Appunti di sistemi operativi

Appunti per il corso universitario di sistemi operativi, riferito a sistemi Unix/Windows.
Si discute su problemi di sincronizzazione, memoria e scheduling dei processi.

SINCRONIZZAZIONE DEI PROCESSI

supponiamo di avere una variabile condivisa:

x = 5

abbiamo 2 processi p e q, con determinate istruzioni:

x = x + 1

p1: r1 = x

p2: r1 = r1 + 1

p3: x = r1

x = x – 1

q1: r2 = x

q2: r2 = r2 - 1

q3: x = r2

qual'è la sequenza di scheduling che crea problemi?

p1, p2, p3, q1, q2, q3 => x = 5 A POSTO

p1, q1, p2, q2, p3, q3 => x = 4 NON VA BENE

p1, q1, q2, q3, p2, p3 => x = 6 NON VA BENE

quindi certe parti di codice hanno bisogno di essere eseguite senza che ci sia un altro processo che esegue codice critico

condizioni di interferenza:

1. 2 o più processi accedono ad una informazione comune
2. la modificano attraverso sequenze di operazioni non atomiche
3. quando il risultato finale dei 2 processi varia al variare dell'ordine relativo delle singole operazioni

termini:

- **sezione critica** = segmento di codice nel quale il processo può modificare variabili comuni

- **sezione critica** = segmento di codice nel quale il processo può modificare variabili comuni, aggiornare una tabella, scrivere un file, ecc.
- **sezione di ingresso** = segmento di codice che chiede il permesso al sistema per l'esecuzione della sezione critica del processo in cui si trova
- **sezione di uscita** = parte che può seguire la sezione critica
- il resto del codice viene considerato **sezione non critica**

SEZIONE CRITICA

proprietà di mutua esclusione = se un processo o un thread è nella sezione critica, allora nessun altro processo o thread deve essere all'interno della sezione critica

il codice critico può essere interrotto purché non venga eseguito codice critico (non si tratta di un'atomicità assoluta)

SOLUZIONI SOFTWARE

vogliamo progettare un codice d'ingresso da eseguire prima della sezione critica, allo scopo di poter garantire la mutua esclusione:

```
<sezione di ingresso>
<sezione critica>
<sezione di uscita>
```

idea 1: supponiamo di avere 2 processi e una flag che indica se abbiamo o meno una sezione di codice critica

P1	*P2*

`lock = false`
 	
`while (1){`
`  ...`
`  while(lock){} // !!!`
`  lock = true;`
`  <sezione critica>`
`  lock = false;`
`  ...`
`}`
	`lock = false`
	
	`while (1){`
	`  ...`
	`  while(lock){}`
	`  lock = true;`
	`  <sezione critica>`
	`  lock = false;`
	`  ...`
	`}`

problema: nel punto “!!!” potrebbe avvenire l'interruzione e passare di nuovo l'esecuzione al processo P2 e a quel punto nessuno avrebbe ancora toccato il valore di lock, quindi se avvenisse una interruzione all'interno della sezione critica di P2 che fa ripartire l'esecuzione di P1, ci sarebbero dei problemi

idea 2: inserire una variabile che identifica chi è autorizzato a eseguire la propria sezione critica

P1	*P2*

`turno = 1; // oppure 2`
 	
`while(1){`
`  while (turno != 1){`
`  ...`
`  <sezione critica>`
`  turno = 2;`
`  ...`
`  }`
`}`
	`turno = 1; // oppure 2`
	`// input bloccante`
	`while (1){`
	`  while (turno != 2){`
	`  ...`
	`  <sezione critica>`
	`  turno = 1;`
	`  ...`
	
	`}`
	
	`}`

problema: mi costringe ad una sequenza alternata sempre uguale, che mi crea dei problemi di efficienza: se ad esempio P2 nel punto “input” fa un input bloccante e altre sue elaborazioni, il P1

efficienza, se ad esempio P2 nel punto "input" fa un input bloccante o altre sue elaborazioni, il P1 deve aspettare che P2 esegua la sezione critica per riavere il turno (il problema si manifesta se P1 deve eseguire più volte la sezione critica, come ad esempio in un ciclo)

questo non va bene perché dobbiamo far rispettare questa proprietà:

progresso = se nessun processo è nella sezione critica l'accesso alla sezione critica deve essere consentito

idea 3: pronto[i]: il processo i vuole accedere alla sezione critica

P1	*P2*

`while(1){`
`  pronto[1] = true; // int`
`  while(pronto[2]){}`
`  <sez. critica>`
 	
`pronto[1] = false;`
 	
`}`
	`while(1){`
	`  pronto[2] = true;`
	`  while(pronto[1]){}`
	`  <sez. critica>`
	
	`pronto[2] = false;`
	
	`}`

presenza del while -> inevitabilmente uno è riuscito a entrare per primo; nel momento in cui l'altro cerca di entrare si deve bloccare per forza -> vale la mutua esclusione

non vale il progresso: i processi possono bloccarsi indefinitamente, nel caso limite in cui sia interrotto P1 al punto "interruzione", infatti sia pronto[1] che pronto[2] risulterebbero pari a 1 e quindi provocherebbero due loop nei rispettivi while

ALGORITMO DI PETERSON (PER 2 PROCESSI)

P1	*P2*

`while(1){`
`  pronto[1] = true;`
`  turno = 2;`
`  while(pronto[2] && turno == 2){}`
`  <sez. critica>`
 	
`pronto[1] = false;`
 	
`}`
	`while(1){`
	`  pronto[2] = true;`
	`  turno = 1;`
	`  while(pronto[1] && turno == 1){}`
	`  <sez. critica>`
	
	`pronto[2] = false;`
	
	`}`

funziona perchè:

- non accade più che un processo si blocchi se l'altro non è nella sezione critica
- prima o poi il processo in attesa si blocca

supponiamo che P1 sia entrato nella sezione critica e P2 cerchi di entrare -> non vale la condizione del while -> ci sono 2 possibilità perché sia falsa:

1. l'altro non è pronto: quindi P2 mette pronto true e da il turno a P1, e si blocca
2. erano pronti entrambi, ma a questo punto deve essere turno = 1 quindi P2 si bloccherà comunque

più in dettaglio (supponiamo di avere uno scheduler RR che esegue una riga di codice per processo):

- eseguo "pronto[1] = **true**;"
- eseguo "pronto[2] = **true**;"
- eseguo "turno = 2;"

- eseguo “turno = 1;”
- a questo punto non entro nel ciclo “**while**(pronto[2] && turno == 2){}” poiché la condizione “turno == 2” non è rispettata; nel frattempo il ciclo “**while**(pronto[1] && turno == 1){}”, pone P2 in attesa poiché entrambe le condizioni sono rispettate
- P1 finisce la sua sezione critica e pone pronto[1] = false => P2 ha terminato la sua esecuzione
- P2 esce dal ciclo “**while**(pronto[1] && turno == 1){}”, poiché ora non è più vera pronto[1] = true => anche P2 può eseguire la sua sezione critica e terminare l'esecuzione dopo aver posto “pronto[2] = false”

altro importante requisito:

assenza di starvation = ossia attesa limitata; un processo non deve attendere indefinitamente prima di accedere alla sezione critica; da non confondere con il progresso (è più individuale come proprietà)

ALGORITMO DEL FORNAIO

```
scelta[i] = true;
numero[i] = max(numero[0], ... , numero[n - 1];
scelta[i] = false;

for (j = 0; j < n; j++){
   while(scelta[j])
      while((numero[j] != 0) && ((numero[j], j) < (numero[i], i)){}
      <sezione critica>
}
```

ci da sia progresso che assenza di starvation

SINCRONIZZAZIONE UTILIZZANDO HARDWARE SPECIFICO

dobbiamo garantire:

- **mutua esclusione**
- **progresso**
- **attesa limitata**

come le possiamo garantire?

