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 [01] Processi, Thread e concorrenza

Riccardo Lazzeretti <u>lazzeretti@diag.uniroma1.it</u>
Paolo Ottolino <u>paolo.ottolini@uniroma1.it</u>
Matteo Cornacchia cornacchia@diag.uniroma1.it



Partecipazione alle esercitazioni

https://forms.gle/6Kr8kgRE2x8dRjdbA





Sommario

- Thread
 - Riepilogo primitive C
 - Esempi
- Processi vs thread
 - Tempi per lancio e terminazione
 - Interazione con la memoria virtuale
- Accesso concorrente a variabili condivise



Obiettivi

- 1. Ripassare le basi della programmazione multi-threading
 - a. Impostare un'applicazione multi-threading
 - 2. Comprendere le problematiche legate all'accesso concorrente
 - 3. Risolvere il problema



Obiettivi Esercitazione

- Imparare ad usare i semafori in C
 - **a.** Come si implementa la mutua esclusione per l'accesso ad una sezione critica?
 - **b.** Quanto vale l'overhead dei semafori?
 - **C.** Come si implementa l'accesso in mutua esclusione a N risorse distinte?





Primitive 1/2

```
int pthread_create (
    phtread_t* thread,
    const pthread_attr_t* attr,
    void* (*start_routine)(void*),
    void* arg);
```

- thread: puntatore a variabile di tipo pthread_t, su cui verrà memorizzato l'ID del thread creato
- □ attr: attributi di creazione ← sempre NULL in questo Corso
 - ☐ https://man7.org/linux/man-pages/man3/pthread_attr_init.3.html
- start_routine: funzione da eseguire (prende sempre come argomento un void* e restituisce un void*)
- ☐ arg: puntatore da passare come argomento alla funzione start routine
- ✓ Return value: 0 in caso di successo, altrimenti la causa dell'errore



Primitive 2/2

```
void pthread_exit (void* value_ptr);

□ Termina il thread corrente, rendendo disponibile il valore puntato da value_ptr ad una eventuale operazione di join
□ All'interno di start_routine, può essere sostituita da return

int pthread_join (phtread_t thread, void** value_ptr);
□ Attende esplicitamente la terminazione del thread con ID thread
□ Se value_ptr!=NULL, vi memorizza il valore eventualmente restituito dal thread (un void*, tramite pthread_exit)
✓ Return value: 0 in caso di successo, altrimenti la causa dell'errore
```

```
int pthread_detach(phtread_t thread);
```

- Notifica al sistema che non ci saranno operazioni di join su thread
- ✔ Return value: 0 in caso di successo, altrimenti la causa dell'errore



Esempio 1

```
#include <stdio.h>
#include <errno.h>
#include <pthread.h>
void* thread stuff(void *arg) {
    // codice thread che non usa argomenti
    return NULL;
int ret;
pthread t thread;
ret = pthread create(&thread, NULL, thread stuff, NULL);
if (ret != 0) {
    fprintf(stderr, "ERROR with pthread create!\n");
    exit(EXIT FAILURE);
// codice main indipendente dal thread
ret = pthread join(thread, NULL);
if (ret != 0)^{-}[...]
```



Esempio 2: Thread Multipli

```
#include <stdio.h>
#include <errno.h>
#include <pthread.h>
#define NUM THREADS 4
void *hello(void *arg) {
      printf («Hello Thread\n»);
main() {
  pthread t tid [NUM THREADS];
  for (int i = 0; i < NUM THREADS; i++)
    ret = pthread create(&tid[i], NULL, hello, NULL);
  for (int i = 0; i < NUM THREADS; i++)
    ret = pthread join(tid[i], NULL);
```



