Sistemi Operativi

Alice Benatti

Sommario

[Sistemi Operativi 1](#_Toc119747307)

[Introduzione ai Sistemi Operativi 4](#_Toc119747308)

[Cos'è un sistema operativo? 4](#_Toc119747309)

[Cosa fa un sistema operativo? 4](#_Toc119747310)

[Organizzazione di un sistema operativo 4](#_Toc119747311)

[Interrupt o Interruzioni 4](#_Toc119747312)

[Implementazione: 4](#_Toc119747313)

[Perché questo meccanismo? 5](#_Toc119747314)

[Sistema parallelo 5](#_Toc119747315)

[Symmetric multiprocessing (SMP) 5](#_Toc119747316)

[Asymmetric multiprocessing 5](#_Toc119747317)

[Sistemi distribuiti 5](#_Toc119747318)

[Sistemi real-time 6](#_Toc119747319)

[Concorrenza 7](#_Toc119747320)

[Processi e programmi 7](#_Toc119747321)

[Stato di un processo 7](#_Toc119747322)

[Cos’è la concorrenza? 7](#_Toc119747323)

[Dove la troviamo? 7](#_Toc119747324)

[Applicazioni multiple 7](#_Toc119747325)

[Applicazioni strutturate su processi 8](#_Toc119747326)

[Struttura del sistema operativo 8](#_Toc119747327)

[Multiprocessing vs Multiprogramming 8](#_Toc119747328)

[Race Condition 8](#_Toc119747329)

[Conclusioni 8](#_Toc119747330)

[Interazioni tra processi 9](#_Toc119747331)

[Classificazione delle modalità di interazione tra processi 9](#_Toc119747332)

[Processi “ignari” l’uno dell’altro 9](#_Toc119747333)

[Processi “indirettamente” a conoscenza l’uno dell’altro 9](#_Toc119747334)

[Processi “direttamente” a conoscenza uno dell’altro 9](#_Toc119747335)

[Mutua Esclusione 10](#_Toc119747336)

[Deadlock 10](#_Toc119747337)

[Starvation 10](#_Toc119747338)

[Azioni Atomiche 10](#_Toc119747339)

[Sezioni Critiche 12](#_Toc119747340)

[Requisiti: 12](#_Toc119747341)

[Approccio Software 12](#_Toc119747342)

[Approccio Hardware 12](#_Toc119747343)

[Approccio Supporto 12](#_Toc119747344)

[Algoritmo di Dekker 13](#_Toc119747345)

[Dijkstra 13](#_Toc119747346)

[Algoritmo di Peterson 14](#_Toc119747347)

[Generalizzazione per N processi 15](#_Toc119747348)

[Disabilitazione degli interrupt 16](#_Toc119747349)

[Problemi 16](#_Toc119747350)

[Test & Set - Istruzioni speciali 16](#_Toc119747351)

[Altre istruzioni speciali 16](#_Toc119747352)

[Semafori 17](#_Toc119747353)

[Cosa sono? 17](#_Toc119747354)

[Semaforo invariante 17](#_Toc119747355)

[Semafori e Critical Section 17](#_Toc119747356)

[Gestione dei processi bloccati 17](#_Toc119747357)

[Come si implementano? 17](#_Toc119747358)

[Problemi “classici” della programmazione concorrente 17](#_Toc119747359)

[Produttore/Consumatore 18](#_Toc119747360)

[Buffer Limitato 18](#_Toc119747361)

[Filosofi a cena 18](#_Toc119747362)

[Lettori e scrittori 18](#_Toc119747363)

[Monitor 18](#_Toc119747364)

[Cosa sono? 18](#_Toc119747365)

[Caratteristiche 18](#_Toc119747366)

[Meccanismi di sincronizzazione 18](#_Toc119747367)

[Signal Urgent 18](#_Toc119747368)

[Implementazione dei semafori 18](#_Toc119747369)

[Message Passing 18](#_Toc119747370)

[Cos’è? 18](#_Toc119747371)

[Operazioni 19](#_Toc119747372)

