


Emiddio Ingenito - Sistemi Operativi - Parte A

 Autore	Emiddio Ingenito @emikodes
--	----------------------------

Modello di Von Neumann:

Esecuzione sequenziale

Attività asincrone e interrupt

Memoria:

Protezione della memoria

Periferiche

Attesa della periferica

Trasferimento dati

Periferica mappata in memoria

Reti informatiche

Mainframe

Sistemi operativi - Struttura e Organizzazione logica

Gestione dei processi

Gestione delle periferiche

Gestione del file system

Gestione dell'UI

Architetture dei sistemi operativi

Sistema Monolitico

Sistema a Strati

Microkernel

Struttura Modulabile (Modular Kernel)

Sistema a Macchine Virtuali (VM)

Generazione del sistema operativo

Avviamento del sistema operativo

Boot "one step"

Boot "two steps"

Primary bootstrap

Secondary bootstrap

Boot "three steps"

Interfacce utenti

Interfacce programmatiche

Multi-tasking

Multi-programmazione

Processo

Salvataggio processo

Process Control Block (PCB)

Stati di un processo

Modelli di computazione

Terminazione di un processo

Realizzazione del multi tasking

Politiche di sospensione dei processi

Politiche di sospensione dei processi nel time-sharing

Context Switching

Thread

Schedulazione (processi e threads)

Short-term-scheduler (o CPU scheduler)

Medium-term-scheduler

Long-term-scheduler

Rilascio dell'utilizzo della CPU

Scelta di Algoritmi di schedulazione

Algoritmi di scheduling

FCFS (First-Come-First-Served)

SJF (Shortest-job-first)

Priorità

Round-Robin (RR)

Coda a più livelli C+L

Schedulazione in sistemi multiprocessore

Schedulazione real-time

Hard real-time

Soft real-time

Schedulazione dei thread

Thread user-level

Thread kernel-level

Processi cooperanti

IPC a comunicazione diretta

IPC a comunicazione indiretta

Processi Indipendenti e Mutua esclusione

Sezione critica

Problema del produttore-consumatore

Sincronizzazione dei processi concorrenti

Approcci a livello istruzioni:

Variabili di turno

Variabili di lock

Approcci a livello del S.O

Semafori

Semafori generali

Monitor

Starvation

Deadlock

Grafo di allocazione delle risorse

Metodi di gestione del deadlock

Prevenzione del deadlock

Evitare il deadlock

Rilevamento e ripristino del deadlock:

Modello di Von Neumann:

Schema concettuale che permette di descrivere un elaboratore generico.

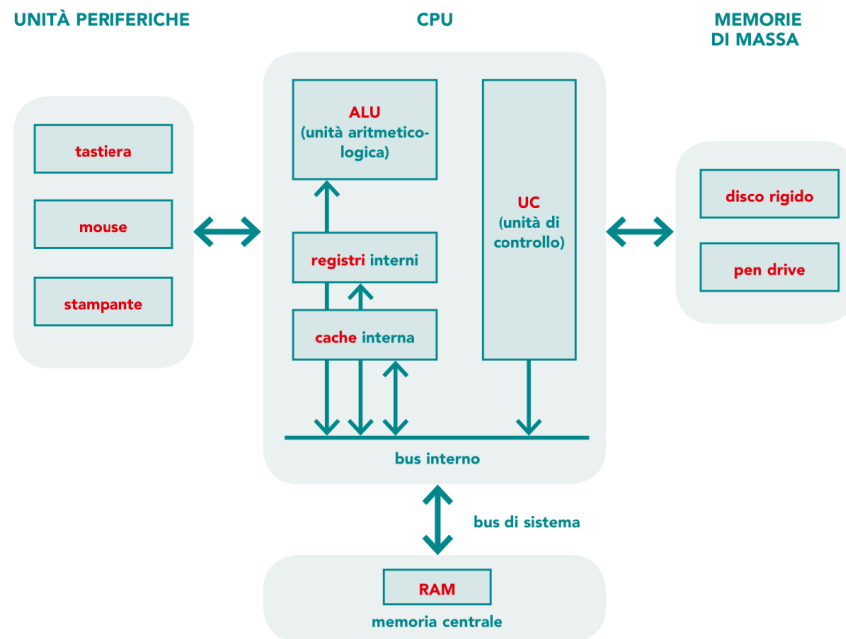
Ogni elaboratore deve avere i seguenti componenti, indipendentemente dal modo in cui ogni produttore li implementa:

- CPU - Unità di elaborazione col compito di eseguire istruzioni in sequenza (operazioni matematiche, logiche, accesso a memoria...)

E' composta da:

- UC - Supervisore dell'esecuzione
- ALU - Esecutore di calcoli logici o aritmetici
- Registri - piccole locazioni di memoria estremamente performanti.
- Memoria - a cui la CPU accede in lettura o scrittura, attraverso una struttura di connessione detta "bus" composta da piste elettriche ricavate sulla piastra madre, suddivisa in blocchi di bit dette "word", a 16,32 o 64 bit.
- Periferiche - Dispositivi fisici esterni, direttamente connessi alla motherboard, che permettono interazioni esterne.
 - Periferiche di I/O

- Periferiche “di massa” (HDD/SSD/Flash Drive), prevedono fenomeni fisici di consumo, oltre a velocità di risposta estremamente ridotte rispetto alle memorie alimentate.



Esecuzione sequenziale

La CPU, per vincoli strutturali e progettuali, permette la sola esecuzione in sequenza.

Cioè, partendo da un indirizzo di memoria, la CPU esegue le istruzioni nell'ordine in cui sono scritte in memoria centrale.

I linguaggi di programmazione moderni, introducono però numerosi costrutti non sequenziali, come cicli, IF, che generano variazioni nel flusso di esecuzione del codice.

Linearizziamo quindi queste strutture, introducendo dei salti.

Esempio di linearizzazione di un costrutto di tipo IF-ELSE (Come scrivevo in assembly MIPS, in questo modo la CPU potrà semplicemente leggere dall'alto verso il basso, valutare, ed eseguire):

High-level code

```
if (i == j)
    f = g + h;
else
    f = f - i;
```

MIPS assembly code

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
    bne $s3, $s4, L1
    add $s0, $s1, $s2
    j    done
L1:   sub $s0, $s0, $s3
done:
```

Stessa cosa viene effettuata con i loops, linearizzati come in MIPS, con sequenze di condizioni e salti.

Attività asincrone e interrupt

La CPU, durante l'esecuzione del codice, deve poter considerare anche eventuali attività asincrone, cioè input derivanti dall'esterno.

Introduco allora un meccanismo, gestito dall'hardware, che permette di interrompere il normale ciclo di esecuzione del codice, in corrispondenza di eventi esterni.

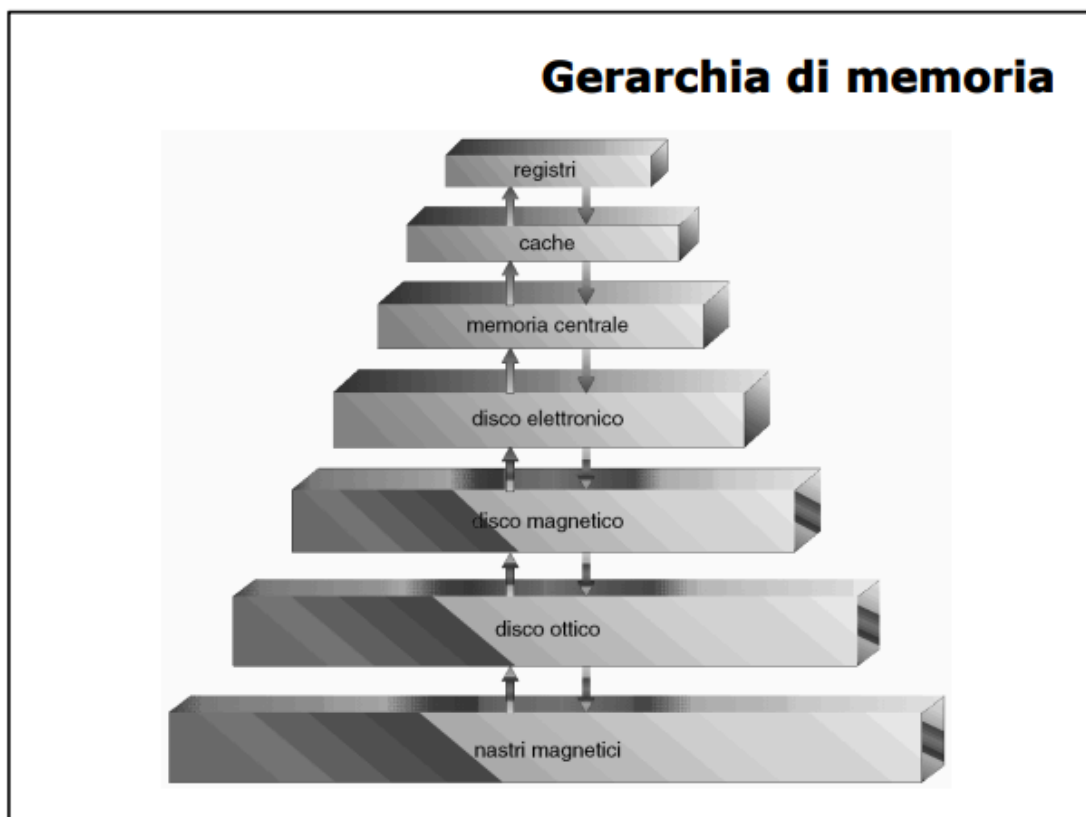
Il meccanismo di interruzione, è come fosse un "sensore", che rileva la presenza di eventi esterni, e richiama l'attenzione della CPU, che interrompe la normale

esecuzione, salva in EPC l'indirizzo della "offending instruction", e procede alla gestione dell'eccezione generata.

Gli interrupt sono interruzioni hardware, mentre le eccezioni sono interruzioni software.

L'Interrupt, come visto in MIPS, non è altro che una chiamata a procedura di "error handling", che avviene in modo asincrono.

Memoria:



Dall'alto verso il basso, gli accessi diventano più lenti, le memorie diventano meno costose.

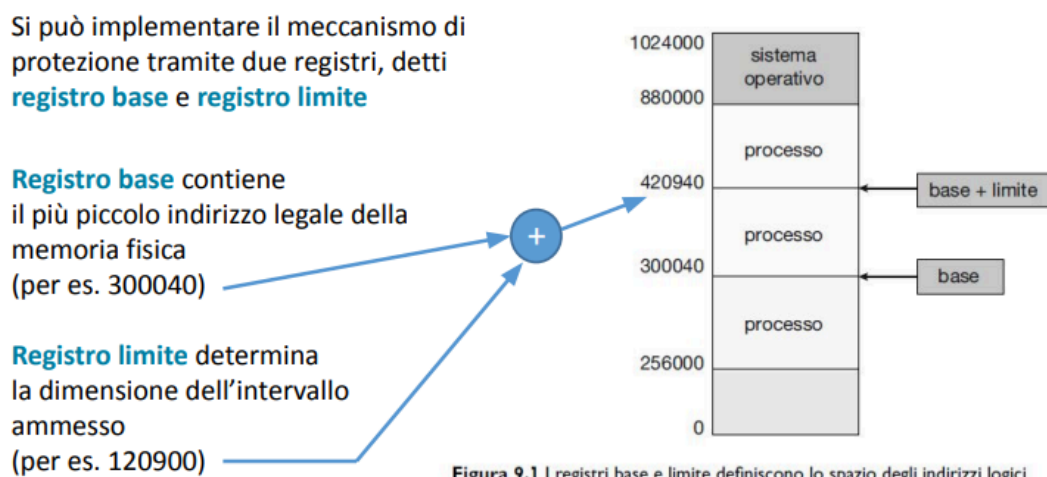
Registri, cache, memoria centrale, dischi elettronici (SSD) , dischi magnetici, dischi ottici, permettono accesso diretto.

Le memoria a nastri magnetici, permettono il solo accesso sequenziale.

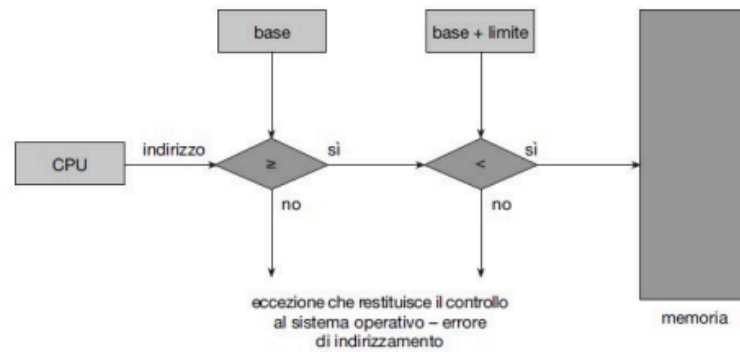
Protezione della memoria

Per garantire la protezione dei dati, viene assegnata ad ogni processo in esecuzione un'area di memoria dedicata.

Per far si che la CPU acceda solo allo spazio di indirizzamento di un processo specifico, viene introdotto un meccanismo di protezione della memoria, implementato a livello hardware per motivi prestazionali.



Qualsiasi tentativo da parte di un programma eseguito in modalità utente di accedere alle aree di memoria riservate al sistema operativo o a una qualsiasi area di memoria riservata ad altri utenti comporta l'invio di una **eccezione** (*trap*) che restituisce il controllo al sistema operativo che, a sua volta, interpreta l'evento come un errore fatale.



Periferiche

La comunicazione dell'elaboratore con le periferiche, avviene per mezzo di elettroniche di controllo, direttamente connesse alle porte di I/O dell'elaboratore e della periferica.

L'insieme delle due elettroniche di comunicazione, direttamente collegate all'elaboratore e alla periferica, e del mezzo trasmissivo che connette le due elettroniche (elaboratore al pc), è detto **canale di comunicazione**.

All'interno dell'interfaccia dell'elaboratore, sono presenti 3 registri:

- Stato - Per verificare che la periferica non sia in utilizzo
- Comando
- Dati

La CPU, legge tramite il bus I/O, il contenuto del registro Stato.

Se la periferica è libera, porrà nel registro Dati, due informazioni

- Codice dell'operazione da effettuare sulla periferica (esempio, nel caso di periferiche di massa: lettura o scrittura)

- Indirizzo del blocco di memoria su cui operare.

Modifica poi il registro Comando, per indicare all'interfaccia di inviare il contenuto del registro Dati all'elettronica di controllo (controller) della periferica.

L'elettronica di controllo della periferica, riceve il contenuto del registro dati, ne esegue quindi il fetch, poi decodifica l'operazione da effettuare, e la esegue.

Nel caso di un disco magnetico, modifica i segnali di controllo in modo da continuare a tenere i motori in movimento, sposta le testine a leggere il blocco indicato, legge il contenuto, lo restituisce tramite lo stesso canale di comunicazione all'interfaccia dell'elaboratore.

L'interfaccia dell'elaboratore, porrà poi nel registro stato, un codice, per indicare che l'operazione è andata a buon fine.

Attesa della periferica

Quando la CPU dovrà iniziare la comunicazione con una periferica, esegue come prima operazione, la lettura del registro Stato dell'interfaccia dell'elaboratore.

Questa lettura, può essere effettuata in maniera:

- Attiva - Quando un processo richiede una periferica, il S.O. effettua periodicamente una lettura del registro stato dell'interfaccia, e ne notificherà la disponibilità al processo.

Nel frattempo, il processo rimane in attesa.

- Interruzione - GUARDARE

Trasferimento dati

Il Trasferimento dati tra la CPU e l'interfaccia della periferica dell'elaboratore (Dalla RAM alla periferica), può avvenire secondo:

- Parole
- Blocchi

Periferica mappata in memoria

Possiamo scegliere di inserire una locazione di memoria, direttamente all'interno della scheda di interfaccia dell'elaboratore, in modo che la CPU scriva direttamente su di essa.

Reti informatiche

Concettualmente funzionano esattamente come una periferica, la comunicazione in rete, a livello del S.O, avviene come fosse su un disco magnetico.

L'interfaccia di rete, è strutturata come nell'esempio prima, con registri stato, comando e dati.

La comunicazione, avviene per blocchi di dati, non per parole.

Il canale di trasmissione, sarà formato dalla NIC di tutti i dispositivi connessi nella rete, i cavi, l'etere, i dispositivi intermedi, che variano in base alla topologia di rete implementata.

Mainframe

Architettura orientata all'esecuzione di programmi non interattivi, cioè che non richiedono input esterni.

Tali programmi, vengono eseguiti come fossero memorizzati in una coda, uno alla volta.

L'elaborazione può essere monoprogrammata (il processore porta a termine un processo, prima di iniziarne un'altro) o multiprogrammata (il processore può operare su altri processi, sfruttando i momenti morti dell'esecuzione).

Alcune versioni più moderne dei mainframe, permettono l'interazione con più utenti contemporaneamente.

Più terminali (schermi e tastiere), direttamente connessi al mainframe, che operano asincronamente.

Dal punto di vista dell'utente, ogni "flusso di attività", deve avvenire come se si stesse usando un PC tutto per se, cioè il mainframe deve permettere a tutti i processi di eseguire "un pò alla volta", senza priorità.

Sistemi operativi - Struttura e Organizzazione logica

Garantisce:

- Astrazione - dal punto di vista dell'applicativo, l'elaboratore appare "semplificato", come fosse internamente fatto secondo la macchina di Von Neumann.

Le richieste di risorse avvengono "senza preoccuparsi" di come esse "sono fatte" o come sono implementate (indipendentemente dal modello, marca..)

- Virtualizzazione - ogni applicativo, è eseguito in un'immagine del sistema, ad esso dedicata.

Dal punto di vista del SW quindi, ogni app agisce come se le risorse fossero dedicate tutte a se, non è a conoscenza di hazard, concorrenza, scheduling, eventuali inconsistenze di dati (tutti fenomeni, dati dalla concorrenza di più processi per l'utilizzo delle stesse risorse).

I programmi sviluppati risulteranno più semplici, il programmatore non deve gestire tali sincronismi.

Dall'hardware, il sistema operativo, ne astrae le risorse:

- Gestione CPU - viene ridondata logicamente, creando più "istanze virtuali" del processore.
- Gestione Memoria Centrale

Garantendo:

- Multi-programmazione (tramite memory protection, ogni processo, vedrà solo una porzione di memoria centrale, tramite meccanismi di protezione. Il resto della memoria, risulterà inaccessibile),
- Allocazione e deallocazione della memoria ai processi
- Caricamento e scaricamento dei processi in memoria

Se viene generato un indirizzo "al di fuori" dell'area riservata, il sistema operativo bloccherà le operazioni di lettura e scrittura su di esso.

Le operazioni di accesso alle periferiche, vengono omogeneizzate dal sistema operativo, in un modo che sia identico indipendentemente dalla sorgente (una tastiera, una NIC, un disco).

Ogni periferica, è allora astratta in un **file**, così che si possa accedere a qualsiasi periferica, utilizzando due uniche funzioni "read" e "write".

▼ "Everything is a file"

Seeing everything as a file is actually a clever design decision made by the creators of Unix. This allows the same set of APIs to be used across different tasks. When you open a file-like entity, it returns a file descriptor. A file descriptor is a unique integer that represents an open file. Standard input has the file descriptor of `0`, while standard output has the file descriptor of `1`. Several system calls in Unix work with file descriptors.

The example below uses the write system call to write **Hello, world!** to standard output.

