# 30-03-2023

# Comunicazione fra processi

- In certi casi **l'output di alcuni processi diventano input di altri processi** (***pipeline***)
  - *Un processo non può andare avanti finchè non ha il risultato da un altro processo*

> *Si deve bypassare il problema dell'ipotesi di **introduzione di interrupt**.*

Per costruire un'efficiente scambio di dati fra processi si devono attenzionare (*vale anche per i thread*):

- come **scambiare i dati** fra processi (si possono avere ***parti di memoria condivise*** fra processi)
- evitare **l'accavallamento di operazioni sulle parti di memoria condivise**
- **sincronizzazione**: le operazioni devono essere **COORDINATE**.

> *Queste problematiche si incontrano nei compiti svolti dal kernel*

## Corse critiche (race conditions)

Fenomeno in cui due o più processi leggono o scrivono dati su ***RISORSE CONDIVSE***. Il risultato dipende da **CHI** e **QUANDO** ha eseguito l'operazione

### Esempi

1. versamenti su conto corrente: in caso di versamento in corso e improvvisa interruzione (*operazione non completata*) potrebbe creare problemi. In caso di ripartenza, l'operazione riprende esattamente da dove si era fermato. Si deve realizzare un "*versamento senza interruzioni*"
2. operazioni di kernel. Ci sono 2 approcci:
   1. **Kernel preemptive**: consente a un processo di poter **essere prelevato**, **stoppato** e **rimosso**. ("*Basta, ora ti fermi e do spazio ad altro!*")
      - Visto che c'è un tempo prestabilito, un processo può essere prelevato e rimosso nel **momento preciso** di manipolazione di una variabile globale (*condivisa*)
   2. **Kernel non-preemptive**: **NON** consente il prelevamento di un processo ma è il **processo stesso che rilascia** in maniera volontaria **l'esecuzione.**
      - In questo caso (a differenza del caso precedente (che introduce race conditions) **NON** c'è il pericolo che due processi si accavallano perchè è il processo stesso rilascia la CPU quando termina le sue operazioni

## Soluzione

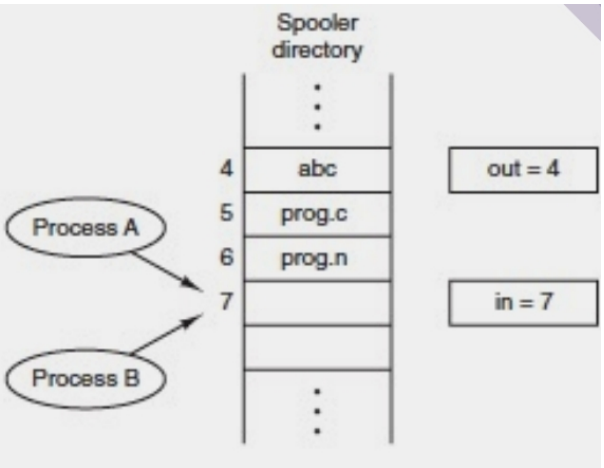
Il problema ***RACE CONDITIONS*** viene risolto con il principio della ***MUTUA ESCLUSIONE***: in ogni singolo istante, **un solo processo alla volta può accedere alla risorsa condivsa** e **può uscire** dalla parte condivisa solo **dopo aver completato l'attività**.

- *Se **SOLO UN PROCESSO** sta lavorando sulla parte conivisa, allora quest'ultima non si può occupare.*

### Esempio pratico: Spool di Stampa (*stampante condivisa*)

E' un processo demone, cioè ogni tanto si sveglia, opera e poi si addormenta.

Nella **DIRECTORY DI SPOOL** vengono memorizzati i nomi dei file che devono essere stampati e questo rappresenta il mezzo di comunicazione fra processi.




`out` : indica la prima posizione del primo processo che deve andare in stampa (*file successivo da stampare*)
`in` : prima posizione vuota, dove si deve scrivere il file che ha richiesto di essere stampato (***RISORSA CONDIVISA***)

Nel caso in cui **due processi richiedono la stampa**: *Processo A e B*:

- *A* accede, legge `in` = 7 e subito dopo viene **INTERROTTO**
- *B* legge `in` (non ancora modificato da *A*) e stampa nella posizione successiva di in (*che viene giustamente incrementata*) **DOPO AVER INTERROTTO A**
- *A* riprende e ha il **VECCHIO VALORE** di `in` che valeva 7 e **riscrive B** e si ha una **sovrascrittura**
- Il risultato (*output*) di *B* viene ***eliminato***

> In definitiva, **mentre A sta lavorando su `in` allora non può essere interrotto da B finchè esso non completa le sue operazioni**

- Se i processi devono **SOLAMENTE LEGGERE**, allora il *problema non si pone*.
- Se i processo **LEGGONO e SCRIVONO**, allora il *problema è presente* e bisogna garantire la **mutua esclusione** (come nei lock delle transazioni in *Basi di Dati* -> `read_lock` e `write_lock` )

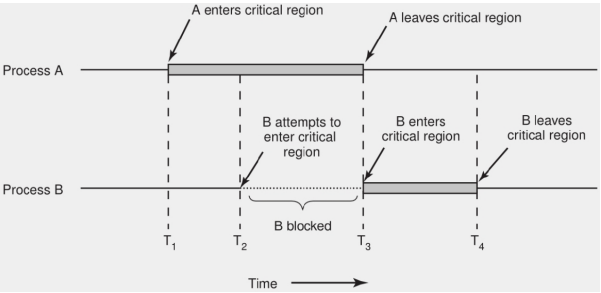
# Regione/sezione critica

Non tutti i processi dello stesso programma agiscono sulle risorse condivise. Si può avere, per *esempio*:

- un blocco di codice che prende **input** da tastiera
- un altro che li sovrasscrive su un file letto da un secondo processo (***RISORSA CONDIVISA***)
- un altro processo stampa a video

Con la **REGIONE / SEZIONE CRITICA** ci si focalizza sull'insieme di istruzioni del processo che agiscono sul file condiviso.

> *Un processo **non deve entrare** nella regione critica mentre **un altro processo è presente nella stessa regione critica**.*




Un processo ha una `SEZIONE CRITICA` (*parte di processo che racchiude **le istruzioni che lavorano sulla parte condivisa***) e una `SEZIONE NON CRITICA`.