DISABILITAZIONE DELLE INTERRUZIONI

- dal punto di vista delle proprietà va bene (le rispetta tutte), ma questo meccanismo, rende la sezione critica un **blocco atomico**, rendendo impossibile l'esecuzione di codice non critico dell'altro processo: se il processo si blocca all'interno della sezione critica, siccome blocca il sistema a livello hardware, **blocca la macchina**, in quanto nemmeno il S.O. può più intervenire per interrompere l'esecuzione
- se vogliamo applicarlo ad una macchina a **più processori**, ogni processore ha un proprio meccanismo di gestione degli interrupt, quindi devo **bloccare tutti i processori**, il che è **complicato** dal punto di vista hardware e inefficiente

P1	***P2***
il sistema maschera le interruzioni <sezione critica>	<sezione critica>

il sistema riabilita le interruzioni	

SINCRONIZZAZIONE INTRODUCENDO ISTRUZIONI MACCHINA SPECIALI

tali istruzioni eseguono più operazioni atomicamente

TEST-AND-SET

questa sono operazioni hardware (perché devono essere atomiche), il codice serve solo a descriverle

```
Test-and-set(boolean x){
boolean val = x;
x = true;
return val;
}
```

la prima soluzione:

```
lock = false;

while(lock){
lock = true;
<sezione critica>
lock = false;
}
```

il problema era che poteva avvenire un'interruzione tra il test di lock e il set della stessa

possiamo correggerlo utilizzando la funzione appena vista:

```
lock = false;

while(1){
while(Test-and-set(lock)){
<sezione critica>
lock = false;
}
}
```

questa risolve il primo caso di interferenze che avevamo visto rendendo atomico il test e il set della variabile lock

XCHG

XCHG

```
swap(x,y){
boolean tmp = x;
x = y;
y = tmp;
}
```

(sempre istruzione hardware atomica come la precedente)

possiamo utilizzarla in questa maniera:

```
do{
chiave = true;
while(chiave == true)
swap(blocco, chiave);
<sezione critica>
blocco = false;
}while(1);
```

questa istruzione è presente nei processori Pentium di Intel, e tramite questa realizzano lo **spin lock**

problemi generici dell'utilizzo di istruzioni macchina:

- la loro **realizzazione** hardware è molto **complessa**
- nel momento in cui blocchiamo un processo con i metodi visti fino ad ora, si tratta di **attesa attiva (busy waiting)**, in quanto ottenuta tramite un ciclo while; questo provoca nel caso di sistemi multiprogrammati lo spreco di quanti di tempo in cui potrebbero essere eseguiti processi attivi

SINCRONIZZAZIONE TRAMITE SISTEMA OPERATIVO

oggetti:

- **semafori** (operiamo su questi tramite chiamate a sistema)
- **monitor** (costrutto linguistico che troviamo ad es. in Java)

P1	***P2***
P(S) <sezione critica> V(S)	P(S) <sezione critica> V(S)

SEMAFORI

Le soluzioni viste precedentemente non sono adatte ad affrontare problemi più complessi, ossia con più processi che richiedono l'esecuzione di sezioni critiche

SEMAFORI CONTATORI

SEMAFORI CONTATORI

idea di **Dijkstra** del 1965

- variabile interna
- **P** (proberen, verificare) "**wait**" (altra notazione per identificare la stessa operazione, anche se tipica dei monitor)
- **V** (verhogen, incrementare) "**signal**"

S = semaforo

contenuto di S:

- se <= 0 (rosso) il processo viene bloccato sul semaforo
- se > 0 (verde), il processo prosegue

codice della wait:

```
P(S){

while(S <= 0){} // non eseguo alcuna operazione

S = S - 1;

}
```

codice della signal:

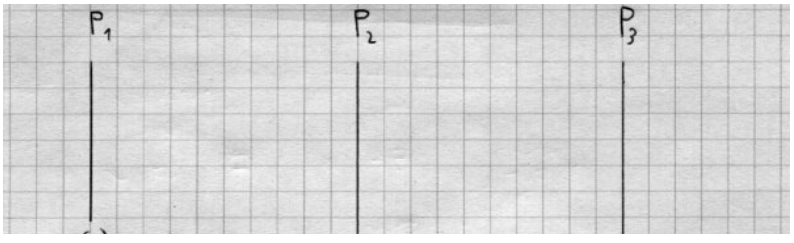
```
V(S){

S = S + 1;

}
```

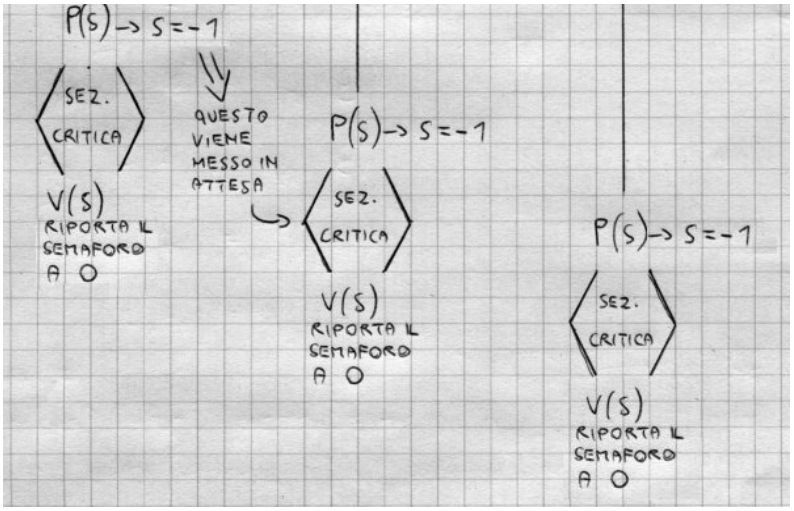
questo tipo di semaforo è anche detto **spin lock**, per il fatto che continua a eseguire il ciclo che contiene (ossia "gira", spin), finché attende (lock); risulta utile per sistemi con più unità di elaborazione in quanto si evita in molti casi la necessità di costosi context-switch (infatti il processo in wait resta in esecuzione mentre se andasse in attesa dovrebbe esserci per forza un context-switch con un altro processo pronto)

se ad esempio inizialmente un semaforo vale 2:
faccio la prima P, il semaforo diventa 1, seconda P diventa 0 e blocco

il valore del semaforo rappresenta:

