
ANNO ACCADEMICO 2024/2025

Sistemi Operativi

Teoria

Dionesalvi's Notes



UNIVERSITÀ
DI TORINO

DIPARTIMENTO DI INFORMATICA

CAPITOLO 1	INTRODUZIONE	PAGINA 1
1.1	Prima Lezione Architetture Single/Multi-Core — 1 • Tipi di Eventi — 1 • Gestione degli Eventi — 1	1
1.2	Struttura della Memoria	2
1.3	Gerarchia delle Memorie	2
1.4	Struttura di I/O	3
1.5	Multitasking e Time-Sharing Time-Sharing — 4	3
1.6	Compiti del sistema operativo Duplice modalità di funzionamento — 5 • Timer — 5 • Protezione della memoria — 6	4
CAPITOLO 2	STRUTTURE DEI SISTEMI OPERATIVI	PAGINA 7
2.1	Interfaccia del Sistema Operativo Interprete dei comandi — 8	8
2.2	Interfaccia grafica	8
2.3	Programmi/servizi di Sistema	8
2.4	Chiamate di sistema (Syscall) Chiamate di sistema: le "API" — 9	8
2.5	Gestione dei processi	9
2.6	Gestione dei file e del filesystem	9
2.7	Macchine Virtuali	10
CAPITOLO 3	GESTIONE DEI PROCESSI	PAGINA 11
3.1	Processi Concetto di processo — 11 • Stato del processo — 12 • Processo Control Block (PCB) — 13	11
3.2	Scheduling dei processi Il cambio di contesto (context switch) — 14 • Code di scheduling — 15 • CPU Scheduler — 16	14
3.3	Operazione sui processi Creazione di un processo — 17 • Creazione di un processo in Unix — 18 • Passi dell'SO all'invocazione delle fork — 18 • Altro esempio — 19 • Osservazioni — 20 • Terminazione di un processo — 20	16
3.4	Comunicazione tra processi	21
3.5	Esempio: il problema Produttore-Consumatore	21

1

Introduzione

1.1 Prima Lezione

Un Sistema Operativo (SO) agisce come intermediario tra l'utente e l'hardware, fornendo gli strumenti per un uso corretto delle risorse della macchina (CPU, memoria, periferiche). Ha due obiettivi principali:

- Dal punto di vista dell'utente: rendere il sistema facile da usare.
- Dal punto di vista della macchina: ottimizzare l'uso delle risorse in modo sicuro ed efficiente.

1.1.1 Architetture Single/Multi-Core

Negli anni 2000 si è passati da processori single-core a multi-core, con CPU dotate di più core in grado di eseguire istruzioni di programmi diversi simultaneamente.

1.1.2 Tipi di Eventi

- **Interrupt:** Eventi di natura hardware, rappresentati da segnali elettrici inviati da componenti del sistema.
- **Eccezioni:** Eventi di natura software, causati dal programma in esecuzione. Le eccezioni si dividono in:
 - *Trap:* Causate da malfunzionamenti del programma (es. accesso a memoria non autorizzato, divisione per 0).
 - *System Call:* Richiesta di servizi al SO, come l'accesso ai file.

1.1.3 Gestione degli Eventi

Quando si verifica un evento:

1. **Salvataggio dello stato della CPU:** Il Program Counter (PC) e i registri della CPU vengono salvati in appositi registri speciali per poter riprendere l'esecuzione successivamente.
2. **Esecuzione del codice del SO:** Il PC viene aggiornato con l'indirizzo del codice del SO che gestisce l'evento, memorizzato in una tabella detta *vettore delle interruzioni*. Questo vettore contiene puntatori a differenti routine di gestione eventi.
3. **Return:** Una volta gestito l'evento, il SO ripristina lo stato precedente e l'esecuzione del programma sospeso riprende.

Note:-

Nel Program Counter viene scritto l'indirizzo in RAM della porzione di codice del So che serve a gestire l'evento che si è appena verificato. All'accensione del computer, il SO stesso carica in aree della RAM che il SO riserva a se stesso le varie porzioni di codice eseguibile che dovranno entrare in esecuzione quando si verifica un'eccezione.

Osservazioni 1.1.1

Nei primi N indirizzi della RAM viene caricato una array di puntatori noto come **vettore delle interruzioni**. Ogni entry del vettore contiene l'indirizzo di partenza in RAM di una delle porzioni di codice del SO del punto precedente.

Quando un certo evento si verifica, il program counter viene aggiornato con il valore che è indicato nella cella di memoria collegata all'entry point dell'eccezione. L'ultima istruzione di ogni procedura di gestione di un evento sarà sempre una istruzione di "return from event" (**ra**).

1.2 Struttura della Memoria

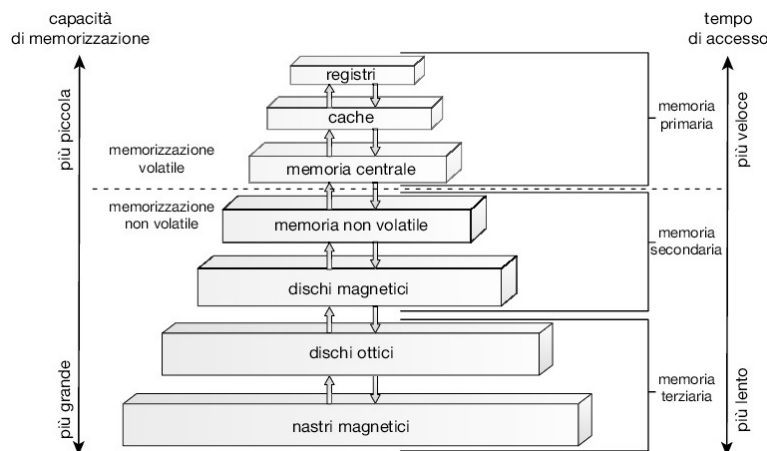
Nel contesto del **SO**, ci sono due principali tipi di memoria:

- **Memoria Principale (RAM)**: Memoria primaria in cui risiedono programmi e dati durante l'esecuzione.
- **Memoria Secondaria**: Memoria di massa, come **hard disk** o **memorie a stato solido**, utilizzata per la conservazione permanente dei dati.

