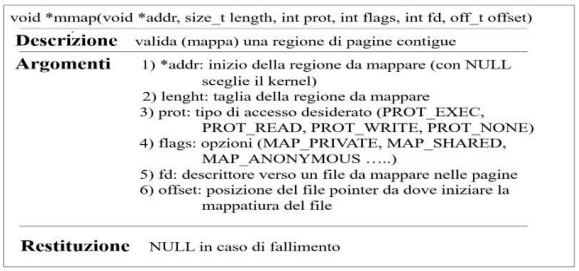
Memoria 2 Mappatura UNIX

mercoledì 19 novembre 2025 14:28

Vediamo come gestire la mappatura della pagine all'interno di un'address space :



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

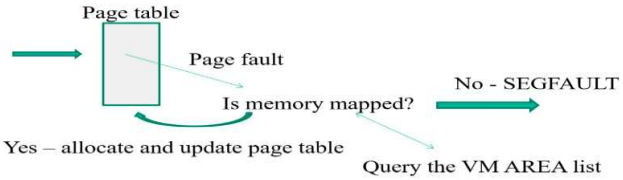
Descrizione valida (mappa) una regione di pagine contigue

Argomenti
1) *addr: inizio della regione da mappare (con NULL sceglie il kernel)
2) lenght: taglia della regione da mappare
3) prot: tipo di accesso desiderato (PROT_EXEC, PROT_READ, PROT_WRITE, PROT_NONE)
4) flags: opzioni (MAP_PRIVATE, MAP_SHARED, MAP_ANONYMOUS …..)
5) fd: descrittore verso un file da mappare nelle pagine
6) offset: posizione del file pointer da dove iniziare la mappatiura del file

Restituzione NULL in caso di fallimento

x86 default page size = 4KB

Che restituisce l'indirizzo della prima zona che è stata mappata. Se PROT_NONE la pagina è inutilizzabile (almeno per ora) . Per quanto riguarda la MAP_SHARED si indica la possibilità di condivisione di una stessa pagina logica (disabilito copy-on-write); se MAP_ANONYMOUS le info all'interno sono NULL (non mappate in memoria fisica: mappate ma non materializzate). **Quindi in generale si possono mappare pagine che contengono anche contenuto di oggetti i/o.** Il primo parametro rappresenta la posizione dove vogliamo mappare le pagine. Attenzione se vado a mappare pagine in zone dove sono già mappate -> ERRORE . Tornando alle pagine anonime : vengono materializzate **solo quando vi è un page fault** (errore su accesso memoria : all'interno della page table non vi è entry corrispondente):



Andiamo a vedere degli esempi :

1. Creazione pagine
   a.
```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/mman.h>

#define PAGE_SIZE (4096)

#define NUM_TARGET_PAGES 10

int main(int argc, char** arv){

        char* buffer;

        //cambio address space
        //in questo caso chiedo 10 pagine di 4096
        //in questo caso copy-on-write
        buffer = (char*)mmap(NULL,PAGE_SIZE*NUM_TARGET_PAGES,PROT_READ|PROT_WRITE,MAP_ANONYMOUS|MAP_PRIVATE,0,0);
        if (buffer == NULL){
                printf("mmap error\n");
                return 1;
        }

        return 0;

}
```
   b. Se compilo torna 0 invece che errore -> le pagine sono state mappate con successo
2. Creazione memoria condivisa

a.

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

#define PAGE_SIZE (4096)

#define NUM_TARGET_PAGES 10000

int gets(char*);

int main(int argc, char** arv){

        pid_t pid;
        char* buffer;
        buffer = (char*)mmap(NULL,PAGE_SIZE*NUM_TARGET_PAGES,PROT_READ|PROT_WRITE,MAP_ANONYMOUS|MAP_SHARED,0,0);
        if (buffer == NULL){
                printf("mmap error\n");
                return 1;
        }
        pid = fork();
        if(pid == -1){
                printf("fork error\n");
                return 2;

