Sistemi Operativi 1

AA 2021/2022

Processi e Threads

Contenuti

- Concetto di processo
- Stati di un processo
- Thread
- Operazioni e relazioni tra processi
- Comunicazione tra processi
- Gestione dei processi del S.O.

CONCETTO DI PROCESSO

Programma e processo

- Processo = istanza di programma in esecuzione
  - programma = concetto statico
  - processo = concetto dinamico
- Processo eseguito in modo sequenziale
  - Un'istruzione alla volta ma…
- … in un sistema multiprogrammato i processi evolvono in modo ***concorrente***
  - Risorse (fisiche e logiche) limitate
  - Il S.O. stesso consiste di più processi

Immagine in memoria

- Processo consiste di:

- Istruzioni (sezione Codice o Testo)
  - Parte statica del codice
- Dati (sezione Dati)
  - Variabili globali
- Stack
  - Chiamate a procedura e parametri
  - Variabili locali
- Heap
  - Memoria allocata dinamica
- Attributi (id, stato, controllo)

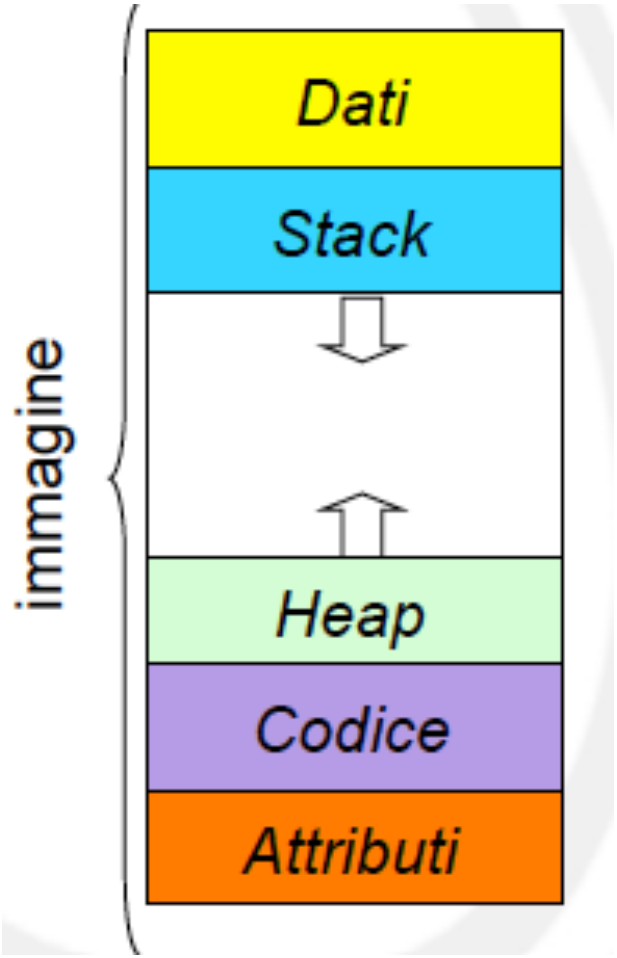
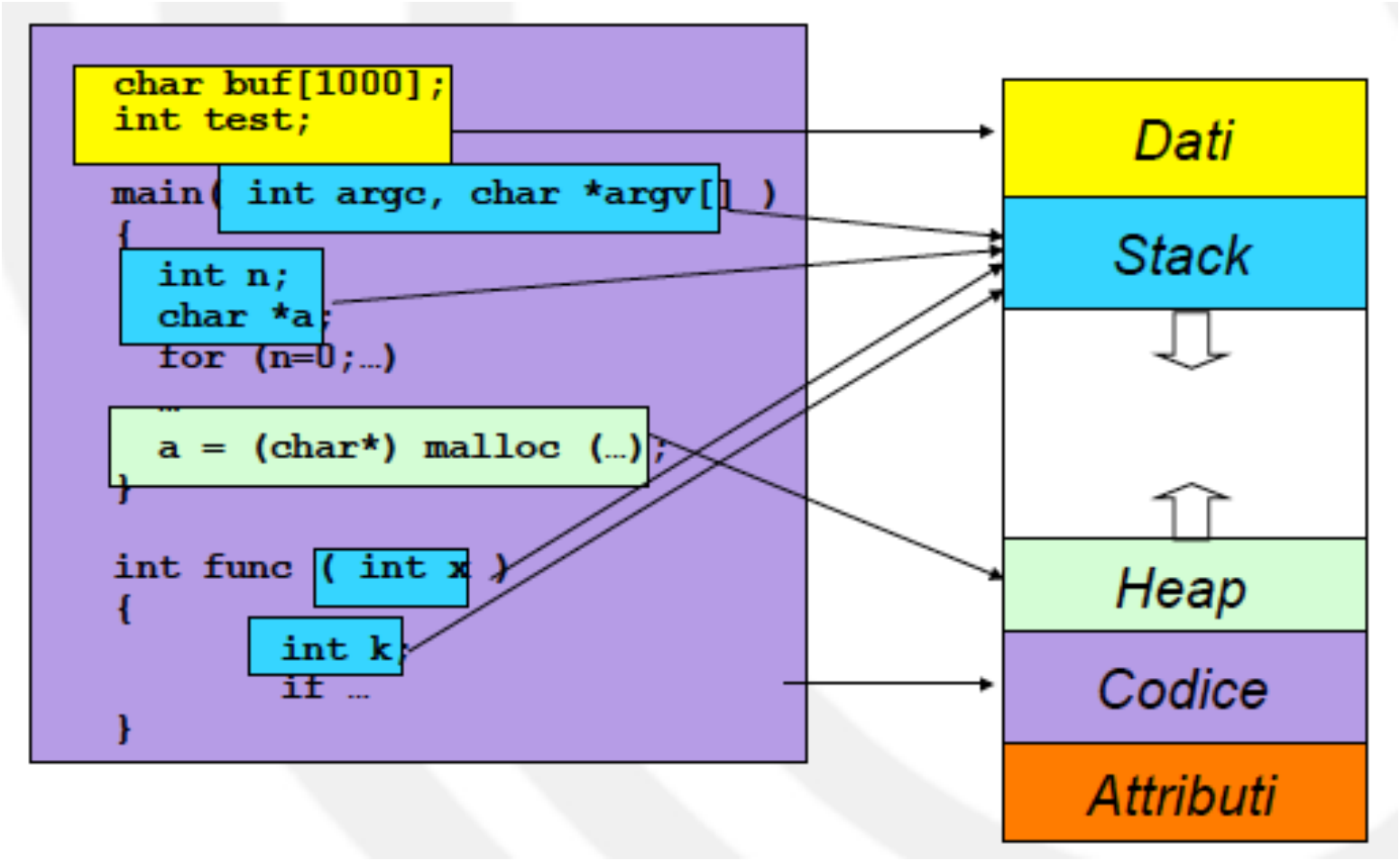
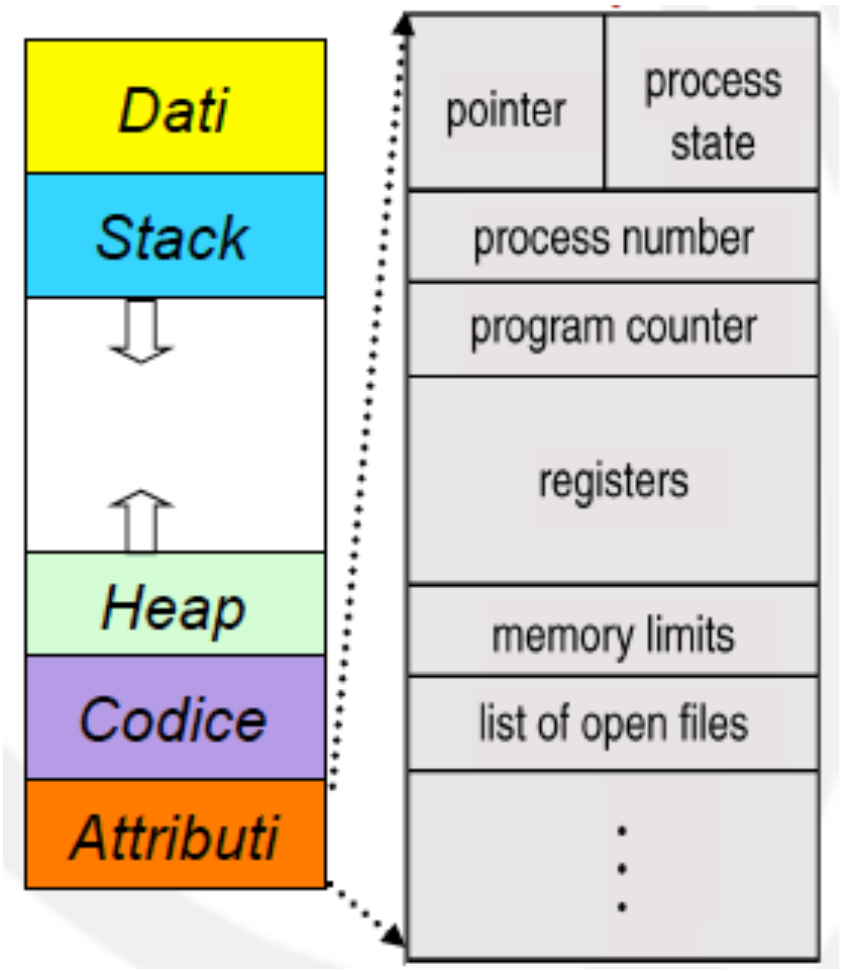


