

# Sistemi Operativi

## Sincronizzazione tra Processi

Docente: Claudio E. Palazzi

[cpalazzi@math.unipd.it](mailto:cpalazzi@math.unipd.it)

# 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 – 2a

- 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**
  - Ovvero A legge che **X** è **10** ma viene prerilasciato prima di poter eseguire la sua operazione, così anche B legge che **X** è **10**
  - A questo punto entrambi eseguiranno le rispettive operazioni sul valore originale di **X**, in concorrenza, ma senza tener conto dell'operazione svolta dall'altro processo
  - 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**)
  - Ovviamente può accadere anche il viceversa (prima legge B e poi A, ecc.)

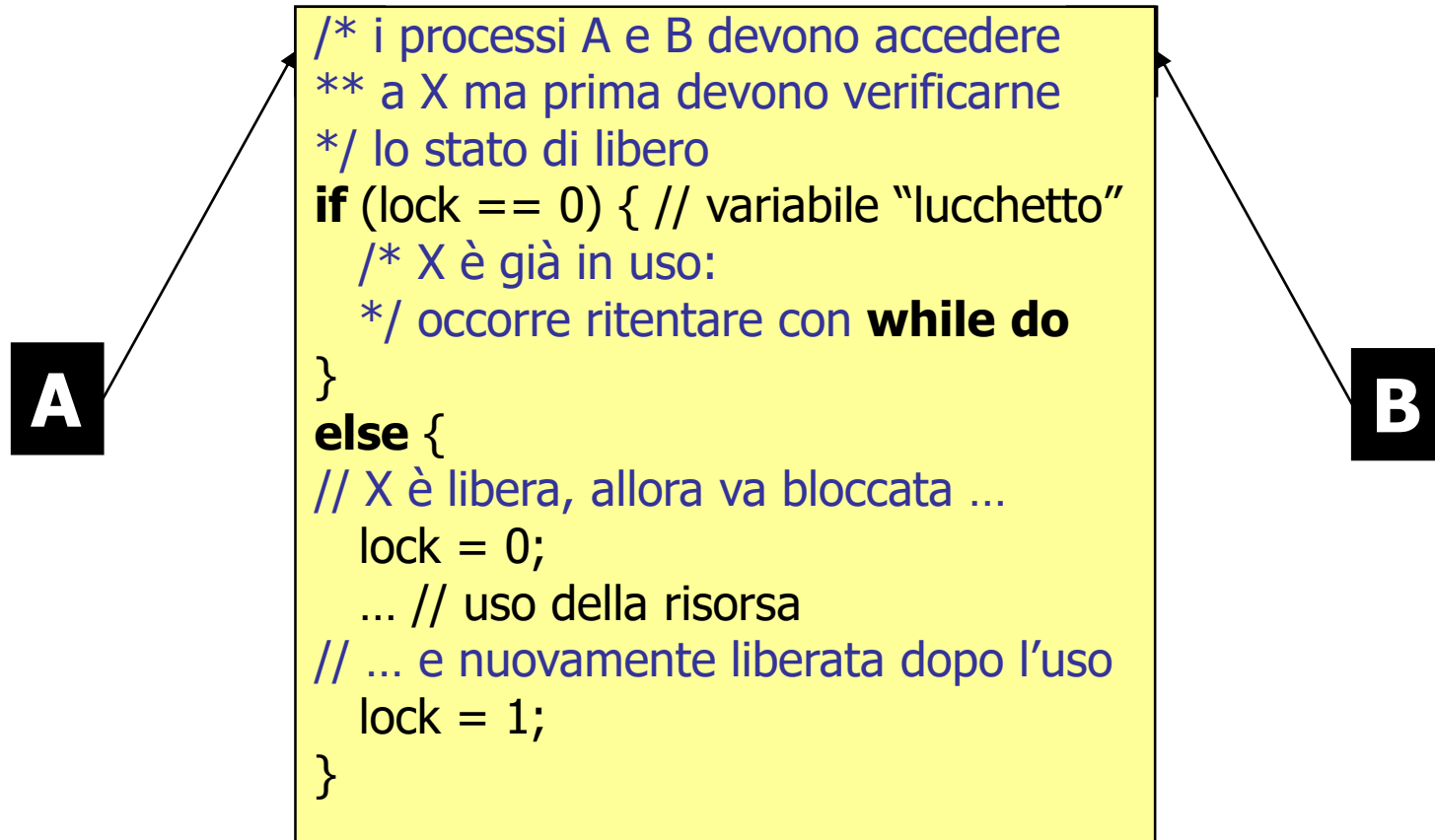
# Sincronizzazione tra processi – 2b

- Il valore finale in **X** è l'ultimo tra **X<sub>A</sub>** e **X<sub>B</sub>** a essere scritto!
  - **12** oppure **6**
- Il valore atteso in **X** invece era **8**
  - Ottenibile **solo** con sequenze (A;B) o (B;A) **indivise** di lettura e scrittura
  - Ovvero, una volta che uno dei due processi ha iniziato a maneggiare (anche solo leggere) la variabile **X**, all'altro processo non deve essere consentito l'accesso alla variabile (non deve essere consentita l'esecuzione di istruzioni che maneggiano la variabile **X**)
- Questo problema è detto *race condition*: il risultato finale dell'esecuzione dei processi dipende dalla temporizzazione o dalla sequenza con cui vengono eseguite le loro istruzioni.
- *Sezione critica (o regione critica)*: la porzione di codice che delimita l'accesso a una risorsa condivisa tra più flussi di esecuzione di un sistema concorrente
  - È dove si determina la race condition

# 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*)
- Ma come implementare la variabile lucchetto?

# Sincronizzazione tra processi – 4



Questa soluzione di uso di lock **non** funziona! Perché?

