💙

Lezione 7.1

Capitolo 5: Sincronizzazione dei processi

- Introduzione
- Problema della sezione critica
- Soluzione di Peterson
- Hardware per la sincronizzazione
- Lock Mutex
- Semafori
- Problemi tipici di sincronizzazione
- Monitor
- Esempi di sincronizzazione
- Approcci alternativi

Tipi di processi

Processi indipendenti:

- Ogni processo non può influire su altri processi nel sistema o subirne l’influsso, chiaramente, essi non condividano dati

Processi cooperanti:

- Ogni processo può influenzare o essere influenzato da altri processi in esecuzione nel sistema
- I processi cooperanti possono condividere dati

Un’esecuzione concorrente di processi cooperanti richiede meccanismi che consentono ai processi di comunicare tra oro e di sincronizzare le proprie azioni

Concorrenza

I temi centrali della progettazione di un SO sono connessi con la gestione di processi e thread:

- **Multiprogrammazione**: processi multipli in un sistema monoprocessore
- **Multiprocessing**: processi multipli in un sistema multiprocessore
- **Processi distribuiti**: gestione di processi multipli eseguiti su sistemi distribuiti

Concetto fondamentale è quello di **concorrenza**, che comprende diversi aspetti di progettazione:

- La comunicazione tra processi
- La condivisione e competizione per le risorse
- La sincronizzazione delle attività di processi multipli
- L’allocazione di tempo di processore ai processi, etc

La concorrenza appare in almeno tre diversi contesti:

- **Applicazioni multiple**: la multiprogrammazione è stata inventata per permettere di dividere dinamicamente il tempo di calcolo tra le applicazioni attive
- **Applicazioni strutturate**: alcune applicazioni possono essere programmate in maniera efficace come insiemi di processi concorrenti
- **Strutture del SO**: spesso anche i SO sono implementati come insieme di processi

In un sistema a singolo processore con multiprogrammazione, i processi sono alternati nel tempo per dare l’illusione dell’esecuzione simultanea

Nonostante l’overhead, l’esecuzione alternata porta benefici dal punto di vista dell’efficienza di esecuzione e della strutturazione dei programmi

In un sistema multiprocessore è possibile non solo alternare le esecuzioni dei processi ma anche sovrapporle

Principi della concorrenza

Nel caso di un singolo processore non è possibile predire le velocità relative dei processi:

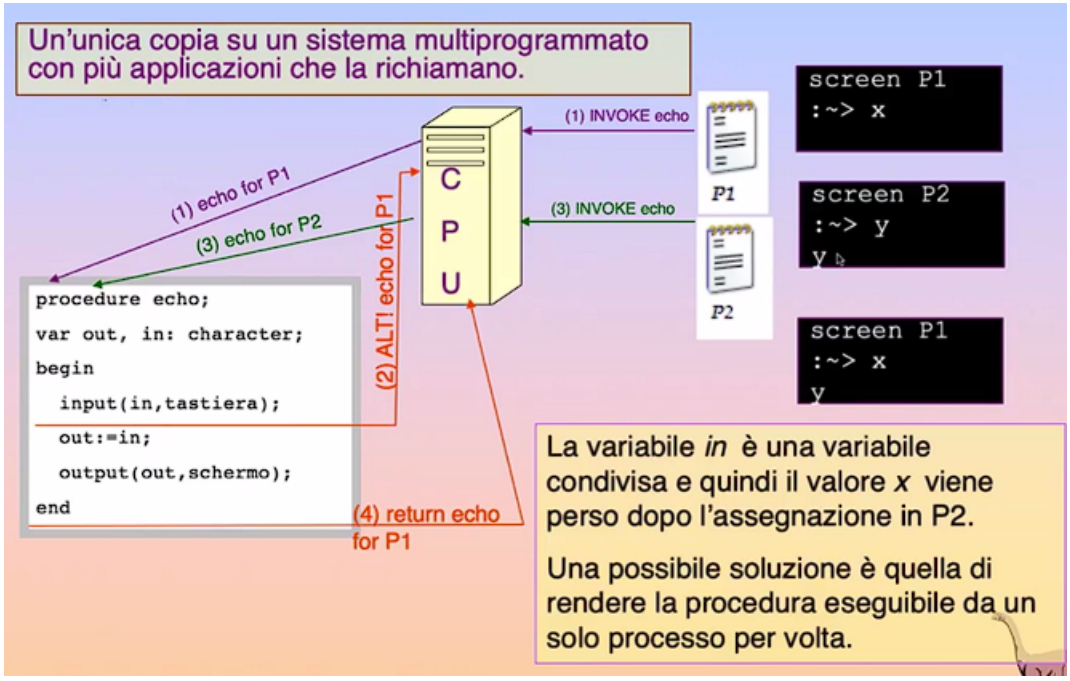
- Poiché queste dipendono dalle attività degli altri processi, dal modo in cui il SO gestisce gli interrupt e della politica di schedulazione del sistema

