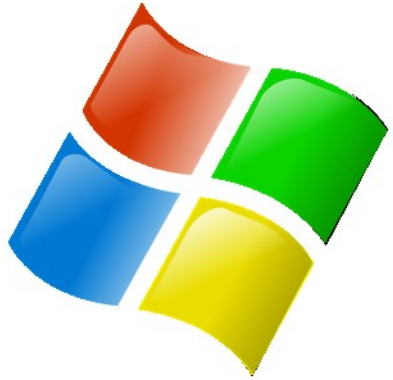
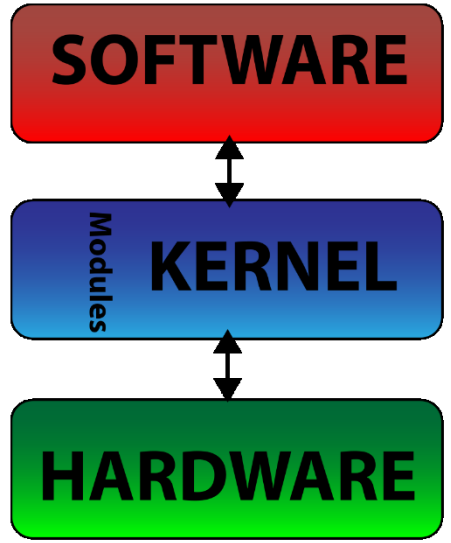
UNIVERSITÀ DEGLI STUDI DELLA BASILICATA



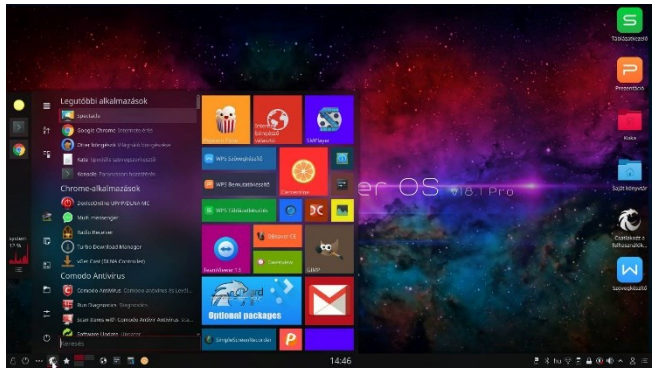
Corso di Sistemi Operativi
A.A. 2019/20

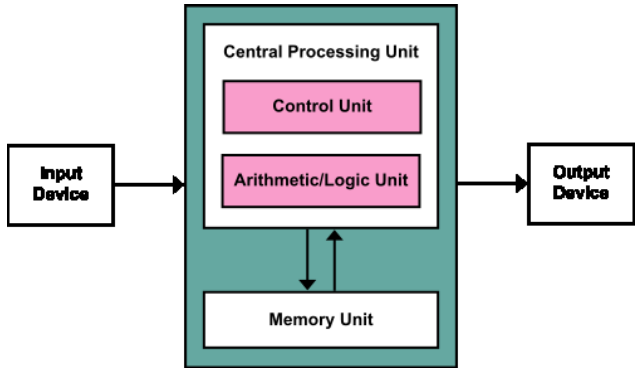


Sincronizzazione dei processi

Docente:
Domenico Daniele Bloisi

Ottobre 2019

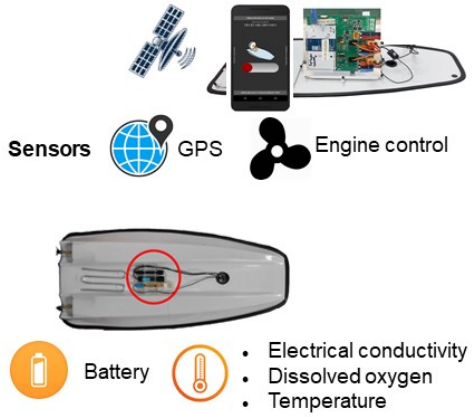


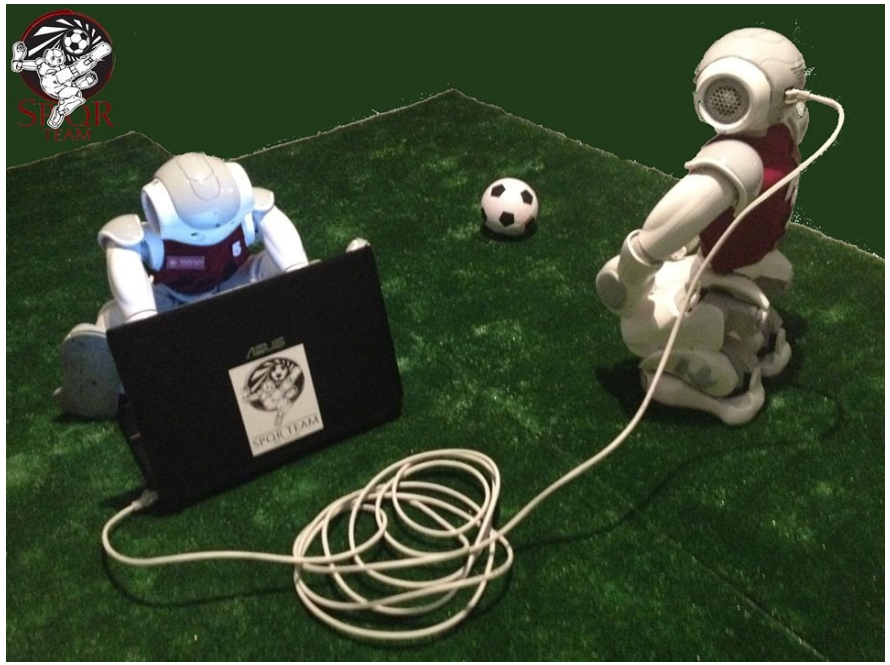


Domenico Daniele Bloisi

- Ricercatore RTD B
  Dipartimento di Matematica, Informatica ed Economia
  Università degli studi della Basilicata
  http://web.unibas.it/bloisi

- SPQR Robot Soccer Team
  Dipartimento di Informatica, Automatica e Gestionale Università degli studi di Roma “La Sapienza”
  http://spqr.diag.uniroma1.it





