

# Corso di Laboratorio di Sistemi Operativi

## Lezione 7

Alessandro Dal Palù

email: `alessandro.dalpalu@unipr.it`

web: `www.unipr.it/~dalpalu`

# Threads

- Un *thread* è l'unità di base per l'utilizzo della CPU.
- Composto da Program Counter, registri, stack.
- Più thread formano un processo classico, e condividono lo stesso spazio di memoria (codici e dati) e le risorse (file, segnali...)
- Utili per computazioni concorrenti.

# Threads

- Vantaggi:
- Context switch veloce: è sufficiente aggiornare poche informazioni (PC, regs e stack!)
- Comunicazioni facili e veloci (condivisione di memoria)
- Svantaggi:
- Le routine di librerie in ambiente multithreading devono essere rientranti (thread-safe), ovvero non causano problemi di condivisione se 2 thread chiamano contemporaneamente la stessa funzione.

# Threads

Esistono 2 implementazioni dei thread:

- Thread a livello *utente*:
  - lo scheduler vede solo i processi.
  - Una libreria utente realizza lo scheduler interno tra threads.
- Thread a livello *kernel*:
  - il sistema operativo offre un insieme di syscall per gestione threads (analoghe a processi).
  - Context switching più costoso della gestione con libreria utente.
  - Esecuzione contemporanea di più thread dello stesso processo.

# Threads e Linux

- Le ultime versioni del kernel forniscono la syscall `clone()` per creare threads a livello kernel.
- Differenza rispetto a Unix.
- La syscall `clone()` specifica di Linux e quindi *non* portabile.
- Si usa uno standard portabile POSIX 1003.1c (pthreads):
- Creazione/terminazione threads
- Sincronizzazione threads: lock, variabili condizione...
- Semafori: POSIX 1003.1b (`<semaphore.h>`)

# Pthreads - creazione

- Un thread è descritto da un intero tid: `pthread_t tid;`  
Il tipo `pthread_t` è dichiarato nell'header file `<pthread.h>`
- Creazione:  
`pthread_create(&tid, &thread_attr, start_routine, arg);`
- `tid` viene caricato con l'identificatore del thread;
- `thread_attr` contiene gli eventuali attributi (generalmente e NULL);
- `start_routine` è il puntatore alla funzione che contiene il codice del thread;
- `arg` parametri dati in pasto a `start_routine`.

# Pthreads - creazione

Esempio:

```
int A, B;
void * codice(void *)
{ /*definizione del codice del thread */ }

main()
{pthread_t t1, t2;
..
pthread_create(&t1,NULL, codice, NULL);
pthread_create(&t2,NULL, codice, NULL);
..
}
```

I due thread appartengono allo stesso processo e *condividono* le variabili globali del programma che li ha generati (ad esempio A e B).

# Pthreads - exit e join

- `pthread_exit(&ret_val)`
- Termina l'esecuzione di un thread, passa un valore di ritorno (`ret_val`) che il chiamante può catturare con `pthread_join()`
- `pthread_join(tid, &ret_val)`
- Attende il termine dell'esecuzione di `thr_name`
- Legge (`ret_val`) il valore di ritorno del thread terminato. Se la funzione termina correttamente il valore di ritorno è 0



# Pthreads - Esercizio

- Scaricare dal sito il file `exCreazioneThread.c`
- Studiare il codice.
- Provare a compilare ed eseguire.
- Tempo 15 minuti.

# Pthreads - Mutex

- Un Mutex (o lock, semaforo binario...) viene utilizzato per sincronizzare l'accesso (in mutua esclusione) ad una sezione critica da parte di più thread.
- Per esempio si protegge l'utilizzo di una risorsa non condivisibile o una modifica a variabili condivise.
- Il valore del mutex può essere 0 (risorsa occupata) o 1 (risorsa libera).
- In Pthreads lock richiede la risorsa e unlock la rilascia.

# Pthreads - Mutex - chiamate

- `pthread_mutex_t mutex_name;`  
Definisce una variabile mutex.
- `pthread_mutex_init(&mutex_name, mutex_attr)`  
Crea il mutex (default: libero). `mutex_attr=NULL`.
- `pthread_mutex_lock(&mutex_name)`  
Blocca `mutex_name`. Se il mutex e già in lock il thread viene sospeso in attesa dellunlock.
- `pthread_mutex_trylock(&mutex_name)`  
Prova a mettere in lock `mutex_name` senza bloccarsi. Ritorna 0 se è riuscito, altrimenti ritorna EBUSY
- `pthread_mutex_unlock(&mutex_name)`  
Sblocca `mutex_name`
- `pthread_mutex_destroy(&mutex_name)`  
Elimina `mutex_name`

# Mutex esercizio

- Scaricare dal sito il file `mutex.c`
- Studiare il codice.
- Provare a compilare ed eseguire.
- Tempo 15 minuti.

# Semafori

- Si usa la libreria `semaphore.h`
- `sem_init`: inizializzazione di un semaforo
- `sem_wait`: wait
- `sem_post`: signal
- `sem_t`: tipo di dato associato al semaforo;

# Semafori - chiamate

- `sem_t my_sem;`  
Definisce una variabile di tipo semaforo.
- `sem_init(sem_t* my_sem, 0, unsigned int value)`  
Attribuisce il valore iniziale del semaforo. `value` il valore iniziale.
- `int sem_wait(sem_t *sem)`  
*wait* di Dijkstra: se il valore del semaforo è 0, sospende il thread chiamante nella coda associata al semaforo; altrimenti ne decrementa il valore.
- `int sem_post(sem_t *sem)`  
*signal* di Dijkstra: se c'è almeno un thread sospeso nella coda associata al semaforo `sem`, viene risvegliato; altrimenti il valore del semaforo viene incrementato.

# Semafori - esercizio

- Scaricare dal sito il file `semthread.c`
- Studiare il codice.
- Provare a compilare ed eseguire.
- Tempo 15 minuti.

# Refs

<https://computing.llnl.gov/tutorials/pthreads/>

<http://yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html>

Google, google, google!!!



# Esercizio con pthreads

Implementare il problema dei lettori e scrittori utilizzando semafori e mutex forniti da pthreads.

Generare 1 scrittore che scrive su un vettore (condiviso!) `int A[100]`. Ogni 5 secondi lo scrittore scrive un numero in una locazione di `A[]`.

Creare 2 thread che ad intervalli regolari e diversi leggono e stampano a video il contenuto di `A`.

# Esercizio con pthreads (pseudo codice)

```
semafori mutex=1, wsem=1;  
int rc=0; //readers counter
```

Reader		Writer
down(mutex);		down(wsem);
rc++;		
if rc=1 then down(wsem);		
up(mutex);		
readdata();		writeadta();
down(mutex);		up(wsem);
rc--;		
if rc=0 then up(wsem);		
up(mutex);		