Guida Completa alla Sincronizzazione per Sistemi Operativi

Da Zero alla Soluzione d'Esame

Basato sui temi d'esame 2022-2024

21 giugno 2025

Indice

1 Capitolo 1: Il Problema Fondamentale

1.1 Perché la Sincronizzazione è Necessaria?

Immagina di avere un conto bancario condiviso con un'altra persona. Sul conto ci sono 100€. Tu e l'altra persona decidete, nello stesso momento, di prelevare 10€ dai due sportelli automatici.

Un'operazione di prelievo non è istantanea. A livello di CPU, potrebbe essere così:

1. Leggi il saldo attuale dal conto (100€).
2. Calcola il nuovo saldo (100€ - 10€ = 90€).
3. Scrivi il nuovo saldo sul conto (90€).

Cosa succede se le operazioni dei due sportelli si intrecciano (interleaving)?

Passo 1: Sportello A: Legge il saldo (100€).

Passo 2: Sportello B: Legge il saldo (100€).

Passo 3: Sportello A: Calcola il nuovo saldo (90€).

Passo 4: Sportello B: Calcola il nuovo saldo (90€).

Passo 5: Sportello A: Scrive il nuovo saldo (90€) sul conto.

Passo 6: Sportello B: Scrive il nuovo saldo (90€) sul conto.

Risultato disastroso: Sono stati prelevati 20€ in totale, ma il saldo finale è 90€, non 80€. Abbiamo perso 10€! Questo fenomeno si chiama **Race Condition** (condizione di gara): il risultato dipende dall'ordine casuale con cui vengono eseguite le istruzioni.

La **sezione critica** è la parte di codice che accede alla risorsa condivisa (il saldo). Dobbiamo garantire che solo un processo alla volta possa entrare nella sua sezione critica. Questo si chiama **Mutua Esclusione**.

2 Capitolo 2: I Monitor, la Stanza Protetta

2.1 L'Analoga della Stanza Protetta

Pensa a un monitor come a una **stanza speciale** che contiene i dati condivisi (il nostro conto bancario).

- **Una sola persona alla volta:** La stanza ha una porta con una serratura magica. Solo una persona (processo) può essere dentro in un dato momento. Quando uno entra, la porta si chiude a chiave. Quando esce, la porta si sblocca. Questa è la **mutua esclusione automatica**.
- **Procedure Entry:** Sono le uniche porte per entrare nella stanza. Sono le funzioni pubbliche del monitor.
- **Dati Condivisi:** Sono gli oggetti e le variabili all'interno della stanza, protetti dalla serratura.

2.2 Il Problema: Attendere Dentro la Stanza

La mutua esclusione non basta. Immagina un problema Produttore-Consumatore. Un produttore mette oggetti in un buffer, un consumatore li prende.

- **Consumatore:** Entra nella stanza (acquisisce il lock del monitor) per prendere un oggetto. Ma... il buffer è vuoto!

- **Cosa può fare?**
  - **Uscire e rientrare? NO.** Sprecherebbe tempo e CPU (busy-waiting).
  - **Aspettare dentro la stanza? NO.** Se aspetta, tiene la porta chiusa a chiave e il produttore non potrà mai entrare per aggiungere un oggetto! È un **deadlock**.

Ci serve un modo per "aspettare intelligentemente": il processo deve poter **rilasciare temporaneamente il lock del monitor** e mettersi in attesa, permettendo ad altri di entrare. Qui entrano in gioco le **Condition Variables**.

2.3 Le Condition Variables: le Sale d'Attesa

Una Condition Variable è come una **sala d'attesa specifica** all'interno della stanza principale.

- Ci possono essere più sale d'attesa, una per ogni "motivo di attesa" (es. "in attesa perché il buffer è pieno", "in attesa perché il buffer è vuoto").
- `c.wait()`: "La condizione che cerco non è vera. Vado nella sala d'attesa `c`. Ecco la chiave della porta principale, qualcun altro può entrare." Il processo si blocca e rilascia il lock del monitor.
- `c.signal()`: "Ho appena reso vera una condizione. Sveglio una persona che stava aspettando nella sala d'attesa `c`." Il processo svegliato non riparte subito, ma si mette in coda per riacquisire il lock della stanza principale appena possibile.

