, per esempio il C o il C++, e chiamando le funzioni della libreria come ad esempio *Pthread* (standard POSIX).

Un semaforo è una variabile intera non negativa, cui è possibile accedere solo tramite le due operazioni P e V. Il numero contenuto nel semaforo rappresenta il numero di risorse di un certo tipo disponibili ai task (processi).

Un semaforo può essere modificato da parte del codice utente **solamente con tre chiamate di sistema**:

- **Inizializzazione**: Il semaforo viene inizializzato con un valore intero e positivo.
- **Operazione P o wait**: Il semaforo viene decrementato. Se, dopo il decremento, il semaforo ha un valore negativo, il task viene sospeso e accodato, in attesa di essere riattivato da un altro task.
- **Operazione V o signal**: Il semaforo viene incrementato. Se ci sono task in coda, uno dei task in coda (il primo nel caso di accodamento FIFO) viene tolto dalla coda e posto in stato di ready (sarà perciò eseguito appena schedulato dal sistema operativo).

Il valore del semaforo, una volta inizializzato, non può più essere letto, se non indirettamente tramite le primitive P e V (i nomi sono stati proposti da Dijkstra come le iniziali delle parole olandesi "probieren" ("verificare") e "verhogen" ("incrementare"))

In fase di dichiarazione di un semaforo **s** ne specificheremo il valore iniziale **i** ($i \geq 0$) assegnando tale valore a **s**:

|| **semaphore s=i;**

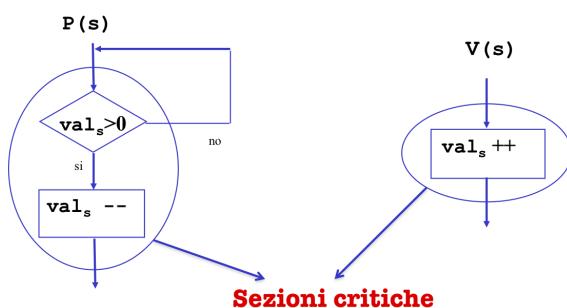
Un oggetto di tipo **semaphore**, in quanto strumento di sincronizzazione tra processi, è sempre condiviso tra da due o più processi che lo utilizzano per sincronizzare le loro velocità di esecuzione. Un processo esegue l'operazione **P** (o wait) su un semaforo per testare se è vera una condizione, concettualmente associata al semaforo e necessaria per la sua prosecuzione, sospendendo la propria esecuzione in caso negativo e rimanendo in attesa che la condizione diventi vera. L'operazione **V** (o signal) viene utilizzata per riattivare l'esecuzione di un processo precedentemente sospeso su una P.

La semanticà delle due operazioni può essere così specificata:

```

|| void P(semaphore s):
    region s << when(val_s>0) val_s--;>>
|| void V(semaphore s):
    region s << val_s++;>>
  
```

dove val_s rappresenta il valore del semaforo. Essendo l'oggetto s condiviso, le due operazioni P e V vengono definite come sezioni critiche da eseguire in mutua esclusione. Le due operazioni devono essere eseguite in forma atomica.



I nomi P e V sono stati proposti da Dijkstra come le iniziali delle parole olandesi "probieren" ("verificare") e "verhogen" ("incrementare").

Il semaforo viene utilizzato come strumento di sincronizzazione tra processi concorrenti:

- **sospensione:** $P(s)$, $\text{val}_s == 0$
- **risveglio:** $V(s)$, se vi è almeno un processo sospeso

Proprietà semafori Dato un semaforo S, siano:

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

Relazione di invarianta → ad ogni istante possiamo esprimere il valore del semaforo come:

$$\text{val}_s = I_s + nv_s - np_s$$

da cui ($\text{val}_s \geq 0$): $np_s \leq I_s + nv_s$

La relazione di invarianta è sempre soddisfatta, per ogni semaforo, qualunque sia il suo valore e comunque sia il programma concorrente che lo usa (Safety property). Possiamo sfruttare questa relazione per dimostrare formalmente le proprietà dei programmi concorrenti che usano i semafori.

4.4 Esempi di utilizzo dei semafori

4.4.1 Semaforo di mutua esclusione

Se il semaforo è inizializzato al valore uno e se ogni processo che lo utilizza esegue prima l'operazione P e successivamente la V, allora il semaforo è sicuramente binario (assume solo valori 0 e 1). Non potrà mai assumere valori maggiori di uno poiché ogni operazione V che ne incrementa il suo valore è sempre preceduta da una P che, terminando, lo decrementa.

Un tipico esempio è quello relativo alla soluzione del caso più semplice di competizione tra processi, che riguarda la necessità di evitare accessi contemporanei a una risorsa condivisa se questa è costituita da un oggetto di un tipo astratto.

```
class tipo_risorsa {
    <struttura dati di ogni istanza della classe>;
    semaphore mutex = 1;
    public void op1() {
        P(mutex);           /*prologo*/
        <sez. critica: corpo della funzione op1>;
        V(mutex);           /*epilogo*/
    } ...
    public void opN() {
        P(mutex);           /*prologo*/
        <sez. critica: corpo della funzione opN>;
        V(mutex);           /*epilogo*/
    }
} ...
tipo_risorsa ris;
ris.op1();
```

Le condizioni necessarie sono soddisfatte:

- a) Sezioni critiche della stessa classe devono essere eseguite in modo **mutuamente esclusivo**.

Dim: $N_{sez} = np - nv$

Dalla relazione invariante: $1 + nv - np \geq 0 \rightarrow np - nv \leq 1 \rightarrow N_{sez} \leq 1$

Inoltre, poiché il protocollo impone che P(mutex) preceda V(mutex) in ogni processo, in ogni istante dell'esecuzione vale sempre la relazione: $np \geq nv \rightarrow np - nv \geq 0 \rightarrow N_{sez} \geq 0$. **c.v.d.**

- b) Non deve essere possibile il verificarsi di situazioni in cui i processi impediscono mutuamente la prosecuzione della loro esecuzione (**deadlock**).

Dim: Per assurdo, se ci fosse deadlock:

- tutti i processi sarebbero in attesa su P(mutex) $\rightarrow Val_{mutex} = 0$
- nessun processo sarebbe nella sezione critica $\rightarrow N_{sez} = np - nv = 0$

Ma sappiamo che: $Val_{mutex} = I_{mutex} - np + nv \rightarrow Val_{mutex} = 1 - N_{sez}$

$0 = 1 - 0$ assurdo! **c.v.d.**

- c) Quando un processo si trova all'esterno di una sezione critica non può rendere impossibile l'accesso alla stessa sezione (o a sezioni della stessa classe) ad altri processi.

Dim: Se nessun processo è nella sezione critica, deve essere possibile ad un qualunque processo di entrare nella sezione critica senza ritardi. Se la sezione critica è libera: $N_{sez} = np - nv = 0$

$Val_{mutex} = I_{mutex} - np + nv \rightarrow Val_{mutex} = 1$

P(mutex) non è bloccante **c.v.d.**

Mutua esclusione tra gruppi di processi

Può capitare che la struttura interna della risorsa e la tipologia delle operazioni previste dal suo tipo siano tali per cui l'esecuzione di una operazione op_i invalidi la precondizione di ogni altra operazione op_j ma non quella della stessa operazione op_i . In questi casi è possibile consentire a processi diversi di eseguire contemporaneamente la stessa operazione sulla risorsa, ma non operazioni diverse.

Il problema può essere risolto facendo ricorso ai semafori di mutua esclusione, seguendo lo stesso criterio:

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

ma differenziando il prologo e l'epilogo di ogni operazione op_i da quelle delle altre operazioni.

- **prologo_i** deve sospendere il processo che ha chiamato l'operazione op_i se sulla risorsa sono in esecuzione operazioni diverse da op_i ; viceversa deve consentire al processo di eseguire op_i .
- **epilogo_i** deve liberare la mutua esclusione solo se il processo che lo esegue è l'unico processo in esecuzione sulla risorsa (è l'ultimo di un gruppo di processi che hanno eseguito la stessa op_i).

Per questo motivo faremo ancora riferimento a un semaforo mutex di mutua esclusione per realizzare, però, l'esclusione mutua non di un processo con tutti gli altri ma bensì di un gruppo di processi.

Prologo ed epilogo di op_i sono sezioni critiche, per questo motivo introduco un secondo semaforo di mutua esclusione m_i :

```
semaphore mutex=1, mi=1;
public void op_i( ) {
    P(mi);
    conti++;
    if (conti==1) {P(mutex);}
    V(mi);
    <corpo della funzione op_i>;
    P(mi);
    conti--;
    if (conti==0) {V(mutex);}
    V(mi);
}
public void op_j( ) { /*j diverso da i*/
    P(mutex);
    <corpo della funzione op_j>;
    V(mutex);
}
```

Problema dei lettori/scrittori

Sia data una risorsa condivisa F (ad esempio un file) che può essere acceduta dai thread concorrenti in due modi: **lettura** per ispezionare il contenuto di F senza modificarlo; **scrittura** per modificare il contenuto di F. Una possibile politica di sincronizzazione degli accessi a F potrebbe essere:

- La lettura è consentita a più processi contemporaneamente;
- La scrittura è consentita a un processo alla volta;
- Lettura e scrittura su F non possono avvenire contemporaneamente.

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

```

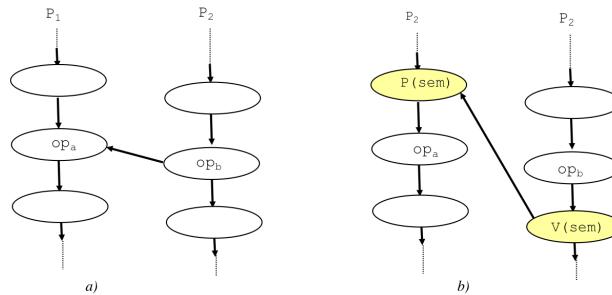
public void lettura(...) {
    P(ml);
    contl++;
    if (contl==1) P(mutex);
    V(ml);
    <lettura del file >;
    P(ml);
    contl--;
    if (contl==0) V(mutex);
    V(ml);
}
public void scrittura(...) {
    P(mutex);
    <scrittura del file >;
    V(mutex);
}

```

4.4.2 Semaforo evento

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

Esempio: op_a deve essere eseguita da P_1 solo dopo che P_2 ha eseguito op_b



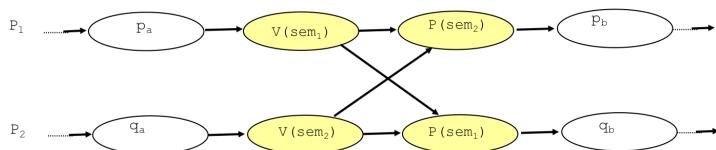
Introduciamo un semaforo sem inizializzato a zero: prima di eseguire op_a , P_1 esegue $P(sem)$; dopo aver eseguito op_b , P_2 esegue $V(sem)$.

```

Semaphore sem = 0;
P1:
...
P(sem);
operazioneB;
...
P2:
...
V(sem);
operazioneA;
...

```

Un diverso uso di semafori evento riguarda la soluzione di un classico problema di sincronizzazione tra processi, noto anche come il problema del **vincolo di rendez-vous** cioè dell'appuntamento tra processi. Per esempio supponiamo che due processi P_1 e P_2 eseguono ciascuno due operazioni, p_a e p_b il primo e q_a e q_b il secondo. Vogliamo imporre che l'esecuzione di p_b da parte di P_1 e q_b da parte di P_2 possono iniziare solo dopo che entrambi i processi hanno completato la loro prima operazione (p_a e q_a), come in figura:



```

Semaphore sem1 = 0;
P1:
Pa
V(sem1);
Semaphore sem2 = 0;
P2:
Qa
V(sem2);

```

```

|| P(sem2);
|| Pb
|| P(sem1);
|| Qb

```

Generalizzando il problema a N processi, l'esecuzione di ogni operazione P_{ib} è subordinata al completamento di tutte le istruzioni P_{ia} ($i = 1, \dots, N$). La realizzazione si avrebbe mediante un **barriera di sincronizzazione**.

Variabili condivise:

```

|| semaphore mutex=1;
|| semaphore barriera=0;
|| int completati=0; // numero dei thread che hanno eseguito la prima operazione P_ia

```

Struttura del thread i-esimo P_i :

```

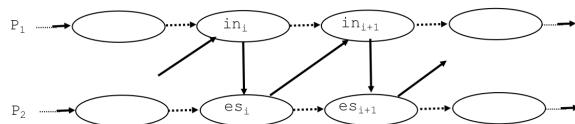
|| <operazione a di Pi>
|| P(mutex);
|| completati++;
|| if (completati==N) {V(barriera);}
|| V(mutex);
|| P(barriera);
|| V(barriera);
|| <operazione b di Pi>

```

4.4.3 Semafori binari composti

Un ulteriore esempio di semafori binari riguarda la soluzione al classico problema dello scambio di dati fra processi. Supponiamo che due processi P_1 e P_2 si debbano periodicamente scambiare un dato di un certo tipo generico T . Per risolvere il problema in un sistema a memoria comune è sufficiente riservare una variabile condivisa di tipo T , chiamiamola *buffer* dove P_1 scrive il dato quando lo vuole inviare a P_2 e da cui P_2 lo preleva quando lo deve ricevere. I vincoli da rispettare sono:

- a) Gli accessi al *buffer* devono essere mutuamente esclusivi se T è un tipo astratto;
- b) P_2 può prelevare un dato solo dopo che P_1 lo ha inserito;
- c) P_1 prima di inserire un dato, deve attendere che P_2 abbia prelevato il precedente.



Utilizziamo due semafori: **vu**, per realizzare l'attesa di P_1 , in caso di buffer pieno; **pn**, per realizzare l'attesa di P_2 , in caso di buffer vuoto:

```

|| buffer inizialmente vuoto
|| valore iniziale vu = 1
|| valore iniziale pn = 0

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

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

```

L'uso dei due soli semafori **pn** e **vu** garantiscono che l'esecuzione di estrai ed inserisci avviene in mutua esclusione anche senza far ricorso al semaforo mutex. La coppia di semafori si comporta nel suo insieme come se fosse un unico semaforo binario di mutua esclusione (vu verde e pn rosso → inserisci viceversa: pn verde e vu rosso → estrai).

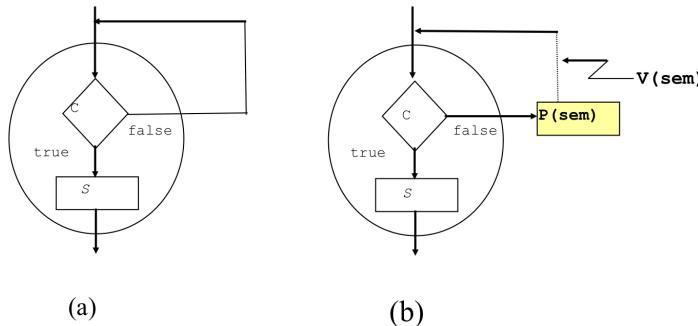
4.4.4 Semafori condizione

Capita spesso che una o più delle operazioni per accedere a una risorsa condivisa R richieda che lo stato interno della risorsa soddisfi una particolare condizione logica C:

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

che indica appunto che op1 è una regione critica, dovendo operare su una risorsa condivisa R; S1 ha come precondizione la validità della condizione C.

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



Si possono effettuare due tipi di schemi per l'accesso alla risorsa R:

- prevede l'**attesa circolare** quindi viene utilizzato un ciclo *while* per verificare la condizione e il ciclo continuerà fino a che la condizione non è verificata;
- l'altra possibilità è quella di utilizzare uno **schema con passaggio di testimone** in cui viene utilizzato un *if* per verificare la condizione e nel caso di condizione non verificata deve abbandonare la struttura ciclica.

Supponiamo di aver riservato il semaforo **sem** inizializzato a zero associato alla condizione C in modo tale che il processo, che ha invocato op1() e ha trovato la condizione C falsa, esca dalla sezione critica e si blocchi su **sem**. Ovviamente dobbiamo supporre che sia disponibile almeno un'ulteriore operazione op2() che, invocata da un altro processo, modifichi lo stato interno di R in modo tale che C diventi vera. In questo caso, è nell'ambito dell'operazione op2() che viene eseguita la primitiva **V(sem)** per indicare che la condizione C è adesso vera.

E' però necessario che chi esegue la primitiva **P(sem)** trovi semaforo rosso per avere la garanzia di bloccarsi. Per questo motivo, oltre alla condizione C associamo, oltre al semaforo **sem**, anche un contatore **csem** inizializzato a zero, che viene incrementato prima di eseguire la **P(sem)** e decrementato dopo che il processo viene risvegliato, in modo tale che il suo valore indichi sempre il numero di processi bloccati su **sem**. In questo modo, nell'operazione op2(), la **V(sem)** viene eseguita soltanto se il contatore **csem** è maggiore di zero, cioè se ci sono processi sospesi sul semaforo, evitando che assuma valori minori di zero.

Schema con attesa circolare

Un processo che invoca op1() entra in mutua esclusione e valuta la condizione C: se è vera, esegue il corpo S1 della funzione e termina rilasciando la mutua esclusione; se è falsa, incrementa il contatore associato al semaforo **sem** e, liberata la mutua esclusione, si blocca su di esso.

Un processo che esegue op2(), completato il corpo S2 della funzione e avendo reso vera la condizione C, può svegliare chi attende questo evento. Per questo esso verifica se su **sem** c'è qualche processo bloccato e, in questo caso esegue, **V(sem)** per svegliarne uno.

```

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

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

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

```

Schema con passaggio del testimone

```

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

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

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

```

Il secondo schema è più efficiente del primo, ma consente di risvegliare un solo processo alla volta poiché ad uno solo può passare il diritto di operare in mutua esclusione. I due schemi presuppongono che la condizione C, che viene testata all'interno della funzione op₁() (precondizione di S1), sia verificata anche all'interno di op₂() (deve essere una post condizione di S2). Ciò significa che non deve contenere variabili locali o parametri della funzione op₁() altrimenti non sarebbe verificabile all'interno di op₂().

4.4.5 Esempio di uso di semafori condizione: Gestione di un pool di risorse equivalenti

Si consideri un insieme (pool) di N risorse tutte uguali: ciascun processo può operare su una qualsiasi risorsa del pool purché libera.

Supponiamo che venga scelta la risorsa di indice 3:

```
|    ris[3].opi();
```

Se la risorsa fosse utilizzata da un altro dei processi applicativi, il processo si sospende per mutua esclusione pur essendoci, magari, altre risorse disponibili. Si ha quindi la necessità di allocare dinamicamente le risorse del pool tramite un gestore che mantenga aggiornato lo stato delle risorse:

1. Ciascun processo, quando deve operare su una risorsa, chiede al gestore l'allocazione di una di esse.
2. Il gestore, se ci sono risorse disponibili, ne sceglie una e la alloca in modo dedicato al processo richiedente restituendo l'indice relativo.
3. Il processo può quindi operare sulla risorsa senza preoccuparsi della mutua esclusione.
4. Al termine il processo rilascia la risorsa al gestore.

```

int richiesta(): region G <<
    when (<ci sono risorse disponibili>)
    <scelta di una risorsa disponibile>;
    int i = <indice della risorsa scelta>;
    <registra che la risorsa di indice i non è più disponibile>;
    return i;
>>

void rilascio(int r): region G <<
    <registra che la risorsa r-esima è di nuovo disponibile>
>>

class tipo_gestore {
    semaphore mutex = 1;      /*sem. di mutua esclusione*/
    semaphore sem = 0;        /*semaforo condizione*/
    int csem = 0;             /*contatore dei proc. sospesi su sem */
    boolean libera[N];       /*indicatori di risorsa libera*/
    int disponibili = N;     /*contatore risorse libere*/

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

    public int richiesta() {
        int i=0;
        P(mutex);
        if (disponibili == 0) {
            csem++;
            V(mutex);
            P(sem);
            csem--;
        }
        while(!libero[i]) i++;
        libero[i]=false;
        disponibili--;
        V(mutex);
        return i;
    }

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

tipo_gestore G; /* def. gestore */

/*struttura del generico processo che vuole accedere a una risorsa del pool: */
process P {
    int    ris;
    ...
    ris = G.richiesta( );
    < uso della risorsa di indice ris>
    G.rilascio (ris) ;
    ...
}

```

4.4.6 Semafori risorsa

Spesso uno stesso problema si presta ad essere risolto usando paradigmi diversi. E' questo, per esempio, il caso visto prima del gestore di un pool di risorse equivalenti. Infatti, ogni volta che il problema da risolvere prevede un'allocazione di risorse, è possibile far riferimento a uno schema diverso che utilizza **semafori generali**, che possono cioè assumere qualunque valore maggiore o uguale a zero. In questi casi, possiamo riservare un unico semaforo **n_ris** inizializzato con un valore uguale al numero di risorse da allocare ed eseguire una **P(n_ris)** in fase di allocazione e una **V(n_ris)** in fase di rilascio. In questo modo, ad ogni allocazione, il valore di **n_ris** viene decrementato e, quando non ci sono più risorse disponibili, il suo valore diventa nullo bloccando una ulteriore richiesta fino a quando non viene eseguito un rilascio. E' per questo motivo che viene indicato come *semaforo risorsa*.

```

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

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

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

```

Problema dei produttori/consumatori

```

coda_di_n_T      buffer;
semaphore         pn = 0;
semaphore         vu = n;
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;
}

```

4.4.7 Semafori privati: specifica di strategia di allocazione

In problemi di sincronizzazione complessi, in particolare quando si voglia realizzare una determinata politica di gestione delle risorse, la decisione di consentire a un processo di proseguire l'esecuzione durante l'accesso a una risorsa condivisa dipende, come si è visto dal verificarsi di una specifica condizione, detta *condizione di sincronizzazione*. Può quindi accadere che più processi siano simultaneamente bloccati durante l'accesso a una risorsa condivisa, ciascuno in attesa che la propria condizione di sincronizzazione sia verificata.

Nasce quindi il problema di quale riattivare nel caso in cui le condizioni di alcuni processi bloccati si verifichino contemporaneamente. Nei casi precedenti la condizione di sincronizzazione era particolarmente semplificata (vedi mutua esclusione) e la scelta di quale processo riattivare veniva effettuata tramite l'algoritmo implementato nella V. Normalmente questo algoritmo, dovendo essere sufficientemente generale ed il più possibile efficiente, coincide con quello FIFO.

Nel seguito verranno riportati alcuni esempi di definizione di politiche di gestione delle risorse per i quali risulta necessario poter programmare esplicitamente *algoritmi di scelta dei processi da riattivare*.

Esempio 1: Su un buffer da N celle di memoria più produttori possono depositare messaggi di dimensione diversa.

Politica di gestione: tra più produttori ha priorità di accesso quello che fornisce il messaggio di dimensione maggiore. Finché un produttore il cui messaggio ha dimensioni maggiori dello spazio disponibile nel buffer rimane sospeso, nessun altro produttore può depositare un messaggio anche se la sua dimensione potrebbe essere contenuta nello spazio libero del buffer.

Condizione di sincronizzazione: il deposito può avvenire se c'è sufficiente spazio per memorizzare il messaggio e non ci sono produttori in attesa. Il prelievo di un messaggio da parte di un consumatore prevede la riattivazione tra i produttori sospesi, di quello il cui messaggio ha la dimensione maggiore, sempre che esista sufficiente spazio nel buffer.

Se lo spazio disponibile non è sufficiente nessun produttore viene riattivato.

Esempio 2: (pool di risorse equivalenti + priorità) Un insieme di processi utilizza un insieme di risorse comuni R₁, R₂, ..., R_n. Ogni processo può utilizzare una qualunque delle risorse.

Condizione di sincronizzazione: l'accesso è consentito se esiste una risorsa libera. A ciascun processo è assegnata una priorità; in fase di riattivazione dei processi sospesi viene scelto quello cui corrisponde la massima priorità.

Un semaforo s si dice **privato** per un processo quando solo tale processo può eseguire la primitiva P sul semaforo s. La primitiva V sul semaforo può essere invece eseguita anche da altri processi. Un semaforo privato viene inizializzato con il valore zero.

Vediamo adesso due diversi schemi di utilizzazione di semafori privati per risolvere problemi di allocazione di risorse permettendo di specificare strategie di allocazione.

Primo schema Sia P_i un processo la cui acquisizione di una risorsa dipende dal verificarsi di una condizione di sincronizzazione. Indicando con i il PID del processo e con $\text{priv}[i]$ un suo semaforo privato, di seguito viene illustrato lo schema di massima del tipo di un gestore che, mediante le operazioni **acquisizione** e **rilascio**, consente di allocare la risorsa secondo una specifica strategia:

```

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

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

    public void rilascio( ) {
        int i;
        P(mutex);
        <rilascio della risorsa>;
        if (<esiste almeno un processo sospeso per il quale la condizione
            di sincronizz. è soddisfatta>)
        {
            <scelta del Pi (fra quelli sospesi) da riattivare>;
            <allocazione della risorsa a Pi>;
            <registrare che Pi non è più sospeso>;
            V(priv[i]);
        }
        V (mutex); }
    }
}

```

Proprietà della soluzione:

- La sospensione del processo, nel caso in cui la condizione di sincronizzazione non sia soddisfatta, non può avvenire entro la sezione critica in quanto ciò impedirebbe ad un processo che rilascia la risorsa di accedere a sua volta alla sezione critica e di riattivare il processo sospeso.
La sospensione avviene al di fuori della sezione critica.
- La specifica del particolare algoritmo di assegnazione della risorsa non è opportuno che sia realizzata nella primitiva V. Nella soluzione proposta è possibile programmare esplicitamente tale algoritmo scegliendo in base ad esso il processo da attivare ed eseguendo V sul suo semaforo privato.

