Sistemi Operativi 1

Marco Casu

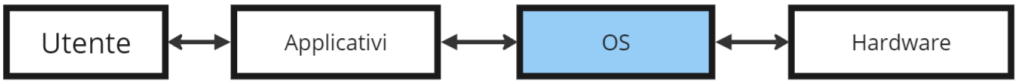
Contents

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 :

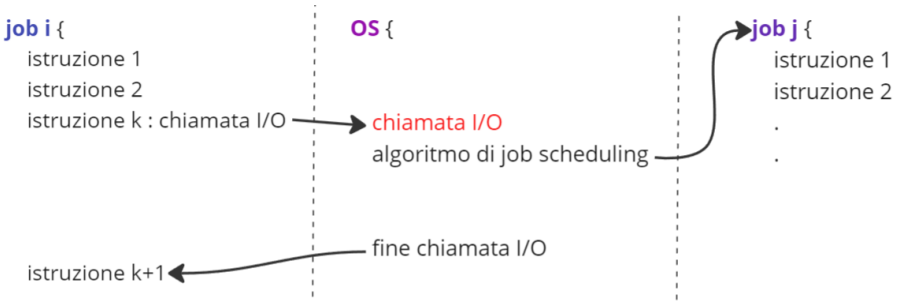
- **port mapped** - I registri dei device controller usano uno spazio di indirizzamento separato dalla memoria principale, ma è necessario estendere l'insieme delle istruzioni elementari del linguaggio macchina per per poter comunicare con questo nuovo spazio.

- **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.

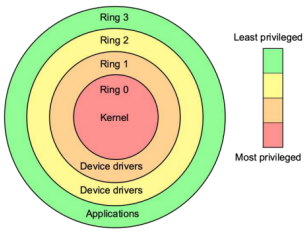


Figure 1: Protection Rings

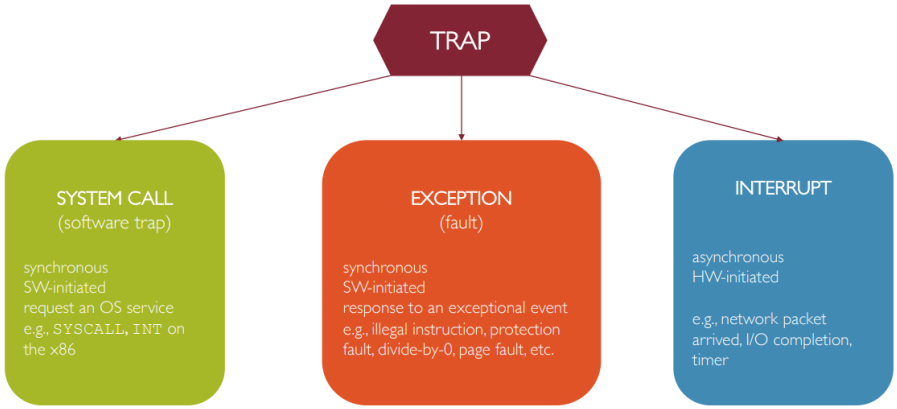
Al minimo è necessario un *bit* per i 2 stati, ma è possibile implementarne molteplici se si vogliono definire dei **protection rings**, ossia più stati di privilegi definiti a strati dove al livello 0 c'è il kernel, e gradualmente si hanno meno privilegi più il livello è alto.

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).

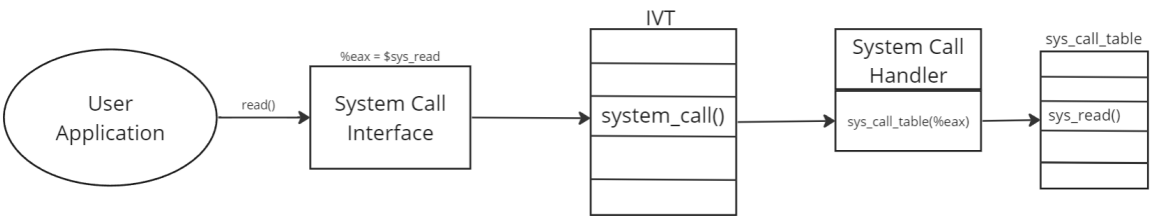


Ci sono 6 categorie principali di system call :

- **Controllo dei processi**
- **Gestione dei file**
- **Controllo dei dispositivi**
- **Manutenzione delle informazioni**
- **Comunicazione 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.

2.5 Design di Noti Modelli di 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.

3 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 :

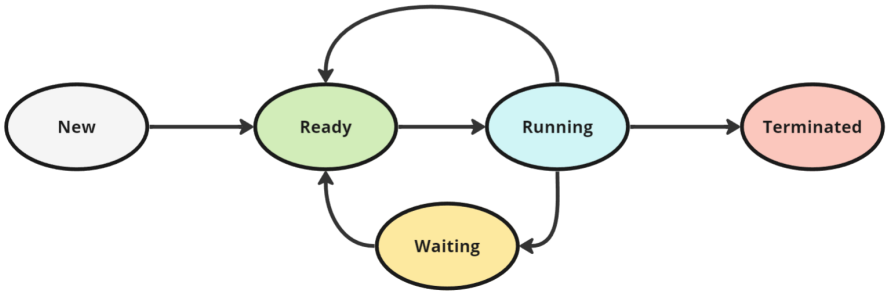
- Parametri della funzione ed indirizzo di ritorno
- Puntatore al precedente dello stack frame precedente
- Variabili locali

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.

3.1 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.



3.2 Creazione dei Processi

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.

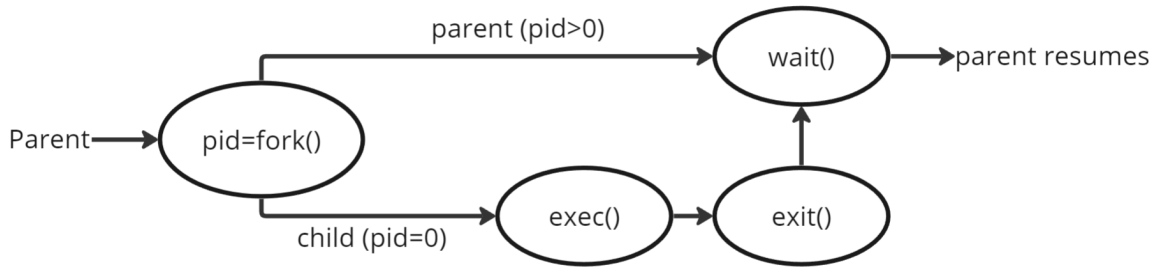
Quando un processo padre genera un figlio, ha due possibili opzioni :

- 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(quindi il padre avrà salvato nella variabile, il PID del figlio) , il processo figlio, avrà la variabile pid=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.

Un processo può esplicitamente richiedere la sua terminazione tramite la chiamata di sistema

`exit()`, dando un codice di uscita, che per convenzione è 0 quando tale processo ha terminato la sua esecuzione senza errori, altrimenti -1, oppure può essere terminato in maniere forzata dal suo processo padre. Un processo non può esistere senza padre, se un processo padre termina ma vi sono ancora dei processi figli in esecuzione, essi, detti processi *orfani*, vengono ereditati dal processo `init`, che si occuperà di terminarli. Quando un processo termina, lo spazio dedicato alle sue risorse viene liberato.

3.3 Scheduling dei Processi

Un sistema operativo deve far conto a due obbiettivi :

- far si che la CPU stia costantemente eseguendo processi
- avere un t empo di risposta accettabile per i programmi che interagiscono con l'utente

Lo scheduler si occupa di decidere quale processo eseguire venendo in contro a tali obbiettivi, anche se essi possono entrare in conflitto fra loro. Il sistema operativo salva tutti i PCBs[1] dei processi in delle code, ci sono 5 code, una per ogni possibile stato. Quando un processo cambia stato, il suo PCB viene eliminato dalla coda precedente ed inserito nella nuova coda corrispondente. Ovviamente, nella *Running Queue* vi può essere un solo processo per volta (in sistemi con architetture single core). Nelle altre code, non ci sono limiti teorici di dimensioni.

Esistono due tipi di scheduler.

- Il **long-term scheduler** si occupa di selezionare i processi da eseguire dalla memoria secondaria (ad esempio, il disco) e di caricarli in memoria principale, viene eseguito poco frequentemente.
- Lo **short-term scheduler** nvece, si occupa di selezionare i processi dalla coda dei pronti e di assegnarli alla CPU per l'esecuzione, viene eseguito molto frequentemente, circa ogni 50-100 millisecondi.

Abbiamo già definit l'operazione di *Content Switch*, ossia quella di interrompere un processo per eseguirne un altro. Tale operazione risulta costosa in termini computazionali, dato che è necessario salvare lostato del processo da interrompere e caricare lo stato del processo da eseguire dai loro rispettivi PCB. Il content switch è quindi operazione da eseguire solo quando necessario, quando avviene un interruzione, oppure quando un processo impiega troppo tempo generando un interruzione del timer. I processi che utilizzano principalmente la CPU per eseguire esclusivamente calcoli (talvolta pesanti), che quindi richiedono più tempo per essere completati, senza però fare richieste di I/O, sono detti *CPU-bound processes*.