1.3 Gerarchia delle Memorie

Nella figura: *Velocità implica complessità maggiore, costo maggiore e capacità minore.*

- **Caching**: Ogni livello di memoria fa da cache per il livello successivo. Esempio: la **RAM** fa da cache per l'**hard disk**, la **CACHE** per la **RAM**, e i **registri della CPU** per la **CACHE**.

**Domanda 1.1**

Sarebbe bello avere 500GB di registri di CPU o Hard Disk veloci quando i registri della CPU?

Per una informazione la si deve copiare in una memoria più veloce (più costosa). La **RAM** fa da cache per l'HDD. La **CACHE** fa da cache per la RAM I **REGISTRI** fanno da cache per la CACHE.

Due tecnologie di memoria RAM

- SRAM: per la cache e i registri della CPU
 - Creato con i FLIP-FLOP, questo porta un costo maggiore ma una maggiore efficienza rispetto alla DRAM.
- DRAM: per la memoria principale/centrale
 - Creato con i Condensatori, che tendono a perdere il loro stato. Questo obbliga a refresharli costantemente portando un dispendio maggiore di energie ma un minore costo di produzione.

1.4 Struttura di I/O

Un generico computer è composto da una CPU e da un insieme di dispositivi di I/O connessi fra loro da un bus comune. Ogni dispositivo di I/O è controllato da un apposito componente hardware detto **controller**. Il controller è a sua volta un piccolo processore, con alcuni registri e una memoria interna, detto **buffer**. Il SO interagisce con il controller attraverso un software apposito noto come **driver del dispositivo**.

Esempio 1.4.1 (Driver)

Il driver del dispositivo, carica nei registri dei controller opportuni valori che specificano le operazioni da compiere.
 Il controller esamina i registri e intraprende l'operazione corrispondente.
 Il controller trasferisce i dati dal dispositivo al proprio buffer.
 Il controller invia un interrupt al SO indicando che i dati sono pronti per essere prelevati.

Questo modo di gestire l'I/O con grandi quantità di dati è molto **inefficiente**. Una soluzione utile è avere un canale di comunicazione diretto tra il dispositivo e la RAM, in modo da non "disturbare" troppo il SO. Tale canale è detto **Direct Memory Access (DMA)**. Il SO, tramite il driver del disco, istruisce opportunamente il controller del disco, con un comando (scritto nei registri del controller) del tipo:

Osservazioni 1.4.1

Trasferisci il blocco numero 1000 del disco in RAM a partire dalla locazione di RAM di indirizzo F2AF

Il controller trasferisce direttamente il blocco in RAM usando il DMA, e ad operazione conclusa avverte il SO mediante un interrupt opportuno.

1.5 Multitasking e Time-Sharing

Quando lanciamo un programma, il SO cerca il codice del programma sull'hard disk, lo copia in RAM, e *"fa partire il programma"*. Noi utenti del SO non dobbiamo preoccuparci di sapere dov'è memorizzato il programma sull'hard disk, né dove verrà caricato in RAM per poter essere eseguito.

Dunque, il SO rende **facile** l'uso del computer. Ma il SO ha anche il compito di assicurare un uso **efficiente** delle risorse del computer, in primo luogo la CPU stessa.

Osservazioni 1.5.1

Consideriamo un programma in esecuzione: a volte deve fermarsi temporaneamente per compiere una operazione di I/O (esempio: leggere dall'hard disk dei dati da elaborare). Fino a che l'operazione non è completata, il programma non può proseguire la computazione, e non usa la CPU.
 Invece di lasciare la CPU inattiva, perché non usarla per far eseguire il codice di un altro.

Questo è il principio della multiprogrammazione (multitasking), implementato da tutti i moderni SO: il SO mantiene in memoria principale il codice e i dati di più programmi che devono essere eseguiti. (Detti anche job)



Domanda 1.2

Alcune applicazioni degli utenti però sono per loro natura interattive, come fa ad esserci una interazione continua tra il programma e l'utente che lo usa?

Oltre a questo, i sistemi di calcolo son multi-utente cioè permettono di essere connessi al sistema e di usare "contemporaneamente" il sistema stesso.

1.5.1 Time-Sharing

È meglio allora **distribuire** il tempo di CPU fra i diversi utenti (i loro programmi in "esecuzione") frequentemente (ad esempio ogni 1/10 di secondo) così da dare una impressione di **simultaneità** (che però è solo apparente).

Questo è il **time-sharing**, che estende il concetto di **multiprogrammazione**, ed è implementato in tutti i moderni sistemi operativi.

1.6 Compiti del sistema operativo

E' necessario tenere traccia di tutti i programmi **attivi** nel sistema, che stanno usando o vogliono usare la CPU, e gestire in modo appropriato il passaggio della CPU da un programma all'altro, nonché **lanciare** nuovi programmi e **gestire** la terminazione dei vecchi.

Note:-

Questo è il problema della gestione dei processi (cap. 3) e dei thread (cap. 4)

Quando la CPU è libera, e più programmi vogliono usare, a quale programma in RAM assegnare la CPU?

Note:-

Questo è il problema di CPU Scheduling (cap. 5)

I programmi in esecuzione devono interagire fra loro senza danneggiarsi ed evitando situazioni di stallo (ad esempio, il programma A aspetta un dato da B che aspetta un dato da C che aspetta un dato da A)

Note:-

Questi sono i problemi di sincronizzazione (cap. 6/7) e di deadlock (stallo dei processi) (cap. 8)

Come gestire la RAM, in modo da poterci far stare tutti i programmi che devono essere eseguiti? Come tenere traccia di quali aree di memoria sono usate da quali programmi?

