Operating Systems 2017/2018

TP Class 08 – Message Queues and Memory Mapped Files

Vasco Pereira (vasco@dei.uc.pt)
Dep. Eng. Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra

Slides based on previous versions from Bruno Cabral, Paulo Marques and Luis Silva.

operating system
noun
the collection of software that directs a computer's operations, controlling and scheduling the execution of other programs, and managing storage, input/output, and communication resources.

Abbreviation: OS

Source: Dictionary.com

FCTUC FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE DE COIMBRA

Updated on
03 November 2017

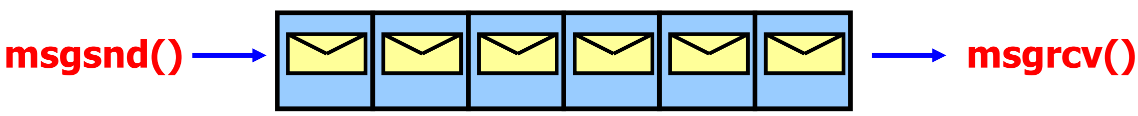
MESSAGE QUEUES

Types of communication

- Streams represent “a flow” of bytes. There are no fixed data boundaries.
  - The sender requests the transmission of N bytes
  - The data starts flowing, the receiver starts getting it
  - The receiver may get several chunks of less then N bytes
- Messages represent a complete fixed structure of data
  - It’s like sending a letter. Either you get if fully or you don’t. You don’t get “half a letter”.

Message Queues

- Another IPC mechanism
  - Based on <u>messages</u>, not on data streams
- Completely <u>asynchronous</u>
  - A process can start executing, write some messages to a message queue and die. Later, another process can come alive and receive them
    - Does not require that both sender and receiver are present at the same time!
  - Message queues are maintained by the operating system. <u>They are not destroyed if a process dies</u>!



Message Queues – System V

- `int msgget(key_t key, int flags)`
  - Obtains an identifier to an existing message queue or creates a new one.
    - "key" can be IPC_PRIVATE (which creates a new unique identifier), or an existing identifier. ftok() can be used to generate a number based on a filename.
    - "flags", normal mode flags. When ORed with IPC_CREAT creates a new one.
    - -1 is returned on error

- `int msgctl(int msqid, int cmd, struct msqid_ds* buff)`
  - Provides a variety of control operations on the message queue.
    - "msqid" is the value returned by msgget()
    - "cmd" is the command (most usually: IPC_RMID to remove it)
    - "buff" a structure used in some control operations

Message Queues – System V (2)

- `int msgsnd(int msqid, const void* message, size_t length, int flags)`
  - Puts a message in a message queue
  - It appends a copy of the message pointed to by message to the message queue specified by *msqid*
    - "msqid" is the value returned by msgget()
    - "message" is a pointer to the message to send
    - "length" represents the length of the **payload** of the message (**not the total**)
    - "flags": 0 or IPC_NOWAIT (non-blocking)
    - The calling process must have write permission on the message queue in order to send a message
    - On error returns -1

Message Queues – System V (3)

- Message Payload
  - In System V a message can be anything. But, it must <u>always</u> have a “long” integer in the beginning
    - This long is called a **message type identifier**

```
typedef struct
{
  long  msgtype;
  int   first;
  int   second;
} numbers_message;
```

Message type (must be >0)!

Payload (may be anything)

Message Queues – System V (4)

- `int msgrcv(int msqid, void* message, size_t length, long msgtype, int flags)`
  - Retrieves a message from a message queue - removes a message from the queue specified by msqid and places it in the buffer pointed to by message.
    - “msqid” is the value returned by msgget()
    - “message” is a pointer to the buffer where the message will be received
    - “length” represents the maximum payload (in bytes) we are willing to receive
    - “msgtype” represent the type of message to receive
      - If 0 the first message in the queue is returned (FIFO)
      - If > 0 the first message in the queue of type *msgtype* is read
      - If < 0 the first message in the queue with the lowest type less than or equal to the absolute value of *msgtype* will be read.
    - “flags”: 0 or IPC_NOWAIT (non-blocking)

