PROGRAMMAZIONE CONCORRENTE

- Concetti di base
- Condizioni di Bernstein
- Costrutti per la programmazione concorrente: fork/join
- Proprieta' dei programmi concorrenti
- Problemi di programmazione concorrente: la mutua esclusione
- Altri problemi di programmazione concorrente
- Semafori
- Monitors

Testi:
Tanenbaum, I moderni sistemi operativi, cap.2
Ben-Ari, cap.2
Deitel, cap.4-5
Peterson e Silbershatz, cap.9-10

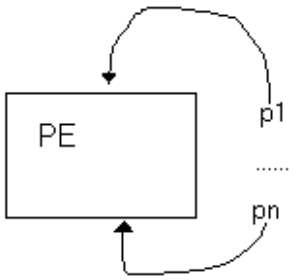
CONCETTI DI BASE

PROGRAMMA SEQUENZIALE: specifica l'esecuzione sequenziale di una lista di istruzioni. La sua esecuzione e' un PROCESSO.

PROGRAMMA CONCORRENTE: specifica due o piu' programmi sequenziali che possono essere eseguiti concorrentemente come processi paralleli.

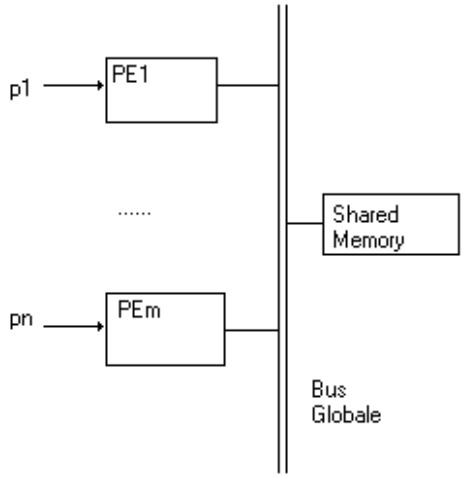
AMBIENTI DI ESECUZIONE

MULTIPROGRAMMAZIONE
Un solo processore, piu' processi

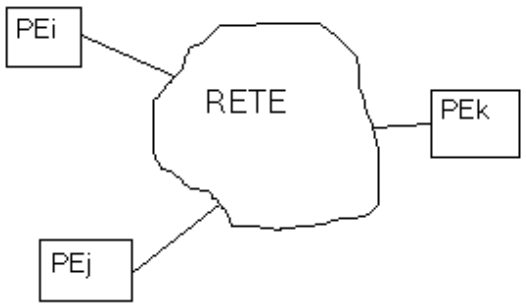


MULTIPROCESSING
Piu' processori su una memoria condivisa (shared memory)
Sia m numero dei processori, n numero dei processi, allora parliamo di:

- m>=n multiprocessing puro;
- m<n multiprocessing con multiprogrammazione



DISTRIBUTED PROCESSING
Piu' processori collegati da una rete di interconnessione (in generale privi di memoria condivisa). Su ogni processore e' possibile avere la multiprogrammazione



Interazione tra processi
Per poter cooperate, i processi concorrenti devono
- comunicare
- sincronizzarsi

Comunicazione: e' il modo attraverso il quale un processo influenza l'esecuzione di un altro processo. E' possibile comunicare attraverso:
- variabili condivise (shared variables)
- trasmissione di messaggi (message passing)

Le variabili condivise costituiscono un ambiente globale di elaborazione, che si realizza su una architettura a memoria comune.
Lo scambio di messaggi consente l'elaborazione in ambiente locale; puo' essere realizzato con o senza una memoria comune.

Sincronizzazione: e' necessaria per una comunicazione corretta.

Vi sono diversi fattori, tra cui l'ambiente di esecuzione mono- o multi-processore, che influenzano la **velocita'** con cui e' eseguito un processo, e la sua velocita' relativa rispetto ai processi con cui coopera.
E' quindi in generale molto difficile formulare ipotesi su tale velocita'.

E' comunque possibile e necessario formulare la seguente ipotesi:
FINITE PROGRESS ASSUMPTION: ***a tutti i processi di un programma concorrente e' garantito di poter progredire in un tempo finito, se almeno uno di essi e' pronto per l'esecuzione.***
PROGREDIRE va inteso

- in senso globale: completare la propria esecuzione
- in senso locale: terminare una sezione critica

Quando l'ipotesi di finite progress assumption viene violata, si verifica una situazione "patologica" di errore come ad es. deadlock, livelock, starvation.

