SAPIENZA
UNIVERSITÀ DI ROMA

**Corso di laurea in Ingegneria dell'Informazione**
**Indirizzo Informatica**

Reti e sistemi operativi

# I processi: creazione, terminazione, IPC. Introduzione al multithreading.

# Creazione dei processi (1/2)

- **I processi sono creati da altri processi**
  - Processo creante: **parent**
  - Processo creato: **children**
- Processo creante e creato possono:
  - Condividere tutte, un sottoinsieme oppure nessuna risorsa (es. file, memoria, ecc...).
  - Essere eseguiti concorrentemente (in “parallelo”) oppure il processo padre può attendere la terminazione del processo figlio.

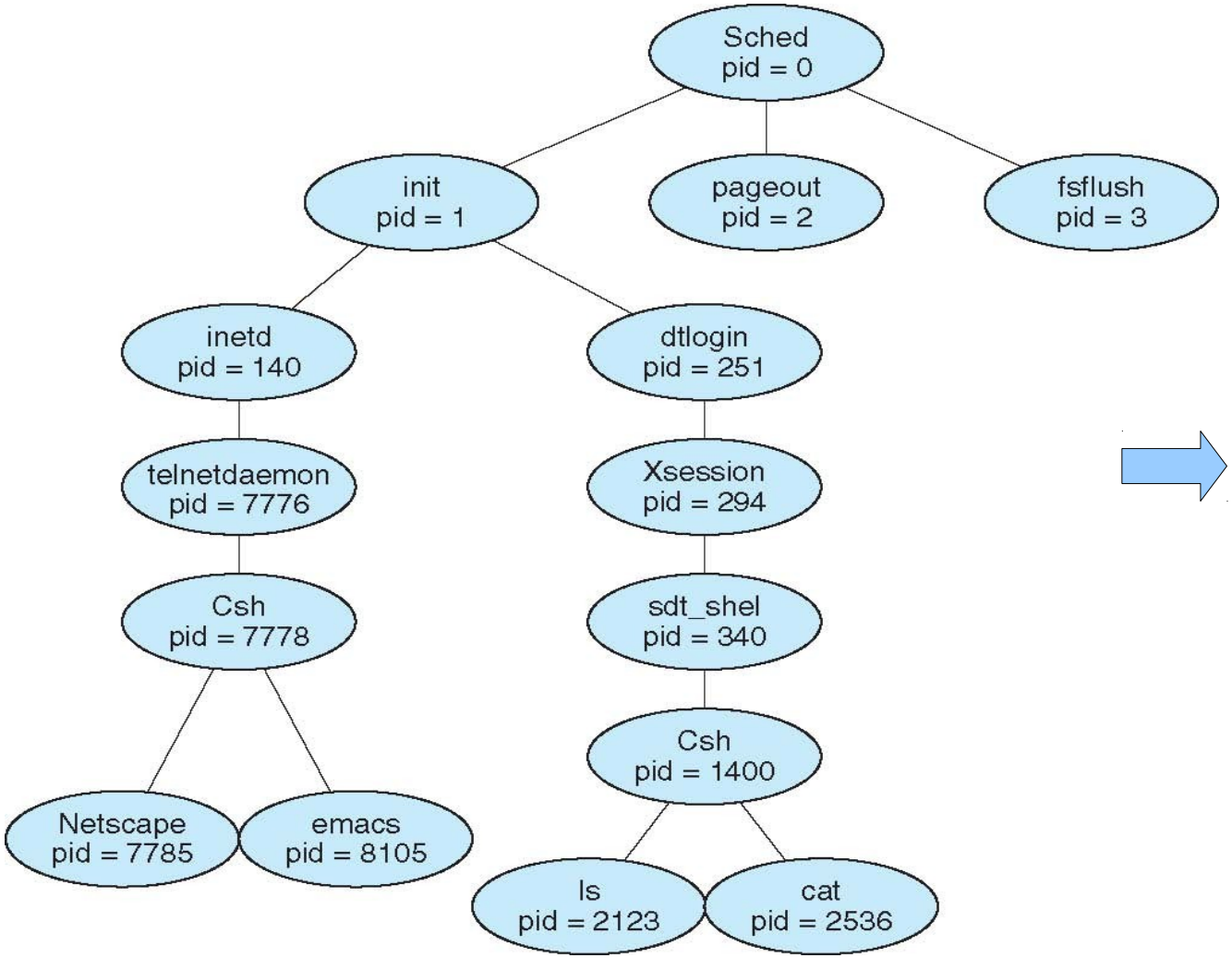
# Creazione dei processi (2/2)

