Sistemi Operativi e Reti di Calcolatori (SOReCa)

Corso di Laurea in *Ingegneria Informatica e Automatica (BIAR)*

Terzo Anno | Primo Semestre

A.A. 2025/2026

Esercitazione [03] Shared Memory

Riccardo Lazzeretti lazzeretti@diag.uniroma1.it
Paolo Ottolino paolo.ottolino@uniroma1.it
Matteo Cornacchia cornacchia@diag.uniroma1.it

DIPARTIMENTO DI INGEGNERIA INFORMATICA AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA UNIVERSITÀ DI ROMA

Sommario

- Soluzione Esercizi precedent (lab2)
- Obiettivi esercitazione
- Shared memory
- Esercizio 1: applicazione multiprocess con memoria condivisa
- Esercizio 2: produttore/consumatore (M/N) con memoria condivisa
- Esercizio3 : produttore/consumatore (1/1) con memoria condivisa e senza semafori

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Sol: Produttore Consumatore

Lab02, Esercizio 1

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Produttore-Consumatore

Lab02-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

```
/* per tutti*/
#include "common.h" // gestione errori
```

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Produttore-Consumatore

Lab02-es1: Produttore Consumatore – 1:1 uso dei semafori

```
/* program bounded buffer 1:1 */
sem_t empty_sem, fill_sem;
```

```
// producer thread
void* performTransactions(void* x) {
    thread_args_t* args = (thread_args_t*)x;
    printf("Starting producer thread %d\n", args->threadId);

    while (args->numOps > 0) {
        // produce the item
        int currentTransaction = performRandomTransaction();

        // make sure there is space in the buffer
        if (sem_wait(&empty_sem)) {
            handle_error("Producer: sem_wait error empty sem");
        }

        // write the item and update write_index accordingly
        transactions[write_index] = currentTransaction;
        write_index = (write_index + 1) % BUFFER_SIZE;

        // notify that a new element just became available
        while (1){
        if (sem_post(&fill_sem)) {
            handle_error("Producer: sem_post error fill sem");
        }
        }

        args->numOps--;
    }

    free(args);
    pthread_exit(NULL);
}
```

```
// consumer thread
void* processTransactions(void* x) {
    thread_args_t* args = (thread_args_t*)x;
    printf("Starting consumer thread %d\n", args->threadId);

    while (args->numOps > 0) {
        // make sure there is data to consume
        if (sem_wait(&fill_sem)) {
            handle_error("Consumer: sem_wait error fill sem");
        }

        // consume the item and update (shared) variable deposit
        deposit += transactions[read_index];
        read_index = (read_index + 1) % BUFFER_SIZE;
        if (read_index % 100 == 0)
            printf("After the last 100 transactions balance is now %d.\n",
deposit);

        // notify that a free cell in the buffer just became available
        if (sem_post(&empty_sem)) {
            handle_error("Consumer: sem_post error empty sem");
        }

        args->numOps--;
        //printf("C %d\n", args->numOps);
    }

    free(args);
    pthread_exit(NULL);
}
```

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Produttore-Consumatore

Lab02-es1: Produttore Consumatore – 1:1 creazione/distruzione semafori

```
/* creazione dei semafori*/

sem_t empty_sem, fill_sem;
// bla bla bla

int main(int argc, char* argv[]) {

// bla bla bla

// initialize read and write indexes
    read_index  = 0;
    write_index = 0;

    // initialize semaphores
    int ret;
    ret = sem_init(&fill_sem, 0, 0);
    if (ret) handle_error("init error fill sem");
    if (sem_init(&empty_sem, 0, BUFFER_SIZE))  handle_error("init error
empty sem");

// pthread_create(), pthread_join()
// bla bla bla

    exit(EXIT_SUCCESS);
}
```

```
/* distruzione dei semafori*/

sem_t empty_sem, fill_sem;
// bla bla bla

int main(int argc, char* argv[]) {

// pthread_create(), pthread_join()

printf("Final value for deposit: %d\n", deposit);

    // destroy semaphores
    if (sem_destroy(&fill_sem)) handle_error("Fill sem destroy error");
    if (sem_destroy(&empty_sem)) handle_error("Empty sem destroy
error");

    exit(EXIT_SUCCESS);
}
```