## Soluzione race conditions

Si devono soddisfare **contemporaneamente** le seguenti **CONDIZIONI**:

1. **MUTUA ESCLUSIONE**: garantire che se la **sezione critica è occupata**, **nessun altro processo può accedere** alla stessa sezione critica
2. **nessuna assunzione** sulla ***VELOCITA' DI ESECUZIONE*** oppure sul ***NUMERO DI CPU***: non si fanno i conti sul tempo di esecuzione di un processo
3. se un processo *A* legge le informazioni da tastiera, *B* può accedere alla sezione critica di *A*:
   - **nessun processo** che è **FUORI** dalla sua sezione critica (*A*) può **bloccare un altro processo**
4. **nessun processo** sta in **ATTESA INFINITA** per entrare nella sua sezione critica

# Modi di realizzazione della mutua esclusione

1. In caso di macchina con **MONOPROCESSORE** (in un **singolo istante** c'è un **SINGOLO PROCESSO** in esecuzione) si possono **disattivare gli interrupt**. Verranno riabilitati all'uscita dalla sezione critica di quel processo.
   - In questo caso, un processo utente gestisce gli interrupt non è una scelta del tutto saggia. Non si hanno garanzie sulla riattivazione degli interrupt.
   - In linea teorica *questo metodo funziona* ma è *molto rischioso*
   - In caso di **MULTIPROCESSORE**, gli interrupt vengono disabilitati sul **SINGOLO PROCESSO** che accede alla propria sezione critica **MA NON SUGLI ALTRI**
   - Generalmente, gli interrupt devono essere gestiti dal **KERNEL** (dal *sistema operativo*)
2. **Gestione a livello SOFTWARE**: usare **variabili di lock**:
   - può avere solo 2 valori: **0** = accesso **libero** , **1** = accesso **negato**
   - Ogni volta che un processo deve accedere alla sezione critica, va a leggere la variabile di lock e in base al suo valore entra oppure no.
   - Se si **entra** nella sezione critica, allora la **variabile di lock** viene cambiata in **1**
   - Quando il **processo termina** le proprie operazioni ed esce dalla propria sezione critica, la **variabile di lock** va cambiata e **diventa 0**
   - Anche questa soluzione riscontra il problema visto nella stampa.
     - Un processo A legge lock = 0 (non è un'operazione ATOMICA perchè può essere interrotta). Accede alla sezione. Non riesce a completare la commutazione di lock e avviene un interrupt (visto che comunque sono attivi)
     - Un processo **NON deve essere interrotto** se non ha ancora completato le sue operazioni (*come nell'esempio soprastante*)

## Alternanza stretta

```
int N=2
int turn

function enter_region(int process)
    while (turn != process) do
        nothing

function leave_region(int process)
    turn = 1 – process
```

- Se un processo vuole accedere a una sezione critica già occupata, rimane sempre in **ATTESA ATTIVA** (while) e questo metodo è detto **BUSY WAITING**: si ha l'azione di testare continuamente una variabile ( `turn` ) finchè da essa non si ha il valore desiderato.
- Il continuo test *stressa la CPU* e la occupa di conseguenza.

> In caso che la variabile da verificare attivamente è un **lock**, si parla di ***SPIN LOCK***

L'alternanza stretta **è efficace** quanto **l'alternanza è breve**. Se un processo è più lento rispetto ad un altro, allora l'approccio non è più efficiente e si viola la **condizione 3**. (nessun processo può bloccare un altro processo se il primo è fuori dalla propria sezione critica)

> **SE SOLO IL PROCESSO 0** entra nella sua sezione critica, in questo esempio, può entrare esclusivamente una volta perchè turn vale 1 e il processo è 0. Affinchè il processo 0 rientri nella propria sezione critica dovrebbe accadere che l'1 dovrebbe accedere alla propria sezione critica così da resettare a 0 la variabile `turn`. In questo caso si viola la condizione 3, cioè un processo blocca un altro processo se esso è fuori dalla sua sezione critica.

## Soluzione di Peterson

```
int N=2
int turn //variabile condivisa
bool interested[N] //contiene N processi. Inizialmente sono tutti falsi o 0

function enter_region(int process)
        other = 1 – process //gli altri(other) processi
        interested[process] = true
        turn = process
        while (interested[other] = true and turn = process)
                do nothing //l'altro non deve aver espresso l'intenzione di accedere, SOLO
IO(interested[process]) L'HO ESPRESSO

function leave_region(int process)
        interested[process] = false
```

Questa soluzione presenta 2 **aspetti negativi**:

- ogni processo che aspetta il proprio tuno, lo aspetta in ***MANIERA ATTIVA*** (pesa sulla *CPU*) -> **BUSY WATING**
- quando si lavora su sistemi multicore/multiprocessore.
- quasi simultaneamente più di un processo chiama `enter_region()`.
  - In questo caso sembra che compaia un problema ma effettivamente non è cosi
  - Il primo processo che entra è quello che si acaparra l'utilizzo della CPU perchè comunque è garantita la proprietà "*In un singolo istante è presente solo un processo*"

# Istruzioni TSL e XCHG

Sono approcci che richiedono **supporto** da parte dell'**HARDWARE**.
**TSL**(*Test and Set Lock*): (operazione **ATOMICA** cioè non si può interrompere e ***blocca l'accesso al bus di memoria***)

- controlla il valore del lock, lo copia in un registro e lo incrementa

*In particolare*:

- vede se il lock era 0:
  - se era 0, lo **imposta** a 1, **accede** alla sezione critica
  - se era diversa da 0, **rimane in attesa** che il lock torni a 0
- Offre un **vantaggio** perchè sono operazioni indivisibili: cioè nessun'altro processo può accedere alla parte di memoria finchè non si sono terminate le operazioni
- quando entra nella sezione critica, **disabilita i bus della memoria** e quindi non c'è nessun'altra comunicazione e questo garantisce che se ci sono più core/processori, nessun altro processo può accedere alla risorsa condivisa
- fa uso della variabile **LOCK** che garantisce l'accesso/il non accesso alla parte condivisa

```
enter_region:
        TSL REGISTER,LOCK //copia nel registro il valore di lock e poi si deve incrementare
        CMP REGISTER,#0 // copia il valore 0 in REGISTER
        JNE enter_region // se NON si verifica la condizione (JNE) avviene un salto -> BUSY
WAITING
        RET

leave_region:
        MOVE LOCK,#0 //assegno 0 a LOCK
        RET
```

Anche in caso di utilizzo di **TSL** si ha **BUSY WAITING**, cioè il processo che non può entrare attende continuamente.

`enter_region` viene chiamata *PRIMA* di entrare nella sezione critica
`leave_region` viene chiamata *DOPO* aver finito le operazioni nella sezione critica

```
enter_region:
        MOVE REGISTER,#1
        XCHG REGISTER,LOCK //scambio i valori fra register e lock
        CMP REGISTER,#0
        JNE enter_region //iterazione se la condizione NON è verificata
        RET

leave_region:
        MOVE LOCK,#0
        RET
```

Con **XCHG** si ha sempre **BUSY WAITING**

L'idea è quella di creare **approcci alternativi** che **NON** consumino inutilmente l'unità di elaborazione, cioè **evitare il busy waiting**.

