

Sommario

- ▶ Overlay
- ▶ Memoria virtuale
- ▶ Algoritmi di *rimpiazzo* (page replacement)
- ▶ Working set
- ▶ Memoria segmentata
- ▶ Architettura x86
- ▶ Gestione della memoria in Minix

Cosa fare quando la memoria non basta

due strategie:

- ▶ **swapping**
un processo viene temporaneamente tolto dalla memoria per far posto ad altri
- ▶ **memoria virtuale**
i processi possono eseguire anche se sono caricati in memoria solo in parte

Problemi con il nostro modello di memoria

Overlay

Assunzione: l'*intero processo* deve risiedere in memoria fisica *contigua* per poter eseguire

Problemi:

- Non posso eseguire processi grandi (più grandi della mem fisica)
- Il numero di processi è limitato dalla memoria
- Frammentazione

Il programmatore suddivide il programma in pezzi

Esempio: input, elaborazione, output

Quando un pezzo non serve più, viene sovrascritto con il prossimo

Il compito di suddividere il programma cade sul programmatore

Problemi e soluzioni

Non posso eseguire processi grandi (più grandi della mem fisica)

Soluzione: overlay

Il numero di processi è limitato dalla memoria

Soluzione: swapping

Frammentazione

Soluzione: tagliare lo spazio del processo in pezzi più piccoli

- separiamo lo spazio per il codice e per i dati
- usiamo due coppie di registri base, limit

Generalizzazione: partizionare il processo in N segmenti, usiamo N copie di registri base,limit \Rightarrow *memoria segmentata*

Memoria paginata

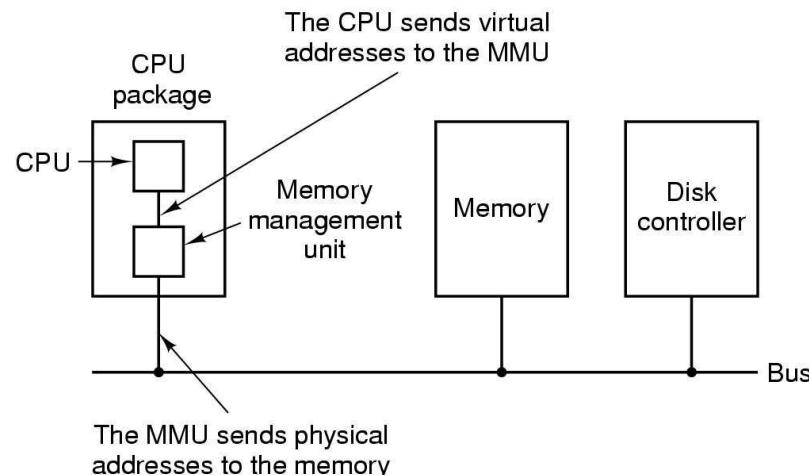
Una *pagina* è un segmento di dimensione fissata (es. 4K)

È sufficiente l'indirizzo di base; limit è implicito

Indirizzo logico: (num. pagina, offset)

Usiamo una tabella (page table) per mappare le pagine

La MMU è incorporata nella CPU

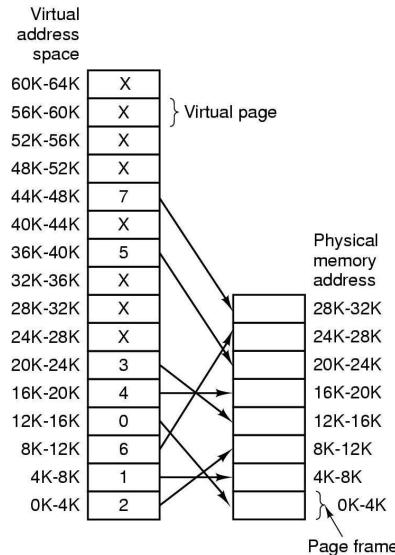


Terminologia

Lo spazio di indirizzamento virtuale è suddiviso in *pagine*

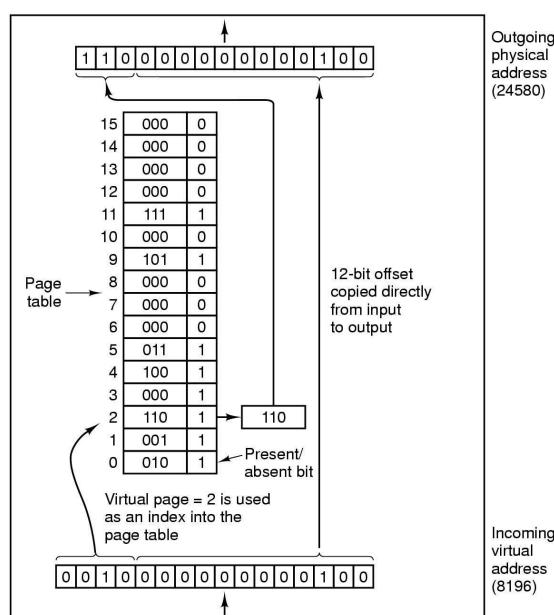
A ciascuna pagina può corrispondere un *page frame* (pagina fisica)

