Matteo Vaccari

Informazioni sul corso:
`http://matteo.vaccari.name/so`

Diario delle lezioni:
`http://matteo.vaccari.name/so/diario`

Forum:
`http://matteo.vaccari.name/forum/`

Leggete il diario!

http://matteo.vaccari.name/so/diario

Dopo ogni lezione trovate:

- gli argomenti
- le sezioni del testo corrispondenti
- i lucidi
- documentazione addizionale
- esercizi

Tutte le letture riportate nel diario **sono parte del programma di esame**, a meno che non siano esplicitamente dichiarate *facoltative*

Modalità di esame

Quest'anno non c'è da fare un elaborato

Al posto dell'elaborato avremo lezioni in laboratorio

Lo scritto comprenderà domande sul lavoro in laboratorio (di cui appariranno note sul diario)

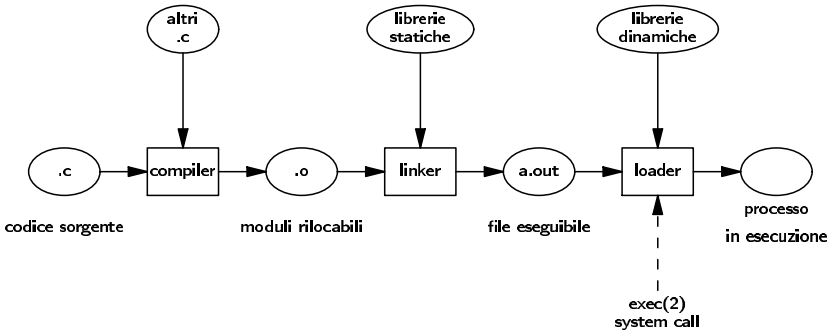
Lo spazio di indirizzamento

Memoria: un array di *byte*

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

I programmi usano indirizzi *logici* 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

Per esempio

```
...
static int gVar;
...
int proc_a(int arg) {
        ...
        gVar = 7;
        put_record(gVar);
        ...
}
```

Gary Nutt, *Operating Systems, A Modern Perspective*

Il modulo rilocabile

Code Segment

```
0000   ...
...
0008  entry proc_a
...
0220  load  =7, R1
0224  store R1, 0036
0228  push  0036
0232  call  'put_record'
...
0400  External reference table
...
0404  'put_record'  0232
...
0500  External definition table
...
0540  'proc_a'  0008
...
0600  (symbol table)
...
0799  (last location in the code segment)
```

Data Segment

```
...
0036 [Space for gVar variable]
...
0049 (last location in the data segment)
```

Il programma eseguibile (assoluto)

Code Segment

```
0000 (Other modules)
...
1008 entry proc_a
...
1220 load =7, R1
1224 store R1, 0136
1228 push 0136
1232 call 2334
...
1399 (End of proc_a)
...  (Other modules)
2334 entry put_record
...
2670 (optional symbol table)
...
2999 (last location in the code segment)
```

Nota: il modulo proc_a è stato rilocato all'indirizzo 1000

Data Segment

```
...
0136 [Space for gVar variable]
...
1000 (last location in the data segment)
```

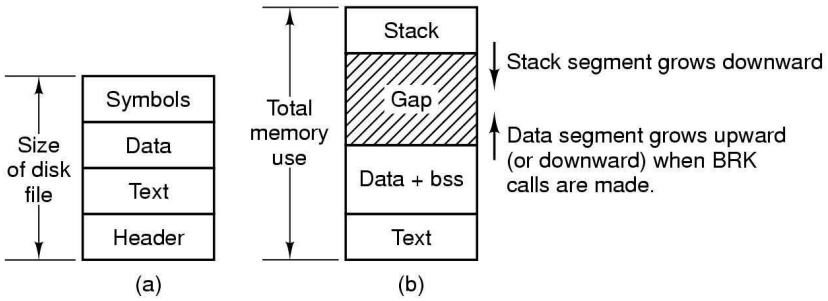
Il programma caricato all'indirizzo 4000

```
0000  (Other process's programs)
4000  (Other modules)
...
5008  entry proc_a
...
5220  load  =7, R1
5224  store R1, 7136
5228  push  7136
5232  call  6334
...
5399  (End of proc_a)
...   (Other modules)
6334  entry put_record
...
6670  (optional symbol table)
...
6999  (last location in the code segment)
7000  (first location in the data segment)
...
7136  [Space for gVar variable]
...
8000  (Other process's programs)
```

