Sistemi Operativi Modulo I

Secondo canale (M-Z)

A.A. 2021/2022

Corso di Laurea in Informatica

5. Thread - IPC

Paolo Ottolino

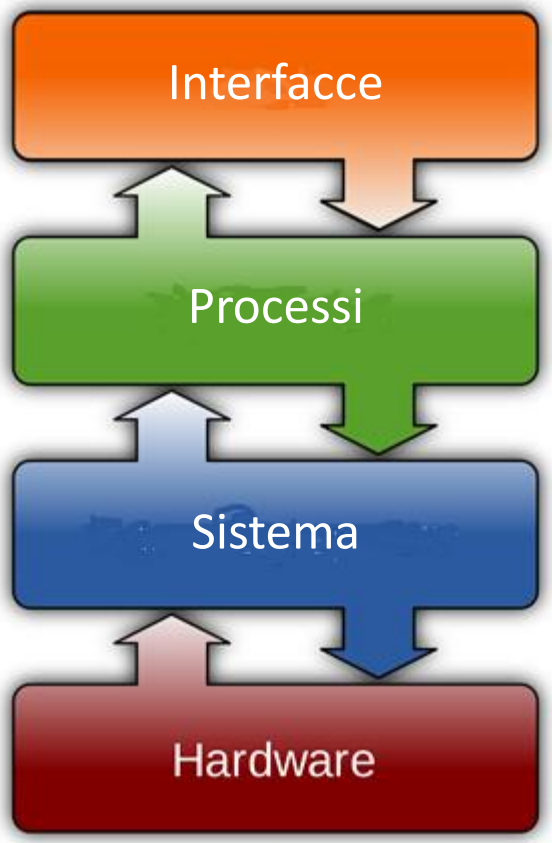
Sapienza Università di Roma

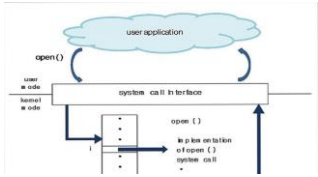
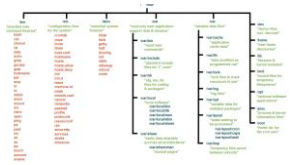
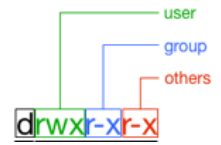
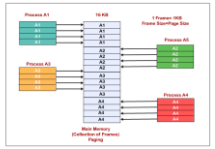
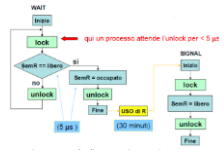
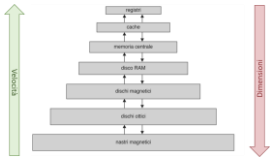
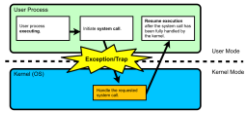
Dipartimento di Informatica

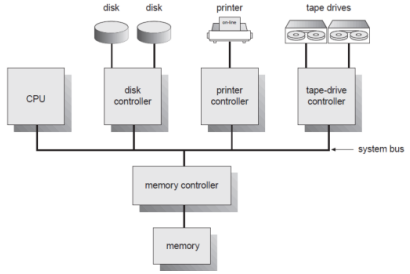
Operating Systems: Course Program by Layers

A. Intro: Operating Systems «Who-What-Why-When-Where-How»
B. A Short Look Back: «Progettazione dei Sistemi Digitali», «Architettura degli Elaboratori»
C. EndPoint: Programmable Computer (CPU, Memory, I/O)
D. Operating Systems - basics: cenni storici, multiprogramming, time sharing, spooling; classificazione.
E. Tipologie di Sistemi di Elaborazione
F. Processes: PCB, Context Switch, Signals, Scheduler
G. Memory Management: Protection, Partitioning (Segmentation, Pagination, page Table), Layers (Cache, TLB, Swapping, LRU)
H. I/O: Buffering, Caching, Scheduling
I. Security & Protection: Threats, Attacks, Risks. CIA: Confidentiality-Integrity-Availability -> Authentication, Authorization, Accounting
J. File System
K. Process Interactions: Race & Communications
L. Starvation, Livelock, Deadlock

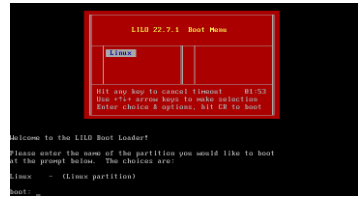
Operating Systems: 3x3 Matrix



	Obiettivi	Funzioni	Servizi
	Astrazione 	File System Sh/GUI/OLTP 	Authentication/ Authorization/ Accounting 
	Virtualizzazione 	Process Mgmt 	IPC System Calls 
	Input/Output 	Memory Mgmt Error Detection, Dual-Mode 	Boot Interrupt Handling 



B	C
Formulas	Formula Results
=1/0	#DIV/0!
=A2/A3	#DIV/0!
=QUOTIENT(A2,A3)	#DIV/0!



Operating Systems: Thread

Obiettivo

Ogni processo tradizionale (mono-thread) esegue un'attività alla volta ➔ molte attività necessitano di molti processi: `fork()`

Ogni `fork()` implica risorse aggiuntive:

- Risorse di memoria duplicate: Area istruzioni e variabili globali, etc
- Scheduling addizionale: creazione nuovo processo e strutture (PCB)
- Context Switch addizionale: aumento del consumo di CPU senza elaborazioni fattuali

Πηνελόπεια

➔ **condivisione delle risorse** (spazio: memoria e tempo: elaborazione) fra attività analoghe

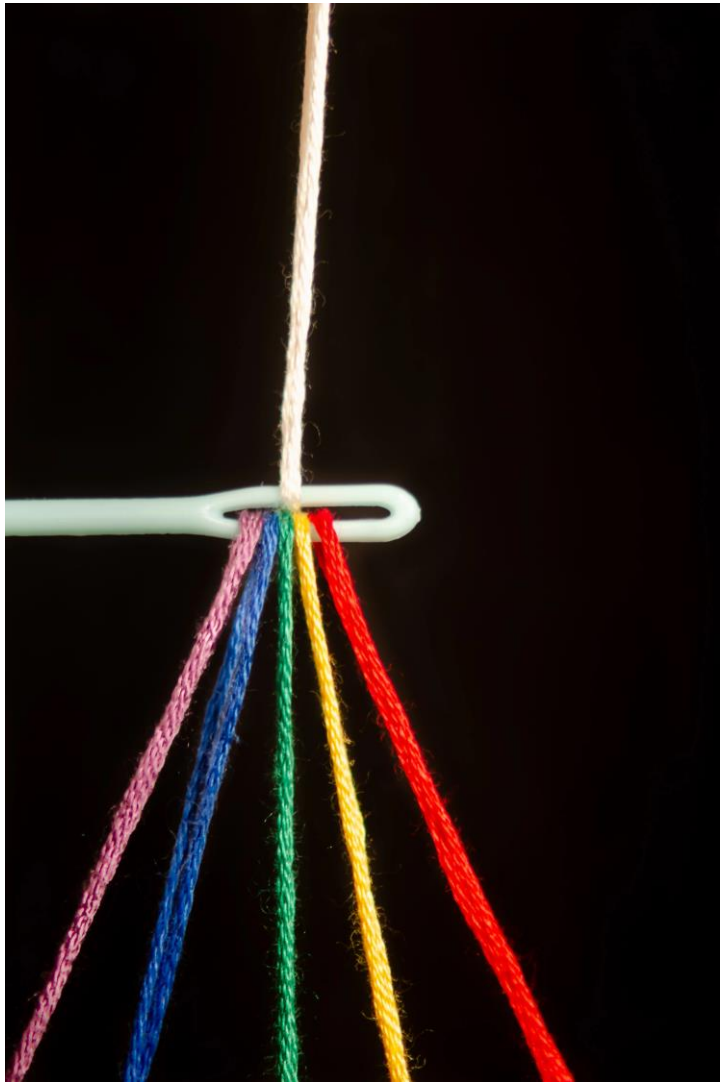
Operating Systems: Thread

Definizione

Thread: flusso di esecuzione (filo logico) indipendente all'interno dello stesso processo.
Lightweight Process (contesto alleggerito, rispetto ad un processo):

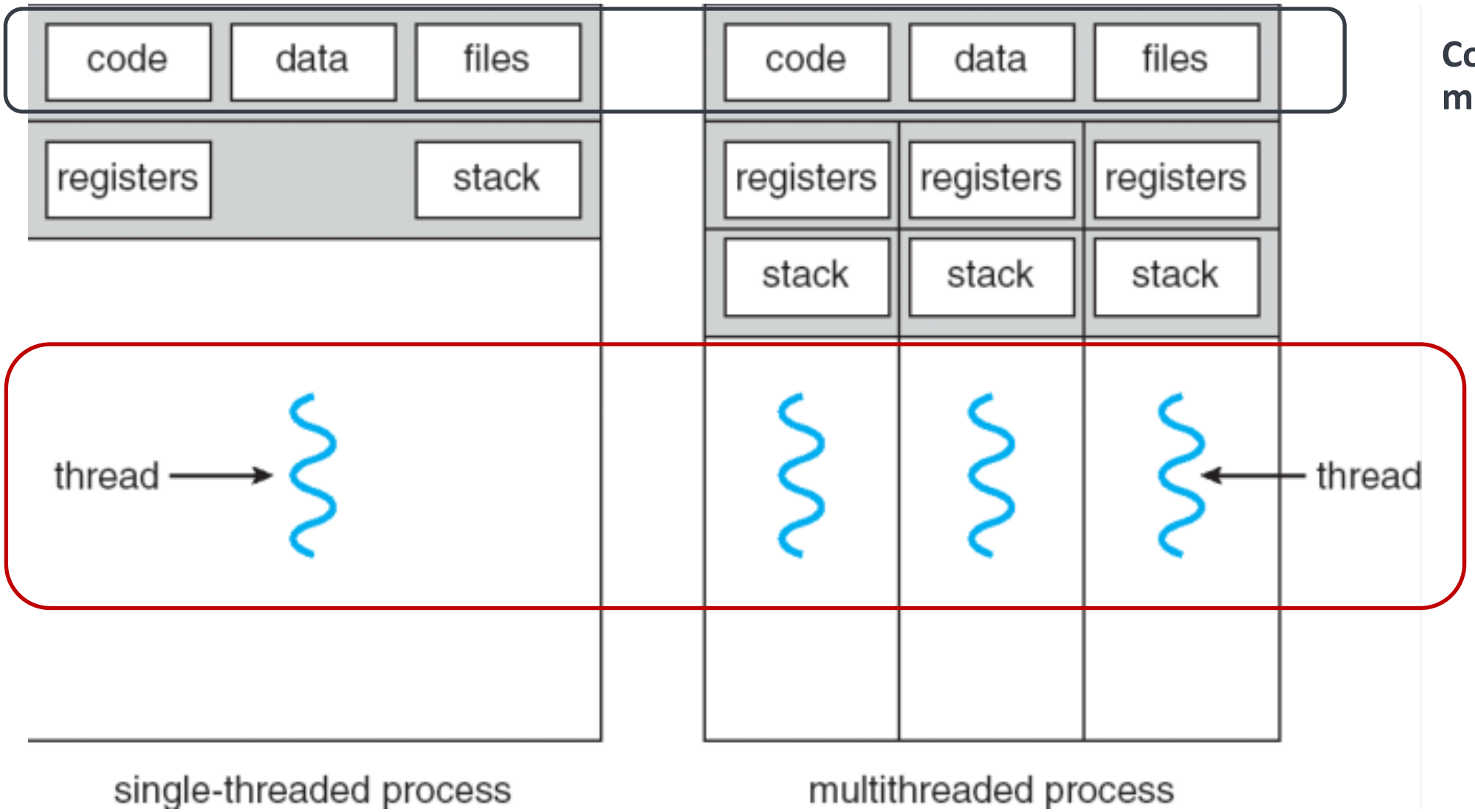
- **Spazio di Indirizzamento**: condiviso con gli altri thread del processo
- **Scheduling**: tramite la struttura di Rappresentazione Thread Control Block (**TCB**) che punta al **PCB** del processo contenitore

Condivisione/Riuso delle risorse.



Operating Systems: Thread

Definizione

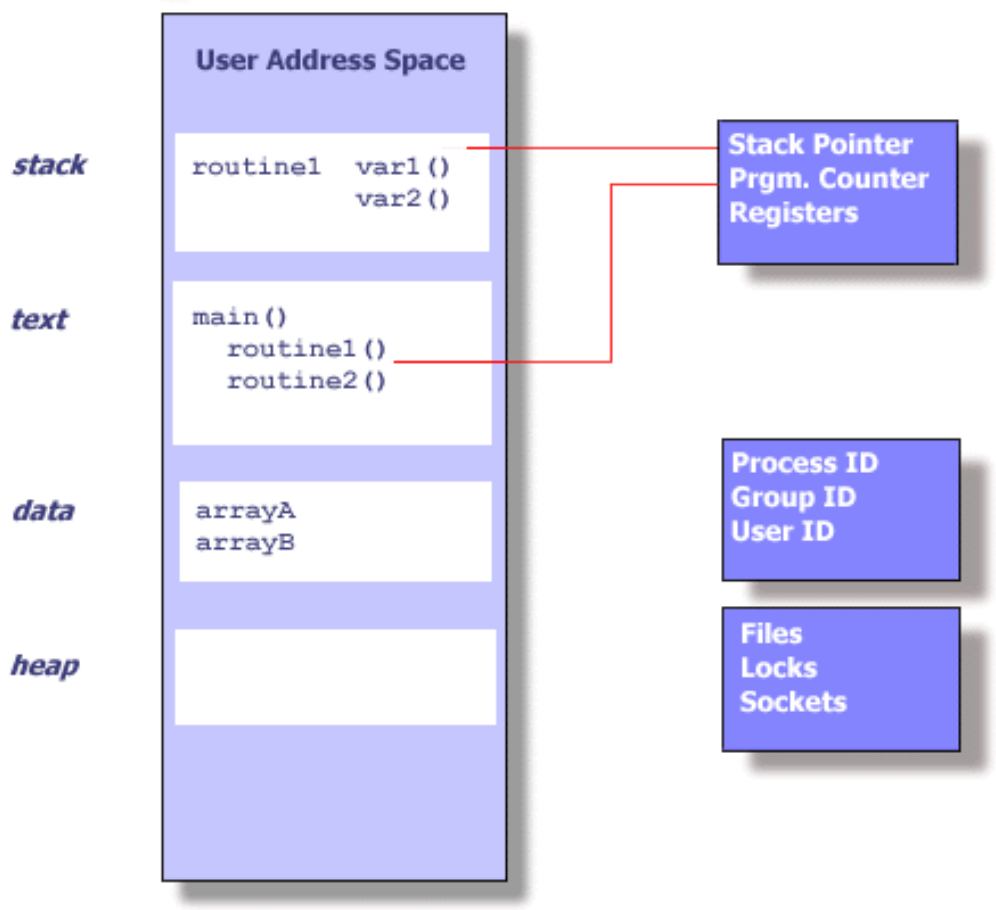


Condivisione delle memoria

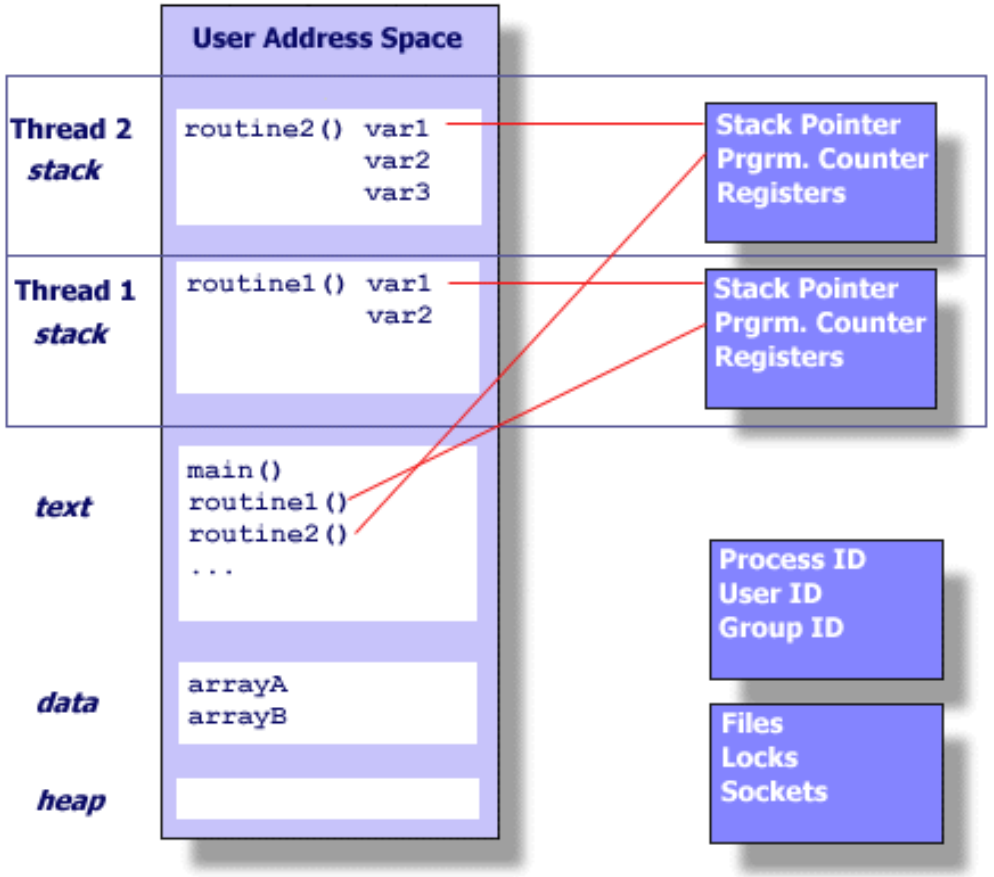
Esecuzione

Operating Systems: Thread

Memoria



Processo single-thread (sequenziale)

