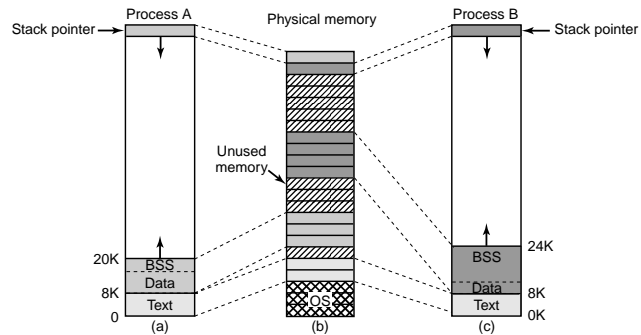


## Lo spazio di indirizzamento di un processo in UNIX



1

## Come funziona malloc(3)?

```
int brk(void *end_data_segment)
```

*brk sets the end of the data segment to the value specified by end\_data\_segment, when that value is reasonable, and the system does have enough memory* — manuale di brk(2)

- end\_data\_segment: indirizzo del primo byte *non* compreso nel data segment desiderato
- return value: il nuovo indirizzo della fine del data segment

2

## Pagine condivise

Il codice eseguibile è read-only

Tutti i processi che eseguono “emacs” usano lo stesso eseguibile /usr/bin/emacs

⇒ risparmiamo memoria se facciamo condividere il segmento “text” fra più processi

Anche le librerie dinamiche sono condivise

3

## Memory mapped files

Permettono di “vedere” un file come un array di caratteri in memoria

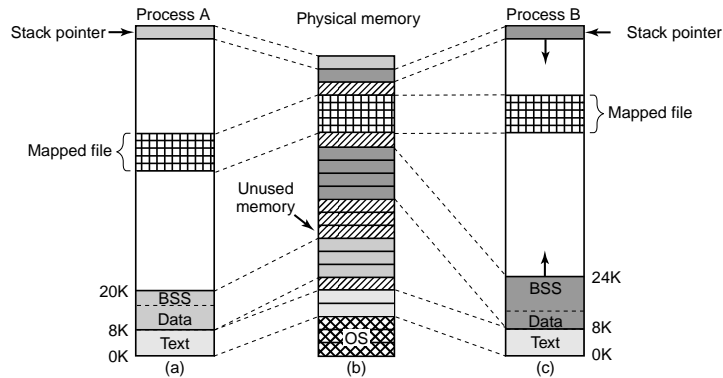
Usi:

- usati internamente dal SO per caricare le librerie dinamiche
- nelle applicazioni, come alternativa a read(2)
  - è più veloce di lseek(2) + read(2)
  - soprattutto per file di grandi dimensioni
- per condividere dati fra processi

Il “backing store” per un mm file è nel file stesso

4

## Memory mapped files



5

## Memory mapped files

```
void * mmap(void *start, size_t length, int prot, int flags, int fd,
off_t offset)
```

The **mmap** function asks to map length bytes starting at offset offset from the file specified by the file descriptor fd into memory, preferably at address start. This latter address is a hint only, and is usually specified as 0. The actual place where the object is mapped is returned by mmap, and is never 0. — manuale di mmap(2)

6

## Memory mapped files

```
void * mmap(void *start, size_t length, int prot, int flags, int fd,
off_t offset)
```

The *prot* argument describes the desired memory protection (and must not conflict with the open mode of the file). It is either PROT\_NONE or is the bitwise OR of one or more of the other PROT\_\* flags.

- PROT\_EXEC Pages may be executed.
- PROT\_READ Pages may be read.
- PROT\_WRITE Pages may be written.
- PROT\_NONE Pages may not be accessed.

7

```
void * mmap(void *start, size_t length, int prot, int flags, int fd,
off_t offset)
```

The *flags* parameter:

MAP\_FIXED Do not select a different address than the one specified. If the specified address cannot be used, mmap will fail. [...] Use of this option is discouraged.

MAP\_SHARED Share this mapping with all other processes that map this object. Storing to the region is equivalent to writing to the file. The file may not actually be updated until msync(2) or munmap(2) are called.