        }
        if(pid == 0){
                printf("give me a string:\n");
                gets(buffer);
                return 0;
        }
        wait(NULL);
        printf("%s\n",buffer);
        return 0;
}
```

b. Input acquisito dal child su pagine e lette dal parent in seguito a fork
c. Se levo MAP_SHARED il child scrive su sue pagine logiche che sono copie private non viste dal parent (copy-on-write)

Analogamente all'allocazione esiste la de allocazione :

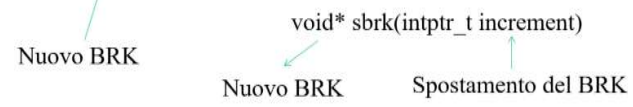
int munmap(void *addr, size_t length)	
Descrizione	de-mappa una regione di pagine contigue
Argomenti	1) *addr: inizio della regione da un-mappare 2) lenght: taglia della regione da un-mappare
Restituzione	-1 in caso di fallimento

Permette quindi di eliminare le pagine sia dall'address space che dalla ram . Vediamo ora invece come cambiare la regola di protezione (da applicare ora) :

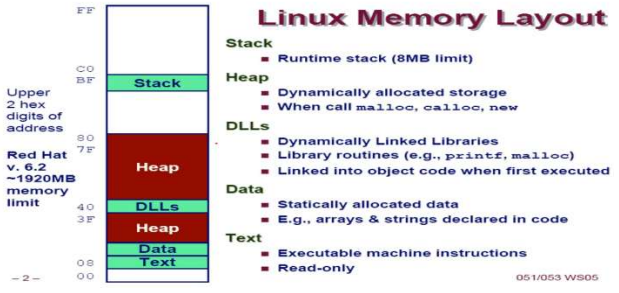
int mprotect(void *addr, size_t length, int prot)	
Descrizione	de-mappa una regione di pagine contigue
Argomenti	1) *addr: inizio della regione da un-mappare 2) lenght: taglia della regione da un-mappare 3) prot: tipo do protezione da applicare (PROT_READ, PROT_WRITE, PROT_EXEC, PROT_NONE)
Restituzione	-1 in caso di fallimento

Con possibile concatenazione delle modalità attraverso la pipe ("|"). Vediamo ora come usate il **program break** (limite memoria <u>logica</u> valida alla fine della sezione .bss fino ad arrivare a trovare pagine valide (dati non inizializzati)) . In dettaglio:

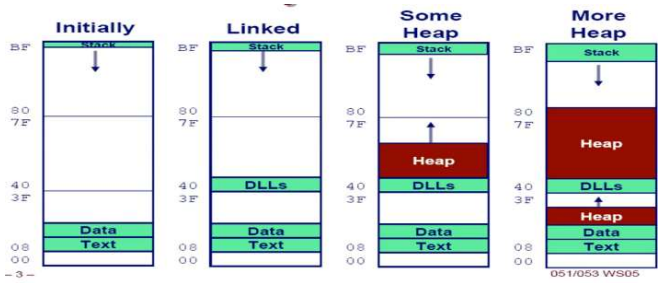
int brk(void *addr)	
Descrizione	definisce il punto di rottura del programma
Argomenti	1) *addr: nuovo punto di rottura
Restituzione	-1 in caso di fallimento



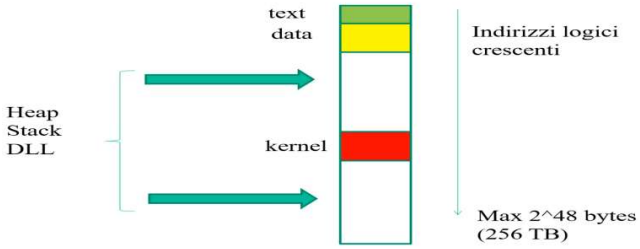
<u>Nota non hanno la dimensione della pagine come paragone</u> (non allineato con la fine di una pagina logica). Vediamo un esempio di un address space (x86 a 32 bit) :



Nel quale lanciando un'applicazione si ha che :



Mentre per quello che riguarda versioni più attuali (x86-64) :



Andiamo ora a vedere altri esempi:

1. Nmap performance
   a.
```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/mman.h>

#define PAGE_SIZE (4096)

#define NUM_TARGET_PAGES 50000

char v[PAGE_SIZE*NUM_TARGET_PAGES]
//il vettore viene allineato a multipli di 4096
__attribute__((aligned(4096)));

int main(int argc, char** arv){
        int i;
        char* buffer;
        buffer = (char*)mmap(NULL,PAGE_SIZE*NUM_TARGET_PAGES,PROT_READ|PROT_WRITE,MAP_ANONYMOUS|MAP_PRIVATE,0,0);
        buffer = v;
#ifndef SAME_PAGE_WRITE
        for (i=0;i<NUM_TARGET_PAGES;i++){
                buffer[i*PAGE_SIZE] = 'f';
        }
#else
        for (i=0;i<NUM_TARGET_PAGES;i++){
                buffer[0] = 'f';
        }
#endif
}
```
   b. Le pagine sono mappate in address space, ma non sono materializzate
   c. Se viene compilata con la macro SAME_PAGE_WRITE si ha che viene quasi eseguita quasi subito nonostante la mappatura in ram ; altrimenti il tempo sale
2. Munmap
   a.
```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/mman.h>

