

Sistemi Operativi

Concorrenza

2024/25

(04.10.2024)

Renzo Davoli
Alberto Montresor

Copyright © 2002-2023 Renzo Davoli, Alberto Montresor, Claudio Sacerdoti-Coen

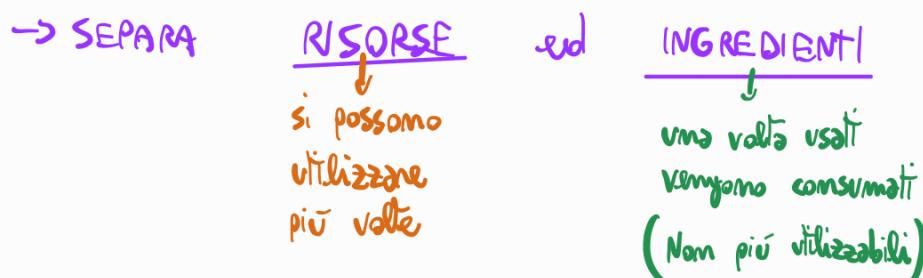
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license can be found at:
<http://www.gnu.org/licenses/fdl.html#TOC1>

Co's é um sistema operativo?

Esempi:

- LINUX
- HURD (h^+)
- XNU → sist op. APPLE
- UNIX → standard
- WINDOWS → sist. op.
- MACOS
- ANDROID
- IOS
- OPEN BSD
- MS-DOS
- TEMPLE OS
- TENEX

Il prof usa metodi interessanti e ci spiega la ricetta della PANNA MONTATA



ANALOGIE DI QUESTO ESEMPIO

- CUOCO : PROCESSORE
- RICETTA : PROGRAMMA (statico, è il testo)
- AZIONE DI PREPARARE LA PANNA : PROCESSO
- ABITO DEL CUOCO : MODALITÀ DEL PROCESSORE : mod. USER o KERNEL
- CAMERIERE : UNITÀ DI OUTPUT (es. CONTROLLER DI OUTPUT)
- ADDETTO DISPENSA : UNITÀ DI INPUT (es. CONTROLLER DI INPUT)
- INGREDIENTI : sono i DATI (es. UNITÀ che prende dati dal DISCO)
- FINESTRE : INTERFAZIE TRA PROCESSORE E INPUT/OUTPUT
- CARTELLO ALLE FINESTRE : è lo STATO DI UN DEVICE
- MANO ALZATA (dei "controller") : INTERRUPT (interruzione del processore)
- FILO DEL CUOCO : LISTA DEI PROCESSI
- GANCI DEL CUOCO : PROCESSO CORRENTE (ATTUALMENTE IN ESECUZIONE)
- FILO DEL CAMERIERE/Addetto : CODE DI ATTESA DEI DEVICE
- GANCI DEL CAMERIERE/ Addetto : PROCESSI CORRENTI DEI DEVICE
- ISTRUZIONI SUGLI ORDINI DEI CUENTI : PCB - PROCESS CONTROL BLOCK
→ STRUTTURA DATI
- LAVAGNETTA : MEMORIA DEL PROCESSO (ogni processo ha la sua lavagnetta)
- VALORE DELLA LINEA DELLA RICETTA : È IL PC - PROGRAM COUNTER

- SCELTA TRA PIÙ ORDINI DEI CLIENTI: è lo SCHEDULER (scopre quale ordine eseguire tra i più ordini)
- ISTRUZIONI DEL CAPO CUOCO K: è il SISTEMA OPERATIVO

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

↳ "che corre a fianco"

Un SISTEMA si dice CONCORRENTE quando ci sono più parti che evolvono in maniera contemporanea

CONCORRENZA → REALE: si ha quando ci sono più processi ↑
↘ APPARENTE: viene creata dal S.O. dalla gestione di più processi in maniera alternata

Introduzione

PROGRAMMAZIONE CONCURRENTE: PROGRAMMI che vengono eseguiti contemporaneamente

- 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**

perché più o meno contemporaneamente? → possiamo esserci più processori
es. **MULTICORE**

(Esecuzione REALE di più processi)
in parallelo

Introduzione

- **In questa serie di lucidi:**
 - Analizzeremo il problema della gestione di attività multiple da un punto di vista astratto → realizzeremo un modello CONCORRENTE
 - 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**

è il testo (linee di codice)
↑

- 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: → è il testo (linee di codice)
 - specifica una sequenza di istruzioni → è un **algoritmo** (codice)
 - non specifica la durata nel tempo dell'esecuzione

- Un processo: → risultato dell'esecuzione di un programma
 - rappresenta l'attività dell'esecuzione di un programma

- **Axioma di finite progress** → definizione che noi consideriamo VERA
in un certo processo

- Ogni processo viene eseguito ad una velocità finita, non nulla, ma sconosciuta

Stato di un processo

- **Ad ogni istante, un processo può essere totalmente descritto dalle seguenti componenti:**
 - *La sua immagine di memoria* → es. lavagnette (es. pamma) → stato della memoria del processo
 - 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 → es. STACK POINTER
 - *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**
un programma \Rightarrow più processi in esecuzione
 - 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

PROGRAMMA : punti

text : codice del programma che viene tradotto in codice macchina

data : contiene il valore delle variabili inizializzate

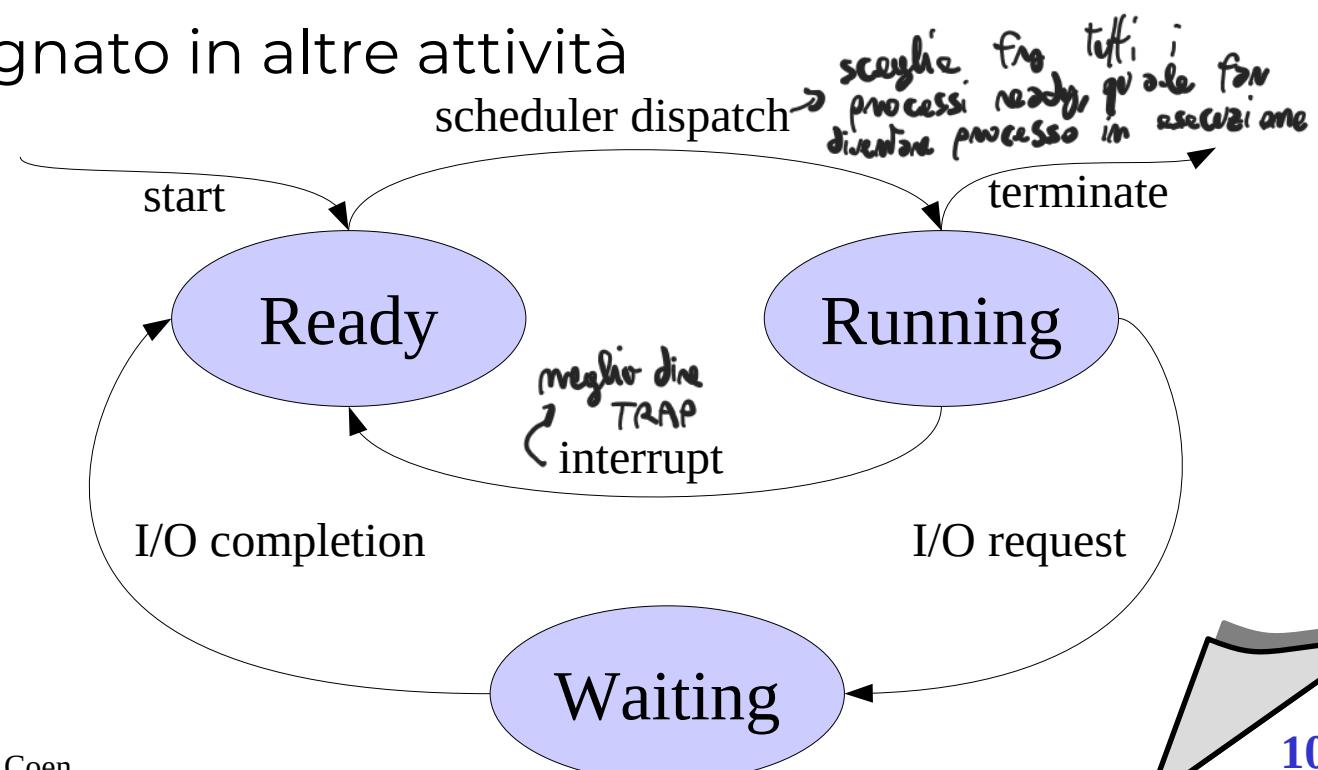
stack

devo essere private di ogni processo

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



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 Es. multicore (32-64 processori)
 - Distributed processing
 - più processi su un insieme di computer distribuiti e indipendenti
 - parallelismo reale

⇒ Si puo' combinare il Distributed processing e il multiprocessing:

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 **① comunicazione** e **② sincronizzazione**

↳ ci possono essere delle condizioni per cui un programma puo' fare qualcosa solo se un altro programma ha già fatto qualcosa' altro

blanda
Esempio: due processi in parallelo: studiare mentre ascolto musica (Esempio) In realtà qua i processi condividono risorse

Dove possiamo trovare la concorrenza?

- **Applicazioni multiple**
 - la multiprogrammazione è stata inventata affinché 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:** MULTIPROCESSING
 - 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:** MULTIPROGRAMMING
 - 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:**

~~MULTIPROCESSING & MULTIPROGRAMMING~~

 - presentano gli stessi problemi
 - che si possono riassumere nel seguente:

non è possibile predire la velocità relativa dei processi

Un esempio semplice

(Pagamento bancario)

- 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?

⇒ No, perché l'esecuzione è CONCORRENTE → può dare 100, 110 o 90

Caso 1: totale = 100

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 → + 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 → -10

P2 sw \$t0, totale

totale=100, \$t0=100, \$a0=-10

Caso 2: totale = 110

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



Caso 3:

Scenario 3: multiprocessing (errato)

- I due processi vengono eseguiti simultaneamente da due processori distinti → ho due core → Non posso leggere dalla RAM

Processo P1:

lw \$t0, totale

add \$t0, \$t0, \$a0

sw \$t0, totale

Processo P2:

lw \$t0, totale

add \$t0, \$t0, \$a0

sw \$t0, totale

con il bus il valore di totale contemporaneamente con entrambi i processori

- Nota:

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

RACE CONDITION: Quando il risultato di un'elaborazione NON dipende solo più dai processi, ma dall'evoluzione temporale del processo (Es. SOMMA su una variab. condivisa da 2 processi)

```

#include <stdio.h>
#include <pthread.h>

#define MAX 100000 ->inizialmente posto a 20000

/*_Atomic*/ volatile int buffer = 0;

void *proc(void *arg){
    int i;
    for(i=0; i<MAX; i++)
        buffer += 1;
    return NULL;
}

int main(int argc, char *argv[]){
    pthread_t p1;
    pthread_t p2;
    pthread_create(&p1, NULL , proc, NULL);
    pthread_create(&p2, NULL , proc, NULL);
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("SUM = %d\n", buffer);
}

```

⇒ Se eseguito in esecuzione più volte, ritorna risultati diversi (buffer = 100000)
 → inizialmente con buffer = 20000 ritorna gli stessi risultati (non si vede)

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, e bloccarsi inesorabilmente quando lo discutete con il docente all'esame...
 - (... un corollario alla legge di Murphy...)

