Università di Roma Tor Vergata
Corso di Laurea triennale in Informatica

Sistemi operativi e reti

A.A. 2016-17

Pietro Frasca

Lezione 22

Giovedì 12-01-2017

Un esempio di sincronizzazione tra thread

- Risolviamo il classico problema del produttore-consumatore.
- Nel caso più generale, più produttori e consumatori possono utilizzare un buffer in grado di contenere, al massimo, *N* messaggi.
- Ricordiamo che i vincoli per accedere al buffer sono due:
  - il produttore non può inserire un messaggio nel buffer se è pieno;
  - il consumatore non può prelevare un messaggio dal buffer vuoto.
- Supponendo, ad esempio, che i messaggi siano dei valori interi, realizziamo il buffer come un vettore di interi, gestendolo in modo circolare. Per gestire il buffer associamo ad esso le seguenti variabili:

- **cont**, il numero dei messaggi contenuti nel buffer;
- **scrivi**, indice del prossimo elemento da scrivere;
- **leggi**, indice del prossimo elemento da leggere.

Essendo il buffer una risorsa condivisa è necessario associare ad esso un **mutex M** per controllarne l'accesso in mutua esclusione.

Oltre al vincolo di mutua esclusione, i thread produttori e consumatori devono rispettivamente sospendersi nel caso in cui il buffer sia pieno o sia vuoto. Per realizzare tale sospensione associamo al buffer due variabili condizione di nome **PIENO,** per la sospensione dei produttori se il buffer è pieno, e di nome **VUOTO,** per la sospensione dei consumatori se il buffer è vuoto.

Da quanto detto rappresentiamo il buffer con una struttura che chiameremo **buffer**:

```
typedef struct {
   int messaggio[DIM];
   pthread_mutex_t M;
   int leggi, scrivi;
   int cont;
   pthread_cond_t PIENO;
   pthread_cond_t VUOTO;
} buffer ;
```

- Per gestire una struttura di tipo buffer, definiamo inoltre tre funzioni:
  - **init** per inizializzare la struttura buffer,
  - **produci**, funzione eseguita da un thread produttore per inserire un messaggio nel buffer
  - **consuma**, funzione eseguita da un thread consumatore per prelevare un messaggio dal buffer.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define FINE (-1)
#define MAX 20
#define DIM 10

typedef struct {
   pthread_mutex_t M;
   pthread_cond_t PIENO;
   pthread_cond_t VUOTO;
   int messaggio [DIM];
   int leggi, scrivi;
   int cont ;
} buffer;

buffer buf;
```

```
void init (buffer *buf);
void produci (buffer *buf, int mes);
int consuma (buffer *buf);

void init (buffer *buf){
   pthread_mutex_init (&buf->M,NULL);
   pthread_cond_init (&buf->PIENO, NULL);
   pthread_cond_init (&buf->VUOTO, NULL);
   buf->cont=0;
   buf->leggi=0;
   buf->scrivi=0;
}
```

```
void produci (buffer *buf, int mes) {
   pthread_mutex_lock (&buf->M);
   if (buf->cont==DIM)   /* il buffer e' pieno? */
       pthread_cond_wait (&buf->PIENO, &buf->M);
   /* scrivi mes e aggiorna lo stato del messaggio */
   buf->messaggio[buf->scrivi]=mes;
   buf->cont++;
   buf->scrivi++;
   /* la gestione è circolare */
   if (buf->scrivi==DIM) buf->scrivi=0;
   /* risveglia un eventuale thread consum. sospeso */
  pthread_cond_signal (&buf->VUOTO);
  pthread_mutex_unlock (&buf->M);
}
```

```
int consuma (buffer *buf){
   int mes;
   pthread_mutex_lock(&buf->M);
   if (buf->cont==0) /* il buffer e' vuoto? */
       pthread_cond_wait (&buf->VUOTO, &buf->M);
   /* Leggi il messaggio e aggiorna lo stato del buffer*/
   mes = buf->messaggio[buf->leggi];
   buf->cont--;
   buf->leggi++;
   /* la gestione è circolare */
   if (buf->leggi>=DIM) buf->leggi=0;
   /* Risveglia un eventuale thread produttore */
   pthread_cond_signal(&buf->PIENO) ;
   pthread_mutex_unlock(&buf->M);
  return mes;
}
```

