4° lezione S.O.

Due Date	@March 13, 2025
Multi-select	Processi Thread
Status	Done

I processi

Modello dei Processi

Un **processo** è un'entità in esecuzione che rappresenta l'istanza di un programma. La creazione di un processo avviene sempre su richiesta di un **processo pre-esistente**, dando origine a una gerarchia di processi.

Uno dei meccanismi fondamentali per la creazione di nuovi processi è la **chiamata di sistema** `fork()`, tipica dei sistemi Unix/Linux.

- La `fork()` genera un nuovo processo, chiamato **processo figlio**, che è un **duplicato** del processo padre.
- Il processo figlio eredita lo **spazio di indirizzamento** del padre, inclusi codice e dati, ma ha un proprio identificatore di processo (**PID**).
- Dopo la `fork()`, sia il processo padre che il processo figlio continuano l'esecuzione, distinguendosi tramite il valore di ritorno della `fork()`.

Si ha anche la funzione `exec()` invocata da un processo ben ben preciso azzera il contenuto del processo che ha invocato la chiamata a que funzione e predispone il contenitore pre-esistente per un nuovo programma. Questo e i parametri passati vengono fornito come parametri alle chiamata di sistema.

 Questi due passaggi vengono unificati per Windows nella funzione `createProcess()` fornendo una serie di parametri permette sempre e comunque di creare un nuovo processo specificando anche cosa dovrà fare.

Creazione e terminazione dei processi

I sistemi UNIX prevedono il tracciamento della “parentela” dei processi, ciò permette di eseguire operazioni a cascata sui processi imparentati (come la chiusura)

Nella terminazione è il codice stesso che comunica la volontà del processo di terminare e in particolare espresso con il comando exit.

Codici di terminazione di un processo

Viene comunicato al sistema operativo la volontà di terminare un singolo processo attraverso una chiamata di sistema e questo risponderà con un valore di ritorno(exit status)

- `exit()` UNIX
- `ExitProcess()` Win

Valori di ritorno nei processi

Il meccanismo di creazione dei processi, come la chiamata di sistema `fork()`, avviene **su richiesta esplicita del programma** e in modo **sincrono**, ovvero il codice stesso decide volontariamente quando generare un nuovo processo.

Una volta terminato, un processo restituisce un **exit status**, che indica il risultato dell’esecuzione:

- **`exit status: 0`** → Il processo è terminato con successo.
- **`exit status: > 0`** → Si è verificato un errore durante l'esecuzione. Il valore specifico può indicare il tipo di errore.

Nel sistema operativo Unix/Linux, i processi figli comunicano il loro stato di uscita al processo padre attraverso la chiamata di sistema `wait()`, che consente al padre di recuperare il valore di ritorno e gestire eventuali errori[...].

Errori Critici ed Eventi Involontari

Questi errori si verificano quando un processo tenta di eseguire un'operazione **non consentita o anomala**, causando una **terminazione involontaria**. Si tratta di meccanismi automatici attivati dal sistema operativo e dall’hardware per prevenire comportamenti errati o dannosi.

Esempi di errori critici:

1. **Accesso a memoria non valida** (`SIGSEGV` – Segmentation Fault)

- Si verifica quando un processo tenta di leggere o scrivere in un’area di memoria non autorizzata.
- Esempio: dereferenziazione di un puntatore nullo (`NULL pointer dereference`).

2. **Divisione per zero** (`SIGFPE`)
   - Genera un'eccezione aritmetica che porta alla terminazione del processo.
3. **Violazioni dei permessi** (`SIGILL`, `SIGBUS`)
   - Tentativo di accesso a file o risorse senza le autorizzazioni necessarie.

Questi errori vengono gestiti **dall'hardware**, che segnala l’anomalia al sistema operativo. Quest'ultimo risponde interrompendo l'esecuzione del processo.

Errori Critici

Negli **errori critici**, **non esistono strategie di recupero**, e il processo viene **terminato immediatamente e in modo irreversibile** dal sistema operativo. Non è possibile gestire internamente questi errori nel codice dell’applicazione, poiché derivano da violazioni fondamentali delle regole del sistema.

