Appunti di Sistemi operativi

Marco Zanchin

September 2023

Contents

1 Storia dei sistemi operativi **4**
1.1 Generazione 1 - tubi a vuoto 4
1.2 Generazione 2 - Transistor 4
1.3 Generazione 3 - circuiti integrati 5
1.3.1 Multiprogramming 6
1.3.2 Multiprogrammazione - gestione della memoria 7
1.3.3 Multiprogrammazione - gestione dispositivi 8
1.3.4 Multiprogrammazione - requisiti hardware 8
1.3.5 Il througput 8
1.3.6 Altri concetti introdotti con il multiprogramming 9
1.3.7 Il timesharing 10
1.4 Generazione 4 - Circuiti integrati LSI/VLSI 11

2 Richiami di architettura e system call **11**
2.1 La CPU 12
2.1.1 I registri di controllo 12
2.2 Il concetto di Interrupt 13
2.2.1 Hardware interrupt 13
2.2.2 Richieste di interrupt 15
2.2.3 Program interrupt 18
2.3 Dispositivi di I/O e DMA 19
2.3.1 Comunicazione tra la CPU e le porte di I/O 19
2.3.2 I/O nei programmi 20
2.4 Introduzione alle system call 22

3 Processi **24**
3.1 Il memory layout dei programmi 24
3.2 Il concetto di processo 30
3.2.1 File e processi 30
3.3 Parallelismo e concorrenza 31
3.4 Stati di un processo 32
3.4.1 Stati di un processo in UNIX system V 33
3.5 Process control block 34

3.5.1 Strutture dati e PCB 36
3.5.2 Introduzione alla gestione della memoria 36
3.6 Implementazione dei processi: il context switch 37
3.6.1 Contesto di un processo 37
3.7 Creazione e terminazione di processi 38
3.7.1 Fork 39
3.7.2 Exec 39
3.7.3 Wait 40
3.8 Tipi di processo 40
3.8.1 Processi utente 40
3.8.2 Processi daemon 41
3.8.3 Processi kernel 42
3.8.4 Processi e booting 42
3.8.5 Terminazione di processi 43
3.9 System call in UNIX 43
3.9.1 Esempio di system call a basso livello 44

4 Threads **46**
4.1 Introduzione ai threads 46
4.1.1 Esempio di thread 47
4.1.2 Threads nello spazio user e kernel 49
4.1.3 Esempio threads in Java 50

5 Sincronizzazione dei processi **51**
5.1 Race conditions 53
5.2 Sezioni critiche 58
5.2.1 Tentativo 1 di implementare le sezioni critiche - disabilitare gli interrupt 59
5.2.2 Tentativo 2 di implementare le sezioni critiche - uso di variabili lock condivise 59
5.2.3 Tentativo 3 di implementare le sezioni critiche - uso di variabili turno condivise 60
5.2.4 Operazione indivisibile 60
5.3 Approccio algoritmico alle sezioni critiche 62
5.3.1 Logica 62
5.3.2 Uso delle variabili turno - secondo tentativo 62
5.3.3 Uso delle variabili turno - terzo tentativo 63
5.3.4 Uso delle variabili turno - terzo tentativo 63
5.3.5 Algoritmo di Dekker 64
5.3.6 Algoritmo di Peterson (processi gentili) 64
5.4 Problemi classici della programmazione concorrente 65
5.4.1 Problema dei produttori e consumatori 65
5.5 Problema dei lettori e scrittori 67
5.5.1 Soluzione 68
5.6 Problema dei filosofi a cena 69
5.6.1 Soluzione con semafori 69

6 Semafori **72**
6.1 Uso dei semafori per l'implementazione delle sezioni critiche . . . 72
6.1.1 Bounded concurrency . . . 73
6.1.2 Impementazione dei semafori . . . 73

7 Gestione della memoria **73**
7.1 Traduzione, linking, loading: alcune precisazioni . . . 73
7.1.1 Esempio programma in assembly . . . 74
7.2 Modello di allocazione della memoria . . . 76
7.3 Gerarchia di memoria . . . 77
7.4 Riuso della memoria . . . 78
7.5 Allocazione della memoria . . . 82
7.5.1 Allocazione contigua . . . 82
7.5.2 Allocazione non contigua . . . 82
7.5.3 Allocazione non contigua - idea . . . 83
7.6 Paginazione e segementazione . . . 84
7.6.1 Paginazione . . . 84
7.6.2 Esempio di paginazione . . . 85
7.6.3 Segmentazione . . . 86
7.6.4 Esempio di segmentazione . . . 87
7.7 Memoria virtuale . . . 87
7.7.1 Demand paging . . . 87
7.7.2 Traduzione con TLB . . . 89
7.7.3 Esempio di TLB . . . 89
7.7.4 Protezione della memoria . . . 92
7.7.5 Dettagli implementativi del paging . . . 94

1 Storia dei sistemi operativi

Sapere la storia dei SO è utile perché certe scelte/caratteristiche/funzionalità dei sistemi attuali sono più facili da capire conoscendo:

- il contesto storico nel quale sono state introdotte
- quanto accadeva prima che fossero introdotte
- perché siano state introdotte

1.1 Generazione 1 - tubi a vuoto

I calcolatori presenti in questa generazione non hanno un sistema operativo, l'idea è che le macchine venivano utilizzati da scienziati per effettuare calcoli. Al contrario della situazione odierna, dove si hanno tanti dati da gestire e pochi calcoli.
...

1.2 Generazione 2 - Transistor

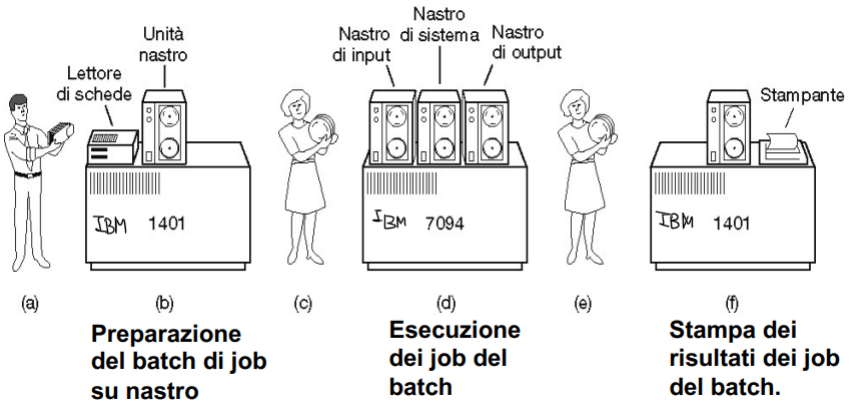
...
L'intoduzione dei transistors negli anni 50 ha cambiato l'industria dei computer, essi diventano abbastanza affidabili da poter essere venduti ad acquirenti. Queste macchine sono quelli che ora chiamiamo mainframes.
In questa generazione per la prima volta Compaiono dei dispositivi, ossia strumenti per gli utenti utilizzati per interagire con il calcolatore.

Alcuni esempi:

- Il lettore di schede (input), il quale prende le informazioni presenti sulle schede e le scarica sulla RAM.
- Nastri: supporto magnetico che memorizza un informazione

Per eseguire un **job**, ossia un programma inciso sulle schede perforate in FORTRAN, bisognava portare la scheda nella sala input, ossia dove si effettuava la lettura delle schede, era tuttavia necessario aspettare molto tempo per l'output, lasciando per molto tempo in inutilizzo la CPU.

Per risolvere questo problema fu inventato il **batch system**: ossia raccogliere un insieme di jobs e leggerli utilizzando un computer economico, come l'IBM 1401, che era pensato per leggere schede perforate e stampare output, ma non per calcoli matematici.
Una volta che tutte le schede erano state scritte nell'*unità nastro* era arrivato il momento di effettuare i calcoli utilizzando una macchina più potente come l'IBM 7094, che possedeva l'antenato di un sistema operativo odierno il quale eseguiva ogni programma scrivendo l'output di ciascuno sul nastro.
Alla fine il nastro di output viene portato su un altra macchina economica e tutti i risultati vengono stampati.



Il sistema operativo presente nel nastro di sistema gestisce la RAM, che è divisa in due zone:

- *System area*: zona dove è stato caricato il sistema operativo stesso
- *User area*: zona dove vengono caricati i job della batch.

Questa divisione in area di sistema e user è presente ancora oggi.

Effetti dell'introduzione del S.O batch:

- Grazie ai batch diminuiscono i tempi di inutilizzi della CPU, infatti dopo un job parte subito quello successivo. Questo è fondamentale perchè i processori sono estremamente costosi, idealmente dovrebbero lavorare 24 ore al giorno.
- Per l'utente il tempo di *turn-around* può dilatarsi, ossia può aspettare di più il risultato della stampante, poichè finche il nastro contiene n lavori bisogna aspettare.

Dunque i sistemi batch permettono un uso più efficiente della CPU, ma non sono finalizzati a migliorare il servizio all'utente.

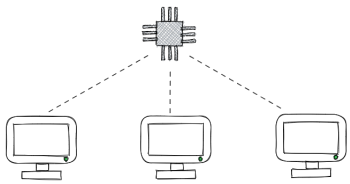
1.3 Generazione 3 - circuiti integrati

In questa generazione le macchine elaborano anche dati a fine commerciali, che richiedono meno CPU ma più I/O.
Viene introdotto:

- Hard disk: ci si accorge che bisogna trattare una grande mole di dati che la RAM non riesce a contenere
- Terminale: inteso come coppia tastiera e monitor, per fare in modo che più persone possano usare lo stesso processore.

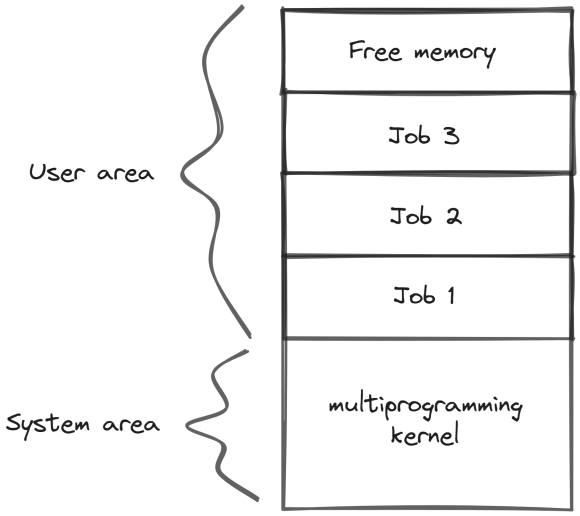
- Linguaggi ad alto livello: ad esempio il C.
- S.O interattivi con *multiprogrammazione* e *timesharing*

Dato il costo elevato dei processori si pensa a *macchine condivise*, ossia più terminali che utilizzano la stessa CPU



1.3.1 Multiprogramming

L'idea è quella che in ogni istante di tempo nell'area utente della RAM possiamo avere n job "in esecuzione".



Quando un job inizia un'operazione di I/O la CPU non rimane inattiva ma viene assegnata ad un altro job.

Esistono alcune differenze con i sistemi odierni:

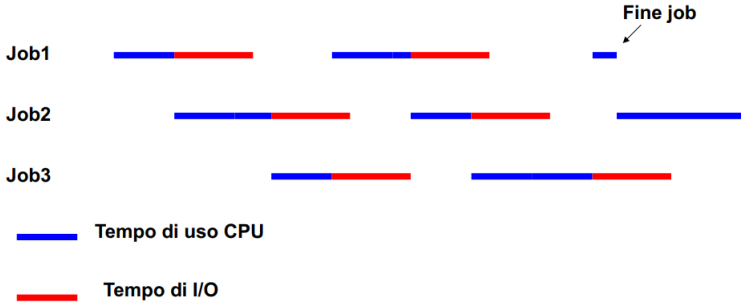
- I job (o processi) non possono occupare zone di RAM non contigue con questa tecnologia
- I programmi devono essere tutti in RAM, con la tecnologia odierna no.

Un esempio di esecuzione con n job in RAM è la seguente:

- La CPU sta eseguendo un'istruzione di *job 1*

- Si passa all'istruzione successiva
- Finchè *job 1* non effettua un'operazione di I/O si continuerà ad eseguire istruzioni del medesimo job, se dovesse eseguire un operazione di I/O *job 1* perde il processore e viene assegnato a un altro job in RAM.

In questo modo evitiamo che quando un job vuole effettuare un'operazione di I/O (che è lenta) si aspetti finchè finisca.



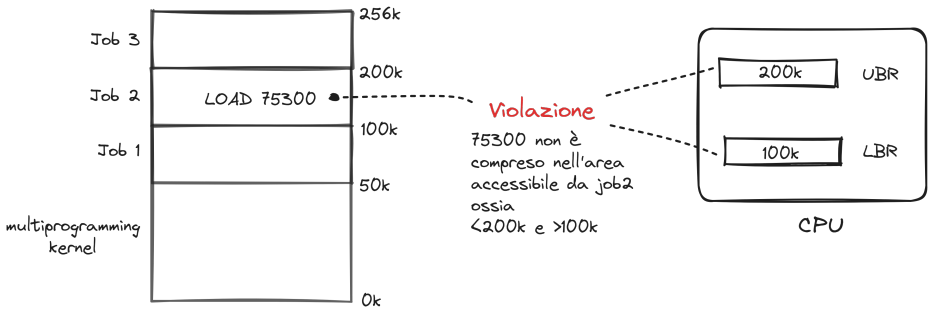
Notare che è necessario un algoritmo di scheduling per scegliere il job da eseguire tra quelli disponibili.

1.3.2 Multiprogrammazione - gestione della memoria

La multiprogrammazione prevede anche di gestire la memoria.

Ad esempio se *job 2* è in esecuzione, il processo non deve scrivere o leggere nelle porzioni di area degli altri job o del kernel.
Per effettuare questo controllo utilizziamo due registri:

- **UBR**: Upper Bound Register, memorizza l'estremo superiore della memoria allocata al job in esecuzione.
- **LBR**: Lower Bound Register, memorizza l'estremo inferiore della memoria allocata al job in esecuzione.



Quando un job è in esecuzione i registri LBR e UBR vengono settati con gli estremi dell'area accessibile dal job nella RAM.

1.3.3 Multiprogrammazione - gestione dispositivi

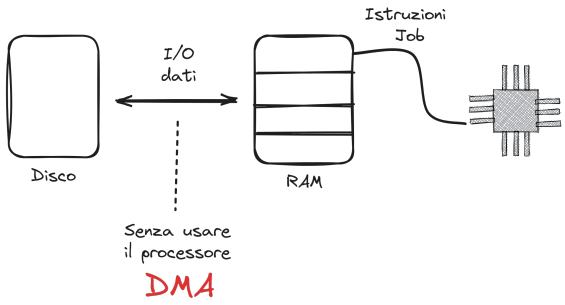
- Ogni job deve poter usare i dispositivi
- Nessun job deve poter interferire sull'uso dei dispositivi da parte degli altri job.

Esistevano due modalità principale atte alla gestione dei dispositivi:

- **partitioning**: ad ogni job vengono assegnati i dispositivi staticamente. Esempio: una porzione riservata su disco
- **pooling**: ad ogni job vengono assegnati i dispositivi dinamicamente. Man mano che un job ha bisogno di spazio su disco lo conferiamo.

1.3.4 Multiprogrammazione - requisiti hardware

- L'hardware deve consentire alla CPU e ai dispositivi di I/O di lavorare in parallelo, ciò è possibile grazie al **DMA** (Direct Memory Access)



- Quando un dispositivo di I/O ha terminato di eseguire un'operazione per conto di un ov il job deve essere inserito nell'elengo dei job schedulabili. Ciò è possibile grazie alla tecnica dell'**interrupt**.

1.3.5 Il througput

Il throughput

Il **troughput** è il rapporto tra il numero di programmi eseguiti e il tempo.

- Programma **CPU-bound**: uso intenso della CPU, poco I/O
- Programma **I/O-bound**: tanto I/O, poco uso CPU

Per ottimizzare il throughput è opportuno che lo scheduler dia priorità ai programmi *I/O-bound*

1.3.6 Altri concetti introdotti con il multiprogramming

Con i sistemi multiprogrammati vengono introdotti altri due concetti:

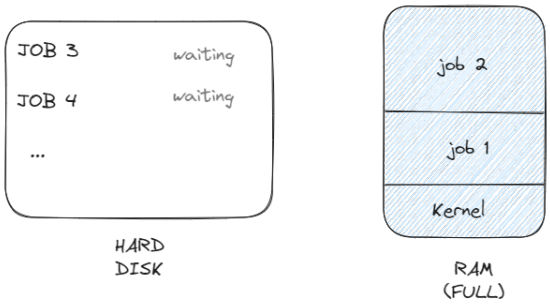
- **Program priority**: Ad ogni programma viene assegnata una priorità, considerata in fase di scheduling per scegliere il programma da schedulare.
- **Preemption**: Quando la CPU viene tolta forzatamente a un processo e data a uno con priorità maggiore.

Per migliorare il *throughput* possiamo avvalerci di questi concetti e utilizzarli nel seguente modo:

- I programmi CPU-bound devono avere priorità inferiore rispetto ai programmi I/O bound.
- Quando un programma I/O bound termina un'operazione di I/O, se è in esecuzione un programma CPU-bound questo subisce la preemption, in modo da dare il processore al programma I/O bound.

L'introduzione degli *hard disk* consente alla multiprogrammazione di arricchirsi con lo **spooling**.
SPOOL: **S**imultaneous **P**heripheral **O**peration **O**n **L**ine

- I programmi possono essere caricati su disco, non appena un job termina e libera la sua partizione RAM, il S.O può assegnarla a uno dei job su disco.



Dunque non servono più macchine dedicate alla preparazione dei batch e alla stampa dei risultati.

1.3.7 Il timesharing

Viene inoltre introdotto il concetto di **timesharing**, ossia una versione della multiprogrammazione:

- Ad ogni job viene assegnato un quanto di tempo (*time slice*) allo scadere del quale la CPU viene assegnata ad un altro job.

  In questo modo si crea l'illusione di esecuzione parallela di n processi.

- Adatto per sistemi dotati di più terminali su cui lavorano utenti diversi: ogni utente ha l'impressione di interagire continuamente con la macchina.

- Siccome abbiamo programmi di diversi utenti sulla stessa macchina dobbiamo garantire la protezione dei dati di ogni utente

Tuttavia il timesharing **penalizza il throughput**, ma **migliora i tempi di risposta**.
Si va a privilegiare la *user convenience* a discapito dell'*efficienza* dell'uso delle risorse.

Riassumendo quindi:

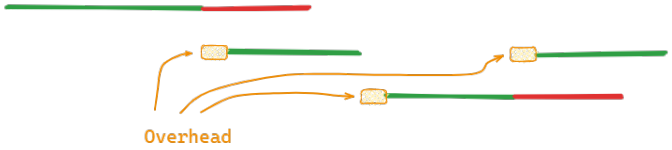
- **Multiprogramazione**: priorità, preemption, throughput
  L'obiettivo è quello di migliorare il throughput

- **Timesharing**: round robin, time slice tempi di risposta
  L'obiettivo è quello di migliorare i tempi di risposta

Osservazione: sottrarre la CPU ad un job e assegnarla ad un altro (*context switch*) è un'operazione onerosa in termini di tempo.
Durante il context switch la CPU non lavora per nessun job, quindi non efffettua

lavoro utile. Si parla di **overhead**.

Avere context switch frequenti garantisce maggiore interattività agli utenti ma accresce l'overhead a danno del throughput. è necessario trovare un equilibrio.



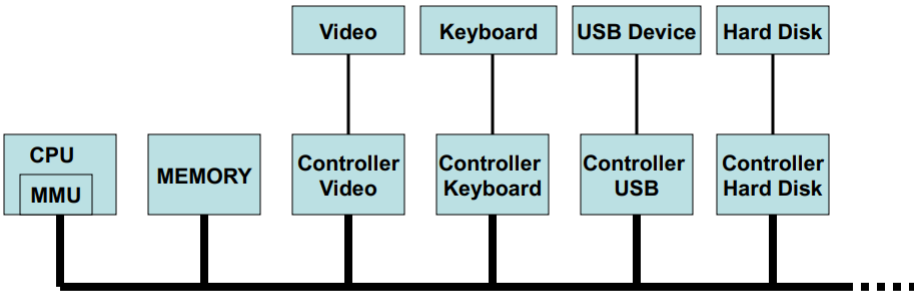
1.4 Generazione 4 - Circuiti integrati LSI/VLSI

Si parla di generazione 4 da quando iniziamo ad avere i circuiti integrati e si diffondono i PC.
Data la diffusione al grande pubblico si cerca di migliorare l'usabilità introducendo le GUI.

- Dagli anni '80 si diffondono anche i sistemi real time per la gestione delle applicazioni real time, cioè programmi che rispondono ad un ambiente esterno
- Dagli anni '90 si diffondono i sistemi operativi distribuiti per la gestione dei sistemi distribuiti.

2 Richiami di architettura e system call

Il sistema operativo interagisce direttamente con l'hardware.
Rappresentiamo l'hardware come nella figura di seguito:



2.1 La CPU

La CPU *preleva* le istruzioni dalla memoria, le *decodifica* e le *esegue*.

Dunque il ciclo tradizionale di funzionamento della CPU è il seguente:

1. fetch
2. decodifica
3. esecuzione
4. GOTO 1

La CPU si avvale di:

- **Registri generali**: costituiscono il ptimo livello della gerarchia della memoria, memorizzano variabili, risultati temporanei, utilizzati fondamentalmente dalla ALU.
- **Registri di controllo**: Servono per controllare il funzionamento della CPU stessa

2.1.1 I registri di controllo

- Program counter (PC): Indirizzo prossima istruzione da prelevare.
- Stack pointer (SP): Puntatore al top della pila dei record di attivazione.
- Program status word (PSW): insieme di registri
  - Privileged mode (PM): bit di modalità *user*/*kernel*(anche le istruzioni privilegiate sono eseguibili)
  - Condition code (CC): Codici di condizione impostati da istruzioni di confronto, codici di proprietà di operazioni aritmetiche
  - Interrupt Mask (IM): bit usati per gestire interrupt
  - Interrupt Code (IC): bit usati per gestire interrupt
  - Memory protection information (MPI): bit usati per fare protezione della memoria (ad esempio UBR/LBR negli anni '80)

Ruolo del bit PM (Privileged Mode):

- Le istruzioni privilegiate, ossia le istruzioni che modificano la PSW, come le istruzioni per cambiare i registri LBR/UBR sono eseguibili solo in *stato kernel*.