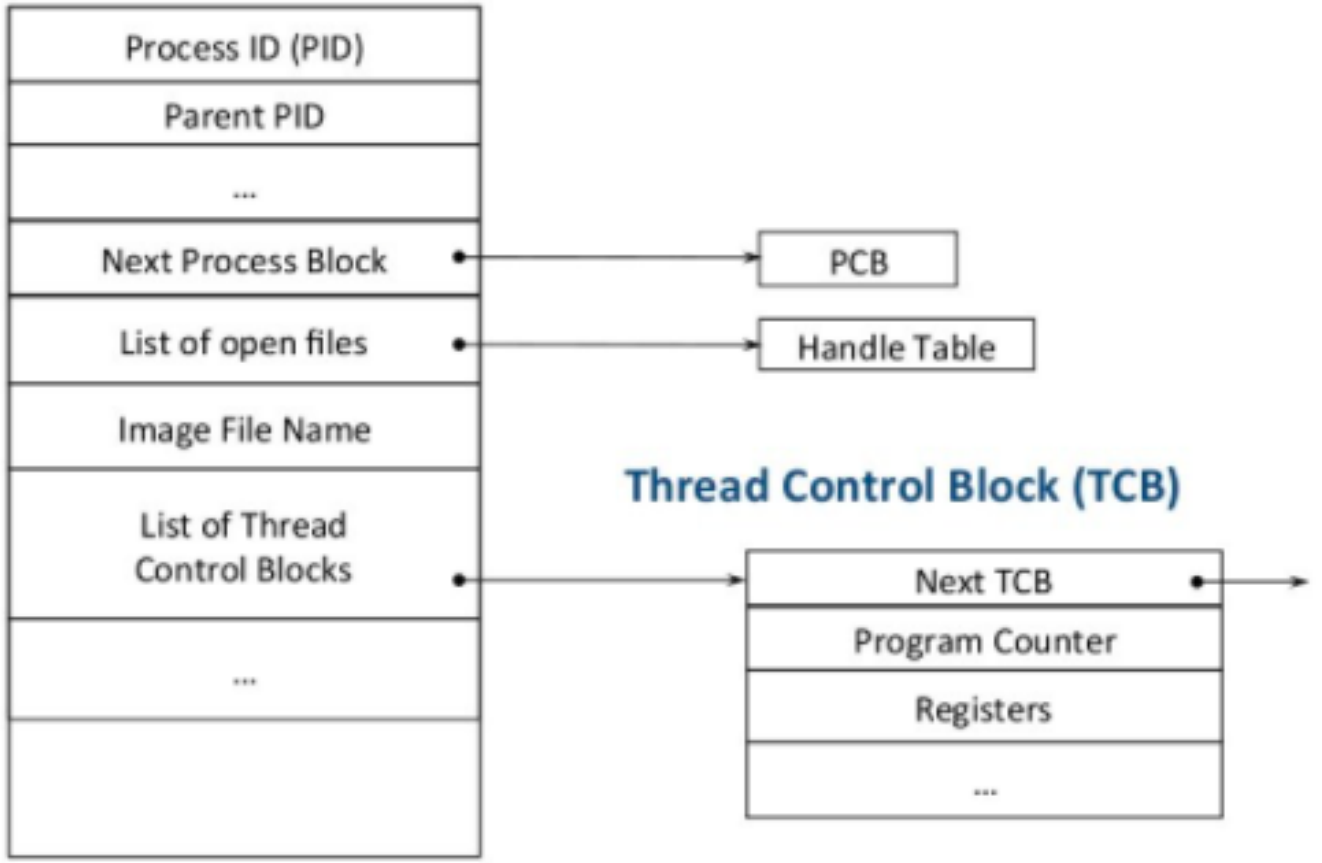


Processo multi-thread (parallelo)

Operating Systems: Thread

Control Block: PCB e TCB

Process Control Block (PCB)



- spazio di memoria
- risorse private (con le corrispondenti tabelle dei descrittori)

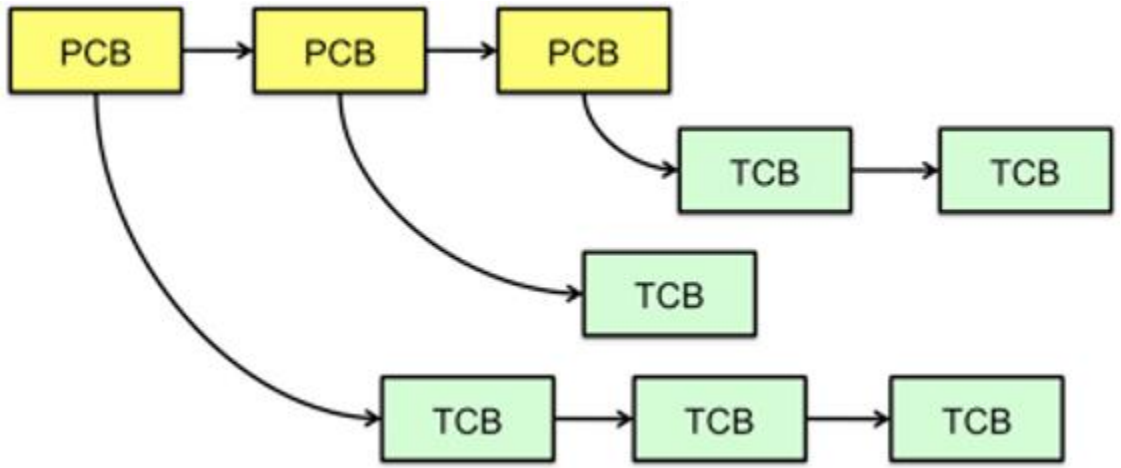
- stato della computazione (registri, stack, PC…)
- attributi (schedulazione, priorità)
- descrittore di thread (tid, priorità, segnali pendenti, …)
- memoria privata: Thread Specific Data (TSD)

Operating Systems: Thread

TCB: esempio Linux

Thread Control Block (TCB): Struttura dati del **kernel** che mantiene le informazioni sul thread

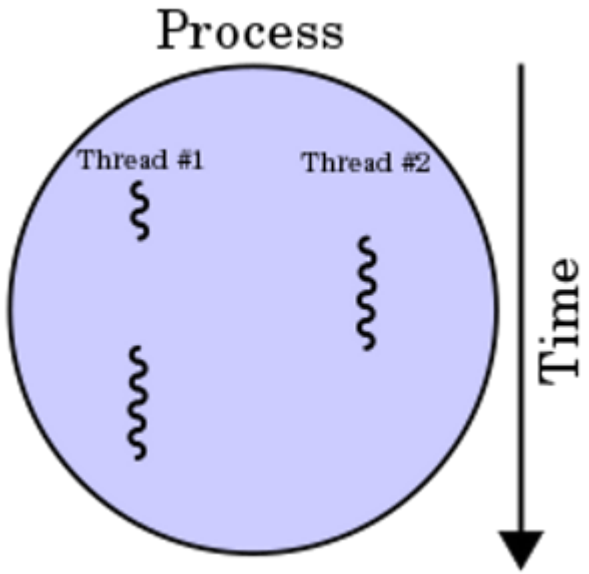
```
struct tcb {
        short           tcb_state;      /* TCP state                    */
        short           tcb_ostate;     /* output state                 */
        short           tcb_type;       /* TCP type (SERVER, CLIENT)    */
        int             tcb_mutex;      /* tcb mutual exclusion         */
        short           tcb_code;       /* TCP code for next packet     */
        short           tcb_flags;      /* various TCB state flags      */
        short           tcb_error;      /* return error for user side   */

        IPaddr          tcb_rip;        /* remote IP address            */
        short           tcb_rport;      /* remote TCP port              */
        IPaddr          tcb_lip;        /* local IP address             */
        short           tcb_lport;      /* local TCP port               */
        struct          netif           *tcb_pni; /* pointer to our interface */

        tcpseq          tcb_suna;       /* send unacked                 */
        tcpseq          tcb_snext;      /* send next                    */
        tcpseq          tcb_slast;      /* sequence of FIN, if TCBF_SNDFIN */
        long            tcb_swindow;    /* send window size (octets)    */
        tcpseq          tcb_lwseq;      /* sequence of last window update */
        tcpseq          tcb_lwack;      /* ack seq of last window update */
        int             tcb_cwnd;       /* congestion window size (octets) */
        int             tcb_ssthresh;   /* slow start threshold (octets) */
        int             tcb_smss;       /* send max segment size (octets) */
        tcpseq          tcb_iss;        /* initial send sequence        */

        int             tcb_srt;        /* smoothed Round Trip Time     */
        int             tcb_rtde;       /* Round Trip deviation estimator */
        int             tcb_persist;    /* persist timeout value        */
        int             tcb_keep;       /* keepalive timeout value      */
        int             tcb_rexmt;      /* retransmit timeout value     */
        int             tcb_rexmtcount; /* number of rexmts sent        */

        tcpseq          tcb_rnext;      /* receive next                 */
        tcpseq          tcb_rupseq;     /* receive urgent pointer       */
        tcpseq          tcb_supseq;     /* send urgent pointer          */

        int             tcb_lqsize;     /* listen queue size (SERVERs)  */
        int             tcb_listenq;    /* listen queue port (SERVERs)  */
        struct tcb *tcb_pptcb;  /* pointer to parent TCB (for ACCEPT)   */
        int             tcb_ocsem;      /* open/close semaphore         */
        int             tcb_dvnum;      /* TCP slave pseudo device number */

        int             tcb_ssema;      /* send semaphore               */
        u_char          *tcb_sndbuf;    /* send buffer                  */
        int             tcb_sbstart;    /* start of valid data          */
        int             tcb_sbcount;    /* data character count         */
        int             tcb_sbsize;     /* send buffer size (bytes)     */

        int             tcb_rsema;      /* receive semaphore            */
        u_char          *tcb_rcvbuf;    /* receive buffer (circular)    */
        int             tcb_rbstart;    /* start of valid data          */
        int             tcb_rbcount;    /* data character count         */
        int             tcb_rbsize;     /* receive buffer size (bytes)  */
        int             tcb_rmss;       /* receive max segment size     */
        tcpseq          tcb_cwin;       /* seq of currently advertised window */
        int             tcb_rseqq;      /* segment fragment queue       */
        tcpseq          tcb_finseq;     /* FIN sequence number, or 0    */
        tcpseq          tcb_pushseq;    /* PUSH sequence number, or 0   */
};
```



Operating Systems: Thread

Threat concorrenti e paralleli

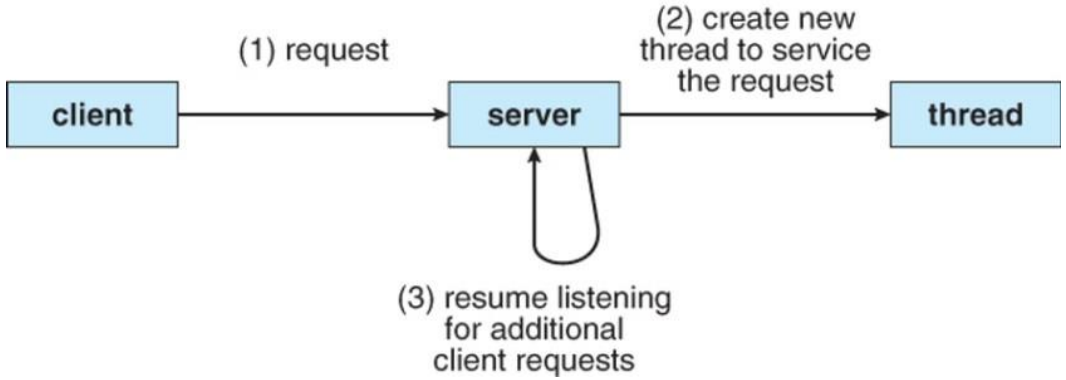
Concurrent Thread



Es. MS Word ha più thread che eseguono diversi compiti in parallelo, tramite thread:

- prendere l'input da tastiera
- controllo ortografico
- conteggio dell'ortografia
- …

Parallel Thread



Es. il web server è un sistema multihreaded dove ogni richiesta è gestita da un hread separato.

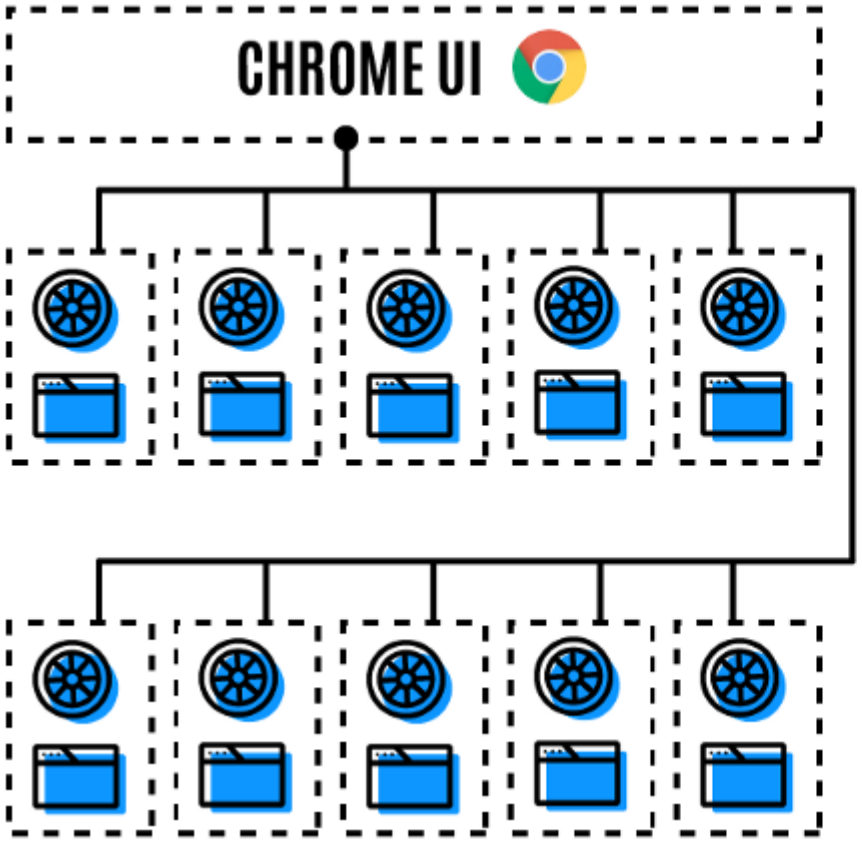
Se il web server fosse single-thrad potrebbe servire solo un client alla volta. Il tempo di attesa per la risposta potrebbe crescere a dismisura.

