

Supporto a Mutua Esclusione e Sincronizzazione  
in Spazio Utente  
da parte di  
livello ISA, Sistema Operativo e linguaggio C

ovvero

come sono implementate le funzionalità delle  
mutexes e condition variables

# Domanda

Il sistema operativo supporta la programmazione concorrente in primo luogo mettendo a disposizione la possibilità di eseguire concorrentemente più processi, e più thread all'interno di ciascun processo.

Inoltre, il sistema operativo fornisce delle funzioni di libreria, che possono essere usate da chi intende scrivere i programmi utente, per organizzare delle sincronizzazioni tra i thread di uno stesso processo (e, come vedremo più avanti, anche tra i processi).

Tali funzioni di libreria sono quelle che permettono di creare ed usare le astrazioni delle mutexes e delle condition variables.

**ora,**

**dopo averle usate per un mese,  
finalmente emerge un dubbio**

...

*ma come beeeep sono implementate  
le maledette mutex, cond, .... ?*

E hanno bisogno di qualche hardware o software particolare per essere implementate?

# Tentativo 1 di Garantire la Mutua Esclusione

100 thread devono lavorare di continuo in mutua esclusione su una sezione critica. Per fare questo usano una variabile intera chiamata *occupata* che vale 1 quando qualcuno occupa la sezione critica e vale 0 se la sezione critica è libera. **All'inizio la variabile occupata vale 0.**

I due thread controllano continuamente il valore della variabile fino a che scoprono che vale 0, a quel punto la settano a 1 e eseguono la sezione critica.

THREAD1

```
while (1) {  
    while ( occupata==1) ;  
    occupata=1;  
    sezione critica  
    occupata=0;  
    sezione non critica (fa altro)  
}
```

THREAD2

```
while (1) {  
    while ( occupata==1) ;  
    occupata=1;  
    sezione critica  
    occupata=0;  
    sezione non critica (fa altro)  
}
```

In questo codice ci sono due problemi:

- 1) **Busy waiting:** finchè la sezione critica è occupata da un thread, gli altri thread eseguono in continuazione degli if. Spreco di CPU
- 2) **Non mutua esclusione:** supponiamo che il thread 1 nel while interno scopra che la variabile occupata vale 0 ma, prima che possa assegnare il valore 1 alla variabile, lo scheduler lo interrompe e fa partire un altro thread 2. Il thread 2 vede la variabile occupata==0, la mette ad 1 e comincia ad eseguire la sezione critica. Mentre il thread 2 sta eseguendo la sezione critica, lo scheduler lo interrompe e fa ripartire il thread 1. Il thread 1 setta ad 1 la variabile occupata e entra pure lui nella sezione critica. Ora i due thread sono entrambi in sezione critica. Violazione.

# Tentativo 2 di Garantire la Mutua Esclusione

Il tentativo 1 non garantisce la mutua esclusione perché:

Io scheduler può interrompere l'esecuzione del thread subito dopo il controllo del valore della variabile *occupata* nella condizione del while interno e quindi prima del successivo assegnamento di 1 ad *occupata*.

```
while ( occupata==1) ;  
occupata=1;
```

interrompibile perché la coppia delle due istruzioni di confronto e di assegnamento NON VIENE ESEGUITA ATOMICAMENTE

Riscrivo il codice per il Tentativo 2 di garantire la Mutua Esclusione

```
while (1) {  
  
    while (1) {  
        int modificata;  
        if ( modificata==(occupata==0) ) {  
            occupata=1;  
            if ( modificata )  
                break;  
        }  
        /* esco con occupata==1 */  
        sezione critica  
        occupata=0;  
        sezione non critica (fa altro)  
    }  
}
```

Occorre una funzione C  
in grado di eseguire  
ATOMICAMENTE  
queste due operazioni di  
di confronto della var *occupata*  
e di assegnamento della stessa.

ed esiste !!!!!

int \_\_sync\_bool\_compare\_and\_swap (  
 int \*ptrVar,  
 int vecchioValore,  
 int nuovoValore );

# Supporto C (o meglio gcc) alla Mutua Esclusione (1)

## Atomic Built-in functions

Il compilatore gcc fornisce alcune estensioni C denominate Atomic Built-in :

sono funzioni C che permettono di eseguire in forma atomica alcune operazioni che sono molto utili per implementare meccanismi di mutua esclusione.

Operano su tipi di dato intero a 1, 2, 4, 8 bytes di lunghezza.

Sfruttano delle istruzioni particolari dei processori, diverse da processore a processore, insomma sfruttano il livello ISA.

Le più note tra queste istruzioni sono: CompareAndSwap (CAS), TestAndSet, ...

Non sempre le atomic built-in function sono disponibili per tutte le architetture.

Per i processori Intel recenti, esistono alcune atomic built-in function molto efficienti.

