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

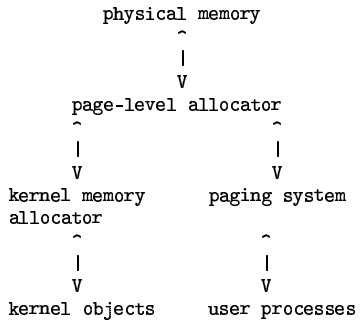
Memory manager handles the memory hierarchy

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

Allocazione della memoria nel kernel



Criteri per valutare un allocatore

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

Idealmente, vorremmo il 100%

In pratica, il 50% è accettabile

Problema: frammentazione!

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

Algoritmo 0: *Resource Map Allocator*

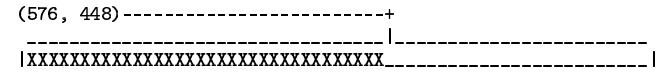
La *resource map* è una lista di coppie $(base, size)$ che monitorano le zone di *memoria libera*

Operazioni:
- malloc($size$)
- rmfree($base$, $size$)

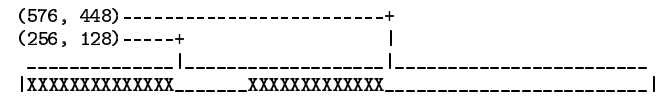
Inizialmente:

Dopo malloc(256)

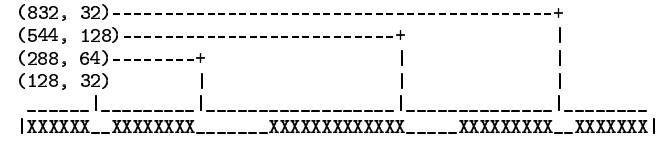
Dopo malloc(320)



Dopo rmfree(256, 128)



Dopo molte allocazioni e deallocazioni...



Differenti strategie

- first fit
- best fit
- worst fit

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

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

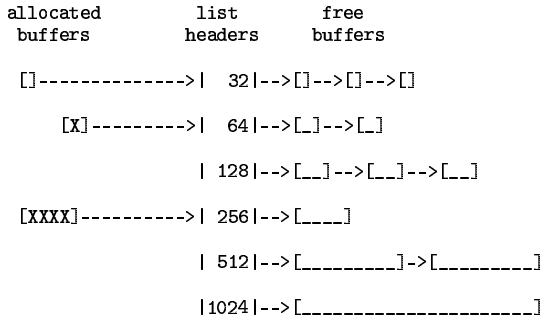
Contro:
- frammentazione; allocazioni grandi sono penalizzate
- ricerca lineare — è lenta!
- problema: come allochiamo la mappa stessa?

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

Power-of-two free lists



Ogni buffer contiene un puntatore
- se è libero, punta al prossimo nella free list
- se è occupato, punta alla testa della sua free list

Analisi

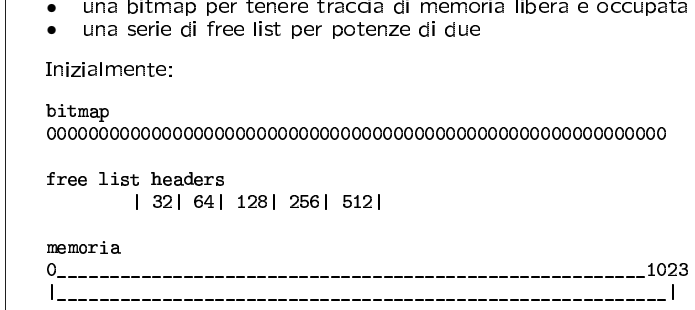
Pro:
- elimina la frammentazione (esterna)
- non penalizza le allocazioni grandi
- veloce (nessuna ricerca lineare)
- semplice (le strutture dati sono di dimensione costante)

Contro:
- frammentazione interna
- non collassa i buchi

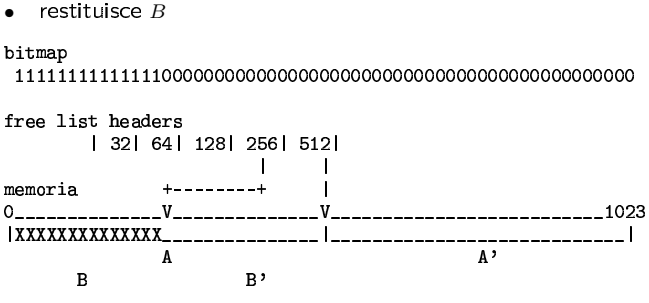
The *Buddy System*

Ingredienti:
- un grosso blocco di memoria da gestire
- una bitmap per tenere traccia di memoria libera e occupata
- una serie di free list per potenze di due

