tmp

u7-s1-thread

Sistemi Operativi

Unità 7: I Thread

I Thread in Linux

Martino Trevisan
Università di Trieste
Dipartimento di Ingegneria e Architettura

Argomenti

1. Concetto di Thread
2. Thread in Linux
3. Funzioni per i Pthread
4. Esempi
5. Thread in Bash

Concetto Teorico di Thread

Definizione di Thread

In Linux (e in quasi tutti i SO), un *processo* può avere molteplici flussi di esecuzione, detti *Thread*

- I thread possono essere visti come un *insieme di processi che condividono la memoria*
- Ma eseguono lo stesso programma

Nota: anche Windows permette di creare thread con la System Call `CreateThread()`

Ogni Thread esegue *lo stesso programma e condivide gli stessi dati*

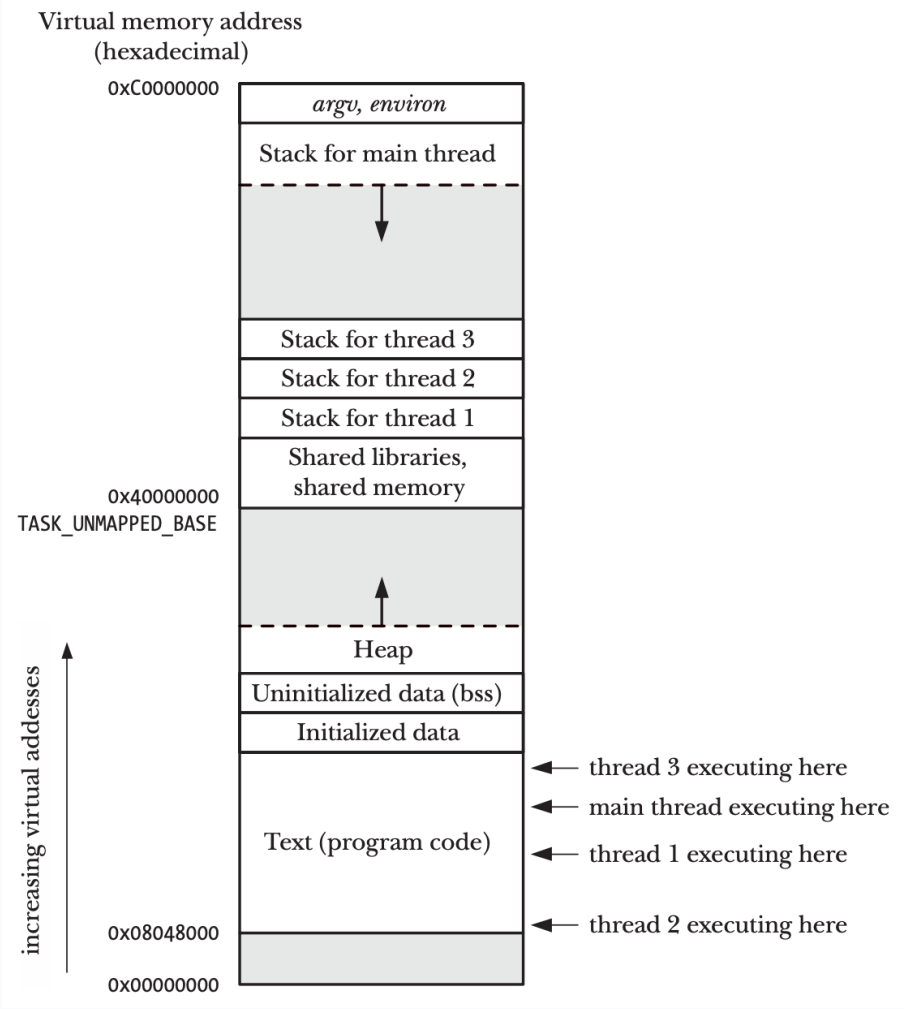
- I segmenti *data*, *heap* e *code* (ζ, δ, ξ) sono condivisi

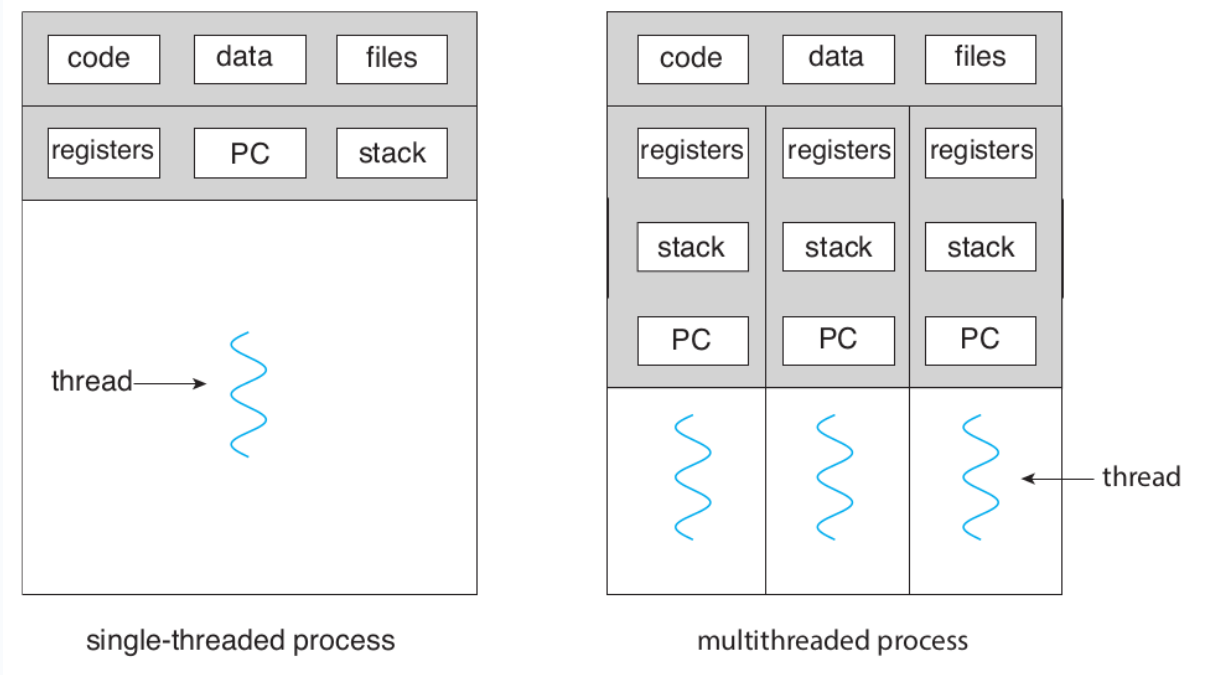
Un Thread é un *flusso del codice in esecuzione*

- Ha il *suo stack*
- Contiene lo *stato delle funzioni in esecuzione*

Ogni *thread ha uno stack*

- E chiaramente opera su *Registri* e ha un *Program Counter*





Comunicazione tra Thread

I Thread possono comunicare tra loro più facilmente che i processi, usando:

- *Variabili globali* in δ (che è *nativamente* condivisa)
- Costrutti di sincronizzazione
  - *Mutex*
  - *Condition Variable* (vedremo solo sommariamente)
  - *Semafori*

Oggigiorno é più spesso usata un'architettura *multi-thread* che *multi-process*. In realtà dipende sono scelte: ad esempio *Chrome* è ha un paradigma *multi-process*. Ognuno ha i suoi vantaggi e svantaggi;

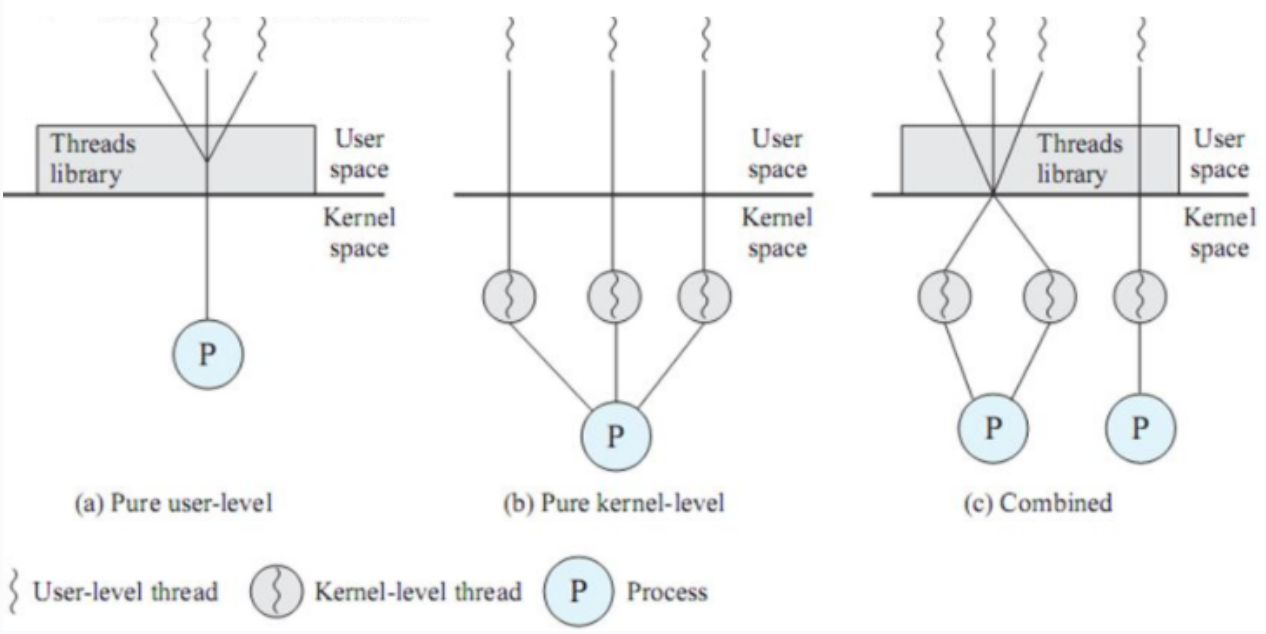
- *Multi-thread:* un po' più *"semplice"* da implementare e sincronizzare. Però se il processo principale cade, gli alti thread cadono assieme
- *Multi-process:* complicato da sincronizzare, tuttavia non ha lo svantaggio di programmi multi-thread

User e Kernel Thread

Esistono due modi per implementare i thread.