    - The calling process must have read permission to receive a message.
    - On error returns -1

mq_pong.c (1)

```c
typedef struct {
  long mtype;
  int  first, second;
} numbers_msg;

// Message queue id
int id;

void cleanup(int signum) {
  msgctl(id, IPC_RMID, NULL);
  exit(0);
}

void main(int argc, char* argv[]) {
  assert( (id = msgget(IPC_PRIVATE, IPC_CREAT|0700)) != 0 );
  signal(SIGINT, cleanup);

  if (fork() == 0)
    ping();
  else
    pong();
}
```


mq_pong.c (2)

```c
void ping()
{
  numbers_msg msg;
  msg.first  = rand() % 100;
  msg.second = rand() % 100;

  while (1) {
    msg.mtype  = 1;

    printf("[A] Sending (%d,%d)\n", msg.first, msg.second);
    msgsnd(id, &msg, sizeof(msg)-sizeof(long), 0);

    msgrcv(id, &msg, sizeof(msg)-sizeof(long), 2, 0);
    printf("[A] Received (%d,%d)\n", msg.first, msg.second);

    ++msg.first;
    ++msg.second;
    sleep(3);
  }
}
```

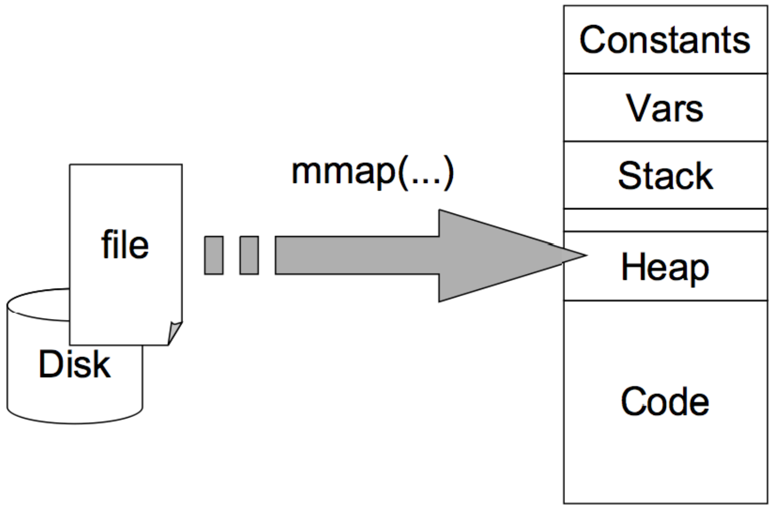
mq_pong.c (3)

```
void pong()
{
  numbers_msg msg;

  while (1) {
    msgrcv(id, &msg, sizeof(msg)-sizeof(long), 1, 0);
    printf("[B] Received (%d,%d)\n", msg.first, msg.second);

    msg.mtype  = 2;
    ++msg.first;
    ++msg.second;

    printf("[B] Sending (%d,%d)\n", msg.first, msg.second);
    msgsnd(id, &msg, sizeof(msg)-sizeof(long), 0);
  }
}
```


MEMORY MAPPED FILES

Memory Mapped Files

- Map a file into virtual memory
- No more read() or write()… just ordinary memory accesses



How MMF works?

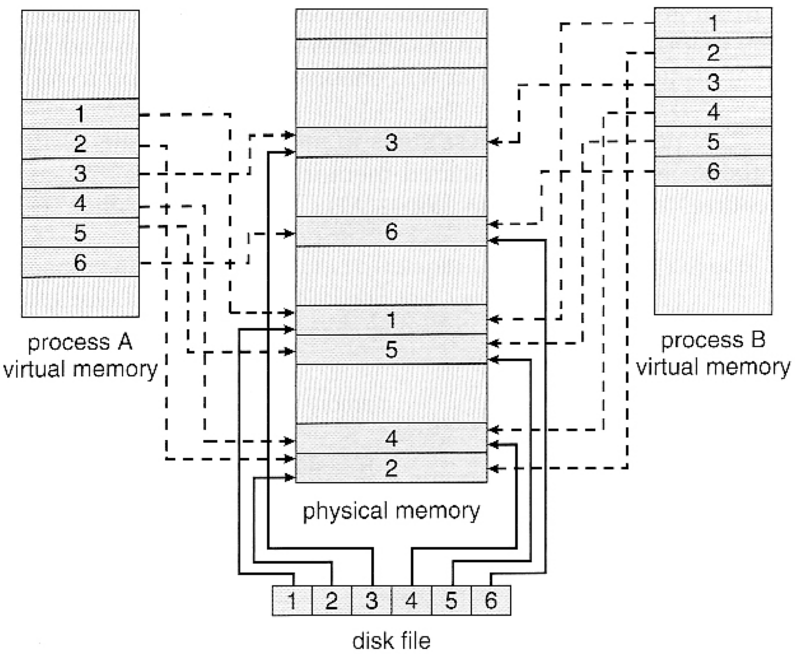


Figure 9.23 Memory-mapped files.

