Guida alla Programmazione Concorrente

Basata sugli esami AA 2021–2024

Alessandro Amella

18 luglio 2025

Indice

1 Metodologia Generale di Risoluzione

Un approccio strutturato è la chiave per non perdersi nella complessità della concorrenza. Applica sistematicamente questi passaggi.

1. **Analisi del Problema:** Leggi attentamente il testo per identificare:
   - **Attori**: Processi (`nave`, `camion`), tipi (`neri`, `rossi`), etc.
   - **Risorse Condivise**: Buffer, stato (`molo_libero`, `last_color`), contatori.

- **Vincoli di Sincronizzazione**: Le regole che governano le interazioni. Esempi: *"un processo nero deve alternarsi con uno rosso"*, *"la nave parte solo se piena"*, *"il quarto processo sblocca i primi tre"*.
- **Obiettivo Finale**: Cosa deve realizzare ogni funzione e il sistema nel suo complesso.

2. **Scelta del Paradigma:** L'esercizio lo specifica. **Non mischiare mai primitive di paradigmi diversi**.
3. **Definizione dello Stato:** Elenca le variabili necessarie a descrivere lo stato del sistema. Questo è il passo più critico.
   - Contatori di processi in attesa: `int waiting_readers`, `int n_rossi_attesa`.
   - Flag di stato della risorsa: `boolean is_busy`, `int last_color`.
   - Strutture Dati: Code, stack, mappe per gestire richieste o dati in attesa.
4. **Implementazione della Logica:** Per ogni procedura, definisci le due fasi cruciali.
   - **Quando attendere?** Un processo deve bloccarsi quando una pre-condizione per la sua azione non è verificata. Usa un `if` (per monitor) o un ciclo `while` (per monitor con Signal-and-Continue) per controllare la condizione.
   - **Cosa notificare?** Un processo, dopo aver modificato lo stato, deve sbloccare i processi che attendevano quel cambiamento. Chi sblocco? Quelli la cui condizione di attesa è ora diventata vera.
5. **Verifica Mentale (Trace):** Simula l'esecuzione con scenari critici (es. due processi dello stesso tipo chiamano in sequenza, il buffer è pieno/vuoto) per individuare **race condition**, **deadlock** e **starvation**.
6. **Consultare il Decalogo:** Usa il decalogo di ogni sezione come checklist finale.

2 Risolvere con i Semafori

I semafori sono contatori atomici usati per mutua esclusione e sincronizzazione. Sono "stupidi": non hanno memoria o logica, sono solo contatori. Tutta la logica sta nel tuo codice.

2.1 Pattern 1: Mutua Esclusione (Mutex)

Protegge una sezione critica. Valore iniziale: 1.

```
semaphore mutex = new semaphore(1);
// ...
mutex.P();
// --- SEZIONE CRITICA --- (es. modifica di contatori o liste condivise)
mutex.V();
```

2.2 Pattern 2: Barriera di Sincronizzazione

Un punto in cui un gruppo di processi deve attendere. L'N-esimo processo che arriva "apre il cancello" per tutti.

2.2.1 wait4 (18 gennaio 2023)

Problema: Scrivere una funzione `wait4` che fa proseguire i processi a blocchi di quattro. I primi tre si bloccano, il quarto fa proseguire tutti.

Pattern Risolutivo

La logica è: serve un contatore per i processi arrivati, protetto da un mutex. Serve un semaforo "cancello" (la barriera) su cui tutti attendono.

```
// Stato globale
semaphore mutex = new semaphore(1);
semaphore barrier = new semaphore(0); // Il cancello e' chiuso
int waiting_count = 0;
const int N = 4;

void wait4(void) {
    mutex.P();
    waiting_count++;
    if (waiting_count < N) {
        // Non sono il quarto, devo aspettare
        mutex.V(); // Rilascio il mutex PRIMA di bloccarmi sulla barriera!
        barrier.P(); // Attendo al cancello
        // Una volta sbloccato, passo il testimone al prossimo
        barrier.V();
    } else {
        // Sono il quarto! Resetto il contatore per il prossimo gruppo
        waiting_count = 0;
        // Apro il cancello per il primo dei tre in attesa
        barrier.V();
        mutex.V(); // Rilascio il mutex
    }
}
```