Il tempo che impiega la CPU per fare content switch è "sprecato" dato che non si stanno effettuando calcoli utili all'esecuzione dei processi, quindi a seconda delle disponibilità, è possibile implementare un quanto di tempo massimo dedicato ad ogni processo, a seconda delle esigenze :

- Un quanto di tempo minore farà si che si eseguiranno più content switch, aumentando la responsività.
- Un quanto di tempo maggiore risulterà in meno content switch, minimizzando il tempo perso della CPU, massimizzandone il suo utilizzo.

[1] Process Control Block

3.4 Comunicazione fra processi

Due processi possono essere fra loro :

- **cooperativi** - possono influire o venire influiti da altri processi per operare su computazioni comuni.
- **indipendenti** - operano in maniera concorrente sul sistema e non si influenzano fra loro.

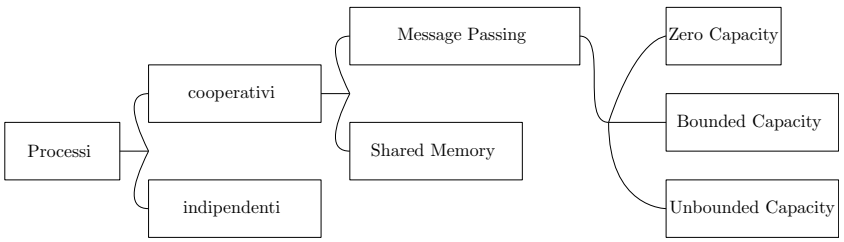
Un processo particolarmente complesso può essere scomposto in più processi *cooperativi*, è necessario predispore i processi cooperativi di adeguati canali di comunicazione, ci sono due possibili modi di farli comunicare :

- **Message Passing** - è il metodo più lento, dato che ogni trasferimento avviene tramite una system call, ma è semplice da mettere in atto quindi preferibile se la quantità o la frequenza dei messaggi non è particolarmente elevata.
- **Shared Memory** - è più complesso da mettere in atto e non funziona perfettamente quando si lavora su più computer, non richiede però l'utilizzo delle system call ed è quindi preferibile quando si devono trasferire grandi quantità di informazioni sullo stesso computer.

Nel sistema *Shared Memory* viene creato uno spazio di memoria condiviso per due processi differenti, inizialmente tale memoria appartiene al PCB di un solo processo, tramite una system call viene poi resa pubblica e disponibile ad altri processi, che possono fare delle system call per accedere ad essa.

Nel *Message Passing*, il sistema operativo deve implementare delle specifiche system call adibite all'invio e ricezione dei messaggi, un collegamento va stabilito tra due processi cooperanti. Il processo mittente deve poter identificare il processo destinatario, è possibile creare un canale "uno ad uno" per ogni coppia di processi comunicanti, oppure creare delle "porte" condivise dove più processi possono comunicare, e trasferire informazioni su tali porte che potranno essere lette da ogni processo. Tali messaggi ovviamente occupano spazio in memoria, solitamente vanno in una coda, ci sono 3 diversi modi di gestire la coda ad essi riservata :

- **Zero Capacity** - I messaggi non hanno una coda nella quale vengono salvati, quindi il mittente si blocca e non può inviare altri messaggi finche il destinatario non risponde.
- **Bounded Capacity** - I messaggi vengono salvati in una coda con una capacità di memoria finita predeterminata, quindi i mittenti potranno inviare messaggi affinché la coda non sarà piena.
- **Unbounded Capacity** - La coda ha una capacità (teoricamente)infinita, quindi i mittenti potranno inviare messaggi senza blocchi.



3.5 Algoritmi di Scheduling

Vediamo in questo paragrafo quali sono alcuni dei criteri presi in considerazione dagli scheduler per selezionare il prossimo processo dalla coda pronti che andrà in esecuzione. Lo scheduler deve far si che si alternino in continuazione processi CPU-bound e processi che fanno operazioni di I/O, tenendo in considerazione che gli accessi alla memoria richiedono svariati cicli di clock, togliendo tempo al calcolo effettivo.

Ci sono 4 possibili situazioni in cui interviene lo short term scheduler per prendere un processo dalla coda pronti :

1. Quando un processo passa dallo stato running allo stato waiting (ad esempio quando vi è una system call).
2. Quando un processo passa dallo stato running allo stato ready (quando scade il timer che determina il tempo massimo assegnato ad ogni processo).
3. Quando il processo passa dallo stato waiting allo stato ready (il processo interrotto perché ha richiesto una risorsa, vede tale risorsa diventare disponibile).
4. Quando viene creato un nuovo processo ed entra nella coda dei processi pronti, oppure quando un processo termina.

Si noti come, nei casi 1 e 4, lo scheduler è costretto ad intervenire, in quanto la CPU necessita di un nuovo processo da eseguire, nei casi 2 e 3 invece, l'intervento non è necessario, ed è a discrezione dello scheduler se intervenire o meno.
Gli scheduler che intervengono esclusivamente quando è necessario, si dicono **Non-preemptive**, quelli che invece, intervongono anche nei casi 2 e 3, sono detti **Preemptive**.

Uno scheduler *Preemptive* può essere problematico se agisce in situazioni delicate (Ad esempio, mentre il Kernel sta eseguendo una system call), per questo prima di eseguire un interruzione, l'OS si occupa di accertarsi che una chiamata di sistema sia completata prima di eseguire il blocco (se necessario, si interrompono momentaneamente le interruzioni).

È importante sapere che il modulo dello scheduler che si occupa specificamente di far eseguire alla CPU il processo selezionato è noto con il nome di **Dispatcher**, si occupa di fare context switch, passare alla user mode, e "saltare" all'indirizzo di memoria dell'ultimo programma caricato.

3.5.1 Criteri e Politiche dello Scheduling

Introduciamo adesso alcune notazioni e definizioni importanti riguardo il calcolo dei tempi che un processo impiega sulla CPU.

- **Arrival Time** ($T^{arrival}$) : Il tempo che impiega un processo per arrivare nella coda pronti.
- **Completion Time** ($T^{completion}$) : Il tempo che impiega un processo per completare la sua esecuzione.
- **Burst Time:** (T^{burst}) : Il tempo totale nella quale il processo è in esecuzione sulla CPU.
- **Turnaround Time** ($T^{turnaround}$) : La differenza fra il Completion Time e l'Arrival Time, definita come $T^{completion} - T^{arrival}$.

- **Waiting Time** ($T^{waiting}$) : La differenza fra il Turnaround Time ed il Burst Time, definita come $T^{turnaround} - T^{burst}$.

Ci sono diversi criteri da poter prendere in considerazione quando si scrive una algoritmo di scheduling, ed in base a tali criteri, esistono algoritmi migliori di altri, tali criteri sono :

- **Utilizzo della CPU** - Il tempo in cui la CPU è occupata ad eseguire calcoli, idealmente, dovrebbe essere occupata il 100% del tempo, l'obbiettivo è quindi di *massimizzare* tale utilizzo.
- **Throughput** - Il numero di processi da completare per unità di tempo, l'obbiettivo è quindi di *massimizzare* tale numero.
- **Tournaround time** - L'obbiettivo è quello di *minimizzare* il tempo impiegato da un processo, dalla sua selezione da parte dello scheduling fino al completamento (includendo lo Waiting Time).
- **Waiting time** - L'obbiettivo è quello di *minimizzare* il tempo che un processo rimane fermo nella CPU in attesa di essere selezionato dallo scheduler.
- **Response time** - Il tempo che interocorre fra la richiesta di un comando, e la risposta ad esso, l'obbiettivo è quello di *minimizzare* tale tempo per rendere il sistema più interattivo e responsivo.

Idealmente sarebbe ottimo scegliere un algoritmo di scheduling che soddisfa tutte le richieste appena elencate, ovviamente in una situazione reale, la massimizzazione di certi criteri porta svantaggio agli altri, rientra quindi il concetto di *trade-off*, sceglieremo quindi un certo algoritmo di scheduling in grado di soddisfare solo un certo tipo di politiche.

- Minimizzare il response time medio fa si che l'utente riceva l'output il più rapidamente possibile, minimizzare il tempo massimo di response time invece, previene dagli scenari in cui un processo ha un response time molto più altro rispetto agli altri, minimizzare la varianza invece, rende il response time dei processi più "predicibile" dall'utente, queste politche son tipiche dei **sistemi interattivi**.
- Massimizzare il throughput si traduce in un utilizzo più efficente delle risorse di sistema, e minimizzare lo waiting time da ad ogni processo la stessa quantità di tempo da spendere sulla CPU (di contro, potrebbe aumentare la media del response time), queste politiche son tipiche dei sistemi batch[2].

Vediamo adesso il funzionamento generale di alcuni algoritmi di scheduling noti, i quali possono essere preemptive oppure non-preemptive.

