LABORATORIO DI SISTEMI OPERATIVI

Corso di Laurea in Ingegneria Informatica

A.A. 2020/2021

Ing. Domenico Minici


domenico.minici@unifi.it

ESERCITAZIONE 9

Thread POSIX nel sistema Linux (parte II)

Sincronizzazione dei thread

- Il mutex è lo strumento messo a disposizione dalla libreria pthread per la sincronizzazione *indiretta* dei thread
  - Permette di garantire l'accesso in mutua esclusione a una risorsa condivisa

- Per la sincronizzazione *diretta* dei thread la libreria definisce le variabili condizione (condition variables)
  - Un thread può sospendersi in attesa del verificarsi di una determinata condizione
  - Permette di realizzare politiche avanzate di accesso alle risorse condivise e di sincronizzare i thread

Variabili condizione

- Una variabile condizione è di fatto una coda nella quale i thread possono sospendersi volontariamente in attesa di una condizione

- Definizione:
  `pthread_cond_t` `C`

- Funzione di inizializzazione:
  ```
  int pthread_cond_init(pthread_cond_t* C,
     pthread_cond_attr_t* attr)
  ```
  - `pthread_cond_t* C`
    Puntatore alla variabile condizione da inizializzare
  - `pthread_cond_attr_t* attr`
    Attributi specificati per la condizione, inizializzata a default se attr=NULL.

Variabili condizione

- Un thread può effettuare due "operazioni" su una variabile condition
  - Sospendersi (*wait*) sulla variabile condition. Il thread, dopo aver verificato una determinata condizione logica, si sospende sulla variabile condition, in attesa di essere "risvegliato" da un altro thread
  - Risvegliare uno dei thread (*signal*) o tutti i thread (*broadcast*) sospesi sulla variabile condition

Wait

- La sospensione (wait) viene utilizzata al verificarsi di una particolare condizione logica
  - Esempio: un thread produttore ha verificato che il buffer condiviso è pieno, quindi si blocca sulla condition "pieno" in attesa che un thread consumatore lo risvegli (dopo aver liberato il buffer)

- Schema generico di utilizzo della wait:

```
while (condizione logica)
    wait(condition_variable);
```

- Perché while e non if?
  - Quando il thread viene risvegliato non va subito in esecuzione (la signal della libreria pthread è di tipo signal & continue): altri thread potrebbero inserirsi e alterare la condizione (nell'esempio del produttore/consumatore, un altro thread produttore potrebbe riempire nuovamente il buffer)
  - E' quindi necessario ricontrollare la condizione dopo essere stati svegliati

Wait e mutua esclusione

- La "condizione logica" (es. `elementi_nel_buffer == MAX`) è basata su una risorsa condivisa (es. `elementi_nel_buffer`)
  - La verifica della condizione deve essere eseguita in mutua esclusione
- Tenendo conto di questo aspetto, la primitiva di wait offerta dalla libreria pthread permette di "associare" una variabile mutex a una variabile condition. In questo modo:
  - L'accesso in mutua esclusione sulla condizione logica viene rilasciato quando il thread si sospende con la wait
  - Il thread, dopo essere stato risvegliato, prova automaticamente ad eseguire di nuovo il lock sulla variabile mutex (se il mutex è occupato, il thread si blocca in attesa che venga liberato prima di proseguire)

Primitiva wait

- `int` **`pthread_cond_wait`**`(pthread_cond_t* C, pthread_mutex_t* M)`
  - `pthread_cond_t* C`
    Variabile condizione su cui sospendersi
  - `pthread_mutex_t* M`
    Mutex associato alla variabile condizione: viene liberato automaticamente quando il thread si sospende; viene eseguito un nuovo lock quando il thread viene risvegliato.

- La chiamata ha due effetti:
  - Il thread viene sospeso nella coda associata a C
  - Il mutex M viene liberato (quando il thread verrà risvegliato, proverà nuovamente a fare lock su M)

Risveglio – primitiva signal

- Il risveglio di un thread sospeso su una variabile condition C avviene mediante la primitiva signal:
  `int` **`pthread_cond_signal`**`(pthread_cond_t* C)`

- Come conseguenza della signal:
  - Se esistono thread in coda sulla condition, **(almeno)** uno viene risvegliato
  - Se non vi sono thread sospesi, non ha alcun effetto

- La politica della signal della libreria pthread è di tipo *signal & continue*
  - Il thread che esegue la signal continua la sua esecuzione e mantiene il controllo del mutex fino al suo esplicito rilascio
  - Il thread che aveva effettuato la wait ed è stato risvegliato deve verificare nuovamente la condizione

Risveglio – primitiva broadcast

- Per risvegliare tutti i thread in coda su una condition, è possibile utilizzare la funzione:
  `int` **`pthread_cond_broadcast`**`(pthread_cond_t* C)`

Esempio 1 – produttori e consumatori

- Dei thread accedono a una risorsa condivisa, ad esempio un buffer di interi gestito in modo circolare (ring buffer)
  - I thread consumatori prelevano valori (leggono) dal buffer
  - I thread produttori inseriscono nuovi valori (scrivono) nel buffer
- La gestione del buffer ha due vincoli:
  - Non si può prelevare dal buffer vuoto
  - Non si può inserire nel buffer pieno

Esempio 1 – produttori e consumatori

- Si può realizzare la risorsa condivisa in questo modo:

```
typedef struct {

  int buffer[BUFFER_SIZE];
  int readInd, writeInd;        // indici di read/write nel buffer
  int cont;                     // elementi nel buffer

  pthread_mutex_t M;            // per garantire accesso esclusivo alle risorse

  pthread_cond_t FULL;          // condition var. buffer pieno
  pthread_cond_t EMPTY;         // condition var. buffer vuoto

} risorsa;
```

