



SAPIENZA
UNIVERSITÀ DI ROMA

UNIVERSITÀ "SAPIENZA" DI ROMA
FACOLTÀ DI INFORMATICA

Sistemi Operativi

Appunti integrati con il libro "Modern Operating Systems",
A. S. Tanenbaum

Author
Simone Bianco

21 gennaio 2023

Indice

0	Introduzione	1
1	Introduzione ai Sistemi Operativi	2
1.1	Sistema operativo e Macchina fisica	5
1.2	Servizi del sistema operativo	7
1.2.1	Kernel/User mode e Protezione della memoria	7
1.2.2	System Calls	8
1.2.3	Scheduling e Sincronizzazione	10
1.2.4	Virtualizzazione della memoria	10
2	Gestione dei processi	11
2.1	Stato dei processi e PCB	14
2.2	Creazione dei processi	15
2.3	Terminazione dei processi	20

Capitolo 0

Introduzione

Capitolo 1

Introduzione ai Sistemi Operativi

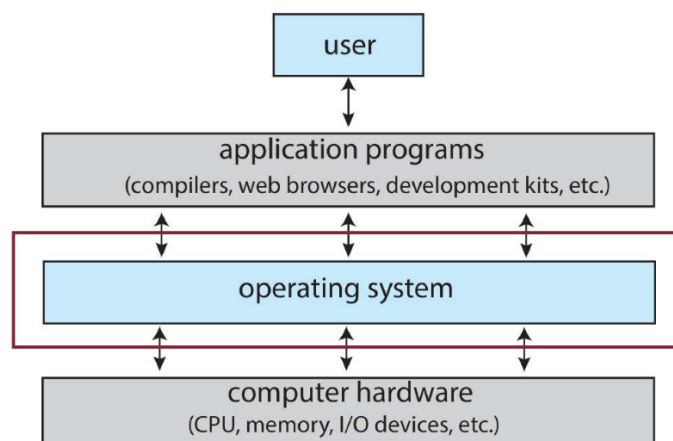
Un **sistema operativo**, abbreviato con OS, è un software di base in grado di gestire le risorse hardware e software della macchina, fornendo servizi di base ai software applicativi.

In particolare, a seconda delle necessità (ad esempio general-purpose, real-time, mobile, ...) e delle scelte di progettazione vengono decisi i compiti e i componenti che costituiscono un sistema operativo. Nonostante ogni OS tenda ad essere diverso dagli altri, ognuno di essi può essere suddiviso in:

- **Kernel**, ossia il "cuore" del sistema operativo, il quale è sempre attivo
- **Programmi di sistema**, ossia ogni altro software facente parte del sistema operativo

Tra i vari compiti svolti da un OS, in particolare troviamo:

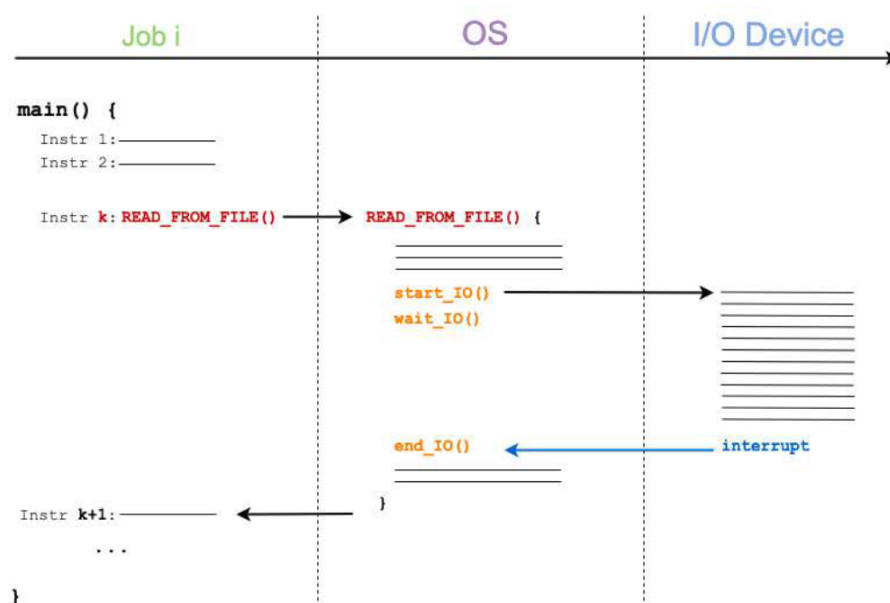
- **Gestione delle risorse fisiche e logiche** utilizzate dai vari software
- **Virtualizzazione delle risorse**, dando l'"illusione" ad ogni software di aver a disposizione "infinite risorse"
- **Interfacciamento tra Hardware e Software**, fornendo un insieme di servizi comuni, ossia delle API, che permettono ai software e agli utenti di interagire con le risorse hardware senza doverle controllare in modo diretto