```
#include <unistd.h>

int main() {
    write(1, "Hello, world!\n", 14);
    return 0;
}
```

The write system call can also be used to write to a file on disk.

```
#include <fcntl.h>
#include <unistd.h>

int main() {
    // Get the file description
    int fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR |
S_IWUSR);

    // write to the file
```

```

write(fd, "Hello, World!\n", 14);

// close the file
close(fd);

return 0;
}

```

The program above opens a file called `output.txt`. Then writes hello world to it. After that it closes the file using the close system call. As you can see both writing to standard output and writing to a file use the same function.

The code snippet below shows a C function that makes a request to a server. It does this by writing to the client socket. Which is file-like because it returns a file descriptor when created.

```

void send_http_request(int client_socket, const char *host, const char *path){
    // Create the HTTP request
    char request[MAX_BUFFER_SIZE];
    snprintf(request, sizeof(request), "GET %s HTTP/1.1\r\nHost: %s\r\n\r\n", path, host);

    // send the HTTP request to the server
    write(client_socket, request, strlen(request));
}

```

The function below reads from the socket. The loop is used to receive the streams of data as they come.

```

void receive_http_response(int client_socket){
    // Recieve and print the HTTP response from the server
    char response[MAX_BUFFER_SIZE];
    ssize_t bytes_received;