Vedere qui per un elenco:

<https://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/Atomic-Builtins.html#Atomic-Builtins>

Ce ne interessa una in particolare: \_\_sync\_bool\_compare\_and\_swap

# Supporto C (o meglio gcc) alla Mutua Esclusione (2)

## Atomic Built-in functions

Ci interessa una particolare atomic built-in function, messa a disposizione direttamente dal compilatore gcc, nelle versioni dal 2007 in poi.

Non sempre le atomic built-in function sono disponibili per tutte le architetture.

```
bool __sync_bool_compare_and_swap ( type *ptrVar,  
                                    type vecchioValore, type nuovoValore );
```

dove type può essere un intero con o senza segno e di diverse dimensioni, in particolare: int8\_t uint8\_t int16\_t uint16\_t int32\_t uint32\_t int64\_t uint64\_t

La funzione effettua atomicamente una comparazione ed un assegnamento così :

```
if ( *ptrVar==vecchioValore ) { *ptrVar=nuovoValore; return ( 1 ); }  
else return ( 0 );
```

Se il valore di **\*ptrVar** è **uguale a vecchioValore**,  
allora il **nuovoValore** è scritto nella locazione di memoria **\*ptrVar**,  
altrimenti  
viene mantenuto il valore esistente nella locazione di memoria.

La funzione restituisce true se il nuovoValore è stato scritto. restituisce false altrimenti.

# Supporto compilatori C Microsoft alla Mutua Esclusione Interlocked API in compilatori C Microsoft

Analogamente al compilatore gcc, anche i compilatori Microsoft forniscono delle funzioni per l'accessi atomico ai dati interi.

Le funzioni con nome InterlockedQUALCOSA sono l'equivalente delle atomic builtins del gcc.

Esistono:

- InterlockedExchange
- InterlockedIncrement
- InterlockedDecrement
- InterlockedCompareExchange
- InterlockedExchangeAdd.

Nei compilatori Microsoft, la funzione equivalente all' atomic built-in function \_\_sync\_bool\_compare\_and\_swap del gcc è la InterlockedCompareExchange.

# Supporto ISA alla Mutua Esclusione (1)

## istruzione compare-exchange con blocco del bus

L'implementazione di "*Atomic Built-in functions*" e "*Interlocked API*" sfrutta delle istruzioni particolarissime messe a disposizione dal livello ISA appositamente per questi compiti. Tali istruzioni sono diverse da processore a processore

Nei processori **Intel** delle famiglie **x86** (**IA-32**) e **x64** (**EM64T,Intel 64**) esiste un'istruzione **cmpxchg** ed un prefisso **lock** che modifica il comportamento di questa istruzione, e di poche altre.

L'istruzione **cmpxchg** vuole due argomenti esplicativi ed uno implicito. Il primo argomento esplicito è l'indirizzo di una variabile intera in memoria, il secondo argomento esplicito è il nome di un registro in cui c'è già il valore da assegnare alla variabile. L'argomento隐式 è il registro eax in cui deve essere posto il valore da confrontare col valore della variabile.

L'istruzione **cmpxchg** preleva il valore della variabile puntata dall'indirizzo, lo confronta col valore contenuto nel registro eax, se i due valori sono uguali assegna alla variabile il nuovo valore (contenuto nel registro passato come secondo argomento). In pratica, se la variabile ha il valore cercato allora le viene assegnato il nuovo valore, altrimenti mantiene il suo vecchio valore.

Il prefisso **lock** opera solo su alcune istruzioni che interagiscono con la memoria. Il prefisso **lock** impone che per tutta la durata dell'istruzione a cui si riferisce, il bus non possa essere usato da altri. E' una specie di accesso in mutua esclusione al bus. Spesso non è un blocco del bus, ma un blocco sui contenuti della cache.

# Supporto ISA alla Mutua Esclusione (2)

## cosa fa l'istruzione compare-exchange con blocco del bus

cosa fa l'istruzione per Intel x86 e x64 :

**lock cpxchg DWORD PTR [rbp-16], edx**

- **DWORD PTR [rbp-16]** specifica un indirizzo di una variabile nello stack
  - mediante un offset (-16) da sommare al contenuto del registro Base Pointer a 64bit (rbp)
  - e, implicitamente, mediante il registro SS (registro selettore di segmento)
  - e dice che quell'indirizzo punta ad una variabile intera a 32 bit.
- edx è il registro a 64 bit che contiene il nuovo valore da assegnare
- eax è implicito e contiene il valore che vogliamo sia già nella variabile per poterle assegnare il nuovo valore.

In pratica, se la variabile in memoria ha il valore contenuto in eax, allora le viene assegnato il nuovo valore contenuto in edx, altrimenti mantiene il suo vecchio valore.

