

Lezione 9 in laboratorio: concorrenza. debugging, prod. cons. in vari modi

✿ attenzione ai dettagli ...

Dalle specifiche del protocollo HTTP

Request for Comments - RFC 2324 - **1 April 1998**

Hyper Text Coffee Pot Control Protocol (HTCPCP/1.0)

418 I'm a teapot



ERROR:

418: I'm a teapot

L'errore HTTP 418 I'm a teapot (418 sono una teiera) fa riferimento ad una burla che è stata inventata durante il primo periodo di diffusione dei protocolli Internet.

The HTTP '**418 I'm a teapot**' client error response code indicates that the server refuses to brew coffee because it is a teapot. This error is a reference to Hyper Text Coffee Pot Control Protocol which was an April Fools' joke in 1998.

Esercizio 418 - I'm a teapot - Correggere (1/2)

SyncPoint con 2 tipi di thread ed uscita a coppie

Esistono due tipi di thread, A e B, ed abbiamo **3** thread A e **3** thread B, ciascuno con un proprio indice intero univoco. Ciascun thread entra nella funzione SyncPoint e **attende che siano entrati ALMENO 4 thread**. Dopo che sono entrati **ALMENO 4 thread**, l'uscita dalla funzione SyncPoint avviene a coppie di thread: devono uscire insieme un thread A ed un thread B.

Esiste un vettore di 2 interi V[2] che contiene l'indice dei 2 thread che stanno per uscire.

Uscire insieme significa che ciascun thread della coppia che sta per uscire mette il proprio indice nel vettore V. **Solo dopo che entrambi hanno messo il loro indice nel vettore, hanno letto l'indice dell'altro e sanno che l'altro ha letto il loro proprio indice**, allora possono uscire dalla funzione SyncPoint. Quando ancora non c'è un thread che sta per uscire l'indice vale -1.

Scaricare il codice contenuto nell'archivio

http://www.cs.unibo.it/~ghini/didattica/sistemioperativi/ESERCIZI/es418_SyncPoint_a_coppie_ERRATO.tgz

Verificare che **NON** funziona correttamente e modificarlo per farlo funzionare.

Nel codice, le due funzioni *esco_come_primo_di_coppia()* e *esco_come_secondo_di_coppia()* usano una condition variable separata **cond_exit** per realizzare il meccanismo con cui

- 1) **il primo della coppia** avvisa di voler uscire e carica il proprio indice nel vettore,
- 2) **il secondo della coppia** a sua volta carica il proprio indice nel vettore, poi legge l'indice dell'altro nel vettore e conferma di essere pronto ad uscire e di avere letto l'indice dell'altro,
- 3) il primo riceve conferma dal secondo, legge l'indice del secondo nel vettore, conferma al secondo di avere letto l'indice e poi esce,
- 4) il secondo riceve conferma di lettura, rimette a -1 gli indici ed esce a sua volta.

Esercizio 418 - I'm a teapot - Correggere (2/2)

SyncPoint con 2 tipi di thread ed uscita a coppie

Le due funzioni *esco_come_primo_di_coppia()* e *esco_come_secondo_di_coppia()* FANNO ESATTAMENTE E CORRETTAMENTE PROPRIO QUELLO CHE E' STATO DETTO NELLA SPIEGAZIONE DELLA PAGINA PRECEDENTE.

L'ERRORE NON RISIEME NELLA IMPLEMENTAZIONE DELLE DUE FUNZIONI *esco_come_primo_di_coppia()* e *esco_come_secondo_di_coppia()*, bensì si trova in altri punti all'interno della funzione *SyncPont()*.

Suggerimento: se non capite cosa accade, editare il file `DBGpthread.c` e decommentare la definizione del simbolo `DEBUG` `'#define DEBUG'` in modo da attivare delle stampe di debug che possono aiutare la comprensione di cosa succede.

Le soluzioni sono nell'archivio

http://www.cs.unibo.it/~ghini/didattica/sistemioperativi/ESERCIZI/es418_SyncPoint_a_coppie_CORRETTO.tgz

Esercizio 418 - I'm a teapot - SOLUZIONI (1/2)

SyncPoint con 2 tipi di thread ed uscita a coppie

L'errore è dovuto al fatto che, nell'implementazione sbagliata, i thread, che entrano nella funzione SyncPoint dal 4° (SYNC_MAX+1) in avanti, non soddisfano la condizione del primo if e quindi saltano tutti nel ramo else svolgendo qui il ruolo di primo di coppia.

