Sistemi Operativi 1

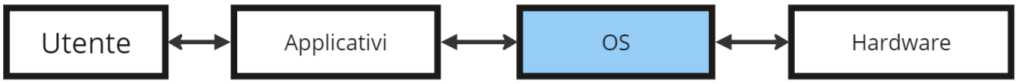
Marco Casu

1 Introduzione

Non esiste una definizione universalmente riconosciuta di sistema operativo, ma una definizione accurata può essere :

> Un sistema operativo, è un implementazione di una macchina virtuale, più facile da programmare rispetto che, lavorando direttamente sull'hardware.

Il sistema operativo (che durante il corso denomineremo come "*OS*"), si interfaccia, o interpone fra l'hardware ed i programmi ed applicazioni di sistema.



Per progettare un OS bisogna avere delle premesse funzionali per capire cosa includere o no dentro tale sistema, esistono macchine diverse, con scopi ed esigenze diverse, durante lo svolgimento di tale corso si tratteranno sistemi operativi per macchine a scopo *generico*.
Un OS è composto da 2 ingredienti :

- **Kernel** - Il nucleo del sistema, costantemente in esecuzione.
- **Programmi di Sistema** - Tutto ciò che non è il nucleo, ossia i programmi che lo circondano.

Non esiste un sistema operativo adatto a qualsiasi circostanza, è sempre necessario scendere a compromessi (concetto di **trade off**) per soddisfare i requisiti necessari. In una macchina, un OS svolge diversi ruoli, il primo è quello di **arbitro**, ossia, rendere equa ed efficente la gestione delle risorse fisiche a disposizione. Un altro ruolo è quello di **illusionista**, ossia servirsi della *virtualizzazione* per dare la parvenza all'utente che le risorse a disposizione siano infinite. Un ultimo ruolo è quello di **collante**, cioè interporsi fra software ed hardware per permettergli di comunicare, facendo interagire gli utenti con il sistema piuttosto che con la macchina direttamente. La componente principale che gestisce un OS è la CPU, la memoria ed i dispositivi di Input ed Output (che durante il corso denomineremo come "*I/O*"). Un componente da tenere in considerazione è il *bus di sistema*, ossia il mezzo di comunicazione fra queste entità, tale bus è suddiviso in :

- DATA BUS - trasporta i dati effettivi sulla quale si sta operando.
- ADRESS BUS - trasporta l'informazione sull'indirizzo dell'istruzione da eseguire.
- CONTROL BUS - trasporta l'informazione sul tipo di operazione da eseguire.

I dispositivi di I/O sono composti da i dispositivi fisici in se ed i loro **device controller**, che ne gestiscono la logica interfacciandoli con l'OS tramite i rispettivi *driver*, riservando ad essi dei registri per determinarne ed immagazzinarne lo stato e la configurazione, per leggere e scrivere dati da essi. Per non fare confusione sul bus quando bisogna comunicare con i dispositivi di I/O, sul bus è previsto uno switch fisico (M/#IO) che indica se si vuole comunicare con la memoria o con i device controller. A tal proposito, le CPU ha 2 modi per comunicare con questi ultimi :

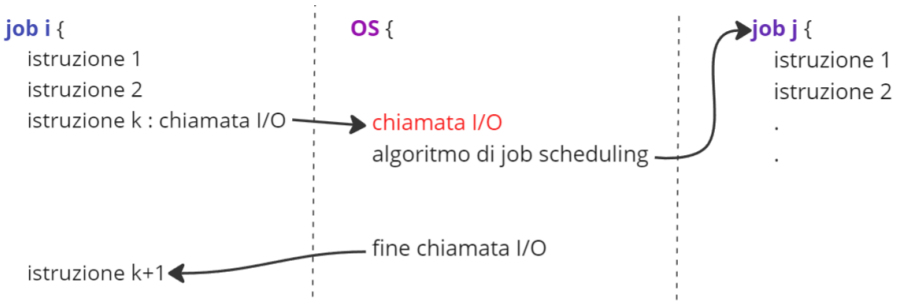
- **port mapped** - I registri dei device controller usano uno spazio di indirizzamento separato dalla memoria principale, ma è necessario estendere l'insieme delle istruzioni elementari del linguaggio macchina per per poter comunicare con questo nuovo spazio.