Ricevimento

- In aula, subito dopo le lezioni
- Martedì dalle 11:00 alle 13:00 presso:
  Campus di Macchia Romana
  Edificio 3D (Dipartimento di Matematica, Informatica ed Economia)
  II piano, stanza 15

Email: domenico.bloisi@unibas.it

Programma – Sistemi Operativi

- Introduzione ai sistemi operativi
- Gestione dei processi
- Sincronizzazione dei processi
- Gestione della memoria centrale
- Gestione della memoria di massa
- File system
- Sicurezza e protezione

Processo cooperante

Un **processo cooperante** è un processo che:

- può influenzare un altro processo in esecuzione nel sistema
- può subire l'influenza di un altro processo in esecuzione nel sistema

I processi cooperanti possono condividere:

- uno spazio logico di indirizzi (codice e dati)
- solo dati attraverso file o messaggi

Race condition

- Una **race condition** si verifica quando i processi hanno accesso concorrente ai dati condivisi e il risultato finale dipende dal particolare ordine in cui si verificano gli accessi.

- Le **race condition** possono portare a valori corrotti dei dati condivisi.

Race condition

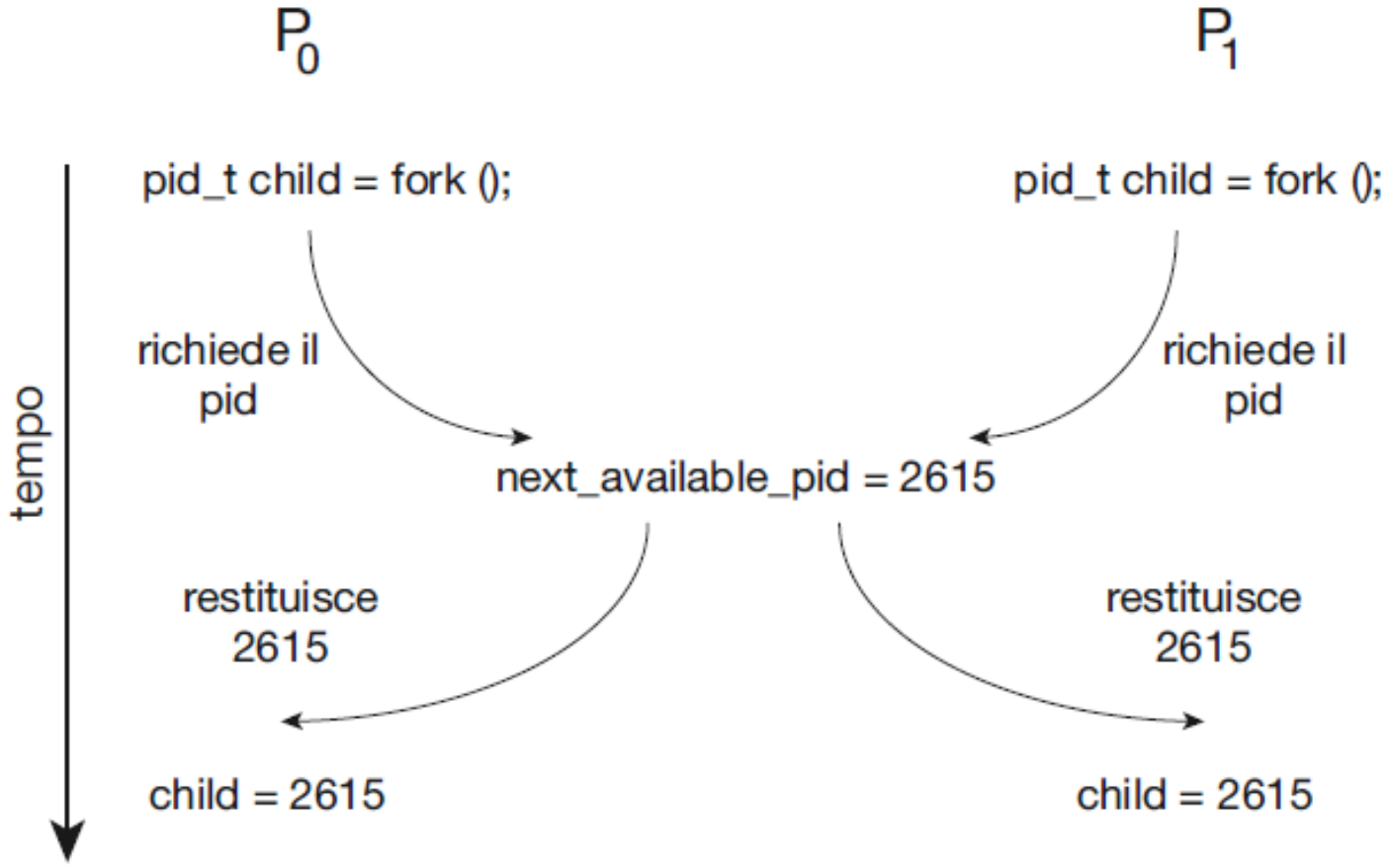


Figura 6.2 Race condition durante l'assegnamento del pid.

Race condition

Supponiamo di avere la seguente situazione:

Variabile condivisa `counter` che vale 5

Processo 1 (produttore)	Processo 2 (consumatore)
`counter++`	`counter--`

Se Processo 1 e Processo 2 sono in esecuzione concorrente, quale sarà il valore di `counter` al termine dell'esecuzione dei due processi?

Race condition

- **counter++** could be implemented as
  register1 = counter
  register1 = register1 + 1
  counter = register1
- **counter--** could be implemented as
  register2 = counter
  register2 = register2 - 1
  counter = register2
- Consider this execution interleaving with "count = 5" initially:

  S0: producer execute **register1 = counter** {register1 = 5}
  S1: producer execute **register1 = register1 + 1** {register1 = 6}
  S2: consumer execute **register2 = counter** {register2 = 5}
  S3: consumer execute **register2 = register2 – 1** {register2 = 4}
  S4: producer execute **counter = register1** {counter = 6 }
  S5: consumer execute **counter = register2** {counter = 4}

Race condition

- How do we solve the race condition?
- We need to make sure that:
  - The execution of

    counter++

    is done as an “atomic” action. That is, while it is being executed, no other instruction can be executed concurrently.

