Sistemi Operativi per LT Informatica

Concorrenza

Docente: Salvatore Sorce

Sommario

- **Introduzione alla concorrenza**
  - Modello concorrente. Processi. Stato di un processo. Multiprogramming e multiprocessing. Notazioni
- **Interazioni tra processi**
  - Tipi di interazione. Proprietà fondamentali. Mutua esclusione. Deadlock. Starvation. Azioni atomiche.
- **Sezioni critiche**
  - Tecniche software: Dekker, Peterson.
  - Tecniche hardware: disabilitazione interrupt, istruzioni speciali.
- **Semafori**
  - Definizione. Implementazione. Semafori generali e binari. Problemi classici.

Sommario

- **Monitor**
  - Definizione. Implementazione tramite semafori. Utilizzazione di monitor per implementare semafori. Risoluzione di problemi classici.
- **Message passing**
  - Definizione. Implementazione tramite semafori. Risoluzione di problemi classici
- **Conclusioni**
  - Riassunto. Rapporti fra paradigmi.

Sezione 1

1. Introduzione alla concorrenza

Introduzione

- Un sistema operativo consiste in un gran numero di *attività* che vengono eseguite più o meno *contemporaneamente* dal processore e dai dispositivi presenti in un elaboratore.
- Senza un modello adeguato, la coesistenza delle diverse attività sarebbe difficile da descrivere e realizzare.
- Il modello che è stato realizzato a questo scopo prende il nome di *modello concorrente* ed è basato sul concetto astratto di *processo*

Introduzione

- **In questa serie di lucidi:**
  - Analizzeremo il problema della gestione di attività multiple da un punto di vista astratto
  - Il modello concorrente rappresenta una rappresentazione astratta di un S.O. multiprogrammato.
- **Successivamente (nei prossimi moduli):**
  - Vedremo i dettagli necessari per la gestione di processi in un S.O. reale
  - In particolare, analizzeremo il problema dello *scheduling*, ovvero come un S.O. seleziona le attività che devono essere eseguite dal processore

Processi e programmi

- **Definizione: *processo***
  - E' un'attività controllata da un programma che si svolge su un processore
- **Un processo non è un programma!**
  - Un programma è un entità *statica*, un processo è *dinamico*
  - Un programma:
    - specifica un'insieme di istruzioni e la loro sequenza di esecuzione
    - non specifica la distribuzione nel tempo dell'esecuzione
  - Un processo:
    - rappresenta il modo in cui un programma viene eseguito nel tempo
- **Assioma di *finite progress***
  - Ogni processo viene eseguito ad una velocità finita ma sconosciuta

Stato di un processo

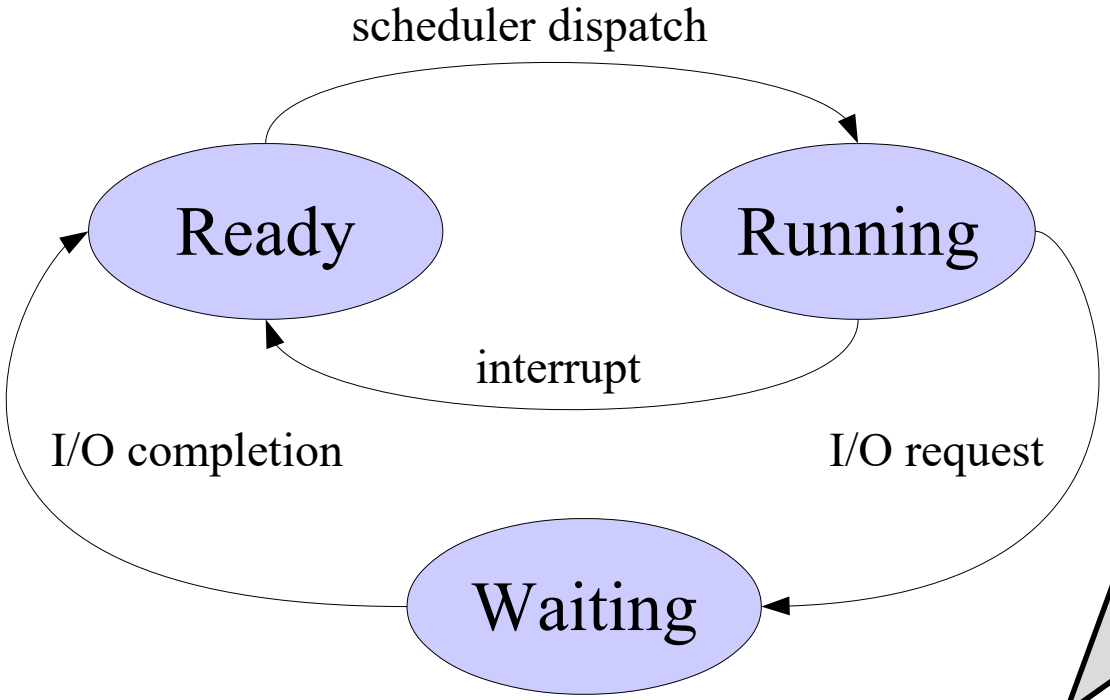
- **Ad ogni istante, un processo può essere totalmente descritto dalle seguenti componenti:**
  - *La sua immagine di memoria*
    - la memoria assegnata al processo (ad es. testo, dati, stack)
    - le strutture dati del S.O. associate al processo (ad es. file aperti)
  - *La sua immagine nel processore*
    - contenuto dei registri generali e speciali
  - *Lo stato di avanzamento*
    - descrive lo stato corrente del processo: ad esempio, se è in esecuzione o in attesa di qualche evento

Processi e programmi (ancora)

- **Più processi possono eseguire lo stesso programma**
  - In un sistema multiutente, più utenti possono leggere la posta contemporaneamente
  - Un singolo utente può eseguire più istanze dello stesso editor
- **In ogni caso, ogni istanza viene considerata un processo separato**
  - Possono condividere lo stesso codice ...
  - ... ma i dati su cui operano, l'immagine del processore e lo stato di avanzamento sono separati

Stati dei processi (versione semplice)

- **Stati dei processi:**
  - *Running*: il processo è in esecuzione
  - *Waiting*: il processo è in attesa di qualche evento esterno (ad es. completamento operazione di I/O); non può essere eseguito
  - *Ready*: il processo può essere eseguito, ma attualmente il processore è impegnato in altre attività



Nota: modello semplificato, nel seguito vedremo un modello più realistico

Cos'è la concorrenza?

- **Tema centrale nella progettazione dei S.O. riguarda la gestione di processi *multipli***
  - *Multiprogramming*
    - più processi su un solo processore
    - parallelismo *apparente*
  - *Multiprocessing*
    - più processi su una macchina con processori multipli
    - parallelismo *reale*
  - *Distributed processing*
    - più processi su un insieme di computer distribuiti e indipendenti
    - parallelismo *reale*

Cos'è la concorrenza?

- **Esecuzione concorrente:**
  - Due programmi si dicono in esecuzione concorrente se vengono eseguiti in parallelo (con parallelismo reale o apparente)
- **Concorrenza:**
  - E' l'insieme di notazioni per descrivere l'esecuzione concorrente di due o più programmi
  - E' l'insieme di tecniche per risolvere i problemi associati all'esecuzione concorrente, quali *comunicazione* e *sincronizzazione*

Dove possiamo trovare la concorrenza?

- **Applicazioni multiple**
  - la multiprogrammazione è stata inventata affinchè più processi indipendenti condividano il processore
- **Applicazioni strutturate su processi**
  - estensione del principio di progettazione modulare; alcune applicazioni possono essere progettate come un insieme di processi o thread concorrenti
- **Struttura del sistema operativo**
  - molte funzioni del sistema operativo possono essere implementate come un insieme di processi o thread

Multiprocessing e multiprogramming: differenze?

Prima di iniziare lo studio della concorrenza, dobbiamo capire se esistono differenze fondamentali nella programmazione quando i processi multipli sono eseguiti da processori diversi rispetto a quando sono eseguiti dallo stesso processore

Multiprocessing e multiprogramming: differenze?

- **In un singolo processore:**
  - processi multipli sono "*alternati nel tempo*" per dare l'impressione di avere un multiprocessore
  - ad ogni istante, al massimo un processo è in esecuzione
  - si parla di *interleaving*
- **In un sistema multiprocessore:**
  - più processi vengono eseguiti *simultaneamente* su processori diversi
  - i processi sono "*alternati nello spazio*"
  - si parla di *overlapping*

Multiprocessing e multiprogramming: differenze?

