

01 - THREAD E PROGRAMMAZIONE CONCORRENTE

PROCESSI E PARALLELISMO

slide 1

Il sistema operativo

Il sistema operativo è un **insieme di programmi** (moduli software) che svolgono **funzioni di servizio** nel calcolatore → costituisce la parte essenziale del cosiddetto **software di sistema** (o di base) in quanto non svolge funzioni applicative, ma ne costituisce un supporto

Lo **scopo** del sistema operativo è quello di:

- mettere a disposizione dell'utente una **macchina virtuale** in grado di eseguire comandi dati dall'utente, utilizzando una macchina reale di livello inferiore
- mettere a disposizione del software applicativo (e quindi del programmatore) un insieme di **servizi di sistema, invocabili tramite chiamate di sistema** (system call)
- **controllare** le **risorse fisiche** del sistema e **creare/gestire il parallelismo**

S.O. come creatore di parallelismo

- Nel modello di **esecuzione sequenziale**, l'esecuzione di N programmi avviene in sequenza, attivando l'i-mo programma solo dopo che è stata terminata l'esecuzione dell'(i-1)-mo programma
 - Questo garantisce che non possono esistere interferenze nell'esecuzione dei vari programmi.

esecuzione sequenziale vs esigenze dei sistemi di calcolo

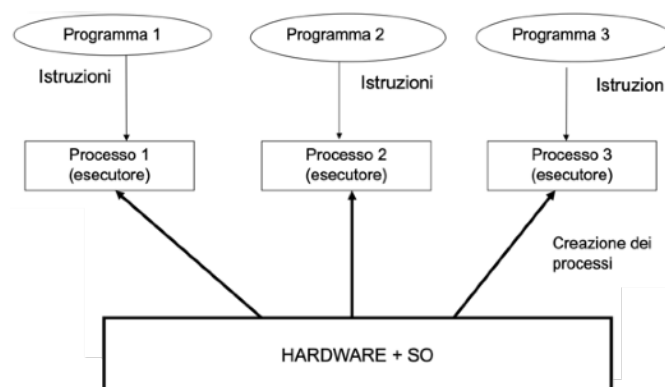
Necessità di parallelismo

- applicazioni distribuite
- calcolatore multiutente
- applicazioni multiple del singolo utente

Modello di esecuzione parallela

- Garanzia di soddisfacimento di due obiettivi contrastanti
 - **Parallelismo**
 - **Virtualizzazione del sistema di calcolo (esecuzione senza interferenze)**
- Gli obiettivi vengono soddisfatti tramite la **creazione dinamica** di tanti **esecutori** quanti sono i programmi da eseguire in parallelo
 - Gli esecutori creati dinamicamente vengono chiamati **processi**
 - Un processo è un esecutore completo
- Il sistema operativo è in grado di creare processi indipendenti e mette a disposizione del programmatore delle **System Call** che permettono di creare e eliminare i processi

Programmi e processi



Virtualizzazione delle risorse di calcolo

- Il sistema operativo gestisce eventuali **conflitti di accesso contemporaneo** alle risorse "reali" condivise, creando delle risorse **virtuali** e accodando i processi richiedenti
- **Accesso a periferiche condivise da più processi**
 - il codice eseguibile di un programma non può contenere istruzioni che accedano direttamente a periferica ma deve utilizzare delle system call che attivano opportuni servizi di sistema (v. implementazione delle librerie di Ingresso/Uscita del C)

Programmazione di sistema e chiamate di sistema (system call)

- Con **programmazione di sistema** si intendono le tecniche utilizzate nella scrittura di programmi utente che interagiscono strettamente con il sistema operativo e che utilizzano i servizi (**system call**) messi a disposizione da quest'ultimo
- Le **chiamate di sistema** sono divise nelle seguenti principali categorie:
 - gestione dei processi
 - segnalazioni
 - gestione di file
 - gestione di direttori e di file system
 - protezione
 - gestione di ora e data

LA PROGRAMMAZIONE DI SISTEMA: PRIMITIVE PER LA GESTIONE DEI PROCESSI

Aspetti generali relativi ai processi

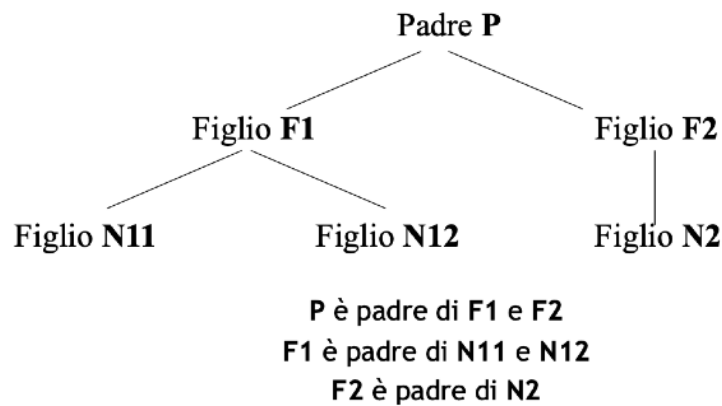
- **Un processo è un programma in esecuzione**
- Ogni processo è identificato in **modo univoco** da un **PID (Process Identifier)** cioè da un numero a 16-bit che viene assegnato sequenzialmente da Linux non appena un processo è stato creato.
- Tutti i processi sono creati da altri processi e quindi hanno un processo **padre** (unica eccezione: il processo "init", primo processo creato all'avviamento del SO, che **non** ha un processo padre).
- Dal punto di vista del programmatore di sistema, la memoria di lavoro associata ad un processo può essere vista come costituita da 3 segmenti:
 - **Segmento codice** (*text segment*): contiene il codice eseguibile del programma
 - **Segmento dati** (*user data segment*): contiene tutte le variabili del programma (globali o statiche, locali allocate su stack, create dinamicamente dal programma tramite `malloc()`)
 - **Segmento di sistema** (*system data segment*): contiene i dati non gestiti esplicitamente dal programma in esecuzione, ma dal S.O. (ad esempio la tabella dei file aperti).

Primitive (system call) per la gestione dei processi

- **Fork**: generare un processo figlio (*child* o anche *slave*) copia del processo padre (*parent* o anche *master*) in esecuzione
- **Exit**: terminare un processo figlio (restituendo un codice al processo *padre*)
- **Wait**: attendere la terminazione di un processo figlio
- **Exec**: sostituire il codice di un processo in esecuzione, cioè sostituire il programma eseguito da un processo

Gerarchia di processi

- Un processo padre P può generare uno o più processi figli F
- Un processo figlio F può a sua volta generare uno o più processi figli N



Generazione di processi: `fork()`

La funzione `fork`:

- Genera un processo **figlio (child)** identico al processo **padre (parent)**; il figlio è una **copia identica** del padre all'istante della `fork`
- **Duplicato il segmento dati e il segmento di sistema**, quindi le variabili, i file aperti, ecc. sono duplicati nel figlio.
 - All'istante della `fork` i segmenti padre e figlio contengono gli stessi valori (tranne che per il valore restituito dalla `fork` stessa). L'evoluzione indipendente dei due processi (può) modifica(re) ovviamente i segmenti.
- L'unica differenza tra figlio e padre è il **valore restituito dalla `fork`**:
 - **nel processo padre** → la funzione `fork` restituisce il *pid* del processo figlio appena generato (quindi il padre conosce il pid del figlio) → **restituisce -1 in caso di errore** (`fork` non eseguita)
 - **nel processo figlio** → la `fork` restituisce il valore 0

- prototipo `fork`:

```
pid_t fork(void)
```

dove `pid_t` è un tipo predefinito

Terminazione dei processi: `exit()`

- La funzione `exit` termina il processo corrente. Però l'esecuzione di un processo può terminare anche senza `exit`.
- Prototipo `exit`

```
void exit(int)
```

- Il **codice di terminazione della `exit` (exit code)** (cioè l'intero passato come parametro) viene "restituito" al padre
 - Per convenzione, un **exit code pari a 0** indica un'esecuzione corretta, mentre un valore **≠ 0** indica che è avvenuto un **errore** durante l'esecuzione
 - Se il processo che termina non ha più un processo padre (è già terminato) il valore viene restituito all'interprete comandi del S.O.

Esempio di uso di fork

```
#include <stdio.h>
#include <sys/types.h>

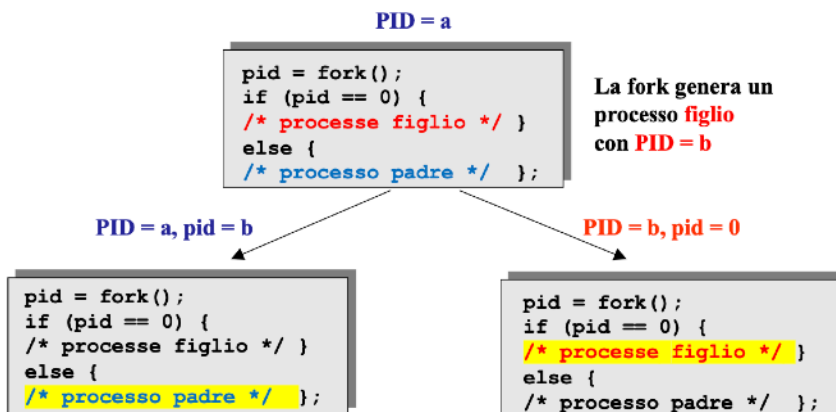
void main (int argc, char * argv[]) {
    pid_t pid;
    pid = fork();

    if (pid==0) { /* processo figlio */
        printf ("sono il processo figlio\n");
        exit();
    }
    else { /* processo padre */
        printf ("sono il processo padre\n");
        exit();
    }
}
```

Dopo la fork, i due processi padre e figlio evolvono in parallelo ed eseguono la stessa porzione di codice scrivendo sullo stesso standard output.

L'ordine secondo il quale saranno eseguite le due printf sarà casuale: potrà essere eseguito per primo il processo padre o il processo figlio.

Costituisce un grave errore di programmazione ipotizzare un preciso ordine di esecuzione di due processi, perchè evolvono in parallelo.