    - actually no other instruction can access **counter**
  - Similarly for

    counter--
- The ability to execute an instruction, or a number of instructions, atomically is crucial for being able to solve many of the synchronization problems.

Sezione critica

Una **sezione critica (CS)** è una porzione di codice in cui i dati condivisi possono essere manipolati

Quando un processo è in esecuzione nella propria sezione critica, non si consente ad alcun altro processo di essere in esecuzione nella propria sezione critica

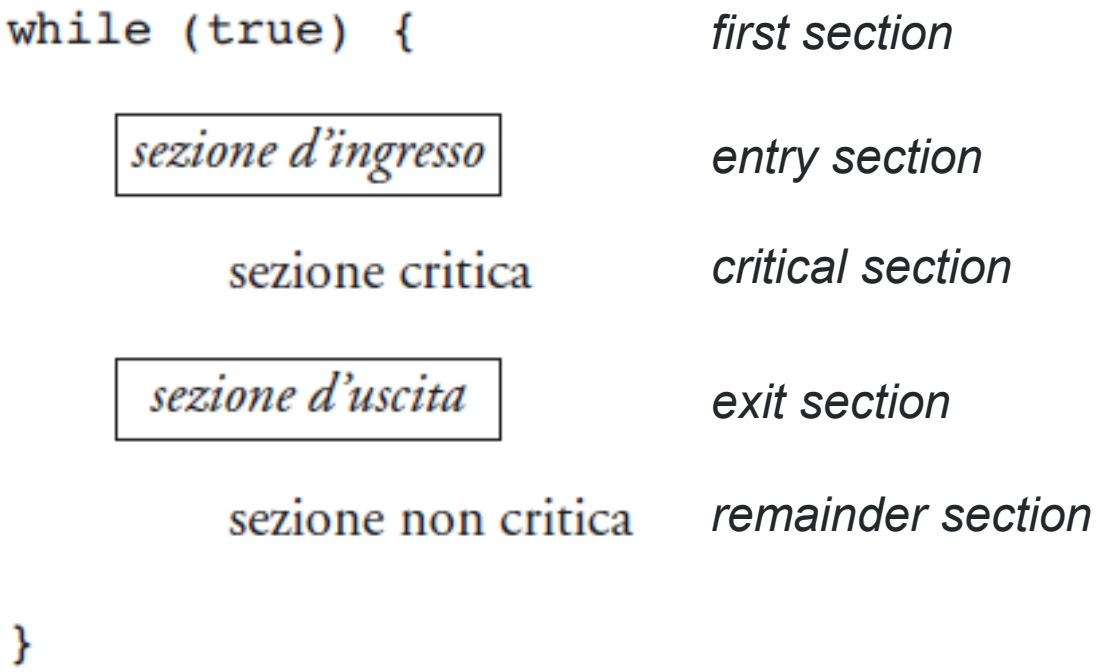


Figura 6.1 Struttura generale di un tipico processo.

Problema della sezione critica

- Il problema della **sezione critica** consiste nel progettare un protocollo che i processi possano usare per cooperare.

- Ogni processo deve chiedere il permesso per entrare nella propria **sezione critica.**

Soluzione al problema della sezione critica

Una soluzione del problema della sezione critica deve soddisfare i tre seguenti requisiti.

- Mutua esclusione
- Progresso
- Attesa limitata

Mutua esclusione

Se un processo P_i è in esecuzione nella propria sezione critica, nessun altro processo P_j può essere in esecuzione nella propria sezione critica

Progresso

Se nessun processo è in esecuzione nella propria sezione critica e qualche processo desidera entrare nella propria sezione critica, solo i processi che si trovano fuori dalle rispettive sezioni non critiche possono partecipare alla decisione riguardante la scelta del processo che può entrare per primo nella propria sezione critica

Questa scelta non si può rimandare indefinitamente

Progresso

Progress: If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

If no process is executing in its critical section

If there is a process executing in its critical section (even though not stated explicitly, this includes the leave section as well), then this means that some work is getting done. So we are making progress. Otherwise, if this was not the case...

https://stackoverflow.com/questions/33143779/what-is-progress-and-bounded-waiting-in-critical-section

Progresso

and some processes wish to enter their critical sections

If no process wants to enter their critical sections, then there is no more work to do. Otherwise, if there is at least one process that wishes to enter its critical section...

then only those processes that are not executing in their remainder section

This means we are talking about those processes that are executing in either of the first two sections (remember, no process is executing in its critical section or the leave section)...

https://stackoverflow.com/questions/33143779/what-is-progress-and-bounded-waiting-in-critical-section

Progresso

can participate in deciding which will enter its critical section next,
Since there is at least one process that wishes to enter its CS, somehow we must choose one of them to enter its CS. But who's going to make this decision? Those process who already requested permission to enter their critical sections have the right to participate in making this decision. In addition, those processes that may wish to enter their CSs but have not yet requested the permission to do so (this means that they are in executing in the first section) also have the right to participate in making this decision.

https://stackoverflow.com/questions/33143779/what-is-progress-and-bounded-waiting-in-critical-section

Progresso

and this selection cannot be postponed indefinitely.
This states that it will take a limited amount of time to select a process to enter its CS. In particular, no deadlock or livelock will occur. So after this limited amount of time, a process will enter its CS and do some work, thereby making progress.

https://stackoverflow.com/questions/33143779/what-is-progress-and-bounded-waiting-in-critical-section

Attesa limitata

Se un processo P_i ha già richiesto l'ingresso nella propria sezione critica, esiste un limite al numero di volte che si consente ad altri processi di entrare nelle rispettive sezioni critiche prima che si accolga la richiesta di P_i

Gestione delle sezioni critiche

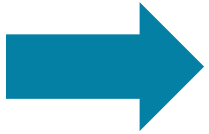
La due strategie principali per la gestione delle sezioni critiche nei sistemi operativi sono:

1. **kernel con diritto di prelazione**

2. **kernel senza diritto di prelazione**

Kernel con diritto di prelazione

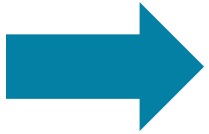
kernel con diritto di prelazione



Consente che un processo funzionante in modalità di sistema sia sottoposto a prelazione, rinviandone in tal modo l'esecuzione. NON è immune da race condition

Kernel senza diritto di prelazione

kernel senza diritto di prelazione



non consente di applicare la prelazione a un processo attivo in modalità di sistema. È immune da race condition

Soluzione di Peterson

- La soluzione di Peterson è una soluzione software al problema della sezione critica
- È limitata a due processi, P_0 e P_1, ognuno dei quali esegue alternativamente la propria sezione critica e la sezione non critica.
- P_0 e P_1 condividono

```
int turn;
boolean flag[2];
```

indica se un processo è pronto a entrare in CS

```
while (true) {

    flag[i]= true;
    turn = j;
    while (flag[j] && turn == j);

      /*sezione critica*/

    flag[i] = false;

      /*sezione non critica*/

}
```

Figura 6.3 Struttura del processo P_i nella soluzione di Peterson.