- **Kernel Thread**: il *kernel* permette di creare thread
  - Sono di fatto dei processi *light*
  - Vedremo questi, principalmente (in Linux)
- **User Thread**: creati dal *programmatore* o da una *libreria*
  - Il processo (in qualche modo) gestisce e orchestra più flussi di esecuzione
  - Il kernel ne è allo scuro
  - Molto complicato! Inoltre ho le limitazioni dell'esecuzione in *User Mode*

  Poi si può fare anche robe strane, come *combinarle*, ma ne staremo allo scuro.



Thread in Linux

LinuxThreads

Inizialmente i Pthread erano implementati dalla libreria *LinuxThreads*

- I thread erano dei processi che condividevano la memoria, i file aperti, ecc.
- Ognuno aveva *diverso PID*
- *Implementazione problematica:* si mischiava concetto di thread e processo. In quell'epoca non c'era ancora il supporto *nativo* di Thread, infatti non avevo altro che delle *fork sofisticate*.

Ora (da 2002), Linux/POSIX usa la libreria *Native POSIX Threads Library (NPLT)*

- Coopera col kernel, che offre supporto ai thread
- Migliori prestazioni

Posix Thread

Nei sistemi POSIX (e Linux), le *funzioni di libreria* per gestire i thread sono chiamate *Pthread*
I thread permettono a un processo:

- Di svolgere più task in maniera concorrente
  - Mentre un thread attende l'I/O o la rete, un altro thread può svolgere un altro compito
- Di sfruttare un sistema *multi-core*
  - Più flussi davvero in esecuzione parallela

I thread in Linux sono *Kernel Thread*

Qui i Posix-Threads condividono:

- La *memoria globale*
- PID e PPID
- File aperti
- Privilegi
- Working directory

Ogni thread ha invece le seguente caratteristiche distinte:

- Un *Thread ID*
  - Il Kernel mantiene la lista dei thread e li *schedula*, facendoli eseguire sulla CPU. Identificativo univoco per il sistema.
- Il suo *stack*
  - Per poter eseguire le funzioni

    - Un thread *mal configurato* puó comunque accedere/corrompere lo *stack* di un altro thread (comunque una cattiva idea!)
- Metadati: scheduling, etc...

Compilazione con Pthreads

Il codice deve includere la direttiva:

```c
#include <pthread.h>
```

Per compilare, bisogna includere la libreria **pthread**

```shell
gcc MyProgram.c -o MyProgram -lpthread
```

Funzioni per i Pthread

Adesso vediamo come *lavorare* con i *Pthread*

Creazione di un thread

```c
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start)(void *), void *arg);
```

Crea un nuovo thread che esegue la funzione **start** chiamata con l'argomento **arg**

- Come se si invocasse **start(arg)** su un *flusso di esecuzione separato*

Nota: Ogni programma, quando nasce, ha un solo thread, detto *main thread*

PARAMETRI E VALORI DI RITORNO.

- L'argomento **arg** é un **void***, ovvero un puntatore a un tipo di dato a piacere. Questo per avere la *massima flessibilità*.
- Similmente, il valore di ritorno di **start** é un **void***.
- Non ci interessa l'argomento **attr** che specifica attributi particolari

- L'argomento `thread` é un puntatore a una variabile `pthread_t` che andrà a contenere il Thread ID, per poterlo usare in successive funzioni di libreria (poi per lavorarci sopra)
- In caso di successo, ritorna 0, altrimenti un codice di errore

Nota Implementativa
La `pthread_create()` è una *funzione di libreria*

Essa usa la System Call `int clone( ... )`

- La `clone()` è simile alla `fork()`
- Crea un *processo figlio*
- Più *flessibile* e precisa della `fork()`
  - Permette di *controllare cosa condividono* padre e figlio
- La `pthread_create()` crea un nuovo processo che *condivide la memoria* col padre
  - Che è la definizione di *Thread*

Terminazione di un thread

Un thread termina se:

- La funzione di lancio `start` esegue una `return` (quindi è *finita*)
- Il thread esegue una `pthread_exit()`
- Il thread viene cancellato tramite una `pthread_cancel(pthread_t thread);`, invocata da un altro thread
- Il processo termina se un qualsiasi thread invoca una `exit()` o il thread principale termina il `main`

```c
include <pthread.h>
void pthread_exit(void *retval);
```

- Termina il *thread corrente* col valore `retval`.
- Equivalente a effettuare una `return` nella funzione di avvio del thread.

Thread ID

```c
include <pthread.h>
pthread_t pthread_self(void);
```

Permette a un thread di ottenere il *proprio Thread ID*.

Il Thread ID va trattato come un *handle opaco*

- Su Linux é un **long int**
- Ma potrebbe essere un puntatore a una struttura dati arbitraria
- Non é affidabile decifrarne il valore

Join di un thread

```c
include <pthread.h>
int pthread_join(pthread_t thread, void **retval);
```

Attende che il thread **thread** *termini*.

- Se é già terminato, ritorna istantaneamente

Immagazzina il valore di ritorno all'indirizzo **retval**

- **retval** é specificato dal *thread morente* tramite **pthread_exit()** o **return**
- **retval** é un **void****, ovvero un puntatore a puntore a **void**
  - E' l'indirizzo di una variabile che contiene un puntatore. Infatti devo salvare su un *puntatore a void*, quindi devo avere l'*indirizzo del puntatore a void*, sarebbe il *puntatore al puntatore a void*.

I thread devono essere tutti attesi tramite una **pthread_join()**, altrimenti diventano zombie

- *Come avviene per i processi*

Usando la funzione **int pthread_detach(pthread_t thread)** è possibile indicare che il thread **thread** non necessita di una **join**

- Il valore di ritorno viene *scartato*
- Il sistema rimuove ogni informazione sul thread quando esso termina

Note:
I thread sono pari tra loro

- Qualunque thread può fare una **pthread_join** su un altro; anche se comunque non è (di solito) una buona idea fare *join* tra fratelli
  Non esiste un modo per aspettare la terminazione di un *qualsiasi* thread
- Coi processi si può invece usare la **wait**.
  Una **pthread_join** é sempre bloccante
- Diverso da **waitpid** con flag **WNOHANG**

Esempio di Creazione di un Thread

Creazione di un Thread

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static void * threadFunc(void *arg){
    printf("From Thread: %s", (char *) arg);
    int * ret = malloc(sizeof(int));
    *ret = strlen(arg);
    return ret ; // Valore di ritorno del thread
    // Equivale a pthread_exit(ret);
}

int main(int argc, char *argv[]){
    pthread_t t1;
    void *res; // Per valore di ritorno
    int s;

    s = pthread_create(&t1, NULL, threadFunc, "Hello world\n"); // Creazione
    if (s ≠ 0){
        printf("Cannot create thread");
        exit(1);
    }

    printf("Message from main()\n");
    s = pthread_join(t1, &res); // Join. Richiede un void **, ovvero &res
    if (s ≠ 0){
        printf("Cannot join thread");
        exit(1);
    }
    printf("Thread returned %d\n", *((int *)res) ); // Utilizzo del valore di ritorno
    free (res); // Needed as that zone was allocated with malloc
    exit(0);
}
```

Esercizio

Esercizio. Si crei un programma che avvia 10 thread che attendono un tempo casuale tra 0 e 5 secondo prima di terminare

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define MAXSLEEP 5
#define THREADNB 10

static void * sleepFunc(void *arg){
    char thread_number = *((char*)arg);
    int n=rand() % MAXSLEEP;
    sleep(n);
    printf("Thread %c terminated after %d seconds\n", thread_number, n);
    return NULL;
}

int main(int argc, char *argv[]){
    int i;
    pthread_t t [THREADNB];
    char names [THREADNB];

    for (i=0;i<THREADNB;i++){
        names[i] = 'A' + i;
        pthread_create(&t[i], NULL, sleepFunc, &names[i]);
    }
    for (i=0;i<THREADNB;i++)
        pthread_join(t[i], NULL);
    return 0;
}
```