Immagine in memoria



Attributi (Process Control Block)

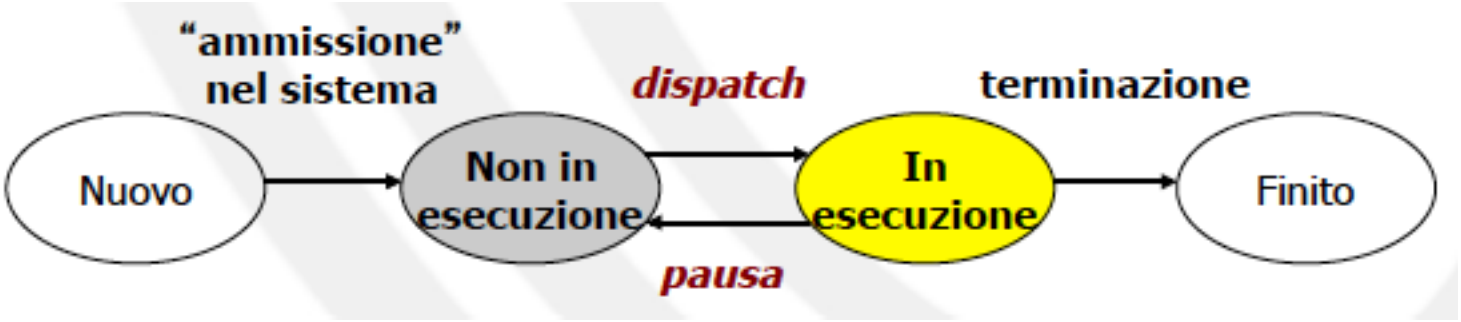
- All'interno del S.O. ogni processo è rappresentato dal process control block (PCB)
  - stato del processo
  - program counter
  - valori dei registri
  - informazioni sulla memoria (es:. registri limite, tabella pagine)
  - informazioni sullo stato dell'I/O (es:. richieste pendenti, file)
  - informazioni sull'utilizzo del sistema (CPU)
  - informazioni di scheduling (es.priorità)