La "memoria" usata da un programma

- Testo del programma, variabili globali
  - Gestiti dal linker
  - la dimensione è fissata e nota al momento della linkaggio
- Variabili automatiche
  - Allocate sullo stack;
  - Allocazione e deallocazione automatiche
  - Dimensione non è nota al momento del linkaggio
- Memoria dinamica
  - Allocazione e deallocazione esplicite
  - Dimensione non è nota al momento del linkaggio
  - malloc(3) implementata grazie a brk(2)

Il modello della memoria di un programma C



Uso dello stack

Questa funzione calcola la funzione fattoriale $n! = 1 * 2 * \cdots * n$.

```
int factorial(n) {
    if (0 == n) return 1;
    return n * factorial(n-1);
}
```

Eseguiamola a mano:

```
  factorial(3)
= return 3 * factorial(2)
= return 3 * 2 * factorial(1)
= return 3 * 2 * 1 * factorial(0)
= return 3 * 2 * 1 * 1
= return 6
```

Quando invochiamo factorial(3), l'invocazione non può terminare prima di avere computato factorial(2), che non può terminare prima di avere computato factorial(1), che dipende da factorial(0). Solo quando arriviamo a factorial(0) tutti i termini della moltiplicazione sono noti, e possiamo risalire a factorial(3). In tutto questo tempo, il programma costruisce una *lista* di tutti i fattori. *Dove è memorizzata questa lista?*

Memoria dinamica

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

```
void * p = malloc(1000); // malloc chiama sbrk(1000)
free(p);                 // la memoria viene conservata
void * q = malloc(500);  // riciclamo la memoria già allocata
void * r = malloc(1000); // malloc chiama sbrk(500)
```

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.

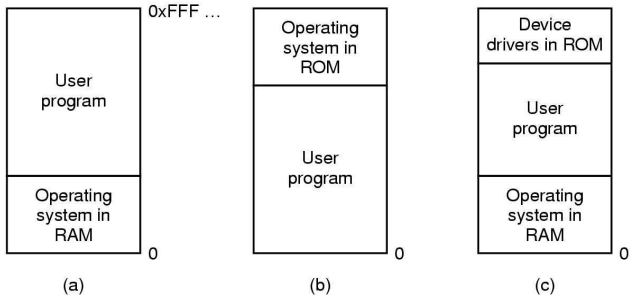
Basic memory management

Tecniche obsolete per s.o. desktop/server, ma valide per:

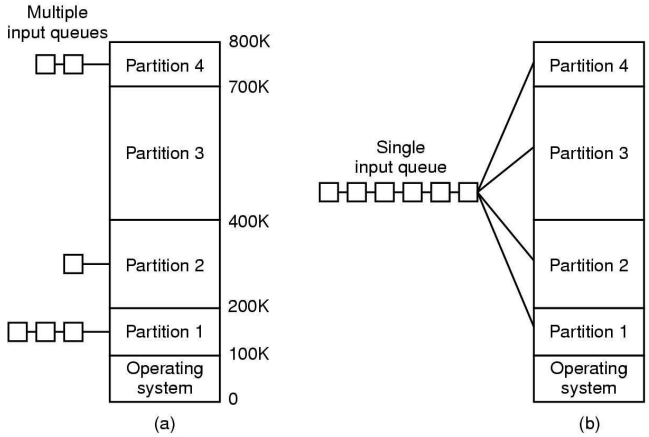
- embedded system
- mobile devices
- smart cards