- **memory mapped** - I registri dei device controller vengono mappati sugli stessi indirizzi riservati alla memoria principale, tale mappatura avviene all'avvio del sistema, non è quindi necessario prevedere nuove istruzioni.

Direct Memory Access Controller
La CPU controlla *periodicamente* lo stato delle richieste di lettura/scrittura. Un altro modo possibile per gestirle è quello delle **interruzioni**, ossia ogni qual volta che la CPU ha richiesto un operazione di I/O, quando questa viene completata viene inviato un segnale dal device controller. Tale segnale, insieme al resto delle comunicazioni fra OS e device controller, avviene su un mezzo di comunicazione dedicato chiamato **DMA** (*Direct Memory Access Controller*), ed il suo scopo è quello di occuparsi di trasferire dati dalla memoria ai dispositivi di I/O, evitando di delegare tale compito alla CPU, soprattutto quando la quantità dei dati da trasferire è considerevole.

1.1 Job scheduling e Time Sharing

I sistemi operativi moderni devono eseguire contemporaneamente un'ampio numero di programmi ed applicazioni (pagine web, editor di testo ecc...). Se in passato i sistemi operativi risiedevano in un ambiente **uniprogrammato**, ossia che nella memoria era salvato un solo programma che veniva eseguito, adesso gli OS moderni godono di un ambiente **multiprogrammato**, dove vengono mantenuti più processi che vengono caricati in memoria. Ciascun processo ha determinate istruzioni (jobs) che vengono caricati in memoria, il sistema operativo, come è di facile intuizione, è costantemente caricato in memoria.
Come organizzare l'esecuzione di più processi? Essi vengono salvati in memoria, se un processo richiede dei dati tramite un operazione di I/O, esso viene sospeso finché la richiesta non verrà terminata, nel mentre la CPU può "portarsi avanti" il lavoro eseguendo altri processi nel mentre. Usiamo il termine **chiamata bloccante** per indicare una chiamata fatta da un processo che, finchè non è terminata, impedisce al processo di essere eseguito. Quando si ha un considerevole numero di chiamate bloccanti si rischia di rallentare troppo l'esecuzione dei programmi, qui agisce lo **scheduler**, ossia un programma di sistema che implementa un algoritmo allo scopo di decidere quale processo deve essere eseguito dalla CPU nel momento in cui un altro processo in esecuzione viene arrestato da una chiamata bloccante.



Assciurando un buon bilanciamento fra processi in esecuzione e chiamate I/O, tramite il job scheduling la CPU non si arresterà mai ed avrà sempre un processo in esecuzione. Sorge però un altro problema, nel caso dovessimo avere un processo considerevolmente lungo, esso verrà eseguito per un *tempo indeterminato* lasciando la CPU sempre occupata, si utilizza quindi un sistema di **time sharing**, in cui ad ogni processo è riservato un **tempo limitato**, in modo tale che esso se troppo lungo, viene momentaneamente arrestato per lasciare spazio ad altri processi (il tempo limite per ogni processo è stabilito dal sistema), essendo tali tempi molto

bassi per la nostra percezione, tramite il time sharing l'utente avrà un'illusione di *parallelismo* (solo apparente dato che una CPU può eseguire un solo processo alla volta).
È importante considerare che sospendere un processo per mandarne in esecuzione un altro (**content switch**) ha un suo costo, in quanto bisogna salvare lo *stato* del processo precedente salvando i valori contenuti nei registri e l'ultima istruzione da eseguire in modo da poterlo poi *ripristinare* correttamente, per cui il tempo limitato dal sistema secondo il time sharing non deve essere troppo piccolo altrimenti si rischia di fare content switch troppe volte rispetto agli effettivi calcoli da eseguire.

