Università di Roma Tor Vergata
Corso di Laurea triennale in Informatica

Sistemi operativi e reti

A.A. 2017-18

Pietro Frasca

Lezione 13

Giovedì 16-11-2017

Variabili condizione (condition)

- La variabile **condizione** è uno strumento di sincronizzazione che permette ai thread di sospendersi nel caso sia soddisfatta una determinata condizione logica.
- Come per il mutex o il semaforo, ad una variabile *condizione* è associata una coda nella quale i thread possono essere sospesi.
- Tuttavia, a differenza del mutex e del semaforo, la variabile condizione non ha uno stato (libero o occupato) ed è quindi rappresentata solo da una coda di thread sospesi.
- Anche per le variabili condizione le operazioni fondamentali sono la **sospensione** e la **riattivazione** di thread.
- Con la libreria pthread la variabile condizione si definisce mediante il tipo di dato ***pthread_cond_t*.**

- La definizione:

  `pthread_cond_t` *`C`*`;`

  crea la variabile condizione ***C***. Una volta definita, una variabile *condition* deve essere inizializzata, ad esempio mediante la SC:

  ```
  int pthread_cond_init (pthread_cond_t* C,
      pthread_cond_attr_t* attr) ;
  ```

  dove:
  - ***C*** è la variabile *condizione* da inizializzare;
  - ***attr*** è un puntatore alla struttura che contiene eventuali *attributi* specificati per la *condition* (se NULL, viene inizializzata ai valori di default).

  Un thread si sospende in una coda di una variabile *condition,* se la condizione logica associata ad essa è verificata.

Ad esempio, nel problema del produttore-consumatore nel caso in cui ci siano più produttori e più consumatori è necessario che i produttori si sospendano se il buffer dei messaggi è pieno; al contrario, i consumatori si devono sospendere se il buffer è vuoto. La sospensione dei produttori si può ottenere associando alla condizione di ***buffer pieno*** una variabile condizione, come nel seguente esempio:

```
/*variabili globali: */
pthread_cond_t C;  /* variabile condizione */
pthread_mutex_t M; /* mutex per accedere in
                      mutua esclusione alla
                      condizione logica*/
int bufferpieno=0; /* variabile per esprimere
                       la condizione logica */
 /* codice produttore: */
pthread_mutex_lock(&M);
if (bufferpieno) <sospensione sulla condition C>;
<inserimento messaggio nel buffer>;
pthread_mutex_unlock(&M);
```

- La variabile che si usa per stabilire l'accesso alla sezione critica (nell'esempio mostrato *bufferPieno*), essendo condivisa tra tutti i produttori, deve essere acceduta in mutua esclusione.
- Per garantire la mutua esclusione alla variabile condizione si associa un mutex.

  Pertanto, la chiamata *wait()* è realizzata con il seguente formato:

  ```
  int pthread_cond_wait (pthread_cond_t * C,
  pthread_mutex_t *M) ;
  ```

  dove:
  - *C* è la variabile condizione e
  - *M* è il mutex associato ad essa.

  Quando un thread *T* chiama la ***pthread_cond_wait()*** si ha che:

  - ***il thread T* viene sospeso nella coda associata a C, e**
  - **il mutex M viene liberato.**

- Quando il thread ***T*** viene riattivato, il mutex ***M*** è **automaticamente** posto a ***occupato***.
- In breve, un thread che si sospende chiamando ***pthread_cond_wait()*** libera il mutex ***M*** associato alla variabile condition ***C***, per poi rioccuparlo successivamente, quando sarà riattivato.

```
/*variabili globali: */
   pthread_cond_t C;
   pthread_mutex_t M; /* per accedere in  mutua
                         esclusione alla condizione
                         logica */
   int bufferpieno=0;

/* codice produttore: */
   pthread_mutex_lock(&M);
   if (bufferpieno) pthread_cond_wait(&C, &M);
   <inserimento messaggio nel buffer>;
   pthread_mutex_unlock(&M);
```

- Per riattivare un thread sospeso nella coda associata a una variabile condizione C si usa la funzione:

  `int pthread_cond_signal(pthread_cond_t * C)`

  Una chiamata a **pthread_cond_signal()** produce i seguenti effetti:
  - **viene riattivato il primo *thread* sospeso nella coda associata a C, se presente;**
  - **se non ci sono *thread* sospesi, la *signal()* non ha alcun effetto.**

