Esercitazione [05]

Shared memory

Riccardo Lazzeretti - lazzeretti@diag.uniroma1.it

Sistemi di Calcolo 2
Programmazione dei Sistemi di Calcolo Multi-Nodo

Corso di Laurea in Ingegneria Informatica e Automatica

A.A. 2020-2021

Sommario

- Soluzione esercitazione precedente
- Obiettivi esercitazione
- Shared memory
- Esercizio: riepilogo con shared memory
- Esercizio: applicazione multiprocess con memoria condivisa
- Esercizio: produttore/consumatore (M/N) con memoria condivisa
- Esercizio: produttore/consumatore (1/1) con memoria condivisa e senza semafori

Obiettivi Esercitazione [5]

- Imparare a usare la memoria condivisa POSIX
- Interprocess communication

POSIX Shared Memory

- Offre funzioni per allocare e deallocare una memoria condivisa
- La memoria condivisa è mappata su un puntatore
- Lettura e scrittura vengono effettuate tramite normali operazioni che coinvolgono il puntatore

POSIX Shared Memory - requisiti

```
#include <sys/mman.h>
#include <sys/stat.h>   /* For mode constants */
#include <sys/types.h>
#include <fcntl.h>      /* For O_* constants */
#include <unistd.h>
```

Link with *-lrt*.

Funzione **shm_open()**

```
int shm_open(const char *name, int oflag, mode_t mode);
```

- Crea e apre una shared memory, oppure apre una shared memory esistente
- Argomenti
  - `name`: specifica l'oggetto di memoria condivisa da creare o aprire. Per un uso portatile, un oggetto di memoria condivisa deve essere identificato da un nome del tipo `"/nome";` vale a dire una stringa
  - `oflag`: parametri
    - `O_CREAT` crea l'oggetto di memoria condivisa se non esiste. Il nuovo oggetto di memoria condivisa inizialmente ha una lunghezza pari a zero
    - `O_EXCL`: se è stato specificato anche `O_CREAT` e esiste già un oggetto di memoria condivisa con il nome specificato, restituisce un errore.
    - `O_RDONLY` apre l'oggetto per l'accesso in lettura.
    - `O_RDWR` apre l'oggetto per l'accesso in lettura / scrittura.
    - Per I nostri scopi:
      - Creazione: `O_CREAT` | `O_EXCL` |`O_RDWR`
      - Apertura: `O_RDWR` oppure `O_RDONLY` (secondo necessità)
  - `mode`: permessi utenti. Per i nostri scopi, `0666` oppure `0660`
- Valore di ritorno
  - In caso di successo, il descrittore della shared memory
  - In caso di errore, `-1`, `errno` è settato

Funzione `ftruncate()`

```
int ftruncate(int fd, off_t length);
```

- dimensiona la memoria condivisa a cui fa riferimento fd a una dimensione di `length`.
- Se la memoria condivisa in precedenza era più grande di `length`, i dati extra andrebbero persi. Se la memoria in precedenza era più corta, viene estesa e la parte estesa viene letta come byte null ('\ 0').
- Argomenti
  - `fd`: descrittore della memoria condivisa ottenuto da `shm_open()`
  - `length`: dimensione della memoria condivisa
- Valore di ritorno
  - In caso di successo, 0
  - In caso di errore, `-1`, `errno` è settato

- Curiosità: può essere applicato su un descrittore di file e la dimensione del file cambia di conseguenza

Funzione **`mmap()`**

```
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

- Mappa la shared memory nella memoria riservata al processo
- Argomenti
  - `addr`: permette di suggerire al kernel dove posizionare la memoria condivisa. Se `NULL` o `0` (**nostra scelta: 0**) il kernel decide autonomamente
  - `length`: dimensione della memoria condivisa
  - `fd`: descrittore della memoria condivisa ottenuto da `shm_open()`
  - `offset`: permette di mappare la memoria condivisa da una posizione diversa da quella iniziale. `offset` deve essere un multiplo della page size. **Per noi 0**

Funzione **mmap()**

```
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

- Argomenti (continua)
  - `prot`: specifica le protezioni sulla modalità di accesso per il processo chiamante. Non deve essere in conflitto con i parametri settati in shm_open()
    - `PROT_READ` permesso di lettura
    - `PROT_WRITE` permesso di scrittura
    - `PROT_EXEC` permesso di esecuzione
    - `PROT_NONE` nessun permesso
    - Per i nostri scopi `PROT_READ , PROT_READ | PROT_WRITE , PROT_WRITE`
  - `flags`:
    - `MAP_SHARED` rende le modifiche effettuate nella memoria condivisa visibili agli altri processi
    - Altri flag esistono, ma non sono di nostro interesse

- Valore di ritorno
  - In caso di successo, il puntatore all'area di memoria dove risiede la shared memory
  - In caso di errore `MAP_FAILED`, `errno` è settato

- Curiosità: può essere applicato su un descrittore di file. Il file è mappato in memoria ed è possibile accedere al suo contenuto tramite il puntatore all'area della memoria, invece che tramite le normali operazioni su file. Utile quando il file contiene dati strutturati.

