

## Gestione della memoria

Ideally programmers want memory that is

- large
- fast
- non volatile

Memory hierarchy

- small amount of fast, expensive memory - cache
- some medium-speed, medium price main memory
- gigabytes of slow, cheap disk storage

**Memory manager handles the memory hierarchy**

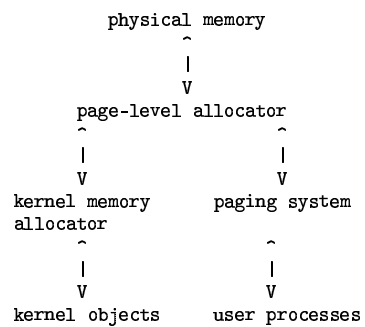
1

## Gerarchia delle memorie

0. registri CPU
  1. cache L1 (on chip)
  2. cache L2 (on board)
  3. RAM
  4. dischi
  5. nastri
- da più veloce a più lenta
  - da più costosa a meno costosa
  - da più piccola a più grande

2

## Allocazione della memoria nel kernel



3

## Criteri per valutare un allocatore

- $efficiency = \frac{\text{memory requested}}{\text{memory used to satisfy requests}}$

Idealmente, vorremmo il 100%

In pratica, il 50% è accettabile

Problema: frammentazione!

4

## Altri criteri di valutazione di un algoritmo di allocazione

- velocità
- semplicità della API
- possibilità di restituire solo *parte* della mem allocata
- capacità di restituire la memoria all'allocatore di pagine

5

## Algoritmo 0: *Resource Map Allocator*

La *resource map* è una lista di coppie (*base, size*) che monitorano le zone di memoria libera

Operazioni:

- `malloc(size)`
- `rmfree(base, size)`

Inizialmente:

```
(0, 1024)
|-----+
|-----|
|-----|
```

Dopo `malloc(256)`

```
(256, 768)-----+
|-----|-----+
|XXXXXXXXXXXXXXX|-----|
```

6

Dopo `malloc(320)`

```
(576, 448)-----+
|-----|-----+
|XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX|
```

Dopo `rmfree(256, 128)`

```
(576, 448)-----+
(256, 128)-----+
|-----|-----+
|XXXXXXXXXXXXXXX|XXXXXXXXXXXXXXX|
```

Dopo molte allocazioni e deallocazioni...

```
(832, 32)-----+
(544, 128)-----+
(288, 64)-----+
(128, 32)-----+
|-----|-----|-----|-----|
|XXXXXX|XXXXXXXX|XXXXXXXXXXXXXXX|XXXXXXXXXX|XXXXXXX|
```

7

## Differenti strategie

- first fit
- best fit
- worst fit

Nessuna è in assoluto migliore ... dipende da come sono fatte le richieste

8

## Analisi del resource map allocator

### Pro:

- facile da implementare
- permette di deallocare solo una parte di un blocco allocato
- la `rmfree()` può collassare buchi adiacenti

### Contro:

- frammentazione; allocazioni grandi sono penalizzate
- ricerca lineare — è lenta!
- problema: come allochiamo la mappa stessa?

9

## Varianti

- per semplificare la API possiamo mantenere una lista di buchi e allocazioni – allora possiamo avere una `free(base)` di un solo argomento
- possiamo usare una *bitmap* invece di una lista

10

## Power-of-two free lists

| allocated buffers | list headers             | free buffers |
|-------------------|--------------------------|--------------|
| []----->          | 32 -->[]-->[]-->[]       |              |
| [X]----->         | 64 -->[]-->[]            |              |
|                   | 128 -->[]-->[]-->[]      |              |
| [XXXX]----->      | 256 -->[____]            |              |
|                   | 512 -->[_____]-->[_____] |              |
|                   | 1024 -->[_____]          |              |

Ogni buffer contiene un puntatore

- se è libero, punta al prossimo nella free list
- se è occupato, punta alla testa della sua free list