2 Architettura Necessaria per i Servizi dell'OS

2.1 User e Kernel Mode

Per mantenere un certo livello di sicurezza, all'interno dell'OS è possibile eseguire istruzioni in due modalità differenti, **user mode** e **kernel mode**. Alcune istruzioni sono più *sensibili*, se spostare il contenuto da un registro ad un altro (MOV) può essere fatto senza problemi, alcune istruzioni di interruzione dovrebbero richiedere un accesso privilegiato, per questo si vuole implementare la *kernel mode*, che ha la possibilità di eseguire qualsiasi istruzione. Fisicamente, si implementa un *bit* che descrive appunto, tramite i suoi 2 stati, se si sta operando da utente, o in modalità kernel. Il sistema, in user mode, non può interagire direttamente con l'I/O, e non può manipolare il contenuto della memoria. Se l'utente necessita di eseguire operazioni in cui è necessaria la kernel mode, utilizzia le **system call**, delle chiamate, che permettono la momentanea transazione in kernel mode per soddisfare la richiesta, per poi ritornare alla modalità utente.

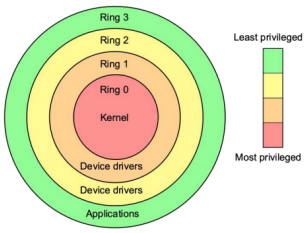


Figure 1: Protection Rings

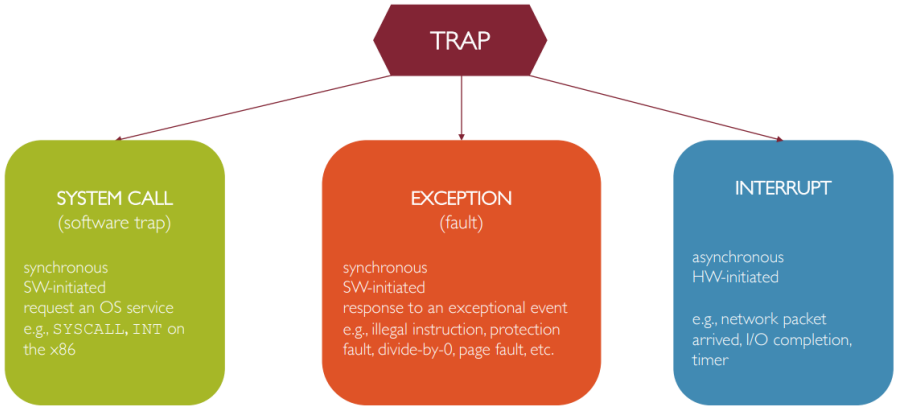
Al minimo è necessario un *bit* per i 2 stati, ma è possibile implementarne molteplici se si vogliono definire dei **protection rings**, ossia più stati di privilegi definiti a strati dove al livello 0 c'è il kernel, e gradualmente si hanno meno privilegi più il livello è alto.

È anche necessario proteggere la memoria, limitanto ogni processo senza dargli la possibilità di poter operare su tutta la memoria disponibile, a livello hardware si implementano due ulteriori registri, denominati **base** e **limit**, quando un processo è in corso, ad esso verrà riservata una limitata partizione di memoria, in *base* sarà contenuto l'indirizzo iniziale dalla quale parte la memoria disponibile, ed in *limit* quello finale, in modo da fornire un *range* di memoria utilizzabile. I valori di tali registri si aggiornano ad ogni *content switch*, dato che ad ogni processo è assegnato il suo range [base,limit].

2.2 Le System Call

Come abbiamo già accennato, l'utente non può interagire direttamente con le istruzioni privilegiate, esistono appositamente le **system call** (o chiamate di sistema), esse richiedono al sistema operativo di eseguire determinate operazioni (come scrivere dati su un disco o inviare dati ad un

interfaccia di rete), quindi esiste una lista di operazioni che l'utente può effettuare tramite esse, sono praticamente l'interfaccia tra l'utente ed il sistema operativo. Tali richieste sono definite **trap**, ossia eventi che causano lo switch da user a kernel mode, tali *trap* non sono esclusivamente le system call, ma anche le **eccezioni** (errori generati dal software, privilegi assenti per un istruzione o tentata divisione per 0), e le **interruzioni** (errori generati dall'hardware, come la scadenza del tempo prefissato per un job tramite il timer).

