UNIVERSITÀ
di VERONA

Dipartimento
di INFORMATICA

Processi e thread

Anno Accademico 2020-2021

Graziano Pravadelli

Sommario

- Concetto di processo
- Stati di un processo
- Operazioni e relazioni tra processi
- Concetto di thread
- 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)

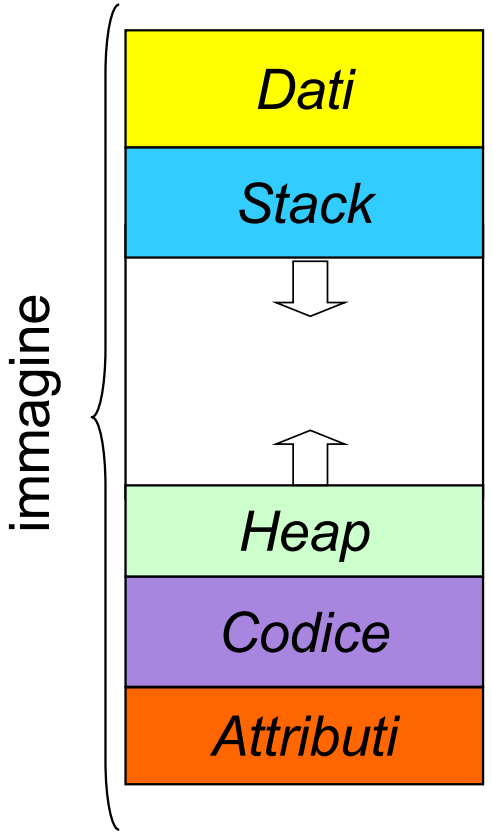
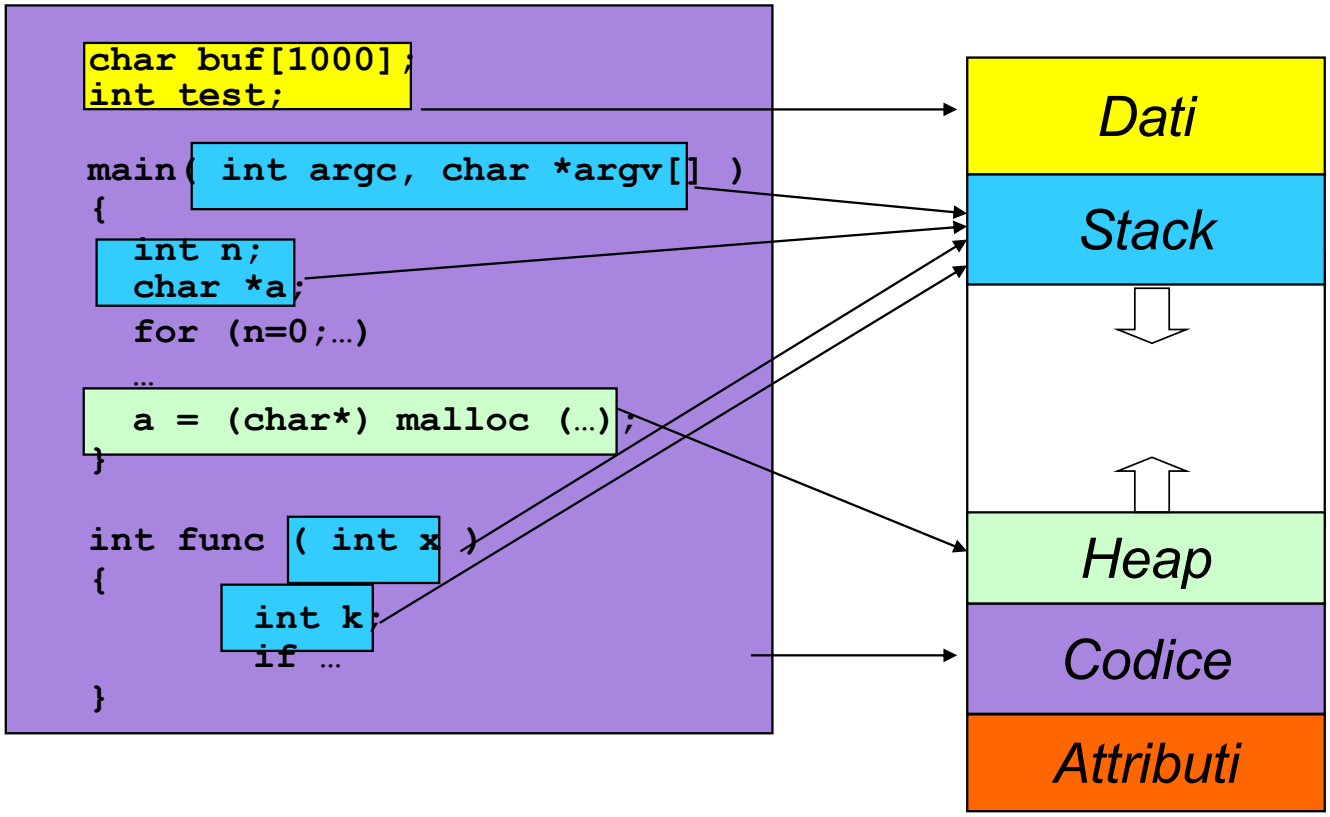
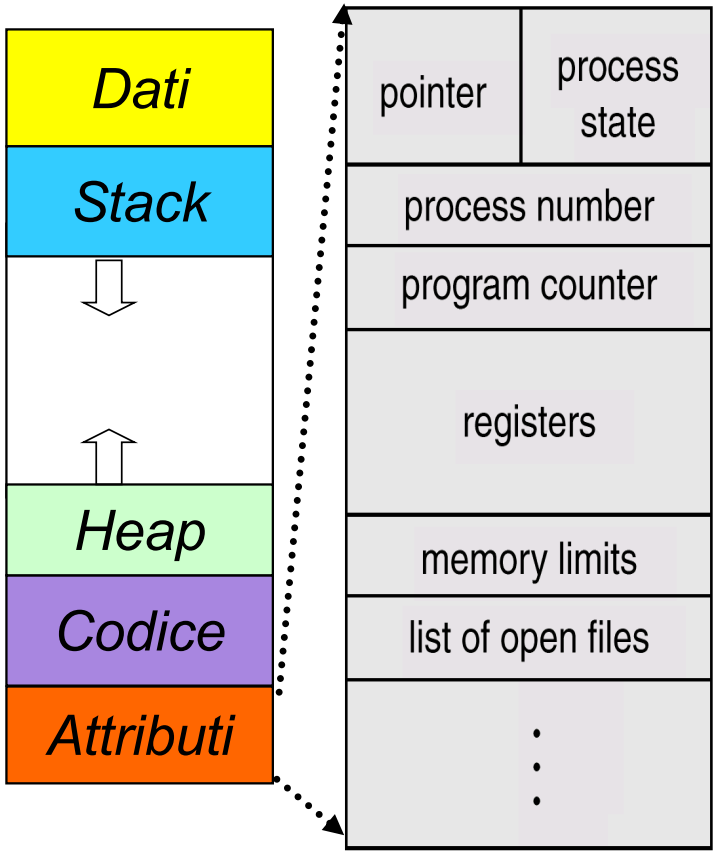