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Produttore-Consumatore

Lab02-es1: Produttore Consumatore – 1:M uso dei semafori

```
/* program bounded buffer 1:M*/
sem_t empty_sem, fill_sem;
sem_t read_sem;
```

```
// producer thread - uguale a 1:1
void* performTransactions(void* x) {
    thread_args_t* args = (thread_args_t*)x;
    printf("Starting producer thread %d\n", args->threadId);

    while (args->numOps > 0) {
        // produce the item
        int currentTransaction = performRandomTransaction();

        // make sure there is space in the buffer
        if (sem_wait(&empty_sem)) {
            handle_error("Producer: sem_wait error empty sem");
        }

        // write the item and update write_index accordingly
        transactions[write_index] = currentTransaction;
        write_index = (write_index + 1) % BUFFER_SIZE;

        // notify that a new element just became available
        while (1){
        if (sem_post(&fill_sem)) {
            handle_error("Producer: sem_post error fill sem");
        }
        }

        args->numOps--;
    }

    free(args);
    pthread_exit(NULL);
}
```

```
// consumer thread
void* processTransactions(void* x) {
    thread_args_t* args = (thread_args_t*)x;
    printf("Starting consumer thread %d\n", args->threadId);

    while (args->numOps > 0) {
        // make sure there is data to consume
        if (sem_wait(&fill_sem)) {
            handle_error("Consumer: sem_wait error fill sem");
        }

        // get exclusive read access <CRITICAL SECTION>
        if (sem_wait(&read_sem)) {
            handle_error("Consumer: sem_wait error read sem");
        }

        // consume the item and update (shared) variable deposit
        deposit += transactions[read_index];
        read_index = (read_index + 1) % BUFFER_SIZE;
        if (read_index % 100 == 0)
                                                printf("After the last 100 transactions
balance is now %d.\n", deposit);

        if (sem_post(&read_sem)) {
            handle_error("Consumer: sem_post error read sem");
        } // </CRITICAL SECTION>

        // notify that a free cell in the buffer just became available
        if (sem_post(&empty_sem)) {
            handle_error("Consumer: sem_post error empty sem");
        }

        args->numOps--;
        //printf("C %d\n", args->numOps);
    }

    free(args);
    pthread_exit(NULL);
}
```

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Produttore-Consumatore

Lab02-es1: Produttore Consumatore – 1:M creazione/distruzione semafori

```
/* creazione dei semafori*/

sem_t empty_sem, fill_sem;
sem_t read_sem;
// bla bla bla

int main(int argc, char* argv[]) {

// bla bla bla

// initialize read and write indexes
    read_index  = 0;
    write_index = 0;

    // initialize semaphores
    if (sem_init(&fill_sem, 0, 0)) handle_error("init error fill sem");
    if (sem_init(&empty_sem, 0, BUFFER_SIZE))  handle_error("init error
empty sem");
    if (sem_init(&read_sem, 0, 1))  handle_error("init error read sem");
// pthread_create(), pthread_join()
// bla bla bla

    exit(EXIT_SUCCESS);
}
```

```
/* distruzione dei semafori*/

sem_t empty_sem, fill_sem;
sem_t read_sem;
// bla bla bla

int main(int argc, char* argv[]) {

// pthread_create(), pthread_join()

printf("Final value for deposit: %d\n", deposit);

    // destroy semaphores
    if (sem_destroy(&fill_sem)) handle_error("Fill sem destroy error");
    if (sem_destroy(&empty_sem)) handle_error("Empty sem destroy
error");
    if (sem_destroy(&read_sem)) handle_error("read sem destroy error");
    exit(EXIT_SUCCESS);
}
```

