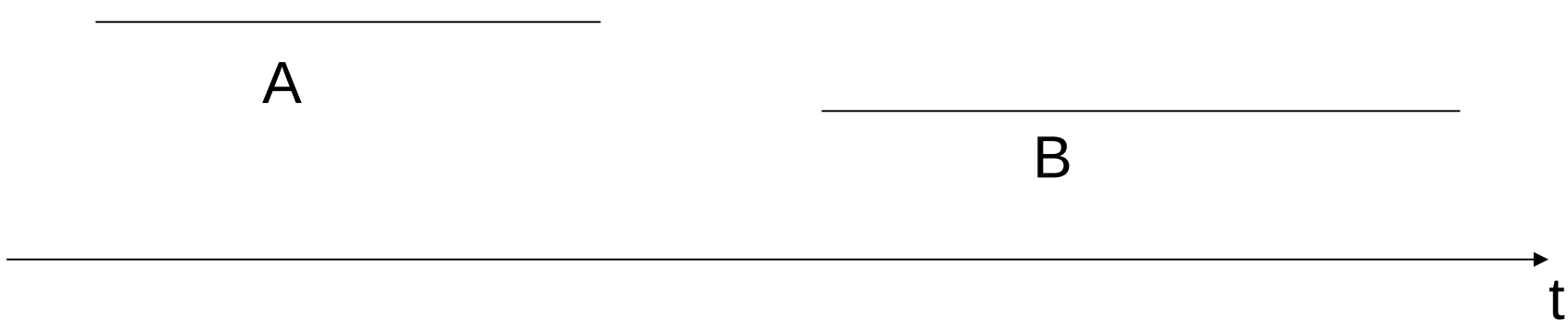
Mutua Esclusione

- **Il problema della mutua esclusione** nasce quando più di un processo alla volta può aver accesso a variabili comuni.
- La regola di mutua esclusione impone che le operazioni con le quali i processi accedono alle variabili comuni ***non si sovrappongano nel tempo***.
- **Nessun vincolo è imposto *sull'ordine*** con il quale le operazioni sulle variabili vengono eseguite.



Esempi di mutua esclusione

Esempio 1

- Due processi P1 e P2 hanno accesso ad una struttura *organizzata a* **pila** rispettivamente per inserire e prelevare dati.
- La struttura dati è rappresentata da un *vettore* ***stack*** i cui elementi costituiscono i singoli dati e da una *variabile* ***top*** che indica la posizione dell'ultimo elemento contenuto nella pila.
- I processi utilizzano le operazioni ***Inserimento*** *e* ***Prelievo*** per depositare e prelevare i dati dalla pila.

```
typedef ... item;
item stack[N];
int top=-1;

void Inserimento(item y)
{    top++;
     stack[top]=y;
}

item Prelievo()
{    item x;
     x= stack[top];
     top--;
     return x;
}
```

- L’esecuzione contemporanea di queste operazioni da parte dei processi può portare ad un uso scorretto della risorsa.
- Possibile sequenza di esecuzione delle due operazioni:

T0: **`top++;`** (P1)
T1: **`x=stack[top];`** (P2)
T2: **`top--;`** (P2)
T3: **`stack[top]=y;`** (P1)

- Viene assegnato a x un valore *non definito* e l’ultimo valore valido contenuto nella pila *viene cancellato* dal nuovo valore di y.
- Analogamente si avrebbe nel caso di esecuzione contemporanea di una qualunque delle due operazioni da parte dei due processi.

Esempi di Mutua esclusione

Esempio 2

- P1 e P2 accedono ad una variabile comune *contatore* che devono incrementare ogniqualvolta effettuano una determinata azione.
- Al completamento dell'esecuzione dei processi *contatore* deve contenere un valore *pari al numero complessivo* delle azioni effettuate dai due processi.
- In termini di istruzioni ***assembler*** l'istruzione

  contatore= contatore+1;

- può essere espressa come:

```
LD contatore;
AD 1;
STO contatore;
```

Se al termine di un'azione i processi eseguono concorrentemente la modifica di *contatore*, si può avere una sequenza del tipo:

T0: ***LD contatore*** *(P1)*
T1: ***LD contatore*** *(P2)*
T2: ***AD 1*** *(P2)*
T3: ***STO contatore*** *(P2)*
T4: ***AD1*** *(P1)*
T5: ***STO contatore*** *(P1)*

Si ha come risultato che il valore della variabile *contatore* viene incrementato di *una sola unità*.

➔ E' necessario che le operazioni di **modifica della variabile *contatore*** siano effettuate in modo ***mutuamente esclusivo***

Istruzioni indivisibili

***Azione atomica*:** esegue una trasformazione di stato *indivisibile*. Può esistere uno stato intermedio nella realizzazione dell'azione, ma non è rilevabile all'esterno.

