Laboratorul 5

Comunicare Inter-Proces

1 Memoria Partajată

În mediile de dezvoltare care respectă standardul POSIX (toate variantele UNIX și în mare parte Windows), obiectele de memorie partajată se creează cu ajutorul funcției `shm_open(3)`

```
int shm_open(const char *path, int flags, mode_t mode);
```

având o semantică aproape identică cu funcția de sistem `open(2)` (vezi Laboratorul 2) motiv pentru care `shm_open(3)` este deobicei doar un wraper peste aceasta. Argumentul `path` este de fapt numele obiectului și nu o cale în sistemul de fișiere. Un apel tipic arată astfel

```
char shm_name[] = "myshm";
int shm_fd;

shm_fd = shm_open(shm_name, O_CREAT|O_RDWR, S_IRUSR|S_IWUSR);
if (shm_fd < 0) {
        perror(NULL);
        return errno;
}
```

unde obiectul `"myshm"`, care dacă nu există este creat (`O_CREAT`), este deschis pentru scriere și citire (`O_RDWR`) oferind drepturi asupra lui doar utilizatorului care l-a creat (`S_IRUSR | S_IWUSR`, vezi Laboratorul 2 și `chmod(2)`). Rezultatul este un descriptor `shm_fd` pe care îl putem folosi mai departe în orice funcție ce manipulează obiecte cu ajutorul descriptorilor precum toate funcțiile de sistem sau din biblioteca standard de C pentru fișiere.

O dată creat primul pas este să îi definim dimensiunea cu ajutorul funcției de sistem `ftruncate(2)`

```
size_t shm_size = 1000;

if (ftruncate(shm_fd, shm_size) == -1) {
        perror(NULL);
        shm_unlink(shm_name);
        return errno;
}
```

Aceasta scurtează sau mărește obiectul asociat descriptorului dat conform noii dimensiuni primite în al doilea argument. În exemplul nostru mărește obiectul `shm_fd` de la 0 bytes la 1000.

Funcția `shm_unlink(3)` șterge obiectele create cu funcția `shm_open(3)` primind numele obiectului ca parametru. Aceasta este din nou o extindere firească de la funcția de sistem `unlink(2)` folosită în mod normal pentru a șterege fișiere de pe disk. Un apel tipic poate fii văzut în exemplul `ftruncate(2)` de mai devreme.

Memoria partajată se încarcă în spațiul procesului cu ajutorul funcției de sistem `mmap(2)`.

```
void * mmap(void *addr, size_t len, int prot, int flags,
            int fd, off_t offset);
```

Parametrii sunt

- `addr` – adresa la care să fie încărcată în proces (de obicei aici folosim 0 pentru a lăsa kernelul să decidă unde încarcă)
- `len` – dimensiunea memoriei încărcate
- `prot` – drepturile de acces (`PROT_READ` sau `PROT_WRITE` de obicei)
- `flags` – tipul de memorie (de obicei `MAP_SHARED` astfel încât modificările făcute de către proces să fie vizibile și în celelalte)
- `fd` – descriptorul obiectului de memorie
- `offset` – locul în obiectul de memorie partajată de la care să fie încărcat în spațiul procesului

iar, când se execută cu succes, rezultatul este un pointer către adresa din spațiul procesului la care a fost încărcat obiectul. Altfel, valoarea `MAP_FAILED` este întoarsă și `errno` este setat corespunzător.

Apelurile cele mai des întâlnite sunt de tipul

```
shm_ptr = mmap(0, shm_size, PROT_READ, MAP_SHARED, shm_fd, 0);
if (shm_ptr == MAP_FAILED) {
        perror(NULL);
        shm_unlink(shm_name);
        return errno;
}
```

unde `shm_ptr` va indica către **toată** zona de memorie (`shm_size`) aferentă descriptorului (`shm_fd`) care va fii doar citită (`PROT_READ`) și împărțită cu restul proceselor (`MAP_SHARED`), sau de tipul

```
shm_ptr = mmap(0, 100, PROT_WRITE, MAP_SHARED, shm_fd, 500);
if (shm_ptr == MAP_FAILED) {
        perror(NULL);
        shm_unlink(shm_name);
        return errno;
}
```

unde `shm_ptr` va indica către o **parte** de **100** bytes care începe de la byte-ul **500** din zona de memorie aferentă descriptorului (`shm_fd`) ce va fii doar scrisă (`PROT_WRITE`) și împărțită cu restul proceselor (`MAP_SHARED`).

ATENȚIE: în Linux dimensiunea trebuie să fie multiplu de pagini: `PAGE_SIZE`. Se poate obține și cu `getpagesize(2)`.

Când nu mai este nevoie de zona de memorie încărcată, se folosește funcția `munmap(2)`

```
int munmap(void *addr, size_t len);
```

care primește pointer-ul către zona încărcată în spațiul procesului și dimensiunea ca argumente. Pentru exemplele anterioare am folosi

```
munmap(shm_ptr, shm_size);
```

pentru când a fost încărcată **toată** zona, și

```
munmap(shm_ptr, 100)
```

pentru când a fost încărcată o **parte**.

2 Sarcini de laborator

1. Ipoteza Collatz spune că plecând de la orice număr $n \in \mathbb{N}$ dacă aplicăm următorul algoritm

$$n = \begin{cases} n/2 & \mod(n,2) = 0 \\ 3n+1 & \mod(n,2) \neq 0 \end{cases}$$

   Implementați un program care să testeze ipoteza Collatz pentru mai multe numere date folosind memorie partajată.

   Pornind de la un singur proces părinte, este creat câte un copil care se ocupă de un singur număr și scrie seria undeva în memoria partajată. Părintele va crea obiectul de memorie partajată folosind `shm_open(3)` și `ftruncate(2)` și pe urmă va încărca în memorie întreg spațiul pentru citirea rezultatelor cu `mmap(2)`.

O convenție trebuie stabilită între părinte și fii astfel încât fiecare copil să aibă acces exclusiv la o parte din memoria partajată unde își va scrie datele (ex. împărțim memoria în mod egal pentru fiecare copil). Astfel, fiecare copil va încărca doar zona dedicată lui pentru scriere folosind dimensiunea cuvenită și un deplasament nenul în `mmap(2)`. Părintele va aștepta să termine execuția fiecare copil după care va scrie pe ecran rezultatele obținute de fii săi.

Arătați că cerințele de sus sunt îndeplinite folosindu-vă de `getpid(2)` și `getppid(2)`. Exemplu:

```
$ ./shmcollatz 9 16 25 36
Starting parent 75383
Done Parent 75383 Me 59702
Done Parent 75383 Me 3281
Done Parent 75383 Me 33946
Done Parent 75383 Me 85263
9: 9 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4
2 1.
16: 16 8 4 2 1.
25: 25 76 38 19 58 29 88 44 22 11 34 17 52 26 13 40
20 10 5 16 8 4 2 1.
36: 36 18 9 28 14 7 22 11 34 17 52 26 13 40 20 10 5
16 8 4 2 1.
Done Parent 96028 Me 75383
```

2. În programul anterior folosiți `shm_unlink(3)` și `munmap(2)` pentru a elibera resursele folosite.