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Produttore-Consumatore

Lab02-es1: Produttore Consumatore – N:1 uso dei semafori

```
/* program bounded buffer 1:M*/
sem_t empty_sem, fill_sem;
sem_t write_sem;
```

```
// producer thread
void* performTransactions(void* x) {
    thread_args_t* args = (thread_args_t*)x;
    printf("Starting producer thread %d\n", args->threadId);

    while (args->numOps > 0) {
        // produce the item
        int currentTransaction = performRandomTransaction();

        // make sure there is space in the buffer
        if (sem_wait(&empty_sem)) {
            handle_error("Producer: sem_wait error empty sem");
        }
 // get exclusive write access <CRITICAL SECTION>
        if (sem_wait(&write_sem)) {
            handle_error("Producer: sem_wait error write sem");
        }

        // write the item and update write_index accordingly
        transactions[write_index] = currentTransaction;
        write_index = (write_index + 1) % BUFFER_SIZE;
 if (sem_post(&write_sem)) {
            handle_error("Producer: sem_post error write sem");
        } // </CRITICAL SECTION>

        // notify that a new element just became available
        while (1){
        if (sem_post(&fill_sem)) {
            handle_error("Producer: sem_post error fill sem");
        }
        }

        args->numOps--;
    }

    free(args);
    pthread_exit(NULL);
}
```

```
// consumer thread - uguale a 1:1
void* processTransactions(void* x) {
    thread_args_t* args = (thread_args_t*)x;
    printf("Starting consumer thread %d\n", args->threadId);

    while (args->numOps > 0) {
        // make sure there is data to consume
        if (sem_wait(&fill_sem)) {
            handle_error("Consumer: sem_wait error fill sem");
        }

        // consume the item and update (shared) variable deposit
        deposit += transactions[read_index];
        read_index = (read_index + 1) % BUFFER_SIZE;
        if (read_index % 100 == 0)
                                        printf("After the last 100
transactions balance is now %d.\n", deposit);

        // notify that a free cell in the buffer just became available
        if (sem_post(&empty_sem)) {
            handle_error("Consumer: sem_post error empty sem");
        }

        args->numOps--;
        //printf("C %d\n", args->numOps);
    }

    free(args);
    pthread_exit(NULL);
}
```

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Produttore-Consumatore

Lab02-es1: Produttore Consumatore – N:1 creazione/distruzione semafori

```
/* creazione dei semafori*/

sem_t empty_sem, fill_sem;
sem_t write_sem;
// bla bla bla

int main(int argc, char* argv[]) {

// bla bla bla

// initialize read and write indexes
    read_index  = 0;
    write_index = 0;

// initialize semaphores
    if (sem_init(&fill_sem, 0, 0)) handle_error("init error fill sem");
    if (sem_init(&empty_sem, 0, BUFFER_SIZE))  handle_error("init error
empty sem");
    if (sem_init(&write_sem, 0, 1))  handle_error("init error write
sem");

// pthread_create(), pthread_join()
// bla bla bla

    exit(EXIT_SUCCESS);
}
```

```
/* distruzione dei semafori*/

sem_t empty_sem, fill_sem;
sem_t write_sem;
// bla bla bla

int main(int argc, char* argv[]) {

// pthread_create(), pthread_join()

printf("Final value for deposit: %d\n", deposit);

    // destroy semaphores
    if (sem_destroy(&fill_sem)) handle_error("Fill sem destroy error");
    if (sem_destroy(&empty_sem)) handle_error("Empty sem destroy
error");
    if (sem_destroy(&write_sem)) handle_error("write sem destroy
error");
    exit(EXIT_SUCCESS);
}
```