Thread in Bash

Normalmente, i comandi **ps** e **top** mostrano solo i processi

Per visualizzare i thread:

- **ps -T opzioni**. Esempio: **ps -T ax**
- **top -H**

Ogni thread presente nel **/proc** file system

- Come se fosse un processo: **/proc/[tid]**
- Per ottenere la lista di thread di un processo: **/proc/[pid]/task**
    - Contiene la lista dei thread di un processo

Domande

Due Thread dello stesso processo condividono le variabili globali?
• Si **• No**
RISPOSTA: *Sì*

La funzione **pthread_join** attende la terminazione:
• Di un qualsiasi thread del sistema
• Di un qualsiasi thread del processo corrente
• Di un thread specifico
RISPOSTA: *Di un thread specifico*

Quando un thread invoca la funzione **pthread_exit**:
• Il thread corrente termina
• Il processo corrente termina
• Il thread specificato come argomento della funzione termina
RISPOSTA: *Il thread corrente termina*

Si consideri il seguente codice:

```c
void * func(void *arg){
    sleep(5);
    exit(0);
}

int main(){
    ...
    pthread_create(&t, NULL, func, NULL);
    sleep (10)
    pthread_join(t, NULL);
    exit(0);
}
```

Dopo quanti secondi temina il processo?
• 5 **• 10** **• 15**
RISPOSTA: *10*

u7-s2-sync

Sistemi Operativi

Unità 7: I Thread

Sincronizzazione

[Martino Trevisan](#)
[Università di Trieste](#)
[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. Perché é necessaria
2. I mutex
3. I semafori

Motivazioni per la Sincronizzazione

Definizioni di Concorrenza e Parallelismo

Diamo delle *definizioni preliminari*.

Concorrenza: un programma con più flussi di esecuzione
Parallelismo: un programma che esegue su più calcoli contemporaneamente

Notiamo subito che non ci dev'essere nessun legame tra di loro, soprattutto del tipo $\iff$.

1. Un programma può essere *concorrente senza essere parallelo*
   - Ha tanti thread che eseguono su un sistema con una sola CPU
2. Un programma può essere *parallelo senza essere concorrente*
   - Le moderne CPU hanno istruzioni che manipolano più dati
   - Paradigma *Single Instruction Multiple Data (SIMD)*
   - Una *singola istruzione per sommare due vettori*, componente per componente
   - La CPU ha una *ALU* che permette di effettuare più operazioni in parallelo
   - Usando un *singolo thread/processo*

Obbiettivi della Programmazione Parallela

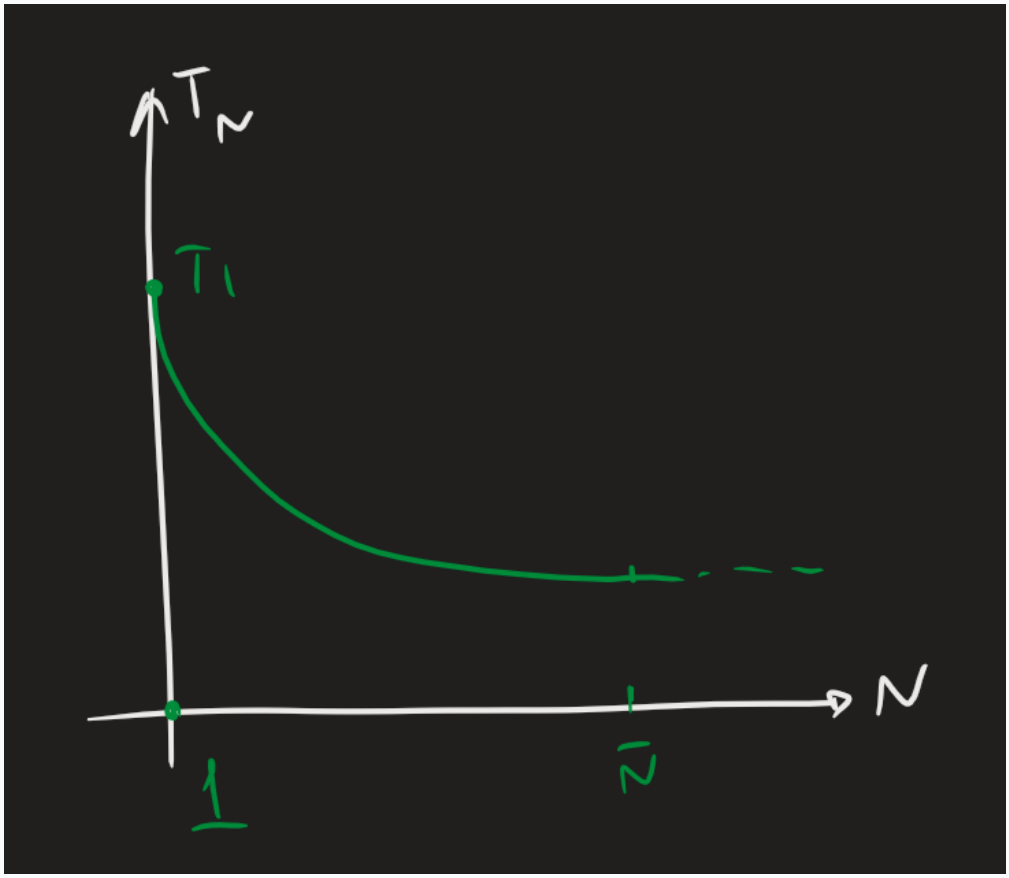
Teoricamente, parallelizzando e usando N core anzinché 1, dovremmo avere:

$$T_N = \frac{T_1}{N}$$

Ovvero *minimizziamo* il tempo T_N con un comportamento del tipo $T_N \to 0$.

In realtà, vale solo per un *numero ridotto di processori e core*.

- Solitamente, con un numero ridotto di core, si ha davvero un *incremento*
- Poi c'è un *appiattimento*. Questo è dovuto alla *legge di Ahmdal*
  Quindi si ha un andamento del tipo



LEGGE. (*Di Ahmdal*, o del *buonsenso*)
"Il miglioramento delle prestazioni di un sistema che si può ottenere ottimizzando una certa parte del sistema è limitato dalla frazione di tempo in cui tale parte è effettivamente utilizzata"

Ovvero: la parte di codice non parallelizzabile, penalizza tutto il programma. Abbiamo dei cosiddetti *"bottleneck"*

Problema: *non tutti gli algoritmi sono parallelizzabili!*

Definizione di Paralelizzabilità (e non)

Definizione: *Esecuzione di un algoritmo tramite più flussi simultanei*. Non tutti gli algoritmi sono parallelizzabili

Parallelizzabile: (esempi)

- Calcolare la somma di un vettore (array); posso spezzare l'array in *due*, farci le somme individuali poi sommare le ridotte.

Non Parallelizzabile: (esempi)

- Calcolare le cifre di $\sqrt{2}$; devo in un modo o l'altro usare i *metodi dell'analisi numerica*, che sono *iterativi* (o addirittura *ricorsive*...)

Attualità della Programmazione Parallela

Ancora oggi questo tema è *attuale*.

C'è *molta ricerca* per tentare di *parallelizzare* gli algoritmi

- Trovando *espedienti matematici*
- Oggi abbiamo sistemi con *tanti core*, e vogliamo sfruttarli al massimo
- Calcolando *soluzioni approssimate* (tipo per $\sqrt{2}$ posso usare gli *sviluppi di Taylor*)

Problema sentito nel *machine learning*

- Addestrare una rete neurale usando molti core (e nodi)
  - Problema risolto
- Algoritmi di *clustering* (classificazione) paralleli
  - Problema in parte aperto

I mutex

Vediamo un *primo costrutto* di sincronizzazione: i *mutex*.

Problema delle Sezioni Critiche

I thread condividono la memoria

- Possono condividere informazioni usando *Variabili Condivise*

E' necessario sincronizzare l'*accesso alle variabili condivise*

- Due thread non devono scrivervi contemporaneamente
- Un thread non deve leggere una variabile condivisa mentre un'altro la scrive

- Altrimenti avrei *casini*!
- Tema accennato con i *segnali*, mediante il *problema dell'incremento perso* (Segnali > ^45417b).