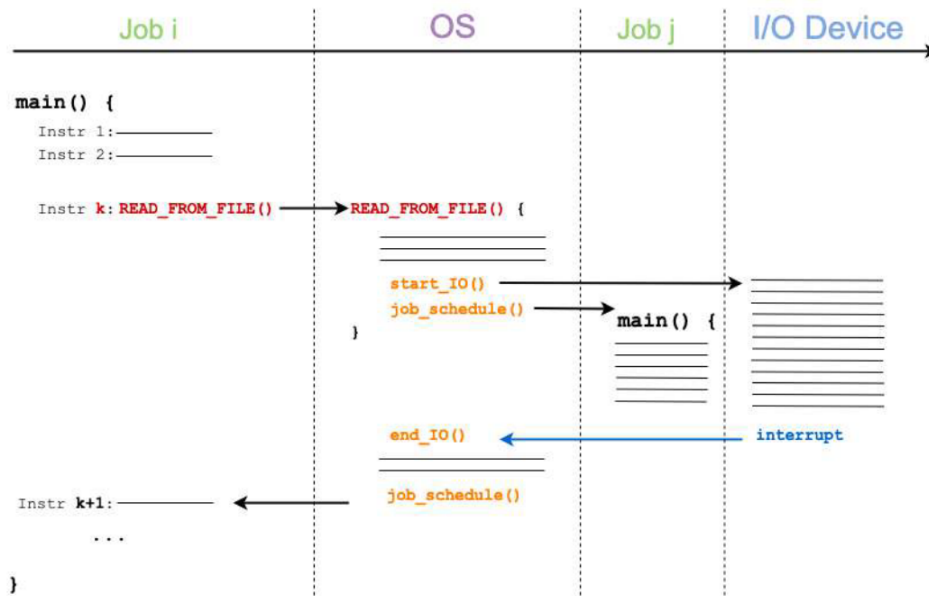
Tipologie di OS

- **Sistemi programmati a mano**, utilizzati da una cerchia ristretta di persone esperte in grado di programmarli manualmente in linguaggio macchina, dunque sprovvisti di sistema operativo agevolante. Il problema principale risultò essere il poco utilizzo e l'eccessivo costo dell'hardware
- **Sistemi con console a singolo utente (Mainframe)**, utilizzati da un utente per volta durante l'esecuzione di un programma, i quali venivano scritti su schede perforate. Ogni operazione veniva eseguita sequenzialmente, prevedendo la presenza di una versione primitiva di OS, ossia un avviatore di programmi, e l'assenza di sovrapposizione tra computazione e interfacciamento con le periferiche di I/O. Per via della sua natura single-user oriented, il problema principale risultò essere l'inefficienza per più utenti.
- **Sistemi di Batch**, in grado di eseguire più "compiti", detti **jobs**, in gruppo, detti **batch**. Ogni utente inseriva un programma tramite schede o nastri, i quali venivano poi periodicamente caricati da un tecnico, per poi essere avviati ed eseguiti da un sistema operativo, risultando in un uso più efficiente dei calcolatori. Tuttavia, ogni job veniva ancora eseguito in modo sequenziale, ossia uno per volta.
- **Sistemi multi-programmabili**, in grado di mantenere caricati in memoria più job contemporaneamente, i quali vengono eseguiti da una CPU in grado di alternare tra di essi. In questi sistemi, l'OS si occupava dell'organizzazione dell'esecuzione dei programmi, detto **job scheduling**, delle operazioni di I/O e della protezione della memoria, impedendo ai vari job di accedere ad aree utilizzate da altri. L'unica problematica di tali sistemi risultò essere la necessità di mantenere la CPU inattiva durante l'esecuzione di operazioni di I/O bloccanti.

Esempio di Sistema Bloccante:

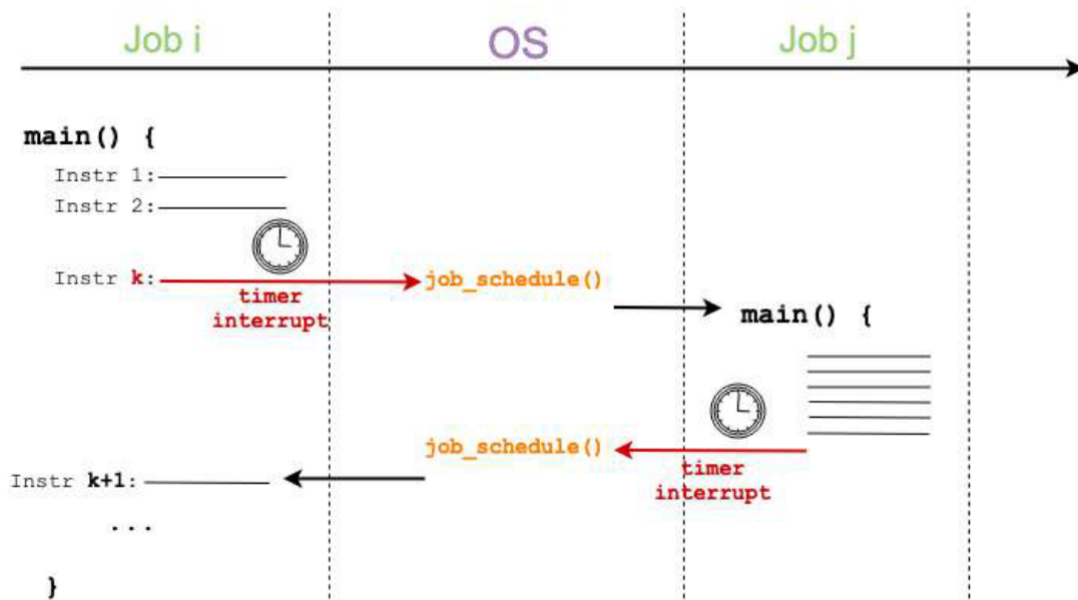


Esempio di Sistema Non-bloccante:



- **Sistemi time-sharing**, dove più utenti utilizzano la stessa CPU tramite console semplici. Viene utilizzato un'interruzione programmata a tempo per alternare la CPU tra i vari job, dando l'illusione di esecuzione parallela (pseudo-parallelismo). In particolare, è famoso il sistema operativo ideato da Ken Thompson e Dennis Ritchie, ossia UNIX, precursore di sistemi operativi come MacOS e le varie distribuzioni di Linux

