Esercitazione [4]

Produttore/Consumatore

Leonardo Aniello - aniello@dis.uniroma1.it
Daniele Cono D'Elia - delia@dis.uniroma1.it

Sistemi di Calcolo - Secondo modulo (SC2)
Programmazione dei Sistemi di Calcolo Multi-Nodo

Corso di Laurea in Ingegneria Informatica e Automatica

A.A. 2014-2015

Sommario

- Soluzione esercizio di riepilogo su processi/thread/semafori
- Produttore/Consumatore: breve riepilogo
- Obiettivi dell'esercitazione
- Singolo Produttore/Singolo Consumatore in C
- Esercizio: Più Produttori/Più Consumatori

Esercizio di riepilogo [1/10]

- Esercizio di riepilogo su processi, thread e semafori
- Sviluppare un'applicazione in C con questa semantica
  1. Il processo main crea N processi figlio tramite fork
  2. Tutti i processi figlio si sincronizzano per iniziare la loro attività, avviata dal processo main
  3. L'attività dei processi figlio consiste nel lanciare M thread per volta
     a. Competono per l'accesso in sezione critica, gestito dal processo main
     b. Una volta in sezione critica, devono scrivere in append su un file l'identità del processo scrivente
  4. Passati T secondi, il processo main deve notificare i processi figlio di cessare la loro attività e terminare (usare la sem_getvalue)
     a. Prima di terminare, un processo deve attendere la fine dei thread attualmente in esecuzione
  5. Infine, il processo main deve identificare il processo che ha effettuato più accessi in sezione critica

Esercizio di riepilogo [2/10]

1. Il processo main crea N processi figlio tramite fork

```
for (i = 0; i < n; i++) {
      pid_t pid = fork();
      if (pid == -1) {
            exit(EXIT_FAILURE);
      } else if (pid == 0) {
            // child process, its id is i
            break;
      } else {
            // main process, go on creating processes
            continue;
      }
}
```

Esercizio di riepilogo [3/10]

2. Tutti i processi figlio si sincronizzano per iniziare la loro attività, avviata dal processo main

- Richiede due diverse sincronizzazioni da effettuare in sequenza
  1. Il main deve aspettare che tutti i figli siano partiti (prima istruzione eseguita)
  2. I figli devono aspettare il «via» dal main, in modo che tutti possano avviare le proprie attività approssimativamente nello stesso istante
     - L'approssimazione è dovuta al fatto che il main «sveglia» un processo per volta e al non-determinismo nell'allocazione dei core ai thread
     - Si può considerare un best-effort, comunque migliore rispetto al non imporre alcuna sincronizzazione all'avvio

Esercizio di riepilogo [4/10]

2. Tutti i processi figlio si sincronizzano per iniziare la loro attività, avviata dal processo main

- Per la prima sincronizzazione
  - Il main deve bloccarsi → `sem_wait`
  - I processi figlio devono sbloccare il main → `sem_post`
  - Come usare il semaforo `main_waits_for_children`
    - Inizialmente il main deve bloccarsi anche se nessun figlio ha ancora notificato il proprio avvio → semaforo inizializzato a 0
    - Il main deve aspettare che tutti i figli abbiano notificato il proprio avvio

```
for (i = 0; i < n; i++)
    sem_wait(main_waits_for_children);
```

    - Ogni figlio deve notificare il proprio avvio

```
sem_post(main_waits_for_children);
```

Esercizio di riepilogo [5/10]

2. Tutti i processi figlio si sincronizzano per iniziare la loro attività, avviata dal processo main

- Per la seconda sincronizzazione
  - I processi figlio devono bloccarsi → `sem_wait`
  - Il main deve sbloccare i figli → `sem_post`
  - Come usare il semaforo `children_wait_for_main`
    - Ogni processo figlio deve potersi bloccare → semaforo inizializzato a 0

      ```
      sem_wait(children_wait_for_main);
      ```

    - Il main deve consentire a tutti i processi figlio di sbloccarsi

      ```
      for (i = 0; i < n; i++)
          sem_post(children_wait_for_main);
      ```

Esercizio di riepilogo [6/10]