Invece i thread dovrebbero comportarsi diversamente. In particolare:

- I primi SYNC_MAX (3) thread che prendono la mutua esclusione nella SyncPoint() prima devono aspettare sulla wait(&cond_sync, ...), al risveglio devono controllare se possono svolgere il ruolo di primo o secondo di coppia e, in caso contrario devono tornare in wait.
- il 4° (SYNC_MAX+1) thread deve svolgere il ruolo di primo di coppia e interagire con secondo di coppia prima di uscire.
- I thread successivi al 4° (SYNC_MAX+1) invece prima devono controllare se possono svolgere il ruolo di primo o secondo di coppia e, solo in caso contrario devono mettersi in wait in attesa del risveglio.

GENERALIZZANDO:

I primi SYNC_MAX (3) thread che prendono la mutua esclusione nella SyncPoint() devono aspettare sulla wait(&cond_sync, ...). Poi, al risveglio, devono fare quello che fanno tutti gli altri, e cioè:

I thread devono controllare se possono svolgere il ruolo di primo o secondo di coppia e, solo in caso contrario, devono mettersi in wait, iterando questa operazione in un loop fino a che riescono a svolgere uno dei due ruoli (primo o secondo di coppia) e quindi terminano.

Esercizio 418 - I'm a teapot - SOLUZIONI (2/2)

SyncPoint con 2 tipi di thread ed uscita a coppie

```
void SyncPoint(int indice, char tipo, int indiceTipo, int indiceAltroTipo) {
    pthread_mutex_lock(&mutex);
    sync_count++;
    if (sync_count <= SYNC_MAX)
        pthread_cond_wait(&cond_sync, &mutex);
    while (1) {
        if( V[indiceTipo]==-1) {
            if( V[indiceAltroTipo]==-1 )
                esco_come_primo_di_coppia(indice,indiceTipo,indiceAltroTipo );
            else {
                esco_come_secondo_di_coppia(indice,indiceTipo,indiceAltroTipo);
                /* prima di uscire dico ad un altro in coda che puo' uscire */
                pthread_cond_signal (&cond_sync);
            }
            break; /* esco dal while */
        } else {
            pthread_cond_signal (&cond_sync );
            pthread_cond_wait(&cond_sync, &mutex);
        }
    } /* fine while */
    pthread_mutex_unlock (&mutex);
}
```

Esercizio 8: N ProduttoriA, M Produttori B, K Consumatori di A+B (seriale).

Un programma e' composto da tre tipi di thread, i ProduttoriA, i ProduttoriB ed i Consumatori. Esistono $N > 0$ ProduttoriA, $M > 0$ ProduttoriB, e K Consumatori.

Tutti i tre tipi di thread eseguono un loop infinito, durante il quale i ProduttoriA costruiscono un dato di tipo A (intero uint64_t), i ProduttoriB costruiscono un dato di tipo B (intero uint64_t) mentre i Consumatori vogliono un dato A assieme a un dato B.

Lo scambio dei dati prodotti avviene per tramite di 1 buffer per i dati A e di 1 buffer per i dati B. **Prelievo e Deposito avvengono in un tempo infinitesimo, quindi i depositi di A e B possono essere eseguiti in sequenza, non occorre eseguire il deposito di A e B in parallelo. Uso stessa mutex.**

Ciascun Consumatore, per svolgere il suo compito, deve prelevare nello stesso momento, un dato A dal buffer A ed un dato B dal buffer B.

Implementare il programma, creando 3 thread ProduttoriA, 5 thread ProduttoriB, e 10 thread Consumatori, implementando le necessarie sincronizzazioni.

I produttori impiegano 1 secondo a produrre il dato ed i consumatori impiegano 1 secondo per consumarlo dopo averlo prelevato dal buffer.

Prendere come base il codice dell'esempio NProdMCons1buffer.c ed estenderlo opportunamente.

SOLUZIONE IN

http://www.cs.unibo.it/~ghini/didattica/sistemioperativi/ESERCIZI/es8_consA+B.tgz