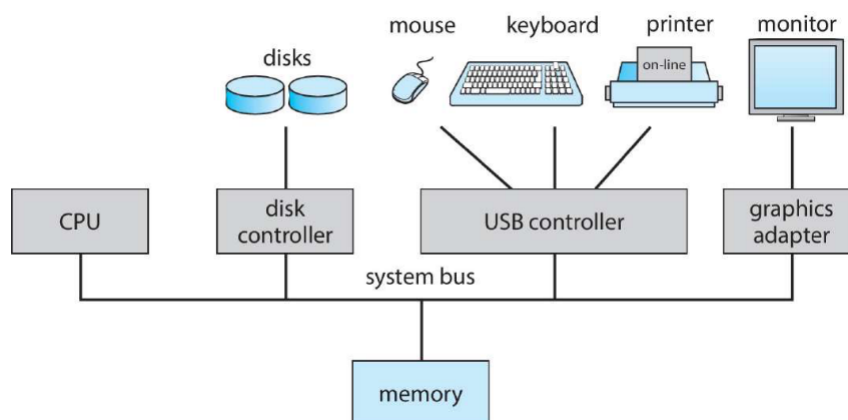
Esempio di Pseudo-parallelismo:



1.1 Sistema operativo e Macchina fisica

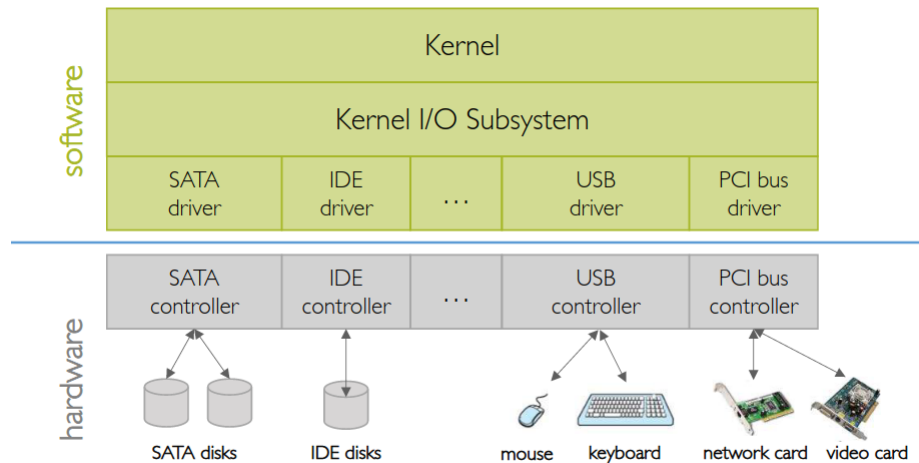
Una moderna architettura è *necessariamente*, composta da:

- **CPU**, il processore che effettua i veri calcoli necessari, operando in base ad un **ciclo di istruzioni**, ossia tre fasi ripetute ciclicamente a velocità elevate:
 1. Fase di **Fetch**: viene prelevata dalla memoria la prossima istruzione da eseguire
 2. Fase di **Decode**: viene interpretata l'istruzione appena recuperata
 3. Fase di **Execute**: viene eseguita l'istruzione appena interpretata
- **Memoria principale**, contiene i dati e le istruzioni utilizzati ed eseguite dalla CPU, la quale può essere immaginata come una sequenza di celle, ognuna composta da una **word**, ossia una sequenza di bit (solitamente 32 o 64), ed ognuna avente un proprio **indirizzo**
- **Dispositivi di I/O**, ossia le periferiche di input ed output (tastiera, mouse, memorie esterne, ...)
- **Bus di Sistema**, ossia un mezzo di comunicazione tra CPU, memoria e dispositivi di I/O, composto a sua volta da:
 - **Data Bus**, il quale trasporta i dati effettivi
 - **Address Bus**, il quale determina dove i dati debbano essere inviati
 - **Control Bus**, il quale indirica quale operazione deve essere effettuata



Le **istruzioni** eseguite dalla CPU vengono descritte tramite il **linguaggio macchina**, rappresentato da sequenze di bit, dove ogni word corrisponde ad una singola istruzione appartenente ad un insieme di istruzioni elementari, le quali interagiscono con i vari **componenti di calcolo**, in particolare l'ALU, e i **registri** della CPU, in grado di immagazzinare temporaneamente una word.

Riguardo ai dispositivi di I/O, è necessario sottolineare come ognuno di essi sia composto da due parti, ossia il **dispositivo fisico** in se e il **device controller**, un insieme di piccoli chip che lo gestisce. Il sistema operativo interagisce con tali dispositivi tramite dei **driver**, ossia dei particolari software adattati per ogni dispositivo che permettono di "tradurre" le richieste dell'OS.



Ogni device controller possiede una serie di registri dedicati con cui comunicare:

- **Registri di stato**, i quali forniscono informazioni alla CPU riguardo il dispositivo
- **Registri di controllo**, i quali vengono utilizzati dalla CPU per configurare e controllare il dispositivo
- **Registri dati**, i quali vengono utilizzati per lo scambio di dati con il dispositivo

Per poter comunicare con i dispositivi di I/O, la CPU deve essere in grado di poter distinguere tra indirizzi della memoria principale ed indirizzi rappresentanti i dispositivi stessi. A tal fine, il control bus è dotato di una **linea speciale** detta **M#IO** in grado di stabilire se la CPU desidera comunicare con la memoria principale o con i dispositivi di I/O.