NONDETERMINISMO

Supponiamo di avere un programma concorrente composto da n>2 processi.
Un processo raggiunge uno stato in cui e' disponibile per comunicare con m altri processi, m>1.
Tra gli m processi, potrebbe essercene piu' di uno contemporaneamente disponibile a tale comunicazione; ma in ogni istante non puo' avvenire piu' di una comunicazione per volta.

In generale non ci sono motivi per privilegiare una tra le m comunicazioni possibili; anzi, spesso imporre un ordine prestabilito a tali comunicazioni potrebbe causare ritardi (un processo e' pronto a comunicare ma quello che lo precede non ancora) e addirittura errori (deadlock o lockout).

E' utile poter esprimere la disponibilita' ad una scelta "nondeterministica".

Proprieta' di un programma concorrente

Safety: SE il programma termina, allora il suo risultato e' corretto. Simile alla "correttezza parziale" dei programmi sequenziali.

Liveness: il programma termina e il suo risultato e' corretto. Simile alla "correttezza totale" dei programmi sequenziali. In altri termini: la proprieta' di liveness esprime il fatto che se qualcosa deve accadere, allora effettivamente accadra' (eventually).

Minacce:

- Deadlock (risorse, comunicazioni)
- Starvation e lockout

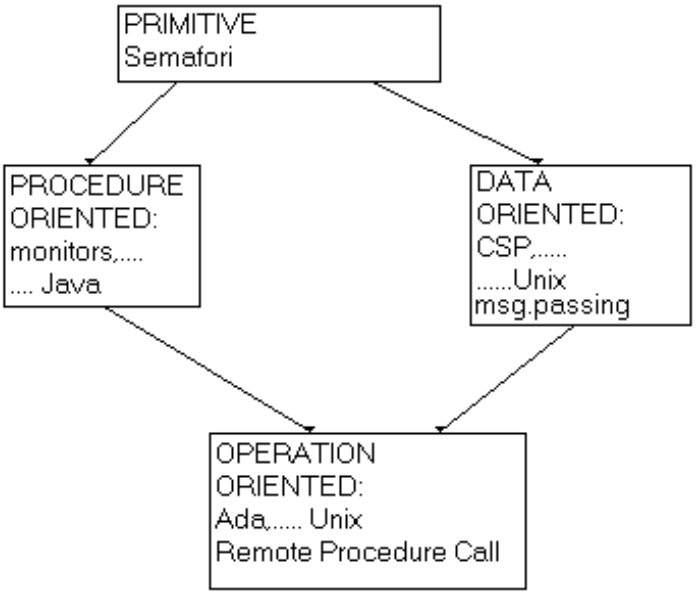
Aiuti:

- Meccanismi di fairness

Una notazione per la programmazione concorrente

- Come indicare l'esecuzione concorrente
- Quale modo di comunicazione tra processi utilizzare
- Quale meccanismo di sincronizzazione tra processi utilizzare
- Come esprimere e controllare il nondeterminismo
- Come poter verificare le proprieta' di un programma concorrente
- Come garantire l'efficienza del programma evitando attese infruttuose (busy waiting) o eccesssivo overhead.

Evoluzione delle notazioni per la programmazione concorrente



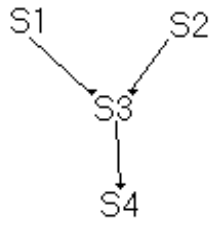
COSTRUTTI LINGUISTICI PER LA PROGRAMMAZIONE CONCORRENTE

- Ordinamenti parziali e grafi di precedenza
- Relazioni di precedenza e condizioni di Bernstein per ottenere la forma massimamente parallela
- Fork e Join
- Semafori
- Monitors

- Consideriamo un programma:

```
s1: a=x+y;
s2: b=z+1;
s3: c=a-b;
s4: w=c++;
```

Quali istruzioni possono essere eseguite in parallelo?
Occorre individuare delle relazioni di precedenza