PROBLEMA.
Immaginiamo due thread che eseguono il seguente codice:

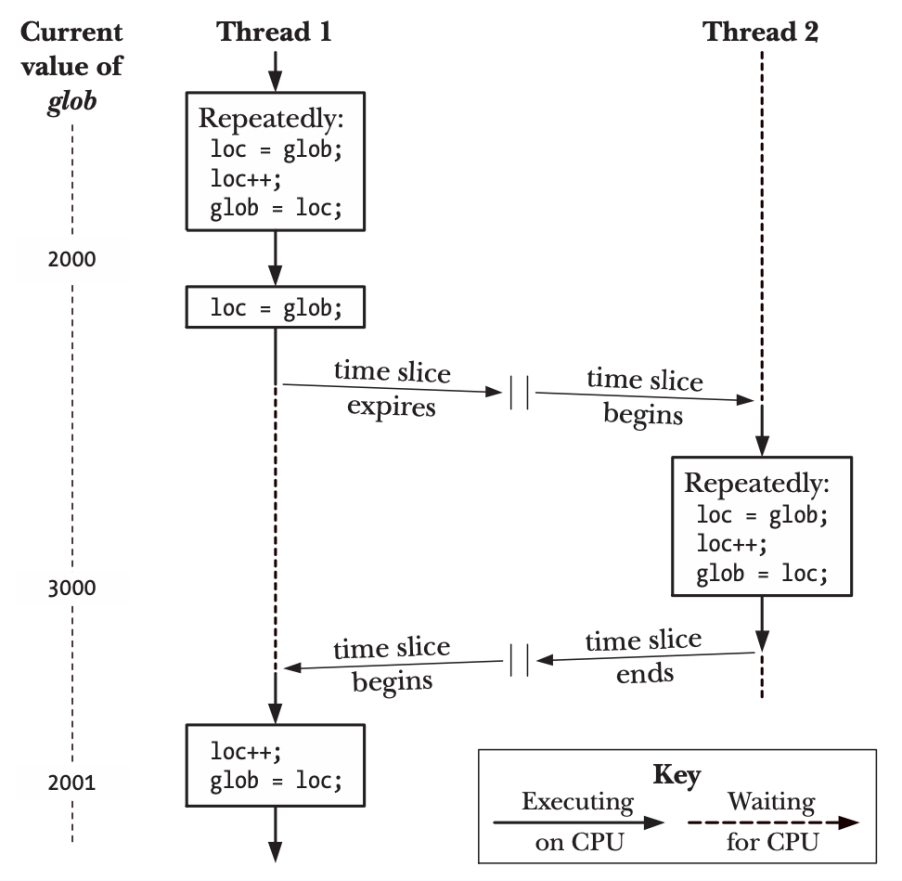
```c
static int glob = 0;
static void * threadFunc(void *arg){
    int loops = *((int *) arg);
    int loc, j;
    for (j = 0; j < loops; j++) {
        loc = glob;
        loc++;
        glob = loc;
    }
    return NULL;
}
```

Il seguente codice produce risultati non predicibili.

Esempio:

- Thread 1 è interrotto durante l'incremento
- Thread 2 effettua l'incremento
- Thread 1 completa l'incremento

L'incremento effettuato dal Thread 2 *è perso!* (o potenzialmente); ho uno *stato inconsistente del programma*. Avrò l'incremento perso circa al $\sim 50\%$.



Osservazioni

Sostituire:

```c
loc = glob;
loc++;
glob = loc;
```

con `glob++;` non risolve il problema.

In molti processori (e.g., ARM) non hanno una istruzione di incremento

- Il compilatore traduce `glob++;` in istruzioni Assembly equivalenti alle 3 righe di codice di cui sopra

Definizione di Sezione Critica

Definizione. (*Sezione critica*)

Una *Sezione Critica* è una sezione di codice la cui esecuzione deve essere *atomica* (nel senso autonoma)

- Non può essere *interrotta* da un altro thread
- Nessun altro thread può eseguire quel codice *contemporaneamente*

Una sezione critica accede a *risorse condivise*

- *Solo un thread per volta vi può fare accesso*

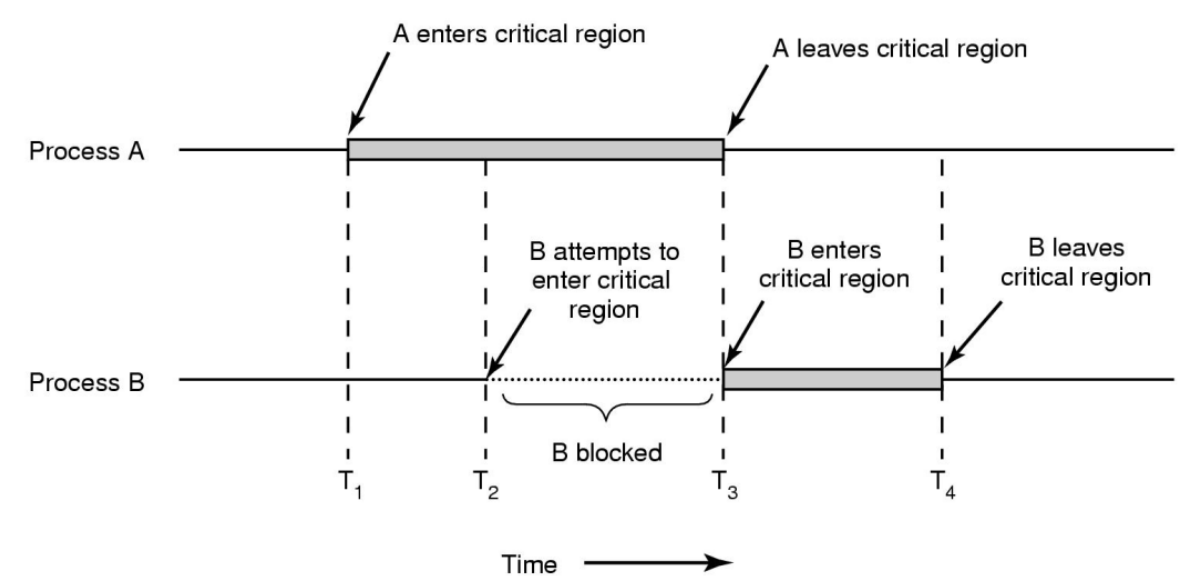
Le sezioni critiche sono anche dette *Regioni Critiche* (sinonimo).

Funzionamento di sezione critica

Prima di vedere il costrutto, vediamo come funzionerebbe (da un punto di vista teorico) una *sezione critica*

L'accesso a una sezione critica avviene in *Mutua Esclusione*

- Un thread si *prenota* per l'accesso
  - Se la sezione critica non è utilizzata, il thread vi accede (*lock*)
  - Altrimenti attende finchè non si libera
- Al termina della sezione critica, il thread *rilascia* la sezione (*post*)



Adesso siamo pronti per vedere il *mutex*.

Definizione di Mutex

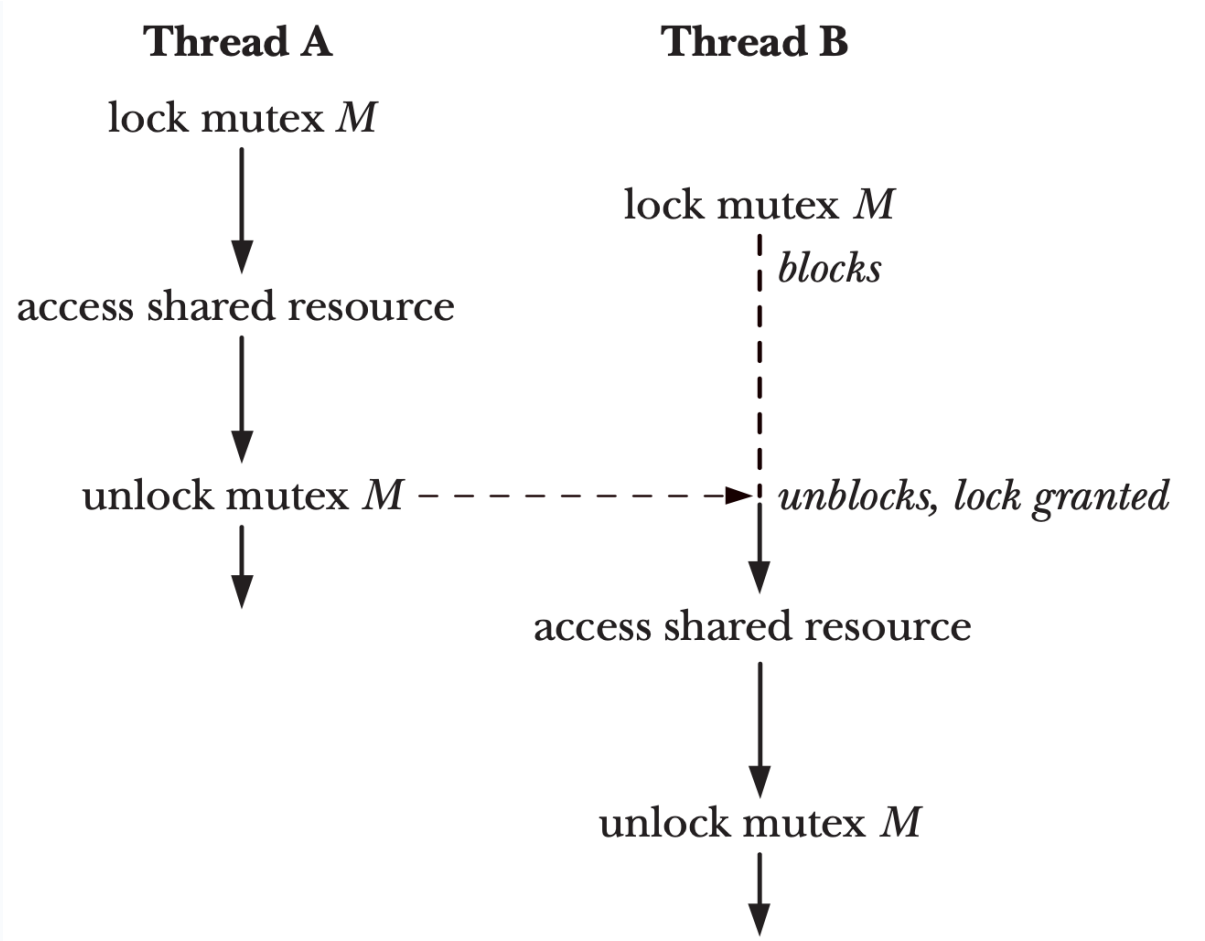
Un *Mutex* è un costrutto di sincronizzazione che gestisce l'accesso a una sezione critica