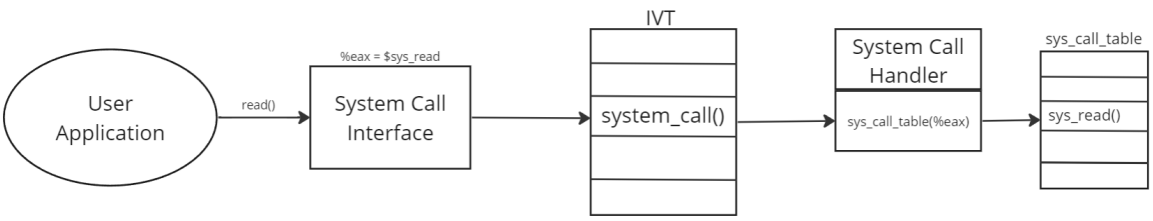


Ci sono 6 categorie principali di system call :

- **Controllo dei processi**
- **Gestione dei file**
- **Controllo dei dispositivi**
- **Manutenzione delle informazioni**
- **Comunicazion tra processi** - per far comunicare due processi, o si utilizza un canale unico di comunicazione tra i due (*message passing*), oppure si fornisce un area di memoria condivisa tra due processi, che andrà poi liberata una volta finita la comunicazione(*shared memory*).
- **Protezione** - forniscono agli utenti accesso temporaneo e limitato ai permessi.

2.3 API per le System Call

Usando le **API** (Application Programming Interface) al posto delle system call direttamente, è possibile fornite maggiore portatilità, e rendere un programma che necessita delle chiamate di sistema indipendente dall'hardware. Un API fornisce una libreria per un linguaggio di programmazione di alto livello (come il *C*), che ci permette di chiamare funzioni che eseguiranno delle chiamate di sistema.



Quando un utente chiama una funzione che richiede una syscall, viene generata un'interruzione, poi in un registro apposito (nell'immagine sovrastante "*%eax*") viene salvato il codice di quella specifica chiamata, il segnale viene poi mandato alla **IVT** (Interrupt Vector Table), ossia un vettore all'interno del kernel che assegna ad ogni interruzione una casua. Quindi l'IVT, capisce che si tratta di una system call e manda il segnale al *System Call Handler*, che si occupa di leggere il contenuto del registro prima citato ("*%eax*"), ed in base al codice, richiamare dalla *System Call Table* la chiamata corretta.

Spesso, i parametri da passare non si limitano al codice identificativo della system call. Esistono 3 diversi modi di passare parametri al sistema operativo :

- Salvare parametri in dei **registri** (ma potrebbero esistere più parametri che registri).
- Salvare i parametri in **blocchi** o "tavole" in un'area di memoria dedicata, passando come parametro nei registri l'indirizzo di tali blocchi.
- Passare i parametri inserendoli (*push*) in uno **stack** dal programma, per poi farli riprendere dallo stack (*pop*) direttamente dal sistema operativo.

Il metodo migliore risulta quello dei *blocchi* o dello *stack*, dato che non hanno limiti sulla quantità di parametri che si possono possibilmente passare.

Le chiamate di I/O eseguite dalle system call possono essere **bloccanti** o **non bloccanti**, le chiamate bloccanti, interrompono il flusso del processo, lasciandolo in "stallo" finchè non si riceveranno i dati richiesti dalla chiamata. Le chiamate non bloccanti invece, richiedono dati tramite le chiamate senza però interrompere il processo, sono quindi più difficili da implementare in quanto il programmatore deve considerare che dopo la chiamata, i dati richiesti potrebbero non essere da subito disponibili.

Come si è accennato precedentemente, risulta utile a livello fisico implementare un **timer** che segna semplicemente l'orario del giorno corrente (detto *time stamp*), esso è utile allo *scheduler*1.1, ad esempio, può generare un interruzione ogni 100 *microsecondi*, in modo che lo scheduler possa prendere il sopravvento sul processo per poi decidere quale altro job va eseguito.

Alcune istruzioni sono dette **atomiche**, ossia, non possono essere fermate dalle interruzioni, le architetture che implementano tali sitruzioni devono far si che esse vengano eseguite per intero, piuttosto, vengono totalmente abortite. Per eseguirle è possibile definirle nel linguaggio macchina come istruzioni speciali che sono nativamente eseguite in maniera atomica, oppure, è possibile *disabilitare* momentaneamente tutte le interruzioni.