La Regola del `while`: *Quando un processo viene risvegliato da una `signal`, non ha la garanzia che la condizione sia ancora vera. Un altro processo potrebbe essere entrato prima e aver cambiato di nuovo lo stato. Per questo, il processo risvegliato deve* ***sempre ricontrollare la condizione in un ciclo*** *`while`.*

```
// SBAGLIATO (funziona solo in casi ideali)
if (buffer_vuoto) {
    cond_non_vuoto.wait();
}

// CORRETTO E SICURO
while (buffer_vuoto) {
    cond_non_vuoto.wait();
}
```

2.4 Esempio Didattico: Un Buffer Limitato

Problema: Creare un buffer che può contenere al massimo `N` interi.

1. **Stato del Monitor:**
   - `int buffer[N]`: l'array per i dati.
   - `int count = 0`: quanti elementi ci sono nel buffer.
   - `int in = 0, out = 0`: indici per inserimento ed estrazione (buffer circolare).
2. **Condizioni di Attesa:**
   - Un produttore deve aspettare se `count == N` (buffer pieno).

- Un consumatore deve aspettare se `count == 0` (buffer vuoto).

3. **Condition Variables:**
   - `condition is_not_full`: sala d'attesa per i produttori bloccati.
   - `condition is_not_empty`: sala d'attesa per i consumatori bloccati.

```
monitor BoundedBuffer {
    const int N = 10;
    int buffer[N];
    int count = 0, in = 0, out = 0;

    condition is_not_full, is_not_empty;

    procedure entry void aggiungi(int item) {
        // 1. Devo aspettare? Sì, se il buffer è pieno.
        while (count == N) {
            is_not_full.wait(); // Vado in sala d'attesa
        }

        // 2. La condizione è vera, posso lavorare.
        buffer[in] = item;
        in = (in + 1) % N;
        count++;

        // 3. Ho cambiato lo stato. Ho reso il buffer non vuoto.
        //    C'era qualcuno che aspettava per questo? Svegliamolo.
        is_not_empty.signal();
    }

    procedure entry int preleva() {
        // 1. Devo aspettare? Sì, se il buffer è vuoto.
        while (count == 0) {
            is_not_empty.wait(); // Vado in sala d'attesa
        }

        // 2. La condizione è vera, posso lavorare.
        int item = buffer[out];
        out = (out + 1) % N;
        count--;

        // 3. Ho cambiato lo stato. Ho reso il buffer non pieno.
        //    C'era qualcuno che aspettava per questo? Svegliamolo.
        is_not_full.signal();

        return item;
    }
}
```

2.5 Walkthrough d'Esame 1: Monitor `porto` (20-07-2022)

Testo (sintesi): Un porto ha una sola banchina. Una nave alla volta può attraccare. Un camion alla volta può scaricare i cereali nella nave. La nave arriva vuota e può salpare solo se è stata

riempita completamente. Se un camion può scaricare solo parzialmente il suo carico (perché la nave si riempie), rimane in porto e aspetta la prossima nave per scaricare il resto.

Passo 1: Identificare lo stato condiviso (la soluzione del prof. Davoli)

Cosa dobbiamo sapere per descrivere lo stato del porto?

- `bool is_molo_free = true`: C'è una nave al molo?
- `bool is_park_free = true`: C'è un camion che sta scaricando?
- `int dariempire = 0`: Quanti cereali mancano per riempire la nave attuale. Se è 0, significa che o la nave è piena, o non c'è nessuna nave.

Passo 2: Identificare le condizioni di attesa

1. Una **nave** arriva ma il molo è occupato → `condition ok2attracca;`
2. Una **nave** è al molo ma non è piena, quindi non può salpare → `condition ok2salpa;`
3. Un **camion** arriva ma un altro camion sta già scaricando → `condition ok2scarica;`
4. Un **camion** ha scaricato parzialmente e deve attendere la nave successiva → `condition ok2ancora;`

