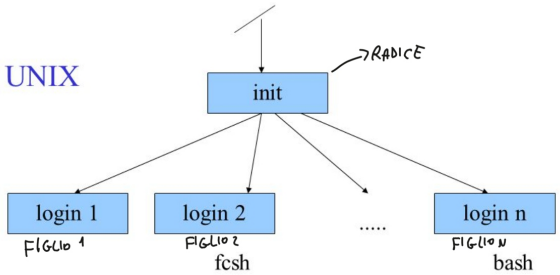
23-03-2023

Riassunto veloce

In informatica, un sistema operativo (abbreviato comunemente con i termini SO oppure OS, dall' inglese operating system, pronunciato operèiting sìstem) è un software di base adibito a gestire le risorse hardware e software di un computer
Ad ogni processi è associato il suo spazio degli indirizzi:
• codice eseguibile;
• dati del programma;
• stack;
• copia dei registri della CPU;
• file aperti;
• allarmi pendenti;
• processi imparentati.

- **Processo**: **porzione di programma** caricata in memoria. E' proprio quella che serve per essere eseguita
- E' possibile **scambiare il tempo di esecuzione** fra processi.
- Ogni processo ha il proprio **spazio degli indirizzi** che serve per interscambiare i processi fra loro
- Nella **tabella dei processi** (*Process Control Block*) ci sono le informazioni necessarie per la **cronologia di attività** svolta da quel determinato processo e serve anche per sapere da dove ripartire se questo è stato cambiato con un altro processo
- Durante lo scambio fra processi serve tenere conto del ***Program Counter*** (PC). Quest'ultimo è **unico** e ogni processo ha un suo PC logico per ogni processo. Poi questo dovrà essere caricato nel PC fisico in modo da continuare da dove si è fermato.
  - Quindi il *PC* logico viene **caricato nel *PC fisico***
  - In caso di **STOP**, il *PC fisico* viene **memorizzato sul *PC logico***
- Un processo può trovarsi in 3 stati diversi: *ready*, *running* e *blocked*.
- **CREAZIONE PROCESSI**. *3 modi*:
  - All'**avvio del sistema** e in questo modo si distinguono:
    - processi **ATTIVI**, consumano
    - processi **INATTIVI**, non appartengono a nessunun utente in particolare e sono in background (***daemon***), ovvero *dormienti* (come *il processo di stampa* che si attiva ogni volta che serve) --> Provare il comando `top` sul *Shell* per vedere i processi con il relativo stato
  - **Su richiesta dell'utente**: doppio click su un software
  - Un processo crea un altro processo tramite chiamate di sistema (`fork`) e ogni figlio, poi, sono due **ENTITA' SEPARATE** per quanto riguarda *l'attività che svolgono*(*spazio degli indirizzi diverso* ecc..)
    - E' sempre possibile trovare la **GERARCHIA DEI PROCESSI** e trovare i relativi padri dei vari processi

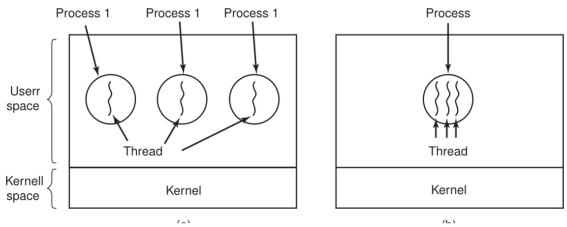


- **TERMINAZIONE PROCESSI**:
  - uscita **normale** (*volontario*) -> si usa la `exit` su UNIX o `ExitProcess` su Win32
  - uscita su **errore** (*volontario*): *per esempio* -> un puntatore punta a una locazione di memoria non istanziata
  - **errore critico**: *per esempio* -> compilazione di un file da terminale, cioè si scrive il nome di un *file inesistente*
  - **terminato da un altro processo**: tramite la chiamata di sistema `kill` (UNIX) oppure `TerminateProcess` (Win32)

- **TABELLA DEI PROCESSI**:
Process management	Memory management	File management
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		
  - Se un processo ha bisogno di più memoria di quanta ce n'è disponibile, allora deve andare a prendere la memoria dall'insieme dei processi dello stesso utente e non di un altro a caso.
  - Chiamata di sistema **Page Fault**: un processo non ha tutte le informazioni necessarie in memoria e deve accedere al disco fisso per caricare queste informazioni in *RAM*. In questo caso il processo viene **BLOCCATO** perchè l'operazione in memoria è molto lenta.
    - Se **c'è spazio in RAM** allora si carica e basta
    - Se ci sono **più utenti** allora si deve occupare la memoria destinata a tale utente di quel processo in modo da non danneggiare altri utenti.
    - Se si ha poca memoria allora è più alta la probabilità che un processo chieda dei dati non presenti in memoria. Di conseguenza, se si ha più memoria la probabilità di tale chiamata di sistema è minore ma, comunque sia, tale chiamata verrà fatta prima o poi perchè, inizialmente, i processi iniziano a lavorare con dei dati incompleti, parziali.
    - Per questo motivo la tabella dei processi memorizza anche il **proprietario del processo**
- Ogni processo si divide in **mini sottoprocessi** ognuno con un compito diverso e in questo modo si ottimizza a **livello tempistico** e ogni miniprocesso è detto **THREAD**

I thread

Un processo è suddiviso in tanti filamenti destinati ad un'attività specifica che permette di simulare un **PARALLELISMO**.