2.4 La Memoria Virtuale

La **memoria virtuale** è un *astrazione* della memoria fisica, da l'illusione ad un processo di avere illimitato spazio di memoria per lavorare, e consente ad esso di non essere totalmente caricato in memoria, caricandolo appunto nella memoria virtuale. Tale memoria è implementata sia a livello hardware(MMU) che software(OS) :

- **MMU** - è il componente che si occupa di tradurre gli indirizzi virtuali in indirizzi fisici.
- **OS** - è responsabile di gestire lo spazio degli indirizzi virtuali.

Un sistema a 64 *bit*, è capace di indirzzare 2^{64} *bytes*, gli indirizzi virtuali sono suddivisi in blocchi della stessa dimensione, chiamati **pagine**, le pagine che non vengono caricate nella memoria principale, vengono salvate sul disco. Facendo ciò si fornisce ad un processo una quantità n di indirizzi virtuali, che sono salvati sia in memoria che su disco, essi vengono mappati tramite quella che si chiama **page table**, e si utilizza anche una cache chiamata **TLB** (Translation Look-aside Buffer), che salva i recenti "indirizzamenti" per potervi accedere più rapidamente. L'OS deve considerare quali pagine sono salvate sul disco, e quali sulla memoria principale.

3 Design ed Implementazione di un Sistema Operativo

La struttura interna di un sistema operativo può variare largamente in base alle necessità di utilizzo di tale sistema, è necessario separare quelle che sono le **politiche** del sistema (Le funzioni che deve svolgere) dal suo **meccanismo** (La possibile implementazione di tali funzioni). Tale distinzione ci permette di rendere l'OS più flessibile alle modifiche, riusabile per implementare nuove politiche, e stabile. I primi sistemi operativi erano totalemnte implementati in linguaggio macchina, ciò consentiva ad essi di essere molto efficenti, di contro però, erano limitati esclusivamente all'hardware sulla quale erano scritti. I sistemi operativi odierni hanno esclusivamente una piccola porzione scritta in linguaggio macchina, il corpo principale è scritto in *C*, ed i programmi di sistema possono essere scritti in *C++*, ed altri linguaggi di scripting come *Python*. Un OS dovrebbe essere partizionato in sotto-sistemi, ognungo con compiti ben definiti. Esistono varie strutture di sistema operativo :

- **Simple Structured** - Un sistema non modulare, nella quale non esiste distinzione tra user e kernel. Risulta facile da implementare, ma pecca di rigidità e sicurezza. Un esempio di un OS che adopera tale struttura è *MS-DOS*.
- **Kernel Monolitico** - Un sistema strutturato in modo che sia tutto un grande ed unico processo, con tutti i servizi che vivono nello stesso spazio di indirizzamento. Risulta efficente, ma essendo un unico processo non ci sono limiti di visibilità tra diverse componenti, risulta quindi poco sicuro. Un esempio di un OS che adopera tale struttura è *UNIX*.
- **Layered Structured** - Un sistema diviso in n strati, dove il livello 0 rappresenta l'hardware, ed ogni livello k implenta delle funzionalità che potranno essere riutilizzate dal livello $k+1$ per implementare nuovi programmi. Essendo modulare, risulta portatile, ma bisogna implementare dei canali di comunicazione fra i vari strati.
- **Micro Kernel** - È l'opposto del *Kernel Monolitico*. Nel kernel si inseriscono esclusivamente le funzionalità di base, tutto il resto sarà gestito dalle applicazioni a livello utente. Risulta sicuro ed estendibile, ma pecca nella comunicazione.
- **Loadable Kernel** - Ogni componente è separata, l'approccio risulta simile ai linguaggi di programmazione *object-oriented*. I moduli vengono caricati separatamente all'interno del kernel. Ogni componente comunica con le altre tramite un interfaccia, è simile al *Layered Structured*, ma più flessibile.

È importante in base alla struttura utilizzati, provvedere al giusto hardware da implementare. I sistemi moderni utilizzano per lo più approcci ibridi.