Basic memory management

Un solo programma alla volta in memoria



Multiprogrammazione con partizioni fisse



Multiprogrammazione con partizioni fisse

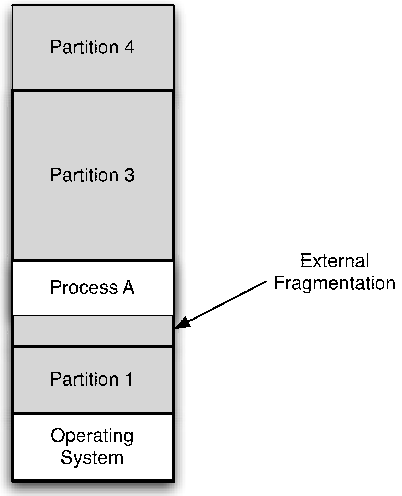
facile da implementare, ma:

problema: come scegliere la partizione più adatta a ciascun processo?

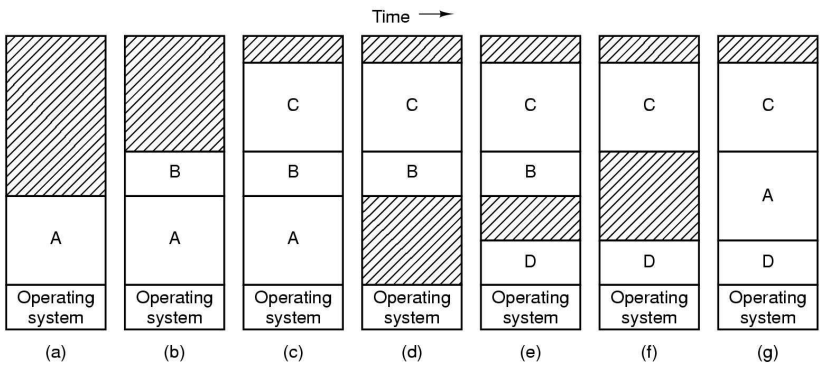
problema: come limitare la frammentazione della memoria?

problema: come rilocare i programmi?

Come scegliere la partizione più adatta?



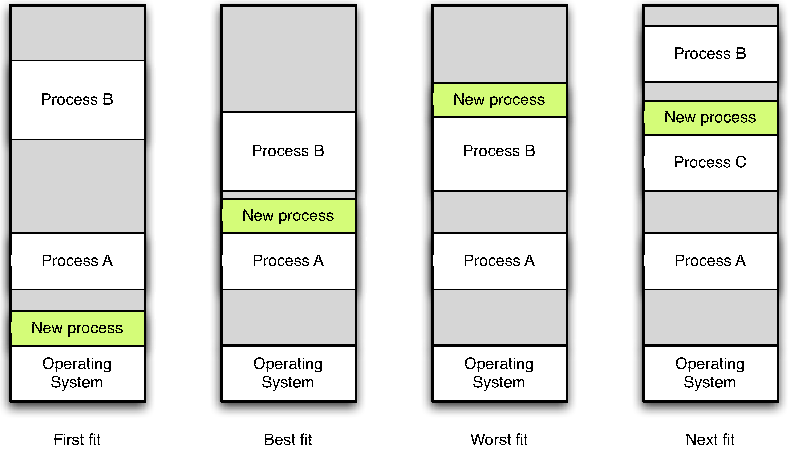
Allocazione con partizioni variabili



La memoria è una lista di "buchi" alternati a "processi"

Problema: frammentazione esterna

Come scegliere la partizione più adatta



Varie strategie, nessuna è chiaramente vincente

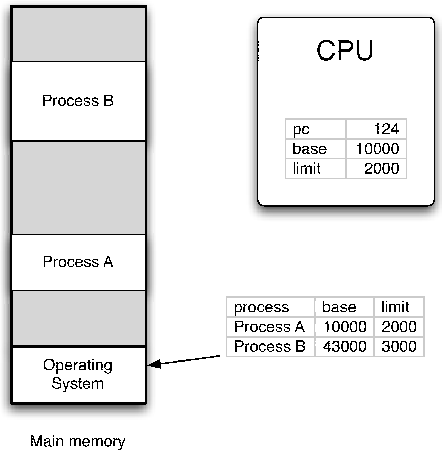
Relocation and protection

```
   :
JUMP   100
   :
```

