

### 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 loro condivisione servono meccanismi di **sincronizzazione di accesso**

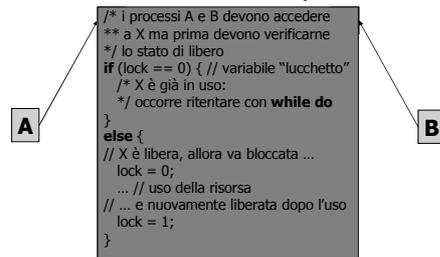
### Sincronizzazione tra processi – 2

- 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<sub>A</sub>** il proprio risultato (**12**)
  - Il processo B scrive in **X<sub>B</sub>** il proprio risultato (**6**)
- Il valore finale in **X** è l'ultimo tra **X<sub>A</sub>** e **X<sub>B</sub>** a essere 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 se la variabile condivisa è al momento 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 appena vista è completamente 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 richiede **attesa attiva** che causa 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*

### Sincronizzazione ammissibile

- Una soluzione al problema della sincronizzazione di processi è ammissibile se soddisfa le seguenti 4 condizioni
  1. Garantire accesso esclusivo
  2. Garantire attesa finita
  3. Non fare assunzioni sull'ambiente di esecuzione
  4. Non subire condizionamenti dai processi esterni alla sezione critica

## Soluzioni "esotiche" – 1

- Mutua esclusione con variabili condivise tramite **alternanza stretta** tra coppie di processi
  - Non ha bisogno di supporto dalla sua macchina virtuale

```

Processo 0 ::
while (TRUE) {
  while (turn != 0)
    /* busy wait */;
  critical_region();
  turn = 1;
  ...;
}

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

Comando di alternanza (fuori dalla sezione critica)

- Tre gravi difetti → soluzione non ammissibile!
  - Uso di attesa attiva (*busy wait*)
  - Condizionamento esterno alla sezione critica
  - Rischio di *race condition* sulla variabile di controllo

## Soluzioni "esotiche" – 2

- Proposta da G. L. Peterson
  - Migliore (!) della precedente ma ingenua rispetto al funzionamento dei processori di oggi
  - Applica solo a coppie di processi

```

IN(int i) :: {
int j = (i - 1); // l'altro
flag[i] = TRUE;
turn = i;
while (flag[j] && turn == j)
  { // attesa attiva };
}

OUT(int i) :: {
flag[i] = FALSE; }

Processo (int i) ::
while (TRUE) {
  IN(i);
  // sezione critica
  OUT(i);
  // altre attività
}
    
```

## Soluzioni "esotiche" – 3

- La condizione di uscita in **IN()** impone che il processo *i* resti in attesa attiva fin quando il processo *j* non abbia invocato **OUT()**
  - Per cui `flag[j] = FALSE`
  - Si ha attesa attiva quando non vi sia coerenza tra la richiesta di `turn` e lo stato dell'altro processo
- Su `flag[]` non vi può essere scrittura simultanea che si ha invece su `turn`
  - La condizione di uscita è però espressa in modo da evitare il rischio di *race condition*
    - Non viene decisa solo dal valore assunto da `turn`!

## Sincronizzazione tra processi – 6

- Tecniche complementari e/o 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
    - Per questo la struttura semaforo si appoggia sopra una macchina virtuale meno potente che fornisce una modalità di accesso indiviso più primitiva
    - 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** (**up**) decrementa il contatore se questo non è già 0, altrimenti accoda il chiamante
  - L'avviso di rilascio **V** (**down**) 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 Pi ::
{ // avanzamento
  P(sem);
  /* uso di risorsa
  condivisa R */
  V(sem);
  // avanzamento
}
```

P(sem) viene invocata per richiedere accesso a una risorsa condivisa R

- Quale R tra tutte?
- V(sem) viene invocata per rilasciare la risorsa

Sincronizzazione tra processi

Sistemi Operativi - T. Vardanega

Pagina 32/57

## Sincronizzazione tra processi – 10

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

– Esempio con semaforo inizialmente bloccato

```
processo A ::
{ // esecuzione indipendente
  ...
  P(sem); // attesa di B
  // 2a parte del lavoro
  ...
}

processo B ::
{ // la parte del lavoro
  ...
  V(sem); // rilascio di A
  // esecuzione indipendente
  ...
}
```

Sincronizzazione tra processi

Sistemi Operativi - T. Vardanega

Pagina 33/57

## Sincronizzazione tra processi – 11

- Il **semaforo binario** (*mutex*) è una struttura composta da un campo valore intero e da un campo coda che accoda tutti i **PCB** dei processi in attesa sul semaforo
  - PCB = *Process Control Block*
- L'accesso al campo valore deve essere **atomico!**

```
void P(struct sem){
  if (sem.valore = 1)
    sem.valore = 0; // busy
  else {
    suspend(self, sem.coda);
    schedule();
  }
}

void V(struct sem){
  sem.valore = 1; // free
  if not_empty(sem.coda){
    ready(get(sem.coda));
    schedule();
  }
}
```