Immagine in memoria



Attributi (Process Control Block)

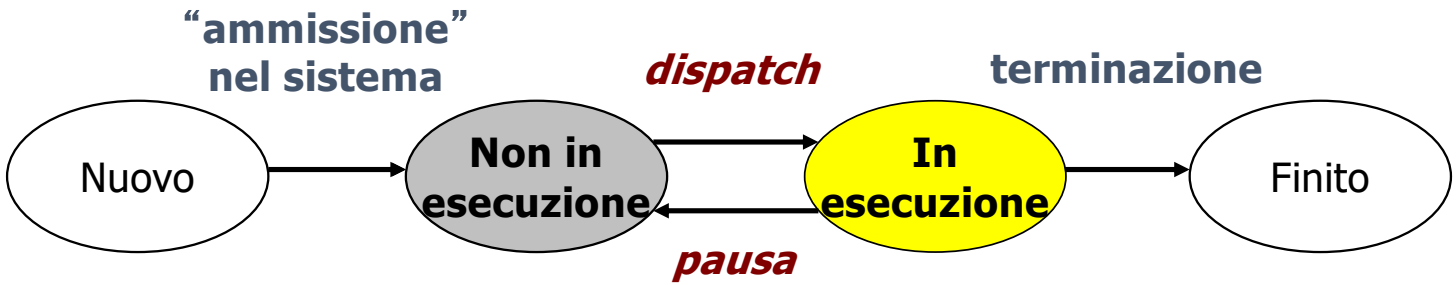
- All'interno del S.O. ogni processo è rappresentato dal process control block (PCB)
  - stato del processo
  - program counter
  - valori dei registri
  - informazioni sulla memoria (es:. registri limite, tabella pagine)
  - informazioni sullo stato dell'I/O (es:. richieste pendenti, file)
  - informazioni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