**Durante l'esecuzione dell'istruzione, nessun altro può usare il bus tra CPU e memoria, perciò nessuno può accedere alla locazione in memoria a quell'indirizzo, quindi il confronto e l'assegnamento avvengono atomicamente.**

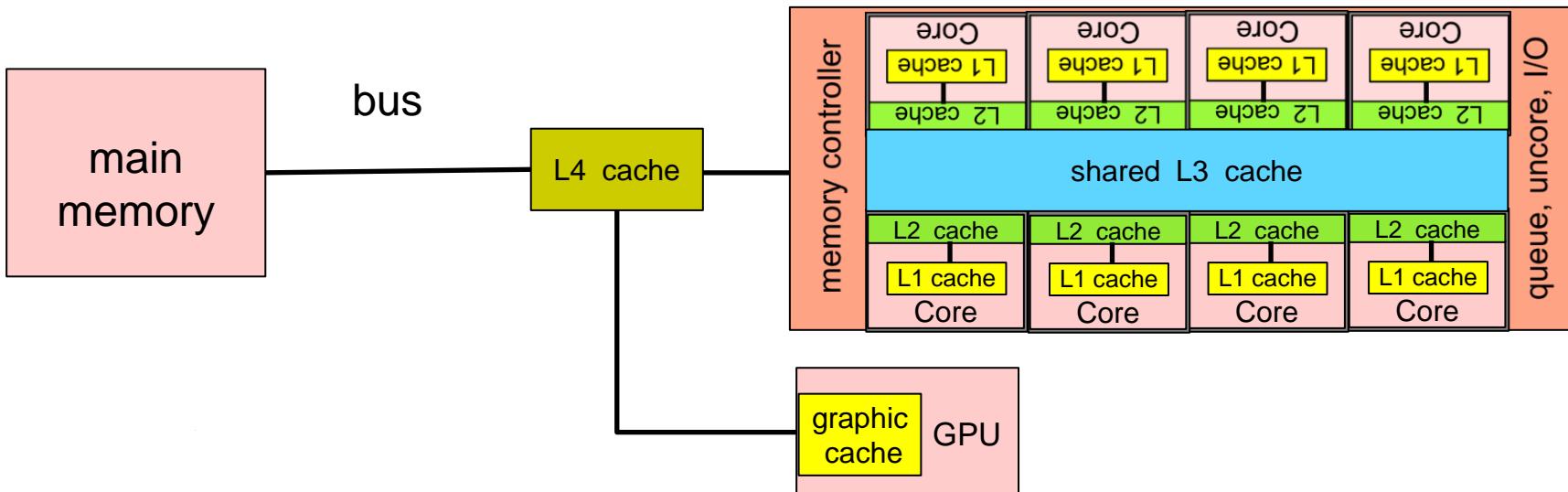
Il bit ZF del registro dei flags EFLAGS, contiene il risultato del confronto, ZF==0 se nella variabile c'era il valore cercato e adesso c'è quello nuovo, ZF!=0 se nella variabile non c'era il valore cercato e c'è rimasto quello.

# Supporto ISA alla Mutua Esclusione (3)

## Viene veramente bloccato il bus ? No!

Viene veramente bloccato il Bus? No

Solitamente si **programma la cache** tra la CPU e la memoria **per bloccare ulteriori accessi all'indirizzo** che si deve proteggere.



per dettagli si veda il documento di Intel

<https://software.intel.com/en-us/articles/implementing-scalable-atomic-locks-for-multi-core-intel-em64t-and-ia32-architectures>

"In the days of Intel 486 processors, the lock prefix used to assert a lock on the bus along with a large hit in performance.

Starting with the Intel Pentium Pro architecture, **the bus lock is transformed into a cache lock**.

A lock will still be asserted on the bus in the most modern architectures if the lock resides in uncacheable memory or if the lock extends beyond a cache line boundary splitting cache lines. Both of these scenarios are unlikely, so most lock prefixes will be transformed into a cache lock which is much less expensive."

# Supporto ISA alla Mutua Esclusione (4)

## Non è pericoloso "bloccare il bus?"

Dubbio amletico:

l'istruzione lock cmpxchg può essere eseguita da chiunque, senza essere root, cioè viene eseguita in user mode, **senza andare in kernel mode**.

Non è un potenziale pericolo?

Non è un pericolo perché quello che viene specificato come indirizzo in memoria DEVE COMUNQUE ESSERE un indirizzo di un'area di memoria che appartiene al processo che esegue l'istruzione.

Se tento di accedere ad un'area di memoria diversa il kernel provoca segmentation fault e mi ammazza il processo.

Il sistema operativo mantiene le tabelle con i permessi di accesso ai segmenti di memoria, e quindi coopera al corretto funzionamento delle istruzioni ISA che utilizzano queste tabelle.