Riordino delle istruzioni

Variabili globali

```
boolean flag = false;
int x = 0;
```

Non ci sono dipendenze tra le variabili `flag` e `x`

Thread 1

```
while(!flag)
    ;
print x;
```

Thread 2

```
x = 100;
flag = true;
```

Cosa viene stampato da Thread 1?

Riordino delle istruzioni

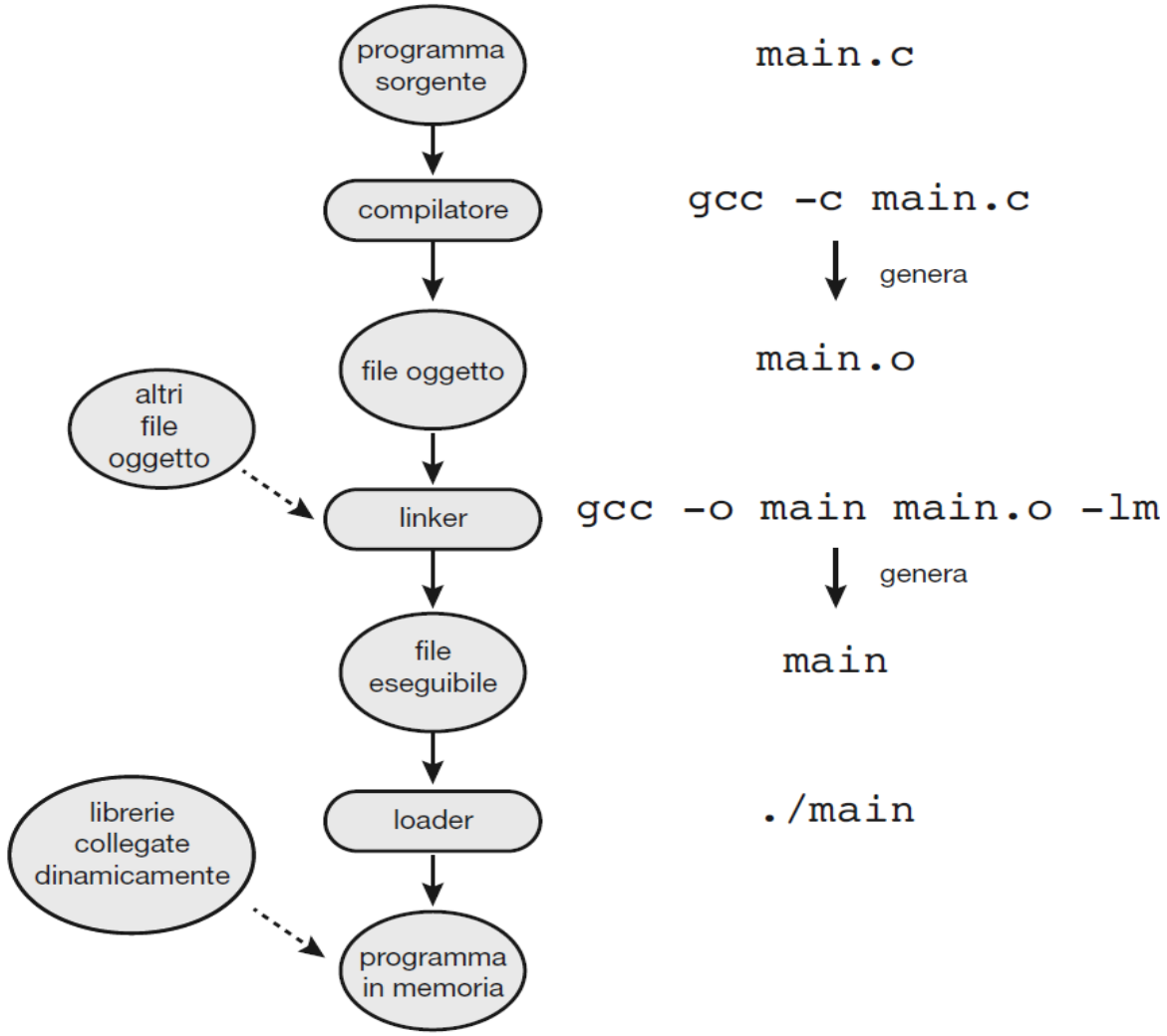


Figura 2.11 Il ruolo di linker e loader.

Soluzione di Peterson

Si consideri ciò che accadrebbe se gli assegnamenti che compaiono nella sezione d'ingresso della soluzione di Peterson della Figura 6.3 venissero riordinati.
In questo caso sarebbe possibile avere entrambi i thread attivi nelle loro sezioni critiche contemporaneamente, come mostrato nella Figura 6.4.

