CAPITOLO 6
PROCESS SYNCHRONIZATION

6.1: Interazioni tra processi

6.2: Corse critiche (race conditions)

6.3: Sezioni critiche

6.4: Problemi classici

6.5: Approccio algoritmico alle sezioni critiche

6.6: Semafori

6.7: Monitor

6.1 INTERAZIONI TRA PROCESSI

Su un sistema uniprocessore, programmare un'applicazione con un insieme di processi/thread concorrenti può dare i seguenti vantaggi:

- **semplicità** di programmazione, nel caso in cui l'applicazione preveda più compiti;

- **efficienza**, se si ha parallelismo tra attività I/O ed attività di CPU;

- possibilità di assegnare **compiti più urgenti** a processi/thread con **priorità più elevata**, se lo scheduler si basa sulle priorità.

Se il processore è multithreading, oppure se il sistema è multiprocessore, i vantaggi in termini di efficienza sono ancor più evidenti.

Nella discussione che segue assumiamo che più processi abbiano una zona di memoria condivisa. Abbiamo detto nel Cap. 4 che i processi hanno memoria separata, ma:

- i S.O. consentono ai processi di accedere a strutture dati del kernel, che risultano quindi condivise dai processi;

- ragionare in termini di memoria condivisa è più facile che ragionare in termini di file condivisi, ed i problemi che analizzeremo sono i medesimi;

- se anziché trattare con i processi trattassimo con i thread l'assunzione sarebbe coerente con quanto trattato nel Cap. 5.

Esempio di memoria condivisa: un sistema banale di prenotazioni aeree usa **n** terminali connessi ad un computer centrale. Abbiamo un processo per ogni terminale. I processi condividono due variabili:

- **next_seat**: indica il prossimo posto da assegnare;
- **max**: indica il numero totale di posti sul volo.

Tutti gli **n** processi eseguono lo stesso programma per prenotare un posto su un volo:

```
…….  /* N.B. questo programma ha dei problemi che vedremo!!! */
if (next_seat <= max) {
   booked = next_seat; /* booked è una variabile privata del processo*/
   next_seat ++;
 }
else{
    printf("sorry, flight sold out")
 }
 ……..
```

Abbiamo 4 tipi di interazioni tra processi:

- **data sharing**: i processi condividono i dati della memoria condivisa ed alcuni file.

- **control synchronization**: un'azione $\mathbf{a_i}$ di un processo $\mathbf{P_i}$ è abilitata solo dopo che un altro processo $\mathbf{P_j}$ ha svolto un azione $\mathbf{a_j}$.

- **message passing**: un processo $\mathbf{P_i}$ invia un messaggio ad un processo $\mathbf{P_j}$ che lo riceve.

- **signals**: un processo $\mathbf{P_i}$ invia un segnale ad un altro processo $\mathbf{P_j}$ per segnalare una situazione particolare.

Nella discussione che segue trattiamo con i processi, ma potremmo indifferentemente ragionare con i thread.

Dato un processo **P_i**, siano:

- **R_i** i dati letti dal processo **P_i** (variabili/file cui si accede in lettura, messaggi/segnali ricevuti da altri processi, ……). L'insieme **R_i** è detto **read_set** di **P_i**;

- **W_i** i dati modificati dal processo **P_i** (variabili/file cui si accede in scrittura, messaggi/segnali inviati ad altri processi, ……). L'insieme **W_i** è detto **write_set** di **P_i.**

Definizione: Due processi concorrenti $\mathbf{P_i}$ e $\mathbf{P_j}$ sono **processi interagenti** se vale almeno una delle seguenti due proprietà:

- $\mathbf{R_i}$ e $\mathbf{W_j}$ hanno intersezione non vuota,
- $\mathbf{R_j}$ e $\mathbf{W_i}$ hanno intersezione non vuota.

Definizione: Due processi concorrenti $\mathbf{P_i}$ e $\mathbf{P_j}$ sono **processi indipendenti** se non sono interagenti.

Osservazione: queste due definizioni valgono anche se $\mathbf{P_i}$ e $\mathbf{P_j}$ sono thread.

Nell'esempio delle prenotazioni aeree, gli **n** processi sono due a due interagenti, perché leggono e modificano la variabile **next_seat**. (**next_seat** è nel read_set e nel write_set di ogni processo.)

Se due processi/thread concorrenti $\mathbf{P_i}$ e $\mathbf{P_j}$ sono indipendenti:

- competono per le risorse (e.g. CPU), rallentadosi a vicenda,
- i loro comportamenti non dipendono dalla loro velocità relativa e sono riproducibili.

Se due processi/thread $\mathbf{P_i}$ e $\mathbf{P_j}$ sono interagenti:

- competono per le risorse (e.g. CPU), rallentadosi a vicenda,
- i loro comportamenti dipendono dalla loro velocità relativa e non sono riproducibili.

Per esempio, se $\mathbf{P_i}$ e $\mathbf{P_j}$ sono due degli **n** processi interagenti dell'esempio precedente, le prenotazioni di un processo influiscono sulle prenotazioni dell'altro. Quindi l'esecuzione di $\mathbf{P_i}$ non è riproducibile, in quanto dipende dalla velocità relativa di $\mathbf{P_i}$ e $\mathbf{P_j}$ e degli altri processi.

Esempio. Sia **x** una variabile **int** condivisa inizializzata a **100**.

Programma di P_i

```
.............
int y = x+5;
printf("%d",y);
.............
```

Programma di P_j

```
.............
x = 10;
.............
```

Il comportamento di P_i non è riproducibile, nel senso che eseguendo P_i e P_j più volte, vale che:

- a volte P_i stampa 105;
- a volte P_i stampa 15.

Il risultato dipende dalle politiche di scheduling, dalle priorità di P_i, P_j e degli altri processi, da quanti e quali altri processi sono in esecuzione, …..

Vedremo nel prossimo paragrafo alcuni esempi concreti, usando le thread POSIX sotto Linux.

Osservazione informale.

Quando abbiamo processi concorrenti, è inevitabile e perfettamente accettabile, e a volte desiderabile, che i loro comportamenti non siano riproducibili.

Questa situazione non va mai confusa con le race condition che introdurremo nel prossimo paragrafo.

Esempio: è perfettamente accettabile che se da un terminale si tenta di effettuare un numero di prenotazioni pari al numero di posti totali del volo solo alcune vadano a buon fine: siamo consapevoli dell'esistenza di altri centri di prenotazione.

6.2 RACE CONDITIONS

Iniziamo con un esempio in cui più thread condividono una variabile. Questo esempio **NON** presenta problemi.

```c
#include <pthread.h> /* Esempio 1: variabile condivisa da 3 thread */
#include <stdio.h>
#include <stdlib.h>
int x; /* x variabile globale condivisa */

void * f1(void *threadid){
   x=100; /* modifica di x */
   pthread_exit(NULL);
}

void * f2(void *threadid){
   int y = x+5; /* lettura di x */
   printf("y vale %d\n",y);
   pthread_exit(NULL);
}

int main( ){ /* x è nel write_set */
   pthread_t t1,t2;
   x=10; /* modifica di x */
   pthread_create(&t1, NULL, f1, (void *)t1); /* x è nel write_set */
   pthread_create(&t2, NULL, f2, (void *)t2); /* x è nel read_set */
   pthread_exit(NULL);}
```

Abbiamo 2 possibili risultati:

```
simone$a.out
y vale 15
simone$


simone$a.out
y vale 105
simone$
```

mette x = 100+5 e il resto non lo esegue

mette x = 10 + 5 e il resto non lo esegue

Nell'esempio precedente il valore della variabile **x** che viene stampato a video dipende dalla *velocità di esecuzione* ***relativa*** dei due thread.

In altre parole, non possiamo prevedere e/o controllare quale operazione venga fatta per prima tra

- la modifica di **x** da parte del thread che esegue **f1** e
- la lettura di **x** da parte del thread che esegue **f2**.

Questo è **perfettamente accettabile**.

Nell'esempio della prossima slide usiamo la funzione POSIX **pthread_join**, che serve per implementare le process synchronization.

