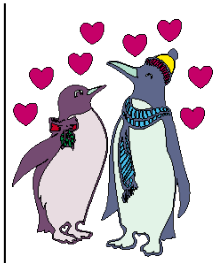
Systems Calls - I/O (II)

Alain NINANE – RSSI UCL

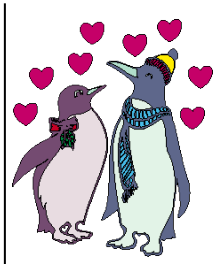
1er mars 2016

Recaps (I)



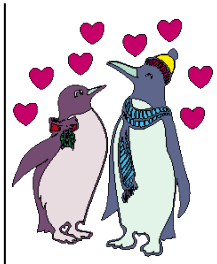
- Systems calls are kernel entry points
  - Systems calls are in finite number in unix
  - Portability issues
- I/O primitives & functions
  - lower-level (raw) - system calls
    - open, creat, read, write, ioctl, close, lseek
  - higher-level - the Standard C I/O library
    - fopen, fread, fgets, getchar, etc ... (could be ∞)

Recaps (II)



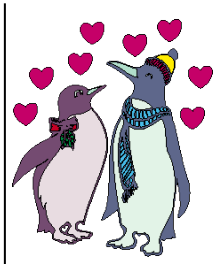
- A process holds, in the kernel, a table with opened file descriptors
  - table
    - table size
      - maximum table size
- Ressources (like fd's) are therefore limited
  - at the user level (e.g. the shell)
  - at the kernel level

Ressources limits (I)



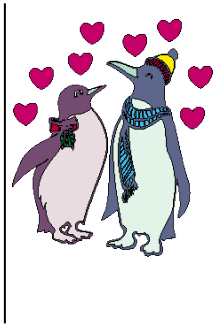
- Ressources can be limited at the shell level
  - soft limit: ulimit -aS
  - hard limit: ulimit -aH
- E.g. open files (ulimit -n)
  - soft: 256 (ulimit -n -S)
  - hard (ulimit -n -H):
    - unlimited (macosx)
    - 1024 (RHEL linux)

Ressources limits (II)



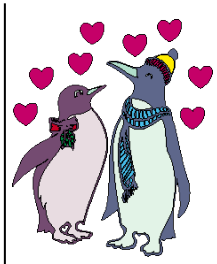
- Demonstration
  - test_mxnf
  - Error after ~ 252 created/opened files
- Change shell soft open files limit
  - ulimit -n unlimited
  - ulimit -n -S
- Run
  - test_mxnf
  - Error after ~ 7800 created/opened files
  - System unusable
    - **Commands not found :-)**
    - **Deny of service !!!!!!**
    - Remember: linux ≠ macosx

User - Kernel ops “mapping”



- For each command/operation at the user level, there is a low-level kernel equivalent
- Examples:
  - file redirection (dup)
  - files operations
    - creation of directories (mkdir)
    - creation of links (link)
    - removal of files (rm, unlink)

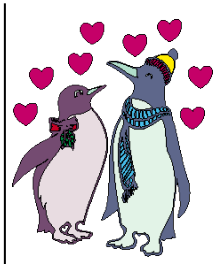
Ex. Implementing “< file”



- /* Close standard Input */
  - int ret = close(0)
- /* Open input file */
  - int fd = open(“file”,O_RDONLY)

- Note: open() returns the lowest fd available ...
  - in this case ... 0
  - subsequent read(0, ...) will read from “file”

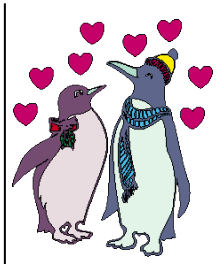
Ex. Implementing “< in > out”



- /* close standard input */
  - int ret = close(0)
- /* ”in” takes fd 0 */
  - int fd = open(“in”,O_RDONLY);

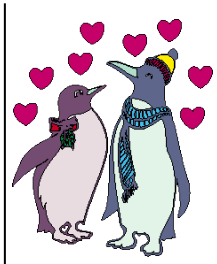
- /* close standard output */
  - int ret = close(1)
- /* “out” takes fd 1 */
  - int fd = open(“out”,O_WRONLY);

“< file” implementation problem



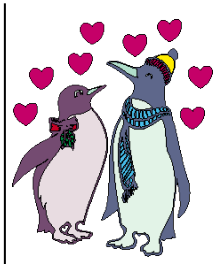
- close(0)
  - The original “device” pointed by fd = 0 is closed
- open(“in”,O_RDONLY)
  - fd = 0 is replaced by “in”
- Lost device
  - Original lost “device” is the keyboard
- Solution
  - int dup(int fd)
    - Duplication of file descriptor