11

## Analisi

### Pro:

- elimina la frammentazione (esterna)
- non penalizza le allocazioni grandi
- veloce (nessuna ricerca lineare)
- semplice (le strutture dati sono di dimensione costante)

### Contro:

- frammentazione interna
- non collassa i buchi

12



## Zone allocator

Usato in Mach e HP-UX

```
struct zone* zinit(size, max, alloc, name)
```

- size: dimensione di un blocchetto
- max: massimo numero di blocchetti
- alloc: dimensione iniziale della memoria da gestire
- name: stringa descrittiva

```
void* zalloc(struct zone * z);  
void zfree(struct zone * z, void *obj);
```

Alloca un oggetto

17

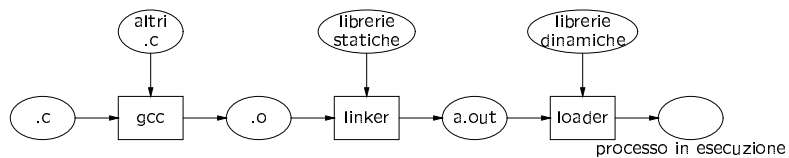
## Lo spazio di indirizzamento

Lo spazio di indirizzamento (address space) di un processo è *l'insieme di locazioni di memoria che può usare*

I programmi usano indirizzi *virtuali* che sono in qualche modo mappati sugli indirizzi *fisici*

18

## I passi che portano dal sorgente all'esecuzione



19

## Contenuti di un modulo rilocabile

- A header that indicates that the file is a link module.
- A list of sizes for the various parts of the module.
- The starting address.
- The machine instructions that constitute the program.
- Data areas initialized with their first values.
- Size indications for uninitialized data regions.
- Relocation information
- Debugging information

20

## La "memoria" usata da un programma

- testo
- variabili globali
- variabili automatiche
- memoria dinamica

21

## Testo del programma e variabili globali

Gestiti dal linker

22

## Variabili automatiche

Sono allocate sullo stack del processore

23

## Memoria automatica

malloc(3) è implementata grazie a brk(2)

24

## La chiamata di sistema brk(2)

brk, sbrk - change data segment size

```
int brk(void *end_data_segment);
void *sbrk(ptrdiff_t increment);
```

brk sets the end of the data segment to the value specified by end\_data\_segment, when that value is reasonable, the system does have enough memory and the process does not exceed its max data size.

sbrk increments the program's data space by increment bytes. sbrk isn't a system call, it is just a C library wrapper. Calling sbrk with an increment of 0 can be used to find the current location of the program break.

On success, brk returns zero, and sbrk returns a pointer to the start of the new area. On error, -1 is returned, and errno is set to ENOMEM.

25

## Il modello della memoria di un processo Unix

26

## Basic memory management (i)

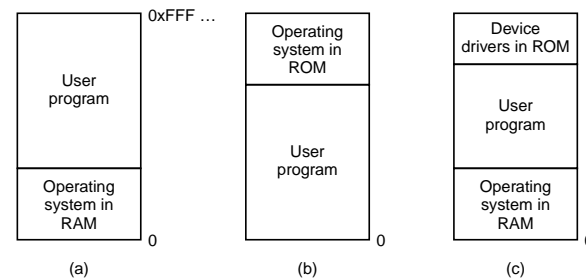
Tecniche obsolete per s.o. contemporanei, eccetto che per:

- embedded system
- palmtop
- smart card

27

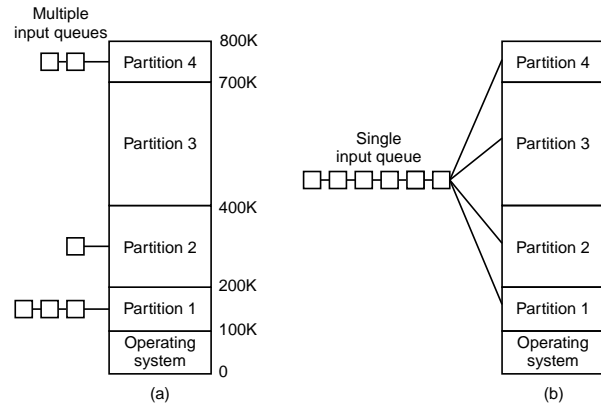
## Basic memory management (ii)