Spiegazione: L'ultimo arrivato fa una `V` sulla barriera, sbloccando il primo. Questo, una volta superata la `P`, fa a sua volta una `V`, sbloccando il secondo, e così via in un "passaggio del testimone" a catena.

2.2.2 sumstop / sumgo (6 settembre 2022)

Problema: `sumstop` blocca i processi. `sumgo` sblocca **tutti** i processi in attesa e restituisce la somma dei loro valori.

Pattern Risolutivo

Simile a una barriera, ma lo sblocco è centralizzato. Il processo che chiama `sumgo` è responsabile di aprire il cancello per tutti, uno per uno.

```
// Stato globale
semaphore mutex = new semaphore(1);
semaphore gate = new semaphore(0);
int waiting_count = 0;
int total_sum = 0;

void sumstop(int v) {
    mutex.P();
```

```
    waiting_count++;
    total_sum += v;
    mutex.V();
    gate.P(); // Tutti si bloccano qui
}

int sumgo(void) {
    mutex.P();
    int sum_to_return = total_sum;
    int count_to_unblock = waiting_count;

    // Reset per il futuro
    total_sum = 0;
    waiting_count = 0;

    // Apro il cancello N volte
    for (int i = 0; i < count_to_unblock; i++) {
        gate.V();
    }

    mutex.V();
    return sum_to_return;
}
```

2.3 Decalogo per i Semafori

- Le operazioni sono `S.P()` e `S.V()`. Non hanno parametri né valore di ritorno.
- **Tutti** gli accessi a dati condivisi (contatori, liste...) devono avvenire in mutua esclusione.
- I semafori **devono** avere un valore iniziale. Inizializzarli a 0 significa "risorsa non disponibile/evento non accaduto". Inizializzarli a 1 è per i mutex.
- **Ordine delle `P()`**: per evitare deadlock, esegui prima le `P()` sui semafori di sincronizzazione (quelli che potrebbero bloccarti in attesa di una condizione) e **dopo** la `P()` sul mutex.
- Un semaforo su cui si fa solo `P()` o solo `V()` è quasi sempre un errore.

3 Risolvere con i Monitor

La mutua esclusione sulle procedure `entry` è automatica. Il tuo compito è gestire la sincronizzazione interna con le variabili di condizione. Ricorda la politica **Signal-and-Urgent**: chi viene svegliato parte subito.

3.1 Pattern 1: Rendezvous (Incontro)

Due o più processi devono "incontrarsi" per poter procedere. Un tipo di processo attende l'altro.

3.1.1 meanblack / meanred (13 giugno 2023)

Problema: Scrivere un monitor con `meanblack(v)` e `meanred(v)`. Ogni chiamata deve sincronizzarsi con una dell'altro colore. Entrambe restituiscono la media dei due valori. I processi in attesa vengono serviti in ordine FIFO.