# Supporto ISA alla Mutua Esclusione (5)

siamo diffidenti: verifichiamo come è implementata la atomic built-in function \_\_sync\_bool\_compare\_and\_swap

```
/* builtin__sync_bool_compare_and_swap.c
   vedere assembly con gcc -S -masm=intel -ansi builtin__sync_bool_compare_and_swap.c
*/
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main()
{
    int32_t occupata=11; /* mettere 13 e vedere che succede */
    int32_t modificata;

modificata = __sync_bool_compare_and_swap( &occupata, 11, 67 );

    printf ( "modificata %i    occupata %i \n", modificata, occupata );
    return(0);
}
```

# Supporto ISA alla Mutua Esclusione (6)

verifichiamo come è implementata la atomic built-in function  
`__sync_bool_compare_and_swap`

; generando l'assembly del codice C precedente ottengo questo (ho tagliato dei pezzi):

```
; inizializzo la locazione di memoria    int 32_t occupata=11
    mov    DWORD PTR [rbp-16], 11
    mov    eax, 11                                ; val da confrontare
    mov    edx, 67                                ; val da assegnare
```

; confronto e, in caso trovo il valore cercato, assegno nuovo valore  
; bloccando, nel frattempo, l'accesso a quella locazione di memoria  
; se trova valore cercato setta ZF=1

**lock cmpxchg**    DWORD PTR [rbp-16], edx

; salvo nel registro al il risultato del confronto precedente  
; se trovato valore cercato (se ZF==1), allora assegno al=1 altrimenti al=0  
sete al

```
; assegno alla variabile int32_t modificata il risultato del confronto  
    movzx  eax, al  
    mov    DWORD PTR [rbp-12], eax
```

# Morale: Supporto a Mutua Esclusione in Spazio Utente

Come ci aspettavamo, la funzione Atomic Built-in del gcc

`_sync_bool_compare_and_swap`

è implementata in x86 x86-64 e x64 mediante l'istruzione ISA

`lock cmpxchg`

Morale:

- Il livello di linguaggio C, supporta la mutua esclusione nei programmi utente implementando delle funzioni (ad esempio `_sync_bool_compare_and_swap`) che permettono l'accesso ed il confronto su variabili intere in memoria, in modo atomico.
  - per fare ciò, Il livello di linguaggio sfrutta il supporto del livello ISA.
- Il livello ISA supporta la mutua esclusione nei programmi utente fornendo delle istruzioni che permettono l'accesso ed il confronto su variabili intere in memoria, in modo atomico.
  - per fare ciò, le istruzioni ISA utilizzano tavole mantenute dal sistema operativo per verificare permessi di accesso ai diversi segmenti.
- Il sistema operativo supporta la mutua esclusione nei programmi utente mantenendo le tavole sullo stato dei processi e dei thread (scheduler) e mantenendo le tavole sui segmenti in memoria (e, vedremo, sulle pagine) con i permessi di accesso.

# Tentativo 3 di Garantire la Mutua Esclusione

Il tentativo 3 è la modifica del tentativo 2 ma con la `__sync_bool_compare_and_swap`.

Riscrivo il codice per il **Tentativo 3 di garantire la Mutua Esclusione**

```
int occupata=0;
```

```
while (1) {                                E' ATOMICA
    while(1) {
        int modificata;
        modificata = __sync_bool_compare_and_swap( &occupata, 0 , 1 );
        if ( modificata ) break;
    }
    /* esco con occupata==1 */
    sezione critica
    occupata=0; /*meglio __sync_bool_compare_and_swap(&occupata, 1, 0 ); ? */
    sezione non critica (fa altro)
}
```

Diagram illustrating the execution flow of the code:

- An arrow labeled "E' ATOMICA" points to the `__sync_bool_compare_and_swap` call.
- An arrow labeled "vecchio valore" points to the current value of `occupata` (0).
- An arrow labeled "nuovo valore" points to the new value being assigned (1).

Nota Bene: l'assegnamento `occupata=0` lo fa solo chi prima ha assegnato 1 ad `occupata`.

**Risolta la Non Mutua Esclusione !**

**Rimane il problema del Busy Waiting: mentre aspetto di prendere la mutua esclusione, itero in continuazione eseguendo vagonate di test sulla variabile `occupata`.**

# Tentativo 4 di Garantire la Mutua Esclusione (1)

Il tentativo 4 parte dal tentativo 3 **ma cerca di evitare Busy Waiting**. Riscrivo il codice di 3. La variabile occupata la tratto come una sorta di mutex

```
while (1) {  
    while (1) {  
        int modificata;  
        modificata = __sync_bool_compare_and_swap( &occupata, 0 , 1 );  
        if ( modificata ) break;  
        else futex_wait ( &occupata, 1 ); /* vedi nota (1) */  
            /* la funzione controlla se var occupata vale 1 e in tal caso  
             aspetta che qualcuno la risvegli, altrimenti prosegue e fa while */  
    }  
    /* quando arrivo qui sicuramente occupata==1 */  
    sezione critica  
    {  
        occupata=0;  
        /* sveglio uno solo 1 di quelli che aspettano su futex_wait(&occupata,...) */  
        futex_signal ( &occupata, 1 );  
    }  
    sezione non critica (fa altro)  
}
```

E' ATOMICA

vecchio valore      nuovo valore

*mutex\_lock*

*mutex\_unlock*

Bello, ma come implemento le due funzioni **viola**, **futex\_wait** e **futex\_signal** ?