Un mutex ha due stati

- **Locked**: la sezione è occupata
- **Free**: la sezione è libera

Un thread può fare due azioni su un mutex:

- **Lock**: prenota l'accesso per l'occupazione della sezione critica
- **Release/Unlock**: rilascia la sezione critica



Implementazione in Pthread

I mutex sono variabili di tipo **pthread_mutex_t**

- Sono solitamente *variabili globali*
- Inizializzate dal **main**
- Usate da *qualsiasi thread*

Necessario includere:

```c
#include <pthread.h>
```

Si utilizzano con le funzioni di libreria **pthread_mutex_***

Inizializzazione dei Mutex

```c
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t * mutex , const pthread_mutexattr_t *
attr );
```

Inizializza il mutex `mutex`, che viene *passato per riferimento* (tipo `pthread_mutex_t *`)
L'argomento `attr` specifica gli attributi, che non vedremo

- Può essere `NULL`

Valore di ritorno, come in tutte le funzioni di Pthread (omesso nelle successive slide):

- 0 in caso di successo
- Il codice di errore altrimenti

Lock di Mutex

```c
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t * mutex );
```

Acquisisce il *lock* del mutex

- Blocca il chiamante finchè il lock non diventa libero

Release di Mutex

```c
#include <pthread.h>
int pthread_mutex_unlock(pthread_mutex_t * mutex );
```

Rilascia il lock

Nota: `mutex` è **sempre** passato per riferimento!

Altre Operazioni

```c
#include <pthread.h>
int pthread_mutex_trylock ( pthread_mutex_t *mutex);
```

Acquisisce il lock

- Se il lock è già preso da qualcun'altro fallisce con errore (valore di ritorno) `EBUSY`

Distruzione di Mutex

```c
#include <pthread.h>
int pthread_mutex_destroy ( pthread_mutex_t *mutex );
```

Rilascia la *memoria occupata* dal lock mutex
Tale lock non sarà più utilizzabile

Esempio

Realizzazione del precedente programma (incremento di una variabile da parte di due thread in parallelo) usando in mutex

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdlib.h>
#include <pthread.h>

static int glob = 0;
static pthread_mutex_t mtx;

static void * threadFunc(void *arg){
    int loops = *((int *) arg);
    int loc, j;
    for (j = 0; j < loops; j++) {
        pthread_mutex_lock(&mtx);      /*    LOCK          */
        loc = glob;                    /*  ┳               */
        loc++;                         /*  ┃ Critical Section */
        glob = loc;                    /*  ┻               */
        pthread_mutex_unlock(&mtx);    /*    RELEASE       */
    }
    return NULL;
}
int main(int argc, char *argv[]){
    pthread_t t1, t2;
    int loops = 10000000;

    pthread_mutex_init(&mtx, NULL);
    pthread_create(&t1, NULL, threadFunc, &loops);
    pthread_create(&t2, NULL, threadFunc, &loops);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_mutex_destroy(&mtx);
    printf("glob = %d\n", glob);
    exit(0);
}
```

Il programma senza l'uso di mutex:

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

static int glob = 0;

static void * threadFunc(void *arg){
    int loops = *((int *) arg);
    int loc, j;
    for (j = 0; j < loops; j++) {
        loc = glob;              /* ┳              */
        loc++;                   /* ┃ Critical Section */
        glob = loc;              /* ┻              */
    }
    return NULL;
}

int main(int argc, char *argv[]){
    pthread_t t1, t2;
    int loops = 10000000;
    pthread_create(&t1, NULL, threadFunc, &loops);
    pthread_create(&t2, NULL, threadFunc, &loops);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("glob = %d\n", glob);
    exit(0);
}
```

La somma non è correttamente 20000000, ma un numero inferiore (e.g., 10493368)

Deadlock

Un *Deadlock* o stallo è una situazione in cui due o più thread risultano bloccati

- *Ognuno attende una condizione che non potrà mai verificarsi*
- Il programma cessa di eseguire

Quando si usano due o più mutex possono capitare situazioni di questo tipo

- Necessario che il programmatore le preveda e le eviti

Esempio Analogico: Vado in segreteria segreteria per chiedere qualcosa relativo al bando ERASMUS+; la segreteria mi manda all'ufficio internazionale per le informazioni. L'ufficio internazionale mi manda alla segreteria studenti (oppure Banana Joe).

Esempio:
Thread A:

```c
pthread_mutex_lock(mutex1); // <-- LOCK 1
pthread_mutex_lock(mutex2); // <-- LOCK 2
... Sezione Critica ...
pthread_mutex_unlock(mutex2);
pthread_mutex_unlock(mutex1);
```

Thread B:

```c
pthread_mutex_lock(mutex2); // <-- LOCK 2
pthread_mutex_lock(mutex1); // <-- LOCK 1
... Sezione Critica ...
pthread_mutex_unlock(mutex1);
pthread_mutex_unlock(mutex2);
```

Come evitare i deadlock:

- Usare altri *tipi di sincronizzazione* quando possibile:
  - Pipe, FIFO
- Usare un *basso numero di mutex*
- *Modellare l'uso di tanti mutex* con espedienti matematici
  - Tecniche basate sui grafi
  - Non vediamo in questo corso
- Usare il buonsenso!

I Semafori

Definizione di Semaforo

Definizione (*Semaforo*)
Un *Semaforo* è un numero **Intero Positivo** condiviso da più thread

- Inizializzato a un certo valore in fase di creazione

Thread concorrenti (in realtà anche processi) possono fare due operazioni:

- *Incremento di* 1
- *Decremento di* 1

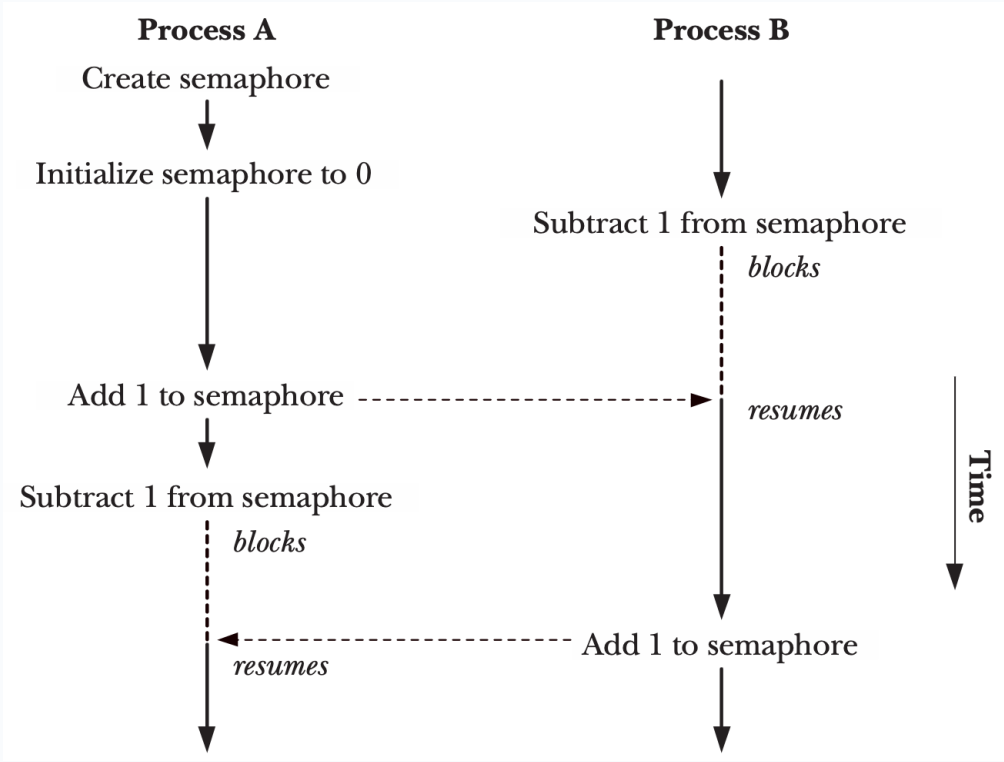
Il semaforo non può *mai* assumere *valori negativi*.
Se il decremento comporta che il semaforo diventi negativo, allora

- Il thread si *blocca*, *attendendo* che un altro thread faccia un incremento

Un *semaforo* è come un *secchio* con dei *gettoni*: o *metto* dei gettoni, o provo a *toglierli*.
Se provo a togliere un secchio senza gettoni, aspetto che il prossimo ne prenda uno.

