Shared Memory (1/2)

Sistemas de Computadores

2018/2019

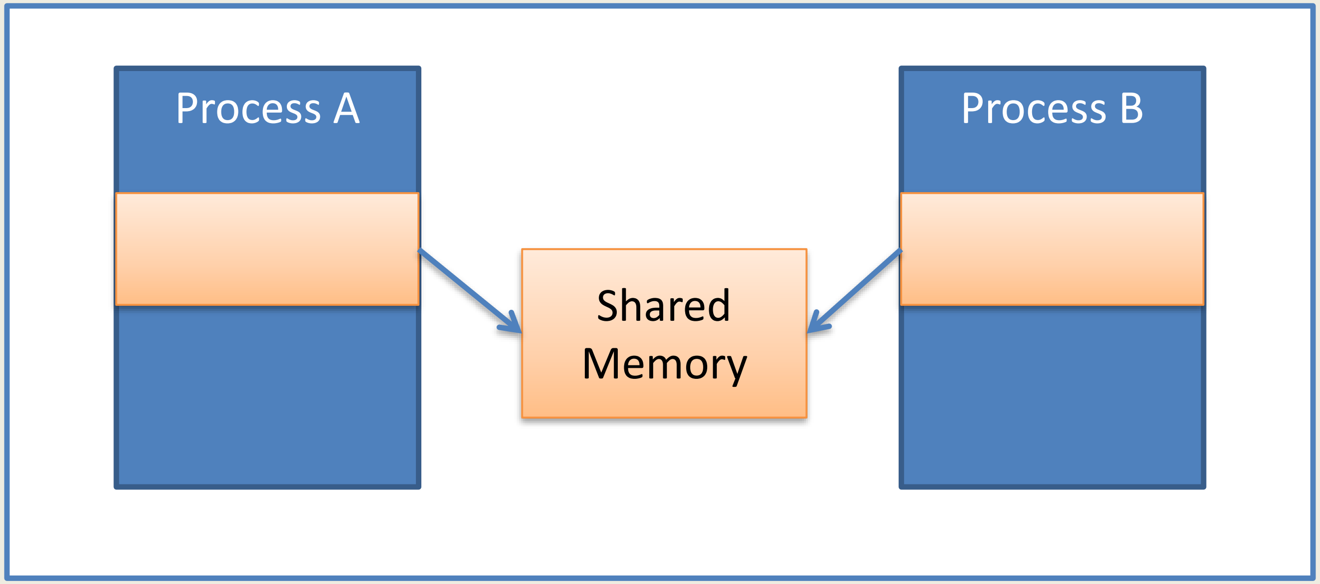
Luís Miguel Pinho / Luis Ferreira / Luís Nogueira / Nuno Pereira

Introduction

- Processes are **isolated** one from the other
  - In general it is not possible for one process to access the memory area of another process
- There are cases where this restriction is not convenient
- It is therefore possible to define a memory area which can be made part of the memory space of two or more processes

Shared memory

- Memory region which is shared by several processes
- Processes with or without an hierarchical link can use this region to communicate



Advantages

- Random access
  - Possible to directly access and change any part of the region
- Efficiency
  - Accesses are direct (contrarily to pipes, for instance), without the intervention of the OS
  - It is the fastest communication mechanism between processes

Disadvantages

- Need to synchronize
  - Concurrent processes need to synchronize access to shared memory
- Pointer sharing
  - We need to assume that the pointers used by one process are **valid only in that particular process**
  - This makes it difficult to implement some data structures (e.g. linked lists or trees)

Introduction

- In Linux/UNIX there are two APIs of shared memory (SHM)

- System V (**shmget(2)**)
  - Original mechanism, still very used
  - Part of this course until a few years ago

- POSIX (**shm_open(3)**)
  - Developed to be more simple to use than older APIs (such as System V)

POSIX Shared Memory in Linux

- Implemented as temporary files
  - tmpfs file systems
  - Persistent (exists until explicitly deleted, or system reboot)
- A process may map a shared memory area (in POSIX referred frequently as Object), make changes, and disconnect
  - These changes are visible for any process that maps this shared memory object