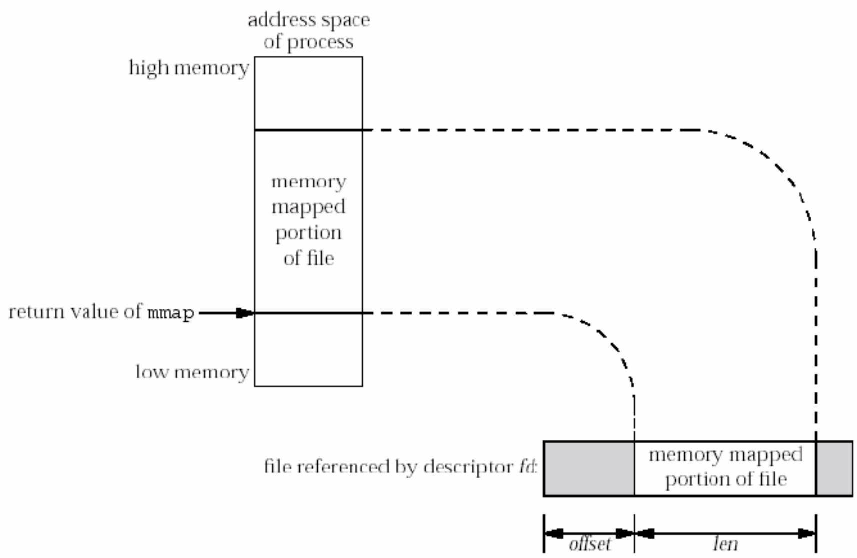
How can MMFs be used?

- **Map a file into the address space of a process**. The file is mapped into virtual memory.
- Simplifies file access by **treating file I/O through memory** rather than read() write() system calls.
- **Page faults** may read a page of file data from disk to memory.
- Allows **several processes to map the same file** allowing the pages in **memory to be shared**. Permits different processes to communicate very efficiently.
- Requires **synchronization** between processes that are storing/fetching information to/from the shared memory region.
- **Faster when copying one file to another.**

MMF
mmap()

- mmap()
  - creates a new mapping in the virtual address space of the calling process.

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



MMF

Using mmap()

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int
flags, int fd, off_t offset);
```

- Returns the address of the new mapping.
- **addr**
  - **addr** is **NULL ->** kernel chooses the address at which to create the mapping (most portable)
  - **addr** is **not NULL →** kernel takes it as a hint; on Linux, the mapping will be created at a nearby page boundary.
- **length**, **offset** and **fd**
  - Initialisation uses **length** bytes starting at **offset** in the file referred to by the file descriptor **fd**
  - **offset** must be a multiple of the page size as returned by sysconf(_SC_PAGE_SIZE)

MMF

Using mmap() (2)

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

- **prot**
  - The **prot** argument describes the desired memory protection of the mapping (and must not conflict with the open mode of the file). It is either PROT_NONE or the bitwise OR of one or more of the following flags:
    - PROT_EXEC Pages may be executed.
    - PROT_READ Pages may be read.
    - PROT_WRITE Pages may be written.
    - PROT_NONE Pages may not be accessed.

MMF

Using `mmap()` (3)

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

- **flags**

  The **flags** argument determines whether updates to the mapping are visible to other processes mapping the same region, and whether updates are carried through to the underlying file.

  - **MAP_SHARED** Updates to the mapping are visible to other processes that map this file, and are carried through to the underlying file. The file may not actually be updated until `msync` or `munmap` is called.
  - **MAP_PRIVATE** Create a private copy-on-write mapping. Updates to the mapping are not visible to other processes mapping the same file. It is unspecified whether changes made to the file after the `mmap()` call are visible in the mapped region.

MMF

Using `mmap()` (4)

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

- **flags** (cont.)
  - **MAP_ANONYMOUS** The mapping is not backed by any file; its contents are initialized to zero.
  - **MAP_FIXED** Don't interpret **addr** as a hint: place the mapping at exactly that address. **addr** must be a multiple of the page size.
  - **MAP_NONBLOCK** Only meaningful in conjunction with MAP_POPULATE. Don't perform read-ahead: only create page tables entries for pages that are already present in RAM.
  - **MAP_POPULATE** Populate page tables for a mapping. For a file mapping, this causes read-ahead on the file. Later accesses to the mapping will not be blocked by page faults.

MMF
Using `mmap()` (5)

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

- Use of a mapped region can result in these signals:
  - **SIGSEGV** - Attempted write into a region mapped as read-only.
  - **SIGBUS** - Attempted access to a portion of the buffer that does not correspond to the file (for example, beyond the end of the file, including the case where another process has truncated the file).

MMF
Beware!!!

- Memory mapped by `mmap()` is **preserved across `fork()`**, with the same attributes.
- A file is **mapped in multiples of the page size**. For a file that is not a multiple of the page size, the remaining memory is **zeroed** when mapped, and writes to that region are not written out to the file.
- The effect of **changing the size** of the underlying file of a mapping, on the pages that correspond to added or removed regions of the file, is **unspecified**.
- A file cannot be appended with `mmap`. The file size must be changed first.
- Closing the file descriptor of the mapped file does not unmap the file from memory.

MMF
mmap2()