Inizialmente:



Chiamiamo allocate(256):
- splitta il blocco in due blocchi A e A', e mette A' nella free-list 512
- splitta A in B e B', mette B' nella free-list 256
- restituisce B



Chiamo allocate(128):
- la free-list 128 è vuota; allora
- toglie B' dalla lista 256
- splitta B' in C e C'
- C' va nella free-list 128
- restituisce C

Figura: TODO

Chiamo allocate(128):
- la free-list 128 è vuota; allora
- toglie B' dalla lista 256
- splitta B' in C e C'
- C' va nella free-list 128
- restituisce C

Figura: TODO

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);
```

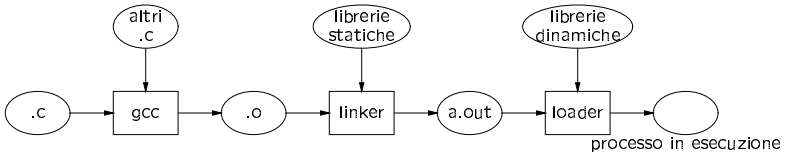
Alloca un oggetto

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*

I passi che portano dal sorgente all'esecuzione



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

La "memoria" usata da un programma

- testo
- variabili globali
- variabili automatiche
- memoria dinamica

Testo del programma e variabili globali

Gestiti dal linker

Variabili automatiche

Sono allocate sullo stack del processore

Memoria automatica

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

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.

Il modello della memoria di un processo Unix

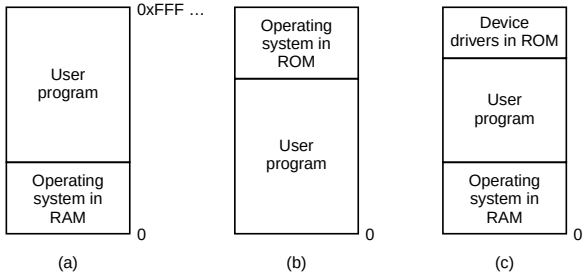
Basic memory management (i)

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

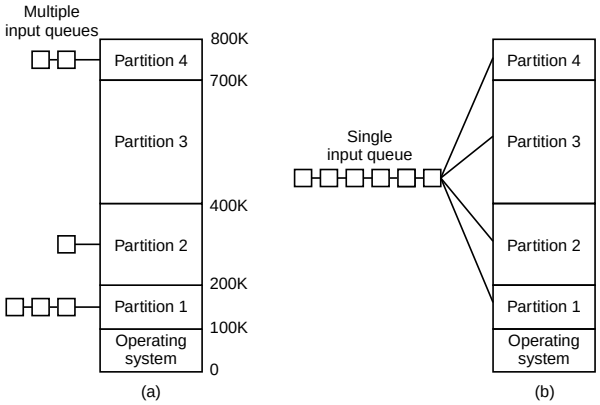
