

Sistemi Operativi Unità 6: La memoria

Organizzazione delle memoria

Martino Trevisan
Università di Trieste
Dipartimento di Ingegneria e Architettura

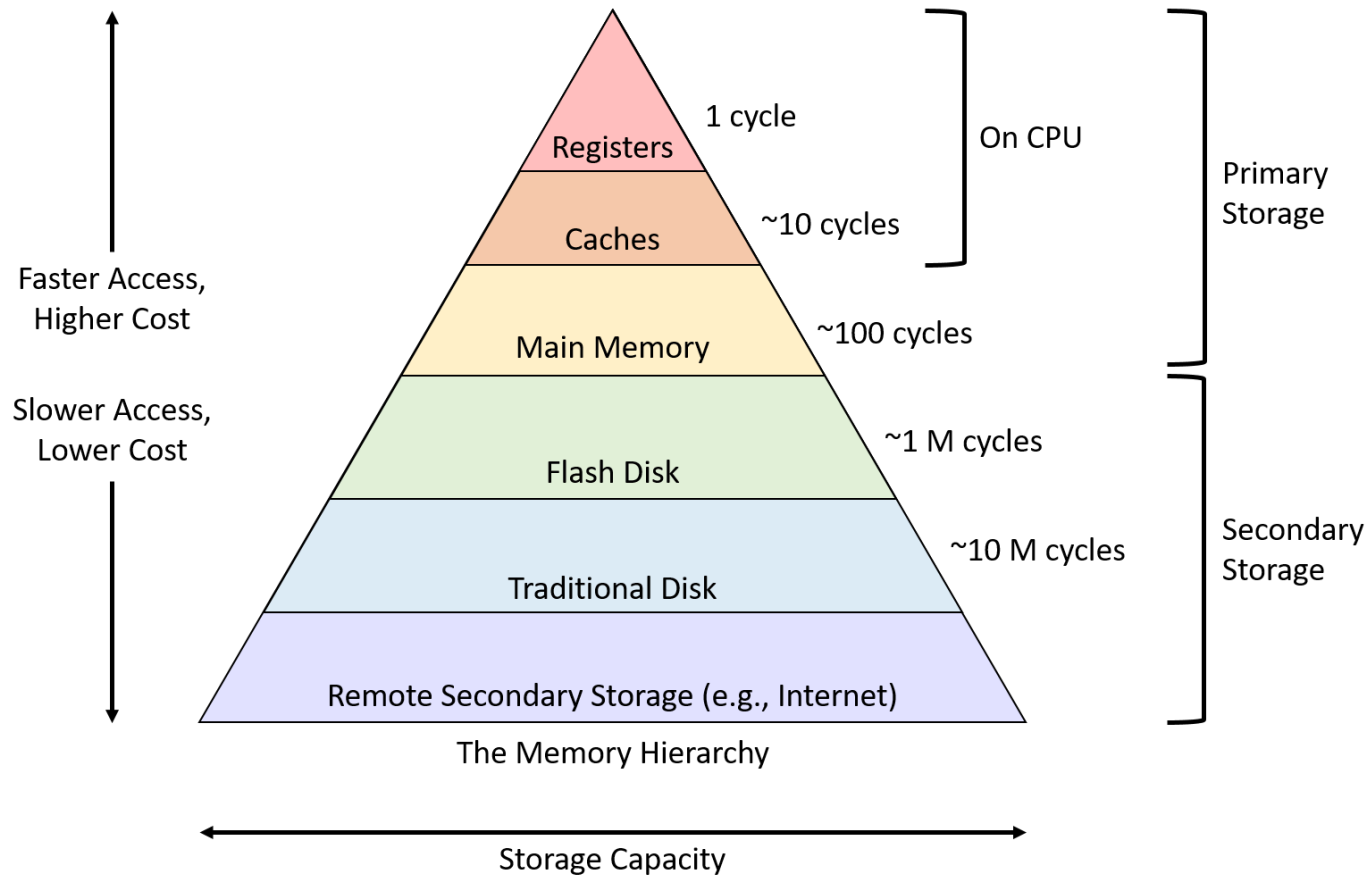
Argomenti

1. Memoria nei sistemi ad processore
2. Approcci storici
3. La memoria virtuale
4. Rimpiazzamento delle pagine
5. Layout della memoria
6. Loader, librerie e pagine condivise
7. Gestione della memoria in Bash

Memoria nei sistemi ad processore

Memoria nei sistemi ad processore

I sistemi ad processore possiedono molte *memorie*



Memoria nei sistemi ad processore

Il compito del SO è gestire l'utilizzo della memoria da parte dei processi, con gli obbiettivi di:

- Alte prestazioni: usare la memoria più veloce possibile
- Isolamento tra processi: evitare problemi di sicurezza e stabilità
- Facilita per il programmatore: si vorrebbe che il SO fosse *trasparente* per chi programma

Approcci storici

Approcci storici

Inizialmente, non vi era SO: l'elaboratore eseguiva un programma per volta

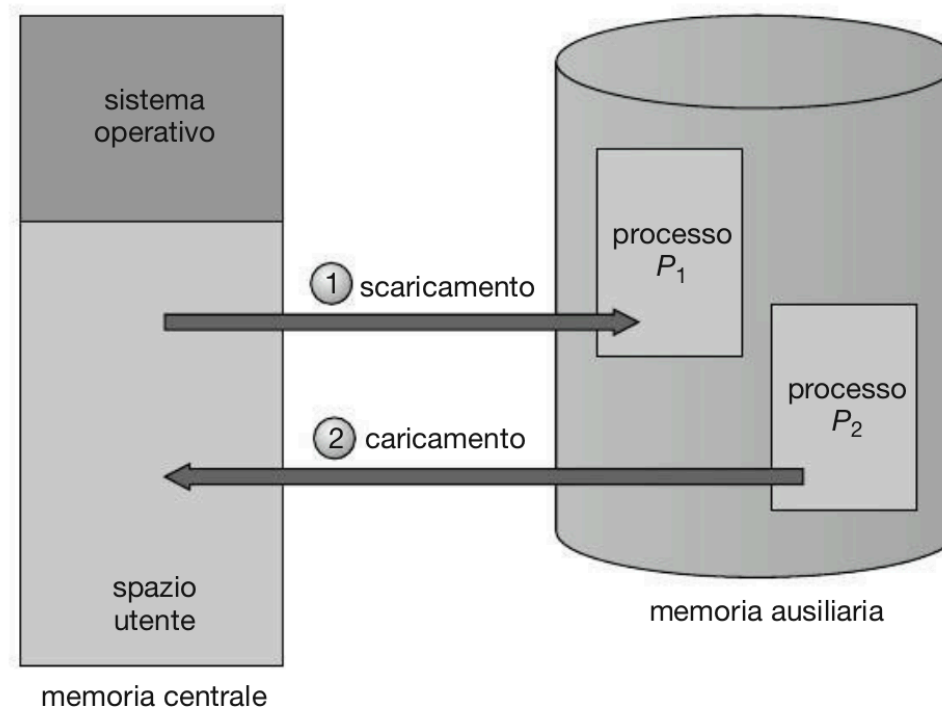
- Un programma poteva accedere a qualsiasi locazione di memoria

Nota: ancora non esisteva la memoria virtuale e la MMU, ecc...

Approcci storici

Approccio a Sostituzione totale

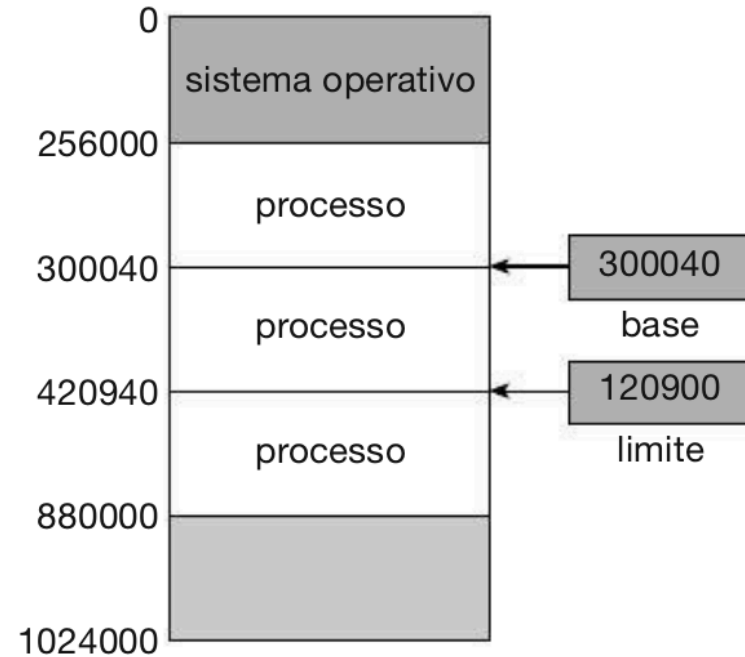
Un SO che sostituisce completamente la memoria principale del programma in esecuzione ad ogni *Context Switching*



