

PCAD

Programmazione Concorrente

Algoritmi Distribuiti

Arnaud Sangnier
arnaud.sangnier@unige.it

MUTEX e VARIABILE CONDIZIONALE

Esempio di dati condivisi

```
volatile int a=0;

int main(){
    pthread_t th1, th2, th3;
    pthread_create(&th1, NULL, inc, NULL);
    pthread_create(&th2, NULL, dec, NULL);
    pthread_create(&th3, NULL, dec, NULL);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    pthread_join(th3, NULL);
    printf("a vale %d\n", a);
    return 0;
}

void *inc(void *ptr){
    a=a+1;
    return NULL;
}

void *dec(void *ptr){
    if(a>0){
        a=a-1;
    }
    return NULL;
}
```

**Questo codice
potrebbe
stampare -1 !!!**

Come proteggere i dati

- L'accesso ai dati condivisi deve essere protetto
- In C, si possono usare dei lock (**mutex**)
- Ecco quale il principio di funzionamento
 - Uno thread che desidera accedere a un dato condiviso richiede il lock
 - Se il lock è libero, prosegue
 - Se il lock è preso, rimane bloccato finché il lock sia liberato
 - Quando ha finito di accedere ai dati condivisi, libera il lock
- Questi lock sono condivisi fra gli threads
- **Buona pratica:** è lo thread che ha preso il lock che lo deve liberare

I locks in C

- La libreria POSIX ha dei lock di tipo:

```
pthread_mutex_t lock;
```

- La prima cosa da fare è di inizializzare il lock
- Il più semplice consiste in fare:

```
lock=PTHREAD_MUTEX_INITIALIZER ;
```

- Per prendere il lock:

```
int pthread_mutex_lock(pthread_mutex_t *mutex) ;
```

- E per liberarlo

```
int pthread_mutex_unlock(pthread_mutex_t *mutex) ;
```

- Queste due funzione ritornano 0 se tutto è andato a buon fine

Esempio di utilizzazione di lock

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

```
void *inc(void *ptr){
    int r=pthread_mutex_lock(&lock);
    if(r!=0){
        printf("Problem with lock\n");
        return NULL;
    }
    a=a+1;
    r=pthread_mutex_unlock(&lock);
    if(r!=0){
        printf("Problem with unlock\n");
        return NULL;
    }
    return NULL;
}
```

```
void *dec(void *ptr){
    int r=pthread_mutex_lock(&lock);
    if(r!=0){
        printf("Problem with lock\n");
        return NULL;
    }
    if(a>0){
        a=a-1;
    }
    r=pthread_mutex_unlock(&lock);
    if(r!=0){
        printf("Problem with unlock\n");
        return NULL;
    }
    return NULL;
}
```

Attenzione ai deadlock!!!

```
void *foo(void *ptr);
void *goo(void *ptr);

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t lock2 = PTHREAD_MUTEX_INITIALIZER;

volatile int a=0;

int main(){
    pthread_t th1, th2;
    pthread_create(&th1, NULL, foo, NULL);
    pthread_create(&th2, NULL, goo, NULL);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    printf("a vale %d\n", a);
    return 0;
}
```

Attenzione ai deadlock!!! (2)

```
void *foo(void *ptr){
    pthread_mutex_lock(&lock);
    printf("foo prende lock1\n");
    pthread_mutex_lock(&lock2);
    printf("foo prende lock2\n");
    a=10;
    pthread_mutex_unlock(&lock2);
    printf("foo rende lock2\n");
    pthread_mutex_unlock(&lock);
    printf("foo rende lock\n");
    return NULL;
}

void *goo(void *ptr){
    pthread_mutex_lock(&lock2);
    printf("goo prende lock2\n");
    //while(a!=10){};
    pthread_mutex_lock(&lock);
    printf("goo prende lock\n");
    a=20;
    pthread_mutex_unlock(&lock);
    printf("goo rende lock\n");
    pthread_mutex_unlock(&lock2);
    printf("goo rende lock2\n");
    return NULL;
}
```