Possibile sequenza di esecuzione:
sono il processo figlio
sono il processo padre

oppure

sono il processo padre
sono il processo figlio

getpid() e getppid()

La funzione `getpid()` consente ad un processo di conoscere il valore del proprio pid

Prototipo `getpid`:

```
pid_t getpid(void)
```

La funzione `getppid()` consente di ottenere il pid del processo padre (parent)

Prototipo `getppid`:

```
pid_t getppid(void)
```

Esempio di uso di fork e getpid

```
#include <stdio.h>
#include <sys/types.h>

void main () {
    pid_t pid;
    printf("prima della fork: PID = %d\n", getpid());
    pid = fork();

    if (pid==0) { /* processo figlio */
        printf("FIGLIO: PID = %d\n", getpid()); /* restituisce il PID del figlio */
        printf("FIGLIO: PID DEL PADRE = %d\n", getppid());
        exit();
    }
}
```

```

else { /* processo padre */
    printf("PADRE: PID = %d\n", getpid()); /* restituisce il PID del padre */
    printf("PADRE: PID DEL FIGLIO = %d\n", pid) /* nel padre il valore restituito
                                                dalla fork è il PID del figlio
                                                appena generato */

    exit();
}
}

```

Primo esempio di generazione di 2 figli

```

#include <stdio.h>
#include <sys/types.h>

void main() {
    pid_t pid1, pid2;

    pid1 = fork();

    if (pid ==0) { /* 1° processo figlio */
        1 printf ("FIGLIO 1: PID = %d\n", getpid());
        exit();
    }
    else { /* processo padre */
        pid2 = fork();

        if (pid ==0) { /* 2° processo figlio */
            2 printf ("FIGLIO 2: PID = %d\n", getpid());
            exit();
        }
        else { /* processo padre */
            printf ("PADRE: PID = %d\n", getpid());
            printf ("PADRE: PID DEL FIGLIO 1 = %d\n", pid1);
            printf ("PADRE: PID DEL FIGLIO 2 = %d\n", pid2);
            exit();
        }
    }
}

```

Secondo esempio di generazione di 2 figli

```

void main() {
    pid_t pid;

    pid = fork();

    if (pid==0) { /* 1° processo figlio */
        printf ("sono il primo figlio con pid: = %d\n", getpid());
        exit();
    }
    else { /* processo padre */
        printf("sono il processo padre\n");
        printf("ho creato un primo figlio con pid: = %d\n", pid);
        printf("il mio pid è: = %d\n", getpid());

        pid = fork();
    }
}

```

```

    if (pid==0) { /* 2° processo figlio */
        printf ("sono il secondo figlio con pid: = %d\n", getpid());
        exit();
    }
    else { /* processo padre */
        printf("sono il processo padre\n");
        printf ("ho creato un secondo figlio con pid: =%d\n", pid);
        exit();
    }
}
}

```

Attesa della terminazione di un figlio: wait()

La funzione `wait()`

- sospende l'esecuzione del processo che la esegue ed attende la terminazione di un qualsiasi processo figlio;
- se un figlio termina prima che il padre esegua la wait, l'esecuzione della wait nel padre terminerà istantaneamente.

Prototipo `wait`:

```
pid_t wait(int*)
```

- Il **valore restituito** dalla funzione wait (di tipo `pid_t`) è il PID del figlio terminato.
- Il parametro passato per indirizzo rappresenta la variabile che contiene il **codice di terminazione (exit code)** restituito dal figlio terminato e moltiplicato per 256 (8 bit più significativi).

esempio di wait-exit

```

void main() {
    pid_t pid;
    int stato_exit, stato_wait;

    pid = fork();

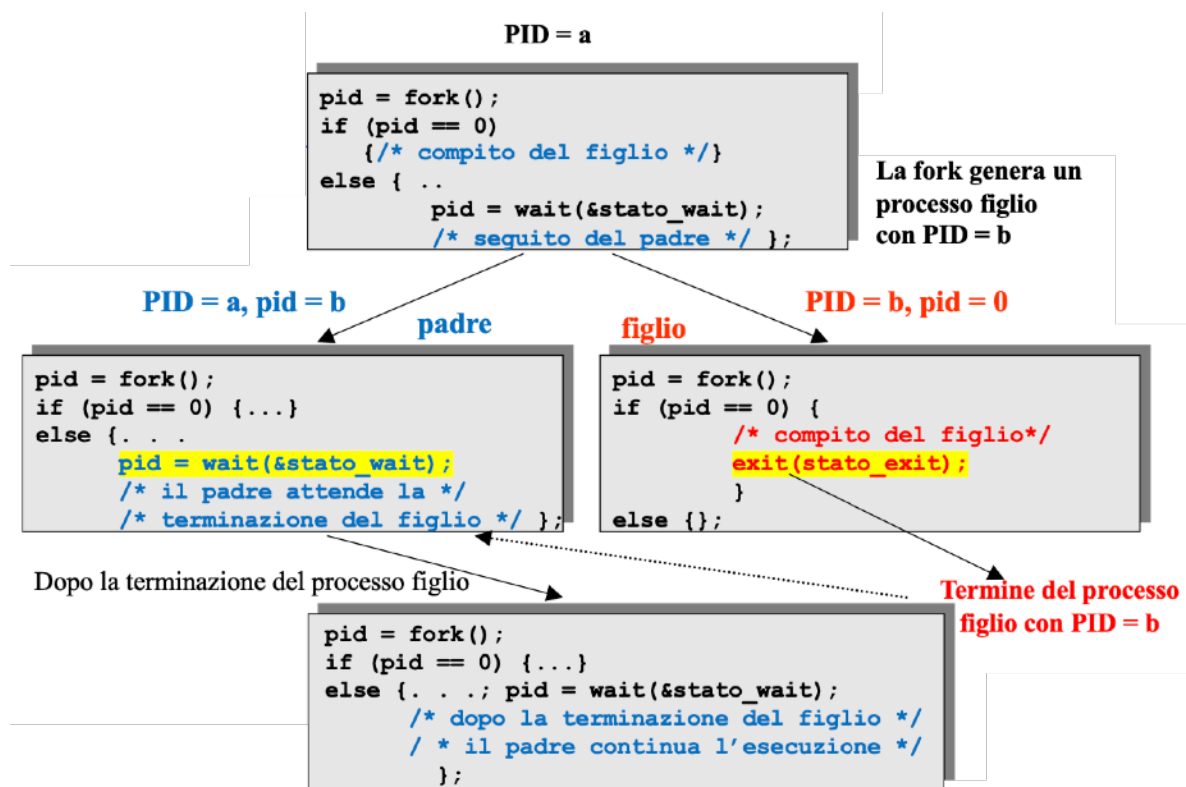
    if (pid==0) { /* processo figlio
        printf ("sono il processo figlio con pid: = %d\n", getpid());
        stato_exit = 5;
        exit(stato_exit); /* il figlio termina
    }
    else { /* processo padre */
        printf("Ho generato il processo figlio con pid %d\n", pid);

        pid = wait(stato_wait); /* il padre attende la terminazione del figlio*/

        /* dopo la terminazione del figlio, il padre continua l'esecuzione*/
        printf("terminato il processo figlio %d con esito %d\n", pid, stato_wait/256);
    }
}

```

Il valore restituito dalla funzione wait è il PID del figlio terminato



Processo zombie

- Se un processo figlio termina quando il padre ha chiamato la funzione `wait` → il processo figlio termina e il codice di uscita del figlio terminato (`exit code`) è passato al processo padre tramite la `wait`.
- Se un processo figlio termina quando il padre non ha ancora chiamato la `wait` → il processo figlio termina e diventa un **processo zombie**
- Quando il processo padre invoca la `wait` → lo stato di terminazione del figlio **zombie** viene recuperato, il processo figlio **zombie** viene terminato e la `wait` ritorna immediatamente.
- Un processo **zombie** è un processo terminato ma che non è stato ancora ripulito: il suo stato di terminazione può essere ancora recuperato dal processo padre.

Funzione `waitpid`

La funzione `waitpid`

- sospende l'esecuzione del processo padre che la esegue ed attende la terminazione del processo figlio di cui viene **fornito il pid**
- Se il figlio termina prima che il padre esegua la `waitpid`, l'esecuzione della `waitpid` nel padre terminerà istantaneamente

Prototipo `waitpid`

```
pid_t waitpid(pid_t pid, int *status, int options)
```

- il valore restituito assume il valore del pid del figlio terminato
- `status` assume il valore del codice di terminazione del processo figlio
- `options` specifica ulteriori opzioni (ipotizziamo > 0)

esempio di `waitpid`

```

void main() {
    pid_t pid, my_pi
    int status;

    pid = fork();

```

```

    if (pid==0) {      /* codice del figlio */
        ...
    }
    else {      /* codice del padre */
        printf("Ho generato il processo figlio con pid %d\n", pid);
        my_pid = waitpid(pid, &status, 1) /* il padre attende la terminazione del figlio*/

        /* dopo la terminazione del figlio, il padre continua l'esecuzione*/
        printf("è terminato il processo %d con esito %d\n", my_pid, status);
    }
}

```

Sostituzione del programma in esecuzione: exec

La funzione `exec`:

- **sostituisce un programma in esecuzione con un altro programma**
- quando un programma chiama la funzione `exec` → il processo rimane lo stesso ma termina immediatamente l'esecuzione del programma in esecuzione ed inizia ad eseguire il nuovo programma (commutazione di codice)
- **sostituisce il segmento codice e il segmento dati** del processo corrente con il codice e dati di un programma contenuto in un file eseguibile specificato
- il segmento di sistema non viene sostituito (file e pocket rimangono aperti e disponibili)
- il processo rimane lo stesso e quindi mantiene lo **stesso PID**
- la funzione `exec` passa dai parametri al programma che viene eseguito, tramite il meccanismo standard di passaggio dei parametri al main `argc` e `argv`

Sintassi `execl`

Sintassi:

```

int execl(char *path_programma, char *arg0, char *arg1,...char *argn);
/* restituisce int */

```

- `path_programma`: pathname completo del file eseguibile del nuovo programma da lanciare
- `arg0, arg1, ..., argn`: puntatori a stringhe che verranno passate come parametri al main del nuovo programma
- `arg0` deve essere il **nome** del programma
- `argn` in chiamata deve essere il valore NULL per indicare la fine dei parametri

Il valore restituito è

- 0 se l'operazione è stata eseguita correttamente
- -1 se c'è stato un errore e l'operazione di sostituzione del codice è fallita

Al momento dell'esecuzione del main del nuovo programma - `void main (int argc, char *argv[])` - `arg0, arg1 ...` vengono resi accessibili tramite l'array di puntatori `argv`

Passaggio di parametri

- `argc`: intero che contiene il numero dei parametri ricevuti
- `argv`: è un vettore di puntatori a stringhe, ognuna delle quali è un parametro. Per convenzione la stringa `argv[0]` contiene sempre il nome del programma in esecuzione

```

#include <stdio.h>
void main (int argc, char *argv[]) {
    int i;
    printf("ho ricevuto %i parametri\n", argc);
    for (i=0; i<argc; i++){
        printf("parametro %i = %s\n", i, argv[i]);
    }
}

```


Esempio di utilizzo di execl

```
/* programma main1 */

#include <stdio.h>

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

    printf ("programma main1 in esecuzione\n");
    printf ("ho ricevuto %d parametri\n", argc);

    for (i=0; i<argc; i++)
        printf("il parametro %d e': %s\n",argv[i]);
}
```

```
/* programma execl */

#include <stdio.h>
#include <sys/types.h>

void main(int argc, char * argv[])
{char P0[]="main1";
 char P1[]="parametro1";
 char P2[]="parametro2";

    printf ("programma execl in esecuzione\n");
    execl("/esempi/main1",P0,P1,P2,NULL);

}
```

