Esercitazione [03]

Sincronizzazione inter-Processo

Daniele Cono D'Elia – delia@dis.uniroma1.it
Riccardo Lazzeretti – lazzeretti@dis.uniroma1.it
Luca Massarelli – massarelli@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. 2017-2018

Sommario

- Gestione degli errori
- Obiettivi dell’esercitazione
- Sincronizzazione inter-processo

Gestione degli Errori

- Tipicamente, una system call POSIX restituisce -1 per segnalare il fallimento dell'operazione richiesta
  - ➢ conoscerne la causa può essere molto utile!

`int errno` *(variabile globale)*

  - Per poterla ispezionare: `#include <errno.h>`
  - In caso di errore, viene settata con un opportuno codice che indica la causa del fallimento
  - È possibile ottenere una descrizione «testuale» del codice di errore tramite la funzione `strerror()` da `<string.h>`

Gestione degli Errori - Esempio

```
#include <errno.h>
#include <stdio.h>
#include <string.h>
...
int ret = my_call(arg1, ..., argN);
if (ret) {
      printf("Error: %s\n", strerror(errno));
      exit(EXIT_FAILURE);
}
```

Gli errori vanno **sempre** gestiti esplicitamente

- possiamo pensare di meccanizzare le operazioni?

Gestione degli Errori – Macro (1/2)

```
#include <stdio.h>
#include <errno.h>

void perror(const char *str);
```

- Stampa il messaggio di errore relativo a errno. Se str è diverso da NULL e punta ad una stringa allora il messaggio di errore viene preceduta dalla stringa contenuta in str.

Gestione degli Errori – Macro (2/2)

```
================================================
// for most common syscalls
#define handle_error(msg) \
do { \
       perror(msg); \
       exit(EXIT_FAILURE); \
} while (0)
================================================
// for phtread library only
#define handle_error_en(ret,msg) \
do { \
       errno = ret; \
       perror(msg); \
       exit(EXIT_FAILURE); \
} while (0)
================================================
```

Obiettivi Esercitazione [3]

- Sincronizzazione inter-processo
  - Come usare i named semaphores per implementare meccanismi di sincronizzazione tra processi diversi
- Paradigma Produttore-Consumatore tra più processi con letture e scritture da un buffer circolare su file.

Sincronizzazione inter-Processo

- Come sincronizzare processi diversi con i semafori?
  - Usando `sem_init` impostare `pshared ≠ 0`
  - Per essere accessibile da altri processi, il semaforo va allocato in un'area di memoria condivisa
    - Accedendo a quest'area di memoria, un altro processo può recuperare il puntatore al semaforo
- Una soluzione più semplice consiste nell'usare i *named* semaphores...

Named Semaphore

- È identificato univocamente dal suo nome
  - Stringa con terminatore che inizia con uno slash (es. `/semaforo`)
  - Processi diversi accedono allo stesso semaforo tramite il nome
    - shared memory gestita direttamente dall'OS
- Creato con **`sem_open`**`()` ← *e non con* `sem_init()`
- Operazioni di `sem_wait()` e `sem_post()` restano invariate
- Ogni processo, terminato il lavoro, esegue **`sem_close`**`()`
- Quando tutti i processi hanno terminato di lavorare con un semaforo named, almeno uno deve invocare **`sem_unlink`**`()`
  - il semaforo viene così rimosso definitivamente dal sistema…

Named Semaphore in C - `sem_open` (1/4)

- Due possibili signature
  - `sem_t *sem_open(const char *name, int oflag);`
  - `sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);`
- Parametri
  - Nome del semaforo (stringa con terminatore che inizia con /)
  - Flag che controllano la open (definiti in `fcntl.h`)
    - `O_CREAT`: il semaforo viene creato se non esiste già
    - `O_CREAT|O_EXCL`: se il semaforo già esiste viene lanciato un errore
    - In caso di creazione, user e group ID del semaforo sono quelli del processo chiamante
  - In assenza di flag da specificare, il valore da utilizzare è `0`
  - Se `O_CREAT` compare nei flag, vanno specificati altri due parametri……