Cosa abbiamo visto

- Manipolazione di thread:
 - `pthread_create`
 - `pthread_join`
 - `pthread_exit`
- Manipolazione di lock:
 - `pthread_mutex_t lock`
 - `pthread_mutex_lock(&lock)`
 - `pthread_mutex_unlock(&lock)`
 - Per inizializzare un lock due opzioni:
 - `lock=PTHREAD_MUTEX_INITIALIZER` funziona solo alla dichiarazione
 - `pthread_mutex_init(&lock, NULL)`
 - `pthread_mutex_destroy(&lock)` per distruggere il lock

Ancora qualche punto sui locks

- I locks permettono di proteggere l'accesso a dei dati condivisi e inducono una certa forma di sincronizzazione
 - Per accedere alla sezione critica, se un altro thread è dentro, devo aspettare che esce e sono bloccato attendo
- Più thread possono essere in attesa su un lock di un mutex... cosa succede quando c'è un lock:
 - uno dei thread in attesa, prenderà il lock
 - gli altri rimangono bloccati in attesa
 - in che ordine ? non è specificato dalla norma POSIX, quindi si deve assumere in qualsiasi ordine
- Importante: **deve essere lo thread che possiede il lock a fare unlock!!**

Limiti dei locks

- Avvolte è necessario avere una forma di sincronizzazione più forte
 - Uno thread deve aspettare che un altro thread abbia finito di fare qualcosa
 - Ad esempio thread 1 produce una value che deve essere consumata da thread 2

Producer/Consumer problem

- Una soluzione:
 - ogni volta che il producer produce una nuova value, mette un boolean condiviso a true
 - e il consumer **aspetta** che il boolean sia a true per consumare
 - Ma cosa vuol dire aspettare ?
 - Può fare un ciclo while finché il boolean è a true
 - ogni volta prende il lock, verifica il boolean, rilascia il lock
 - si chiama aspetta attiva (active waiting)
 - **non è ottimale in termine di concorrenza**, perché usa risorse per non fare nulla

Esempio

- Consideriamo la struttura seguente

```
typedef struct shared_buf{  
    volatile int val;  
    volatile bool full;  
    pthread_mutex_t lock;  
} shared_buf;
```

- Abbiamo un oggetto di questa struttura condivisa fra varie thread
 - gli producer, scrivono in **val** se full è false e mettono **full** a true

```
void *produce(void *ptr);
```

- gli consumer leggono in **val** se full è true e mettono **full** a false

```
void *consume(void *ptr);
```

- Ogni value scritta deve essere letta da un solo consumer

Main

```
int main() {
    B.val=0;
    B.full=false;
    pthread_mutex_init(&(B.lock), NULL);
    pthread_t th1, th2, th3;
    char *prod1="Prod1";
    char *cons1="Cons1";
    char *cons2="Cons2";
    pthread_create(&th1, NULL, consume, cons1);
    pthread_create(&th2, NULL, produce, prod1);
    pthread_create(&th3, NULL, consume, cons2);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    pthread_join(th3, NULL);
    printf("END\n");
    pthread_mutex_destroy(&(B.lock));
    return 0;
}
```

```
return NULL;
```

```
}
```

Producer

```
void *produce(void *ptr){
    char *id=(char *)ptr;
    for(unsigned i=1;i<=10;++i){
        pthread_mutex_lock(&(B.lock));
        while(B.full){
            pthread_mutex_unlock(&(B.lock)); ← Attesa attiva
            pthread_mutex_lock(&(B.lock));
        }
        B.val=i;
        printf("--> Production of %d by %s\n",i,id);
        B.full=true;
        pthread_mutex_unlock(&(B.lock));
    }
    return NULL;
}
```

Consumer

```
void *consume(void *ptr){
    char *id=(char *)ptr;
    for(unsigned i=1;i<=5;++i){
        pthread_mutex_lock(&(B.lock));
        while(!B.full){
            pthread_mutex_unlock(&(B.lock)); ← Attesa attiva
            pthread_mutex_lock(&(B.lock));
        }
        int x=B.val;
        printf("--> Consumption of %d by %s\n",x,id);
        B.full=false;
        pthread_mutex_unlock(&(B.lock));
    }
    return NULL;
}
```

Come evitare l'attesa attiva ?