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

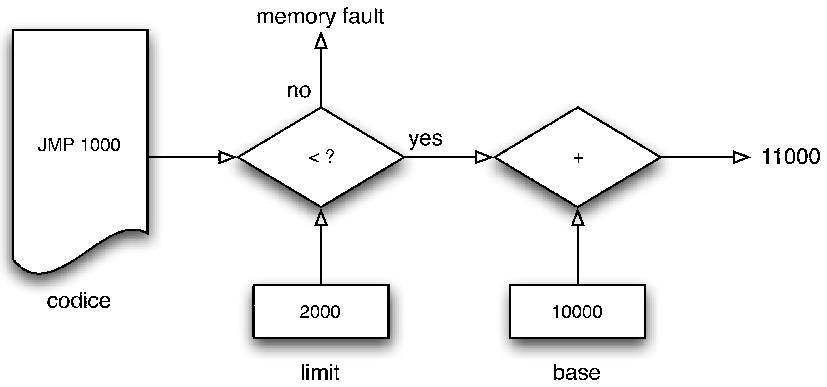
Uso dei registri *base* e *limit* (i)



Una CPU può eseguire un solo processo per volta

Il S.O. aggiorna i registri *base* e *limit* ad ogni cambio di contesto

Uso dei registri *base* e *limit* (ii)



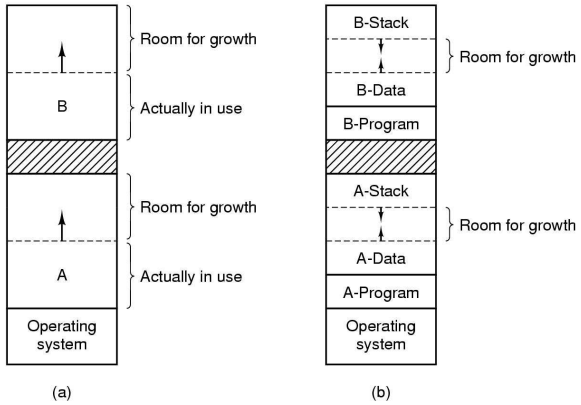
Problemi

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

Compattazione: possibile ma costosa

Problema: cosa fare quando un processo cresce di dimensione?

Permettere a un processo di "crescere"

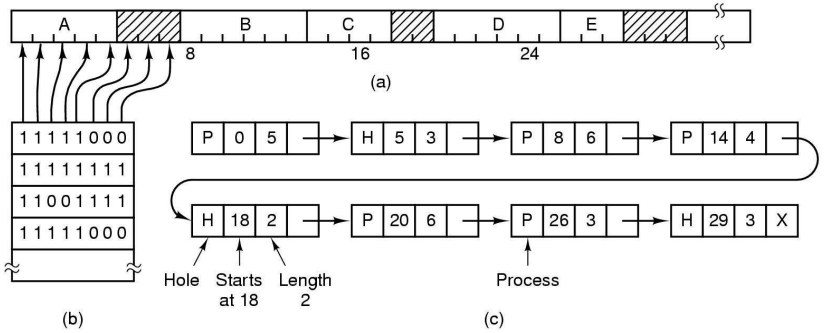


Ora però abbiamo anche una frammentazione *interna*

Che succede se lo spazio previsto per crescere non è sufficiente?