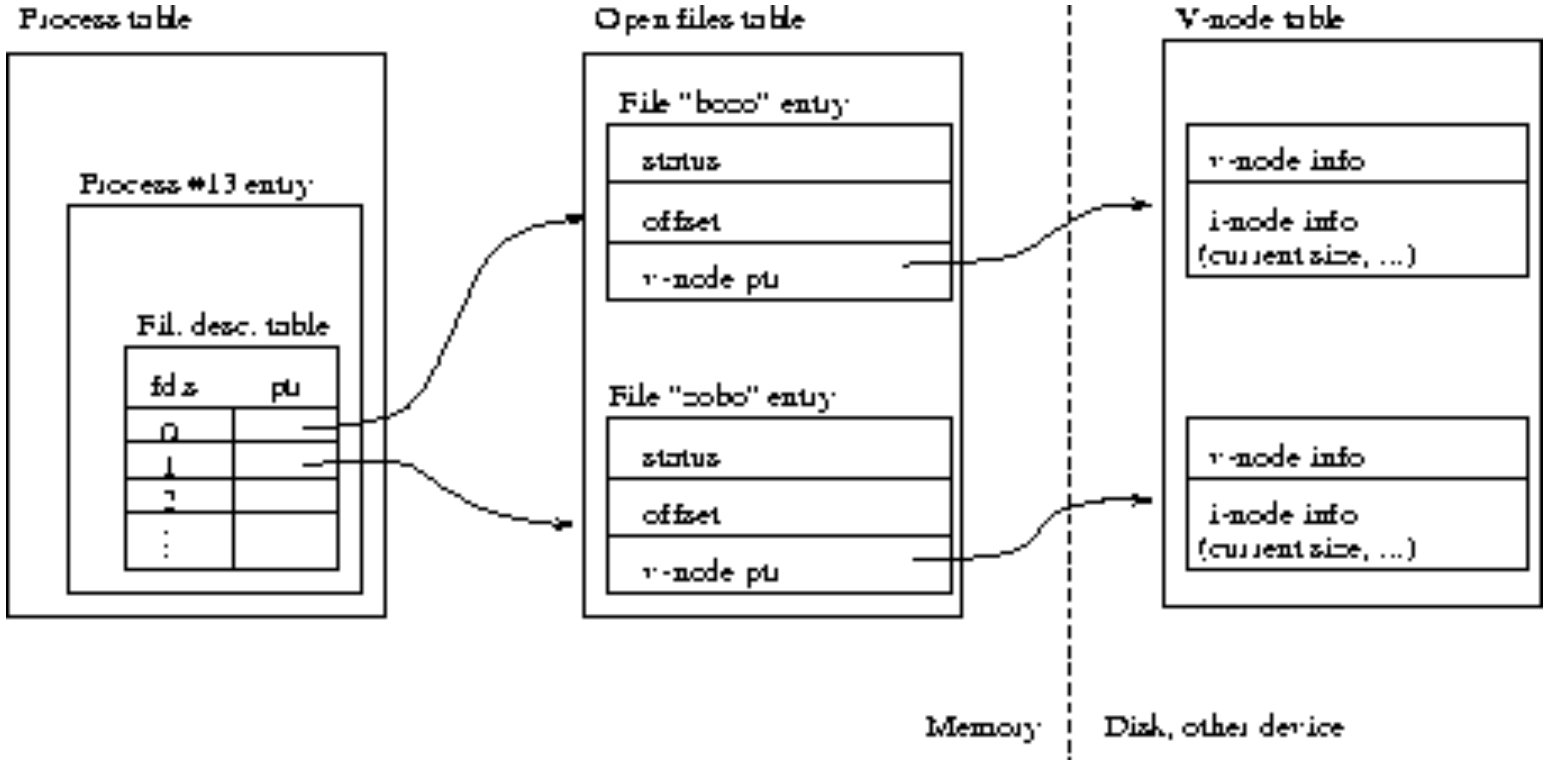
dup() system call



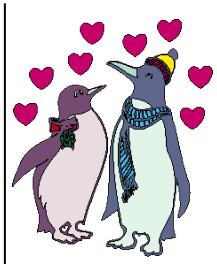
- int fd2 = dup(int fd1)
  - fd2 is a duplication of fd1
  - fd2 shares the same file entry than fd1
    - Share same file status flags & access mode
      - read, write, append, ...
    - Share same current file offset
  - fd2 is the lowest numbered descriptor available
  - fd1 can be closed (before being re-assigned)
  - dup() can fails !!!
    - errno: EBADF, EMFILE

dup() system call picture



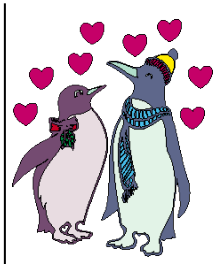


Ex. Implementing “< file” (II)



- /* Duplicate keyboard */
  - keyb = dup(0);
- /* Close standard Input */
  - close(0)
- /* Open input file ----> fd = 0 */
  - fd = open(“file”,O_RDONLY)
  - read(0, ...) reads from file instead of keyboard
- /* Still read keyboard */
  - read(keyb,buffer,80);

dup2() system call

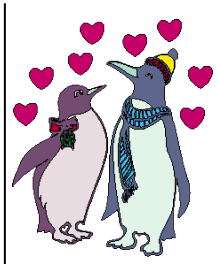


- int fd = dup2(int fd1, fd2);
  - fd1 = old file descriptor
  - fd2 = candidate file descriptor
    - if fd2 opened close it
  - fd2 is a dup() of fd1
    - fd2 is set to point to the same file as fd1
  - dup2(fd1,fd2)
    - close(fd2)
    - fd2 = dup(fd1)