Quindi, esistono in particolare due modi con cui la CPU può comunicare con i device controller:

- **Port-mapped I/O**, dove i riferimenti ai controller vengono effettuati utilizzando uno spazio di indirizzi aggiuntivo dedicati solo all'accesso ai dispositivi di I/O, necessitando l'implementazione di istruzioni aggiuntive del tipo IN ed OUT che vadano a specificare se lavorare sullo spazio di indirizzi della memoria e dei dispositivi I/O

```
MOV DX, 1234h    //carica nel registro DX il valore 0x1234

MOV AL, [DX]     //carica nel registro DX il valore contenuto
                 //nell'indirizzo 0x1234 della memoria

IN AL, DX        //carica nel register DX il valore letto dal
                 //dispositivo connesso alla porta 0x1234
```

- **Memory-mapped I/O**, dove parte degli indirizzi della memoria viene "sprecato" per poter essere utilizzato per effettuare i riferimenti ai dispositivi I/O, non necessitando tuttavia istruzioni aggiuntive e permettendo alla CPU di trattare i registri dei control device come se fosse dei suoi registri aggiuntivi

1.2 Servizi del sistema operativo

1.2.1 Kernel/User mode e Protezione della memoria

Alcune istruzioni eseguite dalla CPU risultano essere più sensibili di altre, ad esempio l'istruzione HLT, in grado di arrestare il sistema, e l'istruzione INT X, in grado di generare un interrupt di sistema. Affinché tali **istruzioni privilegiate** vengano utilizzate esclusivamente dal sistema operativo, la CPU può essere impostata in due modalità specifiche a seconda del programma in esecuzione.

Definition 1. Kernel mode e User mode

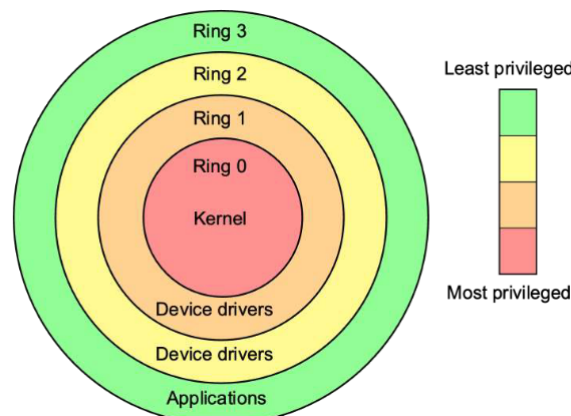
La CPU può essere impostata in:

- **Kernel mode**, ossia in modalità senza alcuna restrizione, permettendo l'esecuzione di qualsiasi istruzione (utilizzata dal sistema operativo)
- **User mode**, dove **non** è possibile:
 - Accedere agli indirizzi riservati ai dispositivi di I/O
 - Manipolare il contenuto della memoria principale
 - Arrestare il sistema
 - Passare alla Kernel Mode
 - ...

Per poter impostare una delle due modalità, viene utilizzato un **bit speciale** salvato in un registro protetto: se impostato su 0 la CPU sarà in Kernel mode, mentre se impostato su 1 la CPU sarà in User mode

Observation 1. Protection rings

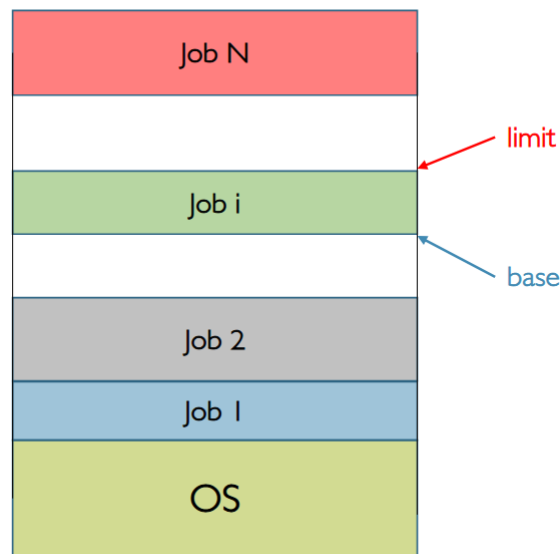
Nelle architetture più moderne, vengono implementati più livelli di protezione, detti **protection rings**. Ogni ring aggiunge restrizioni sulle istruzioni eseguibili dalla CPU.



Oltre alle protezioni imposte sulle istruzioni eseguite, è necessario imporre delle protezioni anche sulla **memoria**, in modo da proteggere reciprocamente gli spazi di memoria riservati ai singoli programmi degli utenti e in modo da proteggere gli spazi di memoria riservati all'OS dai programmi utente.

La tecnica più semplice per ottenere tale protezione è l'utilizzo di due registri dedicati, i quali vengono caricati dall'OS all'avvio di un programma. L'OS, per poi controllare che ogni indirizzo richiesto durante l'esecuzione del programma ricada nell'intervallo specificato

- **Registro base**, contenente l'indirizzo iniziale dello spazio utilizzabile
- **Registro limite**, contenente l'indirizzo finale dello spazio utilizzabile



1.2.2 System Calls

Definition 2. Trap di sistema

Definiamo come **trap di sistema** un qualsiasi evento che richiede il passaggio da Kernel mode ad User mode.