Note:-

La soluzione a questi problemi passa attraverso i concetti di gestione della memoria centrale (cap. 9) e di memoria virtuale (cap. 10).

Infine, un generico computer è spesso soprattutto un luogo dove gli utenti **memorizzano** permanentemente, organizzano e recuperano vari tipi di informazioni, all'interno di "contenitori" detti **file**, a loro volta suddivisi in cartelle (o folder, o directory) che sono organizzate in una struttura gerarchica a forma di albero (o grafo aciclico) nota come **File System**.

Note:-

Il SO deve gestire in modo efficiente e sicuro le informazioni memorizzate nella memoria di massa (o secondaria) (cap. 11) deve permettere di organizzare i propri file in modo efficiente, ossia fornire una adeguata interfaccia col file system (cap. 13), deve implementare il file system (cap. 14)

Domanda 1.3

come fa il SO a mantenere sempre il controllo della macchina?

Soprattutto, come fa anche quando non sta girando? Ad esempio, come evitare che un programma utente acceda direttamente ad un dispositivo di I/O usandolo in maniera impropria? Oppure, che succede se un programma, entra in un loop infinito? E' necessario prevedere dei modi per proteggersi dai malfunzionamenti dei programmi utente (voluti, e non)

1.6.1 Duplice modalità di funzionamento

Nei moderni processori le istruzioni macchina possono essere eseguite in due modalità diverse:

1. normale (modalità utente)
2. di sistema (modalità privilegiata, o kernel / monitor / supervisor mode)

La CPU è dotata di un "bit di modalità" di sistema (0) o utente (1), che permette di stabilire se l'istruzione corrente è in esecuzione per conto del SO o di un utente normale.

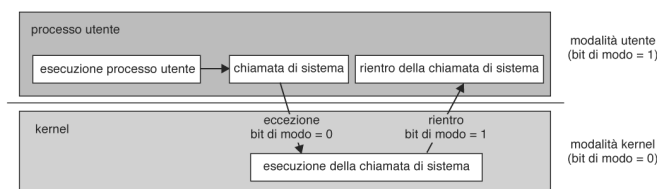
Osservazioni 1.6.1

Le istruzioni macchina **sensibili**, nel senso che se usate male possono danneggiare il funzionamento del sistema nel suo complesso, possono essere eseguite solo in modalità di sistema, e quindi solo dal SO, altrimenti se nel codice di un programma normale in esecuzione è contenuta una istruzione delicata, quando questa istruzione entra nella CPU viene generata una **trap**.

I programmi utente hanno a disposizione le **system call** (chiamate di sistema) per compiere operazioni che richiedono l'esecuzione di istruzioni privilegiate.

Una system call si usa in un programma come una normale subroutine, ma in realtà provoca una **eccezione**, e il controllo passa al codice del SO di gestione di quella eccezione.

Ovviamente, quando il controllo passa al SO, il bit di modalità viene settato in modalità di **sistema** in modo automatico, via **hardware**.



Si dice di solito che il processo utente sta eseguendo in **kernel mode**

1.6.2 Timer

Domanda 1.4

Che succede se un programma utente, una volta ricevuto il controllo dalla CPU, si mette ad eseguire il seguente codice: `for(;;)i++;`?

Per evitare questo tipo di problemi, nella CPU è disponibile un **Timer**, che viene inizializzato con la quantità di tempo che si vuole concedere **consecutivamente** al programma in esecuzione. Qualsiasi cosa faccia il programma in esecuzione, dopo 1/10 di secondo il Timer invia un **interrupt** alla CPU, e il controllo viene restituito al sistema operativo. Il SO **verifica** che tutto stia procedendo regolarmente, riinizializza il Timer e decide quale programma mandare in esecuzione, questa è l'essenza del **time-sharing**

Osservazioni 1.6.2

Ovviamente, le istruzioni macchina che gestiscono il timer, sono istruzioni privilegiate. Altrimenti un programma utente potrebbe modificare semplicemente i valori :D

1.6.3 Protezione della memoria

Domanda 1.5

Cosa succede se un programma in esecuzione scrive i dati di un altro programma in "esecuzione"?

E' necessario proteggere la memoria primaria da accessi ad aree riservate.

Due possibili soluzioni

Una possibile soluzione: in due registri appositi della CPU (base e limite) il SO carica gli indirizzi di inizio e fine dell'area di RAM assegnata ad un programma.

Ogni indirizzo I generato dal programma in esecuzione viene **confrontato** con i valori contenuti nei registri base e limite.

$$\text{Se } I < \text{base} \vee I > \text{limite} \implies \text{TRAP!}$$

I controlli vengono fatti in parallelo a livello hardware, altrimenti richiederebbero troppo tempo.

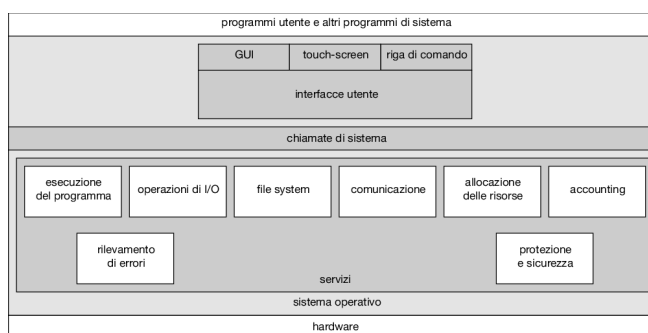
Un'altra variante: simile: in due registri appositi della CPU il SO carica rispettivamente l'indirizzo di inizio (base) e la dimensione (offset) dell'area di RAM assegnata ad un programma.

$$\text{Se } I < \text{base} \vee I > \text{base} + \text{offset} \implies \text{TRAP!}$$

2

Strutture dei Sistemi Operativi

Un sistema operativo mette a disposizione degli utenti (e dei loro programmi) molti servizi



Alcuni di questi servizi sono completamente invisibili agli utenti, altri sono parzialmente visibili, e altri sono direttamente usati dagli utenti. Ma il *grado di visibilità* dipende anche dal tipo di utente (Root, user, group)

Esempio 2.0.1 (Esempi di visibilità)

- Interfaccia col sistema operativo (terminale) (visibili)
- Chiamate di sistema (quasi sempre visibili)
- Gestione di processi (praticamente invisibili)

2.1 Interfaccia del Sistema Operativo

L'interfaccia è lo strumento con il quale gli utenti interagiscono con il So, e ne sfruttano i servizi offerti.

