DEADLOCK AND STARVATION PROBLEM

Il blocco permanente di un insieme di processi che competono per risorse di sistema o comunicano tra loro. Un insieme di processi è in deadlock quando è in attesa che un evento venga causato da un altro processo in deadlock. E' una situazione permanente e non ci sono soluzioni efficienti. Va evitato.
Una risorsa si dice riusabile se può essere usata da un processo alla volta in sicurezza e non è consumata dallo stesso. Si dice consumabile se può essere creata e distrutta.
Esempio deadlock con reusable resource:
Ipotesi spazio disponibile 200KB

| P1 | P2 |
|---|---|
| Richiede 80KB | Richiede 70KB |
| Richiede 60KB | Richiede 80KB |

Se i processi fossero svolti sequenzialmente non succederebbe ma se arrivassero entrambi alla seconda richiesta entrerebbero in deadlock poiché prima di liberare memoria entrambi ne richiedono più della disponibile…

Esempio deadlock con consumable resource:

| P1 | P2 |
|---|---|
| Receive(P2); | Receive(P1); |
| … | … |
| Send(P2, M1); | Send(P1, M2); |

In questo caso ipotizzando una receive bloccante entrambi i processi risulterebbero in attesa di un messaggio da parte dell'altro processo… DEADLOCK.

CONDIZIONI PER DEADLOCK
3 condizioni necessarie ma non sufficienti:

1. Mutua esclusione: un solo processo alla volta può accedere ad una data risorsa.
2. Possesso e attesa: un processo può mantenere il possesso di risorse allocate mentre aspetta altre risorse.
3. Assenza di prelascio: Un processo non è forzato a rilasciare una risorsa in suo possesso.

Un'altra condizione che risulta essere conseguenza delle prime 3 è l'Attesa circolare, ossia un catena chiusi di processi in cui tutti hanno almeno una risorsa bloccata richiesta da un altro processo in attesa.

Ci sono 3 tipi diversi di approcci verso il deadlock: uno preventivo, dove si cerca di eliminare una delle condizioni necessarie per il deadlock, uno d'esclusione, in cui dinamicamente di effettuano delle scelte in base al fatto che una richiesta potrebbe causare deadlock, e uno di detenzione, ossia captare la creazione di deadlock e cercare di ripararlo.

DEADLOCK PREVENTION STRATEGY
Creare un sistema in cui la possibilità di deadlock sia esclusa. Due differenti metodi: quello indiretto, dove si cerca di prevenire il verificarsi di una delle prime 3 condizioni necessarie, e uno diretto in cui si cerca direttamente di evitare che si formi un'attesa circolare (quarta condizione).

Nel caso in cui l'accesso ad una risorsa prevede la mutua esclusione, quest'ultima deve essere supportata dall'OS.
Per quanto riguarda il possesso e l'attesa, si può obbligare ogni processo a richiedere tutte le risorse all'inizio e nel caso le richieste non possano essere soddisfatte lo si blocca. Davvero una brutta soluzione per vari motivi: tra cui il fatto che molte risorse verrebbero bloccate in partenza per poi magari essere utilizzate alla fine, si viene anche meno alla programmazione modulare.
Per quanto riguarda la non liberazione delle risorse, ad un processo potrebbe essere rifiutate delle altre nuove richieste obbligando quel processo a rilasciare le precedenti risorse richiedendole di nuovo con l'aggiunta delle nuove risorse, oppure, l'OS potrebbe anticipare un secondo processo forzandolo a rilasciare una determinata risorsa. Si può fare questo discorso solo su risorse il cui stato è facilmente salvabile e ripristinabile.
Per l'attesa circolare si può ipotizzare di assegnare a ogni risorsa un indice naturale crescente con la condizione che si è in possesso di una risorsa i-esima si può solo richiedere un'altra risorsa j-esima con $j > i$. Questa soluzione però rallenta moltissimo i processi impedendo l'uso di alcune risorse.

DEADLOCK AVOIDANCE STRATEGY
L'approccio prevede di prendere dinamicamente decisioni su richieste di allocazione risorse che potenzialmente potrebbero portare ad un deadlock. Questo richiede di conoscere le future richieste del processo. Questo approccio migliora la concorrenza rispetto alla deadlock prevention strategy.
Abbiamo due modalità:

1. Rifiuto del permesso di esecuzione: ossia non permettere l'avvio di un processo se la sua richiesta potrebbe portare ad un deadlock
2. Rifiuto di allocare una risorsa: non concedere un ulteriore richiesta di risorsa ad un processo se questa allocazione potrebbe portare ad un deadlock.

