# Concorrenza: mutua esclusione e sincronizzazione

Traduzione e riorganizzazione delle slide originali

Corso: Sistemi di Calcolo 2
Docente: Riccardo Lazzeretti
Crediti speciali: Daniele Cono D'Elia, Leonardo Aniello, Roberto Baldoni

## Fonti

Queste note sono una traduzione fedele e una riorganizzazione delle slide *"Concurrency: mutual exclusion and synchronization"*, in gran parte tratte da *Operating Systems: Internals and Design Principles* (8a ed.) di William Stallings (cap. 5). Dove le slide contenevano figure, qui forniamo una descrizione testuale essenziale.

## Indice

# 1 Contesto

La progettazione di un sistema operativo riguarda la gestione di processi e thread in ambienti di multiprogrammazione, multiprocessore e distribuiti. Le applicazioni possono essere multiple, strutturate (estensione del design modulare) e lo stesso sistema operativo può essere implementato come insieme di processi o thread concorrenti.

# 2 Termini chiave della concorrenza

**Operazione atomica** Sequenza di una o più istruzioni che appare indivisibile: non è osservabile alcuno stato intermedio, o l'intera sequenza avviene, o non avviene affatto.

**Sezione critica** Porzione di codice che accede a risorse condivise e non deve essere eseguita contemporaneamente da più processi che accedono alle medesime risorse.

**Mutua esclusione** Condizione per cui, se un processo è in una sezione critica che accede a risorse condivise, nessun altro processo può entrare in una sezione critica che accede a quelle stesse risorse.

**Race condition** Situazione in cui processi/thread concorrenti leggono/scrivono una variabile condivisa e il risultato finale dipende dall'ordine di esecuzione relativo.

**Deadlock** Due o più processi non possono procedere poiché ciascuno attende un'azione dell'altro.

**Livelock** Due o più processi continuano a cambiare stato in risposta reciproca senza fare lavoro utile.

**Starvation** Un processo pronto viene trascurato indefinitamente dallo scheduler.

# 3 Concorrenza in multiprogrammazione

- L'output di ciascun processo deve essere indipendente dalla velocità di esecuzione altrui.
- Interleaving e overlapping sono entrambi esempi di elaborazione concorrente e introducono gli stessi problemi.
- Su un uniprocessore, la velocità relativa dei processi è imprevedibile (dipende da attività altrui, gestione delle interruzioni e politiche di scheduling).
- Condivisione di risorse globali: l'allocazione ottimale è ardua, e gli errori sono difficili da riprodurre.

# 4 Preoccupazioni del sistema operativo

Il kernel deve: tracciare i processi, allocare/deallocare risorse, proteggere dati e risorse fisiche tra processi, garantire che processi e output siano indipendenti dalla velocità di elaborazione.

# 5 Concorrenza e competizione per le risorse

Processi concorrenti entrano in conflitto quando competono per la stessa risorsa (dispositivi I/O, memoria, CPU, clock). Tre problemi fondamentali: mutua esclusione, deadlock, starvation.

# 6 Mutua esclusione: esempi e proprietà

## 6.1 Schema generale

Esempio con risorsa condivisa `Ra`:

Listing 1: Schema concettuale di sezione critica

```
/* Processo i */
void Pi(void) {
  while (true) {
    /* codice precedente */
    entercritical(Ra);
    /* sezione critica */
    exitcritical(Ra);
    /* codice seguente */
  }
}
```

## 6.2 Requisiti

- Deve essere garantita (nessuna violazione).
- Se un processo si ferma fuori dalla sezione critica, non deve interferire con altri.
- Nessun deadlock o starvation (un processo non deve essere negato sezione critica quando libera).
- Nessuna ipotesi su velocità relative né numero di processi.
- Permanenza finita nella sezione critica.

# 7 Supporto hardware alla mutua esclusione

## 7.1 Disabilitazione delle interruzioni (uniprocessore)

Garantisce mutua esclusione ma degrada l'efficienza e non funziona su multiprocessori.

## 7.2 Istruzione `compare_and_swap` (CAS)

Confronta un valore di memoria con un valore atteso e, se uguale, lo sostituisce con un nuovo valore, atomicamente (x86, IA-64, SPARC, IBM). Esempio di spinlock:

Listing 2: Mutua esclusione con CAS

```
const int n = /* numero di processi */;
int bolt;

void P(int i) {
  while (true) {
    while (compare_and_swap(&bolt, 0, 1) == 1)
      ; // busy wait
    /* sezione critica */
    bolt = 0;
    /* resto */
  }
}

int main(void) {
  bolt = 0;
  parbegin(P(1), P(2) /*, ... , P(n)*/);
}
```

## 7.3 Istruzione di scambio (`exchange/xchg`)

Scambia contenuto di un registro con una locazione di memoria (Pentium/Itanium). Schema analogo con variabile `bolt` e chiave locale `key`.