Se un thread esegue il codice seguente:

istruzione A;

pthread_join(t, p); /* attendo la terminazione di t */

istruzione B;

con **t** un **pthread_t** e **p** un pointer, accade che:

- quando esegue la **pthread_join**, il thread va in waiting. L'evento atteso è la terminazione del thread **t**;
- quando il thread **t** termina, il thread in waiting torna in ready, pronto ad eseguire l'istruzione B. Se **p** non è **null**, allora nella locazione puntata da **p** viene messo il codice di terminazione di **t**. Su questo ultimo punto non indagheremo ulteriormente.

```
#include <pthread.h> /* Esempio 2: variabile condivisa da 3 thread */
#include <stdio.h>
#include <stdlib.h>
int x = 100; /* x variabile globale variabile condivisa */

void * f1(void *threadid){
   x=x+5; /* modifica di x */
   pthread_exit(NULL);
}

void * f2(void *threadid){
   x = x+10; /* modifica di x */
   pthread_exit(NULL);
}

int main( ){ /* x è nel read_set */
   pthread_t t1, t2;
   pthread_create(&t1, NULL, f1, (void *)t1); /* x è nel write_set */
   pthread_create(&t2, NULL, f2, (void *)t2); /* x è nel write_set */
   pthread_join(t1,NULL); /* attendo che termini il primo thread */
   pthread_join(t2,NULL); /* attendo che termini il secondo thread */
   printf("x vale %d\n",x); /* lettura di x */
}
```

Ci si *aspetterebbe* un solo risultato:

```
simone$a.out
x vale 115
simone$
```

Ma, in realtà:

```
simone$a.out
x vale 115
simone$a.out
x vale 110
simone$a.out
x vale 105
simone$a.out
```

Cosa succede????

```
x = x+5;  →   l1: load  X,  R0
              a1: add   5,  R0
              s1: store R0, X

x = x+10; →   l2: load  X,  R0
              a2: add   10, R0
              s2: store R0, X
```

Per chiarirlo, anziché ragionare con le istruzioni in **C**, ragioniamo con le istruzioni macchina.

Probabilmente le istruzioni **x=x+5** e **x=x+10** sono realizzate con sequenze **load-add-store** come quelle sopra riportate, dove **X** è l'indirizzo della variabile **x** e **R0** è un registro generale.

Vediamo le possibili esecuzioni, dovute agli interrupt / scheduling.

x = x+5; →

l1: load X, R0
a1: add 5, R0
s1: store R0, X

x = x+10; →

l2: load X, R0
a2: add 10, R0
s2: store R0, X

l1, a1, s1, l2, a2, s2 → risultato 115.
l2, a2, s2, l1, a1, s1 → risultato 115.

l1, a1, l2, a2, s2, s1 → risultato 105.
l1, l2, a1, a2, s2, s1 → risultato 105.
l1, l2, a2, a1, s2, s1 → risultato 105.
l1, l2, a2, s2, a1, s1 → risultato 105.
l2, l1, a1, a2, s2, s1 → risultato 105.
l2, l1, a2, a1, s2, s1 → risultato 105.
l2, l1, a2, s2, a1, s1 → risultato 105.
l2, a2, l1, a1, s2, s1 → risultato 105.
l2, a2, l1, s2, a1, s1 → risultato 105.

l2, a2, l1, a1, s1, s2 → risultato 110.
l2, l1, a2, a1, s1, s2 → risultato 110.
l2, l1, a1, a2, s1, s2 → risultato 110.
l2, l1, a1, s1, a2, s2 → risultato 110.
l1, l2, a2, a1, s1, s2 → risultato 110.
l1, l2, a1, a2, s1, s2 → risultato 110.
l1, l2, a1, s1, a2, s2 → risultato 110.
l1, a1, l2, a2, s1, s2 → risultato 110.
l1, a1, l2, s1, a2, s2 → risultato 110.

Torniamo all'esempio delle prenotazioni aeree, e guardiamo una possibile traduzione in codice macchina.

```
S1   if (next_seat <= max) {

S2       booked = next_seat;

S3       next_seat ++;
     }
     else{

S4       printf("sorry, sold out")
     }
S5   ……
```

```
S1,1   load     max, R0
S1,2   sub      R0, next_seat
S1,3   jmpneg   R0, S4,1

S2,1   move     next_seat, booked

S3,1   load     next_seat, R1
S3,2   add      R1, 1
S3,3   store    R1, next_seat
S3,4   jmp      S5,1

S4,1   move     "sorry", Rvideo

S5,1   ……
```

Siano **max = 200**, **next_seat = 200**. Abbiamo un solo posto libero. I processi P_1 e P_2 tentano di prenotare il volo. Esaminiamo 3 casi.

Time	Caso 1 P_1	Caso 1 P_2	Caso 2 P_1	Caso 2 P_2	Caso 3 P_1	Caso 3 P_2
1	$S_{1,1}$		$S_{1,1}$		$S_{1,1}$	
2	$S_{1,2}$		$S_{1,2}$		$S_{1,2}$	
3	$S_{1,3}$		$S_{1,3}$		$S_{1,3}$	
4	$S_{2,1}$			$S_{1,1}$	$S_{2,1}$	
5	$S_{3,1}$			$S_{1,2}$	$S_{3,1}$	
6	$S_{3,2}$			$S_{1,3}$		$S_{1,2}$
7	$S_{3,3}$			$S_{2,1}$		$S_{1,2}$
8	$S_{3,4}$			$S_{3,1}$		$S_{1,3}$
9		$S_{1,1}$		$S_{3,2}$		$S_{2,1}$
10		$S_{1,2}$		$S_{3,3}$		$S_{3,1}$
11		$S_{1,3}$		$S_{3,4}$		$S_{3,2}$
12		$S_{4,1}$	$S_{2,1}$			$S_{3,3}$
13			$S_{3,1}$			$S_{3,4}$
14			$S_{3,2}$		$S_{3,2}$	
15			$S_{3,3}$		$S_{3,3}$	
16			$S_{3,4}$		$S_{3,4}$	

Caso 1 - tutto ok: P_1 prenota ($S_{2,1}$), P_2 no.

Caso 2 – not ok: P_2 prenota il posto 200, P_1 prenota il posto 201, che non esiste!! **next_seat** assume il valore 202.

Caso 3 – not ok: Entrambi P_1 e P_2 prenotano il posto 200!! **next_seat** assume valore 201.

Definizione:

Sia **d** un dato condiviso dai processi P_1 e P_2.

Siano o_1,o_2 operazioni su **d** eseguite da P_1, P_2, rispettivamente.

Siano f_1, f_2 funzioni tali che, se **d** ha valore v_d:

- $f_1(v_d)$ è il valore assunto da **d** eseguendo o_1;
- $f_2(v_d)$ è il valore assunto da **d** eseguendo o_2.

Le operazioni o_1 e o_2 danno luogo ad una **race condition** (**corsa critica**) sul dato condiviso **d** se, eseguendo o_1 e o_2 con **d** avente un valore iniziale v_d, accade che **d** assume un valore v'_d diverso da $f_1(f_2(v_d))$ e da $f_2(f_1(v_d))$.

```
#include <pthread.h> /* Esempio 2: variabile condivisa da 3 thread */
#include <stdio.h>
#include <stdlib.h>
int x = 100; /* x variabile globale variabile condivisa */

void * f1(void *threadid){
   x=x*5; /* modifica di x */
   pthread_exit(NULL);
}

void * f2(void *threadid){
   x = x+10; /* modifica di x */
   pthread_exit(NULL);
}

int main( ){ /* x è nel read_set */
   pthread_t t1, t2;
   pthread_create(&t1, NULL, f1, (void *)t1); /* x è nel write_set */
   pthread_create(&t2, NULL, f2, (void *)t2); /* x è nel write_set */
   pthread_join(t1); /* attendo che termini il primo thread */
   pthread_join(t2); /* attendo che termini il secondo thread */
   printf("x vale %d\n",x); /* lettura di x */
}
```

Ci si *aspetterebbero* due risultati:

simone$a.out
x vale 510
simone$a.out
x vale 550

Ma, in realtà:

simone$a.out
x vale 500
simone$a.out
x vale 510
simone$a.out
x vale 110
simone$a.out
X vale 550

Nell'esempio a Pag. 14:

l'operazione $\mathbf{o_1}$ è l'istruzione **x=x+5**,
l'operazione $\mathbf{o_2}$ è l'istruzione **x=x+10**,
$\mathbf{f_1}$ è la funzione tale che $\mathbf{f_1(z)=z+5}$
$\mathbf{f_2}$ è la funzione tali che $\mathbf{f_2(z)=z+10}$
$\mathbf{f_1(f_2(100)) = 115}$
$\mathbf{f_2(f_1(100)) = 115}$. (La somma è commutativa…)

Nell'esempio a Pag. 23:

l'operazione $\mathbf{o_1}$ è l'istruzione **x=x*5**,
l'operazione $\mathbf{o_2}$ è l'istruzione **x=x+10**,
$\mathbf{f_1}$ è la funzione tale che $\mathbf{f_1(z)=z*5}$ e
$\mathbf{f_2}$ è la funzione tali che $\mathbf{f_2(z)=z+10}$,
$\mathbf{f_1(f_2(100)) = 550}$,
$\mathbf{f_2(f_1(100)) = 510}$.

- Esempio: nel caso 3 dell'esempio delle prenotazioni aeree, le due istruzioni **next_seat++** danno luogo ad una race condition su **next_seat**.

- Esempio: nel caso 2 dell'esempio delle prenotazioni aeree, sono le due istruzioni **if-else** a dar luogo ad una race condition su **next_seat.**

N.B.: eseguendo più volte questi programmi, si osserva che a volte si verificano le race condition, mentre a volte no.