[Tassonomia 19](#_Toc119747373)

[Sincrono 19](#_Toc119747374)

[Asincrono 19](#_Toc119747375)

[Completamente Asincrono 19](#_Toc119747376)

[Altro 19](#_Toc119747377)

[Attività di un sistema operativo 20](#_Toc119747378)

[Multiprogrammazione e multitasking 20](#_Toc119747379)

[Duplice modalità di funzionamento 20](#_Toc119747380)

[Timer 20](#_Toc119747381)

[Gestione delle Risorse 20](#_Toc119747382)

[Processi 20](#_Toc119747383)

[Memoria File 20](#_Toc119747384)

[Memoria di massa 20](#_Toc119747385)

[Cache 20](#_Toc119747386)

[I/O 20](#_Toc119747387)

[Sicurezza e protezione 20](#_Toc119747388)

[Virtualizzazione 20](#_Toc119747389)

[Sistemi distribuiti 20](#_Toc119747390)

[Strutture dati del Kernel 20](#_Toc119747391)

[Liste 20](#_Toc119747392)

[Stack 20](#_Toc119747393)

[Code 20](#_Toc119747394)

[Alberi 20](#_Toc119747395)

[Funzioni 20](#_Toc119747396)

[Mappe hash 20](#_Toc119747397)

[Bitmap 20](#_Toc119747398)

[Ambienti di elaborazione 20](#_Toc119747399)

[Elaborazione tradizionale 20](#_Toc119747400)

[Mobile 20](#_Toc119747401)

[Client-Server 21](#_Toc119747402)

[Peer-to-peer 21](#_Toc119747403)

[Cloud 21](#_Toc119747404)

[Embedded real-time 21](#_Toc119747405)

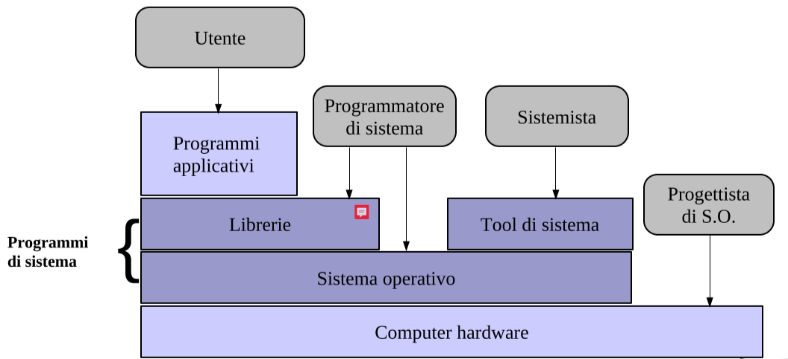
Introduzione ai Sistemi Operativi

# Cos'è un sistema operativo?

Un **sistema operativo** è livello di astrazione che realizza il concetto di processo.  
Il linguaggio fornito dal SO è definito dalle system call.

È implementato da:  
 - insieme di programmi [software] che gestiscono elementi fisici di un calcolatore [hardware];  
 - fornisce una piattaforma ai programmi applicativi   
 - agisce da intermediario fra l'utente e la struttura fisica del calcolatore

# Cosa fa un sistema operativo?



**System call**

**Efficienza** e **Semplicità**:  
Il sistema operativo **controlla l'hardware** e ne **coordina** l'utilizzo da parte dei programmi applicativi per gli utenti.

Hardware, memoria, I/O device forniscono al sistema le risorse elaborative fondamentali.

Programmi applicativi definiscono il modo in cui si usano queste risorse per la risoluzione dei problemi computazionali degli utenti.

Figura Visione a strati dei componenti hardware e software che compongono un elaboratore

# Organizzazione di un sistema operativo

Ogni moderno sistema elaborativo è composto da CPU, dischi, I/O devices hanno dei controllori che sono connessi tramite canali di comunicazione bus.

Questi bus permettono di accedere alla memoria condivisa dal sistema.