Esecuzione:

```
$ ./execl
programma execl in esecuzione

programma main1 in esecuzione
ho ricevuto 3 parametri
il parametro 0 e':main1
il parametro 1 e':parametro1
il parametro 2 e':parametro2
```

Utilizzo fork-exec

- la sostituzione di codice non implica necessariamente la generazione di un figlio (vedi es precedente)
 - In questo caso, quando il programma che è stato lanciato in esecuzione tramite la `execl` termina, termina anche il processo che lo ha lanciato (sono lo stesso processo!!)
- È necessario creare un nuovo processo, che effettua la sostituzione di codice (utilizzo di `fork-exec`), quando è necessario "mantenere in vita" il processo di partenza, dopo l'esecuzione del codice sostituito
 - Spesso questo implica che il padre attenda la terminazione del programma lanciato con mutazione di codice
- Modalità per eseguire una funzione:
 - Il programma chiamante continua l'esecuzione del processo padre
 - Nel processo figlio, il programma viene sostituito dal codice della funzione

esempio 1 di utilizzo di fork-exec

```
/* funzione main1 */

#include <stdio.h>

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

    printf ("programma main1 in esecuzione\n");
    printf ("ho ricevuto %d parametri\n", argc);

    for (i=0; i<argc; i++)
        printf("il parametro %d e': %s\n",argv[i]);
}
```

```
/* programma forkexecl */

#include <stdio.h>
#include <sys/types.h>

void main(int argc, char * argv[])
{pid_t pid;
 int stato_wait;
 char P0[]="main1";
 char P1[]="parametro1";
 char P2[]="parametro2";

    pid=fork();
    if (pid==0) /* sono il processo figlio */
    {printf ("programma forkexecl in
        esecuzione\n");
        printf ("lancio in esecuzione main1\n");
        execl("/esempi/main1",P0,P1,P2,NULL);
        exit();}
    else /* sono il processo padre */
    {wait(&stato_wait);
        printf ("processo figlio terminato\n");
        exit();}
```

esempio 2 di utilizzo di fork-exec

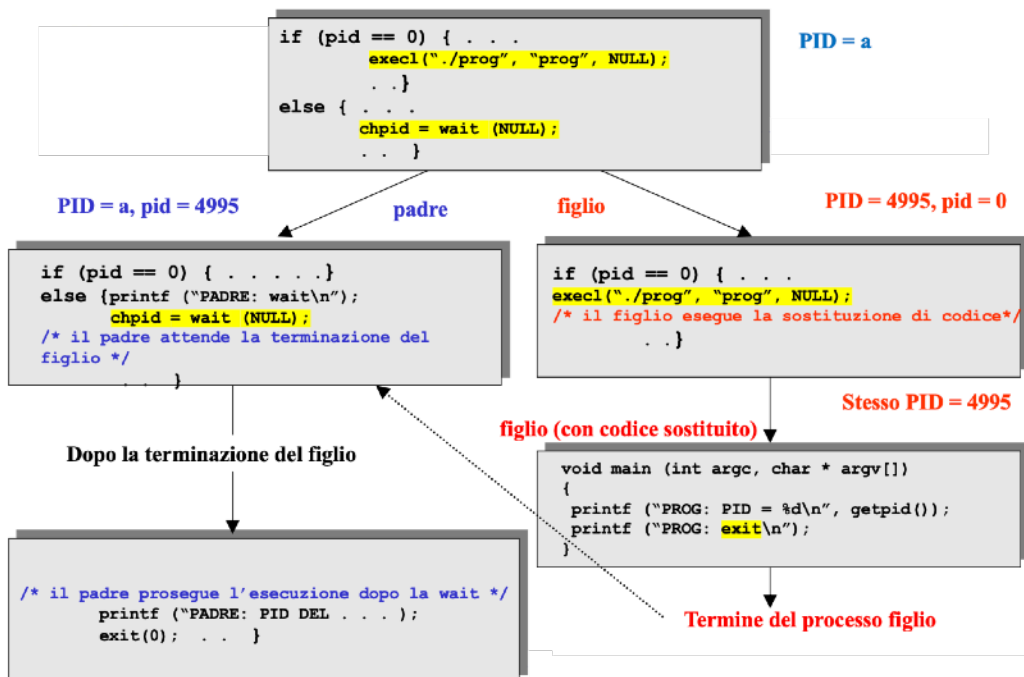
```
void main()
{ pid_t pid, chpid;

  pid = fork();
  if (pid==0) /* PROCESSO FIGLIO*/
    {printf ("FIGLIO: prima del cambio di codice\n")
     printf ("FIGLIO: PID = %d\n", getpid());
     execl("./prog", "prog", NULL);
     printf ("FIGLIO: errore nel cambio di codice\n");
     exit(1);}
  else /* PROCESSO PADRE */
    {printf ("PADRE: wait\n");
     chpid = wait (NULL);
     printf ("PADRE: PID DEL FIGLIO = %d\n", chpid);
     exit(0); }
```

```
void main (int argc, char * argv[])
{
  printf ("PROG: PID = %d\n", getpid());
  printf ("PROG: exit\n");
}
```

Esecuzione nell'ipotesi che venga eseguito prima il padre e poi il figlio →

```
PADRE: wait
FIGLIO: prima del cambio del codice
FIGLIO: PID = 4995
PROG: PID = 4995
PROG: exit
PADRE: PID DEL FIGLIO = 4995
```



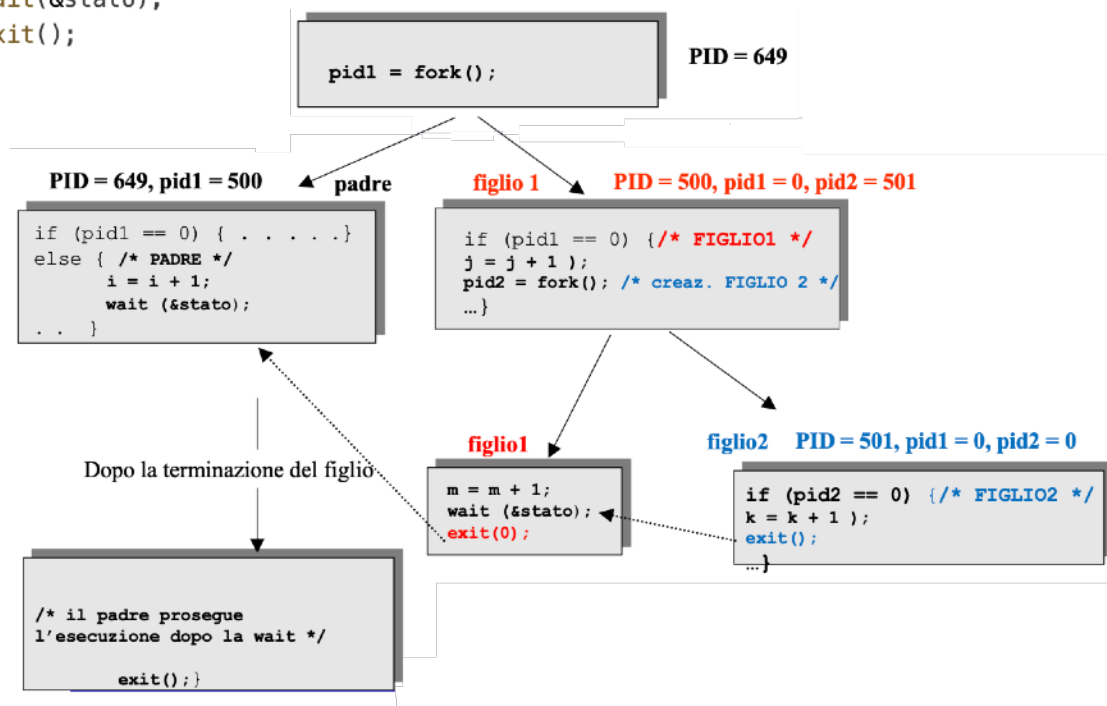
Esercizio conclusivo

```
main() {
    int i, j, k, m, stat;
    pid_t pid1, pid2;
    i = 10;
    j = 20;
    k = 30;
    m = 40;

    pid1 = fork(); /* creazione del primo figlio */

    if (pid1 == 0) { /* primo figlio */
        j = j+1;
        pid2 = fork(); /* creazione del secondo figlio */

        if (pid2 == 0) { /* secondo figlio */
            k = k+1;
            exit();
        }
        else { /* primo figlio */
            m = m+1;
            wait(&stato);
            exit();
        }
    }
    else { /* padre */
        i = i+1;
        wait(&stato);
        exit();
    }
}
```



Commenti:

- L'ordine di terminazione dei processi è deterministico: secondo figlio, primo figlio, padre.
- Non possiamo prevedere l'ordine di esecuzione delle istruzioni di incremento delle 4 variabili i, j, m, k:

```

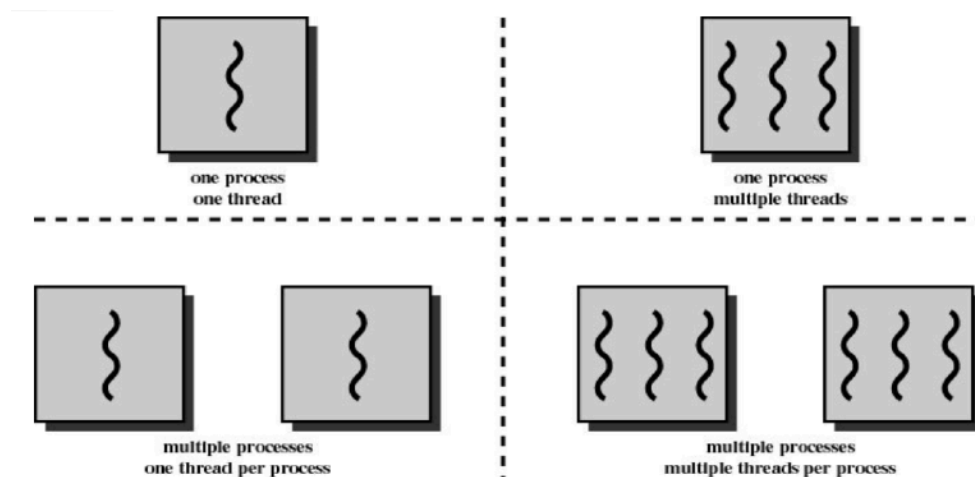
i=i+1;      j=j+1;      k=k+1;
            m=m+1;