Un thread è caratterizzato da:
• PC, registri, stack, stato;
• condivide tutto il resto;
non protezione di memoria.

> *Entità indipendente da altri processi che esegue dei dati e per fare questo si tiene traccia di molte informazioni. Quindi un processo è un entità che raggruppa risorse utili al processo.

- - Un processo, in generale, gestisce le risorse e poi le esegue.
- Le **risorse sono sempre uguali** ma il **flusso si può spezzare** in diversi (*almeno 2*) **sotto-flussi** di uno **STESSO PROCESSO** e ognuno di essi è detto **THREAD**

> *In caso si abbia solo **un processore/un solo core** allora si parla sempre di **PSEUDO-PARALLELISMO***

Vantaggi nell'uso dei thread

Un programma è fatto dal gruppo di risorse e dalla parte di esecuzione. Se la parte di **esecuzione si divide** su più attività distinte, le **risorse** rimangono le **stesse**. Quindi ogni thread usa le stesse risorse.

> Esempio: *I **file globali** sono **uniche** per tutti i thread*
> **Cambiano** solo **dati specifici** per quel thread, come le *variabili locali*

- **Si ha tutto caricato in memoria** relativa al processo e ogni thread si alterna fra loro e ogni scambio fra thread **non c'è bisogno di ricaricare dal disco nuovi dati** (che sono **pesanti**).
- In sintesi: uso **thread diversi** ma uso gli **stessi dati** che il processo ha caricato in memoria.
- Si ricava **un particolare guadagno di prestazioni** quando le diverse attività si alternano e quindi quando i processi sono ***I/O bound*** (fanno SPESSO richieste I/O)
  - Se un thread fa richieste I/O allora si blocca ***SOLO QUEL THREAD*** e gli altri vanno avanti

> *Rendono un sistema più veloce e riducono i tempi di attesa:*

Esempi

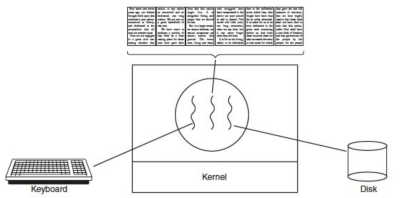


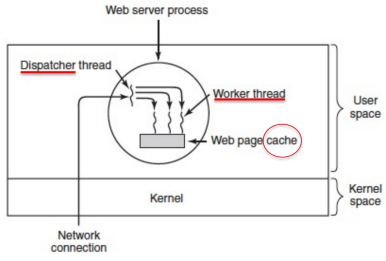
Figure 2-7. A word processor with three threads.

Un esempio può essre un **programma di video-scrittura** con un *testo già scritto* (il testo è già formattato)

- A partire dalla prima pagina costruisce la prima pagina per capire e formattare le successive
- In caso di *modifica di testo*, allora cambia tutto l'intero testo.
- Se ho **un solo processo**:
  - Esso si occupa dell'input/output e allo stesso tempo formattare
  - Se si cerca una parola, allora si deve analizzare tutto il testo e valutare il punto in cui si è arrivati in un certo istante di ricerca
- Se **usassi i thread**:
  - ognuno si potrebbe occupare di un'attività diversa.
  - Potrei usare 2 thread: **ricevere I/O** e **formattare**
  - Potrei usarne un altro thread che si occupa di salvare su disco fisso il file (*daemon*)

Un altro esempio:

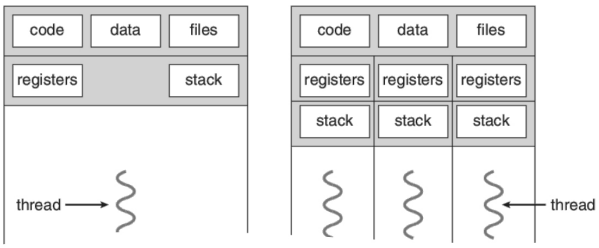
Un **web server process** usa un sistema **CACHE** in modo tale da fornire *più velocemente possibile* le pagine *più spesso visualizzate*



