Sistemi Operativi M

Kevin Michael Frick

24 gennaio 2021

1. **Virtualizzazione**

   (a) Si confrontino Fast Binary Translation e paravirtualizzazione. In termini di efficienza è preferibile la paravirtualizzazione o l'architettura naturalmente virtualizzabile?

   > **R:** Fbt e paravirtualizzazione sono due possibili soluzioni per virtualizzare via software se il processore non fornisce supporto alla virtualizzazione. La fbt prevede che il VMM scansioni dinamicamente il codice del sistema operativo ospitato e sostituisca a tempo di esecuzione i blocchi che contengono istruzioni privilegiate con blocchi equivalenti contenenti chiamate al VMM. In questo modo la macchina virtuale è una replica esatta di quella fisica, ma la traduzione dinamica è molto costosa e le prestazioni ne soffrono. La paravirtualizzazione, invece, prevede che il VMM offra al sistema operativo una hypercall API che includa chiamate privilegiate al VMM, il quale le tradurrà al livello sottostante. I kernel dei SO ospiti devono essere modificati per supportare la chiamata di istruzioni di quel VMM, ma la paravirtualizzazione offre prestazioni migliori.

   (b) Si descriva dettagliatamente il funzionamento di un VMM di sistema (gestione trap, architettura). Quali problemi possono sorgere utilizzando un VMM di sistema e come si possono risolvere?

   > **R:** Un VMM di sistema prevede funzionalità di virtualizzazione integrate in un sistema operativo leggero, al contrario di quanto accade con un VMM ospitato che esegue come se fosse un'applicazione qualunque in un sistema operativo esistente. L'architettura di una CPU solitamente prevede due livelli di protezione o *ring*: supervisore e utente. Un VMM di sistema opera a ring supervisore, mentre il SO ospitato e le applicazioni della VM operano a ring utente.
   >
   > Si possono presentare due problemi: *ring deprivileging*, che porta un SO ospitato a non poter eseguire istruzioni privilegiate in quanto opera a ring utente, e *ring compression/aliasing*, che porta istruzioni privilegiate e non a eseguire allo stesso ring.
   >
   > Un'architettura si dice naturalmente virtualizzabile quando, per ogni istruzione privilegiata eseguita da un ring diverso da quello del supervisore, viene inviata una notifica al supervisore stesso, in modo da eseguire una routine di emulazione per ogni trap (*trap-and-emulate*). Questo risolve il problema del ring deprivileging, permettendo al SO ospitato di eseguire istruzioni privilegiate passando per il VMM. Il problema del ring compression può essere risolto mediante fbt o paravirtualizzazione, oppure servendosi di architetture naturalmente virtualizzabili che dispongono di almeno tre ring e permettono al SO ospitato di eseguire a un ring intermedio tra il VMM e le applicazioni.

   (c) Si descrivano architettura, paginazione, gestione delle interruzioni e dei driver di Xen. Cos'è un balloon process?

R: Xen è un VMM open source che opera in paravirtualizzazione. Le VM eseguono in autonomia le istruzioni non privilegiate, mentre l'esecuzione di quelle privilegiate viene delegata al VMM mediante le hypercall. In Xen il VMM è detto *hypervisor* e le VM sono *domains*. Una VM particolare, il *domain 0*, gode di privilegi diversi e deve rimanere sempre attiva per permettere il funzionamento degli altri domain.

I sistemi operativi ospitati gestiscono la memoria virtuale mediante la tradizionale paginazione: le page table delle VM vengono mappate in memoria fisica da Xen, il quale è l'unico a potervi accedere in scrittura su richiesta delle VM. L'accesso in lettura, invece, è permesso anche ai sistemi operativi ospitati. Xen risiede nei primi 64MiB dei virtual address space. Per ogni VM, lo spazio di indirizzamento virtuale è strutturato in modo da contenere Xen e il kernel in segmenti separati. Al momento della creazione di un processo, il SO ospitato richiede una page table a Xen, che ne restituisce una alla quale sono state aggiunte le pagine del segmento di Xen. Ogni successivo aggiornamento da parte del SO ospitato provoca una *protection fault* che verrà catturato e gestito da Xen. Per reclamare e ottenere, se necessario, pagine di memoria non utilizzate dalle diverse VM, su ognuna di esse è sempre attivo un *balloon process* che comunica con Xen e può venire chiamato in modo che richieda altre pagine al SO ospitato, il quale le allocherà e le cederà a Xen.

Ogni interruzione viene gestita direttamente dal SO ospitato, eccezion fatta per la *page fault*, che richiede accesso al registro CR2 accessibile solo a ring 0. La routine di gestione eseguita da Xen legge CR2 e lo copia in una variabile nello spazio del SO ospitato, al quale viene restituito il controllo per poter poi gestire la page fault.