Notazione per descrivere processi concorrenti

- **Notazione esplicita**

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

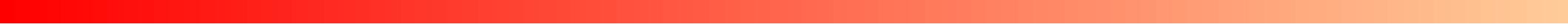
è come avere un main

(quindi se ho più processi, è come se avessi
tanti main indipendenti tra loro)

- **Esempio**

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

Sezione 2



2. Interazioni tra processi

Interazioni tra processi → 1) PROCESSI TOTALMENTE IGNARI

- E' possibile classificare le modalità di interazione tra processi in base a quanto sono "**consapevoli**" uno dell'altro.
- Processi totalmente "ignari" uno dell'altro: (1)
 - processi indipendenti non progettati per lavorare insieme
 - sebbene siano indipendenti, vivono in un ambiente comune
- Come interagiscono? Sono INDEPENDENTI ma
 - competono per le stesse risorse
 - • devono sincronizzarsi nella loro utilizzazione (delle risorse)
- Il sistema operativo:
 - deve arbitrare questa **competizione**, fornendo sincronizzazione

Interazioni tra processi → z) PROCESSI INDIRETTAMENTE a conoscenza l'uno dell'altro

- **Processi "indirettamente" a conoscenza uno dell'altro (z)**
 - 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 → 3) PROCESSI DIRETTAMENTE a conoscenza l'uno dell'altro

- **Processi "direttamente" a conoscenza uno dell'altro (3)**
 - 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

Modelli di interazione

→ Ci sono 2 modelli per far interagire
due processi

• 1) memoria condivisa:

- I processi condividono la memoria
 - Comunicazione tramite sincronizzazione $C \leftarrow S$
 - Più processi che condividono la memoria

(Se p1 agisce su x
e se p2 agisce su x,
la var. x è la stessa)

• 2) Memoria privata

- I processi non condividono la memoria
 - Sincronizzazione tramite comunicazione $S \leftarrow C$

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: si dividono

- Safety ("nothing bad happens") → il programma non fa qualcosa di errato
 - 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 "vocale"

→ il programma eventualmente fa qualcosa di buono ma si blocca

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)

→ serve per evitare che se alcuni processi funzionano scorrettamente non compromettano il risultato finale del sistema

• 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) cioé se il progetto non è in CRASH, ovvero NON È BLOCCATO
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
- **Quali dovrebbero essere le proprietà comuni a tutti i programmi concorrenti?**

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 (^{N.B.} 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 condition
↳ *di fatto, il codice*

Proprietà - programmi concorrenti

- **Proprietà di *liveness***
 - i meccanismi di sincronizzazione utilizzati non devono prevenire l'avanzamento del programma
 - 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) → è una proprietà SAFETY

- **Definizione**

DEF ACCESSO AD UNA RISORSA MUTUALMENTE ESCLUSIVO

- l'accesso ad una risorsa si dice mutualmente esclusivo se ad ogni istante, al massimo un processo può accedere a quella risorsa

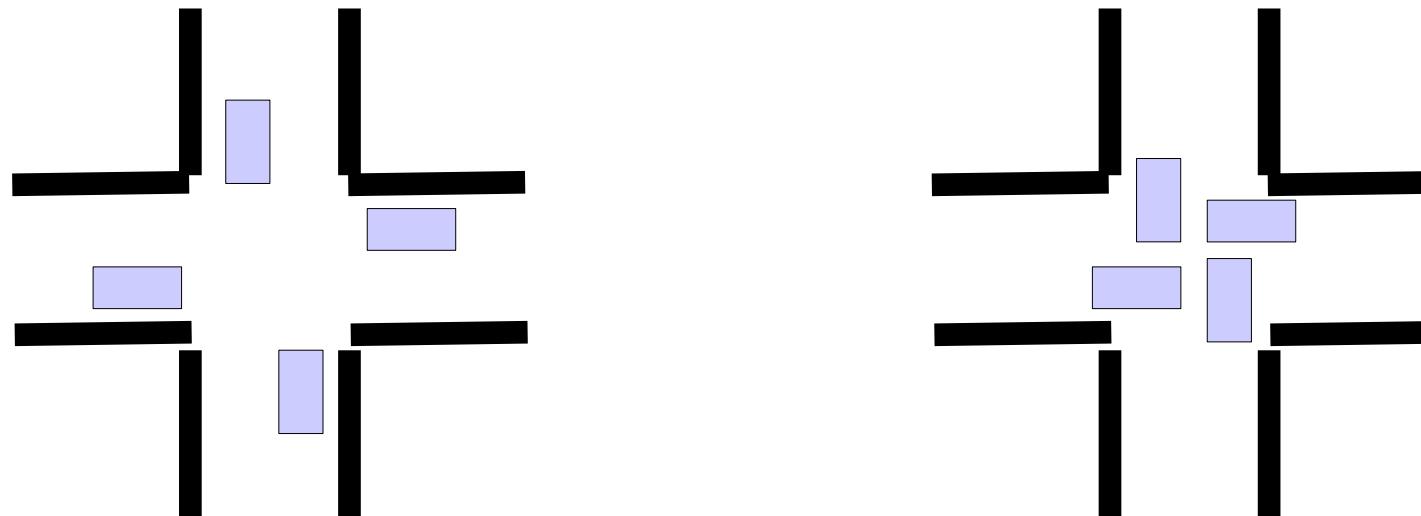
*Es. Piú che devono accedere alla stampante
(devono essere eseguiti uno alla volta)
→ se no ho una race condition*

- **Esempi da considerare:**

- due processi che vogliono accedere contemporaneamente a una stampante
- due processi che cooperano scambiandosi informazioni tramite un buffer condiviso
- Due processi che devono aggiornare la variabile condivisa totale (v. lucidi precedenti)

Deadlock

- **Considerazioni:**
 - la mutua esclusione permette di risolvere il problema della non interferenza
 - ma può causare il blocco permanente dei processi
 - La assenza di deadlock è una proprietà di safety
- **Esempio: incrocio stradale**



Deadlock (stallo)

- **Esempio:**
 - siano R_1 e R_2 due risorse (es. variabili in memoria, documento da stampare)
 - siano P_1 e P_2 due processi che devono accedere a R_1 e R_2 contemporaneamente, prima di poter terminare il programma
 - supponiamo che il S.O. assegni R_1 a P_1 , e R_2 a P_2
 - i due processi sono bloccati in attesa circolare
 - ↳ P_1 dice allora che aspetta R_2 e P_2 dice contemporaneamente che aspetta R_1
→ bisogna uccidere i processi
- **Si dice che P_1 e P_2 sono in deadlock**
 - è una condizione da evitare
 - è definitiva → È UNA CONDIZIONE DEFINITIVA (Una volta che ci entri, ci rimani)
 - nei sistemi reali, se ne può uscire solo con metodi "distruttivi", ovvero uccidendo i processi, riavviando la macchina, etc.

Starvation

→ condizione di impossibilità di proseguire di un processo per colpa degli altri processi che gli impediscono di accedere alle risorse

Considerazioni:

necessarie (Es. processi con priorità diverse)

- 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"
- L'assenza di starvation è una proprietà di liveness

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

↳ se si è fortunati, si risolve automaticamente

Es. Una risorsa che è sempre occupata crea STARVATION

- **Esempio**
 - sia **R** una risorsa
 - siano **P₁, P₂, P₃** tre processi che accedono periodicamente a **R**
 - supponiamo che **P₁** e **P₂** si alternino nell'uso della risorsa
 - **P₃** non può accedere alla risorsa, perché utilizzata in modo esclusivo da **P₁** e **P₂**
- Si dice che **P₃** è in **starvation** ↳ NON è cond. DEFINITIVA
 - 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

- **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

→ quelle azioni che vengono fatte "tutto o niente"

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



→ L'azione non può venire per esempio a metà di un'azione

(il processo o fa l'intera istruzione o non la fa)

Azioni atomiche - Esempi

- **Le singole istruzioni del linguaggio macchina sono atomiche**

- **Esempio:** sw \$a0, (\$t0)

- **Nel caso di parallelismo apparente:**

P. 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:**

P. 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

(Perché il bus non può fare due o più azioni contemporaneamente)

Azioni atomiche - Controesempi

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

Sequenza
di istruzioni
assembly **No!**

lw \$t0, (\$a0) →
add \$t0, \$t0, \$a1 →
sw \$t0, (\$a0) →

Prese singolarmente
ciascuna di queste istruzioni
è un' azione atomica

- Attenzione:

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

Azioni atomiche NEL LINGUAGGIO C

- 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

Se c'è l'operazione
INCR come istruz. assembly
è atomico (processore Intel)

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

} perché se ci pensi ognuna di queste componete a sequenze di istruzioni assembly
(si accede in memoria più volte in)
(ogni operazione → più istruzione assembly)

N.B. Farebbe eccezione l'operazione di incremento se ci fosse un'unica operazione INCR nell'instruction set di una certa architettura

- Operazioni tipo:

- assegnamento di un valore costante ad una variabile

sono atomiche

↳ un solo accesso in memoria

Azioni atomiche

- **Una notazione per le operazioni atomiche**
 - Nel seguito, utilizzeremo la notazione $\langle S \rangle$ per indicare che lo statement S deve essere eseguito in modo atomico
 - Esempio:
 - $\langle x = x + 1; \rangle$
 - È solo una definizione sintattica

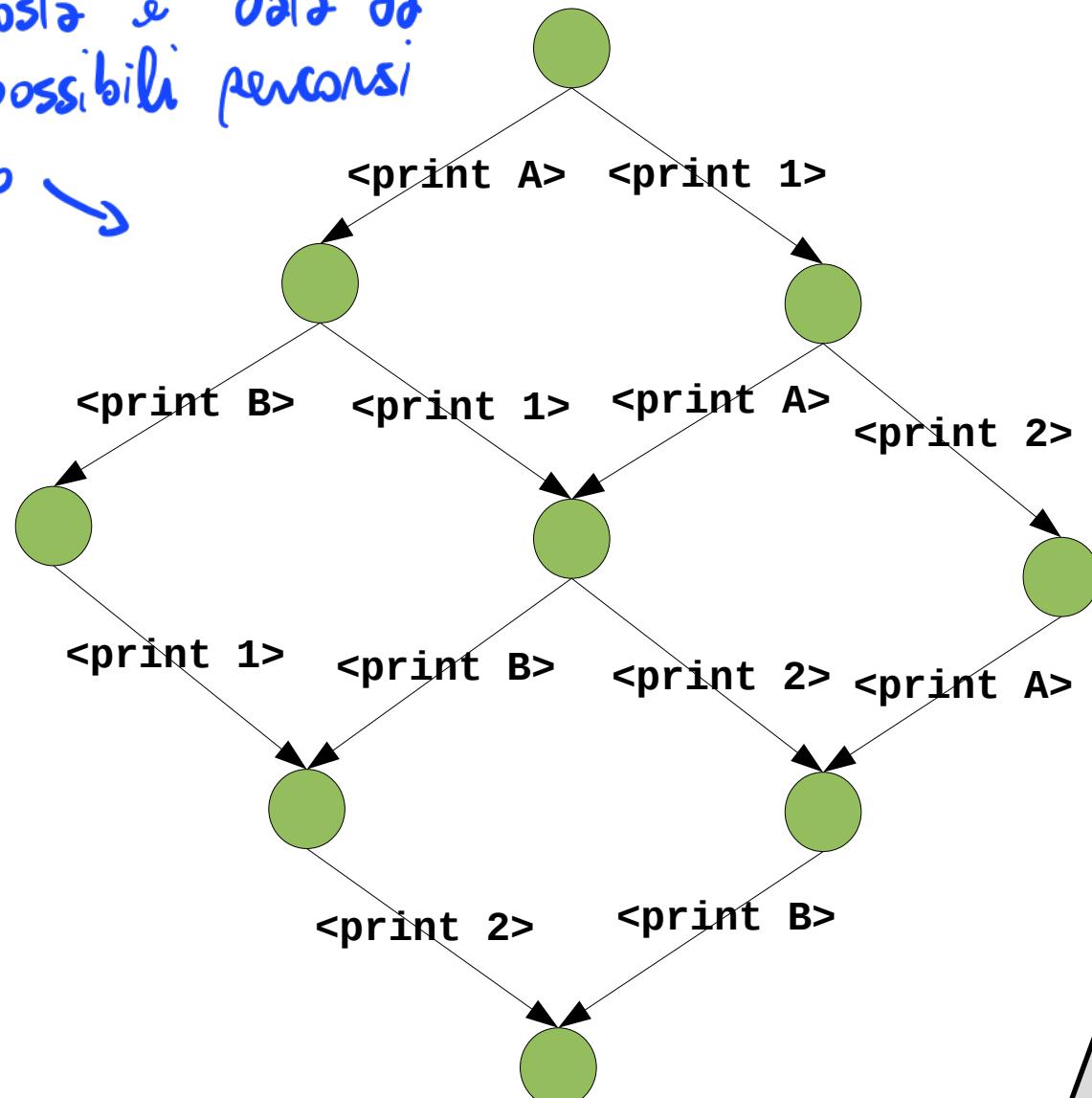
\Rightarrow Se non specificato l' $\langle \rangle$, l'unica azione atomica è
l'assegnazione di costanti

Interleaving di azioni atomiche

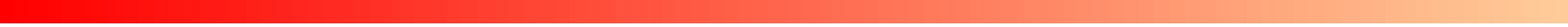
- Cosa stampa questo programma? →

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

La risposta è data da
tutti i possibili percorsi
del grafo ↘



Sezione 3



3. Sezioni critiche

"Non-interferenza"

- **Problema**

- Se le sequenze di istruzioni non vengono eseguite in modo modo atomico, come possiamo garantire la non-interferenza?
trova il modo che istruzioni non atomiche devono essere fatte in contemporanea
- **Idea generale** → *trova il modo che istruzioni non atomiche devono essere fatte in contemporanea*
 - Dobbiamo trovare il modo di specificare che certe parti dei programmi sono "speciali", ovvero devono essere eseguite in modo atomico (senza interruzioni)
*o viene fatto tutto in un colpo solo
(unica istruzione) o niente <? verifica*
 - Serve la mutua esclusione ma occorre che il meccanismo soddisfi anche altre proprietà...

