Sistemi Operativi per LT Informatica
A.A. 2017-2018

Semafori

Docente: Salvatore Sorce

Semafori - Introduzione

- **Nei prossimi lucidi**
  - vedremo alcuni meccanismi dei S.O. e dei linguaggi per facilitare la scrittura di programmi concorrenti
- **Semafori**
  - il nome indica chiaramente che si tratta di un paradigma per la sincronizzazione (così come i semafori stradali sincronizzano l'occupazione di un incrocio)
- **Un po' di storia**
  - Dijkstra, 1965: Cooperating Sequential Processes
  - Obiettivo:
    - descrivere un S.O. come una collezione di processi sequenziali che cooperano
    - per facilitare questa cooperazione, era necessario un meccanismo di sincronizzazione facile da usare e "pronto all'uso"

Semafori - Definizione

- **Principio base**
  - due o più processi possono cooperare attraverso semplici segnali, in modo tale che un processo possa essere bloccato in specifici punti del suo programma finché non riceve un segnale da un altro processo
- **Definizione**
  - E' un tipo di dato astratto per il quale sono definite due operazioni:
  - ***V*** *(dall'olandese verhogen):*
    viene invocata per inviare un segnale, quale il verificarsi di un evento o il rilascio di una risorsa
  - ***P*** *(dall'olandese proberen):*
    viene invocata per attendere il segnale (ovvero, per attendere un evento o il rilascio di una risorsa)

Semafori - Descrizione informale

- **Descrizione informale:**
  - un semaforo può essere visto come una *variabile intera*
  - questa variabile viene inizializzata ad un valore *non negativo*
  - l'operazione **P**
    - attende che il valore del semaforo sia positivo
    - decrementa il valore del semaforo
  - l'operazione **V**
    - incrementa il valore del semaforo
- **Nota:**
  - le azioni **P** e **V** sono atomiche;
  - **quella a fianco non è un'implementazione**

```
class Semaphore {

  private int val;

  Semaphore(int init) {
   val = init;
  }

  void P() {
    < while (val<=0); val-- >
  }


  void V() {
    < val++ >
  }
}
```

Semafori - Implementazione di CS

```
Semaphore s = new Semaphore(1);
process P {
   while (true) {
      s.P();
      critical section
      s.V();
      non-critical section
   }
}
```

- **Si può dimostrare che le proprietà sono rispettate**
  - mutua esclusione, assenza di deadlock, assenza di starvation, assenza di ritardi non necessari

Semafori - Considerazioni

- **Implementazione**
  - l'implementazione precedente è basata su busy waiting, come le soluzioni software
  - se i semafori sono implementati a livello del S.O., è possibile limitare l'utilizzazione di busy waiting
  - per questo motivo:
    - l'operazione **P** può *sospendere* il processo invocante
    - l'operazione **V** può *svegliare* uno dei processi sospesi
  -

```
class Semaphore {
 private int val;
 Semaphore(v) { val = v; }

 void P() {
   val--;
   if (val < 0) {
     suspend this process
   }
  }

 void V() {
   val++;
   if (val <= 0) {
     wakeup one of the
     suspended processes
   }
 }
}
```

Semafori - Politiche di gestione dei processi bloccati

- **Per ogni semaforo,**
  - il S.O. deve mantenere una struttura dati contenente l'insieme dei processi sospesi
  - quando un processo deve essere svegliato, è necessario selezionare uno dei processi sospesi
- **Semafori FIFO**
  - politica first-in, first-out
  - il processo che è stato sospeso più a lungo viene svegliato per primo
  - è una politica fair, che garantisce assenza di starvation
  - la struttura dati è una *coda*

Semafori - Politiche di gestione dei processi bloccati

- **Semafori generali**
  - se non viene specificata l'ordine in cui vengono rimossi, i semafori possono dare origine a starvation
- **Nel seguito**
  - se non altrimenti specificato, utilizzeremo sempre semafori FIFO

Semafori - Implementazione

- **Primitive P e V fornite dal sistema operativo**

```
void P() {
  value--;
  if (value < 0) {
    pid = <id del processo
           che ha invocato P>;
    queue.add(pid);
    suspend(pid);
   }
 }

void V() {
  value++;
  if (value <= 0)
    pid = queue.remove();
    wakeup(pid);
 }
}
```

Il process id del processo bloccato viene messo in un insieme **queue**

Con l'operazione **suspend**, il s.o mette il processo nello stato **waiting**

Il process id del processo da sbloccare viene selezionato (secondo una certa politica) dall'insieme **queue**

Con l'operazione **wakeup**, il S.O. mette il processo nello stato **ready**

Semafori - Implementazione

- **L'implementazione precedente non è completa**
  - **P** e **V** devono essere eseguite in modo atomico
- **In un sistema uniprocessore**
  - è possibile disabilitare/riabilitare gli interrupt all'inizio/fine di **P** e **V**
  - note:
    - è possibile farlo perchè **P** e **V** sono implementate direttamente dal sistema operativo
    - l'intervallo temporale in cui gli interrupt sono disabilitati è molto breve
    - ovviamente, eseguire un'operazione **suspend** deve comportare anche la riabilitazione degli interrupt
- **In un sistema multiprocessore**
  - è possibile disabilitare gli interrupt?

Semafori - Implementazione

- **In un sistema multiprocessore**
  - è necessario utilizzare una delle tecniche di critical section viste in precedenza
    - tecniche software: Dekker, Peterson
    - tecniche hardware: test&set, swap, etc.

```
void P() {
    [enter CS]
    value--;
    if (value < 0) {
      int pid = <id del processo
              che ha invocato P>;
      queue.add(pid);
      suspend(pid);
     }
   }
   [exit CS]
}
```

```
void  V() {
    [enter CS]
    value++;
    if (value <= 0)
      int pid = queue.remove();
      wakeup(pid);
    [exit CS]
 }
}
```

Semafori - Vantaggi

- **Nota:**
  - utilizzando queste tecniche, non abbiamo eliminato busy-waiting
  - abbiamo però limitato busy-waiting alle sezioni critiche di **P** e **V**, e queste sezioni critiche sono molto brevi
  - in questo modo
    - la sezione critica non è quasi mai occupata
    - busy waiting avviene raramente

Semafori - Vantaggi

Senza semafori

```
<enter CS>
/*codice critico*/
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
/*fine cod.crit.*/
 <exit CS>
```

potenziale busy waiting

Con semafori

```
P  <enter CS>
     ...
   <exit CS>
/*codice critico*/
   ...
   ...
   ...
   ...
   ...
   ...
   ...
   ...
   ...
   ...
   ...
   ...
   ...
   ...
   ...
   ...
   ...
/*fine cod.crit.*/
V  <enter CS>
     ...
   <exit CS>
```