# Sincronizzazione tra processi – 5

- La soluzione appena vista è inadeguata
  - Ciascuno dei due processi può essere prerilasciato **dopo** aver letto la variabile *lock* ma **prima** di esser riuscito a modificarla
    - Questa situazione è ancora una **race condition** e può generare pesanti inconsistenze
  - Inoltre l'algoritmo mostrato richiede **attesa attiva (busy wait)** che causa spreco di tempo di CPU a scapito di altre attività a maggior valore aggiunto
- **Busy wait**: Attesa con verifica continua che una variabile cambi valore
  - Il processore rimane occupato con questa inutile verifica continua, invece che progredire con operazioni che possano far cambiare valore a lock

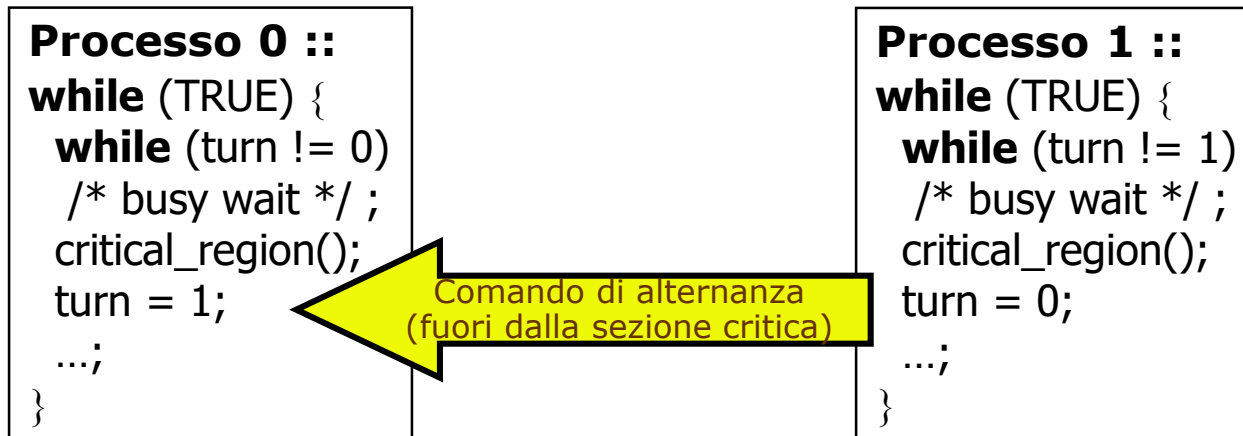
# Sincronizzazione ammissibile

- 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



- 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 (multicore) di oggi
  - Si applica solo a coppie di processi

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

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

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

# Soluzioni “esotiche” – 2b

- Immaginiamo ci siano due processi concorrenti che eseguono esattamente lo stesso codice, alternandosi sul processore
  - Ognuno dei due processi vedrà se stesso come il processo  $i$  e l'altro come il processo  $j$
  - Potendo essere  $i$  uguale solo a 0 o 1 ed essendo la controparte  $j = 1 - i$ , allora  $j$  è 0 quando  $i$  è 1 e viceversa
- Ognuno dei due processi lascerà passare avanti l'altro se vede `flag[j]` come 1 e `turn` come se stesso (ovvero come  $i$ , che per un processo corrisponde a 0 e per l'altro corrisponde a 1)
  - Significato semplificato: «anche l'altro processo ha richiesto l'accesso e io sono arrivato per secondo visto che `turn` ha il mio identificativo (ho quindi sovrascritto l'identificativo dell'altro)»