Lo schema presentato può, in certi casi, presentare degli inconvenienti: l'operazione P sul semaforo privato viene sempre eseguita anche quando il processo richiedente non deve essere bloccato.

Inoltre il codice relativo all'assegnazione della risorsa viene duplicato nelle procedure acquisizione e rilascio. Ciò non è sempre possibile, in particolare se l'allocazione prevede che debba essere restituito un valore al processo richiedente tramite un parametro.

Secondo schema Un secondo schema può essere ottenuto ricorrendo al concetto di semaforo condizione, solo che questo schema prevede di associare, a ogni condizione, un semaforo privato per ogni processo invece di un solo semaforo. Lo schema non soffre sei precedenti inconvenienti:

```

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

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

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

```

Il risveglio segue lo schema del passaggio di testimone!

A differenza della soluzione precedente, in questo caso risulta più complesso realizzare la riattivazione di più processi per i quali risulti vera contemporaneamente la condizione di sincronizzazione. Infatti adottando il criterio del passaggio del testimone, non è possibile risvegliare più di un processo alla volta: a uno solo può essere trasferito il diritto di operare in mutua esclusione.

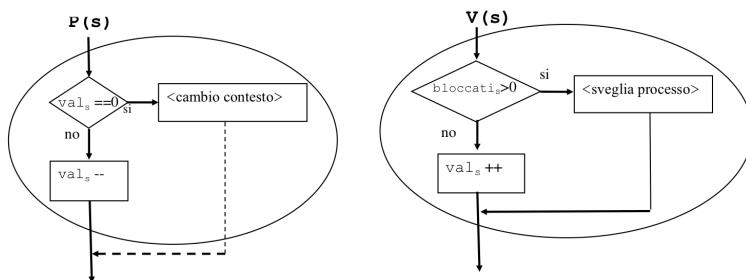
4.5 Realizzazione dei semafori

In sistemi operativi multiprogrammati, il semaforo viene realizzato dal kernel, che elimina ogni forma di attesa attiva dei processi (conseguenza della definizione della primitiva P).

Per motivi di efficienza infatti ogni processo che non può proseguire l'esecuzione, anziché ripetere continuamente l'esame del valore del semaforo, verrà sospeso e il controllo del processore fisico su cui è eseguito assegnato a un altro processo. Il processo sospeso verrà successivamente risvegliato come conseguenza dell'esecuzione da parte di un altro processo di una operazione V sullo stesso semaforo.

Partendo dalla definizione astratta della primitiva P possiamo ottenere un nuovo schema dove il ciclo di attesa attiva viene sostituito con un cambio di contesto, realizzato utilizzando le funzioni del nucleo, e dove il simbolo grafico ovale, che racchiude le operazioni, rappresenta il atto che tali operazioni devono essere eseguite in forma atomica. Le linee tratteggiate rappresentano il flusso di controllo nel momento in cui il processo che si sospende verrà riattivato.

Avendo eliminato il ciclo di attesa attiva dall'interno della P, il processo sospeso non è in grado, da solo, di verificare quando il semaforo torna verde. Dobbiamo modificare anche lo schema della primitiva V, in modo che sia questa a svolgere il compito di svegliare uno dei processi in attesa quando viene eseguita.



Le due primitive sono strutturate ciascuna come una sola istruzione **if-then-else**. Ciò significa che non tutte le operazioni V implicano un incremento del valore semaforico, così come non tutte le P terminano decrementando. Ciò potrebbe sembrare una violazione della relazione invariante, ma in realtà a ogni V che termina eseguendo il ramo **then** corrisponde la terminazione di una P.

La realizzazione dei semafori comporta alcune aggiunte alle strutture del nucleo per fare fronte alle operazioni di creazione e terminazione dei processi. Il descrittore di un semaforo viene così definito:

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

Una P su un semaforo con contatore a 0, sospende il processo nella coda queue, altrimenti il contatore viene decrementato.

Una V su un semaforo la cui coda queue non è vuota, estrae un processo dalla coda, altrimenti incrementa il contatore.

L'implementazione di P e V è parte del nucleo della macchina concorrente e dipende dal tipo di architettura hardware (monoprocesso, multiprocesso, ecc.) e da come il nucleo rappresenta e gestisce i processi concorrenti.

Capitolo 5

Nucleo di un sistema multiprogrammato

Il modello a processi prevede l'esistenza di tante unità di elaborazione (macchine virtuali) quanti sono i processi; ogni macchina possiede come set di istruzioni elementari quelle corrispondenti all'unità centrale reale più le istruzioni relative alla creazione ed eliminazione dei processi, al meccanismo di comunicazione e sincronizzazione (compresa la comunicazione con i dispositivi di I/O visti come processori esterni). Questo modello consente di mettere in evidenza le proprietà logiche di comunicazione e sincronizzazione tra processi senza doversi occupare degli aspetti implementativi legati alle particolari caratteristiche del processore fisico (es. gestione delle interruzioni).

5.1 Nucleo

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

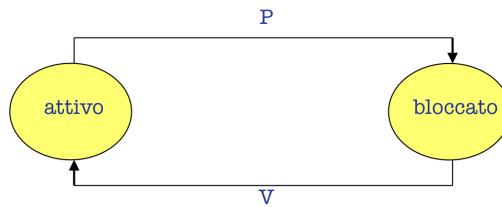
Il nucleo costituisce il *livello più interno* di un qualunque sistema basato su processi (ad esempio il livello più elementare di un sistema operativo multiprogrammato o il supporto a tempo di esecuzione di un linguaggio per la programmazione concorrente).

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

Caratteristiche fondamentali del nucleo sono:

- **Efficienza:** condiziona l'intera struttura a processi. Per questo motivo esistono sistemi in cui alcune o tutte le operazioni del nucleo sono realizzate in hardware o mediante microprogrammi.
- **Dimensioni:** la semplicità delle funzioni richieste al nucleo fa sì che la sua dimensione risulti estremamente limitata.
- **Separazione tra meccanismi e politiche:** il nucleo deve, per quanto possibile, contenere solo meccanismi consentendo così, a livello di processi, di utilizzarli per la realizzazione di diverse politiche di gestione a seconda del tipo di applicazione.

Stati di un processo: Le transizioni tra i due stati sono implementate dai meccanismi di sincronizzazione realizzati dal nucleo.



Quando un processo perde il controllo del processore, il contenuto dei registri del processore viene salvato in un'area di memoria associata al processo, chiamata **descrittore**. Ciò consente una maggiore flessibilità nella politica di assegnazione del processore ai processi, rispetto alla soluzione di salvare le informazioni nello stack : infatti, in un istante generico, uno qualunque dei processi pronti può ottenere l'uso del processore e ricaricare i registri con i valori salvati nel descrittore di processo.

Contesto di un processo: è l'insieme delle informazioni contenute nei registri del processore e relative al processo.

La funzione fondamentale del nucleo di un sistema a processi è la **gestione delle transizioni di stato dei processi**. Più precisamente il nucleo deve:

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

5.2 Realizzazione del Nucleo (Architettura Monoprocessore)

5.2.1 Descrittore del processo

Contiene le seguenti informazioni:

- **Identificazione del processo:** Ad ogni processo è associato un nome che lo identifica univocamente durante il suo tempo di vita.
- **Modalità di servizio dei processi:** contiene parametri di scheduling. Ad esempio:
 - FIFO
 - Priorità (fissa o variabile)
 - Deadline. Il descrittore contiene un valore che sommato all'istante di richiesta di servizio da parte del processo determina il tempo massimo entro il quale la richiesta può essere soddisfatta.

– Quanto di tempo. Sistemi time sharing HP: scheduling con priorità pre-emptive

- **Contesto del processo:** contatore di programma, registro di stato, registri generali, indirizzo dell'area di memoria privata del processo.
- **Identificazione del processo successivo:** a seconda del loro stato i processi vengono inseriti, come si vedrà, in apposite code. Ogni descrittore contiene pertanto l'identificatore del processo successivo nella stessa coda.

Esempio di realizzazione:

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

typedef struct {
    int nome;
    modalità_di_servizio servizio;
    tipo_contesto contesto;
    int successivo;
} descrittore_processo;

/* insieme di tutti i descrittori:*/
descrittore_processo descrittori[num_max_proc];
```

5.2.2 Coda dei processi pronti

Esistono una o più code di processi pronti (caso di processi con priorità). Quando un processo è riattivato per effetto di una V, viene inserito al fondo della coda corrispondente alla sua priorità.

La coda dei processi pronti contiene sempre almeno un processo fittizio (dummy process) che va in esecuzione quando tutte le altre code sono vuote. Il processo dummy ha la priorità più bassa ed è sempre nello stato di pronto.

Il processo dummy rimane in esecuzione fino a quando qualche altro processo diventa pronto, eseguendo un ciclo senza fine (oppure una funzione di servizio, o una di bassa priorità).

Esempio di realizzazione:

```
typedef struct {
    int primo, ultimo;
} descrittore_coda;
typedef descrittore_coda coda_a_livelli[Npriorità];
coda_a_livelli coda_processi_pronti;
```

Inserimento di un descrittore in coda:

```
void Inserimento(int P, descrittore_coda C) {...}
/* inserisce il processo di indice P nella coda C */
```

Prelievo di un descrittore da una coda:

```
int Prelievo(descrittore_coda C) {...}
/* estrae il primo processo dalla coda C e restituisce il suo indice */
```

5.2.3 Descrittori liberi

Coda nella quale sono concatenati i descrittori disponibili per la creazione di nuovi processi e nella quale sono ritornati i descrittori dei processi eliminati:

```
descrittore_coda descrittori_liberi;
```

5.2.4 Processo in esecuzione:

Il nucleo necessita di conoscere quale processo è in esecuzione. Questa informazione, rappresentata dall'indice del descrittore del processo, viene contenuta in un particolare registro del processore :

```
|| int processo_in_esecuzione;  
|| /* indice del processo che sta usando la CPU */
```

Quando il nucleo è inizializzato (il che avviene durante l'operazione di bootstrap dell'elaboratore), viene creato un processo e l'indice del processo viene inserito nel registro processo in esecuzione.

5.3 Funzioni del Nucleo

Le funzioni del nucleo realizzano le operazioni di transizione di stato per i singoli processi. Ogni transizione prevede il prelievo, da una coda, del descrittore del processo coinvolto ed il suo inserimento in un'altra coda.

Si utilizzano a questo scopo due procedure: Inserimento e Prelievo di un descrittore da una coda.

Se la coda è vuota si adotta l'ipotesi che il campo primo assuma il valore -1. Analogamente verrà assegnato il valore -1 al campo successivo dell'ultimo descrittore nella coda.

Le funzioni del nucleo possono essere suddivise in due livelli:

- **Livello superiore:** contiene tutte le funzioni direttamente utilizzabili dai processi sia esterni (dispositivi di I/O) sia interni, quali risposta ai segnali di interruzione e primitive per la creazione, eliminazione e sincronizzazione dei processi.
- **Livello inferiore:** realizza le funzionalità di cambio di contesto: salvataggio del contesto del processo che si sospende nel suo descrittore, scelta di un nuovo processo da mettere in esecuzione tra quelli pronti e ripristino del suo contesto.

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

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

Le funzioni del nucleo devono essere eseguite in modo mutuamente esclusivo.

I due ambienti di esecuzione (nucleo e processi utente) corrispondono a stati diversi di operazione dell'elaboratore (kernel e user). Il meccanismo di passaggio da uno all'altro è basato sul meccanismo delle interruzioni. In particolare:

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

5.3.1 Funzioni del livello inferiore: Cambio di Contesto

1. Salvataggio del contesto del processo in esecuzione nel suo descrittore. (**Salvataggio_stato**)
2. Inserimento del descrittore nella coda dei processi bloccati o dei processi pronti.
3. Rimozione del processo a maggior priorità dalla coda dei pronti e caricamento dell'identificatore di tale processo nel registro processo in esecuzione (**Assegnazione_CPU**)
4. Caricamento del contesto del nuovo processo nei registri di macchina. (**Ripristino_stato**)

```

void Salvataggio_stato( ) {
    int j;
    j = processo_in_esecuzione;
    descrittori[j].contesto = <valori dei registri CPU>;
}
void Ripristino_stato( ) {
    int j;
    j = processo_in_esecuzione;
    <registro-temp> = descrittori[j].servizio.delta_t;
    <registri-CPU> = descrittori[j].contesto ;
}
void Assegnazione_CPU( ) {      // scheduling
    int k=0,j;
    while (coda_processi_pronti[k].primo)==-1)
        k++;
    j = Prelievo (coda_processi_pronti[k]);
    processo_in_esecuzione = j;
}

```

5.3.2 Gestione del temporizzatore

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

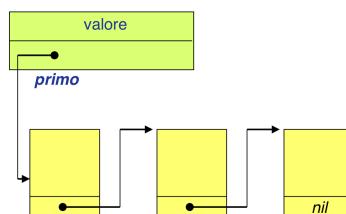
```

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

```

5.3.3 Semafori (Ambiente monoprocessore)

Nel nucleo di un sistema monoprocessore il semaforo può essere implementato tramite una variabile intera che rappresenta il suo valore (≥ 0) e un puntatore ad una lista di descrittori di processi in attesa sul semaforo (bloccati).

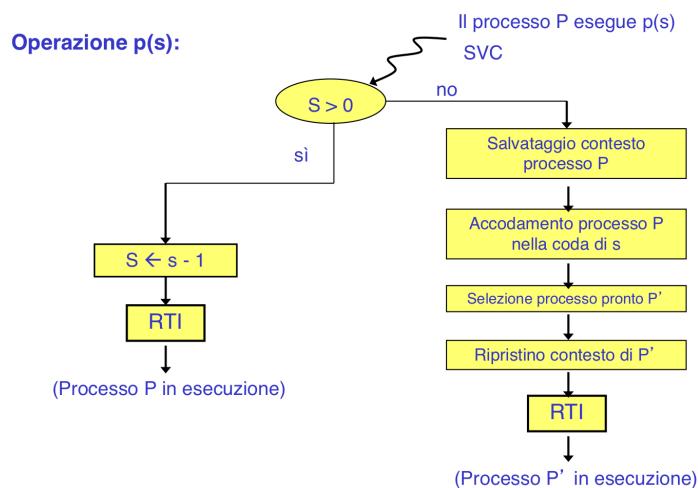


Se non ci sono processi in coda, il puntatore contiene la costante *nil* (ad esempio, -1).

La coda viene gestita con politica FIFO: i processi risultano ordinati secondo il loro tempo di arrivo nella coda associata al semaforo.

Il descrittore di un processo viene inserito nella coda del semaforo come conseguenza di una primitiva P non passante; viene prelevato per effetto di una V.

```
||| typedef struct {
    int contatore;
    coda_a_livelli coda;
} descr_semaforo;
```

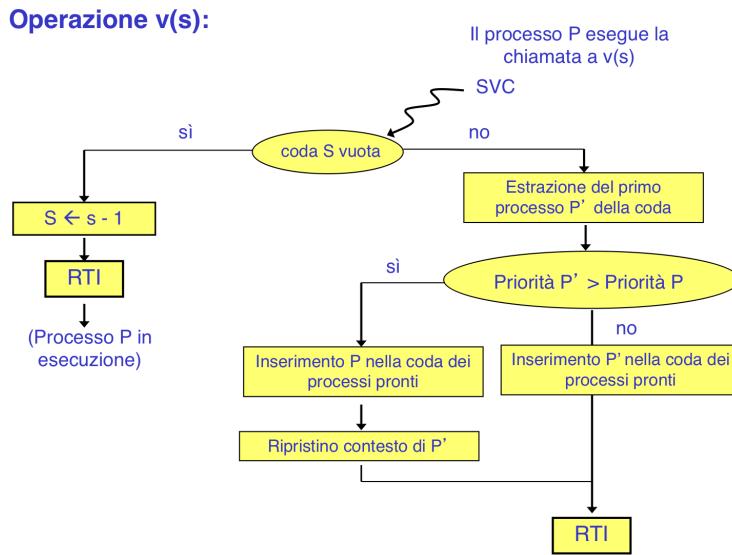


```
/* insieme di tutti i semafori: */
descr_semaforo semafori[num_max_sem];

/*ogni semaforo e' rappresentato dall'indice che lo
individua nel vettore semafori/ */

typedef int semaforo;

void P (semaforo s) {
    int j, k;
    if (semafori[s].contatore==0) {
        Salvataggio_stato();
        j=processo_in_esecuzione;
        k=descrittori[j].servizio.priorità;
        Inserimento(j,semafori[s].coda[k]);
        Assegnazione_CPU();
        Ripristino_stato();
    }
    else contatore--;
}
```



```

void V(semaforo s) {
    int j,k,p,q; /* j,k: processi; p,q: indici priorità*/
    q=0;
    while (semafori[s].coda[q].primo== -1 && q < min_priorità)
        q++;
    if (semafori[s].coda[q].primo!= -1) {
        k = Prelievo(semafori[s].coda[q]);
        j = processo_in_esecuzione;
        p = descrittori[j].servizio.priorità;
        if( p < q) // il processo in esecuzione è prioritario
            Inserimento (k,coda_processi_pronti[q]);
        else { // preemptio
            Salvataggio_stato();
            Inserimento(j, coda_processi_pronti[p]);
            processo_in_esecuzione = k;
            Ripristino_stato();
        }
    }
    else semafori[s].contatore++;
}
  
```

5.3.4 Meccanismo di passaggio dall'ambiente di nucleo all'ambiente dei processi e viceversa.

E' costituito dal meccanismo di interruzioni (esterne o asincrone, interne o sincrone). In entrambi i casi, al completamento della funzione richiesta, il trasferimento all'ambiente di utente avviene utilizzando il meccanismo di ritorno da interruzione.

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

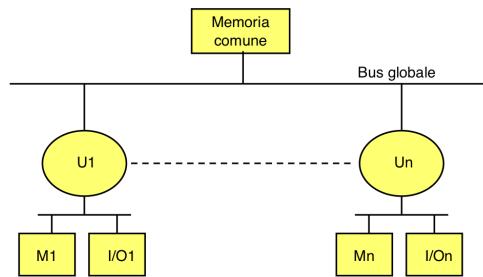
Registri: PC e PS (registro di stato Program Counter e Program Status), R1, R2,.. Rn, R1', R2'.. Rn', SP1, SP1' (registri generali e stack pointer) associati rispettivamente agli ambienti di nucleo e dei processi.

L'esecuzione di una primitiva da parte di P corrisponde all'esecuzione di una istruzione di tipo SVC (SuperVisor Call):

1. **Interruzione** di tipo sincrono (esterna, quelle interne sono di tipo sincrono)
2. **Salvataggio di PC e PS** relativi a P in cima alla pila del nucleo.

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

5.4 Realizzazione nucleo (Architettura Multiprocessore)



Per quanto riguarda le **architetture multiprocessore** sono possibili **vari modelli**:

- **Modello SMP:** unica copia del nucleo condivisa tra tutte le CPU (l'unico nucleo si occupa di tutto)
- **Modello a nuclei distinti:** più istanze di nucleo concorrenti (esiste una collezione di nuclei che esegue in modo concorrente)

5.4.1 Simmetric Multi Processing

Esiste un'unica copia del nucleo nella memoria comune che si occupa della gestione di tutte le risorse disponibili (es. CPU); ogni processo può operare su ogni unità di elaborazione.

La competizione tra CPU nell'esecuzione del nucleo porta alla necessità di **sincronizzazione**:

- **Soluzione ad un solo lock:** Accesso esclusivo alle sue strutture dati può essere ottenuto con le primitive *lock* e *unlock* su un *unico lock* associato al nucleo condiviso (soluzione hardware al problema della mutua esclusione).
 - Limitazione del grado di parallelismo, escludendo a priori ogni possibilità di esecuzione contemporanea di più funzioni del nucleo (ad esempio due P su due semafori diversi).
- **Soluzione a più lock:** Un maggior grado di concorrenza può essere ottenuto suddividendo il nucleo in molteplici sezioni critiche indipendenti. Ad esempio, proteggendo: la coda dei processi pronti; i singoli semafori (*tramite lock distinti, gestiti con lock e unlock*).

Anche se un processo può essere eseguito da ogni processore, può risultare più efficiente assegnarlo ad un determinato processore: i processori possono accedere più rapidamente alla loro memoria privata piuttosto che a quella remota (sistemi con accesso alla memoria non uniforme); il processo dovrebbe essere eseguito sul processore la cui memoria privata contiene il suo codice.

5.4.2 Realizzazione SMP

Tutte le CPU condividono lo stesso nucleo, quindi per sincronizzare gli accessi al nucleo, **le strutture dati del nucleo vengono protette tramite lock** in particolare utilizzando **singoli semafori e code dei processi distinti**. In questo caso due operazioni P su semafori diversi possono operare in modo **contemporaneo**, in caso contrario vengono **sequenzializzati** solo gli accessi alla coda dei processi pronti.

Se si ha uno **scheduling pre-emptive basato su priorità** allora si rischia che l'esecuzione di una V possa portare l'attivazione di un processo con priorità superiore a quella di almeno uno dei processi in esecuzione. Quindi il nucleo deve provvedere a revocare la CPU al processo con priorità più bassa ed assegnarla al processo risvegliato dalla V. Inoltre sarà necessario un **meccanismo di segnalazione** tra le unità di elaborazione.

Segnalazione inter-processore Ad esempio, siano:

P_i il processo che esegue la V eseguendo sul nodo U_i

P_j il processo riattivato

P_k il processo a più bassa priorità che esegue su U_k :

Il nucleo attualmente eseguito da U_i invia un segnale di interruzione a U_k .

U_k , utilizzando le funzioni del nucleo, inserisce P_k nella coda dei processi pronti e mette in esecuzione il processo P_j .

5.4.3 Nuclei distinti

Ogni singola CPU viene gestita da un nucleo distinto. Questo modello si basa sull'ipotesi che l'insieme di processi sia partizionabile in tanti **nodi virtuali**, ciascuno dei quali è **assegnato ad un nodo fisico**. Tutte le informazioni relative al nodo virtuale vengono allocate sulla memoria privata del nodo fisico.

Se nella memoria privata vengono allocate anche le funzioni del nucleo allora tutte le iterazioni locali al nodo virtuale possono avvenire **indipendentemente** e in modo **concorrente** a quelle degli altri nodi virtuali. Solo le iterazioni tra processi appartenenti a nodi virtuali diversi utilizzano la memoria comune.

Distinzione tra:

- **Semafori privati** di un nodo U, cioè utilizzati da processi appartenenti al nodo virtuale U. Vengono realizzati e gestiti come nel caso monoprocesso.
- **Semafori condivisi** tra nodi, cioè utilizzati da processi appartenenti a nodi virtuali differenti.

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

5.4.4 Realizzazione nuclei distinti

Ogni semaforo condiviso viene rappresentato nella memoria comune da un intero non negativo; l'accesso al semaforo sarà protetto da un lock x usato da $lock(x)$ e $unlock(x)$.

Rappresentante del processo:

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

Per ogni semaforo condiviso S vengono mantenute 2 tipi di code diverse:

- Su ogni nodo N: una coda locale a N contenente i descrittori dei processi locali sospesi su S; la coda risiede nella memoria privata del nodo e viene gestita esclusivamente dal nucleo del nodo.
- una coda globale dei rappresentanti di tutti i processi sospesi su S, accessibile da ogni nucleo.

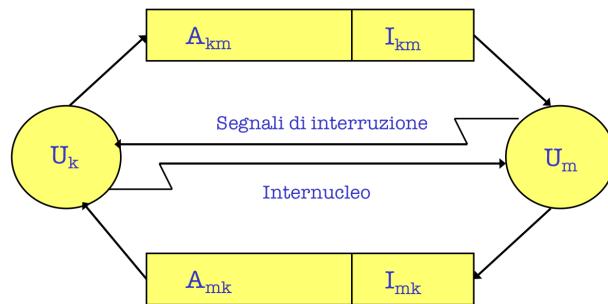
Esecuzione di una P sospensiva su un semaforo S condiviso tra nodi virtuali: il nucleo N_k del nodo fisico k-esimo sul quale opera il processo P_i che ha chiamato la P, provvede a:

- inserire il rappresentante di P_i nella coda globale dei rappresentanti associata a S.
- inserire il descrittore di P_i nella coda locale a N_k associata a S.

Esecuzione di una V su un semaforo (con rappresentanti in coda): Assumendo che la V sia chiamata dal processo P_j appartenente al nodo U_k :

1. il rappresentante del processo P_j che appartiene al nodo U_m viene eliminato dalla coda associata al semaforo (es. FIFO);
2. il nucleo N_k provvede a **comunicare** l'identità del processo P_j risvegliato al nucleo N_m ;
3. N_m provvede a riattivare il processo P_j , estraendo il suo descrittore dalla coda locale.

Comunicazione tra nuclei



A_{km} , A_{mk} sono aree di comunicazione tra i nodi U_k e U_m , entrambe mappate in memoria comune:

1. N_k inserisce in A_{km} l'identità del processo P_j risvegliato e lancia un'interruzione a N_m .
2. Rispondendo a questa interruzione N_m provvede a prelevare le informazioni contenute in A_{km} ed a eseguire le operazioni già descritte.

Analogamente per le comunicazioni tra N_m e N_k .

Per sincronizzare gli accessi alle aree di comunicazione:

1. I_{km} e I_{mk} sono indicatori che servono per sincronizzare le operazioni dei due nuclei.
2. Se I_{km} ha valore 1 significa che l'informazione contenuta in A_{km} non è ancora stata prelevata da N_m ; se ha valore 0 significa che A_{km} è disponibile per un ulteriore messaggio da N_k a N_m