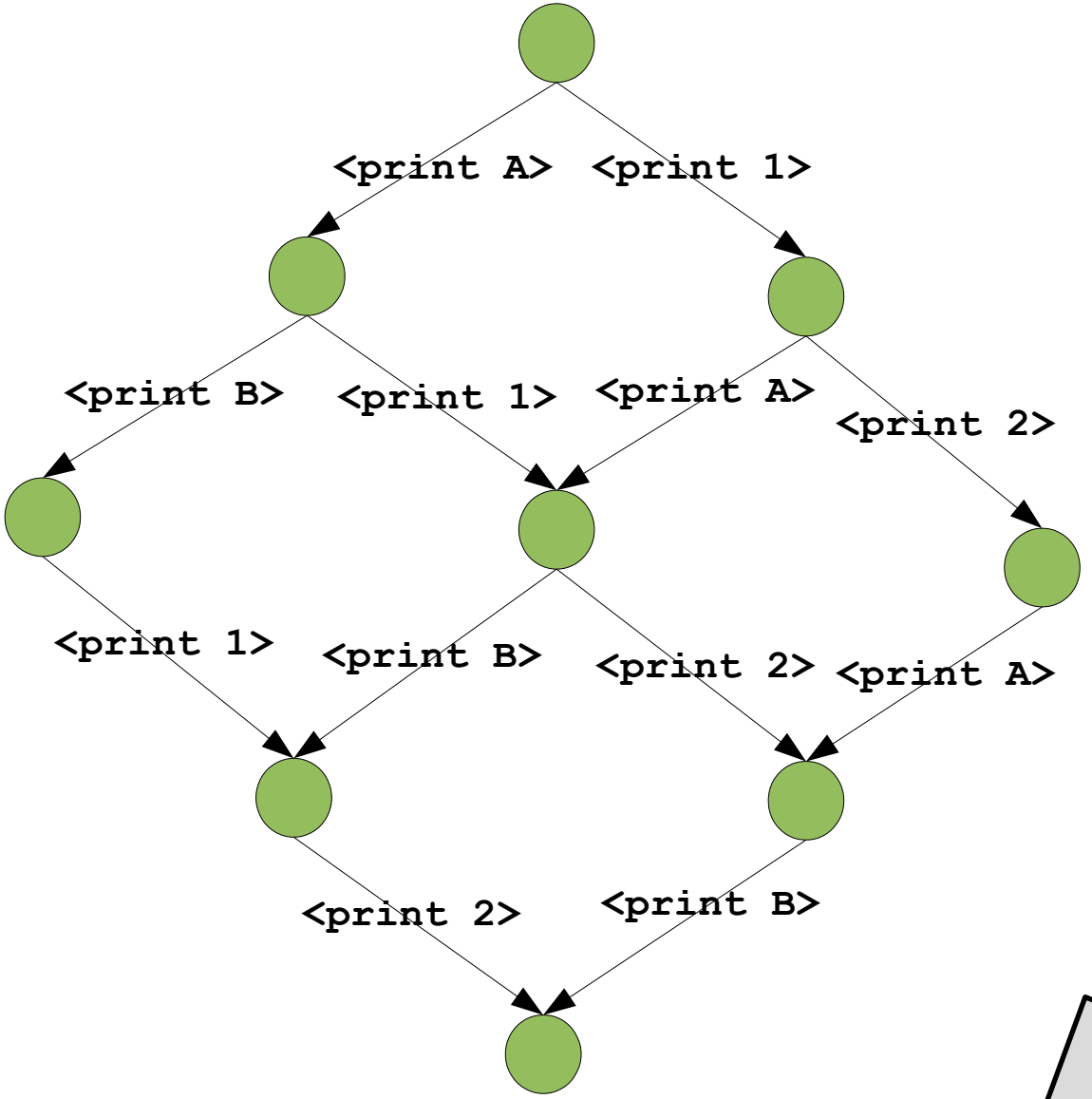
potenziale busy waiting

potenziale busy waiting

Interleaving di azioni atomiche

- **Cosa stampa questo programma? (Vi ricordate?)**

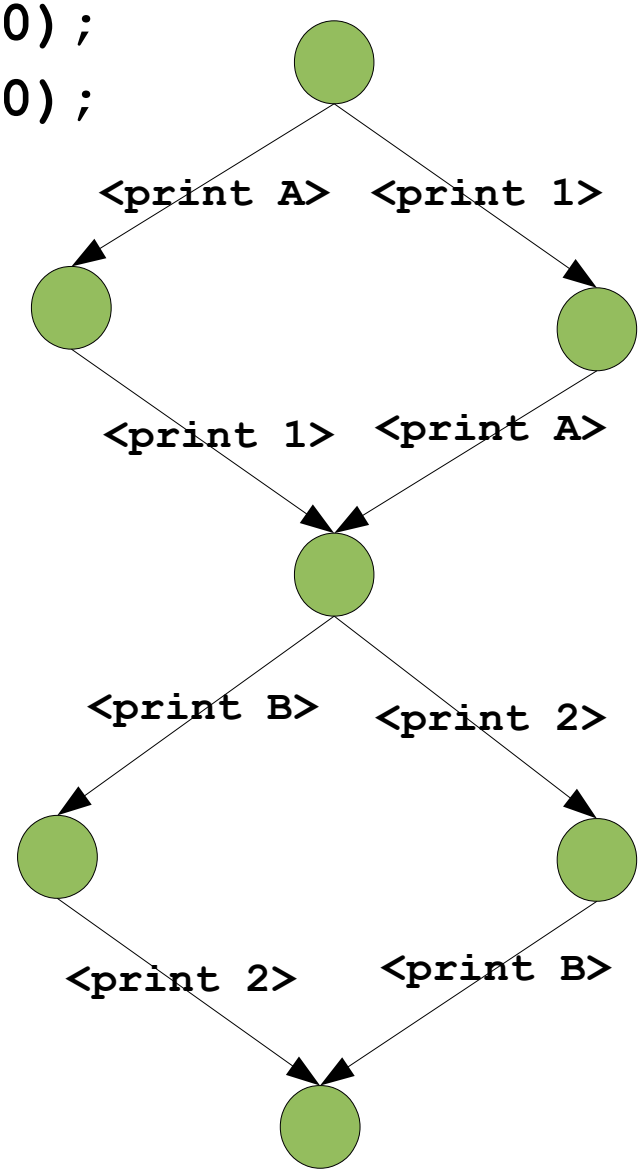
```
process P {
   <print A>
   <print B>
}
process Q {
   <print 1>
   <print 2>
}
```



Interleaving con semafori

- **Cosa stampa questo programma?**

```
Semaphore s1 = new Semaphore(0);
Semaphore s2 = new Semaphore(0);
process P {
   <print A>
   s1.V()
   s2.P()
   <print B>
}
process Q {
   <print 1>
   s2.V()
   s1.P()
   <print 2>
}
```