## 7.4 Vantaggi e svantaggi delle istruzioni speciali

**Pro**: semplici, verificabili, applicabili a più CPU, supportano più sezioni critiche (una variabile per sezione).
**Contro**: busy waiting, possibilità di starvation, possibili deadlock (priorità), dipendenza dall'architettura.

# 8 Meccanismi comuni di concorrenza

**Semaforo** Intero con tre operazioni atomiche: inizializza, decrementa (può bloccare), incrementa (può sbloccare).

**Semaforo binario** Valori 0/1.

**Mutex** Simile a semaforo binario, ma chi blocca deve essere chi sblocca.

**Variabile di condizione** Per bloccare un thread finché una condizione non diventa vera.

**Monitor** Costrutto di linguaggio che incapsula dati, procedure e inizializzazione; un solo processo attivo nel monitor; code d'attesa interne.

**Event flags** Parola di memoria con bit che rappresentano eventi (AND/OR attese).

**Mailbox/Messaggi** Scambio informazioni e sincronizzazione tra processi.

**Spinlock** Mutua esclusione con attesa attiva.

# 9 Semafori: definizioni ed implementazione

## 9.1 Primitivi concettuali

Non è possibile ispezionare lo stato se non tramite le tre operazioni; non si può sapere ex-ante se un `semWait` bloccherà; su uniprocessore non si può sapere chi procederà dopo un `semWait`.

## 9.2 Semaforo binario: pseudocodice

Listing 3: Semaforo binario: definizione

```
struct binary_semaphore {
  enum { zero, one } value;
  queueType queue;
};

void semWaitB(binary_semaphore *s) {
  if (s->value == one) s->value = zero;
  else { enqueue(s->queue, this_process); block(); }
}

void semSignalB(binary_semaphore *s) {
  if (empty(s->queue)) s->value = one;
  else { P = dequeue(s->queue); make_ready(P); }
}
```

**Semafori forti**: FIFO sulla coda. **Semafori deboli**: ordine non specificato.

## 9.3 Accesso protetto con semafori

Listing 4: Accesso a dati condivisi con semaforo

```
semaphore lock = 1;

void processA(void) {
  /* ... */
  semWait(lock);
  /* sezione critica */
  semSignal(lock);
  /* ... */
}
```

### 9.4 Implementazione dei semafori

Le operazioni devono essere atomiche (il loro stesso accesso è un problema di mutua esclusione). Possibili implementazioni:

1. Basata su CAS/spinlock con campo `flag` che protegge `count` e la coda.
2. Disabilitando le interruzioni nelle sezioni critiche dei primitivi (*kernel* o firmware).

## 10 Semafori POSIX (C)

Listing 5: API POSIX dei semafori

```c
int sem_init(sem_t *sem, int pshared, unsigned value);
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_destroy(sem_t *sem);
/* pshared: 0 tra thread, 1 tra processi; value: valore iniziale; return 0/-1 e errno
    */
```

Listing 6: Esempio di semaforo POSIX

```c
#include <pthread.h>
#include <semaphore.h>

sem_t sem;

void* worker(void* arg){
  sem_wait(&sem);
  /* sezione critica */
  sem_post(&sem);
  return NULL;
}

int main(void){
  sem_init(&sem, 0, 1);
  pthread_t t1, t2;
  pthread_create(&t1, NULL, worker, NULL);
  pthread_create(&t2, NULL, worker, NULL);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  sem_destroy(&sem);
  return 0;
}
```

## 11 Semafori in Java e Python

### 11.1 Java

Listing 7: Semafori in Java

```java
// new Semaphore(int permits)
// new Semaphore(int permits, boolean fair) // true => forte (FIFO)
import java.util.concurrent.*;
class MyThread extends Thread {
  private final Semaphore sem;
  public MyThread(Semaphore sem, String name){ super(name); this.sem = sem; }
```

```
  public void run(){
    try {
      sem.acquire(); // wait
      // sezione critica
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
    } finally {
      sem.release(); // signal
    }
  }
}
```

### 11.2 Python

Listing 8: Semafori in Python

```
import threading
sem = threading.Semaphore()  # default=1
def fun():
    while True:
        sem.acquire()   # wait
        # sezione critica
        sem.release()   # signal

t1 = threading.Thread(target=fun); t1.start()
t2 = threading.Thread(target=fun); t2.start()
```

## 12 Monitor e variabili di condizione

### 12.1 Caratteristiche

Solo un processo alla volta è attivo nel monitor; i dati locali sono accessibili solo dalle procedure del monitor; sincronizzazione tramite variabili di condizione con primitive `cwait(c)` e `csignal(c)`.

### 12.2 Struttura concettuale di un monitor

*Descrizione testuale della figura*: il monitor contiene codice di inizializzazione, $k$ procedure pubbliche critiche, dati locali, e un'area di attesa con code urgenti e code per ogni condizione $c_i$. Le chiamate a `cwait` sospendono nel monitor; `csignal` risveglia un processo in attesa su quella condizione, con politica di precedenza definita (p.es. coda urgente).