Processo terminato da un altro processo

Il mittente di una richiesta di terminazione è solitamente un **processo**, che si riferisce a un altro processo.

Questo tipo di richiesta avviene generalmente **all'interno dello stesso utente**, ad esempio quando un utente chiude un'applicazione in esecuzione.

Nei sistemi multiutente, un utente **non può terminare i processi di altri utenti**, a meno che non sia **un amministratore (root)**, che ha i privilegi per gestire tutti i processi del sistema(super partes).

Inoltre, esistono **diverse tipologie di richieste di terminazione**, che possono avere connotazioni differenti a seconda del contesto e del tipo di segnale inviato al processo come cortesi e non cortesi. Si sta chiedendo al processo di interrompersi, non è detto che questo si interrompa veramente, magari perché se seguendo operazioni delicate

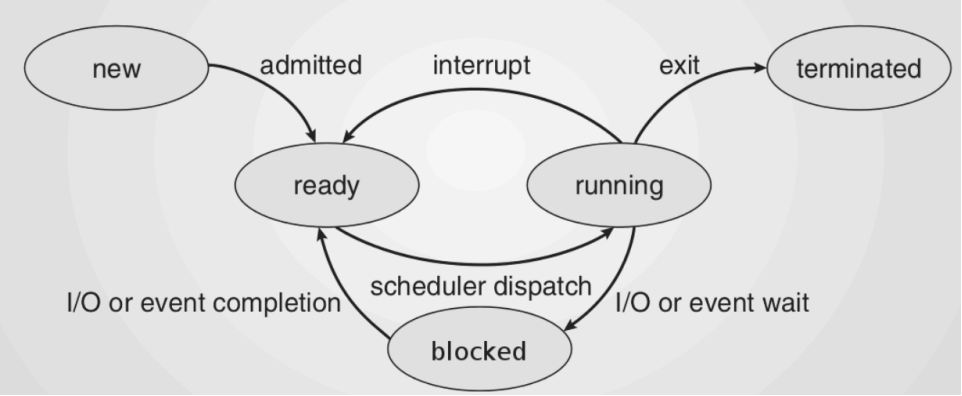
Queste richieste vengono generate in modo **asincrono**, similmente a un **interrupt**. Una volta ricevuta la richiesta, il sistema esegue una procedura di

gestione e, al termine, verifica se il processo debba effettivamente essere chiuso.

Questo meccanismo è particolarmente utile nei programmi con compiti critici, come un **database**, dove una terminazione improvvisa potrebbe causare danni, compromettendo l'integrità dei dati. Per evitare che il database si interrompa bruscamente e si corrompa, il sistema può gestire la richiesta in modo controllato, eseguendo prima eventuali operazioni di pulizia.

Esista la possibilità per la terminazione brutale di una applicazione. Il pro è che il processo non si può rifiutare (es. come quelle app che “non rispondono” e si usa il gestione attività per chiuderle). Il comando `kill` per UNIX e `TerminateProcess` per windows permette di chiedere gentilmente/brutalmente, in base al tipo di richiesta, di uccidere un processo.

Stato di un processo



Ogni processo ha un proprio **arco di vita**, che inizia con la **creazione** e termina con la **terminazione**.

Abbiamo analizzato lo **stato di un processo**, ma non il suo **stato interno nella PCB (Process Control Block)**. Ogni processo ha infatti uno **stato interno** che descrive le varie fasi della sua esecuzione.

Quali stati vi sono?

Gli stati principali di un processo sono **tre**, ai quali si aggiungono la fase di **creazione** e quella di **terminazione**. Gli stati addizionali includono la fase in cui il processo **finalizza le strutture dati** prima della sua effettiva creazione, ovvero quando non è ancora pronto per l'esecuzione.

1. **Pronto(new)**

2. **In esecuzione(running)**
3. **Bloccato(blocked)**
4. **Nuovo(new)**
5. **Terminato(terminated)**

La transizione **admitted** collega lo stato **new** con lo stato **ready**. Questo significa che, dopo la creazione di un processo, il sistema operativo verifica che siano disponibili le risorse necessarie e lo sposta nello stato **ready**, rendendolo idoneo per l'esecuzione quando la CPU sarà disponibile.