# Sleep e wakeup

Tutte le soluzioni viste fino ad ora fanno *SPIN LOCK*. Si ha il problema **dell'inversione di priorità.**

## Esempio:

- Se eseguo un processo con **priorità 3**, e nel frattempo arriva un processo con **priorità 5**, allora esso deve essere eseguito prima **stoppando** quello con priorità 3.
- Quello con priorità **più alta** prova a entrare nella sezione crtica ma **rimane bloccato** perchè comunque c'è il processo prima che ancora non è uscito.
- Quindi il processo con priorità più alta va in **BUSY WAITING** dipendendo da processi con priorità più bassa (**inversione di priorità**)

> Ci sono processi con priorità più alta *che vengono eseguiti per prima*
> In un sistema interattivo, la priorità ce l'ha chi deve dare la risposta all'utente
> I processi vengono eseguiti **in base alla priorità**.

## Soluzione al busy waiting

Dare la possibilità al processo di bloccarsi in **modo passivo** senza rimanere nel `WHILE` controllando ogni volta che la variabile cambi, quindi si ha la **RIMOZIONE DAI PROCESSI *READY*** e viene messo in una **coda a parte**.

- **NON** si ha più busy waiting
- *riduco la probabilità* che si verifichi l'inversione delle priorità
- si deve poter "*addormentare il processo*"

> Quando la parte condivisa si **libera**, si sveglia il processo mediante un segnale apposito e si rimanda in *READY*

- si usano le due chiamate di sistema **primitive** `sleep()` e `wakeup()`:
  - **sleep** viene chiamata se il processo **NON** può usare la risorsa condivisa. Allora viene addormentato e viene aggiunto a una **CODA DI DORMIENTI** (*rimosso dai processi READY* che competono per la *CPU*)
  - **wakeup** è il segnale che viene usato per risvegliare i processi dormienti (*e si scrive come ultma istruzione della* `leave_region`)

# 13-04-2023

## Problema produttore-consumatore

In questo caso 2 processi **condividono uno STESSO BUFFER** di una certa dimensione limitata (*N*). In particolare:

- il **PRODUTTORE** inserisce dati nel buffer
- il **CONSUMATORE** legge il dato prelevato dal buffer

```
count = 0
function producer()
        while (true) do
                item = produce_item() //crea un oggetto
                if (count = N) // count indica il numero di elementi memorizzati nel buffer
                        sleep()

                insert_item(item)
                count = count + 1 //incrementa il contatore

                if (count = 1)
                        wakeup(consumer) //serve per evitare di lasciare qualche consumer in
sleep() perchè se il buffer era vuoto, c'è la possibilità che un consumatore sia in sleep
```

```
function consumer()
        while (true) do
        if (count = 0) //non ci sono item nel buffer
                sleep()
        item = remove_item()
        count = count - 1

        if (count = N - 1)
                wakeup(producer) //risveglia un potenziale producer dormiente

        consume_item(item)
```

- Il **consumatore** legge solo se ci sono informazioni e quindi il buffer **NON** è vuoto
- il **produttore** scrive sul buffer e può scrivere **SOLO se c'è spazio**, quindi se il buffer non è pieno, altrimenti va in **SLEEP** fino a quando un consumatore toglie un elemento del buffer e quindi *libera spazio*.

## Limiti di questa soluzione:

- le istruzioni **non sono ATOMICHE**
- la count è slegata da sleep e wakeup e rappresenta la variabile condivisa
- Se il consumatore deve fare un'operazione con buffer vuoto, allora va in `sleep()` e viene **stoppato**
- Parte il produttore, legge count = 0, carica gli elementi e incrementa count -> vede count = 1 pensando che qualche consumatore stia dormendo e manda il segnale wakeup.

- Il consumatore riparte quando count = 0 e va in sleep dopo che il produttore ha fatto wakeup e il produttore continua a caricare
- Quindi il count vale 2,3... e non vengono eseguite altre `wakeup` di consumatore
- Ad una certa `count = N` e quindi il produttore va in sleep per buffer pieno
  Il produttore e il consumatore vanno in **sleep**
- Il problema viene sollevato perchè è stato inviato un `wakeup` inviato **A VUOTO**.
- Si risolve aggiungendo **un bit di attesa wakeup**:
  - se il consumer legge `count = 0` viene stoppato.
  - Il bit wakeup viene **settato a 1** quando viene **chiamata una wakeup dal produttore**.
  - Stavolta, il consumatore, *prima di andare in sleep*, **controlla il valore del bit wakeup** e se vale 1, allora vuol dire che è stato inviato un wakeup e non va in sleep
- Questa problematica viene risolta solo se c'è un produttore e uno/più consumatori. Se avessimo **più produttori** allora il **problema si risolleverebbe**.

# Semafori

Con ***DJKISTRA*** si ha una soluzione dove viene mantenuta una variabile che viene usata per contare il numero di *wakeup* inviate, detta **SEMAFORO** ed è una variabile condivisa fra più processi.

> *Inizialmente vale 0, cioè non sono stati inviati wakeup.*

Vengono suggerite 2 operazioni: **down** e **up** (dette anche **wait** e **signal**)

> In un certo senso, quindi, l'`if` che controlla `count` e l'operazione sleep che contiene, sono unite, evitando quindi la possibilità che l'`if` possa essere sospeso prima di fare la `sleep`. Quindi viene sospeso **PRIMA oppure DOPO**, garantendo **ATOMICITA'**

- `down` prende in input un *semaforo* e ne **controlla il valore**: (**ENTRA nella sezione critica**)
  - se è `>0` allora lo **DECREMENTA** e **accede alla sezione critica** e lo fa finchè il semaforo è `>0`
  - se è `<0` allora significa che la **sezione critica è occupata** e quindi va in `sleep`
  - questa istruzione **risveglia eventuali processi dormienti**
- `up` controlla il valore del semaforo e ne **INCREMENTA** il valore (**ESCE dalla sezione critica**) ed è l'operazione che viene fatta ***subito dopo essere usciti dalla sezione critica***

> Questo sistema **evita** il problema della **BUSY WAITING** e le operazioni sono ATOMICHE (*eseguite senza interruzione*) e quindi *gestisce* la ***MUTUA ESCLUSIONE***

## Problema produttore-consumatore con i semafori

Per gestire nel migliore dei modi questo problema, servono **3 semafori**:

1. uno in comune fra tutti che dice se l'operazione si può effettuare o no: nel buffer c'è il produttore oppure il consumatore quindi **garantisce la mutua esclusione**. (detto ***MUTEX***)
2. ***FULL*** = indica il **numero di posizioni piene** all'interno del buffer
3. ***EMPTY*** = indica il **numero di posizioni vuote** all'interno del buffer

Quindi, i vincoli sono:

- il produttore scrive **solo se c'è spazio**
- il consumatore legge **solo se c'è informazione**

```
int N=100
semaphore mutex = 1 //binario
semaphore empty = N
semaphore full = 0

function producer()
        while (true) do
                item = produce_item()
                down(empty) //se empty == 0 non ci sono posizioni vuote, quindi implica la sleep, altrimenti vuol dire che c'è spazio e quindi si possono inserire dati e decrementa empty
                down(mutex) // c'è qualcun'altro che sta usando il buffer. se >0 vuol dire che nessuno occupa il buffer. Se mutex = 0 vuiol dire che la sezione critica è occupata quindi va in sleep()
                insert_item(item)
                up(mutex) //segnala che lascia la condivisione del buffer
                up(full) // incremento il semaforo full di uno

function consumer() //preleva dati quando CI SONO dati
        while (true) do
                down(full) // full > 0 allora ci sono dati da prelevare e quindi continua, altrimenti sleep()
                down(mutex) // verifica se può accedere alla sezione critica
                item = remove_item()
                up(mutex) // indica che ha lasciato la sezione critica
                up(empty) // indica che c'è uno spazio vuoto in più perchè l'elemento viene rimosso

                consume_item(item) // viene usato l'item prelevato
```

## Osservazioni

- Vengono usati 3 semafori con **scopi diversi**
  - **mutex** gestisce la **MUTUA ESCLUSIONE**, sul buffer si lavora singolarmente, a maggior ragione quando ci sono più produttori e consumatori
  - **full** ed **empty** vengono usati per **SINCRONIZZARE** le operazioni fra produttore e consumatore visto che serve per capire se il produttore può operare oppure deve attendere un consumatore che gli permatta di poter operare
    - il produttore fa `down(empty)` (*quindi risveglia eventuali **consumatori addormentati***) mentre il consumatore fa `up(empty)` (*quindi risveglia eventuali **produttori addormentati***)
    - Il consumatore può accedere se `full > 1`: il produttore lo incrementa e il consumatore lo decrementa. Vale il viceversa per la variabile `empty`
- l'**ordine delle operazioni** sui **semafori** è **FONDAMENTALE**:
  - Se le operazioni iniziali di produttore e consumatore vengono invertire si potrebbe creare **DEADLOCK**
  - **MAI bloccare la sezione critica se non si è sicuri di poter eseguire l'operazione**

## Esempio di deadlock (inversione delle istruzioni)

```
int N=100
semaphore mutex = 1
semaphore empty = N
```

```
semaphore full = 0

function producer()
        while (true) do
                item = produce_item()
                down(mutex) //inversione
                down(empty) //inversione
                insert_item(item)
                up(mutex)
                up(full)

function consumer()
        while (true) do
                down(mutex) //inversione
                down(full) //inversione
                item = remove_item()
                up(mutex)
                up(empty)

                consume_item(item)
```

In questo caso:

- il produttore esegue down(mutex) e in quel momento è 1. Quindi viene portato a 0 e accede alla sezione critica.
- fa down(empty) e va in **sleep**
- il consumatore, quando accede alla sezione critica, trova `mutex = 0` quindi va in **sleep**.
- il produttore, con una *up(mutex)* dovrebbe *risvegliare il consumatore* ma è in sleep perchè l'ha causato down(empty) e il consumatore è in sleep per down(mutex).
- I due processi rimangono **bloccati per sempre** e si ha **DEADLOCK**

# 18-04-2023

## Analisi del semaforo MUTEX

- Sono utili nella **gestione delle risorse condivise** per la mutua esclusione.
- Sono validi quando si fa uso di **thread implementati nello spazio utente**:
  - **non vengono riconosciuti dal Kernel** e quindi non sono soggetti a scheduling
  - Eventuale **scheduling è gestito dallo stesso processo**
  - Quando un **thread si blocca**, **blocca l'intero processo** visto che il kernel non li conosce

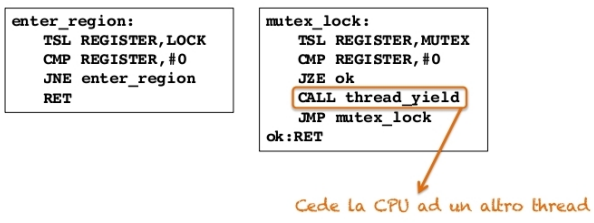
Mutex si può trovare in 2 stati: **locked** (valore 1) e **unlocked** (valore 0)
Si usa `mutex_lock()` e `mutex_unlock()`:

```
mutex_lock:
        TSL REGISTER,MUTEX -- Mutex viene copiata in REGISTER e viene incrementato il valore
        CMP REGISTER,#0
        JZE ok --Accede alla sezione critica se MUTEX = 0
        CALL thread_yield
        JMP mutex_lock
ok:RET


mutex_unlock:
        MOVE MUTEX,#0
        RET
```

Sono molto simili a enter_region/leave_region ma:

- devono essere possibili l'uso delle istruzioni TSL/XCHG visto che si è a **livello utente**
- Non si ha busy waiting se si usa MUTEX

## Differenze fra le 2 versioni:




- **Sinistra**: Se il CMP fallisce allora avviene l'iterazione infinita fino a quando viene soddisfatta la condizione
- **Destra**: *I thread a livello utente non sono conosciuti dal kernel, allora non sono soggetti a scheduling dal kernel ma è fatto dal processo stesso*.
  - Si ha un **vantaggio** perchè si può usare uno **scheduling personalizzato** ma **il processo non ha il clock** che ha il kernel e quindi non si può decidere per quanto tempo eseguire un processo. **E' il processo stesso a rilasciare la CPU volontariamente**

- Se si trova **MUTEX occupato** (locked), allora si esegue `CALL thread_yield` ed è la chiamata di sistema che **rilascia la CPU volontariamente** e **viene eseguito un altro thread dello stesso processo.**
- Questa operazione è veloce perchè **gestita dallo stesso processo**

*Quindi `mutex_lock` e `mutex_unlock` non fanno chiamate al kernel, quindi si **velocizzano le varie operazion**i. Esse sono molto utili a **LIVELLO UTENTE***

## Futex

Gli spin-lock, pur consumando la CPU, può essere breve ma se non lo è si ha lo spreco dell'unità di elaborazione
Futex è una caratteristica tipica in Linux:

- un sistema è in grado di gestire numerose contese nelle parti condivise e il kernel si occupa di risvegliarlo
- diventa efficiente quando le contese sono frequenti. Se non lo fossero, le chiamate al kernel (più costose) penalizzano e aumentano il costo
- implementa un sistema lock, simile a MUTEX, senza attivare il kernel e quindi non coninvolgerlo quasi mai
- E' formato da 2 parti:
  - **servizio kernel**: che fornisce una coda di attesa che consente a più processi di andare in pausa (lock). E' il kernel stesso che li sblocca in **maniera esplicita**. Attraverso una chiamata di sistema si fanno tali operazioni.
  - **libreria utente**:

**FUTEX** agisce interamente nello spazio utente:

- se `lock = 1`, un thread acquisisce il lock e accede alla sezione critica
- quindi lock viene decrementato così da garantire la mutua esclusione
- Se lock è già occupato da un altro, quindi vale 0, allora la libreria FUTEX, **piuttosto che fare busy waiting**, allora questo thread **viene messo nella coda di attesa** che si trova nel kernel tramite *chiamata di sistema*
- Quando lock = 1 (*di nuovo*), allora il kernel **risveglia** il thread che è stato messo in attesa

> *Se non ci sono contese alla variabile lock, allora non ci sono chiamate al kernel e quindi le esecuzioni sono più veloci*
> *Se ho contese alla variabile lock, allora probabilmente devo fare chiamate di sistema al kernel*

# Monitor

Soluzione efficiente per gestire la ***mutua esclusione***:

- è una *raccolta di variabili*, procedure ecc che sono raggruppate in un **PACCHETTO**
- sono dei *costrutti* in un determinato linguaggio di programmazione

> *Per esempio, con la **OOP** ci sono parti private (variabili, strutture dati e qualche metodo) e pubbliche (tutto il resto)*

Quando si vuole accedere a parti private di un oggetto si ottiene un errore visto che esse possono essere lette solo dai metodi pubblici nello stesso oggetto.