3.5.2 First Come First Served (FCFS)

Tale algoritmo funziona, ed è implementato come una coda *First-In-First-Out*, è quindi molto semplice, in quanto manda in esecuzione i processi in ordine di arrivo, lo scheduler entra in funzione solamente quando un processo in esecuzione fa una richiesta di I/O (oppure termina la sua esecuzione).

[2]tipo di sistema in cui i processi vengono eseguiti in gruppi o lotti. Invece di eseguire ogni processo in modo interattivo e immediato, un batch system raccoglie una serie di processi o comandi e li esegue in sequenza, senza richiedere l'interazione dell'utente durante l'esecuzione.

I processi non sono limitati da un tempo massimo quindi possono rimanere in esecuzione sulla CPU per un tempo indeterminato, è quindi uno scheduler *non-preemptive*.

- **Pro** - Risulta molto semplice da implementare.
- **Contro** - Il tempo di attesa (waiting time) è molto variabile. Vi è poca alternanza di processi I/O-bound e processi CPU-bound dato che un processo che fa richiesta di I/O potrebbe trovarsi dopo (e quindi attendere la terminazione) di un processo che impiegherà molto tempo sulla CPU.

3.5.3 Round-Robin (RR)

È un algoritmo simile al FCFS, solo che impone un limite di tempo (detto quanto temporale) ai processi che impiegano troppo tempo sulla CPU. Quando un job viene assegnato alla CPU, parte un timer (implementato a livello hardware).

Se il processo termina prima della scadenza del timer, lo scheduler manda in esecuzione il prossimo come un semplice FCFS, se invece il timer scade durante l'esecuzione del processo, lo scheduler interviene dando in esecuzione alla CPU il prossimo processo, quello interrotto invece verrà re-inserito in fondo alla coda, è quindi un sistema *preemptive*.

Come implicato, i processi pronti sono gestiti in una coda circolare, è uno scheduler equo in quanto garantisce a tutti i processi lo stesso tempo di esecuzione sulla CPU. Le performance di questo algoritmo sono sensibili alla durata del quanto temporale scelto, se il tempo del quanto è molto grande, l'algoritmo si comporterà come un FCFS, se troppo piccolo, verranno fatti troppi context switch.

3.5.4 Shortest-Job-First (SJB)

L'idea, è di dare la priorità ai processi che hanno il "carico di lavoro stimato" più grande sulla CPU, con carico di lavoro, si intende il tempo che impiegheranno in esecuzione, prima della prossima richiesta di I/O o della terminazione stessa.

- **Pro** - È ottimale quando l'obbiettivo è quello di minimizzare il tempo di attesa.
- **Contro** - È quasi impossibile sapere con precisione quale sia il prossimo processo con il minor carico di lavoro da eseguire. Inoltre, vi è il rischio che i processi con un carico di lavoro elevato entrino nella condizione di *starving*[3].

Come si può però "predirre" il tempo che un processo impiegherà sulla CPU? L'idea, è quella di basarsi sui tempi di esecuzione dei processi precedenti, tramite una tecnica nota come **exponential smoothing**.

x_t = Tempo attuale sulla CPU del t-esimo processo (valore noto).
s_t = Tempo stimato sulla CPU del t-esimo processo (valore noto).
Dati x_t e s_t, vogliamo stimare il tempo di attesa del prossimo processo, ossia s_{t+1}.
Sia α un coefficente reale : $\alpha \in [0, 1]$, la predizione sarà :

$$s_{t+1} = \alpha \cdot x_t + (1 - \alpha) \cdot s_t$$

[3]Restano nella coda senza mai essere selezionati dallo scheduler, in quanto hanno bassa proprità di essere selezionati.

Facciamo alcune osservazioni :

$\alpha = 0 \implies s_{t+1} = s_t \implies$ Si ignora il tempo misurato del processo precedente. e si assume un tempo di lavoro costante per tutti i processi.

$\alpha = 1 \implies s_{t+1} = x_t \implies$ Si assume che il tempo di lavoro del prossimo processo sia lo stesso del processo precedente.

Solitamente, si decide che $\alpha = \frac{1}{2}$. Tale algoritmo può essere implementato sia in maniera *preemptive* che *non-preemptive* :

- Una volta che un processo viene assegnato, rimane sulla CPU fino alla prossima interruzione (o terminazione).
- Quando un nuovo processo arriva nella coda pronti, si controlla se esso ha un tempo di lavoro stimato maggiore del processo in esecuzione, se si, viene eseguito lo switch.

3.5.5 Priority Scheduling

È il caso generale del SJF, in cui ad ogni job viene assegnato un grado di priorità in base a certi parametri, e lo scheduler prediligerà i processi con la priorità più alta (nel caso del SJF, più il carico di lavoro è basso, più il processo è prioritario). Il livello di priorità non è altro che un valore intero, mantenuto in un certo range (usualmente, più il numero è basso più la priorità è alta).

Le politiche possono essere assegnate :

- **Internamente** - Il grado di priorità è assegnato dal sistema operativo considerando politiche e criteri interni come il tempo di lavoro sulla CPU, l'alternanza fra richieste di I/O ed esecuzione ecc..
- **Esternamente** - Il grado di priorità è assegnato dall'utente basandosi su criteri totalemente arbitrari in base all'importanza del processo.

Come visto prima, tale algoritmo può essere implementato sia in maniera *preemptive* che *non-preemptive*.

Si è accennato precedentemente della **starvation**, ossia della condizione in cui un processo con bassa priorità, rischia di non essere mai selezionato in quanto surclassato da processi sempre con gradi di priorità superiori. La soluzione più ovvia risulta essere quella denominata con **aging**, ossia, l'operazione di aumentare il grado di priorità di un processo nel tempo, in modo che, anche i processi poco prioritari, con l'andare avanti nel tempo verranno prima o poi schedulati.

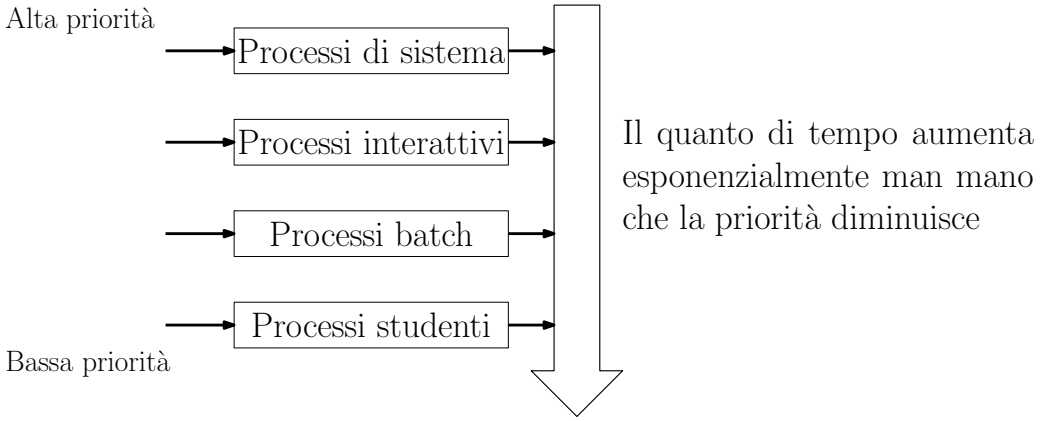
3.5.6 Multi-Level-Queue (MLQ)

L'insieme dei processi viene diviso in partizioni, in base alla categoria di ogni processo (ad esempio, una categoria per i processi grafici, una per i processi correlati all'audio, ecc..), e si applicano algoritmi diversi di scheduling su ogni categoria.

Un esempio è l'applicazione della **Strict Priority**, in cui i processi vengono divisi in diverse

code in base ad un grado di priorità deciso, e poi ogni singola coda avrà il proprio algoritmo di scheduling, ad esempio, il RR, dove ogni coda ha un quanto temporale diverso dalle altre.

Attenzione : Nessun processo può passare da una coda ad un altra.



3.5.7 Multi-Level-Feedback-Queue (MLFQ)

L'idea è simile alla normale MLQ, solo che in questo caso, i processi possono *muoversi fra le diverse code*. Ciò può essere necessario in diverse situazioni, per rendere lo scheduling più adattivo, ad esempio :

- Un processo cambia le sue caratteristiche, alternandosi nell'essere un CPU-Bound ed un processo che fa molte richieste di I/O.
- Un processo può rimanere in attesa troppo tempo, quindi l'aging interviene spostandolo in una coda con priorità maggiore.

Di default, i processi vengono tutti inizializzati nella coda con priorità più alta, che ha a sua volta il quanto temporale più corto. Se il tempo a disposizione scade, vengono spostati nella coda successiva, di un grado di priorità inferiore, e così via.

Se il quanto temporale di un processo invece non lo blocca (ad esempio, il processo si interrompe con richieste di I/O sempre prima che il suo tempo a disposizione scada), allora viene spostato nella coda superiore, con priorità più alta. I processi che richiedono molto calcolo e tempo sulla CPU, cadranno rapidamente nelle code più "basse" con meno priorità, differentemente, i processi che fanno molte richieste di I/O rimarranno nelle code più "alte".

