

# Argomenti

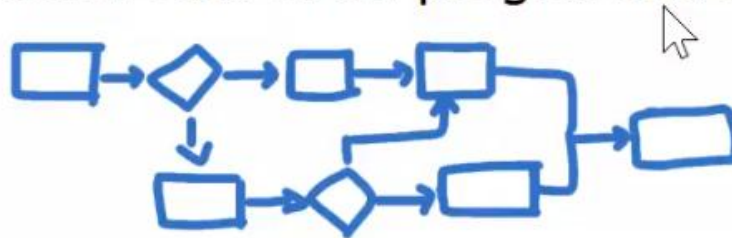
- PROCESSI:
  - Il modello di processo
  - Creazione del processo
  - Chiusura del processo
  - Gerarchie di processi
  - Stati di un processo
  - Realizzazione di processi
  - Modellazione della multiprogrammazione

# Argomenti

- THREAD:
  - Uso dei thread
  - Il modello a thread classico
  - Thread POSIX
  - Realizzazione dei threads
  - Attivazioni dello scheduler
  - Thread pop-up
- COMUNICAZIONI TRA PROCESSI:
  - Corse critiche
  - Regioni critiche
  - Mutua esclusione con busy waiting

# Processi

- Il concetto centrale in qualsiasi sistema operativo è il **processo**: l'astrazione di un programma in esecuzione.



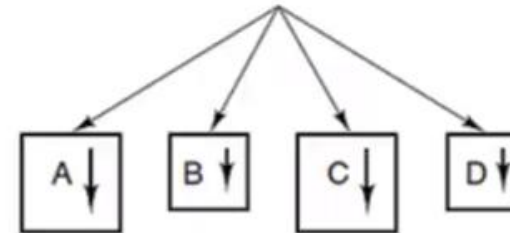
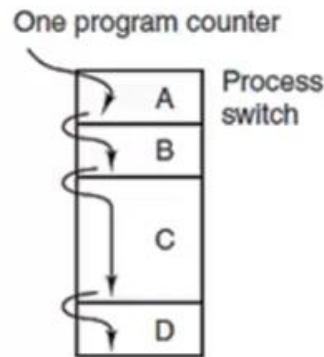
- Tutto il resto dipende da questo concetto.
- I processi sono una delle più vecchie e più importanti astrazioni dei sistemi operativi.
- Grazie a questo concetto gli utenti possono immaginare che ci siano più operazioni concorrenti anche se si dispone di una sola CPU (time sharing).

# Processi

- Tutti i computer spesso fanno molte cose allo stesso tempo (es. le pagine inviate da un web server).
- Per gestire queste attività è necessario un sistema di **multiprogrammazione con più processi**.
- In un sistema multiprogrammato la CPU passa da un processo all'altro rapidamente (10 ÷ 100 ms).
- In questo modo la CPU fornisce l'illusione di eseguire più processi contemporaneamente (**pseudoparallelismo**).

# Il modello di processo

- I **processi** sono il software che è in esecuzione o «gira» sul computer (compreso il SO).
- Il processo è un'istanza dell'esecuzione di un programma, inclusi i valori attuali del **Program Counter**, dello **Stack**, dei **registri** e delle **variabili**.
- Ogni processo quindi ha una propria CPU virtuale, in realtà la CPU commuta il controllo tra processi (multi-programmazione).



**Quattro PC logici**



# Creazione del processo

- I sistemi operativi devono essere in grado di creare processi.
- Un processo viene creato durante uno di questi eventi:
  - Inizializzazione del sistema.
  - Esecuzione di una chiamata di sistema dedicata.
  - Richiesta dell'utente di creare un nuovo processo.
  - Inizio di un job in modalità batch.
- Alla partenza del sistema operativo vengono molti processi, tra cui:
  - i **Processi attivi** che interagiscono con gli utenti e svolgono un compito per loro.
  - i **Processi in background** (invisibili), non associati ad un utente in particolare, che svolgono funzioni specifiche (i **demoni** si svegliano in corrispondenza di un evento e svolgono il lavoro per cui sono stati progettati).

## Creazione del processo

- In UNIX, l'unica chiamata di sistema per creare un processo è la **fork()**.
- **fork()** crea un clone esatto (figlio) del processo chiamante (padre) con la stessa immagine della memoria, le stesse stringhe di ambiente e gli stessi file aperti.
- Di solito, il processo figlio esegue **execve()** o una chiamata di sistema simile per cambiare la sua immagine in memoria ed eseguire un nuovo programma.
- In Windows viene utilizzato **CreateProcess()** per creare un nuovo processo e caricare il programma corretto.
- Sia in UNIX e Windows, il genitore e figlio hanno spazi di indirizzi **distinti**: una modifica ad una variabile dell'uno non influenza l'altro.
- I riferimenti a medesime risorse sono invece condivisi (es. file).