Esempio
Supponiamo di avere *due processi* e *un semaforo*

1. Il *semaforo* è inizializzato a 0
2. B decrementa
    - Il semaforo non può asusmere valori negativi
    - B entra in attesa
3. A incrementa
    - B si sblocca
    - Il semaforo ha valore 0
4. A decrementa
    - A si blocca
5. B incrementa
    - A si sblocca
6. Il semaforo ha valore 0



Storia dei Semafori

Sono un costrutto di sincronizzazione semplice, potente e flessibile

- Inventato da *Dijkstra* nel 1965
- Usato per svariati scopi in tutti i linguaggi di programmazione e sistemi operativi

In Linux, due implementazioni

- *System V semaphores*: più vecchi, complessi. Non li vedremo
- *POSIX semaphores*: li vedremo

NOTA: possono essere usati anche tra processi diversi (e non solo tra thread di uno stesso processo)

Tipologie di Semafori

I *POSIX semaphores* possono essere:

- *Named*: hanno un nome univoco. Possono essere usati da più processi indipendenti (anche senza relazioni di parentela)
  - Il più pratico
- *Unnamed*: non hanno nome. Possono essere condivisi tra:
  - Thread, senza particolari accorgimenti (l'unico caso in cui diventa pratico)
  - Processi: se creati tramite `fork` e risiedono in una zona di memoria condivisa (con `shmget` o `mmap`) (il caso meno pratico, anche se possibile)

Il principio di funzionamento è lo stesso:

1. Il semaforo viene creato/inizializzato
2. I processi/thread possono effettuare delle:
   - *Post* per incrementare il semaforo
   - *Wait* per decrementare il semaforo (ed eventualmente attendere)
3. Il semaforo viene distrutto/chiuso

Named Semaphores

Si utilizzano le seguenti funzioni:

1. `sem_open()`
2. `sem_post(sem)`, `sem_wait(sem)` e `sem_getvalue()`
3. `sem_close()` e `sem_unlink()`

Necessario includere l'header:

```c
#include <semaphore.h>
```

I semafori sono handle opachi di tipo:

```
sem_t
```

1. *Creazione*

```c
#include <fcntl.h> /* Defines O_* constants */
#include <sys/stat.h> /* Defines mode constants */
#include <semaphore.h>

sem_t *sem_open(const char * name , int oflag , ...
            /* mode_t mode , unsigned int value */ );
```

Argomenti obbligatori:
Crea un semaforo dal nome **name**

- Deve iniziare con **/**
- Può essere un qualsiasi identificativo
  Esempio: **/mysem**
- L'argomento **oflag** specifica cosa fare *se il semaforo esiste* o no:
  - **O_CREAT**: crea e apre se non esiste. Apre se esiste
  - **O_CREAT | O_EXCL**: crea e apre. Fallisce se già esiste

Argomenti opzionali:

- **value** specifica il valore iniziale
- **mode** specifica i permessi, come per i file

Se si usa il flag **O_CREAT**, **value** vanno specificati!

Valore di ritorno: il semaforo in caso di successo, se no **SEM_FAILED**

2. *Chiusura e distruzione*

```c
#include <semaphore.h>
int sem_close(sem_t * sem );
int sem_unlink(const char * name );
```

sem_close chiude il semaforo per il processo corrente
sem_unlink rimuove il semaforo per tutti i processi

Valore di ritorno: 0 in caso di successo, se no -1

3. *Incrementa/Decrementa*

```c
#include <semaphore.h>
int sem_wait(sem_t * sem );
int sem_post(sem_t * sem );
```

sem_wait decrementa di 1 il semaforo

- Se il semaforo dovesse assumere valori negativi, blocca il chiamante

sem_post incrementa di 1 il semaforo

Valore di ritorno: 0 in caso di successo, se no -1

3. *Operazioni particolari*

```c
#include <semaphore.h>
int sem_trywait(sem_t *sem);
int sem_getvalue(sem_t *restrict sem, int *restrict sval);
```

sem_trywait come la **sem_wait**

- Ma non blocca in caso il semaforo vada in negativo
- Ma fallisce

sem_getvalue colloca nell'intero puntato da **sval** il valore del semaforo

Esempio di Semafori Non Anonimi

Si creino due programmi che comunicano tramite un semaforo.

- Il primo effetua una **post** ogni volta che l'utente preme *Enter*
- Il secondo stampa una stringa ogni volta che il primo effettua una **post**

Programma 1

```c
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <errno.h>
#include <semaphore.h>
#include <string.h>

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

    s = sem_open("/semaforo", O_CREAT , S_IRUSR | S_IWUSR, 0);
    if(s == SEM_FAILED) {
        printf("Error creating/opening the semaphore %s\n", strerror(errno));
        exit (1);
    }

    while(1){
        printf("Premi enter per una post: ");
        getchar();
        sem_post(s);
    }
    sem_close(s); /* Codice irraggiungibile*/
    return 0;
}
```

Programma 2

```c
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <errno.h>
#include <semaphore.h>
#include <string.h>

int main(int argc, char *argv[]){
    sem_t * s;
    int i = 0;

    s = sem_open("/semaforo", O_CREAT , S_IRUSR | S_IWUSR, 0);
    if(s == SEM_FAILED) {
        printf("Error creating/opening the semaphore %s\n", strerror(errno));
        exit (1);
    }

    while(1){
        sem_wait(s);
        printf("Wait %d effettuata\n", i);
        i++;
    }
    sem_close(s); /* Codice irraggiungibile*/
    return 0;
}
```

Osservazioni:

- Il valore del semaforo è persistente. Se Programma 2 non viene eseguito, il semaforo può crescere di valore
- Si possono eseguire più istanze di entrambi i programmi
    - Più istanze di Programma 1 accumulano valore nel semaforo
    - Se ci sono più istanze di Programma 2, solo una può essere sbloccata per ogni incremento
    - Il sistema operativo tendezialmente è *fair*. Fa load balancing tra più semafori in attesa

Unnamed semaphores

Si utilizzano in maniera simile, ma più semplice rispetto ai *Named Semaphores*
Diversa procedure di aperture chiusura

1. *Creazione*

```c
#include <semaphore.h>
int sem_init(sem_t * sem , int pshared , unsigned int value );
```

Crea il semaforo e lo colloca in **sem**, inizializzato a **value**

Importante:
sem_open ritorna un puntatore a semaforo (**sem_t ***), che viene allocato dalla libreria
sem_init colloca il puntatore a semaforo in **sem**

- Il programmatore deve devidere dove allocare il semaforo, di tipo **sem_t**
- Può esser una variabile globale, locale, allocata dinamicamente o su una regione di memoria condivisa

Argomenti obbligatori
Se **pshared** è 0, il semaforo non viene condiviso tra processi, ma solo tra thread

- **sem** può essere una comune variabile globale

Se **pshared** è $\neq 0$, il semaforo viene condiviso tra processi (tramite **fork**)

- **sem** deve essere in una zona di memoria condivisa

Conseguenza: meglio usare Named Semaphore con applicazioni multi-processo

2. *Distruzione*

```c
#include <semaphore.h>
int sem_destroy(sem_t * sem );
```

Distrugge il semaforo **sem**.
Se esso è condiviso tra processi, tutti i processi devono invocare **sem_destroy**

Nota: **sem_close** e **sem_unlink** sono usato solo coi *Named Semaphores*

3. *Incremento/Decremento*
   Si usano **sem_post()** e **sem_wait()** come per i *Named Semaphores*

Unnamed semaphores - Esempio

Si crei un programma con due thread. Il primo ogni secondo manda un messaggio al secondo, usando una variabile globale condivisa (di tipo **char[]**).

Il secondo lo stampa.

Struttura del programma:

```c
#include <stdio.h>
#include <unistd.h>
#include <semaphore.h>
#include <pthread.h>

sem_t s_scrittura, s_lettura;     /* Due semafori */
char buffer [50]; /* Buffer condiviso tra Thread */

void * sender(void *arg){
    ...
}

void receiver(){
    ...
}

int main(int argc, char *argv[]){
    pthread_t t;
    sem_init(&s_scrittura, 0, 0);
    sem_init(&s_lettura, 0, 1);
    pthread_create(&t, NULL, sender, NULL); /* Thread creato per sender */
    receiver();                             /* Il Main fa da receiver */
}
```

Logica del programma:
Bisogna evitare che un thread legga mentre un altro scrive

- Si potrebbe leggere una stringa in stato inconsistente!
- Senza terminatore!