Stati di un processo

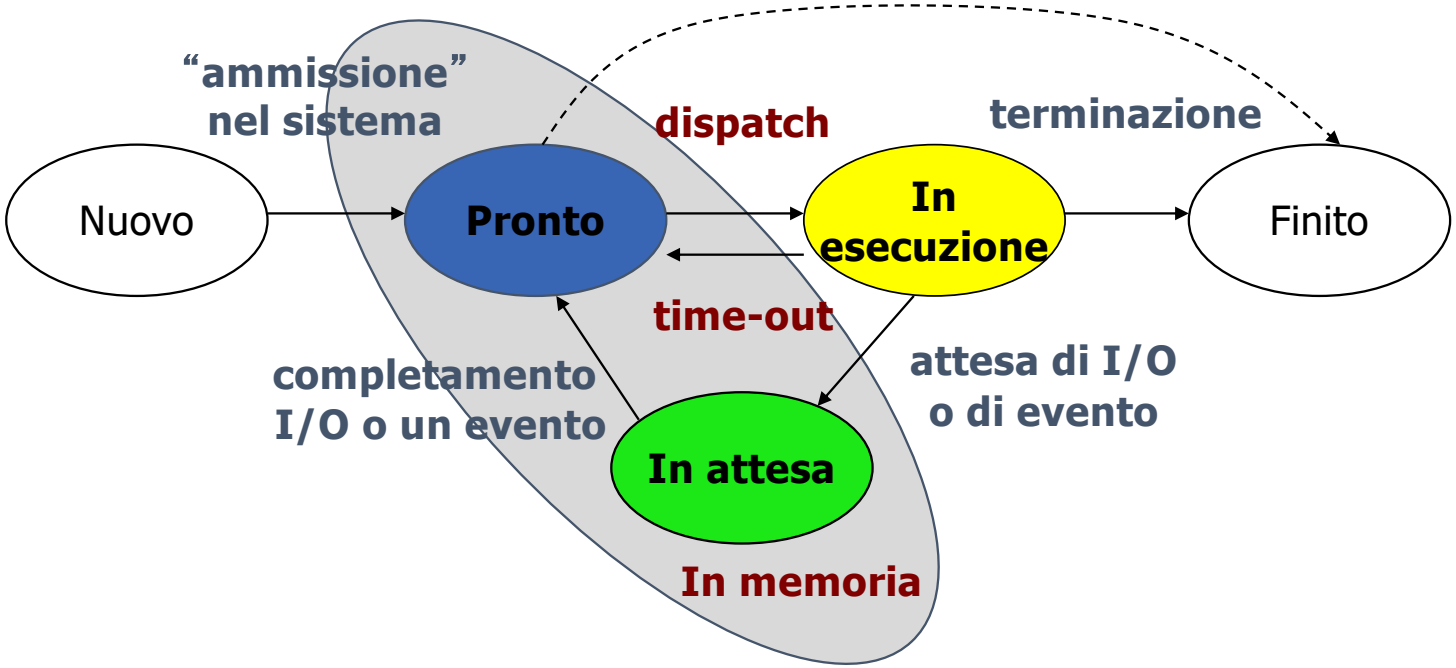
Stati di un processo

- 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

Code di scheduling

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

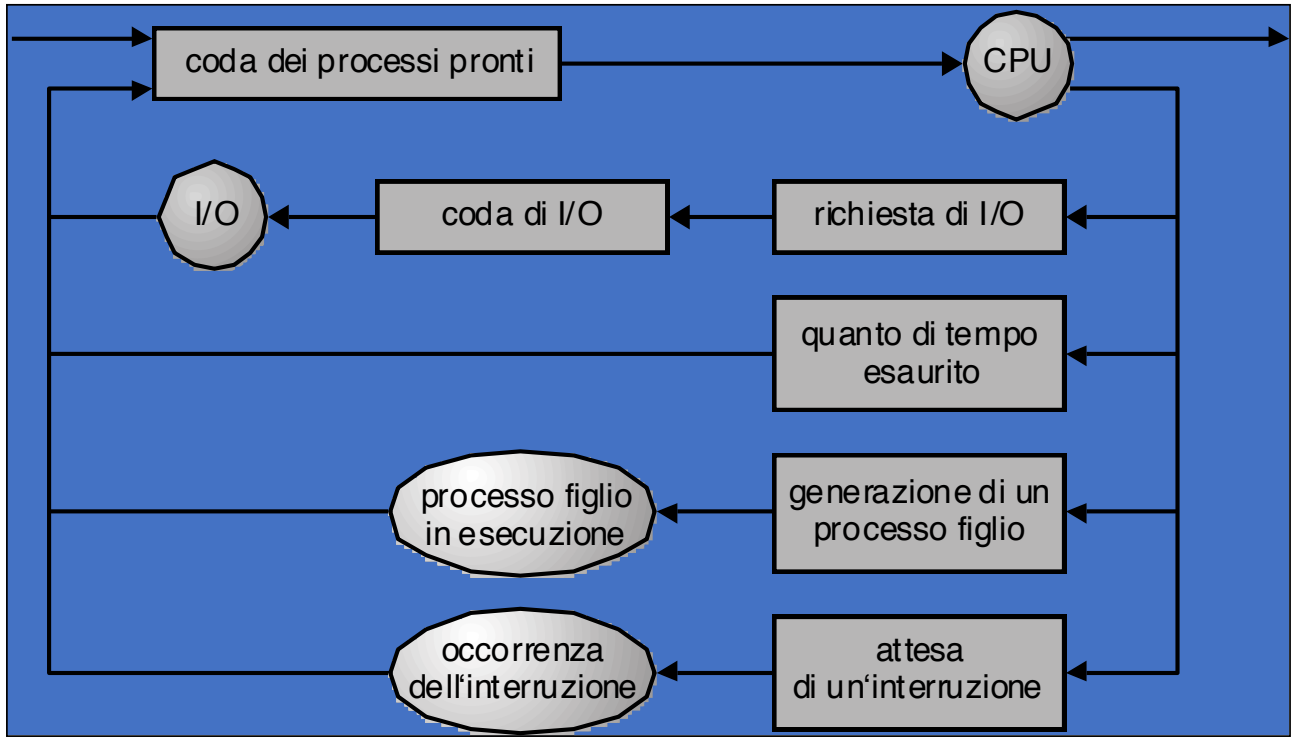
Code di scheduling

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

Cambio di contesto

salvataggio PCB del processo che esce e caricamento del PCB del processo che entra

Passaggio alla modalità utente

(all'inizio della fase di dispatch il sistema si trova in modalità kernel)

Salto all'istruzione da eseguire del processo appena arrivato nella CPU

Cambio di contesto (context switch)