Può essere un **interprete di comandi**, o un **interfaccia grafica** con finestre e menù, ma di solito è possibile usare una combinazione di entrambi

2.1.1 Interprete dei comandi

Normalmente non fa parte del **kernel** SO; ma è un programma (o collezione di essi) fornito insieme al SO.

Un esempio d'interprete è la **shell** dell'MS-Dos oppure la **shell** Unix.

Una shell rimane semplicemente in attesa di ciò che l'utente scrive da linea di comando, ed ovviamente, esegue

il comando stesso. Spesso, i comandi che possono essere usati dagli utenti del SO sono dei semplici **eseguibili**. L'interprete si occupa di trovare sull'hard disk e lanciare il codice dell'eseguibile passando eventuali argomenti specificati.

Esempio 2.1.1 (Comando shell unix)

1. L'utente scrive *rm myfile*
2. l'interprete cerca un file eseguibile di nome "rm" e lo lancia, passandogli come parametro "myfile"

Note:-

Un comando utile può essere *ps* che ti permette di vedere i processi attaccati alla tua shell

2.2 Interfaccia grafica

I moderni SO offrono anche una interfaccia grafica (GUI) per gli utenti, spesso più facile da imparare ed usare. **Unix** offre varie interfacce grafiche, sia proprietarie che open-source, come **KDE** e **GNOME**, e ogni utente del SO può scegliersi la sua

2.3 Programmi/servizi di Sistema

Non fanno parte del kernel del SO, ma vengono forniti insieme al SO, e rendono più facile, comodo e conveniente l'uso del Sistema.

Gli interpreti dei comandi e le interfacce grafiche sono gli esempi più evidenti di programmi di sistema.

Oltre a questi: editor, compilatori, browser, task manager etc etc.

2.4 Chiamate di sistema (Syscall)

Da ora in poi, chiameremo un programma in "esecuzione" come **processo**. Le system call costituiscono la vera e propria interfaccia tra i processi degli utenti e il Sistema Operativo.

Ad esempio, in Unix assumono la forma di procedure che possono essere inserite direttamente in programmi scritti con linguaggi ad alto livello (C, C++, ...)

Sembra di usare una **subroutine**, ma l'esecuzione della system call trasferisce il controllo al SO, e in particolare alla porzione di codice del SO che implementa la particolare System Call invocata.

Ad esempio, in un programma C, per scrivere dentro ad un file:

```
fd = open("nomefile", O_WRONLY);
i = write(...)
close(fd)
```

Open, write e close sono delle syscall

2.4.1 Chiamate di sistema: le "API"

Application Programming Interface Le API non sono altro che uno strato intermedio tra le applicazioni sviluppate dai programmatori e le syscall, per rendere più **facile** l'uso e migliorare la **portabilità** tra versioni.

Esempio 2.4.1 (Chiamate)

Ad esempio, la libreria C dell'ambiente Unix è una semplice forma di API. In questa libreria esiste la funzione per aprire un file:

fopen, fprintf e fclose

2.5 Gestione dei processi

In un dato istante, all'interno di un SO sono attivi più processi (anche se uno solo è in esecuzione, in un dato istante). Si parla allora di **Processi Concorrenti**, perchè si contendono l'uso delle risorse hardware della macchina.

1. La CPU
2. Lo spazio in memoria primaria e secondaria
3. I dispositivi di input e output

Il SO ha la responsabilità di fare in modo che ogni processo abbia la sua parte di risorse, senza danneggiare e venire danneggiato dai altri processi.

Il SO quindi deve gestire tutti gli aspetti riguardo la vita dei processi.

- Creazione e cancellazione dei processi
- Sospensione e riavvio dei processi
- Sincronizzazione tra i processi
- Comunicazione tra processi

Per eseguire un programma deve essere caricato in memoria principale.

In un sistema time-sharing, più processi possono essere contemporaneamente attivi: il loro codice e i loro dati sono caricati in qualche area della RAM. Quindi il SO deve:

- Tenere traccia di quali parti della RAM sono utilizzati e da quale processo
- Distribuire la RAM tra i processi
- Gestire la RAM in base alla necessità e ai cambiamenti

2.6 Gestione dei file e del filesystem

Quasi ogni informazione presente in un sistema è contenuta in un file: una raccolta di informazioni denotata da un nome (e di solito da altre proprietà).

I file sono organizzati in una struttura **gerarchica** detta File System, mediante le cartelle (o directory, o folder) Il SO è responsabile della:

- Creazione e cancellazione
- Fornitura di strumenti per gestire i file e dir
- Memorizzazione efficiente del file system in memoria secondaria.

I file sono memorizzati permanentemente in memoria secondaria, di solito su un hard disk.

Il SO deve:

- decidere dove e come memorizzare i file su disco, ed essere in grado di ritrovarli velocemente.
- Trovare spazio libero velocemente quando un file è creato o aumenta di dimensione, e recuperare spazio alla rimozione di un file.
- Gestire efficientemente accessi concorrenti ai file dai vari processi attivi.