- il numero di accessi consentiti (ad esempio può indicare il numero di risorse libere) se maggiore o uguale a 0
- il numero di processi in attesa se minore di 0 (valido solo per i semafori dotati di una coda, che vedremo tra poco)





il tipo di semaforo appena visto **non permette di evitare il problema dell'attesa attiva**, ossia lo spreco di cicli per far attendere un processo; per ovviare a questo, possiamo definire il semaforo come una struttura:

- int valore
- queue coda = lista di processi

si può facilmente utilizzare questo modello aggiungendo un campo puntatore al PCB dei processi, per poter così creare la lista di processi che attendono il semaforo

```
typedef struct{
int valore;
struct processo *coda;
}semaforo;

P(semaforo S){
S.valore = S.valore - 1; // è come se il test fosse S <= 0
if(S < 0)
<aggiungi processo corrente in S.coda>
block();
}

V(semaforo S){
S.valore = S.valore + 1;
if (S.valore <= 0){
<togli processo P dalla S.coda>
wakeup(P);
}
}
```

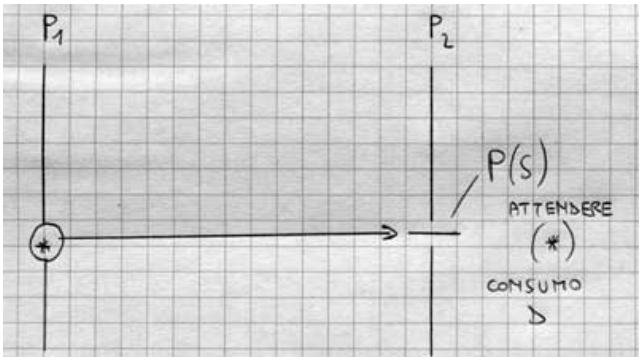
precisazioni relative al codice qui sopra:

- |S| = numero di processi in attesa
- l'operazione **block** sospende il processo che la invoca
- **wakeup**(P) pone in stato di pronto per l'esecuzione un processo P bloccato
- mentre la definizione classica (Djkstra) del semaforo ad attesa attiva è tale che il valore del semaforo non è mai negativo (a meno che non lo inizializziamo appositamente), questa può condurre a valori negativi (che rappresentano il numero di processi nella coda di attesa del semaforo) (vedi pag. 210)
- l'incremento è prima della verifica perché dobbiamo sapere la quantità di processi in coda (se in V S.valore <= 0 significa che prima dell'assegnamento S.valore era strettamente minore di 0)

la wait e la signal sono operazioni atomiche; per eseguirle:

- ambiente a CPU singola: è sufficiente disabilitare gli interrupt
- ambiente a CPU multipla: bisogna utilizzare le istruzioni atomiche (Test-and-set ecc.)

istanza del problema del produttore-consumatore (P1 deve produrre, P2 deve consumare):



PRODUTTORE/CONSUMATORE CON I SEMAFORI

PRODUTTORE

```
while(1){                       // vuote = DIM
  <Produci D>
  // (1)
  P(vuote)
  buffer[inserisci] = D;  // buffer di dimensione DIM, posso far entrare DIM
                                    processi;
  inserisci = (inserisci + 1) % DIM;
  V(piene) // salta a (2)
}
```

CONSUMATORE

```
while(i){                       // piene = 0
  // (2)
  P(piene)
  D = buffer[preleva];
  preleva = (preleva + 1) % DIM;
  V(vuote); // alta a (1)
  <Consuma D>
}
```

nelle istruzioni:

buffer[inserisci] = D;

inserisci = (inserisci + 1) % DIM;

potrei avere dei problemi, quindi devo introdurre una sezione critica, utilizzando un mutex (semaforo di mutua esclusione):

```
while(i){                       // piene = 0
  // (2)
  P(piene)
  P(mutex)
  D = buffer[preleva];
  preleva = (preleva + 1) % DIM;
  V(mutex)
  V(vuote); // alta a (1)
  <Consuma D>
}
```

STALLO E ATTESA INDEFINITA CON I SEMAFORI

un semaforo con coda d'attesa può generare delle situazioni di stallo (**deadlock**), ossia di attesa indefinita di un evento

es.

P0	***P1***
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`.`	`.`
`.`	`.`
`.`	`.`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

Il processo P0 esegue wait(S), quindi P1 deve aspettare signal(S) prima di entrare nella sezione critica; d'altro canto P1 esegue wait(Q), quindi P0 deve aspettare signal(Q) prima di poter eseguire la sua sezione critica e, in seguito, signal(S); ciò comporta una situazione di stallo

lo stallo si può anche presentare come starvation, ad esempio se la coda dei processi in attesa è gestita con un algoritmo LIFO

SEMAFORI BINARI

i semafori visti fino ad ora sono detti semafori contatori, in quanto possono assumere qualsiasi valore intero; i semafori binari possono assumere solo 2 valori: 1 oppure 0

$P_B(S)$:

- se S==1 (verde) decrementa
- se S==0 blocca (rosso -> si forma una coda di processi in attesa)

$V_B(S)$:

- sblocca il primo processo in coda
- rende il semaforo verde (S=1) se non ci sono processi bloccati

quindi posso utilizzarle in questa maniera:

$P_B(S)$

sezione critica

$V_B(S)$

è **possibile realizzare un semaforo contatore utilizzando 2 semafori binari**:

inizialmente S1 = 1 e S2 = 0, l'intero C si imposta al valore iniziale del semaforo contatore S

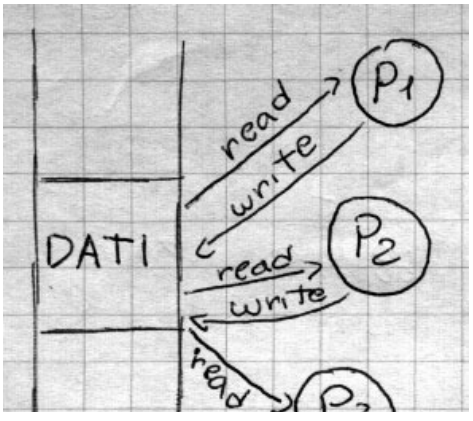
codice operazione wait (P):

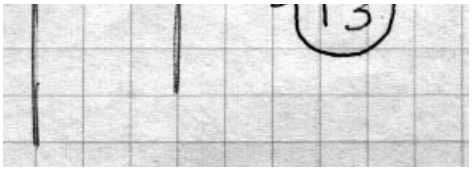
```
wait(S1);
C--;
if (C < 0){
signal(S1);
wait(S2);
}
signal(S1);
```

codice operazione signal (V):

```
wait(S1);
C++;
if (C <= 0)
signal(S2);
else
signal(S1);
```