```

Introduzione ai thread

- I **thread**, come i processi, costituiscono un meccanismo che consente ad un programma l'esecuzione concorrente
- **Un thread esiste all'interno di un processo**
- Quando si invoca un programma, Linux crea un nuovo processo con un unico thread, che segue il programma sequenzialmente
- Il processo (thread) può creare altri thread che procedono in **parallelo** tra loro
- Il concetto di **multithreading** si può applicare a sistemi mono-processore o multi-processore
- **thread = flusso di controllo** che può essere svolto in parallelo con altri thread nell'ambito di uno **stesso processo**
 - Flusso di controllo = esecuzione sequenziale di istruzioni
- Il flusso di controllo di un thread è costituito da una **funzione** che viene attivata alla creazione del thread
- **Un thread o processo leggero (lightweight process)** consente un grado di cooperazione elevato
 - Lo scambio di informazioni tra thread è facilitato
 - La gestione dei thread da parte del S.O. è meno onerosa rispetto alla gestione dei processi (in termini di strutture dati e funzioni del S.O.)
- Dato che i thread sono attività parallele in uno stesso processo, **i thread condividono lo stesso spazio d'indirizzamento del processo (tranne la pila o stack)** per rendere efficiente l'implementazione dello scambio d'informazioni
 - Le variabili locali della funzione eseguita da un thread appartengono solo a quel thread perchè sono allocate sulla sua pila
- Scopo dello studio del thread: capirne i principi di funzionamento per comprendere alcuni dei problemi principali che si incontrano nella **programmazione parallela o concorrente**

Processi e thread



Esecuzione sequenziale e concorrente

Esecuzione sequenziale

- date due "attività" A e B nel codice di un programma, esse sono **sequenziali** se – esaminando il codice del programma stesso – si può prevedere a priori se A verrà svolta sempre prima di B oppure se B verrà svolta sempre prima di A

Esecuzione concorrente (o parallela)

- date due “attività” A e B nel codice di un programma, esse sono concorrenti se – esaminando il codice del programma stesso – **NON** si può prevedere a priori se A verrà svolta sempre prima di B oppure se B verrà svolta sempre prima di A

L'esecuzione **concorrente (parallela)** di processi/thread può essere:

- **Simulata**: in un sistema di tipo mono-processore, i processi/thread sono in esecuzione a turno sul processore (condivisione di tempo - time-sharing)
- **Reale**: in un sistema di tipo multi-processore, i processi/thread sono in esecuzione contemporaneamente su processori diversi

In entrambi i casi i processi/thread sono **concorrenti** e per l'utente il modello del sistema è sempre lo stesso (naturalmente un sistema multi-processore potrà raggiungere prestazioni superiori)

Anche su un sistema multi-processore, il numero di processi/thread può eccedere il numero di processori, pertanto spesso il parallelismo è in parte simulato

Nota: le considerazioni che verranno fatte nel seguito sono indipendenti dal numero di processori

Granularità di un thread

- il concetto di **multithreading** si può applicare a sistemi mono-processore o multi-processore
- la quantità di istruzioni (**granularità**) assegnata ad ogni thread può variare
 - da poche istruzioni — più adatta nel caso di multithreading su mono-processore (n thread, 1 processore)
 - a centinaia o migliaia di istruzioni — più adatta nel caso di multithreading su multi-processore (n thread, n processori)

Thread POSIX: pthread

- esistono svariati modelli implementativi di thread: noi consideriamo lo **standard POSIX** (Portable Operating System Interface for Computing Environment)
- **Standard POSIX**:
 - Insieme di interfacce applicative (API) di S.O.
 - Utilizzando le funzioni/servizi delle API di POSIX è garantita la portabilità dell'applicazione su tutti i SO che adottano POSIX
 - Tutte le funzioni per la gestione dei thread e i tipi di dato sono dichiarati nel file header `<pthread.h>`
- **pthread = thread di POSIX**

Pthread e processi

- Un thread è attivato nell'ambito di un processo
- Un processo può attivare uno, due o più thread
- Quando il processo termina, terminano forzatamente anche tutti i suoi thread, qualunque sia il loro punto di esecuzione
- È necessario garantire la **terminazione coerente** dei thread in modo da garantire che un processo non termini **prima** che siano terminate tutti i suoi thread
- Se prescindiamo dal fatto che i thread sono sempre contenuti in un processo, i thread possono essere considerati dei veri e propri processi (anche se «leggeri») perché:
 - Sono eseguiti in parallelo tra loro
 - L'esecuzione di un thread può essere sospesa – per esempio può essere posto in attesa di un evento per un'operazione di ingresso/uscita – in modo totalmente indipendente rispetto agli altri thread dello stesso processo

- Ogni pthread ha associati:
 - un **identificatore di thread (thread ID)**, di tipo `pthread_t` che lo identifica univocamente
 - un **identificatore del processo (process ID)** che lo contiene, di tipo `pid_t`
- La funzione `getpid()`, eseguita all'interno dei thread di un stesso processo, restituisce il PID del processo al quale appartengono

PID e TGID

- Linux associa internamente ad ogni processo (normale o leggero) un diverso identificatore. Lo standard Posix richiede che tutti i Thread di uno stesso processo condividano lo stesso **PID** → necessario un adattamento tra i due modelli.
- È necessario associare ad ogni **thread** l'identificatore del **processo** a cui appartiene:
 - un **Thread Group** è costituito dall'insieme di thread (processi leggeri) che appartengono allo stesso processo
 - il **TGID** è uguale al **PID** del *thread group leader*, cioè del primo processo del gruppo (quello col thread di default)
 - i processi con un unico thread (il thread di default) hanno quindi un **TGID** uguale al loro **PID**.

Thread ID, PID e TGID

- Per ogni **thread** è associata una coppia di identificatori: uno per identificare il **thread** e uno per il **Thread Group**
- Per un processo single-thread
 - `gettid()` restituisce il **thread ID** -- che è uguale al **PID**
 - `getpid()` restituisce il **PID** del processo -- che corrisponde al **TGID**
- Per un processo multi-thread
 - `gettid()` restituisce il **thread ID** che è unico per ogni thread
 - `getpid()` restituisce il **PID** del processo che corrisponde al **TGID** che è identico per i thread dello stesso gruppo

Pthread: Creazione - `pthread_create()`

- Creazione di un thread `pthread_create()`
- `pthread_create()` è simile a `fork()`
- Ma un thread viene creato passandogli, con `pthread_create`, il nome della funzione che deve eseguire (**thread function**)
- Un thread inizia a eseguire il suo codice sempre dallo stesso punto (ossia dall'inizio della funzione) indipendentemente dal punto di esecuzione del processo che lo ha creato

Parametri di `pthread_create`:

```
pthread_create (&tID1, NULL, &tf1, (void *) n);
```

1. indirizzo di una variabile di tipo `pthread_t`
 - per contenere l'identificatore del thread creato (thread_ID)
2. puntatore agli attributi del thread
 - se NULL il thread avrà gli attributi di default
3. indirizzo della funzione eseguita dal thread (**thread_function**)
4. indirizzo dell'argomento che si vuole passare alla **thread_function**
 - si può passare un solo argomento che deve essere di tipo `void *`
 - se si vogliono passare più parametri, è necessario creare una struct e passarne l'indirizzo

Pthread: Terminazione - pthread_join()

- Attesa di terminazione tra thread `pthread_join ()`
- `pthread_join()` -- simile a `waitpid()`
- Si deve sempre specificare il thread di cui si vuole attendere la terminazione
- Si utilizza per garantire la terminazione coerente dei thread nel caso di terminazione del processo che ha creato i thread
- Dato che un thread esegue una funzione, termina quando esegue la `return()` o alla fine del codice eseguibile della funzione
- Un thread che termina può passare, tramite la `return()`, un codice di terminazione (exit code) al processo che lo ha creato
- Il passaggio avviene se e quando il processo creatore esegue la funzione `join`
- Il codice di terminazione è passato tramite il secondo parametro di `join`
- Più propriamente, il codice di terminazione del thread è il valore restituito (tramite `return`) dalla funzione eseguita dal thread

Parametri di `pthread_join()`

```
pthread_join (tID1, (void *) &thread_exit);
```

1. **Variabile** di tipo `pthread_t` che contiene l'identificatore del thread (thread_ID) di cui si vuole attendere la terminazione
2. **Puntatore ad una variabile** che conterrà il **codice terminazione** (exit code) passato dalla thread_function

```
// thread function
void * tf1 (void * arg) {
    // codice della funzione
    return (void *) valore_di_ritorno;
} /* tf1 */
```

Esempio: codice tf1

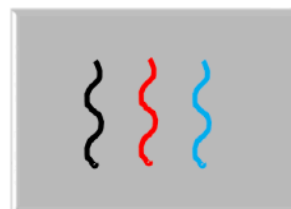
```
#include <pthread.h> <stdio.h>

void * tf1 (void * tID) {
    /* codice thread function tf1 */
} /* tf1 */

int main (void) {
    pthread_t tID1, tID2;
    pthread_create (&tID1, NULL, &tf1,
        (void *) 1); /* genera tID1 */
    pthread_create (&tID2, NULL, &tf1,
        (void *) 2); /* genera tID2 */
    pthread_join (tID1, NULL);
    pthread_join (tID2, NULL);
    return 0;
} /* main */
```

```
/* thread function tf1 */
void * tf1 (void * tID) {

    int conta = 0; /* var. locale */
    conta++;
    printf ("sono il thread n: %d;
        conta = %d \n", (int) tID, conta );
    return NULL;
} /* tf1 */
```



Thread principale

- Un thread viene creato nell'ambito di un processo che ha un suo flusso di controllo: `main ()`
- Si indica **main** come **thread principale** o **di default**
- Quando un codice eseguibile viene lanciato, nel processo viene creato un solo thread, chiamato "thread principale"

- Il thread principale (ossia il modulo main del codice del processo) può creare altri thread tramite `pthread_create`
- A sua volta un thread qualunque può creare altri thread
- Quando un nuovo thread viene creato, si ha esecuzione concorrente dei due thread

Thread e condivisione della memoria

- Un thread viene eseguito nello spazio d'indirizzamento del processo che lo ha creato → **tutti i thread di un processo condividono lo stesso spazio di indirizzamento (la stessa memoria)**
- Attenzione: **ciascun thread di un processo ha la propria pila**, indipendente da quella degli altri thread e allocata ad un indirizzo differente all'interno dello stesso spazio d'indirizzamento del processo
 - Le **variabili locali** della funzione eseguita dal thread appartengono solo a quel thread, poiché sono allocate sulla sua pila
 - Le **variabili statiche** e **quelle globali**, non allocate in pila ma nel segment dati, sono condivise tra tutti i thread di un processo
- Tale condivisione che facilita lo scambio d'informazione tra thread, ma che rende più difficile garantire la correttezza di un programma multithreading

Esempio tf1: uso di variabile locale

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

void * tf1 (void * tID) {
    /* codice thread function tf1 */
} /* tf1 */

int main (void) {
    pthread_t tID1, tID2;
    pthread_create (&tID1, NULL, &tf1,
        (void *)1); /* genera tID1 */
    pthread_create (&tID2, NULL, &tf1,
        (void *)2); /* genera tID2 */
    pthread_join (tID1, NULL);
    pthread_join (tID2, NULL);
    return 0;
} /* main */
```