Problemi classici

- **Esistono un certo numero di problemi "classici" della programmazione concorrente**
  - *produttore/consumatore* (producer/consumer)
  - *buffer limitato* (bounded buffer)
  - *filosofi a cena* (dining philosophers)
  - *lettori e scrittori* (readers/writers)
- **Nella loro semplicità**
  - rappresentano le interazioni tipiche dei processi concorrenti

Produttore/consumatore

- **Definizione**
  - esiste un processo "produttore" **`Producer`** che genera valori (record, caratteri, oggetti, etc.) e vuole trasferirli a un processo "consumatore" **`Consumer`** che prende i valori generati e li "consuma"
  - la comunicazione avviene attraverso una singola variabile condivisa
- **Proprietà da garantire**
  - **`Producer`** non deve scrivere nuovamente l'area di memoria condivisa prima che **`Consumer`** abbia effettivamente utilizzato il valore precedente
  - **`Consumer`** non deve leggere due volte lo stesso valore, ma deve attendere che **`Producer`** abbia generato il successivo
  - assenza di deadlock

Produttore/consumatore - Implementazione

```
shared Object buffer;

Semaphore empty =
   new Semaphore(1);

Semaphore full =
   new Semaphore(0);

cobegin

   Producer

//

   Consumer

coend
```

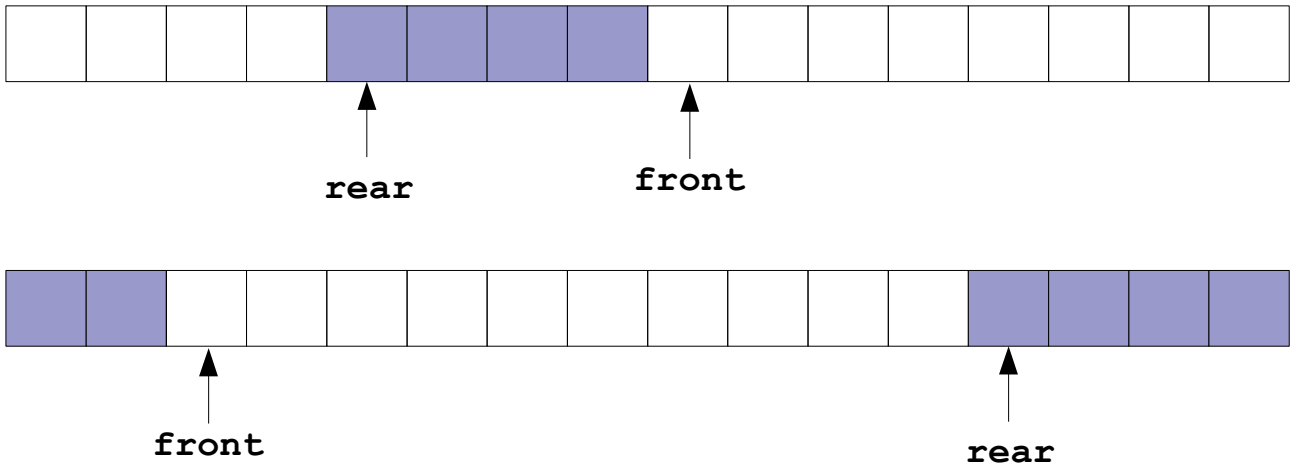
```
process Producer {
   while (true) {
      Object val = produce();
      empty.P();
      buffer = val;
      full.V();
   }
}

process Consumer {
   while (true) {
      full.P();
      Object val = buffer;
      empty.V();
      consume(val);
   }
}
```

Buffer limitato

- **Definizione**
  - è simile al problema del produttore / consumatore
  - in questo caso, però, lo scambio tra produttore e consumatore non avviene tramite un singolo elemento, ma tramite un buffer di dimensione limitata, i.e. un vettore di elementi
- **Proprietà da garantire**
  - `Producer` non deve sovrascrivere elementi del buffer prima che `Consumer` abbia effettivamente utilizzato i relativi valori
  - `Consumer` non deve leggere due volte lo stesso valore, ma deve attendere che `Producer` abbia generato il successivo
  - assenza di deadlock
  - assenza di starvation

Buffer limitato - struttura dei buffer

- **Array circolare:**
  - si utilizzano due indici **`front`** e **`rear`** che indicano rispettivamente il prossimo elemento da scrivere e il prossimo elemento da leggere
  - gli indici vengono utilizzati in modo ciclico (modulo l'ampiezza del buffer)
- **Esempi:**



Buffer limitato - Implementazione

```
Object buffer[SIZE];
int front = 0;
int rear = 0;
Semaphore empty =
  new Semaphore(SIZE);
Semaphore full =
  new Semaphore(0);
cobegin
  Producer
//
  Consumer
coend
```

```
process Producer {
  while (true) {
    Object val = produce();
    empty.P();
    buf[front] = val;
    front = (front + 1) % SIZE;
    full.V();
  }
}
process Consumer {
  while (true) {
    full.P();
    Object val = buf[rear];
    rear = (rear + 1) % SIZE;
    empty.V();
    consume(val);
  }
}
```

Generalizzare gli approcci precedenti

- **Questione**
  - è possibile utilizzare il codice del lucido precedente con produttori e consumatori multipli?
- **Caso 1: Produttore/Consumatore**
  - è possibile che un valore sia sovrascritto?
  - è possibile che un valore sia letto più di una volta?
- **Caso 2: Buffer limitato**
  - è possibile che un valore sia sovrascritto?
  - è possibile che un valore sia letto più di una volta?
  - possibilità di deadlock?
  - possibilità di starvation?

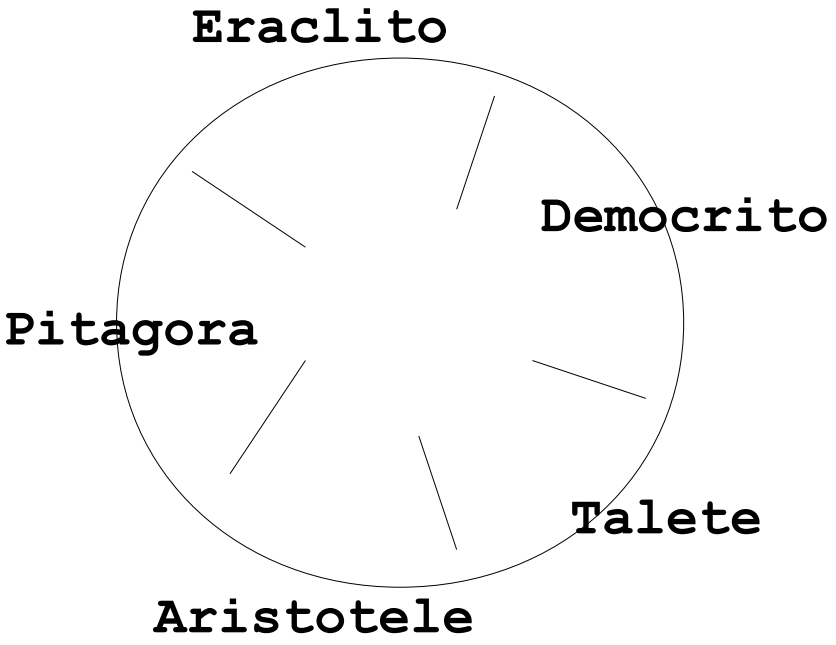
Buffer limitato - Produttori/Consumatori multipli

```
Object buffer[SIZE];
int front = rear = 0;
Semaphore mutex = new Semaphore(1);
Semaphore empty =
  new Semaphore(SIZE);
Semaphore full = new Semaphore(0);
cobegin Producer // Consumer coend

process Producer {
  while (true) {
    Object val = produce();
    empty.P();
    mutex.P();
    buf[front] = val;
    front = (front + 1) % SIZE;
    mutex.V();
    full.V();
  }
}
```

```
process Consumer {
  while (true) {
    full.P();
    mutex.P();
    Object val = buf[rear];
    rear = (rear + 1) % SIZE;
    mutex.V();
    empty.V();
    consume(val);
  }
}
```