*Il meccanismo si chiama
SEZIONE CRITICA*

Sezioni critiche

- **Definizione**

DEF SEZIONE CRITICA /CS:

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

- **Esempio**

```
process P1
  a1 = read();
  totale = totale + a1; } CS
}
}
```

```
process P2 {
  a2 = read();
  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 aver rispettato le seguenti proprietà
 - Vogliamo garantire che le sezioni critiche siano eseguite in modo mutualmente esclusivo (atomico)
 - Vogliamo evitare situazioni di blocco, sia dovute a deadlock **a causa di 1)**
2) sia dovute a starvation → NO DEADLOCK o STARVATION
 - Vogliamo evitare attese non necessarie
 - Un processo può far attendere altri processi solo se questi ultimi devono usare una sezione critica attualmente occupata dal primo.

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

Process P

[enter cs]; x = x+1; [exit cs];

Process Q

[enter cs]; x = x+1; [exit cs];

Sezioni critiche

- **Esempio:**

Process P

```
val = rand();  
a = a + val;  
b = b + val
```

Process Q

```
val = rand();  
a = a * val;  
b = b * val;
```

- **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 l'invariante
- L'unica soluzione possibile per il S.O. è non eseguire i due processi in parallelo
- Ma così perdiamo i vantaggi!

→ Se svolgiamo i processi in parallelo ho dei vantaggi

Sezioni critiche

- **Esempio:**

Process P

```
val = rand();  
[enter cs]  
a = a + val;  
b = b + val  
[exit cs]
```

Process Q

```
val = rand();  
[enter cs]  
a = a * val;  
b = b * val;  
[exit cs]
```

- **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 coerenza

In questo caso prima a,b sono modificate da p, in seguito da q

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 → PROPRIETÀ CHE VOGLIAMO GARANTIRE

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

- utilizzano istruzioni speciali del linguaggio, progettate apposta
- efficienti
- problemi
 - non sono adatti come soluzioni general-purpose

↳ è un ciclo che
Non fa nulla, controlla
se si verificano certe
condizioni

→ Una soluzione GENERAL-PURPOSE è un tipo di soluzione che non è specifica per un problema, ma è generica e permette la risoluzione di una quantità più ampia di problemi

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** → Il supporto hardware viene mediato dal S.O. in modo che l'accesso sia moderato
 - Semafori
 - Monitor
 - Message passing
- memoria condivisa
- memoria privata

Algoritmo di Dekker

→ puo' fare solo 2 processi (in maniera concorrente)

- Dijkstra (1965)

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

Tentativo 1 (ALGORITMO DI DEKKER)

```
shared int turn = P;  
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  
    }  
}
```

Diagram illustrating the Dekker algorithm's execution flow:

- Process P:** Enters a loop. Inside, it checks if `turn == Q`. If true, it enters a *busy waiting* state (indicated by a red box and a red dotted line). If false, it enters a *critical section*, updates `turn = Q`, and then exits the *critical section*.
- Process Q:** Enters a loop. Inside, it checks if `turn == P`. If true, it enters a *busy waiting* state (indicated by a red box and a red dotted line). If false, it enters a *critical section*, updates `turn = P`, and then exits the *critical section*.

critical section is bracketed under both processes, and *non-critical section* is also bracketed under both processes.

Note

- la variabile `turn` è condivisa
- può essere acceduta solo da un processo alla volta (in lettura o scrittura)

→ C'è il controllo iterativo su una condizione di accesso viene detto **busy waiting**
→ ATTESA NON NECESSARIA? SÌ, perché c'è il busy waiting
→ C'è DEADLOCK? NO → fanno a fummi P e Q
→ C'è STARVATION? NO, P non entra due volte

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"

busy waiting
↑

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 (ALGORITMO DI DEKKER)

```
shared boolean inp = false; shared boolean inq = false;

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

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

Sono contemporaneamente nella CS!
(Entro prima P, poi Q e da quel momento sono entrambi nella CS)

Tentativo 3 (ALGORITMO DI DEKKER)

```
shared boolean inp = false; shared boolean inq = false;

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 (ALGORITMO DI DEKKER)

```
shared boolean inp = false; shared boolean inq = false;

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;

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

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

process P {
    while (true) {
        /* entry protocol */
        needp = true;
        while (needq)
            if (turn == Q) { turn puro esse P o 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

- funziona SOLO per due processi (^{dai 3 in poi è difficilissimo}
^{implementarlo})
- Usa busy waiting (spreca tempo di calcolo)
- Codice complesso da comprendere

Algoritmo di Peterson

→ per due processi (Ne può avere anche di più)
a differenza di Dekker

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

```
process P {  
    while (true) {  
        /* entry protocol */  
        needp = true; | → così se l'altro processo  
        turn = Q; | NON ha bisogno della critical  
        while (needq && turn != P) | section, Non si fa aspettare  
            ; /* do nothing */ | ogni attesa non è casuale  
        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  
    }  
}
```

quando
esco
dalla
cs
needp
turn
true

Algoritmo di Peterson - Dimostrazione

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

Algoritmo di Peterson - Dimostrazione

- **Dimostrazione (assenza di deadlock)**
 - supponiamo che per assurdo che **P** e **Q** siano entrambi bloccati nel ciclo while. *(e quindi suppongo per assurdo che ci sia DEADLOCK)*
 - questo significa che:
 - **needp = true, needq = true, turn == P e turn == Q** per sempre
 - Ma **turn** non può assumere due valori diversi.

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

Algoritmo di Peterson – Generalizzazione per N processi

```
shared int[] stage = new int[N]; /* 0-initialized */
shared int[] last = new int[N]; /* 0-initialized */

process Pi { /* i = 0...N-1 */
    while (true) {
        /* Entry protocol */
        for (int j=0; j < N; j++) {
            stage[i] = j; last[j] = i;
            for (int k=0; k < N; k++) {
                if (i != k)
                    while (stage[k] >= stage[i] && last[j] == i)
                        ;
            }
        }
        critical section
        stage[i] = 0;
        non-critical section
    }
}
```

Algoritmo di Peterson – Generalizzazione per N processi

- Dimostrazione: per esercizio...

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!

→ attese che NON fanno nulla → riduzione di efficienza del processore e spazio di energia
→ riduzioni delle prestazioni

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 → MASCHERAMENTO DEGLI INTERRUPT

- **Idea**

- nei sistemi uniprocessore, i processi concorrenti vengono "alternati" tramite il meccanismo degli interrupt
- allora facciamone a meno! → disabilito temporaneamente gli interrupt

- **Esempio:**

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

⇒ FUNZIONA SOLO SU macchine UNICORE questo meccanismo di mascheramento degli interrupt

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 SECTION

- **Istruzioni speciali**
 - istruzioni che realizzano due azioni in modo atomico
 - esempi
 - lettura e scrittura
 - test e scrittura
 - Le (quasi) sezioni critiche realizzate con istruzioni speciali vengono chiamate **spinlock**

- **Test & Set** *eSEMPIO di ISTRUZIONE SPECIALE*
 - $TS(x, y) := \langle y = x ; x = 1 \rangle$ *tra parentesi angolose ciò che scrivo è atomico*
 - spiegazione
 - ritorna in **y** il valore precedente di **x**
 - assegna 1 ad **x**
- z variabili x, y
↑ passate per indirizzo*
- Esempio: $x=0$*
- \boxed{x} \boxed{y}
- ↓
- \boxed{x} \boxed{y}

Test & Set

shared lock=0;

process P {

int vp; ^{V.DN.} _{var. locale}
while (true) {

do {

TS(lock, vp);

} while (vp);

critical section

assegnamento

lock=0;

non-critical section

}

} ^{perché realizzata con istruzioni speciali}

process Q {

int vq; ^{V.DN.} _{var. locale}

while (true) {

do {

TS(lock, vq);

} while (vq);

critical section

lock=0; ^{assegnamento di costante} _(è atomico)

non-critical section

}

Se lock = 1 la CS è libera

Se lock = 0 la CS
è occupata

lock viene posto a 1
e vp diventa il valore di
lock precedente a tale
assegnamento

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

^{→ In realtà potrebbero verificarsi}

- No, se non assumiamo qualcosa di più

Altre istruzioni possibili

- **test&set non è l'unica istruzione speciale** (*cioè ATOMICHE*)
- **altri esempi:**
 - fetch&set
 - compare&swap
 - etc.
 - atomic swap $\rightarrow \langle x, y = y, x \rangle$

ESERCIZIO:

Sia $F(x)$ una funzione:

$$F(x) = \langle x = x/z ; \text{return } x \rangle$$

Può essere sostituita al Test & Set? Si perché fatto da istruzioni atomiche messe tra virgolette (quindi è eseguita come istruzione atomica)

Riassumendo...

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

Riassumendo...

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

RIASSUMENDO:

MUTUA ESCLUSIONE:

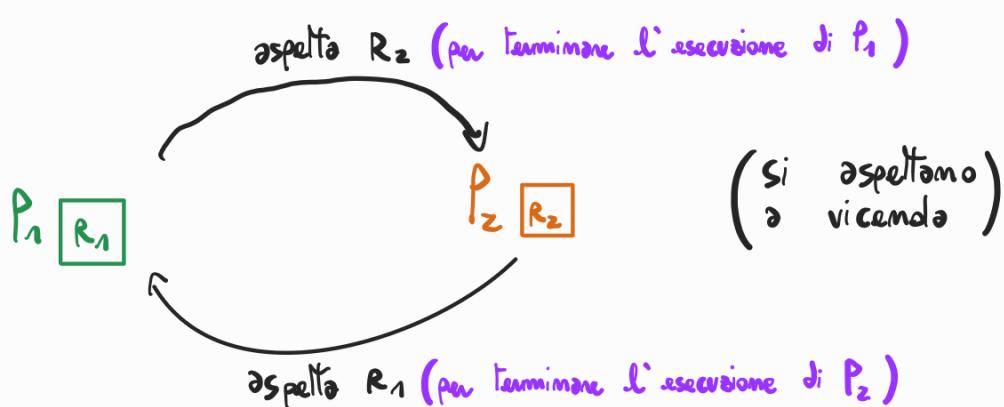
L'accesso ad una risorsa si dice mutualmente esclusivo se ad ogni istante, al massimo un processo può accedere a quella risorsa → risolve la "NON INTERFERENZA" tra processi

DEADLOCK (STALLO):

Due processi sono bloccati in attesa circolare

R_1, R_2 risorse assegnate rispettivamente a P_1, P_2

P_1, P_2 due processi che dovranno accedere a R_1, R_2 CONTEMPORANEAMENTE



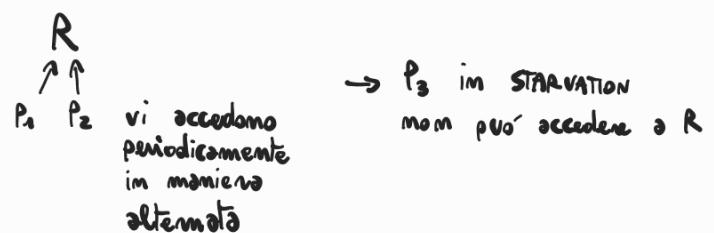
condizione definitiva (per uscire bisogna usare un metodo distruttivo)

STARVATION:

Condizione di impossibilità di proseguire di un processo per colpa degli altri processi che gli impediscono di accedere alle risorse

Siamo P_1, P_2, P_3 processi

Sia R una risorsa



Sezione 4

4. Semafori → paradigma di sincronizzazione
per gestire programmi
concorrenti costitutivi
da più fili esecutivi

Semafori - Introduzione

↳ vanno pensati come un tipo di dato (Es. classe semaforo)

- **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: (L' oggetto della classe semaforo ha due tipi V e U)
 - **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 (NON NEGATIVA)
- questa variabile viene inizializzata ad un valore non negativo ($\text{INIT} \geq 0$)

INIZIALIZZAZIONE →

- l'operazione **P** → metodi senza parametri e senza valori di ritorno
 - attende che il valore del semaforo sia positivo
 - decremente il valore del semaforo ($v--$)
- l'operazione **V**
 - incrementa il valore del semaforo ($v++$)

] Se è zero attende
(attende una V)

• Nota:

- le azioni **P** e **V** sono atomiche;

```
class Semaphore {  
    semaphore(int v);  
    void P(void);  
    void V(void);  
}
```

Semaforo - Invariante

- **Siano**
 - n_P il numero di operazioni P completate
 - n_V il numero di operazioni V completate
 - $init$ il valore iniziale del semaforo ($init$ è inizializzato con un valore positivo)
- Vale il seguente invarianto:
 - $n_P \leq n_V + init$
- **In altre parole, detto valore del semaforo:** $n_V + init - n_P$
 - il valore del semaforo deve sempre essere non negativo (≥ 0)
- **Due casi d'uso:** → 1) Contesa di accesso alle risorse → $init = 1$
 - eventi (init = 0) z) Sincronia
 - 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

N.B. Ho semaforo $INIT = 0$, ho

im attesa
S.PC()
S.PL()
S.PC()

One quale tra i
→ tre processi di tipo P()
viene risvegliato?