- La memoria assegnata al processo figlio (spesso definita *address space*) alla creazione è di solito un duplicato della memoria assegnata al processo padre. In particolare, entrambi stanno eseguendo lo stesso programma
  - In realtà nei sistemi operativi attuali questa copia non viene più effettuata automaticamente
- E' ovviamente possibile assegnare al processo figlio un programma diverso

# Terminazione di processi

- Quando un processo termina la sua esecuzione (implicitamente o esplicitamente, ad esempio attraverso la syscall **exit**(), o indirettamente, eccezione o richiesta esterna):
  - Tutte le resorse vengono deallocate, i file chiusi, ecc...
  - Può restituire un valore (tipicamente un intero) al padre (syscall **wait()**)
- Molti sistemi operativi non permettono ai processi figli di continuare nell'esecuzione se il processo padre termina → **chiusura in cascata**

# Processi in Unix/Linux (1/3)



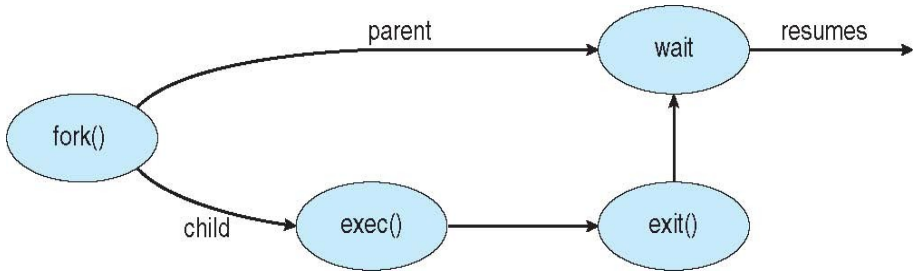

Da terminale:
**ps -ejH**

# Processi in Unix/Linux (2/3)

- Creazione di processi: syscall **fork()**. Il processo figlio è una copia del processo padre con qualche eccezione:
  - Ha un PID diverso, e mantiene il PID del parent
  - I contatori di utilizzo delle risorse sono resettati
  - Il figlio non eredita nessun timer, e non resta in attesa di eventuali eventi, ecc.... attesi dal padre.
- Esecuzione di un nuovo programma: syscall **exec()**, di solito chiamata subito dopo fork().
  - Carica in memoria (nel segmento **Text**) del processo chiamante un nuovo programma → il contenuto precedente viene distrutto!
  - Inizia ad eseguire il nuovo programma