- **A prima vista:**
  - si potrebbe pensare che queste differenze comportino problemi distinti
  - in un caso l'esecuzione è simultanea
  - nell'altro caso la simultaneità è solo simulata
- **In realtà:**
  - presentano gli stessi problemi
  - che si possono riassumere nel seguente:

non è possibile predire la velocità relativa dei processi

Un esempio semplice

- **Si consideri il codice seguente:**

In C:

```
void modifica(int valore) {
  totale = totale + valore
}
```

In Assembly:

```
.text
modifica:
  lw $t0, totale
  add $t0, $t0, $a0
  sw $t0, totale
  jr $ra
```

- **Supponiamo che:**
  - Esista un processo P_1 che esegue `modifica(+10)`
  - Esista un processo P_2 che esegue `modifica(-10)`
  - P_1 e P_2 siano in esecuzione concorrente
  - `totale` sia una variabile condivisa tra i due processi, con valore iniziale 100
- **Alla fine, `totale` dovrebbe essere uguale a 100. Giusto?**

Scenario 1: multiprogramming (corretto)

P1	`lw $t0, totale`	totale=100, $t0=100, $a0=10
P1	`add $t0, $t0, $a0`	totale=100, $t0=110, $a0=10
P1	`sw $t0, totale`	totale=110, $t0=110, $a0=10
S.O.	`interrupt`	
S.O.	`salvataggio registri P1`	
S.O.	`ripristino registri P2`	totale=110, $t0=? , $a0=-10
P2	`lw $t0, totale`	totale=110, $t0=110, $a0=-10
P2	`add $t0, $t0, $a0`	totale=110, $t0=100, $a0=-10
P2	`sw $t0, totale`	totale=100, $t0=100, $a0=-10

Scenario 2: multiprogramming (errato)

P1	`lw $t0, totale`	totale=100, $t0=100, $a0=10
S.O.	`interrupt`	
S.O.	`salvataggio registri P1`	
S.O.	`ripristino registri P2`	totale=100, $t0=? , $a0=-10
P2	`lw $t0, totale`	totale=100, $t0=100, $a0=-10
P2	`add $t0, $t0, $a0`	totale=100, $t0= 90, $a0=-10
P2	`sw $t0, totale`	totale= 90, $t0= 90, $a0=-10
S.O.	`interrupt`	
S.O.	`salvataggio registri P2`	
S.O.	`ripristino registri P1`	totale= 90, $t0=100, $a0=10
P1	`add $t0, $t0, $a0`	totale= 90, $t0=110, $a0=10
P1	`sw $t0, totale`	totale=110, $t0=110, $a0=10

Scenario 3: multiprocessing (errato)

- I due processi vengono eseguiti simultaneamente da due processori distinti

Processo P1:

```
lw $t0, totale
add $t0, $t0, $a0
sw $t0, totale
```

Processo P2:

```
lw $t0, totale
add $t0, $t0, $a0
sw $t0, totale
```

- **Nota:**
  - i due processi hanno insiemi di registri distinti
  - l'accesso alla memoria su **`totale`** non può essere simultaneo

Alcune considerazioni

- **Non vi è sostanziale differenza tra i problemi relativi a multiprogramming e multiprocessing**
  - ai fini del ragionamento sui programmi concorrenti si ipotizza che sia presente un "processore ideale" per ogni processo
- **I problemi derivano dal fatto che:**
  - non è possibile predire gli istanti temporali in cui vengono eseguite le istruzioni
  - i due processi accedono ad una o più risorse condivise

Race condition

- **Definizione**
  - Si dice che un sistema di processi multipli presenta una *race condition* qualora il risultato finale dell'esecuzione dipenda dalla temporizzazione con cui vengono eseguiti i processi
- **Per scrivere un programma concorrente:**
  - è necessario eliminare le race condition

Considerazioni finali

- **In pratica:**
  - scrivere programmi concorrenti è più difficile che scrivere programmi sequenziali
  - la correttezza non è solamente determinata dall'esattezza dei passi svolti da ogni singola componente del programma, ma anche dalle interazioni (volute o no) tra essi
- **Nota:**
  - Fare debug di applicazioni che presentano race condition non è per niente piacevole...
  - Il programma può funzionare nel 99.999% dei casi durante i test, e bloccarsi inesorabilmente quando lo dimostrate per la vendita
  - (... un corollario alla legge di Murphy...)

Notazioni per descrivere processi concorrenti - 1

- **Notazione esplicita**

```
process nome {
   ... statement(s) ...
}
```

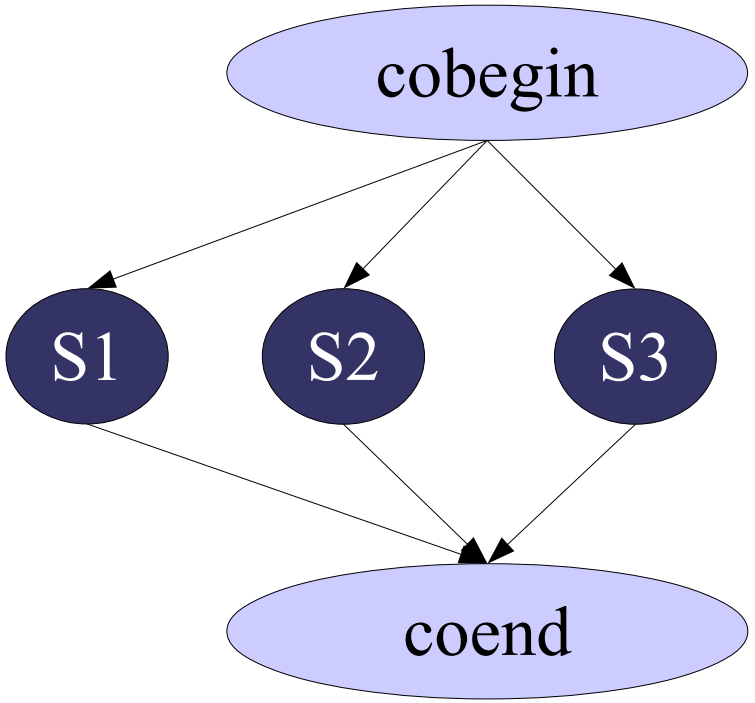
- **Esempio**

```
process P1 {
   totale = totale + valore;
}
process P2 {
   totale = totale - valore;
}
```

Notazioni per descrivere processi concorrenti - 2

- **Notazione cobegin/coend**

```
cobegin
   ... S1 ...
//
   ... S2 ...
//
   ... S3 ...
//
   ...
coend
```



- **Ogni statement viene eseguito in concorrenza**
- **Le istruzioni che seguono il coend verranno eseguite solo quando tutti gli statement sono terminati**

Esempio - Mergesort

- **Supponiamo di aver a disposizione le seguenti funzioni**
  - `sort(v,i,j)`
    - ordina gli elementi del vettore **v** dall'indice **i** all'indice **j**
  - `merge(v,i,j,k)`
    - dati due segmenti già ordinati del vettore **v** (il segmento dall'indice **i** all'indice **j** e quello dall'indice **j+1** all'indice **k**) effettua il merge rendendo ordinato il segmento dall'indice **i** all'indice **k**
- **Scriviamo una versione di mergesort parallelo**
  - vediamo come sia semplice commettere errori

Mergesort - errato

```
sort(v, len) {
  m = len/2
  cobegin
    sort(v, 1 , m);
  //
    sort(v, m+1, len);
  //
    merge(v, 1, m, len);
  coend
}
```

Mergesort - corretto

```
sort(v, len) {
  m = len/2
  cobegin
    sort(v, 1 , m);
  //
    sort(v, m+1, len);
  //
  coend
  merge(v, 1, m, len);
}
```

Sezione 2

2. Interazioni tra processi

Interazioni tra processi

- **E' possibile classificare le modalità di interazione tra processi in base a quanto sono "*consapevoli*" uno dell'altro.**
- **Processi totalmente "ignari" uno dell'altro:**
  - processi indipendenti non progettati per lavorare insieme
  - sebbene siano indipendenti, vivono in un ambiente comune
- **Come interagiscono?**
  - *competono* per le stesse risorse
  - devono *sincronizzarsi* nella loro utilizzazione
- **Il sistema operativo:**
  - deve arbitrare questa *competizione*, fornendo meccanismi di *sincronizzazione*

Interazioni tra processi

- **Processi "indirettamente" a conoscenza uno dell'altro**
  - processi che condividono risorse, come ad esempio un buffer, al fine di scambiarsi informazioni
  - non si conoscono in base ai loro id, ma interagiscono indirettamente tramite le risorse condivise
- **Come interagiscono?**
  - *cooperano* per qualche scopo
  - devono *sincronizzarsi* nella utilizzazione delle risorse
- **Il sistema operativo:**
  - deve facilitare la *cooperazione*, fornendo meccanismi di *sincronizzazione*

Interazioni tra processi

- **Processi "direttamente" a conoscenza uno dell'altro**
  - processi che comunicano uno con l'altro sulla base dei loro id
  - la comunicazione è diretta, spesso basata sullo scambio di messaggi
- **Come interagiscono**
  - *cooperano* per qualche scopo
  - *comunicano* informazioni agli altri processi
- **Il sistema operativo:**
  - deve facilitare la *cooperazione*, fornendo meccanismi di *comunicazione*

Proprietà

- **Definizione**
  - Una *proprietà* di un programma concorrente è un attributo che rimane vero per ogni possibile storia di esecuzione del programma stesso
- **Due tipi di proprietà:**
  - *Safety* ("nothing bad happens")
    - mostrano che il programma (se avanza) va "nella direzione voluta", cioè non esegue azioni scorrette
  - *Liveness* ("something good eventually happens")
    - il programma avanza, non si ferma... insomma è "*vivo*"

Proprietà - Esempio

- ***Consensus*, dalla teoria dei sistemi distribuiti**
  - Si consideri un sistema con **N** processi:
    - All'inizio, ogni processo *propone* un valore
    - Alla fine, tutti i processi si devono accordare su uno dei valori proposti (*decidono* quel valore)
- **Proprietà di safety**
  - Se un processo decide, deve decidere uno dei valori proposti
  - Se due processi decidono, devono decidere lo stesso valore
- **Proprietà di liveness**
  - Prima o poi ogni processo corretto (i.e. non in crash) prenderà una decisione