Xen condensa i (*back-end*) driver nel domain 0, offrendo però alle VM un'interfaccia detta *front-end driver* che si occupi di trasferire la richiesta al back-end. Questo approccio garantisce portabilità, isolamento e semplifica il VMM, ma necessita di continua comunicazione con il back-end.

2. **Protezione e sicurezza**

   (a) Si definiscano i concetti di modelli, politiche e meccanismi di protezione. Si definiscano le politiche DAC, MAC e RBAC.

R: Un modello di protezione definisce *soggetti, oggetti, diritti di accesso*. I soggetti sono la parte attiva di un sistema e possono richiedere accesso alle risorse fisiche e logiche dette oggetti (e.g. i processi), oppure ad altri soggetti, mediante operazioni definite dai diritti d'accesso. A ogni soggetto viene associato un dominio di protezione, il quale rappresenta l'ambiente di protezione nel quale il soggetto esegue e specifica i diritti d'accesso nei confronti di ogni risorsa. Un processo può eventualmente cambiare dominio durante l'esecuzione.

Le politiche di protezione definiscono le regole secondo le quali i soggetti accedono agli oggetti. Solitamente le politiche vengono scelte da chi opera su un determinato sistema. Esistono tre tipi di politiche: *Direct Access Control* (DAC), secondo cui il proprietario di un oggetto (non necessariamente sempre lo stesso) decide la politica per quell'oggetto, decentralizzando quindi la gestione delle politiche, come accade in UNIX; *Mandatory Access Control* (MAC), secondo cui i diritti vengono definiti in maniera centralizzata; *Role-based Access Control* (RABC), secondo cui a ogni ruolo sono assegnati diversi diritti di accesso e ogni utente può avere diversi ruoli. A prescindere dalla politica attuata, è desiderabile che si rispetti il principio del privilegio minimo: ad ogni soggetto devono essere garantiti diritti d'accesso solo nei confronti degli oggetti strettamente necessari per la sua esecuzione.

I meccanismi di protezione sono gli strumenti messi a disposizione per imporre una determinata politica. Vanno realizzati rispettando i principi di flessibilità, secondo cui i meccanismi devono essere sufficientemente generali per consentire l'applicazione di diverse politiche, e di separazione tra meccanismi e politiche, secondo cui è la politica a definire "cosa vada fatto" mentre i meccanismi si preoccupano di "come fare".

(b) Si descriva il modello della matrice degli accessi e i concetti di copy flag, owner, cambio di dominio. Come è rappresentato il dominio di protezione? Si discuta la modifica dello stato di protezione (primitive di Graham e Danning). Si descrivano i diritti di accesso e la propagazione dei diritti.

R: Un dominio di protezione definisce un insieme di coppie formate dall'id di un oggetto e dall'insieme delle operazioni che il soggetto associato al dominio può eseguire su ciascun oggetto. Ogni dominio è associato univocamente a un soggetto, il quale può accedere solamente agli oggetti definiti nel proprio dominio utilizzando i diritti ivi specificati.

L'associazione tra processi e domini può essere statica o dinamica, in base alla presenza della possibilità per l'insieme delle risorse disponibili a un processo di variare durante il tempo di vita del processo stesso. L'associazione statica non permette di rispettare con continuità il principio del privilegio minimo.

È possibile rappresentare un sistema di protezione a livello astratto servendosi di una *matrice degli accessi* nella quale ogni riga è associata a un soggetto e ogni colonna a un oggetto. Un meccanismo associato a questo modello, per verificare il rispetto dei vincoli di accesso, controlla per ogni operazione M eseguita sull'oggetto O_j nel dominio D_i che la casella i, j della matrice degli accessi contenga effettivamente M. Per modificare lo stato di protezione, invece, il meccanismo associato deve implementare le operazioni di creazione/cancellazione di soggetti e oggetti e di lettura, grant, cancellazione, trasferimento dei diritti di accesso.

Un diritto di accesso può essere trasferito da un oggetto a un altro se possiede il *copy flag*, indicato con un asterisco ($*$) nella relativa cella della matrice degli accessi. Se è presente il copy flag è possibile trasferire un diritto e il relativo copy flag, perdendoli, o copiare il diritto senza copy flag, mantenendo però il diritto e il copy flag nella casella originaria.

Il diritto *owner* è un diritto particolare che permette a un soggetto designato come owner di un oggetto di assegnare o revocare qualunque altro diritto di accesso a tale oggetto ad altri soggetti. La presenza dei diritti owner e della copy flag indicano esplicitamente l'uso di una politica DAC.

Il diritto *control* permette a un soggetto di revocare diritti di accesso arbitrari a un altro soggetto sul quale ha diritto di control.

Il diritto *switch* permette a un soggetto di commutare al dominio di un altro soggetto con il quale ha il diritto switch.