Corrispondenza fra pagine virtuali e fisiche



Traduzione degli indirizzi

Per ogni indirizzo di memoria nel codice occorre eseguire una traduzione
 ⇒ deve essere estremamente efficiente
 deve essere implementata in hardware



Cosa succede se il processo fa accesso a una pagina assente?

MMU nota che la pagina è assente
 esegue un **fault** (interrupt sincrono)

il S.O. deve gestire il fault

- trova una pagina fisica poco usata
- salva il contenuto della pag. fisica su disco
- carica il contenuto della pagina desiderata dal disco
- modifica la page table
- riprende il processo dall'istruzione che ha causato il fault

Page Tables

Problema: dimensione della page table

mappano le **pagine virtuali** su **page frames**

Problemi:

- ▶ la tabella può essere troppo grande
- ▶ la mappatura dev'essere veloce

Tipica architettura a 32 bit, con pagine di 4K

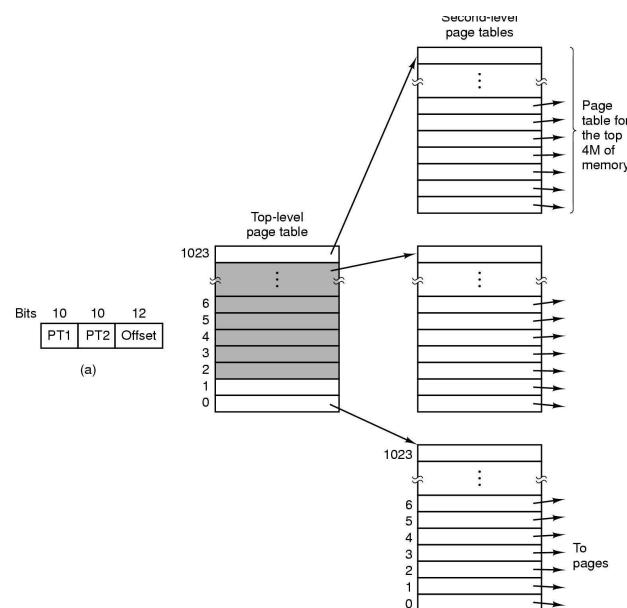
Spezziamo l'indirizzo virtuale in 20 (page index) + 12 (offset)

⇒ la page table ha 2^{20} righe

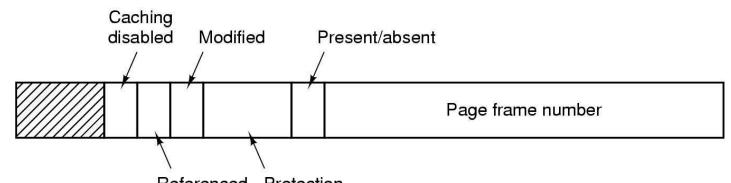
una riga occupa almeno 3 byte (# page frame + bits)

⇒ la page table occupa $3 * 2^{20}$ byte = 3MB!

Multi-level page tables



Tipica riga di una page table



Problema: # accessi alla memoria

Translation Lookaside Buffer

Gli accessi alla RAM sono costosi

Con mem virt paginata a 2 livelli, per un accesso rischio di farne 3

Soluzione: Translation Lookaside Buffer (TLB)

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

TLB

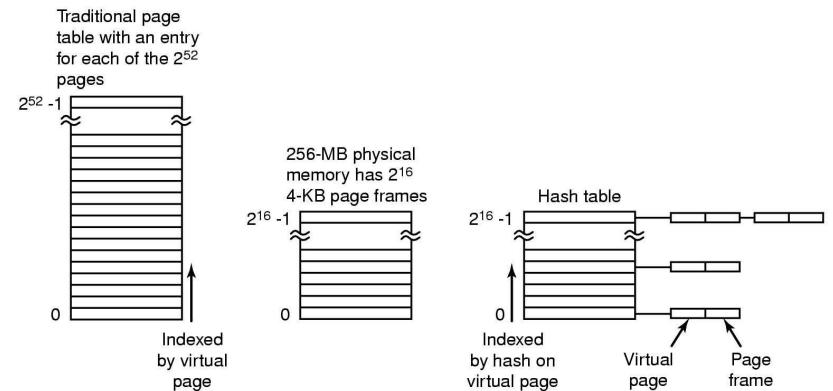
Il contenuto della TLB è relativo a un processo

Context switch fra processi \Rightarrow svuoto (**flush**) la TLB

\Rightarrow non conviene!