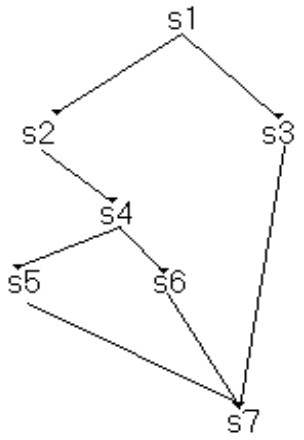
Ordinamenti totali ed ordinamenti parziali
Punto di vista sequenziale = ordinamento totale

P::{s_1; s_2; s_3; ... s_n;}
s_i statements del programma P;
l'insieme {s_i} e' totalmente ordinato, l'ordinamento e' dato dalla relazione di precedenza <.
s_i <. s_j significa che l'esecuzione di s_i precede l'esecuzione di s_j

Punto di vista concorrente = ordinamento parziale
Dato il programma P, possiamo passare dall'ordinamento totale a un ordinamento parziale dove si mantengono solo le relazioni di precedenza strettamente necessarie

Esempio:
supponiamo di avere le seguenti relazioni:
s1 <. s2
s1 <. s3
s2 <. s4
s4 <. s5
s4 <. s6
s5 <. s7
s6 <. s7
s3 <. s7
Naturalmente da queste si deducono per transitivita' anche altre relazioni, ad es. s1 <. s4, ma comunque non si ha un ordinamento totale; s2 e s3 ad es. non sono in relazione in alcun modo.

Possiamo rappresentare la situazione attraverso un grafo orientato aciclico detto GRAFO DI PRECEDENZA.



Un **grafo di precedenza** si ottiene
- inserendo un *nodo* per ogni operazione
- inserendo un *arco orientato* per ogni relazione di precedenza tra coppie di operazioni

Se c'e` un cammino orientato tra due nodi, allora vi e` una relazione di precedenza tra le operazioni corrispondenti (es. s1 e s7, s2 e s6, s1 e s5); altrimenti le due operazioni sono indipendenti (ad es. s2 e s3, s4 e s3, s3 e s5, s5 e s6).

Per la determinazione delle relazioni di precedenza tra operazioni (e quindi per la costruzione di un grafo di precedenza) abbiamo le seguenti

CONDIZIONI DI BERNSTEIN

Dato un programma P::{s_1; s_2; s_3; ... s_n;} supponiamo che ogni operazione s_i possa essere pensata come una "funzione" che preso un insieme di variabili in input (detto I_i), modifica il valore di un insieme di variabili di output (detto O_i):
s_i : I_i->O_i.

Condizione sufficiente perche` due operazioni si, sj siano indipendenti e` che valgano le tre condizioni seguenti (condizioni di Bernstein)

1. Ii Oj = O
2. Ij Oi = O
3. Oi Oj = O

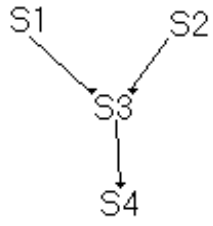
Esempio:
consideriamo la seguente sequenza di operazioni (per semplicita' sono tutte assegnazioni):
s1: `x = z - y;`
s2: `x = a + 2;`
Queste due istruzioni non sono indipendenti perche' non soddisfano la terza condizione di Bernstein:
infatti le variabili di output di s1 (insieme O1) sono le stesse delle variabili di output di s2 (insieme O2)
O1 = O2 = { `x` } e quindi O1 O2 = O
Di conseguenza le istruzioni s1 e s2 andranno eseguite nell'ordine e non in parallelo.

Consideriamo:
s1: `a = x + y;`
s2: `b = z + 1;`
s3: `c = a - b;`
s4: `w = c + 1;`
Calcoliamo gli insiemi di variabili di input e di output:

I1: { x , y }	O1: { a }
I2: { z }	O2: { b }
I3: { a , b }	O3: { c }
I4: { c }	O4: { w }

Controlliamo l'indipendenza di s1 e s2:
O1 ∩ I2 = O
I1 ∩ O2 = O
O1 ∩ O2 = O
quindi sono indipendenti;
controlliamo s1 e s3: abbiamo O1 ∩ I3 = { a } quindi s1 <. s3
controlliamo s2 e s3: abbiamo O2 ∩ I3 = { b } quindi s2 <. s3
controlliamo s3 e s4: abbiamo I4 ∩ O3 = { c } quindi s3 <. s4

Il grafo di precedenza risultante e` quindi:



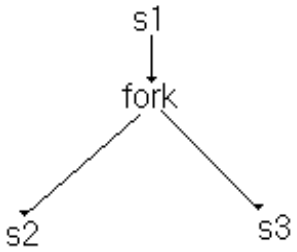
Specifica di una esecuzione concorrente all'interno di un programma

Vengono introdotte due primitive: `fork` e `join` (Conway, Dennis & Van Horn)

`fork` produce 2 esecuzioni concorrenti in un programma:
esempio:

```
      s1;
      fork L;
      s2;
      ...
L:    s3;
      ...
```

risulta nel grafo:

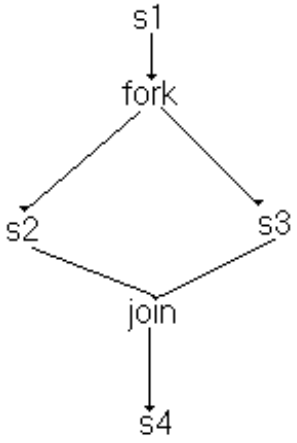


Quindi all'esecuzione di `fork`, parte una "nuova" computazione concorrente iniziante con s3, mentre la "vecchia" prosegue con s2.

`join` riunisce due (o piu') esecuzioni concorrenti
esempio:

```
     count = 2; /* istruzione "s1" */
     fork L1;
     s2;
     goto L2;
L1:  s3;
L2:  join count;
     s4;
```

risulta nel grafo:



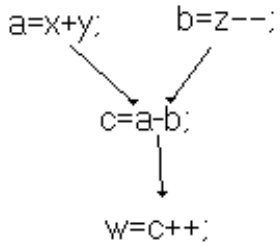
- `join` e` eseguita come una operazione indivisibile.
- La computazione che la esegue per prima termina, solo quella che la esegue per ultima prosegue.
- La variabile `count` tiene conto di quante computazioni si devono "riunire" (e quindi di quale sara` l'ultima che proseguira`); il valore iniziale di `count` e` appunto tale numero.
- L'esecuzione di `join count;` ha l'effetto:

```
count--;
if (count) termina il processo;
```

Esempio:
Scrivere un programma concorrente equivalente alla sequenza di istruzioni:

```
a = x + y;
b = z --;
c = a - b;
w = c++;
```

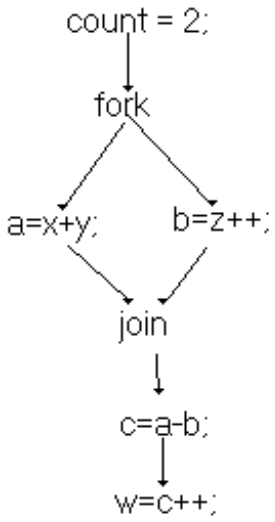
Il grafo di precedenza e`:



Un programma parallelo con `fork` e `join` e`:

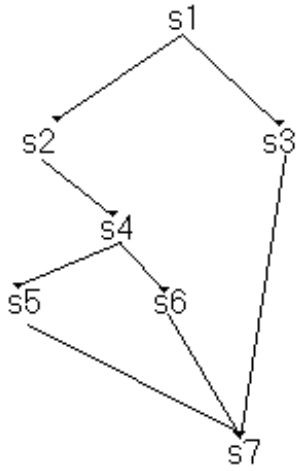
```
     count=2;
     fork L1;
     a=x+y;
     goto L2;
L1:  b=z--;
L2:  join count;
     c=a-b;
     w=c++;
```

che si puo' anche rappresentare con il grafo:



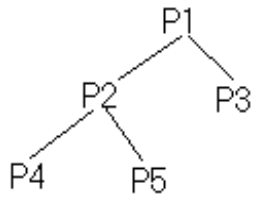
- Il grafo di precedenza fin qui utilizzato consente di individuare le possibili computazioni concorrenti.
- Possiamo pensare anche ad ogni nodo del grafo di precedenza come a un processo sequenziale.
- L'esecuzione di una `fork` da parte di un processo Pi implica la *creazione* di un altro processo Pj. In questo caso possiamo considerare Pi il "padre" di Pj e costruire una "gerarchia" di processi padre-figli (*albero dei processi*)
- L'esecuzione da parte di entrambi i processi Pi e Pj di una `join` corrisponde ad un punto di *sincronizzazione* tra i due processi, oltre il quale ne prosegue solo uno (in generale il padre).