(c) Si definiscano e si confrontino le ACL e le capability list. Sono possibili soluzioni miste? Come è possibile evincere la politica di access control guardando la matrice degli accessi?

R: La matrice degli accessi è una notazione astratta, ma nella pratica sarebbe molto sparsa visto che il numero di oggetti e soggetti è potenzialmente molto alto. Per questo motivo, quindi, si memorizzano solo le righe o le colonne.

La memorizzazione per righe prende il nome di *Access Control List* (ACL) e prevede

che a ogni oggetto sia associata una lista di coppie (soggetto, diritti) che contiene tutti i soggetti che possono accedervi e i relativi diritti di accesso. Prima di ogni operazione su un oggetto si verifica nella ACL del soggetto operante la presenza della coppia contenente il soggetto stesso e l'operazione. La ricerca viene eseguita anche su una lista che contiene i diritti d'accesso comuni a tutti i soggetti. La ACL può essere definita anche per gruppi di utenti, nel qual caso ha la forma (utente, gruppo, diritti). L'uso di ACL permette di revocare i diritti di accesso in maniera molto semplice, cancellando i diritti di accesso che si vogliono revocare direttamente dalla ACL dell'oggetto coinvolto. L'ACL risulta più vantaggiosa ogni volta che si compiono azioni che coinvolgono tutti i soggetti associati a un oggetto, ad esempio la rimozione di un file.

La memorizzazione per colonne prende il nome di *Capability List* (CL). La CL prevede, che per ogni soggetto, si abbia una lista di coppie (oggetto, diritti) le quali prendono il nome di *capability*. Per proteggere le CL da manomissioni esse vengono memorizzate nello spazio del kernel e l'utente può solo far riferimento a un puntatore che identifica la propria posizione all'interno della lista. Revocare i diritti di accesso usando le CL è più complesso rispetto al caso della ACL perché è necessario, per ogni dominio, verificare se esso contiene capability che fanno riferimento all'oggetto considerato.

È possibile adottare una soluzione mista: memorizzare in forma persistente le ACL e, al momento del primo accesso a un oggetto, restituire la CL relativa al soggetto richiedente se il diritto invocato è presente nella ACL. In questo modo è possibile per il soggetto accedere all'oggetto più volte senza dover interrogare più volte la ACL. In seguito all'ultimo accesso la CL viene distrutta.

(d) Si definiscano di modelli Biba e Bell-LaPadula e le regole di sicurezza associate, fornendo degli esempi di applicazione.

R: I modelli MAC più utilizzati sono Biba e Bell-LaPadula. Questi due modelli sono in conflitto e non possono essere usati contemporaneamente. Entrambi sono modelli multilivello: gli oggetti sono classificati come NC, confidenziali, segreti o top secret; i soggetti, invece, sono classificati in quattro livelli a seconda dell'oggetto più segreto che sono autorizzati a esaminare. La classificazione degli oggetti prende il nome di *sensitivity level*, quella dei soggetti di *clearance level*.

Il modello Bell-LaPadula è progettato per garantire segretezza e non integrità. Questo modello prevede due regole: la *proprietà di semplice sicurezza* garantisce che ogni processo in esecuzione a un dato livello di sicurezza possa leggere solo oggetti con un livello di sicurezza pari o inferiori, mentre la *proprietà di integrità* prevede che un processo possa scrivere solo su oggetti a livello di sicurezza pari o superiore.

Il modello Biba, invece, è progettato per garantire integrità e prevede che un processo possa leggere solo a livelli pari o superiori e scrivere solo a livelli pari o inferiori.

Le politiche di sicurezza multilivello coesistono con le regole ACL/CL e hanno la priorità su di esse.

3. **Modello a memoria comune**

(a) Si descriva il concetto di semaforo, le funzioni `up` e `down`, il funzionamento logico.

R: Un semaforo è una particolare struttura dati che permette di implementare semplicemente meccanismi *produttore-consumatore* e di mutua esclusione. Esso consiste in una variabile intera non-negativa `value`, e due procedure `up` e `down`. Quando un processo

vuole operare sulla risorsa gestita dal semaforo esso chiama la funzione `down`. Se `value` è non nullo, il processo lo decrementa, esegue le proprie operazioni e alla fine chiama la funzione `up`. In caso contrario il processo si mette in attesa finché non viene chiamata la funzione `up` da un altro processo che ha al momento accesso alla risorsa. `up` incrementa `value`.

Le due operazioni sono *atomiche* ovvero vengono eseguite in un'unica soluzione indivisibile. Questa proprietà permette di evitare corse critiche che possono sorgere se, ad esempio, due processi chiamano `down` contemporaneamente.

Un semaforo è realizzato tramite un intero non negativo che può avere un valore iniziale non nullo e una coda FIFO contenente i processi in attesa.