In particolare, individuiamo tre tipi di trap:

- **System calls**, ossia la richiesta di un servizio dell'OS, svolte in modo sincrono e innescate dai software
- **Exception**, ossia la gestione di errori dovuti ad eventi inattesi, svolte in modo sincrono e innescate dai software
- **Interrupt**, ossia il completamento di una richiesta in attesa, svolte in modo asincrono e innescate dall'hardware

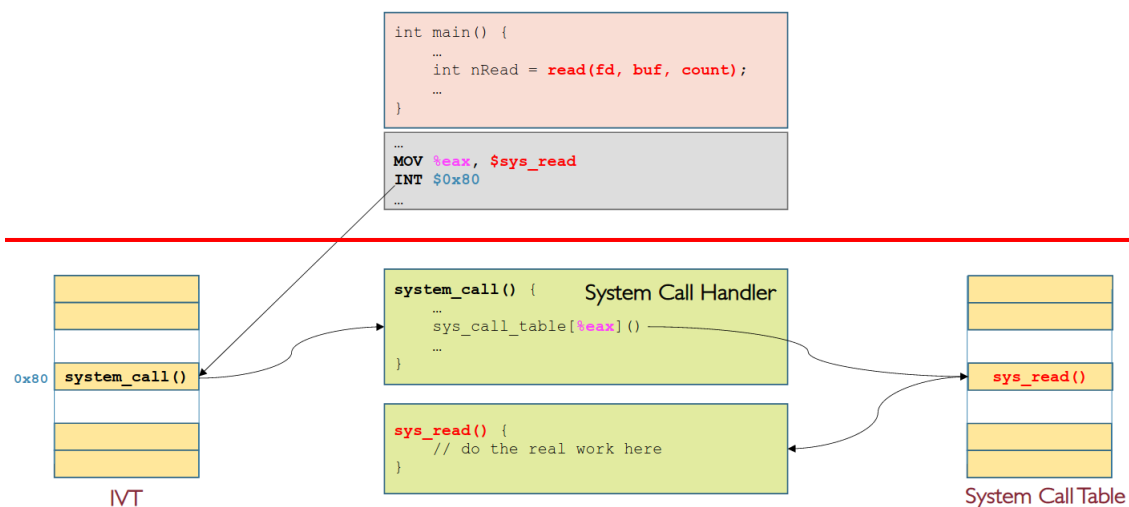
Definition 3. System Calls

Le **System calls** sono delle procedure messe a disposizione dal sistema operativo per permettere agli altri software di poter accedere ai servizi messi a disposizione dal sistema operativo, in particolare l'esecuzione di alcune istruzioni privilegiate (ad esempio l'accesso ad un dispositivo I/O).

Le System call ricadono in **sei categorie**:

- **Process control**, include procedure come end, abort, load, execute, create process, terminate process, get/set process attributes, wait for time or event, signal event, allocate and free memory
- **File management**, include procedure come create file, delete file, open, close, read, write, reposition, get and set file attributes
- **Device management**, include procedure come request device, release device, read, write, reposition, get and set device attributes, logically attach or detach devices
- **Information maintenance**, include procedure come get and set time, get and set date, get and set system data, get and set process, get and set file information, get and set device attributes
- **Communications**, include procedure come create and delete communication connection, send and receive messages, transfer status information, attach or detach remote devices

Quando un programma utente richiede l'esecuzione di una syscall tramite un'API fornita dal sistema operativo, tale richiesta viene prima convertita in linguaggio macchina, per poi venir gestita da **System call Handler**, il quale si occuperà di salvare lo stato precedente dei registri, i quali verranno poi alterati durante l'esecuzione della syscall, per poi essere ripristinati



1.2.3 Scheduling e Sincronizzazione

Per poter gestire lo scheduling delle operazioni, viene implementato un **timer** nell'hardware, il quale genera un interrupt dopo una breve quantità di tempo, permettendo al **CPU scheduler** di prendere il controllo e decidere quale sia la prossima operazione da eseguire, impedendo che la CPU venga monopolizzata da un job "egoista"

Inoltre, poiché le interrupt sono asincrone, esse potrebbero essere innescate in qualsiasi momento, interferendo con i programmi in esecuzione. Il sistema operativo, quindi, deve essere in grado di sincronizzare le operazioni di processi concorrenti e cooperanti. Affinché ciò sia possibile, è necessario che l'hardware si assicuri che piccole sequenze di istruzioni vengano eseguite in modo **atomico**, disabilitando le interruzioni durante la loro esecuzione

1.2.4 Virtualizzazione della memoria

La **virtualizzazione della memoria** è un'astrazione logica della memoria fisica, dando ad ogni processo l'illusione di una memoria fisica strutturata come uno spazio di indirizzi contiguo, permettendo inoltre l'esecuzione di programmi senza la necessità che essi siano completamente caricati nella vera memoria principale (restando ovviamente caricati completamente nella memoria virtuale).

Può essere implementata sia tramite hardware, in particolare tramite una Memory Management Unit (MMU), sia tramite software, dove l'OS stesso si occupa di tradurre gli indirizzi virtuali in quelli fisici.

Capitolo 2

Gestione dei processi

Definition 4. Programmi e Processi

Un **programma** è un file eseguibile salvato in memoria secondaria. Contiene l'insieme di istruzioni necessarie a svolgere un compito richiesto.