Semafori - Implementazione di CS

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

- **Dimostrare che le proprietà sono rispettate**

- mutua esclusione, assenza di deadlock, assenza di starvation, assenza di ritardi non necessari

Semafori - Politiche di gestione dei processi bloccati

- **Per ogni semaforo,**

- Occorre mantenere una struttura dati contenente l'insieme dei processi sospesi

- quando un processo deve essere svegliato, è necessario selezionare uno dei processi sospesi *(me viene scelto uno tra quelli sospesi)*

- **Semafori FIFO / fair**

- politica first-in, first-out

*(é la QUEUE
con modalità di
gestione FIFO)*

delete: in testa (si chiama remove)

append: in coda (si chiama add)

- 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 **(QUEUE)**

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

i processi da svegliare dall'
insieme dei processi
bloccati/
sospesi

- **Nel seguito**

- se non altrimenti specificato, utilizzeremo sempre semafori FIFO

Semafori – Implementazione nei sistemi operativi

- Primitive P e V (da eseguire in una critical section)

```
void P() {  
    INIT  
    if (value > 0)  
        Value--;  
    else { //value ≤ 0  
        pid = <id del processo  
                che ha invocato P>;  
        queue.add(pid); → struttura d'attesa  
        suspend(pid); ←  
    }  
}  
}
```

```
void V() {  
    if (queue.empty())  
        Value++;  
    else {  
        pid = queue.remove();  
        wakeup(pid); ←  
    }  
}  
}
```

Il process id del processo bloccato viene messo in una struttura d'attesa del semaforo → queue (Semafori FIFO)
↳ non è più nei processi ready
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) dalla struttura **queue**

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

Semafori - Implementazione

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

```
void V() {  
    [enter CS]  
    if (queue.empty())  
        value++  
    else {  
        int pid = queue.dequeue();  
        wakeup(pid);  
    }  
    [exit CS]  
}
```

Semafori - Implementazione

• 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

- **Non** è possibile disabilitare gli interrupt? **No!**
- ~~occorrono spinlock~~ di fatto c'è il lock di Test&Set

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 **VANTAGGI dell' USO DEI SEMAFORI**
 - la sezione critica non è quasi mai occupata
 - busy waiting avviene raramente

Semafori - Vantaggi - DIFFERENZE CON - SENZA SEMAFORI

Senza semafori

*potenziale
busy waiting*

Con semafori

```
P | <enter CS>          ] potenziale  
| ...  
| <exit CS>           ] busy waiting  
| /*codice critico*/  
|  
| ...  
| ...  
| ...  
| ...  
| ...  
| ...  
| ...  
| ...  
| ...  
| ...  
| ...  
| ...  
| ...  
| ...  
| ...  
| ...  
| ...  
| ...  
| /*fine cod.crit.*/ ] potenziale  
V | <enter CS>           ] busy waiting  
| ...  
| <exit CS>
```

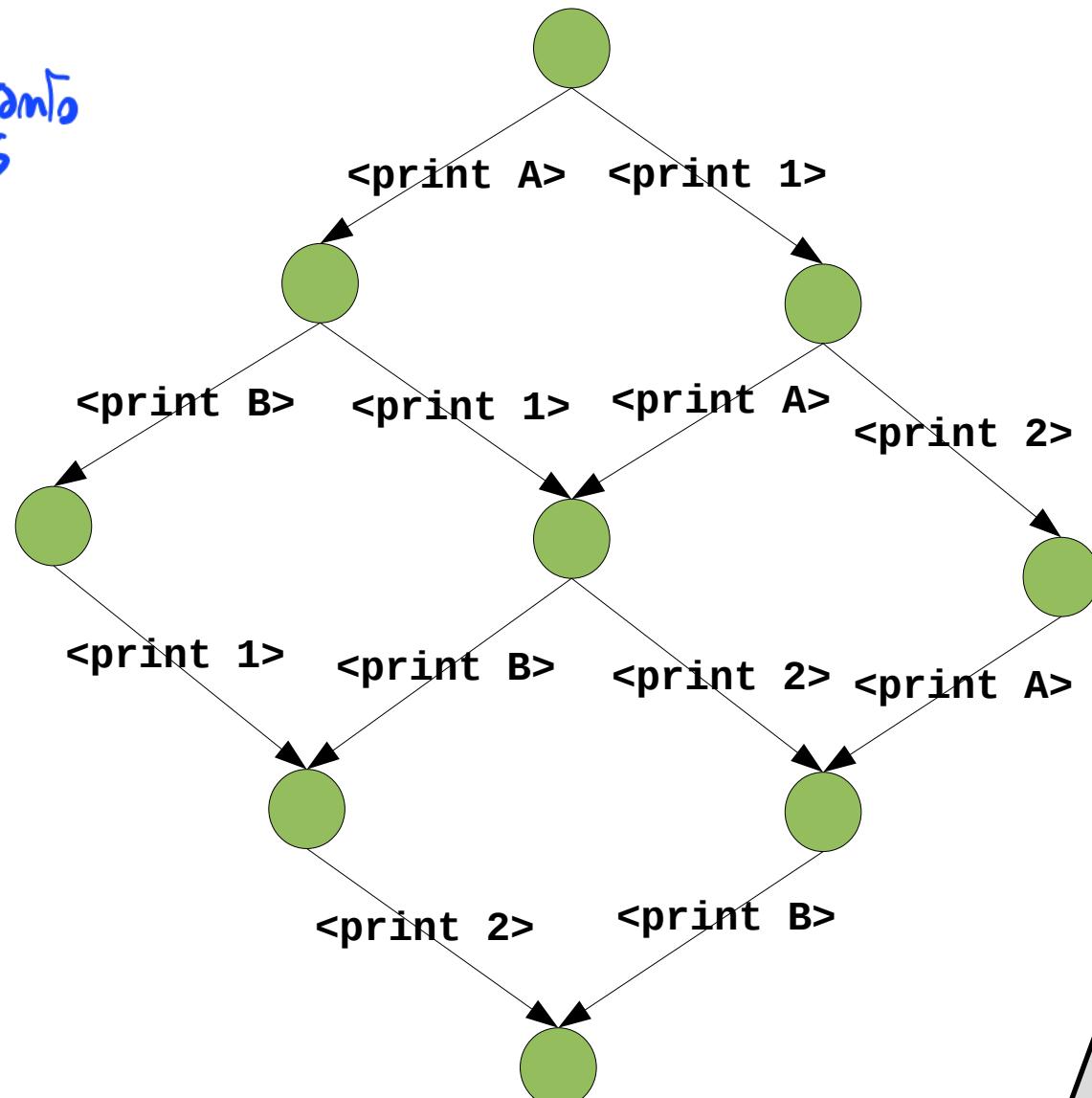
Interleaving di azioni atomiche

- Cosa stampa questo programma?

(Vi ricordate?) No, ma tanto

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

e' GRATIS



Interleaving con semafori

- Cosa stampa questo programma?

```
Semaphore s1 = new Semaphore(0);
```

```
Semaphore s2 = new Semaphore(0);
```

```
process P {
```

<print A> → azione domica

s1.V()
} simonia

s2.P()

<print B>

}

```
process Q {
```

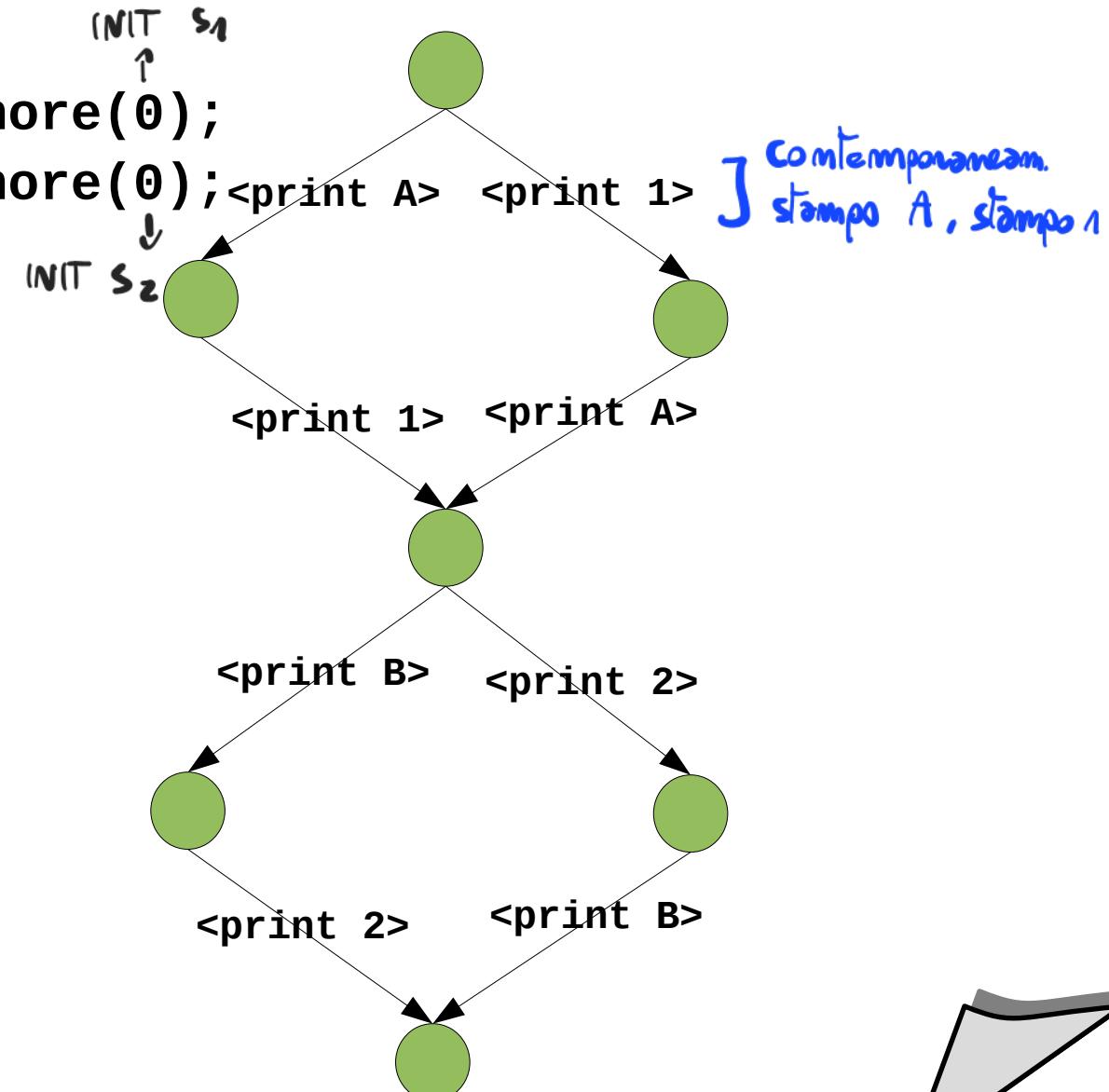
<print 1>

s2.V()
} simonia

s1.P()

<print 2>

}



Semafori binari

Definizione

- variante dei semafori in cui il valore può assumere solo i valori 0 e 1

value
0
1

Cosa servono?