Uno scheduling MLFQ è estremamente flessibile, ma anche molto complesso da implementare, bisogna gestire svariati parametri, quali :

1. Il numero delle code.
2. L'algoritmo di scheduling per ogni coda.
3. Le politiche adoperate per spostare i processi fra le code.
4. Il metodo utilizzato per determinare in quale coda un processo dovrebbe essere inizializzato.

Questo algoritmo, si comporta in maniera simile al SJF in termini di tempo di attesa, in quanto cerca di privilegiare i processi corti, potrebbe quindi essere poco equo.

3.5.8 Lottery Scheduling

Esiste un tipo di scheduling inusuale basato sulla casualità. Ad ogni processo, viene assegnato un certo numero di "ticket". Ad ogni intervallo di tempo, si estrae **casualmente** (con probabilità uniforme) un numero. Il processo in possesso del ticket con il numero estratto, verrà schedulato. Con l'andare avanti del tempo, per la *legge dei grandi numeri*, prima o poi ogni processo verrà schedulato.

Risulta opportuno dare più ticket ai processi più corti, in modo che possano essere schedulati più frequentemente (simulando il SJF). Per evitare la starvation, si assegna ad ogni processo alemeno un ticket. Inoltre, aggiungere o rimuovere processi dalla code, condiziona in maniera proporzionale la probabilità di essere selezionati di tutti gli altri processi. Questo tipo di scheduling quindi, **non è deterministico**, e si basa sulla *randomness*.

$$
\begin{aligned}
m_i &:= \text{numero di ticket assegnati al processi } i \\
N &:= \text{numero dei processi} \\
M = \sum_{i=1}^{N} &:= \text{numero totale dei ticket} \\
\mathbb{P}(i) = \frac{m_i}{M} &:= \text{probabilità che il processo } i \text{ venga schedulato.}
\end{aligned}
$$

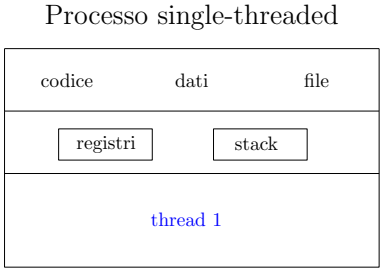
4 I Threads

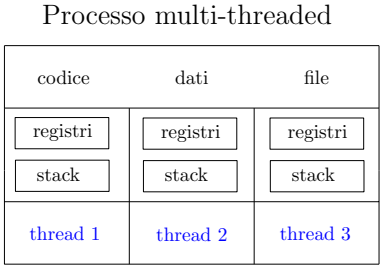
4.1 Definizione e Motivazioni

Fino ad ora, abbiamo sempre trattato ogni processo come un entità singola (ossia, *single-threaded*), ma i sistemi operativi moderni sono per la maggiorparte orientati alla gestione dei processi *multi-threaded*.

Un **thread**, è l'unità più piccola schedulabile sulla CPU, è composto da un program counter, uno stack, un insieme di registri dedicati ed un thread ID. I processi, sono composti da più thread, che risultano essere più unità di controllo. Diversi thread di un singolo processo, hanno differenti PC, stack e registri, ma condividono lo stesso codice e la stessa quantità di memoria dedicata.

- Un **processo** definisce uno spazio di indirizzamento, il codice, e le risorse.
- Un **thread** definisce una singola sequenza/flusso di esecuzione all'interno di un processo.





Un thread ovviamente non esiste "singolarmente", ma è sempre parte di un processo. "Vivendo" nello stesso codice e nella stessa memoria, la cooperazione fra più thread di un processo risulta facile e non necessita di chiamate di sistema.

Quando un processo deve svolgere più compiti è oppurtuno suddividerlo in diversi thread, soprattutto quando un particolare compito potrebbe interrompersi, se esso viene programmato come thread singolo, la sua interruzione non causerà l'interruzione dell'intero processo.

Teoricamente, ogni compito di un processo potrebbe essere implementato come un nuovo processo single-threaded, ma non risulta essere la migliore opzione, in quanto la comunicazione fra thread di uno stesso processo è più rapida, ed i context-switch lo sono altrettanto. L'utilizzo dei thread, porta **4 benefici principali** :

1. **Responsività** - Un thread provvede ad essere rapido e responsivo, in quanto non viene interrotto, se il resto dei thread del suo stesso processo sono bloccati o rallentati da computazione intensa.
2. **Condivisione delle risorse** - Diversi thread condividono lo stesso codice e spazio di indirizzamento.
3. **Convenienza** - Creare e gestire diversi thread è più rapido rispetto ad eseguire le stesse operazioni ma con processi differenti.
4. **Scalabilità** - Nei sistemi a più processori, un singolo processo può vedere diversi thread venire eseguiti contemporaneamente sui diversi processori.

Spesso, le architetture moderne, prevedono diversi *core*, ossia, più CPU sono disponibili al calcolo, permettendo il vero parallelismo. Ci sono due modi differenti per "parallelizzare" il carico di lavoro :

- **Parallelismo dei dati** - Si divide la porzione di dati su cui lavorare in diversi core, gestiti da diversi thread, e si performa l'operazione su entrambe le porzioni di dati.
- **Parallelismo delle operazioni** - Si divide il compito da svolgere in diverse operazaioni che verranno eseguite simultaneamente su diversi core.

Esistono diversi problemi complessi che svolgono sia operazioni intensive di I/O, che calcoli intensi sulla CPU. La suddivisione in thread può risultare utile anche in architetture a single-core. Dividendo le operazioni I/O e di calcolo, è possibile eseguirle nel solito pseudo parallelismo alla quale siamo abituati, senza che il calcolo intensivo blocchi le richieste di I/O e viceversa.

Anche se suddividere in thread un operazione puramente CPU-bound può essere contro producente, ciò risulta comunque più efficace in quanto elimina il tempo di attesa che il processo dovrebbe attendere nel mentre che una richiesta di I/O non è stata ancora completata.

4.2 User e Kernel Thread

La gestione dei thread può avvenire :

- A livello del Kernel, che gestisce i così detti **Kernel thread**.
- A livello utente, gestiti nello spazio utente da delle **thread library**, senza la necessità che intervenga l'OS.

Un **Kernel thread** è la più piccola unità schedulabile dall'OS, e quest'ultimo è responsabile di gestirli, ad ogni processo è associato un *Process Control Block* (PCB), e ad ogni thread un *Thread Control Block* (TCB). Il sistema operativo mette a disposizione dell'utente una serie di chiamate di sistema per creare e gestire i thread.

- **Pro** : Il kernel ha piena conoscenza di questi thread, lo scheduler quindi, sapendo quanti thread sono associati ad un processo, può riservargli più tempo sulla CPU. Sono ottimi nelle applicazioni che prevedono svariate interruzioni ed eseguire lo switch fra i thread risulta più rapide che eseguire lo switch fra processi.
- **Contro** : La complessità del Kernel aumenta significativamente, inoltre, invocare troppe volte il kernel per la gestione di questi thread è inefficente. Anche se il context-switch è più rapido, richiede sempre l'avvento del kernel.

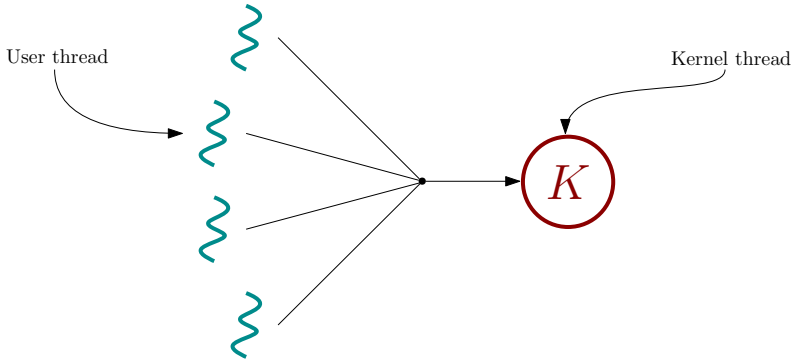
Gli **user thread** sono gestiti totalmente durante l'esecuzione da delle librerie a livello utente. Il Kernel, non sa dell'esistenza di questi thread, e li tratta/vede come se fossero dei processi a single-thread. Idealmente, le operazioni fra thread di uno stesso processo dovrebbero essere rapide come delle chiamate di funzione.

- **Pro** : Molto veloci e leggeri, e le politiche di scheduling sono flessibili. Possono essere implementati in sistemi che non supportano nativamente il multi-threading, e non richiedono chiamate di sistema, ma solo chiamate di funzione, non avvengono dei veri content-switch.
- **Contro** : Non avviene una vera concorrenza fra i vari thread, e le possibili decisioni che può prendere lo scheduler sono limitate, il Kernel non sa nulla di essi, quindi potrebbe far competere per un quanto di tempo un processo con 100 thread con uno disposto di un singolo thread, richiede delle chiamate di sistema non bloccanti, d'altro canto gli altri thread del processo devono comunque attendere.