Ogni controllore di sistema ha un proprio **driver del dispositivo** permettendo al sistema operativo di gestire le specificità del controllore e fungendo da interfaccia uniforme con il resto del sistema.

# Interrupt o Interruzioni

## Implementazione:

1. L’hardware della CPU dispone di una linea di richiesta di interruzione che la CPU controlla dopo l'esecuzione di ogni istruzione
2. Se la CPU rileva che un controllore ha ricevuto un segnale sulla linea di richiesta di interruzione,  
   allora legge il numero di interruzione e salta alla routine di gestione dell’interruzione con il numero letto come indice nell’array dedicato
3. Si avvia l’esecuzione all’indirizzo associato all’indice
4. Il gestore salva le informazioni di stato, determina la causa dell’interruzione, esegue l’elaborazione necessaria, ripristina lo stato ed esegue un’istruzione di ritorno dall’interruzione per riportare la CPU allo stato di esecuzione precedente all’interruzione.

## Perché questo meccanismo?

Consente alla CPU di rispondere ad un evento asincrono. Ma mancano ancora delle funzioni necessarie:

* possibilità di posticipare la gestione degli interrupt durante elaborazioni critiche
* inviare efficientemente l’interrupt al gestore adeguato al dispositivo
* interrupt multilivello: per permettere al SO di distinguere la priorità e rispondere in base ad essa

# Sistema parallelo

Con **sistema parallelo** si intende un singolo elaboratore che possiede più unità di elaborazione.

Tassonomia della struttura:

* SIMD [Single Instruction, Multiple Data] 🡪 CPU eseguono all’unisono lo stesso programma.
* MIMD [Multiple Instruction, Multiple Data] 🡪 CPU eseguono programmi differenti su dati differenti.

Tassonomia della dimensione:

* Sistemi a basso parallelismo: pochi processori molto potenti;
* Sistemi a massiccio parallelismo: molti processori con potenza non elevata.

Sistemi Coupled:

* **Tightly**: memoria condivisa & basso livello di parallelismo;
* **Loosely**: memoria privata con canali & alto livello di parallelismo.

PRO: incremento delle prestazioni

## Symmetric multiprocessing (SMP)

Ogni processore esegue una copia identica del sistema operativo.  
Processi diversi possono essere eseguiti contemporaneamente.  
Molti sistemi operativi moderni supportano SMP

## Asymmetric multiprocessing

Ogni processore è assegnato ad un compito specifico; un processore master gestisce l’allocazione del lavoro ai processori slave.  
Più comune in sistemi estremamente grandi.

# Sistemi distribuiti

Sono sistemi composti da più elaboratori indipendenti (con proprie risorse e proprio sistema operativo), collegati da una rete, appaiono come se fossero un unico sistema.

|  |  |
| --- | --- |
| PRO | CONTRO |
| Condivisione di risorse | - autonomia tra computer |
| Suddivisione di carico 🡪 incremento prestazioni | Sembra che un SO controlli tutti gli elaboratori che appartengono al sistema distribuito |
| Affidabilità |  |
| + latenza vs sys paralleli |  |

# Sistemi real-time

Sono i sistemi per i quali la correttezza del risultato non dipende solamente dal suo valore ma anche dall'istante nel quale il risultato viene prodotto.

Real-time ≠ esecuzione veloce

Concorrenza

# Processi e programmi

Con **processo** si intende un’attività controllata da un programma che si svolge su un processore.

Un **programma** specifica una sequenza di istruzioni.

Assioma di **finite progress**

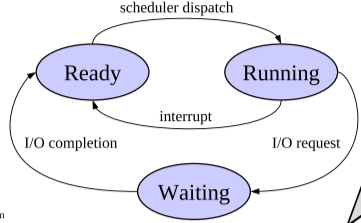
Ogni processo viene eseguito ad una velocità finita, non nulla, ma sconosciuta.

## Stato di un processo

Lo si può descrivere dai componenti seguenti:

* La sua immagine di memoria
* La sua immagine nel processore
* Lo stato di avanzamento

In un sistema multiutente, più utenti possono eseguire lo stesso programma. Infatti, sullo stesso computer più utenti possono usare un servizio di posta elettronica anche se non svolgono le stesse azioni: uno scrive un nuovo messaggio mentre l’altro invece ne sta leggendo una.