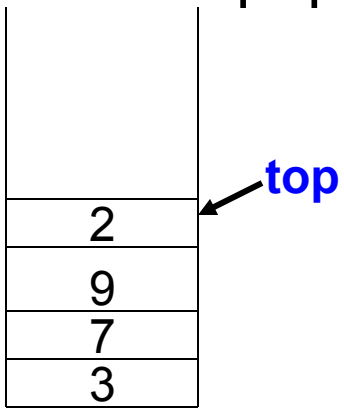
Fare il debugging di questi programmi non è pertanto semplice.

Le race condition hanno due conseguenze:

- il comportamento dei processi/thread non è corretto;
- i dati cu cui ha luogo la race condition diventano **inconsistenti**, cioè assumono uno stato non previsto. Esempio:

  - nel caso 2 dell'esempio delle prenotazioni aeree **next_seat** assume il valore 202, che non verrebbe mai assunto in assenza di race condition.

Un altro esempio: due processi usano la medesima pila, un processo fa la push, l'altro la pop.




```
.....
a  top=top+1;
b  stack[top]=x;
.....
```

```
.....
c  y=stack[top];
d  top=top-1;
.....
```

- Istruzioni eseguite in ordine **a**, **c**, **d**, **b**:

  - **y** assume il valore di qualcosa che non è nella pila;

  - **x** viene inserito al posto dell'elemento **2** che era al **top** e che nessuno ha letto.

- Se **a** è implementata con

  a_1: load top,R_0; a_2: add R_0,1; a_3: store R_0, top;

  e l'ordine è a_1, **c**, **d**, a_2, a_3, **b**, l'elemento **x** viene inserito due posizioni sopra il **9**, dal punto di vista logico la pila ha un "gap". Il processo vede uno stato inconsistente della pila.

Come prevenire le race condition?

Gli accessi ai dati condivisi devono essere sincronizzati, cioè devono avvenire in **mutua esclusione**: quando un processo accede ad un dato condiviso **d**, nessun altro processo deve poter accedere concorrentemente a **d**.

6.3 SEZIONI CRITICHE

Le race condition su un dato condiviso **d** hanno luogo perché i processi accedono a **d** concorrentemente.

Definizione: una **sezione critica** (**critical section** - **CS**) per un dato condiviso **d** è una porzione di codice che viene certamente eseguita non concorrentemente con se stessa o con altre sezioni critiche per **d**. (Problema: come si implementa???)

Si parla anche di **mutua esclusione**, nel senso che le CS su un dato **d** sono mutualmente esclusive.

Se i processi modificano il dato **d** solo nelle CS per **d**, allora le race condition su **d** non possono aver luogo.

Se i processi accedono in lettura al dato **d** solo nelle CS per **d**, allora i processi vedono solo stati consistenti di **d**.

Nell'esempio delle prenotazioni aeree, le race condition sono evitate se le istruzioni S_1, S_2, S_3 formano una CS:

Processo P_1

```
S1  if (next_seat <= max) {

S2      booked = next_seat;

S3      next_seat ++;
    }
    else{

S4      printf("sorry, sold out")
    }
S5  ……
```

Processo P_2

```
S1  if (next_seat <= max) {

S2      booked = next_seat;

S3      next_seat ++;
    }
    else{

S4      printf("sorry, sold out")
    }
S5  ……
```

Prima di vedere come si possano implementare le CS, osserviamo che i processi non dovrebbero rimanere per tanto tempo nelle CS, perché così facendo impedirebbero ad altri processi di eseguire e le prestazioni del sistema verrebbero degradate:

- il S.O. dovrebbe evitare la preemption di un processo che sta eseguendo una CS. Ciò è difficile da implementare, perché il S.O. dovrebbe tener traccia di quando i processi eseguono le CS;

- all'interno delle CS non dovrebbero esserci system call che possano causare il passaggio del processo allo stato di waiting.

Proprietà richieste alle implementazioni delle CS:

- **correttezza**: le CS per un dato **d** non possono essere eseguite concorrentemente, cioè deve essere garantita la mutua esclusione;

- **progress**: se nessun processo sta eseguendo una CS per **d**, e alcuni processi manifestano la volontà di eseguire CS per **d**, allora uno di essi deve poter eseguire la propria CS per **d**;

- **bounded wait**: dopo che un processo $\mathbf{P_i}$ manifesta la volontà di accedere ad una CS per **d**, il numero di accessi alle CS per **d** da parte di un qualsiasi altro processo $\mathbf{P_j}$ che precedono l'accesso di $\mathbf{P_i}$ deve essere <= di un dato intero **k**.

Il progress e la bounded wait prevengono la **starvation**, cioè l'attesa infinita da parte dei processi.

Implementare le CS - tentativo errato 1 – disabilitare gli interrupt.

```
.....
disable_interrupt; /* istruzione che modifica i bit IM della PSW */
{CS}; /* notazione che useremo per indicare una generica CS*/
enable_interrupt; /* istruzione che modifica i bit IM della PSW */
.....
```

Problemi:

- la soluzione è corretta solo su sistemi uniprocessore;
- la soluzione richiede che le istruzioni per disabilitare gli interrupt siano eseguibili in user mode: un programma potrebbe usare queste istruzioni per monopolizzare la CPU e non per eseguire le CS.

Implementare le CS - tentativo errato 2 – uso di *variabili lock* condivise.

Sia **lock** una variabile condivisa inizializzata a **0**.

```
.....................
while (lock != 0){  }  /* lock = 0 means lock open */
lock=1;                /* lock = 1 means lock closed */
{CS}
lock=0;
.....................
```

Questa soluzione non è corretta, perché la race condition può aver luogo sulla variabile **lock**: più processi potrebbero trovare **lock** = 0 (istruzioni **load lock,R** eseguite da più processi prima dell'aggiornamento di **lock**), porre **lock** ad 1 ed entrare nella CS.

Osservazione: abbiamo tentato di usare un dato condiviso per risolvere un problema sui dati condivisi …….

Implementare le CS - tentativo errato 3 – uso di variabili turno condivise. Vediamo il caso semplice, con due soli processi.

Processo P_0

```
……..
while (turn != 0){ }
{CS}
turn = 1;
……..
```

Processo P_1

```
……..
while (turn != 1){ }
{CS}
turn = 0;
……..
```

Questa soluzione è corretta ma non soddisfa la proprietà del progresso: se **turn** vale inizialmente **0** e P_1 è il primo processo a voler entrare nella CS, non riesce a farlo, nonostante P_0 non sia nella propria CS. Se P_0 non entra mai nella propria CS allora P_1 è bloccato all'infinito: abbiamo starvation.

Definizione: un'**operazione indivisibile** su un dato condiviso **d** è un'operazione che è con certezza eseguita in modo non concorrente rispetto ad altre operazioni su **d**.

Un'operazione indivisibile viene talvolta detta **atomica**.

Per definizione, le operazioni indivisibili su **d** non possono dar luogo a race condition su **d**.

Istruzione Test and Set (**TS instruction**).

L'istruzione TS, proposta originalmente per l'IBM/370, esegue due azioni:

1. controlla se una locazione di memoria ha valore **0**, ponendo il risultato nel bit **CC** (**C**ondition **C**ode) della **PSW**;
2. imposta la locazione di memoria con una sequenza di **1**.

Gli interrupt non possono interrompere una singola istruzione, indipendentemente dal microprogramma che la implementa, pertanto su un'architettura uniprocessor l'istruzione TS implementa un'operazione indivisibile sulla locazione di memoria.

E su un'architettura multiprocessor?

- anche sulle architetture multiprocessor, quali l'IBM/370, l'istruzione TS implementa un'operazione indivisibile: tra il test della locazione di memoria ed il suo aggiornamento la locazione è resa non accessibile ad altre CPU;

- ciò è realizzabile, per esempio, disabilitando gli accessi al bus della memoria;

- le istruzioni che implementano operazioni indivisibili vengono dette **istruzioni indivisibili**.

Implementare le CS con variabili lock condivise ed istruzioni TS.

Idea in pseudo_codice:

```
entry_test  if(lock = closed){goto entry_test;}
            lock = closed;
            {CS}
            lock=open;
```

Implementazione in IBM/370 con convenzione **lock 0** = aperto.

```
LOCK        DC X '00'   /* LOCK inizializzata con l'esadecimale 0 */
ENTRY_TEST  TS LOCK     /* Test and Set sulla variabile LOCK */
            BC 7, ENTRY_TEST /* Branch ad ENTRY_SET se i 3 bit
                                CC hanno valore 111, cioè 7 in
                                esadecimale */
            {CS}
            MVI LOCK, X'00' /* assegna l'esadecimale 0 a LOCK */
```

- L'istruzione TS a volte si chiama **TSL** – **t**est and **s**et **l**ock.

- L'istruzione TS ha uno oppure due parametri:

  - la variabile, come nel caso dell'IBM/370;

  - il registro in cui memorizzare il valore del test, se non viene usato il Condition Code della PSW.