## 13 Passaggio di messaggi

### 13.1 Primitive e caratteristiche

Coppia di primitive: `send(dest, msg)` e `receive(src, msg)`. Dimensioni e formato messaggio (header con sorgente/destinazione/lunghezza/controllo; body con contenuto).

### 13.2 Scelte progettuali

- **Sincronizzazione**: invio/ ricezione bloccanti o non bloccanti (quattro combinazioni; il *rendezvous* blocca entrambi).

- **Indirizzamento**: diretto (sorgente/destinazione espliciti o impliciti) o indiretto (mailbox/porte).
- **Accodamento**: FIFO, priorità.
- **Formato**: lunghezza fissa o variabile.

### 13.3 Indirizzamento indiretto (mailbox)

*Descrizione testuale della figura*: configurazioni uno-a-uno, molti-a-uno, uno-a-molti, molti-a-molti, con un contenitore *mailbox* (o porta) che funge da coda di messaggi condivisa tra mittenti e destinatari.

## 14 Problema Produttore/Consumatore

### 14.1 Enunciato generale

Uno o più produttori generano dati inserendoli in un buffer; uno o più consumatori prelevano gli elementi uno alla volta. Accesso esclusivo al buffer; il produttore non deve inserire in buffer pieno e il consumatore non deve prelevare da buffer vuoto.

### 14.2 Buffer infinito: soluzioni errate (con semafori binari)

Mostriamo due schemi errati (come nelle slide) dove, per via di deschedulazione nel punto sbagliato e test non protetti, si può arrivare a consumare elementi inesistenti (es. $n$ che scende a $-1$). *Descrizione testuale delle tabelle*: sequenza di passi dove `s`, `n`, `delay` assumono valori che portano alla condizione errata.

### 14.3 Possibile correzione (solo semafori binari)

Usare una copia locale $m = n$ letta in sezione critica, in modo da testare correttamente fuori dalla sezione critica senza condizioni di gara.

### 14.4 Buffer infinito: semafori generali

Listing 9: Produttore/Consumatore con semafori generali

```
semaphore n = 0;   // elementi disponibili
semaphore s = 1;   // mutua esclusione sul buffer

void producer(void){
  while (true){
    produce();
    semWait(s);
    append();
    semSignal(s);
    semSignal(n);
  }
}

void consumer(void){
  while (true){
    semWait(n);
    semWait(s);
    take();
    semSignal(s);
```

```
    consume();
  }
}
```

### 14.5 Buffer circolare finito

*Descrizione testuale della figura*: buffer circolare di dimensione $N$ con indici `in` e `out`. Variante (a) con buffer pieno/vuoto distinguibili tramite conteggio; variante (b) con $N-1$ elementi utili per distinguere pieno da vuoto.

### 14.6 Buffer limitato con Monitor

**Idea**: incapsulare il buffer e la sincronizzazione nel monitor. Esempio concettuale:

Listing 10: Monitor per bounded buffer (pseudocodice)

```
monitor BoundedBuffer {
  Item buffer[N];
  int nextin = 0, nextout = 0, count = 0;
  condition not_full, not_empty;

  procedure append(Item x) {
    while (count == N) cwait(not_full);
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;
    csignal(not_empty);
  }

  procedure take(Item *x) {
    while (count == 0) cwait(not_empty);
    *x = buffer[nextout];
    nextout = (nextout + 1) % N;
    count--;
    csignal(not_full);
  }
}
```

*Nota*: l'algoritmo supporta anche molteplici consumatori/produttori.

### 14.7 Soluzione con messaggi (bounded buffer)

*Descrizione testuale*: due mailbox, `mayproduce` inizialmente piena (con $N$ *token*) e `mayconsume` inizialmente vuota; il produttore deve ricevere un token da `mayproduce` per produrre e poi inviare un elemento su `mayconsume`; il consumatore fa l'opposto. Ciò serializza correttamente gli accessi e rispetta i vincoli di capacità.

## 15 Problema Lettori/Scrittori

Area dati condivisa con lettori (sola lettura) e scrittori (scrittura esclusiva). Requisiti: lettori multipli in parallelo; un solo scrittore alla volta; se uno scrittore scrive, nessun lettore legge.

### 15.1 Schema con semafori

Scrittore: acquisisce lock esclusivo (semaforo binario `wsem`).
Lettore: usa semaforo binario `x` per aggiornare `readcount`; se `readcount==1` allora blocca `wsem`;

a fine lettura decrementa `readcount` e se torna a 0 rilascia `wsem`. *Attenzione*: possibile starvation degli scrittori; occorrono estensioni eque.

## 16 Note conclusive

I meccanismi di sincronizzazione vanno scelti valutando correttezza (assenza di race/deadlock/-starvation), prestazioni (evitare busy-wait non necessario), portabilità e supporto del linguaggio/piattaforma.