2.7 Macchine Virtuali

Un moderno SO trasforma una macchina reale in una sorta di macchina virtuale (MV).

3

Gestione dei processi

3.1 Processi

Il processo è l'unità di lavoro del sistema operativo, perché ciò che fa un qualsiasi SO è innanzi tutto amministrare la vita dei processi che girano sul computer gestito da quel SO. Il sistema operativo è responsabile della creazione e cancellazione dei processi degli utenti, gestisce lo scheduling dei processi, fornisce dei meccanismi di sincronizzazione e comunicazione fra i processi.

3.1.1 Concetto di processo

- Un **processo** è più di un semplice programma in esecuzione, infatti, ha una struttura in memoria primaria, suddivisa in più parti assegnategli dal sistema operativo (vedi fig. 3.1).
- Le principali componenti della struttura di un processo sono:
 - **Codice** da eseguire (il "testo")
 - **Dati**
 - **Stack** (per le chiamate alle procedure/metodi e il passaggio dei parametri)
 - **Heap** (memoria dinamica)
- La somma di queste componenti forma l'immagine del processo:

codice + dati + stack + heap = immagine del processo

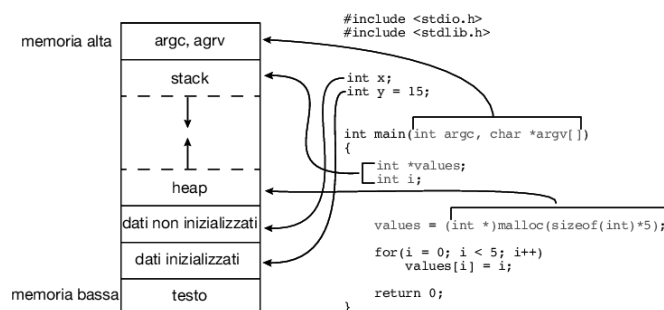


Figure 3.1: Concetto di processo

È anche corretto osservare che attraverso un programma si possono definire più processi, infatti:

- Lo stesso programma può contenere codice per generare più processi
- Più processi possono condividere lo stesso codice

Tuttavia, la distinzione fondamentale tra processo e programma è che un processo è **un'entità attiva**, mentre un programma è **un'entità statica**.

Domanda 3.1

Lo stesso programma lanciato due volte può dare origine a due processi diversi (perché?)

Attenzione: processo, task, job sono **sinonimi**.

Un programma si **trasforma** in un processo quando viene lanciato, con il doppio click o da riga di comando. Un processo può anche **nascere** a partire da un altro processo, quando quest'ultimo esegue una opportuna system call (fork, spawn, etc)

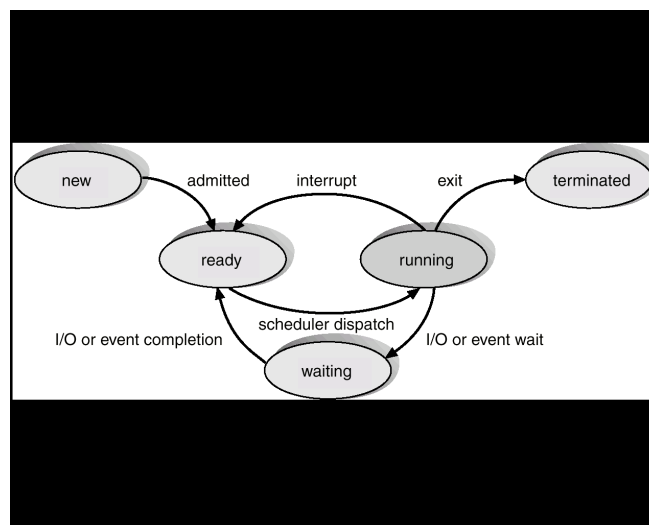
Definizione 3.1.1: Processo

In realtà, non sono due meccanismi distinti: un processo nasce sempre a partire da un altro processo, e sempre sotto il controllo e con l'intervento del SO (con un'unica eccezione, all'accensione del sistema).

3.1.2 Stato del processo

Da quanto nasce a quando termina, un processo passa la sua esistenza muovendosi tra un insieme di stati, e in ogni stante ogni processo si trova in un ben determinato stato.

Lo stato di un processo evolve a causa del codice eseguito e dell'azione del SO sui processi presenti nel sistema in un dato istante, secondo quanto illustrato dal diagramma di transizione degli stati di un processo.



Gli stati

Gli stati in cui può trovarsi un processo sono:

Definizione 3.1.2: Stati del processo

- **New:** Il processo è appena stato creato
- **Ready (to Run):** Il processo è pronto per entrare in esecuzione
- **Running:** La CPU sta eseguendo il codice del processo
- **Waiting:** Il processo ha lasciato la CPU e attende il completamento di un evento
- **Terminated:** Il processo è terminato, il SO sta recuperando le strutture dati e le aree di memoria liberate

Il diagramma di transizione degli stati di un processo sintetizza una serie di possibili varianti del modo in cui un sistema operativo (SO) può amministrare la vita dei processi di un computer.

- Infatti, nel caso reale lo sviluppatore del SO dovrà decidere quali scelte implementative fare quando (ad esempio):
 - Mentre il processo P_x è *running*, un processo entra nello stato *Ready to Run*
 - Mentre il processo P_x è *running*, un processo più importante di P_x entra nello stato *Ready to Run*
 - Mentre il processo P_x è nello stato *Ready to Run*, un processo più importante di P_x entra nello stato *Ready to Run*

Domanda 3.2

Che significato ha eliminare l'arco "interrupt"?

