Sincronizzazione tra processi – 1

- Processi **indipendenti** possono avanzare concorrentemente senza alcun vincolo di ordinamento reciproco
- In realtà molti processi condividono risorse e informazioni funzionali
  - Per gestire la condivisione occorrono meccanismi di **sincronizzazione di accesso**

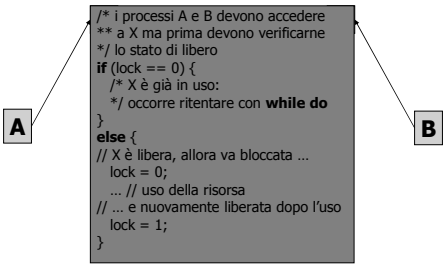
Sincronizzazione tra processi – 2

- Siano A e B due processi che condividono la variabile **X**, inizializzata al valore **10**
  - Il processo A deve incrementare **X** di **2** unità
  - Il processo B deve decrementare **X** di **4** unità
- A e B leggono **concorrentemente** il valore di **X**
  - Il processo A scrive in **X** il proprio risultato (**12**)
  - Il processo B scrive in **X** il proprio risultato (**6**)
- Il valore finale in **X** è l'ultimo scritto!
- Il valore atteso in **X** invece era **8**
  - Ottenibile **solo** con sequenze (A;B) o (B;A) **indivise** di lettura e scrittura

Sincronizzazione tra processi – 3

- La modalità di accesso indivisa a una variabile condivisa viene detta **in mutua esclusione**
  - L'accesso consentito a un processo inibisce quello simultaneo di qualunque altro processo utente fino al rilascio della risorsa
- Si utilizza una variabile logica "lucchetto" (*lock*) che indica quando la variabile condivisa è in uso a un altro processo
  - Detta anche struttura *mutex (mutual exclusion*)

Sincronizzazione tra processi – 4



Questa soluzione **non** funziona! Perché?

Sincronizzazione tra processi – 5

- La soluzione mostrata è totalmente inadeguata
  - Ciascuno dei due processi può essere prerilasciato **dopo** aver letto la variabile *lock* ma **prima** di esser riuscito a modificarla
    - Questa situazione è detta ***race condition*** e può generare pesanti inconsistenze
  - Inoltre l'algoritmo mostrato comporta **attesa attiva**, provocando spreco di tempo di CPU a scapito di altre attività a maggior valore aggiunto
    - La tecnica di sincronizzazione tramite attesa attiva viene detta *busy wait* (o *spin lock*)

Sincronizzazione tra processi – 6

- Tecniche alternative
  - **Disabilitazione delle interruzioni**
    - Previene il prerilascio dovuto all'esaurimento del quanto di tempo e/o la promozione di processi a più elevata priorità
    - Può essere inaccettabile per sistemi soggetti a interruzioni frequenti
  - Supporto *hardware* diretto: ***Test-and-Set-Lock***
    - Cambiare **atomicamente** valore alla variabile di *lock* se questa segnala "libero"
    - Evita situazioni di *race condition* ma comporta **sempre** attesa attiva

Sincronizzazione tra processi – 7

```
!! Chiamiamo regione critica la zona di programma
!! che delimita l'accesso e l'uso di una variabile
!! condivisa
enter_region:
 TSL R1, LOCK       !! modifica il valore di
                    !! LOCK (se vale 0) e lo pone in R1
 CMP R1, 0          !! verifica l'esito
 JNE enter_region   !! attesa attiva se =0
 RET                !! altrimenti ritorna al chiamante
                    !! con possesso della regione critica
leave_region:
 MOV LOCK, 0        !! scrive 0 in LOCK (accesso libero)
 RET                !! ritorno al chiamante
```