Ipotesi:

- I valori dei tipi base (es. interi) sono memorizzati in parole di memoria che vengono lette e scritte in *modo atomico*.
- I valori sono manipolati caricandoli nei registri, operando sui *registri* e memorizzando il risultato in *memoria*
- Ciascun processo ha il proprio set di registri. Ciò si realizza, in genere, con il *context switch*.
- Ogni *risultato intermedio* durante la valutazione di un'espressione viene valutato e memorizzato in registri o in memoria privata del processo in esecuzione (es. *stack privato*)

- Con questo modello di macchina se in un processo un'espressione ***e*** non fa riferimento a variabili modificate da un altro processo, la valutazione dell'espressione è *atomica* anche se risulta composta da azioni atomiche più elementari.
- Infatti:
  - nessuno dei valori da cui e dipende possono cambiare durante la valutazione di ***e***;
  - nessun altro processo può vedere valori temporanei che potrebbero essere creati durante la valutazione di ***e***.

Sezione Critica

- La sequenza di istruzioni con le quali un processo accede e modifica un insieme di variabili comuni prende il nome di **sezione critica.**
- Ad un insieme di variabili comuni possono essere associate *una sola sezione critica* (usata da tutti i processi) o *più sezioni critiche (classe di sezioni critiche).*
- La ***regola di mutua esclusione*** stabilisce che*:*

Sezioni critiche appartenenti alla stessa classe devono escludersi mutuamente nel tempo.

oppure

Una sola sezione critica di una classe può essere in esecuzione ad ogni istante.

Realizzazione della regola di mutua esclusione

- Tempificazione dell'esecuzione dei singoli processi da parte del programmatore:

Errori *time-dependent*

- Inibizione delle interruzioni del processore durante l'esecuzione della sezione critica:

Soluzione parziale ed inefficiente

➔ ***Strumenti di sincronizzazione***

Schema Generale

- Ogni processo prima di entrare in una sezione critica deve chiedere l'autorizzazione eseguendo un serie di istruzioni che gli garantiscono *l'uso eslusivo della risorsa*, se questa è libera, oppure *ne impediscano* l'accesso se questa è già occupata (***PROLOGO***) .

- Al completamento dell'azione il processo deve eseguire una sequenza di istruzioni per dichiarare libera la sezione critica (***EPILOGO***)

MUTUA ESCLUSIONE: Analisi di alcune soluzioni e definizione dei requisiti

Soluzione 1: Disabilitazione delle interruzioni durante le sezioni critiche:

- **Prologo: disabilitazione delle interruzioni**
- **Epilogo: abilitazione delle interruzioni**

```
/* struttura processo: */
main()
{     ...
      <disabilitazione delle interruzioni>;
      <sezione critica A>;
      <abilitazione delle interruzioni>;
      ...
}
```

Problemi:

• La soluzione è *parziale* in quanto è valida solo per sezioni critiche che operino sullo stesso processore.

• *Elimina* ogni possibilità di parallelismo.

• Rende *insensibile* il sistema ad *ogni stimolo esterno* per tutta la durata di qualunque sezione critica

Soluzione 2: (***A,B***) **classe** di sezioni critiche, *libero* variabile logica, inizializzata al valore *true,* associata a tale classe:

`int libero=1`;

```
   /* processo P1: */
   main()
   {   ...
       while (!libero);
       libero=0;
       <sezione critica A>;
       libero=1;

       ...
}
```

```
   /* processo P2: */
   main()
   {   ...
       while (!libero);
       libero=0;
       <sezione critica B>;
       libero=1;

       ...
   }
```

- La soluzione **non soddisfa** la proprietà di mutua esclusione nell'esecuzione delle sezioni critiche.

Esempio:

T0 : P1 esegue l'istruzione **while** e trova *libero = 1*

T1 : P2 esegue l'istruzione **while** e trova *libero = 1*

T3 : P1 pone ***libero=0*** ed entra nella sezione critica

T4 : P2 pone ***libero=0*** ed entra nella sezione critica

➔ Tale sequenza ha come risultato che entrambi i processi sono contemporaneamente nella sezione critica

Soluzione 3: alla classe di sezioni critiche (A,B..) viene associata la variabile *turno* che può assumere i valori 1 e 2 ed inizializzata a 1.

```
int turno=1;
```

```
/* processo P1: */
main()
{   ...
    while (turno!=1));

    <sezione critica A>;

    turno=2;

    ...
}
```

```
/* processo P2: */
main()
{   ...
    while (turno!=2);

    <sezione critica B>;

    turno=1;

    ...
}
```

- La soluzione assicura che un solo processo alla volta può trovarsi nella sezione critica.
- Essa tuttavia impone un **vincolo di alternanza** nella esecuzione delle sezioni critiche.
- Ad esempio, se *turno* = 2, il processo P1 non può entrare nella sua sezione critica, anche se questa non è occupata da P2.
- Solo quando P2 avrà eseguito la sezione critica B, P1 potrà eseguire la propria.

