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

#define MAXPAROLA 30
#define MAXRIGA 80

int main(int argc, char *argv[])
{
  int freq[MAXPAROLA] ; /* vettore di contatori
  delle frequenze delle lunghezze delle parole */
  char riga[MAXRIGA] ;
  int i, inizio, lunghezza ;
  FILE * f ;

  for(i=0; i<MAXPAROLA; i++)
    freq[i]=0 ;

  if(argc != 2)
  {
    fprintf(stderr, "ERRORE, serve un parametro con il nome del file\n");
    exit(1);
  }
  f = fopen(argv[1], "rt") ;
  if(f==NULL)
  {
    fprintf(stderr, "ERRORE, impossibile aprire il file %s\n", argv[1]);
    exit(1);
  }

  while( fgets( riga, MAXRIGA, f ) != NULL )
```

System and Device Programming

# Memory Mapping

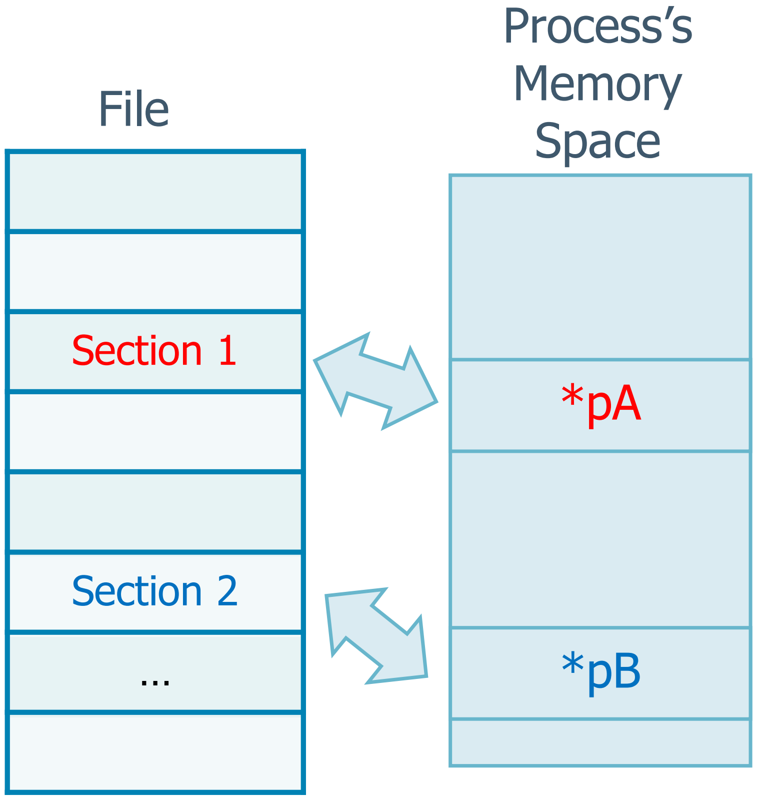
Stefano Quer

Dipartimento di Automatica e Informatica

Politecnico di Torino

# Memory Management

- OSs provide memory-mapped files to