Pattern Risolutivo

Servono due contatori per i processi in attesa di ogni colore e due condition variable separate. Quando un processo arriva, controlla se c'è un "partner" in attesa. Se sì, lo sblocca e procede. Se no, si mette in attesa sulla sua condition variable.

```
monitor rb {
    // Stato
    int n_waiting_reds = 0;
    int n_waiting_blacks = 0;
    float red_value, black_value;
    condition ok2go_red, ok2go_black;

    // Funzione interna per evitare duplicazione di codice
    float do_rendezvous(float my_value, int *my_waiting_count, int *other_waiting_count,
                        float *my_value_storage, float *other_value_storage,
                        condition *my_cond, condition *other_cond) {

        (*my_waiting_count)++;
        if (*other_waiting_count > 0) {
            // C'e' un partner che aspetta!
            (*other_waiting_count)--;
            *my_value_storage = my_value;
            // Calcolo la media, la salvo nel risultato e sveglio l'altro
            float result = (*my_value_storage + *other_value_storage) / 2.0;
            other_cond.signal();
            return result;
        } else {
            // Nessun partner, devo aspettare
            *my_value_storage = my_value;
            my_cond.wait();
            // Quando mi sveglio, l'altro ha gia' calcolato la media
            return (*my_value_storage + *other_value_storage) / 2.0;
        }
    }

    float meanred(float v) {
        // La logica e' simmetrica, la incapsulo in una funzione helper
        return do_rendezvous(v, &n_waiting_reds, &n_waiting_blacks,
                             &red_value, &black_value, &ok2go_red, &ok2go_black);
    }

    float meanblack(float v) {
        // Stessa logica con parametri invertiti
        return do_rendezvous(v, &n_waiting_blacks, &n_waiting_reds,
                             &black_value, &red_value, &ok2go_black, &ok2go_red);
    }
}
```

3.2 Pattern 2: Sincronizzazione su Condizione Complessa

Un processo attende finché lo stato del monitor non soddisfa una certa condizione.

3.2.1 porto (20 luglio 2022)

Problema: Un monitor `porto` gestisce un molo. Una sola nave `attracca` e attende di essere riempita. Un solo `camion` alla volta `scarica`. La nave `salpa` solo quando la sua `capacita` è raggiunta.

Pattern Risolutivo

Le variabili di stato sono cruciali: `molo_libero`, `camion_park_libero`, `carico_attuale`, `capacita_nave`. Servono `condition` per la nave in attesa di salpare (`ok2salpa`), per la nave in attesa di attraccare (`ok2attracca`) e per i camion in attesa di scaricare (`ok2scarica`).

```
monitor porto {
    boolean molo_occupato = false;
    boolean postazione_scarico_occupata = false;
    int capacita_nave_corrente = 0;
    int carico_attuale = 0;

    condition nave_puo_attraccare, nave_puo_salpare, camion_puo_scaricare;

    void attracca(int capacita) {
        if (molo_occupato) {
            nave_puo_attraccare.wait();
        }
        molo_occupato = true;
        carico_attuale = 0;
        capacita_nave_corrente = capacita;
        // Sveglio eventuali camion in attesa di una nave
        camion_puo_scaricare.signal();
    }

    void salpa() {
        if (carico_attuale < capacita_nave_corrente) {
            nave_puo_salpare.wait();
        }
        molo_occupato = false;
        // Il molo è libero, sveglio una nave in attesa
        nave_puo_attraccare.signal();
    }

    void scarica(int quantita) {
        if (postazione_scarico_occupata || !molo_occupato) {
            camion_puo_scaricare.wait();
        }
        postazione_scarico_occupata = true;

        carico_attuale += quantita;

        if (carico_attuale >= capacita_nave_corrente) {
            nave_puo_salpare.signal();
        }

        postazione_scarico_occupata = false;
        // La postazione di scarico è libera, sveglio un altro camion
        camion_puo_scaricare.signal();
    }
}
```

Nota sul camion che scarica parzialmente: il testo del 20/07/2022 ha una complicazione extra. Se un camion ha più merce di quanta ne serva per riempire la nave, deve scaricare solo il necessario, attendere che la nave salpi e arrivi la successiva, e poi completare lo scarico. La soluzione completa richiederebbe un ciclo `while` dentro `scarica` e un'altra `condition` per far attendere il camion. La versione qui sopra è semplificata per mostrare il pattern di base.

3.3 Pattern 3: Coda di Condizioni (Selezione Selettiva)

A volte non basta svegliare "un" processo a caso, ma bisogna svegliare quello giusto. Questo si fa mantenendo una coda di `condition variable`, una per ogni processo in attesa.