```

void P(semaforo sem) {
    if (<sem appartiene alla memoria privata>
        <come nel 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>
        <caso monoprocessore>
    else {
        lock(x)
        if (<la coda non è vuota>
            if (<il processo appartiene al nodo del segnalante>
                <caso monoprocessore>;
            else {
                <estraz. processo dalla coda dei rappresentanti>;
                <"si determina l'area di comunicazione con il nodo cui il
                    processo appartiene">;
                if (<area occupata>) <si aspetta>;
                else <"si inserisce nell'area l'identificatore del
                    processo riattivato e si pone l'indicatore a 1">;
                <"si invia interrupt al nodo cui appartiene il processo">;
            }
        else <si incrementa il valore del semaforo>;
        unlock(x)
    }
}

```

5.5 SMP vs Nuclei Distinti

- Grado di parallelismo tra CPU: il secondo modello è più vantaggioso, in quanto il grado di accoppiamento tra CPU è più basso; maggiore scalabilità.
- Distribuzione del carico computazionale: il primo modello fornisce i presupposti per un migliore bilanciamento del carico (mantenimento del carico delle CPU bilanciato) perché lo scheduler può decidere di allocare ogni processo su qualunque CPU. Il secondo modello, invece, vincola ogni processo ad essere schedulato sempre sullo stesso nodo.

Capitolo 6

Modello a scambi di messaggi

Ogni processo può accedere esclusivamente alla risorse allocate nella propria memoria (virtuale) locale poiché si utilizza un'architettura del tipo **distributed-memory multicomputers and networks**. Inoltre ogni risorsa del sistema è accessibile ad un solo processo, quindi se una risorsa è necessaria a più processi ciascuno di questi (processi Client) dovrà richiedere all'unico processo che può operare sulla risorsa (processo Server) di eseguire l'operazione richiesta e restituirgli i risultati.

Il processo Server rappresenta anche il gestore della risorsa in questo modello architettonico. Il meccanismo base utilizzato dai processi per qualunque tipo di iterazione è costituito dal meccanismo di scambio di messaggi.

6.1 Canali di comunicazione

Un canale è un collegamento logico mediante il quale due o più processi comunicano. Il nucleo della macchina concorrente realizza l'astrazione “canale” come meccanismo primitivo per lo scambio di informazioni.

E' compito del linguaggio di programmazione offrire gli strumenti di alto livello per specificare i canali di comunicazione e utilizzarli per le iterazioni tra processi.

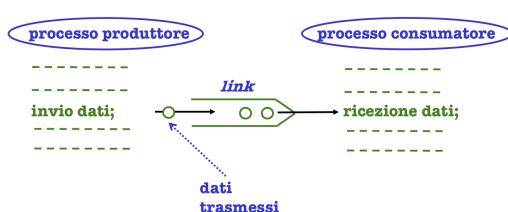
Parametri che caratterizzano il concetto di canale

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

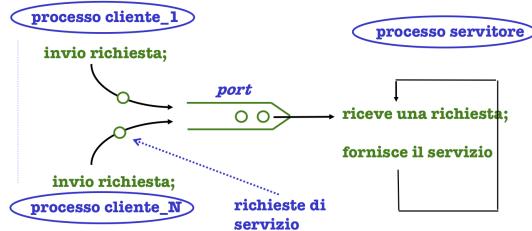
In base al **flusso di dati** distinguiamo i canali in **monodirezionali**, che prevedono il flusso di dati in una sola direzione (mittente verso ricevente), e **bidirezionali**, usati sia per inviare che per ricevere informazioni (es. cliente-servitore).

Inoltre i canali possono essere di 3 tipi in relazione al numero di processi comunicanti:

1. **Link** canale simmetrico: da uno a uno (tipo Produttore/Consumatore)



2. Port canale asimmetrico: da molti a uno (tipo Client/Server)



3. Mailbox canale asimmetrico: da molti a molti

I canali possono avere **3** differenti tipo di sincronizzazione:

- **Comunicazione asincrona:** il processo mittente continua la sua esecuzione immediatamente dopo che il messaggio è stato inviato.

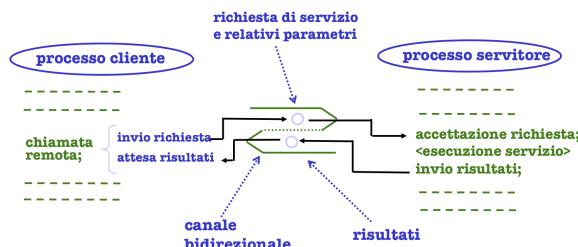
La send quindi non è un punto di sincronizzazione (è necessario prevedere uno scambio esplicito di messaggi), e la comunicazione asincrona crea un disaccoppiamento tra mittente e destinatario: il messaggio ricevuto contiene informazioni che non possono essere associate allo stato attuale del mittente (carenza espressiva).

A livello implementativo richiede la presenza di un **buffer** di capacità illimitata per poter memorizzare i messaggi inviati e non ancora ricevuti. In caso di coda dei messaggi pieni c'è bisogno di sospendere il processo che invia i messaggi.

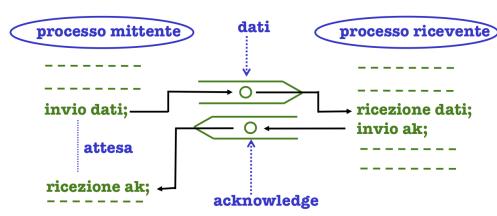
- **Comunicazione sincrona (rendez-vous semplice):** il primo dei due processi comunicanti che esegue o l'invio o la ricezione si sospende in attesa che l'altro sia pronto ad eseguire l'operazione duale.

La send è un punto di sincronizzazione. Non è necessario un buffer perché un messaggio viene inviato solo se l'altro è pronto a riceverlo.

- **Comunicazione con sincronizzazione estesa** (rendez-vous esteso): il processo mittente rimane in attesa fino a che il ricevente non ha terminato di svolgere l'azione richiesta. Modello Client/Server in cui il Client rimane in attesa fino a che il server non ha terminato l'operazione richiesta.



E' possibile rendere una comunicazione sincrona a partire da una comunicazione asincrona mediante l'utilizzo di un ack:



6.2 Primitive di comunicazione

6.2.1 Port

Un canale port identifica un canale asimmetrico **molti-a-uno**.

Dichiarazione canale: `port <tipo> <identificatore>`

```
|| port int ch1;
```

L'identificatore `ch1` denota un canale utilizzato per trasferire messaggi di tipo integer.

port viene dichiarato locale a un processo (il ricevente) ed è visibile ai processi mittenti mediante la dot notation:

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

6.2.2 Send

Primitiva di invio: `send(<valore>) to <porta>;`

- `<porta>` identifica in modo univoco il canale a cui inviare il messaggio ad esempio:
`<nome processo>.<nome locale della porta>;`
- `<valore>` identifica una espressione dello stesso tipo di `<porta>` e il cui valore rappresenta il contenuto del messaggio inviato

```
|| port int ch1;
  send(125) to P.ch1;
```

Il processo che la esegue invia il valore 125 al processo P tramite il canale ch1 da cui solo P può ricevere.

6.2.3 Receive

Primitiva di ricezione: `receive(<variabile>) from <porta>;`

- `<porta>` identifica il canale, locale al processo ricevente, dal quale ricevere il messaggio;
- `<variabile>` è l'identificatore di una variabile dello stesso tipo di `<porta>` a cui assegnare il valore del messaggio ricevuto.

Semantica bloccante: La primitiva sospende il processo se non ci sono messaggi sul canale e restituisce un valore del tipo predefinito `process` che identifica il nome del processo mittente (utilizzando un canale asincrono devo risalire a chi ha inviato il messaggio).

```
|| proc = receive(m) from ch1;
```

Il processo che la esegue si sospende se sul proprio canale `ch1` non ci sono messaggi altrimenti estrae il primo di loro e ne assegna il valore a `m`; assegna alla variabile `proc` (di tipo `process`), il nome del processo che ha inviato il messaggio.

Receive bloccante e Modello Client-Server Dal lato server vi è la possibilità di ricevere diverse richieste di servizio; più canali di ingresso ciascuno dedicato ad un tipo di richiesta.

Scelta del canale sul quale eseguire la receive. Può compromettere il blocco su un canale in presenza di messaggi su un altro (eventuale attesa infinita). Per risolvere questo problema si utilizza un **receive con semantica non bloccante**: verifica lo stato del canale, restituisce un messaggio (se presente) o un'indicazione di canale vuoto (non bloccante).

Il server può eseguire un ciclo di ispezione/ricezione di/da tutti i canali. In caso di presenza contemporanea di messaggi su più canali la scelta può essere fatta secondo diversi criteri (priorità, stato della risorsa), in modo non deterministico. Di contro si ha il **problema dell'attesa attiva**: in caso di assenza di messaggi su tutti i canali, il server continua a ripetere.

Un meccanismo ideale sarebbe quello che:

- consente al processo server di verificare contemporaneamente la disponibilità di messaggi su due (o più) canali;
- abilita la ricezione di un messaggio da uno qualunque di essi contenente messaggi;
- blocca il processo in attesa che arrivi un messaggio, qualunque sia il canale su cui arriva, quando nessun canale contiene messaggi.

Questo meccanismo è realizzabile tramite i comandi con guardia.

6.2.4 Comando con guardia

Sintassi:

```
||| <guardia> -> <istruzione>;
||| <guardia> := (<espressione booleana>; <primitiva receive>);
```

Una guardia viene valutata e può fornire tre diversi valori:

- **guardia fallita**: se l'espressione booleana ha il valore **false**.
Semantica: il comando fallisce (non produce alcun effetto).
- **guardia ritardata**: se l'espressione booleana ha valore **true**, ma sul canale su cui viene eseguita non ci sono messaggi (la receive è bloccante).
Semantica: il processo che esegue il comando viene sospeso; quando arriva il primo messaggio il processo viene riattivato, esegue la receive e successivamente esegue <istruzione>.
- **guardia valida**: se l'espressione booleana ha valore **true** e la receive può essere eseguita senza ritardi.
Semantica: la receive viene eseguita ed il processo esegue <istruzione>.

6.2.5 Comando con guardia alternativo

Sintassi:

```
if
  [ ] <guardia_1> -> <istruzione_1>;
  [ ] <guardia_2> -> <istruzione_2>;
  ...
  [ ] <guardia_n> -> <istruzione_n>;
fi
```

Semantica: Vengono valutate le guardie di tutti i rami. Si possono verificare 3 casi:

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

6.2.6 Comando con guardia ripetitivo

Sintassi:

```
|| do
  [ ] <guardia_1> -> <istruzione_1>;
  ...
  [ ] <guardia_n> -> <istruzione_n>;
|| od
```

Semantica: Vengono valutate le guardie di tutti i rami:

1. **se una o più guardie sono valide** viene scelto, in maniera non deterministica, uno dei rami con guardia valida e la relativa guardia viene eseguita (viene cioè eseguita la receive contenuta nella guardia scelta); viene quindi eseguita l'istruzione relativa al ramo scelto e, successivamente, l'esecuzione dell'intero comando viene ripetuta.
2. **se tutte le guardie non fallite sono ritardate**, il processo in esecuzione si sospende in attesa che arrivi un messaggio che abilita la transizione di una guardia da ritardata a valida; a quel punto procede come nel caso precedente;
3. **se tutte le guardie sono fallite** l'esecuzione del comando termina.

Il funzionamento del comando con guardia ripetitivo è lo stesso del comando con guardia alternativo solo che la valutazione ed esecuzione dei rami viene svolta in maniera ciclica. Se tutte le guardie sono fallite solo in quel caso si esce dal ciclo.

Esempio di processo servitore

```
process server {
    port int servizioA;      //esecuzione di A() su R
    port real servizioB;     //esecuzione di B() su R
    Tipo_di_R R;
    int x;
    real y;
    do
        [] (condA); receive (x) from servizioA; ->
            { R.A(x);
              < eventuale restituzione dei risultati al cliente>; }
        [] (condB); receive (y) from servizioB; ->
            { R.B(y);
              < eventuale restituzione dei risultati al cliente>; }
    od
}
```

6.3 Primitive Asincrone

Nel modello a scambio di messaggi, lo strumento di comunicazione di più basso livello è la **send asincrona**.

Prendiamo in considerazione alcuni **tipici problemi** di interazione nel modello a scambio di messaggi: accesso a risorse “condivise” → processi servitori:

1. Una sola operazione
2. Più operazioni mutuamente esclusive
3. Più operazioni con condizioni di sincronizzazione

Ad ogni operazione associata alla risorsa corrisponde un diverso servizio.

L'obiettivo è quello di individuare schemi di soluzioni ai diversi problemi basati sulla send asincrona e confrontarli con le soluzioni dello stesso problema nel modello a memoria comune usando il monitor.

6.3.1 Risorsa condivisa con una sola operazione

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

A1) **Soluzione nel modello a memoria comune:** monitor con una operazione entry.

```
monitor tipo_ris {
    tipo_var var;
    <eventuale istruzione di inizializzazione>
    entry tipo_out fun (tipo_in x) {
        <corpo della funzione fun>;
    }
}
tipo_ris ris;

process client {
    tipo_in a;
    tipo_out b;
    .....
    b=ris.fun(a);
    .....
}
```

B1) **Soluzione nel modello a scambio di messaggi:** processo servitore che offre un unico servizio senza condizioni di sincronizzazione.

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

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

Un solo canale risposta di tipo *tipo_out* e un solo canale input di tipo *tipo_in*.

6.3.2 Risorsa condivisa con più operazioni

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

A2) **Soluzione nel modello a memoria comune:** monitor con due operazioni entry.

```
monitor tipo_ris {
    tipo_var var;
    { <eventuale istruzione di inizializzazione>; }
    entry tipo_out fun1 (tipo_in1 x1) {
        <corpo della funzione fun1>;
    }
    entry tipo_out fun2 (tipo_in2 x2) {
        <corpo della funzione fun2>;
    }
}
...
tipo_ris ris
```

B2.1) **Soluzione nel modello a scambio di messaggi:** Processo servitore che offre 2 servizi senza condizioni di sincronizzazione.

Soluzione senza comandi con guardia. un solo canale per entrambi i tipi di richiesta.

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

tipo_out1 fun1 (tipo_in1 x1);
tipo_out2 fun2 (tipo_in2 x2);
process server {
    port in_mes input;
    tipo_var var;
    process p;
    in_mes richiesta;
    tipo_out1 y1;
    tipo_out2 y2;
    while (true) {
        p=receive (richiesta) from input;
        switch (richiesta.servizio) {
            case fun1: {
                y1=fun1(richiesta.parametri.x1);
                send(y1) to p.risposta1;
                break; }
            case fun2: {
                y2=fun2(richiesta.parametri.x2);
                send(y2) to p.risposta2;
                break; }
        }
    }
}

process cliente {
    port tipo_out1 risposta1;
    port tipo_out1 risposta1;
    tipo_in1 a1;
    tipo_in2 a2;
    tipo_out1 b1;
    tipo_out2 b2;
    process p;
    .....
    if (<servizio1>
        .....
    then {
```

```

        send(a1)to server.input;
        p=receive(b1)from risposta1;
    }
    else {
        send(a2)to server.input;
        p=receive(b2)from risposta2;
    }
    .....
}

```

B2.2) **Soluzione con comandi con guardia:** due diversi canali per i due tipi di risorse.

```

tipo_out1 fun1(tipo_in1 x1);
tipo_out2 fun2(tipo_in2 x2);
process server {
    port tipo_in1    input1;
    port tipo_in2    input2;
    tipo_var var;
    process p;
    tipo_in1 x1; tipo_in2 x2;
    tipo_out1 y1; tipo_out2 y2;
{< eventuale istruzione di inizializzazione>};
do
    [] p = receive (x1) from input1; ->
        y1=fun1(x1);
        send (y1) to p.risposta1;
    [] p = receive (x2) from input2; ->
        y2=fun2(x2);
        send (y2) to p.risposta2;
od;
}

```

6.3.3 Risorsa condivisa con più operazioni e condizioni di sincronizzazione

Risorsa condivisa che mette a disposizione di un insieme di processi clienti due operazioni con condizioni di sincronizzazione.

A3) **Soluzione nel modello a memoria comune:** monitor con 2 entry e 2 variabili condizione:

```

entry tipo_out1 op1 (tipo_in1 x1) {
    .....
    if (!cond1) wait (c1); // c1 variabile condizione
    .....
    signal (c2); // c2 variabile condizione
    .....
}

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

```

B3) **Soluzione nel modello a scambio di messaggi:** processo servitore con piu' servizi (due) con la specifica di condizioni di sincronizzazione

```

tipo_out1 fun1(tipo_in1 x1);
tipo_out2 fun2(tipo_in2 x2);
process server {
    port tipo_in1    input1;
    port tipo_in2    input2;
    tipo_var var;
    process p;
    tipo_in1 x1; tipo_in2 x2;
    tipo_out1 y1; tipo_out2 y2;
    { < eventuale istruzione di inizializzazione>; }
    do
        [] (cond1); p = receive (x1) from input1; ->
            y1=fun1(x1);
            send (y1) to p.risposta1;
        [] (cond2); p = receive (x2) from input2; ->
            y2=fun1(x2);
            send (y2) to p.risposta2;
    od;
}

```

6.4 Corrispondenza tra monitor e processi servitori

modello a memoria comune	corrisponde	modello a scambio di messaggi
risorsa condivisa: istanza di un monitor	→	risorsa condivisa: struttura dati locale a un processo server
identificatore di funzione di accesso al monitor	→	porta del processo server
tipo dei parametri della funzione	→	tipo della porta
tipo del valore restituito dalla funzione	→	tipo della porta da cui il processo cliente riceve il risultato
per ogni funzione del monitor	→	un ramo (comando con guardia) dell'istruzione ripetitiva che costituisce il corpo del server

modello a memoria comune	corrisponde	modello a scambio di messaggi
condizione di sincronizzazione di una funzione entry	→	guardia logica componente del ramo corrispondente alla funzione
chiamata di funzione entry	→	invio (send) della richiesta sulla corrispondente porta del server e attesa (receive) dei risultati sulla propria porta.
esecuzione in mutua esclusione fra le chiamate alle funzioni del monitor	→	scelta di uno dei rami con guardia valida del comando ripetitivo del server
corpo della funzione	→	istruzione del ramo corrispondente alla funzione

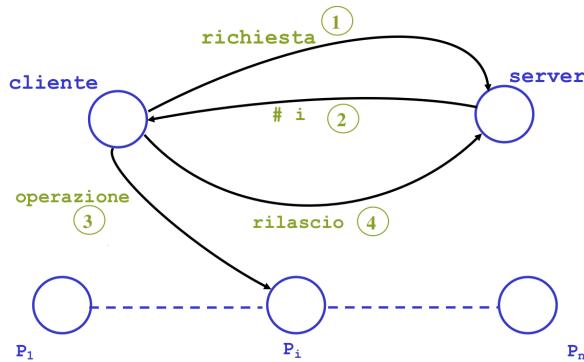
6.5 Esempi di processi servitori

6.5.1 Pool di risorse equivalenti

Ogni risorsa è gestita da un **processo**:

Il processo **server** indica un processo P_i la cui risorsa è attualmente disponibile.

Il client invia al server una richiesta per sapere l'indice di una risorsa disponibile. Il server restituisce l'indice i del processo che gestisce la risorsa disponibile; successivamente il cliente interagisce direttamente con P_i per richiedere l'esecuzione di operazioni sulla risorsa R_i . Quando la risorsa non è più necessaria il client invia al server un messaggio per indicare il rilascio della risorsa.



```

process cliente {
    port int risorsa;
    signal s;
    int r;
    process p, server;
    .....
    send(s) to server.richiesta;      // richiesta della risorsa
    p=receive(r) from risorsa;        // attesa della risposta
        <uso delle risorse r-sima>
    send(r) to server.rilascio;        // rilascio della risorsa
    ...
}

```

```

process server {
    port signal richiesta;
    port int rilascio;
    int disponibili=N;
    boolean libera[N];
    process p ;
    signal s;
    int r;
    for (int i=0; i<N; i++) libera[i]=true; //inizializzazione
    do
        [] (disponibili>0); p= receive(s)from richiesta; ->
            int i=0;
            while (!libera[i]) i++;
            libera[i] = false;
            disponibili--;
            send(i) to p.risorsa;
        [] p=receive(r)from rilascio; ->
            disponibili++;
            libera[r] = true;
    od;
}

```

6.5.2 Simulazione di un semaforo

```

process semaphore {
    port signal P;
    port signal V;
    int valore = 1;
    process proc ;
    signal s;
    do
        [] (valore>0); p=receive(s)from P; ->
            valore--;
            send (s)to proc.risposta;}
    [] p=receive (s) from V; ->
            valore++;
    od; }

```

Cliente:

```

signal s;
/* chiamata di P:*/
send(s) to semaphore.P;
receive(s) from risposta;
...
/* chiamata di V: */
send(s) to semaphore.V;

```

6.5.3 Pool di risorse con priorità

Si consideri un server che gestisce un pool di risorse adottando una politica basata su un **criterio di priorità** (il server deve privilegiare le richieste di P0 rispetto a quelle di P1 e queste rispetto a quelle di P2 ...).

Pertanto:

- Quando un qualunque processo Pi invia al server una richiesta, questa **viene servita se ci sono risorse disponibili**, mentre, in caso contrario, Pi rimane sospeso.
- Quando una risorsa viene rilasciata, il server deve essere in grado di riconoscere se ci sono processi clienti sospesi e, in questo caso, quali sono.
- Tra i clienti sospesi, **va scelto quello a priorità più alta** (indice più basso).

```

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

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; j<M; j++) bloccato[j]=false;
    client[0]=P0..... client[M-1]=PM-1;
}
do
[] p=receive(s)from richiesta; ->
    if (disponibili>0 {
        int i=0;
        while (!libera[i]) i++;
        libera[i] = false;
        disponibili--;
        send (i) to p.risorsa; }
    else { int j=0; sospesi++;
        while(client[j]!=p) j++;
        bloccato[j]=true; }
[] p:= receive (r) from rilascio; ->
    if (sospesi == 0) {
        disponibili++;
        libera[r] = true; }
    else { int i=0;
        while (!bloccato[i]) i++;
        sospesi--;
        bloccato[i] = false;
        send (r)to client[i].risorsa; }
}
od;
}

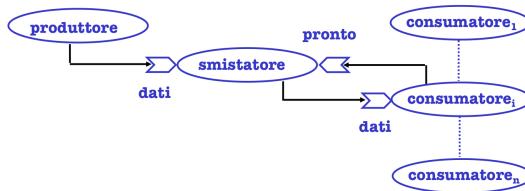
```

6.5.4 Produttori & Consumatori

Nel caso in cui vi siano più produttori e uno ed un solo consumatore **l'introduzione di un processo server è superflua** in quanto la soluzione è offerta direttamente dalla porta a cui un produttore invia i messaggi ed il produttore li preleva.

Nel caso in cui si prevedano più processi consumatori (molti-a-molti) è necessario interporre tra produttori e consumatori un processo server (smistatore), a cui i produttori inviano i messaggi , con il compito di scegliere il consumatore cui inviare il messaggio in base alla disponibilità a ricevere messaggi che i singoli consumatori comunicano al server.

L'uso della send asincrona elimina la necessità di introdurre esplicitamente un buffer in quanto i canali di comunicazione contengono già, al loro interno, un buffer di dimensione teoricamente illimitata.



Il processo smistatore ha due porte:

- dati per ricevere dati dai produttori
- pronto per ricevere il messaggio inviato dai consumatori pronti a ricevere il messaggio.

Il messaggio inviato dal consumatore è privo di contenuto informativo (messaggio di tipo **signal**); è semplicemente un segnale di controllo.

T è il tipo generico dei messaggi inviati dai produttori: sia la porta dati del server che quella del consumatore è di tipo T.