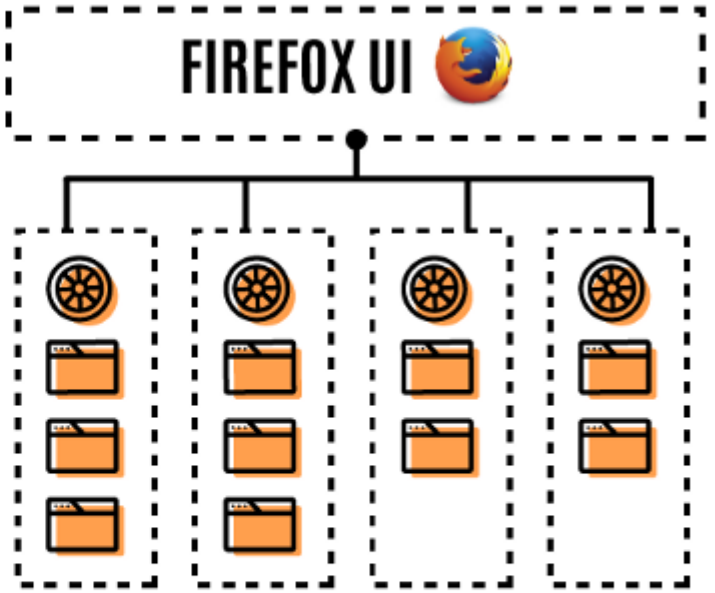
Operating Systems: Thread

Thread in processi utente: esempi

Ad ogni tab corriponde un processo separato. All'interno di questo sono attivati **thread concorrenti** (trasporto dati, rendering, etc).

Alte performance ma consumo di memoria e batterie





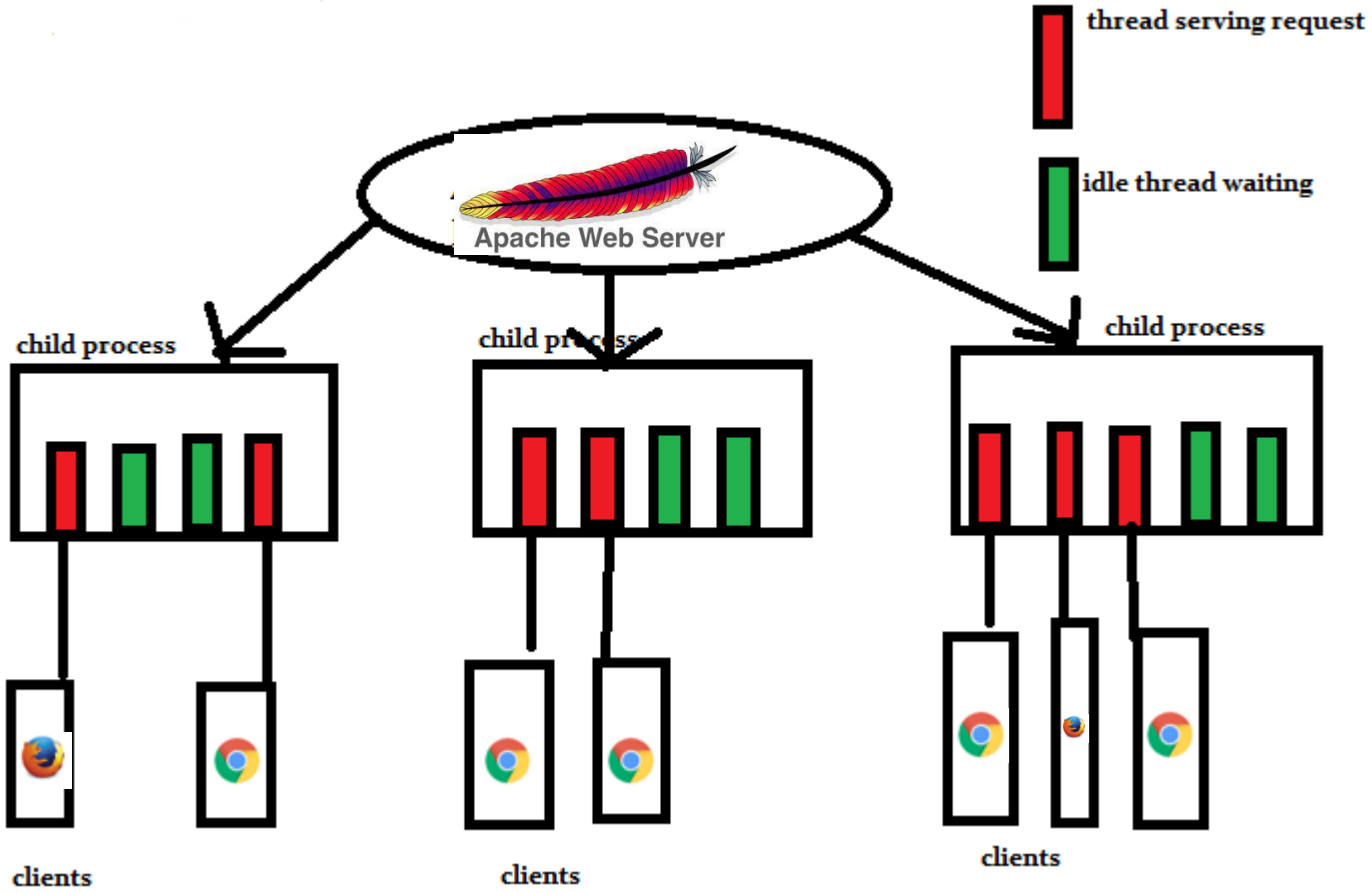
I primi 4 tab sono implementati in processi diversi. I successivi sono implementati tramite **thread paralleli**.

Prestazioni minori ma risparmio di risorse.

Operating Systems: Thread

Thread in processi server (daemon): esempio

Il web server di Apache (httpd: HTTP Daemon) crea un certo numero di figli ed in ognuno di essi viene creato un certo numero di **thread paralleli**.



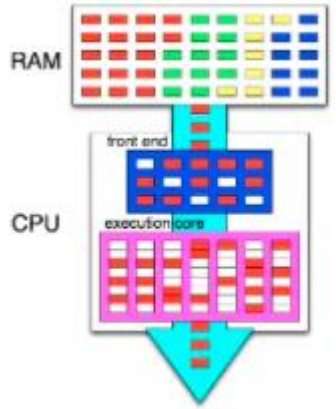
Operating Systems: Thread

Vantaggi/Svantaggi

Ambito	Vantaggi	Svantaggi
Tempi di Attesa	Un programma può continuare la computazione anche se un suo thread è bloccato (e.g. attesa I/O)	
Condivisione delle Risorse	• condivisione delle informazioni tramite IPC (memoria condivisa o messaggi) • condivisione della memoria e delle risorse del processo che li genera (i.e. molti thread nello stesso spazio di indirizzi)	Difficoltà di ottenere risorse private: per ottenere memoria privata all'interno di un thread è possible ricorrere ad appositi meccanismi
Concorrenza	Creazione e context switch molto più leggeri rispetto a quelli dei processi	Pericolo di interferenza • la condivisione delle risorse accentua il pericolo di interferenza • gli accessi concorrenti devono essere sincronizzati di modo da evitare interferenze (thread safeness)
Scalabilità	I thread possono essere eseguiti in parallelo su architettura multiprocessore	

Operating Systems: Thread

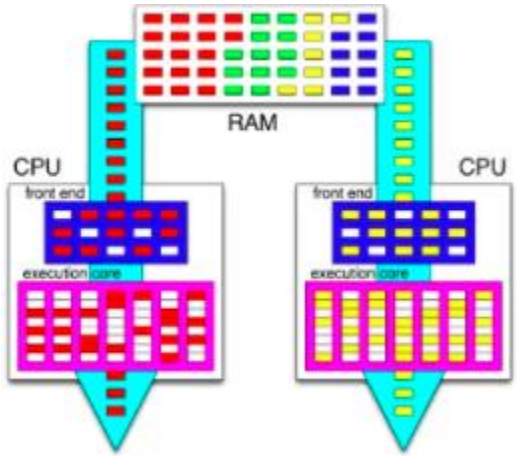
Programmazione Multicore



Single-Core: In un sistema con un'unica unità di calcolo l'esecuzione concorrente consiste nell'interfogliazione dei thread

Multi-Core (es. SMP: Symmetric MultiProcessor) l'esecuzione concorrente consente ai thread di essere eseguiti in parallelo. I SO e le appicazioni devono essere appositamente progettate:

- **SO**: devono fornire schedulatori che utilizzano diverse unità di calcolo
- **Applicazioni**: devono essere progettate tenendo in considerazione diversi fattori:
  - **Separazione dei task**: individuazione dei task che possono essere eseguiti in parallelo
  - **Bilanciamento dei task**: devono eseguire compiti che richiedono all'incirca tempo e risorse paragonabili
  - **Suddivisione dei dati**: devono essere definiti dei dati specifici per i vari task
  - **Dipendenza dei dati**: l'accesso ai dati condivisi deve essere fatto in modo sincronizzato (thread safeness)
  - **Test e debugging**: a causa dei diversi flussi di esecuzione test e debugging sono più difficili

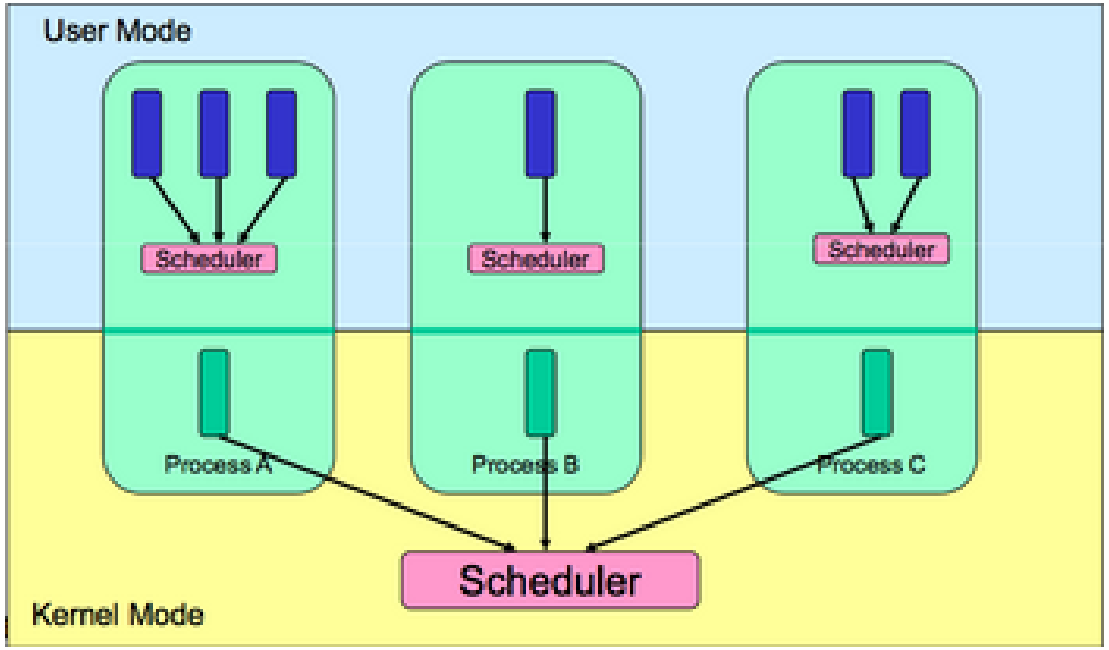


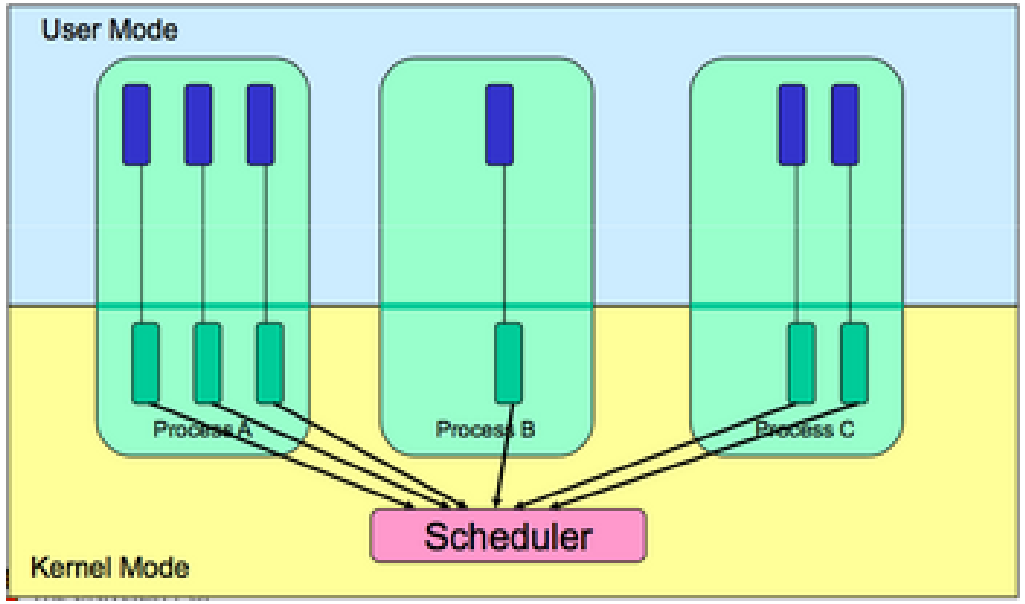
Operating Systems: Thread

Modello di Thread: User vs Kernel Level

User-Level Thread: thread all'interno del processo utente gestiti da una libreria specifica (da un supporto run-time)

- sono gestiti senza l'aiuto del kernel
- lo switch non richiede chiamate al kernel



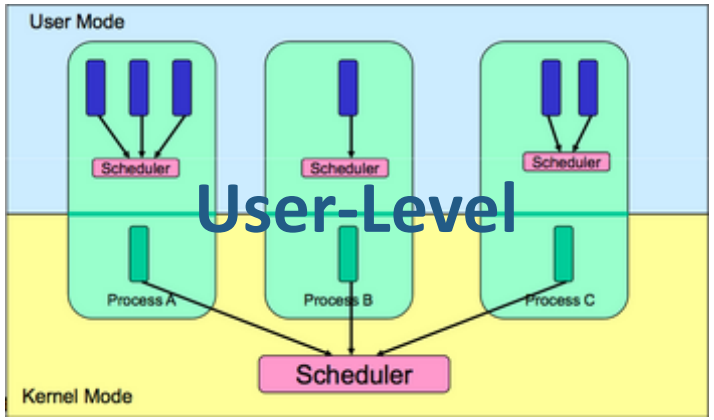
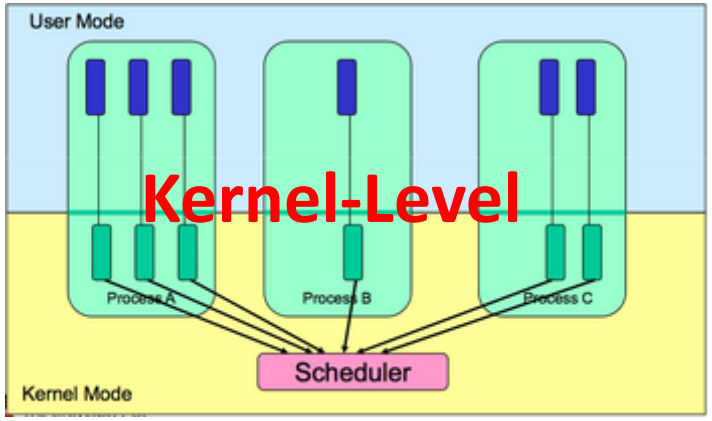


Kernel-Level Thread: gestione dei thread affidata al kernel tramite chiamate di sistema

- gestione integrata processi e thread dal kernel
- lo switch è provocato da chiamate al kernel