```
void *produttore (void *arg){
   int n;
   for (n=0; n<MAX; n++){
        printf ("produttore %d -> %d\n", (int)arg,n);
        produci (&buf, n);
        sleep(1);
   }
   produci (&buf, FINE);
}
void *consumatore (void *arg){
   int d;
   while (1){
        d=consuma (&buf) ;
        if (d == FINE) break;
        printf ("      %d <- consumatore %d\n",d,(int)arg);
        sleep(2);
   }
}
```

```
int main () {
   int i;
   int nprod=1,ncons=1;
   pthread_t prod[nprod], cons[ncons];
   init (&buf);
   /*Creazione thread */
   for (i=0;i<nprod;i++)
       pthread_create(&prod[i], NULL, produttore,
   (void*)i);
   for (i=0;i<ncons;i++)
       pthread_create(&cons[i], NULL, consumatore,
   (void*)i);
   /* Attesa teminazione threads creati: */
   for (i=0;i<nprod;i++)
       pthread_join (prod[i], NULL);
   for (i=0;i<ncons;i++)
       pthread_join (cons[i], NULL);
   return 0;
}
```

Problema dei cinque filosofi

- Il problema dei 5 filosofi a cena è un esempio che mostra un problema di sincronizzazione tra thread (o processi). Cinque filosofi stanno cenando in un tavolo rotondo. Ciascun filosofo ha il suo piatto di spaghetti e una bacchetta a destra e un bacchetta a sinistra che condivide con i vicini. Ci sono quindi solo cinque bacchette e per mangiare ne servono 2 per ogni filosofo. Immaginiamo che durante la cena, un filosofo trascorra periodi in cui mangia e in cui pensa, e che ciascun filosofo abbia bisogno di due bacchette per mangiare, e che le bacchette siano prese una alla volta. Quando possiede due bacchette, il filosofo mangia per un po' di tempo, poi lascia le bacchette, una alla volta, e ricomincia a pensare.
- Il problema consiste nel trovare un algoritmo che eviti sia lo stallo (deadlock) che l'attesa indefinita (starvation).
- Lo stallo può verificarsi se ciascuno dei filosofi acquisisce una bacchetta senza mai riuscire a prendere l'altra. Il filosofo F1 aspetta di prendere la bacchetta che ha in mano il filosofo F2, che aspetta la bacchetta che ha in mano il

filosofo F3, e così via (condizione di attesa circolare).

- La situazione di starvation può verificarsi indipendentemente dal deadlock se uno dei filosofi non riesce mai a prendere entrambe le bacchette.
- La soluzione qui riportata evita il verificarsi dello stallo evitando la condizione di attesa circolare, imponendo che i filosofi con indice pari (considerando 0 pari) prendano prima la bacchetta alla loro destra e poi quella alla loro sinistra; viceversa, i filosofi con indice dispari prendano prima la bacchetta che si trova alla loro sinistra e poi quella alla loro destra.

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

```c
#define NUMFILOSOFI 5
#define CICLI 100

typedef struct{
    int id;
    pthread_t thread_id;
    char nome[20];
  } Filosofo;

/* Le bacchette sono risorse condivise, quindi ne gestiamo
   l'accesso in mutua esclusione mediante l'uso di mutex*/
pthread_mutex_t bacchetta[NUMFILOSOFI];

/* sospende per un intervallo di tempo random l'esecuzione del
   thread chiamante */
void tempoRnd(int min, int max) {
  sleep(rand()%(max-min+1) + min);
}
```

```
void *filosofo_th(void *id){
  Filosofo fil=*(Filosofo *)id;
  int i;
  for (i=0; i<CICLI; i++){
    printf("Filosofo %d: %s sta pensando \n",fil.id+1,fil.nome);
    tempoRnd(3, 12);
    printf("Filosofo %d: %s ha fame\n", fil.id+1,fil.nome);
    /* condizione che elimina l'attesa circolare */
    if (fil.id % 2){
      pthread_mutex_lock(&bacchetta[fil.id]);
      printf("Filosofo %d: %s prende la bacchetta destra (%d)\n",
        fil.id+1,fil.nome,fil.id+1);
      tempoRnd(1,2);
      pthread_mutex_lock(&bacchetta[(fil.id+1)%NUMFILOSOFI]);
      printf("Filosofo %d: %s prende la bacchetta sinistra
        (%d)\n", fil.id+1, fil.nome,(fil.id+1)%NUMFILOSOFI+1);
     }
```