MAP\_PRIVATE Create a private copy-on-write mapping. Stores to the region do not affect the original file. It is unspecified whether changes made to the file after the mmap call are visible in the mapped region.

You must specify exactly one of MAP\_SHARED and MAP\_PRIVATE

8

## Esempio: copia di un file con read(2)

```
#define BUF_SIZE 8192
char buf[BUF_SIZE];

int main() {
    int n;

    n = read(0, buf, BUF_SIZE);
    while (n > 0) {
        if (n != write(1, buf, n)) {
            perror("Errore in scrittura");
            exit(0);
        }
        n = read(0, buf, BUF_SIZE);
    }
    if (n < 0) {
        perror("errore in lettura");
    }
    exit(0);
}
```

9

## Esempio: copia di un file con mmap(2) (a)

```
int main() {
    int n;
    int m;
    off_t filesize;
    struct stat info;
    char * input;

    /* otteniamo la dimensione del file di input con fstat(2) */
    /* nota: STDIN_FILENO è il filedescriptor dell'input (0) */
    if (fstat(STDIN_FILENO, &info) < 0) {
        perror("errore in fstat");
        exit(0);
    }
    filesize = info.st_size;

    /* Mappa il file di input in memoria; il puntatore risultante e'
       "input"
    mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset)
    */
    input = mmap(0, filesize, PROT_READ, MAP_PRIVATE, STDIN_FILENO, 0);
    if (MAP_FAILED == input) {
        perror("errore in mmap");
        exit(EXIT_FAILURE);
    }
}
```

10

## Esempio: copia di un file con mmap(2) b)

```
/* copia l'input nell'output, a blocchi di BUF_SIZE caratteri. */
m = 0;
while (m < filesize) {
    n = min(BUF_SIZE, filesize - m);
    if (n != write(1, input + m, n)) {
        perror("errore in scrittura");
        exit(EXIT_FAILURE);
    }
    m = m + n;
}
exit(0);
}
```

11

## Rimuovere un MM file

```
int munmap(void *start, size_t length);
```

*The munmap system call deletes the mappings for the specified address range, and causes further references to addresses within the range to generate invalid memory references. The region is also automatically unmapped when the process is terminated. On the other hand, closing the file descriptor does not unmap the region.* — manuale di munmap(2)

Nota: posso mappare un file intero, e poi rimuovere un pezzo di mappa in mezzo!

12

## Memoria condivisa

Come un MM file, ma ... senza il file!

- `shmget(2)`
- `shmat(2)`
- `shmdt(2)`

Meccanismo di comunicazione fra processi

La sincronizzazione deve essere ottenuta con altri mezzi (semafori, ecc.)

13

## Allocazione di memoria in Linux

Linux usa "demand paging"

Una richiesta di memoria di un processo è considerata "non urgente"

I page frame sono allocati a un processo solo in risposta a un page fault (cioè solo quando è assolutamente indispensabile)

Quando un processo richiede memoria con `brk(2)` o `mmap(2)` non ottiene frames; ottiene solo il *diritto* di usare un insieme di indirizzi

Il meccanismo è basato sulle *memory regions*

14

## Memory regions

*address space*: insieme degli indirizzi che il processo può usare

*memory region*: un intervallo di indirizzi con determinati *diritti di accesso*

Vengono assegnate nuove regioni a un processo quando:

- il processo "nasce" da una `fork(2)`
- un processo esegue `exec(2)`
- un processo esegue `mmap(2)`
- un processo usa memoria condivisa (`shmget(2)` ecc.)

