Comunicazione e sincronizzazione tra processi
Indice

Comunicazione e sincronizzazione
Monitor - 1

- L'uso di semafori a livello di programma è difficile e rischioso
  - Il posizionamento improprio delle **P** può causare situazioni di blocco infinito (*deadlock*) ed 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, 0
blocca */
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!

Comunicazione e sincronizzazione
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*

Comunicazione e sincronizzazione
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

Comunicazione e sincronizzazione
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 cui codice è inserito dal compilatore nel programma eseguibile

Comunicazione e sincronizzazione
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;
```


Comunicazione e sincronizzazione
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*

Comunicazione e sincronizzazione
Monitor - 7

- Java offre un costrutto simil-monitor, tramite classi con metodi **synchronized**, ma senza *condition variables*
- 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!

Comunicazione e sincronizzazione
Monitor - 8

- In ambiente locale si hanno 3 possibilità
  - (1) Linguaggi concorrenti con supporto esplicito per strutture monitor (alto livello)
  - Linguaggi sequenziali senza alcun 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 alcun supporto dal sistema operativo (bassissimo livello)
- Monitor e semafori non sono utilizzabili per realizzare scambio di informazione tra elaboratori

Comunicazione e sincronizzazione
Scambio messaggi - 1

- Utilizzabile allo stesso modo sia in locale che in remoto
  - Caratteristica assai desiderabile
- Due dimensioni di scelta di modello
  - Sincrono / asincrono
  - Mediante chiamate di sistema / costrutti di linguaggio

Comunicazione e sincronizzazione
Scambio messaggi - 2

- Primitive di scambio messaggi
  - **`Send`** (destinatario, messaggio)
    // per copia o per riferimento
  - **`Receive`** (mittente, messaggio)
    // per copia o per riferimento
- Struttura di comunicazione
  - Canale simmetrico
    - 1 mittente : 1 destinatario
  - Canale asimmetrico
    - 1 mittente : N destinatari
    - N mittenti : 1 destinatario
  - Casella postale (*mailbox*)
    - N mittenti : M destinatari

Comunicazione e sincronizzazione
Scambio messaggi - 3

- Comunicazione sincrona (*rendezvous*)
  - Mittente e destinatario sono entrambi titolari di uno stesso canale di comunicazione
  - Canale tipicamente simmetrico

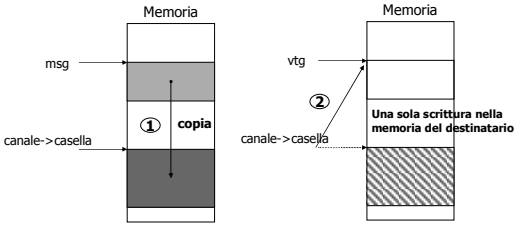
Mutua esclusione!

```
Send(canale, msg){
   if canale->stato {
      canale->casella = msg;
      canale->stato = false;
      <sveglia destinatario>;}
   else {
①     canale->stato = true;
      canale->casella = msg;
      <sospendi mittente>;}
}
```

```
Receive(canale, vtg){
   if canale->stato {
      vtg = canale->casella;
      canale->stato = false;
      <sveglia mittente>;}
   else {
②     canale->stato = true;
      &canale->casella = &vtg;
      <sospendi destinatario>;}
}
```


Comunicazione e sincronizzazione
Scambio messaggi - 4

Visione lato mittente Visione lato destinatario



Casella postale (*mailbox*) ad 1 posizione

Comunicazione e sincronizzazione
Scambio messaggi - 5

- Comunicazione asincrona
  - Il mittente deposita il messaggio e procede
  - Il destinatario attende la consegna per procedere
  - Richiede una casella capace di N posizioni, per N messaggi che il mittente può depositare prima di un prelievo
    - N è il grado di asincronia
  - Il mittente scrive direttamente in `vtg` se il destinatario è in attesa, oppure nella prima posizione libera, altrimenti si sospende
    - Rischio di attesa infinita (meglio sovrascrittura su casella strutturata ad anello)

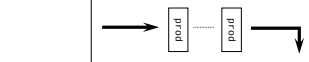
Comunicazione e sincronizzazione
Scambio messaggi - 6

- In presenza di linee di comunicazione inaffidabili
  - Conferma di ricezione (*acknowledgment*)
    - Per prevenire la perdita del messaggio inviato
  - Controllo della duplicazione
    - Per prevenire la perdita della conferma di ricezione
- Identificazione non ambigua di mittente e destinatario
  - Denominazione (*naming*)
  - Autenticazione (*authentication*)

Esempio 4

```
void produttore(){
 int prod;
 message msg;
 do {
  prod = produci();
  receive(consumatore,&msg);
  crea(&msg,prod);
  send(consumatore,&msg);
 } while (1);
}
```

Invia messaggi con prodotto




```
void consumatore(){
 int prod;
 message msg;
 crea(&msg, NULL);
 for(I=0;I<N;I++)
  send(produttore,&msg);
 do {
  receive(produttore,&msg);
  prod = estrai(&msg);
  consuma(prod);
  send(produttore,&msg);
 } while (1);
}
```

Restituisce messaggi vuoti

Inizialmente N, poi 1 per volta

- Modalità di comunicazione asincrona, per riferimento
- Messaggi non ricevuti accodati nel canale di comunicazione

Comunicazione e sincronizzazione
Barriere

- Consente 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

Comunicazione e sincronizzazione
Problemi classici

- Metodo per valutare l'efficacia e l'eleganza di modelli e meccanismi per la comunicazione e 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*)

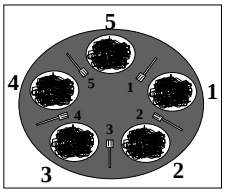
Comunicazione e sincronizzazione
Filosofi a cena - 1

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

Comunicazione e sincronizzazione
Filosofi a cena - 2

Soluzione A con stallo (*deadlock*)

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




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

Ogni forchetta modellata come un semaforo binario

Comunicazione e sincronizzazione
Filosofi a cena - 3

Soluzione B con stallo (*starvation*)

```
void filosofo_i_esimo(){
 OK = 0;
 do {
  medita();
  do {
   P(f[i]);
   if (!f[(i+1)%N]) {
    V(f[i]);
    sleep(T);}
   else {
    P(f[(i+1)%N]);
    OK = 1;
   } while (!OK);
  mangia();
  V(f[(i+1)%N]);
  V(f[i]);
 } while (1); }
```

Un attesa a durata fissa difficilmente genera una situazione differente!

Comunicazione e sincronizzazione
Filosofi a cena - 4

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

Comunicazione e sincronizzazione
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

Comunicazione e sincronizzazione
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, ma essere in grado di riconoscerlo e possedere una procedura di recupero (sblocco)
  - **Indifferenza**
    - Considerare molto bassa la probabilità di stallo e non prendere alcuna precauzione contro di esso
      - Che succede se esso si verifica?

Comunicazione e sincronizzazione
Stallo: prevenzione - 2

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

Comunicazione e sincronizzazione
Stallo: prevenzione - 3

- Prevenzione sulle richieste di accesso
  - Ad ogni richiesta di accesso, verificare se questa può portare allo stallo
    - In caso affermativo non è però chiaro cosa convenga fare
    - La verifica è un onere pesante ad ogni richiesta
  - Una alternativa è richiedere preventivamente a tutti i processi quali risorse essi richiederanno così da ordinarne l'attività in maniera conveniente

Comunicazione e sincronizzazione
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