Si presentano queste difficoltà:

- La condivisione di risorse globali è pericolosa:
  - Es. Se due processi fanno usi della stessa variabile globale ed effettuano letture e scritture, allora l’ordine in cui le operazioni sono eseguite è critico
- Per il SO è difficile gestire l’assegnazione di risorse in maniera ottimale
  - Es. Un processo A può richiedere l’uso di un particolare canale di I/O ed essere sospeso prima di poterlo usare
  - Per il SO bloccare semplicemente il canale ed impedirne l’uso da parte degli altri processi può essere inefficiente
- Trovare un errore di programmazione diventa molto difficile perché i risultati e la sequenza di esecuzioni interlacciate dei vari processi non sono sempre facilmente riproducibili

Esempio:

Principi di concorrenza nei SO



Principi di progettazione e di gestione che nascono a causa della concorrenza:

- Il SO deve poter tenere traccia dei processi attivi
- Il SO deve allocare e deallocare varie risorse per ogni processo attivo, ad es.:
  - Tempo di elaborazione
  - Memoria
  - File
  - Dispositivi di I/O
- Il SO deve proteggere i dati e le risorse fisiche di ogni processo da interferenze da parte di altri processi
- Il risultato di un processo deve essere indipendente dalla sua velocità di esecuzione (relativamente a quella degli altri processi concorrenti)

Interazione fra processi

Grado di conoscenza	Relazione	Influenza che un processo ha su altri	Problemi di controllo
Processi che non si vedono tra di loro	Competizione	I risultati di un processo non dipendono dalle informazioni ottenute dagli altri Il tempo di esecuzione dei processi può cambiare	Mutua esclusione Stallo (risorse riutilizzabili) Starvation
Processi che vedono altri processi	Cooperazione tramite condivisione	I risultati di un processo possono dipendere dalle informazioni ottenute dagli	Mutua esclusione Stallo (risorse riutilizzabili)

indirettamente (es. oggetti condivisi)		altri Il tempo di esecuzione dei processi varia	Starvation Coerenza dei dati
Processi che vedono altri processi direttamente (usano primitive di comunicazione)	Cooperazione tramite comunicazione	I risultati di un processo possono dipendere dalle informazioni ottenute dagli altri Il tempo di esecuzione dei processi può cambiare	Stallo (risorse riutilizzabili) Starvation

Produttore e consumatore

Un processo produttore produce informazioni che sono consumate da un processo consumatore

Per permettere un'esecuzione concorrente, occorre disporre di un vettore di elementi che possano essere inseriti dal produttore e prelevati dal consumatore

Un produttore può produttore un elemento mentre il consumatore ne sta consumando un altro (sono concorrenti)

Produttore e consumatore devono essere sincronizzati in modo che il consumatore non tenti di consumare un elemento che non è stato ancora prodotto

Se la memoria fosse illimitata, non avremmo limiti alla dimensione del vettore, il consumatore potrebbe trovarsi ad attendere nuovi elementi, ma il produttore può sempre produrne

Tuttavia la memoria è limitata, la dimensione del vettore deve essere fissata

In questo caso il consumatore deve attendere se il vettore è vuoto e il produttore se è pieno

Il vettore può essere fornito dal sistema operativo attraverso l'uso di una funzione di comunicazione tra processi, oppure può essere implementato tramite memoria condivisa

L'accesso concorrente a dati condivisi può creare problemi di inconsistenza dei dati