Istruzione swap: istruzione indivisibile che scambia il contenuto di due locazioni di memoria.

Può essere usata come la TS per implementare le CS. Vediamo un esempio.

```
TEMP        DS  1   /* viene riservato 1 byte per TEMP */
LOCK        DC X '00'  /* il lock è inizialmente aperto */
            MVI TEMP X'FF' /* TEMP assume valore 11111111*/
ENTRY_TEST  SWAP LOCK, TEMP /* testa e chiudi il lock*/
            COMP TEMP, X'00' /* il risultato del test (TEMP == 0)
                                va  nel CC della PSW*/
            BC 7, ENTRY_TEST
            {CS}
            MVI LOCK, X'00' /* riapri il lock */
```

Nelle cpu **x86** la **swap** si chiama **exchange** (**XCHG**) ed ha come secondo parametro un registro.

Abbiamo visto come implementare le CS a basso livello. Ed a alto livello? Vedremo 3 strategie:

1. **approccio algoritmico**: i processi effettuano controlli prima di eseguire le CS al fine di garantire la mutua esclusione:
   - non servono accorgimenti hw, come quali le istruzioni macchina TS e SWAP, kernel o di linguaggio
   - l'approccio è difficile – vedi i tentativi errati di pag. 33 e 34 che sono esempi di approccio algoritmico;

2. uso di **primitive software** (chiamate a funzioni di libreria o system call) per garantire la mutua esclusione;

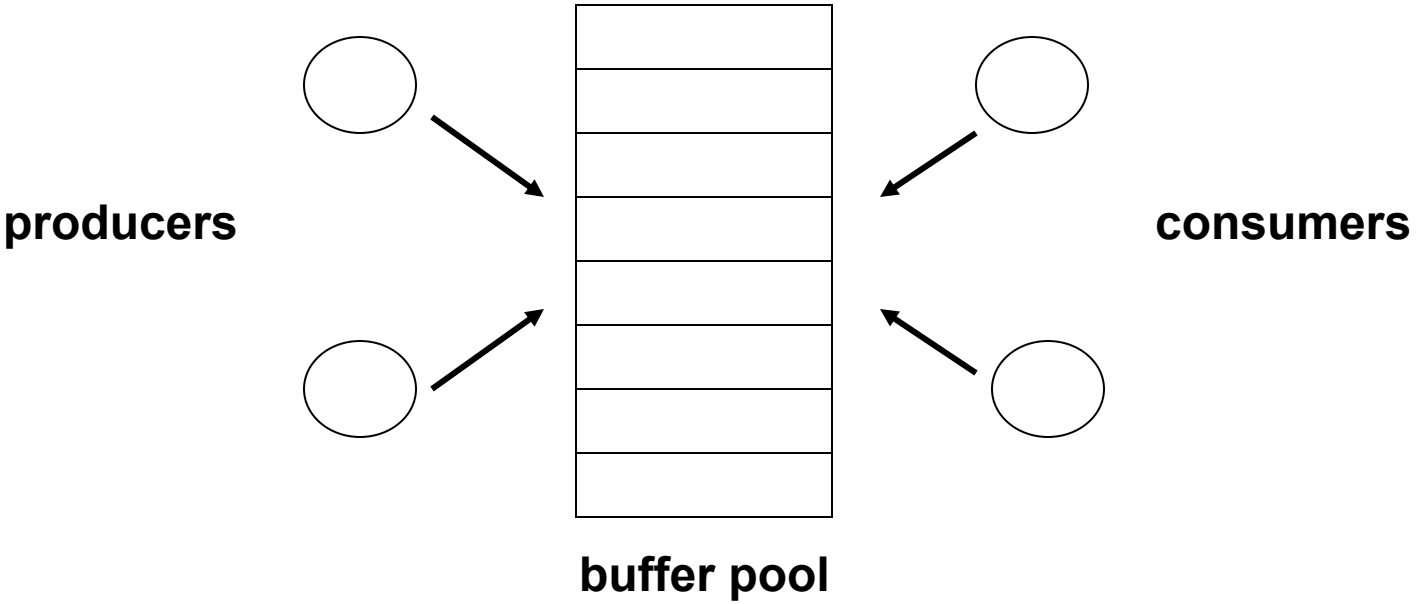
3. uso di **costrutti** ad hoc per la **programmazione concorrente**.

Le soluzioni 2 e 3 si vengono realizzate sulla base delle istruzioni indivisibili, quali la TS e la SWAP.

6.4 PROBLEMI CLASSICI

Il problema dei produttori e consumatori. Si assumano:

- un **pool finito di buffer**, ognuno dei quali può essere pieno, cioè contenere un record di informazione, oppure vuoto;
- un insieme di **processi produttori**, che producono singoli record di informazione con cui riempire i buffer;
- un insieme di **processi consumatori**, che consumano singoli record di informazione svuotando i buffer.



Soluzione valida del problema:

- l'accesso ai singoli buffer avviene in mutua esclusione;
- i produttori non possono riempire i buffer pieni, cioè già riempiti e non ancora svuotati;
- i consumatori non possono svuotare i buffer vuoti, cioè mai riempiti oppure riempiti e già svuotati.

Talvolta c'è una condizione aggiuntiva:

- i buffer devono essere svuotati nello stesso ordine in cui sono stati riempiti (politica FIFO).

Il problema astratto dei produttori/consumatori ha numerose istanze concrete. Esempio: il pool è una coda di stampa, i produttori sono i processi utente, c'è un solo consumatore che è il daemon di stampa.

Il problema è anche noto come problema del **bounded buffer**.

Vediamo una possibile soluzione, assumendo che il pool sia un array di interi da 100 elementi gestito in modo “circolare”.

Inizializzazioni nel main:

```
int count = 0; /* numero di buffer pieni, è ovviamente <= 100 */
int i = 0;         /* indice del primo buffer vuoto, valido se count <= 99 */
int j = 0;        /* indice del primo buffer pieno, valido se count >= 1 */
```

SOLUZIONE CON RACE CONDITION POSSIBILI:

Programma del producer

```
int item;
while(true){
    …….
    item = produce_item;
    while(count == 100){  }
    buffer[i] = item;
    count ++;
    i = (i+1) % 100;
    …….
}
```

Programma del consumer

```
int item;
while(true){
    ……..
    while(count == 0){  }
    item = buffer[j];
    count - -;
    j = (j+1) % 100;
    ……..
}
```

Per prevenire le race condition, è sufficiente che gli accessi ai dati condivisi avvengano nelle CS, riportate in blu.

Programma del producer:

```
int item;
while(true){
    ..........
    item = produce_item;
    while(count == 100){  }
    buffer[i] = item;
    count++;
    i = (i+1) % 100;
    ......
}
```

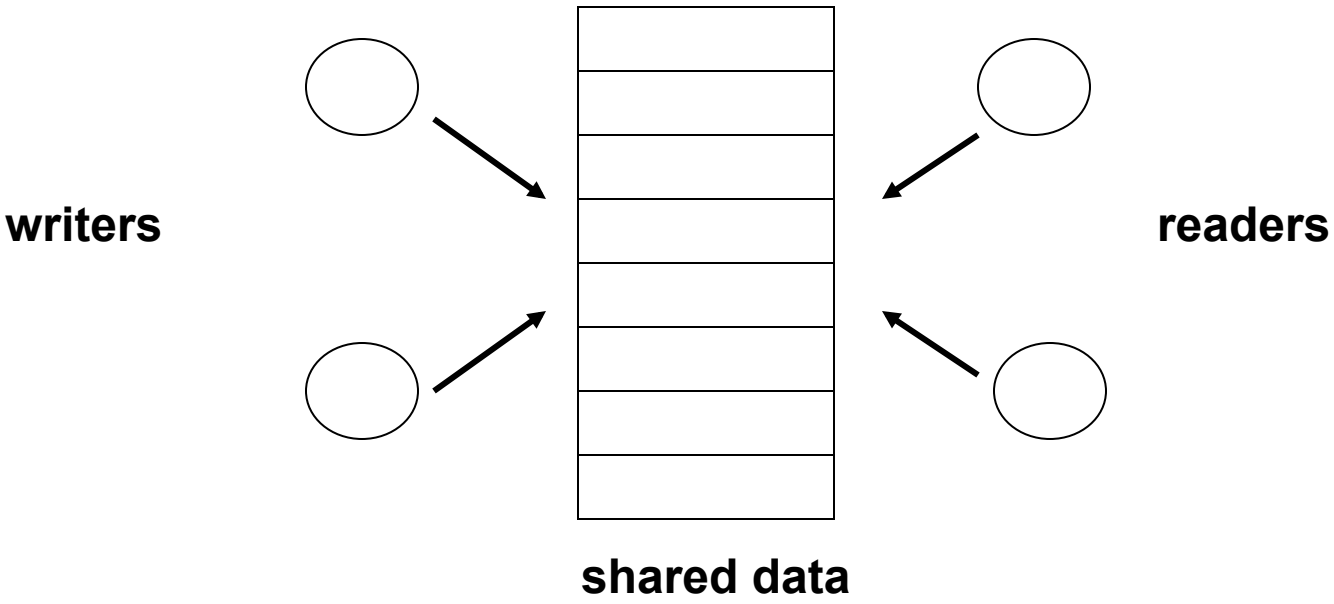
Programma del consumer:

```
int item;
while(true){
    ..........
    while(count == 0){  }
    item = buffer[j];
    count--;
    j = (j+1) % 100;
    ..........
}
```

Problema: in questa soluzione abbiamo **busy wait**: i processi occupano la CPU attendendo una condizione che non può verificarsi finchè non sarà qualche altro processo ad eseguire. Vedremo soluzioni migliori.