- Per fare un esempio dell'uso della *condition,* consideriamo il caso in cui una risorsa può essere acceduta contemporaneamente da, al massimo, **MAX** thread. A tal fine, usiamo la variabile *condition* **PIENO**, nella cui coda saranno sospesi i thread che vogliono accedere alla risorsa quando questa è già usata da un numero MAX di thread.
- Indichiamo con la variabile intera **numTH** il numero di thread che stanno operando sulla risorsa:

```
#define MAX 10
// variabili globali:
int numTH=0; // numero di thread che usano la risorsa
pthread_cond_t PIENO; /* variabile condition */
pthread_mutex_t M; // mutex per mutua esclusione

void codice_thread () {
   pthread_mutex_lock(&M); /* prologo */
    /* controlla la condizione di accesso */
   if (numTH == MAX) pthread_cond_wait(&PIENO, &M);
   /* aggiorna lo stato della risorsa */
   numTH++;
   pthread_mutex_unlock(&M) ;
   <uso della risorsa>
   pthread_mutex_lock(&M); /* epilogo: */
   /* aggiorna lo stato della risorsa */
   numTH--;
   pthread_cond_signal(&PIENO);
   pthread_mutex_unlock(&M);
}
```

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 un tipo dato struttura che chiameremo **buffer_t**:

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

- Per gestire una struttura di tipo buffer, definiamo inoltre tre funzioni:
  - **Init,** per inizializzare la struttura tipo *buffer_t*,
  - **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_t;

buffer_t buf;
```

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

void init (buffer_t *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_t *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 del buffer 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_t *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 thread 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>
```

```
#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

}
```

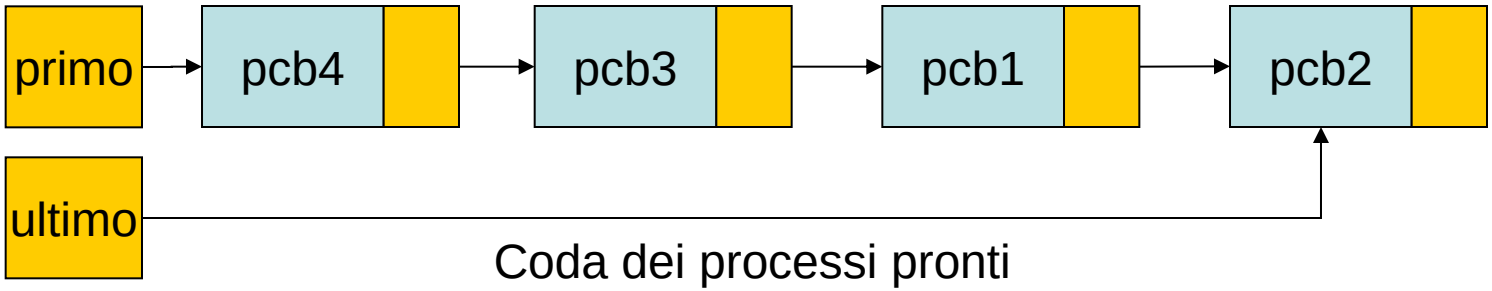
```
int main(int argc, char *argv[]){
  int i;
  char nome[][20]={"Socrate","Platone","Aristotele","Talete",
   "Pitagora"};
  Filosofo filosofo[NUMFILOSOFI];
  srand(time(NULL));

  /* 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;
}
```

Scheduling

Scheduling a breve termine

- Lo **scheduler a breve termine** (short term scheduler) è il componente del SO che si occupa di selezionare, dalla coda di pronto, il processo a cui assegnare la CPU.
- Lo scheduler è eseguito molto frequentemente e deve essere quindi realizzato in modo molto efficiente in termini di velocità d'esecuzione.
- Spesso si indica con il termine **scheduler** la parte che implementa le politiche (strategie) mentre il componente che implementa i meccanismi (cambio di contesto) prende il nome di **dispatcher**.



- In alcuni sistemi operativi, oltre lo scheduling a breve termine, sono previsti altri livelli di scheduling
  - **Scheduling a lungo termine (long term scheduling)**
  - **Scheduling a medio termine (medium term scheduler)**

Scheduling a lungo termine

- Nei sistemi batch è la funzione del SO che provvede a scegliere i programmi memorizzati in memoria secondaria da trasferire in memoria principale, da inserire nella coda dei processi pronti e quindi essere eseguiti.
- La selezione è eseguita in modo da bilanciare la presenza nella coda di pronto di processi di tipo **CPU-bound** e processi di tipo **I/O-bound,** per evitare che una prevalenza di uno dei due tipi di processo porti ad un uso non ottimo della CPU e delle risorse.

- Un altro importante compito dello scheduler a lungo termine è di controllare il **grado di multiprogrammazione**, cioè il numero di processi che sono presenti in memoria principale nello stesso tempo.

Scheduling a medio termine

- Lo **scheduling a medio termine** (medium term scheduling) si occupa di trasferire temporaneamente processi dalla memoria ram alla memoria secondaria (dischi), operazione di **swap-out** e viceversa, operazione di **swap-in**.
- Lo scheduler a lungo termine e lo scheduler a medio termine sono eseguiti con frequenze molto inferiori rispetto a quella dello scheduler a breve termine, in quanto sono molto complesse.

livelli di scheduling