Deadlocks

Deadlock

Situazione permanente in cui una serie di processi competono per delle risorse di sistema o per comunicare tra loro; in questa situazione si bloccano (stallo)

→ un insieme di processi è in deadlock quando ogni processo nell'insieme è bloccato mentre aspetta un evento che può essere sbloccato solo da un altro processo bloccato nell'insieme

N.B. non esistono soluzioni efficienti

Il deadlock si può verificare su più tipi di risorse:

- risorse riutilizzabili → risorse utilizzabili da un processo alla volta (non cancellate alla fine)

Process P

Step	Action
p_0	Request (D)
p_1	Lock (D)
p_2	Request (T)
p_3	Lock (T)
p_4	Perform function
p_5	Unlock (D)
p_6	Unlock (T)

Process Q

Step	Action
q_0	Request (T)
q_1	Lock (T)
q_2	Request (D)
q_3	Lock (D)
q_4	Perform function
q_5	Unlock (T)
q_6	Unlock (D)

2 risorse: T, D

2 processi: P, Q

arrivati a p_1 e q_1 in parallelo i due processi si bloccano perché la risorsa è assegnata all'altro → soluzione: scambio q_0 q_1 con q_2 q_3

- risorse consumabili → risorse che possono essere create e distrutte

P1	P2
...	...
Receive (P2);	Receive (P1);
...	...
Send (P2, M1);	Send (P1, M2);

2 processi: P1, P2

Ogni processo per andare avanti ha bisogno dell'altro → ho deadlock se la receive è bloccante

Condizioni necessarie per avere deadlock:

- Mutua esclusione → solo un processo alla volta può accedere alla risorsa
- Hold&wait → un processo mantiene una risorsa allocata mentre ne aspetta delle altre (ma la prima non la rilascio)
- No preemption → non posso rimuovere forzatamente una risorsa da un processo che la mantiene

Condizione sufficiente: coda circolare → creo una catena di n processi in cui ognuno tiene una risorsa che serve al successivo e aspetta una risorsa dal precedente

3 approcci con cui il sistema operativo può gestire i deadlock:

- **Prevenire deadlock** → adotto una politica che elimina una delle 4 condizioni; molto conservativa; 2 metodi possibili:
  1. Indiretto ⇨ prevenire le condizioni necessarie
     i. Mutua esclusione: se l'accesso a una risorsa richiede mutua esclusione allora esso deve essere controllato dal sistema operativo
     ii. Hold&wait: il processo deve richiedere tutte le risorse necessarie e il sistema operativo lo blocca finché tutte le risorse necessarie non sono disponibili simultaneamente; è un metodo inefficiente perché un processo può aspettare molto prima che le risorse siano disponibili oppure mantenerle senza nemmeno usarle; inoltre, il processo potrebbe non sapere a priori di quali risorse avrà bisogno (es. if nel codice)
     iii. No pre-emption: se un processo detiene alcune risorse e gli è impedito di fare altre richieste, il processo dovrà rilasciare le risorse e richiederle nuovamente dopo oppure il sistema operativo toglierà le risorse ad un altro processo che detiene le risorse richieste (il sistema operativo potrà scegliere in base alla priorità); può essere attuato solo se è possibile salvare lo stato delle risorse e ripristinarlo successivamente (es. non è possibile fermare una stampante a metà stampa)
  2. Diretto ⇨ prevenire la condizione sufficiente (coda circolare); bisogna definire un ordinamento lineare per le risorse → un processo non può richiedere una risorsa che viene prima di una risorsa già richiesta in precedenza (non posso richiedere R1 dopo aver richiesto R2); ha lo stesso problema della hold&wait (non sempre so a priori di quali risorse ho bisogno)
- **Evitare deadlock** → durante l'esecuzione, quando un processo richiede una risorsa, il sistema operativo deve valutare se fornirgliela o meno; permetto maggiore concorrenza rispetto alla prevenzione dei deadlock (il sistema operativo blocca i processi solo quando si sta per verificare il deadlock)

  ➔ è richiesta una conoscenza a priori delle richieste future dei processi

  1. Process Initiation Denial → un processo viene ritardato se il suo avvio può portare a un deadlock; il sistema operativo permette l'esecuzione del nuovo processo solo se la somma delle risorse richieste dai processi in esecuzione e le risorse richieste dal nuovo processo è minore delle risorse totali disponibili
  2. Resource Denial Allocation → non garantire una risorsa se essa può portare a un deadlock