Ogni istanza viene infatti considerata un processo separato.

Running = processo in esecuzione

Waiting = processo in attesa di un evento esterno 🡪 non può essere eseguito

Ready = processo può essere eseguito ma il processore è impegnato in altre attività

# Cos’è la concorrenza?

Due programmi si dicono in **esecuzione concorrente** se vengono eseguiti in parallelo (sia con parallelismo reale, sia apparente). (“Sistemi Operativi Concorrenza 2022/23”)

È l’insieme di notazioni per descrivere l’esecuzione concorrente di 2+ programmi.  
È anche l’insieme di tecniche per risolvere i problemi associati all’esecuzione concorrente: *comunicazione* e *sincronizzazione*.

# Dove la troviamo?

## Applicazioni multiple

Per permettere che + 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/thread concorrenti.

## Struttura del sistema operativo

Molte funzioni di un sistema operativo possono essere implementate come insieme di processi/thread.

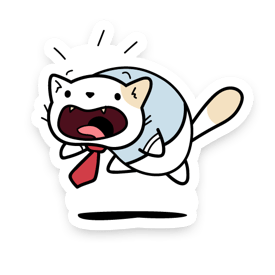
# Multiprocessing vs Multiprogramming

|  |  |
| --- | --- |
| **MULTIPROCESSING** | **MULTIPROGRAMMING** |
| ALTERNATI NELLO SPAZIO | ALTERNATI NEL TEMPO ⇒ danno l’impressione di avere un multiprocessore |
| + processi eseguiti simultaneamente su processori diversi | ad ogni istante è attivo max un processo |
| Overlapping | Interleaving |
| Esecuzione simultanea | Esecuzione Simulata |

In realtà presentano entrambi gli stessi problemi:

1. "non è possibile predire gli istanti temporali in cui vengono eseguite le istruzioni" (“Sistemi Operativi Modulo 2: Concorrenza - unibo.it”)
2. i due processi accedono ad 1+ risorse condivise

# Race Condition

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. (“Sistemi Operativi Concorrenza 2021/22 - unibo.it”)

È fondamentale eliminarle per avere un programma concorrente.



# Conclusioni

Scrivere programmi concorrenti è + difficile di scrivere programmi sequenziali.

La loro **correttezza** si determina non solo dall’esattezza dei passi svolti da ogni componente del programma, ma soprattutto dalle interazioni tra di loro (volute o meno).

In particolare, fare debug di applicazioni con Race Condition.

Interazioni tra processi

pagina 26/200 delle slide 01-concorrenza.pdf

# Classificazione delle modalità di interazione tra processi

🡪 attraverso la consapevolezza l’uno dell’altro

### Processi “ignari” l’uno dell’altro

I processi sono indipendenti non progettati per lavorare insieme, ma convivono in un ambiente comune.

Come interagiscono? Competono per le stesse risorse  
 Sincronizzarsi nel loro utilizzo

⇒ il SO deve controllare questa **competizione** con meccanismi di **sincronizzazione**.

### Processi “indirettamente” a conoscenza l’uno dell’altro

Condividono risorse [buffer] per scambiarsi informazioni.  
Interagiscono indirettamente con le risorse condivise.

Come interagiscono? Cooperazione per qualche scopo  
 Sincronizzarsi nell’utilizzo delle risorse

⇒ il SO deve semplificare la **cooperazione** con meccanismi di **sincronizzazione**.

### Processi “direttamente” a conoscenza uno dell’altro

Comunicano tra loro in base ai propri ID: con comunicazione **diretta** basata su uno scambio di messaggi.

Come interagiscono? Cooperazione per qualche scopo  
 Comunicazione di informazioni agli altri processi

⇒ il SO deve aiutare nella **cooperazione** con meccanismi **comunicativi**.