```
#include <sys/mman.h>
void *mmap2(void *addr, size_t length,int prot, int
flags,int fd,  off_t pgoffset);
```

- The `mmap2()` system call operates in exactly the same way as `mmap()`, except that:
  - The final argument specifies the offset into the file in 4096-byte units (instead of bytes, as is done by `mmap()`).
    - This enables applications that use a 32-bit `off_t` to map large files (up to 2^{44} bytes).

MMF
munmap()

```
#include <sys/mman.h>
int munmap(void *addr, size_t length);
```

- The `munmap()` system call deletes the mappings for the specified address range
- The region is also automatically unmapped when the process is terminated.
- On the other hand, closing the file descriptor does not unmap the region.
- The address **addr** must be a multiple of the page size. All pages containing a part of the indicated range are unmapped, and subsequent references to these pages will generate `SIGSEGV`. It is not an error if the indicated range does not contain any mapped pages.

MMF
Auxiliary functions

```
int msync (void *address, size_t length, int flags)
```

- In shared mappings, it is the kernel that decides when to write to the underlying file.
- **msync()** flushes the changes made in memory to the underlying file. Specifically, it updates the part of the file that corresponds to the memory area starting at **addr** and having length **length**.
- Without this call, there is no guarantee that changes are written back before **munmap()** is called.

- **flags**
  - **MS_SYNC** - This flag makes sure the data is actually written to disk. Normally msync only makes sure that accesses to a file with conventional I/O reflect the recent changes.
  - **MS_ASYNC** - This tells msync to begin the synchronization, but not to wait for it to complete.
- Return
  - msync returns 0 for success and -1 for error.

MMF
Auxiliary functions (2)

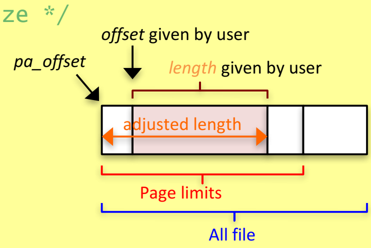
- Page-aligned mapping
  - Memory mapping only works on entire pages of memory.
  - Addresses for mapping must be page-aligned, and length values will be rounded up.
  - To determine the size of a page use:

```
#include sys/mman.h
size_t page_size = (size_t) sysconf (_SC_PAGESIZE);
```

MMF
Example

```
int main(int argc, char *argv[])
{
   char *addr;
   int fd;
   struct stat sb;
   off_t offset, pa_offset;
   size_t length;
   ssize_t s;

   fd = open(argv[1], O_RDONLY);        /* Open file */
   if (fstat(fd, &sb) == -1)             /* To obtain file size */
        perror("fstat");

   offset = atoi(argv[2]);
   /* offset for mmap() must be page aligned
      A binary AND is made between offset and
        the negation of the system page size */
   pa_offset = offset & ~(sysconf(_SC_PAGE_SIZE) - 1);

   length = atoi(argv[3]);

   addr = mmap(NULL, offset-pa_offset+length, PROT_READ, MAP_PRIVATE, fd, pa_offset);

   s = write(STDOUT_FILENO, addr + offset - pa_offset, length);