La stessa cosa avviene con i **Monitor**:

- Il compilatore sa che la struttura dati (l'insieme dei dati) è una parte speciale e quindi va **gestita in modo diverso rispetto ad altri metodi/procedure**
- **Tutti i processi** possono chiamare le **procedure di un monitor** ma **non si può accedere alla struttura interna** di questo mondo se non attraverso le procedure offerte dal monitor stesso
- Per ottenere una **corretta mutua esclusione**, in ogni istante **può essere attivo un solo processo alla volta** all'interno di questo mondo
- La prima istruzione, infatti, verifica che non ci sia nessun altro processo che sia attivo
- Se nessun processo sta usando il monitor, allora quel processo stesso può usare le procedure del monitor

# Mutua esclusione

La mutua esclusione è **gestita dal compilatore** e non dal programmatore visto che viene gestita tramite una **MUTEX** o tramite un **semaforo binario**

Con i monitor è possibile **gestire le race conditions** convertendo le regioni critiche in **procedure di monitor** -> si ha che **MAI due processi si trovano allo stesso tempo in esecuzione all'interno del monitor**.

> *Nel caso produttore-consumatore come si blocca un processo che non può proseguire? (buffer pieno) Come si gestisce la sincronizzazione?*

- si usano **variabili condizioni**, insieme alle operazioni **wait** e **signal**, permettono di gestire la sincronizzazione delle operazioni
- si deve gestire quando un processo può eseguire la sua operazione e quando non è possibile.
  - La mutua esclusione è gestita dai monitor in modo automatico tramite la ***conversione delle regione critiche in procedure del monitor stesso***
- la **wait sospende il processo chiamante** finchè non viene invocata la sua alternativa, cioè la **signal che risveglia il processo addormentato da wait**
  - Se viene invocata **signal** e non ci sono processi bloccati, allora non si avranno effetti

## Produttore-consumatore

Si deve gestire che il produttore può eseguire la sua operazione finchè il buffer non è pieno e viceversa per il consumatore.

- Si esegue una wait su una variabile condizione (per esempio la `full` che si era usata come semaforo). Se è piena allora il processo chiamante viene bloccato e in questo caso si può consentire agli altri di proseguire le proprie operazioni.
- quindi il consumatore può operare che, più avanti, userà **signal** sulla variabile condizione (**full**)

> *Cosa accade dopo signal? per evitare che **2 processi siano attivi allo stesso tempo nel monitor**, allora si deve decidere chi deve continuare la sua operazioni dopo la signal.*

1. Il **metodo Hoare**: il processo che deve essere eseguito è quello **APPENA SVEGLIATO**
2. Il **metodo Brinch-Hansen**: il processo che esegue signal deve **uscire subito dal monitor** e rappresenta proprio **la sua ultima istruzione** che appare all'interno di quel processo. Se viene fatto una signal come ultima istruzione di una procedura su cui **ci sono in attesa più processi** su questa

variabile condizione, verrà **selezionata uno di questo** "*in attesa*" come **nuovo processo risvegliato**

3. L'ultimo metodo: si **manda la signal** ma **in esecuzione continua a rimanere chi ha eseguito la signal**. Chi si è appena svegliato **continua ad aspettare che il chiamante termini la sua esecuzione**

> *La wait deve arrivare prima di una signal altrimenti verrà persa*

## DIfferenze cruciali fra sleep, wakeup

Quando la wakeup andava a vuoto, con i monitor questo non può succedere perchè la mutua esclusione è garantita dalle stesse procedure dei monitor

- `wait` -> ***sospende*** un processo (in caso di regione critica occupata) finchè non viene riattivato
- `signal` -> ***risveglia*** un processo sospeso, quindi in `wait()`. Se non c'è nessun processo in wait allora questa chiamata va a vuoto

*Esempio*: *Dato un processo P che esegue una signal(x) e un processo Q che è in sleep(x). Entrambi i processi possono andare in esecuzione ma chi ci va? Continua P o si esegue Q?*

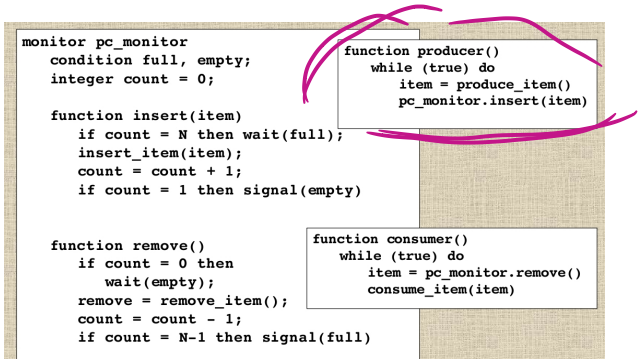
- **Hoare**: **signal & wait** -> P viene sospeso (sleep) e Q va in esecuzione
- **Mesa**: **signal & continue** -> Q riceve la signal ma aspetta che P termina la sua operazione
- **Compromesso**: **signal & return** -> usato con *concurrent Pascal* cioè: P fa signal, lascia il monitor a Q che viene immediatamente ripreso

> Molti linguaggi di programmazione gestiscono i monitor (C#, Java) mentre altri non hanno i monitor (C) e in questo caso va implementato diversamente.
> *In Java si usa la parola chiave **SYNCHRONIZED** per identificare un metodo monitor e vuol dire che nessun altro thread può eseguire un altro metodo synchronized dello stesso oggetto*

## Svantaggi dei monitor

- Non sono gestiti da tutti i linguaggi di programmazione e quindi non sono gestiti dal compilatore

## Esempio produttore-consumatore con i monitor




- **count** gestisce la **variabile condizione** e indica il *numero di elementi nel buffer*
- Si usano 4 metodi di una classe *pc_monitor*
- Viene usato il **modello Hoare**, cioè **signal & wait**
- Si usano le variabili condizione `full` e `empty` -> *full* serve per sospendere il produttore, *empty* serve per sospendere il consumatore

*In questo esempio si è applicato il metodo Brinch-Hansen*

# 27-04-2023

# Scambio di messaggi fra processi

A differenza degli altri metodi, questo metodo permette di mettere in comunicazione processi fra ***macchine diverse***.