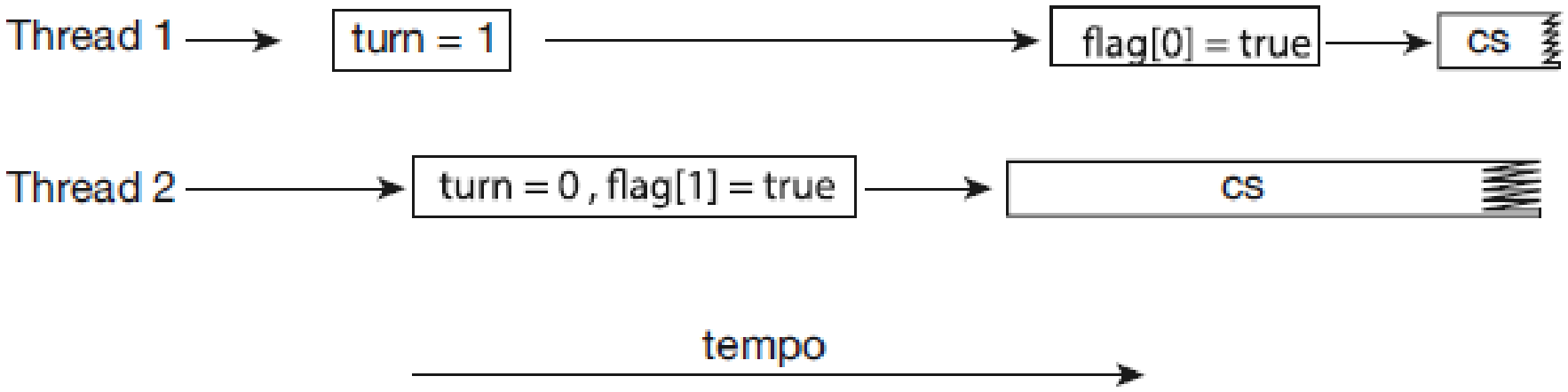


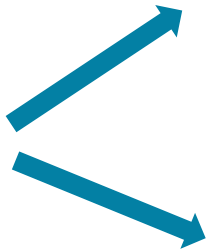
Figura 6.4 Effetti del riordino delle istruzioni nella soluzione di Peterson.

Supporto hardware per la sincronizzazione

- Soluzioni basate sul software come quella di Peterson non garantiscono il loro funzionamento su architetture elaborative moderne

Modello di memoria

un modello di memoria rientra in una delle due categorie



Fortemente ordinato una modifica alla memoria su un processore è immediatamente visibile a tutti gli altri processori

Debolmente ordinato le modifiche alla memoria su un processore potrebbero non essere immediatamente visibili agli altri processori

Barriere di memoria

barriere di memoria (*memory barrier)* o **recinzioni di memoria *(memory fence)***

➡ assicurano che le operazioni di store siano completate e visibili ad altri processori prima che vengano eseguite le operazioni di load e store future

Barriere di memoria

Variabili globali

```
boolean flag = false;
int x = 0;
```

Non ci sono dipendenze tra le variabili `flag` e `x`

Thread 1

```
while(!flag)
    memory_barrier();
print x;
```

Thread 2

```
x = 100;
memory_barrier();
flag = true;
```

Cosa viene stampato da Thread 1?

Istruzioni hardware

Molte delle moderne architetture offrono particolari istruzioni che permettono di controllare e modificare il contenuto di una parola di memoria, oppure di scambiare il contenuto di due parole di memoria, in modo **atomico** – cioè come un'unità non interrompibile.

```
boolean test_and_set(boolean *obiettivo){
    boolean valore = *obiettivo;
    *obiettivo = true;

    return valore;
}
```

Figura 6.5 Definizione dell'istruzione atomica `test_and_set()`.

Istruzioni hardware

Le istruzioni hardware sono utilizzabili per risolvere il problema della sezione critica in modo relativamente semplice

`lock` inizializzata a `false`

```
do {
  while (test_and_set(&lock));
    ; /*non fa niente*/

    /*sezione critica*/

    lock = false;

    /*sezione non critica*/
} while (true);
```

Figura 6.6 Realizzazione di mutua esclusione con `test_and_set()`.

compare_and_swap()

```
int compare_and_swap(int *value, int expected, int new_value) {
  int temp = *value;

  if (*value == expected)
    *value = new_value;

  return temp;
}
```

Figura 6.7 Definizione dell'istruzione atomica `compare_and_swap()`.

CAS viene eseguita atomicamente. Se due istruzioni CAS vengono schedulate simultaneamente (ciascuna su un core diverso), verranno eseguite sequenzialmente in ordine arbitrario.

compare_and_swap()

`lock` inizializzata a `0`

```
while (true) {
   while (compare_and_swap(&lock, 0, 1) != 0)
          ; /* non fa niente */

          /* sezione critica */

   lock = 0;

          /* sezione non critica */
}
```

Figura 6.8 Realizzazione di mutua esclusione con `compare_and_swap()`.

Questo algoritmo soddisfa il requisito della mutua esclusione, ma non quello dell'attesa limitata

`compare_and_swap()` con attesa limitata

Variabili globali

```
boolean waiting[n];
boolean lock;
```

`lock` inizializzata a `0`

tutti gli elementi di `waiting` inizializzati a `false`

```
while (true) {
    waiting[i] = true;
    key = 1;
    while (waiting[i] && key ==1)
       key = compare_and_swap(&lock, 0, 1);
    waiting[i] = false;

       /* sezione critica */

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
       j = (j + 1) % n;

    if (j == i)
       lock = 0;
    else
       waiting[j] = false;

       /* sezione non critica */
}
```

Figura 6.9 Mutua esclusione con attesa limitata con `compare_and_swap()`.

Variabile atomica

variabile atomica → fornisce operazioni atomiche su tipi di dati di base come interi e booleani.

Le variabili atomiche possono essere utilizzate per garantire la mutua esclusione in situazioni in cui potrebbe esserci una race condition su una singola variabile durante il suo aggiornamento

Lock mutex

strumenti più robusti che risolvono le race condition

lock mutex

protegge le regioni critiche e quindi prevenire le race condition

Il termine mutex deriva da **mut**ual **ex**clusion

```
while (true) {

    acquisisci lock

        sezione critica

    rilascia lock

        sezione non critica

}
```

Figura 6.10 Soluzione al problema della sezione critica con lock mutex.

Lock mutex

```
acquisisci_lock() {
   while(!disponibile)
           ; /* busy wait */
   disponibile = false;
}


rilascia_lock() {
    disponibile = true;
}
```

I lock mutex generano busy waiting, per questo sono anche detti spinlock

Busy waiting

Si verifica una situazione di attesa attiva (“busy waiting”) quando un processo che vorrebbe entrare nella propria sezione critica è costretto a ciclare continuamente richiedendo
“posso accedere alla mia sezione critica?”

Il busy waiting comporta uno speco di cicli di CPU

Vantaggio degli spinlock

Gli spinlock hanno il vantaggio di non rendere necessario alcun context switch (che richiede un considerevole aggravio in termini di costi di esecuzione) quando un processo deve attendere un lock.