POSIX Shared Memory in Linux

- Create
  - **`shm_open(), ftruncate(), mmap()`**
- Use the memory
  - With a pointer similarly to any dynamic memory
- Remove (next class)
  - **`munmap(), close(), shm_unlink()`**

POSIX Shared Memory in Linux

- Usual headers
  - <fcntl.h> - file control options
  - <sys/types.h> - data types used by the API
  - <sys/stat.h> - constants used for opening
  - <sys/mman.h> - declarations related with management of shared memory : **`shm_*`** and **`mmap()`**

POSIX Shared Memory in Linux

- Compiling
  - In Linux, the functions are in what is called “POSIX realtime library (librt)”, so it is necessary to compile and link with the library (rt)
- Exemplo:

```
gcc --Wall -o shm_test shm_test.c -lrt
```

Create

1. **`shm_open()`** : Creates and opens a shared memory area
   - If already exists, just opens to use
   - Returns a file descriptor to be used in other functions

2. **`ftruncate()`** : Defines the size

3. **`mmap()`** : Maps the shared area in the process address space

shm_open()

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

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

- Creates and opens a shared memory area, or opens an existing one
- Returns a file descriptor (the shared memory), or -1 in case of error

shm_open()

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

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

- **name**: name to be used to identify the area
  - Must start with '/'

shm_open()

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

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

- **oflag**: Creation options
  - **O_CREAT**: creates, if not existing, or use existing
    - Without O_CREAT tries to use existing (error if none)
  - **O_EXCL**: (with **O_CREAT**) **just creates**, error if already exists
  - **O_TRUNC**: eliminates contents already existing in the shared memory area
- Access Flags
  - **O_RDONLY**: open for reading only
  - **O_RDWR**: open for reading and writing (no flag to just write)

shm_open()

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

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

- **mode**: Defines permissions, as per Linux usual style
  - RWX for user/group/others
  - Makes AND with the complement of the process umask
  - Should be 0 (zero) if opening an existing area

ftruncate()

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

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

- Defines the size of the area and initializes it to zero
  - Must be used after **shm_open()**, as memory is created with size zero
  - If area already exists, only changes if size is different
    - If new size > than existing, the size is increased with zero in the new area
    - If new size < than existing, size is reduced (losing the extra data)
- Returns 0 if successful, or -1 in case of error (*errno* has the error)

ftruncate()

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

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

- **fd**:
  - File descriptor (returned by **shm_open()**)
- **length**:
  - size, in bytes, of memory object

mmap()

```
#include <sys/mman.h>

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

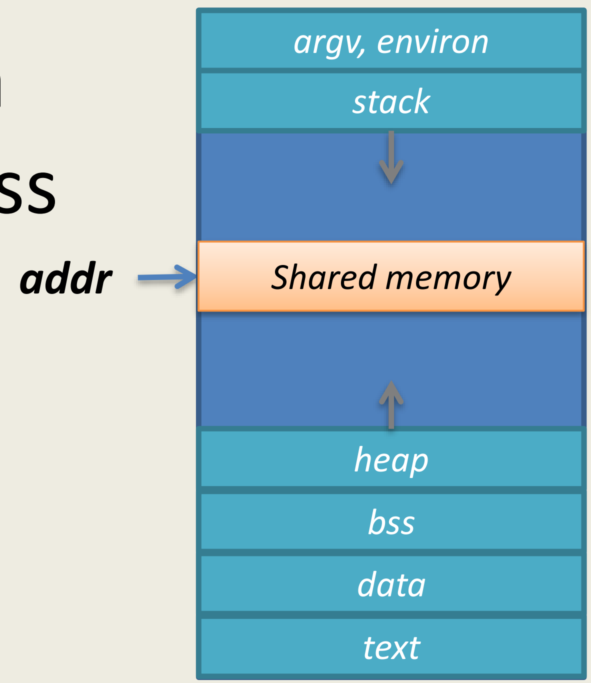
- Maps a shared memory object in the process address space
  - In practice, we are required to handle arguments ***lenght*** and ***prot***
- Returns a pointer to the object or **MAP_FAILED** (**(void *) -1**), with *errno* set with the error

mmap()