- servono a garantire mutua esclusione, semplificando il lavoro del programmatore
- hanno lo stesso potere espressivo dei semafori "normali"

Invariante dei semafori binari:

- $0 \leq n_V + \text{init} - n_P \leq 1$, oppure
- $0 \leq s.value \leq 1$

Nota:

- molti autori considerano una situazione di errore un'operazione **V** su un semaforo binario che abbia già valore 1

I programmi che posso scrivere con i semafori, possono essere scritti anche con i semafori binari (in modo del tutto equivalente)? Sì

(I semafori binari hanno lo stesso potere espressivo)

Semafori binari – Implementazione nei sistemi operativi

```
class BinarySemaphore {  
    private int value;  
    Queue queue0 = new Queue(); → Coda dei processi che devono aspettare  
    Queue queue1 = new Queue(); → Coda dei processi che hanno finito sbloccati  
    BinarySemaphore() { value = 1; }  
    void P() {  
        [enter CS]  
        int pid = <process id>;  
        if (value == 0) {  
            queue0.enqueue(pid);  
            suspend(pid); → ≠ Ø  
        } else if (queue1.size() > 0) {  
            int pid = queue1.dequeue();  
            wakeup(pid); ↑ prende il primo processo tra quelli in attesa  
        } else  
            value--;  
        [exit CS]  
    }  
}
```

*dei processi che devono aspettare
che hanno finito sbloccati*

```
void V() {  
    [enter CS]  
    int pid = <process id>;  
    if (value == 1) {  
        queue1.enqueue(pid);  
        suspend(pid);  
    } else if (queue0.size() > 0) {  
        int pid = queue0.dequeue();  
        wakeup(pid);  
    } else  
        value++;  
    [exit CS]
```

Esempio:

Binary semaphore \rightarrow INIT = 1 (value = 1)

1) P() (else P) pid = p₁

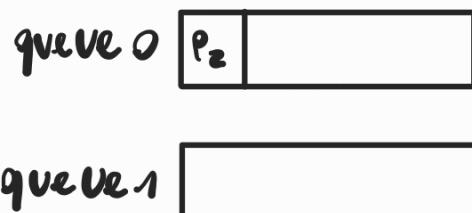
value : 0

suspend p₁



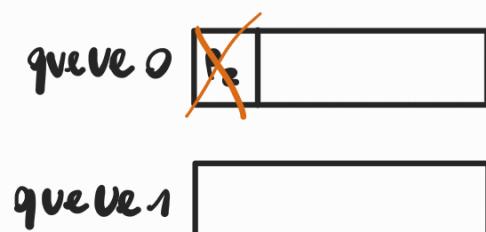
2) P() (if P) pid = p₂

value : 0



3) V() (else if V) pid = v₁

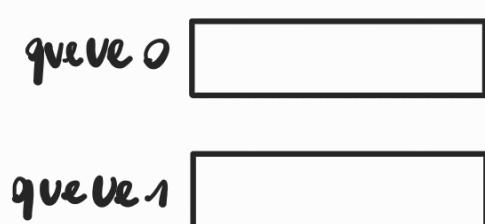
value : 0 wakeup(p₂) \leftarrow queue 0



p₂ viene terminato

4) V() (else V) pid = v₂

value : 1



5) V() (else V) pid = v₃

value : 1

queue 0

queue 1 v₃

6) P() (else if P) pid = p₃

value : 1

queue 0

wakeup(v₃) ← queue 1 v₃

v₃ viene terminato

7) P() (else P) pid = p₄

value : 0

queue 0

queue 1

Semafori - Implementazione tramite semafori binari

- E' possibile utilizzare semafori binari per implementare semafori generali (per farlo servono)
 - un semaforo mutex per garantire mutua esclusione sulle variabili
 - un semaforo privato creato in allocazione dinamica per ogni processo che partecipa
 - una coda per garantire fairness

```
class Semaphore {  
  
    private BinarySemaphore mutex(1);  
    int value;  
  
    QueueOfBinSem queue = new QueueOfBinSem();  
  
    Semaphore(int v) { // +fail if v < 0  
        value = v;  
    }  
}
```

Semafori - Implementazione tramite semafori binari

→ è un semaforo binario (può essere anche "semaphore mondiale")

```

void P() {
    mutex.P(); garantisce la mutua esclusione
    if (value > 0) { mette mutex a 0 del semaforo generale
        value--;
        mutex.V();
    } else { con valore zero
        S = new BinarySemaphore(0);
        queue.enqueue(S);
        mutex.V();
    } S.P();
    free(S);
}

```

```

void V() {
    mutex.P(); così mi assicuro che solo V operi sulla coda (che si condivide)
    if (queue.empty()) garantisce mutua esclusione
        value++;
    else {
        BinarySemaphore s = queue.dequeue();
        s.V();
    }
    mutex.V(); mette a 1 il valore del semaforo generale
}

```

Vogliungo il semaforo creato con valore iniziale 0 in coda (con i semafori in attesa)

E. partendo da

MUTEX : 1
value : 1
Queue :

Esempio ERRATO :

Esempio con semafori generali con semafori binari ERRATO

```
value = 0;  
count = 0;  
BinSem mutex(1);  
BinSem s(0);
```

void P(void):

 mutex.P(); → Se si può fare, decrementa il valore di mutex di 1

 if value > 0:

 value--;

 mutex.V();

 else:

 count++;

 mutex.V(); → Se si può fare incrementa di 1 mutex

 s.P(); → Se si può fare decrementa mutex, altrimenti ATTENDE

void V(void):

 mutex.P();

 if count > 0:

 count--;

 s.V();

 else:

 value++;

 mutex.V();

→ NON è riconcisa!

(Vn è un metodo del semaforo s!)

Siamo partiti dal valore del semaforo s pari a 0
⇒ con semafori generali FIFO → l'inviamante è comunque verificato

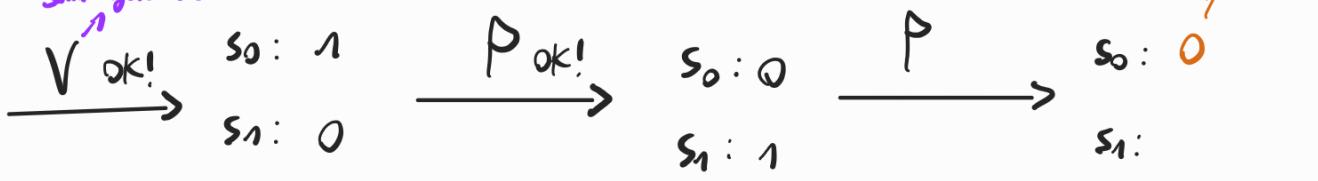
Semafori binari - Implementazione tramite semafori generali

```
class BinarySemaphore {  
  
    private Semaphore s0, s1;      //semafori generali  
    //Non ci sono variabili condivise  
  
    BinarySemaphore(int v) {        // +fail if v not in {0,1}  
        s0 = new semaphore(v)       $\rightarrow$  v può essere 1 o 0 → gli do' il valore del semaforo binario  
        s1 = new semaphore(1-v)     $\rightarrow$   $\bar{v}$  (valore NOT del semaforo binario  
    }  
    void P(void) {  
        s0.P();  
        s1.V();  
    }  
    void V(void) {  
        s1.P();  
        s0.V();  
    }  
}
```

ESECUZIONE ESEMPIO:

1° CASO:

$s_0 : 0$
 $s_1 : 1$



DA RICORDARE:

Semafori

- Le operazioni sui semafori (in notazione ad oggetti) sono S.P() e S.V(). Non hanno alcun parametro ne' alcun valore di ritorno. Se invece volete usare la notazione procedurale le operazioni sono P(S) e V(S), unico parametro e' il semaforo e nessun valore di ritorno.
- Quando si usano semafori tutti gli accessi ai dati condivisi devono avvenire in mutua esclusione
- Le code dei semafori e il valore dei semafori sono strutture e variabili private gestite dall'implementazione del paradigma e non sono accessibili
- Se non altrimenti specificato i semafori sono generali e fair (FIFO)
- I semafori ***Devono*** avere un valore iniziale.
- E' sicuramente errato un programma che usa un semaforo solo con operazioni P e nessuna V o viceversa.

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 *→ problema più semplice di iterazione tra processi*

• 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 → *il produttore mette l'oggetto progettato in modo che possa essere "consumato" dal consumatore*

• 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 con SEMAFORI GENERALI

```
shared Object buffer;
Semaphore empty = new Semaphore(1);
Semaphore full = new Semaphore(0);
```

→ con questo nome
indica che il buffer è vuoto

→ rappresenta la condizione
dove il buffer è pieno

ha un oggetto
che non è ancora stato consumato

```
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);
    }
}
```

PRODUTTORE

- PRODUCE 5 (val)
 - EMPTY, P → EMPTY = 0
 - METTO 5 (val) nel BUFFER
 - FULL, V → FULL = 1

CONSUMATORE

- FULL. P → FULL = 0
 - val locale = 5
 - EMPTY. V → EMPTY = 1
 - CONSUMO 5 (val locale)
2º noVmd

PRODUTTORE

- PRODUCE O (val)
 - EMPTY. P → EMPTY = 0
 - METTO O (val) nel BUFFER
 - FULL. V → FULL = 1

produttore → scatto da parte del produttore e produce un

- PRODUCO Z (val)
- EMPTY, P \rightarrow EMPTY = Até A \rightarrow NO!
val
while

(Il buffer è occupato)

CONSUMATORE

- FULL. P → FULL = 0
 - val locale = 3
 - EMPTY. V → EMPTY = 1
 - CONSUMO 3 (val locale)

PRODUTTORE: (sboccata l' attesa nel while)

- EMPTY. P \rightarrow EMPTY = 0
 - METTO Z (val) nel BUFFER
 - FULL. V \rightarrow FULL = 1

→ Funziona ma la velocità è la minima fra i due

Buffer limitato → avendo il buffer limitato miglion i
ultimi di consumo (> velocità) che vengono nel
tempo

- **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 - Implementazione

Se decide una V ha finito e posso uscire SIZE processi (SIZE + 1 bloccato fino all'arrivo di una V, se inizio con P)

Queue q(maxsize = SIZE); → numero massimo di processi P che si possono fare → buffer limitato condiviso

Semaphore empty =

new Semaphore(SIZE);

Semaphore full =

new Semaphore(0); (Semaforo di mutua esclusione)

Semaphore mutex = → serve a garantire

new Semaphore(1); mutua esclusione:
CONSUMATORE e PRODUTTORE
NON dovemo accedere assieme
al BUFFER (queue)

process Producer {

while (true) {

Object val = produce();

empty.P();

mutex.P();

q.enqueue(val);

mutex.V();

full.V();

}

}

process Consumer {

while (true) {

Object val;

full.P();

mutex.P();

val = q.dequeue();

mutex.V();

empty.V();

consume(val);

}

CONSUMATORE:

- mutex 1
- empty 0
- full 0

- buffer

bloccato

PRODUTTORE:

- produce 9
 - mutex 1
 - empty 0
 - full 3
- buffer 6, 15, 10

bloccato (coda piena)

PRODUTTORE:

- produce 6
 - mutex 1
 - empty 2
 - full 1
- buffer 6

CONSUMATORE:

- mutex 1
 - empty 1
 - full 2
- buffer 15, 10

PRODUTTORE:

- produce 15
 - mutex 1
 - empty 1
 - full 2
- buffer 6, 15

Ora possono andare avanti entrambi (Sia consumazione che produzione)

CONSUMATORE:

- mutex 1
 - empty 2] N.B. rappresentano la situazione della coda
 - full 1]
- buffer 10

PRODUTTORE:

- produce 10
 - mutex 1
 - empty 0
 - full 3
- buffer 6, 15, 10

PRODUTTORE:

- produce 9
 - mutex 1
 - empty 1
 - full 2
- buffer 10, 9

Generalizzare gli approcci precedenti

- **Questione**
 - è possibile utilizzare il codice del lucido precedente con produttori e consumatori multipli? **Sì**
- **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?