Sincronizzazione tra processi – 8

- Soluzione mediante **semaforo**
  - Dovuta a E. W. Dijkstra (1965)
  - Richiede accesso **indiviso** (atomico) alla variabile di controllo detta semaforo
    - Semaforo **binario** (contatore Booleano che vale 0 o 1)
    - Semaforo **contatore** (consente tanti accessi simultanei quanto il valore iniziale del contatore)
  - La richiesta di accesso, **P**, decrementa il contatore se questo non è già 0, altrimenti accoda il chiamante
  - L'avviso di rilascio, **V**, incrementa di 1 il contatore e chiede al *dispatcher* di porre in stato di "pronto" il primo processo in coda sul semaforo

Sincronizzazione tra processi – 9

L'uso di una risorsa condivisa **R** è racchiuso entro le chiamate di **P** e **V** sul semaforo associato a **R**

```
Processo ::
{ // avanzamento
  P(sem);
  // uso di risorsa R
  V(sem);
  // avanzamento
}
```

P(sem) viene invocata per richiedere accesso alla risorsa

V(sem) viene invocata per rilasciare la risorsa

Sincronizzazione tra processi – 10

Mediante uso intelligente di semafori binari, più processi possono anche **coordinare** l'esecuzione di attività collaborative

```
processo A ::
{// esecuzione indipendente
 …
 P(sem);
 // 2a parte del lavoro
 …
}
```

```
processo B ::
{// 1a parte del lavoro
 …
 V(sem);
 // esecuzione indipendente
 …
}
```

Contatore inizialmente a 0 (bloccante)

Sincronizzazione tra processi – 11

Un semaforo contatore è una struttura composta da un campo intero valore e da un campo lista che accoda tutti i **PCB** dei processi in attesa su quel semaforo

PCB: *Process Control Block*

```
void P(struct sem){
  sem.valore -- ;
  if (sem.valore < 0){
     put(self, sem.lista);
     suspend(self);
  };
}
void V(struct sem){
  sem.valore ++ ;
  if (sem.valore <= 0)
     wakeup(get(sem.lista));
}
```

P attende un evento (di rilascio) e se disponibile lo consuma
V notifica un evento (di rilascio)
`sem.val` > 0 denota eventi non consumati
`sem.val` < 0 denota eventi attesi ma non (ancora) notificati

Monitor – 1

- L'uso di semafori a livello di programma è ostico e rischioso
  - Il posizionamento improprio delle **P** può causare situazioni di blocco infinito (*deadlock*) o anche esecuzioni erronee di difficile verifica (*race condition*)
  - È indesiderabile lasciare all'utente il pieno controllo di strutture così delicate

Esempio 1

```
#define N …              /* posizioni del contenitore   */
typedef int semaforo;  /* P decrementa, V incrementa,
                          il valore 0 blocca la P     */
semaforo mutex = 1;
semaforo non-pieno = N;
semaforo non-vuoto = 0;

void produttore(){                  void consumatore(){
   int prod;                           int prod;
   while(1){                           while(1){
      prod = produci();                   P(&non-vuoto);
      P(&non-pieno);                      P(&mutex);
      P(&mutex);                          prod = preleva();
      inserisci(prod);                          V(&mutex);
      V(&mutex);                          V(&non-pieno);
      V(&non-vuoto);                      consuma(prod);
   }                                   }
}                                   }
```

Il corretto ordinamento di P e V è critico!

Monitor – 2

- Un diverso ordinamento delle **P** nel codice utente di Esempio 1 potrebbe causare situazioni di blocco infinito (*deadlock*)

Codice del produttore

```
P(&mutex);     /* accesso esclusivo al contenitore */
P(&non-pieno); /* attesa spazi nel contenitore */
```

- In questo modo il consumatore non può più accedere al contenitore per prelevarne prodotti, facendo spazio per l'inserzione di nuovi → stallo = *deadlock*

Monitor – 3

- Linguaggi evoluti di alto livello (e.g.: Concurrent Pascal, Ada, Java) offrono strutture **esplicite** di controllo delle regioni critiche, originariamente dette *monitor* (Hoare, '74; Brinch-Hansen, '75)
- Il *monitor* definisce la regione critica
- Il compilatore (non il programmatore!) inserisce il codice necessario al controllo degli accessi