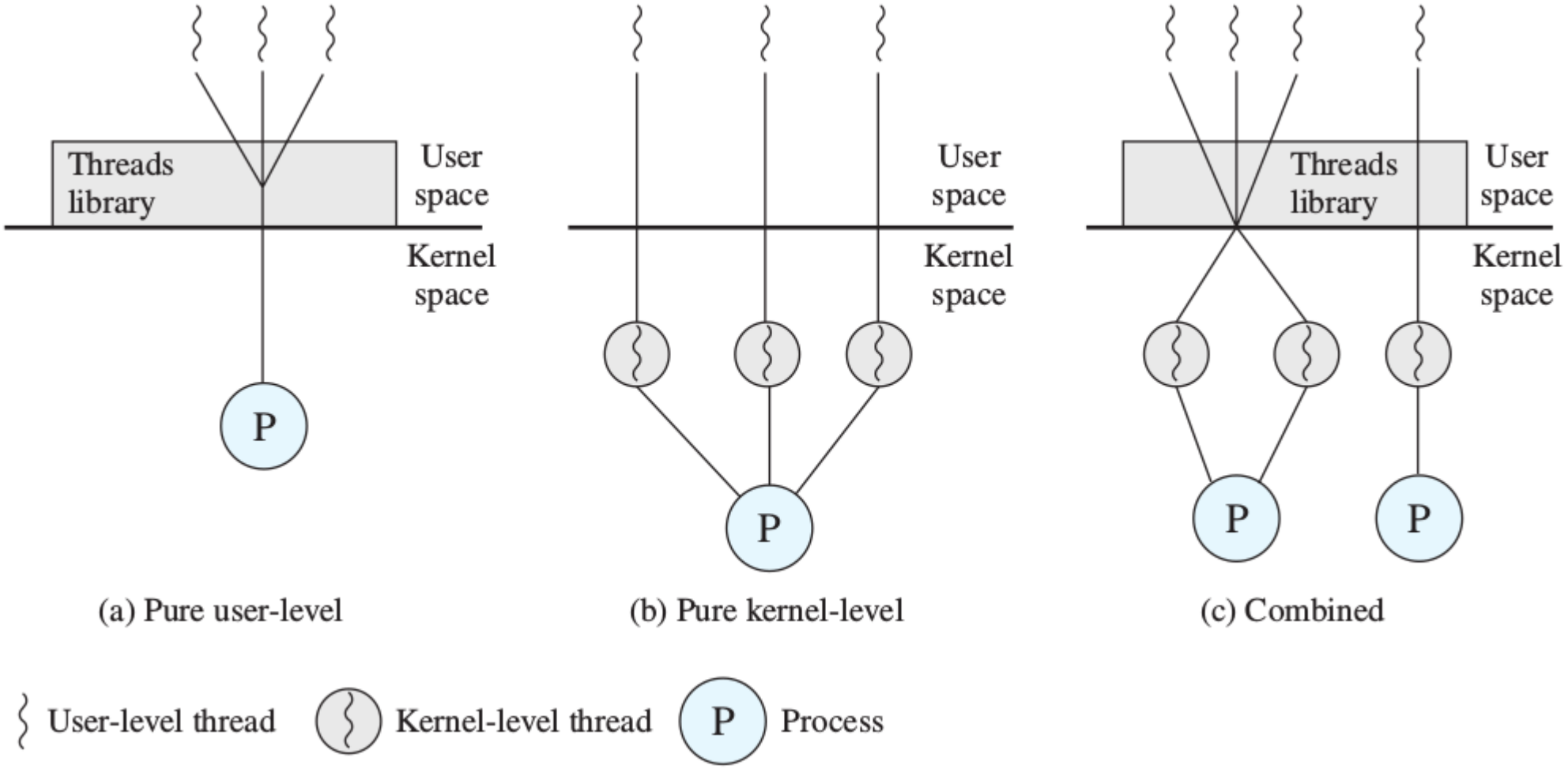
Operating Systems: Thread

User vs Kernel Level

Ambito	Vantaggi	Svantaggi
	• Lo switch non coinvolge il kernel, e quindi non ci sono cambiamenti della modalità di esecuzione • Maggiore libertà nella scelta dell'algoritmo di scheduling che può anche essere personalizzato • Poiché le chiamate possono essere raccolte in una libreria, c'è maggiore portabilità tra SO	• Una chiamata al kernel può bloccare tutti i thread di un processo, indipendentemente dal fatto che in realtà solo uno dei suoi thread ha causato la chiamata bloccante • In sistemi a multiprocessore simmetrico (SMP) due processori non possono essere associati a due thread del medesimo processo
	• Il kernel può eseguire più thread dello stesso processo anche su più processori • Il kernel stesso può essere scritto multithread	• Lo switch coinvolge chiamate al kernel e questo comporta un costo • L'algoritmo di scheduling è meno facilmente personalizzabile • Meno portabile

Operating Systems: Thread

ULT e/o KLT



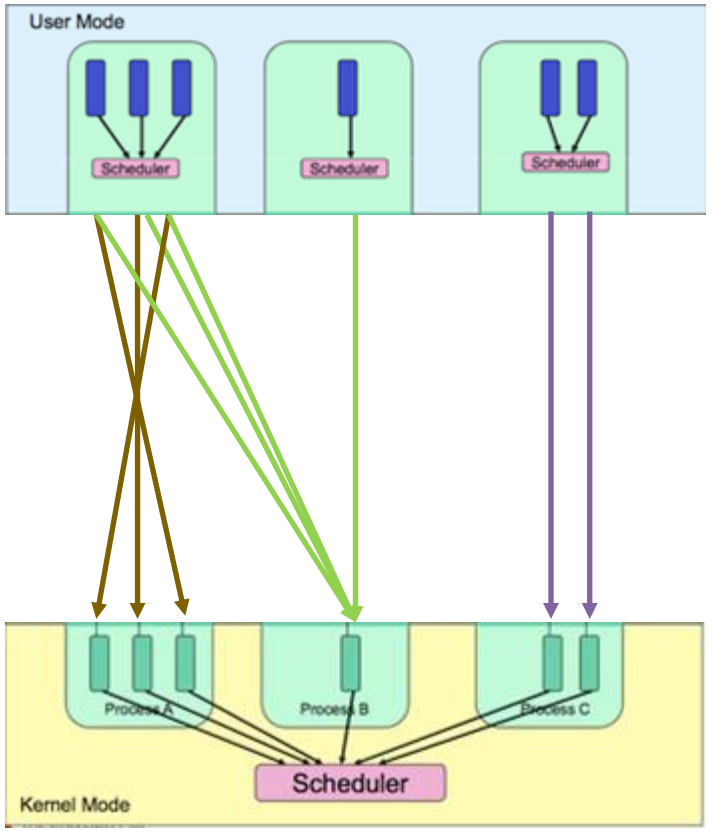
Operating Systems: Thread

Modello multi-Thread

I **thread a livello utente**

vengono messi in relazione per permettere l'accesso alle risorse

con **I thread a livello kernel**

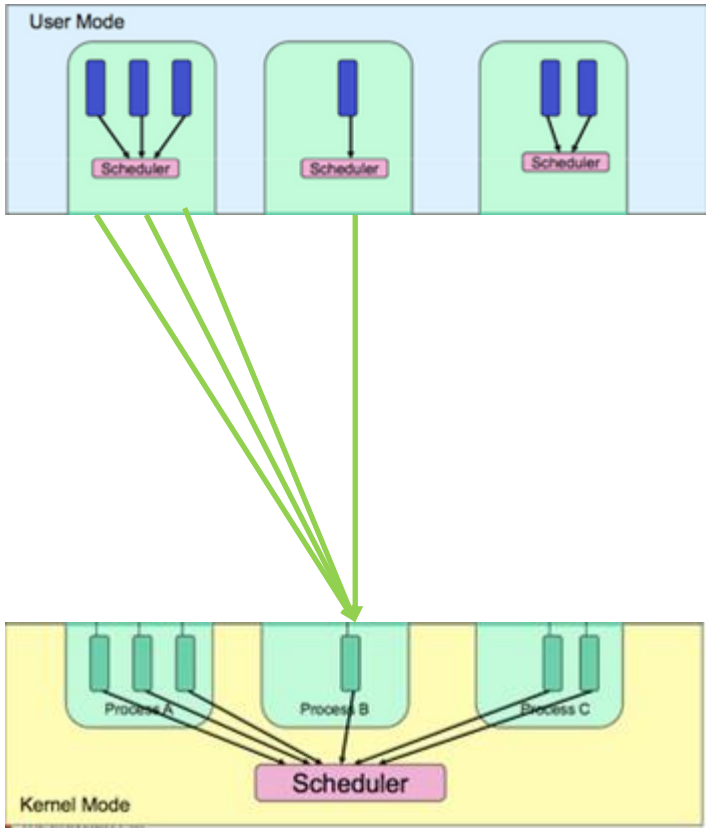


Esistono diversi tipi di relazione tra thread a livello utente e thread a livello kernel:

- **Molti-a-uno**
- **Uno-a-uno**
- **Molti-a-molti**
- **A due livelli**

Operating Systems: Thread

multi-Thread: Molti-a-Uno



Molti-a-uno

riunisce molti thread di livello utente in un unico kernel thread

Vantaggi

- Gestione dei thread efficiente

Svantaggi

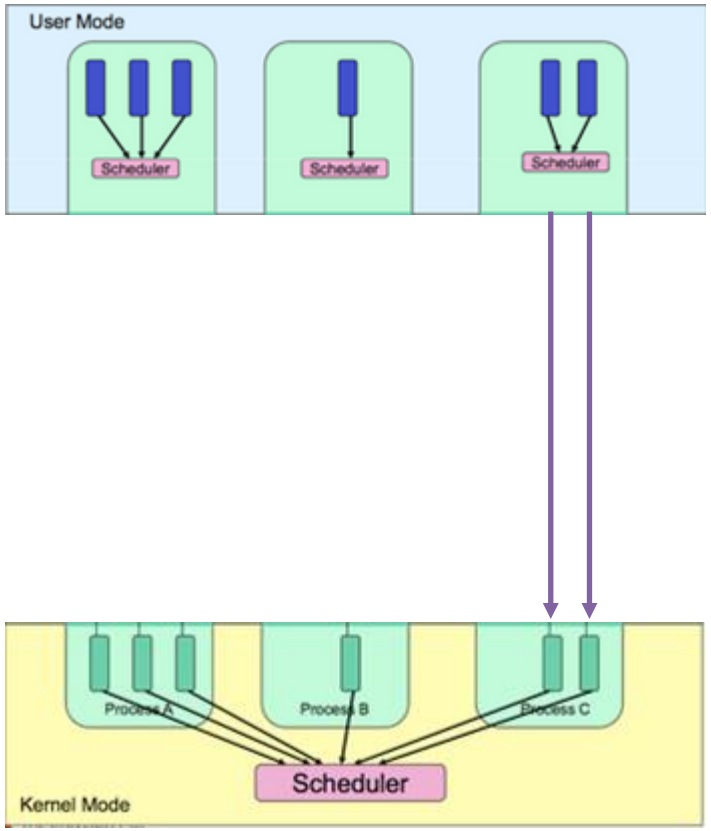
- Intero processo bloccato se un thread invoca una chiamata di sistema bloccante
- Un solo thread può accedere al Kernel (Impossibile eseguire thread in parallelo)

Esempi:

- Solaris Green Threads
- GNU Portable Threads

Operating Systems: Thread

multi-Thread: Uno-a-Uno



Uno-a-uno

mappa ciascun thread utente in un kernel thread

Vantaggi

- Maggiore concorrenza (possibili thread in parallelo e le chiamate bloccanti non bloccano tutti i thread)

Svantaggi

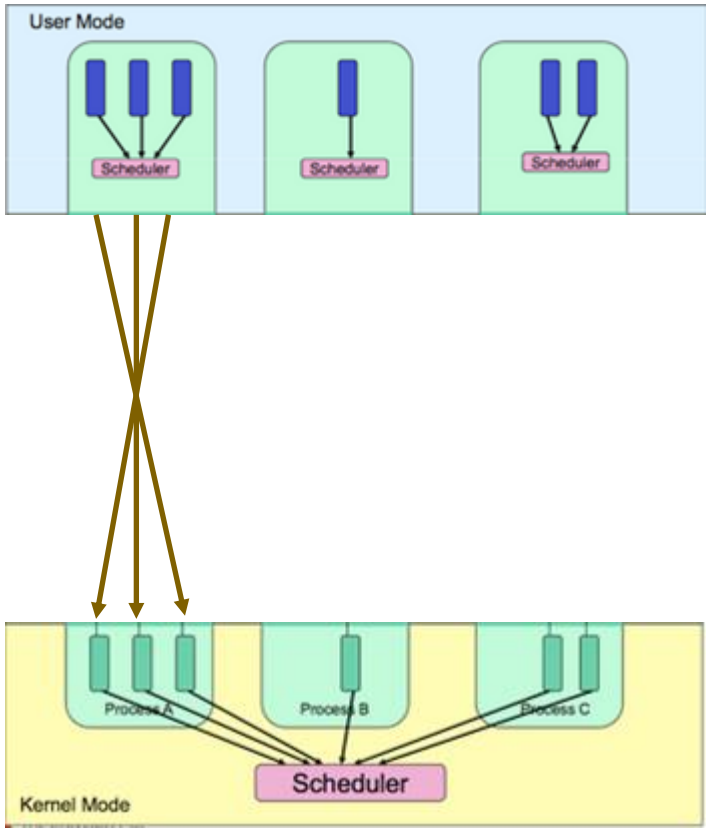
- La creazione di molti thread a livello kernel compromette le prestazioni dell'applicazione (es. limite del numero di thread a livello kernel)

Esempi:

- Windows NT/XP/2000
- Linux
- Solaris 9

Operating Systems: Thread

multi-Thread: Molti-a-Molti



Molti-a-molti

Permette di aggregare molti thread a livello utente verso un numero più piccolo o equivalente di kernel thread

Vantaggi

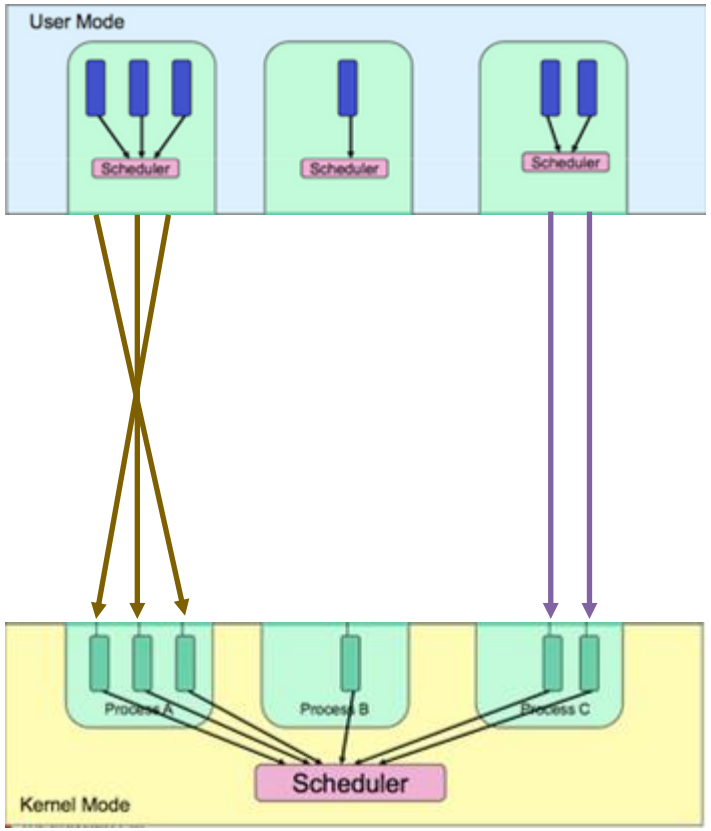
- Risolve le limitazioni dei modelli precedenti (Il numero max di processi a livello Kernel può essere personalizzato in base all'architettura; n_max maggiore per multicore)

Esempi:

- Solaris, versioni precedenti alla 9
- Windows NT/2000 (pacchetto ThreadFiber)

Operating Systems: Thread

multi-Thread: a Due Livelli



A due livelli

Simile al modello molti-a-molti, eccetto che permette anche di associare un thread di livello utente ad un kernel thread

Esempi:

- IRIX
- HP-UX
- Tru64 UNIX

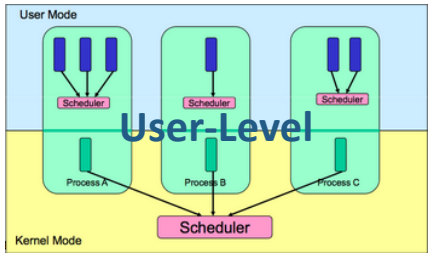
Operating Systems: Thread

Librerie per i Thread

La gestione dei thread è effettuata attraverso specifiche librerie

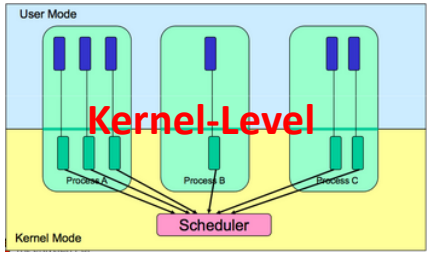
Libreria collocata a livello utente (POSIX Pthreads)

- Codice e strutture dati per la libreria risiedono nello spazio utente
- Invocare una funzione della libreria non significa invocare una chiamata di sistema



Libreria collocata a livello kernel (POSIX Pthreads, Win32 thread)

- Codice e strutture dati per la libreria risiedono nello spazio del kernel
- Invocare una funzione della libreria significa invocare una chiamata di sistema



N.B. Java thread (dipende dal sistema operativo ospitante)

Operating Systems: Thread

Librerie per i Thread: Pthread – User Level Thread

standard POSIX (IEEE 1003.1c) API per la creazione e la sincronizzazione dei thread

- Fornisce una specifica per il comportamento della libreria dei thread
- i progettisti di sistemi operativi possono implementare la specifica nel modo che desiderano
- Frequente nei sistemi operativi di UNIX (Solaris, Linux, Mac OS X)

```
#include <pthread.h>
```

`pthread_t`: specifica l'identificatore di un thread
`pthread_attr`: specifica gli attributi di un thread (e.g. proprietà per lo scheduling)
`pthread_attr_init(&pthread_attr)`: inizializza gli attributi di un thread ai valori di default
`pthread_create(pthread_t *tid, pthread_attr *attr, void*(), void*argv)`: crea un user thread al quale associare gli attributi inizializzati, la funzione da eseguire e gli argomenti
`pthread_join (pthread_t *tid, <value>)`: mette in pausa il thread corrente fino alla terminazione del thread tid. Se il secondo parametro non è nullo viene usato per restituire il valore di ritorno del thread

Operating Systems: Thread

System Call fork() ed exec()

Solitamente, vi sono varie versioni della funzione `fork()` a seconda che si desideri:

- duplicare solo il thread che la invoca
- Duplicare tutti i thread del processo ospitante

In linux esiste la funzione `clone()`: una nuova chiamata di sistema versatile che può essere utilizzata per creare un nuovo thread di esecuzione. A seconda delle opzioni passate, il nuovo thread di esecuzione può aderire alla semantica di un processo UNIX, un thread POSIX, una via di mezzo o qualcosa di completamente diverso (come un contenitore diverso). È possibile specificare tutti i tipi di opzioni che determinano se la memoria, i descrittori di file, i vari spazi dei nomi, i gestori dei segnali e così via vengano condivisi o copiati. La funzione `fork()` è attualmente implementata come wrapper che richiama `clone()`.