Il problema dei lettori e scrittori. Si assumano:

- un **insieme di dati condivisi**;
- un insieme di **processi lettori**, che accedono ai dati in sola lettura;
- un insieme di **processi scrittori**, che accedono ai dati in sola scrittura.



Soluzione valida del problema:

- sono consentite le letture concorrenti;
- non è ammessa la concorrenza tra le scritture;
- non è ammessa la concorrenza tra scritture e letture.

Il problema astratto dei lettori/scrittori ha numerose istanze concrete. Esempio: il dato è un conto bancario, gli scrittori sono i processi che gestiscono prelievi, depositi, modifiche di vario tipo, …. i lettori sono i processi che stampano i dati, leggono il saldo, prelevano dati statistici, ……

Vediamo l'idea della soluzione:

Programma del reader

```
.............
while(true){
   ……..
   while(someone is writing){  }
   set (l'm reading);
   {read};
   set (l'm not reading);
   ……..
}
```

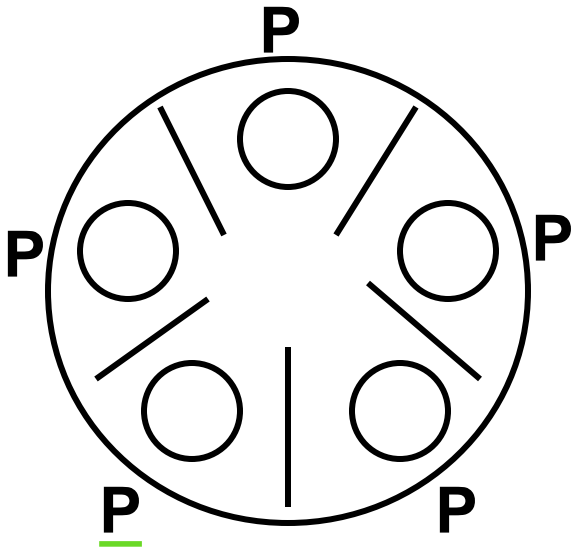
Programma del writer

```
 ……..
while(true){
    ……..
    while(someone is working){ }
     set (l'm writing);
     {write};
      set (l'm not writing);
    ……
 }
```

Vedremo possibili soluzioni.

Il problema dei filosofi a cena.

5 filosofi sono seduti intorno ad un tavolo, ed alternano momenti in cui pensano a momenti in cui mangiano. Per mangiare hanno bisogno di due bacchette. Le bacchette disponibili sono 5, ogni filosofo ne ha una alla sua destra ed una alla sua sinistra, per mangiare ha bisogno di entrambe.



Soluzione valida del problema:

- nessun filosofo muore di fame, cioè alterna momenti in cui pensa e momenti in cui mangia, all'infinito.

Vanno evitati:

- **deadlock**, cioè la situazione in cui i processi si bloccano in attesa uno dell'altro;
- **livelock**, cioè la situazione in cui i processi, pur non essendo bloccati, impediscono l'un l'altro all'infinito di acquisire le due bacchette necessarie a mangiare.

6.5 APPROCCIO ALGORITMICO ALLE SEZIONI CRITICHE

L'**approccio algoritmico** prevede di implementare le CS senza usare istruzioni hardware ad hoc, system call o costrutti del linguaggio ad hoc:

- un processo controlla alcune condizioni logiche prima di entrare in una CS;

- se il controllo è superato, il processo entra nella CS;

- se il controllo non è superato, il processo reitera il controllo: siamo di fronte ad un caso di **busy wait**.

  La busy wait sarebbe evitabile se si ponesse il processo in stato di waiting, ma ciò richiederebbe l'invocazione di una system call, possibilità che è stata esclusa.

Approccio algoritmico alle CS - idea generale:

```
.....
while(! ok){ /* busy wait!!! */ }
{CS}
.....
```

Esempio (già visto): uso delle variabili turno, con 2 processi:

Processo P_0

```
.....
while(true){
    .....
    while (turn != 0){  }
    {CS}
    turn = 1;
   .....
}
```

Processo P_1

```
.....
while(true){
    .....
    while (turn != 1){  }
    {CS}
    turn = 0;
    .....
}
```

Abbiamo già detto che questa soluzione viola la proprietà del progresso.

Uso delle variabili turno, secondo tentativo:

boolean c_0=F, c_1=F; /* c_i=T means P_i in CS, c_i=F means P_i not in CS */

Processo P_0

```
.....
while(true){
    .....
    while (c1){  }
    c0=T;
    {CS}
    c0=F;
   .....
}
```

Processo P_1

```
.....
while(true){
    .....
    while (c0){  }
    c1=T;
    {CS}
    c1=F;
    .....
}
```

La proprietà del progresso è rispettata, ma questa soluzione non è corretta, cioè la mutua esclusione non è garantita. Infatti, per esempio, $\mathbf{P_0}$ può essere interrotto dopo il controllo di $\mathbf{c_1}$ e prima dell'assegnamento a $\mathbf{c_0}$, in tal modo anche $\mathbf{P_0}$ può entrare nella CS.

Uso delle variabili turno, terzo tentativo:

boolean c_0=F,c_1=F; /* c_i=T means P_i in CS, c_i=F means P_i not in CS */

Processo P_0

```
……..
while(true){
    ……..
    c0=T;
    while (c1){  }
    {CS}
    c0=F;
   ……..
}
```

Processo P_1

```
……..
while(true){
    ……..
    c1=T;
    while (c0){  }
    {CS}
    c1=F;
    ……..
}
```

Ora la mutua esclusione è garantita, ma **c_0** e **c_1** potrebbero essere settate a **T** prima dei controlli delle guardie dei **while**: situazione di **stallo** (**deadlock**), i processi si bloccano a vicenda. Ovviamente è violata la proprietà del progresso.

Uso delle variabili turno, quarto tentativo:

boolean c_0=F, c_1=F; /* c_i=T means P_i in CS, c_i=F means P_i not in CS */

Processo P_0

```
.....
while(true){
    .....
a   c0=T;
    while (c1){c0=F;  goto a}
    {CS}
    c0=F;
   .....
}
```

Processo P_1

```
.....
while(true){
    .....
a   c1=T;
    while (c0){c1=F; goto a}
    {CS}
    c1=F;
    .....
}
```

Ora la mutua esclusione è garantita e non può esserci deadlock, ma entrambi i processi potrebbero eseguire all'infinito l'istruzione etichettata **a** ed il while: situazione di **livelock**, cioè i processi si ostacolano a vicenda. Ovviamente è violata la proprietà del progresso.

Uso delle variabili turno, **algoritmo di Dekker**. Soluzione vincente, ma valida solo per 2 processi.

```
boolean c0=F, c1=F;  int turn = 0;
Processo P0
.....
while(true){
    .....
    c0=T;
    while(c1){
       if(turn==1){
           c0=F;
           while(turn==1){  }
           c0=T;
       }
    }
    {CS}
    turn=1;
    c0=F;
    .....
}
```

```
Processo P1
.....
while(true){
    .....
    c1=T;
    while(c0){
          if(turn==0){
              c1=F;
              while(turn==0){  }
               c1=T;
          }
     }
     {CS}
     turn=0;
     c1=F;
     .....
 }
```

Uso delle variabili turno, **algoritmo di Peterson**. Soluzione vincente, ma valida solo per 2 processi.

```
boolean[ ] flag = {F;F};  int turn = 0;
```

Processo P_0

```
……….
while(true){
    ……..
    flag[0]=T;
    turn=1;
    while(flag[1] & turn==1){  }
    {CS}
    flag[0]=F;
    ……..
}
```

Processo P_1

```
……….
while(true){
    ……..
    flag[1]=T;
    turn=0;
    while(flag[0] & turn==0){  }
    {CS}
    flag[1]=F;
    ……..
}
```

Se i processi sono N>2? Due algoritmi noti sono i seguenti:

- l'algoritmo di **Eisenberg e McGuire**, che estende al caso N l'algoritmo di Dekker;

- il **Bakery algorithm di Lamport**.