STATI DI UN PROCESSO

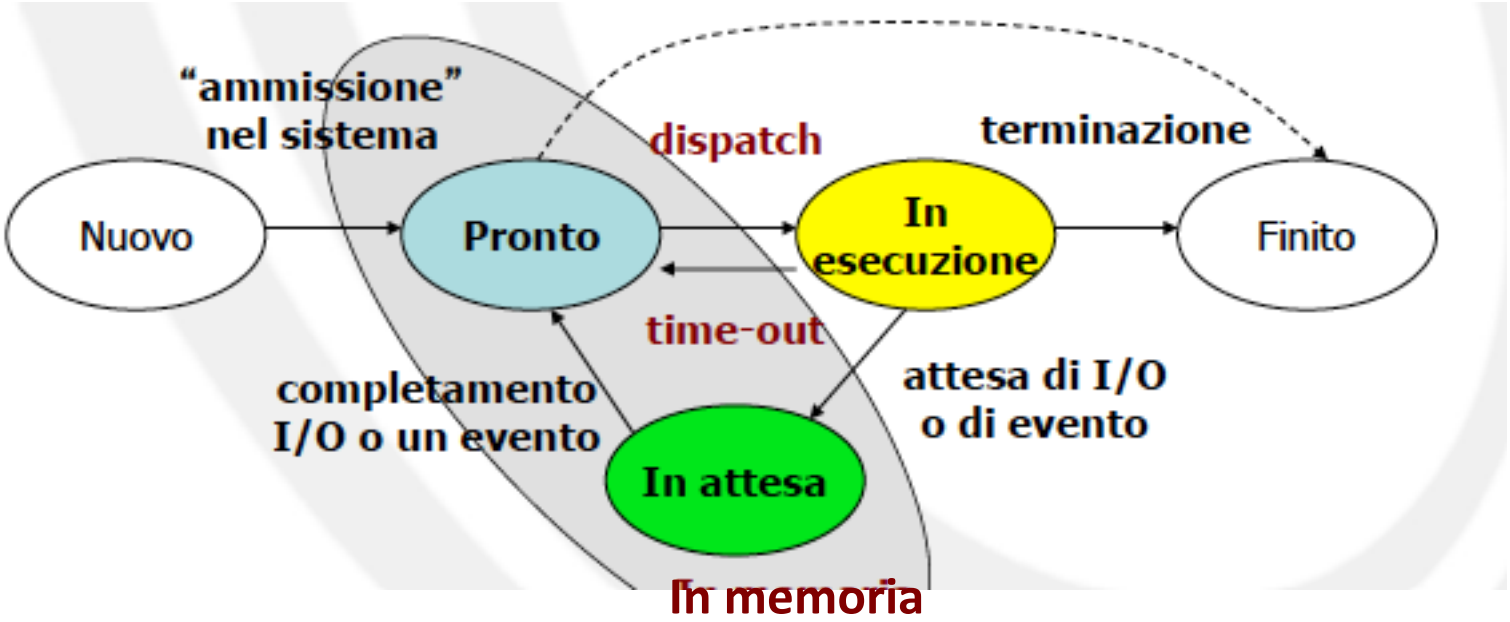
Stato di un processo

- Durante la sua esecuzione, un processo evolve attraverso diversi stati
  - Diagramma degli stati diverso per S.O. diversi
- Lo schema base è il seguente:



Stati di un processo

- Evoluzione di un processo (schema con stato di attesa)



Scheduling

- Selezione del processo da eseguire nella CPU al fine di garantire:
  - Multiprogrammazione
    - Obiettivo: massimizzare uso della CPU → più di un processo in memoria
  - Time-sharing
    - Obiettivo: commutare frequentemente la CPU tra processi in modo che ognuno creda di avere la CPU tutta per sè

Scheduling

- **Long-term scheduler** (o job scheduler) – seleziona quali processi devono essere trasferiti nella coda dei processi pronti

- **Short-term scheduler** (o CPU scheduler) – seleziona quali sono i prossimi processi ad essere eseguiti e alloca la CPU di conseguenza

Scheduling

- Short-term scheduler é invocato molto di frequente (milliseconds) $\Rightarrow$ (deve essere veloce)

- Long-term scheduler è invocato meno frequentemente (seconds, minutes) $\Rightarrow$ (può essere lento)
- Il long-term scheduler controlla il *grado di multiprogrammazione*