Cena dei Filosofi

- **Descrizione**
  - cinque filosofi passano la loro vita a pensare
    e a mangiare (alternativamente)
  - per mangiare fanno uso di una tavola rotonda con 5 sedie, 5 piatti e 5 posate fra i piatti
  - per mangiare, un filosofo ha bisogno di entrambe le posate (destra/sinistra)
  - per pensare, un filosofo lascia le posate dove le ha prese



Cena dei Filosofi

- **Note**
  - nella versione originale, i filosofi mangiano spaghetti con due forchette

- **La nostra versione**
  - filosofi orientali
  - riso al posto di spaghetti
  - bacchette (chopstick) al posto di forchette

Filosofi perché?

- **I problemi produttore/consumatore e buffer limitato**
  - mostrano come risolvere il problema di accesso esclusivo a una o più risorse indipendenti
- **Il problema dei filosofi**
  - mostra come gestire situazioni in cui i processi entrano in competizione per accedere ad insiemi di risorse a intersezione non nulla
  - le cose si complicano....

La vita di un filosofo

```
process Philo[i] {  /* i = 0...4 */
   while (true) {
      think
      acquire chopsticks
      eat
      release chopsticks
   }
}
```

- **Le bacchette vengono denominate:**
  - `chopstick[i]` con `i=0...4;`
- **Il filosofo i**
  - accede alle posate `chopstick[i]` e `chopstick[(i+1)%5];`

Cena dei Filosofi - Soluzione errata

```
Semaphore chopsticks =
  { new Semaphore(1), ..., new Semaphore(1) };

process Philo[i] {  /* i = 0...4 */
  while (true) {
    think
    chopstick[i].P();
    chopstick[(i+1)%5].P();
    eat
    chopstick[i].V();
    chopstick[(i+1)%5].V();
  }
}
```

- **Perché è errata?**

Cena dei Filosofi - Soluzione errata

- **Perché è errata?**
  - Perché tutti i filosofi possono prendere la bacchetta di sinistra (indice `i`) e attendere per sempre che il filosofo accanto rilasci la bacchetta che è alla destra (indice `(i+1)%5`)
  - Nonostante i filosofi muoiano di fame, questo è un caso di deadlock...
- **Come si risolve il problema?**

Cena dei Filosofi - Soluzione corretta

- **Come si risolve il problema?**
  - Eliminando il caso di attesa circolare
  - Rompendo la simmetria!
  - E' sufficiente che uno dei filosofi sia mancino:
    - cioè che prenda prima la bacchetta opposta rispetto a tutti i colleghi, perché il problema venga risolto

Cena dei Filosofi - Soluzione corretta

```
Semaphore chopsticks =
  { new Semaphore(1), ..., new Semaphore(1) };

process Philo[0] {
 while (true) {
  think
  chopstick[1].P();
  chopstick[0].P();
  eat
  chopstick[1].V();
  chopstick[0].V();
 }
}
```

```
process Philo[i] {  /* i = 1...4 */
 while (true) {
  think
  chopstick[i].P();
  chopstick[(i+1)%5].P();
  eat
  chopstick[i].V();
  chopstick[(i+1)%5].V();
 }
}
```

Cena dei Filosofi - Soluzione corretta

- **Filosofi: altre soluzioni**
  - i filosofi di indice pari sono mancini, gli altri destri
    - in caso di collisione, un filosofo deve attendere che i due vicini abbiano terminato
  - al più quattro filosofi possono sedersi a tavola
    - agente esterno controllore
  - le bacchette devono essere prese insieme
    - necessaria un'ulteriore sezione critica
- **Cosa dire rispetto a starvation?**

Lettori e scrittori

- **Descrizione**
  - un database è condiviso tra un certo numero di processi
  - esistono due tipi di processi
  - i **lettori** accedono al database per leggerne il contenuto
  - gli **scrittori** accedono al database per aggiornarne il contenuto
- **Proprietà**
  - se uno scrittore accede a un database per aggiornarlo, esso opera in mutua esclusione; nessun altro lettore o scrittore può accedere al database
  - se nessuno scrittore sta accedendo al database, un numero arbitrario di lettori può accedere al database in lettura

Lettori e scrittori

- **Motivazioni**
  - la competizione per le risorse avviene a livello di *classi di processi* e non solo a livello di processi
  - mostra che mutua esclusione e condivisione possono anche coesistere
- **Definizioni**
  - sia **`nr`** il numero dei lettori che stanno accendo al database
  - sia **`nw`** il numero di scrittori che stanno accedendo al database
  - Condizioni possibili:

    ```
    (nr > 0 && nw==0) || (nr == 0 && nw <= 1)
    ```
- **Note**
  - il controllo può passare dai lettori agli scrittori o viceversa quando:

    ```
    nr == 0 && nw == 0
    ```

Vita dei lettori e degli scrittori

```
process Reader {
  while (true) {
    startRead();
    read the database
    endRead();
  }
}
```

- **Note:**
  - `startRead()` e **`endRead()`** contengono le operazioni necessarie affinché un lettore ottenga accesso al db

```
process Writer {
  while (true) {
    startWrite();
    write the database
    endWrite();
  }
}
```

- **Note:**
  - `startWrite()` e **`endWrite()`** contengono le operazioni necessarie affinchè uno scrittore ottenga accesso al database

Lettori e scrittori

- **Il problema dei lettori e scrittori ha molte varianti**
  - molte di queste varianti si basano sul concetto di priorità
- **Priorità ai lettori**
  - se un lettore vuole accedere al database, lo potrà fare senza attesa a meno che uno scrittore non abbia già acquisito l'accesso al database
  - scrittori: possibilità di starvation
- **Priorità agli scrittori**
  - uno scrittore attenderà il minimo tempo possibile prima di accedere al db
  - lettori: possibilità di starvation

Lettori e scrittori - Soluzione

```
/* Variabili condivise */
int nr = 0;
Semaphore rw = new
   Semaphore(1);
Semaphore mutex = new
   Semaphore(1);

void startRead() {
  mutex.P();
  if (nr == 0)
    rw.P();
  nr++;
  mutex.V();
}

void startWrite() {
  rw.P();
}
```

```
void endRead() {
  mutex.P();
  nr--;
  if (nr == 0)
    rw.V();
  mutex.V();
}

void endWrite() {
  rw.V();
}
```

- **Problemi**
  - è possibile avere starvation per i lettori?
  - è possibile avere starvation per gli scrittori?