Sono quindi molto utili quando si prevede che i lock saranno trattenuti per tempi brevi.

Semafori

semafori 

uno strumento più robusto in grado di comportarsi in modo simile a un lock mutex, ma capace anche di fornire metodi più complessi per la sincronizzazione delle attività dei processi

Semafori

Un **semaforo S** è una variabile intera cui si può accedere, escludendo l'inizializzazione, solo tramite due operazioni atomiche predefinite: wait() e signal().

```
wait(S) {
    while(S <= 0)
        ; /* busy wait */
    S--;
}
```

```
signal(S) {
    S++;
}
```

Uso dei semafori

Can solve various synchronization problems

- A solution to the CS problem.
  - Create a semaphore "**synch**" initialized to 1

```
wait(synch)
    CS
signal(synch);
```

- Consider P_1 and P_2 that require code segment S_1 to happen before code segment S_2
  - Create a semaphore "**synch**" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

Semafori contatore

semafori
contatore

trovano applicazione nel controllo dell'accesso a una data risorsa presente in un numero finito di esemplari.

Semafori binari

semafori binari

sono simili ai **lock mutex** e vengono utilizzati al loro posto per la mutua esclusione nei sistemi dove i lock mutex non sono disponibili

Semafori

- Benché i *semafori* costituiscano un meccanismo pratico ed efficace per la sincronizzazione dei processi, il loro uso scorretto può generare errori difficili da individuare

Monitor

Una strategia per gestire possibili errori nell'uso dei semafori consiste nell'incorporare semplici strumenti di sincronizzazione in costrutti linguistici di alto livello: i **monitor**

```
monitor monitor name
{
    /* dichiarazione di variabili condivise */

    function P1 ( . . . ) {
        . . .
    }
    function P2 ( . . . ) {
        . . .
    }
        .
        .
        .
    function Pn ( . . . ) {
        . . .
    }
    initialization_code ( . . . ) {
        . . .
    }
}
```

Figura 6.11 Sintassi di un monitor in pseudocodice.

Monitor

Il **costrutto monitor** assicura che all'interno di un monitor possa essere attivo un solo processo alla volta, in modo tale che non si debba codificare esplicitamente il vincolo di mutua esclusione (Figura 6.12)

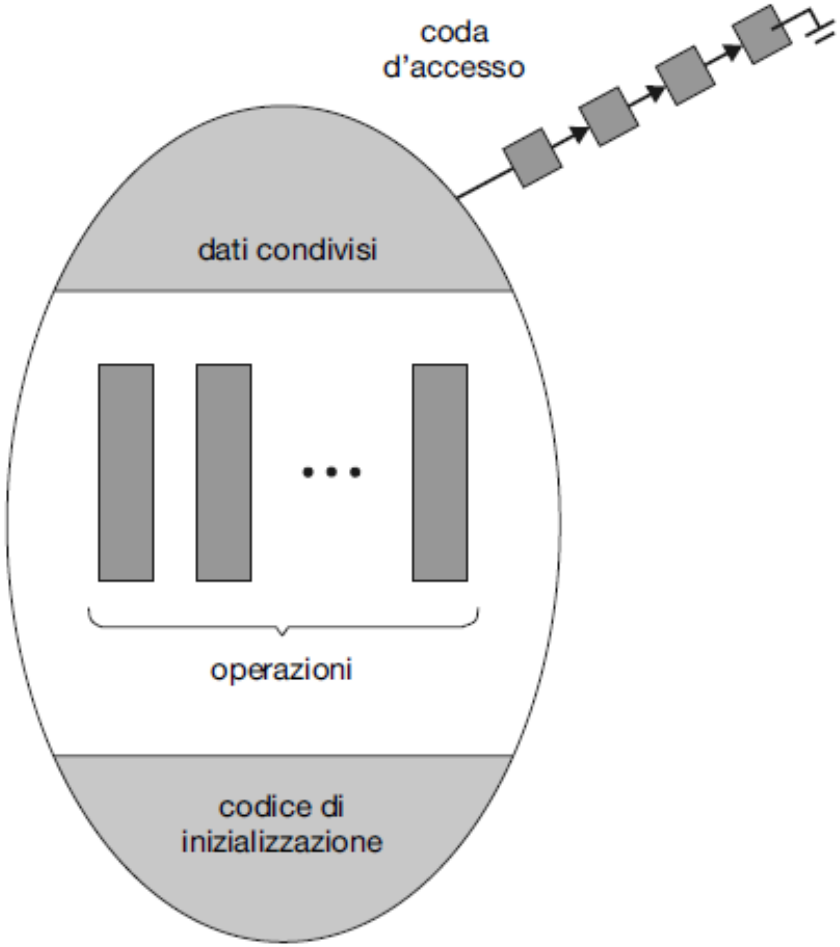


Figura 6.12 Schema di un monitor.

Monitor

Un programmatore che necessiti di implementare un proprio particolare schema di sincronizzazione può definire una o più **variabili di tipo *condition* *(condizionali):***

condition x, y;

Un monitor utilizza variabili condizionali per consentire ai processi di attendere che si verifichino determinate condizioni e di segnalarsi reciprocamente quando ciò avviene