Proprietà - programmi sequenziali

- **Nei programmi sequenziali:**
  - le proprietà di *safety* esprimono la correttezza dello stato finale (il risultato è quello voluto)
  - la principale proprietà di *liveness* è la terminazione

Proprietà - programmi concorrenti

- **Proprietà di *safety***
  - i processi non devono "*interferire*" fra di loro nell'accesso alle risorse condivise
  - questo vale ovviamente per i processi che condividono risorse (non per processi che cooperano tramite comunicazione)
- **I meccanismi di sincronizzazione servono a garantire la proprietà di safety**
  - devono essere usati propriamente dal programmatore, altrimenti il programma potrà contenere delle race conditions

Proprietà - programmi concorrenti

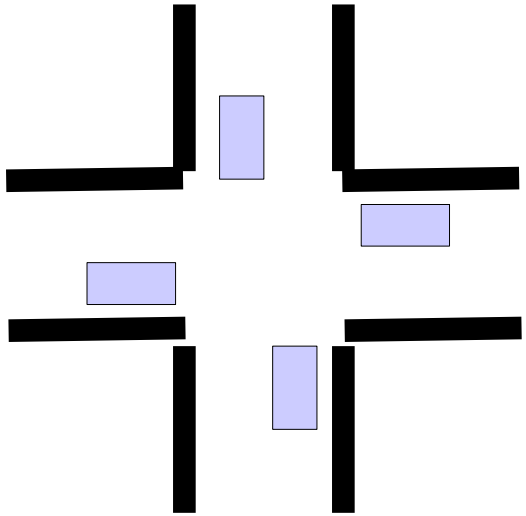
- **Proprietà di *liveness***
  - i meccanismi di sincronizzazione utilizzati non devono prevenire l'avanzamento del programma
    - non è possibile che *tutti* i processi si "*blocchino*", in attesa di eventi che non possono verificarsi perché generabili solo da altri processi bloccati
    - non è possibile che *un* processo debba "*attendere indefinitamente*" prima di poter accedere ad una risorsa condivisa
- **Nota:**
  - queste sono solo descrizioni informali; nei prossimi lucidi saremo più precisi

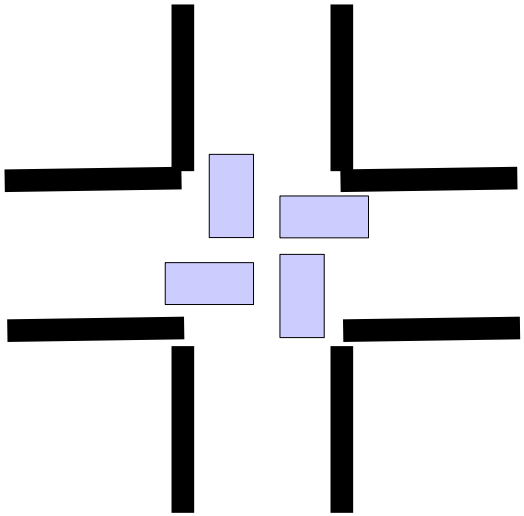
Mutua esclusione (safety)

- **Definizione**
  - l'accesso ad una risorsa si dice *mutualmente esclusivo* se ad ogni istante, al massimo un processo può accedere a quella risorsa
- **Esempi da considerare:**
  - due processi che vogliono accedere contemporaneamente a una stampante
  - due processi che cooperano scambiandosi informazioni tramite un buffer condiviso

Deadlock (stallo) (liveness)

- **Considerazioni:**
  - la mutua esclusione permette di risolvere il problema della non interferenza
  - ma può causare il blocco permanente dei processi
- **Esempio: incrocio stradale**





Deadlock (stallo)

- **Esempio:**
  - siano $\mathbf{R_1}$ e $\mathbf{R_2}$ due risorse
  - siano $\mathbf{P_1}$ e $\mathbf{P_2}$ due processi che devono accedere a $\mathbf{R_1}$ e $\mathbf{R_2}$ contemporaneamente, prima di poter terminare il programma
  - supponiamo che il S.O. assegni $\mathbf{R_1}$ a $\mathbf{P_1}$, e $\mathbf{R_2}$ a $\mathbf{P_2}$
  - i due processi sono bloccati in attesa circolare
- **Si dice che P_1 e P_2 sono in deadlock**
  - è una condizione da evitare
  - è definitiva
  - nei sistemi reali, se ne può uscire solo con metodi "distruttivi", ovvero uccidendo i processi, riavviando la macchina, etc.

Starvation (inedia) (liveness)

- **Considerazioni:**
  - il deadlock è un problema che coinvolge tutti i processi che utilizzano un certo insieme di risorse
  - esiste anche la possibilità che un processo non possa accedere ad un risorsa perché "sempre occupata"
- **Esempio**
  - se siete in coda ad uno sportello e continuano ad arrivare "furbi" che passano davanti, non riuscirete mai a parlare con l'impiegato/a

Starvation (inedia)

- **Esempio**
  - sia **R** una risorsa
  - siano $\mathbf{P_1}$, $\mathbf{P_2}$, $\mathbf{P_3}$ tre processi che accedono periodicamente a **R**
  - supponiamo che $\mathbf{P_1}$ e $\mathbf{P_2}$ si alternino nell'uso della risorsa
  - $\mathbf{P_3}$ non può accedere alla risorsa, perché utilizzata in modo esclusivo da $\mathbf{P_1}$ e $\mathbf{P_2}$
- **Si dice che P3 è in *starvation***
  - a differenza del deadlock, non è una condizione definitiva
  - è possibile uscirne, basta adottare un'opportuna politica di assegnamento
  - è comunque una situazione da evitare

Riassunto

Tipo	Relazione	Meccanismo	Problemi di controllo
processi "ignari" uno dell'altro	competizione	sincronizzazione	mutua esclusione deadlock starvation
processi con conoscenza indiretta l'uno dell'altro	cooperazione (sharing)	sincronizzazione	mutua esclusione deadlock starvation
processi con conoscenza diretta l'uno dell'altro	cooperazione (comunicazione)	comunicazione	deadlock starvation

Riassunto

- **Nei prossimi lucidi:**
  - vedremo quali tecniche possono essere utilizzate per garantire mutua esclusione e assenza di deadlock e starvation
  - prima però vediamo di capire esattamente quando due o più processi possono interferire

Azioni atomiche

- **Definizione**
  - le azioni atomiche vengono compiute in modo indivisibile
  - soddisfano la condizione: *o tutto o niente*
- **Nel caso di parallelismo reale:**
  - si garantisce che l'azione non interferisca con altri processi durante la sua esecuzione
- **Nel caso di parallelismo apparante**
  - l'avvicendamento (*context switch*) fra i processi avviene *prima* o *dopo* l'azione, che quindi non può interferire

Azioni atomiche - Esempi

- **Le singole istruzioni del linguaggio macchina sono atomiche**
- **Esempio:** `sw $a0, ($t0)`
- **Nel caso di parallelismo apparente:**
  - il meccanismo degli interrupt (su cui è basato l'avvicendamento dei processi) garantisce che un interrupt venga eseguito prima o dopo un'istruzione, mai "durante"
- **Nel caso di parallelismo reale:**
  - anche se più istruzioni cercano di accedere alla stessa cella di memoria (quella puntata da `$t0`), la *politica di arbitraggio del bus* garantisce che una delle due venga servita per prima e l'altra successivamente

Azioni atomiche - Controesempi

- **In generale, sequenze di istruzioni in linguaggio macchina non sono azioni atomiche**
- **Esempio:**

```
lw $t0, ($a0)
add $t0, $t0, $a1
sw $t0, ($a0)
```

- **Attenzione:**
  - le singole istruzioni in linguaggio macchina sono atomiche
  - le singole istruzioni in assembly possono non essere atomiche
  - esistono le pseudoistruzioni!