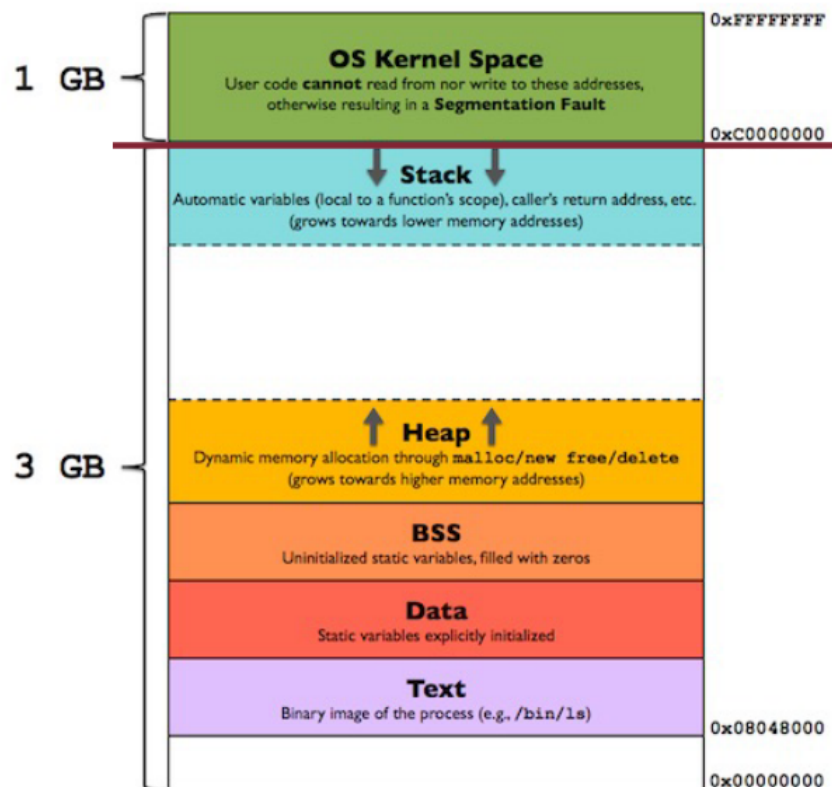
Un **processo** è una particolare istanza attiva di un programma caricata in memoria principale, eseguendo sequenzialmente le istruzioni descritte nel programma stesso. Più processi potrebbero corrispondere allo stesso programma, tuttavia ognuno di essi possiede un proprio stato attuale.

Observation 2. Virtual address space

Il sistema operativo fornisce la stessa quantità di **virtual address space** ad ogni processo, la quale è suddivisa in più sezioni:

- **Text**, contenente le istruzioni dell'eseguibile
- **Data**, contenente i dati statici già inizializzati all'avvio
- **BSS**, contenente i dati statici non inizializzati all'avvio
- **Stack**, contenente tutti i dati utilizzati da ogni funzione, ossia i vari **stack frame**
- **Heap**, utilizzato per allocazioni dinamiche, ossia di dimensione non fissa
- **Spazio libero**, interposto tra stack ed heap, utilizzato da entrambi per "espandersi"

Di seguito vi è un esempio di suddivisione del virtual address space di un'architettura a 32bit:



Esempio:

- Consideriamo il seguente programma:

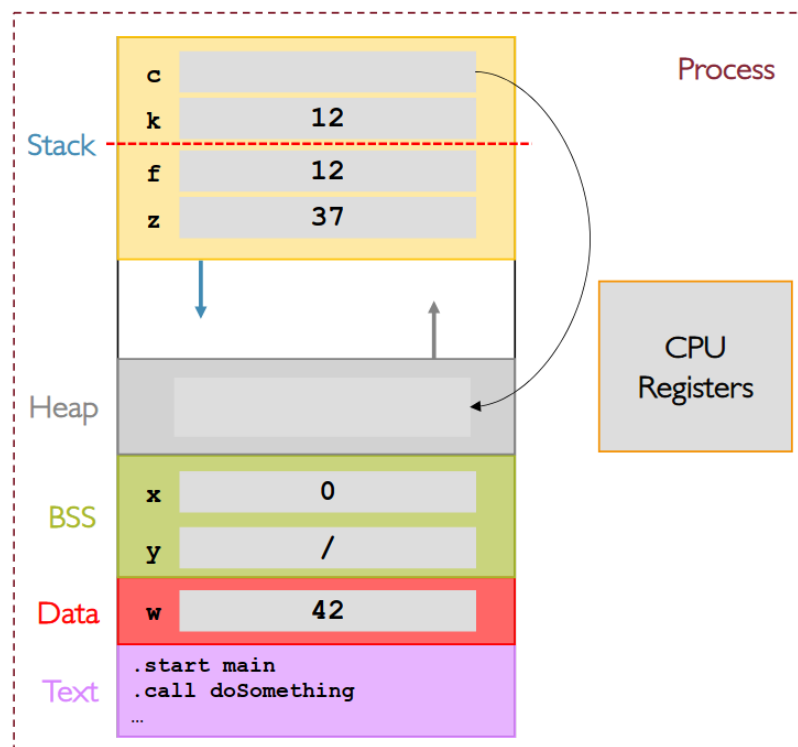
```
int w = 42;
int x = 0;
float y;

void doSomething(int f) {
    int z = 37;
    z += f;
    ...
}

int main() {
    char* c = malloc(128);
    int k = 12;
    doSomething(k);
    ...
}
```

- La variabile `w` sarà salvata all'interno della sezione `Data`, poiché essa è statica ed è già definita all'avvio

- Le variabili `x` e `y` verranno salvate all'interno della sezione BSS, poiché entrambe statiche ma non definite all'avvio (di default, il valore di una variabile di tipo `int` è già 0, dunque l'assegnamento iniziale inserito verrebbe semplicemente ignorato dal compilatore)
- Le istruzioni presenti nelle funzioni `main()` e `doSomething()` vengono inserite nella sezione Text
- Gli stack frame delle funzioni `main()` e `doSomething()` vengono salvati nello Stack.
- In particolare, la variabile `char* c` è un puntatore salvato nello Stack facente riferimento ad una zona dell'Heap di 128 bit, corrispondente quindi a 16 caratteri.