# Processi in Unix/Linux (3/3)

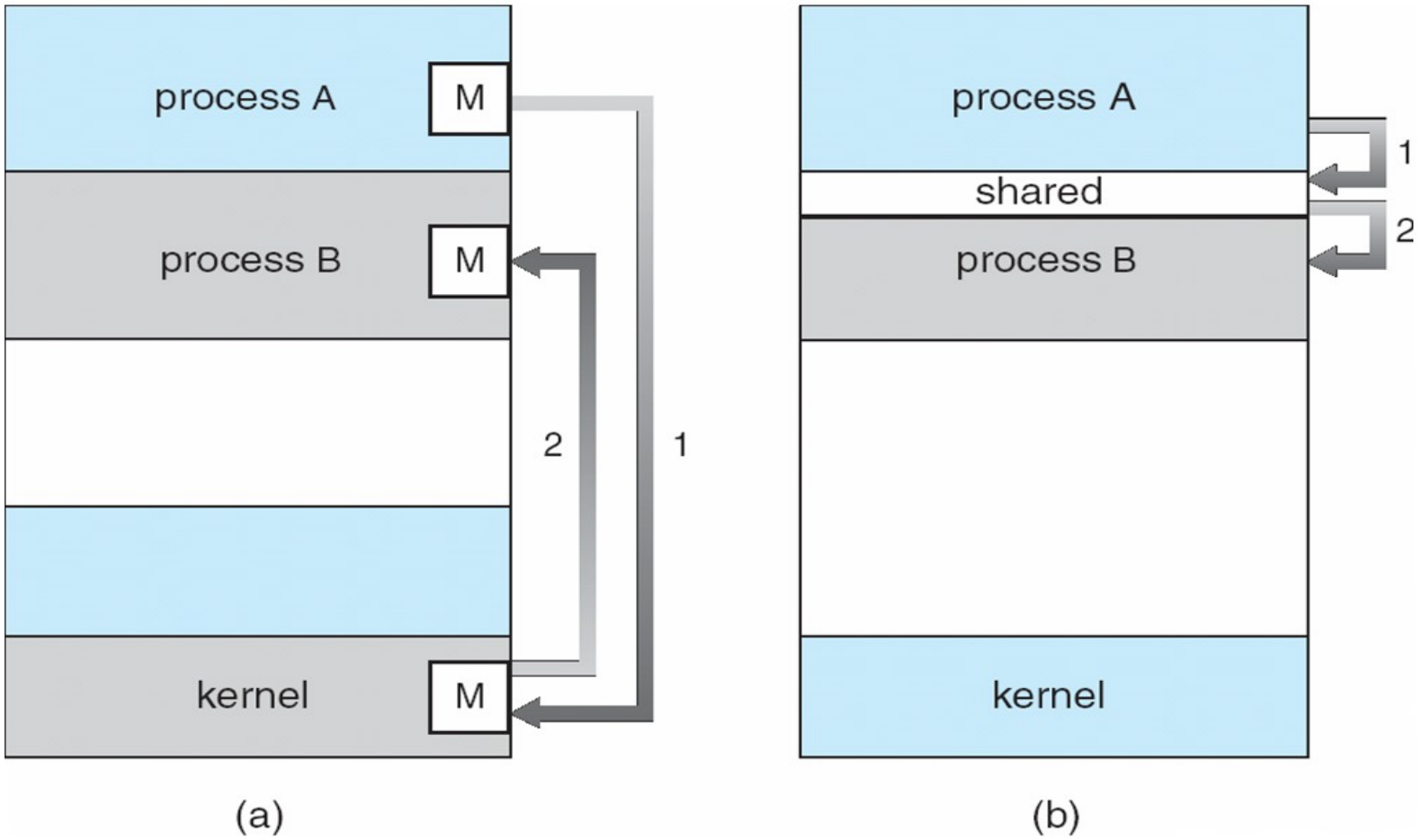
```
#include <sys/types.h>
#include <studio.h>
#include <unistd.h>
int main()
{
pid_t  pid;
     /* fork another process */
     pid = fork();
     if (pid < 0) { /* error occurred */
          fprintf(stderr, "Fork Failed");
          return 1;
     }
     else if (pid == 0) { /* child process */
          execlp("/bin/ls", "ls", NULL);
     }
     else { /* parent process */
          /* parent will wait for the child */
          wait (NULL);
          printf ("Child Complete");
     }
     return 0;
}
```



# Inter process communication (IPC)

- I processi spesso hanno bisogno di comunicare con altri processi:
  - Condivisione delle informazioni
  - Aumento dell'efficienza computazionale (in casi si sistemi multicore/multi processor)
  - Modularità
  - Controllo reciproco (robustezza agli errori)
  - …
- IPC di base:
  - Memoria condivisa
  - Passaggio di messaggi