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Produttore-Consumatore

Lab02-es1: Produttore Consumatore – N:M uso dei semafori

```
/* program bounded buffer 1:M*/
sem_t empty_sem, fill_sem;
sem_t read_sem, write_sem;
```

```
// producer thread - come N:1
void* performTransactions(void* x) {
    thread_args_t* args = (thread_args_t*)x;
    printf("Starting producer thread %d\n", args->threadId);

    while (args->numOps > 0) {
        // produce the item
        int currentTransaction = performRandomTransaction();

        // make sure there is space in the buffer
        if (sem_wait(&empty_sem)) {
            handle_error("Producer: sem_wait error empty sem");
        }
 // get exclusive write access <CRITICAL SECTION>
        if (sem_wait(&write_sem)) {
            handle_error("Producer: sem_wait error write sem");
        }

        // write the item and update write_index accordingly
        transactions[write_index] = currentTransaction;
        write_index = (write_index + 1) % BUFFER_SIZE;
 if (sem_post(&write_sem)) {
            handle_error("Producer: sem_post error write sem");
        } // </CRITICAL SECTION>

        // notify that a new element just became available
        while (1){
        if (sem_post(&fill_sem)) {
            handle_error("Producer: sem_post error fill sem");
        }
        }

        args->numOps--;
    }

    free(args);
    pthread_exit(NULL);
}
```

```
// consumer thread
void* processTransactions(void* x) {
    thread_args_t* args = (thread_args_t*)x;
    printf("Starting consumer thread %d\n", args->threadId);

    while (args->numOps > 0) {
        // make sure there is data to consume
        if (sem_wait(&fill_sem)) {
            handle_error("Consumer: sem_wait error fill sem");
        }

        // get exclusive read access <CRITICAL SECTION>
        if (sem_wait(&read_sem)) {
            handle_error("Consumer: sem_wait error read sem");
        }

        // consume the item and update (shared) variable deposit
        deposit += transactions[read_index];
        read_index = (read_index + 1) % BUFFER_SIZE;
        if (read_index % 100 == 0)
                                                printf("After the last 100 transactions
balance is now %d.\n", deposit);

        if (sem_post(&read_sem)) {
            handle_error("Consumer: sem_post error read sem");
        } // </CRITICAL SECTION>

        // notify that a free cell in the buffer just became available
        if (sem_post(&empty_sem)) {
            handle_error("Consumer: sem_post error empty sem");
        }

        args->numOps--;
        //printf("C %d\n", args->numOps);
    }

    free(args);
    pthread_exit(NULL);
}
```

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Produttore-Consumatore

Lab02-es1: Produttore Consumatore – N:M creazione/distruzione semafori

```
/* creazione dei semafori*/

sem_t empty_sem, fill_sem;
sem_t read_sem, write_sem;
// bla bla bla

int main(int argc, char* argv[]) {

// bla bla bla

// initialize read and write indexes
    read_index  = 0;
    write_index = 0;

// initialize semaphores
    if (sem_init(&fill_sem, 0, 0)) handle_error("init error fill sem");
    if (sem_init(&empty_sem, 0, BUFFER_SIZE))  handle_error("init error
empty sem");
    if (sem_init(&read_sem, 0, 1))  handle_error("init error read sem");
    if (sem_init(&write_sem, 0, 1))  handle_error("init error write
sem");

// pthread_create(), pthread_join()
// bla bla bla

    exit(EXIT_SUCCESS);
}
```

```
/* distruzione dei semafori*/

sem_t empty_sem, fill_sem;
sem_t read_sem, write_sem;
// bla bla bla

int main(int argc, char* argv[]) {

// pthread_create(), pthread_join()

printf("Final value for deposit: %d\n", deposit);

    // destroy semaphores
    if (sem_destroy(&fill_sem)) handle_error("Fill sem destroy error");
    if (sem_destroy(&empty_sem)) handle_error("Empty sem destroy
error");
    if (sem_destroy(&read_sem)) handle_error("read sem destroy error");
    if (sem_destroy(&write_sem)) handle_error("write sem destroy
error");
    exit(EXIT_SUCCESS);
}
```

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Sol: Sincronizzazione Inter-Process

Lab02, Esercizio 2

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Sincronizzazione Inter-Processo

Lab02-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

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Produttore-Consumatore

Lab02-es2: Named Semaphore – Server

```
/* server semaphore management/

void cleanup() {
    /** Close and then unlink the named semaphore **/
    printf("\rShutting down the server...\n");

    if (sem_close(named_semaphore)) handle_error("sem_close error");

    /* We unlink the semaphore, otherwise it would remain in the
     * system after the server dies. */
    if (sem_unlink(SEMAPHORE_NAME)) handle_error("sem_unlink error");

    exit(0);
}
int main(int argc, char* argv[]) {
    int ret;

    /** Create a named semaphore to be shared with different processes.[…] */
// bla bla bla

sem_unlink(SEMAPHORE_NAME);
    named_semaphore = sem_open(SEMAPHORE_NAME, O_CREAT | O_EXCL, 0600, NUM_RESOURCES);

    if (named_semaphore == SEM_FAILED) {
        handle_error("Could not open the named semaphore");
    }
return 0;
}
```