Monitor – 4

- Un *monitor* è un aggregato di sottoprogrammi, variabili e strutture dati
- Solo i sottoprogrammi del *monitor* possono accederne le variabili interne
- Solo un processo alla volta può essere attivo entro il *monitor*
  - Proprietà garantita dai meccanismi del **supporto a tempo di esecuzione** del linguaggio di programmazione concorrente
  - Il codice necessario è inserito dal compilatore direttamente nel programma eseguibile

Monitor – 5

- La sola garanzia di mutua esclusione può **non** bastare ad affrontare il problema
- Due procedure operanti su variabili speciali (non contatori!) dette *condition variables*, consentono di modellare **condizioni logiche** specifiche del problema
  - **Wait**(<cond>) /* forza l'attesa del chiamante */
  - **Signal**(<cond>) /* risveglia il processo in attesa */
- Il segnale di risveglio **non ha memoria**
  - Va perso se nessuno lo attende

Esempio 2

```
monitor PC
 condition non-vuoto, non-pieno;
 integer contenuto := 0;
 procedure inserisci(prod : integer);
 begin
  if contenuto = N then wait(non-pieno);
  <inserisci nel contenitore>;
  contenuto := contenuto + 1;
  if contenuto = 1 then signal(non-vuoto);
 end;
 function preleva : integer;
 begin
  if contenuto = 0 then wait(non-vuoto);
  preleva := <preleva dal contenitore>;
  contenuto :=  contenuto – 1;
  if contenuto = N-1 then signal(non-pieno);
 end;
end monitor;
```

```
procedure Produttore;
begin
 while true do begin
  prod := produci;
  PC.inserisci(prod);
 end;
end;
```

```
procedure Consumatore;
begin
 while true do begin
  prod := PC.preleva;
  consuma(prod);
 end;
end;
```

Monitor – 6

- La primitiva **`Wait`** permette di bloccare il chiamante qualora le condizioni logiche della risorsa non consentano l'esecuzione del servizio
  - Contenitore pieno per il produttore
  - Contenitore vuoto per il consumatore
- La primitiva **`Signal`** notifica il verificarsi della condizione attesa al (primo) processo bloccato, risvegliandolo
  - Il processo risvegliato compete con il chiamante della **`Signal`** per il possesso della CPU
- **`Wait`** e **`Signal`** sono invocate in mutua esclusione
  - Non si può verificare *race condition*

Monitor – 7

- Java offre un costrutto simile al monitor tramite classi con metodi **`synchronized`**
  - Ma senza *condition variable*
- Le primitive **`wait`**() e **`notify`**() invocate all'interno di metodi **`synchronized`** evitano il verificarsi di *race condition*
  - In realtà il metodo **`wait`**() può venire interrotto, e l'interruzione va trattata come eccezione!

Esempio 3

```
class monitor{
 private int contenuto = 0;
 public synchronized void inserisci(int prod){
  if (contenuto == N) blocca();
  <inserisci nel contenitore>;
  contenuto = contenuto + 1;
  if (contenuto == 1) notify();
 }
 public synchronized int preleva(){
  int prod;
  if (contenuto == 0) blocca();
  prod = <preleva dal contenitore>;
  contenuto = contenuto - 1;
  if (contenuto == N-1) notify();
  return prod;
 }
 private void blocca(){
  try{wait();
  } catch(InterruptedException exc) {};}
}
```

```
static final int N = <…>;
static monitor PC =
   new monitor();
// … produttore …
PC.inserisci(prod);
// … consumatore …
prod = PC.preleva();
```

Attesa e notifica sono responsabilità del programmatore!