Proprietà di un programma concorrente

È un attributo che rimane vero per ogni possibile storia di esecuzione del programma stesso.

Tipo 1. **Safety** *[“nothing bad happens”]* il programma va nella direzione voluta  
-> non esegue azioni scorrette

Tipo 2. **Liveness** *[“something good eventually happens”]* il programma avanza senza fermarsi (“vitale”)

**Consensus**: dalla teoria dei sistemi distribuiti

Si consideri un sistema con N processi: ogni processo all’inizio propone un valore e alla fine tutti i processi devono accordarsi su uno dei valori proposti

**Safety:** se 1 processo decide ⇒ deve decidere uno dei valori proposti  
 se 2 processi decidono ⇒ devono decidere lo stesso valore

**Liveness**: ogni processo corretto [non in crash] deciderà a suo tempo

Nei programmi sequenziali  
**Safety** = correttezza dello stato finale ⇒ il risultato è quello voluto  
**Liveness** = terminazione

Nei programmi concorrenti  
**Safety** = se i processi condividono risorse ⇒ non devono “interferire” tra di loro all’accesso delle risorse  
**Liveness** = la sincronizzazione non deve prevenire l’avanzamento del programma

# Mutua Esclusione

**Mutua Esclusione** si ha quando al massimo un processo può accedere in ogni istante ad una risorsa.

Esempio 1. Due processi che vogliono accedere contemporaneamente ad una stampante

Esempio 2. Due processi che cooperano scambiandosi informazioni attraverso un buffer condiviso

# Deadlock

Si ha quando date due risorse e due processi che devono accedere a quelle due risorse contemporaneamente prima di terminare il programma.

Supponendo che il SO assegni il primo processo alla prima risorsa e il secondo processo alla seconda risorsa, i due processi sono bloccati in attesa circolare.

# Starvation

Processo che non può accedere ad una risorsa perché “sempre occupata”

Sei in coda ad uno sportello e continuano ad arrivare i “furbetti” che superano e passano avanti, non riuscirai mai a parlare con l’impiegato.

# Azioni Atomiche

Le **azioni atomiche** vengono compiute in modo indivisibile e soddisfano la condizione “o tutto o niente”.

Se si ha parallelismo reale ⇒ si garantisce che l’azione non interferisca con altri processi durante l’esecuzione

Se si ha parallelismo apparente ⇒ il context switch fra i processi avviene prima o dopo l’azione che non può interferire

**Esempio 1** Le singole istruzioni del linguaggio macchina sono atomiche

**Esempio Parallelismo Reale**  
 Se più istruzioni tentano di accedere alla stessa cella di memoria, la politica di arbitraggio del bus garantisce che una delle due venga servita per prima e l’altra successivamente

**Esempio Parallelismo Apparente**  
 il meccanismo degli interrupt garantisce che un interrupt venga eseguito prima o dopo un’istruzione, mai *durante*

**Esempio compiti di concorrenza**Assumiamo che in ogni istante, vi possa essere al massimo un accesso alla memoria alla volta

Questo significa che operazioni tipo:   
- aggiornamento di una variabile  
- incremento di una variabile   
- valutazione di espressioni  
- etc.   
non sono atomiche.

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

Sezioni Critiche

pag 47/200 delle slides 01-concorrenza.pdf

Con **sezioni critiche** si intende la parte di un programma che utilizza 1+ risorse condivise, detta anche *critical section* o *CS*

è una sezione critica poiché accede alla risorsa condivisa “totale” quando a1 e a2 non sono condivise.

Se un processo entra in una critical section, prima o poi ne uscirà: in quanto un processo può **terminare** **solo fuori dalla sua sezione critica**

## Requisiti:

1. Mutua Esclusione Solo un processo alla volta deve essere all’interno della CS, tra tutti quelli che hanno una CS per la stessa risorsa condivisa. (“Slide 60”)
2. Assenza di Deadlock Uno scenario in cui tutti i processi restano bloccati definitivamente non è ammissibile (“Slide 71”)
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

## Approccio Software

* Responsabilità 🡪 sui processi che vogliono accedere ad un oggetto distribuito
* Problemi:
  + Soggetto ad errori
  + Costoso per l’esecuzione (busy waiting)

Approccio Hardware

* Istruzioni “speciali” del linguaggio macchina
* Efficienza
* Problemi 🡪 non adatti per soluzioni general-purpose

## Approccio Supporto

Il Sistema Operativo o il Linguaggio deve garantire mutua esclusione.