# Tentativo 4 di Garantire la Mutua Esclusione (1a)

Nota (1) verifichiamo il valore di occupata appena fuori dal while interno

consideriamo il pezzo di codice di inizio del while interno:

```
modificata = __sync_bool_compare_and_swap( &occupata, 0, 1 );  
  
/* supponiamo che la __sync_bool abbia messo occupata ad 1,  
quindi ora modificata vale 1 ed occupata vale 1,  
per come è costruito l'algoritmo, QUI in mezzo  
nessun altro thread può settare a 0 la variabile occupata  
*/  
if ( modificata ) break;  
else  
    futex_wait( &occupata, 1 ); /* vedi nota (1) */
```

Se la \_\_sync\_bool\_compare\_and\_swap ha settato ad 1 la variabile occupata,  
sicuramente esco dal ciclo while interno  
e sicuramente appena uscito dal while interno la variabile occupata ha valore 1

# Tentativo 4 di Garantire la Mutua Esclusione (1b)

**Nota (1) ) verifichiamo il valore di occupata appena fuori dal while interno**  
consideriamo ancora il pezzo di codice di inizio del while interno:

```
modificata = __sync_bool_compare_and_swap( &occupata, 0, 1 );
```

**/\* supponiamo che la \_\_sync\_bool NON ABBIA MESSO occupata ad 1 perchè occupata valeva già 1,**

quindi ora *modificata* vale 0 e *occupata* vale 1,

**QUI in mezzo, potrebbe capitare che**

**l'altro thread che aveva messo occupata ad 1  
setti a 0 la variabile occupata.**

**In tal caso la futex\_wait non si bloccherà e  
si rifarà il while interno cercando di assegnare 1 ad occupata**

**\*/**

```
if ( modificata ) break;
```

```
else
```

```
futex_wait ( &occupata, 1 );
```

Se qualcuno ha settato a 0 la var *occupata* dopo la *\_sync\_bool\_compare\_and\_swap*,  
allora la chiamata alla **futex\_wait ( &occupata, 1 );** **NON SI BLOCCA !**  
restituisce -1 e setta errno a EAGAIN,  
quindi si ripete il while interno,  
cercando di settare nuovamente la var *occupata* ad 1 e di uscire dal while interno.  
In pratica, si esce dal while interno solo quando *occupata* vale 1

# La system call futex (Fast User Mutex) (1)

- La Futex è una system call specifica di Linux, non è conforme a nessuno standard POSIX.
- Può essere chiamata da programmi utente e svolge operazioni diverse a seconda dei parametri che le vengono passati. **In generale effettua sincronizzazioni.**
- Essendo una system call, i programmi utente dovranno chiamare la futex indirettamente, cioè mediante una funzione apposita detta **syscall()**.
- L'insieme dei thread che vogliono sincronizzarsi tra loro mediante la **futex**, usano, senza saperlo, **una stessa specifica coda di thread in attesa**.
  - In un sistema potrebbero esserci più code di attesa.
  - Per identificare una specifica coda di thread in attesa si usa un identificatore univoco.
  - L' **identificatore** di una coda di thread in attesa **contiene l'indirizzo di una variabile intera**, detta **futex word, condivisa dai thread** che vogliono sincronizzarsi tra loro, e scelta da questi thread.
  - La variabile condivisa scelta dai thread rappresenta le operazioni di sincronizzazioni tra i thread.
  - Ogni volta che un thread vuole effettuare una operazione di sincronizzazione dovrà chiamare la **Futex** e passarle come **primo argomento l'indirizzo della variabile intera condivisa**.

# La system call futex (Fast User Mutex) (2)

- La Futex deve essere chiamata mediante la funzione long syscall( long number, ....);
- Nel modulo in cui si vuole chiamare la futex occorre definire dei simboli e includere alcuni header.

```
#define _GNU_SOURCE
#include <unistd.h>
#include <sys/syscall.h>} /* per syscall() */
#include <sys/time.h>      /* per struct timespec */
#include <linux/futex.h>     /* per futex() */
```

- Il prototipo della system call è il seguente:

```
int futex ( int *uaddr, int futex_op, int val,
            const struct timespec *timeout /* or: uint32_t val2 */,
            int *uaddr2, int val3 );
```

- Il primo argomento è il puntatore alla variabile intera in spazio utente (la futex word) usata dai thread per rappresentare le operazioni di sincronizzazione tra di loro.
- Il valore passato come secondo argomento futex\_op determina l'operazione da eseguire, e determina il significato degli argomenti 3, 4, 5 e 6.
- Nei casi che vedremo, i 3 ultimi argomenti avranno sempre valori NULL, NULL, 0.

La Futex deve essere chiamata mediante la funzione long syscall ( long number, ... ).