La funzione `exec()`, viceversa, funziona analogamente a quanto visto prima della introduzione dei thread: rimpiazza l'intero processo, quindi anche tutti i thread.

Operating Systems: Thread

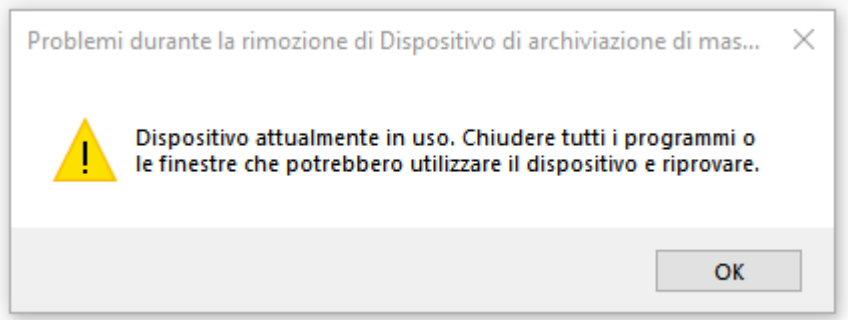
Cancellazione dei Thread

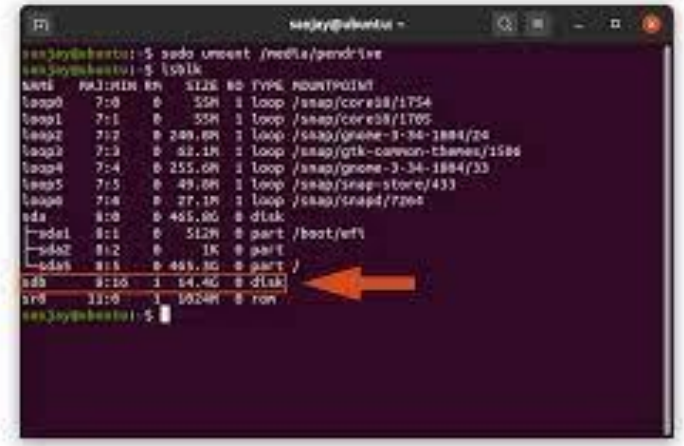
Terminazione di un thread prima che abbia completato l'esecuzione

Può avvenire in due differenti scenari:

- **Cancellazione asincrona**: un thread termina immediatamente il thread target (se il thread sta operando su un file potrebbe portare ad una situazione in cui la risorsa non è libera nonostante il processo sia terminato)

- **Cancellazione differita**: il thread target può periodicamente controllare se deve terminare, cosi da terminare in modo opportuno (se il thread sta operando su un file la risorsa viene liberata opportunamente)

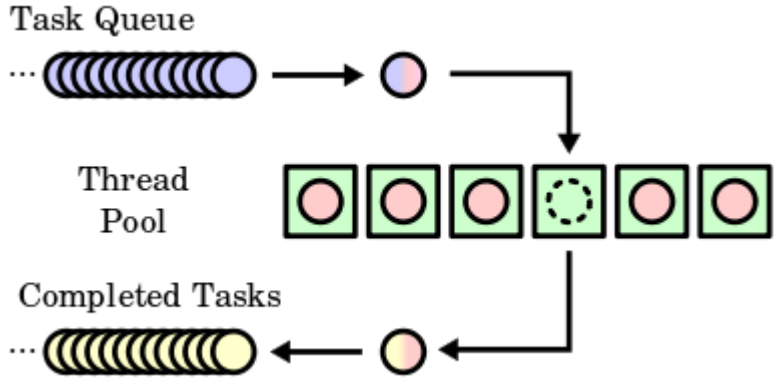




Operating Systems: Thread

Thread Pool

Thread Pool: creare un gruppo di thread (pool) all'avvio del processo ed assegnarli un lavoro quando richiesto. Al completamento del lavoro il thread torna nel gruppo d'attesa.

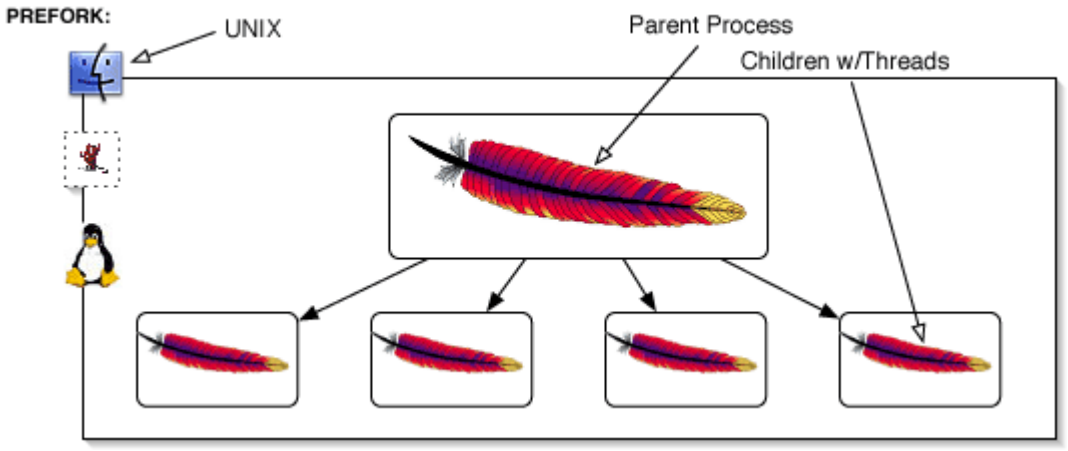


Vantaggi:

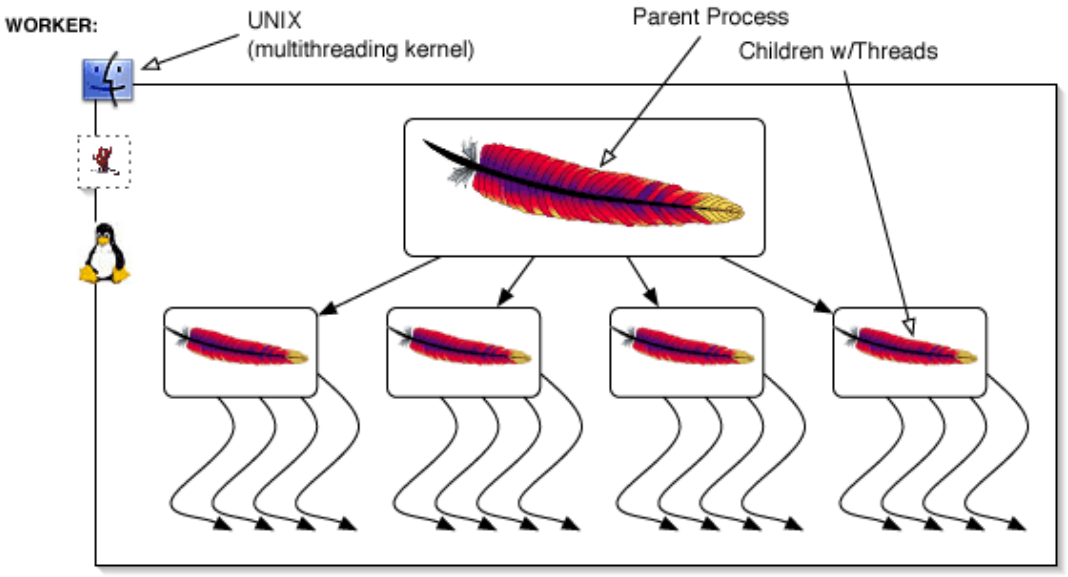
- Servire la richiesta all'interno di un thread esistente è tipicamente più veloce che attendere la creazione di un thread
- Un pool di thread limita il numero di thread esistenti in contemporanea (Il numero massimo può essere adattato a runtime in architetture raffinate)

Operating Systems: Thread

Thread Pool: es. Apache



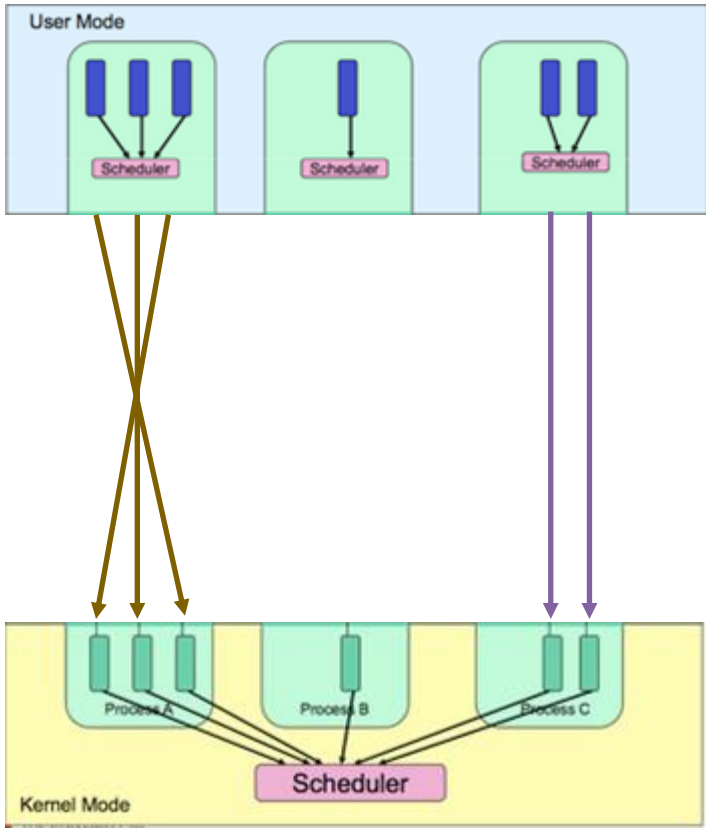
Prefork (no thread): Apache 1.x.y si avvia come processo server padre e da esso vengono generati i processi figli. Il server mantiene alcuni figli generati in esecuzione, per gestire le richieste in arrivo, e il server può uccidere i processi figli se ne vengono avviati troppi, evitando che il server vada in crash.



Worker (thread): Apache 2.x.y fa uso del worker, molto simile al modulo PREFORK, ma è un MPM (multi-Processing Module) ibrido thread/processo. Questo significa che si comporta come PREFORK generando figli, e questi figli generano un numero statico di thread. Il worker ha, quindi, un thread per ogni figlio che ascolta la rete, e ogni processo può servire molti processi contemporaneamente.

Operating Systems: Thread

Scheduling: Attivazione 1/3



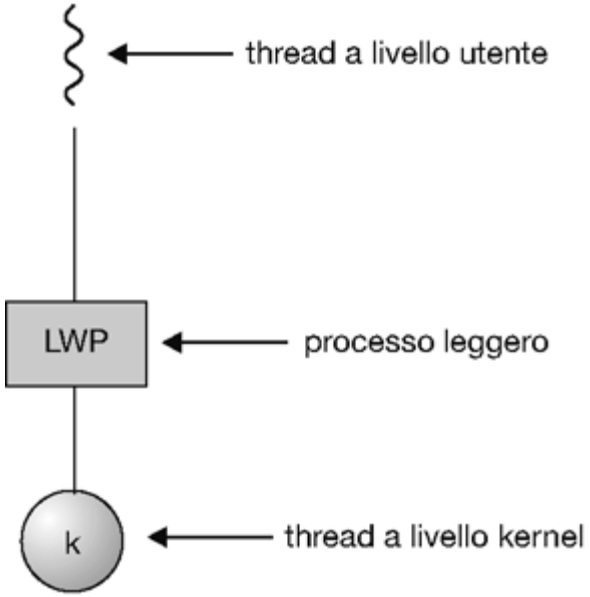
Le applicazioni multi-thread con modello molti-a-molti o a due livelli richiedono una comunicazione con il kernel:

- per mantenere l'appropriato numero di kernel thread allocati (prestazioni)
- per far sì che il kernel informi l'applicazione di certi eventi (segnali), come il blocco di un thread dell'applicazione (ad es. per un evento di I/O)

Operating Systems: Thread

Scheduling: Attivazione 2/3

Struttura dati posta tra i thread kernel e i thread utente: **LightWeight Process** (**LWP**)



Il LWP è visto dalla libreria dei thread utente come un processore virtuale su cui schedulare i thread utente

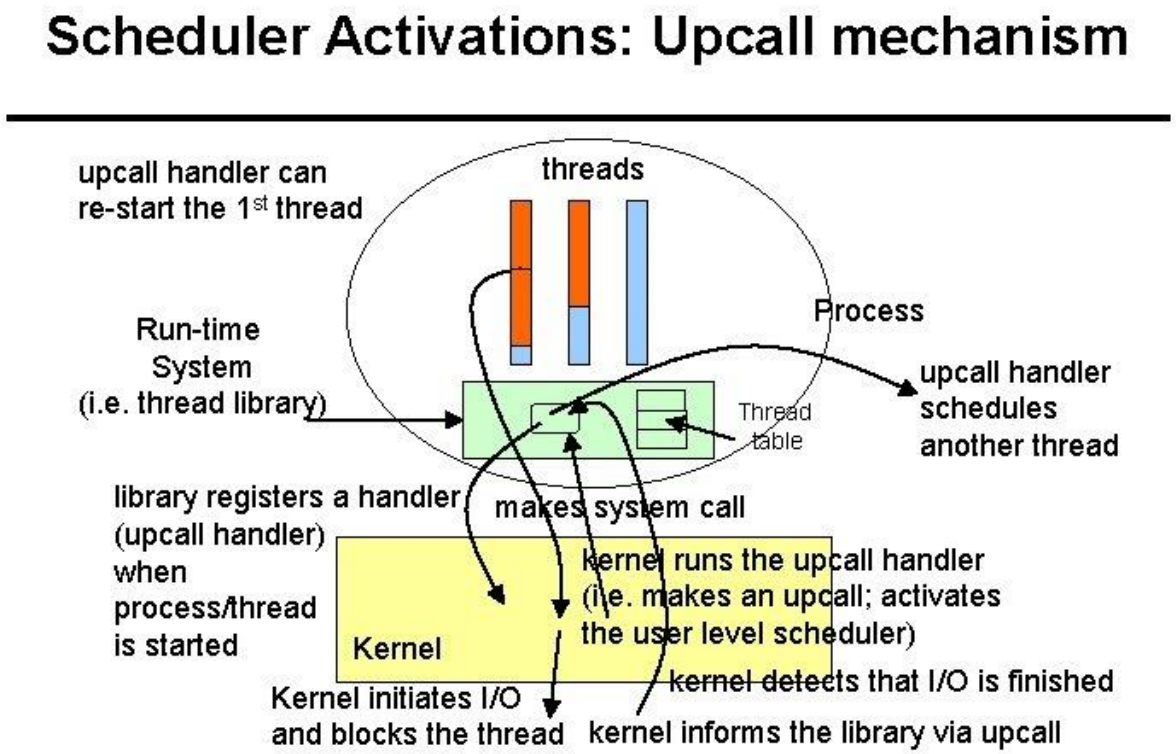
Il blocco di un thread kernel (e.g. I/O) causa il blocco del LWP ➔ si bloccano i thread utente schedulati sul LWP ➔ necessario un LWP per ogni chiamata di sistema concorrente bloccante
(e.g. se un'applicazione ha bisogno di fare 5 letture da file servono 5 LWP)