3.3.1 choicesem (25 giugno 2024)

Problema: Implementare un semaforo `choicesem` con `P()` e `V(n)`. `V(n)` sblocca il processo in posizione $n+1$ nella coda di attesa (o il primo se ci sono al più n processi in attesa).

Pattern Risolutivo

La `V` non può fare una semplice `signal` su una `condition` condivisa. Deve poter scegliere chi svegliare. La soluzione è dare ad ogni processo che si blocca una sua `condition` personale e metterla in una coda.

```
monitor choicesem {
    int value = 1; // Inizializzato come un semaforo binario
    queue<condition> waiting_queue;

    void P(void) {
        if (value > 0) {
            value--;
        } else {
            condition my_turn = new condition();
            waiting_queue.enqueue(my_turn);
            my_turn.wait();
            // Quando mi sveglio, il mio turno è arrivato.
            // Non devo toccare 'value', la V lo ha gestito per me.
            delete my_turn;
        }
    }

    void V(int n) {
        if (waiting_queue.isEmpty()) {
            value++;
        } else {
            // C'è qualcuno in attesa. Devo scegliere chi svegliare.
            condition to_wake;
            int index_to_wake = n; // L'indice è n, non n+1, perché le code sono 0-based
            if (index_to_wake >= waiting_queue.size()) {
                // n è troppo grande, sveglio il primo (indice 0)
                to_wake = waiting_queue.dequeue_at(0);
            } else {
                // Sveglio quello all'indice n
                to_wake = waiting_queue.dequeue_at(index_to_wake);
            }
            to_wake.signal();
        }
```

```
    }
}
```

3.4 Decalogo per i Monitor

- Le operazioni sono `c.wait()` e `c.signal()`, senza parametri.
- **Non usare mai busy-wait** all'interno di un monitor: è un **deadlock** garantito. Il processo terrebbe il lock del monitor girando a vuoto, impedendo a chiunque altro di entrare per modificare la condizione che sta aspettando.
- Il costruttore non può contenere `wait` o `signal`.
- **Usa `if` e non `while` per la condizione di attesa**. Con la politica **Signal-and-Urgent**, quando un processo viene svegliato, riparte immediatamente. La condizione che lo ha sbloccato è garantita essere ancora vera. Un `while` non è sbagliato, ma è ridondante.
- È errato avere una `wait` come prima istruzione di ogni procedura `entry`. Significherebbe che il monitor si blocca sempre, senza mai fare nulla.

4 Risolvere con Message Passing

La sincronizzazione è un effetto collaterale dello scambio di messaggi. Non ci sono variabili condivise tra processi; lo stato è distribuito o centralizzato in un processo server.

4.1 Pattern 1: Sincronizzazione tramite Messaggio di "Completamento"

Spesso un'operazione asincrona deve diventare sincrona o semi-sincrona. Un modo per farlo è inviare un messaggio a sé stessi come "marcatore" per sapere quando si sono ricevuti tutti i messaggi precedenti.

4.1.1 nbl_receive (13 giugno 2023)

Problema: Implementare `nbl_receive(sender)` che restituisce **tutti** i messaggi già presenti nella mailbox dal mittente `sender`, senza attendere.

Pattern Risolutivo

Come faccio a sapere quanti messaggi leggere? Non posso saperlo a priori. L'idea è:

1. Inviare un messaggio speciale (`TAG`) a me stesso.
2. Iniziare un ciclo di ricezione.
3. Continuare a ricevere finché non trovo il mio `TAG`. A quel punto so che tutti i messaggi che erano in coda *prima* del `TAG` sono stati ricevuti e processati.

```
// Questo codice viene eseguito nel contesto del processo che chiama nbl_receive

list<msg_t> nbl_receive(pid_t sender) {
    list<msg_t> received_messages;

    // 1. Mando un "marcatore" a me stesso per delimitare la coda dei messaggi.
    asend(TAG, getpid());
```

```
    // 2. Ciclo di ricezione
    while (true) {
        msg_t msg = arecv(ANY); // Ricevo da chiunque
        pid_t source = get_sender_pid(msg); // Funzione di libreria fittizia

        if (source == getpid() && is_tag_message(msg)) {
            // 3. Ho trovato il mio marcatore. Ho finito.
            break;
        }

        // È un messaggio normale. Lo metto da parte in un buffer temporaneo
        // perché potrebbe non essere per la richiesta corrente.
        // Questo è il punto debole: serve un buffer locale per i messaggi "inattesi".
        temporary_buffer.add(source, msg);
    }

    // Ora che ho svuotato la mailbox fino al tag, estraggo i messaggi che mi
    ↪  interessano
    // dal buffer temporaneo.
    received_messages = temporary_buffer.get_all(sender);

    return received_messages;
}
```

4.2 Pattern 2: Implementare un Servizio Sincrono su uno Asincrono

Il classico "scambio" sincrono (rendezvous) implementato con primitive asincrone.

4.2.1 xchange (20 gennaio 2023)

Problema: Implementare `T xchange(T msg, pid_t dest)` usando message passing asincrono. Quando P chiama `xchange(m1, Q)` e Q chiama `xchange(m2, P)`, entrambi si sbloccano e P riceve `m2` mentre Q riceve `m1`.

Pattern Risolutivo

Questo è un rendezvous. Ogni processo deve sapere con chi sta cercando di comunicare. Un processo server (o un database condiviso accessibile tramite messaggi) è necessario per memorizzare le "offerte" di scambio.

```
// PROCESSO SERVER (DB_Manager)
// Stato: una mappa di "offerte di scambio"
// map<pid_t, {pid_t dest, T msg}> offers;

while(true) {
    <sender, request> = arecv(ANY); // Riceve una richiesta di scambio

    pid_t dest = request.dest;

    if (offers.contains(dest) && offers[dest].dest == sender) {
        // C'è un match! L'altro processo mi ha già contattato.
        // 1. Prendo l'offerta dell'altro processo.
        auto other_offer = offers.remove(dest);
```

```
        // 2. Invio i messaggi scambiati ad entrambi.
        asend(other_offer.msg, sender); // Mando a chi mi ha appena scritto il msg
        ↪ dell'altro
        asend(request.msg, dest);       // Mando al destinatario il msg di chi mi ha
        ↪ appena scritto
    } else {
        // Nessun match. Memorizzo l'offerta.
        offers.add(sender, {dest, request.msg});
    }
}

// FUNZIONE CLIENT (xchange)
T xchange(T msg, pid_t dest) {
    // 1. Invia la mia offerta al server
    asend({dest, msg}, DB_MANAGER_PID);
    // 2. Attendo la risposta del server (che arriverà solo quando ci sarà un match)
    T received_msg = arecv(DB_MANAGER_PID);
    return received_msg;
}
```

4.3 Decalogo per il Message Passing

- **No Variabili Globali Condivise**. Lo stato è locale a ogni processo, o centralizzato in un processo server.
- I servizi come `arecv` sono **FIFO** se non diversamente specificato.
- Un processo server (o "dispatcher", "manager") è quasi sempre la soluzione giusta quando più client devono interagire in modo coordinato o accedere a una risorsa logica condivisa.
- I messaggi devono contenere tutte le informazioni necessarie, incluso (spesso) il `PID` del mittente, perché `arecv(ANY)` perde questa informazione. Negli esami più recenti, la primitiva spesso restituisce una coppia (`sender`, `msg`). Leggi bene il testo.
- Per implementare un protocollo complesso, può essere necessario definire diversi "tipi" di messaggio (es. `RICHIESTA`, `RISPOSTA`, `TAG`) per permettere al ricevente di distinguere le varie fasi della comunicazione.