Soluzione Esercizio 8: N ProdA M ProdB K Cons di A+B, 1 Buffer A 1 Buffer B:

```
#define NUMBUFFER 1
void * ProdA ( void * arg ) {
    while(1) { /* qui il ProdA PRODUCE IL DATO, un intero. */ sleep(1);
        pthread_mutex_lock( &mutex);
        while ( numBufferAPieni >= NUMBUFFER )
            pthread_cond_wait ( &condProdA, &mutex);
        valGlobaleA = valProdottoA; /* SEZ CRITICA : riempie il buffer A col dato prodotto */
        numBufferAPieni++;
        pthread_cond_signal ( &condCons ); /* risveglio 1 Cons per svuotare 1 buffer */
        pthread_mutex_unlock ( &mutex ); /* rilascio mutua esclusione */
    }
}
void * ProdB ( void * arg ) {
    while(1) { /* qui il ProdB PRODUCE IL DATO, un intero. */ sleep(1);
        pthread_mutex_lock( &mutex);
        while ( numBufferBPieni >= NUMBUFFER )
            pthread_cond_wait ( &condProdB, &mutex);
        valGlobaleB = valProdottoB; /* SEZ CRITICA : riempie il buffer A col dato prodotto */
        numBufferBPieni++;
        pthread_cond_signal ( &condCons ); /* risveglio 1 Cons per svuotare 1 buffer */
        pthread_mutex_unlock ( &mutex ); /* rilascio mutua esclusione */
    }
}
void * Cons ( void * arg ) {
    while(1) {
        pthread_mutex_lock( &mutex);
        while ( ( numBufferAPieni <= 0 ) || ( numBufferBPieni <= 0 ) )
            pthread_cond_wait ( &condCons, &mutex);
        val = valGlobaleA+valGlobaleB; /* SEZ CRITICA : prendo ciò che sta nei due buffer di scambio */
        numBufferAPieni--;
        numBufferBPieni--;
        pthread_cond_signal ( &condProdA ); /* risveglio 1 ProdA per riempire il buffer */
        pthread_cond_signal ( &condProdB ); /* risveglio 1 ProdB per riempire il buffer */
        pthread_mutex_unlock ( &mutex ); /* rilascio mutua esclusione */
        sleep(1); /* consumo */
    }
}
```

Esercizio 8b: N Produttori A, M Produttori B, K Consumatori di A+B (**parallelo**).

Un programma e' composto da tre tipi di thread, i Produttori A, i Produttori B ed i Consumatori. Esistono $N > 0$ Produttori A, $M > 0$ Produttori B, e K Consumatori.

Tutti i tre tipi di thread eseguono un loop infinito, durante il quale i Produttori A costruiscono un dato di tipo A (intero `uint64_t`), i Produttori B costruiscono un dato di tipo B (intero `uint64_t`) mentre i Consumatori vogliono un dato A assieme a un dato B.

Lo scambio dei dati prodotti avviene per tramite di 1 buffer per i dati A e di 1 buffer per i dati B. Ciascun deposito dura 1 secondo, il prelievo in contemporanea dai due buffer dura 1 secondo. **Prelievo e Deposito avvengono in un tempo NON infinitesimo, occorre eseguire i DEPOSITI di A e B possibilmente in parallelo tra loro, per risparmiare tempo, non in sequenza.**

Ciascun Consumatore, per svolgere il suo compito, deve prelevare nello stesso momento, un dato A dal buffer A ed un dato B dal buffer B.

Implementare il programma, creando 3 thread Produttori A, 5 thread Produttori B, e 10 thread Consumatori, implementando le necessarie sincronizzazioni.

I produttori impiegano 1 secondo a produrre il dato ed i consumatori impiegano 1 secondo per consumarlo dopo averlo prelevato dal buffer.

Prendere come base il codice dell'esempio `NProdMCons1buffer.c` ed estenderlo opportunamente.

SOLUZIONE IN

http://www.cs.unibo.it/~ghini/didattica/sistemioperativi/ESERCIZI/es8PAR_consA+B.tgz

Soluzione Esercizio 8: N ProdA M ProdB K Cons di A+B, 1 Buffer A 1 Buffer B:

PARALLELO

```
#define NUMBUFFER 1
```

```
int numBufferAPieni=0, numBufferBPieni=0, accessoBufferA=0, accessoBufferB=0;
```

```
void * ProdA ( void * arg ) {  
    while(1) { /* qui il ProdA PRODUCE IL DATO, un intero. */ sleep(1);  
        pthread_mutex_lock( &mutex);  
        while ( numBufferAPieni >= NUMBUFFER || accessoBufferA==1 )  
            pthread_cond_wait ( &condProdA, &mutex);  
        accessoBufferA=1;  
        pthread_mutex_unlock( &mutex);  
        sleep(1); valGlobaleA = valProdottoA; /* SEZ CRITICA : riempie il buffer A col dato prodotto */  
        pthread_mutex_lock( &mutex);  
        numBufferAPieni++;  
        accessoBufferA=0;  
        pthread_cond_signal ( &condCons ); /* risveglio 1 Cons per svuotare 1 buffer */  
        pthread_mutex_unlock ( &mutex ); /* rilascio mutua esclusione */  
    }  
}  
void * ProdB ( void * arg ) { /* è simile ad A */  
}  
void * Cons ( void * arg ) {  
    while(1) {  
        pthread_mutex_lock( &mutex);  
        while ( (numBufferAPieni <= 0) || (numBufferBPieni <= 0) || (accessoBufferA!=0) || (accessoBufferB!=0) )  
            pthread_cond_wait ( &condCons, &mutex);  
        accessoBufferA=1; accessoBufferB=1;  
        pthread_mutex_unlock( &mutex);  
        sleep(1); val = valGlobaleA+valGlobaleB; /* SEZ CRITICA : prendo ciò che sta nei due buffer di scambio */  
        pthread_mutex_lock( &mutex);  
        numBufferAPieni--; numBufferBPieni--;  
        accessoBufferA=0; accessoBufferB=0;  
        pthread_cond_signal ( &condProdA ); /* risveglio 1 ProdA per riempire il buffer */  
        pthread_cond_signal ( &condProdB ); /* risveglio 1 ProdB per riempire il buffer */  
        pthread_mutex_unlock ( &mutex ); /* rilascio mutua esclusione */  
        sleep(1); /* consumo */  
    }  
}
```

