

Tutoriat 4 SO



Interprocess Communication

- Processes within a system may be *independent* or *cooperating*
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
 - Information sharing
 - Computation speedup
 - Modularity
 - Convenience
- Cooperating processes need **interprocess communication (IPC)**
- Two models of IPC
 - **Shared memory**
 - **Message passing**



Shared Memory

Fisier 1

Cream si deschidem un obiect de memorie partajata cu numele "tutoriat4", ii modificam dimensiunea, il aducem in spatiul de adresare al procesului curent iar apoi scriem un sir de caractere in acesta.

Apoi, apelam munmap() pentru a sterge maparea din spatiul de adresare curent.

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

#include <sys/mman.h>
#include <sys/stat.h>      /* For mode constants */
#include <fcntl.h>         /* For O_* constants */

#include <unistd.h>
#include <sys/types.h>

#include <sys/mman.h>

#include <string.h>

int main() {

/*

shm_open,  shm_unlink  -  create/open or unlink POSIX shared memory
objects

int shm_open(const char *name, int oflag, mode_t mode);

int shm_unlink(const char *name);

Link with -lrt.

shm_open() creates and opens a new, or opens an existing, POSIX shared
memory object.  A POSIX shared memory object is in effect a handle
which can be used by unrelated processes to mmap(2) the same region of
```

shared memory. The `shm_unlink()` function performs the converse operation, removing an object previously created by `shm_open()`.

- name specifies the shared memory object to be created or opened.

- oflag is a bit mask created by ORing together exactly one of `O_RDONLY` or `O_RDWR` and any of

`O_RDONLY`, `O_RDONLY`, `O_CREAT`, `O_EXCL`, `O_TRUNC`.

The operation of `shm_unlink()` is analogous to `unlink(2)`: it removes a shared memory object name, and, once all processes have unmapped the object, de-allocates and destroys the contents of the associated memory region. After a successful `shm_unlink()`, attempts to `shm_open()` an object with the same name fail (unless `O_CREAT` was specified, in which case a new, distinct object is created).

Return:

On success, `shm_open()` returns a nonnegative file descriptor. On failure, `shm_open()` returns -1. `shm_unlink()` returns 0 on success, or -1 on error.

*/

```
int shm_fd = -1;
```

```
shm_fd = shm_open("tutoriat4", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR); //  
creeaza o mem partajata (sau deschide) cu numele "tutoriat4"
```

```
if (shm_fd == -1) {
```

```
    puts("eroare la shm_open\n");
```

```
    return errno;
```

```
}
```

/*

truncate a file to a specified length

```
int truncate(const char *path, off_t length);
```

```
int ftruncate(int fd, off_t length);
```

The `truncate()` and `ftruncate()` functions cause the regular file named by path or referenced by fd to be truncated to a size of precisely

length bytes.

If the file previously was larger than this size, the extra data is lost. If the file previously was shorter, it is extended, and the extended part reads as null bytes ('\0').

On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

```
*/
```

```
int shm_size = 8;
int ret = ftruncate(shm_fd, shm_size);
if (ret == -1) {
    printf("eroare la truncate\n");
    return errno;
}
```

```
/*
```

mmap, munmap - map or unmap files or devices into memory

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

mmap() creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in addr. The length argument specifies the length of the mapping (which must be greater than 0).

If addr is NULL, then the kernel chooses the (page-aligned) address at which to create the mapping; this is the most portable method of creating a new mapping.

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.

On success, `mmap()` returns a pointer to the mapped area. On error, the value `MAP_FAILED` (that is, `(void *) -1`) is returned, and `errno` is set to indicate the cause of the error.

On success, `munmap()` returns 0. On failure, it returns -1, and `errno` is set to indicate the cause of the error (probably to `EINVAL`).

- `addr` - adresa la care să fie încărcată în proces
- `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 spațiu, iul procesului

*/

```
char* adresa_fisier = mmap(NULL, shm_size, PROT_WRITE, MAP_SHARED, shm_fd, 0);
```

/*

NAME

`memcpy` - copy memory area

SYNOPSIS

```
#include <string.h>
```

```
void *memcpy(void *dest, const void *src, size_t n);
```

DESCRIPTION

The `memcpy()` function copies `n` bytes from memory area `src` to memory area `dest`. The memory areas must not overlap. Use `memmove(3)` if the memory areas do overlap.

```

RETURN VALUE
    The memcpy() function returns a pointer to dest.

*/

memcpy(adresa_fisier, "1234567\0", shm_size);
munmap(adresa_fisier, shm_size);

return 0;
}

```

Fisier 2

Deschidem obiectul de memorie partajata cu numele "tutoriat4", il aducem in spatiul de adresare al procesului curent, accesam sirul de caractere din obiectul de memorie partajata. Apoi, apelam munmap() pentru a sterge maparea din spatiul de adresare curent si shm_unlink() pentru a elimina obiectul.

