Comunicazioni interprocesso

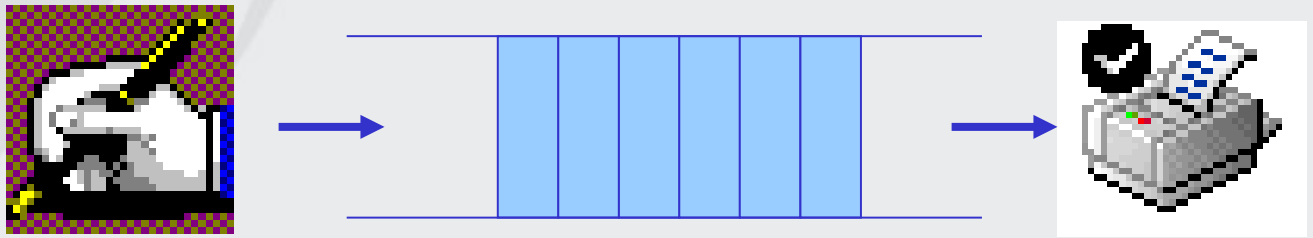
Background
La sezione critica
Sincronizzazione Hardware
Semafori
Regioni critiche
Monitor
Barriere
Problemi classici di sincronizzazione

Background

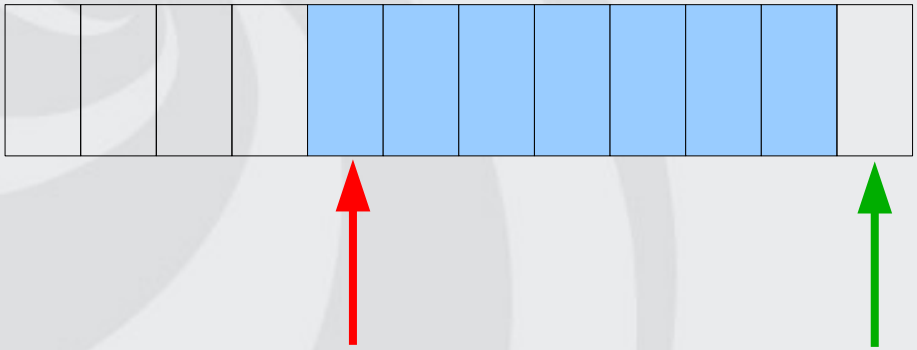
- L’accesso concorrente a dati condivisi può risultare in inconsistenza dei dati stessi
- Per garantire la consistenza sono necessari meccanismi per coordinare l’esecuzione dei processi cooperanti

Problema Produttore-consumatore

- Problema produttore-consumatore (bounded buffer problem)
  - due processi condividono un buffer comune di dimensione fissata
  - il processo ***Produttore*** mette le informazioni nel buffer
  - il processo ***Consumatore*** le preleva

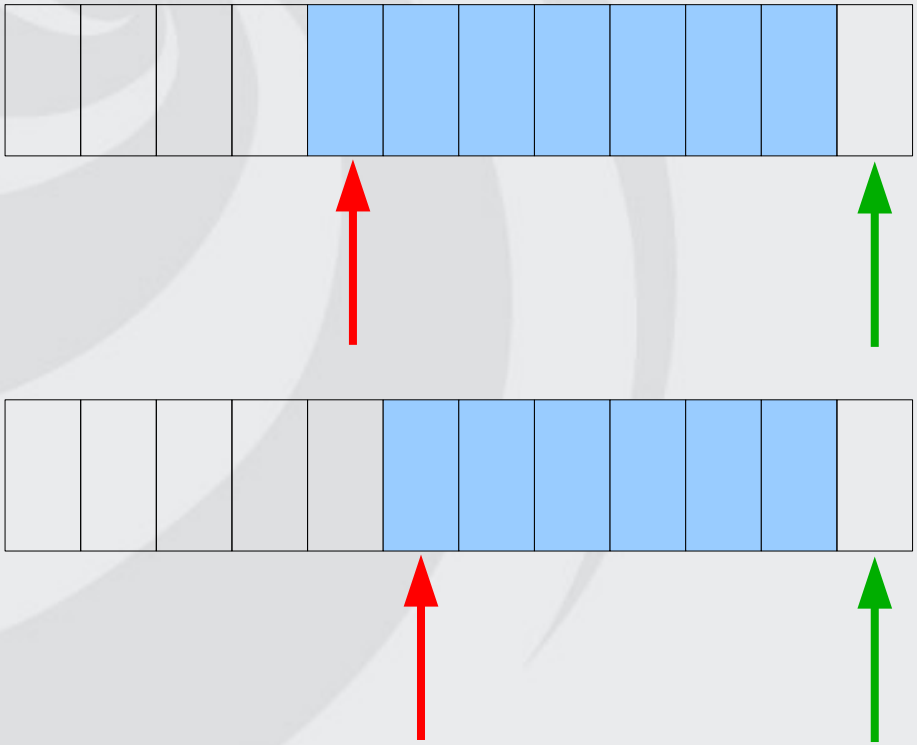


Bounded buffer



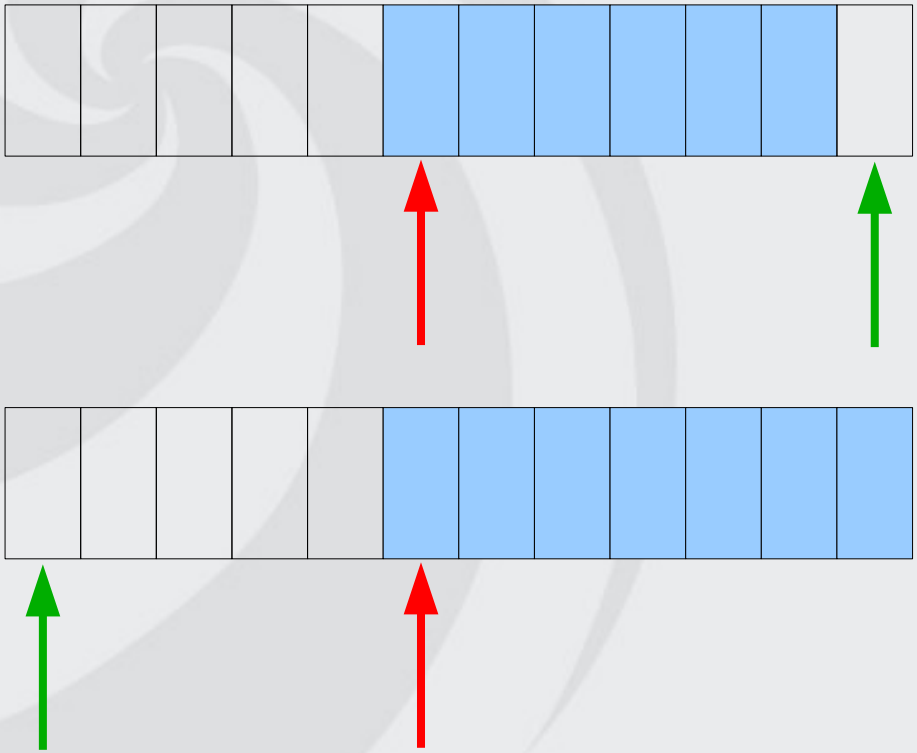
- **Proprietà**
  - Il buffer ha dimensione fissa
  - Si utilizzano 2 indici (puntatori)
  - L'inserimento avviene nell'elemento puntato dall'indice di ***Inserimento***
  - L'estrazione avviene nell'elemento puntato dall'indice di ***Estrazione***
  - L'aggiornamento avviene in un tempo costante (non dipende dal numero di dati memorizzati)