# Algoritmo di Dekker

## Dijkstra

* Algoritmo basato per la mutua esclusione
* Soluzione sviluppata in fasi

**Dimostrazione – mutua esclusione**Per Assurdo supponiamo che P e Q siano in critical section contemporaneamente:

Poiché: accessi in memoria sono esclusivi e per entrare in CS devono aggiornare/valutare le variabili di controllo needq e needpIpotizziamo che entra in CS per primo il process Q:

needq = true fino a quando Q non uscirà dal while

Il process P entra in CS mentre Q è in CS ⇒ ⱻ un istante temporale in cui:

needq = false mentre Q è in CS

Il ché è assurdo!

~*qed*

**Dimostrazione - assenza di deadlock**Per Assurdo supponiamo che né il process P né il process Q possano entrare in critical section.

⇒ P e Q devono entrare in CS nel primo while

Figura Algoritmo di Dekker o Dijkstra

⇒ ⱻ un istante t dopo che needp, needq = true

Supponiamo che all’istante t, turn = Q  
avremo che turn sarà modificato di nuovo solo quando Q entra in CS

⇒ Dopo l’istante t, turn = Q

P entra nel primo while e imposta needp = false  
il ché è assurdo!

*~qed*

**Dimostrazione – assenza di ritardi non necessari**

Se process Q esegue codice non critico ⇒ needq = false

Dunque, process P può entrare in critical section.

*~qed*

**Dimostrazione – assenza di Starvation**

Se process Q richiede accesso alla CS ⇒ needq = true

Se process P esegue codice non critico ⇒ Q entra in CS

Process P esegue il codice rimanente: CS>entra>esce  
 Quando esce passa il turno a Q ⇒ Q potrà rientrare

*~qed*

# Algoritmo di Peterson

* + semplice e lineare dell’[Algoritmo di Dekker](#_Algoritmo_di_Dekker)/[Dijkstra](#_Dijkstra)
* + generalizzabile per processi multipli



Figura Algoritmo di Peterson

**Dimostrazione – mutua esclusione**

Supponiamo che process P sia in CS:  
dimostriamo che Q non può entrare.

Sapendo che needp = true, Q entra solo se turn = Q quando esegue il while.

Consideriamo lo stato al momento in cui process P entra in CS:  
 needq == false ∨ turn == P  
 - se needq == false ⇒ Q deve ancora eseguire needq == true, e deve eseguirlo dopo l’ingresso di PC in CS impostando turn = P impedendosi di entrare

- se turn == P ⇒ fai come sopra

*~qed*

**Dimostrazione – assenza di deadlock**

Per Assurdo supponiamo che process P vuole entrare in CS ma è bloccato nel suo while:  
avremo quindi needp = true, needq = true, turn = Q per sempre.

Possono esserci 3 casi:

1. Q non vuole entrare in CS ⇒ impossibile poiché turn = Q

2. Q bloccato nel ciclo while ⇒ impossibile per turn = Q

3. Q è in CS e ne esce prima o poi ⇒ impossibile needq diventerà = false quando Q uscirà da CS

*~qed*

**Dimostrazione – assenza di ritardi non necessari**

Se process Q esegue codice non critico ⇒ needq = false

Dunque, process P può entrare in critical section.

*~qed*

**Dimostrazione – assenza di Starvation** [simile a deadlock]

Q continua ad entrare ed uscire dalla sua CS, prevenendo l’ingresso di P.

È impossibile perché:  
- quando Q prova ad entrare nella CS turn = P  
- per needp = true  
⇒ Dunque Q deve attendere che P entri nella CS, ma non accadrà mai.