Di avere un sistema non time-sharing

3.1.3 Processo Control Block (PCB)

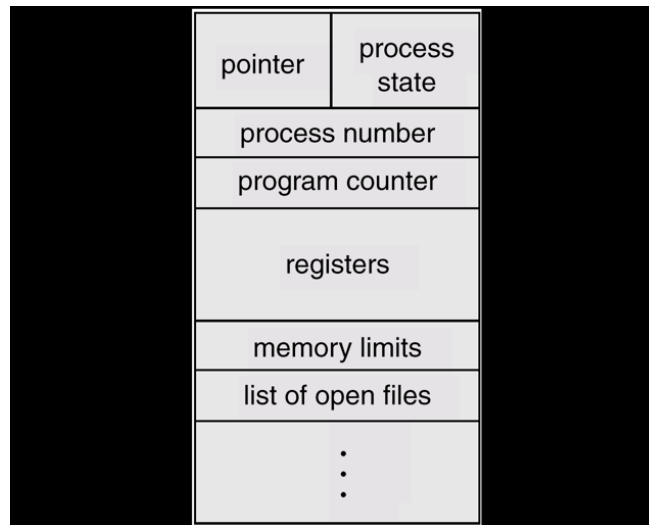
Per ogni processo, il sistema operativo (SO) mantiene una struttura dati chiamata *Process Control Block* (PCB), che contiene le informazioni necessarie per amministrare la vita di quel processo, tra cui:

- Il numero del processo (o *Process ID*)(PID)
- Lo stato del processo (*ready, waiting,...*)
- Il contenuto dei registri della CPU salvati nel momento in cui il processo è stato sospeso (valori significativi solo quando il processo non è *running*)
- Gli indirizzi in RAM delle aree dati e codice del processo
- I file e gli altri dispositivi di I/O correntemente in uso dal processo
- Le informazioni per lo *scheduling* della CPU (ad esempio, quanta CPU ha usato fino a quel momento il processo)

3.2 Scheduling dei processi

Conosciamo già i seguenti due concetti:

- **Multiprogrammazione:** avere sempre un processo *running* \Rightarrow massima utilizzazione della CPU.
- **Time Sharing:** distribuire l'uso della CPU fra i processi a intervalli prefissati. Così più utenti possono usare "allo stesso tempo" la macchina, e i loro processi procedono in "parallelo" (notate sempre le virgolette).



Definizione 3.2.1: Scheduling

Per implementare questi due concetti, il sistema operativo deve decidere periodicamente quale sarà il prossimo processo a cui assegnare la CPU. Questa operazione è detta *Scheduling*.

In un sistema time sharing single-core, attraverso lo scheduling, ogni processo “crede” di avere a disposizione una macchina “tutta per sé”... Ci pensa il SO a farglielo credere, **commutando** la CPU fra i processi (ma succede la stessa cosa in un sistema ad n-core se ci sono più di n processi attivi contemporaneamente)

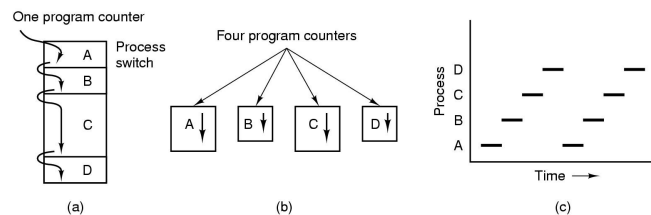


Figure 3.2: a) Ciò che succede in realtà
b) ciò che vede ogni singolo processo
c) Il risultato finale

3.2.1 Il cambio di contesto (context switch)

Per commutare la CPU tra due processi, il sistema operativo deve:

1. Riprendere il controllo della CPU (ad esempio attraverso il meccanismo del *Timer* visto nel capitolo 1).
2. Con l'aiuto dell'hardware della CPU, salvare lo stato corrente della computazione del processo che lascia la CPU, ossia copiare il valore del *Program Counter* (PC) e degli altri registri nel suo *Process Control Block* (PCB).
3. Scrivere nel PC e nei registri della CPU i valori relativi contenuti nel PCB del processo utente scelto per entrare in esecuzione.

Questa operazione prende il nome di: **cambio di contesto**, o *context switch*.

Notate che, tecnicamente, anche il punto 1 è già di per sé un *context switch*.

- Il *context switch* richiede tempo, perché il contesto di un processo è composto da molte informazioni (alcune le vedremo quando parleremo della gestione della memoria).

- Durante questa frazione di tempo, la CPU non è utilizzata da alcun processo utente.
- In generale, il *context switch* può costare da qualche centinaio di nanosecondi a qualche microsecondo.
- Questo tempo “sprecato” rappresenta un *overhead* (sovraccarico) per il sistema e influisce sulle sue prestazioni.

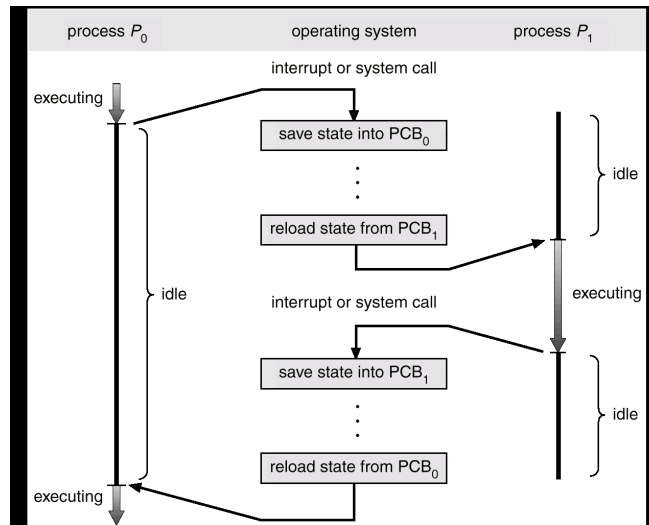
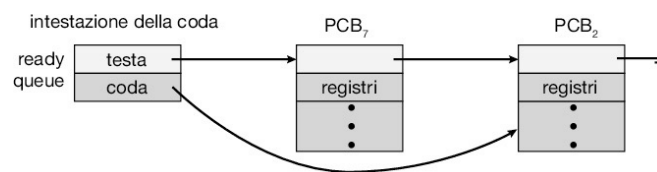


Figure 3.3: Fasi dello scheduling tra un processo e un altro

3.2.2 Code di scheduling

Per **amministrare** la vita di ciascun processo, il SO gestisce varie **code** di processi. Ogni processo “si trova” in una di queste code, a seconda di cosa sta facendo. Una coda di processi non è altro che una lista di PCB, mantenuta in una delle aree di memoria primaria che il SO riserva a se stesso.