  - Associate a process's address space with a file
  - Allow the OS to manage all data movement between the file and memory while the user just cope with memory address space
  - Permit the programmer to manipulate files without file I/O functions
    - No need to use open, read, write, lseek, etc.



# Memory Management

- The advantages to mapping the virtual memory space directly to normal files include
  - Applications can
    - Be significantly **faster**
      - **In-memory** algorithms (string processing, sorts, search trees) can directly process files
    - Manipulate larger quantity of data
      - Files may be much larger than the available physical memory
  - There is no need to manage buffers and the file data they contain
  - Multiple processes can **share** memory, and the file views will be coherent

# Function mmap

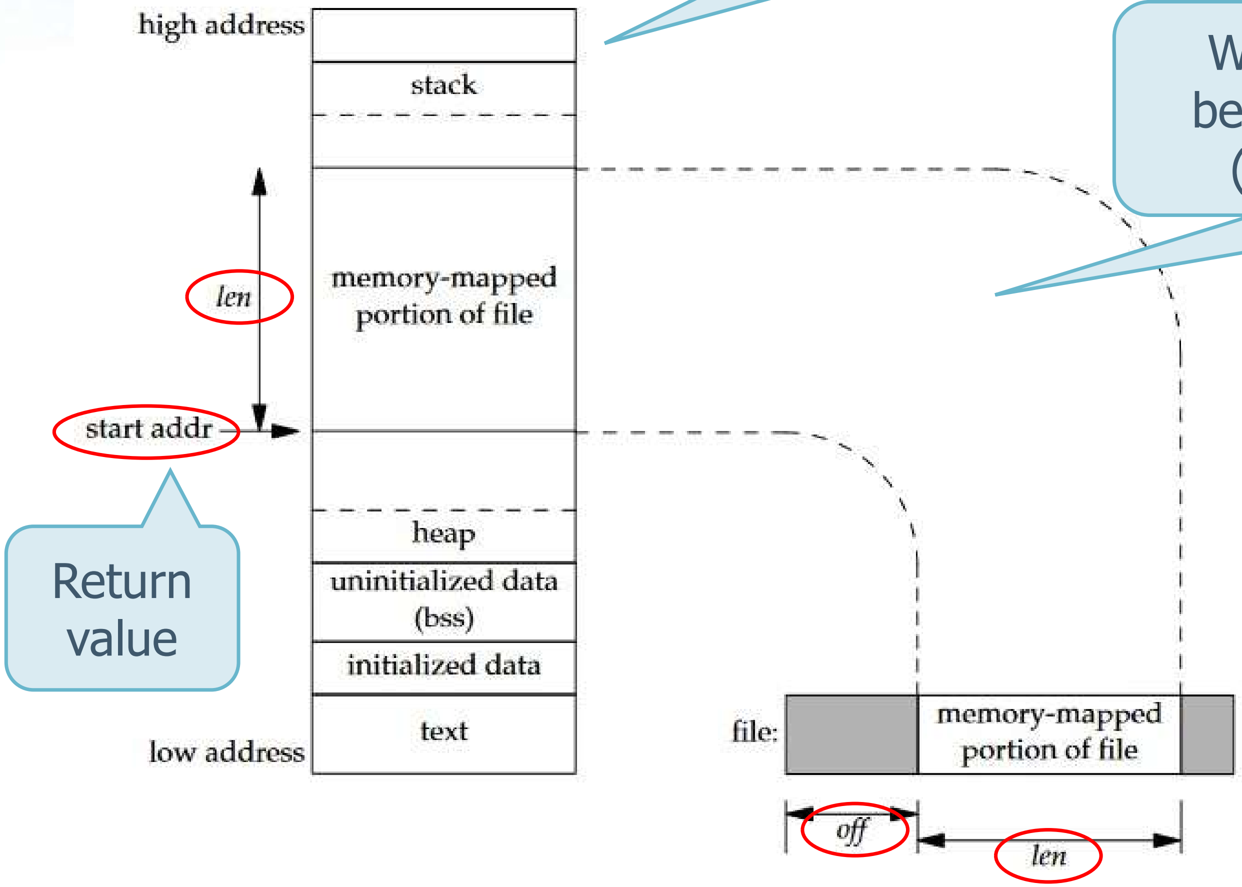
```
include <sys/mman.h>