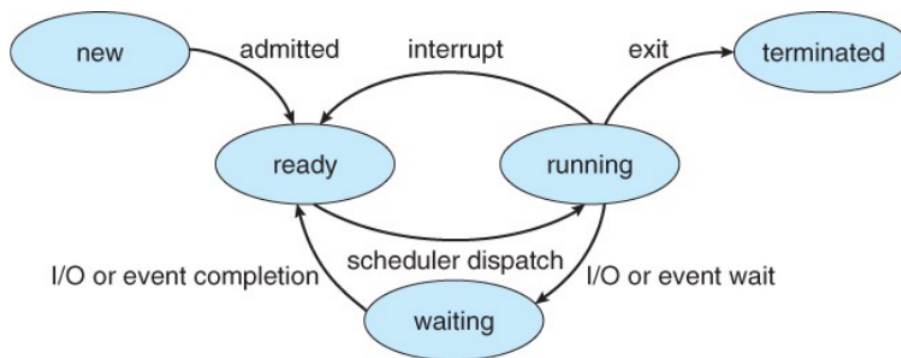


2.1 Stato dei processi e PCB

Definition 5. Stato di un processo

In qualsiasi momento, uno processo può assumere uno dei seguenti stati:

- **New**, ossia il processo è appena stato avviato
- **Ready**, il processo è pronto ad essere eseguito ma è in attesa di essere schedato nella CPU
- **Running**, le istruzioni del processo stanno venendo eseguite dalla CPU
- **Waiting**, il processo viene sospeso in attesa che una risorsa necessaria sia disponibile o che un evento sia completato o avvenga (ad esempio input da tastiera, accesso al disco, ...)
- **Terminated**, il processo ha completato la sua esecuzione e l'OS può eliminarlo



Observation 3

La maggior parte delle syscalls è **bloccante**:

- Dal lato **user space**, il processo chiamante non può svolgere operazioni finché la syscall termina la richiesta
- Dal lato **kernel space**, l'OS imposta il processo corrente nello stato **waiting** e schedula il primo processo in stato di **ready** presente nella **state queue**
- Una volta che la syscall termina, il precedente processo bloccato passa in stato di **ready**, rimanendo in attesa di essere schedato nuovamente

NB: solo il processo che ha effettuato la chiamata viene bloccato, non l'intero sistema operativo

Definition 6. Process Control Block (PCB)

Il **Process Control Block (PCB)** è la struttura dati principale utilizzata dall'OS per tenere traccia dello stato e della posizione in memoria di un processo. All'avvio di un processo, l'OS alloca un nuovo PCB, aggiungendolo alla state queue, per poi de-allocarlo una volta che il processo associato è terminato

Ogni PCB è composto da:

- **Process state**, contenente lo stato del processo
- **Process number**, contenente un'identificatore univoco per il processo
- **Program counter, Stack Pointer e registri vari** relativi al processo in esecuzione
- **Informazioni per lo scheduling**, contenente l'indice di priorità e il puntatore alla state queue
- **Informazioni per la gestione della memoria e contabilità**, contenenti le page table relative al processo e il tempo in user e kernel mode del processo
- **Status I/O**, contenente la lista dei file aperti dal processo

2.2 Creazione dei processi

Definition 7. Processo padre e figlio

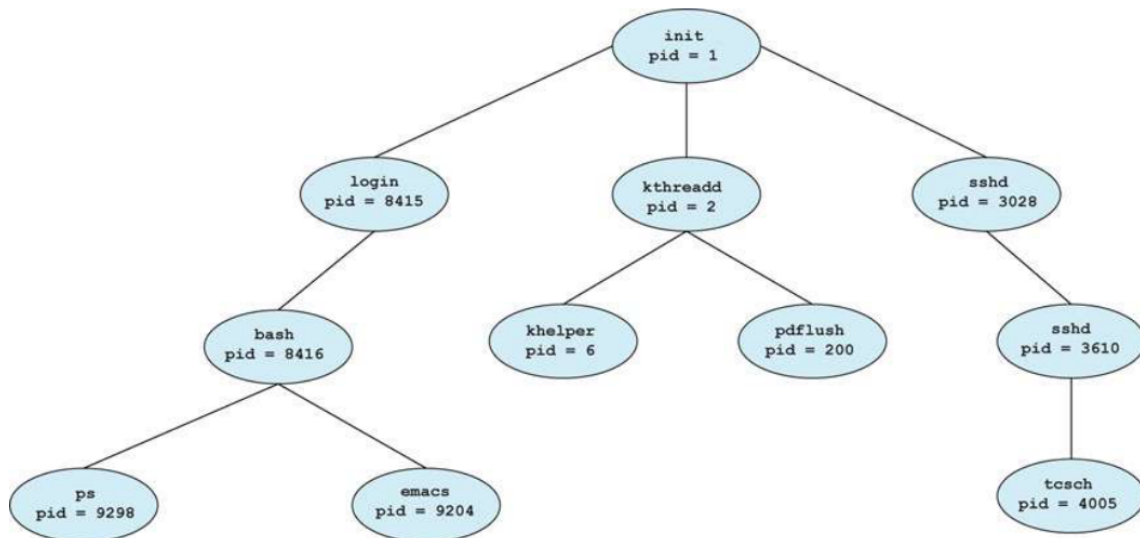
Un processo può creare altri processi tramite specifiche syscalls. Il processo creatore viene detto **processo padre**, mentre il processo creato viene detto **processo figlio**.

Ad ogni processo è associato un **process identifier (PID)**, che lo identifica univocamente, ed un **parent process identifier (PPID)**, che identifica univocamente il suo processo padre.

Definition 8. Processo sched e init

In un tipico sistema UNIX, il processo scheduler è chiamato **sched** e il suo PID è 0. La prima operazione eseguita dal processo sched all'avvio del sistema operativo è avviare il processo **init**, avente PID pari a 1. Il processo **init** si occupa di avviare tutti i **processi daemon**, ossia processi eseguiti in background, e i processi relativi ai login degli utenti, diventando quindi l'antenato di tutti i processi che verranno avviati in seguito.

