Thread: sincronizzazione
Esercitazioni del 09 Ottobre 2009

Luca Fossati, Fabrizio Castro, Vittorio Zaccaria

October 10, 2009

Esercizio 1: Sincronizzazione - 1

Qual'è il problema nella seguente porzione di codice?

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

int conto = 0;

void * th_fun1(void *arg){
    conto++;
    printf("th_fun1 ha depositato sul conto - saldo: %d\n", conto);
    return NULL;
}
void * th_fun2(void *arg){
    conto++;
    printf("th_fun2 ha depositato sul conto - saldo: %d\n", conto);
    return NULL;
}

int main(int argc, char * argv[]){
    pthread_t th1, th2;

    pthread_create(&th1, NULL, th_fun1, NULL);
    pthread_create(&th2, NULL, th_fun2, NULL);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);

    return 0;
}
```

Il problema consiste nel fatto che i due thread operano sulla variabile **conto** con una sequenza lettura/scrittura (operatore ++) che puo' rendere inconsistente lo stato della variabile stessa. Cio' dovrebbe avvenire introducendo una mutua esclusione sulla sequenza di accesso alla variabile **conto** come nella seguente soluzione:

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

int conto = 0;

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void * th_fun1(void *arg){
    int temp,i;
    pthread_mutex_lock(&lock);
    printf( "th_fun1: saldo del conto prima del deposito: %i\n", conto );
    conto++;
    pthread_mutex_unlock(&lock);
    printf( "th_fun1 ha depositato sul conto - saldo: %d\n", conto );
```

```
    return NULL;
}
void * th_fun2(void *arg){
    int temp,i;
    pthread_mutex_lock(&lock);
    printf( "th_fun2: saldo del conto prima del deposito: %i\n", conto );
    conto++;
    printf( "th_fun2 ha depositato sul conto - saldo: %d\n", conto );
    pthread_mutex_unlock(&lock);
    return NULL;
}

int main(int argc, char * argv[]){
    pthread_t th1, th2;

    pthread_create(&th1, NULL, th_fun1, NULL);
    pthread_create(&th2, NULL, th_fun2, NULL);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);

    return 0;
}
```

Esercizio 2: Mutex - 1

Problema del lettore-scrittore: un thread si occupa di scrivere una stringa, mentre altri si occupano della sua lettura. Non è possible avere letture e scritture contemporaneamente. Risolvere il problema usando solamente mutex.

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

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

char buffer[10];
int pronto = 0;

void * reader(void *arg){
    while( pronto == 0 );
    pthread_mutex_lock(&mutex);
    printf("Ho letto \"%s\"\n", buffer);
    pthread_mutex_unlock(&mutex);
    return NULL;
}

void * writer(void *arg){
    pthread_mutex_lock(&mutex);
    sprintf(buffer, "thread-%i", (int)pthread_self());
```

```
    pronto = 1;
    pthread_mutex_unlock(&mutex);
    return NULL;
}

int main(int argc, char * argv[]){
    pthread_t th_r1, th_r2, th_r3;
    pthread_t th_w1;

    pthread_create(&th_r1, NULL, reader, NULL);
    pthread_create(&th_r2, NULL, reader, NULL);
    pthread_create(&th_r3, NULL, reader, NULL);

    pthread_create(&th_w1, NULL, writer, NULL);

    pthread_join(th_r1, NULL);
    pthread_join(th_r2, NULL);
    pthread_join(th_r3, NULL);

    pthread_join(th_w1, NULL);

    return 0;
}
```

Esercizio 3: Sincronizzazione - 2

Qual'è il problema nella seguente porzione di codice?

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

char carattere = '\x0';

void * th_fun1(void *arg){
    printf("Letto %c\n", carattere);
    return NULL;
}

void * th_fun2(void *arg){
    int i;
    carattere = (rand()%93)+33;
    return NULL;
}

int main(int argc, char * argv[]){
    pthread_t th1, th2;

    pthread_create(&th1, NULL, th_fun1, NULL);
```

```
    pthread_create(&th2, NULL, th_fun2, NULL);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);

    return 0;
}
```

Il problema sta nel fatto che th_fun2 dovrebbe essere eseguita sempre prima di th_fun1 in modo tale da leggere un carattere valido dal buffer. La serializzazione e' un caso particolare di *sincronizzazione* che puo' essere gestito attraverso dei semafori. L'esercizio precedente puo' essere risolto tramite l'introduzione di un semaforo opportuno:

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

char carattere = '\x0';

sem_t semaforo;

void * th_fun1(void *arg){
    int i;
    sem_wait(&semaforo);
    printf("Letto %c\n", carattere);
    return NULL;
}
void * th_fun2(void *arg){
    int i;
    carattere = (rand()%93)+33;
    sem_post(&semaforo);
    return NULL;
}

int main(int argc, char * argv[]){
    pthread_t th1, th2;
    srand( time( NULL ) );
    sem_init(&semaforo, 0, 0);

    pthread_create(&th1, NULL, th_fun1, NULL);
    pthread_create(&th2, NULL, th_fun2, NULL);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);

    return 0;
}
```

Esercizio 4: Semafori - 1

Quattro amici fanno una scommessa, il premio per il vincitore è di poter fare bere uno dei rimanenti amici. Chi vince la scommessa sceglie casualmente a chi tocca bere il cocktail tutto d'un fiato mentre gli altri stanno a guardare. Alla fine, viene proposta un'ulteriore scommessa e il gioco va avanti all'infinito. Rappresentare il problema utilizzando i semafori.

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

sem_t sem_turno[3];
sem_t sem_bevuto;

void * amico(void *arg){
    while(1){
        sem_wait(&sem_turno[(int)arg]);
        // E' STATO SCELTO PER BERE
        sem_post(&sem_bevuto);
        // HA FINITO DI BERE
    }
    return NULL;
}
void * vincitore(void *arg){
    int turno = 0;
    while(1){
        sem_wait(&sem_bevuto);
        turno = rand()%3;
        sem_post(&sem_turno[turno]);
    }
    return NULL;
}

int main(int argc, char * argv[]){
    pthread_t th_amici[3];
    pthread_t th_vincitore;
    int i = 0;

    for(i = 0; i < 3; i++){
        sem_init(&sem_turno[i], 0, 0);
    }
    sem_init(&sem_bevuto, 0, 1);

    for(i = 0; i < 3; i++){
        pthread_create(&(th_amici[i]), NULL, amico, (void *)i);
    }
    pthread_create(&th_vincitore, NULL, vincitore, NULL);

    for(i = 0; i < 3; i++){
```

```
        pthread_join(th_amici[i], NULL);
    }
    pthread_join(th_vincitore, NULL);

    return 0;
}
```

Esercizio 5: Semafori - 2

Negozio del barbiere: il barbiere dorme fino a che non ci sono clienti: all'arrivo di un cliente esso può:

1. svegliare il barbiere (nel caso stesse dormendo);
2. sedersi e aspettare che il barbiere abbia finito con il cliente che sta radendo al momento;
3. se tutte le sedie della sala d'attesa sono occupate il cliente se ne va.

Implementare un programma che simula il comportamento appena descritto. Si fa l'ipotesi che la particolare implementazione dei semafori sblocchi i thread secondo l'ordine di chiamata della sem_wait.

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

#define NUMERO_DI_SEDIE 5
#define NUMERO_DI_CLIENTI 20

sem_t sem_barbiere_disponibile, sem_cliente_arrivato;
pthread_mutex_t stanza = PTHREAD_MUTEX_INITIALIZER;
int sedie_disponibili = NUMERO_DI_SEDIE;

void * barbiere( void * arg )
{
    while( 1 )
    {
        //IL BARBIERE DORME ASPETTANDO L'ARRIVO DI UN CLIENTE
        sem_wait( &sem_cliente_arrivato );
        //IL BARBIERE SI SVEGLIA E FA ACCOMODARE IL CLIENTE
        sleep(1) /* TEMPO DI SERVIZIO */
        sem_post( &sem_barbiere_disponibile );
        //IL BARBIERE E' PRONTO A FARE LA BARBA
    }
    return NULL;
}