```
#include <sys/mman.h>

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

- **`addr`**: allows to request the area to be mapped in a specific address
  - ***NULL*** in the usual case to let the OS define the address



mmap()

```
#include <sys/mman.h>

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

- **`length`**: Required size to be used
  - ≤ than size defined by **`ftruncate()`**
  - OS usually rounds to a multiple of the size of a memory page

mmap()

```
#include <sys/mman.h>

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

- **prot**: protection flags
  - **PROT_READ**: just read
  - **PROT_READ | PROT_WRITE**: read and write

- Must be consistent with **shm_open()**
  - Do not use **O_RDONLY** in **shm_open()** and then **PROT_READ | PROT_WRITE** in **mmap()**

mmap()

```
#include <sys/mman.h>

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

- **flags**: control behaviour of **mmap()**
  - In the case of shared memory, always use **MAP_SHARED**, to allow changes be seen by other processes
- **fd**: File descriptor
  - Returned by **shm_open()**

mmap()

```
#include <sys/mman.h>

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

- **`offset`**: beginning of mapping in the memory area
  - Usually zero, to use all memory area from the beginning
  - Must be a multiple of memory page

Create and open

```
1.int fd;
2.void * addr;
3.fd = shm_open(“/shmtest”, O_CREAT|O_EXCL|
   O_RDWR, S_IRUSR|S_IWUSR);
4.ftruncate (fd, 100);
5.addr = mmap(NULL, 100,PROT_READ|PROT_WRITE,
   MAP_SHARED, fd, 0);
```

Line 3: Creates (`O_CREAT`) shared memory area (with error if existis because of `O_EXCL`) with name “`/shmtest`”, and read/write permissons (`O_RDWR`), and open for user to read (`S_IRUSR`) and write (`S_IWUSR`).

Criar e Abrir Memória Partilhada

```
1.int fd;
2.void * addr;
3.fd = shm_open(“/shmtest”, O_CREAT|O_EXCL|
   O_RDWR, S_IRUSR|S_IWUSR);
4.ftruncate (fd, 100);
5.addr = mmap(NULL, 100,PROT_READ|PROT_WRITE,
   MAP_SHARED, fd, 0);
```

Line 4: Defines a 100 bytes size

Criar e Abrir Memória Partilhada

```
1.int fd;
2.void * addr;
3.fd = shm_open(“/shmtest”, O_CREAT|O_EXCL|
   O_RDWR, S_IRUSR|S_IWUSR);
4.ftruncate (fd, 100);
5.addr = mmap(NULL, 100,PROT_READ|PROT_WRITE,
   MAP_SHARED, fd, 0);
```

Line 5: Maps the area in na address decided by the OS (**NULL**), requring to use all the size (**100**), with read and write permissions (**PROT_READ | PROT_WRITE**) , consistente with open (continues)

Criar e Abrir Memória Partilhada

```
1.int fd;
2.void * addr;
3.fd = shm_open("/shmtest", O_CREAT|O_EXCL|
   O_RDWR, S_IRUSR|S_IWUSR);
4.ftruncate (fd, 100);
5.addr = mmap(NULL, 100,PROT_READ|PROT_WRITE,
   MAP_SHARED, fd, 0);
```

Line 5: (continued) Area is shared (**`MAP_SHARED`**), and it refers to the previous open (**`fd`**). All memory área will be used so starting from beginning (offset is **0**).

Use

- Use as normal dynamic memory
  - The pointer obtained with **`mmap()`** can be used the same way as pointers given by **`malloc()`**
  - It is common to have a structure defined for the data in the memory area
- Next example
  - Shared memory between unrelated processes (no parent - child)
  - Reading / writing in a data structure
  - No synchronization - writer needs to be executed first

Escritor

```c
typedef struct {
      int var1;
      int var2;
} shared_data_type;

int main(int argc, char *argv[]) {
   int fd, data_size = sizeof(shared_data_type);
   shared_data_type *shared_data;
   int fd = shm_open("/shmtest", O_CREAT|O_EXCL|O_RDWR,
     S_IRUSR|S_IWUSR);
   ftruncate (fd, data_size);
   shared_data = (shared_data_type*)mmap(NULL,
     data_size,PROT_READ|PROT_WRITE,MAP_SHARED,fd,0);
   shared_data->var1 = 100;
   shared_data->var2 = 200;
}
```