Estrazione



- Il puntatore avanza di una posizione

Inserimento



- Il puntatore avanza di una posizione
- Il buffer è considerato come un nastro chiuso

Bounded-Buffer - Produttore

```
class Produttore extends Thread {
  public void run() {
    int appenaProdotto = 0;
    int inserisci = 0;
    while(running) {
      appenaProdotto++;
      aspetta(milliProduttore);//produco un dato
      while(contatore == vettore.length) ;
      vettore[inserisci] = appenaProdotto;
      inserisci=(inserisci+1)%vettore.length;
      incrementaContatore(); // contatore++
    }
  }
}
```

Bounded-Buffer - Consumatore

```
class Consumatore extends Thread {

  public void run() {
    int preleva = 0;
    int daConsumare = 0;
    while(running || contatore>0) {
      while(contatore == 0) ;
      daConsumare = vettore[preleva];
      preleva = (preleva+1)%vettore.length;
      decrementaContatore(); // contatore--
      aspetta(milliConsumatore);//uso il dato
    }
  }
}
```

Race condition

- Le istruzioni

  contatore++; (incrementaContatore)
  contatore--; (decrementaContatore)

  - devono essere eseguite in modo atomico

- Una operazione atomica è una operazione che si completa senza interruzioni

Race condition

- In realtà “contatore++” si traduce in linguaggio macchina come:

  register1 = contatore
  register1 = register1 + 1
  contatore = register1

- In modo analogo “contatore--” :

  register2 = contatore
  register2 = register2 – 1
  contatore = register2

Race condition

- Se sia il produttore che il consumatore tentano di aggiornare il buffer concorrentemente le istruzioni assembler possono risultare intercalate

- La sequenza effettiva dipende dalla schedulazione dei due processi

Race condition

- Se contatore vale inizialmente 5. Una possibile sequenza è:

  producer: register1 = contatore (*register1 = 5*)
  producer: register1 = register1 + 1 (*register1 = 6*)
  consumer: register2 = contatore (*register2 = 5*)
  consumer: register2 = register2 – 1 (*register2 = 4*)
  producer: contatore = register1 (contatore = *6*)
  consumer: contatore = register2 (contatore = *4*)

- Il valore finale di contatore può essere 4 o 6 (invece di 5)

Programma completo

```
public class ProduttoreConsumatore {

 static public void main(String[] args) {
  if(args.length>0) milliProduttore = Integer.parseInt(args[0]);
  if(args.length>1) milliConsumatore = Integer.parseInt(args[1]);
  if(args.length>2) milliAspetta = Integer.parseInt(args[2]);
  if(args.length>3) milliMain = Integer.parseInt(args[3]);
  new ProduttoreConsumatore();
 }

 static public int milliProduttore = -80;
 static public int milliConsumatore = -100;
 static public int milliAspetta = 10;
 static public int milliMain = 5000;

 public ProduttoreConsumatore() {
  running = true;
  new Thread( new Consumatore() ).start();
  new Thread( new Produttore() ).start();
  aspetta(milliMain);
  running = false;
  System.out.println("maxContatore="+maxContatore);
 }

 volatile private boolean running;
 volatile private int contatore = 0, maxContatore = 0;
 private int[] vettore = new int[10];
```

```
 class Produttore extends Thread {

  public void run() {
   int appenaProdotto = 0;
   int inserisci = 0;
   while(running) {
    appenaProdotto++;
    while(contatore == vettore.length) ;
    vettore[inserisci] = appenaProdotto;
    inserisci = (inserisci+1) % vettore.length;
    incrementaContatore();
    aspetta(milliProduttore);
   }
   System.out.println(appenaProdotto);
  }
 }

 class Consumatore extends Thread {

  public void run() {
   int preleva = 0;
   int daConsumare = 0;
   while(running || contatore>0) {
    while(contatore == 0) ;
    daConsumare = vettore[preleva];
    preleva = (preleva+1) % vettore.length;
    decrementaContatore();
    aspetta(milliConsumatore);
   }
   System.out.println(daConsumare);
  }
 }
```

```
 private void incrementaContatore() {
  int tmp = contatore + 1;
  if(tmp>maxContatore) maxContatore = tmp;
  aspetta(milliAspetta);
  contatore = tmp;
 }

 private void decrementaContatore() {
  int tmp = contatore;
  aspetta(milliAspetta);
  contatore = tmp - 1;
 }

 private static void aspetta(int milli) {
  if(milli==0) return;
  try {
   if(milli<0) milli = (int) (-milli*Math.random());
   Thread.sleep(milli);
  } catch(InterruptedException e) {}
 }
}
```

Programma completo

- Il programma potrebbe non terminare mai
  - Quando?
  - Come si risolve il problema?

- È possibile accettare più produttori e/o più consumatori?

Comunicazione interprocesso

- **Race Condition**: situazione in cui due o più processi stanno leggendo o scrivendo qualche dato condiviso e il risultato finale dipende da chi esegue e quando
  - Per prevenire le **Race Condition** i processi concorrenti devono essere sincronizzati
- Il problema della sezione critica:
  - N processi competono per usare dati condivisi
  - Ogni processo ha un segmento di codice (**Sezioni critiche**) in cui i dati condivisi sono utilizzati
  - Problema: garantire che quando un processo sta eseguendo la sua sezione critica, nessun altro processo possa entrarvi

Sezioni critiche

- La scelta di appropriate operazioni primitive per ottenere la mutua esclusione è uno dei problemi fondamentali della progettazione di un sistema operativo.
- Sono state formalizzate alcune condizioni da rispettare per avere una buona soluzione:
  1. due processi non devono essere mai simultaneamente in una sezione critica
  2. nessuna assunzione a priori può essere fatta a proposito della velocità o del numero di CPU
  3. nessun processo in esecuzione fuori dalla sua sezione critica può bloccare altri processi
  4. nessun processo deve aspettare a oltranza per entrare nella sua sezione critica

Mutua esclusione

- Per ottenere la **Mutua esclusione** sono state proposte varie soluzioni: ognuna può essere distintiva per qualche implementazione di sistema operativo.
- Mentre un processo è occupato aggiornando la memoria condivisa nella sua regione critica, nessun altro processo entrerà nella sua regione critica a "causare guai".
  - Variabili di lock
  - Alternanza stretta
  - Soluzione di Peterson
  - Disabilitazione degli interrupt
  - Istruzione TSL

Schema generale

- prima di usare le variabili condivise (regione critica) ciascun processo chiama una

  enter_region

  con il proprio numero di processo (0 o 1) (può dover aspettare fino a che risulti sicuro entrare nella regione critica)
- dopo aver finito di lavorare con le variabili condivise, il processo chiama una

  leave_region

  per permettere a un altro processo di entrare

Schema generale

```
void processo(int proc) {
   while(true) {
      enter_region(proc);
      critical_region();
      leave_region(proc);
      non_critical_region();
   }
}
```

Variabili di lock

- Quando un processo vuole entrare nella sua regione critica esegue un test su una singola variabile lock, condivisa da tutti i processi e inizializzata a 0.
  - Se lock è 0 (falso), il processo la imposta ad 1 ed entra nella sua regione critica
  - Se lock è 1 (vero), il processo attende finché diventa 0
  - lock = 0 indica che nessun processo è nella sua regione critica
  - lock = 1 indica che un processo è nella sua regione critica

Variabili di lock

```
volatile boolean lock = false;
while(true) {
   while(lock);
   lock = true;
   critical_region();
   lock = false;
   non_critical_region();
}
```

Problema: due processi possono entrare contemporaneamente nella loro regione critica

- Il processo 1 fa il test di lock (è falso)
- Il processo 1 viene interrotto
- Il processo 2 fa il test di lock (è falso)
- Il processo 2 pone lock a vero

....