- Usare le **variabile condizionali** (monitor del C)
- Una variabile condizionale
 - è associata ad un mutex
 - permette a un thread di aspettare su questa variabile (in un modo passivo)
 - permette a un thread di svegliare dei thread che stanno aspettando su questa variabile
- Spesso le variabile condizionale vengono associate a delle condizione su variabile condivise da sorvegliare
- Uso: se una condizione necessaria allo thread non è verificata, il thread aspetta in un modo passivo finché la condizione viene verificata

Le variabile condizionali

- Dichiarazione:

```
pthread_cond_t varcond;
```

- Inizialisazion :

```
pthread_cond_init (&varcond, NULL) ;
```

- Per aspettare:

```
pthread_cond_wait (&varcond, &lock) ;
```

- **lock** è un lock preso al momento dello wait
- questa funzione libera lock e 'addormenta' lo thread

- Per svegliare:

```
pthread_cond_signal (&varcond) ;
```

- 'sveglia' al meno un thread addormentato sur varcond (se ce ne sono, altrimenti non fa nulla)

```
pthread_cond_broadcast (&varcond) ;
```

- 'sveglia' tutti gli thread addormentati sur varcond (se ce ne sono, altrimenti non fa nulla)

Quando uno thread è svegliato, è in lista d'attesa per essere scelto dallo scheduler e quando sarà il suo turno la prima cosa che farà è prendere il lock (ed aspettare se il lock è preso)

Le variabile condizionali - uso

- Struttura abituale del codice:

```
pthread_mutex_lock(&lock);  
while(...) { //si aspetta su questa condizione  
    pthread_cond_wait(&varcond, &lock);  
}  
  
...  
pthread_mutex_unlock(&unlock);
```

Esempio

- Modifichiamo la struttura del buffer

```
typedef struct shared_buf{  
    volatile int val;  
    volatile bool full;  
    pthread_mutex_t lock;  
    pthread_cond_t varcond;  
} shared_buf;
```

Main

```
int main() {
    B.val=0;
    B.full=false;
    pthread_mutex_init(&(B.lock), NULL);
    pthread_cond_init(&(B.varcond), NULL);
    pthread_t th1, th2, th3;
    char *prod1="Prod1";
    char *cons1="Cons1";
    char *cons2="Cons2";
    pthread_create(&th1, NULL, consume, cons1);
    pthread_create(&th2, NULL, produce, prod1);
    pthread_create(&th3, NULL, consume, cons2);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    pthread_join(th3, NULL);
    printf("END\n");
    pthread_mutex_destroy(&(B.lock));
    return 0;
}
```

Producer

```
void *produce(void *ptr){
    char *id=(char *)ptr;
    for(unsigned i=1;i<=10;++i){
        pthread_mutex_lock(&(B.lock));
        while(B.full){
            pthread_cond_wait(&(B.varcond), &(B.lock));
        }
        B.val=i;
        printf("--> Production of %d by %s\n", i, id);
        B.full=true;
        pthread_cond_signal(&(B.varcond));
        pthread_mutex_unlock(&(B.lock));
    }
    return NULL;
}
```

Consumer

```
void *consume(void *ptr){
    char *id=(char *)ptr;
    for(unsigned i=1;i<=5;++i){
        pthread_mutex_lock(&(B.lock));
        while(!B.full){
            pthread_cond_wait(&(B.varcond), &(B.lock));
        }
        int x=B.val;
        printf("--> Consumption of %d by %s\n", x, id);
        B.full=false;
        pthread_cond_signal(&(B.varcond));
        pthread_mutex_unlock(&(B.lock));
    }
    return NULL;
}
```

Problema

- La soluzione precedente non è corretta
- C'è un scenario in cui tutti sono bloccati in attesa
 - Cons1 è bloccato
 - -Cons2 è bloccato
 - Prod1 sveglia Cons1
 - **Prod1 è bloccato**
 - Cons1 sveglia Cons2
 - **Cons2 è bloccato**
 - **Cons1 è bloccato**

Soluzione possibile: usare broadcast al posto di signal

Esempio

- Altra possibilità: usare due variabile condizionali
- Modifichiamo la struttura del buffer

```
typedef struct shared_buf{  
    volatile int val;  
    volatile bool full;  
    pthread_mutex_t lock;  
    pthread_cond_t varcond_full;  
    pthread_cond_t varcond_notfull;  
} shared_buf;
```