Algoritmo di Eisenberg e McGuire (1972) per processi $P_0,..,P_{N-1}$.

```
{idle, want_in, in_cs} [N] flag = {idle,…,idle};  int turn=0;
PROCESSO Pi:
while(true){
 repeat{
   flag[i] := want_in; /* segnalo che voglio entrare in CS */
   j := turn;
   while(j != i){ /* controllo i Pj con maggior priorità */
     if(flag[j] != idle) {j := turn; /* aspetto: Pj ha la priorità */ }  else {j := j+1%N;}
   }
   flag[i] := in_cs; /* dico di essere in CS, anche se non ci sono ancora */
    /* anche un altro Pj potrebbe essersi dichiarato in CS: devo controllare */
   j := 0;  while (j < N AND (j = i OR flag[j] != in_cs)){j := j+1;}
     /* se un altro Pj si è dichiarato in CS vale che j<N */
 }
 until (j >= N AND (turn = i OR flag[turn] = idle))
 turn := i;
 {CS}
 j := turn+1 mod N ;
 while (flag[j] = idle){j := j+1 % n};
 turn := j ;
 flag[i] := idle;
 ……................
}
```

L'algoritmo di Eisenberg-McGuire si basa su una priorità dinamica: $\mathbf{P_{turn}}$, $\mathbf{P_{turn+1}}$, .., $\mathbf{P_{N-1}}$, $\mathbf{P_0}$, …, $\mathbf{P_{turn-1}}$. Ogni $\mathbf{P_i}$ fa i seguenti passi prima di entrare nella CS:

1. Il processo $\mathbf{P_i}$ controlla che i processi con maggior priorità non abbiano dichiarato di voler entrare nella CS impostandosi lo stato a **want_in** oppure **in_cs**. Se il controllo è ok, $\mathbf{P_i}$ imposta il suo stato a **in_cs**.
2. Più processi potrebbero aver fatto il controllo in concorrenza, impostandosi lo stato a **in_cs**. Quindi, dopo essersi messo in stato **in_cs**, $\mathbf{P_i}$ deve controllare se qualcun altro ha fatto la stessa cosa. In caso affermativo, $\mathbf{P_i}$ si rimette in **want_in** e reitera i controlli partendo dal punto 1. Ciò garantisce la mutua esclusione.
3. Se i processi si rimettono in **want_in** secondo quando detto al punto 2 e ripartono dal punto 1, il processo con maggior priorità sarà ora l'unico a mettersi in **in_cs** e potrà accedere alla CS. Non esiste rischio di livelock, il progress è garantito.

NB: La bounded wait è garantita dall'assegnamento a **turn** che il processo fa uscendo dalla CS.

Bakery algorithm di Lamport (1974)

```
int[N] number = {0,…,0}; boolean[N] choosing = {F,…,F}

PROCESSO Pi:
while(true){
   choosing[i] = true;   /* I'm choosing a token */
   number[i] = max(number[0],..,number[N-1]) + 1;
                              /* I take the greatest token ever issued */
   choosing[i] = false; /* I'm no more choosing */
   for(int j=0; j<=N-1; j++){
      while(choosing[j]){ /* busy wait if Pj is choosing */  }
      while(number[j] != 0 & (number[j]<number[i] |
                                    number[j]=number[i] & j<i){
        /* busy waiting if Pj has highest priority */
      }
   }
   {CS}
   number[i]=0;
   ……..……..
}
```

- L'algoritmo di Lamport si basa su token interi.

- Prima di entrare nella CS, un processo $\mathbf{P_i}$ ottiene un token, che è maggiore di tutti quelli in possesso degli altri processi.

- Se più processi intendono entrare nella CS, vince il processo con il token minore, cioè il token "più anziano": ciò serve per garantire ai processi di entrare nella CS rispettando l'ordine in cui manifestano la volontà di farlo. L'algoritmo di Eisenberg e McGuire non prevede questa caratteristica.

- Se più processi ottengono il token concorrentemente potrebbero ottenere il medesimo: in tal caso vince il processo con l'indice minore.

6.6 SEMAFORI

Definizione (Dijkstra, 1965): un **semaforo** è una **variabile intera condivisa** che può assumere solo **valori non negativi** e su cui sono possibili solo le seguenti tre operazioni:

- **inizializzazione** (con un valore >= 0);
- **operazione indivisibile wait**:

  - se il semaforo ha valore > 0, viene decrementato;

  - se il semaforo ha valore = 0, il processo che esegue la **wait** viene "bloccato sul semaforo", cioè è impossibilitato a proseguire. Di fatto, va in waiting;

- **operazione indivisibile signal**:

  - se c'è almeno un processo bloccato sul semaforo, uno dei processi bloccati sul semaforo viene "sbloccato", cioè può riprendere l'esecuzione. Di fatto, va in ready;

  - se nessun processo è bloccato sul semaforo, il semaforo viene incrementato.

Osservazioni:

- Nella definizione non viene precisato come siano implementate **wait** e **signal**, in particolare come sia garantita la loro indivisibilità.

- Nella definizione non viene precisata la politica con cui la **signal** sceglie quale processo svegliare se i processi bloccati sul semaforo sono più di uno.

Pur non conoscendo questi dettagli (che chiariremo più avanti), possiamo vedere come si possano usare i semafori. Assumeremo la sintassi **wait(x)** e **signal(x)** per invocare **wait** e **signal** su un semaforo **x**.

Uso dei **semafori per implementare le sezioni critiche**:

Sia sem un semaforo inizializzato a 1.

Processo Pi :

……

```
while(true){
    ……..
    wait(sem);
    {CS}
    signal(sem);
    ……
}
```

Questa soluzione si basa su:

- indivisibilità di **wait** e **signal**;
- inizializzazione del semaforo a 1.

Osservazioni:

- **sem** assume solo i valori 0 e 1;
- se un processo è nella CS, **sem** vale 0;
- se un processo si blocca eseguendo la **wait**, **sem** ha già valore 0,
- se un processo viene sbloccato da una **signal**, **sem** rimane a 0,
- **sem** assume valore 1 quando un processo esegue una **signal** e nessun altro processo è bloccato su **sem**.

Usare i semafori inizializzati a 1 per implementare le CS è una soluzione valida?

1. La mutua esclusione è garantita dal fatto che la **wait** precede la CS.

2. Il progresso è garantito dal fatto che la **signal** segue la CS.

3. La bounded wait è garantita o meno in base all'implementazione della **signal**. Per esempio, se la **signal** sblocca i processi con lo stesso ordine in cui sono stati bloccati, la bounded wait è garantita.

Ovviamente tutto funziona a patto che il programmatore rispetti la sequenza **wait** – {CS} – **signal**.

Uso dei **semafori per la bounded concurrency**.

Usando i semafori è possibile fare in modo che al massimo **C** processi su un insieme di **N** > **C** eseguano determinate operazioni concorrentemente. Esempio: abbiamo **N** processi che condividono **C** stampanti:

sem è un semaforo inizializzato a C

Processo Pi:

……..

while(true){

…….

wait(sem); /*process blocks if and only if all printers already in use*/

{use a printer}

signal(sem);

…….

}

Uso dei semafori per il problema dei produttori e consumatori. Caso facile: abbiamo un solo buffer.

```
semaphore full = 0;        /*      full = 1 if and only if buffer full */
semaphore empty = 1;  /* empty = 1 if and only if buffer emtpy*/
```

Programma del producer:

```
int item;
while(true){
    …….
    item = produce_item;
    wait(empty);
    buffer = item;
    signal(full);
    ……..
}
```

Programma del consumer:

```
int item;
while(true){
    …….
    wait(full);
    item = buffer;
    signal(empty);
    ……..
}
```

Caso generale, con array di N buffer gestito circolarmente.

```
semaphore full = 0;        /*  full = number of full buffers */
semaphore empty = N;       /* empty = number of empty buffers*/
semaphore sem_p = 1        /* mutual exclusion over index i */
semaphore sem_c = 1        /* mutual exclusion over index j */
```

Programma del producer:

```
int item;
while(true){
    ……….
    item = produce_item;
    wait(empty);
    wait(sem_p);
    buffer[i] = item;
    i=(i+1) % N;
    signal(sem_p);
    signal(full);
    ……..
}
```

Programma del consumer:

```
int item;
while(true){
    ……….
    wait(full);
    wait(sem_c);
    item = buffer[j];
    j=(j+1) % N;
    signal(sem_c);
    signal(empty);
    ……..
}
```

Uso dei semafori per il problema dei lettori e scrittori.

Assumiamo di avere **R** lettori e **W** scrittori.