*~qed*

## Generalizzazione per N processi



# Immagine che contiene testo Descrizione generata automaticamenteDisabilitazione degli interrupt

Figura Esempio di Disabilitazione degli interrupt integrando le critical section

Nei sistemi uniprocessore, i processi concorrenti vengono “alternati” grazie al meccanismo degli interrupt.

Gli algoritmi di Dekker e Peterson come istruzioni atomiche hanno solamente operazioni di Load e Store, ma possiamo ovviare a ciò con gli interrupt.

## Problemi

* Lasciare ai processi di riattivare gli interrupt → molto pericoloso ⚠
* Ci permette di ridurre il parallelismo del processore.
* Non funziona sui sistemi multiprocessore

### Test & Set - Istruzioni speciali

Le istruzioni speciali realizzano due azioni in modo atomico (lettura e scrittura, test e scrittura).

TS (x, y) ≔ < y = x; x = 1

Ritorno in y il valore precedente di x, assegno x = 1

#### Mutua esclusione

Entra solo chi riesce a settare il primo lock

#### No deadlock

Il primo che riesce ad eseguire TS() entra senza problemi

#### No unnecessary delay

Un processo fuori dalla Critical Section non blocca altri processi

#### No stavation

Non si verifica se non assumiamo qualcosa di più

Figura Esempio di utilizzo di Test&Set

### Altre istruzioni speciali

[Test & Set](#_Test_&_Set) non è l’unica istruzione speciale, esistono anche Fetch & Set, Compare & Swap e molte altre.

Semafori

# Cosa sono?

Si tratta di un paradigma per la sincronizzazione. Di base 2+ processi possono cooperare attraverso semplice segnali, per permettere che un processo possa essere bloccato in un punto preciso del suo programma fino a quando non riceve un segnale da un altro processo.

Il **semaforo** è un tipo di dato astratto per il quale sono definite 2 operazioni:  
 - **V (Verhogen)** per inviare un segnale: verifica di un evento/rilascio di una risorsa  
 - **P (Passeren)** per attendere il segnale: aspettare un evento/rilascio di una risorsa

##### Informalmente

Un semaforo è simile ad una variabile intera: la variabile deve essere inizializzata ad un valore non negativo.  
L’operazione V (verhogen) incrementa il valore del semaforo, mentre l’operazione P (passeren) attende che il valore del semaforo sia positivo per poi decrementarlo.

Sia P che V operation sono atomiche.

# Semaforo invariante

Un semaforo è invariante quando dati:  
- n­­p  numero di operazioni di P completate  
- nv numero di operazioni di V completate  
- init valore iniziale del Semaforo  
Vale che np ≤ nv + init  
ovvero considero come valore del semaforo nv + init – npil valore del semaforo deve sempre essere ≥ 0

## Semafori e Critical Section

Dimostrazione

# Gestione dei processi bloccati

# Come si implementano?

* Con semafori binari

PRO & CONTRO

Problemi “classici”   
della programmazione concorrente

# Produttore/Consumatore

# Buffer Limitato

# Filosofi a cena

# Lettori e scrittori

Monitor

# Cosa sono?

# Caratteristiche

# Meccanismi di sincronizzazione

# Signal Urgent

Rappresentazione

# Implementazione dei semafori

Message Passing

# Cos’è?

# Operazioni

# Tassonomia

## Sincrono

## Asincrono

## Completamente Asincrono

Altro

# Attività di un sistema operativo

## Multiprogrammazione e multitasking

## Duplice modalità di funzionamento

## Timer

# Gestione delle Risorse

## Processi

## Memoria File

## Memoria di massa

## Cache

## I/O

# Sicurezza e protezione

# Virtualizzazione

# Sistemi distribuiti

# Strutture dati del Kernel

## Liste

## Stack

## Code

## Alberi

## Funzioni

## Mappe hash

## Bitmap

# Ambienti di elaborazione

## Elaborazione tradizionale

## Mobile

## Client-Server

## Peer-to-peer

## Cloud

## Embedded real-time