Main

```
int main() {
    B.val=0;
    B.full=false;
    pthread_mutex_init(&(B.lock), NULL);
    pthread_cond_init(&(B.varcond_full), NULL);
    pthread_cond_init(&(B.varcond_notfull), NULL);
    pthread_t th1, th2, th3;
    char *prod1="Prod1";
    char *cons1="Cons1";
    char *cons2="Cons2";
    pthread_create(&th1, NULL, consume, cons1);
    pthread_create(&th2, NULL, produce, prod1);
    pthread_create(&th3, NULL, consume, cons2);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    pthread_join(th3, NULL);
    printf("END\n");
    pthread_mutex_destroy(&(B.lock));
    return 0;
}
```


Producer

```
void *produce(void *ptr){
    char *id=(char *)ptr;
    for(unsigned i=1;i<=10;++i){
        pthread_mutex_lock(&(B.lock));
        while(B.full){
            pthread_cond_wait(&(B.varcond_notfull), &(B.lock));
        }
        B.val=i;
        printf("--> Production of %d by %s\n", i, id);
        B.full=true;
        pthread_cond_signal(&(B.varcond_full));
        pthread_mutex_unlock(&(B.lock));
    }
    return NULL;
}
```

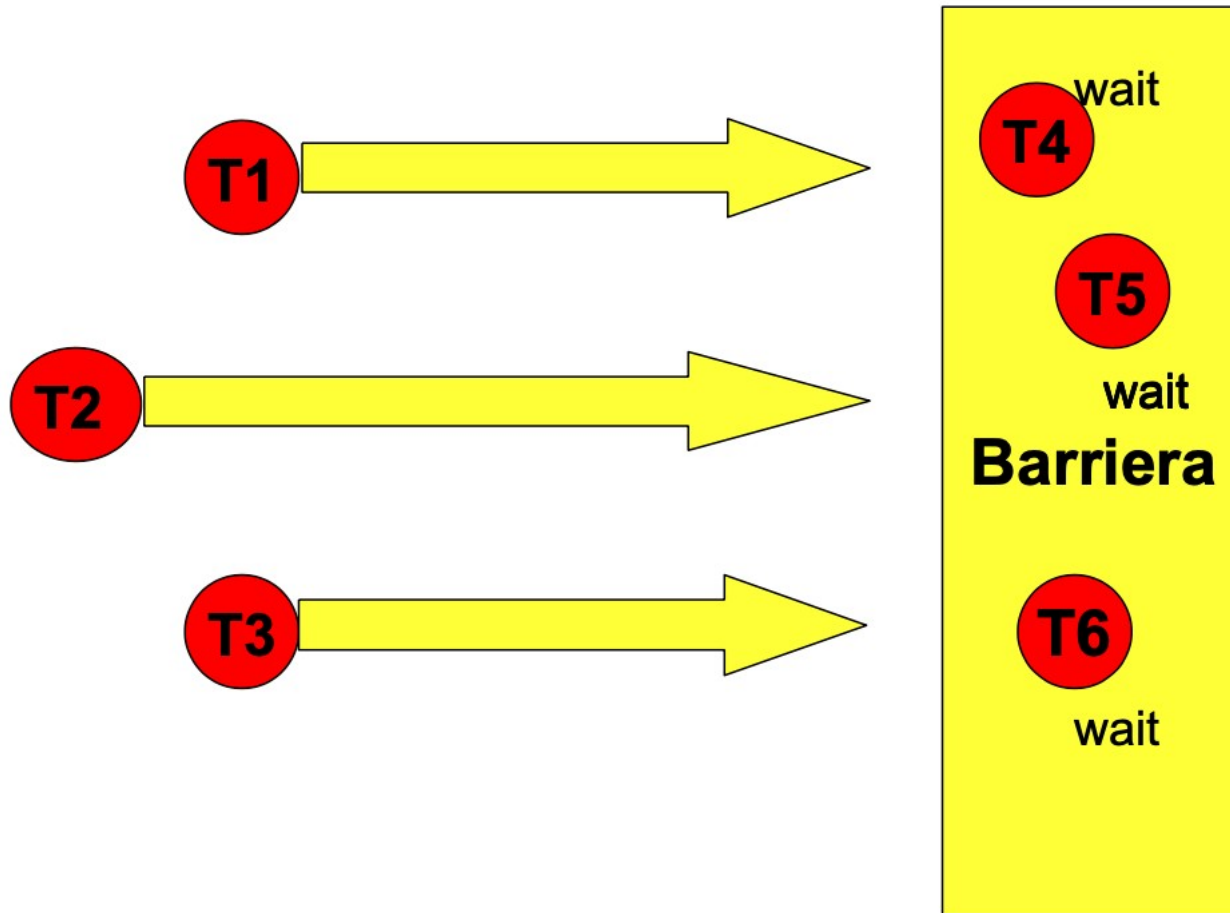
Consumer

```
void *consume(void *ptr){
    char *id=(char *)ptr;
    for(unsigned i=1;i<=5;++i){
        pthread_mutex_lock(&(B.lock));
        while(!B.full){
            pthread_cond_wait(&(B.varcond_full),&(B.lock));
        }
        int x=B.val;
        printf("--> Consumption of %d by %s\n",x,id);
        B.full=false;
        pthread_cond_signal(&(B.varcond_notfull));
        pthread_mutex_unlock(&(B.lock));
    }
    return NULL;
}
```