Esempio 3: Passaggio di Argomenti

lancio di N thread con argomenti necessariamente distinti

```
#include <errno.h>
#include <pthread.h>
void* foo(void *arg) {
  type t* obj ptr = (type t*) arg; // cast ptr argomenti
  /* <corpo del thread> */
  free (obj ptr);
  return NULL;
int ret;
pthread t* threads = malloc(N * sizeof(pthread t));
type t* objs = malloc(N * sizeof(type t));
for (i=0; i<N; i++) {
  objs[i] = [...] // imposto argomenti thread i-esimo
  ret = pthread create(&threads[i], NULL, foo, &objs[i]);
  if (ret != 0) [...]
```



Accesso concorrente a variabili condivise

☐ LabO1, Esercizio 1



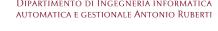
Lab01-es1: Accesso concorrente a variabili condivise - presentazione

- Cosa succede quando più thread accedono in scrittura ad una variabile condivisa in concorrenza?
- ☐ Sorgente: concurrent threads.c
- U Compilazione: gcc -o concurrent_threads concurrent_threads.c -lpthread
- N thread in parallelo che aggiungono M volte un valore V ad una variabile condivisa (inizializzata a 0)
- La variabile condivisa alla fine dovrebbe valere N*M*V: succede sempre?



Lab01-es1: Accesso concorrente a variabili condivise - specifiche

- Nei prossimi esercizi presenteremo meccanismi di sincronizzazione pensati per risolvere questi problemi
- Tuttavia, è possibile implementare una soluzione che non usa meccanismi di sincronizzazione, pur mantenendo la semantica originale:
 - N thread effettuano in parallelo M incrementi di valore V
 - Al termine, il main thread verifica che tali incrementi equivalgano complessivamente a N*M*V
- Modificare concurrent_threads.c di conseguenza
 - Suggerimento: lavorare sulle strutture dati per evitare accessi concorrenti in scrittura





Lab01-es1: Accesso concorrente a variabili condivise – soluzione 1/3

- Esercizio: implementare una soluzione al seguente problema <u>senza</u> meccanismi di sincronizzazione:
 - N thread effettuano in parallelo M incrementi di valore V
 - Al termine, il main thread verifica che tali incrementi equivalgano complessivamente a N*M*V
 - Suggerimento: lavorare sulle strutture dati per evitare accessi concorrenti in scrittura

Soluzione

- ogni thread incrementa una locazione di memoria diversa
- alla fine il main thread somma tutti i valori
- sorgente: sol concurrent threads.c





Lab01-es1: Accesso concorrente a variabili condivise – soluzione 2/3 [recap]

Compilazione

```
gcc -o sol_concurrent_threads sol_concurrent_threads.c
-lpthread
```

- Esecuzione
 - ./sol concurrent threads <N> <M> <V>
- Non dovrebbero risultare add perse
- Cosa succede se invece di usare &thread_ids[i] usiamo &i?



DIPARTIMENTO DI INGEGNERIA INFORMATICA



Lab01-es1: Accesso concorrente a variabili condivise – soluzione 3/3 [recap]

 Non si può avere alcuna garanzia riguardo a quando verrà eseguita l'istruzione

```
int thread_idx = *((int*)arg);
```

- Nel mentre, può succedere che il valore nella locazione di memoria puntata da arg venga cambiato
 - È il valore del contatore i
 - Più thread con la stessa «identità» (thread idx)
 - Si ripropone il problema dell'accesso concorrente





Lab01-es1: Accesso concorrente a variabili condivise – soluzione alternativa

- Ogni thread lavora su variabili locali e restituisce il valore tramite pthread exit
- Il main raccoglie i valori tramite pthread join e li somma
- Compilazione

```
gcc -o sol2_concurrent_threads
sol2 concurrent threads.c -lpthread
```

Esecuzione

```
./sol2 concurrent threads <N> <M> <V>
```



DIPARTIMENTO DI INGEGNERIA INFORMATICA



Accesso in sezione critica in mutua esclusione

☐ Lab01, Esercizio 2



Semafori 1/7

1. <u>Inizializzazione</u>

Assegna un valore iniziale non negativo al semaforo

Operazione semWait

Decrementa il valore del semaforo, se il valore è negativo il processo/thread viene messo in attesa in una coda, altrimenti va avanti