(b) Si dimostri la correttezza della soluzione con semafori al problema della mutua esclusione servendosi della relazione d'invarianza.

R: Dato un semaforo S, siano v_S l'intero non negativo associato, i_S il valore iniziale di v_S, n_{DS} il numero di `down` eseguite e n_{US} il numero di `up` eseguite. In ogni istante, $v_S = i_S + n_{US} - n_{DS}$. Allora, dato che $v_S \geq 0$, si ha che

$$n_{DS} \leq i_s + n_{US} \tag{1}$$

La eq. (1) prende il nome di relazione di invarianza.

Data la relazione di invarianza, è possibile dimostrare che un semaforo con $i_S = 1$ risolve correttamente il problema della mutua esclusione. Le tre condizioni necessarie sono:

- le sezioni critiche devono eseguire in modo mutualmente esclusivo;
- non devono verificarsi deadlock;
- un processo che non stia eseguendo la sua sezione critica non deve impedire ad altri processi di eseguire la propria;

La tesi di mutua esclusione è equivalente a richiedere che il numero di processi nella sezione critica sia $n_z \in \{0, 1\}$. Dato che è necessario eseguire una `down` per entrare nella sezione critica e una `up` per uscirne si ha

$$n_z = n_{DS} - n_{US} \tag{2}$$

Ma

$$n_{DS} \leq 1 + n_{US} \implies n_{DS} - n_{US} \leq 1 \implies n_z \leq 1 \tag{3}$$

e per costruzione a una `down` precede sempre una `up`, quindi

$$n_{DS} \geq n_{US} \implies n_{DS} - n_{US} \geq 0 \implies n_z \geq 0 \tag{4}$$

. □

L'assenza di deadlock si dimostra per assurdo: se ci fosse un deadlock, $n_z = 0$ e $v_S = 0$, ovvero nessun processo sarebbe in sezione critica e tutti i processi avrebbero eseguito una `down`, portando il contatore del semaforo a 0 e aspettando in coda.

Ma

$$n_z = 0 \implies n_{DS} = n_{US} \implies v_S = 1 + 2n_{US} \geq 1 \implies 0 \geq 1 \tag{5}$$

che è assurdo. □

La non interferenza di processi fuori dalla sezione critica è equivalente a richiedere che se $n_z = 0$ allora $v_S > 0$.

Dalla definizione di n_z

$$n_z = 0 \implies n_{DS} = n_{US} \tag{6}$$

per cui

$$v_S = 1 + n_{US} - n_{DS} = 1 \tag{7}$$

. □

(c) Servendosi della relazione di invarianza, si dimostri la correttezza della soluzione con semaforo evento al problema del vincolo di precedenza.

R: Dato un processo p che esegue una operazione a, si vuole che a possa essere eseguita solo dopo che un altro processo q abbia eseguito una operazione b. Si introduce un semaforo S con $i_S = 0$. Prima di eseguire a, p esegue `down(S)` e dopo aver eseguito b, q esegue `up(S)`. Si vuole dimostrare che a viene eseguita sempre dopo b.

Se per assurdo a venisse eseguita prima di b, in un qualunque istante di tempo compreso tra l'inizio di a e l'inizio di b sarebbe stata eseguita `down(S)` ma non `up(S)`, quindi

$$n_{DS} = 1, n_{US} = 0 \tag{8}$$

Ma per la relazione di invarianza

$$n_{DS} \leq i_S + n_{US} \implies 1 \leq 0 + 0 \tag{9}$$

che è impossibile. □

(d) Si descriva il concetto di barriera e la sua utilità. Si fornisca una implementazione in pseudocodice.

R: Si considerino due processi A, B che devono eseguire rispettivamente le azioni a_1, a_2 e b_1, b_2, con il vincolo che l'esecuzione di a_2 e b_2 richieda che siano state completate sia a_1 che b_2.

Per risolvere questo problema si introducono due semafori `s1, s2` con contatore inizializzato a 0. Il processo A chiama `up(s1); down(s2);` quando termina l'esecuzione di a_1, mentre il processo B chiama `up(s2); down(s1);` quando termina l'esecuzione di b_1. In questo modo, il processo che termina per primo blocca sulla `down` in attesa dell'altro processo.

Se i processi sono $n > 2$ ci si serve di una struttura più complessa detta *barriera di sincronizzazione*, composta da due semafori `mutex, barrier`, inizializzati a 0 e 1, e un

contatore `done`. Ogni processo che termina prende il mutex (`down(mutex)`), incrementa `done` e, solo se `done == n`, chiama `up(barrier)`. In seguito il processo rilascia il mutex (`up(mutex)`) e chiama `down(barrier); up(barrier);`.

In questo modo ogni processo attende la `up(barrier)` eseguita dall'ultimo processo che completa la propria operazione prima di chiamare le rispettive `up` e far tornare il semaforo `barrier` a 0.

