

Sistemi Operativi e Reti di Calcolatori (SORECa)

Corso di Laurea in *Ingegneria Informatica e Automatica (BIAR)*
Terzo Anno | Primo Semestre
A.A. 2024/2025

Esercitazione [03] Sincronizzazione Inter-Processo

Riccardo Lazzeretti lazzeretti@diag.uniroma1.it

Paolo Ottolino

Alessio Izzillo izzillo@diag.uniroma1.it

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI



SAPIENZA
UNIVERSITÀ DI ROMA

Sommario

- Soluzione esercitazione precedente (Lab01-es3)
- Gestione degli errori (`errno.h`)
- Produttore – Consumatore (Bounded-Buffer)
- Sincronizzazione inter-processo (Named-Semaphore + Makefile)
- Esercizio per casa (Bounded-Buffer + Named-Semaphore)

Accesso in mutua esclusione a N risorse

→ Lab01, Esercizio 3

Concorrenza in C

Lab01-es3: Accesso in mutua esclusione a N risorse - presentazione

- Disponibilità di un numero N di risorse, ognuna delle quali può essere usata in mutua esclusione
 - Pool di connessioni a DB
 - Pool di thread
 - etc...
- M thread in concorrenza devono accedere a queste risorse ($M > N$)
- Come implementarlo con i semafori?
 - Suggerimento: mentre prima solo un thread alla volta poteva accedere alla sezione critica, ora vogliamo che ciò sia possibile per N thread alla volta

Concorrenza in C

Lab01-es3: Accesso in mutua esclusione a N risorse - implementazione

- Soluzione: inizializzare il semaforo a N invece che a 1
- Codice: `scheduler.c`
- Compilazione
`gcc -o scheduler scheduler.c -lpthread`
- Come si usa
 - Lanciare `./scheduler`
 - Premendo INVIO, vengono lanciati `THREAD_BURST` thread che tentano di accedere in parallelo a `NUM_RESOURCES` risorse e le usano per processare ciascuno `NUM_TASKS` work item
 - Un work item richiede un tempo random compreso tra 0 e `MAX_SLEEP` secondi
 - Premendo CTRL+D, il programma termina
 - Osservare l'interleaving dei vari thread e il fatto che non ci sono mai nello stesso momento più di `NUM_RESOURCES` thread che hanno accesso ad una delle risorse

Concorrenza in C

Lab01-es3: Accesso in mutua esclusione a N risorse – perché funziona

- Inizializzando il semaforo a N, i primi N thread che eseguiranno la `sem_wait()` vedranno un valore non negativo dopo il proprio decremento e potranno accedere alla sezione critica
- I thread successivi effettueranno un decremento (valore del semaforo negativo) e verranno messi in attesa in coda
- Quando uno dei primi N thread esegue la `sem_post()`, il semaforo viene incrementato; siccome il valore era negativo, esso al massimo può diventare 0, e quindi uno dei thread in coda viene svegliato
 - Gli altri thread in coda rimangono lì in attesa
- Ad ogni successiva `sem_post()`, il semaforo viene incrementato
 - Finchè il semaforo non diventa positivo, vuol dire che ci sono thread in coda che verranno svegliati ad ogni `sem_post()`

Concorrenza in C

Lab01-es3: Accesso in mutua esclusione a N risorse – variante1

- Modificare il codice per implementare la seguente semantica
 - Invece di usare la risorsa per processare ininterrottamente tutti i work item, ogni thread deve rilasciare la risorsa dopo aver completato una coppia di work item, e rimettersi quindi in coda per ottenere nuovamente l'accesso ad una risorsa
 - Una volta acquisita una risorsa, il thread deve completare la coppia successiva di work item e così via
 - Rilasciare definitivamente ogni risorsa dopo che tutti i work item sono stati processati

Concorrenza in C

Lab01-es3: Accesso in mutua esclusione a N risorse – variante2

- Provate a aumentare sensibilmente il tempo nella sleep
- Cosa può succedere?
 - Il main dopo aver creato i thread esce (avete premuto CTRL+D prima che i thread avessero finito il lavoro), distrugge il semaforo e libera la sua memoria
 - Un thread ancora in esecuzione potrebbe non trovare più il semaforo

Concorrenza in C

Lab01-es3: Accesso in mutua esclusione a N risorse – variante2

- Provate a aumentare sensibilmente il tempo nella sleep
- Cosa può succedere?
 - Il main dopo aver creato i thread esce (avete premuto CTRL+D prima che i thread avessero finito il lavoro), distrugge il semaforo e libera la sua memoria
 - Un thread ancora in esecuzione potrebbe non trovare più il semaforo

`sem_destroy()` destroys the unnamed semaphore at the address pointed to by `sem`.

Only a semaphore that has been initialized by `sem_init(3)` should be destroyed using `sem_destroy()`.

Destroying a semaphore that other processes or threads are currently blocked on (in `sem_wait(3)`) produces undefined behavior.

Using a semaphore that has been destroyed produces undefined results, until the semaphore has been reinitialized using `sem_init(3)`.