    // Read response streams

```

```
while ((bytes_received = read(client_socket, response, sizeof(response)
- 1)) > 0 ) {
    // Add string termination
    response[bytes_received] = '\0';
    printf("%s", response);
}
}
```

A file is any resource that streams its data content. This content can be read from or written to. A lot of resources on Unix-like Operating systems follow this pattern and that is why **everything is a file**.

Gestione dei processi

- Creazione e terminazione
- Sospensione e riattivazione
- Schedulazione
- Sincronizzazione
- Gestione stalli
- Comunicazione

Gestione delle periferiche

- Configurazione e inizializzazione
- Interfaccia generale e omogenea (astrae le periferiche come fossero files, che verranno poi gestiti dal Gestore del file system)

- Gestione ottimizzata
 - Protezione
 - Bufferizzazione
 - Caching
-

Gestione del file system

- Creazione e cancellazione
 - Lettura e scrittura
 - Copiatura
 - Ricerca
 - Protezione e sicurezza
 - Backup e ripristino
-

Gestione dell'UI

- Interprete dei comandi a livello utente (trasforma click in comandi)
 - Interprete dei comandi a livello programmi
 - Librerie
 - Autenticazione
 - Gestione errori e malfunzionamenti
-

Architetture dei sistemi operativi

Un sistema operativo moderno, deve essere progettato con cura, per funzionare correttamente e poter essere facilmente modificato.

A tal fine, i sistemi operativi sono stati sviluppati con approcci che variano dal modello monolitico, al sistema a strati, fino a modelli più modulari e microkernel.

Ogni approccio ha i propri vantaggi e svantaggi, che vanno considerati a seconda dei requisiti del sistema.

Sistema Monolitico

Il sistema monolitico è una delle prime architetture per i sistemi operativi.

Tutte le funzionalità del sistema operativo sono integrate in un singolo programma che esegue in un unico spazio di indirizzamento.

Non vi è alcuna separazione tra i vari moduli: il kernel integra i gestori della CPU, memoria, periferiche e l'interfaccia utente.

Questo approccio ha come vantaggio principale l'efficienza: poiché tutto il sistema operativo è eseguito in un unico spazio, le comunicazioni tra le diverse parti del sistema sono molto veloci, senza la necessità di passare attraverso meccanismi di comunicazione complessi. Inoltre, è compatto e veloce, il che lo rende adatto per sistemi semplici dove non sono richiesti frequenti aggiornamenti.

Tuttavia, la mancanza di modularità rende la manutenzione e l'espansione del sistema molto difficili, richiedendo la modifica dell'intero kernel.

Se si presenta un bug in una parte del sistema, questo potrebbe compromettere l'intero sistema operativo.

Sistema a Strati

Il sistema operativo è suddiviso in livelli funzionali, dove ogni livello è responsabile di una porzione del sistema. Il livello più basso corrisponde all'hardware, e i livelli superiori forniscono funzionalità via via più complesse, avvalendosi delle funzioni dei livelli inferiori.

Un vantaggio di questo approccio è che semplifica lo sviluppo e il debugging. Ogni livello verrà testato dai livelli superiori, se si trova un errore in un livello,

andrà corretto il solo codice relativo a quello specifico.

Un primo problema di questa architettura è che la separazione tra i vari livelli non è sempre chiara, rendendo difficile definire le interazioni.

Inoltre, ogni richiesta deve attraversare più strati per poter essere evasa, riducendo le prestazioni del sistema.

Microkernel

Solo le funzioni essenziali (come la gestione dei processi, la memoria e la comunicazione) sono mantenute nel kernel. Tutti gli altri servizi, come il file system, i driver, lo scheduler, vengono eseguiti come processi utente separati, al di fuori del kernel. La comunicazione tra il kernel e i servizi avviene tramite scambi di messaggi.

Il principale vantaggio di un microkernel è la modularità. Poiché la maggior parte dei servizi è eseguita a livello utente, è facile estendere il sistema operativo e aggiungere nuovi servizi senza modificare il kernel. Inoltre, l'affidabilità e la sicurezza sono migliorate, poiché i fallimenti dei servizi utente non influiscono sul kernel.

Tuttavia, poiché i servizi devono comunicare tramite messaggi tra spazi di indirizzamento separati, si introduce un notevole overhead che può influire negativamente sulle prestazioni.

Struttura Modulabile (Modular Kernel)

Un approccio più recente, che si avvicina sia al modello monolitico che al microkernel, è la struttura modulare. Il kernel ha un set minimo di funzioni di base e può caricare dinamicamente nuovi moduli sia al momento del boot che durante il funzionamento del sistema. Ciò consente una maggiore flessibilità rispetto ai sistemi monolitici pur mantenendo una buona efficienza.

I moduli possono essere caricati e scaricati in tempo reale, migliorando l'estendibilità del sistema senza compromettere le prestazioni generali. Ad esempio, un modulo potrebbe essere caricato per supportare un nuovo tipo di file system, senza la necessità di modificare il codice del kernel. I sistemi come Linux utilizzano questa architettura, consentendo una gestione dinamica delle risorse.

Sistema a Macchine Virtuali (VM)

Un'ulteriore evoluzione è rappresentata dai sistemi a macchine virtuali (VM), in cui vi è la presenza di più macchine virtuali, che eseguono il proprio sistema operativo come se fossero macchine fisica separata. Il kernel emula l'hardware e gestisce l'isolamento tra le varie VM, che possono eseguire anche sistemi operativi diversi.

La gestione delle risorse, come la CPU e la memoria, è condivisa tra le VM (Unico HW sottostante), ma ogni VM è isolata dalle altre, garantendo elevata sicurezza.

Tuttavia, la virtualizzazione introduce un overhead significativo, poiché le richieste di I/O devono passare attraverso vari livelli di gestione, riducendo le prestazioni complessive.

Generazione del sistema operativo

Per generare il sistema operativo, devo eseguire una serie di operazioni:

- **Valutare l'ambiente in cui esso dovrà essere eseguito**

Che tipo di applicazioni andranno eseguite? (se note a priori)

Esempio: software di grafica, giochi, software di produzione musicale...

Ciò è utile per capire quale sarà il carico di lavoro prodotto da ogni applicativo, come vengono utilizzati, che prestazioni devo aspettarmi dal sistema operativo.

Quali interazioni avranno con le periferiche?

Ci sono applicazioni con "livello di priorità" più elevato? Oppure possono essere considerate tutte con turnazione omogenea?

- **Definizione dei parametri del sistema operativo**

Dall'analisi precedente, definirò i valori dei parametri specifici per l'installazione del sistema operativo, per un determinato ambiente, in modo da ottenere prestazioni massime, compatibilmente col carico di lavoro identificato nella fase precedente.

- **Applicazione dei parametri**

Modifica dei file di configurazione del sistema operativo, oppure modifico direttamente il codice sorgente del SO, e nuova generazione del codice eseguibile "ad hoc" per la specifica installazione

- **Memorizzazione del sistema operativo nel sistema di elaborazione**
-

Avviamento del sistema operativo

Boot "one step"

La memoria centrale, prevederà una porzione a sola lettura (ROM), e una a lettura randomica (RAM)

Il Sistema operativo, sarà memorizzato nella ROM (Memoria persistente, a sola lettura).

Questo mi permetterà di avere accensioni fulminee, perchè il codice del sistema operativo sarà disponibile fin da subito (il processore, semplicemente inizierà a caricarne le istruzioni una ad una, partendo da un indirizzo prefissato), ma allo stesso tempo, il esso non sarà modificabile, poichè memorizzato in una memoria a sola lettura (ci sono varianti che prevedono uso di eprom, per cui posso cambiare il sistema operativo, "riflashando" la memoria)

Boot "two steps"

Primary bootstrap

Una ROM contiene il "bootloader", una funzione elementare che inizializza la periferica dov'è contenuto il sistema operativo (SSD, HDD, USB...) e sapendo da che traccia del disco partire, lo carica nella memoria centrale.

Questo mi permette di mantenere modificabile il sistema operativo, memorizzato su disco.

Secondary bootstrap

Il bootloader ha correttamente scaricato il sistema operativo dal disco nella RAM, ne concede il controllo alla macchina.

Boot "three steps"

- In ROM, è memorizzato un caricatore elementare, che inizializza la periferica dov'è contenuto il sistema operativo, e scarica da esso, un ulteriore caricatore "complesso", memorizzato in una posizione nota.
- Il caricatore complesso, scaricato in ram, andrà a caricare in RAM i vari "pezzi" del sistema operativo, man mano che servono durante l'esecuzione.

Limite così lo spazio necessario in RAM, perchè non vado a caricare tutto il sistema operativo (applicazioni incluse) direttamente all'avvio, ma lo carico "in più passi", man mano che diversi componenti sono richiesti.

Interfacce utenti

L'interfaccia che permette l'interazione tra l'utente e il sistema operativo, è detta Shell (guscio, è lo strato più esterno del sistema operativo)

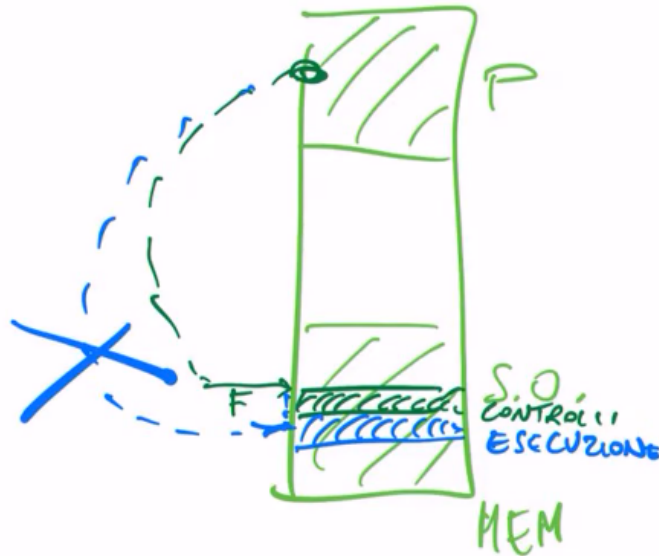
La Shell, Acquisisce comandi (Fetch), ne verifica la correttezza (Decode, cioè se il comando inserito è ammissibile, e i parametri sono inseriti correttamente), e li esegue (Execute, lanciando un nuovo processo asincrono)

Interagisco con la Shell, secondo un interfaccia testuale (TUI) o grafica (GUI).

Esempio: La Windows shell, è l'interfaccia grafica di windows, composta da alcuni elementi cui il Desktop (una finestra full screen, renderizzata dietro tutte le altre finestre), una taskbar, un task switcher (TAB)...

Interfacce programmatiche

Assicura che le system call, avvengano proteggendo il sistema operativo, garantendo l'integrità dei dati.



Quando nel programma P, voglio effettuare la system call alla funzione F, l'interfaccia programmatica fa sì che il "salto", venga effettuato ad una porzione di codice che effettua il controllo sull'esecuzione della funzione F, e che non si possa effettuare il salto diretto al codice della funzione F.

Multi-tasking

Sfruttare al meglio il processore, implica ridurre al minimo i tempi morti (dovuti all'accesso alle periferiche per esempio), che possono generarsi durante l'esecuzione.

Generalmente, i "tempi morti", sono di diversi ordini di grandezza superiori ai tempi di esecuzione delle istruzioni del programma.

Implemento un meccanismo (context switch), per cui un programma, una volta inviato un'ordine di I/O su una periferica, può essere "congelato" nella sua esecuzione, esser messo da parte, così che nel frattempo un altro programma possa proseguire nell'esecuzione.

Questo, in un sistema mono-processore, permette di dare l'illusione all'utente, che più programmi possano avanzare in parallelo, contemporaneamente.

Multi-programmazione

La maggior parte dei processori, può effettuare il fetch dei programmi SOLO dalla memoria centrale, pertanto per far sì che si possa realizzare il meccanismo di "cambio del processo in esecuzione", è necessario che più programmi si trovino in memoria centrale.

Processo

Programma (**entità passiva**) in esecuzione (**entità attiva**), caratterizzato in memoria centrale da diverse porzioni di memoria:

- Spazio del **codice** del programma (immutabile)
- Spazio delle variabili **globali** del programma, che esiste per tutta la durata dell'esecuzione del processo
- Spazio delle variabili "**dinamiche**" (**heap**)
- **Stack**, dove vengono impilati i **frame d'attivazione** delle procedure chiamate (anche le variabili definite nella funzione **main**, sono allocate sullo **stack**, essendo essa una procedura)

E da alcuni registri:

- **Program Counter (PC)**, utilizzato per tenere conto dell'avanzamento della computazione, in quanto il codice è immutabile.
- Stack pointer (**ESP**), Base pointer (**BSP**)
- Valore dei registri intermedi (variabili temporanee introdotte dal compilatore)

Durante l'esecuzione di alcune operazioni complesse, potrebbero essere utilizzati alcuni **registri**, per la memorizzazione di alcuni risultati intermedi

Esempio: Esecuzione dell'istruzione $A = (2 * 3) + (4 * 2)$.

Viene prima calcolato $2 * 3$, posto in un registro, poi $4 * 2$, e memorizzato in un altro registro, poi calcolata la **somma dei prodotti**, e finalmente memorizzato in memoria centrale (**stack**)

Durante il **context switch**, potrebbe essere necessario salvare il valore dei registri, nel caso l'esecuzione fosse stata interrotta prima della fase di WB (esempio: è stato calcolato $2 * 3$, $4 * 2$, ma la somma dei prodotti non è ancora stata salvata sullo stack)

Ovviamente se la fase di **WB** è stata completata, il salvataggio dei registri temporanei può essere omesso.

Un processo, può essere considerato una macchina a stati finiti, in cui ogni stato è rappresentato dall'insieme dei valori del processo in memoria centrale (quanto descritto sopra), e che produce un nuovo stato, in corrispondenza di una nuova istruzione da eseguire (input)

Salvataggio processo

Per poter cambiare il processo in esecuzione, è necessario salvare il suo stato di evoluzione.

Ogni processo esegue in un'area di memoria protetta e ad esso riservata, pertanto, i valori delle variabili globali, locali (**stack**) e dinamiche (**heap**), possono essere modificati solo ed unicamente dal processo stesso.

Ciò significa che durante il cambio di contesto, essendo che la porzione di memoria associata ad un processo può essere modificata solo dal processo ad esso associata, posso omettere il salvataggio della memoria.

Quello che mi serve davvero salvare, è il contenuto dei registri della CPU, che solitamente avviene in blocco (operazione di push del valore di TUTTI i registri della CPU sullo stack del processo stesso)

A Questo punto, ho memorizzato in memoria centrale, l'intero ambiente d'esecuzione del processo, l'unica informazione necessaria per ripristinare l'esecuzione, è il valore dello stack pointer, con cui poi potrò prelevare il PC e tutti gli altri registri della CPU (la porzione di memoria invece, non è stata modificata)

Il valore dello stack pointer, viene salvato nel PCB

Process Control Block (PCB)

Insieme di informazioni sul processo, che permettono al processore di gestire l'esecuzione.

- Process ID (PID)
- Stato del processo
- Stack pointer
- Informazioni per la schedulazione della CPU
- Limiti della porzione di memoria assegnata al processo (Indirizzo di inizio e fine)
- Informazioni sullo stato dell'I/O (risorse utilizzate, files aperti..)
- Informazioni per l'accounting

L'informazione sullo stato del processo, è utilizzata dal S.O per raggruppare i vari processi in diverse code, una per ogni stato

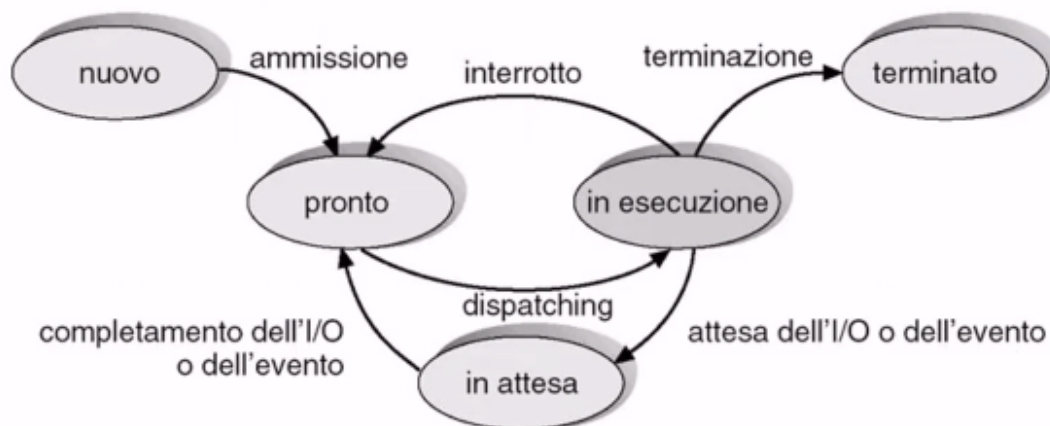
Stati di un processo

Durante la computazione, un processo può:

- **Evolvere**, utilizzare il processore
- **Attendere** l'uso del **processore**, pur avendo tutte le altre risorse fisiche ed informative (dati)
- **Attendere** una risorsa **informativa o fisica**

Ciò, dal punto di vista del processore, si traduce in alcuni stati che il processo può assumere:

- **New** - Creazione del processo
- **Running** - Il sistema operativo ha concesso l'utilizzo del processore
- **Waiting** - Attesa di una risorsa esterna al processore (informativa o fisica)
- **Ready-To-Run** - Attesa dell'uso del processore
- **Terminated** - Il processo viene rimosso dalla memoria centrale



Modelli di computazione

Al momento della scrittura del codice dell'applicazione, posso scegliere diversi approcci di sviluppo

- Processo monolitico - Unico processo che esegue tutte le attività dell'applicazione, può andar bene in caso la nostra applicazione non richieda di eseguire più operazioni contemporaneamente

Esempio: se sto sviluppando un'applicazione di controllo di un nastro trasportatore, io voglio poter sviluppare un'applicazione che monitori il nastro, ma che allo stesso tempo possa ricevere input da remoto, per la modifica dei parametri di produzione.

Allora questo modello introduce dei limiti, se volessi gestire entrambe le "funzionalità" con un solo processo, allora ogni N secondi, dovrei interrompere il movimento del nastro, verificare se sono stati inseriti dei comandi remoti, per poi riprendere l'esecuzione.

- Processi cooperanti - Insieme di processi, che cooperano tra loro, occupandosi di funzionalità diverse dell'applicazione.

Questi approcci possono essere implementati con:

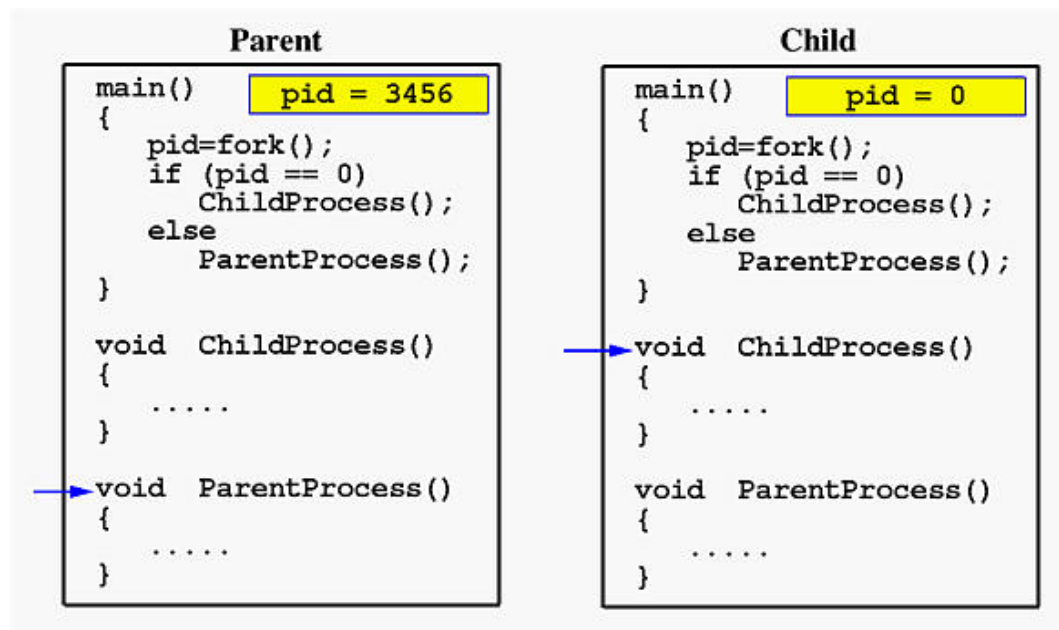
- Programmi separati, eseguiti in parallelo, in cooperazione tra loro
- Programma monolitici, eseguiti come processi monolitici
- Programma monolitico (processo padre), che genera processi cooperanti (processi figli, fork)

Unico file eseguibile, che si occuperà di lanciare processi asincroni

I Processi generati (figli) possono avere risorse indipendenti, o in comune col processo padre.

Lo spazio di indirizzamento del processo figlio, è il duplicato distinto dello spazio del padre, con stesso programma, e (inizialmente) lo stesso contenuto di heap e dati globali.

Essendo poi due processi separati, questi evolveranno separatamente



```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    int pid;

    /* genera un altro processo */
    pid = fork();