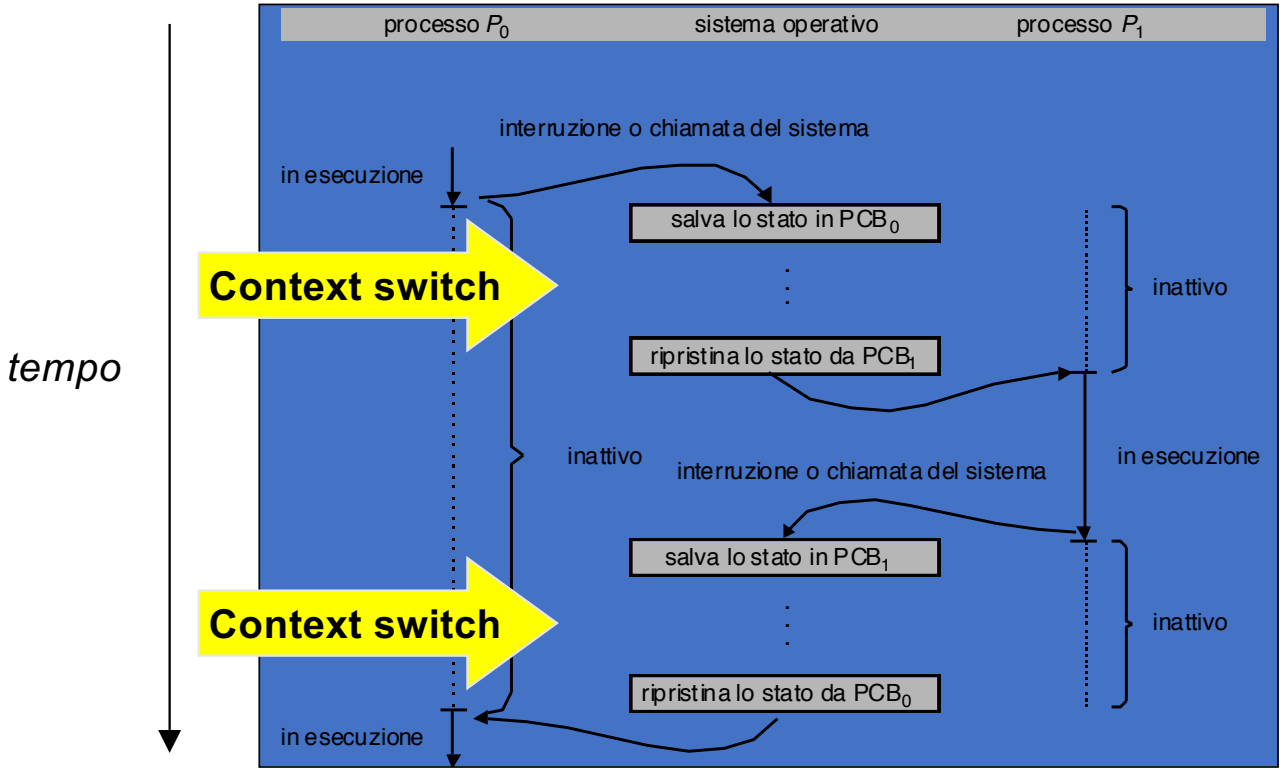
Passaggio della CPU a un nuovo processo

- Registrazione dello stato del processo vecchio
- 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

Creazione di un processo

Un processo può creare un figlio

- Figlio ottiene risorse dal S.O. o dal padre per
- Spartizione
- Condivisione

Tipi di esecuzione

- Sincrona
  - Padre attende la terminazione dei figli
- Asincrona
  - Evoluzione “parallela” 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, …)

Relazioni tra processi

Processi indipendenti

- Esecuzione deterministica (dipende solo dal proprio input) e riproducibile
- Non influenza, né viene influenzato da altri processi
- Nessuna condivisione dei dati con altri processi

Processi cooperanti

- Influenza e può essere influenzato da altri processi
- Esecuzione non deterministica e non riproducibile

Perché processi cooperanti?

Condivisione informazioni

Accelerazione del calcolo

- Esecuzione parallela di "*subtask*" su multiprocessore

Modularità

- Funzioni distinte su vari processi

Convenienza

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

Multithreading

In un S.O. classico:

- 1 processo = 1 thread

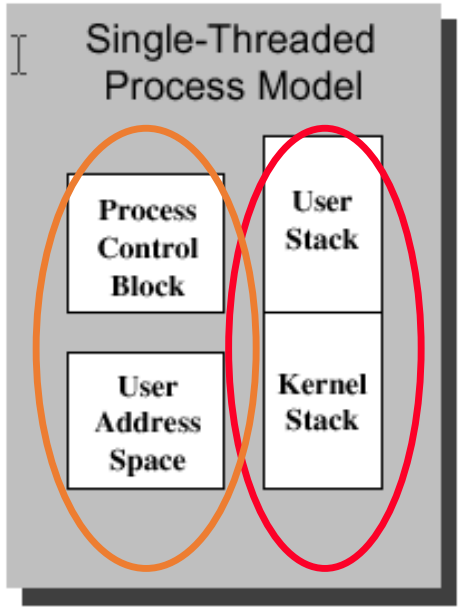
Multithreading

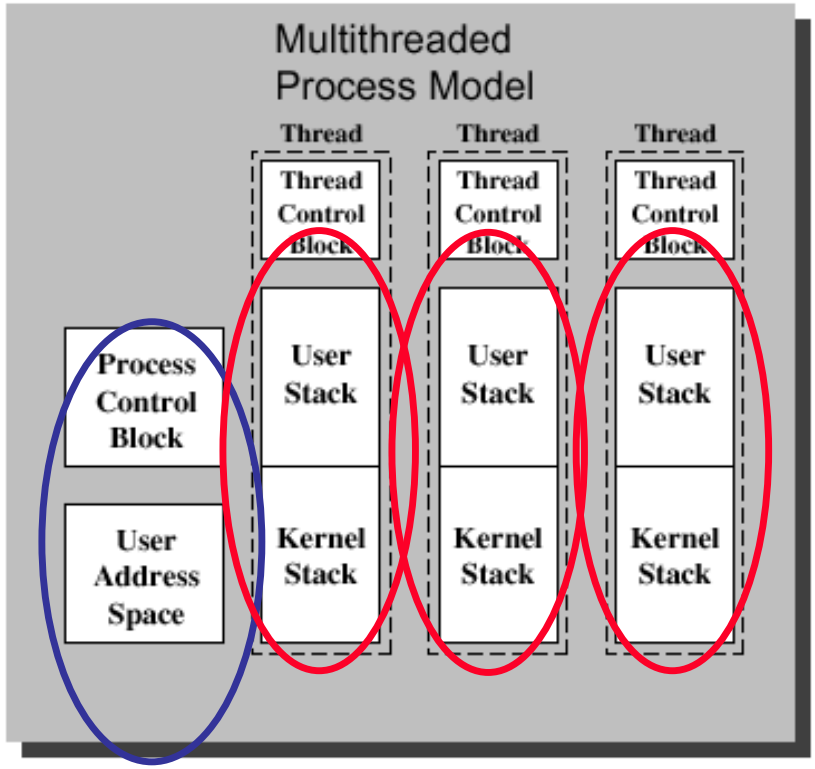
- possibilità di supportare più thread per un singolo processo

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

Multithreading





Vantaggi delle 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 memora senza dover introdurre tecniche esplicite di condivisione come avviene per i processi → sincronizzazione, comunicazione agevolata

Vantaggi delle 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
  - una thread in esecuzione su ogni processore

Stati di una thread

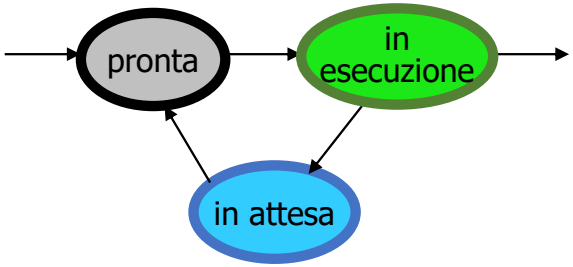
Come un processo:

- Pronta
- In esecuzione
- In attesa

Stato del processo può, in generale, non coincidere con lo stato della thread

Problema:

- Una thread in attesa deve bloccare l'intero processo?
- Dipende dall'implementazione…