```
/* thread function tf1 */
void * tf1 (void * tID) {

    int conta = 0; /* var. locale */
    conta++;
    printf ("sono il thread n: %d;
        conta = %d \n", (int) tID, conta );
    return NULL;
} /* tf1 */
```

Esecuzione

```
sono thread n: 1; conta = 1
sono thread n: 2; conta = 1
```

oppure

```
sono thread n: 2; conta = 1
sono thread n: 1; conta = 1
```

Esempio tf1: uso di variabile globale

```
#include <pthread.h> <stdio.h>
int conta = 0; /* var. globale */

void * tf1 (void * tID) {
    /* codice thread function tf1 */
} /* tf1 */

int main (void) {
    pthread_t tID1, tID2;
    pthread_create (&tID1, NULL, &tf1,
        (void *)1); /* genera tID1 */
    pthread_create (&tID2, NULL, &tf1,
        (void *)2); /* genera tID2 */
    pthread_join (tID1, NULL);
    pthread_join (tID2, NULL);
    return 0;
} /* main */
```

```
/* thread function tf1 */
void * tf1 (void * tID) {

    conta++;
    printf ("sono il thread n: %d;
        conta = %d \n", (int) tID, conta );
    return NULL;
} /* tf1 */
```

Esecuzione

```
sono thread n: 1; conta = 1
sono thread n: 2; conta = 2
```

oppure

```
sono thread n: 2; conta = 1
sono thread n: 1; conta = 2
```


Esempio tf1: uso di variabile globale

```
#include <pthread.h> <stdio.h>
int conta = 0; /* var. globale */

void * tf1 (void * tID) {
    /* codice thread function tf1 */
} /* tf1 */

int main (void) {
    pthread_t tID1, tID2;
    pthread_create (&tID1, NULL, &tf1,
        (void *)1); /* genera tID1 */
    pthread_create (&tID2, NULL, &tf1,
        (void *)2); /* genera tID2 */
    pthread_join (tID1, NULL);
    pthread_join (tID2, NULL);
    return 0;
} /* main */
```

```
/* thread function tf1 */
void * tf1 (void * tID) {

    conta++;
    printf ("sono il thread n: %d;
        conta = %d \n", (int) tID, conta );
    return NULL;
} /* tf1 */
```

Esecuzione

```
sono thread n: 1; conta = 1
sono thread n: 2; conta = 2
```

oppure

```
sono thread n: 2; conta = 1
sono thread n: 1; conta = 2
```

Esempio tf1: uso di variabile statica

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

void * tf1 (void * tID) {
    /* codice thread function tf1 */
} /* tf1 */

int main (void) {
    pthread_t tID1, tID2;
    pthread_create (&tID1, NULL, &tf1,
        (void *)1); /* genera tID1 */
    pthread_create (&tID2, NULL, &tf1,
        (void *)2); /* genera tID2 */
    pthread_join (tID1, NULL);
    pthread_join (tID2, NULL);
    return 0;
} /* main */
```

```
/* thread function tf1 */
void * tf1 (void * tID) {

    static int conta = 0; /* var. statica */
    // Variabile statica
    // dichiarata localmente, ma conserva il
    // valore tra una chiamata della funzione
    // e la chiamata successiva
    // Funziona come una variabile globale
    // però ha visibilità solo locale!

    conta++;
    printf ("sono il thread n: %d;
        conta = %d \n", (int) tID, conta );
    return NULL;
} /* tf1 */
```

Esecuzione

```
sono thread n: 1; conta = 1
sono thread n: 2; conta = 2
```

oppure

```
sono thread n: 2; conta = 1
sono thread n: 1; conta = 2
```

Passaggio di parametro ad un thread - esempio

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

void *tf (void *arg) {
    /* codice thread function tf */
} /* tf */

int main (void) {
    int argomento = 10;
    int thread_exit;
    pthread_t tID1;
    pthread_create (&tID1, NULL, &tf,
        &argomento);
    pthread_join (tID1,
        (void *) &thread_exit);
    printf ("sono il main: thread_exit =
        %d\n", thread_exit);
    return 0;
} /* main */
```

```
/* thread function tf */
void *tf (void *arg) {
    // cast per conversione tipo
    int i = * ((int *) arg);
    printf ("sono thread_function:
        valore argomento = %d \n", i);
    i++;
    return (void *) i;
} /* tf */
```

Esecuzione

```
sono thread_function: valore argomento = 10
sono il main: thread_exit = 11
```

Thread e processi: considerazioni d'uso

- **Efficienza:** la copia di memoria per un nuovo processo richiede tecniche di gestione e risorse più onerose rispetto a un thread.
 - in generale il **thread** è più efficiente del processo
- **Protezione:** un thread con errori può danneggiare altri thread nello stesso processo; invece un processo non può danneggiarne un altro (hanno spazi d'indirizzamento disgiunti)
 - i **processi** sono più protetti uno dall'altro
- **Flessibilità:** Il cambiamento di codice eseguibile è possibile solo con un processo; un thread può eseguire solamente il codice della funzione ad esso associata
 - un **processo figlio** può, tramite exec, sostituire l'intero programma eseguibile (maggiore flessibilità)
- **Condivisione dei dati:** la condivisione di dati tra **thread** è molto semplice, mentre tra processi è complicata (richiede l'impiego di meccanismi di comunicazione tra processi – IPC)

Programmazione concorrente (o parallela)

- **Modello di esecuzione sequenziale**
 - Istruzioni eseguite **in ordine predeterminabile**
 - In base al codice del programma e ai valori dei dati in ingresso
- **Modello di programmazione concorrente o parallela**
 - Flussi di controllo (sequenze di istruzioni) eseguiti **in parallelo che interferiscono tra loro**: tipicamente condividono risorse ad esempio lo stesso spazio di indirizzamento (strutture dati)
 - Ordine di esecuzione delle istruzioni **non deterministico**
 - La relazione ingresso-uscita dipende non solo dai dati in ingresso, ma anche dall'ordine di esecuzione che essendo non deterministico può influire la **correttezza** del risultato dell'esecuzione
- Le tecniche di programmazione concorrente si applicano sia ai processi sia ai thread.
- I thread sono maggiormente utilizzati per realizzare flussi di controllo paralleli ma **fortemente cooperanti** perchè condividono lo stesso spazio di indirizzamento (tranne la pila).
- Un programma multi-threading è **potenzialmente non-deterministico nell'esecuzione**, anche in assenza di dipendenze sui dati in ingresso:
 - Impossibile prevedere la prossima istruzione eseguita
 - Potenziale errori introdotti dall'esecuzione concorrente

Esecuzione sequenziale vs esecuzione concorrente**Esecuzione sequenziale**

date due "attività" *Ai* e *Bj* nel codice di un programma, esse sono sequenziali se – esaminando il codice del programma ed eventualmente i dati di ingresso – **si può prevedere** se *Ai* verrà svolta sempre prima di *Bj* oppure se *Bj* verrà svolta sempre prima di *Ai*

$Ai < Bj$ sempre oppure $Bj < Ai$ sempre

Esecuzione concorrente

date due "attività" *Ai* e *Bj* nel codice di un programma, esse sono concorrenti se – esaminando il codice del programma ed eventualmente i dati di ingresso – **NON si può prevedere** se *Ai* verrà svolta sempre prima di *Bj* oppure se *Bj* verrà svolta sempre prima di *Ai*

cioè può essere $Ai < Bj$ ma anche $Bj < Ai$
(non determinismo dell'ordine temporale di esecuzione)

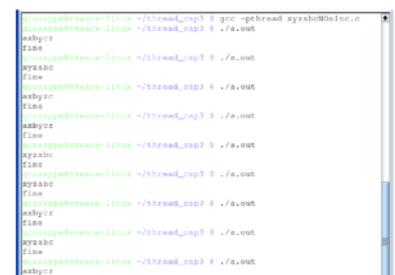
esempio - programmazione concorrente

```
1. #include <pthread.h>
2. #include <stdio.h>

3. void * tf1 (void * arg) {
4.     printf ("x");
5.     printf ("y");
6.     printf ("z");
7.     return NULL;
8. } /* tf1 */

9. void * tf2 (void * arg) {
10.    printf ("a");
11.    printf ("b");
12.    printf ("c");
13.    return NULL;
14. } /* tf2 */

15. int main (void) {
16.     pthread_t tID1, tID2;
17.     pthread_create (&tID1, NULL,
18.                     tf1, NULL);
19.     pthread_create (&tID2, NULL,
20.                     tf2, NULL);
21.     pthread_join (tID1, NULL);
22.     pthread_join (tID2, NULL);
23.     printf ("\n fine\n");
24.     return 0;
25. } /* main */
```



Ciascuno dei due thread è sequenziale al suo interno, ma i due thread sono in concorrenza tra loro!

Correttezza di un programma concorrente

- dimostrazione formale di correttezza → basata su logica formale - difficile
- testing o verifica il comportamento d'uscita del programma in funzione degli ingressi
 - È uno strumento efficace per la verifica **nell'esecuzione sequenziale**
 - Nei **programmi concorrenti**, la relazione tra ingresso-uscita dipende non solo dai dati, ma anche dall'ordine di esecuzione delle istruzioni, che **non è deterministico** → sul essere determinante per la **correttezza del risultato** dell'esecuzione

Sommario

Tecniche/costrutti di programmazione ad hoc per risolvere:

1. **Sequenze critiche e mutua esclusione (mutex)**
2. **Deadlock**
3. **Sincronizzazione e semafori**

SEQUENZE CRITICHE E MUTUA ESCLUSIONE

Sequenza critica

- **Sequenza** = successione sequenziale di operazioni/istruzioni eseguite da un thread
 $ti.j$ = istruzione j eseguita dal thread i
- Relazione temporale tra istruzioni
 - Dello stesso thread $ti.1 < ti.2 < \dots < ti.n$
 - Di thread diversi ($t1$ e $t2$) $t1.1 < t1.2 < t2.1 < t1.3 < \dots < ti.n$
- **Sequenza critica**
 - sequenza di istruzioni eseguite dai thread paralleli che **NON DEVONO ESSERE "MESCOLATE" TRA LORO** affinché il risultato dell'esecuzione sia **sempre** corretto
- bisogna **eseguire in sequenza (sequenzializzare)** la sequenza critica **tra i vari thread** in modo che:
 - Se un qualsiasi thread comincia l'esecuzione della sequenza
 - Tutta la sequenza deve essere eseguita da quel thread
 - Prima di essere eseguita da un qualsiasi altro thread (si riduce la concorrenza)
- La **mutua esclusione** è la proprietà che si vuole garantire per **l'esecuzione concorrente delle sequenze critiche**, affinché il risultato sia corretto

esempio 1a - sequenza critica

- **Problema:** Siano dati 2 conti correnti bancari, **contoA** e **contoB**, si vuole realizzare una funzione **trasferisci()**, che prelevi un importo dal **contoB** e lo depositi sul **contoA**.
- Dopo l'esecuzione di tale funzione la somma dei due conti dovrà essere invariata: una proprietà di questo tipo è detta **invariante** della funzione.

Passi da fare:

1. Memorizza **contoA** in una variabile locale **cA**
2. Memorizza **contoB** in una variabile locale **cB**
3. Aggiorna **contoA** al nuovo valore **cA + importo**
4. Aggiorna **contoB** al nuovo valore **cB - importo**

Operazioni di trasferimento (attivabili in parallelo) tra due conti correnti **contoA** e **contoB** (saldo dei cc) variabili globali

Val iniziale **contoA** = 100

Val iniziale **contoB** = 200

Alla fine l'invariante è 300, ovvero la somma dei due conti

Trasferimenti da B ad A: **t1**: importo = 10, **t2**: importo = 20

Se trasferimento corretto: val finale **contoA** = 130, val finale **contoB** = 170 → la somma è appunto 300

3 tra le possibili soluzioni

S1) **t1.1** < **t1.2** < **t1.3** < **t1.4** < **t2.1** < **t2.2** < **t2.3** < **t2.4** → tutto ok

S2) **t2.1** < **t2.2** < **t2.3** < **t2.4** < **t1.1** < **t1.2** < **t1.3** < **t1.4** → tutto ok

S3) **t1.1** < **t1.2** < **t1.3** < **t2.1** < **t2.2** < **t2.3** < **t2.4** < **t1.4** → non va bene

Perchè non va bene la terza?

		dopo							
	inizio	t1.1	t1.2	t1.3	t2.1	t2.2	t2.3	t2.4	t1.4
contoA	100			110			130		
contoB	200							180	190
cA (in t1)		100							
cB (in t1)			200						
cA (in t2)					110				
cB (in t2)						200			

t1 trasferisce 10 da contoB a contoA
t2 trasferisce 20 da contoB a contoA

t2 memorizza contoB in cB prima che contoB sia stato aggiornato a 190 da parte di t1.4