```

#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>
#include <errno.h>

int main()
{
    int shm_fd;
    int shm_size = 8;
    char* adresa_fisier = NULL;

    shm_fd = shm_open("tutoriat4", O_RDWR, 0);
    if(shm_fd == -1)
    {
        puts("Nu am putut deschide sau nu exista mem partajata cu acest nume\n");
        return errno;
    }
}

```

```

    puts("Am deschis mem partajata");

    adresa_fisier = mmap(NULL, shm_size, PROT_READ, MAP_SHARED, shm_fd, 0);
    printf("Am citit: %s\n", adresa_fisier);

    munmap(adresa_fisier, shm_size);
    shm_unlink("tutoriat4");

    return 0;
}

```

Executare:

```

cosmin@cosmin-Legion-Y540-15IRH: ~/Documents/Facultate/An3-Sem1/Tutoriat S0/4
cosmin@cosmin-Legion-Y540-15IRH:~/Documents/Facultate/An3-Sem1/Tutoriat S0/4$ ls
tutoriat4-1.c  tutoriat4-2.c
cosmin@cosmin-Legion-Y540-15IRH:~/Documents/Facultate/An3-Sem1/Tutoriat S0/4$ gcc tutoriat4-1.c -o 1 -lrt
cosmin@cosmin-Legion-Y540-15IRH:~/Documents/Facultate/An3-Sem1/Tutoriat S0/4$ gcc tutoriat4-2.c -o 2 -lrt
cosmin@cosmin-Legion-Y540-15IRH:~/Documents/Facultate/An3-Sem1/Tutoriat S0/4$ ./1
Am deschis mem partajata
Am citit: 1234567
cosmin@cosmin-Legion-Y540-15IRH:~/Documents/Facultate/An3-Sem1/Tutoriat S0/4$ ./2
Nu am putut deschide sau nu exista mem partajata cu acest nume
cosmin@cosmin-Legion-Y540-15IRH:~/Documents/Facultate/An3-Sem1/Tutoriat S0/4$

```

Observatii:

1. Pentru compilare, trebuie sa folosim flag-ul `-lrt`
2. Procesul 2 apeleaza `shm_unlink()`, distrugand obiectul de memorie partajata, motiv pentru care daca executam din nou al doilea program, nu mai exista un obiect de memorie partajata cu numele "tutoriat4" pe care sa il putem deschide.

Exercitiu:

Transformati problema cu sirurile Fibonacci de la tutoriatul 3 astfel incat procesul parinte sa adune rezultatele fiecarui proces din obiectul de memorie partajata.

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <sys/stat.h>

```

```

#include <unistd.h>
#include <errno.h>
#include <sys/wait.h>
#include <string.h>

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

    int shm_fd;
    char* shm_name = "shm_fibo";

    shm_fd = shm_open(shm_name, O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);

    if (shm_fd == -1) {
        perror("Eroare la shm_open\n");
        return errno;
    }

    int page_size = getpagesize();
    int total_size = argc * page_size; // ii alocam un spatiu de page_size
    // fiecarui proces

    if (ftruncate(shm_fd, total_size) == -1) {
        perror("Eroare la truncate\n");
        shm_unlink(shm_name); // stergem obiectul
        return errno;
    }

    printf("Starting parent %d\n", getpid());
    for (int i = 1; i < argc; i++) {
        pid_t child = fork(); // un copil pt fiecare task
        if (child < 0) {
            perror("Eroare la fork");
            return -1;
        }
        else if (child == 0)
        {
            char* shm_ptr;
            shm_ptr = mmap(NULL, page_size, PROT_WRITE, MAP_SHARED, shm_fd,
                (i - 1) * page_size);

```



```

        // copilului i i se va mapa o bucata de lungime egala cu
page_size
        // mai exact, de la (i-1)*page_size la i*page_size
        if (shm_ptr == MAP_FAILED) {
            perror("Eroare la mmap, din procesul copil");
            shm_unlink(shm_name);
            return errno;
        }
        int n = atoi(argv[i]);
        int car_scrise;
        car_scrise = sprintf(shm_ptr, "%d: ", n);
        shm_ptr += car_scrise; // deplasam pointerul cu nr de caractere
scrise
        if (n < 1) {
            car_scrise = sprintf(shm_ptr, "n has to be > 0\n"); // aici
am facut direct
            shm_ptr += car_scrise;
        }
        else {
            int f1 = 0, f2 = 1, i;
            shm_ptr += sprintf(shm_ptr, "%d ", f1);
            for (i = 1; i < n; i++) {
                shm_ptr += sprintf(shm_ptr, "%d ", f2);
                int next = f1 + f2;
                f1 = f2;
                f2 = next;
            }
            shm_ptr += sprintf(shm_ptr, "\n");
        }
        printf("Done. Parent = %d, Me = %d\n", getppid(), getpid());
        munmap(shm_ptr, page_size);
        exit(0); //normal process termination
    }
}
for (int i = 1; i < argc; i++) {
    wait(NULL);
}
for (int i = 1; i < argc; i++) {
    // incarcam pe rand in memorie "bucatica" fiecarui proces copil

```

```

    char* shm_ptr = mmap(NULL, page_size, PROT_READ, MAP_SHARED,
shm_fd, (i-1)*page_size);
    if(shm_ptr == MAP_FAILED)
    {
        perror("Eroare la mmap din parinte\n");
        shm_unlink(shm_name);
        return errno;
    }
    printf("%s", shm_ptr);
    munmap(shm_ptr, page_size);
}
printf("Parent done. Parent = %d, Me = %d\n", getppid(), getpid());

shm_unlink(shm_name);

return 0;
}

```

Pipes:

<https://www.usna.edu/Users/cs/wcbrown/courses/IC221/classes/L13/Class.html>

https://www.tutorialspoint.com/inter_process_communication/inter_process_communication_pipes.htm