Produttore-Consumatore

Lab02-es2: Named Semaphore – Client

```
/* client semaphore utilization/

void* client(void *arg_ptr) {
    thread_args_t* args = (thread_args_t*) arg_ptr;
    char errorStr[100];

    /*** Open an existing named semaphore - COMPLETE HERE ***/
    sem_t* my_named_semaphore = sem_open(SEMAPHORE_NAME, 0); // oflag is 0: sem_open is not allowed to create it!
    if (my_named_semaphore == SEM_FAILED) {
        sprintf(errorStr,"Could not open the named semaphore from thread %d", args->ID);
        handle_error(errorStr);
    }

    /*** Acquire the resource ***/
    if (sem_wait(my_named_semaphore)) {
        sprintf(errorStr,"Could not lock the semaphore from thread %d", args->ID);
        handle_error(errorStr);
    }
    // bla bla bla

/*** Free the resource ***/
    if (sem_post(my_named_semaphore)) {
        sprintf(errorStr,"Could not unlock the semaphore from thread %d", args->ID);
        handle_error(errorStr);
    }

    printf("[@Thread%d] Done. Resource released!\n", args->ID);

    /*** Close the named semaphore - COMPLETE HERE ***/
    if(sem_close(my_named_semaphore)){
        sprintf(errorStr,"Could not close the semaphore from thread %d", args->ID);
        handle_error(errorStr);

    }

    free(args);
    return NULL;
}
```

Sol: Produttori-Consumatori su File

Lab02, Esercizio 3

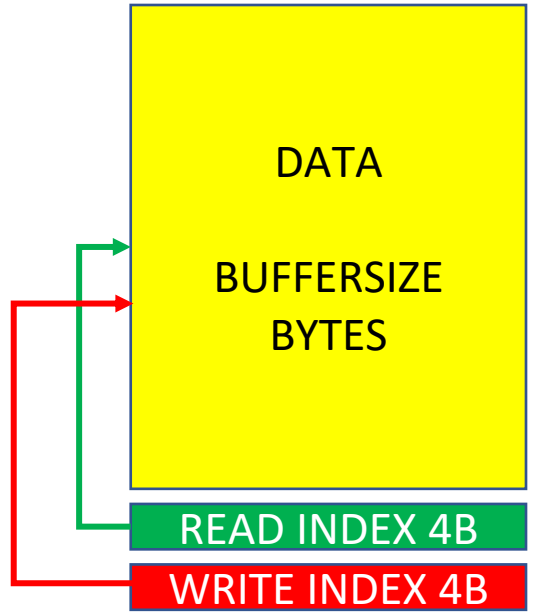
Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Esercizio per Casa

Lab02-es3 Buffer Circolare su File

- Paradigma Produttore-Consumatore tra più processi con letture e scritture da un 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



Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Produttore-Consumatore

Lab02-es3 Buffer Circolare su File – semaphore initialization

```
/* server semaphore management/

void initSemaphores() {
    // delete stale semaphores from a previous crash (if any)
    sem_unlink(SEMNAME_FILLED);
    sem_unlink(SEMNAME_EMPTY);
    sem_unlink(SEMNAME_CS);

/* TODO: create the semaphores as described above */
    sem_filled = sem_open(SEMNAME_FILLED, O_CREAT | O_EXCL,
0600, 0);
    if (sem_filled == SEM_FAILED) handle_error("sem_open
filled");

    sem_empty = sem_open(SEMNAME_EMPTY, O_CREAT | O_EXCL,
0600, BUFFER_SIZE);
    if (sem_empty == SEM_FAILED) handle_error("sem_open
empty");

    sem_cs = sem_open(SEMNAME_CS, O_CREAT | O_EXCL, 0600, 1);
    if (sem_cs == SEM_FAILED) handle_error("sem_open cs");
}

void closeSemaphores() {
    int ret = sem_close(sem_filled);
    if (ret) handle_error("sem_close filled");

    ret = sem_close(sem_empty);
    if (ret) handle_error("sem_close empty");

    ret = sem_close(sem_cs);
    if (ret) handle_error("sem_close cs");
}
```

```
/* client semaphore init/
oid openSemaphores() {

    /* We have to open 3 named semaphores */

    /* TODO: open the semaphores as described above */

    sem_filled = sem_open(SEMNAME_FILLED, 0);
    if (sem_filled == SEM_FAILED) handle_error("sem_open filled");

    sem_empty = sem_open(SEMNAME_EMPTY, 0);
    if (sem_empty == SEM_FAILED) handle_error("sem_open empty");

    sem_cs = sem_open(SEMNAME_CS, 0);
    if (sem_cs == SEM_FAILED) handle_error("sem_open cs");}

oid closeAndDestroySemaphores() {
    int ret;

    // first close
    ret = sem_close(sem_filled);
    if (ret) handle_error("sem_close filled");

    ret = sem_close(sem_empty);
    if (ret) handle_error("sem_close empty");

    ret = sem_close(sem_cs);
    if (ret) handle_error("sem_close cs");

    // then unlink
    ret = sem_unlink(SEMNAME_FILLED);
    if (ret) handle_error("sem_unlink filled");

    ret = sem_unlink(SEMNAME_EMPTY);
    if (ret) handle_error("sem_unlink empty");

    ret = sem_unlink(SEMNAME_CS);
    if (ret) handle_error("sem_unlink cs");

}
```