Ogni consumatore quando desidera ricevere un messaggio da un produttore, invia un messaggio di tipo signal sulla porta pronto del server e si mette in attesa di ricevere il messaggio sulla propria porta dati.

```

process smistatore {
    port T dati;      //dimensione buffer data dalla capacità del canale dati
    port signal pronto;
    T messaggio;
    process prod, cons;
    signal s;
    while (true) {
        cons = receive(s)from pronto;
        prod = receive(messaggio)from dati;
        send(messaggio)to cons.dati;
    }
}

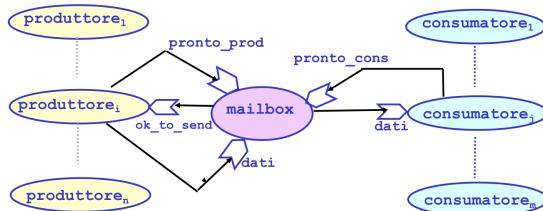
```

Esempio di seritori: mailbox

In questo caso siamo sempre in presenza di un solo produttore e di un solo consumatore ma imponiamo il vincolo che siano al più N i messaggi già inviati ma non ancora ricevuti (equivale a imporre un **limite superiore della dimensione del buffer dei messaggi**).

Per garantire ciò il produttore non può inviare un nuovo messaggio se il **buffer è pieno** (cioè, i precedenti N messaggi non sono stati ancora ricevuti); in tal caso attende. Quindi prima di inviare un messaggio deve chiedere al server se la precedente condizione è vera inviando un segnale tramite la porta **pronto_prod** e successivamente attende il segnale dal server tramite la sua porta **ok_to_send**.

Il consumatore, a sua volta, non può estrarre un messaggio se il buffer è vuoto; in tal caso attende. Quando è pronto a ricevere un messaggio invia un segnale tramite la porta **pronto_cons** e quindi si pone in attesa di ricevere il messaggio tramite la porta **dati**.



```

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;
            contatore--;
            send(messaggio)to cons.dati;
    od;
}
process produttore_i {
    port signal ok_to_send;
    T messaggio; process p; signal s;
    .....
    <producere il messaggio>;
    send(s) to mailbox.pronto_prod;
    p=receive (s) from ok_to_send;
    send (messaggio) to mailbox.dat;
    .....
}

```

```

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

```

6.6 Realizzazione delle primitive asincrone

Nei precedenti paragrafi abbiamo supposto di disporre, a livello di linguaggio di programmazione, delle primitive di comunicazione e del costrutto `port` per definire i canali utilizzati dalle stesse primitive. Vediamo adesso come implementare tali meccanismi linguistici utilizzando gli strumenti di comunicazione messi offerti direttamente dalla macchina concorrente.

Le primitive asincrone sono sicuramente quelle più semplici e di basso livello.

Il meccanismo di comunicazione asincrona può essere realizzato a livello del nucleo del sistema secondo tre diverse tipologie architetture fisiche: monoelaboratore, multielaboratore e fisicamente distribuite.

6.6.1 Architetture mono e multielaboratore

Per specificare il meccanismo di comunicazione primitivo da realizzare facciamo le seguenti ipotesi semplificative:

1. Tutti i messaggi scambiati tra i processi sono di un **unico tipo T predefinito a livello di nucleo**(es., stringa di byte di dimensione fissa).
2. Tutti i **canali sono da molti ad uno** (port) e quindi associati al processo ricevente..
3. Essendo le primitive asincrone, **ogni porta deve contenere un buffer** (coda di messaggi) **di lunghezza indefinita**.

Per garantire l'ultimo punto andiamo a definire la struttura `messaggio` realizzando la coda di messaggi associata a una porta mediante una lista. Possiamo poi implementare la struttura `coda_di_messaggi` che realizza la coda come una lista di elementi di tipo `messaggi`:

```

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

typedef struct {
    messaggio * primo;
    messaggio * ultimo;
} cosa_di_messaggi;

```

Le funzioni per inserire un elemento nella lista(`inserisci`), per togliere un elemento da una lista(`estrai`) e la funzione booleana `coda_vuota` che restituisce true se la lista non contiene elementi:

```

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

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

```

```

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

```

Possiamo adesso definire il descrittore di una porta(`des_porta`), cioè la struttura che la rappresenta e il tipo puntatore `p_porta`:

```

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

typedef des_porta *p_porta

```

Poiché, come indicato nel punto 2), ogni porta è associata a un processo, modifichiamo leggermente la struttura dati `descrittore_processo`:

```

typedef struct {
    p_porta porte_processo[M];
    PID nome;
    modalità_di_servizio servizio;
    tipo_contesto contesto;
    tipo_stato stato;
    PID padre;
    int N_figli;
    des_fioglio prole[max_figli];
    p_des successivo;
} des_processo;

```

Il campo `stato` registra se il processo è attivo oppure bloccato; in questo caso tiene traccia delle porte sulle quali è in attesa. Quindi utilizzando tale campo del descrittore di un processo così definito possiamo specificare le seguenti funzioni per testare lo stato di un processo o per registrare una commutazione di tale stato nel processo in esecuzione:

```

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

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

```

Per specificare le due funzioni di libreria da utilizzare per inviare (`send`) o per ricevere (`receive`) un messaggio di tipo T, bisogna prima definire tre primitive di nucleo:

- `testa_porta` per se su una porta sono presenti messaggi, bloccando, in caso contrario, il processo che la esegue;
- `estrai_da_porta` per estrarre un messaggio dalla coda associata a una porta (coda non vuota), restituendone il puntatore;
- `inserisci_porta` per inserire un messaggio nella coda di messaggi associata a una porta.

```

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

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

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

```

Tramite queste tre primitive di nucleo è adesso molto semplice dettagliare il corpo delle due funzioni di libreria prima definite:

```

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

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

```

Per implementare un comando con guardia è utile disporre anche di una terza funzione di libreria, che consenta a un processo di ricevere un messaggio non da una specifica porta ma da una qualunque fra un insieme di porte:

```

int receive_any(T *inf, PID *proc, int ip[], int n) {
    messaggio * mes;
    int indice_porta;
    do
        indice_porta=testa_porta(ip,n);
    while(indice_porta== -1);
    mes = estrai_da_porta(indice_porta);
    proc = &(mes->mittente);
    inf = &(mes ->informazione);
    return indice_porta;
}

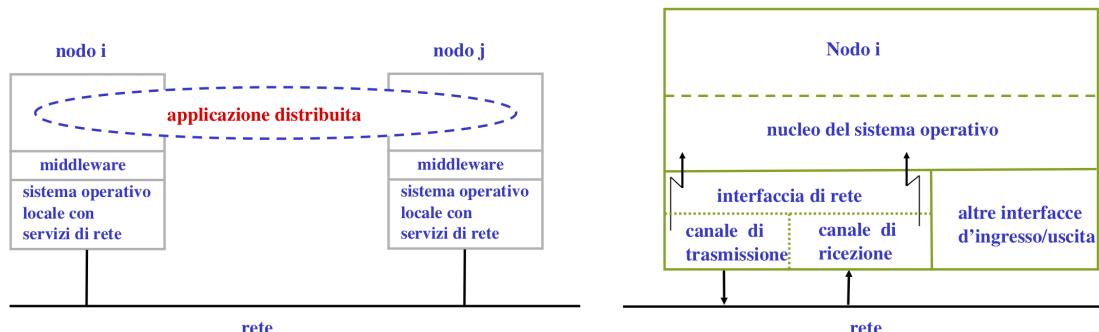
```

6.7 Architetture distribuite

Con il termine architettura distribuita si fa riferimento a un insieme di architetture fisiche caratterizzate dalla presenza di più calcolatori interconnessi tra loro al fine di costruire un unico sistema operativo e dove, a differenza delle architetture multielaboratore, non **esiste alcuna memoria condivisa**.

Indipendentemente dalla struttura fisica del sistema, ci preme mettere in evidenza la struttura astratta così come viene recepita dall'utente. Dal punto di vista software, si tende a distinguere fra due diverse tipologie di sistemi operativi:

- **sistemi operativi distribuiti (DOS - Distributed Operating System)**: caratterizzati da un insieme di nodi tra loro omogenei e tutti dotati dello stesso sistema operativo, in particolare dello stesso nucleo. Sono quindi strettamente connessi e per i quali le funzionalità del nucleo sono concettualmente le stesse del nucleo di un sistema mono o multielaboratore.
- **sistemi operativi di rete (NOS - Network Operating System)**: siamo in presenza di nodi eterogenei e, al limite, anche con sistemi operativi diversi e autonomi, nodo per nodo. Sono quindi sistemi lasciamente connessi dove però ogni nodo della rete è in grado di offrire servizi a clienti remoti presenti su altri nodi della stessa. La trasparenza della distribuzione delle risorse non viene ottenuta tramite nucleo del sistema operativo, ma per mezzo di uno strato software (middleware), che viene interposto su ogni nodo tra il sistema operativo e le applicazioni che girano sul nodo stesso.



L'unità di trasmissione tra nodi è il pacchetto. La struttura del pacchetto dipende dalla realizzazione (cioè dai protocolli di comunicazione adottati dalla rete).

Ogni interfaccia è composta da due parti distinte, canale di trasmissione e canale di ricezione.

Il canale di trasmissione viene acceduto per realizzare l'invio di pacchetti; ad esso sono associati una **coda di pacchetti**, nella quale ogni processo sender deposita il proprio pacchetto se il canale è occupato da un altro processo e un **registro buffer** della dimensione di un pacchetto.

Il canale di ricezione viene acceduto per realizzare la ricezione di pacchetti; ad esso è associato un **registro buffer**, nel quale viene depositato ogni pacchetto inviato al nodo.

Arrivo/partenza di pacchetti verso/da canali di ricezione/trasmissione vengono notificati tramite interruzioni.

Il pacchetto è l'informazione che viene trasmessa attraverso il canale: contiene, oltre al **messaggio** (inf, mittente), anche le informazioni relative al processo **destinatario** e alla **porta** di destinazione. La struttura del pacchetto dipende dalla realizzazione.

```

void invia_pacchetto (packet p) {
    if (<canale di trasmissione occupato>)
        packet_queue.inserisci(p);
    else {
        <inserimento di p nel registro buffer del canale>;
        <attivazione trasmissione>; //al termine:interruzione
    }
}

```

```

void tx_interrupt_handler() {
    packet p;
    salvataggio_stato();
    if (!packet_queue.vuota()) {
        p = packet_queue.estrai();
        <inserimento di p nel registro buffer del canale>;
        <attivazione trasmissione>;
    }
    ripristino_stato();
}

typedef struct {
    int indice_nodo;
    int PID_locale;
} PID;

void send(T inf, PID proc, int ip) {
    if (proc.indice_nodo == nome_nodo)
        local_send(inf, proc, ip);
    else remote_send(inf, proc, ip);
}

void remote_send(T inf, PID proc, int ip){
    packet p;
    PID mit=processo_in_esecuzione;
    int indice_nodo_destinatario= proc.indice_nodo;

    <vengono riempiti i vari campi del pacchetto p: in particolare, viene
     inserito nel campo relativo al nodo a cui inviare il pacchetto il
     valore indice_nodo_destinatario. Inoltre, nel campo del pacchetto
     destinato a contenere le informazioni da inviare vengono inseriti il
     nome mit del processo che invia, e i tre parametri della funzione inf,
     proc, e ip>;

    invia_pacchetto(p);
}

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 del canale>;
    <attivazione ricezione>;
    <estrazione dal campo de lpacchetto p, contenente le informazioni ricevute,
     del nome mit del mittente e dei tre parametri della funzione send inf
     , proc, e ip>;

    messaggio * m=new messaggio;
    m -> informazione = inf;
    m -> mittente = mit;
    inserisci_porta(m, proc, ip);
    ripristino_stato();
}

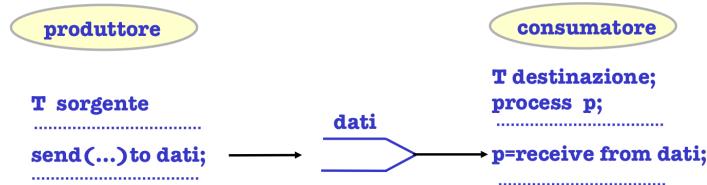
```

6.8 Primitive di comunicazione sincrone

La differenza fondamentale fra le primitive sincrone e quelle asincrone risiede nella semantica della primitiva **send** che, come è noto, prevede che il processo mittente si sincronizzi con l'esecuzione della primitiva **receive** da parte del destinatario, in modo che il trasferimento dell'informazione avvenga quando entrambi i processi sono pronti a comunicare. Ciò ha due conseguenze:

- Minor grado di concorrenza nelle primitive sincrone rispetto quelle asincrone.
- Con le primitive sincrone non è più necessaria la presenza dei buffer nei canali di comunicazione.

Per comprendere meglio quest'ultima considerazione, supponiamo di prendere in esame il caso di studio di due processi (produttore e consumatore) che si scambiano messaggi di tipo T tramite un canale simmetrico (la porta dati), nel quale cioè il solo processo produttore può inviare messaggi:



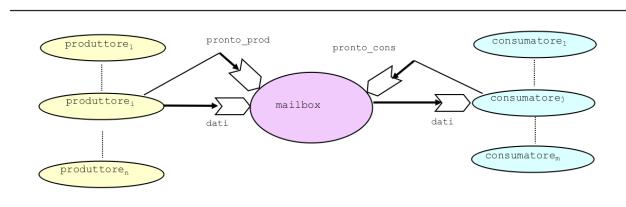
Quando il produttore esegue la **send** per inviare il messaggio (contenuto nella variabile sorgente) è del tutto inutile che tale messaggio sia memorizzato temporaneamente all'interno della porta dati in attesa di essere ricevuto. Infatti il produttore, prima di proseguire ed eventualmente inviare un nuovo messaggio, deve comunque attendere che il consumatore sia disponibile a riceverlo e a quel punto, almeno dal punto di vista teorico, il contenuto del messaggio può essere trasferito direttamente dalla variabile sorgente alla variabile destinazione, in cui il consumatore desidera riceverlo.

La differenza rispetto a un canale asimmetrico è che, in un certo istante, più processi produttori possono aver allocato la **send** e quindi, quando il consumatore esegue la **receive**, questo deve entrare in *rendez-vous* con un solo produttore, quello che ha invocato la **send** per primo, mentre gli altri restano in attesa di una successiva **receive**.

Ciò porta ad un vantaggio delle primitive sincrone rispetto alle primitive asincrone dovuto a una semplice implementazione; non è più necessaria una coda di messaggi di lunghezza infinita per ogni porta. Un secondo vantaggio è quello che riguarda la maggiore semplicità con cui vengono risolte alcune particolari interazioni tra processi, in particolare quelle che prevedono la necessità di sincronizzare la velocità dei processi stessi.

6.8.1 Mailbox di dimensioni finite

Nel caso di primitive sincrone, il problema di gestire una coda di messaggi è molto più generale e ha senso anche nell'ipotesi di un solo processo produttore e di un solo consumatore, in quanto il canale sincrono non dispone di nessun buffer al proprio interno.



Nel descrivere la soluzione, per semplicità, non viene dettagliata la realizzazione del tipo astratto **coda_messaggi** di cui, locale al processo **mailbox**, viene dichiarata l'istanza coda. In particolare, supporremo che su oggetti del tipo **coda_messaggi** si possa operare con le seguenti funzioni:

void inserimento(T mes);	<i>//per inserire mes nella coda</i>
T estrazione();	<i>//restituisce il primo elemento della coda</i>
boolean piena();	<i>// restituisce true se la coda è piena</i>
boolena vuota();	<i>// restituisce true se la coda è vuota</i>

```

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

process produttore_i {
    T messaggio;
    signal s;
    .....
    <produci il messaggio>;
    send(s) to mailbox.pronto_prod;
    send(messaggio) to mailbox.dati;
    .....
}

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

```

Possiamo eliminare l'invio del segnale sulla porta `pronto_prod`, che serve soltanto a dichiarare la disponibilità del processo a inviare un messaggio. Tale disponibilità, in virtù della sincronizzazione implicita nella `send`, può essere specificata inviando subito il messaggio nella porta dati, sapendo che comunque il processo `mailbox` lo riceverà soltanto quando la coda non è piena. Da queste considerazioni si ricava la **versione finale della `mailbox`** e di conseguenza anche quella del **generico produttore**:

```

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

```

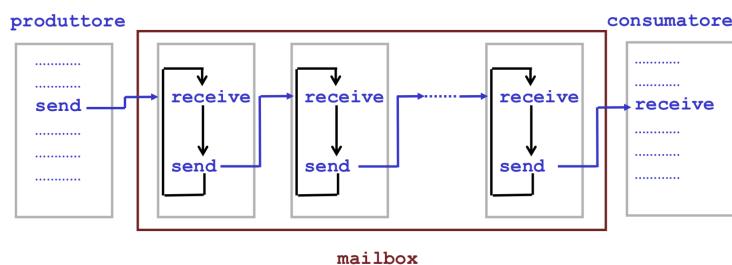
```

process produttore_i {
    T messaggio;
    process p;
    signal s;
    .....
    <produci il messaggio>;
    send(messaggio) to mailbox.dat;
    .....
}

```

Mailbox concorrente

Nel seguente caso viene realizzata una mailbox, invece che con un solo processo servitore, con un array di N processi servitori estremamente semplici e tutti uguali tra loro. Tale soluzione simula, in software, il comportamento di un registro in traslazione. Ogni elemento del registro viene simulato mediante un processo ciclico che si limita a ricevere, in un proprio buffer locale, un messaggio tramite la sola porta di cui dispone. Successivamente, invia il messaggio ricevuto alla porta del processo che lo segue nell'array e torna in testa al ciclo:



```

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

```

6.9 Specifica di strategia di priorità

Se volessimo riscrivere la soluzione relativa alla realizzazione di un processo server che alloca N risorse equivalenti agli M processi P_0, P_1, \dots, P_{M-1} in modo tale da privilegiare le richieste di P_0 rispetto a quelle di P_1 e queste rispetto a quelle di P_2 e così via.

Riportiamo di seguito la stessa soluzione nel caso particolare di una risorsa ($N=1$), soluzione di nuovo identica a quella che otterremmo utilizzando primitive asincrone:

```

process server {
    port signal richiesta;
    port int rilascio;
    boolean libera;
    process p ;
    signal s;
    int sospesi=0;
    boolean bloccato[M];
    process client[M]
    { //inizializzazione
        libera=true;
        for (int j=0; j<M; j++) bloccato[j]=false;
        client[0]='P0';.....client[M-1]='PM-1';
    }

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

```

Realizzazione delle primitive sincrone mediante semafori

```

semaphore M=0; //per la sospensione del mittente
semaphore R=0; //per la sospensione del ricevente messaggio buffer;

public void invio(T dato) {
    messaggio mes;
    mes.informazione = dato;
    mes.mittente = processo_in_esecuzione;
    buffer=mes;
    V(R);
    P(M);
}

public void ricezione(T &dato, PID &mit) {
    messaggio mes;
    P(R);
    mes=buffer;
    V(M);
    dato=mes.informazione;
    mit=mes.mittente;
}

```

Realizzazione delle primitive sincrone mediante primitive asincrone

```

void send (T inf, PID proc, int ip) {
    signal s;
    a_send(inf, proc, ip);
    a_receive(s, proc, ak);
}

void receive (T &inf, PID &proc, int ip) {
    signal s;
    a_receive (inf, proc, ip);
    a_send (s, proc, ak);
}

```

Realizzazione delle primitive sincrone come primitive di nucleo

```

typedef struct {
    T informazione;
    PID mittente;
} messaggio;

typedef struct {
    messaggio buffer[N];      // N num. massimo mittenti
    int primo, ultimo, cont;
} coda_di_N_messaggi;

typedef struct {
    coda_di_N_messaggi coda;
    p_porta successivo;
} des_porta;

typedef des_porta *p_porta;

void inserisci(messaggio m, coda_di_N_messaggi c) {
    // inserisce il messaggio m nella coda di messaggi c: in particolare un
    // messaggio proveniente da Pi viene assegnato a buffer[i]
    .....
}

messaggio estrai (coda_di_N_messaggi c) {
    // estraе dalla coda di messaggi c un messaggio e lo restituisce
    .....
}

boolean coda_vuota (coda_di_N_messaggi c) {
    // testa la coda di messaggi c per verificare se il suo buffer è vuoto
    .....
}

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

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

```

```

void testa_porta (int ip) {
    // testa la porta di indice ip del processo in esecuzione bloccandolo se
    // vuota
    p_des esec = processo_in_esecuzione; p_porta pr = esec->porte_processo[ip];
    ];
    if (coda_vuota(pr->coda)) {
        blocca_su(ip);
        assegnazione_CPU;
    }
}

void inserisci_porta (messaggio mes, PID proc, int ip) {
    // inserisce il messaggio mes nella porta di indice ip del processo proc e
    // , se questo é in attesa sulla porta , lo attiva
    p_des destinatario = descrittore(proc);
    p_porta pr = destinatario->porte_processo[ip];
    inserisci(m, pr->coda);
    if(bloccato_su(destinatario,ip)) attiva(destinatario);
}

messaggio estrai_da_porta (int ip) {
    // estrae dalla porta di indice ip (porta sicuramente non vuota) del
    // processo in esecuzione un messaggio, lo restituisce e attiva il
    // mittente del messaggio ricevuto.
    messaggio mes; p_des mit;
    p_des esec = processo_in_esecuzione;
    p_porta pr = esec->porte_processo[ip];
    mes = estrai(pr->coda); mit=descrittore(mes.mittente);
    attiva(mit);
    return mes;
}

void attendi_ricezione() {
    p_des esec = processo_in_esecuzione;
    esec->stato = <bloccato sulla send>;
    assegnazione_CPU();
}

void send (T inf, PID proc, int ip) {
    messaggio mes;
    mes.informazione = inf;
    mes.mittente = processo_in_esecuzione;
    inserisci_porta(mes, proc, ip);
    attendi_ricezione();
}

void receive (T &inf, PID &proc, int ip) {
    messaggio mes;
    testa_porta(ip);
    mes=estrai_da_porta(ip);
    proc=mes.mittente;
    inf=mes.informazione;
}

```

Capitolo 7

Linguaggio GO

Go è un linguaggio di programmazione open source sviluppato da Google.

Soddisfa le esigenze della programmazione concorrente, con primitive per lo scambio di messaggi, ed è stato progettato per ottimizzare i tempi di compilazione anche per hardware modesti. La sintassi è vicina al C eccetto per la dichiarazione dei tipi e per la mancanza di parentesi tonde nei costrutti for e if. Ha un sistema di garbage collection che si occupa autonomamente della gestione della memoria. Non include l'intercettazione di eccezioni, l'eredità dei tipi, la programmazione generica, le asserzioni e l'overloading dei metodi.

Esempio Hello World in Go:

```
|| package main
  import "fmt"
  func main() {
    fmt.Printf("Hello, World\n")
  }
```

Per compilare ed eseguire, dalla linea di comando si usa il comando go:

```
|| $ go run hello.go
```

Per la sola compilazione, il comando:

```
|| $go build hello.go
```

7.1 Costrutti del linguaggio

7.1.1 Dichiarazione

Ogni dichiarazione inizia con una **keyword** (var, const, type, func) seguita dal nome dell'oggetto dichiarato e dalle sue proprietà:

```
|| var i int                                //variabile i di tipo intero
  var k=0.99                                 // k variabile reale con v.i. =0,99
  const PI = 22./7.                           //costante PI=3.14
  type Point struct { x, y int }              //tipo Point
  func sum(a, b int) int { return a + b }     // funzione
```

Soltanto all'interno del corpo di funzioni le dichiarazioni del tipo:

```
|| var v = value
```

Possono essere espresse in modo più sintetico nella forma:

```
|| v := value
```

Il tipo è quello della costante **value**.

7.1.2 Assegnazione

L'assegnamento di valori a variabili si esprime nella forma seguente:

```
|| a = b
```

E' possibile anche l'assegnamento multiplo:

```
|| x, y, z = f1(), f2(), f3()  
|| a, b = b, a // swap
```

Le funzioni possono ritornare più di un risultato (vedi dopo), pertanto:

```
|| nbytes, error := Write(buf)
```

7.1.3 If

La condizione *if* può essere espressa con la seguente sintassi:

```
|| if <Cond> { <istruzione 1> } else {<istruzione 2>}
```

Oppure:

```
|| if <C1> {  
    <istruzione 1>  
} else if <C2> {  
    <istruzione 2>  
} else {<istruzione 3>}
```

Possibilità di inizializzazione nella condizione con il punto e virgola (funge da separatore).

7.1.4 For

Sintassi:

```
|| for i := 0; i < 10; i++ { ... }
```

E' possibile omettere gli argomenti del for:

```
|| for ;; { fmt.Printf("ciclo infinito") }
```

Oppure anche:

```
|| for { fmt.Printf("ciclo infinito") }
```

Possibilità di usare assegnamenti multipli:

```
|| for i,j := 0,N; i < j; i,j = i+1,j-1 {...}
```

7.1.5 Switch

Simile a quello del C, ma più espressivo. Come nel C:

```
|| switch a {  
    case 0: fmt.Printf("0")  
    default: fmt.Printf("non-zero")  
}
```

Diversamente dal C:

```
|| switch a,b := x[i], y[j] {  
    case a < b: return -1  
    case a == b: return 0  
    case a > b: return 1  
}
```

Espressioni di vario tipo, etichette specificate da espressioni.