Passo 3: Analisi del codice (soluzione del prof. Davoli)

```
monitor porto {
  bool is_molo_free = true;
  bool is_park_free = true;
  int dariempire = 0;
  condition ok2attracca, ok2salpa, ok2scarica, ok2ancora;

  procedure entry void attracca(int capacita) {
    // 1. Devo aspettare? Sì, se il molo è occupato.
    while (is_molo_free == false)
      ok2attracca.wait();

    // 2. Il molo è mio. Lo occupo e imposto il mio stato.
    is_molo_free = false;
    dariempire = capacita;

    // 3. Ho cambiato lo stato. C'era un camion in attesa della "prossima nave"?
    //    Se sì, lo sveglio.
    ok2ancora.signal();
  }

  procedure entry void salpa() {
    // 1. Devo aspettare? Sì, se non sono ancora pieno.
    while (dariempire > 0)
      ok2salpa.wait();

    // 2. Sono pieno, posso salpare. Libero il molo.
    is_molo_free = true;

    // 3. Ho cambiato lo stato. C'era una nave in attesa di attraccare?
    ok2attracca.signal();
  }
```

```
  procedure entry void scarica(int quantita) {
    // 1. Posso parcheggiare per scaricare? No, se il posto è occupato.
    while (is_park_free == false)
      ok2scarica.wait();
    is_park_free = false; // Parcheggio.

    // 2. Ciclo per gestire lo scarico parziale.
    //    Questa logica viene eseguita finché il camion ha cereali da scaricare.
    while (quantita > dariempire) {
      // Caso: il camion ha più cereali di quelli che la nave può contenere.
      quantita -= dariempire; // Scarico ciò che posso.
      dariempire = 0; // La nave ora è piena.

      ok2salpa.signal(); // Dico alla nave che può partire.
      ok2ancora.wait(); // Mi metto in attesa della prossima nave.
      // Quando vengo svegliato da attracca(), il ciclo ricomincia
      // con la `quantita` rimanente e il `dariempire` della nuova nave.
    }

    // 3. Caso: il camion ha meno o tanti cereali quanti ne servono.
    dariempire -= quantita;
    if (dariempire == 0) // L'ho riempita esattamente?
      ok2salpa.signal(); // Dico alla nave di partire.

    // 4. Ho finito (per ora). Libero il parcheggio per il prossimo camion.
    is_park_free = true;
    ok2scarica.signal();
  }
}
```

2.6 Walkthrough d'Esame 2: Monitor `redblack` (15-02-2023)

Testo (sintesi): Implementare un monitor `redblack` con una sola entry `rb(color, value)`. I processi devono completare l'esecuzione in modo alternato per colore: se l'ultimo a finire era `red`, il prossimo deve essere `black`, e viceversa. La funzione ritorna la media dei valori passati da tutti i processi di quel colore sbloccati fino a quel momento.

Passo 1: Identificare lo stato condiviso

- `int last = -1`: Per sapere quale colore è stato l'ultimo a finire. Inizializzato a un valore nullo.
- `double sum[2]; int count[2]`: Due array per tenere traccia della somma e del conteggio dei valori per ciascun colore (`red=0`, `black=1`).

Passo 2: Identificare le condizioni di attesa

Un processo deve attendere se arriva e non è il suo turno.

- Un processo **rosso** arriva ma `last == red` → deve aspettare.
- Un processo **nero** arriva ma `last == black` → deve aspettare.

Possiamo usare un array di condition variables per gestire le due code di attesa separate. → `condition ok2col[2];`

Passo 3: Analisi del codice (soluzione del prof. Davoli)

```
monitor redblack {
    int last = -1; // -1: nessuno, 0: red, 1: black
    double sum[2] = {0.0, 0.0};
    int count[2] = {0, 0};
    condition ok2col[2]; // ok2col[0] per i rossi, ok2col[1] per i neri

    procedure entry double rb(int color, double value) {
        // 1. Devo aspettare? Sì, se l'ultimo colore è uguale al mio.
        if (color == last) {
            ok2col[color].wait();
        }

        // 2. È il mio turno. Aggiorno lo stato PRIMA di sbloccare altri.
        last = color;
        sum[color] += value;
        count[color]++;
        double mean = sum[color] / count[color];

        // 3. Ho finito. Sveglio un processo del colore opposto, se c'è.
        // La signal non blocca, il processo corrente prosegue e ritorna.
        ok2col[1-color].signal(); // 1-0=1 (sveglio black), 1-1=0 (sveglio red)

        return mean;
    }
}
```

Nota chiave: La `wait` è dentro un `if` e non un `while`. Qui è accettabile perché la logica è "a turni". Una volta che un nero sblocca un rosso, nessun altro nero può passare finché quel rosso non ha finito e ha sbloccato a sua volta un nero. La condizione `last != color` sarà sicuramente vera al risveglio.