PROBLEMA DEI LETTORI E DEGLI SCRITTORI





si consideri un insieme di dati condivisi tra numerosi processi concorrenti, che potrebbero richiedere l'accesso sia in lettura che in scrittura;

tali processi possono essere:

- lettori: sono interessati solo alla lettura
- scrittore: possono **aggiornare** i dati (letture+scritture)

il **problema** non nasce in lettura ma solo in **scrittura**, quindi ci interessa proibire solo le scritture contemporanee; in particolare il problema dei lettori e degli scrittori ha 2 varianti:

1. nessun lettore deve attendere a meno che uno scrittore abbia già ottenuto il permesso di usare l'insieme dei dati condiviso
2. se uno scrittore attende per l'accesso all'insieme dei dati, nessun nuovo lettore deve iniziare la lettura altrimenti potrebbe esserci uno stallo dei scrittori

insomma:

1. se qualcuno sta leggendo nessuno deve scrivere
2. se qualcuno sta scrivendo nessuno deve leggere
3. se qualcuno vuole scrivere, chi sta leggendo finisce e se ne va, e nessun altro inizia a leggere

soluzione: accesso a n lettori oppure 1 scrittore

come implementiamo la soluzione con i semafori?

LETTORE	***SCRITTORE***
`semaforo mutex = 1` mutex è semplicemente il nome del semaforo, come dire int pippo	

`P(mutex)`
 	
`<lettura>`
 	
`V(mutex)`
	`P(mutex)`
	
	`<scrittura>`
	
	`V(mutex)`

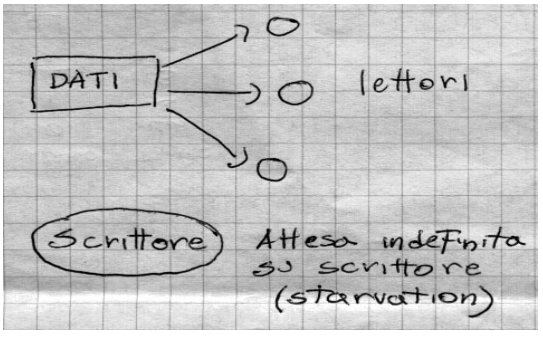
questo ci fornisce un lettore oppure uno scrittore, funziona ma noi vogliamo la nostra soluzione (*) che è meno restrittiva

utilizziamo quindi due semafori:

LETTORE	***SCRITTORE***

`semaforo mutex = 1`
`semaforo scrittura = 1`
`num_lettori = 0`	

`P(mutex)`
`num_lettori++`
`if(num_lettori==1)`
`P(scrittura)`
`V(mutex)`
`<lettura>`
`P(mutex)`
`num_lettori--`
`if(num_lettori==0)`
`V(scrittura)`
`V(mutex)`
	`P(scrittura)`
	`<scrivi>`
	`V(scrittura)`



però questa soluzione ha un problema di starvation degli scrittori rispetto ai lettori; solo il primo lettore passa per il primo if, gli altri saltano ed entrano "gratis" e quindi se ho tanti lettori.. lo scrittore va in starvation (come scritto all'inizio)

PROBLEMA DEI CINQUE FILOSOFI

esempio classico dei filosofi a cena (dining philosophers)

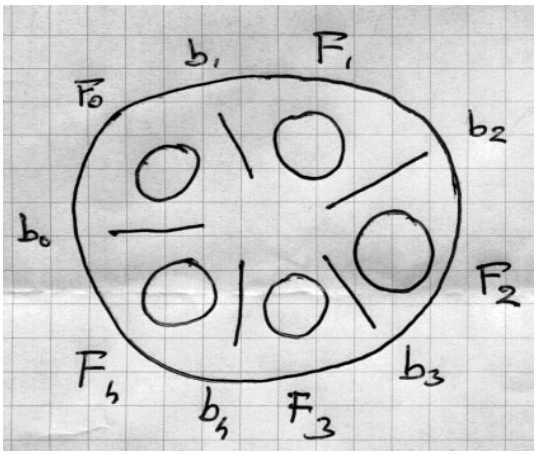
vi sono 5 filosofi seduti ad un tavolo che possono compiere 2 attività non contemporaneamente:

- pensare

- mangiare

sul tavolo ci sono 5 bacchette: ogni filosofo per mangiare deve avere 2 bacchette in mano:

- può prendere solo quelle immediatamente alla propria destra e sinistra
- non può prendere una bacchetta se è in mano ad un altro filosofo => problema di sincronizzazione



risorse: bacchette b_0...b_4

processi: filosofi F_0...F_4

soluzione con i semafori:

5 semafori mutex inizializzati ad 1 (dato che ci sono 5 bacchette diverse che sono 5 risorse critiche diverse)

Filosofo(i) {

while(1) {

<pensa>

$P(b_i)$

$P(b_{(i+1)\%5})$

<mangia>

$V(b_i)$

$V(b_{(i+1)\%5})$

}

}

stallo: tutti e 5 i filosofi raccolgono la bacchetta a dx e attendono la sx!! quindi tutti i processi sono bloccati, non uscirò mai più!

nel codice avviene quando si interrompe dopo la terza riga (dopo $P(b_i)$)

per evitare questo problema potrei usare dei timeout (nella pratica si fa) ma se non voglio introdurre nuovi tipi di semafori con timer e voglio sempre utilizzare i semafori di base diventa molto

nuovi tipi di semaforo con timer e voglio sempre utilizzare i semafori di base diventa molto complesso

quindi le soluzioni sono:

1. rendere atomica l'acquisizione delle bacchette (il che si fa tramite i monitor e non con i semafori di base)
2. un filosofo può prendere le sue bacchette solo se sono entrambe disponibili (va eseguita in una sezione critica)
3. soluzione asimmetrica: un filosofo dispari prende per prima la bacchetta di sinistra e poi quella di destra, mentre un filosofo pari prende prima quella di destra e poi quella di sinistra
4. solo 4 filosofi alla volta possono stare a tavola contemporanemente: aggiungo una risorsa tavola e un semaforo tavolo inizializzato a 4

soluzione 3:

scrivo un filosofo "mancino" con codice diverso dagli altri (posso mettere un if) che invece che prendere prima la bacchetta a destra prende prima quella a sinistra

per implementarla se si scrive un codice diverso per ogni filosofo è sufficiente invertire l'ordine delle bacchette, altrimenti nel caso in cui si utilizzi un codice unico bisognerà necessariamente inserire un if

```
Filosofo(i) {
while(1) {
<pensa>
if(i==3) {
P(b_(i+1)%5)
P(b_i)
} else {
P(b_i)
P(b_(i+1)%5)
}
<mangia>
V(b_i)
V(b_(i+1)%5)
}
}
```

soluzione 4:

```
tavolo = 4
```

```
Filosofo(i) {

while(1) {

<pensa>

P(tavolo)

P(b_i)

P(b_(i+1)%5)

<mangia>

V(b_i)

V(b_(i+1)%5)

V(tavolo)

}

}
```