4 La Gestione dei Processi

Definiamo la differenza tra **programma** e **processo** :

- Programma - rappresenta l'eseguibile di un certo applicativo, contiene le istruzioni da eseguire ed è contenuto sul disco fisso.
- Processo - rappresenta l'istanza del programma che viene avviato e caricato sulla memoria principale, una volta avviato, l'OS si occuperà di tale processo.

Quindi un programma viene *istanziato* in un processo, che è un entità dinamica e viene eseguito dalla CPU, ogni processo è un entità indipendente, e possono coesistere due processi istanza dello stesso programma. Ad ogni processo viene assegnata la sua quantità di memoria disponibile, e le sue istruzioni sono eseguite in maniere sequenziale. Il sistema operativo si occupa di creare, distruggere, e gestire gli stati dei processi, dedica ad essi la stessa quantità di memoria virtuale, ed il numero di indirizzi disponibili dipendono dall'architettura della macchina (ad esempio, con un processore a 32 bit, si hanno 2^32 indirizzi disponibili).

Quando si crea un processo, ad esso viene assegnata una quantità di memoria divisa in 5 unità logiche :

- `Text` - contiene le istruzioni eseguibili, ossia il risultato della compilazione.
- `Data` - contiene le variabili globali o statiche inizializzate.
- `Data` - contiene le variabili globali o statiche non inizializzate, o inizializzate a 0.
- `Stack` - struttura LIFO utilizzata per memorizzare i dati ed i parametri necessari alle chiamate di funzioni.
- `Heap` - struttura dati utilizzata per l'allocazione dinamica della memoria.

Per ogni processo quindi, esiste tale area di memoria suddivisa in 5 unità. Lo **Stack** ha su di esso due operazioni, `push()` e `pop()`, ed un registro dedicato chiamato *Stack Pointer* memorizza l'indirizzo alla cima dello stack. Ogni funzione utilizza una porzione dello stack che viene denominata **Stack Frame**, quindi quando si chiamano funzioni dentro altre funzioni, coesisteranno simultaneamente più stack frame, anche se esclusivamente uno di essi sarà attivo, ossia quello sulla quale risiede lo stack pointer (lo stack "cresce" verso il basso, quindi l'ultima funzione chiamata sarà quella attiva).

Lo stack frame contiene :

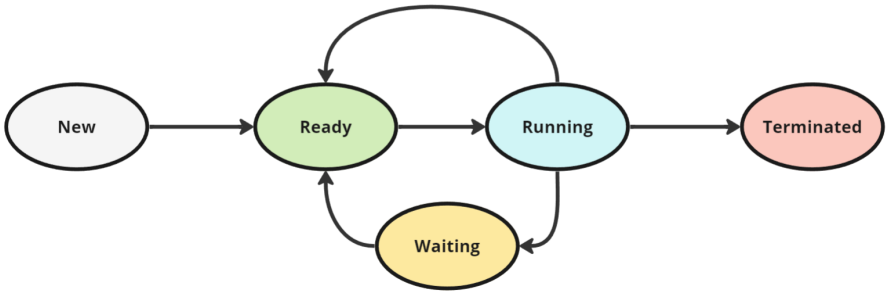
- Parametri della funzione ed indirizzo di ritorno
- Puntatore al precedente dello stack frame precedente
- Variabili locali

Quando si chiama una funzione che richiede dei parametri, essi verrano inseriti (`push()`) nello stack, il valore dello stack pointer verrà aggiornato, e verrà inserito nello stack anche l'indirizzo dell'istruzione di ritorno. Il problema è che lo stack pointer viene aggiornato ogni volta che si chiama una nuova funzione, quindi si usa un altro puntatore detto *Base Pointer* sul fondo dello stack, che rimane fisso per ogni stack frame senza aggiornarsi, differentemente dallo stack pointer che per forza di cosa, si aggiorna ogni qual volta viene aggiunto un nuovo valore.