Concorrenza in C

Lab01-es3: Accesso in mutua esclusione a N risorse – variante2

- Come risolvere il problema?
 - Uso di una variabile che conta quanti thread sono attualmente aperti
 - Il main la incrementa quando crea un thread
 - Il thread la decrementa prima di terminare
 - Solo quando la variabile è 0 si può chiudere e distruggere il semaforo
 - Attenzione: l'accesso alla variabile potrebbe causare problemi di concorrenza
 - Bisogna usare un semaforo per garantire la mutua esclusione
 - Possiamo usare lo stesso semaforo o è meglio usarne uno nuovo?

Concorrenza in C

Lab01-es3: Accesso in mutua esclusione a N risorse – variante2 ➔ Protocol Idea

```
int main(int argc, char* argv[]) {  
    ...  
    cont = 0;  
    sem_init(&sem2, 0, 1);  
    for (int i = 0; i < THREAD_BURST; ++i) {  
        ...  
        pthread_create(&thread_handle, NULL, client, args);  
        sem_wait(&sem2);  
        ++cont;  
  
        sem_post(&sem2);  
    }  
    pthread_detach(thread_handle);  
    ...  
    while (cont>0);  
    ...  
    sem_destroy(semaphore);  
    sem_destroy(&sem2);  
    free(semaphore);  
    free(&sem2);  
    pthread_exit(NULL);  
}
```

```
void* client(void* arg_ptr) {  
    ...  
    sem_wait(&sem2);  
    --cont;  
  
    sem_post(&sem2);  
    return NULL;  
}
```

Concorrenza in C

Lab01-es3: Accesso in mutua esclusione a N risorse – variante2

- La soluzione comporta busy waiting del thread main
- Possiamo risolvere il problema senza avere problemi di mutua esclusione su una variabile contatore?

Concorrenza in C

Lab01-es3: Accesso in mutua esclusione a N risorse – variante2 → Protocol Idea

```
int main(int argc, char* argv[]) {  
    ...  
    thread_ID = 0;  
    sem_init(&sem2, 0, 0);  
    for (int i = 0; i < THREAD_BURST; ++i) {  
        ...  
        pthread_create(&thread_handle, NULL, client, args);  
        ++thread_ID;  
    }  
    pthread_detach(thread_handle);  
    ...  
    for (int i=0; i < thread_ID; ++i)  
        sem_wait(&sem2);  
    ...  
    sem_destroy(semaphore);  
    sem_destroy(&sem2);  
    free(semaphore);  
    free(&sem2);  
    pthread_exit(NULL);  
}
```

Diagram illustrating the flow of execution between the main function and the client function:

- An orange arrow points from the `sem_wait(&sem2);` line in the main function to the `sem_post(&sem2);` line in the client function, indicating that the client releases the semaphore after its execution.

```
void* client(void* arg_ptr) {  
    ...  
    sem_post(&sem2);  
    return NULL;  
}
```

Gestione degli Errori

`errno.h` + `string.h`

- Tipicamente, una system call POSIX restituisce -1 per segnalare il fallimento dell'operazione richiesta
 - conoscerne la causa può essere molto utile!
 - Alcune funzioni settano il parametro `errno`

`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

errno.h + string.h - Esempio

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

Gli errori vanno **sempre gestiti esplicitamente**

➤ possiamo pensare di **meccanizzare le operazioni?**

Gestione degli Errori

`perror(const char * str)` – Macro 1/2

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

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

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

- Stampa la stringa `str`, seguita dal messaggio di errore relativo ad `errno`.
- Se `str` è `NULL` stampa solo messaggio di errore relativo ad `errno`.
- L'output è stampato sul default error stream (la console se non sostituito)

Gestione degli Errori

perror(const char * str) – Macro 2/2

```
=====
// for most common syscalls
#define handle_error(msg) \
do { \
    perror(msg); \
    exit(EXIT_FAILURE); \
} while (0)

// for pthread library only and other functions that do not set errno
#define handle_error_en(ret,msg) \
do { \
    errno = ret; \
    perror(msg); \
    exit(EXIT_FAILURE); \
} while (0)
=====
```

Macro definite in common.h

Produttore Consumatore

→ Lab03, Esercizio 1

Produttore-Consumatore

Lab03-es1: Produttore Consumatore - presentazione

- Prendiamo il codice `producer_consumer.c` in cui non viene garantita la mutua esclusione
- Ruolo delle variabili globali:
 - R: numero di risorse (lunghezza dell'array)
 - N: numero di produttori
 - M: numero di consumatori
 - O: numero di operazioni per produttori e consumatori
- Compilazione:
`gcc -o producer_consumer producer_consumer.c -lpthread`

Produttore-Consumatore

Lab03-es1: Produttore Consumatore - richieste

- Risolvere il problema nei seguenti quattro scenari, come illustrato nella lezione teorica
 - $1 : 1$ (1 produttore, 1 consumatore)
 - $1 : M$ (1 produttore, M consumatori)
 - $N : 1$ (N produttori, 1 consumatore)
 - $N : M$ (N produttori, M consumatori)
- Si consiglia di creare 4 soluzioni distinte