MONITOR

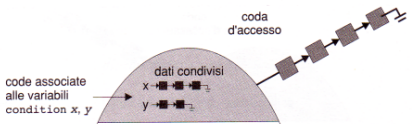
è un costrutto linguistico (ed esiste in java), introdotto da Charles Hoare nel '74

```
Monitor <nome> {

<variabili condivise>

proc1() {

...

}

procn() {

....

}

<codice di inizializzazione>

}
```

- c'è mutua esclusione nell'uso delle procedure
- le variabili non sono accessibili "dall'esterno" del monitor
- variabili "speciali" chiamate **condition** V che sono **code**:
  - **V.wait** = sempre bloccante
  - **V.signal** = sblocca il primo processo in coda (se non c'è nessun processo in coda non fa nulla)

è come se fosse un semaforo "unario" sempre rosso

il risultato è una programmazione molto più chiara e meno a rischio di errori



PRODUTTORE/CONSUMATORE CON I MONITOR

```
Monitor buffer {
condition vuoto,pieno;
int buffer[DIM];
int contatore=0;
int preleva=0, inserisci=0;

riempi(int p) {
if(contatore==DIM)
vuoto.wait; // aspetta che si svuoti
<scrive nel buffer>
contatore++;
pieno.signal; // segnala di aver riempito
}

int svuota() {
if(contatore==0)
pieno.wait; // aspetta che si riempia
<leggo d dal buffer>
contatore--;
vuoto.signal; // segnala di aver svuotato
return d;
}
}

Produttore {
while(true) {
<produci p>
buffer.riempi(p)
}
}

Consumatore {
while(true) {
```

```
while(true) {

p = buffer.svuota();

<consuma p>

}

}
```

(praticamente vuoto è una coda di scrittori, mentre pieno è una coda di lettori)

problema!!

signal: mutua esclusione --> non posso avere due processi attivi nel monitor quindi:

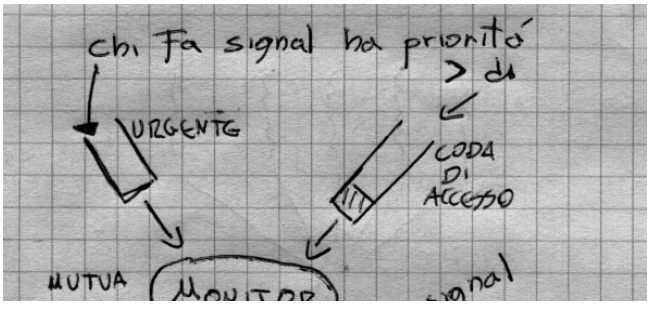
1. o proseguo con l'esecuzione del processo che ha fatto la signal e metto il processo sbloccato in una coda di processi pronti
2. o blocco il processo che ha lanciato la signal, e lo metto in una coda **urgente** di processi pronti ed eseguo il processo sbloccato

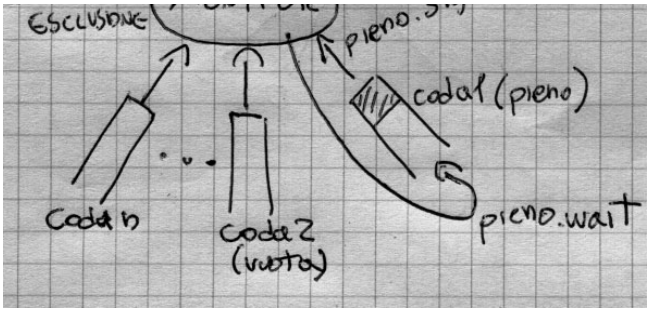
```
Monitor buffer {
  ...
  void riempi(int p) {
    if (contatore == dim)
      vuoto.wait;
    <scrivi su buffer>
    contatore++;
    pieno.signal;
  }

  int svuota(){
    if (contatore == 0)
      pieno.wait;
    <leggi da buffer p>
    contatore--;
    vuoto.signal;
    return (p);

}

}
```

il processo che ha effettuato la signal entra in una coda urgente e attende, il difetto è che blocco un processo che non dovrebbe essere bloccato

il comportamento di default del monitor è quello di bloccare la signal e sbloccare subito il “riempi”, affinchè possiamo essere sicuri che la condizione contatore == dim sia falsa

chi fa una signal deve attendere meno e va nella coda “urgente”, che ha priorità maggiore (i processi sbloccati con una signal devono avere priorità rispetto ai nuovi processi in arrivo)





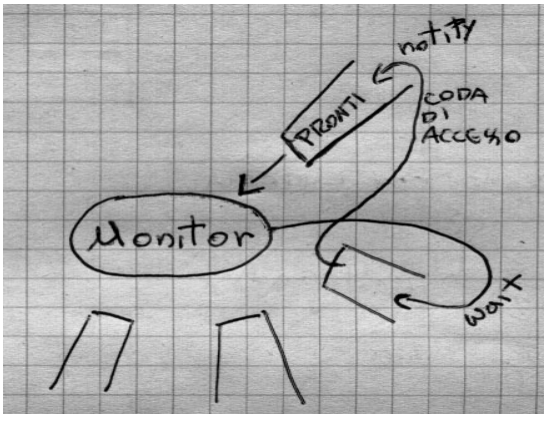
i processi entrano nella coda 1 con una wait ed escono con una signal

come modifico il codice affinché non vada avanti sempre ma controlli contatore == dim:

```
Monitor buffer{
   ...
   void riempi(int p)
      while (contatore == dim)
         vuoto.wait;
      <scrivi su buffer>
      contatore++;
      pieno.signal;
   }

   int svuota(){
      while (contatore == 0)
         pieno.wait;
      <leggi da buffer p>
      contatore--;
      vuoto.signal;
      return (p);

}

}
```

un'altra soluzione al problema precedente è di utilizzare il costrutto linguistico **notify** invece di signal: notify sblocca un processo ma non lo esegue subito, ossia permette al codice che ha chiamato la notify di continuare la sua esecuzione

osservazione: la notify di java sblocca uno dei processi in attesa, ma il chiamante non sa quale, dipende dalla JVM



notifyall sblocca tutti i processi in attesa sulla condizione: semplifica molto la programmazione, ma non è molto efficiente

per implementare i monitor in Java si usa il costrutto Synchronized, che permette la mutua esclusione:

```
class Monitor{
   Synchronized <metodo>(){
      ...
   }
}
```

quando si invoca un metodo sincronizzato, l'oggetto viene bloccato, abbiamo mutua esclusione su di esso, nessun altro processo può invocare nessun altro metodo dello stesso oggetto; questi metodi sono quelli che corrispondono al monitor

questo sistema è un po' più flessibile, ma possiamo fare solo wait generiche sull'unica condizione presente, in quanto non possiamo specificarne altre

la wait di java invocata senza parametri resta in attesa fino a che non viene risvegliata da una notify, in alternativa è possibile specificare un parametro che indica il numero di millisecondi da attendere prima del risveglio:

wait(long timeout)