2.2 Il concetto di Interrupt

Interrupt

Gli **interrupt** costituiscono un meccanismo per notificare alla CPU un evento o una condizione avvenuti nel sistema che devono essere trattati dal sistema operativo. L'obiettivo è:

- Interrompere il normale ciclo di esecuzione della CPU
- Richiedere l'intervento del sistema operativo

Come si manifesta l'interrupt? Con un segnale su una linea del bus di sistema. Questo segnale può essere inviato in due casi:

- **Hardware interrupt**: Controllore di un dispositivo o clock (ad esempio un ciclo infnito)
- **Program interrupt**: Generato dal programma in esecuzione

2.2.1 Hardware interrupt

Distinguiamo tra:

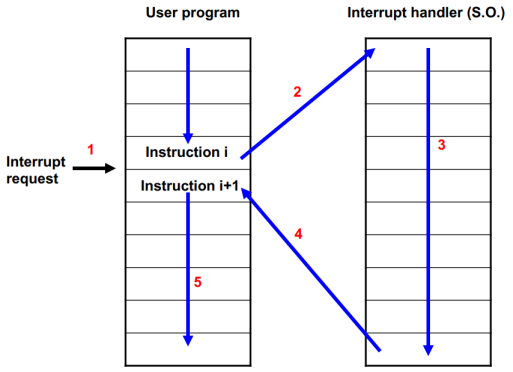
- *I/O interrupt*: usati dai dispositivi I/O per notificare eventi alla CPU che richiedono di essere trattati dal SO.

- *Timer interrupt*: vengono usati per misurare intervalli di tempo e innescare eventi periodici, usati dal clock per notificare il tick.

Si tratta di eventi **asincroni** rispetto al programma in esecuzione, infatti il programma non ha modo di prevedere la loro esecuzione.

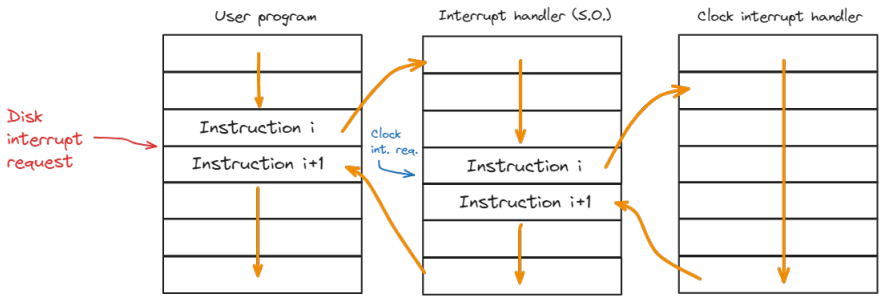
Fasi dell'interrupt hardware

Abbiamo due programmi, un programma utente che gira in programma user e l'interrupt handler che gira in modalità kernel (questo contiene ad esempio il codice per mettere un programma in schedulazione dopo un'attività di I/O).



1. La CPU mentre sta eseguendo l'istruzione *I* di un programma P riceve un interrupt request che verrà presa in considerazione prima di prelevare l'istruzione *I+1*
2. Terminata l'esecuzione dell'istruzione I la CPU sospende P e salta alla procedura di gestione dell'interrupt
3. L'interrupt handler, che è parte del SO, gestisce l'interrupt
4. L'interrupt handler restituisce il controllo a P o P' (in base all'algoritmo di scheduling)
5. P o P' riprende l'esecuzione

Cosa succede se arriva un'altra interrupt mentre l'interrupt handler è in esecuzione?



L'interrupt handler viene fermato e viene eseguito il clock interrupt handler, il meccanismo è uguale all'esempio precedente se non fosse che l'interrupt handler e il clock interrupt handler girano entrambi in modalità kernel

2.2.2 Richieste di interrupt

Come vengono gestiti gli interrupt a cascata? Gli interrupt vengono organizzati in classi di priorità.

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

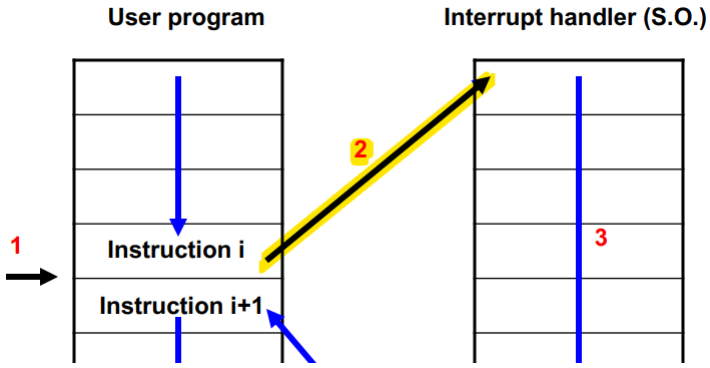
Quando si gestisce un interrupt request si *ignorano* temporaneamente le richieste di interrupt della medesima classe e edelle classi con minor priorità, la cui gestione viene rimandata. Tale richieste rimangono **pendenti**.

Come vengono ignorate le interrupt request?
Il registro **PSW** contiene alcuni bit che svolgono il ruolo di *interrupt mask* (IM). L'interrupt mask è una sequenza di bit che mi dicono quali interruzioni sono abilitate e quali mascherate.
Esistono due implementazioni possibili per l'interrupt mask:

- l'IM contiene un bit (enabled/masked off) per ogni classe di interrupt
- l'IM contiene un valore m per abilitare tutti e soli gli interrupt di priorità $>= m$

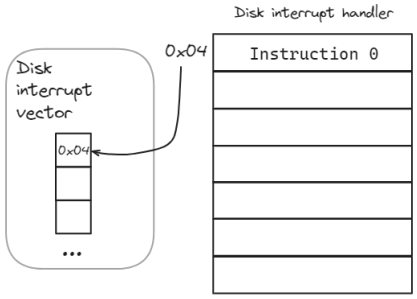
Quando il programma esegue in modalità user, tutti gli interrupt sono abilitati. Quando il controllo passa ad un interrupt handler, l'IM viene settato in modo opportuno. Come? Chiariamo subito.

Andiamo a vedere come a livello hardware vengono settati i registri per passare da user a kernel ed eseguire una porzione di S.O. Oppure come nel secondo esempio, eseguire una diversa porzione del S.O.



Per ogni classe di interrupt abbiamo un **interrupt vector**, memorizzato nella system area, che memorizza:

- L'indirizzo della prima istruzione dell'interrupt handler, da assegnare al PC per avviare l'interrupt handler.



- Il valore da assegnare alle componenti rimanenti della **PSW**.
  Per esempio quando è in esecuzione il disk interrupt handler settiamo l'interrupt mask per mascherare tutte le interrupt di priorità <= a quella del disco.

Gli interrupt vector vengono inizializzati in memoria durante la fase di booting.

Per quanto riguarda il salvataggio dello stato dei registri del programma che subisce l'interrupt abbiamo la **SRIA**, ossia **S**aved **R**egister **I**nformation **A**rea. Essa salva lo stato di:

- Program counter
- Stack pointer
- Program Status word

Ora spieghiamo dettagliatamente, avvalendoci delle informazioni acquisite, cosa succede quando arriva un interrupt:

1. Quando arriva il segnale di interrupt, l'hardware imposta l'**interrupt code** (IC) della PSW che serve per comunicare al SO la causa dell'interruzione, che verrà esaminata dall'interrupt handler.
2. I valori dei registri PC, SP e PSW (tutta la PSW o PSW + PC e SP in base alla scuola di pensiero) vengono salvati nell' apposita **SRIA**. I registri generali non sono stati ancora salvati.
3. I registri PC e PSW vengono impostati in base ai valori memorizzati nell'**interrupt vector**, L'interrupt handler può pertanto partire.

Fase 3, cosa fa l'interrupt handler:

1. Nelle prime istruzioni l'interrupt handler si occupa di salvare anche i registri generali. A questo punto i valori di tutti i registri della CPU relativi all'esecuzione del programma P sono stati salvati. (Sono normalissime istruzioi software che spostano il contenuto dei registri in una zona della RAM)
2. Esegue il codice apposito per gestire l'interrupt. Questo codice sfrutta il valore dell'IC in modo da gestire l'interrupt.
3. Salta allo scheduler, che seleziona il programma a cui assegnare il procesore.

Dettagli fase 4: cosa fa lo scheduler dopo che ha selezionato il programma P' che deve riprendere l'esecuzione

1. I registri generali che abbiamo salvato nelle prime istruzioni dell'interrupt handler vengono ripristinati
2. Vengono ripristinati i registri di controllo che erano stati salvati nella **SRIA**. Spesso l'architettura offre l'istruzione **Interrupt RETurn** (IRET) per ripristinare, tutti insieme, i registri di controllo.

Alcune considerazioni riguardo il salvataggio del registri nella CPU che viene interrotta:

- Il **PC** deve essere salvato via hardware perchè salvare via software significa **eseguire un istruzione**, se noi stiamo eseguendo un qualsiasi programma questa istruzione di salvataggio PC non sarà presente.
- Ovviamente i registri della CPU devono essere salvati anche quando è un interrupt handler ad essere interrotto da un interrupt.

...

2.2.3 Program interrupt

Oltre agli hardware interrupt, esistono due tipi di program interrupt: eccezioni e software interrupt.

- **Eccezioni** o interrupt interni: sono errori a tempo di esecuzione
  - Eccezioni aritmetiche (divisione per zero)
  - Eccezioni di indirizzamento (pagge fault)
  - Violazione delle protezioni di memoria

  Queste situazioni non sono asicroni rispetto all'esecuzione del programma, ma sono **causati** dal programma.
- **Software interrupt** detti anche trap
  - Sono causati da un'istruzione apposita, solitamente chiamata software interrupt instruction, o semplicemente trap.
  - Sono usati dai programmi per chiedere esplicitamente l'intervento del S.O
  - Il linguaggio macchina prevede un'unica istruzione per chiedere tutti i possibili interventi del S.O., l'istruzione TRAP.
  - l'istruzione TRAP ha un parametro che specifica il tipo di intervento richiesto. Il parametro può essere passato in 2 modi:
    * Come operamdp della TRAP, se la TRAP prevede operandi
    * Ponendolo sullo stack

...

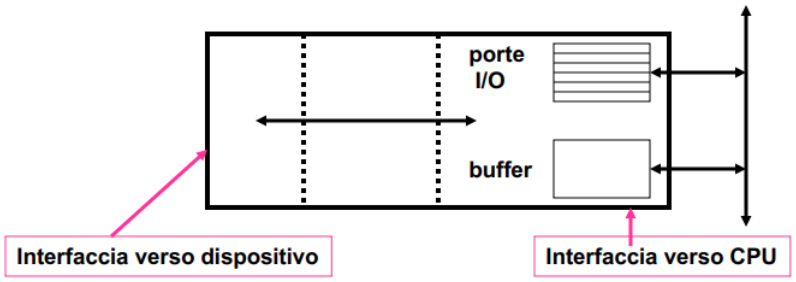
2.3 Dispositivi di I/O e DMA

Quando parliamo di un dispositivo di I/O come una tastiera, un hard disk, ecc. noi distinguiamo tra

- **Controller** o adapter: componente elettronica che comunica con il processore
- **Dispositivo vero e proprio**: componente mecanica che viene gestita dal controller, la CPU non comanda queste componenti meccaniche, lo fa il controller. Nel caso di un hard disk, testina mossa da un braccio meccanico

Le interfacce tra controller e dispositivo sono solitamente standard (SATA, USB, ecc.) e vengono usata dai programmi del S.O. che gestiscono i dispositivi (**driver**)
Questa interfaccia viene usata in modalità kernel e prevede:

- **registri di controllo**, detti anche porte di I/O:
  - Vengono usati dalla CPU per inviare comandi al controller (ad esempio: invia dati, accetta dati, spegniti o accenditi ecc.)
  - vengono usati dal controller per comunicare i risultati dei comandi e lo stato del dispositivo alla CPU
- un **buffer**: serve per memorizzare dati durante le operazioni di I/O



2.3.1 Comunicazione tra la CPU e le porte di I/O

- Soluzione 1: **porte di I/O gestite con istruzioni macchina ad hoc** ad esempio:

  $IN\ R, P$ Il valore dela porta P viene caricato nel registro R

  $OUT\ P, R$ La CPU carica il contenuto del registro R nella porta P

  Molti computer dei primi tempi funzionavano in questo modo, ad esempio l'IBM 360

- Queste istruzioni devono essere **privilegiate**, per impedire che in modalità user si riesca ad accedere ai dispositivi.
- Alcuni parti del driver devono essere in assembly, perchè ad alto livello non si possono avere istruzioni corrispondenti (IN R,P OUT R,P)

- Soluzione 2: **memory mapped I/O**
  Mancano delle istruzioni che facciano esplicito riferimento alle porte di I/O, ad ogni porta di I/O è assegnato un indirizzo di memoria (conservato in system area)
  - Questi indirizzi non sono visibili dai programmi, che devono invocare il S.O per accedere ai dispositivi
  - I driver possono essere scritti in C
  - Dato che una piccola porzione di RAM è assegnata alle porte I/O, il dispositivo ha accesso a una parte di RAM. Se queste parti vengono cachate, anche il dispositivo ha accesso alla cache, ciò può provocare problemi.
  - Se avessimo un bus dedicato tra CPU e memoria il tutto si complica poichè la MMU alcuni indirizzi non si riferiscono alla RAM ma a indirizzi I/O

2.3.2 I/O nei programmi

Abbiamo tre possibili soluzioni per eseguire l' I/O per conto di unprogramma P. Supponiamo di voler trasferire n byte verso il (buffer del controller del) dispositivo da un buffer di memoria b:

- Soluzione 1 - Programmed I/O
  Codice eseguito dal S.O su richiesta di P

```
for(i=0;i<n;i++){
    while(device_status_reg != READY){}
    buffer = b[i] // leggo da RAM e scrivo su buffer
}
```

  - buffer: buffer I/O per lo scambio dei dati.
  - device status register: tramite questa variabile il driver del dispositivo sa se il dispositivo è pronto per effettuare l'operazione.

  Tuttavia con questa soluzione la CPU rimane inutilizzata mentre aspetta che il dispositivo abbia finito di trattare ogni singolo byte. (attesa attiva o busy waiting) In sostanza il problema è che non consente il parallelismo tra il lavoro di un job e il lavoro di I/O per conto di un altro programma.

- Soluzione 2 - Interrupt driven I/O:
  Codice eseguito dal S.O. su richiesta di P:

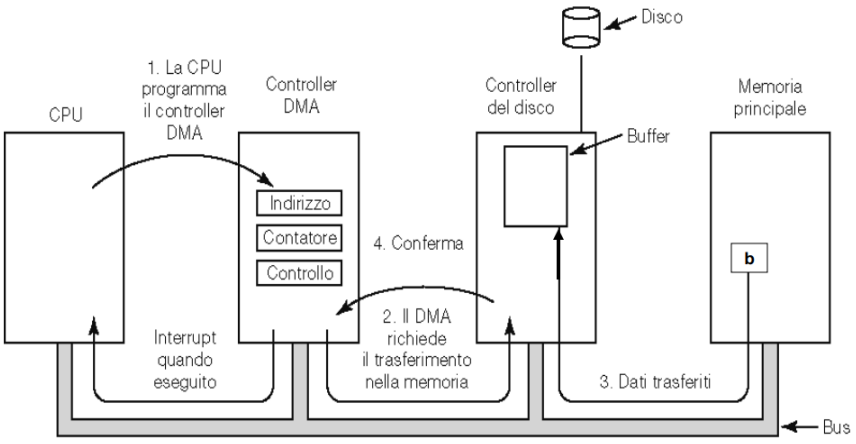
```
while(device_status_reg != READY){} // busy waiting solo all'inizio
buffer = b[0] // quando il dispositivo è libero trasferisco il primo byte
c=1,i=1
scheduler() // p lascia la CPU, la riotterrà ad operazione conclusa
```

  A questo punto il programma P ha trasferito un byte e ha perso il processore.
  Codice eseguito dall'interrupt handler:

```
if(c==n){ // se il byte trasferito è stato l'ultimo
    unblock_user() // Il programma P viene rischedulato
}else{ // altrimenti trasferisco il byte successivo
    buffer = b[i]
    i=i+1, c=c+1
}
return_from_interrupt()
```

  Anche questa soluzione ha un problema: per trasferire n byte abbiamo n interrupt, che portano con sè dell'overhead.

- Soluzione 3
  L'architettura prevede un controllore DMA capace di accedere direttamente alla memoria e di lavorare in parallelo con la CPU.



La CPU non comunica direttamente con il controllore del disco ma con il DMA.
Codice eseguito su richiesta di P:

```
set_up_DMA_controller()
scheduler()
```

In pratica interagiamo con il controllore del DMA per settarlo in modo da trasferire n byte. Successivamente chiamo lo scheduler per lavorare in parallelo con un altro processo.

Codice eseguito dall'interrupt handler che gestisce l'interrupt inviato dal DMA per comunicare la conclusione dell'operazione:

```
unblock_user()
return_from_interrupt()
```

Il ciclo con il trasferimento degli n byte viene effettuato dal controller DMA che al termine del ciclo invia un interrupt alla CPU, Il tutto mentre la CPU fa altro lavoro.

2.4 Introduzione alle system call

```
#include <fcntl.h>
main( ){
    int fd;
    char buf[20], buff[30];
    fd = open( "fileDiProva" ,O_RDONLY);
    read(fd,buf,20);
    read(fd,buff,30);
}
```

La *open* è una UNIX system call con due parametri: nome del file da aprire e tipo di apertura
La *open* restituisce un file descriptor, ossia un accesso al file

La funzione di libreria *open* ha lo stesso nome della system call, invocabile specificando come parametro il nome del file della quale si richiede la lettura e la modalità di apertura del file.
La funzione open invoca la system call open, cioè invoca la TRAP con il parametro che corrisponde alla open.
La funzione open potrebbe avere qualsiasi nome

A questo punto, una volta ottenuto il file descriptor lo utilizziamo per andare a effetuare delle operazioni su disco.
La system call read ha bisogno di tre parametri:

1. il file descriptor (dove voglio leggere)
2. dove va immessa l'informazione, in questo caso in due variabili (buf e buff)
3. quanta informazione leggere (20 e 30 byte)

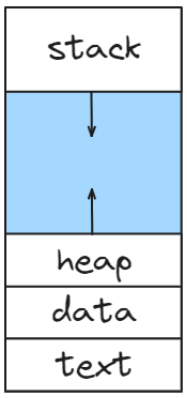
La funzione read invoca la system call read, cioè invoca la TRAP con il parametro che corrisponde alla read.

3 Processi

I computer iniziali consentivano l'esecuzione di un solo programma alla volta. Questo programma aveva il controllo completo del sistema e aveva accesso a tutte le risorse del sistema. In contrasto, i sistemi informatici contemporanei consentono il caricamento in memoria ed l'esecuzione simultanea di più programmi. Questa evoluzione ha richiesto un controllo più rigoroso e una maggiore compartimentazione dei vari programmi; e queste esigenze hanno portato all'idea di un processo, che **è un programma in esecuzione**. Un processo è l'unità di lavoro in un sistema informatico moderno.

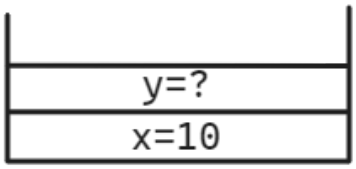
3.1 Il memory layout dei programmi

L'esecuzione di un programma necessita di tre aree di memoria (regioni)



- **Area di testo**: contiene il codice eseguibile del programma. Non è modificabile da esso.
- **Area dati**: contiene le variabili globali del programma. Questa area ha dimensione e contenuto variabile (anche se minore dell'area di stack).
- **Area di stack**: contiene i record di attivazione (frame) delle procedure già chiamate e non ancora terminate. Anche in questo caso questa area ha dimensione e contenuto variabile.
  Per le strutture dinamiche delle procedure si usa una parte dell'area dai detta **heap**, cui le variabili fanno riferimento

```c
#include <stdio.h>
int a=5; /*variabile globale - area dati*/
int b=6; /*variabile globale - area dati*/
main( ){
    int x = 10;
    /*(punto 1)*/
    int y = f1(x,b);
    /*(punto 5)*/
    printf("ecco il valore di y: %d\n",y);
}
int f1(int s, int t){
    int u = a+s+t;
    /*(punto 2)*/
    int v = f2(u);
    /*(punto 4)*/
    return v;
}
int f2 (int h){
    int k = h+15;
    /*(punto 3)*/
    return k;
}
```



Nei record di attivazione sono presenti le variabili che appartengono alla funzione cui è riferito quel record di attivazione, In questo caso x e y. La x in questo punto ha già un valore, la y non ha ancora un valore

Figure 1: Record di attivazione al punto 1

Successivamente viene eseguita l'istruzione

$$int\ y\ =\ f1(x,b)$$

Viene chiamata la funzione f1 e il risultato verrà assegnato a y.
Nel momento in cui si chiama f1 viene inserito nello stack il record di attivazione di f1.
Esso deve contenere lo spazio per tutte le variabili di *f1* (u,v), nonchè per tutti i parametri(s,t).
Ogni record di attivazione contiene un **riferimento al record di attivazione precedente**, rappresentato nelle illustrazioni successive con una freccia di colore blu. Successivamente f1 chiama f2 tramite l'istruzione seguente:

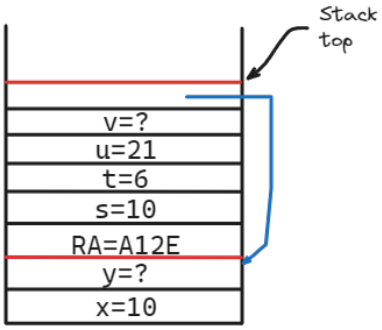


Figure 2: Record di attivazione al punto 2

Il valore di s e t sono i parametri attuali x e b.
La u è il risultato della prima istruzione della funzione, ossia la somma di a,s e t.
Quando *f1()* terminerà bisognera riprendere a eseguire il main dall'ultima istruzione eseguita. Per sapere l'indirizzo di questa istruzione basta guardare il record di attivazione di F1, dove il **program counter ha salvato l'indirizzo** prima del "salto" nell'esempio l'indirizzo è A12E. RA = return address

$$int\ v\ =\ f2(u)$$

Bisogna impilare un record di attivazione.