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Produttore-Consumatore

Lab02-es3 Buffer Circolare su File – semaphore usage

```
/* server semaphore usage/

void produce(int id, int numOps) {
    int localSum = 0;
    while (numOps > 0) {
        // NCS: producer, just do your thing!
        int value = performRandomTransaction();

        int ret = sem_wait(sem_empty);
        if (ret) handle_error("sem_wait empty\n");

        ret = sem_wait(sem_cs);
        if (ret) handle_error("sem_wait cs");

          //CS
        writeToBufferFile(value, BUFFER_SIZE,
BUFFER_FILENAME);

        ret = sem_post(sem_cs);
        if (ret) handle_error("sem_post cs");

        ret = sem_post(sem_filled);
        if (ret) handle_error("sem_post filled");

        localSum += value;
        numOps--;
    }
    printf("Producer %d ended. Local sum is %d\n",
id, localSum);
}
```

```
/* client semaphore usage/

void consume(int id, int numOps) {
    int localSum = 0;
    while (numOps > 0) {
        int ret = sem_wait(sem_filled);
        if (ret) handle_error("sem_wait filled");

        ret = sem_wait(sem_cs);
        if (ret) handle_error("sem_wait cs");

        // CS
        int value = readFromBufferFile(BUFFER_SIZE,
BUFFER_FILENAME);

        ret = sem_post(sem_cs);
        if (ret) handle_error("sem_post cs");

        ret = sem_post(sem_empty);
        if (ret) handle_error("sem_post empty");

          //NCS
        localSum += value;

        numOps--;
    }
    printf("Consumer %d ended. Local sum is %d\n", id,
localSum);
}
```

IPC: Usare la Shared Memory POSIX

- Obiettivi dell'Esercitazione

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Shared Memory

POSIX Shared Memory - presentazione

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

Shared Memory

POSIX Shared Memory - requisiti

```
#include <sys/mman.h>   /* memory management */
#include <sys/stat.h>   /* mode constants */
#include <sys/types.h>  /* type definition */
#include <fcntl.h>      /* O_* constants */
#include <unistd.h>     /* API of POSIX standard */
```

Link with *-lrt*. (Real-Time library), for memory management.

Shared Memory

Funzione `shm_open()` - caratteristichei

```
#include <sys/mman.h>   /* memory management */
#include <sys/stat.h>     /* mode constants */
#include <fcntl.h>        /* O_* constants */

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 `"/name";` vale a dire una stringa (apparirà il file `/dev/shm/name`)
  - `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

Shared Memory

Funzione `ftruncate()` - caratteristichei

```
#include <unistd.h>        /* API of POSIX standard */
#include <sys/types.h>     /* type definitions */

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

Shared Memory

Funzione `mmap()` - caratteristichei

```
#include <sys/mman.h>   /* memory management */

void *mmap(void addr[.length], 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**
  - `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.

Shared Memory

Funzione `munmap()` - caratteristichei

```
#include <sys/mman.h>   /* memory management */

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

Shared Memory

Funzione `shm_unlink()` - caratteristichei

```
#include <sys/mman.h>   /* memory management */
#include <sys/stat.h>     /* mode constants */
#include <fcntl.h>        /* O_* constants */

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

Shared Memory

Funzione `close()` - caratteristichei

```
#include <unistd.h>        /* API of POSIX standard */

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

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Scrivere e Leggere in ShmMem