Servono due semafori:

- **`s_scrittura`** notifica che **`sender`** ha terminato una scrittura
  - **`sender`** mette un *gettone* quando finisce la scrittura, **`receiver`** attende il gettone per iniziare la lettura
- **`s_lettura`** notifica che **`receiver`** ha terminato la lettura
  - **`receiver`** mette un *gettone* quando finisce la lettura, **`sender`** attende il gettone per iniziare la nuova scrittura

`s_scrittura` deve essere inizializzato a 0 perchè `receiver` aspetti la prima scrittura
`s_lettura` deve essere inizializzato a 1 perchè `sender` possa fare la prima scrittura

Sender:

1. `sem_wait(s_lettura)`: per essere sicuro che `receiver` abbia terminato la lettura
2. Scrive su `buffer`
3. `sem_wait(s_scrittura)`: per notificare termine scrittura

Receiver:

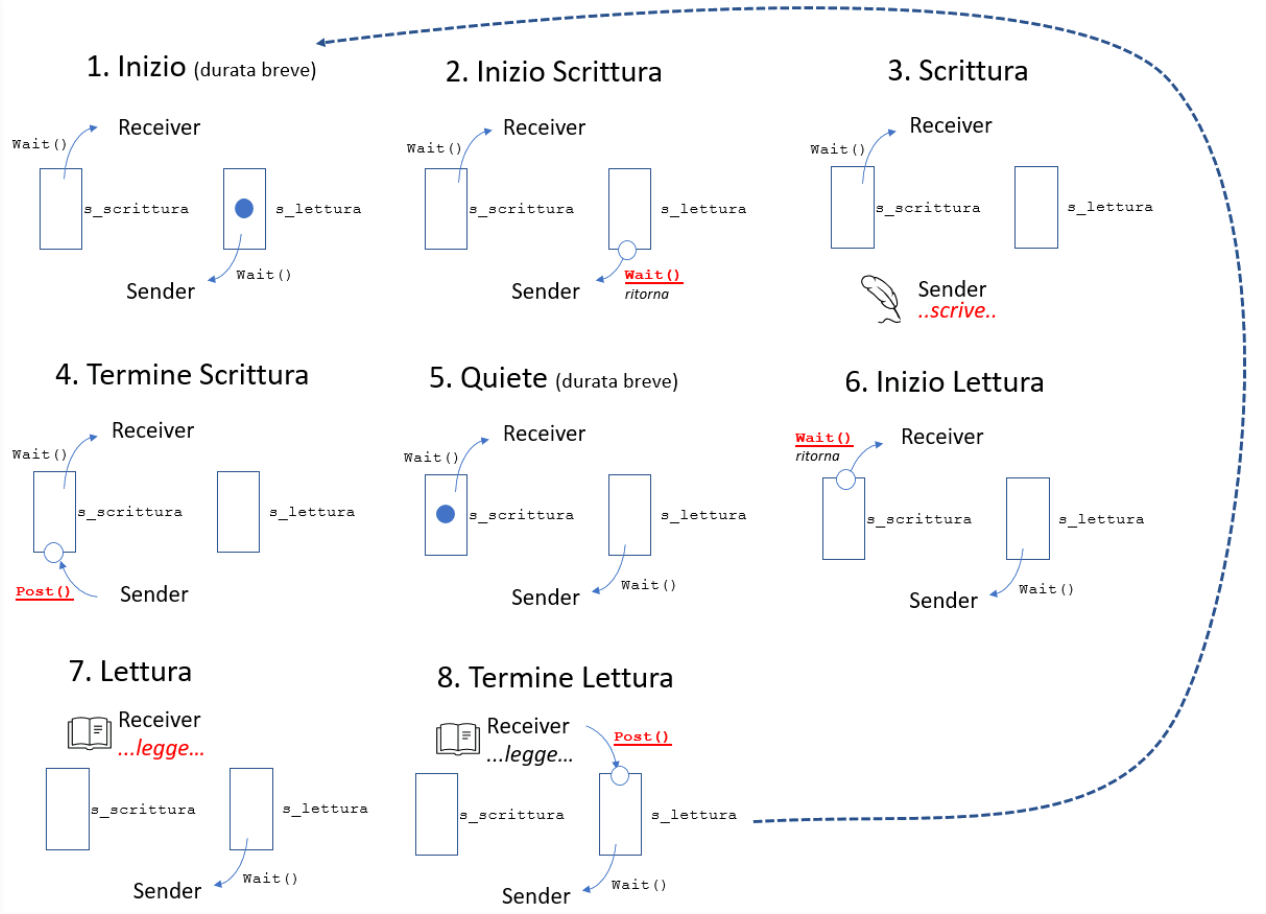
1. `sem_wait(s_scrittura)`: per essere sicuro che `sender` abbia terminato la scrittura
2. Legge su `buffer`
3. `sem_post(s_lettura)`: per notificare termine lettura

Sender e Receiver:

```c
void * sender(void *arg){
    int i = 0;
    while (1){
        sem_wait(&s_lettura);
        sprintf(buffer, "Message %d\n", i);
        sem_post(&s_scrittura);
        i++;
        sleep(1);
    }
}

void receiver(){
    while (1){
        sem_wait(&s_scrittura);
        printf("Received: %s\n", buffer);
        sem_post(&s_lettura);
    }
}

...
sem_init(&s_scrittura, 0, 0);
sem_init(&s_lettura, 0, 1);
```



Domande

La parallelizzazione è una soluzione per migliorare le prestazioni:

- **di qualsiasi algoritmo**
- **solo di algoritmi che accedono al disco**
- **solo di algoritmi che posono eseguiti per mezzo di più flussi contemporanei**

Risposta: *Solo di algoritmi che possono eseguiti per mezzo di più flussi contemporanei*

Il seguente codice è corretto?

```c
pthread_mutex_lock(&mtx);
var++;
pthread_mutex_lock(&mtx);
```

- **Si, il lock viene rilasciato**
- **No, il thread entra in uno stato di attesa perpetuo**

Risposta: *No, il thread entra in uno stato di attesa perpetuo*

Un semaforo può essere inizializzato:

- **A qualsiasi valore intero**

- **A qualsiasi intero non negativo**
- **A qualsiasi intero positivo**

Risposta: *A qualsiasi interno non negativo*

Un programma esegue il seguente codice:

```c
sem_init(&s, 0, 0);
for (i = 0; i<10; i++){
   sem_wait(&s);
   sem_post(&s);
}
```

Al termine del programma che valore assume il semaforo?

- **0**
- **10**
- **Il programma non termina perché entra in uno stato di attesa perpetuo**

Risposta: *Il programma non termina perché entra in uno stato di attesa perpetuo*

Si immaginino due thread di un processo che operano su semaforo `s` inizializzato a 1.
Il Thread 1 esegue:

```c
void * t1(void *arg){
   sem_post(&s);
   sem_post(&s);
}
```

Il Thread 2 esegue:

```c
void * t2(void *arg){
   sem_wait(&s);
   sem_wait(&s);
   sem_wait(&s);
   sem_post(&s);
}
```

Il programma:

- **Termina**

• **Entra in uno stato di attesa indefinito**

Risposta: *Termina*

u7-s3-sync-problems

Sistemi Operativi

Unità 7: I Thread

Problemi di Sincronizzazione

Martino Trevisan
Università di Trieste
Dipartimento di Ingegneria e Architettura

Argomenti

1. Mutex e Semafori
2. Grafi di precedenza
3. Produttore e consumatore

Mutex e Semafori

I **Mutex** regolano l'accesso a una sezione critica:

- Solo un thread per volta può avere il lock
- Operazioni: **lock** **unlock**

I **Semafori** sono degli interi positivi condivisi:

- Simili a un contenitore di gettoni
- Operazioni: **post** **wait**

I *Semafori* sono un costrutto più generale

- Un *Semaforo* può facilmente essere usato come *mutex*. Non vale il contrario!

Costruzione di Mutex con Semaforo

Inizializzazione:
Mutex

```c
pthread_mutex_t lock;
pthread_mutex_init(&lock, NULL);
```

Semaforo: deve essere inizializzato al valore 1

```c
sem_t sem;
sem_init(&sem, 0, 1);
```

Lock:
Mutex

```c
pthread_mutex_lock(&lock);
```

Semaforo

```c
sem_wait(&sem);
```

Release:
Mutex

```c
pthread_mutex_unlock(&lock);
```

Semaforo

```c
sem_post(&sem);
```

Per implementazione completa, vedi implementazione in **esercizi/myMutex.c**
myMutex.c

Idea. (implementazione di lock, unlock)

```c
typedef struct{
    sem_t s;
} myMutex;

myMutex myMutex_init(){
    myMutex m;
    sem_init(&(m.s), 0, 1);
    return m;
}

void myMutex_lock(myMutex * m){
    sem_wait( &(m->s) );
}

void myMutex_unlock(myMutex * m){
    sem_post( &(m->s) );
}
```

Costruzione di Semafori con Mutex

Si può costruire un semaforo con un **mutex**, ma é *inefficiente*

- Un semaforo é un intero condiviso *positivo*
- Un mutex protegge l'accesso a questo intero

Funzionamento:

- In caso venga effettuato un decremento (**wait**) quando il semaforo ha valore 0:
  Il thread attende che un altro thread effettui un incremento (**post**)
- L'unico modo con cui si attendere, é *busy waiting*
  - Un ciclo **for** che verifica ripetutamente
  - Inefficiente