# Modelli di comunicazione

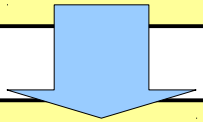


# Shared Memory in UNIX

- Creazione di un segmento di memoria condivisa:
  - segment id = shmget(IPC PRIVATE, size, S IRUSR | S IWUSR);
- Un altro processo che desidera accedere a tale segmento di memoria condivisa deve **conoscere il segment_id** ed eseguire l'attach:
  - shared memory = (char *) shmat(id, NULL, 0);
  - shmdt(shared memory); // detach

# Esempio: Produttore-consumatore

```
while (true) {
   /* Produce an item */

   while (((in = (in + 1) %
       BUFFER SIZE count)  == out)

     ;   /* do nothing */

    buffer[in] = item;

    in = (in + 1) % BUFFER SIZE;

}
```



```
while (true) {

     while (in == out)

            ; // do nothing

     // remove an item from the buffer

     item = buffer[out];

     out = (out + 1) % BUFFER SIZE;

     return item;

}
```

# Passaggio di messaggi

- A differenza della memoria condivisa è necessario ogni volta l'intervento del S.O.→**maggiore overhead**
- **Implementazione più semplice**
- 2 operazioni (syscall):
  - **send**(destinatario/mailbox, messagio)
  - **receive**(mittente/mailbox, messaggio)
- Comunicazione:
  - Sincrona o asincrona (chiamate bloccanti e non bloccanti)
  - Diretta o indiretta (link o mailbox)
  - Limitata o illimitata

# Limitazione dei processi (1/2)

- Si supponga di voler implementare un programma che controlli N attivatori (es., i motori dei singoli giunti di un robot manipolatore), legga gli encoder degli N giunti per effettuare un controllo in retroazione, legga l'input proveniente da una sonda installata sul giunto terminale del manipolatore, sia in attesa di comandi provenienti da un'interfaccia testuale e da un cavo di rete.
- Il programma in questione dovrebbe essere in grado di effettuare N chiamate a funzioni(syscall)di tipo **write()** in parallelo sugli attuatori, N chiamate a funzioni **read()** in parallelo sugli encoder, ricalcolare e rigenerare di conseguenza i controlli, e così via. In parallelo dovrebbe inoltre leggere i dati dalla sonda, e restare in attesa sia di comandi inseriti da tastiera, sia di comandi provenienti dal cavo di rete.