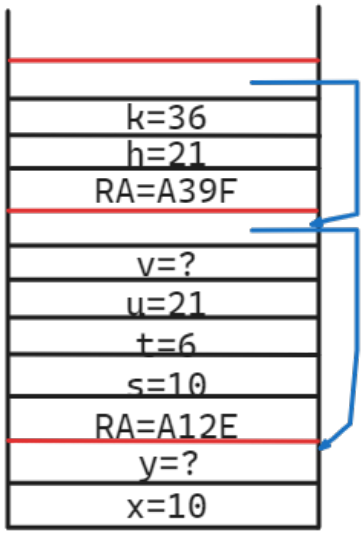
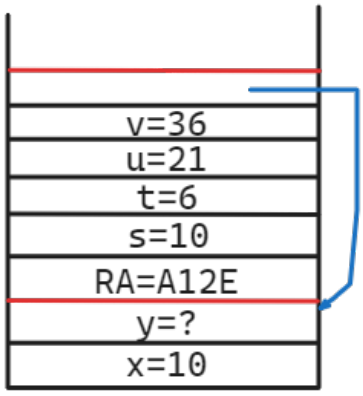


Figure 3: Record di attivazione al punto 3

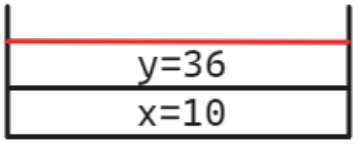
h è il parametro formale di f2, dato da u.
k è una variabile dichiarata all'interno di f2, è la somma di h + 15.
Abbiamo ancora il return address che punta all'istruzione da cui ripartire in f1.

Quando si arriva all'istruzione *return k* il record di attivazione di f2 verrà distrutto e il valore di k deve essere dato a chi lo chiedeva (v).



Il valore di ritorno di f1 viene assegnato a v
La distruzione del record di attivazione di f2 avviene ponendo il top della pila al primo indirizzo del record di attivazione di f1

Figure 4: Record di attivazione al punto 4



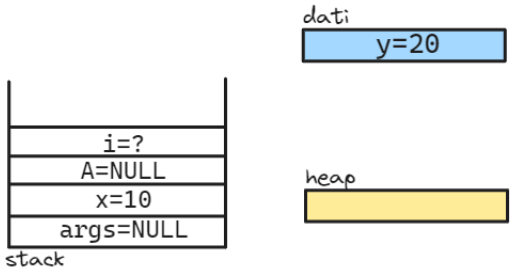
A questo punto f1 termina, il valore di ritorno della funzione viene assegnato alla variabile del main y:

Figure 5: Record di attivazione al punto 5

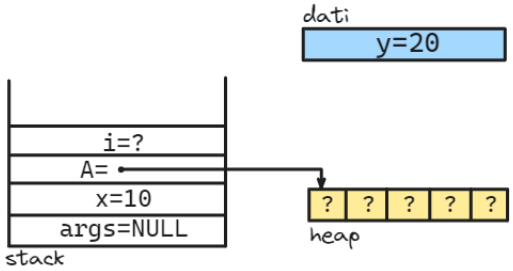
Esempio di programma in java, ricordiamo che in questo linguaggio gli array sono di fatto puntatori.

```
class Cl{
    int y = 20;
    public static void main(String[] args){
    int x = 10;
    int[ ] A;
    // (punto 1)
    A = new int[5];
    // (punto 2)
    for (int i=0,i<5,i++){
        A[i] = y+x+i;
    }
    // (punto 3)
    }
}
```

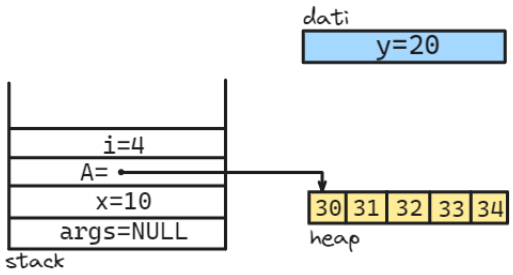
Punto 1: supponiamo che non sia specificato l'array args



Successivamente l'array viene inizializzato, nell'area di heap verrà riservata un'area di ram tale da ospitare 5 interi.



Successivamente per ogni iterazione del ciclo assegnamo il valore di y+x+i alla poisizione i del ciclo.

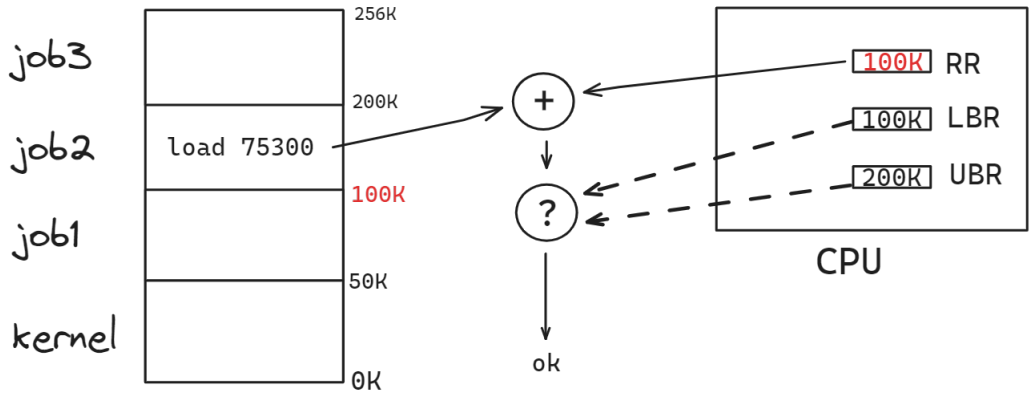


Osservazione: il codice di un programma contiene indirizzi di memoria che si riferiscono:

- All'area di testi
- All'area dati
- All'area di stack

Questi sono *indirizzi virtuali* che sono tradotti dalla MMU a tempo di esecuzione.

Esempio: supponiamo che il programma parta dall'indirizzo 0 immaginando abbia tutta la RAM a sua disposizione, anche se in memoria fisica parte dall'indirizzo 100k



Dunque usiamo un registro chiamato *relocation register*, nel quale mettiamo il valore di memoria fisica nella quale parte il programma (nell'esempio 100k) tutte le volte che viene eseguita un'istruzione con un riferiemnto alla memoria, viene corretto sommando il valore al RR. Il modulo che compie questa azione si chiama **MMU**. Dunque la MMU corregge tutti gli indirizzi sbagliati sommandoli istruzione per istruzione con l'indirizzo di memoria fisica reale.

3.2 Il concetto di processo

Processo

Un processo è un'esecuzione di un programma (M.J. Bach).
Questa definizione prevede che possa lanciare più volte un programma non necessariamente attendendo la terminazione del programma stesso.

Processo

Un processo è un'istanza di un programma in esecuzione (D.M. Dhamdhere, A.S Tanebaum).

Possiamo avere **più processi relativi al medesimo programma**, in quanto possiamo avere più esecuzioni, anche in contemporanea, del programma.

- Il programma è un'entità passiva che non esegue nessuna azione di per sè
- L'esecuzione del programma, chiamata **processo**, concretizza le azioni specificate nel programma
- Il S.O. schedula processi, non programmi

Diciamo inoltre che un processo consiste delle seguenti componenti:

- Stato della CPU: caratterizza lo stato di avanzamento del nostro processo (PC)
- Area di testo
- Area di dati
- Area di stack
- Risorse logiche e fisiche assegnate al processo

3.2.1 File e processi

```
    #include <stdio.h>
int main( ){
    char stringa[20];
    FILE *str1, *str2;
    str1 = fopen("prova.txt","wt");
    for(int i=0; i<5; i++){
        printf("dammi una parola: ");
        gets(stringa);
        fwrite(stringa, sizeof(stringa), 1, str1);
    }
    fclose(str1);
```

```
    str2 = fopen("prova.txt","r");
    for(int i=0; i<5; i++){
        fread(&stringa, sizeof(stringa), 1, str2);
        printf("parola numero %d: %s\n", i, stringa);
    }
    fclose(str2);
}
```

I processi relativi a questo programma avranno i propri accessi al file prova.txt (str1, accesso in lettura/scrittura, e str2, accesso in sola lettura), il proprio accesso alla tastiera per ottenere dati ed il proprio accesso al video per mostrare messaggi. Gli accessi a video/tastiera sono aperti di default

In UNIX i dispositivi vengono visti come file, quindi anche per i dispositivi abbiamo i file descriptor. Pertanto se il programma gira su UNIX abbiamo i file descriptor anche per la tastiera ed il video. In verità ogni processo dispone di default dei file descriptor per la tastiera, detto standard input, e per il video, detto standard output

3.3 Parallelismo e concorrenza

Parallelismo

Due eventi sono **paralleli** se occorrono nello stesso momento

Concorrenza

La **concorrenza** è l'**illusione** del parallelismo. Due attività sono concorrenti se si ha l'illusione che vengano eseguite in parallelo, mentre, in realtà, in ogni singolo istante solo una di esse viene eseguita.

Assumiamo che l'hardware offra una sola CPU e un DMA controller: Assegnando la CPU a turno ai vari processi il S.O. realiza l'**esecuzione concorrente** di tali processi, in quanto in ogni istante al più un solo processo può usare la CPU.
La velocità di avanzamento di un processo non è uniforme nel tempo, in quanto in alcuni momenti dispone della CPU e in altri no.

E se avessimo più di una CPU distingueremmo due casi:

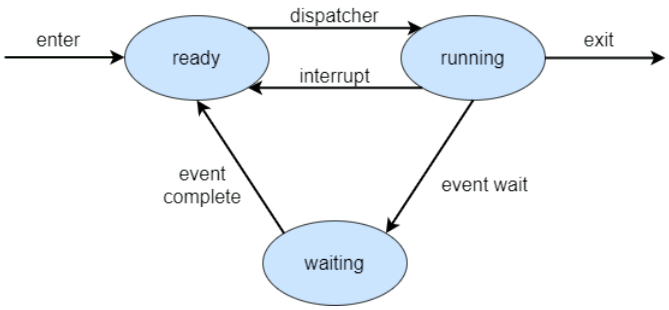
- **Multiprocessing**: più processi possono eseguire su una macchina dotata di più CPU
- **Distributive processing**: più processi possono eseguire su più macchine distribuite e indipendenti.

In questi casi il parallelismo può essere reale.

3.4 Stati di un processo

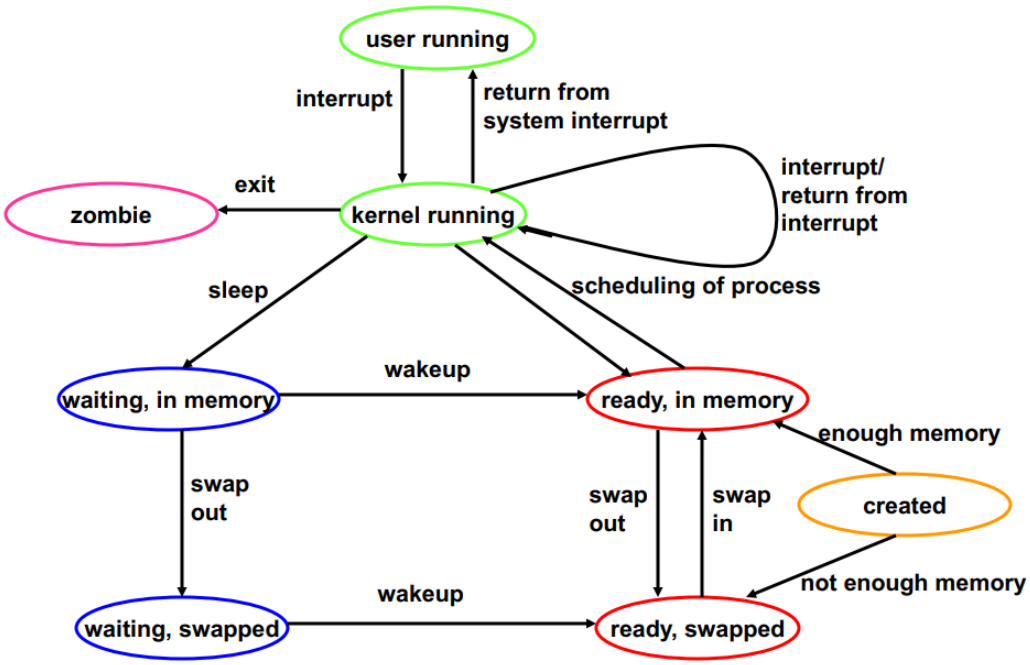
In ogni istante un processo si trova in un certo stato:

- **Running**: il processo è in esecuzione, la CPU sta eseguendo il programma associato al processo. In un sistema uniporocessor solo un processo è e*running*.
- **Ready**: il processo non è in esecuzione, ma se ottenesse la CPU potrebbe eseguire. Si trova in questo stato ad esempio un processo che ha appena finito un'operazione di I/O
- **Waiting**: il processo non è in esecuzione e non sarebbe in grado di eseguire se ottenesse la CPU. Esso acquisirà la capacità di eseguire quando si verificherà un evento che il processo sta attendendo. Esempio: processo che sta aspettando che un'operazione di I/O finisca.



- Caso **ready - running**: lo scheduler seleziona il processo per l'esecuzione tramite una politica di scheduling.
- Caso **running - ready**: Il processo perde il processore, subisce una **preemption** perchè un processo con priorità maggiore diventa ready, in un sistema con timesharing può anche verificarsi la scadenza del timeslice.
- Caso **running - waiting**: Il processo viene messo in stato di waiting poichè si trova impossibilitato ad eseguire in attesa di un evento sbloccante. Ad esempio un avvio di un'operazione I/O dove l'evento atteso è il completamento dell' I/O.
- Caso **waiting - ready**: Si verifica l'evento sbloccante che il processo attendeva.

3.4.1 Stati di un processo in UNIX system V



Lo stato *running* è diviso in due parti:

- User running
- Kernel running

Lo stato *ready* e *waiting* sono stati divisi in due parti a loro volta:

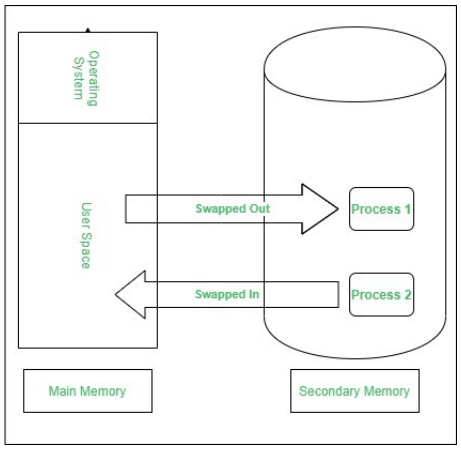
- In memory
- Swapped (non in RAM)

Successivamente abbiamo lo stato *zombie*(quando un processo è terminato e viene solamente tenuto traccia della sua esistenza) e *created*.

Alcuni precisazioni:

- Il S.O. può gestire un numero di processi che richiedano più memoria di quella disponibile: alcuni processi hanno le aree testo, dati e stack interamente in memoria, altri processi hanno le aree testo, dati e stack interamente sull'hard disk in una porzione chiamata swap device. (questa

metodologia è obsoleta)
Si parla di *swap in* quando il processo va verso la RAM e *swap out* quando va verso l'hard disk.



Lo *swap in* viene effettuato a intervalli regolari, portando processi *ready* in RAM.

- **Created - ready swapped/in memory**: Se un processo appena creato richiede una quantità di RAM disponibile va in RAM pronto per essere schedulato, se richiede più RAM di quella libera viene messa nello swap device dal S.O. Prima o poi verrà schedulato.
- **User running - kernel running**: Se un processo è user running e arriva una intterrupt(ad esempio una TRAP) esso diventa kernel running. Quando l'interrupt è stato gestito completamente dal S.O e lo scheduler schedula proprio questo processo, esso diventerà di nuovo user running.

  Ci si può trovare in stato di kernel running dopo una interrupt in caso finisca l'interrupt handler di priorità maggiore rispetto a quello che stava venendo eseguito.
- **Zombie**: Quando un processo ha finito la sua esecuzione. Ha comunque un posto nella tabella dei processi.

3.5 Process control block

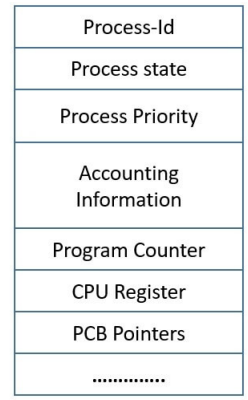
Il S.O. deve ovviamente tener traccia di tutti i processi presenti nel sistema. Per esempio, ciò è necessario in fase di scheduling, perché il S.O. deve sapere quali processi chiedono la CPU.
Il S.O. mantiene una process table, con una voce, Process Control Block (PCB), per ogni processo.

Voci presenti in ogni PCB:

- **Identificativo del processo**: (Process IDentifier - **PID**) lo identifica univocamente.
- **Stato del processo**: (running, waiting, ready, ...) bit necessari per identificare lo stato.
- **Stato della CPU**: zona che tiene traccia dei *registri* quando il processo non è running, verranno utilizzati quando il processo riprenderà la sua esecuzione.
- **Priorità** ed altri parametri da usare per lo scheduling
- Puntatore ad una zona di memoria che contiene le informazioni necessarie per accedere all' **area di testo**
- Puntatore ad una zona della memoria che contiene le informazioni per accedere all' **area dati**
- Puntatore ad una zona della memoria che contiene le informazioni per accedere all' **area di stack**
- Puntatori a zone della memoria che contengono le informazioni per accedere ai file aperti (*). Questi puntatori si chiamano **file descriptor**.

  ```
  fd = open( "fileDiProva" ,O_RDONLY);
  ```

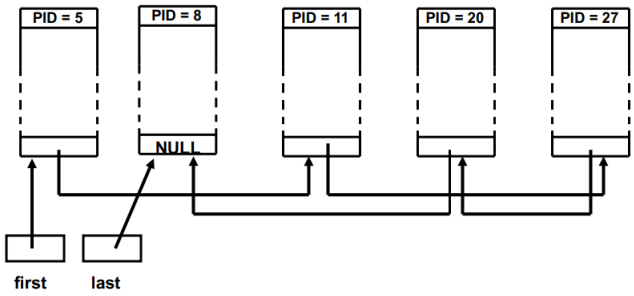
- Puntatori a zone della memoria che contengono le informazioni per accedere ai **dispositivi aperti**
- Puntatori ad altri PCB, in modo da creare strutture dati dinamiche composte da PCB per realizzare operazioni come inserimento e cancellazione.



Process Control Block

3.5.1 Strutture dati e PCB

Esempio di lista di PCB realizzata con puntatori:



Per le politiche di gestione spiegate successivamente questa potrebbe essere una coda utilizzata per lo scheduling di processi in base al tempo di attesa.

Chi è in stato di ready da più tempo sarà colui che verrà schedulato
Quando un processo diventa ready viene inserito in coda, abbiamo quindi complessità temporale $O(n)$per inserimento e estrazione di processi.

- Per acedere al processo più "anziano" uso il puntatore **first**, in questo modo estraggo dalla coda
- Per accedere al processo più "giovane" utilizzo il puntatore **last**, in questo modo inserisco nella coda

3.5.2 Introduzione alla gestione della memoria

La RAM è divisa in porzioni equidimensionali chiamati **page frame**. Assumiamo di avere una memoria da 2^{32} byte divisa in 2^{22} pagine(page frame) da 2^{10} byte ciascuna.

Ogni indirizzo assume il seguente significato:

- I primi 22 a sinistra individuano il numero della pagina.
- Gli ultimi 10 individuano la distanza rispetto all'inizio della pagina.

Le aree di testo, dati e stack vengono divise in pagine da 2^{10} byte, l'idea è quella che le partizioni possono essere ripartite nella RAM casualmente.
La MMU dovrà svolgerà il compito di tradurre numeri di pagina in numeri di frame.

3.6 Implementazione dei processi: il context switch

Assegnare la CPU a un altro processo richiede di eseguire un salvataggio dello stato del processo attuale e un ripristino dello stato di un diverso processo. Questa operazione è conosciuta come "context switch".

3.6.1 Contesto di un processo

Contesto di un processo

Il contesto di un processo, detto anche ambiente è costituito da:

- *Register context*: il contenuto dei registri della CPU (accessibile al processo stesso)
- *User-level context*: zona di RAM che il processo riesce a manipolare. (aree testo, dati e stack) variabili, procedure, ecc.
- *System-level context*: (Inaccessibile dal programma user)
  - PCB
  - Informazioni per acedere a memoria, file e dispositivi cui punta il PCB
  - Kernel stack: contiene i frame degli interrupt handler e delle procedure chiamate dagli interrupt handler

Il sistema operativo effettua 3 operazioni fondamentali sui processi:

- **Context save**: Quando il processo running va in ready o waiting, il S.O. salva nel PCB tutti i dati necessari a far ripartire il processo in futuro.
- **Scheduling**: Il S.O sceglie un processo tra quelli ready per l'esecuzione.
- **Dispatching**: Dopo che un processo viene schedulato, il S.O. deve ripristinarne il contesto sfruttando i dati salvati nel PCB al momento del context save

Context switch significa *context save* per un processo e *dispatching* per un altro processo. Notare che mentre il S.O effettua il context save di P lo scheduling e il dispatching di P', il processo P risulta essere in running.

E' evidente che il context switch causi **overhead**, specie se l'algoritmo di scheduling è complicato.

Inoltre, ogni volta che effettuiamo un context switch la cache non viene usata, risultano cache miss.

Esempio 1: La CPU ha i control register PC, SP, PSW ed i registri generali R1, R2. La CPU è a 16 bit. Assumiamo inoltre che la SRIA sia nel kernel stack. Assumiamo che la SRIA sia nel kernel stack. Il processo 5 è running, il processo 9 è ready. Il processo 5 esegue una TRAP, che verrà gestita dall'apposito interrupt handler. Al ritorno dall'interrupt handler, il processo 5 riprenderà ad eseguire.