Approcci storici

Approccio Base e Limit

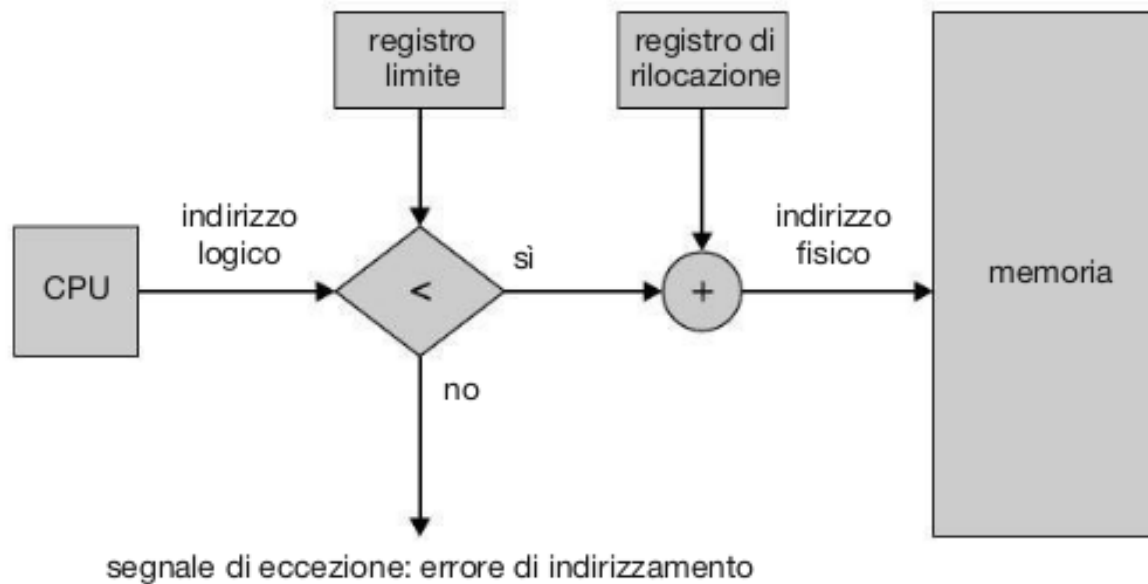
- Più processi condividono la memoria
- Hanno il permesso di accedere a una sola zona di memoria



Approcci storici

Approccio Base e Limit

- Nasce il concetto di **Indirizzo Logico** o **Virtuale**
- Una (rozza) MMU traduce usando i registri *Base* e *Limit*, settati dal SO
- I processi possono accedere solo alla loro fetta di memoria fisica



Approcci storici

Approccio Base e Limit

Pro:

- Permette di avere più processi
- Sicuro: un processo non può accedere a memoria di altri

Contro:

- Allocazione contigua: si rischia spreco di memoria
- Poca flessibilità

Approcci storici

Approccio a memoria segmentata

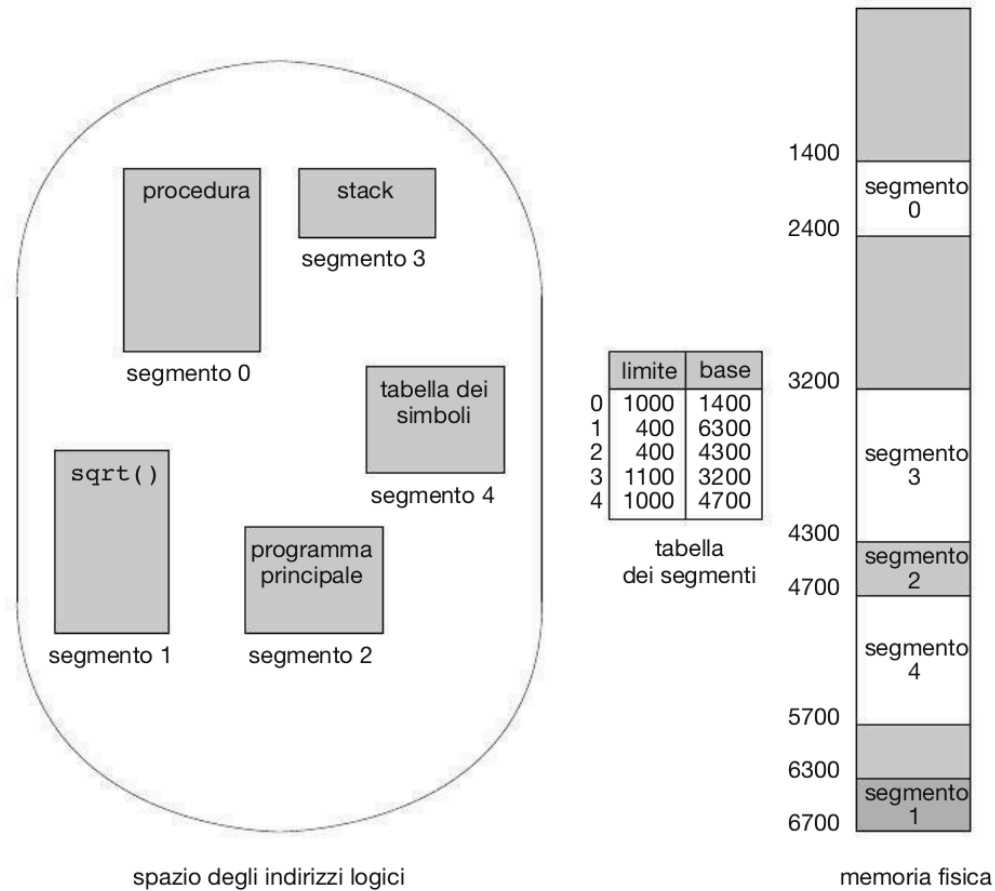
Come Base + Limit, ma ogni processo ha a disposizione più **segmenti**

Solitamente, ogni segmento ha scopi diversi:

- Segmento di Codice
- Segmento di Dati (Variabili Globali e Costanti)
- Segmento di Stack (Variabili di funzioni)

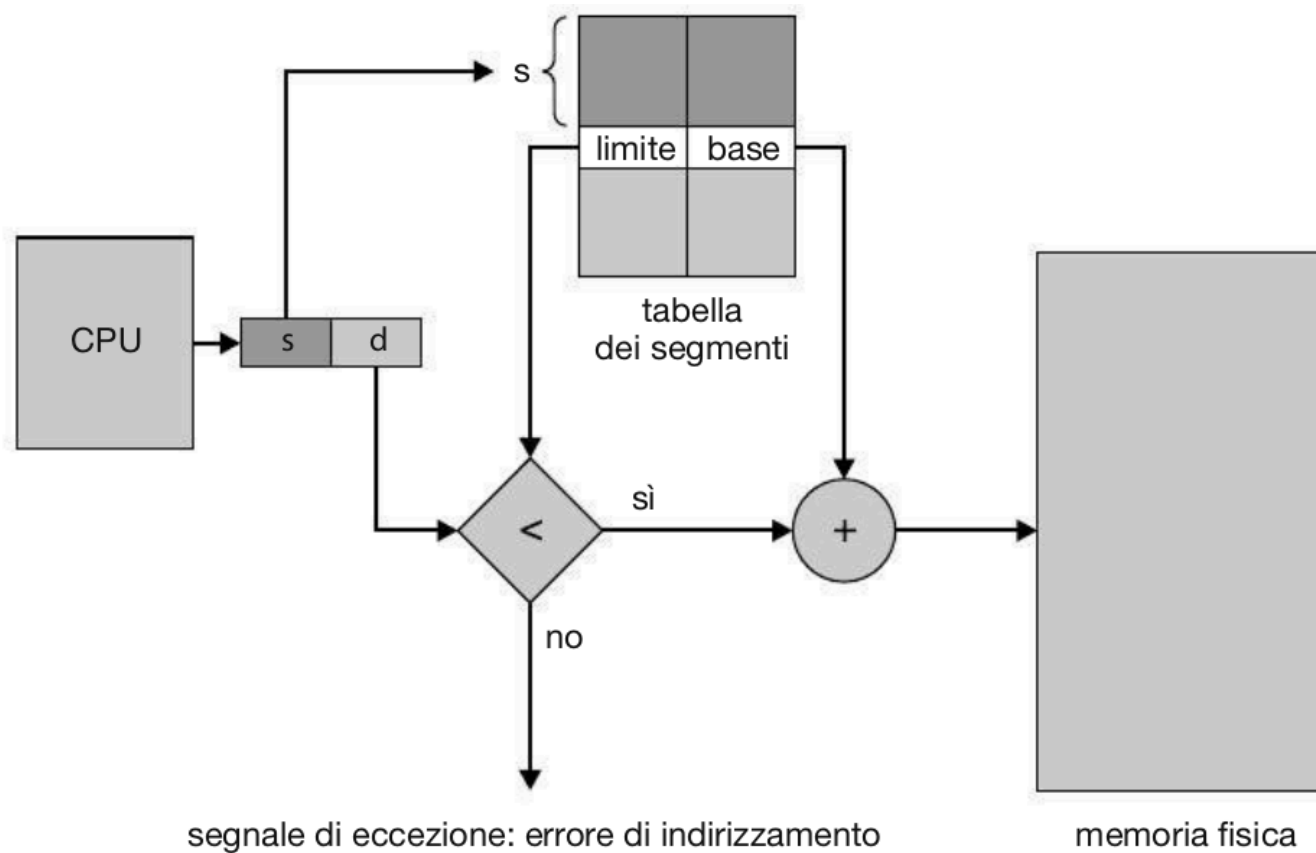
Approcci storici

Approccio a memoria segmentata



Approcci storici

Approccio a memoria segmentata



Approcci storici

Approccio a memoria segmentata

Pro:

- Abbastanza flessibile
- Semplice da implementare
- Veloce

Contro:

- Segmenti di lunghezza diversa sono problematici da gestire
- Introducono frammentazione come in *Base + Limit*

E' stato comunque molto usato negli anni '80 e '90

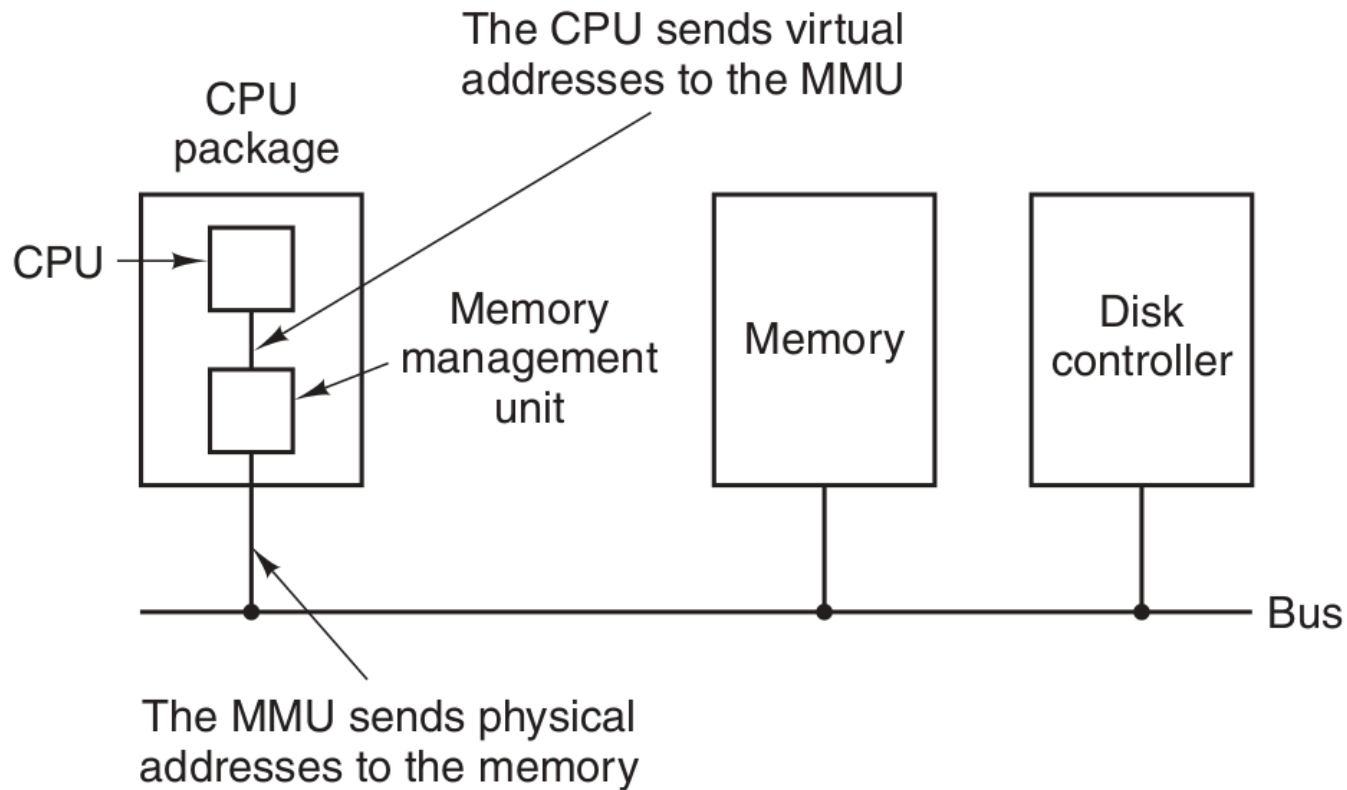
Approcci storici

Approccio a paginazione

- La memoria fisica è organizzata in **Pagine** di dimensione fissa
- Un processo emette **Indirizzi Virtuali** (sempre organizzati in pagine)
- La **Memory Management Unit** li traduce in **Indirizzi Fisici**
- Usando una **Tabella** che mappa pagine dallo spazio virtuale a quello fisico

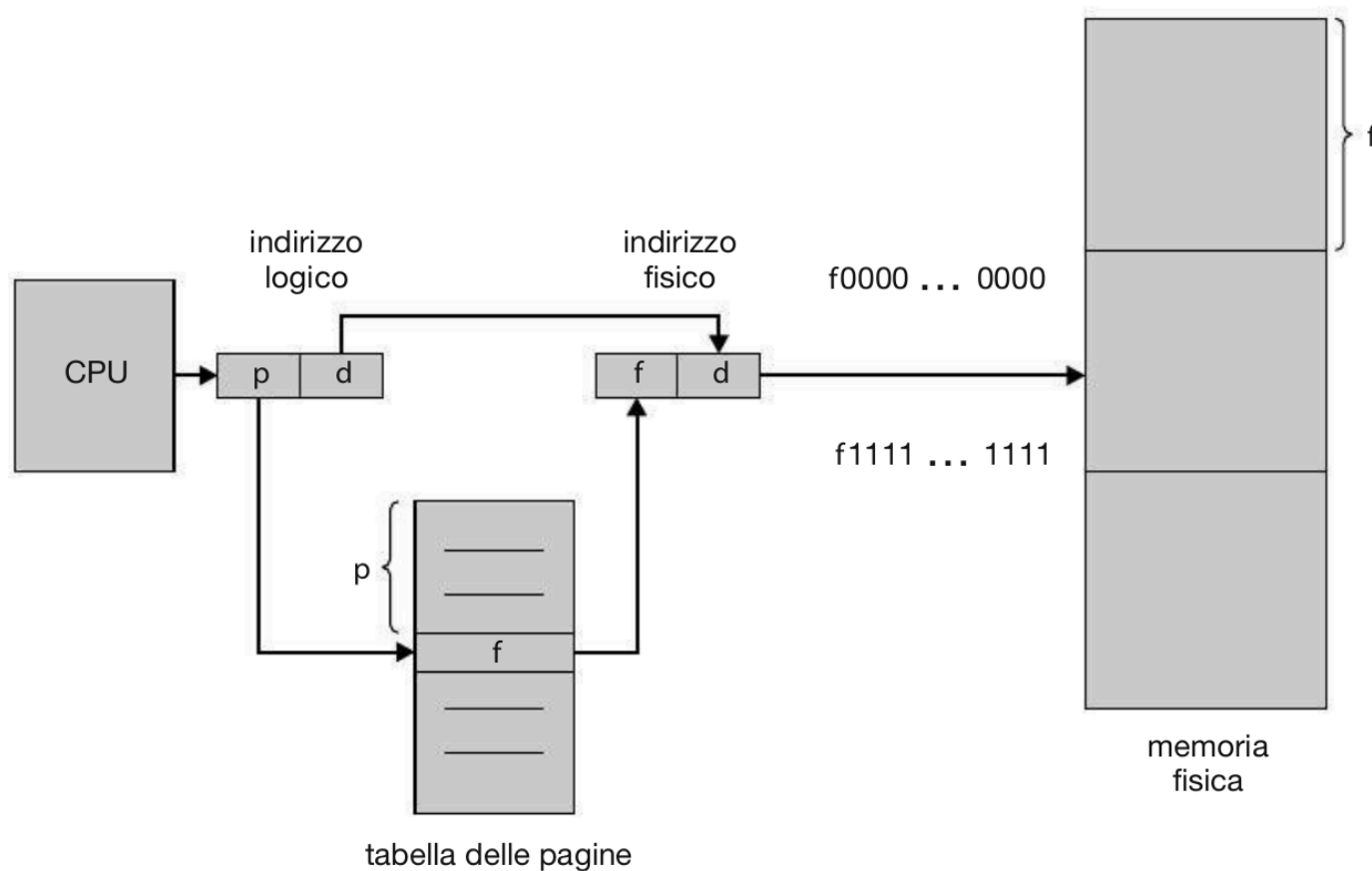
Approcci storici

Approccio a paginazione



Approcci storici

Approccio a paginazione



Approcci storici

Approccio a paginazione

- Il processo vede uno spazio virtuale che è diviso in pagine di grandezza dimensione fissa
- La tabella delle pagine indica come sono state allocate nella memoria fisica
- La MMU effettua la traduzione
- Il SO imposta/programma la MMU