Azioni atomiche

- **E nel linguaggio C?**
  - Dipende dal processore
  - Dipende dal codice generato dal compilatore
- **Esempi**
  - `a=0;  /* int a */`

    questo statement è atomico; la variabile a viene definita come un intero di lunghezza "nativa" e inizializzata a 0
  - `a=0; /* long long a */`

    questo statement non è atomico, in quanto si tratta di porre a zero una variabile a 64 bit; questo può richiedere più istruzioni
  - `a++;`

    anche questo statement in generale non è atomico, ma dipende dalle istruzioni disponibili in linguaggio macchina

Azioni atomiche

- **E nei compiti di concorrenza?**
  - Assumiamo che in ogni istante, vi possa essere al massimo un accesso alla memoria alla volta
  - Questo significa che operazioni tipo:
    - aggiornamento di una variabile
    - incremento di una variabile
    - valutazione di espressioni
    - etc.

    non sono atomiche
  - Operazioni tipo:
    - assegnamento di un valore costante ad una variabile

    sono atomiche

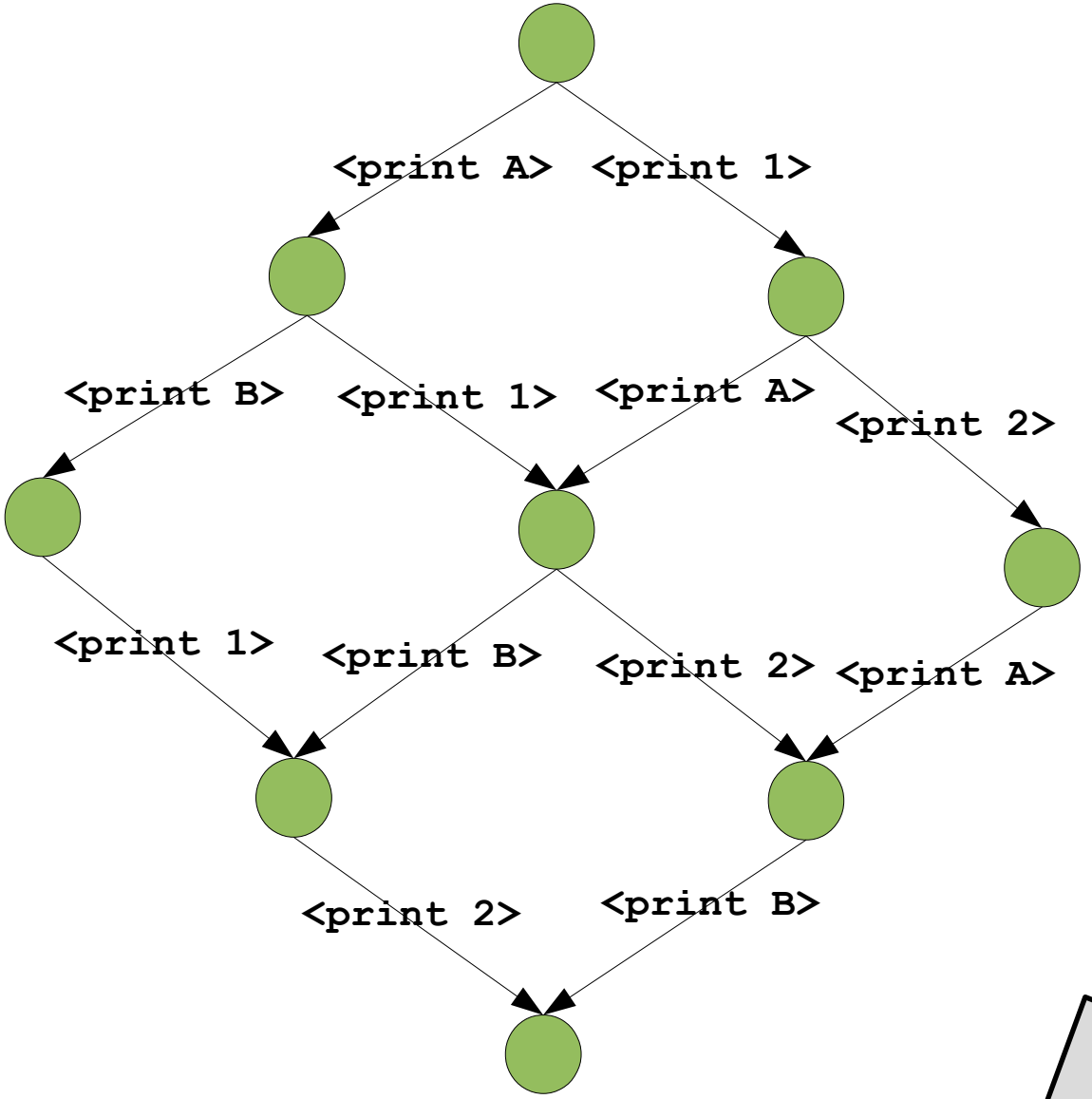
Azioni atomiche

- **Una notazione per le operazioni atomiche**
  - Nel seguito, utilizzeremo la notazione <*S*> per indicare che lo statement S deve essere eseguito in modo atomico
  - Esempio:
    - `< x = x + 1; >`

Interleaving di azioni atomiche

- **Cosa stampa questo programma?**

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



3. Sezioni critiche

"Non-interferenza"

- **Problema**
  - Se le sequenze di istruzioni non vengono eseguite in modo atomico, come possiamo garantire la non-interferenza?
- **Idea generale**
  - Dobbiamo trovare il modo di specificare che certe parti dei programmi sono "speciali", ovvero devono essere eseguite in modo atomico (senza interruzioni)

Sezioni critiche

- **Definizione**
  - La parte di un programma che utilizza una o più risorse condivise viene detta *sezione critica (critical section, o CS)*
- **Esempio**

```
process P1
   a1 = rand();
   totale = totale + a1;
}
```

```
process P2 {
   a2 = rand();
   totale = totale + a2;
}
```

- **Spiegazione:**
  - La parte evidenziata è una sezione critica, in quanto accede alla risorsa condivisa **totale**; mentre **a1** e **a2** non sono condivise

Sezioni condivise

- **Obiettivi**
  - Vogliamo garantire che le sezioni critiche siano eseguite in modo mutualmente esclusivo (atomico)
  - Vogliamo evitare situazioni di blocco, sia dovute a deadlock sia dovute a starvation

Sezioni critiche

- **Sintassi:**
  - `[enter cs]` indica il punto di inizio di una sezione critica
  - `[exit cs]` indica il punto di fine di una sezione critica
- **Esempio:**

```
x:=0
cobegin
   [enter cs]; x = x+1; [exit cs];
//
   [enter cs]; x = x-1; [exit cs];
coend
```

Sezioni critiche

- **Esempio:**

```
cobegin
   val = rand();
   a = a + val;
   b = b + val
//
   val = rand();
   a = a * val;
   b = b * val;
coend
```

- **Perchè abbiamo bisogno di costrutti specifici?**
  - Perchè il S.O. non può capire da solo cosa è una sezione critica e cosa non lo è
- **In questo programma:**
  - Vorremmo garantire che **a** sia sempre uguale a **b** (*invariante*)
- **Soluzione 1:**
  - Lasciamo fare al sistema operativo...
  - Ma il S.O. non conosce il vincolo dell'invarianza
  - L'unica soluzione possibile per il S.O. è non eseguire le due parti in parallelo
  - Ma così perdiamo i vantaggi!

Sezioni critiche

- **Esempio:**

```
cobegin
   val = rand();
   [enter cs]
   a = a + val;
   b = b + val
   [exit cs]
//
   val = rand();
   [enter cs]
   a = a * val;
   b = b * val;
   [exit cs]
coend
```

- **In questo programma:**
  - Vorremmo garantire che **a** sia sempre uguale a **b** (*invariante*)
- **Soluzione 2:**
  - Indichiamo al S.O. cosa può essere eseguito in parallelo
  - Indichiamo al S.O. cosa deve essere eseguito in modo atomico, altrimenti non avremo consistenza

Sezioni critiche

- **Problema della CS**
  - Si tratta di realizzare **N** processi della forma

```
process Pi {   /* i=1...N */
  while (true) {
    [enter cs]
    critical section
    [exit cs]
    non-critical section
  }
}
```

in modo che valgano le seguenti proprietà:

Sezioni critiche

- **Requisiti per le CS**

  1) *Mutua esclusione*
     - Solo un processo alla volta deve essere all'interno della CS, fra tutti quelli che hanno una CS per la stessa risorsa condivisa

  2) *Assenza di deadlock*
     - Uno scenario in cui tutti i processi restano bloccati definitivamente non è ammissibile

  3) *Assenza di delay non necessari*
     - Un processo fuori dalla CS non deve ritardare l'ingresso della CS da parte di un altro processo

  4) *Eventual entry (assenza di starvation)*
     - Ogni processo che lo richiede, prima o poi entra nella CS

Sezioni critiche

- **Perché il problema delle CS è espresso in questa forma?**
  - Perché descrive in modo generale un insieme di processi, ognuno dei quali può ripetutamente entrare e uscire da una sezione critica
- **Dobbiamo fare un'assunzione:**
  - Se un processo entra in una critical section, prima o poi ne uscirà
  - Ovvero, un processo può terminare solo fuori dalla sua sezione critica

Sezioni critiche - Possibili approcci

- **Approcci software**
  - la responsabilità cade sui processi che vogliono accedere ad un oggetto distribuito
  - problemi
    - soggetto ad errori!
    - vedremo che è costoso in termini di esecuzione (busy waiting)
  - interessante dal punto di vista didattico
- **Approcci hardware**
  - Uso di istruzioni speciali del linguaggio macchina, progettate apposta
  - efficienti
  - problemi
    - non sono adatte come soluzioni general-purpose

Sezioni critiche - Possibili approcci

- **Approcci basati su supporto nel S.O. o nel linguaggio**
  - la responsabilità di garantire la mutua esclusione ricade sul S.O. o sul linguaggio (e.g. Java)
- **Esempi**
  - Semafori
  - Monitor
  - Message passing

Algoritmo di Dekker

- **Dijkstra (1965)**
  - Riporta un algoritmo per la mutua esclusione
  - Progettato dal matematico olandese *Dekker*
  - Nell'articolo, la soluzione viene sviluppata in fasi
  - Seguiremo anche noi questo approccio

Tentativo 1

```
shared int turn = P; cobegin P // Q coend
process P {
  while (true) {
    /* entry protocol */
    while (turn == Q)
      ; /* do nothing */
    critical section
    turn = Q;
    non-critical section
  }
}
```

```
process Q {
  while (true) {
    /* entry protocol */
    while (turn == P)
      ; /* do nothing */
    critical section
    turn = P;
    non-critical section
  }
}
```

