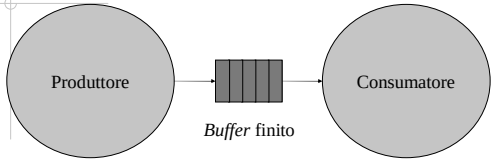
Problemi classici di sincronizzazione

Due esercizi sulla sincronizzazione dei processi

Motivazioni

- Metodo per valutare l'efficacia e l'eleganza di **modelli** e **meccanismi** per la sincronizzazione tra processi
  - **Lettori - scrittori** : accessi concorrenti a basi di dati
    - Chiamato anche "produttore – consumatore" sia al singolare che al plurale (differenza importante!)
  - **Filosofi a cena** : accesso esclusivo a risorse limitate
  - **Barbiere che dorme** : prevenzione di *race condition*
- Problemi pensati per illustrare tipiche situazioni di rischio
  - Stallo con blocco (*deadlock*)
  - Stallo senza blocco (*starvation*)
  - Esecuzioni non predicibili (*race condition*)

Produttore - Consumatore



- Non si scrive su *buffer* pieno
- Non si legge da *buffer* vuoto
- Produttore e consumatore lavorano a velocità diverse

Soluzione 1 (a)

```
#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);
   }                                  }
}                                  }
```


Soluzione 1 (b)

- La correttezza della soluzione 1 dipende dal corretto posizionamento delle P e V in entrambi i processi!
  - Soluzione estremamente fragile
- Cosa accade se invertiamo le P nel codice del processo produttore?

Codice del produttore

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


Soluzione 2 (a)

```
int itemCount // condivisa ma non protetta

procedure producer() {
   while (true) {
      item = produceItem();

      if (itemCount == BUFFER_SIZE) {
         sleep() // si sospende
      }

      putItemIntoBuffer(item);
      itemCount = itemCount + 1

      if (itemCount == 1) {
         wakeup(consumer);
      }
   }
}
```

```
procedure consumer() {
   while (true) {

      if (itemCount == 0) {
         sleep() // si sospende
      }

      item = removeItemFromBuffer();
      itemCount = itemCount - 1

      if (itemCount == BUFFER_SIZE - 1) {
         wakeup(producer);
      }

      consumeItem(item);
   }
}
```

Soluzione 2 (b)

- **Ausili**
  - La primitiva **sleep** sospende il processo chiamante
  - La primitiva **wakeup** (<*proc*>) invia un segnale di sveglia al processo <*proc*> designato
    - Il segnale si perde se il processo non è sospeso
- **Difetti**
  - *Race condition* su ItemCount che porta a stallo
    - Sospensione non legata atomicamente allo stato del *buffer*
    - Il prerilascio del consumatore prima di **sleep** porta i due processi a diversa percezione dello stato del sistema
      - Da cui deriva stallo

Soluzione 3 (a)

```
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;
```


Soluzione 3 (b)

- L'uso del **monitor** elimina il rischio di *race condition*
- Però il programmatore deve comunque scegliere le condizioni logiche giuste per invocare **wait** e **signal** sulle variabili di condizionamento
  - Meno rischio di errore perché c'è un solo **monitor** invece che diversi utilizzatori di primitive di sincronizzazione

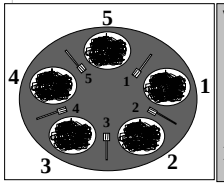
Filosofi a cena

- **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

Soluzione 1

Soluzione 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]);
  };
}
```


Soluzione 2

Soluzione 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!

Soluzione 3

- 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 13/13