- Esempio

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Shared Memory

Scrivere: **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);
        munmap(ptr, SIZE);
        sprintf(ptr, "%s", msg);
        close(shm_fd);
        return 0;
}
```

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Shared Memory

Leggere: **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);
        munmap(ptr, SIZE);
        close(shm_fd);
        shm_unlink(name);
        return 0;
}
```

Applicazione Modulare

- Lab03, Esercizio 1

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Shared Memory

Lab03-es1: 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`)

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Produttore/Consumatore

- Lab03, Esercizio 2

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Shared Memory

Lab03-es2: Produttore/Consumatore

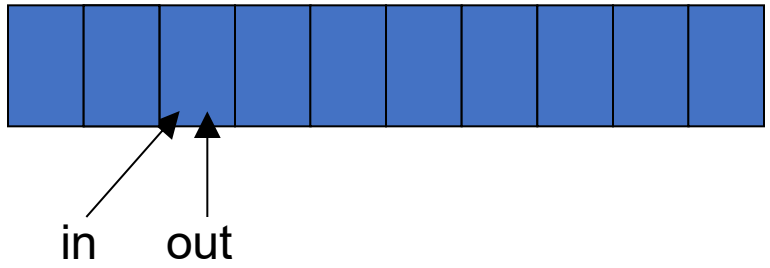
- L'applicazione è sviluppata in due moduli separati.
- Si tiene conto della configurazione con `NUM_CONSUMERS` consumatori e `NUM_PRODUCERS` produttori
- Il buffer e le posizioni di in e out sono posizionati in memoria condivisa

- 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`

Shared Memory

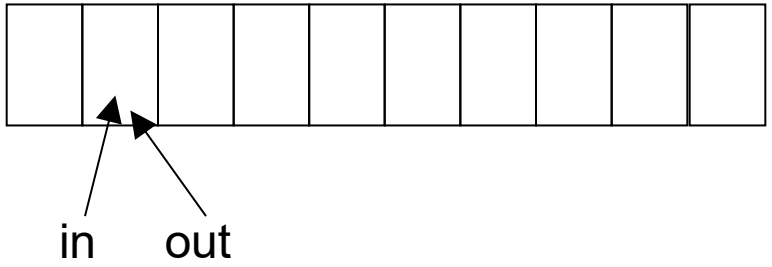
Lab03-es2: Buffer State in Shared Memory

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

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Producer/Consumer senza semafori

- Lab03, Esercizio 3

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Shared Memory

Lab03-es3: Producer/Consumer 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`

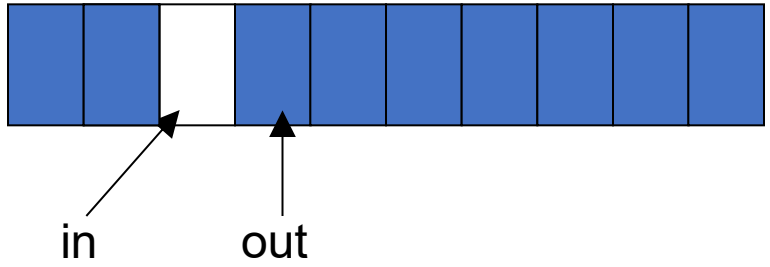
Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Shared Memory

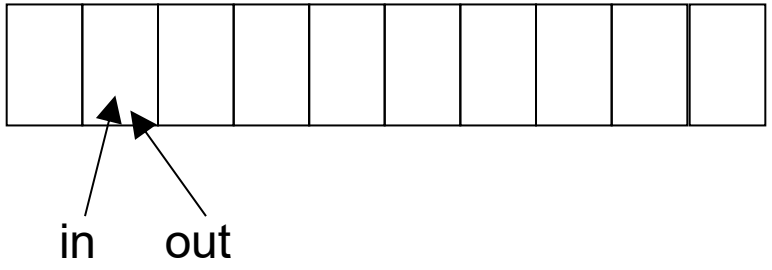
Lab03-es3: Buffer State in Shared Memory (senza semafori)

Buffer Full



(in +1) % BUFFER_SIZE == out

Buffer Empty



in == out

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Questionario di Autovalutazione

Link

https://forms.gle/UxDTVRKBLsBDpq1S6