- **Note**
  - la variabile **turn** è condivisa
  - può essere acceduta solo da un processo alla volta (in lettura o scrittura)
  - il controllo iterativo su una condizione di accesso viene detto *busy waiting*

Tentativo 1

- **La soluzione proposta è corretta?**
- **Problema:**
  - Non rispetta il requisito 3: assenza di delay non necessari
    - "Un processo fuori dalla CS non deve ritardare l'ingresso nella CS da parte di un altro processo"

Tentativo 1 - Problema

- **Si consideri questa esecuzione:**
  - **P** entra nella sezione critica
  - **P** esce dalla sezione critica
  - **P** cerca di entrare nella sezione critica
  - **Q** è molto lento; fino a quando **Q** non entra/esce dalla CS, **P** non può entrare

Tentativo 2

```
shared boolean inp = false; shared boolean inq = false;
cobegin P // Q coend
process P {                                process Q {
  while (true) {                             while (true) {
    /* entry protocol */                       /* entry protocol */
    while (inq)                                while (inp)
      ; /* do nothing */                          ; /* do nothing */
    inp = true;                                inq = true;
    critical section                           critical section
    inp = false;                               inq = false;
    non-critical section                       non-critical section
  }                                          }
}                                          }
```

- **Note**
  - ogni processo è associato ad un flag
  - ogni processo può esaminare il flag dell'altro, ma non può modificarlo

Tentativo 2

- **La soluzione proposta è corretta?**
- **Problema:**
  - Non rispetta il requisito 1: *mutua esclusione*
    - " solo un processo alla volta deve essere all'interno della CS "

Tentativo 2 - Problema

- **Si consideri questa esecuzione:**
  - **P** attende fino a quando **`inq=false`**; vero dall'inizio, passa
  - **Q** attende fino a quando **`inp=false`**; vero dall'inizio, passa
  - **P** **`inp = true`**;
  - **P** entra nella critical section
  - **Q** **`inq = true`**;
  - **Q** entra nella critical section

Tentativo 3

```
shared boolean inp = false; shared boolean inq = false;
cobegin P // Q coend
process P {
  while (true) {
    /* entry protocol */
    inp = true;
    while (inq)
      ; /* do nothing */
    critical section
    inp = false;
    non-critical section
  }
}
```

```
process Q {
  while (true) {
    /* entry protocol */
    inq = true;
    while (inp)
      ; /* do nothing */
    critical section
    inq = false;
    non-critical section
  }
}
```

- **Note**
  - Nel tentativo precedente, il problema stava nel fatto che era possibile che un context switch occorresse tra il controllo sul flag dell'altro processo e la modifica del proprio. Abbiamo trovato una soluzione?

Tentativo 3

- **La soluzione proposta è corretta?**
- **Problema:**
  - Non rispetta il requisito 2: *assenza di deadlock*
    - "Uno scenario in cui tutti i processi restano bloccati definitivamente non è ammissibile”

Tentativo 3 - Problema

- **Si consideri questa esecuzione:**
  - **P** `inp = true;`
  - **Q** `inq = true;`
  - **P** attende fino a quando `inq=false`; bloccato
  - **Q** attende fino a quando `inq=false`; bloccato

Tentativo 4

```
shared boolean inp = false; shared boolean inq = false;
cobegin P // Q coend
process P {
  while (true) {
    /* entry protocol */
    inp = true;
    while (inq) {
      inp = false;
      /* delay */
      inp = true;
    }
    critical section
    inp = false;
    non-critical section
  }
}
```

```
process Q {
  while (true) {
    /* entry protocol */
    inq = true;
    while (inp) {
      inq = false;
      /* delay */
      inq = true;
    }
    critical section
    inq = false;
    non-critical section
  }
}
```

Tentativo 4

- **Che sia la volta buona?**
- **Problema 1**
  - Non rispetta il requisito 4: *eventual entry*
    - " ogni processo che lo richiede, prima o poi entra nella CS "

Tentativo 4 - Problema

- **Si consideri questa esecuzione:**
  - **P** `inp = true;`
  - **Q** `inq = true;`
  - **P** verifica `inq`
  - **Q** verifica `inp`
  - **P** `inp = false;`
  - **Q** `inq = false;`
- **Note**
  - questa situazione viene detta "*livelock*", o situazione di "*mutua cortesia*"
  - difficilmente viene sostenuta a lungo, però è da evitare...
  - ... anche per l'uso dell'attesa come meccanismo di sincronizzazione

Riassumendo - una galleria di {e|o}rrori

- **Tentativo 1**
  - L'uso dei turni permette di evitare problemi di deadlock e mutua esclusione, ma non va bene in generale
- **Tentativo 2**
  - "verifica di una variabile + aggiornamento di un altra" non sono operazioni eseguite in modo atomico
- **Tentativo 3**
  - il deadlock è causato dal fatto che entrambi i processi insistono nella loro richiesta di entrare nella CS - *in modo simmetrico*
- **Tentativo 4**
  - il livelock è causato dal fatto che entrambi i processi lasciano il passo all'altro processo - *in modo simmetrico*

Riassumendo

- **Quali caratteristiche per una soluzione?**
  - il meccanismo dei turni del tentativo 1 è ideale per "rompere la simmetria" dei tentativi 3 e 4
  - il meccanismo di "prendere l'iniziativa" del tentativo 3 è ideale per superare la stretta alternanza dei turni del tentativo 1
  - il meccanismo di "lasciare il passo" del tentativo 4 è ideale per evitare situazioni di deadlock del tentativo 2

Algoritmo di Dekker

```
shared int turn = P;
shared boolean needp = false; shared boolean needq = false;
cobegin P // Q coend

process P {
  while (true) {
    /* entry protocol */
    needp = true;
    while (needq)
      if (turn == Q) {
        needp = false;
        while (turn == Q)
          ; /* do nothing */
        needp = true;
      }
    critical section
    needp = false; turn = Q;
    non-critical section
  }
}
```

```
process Q {
  while (true) {
    /* entry protocol */
    needq = true;
    while (needp)
      if (turn == P) {
        needq = false;
        while (turn == P)
          ; /* do nothing */
        needq = true;
      }
    critical section
    needq = false; turn = P;
    non-critical section
  }
}
```

Algoritmo di Dekker - Dimostrazione

- **Dimostrazione (mutua esclusione)**
  - per assurdo:
    - supponiamo che **P** e **Q** siano in CS contemporaneamente
  - poiché:
    - gli accessi in memoria sono esclusivi
    - per entrare devono almeno aggiornare / valutare entrambe le variabili **`needp`** e **`needq`**
  - → uno dei due entra per primo; diciamo sia **Q**
  - → **`needq`** sarà **`true`** fino a quando **Q** non uscirà dal ciclo
  - → poiché **P** entra nella CS mentre **Q** è nella CS, significa che esiste un istante temporale in cui **`needq = false`** e **Q** è in CS
  - → ASSURDO!

Algoritmo di Dekker - Dimostrazione

- **Dimostrazione (assenza di deadlock)**
  - per assurdo
    - supponiamo che né **P** ne **Q** possano entrare in CS
  - → **P** e **Q** devono essere bloccati nel primo **`while`**
  - → esiste un istante **t** dopo di che **`needp`** e **`needq`** sono sempre **`true`**
  - supponiamo (senza perdita di gen.) che all'istante **t**, **`turn`** = **Q**
  - l'unica modifica a **`turn`** può avvenire solo quando **Q** entra in CS
  - → dopo **t**, **`turn`** resterà sempre uguale a **Q**
  - **P** entra nel primo ciclo, e mette **`needp`** = **`false`**
  - ASSURDO!

Algoritmo di Dekker - Dimostrazione

- **Dimostrazione (assenza di ritardi non necessari)**
  - se **Q** sta eseguendo codice non critico, allora **`needq = false`**
  - allora **P** può entrare nella CS
- **Dimostrazione (assenza di starvation)**
  - se **Q** richiede di accedere alla CS
    - **`needq = true`**
  - se **P** sta eseguendo codice non critico:
    - **Q** entra
  - se **P** sta eseguendo il resto del codice (CS, entrata, uscita)
    - prima o poi ne uscirà e metterà il turno a **Q**
    - **Q** potrà quindi entrare

Algoritmo di Peterson

- **Peterson (1981)**
  - più semplice e lineare di quello di Dijkstra / Dekker
  - più facilmente generalizzabile al caso di processi multipli

Algoritmo di Peterson

```
shared boolean needp = false;
shared boolean needq = false;
shared int turn;

cobegin P // Q coend
process P {
  while (true) {
    /* entry protocol */
    needp = true;
    turn = Q;
    while (needq && turn != P)
      ; /* do nothing */
    critical section
    needp = false;
    non-critical section
  }
}
```

```
process Q {
  while (true) {
    /* entry protocol */
    needq = true;
    turn = P;
    while (needp && turn != Q)
      ; /* do nothing */
    critical section
    needq = false;
    non-critical section
  }
}
```

Algoritmo di Peterson - Dimostrazione

- **Dimostrazione (mutua esclusione)**
  - supponiamo che P sia entrato nella sezione critica
  - vogliamo provare che Q non può entrare
  - sappiamo che needP == true
  - Q entra solo se turn = Q quando esegue il while
  - si consideri lo stato al momento in cui P entra nella critical section
    - due possibilità: needq == false or turn == P
    - se needq == false, Q doveva ancora eseguire needq == true, e quindi lo eseguirà dopo l'ingresso di P e porrà turn=P, precludendosi la possibilità di entrare
    - se turn==P, come sopra;