7.1.6 Funzioni

Dichiarazione tramite la keyword func:

```
|| func square(f float64) float64 {  
||     return f*f  
|| }
```

Una funzione può restituire più di un valore:

```
|| func MySqrt(f float64) (float64, bool) {  
||     if f >= 0 {  
||         return math.Sqrt(f), true  
||     } else{  
||         return 0, false  
||     }  
|| }
```

7.1.7 Tipi strutturati

Array

Sintassi:

```
|| var ar [10] int
```

dove ar è un array di 10 interi (valori iniziali a 0)

Per ottenere la dimensione di un vettore, len:

```
|| X := len(ar)  
|| For i:=0; i<len(ar); i++ {  
||     ar[i]=i*i  
|| }
```

Possibilità di definire slice come sottovettori:

```
|| A := ar[2:8]           // A riferimento al sottovettore
```

Struct

Affinià con il C:

```
|| type Point struct {  
||     x, y float64  
|| }  
|| var p Point  
|| p.x = 7  
|| struct  
|| p.y = 23.4  
|| var pp *Point = new(Point) //alloc. dinamica  
|| *pp = p  
|| pp.x = 3.14 // equivalente a (*pp).x
```

Non c'è la notazione "->" per i puntatori a struct, la dereferenziazione è automatica.

7.2 Struttura dei programmi go

Un programma è composto da un insieme di moduli detti **package** ognuno definito da uno (o più) file sorgenti.

Ogni programma contiene almeno il package **main** (dal quale parte l'esecuzione); il codice di ogni package è costituito da un insieme di funzioni, che a loro volta possono riferire nomi (funzioni, tipi, variabili, costanti) esportati da altri package, usando la sintassi: "packagename.Itemname"

L'eseguibile è costruito linkando l'insieme dei package utilizzati.

L'**importazione** di nomi definiti in un package rispetta la seguente sintassi:

```
|| import "pippo"
|| ...
|| pippo.Util()
```

L'**esportazione** rispetta una regola: i nomi esportabili iniziano con una maiuscola:

```
|| package pippo
|| const GLO =100           // esportata
|| var priv float64=0,99    // privata
|| func Util(){...}          // esportata
```

Esempio package

```
|| package pigreco
|| import "math"
|| var Pi float64

|| func init() {    // init inizializza la var globale Pi
||     Pi = 4*math.Atan(1)
|| }
=====
|| package main
|| import (
||     "fmt"
||     "pigreco"
|| )
|| var twoPi = 2*pigreco.Pi
|| func main() {
||     fmt.Printf("2*Pi = %g\n", twoPi)
|| }
```

7.3 Concorrenza

L'unità di esecuzione concorrente è la **goroutine**:

è una funzione che esegue concorrentemente ad altre goroutine nello stesso spazio di indirizzamento.

Un programma go in esecuzione è costituito da una o più goroutine concorrenti.

Non sempre una goroutine corrisponde ad un **thread**, dipende dall'implementazione:

- Goroutine mappate su pthreads, 1 goroutine per thread (gccgo)
- più goroutine per thread (6g)

Per la creazione goroutine si ha la seguente sintassi:

```
|| go <invocazione funzione>
```

Ad esempio:

```
|| func IsReady(what string, minutes int64) {
||     time.Sleep(minutes*60*1e9)           // unità: nanosecondi
||     fmt.Println(what, "is ready")
|| }
|| func main() {
||     go IsReady("tea", 6)
||     go IsReady("coffee", 2)
||     fmt.Println("I'm waiting...")
|| }
```

Si hanno tre goroutine : main, Isready("tea"...), Isready("coffee"....)

7.3.1 Interazioni: canali

*"Do not communicate by sharing memory.
Instead, share memory by communicating."*

L'interazione tra processi può essere espressa tramite comunicazione attraverso canali. Il canale permette sia la comunicazione che la sincronizzazione tra goroutines.

I canali sono oggetti di prima classe:

```
|| var C chan int  
|| C=<espressione>  
|| 0p(C)
```

Proprietà del canale in go

- Simmetrico/asimmetrico, permette la comunicazione:
 - 1 - 1 ;
 - 1 - molti ;
 - molti - molti
 - molti - 1
- Comunicazione sincrona e asincrona
- bidirezionale, monodirezionale
- Oggetto tipato

Definizione:

```
|| var ch chan <tipo>           // <tipo> dei messaggi
```

Inizializzazione

Una volta definito, ogni canale va inizializzato:

```
|| var C1, C2 chan bool  
|| C1=make(chan bool)      // canale non bufferizzato: send sincrone  
|| C2=make(chan bool, 100) // canale bufferizzato: send asincrone
```

Oppure

```
|| C1:= make(chan bool)  
|| C2:= make(chan bool, 100)
```

Il valore di un canale non inizializzato è la costante **nil**.

7.3.2 Uso del canale

L'operatore di comunicazione **<->** permette di esprimere sia send che receive:

Send

```
||      <canale> <-> <messaggio>
```

Esempio:

```
|| c := make(chan int)      // c non bufferizzato  
|| c<-1                  //send il valore 1 in c (send sincrona!)
```

La freccia punta nella direzione del flusso dei messaggi !

Receive

```
||      <variabile> <- <canale>
```

Esempio:

```
|| v = <- c          // riceve un valore da c, da assegnare a v
|| <- c              // riceve un messaggio che viene scartato
|| i := <- c          // riceve un messaggio, il cui valore inizializza i
```

Semantica Di default (canali non bufferizzati), la comunicazione è sincrona. Quindi:

1. la **send** blocca il processo mittente in attesa che il destinatario esegua la receive
2. la **receive** blocca il processo destinatario in attesa che il mittente esegua la send

In questo caso la comunicazione è una forma di **sincronizzazione** tra goroutines concorrenti.

Bisogna tener conto che una receive da un canale non inizializzato (**nil**) è **bloccante**.

Esempio

```
|| func partenza(ch chan<- int) {
||     for i := 0; ; i++ { ch <- i }    // invia
|| }
|| func arrivo(ch <-chan int) {
||     for { fmt.Println(<-ch) }        // ricevi e stampa
|| }
...
ch1 := make(chan int)
go partenza(ch1)
go arrivo(ch1)
...
```

7.3.3 Sincronizzazione padre-figlio

Per imporre al padre l'attesa della terminazione di un figlio si fa uso un canale dedicato alla sincronizzazione:

```
|| ...
|| var done=make(chan bool)
|| func figlio() {
||     ...
||     done<-true
|| }
|| func main() {
||     go figlio
||     <-done // attesa figlio
|| }
```

7.3.4 Funzioni & canali

Una funzione può restituire un canale:

```
|| func partenza() chan int {
||     ch := make(chan int)
||     go func() {
||         for i := 0; ; i++ { ch <- i }
||     }
||     return ch
|| }
|| stream := partenza()    // stream è un canale int
|| fmt.Println(<-stream)    // stampa il primo messaggio: 0
```

7.3.5 Chiusura canale: close

Un canale può essere chiuso (dal sender) tramite close:

```
|| close(ch)
```

Il destinatario può verificare se il canale è chiuso nel modo seguente:

```
|| msg, ok := <- ch
```

se il canale è ancora aperto, ok è vero altrimenti è falso (il sender lo ha chiuso).

Esempio:

```
|| for {
    if ok {fmt.Println(v)} else {break}
}
v, ok := <- ch
```

range La clausola nel for ripete la receive dal canale specificato fino a che il canale non viene chiuso.

Esempio:

```
|| for v := range ch { fmt.Println(v) }
```

equivale a:

```
|| for {
    v, ok := <- ch
    if ok {fmt.Println(v)} else {break}
}
```

7.3.6 Send asincrone

Creazione di un canale bufferizzato di capacità 50:

```
c := make(chan int, 50)
go func() {
    time.Sleep(60*1e9)
    x := <- c
    fmt.Println("ricevuto", x)
}
fmt.Println("sending", 10)
c <- 10 // non è sospensiva!
fmt.Println("inviato", 10)
```

Output: sending 10 (subito) inviato 10 (subito) ricevuto 10 (dopo 60 secondi)

7.3.7 Comandi con guardia: select

Select è un'istruzione di controllo analoga al comando con guardia alternativo.

Sintassi:

```
select{
    case <guardia1>:
        <sequenza istruzioni1>
    case <guardia2>:
        <sequenza istruzioni2>
    ...
    case <guardiaN>:
        <sequenza istruzioniN>
}
```

Selezione non deterministica di un ramo con guardia valida, altrimenti attesa.

Nella select le guardie sono receive (o send): il linguaggio go non prevede la guardia logica. Si avranno guardie valide o ritardate.

Esempio:

```
|| ci, cs := make(chan int), make(chan string) select {
  ||   case v := <-ci:
  ||     fmt.Printf("ricevuto %d da ci\n", v)
  ||   case v := <-cs:
  ||     fmt.Printf("ricevuto %s da cs\n", v)
|| }
```

Possibilità di un ramo default, sempre valido.

7.4 Guardia logica

Nella select le guardie sono receive (o send): il linguaggio go non prevede la guardia logica.

Possiamo costruire le guardie logiche tramite una funzione che restituisce un canale:

```
|| func when(b bool, c chan int) chan int {
  ||   if !b {
  ||     return nil
  ||   }
  ||   return c
|| }
```

When(condizione, ch) ritorna:

- il canale **ch** se la condizione è vera
- **nil** se la condizione è falsa

Uso di when Esempio produttori/consumatori (*v. procons_sync.go*):

```
|| var pronto_prod = make(chan int)
  || var pronto_cons = make(chan int)
  || var dati = make(chan int)
  || var DATI_CONS [MAXPROC]chan int
  || var contatore int = 0
  ...
  || select {
  ||   case x := <-when(contatore < N, pronto_prod):
  ||     contatore++
  ||     msg = <-dati      // ricezione messaggio da inserire
  ||     <inserimento msg nel buffer >
  ||   case x := <-when(contatore > 0, pronto_cons):
  ||     contatore--
  ||     <estrazione msg dal buffer >
  ||     DATI_CONS[x] <- msg      // consegna messaggio a consumatore
  ||   ...
  || }
```

7.4.1 Esempio pool priorità

```

package main import (
    "fmt"
    "time" )
const MAXPROC = 10 const MAXRES = 3
const MAXBUFF = 20
var richiesta = make(chan int, MAXBUFF)
var rilascio = make(chan int, MAXBUFF)
var risorsa [MAXPROC]chan int
var done = make(chan int)           //sincro padre-figli
var termina = make(chan int)        //term.server

func client(i int) {
    richiesta <- i
    r := <-risorsa[i]

    // uso della risorsa r:
    fmt.Printf("\n [client %d] uso ris. %d\n", i, r)
    time.Sleep(time.Second * 2)
    rilascio <- r
    done <- i           //terminaz. e sincron. col padre
}

func server() {
    var disponibili int = MAXRES
    var res, p, i int
    var libera [MAXRES]bool
    var sospesi = 0
    var bloccato [MAXPROC]bool
    // inizializzazioni:
    for i := 0; i < MAXRES; i++ {
        libera[i] = true
    }
    for i := 0; i < MAXPROC; i++ {
        bloccato[i] = false
    }
    for {
        select {
        case res = <-rilascio:
            if sospesi == 0 {
                disponibili++
                libera[res] = true
            } else { // prio ai processi con indice maggiore:
                for i = MAXPROC - 1; i >= 0 && !bloccato[i]; i-- {}
                bloccato[i] = false
                sospesi--
                risorsa[i] <- res
            }
        case p = <-richiesta:
            if disponibili > 0 {
                for i = 0; i < MAXRES && !libera[i]; i++ {}
                libera[i] = false
                disponibili--
                risorsa[p] <- i
            } else {
                sospesi++
                bloccato[p] = true }
        case <-termina: // quando tutti i clienti hanno finito
            done <- 1
            return
        }
    }
}
//fine server

```

```
func main() {
    var cli int
    fmt.Printf("\n quanti clienti (max %d)? ", MAXPROC) fmt.Scanf("%d", &cli)
    fmt.Println("clienti:", cli)

    //inizializzazione canali clienti
    for i := 0; i < cli; i++ {
        risorsa[i] = make(chan int, MAXBUFF)
    }
    for i := 0; i < cli; i++ {
        go client(i)
    }
    go server()

    for i := 0; i < cli; i++ {<-done}
    termina <- 1      // terminazione server
    <-done    //attesa terminazione server
}
```

Capitolo 8

Comunicazione con sincronizzazione estesa

Come visto nelle interazioni di tipo client/servitore, è stato proposto un meccanismo di comunicazione/sincronizzazione che prevede che il processo mittente si sincronizzi con il processo servitore restando **sospeso** in attesa che il servizio richiesto sia stato completamente eseguito e che gli siano arrivati gli eventuali risultati.

Questo meccanismo è noto con il nome di **chiamata di operazione remota**. Esiste, infatti, un'analogia semantica con il meccanismo relativo alle chiamate di funzioni. Come nel caso di una chiamata di funzione, il programma chiamante continua solo dopo che l'esecuzione della funzione è terminata. La differenza sostanziale risiede nel fatto che la funzione (il servizio) viene eseguita, in realtà, remotamente da un processo diverso da chiamante.

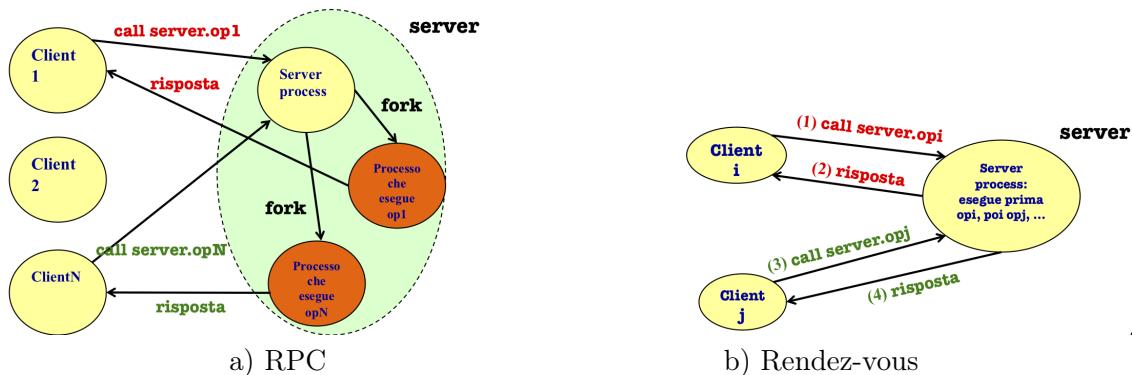
Per tenere conto delle diverse modalità di esecuzione del servizio richiesto, introdurremo **due diverse modalità di implementazione lato ricevente** (server):

- **Chiamata di procedura remota (Remote Procedure Call)**

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

- **Rendez-vous (introdotta nel linguaggio ADA)**

Prevede che l'operazione richiesta sia specificata come un insieme di istruzioni che può comparire in un punto qualunque del processo servitore. Il processo servitore utilizza un'istruzione di input (accept) che lo sospende in attesa di una richiesta dell'operazione. All'arrivo della richiesta il processo esegue il relativo insieme di istruzioni e i risultati ottenuti vengono inviati al processo chiamante.



Come si può notare la **RPC** rappresenta solamente un meccanismo di comunicazione tra processi; la possibilità che più operazioni siano eseguite concorrentemente comporta che si debba prevedere

separatamente a una sincronizzazione tra processi servitori, per esempio nell'accesso a variabili comuni. La sincronizzazione è a carico del programmatore. (ES: Java RMI, Distributed Processes).

Il **rendez-vous** combina comunicazione con sincronizzazione. Esiste, infatti, un solo processo servitore al cui interno sono definite le istruzioni che consentono di realizzare il servizio richiesto. Il processo servitore si sincronizza con il processo cliente quando esegue l'operazione di `accept`.

8.1 Chiamata di procedura remota

Le procedure chiamabili da un processo cliente possono essere esportate da uno specifico modulo di programmazione o, più in particolare, da un processo. La proposta dei *Distributed Processes*, dovuta da Brinch Hansen, che costituisce il primo esempio di linguaggio per applicazioni in tempo reale della chiamata di procedura remota, prevedeva che le procedure comuni fossero dichiarate all'interno dei processi. Ogni processo poteva accedere alle variabili locali e chiamare procedure comuni definite entro altri processi.

Nel seguito faremo l'ipotesi più generale che esista un unico componente di programmazione, il modulo, che contiene le procedure corrispondenti alle operazioni chiamabili dai processi clienti (*procedure entry*). Si supporrà inoltre che il modulo possa contenere anche processi locali, che non vanno confusi con i processi servitori creati per eseguire le operazioni chiamate.

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

```
|| module <nome_del_modulo> // server
  <dichiarazione delle procedure entry>
  {
    <dichiarazione delle variabili locali>;
    <inizializzazione delle variabili locali>;
    entry op_1 (<parametri formali>)
      { <corpo della procedura op1>; }
    .....
    entry op_n (<parametri formali>)
      { <corpo della procedura opn >; }
    <dichiarazione di procedure locali>;
    <dichiarazione di processi locali>;
  }
```

Le procedure `entry` sono le procedure esportate dal modulo e che possono essere chiamate da un processo situato su un diverso nodo (*procedure remote*).

La chiamata della procedura remota `opi` da parte di un processo cliente avviene tramite la notazione

```
|| call <nome_del_modulo>.op_i(<parametri formali>)
```

La realizzazione della chiamata di procedura remota appartenente a un modulo comporta che essa sia eseguita da un processo servitore situato sullo stesso nodo. Per ogni richiesta viene creato un processo servitore il cui compito è quello di eseguire la procedura corrispondente.

La presenza di più processi servitori, in esecuzione contemporaneamente all'interno di un modulo per soddisfare le richieste di determinati servizi, richiede che si provveda alla loro sincronizzazione nell'accesso alle risorse del modulo.

Esempio: servizio di sveglia

Si vuole realizzare tramite RPC un allarme che ha il compito di risvegliare un insieme di processi clienti che richiedono questo servizio dopo un tempo da loro prefissato.

CLIENT:

```
|| ...
  call allarme.richiesta_sveglia(60, 1);
  ...
```

ogni processo cliente richiede il servizio di sveglia chiamando la procedura *richiesta_sveglia* e passando, come parametri, l'intervallo di tempo per il quale vuole attendere (timeout) e il suo identificatore (id).

SERVER:

```

module allarme
{
    int time;
    semaphore mutex=1;
    semaphore priv[N]=0; // semafori privati per la sospensione dei proc.
    coda_richieste coda; // struttura contenente le richieste 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>;
        V(mutex);
        P(priv[id]); // attesa della sveglia.
    }

    process clock { // 'demone'
        int tempo_di_sveglia
        <avvia il clock>;
        while (true) {
            // <attende per l'interruzione, quindi riavvia il clock>;
            time++;
            P(mutex);
            tempo_di_sveglia= // <più piccolo valore di sveglia in
                coda>;
            while (time >= tempo_di_sveglia) {
                // <rimozione di tempo_di_sveglia e id corrisp.
                dalla coda>;
                V(priv[id]); // risveglio del processo id
            }
            V(mutex);
        }
    }
} // fine modulo

```

Il processo servitore esegue la procedura *richiesta_sveglia* e provvede ad accordare la richiesta del cliente rispettando l'ordine di risveglio.

Il processo servitore si sospende quindi su un semaforo privato in attesa di essere risvegliato dal processo *clock*. Poiché servitore e cliente sono sincronizzati per tutta la durata del servizio anche il processo cliente risulta bloccato.

Il processo *clock* tiene aggiornato il tempo corrente e, in mutua esclusione, va a esaminare la coda delle richieste per verificare se un intervallo di attesa si è completato; in caso affermativo esegua una *V* sul relativo semaforo privato, risvegliando il processo servitore e quindi il corrispondente processo cliente.

8.2 Rendez-vous

Le modalità di richiesta di un servizio è la stessa come nel caso della RPC, tuttavia, l'operazione corrispondente è eseguita da un processo servitore esistente.

Il processo servitore fa uso di un'apposita istruzione per attendere le richieste di servizio e quindi per eseguire il servizio stesso. Nel seguito si farà riferimento alla soluzione adottata dal linguaggio ADA che prevede che l'istruzione d'ingresso sia del tipo:

```
|| accept <servizio> (in <param_ingresso>, out <param_uscita>); -> {S1,...,Sn};
```

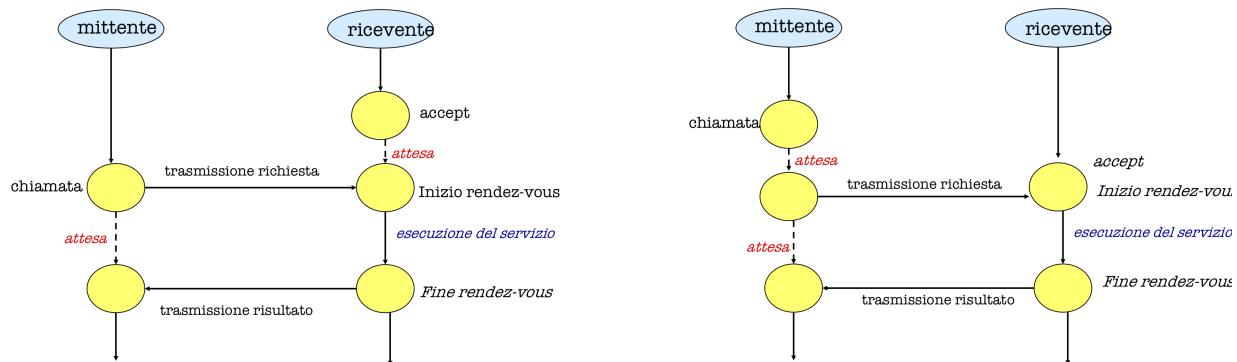
dove S_1, \dots, S_n rappresentano le azioni eseguite dal processo servitore come risposta a una richiesta di esecuzione del processo cliente; **in** sta a indicare i parametri d'ingresso e **out** quelli di uscita.

Se non sono presenti richieste di servizio, l'esecuzione di **accept** provoca la **sospensione** del processo servitore. All'arrivo della richiesta, il processo servitore esegue S_1, \dots, S_n utilizzando i parametri d'ingresso; durante l'esecuzione i due processi rimangono sincronizzati. Al termine, i risultati vengono forniti al processo cliente sotto forma di parametri di uscita e i due processi proseguono indipendentemente la loro esecuzione.

Lo stesso servizio può essere richiesto da più processi clienti prima che il processo servitore esegua il corrispondente **accept**. In tal caso vengono inserite in una coda associata alla particolare operazione richiesta gestita, generalmente, in modo **FIFO**.

A una stessa operazione possono essere associati più **accept**; pertanto, a una richiesta, possono corrispondere diverse azioni in funzione del punto di elaborazione del processo servitore. Ciò mette in evidenza una netta distinzione con la RPC, che si basa sull'identificazione di una particolare operazione con una determinata procedura.

Se il ricevente esegue **accept** prima della chiamata da parte del mittente, esso viene sospeso. Al momento dell'esecuzione della chiamata i parametri d'ingresso sono copiati dal mittente al ricevente, il mittente viene sospeso e il ricevente esegue il codice associato ad **accept**. Quando tale compito è completato, i parametri di uscita vengono inviati dal ricevente al mittente che riprende l'esecuzione dal punto in cui ha fatto la chiamata.



Se la richiesta è effettuata prima che sia stato eseguito **accept**, il chiamante è sospeso come nel caso precedente. Quando il ricevente esegue **accept** e vi sono più richieste in attesa, viene scelta quella che è in attesa da più tempo; viene quindi eseguito il codice associato all'istruzione **accept** e tutto procede come nel caso precedente.