3. L'attività dei processi figlio consiste nel lanciare M thread per volta

```
pthread_t* thread_handlers =
    malloc(m * sizeof(pthread_t));
……
for (j = 0; j < m; j++) {
    thread_args_t *t_args = ...;
    t_args->process_id = process_id;
    t_args->thread_id = thread_id++;
    pthread_create(&thread_handlers[j], NULL,
          thread_function, t_args);
}
```

- E poi deve attenderne il termine

```
for (j = 0; j < m; j++)
    pthread_join(thread_handlers[j], NULL);
```

Esercizio di riepilogo [7/10]

Operazioni del singolo thread

- Accesso in sezione critica
- Scrittura in append su un file dell'identità del processo

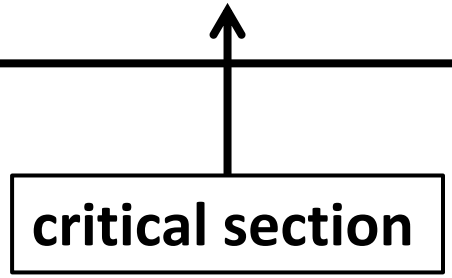
```
thread_args_t *args = (thread_args_t*)arg_ptr;

sem_wait(critical_section);

    int fd = open(FILENAME, O_WRONLY | O_APPEND);
    write(fd, &(args->process_id), sizeof(int));
    close(fd);

sem_post(critical_section);

free(args);
```



Esercizio di riepilogo [8/10]

4. Passati T secondi, il main deve notificare i processi figlio di cessare la loro attività e terminare

- Semaforo `end_children_activities`
  - Valore 0: continuare le attività (valore iniziale)
  - Valore 1: terminare le attività
- Il main aspetta T secondi (`sleep`) e notifica i figli di terminare le loro attività (`sem_post`)

```
sleep(t);
sem_post(end_children_activities);
```

- Prima di terminare, un processo deve attendere la fine dei thread attualmente in esecuzione
  - Dopo il ciclo di pthread_join, il processo figlio può verificare la presenza di tale notifica con una `sem_getvalue`

```
sem_getvalue(end_children_act,&main_notification);
if (main_notification) break;
```

Esercizio di riepilogo [9/10]

5. il processo main deve identificare il processo che ha effettuato più accessi in sezione critica

- Attesa del termine effettivo di tutti i figli

```
int child_status;
for (i = 0; i < n; i++)
    wait(&child_status);
```

- Lettura statistiche di accesso da file

```
int *access_stats = (int*)calloc(n, sizeof(int));
int fd = open(FILENAME, O_RDONLY);
size_t read_bytes;  int read_byte;
do { read_bytes = read(fd, &read_byte, sizeof(int));
    if (read_bytes > 0) access_stats[read_byte]++;
} while(read_bytes > 0);
close(fd);
```

Esercizio di riepilogo [10/10]

5. il processo main deve identificare il processo che ha effettuato più accessi in sezione critica

- Identificazione del processo che ha effettuato più accessi

```
int max_process_id = -1, max_accesses = -1;
for (i = 0; i < n; i++) {
    if (access_stats[i] > max_accesses) {
        max_accesses = access_stats[i];
        max_process_id = i;
    }
}
```

- Cleanup
  - Close e unlink di tutti i semafori
  - Free della memoria allocata

Produttore/Consumatore

- Punti chiave
  - Un buffer (array di entry) di dimensione finita/infinita
    - Nella pratica, la dimensione è sempre finita!!!
  - Produttori: producono entry da inserire poi nel buffer
  - Consumatori: consumano entry dopo averle tolte dal buffer
- Proprietà
  - Un consumatore non può togliere entry da un buffer vuoto
    - Se il buffer è vuoto, il consumatore deve aspettare
  - Un produttore non può inserire entry in un buffer pieno
    - Se il buffer è pieno, il produttore deve aspettare
  - Un consumatore (produttore) può togliere (inserire) un'entry alla volta
  - Un'entry può essere tolta e consumata (prodotta e inserita) da un solo consumatore (produttore)

Obiettivi Esercitazione [4]

- Usare i semafori per implementare soluzioni al problema del produttore/consumatore nelle seguenti configurazioni
  - un produttore / un consumatore
  - più produttori / un consumatore
  - un produttore / più consumatori
  - più produttori / più consumatori