2.7 Esempio di Riferimento: Monitor `bss` (26-05-2021)

Testo (sintesi): Un "buffer sincrono strampalato". Se più processi chiamano `put` quando nessun `get` è in attesa, si bloccano tutti. Quando arriva un `get`, riceve la lista di tutti gli elementi messi e sblocca tutti i `put`. Viceversa, se più `get` arrivano con buffer vuoto, si bloccano. Quando arriva un `put`, tutti i `get` vengono sbloccati e ricevono lo stesso valore.

Idea della soluzione: Usare dei contatori per i processi in attesa di `put` (`putting`) e di `get` (`getting`). Il comportamento cambia radicalmente a seconda se `getting > 0` o `putting > 0`.

```
monitor bss<T> {
  Vec<T> v = [];
  int putting = 0;
  int getting = 0;
  condition wait_for_get, wait_for_put;

  procedure entry void put(T value) {
    ++putting;
    v.push(value);

    // Se non c'è nessun "getter" in attesa, mi blocco con gli altri "putter".
    if (getting == 0) {
      wait_for_get.wait();
    }
```

```
    --putting;
    // Passaggio del testimone fra i "putter" sbloccati da un "getter".
    if (putting > 0) {
      wait_for_get.signal();
    }

    // Se c'erano "getter" in attesa, li sblocco.
    if (getting > 0) {
      wait_for_put.signal();
    }
  }

  procedure entry T[] get() {
    ++getting;

    // Se non ci sono "putter" in attesa (e quindi il buffer è vuoto), aspetto.
    if (putting == 0) {
      wait_for_put.wait();
    }

    Vec<T> res = v.copy();
    --getting;

    // Sblocco a cascata gli altri "getter" in attesa.
    if (getting > 0) {
      wait_for_put.signal();
    } else {
      // Sono l'ultimo getter, pulisco il buffer per il prossimo round.
      v.clear();
    }

    // Sblocco i "putter" che erano in attesa.
    if (putting > 0) {
      wait_for_get.signal();
    }
    return res;
  }
}
```

3 Capitolo 3: I Semafori, i Gettoni d'Ingresso

3.1 L'Analoga dei Gettoni

Un semaforo è più semplice di un monitor. Immagina un'attrazione a numero chiuso (es. una stanza con capienza massima 5 persone).

- **Semaforo Contatore (`S` init a 5):** All'ingresso c'è un cesto con 5 gettoni.
  - `P(S)` **(Prendi):** Per entrare, devi prendere un gettone dal cesto. Se non ci sono gettoni, aspetti fuori finché qualcuno non ne rimette uno.
  - `V(S)` **(Versa):** Quando esci, rimetti il tuo gettone nel cesto.
- **Semaforo Binario (Mutex, `S` init a 1):** C'è un solo gettone. Viene usato per proteggere una sezione critica, garantendo che solo un processo alla volta possa "possedere il gettone".

Regola dal Decalogo: Semafori *Un semaforo **deve** avere un valore iniziale. È errato un programma che usa un semaforo solo con `P` o solo con `V`. Tutti gli accessi a dati condivisi devono avvenire in mutua esclusione (usando un semaforo binario).*

3.2 Esempio Didattico: Produttore-Consumatore con Semafori

Problema: Buffer di `N` posti, produttori e consumatori.

Risorse da gestire:

1. **Accesso al buffer:** Solo un processo alla volta può modificare il buffer. → `semaphore mutex = 1;`
2. **Posti vuoti:** I produttori devono aspettare se non ci sono posti vuoti. Contiamo i posti vuoti. → `semaphore empty = N;`
3. **Posti pieni:** I consumatori devono aspettare se non ci sono item. Contiamo gli item. → `semaphore full = 0;`

```
// Stato condiviso globale
Item buffer[N];
int in = 0, out = 0;

semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void produttore() {
    while(true) {
        Item item = produce_item();

        // 1. C'è un posto vuoto? Prendo un "gettone empty".
        empty.P();

        // 2. Posso accedere al buffer? Prendo il "gettone mutex".
        mutex.P();

        // SEZIONE CRITICA
        buffer[in] = item;
        in = (in + 1) % N;

        // 3. Libero l'accesso al buffer.
        mutex.V();

        // 4. Ho aggiunto un item. Metto un "gettone full".
        full.V();
    }
}

void consumatore() {
    while(true) {
        // 1. C'è un item da consumare? Prendo un "gettone full".
        full.P();
```

```
        // 2. Posso accedere al buffer? Prendo il "gettone mutex".
        mutex.P();

        // SEZIONE CRITICA
        Item item = buffer[out];
        out = (out + 1) % N;

        // 3. Libero l'accesso al buffer.
        mutex.V();

        // 4. Ho liberato un posto. Metto un "gettone empty".
        empty.V();

        consume_item(item);
    }
}
```

3.3 Walkthrough d'Esame 1: Semaforo `wait4` (18-01-2023)

Testo (sintesi): Scrivere una funzione `wait4` che usa solo contatori e semafori. I processi che la chiamano devono proseguire a blocchi di quattro: i primi tre si bloccano, il quarto li sblocca tutti e prosegue anch'esso.