- embedded system
- palmtop
- smart card

Basic memory management (ii)

Un solo programma alla volta in memoria



Multiprogrammazione con partizioni fisse



Multiprogrammazione con partizioni fisse

facile da implementare, ma:

problema: algoritmo di allocazione

problema: frammentazione della memoria

Relocation and protection

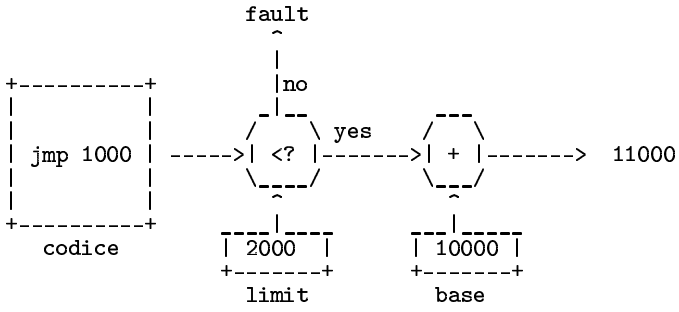
Cannot be sure where program will be loaded in memory

- 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

Uso dei registri *base* e *limit*



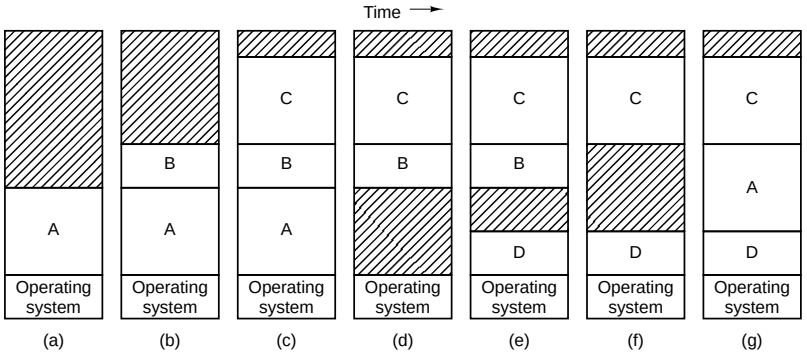
Cosa fare quando la memoria non basta

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

Swapping



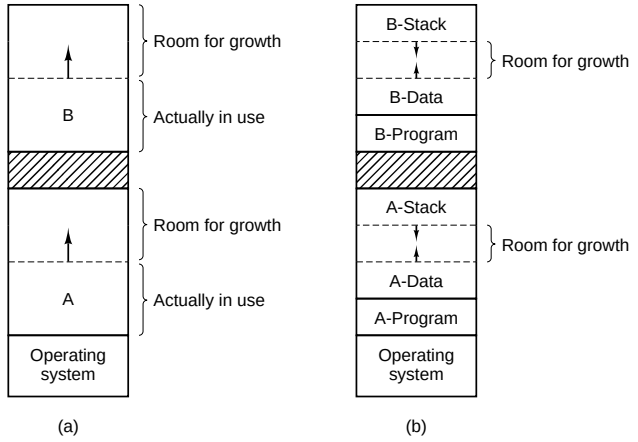
Swapping (ii)

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

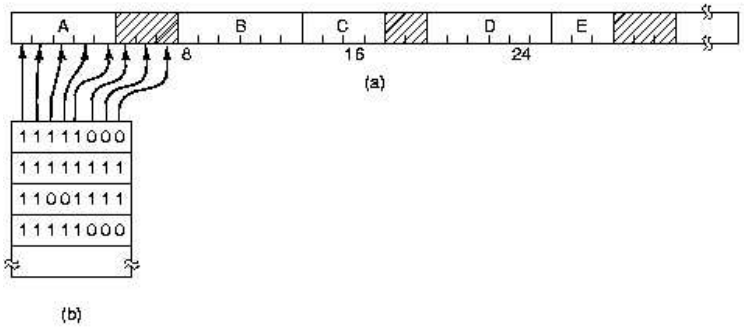
Compattazione: possibile ma costosa

Problema: cosa fare quando un processo cresce di dimensione?

Swapping (iii)



Gestione della memoria con bitmap



Bitmap

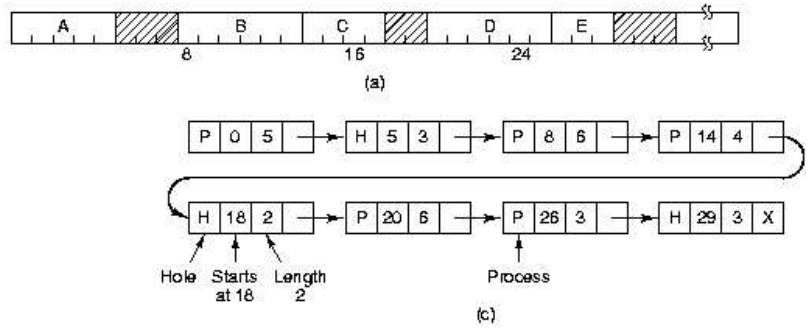
vantaggio: piccole dimensioni

Esempio:

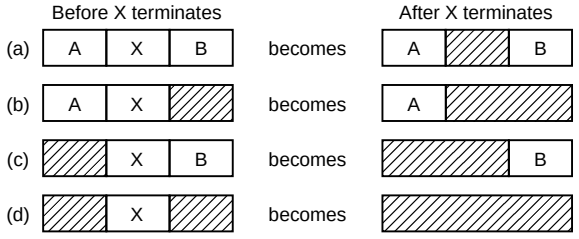
- suddivisione della memoria in pagine di 1K
- un bit per pagina
- 1MB di memoria $\Rightarrow$ 1024 bit = 128 byte

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

Gestione della memoria con lista linkata



Collassamento dei buchi alla terminazione di un processo