Selezione richieste

Nel modello rendez-vous, il server può selezionare le richieste da servire in base al suo stato interno (ad esempio lo stato delle risorse gestite), utilizzando i comandi con guardia:

```

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

Per permettere a un task di selezionare (o impedire) un insieme di comunicazioni sulla base del suo stato interno, ADA consente di premettere guardie logiche, precedute dalla parola chiave **when**, innanzi a ogni alternativa di una istruzione **select**:

```

select
  when cond_1 -> accept entry_1
  or when cond_2 -> accept entry_2
  ...
  
```

```

    or when cond_n -> accept entry_n
end select;

```

Esempio produttore/consumatore

```

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

```

La sincronizzazione tra processo chiamante (client) e processo chiamato (server) è limitata alle sole istruzioni comprese nel blocco di accept (cioè quelle comprese in → {...})

```

process produttore-i {
    messaggio dati;
    for(; ;)
        <produc i dati>;
        call buffer.inserisci(dati);
    }
}

process consumatore-j {
    messaggio dati;
    for(; ;)
        call buffer.preleva(dati);
        <consuma dati>;
    }
}

```

Selezione delle richieste in base ai parametri di ingresso

La decisione se servire o no una richiesta può dipendere, oltre che dallo stato della risorsa, anche dai parametri della richiesta stessa (esempio: politiche di accettazione delle richieste basate su priorità). In questo caso, infatti, la guardia logica che condiziona l'esecuzione dell'azione richiesta deve essere espressa anche in termini dei parametri di ingresso.

E' pertanto necessaria una doppia interazione tra processo cliente e processo servitore; la prima per trasmettere i parametri della richiesta e la seconda per richiedere il servizio.

Vettore di operazioni di servizio

Nell'ipotesi di un numero limitato di differenti richieste si può ottenere una semplice soluzione al problema associando ad ogni richiesta una differente operazione di servizio, il vettore di operazioni di servizio (linguaggio Ada).

Esempio: sveglia

Si consideri ad esempio il caso del processo (server) allarme il cui compito sia di inviare una segnalazione di sveglia ad un insieme di processi che richiedono questo servizio dopo un tempo da essi stabilito.

Il processo allarme interagisce periodicamente con un processo clock per tenere traccia del tempo.

Il server può ricevere tre tipi di richieste:

- **tick**: aggiornamento del tempo (da clock a allarme)
- **richiesta_di_sveglia(T)**: impostazione della sveglia per il cliente mittente (da cliente generico ad allarme)
- **svegliami[T]** (da cliente generico ad allarme): invio del segnale di allarme al tempo specificato

L'ordine con cui il processo allarme risponde alle richieste del tipo svegliami dipende solo dal parametro T (intervallo di attesa) trasferito con la richiesta.

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

Possiamo associare ad ogni richiesta di sveglia, un diverso elemento di un vettore di operazione di servizio:

```
typedef struct {
    int risveglia;
    int intervallo;
} dati_di_risveglia;

// vettore delle richieste di servizio:
dati_di_risveglia tempo_di_sveglia[N];

process allarme {           //server
    entry tick;
    entry richiesta di sveglia(in int intervallo);
    entry svegliami[first..last];   //vettore di entry int tempo;
    typedef struct {
        int risveglia;
        int intervallo;
    } dati_di_risveglia;
    dati_di_risveglia tempo_di_sveglia[N];

    do {
        [] accept tick; -> {tempo++;} // dal processo clock

        [] accept richiesta di sveglia (in int intervallo)
            -> {<inserimento tempo + intervallo ed intervallo in tempo
                 di sveglia in modo da mantenere tale vettore ordinato
                 secondo valori non decrescenti di risveglia>}

        [] (tempo==tempo_di_sveglia[1].risveglia);
        accept svegliami [tempo_di_sveglia[1].intervallo];
        -> { <riordinamento del vettore tempo_di_sveglia>; }
    }
}
```

Capitolo 9

Linguaggio ADA

ADA fu sviluppato per conto del Dipartimento della Difesa degli Stati Uniti come standard per le applicazioni di tipo militare, comprese le applicazioni in tempo reale. Per il periodo in cui fu sviluppato, il linguaggio presentava caratteristiche fortemente innovative sia nella parte sequenziale che in quella concorrente.

Per quanto riguarda l'aspetto della gestione della concorrenza, ADA costituisce un esempio di linguaggio che adotta, come strumento di iterazione tra processi, quello del rendez-vous introdotto in precedenza. In una più recente versione di ADA sono stati introdotti altri strumenti quali **protected type**, simili al monitor, e l'istruzione **requeue** per dare al programmatore maggior controllo sulla sincronizzazione e sullo scheduling.

9.1 Tipi di dato

Ada è un linguaggio **fortemente e staticamente tipato**. La maggior parte degli errori possono essere rilevati in fase di compilazione.

Il programmatore può definire nuovi tipi per esprimere pienamente le caratteristiche del dominio applicativo:

Due tipi diversi non possono essere confrontati (errori rilevati a compile time).

9.1.1 Tipi scalari

```
package Apples_And_Oranges is
    type Number_Of_Apples is range 1 .. 20;           --integer
    type Number_Of_Oranges is range 1 .. 40;          --integer
    type Mass is digits 4 range 0.0 .. 4000.0;        --real
    type Colour is (Red, Green, Blue);                --enumeration
end Apples_And_Oranges;
```

Number_of_Apples, Number_of_Oranges sono particolari Integer che riflettono i vincoli del problema.

I tipi scalari sono caratterizzati anche da attributi:

- T'First, T'Last : costanti che individuano estremi inferiore e superiore di un intervallo.
- T'Range = T'First .. T'Last (intervallo di valori permessi)
- T'Pred(X), T'Succ (X) : funzioni che ritornano il valore precedente o successivo nel dominio
- T'Image (X) : la rappresentazione di X come stringa (per la stampa).

9.1.2 Array

```
|| A: array (4 .. 8) of Integer;
```

Gli array hanno attributi: Range, Length. Ad esempio:

- A'Range è l'intervallo dell'indice → A'Range = 4 .. 8;
- A'Length è la dimensione dell'array (5)

9.1.3 Record

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure Record1 istype DATE is
    record
        Month : INTEGER range 1..12;
        Day   : INTEGER range 1..31;
        Year  : INTEGER range 1776..2018;
    end record;

    Independence_Day : DATE;

begin
    Independence_Day.Month := 7;
    Independence_Day.Day   := 4;
    Independence_Day.Year  := 1776;
end Record1;
```

9.1.4 Puntatori: access

I puntatori si definiscono tramite il costrutto access.

Non è possibile combinare in espressioni puntatori a tipi diversi (tipaggio forte):

```
procedure Access1 is
type POINT_SOMEWHERE is access INTEGER;
Index: POINT_SOMEWHERE;

begin
    Index := new INTEGER;
    Index.all := 13;
    Put("The value is");
    Put(Index.all, 6); New_Line;
end Access1;
```

9.2 Istruzioni di controllo

9.2.1 Alternativa: if

```
if Index < 15
    then Put_Line(" and is less than 15.");
    else Put_Line(" and is 15 or greater.");
end if;
```

9.2.2 Ripetizione : *loop*

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure LoopDemo is
    Index, Count : INTEGER;

begin
    Index := 1;
    loop
        Put("Index =");
        Put(Index, 5); New_Line;
        Index := Index + 1;
        exit when Index = 5;
    end loop;
end LoopDemo;
```

9.2.3 Ripetizione: *for*

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure ForDemo is
    Index: INTEGER;

begin
    Index := 1;
    for Index in 1..4
        Put("Doubled index =");
        Put(2 * Index, 5); New_Line;
    end loop;
end ForDemo;
```

9.3 Task

I processi, chiamati *task*, possono essere creati, attivati e terminati dinamicamente. Due sono i meccanismi di creazione e attivazione dei task. Il primo segue lo **scope rule** del linguaggio: tutti i task dichiarati localmente a un blocco vengono creati quando sono elaborate le corrispondenti dichiarazioni e, quindi quando il controllo raggiunge il blocco in questione. I task così creati vengono quindi inizializzati (attivati) e la loro esecuzione avviene in parallelo a quella del blocco.

Oltre alla dichiarazione di task esiste la possibilità di dichiarare **task type**; successivamente possono essere dichiarate varie istanze che condividono lo stesso corpo.

Anche la terminazione dei task segue le scope rule del linguaggio: il completamento del blocco in cui i task sono dichiarati è condizionato dalla terminazione di tutti i task.

In ADA la comunicazione tra task è di tipo asimmetrico e sincrono a rendez-vous. Il rendez-vous tra due task viene stabilito quando entrambi esprimono la volontà di eseguire una stessa operazione. Tale operazione presenta l'aspetto di una procedura e prende il nome di **entry**. Una entry, definita in un task P e resa visibile all'esterno di P, può essere chiamata da un altro task Q secondo il normale meccanismo di chiamata a una procedura (entry call).

Lo schema di un task prevede una parte di specifica che definisce le operazioni entry e una parte body che contiene la realizzazione di tali operazioni.

Per la parte di specifica si ha:

```
task <nome_task> is
    < dichiarazione delle entry >
end;
```

Le entry definiscono le operazioni servite dai task e hanno la forma:

```
|| entry identifier (<parametri formali>);
```

Per la parte body si ha:

```
|| task body <nome_task> is
    < dichiarazione locali >
begin
    < istruzioni del task >
end <nome_task>;
```

Esempio con 3 processi

```
with Ada.Text_IO, Ada.Integer_Text_IO; --importaz.package
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure Task1 is -- procedure "main"
    task First_Task; --def primo processo
    task body First_Task is
        begin --corpo
            for Index in 1..4 loop
                Put("This is in First_Task, pass number ");
                Put(Index, 3);
                New_Line;
            end loop;
        end First_Task;

    task Second_Task; -- def. secondo processo
    task body Second_Task is
        begin -- corpo
            for Index in 1..7 loop
                Put("This is in Second_Task, pass number");
                Put(Index, 3);
                New_Line;
            end loop;
        end Second_Task;

        begin -- def main
        Put_Line("Questo è il main task..");
        end Task1;
```

Una entry è un'operazione che un task rende disponibile agli altri task:

```
|| task S is -- dichiarazione
    entry E (<lista_parametri>);
end S;
task body S is --definizione
begin
    <definizione di S e delle sue entry>
end S;
```

I parametri possono essere di tipo IN, OUT.

9.4 Rendez-vous

Una entry, definita in un task server S e resa visibile all'esterno di S, può essere chiamata da un task cliente C (contenuto nello scope di S) secondo il normale meccanismo di chiamata a procedura call:

```
|| call S.entryname (<parametri effettivi>);
```

La comunicazione tra C e S avviene quando S esprime la volontà di eseguire la entryname invocata da C, mediante l'istruzione accept:

```

accept entryname (in <par_in>, out <par_out>);
do I1; I2; .. In;
end entryname;
```

Dove `I1; I2; .. In`; rappresentano le azioni eseguite da S quando la comunicazione ha luogo; durante la loro esecuzione il processo C rimane sincronizzato col processo S.

L'esecuzione di `accept entryname` da parte di S sospende il task fino a quando non vi è una chiamata di `entryname`. In quel momento i parametri effettivi sono copiati nei parametri di ingresso e viene eseguita la lista di istruzioni; al loro completamento, i risultati sono copiati nei parametri di uscita. A questo punto cessa la sincronizzazione tra S e C ed entrambi continuano la loro esecuzione.

Una stessa entry può essere invocata da più processi prima che il task che la definisce esegua la corrispondente istruzione di `accept`. In tal caso le chiamate vengono inserite in una coda associata a tale entry gestita in modo FIFO.

Esempio interazione

```

with Ada.Text_IO;
use Ada.Text_IO;

procedure HotDog is
    task Gourmet is --dichiarazione
        entry Make_A_Hot_Dog;
    end Gourmet;

    task body Gourmet is --definizione
begin
    for Index in 1..4 loop
        accept Make_A_Hot_Dog do --def.entry
            delay 0.8; --cfr. sleep
            Put("Metto hot dog nel pane..");
            Put_Line("Aggiungo senape");
            end Make_A_Hot_Dog;
    end loop;
end Gourmet; 22

begin --task main
    for Index in 1..4 loop
        Gourmet.Make_A_Hot_Dog; --entry call
        delay 0.1;
        Put_Line("Mangio l'hot dog");
        New_Line;
    end loop;
end HotDog;
```

9.4.1 Select

Ad una stessa entry possono essere associate più `accept`. Pertanto a esse possono, in generale, corrispondere azioni diverse in funzione del punto di elaborazione del task che la definisce. L'istruzione:

```

select
    accept entry_1 do ... end;
    or accept entry_2 do ... end;
    ...
    or accept entry_n do ... end;
end select;
```

rappresenta la forma più semplice di un comando con guardia in ADA e viene usata per indicare che un task può comunicare con un qualsiasi altro task che abbia chiamato una delle entry elencata dell'istruzione `accept`.

- Se nessuna di essa viene chiamata, il task si pone in attesa;
- Se una sola è chiamata, essa viene immediatamente accettata;

- Se più entry sono già state chiamate, ne viene scelta una in modo non deterministico.

Per permettere a un task di selezionare (o impedire) un insieme di comunicazioni sulla base del suo stato interno, ADA consente di premettere guardie logiche, precedute dalla parola chiave `when`, innanzi a ogni alternativa di una istruzione `select`. Si ha cioè:

```
||| select
    when cond_1 --> accept entry_1 ...
    or when cond_2 --> accept entry_2 ...
    ...
    or when cond_n --> accept entry_n ...
||| end select;
```

L'esecuzione dell'istruzione `select` avviene in questo modo:

- vengono valutate le condizioni. Le alternative per le quali la condizione è vera vengono dette aperte;
- viene scelta arbitrariamente una delle alternative aperte per cui è immediatamente possibile il corrispondente rendez-vous; se nessun rendez-vous è possibile, si attende che vi sia tale possibilità.

Ovviamente il caso in cui tutte le guardie siano di valore falso rappresenta un errore di programmazione che genera un'eccezione (program error).

Una comunicazione tra processi in ADA, una volta stabilita attraverso i meccanismi visti precedentemente, non può venire sospesa. Ciò, se da un lato consente di mantenere per il task il normale significato di processo sequenziale, influenza da un altro la potenza espressiva del linguaggio relativamente a particolari categorie di problemi quali, per esempio, algoritmi di assegnazione delle risorse.

In particolare, con i soli meccanismi di comunicazione e sincronizzazione fin qui esaminati, risultano di difficile definizioni le politiche dipendenti dal tipo della richiesta e da parametri associati al task.

Per consentire l'espressione di tali politiche, ADA utilizza il concetto di famiglie di entry:

```
||| entry <entryname> (first..last) (in..out...);
```

Tramite questo costrutto è possibile suddividere una coda di attesa in un insieme di code selezionabili individualmente.

Il costrutto `select` consente inoltre di eseguire delle chiamate condizionali di una entry (conditional entry call). La sequenza di istruzioni:

```
||| select
    < chiamata di una entry >
    else < comandi >
||| end select;
```

è equivalente alla chiamata di una entry, se il processo in cui è stata definita è sospeso su una `accept` per la `entry` stessa; nel caso opposto vengono eseguiti i comandi che seguono la clausola `else`.

Un'altra possibilità di uso della `select` è quello di esprimere chiamate cui deve essere assicurato un servizio entro un tempo prefissato. In caso contrario la chiamata viene annullata e il task chiamante prosegue l'esecuzione:

```
||| select
    < chiamata di una entry >;
or
    delay < intervallo di tempo >;
||| end select;
```

La sequenza di istruzioni specifica che il task chiamante attende, per l'esecuzione del comando, al più un tempo pari a un intervallo fissato.

Esempio select

```
with ada.text_io; -- include libreria text_io
with ada.integer_text_io; -- include libreria integer_text_io
procedure task_demo is
    task type intro_task is -- dichiarazione tipo di task
        entry start; -- dichiarazioni entry
        entry turn_left;
        entry turn_right;
        entry stop;
    end intro_task;

task body intro_task is
begin
    accept start; --def. entry
    loop
        select
            accept turn_left; --def. entry
                Put_line ("turning left");
            or
            accept turn_right; --def. entry
                Put_line ("turning right");
            or
            accept stop; --def. entry
                Put_line ("stop received");
                exit; -- exit the loop
            else
                Put_line ("moving straight");
            end select;
            delay 0.5;
        end loop;
    end intro_task;

task_1 : intro_task; -- creazione task
begin
    task_1.start;
    delay 2.0;
    task_1.turn_left;
    delay 2.0;
    task_1.turn_right;
    delay 1.0;
    task_1.turn_right;
    delay 2.0;
    task_1.stop;
end task_demo;
```

Select con guardie logiche

```
|| select
||   when condizione1 => accept E1(...)
||     do ...
||     end E1;
||   or
||   when condizione2 => accept E2(...)
||     do ...
||     end E2;
|| end select;
```

Stessa semanticà del comando con guardia alternativo.

Comandi con guarda ripetitivo

```
|| loop
||   select
||     when condizione1 => accept E1(...)
||       do ...
||       end E1;
||     or
||     when condizione2 => accept E2(...)
||       do ...
||       end E2;
||     or ...
||   end select;
|| end loop;
```

9.5 Esempio produttore / consumatore

```
|| loop
||   select
||     when pieno < Bufsize =>
||       accept Inserisci( v : IN Integer) do
||         ...
||       end Inserisci;
||     or
||     when pieno > 0 =>
||       accept Estrai( v : out Integer) do
||         ...
||       end Estrai;
||   end select;
|| end loop;
```

9.6 Corrispondenza tra monitor e processi servitori in ADA

modello a memoria comune	corrisponde	modello a scambio di messaggi (ADA)
risorsa condivisa: istanza di un monitor	→	risorsa condivisa: struttura dati locale a un task server
identificatore di funzione di accesso al monitor	→	nome di entry offerto dal server
tipo dei parametri della funzione	→	tipo dei parametri in della entry
tipo del valore restituito dalla funzione	→	tipo parametri out della entry
per ogni funzione del monitor	→	un ramo (comando con guardia) dell'istruzione loop-select che costituisce il corpo del server

modello a memoria comune	corrisponde	modello a scambio di messaggi (ADA)
condizione di sincronizzazione di una funzione	→	espressione logica (when) nel ramo corrispondente alla funzione
chiamata di funzione	→	chiamata (da parte del client) della entry corrispondente nel server
esecuzione in mutua esclusione fra le chiamate alle funzioni del monitor	→	scelta di uno dei rami con guardia valida del loop-select del server
corpo della funzione	→	istruzione del ramo corrispondente alla funzione

Capitolo 10

Algoritmi di Sincronizzazione Distribuiti

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

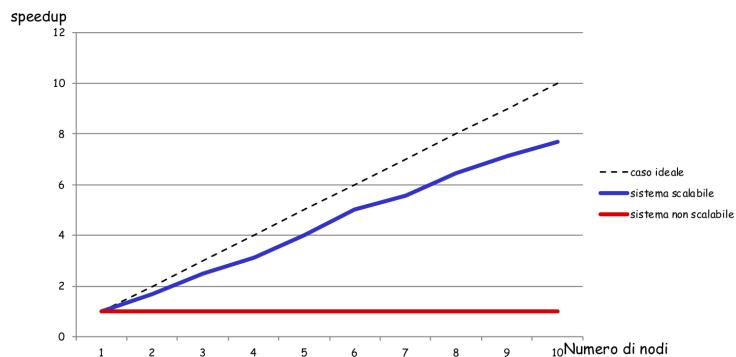
Caratteristiche dei sistemi distribuiti:

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

10.1 Proprietà desiderate

Nelle applicazioni distribuite è importante poter contare su alcune proprietà:

1. **scalabilità**: nell'applicazione distribuita le prestazioni dovrebbero aumentare al crescere del numero di nodi utilizzati.



Lo **speedup** per n nodi è il rapporto tra il tempo di esecuzione dell'applicazione ottenuto con 1 solo nodo e quello ottenuto con n nodi:

$$\text{Speedup}(n) = \text{Tempo}(1) / \text{Tempo}(n)$$

2. **tolleranza ai guasti**: l'applicazione è in grado di funzionare anche in presenza di guasti (es. crash di un nodo).

Il sistema distribuito riesce a erogare i propri servizi anche in presenza di guasti. I guasti possono presentarsi in diversi tipi:

- Transienti

- Intermittenti
- Persistenti

Un sistema tollerante ai guasti deve «nascondere» (mascherare) i guasti agli altri processi.

Questo obiettivo può essere raggiunto con tecniche di **ridondanza**:

Vengono mantenute più istanze degli stessi componenti, in modo da poter rimpiazzare l'elemento guasto con un elemento equivalente.

10.2 Algoritmi di Sincronizzazione

Come nel modello a memoria comune, anche nel modello a scambio di messaggi è importante poter disporre di algoritmi di sincronizzazione tra i processi concorrenti, che consentano di risolvere alcune problematiche comuni, coordinando opportunamente i vari processi.

Ad esempio:

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

E' desiderabile che gli algoritmi distribuiti godano delle proprietà di scalabilità e di tolleranza ai guasti.

10.3 Algoritmi per la gestione del tempo

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

Ciò comporta possibili problemi in applicazioni distribuite (Ad esempio compilazione con make, utility basata sulla data dei file).

In applicazioni distribuite può essere necessario:

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

Orologi logici (Lamport, 1979)

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

Relazione di precedenza tra eventi (Happened-Before, \rightarrow):

1. se a e b sono eventi in uno stesso processo ed a si verifica prima di b: **$a \rightarrow b$**
2. se a è l'evento di invio di un messaggio e b è l'evento di ricezione dello stesso messaggio: **$a \rightarrow b$**
3. se **$a \rightarrow b$** e **$b \rightarrow c$** allora **$a \rightarrow c$**

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

1. **$a \rightarrow b$** , cioè a avviene prima di b
2. **$b \rightarrow a$** , cioè b avviene prima di a
3. a e b non sono legati dalla relazione HB → a e b sono concorrenti

Orologi logici

Assumiamo che ad ogni evento e venga associato un timestamp $C(e)$.

Si vuole definire un modo per misurare il concetto di tempo tale per cui ad ogni evento a possiamo associare un timestamp $C(a)$ sul quale tutti i processi siano d'accordo.

Il tempo $C(a)$ deve soddisfare la seguente proprietà:

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

Quindi:

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

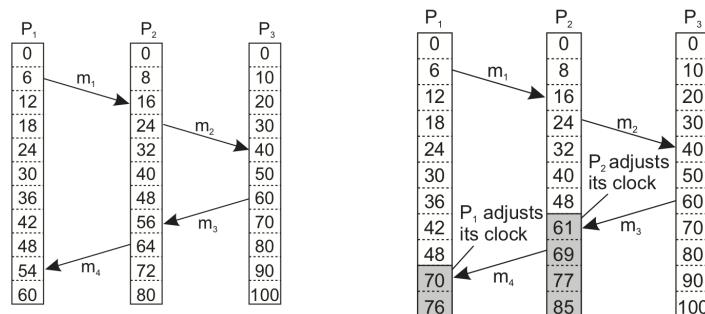
10.3.1 Algoritmo di Lamport

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

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

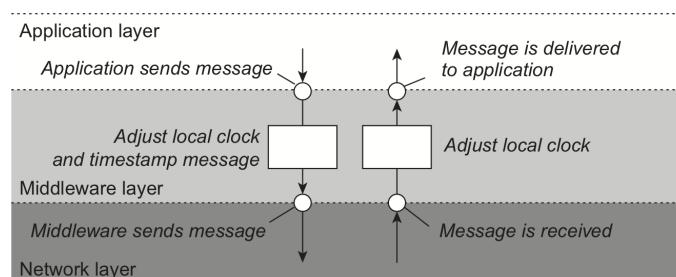
Esempio

Consideriamo 3 processi in esecuzione su 3 nodi, ognuno con il proprio clock (frequenze diverse):



10.3.2 Lamport: aggiornamenti degli orologi

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



10.4 Mutua esclusione distribuita

L'obiettivo è quello di garantire che due o più processi non possano eseguire contemporaneamente certe attività (ad esempio accesso a risorse condivise: file, stampanti, ecc.).

Le soluzioni da adottare sono:

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

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

In generale, possiamo suddividere le soluzioni in due categorie:

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

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

10.4.1 Soluzione centralizzata

L'algoritmo si basa su permessi (permission-based): la risorsa viene gestita da un processo coordinatore al quale ogni processo che vuole eseguire la sua sezione critica si rivolge per ottenere il permesso.

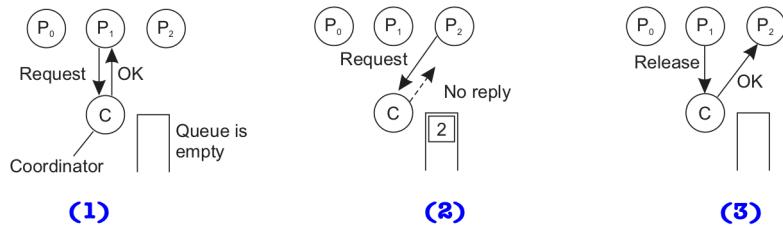
Per eseguire la propria sezione critica ogni processo Pi:

1. **Richiesta**: Pi invia una richiesta di autorizzazione al coordinatore
2. **Autorizzazione**: Pi riceve il permesso
3. <esegue la sezione critica>
4. **Rilascio**: Pi comunica al coordinatore il termine della sezione critica

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

Esempio

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



I vantaggi della soluzione centralizzata sono nell'algoritmo che risulta essere **equo** (non c'è starvation). Prevede solo 3 messaggi (richiesta-autorizzazione-rilascio) per ogni sezione critica.

Di contro la soluzione è **poco scalabile** perché al crescere del numero dei processi il coordinatore può diventare un collo di bottiglia.

Un altro svantaggio è la **tolleranza ai guasti**: il coordinatore può essere soggetto a guasto e, in questo caso, l'intero sistema si blocca (single point of failure). Inoltre, se un processo P che ha fatto una richiesta non ottiene una risposta, P non può distinguere le due possibili cause ovvero autorizzazione non concessa da coordinatore guasto.

10.4.2 Algoritmo Ricart-Agrawala

Algoritmo decentralizzato basato su **permessi** (permission-based). Il sistema è costituito da un **insieme di processi in competizione**.

Ad ogni processo sono associate 2 attività concorrenti (thread):

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

Come requisito si assume che vi sia un **temporizzatore globale** per tutti i nodi (orologio logico): ogni messaggio è corredata da un timestamp che ne indica l'istante di invio.