void *mmap (
  void *addr,size_t len,int prot,int flag,int fd,off_t off
);
```

- We need to inform the the kernel to map a given file to a region in memory using mmap
- Return value
  - The starting address of the mapped region, in case of success
  - The constant MAP_FAILED, in case of errors

# Function mmap

Layout of a typical process

We suppose the file is mapped between the heap and the stack (implementation dependent)



Return value

```
void *mmap (
  void *addr, size_t len, int prot,
  int flag, int fd, off_t off
);
```

# Function mmap

- Parameters
  - **addr** specifies the address where we want the mapped region to start
    - We normally set this value to 0 to allow the system to choose the starting address
  - **len** is the number of bytes to map

```
void *mmap (
  void *addr, size_t len, int prot,
  int flag, int fd, off_t off
);
```

# Function mmap

- **prot** specifies the protection of the mapped region
  - PROT_READ to read
  - PROT_WRITE to write
  - PROT_EXEC to execute
  - A bitwise OR of the previous flags
  - PROT_NONE indicates that the region cannot be accessed

```
void *mmap (
  void *addr, size_t len, int prot,
  int flag, int fd, off_t off
);
```

# Function mmap

- **flag** affects various attributes of the region
  - MAP_FIXED
    - Forces the return value to be equal to addr
    - Its use is discouraged, as it hinders portability
  - MAP_SHARED
    - This flag specifies that a store operation is equivalent to a write to the file
    - Either this flag or the next (not both) must be specified
  - MAP_PRIVATE
    - Store operations into the mapped region cause a private copy of the mapped file to be created

```
void *mmap (
  void *addr, size_t len, int prot,
  int flag, int fd, off_t off
);
```

# Function mmap

- **fd** argument is the file descriptor specifying the file that is to be mapped
  - We have to open this file before we can map it into the address space
- **off** is the starting offset in the file of the bytes to map

```
void *mmap (
  void *addr, size_t len, int prot,
  int flag, int fd, off_t off
);
```

# Function munmap

```
include <sys/mman.h>

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

- A memory-mapped region is unmapped when
  - The process terminates
  - We unmap a region by calling the **munmap** function
  - Note that closing the file descriptor does not unmap the region
- Return value
  - The value 0, in case of success
  - The value -1, in case of errors

# Function munmap

- A call to munmap does not cause the contents of the mapped region to be written to the disk file
  - The kernel updates the disk file (for a MAP_SHARED region) automatically sometime after we write into the memory-mapped region
  - Modifications to memory in a MAP_PRIVATE region are discarded when the region is unmapped

# Exercise

- Using memory mapping copy a source file into a destination (equaivalente) file
  - In other words, implement the shell command “cp“ adopting memopry mapped files
- The program receives the source and the destination file names on the command line

# Example

```
#include "apue.h"
#include <fcntl.h>
#include <sys/mman.h>

#define COPYINCR (1024*1024*1024)

int main(int argc, char *argv[]) {
  int fdin, fdout;
  void *src, *dst;
  size_t copysz;
  struct stat sbuf;
  off_t fsz = 0;

  if (argc != 3) {
   ... error ...
  }
```

We limit the amount of memory we copy to 1GByte to avoid exceeding the memory used

# Example

Open the source file in reading mode

Open the destination file in R/W mode

```
fdin = open (argv[1], O_RDONLY));
if (fdin<0)
  ... error ...

fdout = open(argv[2], O_RDWR|O_CREAT|O_TRUNC, FILE_MODE);
if (fdout < 0)
  ... error ...

if (fstat(fdin, &sbuf) < 0)
  ... error ...

if (ftruncate(fdout, sbuf.st_size) < 0)
  ... error ...
```

Get source file size (we need it to map)

Set destination file size equal to source file size

If we do not do that, we can map anyway, but wrw can have problems

# Example

We iterates until the entire file is copied

```
while (fsz < sbuf.st_size) {
  if ((sbuf.st_size - fsz) > COPYINCR)
    copysz = COPYINCR;
  else
    copysz = sbuf.st_size - fsz;
  if ((src = mmap(0,copysz,PROT_READ,MAP_SHARED,fdin,fsz))
       == MAP_FAILED)
    ... error ...
  if ((dst = mmap(0, copysz, PROT_READ | PROT_WRITE,
       MAP_SHARED, fdout, fsz)) == MAP_FAILED)
    ... error ...
  memcpy(dst, src, copysz);
  munmap(src, copysz);
  munmap(dst, copysz);
  fsz += copysz;
  }
  return (0);
}
```

We move on 1GByte at a time at most

Map source file

Map destination file

Copy

Unmap

# Guidelines

- With mapping we have to be careful with the file size
  - We cannot map too much
  - The size does not have to change
  - The mapped memory must be manipulated through pointers
    - Not need to use read and write system calls

```
...
memcpy(dst, src, copysz);
...
```

```
for (i=0; i<copysz; i++) {
  *dst = *src;
  dst++;
  src++;
}
```

Memory copy

# Guidelines

- On modern UNIX-like operating system memory mapping can be faster or slower than direct read/write
  - With **read/write**, we activate more system calls
    - We copy the data from the kernel's buffer to the application's buffer (read), and then from the application's buffer to the kernel's buffer (write)
  - With **mmap/memcpy**, we copy the data directly from one kernel buffer mapped into our address space into another kernel buffer mapped into our address space