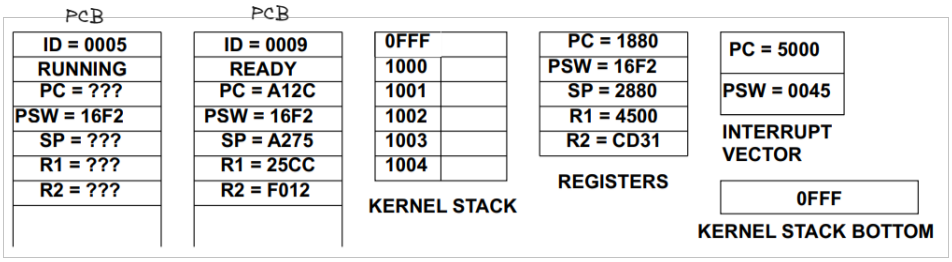


Figure 6: Situazione iniziale, prima della TRAP – Il bit PM della PSW è 0,

Figure 7: Esempio di PSW

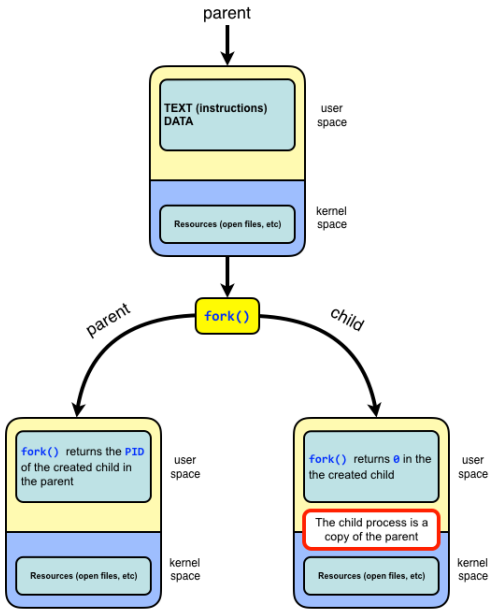
- In questo momento sta eseguendo il processo 5, nei registri sono salvati i dati di quel processo.

...

3.7 Creazione e terminazione di processi

Il S.O. mette a disposizione un 'apposita system call per la creazione di processi. Ad esempio la *fork()* in UNIX/Linux.

In alcuni S.O., come UNIX/Linux, tra il processo che esegue la sysem call, detto processo padre, ed il processo creato, detto processo figlio, si crea una relazione gerarchica. In questo caso, un processo può avere più figli ma un solo padre. I figli possono avere, a loro volta, figli.



3.7.1 Fork

La fork restituisce 0 al figlio ed il PID del figlio al padre.

- Dopo la fork, il padre ed il figlio sono due processi distinti ma quasi uguali:
  - entrambi hanno il proprio PCB (quello del figlio è creato dalla fork), i PID sono diversi, la getpid da risultati diversi
  - i registri generali hanno il medesimo valore, eccetto il registro usato per il risultato della system call
  - i registri **PC**, **PSW**, **SP** hanno i medesimi valori

Testare il risultato della *getpid* è il modo classico per far eseguire a padre e figlio porzioni di programma diverse.

3.7.2 Exec

La system call **exec** serve per "cambiarsi il programma ".

- La *execl* è una delle funzioni di libreria involucro della exec.
- Le altre funzioni di libreria involucro della exec (execle, execlp, execv, execve, execvp, fexecve) usano parametri diversi in numero e/o tipo
- Il primo parametro è il path dell'eseguibile che si chiede di eseguire, il secondo è il nome dell'eseguibile e il terzo sono gli argomenti passati al programma eseguibile

Tramite la exec:

- Il processo ottiene nuove aree testo, dati e stack
- I registri PSW, PC, SP assumono valori nuovi
- i registri generali vengono "azzerati"
- i file vengono chiusi ed i dispositivi rilasciati

3.7.3 Wait

La system call wait è una richiesta al S.O. di mettersi in stato di WAITING. L'evento atteso è la terminazione di un processo figlio.

- La wait ha un parametro: un pointer ad una variabile dove viene memorizzato il valore che codifica la causa della terminazione del figlio. Il valore 0 significa che la terminazione è regolare
- Quando un processo termina, esegue la system call exit e va in stato ZOMBIE. Il compiler C inserisce la *exit(0)* al termine del programma
- Quando un processo esegue la exit, l'interrupt handler controlla se la terminazione è attesa dal padre. Se è cosi il padre passa da WAITING a READY

3.8 Tipi di processo

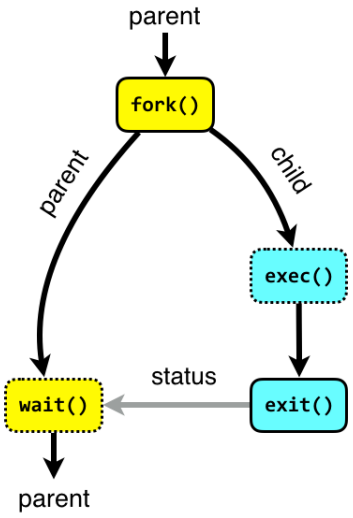
Non tutti i processi sono uguali. Esempio, in UNIX abbiamo processi utente, processi deamon e processi kernel.

3.8.1 Processi utente

I processi utente eseguono un calcolo richiesto dall'utente, dunque sono associati al **terminale dell'user**. (avranno il bit PM settato a 0 per la user mode)

Concetto che nasce negli anni 80, Dove avevamo un processore costoso che doveva essere diviso tra diversi utenti. Di seguito mostriamo due esempi di esecuzioni di un processo utente dopo aver chiesto alla shell di mandare in esecuzione un programma chiamato *a.out*:

- Esecuzione classica *a.out*:
  La shell fa una fork. Il figlio fa la exec su a.out. Il padre(shell) resterà in attesa della terminazione.



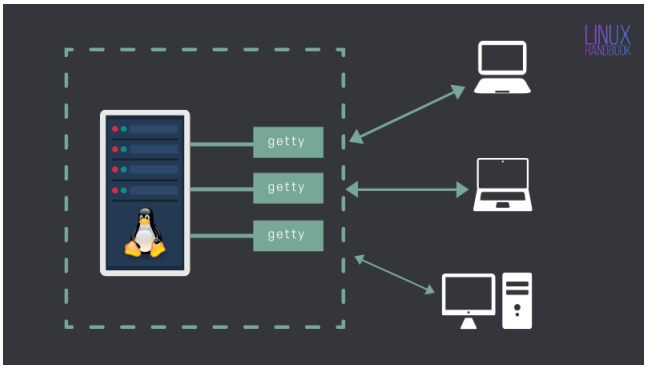
- Esecuzione "in background" *a.out* &: La shell fa una fork. Il figlio fa la exec su a.out. Il padre(shell) si mette in attesa di un altro comando da parte dell'utente.

3.8.2 Processi daemon

I processi daemon sono processi non associati a nessun utente che eseguono **funzioni vitali** per il sistema (controllo email, gestione coda di stampa, controllo richieste web page, ecc.) e dunque tipicamente non terminano.
Eseguono in modalità user.

Esempio: il processo **getty**

- Gestisce il processo di login di un utente al terminale
- Se il login va a buon fine, esegue una *fork*, il figlio fa la *exec* del programma di shell, il padre(getty) si mette in attesa che il figlio termini.



3.8.3 Processi kernel

I processi kernel sono **daemon** che eseguono in **modalità kernel**.

- Accedono a strutture del kernel senza dover invocare le system call

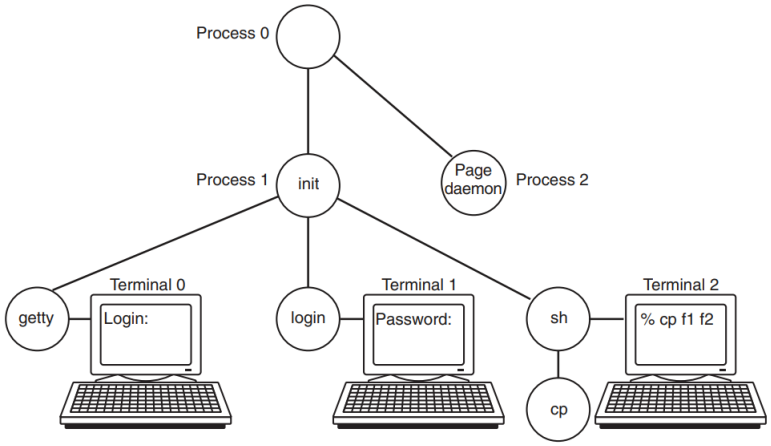
Esempi:

- Nei sistemi con paginazione della memoria, lo **stealer process** viene svegliato ad intervalli regolari e fa lo **swap out** delle pagine che recentemente non hanno avuto accessi.
- Nei sistemi con swapping, lo **swapper process** viene svegliato ad intervalli regolari e fa lo swap out dei processi con maggior anzianità in memoria, e lo **swap in** dei processi da più tempo in READY SWAPPED.

3.8.4 Processi e booting

Al momento del boot è necessario che venga eseguito del codice di inizializzazione che "costruisce" almeno un processo, dal quale possono discendere tutti gli altri.
In alcuni sistemi linux:

- Al momento del boot viene creato il processo 0, chiamato **swapper**, che è chiamato swapper ed esegue in modalità **kernel**.
- Lo swapper crea il processo 1, chiamato **init** (tramite procedura *fork()* o istruzioni dirette, eseguendo in modalità kernel), il quale si crea un user context. Si cambia modalità in user (modificando il bit della PSW) e diventa un normale *daemon*.
- lo swapper crea altri processi kernel (esempio lo stealer nei sistemi con paginazione) e, nei sistemi con swapping, fa quanto detto nella slide precedente;
- Il processo **init** crea altri daemon, per esempio un getty per ogni terminale (quindi un solo getty nei PC) e si mette in waiting attendendo che terminino.



3.8.5 Terminazione di processi

Esistono alcune circostanze che fanno si che un processo termini:

- Tramite system call *exit* in UNIX e *ExitProcess* in APIWin32
- L'*interrupt handler* delle eccezioni può **forzare** la terminazione del processo. Accade, ad esempio, in caso di errori di overflow
- Mediante un altro processo killer. Esempi: **kill** in UNIX

Quando un processo termina:

- Rilascia tutte le risorse, dunque non è più schedulabile
- La distruzione del PCB non è sempre immediata. Ad esempio in UNIX il PCB di un processo ucciso è cancellato dal padre. Prima che ciò avvenga il PCB è in stato ZOMBIE.
- Se un processo termina ed ha figli essi non vengono uccisi, ma diventano figli del processo init.

3.9 System call in UNIX

Oltre al PCB, UNIX assegna ad ogni processo una user area, detta anche *u area*.
Il PCB ha un pointer alla *u area*.
Le informazioni riguardanti il processo sono inserite:

- Nella *u area* se servono solo quando il processo è in stato di running. (es: file descriptor, pointer a info area testo, dati e stack)
- Nel *PCB* se servono sempre. (es: stato de; processo, pointer alla u area, parametri di scheduling)

La *u area* serve anche per memorizzare i parametri, il risultato e l'eventuale codice di errore delle **system call**

3.9.1 Esempio di system call a basso livello

Supponiamo che il **main** di un programma in **C** esegua la chiamata seguente:

```
int fd = create("file_di_prova", O_RDWT)
```

La variabile **fd** assume un valore $n \geq 0$ se la chiamata va a buon fine, -1 altrimenti.

Questa istruzione è una normale chiamata di funzione, riportiamo di seguito il corrispettivo in linguaggio macchina Motorola 68000s:

```
# codice del main

58: mov &0x1b6, %sp
5e: mov &0x204, -%sp
64: jsr 0x7a
```

- **mov** e **jsr** sono i codici della *move* e *jump to subroutine*
- **1b6** rappresenta il codice di **O_RDWT**(read and write)
- **%sp** denota lo **stack pointer**
- Dunque nella prima istruzione spostiamo 1b6 nello stack
- Nella seconda istruzione decrementiamo lo stack pointer (in UNIX lo stack cresce verso il basso) e spostiamo la variabile contenente il nome del file nello stack.
- L'ultima istruzione permette di fare una jump all'indirizzo 0x74, ossia dove è presente il codice della *creat*.

```
# codice di libreria della funzione creat

7a: movq &0x8,%d0
7e: trap &0x0
# esecuzione dell'interrupt handler
```

```
# ritorno alla funzione chiamante
7e: bcc &0x6 <86>
80: jmp 0x13c
86: rts
```

Supponiamo che questa architettura faccia si che l'operando della TRAP sia il registro che contiene il parametro. (il parametro potrebbe essere in qualsiasi registro, scelgo che sia nel registro 0)

- Dunque inserisco il parametro della TRAP (8) nel registro 0. Il valore del parametro serve per determinare che la system call è la *creat*

- Succesivamente eseguo la TRAP passandogli come operando il registro che contiene il parametro (registro 0). A questo punto eseguendo un interrupt (TRAP) avviene il salvataggio dei registri nella SRIA
- L'interrupt handler della *creat* salva il valore dei registri generali, esegue le proprie funzioni:
  - **Copia nella u area** del processo i parametri che il main aveva posto sullo stack. In questo modo il sistema operativo può lavorare utilizzando solo la sua zona di RAM (senza che il processo sia schedulato).
  - L'interrupt handler lavora sul disco **memorizzando il risultato della system call e il codice di errore** nella u area.

  Successivamente chiama lo scheduler.

  In caso di errore prima di chiamare lo scheduler:
  - viene settato a 1 il Carry Bit (è un bit del Condition Code) della PSW viene inserito il codice di errore nel data register 0
  - Se non c'è errore, prima di chiamare lo scheduler il Carry Bit della PSW rimane a 0 e il risultato viene posto nel data register 0.
- Quando il processo riprende il controllo, riparte la funzione di libreria dall'istruzione ad indirizzo 7e, dove è presente una istruzione di branch condizionato:
  - Carry bit **0**: la *creat* termina tramite la **rts** e il main esegue dall'istruzione successiva alla *jsr*. Il valore del data register 0 è da assegnare alla variabile fd
  - Carry bit **1**: la *creat* chiama la subroutine che gestisce l' errore (indirizzo *13c*).

```
# Routine di gestione dell'errore

13c: mov %d0, &0x20e
142: mov &0x-1, %d0
144: rts
```

- Il codice dell' errore viene inserito all'indirizzo **20e**, vale a dire l'indirizzo della variabile **errno**.
- il risultato -1 (errore) viene posto nel data register 0
- Il main riprende ad eseguire, Il valore del data register 0 (cioè -1) è da assegnare alla variabile fd. main può analizzare l' errore, trovandolo all'indirizzo 20e (variabile error number)

4 Threads

4.1 Introduzione ai threads

Definition 4.1 (Task)**.** Un **task** è uno dei compiti che un'applicazione deve portare avanti in modo simultaneo.

Se il linguaggio offre i costrutti la programmazione **concorrente**, in modo da definire flussi di esecuzione sequenziale indipendenti per i vari task e eseguire i flussi concorrentemente, si può guadagnare in termini di:

- **Semplicità** di programmazione: è piu facile suddividere il codice in task, eseguendoli poi concorrentemente.
- **Efficienza**: facendo eseguire task in parallelo avremo
  - parallelismo tra lavoro di CPU di un task e lavoro di I/O di un altro task;
  - parallelismo tra lavoro di CPU di due task, possibile solo se esistono più processori.

Definition 4.2. Un thread è un'esecuzione di un programma che usa le risorse di un processo

Un processo può avere più thread. Possibili implementazioni

1. Un processo ha più thread e questi condividono le aree di testo e dati e le risorse logiche e fisiche sono condivise (La comunicazione tra thread è semplice)
2. Ogni thread ha il proprio identificatore, CPU state, stack e stato. (un processo con n *program counter*)

Per ogni thread è necessario avere un thread control block con identificatore dell thread, CPU state, stack e stato, i TCB vengono organizzati in una thread table.

Il **thread switching** tra due thread dello stesso processo introduce meno overhead rispetto al classico context switch, a vantaggio dell'efficienza, infatti:

- Non vanno aggiornati i dati della MMU relativi alle aree testo e dati
- la condivisione di testo e dati fa sì che sia possibile che la cache della memoria richieda meno aggiornamenti;
- la condivisione dei file fa sì che sia possibile che la cache del disco richieda meno aggiornamenti.

Inoltre la creazione di un thread è più veloce della creazione di un processo, perché il TCB ha meno dati del PCB e perché non va allocata memoria per testo e dati.

4.1.1 Esempio di thread

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void *PrintHello(void *threadid){ /* "programma" per i thread */
   long tid = (long)threadid;
   printf("Ciao, sono la thread numero #%ld!\n", tid);
   pthread_exit(NULL);
}

int main(  ){
   pthread_t th[5];
   long t;
   for(t=0; t<5; t++) {/* creo 5 thread, ognuno esegue la funzione PrintHello*/
      printf("Creo la thread numero %ld\n", t);
      pthread_create(&th[t], NULL, PrintHello, (void *)t);
   }
   pthread_exit(NULL);
}
```

- Per prima cosa dichiaro un array di 5 elementi di tipo *pthread_t*
  - *pthread_t* è un tipo di dato presente nella libreria <*pthread.h*>, esso specifica un *thread identifier*.
- Successivamente all'interno del ciclo eseguo l'istruzione *pthread_create()*
  - *pthread_create()* serve per creare i thread e ha bisogno di 4 parametri:
    1. memory address ove memorizzare il TID del thread creato
    2. attributi (con NULL si forza l'uso degli attributi di default)
    3. funzione di tipo *void che deve essere eseguita dal thread creato
    4. parametro di tipo *void da passare a tale funzione
- All'interno della funzione eseguita da ogni thread (*PrintHello*) come prima cosa salvo nella variabile *tid* l'indice relativo al ciclo del thread che sta eseguendo la funzione
- Successivamente tramite l'istruzione *pthread_exit(NULL)* faccio terminare il thread liberando lo stack di esso, il thread control block del thread arrivato a questo punto avrà un CPU state paragonabile a quello di zombie.

```c
#include <pthread.h> /* Piccola modifica: definiamo una variabile x */
#include <stdio.h> /* che viene condivisa tra i vari thread */
#include <stdlib.h>

int x; /* il main e tutti i thread condividono x */

void *PrintHello(void *threadid){
    long tid = (long)threadid;
    printf("Ciao, sono la thread numero #%ld!\n", tid);
    printf("x vale %d\n", x);
    x=x+1;
    pthread_exit(NULL);
}

int main( ){
    pthread_t th[5];
    long t;
    x=10;
    for(t=0; t<5; t++){
        printf("Creo la thread numero %ld\n", t);
        pthread_create(&th[t], NULL, PrintHello, (void *)t);
    }
}
```

- Osserviamo come la variabile globale x è comune a tutti thread
- É possibile che i thread eseguano le prime due printf di *printHello* e perdano il processore, dunque i valori di x potrebbero seguire un ordine non ordinario.

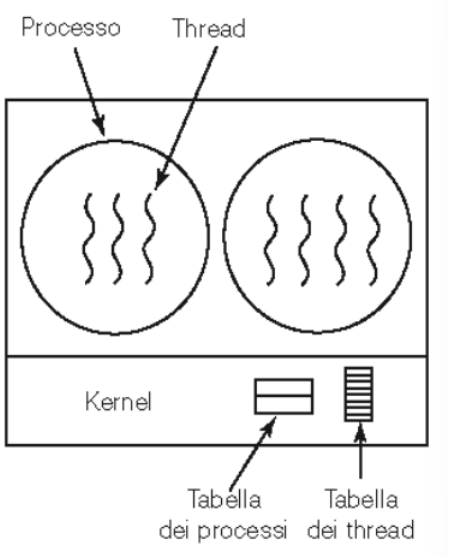
Osservazioni: la *pthread_create* restituisce:

- 0 se è tutto ok
- Un codice di errore altrimenti

Sarebbe opportuno controllarne il risultato con un if.

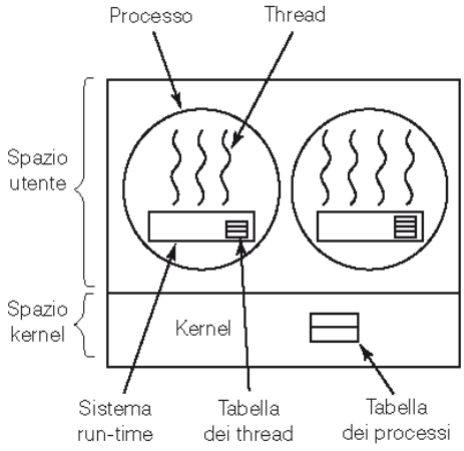
4.1.2 Threads nello spazio user e kernel

- Kernel space threads



  - Il sistema operativo gestisce i thread
  - i TCB sono strutture dati gestite dal S.O.
  - il S.O. offre system call per creare/terminare thread
  - lo scheduler del S.O. deve schedulare thread anziché processi
  - il programmatore usa i thread chiamando le funzioni di libreria wrapper di system call

- User space thread



- Il S.O non sa nulla dei thread, gestisce solo processi, mentre i thread sono gestiti da una libreria di procedure (sistema runtime) che eseguono in modalità user
- I TCB sono strutture dati gestite dalla libreria di procedure
- esistono procedure per creare/terminare thread, per far cambiare loro il programma, ecc. Esiste anche una procedura (*pthread_yeld* in POSIX) per invocare lo scheduler al fine di "cedere" il controllo agli altri thread
- lo scheduler del S.O. schedula processi, una delle procedure della libreria è lo scheduler delle thread;
- il programmatore maneggia i thread usando le procedure della libreria

Il vantaggi degli user space threads sono:

- Funzionamento anche su S.O che non implementano threads, basta la libreria
- il thread switching non richiede l'intervento del S.O., azzera il minimo overhead dei kernel space threads.

Svantaggi:

- in CPU multiprocessor o multithreading il parallelismo tra thread è possibile solo con kernel thread,
- se un thread fa una system call che manda un thread in waiting, quale un I/O su disco, in realtà per il S.O. va in waiting il processo, quindi si bloccano tutti i thread.

4.1.3 Esempio threads in Java

Java, anzichè offrire le chiamate alla funzioni che gestiscono thread, li supporta direttamente.