Un solo programma alla volta in memoria



28

## Multiprogrammazione con partizioni fisse



29

## Multiprogrammazione con partizioni fisse

facile da implementare, ma:

problema: algoritmo di allocazione

problema: frammentazione della memoria

30

## Relocation and protection

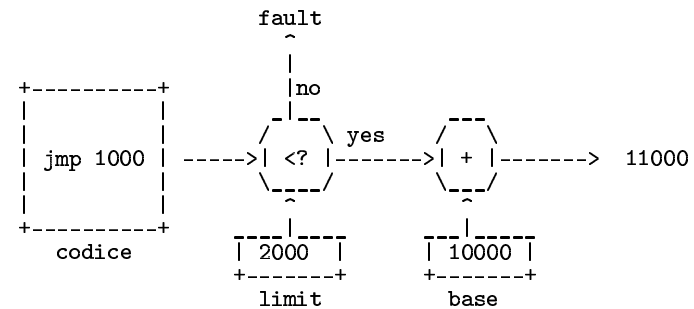
- Cannot be sure where program will be loaded in memory
- address of variables, code routines cannot be absolute
  - must keep a program out of other processes' partitions

Use *base* and *limit* values

- address locations added to base value to map to physical addr
- address locations larger than limit value is an error

31

## Uso dei registri *base* e *limit*



32

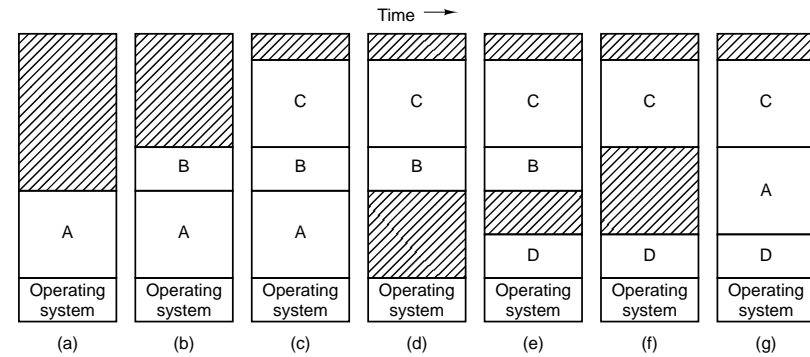
## Cosa fare quando la memoria non basta

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

33

## Swapping



34

## Swapping (ii)

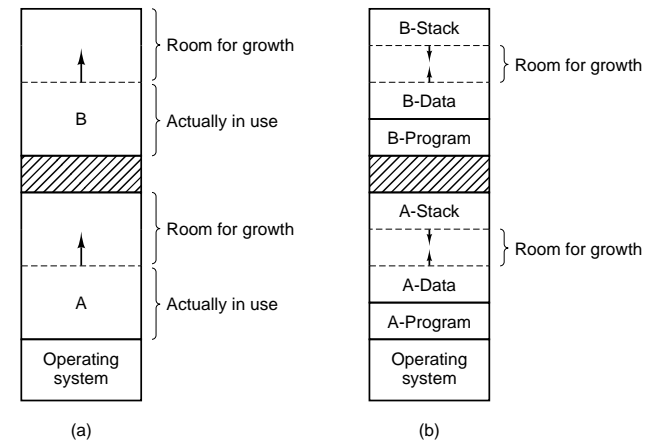
Problema: si formano “buchi” nella memoria sempre più piccoli:  
*frammentazione*

Compattazione: possibile ma costosa

Problema: cosa fare quando un processo cresce di dimensione?