    if (pid < 0) { /* si è verificato un errore */
        fprintf(stderr, "Fork fallita");
        exit(-1);
    }
    else if (pid == 0) { /* processo figlio */
        execlp("/bin/ls", "ls", NULL);
    }
}
```

Una volta generato il processo figlio, tramite la funzione `execlp`, è possibile caricare un nuovo programma all'interno del processo figlio, così che esso possa svolgere un'operazione completamente diversa da quella del padre.

Il processo padre, può attendere la terminazione del figlio, oppure continuare la propria esecuzione.

Terminazione di un processo

Un processo può terminare dopo l'esecuzione dell'ultima istruzione del programma, a cui seguirà la restituzione di un valore di stato (return), e la deallocazione delle risorse, per volontà esplicita (invocando la syscall **exit**), oppure a seguito di un'anomalia durante l'evoluzione del processo (**abort**).

Realizzazione del multi tasking

Gestire la turnazione dei processi sul processore, nei casi di:

- Processi **I/O bound** (legati all'I/O), vanno "congelati" tolti dall'utilizzo del processore, una volta inviata la richiesta alla periferica da utilizzare, per **eliminare tempi morti**
- Processi **CPU bound** (legati al processore), composti da molte operazioni A/L, a livello ipotetico potrebbero non fermarsi mai, sarà il sistema operativo a congelarli ogni tanto, per garantire che i processi evolvano **uniformemente**.

In particolare, la gestione della turnazione si compone di 4 attività principali:

- Sospensione del processo in esecuzione (Con salvataggio del suo contesto di esecuzione)
 - Ordinamento dei processi in stato di Ready (scheduling)
 - Selezione del nuovo processo in stato di Ready (dispatching)
 - Riattivazione del processo selezionato (ripristino del suo contesto di esecuzione)
-

Politiche di sospensione dei processi

Un processo può essere sospeso:

- Dopo aver effettuato una richiesta di **I/O**

Assumendo che le interazioni con le periferiche siano lente, il tempo di risposta della periferica, tolti i tempi necessari al cambio di contesto e ripristino del processo, è comunque abbastanza lungo per permettere ad un altro processo di evolvere

- Dopo aver creato un **sottoprocesso**, attendendone la terminazione

Il processo padre (generante) rimarrebbe fermo, aspettando la terminazione del figlio.

Sfrutto questo tempo per far evolvere un altro processo

- Rilascio **volontario** del processore (scelta esplicita del programma)

Ma se il nostro processo fosse fortemente **CPU bound**? Potrebbe non verificarsi alcuna delle casistiche elencate.

Bisogna forzare il rilascio del processore (**pre-emption**), a termine di un quanto di tempo specifico

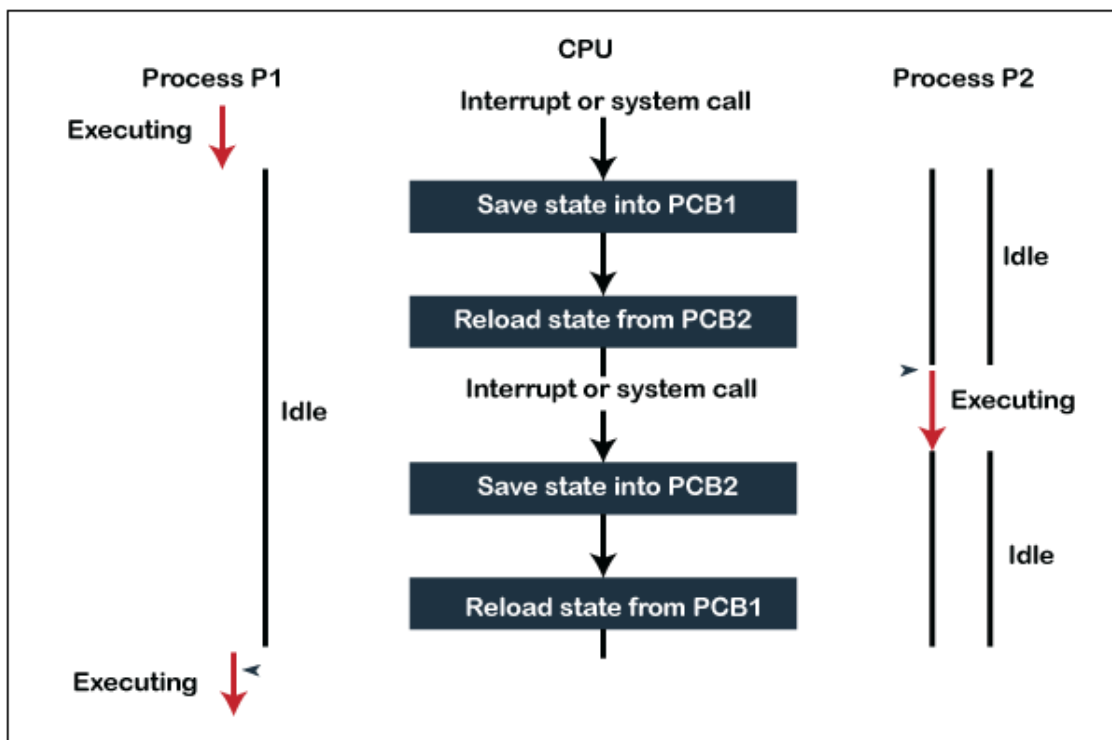
Time slice = intervallo di tempo massimo di uso consecutivo del processore consentito a ciascun processo

Politiche di sospensione dei processi nel time-sharing

- Dopo aver effettuato una richiesta di **I/O**

- Dopo aver creato un **sottoprocesso**, attendendone la terminazione
- Rilascio **volontario** del processore (scelta esplicita del programma)
- Scadenza del quanto di tempo

Context Switching

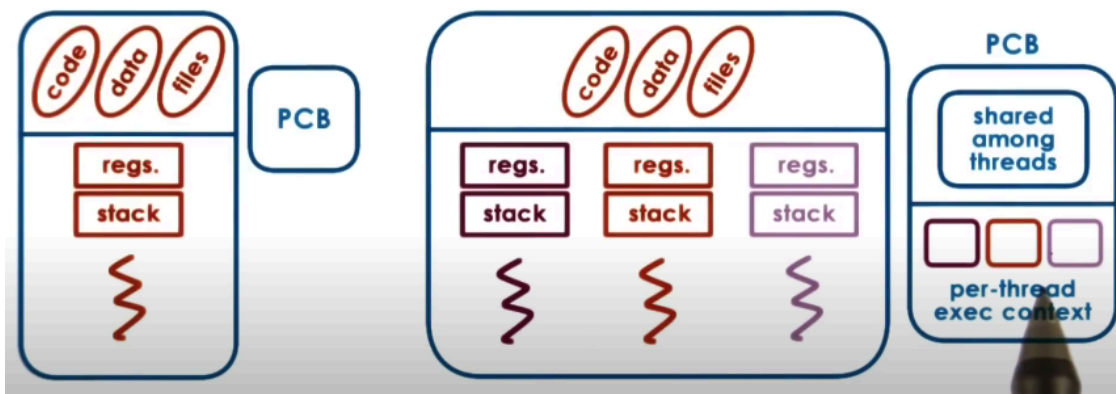


- Il Processo P1 sta eseguendo.
- Arriva un **Interrupt** di **RTC** (Real time clock, che dice che il quanto di tempo è finito), oppure viene invocata una system call (richiesta I/O, creazione sottoprocesso o rilascio volontario)

- P1 viene sospeso, viene effettuato il salvataggio delle informazioni sullo stack, salvato il valore dello stack pointer nel PCB di P1
- Il Sistema operativo, individua il nuovo processo da mandare in esecuzione, a seguito di schedulazione e dispatching
- Viene effettuato il caricamento dello stato del processo P2 (Identificare il valore dello stack pointer di P2, caricarlo, scaricare i valori dei registri dallo stack)
- Il Processo P2 sta eseguendo
- P2 viene nuovamente sospeso, carico sullo stack, salvo stack pointer
- Ri-effettuo la schedulazione, l'operazione di dispatching seleziona il processo a priorità maggiore
- Viene ricaricato lo stato del processo da PCB1

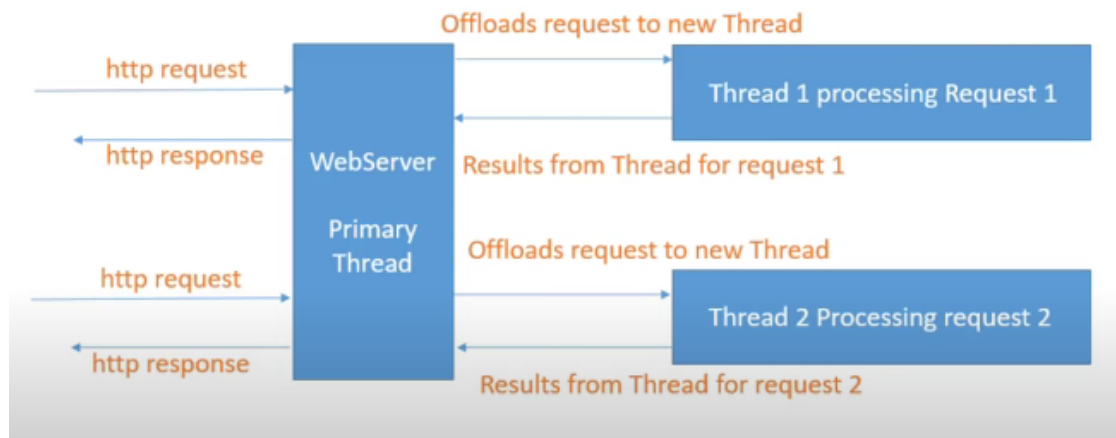
Thread

Unità di esecuzione, all'interno di un processo.



Un'applicazione può avere più processi.

Ogni processo, può avere uno o più threads, che condividono lo stesso codice, file aperti, dati globali e heap (ma mantengono propri registri e stack, poichè eseguiranno porzioni diverse del codice)



Schedulazione (processi e threads)

Politiche che gestiscono l'ordinamento dei processi per l'utilizzo del processore.

Short-term-scheduler (o CPU scheduler)

Si occupa di gestire la turnazione di processi già presenti in RAM, nello stato di pronto. (dallo stato di ready allo stato di running, processi a cui manca solo la CPU)

I Processi sono ordinati e scelti per livelli di priorità.

E' necessario ordinare la lista dei processi per livelli di priorità in maniera veloce, per garantire turnazione rapida dei processi.

Medium-term-scheduler

Permette di gestire la turnazione dei processi **attivi**, temporaneamente memorizzati nell'area di swap su disco (per eventuale mancanza di memoria RAM, o perchè tale processo era precedentemente in attesa di una risorsa molto lenta)

A running process may be suspended because of an I/O request or by a **system call**. Such a suspended process is then

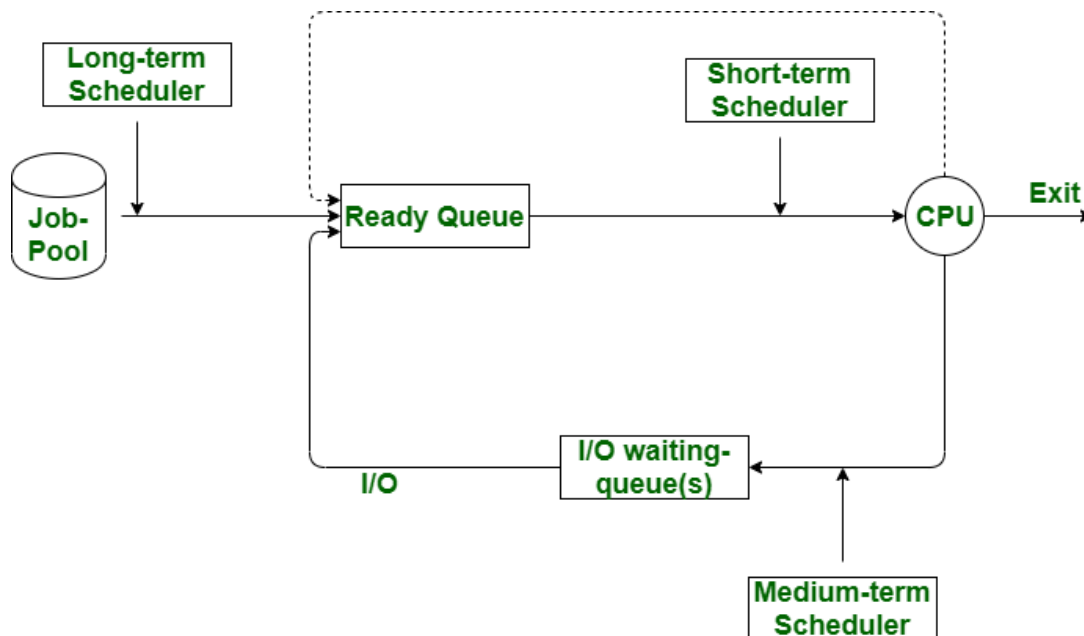
removed from the main memory and is stored in a swapped queue in the **secondary memory** in order to create a space for some other process in the main memory.

I processi nell'area di swap in stato di ready (che hanno completato le operazioni di I/O), hanno bisogno quindi di:

- Essere caricati in RAM, dal Medium-term-scheduler
- Ricevere l'uso della CPU, dopo l'azione del LTS, una volta caricati in RAM, da parte dello STS

Long-term-scheduler

Gestisce il caricamento dei programmi (quindi ancora su disco, al momento del loro lancio), in memoria centrale (dallo stato di New allo stato di Ready)



Rilascio dell'utilizzo della CPU

Gli scheduler, si dividono in base alle condizioni per le quali viene "rimosso" l'utilizzo della CPU ad un processo in esecuzione

- Senza Preemption (senza prelazione): la cpu viene assegnata ad un processo finché questo non lo rilascia esplicitamente o si mette in attesa di un evento / periferica.
- Con Preemption (prelazione): Tipico di sistemi time-sharing, il sistema forza il rilascio della CPU allo scadere del quanto di tempo.

Scelta di Algoritmi di schedulazione

Per valutare la bontà di un algoritmo di schedulazione, bisogna tenere conto dei seguenti criteri:

- **Massimizzazione dell'utilizzo della CPU**
- **Massimizzazione Throughput:** quanti processi vengono completati in un'unità di tempo.
- **Minimizzazione Tempo di turnaround:** Tempo totale per completare esecuzione di un processo, contando anche context switching
- **Minimizzazione Tempo di attesa** Quanto tempo aspetto prima che un processo venga eseguito
- **Minimizzazione Tempo di risposta** Quanto tempo aspetto prima che il processore reagisca ad un evento I/O

Algoritmi di scheduling

FCFS (First-Come-First-Served)

I Processi vengono caricati in una coda FIFO, da cui vengono prelevati per l'utilizzo del processore.

Approccio Non pre-emptive, l'arrivo di un nuovo processo nella coda, non modifica l'esecuzione del processo attuale.

Questo può rappresentare un problema: in caso il processo in esecuzione fosse CPU-bound, questo potrebbe "monopolizzare" l'utilizzo del processore, rilasciandolo dopo un tempo molto elevato.

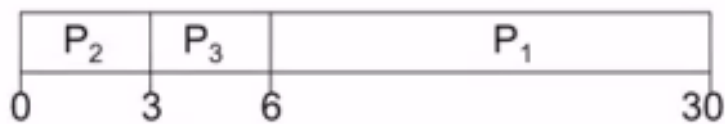
<u>Processo</u>	P_1	P_2	P_3
<u>Tempo di elaborazione</u>	24	3	3

Ordine di arrivo: P_1, P_2, P_3



Tempo di attesa medio: 17

Ordine di arrivo: P_2, P_3, P_1



Tempo di attesa medio: 3

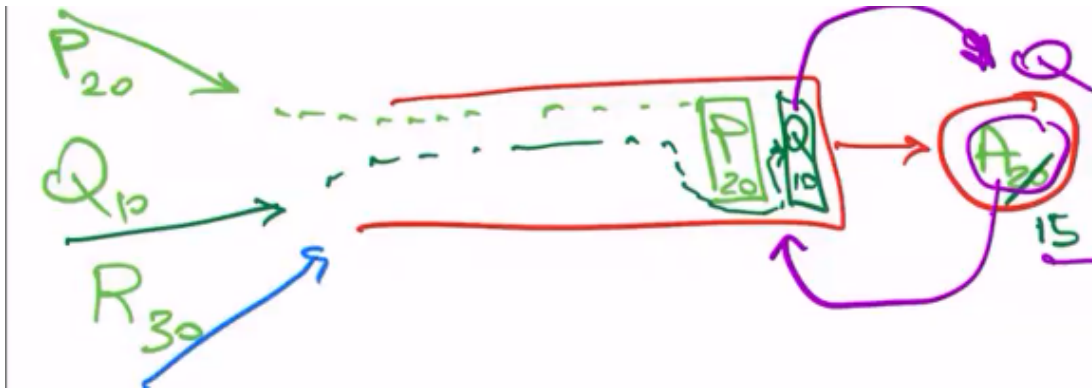
L'ordine di arrivo, influenza fortemente il tempo di attesa.

SJF (Shortest-job-first)

I Processi vengono ordinati nella coda dei processi pronti, per tempo di esecuzione minore.

Se si utilizza il modello pre-emptive, l'esecuzione di un processo può essere interrotta (a run-time, se al processo in esecuzione A mancano ancora 20ms per la terminazione, ma entra nella coda un processo B con tempo di esecuzione di 10ms, il processo A viene interrotto e rimesso in coda, per poter eseguire il

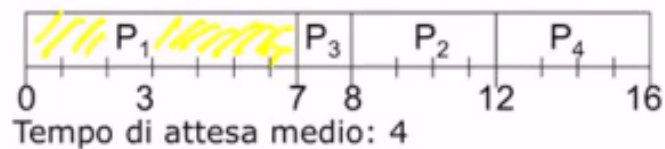
processo B , che avendo tempo di esecuzione minore, ha più diritto di utilizzare il processore)



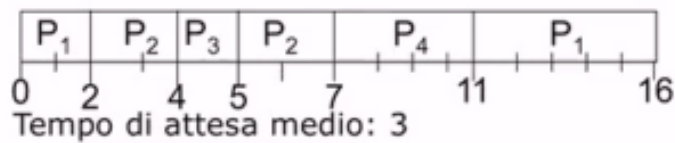
Se si utilizza il modello Non pre-emptive, il processo in esecuzione non può essere interrotto.

Processo	P_1	P_2	P_3	P_4
Tempo di arrivo	0	2	4	5
Tempo di elaborazione	7	4	1	4

- Non pre-emptive



- Pre-emptive



Priorità

SJF, valuta l'ordinamento della coda dei processi, solo in base al tempo di attesa minimo.

Potrebbero esserci però altri criteri che indicano se un processo è più "importante" di un'altro

Esempio: gestione di un sistema di controllo di un aereo, è più importante un processo che gestisce i motori o uno che gestisce l'aria condizionata?

Gestione esecuzione dei processi su di un sistema distribuito tra più aziende, chi paga di più (per un accordo "premium" per esempio, avrà priorità più alta, ed i suoi processi dovranno essere eseguiti prima)

Associo ad ogni processo, un indice di priorità, che può essere basso o alto per indicare alto livello di priorità.

Anche per questo algoritmo, posso adottare un modello con o senza prelazione.

Problema: un processo ready-to-run a priorità bassa, potrebbe aspettare un tempo indefinito, nel caso continuassero ad arrivare processi a priorità più elevata (lo scavalcano, starvation)

Soluzione: Tramite un meccanismo di aging, se mi accorgo che un processo sta aspettando troppo, ne alzo man mano il livello di priorità, finchè non riceve il processore. Se il processo non ha terminato l'esecuzione, viene reinserito nella coda dei processi pronti, ma con il suo livello di priorità originario, non quello "aumentato artificialmente".

Round-Robin (RR)

Tipico dei sistemi time-sharing (quanto di tempo)

Utilizza prelazione

"Raccolgo" circolarmente, in ordine di arrivo, i processi in stato Ready-to-run.

Il comportamento dell'algoritmo, dipende dal dimensionamento del quanto di tempo.

Se questo è molto lungo, ogni processo viene eseguito interamente una volta selezionato → equivalente a FCFS

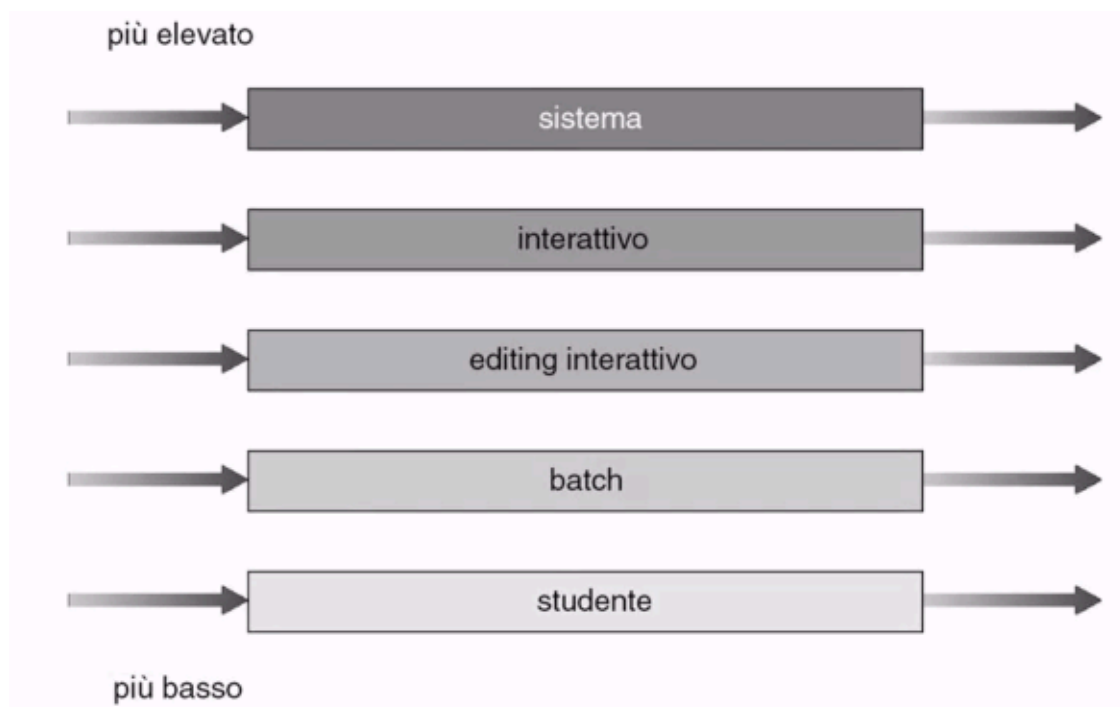
Se il quanto di tempo è troppo corto, il tempo del context switch, potrebbe essere maggiore ai tempi di esecuzione dei processi, molto inefficiente.

Ideale empirico: L'80% delle richieste di elaborazione, devono essere completate in un quanto di tempo.

Coda a più livelli C+L

Nei sistemi operativi, possiamo notare diverse tipologie di processi, con priorità diverse

E' intelligente, assegnare ogni processo ad una diversa coda di schedulazione, ognuna gestita con un diverso algoritmo di schedulazione.



Il sistema ha priorità più elevata, subito seguiti dai processi interattivi, la cui esecuzione deve proseguire "real-time", ecc...

Spesso questo insieme di code, è implementato con reatroazione, ciò vuol dire che un processo può essere in qualsiasi momento cambiato di priorità, mediante algoritmi di promozione e degradazione

Ciò permette, nel caso ci fossero processi CPU bound, di abbassargli la priorità, per far sì che anche gli altri processi ricevano l'uso del processore.

Schedulazione in sistemi multiprocessore

La schedulazione, deve considerare diverse caratteristiche dell'architettura dell'elaboratore:

- Processori: Sono tutti dello stesso tipo? Possono svolgere tutti le stesse operazioni?
- Memoria: Solo condivisa tra i vari processori? o ogni processore ha anche memoria locale?
- Periferiche: Accessibili da tutti i processori? o da solo uno in particolare?

Se il sistema ha processori omogenei, sola memoria condivisa, e periferiche accessibili da tutti i processori, implementiamo una coda unica in memoria condivisa, oppure più code, una per ogni processore, tutte in memoria condivisa (gestite da un gestore di suddivisione del carico)

Se il sistema ha processori omogenei, memoria sia condivisa che locale, e periferiche accessibili da tutti i processori, posso implementare una coda unica, più code in memoria condivisa, o più code, ognuna in memoria locale (vi sarà la suddivisione dei processi dalla memoria condivisa a quella locale).

Se il sistema ha processori omogenei, sola memoria condivisa, periferiche accessibili solo da alcuni processori, implemento code diverse in memoria condivisa, assegnerò i processi I/O alla coda del processore specifico incaricato.

Se il sistema ha processori omogenei, memoria sia condivisa che locale, e periferiche accessibili solo da alcuni processori, posso o implementare la soluzione precedente, oppure diverse code, ognuna in memoria locale, con suddivisione dei processi tra memoria condivisa e locale.

Se il sistema ha processori eterogenei (diversi), avrò code diverse, una per ogni processore, i processi saranno ordinati in base al processore specifico per la sua esecuzione.

Schedulazione real-time

Vi è la necessità di gestire la computazione di alcune attività, in un tempo massimo.

Esempio: gestire un ambiente manifatturiero chimico, se le reazioni non sono gestite entro un tempo massimo, potrei rovinare il prodotto.

Hard real-time

Per garantire la correttezza dell'applicazione, è critico e **obbligatorio** che un processo **termini (produca i propri effetti)** entro un tempo massimo dalla sua attivazione.

Esempi:

Se sto gestendo un aereo, e non controllo l'assetto delle ali entro un tempo massimo, l'aereo precipita.

Se sto gestendo un'auto a guida autonoma, e non fermo l'auto immediatamente quando viene rilevata una persona, farò male a qualcuno

In questi casi, lo schedulatore effettua una stima sul tempo di completamento del processo, controllando che tutte le risorse necessarie siano disponibili, considerando l'algoritmo di scheduling in utilizzo e sceglie se accettare il processo (garantendo che sia eseguito entro il tempo massimo) o rifiutarlo (il sistema non può garantire il completamento entro tempo massimo).

Soft real-time

Ci sono processi critici che vanno necessariamente eseguiti entro un tempo massimo, ed altri che possono essere eseguiti con tempo maggiore.