Uno stato è **safe** se esiste una sequenza di allocazione di risorse tale che non porta a un deadlock; posso determinare se uno stato è safe tramite un algoritmo:

```
struct state {
      int resource[m];
      int available[m];
      int claim[n][m];
      int alloc[n][m];
}
```

```
if (alloc [i,*] + request [*] > claim [i,*])
    < error >;                                  /* total request > claim*/
else if (request [*] > available [*])
    < suspend process >;
else {                                          /* simulate alloc */
    < define newstate by:
    alloc [i,*] = alloc [i,*] + request [*];
    available [*] = available [*] - request [*] >;
}
if (safe (newstate))
    < carry out allocation >;
else {
    < restore original state >;
    < suspend process >;
}
```

```
boolean safe (state S) {
   int currentavail[m];
   process rest[<number of processes>];
   currentavail = available;
   rest = {all processes};
   possible = true;
   while (possible) {
      <find a process Pk in rest such that
         claim [k,*] - alloc [k,*] <= currentavail;>
      if (found) {                          /* simulate execution of Pk */
         currentavail = currentavail + alloc [k,*];
         rest = rest - {Pk};
      }
      else possible = false;
   }
   return (rest == null);
}
```

stato del sistema → risorse totali, risorse disponibili, una matrice dove ogni processo in esecuzione dice di quante risorse avrà bisogno e una matrice che descrive quante risorse sono allocate

innanzitutto, se la richiesta di un processo + il numero delle risorse già allocate è maggiore di quelle che aveva dichiarato che avrebbe chiesto all'inizio ci sarà un errore; se le risorse che richiede il processo sono maggiori delle risorse disponibili allora il processo sarà sospeso; altrimenti definisco un nuovo stato in cui nella matrice di allocazione aggiungerò le richieste e nell'array delle risorse disponibili toglierò le richieste → successivamente controllo se il nuovo stato è safe; se lo è continuo con l'allocazione, altrimenti torno allo stato originale e sospendo il processo

controllo che lo stato sia safe tramite l'algoritmo "del banchiere"

gestire il while (possible) significa avere un grande overhead per il sistema operativo dato che ci sono tante risorse e tanti processi da gestire

3. Ci sono alcune restrizioni per evitare deadlock:
    1. Il numero massimo di risorse richieste per ogni processo deve essere conosciuto a priori
    2. I processi considerati devono essere indipendenti e senza richiesta di sincronizzazione
    3. Il numero di risorse da allocare deve essere finito
    4. Nessun processo può terminare mentre mantiene le risorse

- **Trovare deadlock** → capire se 2+ processi sono in deadlock; il sistema operativo da a tutti le risorse necessarie (non conservativa) e analizza tutti i processi, le risorse e come vengono utilizzate
  ➔ controlla che non ci siano deadlock tanto frequentemente tanto quanta è la probabilità che si creino deadlock (algoritmo semplice e scopre velocemente se c'è deadlock ma ha costo su CPU)

es. P4 non ha risorse quindi sicuramente non sarà in deadlock

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix A

R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

R1	R2	R3	R4	R5
0	0	0	0	1

Allocation vector

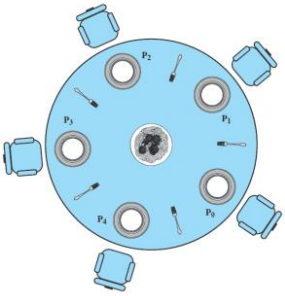
Se viene data a P3 la risorsa necessaria (R5) termina e quindi sicuramente non va in deadlock

Anche se P3 rilascia le sue risorse né P1 né P2 possono terminare → DEADLOCK

➔ cosa si può fare sapendo che 2+ processi sono in deadlock?

1. tutti i processi in deadlock vengono uccisi (rilascio risorse)
2. si tiene traccia dei checkpoint (periodicamente) e quando avviene un deadlock si ritorna allo stato del checkpoint precedente in cui non c'era deadlock (ma s potrebbe tornare nella situazione di deadlock)
3. i processi in deadlock vengono uccisi uno alla volta controllando che non siano più in deadlock

Problema della cena dei filosofi



"Cinque filosofi siedono ad una tavola rotonda con un piatto di spaghetti davanti e una forchetta a sinistra. Ci sono dunque cinque filosofi, cinque piatti di spaghetti e cinque forchette.

Si immagini che la vita di un filosofo consista di periodi alterni di mangiare e pensare, e che ciascun filosofo abbia bisogno di due forchette per mangiare, ma che le forchette vengano prese una per volta. Dopo essere riuscito a prendere due forchette il filosofo mangia per un po', poi lascia le forchette e ricomincia a pensare"

2 filosofi non posso usare la stessa forchetta nello stesso momento (mutua esclusione); nessun filosofo deve morire di fame (evitare deadlock e starvation)

➔ uso N semafori; questa soluzione può portare al deadlock

```
semaphore fork [5] = {1};
void philosopher (int i)
{
     while (true) {
          think();
          wait (fork[i]);
          wait (fork [(i+1) mod 5]);
          eat();
          signal(fork [(i+1) mod 5]);
          signal(fork[i]);
     }
}
```

➔ "butto una sedia" → se ci sono già 4 filosofi non può entrarne un altro; finché ho 5 forchette ci possono stare 4 filosofi

```
semaphore fork[5] = {1};
semaphore room = {4};
void philosopher (int i)
{
     while (true) {
          think();
          wait (room);
          wait (fork[i]);
          wait (fork [(i+1) mod 5]);
          eat();
          signal (fork [(i+1) mod 5]);
          signal (fork[i]);
          signal (room);
     }
}
```

Approccio software alla mutua esclusione

La mutua esclusione può essere implementata anche a livello puramente software

Per questi algoritmi assumo che:

- la mutua esclusione è garantita a livello di accesso a memoria
- read/write per la stessa posizione in memoria sono serializzate da un gestore della memoria (es. 2 write chiamate nello stesso momento non vengono fatte nello stesso momento ma non posso sapere quale venga fatta prima o dopo)
- non ho supporto tramite sistema operativo, hardware o linguaggio di programmazione per garantire mutua esclusione

Algoritmo di Dekker

Dekker ha progettato un algoritmo di mutua esclusione per 2 processi

procedo per tentativi:

- tentativo 1 → un processo entra in sezione critica e successivamente passa il turno all'altro; deve aspettare che il secondo processo entri ed esca dalla sezione critica; questo può causare problemi se il primo processo entra ed esce dalla sezione critica mentre il secondo lo fa meno frequentemente

```
/* global */
int turn = 0;

// assignments valid for P0 (flip for P1)
int me = 0, other = 1;
while (true) {
   /*NCS*/
   while (turn != me)
       /* busy wait */ ;
   /* CS */
   turn = other;
}
```

- tentativo 2 → un processo cerca di capire se l'altro è in sezione critica attraverso i flag; in una situazione di parallelismo perfetto però due processi potrebbero essere entrambi fuori dalla sezione critica e controllare il flag dell'altro entrando quindi entrambi in sezione critica

```
/* global */
boolean flag[2] = {false, false};

// assignments valid for P0 (flip for P1)
int me = 0, other = 1;

while (true) {
   /*NCS*/
   while (flag[other])
       /* busy wait */ ;
   flag[me] = true;
   /* CS */
   flag[me] = false;
}
```

- tentativo 3 → un processo segnala prima che vuole entrare in sezione critica e poi controlla lo stato dell'altro (garantisce mutua esclusione); in questo caso possono verificare deadlock perché posso avere la coda circolare

```
// assignments valid for P0 (flip for P1)
int me = 0, other = 1;

while (true) {
   /*NCS*/
   flag[me] = true;
   while (flag[other])
       /* busy wait */ ;
   /* CS */
   flag[me] = false;
}
```

es. processo1

```
flag[me]=true;
*controllo l'altro*
flag[me]=false;
*esco da busy waiting*
flag[me]=true;
```

processo2

```
flag[me]=true;
*controllo l'altro*
flag[me]=false;
*esco da busy waiting*
flag[me]=true;
```

- tentativo 4 → se un processo si accorge che anche l'altro processo vuole entrare in sezione critica, lascia entrare l'altro mettendo il flag a false; si può generare comunque un livelock se c'è perfetto parallelismo

```
// assignments valid for P0 (flip for P1)
int me = 0, other = 1;

while (true) {
   /*NCS*/
   flag[me] = true;
   while (flag[other]) {
       flag[me] = false;
       /* delay */
       flag[me] = true;
   }
   /* CS */
   flag[me] = false;
}
```

- Soluzione finale → vengono reintrodotti i turni; la precedenza è di chi non è entrato in sezione critica da più tempo

```
int me = 0, other = 1; // P0 (flip for P1)

while (true) {
   /*NCS*/
   flag[me] = true;
   while (flag[other]) {
      if (turn == other) {
         flag[me] = false;
         while (turn == other) /* busy wait */ ;
         flag[me] = true;
      }
   }
   /* CS */
   turn = other;
   flag[me] = false;
}
```

➔ questo sistema funziona per 2 processi ma non per di più

"io voglio entrare, se tu vuoi entrare ed è il tuo turno io non voglio entrare più, se è il tuo turno io aspetto, poi voglio entrare…faccio quello che devo…dopo entri tu, io non voglio più"

Algoritmo di Dijkstra

Generalizzazione di Dekker per n entità

3 variabili globali:

1. array interested → una cella per ogni processo; il processo i indica che è interessato ad entrare in sezione critica mettendo true la cella i-esima
2. array passed → una cella per ogni processo; il processo i indica che è "quasi entrato" in sezione critica mettendo true la cella i-esima
3. int k → indica il turno

```
while (true) {
   /*NCS*/
1.  interested[i] = true
2.  while (k != i) {
3.    passed[i] = false
4.    if (!interested[k]) then k = i
    }
5.  passed[i] = true
6.  for j in 1 ... N except i do
7.    if (passed[j]) then goto 2
8.  <critical section>
9. passed[i] = false; interested[i] = false
}
```

Il processo i vuole entrare in sezione critica → dichiara di essere interessato (1); finché non ha il turno (2) prova ad ottenerlo, segnalando che non è uscito dal ciclo (3) e controllando se l'ultimo processo che è entrato in sezione critica è ancora interessato (4) (→ se non lo è significa che è uscito); esce dal ciclo e segnala che sta per passare (5)

→ per l'interleaving potrebbero esserci + processi che sono passati; per questo il processo controlla che non ci siano (6-7)

→ quando è sicuro di poter entrare in sezione critica (8); dopo essere uscito lo segnala (9)

Caratteristiche:

- mutua esclusione
- no deadlock
- non è garantita l'assenza di starvation
- necessita di read e write atomiche e di memoria condivisa per k