Sincronizzazione 3 Istruzione RMW

martedì 25 novembre 2025 10:41

Vediamo ora come implementare una sezione critica in modo semplice , magari disabilitando le istruzioni , magari usando istruzioni di basso livello per manipolare le info in modo atomico. Quindi vediamo in dettaglio come usiamo le **istruzioni read modify write (RMW)**:

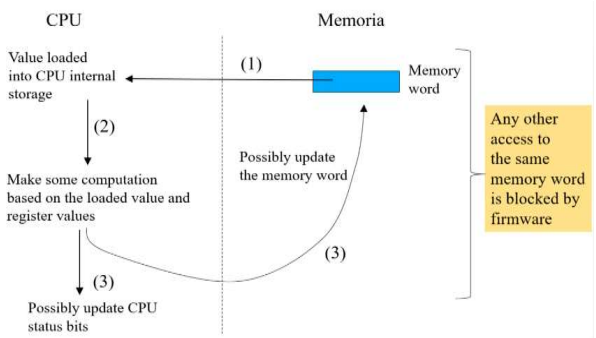
```
function test_and_set(var z: int) : boolean;
{
 if (!z) {
        z := 1;
        test_and_set := TRUE;
}
else test_and_set := FALSE;
}
```

```
var serratura: int;

Processo X

While !test_and_set(serratura) do no-op;
<sezione critica>;
serratura := 0
```

Dove z è locazione di memoria . Vediamo in dettaglio:



Le quali istruzioni atomiche consentono di gestire una sola struttura dati . Quindi in presenza di multiple uso una sorta di "serratura" . In dettaglio :

1. **Test e set**
   a. Testo la locazione di memoria e la aggiorno se opportuno
   b. Implementata da compare and swat (CAS)
   c. Su architettura x86 si usa l'istruzione **CMPXCHG**
   d. Questa istruzione compara il valore di una locazione di memoria con quello del registro RAX , il quale risultato viene immagazzinato in altro registro (Esempio RBX)
   e.

```
int try_lock(void * uadr){
     unsigned long r =0 ;
     asm volatile(
          "xor %%rax,%%rax\n"
          "mov $1,%%rbx\n"
          "lock cmpxchg %%ebx,(%1)\n"
          "sete (%0)\n"
          : : "r"(&r),"r" (uadr)
          : "%rax","%rbx"
     );
     return (r) ? 1 : 0;
}
```

Set equal
Target memory word
If they were equal return 1

   f. XOR %rax,%rax azzera il registro %rax
   g. Vediamo esempio

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>


unsigned long global_lock   __attribute__ ((aligned (64))) = 0;


int lock(void * uadr){//this takes the address of the memory area where the lock variable resides
    unsigned long r =0 ;
    asm volatile(
        "xor %%rax,%%rax\n"
        "mov $1,%%rbx\n"
        "lock cmpxchg %%ebx,(%1)\n"
        "sete (%0)\n"
        : : "r"(&r),"r" (uadr)
        : "%rax","%rbx"
    );
    return (r) ? 1 : 0;
}


#define SIZE (100000)
#define END (10000000)

#define AUDIT if(0)

long v[SIZE] = {[0 ... (SIZE-1)] -1};

long counter = 0;
```

```c
void * producer(void* dummy){

        long data = 0;
        long my_index = 0;

        printf("ready to produce\n");

retry:
        if(lock(&global_lock)){

                if(counter < SIZE){
                        v[my_index] = data;
                        my_index = (my_index+1)%SIZE;
                        data++;
                        counter++;

                }
                global_lock = 0;
        }

        goto retry;


}
```

```c
void * consumer(void* dummy){

        long data = 0;
        long my_index = 0;
        long value;

        printf("ready to consume\n");

retry:
        if(lock(&global_lock)){
                if(counter > 0){
                        value = v[my_index];

                        AUDIT
                        printf("consumer got value %ld\n",value);
                        if(value != data){
                                printf("consumer: synch protocol broken at expected value: %ld - real is %ld!!\n",data,value);
                                exit(EXIT_FAILURE);
                        };
                        if (value == END){
                                printf("ending condition met - last read value is %ld\n",value);
                                exit(0);
                        }
                        my_index = (my_index+1)%SIZE;
                        data++;
                        counter--;
                }
                global_lock = 0;
        }
        goto retry;
}
```

```c
int main(int argc, char** argv){

        pthread_t prod, cons;

        pthread_create(&cons,NULL,consumer,NULL);
//      sleep(1);
        pthread_create(&prod,NULL,producer,NULL);

        pause();

}
```

h. Bypassa il problema della sincronizzazione restituendo un valore consono
i. Vediamo una versione migliorata :

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>


//unsigned long global_lock   = 0;

int lock(void * uadr){//this takes the address of the memory area where the lock variable resides
    unsigned long r =0 ;
    asm volatile(
        "xor %%rax,%%rax\n"
        "mov $1,%%rbx\n"
        "lock cmpxchg %%ebx,(%1)\n"
        "sete (%0)\n"
        : : "r"(&r),"r" (uadr)
        : "%rax","%rbx"
    );
    return (r) ? 1 : 0;
}


#define SIZE (100000)
#define END (100000000)

#define AUDIT if(0)

long v[SIZE] = {[0 ... (SIZE-1)] -1};
long available_data[SIZE] __attribute__ ((aligned (64))) = {[0 ... (SIZE-1)] 1};
long available_slot[SIZE] __attribute__ ((aligned (64))) = {[0 ... (SIZE-1)] 0};

long counter = 0;
```

```c
void * producer(void* dummy){

        long data = 0;
        long my_index = 0;

        printf("ready to produce\n");

retry:
        if(lock(&available_slot[my_index])){

                v[my_index] = data;
                available_data[my_index] = 0; //set the data available spin to 0
                my_index = (my_index+1)%SIZE;
                data++;
        }

        goto retry;


}
```

```c
void * consumer(void* dummy){

        long data = 0;
        long my_index = 0;
        long value;

        printf("ready to consume\n");

retry:
        if(lock(&available_data[my_index])){
                value = v[my_index];
                AUDIT
                printf("consumer got value %ld\n",value);
                if(value != data){
                        printf("consumer: synch protocol broken at expected value: %ld -
                         real is %ld!!\n",data,value);
                        exit(EXIT_FAILURE);

                };
                if (value == END){
                        printf("ending condition met - last read value is %ld\n",value);
                        exit(0);
                }
                available_slot[my_index] = 0;//set the available slot spin to 0
                my_index = (my_index+1)%SIZE;
                data++;
        }
        goto retry;
}
```

```c
int main(int argc, char** argv){

        pthread_t prod, cons;

        pthread_create(&cons,NULL,consumer,NULL);
//      sleep(1);
        pthread_create(&prod,NULL,producer,NULL);

        pause();

}
```

j. Quindi la sincronizzazione avviene attraverso più elementi (differenti "serrature"), quindi è una versione più rapida: la "serratura" globale non viene rispettata : lettore e scrittore possono interagire in diversi punti usando serratura differenti