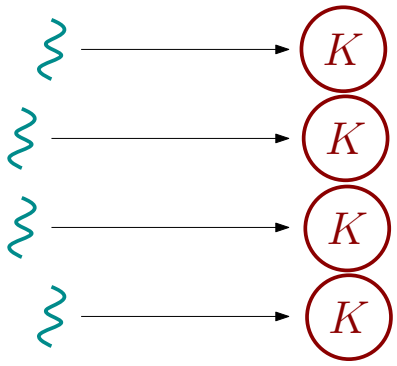
4.2.1 Modelli di Multi-Threading

Quando si vuole implementare un sistema multi-threading, bisogna decidere come gli user thread verranno "mappati" ai Kernel thread, per essere schedulati sulla CPU, vi sono diversi modi.

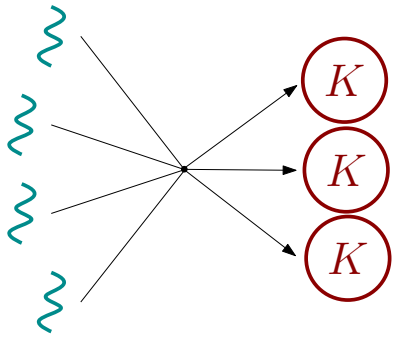
Many-to-One : Più user thread vengono mappati in un solo Kernel thread, quindi il processo, vedrà i suoi thread venire eseguiti uno alla volta, dato che un solo kernel thread schedulato sulla CPU vi è associato, quindi non possono essere suddivisi su diversi core, dato che su ogni core può lavorare un solo kernel thread. Una chiamata di sistema può bloccare l'intero processo, anche se i thread potrebbero idealmente continuare.



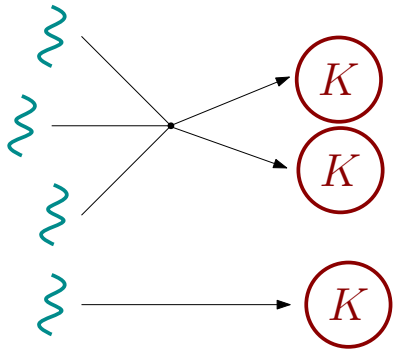
One-to-One : Ogni singolo user thread viene mappato su un Kernel thread, non vi sono più limitazioni sulle chiamate bloccanti e si può dividere un processo su più CPU. Gestire però un sistema di questo tipo può è più complesso e può rallentare l'esecuzione, molte implementazioni di questo tipo, prevedono delle restrizioni sul numero di thread che possono coesistere simultaneamente.



Many-to-Many : Un qualsiasi numero di user thread può essere mappato in un qualsiasi numero di Kernel thread, l'utente non ha alcun tipo di restrizione sul numero di thread che possono essere creati, i processi possono essere divisi su più CPU e le chiamate bloccanti non interrompono l'intero processo.



Two-Level : È una variante del modello *Many-to-Many*, in cui un singolo user thread, ha la possibilità di essere mappato su un singolo Kernel thread, ciò aumenta la flessibilità delle politiche di scheduling.



4.2.2 Le Librerie per la Gestione dei Thread

Esistono delle librerie che forniscono all'utente delle API per creare e gestire i thread, e ci sono due modi principali per implementarle.

- Con delle funzioni interamente definite da delle API all'interno dello spazio utente.
- Implementate nel Kernel, con delle system call (è necessario che il Kernel supporti la programmazione multi-thread).

Vediamo come funziona una delle thread library più in uso tutt'oggi, ossia la *Java Thread Library*. I thread, in questa libreria, son detti *Java thread*, e possono essere creati in due modi, estendendo la classe `Thread`, oppure implementando l'interfaccia `Runnable`, per entrambe le soluzioni, è richiesto che l'utente esegua l'override del metodo `run()`. Essendo che ogni classe può estendere solo una classe, è prefiribile implementare l'interfaccia `Runnable`.

4.2.3 Thread Pools

Nell'esempio della Java Thread Library, abbiamo visto come, gestire una certa richiesta creando più thread, piùttosto che più processi, è più efficente. Nonostante ciò, possono occorrere dei problemi, il numero dei possibili thread coesistenti non è limitato, e può diventare pesante creare troppi thread se ci sono troppe richieste.

Vediamo cos'è una *thread pool*, quando un processo viene creato, si genera un numero definito di thread, tali thread sono "vuoti", e vengono messi in attesa che arrivi una richiesta da svolgere. Quando occorre una richiesta, uno dei thread dalla pool verrà selezionato ed adoperato a dovere, una volta terminata tale richiesta, il thread sarà di nuovo in attesa, disponibile per essere selezionato. Se non ci sono thread disponibili, la richiesta dovrà attendere.

L'utilizzo delle thread pool porta svariati benefici :

- Assegnare una richiesta ad un thread già esistente, è più efficente piuttosto che crearne uno ogni volta.
- la pool, limita il possibile numero di thread che possono esistere contemporaneamente.
- Separare il compito da svolgere, dal meccanismo che si occupa di eseguirlo, permette di usare differenti strategie per eseguire un compito(ad *esempio*, potremmo far si che un'operazione venga eseguita dopo un certo delay temporale definito, oppure periodicamente).

Che succede su viene eseguito un `fork()` su un thread? Viene duplicato esclusivamente il thread o l'intero processo? Tale decisione, dipende dalle specifiche del sistema operativo, se il nuovo processo chiama immediatamente un `exec()`, non c'è bisogno di copiare tutti i thread associati a quel processo. Molte versioni di UNIX, implementando differenti tipi della chiamata `fork()` a seconda della situazione.

Se un processo multi-thread riceve un segnale, quale dei thread associati dovrà riceverlo? Ci sono 4 possibili modi di gestire i segnali :

1. Inviare il segnale allo specifico thread che lo richiede.
2. Inviare il segnale a tutti i thread del processo.
3. Inviare il segnale ad un certo gruppo di thread.
4. Definire un thread specifico, che avrà lo scopo di ricevere e gestire tutti i segnali.

I thread competono per essere schedulati sulla CPU in diversi modi, un modo, detto **process contention scope (PCS)**, prevede la concorrenza fra diversi thread dello *stesso processo*, nei sistemi che implementano i modelli many-to-one e many-to-many. Nei sistemi one-to-one, occorre il **system contention scope (SCS)**, lo scheduler di sistema, schedula i kernel thread per essere eseguiti su una, o più CPU.

Molte implementazioni prevedono un interfaccia virtuale posta fra gli user thread ed i kernel thread, il numero di kernel thread potrebbe variare dinamicamente, essi saranno schedulati sui processori fisici.

Le librerie dei thread a livello utente, si occupano anche di schedulare gli user thread nei kernel thread, tali metodi di scheduling possono essere cooperativi o preemptive. Nei sistemi **cooperativi** gli user thread sono mappati sul kernel thread affinché non daranno *volontariamente* il controllo del thread al kernel. In quelli **preemptive** è previsto un timer, in maniera simile agli scheduler preemptive dei processi, che una volta scaduto lo user thread lascierà libero il kernel thread.

4.3 Sincronizzazione dei Thread e Processi

Abbiamo già accennato al fatto che diversi processo/thread possono cooperare, il problema sorge quando uno di essi, deve eseguire una regione *critica* di codice, che richiede l'esecuzione in maniera atomica, è quindi necessario implementare delle metodologie **primitive** di sincronizzazione.

Un processo/thread, prima di entrare in una sezione critica di codice, deve assicurarsi un **lock**, in modo da far attendere tutti gli altri processi nel mentre che esegue tale sezione (il problema tipico da evitare, è l'accesso da parte di due differenti processi alla stessa sezione di memoria, rischiando di perdere informazioni), i meccanismi di sincronizzazione quindi si basano sull'attesa dei processo/thread, ogni soluzione di sincronizzazione deve garantire 3 proprietà :

- **Esclusività** - Un solo processo/thread alla volta può eseguire la sua sezione critica.
- **Vivibilità** - Se nessun processo/thread è nella sezione critica, ed un altro o più processi vogliono eseguire la propria, allora devono tutti avere la possibilità di poterla eseguire
- **Attesa Limitata** - Prima o poi un processo che richiede di eseguire la sezione criticà, otterrà il permesso, e vi è un limite al numero di altri processi che possono precederlo.

Si necessita di appositi strumenti forniti dai linguaggi di programmazione da utilizzare come blocchi di codice da eseguire atomicamente per la sincronizzazione. Tali strumenti sono :

- **Lock** - Un solo processo per volta può richiedere una lock, eseguire la sua sezione critica indisturbatamente, per poi rilasciarla.
- **Semafori** - Una generalizzazione delle lock.
- **Monitor** - Per connettere i dati condivisi alle primitive di sincronizzazione.

Tali metodi richiedono l'intervento del sistema operativo.