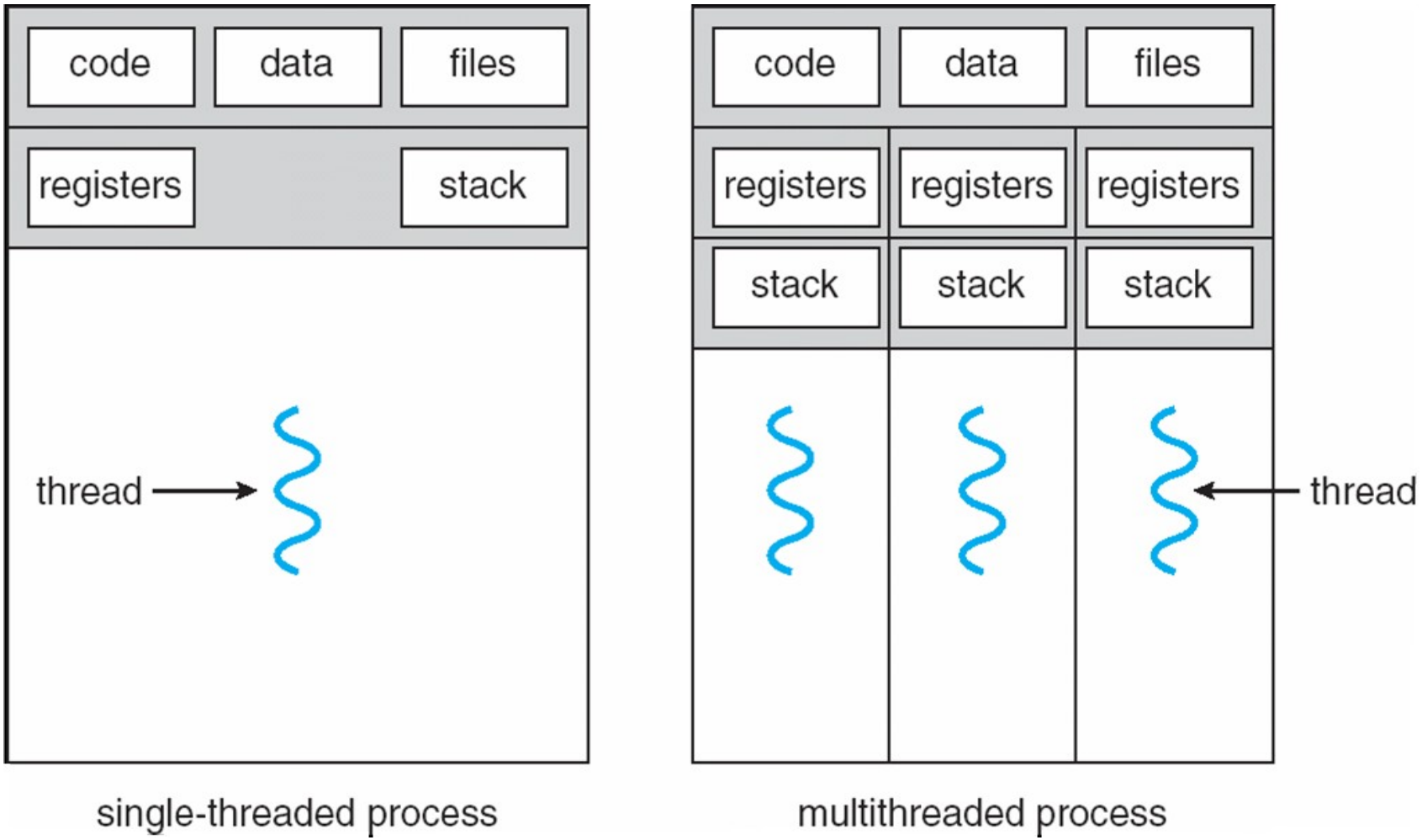
# Limitazione dei processi (2/2)

- Purtroppo ogni programma viene eseguito sequenzialmente: quando ad esempio verrà chiamata una funzione **scanf()** o simile per leggere da tastiera, o la funzione (syscall) **read()** per leggere dati da un encoder, il **programma sospederà la sua esecuzione finchè l'input atteso non venga correttamente inserito, bloccando di conseguenza ogni altra attività** (es., spedizione dei comandi agli attivatori).
- Si potrebbe pensare di implementare un programma associato ad ogni azione (lettura, scrittura, calcolo, ecc..) ma questo porterebbe a due problemi:
  - IPC complesso e necessariamente sincronizzato
  - Overhead nella creazione di nuovi processi e nei molti context switch

# I thread (1/3)

- I thread sono dei ”mini-processi” contenuti in un processo convenzionale (detto ”processo padre”): essi possono essere definiti ”**lightweight process**” perché **condividono lo spazio di memoria del processo padre** (es., possono accedere allo heap e ai dati globali del padre) e possiedono solo un sottoinsieme di dati indipendenti che ne caratterizzano lo stato (essenzialmente, un proprio stack ed i registri utilizzati, tra cui PC e SP).
- La schedulazione è simile a quella dei processi convenzionali: i thread possiedono un loro flusso di esecuzione (ovvero, un loro program counter ed un loro stack) e possono essere assegnate priorità differenti a thread che appartengono ad uno stesso processo.

# I thread (2/3)




single-threaded process

multithreaded process

# I thread (3/3)

- La creazione ed il context switch dei thread è **molto più rapido** rispetto ai processi convenzionali (es. ~20 e ~5 volte più rapida, rispettivamente).
- **La condivisione di dati tra vari thread è semplice ed immediata**: basterà infatti accedere da essi a variabili globali o locazioni dell'heap. E' necessario prestare attenzione alla sincronizzazione di questi accessi.
- Quando il processo padre termina, termineranno tutti i suoi thread.