Esempio 1 – produttori e consumatori

- La risorsa deve essere opportunamente inizializzata:

```
risorsa r; // variabile globale, condivisa da tutti i thread

int main() {
    pthread_mutex_init(&r.M, NULL);

    pthread_cond_init(&r.FULL, NULL);
    pthread_cond_init(&r.EMPTY, NULL);

    r.readInd = r.writeInd = r.cont = 0;
    …
}
```

Esempio 1 – consumatore

- Il consumatore deve
  - Assicurarsi che il buffer non sia vuoto prima di prelevare un dato
  - Risvegliare un produttore (se c'è) dopo aver prelevato un dato

```
…
int val; // per il dato che verrà prelevato dal buffer

pthread_mutex_lock(&r.M);
while (r.cont == 0) // buffer vuoto?
     pthread_cond_wait(&r.EMPTY, &r.M); // buffer vuoto, attendi…

// Preleva un dato e aggiorna lo stato del ring buffer
val = r.buffer[r.readInd];
r.cont--;
r.readInd = (r.readInd+1) % BUFFER_SIZE; // gestione circolare

// Risveglia un eventuale thread produttore
pthread_cond_signal(&r.FULL);
pthread_mutex_unlock(&r.M);
…
```

Esempio 1 – produttore

- Il produttore deve
  - Assicurarsi che il buffer non sia pieno prima di inserire un dato
  - Risvegliare un consumatore eventualmente sospeso

```
…
pthread_mutex_lock(&r.M);
while (r.cont == BUFFER_SIZE) // buffer pieno?
     pthread_cond_wait(&r.FULL, &r.M); // buffer pieno, attendi…

// Inserisci un dato e aggiorna lo stato del ring buffer
r.buffer[r.writeInd] = val;
r.cont++;
r.writeInd = (r.writeInd+1) % BUFFER_SIZE; // gestione circolare

// Risveglia un eventuale thread consumatore
pthread_cond_signal(r.EMPTY);
pthread_mutex_unlock(&r.M);
…
```

Esempio 2 – accesso limitato a risorsa

- L'utilizzo delle variabili condition permette di realizzare politiche di accesso a una risorsa più avanzate
- Ad esempio, si immagini uno scenario in cui
  - NTHREADS utilizzano (periodicamente) una risorsa
  - La risorsa può essere utilizzata contemporaneamente da un numero massimo MAX_T di thread
  - NTHREADS > MAX_T

Esempio 2 – accesso limitato a risorsa

- In questo caso si può utilizzare una variabile condition PIENO associata alla condizione:
  #thread che usano la risorsa == MAX_T
- Un thread esegue una "fase di ingresso" prima di usare la risorsa e una "fase di uscita" dopo aver usato la risorsa

Esempio 2 – accesso limitato a risorsa

- Variabili globali

```
#DEFINE MAX_T 10

int n_users = 0;        // numero di thread che stanno usando la risorsa
pthread_cond_t FULL;    // condition var. limite di utilizzo raggiunto
pthread_mutex_t M;      // Mutex per l'accesso esclusivo a n_users
```

Esempio 2 – accesso limitato a risorsa

- Fase di ingresso
  - Il thread controlla se è stato raggiunto il numero massimo di utilizzatori, ed eventualmente si sospende

```
…

pthread_mutex_lock(&M);
while (n_users == MAX_T) // massimo numero di users raggiunto
     pthread_cond_wait(&FULL, &M); // attendi…
n_users++;
pthread_mutex_unlock(&M); // rilascio il lock

…
// Uso della risorsa
…
```

Esempio 2 – accesso limitato a risorsa

- Fase di uscita
  - Il thread, dopo aver usato la risorsa, aggiorna lo stato della risorsa e risveglia un thread eventualmente sospeso

```
…

// Uso della risorsa
…

pthread_mutex_lock(&M);
n_users--;
pthread_cond_signal(&FULL);
pthread_mutex_unlock(&M);
```

ESERCIZI

Esercizio 1

- Completare il codice in es1.c in modo che le funzioni deposit e withdraw operino correttamente sul saldo (balance) di un conto corrente

Esercizio 2

- Completare il file es2.c in modo da ottenere questa sincronizzazione fra i thread:
  - I thread, prima di terminare, "aspettano" che anche gli altri abbiano terminato la sleep
    - Utilizzare un intero condiviso "checked_threads" per contare i thread che hanno terminato la sleep
    - Utilizzare una variabile condition per sospendere i thread in attesa degli altri
    - L'ultimo thread a raggiungere il "checkpoint" dopo la sleep provvede a svegliare gli altri, in modo che tutti possano terminare.

Esercizio 3

- Il codice in es3.c simula il comportamento di alcuni giocatori che provano a indovinare un numero sorteggiato dal main
  - Il main sceglie un numero
  - I thread figli provano a indovinare
  - Se nessuno ha indovinato, il main fornisce un suggerimento (intervallo) e la scommessa viene ripetuta
- Completare il codice dei thread "giocatori" in modo da garantire la corretta sincronizzazione fra thread "giocatori" e thread main
  - Il thread giocatore deve inserire la sua "scommessa" nel buffer e risvegliare il thread main se tutti i giocatori hanno già fatto la loro scommessa
  - Il thread giocatore deve attendere che i risultati siano pronti prima di poter giocare di nuovo (se è necessario un altro turno per stabilire il vincitore)