# 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`!

# Soluzioni “esotiche” – ESEMPIO

- Supponiamo vada in esecuzione prima il **processo 0 (P0)**
  - Esso vedrà  $i$  come 0 e  $j$  come 1
  - `flag[0]` sarà posta a 1
- Supponiamo che ora P0 sia prerilasciato prima di modificare la variabile `turn` e vada in esecuzione il **processo 1 (P1)**
  - Esso vedrà  $i$  come 1 e  $j$  come 0
  - `flag[1]` sarà posta a 1
  - `turn` sarà posta a 1 (ovvero se stesso)

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

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

# Soluzioni “esotiche” – ESEMPIO

- Supponiamo che ora anche P1 sia prerilasciato e torni in esecuzione P0 (continuando l’esecuzione precedente)
  - `turn` sarà posta a 0 (ovvero se stesso)
  - La condizione `while` è verificata e P0 si trova in **attesa attiva** (ripetendo `while` senza sosta fino al nuovo prerilascio)
- A un certo punto P0 sarà prerilasciato a favore di P1
  - P1 non soddisfa la condizione `while` e quindi prosegue nella **sezione critica gestita in mutua esclusione**
  - Se P1 dovesse essere prerilasciato, P0 si troverebbe ancora in **attesa attiva** fino a rilasciare il processore; potrà entrare in sezione critica solo quando P1 concluderà la sua sezione critica, chiamando `OUT` e ponendo quindi `flag[1] = FALSE`

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

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

# Sincronizzazione tra processi – 4

- 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
    - Sconsigliabile lasciare controllo interrupt a utenti (e se non li riattivano?)
    - Inutile con due processori

# Sincronizzazione tra processi – 5

- Tecniche complementari e/o alternative
  - 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 – 6

**enter\_region**

!! regione critica: la zona di programma che delimita  
!! l'accesso e l'uso di una variabile condivisa

**leave\_region**

---

**enter\_region:**

```
TSL R1, LOCK    !! Copia il valore di LOCK in R1
                !! e pone LOCK = 1 (bloccato)
                !! inoltre, blocca memory bus

CMP R1, 0       !! verifica se LOCK era 0 tramite R1
JNE enter_region !! attesa attiva se R1==1 (tornando a
                !! enter_region)

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 – 7

- Sia la soluzione di Peterson che quella TSL sono corrette ma hanno il difetto di eseguire **busy wait**
- Ulteriore possibile problema: **Inversion priority**
  - Consideriamo due processi:
    - H (ad alta priorità) e L (a bassa priorità)
  - Supponiamo che H prerilasci il processore per eseguire I/O
  - Supponiamo che H concluda le operazioni di I/O mentre L si trova nella sua sezione critica
  - H rimarrà bloccato in busy wating perché L non avrà più modo di concludere la sezione critica

# Il problema produttore- consumatore

```
#define N 100                                /* number of slots in the buffer */
int count = 0;                               /* number of items in the buffer */

void producer(void)
{
    int item;

    while (TRUE) {                            /* repeat forever */
        item = produce_item();                /* generate next item */
        if (count == N) sleep();              /* if buffer is full, go to sleep */
        insert_item(item);                    /* put item in buffer */
        count = count + 1;                    /* increment count of items in buffer */
        if (count == 1) wakeup(consumer);    /* was buffer empty? */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                            /* repeat forever */
        if (count == 0) sleep();              /* if buffer is empty, got to sleep */
        item = remove_item();                 /* take item out of buffer */
        count = count - 1;                    /* decrement count of items in buffer */
        if (count == N - 1) wakeup(producer); /* was buffer full? */
        consume_item(item);                   /* print item */
    }
}
```

# Sincronizzazione tra processi – 8

- Soluzione sleep & wakeup
- Wake up non vengono memorizzate
  - Se non c'è una sleep in attesa, le wakeup vengono perse
- **Rischio race condition su variabile count**
  - Il buffer è vuoto e il consumer ha appena letto che count = 0
  - Prima che il consumer istanzi la sleep, lo scheduler decide di fermare il consumer e di eseguire il producer
  - Il producer produce un elemento e imposta count = 1
  - Siccome count = 1 il producer emette wakeup (che nessuno ascolta)
  - A un certo punto lo scheduler deciderà di eseguire di nuovo il consumer il quale istanzia finalmente la sleep, ma siccome count è già stato riportato a 1 e la corrispondente wakeup è già stata emessa, il consumer non verrà più risvegliato

# Sincronizzazione tra processi – 9

- 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**, oppure **down**, decrementa il contatore se questo non è già 0, altrimenti accoda il chiamante
  - L'avviso di rilascio **V**, oppure **up** 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 – 10

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

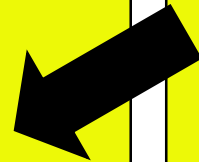
- Quale R tra tutte?  
**V(sem)** viene invocata per rilasciare la risorsa

# Sincronizzazione tra processi – 11

- 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 ::  
{// 1a parte del lavoro  
...  
V(sem); // rilascio di A  
// esecuzione indipendente  
...  
}
```