Tale funzione restituisce -1 in caso di errori e scrive nella variabile errno il codice di errore.

In caso di successo la syscall() restituisce un risultato che dipende dall'operazione che era stata richiesta alla futex.

# Attesa mediante la system call futex

- Se l'operazione indicata dal secondo argomento è **FUTEX\_WAIT** oppure è **FUTEX\_WAIT\_PRIVATE** allora il terzo argomento val è un valore intero di confronto.
- in tali casi la futex controlla se il valore contenuto nella futex word (puntata dall'indirizzo al primo argomento) è uguale al valore passato come terzo argomento.

Se i due valori sono uguali la futex blocca il thread e non termina fino a che un altro thread effettuerà una specie di signal mediante una altra futex con operazione **FUTEX\_WAKE**. A questo punto la futex termina e restituisce valore 0.

Se invece i due valori non sono uguali, allora la futex termina subito senza bloccare il thread e restituisce valore -1 e mette il codice d'errore EAGAIN o EWOULDBLOCK.

**Il caricamento del valore della futex word dalla memoria, il confronto con il valore specificato dal terzo argomento e l'inizio della messa in attesa del thread sono operazioni che vengono eseguite atomicamente rispetto alle altre operazioni futex relative alla stessa variabile condivisa futex word.**

Se il thread che chiama la futex viene messo in attesa, il thread viene inserito nella coda di attesa dei thread realtivi a quella futex word.

**esempio:** `syscall( SYS_futex, ptr, FUTEX_WAIT_PRIVATE, 97, NULL, NULL, 0);`  
si blocca se il valore della futex word (puntato da ptr) vale 97.

NB: L'operazione **FUTEX\_WAIT\_PRIVATE** opera solo su thread. L'operazione **FUTEX\_WAIT** invece opera anche su processi a patto che la variabile condivisa futex word si trovi in un segmento di memoria condivisa tra quei processi.

# Risveglio mediante la system call futex

- Se l'operazione indicata dal secondo argomento è **FUTEX\_WAKE** oppure è **FUTEX\_WAKE\_PRIVATE** allora il terzo argomento val è il numero di thread da risvegliare.
- in tali casi la futex risveglia fino ad un massimo di val thread che erano stati bloccati da una precedente FUTEX\_WAIT fatte sulla stessa futex word di cui è stato passato l'indirizzo come primo argomento.
- Non è specificato quale dei thread in attesa viene risvegliato.
- Se val vale 1 riveglia un solo thread.
- Se val vale **INT\_MAX** risveglia tutti i thread in attesa.
- I thread risvegliati vengono messi nella coda dei processi ready, sarà poi lo scheduler a decidere quando far loro continuare l'esecuzione.
- Al termine la futex restituisce il numero di thread risvegliati, oppure -1 in caso di errore, nel qual caso mette il codice d'errore in errno.

**esempio:** `syscall( SYS_futex, ptr, FUTEX_WAKE_PRIVATE, 1, NULL, NULL, 0);`  
risveglia 1 dei thread bloccati in attesa sulla futex word puntata da ptr.

- NB: L'operazione FUTEX\_WAKE\_PRIVATE opera solo su thread. L'operazione FUTEX\_WAIT invece opera anche su processi a patto che la variabile condivisa futex word si trovi in un segmento di memoria condivisa tra quei processi.

# Cenni su Implementazione system call futex (1)

La Futex realizza un meccanismo per sincronizzare thread o processi. La Futex è usabile dallo spazio utente cioè da programmi di alto livello, ed è implementata nel kernel. La Futex consente ad un thread (o task) di verificare una condizione in spazio utente, e in base a questa condizione mettersi in attesa di un segnale che proviene da altri thread (o task) oppure proseguire la propria esecuzione.

La Futex, oltre a usare una variabile passata per indirizzo, ha altri 2 componenti interni:

1. una variabile intera di stato in spazio utente (non quella passata per indirizzo alla futex) che indica se qualche thread detiene la mutua esclusione e se ci sono thread in attesa.
  2. una coda di thread (task) in attesa (waitqueue ) realizzata nello spazio kernel.
- La verifica del valore della variabile passata per indirizzo e la modifica della variabile interna in spazio utente sono svolti in maniera atomica, tipicamente mediante una funzione come `__sync_bool_compare_and_swap()` o `test_and_set()`.
  - Una waitqueue è una lista contenente gli identificatori dei thread (o task) in attesa di un certo evento.

Si parla di task in attesa esclusiva quando solo un task alla volta deve essere risvegliato. Si parla di attesa non esclusiva se tutti i task devono essere risvegliati. Una waitqueue può contenere alcuni task in attesa esclusiva e altri in attesa non esclusiva. La waitqueue viene gestita per mezzo di funzioni del kernel per mettere correttamente in attesa i task. Si usano le funzioni `wait_event_interruptible_exclusive()` e `wait_event_interruptible()` per mettere un task in attesa rispettivamente esclusiva e non esclusiva. Le funzioni inseriscono i task nelle code di attesa e li etichettano indicando se sono in attesa esclusiva o no.