Ad esempio si consideri il grafo:



Possiamo pensare a 5 processi:
- P1 esegue s1; da' origine ai processi figli P2 e P3, attende la terminazione dei figli e infine esegue s7
- P2 esegue s2; quindi s4 e attende la terminazione dei processi figli P4 e P5; dopodiche' termina
- P3 esegue s3 e termina
- P4 esegue s5 e termina
- P5 esegue s6 e termina

La gerarchia dei processi e`:

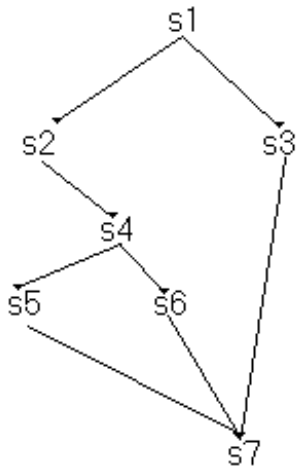


Esercizio proposto:
Scrivere un programma in C che utilizzando le primitive di Unix - Linux - Minix esegua in modo concorrente le seguenti operazioni:

```
lettura di valori per x,y,z;
a = x+y;
b = z--;
c = a-b;
w = c++;
scrittura dei valori di a,b,c,w;
```

Tracciarne anche il grafo di precedenza

Esercizio proposto:
Si consideri il seguente grafo di precedenza:

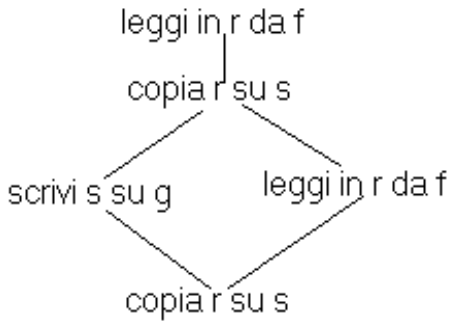


Scrivere un programma concorrente che usa fork e join, in cui l'istruzione `Si` produce in output il valore i. Implementarlo in C con le primitive di Unix-Linux-Minix.

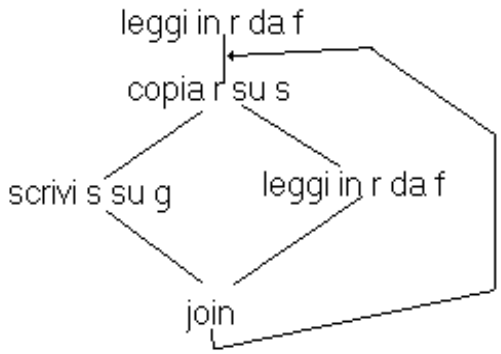
Esempio:

Scriviamo un programma che copia da un file sequenziale `f` ad un file sequenziale `g` utilizzando due diversi buffers (rispettivamente `r, s`).
E' possibile scrivere un programma in cui lettura e scrittura avvengono in modo concorrente. Ad esempio, dopo aver letto il primo blocco in `r,` lo si ricopia in `s` e si puo' leggere in `r` il secondo blocco contemporaneamente alla scrittura del primo blocco.

Il grafo di precedenza (per le prime due letture) sarebbe quindi:



Possiamo pensare a un ciclo `while` dentro al quale vengono eseguite le due operazioni concorrenti, mentre occorre una sincronizzazione prima di eseguire la copia.
Il grafo risultante e' quindi:



Il ciclo termina quando la lettura da `f` incontra la fine del file.

RACE CONDITIONS, MUTUA ESCLUSIONE E SOLUZIONI ELEMENTARI

Consideriamo due processi P_1 e P_2 eseguiti su un sistema multiprogrammato (quindi in interleaving).

I due processi condividono una risorsa, o una variabile (nell'esempio e' A).

- non tutti i possibili interleaving sono leciti (**race condition,** in italiano "corsa critica");
- l'accesso alla variabile o risorsa deve avvenire in modo mutuamente esclusivo.

Esempio:

```
P1:   .....                          P2:   .....
      A=A+1; /*istruzione A1*/             A=A+2; /*istruzione A2*/
      .....                                .....
```

Sappiamo che il valore iniziale di A e' 0. Ci aspettiamo che al termine dell'esecuzione di P_1 e P_2 il valore di A sia 3.