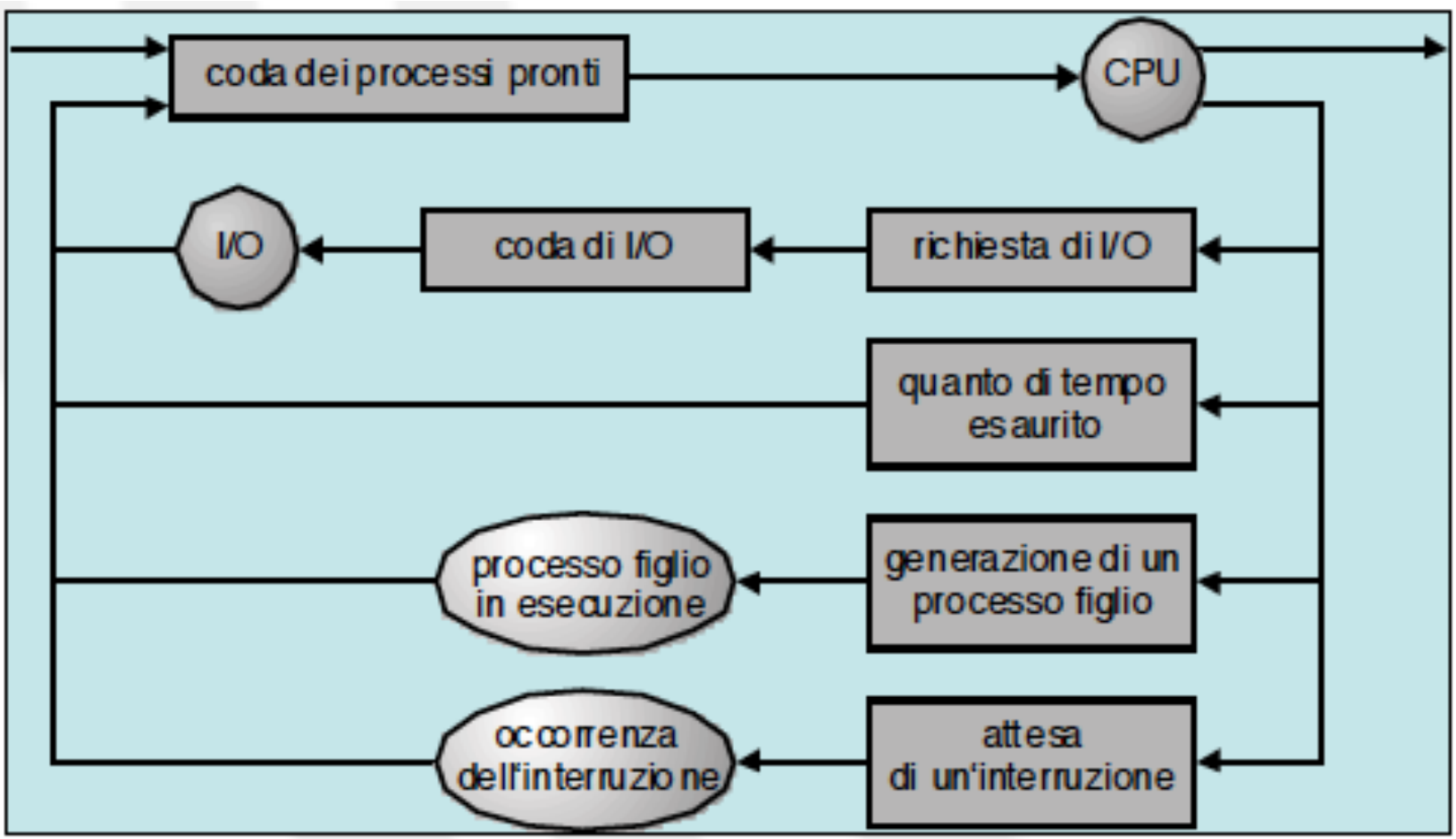
Code di scheduling

- Ogni processo è inserito in una serie di code:
  - Coda dei processi pronti (ready queue)
    - coda dei processi pronti per l'esecuzione
  - Coda di un dispositivo
    - coda dei processi in attesa che il dispositivo si liberi

Code di scheduling

- All'inizio il processo è nella ready queue fino a quando non viene selezionato per essere eseguito (dispatched)
- Durante l'esecuzione può succedere che:
  - Il processo necessita di I/O e viene inserito in una coda di un dispositivo
  - Il processo termina il quanto di tempo, viene rimosso forzatamente dalla CPU e re-inserito nella ready queue
  - Il processo crea un figlio e ne attende la terminazione
  - Il processo si mette in attesa di un evento

Diagramma di accodamento



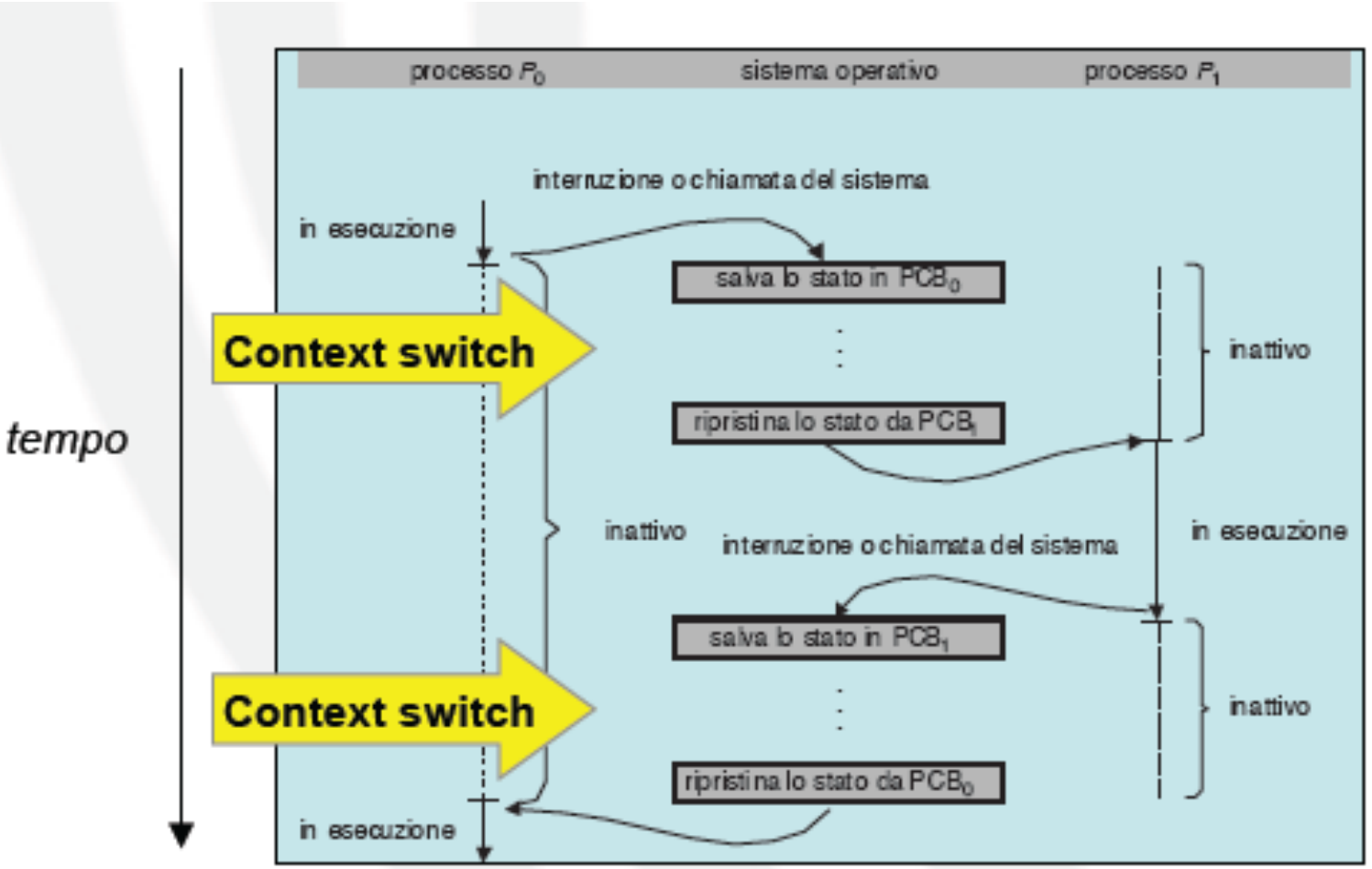
Operazione di dispatch

1. Cambio di contesto
   - salvataggio PCB del processo che esce e caricamento del PCB del processo che entra
2. Passaggio alla modalità utente
   - (all'inizio della fase di dispatch il sistema si trova in modalità kernel)
3. Salto all'istruzione da eseguire del processo appena arrivato nella CPU

Cambio di contesto (context switch)

- Passaggio della CPU a un nuovo processo
  - Registrazione dello stato del processo vecchio e caricamento dello stato (precedentemente registrato) del nuovo processo
  - Il tempo necessario al cambio di contesto è puro sovraccarico
    - Il sistema non compie alcun lavoro utile durante la commutazione
    - La durata del cambio di contesto dipende molto dall'architettura

Commutazione della CPU



OPERAZIONI SUI PROCESSI

UNIVERSITY
OF TRENTO

Creazione di un processo

- Un processo può creare un figlio
  - Figlio ottiene risorse dal S.O. o dal padre (spartizione, condivisione)
  - Tipi di esecuzione
    - Sincrona
      - Padre attende la terminazione dei figli
    - Asincrona
      - Evoluzione “parallela” (concorrente) di padre e figli