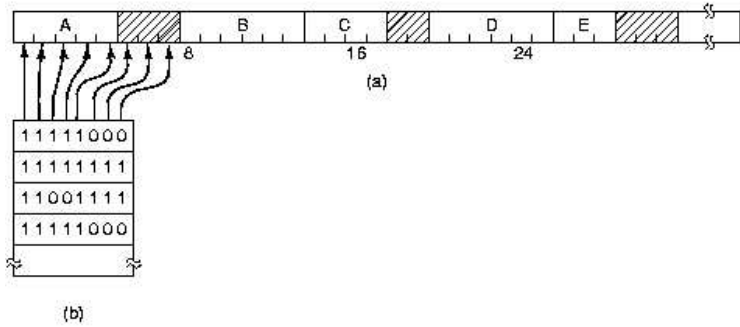
35

## Swapping (iii)



36

## Gestione della memoria con bitmap



37

## Bitmap

vantaggio: piccole dimensioni

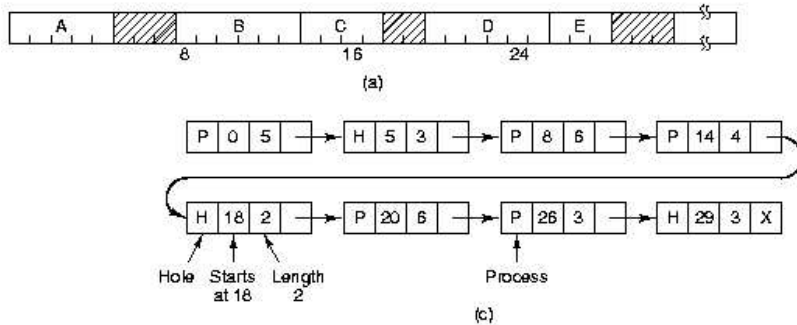
Esempio:

- suddivisione della memoria in pagine di 1K
- un bit per pagina
- 1MB di memoria  $\Rightarrow$  1024 bit = 128 byte

svantaggio: ricerca di un buco di dimensioni date può essere lenta

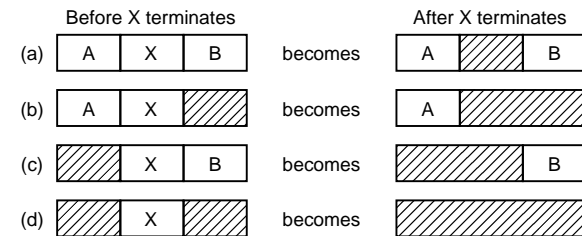
38

## Gestione della memoria con lista linkata



39

## Collassamento dei buchi alla terminazione di un processo



40

## Algoritmi di allocazione

- first fit
- best fit
- worst fit
- quick fit

### Ottimizzazione:

- liste separate per buchi e processi
- conservare la lista dei buchi in ordine di grandezza

41

## Problemi con il nostro modello di memoria

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

42

## Swap

Se un processo non userà la CPU per un lungo periodo, dovrebbe rilasciare la RAM che usa

43

## Overlay

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

44

## 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  $N$  coppie di registri base, limit

⇒ *memoria segmentata*

45

## Memoria segmentata

Lo spazio dei processi è suddiviso in più *segmenti*

A discrezione del programmatore

I segmenti hanno dimensione diversa

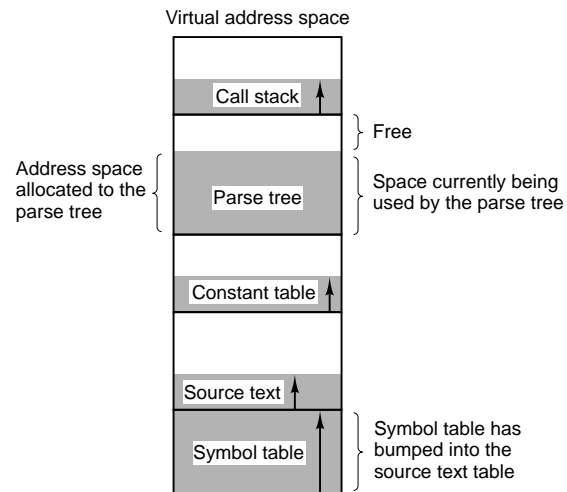
⇒ problema: la gestione della memoria diventa complicata