Operating Systems: Thread

Scheduling: Attivazione 3/3

I thread kernel possono comunicare eventi alla libreria dei thread tramite le **upcall** (chiamate al thread)

Le **upcall** sono gestite a <u>livello utente</u> con un gestore di upcall in esecuzione “just in time” (cioè quando avviene una comunicazione dal kernel all’applicazione) su di un LWP



Operating Systems: Thread

es. Thread in Windows

Implementa la mappatura uno-a-uno. Ogni thread contiene:

- Un identificatore del thread
- Il contesto del thread: set di registri (stato del processore), User stack, Kernel stack, area di memoria privata

ETHREAD (kernel): blocco di esecuzione del thread

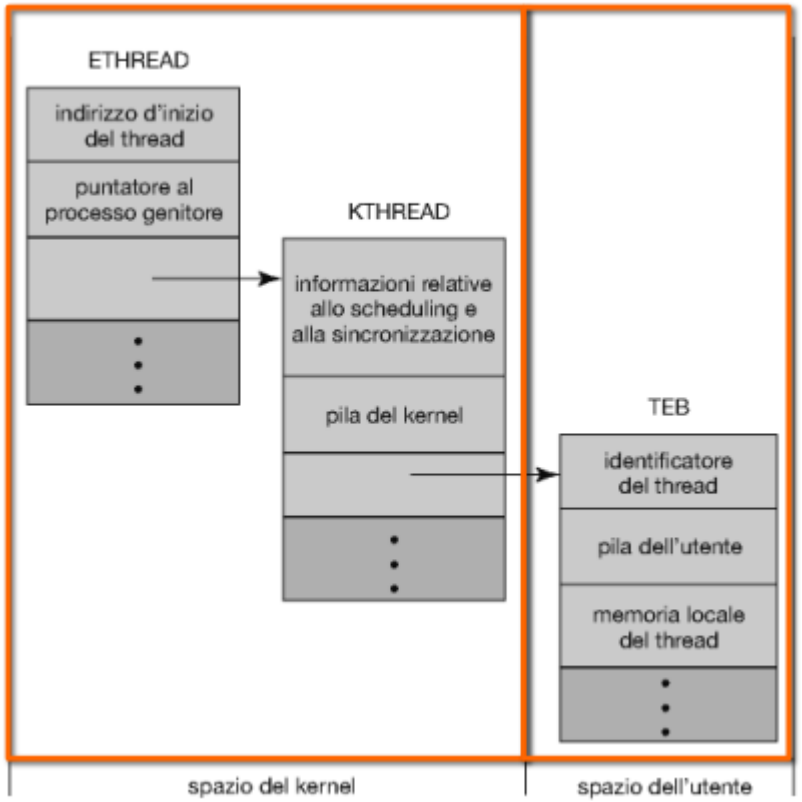
- Puntatore al processo padre
- Indirizzo della funzione eseguita dal thread

KTHREAD (kernel): blocco di kernel del thread

- Informazioni relative allo scheduling e sincronizzazione
- Pila del kernel

Thread Environment Block TEB (user): blocco di ambiente del thread

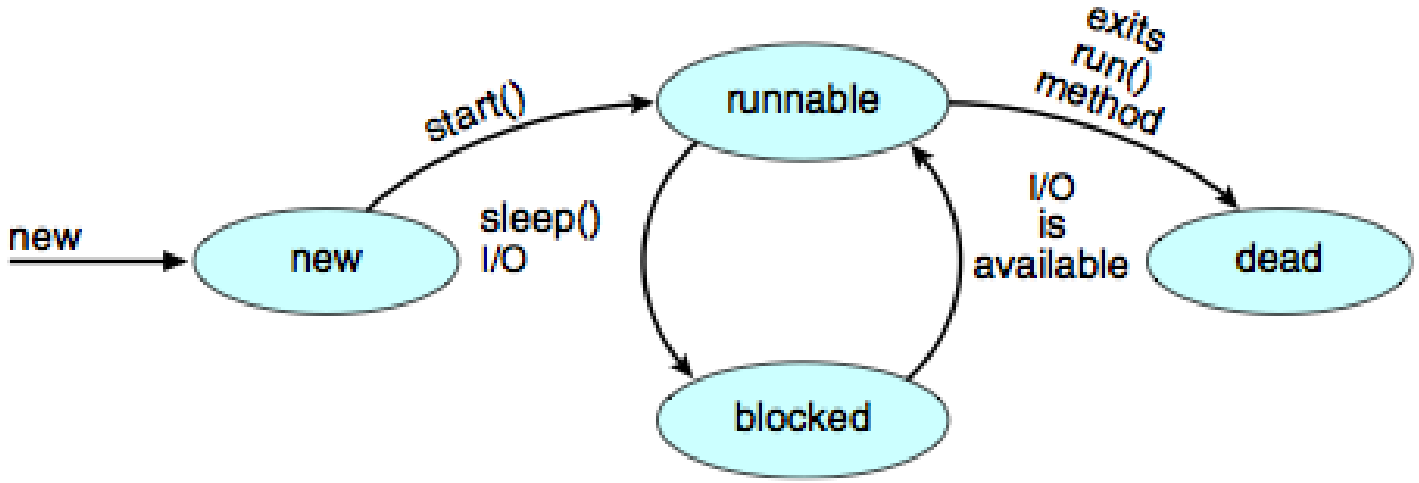
- Pila dell'utente
- ID del thread
- Vettore per dati specifici del thread non condivisi (memoria locale)



Operating Systems: Thread

es. Thread in Java

I thread di Java sono gestiti dalla **JVM**



I thread di Java possono essere creati tramite:

- L'estensione della classe **Thread**
- L'interfaccia **Runnable**

Operating Systems: Thread

es. Thread in Linux

Linux li definisce come ***task*** piuttosto che *thread*

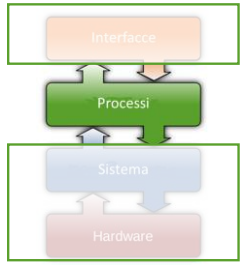
La creazione di un thread avviene attraverso la chiamata di Sistema `clone()`
Questa permette di stabilire il "grado di condivisione" tramite flag parametri

Flag	Significato
CLONE_FS	Condivisione delle informazioni sul file system
CLONE_VM	Condivisione dello stesso spazio di memoria
CLONE_SIGHAND	Condivisione dei gestori dei segnali
CLONE_FILES	Condivisione dei file aperti

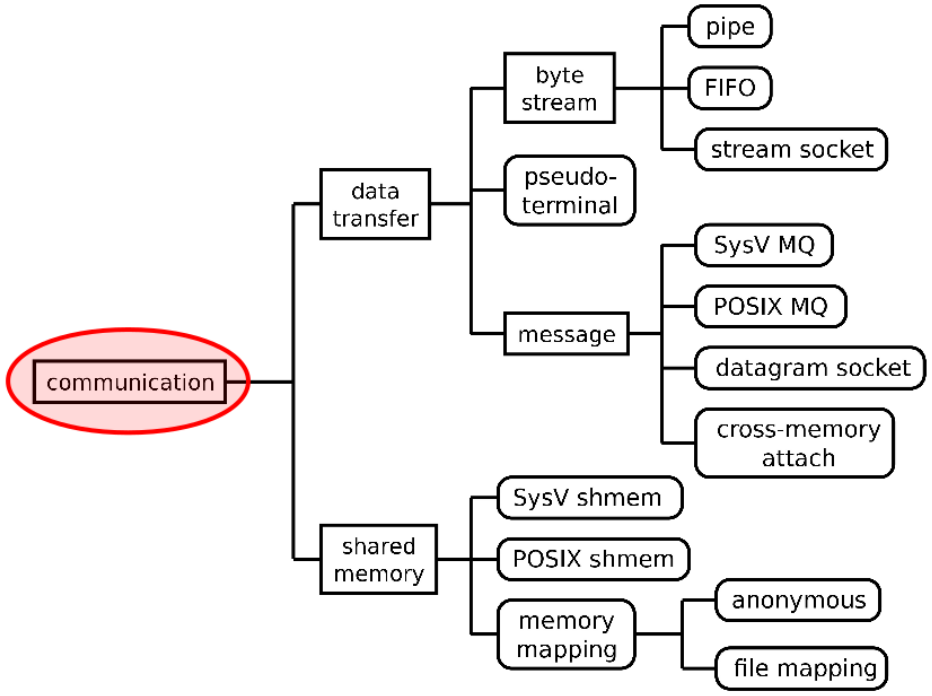
se nessun flag è impostato, non c'è alcuna condivisione e l'effetto di `clone()` è simile a quello della `fork()` (funzione "wrapper")

Operating Systems: Obiettivi Funzioni Servizi

IPC



Communication



Inter-Process Communication

- **Segnale** (signals), per eventi asincroni
- **Pipe**, per reindirizzare il risultato della elaborazione
- **Socket**, scambio di datagram
- **Code di Messaggi** (Message Queue), per comunicazioni in multicast
- **Memoria Condivisa** (Shared Memory), per condividere dati fra processi con trust reciproco
- **Semaforo** (Semaphore), per coordinare processi che lavorano su una stessa risorsa (-> critical region)

Permettere ai processi di comunicare fra di loro.

SAPIENZA UNIVERSITÀ DI ROMA

Operating Systems: IPC
Processi Indipendenti o Cooperanti

Un **processo indipendente** non può influenzare o essere influenzato dagli altri processi in esecuzione nel sistema

➔ non condivide risorse con altri processi

Un **processo cooperante** può influenzare o essere influenzato da altri processi in esecuzione nel sistema

➔ condivide risorse con altri processi

(es. Client-Server, Compiler-Assembler-Loader, Produttore-Consumatore)

Operating Systems: IPC
Cooperazione

Cooperazione = lavoro congiunto di processi per raggiungere scopi applicativi comuni con condivisione e scambio di informazioni

Vantaggi del processo di cooperazione:

- Condivisione delle informazioni (ad es. un file o una directory condivisa) ➔ Accesso concorrente (!!!)
- Velocizzazione della computazione (possibile in verità solo con più CPU o canali di I/O) ➔ Esecuzione di sotto-attività in parallelo
- Modularità: dividendo le funzioni del sistema in processi o thread separati (es. sistemi operativi modulari)
- Convenienza (un singolo utente può lavorare su molte attività) ➔ Multi tasking

Operating Systems: IPC
Processi Cooperanti

- Hanno uno scopo applicativo comune
- Possono condividere informazioni ➔ **Comunicazione**
- Possono influenzare o essere influenzati da altri processi ➔ **Sincronizzazione**

Operating Systems: IPC

Inter- Process Communication: Caratteristiche

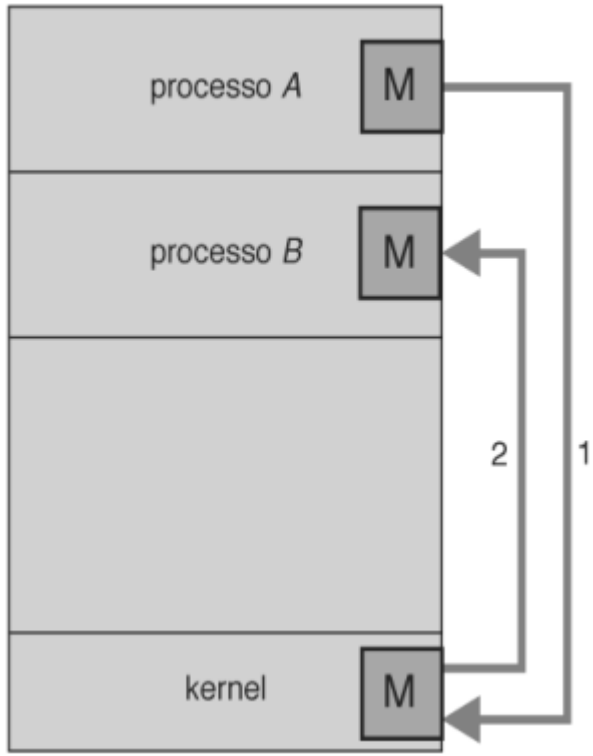
- Quantità di informazioni da trasmettere
- Velocità di esecuzione
- Scalabilità
- Semplicità di uso nelle applicazioni
- Omogeneità delle comunicazioni
- Integrazione nel linguaggio di programmazione
- Affidabilità
- Sicurezza
- Protezione

Operating Systems: IPC

IPC: Implementazioni a Scambio di Messaggi

- Quantità di informazioni da trasmettere
- Velocità di esecuzione
- Scalabilità
- Semplicità di uso nelle applicazioni
- Omogeneità delle comunicazioni
- Integrazione nel linguaggio di programmazione
- Affidabilità
- Sicurezza
- Protezione

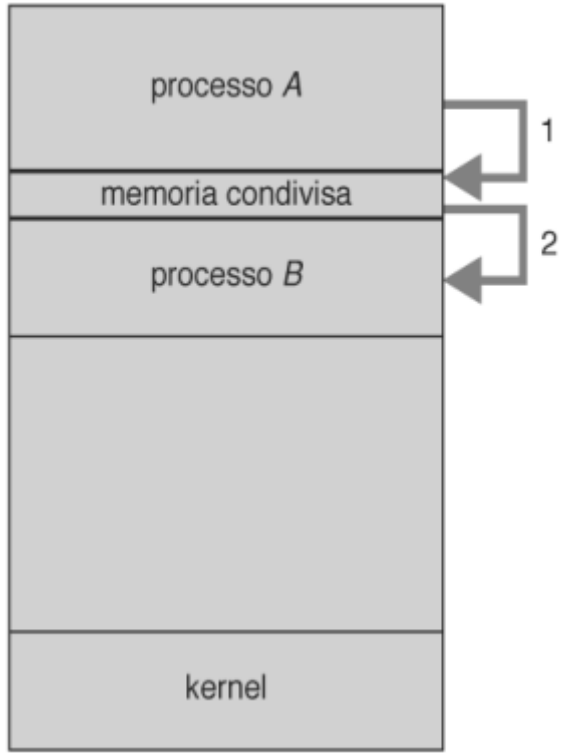
(es. Segnale, Socket, Message Queue, Semaforo)



Operating Systems: IPC

IPC: Implementazioni a Memoria Condivisa

- Quantità di informazioni da trasmettere
- Velocità di esecuzione
- Scalabilità
- Semplicità di uso nelle applicazioni
- Omogeneità delle comunicazioni
- Integrazione nel linguaggio di programmazione
- Affidabilità
- Sicurezza
- Protezione



(es. Pipe, Memoria Condivisa)

Operating Systems: IPC

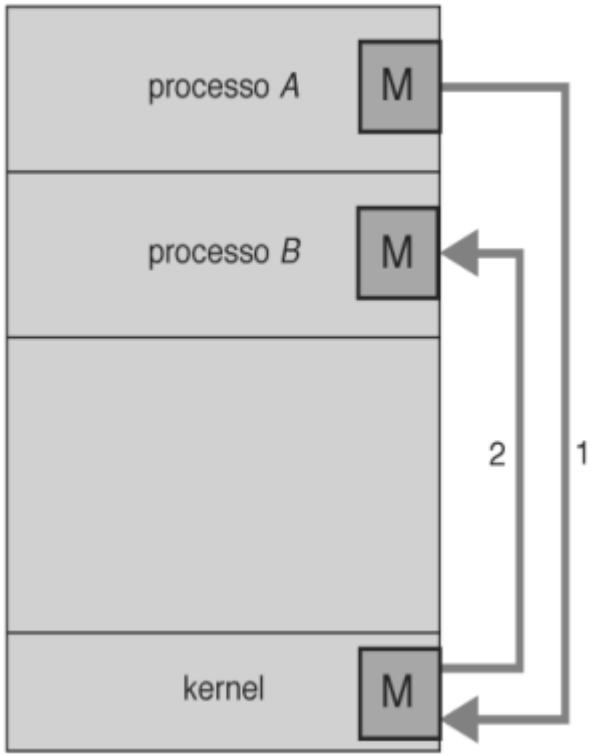
IPC: Implementazioni a Scambio di Messaggi

Segnale

Socket

Message Queue

Semaforo



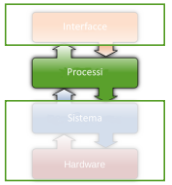
Operating Systems: IPC

Gestione dei Segnali

Nei sistemi UNIX-like per notificare ad un processo che si è verificato un particolare evento si usa un segnale

Operating Systems: Obiettivi Funzioni Servizi

IPC: Signal in Unix



Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at terminal
SIGTTIN	21,21,26	Stop	Terminal input for background process
SIGTTOU	22,22,27	Stop	Terminal output for background process

The signals **SIGKILL** and **SIGSTOP** cannot be caught, blocked, or ignored.