```
class ThreadUno extends Thread{
    private String threadName;

    public ThreadUno(String name){
        threadName = name;
    }

    public void run( ) {
        for( ; ; ) {
            System.out.println(threadName+" "+MainThread.x);
        }
    }
}
```

```
class MainThread {
    protected static int x = 123;

    public static void main(String[ ] args) {
        ThreadUno t = new ThreadUno("EsempioThread");
        t.run( );
    }
}
```

5 Sincronizzazione dei processi

Su un sistema uniprocessore, programmare un'applicazione con un insieme di processi e thread concorrenti può portare i seguenti vantaggi:

- **semplicità**: di programmazione, nel caso in cui l'applicazione preveda più compiti
- **efficienza**: se si ha parallelismo tra attività I/O ed attività di CPU
- Possibilità di assegnare compiti più urgenti a processi o thread con priorità più elevata.

Disclaimer: nella discussione che segue assumiamo che più processi abbiano una zona di memoria condivisa. Abbiamo detto che i processi hanno memoria separata ma:

- Il sistema operativo consente ai processi di accedere a strutture dati del kernel, che risultano condivise dai processi.

Esempio di memoria condivisa: un sistema banale di prenotazioni aeree usa n terminali connessi ad un computer centrale. Abbiamo un processo per ogni terminale. I processi condividono due variabili:

- *next_seat*: indica il prossimo posto da assegnare
- *max*: indica il numero totale di posti sul volo

Tutti gli n processi eseguono lo stesso programma per prenotare un posto su un volo:

```
if (next_seat <= max) {
     booked = next_seat;
     next_seat++;
 }
else{
     printf("sorry, flight sold out")
 }
```

Dato che l'if verrà eseguito da più processi che condivisono la variabile *next_seat* possiamo indicare 4 tipi di interazione di processi:

1. **data sharing**: i processi condividono i dati della memoria condivisa e alcuni file (caso *next_seat*)
2. **control synchronization**: un'azione a_i di un processo P_i è abilitata solo dopo che un altro processo P_j ha svolto un azione a_j .
   Ho i flussi di esecuzione dei due processi, questa azione richiede che l'altro processo abbia gia fatto qualcos'altro. Questo perche sono dipendenti tra loro.
3. **message passing**: un processo P_i invia un messaggio ad un processo P_j che lo riceve.
4. **signals**: un processo P_i invia un segnale (un messaggio privo di contenuto) ad un altro processo P_j per segnalare una situazione particolare.
   Ad esempio con la syscall kill un processo invia un segnale di terminazione ad un altro processo.

Dato un processo (o thread) P_i siano:

- R_i: i dati letti dal processo P_i (variabili/file cui si accede in lettura, messaggi/segnali ricevuti da altri processi, ...). L'insieme R_i è detto *read_set* di P_i
- W_i: i dati modificati dal processo P_i (variabili/file cui si accede in scrittura, messaggi/segnali ricevuti da altri processi, ...). L'insieme W_i è detto *write_set* di P_i

Processi interagenti

Due processi concorrenti P_i e P_j processi interagenti se vale almeno una delle seguenti due proprietà

- R_i e W_j hanno intersezione non vuota,
- R_j e W_i hanno intersezione non vuota

Quindi se il processo i invia informazioni al processo j, oppure il processo j invia informazioni al processo i, oppure entrambe le cose.

Nell'esempio delle prenotazioni aeree, gli n processi sono due a due interagenti, perché leggono e modificano la variabile *next_seat*. (*next_seat* è nel *read_set* e nel *write_set* di ogni processo.)

Processi indipendenti

Due processi concorrenti P_i e P_j sono processi indipendenti se non sono interagenti

Se due processi/thread concorrenti Pi e Pj sono indipendenti:

- competono per le risorse (e.g. CPU), rallentadosi a vicenda
- i loro comportamenti non dipendono dalla loro velocità relativa e **sono riproducibili**.

Se due processi/thread concorrenti Pi e Pj sono interagenti:

- competono per le risorse (e.g. CPU), rallentadosi a vicenda
- i loro comportamenti dipendono dalla loro velocità relativa e **non sono riproducibili**.

Esempio: sia x una variabile intera condivisa inizializzata a 100.

Programma di P_i

```
int y = x+5;
printf("%d", y)
```

Programma di P_j

```
x=10
```

Il comportamento di P_i non è riproducibile, eseguendo P_i e P_j più volte vale che:

- a volte P_i stampa 105
- a volte P_i stampa 15

Il risultato dipende dalle politiche di scheduling, dalle priorità di P_i , P_j e degli altri processi, da quanti e quali altri processi sono in esecuzione, ecc.

5.1 Race conditions

```c
#include <pthread.h> /* Esempio 1: variabile condivisa da 3 thread */
#include <stdio.h>
#include <stdlib.h>

int x; /* x variabile globale condivisa */

void * f1(void *threadid){
     x=100; /* modifica di x */
     pthread_exit(NULL);
}

void * f2(void *threadid){
     int y = x+5; /* lettura di x */
     printf("y vale %d\n",y);
```

```
     pthread_exit(NULL);
}

int main( ){ /* x è nel write_set */
     pthread_t t1,t2;
     x=10; /* modifica di x */
     pthread_create(&t1, NULL, f1, (void *)t1); /* x è nel write_set */
     pthread_create(&t2, NULL, f2, (void *)t2); /* x è nel read_set */
     pthread_exit(NULL);
 }
```

Abbiamo due possibili risultati:

- y vale 15
- y vale 105

Lo scheduler puo schedulare prima il thread che esegue *f1* oppure quello che esegue *f2*.

Nell'esempio precedente il valore della variabile x che viene stampato a video dipende dalla velocità di esecuzione relativa dei due thread.

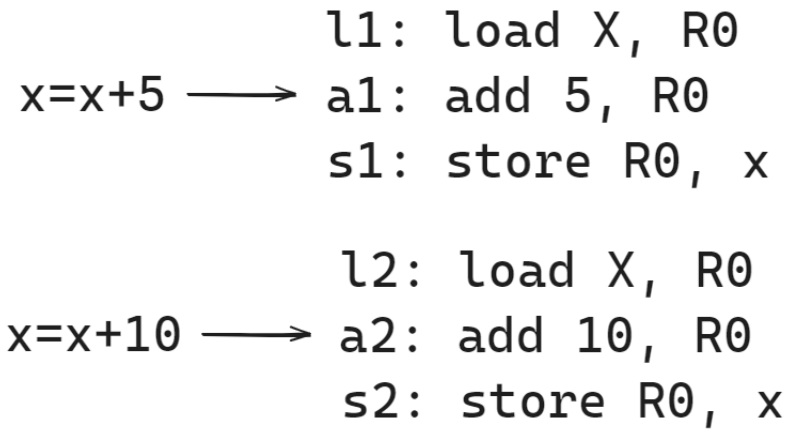
Nell'esempio seguente usiamo la funzione POSIX *pthread_join*, che serve per implementare le process synchronization. Quando un thread esegue una *pthread_join* chiedo di essere messo in waiting in attesa di un thread.

```
#include <pthread.h> /* Esempio 2: variabile condivisa da 3 thread */
#include <stdio.h>
#include <stdlib.h>
int x = 100; /* x variabile globale variabile condivisa */

void * f1(void *threadid){
     x=x+5; /* modifica di x */
     pthread_exit(NULL);
}

void * f2(void *threadid){
     x = x+10; /* modifica di x */
     pthread_exit(NULL);
}

int main( ){ /* x è nel read_set */
     pthread_t t1, t2;
     pthread_create(&t1, NULL, f1, (void *)t1); /* x è nel write_set */
     pthread_create(&t2, NULL, f2, (void *)t2); /* x è nel write_set */
     pthread_join(t1,NULL); /* attendo che termini il primo thread */
     pthread_join(t2,NULL); /* attendo che termini il secondo thread */
     printf("x vale %d\n",x); /* lettura di x */
}
```

In questo esempio ci si aspetterebbe un solo risultato:

$$x = 115$$

Ma in realtà ce ne dobbiamo aspettare anche altri. Per chiarirlo ragioniamo utilizzando le istruzioni macchina invece di quelle in C.



Può capitare che il thread che stia eseguendo x=x+5 dopo la load perda il processore dando luogo a risultati differenti:

- l1, a1, s1, l2, a2, s2 risultato 115.
- l2, a2, s2, l1, a1, s1 risultato 115.
- l1, a1, l2, a2, s2, s1 risultato 105.
- l1, l2, a1, a2, s2, s1 risultato 105.
- l2, a2, l1, a1, s1, s2 risultato 110.
- l2, l1, a2, a1, s1, s2 risultato 110.

Torniamo all'esempio delle prenotazioni aeree, e guardiamo una possibile traduzione in codice macchina

S_1	if (next_seat <= max) {	$S_{1,1}$	load max, R0
		$S_{1,2}$	sub R0, next_seat
		$S_{1,3}$	jmpneg R0, $S_{4,1}$
S_2	booked = next_seat;	$S_{2,1}$	move next_seat, booked
S_3	next_seat ++;	$S_{3,1}$	load next_seat, R1
	}	$S_{3,2}$	add R1, 1
	else{	$S_{3,3}$	store R1, next_seat
		$S_{3,4}$	jmp $S_{5,1}$
S_4	printf("sorry, sold out")	$S_{4,1}$	move "sorry", Rvideo
	}		
S_5	.....	$S_{5,1}$	.....

Siano $max = 200$, $next\_seat = 200$. è dunque rimasto un solo posto libero sul volo. I processi P1 e P2 tentano di prenotare il volo. Esaminiamo 3 casi.

	Caso 1		Caso 2		Caso 3	
Time	P_1	P_2	P_1	P_2	P_1	P_2
1	$S_{1,1}$		$S_{1,1}$		$S_{1,1}$	
2	$S_{1,2}$		$S_{1,2}$		$S_{1,2}$	
3	$S_{1,3}$		$S_{1,3}$		$S_{1,3}$	
4	$S_{2,1}$			$S_{1,1}$	$S_{2,1}$	
5	$S_{3,1}$			$S_{1,2}$	$S_{3,1}$	
6	$S_{3,2}$			$S_{1,3}$		$S_{1,2}$
7	$S_{3,3}$			$S_{2,1}$		$S_{1,2}$
8	$S_{3,4}$			$S_{3,1}$		$S_{1,3}$
9		$S_{1,1}$		$S_{3,2}$		$S_{2,1}$
10		$S_{1,2}$		$S_{3,3}$		$S_{3,1}$
11		$S_{1,3}$		$S_{3,4}$		$S_{3,2}$
12		$S_{4,1}$	$S_{2,1}$			$S_{3,3}$
13			$S_{3,1}$			$S_{3,4}$
14			$S_{3,2}$		$S_{3,2}$	
15			$S_{3,3}$		$S_{3,3}$	
16			$S_{3,4}$		$S_{3,4}$	

- **Caso 1**: tutto ok - P1 prenota, P2 no.
  P1 viene schedulato per primo, dopo aver aumentato il contatore perde il processore. P2 quando viene schedulato non entra nell'if
- **Caso 2**: non ok - P1 chiede se $nextseat \leq max$, la risposta è true dunque vorrebbe entrare nell'if, tuttavia perde il processore e P2 trovando la condizione precedente true riesce a prenotare. P1 non riuscirà a prenotare
- **Caso 3**: not ok - Il processo P1 esegue la prenotazione e prima di incrementare il contatore *next_seat* gli viene sottratto il processore. Siccome *next_seat* vale ancora 200 perchè non è stata aggiornata l'altro processo troverà l'if true ed effettuerà la prenotazione.

Dunque entrambi i processi prenotano il posto 200.

Definition 5.1. Sia d un dato condiviso dai processi P_1 e P_2.

Siano o_1, o_2 operazioni su d eseguite da P_1 e P_2 rispettivamente

Siano f_1, f_2 funzioni tali che, se **d** ha valore v_d:

- $f_1(v_d)$ è il valore assunto da **d** eseguendo o_1
- $f_2(v_d)$ è il valore assunto da **d** eseguendo o_2

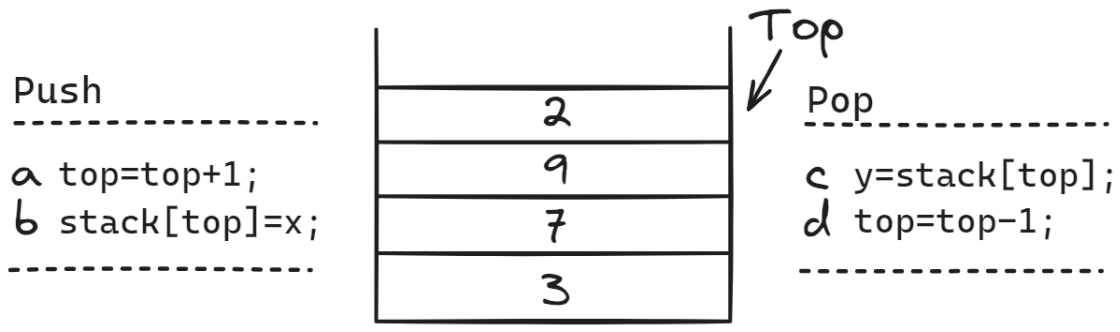
Le operazioni o_1 e o_2 danno luogo a una **race condition** sul dato condiviso **d** se eseguendo o_1 e o_2 con **d** avente un valore iniziale v_d, accade che **d** assume un valore v'_d diverso da $f_1(f_2(v_d))$ e da $f_2(f_1(v_d))$

Nell'esempio

- L'operazione o_1 è l'istruzione $x = x + 5$
- L'operazione o_2 è l'istruzione $x = x + 10$
- f_1 è la funzione tale che $f_1(z) = z + 5$
- f_2 è la funzione tale che $f_2(z) = z + 10$
- $f_1(f_2(100)) = 115$
- $f_2(f_1(100)) = 115$

Il valori che accetto sono 115 poichè sono quelli risultato della combinazione delle funzioni. Per gli altri valori ho race condition

Un esempio: due processi usano la medesima pila, un processo fa **push**, l'altro **pop**.



Supponiamo che l'ordine delle istruzioni sia *a,c,d,b*

- istruzione *a*: mi preparo per la push aumentato il puntatore al top della pila

- istruzione c: **y** assume il valore di qualcosa che **non è nella pila**.
- **x** viene inserito al posto dell'elemento 2 che era al top e che nessuno ha letto.

Se **a** viene implementata con:

- a_1: load top, R_0
- a_2: add R_0, 1
- a_3: store R_0, top

Puo darsi che il processore venga perso dopo l'istruzione a_1, dunque dopo che la variabile top sia stata letta in un registro,
successivamente il 2 viene estratto e il nuovo top dovrebbe individuare il 9.
adesso riprendiamo l'istruzione del thread uno (istruzione a_2) aggiungendo a R_0 (vecchio valore del top)
Il 100 dovrebbe rimpiazzare il 2. Tuttavia viene inserito una posizione dopo il 2.

Come prevenire le **race condition**? Gli accessi ai dati condivisi devono essere sincronizzati, cioè devono avvenire in **mutua esclusione:** *quando un processo accede ad un dato condiviso* `d`*, nessun altro processo deve poter accedere concorrentemente a* `d`.

5.2 Sezioni critiche

sezione critica

Una **sezione critica** per un dato condiviso **d** è una porzione di codice che viene eseguita non concorrentemente con se stessa o con altre sezioni critiche per d

è un pezzo di codice che lavora su un dato e si è certi che mentre queste istruzioni laorano su quel dato esso non viene modificato da un altra entità. Chi esegue la sezione critica ha esclusività sul dato.

Definition 5.2 (Mutua esclusione)**.** Si parla anche di **mutua esclusione**, nel senso che le sezioni critiche su un dato d sono mutualmente esclusive.

Se i processi modificano il dato d solo nelle CS per d, allora le race condition su d non possono aver luogo.

Se i processi accedono in lettura al dato d solo nelle CS per d, allora i processi vedono solo stati consistenti di d.

Esistono alcune proprietà richieste alle implementazioni delle CS:

- **correttezza**: le CS per un dato d non possono essere eseguite concorrentemente, cioè deve essere garantita la mutua esclusione;

- **progress**: se nessun processo sta eseguendo una CS per d, e alcuni processi manifestano la volontà di eseguire CS per d, allora uno di essi deve poter eseguire la propria CS per d;
- **bounded wait**: dopo che un processo P_i manifesta la volontà di accedere ad una CS per d, il numero di accessi alle CS per d da parte di un qualsiasi altro processo P_j che precedono l'accesso di P_i deve essere $\leq$ di un dato intero k. Ogni k accessi di P_j effettuiamo un accesso di P_i. *(hint: semafori)*

Il progress e la bounded wait prevengono la starvation, cioè l'attesa infinita da parte dei processi.

5.2.1 Tentativo 1 di implementare le sezioni critiche - disabilitare gli interrupt

```
disable_interrupt; /* istruzione che modifica i bit IM della PSW */
 {CS}; /* notazione che useremo per indicare una generica CS*/
enable_interrupt; /* istruzione che modifica i bit IM della PSW */
```

Questa "soluzione" ha alcuni problemi:

- la soluzione è corretta solo su sistemi uniprocessore.
- la soluzione richiede che le istruzioni per disabilitare gli interrupt siano eseguibili in user mode: un programma potrebbe usare queste istruzioni per monopolizzare la CPU

5.2.2 Tentativo 2 di implementare le sezioni critiche - uso di variabili lock condivise

L'idea è quella che un processo per entrare nella sezione critica deve passare attraverso una "porta", se essa è aperta il processo entra e la chiude. Se questa è chiusa rimane in attesa della sua riapertura.

Sia `lock` una variabile condivisa inizializzata a 0.

```
while (lock != 0){ } /* finche la porta è chiusa continuo a ciclare (waiting) */
lock=1; /* chiudo la porta e eseguo la sezione critica */
{CS}
lock=0; /* riapro la porta */
```

Questa soluzione non è corretta, perché la race condition può aver luogo sulla variabile lock: più processi potrebbero trovare lock = 0 (istruzioni load lock,R eseguite da più processi prima dell'aggiornamento di lock), porre lock ad 1 ed entrare nella CS, dunque verrebbe eseguita in concorrenza.

5.2.3 Tentativo 3 di implementare le sezioni critiche - uso di variabili turno condivise

Vediamo il caso semplice, con due processi:

```
Processo P0                    Processo P1

while(turn ≠ 0){}              while(turn ≠ 1){}
    {cs}                           {cs}
turn = 1;                      turn = 0;
```

Questa soluzione è corretta ma non soddisfa la proprietà del **progresso**: se turn vale inizialmente 0 e P1 è il primo processo a voler entrare nella CS, non riesce a farlo, nonostante P0 non sia nella propria CS. Se P0 non entra mai nella propria CS allora P1 è bloccato all'infinito: **abbiamo starvation**.

5.2.4 Operazione indivisibile

Definition 5.3. Un'operazione indivisibile su un dato condiviso `d` è un'operazione che è con certezza eseguita in modo non concorrente rispetto ad altre operazioni su d.

Un'operazione indivisibile viene talvolta detta atomica.

Per definizione, le operazioni indivisibili su `d` non possono dar luogo a race condition su `d`.

Dunque come possiamo creare delle operazioni indivisibili?

Operazione test and set:

L'istruzione TS, proposta originalmente per l'IBM/370, esegue due azioni :

1. Controlla se una locazione di memoria ha valore 0, ponendo il risultato nel bit CC (Condition Code) della PSW;
2. imposta la locazione di memoria con una sequenza di 1

in breve accede ad una locazione di memoria sia in lettura che in scrittura.

Vediamo il suo utilizzo concretamente:

- Idea in pseudocodice

  ```
  entry_test if(lock=closed){goto entry_test;}
      lock = closed;
      {CS}
      lock=open;
  ```

  Se lock è closed continuo a ciclare nell'entry test, se è aperto lo chiudo ed eseguo la sezione critica. (è fondamentale non perdere il processore tra il test di lock e la modifica.)

- Implementazione in n IBM/370 con convenzione lock 0 = aperto

```
LOCK      DC X '00' /* LOCK inizializzata con l'esadecimale 0 */
ENTRY_TEST   TS LOCK /* Test and Set sulla variabile LOCK */
     BC 7, ENTRY_TEST /* Branch ad ENTRY_SET se i 3 bit
     CC hanno valore 111 */
 {CS}
     MVI LOCK, X'00' /* assegna l'esadecimale 0 a LOCK */
```

- Inizialmente inizializziamo la variabile a 0
- Successivamente, finchè la variabile non vale 1, aspetto. (se la variabile vale 0, la branch non viene eseguita e quindi eseguo la sezione critica)
- succesivamente assegno lock a 0.

In questo caso il controllo e l'assegnazione vengono fatte con una singola istruzione macchina **quindi non posso perdere il processore**.

Una versione alternativa della ts è la **swap**.

Definition 5.4. Istruzione swap istruzione indivisibile che scambia il contenuto di due locazioni di memoria.