Vengono usate **2 chiamate di sistema** e non sono costrutti del linguaggio di programmazione:

- **send** ->Invia un messaggio. `send(destinazione,messaggio)`
- **receive** -> Riceve un messaggio. `receive(sorgente,messaggio)`
  - La receive è attiva se deve riceve messaggi, altrimenti non lo è.

***Vantaggio*** di questo approccio: permette di far **comunicare**/gestire **processi fra macchine diverse** (ovviamente collegate fra loro)

***Svantaggio*** di questo approccio: rispetto ai semafori/monitor, vengono **usate chiamate di sistema**, quindi risultano essere meno efficienti proprio per questo motivo

*Il messaggio potrebbe ***perdersi nella rete durante la communicazione***.* In questo caso si potrebbe usare un meccanismo del tipo: quando un processo riceve un messaggio, si rimanda (all'indietro) un **messaggio di conferma**. Se il messaggio di conferma non è stato ricevuto, allora il messaggio viene rispedito.

Questa soluzione non è definitiva perchè ***si può perdere anche un messaggio di conferma.*** Bisogna distinguere un messaggio ***ORIGINALE*** da un messaggio ***RITRASMESSO*** tramite un *codice identificativo*

## Metodi di indirizzamento (scambio messaggio)

***Diretto*** -> Impostazione di un id ad un messaggio e usare questo per comunicare in maniera uniovoca.
***Maibox*** -> si usa un buffer che memorizza un numero $n$ limitato di messaggi, e vengono usati per invio/ricezione di questi **messaggi spediti ma non ricevuti**

- se la mailbox è **piena**, il **processo viene sospeso** finchè questo buffer non viene svuotato di almeno una posizione
- Rendezvous -> evita l'uso del buffer. Destinazione e Sorgente sono collegate insieme

Lo scambio di messaggi può essere in 2 modi:

- ***ASINCRONA***: Quando un processo invia un messaggio, **continua la sua esecuzione**.
  - In questo caso, il processo invia un messaggio, anche se il destinatario non l'ha ricevuto, esso continua ad andare avanti. Chi riceve il messaggio può non cambiare nulla o l'informazione che riceve contiene delle ***informazioni non valide*** rispetto a quando è stato inviato (*visto che il mittente è andato avanti con la sua esecuzione*)
- ***SINCRONA*** (meglio nota come rendezvous): Quando viene inviato un messaggio, il mittente **si sospende**
  - In questo caso il messaggio può essere inviato **SOLO** se il ricevente è pronto a riceverlo
  - Il messaggio spedito contiene informazioni che riguardano lo stato attuale del mittente sospeso

- Non vi è necessità della presenza di un buffer visto che i messaggi non si accumulano. Dopo la spedizione di un messaggio, il processo mittente si sospende e, quindi, ***non accumula messaggi da inviare***.

## Problema produttore-consumatore con scambio di messaggi

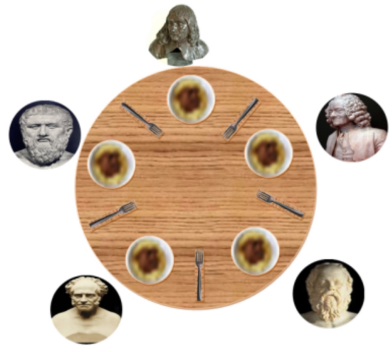
```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                                 /* message buffer */

     while (TRUE) {
          item = produce_item( );               /* generate something to put in buffer */
          receive(consumer, &m);                /* wait for an empty to arrive */
          build_message(&m, item);              /* construct a message to send */
          send(consumer, &m);                   /* send item to consumer */
     }
}

void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
     while (TRUE) {
          receive(producer, &m);                /* get message containing item */
          item = extract_item(&m);              /* extract item from message */
          send(producer, &m);                   /* send back empty reply */
          consume_item(item);                   /* do something with the item */
     }
}
```

Può capitare che uno dei due più veloce rispetto all'altro e quindi si potrebbero verificare letture/invii di messaggi vuoti.

# Problema dei 5 filosofi a cena

RIguarda il problema della sincronizzazione:

- Si hanno 5 filosofi a tavola dove ci sono 5 piatti di spaghetti e ogni filosofo ne ha uno.
- Ognuno di loro hanno una forchetta e gli spaghetti sono scivolosi che richiedono due forchette per essere mangiati.
- Fra un piatto e un altro c'è solo una forchetta
- Quando un filosofo mangia, usa le 2 forchette ai lati e dopo aver finito le posa.
- Ogni filosofo si limita a pensare e mangiare



> *E' possibile scrivere un algoritmo per ogni filosofo in modo che non si blocchi mai?*

## Prima soluzione

Per poter mangiare, un filosofo deve avere **l'uso esclusivo** di entrambe le forchette. Le risorse(forchette) sono limitate e il "prendere una forchetta" è in concorrenza con i filosofi che si hanno accanto.

```
int N=5
function philosopher(int i)
        think()
        take_fork(i) //prende la forchetta destra
        take_fork((i+1) mod N) //prende la forchetta sinistra
        eat() //mangia
        put_fork(i) //posa la forchetta
        put_fork((i+1) mod N)
```

Se tutti volessero mangiare allo stesso tempo:

- Se tutti prendono la forchetta a sinistra, quella a destra non c'è, allora posano la forchetta e tornano a pensare.
- Dopo un po' si ripete la situazione e nessuno mangia.
- Si ha uno stato di deadlock e nessuno mangia

## Seconda soluzione

Si controlla se una forchetta è disponibile. Se non lo è viene rilasciata e si riprova in seguito

> *Le due soluzioni non sono corrette e portaono (o possono portare) a un deadlock.*

La prima soluzione (dove ognuno prende la forchetta sinistra allo stesso tempo per poi posarla e non mangiare) viene detta ***STARVATION***: i **programmi** vengono **eseguiti** indefinitamente **senza avanzare mai con il proprio stato**.

## Terza soluzione

Ripetere l'ipotesi della prima soluzione, dove il problema era il fare l'azione nello stesso tempo, allora si inserisce un'attesa random fra le interazioni.

- un filosofo prende la forchetta sinistra, vede se c'è a destra e se non c'è la posa e aspetta un tempo random.
- Nella maggior parte dei casi un filosofo prende la forchetta in tempi diversi rispetto ad altri. Almeno un filosofo, teoricamente, dovrebbe prendere entrambe le forchette e quindi mangiare.
- C'è la **POSSIBILITA'** che un filosofo prenda 2 forchette e mangi ma non c'è **CERTEZZA**, quindi non è un approccio valido da poter usare, quindi serve una soluzioni che vada bene ***sempre***.