void * cliente( void * arg )
{
    pthread_mutex_lock( &stanza );
```

```
    if( sedie_disponibili > 0 )
    {
        //IL CLIENTE SI SIEDE IN SALA D'ATTESA
        sedie_disponibili--;
        pthread_mutex_unlock( &stanza );
        //IL CLIENTE AVVISA IL BARBIERE DELLA SUA PRESENZA
        sem_post( &sem_cliente_arrivato );
        //IL CLIENTE ASPETTA CHE IL BARBIERE SIA DISPONIBILE
        sem_wait( &sem_barbiere_disponibile );
        pthread_mutex_lock( &stanza );
        //IL CLIENTE SI ALZA DALLA SEDIA E SI ACCOMODA SULLA POLTRONA DEL BARBIERE
        sedie_disponibili++;
        pthread_mutex_unlock( &stanza );
        //FINALMENTE IL CLIENTE RIESCE A FARSI LA BARBA
    }
    else
    {
        //IL CLIENTE ESCE DAL NEGOZIO DEL BARBIERE
        pthread_mutex_unlock( &stanza );
    }
    return NULL;
}

int main( int argc, char **argv )
{
    pthread_t th_clienti[ NUMERO_DI_CLIENTI ];
    pthread_t th_barbiere;
    int i;

    sem_init( &sem_barbiere_disponibile, 0, 0 );
    sem_init( &sem_cliente_arrivato, 0, 0 );

    pthread_create( &th_barbiere, NULL, barbiere, NULL );
    for( i = 0; i < NUMERO_DI_CLIENTI; i++ )
        pthread_create( &th_clienti[ i ], NULL, cliente, NULL );

    for( i = 0; i < NUMERO_DI_CLIENTI; i++ )
        pthread_join( th_clienti[ i ], NULL );

    pthread_join( th_barbiere, NULL );

    return 0;
}
```

La soluzione usa due semafori ed un mutex: i due semafori sono usati per sapere quando il barbiere è pronto (**sem_barbiere_disponibile**) e quanti clienti sono in attesa nella stanza(**sem_cliente_arrivato**). Il mutex **(stanza)** viene invece usato per garantire la mutua esclusione nella modifica e nell'accesso alle sedie della stanza.

Esercizio 6: Semafori: implementazione tramite mutex

La seguente porzione di codice mostra come creare un semaforo utilizzando due mutex ed un ciclo di *busy wait*. Il semaforo viene quindi utilizzato da due thread per sincronizzarsi in maniera seriale.

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

unsigned int valoreSemaforo;
pthread_mutex_t mutex_sem_mod = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex_sem_wait = PTHREAD_MUTEX_INITIALIZER;

void my_sem_init(unsigned int valoreIniziale){
    valoreSemaforo = valoreIniziale;
}

void my_sem_post(){
    pthread_mutex_lock(&mutex_sem_mod);
    valoreSemaforo++;
    pthread_mutex_unlock(&mutex_sem_mod);
}

void my_sem_wait(){
    pthread_mutex_lock(&mutex_sem_wait);
    while(valoreSemaforo <= 0)
        ;
    pthread_mutex_lock(&mutex_sem_mod);
    valoreSemaforo--;
    pthread_mutex_unlock(&mutex_sem_mod);
    pthread_mutex_unlock(&mutex_sem_wait);
}

void * th_fun1(void *arg){
    my_sem_post();
    return NULL;
}
void * th_fun2(void *arg){
    my_sem_wait();
    return NULL;
}

int main(int argc, char * argv[]){
    pthread_t th1, th2;
    my_sem_init(0);
    printf("Creazione thread\n");
    pthread_create(&th1, NULL, th_fun1, NULL);
    pthread_create(&th2, NULL, th_fun2, NULL);
    printf("Attesa sincronizzazione\n");
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
```

```
    printf("Fine programma\n");
    return 0;
}
```

Esercizio 7: Produttore/Consumatore - 1

Individuare il problema della seguente versione del problema produttore/consumatore:

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

char buffer;
int fine = 0;

void * scrivi(void *arg){
    int i;
    for (i=0; i < 5; i++){
        buffer = 'a' + i;
        printf( "scrivi: ho scritto '%c'\n", buffer );
        sleep(3);
    }
    fine = 1;
    return NULL;
}
void * leggi(void *arg){
    char miobuffer = '\x0';
    while (fine == 0){
        sleep(1);
        miobuffer = buffer;
        printf( "leggi: ho letto '%c'\n", miobuffer );
    }
    return NULL;
}
int main(void){
    pthread_t th1, th2;