Posso quindi distinguere tra processi critici (a cui assegno priorità elevata) e non critici (a cui assegno priorità più bassa, eventualmente aumentandola solo se si osserva starvation, senza però contrastare l'esecuzione dei processi critici)

Schedulazione dei thread

Parameters	User Level Thread	Kernel Level Thread
Implemented by	User threads are implemented by user-level libraries.	Kernel threads are implemented by Operating System (OS).
Recognize	The operating System doesn't recognize user-level threads directly.	Kernel threads are recognized by Operating System .
Implementation	Implementation of User threads is easy.	Implementation of Kernel-Level thread is complicated.
Context switch time	Context switch time is less.	Context switch time is more.
Hardware support	No hardware support is required for context switching.	Hardware support is needed.
Blocking operation	If one user-level thread performs a blocking operation then the entire process will be blocked.	If one kernel thread performs a blocking operation then another thread can continue execution.

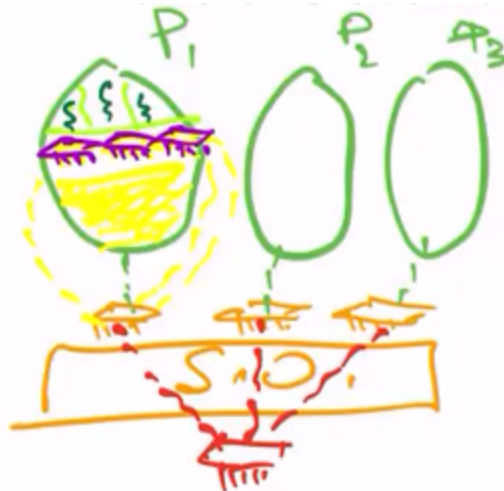
Thread user-level

Creati dal programmatore utilizzando libreria specifiche del linguaggio di programmazione.

I Threads sono interni al processo padre, pertanto il S.O si occuperà di fornire un'immagine virtuale del processore al solo processo padre, poi la libreria threads, si occuperà di moltiplicare a sua volta l'immagine virtuale, in altre immagini virtuali minori, per ogni thread.

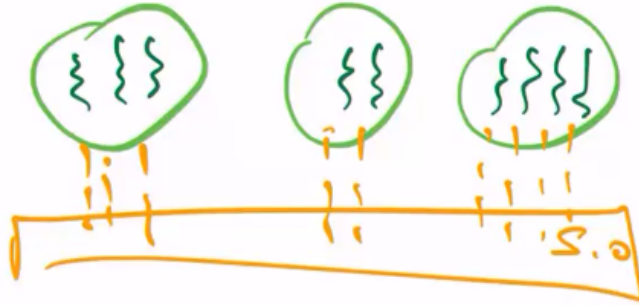
Il S.O quindi non vede i threads direttamente, lui si occupa solo della schedulazione dei processi padre.

I Threads all'interno del processo, saranno gestiti da uno "scheduler dei threads", integrato nella libreria dei threads.



Thread kernel-level

Gestiti nativamente dal S.O, riconosciuti da esso come fossero processi differenti (il S.O in questo caso, crea 3 immagini del processore virtuali)



Processi cooperanti

Cooperano per il raggiungimento di uno scopo applicativo comune, coordinandosi e comunicando tra loro, scambiandosi informazioni.

Identifichiamo in questo scenario, due entità:

- Processi produttori dell'informazione
- Processi "consumatori", che necessitano delle informazioni prodotte da altri processi per poter operare.

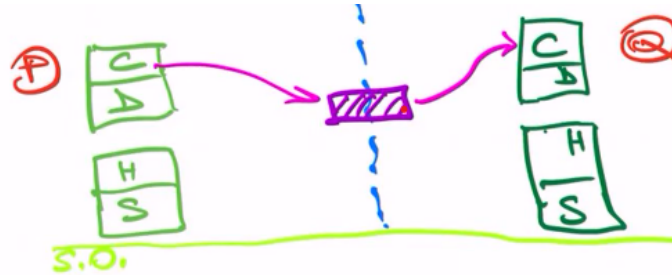
E' necessario introdurre un meccanismo di comunicazione (IPC, Inter Process Communication), implementabile secondo due modalità:

- Comunicazione diretta, il processo mittente conosce e comunica direttamente col processo destinatario
- Comunicazione indiretta, in processo mittente non conosce direttamente il destinatario, il mittente pone i suoi messaggi in una struttura dati, senza sapere chi li leggerà.

IPC a comunicazione diretta

- Memoria condivisa
Questa modalità di comunicazione si basa sulla condivisione di una porzione di memoria (variabili globali o buffer)

Ipotizziamo di avere un processo P produttore, questo produrrà l'informazione, e la porrà in un'area di memoria condivisa col processo consumatore Q, che potrà leggerla.



Le operazioni di lettura e scrittura, sono incompatibili con loro: Ipotizziamo un'area di memoria condivisa di 4 bytes, in un sistema in cui le operazioni di scrittura avvengono 2 bytes alla volta, se mentre il processo P sta aggiornando i dati nell'area condivisa, il processo Q inizia un'operazione di lettura, esso si troverebbe a leggere dati inconsistenti

E' necessario introdurre meccanismi di mutua esclusione (è concessa una sola operazione alla volta, o lettura o scrittura) e di sincronizzazione.

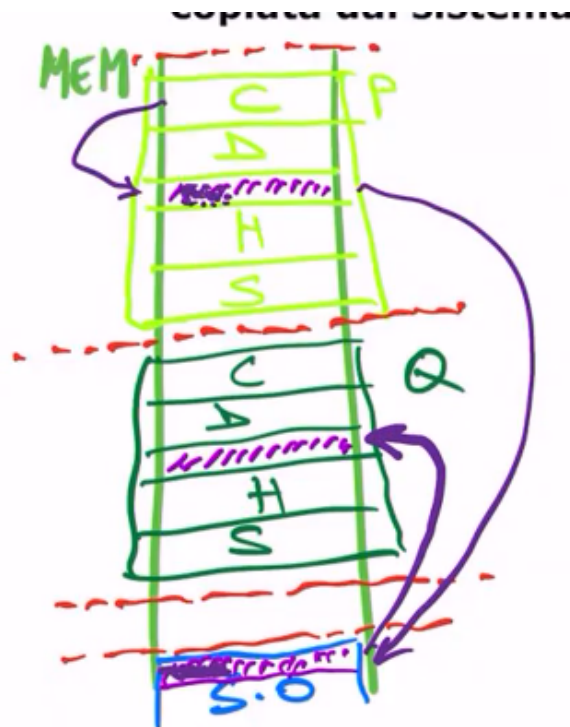
La scrittura/lettura dei messaggi, può avvenire secondo due modalità:

- Area comune copiata dal sistema operativo

Il processo P mittente, scrive le informazioni in una propria area riservata, all'interno del suo spazio di indirizzamento

Al termine della scrittura, il S.O copia il contenuto di quell'area, in un'area di memoria centrale del S.O

Successivamente, quando il processo Q destinatario vorrà leggere i dati, il S.O copierà il contenuto dell'area di memoria centrale del S.O, nell'area condivisa di Q.



- Area comune fisicamente condivisa.

Un'area di memoria dello spazio utente, viene associata al processo P e al processo Q, i quali potranno leggerla e scriverla direttamente.

Nella condivisione di buffer ciò che cambia rispetto a prima è l'utilizzo di un buffer.

La comunicazione rimane diretta, ma il processo mittente scriverà stavolta le sue informazioni all'interno del buffer dal quale poi il processo ricevente andrà a recuperarle.

- Scambio di messaggi

La comunicazioni avviene per mezzo di messaggi, all'interno dei quali sono contenute le informazioni da trasmettere, il processo mittente, quello destinatario, ed eventuali altri parametri supportati dal sistema operativo, per gestire l'ordinamento dei messaggi, la formattazione ecc...

I messaggi vengono memorizzati in buffer nello spazio di indirizzamento del S.O, che possono essere di uso generale (S.O mette a disposizione dei buffer utilizzabili da tutti i processi che ne necessitano), oppure specifici per una coppia di processi (assegnati dal S.O per ogni coppia di processi in comunicazione).

Le funzioni messe a disposizione per la gestione dei messaggi sono quelle per l' invio e la ricezione:

- `send(proc_ricevente, messaggio)`.
Deposita il messaggio in un buffer libero. La funzione è bloccante, ovvero blocca il processo mittente nel caso in cui non ci fosse spazio disponibile. Una volta liberato un buffer, la funzione completa la comunicazione con il destinatario e sblocca il mittente.
- `cond_send(proc_ricevente, messaggio): error status`.
A differenza della funzione di invio precedente, non è bloccante. Se al momento di depositare il messaggio non è presente alcun buffer libero, ritornerà un messaggio di errore e il messaggio non sarà più depositato.
Sarà responsabilità del mittente decidere se rimandarlo o meno.
- `receive(proc_mittente, messaggio)`.
Riceve il messaggio presente nel buffer. E' bloccante, ovvero blocca il destinatario fino a quando non c'è un messaggio nel buffer da leggere.
- Ricezione condizionale: `cond_receive(proc_mittente, messaggio): error status`.
Il processo ricevente preleverà il messaggio dal buffer; se non ci sono messaggi ritornerà una condizione di errore, senza bloccare il destinatario.

La comunicazione tramite buffer in generale è asincrona, ovvero il mittente può spedire il messaggio in qualsiasi momento della computazione senza preoccuparsi se c'è qualche ricevente in grado di raccoglierlo.

In una comunicazione sincrona lo scambio delle informazioni avviene invece solo quando entrambi gli interlocutori sono pronti.

Aspetto molto interessante dello scambio di messaggi, è che può essere adottato sia in casi di comunicazione diretta (simmetrica, mittente e destinatario sono univocamente identificati), sia generalizzando in una comunicazione indiretta (asimmetrica).

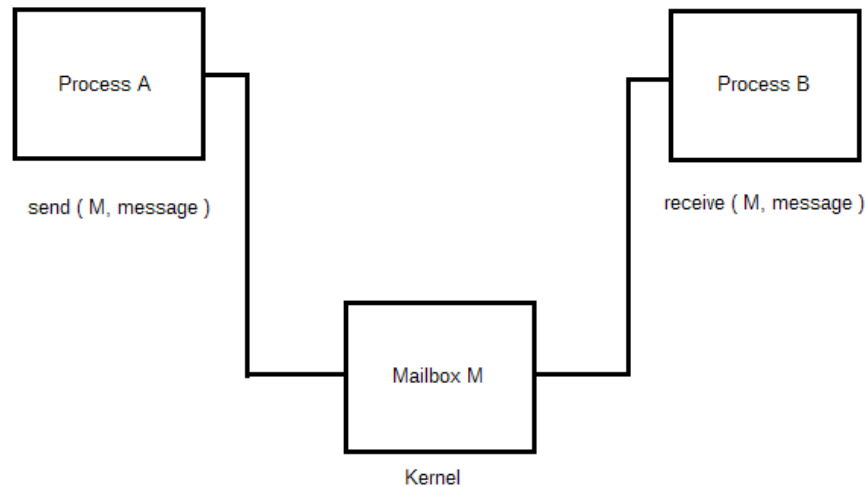
Potrebbe capitare infatti, che il messaggio prodotto da un processo, debba essere utilizzato da un gruppo di processi, e non da un singolo.

Al contrario, potrebbe esserci un pool di processi che erogano lo stesso servizio, e da cui il processo utilizzatore vuole leggere, indipendentemente da chi sia il produttore specifico (il processo utilizzatore tanto sa che ogni processo eroga il medesimo servizio)

Al posto di far partire dal processo mittente, una serie di `cond_send` per ogni processo utilizzatore, o una serie di `cond_receive` nel processo destinatario, posso "scaricare" questo onere al sistema operativo, specificando un gruppo di processi, e a quel punto potrò utilizzare l'identificatore del gruppo nelle funzioni `send` e `receive`

IPC a comunicazione indiretta

- Comunicazione con mailbox (generalizzazione dei buffer)
È un sistema di comunicazione indiretta, attraverso una chiamata di sistema il processo mittente deposita un messaggio all'interno di una struttura dati nello spazio di indirizzamento del sistema operativo (mailbox); questo messaggio viene poi estratto dal processo destinatario attraverso un'altra chiamata di sistema.



Notiamo che il processo A mittente, conosce solo la mailbox (nella chiamate di sistema `send`, viene identificata solo M).

All'interno dei messaggi lasciati nella mailbox, è però presente l'informazione relativa al processo mittente, per far sì che il processo B possa eventualmente rispondere mediante comunicazione diretta al mittente.

Va ricordato che non stiamo assegnando i messaggi ad una coppia di processi, ma li stiamo inserendo in una struttura accessibile (teoricamente) da tutti; eventuali restrizioni potranno essere applicate con delle politiche di accesso stabilite dal sistema operativo. In generale la mailbox non è proprietà del processo, ma del sistema operativo (ricordiamo che è nel suo spazio di indirizzamento). Esistono però delle procedure per attribuirle ad un processo proprietario, al quale il sistema può dare il diritto esclusivo di ricezione. In questo caso, quando il processo proprietario termina, la mailbox viene deallocata con esso.

Le funzioni di invio (normale e condizionato), e ricezione (normale e condizionata), sono simili alla comunicazione tramite scambio di messaggi, in

più abbiamo le funzioni create(M) e delete(M), per la creazione e l'eliminazione di una mailbox.

La mailbox può avere una capienza limitata o illimitata, e può essere utilizzata per implementare scenari:

- Comunicazione multi-a-uno, In cui ci sono più processi client (mittenti), che depositano le loro richieste di un servizio all'interno della mailbox, e un singolo processo server (destinatario), che elabora le singole richieste, estraendole dalla mailbox una ad una.
 - Comunicazione uno-a-molti, Singolo processo client mittente, e una serie di processi server, per far sì che si possano soddisfare più richieste provenienti dallo stesso client.
 - comunicazioni-multi-a-molti, Più processi client che comunicano con più processi server.
- Comunicazione con file e mediante pipe

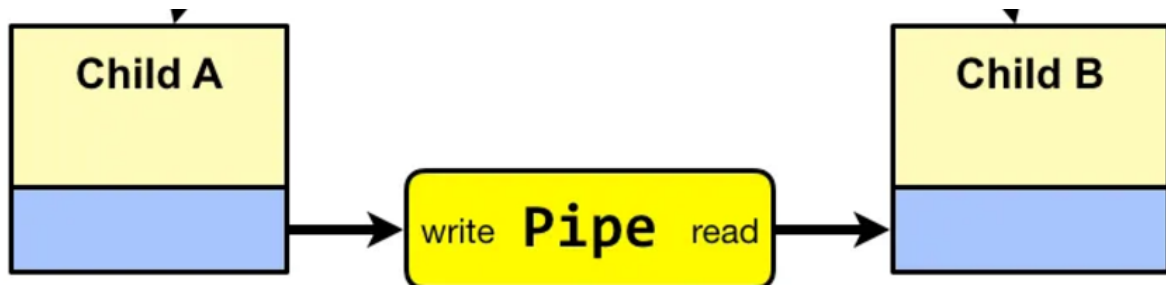
La comunicazione con file, come suggerisce il nome, avviene attraverso la condivisione di file.

Il processo P mittente, utilizzerà apposite chiamate di sistema per scrivere su di un file in memoria di massa, un processo Q mittente estrarrà le informazioni dal file.

È una estensione delle mailbox, tuttavia a differenza della mailbox che risiede in memoria centrale, i file risiedono in memoria di massa. Sarà poi compito del sistema operativo la gestione delle operazioni.

Questo sistema, permette di implementare comunicazioni multi a uno, uno a molti, molti a molti.

La comunicazione mediante pipe avviene utilizzando le pipe come strutture di appoggio, simili ad una coda, che collegano lo stdout di un programma, allo stdin di un'altro.



Le operazioni di scrittura e lettura avvengono su questi due insiemi, secondo una logica FIFO.

- Comunicazione con socket

Permette di generalizzare la comunicazione, sia essa interprocesso oppure in rete, utilizzando un'API fornita direttamente dal sistema operativo.

A livello logico, un socket è rappresentato da due valori, un indirizzo IP a 32 bit, e un numero di porta, per identificare il processo in comunicazione in esecuzione sull'elaboratore.

Concettualmente la comunicazione con socket funziona come la comunicazione con pipe, tuttavia la pipe viene "spezzata", ed i due estremi vengono posti sui processi mittente e destinatario. Solitamente questo approccio viene implementato tra più calcolatori.

Il sistema operativo per collegare gli estremi delle pipe si appoggia alla rete.

Processi Indipendenti e Mutua esclusione

Non hanno scopi comuni con altri processi, non utilizzano informazioni condivise.

Il meccanismo di coordinamento della computazione, è comunque richiesto per l'utilizzo delle risorse condivise: parliamo di concorrenza quando ho più processi

che richiedono l'accesso a risorse condivise usabili solo in mutua esclusione. Tali processi vengono definiti concorrenti.

Si parla di esecuzione in "mutua esclusione" per indicare una serie di operazioni incompatibili che non possono essere compiute contemporaneamente sulla stessa risorsa.

Ad esempio una scrittura e una lettura sullo stesso dato non sono concesse, due letture sì.

Il suo scopo è dunque preservare la consistenza dell'informazione, o si avrebbero casi in cui i processi computino dati scorretti.

La sincronizzazione è quindi quell'insieme di politiche e meccanismi che si occuperanno di garantire tale principio per l'uso di risorse condivise, che queste siano fisiche (come le periferiche) o informative (come file o altri dati).

Notare come sia un problema peculiare dei sistemi multi-tasking, dal momento che se avessi un'esecuzione seriale dei processi non avrei mai richieste di accesso parallele

Sezione critica

La sezione critica è una porzione di codice che può generare errori se eseguita in maniera concorrente (modifiche di variabili comuni, scrittura su file condivisi, eccetera).

L'intento è quindi quello di individuarle, e di implementare un protocollo, una serie di regole, per eseguirle evitando problemi di inconsistenza, garantendo:

- Mutua esclusione, se un processo sta eseguendo una sezione critica, nessun'altro processo può fare altrettanto.
- Progresso, ovvero garantire che i processi si evolvano senza rimanere bloccati in attesa

- Attesa limitata, ovvero garantire che i tempi di attesa non risultino troppo lunghi.

In sintesi, dobbiamo gestire le sezioni critiche del codice, garantendo l'accesso alle variabili

condivise, che questo accesso sia rapido, e che nessun processo rimanga in attesa all'infinito.

Problema del produttore-consumatore

Il problema del produttore-consumatore è un modello classico per lo studio della sincronizzazione nei sistemi operativi.

Esso consiste nel sincronizzare due processi, uno (il produttore) che vuole inviare informazioni (sui suoi prodotti) e l'altro (il consumatore) che vuole leggerli.

Entrambi i processi utilizzano un buffer circolare di n elementi come struttura dati di appoggio condivisa.

I codici potrebbero essere i seguenti:

Produttore

