# Deadlock e Starvation

Prevenzione, evitamento, rilevamento e recupero

Corso: Sistemi di Calcolo 2
Docente: Riccardo Lazzeretti
Crediti speciali: adattato da W. Stallings, *Operating Systems: Internals and Design Principles* (cap.

## Fonti

Queste note sono una traduzione e riorganizzazione delle slide *"Concurrency: Deadlock and Starvation Problems"* con ulteriori esempi e pseudocodice esplicativo. Le figure presenti nelle slide originali sono descritte testualmente dove utile.

## Indice


**1 Cos'è un deadlock** **2**

**2 Esempi intuitivi** **2**
2.1 Deadlock potenziale e reale su griglia 2x2 . . . 2
2.2 Risorse riusabili e consumabili . . . 2
2.3 Esempio: richiesta di memoria . . . 2
2.4 Esempio: risorse consumabili . . . 2

**3 Le quattro condizioni di Coffman** **2**

**4 Grafi di allocazione delle risorse (RAG)** **3**

**5 Strategie generali** **3**

**6 Prevenzione del deadlock** **3**
6.1 Mutua esclusione . . . 3
6.2 Hold-and-wait . . . 3
6.3 No preemption . . . 3
6.4 Attesa circolare . . . 3

**7 Evitamento del deadlock** **4**
7.1 Idea di base . . . 4
7.2 Negazione di avvio (process initiation denial) . . . 4
7.3 Negazione di allocazione (resource allocation denial) . . . 4
7.4 Algoritmo del banchiere (sketch) . . . 4

**8 Stato sicuro: esempio concettuale** **4**

**9 Rilevamento del deadlock** **4**
9.1 Quando controllare . . . 4
9.2 Schema di rilevamento (marcatura iterativa) . . . 4

**10 Recupero dal deadlock** **4**

**11 Il problema dei Filosofi a Cena** **5**
11.1 Specifiche . . . . . 5
11.2 Soluzioni . . . . . 5

**12 Approcci software alla mutua esclusione** **5**
12.1 Assunzioni . . . . . 5
12.2 Algoritmo di Dekker (2 processi) . . . . . 5
12.3 Generalizzazione di Dijkstra ($N$ processi) . . . . . 6

**13 Considerazioni finali** **6**

# 1 Cos'è un deadlock

**Deadlock**: blocco permanente di un insieme di processi che competono per risorse o che comunicano tra loro. L'insieme è in *deadlock* quando ogni processo è in attesa di un evento che solo un altro processo (anch'esso bloccato) può generare. Non esiste una soluzione efficiente *ex post* in generale; occorre prevenirlo, evitarlo o rilevarlo e recuperare.

**Esempio reale (parafrasi)** Due treni si incontrano a un incrocio: entrambi devono fermarsi e nessuno può ripartire finché l'altro non è passato. Situazione di stallo.

# 2 Esempi intuitivi

## 2.1 Deadlock potenziale e reale su griglia 2x2

*Descrizione testuale*: quattro processi A, B, C, D vogliono ciascuno due *quadranti* (risorse) della griglia: A vuole (A,B), B vuole (B,C), C vuole (C,D), D vuole (D,A). Se tutti acquisiscono la prima risorsa e poi attendono la seconda, si forma un **ciclo di attesa** e quindi deadlock effettivo: ogni processo è in HALT finché la risorsa successiva non si libera, ma nessuno può progredire.

## 2.2 Risorse riusabili e consumabili

**Riusabili**: usate da un solo processo alla volta senza essere consumate (CPU, canali I/O, memoria, dispositivi, file, database, semafori).
**Consumabili**: vengono create/distrutte con l'uso (interrupt, segnali, messaggi, dati in buffer I/O).

## 2.3 Esempio: richiesta di memoria

Spazio libero 200 KB. Sequenza: $P_1$ chiede 80 KB; $P_2$ chiede 70 KB (restano 50); se $P_1$ chiede altri 60 e $P_2$ altri 80 si ottiene deadlock potenziale: nessuno dei due può essere soddisfatto e nessuno rilascia memoria.