```
TEMP     DS 1 /* viene riservato 1 byte per TEMP */
LOCK     DC X '00' /* il lock è inizialmente aperto */
         MVI TEMP X'FF' /* TEMP assume valore 11111111*/
ENTRY_TEST  SWAP LOCK, TEMP /* testa e chiudi il lock*/
         COMP TEMP, X'00' /* il risultato del test (TEMP == 0)
 va nel CC della PSW*/
         BC 7, ENTRY_TEST
         {CS}
         MVI LOCK, X'00' /* riapri il lock */
```

- Inizialmente viene riservato 1 byte per TEMP
- Inizializzo la lock a 0 (aperto)
- Assegno a temp il valore 1
- Dopodichè scambio il contenuto di lock e temp
- Vado a confrontare il valore di temp con il valore 0 (testo se temp vale 0, quindi se la porta era inizialmente aperta)
- Se la porta è chiusa ricomincio da capo, rifaccio il ciclo partendo da entry test finchè è chiusa.

5.3 Approccio algoritmico alle sezioni critiche

5.3.1 Logica

L'**approccio algoritmico** prevede di implementare le CS senza usare istruzioni hardware ad hoc, *system call* o costrutti del linguaggio ad hoc:

- un processo controlla alcune condizioni logiche prima di entrare in una CS;
- se il controllo è superato, il processo entra nella CS;
- se il controllo non è superato, il processo reitera il controllo: siamo di fronte ad un caso di **busy wait**. La busy wait sarebbe evitabile se si ponesse il processo in stato di waiting, ma ciò richiederebbe l'invocazione di una system call, possibilità che è stata esclusa.

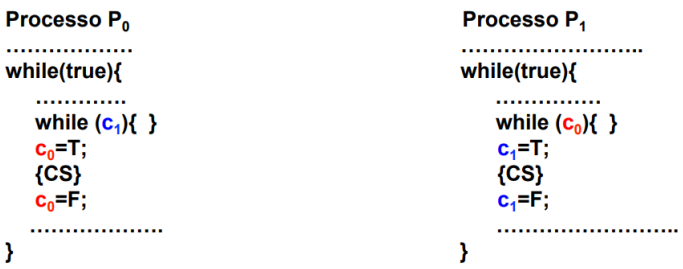
Idea generale

```
while(not ok){ /* busy wait*/ }
{CS}
```

5.3.2 Uso delle variabili turno - secondo tentativo

```
boolean c0=false, c1=false
```

quando la variabile c_i è true significa che il processo p_i sta eseguendo la sezione critica, altrimenti no.



- Finchè c_1 è true, ossia il processo c_1 sta lavorando nella sezione critica, rimango in attesa (nel loop).
- quando il processo "rivale" non sta lavorando, imposto la variabile relativa al processo 0 su true e eseguo la sezione critica. Successivamente reimposterò la variabile a false.
- L'altro processo compirà le stesse azioni, ma con verso opposto.

La proprietà del progresso è rispettata, ma questa soluzione **non è corretta**, cioè l**a mutua esclusione non è garantita**. Infatti, per esempio, P0 può essere interrotto dopo aver controllato la variabile c1 e prima dell'assegnamento a c0, in tal modo anche P0 può entrare nella CS.

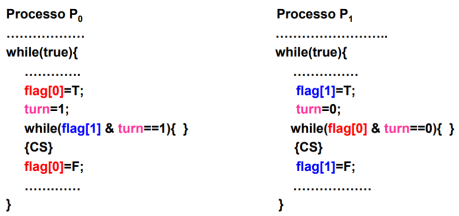


Figure 8: Enter Caption

5.3.3 Uso delle variabili turno - terzo tentativo

Potrei quindi pensare prima di assegnare la variabile e successivamente effettuare il controllo.

```
boolean c0=false, c1=false
```

Processo P_0
```
while(true){
    c0=T;
    while (c1){ }
    {CS}
    c0=F;
}
```

Processo P_1
```
while(true){
    c1=T;
    while (c0){ }
    {CS}
    c1=F;
}
```

Ora la mutua esclusione è garantita, ma potrebbe accadere che p0 setti la variabile a true, successivamente perda il processore e p1 setti a sua volta la varaibile a true. In questo casi ci troveremmo in una situazione di stallo (**deadlock**), i processi si bloccano a vicenda. Ovviamente è violata la proprietà del progresso

5.3.4 Uso delle variabili turno - terzo tentativo

L'idea qui è quella che p0 "alzi la bandiera" (c0 =true) e se anche l'altro processo l'ha alzata la abbasso, per evitare il deadlock.

Processo P_0
```
while(true){
a   c0=T;
    while (c1){c0=F;  goto a}
    {CS}
    c0=F;
}
```

Processo P_1
```
while(true){
a   c1=T;
    while (c0){c1=F; goto a}
    {CS}
    c1=F;
}
```

Ora la mutua esclusione è garantita e non può esserci deadlock, ma entrambi i processi potrebbero eseguire all'infinito l'istruzione etichettata a ed il while:

situazione di **livelock**, cioè i processi si ostacolano a vicenda. Ovviamente è violata la proprietà del progresso.

5.3.5 Algoritmo di Dekker

```
boolean c0=false, c1=false, turn=0
```

```
Processo P0
.....
while(true){
   .....
   c0=T;
   while(c1){
      if(turn==1){
         c0=F;
         while(turn==1){ }
         c0=T;
      }
   }
   {CS}
   turn=1;
   c0=F;
   .....
}
```

```
Processo P1
.....
while(true){
   .....
   c1=T;
   while(c0){
      if(turn==0){
         c1=F;
         while(turn==0){ }
         c1=T;
      }
   }
   {CS}
   turn=0;
   c1=F;
   .....
}
```

- Inizialmente alzo la bandiera
- Controllo se anche l'altro processo ha alzato la bandiera
  - Se è abbassata lavoro e successivamente la abbasso
  - altrimenti **uso il turno**: se il turno è di un altro processo allora si dovrà abbassare la bandiera, e si aspetterà finchè non è il mio turno. Successivamente rialzo la bandiera e rientro nel while.

Questa è una soluzione vincente ma valida solo per 2 processi.

5.3.6 Algoritmo di Peterson (processi gentili)

```
boolean[] flag={F,F},int turn=0
```

```
Processo P0
.....
while(true){
   .....
   flag[0]=T;
   turn=1;
   while(flag[1] & turn==1){ }
   {CS}
   flag[0]=F;
   .....
}
```

```
Processo P1
.....
while(true){
   .....
   flag[1]=T;
   turn=0;
   while(flag[0] & turn==0){ }
   {CS}
   flag[1]=F;
   .....
}
```

Anche questa è una soluzione vincente ma valida solo per 2 processi.

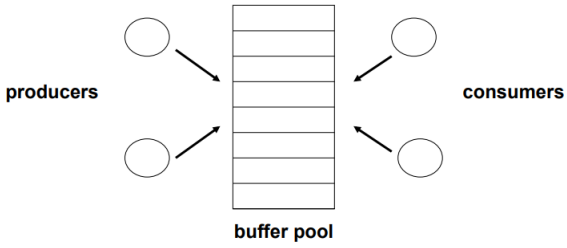
- Inizialmente p0 alza la bandiera, facendo presente l'intenzione di entrare nella sezione critica, dopodichè lascia il turno al rivale. p1 fa la stessa cosa.
- Finche il rivale ha la bandiera alzata ed è il suo turno aspetto
- altrimenti lavoro io

5.4 Problemi classici della programmazione concorrente

5.4.1 Problema dei produttori e consumatori

Si assumano

- Un **pool finito di buffer** (banalmente una struttura dati condivisa), ognuno dei quali può essere pieno, cioè contenere un record di informazione, oppure vuoto
- un insieme di **processi produttori**, che producono singoli record di informazione con cui riempire i buffer;
- un insieme di **processi consumatori**, che consumano singoli record di informazione svuotando i buffer.



Soluzione valida al problema:

- l'accesso ai singoli buffer avviene in mutua esclusione;
- i produttori non possono riempire i buffer pieni, cioè già riempiti e non ancora svuotati;
- i consumatori non possono svuotare i buffer vuoti, cioè mai riempiti oppure riempiti e già svuotati.

Talvolta c'è una condizione aggiuntiva: i buffer devono essere svuotati nello stesso ordine in cui sono stati riempiti (politica FIFO).

Vediamo una possibile soluzione, assumendo che il pool sia un array di interi da 100 elementi gestito in modo "circolare".

```
int count = 0; /* numero di buffer pieni, è ovviamente <= 100 */
int i = 0; /* indice del primo buffer vuoto, valido se count <= 99 */
int j = 0; /* indice del primo buffer pieno, valido se count >= 1 */
```

Soluzione con race condition possibili:

```
Programma del producer
 int item;
 while(true){
    ……..
    item = produce_item;
    while(count == 100){ }
    buffer[i] = item;
    count ++;
    i = (i+1) % 100;
    ……..
}
```

```
Programma del consumer
 int item;
 while(true){
    ……..
    while(count == 0){ }
    item = buffer[j];
    count - -;
     j = (j+1) % 100;
     ……..
 }
```

Programma del producer

- Creo un elemento da inserire nella pool, supponiamo di chiamarlo `item`
- Finchè la pool è piena non faccio nulla
- Quando l'array non è pieno l'elemento di indice i lo riempio con l'item
- Successivamente incremento la variabile *count* e la variabile i(%100 per gestirlo in modo circolare, dopo la posizione 99 c'è la 0)

Programma del consumer

- Finchè la pool è vuota aspetto
- quando è presente qualcosa, leggo l'elemento in posizione j
- Successivamente decremento la variabile *count* e la variabile i

Per prevenire le race condition, è sufficiente che gli accessi ai dati condivisi avvengano nelle CS, riportate in blu.

```
Programma del producer:
 int item;
 while(true){
    ……..
    item = produce_item;
    while(count == 100){ }
    buffer[i] = item;
    count++;
    i = (i+1) % 100;
    ……
}
```

```
Programma del consumer:
 int item;
 while(true){
    ……..
    while(count == 0){ }
    item = buffer[j];
    count--;
     j = (j+1) % 100;
    ……..
}
```

Uso dei **semafori** per il problema dei produttori e consumatori. caso facile: abbiamo un solo buffer.

```
semaphore full = 0;      /*     full = 1 if and only if buffer full */
semaphore empty = 1;  /* empty = 1 if and only if buffer emtpy*/

Programma del producer:
 int item;
 while(true){
    ……..
    item = produce_item;
    wait(empty);
    buffer = item;
    signal(full);
    ……..
 }

Programma del consumer:
int item;
while(true){
   ……..
   wait(full);
   item = buffer;
   signal(empty);
   ……..
 }
```

- la variabile `buffer` è condivisa
- la variabile `full` e `empty` assumono valori tra 1 e 0. Indicano rispettivamente il numero di buffer pieni e vuoti.

Programma del produttore:

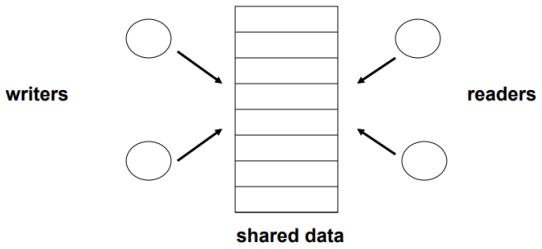
- Un volta prodotto item lo voglio inserire nel buffer
- posso eseguire `buffer=item` solo se il buffer è vuoto
- intuitivamente posso inserire nel buffer solo se `empty=1`, ossia ho un elemento vuoto.
- successivamente aggiorno full, valeva zero, ora vale uno.
- potrebbe capitare che ci sia un consumatore in attesa, in quel caso full non viene incrementato ma viene bensi svegliato il consumatore in attesa.

5.5 Problema dei lettori e scrittori

Si assumano:

- Un insieme di dati condivisi
- un insieme di processi **lettori**, che accedono ai dati in sola lettura;
- un insieme di processi **scrittori**, che accedono ai dati in sola scrittura.

L'accesso alla struttura dati o viene effettuato in un certo istante da n lettori oppure può accedere un solo scrittore. I lettori escludono gli scrittori, il singolo scrittore esclude i lettori.



Vediamo l'idea generale della soluzione

```
Programma del reader
.............
while(true){
  ........
  while(someone is writing){ }
  set (I' m reading);
  {read};
  set (I' m not reading);
  ........
}
```

```
Programma del writer
.............
while(true){
  .............
  while(someone is working){ }
  set (I' m writing);
  {write};
  set (I' m not writing);
  .........
}
```

5.5.1 Soluzione

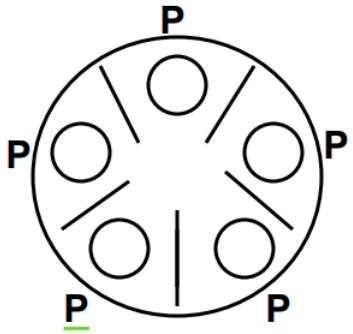
Uso dei semafori per il problema dei lettori e scrittori.
Assumiamo di avere **R** lettori e **W** scrittori. Usiamo i seguenti semafori e variabili:

- **semR**: semaforo inizializzato a 0 – serve per dare l'ok ai lettori
- **semW**: semaforo inizializzato a 0 – serve per dare l'ok agli scrittori
- **mutex**: sem. inizializzato a 1 - serve per la mutua esclusione sulle variabili seguenti
- **runR**: numero di lettori che stanno leggendo
- **runW**: numero di scrittori che stanno scrivendo
- **totR**: numero di lettori che stanno leggendo o che vorrebbero leggere
- **totW**: numero di scrittori che stanno scrivendo o che vorrebbero farlo

...

5.6 Problema dei filosofi a cena

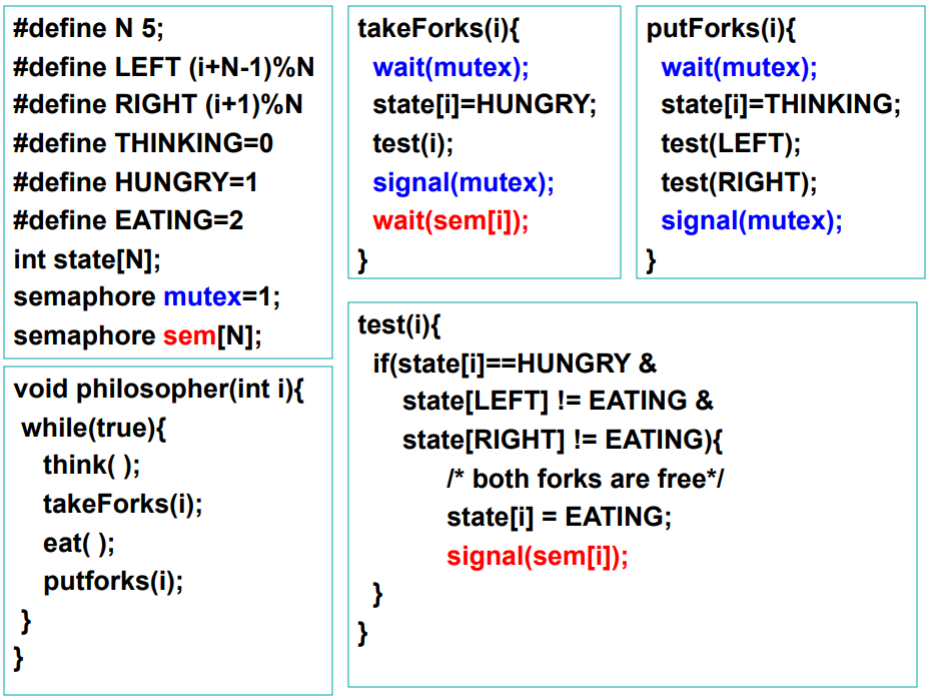
5 filosofi sono seduti intorno ad un tavolo, ed alternano momenti in cui pensano a momenti in cui mangiano. Per mangiare hanno bisogno di due bacchette. Le bacchette disponibili sono 5, ogni filosofo ne ha una alla sua destra ed una alla sua sinistra, per mangiare ha bisogno di entrambe.



5.6.1 Soluzione con semafori

Utilizziamo le seguenti strutture dati:

- Un array di stati con valori HUNGRY, EATING, THINKING
- Un semaforo `mutex`, per accedere all'array in mutua esclusione
- Un semaforo `sem[i]` per ogni filosofo P_i che serve per dare l'ok a P_i



`void philosopher(int i)`, Il filosofo `i` all'infinito

1. pensa
2. esegue la procedura `takeForks(i)`: se le forchette sono libere le prendo e mangio, altrimenti mi addormento in attesa
3. mangia
4. esegue la procedura `putForks(i)`

La procedura `takeForks(i)` funziona nel seguente modo:

1. `wait(mutex)` ...`signal(mutex)` voglio evitare race condition sull'array condiviso `state`
2. `state[i]=HUNGRY` dico a tutti che ho fame
3. applico la procedura `test(i)`
   - se il filosofo a destra e anche quello a sinistra non stanno mangiando allora mangio.
   - il mio semaforo, che in precedenza valeva 0, ora vale 1.
4. quando finisco la procedura test rilascio la mutua esclusione con la signal
5. faccio la wait sul semaforo, essendo a uno non vado in waiting

6. successivamente eseguo la `eat()`

Se uno dei filosofi vicini stesse mangiando non avrei eseguito la `state[i]=EATING`, `signal(sem[i])`, dunque il semaforo sarebbe rimasto a 0, e mi sarei messo in attesa con la `wait(sem[i])` Quando finisco di mangiare eseguo la procedura

`putForks(i)`

- prima di tutto cambio lo stato in THINKING
- eseguo la procedura `test()` sui miei vicini
- fare la procedura `test()` sul vicino di sinistra(destra) significa che se lui è hungry e i suoi vicini non stanno mangiando lui può mangiare. Dunque dico che è nello stato EATING e lo sveglio.

6 Semafori

Semaforo

Un **semaforo** è una variabile intera condivisa che **può assumere solo valori non negativi** e su cui sono possibili solo le seguenti tre operazioni:

- Inizializzazione: (con un valore ≥ 0)
- Operazione indivisibile *wait*:
  - Se il semaforo ha valore > 0, viene decrementato.
  - Se il semaforo ha valore $= 0$ il processo che esegue la wait viene "bloccato sul semaforo", cioè è impossibilitato a proseguire. Di fatto va in waiting.
- Operazione indivisibile *signal*:
  - Se c'è almeno un processo bloccato sul semaforo, uno dei processi bloccati sul semaforo viene "sbloccato", cioè può riprendere l'esecuzione. Di fatto va in stato di **ready**
  - Se nessun processo è bloccato sul semaforo, il semaforo viene incrementato.

6.1 Uso dei semafori per l'implementazione delle sezioni critiche

Supponiamo che `sem` sia un semaforo inizializzato a 1.

```
while(true){
 /*..............*/
     wait(sem);
     {CS}
     signal(sem);
 /*..............*/
}
```

1. Quando un processo fa la wait, il semaforo andrà a 0.
   Il primo processo che fa la wait mette il semaforo a 0, ed esegue la sezione critica. Una volta finita la sezione critica viene eseguita a signal e il tutto ricomincia.
2. Una situazione che può capitare è quella di perdere il processore quando si è nella sezione critica. Quando il processo che ha ottenuto il processore esegue la wait, trova il semaforo a 0: dunque rimane in waiting. Quindi abbiamo la certezza che nessun processo può entrare la sezione critica, la **correttezza** è soddisfatta.

6.1.1 Bounded concurrency

Ho un insieme di n processi che condividono c risorse.
Esempio: abbiamo **N** processi che condividono **C** stampanti: *sem è un semaforo inizializzato a C*

```
while(true){
 /*..............*/
 wait(sem); /*process blocks if and only if all printers already in use*/
 {use a printer}
 signal(sem);
 /*..............*/
}
```

Inizializzando il semaforo a c, i processi eseguiranno la wait, useranno la stampante e infine eseguiranno la signal.

6.1.2 Impementazione dei semafori

Un semaforo richiede:

- Un intero per memorizzare il valore del semaforo
- Una variabile lock per garantire l'indivisibilità di `wait` e `signal`
- Una lista di processi bloccati sul semaforo:
  - può essere realizzata usando i pointer dei PCB
  - viene solitamente gestita come coda per assicurare che i processi bloccati sul semaforo vengano svegliati con la politica FIFO.

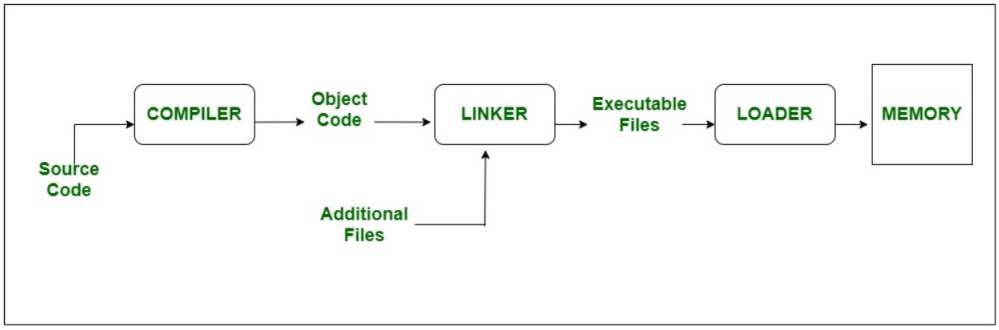
Implementazione dei semafori a **kernel level**:

- I semafori sono strutture dati del kernel: la lista dei processi bloccati sul semaforo sul semaforo può essere realizzata sfruttando i pointer dei PCB, di cui il kernel è "proprietario";
- wait e signal sono system call. Ovviamente possono esserci funzioni wrapper di libreria wait e signal per invocarle; la loro indivisibilità è ottenuta usando variabili lock ed istruzioni indivisibili (TS, SWAP).

7 Gestione della memoria

7.1 Traduzione, linking, loading: alcune precisazioni

Traduzione, linking e loading di un programma sorgente



1. Il codice sorgente viene preso in carico da un compilatore, il risultato della traduzione del compilatore si chiama *modulo oggetto*(o codice oggetto)
2. Il *modulo oggetto* è qualcosa scritto in linguaggio macchina ma non necessariamente è pronto ad essere eseguito poichè è possibile che nel codice sorgente vengano utilizzate variabili o funzioni che non ho dichiarato all'interno del programma, ma bensì in librerie.
3. Il linker guarda se le variabili e funzioni che non ho dichiarato sono presenti in qualche libreria.
4. L'eseguibile ora è disponibile sull'hard-disk, se si va a chiedere l'esecuzione di questo eseguibile il - carica il programma in memoria e lo mette in esecuzione.

7.1.1 Esempio programma in assembly