Soluzione 4: Alla classe di sezioni critiche (A,B,..) vengono associate due variabili logiche libero1 e libero2 inizializzate al valore *false (0):*

```
int libero1=0;

int libero2=0;
```

```
/* processo P1: */
main()
{   ...
    libero1=1;
    while (libero2!=0);
    <sezione critica A>;
    libero1=0;

    ...
}
```

```
/* processo P2: */
main()
{   ...
    libero2=1;
    while (libero1!=0);
    <sezione critica B>;
    libero2=0;

    ...
}
```

- La soluzione assicura che **un solo processo alla volta** può trovarsi in una delle sezioni critiche.
- E' eliminato l'inconveniente della soluzione 2) in quanto la variabile *libero* associata ad un processo mantiene il valore *false* per tutto il tempo che il processo rimane all'esterno della sua sezione critica.
- Possono presentarsi condizioni in cui, a seconda della velocità relativa dei processi, questi **non possono entrare** nella loro sezione critica, pur essendo tali sezioni libere (**deadlock**).

  To : P1 pone *libero1= 1;*

  T1 : P2 pone *libero2 = 1;*

- P1 e P2 ripetono indefinitamente l'esecuzione di **while** senza poter entrare nelle rispettive sezioni critiche.

Soluzione 5: Nella soluzione precedente P1 pone **`libero1=1`** senza conoscere lo stato di P2; in particolare P1 non sa se P2 è pronto a porre **`libero2=1.`**

```
/* processo P1: */
main()
{   ...
    libero1=1;
    while (libero2!=0)
    {  libero1=0;
       while (libero2)
           libero1=1;
    }
    <sezione critica A>;
    libero1=0;
    ...
}
```

```
/* processo P2: */
main()
{   ...
    libero2=1;
    while (libero1!=0)
    {  libero2=0;
       while (libero1)
           libero2=1;
    }
    <sezione critica B>;
    libero2=0;
    ...
}
```

- Il processo P1 analizza lo stato della variabile *libero2*; se essa ha il valore *true*, cioè se il processo P2 è entrato nel prologo, P1 assegna il valore *false* alla variabile *libero1* e si mette in attesa che P2 abbia completato la sezione critica
- La stessa cosa fa il processo P2.
- Se i processi partono allo stesso istante e procedono alla stessa velocità entrambi ripetono indefinitamente i cicli dell'istruzione **while** e nessuno entra nella sezione critica (***deadlock***)

Soluzione 6: Algoritmo di *Dekker*

Garantisce le proprietà di *mutua esclusione* e di assenza di *deadlock*.

- Non elimina l'inconveniente che ad un processo venga indefinitamente impedito di entrare nella propria sezione critica pur essendo verificate le condizioni logiche per il suo accesso.
- Problema di **starvation:** la sezione critica viene ripetutamente eseguita da altri processi.
- Può essere esteso al caso di n processi

```
int libero1 =0;
int libero2 =0;
int turno=1; /*dominio {1,2}*/
```

- Il valore iniziale di *turno* è indifferente.

```
int libero1 =0;
int libero2 =0;
int turno=1; /*dominio {1,2}*/
```

```
/* processo P1: */
main()
{   ...
    libero1=1;
    while (libero2)
       if (turno==2)
       {   libero1=0;
           while(turno!=1);
           libero1=1;
        }
    <sezione critica A>;
    turno=2;
    libero1=0;
    ...
}
```

```
/* processo P2: */
main()
{   ...
    libero2=1;
    while (libero1)
       if (turno==1)
       {   libero2=0;
           while(turno!=2);
           libero2=1;
        }
    <sezione critica B>;
    turno=1;
    libero2=0;
    ...
}
```

Soluzione 7: Algoritmo di Peterson

- Risulta più semplice di quello di Dekker
- Elimina la possibilità di **starvation**
- Le variabili utilizzate per la sincronizzazione sono:

```
int libero1 =0;
int libero2 =0;
int turno=1; /*dominio {1,2}*/
```

```
int libero1 =0;
int libero2 =0;
int turno=1; /*dominio {1,2}*/
```

```
/* processo P1: */
main()
{ ...
 libero1=1;
 turno=2;
 while(libero2 && turno==2);
 <sezione critica A>;
 libero1=0;
    ...
}
```

```
/* processo P2: */
main()
{ ...
 libero2=1;
 turno=1;
 while(libero1 && turno==1);
 <sezione critica B>;
 libero2=0;
    ...
}
```