Le barriere

- Le barriere sono un meccanismo di sincronizzazione utilizzato ad esempio nel calcolo parallelo
- Idea: una barriera rappresenta un punto di sincronizzazione per N thread
 - I thread che arrivano alla barriera aspettano gli altri
 - Solo quando tutti gli N thread arrivano alla barriera allora possono proseguire

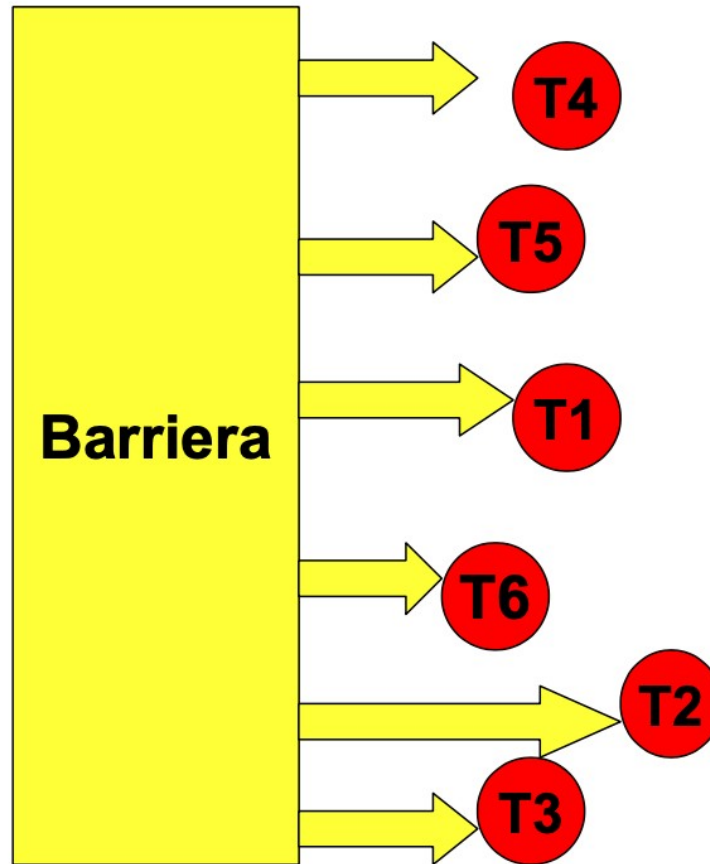
Le barriere



Le barriere



Le barriere



Le barriere

- Dichiarazione:

```
pthread_barrier_t bar;
```

- Inizializzazione :

```
pthread_barrier_init (&bar, NULL, 5) ; //per dire 5  
threads devono arrivare
```

- Per aspettare:

```
pthread_barrier_wait (&bar) ;
```

- se non ci sono 5 threads in attesa, blocca
- quando il quinto thread arriva, tutti vengono svegliati

**Possono essere implementate
facilmente con variabile
condizionale**