Monitor – 8

- In ambiente locale si hanno 3 possibilità
  1. Linguaggi concorrenti **con** supporto esplicito per strutture *monitor* (**alto livello**)
  - Linguaggi sequenziali **senza** supporto per *monitor* o semafori
    2. Uso di semafori tramite strutture primitive del sistema operativo e chiamate di sistema (**basso livello**)
    3. Realizzazione di semafori primitivi, in linguaggio *assembler*, senza supporto dal sistema operativo (**bassissimo livello**)
- Monitor e semafori **non sono utilizzabili** per realizzare scambio di informazione tra elaboratori
  - Perché?

Barriere

- Consentono di sincronizzare **gruppi** di processi
  - Attività cooperative suddivise in fasi ordinate
- La barriera blocca **tutti** i processi che la raggiungono fino all'arrivo dell'ultimo
  - Si applica indistintamente ad ambiente locale e distribuito
- Non comporta scambio di messaggi esplicito

Considerazioni conclusive

- Le soluzione ammissibili per gestire la sincronizzazione tra processi soddisfano **4 condizioni**
  1. Garanzia di accesso esclusivo
  2. Garanzia di attesa finita
  3. Nessuna assunzione sull'ambiente di esecuzione
  4. I processi esterni alla sezione critica non possono condizionarne l'accesso

Soluzioni "al limite"

- **Alternanza stretta per coppie di processi**
  - Regolata dal valore della variabile condivisa `turn`
  - 2 difetti importanti
    - Si basa su ***busy wait***
    - La regola di alternanza viola la condizione 4!

```
Processo 0 ::
while (TRUE) {
  while (turn != 0)
   /* busy wait */ ;
  inside_region();
  turn = 1;
  outside_region();
}
```

```
Processo 1 ::
while (TRUE) {
  while (turn != 1)
   /* busy wait */ ;
  inside_region();
  turn = 0;
  outside_region();
}
```


La soluzione di G. L. Peterson – 1

- Migliore della precedente ma "ingenua" rispetto agli elaboratori di oggi
- Applica anch'essa a **coppie** di processi

```
IN(int i) :: {
int j = (i – 1); // l'altro
flag[i] = TRUE;
turn = i;
while (flag[j] && turn == i)
 /* busy wait */ ; }
OUT(int i) :: {
flag[i] = FALSE; }
```

```
Processo (i = 0 / 1) ::
while (TRUE) {
  IN(i);
  /* sezione critica */
  OUT(i);
  /* altro lavoro    */
}
```


La soluzione di G. L. Peterson – 2

- La condizione di accesso in **IN**() stipula che al processo `i` tocchi attesa attiva fin quando l'altro processo non sia uscito dalla sezione critica
  - Indicato da `flag[j] = FALSE`
- Mentre su `flag[]` non vi può essere scrittura simultanea questa si ha invece su `turn`
  - Ma la condizione di accesso è espressa in modo da non incorrere in *race condition*
    - Si ha attesa attiva quando non vi sia coerenza tra il turno e la richiesta
    - Indipendentemente dal valore assunto da `turn`!

Problemi classici di sincronizzazione

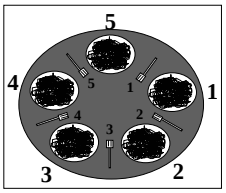
- Metodo per valutare l'efficacia e l'eleganza di modelli e meccanismi per la sincronizzazione tra processi
  - **Filosofi a cena** : accesso esclusivo a risorse limitate
  - **Lettori e scrittori** : accessi concorrenti a basi di dati
  - **Barbiere che dorme** : prevenzione di *race condition*
- Problemi pensati per rappresentare tipiche situazioni di rischio
  - Stallo con blocco (*deadlock*)
  - Stallo senza blocco (*starvation*)
  - Esecuzioni non predicibili (*race condition*)

Filosofi a cena – 1

- *N* filosofi sono seduti a un tavolo circolare
- Ciascuno ha davanti a se 1 piatto e 1 posata alla propria destra
- Ciascun filosofo necessita di 2 posate per mangiare
- L'attività di ciascun filosofo alterna pasti a momenti di riflessione

Filosofi a cena – 2

Soluzione A con stallo (*deadlock*)




```
void filosofo (int i){
  while (TRUE) {
    medita();
    P(f[i]);
    P(f[(i+1)%N]);
    mangia();
    V(f[(i+1)%N]);
    V(f[i]);
  };
}
```

L'accesso alla prima forchetta non garantisce l'accesso alla seconda!