Algoritmo di Peterson - Dimostrazione

- **Dimostrazione (assenza di deadlock)**
  - supponiamo che per assurdo che **P** voglia entrare nella CS e sia bloccato nel suo ciclo **`while`**
  - questo significa che:
    - **`needp`** = **`true`**, **`needq`** = **`true`**, **`turn`** = **`Q`** per sempre
  - possono darsi tre casi:
    - **Q** non vuole entrare in CS
      - impossibile, visto che **`needq`** = **`true`**
    - **Q** è bloccato nel suo ciclo while
      - impossibile, visto che **`turn`** = **`Q`**
    - **Q** è nella sua CS e ne esce (prima o poi)
      - impossibile, visto che prima o poi **`needq`** assumerebbe il valore **`false`**

Algoritmo di Peterson - Dimostrazione

- **Dimostrazione (assenza di ritardi non necessari)**
  - se **Q** sta eseguendo codice non critico, allora `needq = false`
  - allora **P** può entrare nella CS
- **Dimostrazione (assenza di starvation)**
  - simile alla dimostrazione di assenza di deadlock
  - aggiungiamo un caso in fondo:
    - **Q** continua ad entrare ed uscire dalla sua CS, prevenendo l'ingresso di **P**
    - impossibile poiché
      - quando **Q** prova ad entrare nella CS pone `turn = P`
      - poiché `needp = true`
      - quindi **Q** deve attendere che **P** entri nella CS

Riassumendo...

- **Le soluzioni software**
  - permettono di risolvere il problema delle critical section
- **Problemi**
  - sono tutte basate su busy waiting
  - busy waiting spreca il tempo del processore
  - è una tecnica che non dovrebbe essere utilizzata!

Soluzioni Hardware

- **E se modificassimo l'hardware?**
  - le soluzioni di Dekker e Peterson prevedono come uniche istruzioni atomiche le operazioni di Load e Store
  - si può pensare di fornire alcune istruzioni hardware speciali per semplificare la realizzazione di sezioni critiche

Disabilitazione degli interrupt

- **Idea**
  - nei sistemi uniprocessore, i processi concorrenti vengono "alternati" tramite il meccanismo degli interrupt
  - allora facciamone a meno!
- **Esempio:**

```
process P {
  while (true) {
    disable interrupt
    critical section
    enable interrupt
    non-critical section
  }
}
```

Disabilitazione degli interrupt

- **Problemi**
  - il S.O. deve lasciare ai processi la responsabilità di riattivare gli interrupt
    - altamente pericoloso!
  - riduce il grado di parallelismo ottenibile dal processore
- **Inoltre:**
  - non funziona su sistemi multiprocessore

Test & Set

- **Istruzioni speciali**
  - istruzioni che realizzano due azioni in modo atomico
  - esempi
    - lettura e scrittura
    - test e scrittura
- **Test & Set**
  - `TS(x,y) := < y = x ; x = 1 >`
  - spiegazione
    - ritorna in `y` il valore precedente di `x`
    - assegna 1 ad `x`

Test & Set

```
shared lock=0; cobegin P // Q coend
process P {
  int vp;
  while (true) {
    do {
      TS(lock, vp);
    } while (vp);
    critical section
    lock=0;
    non-critical section
  }
}
process Q {
  int vp;
  while (true) {
    do {
      TS(lock, vp);
    } while (vp);
    critical section
    lock=0;
    non-critical section
  }
}
```

- **Mutua esclusione**
  - entra solo chi riesce a settare per primo il lock
- **No deadlock**
  - il primo che esegue TS entra senza problemi
- **No unnecessary delay**
  - un processo fuori dalla CS non blocca gli altri
- **No starvation**
  - No, se non assumiamo qualcosa di più

Altre istruzioni possibili

- **test&set non è l'unica istruzione speciale**
- **altri esempi:**
  - fetch&set
  - compare&swap
  - etc.

Riassumendo...

- **Vantaggi delle istruzioni speciali hardware**
  - sono applicabili a qualsiasi numero di processi, sia su sistemi monoprocessore che in sistemi multiprocessori
  - semplice e facile da verificare
  - può essere utilizzato per supportare sezioni critiche multiple; ogni sezione critica può essere definita dalla propria variabile
- **Svantaggi**
  - si utilizza ancora busy-waiting
  - i problemi di starvation non sono eliminati
  - sono comunque complesse da programmare

Riassumendo...

- **Vorremmo dei paradigmi**
  - che siano implementabili facilmente
  - consentano di scrivere programmi concorrenti in modo non troppo complesso

Sezione 4

4. Semafori

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++ >
  }
}
```

Semaforo - Invariante

- **Siano**
  - $\mathbf{n_P}$ il numero di operazioni **P** completate
  - $\mathbf{n_V}$ il numero di operazioni **V** completate
  - **init** il valore iniziale del semaforo
- **Vale il seguente invariante:**
  - $\mathbf{n_P \leq n_V + init}$
- **Due casi:**
  - *eventi* (**init** = 0)
    - il numero di eventi "consegnati" deve essere *non superiore* al numero di volte che l'evento si è verificato
  - *risorse* (**init** > 0)
    - il numero di richieste soddisfatte non deve essere superiore al numero iniziale di risorse + il numero di risorse restituite

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** deve *sospendere* il processo invocante
    - l'operazione **V** deve *svegliare* uno dei processi sospesi
  - Nota:
    - in questa versione, la variabile **`val`** può assumere valori negativi

```
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>
```

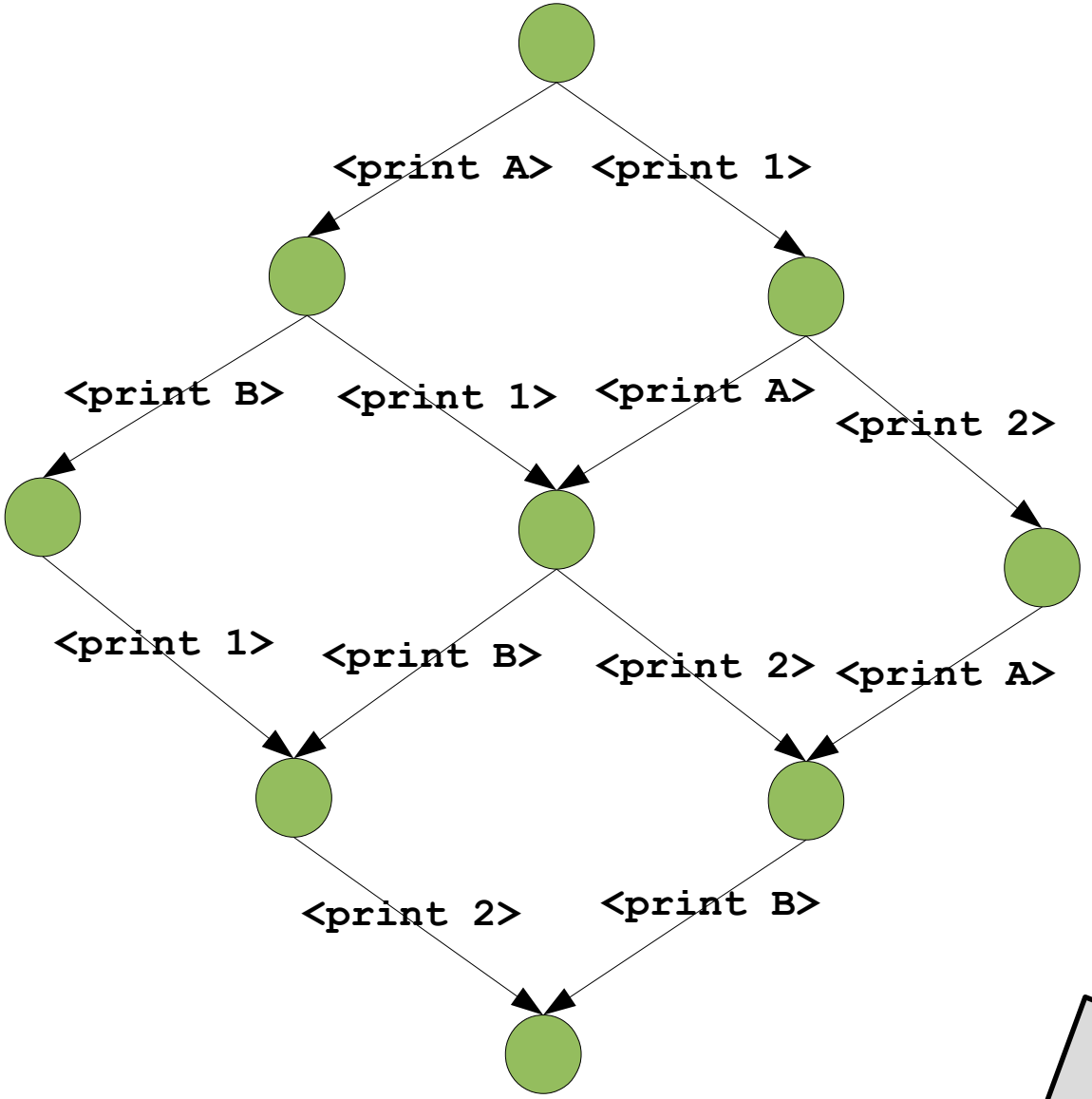
potenziale busy waiting

potenziale busy waiting

Interleaving di azioni atomiche

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

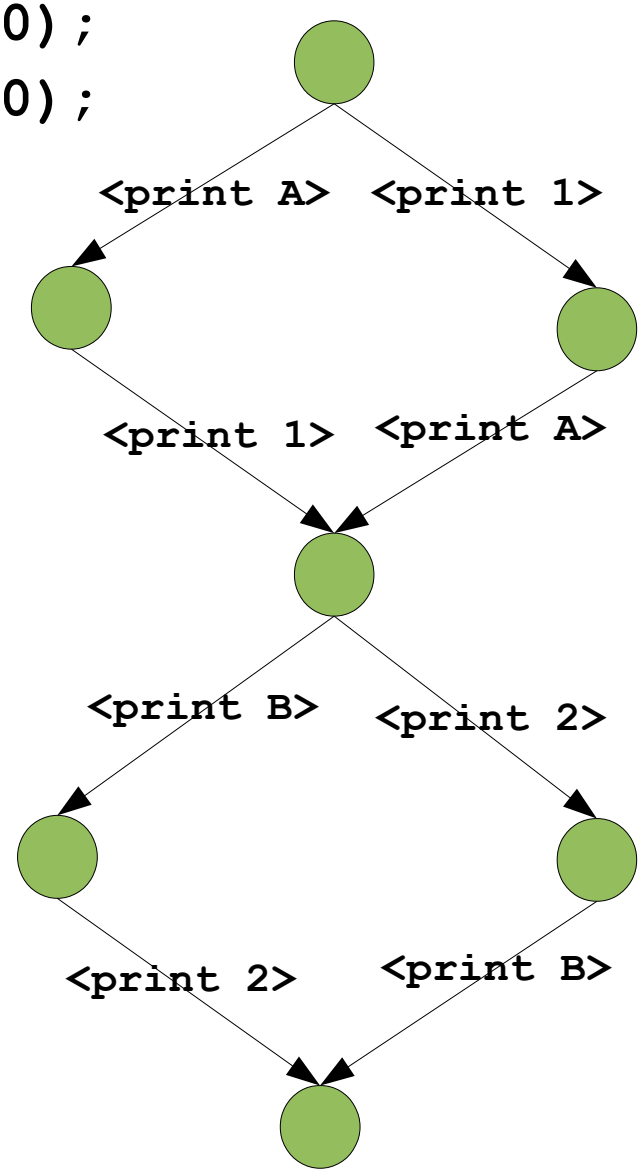
```
process P {
   <print A>
   <print B>
}
process Q {
   <print 1>
   <print 2>
}
```



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
```