Ecco perché lo scheduler di Linux dà un bonus al processo corrente

Inverted page tables



Four times when OS involved with paging

- Process creation
 - create page table
- Process execution
 - MMU reset for new process
 - TLB flushed
- Page fault time
 - determine virtual address causing fault
 - swap target page out, needed page in
- Process termination time
 - release page table, pages

Page fault handling (i)

0. Hardware traps to kernel
1. General registers saved
2. OS determines which virtual page needed
3. OS checks validity of address, seeks page frame
4. If selected frame is dirty, write it to disk

Page fault handling (ii)

5. OS brings schedules new page in from disk
6. Page tables updated
7. Faulting instruction backed up to when it began
8. Faulting process scheduled
9. Registers restored
10. Program continues

Algoritmi di rimpiazzo delle pagine

È necessario mantenere un certo numero di pagine libere
Problema: quale pagina rimuovere?
Meglio scegliere una pagina poco usata
Se la pagina è modificata va salvata

Algoritmo ottimale

Rimpiazzare la pagina che sarà usata nel futuro più remoto

Ottimale, ma irrealizzabile

Occorre usare una **stima**; quale?

- usare la pagina che è inutilizzata da più tempo
- usare la pagina che è in memoria da più tempo
- usare una pagina dal processo che ne ha di più

Not Recently Used page replacement algorithm

Each page has Reference bit, Modified bit

bits are set when page is referenced, modified

referenced	modified
0	0
0	1
1	0
1	1

Pages are classified:

NRU removes page at random from lowest numbered non empty class

FIFO page replacement algorithm

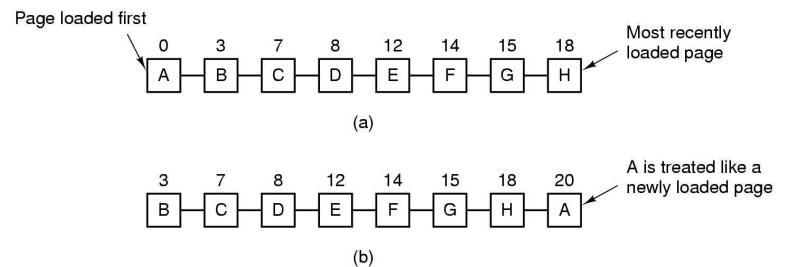
Maintain a linked list of all pages

in order they came into memory

Page at beginning of list replaced

Disadvantage: page in memory the longest may be often used

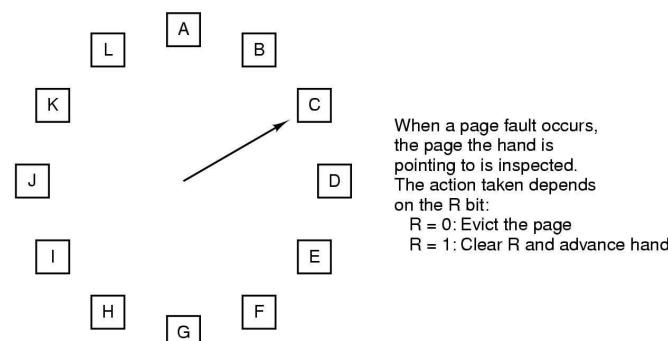
Second chance page replacement algorithm



Le pagine sono in ordine di arrivo

Se la pagina in testa ha il bit **referenced**, viene rimessa in fondo alla coda

Clock page replacement algorithm



Least Recently Used (LRU)

Euristica: una pagina usata di recente verrà usata di nuovo \Rightarrow butta via la pagina non usata da più tempo

Conserva tutte le pagine in una lista

- aggiorna la lista ad ogni riferimento a memoria

Ottimale: conserva un timestamp per ogni pagina

- aggiorna il timestamp ad ogni rif. a memoria

Impossibile da implementare senza supporto HW

In pratica questo algoritmo è poco usato

Not Frequently Used (aging)