```
    else{
      pthread_mutex_lock(&bacchetta[(fil.id+1) % NUMFILOSOFI]);
      printf("Filosofo %d: %s prende la bacchetta sinistra
       (%d)\n", fil.id+1, fil.nome,(fil.id+1)%NUMFILOSOFI+1);
      tempoRnd(1,2);
       pthread_mutex_lock(&bacchetta[fil.id]);
       printf("Filosofo %d: %s prende la bacchetta destra
        (%d)\n",fil.id+1,fil.nome,fil.id+1);
    }
    printf("Filosofo %d: %s sta mangiando \n", fil.id+1,
      fil.nome);
    tempoRnd(3, 10);
    pthread_mutex_unlock(&bacchetta[fil.id]);

    printf("Filosofo %d: %s posa la bacchetta destra (%d)\n",
      fil.id+1, fil.nome,fil.id+1);
    pthread_mutex_unlock(&bacchetta[(fil.id+1) % NUMFILOSOFI]);

    printf("Filosofo %d: %s posa la bacchetta sinistra (%d)\n",
      fil.id+1, fil.nome, (fil.id+1)%NUMFILOSOFI+1);
  } //ciclo for

}
```

```
int main(int argc, char *argv[]){
  int i;
  char nome[][20]={"Socrate","Platone","Aristotele","Talete",
   "Pitagora"};
  Filosofo filosofo[NUMFILOSOFI];
  srand(time(NULL));

  /* inizializza i mutex */
  for (i=0; i<NUMFILOSOFI; i++)
    pthread_mutex_init(&bacchetta[i], NULL);

  /* crea e avvia i threads */
  for (i=0; i<NUMFILOSOFI; i++){
    filosofo[i].id=i;
    strcpy(filosofo[i].nome,nome[i]);
    if (pthread_create(&filosofo[i].thread_id, NULL, filosofo_th,
       &filosofo[i]))
      perror("errore pthread_create");
    }
```

```
/* il thread main attende che i filosofi terminino */
  for (i=0; i<NUMFILOSOFI; i++)
    if (pthread_join(filosofo[i].thread_id, NULL))
      perror("errore pthread_join");
  return 0;
}
```

Gestione della memoria

- Lo spazio di indirizzamento di un processo Unix è segmentato, ed è costituito da tre segmenti separati: **codice, dati** e **stack**. Il segmento dati è diviso in due parti: i dati inizializzati e i dati non inizializzati (BSS, Block started by symbol). Tutte le variabili BSS sono inizializzate a zero dopo il caricamento.
- Il comando **size** visualizza le dimensioni dei segmenti codice (text), dati (data) e BSS di un file eseguibile .
- Molti programmi richiedono di allocare memoria dinamicamente durante l'esecuzione. La chiamata di sistema che consente l'allocazione dinamica della memoria è **brk**. La funzione di libreria C **malloc**, generalmente usata per allocare memoria, utilizza questa chiamata di sistema. L'area di memoria allocata in modo dinamico è detta **heap**.
- Il sistema di gestione della memoria utilizza il modello di segmentazione paginata. In particolare, l'allocazione dei segmenti avviene con la tecnica della paginazione su richiesta.

- Per ogni processo, la corrispondenza tra pagine virtuali e pagine fisiche è realizzata mediante la **tabella delle pagine**.
- Inoltre il kernel gestisce lo stato di allocazione delle pagine fisiche mediante la **tabella delle pagine fisiche** detta **core map**. Ogni elemento della **core map** specifica se la relativa pagina fisica è libera o allocata. Se la pagina fisica è allocata, l'elemento indica quale pagina virtuale è in essa contenuta e il PID del processo a cui appartiene.

- Il rimpiazzamento delle pagine è eseguito dal processo di sistema **page daemon** che utilizza una strategia che si basa sull'algoritmo di seconda scelta (orologio). Il page daemon ha pid pari a 2 e viene avviato da init che è il processo numero uno (pid=1).
- Il page daemon va in esecuzione periodicamente, ad esempio ogni 250 ms, ma interviene solo se il numero di pagine libere scende al disotto di una soglia prestabilita, il cui valore è stabilito dalla variabile di sistema **lotsfree.**
- Se l'intervento del page daemon non è sufficiente a mantenere bassa la frequenza di paginazione, nonostante il numero di pagine libere rimanga sotto la soglia stabilita da lotsfree, interviene lo **swapper** che sposta un certo numero di pagine di processi dalla memoria all'area di swap su disco.