Esercitazione [9]

Riepilogo sui Semafori

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 sul Logger
- Soluzione esercizio sull'EchoProcess su FIFO
- Riepilogo semafori
  - Sezione critica
  - Limite sugli accessi concorrenti
  - Produttore/Consumatore
    - Singolo Produttore/Singolo Consumatore
    - Più Produttori/Più Consumatori
- Esercizio sul Produttore/Consumatore
- Esercizio sul limite sugli accessi concorrenti

Soluzione Esercizio sul Logger

- Nel `main()` - *processo Server*
  - Dopo la chiamata `pipe()` va chiuso
    - Il descrittore della socket nel figlio
    - I descrittori del file di log e di lettura da pipe nel padre
  - Dopo il `while(1)`, va chiuso il descrittore di scrittura su pipe
    - Per «pulizia», anche se il codice non viene mai eseguito!!!
- Nello `startLogger()` - *processo Logger*
  - Chiudere il descrittore di scrittura su pipe
  - Lettura da pipe con gestione casi particolari
    - La pipe è stata chiusa → `break`
    - La lettura è stata interrotta da un segnale → `continue`
    - Si è verificato un altro errore → `exit`
  - All'uscita, chiudere il descrittore di lettura da pipe

Soluzione Esercizio sull'EchoProcess su FIFO

- Creazione FIFO tramite `mkfifo()`
- Apertura FIFO tramite `open()`
  - Il server apre `fifo_client` in lettura ed `fifo_echo` in scrittura
  - Il client apre `fifo_client` in scrittura ed `fifo_echo` in lettura
  - Entrambi seguono lo stesso ordine di apertura, pena un deadlock!
- Scrittura su FIFO
  - La dimensione del messaggio è nota, quindi la `write()` va gestita finché non sono stati scritti tutti i byte
  - Solita gestione di interruzioni ed errori
- Lettura da FIFO
  - La dimensione del messaggio non è nota, quindi ha senso gestire letture parziali (cioè dei soli dati disponibili in quel momento)
  - Solita gestione di interruzioni ed errori

Riepilogo Semafori

- Inizializzazione: assegna un valore non negativo
  - Unnamed semaphore: funzione `sem_init()`
  - Named semaphore: funzione `sem_open()`
- semWait: decrementa il valore, se è negativo il thread viene messo in attesa, altrimenti va avanti
  - Funzione `sem_wait()`
- semSignal: incrementa il valore, se non è positivo un thread viene risvegliato
  - Funzione `sem_post()`
- Chiusura: `sem_destroy(`*unnamed*`)`, `sem_close(`*named*`)`
  - Per i named semaphore, va invocata alla fine la `sem_unlink()` nel processo responsabile della sua rimozione

Uso dei semafori per implementare una sezione critica

- Una sezione critica è una porzione di codice che deve essere eseguita in mutua esclusione
  - Non devono esserci più thread che eseguono quella porzione contemporaneamente
- Implementazione
  - Il semaforo va inizializzato a 1
  - `sem_wait` all'inizio della sezione critica
  - `sem_post` alla fine della sezione critica

Uso dei semafori per imporre un limite all'accesso concorrente

- Si vuole mettere un limite al numero di thread che eseguono in contemporanea una certa porzione di codice
  - Non devono esserci più di N thread che eseguono quella porzione contemporaneamente
- Implementazione
  - Il semaforo va inizializzato a N
  - `sem_wait` all'inizio della porzione di codice
  - `sem_post` alla fine della porzione di codice

Produttore/Consumatore

- Thread produttori devono inserire delle entry nel buffer
  - Puntatore `w_idx`, inizializzato a 0
- Thread consumatori devono togliere delle entry dal buffer
  - Puntatore `r_idx` , inizializzato a 0

- Un consumatore non può togliere entry da un buffer vuoto, deve aspettare
- Un produttore non può inserire entry in un buffer pieno, deve aspettare

- Un consumatore può togliere una entry alla volta
- Un produttore può inserire una entry alla volta

- Una entry può essere tolta da un solo consumatore
- Una entry può essere inserita da un solo produttore

Singolo Produttore/Singolo Consumatore

- Un semaforo per contare le posizioni occupate nel buffer, `fill_count`, inizializzato a 0
- Un semaforo per contare le posizioni libere nel buffer, `empty_count`, inizializzato a N (dimensione massima del buffer)

Produttore

```
sem_wait(empty_count)

buffer[w_idx] = entry
w_idx = (w_idx + 1) mod N

sem_post(fill_count)
```

Consumatore

```
sem_wait(fill_count)

entry = buffer[r_idx]
r_idx = (r_idx + 1) mod N

sem_post(empty_count)
```

Più Produttori/Più Consumatori

- L'inserimento di una entry è una sezione critica da proteggere con un semaforo `w_mutex`
- La rimozione di una entry è una sezione critica da proteggere con un semaforo `r_mutex`

Produttore

```
sem_wait(empty_count)
sem_wait(w_mutex)
buffer[w_idx] = entry
w_idx = (w_idx + 1) mod N
sem_post(w_mutex)
sem_post(fill_count)
```

Consumatore

```
sem_wait(fill_count)
sem_wait(r_mutex)
entry = buffer[r_idx]
r_idx = (r_idx + 1) mod N
sem_post(r_mutex)
sem_post(empty_count)
```

Esercizio
EchoServer multi-thread con Logger

- Nell'EchoServer viene lanciato un thread Logger che «vive» nella funzione logger() e che si occupa di scrivere messaggi di log su file
- Nei thread che gestiscono le connessioni client, viene usata una funzione log() per «produrre» messaggi di log
- Questi messaggi di log vengono «consumati» dal *thread* Logger
- In ottica produttore/consumatore
  - Ci sono più produttori (thread che gestiscono le connessioni client) ed un singolo consumatore (thread Logger)
  - Il buffer contiene messaggi di log
- Esercizio: completare il codice dell'EchoServer
  - È anche disponibile un client multi-thread che può effettuare un cospicuo numero di richieste parallele così da far generare al server molti messaggi di log in concorrenza

Esercizio
EchoServer multi-process con limite al # di connessioni gestite in parallelo

- Nella pratica, un server non può gestire un numero illimitato di connessioni simultaneamente (e.g., *HTTP 503 Server unavailable*)
- Possibile approccio: semaforo per limite all'accesso concorrente
- Scenario: server multi-process, limite `MAX_CONCURRENCY`
  - Un semaforo anonimo richiederebbe uso di shared memory
  - Un semaforo named può essere acceduto tra processi tra loro «related» usando lo stesso puntatore, poiché è un oggetto IPC speciale allocato su shared memory dal sistema operativo stesso!
- Esercizio: estendere il codice dell'EchoServer multi-process
  - Nei commenti `/* ===> HINT: insert code here <=== */` vi viene suggerito dove estendere il programma, ma non in che modo…
  - *Esercizio proposto*: estendere la versione multi-thread (cosa cambia?)