Per mantenere la consistenza dei dati sono necessari meccanismi che assicurino l'esecuzione ordinata e coordinata dei processi cooperanti

La precedente soluzione del problema del produttore e del consumatore con memoria limitata consente la presenza contemporanea nel vettore d non più di **n - 1** elementi

- Facendo uso di memoria condivisa

Una soluzione per utilizzare tutti gli **n** elementi del vettore non è semplice:

- Potremmo aggiungere una variabile intera, `contatore`, inizializzata a 0, che:
  - Si incrementa ogni volta che si inserisce un nuovo elemento nel vettore
  - Si decrementa ogni volta che si preleva un elemento dal vettore

Buffer di dimensioni limitate:

```
// Dati condivisi
```

```
#define DIM_VETTORE 10

typedef struct {
  ...
} item;

item vettore[DIM_VETTORE];
int inserisci = 0;
int preleva = 0;
int contatore = 0;
```

```
// Processo produttore

item appena_prodotto;

while(1) {
  while (contatore == DIM_VETTORE) // Non fare niente
    vettore[inserisci] = appena_prodotto;
    inserisci = (inserisci + 1) % DIM_VETTORE;
    contatore++;
}

// Processo consumatore

item da_consumare;

while (1) {
    while (contatore == 0) // Non fare niente
    da_consumare = vettore[preleva];
    preleva = (preleva + 1) % DIM_VETTORE;
    contatore--;
}
```


Nella versione precedente una cella era utilizzata per il controllo del buffer, se pieno o vuoto

Gli statement `contatore++;` e `contatore--;` devono essere eseguiti **atomicamente**

Una operazione è **atomica** se non può essere interrotta fino al suo completamento

Lo statement `contatore++;` può essere implementato in linguaggio macchina come:

```
registro1 = contatore
registro1 = registro1 + 1
contatore = registro1
```

Lo statement `contatore--;` può essere implementato in linguaggio macchina come:

```
registro2 = contatore
registro2 = registro2 - 1
contatore = registro2
```

Se il produttore ed il consumatore tentano di accedere concorrentemente al buffer, le esecuzioni in linguaggi macchina dell'incremento e del decremento del contatore potrebbero interfogliarsi (**interleaving**)

Il risultato dell'interfogliamento dipende da come produttore e consumatore sono schedulati

Supponiamo che `contatore` sia inizializzato a 5 e che ci siano un elemento prodotto ed uno consumato

Un possibile interfogliamente delle esecuzioni di produttore e consumatore è:

produttore: `registro1 = contatore` (registro 1 = 5)

produttore: `registro1 = registro1 + 1` (registro 1 = 6)

consumatore: `registro2 = contatore` (registro2 = 5)

consumatore: `registro2 = registro2 - 1` (registro2 = 4)

produttore: `contatore = registro1` (contatore = 6)

consumatore: `contatore = registro2` (contatore = 4)

Il valore di `contatore` è quindi 4, ma il risultato corretto è 5

Race condition

La **race condition** è la situazione in cui i processi accedono e modificano gli stessi dati concorrentemente e i risultati dipendono dall'ordine degli accessi

Per prevenire le race condition, i processi concorrenti devono essere **sincronizzati**

Per evitare situazioni di questo tipo, in cui più processi accedono e modificano gli stessi dati in modo concorrente e i risultati dipendono dagli ordini degli accessi, occorre garantire che un solo processo alla volta possa modificare dati condivisi

Problema della sezione critica

Per evitare il problema della race condition si introduce l'idea di **sezione critica**:

- Una sezione critica è un segmento di codice nel quale il processo può modificare variabili comuni, scrivere un file, aggiornare tabelle, etc.