In realta' una assegnazione a variabile viene eseguita dalla macchina con una sequenza di istruzioni macchina elementari, ad esempio potrebbe essere la seguente:

--carico il valore di A in un registro;
--sommo una costante (1 oppure 2) al registro;
--trasferisco il valore del registro all'indirizzo di memoria di A.

L'interleaving si verifica a livello di istruzioni macchina, quindi e' possibile che al termine di una esecuzione "arbitraria" delle due asegnazioni il valore di A sia 1 oppure 2, invece del valore desiderato 3!

Diremo quindi che

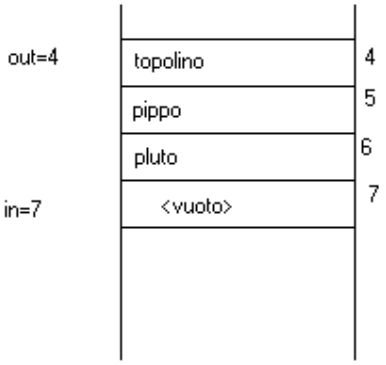
- ***P1 e P2 sono due processi concorrenti,***
- ***le istruzioni A1 e A2 devono essere eseguite in mutua esclusione;***
- ***le istruzioni A1 e A2 rappresentano due sezioni critiche.***

Esempio: spooler di stampa

Per la gestione di una sola stampante (risorsa condivisa) da parte di piu' processi solitamente i sistemi operativi usano un meccanismo detto spooler di stampa.
I processi non stampano direttamente il file ma richiedono la stampa di un file inserendo il nome del file in una apposita tabella detta "directory di spool". Possiamo supporre che questa tabella abbia dimensione "infinita", ovvero che sia sempre possibile inserire il nome di un ulteriore file da stampare, riempiendo successivamente le posizioni di indice 0, 1, 2, ecc.

Vi e' un processo detto "demone di stampa" che controlla il contenuto della directory di spool e, se c'e' il nome di un file, si incarica di stamparlo e di rimuovere quindi tale nome dalla directory.
In questo modo la stampante **NON** viene condivisa, ma e' riservata al demone di stampa. I vari processi, per poter inserire il nome dei files nella directory, devono conoscere l'indice delle posizione libera nella tabella (***in***), mentre il demone di stampa gestisce l'indice del file da stampare (***out***). Chiameremo la directory di spool ***slot***.

Consideriamo questa situazione:



Due processi P1 e P2 vogliono entrambi stampare "contemporaneamente", con le seguenti sequenze di istruzioni:

```
P1:   .....                                  P2:   .....
      index1=in; /*istr.1-1*/                      index2=in; /*istr.2-1*/
      slot[index1]="orazio"; /*1-2*/               slot[index2]="minni";/*2-2*/
      in=index1++; /*1-3*/                         in=index2++; /*2-3*/
      .....                                        .....
```

Supponiamo allora che l'interleaving tra i due processi faccia eseguire nell'ordine le istruzioni seguenti:

prima la 1-1, poi 2-1 e 2-2, poi 1-2 e 1-3, infine 2-3.

Quali files verranno stampati?

Le ***race conditions*** sono situazioni in cui, come nei due esempi visti, vi sono piu' processi che condividono variabili o risorse e in cui diversi interleaving portano a risultati diversi.

Le ***sezioni critiche*** di un processo concorrente sono le istruzioni (o gruppi di istruzioni) in cui il processo accede a risorse o variabili condivise.

Per evitare le race conditions occorre
- individuare le ***sezioni critiche*** di ogni processo concorrente;
- eseguire le sezioni critiche in ***mutua esclusione***.

Piu' precisamente, si vuole che:
- due o piu' processi concorrenti non si trovino contemporaneamente nelle rispettive sezioni critiche;
- nessun processo fermo fuori dalla sua sezione critica impedisca ad altri processi di procedere;
- nessun processo attenda un tempo infinito per entrare nella propria sezione critica;
- i tre criteri precedenti siano sempre validi qualunque sia la velocita' relativa dei processi.