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

memoria logica

0	5
1	6
2	1
3	2

tabella
delle pagine

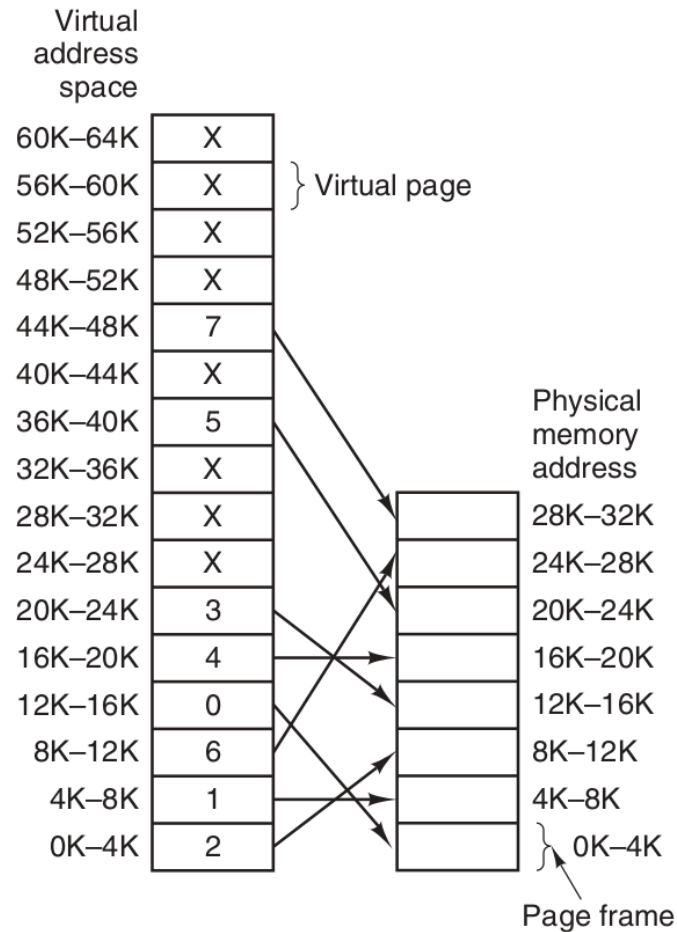
0	
4	i j k l
8	m n o p
12	
16	
20	a b c d
24	e f g h
28	

memoria fisica

Approcci storici

Approccio a paginazione

- Tipicamente lo spazio degli indirizzi virtuali è più grande di quello degli indirizzi fisici
- **Esempio:** Il programma può emettere indirizzi su 1024 pagine, ma in memoria ce ne stanno solo 80
- Il programmatore non deve sapere quanta memoria ha il sistema



Approcci storici

Approccio a paginazione

Pro:

- No frammentazione
- Flessibile

Contro:

- Richiede un Hardware veloce, che assiste tutto il processo

E' lo standard *de-facto*

- Utilizzato in tutti i moderni processori e SO

La memoria virtuale

La memoria virtuale

Ogni processo vive in uno spazio di indirizzi virtuale, mappato sulla memoria fisica

- E' il naturale effetto della memoria paginata

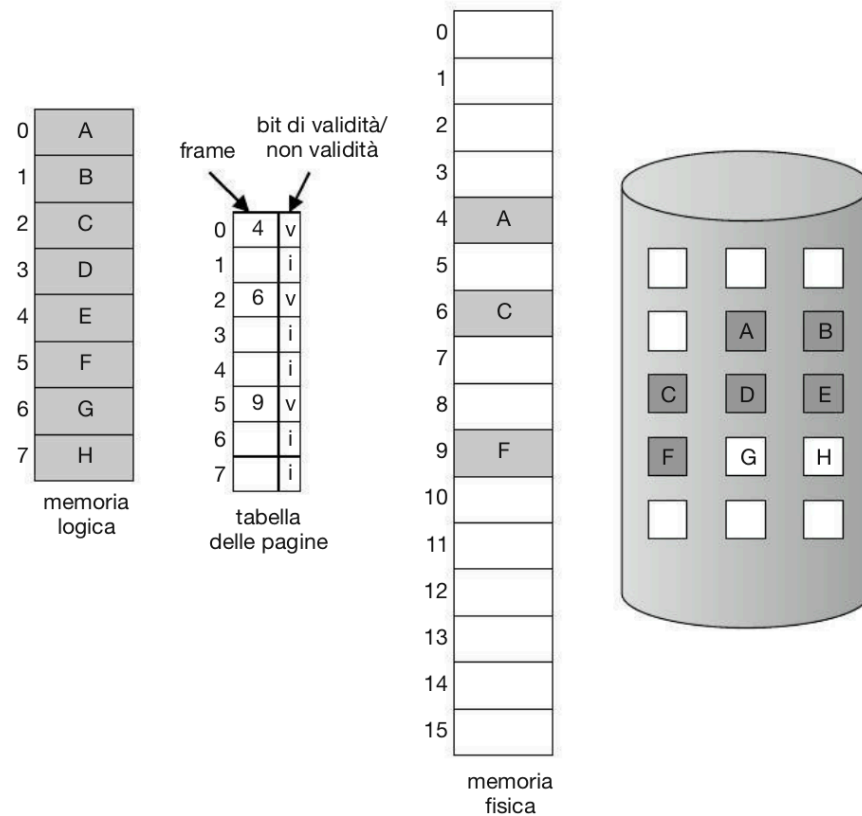
Architettura x86-64:

- Indirizzi Virtuali su 64bit, ma solo 48 utilizzati
 - Memoria virtuale di 256TB
- Pagine da 4KB
 - Offset da 12 bit
- La MMU traduce ad indirizzi di 48bit
 - Ma la memoria fisica è sempre **molto** più piccola

La memoria virtuale

Se le pagine non stanno tutte in memoria, si mettono su disco.

Lo spazio su disco che contiene le pagine non in memoria si chiama **Swap**

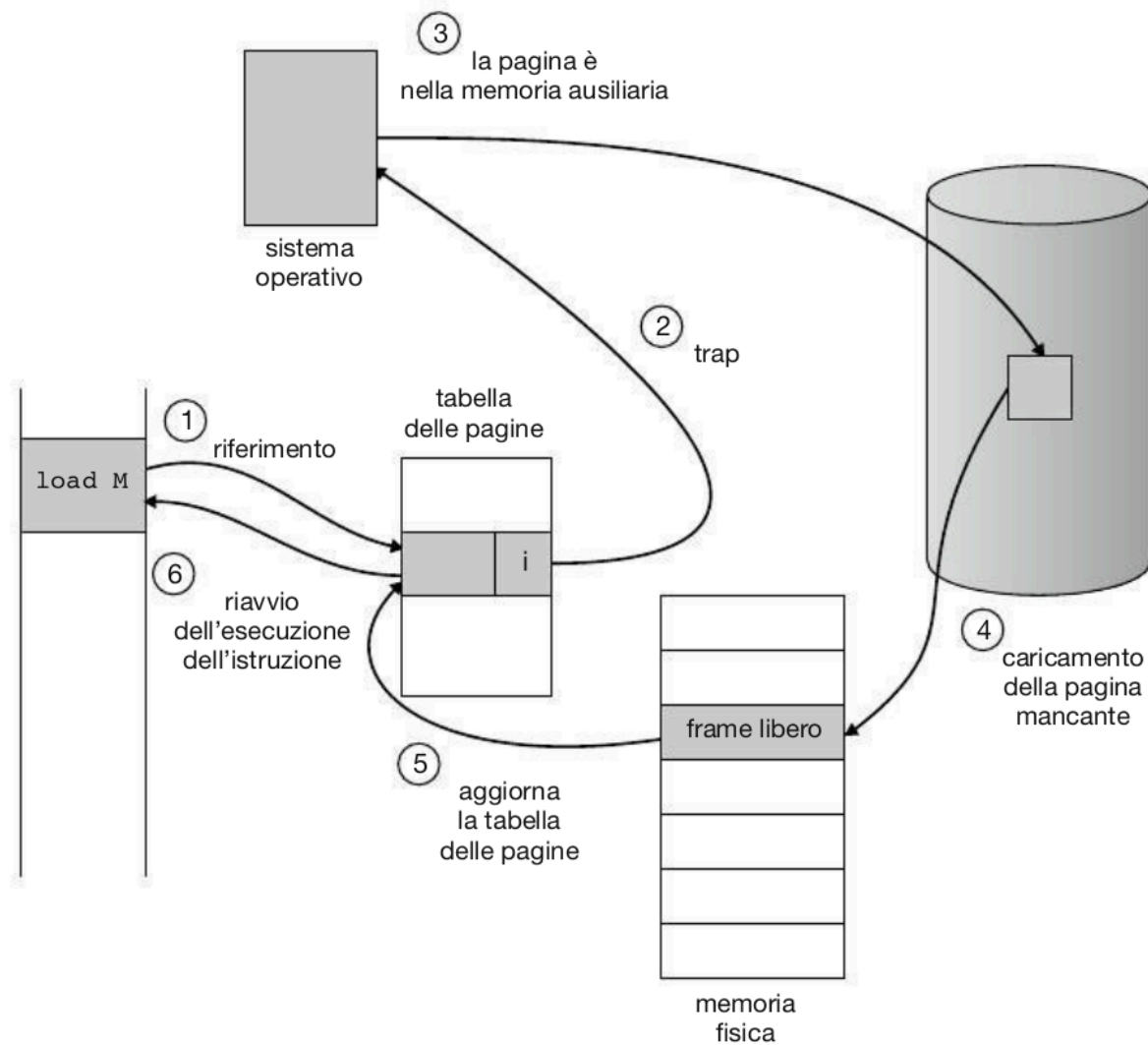


La memoria virtuale

Se il processo emette un indirizzo di una pagina che non è in memoria, si verifica un **Page Fault**:

- La MMU avverte il SO
- Il SO interrompe il processo
- Il SO carica la pagina (o la crea) da disco
- Il SO reimposta la MMU
- Il processo riprende

La memoria virtuale



Rimpiazzamento delle pagine

Rimpiazzamento delle pagine

La memoria fisica è sempre più piccola di quella virtuale

Se è piena di pagine utilizzate da processi attivi, il SO deve scegliere quale pagine eliminare e salvare su disco

Esistono diversi algoritmi di rimpiazzamento per effettuare ciò in maniera furba

Rimpiazzamento delle pagine

Algoritmo FIFO

Rimuovo dalla memoria la pagina caricata da più tempo

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7	7	7	2
	0	0	0
		1	1

2	2	4	4	4	0
3	3	3	2	2	2
1	0	0	0	3	3

0	0
1	1
3	2

7	7	7
1	0	0
2	2	1

Semplice, ma inefficace

Rimpiazzamento delle pagine

Algoritmo Ottimo

Rimuovo la pagina che non mi servirà per più tempo **nel futuro**

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



frame delle pagine

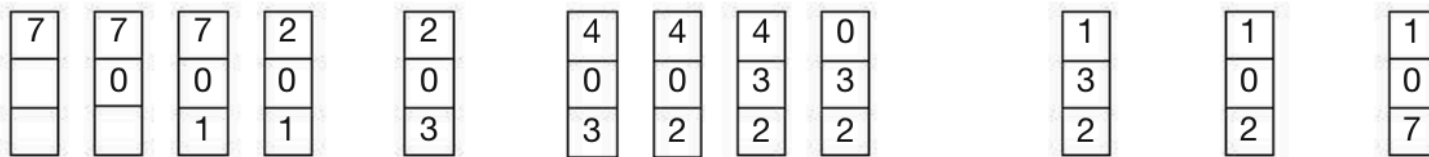
Ottimo, ma impossibile prevedere il futuro

Rimpiazzamento delle pagine

Algoritmo Last Recently Used

Rimuovo la pagina che non viene usata da più tempo

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



frame delle pagine

Semplice, efficace.

Serve collaborazione della MMU per tenere traccia di accessi

Usato quasi sempre (con varianti)

Algoritmo Least Frequently Used

Teoricamente meglio, difficile da realizzare

Layout della memoria

Layout della memoria

Un processo può accedere a qualsiasi locazione di memoria nello spazio degli indirizzi virtuali

- Lo spazio degli indirizzi virtuali è diviso in pagine
- Se la pagina è in memoria, la MMU traduce in indirizzo fisico
- Se la pagina non è in memoria, il SO la creerà/preleverà da disco

Layout della memoria

Un programma che accede a indirizzi "casuali" non è efficiente

- Utilizzo di pagine e memoria sarebbe molto penalizzato

Un processo inizia a usare indirizzi a partire da quelli "bassi":

- Si inizia a utilizzare indirizzo `00 00 00 00`, poi `00 00 00 01`
- Così si riempie una pagina completamente, poi se inizia a usare un'altra

Ci sono diverse **convenzioni**, che dipendono da architettura dell'elaboratore e OS. Noi vediamo **Linux** e **x86**.

- Implementate **dal compilatore**

Layout della memoria

Attualmente, si usano sia indirizzi all'estremo alto che all'estremo basso

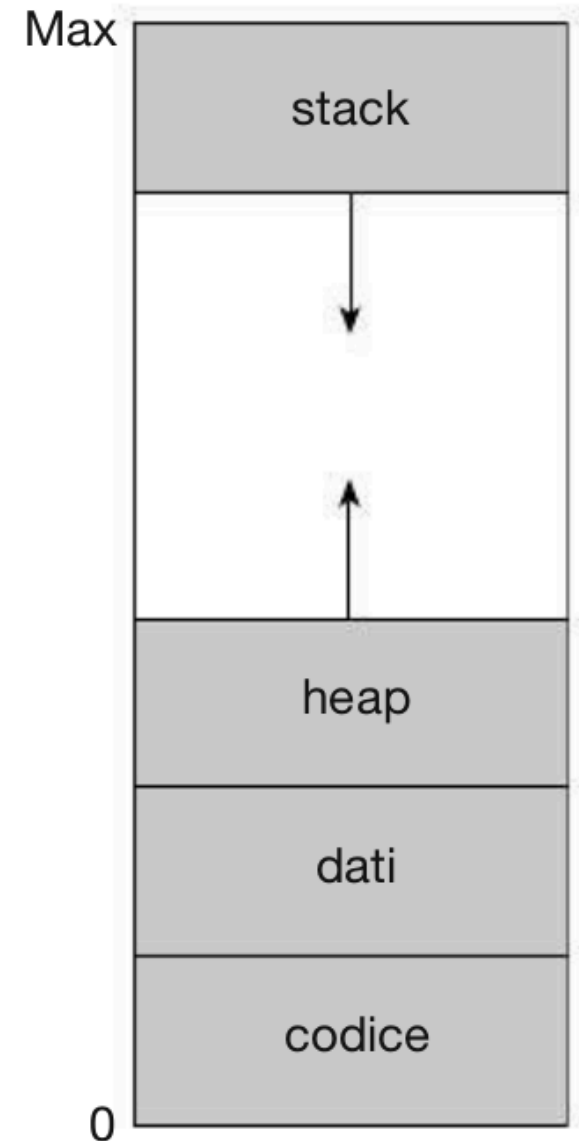
- La memoria può crescere in due direzioni
 - Posso avere due zone di memoria che crescono a seconda dell'esigenza del programmatore
1. **Heap:** cresce dal basso verso l'alto. Usato dal programmatore per allocare memoria quando gli serve
 2. **Stack:** cresce dall'alto verso il basso. Usato dal compilatore per posizionare le variabili delle funzioni

Layout della memoria

Un processo può accedere a qualsiasi locazione di memoria.
Per convenzione e prestazioni si preferisce iniziare gli estremi

Ci sono 4 zone di memoria:

- Codice
- Dati
- Stack
- Heap



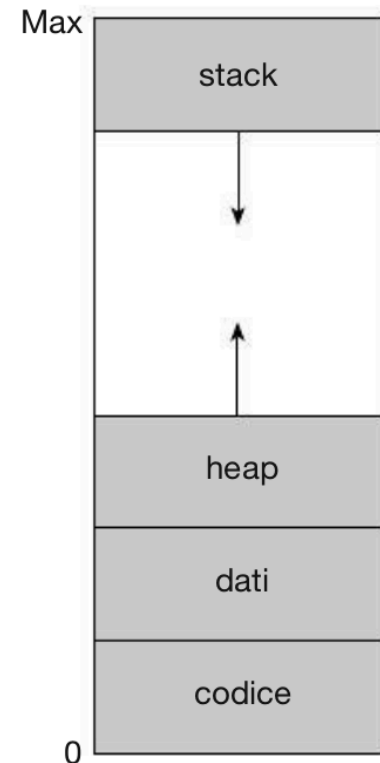
Layout della memoria

Codice

Il SO copia il codice del programma dal disco verso gli indirizzi più bassi

- Il codice deve obbligatoriamente trovarsi in memoria
- Il registro *Program Counter* della CPU punta a un indirizzo in questo range

Questa parte della memoria è *Read Only*: un programma non può modificare se stesso

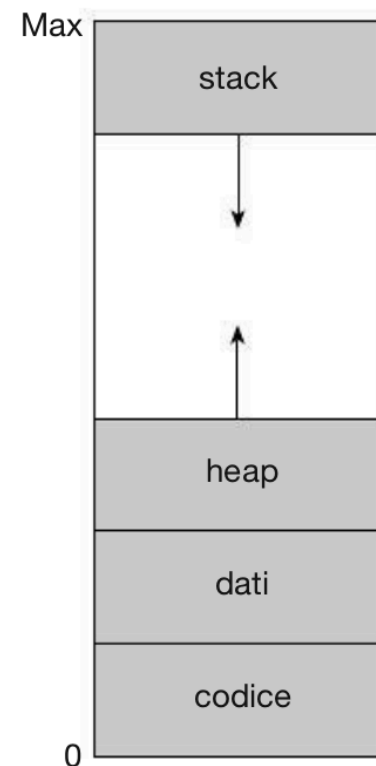


Layout della memoria

Dati

Gli indirizzi immediatamente *maggiori* del codice, sono usati per le variabili globali

- Il compilatore usa questi indirizzi per le variabili globali
- Le variabili globali *inizializzate* vengono riempite direttamente dal SO quando viene avviato il processo
- Le altre contengono tutti `'\0'`