Quando un processo è in esecuzione nella propria sezione critica non si deve consentire a nessun altro processo di entrare in esecuzione nella propria sezione critica

L'esecuzione delle sezioni critiche da parte dei processi è **mutuamente esclusiva** nel tempo

Consideriamo di avere n processi in competizione per l'uso di dati condivisi

Ciascun processo ha un segmento di codice chiamato sezione critica, in cui avviene l'accesso ai dati condivisi

Problema:

- Assicurare che quando un processo P esegue la sua sezione critica, nessun altro processo Q possa eseguire la propria sezione critica

Il problema della sezione critica si affronta progettando un protocollo che i processi possono usare per cooperare

Ogni processo deve chiedere il permesso di entrare nella propria sezione critica

La sezione di codice che realizza questa richiesta si chiama **sezione di ingresso**

La sezione critica può essere eseguita da una **sezione di uscita**, e, la restante parte del codice è detta **sezione non critica**

Ciclicamente un processo eseguirà una sezione critica e una sezione non critica

Soluzione al problema della sezione critica

Per risolvere il problema della sezione critica, occorre soddisfare i tre seguenti requisiti:

1. **Mutua esclusione** - se il processo P_i è nella sua sezione critica, allora nessun altro processo può entrare nella sua sezione critica
2. **Progresso** - se nessun processo è in esecuzione nella sua sezione critica ed esiste qualche processo che desidera entrare nella propria sezione critica, allora la decisione riguardante la scelta del processo che potrà entrare per primo nella propria sezione critica non può essere rimandata indefinitamente
3. **Attesa limitata** - se un processo ha già richiesto l'ingresso nella sua sezione critica, esiste un limite al numero di volte che si consente ad altri processi di entrare nelle rispettive sezioni critiche prima che si accordi la richiesta del primo processo (i processi andranno ciclicamente in sezione critica, non ci sarà un processo a cui verrà posticipato l'ingresso in sezione critica)
    - Assumiamo che ogni processo sia eseguito a una velocità diversa da 0
    - Nessuna ipotesi sulla velocità relativa degli n processi

In un determinato momento è possibile che più processi in modalità kernel siano attivi: in questi casi il **codice del kernel** è soggetto a molte race condition.

Per gestire questa condizione si utilizzano due strategie:

- **kernel con diritto di prelazione** → Permette che un processo funzionante in modalità kernel sia sottoposto a prelazione (preferenza o meno rispetto un altro processo in modalità kernel), rinviandone l'esecuzione.
- **kernel senza diritto di prelazione** → Non permette di applicare la prelazione sui processi in modalità kernel: l'esecuzione di questo continuerà finché lui stesso non uscirà dalla modalità kernel, si blocchi o ceda volontariamente il controllo della CPU.

I kernel senza diritto di prelazione sono privi di race condition sulle strutture dati del kernel. Invece, bisogna stare attenti alla struttura dei codici per i kernel con diritto di prelazione

In ogni caos, i kernel con diritto di prelazione sono preferiti rispetto agli altri poiché hanno più prontezza nelle risposte, grazie al basso rischio di

eseguire processi per un tempo troppo duraturo

Soluzione per due processi

Ho due processi P_0 e P_1

Struttura generale di un tipico processo P:

```
do {
    [sezione di ingresso]
    sezione critica
    [sezione di uscita]
    sezione non critica
} while (true);
```

I processi possono utilizzare delle variabili condivise per sincronizzarsi

Algoritmo 1

Variabile condivise:

- **int turno;**

  inizialmente **turno = 0 (**potrà assumere valore 0 o 1)

- **turno == i** → P_i può entrare nella sua sezione critica

Processo P_i


```
do {
    while (turno != i);
        // sezione critica
    turno = j;
        // sezione non critica
} while (true);
```

Supponiamo di essere il processo P_i.

Quando diventa il punto di Pi (quindi quando turno = i) entra in sezione critica. Poi c'è il turno di Pj (a turno viene assegnato j). Pi entrerà poi in sezione non critica

soddisfa la mutua esclusione, ma non il progresso

Se turno == 0, P_1 non può entrare nella sua regione critica anche se P_0 è nella propria regione non critica

Questo algoritmo segue la mutua esclusione ma non il progresso (nel caso uno dei due processi non volesse entrare in sezione critica, l'altro dovrebbe aspettare, anche indefinitivamente in caso)

Dal libro:

La soluzione iniziale quella di far condividere una variabile intera `turno` inizializzata a 0 (oppure ad 1)

Se turno == i allora si permetta al processo P_i di entrare nella sua sezione critica

Questo algoritmo assicura che solo un processo si possa trovare in sezione critica (mutua esclusione), ma non soddisfa il progresso poiché richiede una stretta alternanza dei processi nell'esecuzione della sezione critica (ad esempio se turno == i, P_j non può entrare nella sua sezione critica, nonostante P_i si troi nella sua sezione non critica