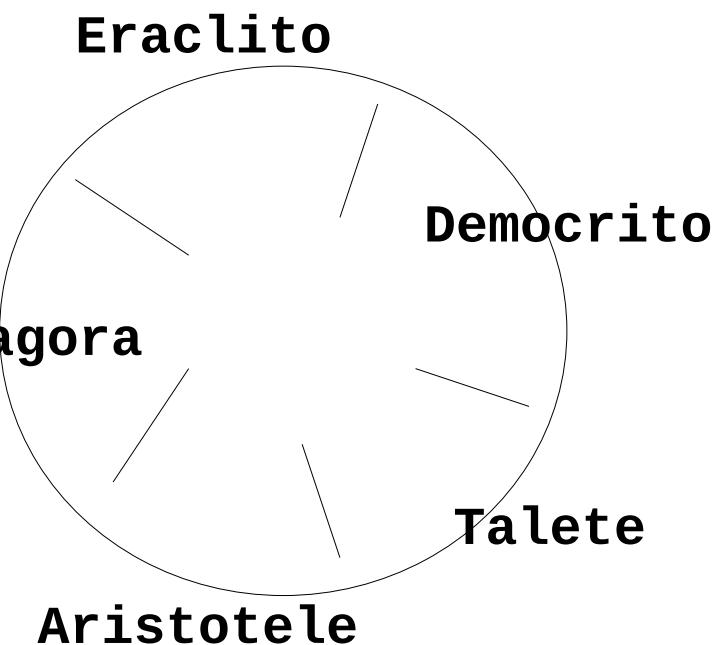
Cena dei Filosofi

• 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

filosofo

→ le posata di sx è condivisa col filosofo di sx e
la posata di dx col filosofo di dx
⇒ Se mangia Eraclito (usa due posate), Democrito
e Pitagora NON possono mangiare



Cena dei Filosofi

- **Note**

- nella versione originale, i filosofi mangiano spaghetti con due forchette (*qui Sono chopstick*)
- 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 */           //Filosofo
  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 preso dal tavolo
- $down_i$ il numero di volte che la bacchetta i viene rilasciata sul tavolo
- **Invariante** → serve a garantire che le bacchette non si creano dal nulla ma vengano rilasciate dai filosofi
 - $down_i \leq up_i \leq down_i + 1$
- **Per comodità:** Pensiamo le bacchette come semafori → tutti inizialmente a 1
 - si può definire $chopstick[i] = 1 - (up_i - down_i)$
(può essere pensato come un semaforo binario)

all'inizio sono tutti disponibili
(sono' erette)

Cena dei Filosofi - Soluzione errata \rightarrow c'è DEADLOCK

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

→ sono FIFO se non si dice NULLA (quindi il primo semaforo che si ferma)
Semaphore chopsticks = (è il primo che riparte)
{ new Semaphore(1), ..., new Semaphore(1) };

```
process Philo[i] { /* i = 0...3 */  
    while (true) {  
        think  
        chopstick[i].P();  
        chopstick[i+1].P();  
        eat  
        chopstick[i].V();  
        chopstick[i+1].V();  
    }  
}
```

```
process Philo[4] {  
    while (true) {  
        think  
        prima prendo →chopstick[0].P();  
        lo banchetto →chopstick[4].P();  
        Se mico prendo eat  
        lo banchetto + chopstick[0].V();  
        chopstick[4].V();  
    }  
}
```

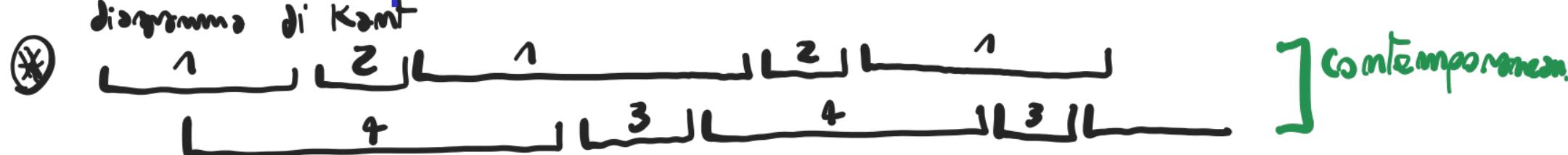
⇒ Può creare starvation sia per gli

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 (*il numero di filosofi è dispari, ≠ soluzioni simmetriche*)
- 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 *% una lista*
- 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

*- I lettori possono leggere la lista (iszione)
- Quando c'è un scrittore, che modifica la lista,
opera in mutua esclusione: mentre uno scrive NON ci sono altri
lettori / scrittori che accedono alla lista*

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:

$$(\text{nr} \geq 0 \ \&\ \& \text{nw} == 0) \ || \ (\text{nr} == 0 \ \&\ \& \text{nw} == 1)$$

- **Note**

- il controllo può passare dai lettori agli scrittori o viceversa quando:

$$\text{nr} == 0 \ \&\ \& \text{nw} == 0$$

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 (1)
 - 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
- Per gli scrittori: possibilità di starvation
- Priorità agli scrittori (2)
 - uno scrittore attenderà il minimo tempo possibile prima di accedere al db
- Per i 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();
}
```

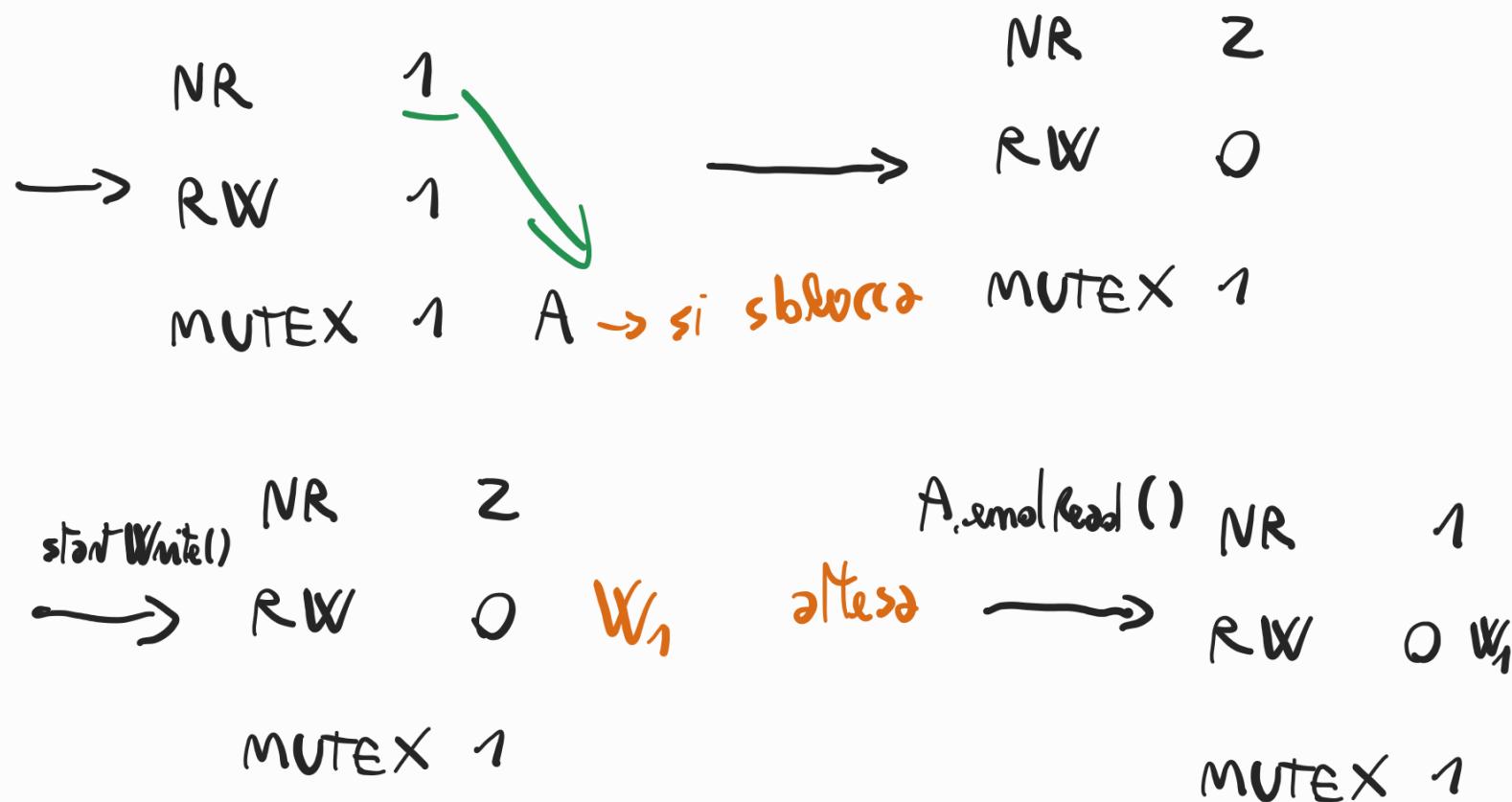
Se è 0 si può cambiare da lettore a scrittore
Se è 1 blocca il passaggio dall' uno all' altro

- **Problemi**

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

Esempio:
La regola P è se è 1, decrementa, se è 0 si blocca (in attesa)

NR	0	startRead()	NR	0	startRead() (attesa)
RW	1	→	RW	1	
MUTEX	1		MUTEX	0	si A



Altro esempio:

NR 0

RW 1

MUTEX 1

A startRead

NR 0

RW 1

MUTEX 0

A'.startWrite

NR 0

RW 0

MUTEX 0

(verso)

Soluzione precedente

• **Problemi**

- limitata a priorità per i lettori
- di comprensione non semplice
- non è chiaro da dove saltano fuori alcuni punti della soluzione

Come derivare una soluzione basata su semafori

- **Alcune definizioni utili (Andrews)**
 - sia **B** una condizione booleana
 - sia **S** uno statement (possibilmente composto)
- **< S >:**
 - esegui lo statement **S** in modo atomico
- **< await(B) → S>**
 - attendi fino a che la condizione **B** è verificata e quindi S
 - l'attesa e il comando vengono eseguiti in modo atomico
 - quindi, quando **S** viene eseguito, **B** è verificata

Come derivare una soluzione basata su semafori

- Andrews suggerisce la seguente procedura

- 1) *Definire il problema con precisione:*

- identificare i processi, specificare i problemi di sincronizzazione, introdurre le variabili necessarie e definire un'invariante

- 2) *Abbozzare una soluzione:*

- produrre un primo schema di soluzione, e identificare le regioni che richiedono accesso atomico o mutualmente esclusivo

- 3) *Garantire l'invariante*

- verifica che l'invariante sia sempre verificato

- 4) *Implementare le azioni atomiche*

- esprimere le azioni atomiche e gli statement **await** utilizzando le primitive di sincronizzazione disponibili

Lettori / scrittori: passi 1, 2, 3

- **Variabili**
 - nr, nw: numero corrente di lettori/scrittori
- **Invariante (vedi lucidi precedenti):**
 - $(nr \geq 0 \ \&\& \ nw == 0) \ || \ (nr == 0 \ \&\& \ nw == 1)$
- **Schema della soluzione**

```
process Reader {
    < await (nw == 0) → nr++ >
    read the database
    <nr-->
}

process Writer {
    < await (nr == 0 && nw == 0) → nw++ >
    write the database
    <nw-->
}
```

startRead
<await nw==0
→ nw++>

endRead
<nw-->

startWrite :
<await nw==0 AND
nw==0 → nw++>

endWrite:
<nw-->

Trasformazione await - semafori

- **Utilizziamo**

- un semaforo **mutex**
 - utilizzato per garantire mutua esclusione (init 1)
- un array di semafori **sem**, dove ad ogni condizione B_j inclusa nelle istruzioni await è associato il semaforo **sem[j]** *dimensione = numero di condiz.*
 - su questi semafori verranno posti in attesa i processi che attendono il verificarsi di una situazione; sono inizializzati a 0
- un array di interi **waiting**, dove ad ogni condizione B_j inclusa nelle istruzioni await è associato l'intero **waiting[j]**
 - questi interi vengono utilizzati per contare il numero di processi in attesa di una certa condizione; sono inizializzati a 0

Trasformazione await - semafori

- $< S >$