Struttura del main:

Quando un processo vuole entrare nella sezione critica:

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

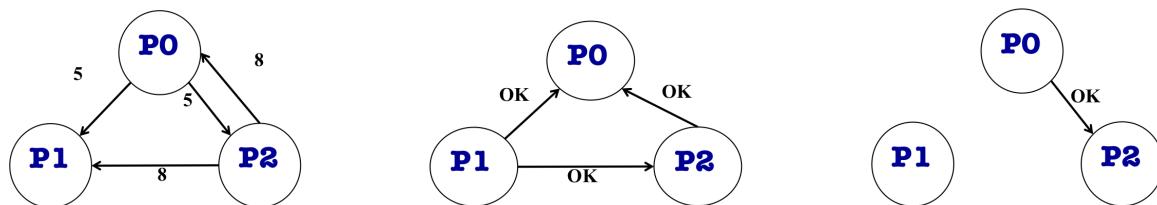
Struttura del receiver:

Quando viene ricevuta una richiesta, il receiver può trovarsi in uno di tre possibili stati:

1. **RELEASED**: Il processo non è interessato ad entrare nella sezione critica: risponde OK.
2. **WANTED**: Il processo è in procinto di entrare nella sezione critica (attende autorizzazione): confronta il timestamp Tr della richiesta ricevuta con quello della richiesta inviata (Ts):
 - se $Tr > Ts$ risponde OK
 - altrimenti non risponde e mette la richiesta ricevuta in coda
3. **HELD**: Il processo sta eseguendo la sezione critica: la richiesta viene messa in coda.

Esempio Consideriamo 3 processi: P0, P1, P2.

1. Se P0 e P2 vogliono eseguire una sezione critica, P0 invia a P1 e P2 una Richiesta (timestamp=5); P2 invia a P0 e P1 una richiesta (timestamp=8)
2. P0 è in stato WANTED: non risponde a P2 (perché $5 < 8$) e mette in coda la sua richiesta.
P1 è in stato RELEASED: risponde ad entrambi OK.
P2 è in stato WANTED: risponde OK a P0 (perché $5 < 8$).
→ P0 esegue la sezione critica.
3. Terminata la sezione critica, P0 invia OK a P2 (la cui richiesta era stata accodata).
→ P2 esegue la sezione critica.



I vantaggi che ne derivano dall'algoritmo Ricart-Agrawala sono in termini di distribuzione quindi scalabilità.

Di contro si hanno i seguenti svantaggi:

Maggior costo di comunicazione per il singolo partecipante: $2*(N-1)$ messaggi per ogni sezione critica.

Non c'è più un single point of failure, ma N points of failure: se uno dei nodi va in crash, non risponderà alle richieste degli altri.

Vi sarà anche l'impossibilità di rilevare il guasto: l'attesa è dovuta a guasto o a sezioni critiche in esecuzione?

Guasti nell'algoritmo Ricart-Agrawala

Si può modificare il protocollo prevedendo che ad ogni richiesta venga sempre fornita una risposta:

- OK
- Attendi (autorizzazione negata): in questo caso il richiedente si mette in attesa di un OK.

Il richiedente può impostare timeout per rilevare guasti del destinatario.

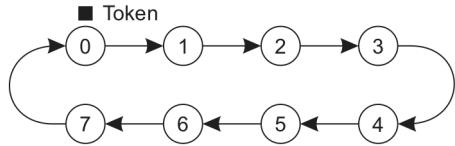
N points of failure: ogni processo ha un ruolo equivalente agli altri; se è guasto, può essere semplicemente escluso dal gruppo.

10.4.3 Algoritmo token ring

Algoritmo distribuito basato su token. Il sistema è costituito da un insieme di processi in competizione, collegati tra di loro secondo una topologia logica ad anello.

Ogni processo conosce i suoi vicini nell'anello.

Un messaggio (detto token), circola attraverso l'anello, nel verso relativo all'ordine dei processi nella topologia.



Il token rappresenta il permesso unico di eseguire sezioni critiche: chi detiene il token può eseguire la sua sezione critica.

Il token viene inizialmente inviato da P0 a P1:

- P1 può essere nello stato **WANTED**: in questo caso trattiene il token ed esegue la sezione critica; al termine passa il token al processo successivo P2.
- P1 può essere nello stato **RELEASED**: in questo caso passa il token al processo successivo P2.

P2 si comporta allo stesso modo di P1, come tutti i processi successivi nell'anello. In questo modo il token circola nell'anello nel verso stabilito dall'ordine dei processi: $P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow \dots \rightarrow P_N \rightarrow P_0 \dots$

I vantaggi sono in termini di distribuzione quindi **scalabilità**.

Di contro si hanno i seguenti svantaggi:

Il numero messaggi per ogni sezione critica non è limitato: [1..inf].

Non c'è più un single point of failure, ma **N points of failure**: se uno dei nodi va in crash, interrompe la catena. Per fare fronte a questo problema, si può modificare il protocollo prevedendo che ad ogni invio del token (da P_i a P_{i+1}), venga sempre restituita una risposta; se la risposta non arriva (allo scadere di un timeout), P_{i+1} viene escluso dall'anello ed il token viene inviato a P_{i+2} .

Possibilità di perdere il token: ad esempio se va in crash il processo che lo detiene, il token va perduto. Difficoltà di rilevazione del problema.

10.4.4 Algoritmi di mutua esclusione a confronto

Algoritmo	num messaggi per sezione critica	Ritardo in entrata nella sez. critica	Problemi
Centralizzato	3	2	scarsa scalabilità crash coordinatore
Ricart-Agrawala	$2*(n-1)$	$2*(n-1)$	crash qualunque processo
Token Ring	1..∞	0..n-1	crash qualunque processo perdita token

10.5 Algoritmi di elezione

In alcuni algoritmi è previsto che un processo rivesta il ruolo speciale (es. il coordinatore nell'algoritmo di mutua esclusione centralizzato).

La **designazione del coordinatore può essere**:

- **statica**: il coordinatore viene deciso in modo arbitrario dal programmatore/amministratore prima dell'esecuzione.
- **dinamica**: il coordinatore viene designato a runtime dai processi del sistema con un algoritmo di elezione.

Esempio: Se il coordinatore di un gruppo di processi subisce un crash, per ripristinare l'operatività del sistema, è necessario individuare a runtime un altro processo del gruppo a cui attribuire il ruolo di nuovo coordinatore.

L'assunzione di base vede che ogni processo è identificato da un unico ID numerico; ogni processo conosce gli ID di tutti gli altri (ma non il loro stato: se è attivo o terminato).

Obiettivo: Viene designato vincitore il processo attivo che ha l'ID più alto.

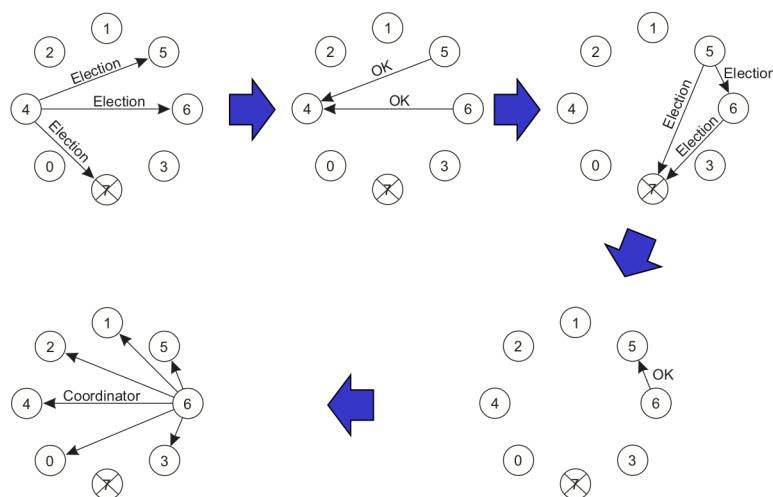
10.5.1 Algoritmo Bully

Consideriamo un sistema composto da N processi $\{P_0, \dots, P_{N-1}\}$; sia $ID(P_i) = i$.

Quando un processo P_k rileva che il **coordinatore non è più attivo**, organizza **un'elezione**:

1. P_k invia un messaggio "ELEZIONE" a tutti i processi con ID più alto: $P_{k+1}, P_{k+2}, \dots, P_{N-1}$.
2. Se nessun processo risponde, P_k vince l'elezione e diventa il nuovo coordinatore; comunica quindi a tutti gli altri processi il nuovo ruolo tramite il messaggio "COORDINATOR".
3. Se un processo $P_j (j > k)$ risponde, P_j prende il controllo e P_k rinuncia ed esce dall'elezione.

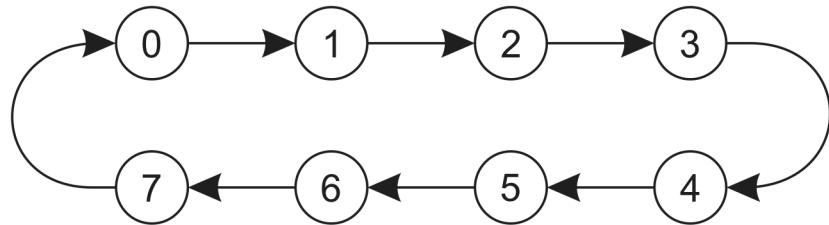
Ogni **processo attivo risponde** ad ogni messaggio di ELEZIONE ricevuto.



10.5.2 Algoritmo di elezione ad Anello

I processi del gruppo sono collegati tramite una topologia logica ad anello. La posizione (ID) che ogni processo occupa all'interno dell'anello rappresenta la sua priorità.

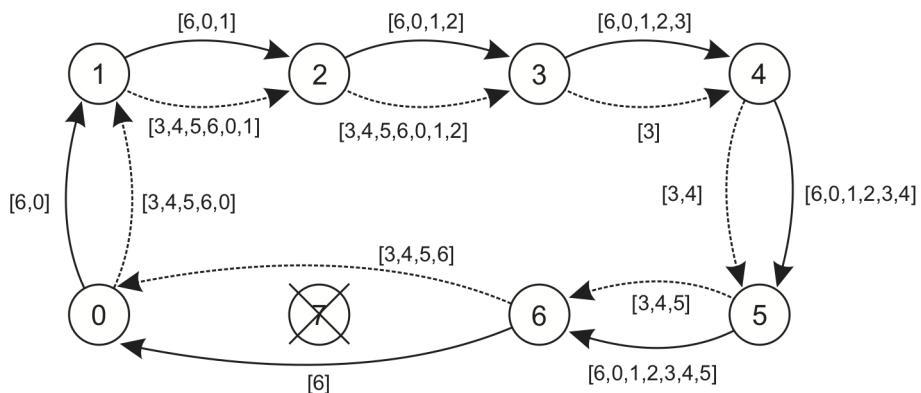
Il processo attivo con la massima priorità viene eletto coordinatore.



Quando un qualunque processo P_i si rende conto che il coordinatore non risponde più inizia un'elezione:

1. P_i invia un messaggio **ELEZIONE** contenente il suo ID al suo successore P_{i+1} . Se il successore P_{i+1} è in crash, il messaggio viene spedito al successore di P_{i+1} ecc.
2. Quando un processo P_j riceve un messaggio **ELEZIONE**:
 - se il messaggio non contiene l'ID di P_j , aggiunge il suo ID al messaggio e lo spedisce al successivo ecc..
 - se il messaggio contiene l'ID di P_j , significa che è stato compiuto un giro completo dell'anello: P_j designa come coordinatore il processo corrispondente all'ID più alto nel messaggio ricevuto e invia al successivo processo un messaggio **COORDINATOR** contenente l'ID del processo designato come nuovo coordinatore.
3. Quando un processo riceve un messaggio **COORDINATOR**, ne prende atto e inoltra lo stesso messaggio al successivo.

Esempio P_3 e P_6 avviano entrambi un'elezione: entrambe le elezioni designeranno P_6 come nuovo coordinatore. Ogni messaggio **COORDINATOR** dopo aver percorso l'anello verrà eliminato.



Capitolo 11

Azioni atomiche

Una azione atomica è uno strumento di alto livello per la strutturazione di programmi concorrenti e/o distribuiti tolleranti ai malfunzionamenti.

Principalmente vengono applicate in programmi concorrenti, tolleranti vari tipi di malfunzionamenti (sistemi operativi distribuiti, applicazioni transazionali ...)

E' un'astrazione realizzata con i meccanismi linguistici offerti dal linguaggio concorrente.



Operazione che porta un insieme di oggetti
 $O = o_1, o_2, \dots, o_n$
da uno stato consistente S_1 a uno stato consistente S_2 .

11.0.1 Consistenza dei dati

Consistenza di un oggetto astratto di tipo T:

- Ogni tipo T ha una sua **relazione invariante**, che lo caratterizza dal punto di vista semantico; la relazione riguarda i valori delle variabili componenti l'oggetto.
- Ogni oggetto può trovarsi in stati **consistenti** o **inconsistenti** a seconda che si sia verificata o meno la relazione **invariante del tipo**.
- Ogni operazione primitiva su dati di tipo T deve essere programmata in modo da lasciare l'oggetto su cui opera **in uno stato consistente** (cioè, in uno stato in cui l'invariante sia soddisfatta).

Sia $O = [o_1, o_2, \dots, o_n]$ l'insieme degli oggetti su cui operano più programmi concorrenti; si ha la necessità di mantenere:

- la consistenza di ogni singolo oggetto O_i appartenente all'insieme
- la consistenza dell'insieme di oggetti.

Consistenza di un insieme di oggetti:

viene data una relazione R (invariante) relativa all'insieme di oggetti: l'insieme è consistente se la relazione R è soddisfatta.

Esempio Applicazioni bancarie

- oggetti: conti correnti
- programmi: operazioni di lettura, modifica, ecc., che riguardano più oggetti

Operazione di trasferimento: dati gli oggetti O_1 e O_2 , spostare x da O_1 a O_2 .

$O = O_1, O_2$

→ La relazione invariante R è: valore $O_1 +$ valore $O_2 =$ costante

11.1 Realizzazione di Azioni Atomiche Proprietà Fondamentali

Per garantire la consistenza dei dati, l'azione atomica deve possedere due proprietà fondamentali:

1. **Serializzabilità** (o atomicità nei confronti della concorrenza)
2. **Tutto o niente** (o atomicità nei confronti di eventi anomali)

11.2 Serializzabilità

L'azione atomica porta l'insieme O da uno stato S1 consistente a uno stato S2 consistente.

Ogni azione atomica esegue in un contesto concorrente: ci possono essere altre azioni concorrenti, eventualmente in competizione per l'uso di oggetti comuni.

11.2.1 Proprietà di serializzabilità:

Durante l'esecuzione dell'operazione l'insieme O può passare attraverso stati inconsistenti, che tuttavia non devono essere visibili ad altre azioni concorrenti.

Esempio Riprendiamo l'esempio delle applicazioni bancarie.

Operazione di trasferimento: dati gli oggetti O1 e O2, l'operazione trasferisce x da O1 a O2.

`O=O1, O2`

→ La relazione invariante R è: `valore O1 + valore O2 = costante`

```
Trasferimento(O1,O2, x)
{
  ...
  O1 = O1 - x;
  O2 = O2 + x;
  ...
}
```

Durante l'esecuzione del programma, quando x è stata tolta da O1 ma non ancora sommata a O2, si ha uno stato complessivo inconsistente (anche se i due oggetti si trovano singolarmente in stato consistente).

Osservazioni: La mutua esclusione sui singoli oggetti garantisce l'atomicità delle operazioni su di essi, ma non dell'intera operazione su tutto l'insieme.

Pertanto è necessario che l'intera parte di programma che realizza l'operazione **sull'insieme di oggetti** possa essere considerata atomica, cioè non divisibile.

La proprietà di serializzabilità delle azioni atomiche è simile a quella di indivisibilità delle azioni primitive:

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

Soluzioni

1. **nuova astrazione** che racchiude tutti gli oggetti e tutte le operazioni possibili (monitor o processo servitore)

I limiti saranno:

- Le operazioni possono non essere note al momento della definizione dell'astrazione
- non è possibile operare direttamente sugli oggetti
- soluzione centralizzata

2. adottare un **protocollo** distribuito che superi i limiti evidenziati (v. Two phase lock protocol)

11.3 Protocollo per la serializzabilità: two phase lock protocol

Sia A un'azione atomica che opera su un insieme di oggetti $O = \{O_1, O_2, \dots, O_n\}$;

Siano Richiesta(O_i) e Rilascio(O_j) le operazioni per richiedere al gestore di ogni oggetto l'uso esclusivo dell'oggetto.

La serializzabilità viene garantita allocando dinamicamente singoli oggetti in modo dedicato alle azioni atomiche secondo un particolare protocollo : **two-phase lock protocol**.

- a) Ogni oggetto deve essere acquisito da un'azione atomica A in modo esclusivo prima di qualunque azione su di esso: Richiesta (O_i) è bloccante se l'oggetto O_i non è disponibile
- b) Nessun oggetto deve essere rilasciato prima che siano eseguite tutte le operazioni su di esso;
- c) Nessun oggetto può essere richiesto dopo che è stato effettuato un rilascio di un altro oggetto.

Esempio

```
A1 : { X= X+20;
        Y= Y+20;
    }
```

```
A2 : { X= X*20;
        Y= Y*20;
    }
```

$$O = \{X, Y\}$$

Sia $X = Y$ la relazione di consistenza:

- Ogni azione atomica preserva la consistenza e quindi anche una qualunque esecuzione sequenziale di A1 e A2
- Se A1 e A2 sono eseguite in parallelo possono nascere inconsistenze

Rispettando i due primi requisiti del protocollo (a e b) si può avere:

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

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

IPOTESI: dopo Rilascio(X) ma prima di Richiesta(Y) da parte di A1 , viene eseguita A2

X_i Y_i valori iniziali di X e Y

X_f Y_f valori finali di X e Y

$$X_i = Y_i \text{ per ipotesi (invariante)}$$

$$X_f = (X_i + 20) * 10$$

$$Y_f = (Y_i * 10) + 20$$

E quindi:

$$X_f \neq Y_f$$

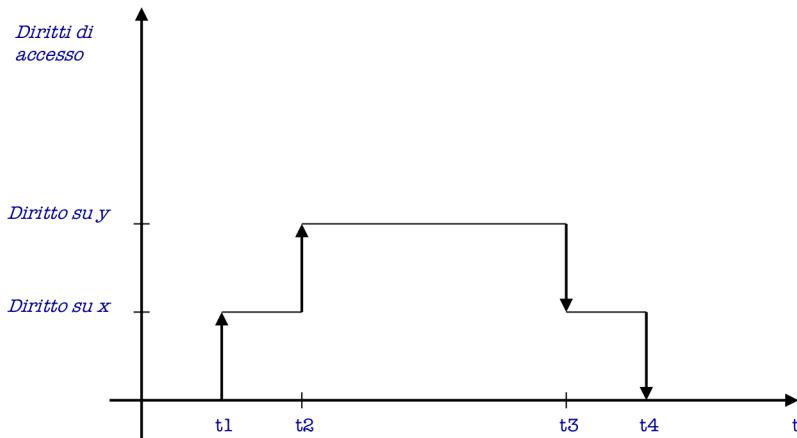
L'inconsistenza è dovuta al non rispetto del requisito c).

Il requisito c) assicura che, quando un'azione rilascia un oggetto contenente il valore finale, nessuno degli altri oggetti su cui l'azione opera sia libero e contenente il valore iniziale.

L'applicazione del two phase lock protocol (o protocollo di lock a due fasi) è sufficiente a garantire la proprietà di serializzabilità.

Nell'esecuzione di ogni azione atomica, definisce 2 fasi successive:

1. **I fase (fase crescente)** l'azione atomica acquisisce in modo esclusivo tutti gli oggetti ed opera su di essi
2. **II fase (fase calante)** inizia non appena viene eseguito il primo rilascio e durante essa non possono essere acquisiti ulteriori oggetti



```

A1 : { Richiesta(X) ; X= X+20; Richiesta(Y) ; Rilascio(X) ; Y= Y+20; Rilascio(Y) ;
} A2 : { Richiesta(X) ; X= X*10; Richiesta(Y) ; Rilascio(X) ; Y= Y*10; Rilascio(Y) ;
}

```

Si può dimostrare che le inconsistenze esaminate precedentemente non possono più verificarsi. E' così soddisfatta la condizione di serializzabilità (s.p.d).

11.4 Proprietà "tutto o niente"

Malfunzionamenti: è possibile che l'esecuzione venga interrotta a causa di una **condizione anomala** (guasto, eccezione, crash, ecc.): affinché gli oggetti non rimangano in uno stato inconsistente, è necessario un meccanismo di recupero che, in seguito ad una condizione anomala, porti gli oggetti in uno stato consistente.

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

$$S' = S_1 \text{ oppure } S' = S_2$$

dove S1 e S2 sono rispettivamente lo stato iniziale e quello finale desiderato (corrispondente al completamento dell'operazione). La proprietà indicata prende il nome di tutto o niente:

“tutto” → completamento dell'operazione ($S' = S_2$)

“niente” → aborto: interruzione e ripristino dello stato iniziale ($S' = S_1$)

Un'azione atomica che gode della proprietà del tutto o niente è in grado di tollerare il verificarsi di condizioni anomale durante la sua esecuzione, senza lasciare gli oggetti in stati inconsistenti.

Esempio dell'applicazione bancaria:

- Evento anomalo dopo che x è stata addebitata ad O1, ma non ancora accreditata ad O2.
- O1 e O2 sono globalmente inconsistenti.

Vi è la necessità di un meccanismo di recupero che riporti lo stato complessivo dell'insieme {O1,O2} in uno stato consistente (S1 o S2):

- S2: $O1' = O1 - x;$ $O2' = O2 + x$
- S1: $O1' = O1;$ $O2' = O2;$

Se non è possibile raggiungere lo stato S2, il meccanismo di ripristino riporterà lo stato al valore S1: l'operazione di distruzione degli effetti di un'azione atomica ed il ripristino del valore iniziale degli oggetti viene indicata con il termine di aborto.

11.4.1 Effetto domino

Consideriamo la possibilità che, durante l'esecuzione di una azione atomica strutturata **conformemente al two-phase lock protocol** si possano verificare malfunzionamenti che interrompono l'elaborazione.

Esempio:

1. **A1** in esecuzione:

- viene acquisito X e viene eseguita $X = X + 20$
- viene acquisito Y
- viene rilasciato X, contenente il valore finale

2. **A2** in esecuzione:

- viene acquisito X e viene eseguita $X = X * 10$
3. A causa di un **malfunzionamento** (es. crash del computer su cui esegue A1) l'oggetto X viene riportato al valore che aveva prima che iniziasse A1 (A1 viene **abortita**);
4. Di conseguenza, è necessario propagare l'**aborto** anche a A2: gli effetti dell'ultima operazione di A2 su X e su tutti gli altri oggetti acceduti da A2 devono essere distrutti.

Effetto domino: l'aborto di un'azione atomica genera, come **effetto collaterale**, l'aborto di una diversa azione atomica (e così via).

Causa: l'azione atomica rilascia un oggetto **prima di aver completato** la sequenza di operazioni su tutti gli oggetti interessati e di aver raggiunto uno **stadio di avanzamento tale da garantire** che, da quel punto in poi, qualunque evento anomalo accada, l'azione atomica **non sarà più abortita**.

Per soddisfare la **proprietà del "tutto o niente"** occorre definire un ulteriore requisito:

d) Nessun oggetto può essere rilasciato prima che l'azione atomica abbia completato la sua esecuzione. In altri termini, i rilasci devono costituire le ultime operazioni dell'azione atomica.

11.4.2 Operazione commit

Per garantire la proprietà tutto o niente è necessario che il meccanismo di recupero si comporti in maniera diversa a seconda dell'istante in cui l'evento anomalo si verifica.

Dovrà:

- **abortire l'azione** se il malfunzionamento avviene quando gli oggetti sono in stato inconsistente (effetto niente)

- garantire il completamento dell'azione quando gli oggetti sono nel nuovo stato consistente S2 (effetto tutto)

A questo scopo viene introdotta l'operazione primitiva **commit**, che discrimina i due tipi di comportamento del meccanismo di recupero da malfunzionamento.

La commit viene eseguita quando tutti gli oggetti sono giunti al valore finale e produce come effetto l'impossibilità di aborto dell'azione atomica (completamento con successo).