Implementazione delle thread

User-level thread

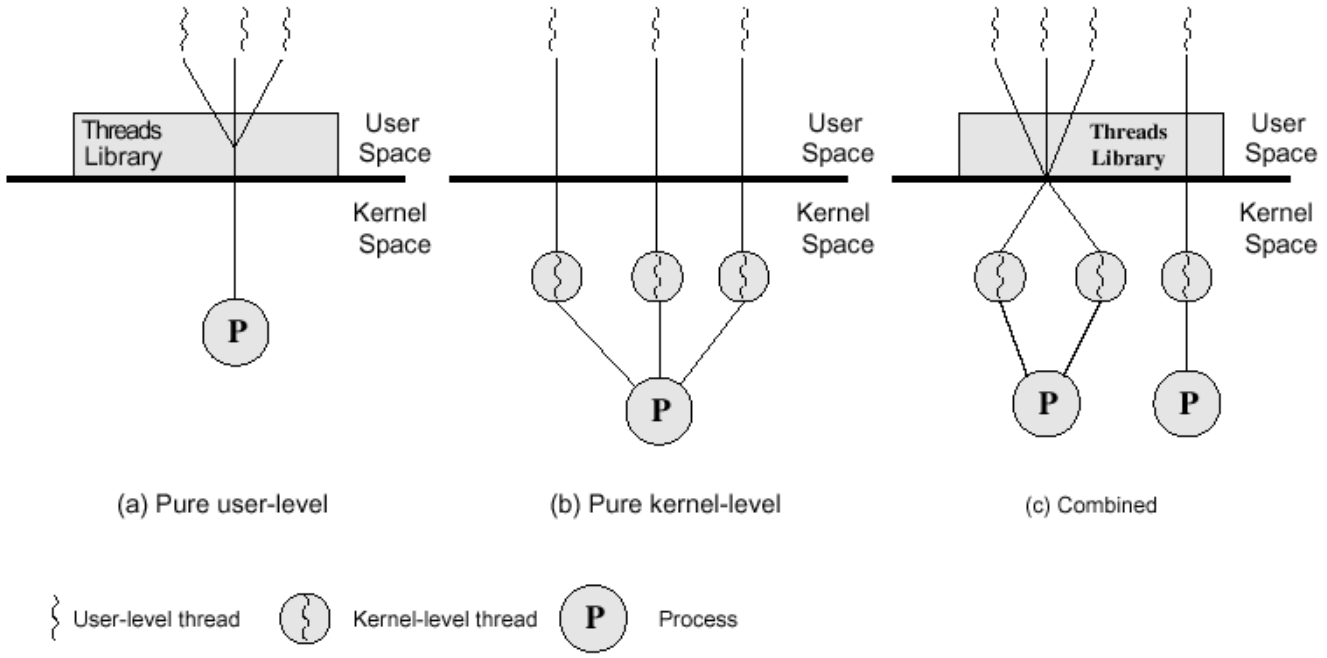
- Gestione affidata alle applicazioni
- Il kernel ignora l'esistenza delle thread
- Funzionalità disponibili tramite una libreria di programmazione

Kernel-level thread

- Gestione affidata al kernel
- Applicazioni usano le thread tramite system call

Approcci combinati

Implementazione delle thread



User-level thread

Vantaggi

- Non necessario passare modalità kernel per usare thread
  - previene due mode switch → efficienza
- Scheduling variabile da applicazione ad applicazione
- Portabilità
  - Girano su qualunque S.O. senza modificare kernel

Svantaggi

- 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 → una 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

Gestione dei processi del sistema operativo*

* Non presente nello Silberschatz, vedere
W. Stallings, “Operating Systems” Prentice Hall

Esecuzione del kernel

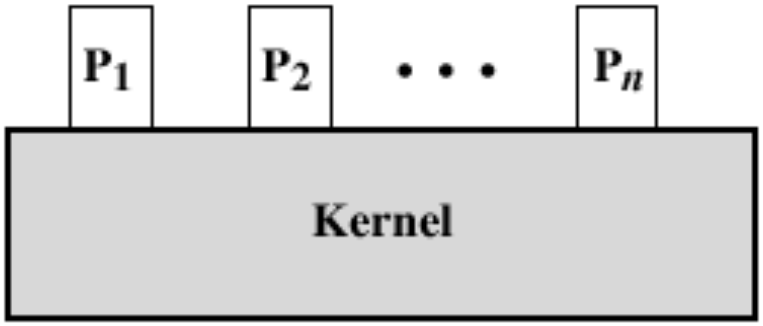
Il S.O. è un programma a tutti gli effetti