## Chiusura di un processo

- Una volta creato il processo gira fino a che non ha svolto il proprio compito (potrebbe anche essere sempre attivo).
- La chiusura di un processo si verifica a seguito di una delle seguenti condizioni:
  - uscita normale (volontaria): ***exit()*** in UNIX e ***ExitProcess()*** in Windows;
  - uscita per errore (volontaria): es. bug nel programma;
  - errore critico (involontario): es. run-time error;
  - ucciso da un altro processo (involontario): ***kill()*** in UNIX e ***TerminateProcess()*** in Win32.



# Gerarchie di processi

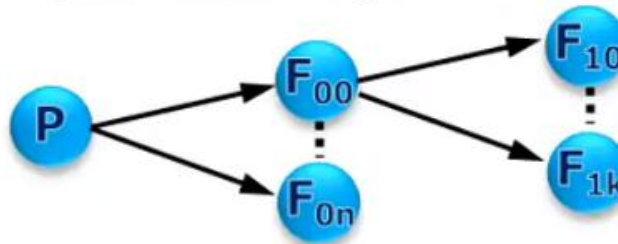
- Generalmente quando un processo (P) crea un altro processo (F), il processo figlio continua ad essere associato al genitore.



NB: ogni figlio ha un solo genitore.

## UNIX

- Il processo figlio può a sua volta creare altri processi, formando una gerarchia di processi.



- Un processo e tutti i suoi discendenti formano un gruppo di processi.

## Stati di un processo

- Ogni processo è un'entità autonoma con un proprio **Program Counter** e un proprio **stato interno**.
- I processi spesso hanno bisogno di interagire con altri processi.
- Un processo può generare un output che un altro processo utilizza come input.

# Stati di un processo


## *Esempio UNIX:*

**cat** capitolo1 capitolo2 | **grep** albero



- il comando cat concatena sequenzialmente i due file e restituisce un unico file in uscita;
- il comando grep seleziona tutte le righe del file in ingresso (il risultato di cat) che hanno la parola «albero».

## Blocco di un processo

- Un processo si blocca quando non è più nelle condizioni di svolgere il proprio compito: 
  - Potrebbe essere in attesa di un risultato che ancora non è disponibile (es. il grep di prima).
  - Il sistema operativo ha deciso di allocare la CPU ad un altro processo.
- Queste due condizioni sono completamente differenti:
  - Nel primo caso la sospensione è inerente il problema.
  - Nel secondo caso è un tecnicismo del sistema (non ci sono abbastanza CPU da assegnare ad ogni processo).



# Stati di un processo

- Un processo non concluso può trovarsi in uno dei seguenti stati:
  - **Esecuzione** (sta utilizzando la CPU)
  - **Pronto** (potrebbe essere eseguito perché tutte le risorse e i risultati che gli occorrono sono disponibili ma è in attesa che gli venga assegnata una CPU).
  - **Bloccato** (non può proseguire perché è bloccato da un evento esterno).



# Segnali

- Un processo **in esecuzione** può ricevere dei segnali di allarme che, analogamente agli interrupt nel caso hardware, ne bloccano l'esecuzione, causano il salvataggio dello stato e fanno sì che il processo esegua una procedura di gestione dei segnali speciali.
  - A termine della procedura il processo potrà riprendere l'esecuzione dal punto dove si era interrotto.
- Molti trap rilevati dall'hardware, come l'esecuzione di una istruzione illegale o di un indirizzo errato, sono convertiti in segnali per il processo responsabile.
- I segnali possono essere attivati anche in modo temporizzato, come nel caso dell'invio di un messaggio per eseguire un controllo sulla ricezione entro un certo timeout.

## Realizzazione dei processi

- Per implementare il modello di processo, il sistema operativo mantiene una **tabella dei processi** che ha una voce (o riga) per ogni processo.
- La voce è detta **process control block** e mantiene lo stato del processo:
  - Il Program Counter.
  - Lo Stack Pointer.
  - l'allocazione della memoria.
  - Lo stato dei file aperti.
  - Le informazioni relative alla gestione e allo scheduling.
- Nella voce è memorizzato tutto ciò che serve salvare nello stato bloccato e pronto affinché sia possibile riavviare il processo come se non si fosse mai fermato.