	R bits for pages 0-5, clock tick 0	R bits for pages 0-5, clock tick 1	R bits for pages 0-5, clock tick 2	R bits for pages 0-5, clock tick 3	R bits for pages 0-5, clock tick 4
Page	1 0 1 0 1 1	1 1 0 0 1 0	1 1 0 1 0 1	1 0 0 0 1 0	0 1 1 0 0 0
0	10000000	11000000	11100000	11110000	01111000
1	00000000	10000000	11000000	01100000	10110000
2	10000000	01000000	00100000	00100000	10001000
3	00000000	00000000	10000000	01000000	00100000
4	10000000	11000000	01100000	10110000	01011000
5	10000000	01000000	10100000	01010000	00101000
(a)	(b)	(c)	(d)	(e)	

Locality of Reference

The **locality principle** states that processes tend to reference memory in patterns, not randomly. Memory references tend to be **clustered**.

- If a page is referenced, it is likely that the same page will be referenced again in the near future (**temporal locality**)
- If a page is referenced, it is likely that nearby pages will also be referenced (**spatial locality**)

Si verifica **sperimentalmente** che molti programmi esibiscono questa località

Gli algoritmi che esibiscono località sono da preferire

Working Set theory (Peter Denning)

At any given time, each process has a working set of pages; that is, the pages that it is actually using.

The working set is usually a small portion of the entire address space of the process. The working set may change over time, triggering page faults, but these will not occur frequently.

The working set theory predicts that if the operating system can keep every process's working set in main memory, there will be few page faults and the system will perform well.

Thrashing

Se l'insieme dei working set dei processi in esecuzione è più grande della RAM disponibile

Vengono generati page fault ogni poche istruzioni

Il sistema passa il suo tempo nel memory manager

Non viene fatto lavoro utile \Rightarrow Thrashing (agitarsi in maniera scomposta)

Stima del working set

Working set: pagine accedute nell'intervallo di tempo $(t, t - \delta)$

wss_i : working set size for process i

$$D = \sum_i wss_i$$

obiettivo: tenere $D <$ numero di pagine fisiche

posso ridurre D swappando alcuni processi

Memoria segmentata

Lo spazio dei processi è suddiviso in più *segmenti*

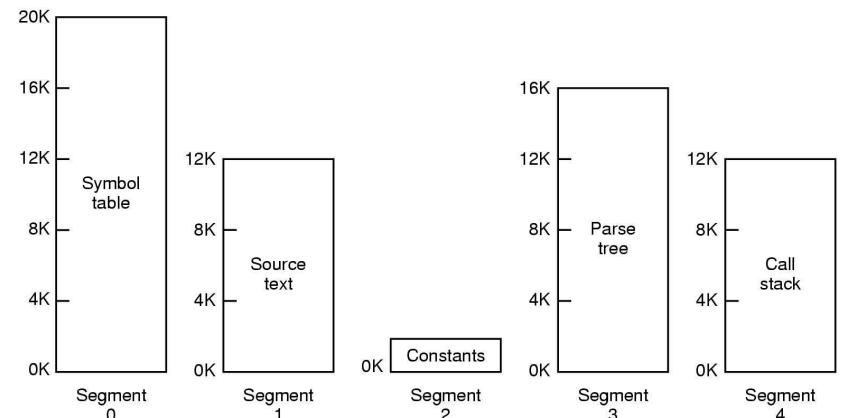
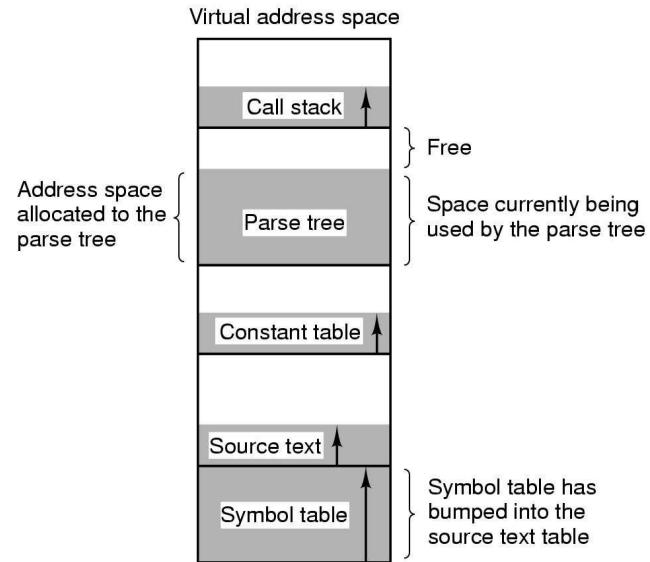
A discrezione del programmatore