Named Semaphore in C - `sem_open` (2/4)

- Il parametro `mode` specifica i permessi del semaforo
  - Maschera ottale nella forma `0xyz`
    - `x` specifica i permessi per il proprietario
    - `y` specifica i permessi per il gruppo
    - `z` specifica i permessi per gli altri utenti
  - `x`, `y`, `z` sono costruiti «sommando» i seguenti valori
    - `0`: nessun permesso
    - `1`: permesso di esecuzione
    - `2`: permesso di scrittura
    - `4`: permesso di lettura
  - es: `0640` significa che
    - Il proprietario può leggere e scrivere
    - Gli utenti del gruppo possono solo leggere
    - Tutti gli altri non possono né leggere né scrivere

Named Semaphore in C - `sem_open` (3/4)

- Per il `mode` è anche possibile usare le macro definite in `sys/stat.h`
  - es: `S_IRUSR` è il permesso di lettura per il proprietario
- Il quarto parametro è il valore con cui inizializzare il semaforo stesso
  - Deve essere *non-negativo*
  - Analogo all'omonimo parametro della `sem_init`
- Se viene specificato `O_CREAT` (non in OR | con `O_EXCL`) ed il semaforo già esiste, gli ultimi due parametri vengono ignorati

Named Semaphore in C - `sem_open` (4/4)

- In caso di successo, ritorna un puntatore al named semaphore appena aperto
- In caso di fallimento, ritorna `SEM_FAILED`
- Se il fallimento è dovuto al fatto che già esiste un semaforo con quel nome, `errno` viene impostato a `EEXIST`

Named Semaphore in C

`sem_close` e `sem_unlink`

- `sem_close` chiude il semaforo per il processo corrente, liberando le risorse allocate per esso
  - argomento: puntatore `sem_t*` ottenuto da `sem_open`
- `sem_unlink` serve a distruggere il semaforo nell'OS
  - prende come argomento la stringa che identifica univocamente il named semaphore (come se fosse un file)
  - il semaforo verrà quindi distrutto non appena ciascun processo che lo ha aperto in precedenza lo avrà chiuso
  - ➢ Nota: nella pratica spesso è un singolo processo ad eseguire unlink dopo che tutti hanno fatto close

Named Semaphore - Esercizio

- Scheduler con sincronizzazione inter-processo secondo il paradigma client-server
  - Il server crea il semaforo per consentire l'accesso in sezione critica al massimo a `NUM_RESOURCES` thread per volta
  - Il client lancia `THREAD_BURST` thread per volta che si sincronizzano tramite il semaforo creato dal server
- Sorgenti: `server.c client.c`
  - *Completare le parti contrassegnate con TODO*
- Compilazione tramite `Makefile`
  - Il server richiede `util.c` e `util.h`
  - Client e server vanno entrambi linkati alla libreria `pthread`
- Esecuzione: lanciare prima il server, poi in un altro terminale avviare una istanza del client
- In util.h sono definite le macro per la gestione degli errori

Funzione sem_getvalue()