- Il processo 1 riprende

Alternanza stretta

```
int turn;

void processo(int proc) {
  while(true) {
    while(turn != proc); //BUSY WAITING
    critical_region();
    turn = 1-turn;
    non_critical_region();
  }
}
```

Alternanza stretta

- Il codice implementa la stretta alternanza nell'esecuzione della regione critica da parte di due processi
- **Attesa attiva (Busy waiting)**: test continuo su una variabile nell'attesa di un certo valore. Dovrebbe essere evitato perché spreca tempo di CPU
- Problemi:
  - Questa soluzione viola la terza condizione per la mutua esclusione, infatti un processo può essere bloccato da un altro che non sta eseguendo la sua regione critica
  - Se un processo si blocca allora anche l'altro è fermo
  - La soluzione può essere generalizzata a più processi, rendendo ancora più critico il problema precedente

Algoritmo 2

```
final static int N=2;          // numero di processi

boolean[] pronto = new boolean[N]; // inizializzato a false

void enter_region(int processo) { // processo 0 o 1
  int altro = 1 - processo;  // l'altro processo 1 o 0

  pronto[processo] = true;   // segnala che si è interessati
  while(pronto[altro]);
}


void leave_region(int processo) {
  pronto[processo] = false;
}
```

Problema: si può aver un ciclo infinito sul `while`

Soluzione di Peterson

```
final static int N=2;          // numero di processi
int turno;

boolean[] pronto = new boolean[N]; // inizializzato a false

void enter_region(int processo) { // processo 0 o 1
  int altro = 1 - processo;  // indice dell'altro processo 0 o 1

  pronto[processo] = true;   // segnala che si è interessati
  turno = altro;
  while(pronto[altro] && turno == altro);
}

void leave_region(int processo) {
  pronto[processo] = false;
}
```

Peterson: esempio

- Nessun processo è nella sezione critica
- Il processo 0 chiama la `enter_region`
- Indica il suo interesse mettendo a `true` il proprio elemento nell'array
- Mette turn a 1
- Se il processo 1 non è interessato, `enter_region` termina subito
- Se il processo 1 chiama la `enter_region`, rimane in attesa fino a che `interested[0]` è `false` (cioè quando il processo 0 chiama la `leave_region`)

Peterson: esempio

- Se entrambi i processi chiamano la `enter_region`, modificano il valore in turn (**ma uno viene perso**)
- se il processo 1 scrive per ultimo nella variabile turn, quando entrambi i processi arrivano al ciclo while, il processo 0 non esegue il ciclo ed entra immediatamente in sezione critica, al contrario il processo 1 esegue il ciclo rimanendo in attesa di entrare nella sezione critica

Algoritmo del fornaio

```
final static int N = Numero_di_processi;
boolean[] scelta = new boolean[N]; // inizializzato a false
int[] numero = new int[N]; // inizializzato a 0

void enter_region(int processo) {
  scelta[processo] = true;
  numero[processo] = max(numero) + 1;  //max massimo del vettore
  scelta[processo] = false;
  for(int j=0; j<N; j++) {
    while(scelta[j]);
    while(numero[j]!=0 && compare(j, processo));
  }
}

void leave_region(int processo) {
  numero[processo] = 0;
}
boolean compare(int i, int j) {
  return numero[i]<numero[j]  || (numero[i]==numero[j] && i<j);
}
```

Disabilitazione degli interrupt

- Ogni processo disabilita tutti gli interrupt dopo essere entrato nella sua regione critica e li riabilita prima di lasciarla
- Se gli interrupt sono disabilitati nessun interrupt del clock può essere attivato
- Questo metodo dà la possibilità di neutralizzare gli interrupt:
  - la disabilitazione delle procedure di interrupt qualche volta è utile se a carico del SO (non può essere lasciata all'utente), ma non è un buon approccio come meccanismo generale per risolvere il problema della **Mutua Esclusione**.

Disabilitazione degli interrupt

```
while(true) {
   /* disabilita gli interrupt */
   critical_region();
   /* abilita gli interrupt */
   non_critical_region();
}
```

Istruzione TSL (Test and Set Lock)

- Istruzione che copia il contenuto della parola di memoria in un registro e quindi memorizza un valore diverso da zero in quell'indirizzo di memoria. Le due operazioni sono indivisibili
- In modo equivalente si può usare una istruzione SWAP che scambia il valore fra due memorie in maniera atomica

Istruzione Test and Set Lock

```
enter_region:
   tsl register,lock  ; copia il lock nel
                      ; registro e pone lock a 1
   cmp register,#0    ; il lock era 0?
   jne enter_region   ; se falso cicla
   ret                ; ritorna al chiamante,
                      ; entra in sezione critica

leave_region:
   move lock,#0       ; metti lock a 0
   ret                ; ritorna al chiamante
```

Sleep and Wakeup

- La soluzione di Peterson e l'utilizzo dell'istruzione TLS sono corrette ma comportano l'attesa attiva (busy waiting) che implica:
  - spreco di tempo di CPU
  - problema dell'inversione di priorità (attesa di un processo a bassa priorità)
- **Soluzione:** uso di system call
  - le system call `sleep` e `wakeup` bloccano i processi invece di sprecare tempo di CPU quando non possono eseguire la loro regione critica
  - `sleep` - system call che blocca il chiamante finché un altro processo lo "risveglia"
  - `wakeup` - system call che ha come unico parametro il processo da "risvegliare"

Produttore-consumatore

- Questo problema costituisce un esempio dell'utilizzo delle primitive *Sleep* e *Wakeup*:
  - se il produttore vuole mettere un nuovo elemento nel buffer pieno, viene messo in attesa per essere risvegliato quando il consumatore ha prelevato qualche elemento
  - se il consumatore vuole rimuovere un elemento dal buffer vuoto, viene messo in attesa per essere risvegliato quando il produttore ha messo qualcosa nel buffer

Produttore-consumatore

```
static final int N=100;          /* numero di posizioni nel buffer */
int count = 0;                   /* numero di elementi nel buffer */

void producer() {
   while(true) {
      produce_item();
      if (count==N)          /* se il buffer è pieno sospendi */
         sleep();
      enter_item();          /* mette un nuovo elemento nel buffer */
      count++;               /* incrementa il numero degli elementi*/
      if(count == 1)         /* il buffer era vuoto? */
         wakeup(consumer);
   }
}
```

Produttore-consumatore

```
void consumer() {
   while(true) {
      if (count==0)          /* se il buffer è vuoto sospendi */
         sleep();
      remove_item();         /* estrae un elemento nel buffer */
      count--;          /* decrementa il numero degli elementi */
      if(count == N-1)       /* il buffer era pieno? */
        wakeup(producer);
      consume_item();
   }
}
```

Produttore-consumatore

- Si ha una corsa critica su count
  - Il consumatore legge count (0) e viene sospeso dal SO
  - Il produttore immette un dato e lancia un wakeup
  - Riprende il consumatore fa il test su 0 e si sospende (il segnale di risveglio viene quindi perso)
  - Quando il buffer si sarà riempito anche il produttore si sospende
  - Entrambi i processi sono sospesi per sempre

Semafori

- Dijkstra introduce una nuova proposta per la soluzione dei problemi relativi alla sincronizzazione tra processi e alle condizioni di gara.