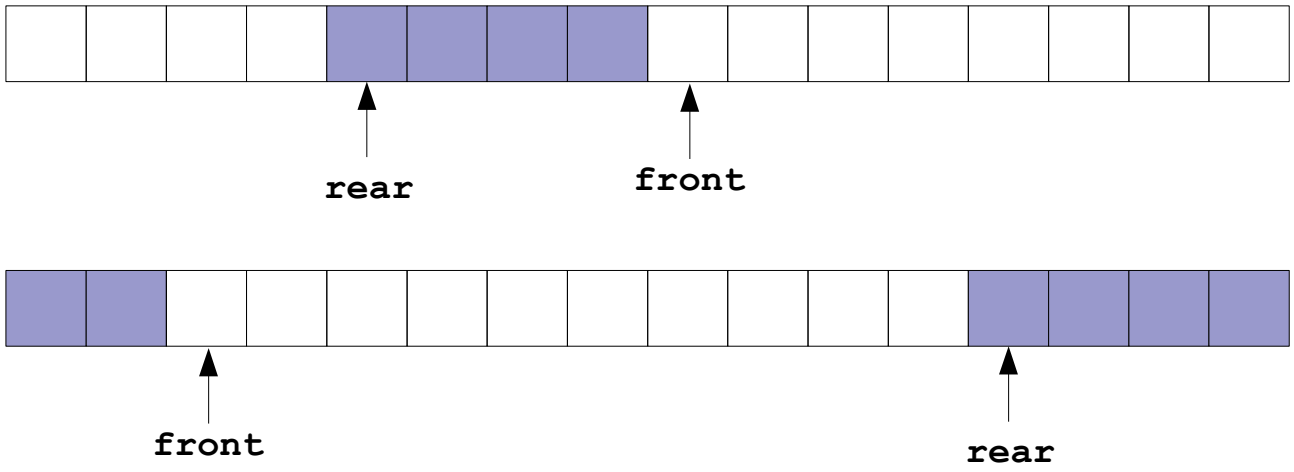
```
process Producer {
   while (true) {
      Object val = produce();
      empty.P();
      buffer = val;
      full.V();
   }
}

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?

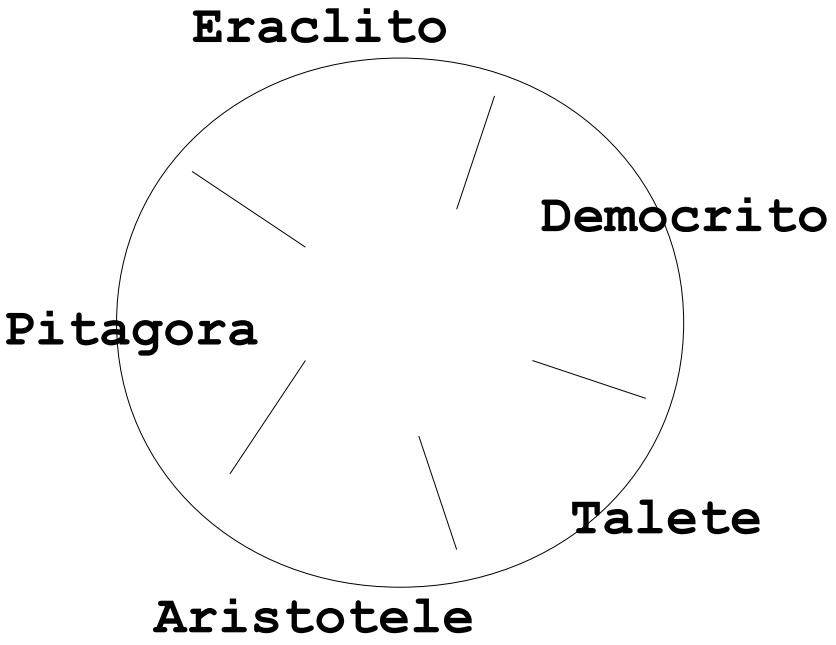
Buffer limitato - Produttori/Consumatori multipli

```
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
  - qualcuno dovrebbe spiegare a Holt come si fa a mangiare gli spaghetti con una sola forchetta
- **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];`

Invarianti

- **Definizioni**
  - up_i il numero di volte che la bacchetta `i` viene presa dal tavolo
  - $down_i$ il numero di volte che la bacchetta `i` viene rilasciata sul tavolo
- **Invariante**
  - $down_i \leq up_i \leq down_i + 1$
- **Per comodità:**
  - si può definire `chopstick[i] = 1 - (up`$_i$`-down`$_i$`)`
    (può essere pensato come un semaforo binario)

Cena dei Filosofi - Soluzione errata

```
Semaphore chopstick =
  { 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
- **Invariante**
  - sia **`nr`** il numero dei lettori che stanno accendo al database
  - sia **`nw`** il numero di scrittori che stanno accedendo al database
  - l'invariante è il seguente:

```
(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"