Sincronizzazione tra processi

Sistemi Operativi - T. Vardanega

Pagina 34/57

## Sincronizzazione tra processi – 12

- Il **semaforo contatore** ha la stessa struttura del *mutex* ma usa una logica diversa per il campo valore
  - (Valore > 0) denota disponibilità non esaurita
  - (Valore < 0) denota richieste pendenti
- Il valore iniziale denota la capacità massima della risorsa

```
void P(struct sem){
  sem.valore -- ;
  if (sem.valore < 0){
    suspend(self, sem.coda);
    schedule();
  }
}

void V(struct sem){
  sem.valore ++ ;
  if (sem.valore <= 0){
    ready(get(sem.coda));
    schedule();
  }
}
```

Sincronizzazione tra processi

Sistemi Operativi - T. Vardanega

Pagina 35/57

## 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 erranee di difficile verifica (*race condition*)
  - È indesiderabile lasciare all'utente il pieno controllo di strutture così delicate

Sincronizzazione tra processi

Sistemi Operativi - T. Vardanega

Pagina 36/57

## 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(){
  int prod;
  while(1){
    prod = produci();
    P(&non-pieno);
    P(&mutex);
    inserisci(prod);
    V(&mutex);
    V(&non-vuoto);
  }
}

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

**Il corretto ordinamento di P e V è critico!**

Sincronizzazione tra processi

Sistemi Operativi - T. Vardanega

Pagina 37/57

## 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 accedervi 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
    - Funzionalmente molto simile al *kernel* del sistema operativo
  - Il codice necessario è inserito dal compilatore direttamente nel programma eseguibile

## Monitor – 5

- La garanzia di mutua esclusione da sola può **non** bastare per consentire sincronizzazione intelligente
- 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;
consumo(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 per supportare sincronizzazione tra processi
  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

- Per 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
  - L'avvenuta sincronizzazione dice implicitamente ai processi del gruppo che tutti hanno raggiunto un dato punto della loro esecuzione

## Problemi classici di sincronizzazione

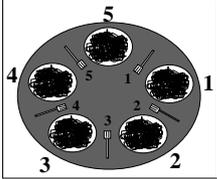
- Metodo per valutare l'efficacia e l'eleganza di modelli e meccanismi per la sincronizzazione tra processi
  - **Filosofi a cena** : accesso esclusivo a risorse limitate
  - **Lettori e scrittori** : accessi concorrenti a basi di dati
  - **Barbieri che dorme** : prevenzione di *race condition*
- Problemi 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*)



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

```

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

Ogni forchetta modellata come un semaforo binario

Sincronizzazione tra processi
Sistemi Operativi - T. Vardanega
Pagina 50/57

## 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!

Sincronizzazione tra processi
Sistemi Operativi - T. Vardanega
Pagina 51/57

## Filosofi a cena – 4

- Il problema ammette diverse 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

Sincronizzazione tra processi
Sistemi Operativi - T. Vardanega
Pagina 52/57

## 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

Sincronizzazione tra processi
Sistemi Operativi - T. Vardanega
Pagina 53/57

## 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?

Sincronizzazione tra processi
Sistemi Operativi - T. Vardanega
Pagina 54/57

## Stallo: prevenzione – 2

- Bisogna impedire il verificarsi di almeno una delle condizioni necessarie e sufficienti
  - Si può fare staticamente (prima di eseguire) oppure a tempo d'esecuzione
    1. **Accesso esclusivo alla risorsa**
      - Però alcune risorse non consentono alternative
    2. **Accumulo di risorse**
      - Però molti problemi richiedono l'uso simultaneo di più risorse
    3. **Inibizione del prerilascio**
      - Però alcune risorse non consentono di farlo
    4. **Attesa circolare**
      - Difficile da rilevare e complessa da evitare o sciogliere

Sincronizzazione tra processi
Sistemi Operativi - T. Vardanega
Pagina 55/57

## Stallo: prevenzione – 3

- **Prevenzione sulle richieste di accesso**
  - A tempo d'esecuzione
    1. A ogni richiesta di accesso si verifica se questa possa portare allo stallo
      - In caso affermativo non è però chiaro cosa convenga fare
      - La verifica a ogni richiesta è un onere molto pesante
  - Prima dell'esecuzione
    2. All'avvio di ogni processo si verifica quali risorse essi dovranno utilizzare così da ordinarne l'attività in maniera conveniente

Sincronizzazione tra processi

Sistemi Operativi - T. Vardanega

Pagina 56/57

## Stallo: riconoscimento – 1

- **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
- **Staticamente**
  - Può essere un problema non risolvibile!

Sincronizzazione tra processi

Sistemi Operativi - T. Vardanega

Pagina 57/57