I segmenti hanno dimensione diversa \Rightarrow problema: la gestione della memoria diventa complicata

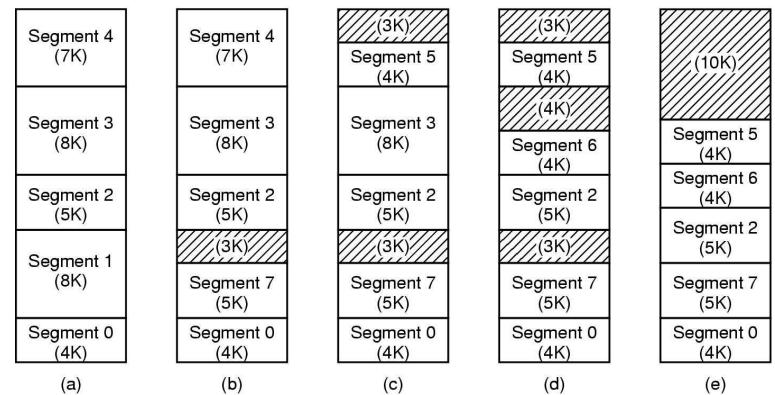
Occorre una *segment table*

Per ogni segmento:

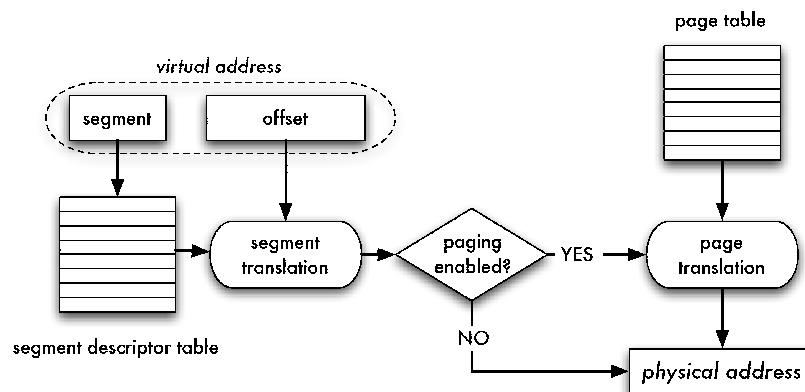
- base
- limit
- permissions



Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection



Memoria segmentata e paginata: l'architettura Intel x86



Memoria segmentata e paginata: l'architettura Intel x86

Spazio di indirizzamento: max 2^{16} segmenti, ciascuno di 4GB

Local Descriptor Table (LDT) per processo

- testo, stack, dati, ...

Global Descriptor Table (GDT) unica

- segmenti di sistema

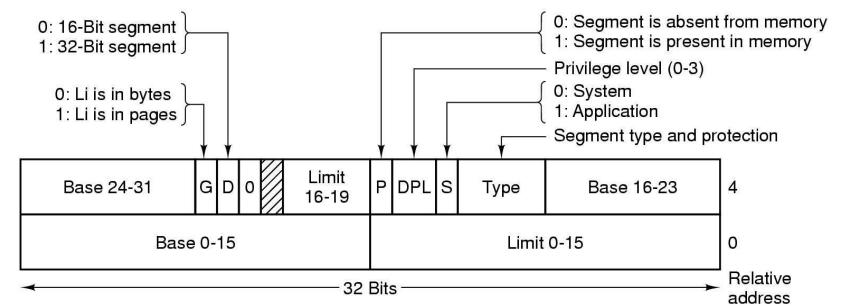
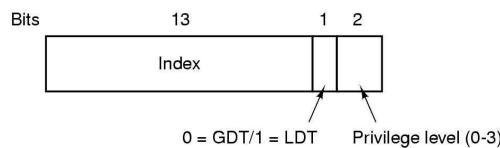
I registri del x86 che gestiscono i segmenti

II segment descriptor del x86

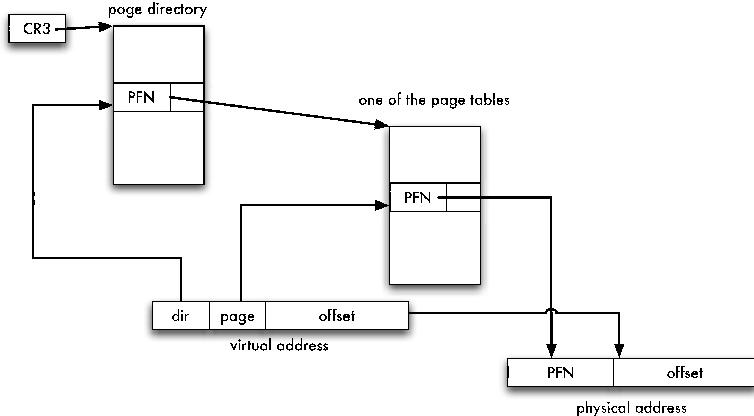
CS: registro che contiene un **selector** per il segmento codice