# Cenni su Implementazione system call futex (2)

- L'indirizzo della variabile passato alla Futex, viene utilizzata come parte di un identificatore che serve ad individuare univocamente la waitqueue creata per mantenere i task in attesa su quella Futex e la variabile di stato interna.
- La runqueue è la lista dei thread (o task) in attesa della CPU, quella usata dallo scheduler per sapere quali sono i task pronti ad eseguire.
- Ogni volta che viene invocata la system call futex per eseguire l'operazione FUTEX\_WAKE su una determinata variabile intera della Futex, il kernel cerca la waitqueue associata a quella Futex, scandisce la lista dei thread e seleziona tanti thread da mettere nella runqueue, quanti ne ha chiesto la futex. Quei thread sono messi nella runqueue.
  - Il kernel fornisce la funzione `wake_up(wait_queue_head_t * wq, ... )`, per:
    - ▶ risvegliare tutti i task in attesa non esclusiva in quella coda puntata da wq,
    - ▶ oppure risvegliare un solo task in attesa esclusiva sulla coda puntata da wq.
  - Per ciascun task risvegliato, la funzione svolge 2 operazioni:
    - 1) ne cambia lo stato da attesa a pronto,
    - 2) lo elimina dalla waitqueue e lo pone nella runqueue.
  - Nota Bene: Risvegliando dei task, wakeup non invoca mai però direttamente lo scheduler, si limita a mettere i task nella runqueue.

# Logica Implementativa della futex (Fast User Mutex)

Per chi fosse interessato a ulteriori dettagli, VEDERE:

**A futex overview and update**

<https://lwn.net/Articles/360699/>

**Using futexes to build locking primitives.**

**"Futex are Tricky"**

<http://people.redhat.com/drepper/futex.pdf>

Meglio ancora, vedere

l'implementazione nel codice del kernel Linux, a partire dalla versione 2.6.0

# Eseguire una system call da programma utente

Per alcune system call, la libreria standard del C mette a disposizione delle funzioni (wrapper) che permettono di comandare l'esecuzione di una system call in modo semplice.

E' il caso della funzione di libreria open che apre un file chiamando la system call open nascondendo i dettagli. Però non c'è un wrapper per tutte le system call.

Per alcune system call occorre invocarle esplicitamente occupandosi dei dettagli dipendenti dall'architettura e dal sistema operativo (cioè dipendenti dall'ABI). I dettagli dipendono dal processore e possono essere ad esempio il tipo di allineamento degli argomenti da passare.

**Per ciascuna system call, a livello assembly è associato un numero che la identifica univocamente, e a questo numero è associato un simbolo. Elenco in sys/syscall.h**

Il compilatore gcc (se compiliamo con il simbolo \_GNU\_SOURCE) fornisce una funzione per chiamare esplicitamente le system call specificandone il numero.

```
#define _GNU_SOURCE  
#include <unistd.h>  
#include <sys/syscall.h> /* per SYS_xxx definitions */  
  
long syscall ( long number , ... );
```

La funzione esegue la system call il cui numero è passato come primo argomento.

**La syscall salva i registri della CPU prima di eseguire la system call e li ripristina dopo la chiamata, salvando il codice d'errore in errno se accade un errore nella system call.**

Il risultato restituito dalla funzione syscall dipende dalla system call chiamata:

in generale la funzione syscall restituisce 0 se tutto ok, -1 in caso di errore nel qual caso il codice d'errore è salvato in errno.

# Implementazione di futex\_wait() e futex\_signal()

```
#define _GNU_SOURCE
#include <stdlib.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <linux/futex.h>
```

NB: se le usate solo per sincronizzare dei thread e non dei processi, usare FUTEX\_WAIT\_PRIVATE e FUTEX\_WAKE\_PRIVATE che ottimizzano

Se \*ptr==val si blocca e aspetta che qualcuno chiami la futex\_signal poi restituisce 0.  
Se invece \*ptr!=val termina subito restituendo -1 e mettendo EAGAIN dentro errno.

```
int futex_wait ( int *ptr, int val )
{
    return syscall ( SYS_futex, ptr, FUTEX_WAIT, val, NULL, NULL, 0);
}
```

Abilita a proseguire (risveglia) 1 thread in attesa sulla futex word \*ptr se ce n'è almeno 1, e poi restituisce il numero di thread risvegliati (0 se non c'era nessun thread, 1 se ce ne era almeno 1).

```
int futex_signal ( int *ptr )
{
    return syscall ( SYS_futex, ptr, FUTEX_WAKE, 1, NULL, NULL, 0);
}
```

Occorre chiamare direttamente le system call, mediante la funzione syscall, perché non c'è una funzione della glibc che fa da wrapper alla syscall futex.