4.3.1 Lock

Una lock fornisce l'esclusività dei dati condivisa tramite due primitive atomiche :

- `Lock.acquire()` - Attende che il lock sia libero, per poi acquisirlo.
- `Lock.release()` - Rilascia il lock, attivando un qualsiasi thread che è in attesa dopo aver richiesto l'acquisizione.

Bisogna sempre acquisire un lock prima di accedere ai dati condivisi, e rilasciarlo sempre e solo dopo averli utilizzati ed eventualmente modificati, all'avvio del sistema, il lock deve essere libero. Un solo processo alla volta acquisice il lock, gli altri dovranno aspettare. Implementare delle primitive di sincronizzazione richiede il supporto hardware a basso livello.

Ci si preoccupa della sincronizzazione perché un context switch nei sistemi preemptive può avvenire inaspettatamente in qualsiasi momento, vogliamo far si che lo scheduler non prevenga nel controllo della CPU nel mentre che un processo ha acquisito una lock, nei sistemi a singola CPU, possiamo impedire che lo scheduler prevenga, **forzando i thread** a non fare richieste di I/O durante le sezioni critiche, e **disabilitando le interruzioni**, bisogna prevedere tutti i possibili casi in cui il thread possa perdere il controllo della CPU, volontariamente o involontariamente.

In quanto è necessario l'intervento dell'OS, vogliamo che i metodi `acquire()` e `release()` siano implementati tramite system call.

Si richiede l'implementazione di un istruzione atomica **read-modify-write**, che in un "singolo colpo" legga un valore dalla memoria in un registro, e scriva un nuovo valore. In un sistema a singolo processore possiamo direttamente implementare una nuova istruzione a livello macchina, nei sistemi multi-core invece, il processore che esegue tale istruzione, deve essere in grado di cancellare ogni copia di tale valore, da modificare, che gli altri processori potrebbero avere nella cache.

Disabilitare le interruzioni può comportare diversi problemi, prima di tutto, richiede ogni volta l'intervento del kernel, inoltre non è flessibile nelle architetture multi-core. I problemi dovuti invece dalle istruzioni atomiche, comportano l'attesa degli altri processi, inoltre non c'è una coda dove i thread possono attendere che il lock sia rilasciato, quindi pecca di equità.

Abbiamo visto come i lock permettono la protezione durante l'esecuzione di sezioni critiche, vediamo adesso delle primitive di sincronizzazione di più alto livello, destinate a casistiche più generali.

4.3.2 Semafori

È un'altra struttura dati che fornisce l'accesso esclusivo alle sezioni critiche, è una generalizzazione dei lock creata da *Dijkstra* nel 1965. Si utilizzano delle variabili (intere) per supportare 2 istruzioni atomiche :

- `wait()` / `P()` - decrementa, interrompe finchè il semaforo è libero.
- `signal()` / `V()` - incrementa, permette ad altri thread di entrare.

Ad ogni semaforo è associata una coda di processi in attesa, se un thread chiama la funzione `wait()`, se il semaforo è aperto, il thread continua, altrimenti si blocca nella coda. La funzione `signal()` apre il semaforo, se un thread sta attendendo nella coda verrà sbloccato, se nessun thread è in attesa nella coda, il segnale resterà libero per il prossimo thread che entrerà.

Ci sono due tipi di semafori, i **semafori binari** garantiscono l'esclusività di accesso alle risorse, ad essi è associata una variabile intera che può valere 0 o 1, vengono inizializzati liberi, con la variabile ad 1. I **counting semaphore**, servono a gestire più risorse condivise, sono inizializzati con il numero delle risorse, ed un processo può accedere ad una risorsa affinché essa è disponibile.

Abbiamo detto che se un processo richiede una `wait()`, se il semaforo è libero eseguirà la sezione critica, altrimenti entrerà nella coda di attesa. D'altro canto, `signal()`, libererà dalla coda uno dei processi in attesa di eseguire la sezione critica.

Abbiamo quindi visto che i semafori hanno 3 scopi principali :

- Garantire l'esclusività come fanno le lock.
- Controllare l'accesso a più aree di memoria condivisa.
- Rinforzare le costrizioni dello scheduling per eseguire i thread secondo uno specifico ordine.

I semafori però hanno vari problemi, sono essenzialmente, delle variabili globali condivise, la loro correttezza dipende dalle abilità del programmatore, da soli hanno più scopi, e non è immediato capire il significato del waiting/signaling. Per questo si utilizzano primitive di più alto livello.

4.3.3 Monitor

Un monitor, è un costrutto, un paradigma di programmazione, utilizzato per il controllo degli accessi alle aree di memoria condivise. Similmente a *Java* e *C++*, delle classi impersonificano i dati, le operazioni e la sincronizzazione, il codice della sincronizzazione è aggiunto dal compilatore.

Un monitor, definisce una lock, e delle *variabili di condizione*, per gestire l'accesso concorrente alla memoria, utilizzano i lock per assicurarsi che un solo thread alla volta sia attivo nel monitor. Si può trasformare una classe di *Java* in un monitore settando tutti i dati a `private`, e facendo si che tutti i metodi siano sincronizzati, nel senso che sono soggetti ad esclusività. Si veda l'**esempio 1.0**.

In tale esempio, il metodo `remove()` dovrebbe attendere che qualcosa sia disponibile nella coda, intuitivamente, il thread dovrebbe eseguire uno `sleep()` dentro la sezione critica, facendo ciò, starà occupando il lock, bloccando tutti gli alri thread dall'accedere alla coda, per aggiungere `item` e svegliare il thread che sta dormendo, si rischia una situazione di stallo che verrà vista in seguito, detta **deadlock**. Una soluzione a ciò, sono le *variabili di condizione*, per far si che un thread possa eseguire uno `sleep` in una sezione critica, ed ogni lock mantenuto da un thread, viene atomicamente liberato prima che esso possa entrare nella condizione dormiente. Ogni variabile di condizione supporta 3 operazioni :

- **wait** - rilascia la lock ed esegue lo `sleep()` atomicamente.

- **signal** - Sveglia un thread in attesa (se ce ne sono).
- **broadcast** - Sveglia tutti i thread in attesa.

Un thread deve per forza occupare una lock nel mentre che esegue un'operazione della variabile di condizione.

Esempio 1.0 :

```
class Queue {
    ...
    private ArrayList<Item> data;
    ...
    public void synchronized add(Item i) {
        data.add(i);
    }
    public Item synchronized remove() {
        if (!data.isEmpty()) {
            Item i = data.remove(0);
            return i;
        }
    }
}
```

4.4 I Deadlock

Una legge del Kansas del 1900 recita :

> "Quando due treni si avvicinano a un incrocio, entrambi devono fermarsi completamente e nessuno dei due deve ripartire finché l'altro non se ne è andato."

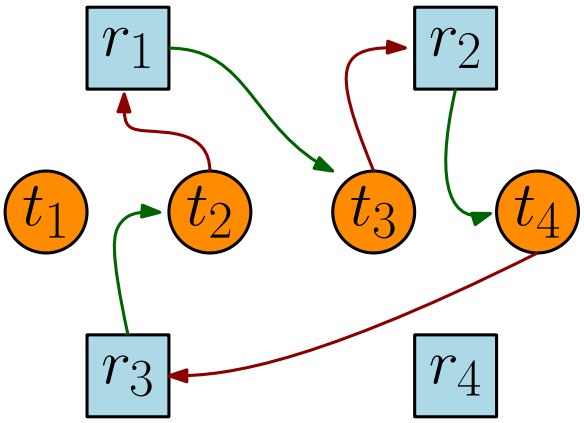
Un *deadlock*, è una condizione di stallo, in cui due o più thread sono bloccati, in attesa di un evento che può essere generato esclusivamente dagli stessi thread. Esso può accadere quanto più thread competono per accedere ad un numero finito di risorse, esistono algoritmi di **deadlock detection**, per trovare delle istanze di deadlock da risolvere, è necessario in primis, scrivere del codice che sia libero dalla possibilità di avvento dei deadlock.

Un deadlock può accadere se le 4 condizioni seguenti avvengono contemporaneamente :

1. Un thread sta occupando una risorsa non condivisibile, in esclusività.
2. Almeno un thread che sta occupando una risorsa non condivisibile, è in attesa che un altra risorsa (occupata da un altro thread) diventi disponibile.
3. Un thread può rilasciare una risorsa solo volontariamente, quindi non vi è un sistema preemptive.
4. Si è in una situazione di attesa circolare, in cui ci sono dei thread in attesa : $t_1, t_2 \ldots, t_n$, dove il thread t_i sta aspettando il thread $t_{(i+1) \mod n}$.

4.4.1 Rilevamento dei Deadlock

Definiamo un **grafo di allocazione** $G = (V, E)$, dove V è l'insieme dei vertici rappresentanti le risorse ed i thread, E è l'insieme degli archi fra risorse e thread. Gli archi possono essere di 2 tipi, gli **archi di richiesta** vanno dai thread alle risorse ed indicano che quel thread ha richiesto quella risorsa, senza averla ancora acquisita, gli **archi di assegnamento** vanno da una risorsa ad un thread, ed indicano che quella risorsa è stata allocata a quel thread.