Algoritmo 2

Variabili condivise:

- **boolean pronto[2];**

  inizialemnte **pronto[0] = pronto[1] = false**

- **pronto[i] == true** P_i pronto ad entrare nella sua sezione critica

Processo P_i:

```
do {
    pronto[i] = true;
    while (pronto[j]);
        // sezione critica
    pronto[i] = false;
        // sezione non critica
} while(true);
```

pronto[i] = true;

while (pronto[j]) (nel caso che pronto[j] fosse vero, cioè nel caso in cui Pj volesse andare in sezione critica, Pi lascerebbe andare prima Pj e poi andrebbe lui)

Spiegazione:

e il valore pronto[i] = tru, significa che P_i è pronto per entrare nella propria sezione critica.., quindi verifica che il processo Pj non sia pronto per entrare nella propria sezione critica. Se P_j fosse pronto, P_i attenderebbe fino a che P_j indica che non intende più restare nella sua sezione critica, cioè finche pront[j] non è false. A questo punto entrerebbe P_i e all'uscita assegnerebbe pronto[i] = false

Soddisfa la mutua esclusione, ma non il progresso

Anche questo non seguirebbe il progresso (si rischia di avere la situazione che entrambi vorrebbero andare in sezione critica (quindi avere le variabili pronto contemporaneamente uguale a true) e nessuno si muoverebbe, entrambi sarebbero fermi nei loro cicli while

turno può avere un unico valore, e farà in modo che per forza uno dei due processi verrà eseguito

Se infatti consideriamo la seguente sequenza di esecuzione:

- T_0: P_0 assegna pronto[0] = true;
- T_1: P_1 assegna pronto[1] = true;

(Si potrebbe avere un'interruzione della CPU immediatamente dopo il passo T_0 e la CPU passa all'altro processo)

P_0 e P_1 entrano in un ciclo infinito nelle rispettive istruzioni while

Questo algoritmo dipende in modo decisivo dalla esatta temporizzazione dei due processi

Algoritmo 3: soluzione di Peterson

Utilizziamo le variabili condivise dei due algoritmi precedenti

Processo P_i:

```
do {
    pronto[i] = true;
    turno = j;
    while(pronto[j] and turno == j);
        // sezione critica
    pronto[i] = false;
        // sezione non critica
} while(true);
```

// Aggiungi da qualche parte

Non assicurerà mai che due processi non ricevano dal saltacode lo stesso numero

Nel caso due processi o più processi abbiano lo stesso numero si va a vedere il PID del processo

a questo punto andrò a controllare il pid e servirò quello che ha il PID minore

L'array scelta[] è per quando i processi devono prendere il numero

Perché i numeri potrebbero essere uguali? Dipende da come vengono schedulati i process, è possibile che un processo k abbia terminato la sua slice di tempo prima di salvare il massimo ed è possibile che lo salvi un altro processo j (per la quale il massimo sarà uguale a quello del processo k), venga salvato il numero per j e poi venga salvato il numero per k. I due processi avranno lo stesso numero

FCFS (first come first serve)

//Aggiungi da qualche parte

Mutua esclusione

- P_i entra nella propria sezione critica solo se pronto[j] == false o turno == i
- Se entrambi i processi fossero contemporaneamente in esecuzione nelle rispettive sezioni critiche, si avrebbe pronto[i] == pronto[j] === true
- L'istruzione while non può essere eseguita da P_i e P_j contemporaneamente perché turno può assumere valore i o j, ma non entrambi
- Supponendo che P_j ha eseguito con successo l'istruzione while avremo che pronto[j] == rue e turo = j, condizione che persiste fino a che P_j si trova nella proprio sezione critica

Attesa limitata e progresso

- Si può impedire a un processo P_i di entrare nella propria sezione critica solo se questo è blocco nel ciclo while dalla condizione pronto[j] == true e turno == j
- Se P_j non è pronto per entrare, allora pronto[j] == false, e P_i può entrare nella propria sezione critica
- Se P_j è pronto per entrare, allora pronto[j] == true se turno == P_i, mentre se turno == j entra P_j
- Se entra P_j, all'uscita imposta pronto[j] == false e P_i può entrar
- P_i entra nella sezione critica (progresso) al massimo dopo un ingresso da perte di P_j (attesta limitata)

Soluzione per più processi: algoritmo del fornaio

E' stato sviluppato per risolvere il problema della sezione critica per n processi

Questo algoritmo si basa su uno schema di servizio usato "nelle panetterie "

Al suo ingresso nel negozio, ogni cliente riceve un numero; viene servito di volta in volta il cliente con il numero più basso.