tabella dei processi

PC	SP	mem	file	sched

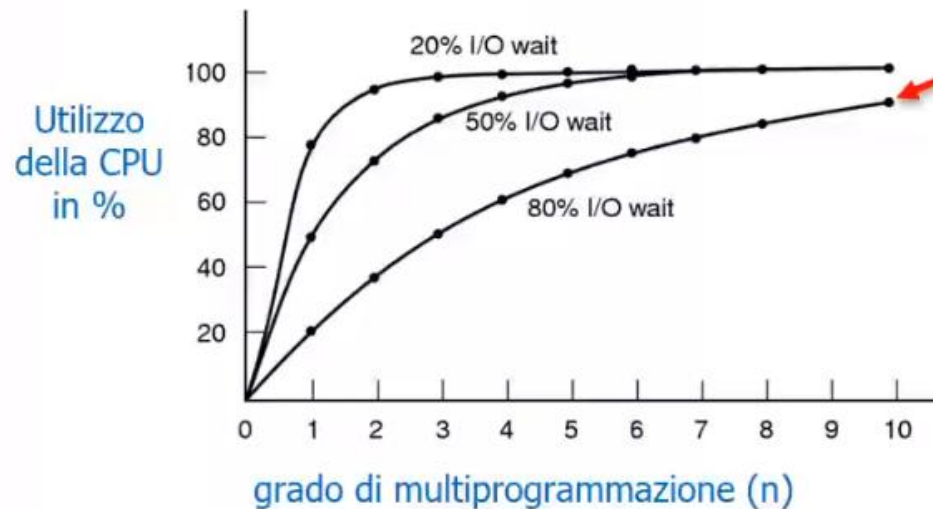


## Un modello per la multiprogrammazione

- L'utilizzo della multiprogrammazione migliora l'utilizzo della CPU.
- Se un processo in media esegue calcoli solo per il 20% del tempo in cui risiede in memoria, con 5 processi contemporaneamente in memoria la CPU dovrebbe essere occupata tutto il tempo (100%).
- In realtà i processi attendono anche l'I/O.
- Da un punto di vista probabilistico: se un processo spende una frazione  $p$  del suo tempo in attesa dell'I/O, con  $n$  processi in memoria la probabilità che stiano tutti aspettando l'I/O è  $p^n$ .
- L'utilizzo della CPU è la probabilità complementare:  $1-p^n$ .



# Un modello per la multiprogrammazione



- se i processi attendono per l'80% del loro tempo in attesa dell'I/O, servono almeno 10 processi per portare la CPU ad un utilizzo del 90%.
  - Questo modello semplificato permette di effettuare previsioni sull'utilizzo della CPU.
- Un computer ha 512MB di memoria, se il SO occupa 128MB e un processo 128MB, può contenere in memoria fino a 3 processi.
  - Se l'attesa media dell'I/O è dell'80%, l'utilizzo della CPU è pari a  $1 - 0,8^3 = 49\%$
  - Aggiungendo 512MB miglioriamo del  $79\% = 1 - 0,8^7$

# THREAD

- I **thread** sono dei **processi leggeri** o **miniprocessi**, a differenza dei processi sono tra loro fortemente correlati poiché condividono spazio degli indirizzi e dati.
- Hanno delle caratteristiche vincenti:
  - Il modello di programmazione diventa più semplice (l'applicazione si può decomporre in thread sequenziali che possono essere eseguiti in modalità quasi-parallela).
  - Essendo più «leggeri» si possono creare o distruggere velocemente.
  - Hanno un migliore utilizzo della CPU quando le attività sono I/O bound.
- Un ambiente che consente a più thread di girare nello stesso processo è chiamato **multithreading**.
- Le CPU moderne che supportano il multithreading riescono a passare da un thread all'altro nell'ordine dei nanosecondi!

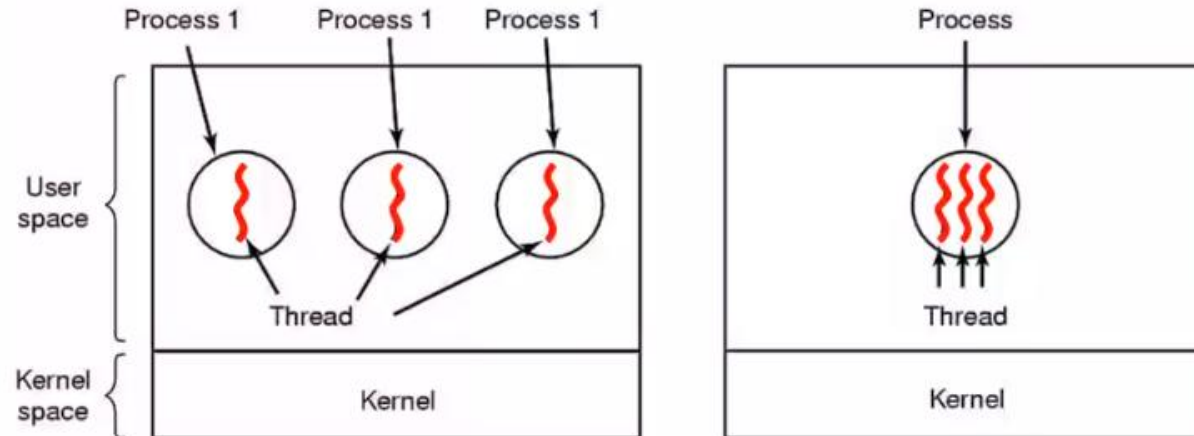
## Il modello a thread classico

- Il **modello di processo** si basa su due concetti indipendenti:
  - 1) Raggruppamento delle risorse.
  - 2) Esecuzione.
- Quindi un processo può essere visto come un modo per raggruppare insieme risorse:
  - uno spazio di indirizzamento (programma e dati).
  - file aperti, i processi figli, allarmi in sospenso, gestori di segnale, informazioni sugli account, ...
- I thread invece sono entità schedate per l'esecuzione.
- Più thread possono essere eseguiti nello stesso ambiente di processo.