Se il grafo non ha cicli, nessun deadlock occorrerà mai. Nel grafo esplicativo, il thread t_4 sta attendendo che r_3 sia libera, ma sarà liberà solo quando t_2 avrà r_1, che sarà libera solo quando t_3 avrà r_2, che si libererà solo quando t_4 avrà r_3! Bisogna analizzare il grafo delle risorse allocate (RAG) per trovare cicli, ed eventualmente, romperli, o eliminando tutti i thread del ciclo, oppure eliminando i thread uno per volta, forzandoli a rilasciare le risorse.

Rilevare i cicli in un grafo è un operazione costosa, ci sono algoritmi basati sulla **depth-first-search (DFS)**, che impiegano tempo $O(|V| + |E|) \sim O(|V|^2)$, in quanto $|E| = O(|V|^2)$ nei grafi densi.

Quando dovremmo eseguire tale algoritmo?

- Prima di dare una risorsa, ma così ogni volta che si esegue tale operazione, si sprecherà tempo $O(|V|^2)$.
- Quando una richiesta non può essere soddisfatta, ma così ogni richiesta fallita sprecherà tempo $O(|V|^2)$.
- Quando la CPU è in uno stato in cui non ha molti calcoli da fare ed è particolarmente libera.

Parliamo innanzitutto di prevenzione dei deadlock, bisogna fare in modo che almeno una delle 4 condizioni precedentemente elencate non sia in corso. Ogni thread fornisce informazioni sul numero massimo di risorse che potrebbe richiedere :

$$m_i = \text{numero massimo di risorse che il thread } i \text{ può richiedere}$$
$$c_i = \text{numero di risorse correnti sul quale il thread } i \text{ sta operando.}$$
$$C = \sum_{i=1}^{n} c_i = \text{numero totale di risorse correnti allocate.}$$
$$R = \text{numero massimo di possibili risorse allocabili.}$$

Una sequenza di thread è in uno **stato sicuro** se :

$$m_i - c_i \leq R - C + \sum_{j=1}^{i-1} c_j$$

Dove :

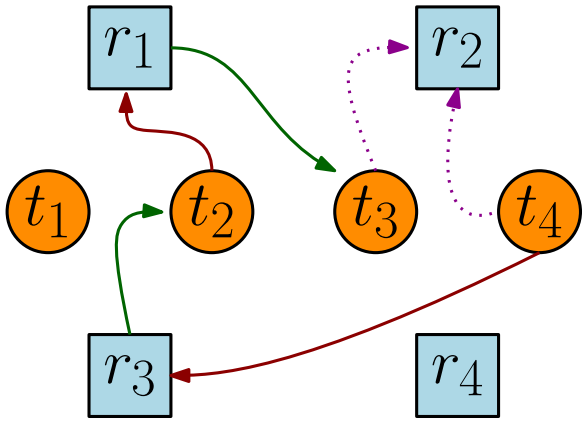
$$m_i - c_i = \text{numero di risorse che il thread } t_i \text{ potrebbe richiedere.}$$
$$R - C = \text{numero di risorse disponibili}$$
$$\sum_{j=1}^{i-1} c_j = \text{risorse correnti allocate a tutti i thread dal primo fino al } i-1\text{-esimo.}$$

Uno stato *non sicuro* non implica necessariamente la presenza di un deadlock. Una buona tattica è di garantire ad un thread, l'accesso ad una risorsa se e solo se la sequenza si troverebbe comunque in uno stato sicuro. Questa politica garantisce la non-esistenza di condizioni di attese circolari.

4.4.2 Esetensione del RAG

Possiamo pensare ad un nuovo tipo di grafo, considerando un nuovo tipo di arco, ossia un **arco di reclamo**, rappresentato graficamente con una freccia puntinata, ed indica che il thread potrebbe in futuro richiedere quella risorsa. Soddisfare una richiesta quindi, significa trasformare un *arco di assegnamento* in un *arco di reclamo*.

In questo nuovo grafo, un ciclo indica uno stato non sicuro, impedendo ad una risorsa di essere allocata ad un thread, in altra parole, se un thread fa una richiesta, l'arco di reclamo verrà trasformato in un arco di richiesta, e tale thread attenderà, questa soluzione non funziona se vi sono più istanze della stessa risorsa.



Se dessimo a t_4 la risorsa r_2, staremmo introducendo un possibile ciclo, entrando in uno stato non sicuro. Se a t_3 dessimo r_2, non staremmo introducendo nessun ciclo, restando quindi in uno stato sicuro.

4.4.3 L'algoritmo del Banchiere

Vediamo adesso uno dei più noti, se non il più noto, algoritmo di rilevamento di stati unsafe. L'algoritmo del banchiere, gestisce anche i casi in cui vi sono più istanze della stessa risorsa, "costringe" i thread a fornire informazioni su quali risorse potrebbero richiedere, si occupa di

allocare risorse, se e soltanto se, tale allocazione lascierà il sistema in uno stato safe.

Vediamo da cosa è composta la struttura dati sulla quale l'algoritmo opera :

- `n` = un valore intero rappresentante il numero dei thread.
- `m` = un valore intero rappresentante il numero delle risorse (escluse istanze).
- `available[m]` = un vettore di lunghezza m, che rappresenta quante istanze vi sono di una specifica risorse. Il j-esimo elemento, conterrà il numero di istanze che la j-esima risorsa ha.
- `max[m][n]` = una matrice $n \times m$, in cui, l'elemento (i, j) rappresenta il numero di risorse del tipo j, che il thread i-esimo potrebbe richiedere.
- `allocation[m][n]` = una matrice $n \times m$, dove l'elemento i, j rappresenta il numero di risorse di tipo j allocate all'i-esimo thread.
- `need[m][n]` = una matrice $n \times m$, dove l'elemento i, j rappresenta il numero di risorse di tipo j, che il thread i-esimo potrebbe richiedere ulteriormente per completare la sua esecuzione.

L'algoritmo performa due operazioni :

- `isSafeState` - Dato lo stato corrente, controlla se esso è sicuro.
- `requestResource` - Dato un thread, richiedente una certa risorsa, controlla se tale richiesta può essere compiaciuta.

Apputno sulla Notazione : In seguito verrà fatto riferimento a righe di codice, seguiranno le seguenti notazioni, se `a` e `b` sono due vettori, allora `a+b` indicherà che ad ogni elemento i-esimo di `a` aggiungo l'elemento i-esimo di `b`. `a+5` indicherà che ad ogni elemento di `a` aggiungo `5`. `a<b`, o qualsiasi altra operazione di comparazione, risulterà vera se ogni i-esimo elemento di `a` sarà minore ad ogni i-esimo elemento di `b`.

La richiesta sarà compiaciuta se porterà ad uno stato sicuro, quindi la seconda operazione, utilizzerà l'output della prima per decidere se operare o no. Vediamo il suo funzionamento diviso nelle due operazioni :

`isSafeState()`

1. Si inizializzano due vettori di rispettive lunghezze m ed n, il vettore `work`, inizializzato identico ad `available`, ed il vettore booleano `finish`, con tutti gli elementi inizializzati a `false`, se l'elemento i-esimo sarà uguale a `true`, ciò vorra dire che tale thread avrà finito la sua esecuzione.
2. Si ricerca un certo elemento di indice i tale che, `finish[i]=false` $\wedge$ `need[i]`$\leq$ `work`, se non esiste tale i, allora si salta allo step 4.
3. Una volta trovato l'elemento i che soddisfa il punto 2, si esegue rispettivamente :
   (a) `work=work+allocation[i];`
   (b) `finish[i]=true`

4. Se ogni elemento di `finish` è uguale a `true`, allora il sistema è in uno stato sicuro.

`requestResource()`
Riceve in input un indice i rappresentante un thread ed un vettore m dimensionali delle richieste del vettore.

1. Se `request>need[i]`, genera un errore in quanto quel thread sta provando a richiedere più risorse di quante ne abbia reclamate, altrimenti, si vada allo step 2.

2. Se `request>available`, il thread dovrà attendere affinché la risorsa torni disponibile, altrimenti si vada allo step 3.

3. Anche se la risorsa è disponbile, si controlla che il possibile nuovo stato sia ancora sicuro, simulando le operazioni :

   (a) `available=available-request`

   (b) `allocation[i]=allocation[i]+request`

   (c) `need[i]= need[i]-request`

   Dopo tali operazioni, si controlla `isSafeState()`, se no, tali operazioni verranno annullate, altrimenti, verranno convalidate.

5 La Memoria

Durante l'introduzione del corso, si è già fatto riferimento a quanto la gestione memoria risulti di cruciale importanza per quanto riguarda l'efficienza di un sistema operativo. Un buon OS, deve massimizzare l'utilizzo della memoria, e garantire sicurezza ed isolamento di essa fra i deiversi processi che la co-abitano (un processo, non dovrebbe essere in grado di accedere ad indirizzi di memoria riservati ad altri processi, a meno che essi non lavorino su una zona condivisa).