Si inizia dallo stato **ready**, quando un processo è pronto ad utilizzare la CPU. Nel caso in cui ci sia una sola CPU ma una **moltitudine di processi**, più processi possono trovarsi contemporaneamente nello stato **ready**. Per gestire questa situazione, i processi vengono **memorizzati in una struttura dati** specifica, che è la **coda dei processi pronti** (ready queue).

La struttura della coda dei processi pronti è utilizzata dal **scheduler**, che è il componente del sistema operativo responsabile di decidere quale processo deve ottenere il controllo della CPU. Lo scheduler si occupa di scegliere, tra i vari processi in attesa nella coda, quello da eseguire successivamente.

Quando si libera la CPU?

Quando un processo termina o cede volontariamente la CPU (ad esempio per un'operazione di **I/O**), lo **scheduler** decide a chi assegnare la CPU. È importante che la CPU non venga assegnata a un processo che è in **wait** (ad esempio, in attesa di una risorsa o di un evento esterno), perché quel processo non è pronto per l'esecuzione.

Quale compito ha il dispatcher?

I **dispatcher** ha il compito di **preparare la CPU** per eseguire un processo selezionato dallo **scheduler**. Questo processo avviene attraverso il **Process Control Block (PCB)**, che contiene tutte le informazioni necessarie per eseguire correttamente il processo, come lo stato del processo, i registri, il contesto, ecc.

In particolare, il dispatcher si occupa di

1. **Caricare il contesto del processo**
2. **Impostare la protezione della memoria**

3. **Controllo della MMU**

Quando si sono eseguite tutte queste operazioni il processo passa allo stato di esecuzione.

Cosa succede se il processo ha bisogno di una risorsa?

Quando un processo ha bisogno di una risorsa esterna, come l'interazione con le **periferiche di I/O** (ad esempio, la lettura o scrittura su un disco), o operazioni di **lettura** o **scrittura su file**, queste operazioni sono tipicamente **lente** rispetto all'esecuzione di un'istruzione della CPU. Queste operazioni non richiedono l'uso intensivo della CPU, ma richiedono che il sistema gestisca l'interazione con dispositivi esterni, che può richiedere un tempo significativamente maggiore.

Tipologie di chiamate e comportamenti associati:

Chiamate bloccanti:

- Quando un processo effettua una chiamata di I/O (ad esempio, per leggere un file o attendere una risposta da una periferica), può **bloccarsi** in attesa che l'operazione venga completata.
- Durante il blocco, il processo non può fare nulla fino a quando l'operazione di I/O non è terminata. Questo significa che la CPU può essere liberata per eseguire altri processi nel frattempo.

In questo caso, il processo passa in uno stato di **waiting** o **blocked**, in cui non è pronto per l'esecuzione.

Perché sono bloccanti?

Le chiamate di I/O sono tipicamente bloccanti perché le operazioni su periferiche esterne, come dischi, reti o dispositivi, sono **lente** rispetto alla velocità della CPU. In altre parole, il tempo necessario per leggere un dato da un disco o per scrivere su un file è molto più lungo rispetto a quello che impiega la CPU per eseguire una semplice istruzione.

Comportamento del sistema

Quando un processo è in stato **bloccato** o **waiting** a causa di una chiamata di I/O, il sistema operativo può **schedulare altri processi** in attesa di CPU.

Quando la richiesta di I/O viene conclusa, il processo dello stato di `blocked` a `ready` viene rimesso nella coda dei processi pronti e dovrà dunque attendere di essere riassegnato alla CPU dallo scheduler.

Chè cos'è il meccanismo di prelevazione?

I meccanismo di **prelazione** è un metodo che permette al sistema operativo di "riprendersi" la CPU da un processo in esecuzione, evitando che un processo monopolizzi la CPU. Se un processo non ha operazioni che richiedono **chiamate bloccanti** o **sistemi di invocazione (syscall)**, potrebbe continuare ad eseguire senza mai cedere il controllo, bloccando altri processi dall'esecuzione. Questo comportamento sarebbe inaccettabile in **sistemi interattivi** (come i desktop), dove è fondamentale che più processi possano essere eseguiti in modo equo e senza ritardi.