La coda dei processi più importante è la coda **ready**, o **ready queue (RQ)**: l'insieme dei processi **ready to run**. Quando un processo rilascia la CPU, ma non termina e non torna nella *ready queue*, vuol dire che si è messo



in **attesa** di “qualcosa”, e il SO lo “parcheggia” in una tra le possibili code, che possiamo dividere in due grandi categorie:

- **Device queues:** code dei processi in attesa per l'uso di un dispositivo di I/O. Una coda per ciascun dispositivo.

Esempio 3.2.1 (Esempi)

- Una coda d'attesa per il primo hard disk
- Una coda per l'ssd
- Una coda per la stampante, etc..

- **Code di waiting:** code di processi in attesa che si verifichi un certo evento. Una coda per ciascun evento (ci torneremo nella sezione 6.6).

Dunque, durante la loro vita, i processi si spostano (meglio: il SO sposta i corrispondenti PCB) tra le varie code. Quindi lo stato **waiting** nel diagramma di transizione degli stati di un processo **corrisponde a più code di attesa**

Possiamo riformulare il diagramma di transizione degli stati di un processo come un **diagramma di accodamento** in cui i processi si muovono fra le varie code

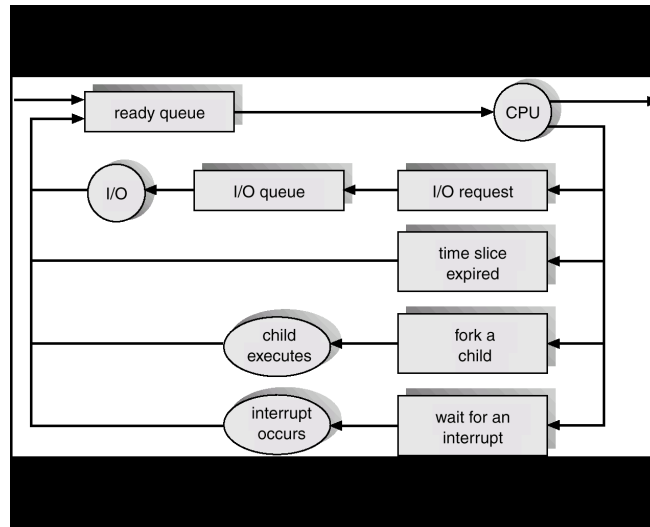


Figure 3.4: SX: new, DX: Terminated

3.2.3 CPU Scheduler

Un componente del Sistema Operativo detto *CPU Scheduler* sceglie uno dei processi nella coda *ready* e lo manda in esecuzione.

- Il *CPU scheduler* si attiva ogni 50/100 millisecondi, ed è responsabile della realizzazione del *time sharing*.
- Per limitare l'*overhead*, deve essere molto veloce.
- Il *CPU scheduler* è anche chiamato *Short Term Scheduler*.

3.3 Operazione sui processi

La creazione di un processo è di gran lunga l'operazione più importante all'interno di qualsiasi sistema operativo. Ogni SO possiede almeno una *System Call* per la creazione di processi, e ogni processo è creato a partire da un altro processo usando la system call relativa (eccetto il processo che nasce all'accensione del sistema).

Il processo "creatore" è detto *processo padre* (o *parent*).

Il processo creato è detto *processo figlio* (o *child*).

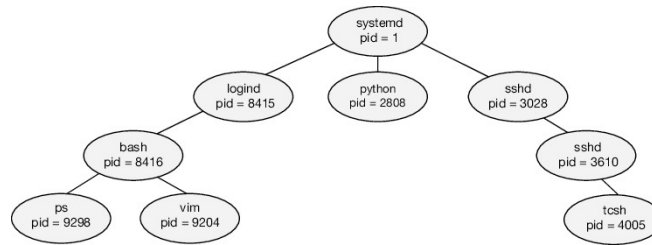
Osservazioni 3.3.1

Poiché ogni processo può a sua volta creare altri processi, nel sistema si forma un "albero di processi".

3.3.1 Creazione di un processo

Quando nasce un nuovo processo, il SO:

- gli assegna un identificatore del processo unico, un numero intero detto **pid** (process-id). È il modo con cui il SO conosce e si riferisce a quel processo.
- recupera dall'hard disk il codice da eseguire e lo carica in RAM (a meno che il codice non sia già in RAM).



- alloca un nuovo *PCB* e lo inizializza con le informazioni relative al nuovo processo.
- inserisce il *PCB* in coda *ready*.

Domanda 3.3

Che cosa fa il processo padre quando ha generato un processo figlio?

- Prosegue la sua esecuzione in modo concorrente all'esecuzione del processo figlio, oppure:
- Si ferma, in attesa del completamento dell'esecuzione del processo figlio

Domanda 3.4

Quale codice esegue il processo figlio?

- al processo figlio viene data una copia del codice e dei dati in uso al processo padre, oppure:
- al processo figlio viene dato un nuovo programma, con eventualmente nuovi dati.