```

1. #include <pthread.h> <stdio.h>
2. int contoA = 100, contoB = 200;
3. int totale;
4. void *trasferisci (void *arg) {
5.     int importo = * ((int *) arg);
6.     int cA, cB;
7.     // inizio sequenza critica
8.     // leggi contoA in var locale
9.     cA = contoA;
10.    // leggi contoB in var locale
11.    cB = contoB;
12.    contoA = cA + importo;
13.    contoB = cB - importo;
14.    // fine sequenza critica
15.    return NULL;
16. } /* importo */

17. int main (void) {
18.     pthread_t t1D1, t1D2;
19.     int import1 = 10, import2 = 20;
20.     pthread_create (&t1D1, NULL, trasferisci, &import1);
21.     pthread_create (&t1D2, NULL, trasferisci, &import2);
22.     pthread_join (t1D1, NULL);
23.     pthread_join (t1D2, NULL);
24.     totale = contoA + contoB;
25.     printf ("contoA = %d contoB = %d\n",
26.            contoA, contoB);
27.     printf ("t1 totale è %d\n", totale);
28.     return 0;
29. } /* main */

```

Per risolvere questo problema uso le **istruzioni atomiche**

Istruzioni atomiche

- Istruzioni da considerarsi **indivisibili o atomiche** durante l'esecuzione:
 - contoA = contoA + importo**
 - contoB = contoB - importo**
- se ciascuna delle istruzioni a. e b. fosse **atomica** => qualsiasi ordinamento di istruzioni relativo ai due thread fornirà sempre un risultato corretto:
 - t1.a** < **t1.b** < **t2.a** < **t2.b** oppure **t2.a** < **t2.b** < **t1.a** < **t1.b** ovvio !!! ma anche:
 - t1.a** < **t2.a** < **t1.b** < **t2.b** (S2)
 - t1.a** < **t2.a** < **t2.b** < **t1.b** (S3) e le altre due permutazioni possibili:
 - t2.a** < **t1.a** < **t1.b** < **t2.b**
 - t2.a** < **t1.a** < **t2.b** < **t1.b**

S2		dopo			
	inizio	t1.a	t2.a	t1.b	t2.b
contoA	100	110	130		
contoB	200			190	170

OK! e termina prima t1

S3		dopo			
	inizio	t1.a	t2.a	t2.b	t1.b
contoA	100	110	130		
contoB	200			180	170

OK! e termina prima t2

Istruzione atomica e linguaggio macchina

- le singole istruzioni (o i costrutti) di un linguaggio di alto livello vengono tradotte dal compilatore in una **sequenza di istruzioni in linguaggio macchina**
- le **single istruzioni in linguaggio macchina sono atomiche (ma non quelle di alto livello!)**
- il programmatore dovrebbe sapere come il linguaggio di alto livello viene tradotto in linguaggio macchina

L'istruzione ad alto livello: **contoA = contoA + 100** è traducibile in linguaggio macchina in 2 modi ≠:

1. Unica istruzione con processore che lavora direttamente in memoria (esempio: Intel):

```
ADD    contoA, 100    // somma 100 a contoA
```

2. Tre istruzioni con processore LOAD/STORE tipo RISC-V con uso di un registro interno t0

```
ld      t0, contoA     // carica contoA in t0
addi    t0, t0, 100     // somma 100 a t0
sd      t0, contoA     // memorizza t0 in contoA
```

Mutua esclusione - mutex

- per garantire la correttezza dell'esecuzione concorrente di sequenze critiche o istruzioni non atomiche, occorre garantire la proprietà di **mutua esclusione** sulla sequenza o istruzione
- in **pthread** il costrutto specializzato per realizzare la mutua esclusione si chiama **mutex**
 - un mutex implementa "un blocco" che "ingloba" la sequenza/istruzione critica e il cui accesso è controllato da un segnale di "occupato"
 - quando un thread attiva un blocco, eventuali altri thread che tentano di accedere al blocco vengono messi in attesa finché il thread non libera o rilascia il blocco
 - l'operazione di attivazione del blocco è chiamata **lock**
 - l'operazione di rilascio del blocco è chiamata **unlock**
- Va dichiarata una variabile di tipo **pthread_mutex_t** (**conti** nel programma)
- Va inizializzata tramite la funzione **pthread_mutex_init**
- Successivamente il mutex va bloccato e sbloccato tramite
 - **pthread_mutex_lock**
 - **pthread_mutex_unlock**
- Quando un thread esegue l'operazione di lock, se il mutex è già bloccato il thread è posto in attesa di sblocco

esempio - mutex

```
// seq. critica con mutex
1. #include <pthread.h> <stdio.h>
2. int contoA = 100, contoB = 200;
3. int totale;
4. pthread_mutex_t conti;
5. // dichiarazione di mutex
6. void * trasferisci (void * arg) {
7.     int importo = * ((int *) arg);
8.     int cA, cB;
9.     pthread_mutex_lock (&conti);
10.    // inizio sequenza critica
11.    // leggi contoA in var locale
12.    cA = contoA;
13.    // leggi contoB in var locale
14.    cB = contoB;
15.    contoA = cA + importo;
16.    contoB = cB - importo;
17.    pthread_mutex_unlock (&conti);
18.    // fine sequenza critica
19.    return NULL;
20. } /* trasferisci */

21. int main (void) {
22.     pthread_t tID1, tID2;
23.     int importo1 = 10, importo2 = 20;
24.     pthread_mutex_init (&conti, NULL);
25.     // inizializza mutex
26.     pthread_create (&tID1, NULL, trasferisci, &importo1);
27.     pthread_create (&tID2, NULL, trasferisci, &importo2);
28.     pthread_join (tID1, NULL);
29.     pthread_join (tID2, NULL);
30.     totale = contoA + contoB;
31.     printf ("contoA = %d contoB = %d\n",
32.            contoA, contoB);
33.     printf ("il totale è %d\n", totale);
34.     return 0;
35. } /* main */

giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
contoA = 130 contoB = 170
il totale è 300
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
contoA = 130 contoB = 170
il totale è 300
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
contoA = 130 contoB = 170
il totale è 300
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
contoA = 130 contoB = 170
il totale è 300
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
contoA = 130 contoB = 170
il totale è 300
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
contoA = 130 contoB = 170
il totale è 300
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
contoA = 130 contoB = 170
il totale è 300
```

esempio - mutex sequenza critica generica

```
// seq. critica generica con mutex
1. #include <pthread.h> <stdio.h>
2. pthread_mutex_t blocca;
3. // dichiarazione di mutex
4. void * trasferisci (void * arg) {
5.     pthread_mutex_lock (&blocca);
6.     // inizio sequenza critica
7.     printf ("thread %d: entro in seq
8.     critica\n", (int) arg);
9.     printf ("thread %d: termino seq
10.    critica\n", (int) arg);
11.     pthread_mutex_unlock (&blocca);
12.     // fine sequenza critica
13.     return NULL;
14. } /* trasferisci */

21. int main (void) {
22.     pthread_t tID1, tID2;
23.     pthread_mutex_init (&blocca, NULL);
24.     // inizializza mutex
25.     pthread_create (&tID1, NULL, trasferisci, (void *) 1);
26.     pthread_create (&tID2, NULL, trasferisci, (void *) 2);
27.     pthread_join (tID1, NULL);
28.     pthread_join (tID2, NULL);
29.     printf ("fine\n");
30.     return 0;
31. } /* main */
```

SEQUENZA CRITICA SENZA MUTEX

- Proviamo ad implementare la sequenza critica e la mutua esclusione anche in assenza di librerie specifiche per la programmazione concorrente (cioè senza i `mutex` della libreria `pthread`)
- In questo caso la gestione della sequenza critica diventa di realizzazione software **più complessa**
- L'esercizio ci consente di analizzare meglio alcuni possibili "rischi" della programmazione concorrente come i **deadlock**

Sequenza critica senza mutex - versione 1

- Senza usare mutex, il blocco di accesso alla sequenza critica è gestito da una variabile globale intera `blocca`
 - se `blocca == 1` sequenza bloccata già in esecuzione da parte di un thread
 - se `blocca == 0` sequenza eseguibile (da parte di un thread)
- `mutex_lock` è implementato tramite un ciclo di attesa (**busy waiting**) finché la variabile `blocca` diventa 0 in modo da entrare in sequenza critica e assegnare `blocca` a 1:

```
while (blocca == 1);
blocca = 1;
```
- `mutex_unlock` (uscita dalla seq. critica) è implementato così:

```
blocca = 0;
```

Versione 1 (mutua esclusione non garantita)

```
// sequenza critica con variabile
// globale intera al posto del mutex
1. #include <pthread.h> <stdio.h>
2. int blocca = 0;
3. void * trasferisci (void * arg) {
4.     // busy waiting
5.     while (blocca == 1);
6.     // inizio sequenza critica
7.     blocca = 1;
8.     printf ("thread %d: entro in seq
        critica\n", (int) arg);
9.     printf ("thread %d: termino seq
        critica\n", (int) arg);
10.    blocca = 0;
11.    // fine sequenza critica
12.    return NULL;
13. } /* trasferisci */
```