## Il modello a thread classico

- Più processi che girano in parallelo un computer non è la stessa cosa di più thread in parallelo:
  - Nel primo caso, i processi condividono solo la memoria fisica, i dischi, le stampanti e altre risorse, nel secondo i thread condividono anche lo spazio di indirizzamento, lo stack, i registri e lo stato.



- I thread in un processo non sono tanto indipendenti quanto i processi concorrenti.

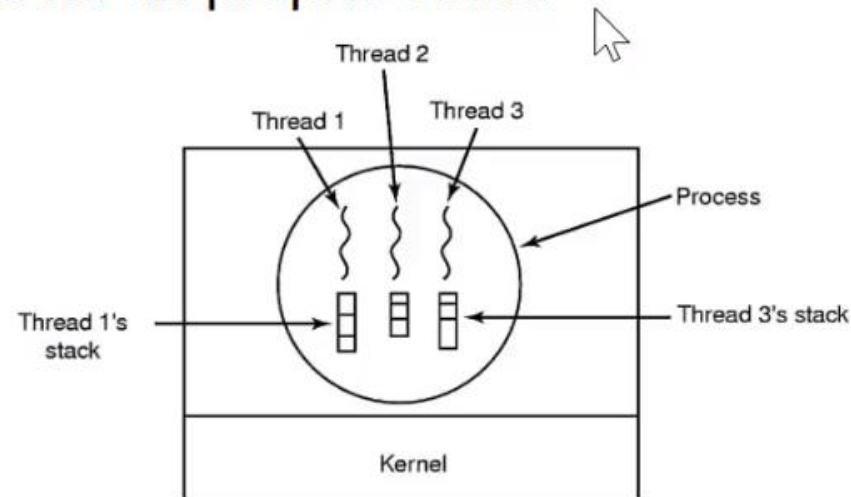


## Confronto tra Thread e Processi

- Poiché tutti i thread hanno lo stesso spazio di indirizzamento (le stesse variabili globali), possono leggere/scrivere/cancellare lo Stack di un altro thread!
- Tra i thread, **non esiste protezione** perché:
  - 1) È impossibile realizzarla.
  - 2) Non dovrebbe essere necessaria.
- Mentre i processi possono essere di proprietà di diversi utenti e **competere** per ottenere risorse comuni, ogni processo (di proprietà di un singolo utente) può creare più thread che dovrebbero **cooperare** non entrando mai in conflitto tra loro.
- L'organizzazione a processi dovrebbe essere utilizzata quando le attività dei processi non sono tra loro correlate. Mentre un modello a thread si dovrebbe utilizzare quando i compiti di ciascuno sono parte dello stesso lavoro e, quindi, è necessaria una stretta collaborazione.

# Il modello a thread classico

- Ogni thread ha un proprio stack.



- In un ambiente multithreading i processi partono con un singolo thread, quest'ultimo ha la facoltà di crearne altri thread attraverso la procedura ***thread\_create()***.
- All'atto della creazione non è necessario specificare nulla poiché il nuovo thread girerà nel medesimo spazio di indirizzamento del thread che l'ha creato.



## Alcune chiamate di sistema interessanti

- I thread possono avere relazioni gerarchiche (padre-figlio) oppure essere allo stesso livello (più comune).
- Il thread che ne crea un altro riceve in uscita l'identificatore del thread creato.
- Quando un thread ha terminato il suo lavoro esso informare il mondo esterno attraverso la chiamata ***thread\_exit()***.
- In alcuni sistemi un thread può aspettare il termine di un altro thread attraverso la procedura ***thread\_join()***.
- Una chiamata molto comune è la ***thread\_yield()***, che permette di rilasciare la CPU ad un altro thread. Questa chiamata è importante perché permette di realizzare la multiprogrammazione **senza che ci sia interruzione**, così come accade per i processi.

# Thread POSIX

- Per garantire portabilità dei programmi che utilizzano i thread, IEEE ha definito uno standard per i thread: **1003.1c**.
- I thread definiti nello standard sono chiamati **Pthread** e sono supportati dai principali sistemi UNIX.
- Lo standard definisce più di 60 chiamate di sistema specifiche per i Pthread.
- Ciscun **Pthread** ha:
  - un **identificatore**;
  - un insieme di registri compreso il Program Counter;
  - un insieme di attributi (dimensione dello stack, parametric per lo scheduling,...) memorizzati in una struttura.