Funzione **`munmap()`**

```
int munmap(void *addr, size_t length);
```

- Cancella il mapping tra il processo e la memoria condivisa.
- Successivi tentativi di accesso tramite il puntatore falliranno
- Argomenti
  - `addr`: il puntatore alla memoria condivisa ottenuto da `mmap`
  - `length`: dimensione della memoria condivisa
- Valore di ritorno
  - In caso di successo, `0`
  - In caso di errore, `-1`, `errno` è settato

Funzione **`shm_unlink()`**

```
int shm_unlink(const char *name);
```

- Rimuove una memoria condivisa
- Una volta che tutti i processi hanno fatto l'unmap della memoria, disalloca e distrugge il contenuto della regione di memoria associata.
- Successivi tentativi di aprire la memoria falliranno (a meno che non venga usata l'opzione `O_CREAT`)
- Argomenti
  - `name`: identificatore della memoria condivisa, stesso usato in `shm_open`
- Valore di ritorno
  - In caso di successo, `0`
  - In caso di errore, `-1`, `errno` è settato

Funzione `close()`

```
int close(int fd);
```

- Chiude il descrittore della memoria condivisa
- Dopo aver effettuato `mmap`, il descrittore può essere chiuso in ogni momento senza influenzare il mapping della memoria
- Argomenti
  - `fd`: descrittore della memoria condivisa, ottenuto da `shm_open`
- Valore di ritorno
  - In caso di successo, `0`
  - In caso di errore, `-1`, `errno` è settato

ESEMPIO: SCRIVERE E LEGGERE IN MEMORIA CONDIVISA

```
/* Program to write some data in shared memory */
int main() {
        const int SIZE = 4096; /* size of the shared page */
                /* name of the shared page */
        const char * name = "MY_PAGE";
        const char * msg = "Hello World!";
        int shm_fd;
        char * ptr;

        shm_fd = shm_open(name, O_CREAT | O_RDRW, 0666);
        ftruncate(shm_fd, SIZE);
        ptr = (char *) mmap(0, SIZE, PROT_WRITE,
                MAP_SHARED, shm_fd, 0);
        sprintf(ptr, "%s", msg);
        close(shm_fd);
        return 0;
}
```

```
/* Program to read some data from shared memory */
int main() {
        const int SIZE = 4096; /* size of the shared page */
                /* name of the shared page */
        const char * name = "MY_PAGE";
        int shm_fd;
        char * ptr;

        shm_fd = shm_open(name, O_RDONLY, 0666);
        ptr = (char *) mmap(0, SIZE, PROT_READ,
                MAP_SHARED, shm_fd, 0);
        printf("%s\n", ptr);
        shm_unlink(name);
        return 0;
}
```

Esercizio (0): esercizio di riepilogo

- Nell'esercizio di riepilogo facevamo uso di un semaforo che segnalava ai processi figli di terminare l'attività
  - Il server esegue una signal
  - Il client legge il contatore nel semaforo
    - Se positivo termina l'attività
    - Altrimenti avvia un altro burst di thread
- Completare il codice dell'applicazione sostituendo il semaforo con una shared_memory
- Sorgenti
  - `makefile`
  - `riepilogo.c`
- Suggerimento: seguire i blocchi di commenti inseriti nel codice

Esercizio (1): Applicazione modulare

- L'applicazione è sviluppata in due componenti.
  - Il primo (requester) carica dati nella memoria condivisa
  - Il secondo (worker) li elabora
  - Il primo li stampa
- L'applicazione è composta da due processi generati tramite fork
- Completare il codice dell'applicazione request/worker
- Sorgenti
  - `makefile`
  - `req_wrk.c`
- Suggerimento: seguire i blocchi di commenti inseriti nel codice
- Suggerimento: inserire elementi per la sincronizzazione
- Test:
  - Lanciate l'applicazione, deve stampare alla fine i valori elaborati (il quadrato dei numeri interi da `0` a `NUM-1`)