```
while(1){
    while (counter == BUFFER_SIZE) ; // non fa nulla

    // aggiungi un elemento nel buffer
    buffer[index] = producedItem;
    index = (index + 1) % BUFFER_SIZE; //posizione "libera" incrementata di 1
    //il modulo è aggiunto per dare struttura circolare al buffer.
    ++counter;
}
```

Consumatore

```
while(1){
    while (counter == 0) ; // se buffer è vuoto, non fa nulla
```

```

// rimuovi un elemento nel buffer
item = buffer[out];
out = (out + 1) % BUFFER_SIZE; //prossima posizione da cui estrarre dato
--counter;
}

```

Se eseguo questi due codici concorrentemente, posso avere dei problemi.

Le singole **istruzioni** del **linguaggio macchina**, sono mutuamente esclusive (nel senso che una volta effettuato il fetch di un'istruzione, essa viene portata a termine in mutua esclusione)

Ma una sequenza di istruzioni, può essere invece interrotta da un interrupt.

Nei due codici vengono effettuate due operazioni di aggiornamento sulla variabile counter, `++counter` per il produttore, `--counter` per il consumatore.

Convertendo queste due istruzioni (dal C++ per esempio) in linguaggio macchina:

Il problema nasce nella condivisione della variabile count.

```

register1 = counter
register1 = register1 + 1
counter = register1

```

```
//++counter
```

```

register2 = counter
register2 = register2 - 1
counter = register2

```

```
//--counter
```

Se io eseguiessi quindi questa serie di 6 istruzioni concorrentemente, l'esecuzione potrebbe essere interrotta (per termine del time-slice per esempio) in questo

modo per esempio:

S0: producer	esegue	register1 = count	{register1 = 5}
S1: producer	esegue	register1 = register1 + 1	{register1 = 6}
S2: consumer	esegue	register2 = count	{register2 = 5}
S3: consumer	esegue	register2 = register2 - 1	{register2 = 4}
S4: producer	esegue	count = register1	{count = 6}
S5: consumer	esegue	count = register2	{count = 4}

Queste 3 istruzioni rappresentano ciascuna la sezione critica del produttore e del consumatore, e devono essere eseguite in mutua esclusione, cioè in modo non interrompibile (senza che possano verificarsi “mescolamenti” con istruzioni di altri processi)

La corsa critica del problema del produttore-consumatore non è legata ad operazioni contemporanee di lettura e scrittura, dal momento che ognuna di esse avviene all'interno dello spazio di indirizzamento dei rispettivi processi; ma è dovuta ai possibili errori nell'aggiornamento del contatore del buffer.

Sincronizzazione dei processi concorrenti

Approcci a livello istruzioni:

Variabili di turno

Variabile condivisa tra processi che devono utilizzare una risorsa condivisa, definisce qual'è il processo che in un certo istante può utilizzare la risorsa.

Tre possibili utilizzi (Supponiamo di avere due processi 0 e 1):

- **Scenario 1**

```
public class Algorithm_1 implements MutualExclusion{  
    private volatile int turn; // la variabile di turno
```

```

// Costruttore, inizializza la variabile di turno
//(per esempio, al processo 0, dando quindi la possibilità
//al processo 0 di accedere alla propria sezione critica)
public Algorithm 1()
{
    turn = TURN 0;
}
/*Entrata nella sezione critica, se turn!=t (t è il processo corrente)
il thread aspetta (yield)*/
public void enteringCriticalSection (int t) // (2)
{
    while (turn != t)
        Thread.yield();
}

/*Uscita dalla sezione critica, rilascio la risorsa
specificando chi sarà il nuovo processo che potrà utilizzare
la risorsa (t sarà diverso dal processo attuale)*/
public void leavingCriticalSection (int t) // (3)
{
    turn = 1 - t; //0 se t=1, 1 se t=0
}
}

```

Questo algoritmo garantisce la mutua esclusione imponendo una stretta alternanza dei processi, con tutti gli svantaggi che questo comporta.

Esempio:

Processo 1 (Produttore), produce un dato ogni secondo, e vuole metterlo nel buffer.

Processo 2 (Consumatore), consuma un dato ogni 10 secondi

Il processo 1 produce un dato, lo mette nel buffer, assegna l'uso della risorsa al processo 2.

Il processo 2, ci impiega 5 secondi prima di entrare nella propria sezione critica e consumare il dato.

Processo 1 quindi aspetta tanto!

- **Scenario 2 - Prenotazione della risorsa**

```
public class Algorithm_2 implements MutualExclusion{
    //flag booleani , visibili da entrambi i processi, che indicano se un processo vuole prenotare l'uso della risorsa.
    private volatile boolean flag0, flag1;

    public Algorithm 2() //Inizializzazione dei flag a false
    {
        flag0 = false; flag1 = false;
    }

    //Prenotazione della risorsa
    public void enteringCriticalSection (int t)
    {
        if (t == 0){
            flag0 = true; //se è il processo 0 a voler utilizzare la risorsa, si pone il suo flag a true (prenotazione)
            while (flag1 == true) //se l'altro processo ha già prenotato la risorsa, aspetto, altrimenti utilizzo la risorsa
                Thread.yield();
        }else{
            flag1 = true; //same
            while (flag0 == true)
                Thread.yield();
        }
    }

    public void leavingCriticalSection (int t) //All'uscita della sezione critica, si pone il flag del relativo processo a false.
```

```

{
    if(t == 0)
        flag0 = false;
    else
        flag1 = false;
}

}

```

Questo meccanismo di prenotazione, mi risolve il problema precedente: non c'è stretta alternanza dei processi, la risorsa potrebbe anche rimanere non prenotata da entrambi i processi.

Non garantisce tuttavia il progresso, potrei avere una situazione in cui un processo utilizzi per un tempo infinito una risorsa, "bloccando" l'altro.

- Scenario 3 - Soluzione combinata, flag e variabile di turno

```

public class Algorithm_2 implements MutualExclusion{
    private volatile boolean flag0, flag1; //flag di prenotazione
    private volatile int turn; //definisce chi ha prenotato la risorsa

    public Algorithm _3()
    {
        flag0 = false;
        flag1 = false;
        turn = TURN_0; //inizializzazione arbitraria
    }

    public void enteringCriticalSection (int t) // (1)
    {
        //garantisce che i due processi si alternino nell'utilizzo della risorsa
        int other = 1 - t;
        turn = other; //il turno successivo, sarà dell'altro processo.
    }
}

```

```

    if (t == 0){
        flag0 = true; //prenoto la risorsa
        while (flag1 == true && turn == other)
            Thread.yield();
    }else{
        flag1 = true;
        while (flag0 == true && turn == other)
            Thread.yield();
    }
}

```

public void leavingCriticalSection (int t) //All'uscita della sezione critica, si pone il flag del relativo processo a false.