# Sincronizzazione tra processi – 12

- 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 – 13

- 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();
    }
}
```

# 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

# 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);
        V(&non-pieno);
        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*

Vedi esempio slide successiva



# Esempio 1 sbagliato

```
#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(&mutex);
        P(&non-pieno);
        inserisci(prod);
        V(&mutex);
        V(&non-vuoto);
    }
}

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

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

# 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 accedere alle 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;
    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*
    - Diverso dunque da sleep e wakeup visti in precedenza

# *Monitor – 8*

- Monitor e semafori **non sono utilizzabili** per realizzare scambio di informazione tra elaboratori
  - Perché?

# Message Passing

```
#define N 100                                /* number of slots in the buffer */

void producer(void)
{
    int item;
    message m;                               /* message buffer */

    while (TRUE) {
        item = produce_item();              /* generate something to put in buffer */
        receive(consumer, &m);              /* wait for an empty to arrive */
        build_message(&m, item);            /* construct a message to send */
        send(consumer, &m);                 /* send item to consumer */
    }
}

void consumer(void)
{
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m); /* send N empties */
    while (TRUE) {
        receive(producer, &m);              /* get message containing item */
        item = extract_item(&m);            /* extract item from message */
        send(producer, &m);                 /* send back empty reply */
        consume_item(item);                 /* do something with the item */
    }
}
```

# 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
  - **Barbiere che dorme** : prevenzione di *race condition*
  - **(Produttore e Consumatore)**
- Problemi pensati per rappresentare tipiche situazioni di rischio
  - Stallo con blocco (*deadlock*)
  - Stallo senza blocco (*starvation*)
  - Esecuzioni non predicibili (*race condition*)

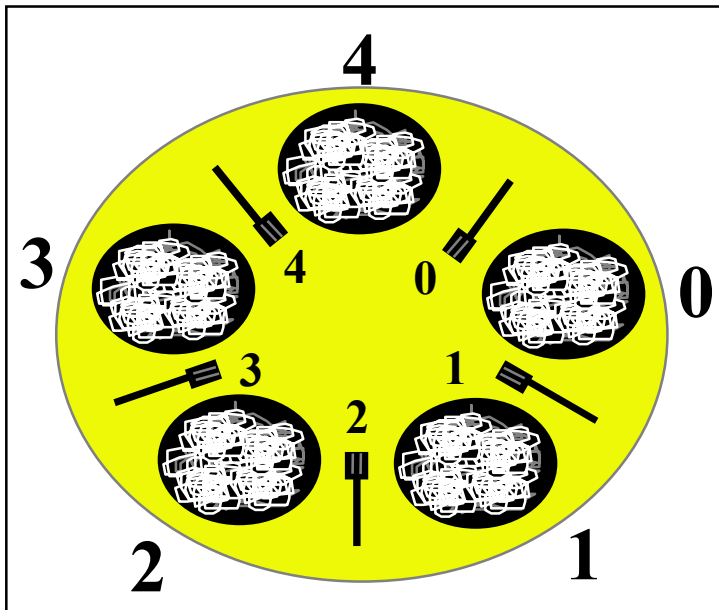
# Filosofi a cena – 1

- $N$  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



# Filosofi a cena – 3

## Soluzione B con stallo (*starvation*)

```
void filosofo (int i){
while (TRUE) {
    OK = FALSE;
    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]);
}
```

*Errato: f contiene semafori che sono strutture dati*

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

# 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

# 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

- 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

# 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

# Stallo: 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

- **Staticamente**

- Può essere un problema non risolvibile!