es. di Monitor in Java (5 filosofi):

```
Monitor Tavolo{
   boolean fork_used[5] = false; // forchette numerate da 0 a 4
   condition filosofo[5]; // se lo vogliamo fare in java, questa la dobbiamo
                                    togliere

   raccogli(int n){
      while(fork_used[n] || fork_used[(n+1)%5])
         filosofo[n].wait();
      fork_used[n] = true;
      fork_used[(n+1)%5] = true;
   }
   // in java dovevi aggiungere:
   // (synchronized)
   deposita(int n){
      fork_used[n] = false;
      fork_used[(n+1)%5] = false;
      filosofo[n].notify();   // se lo voglio fare in java devo togliere
                                                queste due "filosofo" e sostituire con
                                                notifyall()
      filosofo[(n+1)%5].notify();
   }
}
Filosofo(i){
   while (true){
      <pensa>
      Tavolo.raccogli(i);
      <mangia>
      Tavolo.deposita(i);
   }
}
```

alla versione precedente possiamo aggiungere altri due metodi (rfork e dfork) che ci danno la possibilità di risolvere il problema con i “semafori” (se implementati in java si utilizzeranno per forza i monitor) e quindi con il filosofo mancino

```
Monitor Tavolo {
  boolean fork_used[5] = false; // forchette numerate da 0 a 4
  condition filosofo[5]; // se lo vogliamo fare in java, questa la dobbiamo
                                   togliere

   raccogli(int n){
      while(fork_used[n] || fork_used[(n+1)%5])
         filosofo[n].wait();
      fork_used[n] = true;
```

```
      fork_used[n] = true;
      fork_used[(n+1)%5] = true;
   }
   // in java dovevi aggiungere:
   // (synchronized)
   deposita(int n){
      fork_used[n] = false;
      fork_used[(n+1)%5] = false;
      filosofo[n%5].notify();   // se lo voglio fare in java devo togliere
queste due "filosofo" e sostituire con notifyall()
      filosofo[(n+1)%5].notify();
   }

   rfork (int j){
      while(fork_used[j])
         fork[j].wait();
      fork_used[j] = true;
   }

   dfork(int j){
      fork_used[j] = false;
      fork[j].notify();
   }
}

Filosofo(i) {
   while (true){
      <pensa>
      Tavolo.rfork(i);          // semafori e quindi filoso mancino
      Tavolo.rfork((i+1)%5);
        // oppure utilizzo la soluzione atomica:  Tavolo.raccogli(i);

      <mangia>
      Tavolo.dfork(i);          // semafori e quindi filoso mancino
      Tavolo.dfork((i+1)%5);
        // oppure utilizzo la soluzione atomica:  Tavolo.deposita(i);
   }
}
```

con i monitor conviene utilizzare la soluzione atomica raccogli/deposita, la soluzione rfork, dfork è una alternativa e si utilizza con i semafori

attenzione però che la soluzione atomica raccogli/deposita soffre, anche se poco, di starvation: se i miei vicini mangiano prima uno e poi l'altro, può accadere che io rimanga sempre fermo! la soluzione in quel caso sarebbe una coda (turnazione)

Il sistema operativo generalmente realizza i semafori (almeno quelli binari)

avendo i semafori si realizzano i monitor, e avendo i monitor si realizzano i semafori

```
Monitor semaforo(int init) {
   int valore=init;
   condition coda;

   P(){
      valore--;
      while(valore < 0)
         coda.wait;  // esco dal monitor alla coda esterna e gli altri possono entrare
   }

   V(){
      valore++;
      if (valore <= 0)
         coda.notify;  // questo test in realtà non serve: da definizione e' così
                          ma anche se io la facessi con valore positivo, quindi
                       togliendo l'if, non cambierebbe nulla dato che non ci
                          sarebbero processi in attesa da notificare
   }
}
```

STALLI

o deadlock

statica signfica che non è una condizione a run-time ma è una realizzazione strutturale del programma (quindi o c'è o non c'è); dinamica invece dipende dall'esecuzione a run-time dei processi

un insieme S di processi è in stallo se ogni processo di S è in attesa di un evento che deve essere causato da un altro processo di S

lo stallo avviene sotto 4 condizioni, che devono verificarsi tutte contemporaneamente:

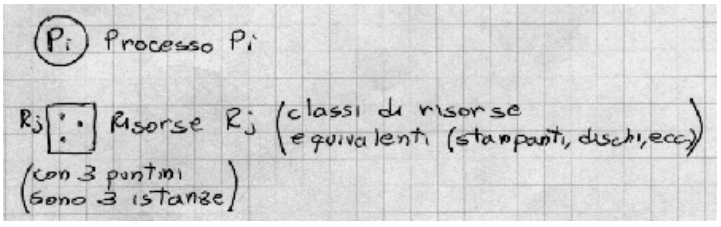
- **attesa circolare** (dinamica) = deve esistere un insieme di processi $\{P_0, P_1, ..., P_n\}$ tale che P_0 attende una risorsa posseduta da P_1, P_1 attende una risorsa posseduta da P_2, P_{n-1} attende una risorsa posseduta da P_n, P_n attende una risorsa posseduta da P_0
- **mutua esclusione** (statica) = almeno una risorsa deve essere non condivisibile (ossia utilizzabile da un solo processo alla volta -> se un altro processo richiede tale risorsa deve attendere fino al suo rilascio)
- **possesso e attesa** (“hold and wait”) (allocazione incrementale delle risorse) (statica) = un processo in possesso di almeno una risorsa, attende di acquisire risorse già in possesso di altri processi
- **impossibilità di prelazione** (no preemption) (statica) = un risorsa può essere rilasciata dal processo che la possiede solo volontariamente, quando ha finito il suo compito (non ci può essere una forzatura da parte di un altro processo)

GRAFO DI ASSEGNAZIONE DELLE RISORSE

rappresenta la situazione dei processi e delle risorse in un sistema:

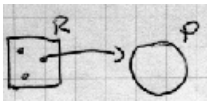
- **cerchi** rappresentano i **processi**
- **rettangoli** rappresentano le **risorse**
- le risorse possono avere più **istanze**: si rappresentano con più **puntini** nei rettangoli
- 3 tipi di archi:

1. di **reclamo** (linea tratteggiata; in una variante del grafo di assegnazione non ci sono) = Pi -> Rj indica che il processo Pi può richiedere la risorsa Rj in un qualsiasi momento futuro
2. di **richiesta** = Pi -> Rj indica che il processo Pi sta richiedendo in questo momento un istanza del tipo di risorsa Rj
3. di **assegnazione** = Pi <- Rj indica che il processo Pi ha ottenuto l'assegnazione della risorsa Rj

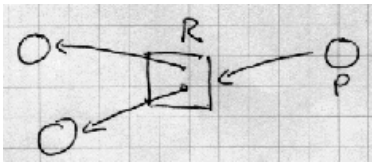


esempi:

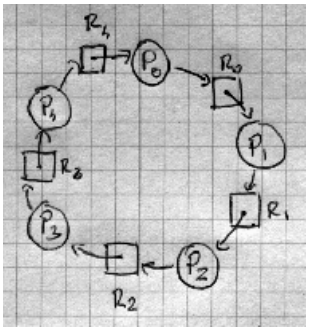
P ha allocato un'istanza di R



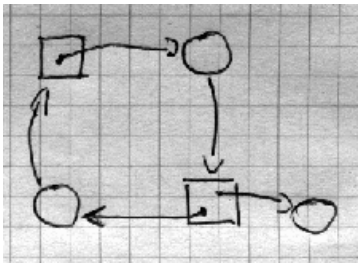
P è in attesa di una istanza di R



nel caso in cui ogni risorsa abbia una sola istanza lo stallo si visualizza con un ciclo nel grafo:



ma non è detto che grafo con ciclo implichi uno stallo infatti:



in questo caso essendoci più istanze di risorse e un processo senza attese, quando questo finisce libera la catena

quindi in generale:

- stallo *implica* ciclo
- ciclo *non implica* stallo
- ma nel caso di una sola istanza per risorsa: stallo *implica* ciclo e ciclo *implica* stallo

GESTIONE DELLO STALLO

PREVENZIONE (è una condizione statica)

= progettare il sistema in modo che non possano verificarsi dead-lock: in genere implica un basso utilizzo delle risorse stesse

il fine è quello di impedire che si verifichi almeno una delle condizioni necessarie allo stallo, in 4 maniere:

Negazione dell'attesa circolare

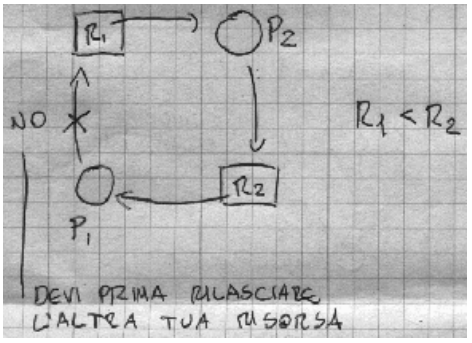
bisogna introdurre un ordinamento totale di tutti i tipi di risorse, e un ordine crescente di numerazione

per le risorse richieste da ciascun processo: ogni processo può richiedere risorse solo seguendo un ordine crescente di numerazione (allocazione gerarchica):

$R_1 < R_2 < R_3 < ... < R_n$

questo ci permette di controllare a run-time se una richiesta porterà dead-lock ed evitare di eseguirla: i processi fanno hold-and-wait ma solo rispettando l'ordinamento: ogni processo può richiedere risorse solo seguendo un ordine crescente di numerazione, quindi se serve una risorsa più piccola si devono rilasciare le risorse più grandi (in base all'ordinamento)

es. del perché funziona:



La dimostrazione per assurdo è che in un ciclo siccome in base all'allocazione gerarchica:

$R_1 < R_2 < R_3 < ... < R_n < R_1$

questa implica $R_1 < R_1$ ma questo è assurdo quindi con questa politica non ci può essere stallo

Negazione della mutua esclusione

Impossibile! tutte le risorse dovrebbero essere condivisibili, ma alcune risorse sono intrinsecamente non condivisibili (es. stampante), e per queste deve valere la mutua esclusione

Negazione della condizione di possesso e attesa

ossia cerco di garantire che un processo che richiede una risorsa non ne possieda altre

posso utilizzare **2 protocolli** diversi:

1. un processo prima di iniziare la sua esecuzione deve richiedere tutte le risorse che gli serviranno
2. un processo non può richiedere altre risorse se prima non ha rilasciato tutte quelle che sta utilizzando

es. processo che prende dati da un'unità a nastri, li copia in un file su un disco, ordina il contenuto e stampa i risultati

1° protocollo: il processo richiede subito l'unità a nastri, il disco e la stampante (anche se questa viene utilizzata solo alla fine)

2° protocollo: il processo richiede inizialmente solo l'unità a nastro e il disco, poi li deve rilasciare e riprendere il disco e la stampante (e chi ci garantisce che i dati sul disco non siano cambiati?); infine rilascia tutto e termina

problemi:

- **inefficienza** poiché molte risorse possono essere assegnate ma non utilizzate per un lungo periodo di tempo
- aumento delle possibilità di **starvation**, in quanto se richiede risorse molto usate può darsi che un processo non si trovi mai nella condizione di poter ottenere contemporaneamente tutte le risorse di cui ha bisogno

Negazione della impossibilità di prelazione

ossia è possibile avere la prelazione su risorse già assegnate; possiamo applicare 2 differenti protocolli:

1. se un processo che possiede una o più risorse ne richiede un'altra che non può ottenere allora deve rilasciare tutte quelle già in suo possesso

deve rilasciare tutte quelle già in suo possesso

2. se un processo richiede delle risorse che non sono disponibili, se queste sono possedute da un processo in attesa, vengono tolte a quest'ultimo e assegnate al processo richiedente; altrimenti il richiedente deve attendere

problema: su risorse che non hanno uno stato proprio si può fare (cpu, ecc) ma non su risorse che hanno uno stato preciso (disco, stampante, ecc)

CONTROLLO

prima di ogni singola assegnazione faccio un controllo (come il banchiere) e lo metto in wait nel caso in cui possa provocare deadlock

la differenza con la prevenzione è che quest'ultima è una politica predefinita che devo seguire, nel controllo invece non ho politiche da seguire ma se rischio uno stallo vengo fermato

IGNORARE LO STALLO

fingere che non possa verificarsi nel sistema: soluzione su cui si basa la maggior parte dei sistemi operativi, UNIX compreso; può portare all'esaurimento delle risorse di sistema (assegnate a processi che non possono più terminare) e alla necessità di un riavvio manuale del sistema

RILEVAMENTO E RIPRISTINO

(o riconoscimento) = rilevamento delle situazioni di stallo ed esecuzione del **ripristino**

il rilevamento è simile al controllo ma lo faccio ad intervalli costanti e se ci sono processi in deadlock cerco di risolvere il problema

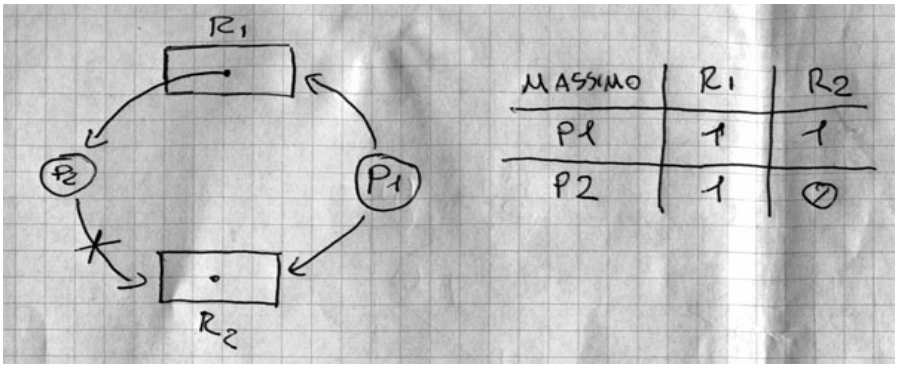
es. utilizziamo la tecnica del controllo