DS: registro che contiene il selector per il segmento dati

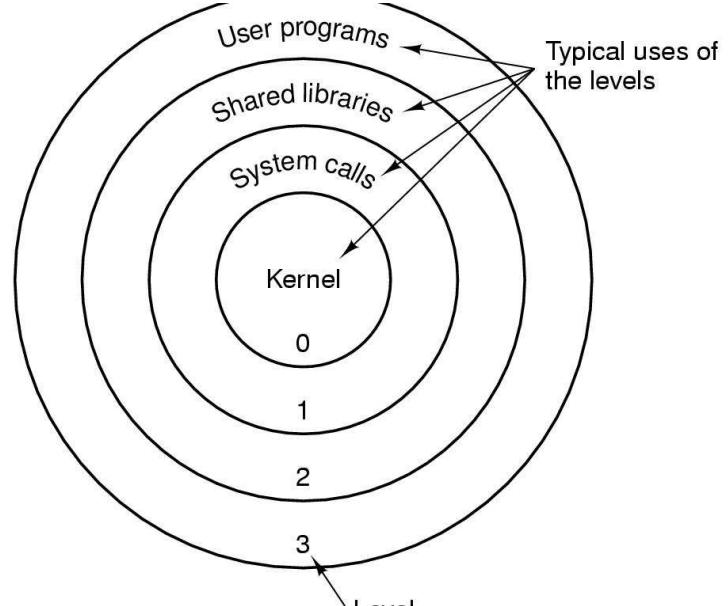
Un **segment selector** è un registro di 16 bit



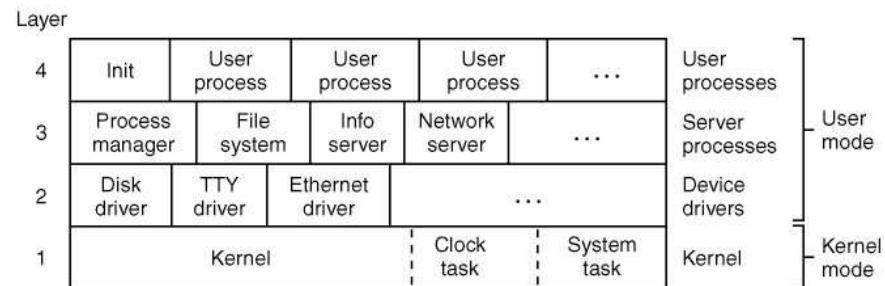
Traduzione di un indirizzo (architettura x86)



Livelli di protezione nell'architettura x86



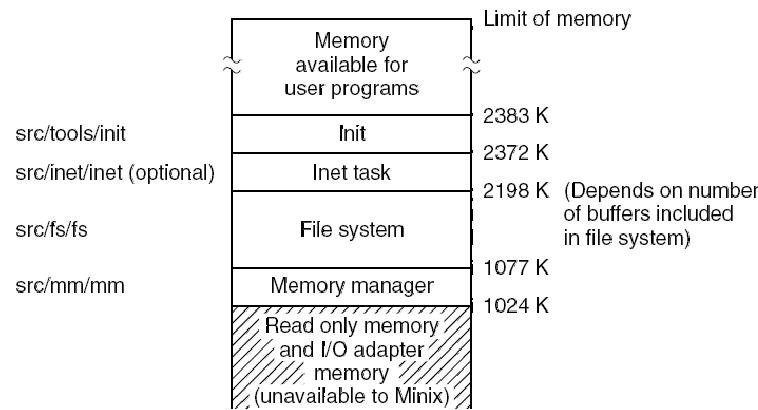
Architettura di Minix



Separazione di responsabilità

- ▶ *policy* (quale processo mettere in memoria) PM
- ▶ *mechanism* (manipolazione registri CPU) system task (kernel)

Memory layout



Gestione della memoria

Minix non usa la paginazione

Allocazione contigua

List of holes ordered by size; best fit

Un programma deve dichiarare quanta memoria desidera

chmem – change memory allocation

Shared text

EXAMPLES

```
chmem =50000 a.out # Give a.out 50K of stack space  
chmem -4000 a.out # Reduce the stack space by 4000 bytes  
chmem +1000 file1 # Increase each stack by 1000 bytes
```