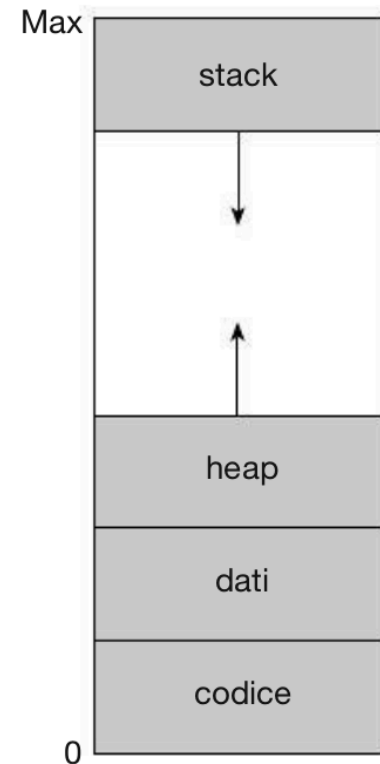


Layout della memoria

Heap

Usato per la **Memoria Dinamica**

- Il programmatore può aver bisogno di memoria la cui dimensione non viene prevista in fase di programmazione
- Gestita tramite le funzioni di libreria *malloc* e *free*
- Vedremo in seguito



Layout della memoria

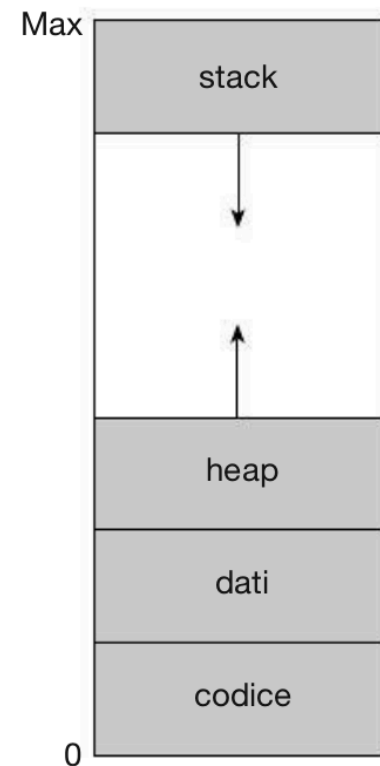
Stack

Usato per le variabili relative a funzioni:
argomenti e variabili interne

- Come dice, questa zona è gestita come se fosse una **pila**

I dati vengono:

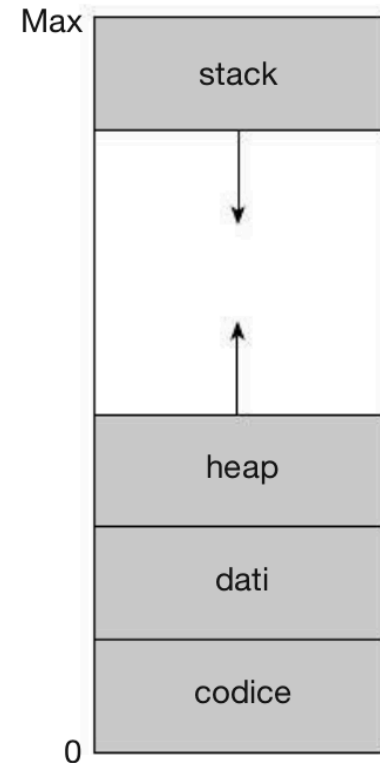
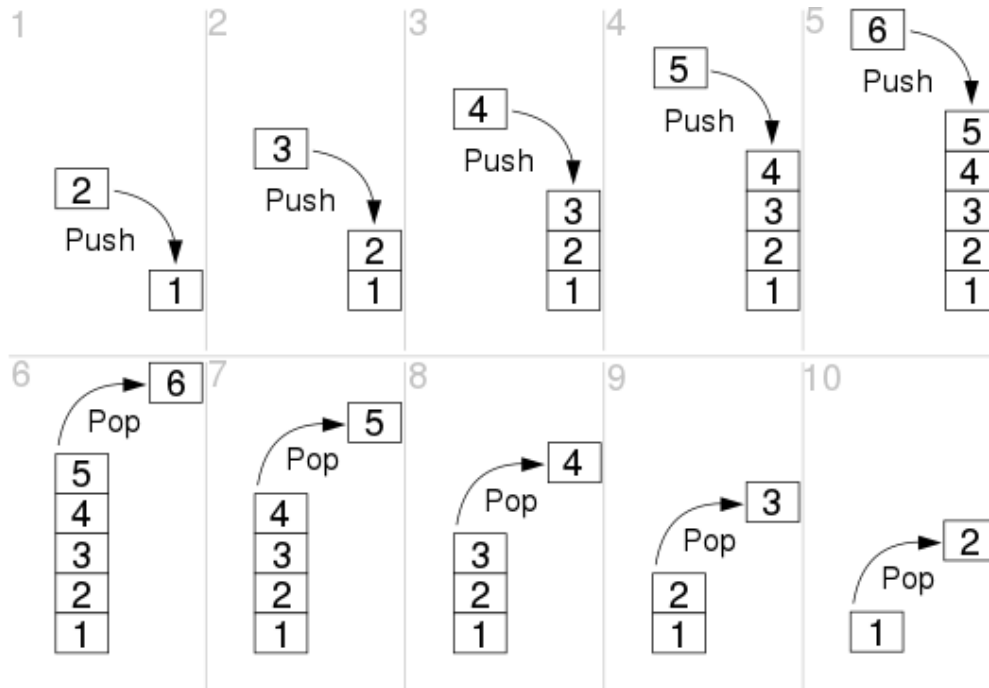
- Impilati per essere aggiunti: **Push**
- Tolti dalla pila quando devono essere usati:
Pop



Layout della memoria

Stack

I dati vengono aggiunti e tolti dalla cima della pila



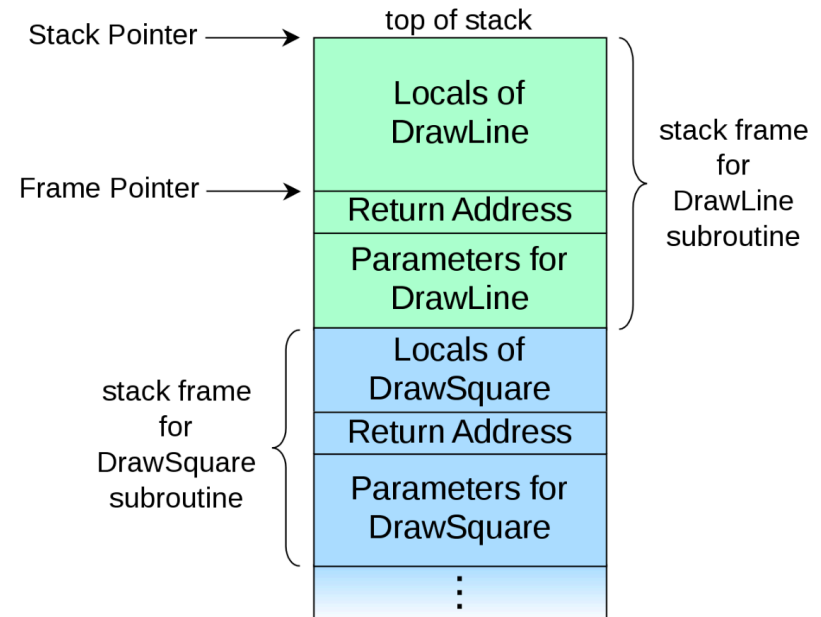
Layout della memoria

Stack

Concetto pratico per gestire le funzioni!
Quando viene chiamata una funzione, si aggiunge un blocco allo stack contente (**Push**):

- Indirizzo di ritorno
- Parametri
- Variabili locali

Quando la funzione ritorna, il blocco si elimina (**Pop**)



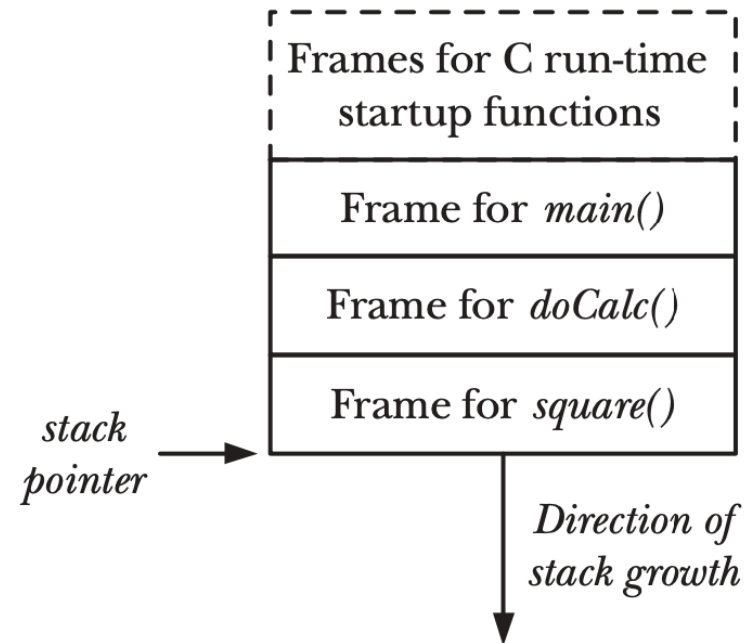
Layout della memoria

Stack

Ogni blocco si chiama **Stack**

Frame

- Creato quando la funzione viene invocata
- Cancellato quando la funzione ritorna