Esercizio 9: N ProduttoriA, M ProduttoriB, K ConsumatoriA, L ConsumatoriB

Un programma e' composto da 4 tipi di thread, i ProduttoriA, i ProduttoriB, i ConsumatoriA ed i ConsumatoriB. Esistono $N > 0$ ProduttoriA, $M > 0$ ProduttoriB, K ConsumatoriA ed L ConsumatoriB.

Tutti i quattro tipi di thread eseguono un loop infinito, durante il quale i ProduttoriA costruiscono un tipo di dato A (un intero `uint64_t`), i ProduttoriB costruiscono un tipo di dato B (un intero `uint64_t`) mentre i ConsumatoriA necessitano di un dato A ed i ConsumatoriB necessitano di un dato B.

Lo scambio dei dati prodotti avviene per tramite di 1 UNICO buffer che puo' contenere o un dato A o un dato B.

Ciascun ConsumatoreX, per svolgere il suo compito, deve prelevare un dato X dal buffer.

Implementare il programma, creando 4 thread ProduttoriA, 5 thread ProduttoriB, 6 thread ConsumatoriA e 4 thread ConsumatoriB, implementando le necessarie sincronizzazioni.

Prendere come base il codice dell'esempio `NProdMCons1buffer.c` ed estenderlo opportunamente.

TROVATE UNA SOLUZIONE IN

http://www.cs.unibo.it/~ghini/didattica/sistemioperativi/ESERCIZI/es9_prodAprodB_consAconsB.tg

Soluzione Esercizio 9 N ProdA M ProdB K ConsA L ConsB, 1 SOLO Buffer

#define NUMBUFFER 1

Nel main, numBufferPieni viene inizializzato a 0

```
void * ProdA ( void * arg ) {
    while(1) {          /* qui il Proda PRODUCE IL DATO, un intero. */
        pthread_mutex_lock( &mutex);
        while ( numBufferPieni >= NUMBUFFER )
            pthread_cond_wait ( &condProd, &mutex);
        valGlobaleA = valProdottoA; /* SEZ CRITICA : riempie il buffer A col dato prodotto */
        tipo='A';
        numBufferPieni++;
        pthread_cond_signal ( &condCons ); /* risveglio 1 Cons per svuotare 1 buffer */
        pthread_mutex_unlock ( &mutex ); /* rilascio mutua esclusione */
    }
}
```

```
void * ConsA ( void * arg ) {
    while(1) {
        pthread_mutex_lock( &mutex);
        while ( ( numBufferPieni <= 0 ) || ((numBufferPieni >0)&&(tipo != 'A' )) ) {
            if ( numBufferPieni > 0 )
                pthread_cond_signal ( &condCons );
            pthread_cond_wait ( &condCons, &mutex);
        }
        val = valGlobaleA+valGlobaleB; /* SEZ CRITICA : prendo cio; che sta nei due buffer di scambio */
        numBufferPieni--;
        pthread_cond_signal ( &condProd ); /* risveglio 1 Prod a caso per riempire il buffer */
        pthread_mutex_unlock ( &mutex ); /* rilascio mutua esclusione */
    }
}
```

I thread ProdB e ConsB sono analoghi, ma occorre sostituire, nel codice, la costante 'A' con la costante 'B'.

correggere errori

Esercizio 6: correggere errori_2

- Scaricare e scompattare il file
http://www.cs.unibo.it/~ghini/didattica/sistemioperativi/ESERCIZI/ESERCIZI_CORR_EGGERE_ERRORI_2.tgz
- Viene creata una directory 2 contenente 3 directory 2.1 2.2 e 2.3
- In ciascuna sotto-directory ci sono un Makefile un modulo .c ed un file di intestazioni printerror.h
- Andate in ciascuna sotto-directory
- Cercate di capire cosa fa il codice.
- Provate a compilare, linkare ed eseguire il codice.
- Accadranno degli errori, o in compilazione, o in linking o in esecuzione.
- Correggete gli errori fino ad ottenere l'esecuzione corretta.

SOLUZIONI di correggere errori

Soluzione Esercizio 6: **correggere errori_2**

scaricare con wget l'archivio .tgz con i file corretti

http://www.cs.unibo.it/~ghini/didattica/sistemioperativi/ESERCIZI/ESERCIZI_CORRETTI_2.tgz