# Curiosità preliminari

definirò ora un tipo di dato così:

```
typedef volatile int pthread_futex_t;
```

volatile significa :

caro compilatore,  
non modificare la variabile tenendo il valore nei registri  
ma solo nella locazione di memoria  
E' una precauzione inutile, poiché,  
nelle implementazioni in cui userò quel tipo di dato  
accederò alla variabile solo con istruzioni atomiche.

# Implementiamo una pthread\_futex\_lock banalotta

```
#define errExit(msg)    do { int myerrno=errno; perror(msg); exit(myerrno); } while (0)
typedef volatile int pthread_futex_t;
int pthread_futex_init( pthread_futex_t *ptroccupata ) {*ptroccupata=0; return(0); }
int pthread_futex_destroy( pthread_futex_t *ptroccupata ) { return(0); }
int pthread_futex_lock( pthread_futex_t *ptroccupata ) {
    int ris; int modificata;
    while (1) {
        modificata = __sync_bool_compare_and_swap( ptroccupata, 0 , 1 );
        if ( modificata ) break;
        else { /* se occupata vale 1 allora aspetto che qualcuno mi svegli */
            ris=futex_wait ( ptroccupata, 1 );
            if ( ris== -1 && errno!=EAGAIN ) errExit("futex-FUTEX_WAIT failed:");
        }
    }
    return(0);
}
int pthread_futex_unlock( pthread_futex_t *ptroccupata ) {
    int ris;
    __sync_bool_compare_and_swap( ptroccupata, 1 , 0 ); /* occupata=0; */
    /* sveglio uno 1 di quelli che aspettano su occupata*/
    ris=futex_signal ( ptroccupata );
    if ( ris== -1 && errno!=EAGAIN ) errExit("futex-FUTEX_WAKE failed:");
    return(0);
}
```

# commenti su pthread\_futex\_lock banalotta

Nell'esempio di implementazione, consideriamo **2 possibili stati della variabile condivisa**, 0 ed 1, per distinguere quando la mutex è occupata e quando non lo è.

Le **implementazioni serie delle mutex** considerano **tre stati della variabile condivisa**, con valore 0, 1 e 2 per distinguere tra i casi in cui:

- La mutex non è occupata.
- La mutex è occupata e non ci sono altri pthread che stanno eseguendo la lock e attendono di occuparla.
- La mutex è occupata e ci sono altri pthread che stanno eseguendo la lock e attendono di occuparla.

Con 3 stati possiamo realizzare algoritmi più efficienti (ma per ora non ci interessa).

In particolare, **potremmo stabilire un ordine FIFO di risveglio dall'attesa**, ordine che invece nella nostra implementazione banalotta non viene definito.

Per definire un ordine di risveglio **dovremmo anche tenere una lista dei pthread in attesa** di risveglio.

# Esempio di mutua esclusione con pthread\_futex\_lock

```
#define NUM_THREADS 10
pthread_t tid [NUM_THREADS];
pthread_futex_t occupata;

int data=4;

int main (void) {
int rc , t;
int *p;

pthread_futex_init (&occupata);

/* creo i pthread */
for(t=0;t < NUM_THREADS;t++) {
    rc = pthread_create( & tid[t] , NULL, func, NULL ) ;
}

/* attendo terminazione dei pthread */
for(t=0;t < NUM_THREADS;t++) {

    rc = pthread_join( tid[t] , (void**) & p ) ;
}

printf ("data = %d \n", data);

pthread_futex_destroy (&occupata);

pthread_exit(NULL);
}
```

codice del pthread

```
void * func( void *arg ) {

    pthread_futex_lock (&occupata);

    if(data>0)
        data--;

    pthread_futex_unlock (&occupata);

    pthread_exit( NULL );
}
```

esempio completo in  
<http://www.cs.unibo.it/~ghini/didattica/sistemioperativi/PTHREAD/FUTEX/>

# Implementazione di Condition Variable

## Supporto da parte del sistema operativo linux

- Per implementare le **condition variable**, si utilizzano le stesse tecniche utilizzate per implementare le mutex.
- Solitamente si usa **una ulteriore variabile intera** su cui appoggiare delle ulteriori chiamate alla futex.
- Il valore di questa variabile viene modificato mediante chiamate a funzioni atomiche, come le `__sync_bool_compare_and_swap`.
- **Viene tenuta una lista dei pthread** in attesa sulle `pthread_cond_wait()`, e si tiene traccia di quali pthread hanno già ottenuto l'abilitazione a proseguire per causa di qualche `pthread_cond_signal` o `pthread_cond_broadcast`.
  - **Solo come curiosità e per i maniaci** : la lista dei pthread viene implementata mediante delle strutture dati, memorizzate in userspace, ma manipolate mediante delle apposite system calls, in modo che eventuali crash di alcuni pthread non blocchino l'accesso alle liste.

Tali liste vengono denominate `robust_list`.

- Alcune system call per manipolarle sono: `sys_set_robust_list`, `sys_get_robust_list`