Sincronizzazione tra processi - 1

- Processi *indipendenti* possono avanzare concorrentemente senza alcun vincolo di ordinamento reciproco
- In realtà, molti processi condividono risorse ed informazioni funzionali
  - La condivisione richiede l'introduzione di meccanismi di sincronizzazione di accesso

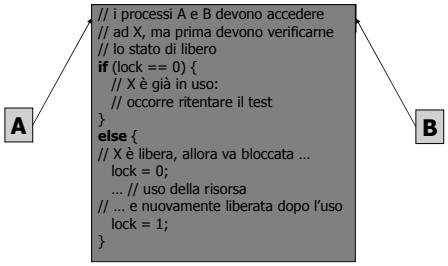
Sincronizzazione tra processi - 2

- 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 ad una variabile condivisa viene detta in mutua esclusione
  - L'accesso consentito ad un processo inibisce quello simultaneo di qualunque altro
- Si utilizza una variabile logica "lucchetto" (*lock*) che indica quando la variabile condivisa è in uso ad un altro processo
  - Anche detta *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 testato 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, con spreco di tempo di CPU a scapito di altre attività a maggior valore aggiunto
    - La tecnica di sincronizzazione tramite attesa attiva viene detta *busy waiting* (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 ad 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 ad 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 ad **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 appropriato uso di semafori binari, 2 processi A e B 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){
  if (sem.valore <= 0){
     put(self, sem.lista);
     suspend(self);
  };
  sem.valore -- ;
}
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