(e) Si descriva l'implementazione di un semaforo nel kernel di un sistema monoprocessore.

R: Un semaforo può essere rappresentato come una struttura dati contenente un contatore `c` e una coda `q`. Una `down` su un semaforo con `c == 0` sospende il processo corrente `p` in `q` mediante una `push(q, p)`, mentre se `c > 0` il contatore viene decrementato. Una `up` su un semaforo con `q` vuota incrementa il contatore, mentre se `q` non è vuota estrae un processo `p` da `q` mediante una `s = pop(q)`.

In pseudo-C, nell'ipotesi che le interruzioni siano disabilitate durante l'esecuzione di `up`, `down` in modo da garantirne l'atomicità:

```
typedef struct {
  int c;
  queue q;
} semaphore;

void down(semaphore s) {
  if (s.c == 0) {
    // sospendi p in q
  }
  else {
    s.c--;
  }
}

void up(semaphore s) {
  if (!empty(s.q)) {
    // estrai p da q
    // sveglia p
  } else {
    s.c++;
  }
}
```

(f) Si descrivano i modelli SMP e a nuclei distinti e, in entrambi i modelli, l'implementazione dei semafori, delle relative operazioni e il meccanismo di segnalazione tra i nuclei nel caso di context switch.

R: In un'architettura multiprocessore, il modello SMP (symmetric multi-processing) prevede che un solo kernel sia condiviso tra tutte le CPU, mentre il modello a kernel distinti prevede che ci siano più istanze di un kernel, raggruppate in una collezione, che eseguano in modo concorrente.

Nel modello SMP un processo può andare in esecuzione su qualsiasi CPU, e la competizione tra le CPU nell'esecuzione (ad esempio per le chiamate alle system call) necessita di un meccanismo di sincronizzazione. Tale sincronizzazione può essere realizzata mediante un *unico lock*, sulle quali vengono chiamate `lock, unlock` prima e dopo ogni richiesta al kernel, limitando però il grado di parallelismo. Operando in questo modo, ad esempio, non è possibile eseguire contemporaneamente funzioni del kernel che operano su strutture diverse (ad esempio una `down` su due semafori diversi). Per aumentare il grado di parallelismo si associa *un lock a ogni risorsa.* Le varie richieste possono essere equamente schedulate tra i vari processori (*load balancing*) ma in alcuni casi può essere vantaggioso assegnare un processo a un determinato processore (es. uso di memorie private per-processore in caso di accesso non uniforme, uso dei dati in cache in un determinato processore), richiedendo però in questo caso una *ready queue* per nodo invece di una sola.

Il modello a kernel distinti prevede come assunzione di base che i processi che eseguono si possano dividere fra più nodi virtuali con poche interazioni reciproche. Ognuno di questi nodi virtuali è mappato a un nodo fisico e tutte le interazioni locali a un nodo virtuale possono eseguire contemporaneamente a quelle locali agli altri nodi. La memoria comune viene utilizzata solo per permettere a processi locali a nodi virtuali diversi di interagire. Nel modello a kernel distinti un processo può eseguire solo sul nodo contenente il relativo descrittore, quindi è vincolato a eseguire sempre sullo stesso nodo e non è possibile il load balancing.

In SMP i semafori sono realizzati proteggendo gli accessi ai contatori e alla coda dei processi pronti mediante lock. Se si usa un lock per ogni risorsa, due operazioni `down` su due semafori diversi possono operare contemporaneamente se e solo se non risultano sospensive, perché i semafori hanno lock diversi ma la coda dei processi pronti è una risorsa condivisa. Ad esempio, nello scheduler pre-emptive con priorità se una `up` porta in esecuzione un processo con priorità superiore a uno dei processi in esecuzione, anche in un'altra CPU, occorre che il kernel revochi l'accesso alla CPU a uno di questi ultimi e lo assegni al processo riattivato. È quindi necessario prevedere che ogni kernel mantenga informazioni sul processo a più bassa priorità in esecuzione e su quale esso operi; si rende inoltre necessario l'invio di interrupt hardware alle varie CPU.

Nel modello a kernel distinti si opera la distinzione tra semafori privati di un nodo virtuale, utilizzati solo da processi appartenenti a tale nodo, e semafori condivisi tra nodi virtuali, le cui relative informazioni sono contenute nella memoria comune. Ogni semaforo condiviso viene rappresentato come:

- un intero in memoria comune, protetto da un lock in memoria comune;
- una coda locale per ogni nodo contenente i descrittori dei processi locali sospesi nel semaforo;
- una coda globale di tutti i rappresentanti dei processi sospesi sul semaforo; il rappresentante di un processo identifica il nodo fisico su cui opera e il descrittore contenuto nella memoria privata del nodo;

Una `down` sospensiva su un semaforo condiviso porta a inserire il rappresentante del processo chiamante nella coda globale e il descrittore nella coda locale; una `up`, invece, estrae un processo dalla coda globale, ne comunica l'identità al nodo virtuale relativo (tramite un'interruzione, per garantire il rispetto della priorità) il quale risveglia il processo estraendo il descrittore dalla propria coda locale.