} /* main */
```

- Receives 3 command-line arguments:
  - File
  - Offset (from beginning of file)
  - Length
- Maps the file for reading to memory and prints to the *stdout* the specified part of the file



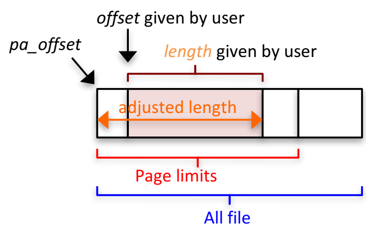
MMF
Example - page alignment offset explained

- Consider a page size of 4 KBytes (4096 bytes) and an offset of 9000 bytes wanted by the user

4096 = $0001\ 0000\ 0000\ 0000_2$ (e.g. system with 16 bits)
4096-1 = $0000\ 1111\ 1111\ 1111_2$

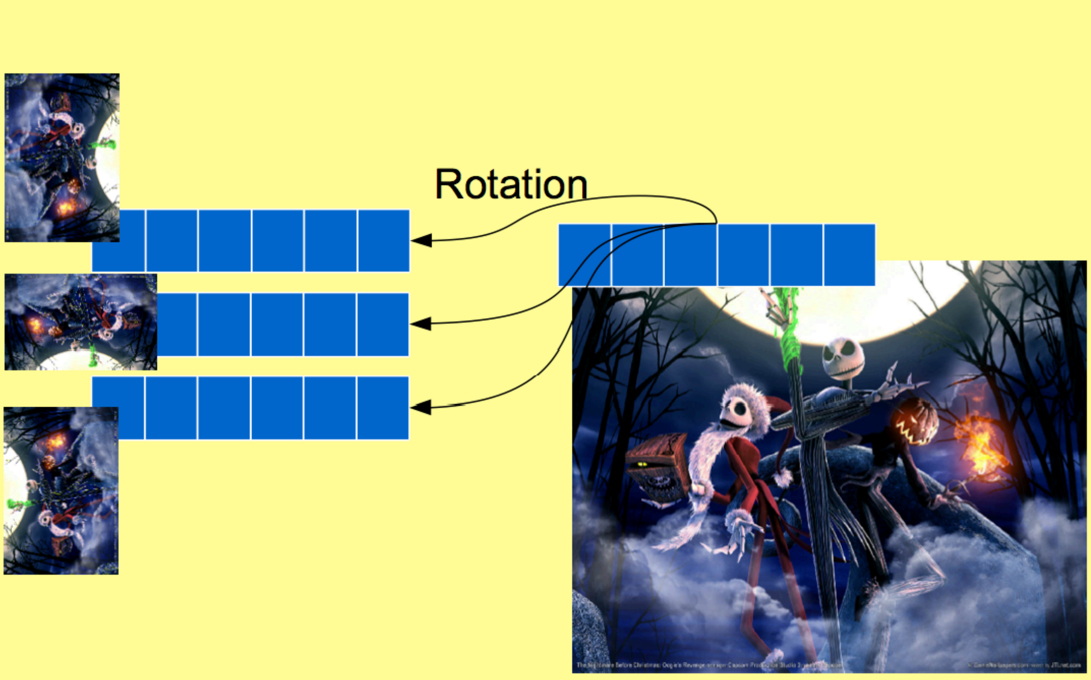
~(4096-1) = $1111\ 0000\ 0000\ 0000_2$
& (binary AND)
9000 = $0010\ 0011\ 0010\ 1000_2$
=
8192 (4096x2) = $0010\ 0000\ 0000\ 0000_2$

Page alignment offset (*pa_offset*)



Demo

Manipulating files as arrays.



INTRODUCTION TO ASSIGNMENT 08 – "MESSAGE QUEUES AND MEMORY MAPPED FILES"

Thank you! Questions?

I keep six honest serving men. They taught me all I knew. Their names are What and Why and When and How and Where and Who.
—Rudyard Kipling