La soluzione al problema precedente si ottiene facendo precedere e seguire tutte le sezioni critiche dei vari processi da opportuni <prologhi> ed <epiloghi>:

```
P1:   .....
      <prologo>
      sezione critica di P1;
      <epilogo>
      .....
```

```
P2:   .....
      <prologo>
      sezione critica di P2;
      <epilogo>
      .....
```

Come scrivere il <prologo> e l' <epilogo> per avere la mutua esclusione sulle regioni critiche????

Possibili soluzioni:

DISABILITARE LE INTERRUZIONI

- Quando un processo entra nella propria regione critica, disabilita le interruzioni
- Quando esce, riabilita le interruzioni.

Problema:
e se il processo (utente) non riabilita piu' le interruzioni???
Il sistema cessa di funzionare!

Disabilitare ed abilitare le interruzioni e' compito SOLO del kernel

VARIABILI DI LOCK

In memoria condivisa c'e' la variabile di lock L inizializzata a 0.

P legge L:
- Se vale 0 puo' entrare nella regione critica, la setta a 1 ed entra.
- Se vale 1 aspetta che torni 0 e procede come sopra.

===> stesso problema che per lo spooler: un altro processo Q potrebbe leggerla a 0 e settarla a 1 prima di Q.
Si potrebbe allora:

- leggere L
- se L=0 allora la rileggo, e
- se e' ancora L=0 la setto a 1 altrimenti attendo

===> e' ancora possibile che Q setti L a 1 dopo la seconda lettura di 0 !!!

Mutua esclusione tra due processi P1 e P2

Lo scopo e' di scrivere l'arbitraggio della memoria condivisa.
Vedremo quattro diversi "tentativi di soluzione", cioe' programmi che risolvono solo parzialmente la mutua esclusione, ma che "a prima vista" possono sembrare corretti.

Primo tentativo: uso della variabile condivisa `turn`, per indicare quale processo puo' entrare nella propria sezione critica.

- garantisce la mutua esclusione
- non da' deadlock
- garantisce in parte liveness
- non da' lockout
- non da' abbastanza concorrenza!!! provare ad esempio con due processi con rapporto di velocita' 1/100
- non funziona se un processo termina o abortisce

Secondo tentativo: ogni processo ha la propria chiave di accesso alla sezione critica (le variabili condivise `c1` e `c2`).

- non garantisce la mutua esclusione

Terzo tentativo: sempre con le variabili condivise `c1` e `c2` ma a controlli invertiti.

- garantisce la mutua esclusione
- si puo' avere deadlock

Quarto tentativo: un processo che non riesce ad entrare nella propria sezione critica rinuncia temporaneamente al tentativo.

- garantisce la mutua esclusione
- non c'e' deadlock
- non c'e' liveness
- si puo' avere lockout

Primo tentativo

```
shared int turn;
process P1
while (true)
{
     while (turn==2);/*busy wait*/
     regione_critica_1;
     turn=2;
     regione_non_critica_1;
}
process P2
while (true)
{
     while (turn==1);/*busy wait*/
     regione_critica_2;
     turn=1;
     regione_non_critica_2;
}
main
{
turn=1;
<facciamo partire P1 e P2 in parallelo>
}
```

- La mutua esclusione e' garantita
- Non c'e' abbastanza concorrenza: infatti un processo non puo' entrare nella propria sezione critica anche se volesse (e potesse): deve comunque aspettare che l'altro entri ed esca dalla propria sezione critica. Considerare ad esempio il caso in cui P1 aspetta nel ciclo `while turn==2);` e P2 sta eseguendo la funzione `regione_non_critica_2;` per un tempo relativamente lungo
- Istruzioni come quelle commentate con `/*busy wait*/` occupano la CPU senza motivo (inefficienza)

Secondo tentativo

```
shared int c1,c2;
process P1
while (true)
{
     while (c2==0);/*busy wait*/
     c1=0;
     regione_critica_1;
     c1=1;
     regione_non_critica_1;
}
process P2
while (true)
{
     while (c1==0);/*busy wait*/
     c2=0;
     regione_critica_2;
     c2=1;
     regione_non_critica_2;
}
main
{
 c1=1;
 c2=1;
<facciamo partire P1 e P2 in parallelo>
}
```

In questo caso il parallelismo e' aumentato rispetto al caso precedente, ma non rispetta piu' la mutua esclusione!
(ad esempio nel caso in cui per l'interleaving si esegue una istruzione per processo a turno)