Semaforo

- variabile intera per il conteggio del numero di wakeup pendenti che può avere valore:
  - 0 - se non è stato salvato alcun wakeup;
  - valore positivo - se ci sono wakeup pendenti.
- down (P) e up (V)
- generalizzazioni di sleep e wakeup
- le operazioni relative a down e up devono essere **Operazioni Atomiche**, vengono eseguite come un'azione singola ed indivisibile

Semafori

```
public interface Semaforo {

    public void up();
    public void down();

}
```

Semafori

```
public static Semaforo getDefaultSemaforo(int n)
   {
   // creo un oggetto che implementa Semaforo
   // inizializzato a n
   Semaforo nuovo = new MioSemaforo(n);
   return nuovo
}
```

Supponiamo di avere a disposizione un metodo statico in grado di creare un semaforo

```
int s;
public void down() {
   while(s<=0);
   s--;
}
public void up() {
   s++;
}
```

Struttura di una possibile implementazione
NB: le due operazioni devono essere atomiche (uso di synchronized)

La classe Semaphore

Class Semaphore

java.lang.Object
    java.util.concurrent.Semaphore

```
public class Semaphore
extends java.lang.Object
```

A counting semaphore. Conceptually, a semaphore maintains a set of permits. Each acquire() blocks if necessary until a permit is available, and then takes it. Each release() adds a permit, potentially releasing a blocking acquirer. However, no actual permit objects are used; the Semaphore just keeps a count of the number available and acts accordingly.

Constructors

Constructor	Description
`Semaphore(int permits)`	Creates a Semaphore with the given number of permits and nonfair fairness setting.

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method	Description
`void`	`acquire()`	Acquires a permit from this semaphore, blocking until one is available, or the thread is interrupted.
`void`	`release()`	Releases a permit, returning it to the semaphore.

Semafori

- Soluzione del problema produttore-consumatore con l'utilizzo dei semafori
- Tre semafori utilizzati per garantire:
  - mutua esclusione
  - sincronizzazione

Semafori

```
static final int N=100;     // numero di posizioni nel buffer */

Semaforo mutex = getDefaultSemaforo(1); // controlla l'accesso in sezione critica
Semaforo empty = getDefaultSemaforo(N); // conta il numero di posizioni vuote nel buffer
Semaforo full = getDefaultSemaforo(0);  // conta il numero di elementi nel buffer

void producer() {
  while(true) {
    int item = produce_item();
    empty.down();        // decrementa il numero delle posizioni vuote
    mutex.down();        // entra in sezione critica
    enter_item(item);    // mette un nuovo elemento nel buffer
    mutex.up();          // abbandona la sezione critica
    full.up();           // incrementa il numero delle posizioni piene
  }
}
```

Semafori

```
void consumer() {
  int item;

  while(true) {
    full.down();          // decrementa il numero delle posizioni piene
    mutex.down();         // entra in sezione critica
    item=remove_item();   // estrae un elemento dal buffer
    mutex.up();           // abbandona la sezione critica
    empty.up();           // incrementa il numero delle posizioni vuote
    consume_item(item);
  }
}
```

Monitor

- La realizzazione di programmi con l'utilizzo di semafori può portare facilmente ad errori subdoli quali il

  Deadlock

  situazione in cui più processi sono definitivamente bloccati e non può essere eseguito alcun lavoro
- Una soluzione che consente la realizzazione di programmi corretti in modo semplice è l'utilizzo di una primitiva di sincronizzazione di più alto livello denominata
- **Monitor**
  - una collezione di procedure, variabili e strutture dati raggruppate in un pacchetto di tipo speciale
  - I processi possono richiamare le procedure contenute in un monitor ma non possono accedere direttamente alle sue strutture dati interne da procedure dichiarate esternamente al monitor

Monitor

- I monitor sono costrutti di linguaggio di programmazione quindi il compilatore gestisce diversamente le chiamate alle procedure del monitor dalle chiamate ad altre procedure

```
monitor example
   integer i;
   condition c;

   procedure producer(x);
   .
   .
   end;

   procedure consumer(x);
   .
   .
   end;
end monitor;
```

Monitor - Proprietà

- I monitor sono utili per ottenere la mutua esclusione infatti in ogni istante solo un processo può essere attivo nel monitor
- Affinché due processi non entrino contemporaneamente nella loro regione critica è sufficiente inserire le loro sezioni critiche nelle procedure del monitor infatti la realizzazione della mutua esclusione nei monitor è demandata al compilatore e non al programmatore

minore possibilità di errore

Monitor - Proprietà

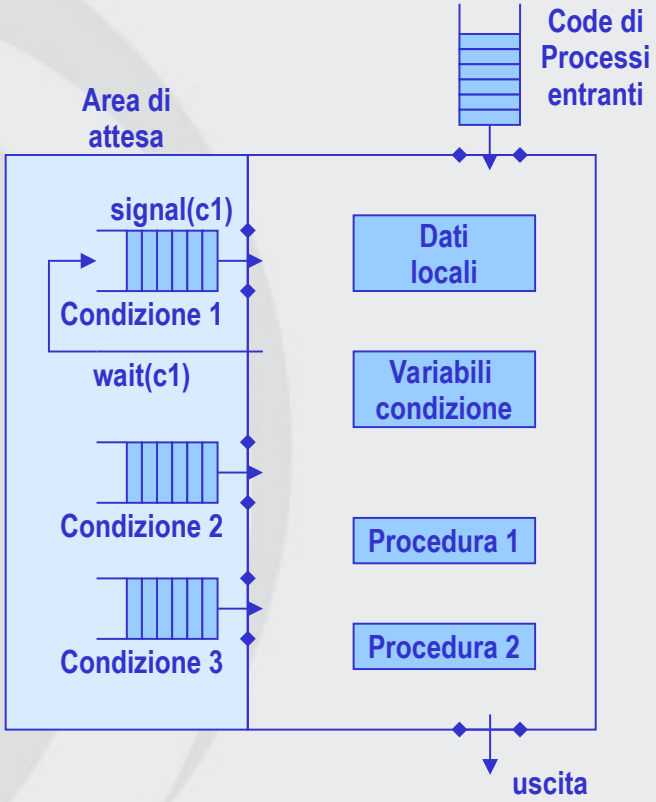
- **Problema:** come bloccare i processi quando non possono procedere?
- **Soluzione:** variabili condizione con due operazioni su di esse (`wait` e `signal`):
- `wait` - operazione su una variabile condizione che blocca il processo chiamante
- `signal` - operazione su una variabile condizione che risveglia il relativo processo
- Una `signal` deve essere l'ultima operazione in una procedura di un monitor per evitare che due processi siano attivi contemporaneamente nel monitor (cioè nella sezione critica)

Monitor - Proprietà

```
monitor ProducerConsumer
  integer count;
  condition full, empty;
  procedure enter;
    if count=N then wait(full);
    enter_item;
    count := count + 1;
    if count=1 then
       signal(empty);
  end;
  procedure remove;
    if count=0 then wait(empty);
    remove_item;
    count := count - 1;
    if count=N-1 then
       signal(full);
  end;
  count := 0;
end monitor;
```

```
procedure producer;
 begin
   while true do begin
     produce_item;
     ProducerConsumer.enter;
   end
 end;

 procedure consumer;
 begin
   while true do begin
     ProducerConsumer.remove;
     consume_item;
   end
 end;

end;
```

Monitor


Code di
Processi
entranti
Area di
attesa
signal(c1)
Condizione 1
wait(c1)
Condizione 2
Condizione 3
Dati
locali
Variabili
condizione
Procedura 1
Procedura 2
uscita

Problemi relativi a monitor e semafori

- Molti linguaggi (ad esempio C e Pascal) non prevedono i monitor e i semafori
- Le primitive utilizzate per l'implementazione non sono applicabili in un ambiente distribuito, dove più CPU, ognuna con la relativa memoria privata, sono connesse da una rete locale

Soluzione:

Passaggio di messaggi

Passaggio di messaggi

- questo metodo di comunicazione tra processi utilizza due primitive

  send e receive

- system call che possono essere inserite in procedure di libreria come
  - **send**(destinazione, &messaggio)
  - **receive**(sorgente, &messaggio)

Esempio linux

```
   // include … per i dettagli: man msgsnd
int msgsnd(int msqid, void *msg, int msgsz, int msgflg);
int msgrcv(int msqid, void *msg, int msgsz, long msgtype,
   int msgflg);
int msgget(key_t key, int msgflg); // restituisce un msqid

int main(int argc, char* argv[])
{
  int msqid = msgget(IPC_PRIVATE, 0777);
  if(fork()) {
    long msg[] = { 0, 0};
    msgrcv(msqid, msg, sizeof(long), 0, 0);
    printf("msg: %ld\n", msg[1]);
    msgctl(msqid, IPC_RMID, 0);   // rimuove la coda
  } else {
    long msg[] = { 1 /* msgtype */, 39 /* messaggio */};
    msgsnd(msqid, msg, sizeof(long), 0);
  }
  return 0;
}
```

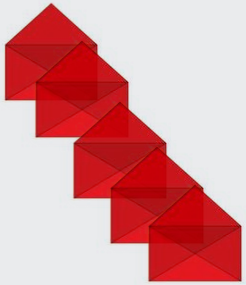
Produttore-consumatore

Con il passaggio di messaggi sono possibili molte varianti per l'indirizzamento dei messaggi:

- assegnare ad ogni processo un indirizzo unico ed indirizzare i messaggi ai processi
- **Mailbox** - struttura dati speciale che consente di bufferizzare un numero di messaggi definito alla creazione della mailbox. I parametri di indirizzamento nelle chiamate s*end e receive* sono le mailbox e non i processi
- I due estremi con l'utilizzo delle mailbox sono rispettivamente:
  - produttore e consumatore possono creare mailbox in grado di contenere gli N messaggi
  - **Rendezvous** - nessuna bufferizzazione - il produttore ed il consumatore lavorano forzatamente allo stesso passo

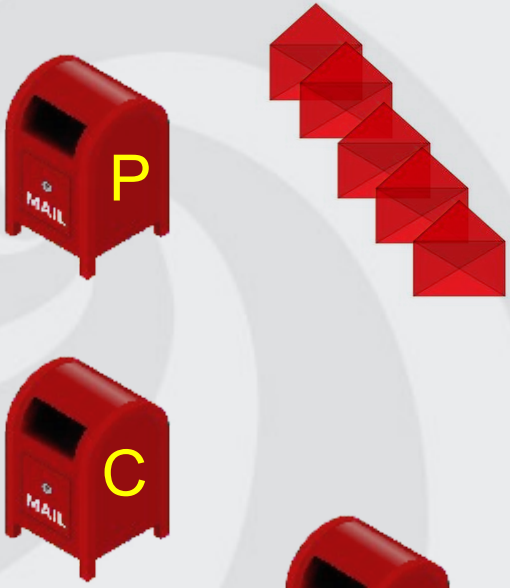
Produttore-consumatore

Il sistema è inizializzato con l'invio di N messaggi vuoti a P

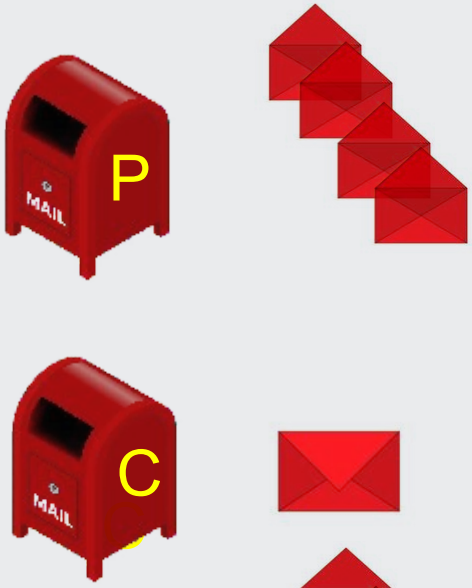


C è in attesa di messaggi
P inizia a produrre

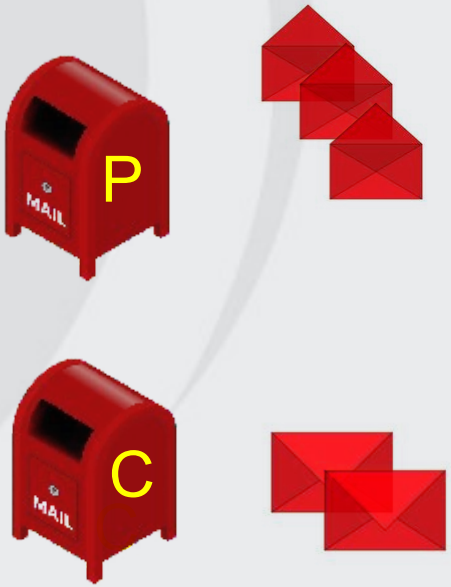
Produttore-consumatore



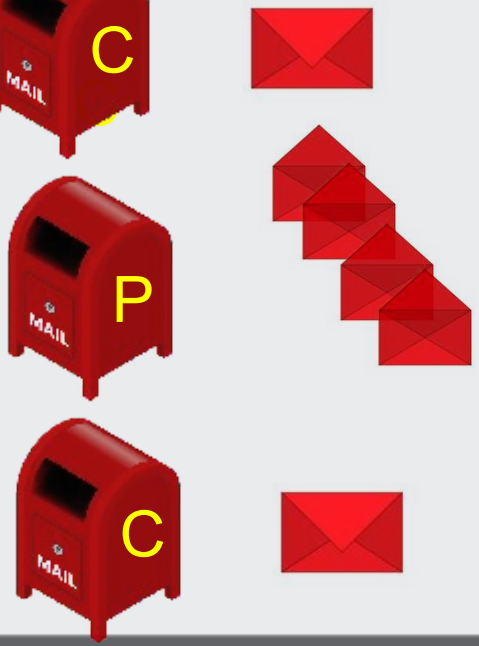
P produce un dato
Viene spedito a C



P produce un nuovo dato che viene spedito a C



C consuma un dato e invia a P un messaggio vuoto



Produttore-consumatore

...
P rimane in attesa non avendo messaggi disponibili

Soluzione – classe Mailbox

```
public abstract class Mailbox {

  public abstract void send(Message m);  // da definire

  public abstract Message receive(); // da definire

  public abstract boolean isEmpty(); // può sempre servire!

}
class DefaultMailbox extends Mailbox {
  public DefaultMailbox() {} // crea una mailbox vuota

  // inizializza una mailbox con N messaggi vuoti
  public DefaultMailbox(int N) {
    for(int i=0; i<N; i++) {
      this.send(new Message());
    }
  }
  // definire poi i metodi send receive e isEmpty
}
```

Soluzione – classe Messaggio

```
// esempio di possibile implementazione di Messaggio

public class Messaggio {

  private Object m;      // il messaggio contenuto

  public Messaggio() {} // crea un messaggio vuoto

  public Messaggio(Object msg) {
    setMessaggio(msg);  // crea ed inizializza un messaggio
  }

  public void setMessaggio(Object msg) {
    this.m = msg; // aggiorna il contenuto di un messaggio
  }

  public Object getMessaggio() {
    return m; // restituisce il contenuto di un messaggio
  }
}
```