```

{
    if(t == 0)
        flag0 = false;
    else
        flag1 = false;
}

```

Garantisce sia mutua esclusione che progresso.

Variabili di lock

La variabile di lock è una variabile condivisa che definisce lo stato di uso di una risorsa, cioè quando è in uso da parte di un processo (che è evidentemente nella sua sezione critica).

- Lock=0 - Risorsa libera
- Lock=1 - Risorsa in uso

Cambia così il punto di vista rispetto alla variabile di turno: non sono più i processi ad alternarsi ma è la risorsa stessa a dire se è disponibile o no.

Le variabili di turno devono essere inizializzate e gestite a seconda del numero di processi che accedono alla stessa risorsa, e nella grande maggioranza dei casi

questo numero non è predicibile a priori; i lock invece scalano benissimo, perché non importa quanti sono i processi che vogliono una risorsa, lui sarà sempre o libero o occupato.

Un processo che vorrà utilizzare la risorsa quindi:

- Legge la variabile di lock
- Se il lock è =0, la risorsa è libera, marco la risorsa "in uso" ponendo la variabile di lock a 1, utilizzo la risorsa
- Se il lock è =1, pongo il processo in attesa.

```
IF lock=0
    lock=1
    //uso risorsa
ELSE
    wait
```

Ma se tra l'if e lock=1 il processo si interrompesse? Il secondo processo, vedrebbe ancora lock=0, e inizierà ad utilizzare la risorsa, poi quando il primo processo verrà ripreso, esso vorrà ancora utilizzare la risorsa, perché è già entrato nell'IF!

E' importante quindi, che l'acquisizione di una risorsa avviene a interruzioni disabilitate:

- Disabilito le interruzioni
- Leggo la variabile di lock
- Se la risorsa è libera (lock = 0), la marco in uso ponendo lock = 1 e riabilito le interruzioni
- Se la risorsa è in uso (lock = 1), riabilito le interruzioni e pongo il processo in attesa che la risorsa si liberi
- Per rilasciare la risorsa pongo lock = 0

Il motivo per cui vanno bloccate le interruzioni è che le operazioni di lettura ed

eventuale scrittura di un lock sono realizzate con una sequenza di istruzioni macchina, quindi interrompibili (solo la singola istruzione macchina è atomica). Se quindi non mi tutelo sospendendo gli interrupt finirei per avere una corsa critica nello stesso sistema che uso per prevenirle.

Una soluzione alternativa è introdurre direttamente nell'hardware del processore dei meccanismi per ottenere la sincronizzazione.

Ad esempio è possibile inserire l'istruzione atomica TEST-AND-SET che traduce la sequenza di istruzioni precedenti in una sola, non rendendo più necessaria alcuna sospensione delle interruzioni.

Essa implementa infatti in un'unica istruzione le seguenti operazioni: legge la variabile di lock e la pone in un flag del processore, quindi la pone uguale a uno, infine se il flag (vecchio valore di lock) era 0 allora significa che la risorsa era libera, altrimenti il processo dovrà attendere.

Il limite di questa soluzione è che non spetta al programmatore la sua implementazione, bensì al progettista del processore.

Approcci a livello del S.O

Semafori

Innalzare il livello di astrazione, spostando la gestione della sincronizzazione in funzioni del sistema operativo, garantendo così che tutte le operazioni di sincronizzazione della computazione avvengano in maniera corretta, evitando usi/implementazioni errate da parte del programmatore.

A tal fine sono stati introdotti i semafori binari S, una struttura dati (in particolare, una variabile binaria) gestita dal sistema operativo che rappresenta lo stato di uso della risorsa condivisa:

- 1 - Risorsa libera
- 0 - Risorsa in uso

Nonostante il significato dei valori sia opposto a quelli di lock, non c'è rischio di confondersi dato che il programmatore non manipolerà direttamente la variabile utilizzata per implementare il semaforo, ma utilizzerà delle procedure di sistema (quindi nativamente atomiche e non interrompibili):

- `acquire(S)` - acquisisce l'uso della risorsa
- `release(S)` - rilascia la risorsa

```
Semaphore S;
```

```
acquire(S);  
criticalSection();  
release(S);
```

Il semaforo può essere implementato, in caso di risorsa non disponibile, per attendere in modo:

- Attivo - La procedura `acquire(S)` sarà implementata in un modo del tipo `while(S==0) Thread.yield()`, pertanto quando la risorsa sarà occupata, il processo che ne ha richiesto l'utilizzo continuerà a verificare lo stato della variabile, comportando uno spreco di CPU, che altri processi potrebbero utilizzare in maniera più produttiva.

Questo fenomeno si chiama spinlock, e potrebbe anche risultare utile in caso i tempi di attesa fossero brevi, in modo da evitare onerosi context switch.

- Rischiedulando - Quando i tempi di attesa attiva diventano troppo lunghi, possiamo ottimizzare l'uso delle risorse introducendo una coda dei processi in attesa delle risorse, così strutturata:
 - Quando un processo esegue un'acquire(S), se la risorsa non è disponibile passa in stato di wait, e viene accodato nella coda di attesa del semaforo
 - Quando un processo esegue una release(S), la risorsa viene rilasciata e viene attivato il primo processo della coda di attesa di S, a cui viene concesso l'accesso.

La coda è ordinata dallo schedatore dei processi, secondo una politica ben definita.

Bisogna comunque prestare attenzione alla progettazione della coda di attesa, o potresti avere fenomeni di stallo (deadlock) in cui due o più processi aspettano un evento che può essere generato solo da uno dei processi in attesa.

In altre parole bisogna prestare attenzione all'ordine con cui avvengono le chiamate di sistema di prenotazione e rilascio delle risorse, o potresti avere attesa circolari senza rilascio.

Un altro pericolo è quello di incorrere in una starvation, il blocco indefinito meglio affrontato nel capitolo sulla schedulazione.

Semafori generali

I semafori generalizzati S sono invece variabili intere che rappresentano lo stato di uso di un insieme di risorse omogenee condivise: il valore della variabile intera S , indica il numero di risorse di quel tipo disponibili.

La sintassi delle funzioni di manipolazioni del semaforo rimangono le stesse, l'acquire(S) (acquisizione d'uso di una risorsa) e la release(S) (rilascio della risorsa) rispettivamente incrementano e decrementano di uno il valore del semaforo.

Quando l'acquire() arriva a 0, ho un'attesa.

Monitor

L'uso dei semafori, presenta ancora alcuni limiti: nonostante si eliminino gli errori derivanti dalla loro implementazione, l'onere del loro corretto utilizzo rimane al programmatore

Spetta ancora al programmatore infatti gestire manualmente l'uso dei semafori: se esso si dimenticasse di chiamare la funzione `acquire`, prima di entrare nella sezione critica? o se si dimenticasse di rilasciare la risorsa? Quella risorsa a quel punto rimarrebbe permanentemente in uso (a meno che il processo termini).

Errori nel codice, potrebbero portare a problemi cui:

- **Deadlock:** Quando due processi restano bloccati perché ciascuno attende una risorsa che l'altro possiede.
- **Starvation:** Quando un processo non riesce mai a ottenere una risorsa a causa della gestione errata delle code d'attesa.

Per superare questi limiti, introduciamo il costrutto linguistico "monitor", che permette di identificare un insieme di funzioni che devono essere eseguite in modo mutuamente esclusivo (ogni funzione del monitor rappresenta quindi la sezione critica di un diverso processo), ed incapsula:

- Variabili condivise tra processi
- Procedure per accedere a tali variabili
- Meccanismi di sincronizzazione

Un esempio di pseudocodice in java è il seguente:

```
monitor NomeMonitor {  
    // Dichiarazione delle variabili condivise  
  
    public entry metodo1() {  
        // Codice che accede alle risorse condivise  
    }  
  