Passo 1: Identificare lo stato e i semafori

- Dobbiamo contare quanti processi sono arrivati nel gruppo corrente. → `int count = 0;`
- L'accesso al contatore `count` deve essere protetto. → `semaphore mutex = 1;`
- I processi devono bloccarsi in attesa che il gruppo si formi. → `semaphore barrier = 0;`

Passo 2: La logica di base e il problema del "furto di V"

L'idea ingenua potrebbe essere:

```
// NON CORRETTO
void wait4_sbagliato() {
    mutex.P();
    count++;
    if (count == 4) { // Sono il quarto!
        count = 0; // Resetto per il prossimo gruppo
        // Sblocco i 3 in attesa
        barrier.V(); barrier.V(); barrier.V();
        mutex.V();
    } else { // Non sono il quarto
        mutex.V();
        barrier.P(); // Aspetto
    }
}
```

Questo è sbagliato! Se un 5° processo arriva molto velocemente, potrebbe eseguire `barrier.P()` e "rubare" una delle `V()` destinate a uno dei primi tre. Dobbiamo usare un meccanismo più robusto: il **passaggio del testimone**.

Passo 3: La soluzione con Passaggio del Testimone

Il quarto processo sblocca solo il primo. Il primo sblocca il secondo. Il secondo il terzo. Il terzo sblocca il mutex per il gruppo successivo.

```
/* Esercizio c.2 del 18/01/2023, soluzione che usa passaggio del testimone */
int n = 0;
semaphore mutex = 1;
semaphore gate = 0; // Il semaforo su cui i primi 3 si bloccano

void wait4() {
    mutex.P();
    n++;
    if (n == 4) { // Sono il quarto
        // Non ho bisogno del mutex, il testimone farà il lavoro.
        // Sblocco il primo della catena.
        gate.V();
        // NB: Il mutex verrà rilasciato dall'ultimo della catena.
    } else {
        mutex.V(); // Rilascio il mutex per far entrare gli altri
        gate.P();  // Aspetto di essere sbloccato
    }

    // Una volta qui, sono stato sbloccato.
    // Devo passare il testimone al prossimo.
    n--;
    if (n > 0) { // Non sono l'ultimo della catena
        gate.V();
    } else { // Sono l'ultimo della catena (quello sbloccato per terzo)
        mutex.V(); // Rilascio il mutex per il prossimo gruppo di 4.
    }
}
```

Nota: In questa soluzione, c'è un piccolo bug. Il 4° processo chiama 'gate.V()' ma non esegue il blocco 'n–' ecc. Una versione più pulita e unificata è spesso preferita per evitare casi speciali. Tuttavia, il concetto di "passaggio del testimone" è il cuore della soluzione corretta. La versione dell'esame richiede che il 4° prosegua con gli altri, quindi la logica deve essere simile per tutti e 4 dopo il punto di sbarramento.

3.4 Esempio di Riferimento: Semaforo a Priorità LIFO (23-06-2021)

Testo (sintesi): Implementare un semaforo a priorità con primitive `PLP(prio)` e `PLV()`. La `PLV` deve sbloccare il processo in attesa con la priorità più alta. A parità di priorità, sblocca l'ultimo arrivato (LIFO).