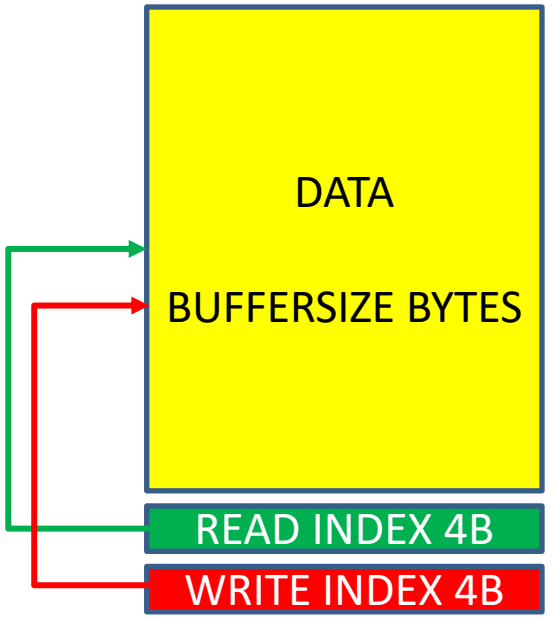
- Consente di leggere il valore corrente di un semaforo
- `int sem_getvalue(sem_t *sem, int *sval);`
  - Parametri di input
    - Puntatore al semaforo
    - Puntatore ad un int che verrà settato al valore del semaforo
  - Ritorna 0 in caso di successo, -1 in caso di errore
- Se la coda di attesa del semaforo non è vuota, `*sval` su sistemi Linux sarà settato a `0`
  - su altri sistemi operativi POSIX-compliant può invece assumere valore negativo (pari al numero di thread in coda)

Secondo Esercizio

- Paradigma Produttore-Consumatore tra più processi con letture e scritture da un buffer circolare su file.
- Questa volta usiamo un buffer su un file che viene scritto dai produttori e letto dai consumatori.

Buffer circolare su file

- Il file che viene usato come buffer circolare è un file binario
- Gli ultimi 4 byte rappresentano l'indice di scrittura
- I penultimi 4 quello di lettura
- Il file viene creato quando si lancia il produttore che inizializza i valori del buffer e gli indici



Lettura di un int da buffer su file (1)

- Il cursore del file può essere spostato attraverso la funzione:
  - int fseek (FILE * stream, long int offset, int origin);

- Per effettuare una lettura ci spostiamo prima alla fine del file e leggiamo l'indice di lettura
- Poi spostiamo il cursore a quell'indice e leggiamo il valore che ci interessa

Lettura di un int da buffer su file (2)

- Una volta fatta la lettura, dobbiamo spostare il cursore verso la fine del file per incrementare l'indice di lettura di sizeof(int) bytes
- N.B. stiamo usando un buffer circolare quindi nell'incrementare dobbiamo sempre effettuare l'operazione modulo buffer_size
- La stessa cosa vale per la scrittura!

Mutua esclusione in lettura e scrittura sul buffer

- Ogni volta che viene effettuata una lettura o una scrittura bisogna aggiornare gli indici...
  - Il file è una risorsa critica che può dar luogo a race conditions, va acceduta in mutua esclusione!
- Abbiamo bisogno di un semaforo named *sem_cs* a cui si può accedere da più processi per regolare l'accesso in sezione critica
- Il semaforo va inizializzato ad 1

Producer / Consumer

- Ci servono altri due semafori:
  - **sem_filled**: controlla se il buffer contiene almeno un elemento da leggere. Se non ci sono elementi da leggere il consumer deve attendere. Il semaforo va inizializzato a 0. Il consumer fa wait su questo semaforo, ed il producer post.
  - **sem_empty**: controlla se il buffer è pieno. Se è pieno il producer deve attendere che si svuoti. Va inizializzato alla dimensione del buffer. Il consumer quindi fa post, ed il producer wait.

In pratica… (1)

- *common.c* e *common.h* contengono le funzioni già implementate che leggono e scrivono sul buffer
- *handle_error*, macro già definita in *common.h* usarla per la gestione degli errori
- Bisogna gestire l'accesso alle risorse in mutua esclusione attraverso semafori named in *producer.c* e *consumer.c*

In pratica… (2)

- **Compilazione:** usare il makefile incluso
- **Esecuzione:** lanciare il producer e poi a parte il consumer. La somma finale dei valori stampati da producer e da consumer deve coincidere
- Il file usato come buffer può essere analizzato con un editor esadecimale (ad es. xxd o hexdump -C da shell)