Creazione di un processo (UNIX)

- System call fork
  - Crea un figlio che è un duplicato esatto del padre
- System call exec
  - Carica sul figlio un programma diverso da quello del padre
- System call wait
  - Per esecuzione sincrona tra padre e figlio

Creazione di un processo (UNIX)

```
#include <stdio.h>
void main(int argc, char *argv[]){
        int pid;
        pid = fork(); /* genera un nuovo processo */
        if (pid < 0) { /* errore */
                fprintf(stderr, "Errore di creazione");
                exit(-1);
        } else if (pid == 0) { /* codice del figlio */
                execlp("/bin/ls", "ls", NULL);
        } else { /* codice del padre */
                wait(NULL); /* padre attende il figlio */
                printf("Figlio ha terminato.");
                exit(0);
        }
}
```

Terminazione di un processo

- Processo finisce la sua esecuzione
- Processo terminato forzatamente dal padre
  - Per eccesso nell'uso delle risorse
  - Il compito richiesto al figlio non è più necessario
  - Il padre termina e il S.O. non permette ai figli di sopravvivere al padre
- Processo terminato forzatamente dal S.O.
  - Utente chiude applicazione
  - Errori (aritmetici, di protezione, di memoria, …)

THREADS – CONCETTO DI THREAD

Processo e thread

- Un processo unisce due concetti
  - Il possesso delle risorse
    - Es.: spazio di memoria, file, I/O…
  - L'utilizzo della CPU (esecuzione)
    - Es.: priorità, stato, registri…
- Queste due caratteristiche sono indipendenti e possono essere considerate separatamente
  - Thread = unità minima di utilizzo della CPU
  - Processo = unità minima di possesso delle risorse

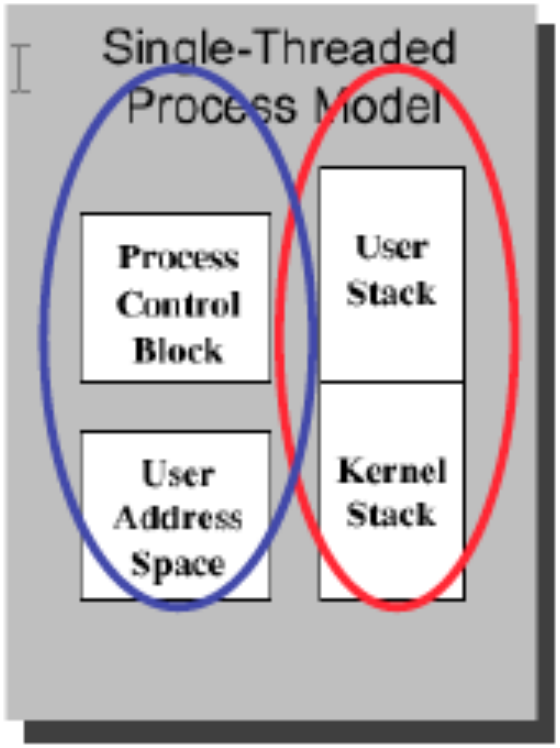
Processo e thread

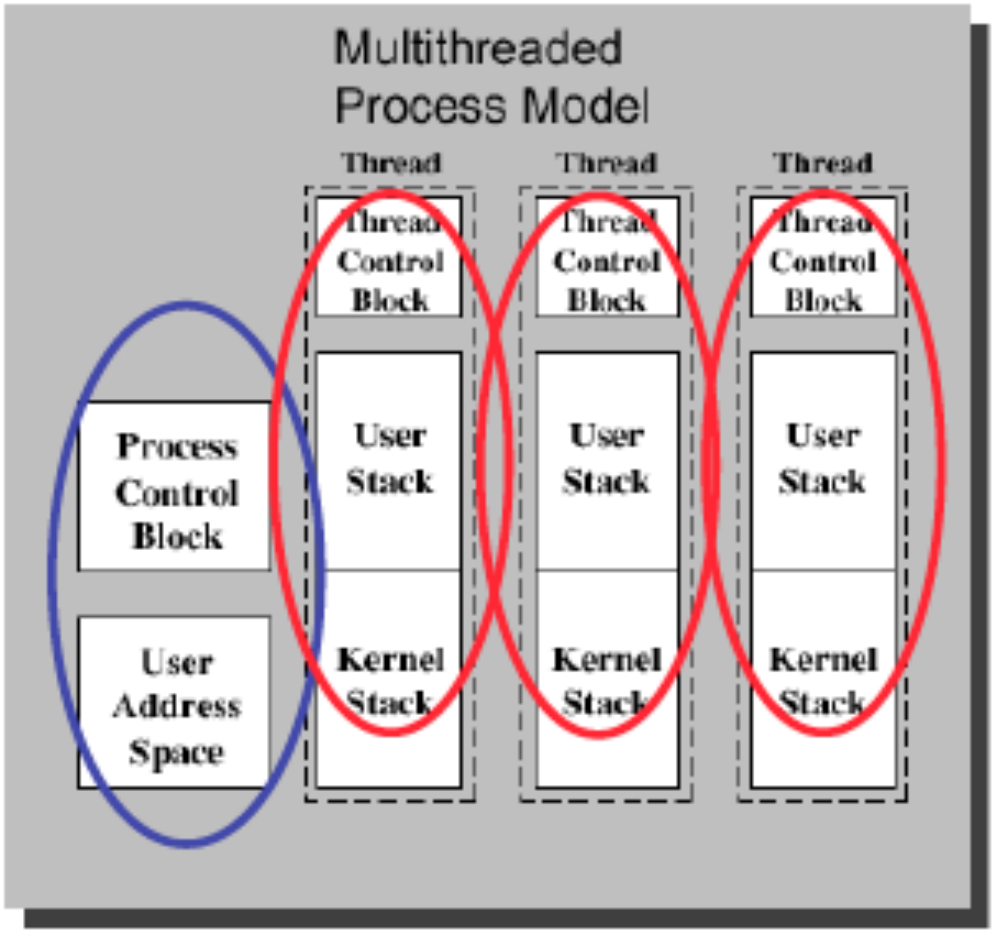
- Sono associati a un processo:
  - Spazio di indirizzamento
  - Risorse del sistema
- Sono associati a una singola thread:
  - *Stato di esecuzione*
  - Contatore di programma (program counter)
  - Insieme di registri (della CPU)
  - Stack
- Le thread condividono:
  - Spazio di indirizzamento
  - Risorse e *stato del processo*