- rilocazione e compattazione
- abort

Gestione della memoria libera



Bitmap versus linked list

Bitmap

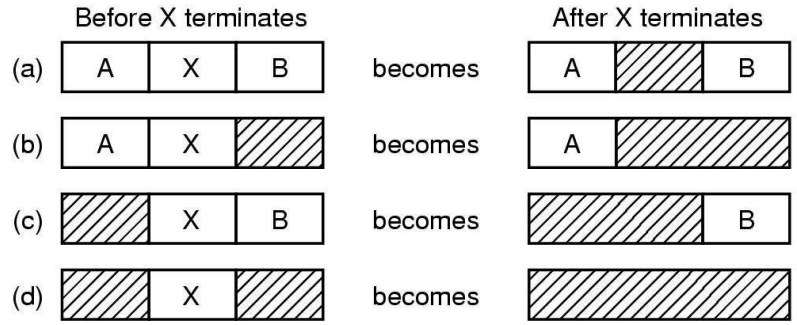
vantaggio: piccole dimensioni

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

Linked list

Collassamento dei buchi alla terminazione di un processo



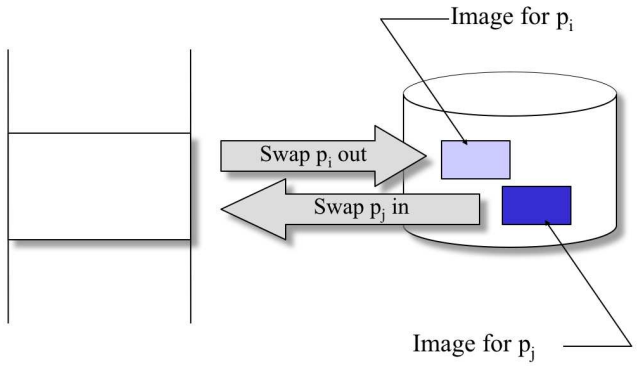
Cosa fare quando la memoria non basta

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

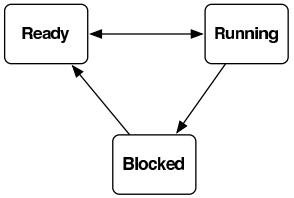
Swap

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



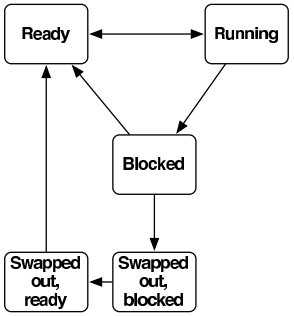
Problema

Come si modifica il diagramma degli stati di un processo se teniamo conto dello swapping?



Problema

Come si modifica il diagramma degli stati di un processo se teniamo conto dello swapping?



Problema

A computer has 1 GB of RAM allocated in units of 64 KB. How many KB are needed if a bitmap is used to keep track of free memory?

Now revisit the previous question using a hole list. How much memory is needed for the list in the best case and in the worst case? Assume the operating system occupies the bottom 512 KB of memory.

Richiami di linguaggio C – puntatori

```
int  a = 3;
int *b = &a;
// come stampare il valore di ‘a’ passando per ‘b’?
printf("%d\n", *b);
// come modificare il valore di ‘a’ passando per ‘b’?
*b = 4;  // ora ‘a’ vale 4
int **c = &b;
// come stampare/modificare il valore di ‘a’ passando per ‘c’?
```

Richiami di linguaggio C – array

```
int a[10];
int *b = &a;
int *c = a;
// che differenza c'è fra il valore di 'b' e quello di 'c'?
// (scrivi un programma C che risponda a questa domanda)
printf("%d %d %d %d\n", a[0], *b, *c, *a);
// che cosa stampa questa riga?  Scrivi un programma per verificare
printf("%d %d %d\n", sizeof(a), sizeof(b), sizeof(c));
// che cosa stampa questa riga?  Scrivi un programma per verificare
// assegna 42 all'elemento di indice 3 di 'a' senza usare le []
*(a + 3) = 42;
```

Richiami di linguaggio C – aritmetica dei puntatori

```
int a = 3;
int *b = &a;
short *c = &((short) a);
double *d = &((double) a);
printf("%d", b);  // stampa 12340
printf("%d", a+1);  // cosa stampa?
printf("%d", b+1);  // cosa stampa?
printf("%d", c+1);  // cosa stampa?
printf("%d", d+1);  // cosa stampa?
```

Altri argomenti da ripassare

- Makefile
- Manipolazione di stringhe: Allocazione, duplicazione, modifica, concatenazione, confronto
- modularizzazione
- organizzazione del codice
- tests