Terzo tentativo

```
shared int c1,c2;
process P1
while (true)
{
     c1=0;
     while (c2==0);/*busy wait*/
     regione_critica_1;
     c1=1;
     regione_non_critica_1;
}
process P2
while (true)
{
     c2=0;
     while (c1==0);/*busy wait*/
     regione_critica_2;
     c2=1;
     regione_non_critica_2;
}
main
{
 c1=1;
 c2=1;
<facciamo partire P1 e P2 in parallelo>
}
```

In questo caso e' garantita la mutua esclusione, ma esiste la possibilita' di deadlock! (ad esempio se per l'interleaving vengono eseguite a turno una istruzione per processo, entrambi si bloccano sui rispettivi busy wait)

Quarto tentativo

```
shared int c1,c2;
process P1
while (true)
{
     c1=0;
     while (c2==0)
     {
        c1=1;
        delay();
        c1=0;
     }
     regione_critica_1;
     c1=1;
     regione_non_critica_1;
}
process P2
while (true)
{
     c2=0;
     while (c1==0)
     {
        c2=1;
        delay();
        c2=0;
     }
     regione_critica_2;
     c2=1;
     regione_non_critica_2;
}
main
{
 c1=1;
 c2=1;
<facciamo partire P1 e P2 in parallelo>
}
```

In questo caso si evita la situazione di deadlock assoluto del tentativo precedente, ma i processi possono rimanere "per sempre" al di fuori della propria sezione critica (ad es. se l'interleaving facesse eseguire a turno le istruzioni del loop piu' interno, da cui nessuno dei due riuscirebbe ad uscire)

ALGORITMO DI DEKKER

```
shared int turn,c1,c2;
process P1
while (true)
{    c1=0;
     while (c2==0)
      if (turn==2)
      { c1=1;
        while (turn==2); /*busy wait */
        c1=0;
     }
     regione_critica_1;
     c1=1; turn=2;
     regione_non_critica_1;
}
process P2
while (true)
{
     c2=0;
     while (c1==0)
      if (turn==1)
      { c2=1;
        while (turn==1); /*busy wait */
        c2=0;
     }
     regione_critica_2;
     c2=1; turn=1;
     regione_non_critica_2;
}
main
{turn=1; c1=1; c2=1;
<facciamo partire P1 e P2 in parallelo>
}
```

Osservazioni:

- Questo algoritmo "immerge" il primo e il quarto tentativo (notare l'uso di `turn` diverso da quello del primo tentativo)
- Un processo puo' insistere nel tentativo di entrare nella sezione critica: cio' evita il lockout del quarto tentativo
- Ciascun processo puo' entrare nella sezione critica anche se l'altro e' terminato

ALGORITMO DI PETERSON (1981)

I processi hanno un numero (0 oppure 1)
Prima di entrare nella sezione critica, il processo chiama la funzione `enter_region()` con il proprio numero come parametro
Al termine della sezione critica, il processo chiama `leave_region()`

```
#define TRUE 1
#define FALSE 0
#define N 2 /* numero dei processi */

int turn;
int interested[N]; /* tutti inizialmente a 0 */

void enter_region (int process)
{ int other;
 other=1-process; /* l'altro processo */
 interested[process]=TRUE;
 turn=process;
 while(turn==process && interested[other]==TRUE);
    /*attesa */
}

void leave_region (int process)
{
  interested[process]=FALSE;
}
```

Se i due processi chiamano contemporaneamente la `enter_region()`, uno dei due (ad es. il numero 1) eseguira' l'assegnazione a `turn` dopo l'altro (cancellando lo 0): allora il processo 0 proseguira' e il processo 1 si blocchera' sul `while` finche' lo 0 non chiamera' `leave_region()`.

ISTRUZIONE TSL (test-and-set-lock)

Su molti calcolatori esiste una istruzione macchina detta TSL che funziona in modo atomico (cioe' non interrompibile) nel modo seguente:

- l'operando (variabile in memoria) e' copiato in un registro,
- un valore non-zero viene messo al posto dell'operando.

Possiamo allora riscrivere la enter_region e la leave_region in assembler:

```
enter_region:
     tsl  register,flag
     cmp  register,#0
     jnz  enter_region
     ret

leave_region:
     mov  flag,#0
     ret
```