Write

Leitor

```
typedef struct {
      int var1;
      int var2;
} shared_data_type;

int main(int argc, char *argv[]) {
   int fd, data_size = sizeof(shared_data_type);
   shared_data_type *shared_data;
   int fd = shm_open("/shmtest", O_RDWR,
     S_IRUSR|S_IWUSR);
   ftruncate (fd, data_size);
   shared_data = (shared_data_type *)mmap(NULL,
     data_size,PROT_READ|PROT_WRITE,MAP_SHARED,fd,0);
   printf("%d\n", shared_data->var1);
   printf("%d\n", shared_data->var2);
}
```

No O_CREAT

Read

Use

- New example with synchronization with active wait (spinning)
  - Processes are related
  - Parent is **writer** and son is **reader**

```
typedef struct {
      int var1;
      int var2;
      int new_data;
} shared_data_type;

int main(int argc, char *argv[]) {
   int fd, data_size = sizeof(shared_data_type);
   shared_data_type *shared_data;
   fd = shm_open("/shmtest", O_CREAT|O_EXCL|O_RDWR,
     S_IRUSR|S_IWUSR);
   ftruncate (fd, data_size);
   shared_data = (shared_data_type *)mmap(NULL,
     data_size,PROT_READ|PROT_WRITE,MAP_SHARED,fd,0);
```

Utilizar a Memória Partilhada

```
    shared_data->new_data = 0;
    pid=fork()
    if (pid > 0) { /* writer*/
        shared_data->var1 = 100;
        shared_data->var2 = 200;
        shared_data->new_data = 1;
        wait();
    }
    else {            /* reader*/
        while(!shared_data->new_data);
        printf("%d\n", shared_data->var1);
        printf("%d\n", shared_data->var2);
    }
}
```

Write (Parent)

Synchronization

Read (Son)

Exercise TP5.1

- Create two unrelated processes:
  - Writer: writes string in shared memory
  - Reader: read and print the message.
- Reader executed after writer

Exercise TP5.2

- Number 5 lotary:
  - Writer: Places random numbers [1] [1,5] in shared memory.
  - Reader: waits for a new number (spinning in a flag), and prints the number.
  - Must guarantee that writer waits for reader to finish processing a number before generating a new one (writer also spins in a flag)
  - Both finish when number is 5. Reader prints how many numbers received.

[1] Random numbers in an interval [1,N] can be generated with:

```
#include <stdio.h>
#include <stdlib.h>
...
/* Generator seed */
srand((unsigned) time(NULL));
...
/* generate number */
num = rand()% N + 1;
```

Solution

```
typedef struct {
    int loto;
    int canRead;
    int canWrite;
} shared_data_type;

int main(int argc, char *argv[]) {
  int fd, data_size = sizeof(shared_data_type);
  shared_data_type *shared_data;
  fd = shm_open("/shmtest", O_CREAT|O_EXCL|O_RDWR,
   S_IRUSR|S_IWUSR);
  ftruncate (fd, data_size);
  shared_data = (shared_data_type *)mmap(NULL,
   data_size,PROT_READ|PROT_WRITE,MAP_SHARED,fd,0
  );
```

Writer process

```
shared_data->canRead = 0;
shared_data->canWrite = 1;
pid=fork()
if (pid > 0) {
    ramdomNum = 0;
    while (ramdomNum != 5) {
        randomNum = generateRandom();
        while(shared_data->canWrite == 0);
        shared_data->canWrite = 0;
        shared_data->loto = randomNum;
        shared_data->canRead = 1;
    }
Else {
```

Reader process

```
ramdomNum = 0
while (ramdomNum != 5)
 {
      while(shared_data->canRead == 0);
      shared_data->canRead = 0;
      randomNum = shared_data->loto;
      shared_data->canWrite = 1;
   }
```