Usiamo i seguenti semafori e variabili.

semR: semaforo inizializzato a 0 – serve per dare l'ok ai lettori
semW: semaforo inizializzato a 0 – serve per dare l'ok agli scrittori
mutex: sem. inizializzato a 1 - serve per la mutua esclusione sulle variabili seguenti

runR: numero di lettori che stanno leggendo
runW: numero di scrittori che stanno scrivendo
totR: numero di lettori che stanno leggendo o che vorrebbero leggere
totW: numero di scrittori che stanno scrivendo o che vorrebbero farlo

N.B.: 0 <= totR <= R, 0 <= totW <= W,
0 <= runR <= R, 0 <= runW <= 1,
runR>0 ➔ runW =0, runW=1 ➔runR=0
0 <= semR <= R, 0 <= semW <= 1,
semR>0 ➔ semW =0, semW=1 ➔semR=0
0 <= mutex <= 1.

```
while(true){
    wait(mutex);
    totR = totR + 1;
    if(runW = 0){
        runR = runR + 1;
        signal(semR); /* (*) */
    }
    signal(mutex);
    wait(semR);
    {Read}
    wait(mutex);
    runR = runR - 1;
    totR = totR - 1;
    if(runR=0 & runW<totW){
        runW = 1;
        signal(semW);
    }
    signal(mutex);
}
```

(*) si parla di *self scheduling*

READER

WRITER

```
while(true){
    wait(mutex);
    totW = totW + 1;
    if(runW = 0 & runR = 0){
        runW = runW + 1;
        signal(semW); /* (*) */
    }
    signal(mutex);
    wait(semW);
    {Write}
    wait(mutex);
    runW = runW - 1;
    totW = totW - 1;
    while(runR<totR){
        runR = runR+1;
        signal(semR);
    }
    if(runR=0 & runW<totW){
        runW=1;
        signal(semW);
    }
    signal(mutex);
}
```

Attenzione: l'uso dei semafori non è banale.
Siano x, y semafori inizializzati a 1.
Può verificarsi **deadlock**:

Processo P1:

```
.......
.......
wait(x);
wait(y);
........
........
signal(x);
signal(y);
.........
```

Processo P2:

```
.........
.........
wait(y);
wait(x);
.........

signal(y);
signal(x);
.........
```

Uso dei semafori per il problema dei filosofi a cena.

Usiamo le seguenti strutture dati:

- un array di stati con valori HUNGRY, EATING, THINKING;
- un semaforo **mutex**, per accedere all'array in mutua esclusione;
- un semaforo **sem[i]** per ogni filosofo $\mathbf{P_i}$, che serve per dare l'ok a $\mathbf{P_i}$.

```
#define N 5;
#define LEFT (i+N-1)%N
#define RIGHT (i+1)%N
#define THINKING=0
#define HUNGRY=1
#define EATING=2
int state[N];
semaphore mutex=1;
semaphore sem[N];
```

```
void philosopher(int i){
 while(true){
    think( );
    takeForks(i);
    eat( );
    putforks(i);
 }
}
```

```
takeForks(i){
  wait(mutex);
  state[i]=HUNGRY;
  test(i);
  signal(mutex);
  wait(sem[i]);
}
```

```
putForks(i){
  wait(mutex);
  state[i]=THINKING;
  test(LEFT);
  test(RIGHT);
  signal(mutex);
}
```

```
test(i){
  if(state[i]==HUNGRY &
      state[LEFT] != EATING &
      state[RIGHT] != EATING){
           /* both forks are free*/
           state[i] = EATING;
           signal(sem[i]);
  }
}
```

Implementazione. Un semaforo richiede:

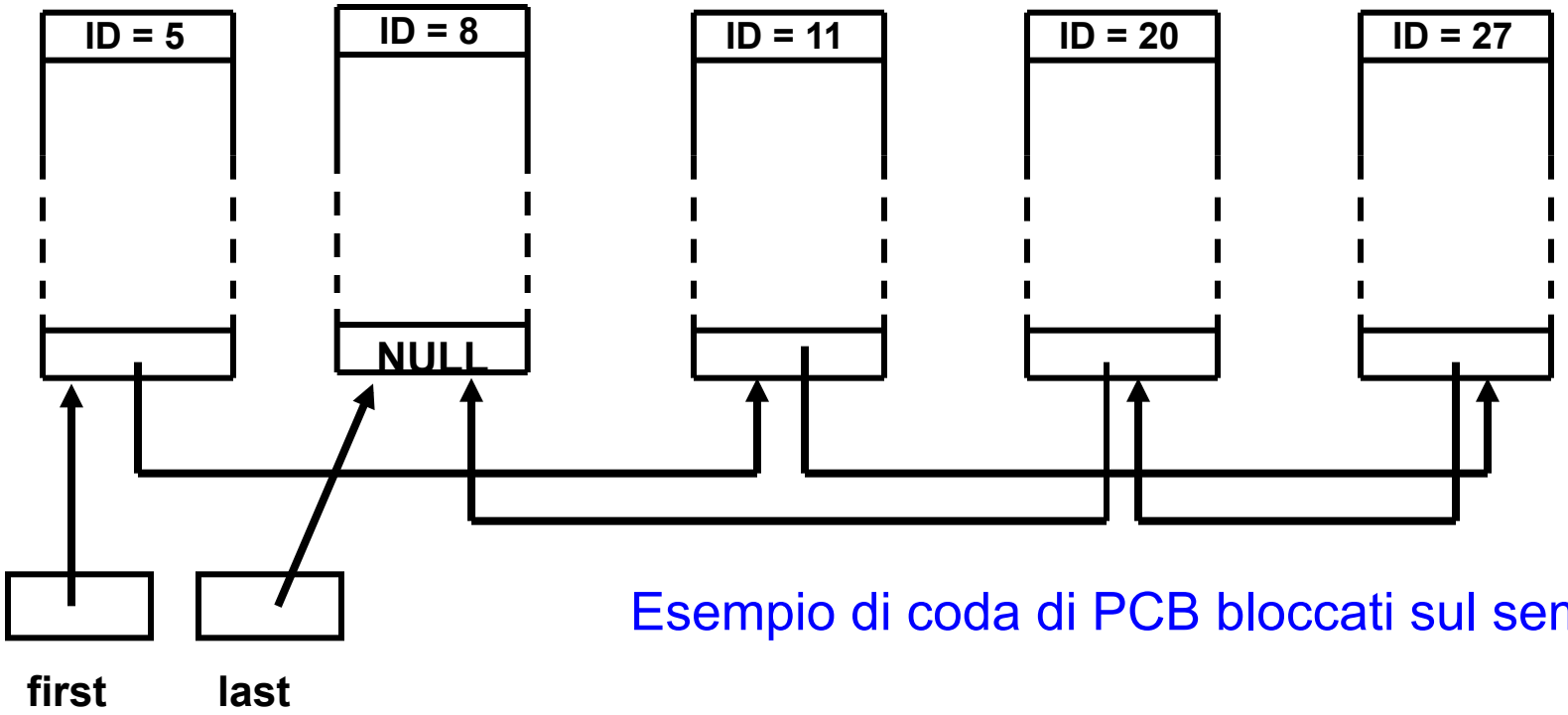
- un intero per memorizzare il valore del semaforo;
- una variabile lock per garantire l'indivisibilità di **wait** e **signal**;
- una lista dei processi bloccati sul semaforo:

  - può essere realizzata usando i pointer dei PCB,
  - viene solitamente gestita come coda per assicurare che i processi bloccati sul semaforo vengano svegliati con la politica FIFO.

Idea di wait e signal:

```
procedure wait(sem){
  if sem.value > 0{
      sem.value - -;
  }
  else{
      insert PCB in sem.list;
      block_me( );
  }
}
```

```
procedure signal(sem){
  if(sem.list not empty){
     remove first process from list;
     wake up process;
  }
  else{
     sem.value ++;
  }
}
```



Esempio di coda di PCB bloccati sul semaforo

Implementazioni dei semafori **kernel level**:

- i semafori sono strutture dati del kernel: la lista dei processi bloccati sul semaforo può essere realizzata sfruttando i pointer dei PCB, di cui il kernel è "proprietario";

- **wait** e **signal** sono **system call**. Ovviamente possono esserci funzioni wrapper di libreria **wait** e **signal** per invocarle;

- l'indivisibilità di **wait** e **signal** è ottenuta usando variabili lock ed istruzioni indivisibili (TS, SWAP).

  Ciò causa busy wait, ma solo per il tempo necessario ad eseguire **wait** e **signal**, che sono brevi.

  Le variabili lock non servono se il sistema è uniprocessor e **wait** e **signal** vengono eseguite con tutti gli interrupt masked off.

- per bloccare/sbloccare i processi, **wait** e **signal** lavorano sui PCB e sulla coda dei PCB.

Implementazione dei semafori **user level**:

- i semafori sono strutture dati accessibili in modalità user: deve essere realizzata una lista di PID di processi bloccati sul semaforo che non sfrutti i pointer dei PCB, che sono accessibili solo in modalità kernel;

- **wait** e **signal** sono **procedure libreria** che eseguono in user mode;

- l'indivisibilità di **wait** e **signal** richiede variabili lock ed istruzioni indivisibili;

- per bloccare/sbloccare i processi, **wait** e **signal** non possono lavorare sui PCB e sulla coda dei PCB, ma devono invocare le system call apposite messe a disposizione dal kernel.