Ogni forchetta modellata come un semaforo binario

Filosofi a cena – 3

Soluzione B con stallo (*starvation*)

```
void filosofo (int i){
 OK = FALSE;
 while (TRUE) {
  medita();
  while (!OK) {
   P(f[i]);
   if (!f[(i+1)%N]) {
    V(f[i]);
    sleep(T);}
   else {
    P(f[(i+1)%N]);
    OK = TRUE;
   };
  mangia();
  V(f[(i+1)%N]);
  V(f[i]);
 }
```

Un'attesa a durata costante difficilmente genera una situazione differente!

Filosofi a cena – 4

- Il problema ammette varie soluzioni
  1. Utilizzare in soluzione A un semaforo a mutua esclusione per incapsulare gli accessi a *entrambe* le forchette
     - Funzionamento garantito
  2. In soluzione B, ciascun processo potrebbe attendere un tempo **casuale** invece che fisso
     - Funzionamento non garantito
  3. Algoritmi sofisticati, con maggiore informazione sullo stato di progresso del vicino e maggior coordinamento delle attività
     - Funzionamento garantito

Stallo

Condizioni necessarie e sufficienti

- **Accesso esclusivo a risorsa condivisa**
- **Accumulo di risorse**
  - I processi possono accumulare nuove risorse senza doverne rilasciare altre
- **Inibizione di prerilascio**
  - Il possesso di una risorsa deve essere rilasciato volontariamente
- **Condizione di attesa circolare**
  - Un processo attende una risorsa in possesso del successivo processo in catena

Stallo: prevenzione – 1

- Almeno tre strategie per affrontare lo stallo
  - **Prevenzione**
    - Impedire almeno una delle condizioni precedenti
  - **Riconoscimento e recupero**
    - Ammettere che lo stallo si possa verificare
    - Essere in grado di riconoscerlo
    - Possedere una procedura di recupero (sblocco)
  - **Indifferenza**
    - Considerare trascurabile la probabilità di stallo e **non** prendere alcuna precauzione contro di esso
      - Che succede se esso si verifica?

Stallo: prevenzione – 2

- **Prevenzione sulle condizioni necessarie e sufficienti**
  1. **Accesso esclusivo alla risorsa**
     - Alcune risorse non consentono alternative
  2. **Accumulo di risorse**
     - Assai ardua da eliminare
  3. **Inibizione del prerilascio**
     - Alcune risorse non consentono alternative
  4. **Attesa circolare**
     - Di riconoscimento difficile ed oneroso

Stallo: prevenzione – 3

- **Prevenzione sulle richieste di accesso**
  1. A ogni richiesta di accesso verificare se questa può portare allo stallo
     - In caso affermativo non è però chiaro cosa convenga fare
     - La verifica a ogni richiesta è un onere molto pesante
  2. All'avvio di ogni processo richiedere preventivamente quali risorse essi utilizzeranno così da ordinarne l'attività in maniera conveniente

Stallo: prevenzione – 4

- **Riconoscimento a tempo d'esecuzione**
  - Assai oneroso
    - Occorre **bloccare** periodicamente l'avanzamento del sistema per analizzare lo stato di tutti i processi e verificare se quelli in attesa costituiscono una lista circolare chiusa
  - Lo sblocco di uno stallo comporta la terminazione forzata di uno dei processi in attesa
    - Il rilascio delle risorse liberate sblocca la catena di dipendenza circolare