Implementazione (by ChatGPT; se neanche il prof. ha voluto fare...):

```c
struct semaphore {
    pthread_mutex_t mutex;
    int count;
};

void semaphore_init(struct semaphore *sem, int count) {
    pthread_mutex_init(&sem->mutex, NULL);
    sem->count = count;
}

void semaphore_wait(struct semaphore *sem) {
    pthread_mutex_lock(&sem->mutex);
    while (sem->count == 0) {
        pthread_mutex_unlock(&sem->mutex);
        pthread_mutex_lock(&sem->mutex);
    }
    sem->count--;
    pthread_mutex_unlock(&sem->mutex);
}

void semaphore_post(struct semaphore *sem) {
    pthread_mutex_lock(&sem->mutex);
    sem->count++;
    pthread_mutex_unlock(&sem->mutex);
}
```

Grafi di precedenza

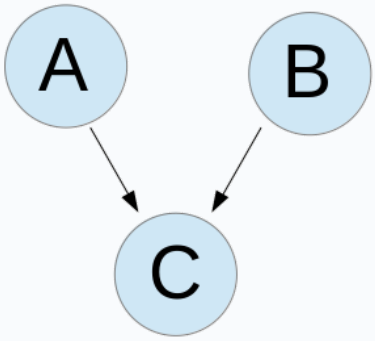
I semafori sono pratici da usare per costruire *grafi di precedenza*

- Un insieme di task che devono essere eseguite in un ordine particolare

I grafi di precedenza modellano molto bene *sistemi distribuiti e concorrenti*

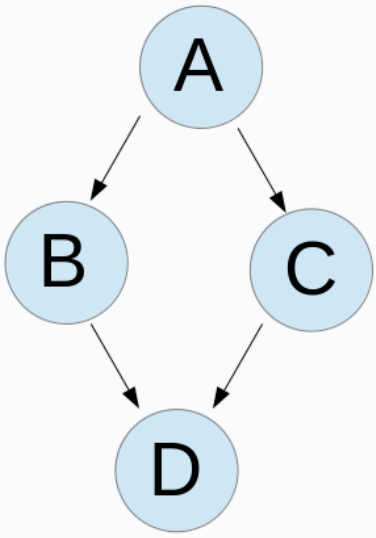
- Le *Reti di Petri* sono un astrazione per trattare grafi di precedenza con l'utilizzo di semafori
- Non vedremo

Esempio 1



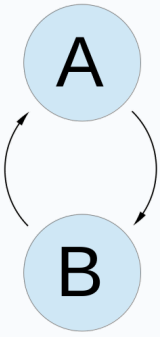

```c
sem_t s1;
void* t_A(void* arg){
    A();
    sem_post(&s1);
}
void* t_B(void* arg){
    B();
    sem_post(&s1);
}
void* t_C(void* arg){
    sem_wait(&s1);
    sem_wait(&s1);
    C();
}
```

Esempio 2



```c
sem_t s1, s2;
void* t_A(void* arg){
    A();
    sem_post(&s1);
    sem_post(&s1);
}
void* t_B(void* arg){
    sem_wait(&s1);
    B();
    sem_post(&s2);
}
void* t_C(void* arg){
    sem_wait(&s1);
    C();
    sem_post(&s2);
}
void* t_D(void* arg){
    sem_wait(&s2);
    sem_wait(&s2);
    D();
}
```

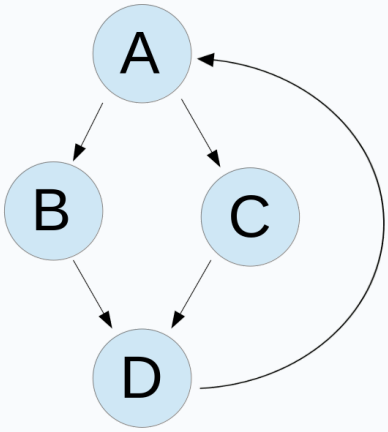
Esempio 3: Grafo ciclico



```c
sem_t s1, s2;
sem_init(&s1, 0, 1); // Inizializzato a 1
sem_init(&s2, 0, 0); // Inizializzato a 0
void* t_A(void* arg){
    while (1){
        sem_wait(&s1);
        A();
        sem_post(&s2);
    }
}
void* t_B(void* arg){
    while (1){
        sem_wait(&s2);
        B();
        sem_post(&s1);
    }
}
```

NOTA: esercizio uguale a lettore/scrittore visto in precedenza

Esempio 4



```c
sem_t s1, s2, s3; // s1 inizializzata a 1, gli altri a 0
void* t_A(void* arg){
    while (1){
        sem_wait(&s1);
        A();
        sem_post(&s2);
        sem_post(&s2);
    }
}
void* t_B(void* arg){
    while (1){
        sem_wait(&s2);
        B();
        sem_post(&s3);
    }
}
void* t_C(void* arg){
    while (1){
        sem_wait(&s2);
        C();
        sem_post(&s3);
    }
}
void* t_D(void* arg){
    while (1){
        sem_wait(&s3);
        sem_wait(&s3);
        D();
        sem_post(&s1);
    }
}
```

Produttore e consumatore

Vediamo un *problema classico* dell'informatica.

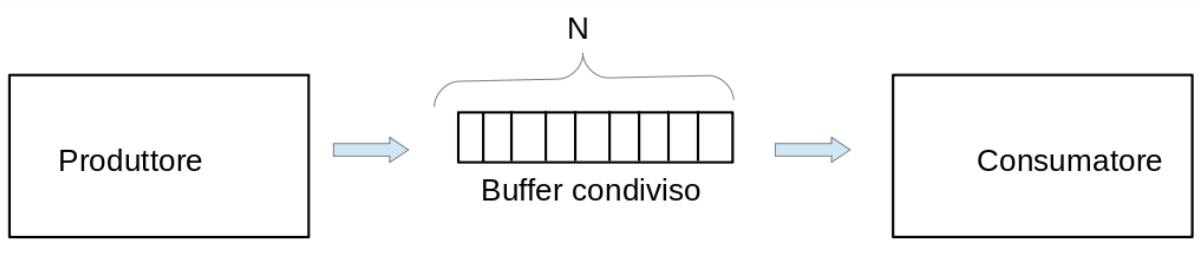
Problema. (*Produttore e consumatore*)
Problema classico dell'informatica, applicabile in molti contesti

- Pacchetti di rete

- Calcolo parallelo

Definizione:

- Due thread comunicano tramite *un buffer di grandezza limitata*, che contiene massimo N oggetti
  - Il thread *producer* inserisce gli oggetti nel buffer
  - Il thread *consumer* estrae gli oggetti dal buffer, nell'ordine in cui sono stati inseriti



Soluzione non-concorrente

Variabili Condivise tra Produttore e Consumatore:

```c
<tipo> buffer [N]; // Il buffer
int contatore = 0; // Indicazione di elementi usati nel buffer
```

Variabili NON Condivise:

```c
int in;  // Indice dove il produttore inserisce in buffer
         // Gestito in artitmetica Modulo N
int out; // Indice dove il consumatore estrare
```

Produttore:

```c
while (1) {
   while (contatore == BUFFER_SIZE); /* non fa niente se il buffer è pieno */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
   contatore++;
}
```

Consumatore:

```c
while (1) {
    while (contatore == 0); /* non fa niente se il buffer è vuoto */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    contatore--;
}
```

Questa
Il codice della slide precedente non funziona.

- C'è accesso *concorrente a variabili condivise*
  Le istruzioni **`contatore++;`** e **`contatore--;`** non possono essere eseguite simultaneamente
- Alcuni incrementi o decrementi potrebbero essere persi
- Il programma ha un *baco!*
- Il programma non è *thread safe*.

Prima Soluzione Concorrente

1. Accesso concorrente a `contatore`: è possibile usare un *mutex*
Nota: non c'è mai accesso concorrente a stesso elemento di **`buffer`**

- Tuttavia le istruzioni **`while (contatore == BUFFER_SIZE);`** e **`while (contatore == 0);`** effettuano *Busy Waiting*
  - Controlla continuamente la variabile **`contatore`**
  - Spreco enorme di CPU!

Soluzione Classica

Si usano due semafori

- Semaforo **`empty`**: conta quanti posti *liberi* ci sono nel buffer
- Semaforo **`full`**: conta quanti posti *occupati* ci sono nel buffer

La variabile **`contatore`** diventa *inutile*. I semafori già contano quanti posti liberi e occupati ci sono

Soluzione completa nel *materiale* in **`esercizi/myProdCons.c`** (myProdCons.c)

Inizializzazione

```c
<tipo> buffer [N];
sem_t empty, full;

int main(){
    ...
    sem_init(&empty, 0, N); /* Inizialmente N posti liberi */
    sem_init(&full, 0, 0);  /* e 0 occupati */
    ...
}
```

Produttore

```c
int in = 0;
while (1) {
    sem_wait(&empty); /* Attende che ci posto libero nel buffer */
    buffer[in] = next_produced;
    in = (in + 1) % N;
    sem_post(&full); /* Un dato un più nel buffer */
}
```

Consumatore

```c
int out = 0;
while (1) {
    sem_wait(&full); /* Attende che ci siano dati da consumare */
    <type> next_consumed = buffer[out];
    out = (out + 1) % N;
    sem_post(&empty); /* Un posto libero in più nel buffer */
}
```