I processi CPU bounded

I **processi CPU bounded** sono quelli che utilizzano principalmente la CPU per le proprie operazioni, senza necessitare di operazioni di I/O (come la lettura o la scrittura su disco). Questi processi tendono a monopolizzare la CPU se non vengono interrotti.

1

Come il sistema operativo "riprende" la CPU?

Il sistema operativo utilizza il meccanismo di **prelazione** per gestire i processi che utilizzano intensamente la CPU. Questo avviene grazie all'uso degli **interrupt**, che permettono di interrompere l'esecuzione di un processo in corso per consentire al sistema di passare a un altro.

Un tipo di interrupt utilizzato per la prelazione è l'**interrupt di clock**. Questo interrupt avviene a intervalli regolari (determinati dalla frequenza di clock del sistema) e può essere configurato per attivare una **procedura di sistema**, nota anche come **handler** o **routine di gestione**. Quando si verifica questo interrupt, il sistema operativo può:

- **Interrompere il processo in esecuzione**.
- **Riprendere il controllo della CPU**.
- **Eseguire un'operazione di schedulazione** per decidere quale processo eseguire successivamente, dando la CPU a un altro processo, se necessario.

Il bloccaggio di un processo non è strettamente legato a dei dispositivi di input output ma ad un concetto più in generale legato ad altri eventi come un processo si mette in attesa di un altro processo o della sua terminazione.

Esempio

Il padre crea un processo figlio e potrebbe mettersi in attesa della terminazione del processo figlio prima di riprendere la sua esecuzione. Questo crea una sorta di **dipendenza** tra i due processi.

Il processo padre entrerà in stato di **attesa**, e solo quando il processo figlio sarà completato potrà riprendere la propria esecuzione.

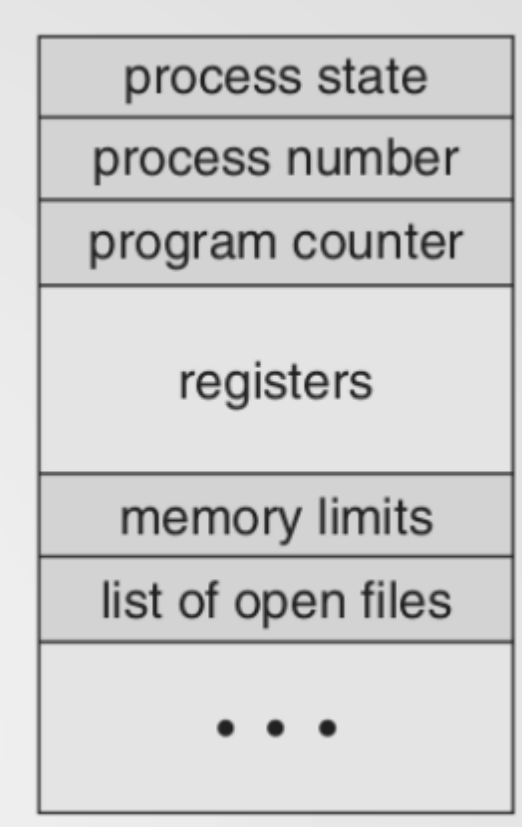
Shell(Esempio)

Un processo gestisce la **shell**. Quando si lancia un comando, questo viene eseguito come un **nuovo processo**, e al suo termine la shell restituisce il prompt all'utente.

Il processo della shell, chiamato **Pshell**, crea un **processo figlio** (ad esempio, **Pcmd**) tramite un'apposita **syscall** (come `fork()` in Unix/Linux).

- Dopo aver creato il processo figlio, **Pshell si blocca** ed entra in stato di **attesa** (`wait()`).
- Il processo **Pcmd** esegue il comando richiesto.
- Quando il processo **Pcmd termina**, la shell **riprende l'esecuzione** e mostra nuovamente il prompt.