```
mutex.P(); → garantisce la  
S;  
mutua esclusione  
SIGNAL();
```

```
void SIGNAL()  
{  
    if ( $B_0 \&& waiting[0] > 0$ )  
        sem[0].V();  
    if  $\text{NON deterministico}$   
     ( $B_1 \&& waiting[1] > 0$ )  
        sem[1].V();  
    if  $\text{NON deterministico}$   
     ( $B_{n-1} \&& waiting[n-1] > 0$ )  
        sem[n-1].V();  
     (!( $B_0 \&& waiting[0] > 0$ )  $\&&$   
        !( $B_1 \&& waiting[1] > 0$ )  $\&&$   
        ...  
        !( $B_{n-1} \&& waiting[n-1] > 0$ ) )
```

- $< \text{await}(B_i) \rightarrow S_i >$

```
mutex.P();  
if (! $B_i$ ) {  
    waiting[i]++;  
    mutex.V(); → lascio la sez.  
    sem[i].P();    critica  
    waiting[i]--;  
}  
S_i;  
SIGNAL(); → dato il nuovo stato del  
            sistema può far andare mutex.V();  
            avanti un altro processo, gli si lascia la mutua  
            esclusione
```

Traformazione await - semafori

- **Questa trasformazione si chiama passaggio del testimone (“passing the baton”)**
- **SIGNAL**
 - verifica se esiste un processo, fra quelli in attesa, che possono proseguire
 - se esiste, "gli passa il testimone"
 - gli "passa" la mutua esclusione
 - altrimenti, rilascia la mutua esclusione
- **Questa tecnica prende il nome di “split binary semaphore”**
 - il sistema agisce come un semaforo binario “suddiviso” tra i vari semafori

R/W trasformato

```
/* Introduced by transformation */

Semaphore mutex = new Semaphore(1);      /* Mutual exclusion */
Semaphore semr = new Semaphore(0);        /* Reader semaphore */
Semaphore semw = new Semaphore(0);        /* Writer semaphore */
int waitingr = 0;                      /* Number of waiting reader */
int waitingw = 0;                      /* Number of waiting writer */

/* Problem variables */
int nr = 0;                            /* Number of current readers */
int nw = 0;                            /* Number of current writers */
```

Note slide precedente:

INT $mr = 0$

INT $mw = 0$

SEM mutex(1)

SEM semr(0) // $mw == 0$

SEM semw(0) // $mw == 0$ & $mr == 0$

INT $wr = 0, ww = 0$

↓
waiting_r ↓
waiting_w

R/W trasformato → SPLIT BINARY SEMAPHORE

```
process Reader {
    while (true) {
        mutex.P();
        if (nw > 0) {
            waitingr++;
            mutex.V();
            semr.P();
            waitingr--;
        }
        nr++;
        SIGNAL();
        read the database
        mutex.P();
        nr--;
        SIGNAL();
    }
}
```

```
process Writer {
    while (true) {
        mutex.P();
        if (nr > 0 || nw > 0) {
            waitingw++;
            mutex.V();
            semw.P();
            waitingw--;
        }
        nw++;
        SIGNAL();
        write the database
        mutex.P();
        nw--;
        SIGNAL();
    }
}
```

R/W trasformato

```
void SIGNAL() {  
  
    if ( (nw == 0) && waitingr > 0)  
        semr.V();  
  
    □ ( (nw == 0 && nr == 0) && waitingw > 0)  
        semw.V();  
  
    □ ( !( (nw == 0) && waitingr > 0 ) &&  
        !( (nw == 0 && nr == 0) && waitingw > 0 ) )  
        { mutex.V() }  
}
```

↳ Se nessuno può andare avanti
nonostante il cambio di stato
del sistema, rilascia la mutua
esclusione

R/W trasformato (SIGNAL ridotto)

```
process Reader {  
    while (true) {  
        mutex.P();  
        if (nw > 0)  
        { waitingr++; mutex.V();  
            semr.P(); waitingr--;}  
        nr++;  
        if (waitingr > 0)  
            semr.V();  
        if (waitingr == 0)  
            mutex.V();  
        read the database  
        mutex.P();  
        nr--;  
        if (nr == 0 && waitingw > 0)  
            semw.V();  
        if (nr > 0 || waitingw == 0)  
            mutex.V();  
    }  
}  
  
process Writer {  
    while (true) {  
        mutex.P();  
        if (nr > 0 || nw > 0) {  
            waitingw++; mutex.V();  
            semw.P(); waitingw--;}  
        nw++;  
        mutex.V();  
        write the database  
        mutex.P();  
        nw--;  
        if (waitingr > 0)  
            semr.V();  
        if (waitingw > 0)  
            semw.V();  
        if (waitingr==0 && waitingw == 0)  
            mutex.V();  
    }  
}
```

R/W trasformato (SIGNAL ridotto, non-determinismo eliminato)

```
process Reader {  
    while (true) {  
        mutex.P();  
        if (nw > 0) {  
            waitingr++;  
            mutex.V();  
            semr.P();  
            Waitingr--;  
        }  
        nr++;  
        if (waitingr > 0)  
            semr.V();  
        else  
            mutex.V();  
        read the database  
        mutex.P();  
        nr--;  
        if (nr == 0 && waitingw > 0)  
            semw.V();  
        else  
            mutex.V();  
    }  
}  
  
process Writer {  
    while (true) {  
        mutex.P();  
        if (nr > 0 || nw > 0) {  
            waitingw++;  
            mutex.V();  
            semw.P();  
            Waitingw--;  
        }  
        Nw++;  
        mutex.V();  
        write the database  
        mutex.P();  
        nw--;  
        if (waitingr > 0)  
            semr.V();  
        else if (waitingw > 0)  
            semw.V();  
        else  
            mutex.V();  
    }  
}
```

R/W trasformato

- **La versione precedente dà priorità ai lettori**
 - starvation per gli scrittori
 - è possibile modificare il codice esistente per dare priorità agli scrittori?
- **Idea**
 - sappiamo se ci sono scrittori in attesa (`waitingw > 0`)
 - possiamo ritardare i lettori, nel caso ci siano scrittori in attesa
- **Cosa cambia?**
 - a seconda di quanto un database venga usato per scrivere / leggere (normalmente: maggior numero di letture) possiamo scegliere uno dei due approcci
 - comunque, abbiamo starvation per i lettori; il problema non è risolto

R/W - Priorità agli scrittori

```
process Reader {  
    while (true) {  
        mutex.P();  
        if (nr > 0 || waitingr > 0)  
            { waitingr++; mutex.V();  
             semr.P(); waitingr--;}  
        nr++;  
        if (waitingr > 0)  
            semr.V();  
        else  
            mutex.V();  
        read the database  
        mutex.P();  
        nr--;  
        if (nr == 0 && waitingw > 0)  
            semw.V();  
        else  
            mutex.V();  
    }  
}
```

```
process Writer {  
    while (true) {  
        mutex.P();  
        if (nr > 0 || nw > 0)  
            { waitingw++;  
             mutex.V();  
             semw.P();  
            }  
        nw++;  
        mutex.V();  
        write the database Modificato  
        mutex.P();  
        nw--;  
        if (waitingr > 0 && waitingw == 0)  
            semr.V();  
        else if (waitingw > 0)  
            semw.V();  
        else  
            mutex.V();  
    }  
}
```

R/W - No starvation

(Né per i lettoni, ma' per gli scrittori)

```
process Reader {  
    while (true) {  
        mutex.P();  
        if (nw > 0 || waitingw > 0)  
        { waitingr++; mutex.V();  
            semr.P(); waitingr--;  
        }  
        nr++;  
        if (waitingr > 0)  
            semr.V();  
        else  
            mutex.V();  
        read the database  
        mutex.P();  
        nr--;  
        if (nr == 0 && waitingw > 0)  
            semw.V();  
        else  
            mutex.V();  
    }  
}
```

Modificato

```
process Writer {  
    while (true) {  
        mutex.P();  
        if (nr > 0 || nw > 0)  
        { waitingw++; mutex.V();  
            semw.P(); waitingw--;  
        }  
        nw++;  
        mutex.V();  
        write the database  
        mutex.P();  
        nw--;  
        if (waitingr > 0)  
            semr.V();  
        else if (waitingw > 0)  
            semw.V();  
        else  
            mutex.V();  
    }  
}
```

Modificato

Problemi - Il barbiere addormentato

- **Descrizione**

- Un negozio di barbiere ha un barbiere, una poltrona da barbiere e **n** sedie per i clienti in attesa
- Se non ci sono clienti, il barbiere si mette sulla sedia da barbiere e si addormenta
- Quando arriva un cliente, sveglia il barbiere addormentato e si fa tagliare i capelli sulla sedia da barbiere
- Se arriva un cliente mentre il barbiere sta tagliando i capelli a un altro cliente, il cliente si mette in attesa su una delle sedie
- Se tutte le sedie sono occupate, il cliente se ne va scocciato!

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"

=> In sintesi i semafori funzionano, ma ci può essere di meglio

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 private
- **Sintassi**
 - originariamente, sarebbe basata su quella del Pascal
 - var, procedure entry, etc.
 - in questi lucidi, utilizziamo una sintassi simile a C/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

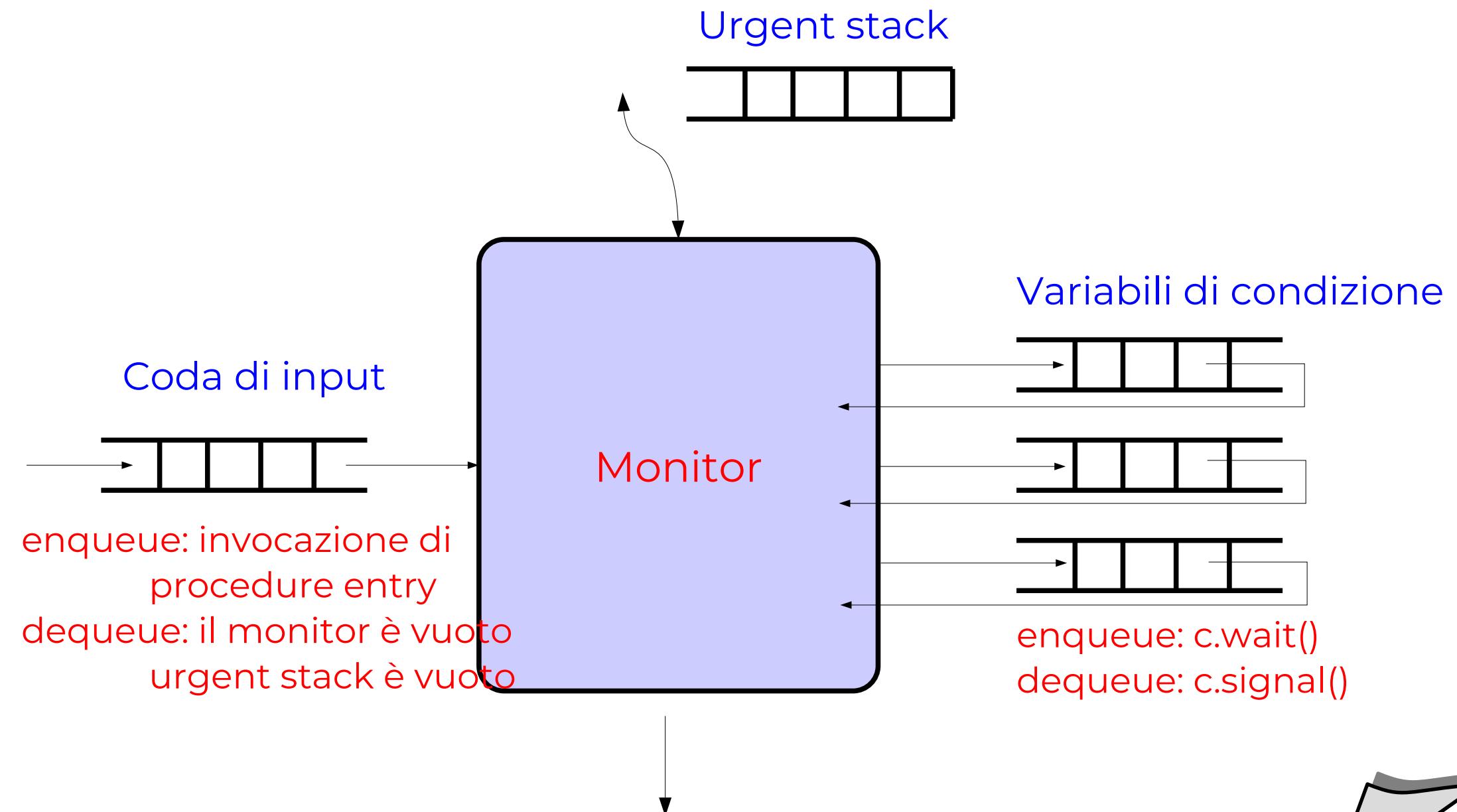
Monitor - Meccanismi di sincronizzazione

- **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 e 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 **e utilizzata in questo corso. Nei compiti d'esame la politica e' signal urgent se non altrimenti indicato.**
- **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() {
        if (value == 0)
            c.wait();
        value--;
    }

    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 false */
        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 la condizione è sempre vera
- eliminando le righe **if** e il blocco di codice condizionato quando è 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();
}
```