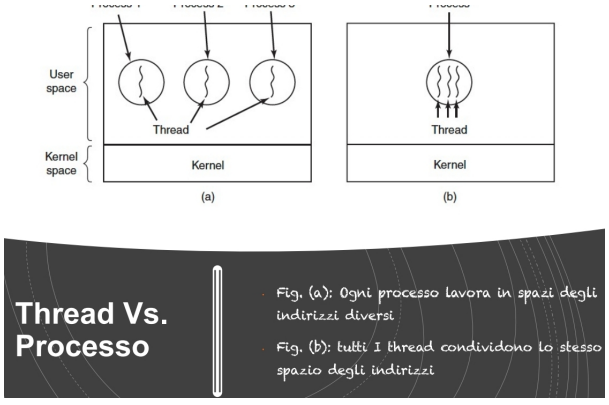
- Si devono usare i thread per assegnare a ognuon di essi un **compito specifico**: **ricevere** la richiesta (*dispatcher thread*), **cercare** la pagina nella cache (*worker thread*) , **caricare** dal disco fisso una pagina SE **non presente in cache**, visualizzare la pagina web
  - In questo *se un thread rimane bloccato* oppure rallenta l'esecuzione gli altri thread possono *continuare a lavorare* cooperando con gli altri per l'obiettivo finale **senza interferire uno sull'altro**

> *il **dispatcher** è un* ciclo infinito *che acquisisce richieste di lavoro e le passa ad un worker thread*

Modello Thread



- Nella *figura a sinistra* c'è un solo thread
- Nella *figura a destra* ci sono 3 thread (*filamenti*) e i dati relativi alla sua attività (`Program Counter` , `registri` , `Stack` , *attività specifica*) sono **PROPRI** del thread
  Si deduce il vantaggio (quando si **cambia un thread** con un altro thread): l'operazione è **più veloce** perchè si **devono scambiare solo le informazioni che riguardano il singolo thread** (che chiaramente sono minori rispetto alle informazioni memorizzate in un *INTERO PROCESSO* che si devono scambiare in caso di **scambi di processi**)



`Figura a:`

- **3 processi** con un solo thread
- **Il passaggio fra proessi** comporta il passaggio delle informazioni dal processo stesso con il successivo
- Questo richiede un notevole **consumo di tempo**

`Figura b:`

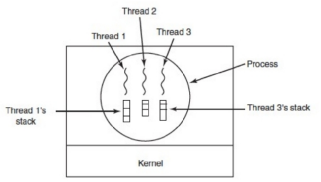
- **3 thread** e 1 processo
- lo **scambio** fra i thread è più **veloce**
- si completa l'operazione molto **più velocemente** rispetto alla `figura a`

Possibile problema (teorico) con i thread

- Thread dello stesso processo sono attività diverse fra loro ma **NON** sono del tutto indipendenti e questo è ai fini della condivisione di risorse e informazioni. (comunicazioni indirette)
- Questo vuol dire che condividono gli stessi dati e ciò potrebbe creare dei problemi perchè non c'è alcuna protezione nelle risorse condivise
    - *Teoricamente*, un thread può leggere, scrivere o cancellare lo stack di un altro thread
- *Praticamente*, però, le **informazioni condivise** sono usate **PER SCAMBIARE** informazioni ma visto che svolgono **attività diverse** allora non useranno mai la stessa parte di memoria e quindi ***NON SI SOVRAPPONGONO***.

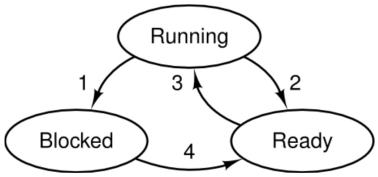
> *Se un thread apre un file, ciò è visibile ad ogni thread del processo. Sembra un problema ma la risorsa è gestita dal processo e non dal thread.*

Differenze fra processo e thread



Per-process items	Per-thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

`A sinistra`: un processo con 3 thread e **ogni thread** ha un **proprio stack**
`A destra`: ci sono le **informazioni relative a un thread e ad un processo** e ogni thread **può assumere DIVERSI STATI** (*Blocked*, *Running*, *Ready*) proprio *come i processi*.



- Il **processo** è **gestore delle risorse**
- Il **thread** è il flusso di esecuzione del processo

Multithread: Ci sono **molteplici Thread** nello stesso processo

> *Lo scambio fra processi è circa 40 volte più lento rispetto a uno scambio fra thread*

Conoscenza del proprietario di un processo

Lo **SCHEDULING dei thread** determina l'**ordine di esecuzione** dei thread. Un processo è scomposto in thread perchè si **velocizzano le operazioni**. Se un thread si blocca, allora viene sostituito da un altro thread (dello stesso processo, chiaramente) seguendo le linee guida dello scheduling. Il SO, se conosce il **proprietario di un thread**, va a ricercare thread dello stesso processo dello stesso utente per utilizzare al massimo il vantaggio dell'uso dei thread.