```
14. int main (void) {
15.     pthread_t tID1, tID2;
16.     pthread_create (
17.         &tID1, NULL,
18.         trasferisci, (void *) 1);
19.     pthread_create (
20.         &tID2, NULL,
21.         trasferisci, (void *) 2);
22.     pthread_join (tID1, NULL);
23.     pthread_join (tID2, NULL);
24.     printf ("fine\n");
25.     return 0;
26. } /* main */
```

```
giuseppesvizzera-linux ~/thread_caps $ ./a.out
thread 1: entro in sequenza critica
thread 2: entro in sequenza critica
thread 1: termino sequenza critica
thread 2: termino sequenza critica
fine
```

Soluzione **errata**: è possibile che si verifichi la sequenza di istruzioni $t1.5$ (= false → ingr. in seq. critica) < $t2.5$ (= false → ingr. in seq. critica) ...

=> I due thread entrano entrambi in sequenza critica ... mutua esclusione non garantita

Versione 2 (mutua esclusione con deadlock)

- Ad ogni thread che esegue la sequenza critica si associa una variabile globale `blocca_x` che rappresenta l'**intenzione** del `thread_X` di accedere alla sequenza critica:
 - se `blocca_x == 1` il thread X **vuole** entrare nella sequenza
 - se `blocca_x == 0` il thread X **non vuole** entrare nella sequenza
- Per poter effettivamente entrare nella sequenza critica, il generico `thread_X` dopo avere dichiarato le sue intenzioni (`blocca_x == 1`)
 - testa ogni altra variabile `blocca_y`
 - e rimane in attesa se una qualsiasi di queste è a 1
- Al termine della sequenza critica, ogni `thread_X` sblocca la sua variabile (`blocca_x == 0`)
- Nota: per svolgere un'azione diversa in base al thread, la funzione `trasferisci` riceve un indice di thread (e non il tID) come argomento


```
// mutua esclusione con deadlock

1. #include <pthread.h> <stdio.h>
2. int bloccal = 0, blocca2 = 0;
3. void * trasferisci (void * arg) {
4.     if ((int) arg == 1) {bloccal = 1;
5.         while (blocca2 == 1);} /* if */
6.     if ((int) arg == 2) {blocca2 = 1;
7.         while (bloccal == 1);} /* if */
8.     // inizio sequenza critica
9.     printf ("thread %d: entro in seq
        critica\n", (int) arg);
10.    printf ("thread %d: termino seq
        critica\n", (int) arg);
11.    if ((int) arg == 1) bloccal = 0;
12.    if ((int) arg == 2) blocca2 = 0;
13.    // fine sequenza critica
14.    return NULL;
15. } /* trasferisci */

16. int main (void) {
17.     pthread_t tID1, tID2;
18.     pthread_create (
19.         &tID1, NULL,
20.         trasferisci, (void *) 1
21.     );
22.     pthread_create (
23.         &tID2, NULL,
24.         trasferisci, (void *) 2
25.     );
26.     pthread_join (tID1, NULL);
27.     pthread_join (tID2, NULL);
28.     printf ("fine\n");
29.     return 0;
30. } /* main */
```

Se uno dei due thread entra in sequenza critica => **è garantita la mutua esclusione**:

- se t1 è in **t1.8 < inizio di t2.7** ovvero t1 entra nella sequenza critica prima che t2 arrivi alla riga 7, dove verrà bloccato nel ciclo di attesa fino a quando t1 non uscirà dalla sequenza critica => **la mutua esclusione è rispettata**;
- se t1 è in **t1.8** ovvero t1 è entrato nella sequenza critica allora deve essere successo che:
 - a) **t1.5 < t2.6** cioè t1 deve aver terminato il ciclo di attesa su blocca_2 prima che t2 abbia asserito blocca_2 alla riga 6 oppure
 - b) **t2.12 < t1.5** cioè t2 ha rilasciato blocca_2 ed è uscito dalla sequenza critica prima che t1 sia entrato nel ciclo di attesa su blocca_2

Si noti che t2.6 e t2.12 sono le uniche due istruzioni che modificano blocca_2

=> **in entrambi i casi la mutua esclusione è rispettata**;

Data la simmetria del problema, lo stesso ragionamento si applica al caso in cui sia t2 nella sezione critica

- tuttavia la seguente sequenza di esecuzione delle istruzioni: **t1.4 < t2.6 < t1.5 < t2.7** blocca in attesa in modo indefinito sia thread_1 sia thread_2 senza che nessuno dei due possa entrare nella sequenza critica liberando la propria variabile e sbloccando il ciclo di attesa dell'altro thread;
- Questa situazione quando un thread aspetta l'azione di un altro thread per poter proseguire e contemporaneamente il secondo aspetta l'azione del primo, si chiama **deadlock** o attesa circolare.
- Un **deadlock** è uno degli errori più gravi nei quali si può incorrere nella programmazione concorrente e sarà necessario terminare il processo da parte del sistema operativo

Versione 3 (eliminazione deadlock - algoritmo di Petersen)

- Estendiamo la soluzione precedente (che garantisce la mutua esclusione) in modo da eliminare il **deadlock** introducendo una terza variabile globale **favorito** in modo che ciascun thread gli assegni come valore l'indice dell'altro thread in modo da rendere l'altro thread preferenziale ad entrare nella sequenza critica.
- L'ultimo thread che esegue l'istruzione di assegnamento a **favorito** è quello che **resta in attesa** finché l'esecuzione della sequenza critica da parte dell'altro non sarà terminata
- Il **deadlock non potrà verificarsi** poiché la variabile **favorito** è unica e il test su di essa risulterà falsa la condizione di in uno dei due cicli di attesa alle righe 6 e 10.

```
// mutua esclusione corretta
1. #include <pthread.h> <stdio.h>
2. int favorito = 1, bloccal = 0, blocca2 = 0;
3. void * trasferisci (void * arg) {
4.     if ((int) arg == 1) {bloccal = 1;
5.         favorito = 2;
6.         while (blocca2 == 1 && favorito == 2);} /* if */
7.     if ((int) arg == 2) {blocca2 = 1;
8.         favorito = 1;
9.         while (bloccal == 1 && favorito == 1);} /* if */
10.    // inizio sequenza critica
11.    printf ("thread %d: entro in seq
        critica\n", (int) arg);
12.    printf ("thread %d: termino seq
        critica\n", (int) arg);
13.    if ((int) arg == 1) bloccal = 0;
14.    if ((int) arg == 2) blocca2 = 0;
15.    // fine sequenza critica
16.    return NULL;
17. } /* trasferisci */

22. int main (void) {
23.     pthread_t tID1, tID2;
24.     pthread_create (
25.         &tID1, NULL,
26.         trasferisci, (void *) 1
27.     );
28.     pthread_create (
29.         &tID2, NULL,
30.         trasferisci, (void *) 2
31.     );
32.     pthread_join (tID1, NULL);
33.     pthread_join (tID2, NULL);
34.     printf ("fine\n");
35.     return 0;
36. } /* main */
```


- Infatti se sono state eseguite in qualsiasi ordine le istruzioni:
 - $t1.4$ (`blocca1 = 1`)
 - $t2.8$ (`blocca2 = 1`)
 e nessuna delle altre istruzioni (situazione che nella versione 2 portava al deadlock)
- allora, se $t1.5 < t2.9$ (ultimo assegnamento da parte di $t2$ alla variabile favorito)
 - **thread 2** rimane in attesa
 - mentre **thread 1** entra in sequenza critica
- viceversa se $t2.9 < t1.5$ (ultimo assegnamento da parte di $t1$ alla variabile favorito)
 - **thread 1** rimane in attesa
 - mentre **thread 2** entra in sequenza critica

DEADLOCK E MUTEX

In generale si genera un **deadlock** o attesa circolare quando:

- due thread devono **bloccare due risorse A e B** con due mutex diversi (per accedervi in mutua esclusione)
- **ma le bloccano in ordine inverso**:
 - **thread₁** ha bloccato **A** e attende di poter bloccare **B**
 - **thread₂** ha bloccato **B** e attende di poter bloccare **A**
- esempio: 2 thread devono eseguire in ordine inverso due sequenze critiche annidate => **deadlock**

esempio - deadlock e mutex

```
pthread_mutex_t mutexA, mutexB;

1. void * lockApoiB (void * arg) {
2.     pthread_mutex_lock (&mutexA);
3.     // inizio sequenza critica 1
4.     printf ("thread %d: entro in seq
      critica 1\n", (int) arg);
5.     pthread_mutex_lock (&mutexB);
6.     // inizio sequenza critica 2
7.     printf ("thread %d: entro in seq
      critica 2\n", (int) arg);
8.     printf ("thread %d: termino seq
      critica 2\n", (int) arg);
9.     pthread_mutex_unlock (&mutexB);
10.    // fine sequenza critica 2
11.    printf ("thread %d: termino seq
      critica 1\n", (int) arg);
12.    pthread_mutex_unlock (&mutexA);
13.    // fine sequenza critica 1
14.    return NULL;
15. } /* lockApoiB */

16. void * lockBpoiA (void * arg) {
17.     pthread_mutex_lock (&mutexB);
18.     // inizio sequenza critica 2
19.     printf ("thread %d: entro in seq
      critica 2\n", (int) arg);
20.     pthread_mutex_lock (&mutexA);
21.     // inizio sequenza critica 1
22.     printf ("thread %d: entro in seq
      critica 1\n", (int) arg);
23.     printf ("thread %d: termino seq
      critica 1\n", (int) arg);
24.     pthread_mutex_unlock (&mutexA);
25.     // fine sequenza critica 1
26.     printf ("thread %d: termino seq
      critica 2\n", (int) arg);
27.     pthread_mutex_unlock (&mutexB);
28.     // fine sequenza critica 2
29.     return NULL;
30. } /* lockBpoiA */

int main (void) {
    pthread_t tID1, tID2;
    pthread_mutex_init (&mutexA, NULL);
    pthread_mutex_init (&mutexB, NULL);

    pthread_create (&tID1, NULL,
        lockApoiB, (void *) 1);
    pthread_create (&tID2, NULL,
        lockBpoiA, (void *) 2);

    pthread_join (tID1, NULL);
    pthread_join (tID2, NULL);

    printf ("fine\n");
    return 0;
} /* main */
```

- Il main attiva due thread $t1$ e $t2$ ciascuno dei quali deve eseguire 2 sequenze critiche annidate ma in ordine inverso: $t1$ deve eseguire **lockApoiB** mentre $t2$ esegue **lockBpoiA**.
- Se $t1$ è riuscito a bloccare entrambi i mutex prima di $t2$, l'esecuzione delle due sequenze critiche sarà corretta:

$$t1.4 < t1.6 < t1.9 < t1.11 < t2.15 < t2.17 < t2.20 < t2.22$$

- In altre situazioni si può verificare un deadlock e sarà necessario terminare il programma dall'esterno.
- Ad esempio, un deadlock può verificarsi se $t1$ arriva a bloccare mutexA e $t2$ arriva a bloccare il mutexB prima che $t1$ riesca a bloccare mutexB e va in attesa e prima che $t2$ riesca a bloccare mutexA e va in attesa:

$$t1.4 < t2.15 < t1.6 < t2.17$$