R/W tramite monitor – semplificato / no starvation

```
procedure entry void startRead() {
    if (nw > 0 || ww > 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) { ww++; okToWrite.wait(); ww--; }
    nw = nw + 1;
}
procedure entry void endWrite() {
    nw = nw - 1;
    okToRead.signal();
    if (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 void write(int val)
    {
        if (isFull)
            empty.wait();
        buffer = val;
        isFull = true;
        full.signal();
    }

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

Buffer limitato tramite Monitor

```
monitor PCController {  
    QueueOfObj q;  
    int maxsz  
    condition okRead; //q.size > 0.  
    condition okWrite; //q.size < maxsz  
  
    PCController(int size) {  
        maxsz = size;  
    }  
  
    procedure entry void write(Obj val)  
    {  
        if (q.size() >= maxsz)  
            okWrite.wait();  
        q.enqueue(val);  
        okRead.signal();  
    }  
}
```

```
procedure entry Object read() {  
    if (q.size() == 0)  
        okRead.wait();  
    Obj retval = q.dequeue();  
    okWrite.signal();  
    return retval;
```

Filosofi a cena

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

Filosofi a cena

```
monitor DPController {
    condition oktoeat[5];
    boolean eating[5];
    procedure entry void startEating(int i) {
        if (eating[(i+1)%5] || eating[(i+4)%5])
            oktoeat[i].wait();
        eating[i] = true;
    }
    procedure entry void finishEating(int i) {
        eating[i] = false;
        if (!eating[(i+2)%5])
            oktoeat[(i+1)%5].signal();
        if (!eating[(i+3)%5])
            oktoeat[(i+4)%5].signal();
    }
    DPcontroller() {
        for(int i=0; i<5; i++) eating[i] = false;
    }
}
```

Filosofi a cena - No deadlock

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

Filosofi a cena - No deadlock

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

Filosofi a cena

```
process Philo[i] {
    while (true) {
        think
        chopstick[MIN(i, (i+1)%5)].pickup();
        chopstick[MAX(i, (i+1)%5)].pickup();
        eat
        chopstick[MIN(i, (i+1)%5)].putdown();
        chopstick[MAX(i, (i+1)%5)].putdown();
    }
}
```

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

- **Ingredienti**

- un modulo di gestione stack (per urgent)

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

- un modulo di gestione code (per waiting queue)

```
interface Queue {  
    void enqueue(Object x);  
    Object dequeue(void);  
    boolean empty(void);  
}
```

- un semaforo di mutua esclusione **mutex**

Implementazione dei monitor tramite semafori

- **Inizializzazione**

```
Semaphore mutex(1);  
Stack urgent;  
Queue waiting[NCOND]
```

- **Entrata nel monitor**

- **mutex.P();**

- **Wait su cond_i**

```
Semaphore ws = new Semaphore(0);  
waiting[i].enqueue(ws);  
if (urgent.empty())  
    mutex.V()  
else {  
    Semaphore s = urgent.pop();  
    s.V();  
}  
ws.P()  
free(ws)
```

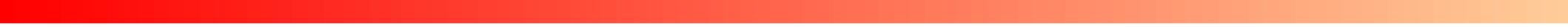
- **Signal su cond_i**

```
If (!waiting[i].empty()) {  
    Semaphore ws =  
        waiting[i].dequeue();  
    Semaphore s = new Semaphore(0);  
    urgent.push(s);  
    ws.V();  
    s.P();  
    free(s)  
}
```

- **Uscita dal monitor**

```
if (urgent.empty())  
    mutex.V()  
else {  
    Semaphore s = urgent.pop();  
    s.V();  
}
```

Sezione 6



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

- **Operazione send sincrona**

- sintassi: **void ssend(msg_t m, pid_t dst)**
- il mittente spedisce il messaggio **m** al processo **dst**, restando bloccato fino a quando **dst** non riceve il messaggio

- **Operazione receive bloccante**

- sintassi: **msg_t sreceive(pid_t snd)**
- il destinatario riceve un messaggio dal processo **snd**; 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 la costante **ANY** per il parametro **snd**)

- **Operazione send asincrona**

- sintassi: **void asend(msg_t m, pid_t dst)**
- il mittente spedisce il messaggio **m** al processo **dst**, senza bloccarsi in attesa che il destinatario riceva il messaggio

- **Operazione receive bloccante**

- sintassi: **msg_t areceive(pid_t snd)**
- il destinatario un messaggio dal processo **snd**; 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 la costante **ANY** per il parametro **snd**)

MP Totalmente Asincrono

- **Operazione send asincrona**

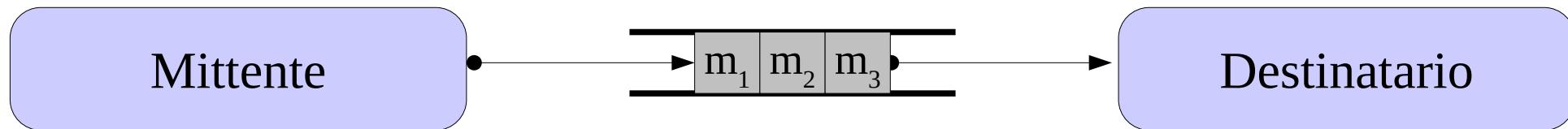
- sintassi: void **asend**(msg_t **m**, pid_t **dst**)
- il mittente spedisce il messaggio **m** al processo **dst**, senza bloccarsi in attesa che il destinatario riceva il messaggio
-

- **Operazione receive non bloccante**

- sintassi: msg_t **nb-receive**(pid_t **snd**)
- il destinatario un messaggio **m** dal processo **snd**; 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 la costante **ANY** per il parametro **snd**)

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(msg_t msg, pid_t dst) {
    Asend(<getpid(), msg>, dst);
    msg_t ack = areceive(dst);
}

msg_t sreceive(pid_t snd) { // snd may be ANY
    msg_t msg;
    pid_t sender;
    <sender, msg> = areceive(snd);
    asend(ack, sender);
    return msg;
}
```

MP asincrono dato quello sincrono (senza gestione ANY)

```
/* p is the calling process */
void asend(msg_t m, pid_t dst) {
    ssend("SND(m,getpid(),dst)", server);
}

msg_t areceive(pid_t snd) {
    ssend("RCV(getpid(),snd)", server);
    msg_t m = sreceive(server);
    return m;
}

process server {
    /* One element x process pair */
    int[][] waiting;
    Queue[][] queue;
    while (true) {
        handleMessage();
    }
}
```

```
void handleMessage() {
    msg = sreceive(0);
    if (msg == <SND(m,p,q)>) {
        if (waiting[p,q]>0) {
            ssend(m, q);
            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, q);
        }
    }
}
```

NB: gli indici
sono invertiti
ma è corretto

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(0);  
    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);  
    }  
}
```

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 distinzione

- **Message passing**
 - da un certo punto di vista, il meccanismo più diffuso
 - può essere poco efficiente (copia dati tra spazi di indirizzamento)

Potere espressivo

- **Definizione**

- si dice che il paradigma di programmazione **A** è espressivo almeno quanto il paradigma di programmazione **B** (e si scrive $\mathbf{A} \geq \mathbf{B}$) quando è possibile esprimere ogni programma scritto con **B** mediante **A**
- ovvero, quando è possibile scrivere una libreria che consenta di implementare le chiamate di un paradigma **B** esprimendole in termini di **A** si avrà $\mathbf{A} \geq \mathbf{B}$

- **Definizione**

- si dice che due paradigmi hanno lo stesso potere espressivo se $\mathbf{A} \geq \mathbf{B}$ e $\mathbf{B} \geq \mathbf{A}$

Potere espressivo

In vari punti di questi lucidi si mostrano delle relazioni tra i vari paradigmi di programmazione mediante funzioni di implementazione.

Si possono tracciare le seguenti classi di paradigmi:

- **Metodi a memoria condivisa**

- semafori, semafori binari, monitor hanno tutti lo stesso potere espressivo
- dekker e peterson, Test&Set necessitano di busy waiting

- **Metodi a memoria privata**

- message passing asincrono ha maggiore potere espressivo
- message passing sincrono
(abbiamo dovuto aggiungere un processo, non solo una libreria)

L'ennalogico della programmazione concorrente

- **Se rispettate le regole dell'ennalogico avete probabilità non nulla che il vostro programma/esercizio sia corretto**
- **Ne non rispettate le regole dell'ennalogico avete la certezza assoluta che il programma/compito sia errato**

Ennalogo parte prima: regole generali

- Chiamate di fantasia quali block, wakeup, restart non devono mai comparire in soluzioni di esercizi
- Semafori, monitor, message passing sono paradigmi indipendenti. Gli esercizi indicano quale usare. Non si possono *mischiare* primitive di paradigmi diversi.
- E' sicuramente errato un programma che usa variabili prima di assegnare loro un valore iniziale
- Soluzioni con semafori, monitor o message passing non devono contenere busy-wait (sono stati creati per evitarlo)
- E' possibile definire strutture dati senza implementarle a patto che (1) non contengano primitive concorrenti (sincronizzazione o comunicazione) (2) non siano miracolose, devono poter essere implementabili quindi i parametri devono contenere tutte le informazioni necessarie per fornire i servizi richiesti.

Ennalogo parte 2: regole per i semafori

- **Le operazioni sui semafori (in notazione ad oggetti) sono S.P() e S.V(). Non hanno alcun parametro ne' alcun valore di ritorno. (Se invece volete usare la notazione procedurale le operazioni sono P(S) e V(S), unico parametro e' il semaforo e nessun valore di ritorno.)**
- **Quando si usano semafori tutti gli accessi ai dati condivisi devono avvenire in mutua esclusione**
- **Le code dei semafori e il valore dei semafori sono strutture e variabili private gestite dall'implementazione del paradigma e non sono accessibili**
- **Se non altrimenti specificato i semafori sono generali e fair (FIFO)**
- **I semafori *Devono* avere un valore iniziale.**
- **E' sicuramente errato un programma che usa un semaforo solo con operazioni P e nessuna V o viceversa.**

Ennalogo parte 3: regole per i monitor

- **Le operazioni definite sulle variabili di condizione sono c.wait() e c.signal(). Non hanno parametri, ne' valore di ritorno.**
- **Bloccarsi (Busy Wait) all'interno di un monitor è deadlock**
- **Le code delle variabili di condizione, lo urgent stack sono strutture private gestite dall'implementazione del paradigma e non sono accessibili**
- **La routine di inizializzazione di un monitor non può contenere wait e signal**
- **se non altrimenti specificato la politica delle variabili di condizione è signal-urgent**
- **E' sicuramente errato un programma che su una variabile condizione effettua solo wait e mai signal. (viceversa la condizione e' inutile).**
- **E' sicuramente errato un programma che ha una wait come prima istruzione di ogni procedure entry**

Ennalogo parte 4: regole per il message passing

- **Esistono tre paradigmi: sincrono, asincrono, completamente asincrono. Si usa quello indicato dall'esercizio.**
- **Se il testo indica "message passing asincrono" allora NON è "completamente asincrono"**
- **E' sicuramente errato un programma che usa solo send e nessuna receive o che usa solo receive e nessuna send**
- **Se non e' indicato diversamente i servizi di message passing sono FIFO**
- **Message Passing e' un paradigma a memoria privata: non possono esistere variabili condivise fra i processi.**
- **Processi server vanno utilizzati se espressamente richiesti o se non è possibile implementare i servizi senza.**