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

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

Swap

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

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

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
$\Rightarrow$ *memoria segmentata*

Memoria segmentata

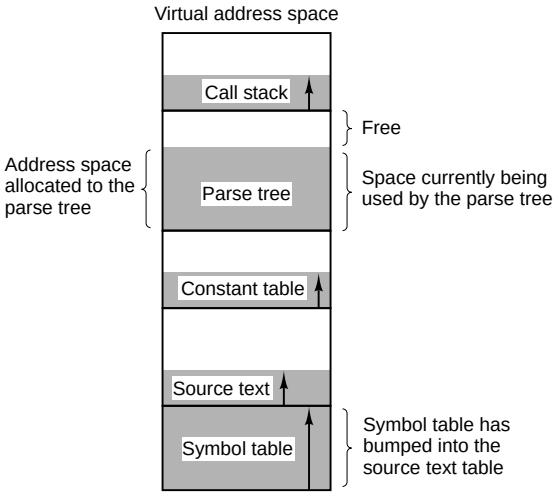
Lo spazio dei processi è suddiviso in più *segmenti*

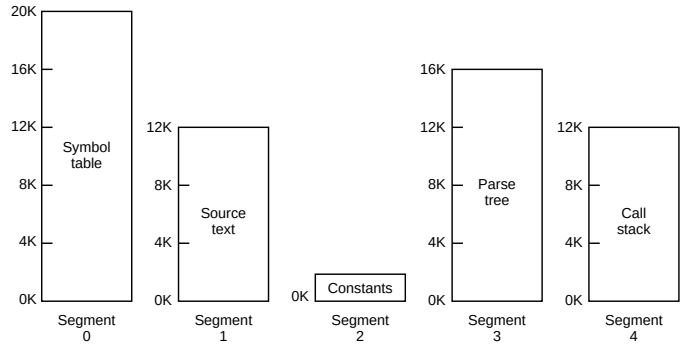
A discrezione del programmatore

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

Occorre una *segment table*

Per ogni segmento:
- start address
- permissions





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

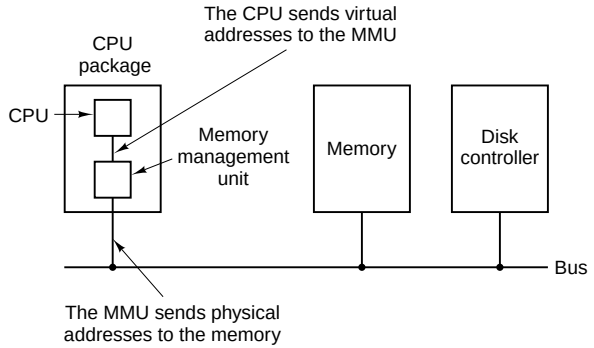
Memoria virtuale paginata

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

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

MMU: Memory Management Unit

La MMU è (di solito) incorporata nella CPU

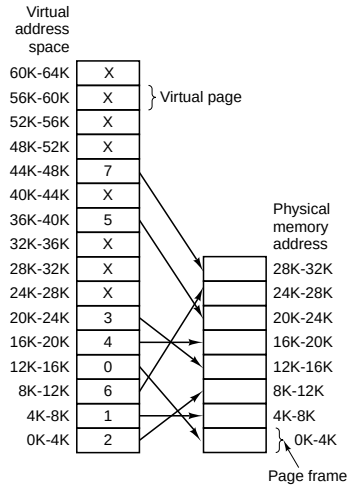


Terminologia

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



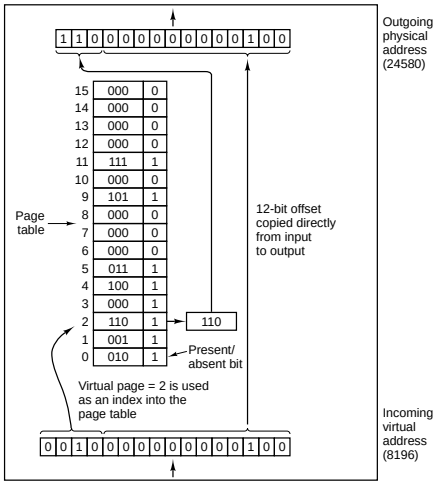
Traduzione degli indirizzi

Per ogni indirizzo di memoria nel codice occorre eseguire una traduzione

$\Rightarrow$ deve essere estremamente efficiente

deve essere implementata in hardware

Funzionamento della MMU



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