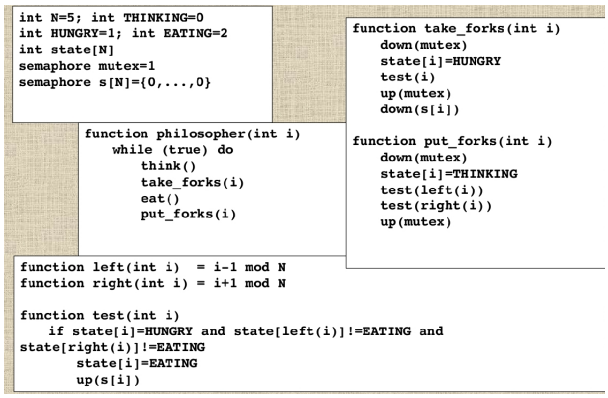
## Quarta soluzione

Uso di un semaforo MUTEX:

- Un filosofo, prima di prende la forchetta, controlla un semaforo e in base al valore prende/non prende la forchetta
- Si garantisce il possesso delle forchette se sono libere e quindi può un filosofo può mangiare
- Dopo aver mangiato, il MUTEX torna allo stato iniziale

Questa soluzione non è del tutto efficiente:

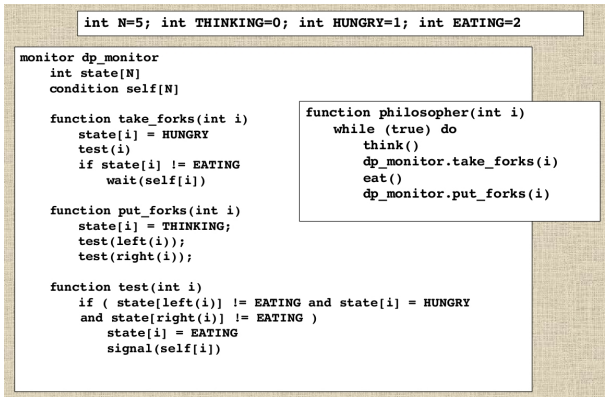
- Teoricamente può funzionare ma con l'uso del semaforo **solo 2 filosofi possono mangiare allo stesso tempo** quindi questa soluzione **NON E' OTTIMALE**




- Un filosofo può trovarsi in 3 stati: *Pensante*, *Ho Fame*, *Mangiando*
- Si usa un array che contiene lo stato dell'i-esimo filosofo e un vettore di semafori, uno per ogni filosofo

L'idea è: il filosofo è nel suo stato Pensante. Può passare allo stato Eating solo se il suo filosofo sinistro e destro non stanno mangiando

## Problema dei 5 filosofi con i monitor




- Si fa uso di un array di contition che è grande quanto il numero dei filosofi

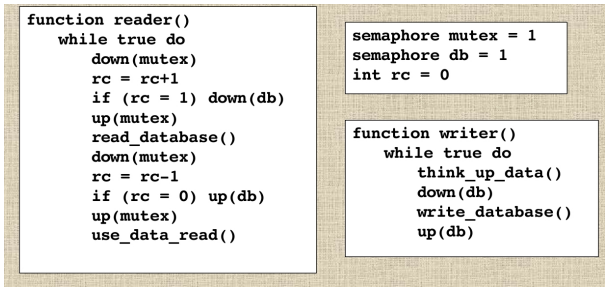
# Problema dei lettori-scrittori

E' un altro **problema di sincronizzazione** che gestisce l'acesso all'interno di un database.
Quando si devono solo leggere dati dal database (*processi di lettura*), allora non ci sono problemi.
Quando qualcuno vuole scrivere sul database (*processi di scrittura*) allora possono nascere dei problemi.

- Se un processo sta scrivendo nel database, nessun'altro processo (lettura e scrittura) può accedere al database in quell'istante.
- Se ci sono processi di lettura, essi potrebbero leggere tutti allo stesso tempo.

*Si può programmare un algoritmo che, quando c'è uno scrittore, nessun altro può acceder e se nessuno scritture sta scrivendo allora tutti i lettori possono leggere allo stesso tempo?*

## Soluzione con i semafori



```
function reader()
   while true do
      down(mutex)
      rc = rc+1
      if (rc = 1) down(db)
      up(mutex)
      read_database()
      down(mutex)
      rc = rc-1
      if (rc = 0) up(db)
      up(mutex)
      use_data_read()
```

```
semaphore mutex = 1
semaphore db = 1
int rc = 0
```

```
function writer()
   while true do
      think_up_data()
      down(db)
      write_database()
      up(db)
```

- `db` gestisce gli accessi al database
- `rc` (***variabile condivisa**, quindi modificata da tutti) si usa come contatore -> si contano il numero di processi che stanno leggendo o che vorrebbero leggere il database (*hanno fatto richiesta di lettura*)
- `mutex` gestisce gli accessi esclusivi alla variabile condivisa `rc`

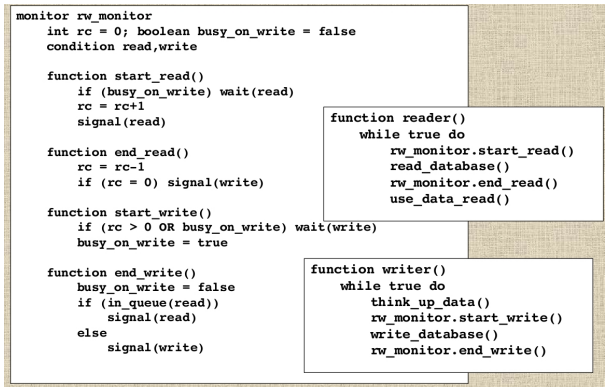
## Problema

- Si sta consentendo a ***più lettori di accedere allo stesso tempo mentre lo scrittore deve aspettare che il lettore termini le sue operazioni***
  - Potrebbe arrivare un altro lettore e, visto questo codice, legge il contenuto del database. Man mano `rc` incrementa
- **All'arrivo di uno scrittore**, trova $db = 0$ e quindi non può accedere e va in sleep() e **aspetta che il lettore/i lettori escano dal database**. (potebbe attendere un tempo indefinito)
  - Quando $rc = 0$ allora nessun lettore sta leggendo il database né tantomeno ha espresso l'intenzione di farlo.