Soluzione - processi

```
Mailbox consumerMailbox = new DefaultMailbox(); // classe derivata
Mailbox producerMailbox = new DefaultMailbox(100); // gestisco 100 dati

void procedura_producer() {
  while(true) {
    int item = produce_item();
    Messaggio m = producerMailbox.receive();   // attendi un messaggio
    build_message(m, item); // costruisce il massagio da inviare
    consumerMailbox.send(m); // invia il nuovo messagio al consumatore
  }
}

void procedura_consumer() {
  while(true) {
    Messaggio m = consumerMailbox.receive();  // riceve un messaggio
    int item = extract_item(m);        // estrae i dati dal messaggio
    producerMailbox.send(m)           // restituisce un messagio
    consume_item(item);
  }
}
```

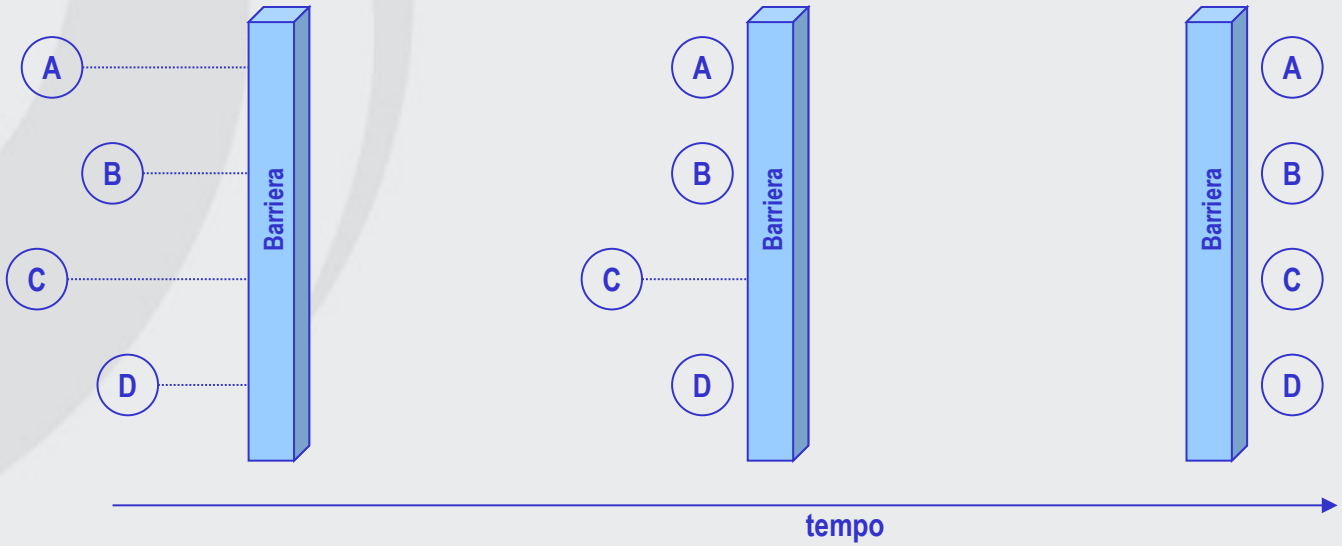
Soluzione alternativa

send e receive relativi al processo

```
void procedura_producer(Produttore producer, Consumatore consumer) {
  while(true) {
    int item = produce_item();
    Messaggio m = receive(producer);    // attende un messaggio
    build_message(m, item); // costruisce il messaggio da inviare
    send(consumer, m);      // invia il messaggio al consumatore
  }
}

void procedura_consumer(Produttore producer, Consumatore consumer) {
  while(true) {
    Messaggio m = receive(consumer);  // riceve un messaggio
    int item = extract_item(m);       // estrae i dati dal messaggio
    send(producer, m);      // restituisce al produttore un messagio
    consume_item(item);
  }
}
void init(Produttore producer, int N) { // inizializza la coda
  for(int i=0; i<N; i++) send(producer, new Messaggio());
}
```

Barriere

- Uso di una barriera
  - I processi si avvicinano ad una barriera
  - Tutti i processi si bloccano alla barriera
  - Solo quando l'ultimo processo arriva, tutti possono attraversare la barriera



Barriere (CyclicBarrier)

```
import java.util.concurrent.CyclicBarrier;
public class TestCyclicBarrier {
   static final int N = 10;
   static CyclicBarrier barrier = new CyclicBarrier(N+1);

   public static void main(String[] args) throws Exception {
      for(int i=0; i<N; i++) {
         (new Thread() {
            public void run() {
               try {
                  barrier.await();
               } catch(Exception e) {}
               System.out.println(getName());
            }
         }).start();
      }
      barrier.await();
      System.out.println(Thread.currentThread().getName());
   }
}
```

Barriere (CountDownLatch)

```
import java.util.concurrent.CountDownLatch;

public class TestCountDownLatch {

    static final int N = 10;
    static CountDownLatch barrier = new CountDownLatch(N);

    public static void main(String[] args) throws Exception {
        for(int i=0; i<N; i++) {
            (new Thread() {
                public void run() {
                    System.out.println(getName());
                    barrier.countDown();
                }
            }).start();
        }
        barrier.await();
        System.out.println(Thread.currentThread().getName());
    }
}
```

Equivalenze fra primitive

- Sono state proposte diverse primitive per la comunicazione tra processi utilizzando una qualsiasi di queste primitive è possibile implementare le altre
- Si può dimostrare l'equivalenza tra:
  - Semafori
  - Monitor
  - Messaggi

Problemi classici di IPC

- IPC (comunicazione fra processi)
  - Il problema dei *"Filosofi a cena"*
  - Il problema dei *"Lettori e Scrittori"*
  - Il problema del *"Barbiere dormiente"*

Filosofi a cena

(problema di sincronizzazione - Dijkstra, 1965)

- N filosofi stanno seduti intorno a un tavolo. Ciascun filosofo ha un piatto di spaghetti: per mangiarli ogni filosofo ha bisogno di usare due forchette; fra ognuno dei piatti vi è una forchetta.
- La vita dei filosofi è fatta di periodi alternati in cui essi ***pensano o mangiano***: quando un filosofo comincia ad avere fame, cerca di prendere possesso della forchetta di sinistra e di quella di destra, una alla volta in ordine arbitrario.
- Quando ha a disposizione entrambe le forchette incomincia a mangiare, quando si è sfamato depone le forchette e riprende a pensare.

Filosofi a cena

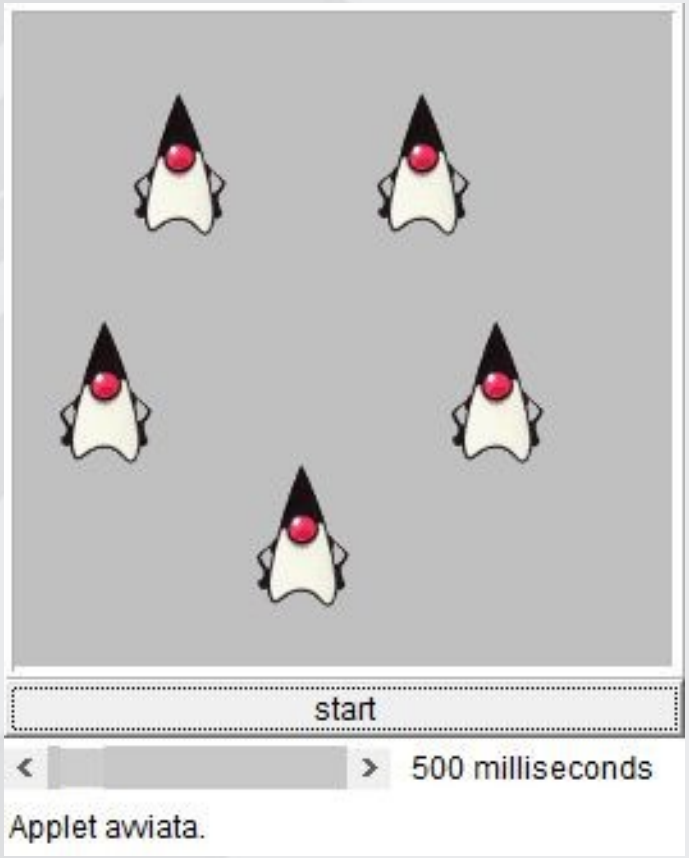
- **Problema:** come scrivere un programma per ciascuno dei filosofi che non si fermi mai?