    public entry metodo2() {  
        // Altro codice che accede alle risorse condivise  
    }  
}
```

La mutua esclusione, è automatica, poiché il compilatore farà in modo che una sola procedura del monitor alla volta possa essere eseguita (e quindi un solo processo alla volta, accederà alla sua sezione critica), forzando l'acquire della risorsa prima dell'esecuzione del codice delle singole funzioni, e la release successivamente.

Starvation

Condizione per cui un processo non ottiene la risorsa di cui ha bisogno per proseguire nella computazione per un lungo periodo di tempo, poichè tale risorsa è assegnata ad altri processi.

Le cause potrebbero essere:

- Utilizzo di una politica di schedulazione della coda dei processi in attesa, che non garantisce l'attesa finita (Esempio: favorendo sempre i processi a più alto livello di priorità)

E' pertanto necessario verificare che le politiche di schedulazione scelte, garantiscano attesa finita.

- Quantità insufficiente di risorse
-

Deadlock

Un gruppo di due o più processi non possono proseguire nella computazione, poichè ognuno attende un evento che può essere generato solo da un altro processo del gruppo

Questi eventi, possono essere di diverso tipo, nel nostro caso, consideremo acquisizione e rilascio di risorse (fisiche o logiche che siano)

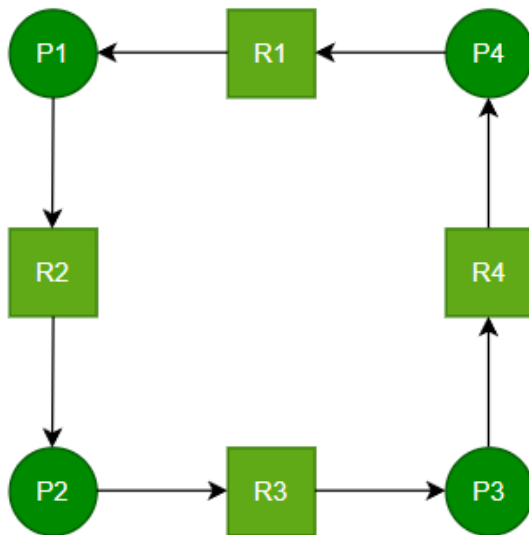
Il Deadlock è un processo infinito, pertanto una volta che un processo è in uno stato di deadlock, non ne uscirà mai.

Esistono sole tecniche di rilevazione, risoluzione e prevenzione, ma non tecniche di stopping del deadlock.

Formalmente, ho una condizione di stallo, se e solo se si verificano contemporaneamente queste 4 condizioni:

- **Mutua esclusione** - La/e risorsa/e coinvolte nella situazione di stallo, sono da utilizzare in mutua esclusione. (Naturalmente, se la risorsa non fosse da utilizzare in mutua esclusione, non avrei competizione per il suo utilizzo, allora non ci sarebbe nemmeno deadlock)
- **Hold & Wait** - Un processo detiene già una o più risorse, ma per completare la computazione, ne ha bisogno di un'altra, e pertanto si mette in attesa.
- **Nessun rilascio anticipato (No pre-emption)** - Non è ammissibile rilasciare la risorsa prima del termine della computazione
- **Attesa circolare (Circular wait)** - Catena di processi in cui si verifica "Hold & Wait":

Example: Imagine four processes—**P1**, **P2**, **P3**, and **P4**—and four resources—**R1**, **R2**, **R3**, and **R4**.



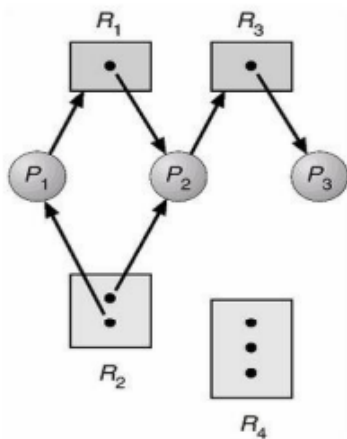
- **P1** is holding **R1** and waiting for **R2** (which is held by **P2**).
- **P2** is holding **R2** and waiting for **R3** (which is held by **P3**).
- **P3** is holding **R3** and waiting for **R4** (which is held by **P4**).
- **P4** is holding **R4** and waiting for **R1** (which is held by **P1**).

Grafo di allocazione delle risorse

Per verificare queste 4 condizioni (e confermare lo stato di deadlock), introduciamo il Grafo di allocazione delle risorse (come in figura sopra), costituito da:

- Nodi (Risorse R_i , rettangoli con pallini dentro, che rappresentano il numero di istanze di quella risorsa disponibili, e processi P_i , cerchi)
- Archi di richiesta di una risorsa (non ancora soddisfatta, pendente, la risorsa è occupata), che vanno da P_i a R_i
- Archi di assegnazione, da R_i a P_i (la risorsa era libera, è stata assegnata al processo P_i)

Senza deadlock



Le risorse R1 ed R3 sono disponibili nel sistema con molteplicità 1, R2 con molteplicità 2, R4 con molteplicità 3.

Il processo P1 ha richiesto R1 che però è già stato assegnato a P2, quindi rimane in attesa.

La risorsa R2 mette a disposizione due istanze, che pur essendo identiche possono essere usate in parallelo.

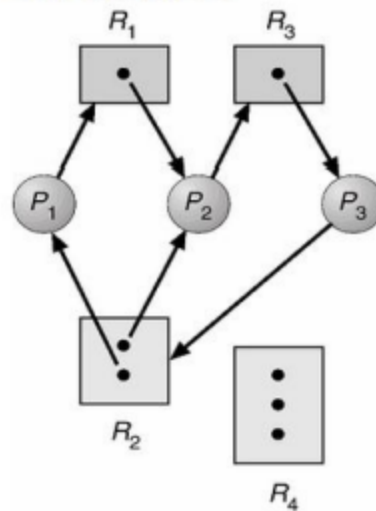
Non ho conflitti dato che il principio di mutua esclusione si applica solo sulla stessa istanza.

Il processo P3 spezza la possibilità di formare un'attesa circolare, non avendo richieste pendenti su altre risorse già occupate.

Una volta che ha terminato l'utilizzo di R3, lo rilascia sbloccando P2 (e per propagazione anche P1).

Non riesco a formare nessun ciclo (che torni alla risorsa stessa), non ho deadlock

Con deadlock



Come prima abbiamo tre processi (P1, P2, P3) e quattro risorse (R1, R2, R3, R4). R1 ed R3 hanno un'unica istanza, mentre R2 ne ha due ed R4 tre.

In particolare:

Il processo P1 ha richiesto R1 che però è già stato assegnato a P2, quindi rimane in attesa.

La risorsa R2 mette a disposizione due istanze, che pur essendo identiche possono essere usate in parallelo.

Non ho conflitti dato che il principio di mutua esclusione si applica solo sulla stessa istanza.

Il processo P3 richiedendo la risorsa R2 che non ha istanze disponibili, si mette in attesa

Si sono creati così due cicli:

$P1 \rightarrow R1 \rightarrow P2 \rightarrow R3 \rightarrow P3 \rightarrow R2 \rightarrow P1$

$P2 \rightarrow R3 \rightarrow P3 \rightarrow R2 \rightarrow P2$

I processi P1, P2 e P3 sono in deadlock.

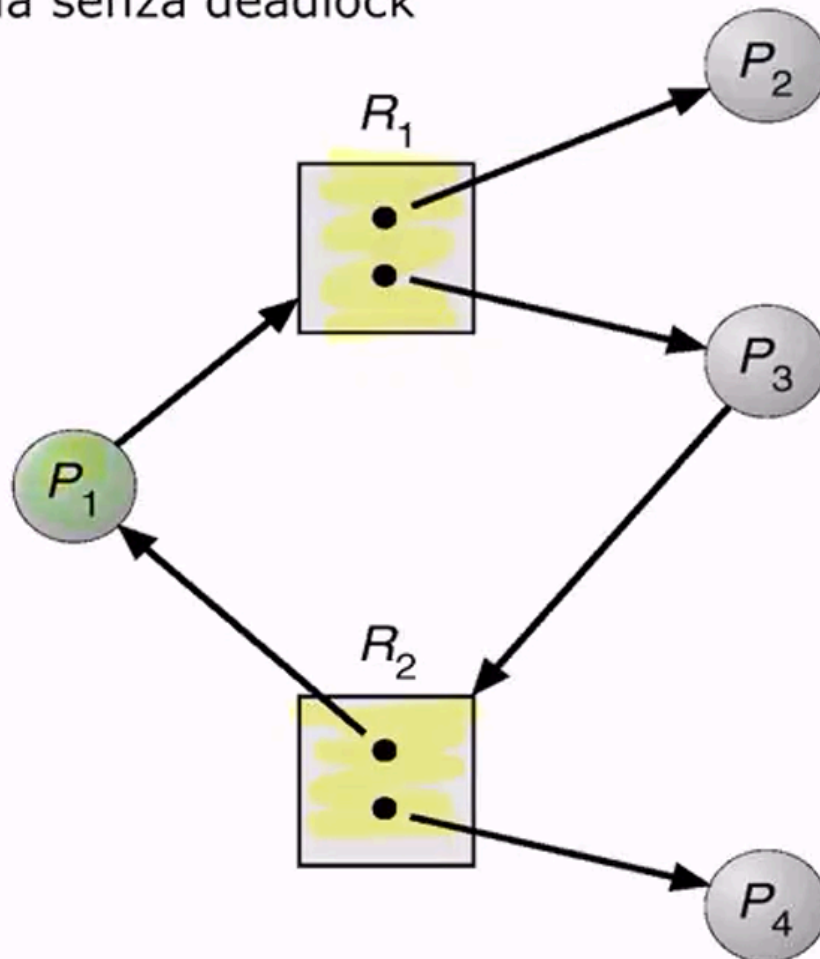
Il processo P2 sta infatti aspettando la risorsa R3 che è tenuta dal processo P3, il quale sta aspettando che P1 o P2 rilascino un'istanza di R2. In più il processo P1 sta aspettando che P2 liberi R1. I

processi potrebbero rimanere in questo stato di attesa indefinitamente.

Se il grafo non contiene cicli, allora nessun processo è in stallo.

Se contiene cicli, può esserci una situazione di stallo: se tra le risorse coinvolte nel ciclo, non c'è almeno un'istanza in più disponibile, ho il deadlock.

Con ciclo ma senza deadlock



Abbiamo quattro processi (P1, P2, P3, P4) e due risorse (R1, R2) ognuna delle quali con due istanze.

In particolare:

Esiste un ciclo tra $P1 \rightarrow R1 \rightarrow P3 \rightarrow R2 \rightarrow P1$.

Ho due processi P2 e P4 che non sono coinvolti nel ciclo.

Questa situazione dimostra che se le risorse hanno più istanze, posso avere dei casi in cui ho cicli ma non deadlock. In questo caso, P2 e P4 prima o poi termineranno la loro computazione e rilasceranno un'istanza, che potrà essere utilizzata da P1 e P3 per superare lo stallo.

Metodi di gestione del deadlock

- Ignorare - Se lo stallo si verifica con probabilità molto bassa, posso evitare di attuare un meccanismo di gestione del deadlock.

Questa politica normalmente consente all'utente che non vede progredire il suo processo, di abortirlo o riavviare tutto il sistema manualmente.

Al riavvio della macchina, sarà difficile che si ripresentino nello stesso ordine le richieste alle risorse che avevano condotto al deadlock precedente.

Può sembrare strano, ma questa soluzione è quella usata dalla maggior parte dei sistemi operativi, Unix e Windows compresi. Perfino la Java Virtual Machine non gestisce deadlock, viene lasciato tutto al programmatore

Se l'evento è quindi molto raro, investire anche una minima quantità di tempo, per eseguire periodicamente un'azione di prevenzione del deadlock, porterebbe nel lungo termine ad una quantità non banale di lavoro.

Risulta più efficiente, semplicemente far terminare i processi non in stallo, e successivamente riavviare la macchina.

- Prevenzione del deadlock - Garantendo a priori, al momento della richiesta delle risorse, che non si verifichino stalli.
- Evitare il deadlock - Bloccando le richieste di risorse, nel caso esse possano causare deadlock.
- Rilevazione e recupero del deadlock - Lascio avvenire il deadlock (evitando frequenti azioni di controllo), verificarne la presenza (in un periodo di tempo più lungo rispetto a quelli dei meccanismi di prevenzione) e recuperare il funzionamento del sistema nel miglior modo possibile

Prevenzione del deadlock

Prevenire il deadlock, significa far sì che almeno una delle 4 condizioni formali, non sia soddisfatta.

Studiamole singolarmente:

- Verificare che il principio di mutua esclusione, sia applicato solo su quelle risorse che sono necessariamente non condivisibili, evitando di estenderlo

anche a risorse condivisibili (per minimizzare le condizioni di stallo)

- Impedire che un processo possa chiedere una nuova risorsa, se ne possiede già un'altra.

Due tecniche:

- Far sì che il processo ottenga tutte le risorse di cui ha bisogno prima dell'inizio dell'esecuzione.

Se ad esempio un processo deve modificare mille file e infine stamparli, accederà a tutte le risorse all'avvio (stampante compresa) e ne manterrà il possesso per tutta la durata della computazione e solo quando termina le rilascerà.

Se il sistema operativo non riesce a dargli tutti gli accessi, il processo non viene nemmeno attivato. Lo svantaggio di questa soluzione non è da poco.

Può infatti passare molto (troppo) tempo prima che il processo usi tutte le risorse da lui bloccate all'apertura, mantenendo inutilmente altri processi in attesa rallentando di conseguenza il parallelismo del sistema

- Prima di poter richiedere nuove risorse, il processo deve necessariamente rilasciare tutte quelle che già detiene.

Bisogna strutturare quindi il codice, per far sì che una risorsa venga utilizzata "nella sua interezza" prima di utilizzarne un'altra, e che venga rilasciata per poter esser modificata da altri processi.

- Imporre la pre-emption ove essa non crea inconsistenza delle informazioni

La **pre-emption** consiste nel forzare un processo a rilasciare risorse che possiede, anche se non ha terminato di utilizzarle. Questo permette di sbloccare risorse e prevenire stalli, ma deve essere applicata con attenzione per non generare inconsistenza nei dati.

Due tecniche:

- **Rilascio Anticipato delle Risorse Detenute**

Quando un processo detiene delle risorse e richiede nuove risorse che non sono immediatamente disponibili (perché assegnate ad altri o in attesa), gli viene imposto di rilasciare tutte le risorse che già possiede.

- **Risultato:** Le risorse rilasciate vengono aggiunte a una lista di disponibilità per gli altri processi.
- **Ripresa:** Il processo rilasciato verrà ripreso solo quando tutte le risorse (vecchie e nuove) saranno contemporaneamente disponibili per essere assegnate.

Vantaggi:

- Si evita che un processo mantenga risorse inutilizzate durante l'attesa.
- Si riduce il rischio di deadlock perché si elimina la condizione di hold and wait.

Svantaggi:

- Introduce overhead per la gestione delle risorse: un processo potrebbe essere interrotto e rilanciato più volte.
- Può rallentare l'esecuzione complessiva se molti processi competono per le stesse risorse.

Esempio:

Un processo ha in uso una stampante e chiede accesso a un file condiviso che non è disponibile. In questa tecnica, il processo rilascia la stampante, e la sua esecuzione viene sospesa fino a quando la stampante e il file saranno entrambe disponibili.

◦ Controllo dinamico sulla prelazione

Se un processo chiede una risorsa che non è disponibile, si va a verificare la condizione del processo che attualmente la detiene:

- **Se il processo detentore della risorsa è in attesa di altre risorse:** Gli viene imposto di rilasciare anticipatamente la risorsa, rendendola disponibile per il nuovo richiedente.
- **Se il processo detentore sta effettivamente utilizzando la risorsa:** Il richiedente deve attendere che questa venga rilasciata normalmente.

Vantaggi:

- Ottimizza l'uso delle risorse "ferme" in attesa di essere usate.
- Introduce un meccanismo dinamico per sbloccare situazioni di stallo potenziale.

In poche parole, si applica la pre-emption solo sui processi in attesa di altre risorse.

- **Impedire attesa circolare**

Introduciamo un ordinamento globale univoco su ogni risorsa all'interno dell'elaboratore: assegnamo un indice ad ogni risorsa, più esso è piccolo, più il suo uso è frequente.

Applichiamo poi la seguente politica:

- Se un processo chiede istanze della risorsa R_j , e detiene già delle risorse R_i , se $i < j$, cioè le sue risorse sono più importanti, le risorse R_j gli vengono assegnate (subito, se disponibili, altrimenti dovrà attendere).
- Se un processo chiede istanze della risorsa R_j , e detiene già delle risorse R_i , se $i \geq j$, cioè le sue risorse sono meno importanti, allora il processo prima deve rilasciare tutte le sue risorse R_j , poi dovrà richiederle tutte, nuove e vecchie.

Ad esempio se il processo P detiene la risorsa R25, potrà chiedere solo risorse indicizzate da R26 in su, e non da R25 in giù, a meno che non rilasci tutte quelle che ha già.

Questa tecnica impedisce che più processi possano attendersi l'un l'altro: gli indici lo impediscono.

Evitare il deadlock

I metodi di prevenzione del deadlock, applicano alcune regole su come devono essere fatte le richieste di accesso alle risorse, garantendo che almeno una delle quattro condizioni perché si abbiano stalli non sia verificata.

Il problema è che sono regole molto restrittive, che determinano un basso rendimento del sistema e di utilizzo delle periferiche.

Vogliamo allora verificare a priori se la sequenza di richieste e rilasci di risorse effettuate da un processo porta a stalli, tenendo conto delle sequenze dei processi già accettati nel sistema.

Per fare ciò, ho bisogno di verificare a priori il comportamento dei processi, in modo tale da capire, se esiste un ordine d'esecuzione delle attività che non porti a stalli.

Uno stato si dice sicuro per una serie di n processi $\langle P_1, P_2, P_3 \dots \rangle$ se esiste almeno una sequenza sicura, ossia una sequenza di n processi per cui le richieste di ogni processo P_i della sequenza possono essere soddisfatte con le risorse attualmente disponibili più quelle detenute dai processi P_j , con $j < i$ (i processi che precedono P_i nell'ordine di esecuzione, rilasceranno le risorse, rendendole quindi disponibili a P_i).

Se non esiste una tale sequenza, lo stato si dice non sicuro.

Uno stato sicuro garantisce l'assenza di deadlock, ma uno stato non sicuro non significa necessariamente deadlock.

In caso di istanze singole delle risorse, possiamo verificare se una richiesta conduca ad una situazione di stallo, utilizzando una variante del grafo di allocazione delle risorse, in cui aggiungiamo gli archi di prenotazione (da un processo a una risorsa, tratteggiati).

All'avvio, un processo deve specificare quali risorse utilizzerà durante tutto il corso della sua evoluzione, così da generare i suoi archi di prenotazione.

Quando richiederà una risorsa (disponibile), si controlla se la conversione del relativo arco di prenotazione ad arco di assegnazione causa cicli nel grafo.

Se ne causa, lo stato non è sicuro e il processo deve attendere.

Per la gestione di istanze di risorse multiple, si può utilizzare l'algoritmo del banchiere.

Per operare:

- Massimo numero di istanze delle risorse deve essere noto.
- Assume che ogni processo rilasci la risorsa utilizzata in un tempo finito.
- Necessita delle seguenti strutture dati per mantenere le informazioni sui processi:
(n = numero processi, m =numero di risorse nel sistema di elaborazione):
 - Available[1 ... m]: Per ogni elemento, numero di istanze del tipo m disponibili.
 - Allocation: matrice $n*m$, per ogni processo (riga), il numero di istanze attualmente assegnate, per ogni tipo di risorsa
 - Max: matrice $n*m$, per ogni processo (riga) il numero massimo di istanze utilizzabili, per ogni tipo di risorsa (colonna)
 - Need: matrice $n*m$, per ogni processo le risorse massime che può ancora richiedere (matrice max - matrice allocation)
- Necessita per operare di un ulteriore algoritmo: partendo da uno stato iniziale, ed aggiungendo una nuova richiesta, vogliamo verificare che il nuovo stato prossimo sia anch'esso sicuro. (Algoritmo di verifica dello stato sicuro)

Quando un processo fa una richiesta, si controlla se è valida (\leq need di quel processo, in caso contrario solleva errore) e se è soddisfacibile (\leq available, altrimenti il processo dovrà attendere), vengono modificate le strutture dati per simulare l'assegnazione della risorsa, ed esegue l'algoritmo di verifica dello stato sicuro:

Si cercano tutti i processi il cui fabbisogno (need) può essere soddisfatto con le attuali disponibilità più le risorse assegnate ai processi analizzati precedentemente (che le rilasceranno al termine)

Se alcuni processi rimangono scoperti, lo stato non è sicuro, si ripristinano i vecchi valori delle strutture dati e il processo deve attendere.

In caso contrario, il processo ottiene le risorse richieste.

Rilevamento e ripristino del deadlock:

Lascio che il deadlock si verifichi, e uso degli algoritmi per rilevarlo e risolverlo. In caso di istanze singole delle risorse si può usare il grafo di attesa: una variante semplificata del grafo di allocazione in cui i nodi risorsa non sono rappresentati.

Se nel grafo ci sono cicli, i processi coinvolti sono in un deadlock.

In caso di istanze multiple si può usare un algoritmo simile all'algoritmo di verifica dell'algoritmo del banchiere.

Utilizza le strutture dati:

- Available[1 ... m]: Per ogni elemento, numero di istanze del tipo m disponibili.
- Allocation: matrice $n \times m$, per ogni processo (riga), il numero di istanze attualmente assegnate, per ogni tipo di risorsa
- Request: matrice $n \times m$, per ogni processo (riga), quante istanze di ciascuna risorsa ha richiesto.

Si cercano tutti i processi le cui richieste possono essere soddisfatte con le attuali disponibilità più le risorse detenute dai processi analizzati precedentemente (si suppone che terminino e le rilasciano, se non lo fanno, il deadlock verrà rilevato successivamente).

Se alcuni processi rimangono scoperti, questi sono coinvolti in un deadlock.

Per risolverlo si possono terminare tutti i processi coinvolti (viene risolto, ma si possono perdere i risultati di lunghe computazioni), oppure terminarli uno per volta in base a qualche criterio, finché non si risolve il deadlock (evitando di terminare sempre gli stessi causando la starvation).

L'algoritmo di rilevamento può essere eseguito ad intervalli di tempo prestabiliti, oppure quando l'utilizzo della CPU scende sotto una certa soglia (se c'è un deadlock, sempre più processi saranno coinvolti quindi continuerà a scendere).