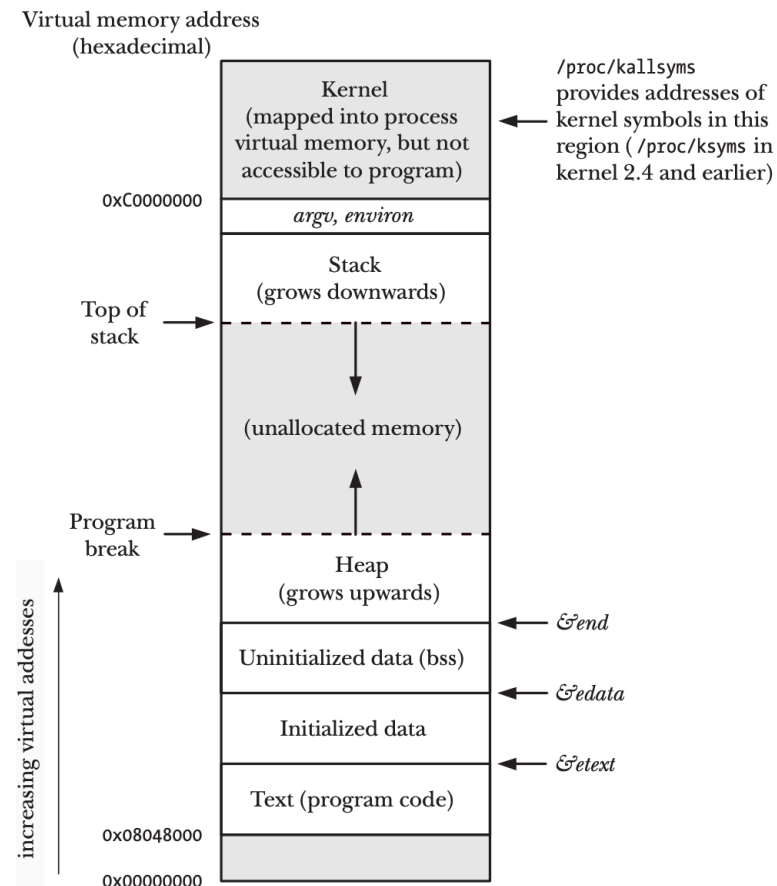
Layout della memoria

In Linux

Il layout visto prima é semplificato.

In Linux, precisiamo che:

- Un segmento per variabili globali inizializzate e uno per quelle non inizializzate
- `argc` e `argv` in indirizzi alti
- **Program Break**: indirizzo massimo per parte bassa
- **Top of Stack**: indirizzo minimo per parte alta



Loader, librerie e pagine condivise

Loader, librerie e pagine condivise

Il **Loader** è il componente del SO che avvia i processi.

I suoi compiti sono:

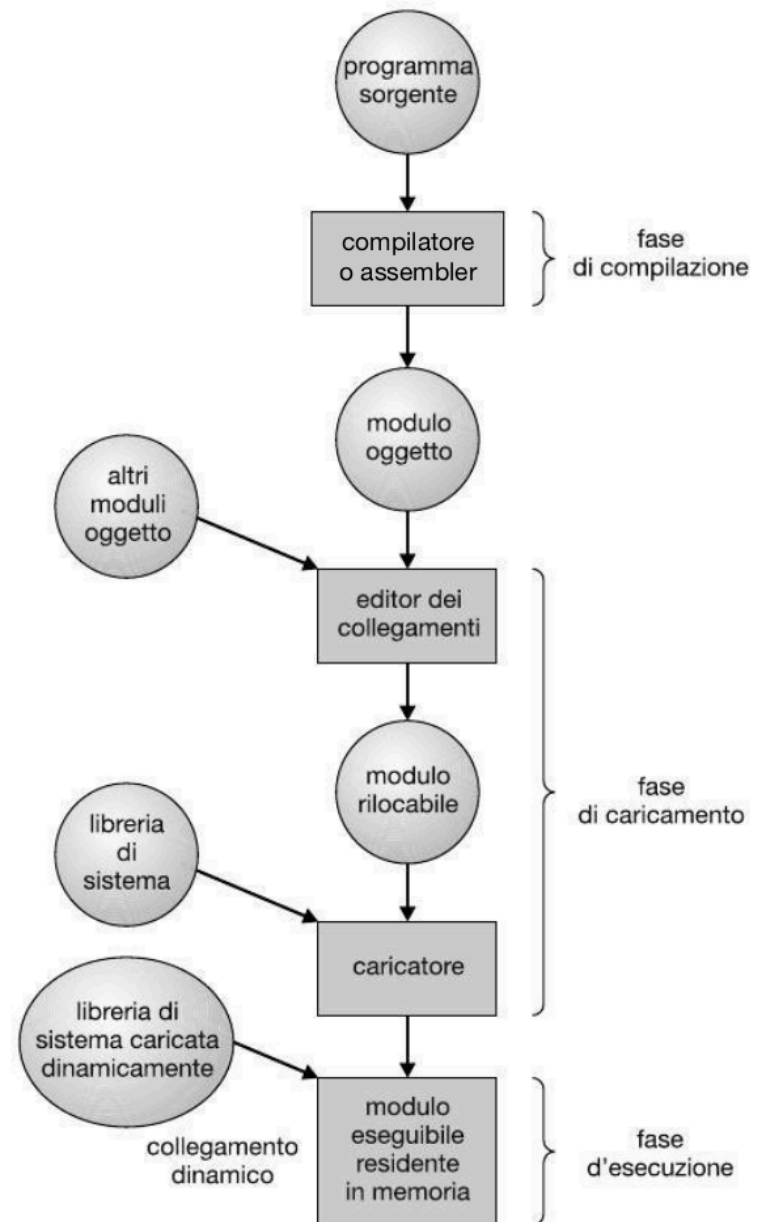
- Verificare che l'utente abbia i permessi
- Copia il codice del programma e le variabili globali inizializzate in memoria
- Caricare le librerie condivise
- Valorizzare `argc` e `argv`
- Avviare il processo dal `main` impostando lo scheduler del SO

Loader, librerie e pagine condivise

Il compilatore crea il codice macchina

Il loader lo carica e ne avvia l'esecuzione

- Deve caricare anche le librerie di sistema, se sono usate



Loader, librerie e pagine condivise

I programmi possono usare **librerie condivise**:

- Offerte dal SO per facilitare la chiamata a System Call
- Installate da utente per scopi particolari (e.g., trigonometria)

Le librerie sono codice che gira in *User Mode*

- Non hanno alcun privilegio rispetto al codice utente
- **Non** sono parte del Kernel

Loader, librerie e pagine condivise

Le **librerie condivise** sono codice eseguibile in cartelle predefinite del sistema

In Linux:

- `/lib`
- `/usr/lib`
- Directory elencate nel file `/etc/ld.so.conf`

In Windows:

- `C:\Windows\SYSTEM32`
- Cartella corrente

Loader, librerie e pagine condivise

Gli eseguibili in Linux sono in formato ELF (*Executable and Linking Format*)

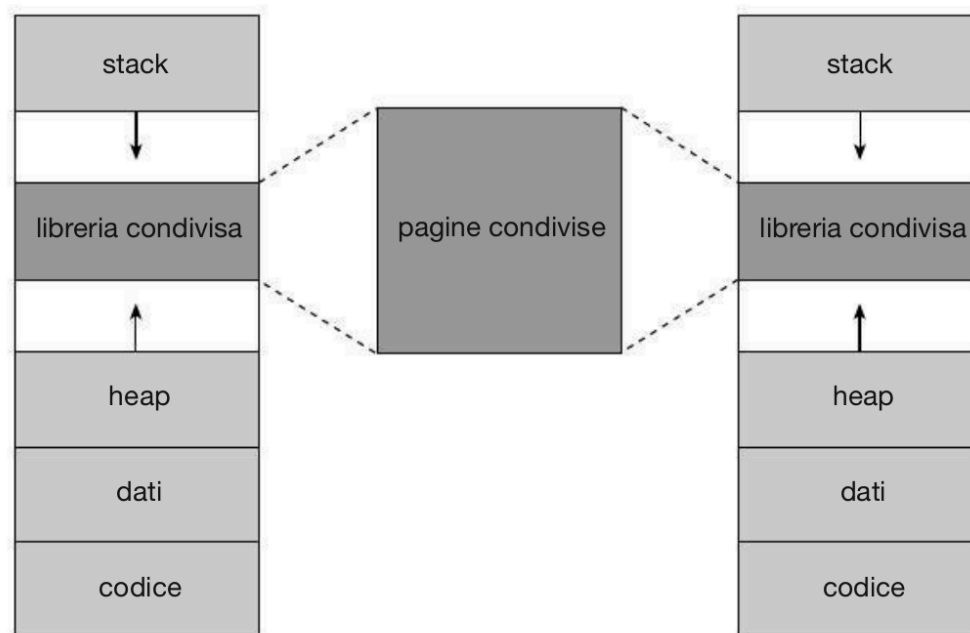
- Oltre il codice, contengono la lista delle librerie di sistema che useranno
- Contenute in una posizione predefinita