## Alcune chiamate di sistema POSIX



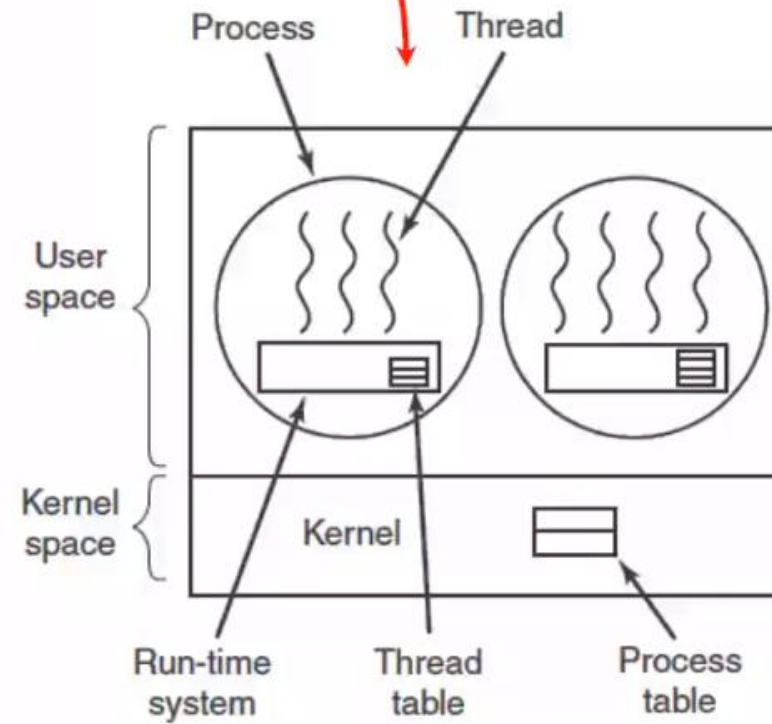
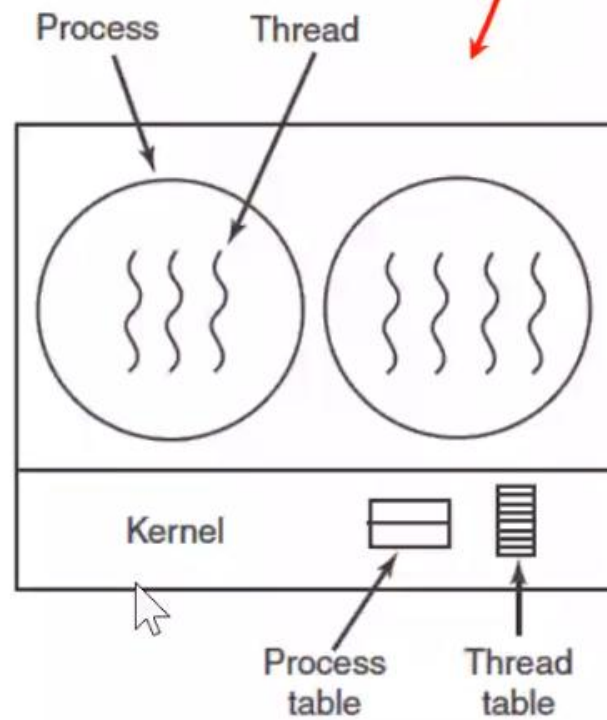
Thread call	Descrizione
Pthread_create	Crea un nuovo thread
Pthread_exit	Termina il thread che la invoca
Pthread_join	Attende che un thread specifico termini
Pthread_yield	Rilascia la CPU perché venga eseguito un altro thread
Pthread_attr_init	Crea ed inizializza la struttura con gli attributi del thread
Pthread_attr_destroy	Rimuove la struttura degli attributi di un thread

# Un programma di esempio che utilizza i thread

```
#include <pthread.h>
#include <Stdio.h>
#include <Stdlib.h>
#define NUMERO_DI_THREAD 5
void *ciao_mondo(void *tid) { /* stampa l'ID del thread e termina */
    printf("Ciao mondo. Saluti dal thread %d0", tid);
    pthread_exit(NULL); /* termina il thread */
}
int main(int argc, char *argv[]) {
    /* Il programma principale crea 5 thread e poi termina l'esecuzione */
    pthread_t thread[NUMERO_DI_THREAD];
    int stato, i;
    for(i=0; i < NUMERO_DI_THREAD; i++) {
        printf("Il programma principale crea il thread n.%d0", i);
        stato = pthread_create(&thread[i], NULL, (void *)&ciao_mondo, (void *)i);
        if (stato != 0) {
            printf("Pthread_create risponde con codice di errore %d0", stato);
            exit( -1 ); /* esce con errore */
        }
    }
    exit(NULL);
}
```

# Realizzazione dei thread

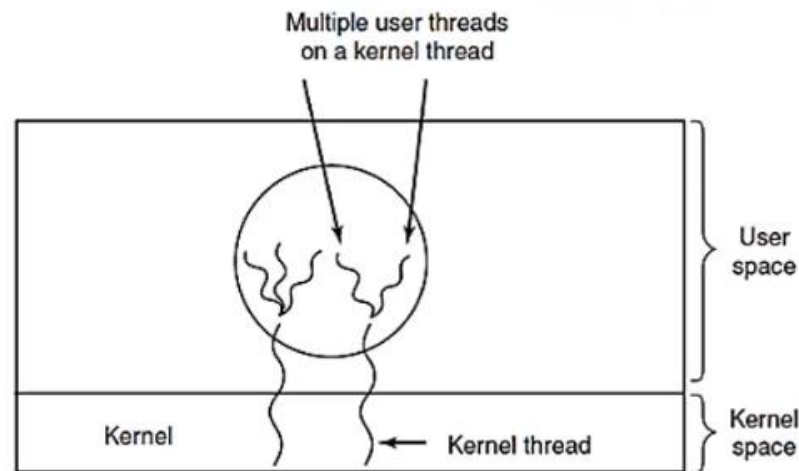
- I thread possono essere gestiti:
  - nello spazio **utente**.
  - nel **kernel**.
  - in entrambi.