Multi-threading

- In un S.O. classico: 1 processo = 1 thread
- Multithreading = possibilità di supportare più thread per un singolo processo
- Conseguenza
  - Separazione tra "flusso" di esecuzione (thread) e spazio di indirizzamento
    - Processo con thread singola
      - Un flusso associato ad uno spazio di indirizzamento
    - Processo con thread multiple
      - Più flussi associati ad un singolo spazio di indirizzamento

Multi-threading





Vantaggi dei thread

- Riduzione tempo di risposta
  - Mentre una thread è bloccata (I/O o elaborazione lunga), un'altra thread può continuare a interagire con l'utente
- Condivisione delle risorse
  - Le thread di uno stesso processo condividono la memoria senza dover introdurre tecniche esplicite di condivisione come avviene per i processi → sincronizzazione, comunicazione agevolata

Vantaggi dei thread

- Economia
  - Creazione/terminazione thread e contex switch tra thread più veloce che non tra processi
    - Solaris: creazione processo 30 volte più lento che creazione thread, contex switch tra processi 5 volte più lento che tra thread
- Scalabilità
  - Multithreading aumenta il parallelismo se l'esecuzione avviene su multiprocessore
    - un thread in esecuzione su ogni processore

Esempio Thread Singolo

- Considerate il seguente programma:

```
main() {
  ComputePI("pi.txt");
  PrintClassList("clist.text");
}
```

- Che cosa si osserva?
  - Il programma non stampera' mail la class list
  - Perché? Perché ComputePI non termina

Uso dei Thread

- Lo stesso programma con due threads:

```
main() {
  CreateThread(ComputePI("pi.txt"));
  CreateThread(PrintClassList("clist.text"));
}
```

- Cosa fa la "CreateThread" ?
  - Inizia thread indipendenti che eseguono la procedura indicata
- Comportemento osservato?
  - Ora la class list viene stampata
  - Il programma si comporta come se ci fossero 2 CPU in realta' 1 CPU utilizzata in concorrenza tra i due thread

Stati di un thread

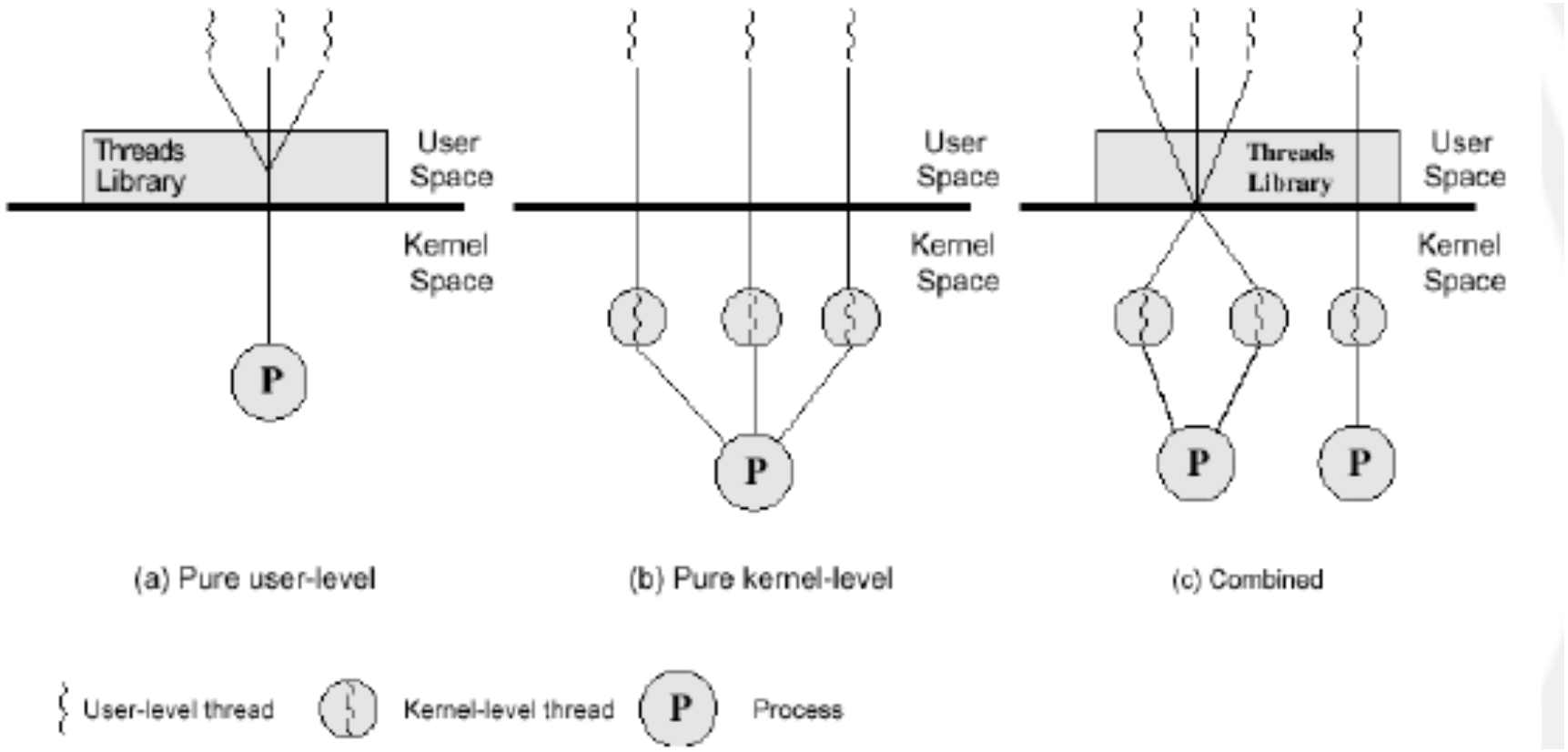
- Come un processo:
  - Pronto
  - In esecuzione
  - In attesa

- Stato del processo può, in generale, non coincidere con lo stato della thread
- Problema:
  - Un thread in attesa deve bloccare l'intero processo?
  - Dipende dall'implementazione…