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

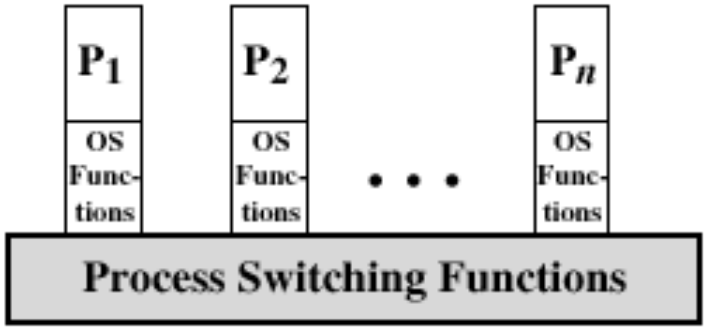
Kernel “separato”

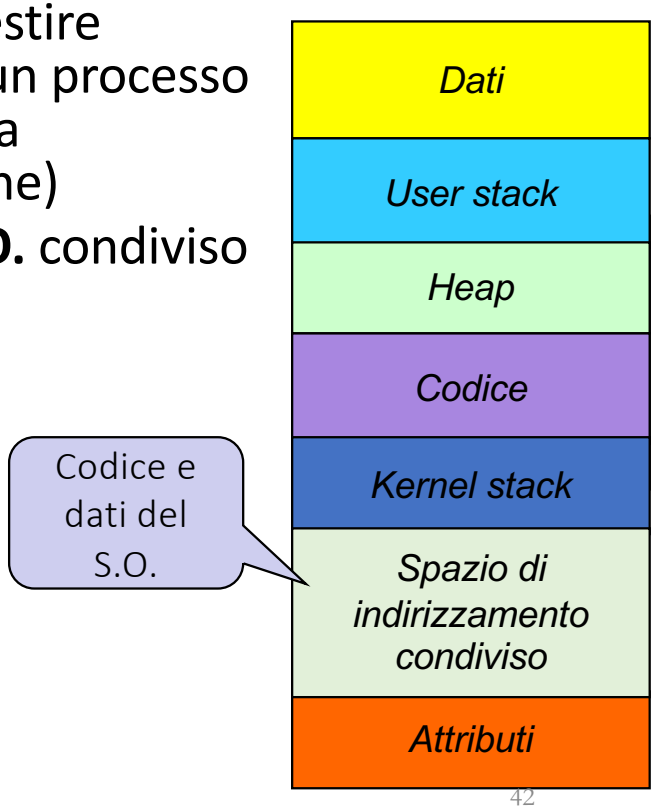
- Kernel esegue “al di fuori” di ogni processo
  - S.O. possiede uno spazio “riservato” in memoria
  - S.O. prende il controllo del sistema
  - S.O. sempre in esecuzione in modo privilegiato
- Concetto di processo applicato solo a processi utente
- Tipico dei primi S.O.



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 funzionamento di un processo in modalità protetta (chiamate a funzione)
    - **Codice/dati del S.O.** condiviso tra processi utente





Kernel in processi utente - Vantaggi

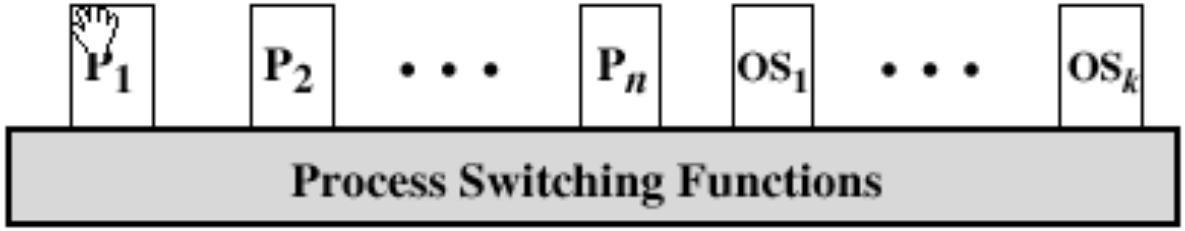
In occasione di interrupt o trap durante esecuzione di un processo utente serve solo mode switch

- 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



Conclusioni

Problematiche

Allocazione di risorse ai processi/thread

- CPU
- Memoria
- Spazio su disco

Coordinamento tra processi/thread (concorrenti)

- Sincronizzazione
- Comunicazione