## Realizzazioni ibride

- Sono stati studiati vari modi per combinare i vantaggi delle due soluzioni.
- Una tecnica è di utilizzare i thread a livello kernel e fare in modo che ciascuno di essi «utilizzi» a turno un thread di livello utente, come mostrato in figura:



- Il kernel è in grado di schedulare i soli thread a livello kernel.

## Attivazioni dello scheduler

- In certe situazioni i thread nel kernel sono migliori di quelli a livello di utente, ma sono anche i più lenti.
- Le attivazioni dello scheduler simulano la funzionalità dei thread nel kernel, ma con migliori prestazioni e maggiore flessibilità.
- L'efficienza è ottenuta evitando inutili transizioni tra lo spazio utente e quello del kernel quando un processo si blocca.
- Il kernel assegna un certo numero di processori virtuali ad ogni processo e fa in modo che il sistema di run-time assegni i thread ai processori.
- Quando il kernel riconosce che un thread è bloccato (ha effettuato una chiamata bloccante o ha causato un page-fault), lo comunica al sistema di run-time (**upcall**):
  - a questo punto il sistema di run-time può rischedulare i suoi thread.
- Le attivazioni dello scheduler dipendono dalle **upcall** che violano il concetto che un livello inferiore possa utilizzare i servizi di uno superiore tipico delle architetture a strati.

## Thread Pop-up

- I thread sono utili negli ambienti distribuiti.
- Un esempio è la gestione dei messaggi in ingresso (richieste di servizio):
  - Quando arriva un nuovo messaggio il sistema crea un nuovo thread (chiamato **pop-up**) per gestire il messaggio stesso.
- Perché un thread pop-up nasce nel momento in cui arriva il messaggio non hanno alcuna storia da ripristinare (PC, registri, stack,...)
- Ognuno parte da zero ed è identico a tutti gli altri, quindi la creazione è velocissima: quello che serve quando arriva un messaggio.
- Il thread pop-up può essere eseguito sia nello spazio utente che nel kernel, di solito si preferisce questa seconda strada per la semplicità di accesso alle tabelle del kernel e ai dispositivi di I/O anche se è più «pericolosa».

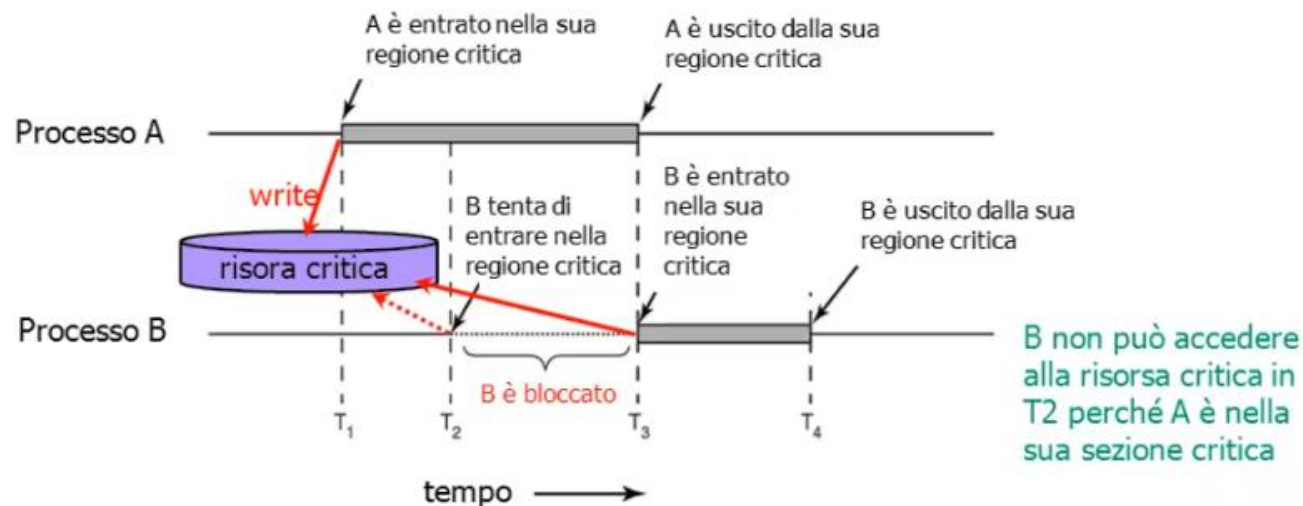


# Comunicazioni tra processi

- I processi hanno bisogno di comunicare con altri processi (**Interprocess Comunucation** o **IPC**).
- I processi dovrebbe comunicare in un modo ben strutturato senza utilizzare gli interrupt.
- Ci sono tre questioni da analizzare:
  - 1) Come un processo può passare informazioni ad un altro processo.
  - 2) Come gestire le situazioni in cui due o più processi competono per accaparrarsi una risorsa comune evitando che si sovrappongano.
  - 3) Definire regole di sincronizzazione tra processi dipendenti (es. produttore/consumatore).