Sezione 5

5. Monitor

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

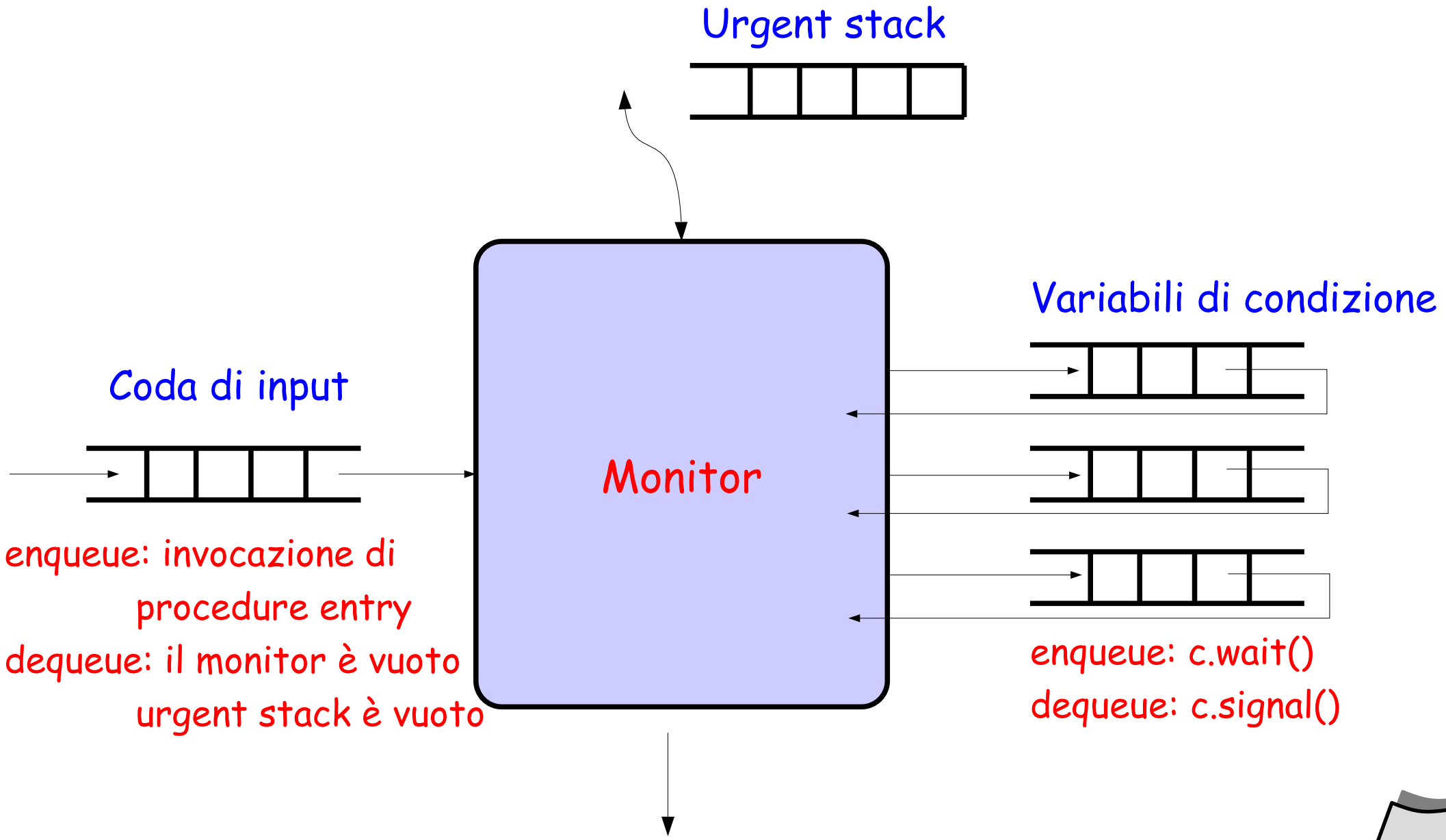
Monitor - Meccanismi di sincronizzazione

- **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: $i \pm h$ corrisponde a

$(i \pm h)\%5$

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();
  }
```

Implementazione dei monitor tramite semafori

- **Per Andrews ("Concurrent programming") è necessario utilizzare:**
  - un semaforo di mutua esclusione **e**
  - per ogni variabile di condizione **cond**$_i$, una coppia (**c**$_i$, **nc**$_i$)
    - **c**$_i$ è un semaforo correlato alla condizione, inizializzato a 0
    - **nc**$_i$ è il numero di processi che sono in attesa del verificarsi della condizione
- **Implementazione**
  - entrata nel monitor `{ e.P(); }`
  - uscita dal monitor `{ e.V(); }`
  - wait su cond$_i$ `{ nc`$_i$`++; e.V(); c`$_i$`.P(); e.P(); }`
  - signal su cond$_i$ `if (nc`$_i$ `> 0) { nc`$_i$`--; c`$_i$`.V(); }`

Implementazione dei monitor tramite semafori

- **Implementazione precedente**
  - è incompleta
  - non implementa signal urgent, ma signal & continue
  - il processo riattivato con la signal viene messo in esecuzione DOPO tutti quelli in coda di ingresso

Implementazione dei monitor tramite semafori

- **Ingredienti**
  - un modulo di gestione stack (per urgent)

```
interface Stack {
  void push(Object x);
  Object pop();
  boolean empty();
}
```

  - un semaforo di mutua esclusione **e**
  - per ogni variabile di condizione **cond**$_i$, una coppia (**c**$_i$, **nc**$_i$)
    - **c**$_i$ è un semaforo correlato alla condizione, inizializzato a 0
    - **nc**$_i$ è il numero di processi che sono in attesa del verificarsi della condizione
  - un "allocatore" di semafori
    (o alternativamente un semaforo per ogni processo)

Implementazione dei monitor tramite semafori

- **Inizializzazione**

```
Semaphore e = new Semaphore(1);
Stack stack = new Stack();
```

- **Entrata nel monitor**

```
e.P();
```

- **Wait su $cond_i$**

```
nc_i++;
if (!stack.empty()) {
  Semaphore s = stack.pop();
  s.V();
} else {
  e.V();
}
c_i.P();
```

- **Signal su $cond_i$**

```
if (nc_i > 0) {
  nc_i--;
  c_i.V();
  Semaphore s =
    new Semaphore(0);
  stack.push(s);
  s.P();
  /* free(s) / garbage coll. */
}
```

- **Uscita dal monitor**

```
if (!stack.empty()) {
  Semaphore s = stack.pop();
  s.V();
} else {
  e.V();
}
```

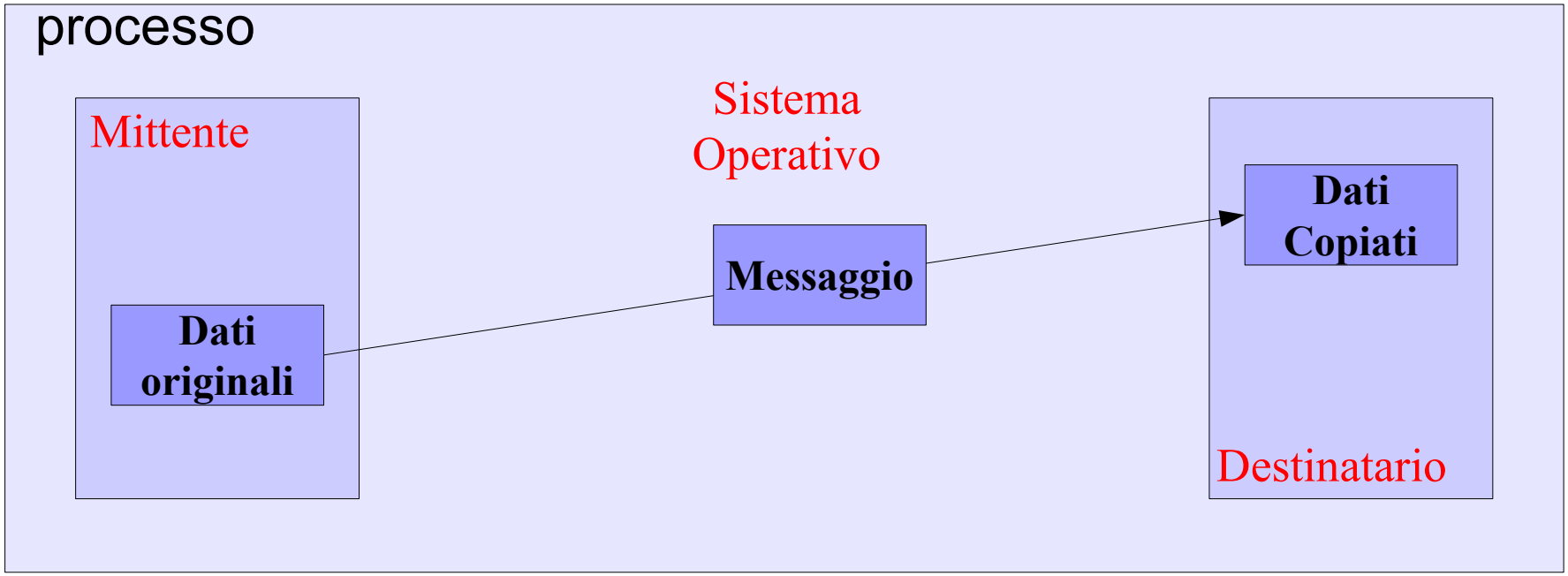
6. Message passing

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

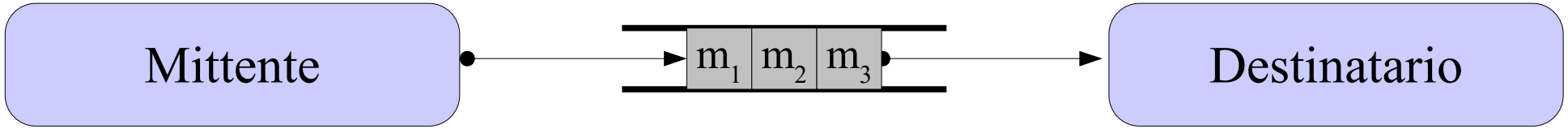
- **Operazione send asincrona**
  - sintassi: **`asend(m, q)`**
  - il mittente **`p`** spedisce il messaggio **`m`** al processo **`q`**, senza bloccarsi in attesa che il destinatario esegua l'operazione **`areceive(m, p)`**
- **Operazione receive bloccante**
  - sintassi: **`m = areceive(p)`**
  - 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

Message Passing - Note

- **In letteratura**
  - ci sono numerose diverse sintassi per descrivere message passing
  - in pratica, ogni autore se ne inventa una (anche noi!)
- **Ad esempio:**
  - invece che indicare il processo destinazione/mittente, si indica il nome di un canale
  - Message passing asincrono con 3 primitive principali: send, receive, reply (Thoth)
    - non la receive, ma solamente la reply sblocca il mittente
    - utile per rendere MP simile alle chiamate di procedura remota

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)