Implementazione dei thread

- Esistono due possibilità:
  1. User-level thread
     - Gestione affidata alle applicazioni
     - Il kernel ignora l'esistenza delle thread
     - Funzionalità disponibili tramite una libreria di programmazione
  2. Kernel-level thread
     - Gestione affidata al kernel
     - Applicazioni usano le thread tramite system call
- Possibili approcci combinati (es.: Solaris)

Implementazione dei thread



User level thread

- Vantaggi
  - Non è necessario passare in modalità kernel per utilizzare thread
    - previene due mode switch → efficienza
  - Meccanismo di scheduling variabile da applicazione ad applicazione
  - Portabilità
    - girano su qualunque S.O. senza bisogno di modificare il kernel

User level thread

- Svantaggi
  - Il blocco di una thread blocca l'intero processo
    - Superabile con accorgimenti specifici
      - Es: I/O non bloccante
  - Non è possibile sfruttare multiprocessore
    - Scheduling di una thread sempre sullo stesso processore
      →un sola thread in esecuzione per ogni processo

User level thread

- Esempi
  - Green thread di Java (JDK1.1)
  - GNU portable thread
  - Libreria POSIX Pthreads (anche kernel-level)
  - Libreria C-threads del sistema Mach
  - UI-threads del sistema Solaris 2
  - ..

Kernel level thread

- Vantaggi
  - Scheduling a livello di thread
    - blocco di una thread NON blocca il processo
  - Più thread dello stesso processo in esecuzione contemporanea su CPU diverse
  - Le funzioni del S.O. stesso possono essere multithreaded
- Svantaggi
  - Scarsa efficienza
    - Passaggio tra thread implica un passaggio attraverso il kernel

Kernel level thread

- Esempi
  - Win32
  - Solaris
  - Tru64 UNIX
  - BeOS
  - Linux
  - Native thread di Java

ESEMPIO: LIBRERIA POSIX PTHREADS

UNIVERSITY
OF TRENTO

Pthreads

- Per usare i pthreads in un programma C, è necessario includere la libreria:

  <pthread.h>

- Per compilare un programma che usa i pthreads occorre linkare la libreria libpthread, usando l'opzione -lpthread:

  $> gcc prog.c -o prog -lpthread

Creazione di un thread

- Un thread ha vari attributi, che possono essere cambiati, ad esempio:
  - la sua priorità (che influenza la frequenza con cui verrà schedulato)
  - la dimensione del suo stack (che specifica la quantità massima di argomenti che gli si possono passare, la profondità delle chiamate ricorsive, e così via)
- Noi però vedremo nell'esempio sempre thread con gli attributi di default

Creazione di un thread

- Gli attributi di un thread sono contenuti in un oggetto di tipo *pthread_attr_t*
- la syscall:

  int pthread_attr_init(pthread_attr_t *attr);

- inizializza con i valori di default un "*contenitore di attributi*" *attr, che potrà poi essere passato alla system call che crea un nuovo thread.

Creazione di un thread

- Un nuovo thread viene creato con la syscall intera pthread_create che accetta quattro argomenti:
  - una varibile di tipo pthread_t, che conterrà l'identificativo del thread che viene creato
  - un oggetto attr, che conterrà gli attributi da dare al thread creato. Se si vuole creare un thread con attributi di default, si può anche semplicemente usare NULL
  - un puntatore alla routine che contiene il codice che dovrà essere eseguito dal thread
  - un puntatore all'eventuale argomento che si vuole passare alla routine stessa

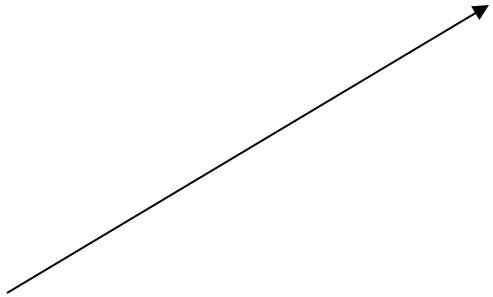
Terminazione di un thread

- Un thread termina quanto finisce il codice della routine specificata all'atto della creazione del thread stesso, oppure quando, nel codice della routine, si chiama la syscall di terminazione:
  - void pthread_exit(void *value_ptr);

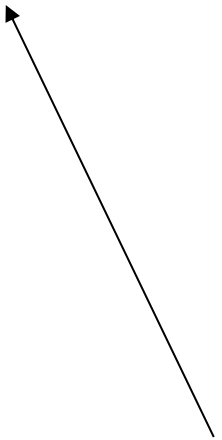
- Quando termina, il thread restituisce il "valore di return" specificato nella routine, oppure, se chiama la pthread_exit, il valore passato a questa syscall come argomento.

Sincronizzazione fra thread

- Un thread può sospendersi, in attesa della terminazione di un altro thread chiamando la syscall:
  - int pthread_join(pthread_t thread, void **value_ptr);



identificativo del thread di cui si attende la terminazione. Naturalmente deve essere un thread appartenente allo stesso processo.



valore restituito dal thread che termina

Exempio: t1.c

funzione che contiene il codice di un peer thread

```
#include <pthread.h>
#include <stdio.h>
void *tbody(void *arg)
{
int j;
printf(" ciao sono un thread, mi hanno appena creato\n");
*(int *)arg = 10;
sleep(2) /* faccio aspettare un pò il mio creatore, poi termino */
pthread_exit((int *)50); /* oppure return ((int *)50); */
}
```

```
main(int argc, char **argv)
{
int i;
pthread_t mythread;
void *result;
printf("sono il primo thread, ora ne creo un altro \n");
pthread_create(&mythread, NULL, tbody, (void *) &i);
printf("ora aspetto la terminazione del thread che ho creato \n");
pthread_join(mythread, &result);
printf("Il thread creato ha assegnato %d ad i\n",i);
printf("Il thread ha restituito %d \n",result);
}
```

Esempio: t1.c

- provate a stampare "mythread" come intero. Che valore ha? E se create altri thread (mythread1, mythread2,...) che valore viene assegnato alle rispettive variabili? Provate anche ad usare la funzione che restituisce l'id di un thread:
  - pthread_t thread_id
  - thread_id = pthread_self(); // id del thread chiamante

- il thread *main* ha generato un secondo thread *tbody*. Poi *main* si è messo in attesa della terminazione del thread creato (con pthread_join), ed è terminato lui stesso.
- Ma che differenza c'è rispetto ad usare fork e wait (eventualmente con una execl)?