Negare l'avvio di un processo non è del tutto ottimale mentre il blocco dell'allocazione delle risorse potrebbe avere dei vantaggi nei confronti della deadlock detection strategy (sotto) ed è meno restrittivo della deadlock prevention strategy.
C'è però da dire che il numero massimo di risorse deve essere espresso a priori.
I processi devono essere indipendenti e senza requisiti di sincronizzazione.
Ci deve essere un numero fisso di risorse da allocare.
Nessun processo può terminare conservando le risorse precedentemente ottenute.

DEADLOCK DETECTION STRATEGY
A differenza della deadlock detection strategy la detection asseconda tutte le richieste di risorse dei processi finchè è possibile.
Possiamo decidere di effettuare un check per il deadlock molto frequentemente cioè a ogni richiesta di risorsa o meno frequentemente… dipende da come il problema potrebbe presentare un deadlock.
Un check a ogni richiesta significherebbe consumare decisamente il tempo della CPU ma attraverso algoritmi semplici è comunque molto semplice farlo.
Ci sono più metodi di recovery per una situazione di deadlock: si possono terminare tutti i processi coinvolti (il più comune approccio), salvare lo stato dei processi coinvolti in punti precedenti del loro task e riavviare tutti i processi, oppure terminare i processi in deadlock finche la situazione di deadlock non si verifichi più, oppure prevenire l'uso di alcune risorse finchè il deadlock non esista più.

SOFTWARE APPROACHES TO MUTUAL EXCLUSION

La mutua esclusione può essere implementata anche a livello software. Esempio concreto più processi comunicano tra loro tramite un'area di memoria condivisa e l'accesso a tale area deve essere in mutua esclusione… vediamo i vari tentativi e i loro problemi.

Primo tentativo

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

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

while (turn != me) /* busy wait */ ;
/* CS */
turn = other;
```

Questo tentative garantisce mutual esclusione tra due processi… il problema però è che se un processo non effettua per nessun motive il proprio task settando infine il token pass con la var other, l'altro processo resterebbe in busy waiting per secoli.

Secondo tentativo

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

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

while (flag[other]) /* busy wait */ ;

flag[me] = true;
/* CS */
flag[me] = false;
```

Questo tentativo non garantisce mutua esclusione poiché se entrambi i processi verificassero la condizione di flag del while prima che qualcuno setti a true una delle due entrambi avrebbero accesso alla critical section.

Terzo tentativo

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

// assignments valid for P0 (flip for P1)
int me = 0, other = 1;
flag[me] = true;
while (flag[other])  /* busy wait */ ;
/* CS */
flag[me] = false;
```

Questo tentativo invece porta ad una condizione di deadlock nel caso entrambi i processi settassero la propria flag a true. Entrambi aspetterebbero un tempo indefinito nel while.

Quarto tentativo

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

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

Questo tentativo evita il deadlock ma non il livelock.

Una corretta soluzione è l'algoritmo di Dekker.

```
int me = 0, other = 1; // P0 (flip for P1)
int turn = (uno qualsiasi all'inizio)

while (true) {
        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;
}
```

Il senso è: finchè ho bisogno di entrare in cs setto la mia flag a true. Poi mi chiedo ma anche l'altro processo ha bisogno di entrare o è entrato? Se la risposta è no entro in sezione critica altrimenti prendo delle precauzioni. Se il turno è il mio rifarò il controllo del ciclio while interno sulla flag dell'altro finchè non acrà settato la sua flag a false. Se non è il mio turno semplicemente mi faccio da parte e setto la mia flag a false e aspetto in busy waiting finchè l'altro non setta il turno a me, dopodichè mi rimetto interessato e riparte l'algoritmo di verifica.

L'algoritmo di Dijkstra generalizza l'algoritmo di Dekker per N processi.
Su altra pagina l'algoritmo.

/* global storage */
boolean interested[N] = {false, ..., false} //vettori in cui ogni posizione i corrisponde a un processo
boolean passed[N] = {false, ..., false}
int k = <any> // k ∈ {0, 1, ..., N-1} //sostituto della var turn alg di dekker

/* local info */
int i = <entity ID> // i ∈ {0, 1, ..., N-1} //var me dell'alg di dekker

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

Sono interessato e piazzo la mia flag a true. Poi verifico se è il mio turno, se non fosse così dico che non sono passato e mi chiedo se quello a cui tocca è interessato, se non lo è allora mi prendo il turno e successivamente dirò di essere passato. Poi controllo se qualcun altro è passato, nel caso di esito positivo torno al punto 2.