Semafori - Conclusione

- **Difetti dei semafori**
  - Sono costrutti di basso livello
  - E' responsabilità del programmatore non commettere alcuni possibili errori "banali"
    - omettere **P** o **V**
    - scambiare l'ordine delle operazioni **P** e **V**
    - fare operazioni **P** e **V** su semafori sbagliati
  - E' responsabilità del programmatore accedere ai dati condivisi in modo corretto
    - più processi (scritti da persone diverse) possono accedere ai dati condivisi
    - cosa succede nel caso di incoerenza?
  - Vi sono forti problemi di "leggibilità"

Sistemi Operativi per LT Informatica
A.A. 2017-2018

Monitor

Docente: Salvatore Sorce

Monitor - Introduzione

- **I monitor**
  - sono un paradigma di programmazione concorrente che fornisce un approccio più strutturato alla programmazione concorrente
- **Storia**
  - introdotti nel 1974 da Hoare
  - implementati in certo numero di linguaggi di programmazione, fra cui Concurrent Pascal, Pascal-plus, Modula-2, Modula-3 e Java

Monitor - Introduzione

- **Un monitor è un modulo software che consiste di:**
  - dati locali
  - una sequenza di inizializzazione
  - una o più "procedure"
- **Le caratteristiche principali sono:**
  - i dati locali sono accessibili solo alle procedure del modulo stesso
  - un processo entra in un monitor invocando una delle sue procedure
  - solo un processo alla volta può essere all'interno del monitor; gli altri processi che invocano il monitor sono sospesi, in attesa che il monitor diventi disponibile

Monitor - Sintassi

```
monitor name {
    variable declarations...
    procedure entry type procedurename1(args...) {
       ...
    }
    type procedurename2(args...) {
       ...
    }
    name(args...) {
       ...
    }
}
```

variabili private del monitor

procedure visibili all'esterno

procedure private

inizializzazione

Monitor - Alcuni paragoni

- **Assomiglia ad un "oggetto" nella programmazione o.o.**
  - il codice di inizializzazione corrisponde al costruttore
  - le *procedure entry* sono richiamabili dall'esterno e corrispondono ai metodi pubblici di un oggetto
  - le procedure "normali" corrispondono ai metodi privati
  - le variabili locali corrispondono alle variabili pubbliche
- **Sintassi**
  - originariamente, sarebbe basata su quella del Pascal
    - var, procedure entry, etc.
  - in questi lucidi, utilizziamo una sintassi simile a Java

Monitor - Caratteristiche base

- **Solo un processo alla volta può essere all'interno del monitor**
  - il monitor fornisce un semplice meccanismo di mutua esclusione
  - strutture dati condivise possono essere messe all'interno del monitor
- **Per essere utile per la programmazione concorrente, è necessario un meccanismo di sincronizzazione**
- **Abbiamo necessità di:**
  - poter sospendere i processi in attesi di qualche condizione
  - far uscire i processi dalla mutua esclusione mentre sono in attesa
  - permettergli di rientrare quando la condizione è verificata

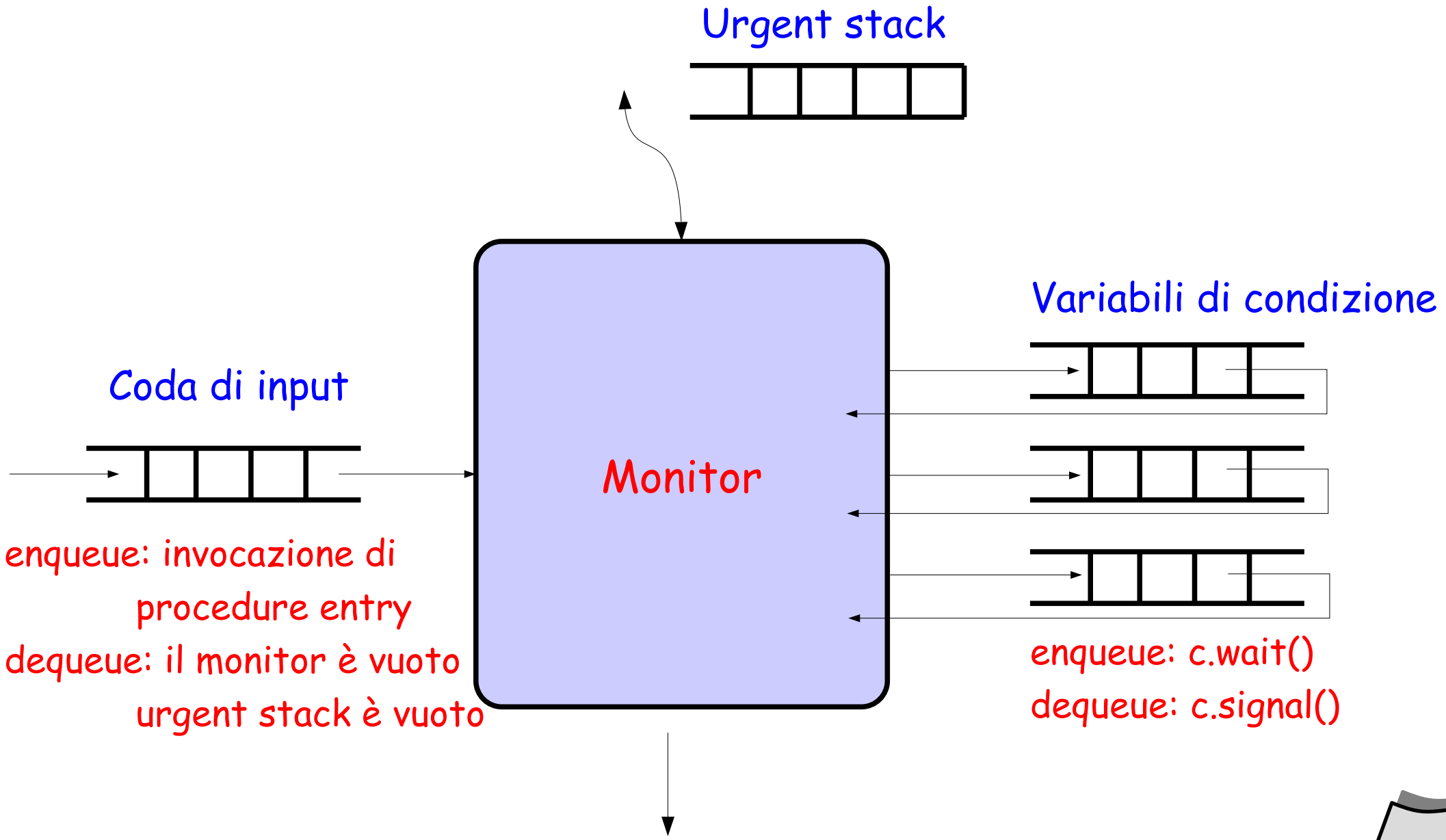
Monitor - Meccanismi di sincronizzazione