# Multithreading

- L'esempio precedente può essere implementato in maniera semplice e diretta usando il **paradigma del multithreading**: si creano N thread, ognuno messo in lettura su di un determinato encoder. Tali thread scriveranno i dati letti su una memoria condivisa che sarà letta da un thread che, data l'azione corrente da portare a termine, eseguirà ciclicamente il controllo in retroazione e si prenderà carico di spedire i rispettivi segnali ai singoli attivatori.
- Altri due thread si prenderanno carico di leggere, rispettivamente, i comandi inseriti da tastiera e provenienti dal cavo di rete, modificando di conseguenza l'azione che il manipolatore dovrà eseguire. Un ultimo thread leggerà ciclicamente la sonda installata, eventualmente notificando a video le letture.

# Benefici del multithreading

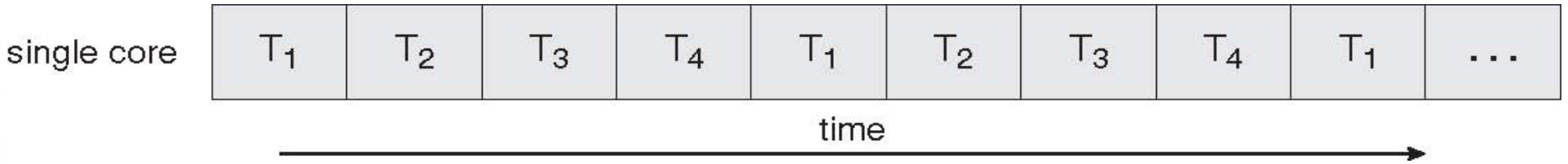
- Efficienza computazionale
- Condivisione delle risorse semplice e diretta
- Risparmio di risorse
- Scalabilità

# Esecuzione parallela di thread (1/2)

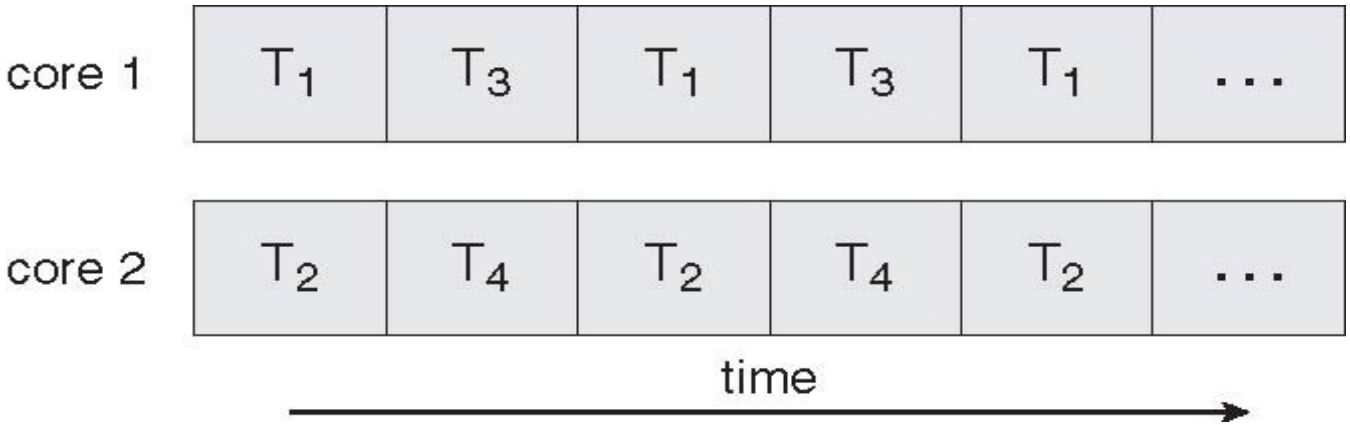
- La diffusione dei sistemi multicore ha fornito una spinta ulteriore all'utilizzo massivo del multithreading.
- Problemi da risolvere nel multithreading su sistemi multicore:
  - Dividere e bilanciare le attività computazionali
  - Dividere i dati evitando cross-dipendenze
  - Testing e debugging difficoltoso