## Regioni critiche

- Possiamo considerare il problema della condizione di competizione da un altro punto di vista:
  - Un processo A fa calcoli e poi deve accedere alla memoria condivisa (**sezione** o **regione critica**). Arriva B ma è bloccato fino a che A non esce dalla regione critica.
- Se si esclude il fatto che due processi siano nella loro sezione critica allo stesso tempo siamo certi che non competeranno per la risorsa.



## Regioni critiche

- In un ambiente concorrente per evitare corse critiche si possono definire quattro condizioni:
  - 1) Due processi non possono rimanere all'interno delle loro regioni critiche allo stesso tempo.
  - 2) Nessuna ipotesi può essere fatta sulla velocità o sul numero delle CPU.
  - 3) Nessun processo in esecuzione al di fuori della sua regione critica può bloccare altri processi.
  - 4) Nessun processo deve restare in attesa infinita per poter entrare nella sua regione critica.



## Mutua esclusione con busy waiting

- La mutua esclusione è possibile utilizzando il busy waiting: mentre un processo è nella sua regione critica gli altri devono attendere.
- Esistono varie soluzioni:
  - **disabilitare gli interrupt;**
  - **utilizzare delle variabili per il lock;**
  - **alternanza stretta;**
  - **la soluzione di Peterson;**
  - **l'istruzione TSL.**



## Disabilitare gli interrupt

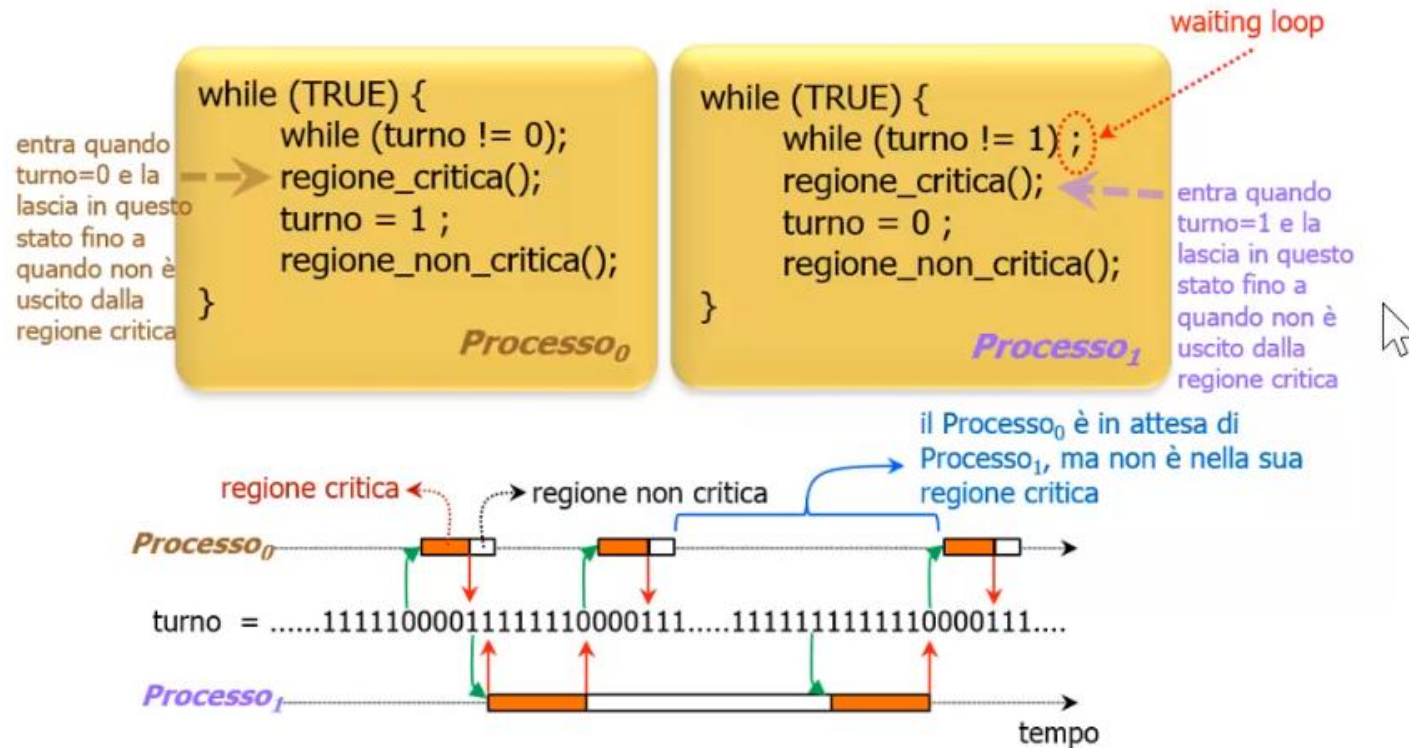
- In un sistema a singolo processore, la soluzione più semplice è quella di disabilitare tutti gli interrupt solo dopo essere entrato nella regione critica e riabilitarli prima di lasciarla.
- Con gli interrupt spenti la CPU non può passare da un processo all'altro.
- Ogni processo può aggiornare le risorse condivise senza timore che qualsiasi altro processo possa interferire.
- Dare ai processi utente il potere di abilitare le interruzioni non è una buona pratica!
- Di solito è il kernel che decide quando disabilitarli se sta eseguendo istruzioni critiche, che un processo utente si intrometta non è buono e può portare a degli errori.
- In un multicore disabilitare gli interrupt di una CPU non impedisce alle altre CPU di averli attivi e quindi di interferire.
- Non è una buona soluzione e sono necessari sistemi più sofisticati...