```
        START   500          /*il programma ha origine all'indirizzo 500*/
        ENTRY   TOTAL        /* simbolo definito nel presente modulo*/
                             /* e "visibile" da altri moduli */
        EXTRN   MAX, ALPHA   /* simboli definiti in altri moduli*/
        READ    A            /* simbolo definito nel presente modulo*/
LOOP    ..                   /* LOOP è una label */
        ..
        MOVE    R1, ALPHA    /*R1 è il registro 1*/
        BC      ANY, MAX     /*ANY codifica una condizione del CC*/
        ..
        ..
        BC      LT, LOOP     /*LT codifica una condizione del CC*/
        STOP
A       DS      1            /* definizione del simbolo A */
TOTAL   DS      1            /* definizione del simbolo TOTAL */
        END
```

- La variabile total puo essere vista anche da altri moduli

- Le variabili `EXTRN` MAX e ALPHA non hanno un valore e un indirizzo, sono definite in altri moduli. Il linker cercherà il modulo dove sono definite e le assegnerà il valore

Abbiamo assunto un linguaggio assembly in cui le istruzioni hanno la forma seguente:

[label] opcode operand1 operand2

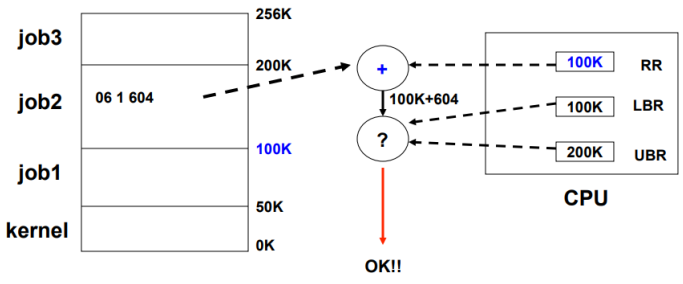
Assumiamo anche che in linguaggio macchina un'operazione occupi 4 byte:

- opcode occupa 2 cifre, dove 09=READ, 04=MOVE, 06=BC.
- operand1 occupa 1 cifra. Solitamente è un registro, ma nella BC l'operando codifica un valore possibile del Condition Code, per esempio ANY, LT. Assumiamo che in linguaggio macchina il codice decimale di ANY sia 6 e di LT sia 1.
- operand2 occupa 5 cifre.

...
Osservazioni:

- Nell'esempio precedente abbiamo visto un esempio di **rilocazione statica** da parte del loader: il loader modifica gli indirizzi del programma prima che esso venga eseguito.
- Con un opportuno supporto hardware, per esempio con l'impiego di un Relocation Register e della logica necessaria a sommare agli indirizzi il valore del Relocation Register, possiamo avere **rilocazione dinamica**

Esempio di rilocazione dinamica



Abbiamo un modulo hardware che sfrutta un registro (relocation register) dove tutte le volte che abbiamo un'istruzione l'indirizzo viene sommato al valore del relocation register dalla MMU a tempo di esecuzione.

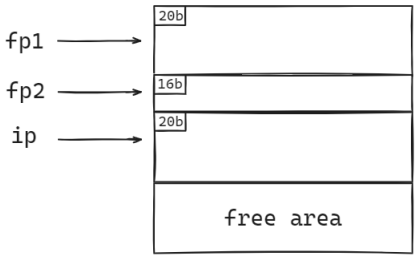
7.2 Modello di allocazione della memoria

Durante l'esecuzione del programma vengono allocati due tipi di dati:

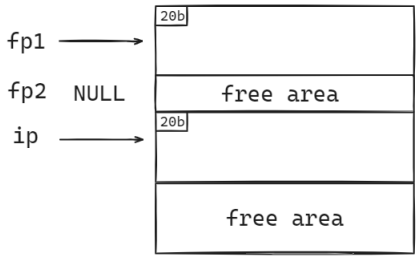
- Le variabili il cui scope è associato a blocchi e funzioni. Queste variabili vengono allocate sullo stack, a causa della natura LIFO della loro allocazione o deallocazione.
- I dati creati dinamicamente con i costrutti appositi, quali malloc e calloc nel linguaggio C oppure new in C++ e Java. Parleremo di dati PCD (Program Controlled Dynamic data). Vengono allocati nell'area di heap

Esempio di uso di dati PCD in C

```
float *fp1, fp2; /* pointers a float */
int *ip; /* pointer a int */

fp1= (float *) calloc(5,sizeof(float));
 /*lo heap allocator alloca nello heap un'area di memoria per 5 float e
restituisce un pointer a tale area*/
fp2= (float *) calloc(4,sizeof(float));
 /*lo heap allocator alloca nello heap un'area di memoria per 4 float
e restituisce un pointer a tale area*/
ip = (int *) calloc(10, sizeof(int));
 /*lo heap allocator alloca nello heap un'area di memoria per 10 int e
restituisce un pointer a tale area*/
free(fp2);
 /*lo heap allocator libera l'area di memoria puntata dal pointer fp2*/
```



Situazione dello heap dopo le tre calloc. Ogni area allocata ha un campo contenente la dimensione: serve per facilitare la deallocazione



Situazione dello heap dopo la free. Il pointer fp2 è ora nullo e l'area di memoria che puntava è diventata libera

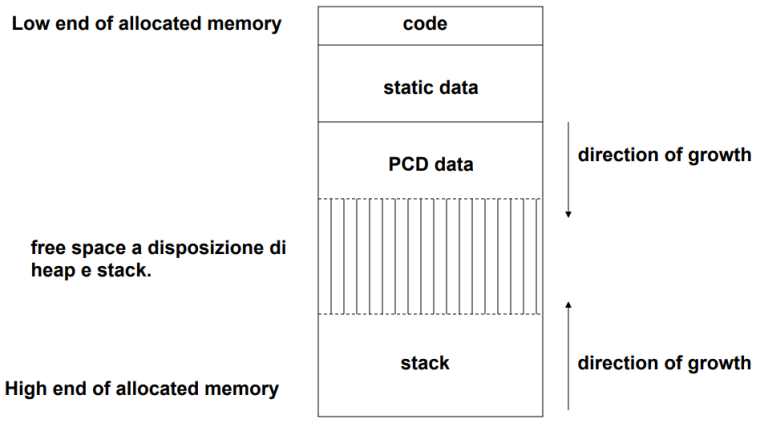
- L'effetto della `calloc` è quello di allocare un'area di RAM (porzione dello heap), questo compito viene svolto dallo **heap allocator**

Abbiamo usato il termine "heap allocator". Cosa intendiamo?

- Se il linguaggio offre i costrutti per allocazione/deallocazione di memoria nello heap, il linguaggio deve offrire delle routine per gestire lo heap che vengono invocate da tali costrutti (`calloc`, `free` ...).
- Queste routine fanno parte del RunTime Support (RTS) del linguaggio, si trovano nella runtime library e vengono collegate al programma dal linker.
- Possiamo chiamare heap allocator/deallocator **le routine che si occupano di allocare/deallocare memoria nello heap**.

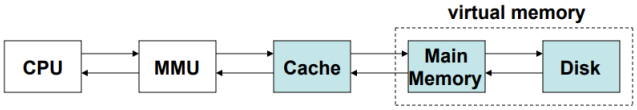
Modello di allocazione della memoria usato dal S.O.:

- il codice ed i dati statici sono allocati in un'area di dimensione statica,
- l'area di heap e lo stack condividono un'area di dimensione statica, ma le due aree hanno dimensione variabile e "crescono" in direzione opposta.



7.3 Gerarchia di memoria

Quando nelle istruzioni dei programmi abbiamo operandi che sono indirizzi (`BRANCH`, `LOAD`, `STORE`) sono intesi essere indirizzi di RAM (dunque si riferiscono a area code, stack, pcd, static).
Il programmatore/compilatore ragiona dunque come se esistesse solo la RAM.



- La CPU accede alla gerarchia di memoria tramite la MMU (Memory Management Unit), che traduce gli indirizzi logici in indirizzi fisici.
- Il codice di un programma contiene riferimenti alle aree testo, dati, stack, heap che sono assunte essere in main memory: gli indirizzi logici lavorati dalla MMU e gli indirizzi fisici prodotti della MMU sono indirizzi della main memory.
- Una volta che la MMU effettua la traduzione può capitare che l'indirizzo sia anche sulla cache. Se non viene trovato nella cache è necessario trasferirlo dalla memoria principale
- Se il sistema offre la **memoria virtuale** una parte delle aree di memoria del processo non si trovano nella main memory ma sul disco. Gli indirizzi fisici generati dalla MMU sono virtuali

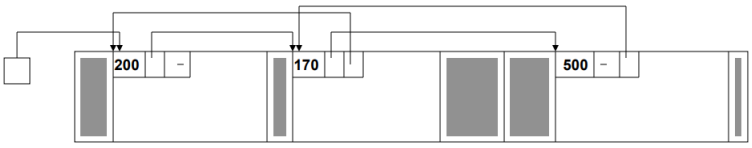
7.4 Riuso della memoria

L'allocazione della memoria è un problema che possiamo vedere a due livelli:

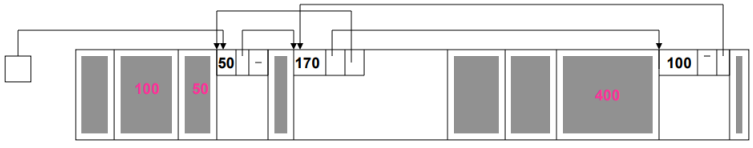
- Il S.O deve allocare memoria per i processi
- Il RTS deve allocare memoria per i dati PCD

In entrambi i casi è necessario riutilizzare la memoria liberata. Per farlo, è possibile tener traccia della memoria libera in una **free list**.
Se avessimo necessità di allocare un area di k byte e avessimo più zone di quella dimensione potremmo utilizzare 3 tecniche per scegliere l'area:

- **First fit**: *si cerca la prima area libera di dimensione* $\geq k$. Se l'area ha dimensione d, si allocano k byte e l'area rimanente di d-k byte rimane nella free list.
  Esempio: abbiamo 3 blocchi liberi: rispettivamente da 200, 170, 500 byte

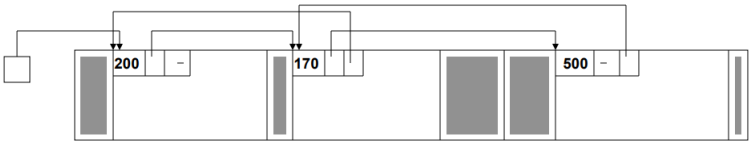


  Immaginiamo di allocare tre aree da 100, 50, 400 byte:

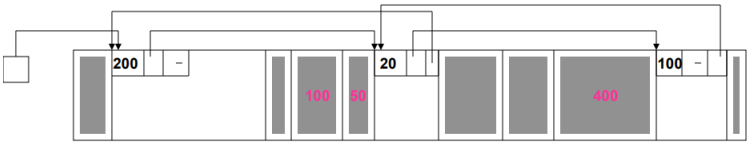


1. Vengono occupati i primi 100 byte della prima area, 100 rimangono liberi
2. Quando si chiede di allocare il secondo blocco da 50 byte, questi 100 vengono ulteriormente divisi. 150 sono stati occupati, 50 sono rimasti liberi.
3. Quando si chiede di allocare 400 byte, notiamo che le prime due aree non sono abbastanza grandi, quindi passiamo alla terza, lasciando 100 byte liberi.

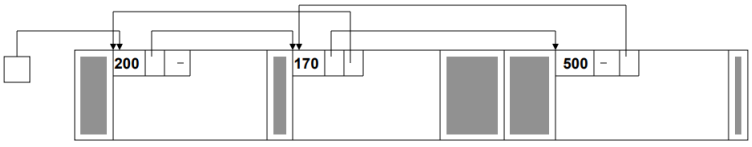
- **Best fit**: *si cerca l'area libera più piccola tra quelle di dimensione* $\geq k$. Se l'area ha dimensione d, si allocano k byte e l'area rimanente di d-k byte rimane nella free list
Esempio:



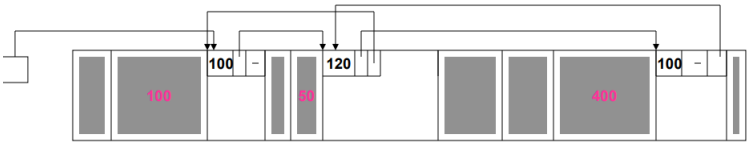
1. L'area più piccola tra quelle libere sufficiente a ospitare 100 byte non è la prima ma la seconda, dunque verranno allocati in quella zona. 70 byte rimarranno liberi
2. Quando si allocano i 50 byte, capiamo che possono stare nell'area precedente dove rimanevano 70 byte. Dunque 20 byte rimangono liberi
3. L'unica area che può ospitare i 400 byte è l'ultima



- **Next fit**: si adotta la first fit partendo, però, **dal punto in cui era stata effettuata l'ultima allocazione**
Esempio:



1. il primo blocco viene allocato nell'area da 200 byte
2. siccome con questa tecnica bisogna partire dal punto in cui era stata effettuata l'ultima allocazione il blocco da 50 byte verrà allocato nell'area da 170 byte
3. mentre l'ultimo blocco in quello da 400



Osservazioni su queste tecniche:

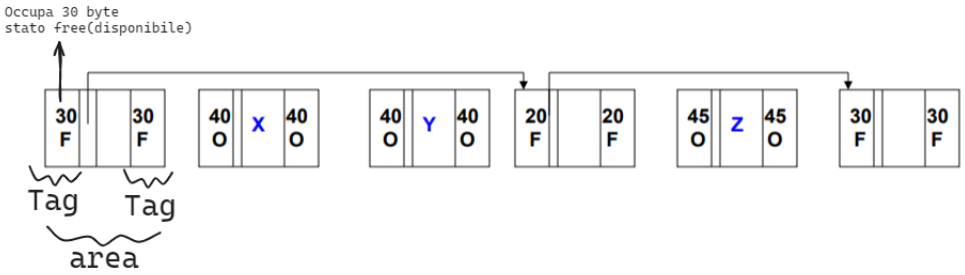
- Nel caso della first fit un'area libera può spezzarsi più volte. C'è la tendenza al formarsi di aree libere di dimensione piccola che rischiano di essere inutili.
- Nel caso della best fit vengono preservate la aree di dimensioni maggiore, ma a lungo andare il problema si presenta ugualmente. Inoltre la gestione della lista, oppure la ricerca, sono costose.
- La next fit è un compromesso
- Nella pratica, la **first fit** e **la next fit** sono **più performanti della best fit**, nel senso che è meno probabile che si formino aree di RAM che non verranno utilizzate.

Frammentazione

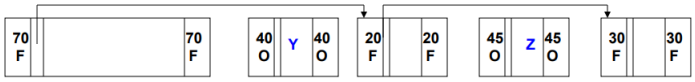
L'esistenza di aree di memoria non utilizzabili in un sistema di computazione è detta **frammentazione**

Esistono tecniche per prevenire la frammentazione:

- **Boundary tags**
  - All'inizio ed alla fine di ogni area c'è un tag, contenente la coppia (*stato free/occupied*, *dimensione*)
  - le aree libere sono collegate da una catena di pointer
  - quando un'area occupata viene liberata, se è possibile la si unisce ad un'area libera confinante (oppure a due)

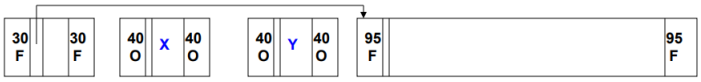


Se si libera l'area X, la si unisce alla free area che la precede



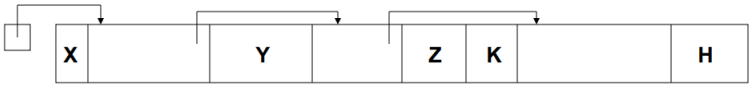
Se si libera l'area Y, la si unisce alla free area che la segue

Se si libera l'area Z, la si unisce alla free area che la precede e che la segue

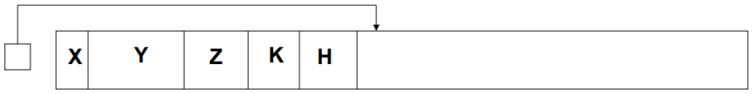


- **Memory compaction**
  - le aree libere sono collegate da una catena di pointer
  - a determinati intervalli di tempo tutte le aree libere vengono unite in un'unica area libera.

Intuitivamente fondiamo le aree di RAM libere



Dopo il compattamento:



Dal punto di vista pratico è richiesta la rilocazione dei programmi cui le aree occupate sono allocate. Ragionevole solo se la rilocazione è semplice, esempio se abbiamo il registro RR, perché in tal caso è sufficiente modificarne il valore. Se ho una rilocazione contigua posso farlo, ed è sufficiente che vada a cambiare il valore del relocation register di tutti questi processi.

7.5 Allocazione della memoria

7.5.1 Allocazione contigua

Si parla di **allocazione contigua** della memoria quando ogni processo è allocato in una singola area di memoria contigua.

- La protezione della memoria è semplice: è sufficiente disporre dei registri LBR / UBR
- La rilocazione dei processi è semplice: è sufficiente disporre del registro RR.
- Allocare in memoria un processo che richiede k byte è possibile solo se è disponibile un'area di memoria dimensione ¿=k. Sono pertanto necessarie le tecniche per prevenire / contrastare la frammentazione della memoria

L'allocazione contigua può essere integrata con lo **swapping**:

- alcuni processi sono in memoria
- altri processi sono sullo swap device (porzione di disco).

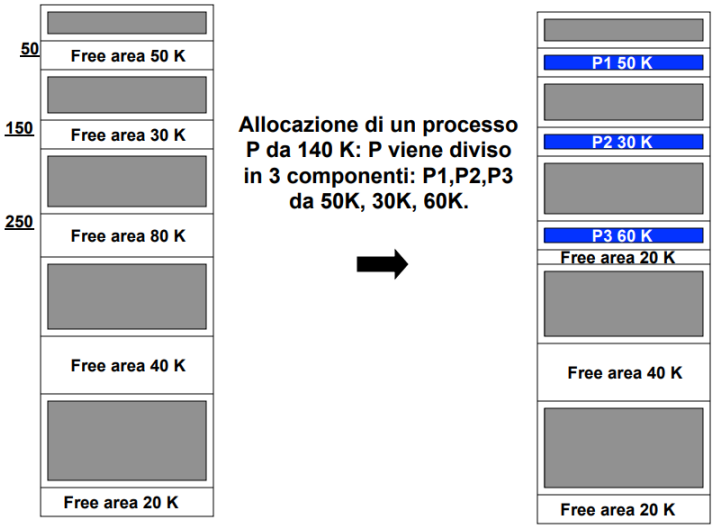
I processi in memoria waiting o ready possono essere **swapped out**. I processi sullo swap device ready possono essere **swapped in**.

7.5.2 Allocazione non contigua

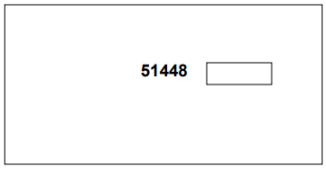
Si parla di **allocazione non contigua** quando ogni processo può essere allocato in più aree di memoria non adiacenti.

- Ogni porzione del processo che viene allocata in un'area contigua è detta componente.
- Allocare in memoria un processo che richiede k byte non necessita di una **singola** area di memoria di dimensione $\geq k$.

7.5.3 Allocazione non contigua - idea



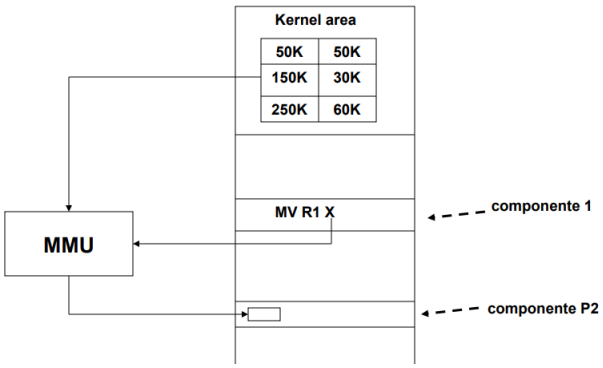
Assumiamo che la variabile x sia all'indirizzo logico 51448 del processo P della slide precedente:



Se un'istruzione fa riferimento ad x, qual è l'indirizzo fisico che l'istruzione deve considerare?

- La componente P1 ha dimensione 51200 byte (50 K).
- La componente P2 ha dimensione 30 K.
- L'indirizzo logico 51448 si trova pertanto nella componente P2, con offset 248 dall'indirizzo iniziale di P2, cioè 150 K .

Come tradurre l'indirizzo logico in indirizzo fisico? Ci pensa la MMU, sfruttando le informazioni relative all'allocazione di P memorizzate dal S.O



La MMU dovrebbe avere a disposizione una tabella dove abbiamo una entry per ogni area abbiamo la dimensione e l'indirizzo di inizio.
Partendo da un indirizzo logico, la MMU determina un indirizzo fisico, vale a dire:

- la componente
- l'offset rispetto all'indirizzo iniziale della componente

7.6 Paginazione e segementazione

7.6.1 Paginazione

- La memoria è divisa in 2^f **page frame** (aree equidimensionali), ciascuno di capacità 2^s byte, dove 2^{f+s} è la dimensione della memoria
- Ognuno dei 2^{f+s} byte della memoria è individuato da un indirizzo che consiste in f+s bit.
  - gli f bit più significativi (*leftmost*) individuano il **page frame**
  - gli s bit meno significativi (*rightmost*) individuano l'**offset** nel page frame.

Esempio, abbiamo una memoria da 4 giga (2^{22}) e la dividiamo in frame di dimensione 1k,
indirizzo 00000000000000000010100000110000:
page frame 10, offset 48

Ogni processo è logicamente diviso in pagine aventi le medesime dimensioni dei page frame. Se il processo ha dimensione d, le pagine sono $\frac{d}{2^s}$ (dove 2^s è la dimensione della pagina)

- Ognuno dei d byte dello spazio del processo è individuato da un indirizzo che consiste in l+s bit:
  - gli l bit più significativi (leftmost) individuano la pagina

– gli s bit meno significativi (rightmost) indicano l'offset nella pagina

- **Allocare** un processo in memoria significa *associare un page frame ad ogni sua pagina.*
- Non esiste frammentazione, fermo restando che ad ogni processo di dimensione d vengono allocati $\frac{d}{2^s} - d$ byte "di troppo", cioè l'ultimo page frame è usato solo in parte.

7.6.2 Esempio di paginazione

Siano s = 10, f= 22.
(Le pagine hanno dimensione 1K).
Assumo un processo di dimensione 4000 byte.
Il processo ha 4 pagine. (96 byte "sprecati".)
Se vengono allocate nei page frame 3, 20, 5, 9, la page table per P ha la forma seguente:

00.....000011
00.....010100
00.....000101
00.....001001

Nella page table associata:

- il frame numero 3 (000011) ospita la prima pagina
- il frame numero 20 (010100) ospita la seconda pagina
- il frame numero 5 (000101) ospita la terza pagina
- il frame 9 (001001) ospita la quarta pagina

Supponiamo di avere l'istruzione