# Esecuzione parallela di thread (2/2)

- Sistema mono-core




- Sistema dual-core



# Thread in Unix/Linux: pthread (1/2)

- Standarizzazione POSIX (Portable Operating System Interface for Unix):
  - IEEE Std 1003.1 (Institute of Electrical and Electronics Engineers).
- Riferimenti:
  - POSIX Threads Programming Tutorial
    - https://computing.llnl.gov/tutorials/pthreads/
  - D. Butenhof, “Programming With POSIX Threads”, Addison Wesley

# Thread in Unix/Linux: pthread (2/2)

- I pthread sono definiti per il linguaggio C, per utilizzarli sarà necessario:
  - Includere l'header di libreria
    - **#include <pthread.h>**
  - Compilare linkando la libreria:
    - **gcc -L/usr/unclude -lpthread ...** oppure:
    - **gcc -pthread …**
- In Linux, i thread creati saranno trattati dal sistema operativo in maniera simile agli altri processi (salvo possedere le caratteristiche citate in precedenza). Se ad esempio l'eseguibile **test_thread** crea un thread, lanciando ps aux si potranno trovare due processi di nome **test_thread** con ID diverso.

# Creazione di un thread

- I thread vanno esplicitamente creati all'interno del programma chiamando la funzione:
  - **int pthread_create(** pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg **)**
    - **pthread_t *thread** : indentificatore del thread (puntatore al tipo ”opaco” pthread_t)
    - **pthread_attr_t *attr**: puntatore alla ”struttura” di tipo ”opaco” pthread_attr_t con i parametri del thread (se NULL, il thread verrà creato con i parametri di default).
    - **void *(*start_routine)(void *)**: puntatore alla funzione che il thread dovrà eseguire, ovvero il ”programma” eseguito dal thread.
    - **void *arg**: puntatore generico con gli argomenti da passare alla funzione che il thread dovrà eseguire.
- pthread_create() restituisce 0 solo in caso di successo.

# Terminazione di un thread

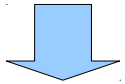
- Per terminare esplicitamente un thread, si chiama la funzione:
  - **void pthread_exit( void *value_ptr )**
    - **void *value_ptr**: valore di ritorno del thread (può essere ottenuto attraverso la funzione join)
- Solitamente non è indispensabile chiamare pthread_exit() dalle funzioni associate ai thread: pthread_exit() viene implicitamente chiamata al termine delle stesse. Per quel che riguarda il main, è invece necessario chiamare pthread_exit() per avere la certezza che tutti i thread abbiano terminato la loro esecuzione.

# ID univoco associato ad un thread

- Come detto, Linux tratta i thread in maniera del tutto similare ai processi convenzionali: come per questi ultimi, ad ogni thread sarà associato un ID univoco all'interno del sistema. E' possibile conoscere l'identificativo associato ad un thread (di tipo ”opaco” pthread_t) chiamando all'interno della routine associata al thread la funzione:
  - **pthread_t pthread_self(void);**
- Ovviamente anche la funzione main() potrà chiamare tale funzione.

# Pthread: un primo esempio (1/2)

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#define NUM_THREADS     5

void *threadFunc(void *thread_str)
{
   char *t_str = (char *)thread_str;
   printf("Stringa passata come \
     Argomento al thread: %s\n", t_str);
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   char *thread_str[NUM_THREADS];
   long unsigned int t;
```



```
   for(t=0; t<NUM_THREADS; t++)
   {
      thread_str[t] = (char *)malloc(11);
      sprintf(thread_str[t],"Codice n.%.1ld",t);
      printf("Creazione thread n. %ld\n", t);
      int rc = pthread_create(&threads[t], NULL,
        threadFunc, (void *)thread_str[t]);
      if (rc)
      {
         fprintf(stderr, "pthread_create() failed! \
             Return code: %d\n", rc);
         exit(EXIT_FAILURE);
      }
   }
   pthread_exit(NULL);
}
```

# Pthread: un primo esempio (2/2)

- Un possibile output:

```
Creazione thread n. 0
Creazione thread n. 1
Creazione thread n. 2
Creazione thread n. 3
Creazione thread n. 4
Stringa passata come argomento al thread: Codice n.0
Stringa passata come argomento al thread: Codice n.2
Stringa passata come argomento al thread: Codice n.4
Stringa passata come argomento al thread: Codice n.3
Stringa passata come argomento al thread: Codice n.1
```

# Sincronizzazione di thread: join

- Un thread (ovviamente anche la funzione main()) può **rimanere in attesa del completamento di un altro thread** invocando la funzione:
  - **int pthread_join(pthread_t thread, void **value_ptr);**
    - **pthread_t *thread:** identificatore del thread di cui attendere la terminazione
    - **void **value:** valore restituito dal thread che termina
- Il thread che invoca pthread_join() si blocca finché lo specifico thread identificato da **pthread_t thread** non termina.
- pthread_join() ritorna 0 in caso di successo.

# Join su esempio precedente (1/2)

- Basterà aggiungere le seguenti righe di codice prima di pthread_exit(NULL):

```
void *status;
for(t=0; t<NUM_THREADS; t++)
{
  int rc = pthread_join(threads[t], &status);
  if (rc)
  {
    printf("pthread_join() failed! ! return code: %d\n", rc);
    exit(EXIT_FAILURE);
  }
  printf("Join con thread %ld completato, stato ritornato : %ld\n",t,
            (long)status);
}

pthread_exit(NULL);
```

# Join su esempio precedente (2/2)

- Un possibile output:

```
Creazione thread n. 0
Creazione thread n. 1
Creazione thread n. 2
Creazione thread n. 3
Creazione thread n. 4
Stringa passata come argomento al thread: Codice n.2
Stringa passata come argomento al thread: Codice n.4
Stringa passata come argomento al thread: Codice n.0
Join con thread 0 completato, stato ritornato : 0
Stringa passata come argomento al thread: Codice n.1
Join con thread 1 completato, stato ritornato : 0
Join con thread 2 completato, stato ritornato : 0
Stringa passata come argomento al thread: Codice n.3
Join con thread 3 completato, stato ritornato : 0
Join con thread 4 completato, stato ritornato : 0
```

# Impostazione attributo di join

- Per default, tutti i thread creati sono creati "*joinable*" (ovvero, è possibile sincronizzarsi con l'uscita degli stessi attraverso la funzione pthread_join()).
- Per esplicitare questo fatto, è necessario settare l'attributo PTHREAD_CREATE_JOINABLE:

```
pthread_attr_t attr; // Tipo (opaco) che rappresenta i parametri del thread

pthread_attr_init(&attr);  // inizializzazione della struttura dei parametri
                           // ed assegnamento valori di default
/* Setting del parametro */
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
/* Creazione del thread con passaggio parametri */
Int  rc = pthread_create(&thread[t], &attr, func, NULL);
...
pthread_attr_destroy(&attr);  // Rilascio risorse occupate dai parametri
```

# Presentazione homework 1 (entro h.24:00 del 14 Aprile, facoltativo)

- Implementare in C l'esempio del produttore/consumatore visto a lezione (vedi anche libro di testo) usando i pthread:
  - Dovranno essere lanciati due thread, un “produttore” ed un “consumatore”, rispettivamente (il dato prodotto potrà ad esempio essere un intero crescente, eventualmente utilizzare uno sleep() di un secondo ad ogni dato prodotto e ricevuto per evitare di saturare il sistema).
  - Ad ogni dato prodotto e consumato, i thread dovranno fornire un output a video
  - Il programma principale dovrà attendere un input da tastiera da parte dell'utente. Una volta ricevuto un input qualsiasi, esso dovrà notificare ai thread di auto-terminarsi e, prima di uscire, attenderne la terminazione effettiva.
- Si ricordi che ogni thread può accedere alle variabili globali e a porzioni di memoria (se ne conosce l'indirizzo) dell'heap del processo creante
- Altre informazioni e consegna attraverso Moodle