Buffer Circolare

- Buffer circolare di dimensione massima `N`
  - `write_index ∈ [0,N-1]` è la posizione nel buffer dove verrà inserita la prossima entry
    - Un produttore inserisce l'entry prodotta in posizione `write_index`, poi incrementa `write_index` di `1` (modulo `N`)
  - `read_index ∈ [0,N-1]` è la posizione nel buffer dalla quale verrà tolta la prossima entry
    - Un consumatore toglie l'entry in posizione `read_index`, poi incrementa `read_index` di `1` (modulo `N`); infine la consuma
  - Se i consumatori non tolgono entry dal buffer quando è vuoto, il `read_index` non «scavalcherà» mai il `write_index`
    - Non vengono tolte (e quindi consumate) entry che in realtà non sono mai state inserite
  - Se i produttori non inseriscono entry nel buffer quando è pieno, il `write_index` non «doppierà» mai il `read_index`
    - Non vengono sovrascritte entry non ancora tolte (e quindi non ancora consumate)

Un Produttore / Un Consumatore

- Un semaforo per contare le posizioni occupate nel buffer
  - Inizializzato a `0`
  - Il consumatore si blocca su questo sem se il buffer è vuoto
  - Il produttore incrementa questo sem quando inserisce entry
- Un semaforo per contare le posizioni libere nel buffer
  - Inizializzato a `N`
  - Il produttore si blocca su questo sem se il buffer è pieno
  - Il consumatore incrementa questo sem quando toglie entry
- Codice: `one_producer_one_consumer.c`
- Compilazione: richiede la libreria `pthread`
- Esecuzione: nessun parametro richiesto

Un Produttore / Un Consumatore Approfondimento (1/2)

- Nelle slide sulla concorrenza presentate a lezione (num 36, fig 5.13) viene mostrata una soluzione al problema «un consumatore/un produttore» con buffer finito
- In quel caso viene usato un ulteriore semaforo `s` per imporre mutua esclusione nell'accesso alla struttura dati tramite le funzioni `append()` e `take()`
- Perché nella nostra soluzione non serve?

Un Produttore / Un Consumatore Approfondimento (2/2)

- In base all'implementazione, una struttura dati condivisa può essere acceduta in modi diversi
  - Nella soluzione presentata a lezione, tramite le funzioni append() e take()
  - Nella soluzione presentata qui, tramite l'accesso esplicito ad opportune posizioni di un array
- Nella soluzione presentata a lezione, non sappiamo come quelle funzioni siano state implementate
  - In particolare, non sappiamo cosa succede se append() e take() vengono eseguite in parallelo da due thread
  - In mancanza di altre informazioni, possiamo adottare un approccio conservativo: evitiamo che quelle funzioni possano essere accedute in parallelo usando un ulteriore semaforo
- Nella soluzione presentata qui, sappiamo come la struttura dati condivisa viene acceduta
  - Siccome produttore e consumatore non accedono mai in parallelo alla stessa posizione del buffer (esercizio di ragionamento: perché?), non abbiamo bisogno di ulteriori semafori

Più Produttori / Più Consumatori

- Con più produttori, è possibile che più entry vengano scritte nella stessa posizione
  - Sovrascrittura di entry = perdita di entry
  - Necessità di mettere in mutua esclusione la porzione di codice che effettua l'inserimento di entry nel buffer
- Con più consumatori, è possibile che una stessa entry venga tolta (letta e poi consumata) più volte
  - Duplicazione di entry
  - Possibile perdita di entry come conseguenza della duplicazione! (dipende dall'implementazione)
  - Necessità di mettere in mutua esclusione la porzione di codice che effettua la lettura di entry dal buffer

Esercizio

- Estendere il sorgente `one_producer_one_consumer.c` per implementare le soluzioni alle seguenti configurazioni del problema produttore/consumatore
  - più produttori/un consumatore
  - un consumatore/più produttori
  - più consumatori/più produttori

  Nota: la semantica del codice proposto non supporta più consumatori che aggiornano il deposito in parallelo (a meno che esso non diventi una globale acceduta in concorrenza). Per semplicità vi si chiede di implementare nel codice del thread consumatore soltanto la gestione dei semafori aggiuntivi necessari per più consumatori, anche se una sola istanza del consumatore sarà poi lanciata dal main.