#define PAGE_SIZE (4096)

#define NUM_TARGET_PAGES 10

int main(int argc, char** arv){

        char* buffer;
        int ret;
        buffer = (char*)mmap(NULL,PAGE_SIZE*NUM_TARGET_PAGES,PROT_READ|PROT_WRITE,MAP_ANONYMOUS|MAP_PRIVATE,0,0);
        if (buffer == NULL){
                printf("mmap error\n");
                return -1;
        }
        ret = munmap(buffer,PAGE_SIZE);//unmapping the initial page of the map
        if (ret == -1){
                printf("munmap error\n");
                return -1;
        }
        //using the still mapped pages
        scanf("%s",buffer+PAGE_SIZE);
        printf("%s\n",buffer+PAGE_SIZE);
        return 0;
}
```
   b. Unmappo solo una pagina
   c. Uso la stessa memoria ma mi sposto alla pagina successiva (&pagina+4096)
3. Map file
   a. Pagine associabili a contenuto file
```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/mman.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <unistd.h>

#define PAGE_SIZE (4096)

int gets(char*);

int main(int argc, char** argv){

        pid_t pid;
        int fd;
        char* buffer;
        if (argc!= 2) {
```

b.

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/mman.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <unistd.h>

#define PAGE_SIZE (4096)

int gets(char*);

int main(int argc, char** argv){

        pid_t pid;
        int fd;
        char* buffer;
        if (argc!= 2) {
                printf("Syntax: prog file_name\n");
                return 1;
        }
        fd=open(argv[1], O_RDWR);
                if (fd== -1) {
         printf("open error\n");
         return 2;
        }
        buffer = (char*)mmap(NULL,PAGE_SIZE,PROT_READ|PROT_WRITE,MAP_SHARED,fd,0);
        if (buffer == NULL){
                printf("mmap error\n");
                return 3;
        }
        pid = fork();
        if(pid == -1){
                printf("fork error\n");
                return 4;

        }
        if(pid == 0){
                gets(buffer);
                return 0;
        }
        wait(NULL);
        printf("%s\n",buffer);
        return 0;
}
```

c. **Echo "nomefile" > testo** creo un file e ci scrivo quel testo
d. Se eseguo il programma a stampare stampa bene, ma nel file vado a cambiare il numero dei byte che erano salvati
e. Se rimuoviamo il file ed eseguiamo si ha <u>bus error (core dump)</u>

4. Brk

a.

```c
// moves brk by INCREMENT - possibly also tries to mmap one page on the new brk
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h>

#define PAGE_SIZE 4096

int INCREMENT=0;

char v[4096];
char c;


extern char _edata; // end of initialized data section
extern char _end; // end of non initialized data section


int main(int argc, char** argv){
 void*addr, *oldboundary;
 char c;
 int i;

 if(argc < 2) {
   printf("usage: prog memory-increment [mmap]\n");
   exit(EXIT_FAILURE);
 }
 printf("edata is %l0p\n",& edata);
 printf("end is %l0p\n",& end);

 addr = oldboundary = (void*)sbrk(0);
 printf("old program boundary is at address %x (boundary page number is %d)\n",(unsigned long)addr, (unsigned long)addr/(unsigned long)PAGE_SIZE);

 INCREMENT =  atoi(argv[1]);
 sbrk(INCREMENT);

 addr = (void*)sbrk(0);
 printf("new program boundary is at address %x (boundary page number is %d)\n",(unsigned long)addr, (unsigned long)addr/(unsigned long)PAGE_SIZE);

 if(argv[2] && (strcmp(argv[2],"mmap")==0))
    mmap(addr,PAGE_SIZE,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS|MAP_FIXED,0,0);

 for(i=0;;i++){ // eventually SEGFAULT
  printf("cycle %d accessing address %x\n",i,(unsigned long)(oldboundary+i));
  fflush(stdout);
  c = *(char*)(oldboundary+i);
 }
}
```

b. Il che compilato ed incrementato di uno al massimo arrivo a 4096 indirizzi dopo
c. E notiamo che l'indirizzo di partenza (vecchio) è allineato a 4096 (2^12)
d. Se invece incremento di 4096 vado alla pagina seguente (segmentation fault)