DESCRIPTION

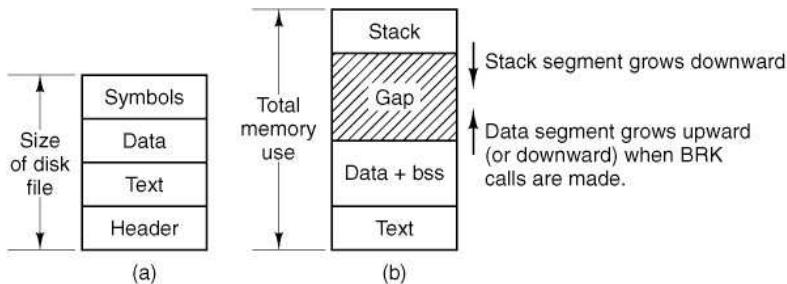
... If the combined stack and data segment growth exceeds the stack space allocated, the program will be terminated.

It is therefore important to set the amount of stack space carefully. If too little is provided, the program may crash. If too much is provided, memory will be wasted,

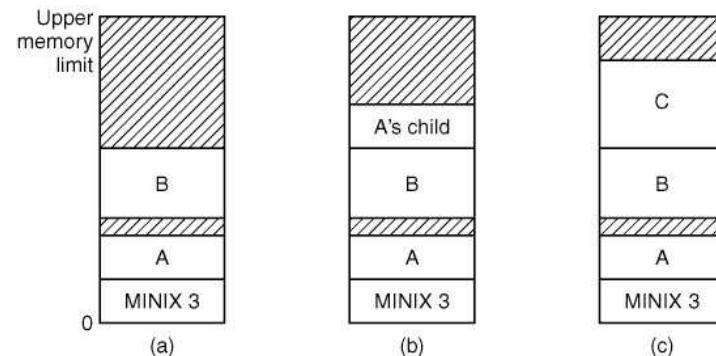
Due maniere di linkare:

- ▶ combined I and D space
- ▶ separate I and D space

Relazione fra file binario e layout di memoria del processo



Allocazione di memoria



(a) Prima. (b) Dopo una *fork*. (c) Dopo che il figlio fa una *exec*.

Message type	Input parameters	Reply value
fork	(none)	Child's PID, (to child: 0)
exit	Exit status	(No reply if successful)
wait	(none)	Status
wailpid	Process identifier and flags	Status
brk	New size	New size
exec	Pointer to initial stack	(No reply if successful)
kill	Process identifier and signal	Status
alarm	Number of seconds to wait	Residual time
pause	(none)	(No reply if successful)
sigaction	Signal number, action, old action	Status
sigsuspend	Signal mask	(No reply if successful)
sigpending	(none)	Status
sigprocmask	How, set, old set	Status
sigreturn	Context	Status
getuid	(none)	Uid, effective uid
getgid	(none)	Gid, effective gid
getpid	(none)	PID, parent PID

setuid	New uid	Status
setgid	New gid	Status
setsid	New sid	Process group
getpgid	New gid	Process group
time	Pointer to place where current time goes	Status
stime	Pointer to current time	Status
times	Pointer to buffer for process and child times	Uptime since boot
ptrace	Request, PID, address, data	Status
reboot	How (halt, reboot, or panic)	(No reply if successful)
svrctl	Request, data (depends upon function)	Status
getsysinfo	Request, data (depends upon function)	Status
getprocnr	(none)	Proc number
memalloc	Size, pointer to address	Status
memfree	Size, address	Status
getpriority	Pid, type, value	Priority (nice value)
setpriority	Pid, type, value	Priority (nice value)
gettimoofday	(none)	Time, uptime

Strutture dati del PM

- ▶ Process table
- ▶ Hole table

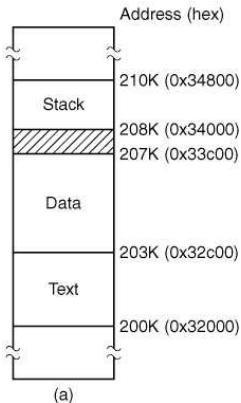
La process table tripartita

Process management	Memory management	File management
Registers Program counter Program status word Stack pointer Process state Time when process started CPU time used Children's CPU time Time of next alarm Message queue pointers Pending signal bits Process id Various flag bits	Pointer to text segment Pointer to data segment Pointer to bss segment Exit status Signal status Process id Parent process Process group Real uid Effective uid Real gid Effective gid Bit maps for signals Various flag bits	UMASK mask Root directory Working directory File descriptors Effective uid Effective gid System call parameters Various flag bits

kernel,

PM, FS

Un processo in memoria



Address (hex)		
	Virtual	Physical
	Length	Length
Stack	0x8	0xd0
Data	0	0xc8
Text	0	0xc8