Le regioni vengono espanse (o contratte) quando

- lo stack cresce
- il processo esegue `brk(2)`, `mmap(2)`, `munmap(2)`

15

## Copy-on-write (COW)

`fork(2)` duplica l'address space di un processo

se il processo è grosso, la copia è molto lenta

prima ottimizzazione: le pagine del *testo* sono condivise perché read-only

seconda ottimizzazione: le pagine dei dati sono condivise in modo *copy-on-write*

quando uno dei processi cerca di modificare una pagina, viene allocato un nuovo frame e la pagina è "sdoppiata"

16

## Implementazione del Copy-on-write in Linux

Durante la fork, tutte le pagine sono marcate read-only

Ma le regioni di dati restano marcate read-write

Quando un processo cerca di modificare una pagina di dati, avviene un page fault

Il S.O. vede che la pagina è *fisicamente* read-only ma *logicamente* read-write

Allora viene allocato un nuovo frame, e le page table di entrambi i processi sono aggiornate

17

## Page fault handling in Linux (overview)

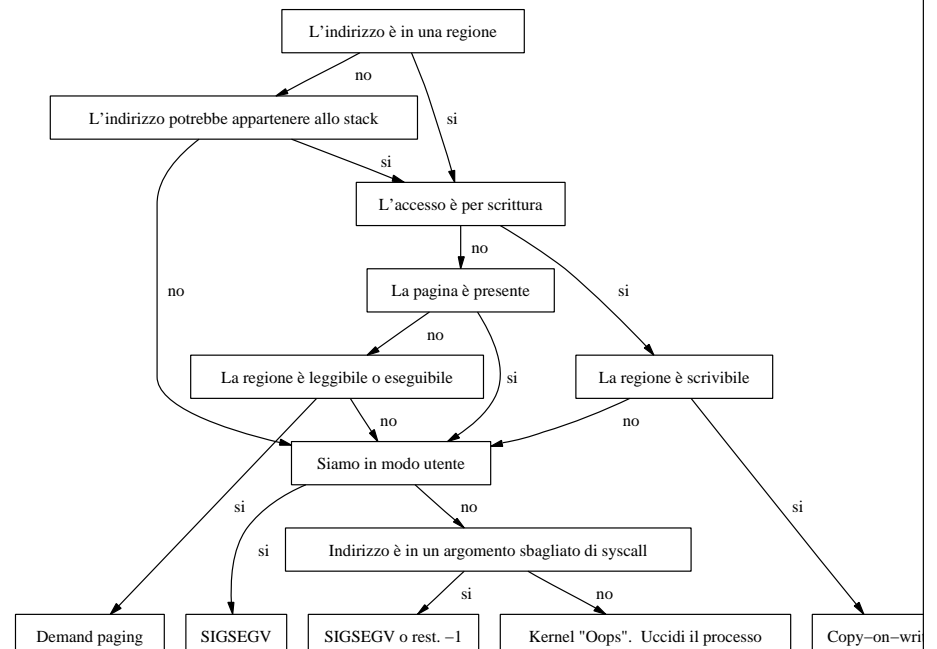


18

## Page fault handling in Linux (detail)

La funzione è `do_page_fault()` in `arch/i386/mm/fault.c`

19



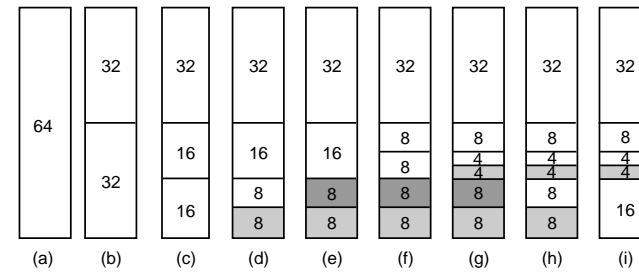
## Allocazione di memoria in Linux

Due meccanismi

- buddy algorithm
- slab allocator

20

## Buddy algorithm



21

## Kernel threads

Alcuni KT collaborano con il Memory Manager

- kswapd mantiene un certo numero di pagine libere

Quando il sistema ha completamente esaurito la memoria, kswapd esegue il famigerato "out-of-memory killer" che, perso per perso, cerca di risolvere il problema eliminando il processo più giovane e più grosso

22

## Esercizio: vm-friendly programs

Verificare la differenza di performance fra programmi che accedono a una grande matrice per righe o per colonne

Nota: la differenza si nota solo se la matrice è così grande da eccedere la dimensione della memoria fisica

23