In pseudo-C:

```
void down(semaphore s) {
```

```
  if (is_private(s)) {
    // down come in monoprocessore
  } else {
    lock(s.common_lock);
    // down
    // se necessario sospende il rappresentante nel processo in s.q
    unlock(s.common_lock);
  }
}

void up(semaphore s) {
  if (is_private(s)) {
    // down come in monoprocessore
  } else {
    lock(s.common_lock);
    if (!empty(s.q)) {
      if (s.node == current_node) {
        // down come in monoprocessore
      } else {
        // estrae p da s.q
        int ch = get_buffer(p.node);
        while (busy(ch)) {}
        send(ch, p.id);
        interrupt(p.cpu);
      }
    } else {
        p.c++;
    }
    unlock(s.common_lock)
  }
}
```

4. **Modello a scambio di messaggi**

   (a) Si definiscano i concetti di canale sincrono, asincrono, simmetrico, asimmetrico e i relativi vantaggi e svantaggi. Cosa sono link, port, mailbox?

R: Esistono tre tipi di canale di comunicazione tra processi: *link*, canali simmetrici uno-a-uno; `port`, canali asimmetrici molti-a-uno; *mailbox*, canali asimmetrici molti-a-molti.

Un canale è asimmetrico se non è uno-a-uno.

I canali asincroni prevedono che i processi mittente continuino la propria esecuzione dopo l'invio del messaggio. I canali sincroni con rendez-vous semplice prevedono che il primo dei due processi comunicanti che accede al canale per inviare o ricevere si sospenda in attesa che l'altro sia pronto ad eseguire l'operazione controparte. I canali sincroni con rendez-vous esteso, nell'ipotesi in cui ogni messaggio inviato rappresenti una richiesta al destinatario di esecuzione di un'azione, prevedono che il processo mittente rimanga in attesa finché l'azione non termina. Il rendez-vous esteso è semanticamente analogo alla chiamata di procedura remota: il processo chiamante invia la richiesta e blocca in attesa del risultato; il processo server accetta la richiesta, la esegue e invia i risultati, che quando vengono ricevuti sbloccano il chiamante.

(b) Si descrivano le primitive di send e receive. In che modo è possibile costruire una send sicrona mediante una asincrona e viceversa?

R: `send(value, ch)` è una primitiva che invia un dato valore a un canale identificato univocamente. La send può essere asincrona nel caso il canale sia bufferizzato e sincrona se il canale ha capacità nulla, nel qual caso il processo attende che il ricevente esegua la `receive` prima di proseguire la propria esecuzione.

`receive(var, ch)` è una primitiva che legge in una variabile il valore ricevuto da un canale identificato univocamente. La receive può essere bloccante se sospende il processo finché non c'è un messaggio nel canale.

L'istruzione di livello più basso è la send asincrona. Per implementare una send sincrona si invia un messaggio e si rimane in attesa di un segnale di `ack`, sfruttando la semantica bloccante della receive.

Per implementare una send asincrona con buffer lungo n da una sincrona, invece, si può utilizzare una mailbox concorrente. Una mailbox è costituita da un insieme di n processi concorrenti collegati a cascata, nel quale ogni processo esegue una receive dal precedente e una send verso il successivo, con il primo che riceve dal produttore e l'ultimo che invia al consumatore.

(c) Si descrivano i comandi con guardia, i motivi per cui vengono utilizzati, i possibili stati della guardia e i costrutti alternativo e ripetitivo. Perché il comando con guardia alternativo non è deterministico?

R: Un meccanismo di ricezione ideale consente al processo ricevente di verificare la disponibilità di eventuali messaggi ed eventualmente riceverli da vari canali, bloccandosi in attesa di un messaggio da un canale qualunque se tutti i canali sono vuoti. Il canale su cui eseguire la `receive`, che è bloccante, deve quindi poter essere scelto in maniera non deterministica. Questo meccanismo viene realizzato mediante i comandi con guardia, che hanno la struttura `guardia -> istruzione`, con `guardia == (condizione, receive)`. `condizione` viene detta guardia logica, `receive` viene detta guardia d'ingresso. Una guardia si dice *fallita* se `condizione == false`, *ritardata* se `condizione == true` ma il canale della `receive` è vuoto, *valida* se `condizione == true` e la `receive` va a buon fine.

I costrutti alternativo (`select`) e ripetitivo (`do`) racchiudono un numero arbitrario di comandi con guardia semplici.