Tabella dei Processi



Process Control Block (PCB)

Il **Process Control Block** è una **tabella dinamica**, dove ogni record rappresenta un **processo** attivo nel sistema. Il contenuto di un PCB dipende dal sistema operativo, ma solitamente include:

- **Identificativo del processo (Process ID, PID)**
- **Stato del processo** (Ready, Running, Waiting, ecc.)
- **Registri di backup**: permette di salvare il contesto del processo durante un'interruzione
- **Memory Limits**: informazioni per la protezione della memoria, indicando l'inizio e la fine delle aree di memoria accessibili dal processo
- **File aperti**: file su cui il processo sta lavorando, **file pointer**, informazioni tracciate nello stato del processo
- **Informazioni multi-utente**: dettagli relativi all'utente proprietario del processo
- **Tempo di esecuzione**: tempo totale di utilizzo della CPU da parte del processo

- **Priorità del processo**: valore che determina l'ordine di esecuzione rispetto ad altri processi

Scheduler

Lo scheduler è l'algoritmo che si occupa di scegliere quali trai i processi presenti nella coda dei processi pronti da assegnare alla CPU. Sfrutta la tecnica della prelazione.

Meccanismo degli interrupt

Gli **interrupt** sono eventi che interrompono l'esecuzione normale di un processo per gestire operazioni critiche, come richieste hardware o eventi software. Il loro trattamento si divide in due fasi: **hardware** e **software**.

Passaggi hardware

- Salvataggio di PC e PSW: vengono pushati nello stack di un processo utente oppure del kernel.
- Tabella degli interrupt: tabella che contiene gli indirizzi delle procedure a cui saltare in caso di interrupt, il salto viene eseguito a livello hardware

Passaggi software

- Salvataggio degli altri registri nel PCB del processo interrotto
- Recupero dei registri del processo dallo stack attuale
- Impostazione di un nuovo stack nello spazio di memoria del kernel responsabile della routine

Ripristino del Processo

Dopo la gestione dell'interrupt, bisogna **ripristinare il processo interrotto**.

Il **sistema operativo** può **decidere** se riprendere l'esecuzione del processo originale o assegnare la CPU a un altro processo.

Se il processo interrotto ha usato **troppo tempo** la CPU, lo scheduler può selezionare un **nuovo processo** da eseguire.

Code e accodamento

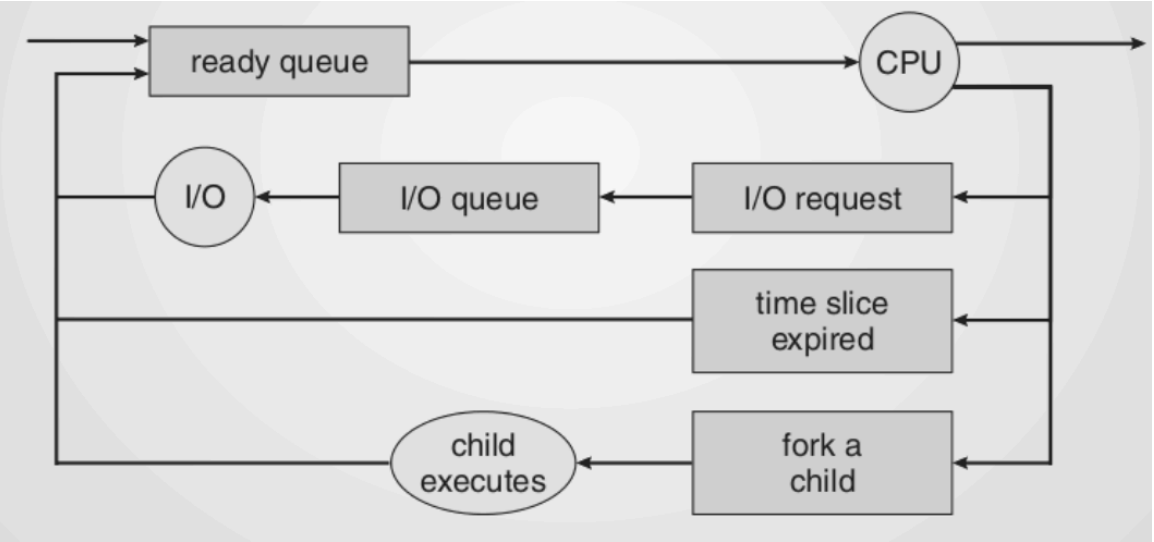
a coda dei processi pronti contiene una serie di riferimenti a processi che condividono uno stato comune, ovvero lo stato pronto. Si tratta di una serie di PCB e può essere implementata attraverso una **double-linked-list**.