Idea della soluzione: Non possiamo usare una singola coda di attesa, perché non è ordinata per priorità. L'idea è creare un semaforo *per ogni processo che si blocca*. Questi semafori "privati" vengono messi in una struttura dati condivisa (una pila ordinata per priorità) protetta da un mutex. La `PLV` estrarrà da questa struttura il semaforo giusto da sbloccare.

```
class OrderedLifoSemaphore {
  BinarySemaphore mutex;
  int initial, nP = 0, nV = 0;
  // Una pila che ordina per priorità, e LIFO a parità di priorità.
  OrderedStack<BinarySemaphore> s;

  constructor(int init) { initial = init; mutex = new BinarySemaphore(1); }
```

```
  void PLP(int prio) {
    mutex.P();
    // L'invariante dei semafori: il valore non deve scendere sotto zero.
    // value = initial - (nP - nV)
    if (initial - nP + nV <= 0) {
      // Devo bloccarmi. Creo il mio semaforo personale.
      BinarySemaphore new_sem = new BinarySemaphore(0);
      s.push(prio, new_sem); // Lo metto nella struttura dati ordinata
      mutex.V();
      new_sem.P(); // Mi blocco sul mio semaforo
      // Quando vengo sbloccato da una V, riparto da qui.
    }
    nP++;
    mutex.V();
  }

  void PLV() {
    mutex.P();
    nV++;
    // C'è qualcuno da sbloccare e il valore del semaforo lo permette?
    if (initial - nP + nV > 0 && !s.empty()) {
      s.pop().V(); // Estraggo il semaforo con priorità max e lo sblocco.
      // Il mutex viene rilasciato dal processo risvegliato.
    }
    else {
      mutex.V();
    }
  }
}
```

4 Capitolo 4: Message Passing, il Servizio Postale

4.1 L'Analoga del Servizio Postale

Con il message passing, i processi non condividono memoria. Sono come case separate. Comunicano inviandosi lettere.

- **Memoria Privata:** Ogni processo ha le sue variabili. Non esistono variabili globali condivise.
- **Asincrono (`asend`, `arecv`):** Come imbucare una lettera. `asend` non si blocca. `arecv` si blocca finché non c'è posta nella propria casella.
- **Sincrono (`send`, `recv`):** Come una consegna a mano. Mittente e destinatario devono essere entrambi presenti e pronti per lo scambio. Entrambi si bloccano.

4.2 Walkthrough d'Esame 1: `dispatcher` (21-06-2022)

Testo (sintesi): Un servizio client-server usa N server. Scrivere un processo `dispatcher` che riceve le richieste dai client e le distribuisce ai server liberi. Se tutti i server sono occupati, le richieste vengono messe in coda. Il dispatcher inoltra anche le risposte dai server ai client corretti.

Passo 1: La logica e lo stato del dispatcher

Il dispatcher è un processo intermediario che gira in un ciclo infinito. Le sue variabili private sono il suo stato.

- `pid_t server_pids[N]`: Un array con i PID degli N server, che sono noti.

- `pid_t serversender[N]`: Un array per tracciare lo stato dei server. `serversender[i] = None` se il server `i` è libero, altrimenti contiene il PID del client che sta servendo.
- `Queue q`: Una coda per le richieste dei client in attesa (`<client_pid, msg>`).

Passo 2: Gestire i messaggi in arrivo

Il dispatcher si mette in attesa di un messaggio da QUALSIASI mittente. Poi, deve capire chi glielo ha mandato.

```
// Codice del processo dispatcher (soluzione del prof. Davoli)
server[N] = ... // pids dei server
serversender[N] = [None, ..., None] // Stato dei server (libero/occupato)
Queue q; // Coda dei client in attesa

process dispatcher:
  while (true):
    <sender, msg> = arecv(ANY)

    // CASO 1: Il messaggio arriva da un server (è una risposta)
    if (sender in server):
      i = indexof(sender, server) // Trovo l'indice del server
      client_pid = serversender[i] // Recupero il client che serviva
      asend(msg, client_pid) // Inoltro la risposta al client

      // Ora il server 'i' è di nuovo libero. C'è lavoro per lui?
      if (!q.empty()):
        (new_client, new_msg) = q.dequeue()
        asend(new_msg, server[i]) // Gli do un nuovo lavoro
        serversender[i] = new_client // Aggiorno il suo stato
      else:
        serversender[i] = None // Nessun lavoro in coda, è libero

    // CASO 2: Il messaggio arriva da un client (è una richiesta)
    else:
      i = find_idle_server(serversender) // C'è un server libero?

      if (i == None): // No, tutti occupati
        q.enqueue((sender, msg)) // Metto il client in coda
      else: // Sì, il server 'i' è libero
        asend(msg, server[i]) // Inoltro la richiesta al server
        serversender[i] = sender // Aggiorno lo stato del server
```

Questo pattern è fondamentale: un singolo processo gestisce tutta la logica di smistamento e accodamento, mantenendo lo stato del sistema nelle sue variabili locali.