Nel caso della `select`, se una o più guardie sono valide viene eseguita, in modo non deterministico per non imporre una politica preferenziale tra i vari servizi, una delle istruzioni relative al ramo di una delle guardie valide, che viene eseguita; l'esecuzione del comando termina. Se tutte le guardie non fallite sono ritardate, il processo si sospende in attesa di una guardia valida. Se tutte le guardie sono fallite, il comando termina.

Nel caso della `do`, il comportamento è analogo a quello della `select`, ma il ciclo ricomincia tutte le volte che viene eseguita un'istruzione e termina solo se tutte le guardie sono fallite.

(d) Si discutano i concetti di rendez-vous e RPC.

R: La sincronizzazione estesa è un meccanismo di comunicazione che prevede che un processo chiamante richieda un servizio e rimanga sospeso fino al completamento del servizio richiesto. Semanticamente, la sincronizzazione estesa è analoga a una chiamata

di funzione, dato che il programma chiamante prosegue solo dopo che l'esecuzione è completata.

Sono possibili due diverse implementazioni lato server: chiamata di procedura remota (RPC) e rendez-vous esteso. Secondo il modello RPC il server dichiara una procedura per ogni servizio che può essere richiesto; al momento dell'effettiva richiesta, il server esegue una `fork` e il processo figlio chiama la procedura corrispondente e invia i risultati al client. Secondo il modello di rendez-vous esteso ogni operazione viene specificata come un insieme di istruzioni preceduta da una istruzione di `accept` che sospende il server in attesa della richiesta, in seguito alla quale il server esegue le istruzioni e invia i risultati al chiamante.

RPC rappresenta solo un meccanismo di comunicazione e delega al programmatore la sincronizzazione tra i processi figli del server. Il rendez-vous esteso, invece, si basa su un solo processo servitore al cui interno sono definite le istruzioni che realizzano i vari servizi, con la `accept` a permettere la sincronizzazione C/S.

La `accept` è bloccante se non sono presenti richieste di servizio; se invece ci sono più richieste per uno stesso servizio, esse vengono inserite in una coda FIFO. A uno stesso servizio possono essere associate più accept; l'utilizzo delle guardie permette di servire le richieste in base ai valori di un insieme di variabili di stato interne.

5. **Sistemi distribuiti**

(a) Si discuta la gestione del tempo in un sistema distribuito. Si descriva l'algoritmo di Lamport.

R: In un sistema distribuito gli orologi di ogni nodo non sono necessariamente sincronizzati. L'ordine nel quale due eventi vengono registrati può quindi essere diverso da quello in cui sono effettivamente accaduti.

Gli orologi utilizzati nelle applicazioni distribuiti si dividono in *fisici*, che forniscono l'ora esatta, e *logici*, che permettono di associare a ogni evento un timestamp coerente con l'ordine in cui gli eventi si sono effettivamente verificati. Per implementare gli orologi logici, si definisce una *relazione happened-before* $\langle\rightarrow\rangle$ tale che:

- se A, B sono eventi in uno stesso processo e A si verifica prima di B allora $A \rightarrow B$;
- se A è l'invio di un messaggio e B è la ricezione dello stesso messaggio allora $A \rightarrow B$;
- vale la proprietà transitiva: $(A \rightarrow B \wedge B \rightarrow) C \implies A \rightarrow C$.

Se a ogni evento e viene associato un timestamp $C(e)$ e si vuole che tutti i processi concordino su questo, è necessario che $A \rightarrow B \iff C(A) < C(B)$. Per garantire il rispetto di questa proprietà, l'*algoritmo di Lamport* prevede che:

- ogni processo p_i gestisca un contatore c_i;
- ogni evento in p_i fa incrementare c_i di 1;
- ogni messaggio m inviato da p_i provoca un incremento di c_i e ha come timestamp $C(m)$ il contatore incrementato;
- quando un processo p_j riceve un messaggio assegna al *proprio* contatore c_j il valore $c_j = \max\{c_j, C(m)\} + 1$.

Questo algoritmo viene solitamente eseguito da uno strato di *middleware* che opera tra il livello rete e quello applicazione.

(b) Con riferimento agli algoritmi di sincronizzazione distribuiti, si descrivano il concetto di gestore centrale della mutua esclusione, l'algoritmo Ricart-Agrawala, l'algoritmo token ring.

R: Nei sistemi distribuiti è spesso necessario garantire che due processi non possano eseguire contemporaneamente alcune attività, ad esempio quelle che prevedono accesso a risorse condivise. Questo problema è risolvibile in maniera centralizzata, delegando la gestione della risorsa a un *coordinatore* al quale tutti i processi si rivolgono, o decentralizzata, sincronizzando i processi mediante algoritmi la cui logica è distribuita tra i processi stessi. In generale, questo secondo approccio è più scalabile dato che il coordinatore costituisce un collo di bottiglia. Le soluzioni al problema della mutua esclusione distribuita si dividono inoltre in *permission-based*, nelle quali ogni processo "chiede il permesso" di eseguire a uno o più altri processi, e *token-based*, nelle quali i processi si passano un token che concede l'autorizzazione a eseguire la propria sezione critica. Le soluzioni token-based sono sempre decentralizzate.