È una struttura dati dinamica progettata per tracciare i processi da rappresentare, mentre la tabella dei processi è una collezione di PCB.

La coda dei processi pronti può essere considerata un sottoinsieme della tabella dei processi. In modo simile, si può immaginare una struttura dati dinamica collegata, in cui ogni record è un riferimento ai processi esistenti presenti nella tabella dei processi. In questo senso, gli stessi elementi risultano appartenere a due liste indipendenti.

È possibile inserire ed estrarre elementi dalla coda semplicemente manipolando i suoi elementi, senza modificare direttamente il contenuto della tabella dei processi.

Diagramma di Accodamento



L'immagine rappresenta il **ciclo di vita di un processo** in un sistema operativo, mostrando le transizioni tra gli stati principali di esecuzione.

Descrizione dei componenti e del flusso:

1. **Ready Queue (Coda dei processi pronti)**
    - I processi in attesa di essere eseguiti si trovano nella coda dei processi pronti.
    - Quando la CPU è disponibile, il processo viene assegnato alla CPU per l'esecuzione.
2. **CPU (Processore)**

- Il processo in esecuzione utilizza la CPU fino a quando non incontra un evento che lo sposta in un altro stato.

3. **Interazione con l'I/O (Input/Output)**
   - Se un processo necessita di un'operazione di I/O, genera una **I/O request (richiesta di I/O)**.
   - Il processo viene quindi spostato nella **I/O queue (coda di I/O)** in attesa che la risorsa diventi disponibile.
   - Dopo il completamento dell'operazione di I/O, il processo ritorna nella **Ready Queue** in attesa di essere rieseguito dalla CPU.
4. **Scadenza del time slice (Time slice expired)**
   - Nei sistemi con **scheduling a time-sharing**, un processo ha un tempo massimo di esecuzione (time slice).
   - Quando il **time slice scade**, il processo viene rimosso dalla CPU e rimesso nella **Ready Queue**, dando spazio ad altri processi.
5. **Creazione di un processo figlio (Fork a child)**
   - Se il processo esegue una chiamata di sistema per creare un processo figlio, avviene una **fork a child**.
   - Il nuovo processo figlio viene eseguito separatamente (**child executes**), mentre il processo padre può continuare la sua esecuzione o attendere la terminazione del figlio.

Thread

Viene permesso di gestire flussi multipli all’interno di un processo.

Esempio

Supponiamo di avere un'applicazione interattiva: la maggior parte dei programmi, una volta lanciati, si presentano all'utente attraverso una o più finestre e attendono input.

Nel **modello dei processi**, la gestione dei flussi multipli di esecuzione dipende dall'architettura dell'applicazione e può portare benefici significativi. Se un utente clicca su un pulsante per eseguire un'azione, il sistema potrebbe dover effettuare una chiamata di sistema, come il salvataggio di un file. Nel modello di

esecuzione sequenziale visto finora, il processo può trovarsi in una situazione problematica:

1. Se la chiamata di sistema richiede tempo, il processo viene **bloccato** e messo in **attesa** del completamento dell'operazione.
2. Poiché il processo è bloccato, non può ricevere la CPU per aggiornare l'interfaccia grafica (GUI).
3. Di conseguenza, la GUI **si congela** fino a quando l'operazione non viene completata.

Questo comportamento è una diretta conseguenza della gestione del **task** all'interno dell'applicazione. Un modo per evitarlo è l'uso del **multithreading**, che permette di separare l'interfaccia utente dalle operazioni di lunga durata. In questo modo, mentre un thread si occupa del salvataggio, un altro può continuare a gestire la GUI, mantenendo l'interattività dell'applicazione.

Ulteriore Esempio

Ad esempio, un **word processor** mentre l'utente modifica un documento può eseguire simultaneamente altri task in background, come il controllo ortografico o il salvataggio automatico. Grazie a questa indipendenza tra i task, il blocco di uno di essi non compromette il funzionamento degli altri né dell'intero programma. Questo principio si applica anche alle **applicazioni non interattive**.