Prima soluzione

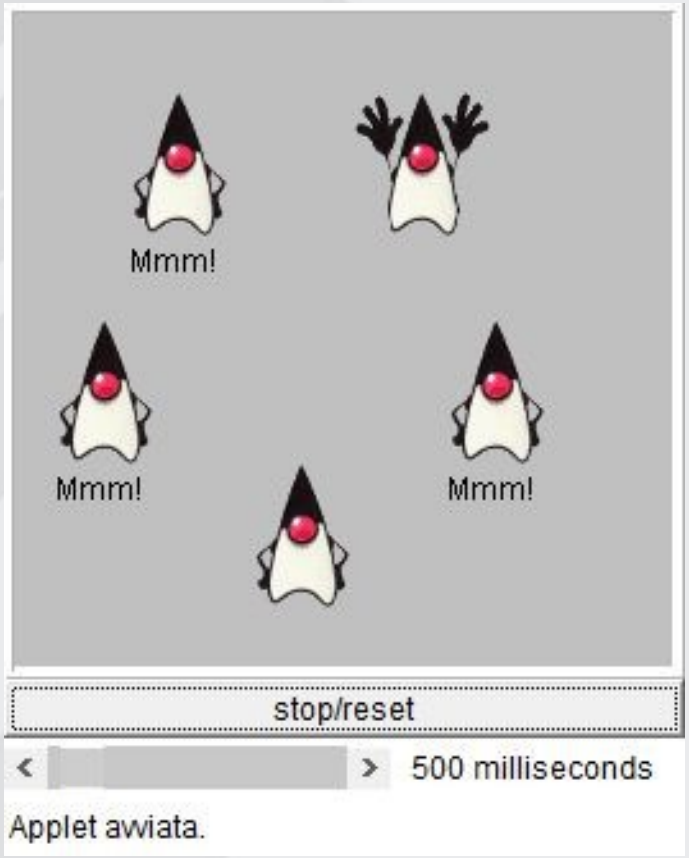
```
static final int N=5;    /* numero di filosofi */

void philosopher(int i) {
  while(true) {
    think();             /* il filosofo sta pensando */
    // take_fork aspetta fino a che la forchetta specificata
    // è disponibile e ne prende possesso
    take_fork(i);        /* prende la forchetta sinistra */
    take_fork((i+1)%N); /* prende la forchetta destra */
    eat(i);
    put_fork(i);         /* ripone le forchette */
    put_fork((i+1)%N);   /* ripone le forchette */
  }
}
```

Stallo

```
% appletviewer filosofi.html
```

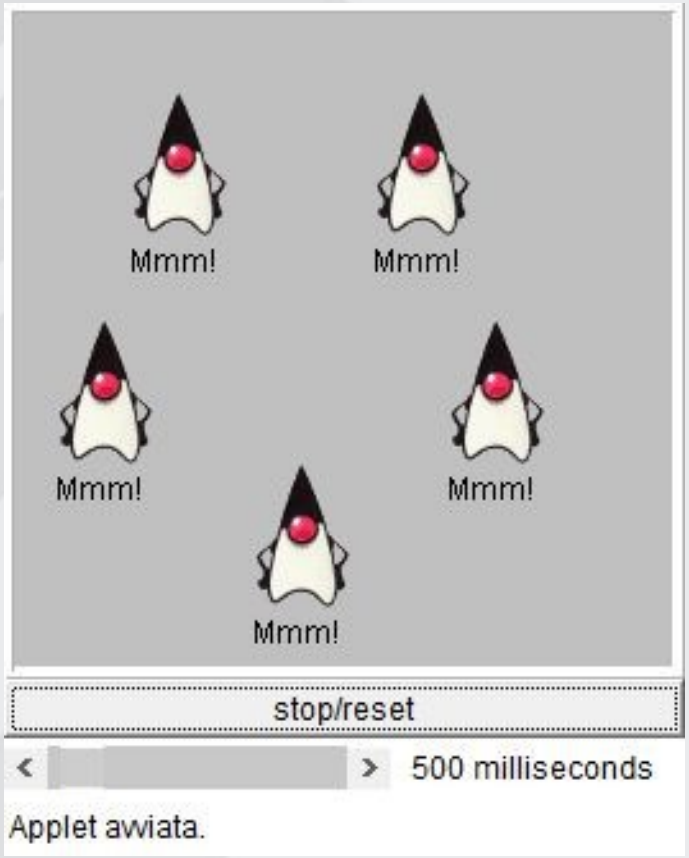


Stallo



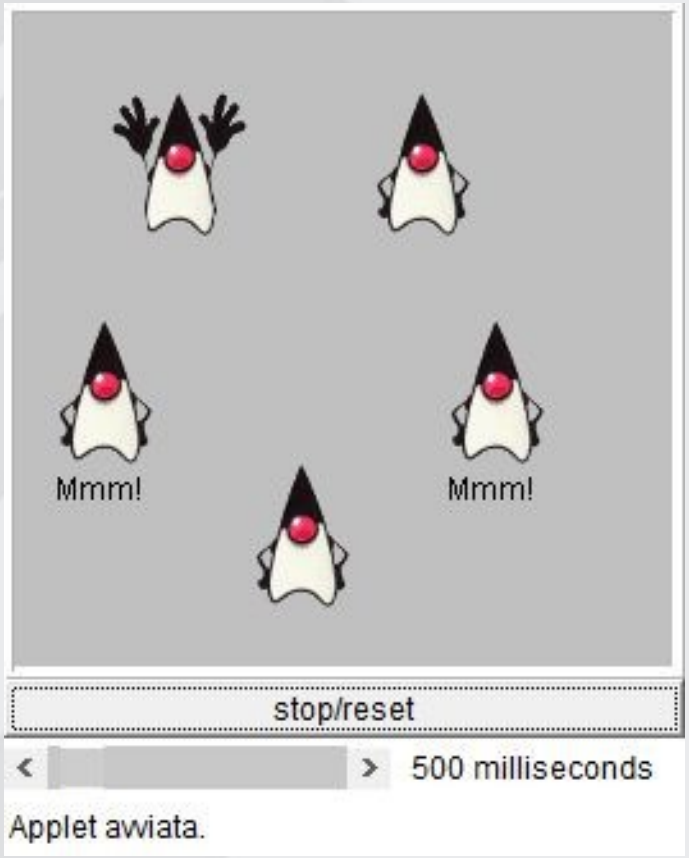
- Esiste la possibilità di stallo: se tutti i filosofi prendono la forchetta di sinistra nello stesso istante, nessuno di loro sarà in grado di prendere quella di destra