## 2.4 Esempio: risorse consumabili

Due processi tentano di `receive` un messaggio l'uno dall'altro prima di `send`: con `receive` bloccante, entrambi restano in attesa per sempre $\Rightarrow$ deadlock.

# 3 Le quattro condizioni di Coffman

Affinché si verifichi deadlock devono valere simultaneamente:

1. **Mutua esclusione**: una risorsa può essere usata da un solo processo alla volta.

2. **Hold-and-wait**: un processo può trattenere risorse già allocate mentre ne attende altre.
3. **Assenza di preemption**: le risorse non possono essere sottratte forzatamente a un processo che le detiene.
4. **Attesa circolare**: esiste una catena chiusa di processi in cui ognuno attende una risorsa detenuta dal successivo.

Le prime tre sono *necessarie ma non sufficienti*; la quarta completa le condizioni per il deadlock.

# 4 Grafi di allocazione delle risorse (RAG)

*Descrizione testuale*: grafo bipartito con nodi processo $P_i$ e risorsa $R_j$; archi di *richiesta* ($P_i \to R_j$) e di *assegnazione* ($R_j \to P_i$). Un ciclo nel grafo è condizione necessaria per il deadlock (e sufficiente se ogni tipo di risorsa ha un solo esemplare).

# 5 Strategie generali

- **Prevenzione**: progettare il sistema in modo da escludere almeno una delle condizioni necessarie.
- **Evitamento**: decidere dinamicamente se concedere una richiesta in base allo stato corrente per restare in uno *stato sicuro*.
- **Rilevamento & recupero**: concedere le richieste e rilevare periodicamente il deadlock per poi recuperare.

# 6 Prevenzione del deadlock

## 6.1 Mutua esclusione

Se necessaria per correttezza (p.es. file in scrittura), non si può eliminare; talvolta si può *rilassare* (letture concorrenti sullo stesso file).

## 6.2 Hold-and-wait

Richiedere **tutte** le risorse in un'unica operazione e bloccare finché non sono disponibili; oppure rilasciare quelle detenute e riprovare. *Contro*: inefficienza, tempi d'attesa lunghi, scarsa utilità delle risorse trattenute.

## 6.3 No preemption

Consentire preemption di alcune risorse: se una richiesta non può essere soddisfatta, il processo rilascia quelle detenute (o si preempta un altro processo con priorità più bassa). Valido solo per risorse con stato salvabile/ripristinabile (CPU).

## 6.4 Attesa circolare

Imporre un **ordinamento totale** dei tipi di risorsa $R_1 \prec R_2 \prec \dots$ e vincolare i processi a richiedere risorse solo in ordine crescente. *Contro*: simile a hold-and-wait, può risultare poco flessibile.

# 7 Evitamento del deadlock

## 7.1 Idea di base

Concedere una richiesta solo se il sistema rimane in **stato sicuro**: esiste una sequenza di completamento che consente a tutti i processi di terminare senza deadlock.

## 7.2 Negazione di avvio (process initiation denial)

Avviare un nuovo processo solo se somma delle sue *dichiarazioni massime* + richieste massime dei processi attivi è $\leq$ alle risorse disponibili. Scelta conservativa.

## 7.3 Negazione di allocazione (resource allocation denial)

Non concedere richieste incrementali che porterebbero a uno stato non sicuro. Questo approccio è alla base dell'**algoritmo del banchiere**.

## 7.4 Algoritmo del banchiere (sketch)

- Ogni processo dichiara a priori il *fabbisogno massimo* per ciascun tipo di risorsa.
- Per ogni richiesta, si simula l'allocazione e si testa la presenza di una sequenza sicura; se esiste, si concede.
- **Assunzioni**: numero fisso di risorse; processi indipendenti e senza vincoli di sincronizzazione; nessun processo termina trattenendo risorse.

# 8 Stato sicuro: esempio concettuale

*Descrizione testuale*: sistema con 4 processi e 3 tipi di risorsa; dopo alcune allocazioni si verifica che esiste una sequenza (p.es. $P_3, P_4, P_1, P_2$) in cui ogni processo può ottenere le risorse residue, terminare e rilasciare: lo stato iniziale è quindi **sicuro**.