4.3 Walkthrough d'Esame 2: Ricezione Selettiva (`sarecv`, 26-05-2021)

Testo (sintesi): Implementare le primitive `sasend` e `sarecv` usando un servizio di message passing asincrono base (`asend`/`arecv`). La `sarecv(senders)` deve restituire un messaggio proveniente da uno dei PID nell'array `senders`. Se arrivano messaggi da altri mittenti, devono essere "messi da parte" per future chiamate a `sarecv`.

Idea della Soluzione

Questo non è un processo server, ma una **libreria**. Lo stato deve essere mantenuto a livello di processo chiamante, o globale se la libreria è condivisa. La chiave è avere un buffer per i messaggi "inattesi".

Stato e Logica

- **Stato:** Un dizionario (o una tabella hash) che mappa un `pid_t` a una coda di messaggi. Chiamiamolo `db`.
- **sasend(msg, dst):** Semplice. Deve solo aggiungere il PID del mittente al messaggio e usare `asend`.
- **sarecv(senders):**
  1. **Prima controlla il buffer:** Scorre l'elenco `senders`. Per ogni `sndr`, controlla se c'è un messaggio da lui nel nostro buffer `db`. Se sì, lo restituisce e ha finito.
  2. **Se il buffer è vuoto (per i mittenti desiderati):** Deve mettersi a ricevere messaggi dal sistema con `arecv(ANY)`.
  3. Entra in un ciclo `do-while`:
     - Riceve un messaggio `<sndr, msg> = arecv()`.
     - Controlla se `sndr` è nell'elenco `senders`.
     - **Se non c'è:** è un messaggio inatteso. Lo mette da parte: `db.get(sndr).push(msg)`.
     - **Se c'è:** ha trovato quello che cercava! Esce dal ciclo.
  4. Restituisce il messaggio trovato.

```
// Stato della libreria (globale o per-processo)
Dic<pid_t, Queue<msg_t>> db;

// sasend aggiunge il PID del mittente al messaggio
void sasend(msg_t msg, pid_t dst) {
  asend((getpid(), msg), dst);
}

msg_t sarecv(pid_t sndrs[]) {
  // 1. Controllo prima il mio buffer di messaggi inattesi.
  for (sndr in sndrs) {
    if (db.has(sndr) && !db.get(sndr).empty()) {
      return db.get(sndr).pop();
    }
  }

  // 2. Nessun messaggio utile nel buffer. Devo ricevere dal sistema.
  (pid_t, msg_t) sndr, msg;
  do {
    (sndr, msg) = arecv(); // Ricevo un messaggio qualsiasi

    // È uno dei mittenti che sto cercando?
    bool sender_is_wanted = (sndr in sndrs);

    if (!sender_is_wanted) {
      // No, metto il messaggio da parte per dopo.
      if (!db.has(sndr)) db.add(sndr, new Queue());
      db.get(sndr).push(msg);
    }
    // Il ciclo continua finché non trovo un mittente desiderato.
  } while (!sender_is_wanted);

  return msg;
}
```

5 Capitolo 5: Strategia Finale per l'Esame

1. **Deconstruisci il Problema:** Leggi il testo e sottolinea i vincoli. "Uno alla volta", "solo se pieno", "in ordine alternato". Questi sono i tuoi requisiti.
2. **Modella lo Stato:** Chiediti: "Quali informazioni devo conservare per sapere sempre in che stato si trova il sistema?". Queste sono le tue variabili condivise (per monitor/semafori) o private (per message passing).
3. **Trova i Punti di Attesa:** Chiediti: "Perché un processo dovrebbe fermarsi?". Ogni risposta è una potenziale `condition.wait()` o `semaphore.P()`.
4. **Scrivi il Codice Iterativamente:** Inizia con la struttura base. Aggiungi la mutua esclusione se usi i semafori. Aggiungi i cicli `while(...) wait()` per le attese. Infine, aggiungi le `signal()` o `V()` nei punti in cui lo stato cambia e potrebbe sbloccare qualcun altro.
5. **Applica il Decalogo come Checklist:** Una volta scritta una bozza, rileggi le regole del decalogo. La mia soluzione le rispetta tutte? Ho inizializzato tutto? Non ho fatto busy-waiting? Non ho mischiato paradigmi?

In bocca al lupo!