- **Dichiarazione di variabili di condizione (CV)**
  - `condition c;`
- **Le operazioni definite sulle CV sono:**
  - `c.wait()`
    attende il verificarsi della condizione
  - `c.signal()`
    segnala che la condizione è vera

Monitor - Politica signal urgent

- `c.wait()`
  - viene rilasciata la mutua esclusione
  - il processo che chiama `c.wait()` viene sospeso in una coda di attesa della condizione `c`
- `c.signal()`
  - causa la riattivazione immediata di un processo (secondo una politica FIFO)
  - il chiamante viene posto in attesa
  - verrà riattivato quando il processo risvegliato avrà rilasciato la mutua esclusione (*urgent stack*)
  - se nessun processo sta attendendo `c` la chiamata non avrà nessun effetto

Monitor - Rappresentazione intuitiva



Monitor - wait/signal vs P/V

- **A prima vista:**
  - `wait` e `signal` potrebbero sembrare simili alle operazioni sui semafori
    P e **V**
- **Non è vero!**
  - `signal` non ha alcun effetto se nessun processo sta attendendo la condizione
    V "memorizza" il verificarsi degli eventi
  - `wait` è sempre bloccante
    P (se il semaforo ha valore positivo) no
  - il processo risvegliato dalla `signal` viene eseguito per primo

Monitor - Politiche di signaling

- **Signal urgent è la politica "classica" di signaling**
  - SU - *signal urgent*
    - proposta da Hoare
- **Ne esistono altre:**
  - SW - *signal wait*
    - no urgent stack, signaling process viene messo nella entry queue
  - SR - *signal and return*
    - dopo la **`signal`** si esce subito dal monitor
  - SC - *signal and continue*
    - la **`signal`** segnala solamente che un processo può continuare, il chiamante prosegue l'esecuzione
    - quando lascia il monitor viene riattivato il processo segnalato

Monitor - Implementazione dei semafori

```
monitor Semaphore {
  int value;
  condition c;          /* value > 0 */

  procedure entry void P() {
    value--;
    if (value < 0)
      c.wait();
  }

  procedure entry void V() {
    value++;
    c.signal();
  }

  Semaphore(int init) {
    value = init;
  }
}
```

R/W tramite Monitor

```
process Reader {
  while (true) {
    rwController.startRead();
    read the database
    rwController.endRead();
  }
}

process Writer {
  while (true) {
    rwController.startWrite();
    write the database
    rwController.endWrite();
  }
}
```

R/W tramite Monitor

```
monitor RWController
  int nr;                          /* number of readers */
  int nw;                          /* number of writers */
  condition okToRead;              /* nw == 0 */
  condition okToWrite;             /* nr == 0 && nw == 0 */

  procedure entry void startRead() {
    if (nw != 0)
      okToRead.wait();
    nr = nr + 1;
    if (nw == 0)                         /* always true */
      okToRead.signal();
    if (nw == 0 && nr == 0)              /* always false */
      okToWrite.signal();
  }
```

R/W tramite Monitor

```
procedure entry void endRead() {
  nr = nr - 1;
  if (nw == 0)                          /* true but useless */
    okToRead.signal();
  if (nw == 0 && nr == 0)
    okToWrite.signal();
}

procedure entry void startWrite() {
  if (!(nr==0 && nw==0))
    okToWrite.wait();
  nw = nw + 1;
  if (nw == 0)                          /* always true */
    okToRead.signal();
  if (nw == 0 && nr == 0)               /* always false */
    okToWrite.signal();
}
```

R/W tramite Monitor

```
procedure entry void endWrite() {
  nw = nw - 1;
  if (nw == 0)                          /* Always true */
    okToRead.signal();
  if (nw == 0 && nr == 0)
    okToWrite.signal();
}

RWController() {                              /* Constructor */
  nr = nw = 0;
}
```

- **E' possibile semplificare il codice**
  - eliminando le righe `if` quando sempre vere
  - elimanando le righe `if` e il ramo opportuno quando sempre falso

R/W tramite monitor - semplificato

```
procedure entry void startRead() {
  if (nw != 0) okToRead.wait();
  nr = nr + 1;
  okToRead.signal();
}
procedure entry void endRead() {
  nr = nr - 1;
  if (nr == 0) okToWrite.signal();
}
procedure entry void startWrite() {
  if (!(nr=0 && nw =0)) okToWrite.wait();
  nw = nw + 1;
}
procedure entry void endWrite() {
  nw = nw - 1;
  okToRead.signal();
  if (nw == 0 && nr == 0) okToWrite.signal();
}
```

Produttore / consumatore tramite Monitor

```
process Producer {
  Object x;
  while (true) {
    x = produce();
    pcController.write(x);
  }
}

process Consumer {
  Object x;
  while (true) {
    x = pcController.read();
    consume(x);
  }
}
```

Produttore / consumatore tramite Monitor

```
monitor PCController {
  Object buffer;
  condition empty;
  condition full;
  boolean   isFull;

  PCController() {
    isFull=false;
  }

  procedure entry Object read() {
    if (!isFull)
      full.wait();
    int retvalue = buffer;
    isFull = false;
    empty.signal();
    return retvalue;
  }

  procedure entry void write(int val)
  {
    if (isFull)
      empty.wait();
    buffer = val;
    isFull = true;
    full.signal();
  }
}
```

Buffer limitato tramite Monitor

```
monitor PCController {
  Object[] buffer;
  condition okRead, okWrite;
  int count, rear, front;

  PCController(int size) {
    buffer = new Object[size];
    count = rear = front = 0;
  }

  procedure entry Object read() {
    if (count == 0)
      okRead.wait();
    int retval = buffer[rear];
    cont--;
    rear = (rear+1) % buffer.length;
    okWrite.signal();
    return retval;
  }

  procedure entry void write(int val)
  {
    if (count == buffer.length)
      okWrite.wait();
    buffer[front] = val;
    count++;
    front = (front+1) %
 buffer.length;
    okRead.signal();
  }
}
```

Filosofi a cena

```
process Philo[i] {
  while (true) {
    think
    dpController.startEating();
    eat
    dpController.finishEating();
  }
}
```