Il supporto fisico della memoria è, ovviamente finito, esiste un meccanismo di astrazione, implementato dall'OS, noto come *memoria virtuale*, che permette al programmatore di avere l'illusione che la memoria disponibile sia *infinita*, senza che esso debba preoccuparsi di gestire gli indirizzi e preoccuparsi di allocare i processi.

Quando si scrive del codice con un linguaggio di programmazione di alto livello come il *C*, si utilizzano costrutti e simboli, come `+`, `if`, e si definiscono variabili, ad esempio `int x = 1`, tali variabili, risiederanno sulla memoria principale, come le istruzioni del codice, ma sarà l'OS a decidere in che indirizzo specifico saranno allocate, non il programmatore.

Tale traduzione degli indirizzi sarà opera di un insieme di strumenti forniti dall'OS, come il compilatore, l'assembler ed il linker, sarà l'OS ad occuparsi di caricare il programma, dal disco alla memoria principale.

Quando un programma viene compilato però, gli indirizzi alla quale farà riferimento, sono **logici/virtuali**, ossia non sono i reali indirizzi fisici/reali alla quale risponde la *RAM*, gli indirizzi logici a disposizione sono in numero di gran lunga superiore a quelli reali, verrà eseguita poi, un operazione di traduzione fra indirizzi virtuali ad indirizzi fisici.

5.1 Address Binding

L'operazione di traduzione è nota con il nome di *address binding*, ed essa può avvenire in diversi momenti prima dell'esecuzione del codice.

- **compile time** - Quando il programma viene compilato, gli indirizzi ad esso riservati saranno direttamente quelli fisici, tale codice sarà *assoluto*, e non vi sarà intervento dell'OS, gli indirizzi logici saranno identici a quelli reali. Per cambiare gli indirizzi, il programma va ricompilato.

- **load time** - L'indirizzo di partenza k di un programma, non viene generato al momento della compilazione, bensì, saranno generati degli indirizzi che fanno riferimento all'offset k. (Ad esempio, si dia il caso che a 3 variabili del programma vengano assegnati gli indirizzi 1,5 e 7. Al momento del caricamento, verrà generato un indirizzo k, e tali variabili verranno allocate negli indirizzi $1 + k$,$5 + k$ e $7 + k$). Tale codice è detto *staticamente rilocabile*, una volta caricato il programma in memoria, l'OS determinerà l'indirizzo fisico di partenza k. Per cambiare gli indirizzi, il programma va ricaricato in memoria.

- **execution time** - Il compilatore, genera un codice *dinamicamente rilocabile*, con degli indirizzi totalmente virtuali, il programma potrà "muoversi" nella memoria principale durante la sua esecuzione. Il sistema operativo esegue il mapping dinamico fra indirizzi virtuali e fisici tramite un unità hardware nota come *Memory Managment Unit* (MMU). Risulta essere la soluzione più flessibile adottata dalla maggiorparte dei sistemi operativi moderni.

Se ci trovassimo in un ambiente uniprogrammato, non dovremmo preoccuparci di riallocare la memoria, in quanto vi sarà un singolo processo in memoria alla quale è riservato uno spazio contiguo (escludendo lo spazio dedicato all'OS).

Differisce però dal caso reale, in cui la maggiorparte dei sistemi sono multiprogrammati, ed i programmi devono condividere la memoria. Ci si pongono 3 principali obbiettivi nella gestione della memoria in un ambiente multiprogrammato :

1. **Condivisione e Trasparenza** - Più processi condividono la stessa area di memoria, ma non devono "porsi" tale problema, ne tanto meno preoccuparsi degli indirizzi fisici nella quale sono allocati, devono essere eseguiti in maniera trasparente, lasciando il compito della gestione all'OS.

2. **Protezione e Sicurezza** - Diversi processi non devono essere in grado di corrompere aree di memoria dedicate ad altri processi o al sistma operativo, e non devono essere in grado di leggere dati destinati ad altri processi.

3. **Efficienza** - Le performance della CPU e della memoria non devono essere intaccate dalla condivisione dei processi.

5.1.1 Rilocazione Statica

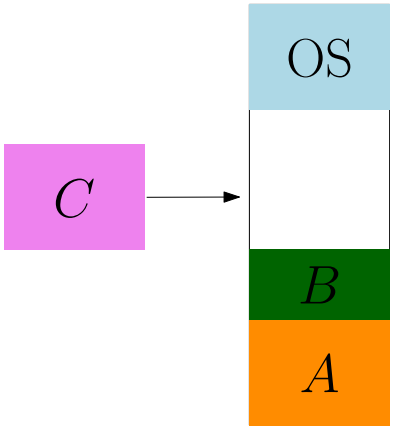
Vediamo un idea iniziale di rilocazione statica, si assuma che il sistema operativo, sia allocato nella parte superiore della memoria, e che gli indirizzi generati da ogni processo utenti vadano da 0, fino a $TOT - OSMEM - 1$, dove TOT= Dimensioni totali della memoria, e
$OSMEM$= Memoria destinata all'OS. I processi vengono allocati nel primo segmento contiguo di memoria disponibile. È garantita la condivisione e la trasparenza, dato che i processi possono

essere allocati in qualsiasi zona della memoria, senza preouccuparsene.

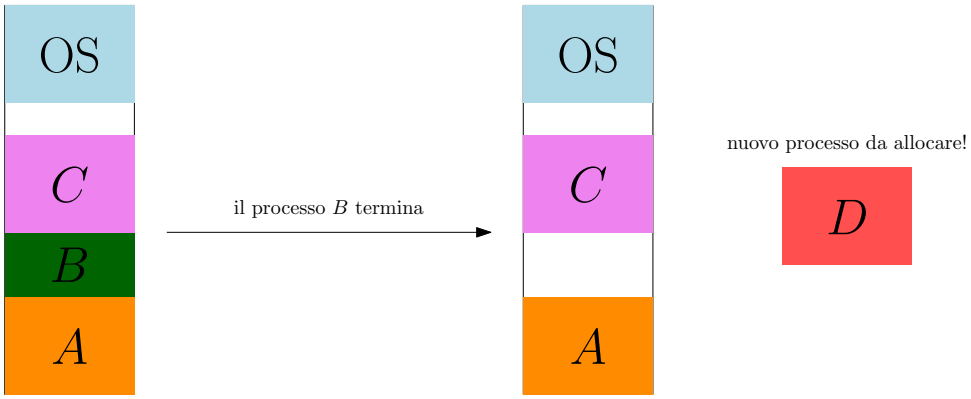
Si adotta un modello di *load time binding*, l'OS decide l'indirizzo di partenza di un processo nel momento in cui viene caricato in memoria.

- **Pro** : Non è necessario alcun supporto hardware aggiuntivo.
- **Contro** : Non vi è protezione, ed un processo può corrompere la memoria di altri processi, gli indirizzi per i processi sono contigui, quindi un processo non può essere "frammentato" fra diversi processi. Una volta allocato un processo, esso non potrà essere spostato dall'OS.

Vediamo un esempio che renda chiari i contro appena elencati, si consideri la seguente memoria, dove sono caricati i processi A e B (ed ovviamente, l'OS), ed è in procinto di essere caricato un processo C :



Il processo C viene caricato, ma il processo B termina, lasciando quindi uno spazio di memoria libero :



Il problema è il seguente, deve essere allocato un nuovo processo D, esso però, può essere allocato esclusivamente in uno spazio contiguo, sebbene la memoria libera disponibile è necessaria per allocare D, essa non è contigua, e D non può essere frammentato fra OS-C e C-A. Inoltre, tale sistema non è in grado di spostare i processi in memoria, quindi D non potrà essere allocato.

5.1.2 Rilocazione Dinamica

Tale rilocazione garantisce la protezione, richiede però il supporto hardware della MMU. Si adotta un modello di *execution time binding*, la MMU traduce dinamicamente ogni indirizzo

logico/virtuale generato dal processo nel corrispettivo indirizzo fisico.

La MMU ha due registri a disposizione, che sono poi correlati ad ogni processo, essi sono i registri `base` e `limit`, già accennati nel capitolo 2.1. `base` , contiene l'indirizzo fisico di partenza associato ad un processo, mentre `limit`, lo spazio che occupa, ogni processo avrà quindi ad esso associata un range di memoria [`base`, `base+limit`]. Quando un processo vuole utilizzare un indirizzo, si controlla se tale indirizzo rientra in questo range, garantendo così protezione.

Aggiungere immagine alla slide 121

- **Pro** : Fornisce protezione degli spazi di memoria, l'OS può spostare i processi durante l'esecuzione, ed essi possono crescere dinamicamente, l'implementazione della MMU è semplice.
- **Contro** : I processi continuano a dover essere allocati in maniera contigua, con possibile spreco di memoria, il numero di processi che possono coesistere è limitato dallo spazio della RAM, ed i programmi non possono condividere lo stesso indirizzo di memoria per comunicare.

La condivisione, la trasparenza e la protezione sono garantiti, l'unico problema da risolvere rimane l'efficienza, in quanto spostare un processo può risultare lento.