```

giuseppe@vmware-linux ~/thread_cap3/esempiMutexETrasf $ ./a.out
thread 1: entro in sequenza critica 1
thread 1: entro in sequenza critica 2
thread 1: termino sequenza critica 2
thread 1: termino sequenza critica 1
thread 2: entro in sequenza critica 2
thread 2: entro in sequenza critica 1
thread 2: termino sequenza critica 1
thread 2: termino sequenza critica 2
fine
giuseppe@vmware-linux ~/thread_cap3/esempiMutexETrasf $ gcc -pthread
CritSecMutex12deadlock.c
giuseppe@vmware-linux ~/thread_cap3/esempiMutexETrasf $ ./a.out
thread 2: entro in sequenza critica 2
thread 1: entro in sequenza critica 1
Terminated
giuseppe@vmware-linux ~/thread_cap3/esempiMutexETrasf $ ./a.out
thread 1: entro in sequenza critica 1
thread 2: entro in sequenza critica 2
Terminated
giuseppe@vmware-linux ~/thread_cap3/esempiMutexETrasf $ ./a.out
thread 2: entro in sequenza critica 2
thread 1: entro in sequenza critica 1
Terminated

```

SINCRONIZZAZIONE E SEMAFORI

Sincronizzazione = relazione temporale deterministica che si vuole imporre tra due o più thread

Il costrutto specializzato per realizzare la sincronizzazione è il **semaforo** (rif. libreria `pthread`)

- il semaforo è una variabile di tipo `sem_t`
- per le operazioni è considerato molto simile a un intero con valore **positivo** o **nullo**
- il semaforo viene inizializzato tramite la primitiva `sem_init(...)`, ad es. `sem_init(&sem1, 0, 0)` dove il secondo parametro è sempre **0** e il terzo indica il valore iniziale;
- un thread utilizza un semaforo tramite due sole primitive (operazioni): `sem_wait(...)` e `sem_post(...)`

`sem_wait (...)`

- se il semaforo aveva **valore > 0**, il thread che ha invocato la `sem_wait` **decrementa il valore del semaforo** e prosegue nell'esecuzione;
- se il semaforo aveva **valore == 0**, allora il thread che ha invocato la `sem_wait` **rimane bloccato** finché non sarà eseguita da un altro thread una `sem_post` in modo che il semaforo diventi > 0 e quindi va al punto precedente;

`sem_post (...)`

- il thread che ha invocato la `sem_post` **incrementa il valore del semaforo** e prosegue nell'esecuzione, eventualmente sbloccando **un thread** accodato

Il valore del semaforo è interpretabile come il “**numero di risorse**” disponibili – ossia associate al semaforo – per i thread che lo usano (rif. libreria `pthread`)

Se il **valore è > 0**, rappresenta il numero di risorse disponibili, cioè quanti thread lo possono decrementare (ossia usare una risorsa) senza andare in attesa; un **incremento** può essere visto come una **restituzione di risorsa**;

Se il **valore è = 0** significa che non ci sono risorse disponibili e ci possono essere uno o più thread bloccati in attesa;

esempio 1 - sincronizzazione e semaforo

- Due thread stampano le due sequenze di caratteri "abc" e "xyz", rispettivamente.
- In uscita si vuole stampare esattamente la sequenza "axbycz"
 - la funzione **tf1** stampa singolarmente i caratteri x, y e z
 - la funzione **tf2** stampa singolarmente i caratteri a, b e c
- Occorrono **2 semafori**:
 - **sem1** blocca il 1° thread che scrive x, y e z
 - **sem2** blocca il 2° thread che scrive a, b e c
- Le due funzioni sono diverse perché si deve iniziare la stampa sempre con una "a"
- Entrambi i semafori sono inizializzati a **0** in modo da bloccare il thread che esegue una **sem_wait** sul semaforo;
- Il primo thread, che esegue **tf1**, esegue subito una **sem_wait(&sem1)** e si blocca, mentre il secondo thread, che esegue **tf2**, stampa una "a", esegue una **sem_post(&sem1)** per sbloccare il primo thread e prosegue l'esecuzione con una **sem_wait(&sem2)** che si blocca a sua volta.
- Dopo la 'a' stampata dal secondo thread, l'unico carattere che può essere stampato è quindi la 'x' stampata dal primo thread che è stato sbloccato, mentre il secondo è bloccato.
- L'alternanza dei due thread procede secondo questo schema

```
1. #include <pthread.h> <stdio.h> <semaphore.h>
2. sem_t sem1, sem2;
3. void * tf1 (void * arg) {
4.     sem_wait (&sem1);
5.     printf ("x");
6.     sem_post (&sem2); sem_wait (&sem1);
7.     printf ("y");
8.     sem_post (&sem2); sem_wait (&sem1);
9.     printf ("z");
10.    return NULL;
11. } /* tf1 */
12. void * tf2 (void * arg) {
13.     printf ("a");
14.     sem_post (&sem1); sem_wait (&sem2);
15.     printf ("b");
16.     sem_post (&sem1); sem_wait (&sem2);
17.     printf ("c");
18.     sem_post (&sem1);
19.     return NULL;
20. } /* tf2 */
```

sem1 blocca il primo thread che scrive x, y e z
sem2 blocca il secondo thread che scrive a, b e c

```
21. int main (void) {
22.     pthread_t tID1, tID2;
23.     sem_init (&sem1, 0, 0);
24.     sem_init (&sem2, 0, 0);
25.     pthread_create (
26.         &tID1, NULL, tf1, NULL
27.     );
28.     pthread_create (
29.         &tID2, NULL, tf2, NULL
30.     );
31.     pthread_join (tID1, NULL);
32.     pthread_join (tID2, NULL);
33.     printf ("\n fine\n");
34.     return 0;
35. } /* main */
```

```
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
axbycz
fine
```

esempio 2 - sincronizzazione e semaforo

- **Produttore/consumatore**: esempio tipico di utilizzo di semafori: si vuole implementare un programma che fa uso di un "buffer" (array di caratteri) dove un thread **produttore** scrive caratteri e un thread **consumatore** legge i caratteri.
- Versione semplice con buffer che contiene **un solo carattere**:
 - un thread (**produttore**) scrive caratteri nel buffer in sequenza
 - un thread (**consumatore**) legge i caratteri dal buffer in sequenza

- Occorrono **2 semafori** che indicano lo stato del buffer

- **pieno** per bloccare il **consumatore**: se il produttore non ha ancora inserito/scritto il carattere (viene inizializzato a 0: non ci sono caratteri presenti)
- **vuoto** per bloccare il **produttore**, se il consumatore non ha ancora prelevato/letto il carattere (viene inizializzato a 1: ci sono posizioni disponibili per scrivere)

<pre>// produttore consumatore con semafori #include <pthread.h> <stdio.h> <semaphore.h> char buffer; int fine = 0; sem_t pieno, vuoto; 1. void * scrivi (void * arg) { 2. int i; 3. for (i = 0; i < 5; i++) { 4. sem_wait (&vuoto); 5. buffer = 'a' + i; 6. sem_post (&pieno); 7. } /* for */ 8. fine = 1; 9. return NULL; 10. } /* scrivi */</pre>	<pre>1. void * leggi (void * arg) { 2. char miobuffer; 3. int i; 4. while (fine == 0) { 5. sem_wait (&pieno); 6. miobuffer = buffer; 7. sem_post (&vuoto); 8. printf ("il buffer contiene 9. %.1s \n", &miobuffer); 10. } /* while */ 11. return NULL; 12. } /* leggi */</pre>
PRODUTTORE	CONSUMATORE

vuoto blocca il thread produttore che scrive nel buffer
pieno blocca il thread consumatore che legge dal buffer

Con errore di lettura ultimo carattere

<pre>1. #include <pthread.h> <stdio.h> <semaphore.h> 2. char buffer; int fine = 0; sem_t pieno, vuoto; 3. void * scrivi (void * arg) { 4. int i; 5. for (i = 0; i < 5; i++) { 6. sem_wait (&vuoto); 7. buffer = 'a' + i; 8. sem_post (&pieno); 9. } /* for */ 10. fine = 1; 11. return NULL; 12. } /* scrivi */ 13. void * leggi (void * arg) { 14. char miobuffer; 15. int i; 16. while (fine == 0) { 17. sem_wait (&pieno); 18. miobuffer = buffer; 19. sem_post (&vuoto); 20. printf ("buffer contiene %.1s \n", &miobuffer); 21. } /* while */ 22. return NULL; 23. } /* leggi */</pre>	<pre>// produttore consumatore con semafori 24. int main (void) { 25. pthread_t tID1, tID2; 26. sem_init (&pieno, 0, 0); 27. sem_init (&vuoto, 0, 1); 28. // buffer vuoto all'inizio 29. pthread_create (&tID1, NULL, scrivi, NULL); 30. pthread_create (&tID2, NULL, leggi, NULL); 31. pthread_join (tID1, NULL); 32. pthread_join (tID2, NULL); 33. printf ("fine\n"); 34. return 0; 35. } /* main */</pre>
--	---

eseguita t1.7 con 'e' + t1.8

se $t1.10 < t2.16$, il consumatore non entra in ciclo e il carattere 'e' non viene "consumato"

```
// produttore consumatore con semafori
#include <pthread.h> <stdio.h>
<semaphore.h>

1. char buffer;
2. int fine = 0;
3. sem_t pieno, vuoto;
4. void * scrivi (void * arg) {
5.     int i;
6.     for (i = 0; i < 5; i++) {
7.         sem_wait (&vuoto);
8.         buffer = 'a' + i;
9.         if (i == 4) fine = 1;
10.        sem_post (&pieno);
11.    } /* for */
12.    return NULL;
13. } /* scrivi */
```

```
14. void * leggi (void * arg) {
15.     char miobuffer;
16.     int i;
17.     while (fine == 0) {
18.         sem_wait (&pieno);
19.         miobuffer = buffer;
20.         sem_post (&vuoto);
21.         printf ("buffer contiene %.1s \n",
22.             &miobuffer);
23.     } /* while */
24.     sem_getvalue (&pieno, &i);
25.     // controllo per l'ultimo carattere
26.     if (i == 1) {
27.         miobuffer = buffer;
28.         printf ("buffer contiene %.1s \n",
29.             &miobuffer);
30.     } /* if */
31.     return NULL;
32. } /* leggi */
```

- se $t1.9 < t2.17$ viene saltato il ciclo che legge 'e'
- Questo caso è verificato tramite la t2.25 => l'ultima lettura viene rifatta