6.7: MONITOR

Il **monitor** è un costrutto offerto da alcuni linguaggi, quali Java.

Definizione: Un **tipo monitor** consiste di:

- dichiarazione di variabili, alcune delle quali sono dette **variabili di condizione**;
- inizializzazione delle variabili;
- procedure che usano le variabili:

  - è garantita, dal monitor, l'esecuzione delle procedure in **mutua esclusione**. Le chiamate di procedure sono servite con la politica FIFO;

  - per ogni variabile di condizione **x**, esiste una coda di processi bloccati e le procedure possono invocare le seguenti operazioni:

  -- **x.wait**: il processo viene bloccato ed inserito nella coda di **x**;

  -- **x.signal**: il processo da più tempo nella coda di **x** viene sbloccato. Se nessun processo è nella coda di **x**, la **signal** non ha effetto.

Un'altra caratteristica del monitor:

- se un processo $\mathbf{P_1}$ è stato sbloccato da una **signal** dopo che ha eseguito una **wait**, ed un altro processo $\mathbf{P_2}$ è stato sbloccato dopo essere stato bloccato a causa della mutua esclusione sull'accesso alle procedure, il monitor da priorità a $\mathbf{P_1}$.

```
monitor ProducerConsumer{
    condition full, empty;
    int c, i, j =0;

    void insert(int x){
       if(c=N){full.wait);}
       buffer[j] = x; (j++)%N; c++;
       if(c=1){empty.signal;}
    }

    int remove( ){
       if(c=0){empty.wait;}
       int x = buffer[i]; (i++)%N; c- -;
       if(c=N-1){full.signal;}
       return x;
    }
}
```

```
void producer( ){
  while(true){
     ……
     int x = produce_item;
     ProducerConsumer.insert(x);
     ……
   }
}

void consumer( ){
  while(true){
     ……
     int x =
       ProducerConsumer.remove( );
     ……
   }
}
```

Monitor in Java:

- Un monitor è una classe i cui metodi hanno l'attributo **synchronized**;
- La mutua esclusione tra i thread che eseguono i metodi è garantita dalla JVM, che usa le system call per bloccare i thread e risvegliarli;
- Vi è una sola variabile di condizione, implicita, ma non ha una coda di thread bloccati, bensì una lista (no FIFO);
- La **wait** sulla variabile di condizione è il metodo **wait( )**, che richiede la **catch** delle **InterruptException**.
- La **signal** sulla variabile di condizione è il metodo **notify( )**. La JVM sblocca un thread, non necessariamente quello da più tempo bloccato.
- Il metodo **notifyall( )** sblocca tutti i thread bloccati.

```
static class MonitorProducerConsumer
    private int[ ] buffer = new int[100];
    private int c=0, i=0; j=0;

    synchronized void insert(int x){
      if(c==100){try{wait( );}catch(InterruptException e){ }}
      buffer[j]=x; (j++)%100; c++;
      if(c==1){notify( );}
    }

    synchronized int remove( ){
      if(c==0){try{wait( );}catch(InterruptException e){ }}
      int x = buffer[i]; (i++)%N; c- -;
      if(count==N-1){notify( );}
      return x;
    }
}
```

Consideriamo la seguente classe **VarX**:

```
class VarX{
   int x=0;
   void op(String s){
      for(int i=0; i<10000; i++){x++;}
      System.out.println(s + " x vale "+x); /* x=10000, 20000, 30000?? */
   }
}
```

Assumiamo un oggetto **var** di tipo **VarX**. Lanciando thread che concorrentemente invocano il metodo **op** di **var**, potrebbe capitare che vengano stampati valori della variabile **x** di **var** non multipli di 10000.

```
class UsaVarX extends Thread{
    protected String nome;
    UsaVarX(String nome){this.nome=nome;}
    public void run( ){
        while(true){
            TestVarX.var.op(nome); /* var is an object defined in TestVarX */
        }
     }
}

class TestVarX{
   protected static VarX var = new VarX( ); /* var is owned by TestVarX*/
   public static void main(String[ ] args){
        UsaVarX a = new UsaVarX("aaaa");    /* thread using var */
        UsaVarX b = new UsaVarX("bbbb");   /* thread using var */
        UsaVarX c = new UsaVarX("cccc");    /* thread using var */
        UsaVarX d = new UsaVarX("dddd");   /* thread using var */
        a.start( ); b.start( ); c.start( ); d.start( );  /* concurrent threads */
   }
}
```

Frammento di un'esecuzione:

```
C:\EserciziInJava>java TestVarX
aaaa x vale 10000
dddd x vale 39958
cccc  x vale 24057
bbbb x vale 29789
cccc  x vale 69789
```

Consideriamo la seguente classe **VarMonX**:

```
class VarMonX{
   int x=0;
   synchronized void op(String s){
      for(int i=0; i<10000; i++){x++;}
      System.out.println(s + " x vale "+x);
   }
}
```

Assumiamo un oggetto **var** di tipo **VarMonX**. Lanciando thread che concorrentemente invocano il metodo **op** di **var**, siamo certi che vengano stampati solo valori della variabile **x** di **var** multipli di 10000.

```
class UsaVarMonX extends Thread{
    protected String nome;
    UsaVarMonX(String nome){this.nome=nome;}
    public void run( ){
        while(true){
            TestVarMonX.var.op(nome);
        }
     }
}

class TestVarMonX{
    protected static VarMonX var = new VarMonX( );
    public static void main(String[ ] args){
        UsaVarMonX a = new UsaVarMonX("aaaa");
        UsaVarMonX b = new UsaVarMonX("bbbb");
        UsaVarMonX c = new UsaVarMonX("cccc");
        UsaVarMonX d = new UsaVarMonX("dddd");
        a.start( ); b.start( ); c.start( ); d.start( );
    }
}
```

Un altro esempio: Realizziamo la classe **PilaStrana**, con tre metodi:

- **push**: classica push delle pile;
- **pop**: classica pop delle pile;
- **stampa**: stampa a video gli elementi della pila. E' un'operazione insensata per le pile, ma ci serve per vedere cosa succede!

Usiamo la classe **Nodo**:

```
class Nodo{
   int  elem;
   Nodo next; \* pointer to another Nodo *\
   Nodo(int elem, Nodo next){
       this.elem=elem; this.next=next;
   }
}
```

```
public class PilaStrana{
  Nodo top;      /* unica variabile di classe: pointer al top della pila */
  PilaStrana( ){ top=null; } /* la pila si crea vuota */

 public synchronized void stampa(String s){
     Nodo n = top;
     System.out.println("\n\n\n\nSONO " + s + " E STAMPO LA PILA\n");
     while(n!=null){
        System.out.println(n.elem);
        n=n.next;
     }
     System.out.println("\nSONO " + s + " ED HO STAMPATO LA PILA
                              \n\n\n\n\n\n");
 }
```

```
public synchronized void push(int x, String s){
    System.out.println("SONO " + s + " ED INSERISCO " + x);
    if(top==null){
       System.out.println("\nSONO " + s +" E NOTIFICO");
       top = new Nodo(x,top); notify( );
    }
    else{ top = new Nodo(x,top); }
    System.out.println("\nSONO " + s + " ED HO INSERITO " + x
                                                    +"\n\n\n\n\n\n");
}
public synchronized int pop(String s){
   if(top==null){
      System.out.println("SONO "+s+ " E VADO A
                                  DORMIRE!!!!!!!!!!!!!!!!!!\n\n\n");
      try{ wait( ); }catch(Exception e){ }
      System.out.println("\nDRIIIIIIIIIIIIIIIIIIIIIIIIIIIN PER " +s+"\n");
   }
   int x = top.elem; top = top.next;
   System.out.println("SONO " + s +" ED ESTRAGGO " + x +
                                                      "\n\n\n\n\n\n");
   return x;
}
```

```
import java.lang.*;
import java.util.*;

class TestPilaStrana{
 static PilaStrana s = new PilaStrana();
 public static void main(String[] args){
   System.out.println("\n\nVIA!!!!!\n\n");
   ThreadPush a = new ThreadPush();
   ThreadPop   b = new ThreadPop();
   ThreadStampa c = new ThreadStampa();
   a.start(); b.start(); c.start();
 }
}

class ThreadPush extends Thread{
  public void run(){
    Random r = new Random();
    while(true){
     int x = r.nextInt(100);
      TestPilaStrana.s.push(x,"IMPILATORE");
     for(int i=0; i<100000; i++){ }
    }
  }
}

class ThreadPop extends Thread{
 public void run(){
  while(true){
   TestPilaStrana.s.pop("ESTRATTORE");
    for(int i=0; i<100000; i++){ }
  }
 }
}

class ThreadStampa extends Thread{
 public void run(){
  while(true){
    TestPilaStrana.s.stampa("STAMPATORE");
     for(int i=0; i<100000; i++){ }
  }
 }
}
```