Filosofi a cena

```
monitor DPController {
  condition[] oktoeat = new condition[5];
  boolean[]   eating  = new boolean[5];
  procedure entry void startEating(int i) {
    if (eating[i-1] || eating[i+1])
      oktoeat[i].wait();
    eating[i] = true;
  }
  procedure entry void finishEating(int i) {
    eating[i] = false;
    if (!eating[i-2])
      oktoeat[i-1].signal();
    if (!eating[i+2])
      oktoeat[i+1].signal();
  }
  DPcontroller() {
    for(int i=0; i<5; i++) eating[i] = false;
  }
}
```

Nota: i ± h corrisponde a

(i ± h)%5

Filosofi a cena - No deadlock

```
monitor DPController {
  condition[] unusedchopstick = new condition[5];
  boolean[] chopstick  = new boolean[5];
  procedure entry void startEating(int i) {
    if (chopstick[MIN(i,i+1)])
      unusedchopstick[MIN(i,i+1)].wait();
    chopstick[MIN(i,i+1)] = true;
    if (chopstick[MAX(i,i+1)])
      unusedchopstick[MAX(i,i+1)].wait();
    chopstick[MAX(i,i+1)] = true;
  }
  procedure entry void finishEating(int i) {
    chopstick[i] = false;
    chopstick[i+1] = false;
    unusedchopstick[i].signal();
    unusedchopstick[i+1].signal();
  }
}
```

Nota: $i \pm h$ corrisponde a $(i \pm h)\%5$

Filosofi a cena - No deadlock

```
monitor DPController {
  condition[] unusedchopstick = new condition[5];
  boolean[]   chopstick  = new boolean[5];
  procedure entry void startEating(int i) {
    if (chopstick[i])
      unusedchopstick[i].wait();
    chopstick[i] = true;
    if (chopstick[i+1])
      unusedchopstick[i+1].wait();
    chopstick[i+1] = true;
  }
  procedure entry void finishEating(int i) {
    chopstick[i] = false;
    chopstick[i+1] = false;
    unusedchopstick[i].signal();
    unusedchopstick[i+1].signal();
  }
}
```

Nota: i ± h corrisponde a

(i ± h)%5

Filosofi a cena

```
process Philo[i] {
  while (true) {
    think
    chopstick[MIN(i,i+1)].pickup();
    chopstick[MAX(i,i+1)].pickup();
    eat
    chopstick[MIN(i,i+1)].putdown();
    chopstick[MAX(i,i+1)].putdown();
  }
}
```

Nota: i ± h corrisponde a

(i ± h)%5

Filosofi a cena

```
monitor chopstick[i] {
  boolean inuse = false;
  condition free;

  procedure entry void pickup() {
    if (inuse)
      free.wait();
    inuse = true;
  }

  procedure entry void putdown() {
    inuse = false;
    free.signal();
  }
```

Sistemi Operativi per LT Informatica
A.A. 2017-2018

Message Passing

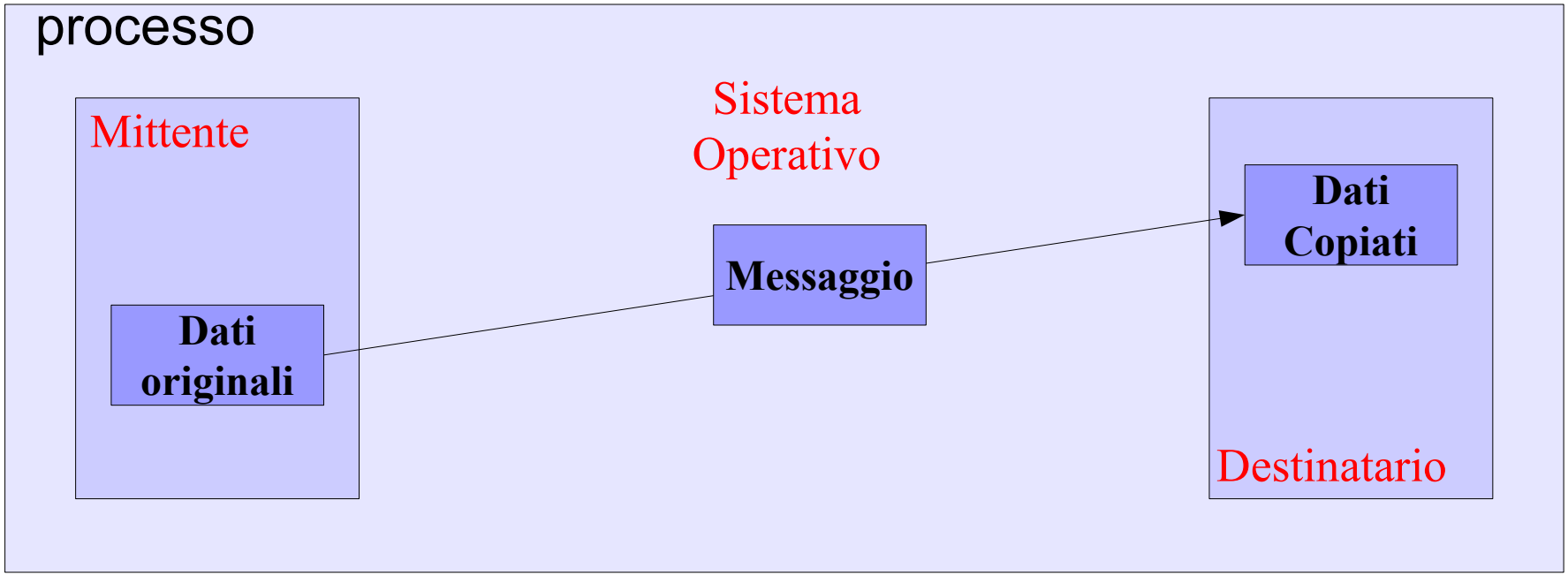
Docente: Salvatore Sorce

Message Passing - Introduzione

- **Paradigmi di sincronizzazione**
  - semafori, monitor sono paradigmi di *sincronizzazione* tra processi
  - in questi paradigmi, la *comunicazione* avviene tramite memoria condivisa
- **Paradigmi di comunicazione**
  - il meccanismo detto *message passing* è un paradigma di *comunicazione* tra processi
  - la *sincronizzazione* avviene tramite lo scambio di messaggi, e non più semplici segnali

Message Passing - Definizioni

- **Un messaggio**
  - è un insieme di informazioni formattate da un processo *mittente* e interpretate da un processo *destinatario*
- **Un meccanismo di "scambio di messaggi"**
  - copia le informazioni di un messaggio da uno spazio di indirizzamento di un processo allo spazio di indirizzamento di un altro processo



Message Passing - Operazioni

- **send:**
  - utilizzata dal processo mittente per "spedire" un messaggio ad un processo destinatario
  - il processo destinatario deve essere specificato
- **receive:**
  - utilizzata dal processo destinatario per "ricevere" un messaggio da un processo mittente
  - il processo mittente può essere specificato, o può essere qualsiasi