    pthread_create(&th1, NULL, &scrivi, NULL);
    pthread_create(&th2, NULL, &leggi, NULL);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    return 0;
}
```

Non essendo usato alcun meccanismo di sincronizzazione, non è garantito l'accesso corretto alla variabile `buffer`. In particolare, il thread leggi potra' leggere piu' volte la stessa lettera dal buffer.

Esercizio 8: Produttore/Consumatore - 2

Soluzione corretta ed efficiente implementata utilizzando i semafori: sono necessari due semafori:

- `vuoto` per segnalare al thread `scrivi` (il produttore), che il consumatore ha letto il buffer;
- `pieno` per segnalare al thread `leggi` (il consumatore), che il produttore ha scritto il buffer.

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

char buffer;
int fine = 0;
sem_t pieno, vuoto;

void * scrivi(void *arg){
    int i;
    for (i=0; i < 5; i++){
        sem_wait(&vuoto);
        buffer = 'a' + i;
        sem_post(&pieno);
    }
    sem_wait(&vuoto);
    fine = 1;
    sem_post(&pieno);
    return NULL;
}

void * leggi(void *arg){
    char miobuffer = '\x0';
    while ( 1 ){
        sem_wait(&pieno);
        if( fine != 0 )
            return NULL;
        miobuffer = buffer;
        sem_post(&vuoto);
    }
}

int main(void){
    pthread_t th1, th2;

    sem_init(&pieno, 0, 0);
    sem_init(&vuoto, 0, 1);
    pthread_create(&th1, NULL, &scrivi, NULL);
    pthread_create(&th2, NULL, &leggi, NULL);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
```

```
    return 0;
}
```

Esercizio 9: Sequenze di esecuzione - 1

Dato il seguente programma, stampare tutte i possibili output, supponendo che l'istruzione `printf` di stampa a schermo sia atomica (cio supponendo che una volta inizata la stampa dell'argomento di una `printf`) essa termini prima che nel sistema venga eseguita un'altra direttiva `printf`.

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

pthread_mutex_t mutexA = PTHREAD_MUTEX_INITIALIZER;

void * th_fun1(void *arg){
    pthread_mutex_lock(&mutexA);
    printf("th_fun1 - 1\n");
    pthread_mutex_unlock(&mutexA);
    printf("th_fun1 - 2\n");
    return NULL;
}
void * th_fun2(void *arg){
    pthread_mutex_lock(&mutexA);
    printf("th_fun2 - 1\n");
    printf("th_fun2 - 2\n");
    pthread_mutex_unlock(&mutexA);
    return NULL;
}

int main(int argc, char * argv[]){
    pthread_t th1, th2;

    pthread_create(&th1, NULL, th_fun1, NULL);
    pthread_create(&th2, NULL, th_fun2, NULL);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);

    return 0;
}
```

Indicare almeno due sequenze di esecuzione e spiegare a parole quali sono tutte le possibili sequenze.

Soluzione

I thread *th1* e *th2* vengono mandati in esecuzione in contemporanea: a questo punto due sequenze di esecuzione sono possibili:

1. *th1* acuiqsice il lock sul `mutexA` prima di *th2*: viene stampato “th_fun1 - 1”. “th_fun2 - 1” e “th_fun2 - 2” vengono sicuramente stampati dopo “th_fun1 - 1” e in un’orine imprecisato rispetto a “th_fun1 - 2”. In poche parole tutte le sequenze sono valide purchè “th_fun1 - 1” venga stampato prima di “th_fun2 - 1” e “th_fun2 - 2”. Le possibili sequenze di esecuzione in questo caso sono:

th_fun1 - 1 th_fun1 - 2 th_fun2 - 1 th_fun2 - 2
th_fun1 - 1 th_fun2 - 1 th_fun1 - 2 th_fun2 - 2
th_fun1 - 1 th_fun2 - 1 th_fun2 - 2 th_fun1 - 2

2. *th2* acuiqsice il lock sul `mutexA` prima di *th1*: vengono stampati “th_fun2 - 1” e “th_fun2 - 2”. “th_fun1 - 1” viene sicuramente stampato dopo di essi; “th_fun1 - 2” segue. In questo caso esiste solo una sequenza di esecuzione:

th_fun2 - 1 th_fun2 - 2 th_fun1 - 1 th_fun1 - 2

In parole povere, il mutex serve per evitare che “th_fun1 - 1” venga stampata tra “th_fun2 - 1” e “th_fun2 - 2”.