Permettere ai processi di comunicare fra di loro.

SAPIENZA UNIVERSITÀ DI ROMA

- **Sincroni**: sono inviati allo stesso processo che ha causato la generazione del segnale (es. divisione per 0, accesso illegale alla memoria)
- **Asincroni**: sono inviati ad un processo differente da quella cha ha causato la generazione del segnale (es. intercettazione di una combinazione di tasti (ctrl + c,ctrl+alt+canc, scadenza di un timer, etc)

Operating Systems: IPC

Gestione dei Segnali con i Thread

I segnali vengono elaborati secondo questo schema:

1. il verificarsi di un particolare evento genera un segnale
2. il segnale generato viene consegnato ad un processo
3. il segnale viene gestito

I segnali possono essere gestiti attraverso

- Il gestore predefinito dello specifico segnale (esiste per ogni segnale)
- Un funzione di gestione definita dall'utente (override del gestore predefinito)

Per i processi a singolo thread la gestione dei segnali è semplice

Per i processi multithread è necessario decidere a quale thread inviare il processo. Diverse opzioni:

1. **UNO**: Consegnare il segnale al thread a cui il segnale viene applicato (sempre per segnali asincroni)
2. **TUTTI**: Consegnare il segnale ad ogni thread del processo (istruire i processi su quali segnali sincroni accettare e quali ignorare)
3. **MOLTI**: Consegnare il segnale al alcuni thread del processo (istruire i processi su quali segnali sincroni accettare e quali ignorare)
4. **Specifico**: Designare un thread specifico che riceva tutti i segnali per il processo

Operating Systems: IPC

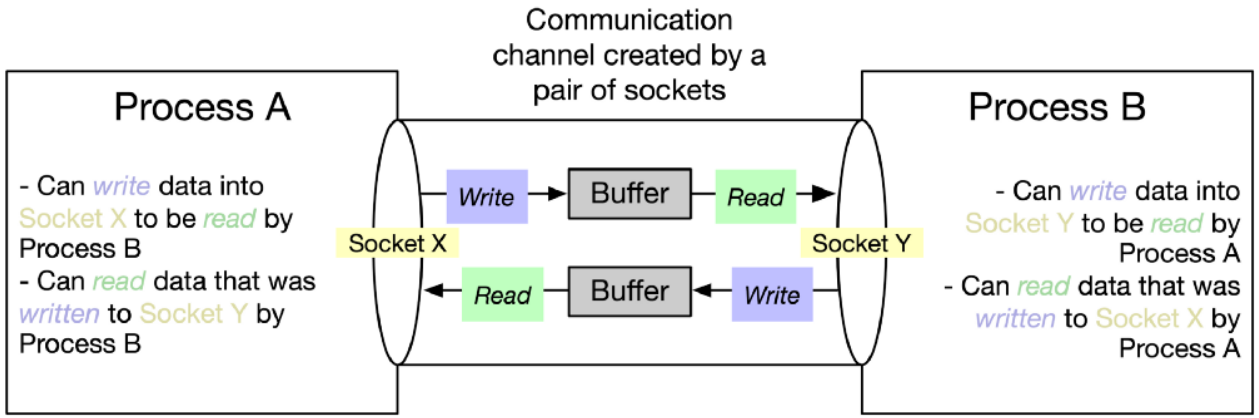
Gestione dei Socket

Socket = estremo di un canale di comunicazione

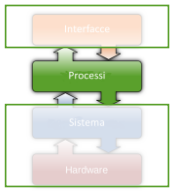
Operating Systems: Obiettivi Funzioni Servizi

IPC: Socket (stream/datagram) in Unix

One-way communication
Flusso (stream) o blocchi di dati (Datagram) Unidirezionale



Permettere a 2 processi di scambiarsi informazioni (anche di notevoli dimensioni) fra loro.



Tutte le connessioni devono essere uniche: se un processo vuole n connessioni verso un server deve creare n socket

Vantaggi:

- Semplice
- Efficiente

Svantaggi:

- Di basso livello: permette la trasmissione di un flusso non strutturato di byte
- È responsabilità di Client e Server interpretare e organizzare i dati in forme complesse

SAPIENZA UNIVERSITÀ DI ROMA

Operating Systems: IPC

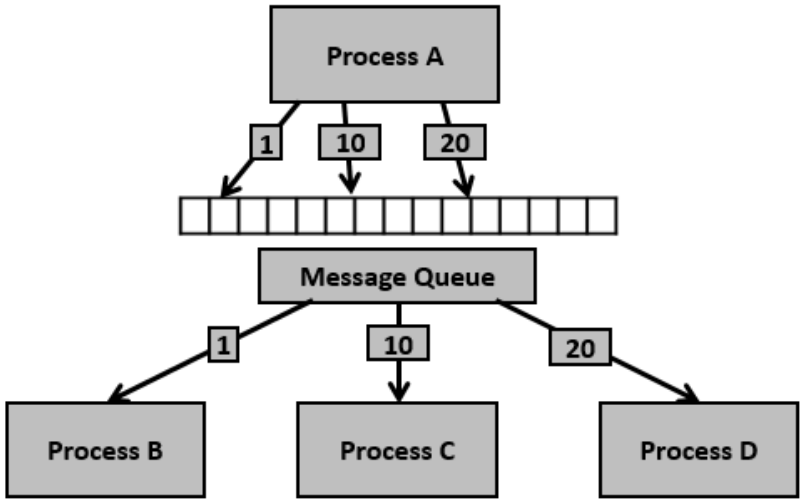
Gestione delle Code di Messaggi

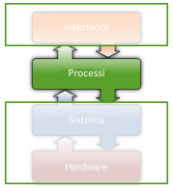
Il SO implementa codice per la realizzazione e la gestione della risorsa condivisa

Operating Systems: Obiettivi Funzioni Servizi

IPC: Message Queue in Unix

Multicast communication





Sincronizzazione per l'accesso ai messaggi gestita implicitamente dal SO fornendo due operazioni:

- send (messaggio)
- receive (messaggio)

Permettere a molteplici processi di scambiarsi informazioni non riservate.

Operating Systems: IPC

Gestione delle Code di Messaggi

Contenuto messaggio

- Processo mittente
- Processo destinatario
- Informazioni da trasmettere
- Eventuali altre informazioni di gestione dello scambio messaggi

Dimensione messaggio

- Fissa
- Facile implementazione a livello SO, difficile a livello applicazione
- Variabile
- Difficile implementazione a livello SO, facile a livello applicazione

Operating Systems: IPC

Gestione delle Code di Messaggi: Canali

Due processi che vogliono comunicare, devono:

- stabilire un canale di comunicazione tra di loro
- scambiare messaggi mediante send/receive

Implementazione di un canale di comunicazione:

- Fisica (come memoria condivisa, hardware bus) o
- Logica (come le proprietà logiche)

1. La denominazione dei processi
   - comunicazione diretta (simmetrica o asimmetrica)
   - comunicazione indiretta
2. La sincronizzazione: Il passaggio dei msg può essere bloccante oppure non bloccante (ovvero **sincrono** oppure **asincrono**)
3. La bufferizzazione: i messaggi scambiati risiedono in una coda temporanea

Operating Systems: IPC

Gestione delle Code di Messaggi: Comunicazione Simmetrica

- I processi devono conoscere **esplicitamente** il nome del destinatario o del mittente:
  - **send (P, msg)** – manda un messaggio al processo P
  - **receive (Q, msg)** – riceve un messaggio dal processo Q

- Proprietà di un canale di comunicazione:
  - Le connessioni sono stabilite automaticamente
  - Una connessione è associata esattamente a due processi (connessione binaria)
  - Fra ogni coppia di processi esiste esattamente una connessione
  - La connessione può essere unidirezionale, ma di norma è bidirezionale

Operating Systems: IPC

Gestione delle Code di Messaggi: Comunicazione Asimmetrica

- Solo il processo che invia il messaggio deve conoscere esplicitamente il nome del destinatario o del mittente:
  - **send (P, msg)** – manda un messaggio al processo P
  - **receive (id, msg)** – riceve un messaggio da un processo il cui identificatore è salvato in id

Operating Systems: IPC

Gestione delle Code di Messaggi: Comunicazione Indiretta

- I messaggi sono mandati e ricevuti attraverso una **mailbox** o **porte**
  - Ciascuna mailbox ha un identificatore univoco
  - I processi possono comunicare solo se hanno una mailbox condivisa

- Le primitive sono definite come
  - **send** (M, msg) – manda un messaggio alla mailbox M
  - **receive** (M, msg) – riceve un messaggio dalla mailbox M
- Viene stabilita una connessione fra due processi solo se entrambi hanno una mailbox condivisa
- Una connessione può essere associata a più di due processi (non binaria)
- Fra ogni coppia di processi comunicanti possono esserci più connessioni
- La connessione può essere unidirezionale o bidirezionale

Operating Systems: IPC

Gestione delle Code di Messaggi: Comunicazione Indiretta

- Una mailbox può appartenere ad un processo o al SO
- In ogni caso esiste sempre un processo che ha il diritto di proprietà

- Se appartiene al processo allora risiede nel suo spazio di indirizzi
  - Viene deallocata alla terminazione del processo
  - Il proprietario è l'unico che può ricevere
  - Tutti gli altri sono **utenti** che inviano messaggi

Operating Systems: IPC

Gestione delle Code di Messaggi: Comunicazione Indiretta

- Per le mailbox appartenenti al SO, il SO stesso mette a disposizione delle chiamate di sistema per:
  - Creare mailbox
  - Cancellare mailbox
  - Inviare e ricevere messaggi
  - Cedere o concedere il diritto di proprietà sulla mailbox
- Il processo creatore ha il **diritto di proprietà**
  - Inizialmente è l'unico che può ricevere
- La concessione del diritto di proprietà può portare ad avere più riceventi

Operating Systems: IPC

Gestione delle Code di Messaggi: Sincronizzazione

- Il passaggio di messaggi può essere bloccante (sincrono) oppure non bloccante (asincrono)

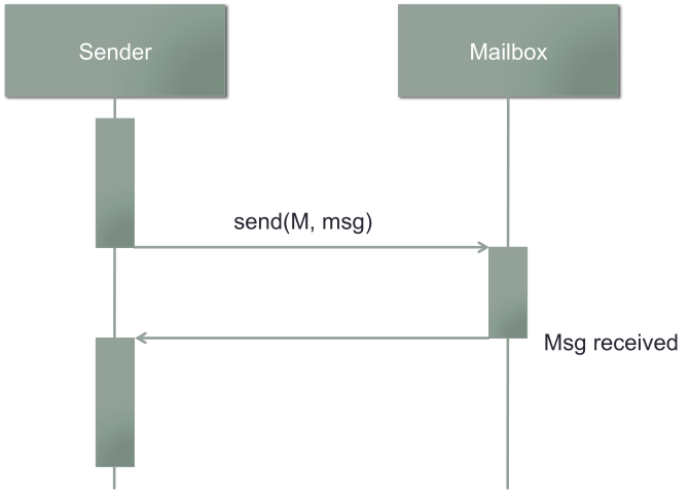
RENDEZVOUS

  - **Invio bloccante**: il processo che invia viene bloccato finché il messaggio viene ricevuto dal processo che riceve o dalla mailbox
  - **Ricezione bloccante**: il ricevente si blocca sin quando un messaggio non è disponibile

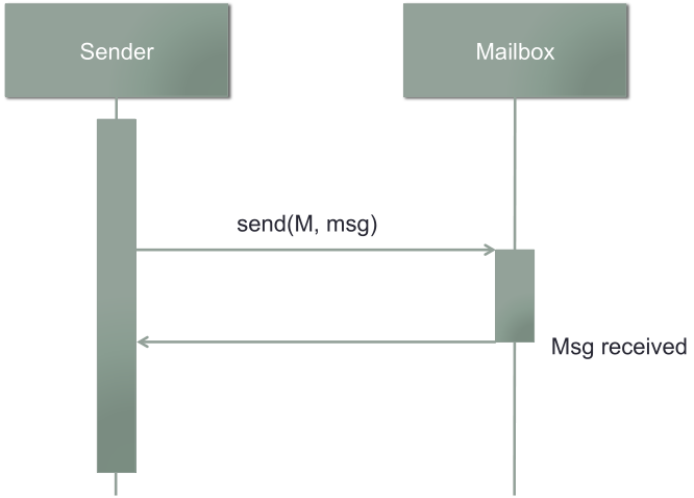
  - **Invio non bloccante**: il processo che invia manda il messaggio e riprende l'attività
  - **Ricezione non bloccante**: il ricevente acquisisce un messaggio o valido o nullo

Operating Systems: IPC

Gestione delle Code di Messaggi: Sincronizzazione - Invio



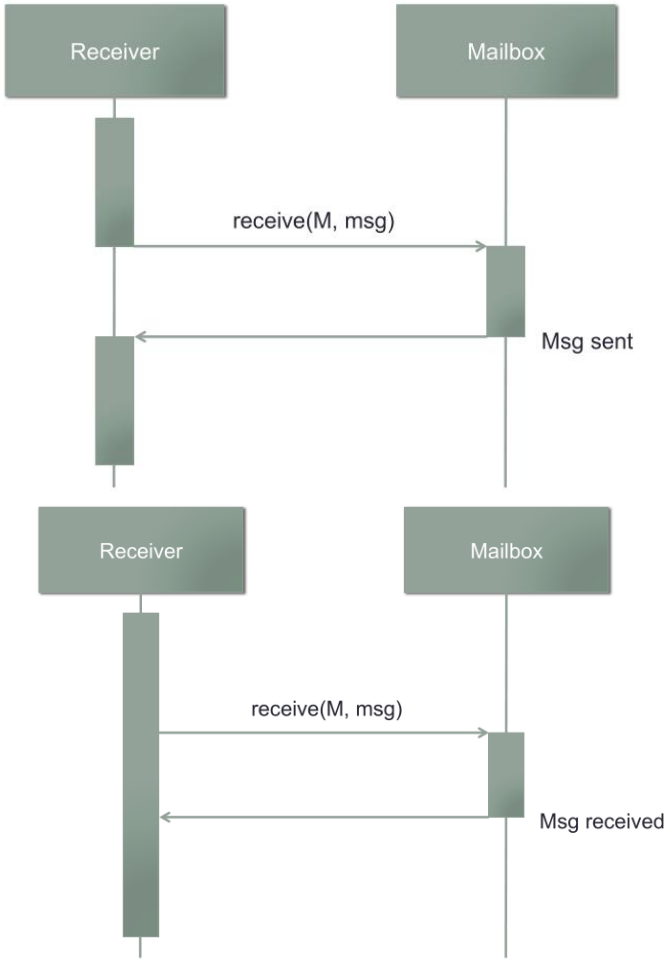
Invio Bloccante



Invio non-Bloccante

Operating Systems: IPC

Gestione delle Code di Messaggi: Sincronizzazione - Ricezione



Ricezione Bloccante

Ricezione non-Bloccante

Operating Systems: IPC

Gestione delle Code di Messaggi: Buffer

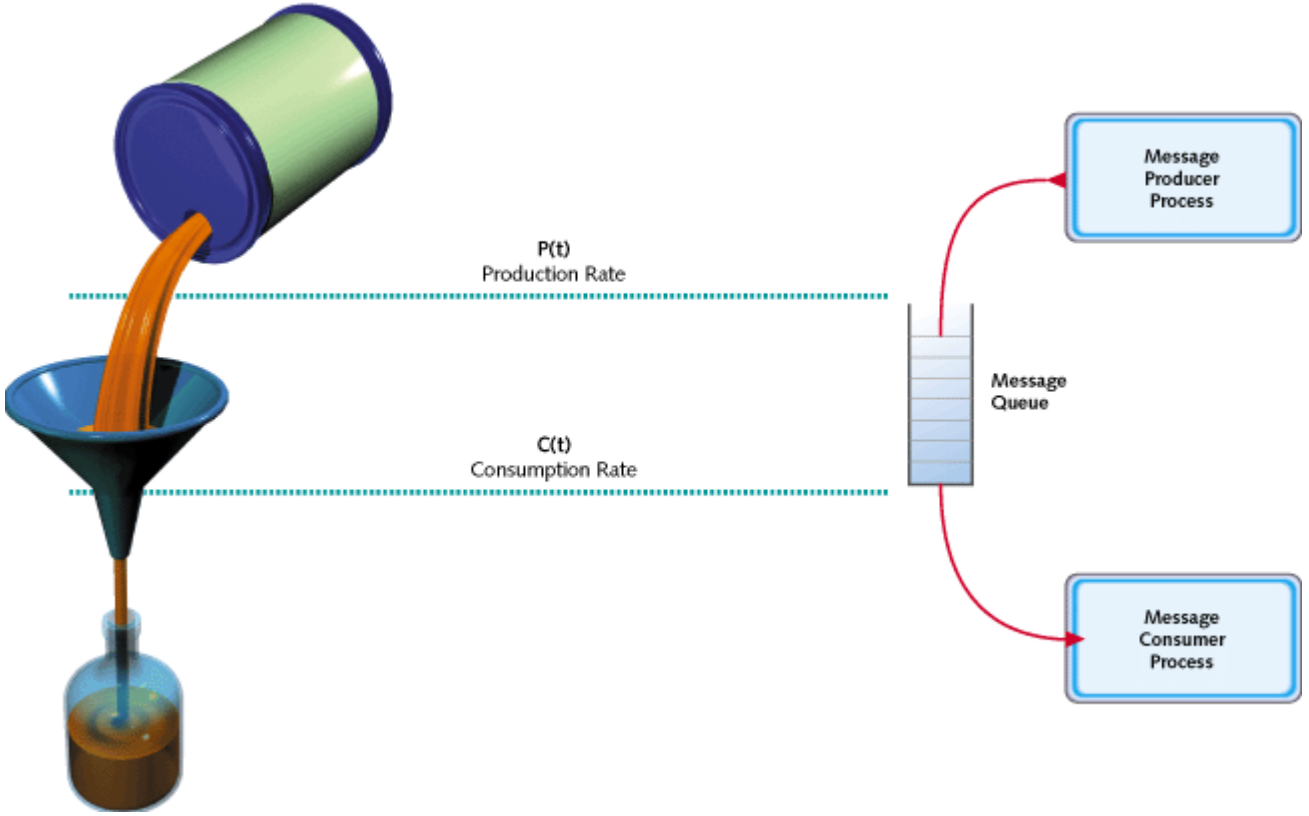
I messaggi scambiati tra processi risiedono in code temporanee (sia comunicazione diretta che indiretta)