Message Passing

- **Note:**
  - il passaggio dallo spazio di indirizzamento del mittente a quello del destinatario è mediato dal sistema operativo (protezione memoria)
  - il processo destinatario deve eseguire un'operazione **receive** per ricevere qualcosa

Message Passing - Tassonomia

- **MP sincrono**
  - Send sincrono
  - Receive bloccante
- **MP asincrono**
  - Send asincrono
  - Receive bloccante
- **MP completamente asincrono**
  - Send asincrono
  - Receive non bloccante

MP Sincrono

- **Operazione send sincrona**
  - sintassi: `ssend(m, q)`
  - il mittente `p` spedisce il messaggio `m` al processo `q`, restando bloccato fino a quando `q` non esegue l'operazione `sreceive(m, p)`
- **Operazione receive bloccante**
  - sintassi: `m = sreceive(p)`
  - il destinatario `q` riceve il messaggio `m` dal processo `p`; se il mittente non ha ancora spedito alcun messaggio, il destinatario si blocca in attesa di ricevere un messaggio
  - è possibile lasciare il mittente non specificato (utilizzando *)

MP Asincrono

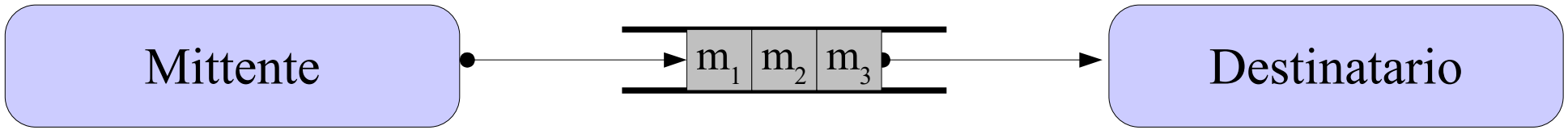
- **Operazione send asincrona**
  - sintassi: **`asend(m, q)`**
  - il mittente **`p`** spedisce il messaggio **`m`** al processo **`q`**, senza bloccarsi in attesa che il destinatario esegua l'operazione **`areceive(m, p)`**
- **Operazione receive bloccante**
  - sintassi: **`m = areceive(p)`**
  - il destinatario **`q`** riceve il messaggio **`m`** dal processo **`p`**; se il mittente non ha ancora spedito alcun messaggio, il destinatario si blocca in attesa di ricevere un messaggio
  - è possibile lasciare il mittente non specificato (utilizzando *)

MP Totalmente Asincrono

- **Operazione send asincrona**
  - sintassi: **`asend(m, q)`**
  - il mittente **`p`** spedisce il messaggio **`m`** al processo **`q`**, senza bloccarsi in attesa che il destinatario esegua l'operazione **`nb-receive(m, p)`**
- **Operazione receive non bloccante**
  - sintassi: **`m = nb-receive(p)`**
  - il destinatario **`q`** riceve il messaggio **`m`** dal processo **`p`**; se il mittente non ha ancora spedito alcun messaggio, la **`nb-receive`** termina ritornando un messaggio "nullo"
  - è possibile lasciare il mittente non specificato (utilizzando *)

MP sincrono e asincrono

Message passing asincrono



Message passing sincrono

MP sincrono dato quello asincrono

```
void ssend(Object msg, Process q) {
  asend(msg, q);
  ack = areceive(q);
}
Object sreceive(p) {
  Object msg = areceive(p);
  asend(ack, p);
  return msg;
}
```

MP asincrono dato quello sincrono

```
/* p is the calling process */
void asend(Object m, Process q) {
  ssend("SND(m,p,q)", server);
}
void areceive(Process q) {
  ssend("RCV(p,q)", server);
  Object m = sreceive(server);
  return m;
}
process server {
  /* One element x process pair
   */
  int[][] waiting;
  Queue[][] queue;
  while (true) {
    handleMessage();
  }
}
```

```
void handleMessage() {
  msg = sreceive(*);
  if (msg == <SND(m,p,q)>) {
    if (waiting[p,q]>0) {
      ssend(m, p);
      waiting[p,q]--;
    } else {
      queue[p,q].add(m);
    }
  } else if (msg == <RCV(q,p)>) {
    if (queue[p,q].isEmpty()) {
      waiting[p,q]++;
    } else {
      m = queue[p,q].remove();
      ssend(m, dest);
    }
  }
}
```

Message Passing - Filosofi a cena

```
process Philo[i] {
  while (true) {
    think
    asend(<PICKUP,i>, chopstick[MIN(i, (i+1)%5)]);
    msg = areceive(chopstick[MIN(i, (i+1)%5)]);
    asend(<PICKUP,i>, chopstick[MAX(i, (i+1)%5)]);
    msg = areceive(chopstick[MAX(i, (i+1)%5)]);
    eat
    asend(<PUTDOWN,i>, chopstick[MIN(i, (i+1)%5)]);
    asend(<PUTDOWN,i>, chopstick[MAX(i, (i+1)%5)]);
  }
}
```

Message Passing - Filosofi a cena

```
process chopstick[i] {

  boolean free = true;
  Queue queue = new Queue();

  while (true) {
    handleRequests();
  }

}
```

```
void handleRequests() {
  msg = areceive(*);
  if (msg == <PICKUP,j>) {
    if (free) {
      free = false;
      asend(ACK, philo[j]);
    } else {
      queue.add(j);
    }
  } else
  if (msg == <PUTDOWN, j>) {
    if (queue.isEmpty()) {
      free = true;
    } else {
      k = queue.remove();
      asend(ACK, philo[k]);
    }
  }
}
```

Message Passing - Produttori e consumatori

```
process Producer  {
  Object x;
  while (true) {
    x = produce();
    ssend(x, PCmanager);
  }
}

process Consumer{
  Object x;
  while (true) {
    x = sreceive(PCmanager);
    consume(x);
  }
}
```

```
process PCmanager  {
  Object x;
  while (true) {
    x = sreceive(Producer);
    ssend(x, Consumer);
  }
}
```

Sezione 7

7. Conclusioni

Riassunto

- **Sezioni critiche**
  - meccanismi fondamentali per realizzare mutua esclusione in sistemi mono e multiprocessore all'interno del sistema operativo stesso
  - ovviamente livello troppo basso
- **Semafori**
  - fondamentale primitiva di sincronizzazione, effettivamente offerta dai S.O.
  - livello troppo basso; facile commettere errori
- **Monitor**
  - meccanismi integrati nei linguaggi di programmazione
  - pochi linguaggi di larga diffusione sono dotati di monitor;
  - unica eccezione Java, con qualche distinguo
- **Message passing**
  - da un certo punto di vista, il meccanismo più diffuso
  - può essere poco efficiente (copia dati tra spazi di indirizzamento)