## Alternativa

Lo scrittore potrebbe ***attendere solo i lettori che lo precedono*** e quindi **NON** qualsiasi lettore. In questo caso si ha lo svantaggio di avere **MINORI PRESTAZIONI**

Soluzione con monitor ma che presenta lo stesso problema: (**PRIORITA' AI LETTORI**)



```
monitor rw_monitor
    int rc = 0; boolean busy_on_write = false
    condition read,write

    function start_read()
        if (busy_on_write) wait(read)
        rc = rc+1
        signal(read)

    function end_read()
        rc = rc-1
        if (rc = 0) signal(write)

    function start_write()
        if (rc > 0 OR busy_on_write) wait(write)
        busy_on_write = true

    function end_write()
        busy_on_write = false
        if (in_queue(read))
            signal(read)
        else
            signal(write)
```

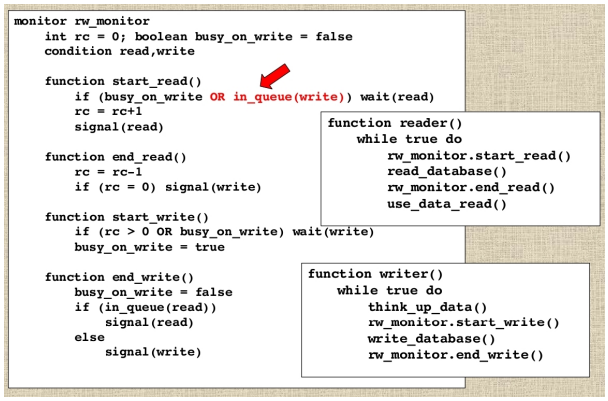
```
function reader()
    while true do
        rw_monitor.start_read()
        read_database()
        rw_monitor.end_read()
        use_data_read()
```

```
function writer()
    while true do
        think_up_data()
        rw_monitor.start_write()
        write_database()
        rw_monitor.end_write()
```

Anche in questo caso gli scrittori restano in attesa che tutti i lettori abbiano finito le proprie operazioni

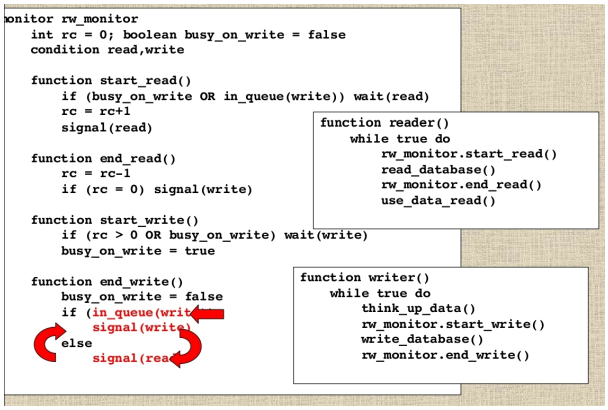
Soluzione che gestisce i successivi lettori che vengono messi in coda rispetto ad uno scrittore già presente. (**LO SCRITTORE RIMANE IN ATTESA SOLO DEI LETTORI CHE LO PRECEDONO**)
Un lettore si blocca quando il database è bloccato o quando c'è una coda di processi scrittori.

- Uno scrittore attende solo i lettori che lo precedono




Soluzione che accoda i lettori finchè c'è una coda di scrittori. (**I LETTORI ATTENDONO TUTTI I POSSIBILI SCRITTORI**)

- Quindi quando c'è una coda di ***scrittori che attendono***, allora i lettori attendono che essa si svuoti




- In questo caso si potrebbe avere una STARVATION sui lettori che attendono tutti i possibili scrittori (anche i successivi) e quindi, probabilmente, non verranno mai eseguiti

# Scheduler dei processi

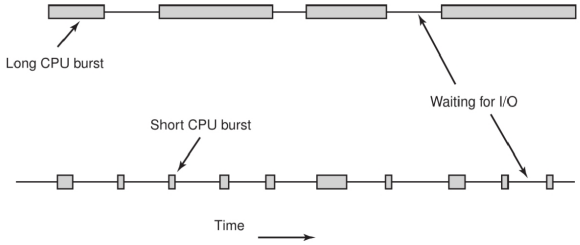
Gestire e determinare l'ordine di esecuzione dei processi.

- In un intervallo di tempo devono **avanzare più processi** e di questo lavoro se ne occupa lo ***SCHEDULER***.
- La tecnica che determina l'ordine esatto prende il nome di ***ALGORITMO DI SCHEDULING***.
- L'algoritmo di scheduling deve tener conto di alcune **caratteristiche**:
  - **Ottimizzazione le prestazioni** al fine di raggiungere l'obiettivo richiesto
  - **Uso efficiente** dell'unità di elaborazione, ovvero *non ci devono essere momenti nei quali la CPU non viene sfruttata*
  - Finalità della macchina: **INDUSTRIALE** (*eseguire più velocemente possibile le azioni richieste*), **INTERATTIVA** (*rispondere nel minor tempo possibile alle richieste dell'utente*) o sistemi **REAL-TIME** (*la risposta del processo deve essere istantanea*)

Ci sono diversi **tipologie di processi**:

- **CPU BOUNDED** -> la maggior parte dell'attività del processo è passata all'interno della CPU, quindi sono processi ***ATTIVI*** e fanno *richieste I/O molto raramente*
- **I/O BOUNDED** -> la maggior parte della loro attività la passano ***IN ATTESA*** di I/O per poi tornano allo stato precedente e consumano il loro tempo di esecuzione in un ***BREVE INTERVALLO***. Inoltre

fanno *richieste I/O molto frequentemente*




> *Dare **troppa priorità** a una tipologia di un processo oppure ad un altra tipologia di processo **non è la scelta migliore** da prendere per l'algoritmo di scheduling*.
> Invece si deve trovare un **corretto bilanciamento** fra questi due tipi di processi in modo da ottenere le prestazioni migliori.

Lo scheduler viene attivato per decidere quale processo deve essere eseguito successivamente, in particolare:

- alla **creazione/terminazione** di un processo ( `fork()` )
- per la ricezione di **chiamate bloccanti** (*sezioni critiche*) e arrivo del relativo interrupt
- per il **blocco I/O**

Tipologie di sistemi in caso di **interrupt periodici**:

- **Preemptive**: lo scheduling **preleva, sospende (*di forza*)** l'esecuzione di un processo e lo sostituisce con un altro
- **Non-Preemptive**: lo scheduling lascia che il processo finisca le sue attività **senza prelevarlo forzatamente**

Un processo ha un `quanto di tempo` *assegnato* ed è sempre minore del tempo complessivo che il processo richiede per essere completato correttamente.

- L'algoritmo di assegnazione di tempo ad un processo è detto ***ALGORITMO PREEMPTIVE*** e tale processo è **obbligato a essere sospeso**. Esso hanno un **INTERRUPT DI CLOCK** che interrompono forzatamente il processo
- Un ***ALGORITMO NON-PREEMPTIVE*** è il viceversa del *PREEMPTIVE* e un processo non viene sospeso finchè non completa le sue operazioni

Il **DISPATCHER** è modulo che da il modulo della cpu al processo selezionato dallo scheduler. Si occupa del context_switch, del cambio fra esecuzione `usermode` o `kernelmode` e/o eseguire salti di riavvio di programma in `usermode`. Il Dispatcher deve essere molto veloce per ridurre i costi e i tempi.
La **latenza di dispatch** è l'intervallo di tempo del dispatcher per interrompere e successivamente avviare un nuovo processo

> - In un **contesto industriale NON** è necessario interrompere un processo ma in linea generale è importante che il processo completi la sua operazione e quindi si ***tenta di usare un algoritmo non-preemptive***
> - Viceversa accade in un **contesto interattivo**, invece, **può risultare interessante interrompere un processo** e, quindi, si tenta di usare un ***algoritmo di scheduling preemptive*** per permettere delle risposte istantanee all'utente che fa le richiest