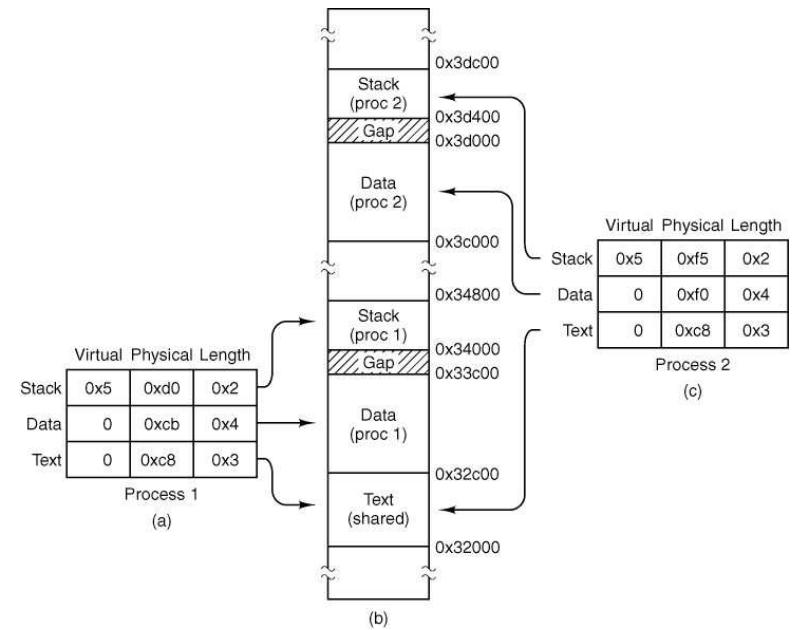
(b)

Address (hex)		
	Virtual	Physical
	Length	Length
Stack	0x5	0xd0
Data	0	0xcb
Text	0	0xc8

(c)

(b) I e D combinati. (c) I e D separati.

I e D separati



La chiamata di sistema fork(2)

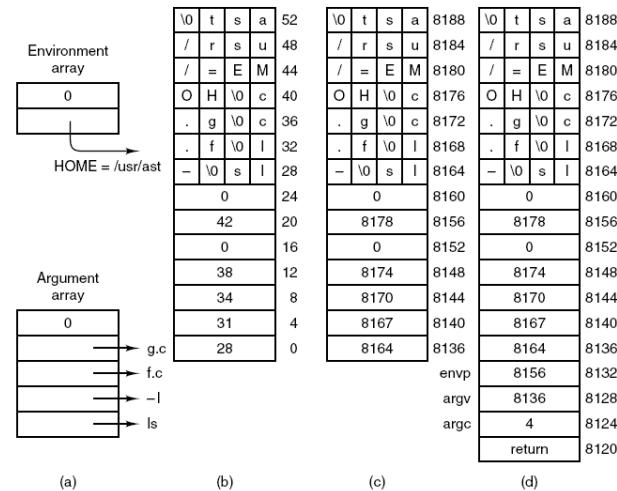
1. Check to see if process table is full.
2. Try to allocate memory for the child's data and stack.
3. Copy the parent's data and stack to the child's memory.
4. Find a free process slot and copy parent's slot to it.
5. Enter child's memory map in process table.
6. Choose a PID for the child.
7. Tell kernel and file system about child.
8. Report child's memory map to kernel.
9. Send reply messages to parent and child.

La chiamata di sistema exec(2)

1. Check permissions—is the file executable?
2. Read the header to get the segment and total sizes.
3. Fetch the arguments and environment from the caller.
4. Allocate new memory and release unneeded old memory.
5. Copy stack to new memory image.
6. Copy data (and possibly text) segment to new memory image.
7. Check for and handle setuid, setgid bits.
8. Fix up process table entry.
9. Tell kernel that process is now runnable.

Esempio di exec

```
ls -l f.c g.c
execve("/bin/ls", argv, envp);
```



(a) array passati a `execve`. (b) Stack costruito da `execve`. (c) Stack dopo la rilocazione. (d) Stack come appare al `main` all'inizio

crtso C runtime start-off routine

```
push ecx          ! push environ
push edx          ! push argv
push eax          ! push argc
call _main        ! main(argc, argv, envp)
push eax          ! push exit status
call _exit        ! force a trap if exit fails
hlt
```

```
main(int argc, char** argv, char** envp)
```