Visto che l'algoritmo non assicura che due clienti (processi) non ricevano lo stesso numero, a parità di numero si serve prima il processo con PID minore: nel caso in cui P_i e P_j hanno lo stesso numero e i < j, viene servito prima P_i

Lo schema di numerazione genera sequenze crescenti di numeri: ad es. 1,2,3,3,3,3,4,5…

Notazione ≤ ordine lessicografico **(ticket #, process id#)**

- (a,b) < (c, d) if a < c oppure se a == c and b < d
- $max(a_0, ..., a_{n-1})$ è un numero k tale che k ≥ a, for i ≠ 0, …, n-1

Algoritmo del fornaio

Variabili condivise:

`boolean scelta[n];`

`int numero[n];`

inizializzate a `false` e `0` rispettivamente

```
do {
   scelta[i] = true;
   numero[i] = max(numero[0], numero[1], ..., numero [n - 1]) + 1;
   scelta[i] = false;
   for (j = 0; j < n; j++) {
       while (scelta[j]);
       while ((numero[j] != 0) && ((numero[j], j) < (numero[i], i)));
   }
       // sezione critica
   numero[i] = 0;
       // sezione non critica
} while (true);
```

j e i sono i pid dei processi

numero[j] e numero[i] sono i numeri del saltacoda

Mutua esclusione

- Se P_i si trova nella propria sezione critica e P_k(k ≠ i) ha già scelto il proprio numero[k] ≠ 0, allora (numero[i],i) < (numero[k],k)
- Se P_i è nella propria sezione critica e P_k tenta di entrare nella propria, il processo P_k esegue la seconda istruzione while per j == i, trova che:
  - numero[i] ≠ 0
  - (numero[i], i) < (numero[k], k)
- Quindi continua il ciclo nell'istruzione while fono a che P_i lascia la propria sezione critica

Progresso e attesa limitata

- Questi requisiti sono garantiti poiché i processi entrano nelle rispettive sezioni critiche secondo il criterio **FCFS**

Hardware per la sincronizzazione

In una macchina singolo processore:

- i processi concorrenti non possono sovrapporsi, possono solamente alternarsi
- quindi un processo continua l'esecuzione fino a quando non chiama un servizio del SO o viene interrotto

Quindi per garantire la mutua esclusione è sufficiente evitare che un processo venga interrotto

Questo può essere realizzato con apposite primitiva del kernel per abilitare e disabilitare gli interrupt:

```
reapeat
    <disattiva le interruzioni>;
    <sezione critica>;
    <attiva le interruzioni>
    <resto del programma>
forever
```

Mutua esclusione: supporto hardware

In un sistema multiprocessore non si garantirebbe la mutua esclusione

Problemi:

- In questo modo la mutua esclusione è garantita, però il processore non può più alternare i programmi liberamente
- Inoltre in un sistema multiprocessore disattivare gli interrupt non garantisce la mutua esclusione

In sistemi con più unità di elaborazione non è sufficiente evitare che un processo venga interrotto:

Esistono istruzioni, rese disponibili da alcune architetture, che possono essere impiegate efficacemente per risolvere il problema della sezione critica

Queste particolari istruzioni atomiche permettono:

- di controllare e modificare il contenuto di una parola di memoria
- oppure di scambiare il contenuto di due parole di memoria

Architetture di sincronizzazione

Passiamo ad un'architettura a k CPU: quello che è stato detto finora non funzionerà, c'è bisogno di trovare una soluzione diversa

Si possono utilizzare istruzioni hardware atomiche di due tipi: uno di queste è l'istruzione **TestAndSet**

Istruzione *TestAndSet*: modifica il contenuto di una variabile in maniera atomica