4.1 Stati di un Processo

Un processo in esecuzione può ritrovarsi in uno dei seguenti 5 **stati** :

- `new` - Il sistema operativo ha indirizzato le strutture per eseguirlo.
- `ready` - Il processo ha tutte le risorse necessarie per iniziare o ricominciare ad essere eseguito.
- `running` - Il processo è in esecuzione sulla CPU.
- `waiting` - Il processo è sospeso, in attesa che venga soddisfatta una chiamata/richiesta che necessita per poter continuare.
- `terminated` - Il processo è concluso, l'OS può liberare la memoria dalle risorse che utilizzava.



Il sistema operativo per creare nuovi processi utilizza delle opportune chiamate di sistema. Per convenzione, un processo *Parent* (detto anche padre) è quello dalla quale si esegue la chiamata per generare nuovi processi, detti *Figli*. Ogni processo ha d.ue valori interi utilizzati per identificare se stesso: `PID` , ed il suo parent : `PPID` . In sistemi come Unix, il process scheduler ha come PID=0, esso inizializza come prima cosa un processo noto come `init` , che ha PID=1, e si occuperà di creare tutti i processi, sarà quindi il parent primario. I processi vengono creati tramite una chiamata di sistema denominata `fork()` . Ogni processo crea più figli, generando una struttura gerarchica ad albero.

La chiamata `fork()` nello specifico, non crea un nuovo processo, ma crea un processo *clone*, identico a quello chiamante, si utilizza poi una chiamata `exec()` , che prendendo come parametri l'indirizzo in memoria di un determinato programma, sostituirà al processo corrente le istruzioni del programma nuovo che si vuole eseguire. Sarà quindi la congiunzione di tali chiamate `fork()` ed `exec()` a generare un nuovo processo.

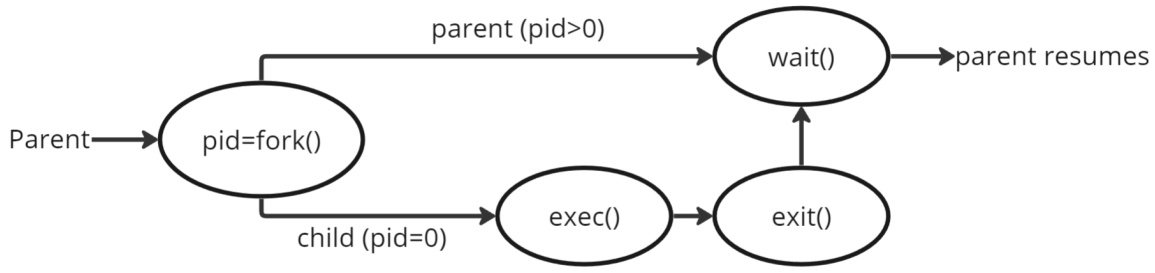
Quando un processo padre genera un figlio, ha due possibili opzioni :

- Arrestare la sua esecuzione, ed attendere che il processo figlio appena generato termini prima di ricominciare, tramite la chiamata `wait()` .
- Continuare la sua esecuzione, in maniera concorrente con il suo processo figlio.

Vediamo come un programma si occupa di generare un nuovo processo tramite la chiamata di sistema :

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

int main(){
    pid_t pid;
    /* fork a child process */
    pid = fork();

    if(pid<0){
        /* error occurred */
        fprint("Fork Failed");
        exit(-1);
    }

    else if(pid==0){
        /* child process */
        execlp("bin/ls","ls",NULL);
    }

    else{
        /* parent process */
        wait(NULL);
        printf("Child Complete");
        exit(0);
    }
}
```

Si osservi il codice sopra mostrato. Quando viene eseguito un `fork()` e creato un clone, i due processi padre e figlio differiranno esclusivamente per il loro PID. La funzione `fork()` ritorna il PID del processo appena clonato, se esso è il processo figlio, il PID sarà 0, per questo si entrà nel blocco di codice che si occuperà di fare l'`exec()` sostituendo le istruzioni con quelle del programma presente all'indirizzo `"bin/ls"`, generando così un nuovo processo. Se il PID è diverso da 0, il programma eseguirà una `wait()`, aspettando che il processo figlio termini, prima di ricominciare. Sarà quindi il programma scritto dall'utente a dover implementare la



logica adeguata (tramite la lettura dei PID) per capire se il processo attualmente in esecuzione è quello padre o quello figlio