Occorre una *segment table*

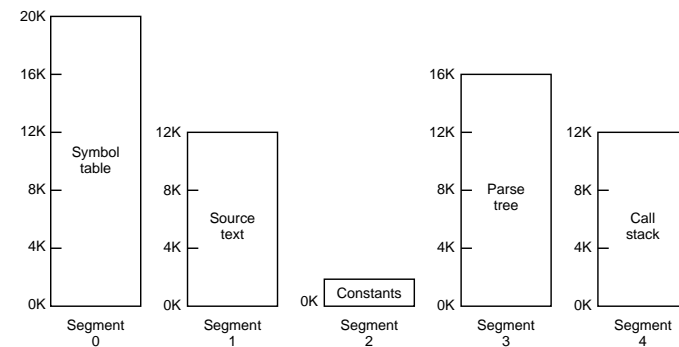
Per ogni segmento:

- start address
- permissions

46



47



48

## 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

49

## Memoria virtuale paginata

Ciascun processo vede uno spazio di indirizzi *virtuale*  $[0 - 2^{32})$

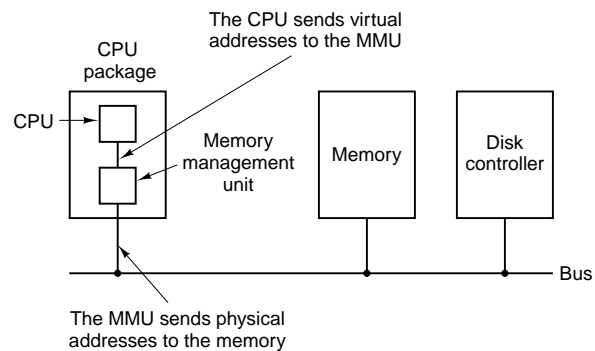
L'indirizzo *virtuale* viene tradotto in un indirizzo *fisico*



MMU: Memory Management Unit

50

## La MMU è (di solito) incorporata nella CPU



51

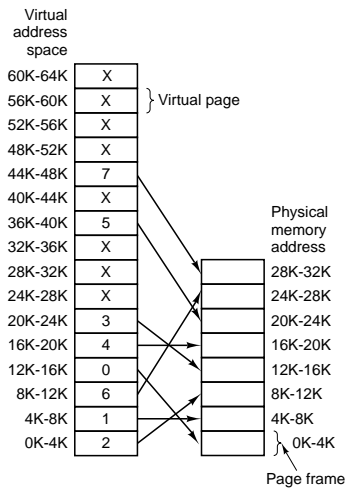
## Terminologia

Lo spazio di indirizzamento virtuale è suddiviso in *pagine*

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

52

## Corrispondenza fra pagine virtuali e fisiche



53

## Traduzione degli indirizzi

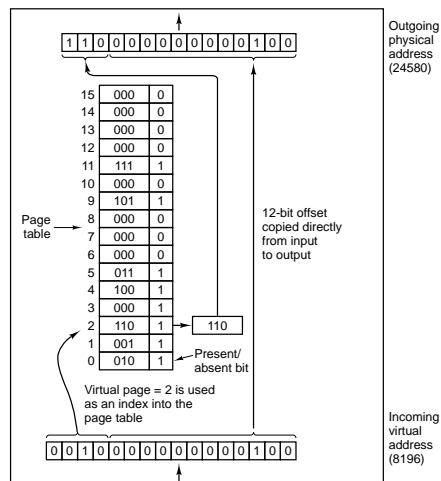
Per ogni indirizzo di memoria nel codice occorre eseguire una traduzione

⇒ deve essere estremamente efficiente

deve essere implementata in hardware

54

## Funzionamento della MMU



55

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

56