Proprieta` della soluzione al problema della mutua esclusione

a) Sezioni critiche della stessa classe devono essere eseguite in modo *mutuamente esclusivo.*

b) Quando un processo si trova all'esterno di una sezione critica *non può rendere impossibile* l'accesso alla stessa sezione (o a sezioni della stessa classe) ad altri processi.

c) Non deve essere possibile il verificarsi di situazioni in cui i processi *impediscono mutuamente* la prosecuzione della loro esecuzione (*deadlock*)

d) Se sono verificate le condizioni logiche per l'accesso ad una sezione critica da parte di un processo, questo non può essere *indefinitamente ritardato* a causa della esecuzione della stessa sezione (o di sezioni sella stessa classe) da parte di altri processi (*starvation*)

e) Devono essere eliminate *forme di attesa attiva (busy form of waiting)* bloccando l'esecuzione di un processo per tutto il tempo in cui non può avere accesso alla sezione critica.

A differenza delle altre proprietà l'ultima non riguarda la correttezza della soluzione ma *l'**efficienza** della realizzazione.*

Soluzioni hardware

- Nelle soluzioni precedenti si è supposto che l'hardware garantisca la mutua esclusione *solo a livello di lettura e scrittura di una singola parola di memoria.*
- L'*indivisibilità* è sempre assicurata solo riguardo all'ispezione o all'assegnamento di un valore ad una singola variabile comune.
- Molte macchine posseggono particolari istruzioni che consentano di *esaminare e modificare* il contenuto di una parola o di *scambiare* il contenuto di due parole *in un ciclo di memoria*.
- In questo caso è possibile dare una *semplice soluzione* al problema della mutua esclusione.

Lock e Unlock

```
void lock(int *x)
{    while (!*x);
     *x=0;
}
void unlock(int *x)
{    *x=1;
}
```

x *riferisce* una variabile logica associata ad una classe di sezioni critiche inizializzata al valore 1 (*true).*

(x=0 risorsa occupata, x=1 risorsa libera)

Lock e Unlock

- Soluzione al problema della mutua esclusione:

```
int x=1;
```

```
/* processo P1: */
main()
{ ...
 lock(&x);
 <sezione critica A>;
 unlock(&x);
       ...
}
```

```
/* processo P2: */
main()
{ ...
 lock(&x);
 <sezione critica B>;
 unlock(&x);
       ...
}
```

Si noti che a differenza della **`lock`**, l'operazione **`unlock`** è **indivisibile.**

Ipotesi: lock(x) e unlock(x) **operazioni indivisibili.**

➔ L'esecuzione contemporanea di due lock(x) (ciascuna su un diverso elaboratore) viene automaticamente sequenzializzata dall'hardware.

- I requisiti a),b),c) sono *soddisfatt*i. Il soddisfacimento del requisito d) *non è implicito* nella soluzione. Per superare l'inconveniente della *starvation* occorre un'opportuna realizzazione del *meccanismo di arbitraggio* per l'accesso in memoria.

- Il requisito e) *non è soddisfatto*, essendo presente nella lock una forma di ***attesa attiva***

Indivisibilita` delle operazioni lock e unlock

Istruzione test and set (x):

Consente la lettura e la modifica di una parola in memoria in modo *indivisibile*, cioè in un *solo ciclo di memoria.*

```
int test-and-set(int *a)
{ int R;
      R=*a;
      *a=0;
  return R;
}
```

- Operazione lock(x):

```
void lock(int *x)
{     while (!test-and-set(x));
}
```

Implementazione di lock e unlock

Se il set di istruzioni dell'architettura prevede la test-and-set (**`tsl`**):

```
lock(x):
   tsl register, x
   cmp register, 1
   jne lock
   ret
```

(copia x nel registro e pone x=0)
(il contenuto del registro vale 1?)
(se x=0 ricomincia il ciclo)
(ritorna al chiamante;
accesso alla sezione critica)

```
unlock(x):
   move x,1
   ret
```

(inserisce 1 in x)
(ritorna al chiamante)

Operazione EXCH A,X:

- Scambia i contenuti del registro A e della parola contenuta nell'indirizzo X in un *ciclo di memoria:*

```
void EXCH (int *a; int *b)
{ int temp;
  temp=*a;
  *a=*b;
  *b=temp;
}
```

```
void lock(int *x)
{  priv=1;
   do    EXCH (x,&priv)
   while (priv==1);
}
```

(*priv* è una variabile locale a ciascun processo)

Proprietà della soluzione basata su lock e unlock

- Si applica in ambiente *multiprocessore*.

- Va bene nel caso di sezioni critiche *molto brevi* (*attesa attiva*)

- Per ridurre al minimo questa attesa è opportuno *disabilitare il sistema di interruzioni* durante l'esecuzione della lock e unlock