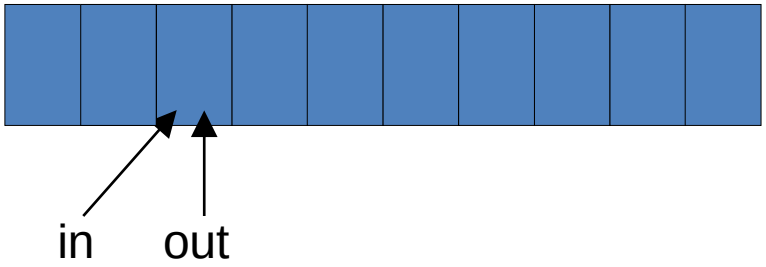
Esercizio (2): Produttore/Consumatore

- L'applicazione è sviluppata in due moduli separati.
- Si tiene conto della configurazione con `NUM_CONSUMERS` consumatori e `NUM_PRODUCERS` produttori
- Il buffer e le posizioni di in e out sono posizionati in memoria condivisa

- Completare il codice dell'applicazione produttore/consumatore
- Sorgenti
  - `makefile`
  - `producer.c`
  - `consumer.c`
- Suggerimento: seguire i blocchi di commenti inseriti nel codice
- Informazione: gli elementi per la sincronizzazione (vedi esercitazione 3 in lab) sono già inseriti
- Test:
  - Lanciate prima `producer` (crea semafori e memoria condivisa) e poi `consumer`

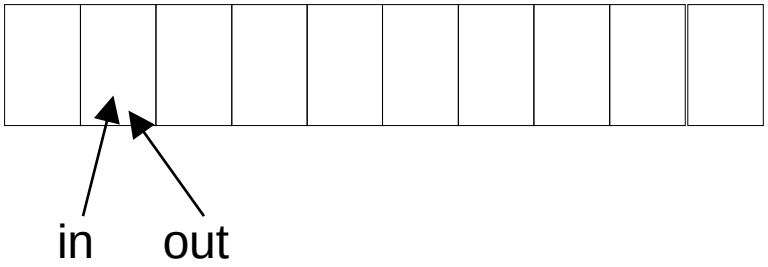
Buffer State in Shared Memory

Buffer Full



in == out; sem_empty.val == 0; sem_filled.val == BUFFER_SIZE

Buffer Empty



in == out; sem_empty.val == BUFFER_SIZE; sem_filled.val == 0

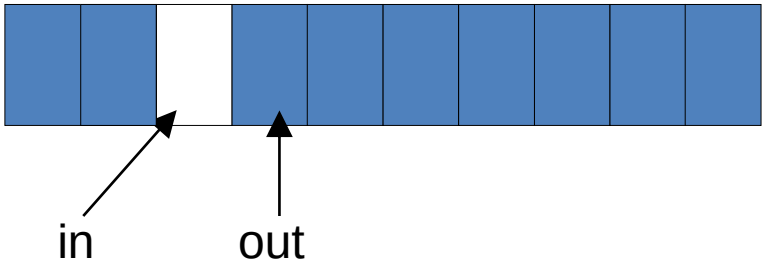
Esercizio (3): Produttore/Consumatore senza semafori

- L'applicazione è sviluppata in due moduli separati.
- Si tiene conto della configurazione con `1` consumatore e `1` produttore
- Il buffer e le posizioni di in e out sono posizionati in memoria condivisa

- Completare il codice dell'applicazione produttore/consumatore
- Sorgenti
  - `makefile`
  - `producer.c`
  - `consumer.c`
- Suggerimento: seguire i blocchi di commenti inseriti nel codice
- Informazione: presenta il vantaggio di non ricorrere a chiamate al kernel per la sincronizzazione, ma sacrifica una posizione del buffer e introduce busy waiting
- Test:
  - Lanciate prima `producer` (memoria condivisa) e poi `consumer`

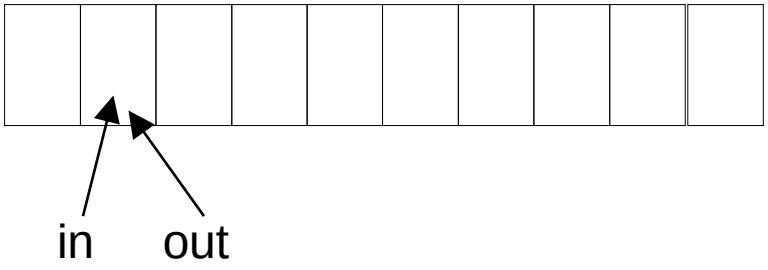
Buffer State in Shared Memory

Buffer Full



(in +1) % BUFFER_SIZE == out

Buffer Empty



in == out