```
MV R1 2100
```

2100 è l'**indirizzo logico**
Dunque per prima cosa dobbiamo quindi capire in quale frame si trova l'indirizzo 2100.

$$2100 = (2048 + 52) = (2K + 52)$$

Corrisponde alla terza pagina (ricordiamoci che sono pagine d 1k) con offset di 52.
è sufficiente prendere i bit più significativi per accedere a questa pagina dove legge il numero del frame associato ad essa e fa la sostituzione dei bit.
Dunque per effettuare la traduzione di un indirizzo consiste nella MMU che prende i bit più significativi e tramite la page table capisce

il page frame associato, successivamente sostituisce page frame associato ai bit più significativi, di fatto la MMU non deve fare alcun calcolo, solo un accesso alla RAM7 (l'offset rimane tale)

- Il S.O deve tener traccia dei page frame liberi per allocare i processi. Di norma vengono organizzati in una **free frame list**.
- La free frame list è una struttura del kernel cui i processi non possono accedere.

7.6.3 Segmentazione

- Ogni **processo** è logicamente **diviso in segmenti** che corrispondono ad entità logiche, quali, ad esempio, funzioni, strutture dati, oggetti
- Ogni segmento di ogni processo ha la propria dimensione e, pertanto, NON può avere un corrispondente segment frame in memoria.
- Ognuno dei byte dello spazio del processo è individuato da un indirizzo composto da due dati:
  - il numero del segmento
  - l'offset nel segmento.

Se facessimo una segmentazione "pura" (non si fa mai)

- Allocare un processo in memoria significa associare un'area di memoria libera ad ogni suo segmento. Non è richiesto che tali aree siano adiacenti.
- Tuttavia sarebbe ancora possibile la frammentazione, perché i segmenti possono avere dimensione maggiore delle dimensioni delle aree di memoria libere.
- La segmentazione **è più adatta a facilitare la condivisione di moduli di codice, strutture dati, oggetti** che a prevenire la frammentazione
- Per ogni processo il S.O. mantiene una segment table, che tiene traccia di due dati:
  - l'indirizzo dell'area di memoria cui è allocato il segmento
  - la dimensione del segmento

  In pratica è una situazione simile a quella dell'allocazione non contigua
- Dato un indirizzo logico che consiste del numero di segmento n e dell'offset o, la MMU lo converte nell'indirizzo fisico calcolato come segue:
  - l'indirizzo iniziale del segmento che si trova nella entry n della segment table viene sommato all'offset o.

  Il calcolo dell'indirizzo con la paginazione era meno costoso.

7.6.4 Esempio di segmentazione

Assumo un processo con 3 segmenti: `main, search, update`, di dimensione 2K, 3K, 4K.
Se vengono allocati agli indirizzi 100K, 50K, 80715, la segment table avrà la forma seguente:

2K	100K
3K	50K
4K	80715

Assumiamo che il processo abbia l'istruzione `call update, f1`, dove *f1* è una funzione nel segmento *update* che si trova all'offset *52*. La MMU accede alla terza riga della segment table ed aggiunge all'indirizzo inziale del segmento, cioè 80715 a 52, dopo aver controllato che 52 sia minore della dimensione del segmento (4K)

La segmentazione di solito viene abbinata alla paginazione.

- I segmenti sono entità logiche divise in pagine che hanno la stessa dimensione dei page frame.
- Ogni entry della segment table contiene un pointer alla page table del segmento
- si hanno i vantaggi di entrambe le tecniche

7.7 Memoria virtuale

Parliamo di **memoria virtuale** quando il sistema operativo crea l'illusone che il sistema abbia più memoria di quella effettiva, L'obiettivo è di rendere i processi indipendenti dalla capacità di memoria del sistema.
L'implementazione si basa sull'uso del disco

7.7.1 Demand paging

- La memoria è divisa in page frame ed i processi sono costituiti da pagine.
- L'intero spazio logico dei processi è memorizzato su un **paging device**, cioè un disco o una sua porzione
- L'area del paging device allocata per un processo viene detta **swap space** del processo
- Quando un processo inzia l'esecuzione è sufficente avere una pagina di esso nella RAM, che è la pagina che contiene la prima istruzione.
- Quando il processo tenta di accedere ad una pagina cui non è assegnato nessun page frame, la pagina viene copiata dallo swap space in un page frame libero

Il demand paging si basa su tre concetti fondamentali:

- **page fault**: eccezione che viene sollevata dalla MMU quando un processo tenta di accedere ad una pagina che non è associata a nessun page frame. *cerco di accedere alla variabile x, non è in ram, interruzione page fault*
- **page-in**: in seguito al *page fault*, il sistema operativo copia la pagina dallo swap space in un page frame libero
- **page-out**: il sistema operativo libera un page frame, se la pagina associata è stata modificata dopo l'ultimo page-in, deve essere copiata sullo swap space
- **page replacement**: consiste nel liberare un page frame e nel successivo page-in di una pagina diversa

Ogni entry della page table deve contenere i seguenti campi:

- **validity bit**: vale 1 se la pagina è in un frame, 0 altrimenti
- **frame #**: numero del page frame associato alla pagina, significativo solo se **validity bit**=1
- **prot** (protection info): codifica dei permessi per accedere alla pagina in lettura/scrittura
- **ref** (reference info): campo che codifica l'informazione circa i recenti accessi alla pagina, utile per scegliere quali pagine devono subire lo swap out;
- **mod bit** (modified bit): vale 1 se la pagina è stata modificata dopo l'ultimo page-in e, pertanto, richiederà il page-out quando il frame verrà liberato, 0 altrimenti;
- **disk address**: indirizzo della pagina nello swap device

Traduzione da parte della MMU di un indirizzo logico di $l + s$ bit, dove gli l bit leftomost costituiscono il page # e gli s rightmost bit l'offset.
(gli l bit potrebbero essere tali per cui $l > f$ (numero di page frame), dove il singolo processo può essere più grande della RAM, infatti una parte del processo può essere non in RAM)

1. Gli l leftmost bit determinano la entry della page table da utilizzare
2. Se *validity bit = 0* (la pagina è su un frame, non è in RAM) viene sollevato un page fault:
   - Il page fault handler usa il valore del campo **disk address** per determinare il blocco dello swap device da leggere
   - il page fault handler deve accedere alla free frame list per assegnare un frame libero alla pagina

- dopo aver copiato la pagina dallo swap device nel page frame, il page fault handler devve settare il validity bit a 1, settare il frame #, e passiamo al punto 3

3. Se *validity bit = 1* l'indirizzo fisico viene determinato concatenando gli f bit del frame # con gli s bit rightmost dell'indirizzo logico

7.7.2 Traduzione con TLB

La traduzione descritta precedentemente prevede due cicli di memoria per ogni accesso alla memoria:

- il primo ciclo per accedere alla page table
- il secondo ciclo per l'accesso vero e proprio

Per risparmiare un ciclo la MMU può usare un TLB o Translation Look-aside Buffer, cioè una memoria associativa contenente coppie (page #, frame #)
Se quando quando vogliamo tradurre l'indirizzo di una pagina la coppia (page #, frame #) si trova sul TLB la traduzione viene fatta senza accedere alla RAM.
Le memorie associative (accesso per contenuto e non per indirizzo) sono costose: il TLB contiene la coppia (page #, frame #) solo per alcune pagine. Dato

l'indirizzo logico page #, offset, la MMU accede al TLB:

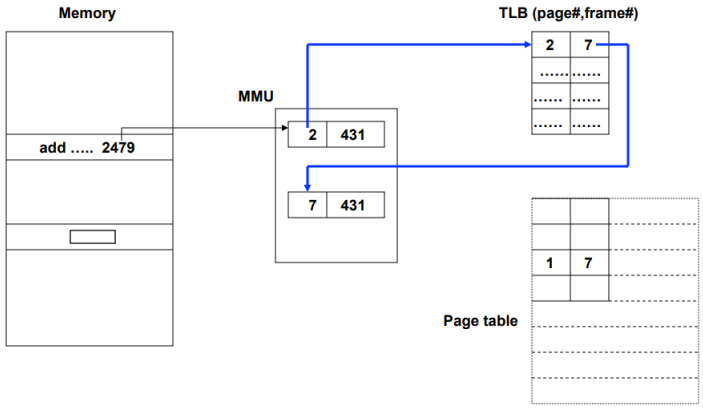
- se nel TLB c'è la coppia (page #, frame #) la MMU costruisce l'indirizzo fisico frame #, offset
- se non c'è nessuna coppia (page #, frame #) per il page # considerato si parla di **TLB miss** e la MMU usa la page table in memoria come descritto in precedenza. In questo caso, la coppia (page #, frame #) viene aggiunta al TLB, eliminando eventualmente una coppia esistente. (meccanismo simile alla cache)

7.7.3 Esempio di TLB

Riassumendo la MMU genera l'indirizzo fisico a partire dall'indirizzo logico nel modo seguente:

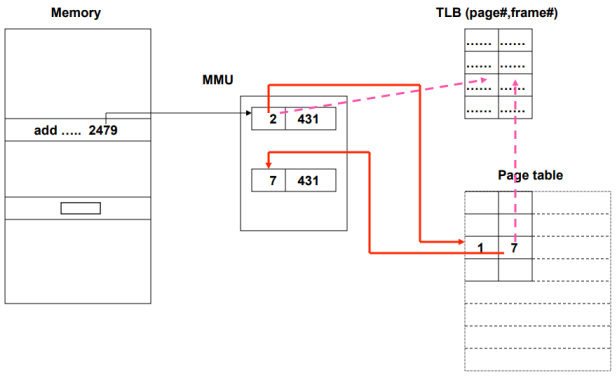
1. Usando la entry del TLB dedicata alla pagina, **se esiste**
2. Sfruttando la entry relativa alla pagina sulla page table, se tale entry ha validity bit 1 e se il caso 1) non è applicabile
3. generando un page fault se i casi 1) e 2) non sono applicabili

Esempio:



- Abbiamo un istruzione il cui operando è un indirizzo di memoria, supponiamo che le nostre pagine siano pagine da 1K, quindi 2479 si trova nella terza pagina.
- Da qualche parte nella RAM si trova la page table, la entry che ha a che fare con questa istruzione è quella con indice numero 2 (0,1,2)
- Assumiamo che la pagina di cui stiamo parlando sia in RAM, quindi sia un page frame e supponiamo che la coppia (2, frame #) sia nel TLB.
- In particolare la pagina due 2 trova nel frame 7, e ha validity bit=1 (è in RAM)
- In questo primo esempio usando i bit più significativi (page number) la MMU accede al TLB, nella speranza che presentando il page number ottenga il frame number
- In questo esempio la cosa accade, la coppia (2, frame number) esiste e quindi il frame number associato (7) viene sostituito al 2

Secondo caso:



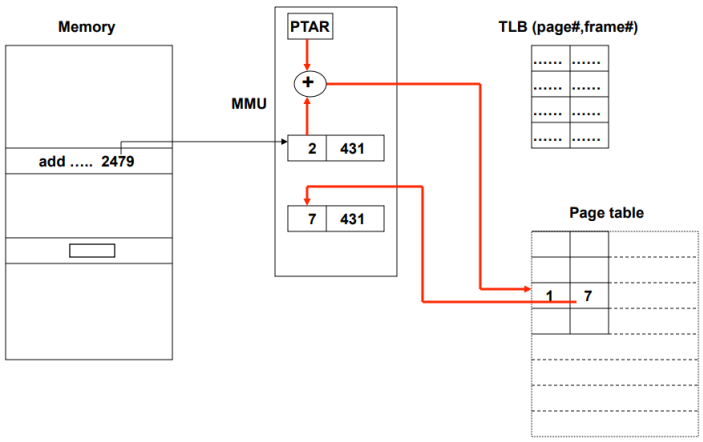
- La coppia (page #, frame #) non esiste nel TLB, tuttavia la pagina è in un frame.
- Dunque la MMU cerca di accedere al TLB, ma la coppia (2, frame #) non esiste, abbiamo dunque un TLB miss.
- La MMU è dunque costretta ad accedere alla page table trovando cosi i bit da sostituire al page #. (sostituisco 7 al 2)
- Oltre ad effettuare la sostituzione viene popolato il TLB con questa nuova entry (meccanismo simile alla cache)

Alcune osservazioni:

- quando avviene un **context switch**, il TLB va azzerato per evitare che il processo schedulato acceda alla memoria del precesso che ha perso il processore (il TLB è unico per tutti i processi);
- il processo schedulato avrà il TLB "vuoto": questo è un elemento che introduce **overhead**, poichè bisogna spostare dei dati

Chiariamo alcune imprecisioni:

- La MMU non accede a una generica page table, ma accede alla page table del proceso running
- in memoria sono presenti le page table di tutti i processi
- L'architettura può offrire il Page Table Address Register (**PTAR**), **contenente l'indirizzo della page table del processo in esecuzione** in quel momento
- Il valore da assegnare al PTAR è memorizzato dal PCB del processo



Il page number 2(offset) viene sommato dalla MMU al PTAR(indirizzo page table)

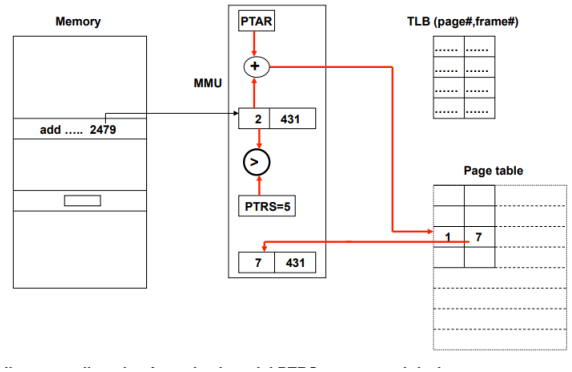
Indirizzo al quale accedere = valore del PTAR + dimensione delle entry della page table × numero pagina

7.7.4 Protezione della memoria

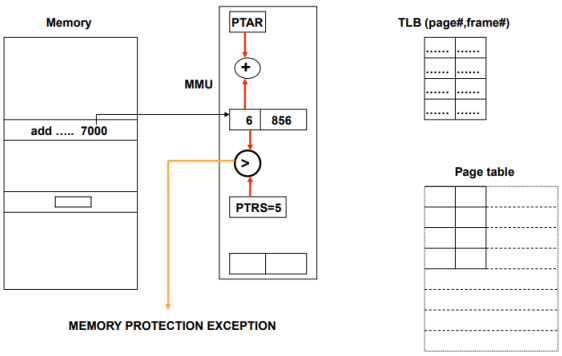
se un processo tenta di accedere ad un indirizzo al di fuori del proprio spazio logico, viene generata un **memory protection exception**

- questo si verifica quando gli **l** bit leftmost dell'indirizzo logico hanno un valore k e la pagina k non esiste, in quanto il processo ha pagine 0,1,. . . ,h con $h < k$.
  In pratica un processo cerca accedere alla pagina numero 18, quando ne esistono 15 (esempio)

- il controllo può essere fatto facilmente se l'architettura offre un Page Table Size Register (**PTSR**) che **memorizza il numero dell'ultima pagina** del processo running. Esso deve essere memorizzato nel PCB del processo.



In questo caso il numero di pagina è $\leq$ al valore del PTRS, dunque non si è verificata alcuna violazione.



In questo caso Il numero di pagina è maggiore del PTRS: la pagina non esiste, viene sollevata l'eccezione.

Altre osservazioni:

- il campo prot della page table codifica le protezioni in lettura/ scrittura delle pagine
- se l'architettura fornisce un adeguato supporto, il valore del campo può essere confrontato con il tipo di accesso che viene effettuato
- se il tipo di accesso non è consentito, la MMU genera una memory protection exception

Principio di località

un indirizzo logico generato eseguendo un'istruzione ha probabilità elevata di essere vicino agli indirizzi logici generati eseguendo le istruzioni più recenti.

- le istruzioni di branch sono solitamente non più del 10-20del totale, quindi i processi eseguono prevalentemente istruzioni memorizzate l'una adiacente all'altra;
- gli accessi alle strutture dati sono spesso effettuati in istruzioni ravvicinate (array, cicli, ...)

Dunque per minimizzare i page fault,quando è necessario effettuare lo swap out di una pagina conviene scegliere tra le pagine che sono state visitate meno di recente

Prima osservazione:
Aumentando i page frame allocati ad ogni processo:

- diminuisce la probabilità di avere page fault (pochi processi in RAM, ognuno con una porzione significativa di essa, + efficienza)
- si hanno meno processi in memoria (- efficienza)

Bisogna quindi trovare un equilibrio.
Se il numero di page frame allocati ad un processo è troppo basso:

- i page fault sono frequenti
- cresce il page I/O
- cresce quindi il numero di context switch

Si parla di **trashing** in questo caso

Seconda osservazione:
Diminuendo la dimensione s delle pagine:

- La memoria sprecata, mediamente *s/2*, diminuisce. Infatti tipicamente ogni processo spreca metà delle pagine in media.
- il numero di page fault diminuisce, a parità di memoria allocata al processo. Infatti è più probabile generare un indirizzo che si trovi in una pagina allocata in un page frame (principio di località)
- Il numero di entry nelle page table è maggiore
- La dimensione del frame # è maggiore
- Lo swap device perde efficienza se usa blocchi di dimensione ridotta. (Conviene recuperare una quantità di dimensione maggiore quando si usa l'hard-disk)

La page table ha due campi che servono per il page replacement:

- **mod**(modifica): si tratta di un semplice bit
  - impostato a 0 al momento del page in (quando viene caricata sulla RAM)
  - aggiornato a 1 quando il page frame viene modificato. Quando il page frame viene liberato, se il bit mod vale 1 allora è necessario il page out.
- ref: anche questo è un semplice bit
  - impostato ad 1 al momento del page in
  - aggiornato a 0 ad intervalli regolari (reset del tempo, cosi vediamo se da questo momento in poi le pagine vengono visitate)
  - aggiornato a 1 quando la pagina viene visitata

  Se ref vale 1 allora il page frame è stato visitato di recente, ed in base al principio di località non è un buon candidato ad essere liberato se servono page frame liberi.

7.7.5 Dettagli implementativi del paging

Superpagine
Chiamiamo **TLB reach** il prodotto page size × TLB entries. Ossia la **memoria coperta dal TLB**
La dimensione delle RAM cresce rapidamente, la dimensione delle memorie associative (TLB) no: il rapporto TLB reach/capacità RAM tende a diminuire, dunque **il numero di TLB miss tende ad aumentare**, con due conseguenze:

- Gli accessi alla memoria che sfruttano il TLB sono pochi

- Gli accessi alla cache sono rallentati dai TLB miss e dalla necessità di accedere alla page table in memoria.

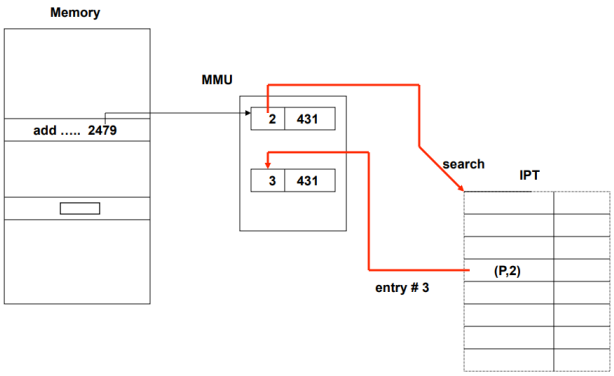
Soluzione possibile: uso delle **superpagine**.

- Una superpagina è come una pagina, però ha dimensione pari a $2^n \times$ dimensione delle pagine, per un **n** opportuno.
- Più pagine contigue con accessi frequenti vengono promosse a superpagina dal S.O.. Parliamo di **promotion**.
- Una TLB entry può riferisi ad una pagina o ad una superpagina, se una TLB entry si riferisce a una superpagina, il TLB reach si alza.
- Se alcune pagine della superpagina non hanno più accessi, la superpagina viene smembrata. Parliamo di **demotion**.

Organizzazione delle page table.
Se i processi hanno page table di dimensioni ampie, una porzione significativa di memoria viene usata per gestire la memoria virtuale e non per mantenere lo spazio indirizzabile dei processi. Per risparmiare memoria esistono due strategie:

- uso delle **Inverted Page Table (IPT)**:
  L'IPT ha una entry per ogny frame, con due campi:
  - l'indicazione che il frame è libero, oppure
  - la coppia (process id, page #) se il frame è occupato

  Esempio:



  pagina 2, indirizzo 431. Non abbiamo come nella tabella originale il frame number che ospita la pagina, ma una entry che corrisponde a quel frame con (processo e pagina)

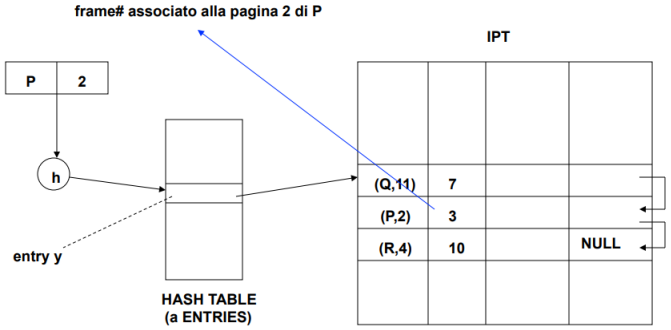
Se la coppia non viene trovata abbiamo un PAGE FAULT.

Per evitare la ricerca sull'IPT si può utilizzare una **funzione hash**:

- ogni entry dell'IPT ha almeno 4 campi
  - stato libero/occupato
  - coppia (process id, page #);
  - frame #, che non è un indice, come invece accadeva nel caso precedente;
  - pointer, per costruire una linked list.
- si assume un numero primo **a** maggiore del numero di frame
- si assume una hash table contenente a pointers verso l'IPT
- si considera la funzione hash h: $x \to x\ MOD\ a$ NB. se $h(x) = y$ allora $0 \leq y \leq a-1$, ed esiste la entry numero y nella hash table

  - Concatenando un process id **P** ed un page # p otteniamo una stringa di bit interpretabile come intero **x** che sarà argomento della funzione **h**
  - Per ogni $0 \leq y \leq a-1$, le coppie (Q, q) che appaiono nell'IPT e tali che $h(Qq) = y$ **sono concatenate in una linked**
  - Per ogni $0 \leq y \leq a-1$ la entry y della hash table punta alla prima entry dell'IPT contenente una coppia (Q,q) tale che $h(Qq) = y$

  Esempio:



  - Supponiamo di voler tradurre pagina due del processo P
  - Applico la funzione di hash
  - Supponiamo che il risultato sia la entry y, dove abbiamo un puntatore alla inverted page table (IPT)
  - Anzichè ispezionare tutta la IPT, parto dal punto individuato dalla hash table

    - se trovo in questo punto la coppia sono a posto, altrimenti scandisco la linked list.

- doppia paginazione