Esempio:



Method 1. Fork e Spawn

Per poter creare i processi figli relativi ad un parente vengono utilizzate due syscall:

- **fork()**, dove il figlio creato è una copia esatta del padre, condividendo con esso le stesse risorse ed ognuno avente il proprio PCB
- **exec()**, dove il figlio creato è un processo legato ad un programma diverso da quello del padre e avente uno spazio d'indirizzamento diverso, dunque con istruzioni e dati diversi dal padre

Observation 4

Poiché utilizzando la syscall **fork()** i due processi condividono lo stesso codice e le stesse risorse, nel processo **padre** essa **ritornerà un PID maggiore di 0**, corrispondente al PID effettivo del figlio, mentre nel processo **figlio**, essa **ritornerà 0 come PID**, corrispondente al PID di sched

Observation 5

Una volta creato un processo figlio, il processo padre ha due opzioni:

- Attendere che il processo figlio termini l'esecuzione utilizzando la syscall **wait()**
- Continuare la sua esecuzione in parallelo con il figlio senza essere bloccato

Esempi:

1. • Consideriamo il seguente codice:

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main(){
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) {
        /* if the returned PID is < 0, an error occurred */

        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) {
        /* execute child process code */

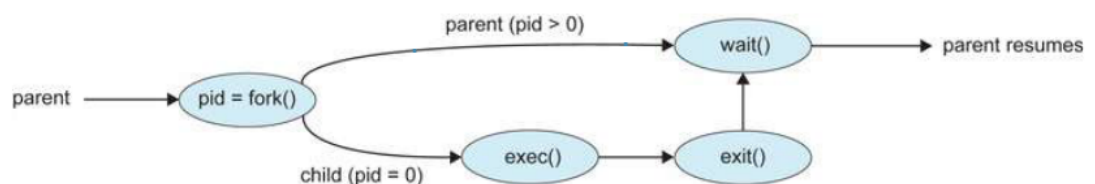
        execlp("/bin/ls", "ls", NULL);
    }
    else {
        /* execute parent process code */

        ...

        /* wait for child to terminate */
        wait(NULL)
        printf("Child terminated");

        exit(0);
    }
}
```

- L'albero decisionale del programma si sviluppa come tale:



2. • Consideriamo il seguente codice:

```
int pid = fork();

if(pid == 0) {      // A's child (B)
    pid = fork();

    if(pid == 0) {  // B's child (C)
        ...
        execlp(...);
    }
    else { // B
        ...
    }
}
else { // A
    ...
}
```

- La gerarchia dei processi generata corrisponde a:



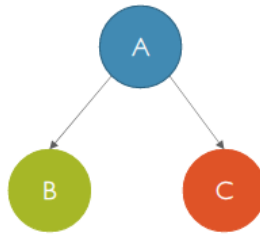
3. • Consideriamo il seguente codice:

```
int pid = fork();

if(pid == 0) {      // A's child (B)
    ...
    execlp(...);
}
else { // A
    pid = fork();

    if(pid == 0) {  // A's child (C)
        ...
        execlp(...);
    }
}
```

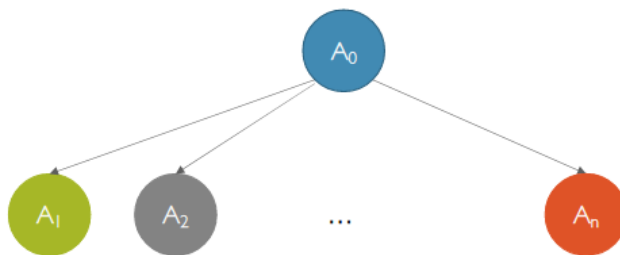
- La gerarchia dei processi generata corrisponde a:



4. • Consideriamo il seguente codice:

```
for(int i=0;i<n;i++) {  
    if(fork() == 0) {        // A0's child  
        ...  
        execlp(...);  
    }  
}  
  
// back in the parent A0  
// wait for all children to terminate  
for(int i=0;i<n;i++) {  
    wait(NULL);  
}
```

- La gerarchia dei processi generata corrisponde a:



2.3 Terminazione dei processi

Method 2. Exit

Ogni processo può richiedere la sua **terminazione immediata** eseguendo la syscall **exit()**, la quale, tipicamente, ritorna un intero. L'intero ritornato viene passato al processo padre nel caso in cui esso stia eseguendo la syscall **wait()**.

Solitamente, viene ritornato 0 se l'esecuzione viene completata senza problemi, altrimenti viene ritornato un valore diverso da 0.

Observation 6

I processi possono anche essere terminati immediatamente dal sistema operativo stesso (ad esempio nel caso in cui sia necessario liberare risorse o nel caso in cui venga eseguito un **comando kill**).

Inoltre, un padre potrebbe uccidere un suo figlio se il compito ad esso assegnato non è più necessario.

Se un processo padre esegue la syscall **exit()** e un suo figlio non è ancora terminato, quest'ultimo viene detto **processo orfano**, venendo solitamente ereditati da **init**, il quale poi si occuperà di ucciderli tramite **kill**.

Observation 7

Quando un processo termina, tutte le sue risorse di sistema vengono liberate (i dati vengono de-allocate, i file aperti vengono chiusi, ...).

Lo stato di processo terminato e il tempo della durata dell'esecuzione viene ritornata al processo padre nel caso in cui esso stia eseguendo la syscall **wait()**

Se un processo figlio tenta di terminare ma il suo processo padre non sta eseguendo la syscall **wait()**, allora tale processo viene detto **processo zombie**, il quale eventualmente verrà ereditato da **init** per poi essere ucciso.