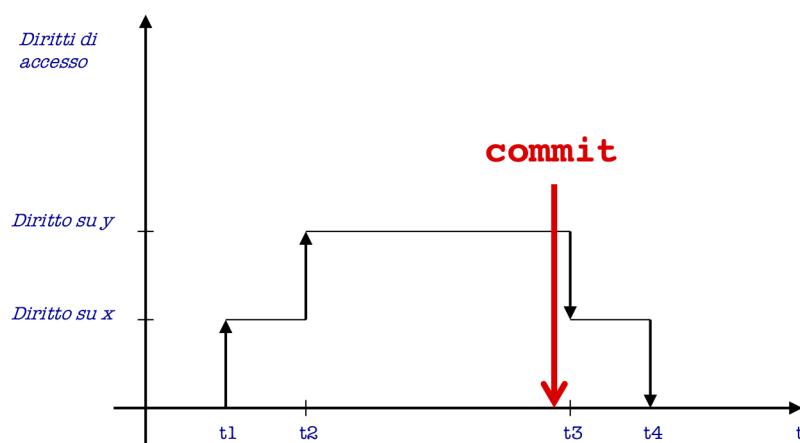
Il requisito d) si può riformulare nel seguente modo:

Ogni azione atomica deve rilasciare i propri oggetti dopo l'operazione commit.

```

A1 : {
    Richiesta(X);
    Richiesta(Y);
    X := X+20;
    Y := Y+20;
    commit;
    Rilascio(X);
    Rilascio(Y);
}
A2 : {
    Richiesta(X);
    Richiesta(Y);
    X := X*10;
    Y := Y*10;
    commit;
    Rilascio(X);
    Rilascio(Y);
}

```



11.4.3 Terminazione

Azione atomica termina in uno dei due modi:

- **Terminazione normale** - l'azione atomica completa l'intera sequenza delle operazioni sugli oggetti. Nuovo stato finale consistente (S2).
- **Terminazione anomala** – l'azione atomica non completa l'intera sequenza di operazioni sugli oggetti che devono essere ripristinati al valore iniziale (azione atomica abortita)

Cause di terminazione anomala:

1. Il verificarsi di un'**eccezione** sollevata durante una delle operazioni sugli oggetti. Il processo esegue la primitiva **abort**
2. Il verificarsi di un **malfunzionamento** del sistema prima che le operazioni siano terminate. Il sistema di recupero da malfunzionamento **forza l'aborto dell'azione atomica**

In entrambi i casi è necessario un meccanismo di supporto alle azioni atomiche in grado di ripristinare, per ogni oggetto, il suo stato iniziale.

Meccanismo di ripristino

Il meccanismo necessita di informazioni relative allo stato corrente delle operazioni ed allo stato iniziale degli oggetti.

Nel caso di eccezioni durante l'esecuzione dei programmi (caso 1) queste informazioni sono facilmente disponibili.

Nel caso di malfunzionamento hardware (interruzione di energia elettrica, guasto fisico, ecc.) si ha la perdita di tutte le informazioni residenti in RAM.

IPOTESI: Le informazioni su memoria di massa rimangono inalterate: le informazioni necessarie per il recupero vengono mantenute in memoria di massa.

11.5 Realizzazione della memoria stabile

Esiste la possibilità che le informazioni contenute nella memoria di massa risultino alterate in seguito a un malfunzionamento.

La memoria stabile rappresenta una astrazione con le seguenti proprietà:

1. non è soggetta a malfunzionamenti
2. le informazioni in essa residenti non vengono perdute o alterate a causa della caduta dell'elaboratore.

La prima proprietà si ottiene usando tecniche che fanno uso di ridondanza (es. tecnologia raid).

La seconda proprietà richiede che le operazioni di lettura e scrittura siano delle operazioni atomiche (proprietà del tutto o niente). Stable read e stable write.

Tipi di malfunzionamento

Si prendono in considerazione i seguenti tipi di errore:

- a) **errori in lettura:** Possono essere eliminati rileggendo le informazioni desiderate. Uso del controllo di ridondanza ciclica CRC (errori transitori)
- b) **errori in scrittura:** Sono rilevabili rileggendo le informazioni scritte e confrontandole con quelle originali. L'errore può essere eliminato riscrivendo le informazioni (errori transitori).
- c) **alterazioni delle informazioni** a causa di disturbi o di guasti hardware che perdurano per un certo numero n di letture consecutive (errori persistenti).

Per i problemi del tipo c) si fa ricorso alla tecnica delle copie multiple. Ridondanza di livello due: duplicazione di tutte le informazioni su due unità a disco distinte.

Classificazione dei malfunzionamenti

- Malfunzionamenti dovuti a disturbi temporanei: gli esempi a) e b)
- Malfunzionamenti dovuti a guasti hardware: esempio c) ed errori prodotti dalla caduta dell'elaboratore durante una scrittura.

Disco permanente: astrazione ottenuta eliminando ogni tipo di guasto temporaneo. Utilizzo di una coppia di dischi permanenti per realizzare l'astrazione memoria stabile.

Un **blocco stabile** è costituito da una coppia di blocchi uno per ogni disco permanente.

Ipotesi di base: le due copie non vengono alterate entrambe dallo stesso malfunzionamento. In tal modo, in fase di lettura almeno una deve contenere il valore corretto. Il valore contenuto in un blocco stabile coincide con il valore del primo blocco se questo contiene dati corretti, altrimenti coincide con il valore del secondo blocco.

La memoria stabile permette di avere le proprietà di serializzabilità, grazie al fatto che viene realizzata come un monitor e questo garantisce l'indivisibilità delle operazioni, e la proprietà del tutto o niente grazie al fatto che Stable read e Stable write godono di questa proprietà.

Stable read gode della proprietà del tutto o niente perché non effettua alcuna modifica.

Stable write se si verifica un guasto durante la operazione di scrittura sul primo blocco, questo viene alterato, ma rimane intatto il secondo (effetto niente della operazione). Se si verifica sul secondo blocco, rimane valido il primo (effetto tutto dell'operazione).

Tra le due operazioni: il risultato è non consistente. Il meccanismo di recupero legge i due blocchi: se uno è alterato viene sostituito con l'altro; se sono entrambi corretti, ma con valori diversi significa che c'è stato un guasto tra le due scritture quindi si copia il primo blocco nel secondo.

Le azioni atomiche vengono spesso utilizzate come strumento per strutturare applicazioni transazionali, ad esempio gestione di basi di dati.

Transazione: sequenza di operazioni effettuate su data base che fanno passare il sistema da uno stato all'altro, ambedue consistenti. Deve rispettare 4 proprietà:

ACID (Atomic, Consistency, Isolation, Durability)

1. ATOMICITA': transazione come entità indivisibile
2. CONSISTENZA: evoluzione del data base da uno stato corretto all'altro
3. ISOLAMENTO: informazioni protette durante l'esecuzione delle transazione (serializzabilità)
4. DURATA: una volta che la transazione sia completata i suoi effetti sul data base hanno carattere duraturo. Possono essere alterati soltanto da altre transazioni e non da malfunzionamenti del sistema.

Per specificare il tipo di operazione che deve seguire il meccanismo di recupero dopo la caduta dell'elaboratore, è necessario caratterizzare lo stato del processo in base allo stato di avanzamento nell'esecuzione dell'azione atomica.

11.6 Azioni Atomiche: caso monoprocesso

Il processo che esegue un'azione atomica può attraversare i seguenti stati:

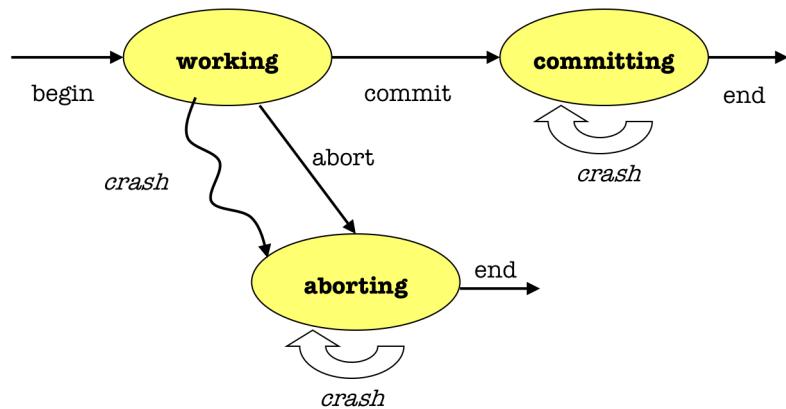
- **Stato working**

Durante l'esecuzione del corpo dell'azione atomica. Quando il processo è in questo stato gli oggetti sono inconsistenti. Se l'elaboratore cade, il meccanismo di recupero deve abortire l'azione atomica.

- **Stato committing**

Durante la terminazione corretta dell'azione. Gli oggetti sono al loro stato finale. Il processo commuta in tale stato tramite la commit. Se l'elaboratore cade il meccanismo di recupero deve completare l'azione (valori finali già disponibili)

- **Stato aborting** Durante la terminazione anomala dell'azione atomica. Gli oggetti devono essere ripristinati al loro valore iniziale. Il processo commuta in tale stato tramite abort. Se l'elaboratore cade durante l'azione di aborto, il meccanismo di recupero deve garantire il completamento del ripristino dei valori iniziali degli oggetti.



11.6.1 Descrittore di azione e primitive

Le informazioni sullo stato in cui si trova un processo che esegue un’azione atomica vengono tenute aggiornate nel **Descrittore di Azione**, una struttura dati allocata in memoria stabile. La realizzazione dell’azione atomica si basa sulle primitive:

- **begin action** crea in memoria stabile un descrittore di azione, inizializzando lo stato del processo a working;
- **abort** e **commit** dovranno commutare atomicamente lo stato del processo rispettivamente ad aborting e committing

Durante la riattivazione dell’elaboratore, il meccanismo di recupero può desumere, analizzando il descrittore dell’azione atomica, lo stato dei processi al momento del malfunzionamento.

11.6.2 Gestione degli oggetti

Gli oggetti risiedono in memoria stabile.

All’inizio dell’azione atomica viene creata una copia degli oggetti (copia di lavoro) in memoria volatile sulla quale eseguire le operazioni.

L’**aborto** dell’azione atomica coincide con la distruzione delle copie di lavoro in memoria volatile (in memoria stabile esistono i valori iniziali). Nella **terminazione**, soltanto se l’azione termina correttamente i valori finali delle copie di lavoro sono salvati nella copia valida in memoria stabile.

```

<creazione descrittore azione atomica>; /*begin action*/
/*fase crescente del two phase lock protocol:*/
< acq. esclusiva oggetti residenti in memoria stabile>;
<creazione copie volatili degli oggetti>;
/*corpo dell’azione atomica:*/
<sequenza di operazioni sulle copie volatili>;
if (<assenza di malfunzionamenti>
    terminazione (descrittore);
else abort (descrittore);
    <distruzione copie volatili>;
/*fase decrescente del two phase lock protocol:*/
<rilascio oggetti in memoria stabile>;
<eliminazione descrittore> /*end action*/

```

La procedura terminazione viene eseguita nel caso di corretto completamento dell’azione atomica per salvare in memoria stabile i valori finali degli oggetti.

Copia delle intenzioni: copia dei valori finali creata in memoria stabile prima della commit (stato working), senza modificare la copia stabile originale.

Solo se la copia delle intenzioni è stata correttamente memorizzata, il suo valore viene successivamente trasferito nella copia originale.

```

void terminazione(descrittore_azione x) {
    <creazione copia intenzioni in mem. stabile>

    commit(x); /* working -> committing */

    <trasferimento valori da copia intenzioni a
     copia oggetti in memoria stabile>

    <distruzione copia intenzioni>

```

Le due fasi della procedura sono intervallate dalla primitiva commit:

- Se l'elaboratore cade durante la prima fase, la copia originale degli oggetti in memoria stabile rimane inalterata al valore iniziale. L'azione viene abortita e viene distrutta la copia delle intenzioni.
- Se la caduta avviene nella seconda fase, la copia finale può corrompersi ma rimane valida in memoria stabile la copia delle intenzioni. In fase di riattivazione la seconda fase della procedura viene eseguita dall'inizio (fase committing).

11.7 Azioni atomiche: caso multiprocesso

In alcune applicazioni può risultare conveniente che le operazioni sugli oggetti di una azione atomica siano eseguite non da un solo processo, ma da più processi.

Le azioni atomiche multiprocesso sono tipiche dei sistemi distribuiti dove gli oggetti possono essere allocati su nodi diversi. L'elaborazione complessiva rimane comunque un'azione atomica se gode delle due proprietà di **serializzabilità** e del **tutto o niente**.

Ipotesi: i singoli processi operano ciascuno su **sottoinsiemi** di oggetti disgiunti. Ciò consente di evitare competizione dei diversi processi su oggetti condivisi.

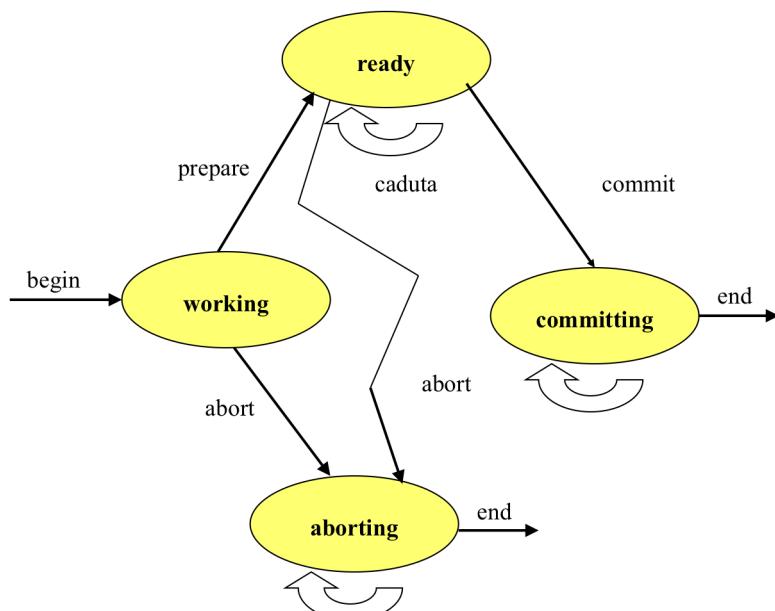
Se ogni processo esegue il **two phase lock protocol**, rilasciando gli oggetti su cui opera dopo la terminazione dell'azione, è assicurata la proprietà di serializzabilità dell'azione atomica.

Per soddisfare la proprietà del tutto o niente è necessario che tutti i processi completino le loro operazioni con lo stesso risultato: o con successo o con abort.

Un comportamento difforme (anche di un solo processo partecipante) porterebbe, alla fine dell'azione, alcuni oggetti al valore finale, altri al valore iniziale (stato inconsistente).

→ Necessità di introdurre un nuovo stato, stato **ready**:

Caratterizza un processo quando ha completato la propria sequenza di azioni ed è pronto a terminare con successo (commit), ma deve attendere che gli altri processi completino le loro operazioni.



La transazione da working a ready si ha quando il processo esegue la primitiva **prepare**. Con essa il processo perde il diritto di abortire unilateralmente. Nello stato ready diventa disponibile a terminare con successo o ad abortire.

Il protocollo che ciascun processo esegue per negoziare il tipo di completamento prende il nome di **two phase commit protocol**:

1. Nella prima fase ciascun processo specifica la propria opzione (completamento con successo o aborto).
2. Nella seconda fase viene verificata l'opzione degli altri: se tutti hanno optato per terminazione con successo, tutti transitano in stato committing; se almeno uno ha abortito, tutti transitano in stato aborting.

La primitiva prepare separa le due fasi del protocollo.

Se cade l'elaboratore quando un processo è in stato ready, il meccanismo di recupero deve ripristinare per tale processo lo stato ready. Infatti il processo non può essere riattivato in stato aborting poiché con prepare ha perduto il diritto di abortire unilateralmente, né in stato committing perché ciò presuppone che tutti siano pronti a terminare correttamente.

Il descrittore di azione atomica in memoria stabile dovrà tenere aggiornato lo stato di tutti i processi partecipanti.

La realizzazione dell'azione multiprocesso cambia a seconda del modello di interazione tra processi: nel modello a memoria comune il descrittore di azione è un monitor; nel modello a scambio di messaggi il descrittore è un oggetto privato di un processo detto coordinatore dell'azione atomica.

11.7.1 Modello a memoria comune

L'azione atomica viene iniziata da un processo che ne crea in memoria stabile il descrittore e attiva in parallelo un certo numero di processi che la eseguono.

Quando tutti i processi hanno completato le loro operazioni, con successo o terminando con aborto, il processo iniziale riprende il controllo e termina l'azione cancellando dalla memoria stabile il descrittore.

Schema del funzionamento del processo i-esimo:

1. Richiesta esclusiva degli oggetti
2. Creazione delle copie volatili degli oggetti
3. Sequenza di operazioni sulle copie volatili
4. Terminazione:
 - con aborto:
Entra nello stato aborting. Lo stato viene riportato sul descrittore della periferica.
Vengono risvegliati i processi nello stato di ready.
 - con successo:
Crea la copia delle intenzioni in memoria stabile;
Transita nello stato di ready attraverso l'esecuzione della prepare; viene modificato il suo stato da working a ready nel descrittore dell'azione atomica;
Verifica se almeno un processo ha abortito (è in stato aborting):
 - In caso affermativo, distrugge la copia delle intenzioni ed esegue abort
 - In caso negativo si possono verificare due casi:
 - * Qualche altro processo è ancora in stato di working: il processo resta nello stato ready e si blocca;

- * Tutti i processi sono nello stato di ready: il processo entra nello stato di committing, trasferisce gli oggetti dalla copia delle intenzioni a quella originale e distrugge la copia delle intenzioni. Risveglia un altro processo che si era precedentemente bloccato; il processo svegliato completa la sua esecuzione, risvegliando a sua volta un altro processo, e così via..

11.7.2 Modello a scambio di messaggi

Processo coordinatore: gestisce il descrittore dell'azione atomica. Può essere uno dei processi che realizzano l'azione atomica. Crea il descrittore dell'azione atomica ed attiva in parallelo tutti i processi partecipanti.

Two phase commit protocol: lato coordinatore

- Il processo **coordinatore**, qualora sia disponibile a terminare con successo l'azione atomica, crea la copia delle intenzioni in memoria stabile ed esegue la prepare per entrare nello stato ready.
Invia ad ogni partecipante un messaggio di richiesta esito e rimane in attesa delle risposte.
- Se arrivano una o più risposte di **esito negativo** l'azione deve essere abortita (passo 5). Se tutti i partecipanti rispondono con esito positivo l'azione deve terminare con successo (passo 3).
- Viene eseguita la primitiva **commit** e viene inviato a ciascun partecipante un messaggio con l'indicazione di **terminare con successo**. Il coordinatore rimane in attesa del messaggio di **avvenuto completamento** da parte di tutti i partecipanti.
- All'arrivo di tutti i messaggi di **avvenuto completamento** il coordinatore termina eliminando il descrittore dell'azione atomica.
- In caso di almeno un esito negativo,viene eseguita la primitiva abort e viene inviato il messaggio di **completare con aborto** a tutti i partecipanti. Il coordinatore termina abortendo a sua volta.

Ogni **partecipante** esegue il seguente algoritmo:

- Terminate le operazioni sugli oggetti, il processo può aver abortito (stato aborting) o aver creato la copia delle intenzioni (stato ready). In entrambi i casi attende il messaggio richiesta-esito da parte del coordinatore.
- Risponde con esito positivo o esito negativo a seconda dello stato in cui si trova. Se è in stato aborting termina abortendo, altrimenti resta in attesa del messaggio di completamento da parte del coordinatore.
- Se viene ricevuto il messaggio completare con aborto, viene eliminata dalla memoria stabile la copia delle intenzioni ed il processo termina abortendo.
- Se viene ricevuto il messaggio completare con successo, il processo termina correttamente trasferendo i valori della copia delle intenzioni nella copia originale degli oggetti ed elimina la copia delle intenzioni. Invia al coordinatore il messaggio avvenuto completamento.

11.7.3 Azioni atomiche multiprocesso e sistemi distribuiti

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

- Caduta/malfunzionamento dei singoli nodi della rete
- Perdita (o invalidazione) dei messaggi in rete

Comportano il non arrivo a destinazione di alcuni messaggi o la non ricezione delle risposte da parte del processo mittente.

Si può utilizzare un temporizzatore per limitare il tempo di attesa di un messaggio. Al termine del tempo previsto, il processo viene riattivato e viene eseguita una procedura per richiedere nuovamente l'informazione oppure per abortire.

Per fronteggiare la caduta dei singoli nodi, ogni processo partecipante deve mantenere in memoria stabile delle variabili di stato da utilizzare in fase di riattivazione.

11.8 Azioni atomiche Nidificate

In base alle proprietà di serializzabilità e tutto o niente, un'azione atomica rappresenta un'unità di programma che non può essere nidificata entro altre azioni atomiche.

Nel caso in cui si hanno A1 e A2 azioni atomiche con A2 nidificata entro A1 (costituisce una delle operazioni eseguite da A1) si presentano problemi che non consentono la sua realizzazione, a meno di modifiche di proprietà delle azioni atomiche.

Problemi

Quando A2 termina, il processo che la esegue rilascia tutti gli oggetti su cui A2 ha operato. Gli oggetti sono disponibili per essere acquisiti da altri processi e quindi per operarvi all'interno di altre azioni atomiche.

→ Violazione della proprietà di serializzabilità per quanto concerne l'azione atomica A1:

Se A2 termina correttamente, essa modifica la copia stabile degli oggetti su cui ha operato e li rilascia.

→ Violazione della proprietà “tutto o niente” per l'azione atomica A1:

Se A1 per qualche motivo non termina correttamente dopo la conclusione di A2, abortisce e disfa tutte le modifiche effettuate sugli oggetti (tutto o niente). Devono essere quindi ripristinati al valore iniziale anche gli oggetti di A2. Operazione difficile perché per la corretta terminazione di A2 questi valori sono andati perduti.

Possibilità di un effetto domino in quanto gli oggetti di A2 possono essere stati acquisiti da altre azioni atomiche.

Limitazioni

Impossibilità di modularizzare un programma strutturato in termini di azioni atomiche.

Se fosse possibile:

- **strutturabilità** delle azioni atomiche
- Alcune delle azioni atomiche nidificate all'interno di un'altra azione atomica potrebbero essere eseguite in concorrenza in quanto per effetto del two phase lock protocol, eventuali competizioni per l'uso di oggetti comuni sarebbero automaticamente risolte.

Un'azione atomica nidificata potrebbe fallire senza implicare la terminazione anomala dell'intera azione atomica. Il programma potrebbe essere strutturato in modo da eseguire un'azione atomica alternativa (sottoazione)

Soluzione

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

1. Una sottoazione che termina con successo rende disponibili gli oggetti all'azione atomica (solo a questa) al cui interno la sottoazione è nidificata. Si evita la visibilità degli stati intermedi di un'azione atomica a causa della corretta terminazione di una sua sottoazione.

- Quando una sottoazione termina con successo, la seconda fase di commitment viene ritardata ed eseguita solo se l'azione più esterna termina con successo. In caso di aborto dell'azione esterna, tutte le sottoazioni devono essere abortite. La modifica degli oggetti in memoria stabile è effettuata dall'azione atomica più esterna.

Per garantire questo tipo di comportamento per ogni oggetto viene mantenuta aggiornata una pila di copie di lavoro. All'inizio di una sottoazione viene generata una nuova copia in testa alla pila. Se la sottoazione termina con successo, la copia in testa alla pila diventa la nuova copia di lavoro per l'azione più esterna; diversamente viene scartata la copia intesta alla pila.

11.8.1 Esempio: Chiamata di procedura remota in sistemi distribuiti

Le azioni atomiche sono particolarmente utili per strutturare sistemi transazionali (accesso da parte di più utenti alle informazioni contenute in un data base).

Il meccanismo di RPC è idoneo per le comunicazioni tra un processo transazionale e i processi che gestiscono la base di dati (modello cliente-servitore). Il cliente, dopo la richiesta di servizio, rimane in attesa della risposta (extended rendez-vous).

RPC rappresenta il meccanismo più idoneo per la strutturazione di sistemi transazionali distribuiti (client-server).

La realizzazione del meccanismo RPC in un sistema distribuito crea una serie di problemi legati ai malfunzionamenti propri di questi sistemi:

- guasti alla sottorete di comunicazione
- caduta dei singoli nodi

Gli effetti dei possibili malfunzionamenti:

- messaggio di richiesta perduto
- messaggio di risposta perduto
- crash del processo servitore durante il servizio

Per evitare attese indefinite da parte del cliente, si ricorre ad un meccanismo di temporizzazione nella RPC (time-out). La semantica delle RPC dipende dalle azioni intraprese allo scadere dell'intervallo di tempo.

Le proprietà delle azioni atomiche consentono di ottenere una di queste possibilità semantiche: "**at most once**:

In presenza di malfunzionamento durante la RPC questa viene abortita senza produrre effetti. Per ottenere ciò, la RPC viene gestita come un'azione atomica, nidificata all'interno di un'azione di più alto livello, costituente l'operazione eseguita dal processo cliente. È possibile abortire la RPC, garantendo la sopravvivenza dell'intera azione svolta dal processo cliente, che può decidere di scegliere strade alternative.

RPC, vista come azione atomica nidificata, rappresenta un'azione multiprocesso:

- processo cliente che esegue la chiamata e si sospende;
- processo servitore creato per eseguire l'azione.

I due processi devono eseguire il two-phase-commit protocol, al termine dell'intera operazione del processo cliente.

At-most-once è una possibile semantica. Esistono altre semantiche per la chiamata di procedura remota che non prevedono l'atomicità della chiamata.

Ad esempio: "**at least once**":

Se il processo è interrotto dal meccanismo di temporizzazione prima della ricezione dei risultati, si invia di nuovo il messaggio di richiesta.

Possibilità di esecuzioni multiple della stessa procedura da parte del processo servitore. Occorre che le procedure siano idempotenti (una loro ripetuta esecuzione produce sempre gli stessi risultati). In caso contrario è necessario ricorrere a particolari algoritmi per riconoscere e scartare i messaggi ripetuti.