THREAD o processo leggero (lightweight)

↳ flussi di esecuzione che può essere attivato in parallelo ad altri thread in uno stesso programma o processo. Esiste all'interno di un processo e può creare altri thread che procedono in parallelo (multithreading, può essere applicato a mono/multi-process)

- attività specifica e mirata ✓
- liv. cooperazione elevato e scambio info facilitato ✓
- permette di realizzare attività parallele fortemente interagenti ✓

	PROCESSI	THREAD
CREAZ. E DISTRUZ.	alloc., copia, dealloc di grandi quantità mem	creazione di uno stack (per il thread)
ERRORI	isolati	non isolati, possono danneggiare altri thread
CODICE	può essere modif. med. cambiamento eseguibile	fissato e presente nella sez. text del proc. a cui appartiene
CONDIVISIONE	spreca molto e va implementata da programmatore	automatica perché tutti i thread condividono la mem del proc. a cui appartengono
MUTUA ESCL.	garantita autom. dall'isolam. proprio dei proc.	realizzata da programmatore Med. semafori, mutex, ...
PRESTAZIONI	limitate da overhead	elevate
CONCORRENZA	limitata difficoltà comunic. (da overhead)	elevate

→ i thread realizzano flussi di controllo (esec. seq. di istr.) tram. funzioni

→ condividono lo stesso processo, spazio di indr, dati del processo. Non condividono la pila (o stack)

→ usati per risolvere problemi prog. concorrente (o parallela)

- RICORDO: • se é def $A \geq B$ o vicev. → SEQUENZIALE
  A,B processi • se non def $A \geq B$ → CONCOR. o PARALLELA → REALE, caso multiproc. (NB: in molti casi è simulata perché i thread su proc sfg più di uno)
  <,> ord. sfrtg. • se $\exists$ $A = B$ → PARALLELA REALE → SIMULATA, caso monoproc.

→ esistono diversi tipi di thread anche se condividono l'idea generale:

es. pthread def dallo standard POSIX (insieme di standard per le interfacce applicative

→ è necessario garantire una terminazione coerente dei thread per evitare che il processo a cui appartengono termini prima di loro

→ thread

ANALOGIE CON PROCESSI → 1. funzione di creazione: pthread_create(...) → thread, attributi, puntatore funz. da eseg. (start_routine), arg (di start_routine)
↳ restituisce 0 se true

ident. thread
THREAD(pthread_id, pid_t)
identif. processo

thread group → appartengono allo stesso processo

↳ ogni proc. ha un flusso di controllo che inizia con main() (thread principale) che chiama gli altri.
VAR LOCALI → su stack personale
VAR GLOB/STATICHE → cond. da tutti i thread } difficile gest. multithreading
→ per var. di grandi dim uso dato in void* → punta a qualsiasi var. della dim(int) e può essere conv. a qualunque var.

2. terminazione: pthread_exit(retval) non ritorna nulla
↳ valore da restituire passato come void*
↳ se usato nel main() termina il main ma non i suoi processi
la funzione exit() nel main() termina tutti i processi

3. attesa: pthread_join(...) → identif. thread
→ indir mem valore ritorno (codice terminazione)
↳ in ing uso void** per passare da thread a chiamante un non intero
↳ attesa terminazione prima di pthread_exit(...)

perché doppio puntatore? in generale per terminare/creare thread uso var void* per cui quando la ...-join(...) termina per evitare che punti a una var locale (su stack) che sparirebbe al termine dell'esec. della funzione, faccio puntare il tutto a una var. statica/globale.

PARADIGMA PROD-CONSUMATORE
- genera dati
- utilizza dati del produttore

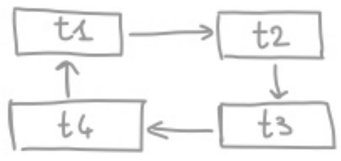
DISACC. PROD-CONS → il prod. genera in maniera indisturbata dati senza aspettare che siano usati mentre il consumatore li usa in maniera seq. creando prob. esecutivi e imprecisioni.