# 9 Rilevamento del deadlock

## 9.1 Quando controllare

Ad ogni richiesta (rilevamento precoce ma costoso) o periodicamente (meno overhead, rischio di accrescere l'insieme bloccato).

## 9.2 Schema di rilevamento (marcatura iterativa)

1. Marca i processi che non detengono risorse.
2. Se esistono risorse sufficienti per far terminare un processo marcabile, marchialo e *rilascia* virtualmente le risorse.
3. Ripeti finché possibile. Se alla fine alcuni processi restano non marcati $\Rightarrow$ **deadlock**.

# 10 Recupero dal deadlock

- **Abortare** tutti i processi in deadlock (semplice ma drastico).
- **Rollback** a un checkpoint precedente e riavvio.

- **Aborti successivi** finché il deadlock scompare (scegliere chi abortire in base a costo/avanzamento/priorità).
- **Preemption delle risorse** con meccanismi di rollback.

# 11 Il problema dei Filosofi a Cena

## 11.1 Specifiche

- **Mutua esclusione**: due filosofi non possono usare la stessa forchetta contemporaneamente.
- **Assenza di deadlock e starvation**.

## 11.2 Soluzioni

- **Asimmetria**: filosofi dispari prendono prima la forchetta sinistra, pari prima la destra (rompe l'attesa circolare).
- **Arbitro** (cameriere): concede fino a $n-1$ permessi contemporanei.
- **Ordinamento delle risorse**: forchette numerate, richiederle in ordine crescente.
- **Token** o **gerarchia di priorità**: previene anche starvation.

# 12 Approcci software alla mutua esclusione

## 12.1 Assunzioni

Processi che comunicano via memoria centrale (su uni/multi-processore); lettura/scrittura sulla stessa locazione serializzata da un arbitro di memoria; nessun supporto diretto da OS/hardware/linguaggio oltre all'atomicità delle letture/scritture singole.

## 12.2 Algoritmo di Dekker (2 processi)

Listing 1: Dekker: soluzione corretta

```
/* globale */
volatile int turn = 0;
volatile int flag[2] = {0, 0};
/* lato P0: me=0, other=1 (scambia per P1) */
int me = 0, other = 1;
while (1) {
  flag[me] = 1;                  // voglio entrare
  while (flag[other]) {          // se anche l'altro vuole
    if (turn == other) {         // ed e' il suo turno
      flag[me] = 0;              // mi tiro indietro
      while (turn == other) ;    // attendo il turno
      flag[me] = 1;              // riprovo
    }
  }
  /* sezione critica */
  turn = other;                  // cedo il turno
  flag[me] = 0;                  // non voglio piu'
  /* sezione non critica */
}
```

**Proprietà**: mutua esclusione, assenza di deadlock, assenza di starvation.

### 12.3 Generalizzazione di Dijkstra ($N$ processi)

Listing 2: Protocollo di Dijkstra (sketch)

```
/* globale */
bool interested[N] = {false};
bool passed[N] = {false};
int k = 0;

/* locale: i = ID processo */
while (1) {
  /* fase di interesse */
  interested[i] = true;
  while (k != i) {
    passed[i] = false;
    if (!interested[k]) k = i;     // arbitraggio
  }
  /* fase uno superata */
  passed[i] = true;
  for (int j = 0; j < N; ++j) {
    if (j == i) continue;
    if (passed[j]) goto retry;     // conflitto: riparti
  }
  /* sezione critica */
  passed[i] = false; interested[i] = false;
  continue;
retry: ; /* torna a while(k!=i) */
}
```

**Nota**: l'algoritmo garantisce ME (mutua esclusione), ND (no deadlock) e NS (no starvation); NS $\Rightarrow$ ND.

## 13 Considerazioni finali

Le scelte tra prevenzione, evitamento, rilevamento e recupero dipendono da carico, costo delle risorse, prevedibilità dei fabbisogni e requisiti di *liveness*. L'uso di politiche eque e di checkpoint può ridurre la probabilità e l'impatto dei deadlock.