- Tre modi per implementare le code:

1. **Capacità zero** (no Buffer) – Il mittente deve bloccarsi finché il destinatario riceve il messaggio (rendezvous)

2. **Capacità limitata** (Buffer) – lunghezza finita di n messaggi. Il mittente deve bloccarsi se la coda è piena

3. **Capacità illimitata** (Buffer) – lunghezza infinita
Il mittente non si blocca mai



Per l'ordinamento delle code dei messaggi nella mailbox sono usate le stesse politiche per la gesatione dei processi in attesa

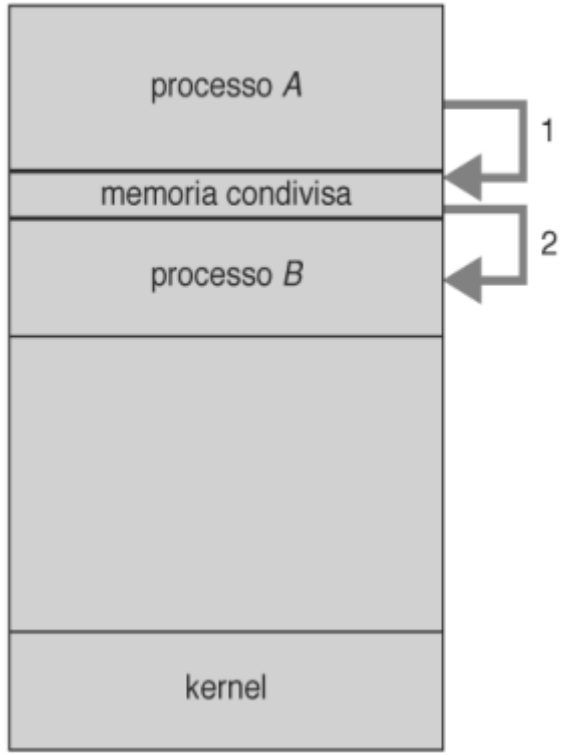
- First In, First Out (FIFO)
- Priorità
- Scadenza

Operating Systems: IPC

IPC: Implementazioni a Memoria Condivisa

Pipe

Memoria Condivisa



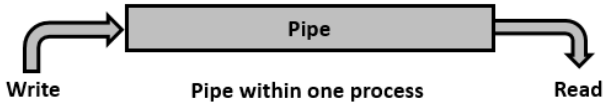
Operating Systems: IPC

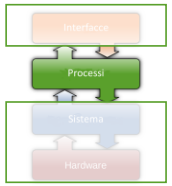
Gestione delle Pipe

Operating Systems: Obiettivi Funzioni Servizi

IPC: Pipe (datagram FIFO) in Unix

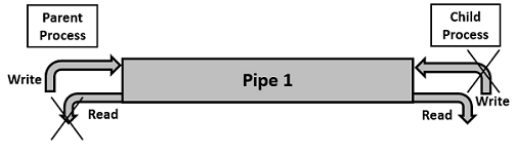
Pipe is one-way communication
Flusso (stream) Unidirezionale



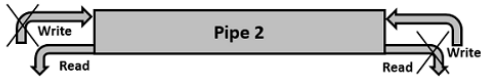



```
rishabh@rishabh:~/GFG$ ls -l | more
total 28
drwxrwxr-x 2 rishabh rishabh 4096 Jan 29 21:11 demo1
-rw-rw-r-- 1 rishabh rishabh   26 Jan 25 23:03 demo1.txt
drwxrwxr-x 2 rishabh rishabh 4096 Jan 29 21:11 demo2
-rw-rw-r-- 1 rishabh rishabh    0 Jan 25 23:04 demo2.txt
drwxrwxr-x 2 rishabh rishabh 4096 Jan 29 21:11 demo3
-rw-rw-r-- 1 rishabh rishabh    0 Jan 25 23:04 demo.txt
-rw-rw-r-- 1 rishabh rishabh  123 Jan 26 16:02 sample1.txt
-rw-rw-r-- 1 rishabh rishabh   44 Jan 26 15:52 sample2.txt
-rw-rw-r-- 1 rishabh rishabh    0 Jan 26 00:12 sample3.txt
-rw-rw-r-- 1 rishabh rishabh   26 Jan 25 23:03 sample.txt
```

Esempio di pipe tra comandi shell



Two-way Communication Using Pipes
(combinando 2 pipe)



Permettere a 2 processi di scambiarsi informazioni fra loro.

Una pipe è un tipo speciale di file condiviso, riconducibili al modello “memoria condivisa”

Una pipe è un tipo speciale di file condiviso. Due tipi

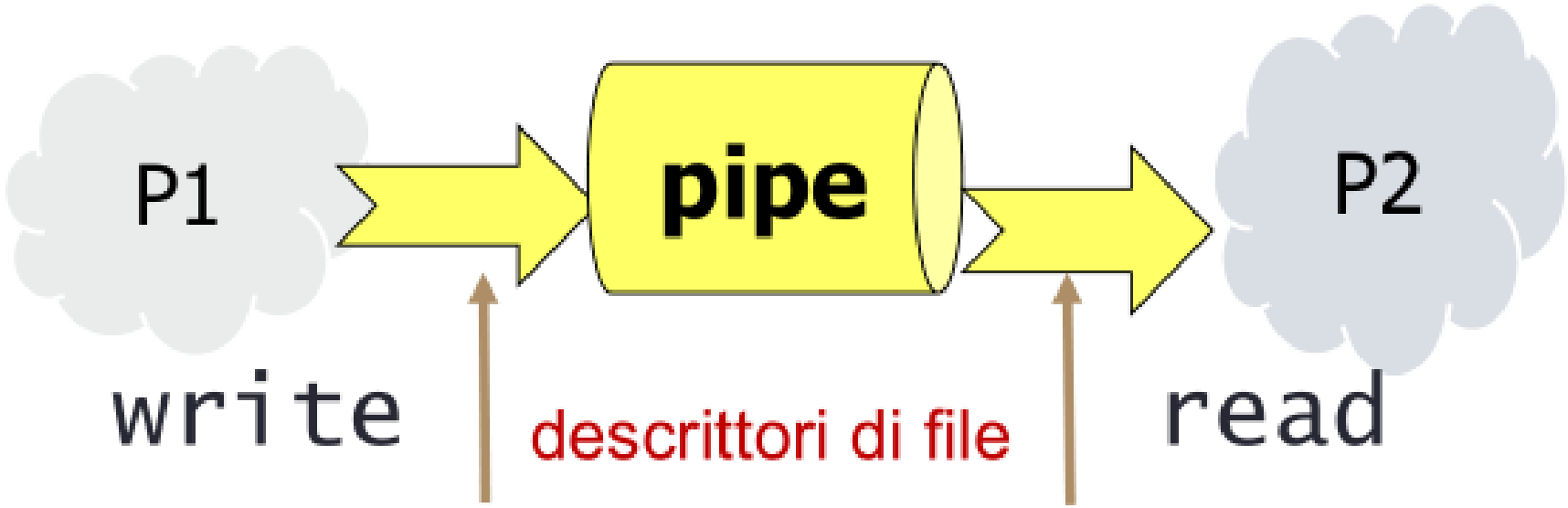
- Convenzionali (o anonime)
- Named pipe

Operating Systems: IPC

Gestione delle Pipe: Convenzionali 1/2

Comunicazione tra due processi secondo la modalità del «produttore e consumatore»

- Il produttore scrive da un'estremità (write-end)
- Il consumatore legge dall'altra estremità (read-end)
- Comunicazione uni-direzionale

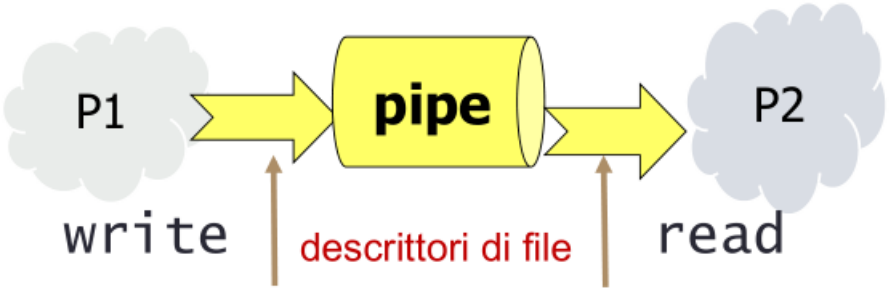


Operating Systems: IPC

Gestione delle Pipe: Convenzionali 2/2

Implementata come file in memoria centrale con scrittura solo in aggiunta e lettura unica solo sequenziale

- Deve esistere una relazione padre-figlio tra i due processi per condividere i descrittori di file
- In Unix può essere creata con la chiamata pipe(int fd[])
- fd è il descrittore del file, fd[0] scrittura, fd[1] lettura
- Lettura e scrittura tramite le chiamata read() e write()

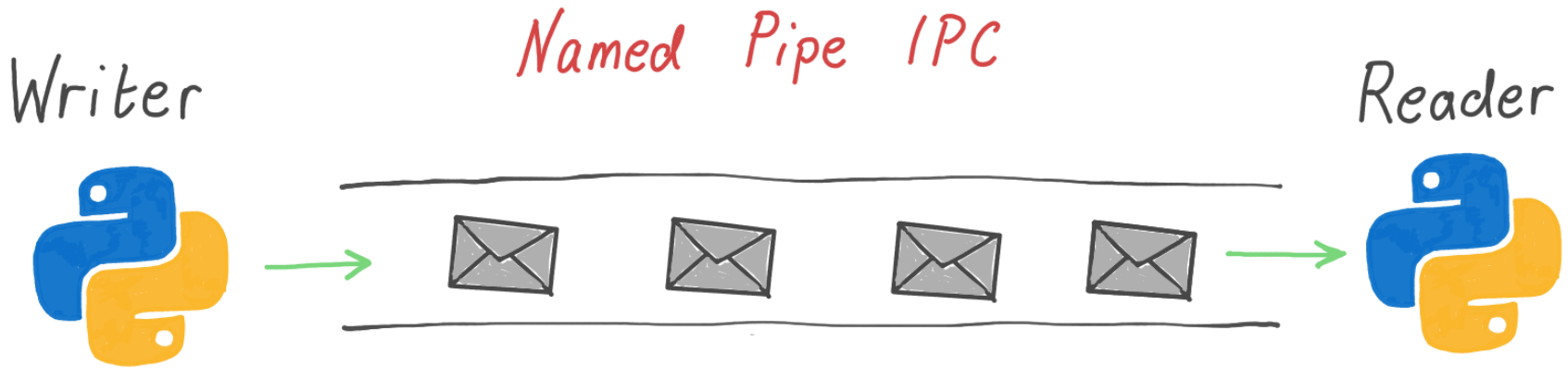


Operating Systems: IPC

Gestione delle Pipe: Named Pipe 1/2

Implementata come file referenziabile tramite File System

- Bidirezionali
- Relazione parentela padre-figlio non necessaria
- Comunicazione fra più di due processi
- Continuano ad esistere anche dopo che i processi comunicanti terminano



Operating Systems: IPC

Gestione delle Pipe: Named Pipe 2/2

Unix:

- Dette FIFO e create con mkfifo(), ma sono normali file del file system
- Half-duplex (i dati viaggiano in un'unica direzione alla volta)
- I programmi comunicanti devono risiedere nella stessa macchina (in alternativa, occorrono i socket)
- Dati byte-oriented

Win32:

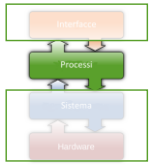
- Meccanismo più ricco: createNamedPipe() (nuova pipe) e ConnectNamedPipe() (pipe già esistente)
- Full-duplex (i dati viaggiano contemporaneamente in entrambe le direzioni)
- I programmi comunicanti possono risiedere in macchine diverse (ambiente client/server)
- Dati byte-oriented e message-oriented

Operating Systems: IPC

Gestione della Memoria Condivisa 1/2

Operating Systems: Obiettivi Funzioni Servizi

IPC: Shared Memory

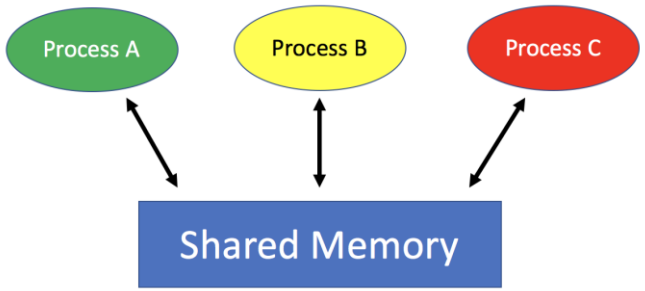


Shared Memory: memoria condivisa fra processi

SHMMAX: massima dimensione (in Byte) di un singolo segmento di memoria condivisa

SHMMNI: numero massimo di segmenti di memoria condivisa presenti sul Sistema

SHMALL: numero totale di pagine di memoria condivisa



Alloca porzioni di memoria condivisa fra processi.

I processi comunicanti colloquiano attraverso un'area di memoria condivisa

- Residente nello spazio degli indirizzi del processo che la alloca
- Gli altri processi annettono questa area al loro spazio degli indirizzi
- Tipicamente i processi non possono accedere alla memoria degli altri ➔ I processi comunicanti devono quindi stabilire un "accordo"

Il SO **non controlla**

- Tipo e allocazione dei dati
- Accesso concorrente alla memoria

SAPIENZA UNIVERSITÀ DI ROMA

Operating Systems: IPC

Gestione della Memoria Condivisa 2/2

Caratteristiche:

- Condivisione variabili globali
- Condivisione buffer di comunicazione (vedi producer-consumer)

Problemi:

- Identificazione dei processi comunicanti (comunicazione diretta)
- Consistenza degli accessi
- Lettura e scrittura sono incompatibili tra loro ➔ Richiede **sincronizzazione dei processi** per accesso in **mutua esclusione**

Operating Systems: IPC

Gestione della Memoria Condivisa: Produttore/Consumatore

Modello molto comune nei processi cooperanti:

- un processo produttore genera informazioni
- che sono utilizzate da un processo consumatore

Un oggetto temporaneo in memoria (buffer) può essere riempito dal produttore e svuotato dal consumatore

1. **buffer illimitato** (unbounded-buffer): non c'è un limite teorico alla dimensione del buffer
   - Il consumatore deve aspettare se il buffer è vuoto
   - Il produttore può sempre produrre

2. **buffer limitato** (bounded-buffer): la dimensione del buffer è fissata
   - Il consumatore deve aspettare se il buffer è vuoto
   - Il produttore deve aspettare se il buffet è pieno

Il buffer può essere:

- fornito dal SO tramite l'uso di funzionalità di comunicazione tra processi InterProcess Communication (IPC)
- oppure essere esplicitamente scritto dal programmatore dell'applicazione con l'uso di memoria condivisa