Restituisce un valore di tipo booleano e prendere come argomento un booleano

Copio in un booleano l'argomento di TestAndSet, in obiettivo memorizzo true e come valore di ritorno restituisco il vecchio valore di obiettivo

Se si eseguono contemporaneamente due istruzioni TestAndSet, ciascuna in una unità di elaboazione diversa, queste vengono eseguite in maniera sequenziale in un ordine arbitrario

```
boolean TestandSet(boolean *obiettivo) {
    boolean valore = *obiettivo;
    *obiettivo = true;
    return valore;
}
```


Posso scrivere un protocollo che garantisce la mutua esclusione

Mutua esclusione con TestandSet

Variabili condivise:

`boolean blocco = false;`

Processo P_i:

```
do {
    while (TestandSet(&blocco));
        // sezione critica
    blocco = false;
        // sezione non critica
} while (true);
```

Il primo processo che esegue il codice trova blocco a false restituisce false e setta blocco a true

Tutti gli altri processi avranno blocco a true e continuano a ciclare (busy waiting)

Una volta terminato il codice P_i setta blocco a false

Soddisfa la mutua esclusione ma non l’attesa limitata

Swap

Un’alta istruzione atomica analoga a TestAndSet è l’istruzione Swap

Agisce sul contentuo di due parole di memoria

```
void Swap(boolean *a, boolean *b) {
    boolean tmp = *a;
    *a = *b;
    *b = tmp;
}
```

Ha due booleani passati per riferimento (a e b) e fa lo swap di a e b

Mutua esclusione con Swap

Variabile condivisa (inizializzata a **false**):

`boolean blocco;`

Processo P_i:

```
do{
    chiave = true;
    while (chiave == true)
        Swap(&blocco, &chiave); // ricorda che chiave non è
                                // condivisa: c'è un continuo swap
                                // true e true (dopo che il primo processo
                                // ha fatto swap)
        // sezione critica
    blocco = false;
        // sezione non critica
} while (true);
```


Possiamo pensare di avere una situazione in cui posso garantire la mutua esclusione

Inizio con una variabile blocco inizializzata a false

Solo il primo che esegue quell’istruzione nel ciclo di while troverà blocco a false, tutti gli altri troveranno blocco a true

Per lo stesso identico discorso di prima, chi esce mette blocco a false e permette a un altro processo di entrare

C’è la mutua esclusione ma non c’è l’attesa limitata

(Non soddisfa il requisito di attesa limitata)

Mutua esclusione con attesa limitata con TestAndSet

Variabili condivise (inizializzate a **false**):

`boolean blocco;`

`blocco attesa[n];`

Processo P_i:

```
do {
    attesa[i] = true;
```

```
    chiave = true;
    while (attesa[i] && chiave) {
        chiave = TestAndSet(&blocco);
    }
    attesa[i] = false;
        // sezione critica
    j = (i + 1) % n;
    while ((j != i) && !attesa[j]) {
        j = (j + 1) % n;
    }
    if (j == i)
        blocco = false;
    else
        attesa[j] = false;
        // sezione non critica
} while (true);
```

chiave = TestAndSet(&blocco) blocco è true e chive è false

Pi sceglie un processo per sbloccarlo: parte da quello dopo di lui, e esso vuole andare in sezione critica allora lo sblocca

(j ≠ i) → non ho fatto tutto il giro, non sono tornato a i

Requisiti

Mutua esclusione:

- Un processo può entrare in sezione critica solo se attesa[i] == false oppure chiave == false
- chiave è impostata a false solo se si esegue TestAndSet
- Il primo processo che esegue TestAndSet trova chiave == false, tutti gli altri devono attendere
- attesa[i] può diventare false solo se un altro processo esce dalla propria sezione critica, e solo una variabile attesa[i] vale false

Progresso:

- Un processo che esce dalla sezione critica o imposta blocco a false oppure attesa[i] per un j ≠ i a false, consentendo quindi ad un processo che attende di entrare

Attesa limitata:

- Un processo, quando esce dalla sezione critica, scandisce il vettore attesa nell’ordinamento ciclico (i + 1, i + 2, …, n-1, 0, …, i - 1) e designa come prossimo processo il primo presente nella sezione di ingresso (attesa[j] == true)
- Ogni processo che lo chiede, entrerà in sezione critica entro n - 1 turni