Stallo



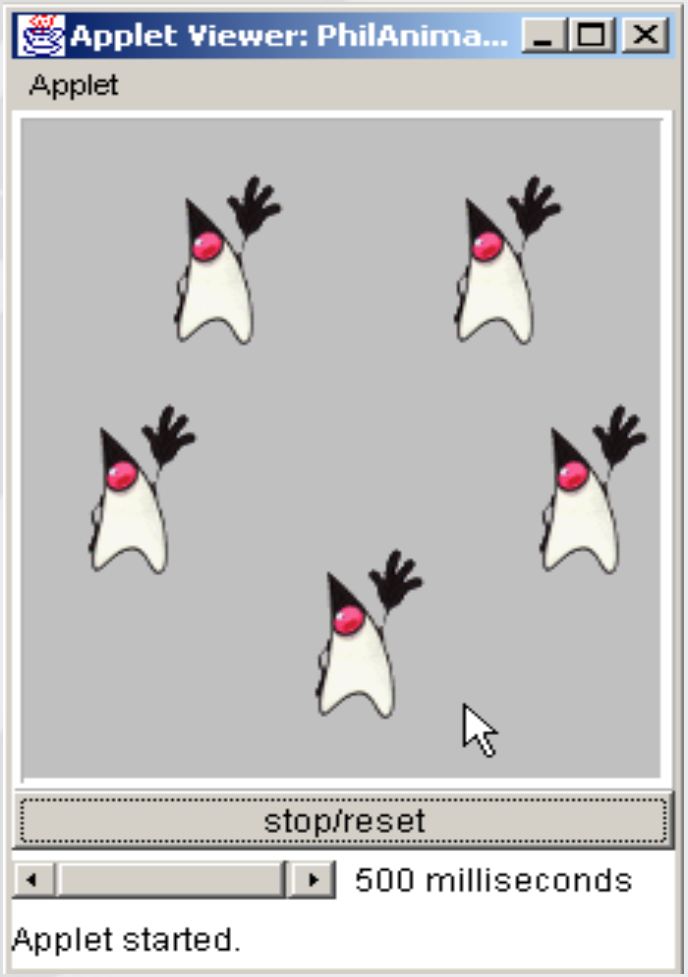
- Esiste la possibilità di stallo: se tutti i filosofi prendono la forchetta di sinistra nello stesso istante, nessuno di loro sarà in grado di prendere quella di destra

Stallo



- Esiste la possibilità di stallo: se tutti i filosofi prendono la forchetta di sinistra nello stesso istante, nessuno di loro sarà in grado di prendere quella di destra

Stallo



- Esiste la possibilità di stallo: se tutti i filosofi prendono la forchetta di sinistra nello stesso istante, nessuno di loro sarà in grado di prendere quella di destra

```
public void run() {
  int grabDelay;
  while (stopRequested == false) {
    try {
      grabDelay = parent.controller.grabDelaySlider.getValue() * 100;
      sleep((int)(Math.random() * grabDelay));
      rightStick.grab(this);
      parent.repaintPhil(position);

      grabDelay = parent.controller.grabDelaySlider.getValue() * 100;
      sleep((int)(Math.random() * grabDelay));
      leftStick.grab(this);
      parent.repaintPhil(position);

      grabDelay = parent.controller.grabDelaySlider.getValue() * 100;
      sleep((int)(Math.random() * grabDelay));
      eat();

      grabDelay = parent.controller.grabDelaySlider.getValue() * 100;
      sleep((int)(Math.random() * grabDelay * 4));
      sated = false;
      parent.repaintPhil(position);
    } catch (java.lang.InterruptedException e) {
    }
  }
}
```

Secondo approccio

- Il filosofo prende la prima forchetta
- se non è disponibile la seconda forchetta, il filosofo rimette la prima sul tavolo e aspetta un certo intervallo di tempo
- se tutti i filosofi iniziano contemporaneamente con una forchetta, e l'altra non è disponibile, la ripongono, aspettano, riprovano da capo ...

Starvation

- situazione in cui ogni programma è in esecuzione senza che ottenga effettivamente alcun progresso

Terza soluzione (costosa)

- Si fa uso di un semaforo binario:
  - prima di prendere possesso delle forchette, un filosofo esegue una `down` sulla variabile mutex
  - dopo aver deposto le forchette, fa una `up` sulla variabile mutex
- la soluzione è corretta, ma costosa in termini di prestazioni: solo un filosofo alla volta può mangiare
  Con 5 forchette due filosofi potrebbero mangiare contemporaneamente
- La soluzione migliore è la seguente: consentire il massimo grado di parallelismo per un numero arbitrario di filosofi (impiego di un vettore di stati/semafori per ogni filosofo: un filosofo inizia a mangiare solo se nessuno dei vicini sta mangiando

Soluzione

```
static final int N=5;           // numero di filosofi
static final int THINKING=0;    // il filosofo sta pensando
static final int HUNGRY=1;      // il filosofo cerca una forchetta
static final int EATING=2;      // il filosofo sta mangiando

int[] state = new int[N];       // vettore degli stati
Semaforo mutex = getDefaultSemaforo(1);     // mutua esclusione
                                // per le sezioni critiche
Semaforo[] s=new Semaforo[N]; // un semaforo per ogni filosofo
                                // occorre inizializzarli a 0

void philosopher(int i) { // indice del filosofo
  while(true) {
    think();
    take_forks(i);
    eat();
    put_forks(i);
  }
}
```

Soluzione

```
void take_forks(int i) {   // indice del filosofo
  mutex.down();            // entra in sezione critica
  state[i] = HUNGRY;       // registra che il filosofo è affamato
  test(i);                 // cerca di ottenere le due forchette
  mutex.up();              // esce dalla sezione critica
  s[i].down();             // si blocca se non ha le forchette
}

void put_forks(int i) {    // indice del filosofo
  mutex.down();            // entra in sezione critica
  state[i] = THINKING;     // il filosofo ha finito di mangiare
  test(left(i));           // controlla se il filosofo di sinistra può mangiare
  test(right(i));          // controlla se il filosofo di destra può mangiare
  mutex.up();              // esce dalla sezione critica
}
```

Soluzione

```
void test(int i) {
  if(state[i] == HUNGRY &&
    state[right(i)] != EATING &&
    state[left(i)] != EATING) {

    state[i] = EATING;
    s[i].up();  // sblocco il filosofo
  }
}

static int left(int i) {
  return ((i-1+N)%N); /* indice vicino di sinistra */
}

static int right(int i) {
  return (i+1)%N; /* indice vicino di destra */
}
```

Lettori e scrittori

Il problema dei "Lettori e Scrittori"
(modella l'accesso a un database)

- **Esempio: un database molto grande con tanti processi in competizione per leggere o scrivere**
- **Molti possono leggere contemporaneamente, ma se un processo sta scrivendo (modifica) nessuno può leggere**

Soluzione

```
Semaforo mutex = getDefaultSemaforo(1);  // controlla l'accesso a rc
Semaforo db = getDefaultSemaforo(1);     // controlla l'accesso al data base
int rc = 0;           // # di processi in lettura

void reader() {
  while(true) {
    mutex.down();              // ottiene l'accesso esclusivo a rc
    rc = rc + 1;               // incrementa il numero di lettori
    if(rc==1) db.down();       // se è il primo lettore ...
    mutex.up();                // rilascia l'accesso esclusivo a rc
    read_data_base();          // accedi ai dati
    mutex.down();              // ottiene l'accesso esclusivo a rc
    rc = rc - 1;               // decrementa il numero di lettori
    if(rc==0) db.up();         // se è l'ultimo lettore ...
    mutex.up();                // rilascia l'accesso esclusivo a rc
    use_data_read();           // sezione non critica
  }
}
```

Soluzione

```
void writer() {
  while(true) {
    think_up_data();          // sezione non critica
    db.down();                // ottieni l'accesso esclusivo
    write_data_base();        // aggiorna i dati
    db.up();                  // rilascia l'accesso esclusivo
  }
}
```

Il primo lettore che ha l'accesso esegue una `down` sul semaforo db; i lettori successivi incrementano (entrando) e decrementano (uscendo) un contatore

L'ultimo a uscire esegue una `up` sul semaforo lasciando via libera ad uno scrittore

Si è data priorità ai lettori, ma ci possono essere soluzioni diverse