La soluzione con gestore centrale della mutua esclusione prevede la presenza di un coordinatore che esponga due primitive di richiesta e rilascio della risorsa. Le richieste vengono gestite una alla volta in una coda FIFO e i processi, se richiedono accesso alla risorsa mentre un altro processo la sta usando, si mettono in attesa finché non vengono svegliati dal coordinatore.

L'algoritmo di Ricart-Agrawala è una soluzione decentralizzata permission-based che richiede la presenza di un orologio logico sincronizzato. Ad ogni processo sono associati due thread concorrenti: `main`, che esegue la sezione critica, e `receiver` che riceve le autorizzazioni. Quando un main vuole entrare nella sezione critica esso manda una richiesta di autorizzazione (con timestamp) ai receiver di tutti gli altri nodi; quando tutti i receiver hanno concesso l'autorizzazione il main esegue la sezione critica; infine il main concede l'autorizzazione a tutte le altre richieste in attesa. Ogni receiver, al momento della ricezione di una richiesta, può trovarsi in stato *released* se il proprio main non ha inviato richieste, nel qual caso autorizza immediatamente; in stato *wanted* se il proprio main è in attesa di autorizzazione, nel qual caso autorizza solo se il timestamp della richiesta ricevuta è inferiore a quello della propria, altrimenti mette la richiesta in coda; in stato *held* se sta eseguendo la sezione critica, nel qual caso la richiesta viene messa in coda.

L'algoritmo token ring è un algoritmo decentralizzato token-based che prevede che i processi siano collegati tra di loro ad anello e si scambino tra loro un messaggio detto token. Il processo che ha il token dà il permesso di eseguire la propria sezione critica. Se un processo che riceve il token è in stato *wanted* allora esegue la sezione critica e passa il token, altrimenti (stato *released*) lo passa immediatamente.

Il gestore centrale è un algoritmo equo che impedisce la starvation ed è implementabile con solo tre messaggi per sezione critica (richiesta, autorizzazione, rilascio), ma il gestore centrale costituisce sia un collo di bottiglia che un *single point of failure*, rendendo la soluzione poco scalabile e non tollerante ai guasti. L'algoritmo di R-A è molto scalabile ma richiede un numero di messaggi che cresce linearmente con il numero di processi; inoltre, nonostante non vi sia un single point of failure, è comunque non tollerante ai guasti, dato che un processo che va in crash non può rispondere alle richieste di autorizzazione e gli altri processi non hanno modo di rilevare il motivo della mancata risposta, che potrebbe essere dovuto semplicemente a una sezione critica in esecuzione. Per risolvere questo ultimo problema è possibile prevedere che ogni richiesta preveda una risposta, positiva o di attesa; a questo punto il richiedente può impostare un timeout e se un processo non risponde viene considerato guasto ed escluso dal gruppo. Anche il token ring è facilmente scalabile, ma presenta lo stesso problema di fault tolerance dell'algoritmo di R-A (che viene risolto allo stesso modo, prevedendo sempre una risposta e implementando un

timeout) oltre a presentare la possibilità di perdere il token se il processo che lo detiene va in crash; quest'ultimo guasto è anch'esso difficile da rilevare. Usando l'algoritmo token ring inoltre, il numero di messaggi per sezione critica è potenzialmente infinito.

Nella gestione con coordinatore quest'ultimo può essere statico o scelto mediante un algoritmo di elezione che cambia coordinatore quando quello attuale smette di rispondere, in modo da rendere il sistema più tollerante ai guasti. L'algoritmo di elezione Bully prevede che ogni processo mandi un messaggio di "elezione" a tutti i processi con ID più alto e diventi coordinatore solo se non riceve nessuna risposta, altrimenti esce dall'elezione e smette di rispondere ai messaggi di elezione. In questo modo viene eletto il processo funzionante con ID più alto. L'algoritmo di elezione ad anello, invece, prevede l'assegnazione di una priorità ai processi, strutturata secondo una topologia ad anello. Quando si rende necessaria un'elezione, il processo che se ne è reso conto manda un messaggio di elezione contenente il proprio ID al primo successore funzionante nell'anello. Il processo che lo riceve aggiunge il proprio ID e inoltra il nuovo messaggio al proprio successore. Una volta che questo messaggio fa un giro completo dell'anello, il processo che se ne è reso conto (leggendo nel messaggio il proprio ID) designa come coordinatore il processo con ID più alto e notifica questa elezione al proprio successore, il quale notificherà il proprio successore ecc. ecc..