Anche un **web browser** utilizza thread multipli per migliorare l'efficienza. Quando si richiede una pagina web, questa è composta da diversi elementi, come HTML, CSS, immagini e script, ciascuno dei quali richiede operazioni di rete potenzialmente bloccanti. Per evitare che il caricamento di un elemento blocchi l'intero browser, ogni operazione viene gestita come un **task separato**. Il rendering della pagina è anch'esso un task indipendente, con il proprio set di thread, garantendo così un'esperienza fluida all'utente.

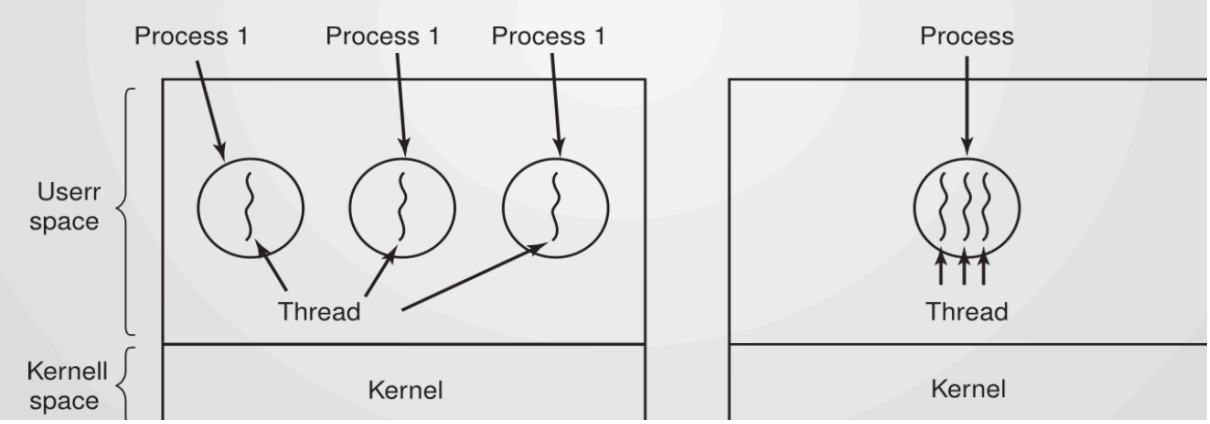
Ulteriore Esempio

I **web server**, eseguiti su macchine remote, ricevono continuamente richieste multiple per l'accesso alle risorse gestite dal server stesso. Se il server non utilizzasse più thread, durante la lettura di un file da disco resterebbe **bloccato** in attesa della risposta dell'I/O, impedendo di gestire altre richieste nel frattempo.

Per evitare questo problema, il **thread principale** può creare **on-demand** nuovi thread per gestire ogni richiesta in parallelo. Questo approccio consente di **ottimizzare il tempo di risposta** del server e garantire una gestione più efficiente delle risorse disponibili.

Modello a thread

Finora, ogni processo esegue un **singolo thread**. Quando un programma viene avviato, parte sempre con un **thread principale**, che rappresenta il flusso di esecuzione iniziale. Tuttavia, durante la sua esecuzione, il programma può creare **altri thread** per svolgere operazioni in parallelo, migliorando l'efficienza e la reattività del sistema.



A sinistra abbiamo tre processi distinti, ognuno con un **singolo thread** e il proprio **spazio di indirizzamento separato**.

Nel **modello a thread** (a destra), un **singolo processo** (identificato da un unico PID) può avere **più thread** che condividono le stesse risorse, come il **codice** e lo **heap**. Tuttavia, ogni thread ha il proprio **stack**, necessario per gestire la sua esecuzione.

Poiché tutti i thread si trovano **nello stesso spazio di indirizzamento del processo**, possono accedere direttamente alle **stesse risorse** senza bisogno di meccanismi di comunicazione inter-processo (IPC), migliorando l'efficienza e la velocità di esecuzione.

Definizioni

Consideriamo un processo P che ha **tre thread**: A, B **e** C.

Ogni thread avrà la propria **CPU virtuale**, il che significa che ognuno disporrà di un proprio **set di registri indipendenti**, tra cui il **Program Counter (PC)**.

Questo approccio consente ai thread di eseguire istruzioni in parallelo, condividendo le risorse del processo ma mantenendo un contesto di esecuzione separato.