3.3.2 Creazione di un processo in Unix

```

int main() {
    /* fig. 3.8 modificata */
    pid_t pid, childpid;

    pid = fork(); /* genera un nuovo processo */
    printf("questa_la_stampano_padre_e_figlio");

    if (pid == 0) {
        /* processo figlio */
        printf("processo_figlio");
        execlp("/bin/ls", "ls", NULL);
    } else {
        /* processo padre */
        printf("sono_il_padre,_aspetto_il_figlio");
        childpid = wait(NULL);
        printf("il_processo_figlio_terminato");
        exit(0);
    }
}

```

3.3.3 Passi dell'SO all'invocazione delle fork

1. Alloca un nuovo *PCB* per il processo figlio e gli assegna un nuovo *PID*; cerca un'area libera in RAM e vi copia le strutture dati e il codice del processo *parent* (si veda più avanti): queste copie verranno usate dal processo figlio.
2. Inizializza il *PC* del figlio con l'indirizzo della prima istruzione successiva alla *fork*.

3. Nella cella di memoria associata alla variabile che riceve il risultato della *fork* del processo figlio scrive 0.
4. Nella cella di memoria associata alla variabile che riceve il risultato della *fork* del processo *parent* scrive il *PID* del figlio.
5. Mette i processi *parent* e figlio in coda *ready*.

Osservazioni 3.3.2

pid == 0 Lo ha solo il processo figlio.

pid = id_c_hildLOhasoloilprocessopadre.

Cossonoingradodidistingueresestooperandoconilfiglioconilpadre

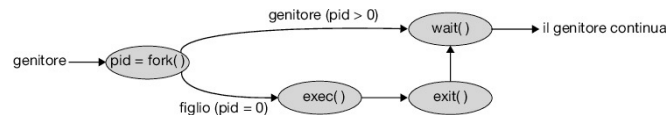
Significato delle altre sys

Execlp: Riceve in input un puntatore ad un file contenente codice eseguibile. Il processo che la invoca prosegue eseguendo il codice specificato, senza più ritornare alla porzione di codice che viene dopo la execlp.

Wait: Invocata da un processo parent, lo sospende fino alla terminazione del processo figlio. La wait restituisce il PID del figlio appena terminato. **Exit:** provoca la terminazione istantanea del processo che la invoca.

Domanda 3.5

Come cambia lo schema se il processo parent non esegue la wait?



3.3.4 Altro esempio

```

int main() {
    /* un altro esempio */
    int a, b, c = 57;
    a = fork(); // genera un nuovo processo
    printf("questa_la_stampano_padre_e_figlio");

    if (a == 0) {
        /* processo figlio */
        c = 64; // ***
        printf("c=%d", c);
    } else {
        /* processo padre */
        printf("c=%d", c);
        b = wait(NULL);
        printf("b=%d", b);
    }
}
  
```

3.3.5 Osservazioni

- Il codice viene condiviso tra padre e figlio, evitando duplicazione e spreco di memoria.
- Lo spazio dati viene duplicato:

- Le modifiche di variabili non sono condivise tra padre e figlio.
- Le nuove variabili dichiarate dopo la **fork** non sono visibili all'altro processo.
- Un padre può chiamare **fork** più volte, e usare il PID dei figli per tracciarli.
- **fork** restituisce 0 al figlio per distinguerlo dal padre.
- Se **fork** restituisse un valore maggiore di 0 al figlio, non si potrebbe distinguere facilmente tra padre e figlio, complicando la gestione delle operazioni diversificate (come illustrato in fig. 3.8).

3.3.6 Terminazione di un processo

Un processo termina dopo l'esecuzione dell'ultima istruzione del suo codice. Esiste una system call chiamata **exit()** per terminare un processo.

I dati di output, come il **pid**, possono essere inviati al processo padre in attesa della terminazione del figlio. Il sistema operativo **rimuove** le risorse allocate al processo terminato, recuperando la RAM e chiudendo eventuali file aperti.

- Un processo può uccidere esplicitamente un altro processo appartenente allo stesso utente tramite la system call **kill** (in Unix) o **TerminateProcess** (in Win32).
- In alcuni casi, il sistema operativo può decidere di terminare un processo utente, ad esempio se:
 - il processo utilizza troppe risorse.
 - il suo processo padre è morto (in questo caso può avvenire una terminazione a cascata, che non avviene però in Unix o Windows).

3.4 Comunicazione tra processi

Processi indipendenti e cooperanti

I processi attivi in un sistema possono essere classificati come:

- **Indipendenti**: quando non si influenzano esplicitamente durante l'esecuzione.
- **Cooperanti**: quando si influenzano a vicenda per:
 - Scambiarsi informazioni.
 - Collaborare su un'elaborazione suddivisa per efficienza o modularità.

I processi cooperanti necessitano di meccanismi di comunicazione e sincronizzazione.

3.5 Esempio: il problema Produttore-Consumatore

Problema del produttore-consumatore

Un classico problema di processi cooperanti è il *problema del produttore-consumatore*:

- Un **processo produttore** produce informazioni che vengono consumate da un **processo consumatore**.
- Le informazioni sono collocate in un buffer di dimensione limitata.
- Un esempio pratico è un **processo compilatore** (produttore) che genera codice assembler.
- Il **processo assembler** (consumatore) traduce il codice assembler in linguaggio macchina.
- L'assembler potrebbe poi diventare un produttore per un modulo che carica in RAM il codice.

```

#define SIZE 10

typedef struct {
    // Definizione della struttura dell'item
    ...
} item;

// Buffer condiviso
item buffer[SIZE]; (shared array)

// Variabili condivise
int in = 0, out = 0;

```

Buffer circolare di SIZE elementi con due puntatori **in** e **out**:

- **in**: indica la prossima posizione libera nel buffer.
- **out**: indica la prossima posizione piena da consumare.
- **Condizione di buffer vuoto**: $in == out$.
- **Condizione di buffer pieno**: $(in + 1) \% SIZE == out$.

Nota: la soluzione utilizza solo **SIZE-1** elementi per evitare conflitti tra la condizione di buffer pieno e vuoto.