Exemples d'ús de `mmap` en C (sense fitxers)

1 Memòria anònima equivalent a `malloc`

El primer exemple utilitza `mmap` amb `MAP_ANONYMOUS` i `MAP_PRIVATE` per obtenir memòria com faríem amb `malloc`.

```c
#include <sys/mman.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <unistd.h>
#define SIZE 100
int main() {
    char buf[128];
    int *vec = mmap(NULL,SIZE * sizeof(int),PROT_READ | PROT_WRITE,MAP_PRIVATE | MAP_ANONYMOUS,-1,0);
    if (vec == MAP_FAILED) {
        perror("mmap");
        return 1;
    }
    for (int i = 0; i < SIZE; i++) vec[i] = i * 2;
    sprintf(buf,"vec[50] = %d\n", vec[50]); write(1,buf,strlen(buf)); // 100
    munmap(vec, SIZE * sizeof(int));
    return 0;
}
```

Llistat 1: mmap com a malloc

2 Dos processos sense compartició

El segon exemple crea un procés fill amb `fork()`. Amb `MAP_PRIVATE`, les modificacions del fill no afecten el pare.

```c
#include <sys/mman.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>
#include <unistd.h>
#define SIZE 10
int main() {
    char buf[128];
    int *vec = mmap(NULL,SIZE * sizeof(int),PROT_READ | PROT_WRITE,MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    if (vec == MAP_FAILED) { perror("mmap"); return 1; }
    for (int i = 0; i < SIZE; i++) vec[i] = i;
    pid_t pid = fork();
    if (pid == -1) { perror("fork"); return 1; }
    if (pid == 0) { // Fill
        vec[0] = 999;
        sprintf(buf,"Fill: vec[0] = %d\n", vec[0]);write(1,buf,strlen(buf));
        munmap(vec, SIZE * sizeof(int));
        return 0;
    } else { // Pare
        wait(NULL);
        sprintf(buf,"Pare: vec[0] = %d (no canviat)\n", vec[0]);write(1,buf,strlen(buf)); // 0
        munmap(vec, SIZE * sizeof(int));
    }
    return 0;
}
```

Llistat 2: Pare i fill, sense compartició

3 Dos processos amb compartició

Ara utilitzem `MAP_SHARED`. Les modificacions del fill són visibles al pare.

```c
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>
#include <unistd.h>
#define SIZE 10
int main() {
    char buf[128];
    int *vec = mmap(NULL, SIZE * sizeof(int),PROT_READ | PROT_WRITE,MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    if (vec == MAP_FAILED) { perror("mmap"); return 1; }
    for (int i = 0; i < SIZE; i++) vec[i] = i;
    pid_t pid = fork();
    if (pid == -1) { perror("fork"); return 1; }
    if (pid == 0) { // Fill
        vec[0] = 999;
        sprintf(buf,"Fill: vec[0] = %d\n", vec[0]);write(1,buf,strlen(buf));
        munmap(vec, SIZE * sizeof(int));
        return 0;
    } else { // Pare
        wait(NULL);
        sprintf(buf,"Pare: vec[0] = %d (canviat)\n", vec[0]);write(1,buf,strlen(buf)); // 999
        munmap(vec, SIZE * sizeof(int));
    }
    return 0;
}
```

Llistat 3: Pare i fill, amb compartició

4 Compatibilitat de flags a `mmap`

MAP_ANONYMOUS	MAP_PRIVATE	MAP_SHARED	fd	Vàlid?	Notes
0	1	0	≥ 0	✓	Fitxer privat. COW.
0	0	1	≥ 0	✓	Fitxer compartit.
1	1	0	-1	✓	Memòria anònima privada. Com `malloc`.
1	0	1	-1	✓	Memòria anònima compartida entre pare i fill.
×	1	1	×	⊗	Conflicte: PRIVATE i SHARED.
1	0	0	×	⊗	Falta PRIVATE o SHARED.

Taula 1: Compatibilitat de flags principals de `mmap` (Linux).

Combinacions vàlides

- `MAP_SHARED` Mapping compartit d'un fitxer; els canvis són visibles per altres processos que mapejin el mateix fitxer. Obligatori (`fd` vàlid). Mapejar un fitxer de memòria compartida o un arxiu per lectura/escriptura.
- `MAP_PRIVATE` Mapping privat "copy-on-write". Els canvis no afecten el fitxer original. Obligatori `fd` $\geq$ `0`. Carregar un fitxer de dades o un binari de manera privada.
- `MAP_ANONYMOUS | MAP_PRIVATE`. Memòria anònima privada; cada procés té la seva còpia. `fd = -1`
- `MAP_ANONYMOUS | MAP_SHARED`. Memòria anònima compartida entre processos (per exemple després d'un `fork()`). `fd = -1`

Combinacions NO vàlides

- `MAP_ANONYMOUS` sense `MAP_PRIVATE` ni `MAP_SHARED`. Cal indicar un dels dos: el kernel necessita saber si és privat o compartit.
- `MAP_SHARED` i MAP_PRIVATE junts. Mutuament excloents. No pots tenir un mapping alhora privat i compartit.
- `MAP_ANONYMOUS` amb `fd` $\neq$ `-1`. No té sentit: és anònim, per tant no està vinculat a cap fitxer.
- `MAP_PRIVATE` o `MAP_SHARED` sense `MAP_ANONYMOUS`, però amb `fd = -1`. No hi ha fitxer per mapejarr; el kernel retorna `EINVAL`.

4.1 Més sobre adreces

Quan crides a `*mmap_addr = mmap(NULL, 4096, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);` el kernel **reserva una nova regió de memòria virtual** a l'espai d'adreces del procés, sense associar-la a cap arxiu (per `MAP_ANONYMOUS` i `fd = -1`).

- Aquesta regió és totalment independent del **heap** i de la **pila**.
- Es troba a una zona diferent del heap, normalment més alta(a Linux, típicament just per sota de les llibreries compartides i de la pila).

Aquest programa mostra clarament les diferències d'adreces entre pila (*stack*), el *heap* (fent servir `malloc`) i una regió obtinguda amb `mmap(MAP_ANONYMOUS)`.

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <unistd.h>

int main(void) {
    int local_var = 0; // Variable a la stack
    void *stack_addr = &local_var;

    // Memoria dinmica (heap)
    void *heap_addr = malloc(1024);

    // Memoria anonima (mmap)
    void *mmap_addr = mmap(NULL, 4096, PROT_READ | PROT_WRITE,
                           MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    if (mmap_addr == MAP_FAILED) {
        perror("mmap");
        return 1;
    }

    printf("Adr de variable local (stack): %p\n", stack_addr);
    printf("Adr de malloc (heap): %p\n", heap_addr);
    printf("Adr de mmap (anon): %p\n", mmap_addr);

    // Mostrar el rang del heap conegut pel sistema
    void *brk_end = sbrk(0);
    printf("Fi actual del heap (sbrk(0)): %p\n", brk_end);

    // Alliberar
    free(heap_addr);
    munmap(mmap_addr, 4096);

    return 0;
}
```

Llistat 4: heap vs mmap

Exemple de sortida:

```
Adr de variable local (stack): 0x7ffd8a6f9abc
Adr de malloc (heap):          0x5567e0d5a2a0
Adr de mmap (anon):    0x7f8e12a00000
Fi actual del heap (sbrk(0)):     0x5567e0d5b000
```