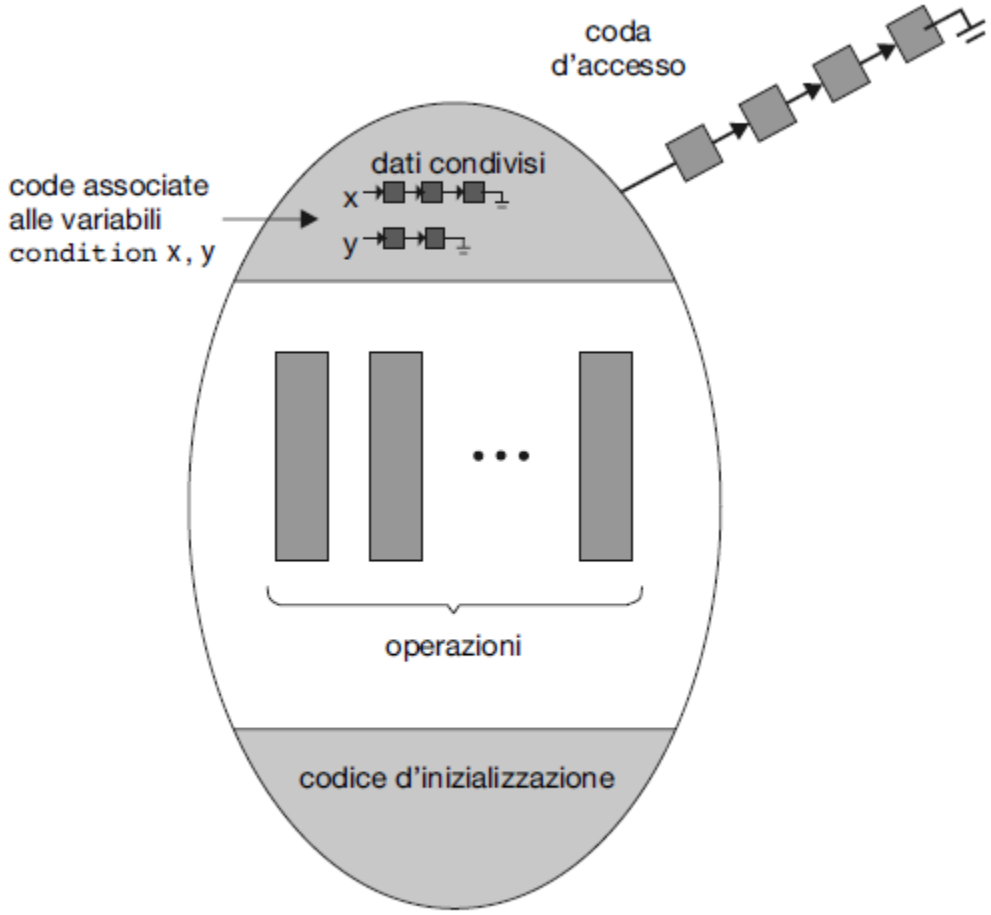


Figura 6.13 Monitor con variabili condition.

Liveness

Il termine **liveness** fa riferimento a un insieme di proprietà che un sistema deve soddisfare per garantire che i processi facciano progressi durante il ciclo di vita della loro esecuzione.

Un processo che attende *indefinitamente* è un esempio di "**mancanza di liveness**" (*liveness failure*).

Liveness

due situazioni possono portare a mancanza di *liveness*:

- stallo o deadlock
- inversione di priorità

Deadlock

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let ***S*** and ***Q*** be two semaphores initialized to 1

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

Starvation

- **Starvation – indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended

La starvation può essere causata per esempio dall'uso di una lista LIFO associata ad un semaforo

Inversione di priorità

- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**

Mars Pathfinder

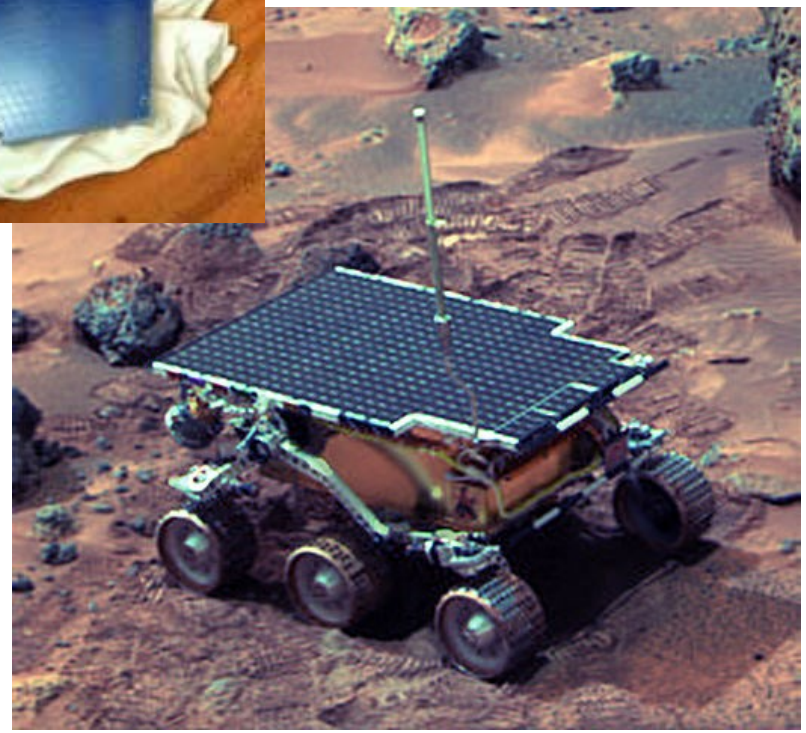
Mars Pathfinder è stata una missione scientifica della NASA per l'esplorazione di Marte

- prima missione ad aver trasportato un rover, Sojourner, sul pianeta.

- Lancio: 4 dicembre 1996
- Arrivo: 4 luglio 1997 (7 mesi dopo)

- Il lander operò come una stazione meteorologica

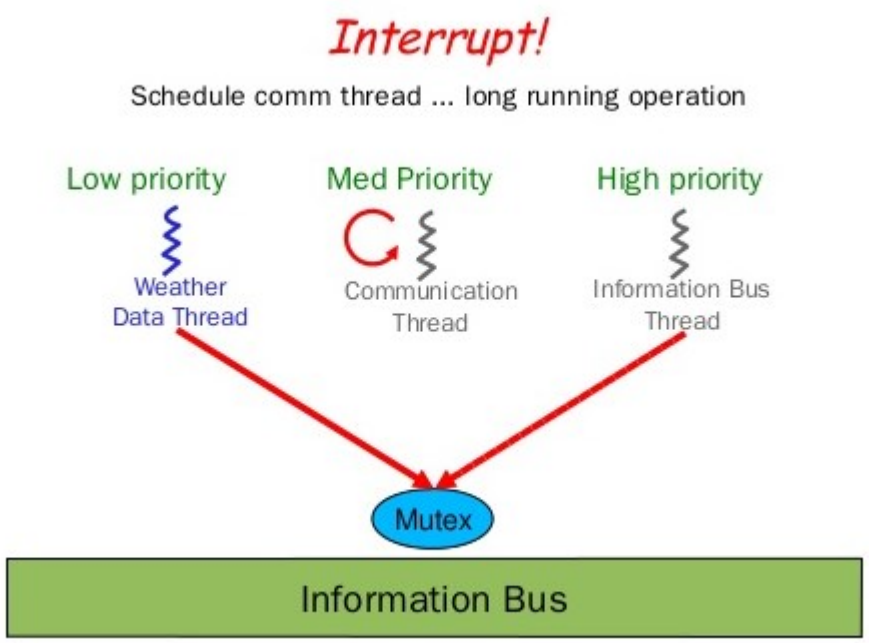
- Il rover analizzò il suolo e le rocce del sito di atterraggio ed effettuò diversi esperimenti sulla superficie.