Operazione semSignal

Incrementa il valore del semaforo, se il valore non è positivo un processo/thread viene risvegliato dalla coda





Semafori 2/7

```
#include <semaphore.h>---
                                               Header da includere
sem t sem;
sem init(&sem, pshared, value)
sem wait(&sem)
sem post(&sem)
sem destroy(&sem)
```



Semafori 3/7

```
#include <semaphore.h>
sem t sem; ——
sem init(&sem, pshared, value)
sem wait(&sem)
sem post(&sem)
sem destroy(&sem)
```

Dichiarazione di una variabile di tipo sem_t, che rappresenta il nostro semaforo



Semafori 4/7

```
#include <semaphore.h>
sem t sem;
sem init(&sem, pshared, value)
sem wait(&sem)
sem post(&sem)
sem destroy(&sem)
```

Inizializzazione del semaforo con valore value.

Se pshared vale 0, il semaforo viene condiviso tra i thread del processo; altrimenti, il semaforo viene condiviso tra processi, a patto che sia in una porzione di memoria condivisa (quest'ultimo caso non verrà esaminato nel corso).

In caso di successo, viene ritornato 0; in caso di errore, -1.



Semafori 5/7

```
#include <semaphore.h>
sem t sem;
sem init(&sem, pshared, value)
sem wait(&sem)
sem post(&sem)
sem destroy(&sem)
```

Operazione semWait sul semaforo sem.

In caso di successo, viene ritornato 0; in caso di errore, -1.



Semafori 6/7

```
#include <semaphore.h>
sem t sem;
sem init(&sem, pshared, value)
sem wait(&sem)
sem post(&sem)
sem destroy(&sem)
```

Operazione semSignal sul semaforo sem.

In caso di successo, viene ritornato 0; in caso di errore, -1.

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti



Semafori 7/7

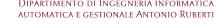
```
#include <semaphore.h>
sem t sem;
sem init(&sem, pshared, value)
sem wait(&sem)
sem post(&sem)
sem destroy(&sem)
```

Distrugge il semaforo sem. In caso di successo, viene ritornato 0; in caso di errore, -1.



Lab01-es2: Accesso sezione critica in mutua esclusione - presentazione

- Riprendiamo concurrent_threads e risolviamo il problema delle race condition
 - Sezione critica: shared variable += v;
 - Va protetta con un semaforo
 - Acquisizione lock sulla sezione critica tramite sem_wait
 - Esecuzione sezione critica
 - Rilascio lock sulla sezione critica tramite sem_post
 - Sorgente: concurrent_threads.c



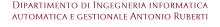


Semafori in C

Lab01-es2: Accesso sezione critica in mutua esclusione - specifiche

- Garantire mutua esclusione utilizzando i semafori
 - Creare una copia del file e chiamarla concurrent_threads_semaphore.c
 - Introdurre opportunamente i semafori
 - Compilazione:

```
gcc -o concurrent_threads_semaphore concurrent_threads_semaphore.c
performance.c -lpthread -lrt -lm
```





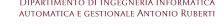
Semafori in C

Lab01-es2: Accesso sezione critica in mutua esclusione - funzionalità

- Misurazione delle prestazioni
 - Viene usata la libreria performance per misurare il tempo di esecuzione
 - effettuare un confronto sui tempi di esecuzione tra questa soluzione e quelle senza semafori

• Compilazione:

```
gcc -o concurrent_threads_semaphore
concurrent_threads_semaphore.c performance.c
-lpthread -lrt -lm
```





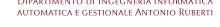
Accesso in mutua esclusione a N risorse

☐ LabO1, Esercizio 3



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





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

- Soluzione: inizializzare il semaforo a ${ t N}$ 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





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 ()





- 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





- 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





- 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?





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





Autovalutazione

https://forms.gle/FrEFp4Rq5RBdW29u8