## Variabili di lock

- Si tratta di una soluzione software semplice: si utilizza una variabile condivisa (**lock**) per bloccare la risorsa (**1=occupata** e **0=libera**).
- Quando un processo vuole entrare nella sua regione critica, verifica prima il lock.
  - Se  $\text{lock}=0$ , il processo imposta lock a 1 e entra nella regione critica.
  - Se  $\text{lock}=1$ , il processo attende solo fino a quando non diventa 0.
- **Problema**
  - si supponga che un processo legge  $\text{lock}=0$  ma prima di riuscire ad impostare  $\text{lock}=1$ , altro processo che aveva visto libera la risorsa imposta  $\text{lock}=1$ . A questo punto sono entrambi nella sezione critica.
  - Non vi è alcuna garanzia che una risorsa ritenuta «libera» perché ha  $\text{lock}=0$  non diventi occupata ancora prima che il processo le assegni il valore di «occupata».



# Alternanza stretta



## La soluzione di Peterson


- Nel 1981, Peterson scoprì un modo per realizzare la mutua esclusione: prima di entrare nella regione critica il processo chiama una funzione con il suo numero di processo come parametro.
- Quando ha terminato ed è fuori dalla sua regione critica attiva un'altra procedura.

```
#define FALSE 0
#define TRUE 1
#define N 2 /* numero dei processi */
int turno; /* Turno stabilisce a chi tocca */
int interessato[N]; /* inizializzato con tutti false (0) */
void entra_in_regione (int processo) { /* il processo vale 0 o 1 */
    int altro; /*
    altro = 1 - processo; /* il processo complementare */
    interessato[processo] = TRUE; /* il processo è interessato! */
    turno = processo; /* è il turno del processo */
    while (turno == processo && interessato[altro] == TRUE); /*
} /*
void lascia_la_regione (int processo){ /* il processo che lascia la
    interessato[processo] = FALSE; /* regione non è più interessato */
}
```

## La soluzione di Peterson

- Anche se i due processi invocassero quasi contemporaneamente la procedura, solo l'ultimo scriverebbe la variabile turno.
- In questo modo solo uno dei due avrebbe la possibilità di non rimanere bloccato dal ciclo while.

```
void entra_in_regione(int processo)
{
    int altro;
    altro = 1 - processo;
    interessato[processo] = TRUE;
    turno = processo;
    while ( turno == processo &&
           interessato[altro] == TRUE);
}
```





## L'istruzione TSL

- Alcune CPU possiedono nativamente l'istruzione **TSL** (**Test & Set Lock**).
- L'istruzione assembly legge il contenuto della variabile lock nel registro RX e poi salva un valore non zero all'indirizzo di memoria della variabile in modo atomico.
- Quando un processo accede alla memoria, nessun altro può accedervi fino a che l'istruzione **TSL** non è terminata.
- La CPU che esegue l'istruzione **TSL** chiude il bus indirizzi e vieta ad altre CPU l'accesso in memoria fino a che il dato non è scritto.

## L'istruzione TSL

- Per realizzare la mutua esclusione si può usare il seguente codice:

**entra\_in\_regione:**

```
TSL  RX, LOCK
CMP   RX, #0
JNE   entra_in_regione
RET
```

**lascia\_la\_regione:**

```
MOVE LOCK, #0
RET
```

- Prima di entrare nella regione critica un processo chiama **entra\_in\_regione** e rimane in busy waiting fino a che il lock non è libero:
  - all'uscita invoca **lascia\_la\_regione**.
- Un'alternativa all'istruzione TSL è la **XCHG** (eXCHanGe), che scambia il contenuto di due posizioni **atomicamente** (es. tra registro e memoria).