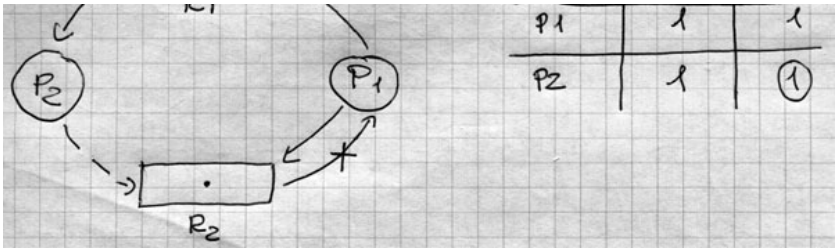
è necessario avere una stima (ci basta un limite massimo) delle richieste future;

quindi ipotizziamo che i processi dichiarino il numero massimo di richieste di risorse

	R1	***R2***	***Rm***
P1	M1	...	M1m
...	...	...	...
Pn	Mn1	...	Mnm

vediamo alcuni esempi del controllo:

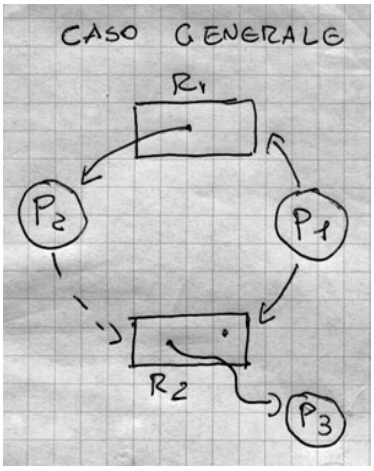




siccome l'algoritmo è prudente, prende gli archi del grafo, e ogni volta che abbiamo un assegnamento di una risorsa, si verifica che non crei un ciclo: se l'assegnazione crea un ciclo allora attendi, altrimenti ok

attenzione all'esame: non confondere deadlock con starvation

es. (con più istanze che risorse):



R2 ha qui 2 istanze: una è occupata da P3, l'altra, siccome da P2 a R2 c'è un arco di reclamo, mentre da P1 a R2 c'è un arco di richiesta, verrà assegnata a P1, in quanto me la sta chiedendo in questo preciso istante (arco di richiesta); dovrei scegliere solo nel caso fossero entrambi di richiesta

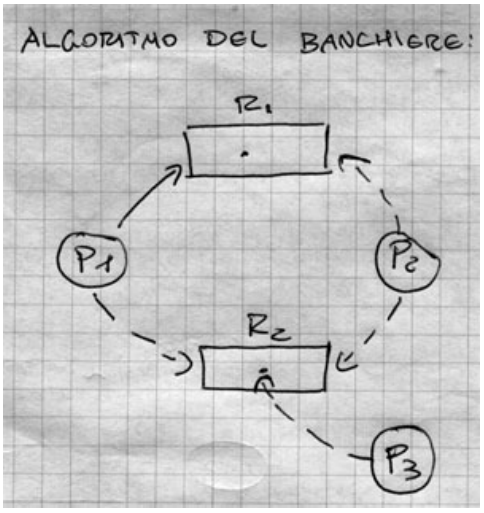
in questo caso nonostante ci sia un ciclo, non abbiamo un deadlock, poiché l'istanza della risorsa R2 assegnata a P3 sarà prima o poi liberata

ALGORITMO DEL BANCHIERE

questo algoritmo può indicare se un sistema si venga a trovare in uno **stato sicuro** o meno nel caso assegnasse una risorsa ad uno dei processi richiedenti

un sistema, nell'allocare le risorse che vengono richieste, deve procedere come farebbe una **banca**: i **processi** sono visti come i **clienti** che possono richiedere del credito presso la banca (fino ad un certo limite individuale) e che le **risorse** allocabili possono essere paragonabili ai **soldi**

il sistema, come la banca, non può permettere a tutti i clienti di raggiungere il loro limite di credito contemporaneamente poiché in tal caso la banca fallirebbe (e il sistema non potrebbe allocare risorse a sufficienza, causando un deadlock)



sequenza sicura = (sequenza possibile di terminazione), sequenza <P1, ..., Pk> in cui a ogni processo per terminare servono tutte le risorse iniziali più quelle allocate dai processi precedenti; in altre parole Pi necessita di una quantità di risorse minore o eguale a (risorse iniziali + risorse allocate ai processi Pj con j < i)

abbiamo **4 matrici**:

1. **Disponibili$_i$** = risorse disponibili
2. **Necessità$_i$** = risorse necessarie al processo i per raggiungere il MAX, ossia la necessità residua di risorse per ogni processo (calcolata sottraendo alla matrice Massimo la matrice Assegnate)
1. **Assegnate$_i$** = risorse assegnate al processo i
2. **Massimo$_i$** = richiesta massima di risorse per ciascun processo

ALGORITMO DI VERIFICA DELLA SICUREZZA

serve per trovare una sequenza sicura e verificare quindi che il sistema si trova in uno stato sicuro

servono 2 vettori:

- **Lavoro := Disponibili**
- **Fine[i] := falso**, per i = 1, 2, ..., n

procedura:

1. cercare un indice i tale che **Fine[i] = falso e Necessità$_i$ <= Lavoro**; se tale i non esiste saltare al passo 3
2. **Lavoro := Lavoro + Assegnate$_i$**

   Fine[i] := vero

   torna al passo 1
3. se **Fine[i] = vero** per ogni i, allora il sistema è in uno **stato sicuro**

ALGORITMO DI RICHIESTA DELLE RISORSE

sia **Richieste$_i$** il vettore delle richieste per il processo Pi: se **Richieste$_i$ [j] = k** allora il processo Pi richiede k istanze del tipo di risorsa Rj

se il processo Pi fa una richiesta di risorse si svolgono le seguenti azioni:

1. **Richiesta$_i$ <= Necessità$_i$**, altrimenti si entra in una condizione di errore

1. se **Richieste$_i$ <= Necessità$_i$** esegue il passo 2 altrimenti riporta una condizione di errore poiché il processo ha superato il numero massimo di richieste
2. se **Richieste$_i$ <= Disponibili** esegue il passo 3 altrimenti Pi deve attendere poiché le risorse non sono disponibili
3. il sistema simula l'assegnazione al processo Pi delle risorse richieste modificando come segue lo stato di assegnazione delle risorse:

   Disponibili := Disponibili - Richieste$_i$

   Assegnate$_i$:= **Assegnate$_i$** + **Richieste$_i$**

   Necessità$_i$:= **Necessità$_i$** - **Richieste$_i$**

se lo stato di assegnazione delle risorse risultante è sicuro la transazione è completata e al processo Pi si assegnano le risorse richieste

se il nuovo stato è non sicuro Pi deve attendere **Richieste$_i$** e si ripristina il vecchio stato di assegnazione delle risorse

continua..

Ritorna sopra | Home page | Xelon