Produttore-Consumatore

Lab03-es1: Produttore Consumatore – Bounded Buffer

```
/* program bounded buffer */  
const int sizeofbuffer * /* buffer size */;  
semaphore s1 = 1, s2 = 1, n = 0, e = sizeofbuffer;
```

```
void producer ()  
{  
    while (true) {  
        produce();  
        semWait(e);  
        semWait(s1);  
        append();  
        semSignal(s1);  
        semSignal(n);  
    }  
}
```

```
void consumer ()  
{  
    while (true) {  
        semWait(n);  
        semWait(s2);  
        take();  
        semSignal(s2);  
        semSignal(e);  
        consume();  
    }  
}
```

What if I swap pairs of adjacent semWait or semSignal operations???

What if I have only a producer or only a reader???

Sincronizzazione Inter-Process

→ Lab03, Esercizio 2

Sincronizzazione Inter-Processo

Lab03-es2: Named Semaphore – bounded-buffer su file

- 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

Lab03-es2: Semafori – processi diversi

- 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*...

Sincronizzazione Inter-Processo

Lab03-es2: Named Semaphore – processi diversi

- 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()`**
- Un processo deve distruggere il semaforo con **`sem_unlink()`**
 - Il processo che esegue il comando non ha più accesso al semaforo
 - Quando tutti i processi hanno terminato di lavorare con un semaforo named, il semaforo viene così rimosso definitivamente dal sistema...

Sincronizzazione Inter-Processo

Lab03-es2: Named Semaphore – `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

- `*name`: Nome del semaforo (stringa con terminatore che inizia con `/`)
- `oflag`: 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.....

Sincronizzazione Inter-Processo

Lab03-es2: Named Semaphore – `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 (ma non eseguire)
 - Gli utenti del gruppo possono solo leggere
 - Tutti gli altri non possono né leggere né scrivere

Sincronizzazione Inter-Processo

Lab03-es2: Named Semaphore – `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

Sincronizzazione Inter-Processo

Lab03-es2: Named Semaphore – `sem_open()` 4/4

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

Sincronizzazione Inter-Processo

Lab03-es2: Named Semaphore – `sem_close()` `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`
 - Se un thread chiude un semaforo, il comportamento di altri thread bloccati sul semaforo non è definibile
- `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

Sincronizzazione Inter-Processo

Lab03-es2: 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

Sincronizzazione Inter-Processo

Lab03-es2: Named Semaphore – `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)

Produttori-Consumatori su File

→ Lab03, Esercizio 1

Esercizio per Casa

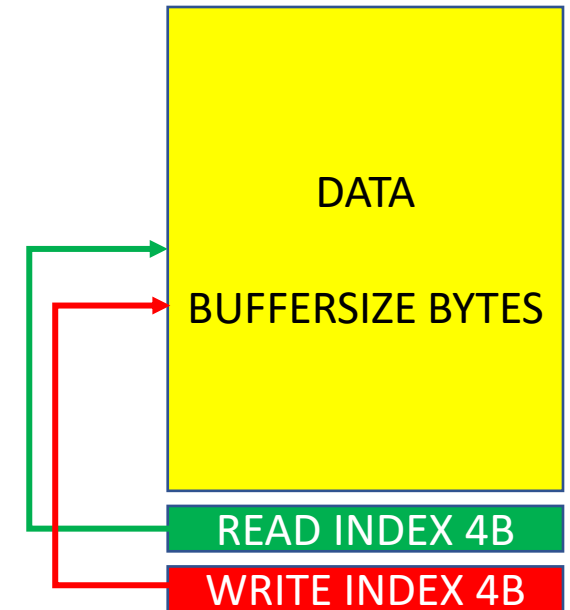
Lab03-es3 Produttore-Consumatore: Buffer Circolare su File - presentazione

- 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.

Esercizio per Casa

Lab03-es3 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



Esercizio per Casa

Lab03-es3 Lettura di un int da buffer su file 1/2

- 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 (SEEK_END) e leggiamo l'indice di lettura
- Poi spostiamo il cursore a quell'indice e leggiamo il valore che ci interessa

Costante	Significato
SEEK_SET	lo spostamento avviene a partire dall'inizio del file.
SEEK_CUR	lo spostamento avviene a partire dall'attuale posizione.
SEEK_END	lo spostamento avviene a partire dalla fine del file.

`int fseek (FILE *fp, long s, int o);`

Esercizio per Casa

Lab03-es3 Lettura di un int da buffer su file 2/2

- 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!

Esercizio per Casa

Lab03-es3 lettura e scrittura sul buffer → Mutua Esclusione

- 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

Esercizio per Casa

Lab03-es3 Produttore/Consumatore → altri 2 semafori

- 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.

Esercizio per Casa

Lab03-es3 in pratica... 1/2

- *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*

Esercizio per Casa

Lab03-es3 in pratica... 2/2

- **Compilazione:** usare il makefile incluso

```
$ make all
```

- **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:

```
$ xxd
```

```
o
```

```
$ hexdump -C (equivalente a hd)
```