Condivisione dello spazio logico

- I due thread condividono lo stesso spazio di indirizzamento, e quindi vedono le stesse variabili: se uno dei due modifica una variabile, la modifica è vista anche dall'altro thread.
- Nel codice di t1, il *main* passa al thread *tbody* il puntatore alla variabile *i* dichiarata nel *main*. il thread tbody modifica la variabile, e questa modifica è vista da main.
- Nel caso dei processi tradizionali, una cosa simile è ottenibile solo usando esplicitamente un segmento di memoria condivisa.

Condivisione dello spazio logico

- Ma i thread di un processo possono condividere variabili in maniera ancora più semplice, usando variabili globali.

```
#include <pthread.h>
#include <stdio.h>
int global_var = 5;
void *tbody(void *arg)
{
printf(" ciao sono un thread, ora modifico una var globale\n");
global_var = 27;
*(int *)arg = 10;
pthread_exit((int *)50); /* oppure return ((int *)50); */
}
```

```
main(int argc, char **argv)
{
pthread_t mythread;
void *result;
pthread_create(&mythread, NULL, tbody, (void *) &i);
printf("ora aspetto la terminazione del thread che ho creato \n");
pthread_join(mythread, &result);
printf("ora global_var vale: %d \n",global_var);
}
```

- tuttavia un thread può anche avere variabili proprie, viste solo dal thread stesso usando la classe di variabili thread_specific_data, che però noi non vedremo

Sincronizzazione fra thread

- Diversi strumenti sono disponibili per sincronizzare fra loro i thread di un processo, fra questi anche i semafori.
- In realtà i semafori non fanno parte dell'ultima versione dello standard POSIX, ma della precedente. Tuttavia sono normalmente disponibili in tutte le versioni correnti dei pthread.
- I pthread mettono anche a disposizione meccanismi di sincronizzazione strutturati, quali le *variabili condizionali*.

GESTIONE DEI PROCESSI DEL SISTEMA OPERATIVO

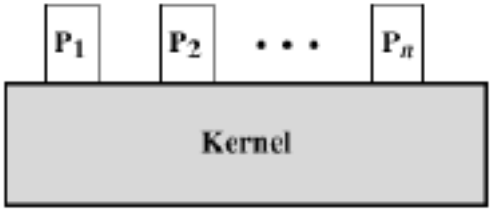
UNIVERSITY
OF TRENTO

Esecuzione del kernel

- Il S.O. è un programma a tutti gli effetti
- Il S.O. in esecuzione può essere considerato un processo?
  - Opzioni:
    - Kernel eseguito separatamente
    - Kernel eseguito all'interno di un processo utente
    - Kernel eseguito come processo

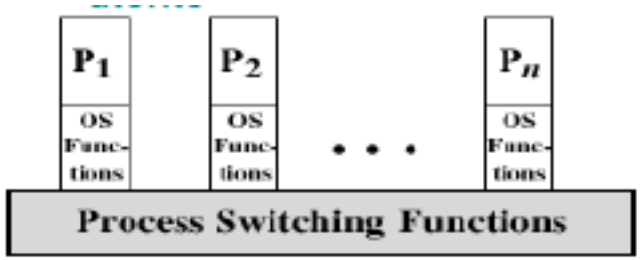
Kernel “separato”

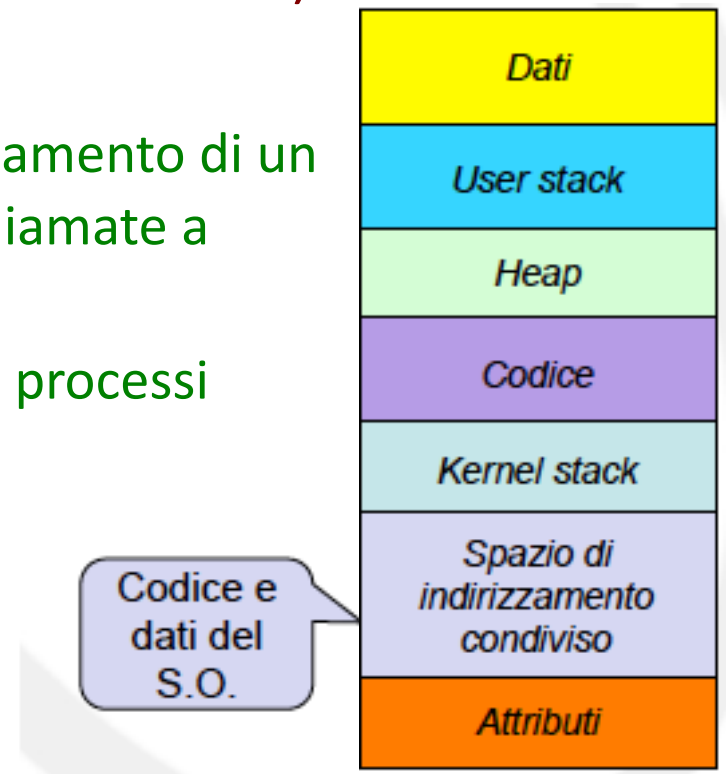
- Kernel esegue “al di fuori” di ogni processo
  - S.O. possiede uno spazio “riservato” in memoria
  - S.O. prende il controllo del sistema
  - S.O. sempre in esecuzione in modo privilegiato
- Concetto di processo applicato solo a processi utente
- Tipico dei primi S.O.



Kernel in processi “utente”

- Servizi del S.O. = procedure chiamabili da programmi utente
  - Accessibili in modalità protetta (kernel mode)
  - Immagine dei processi prevede
    - Kernel stack per gestire il funzionamento di un processo in modalità protetta (chiamate a funzione)
    - Codice/dati del S.O. condiviso tra processi





Kernel in processi “utente”

- Vantaggi:
  - In occasione di interrupt o trap durante l’esecuzione di un processo utente serve solo mode switch
    - Mode switch = il sistema passa da user mode a kernel mode e viene eseguita la parte di codice relativa al S.O. senza context switch
    - Più leggero rispetto al context switch
  - Dopo il completamento del suo lavoro, il S.O. può decidere di riattivare lo stesso processo utente (mode switch) o un altro (context switch)

Kernel come processo

- Servizi del S.O. = processi individuali
  - Eseguiti in modalità protetta
  - Una minima parte del S.O. deve comunque eseguire al di fuori di tutti i processi (scheduler)
  - Vantaggioso per sistemi multiprocessore dove processi del S.O. possono essere eseguiti su processore ad hoc