Le librerie e condivise sono identificate dal nome e dalla versione

Loader, librerie e pagine condivise

Le librerie sono caricate dal Loader in indirizzi intermedi

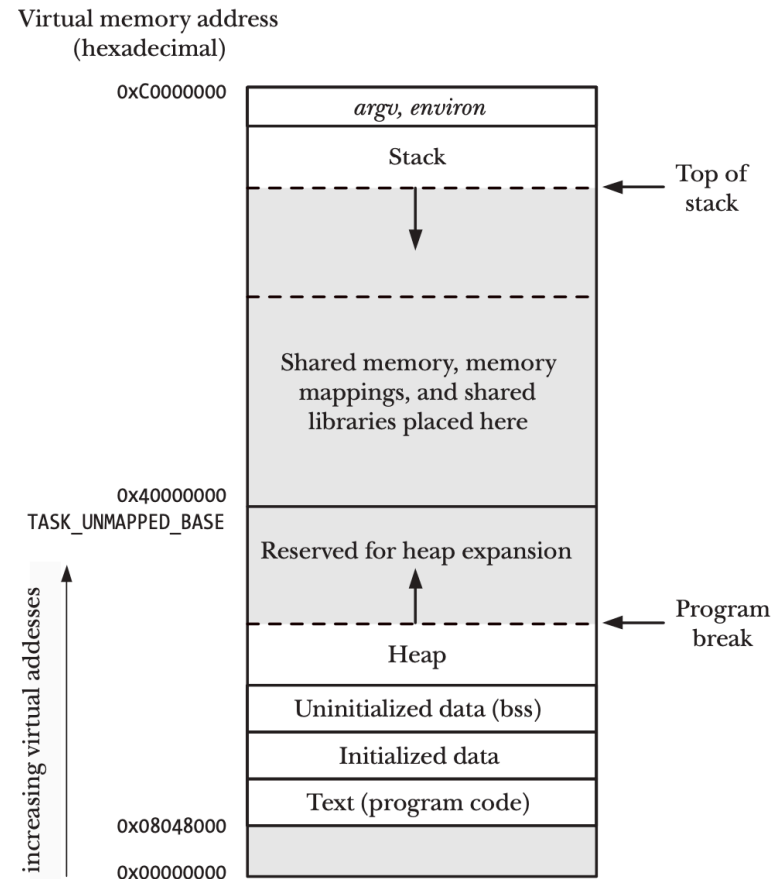
Se più processi usano la stessa libreria, la pagina viene condivisa



Loader, librerie e pagine condivise

La memoria condivisa tra processi funziona nello stesso modo

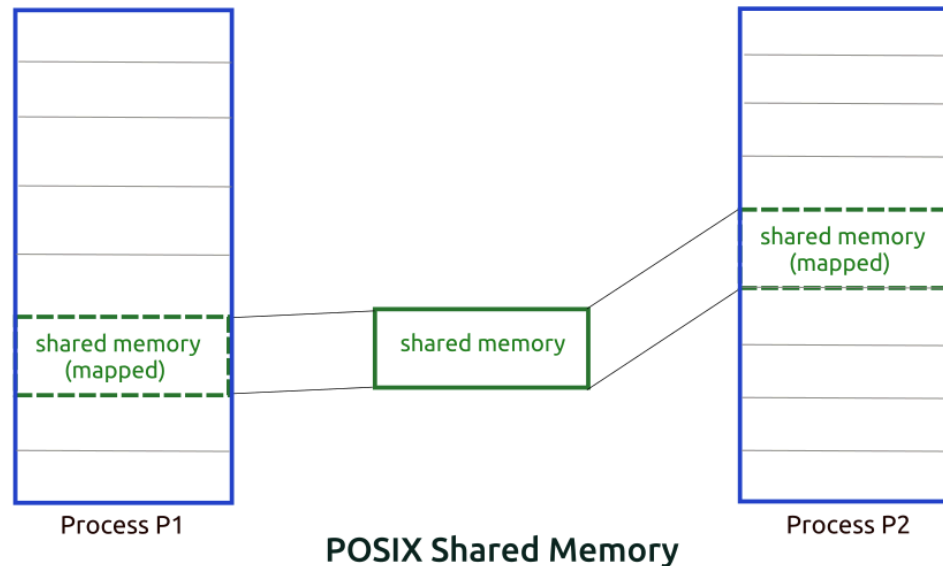
- Gli indirizzi tra heap e stack sono usati per tutto ciò che é condiviso



Loader, librerie e pagine condivise

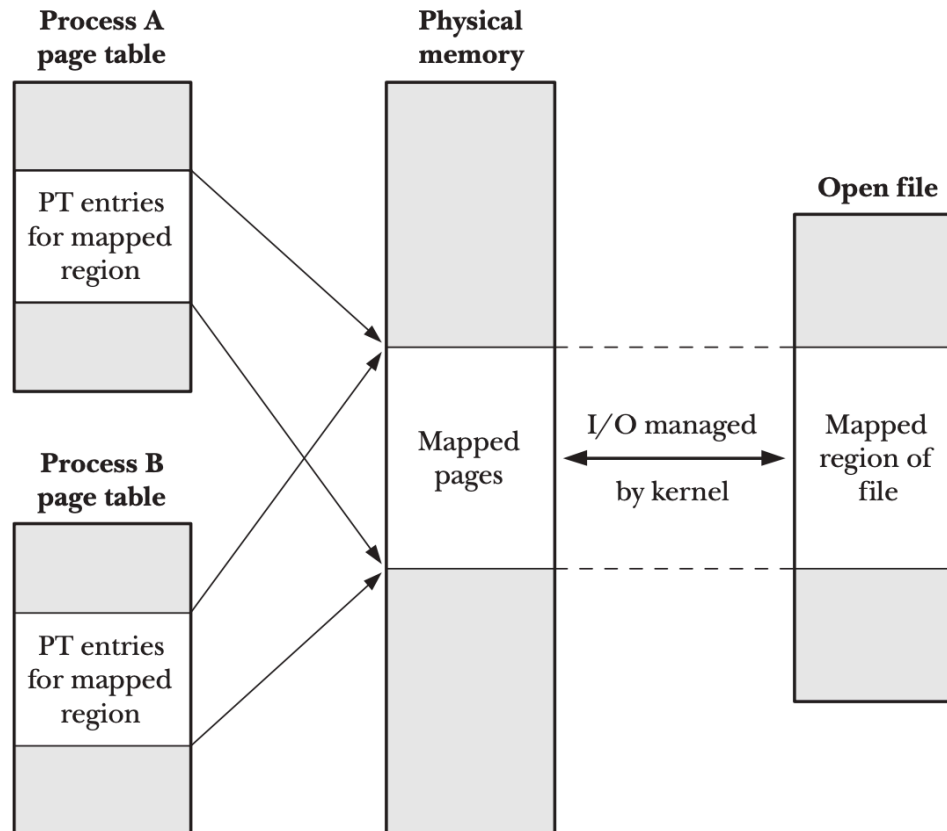
Tutto ciò è possibile grazie alla MMU

- Il SO imposta la MMU per implementare questo schema



Loader, librerie e pagine condivise

In caso di `mmap` con persistenza su file, il kernel si occupa di allineare la zona di memoria condivisa su disco



Gestione della memoria in Bash

Gestione della memoria in Bash

- `free` : mostra quanta memoria è disponibile/utilizzata/libera nella macchina
- `top` : mostra varie informazioni sui processi
 - Colonna `RES` : *Resident Set Size* quante pagine dello spazio virtuale di un processo sono caricate in memoria fisica
 - Colonna `VIRT` : *Virtual Set Size* quante pagine sono state usate dal processo nella sua storia
 - Colonna `%MEM` : $\text{RES} / \text{totale}$, ovvero percentuale di memoria fisica della macchina contenente pagine del processo

Gestione della memoria in Bash

- `cat /proc/meminfo` : mostra informazioni dettagliate su memoria della macchina
- `ldd eseguibile` : mostra quali libreri condivise esso richiede
- `objdump -p eseguibile` : dissector del formato ELF

Domande

Il compito della Memory Management Unit è:

- Gestire il funzionamento della cache
- Allocare zone di memoria
- Tradurre gli indirizzi da virtuali a fisici

Cosa fa la MMU quando una pagina non è in memoria?

- La carica
- Termina il processo che ha generato l'indirizzo
- Avverte il SO

La zona di memoria Stack viene utilizzata per:

- Memorizzare variabili globali
- Memorizzare il codice del programma
- Contenere le variabili relative alle funzioni
- Allocare la memoria dinamica

Domande

Un sistema ha pagine da 1KB, indirizzi virtuali da 32bit e fisici da 16bit. Quanti bit sono dedicati all'offset di pagina?

- 6
- 10
- 22

Un sistema ha pagine da 1KB, indirizzi virtuali da 32bit e fisici da 16bit. Di quante pagine dispone un processo nello spazio degli indirizzi virtuali?

- 64
- 1024
- circa 4 Milioni

In Linux, il Loader è:

- Un componente del SO
- La funzione principale di un programma
- La zona di memoria dove è memorizzato il codice del programma in esecuzione

In Linux, le librerie condivise sono mappate in memoria:

- In una zona intermedia tra stack e heap
- Nella zona di dati
- Nella zona di codice
- Nello stack