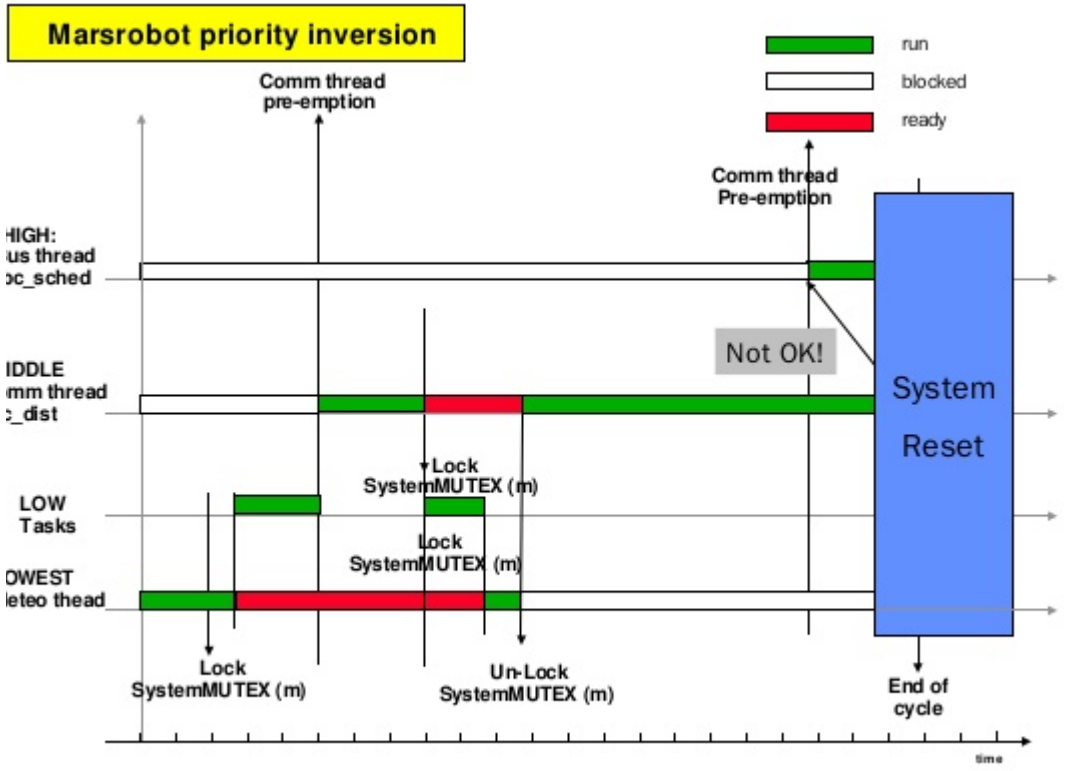


https://space.skyrocket.de/doc_sdat/mars_pathfinder.htm

https://mrbit.me/Pathfinder-Priority-Inversion/

Inversione di priorità





Livello di contesa

I vari strumenti che possono essere utilizzati per risolvere il problema della sezione critica e per sincronizzare l'attività dei processi possono essere valutati a seconda del livello di contesa.

Alcuni strumenti funzionano meglio con un certo livello di contesa rispetto ad altri.

Livello di contesa

Le seguenti linee guida identificano delle regole generali per distinguere le prestazioni della sincronizzazione basata su `compare_and_swap()` (CAS) e della sincronizzazione tradizionale (come i lock mutex e i semafori) al variare del livello di contesa:

- Nessuna contesa
- Contesa moderata
- Alta contesa

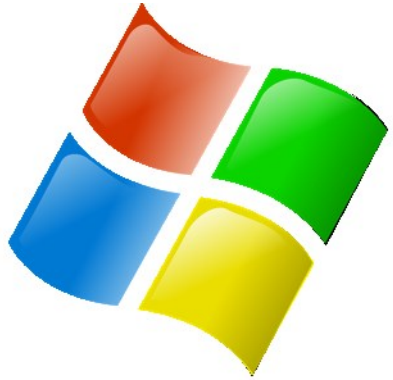
Valutazione delle soluzioni

- **Nessuna contesa.** Sebbene entrambe le opzioni siano generalmente veloci, la protezione CAS sarà leggermente più veloce della sincronizzazione tradizionale.

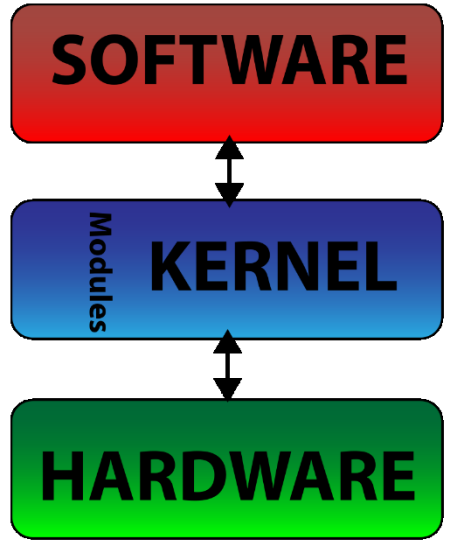
- **Contesa moderata.** La protezione CAS sarà più veloce e in alcuni casi molto più veloce rispetto alla sincronizzazione tradizionale.

- **Alta contesa.** Con carichi molto elevati, la sincronizzazione tradizionale sarà in definitiva più veloce della sincronizzazione basata su CAS.

UNIVERSITÀ DEGLI STUDI DELLA BASILICATA



Corso di Sistemi Operativi
A.A. 2019/20



Sincronizzazione dei processi

Docente:
Domenico Daniele Bloisi

Ottobre 2019