→ è infatti impossibile prevedere quale istr verrà eseguita prima del momento, creando problemi a:
- sez. critiche = succ. seq. di istr. eseguite da thread che non vanno mescolate o interrotte;
- istr atomiche = non vanno interrotte prima della conclusione;
- deadlock (o stallo) = thread_1 aspetta azione di thread_2, che a sua volta aspetta azione da thread_1
- sincronizzazione = voler imporre relazione temporale tra thread

1→ SEQ. CRITICHE → riduco la prog. concorrente, se sto eseg. una seq. blocco gli altri thread.
↳ risolvo con MUTEX (mutua esclusione): pthread_mutex_t m;
costrutto risolutivo, lo dichiaro e inizializzo e metto la seq:
pthread_mutex_lock(&ind); → pthread_mutex_init
⋮
pthread_mutex_unlock(&conti);

2→ ISTR. ATOMICA → tram. mutex, poiché danno orig. a seq. critiche
↳ è possibile implementare anche senza MUTEX usando arg. BLOCCA e un while

3→ DEADLOCK (o STALLO) → errore più grave nella prog. concorrente. La mutua escl. (se un thread entra in sez. critica non vi entra un altro) è garantita se ad esempio vale: t1.8 < inizio t2.7 (e viceversa)
↳ si può creare da sez. critiche risolte senza mutex.



3.1 rappresentazione deadlock tramite grafo orientato

→ i nodi del grafo rappresentano i thread mentre gli archi da i a j rappresentano la richiesta di una risorsa x da parte di t_i a t_j

↳ risolvo bloccandone uno dentro l'altro:

es.
```
void* lockApoiB (void* arg){
    pthread_mutex_loc(&mutexA);
    printf("thread %d: entro in sez. critica 1\n",(int)arg);
    pthread_mutex_lock(&mutexB)        // aspetto esec. B
     printf("....: entro in sez. critica 2\n",...);
     printf("....: termine sez. critica 2\n",...);
    pthread_mutex_unlock(&mutexB)
    printf("....: termine sez critica 1\n",...);
    pthread_mutex_unlock(&mutexA)
    return NULL;
}
```
→ scrivo funzione void* lockBpoiA .... uguale ma invertendo ordine A e B

4→ SINCRONIZZAZIONE → per realizzarla si usa il **SEMAFORO**, costrutto specializzato

Simile ad una variabile che assume valore (pos, neg, nullo):
- inizializzo a valore positivo tramite `sem_init(..)`;
- utilizza solo due funzioni: `sem_wait()` e `sem_post()`

`sem_wait()`: decrementa valore semaforo
- se >0, viene decrement. ma il thread prosegue l'esecuzione
- se ≤0, decrementa e il thread rimane bloccato fino a diventare ≥0 poi prosegue esecuzione

`sem_post()`: incrementa semaforo continuando esec. e sbloccando altri thread (*)

(*) NB: se sem=0 e ci sono su dei thread, prima vengono sbloccati e poi si incrementa;
se sem=0 e non ci sono thread si incrementa direttamente;

→ il valore del semaforo rappresenta il 'num di risorse' disponibili ai thread che lo usano
- se sem>0 rappresenta quanti thread lo possono decrementare senza attesa;
- se sem<0 rappresenta quanti thread sono in attesa che una risorsa si liberi;
- Se sem=0 non ci sono né risorse né thread, ma se uno esegue una wait() rimarrà in attesa;

→ sem=1 può creare mutex

→ nello standard POSIX: si considera sem≥0

es. input: s2='abc', s1='xyz' } risolvo con 2 semafori per bloccare ogni stringa
output: 'axbycz'

```
sem_t sem1, sem2;  // inizializzo sem globalmente
void* tp1 (void* arg) {
    sem_wait(&sem1);
    printf('x');
    sem_post(&sem2); sem_wait(&sem1);   // blocca stampa s1
    printf('y');
    sem_pos(&sem2); sem_wait(&sem1);
    printf('z');
    return NULL;
}
```

faccio andare S2 con stampa

→ ovviamente tp2 per s2 è uguale ma contraria (e nel main creo thread con questi attributi)