Asynchronous Operation Possible (signal ())

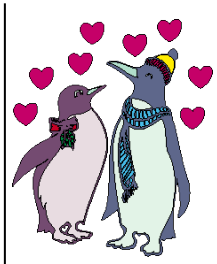
- dup2() is an atomic operation

I/O Mechanisms



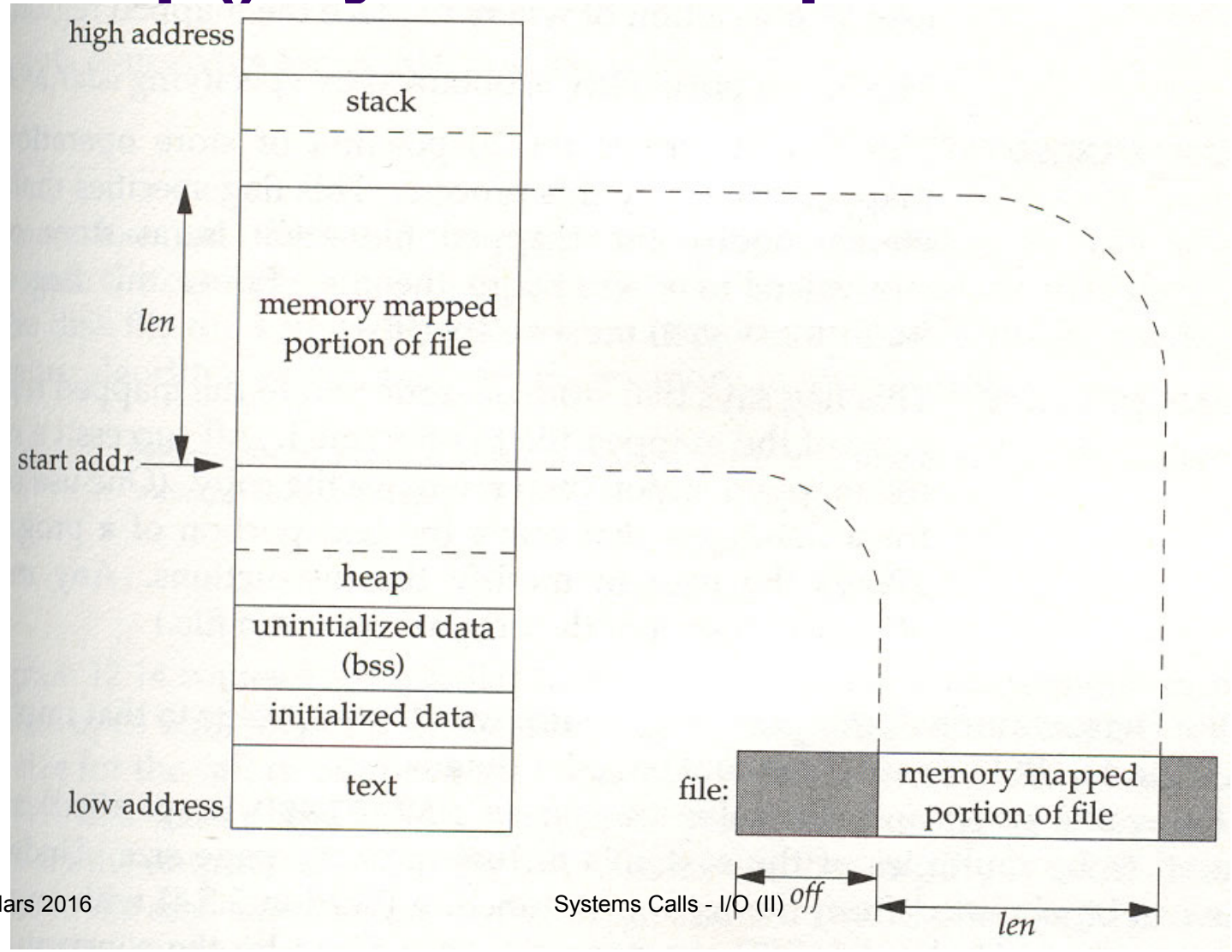
- Basic UNIX I/O
  - data are read/written from files by a process
  - i/o seen as an unstructured sequence of bytes
  - files, pipes, terminals, tapes, network sockets, ...
- Memory Mapped I/O
  - externals objects are mapped in the process virtual memory
  - files, some peripherals (e.g. memory of network controller)
- Message passing I/O

Basic UNIX I/O Problem

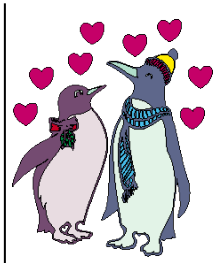


- Add 10 to a line in a file
  - read(fd,&val,sizeof(val));
  - val += 10;
  - lseek(fd,-sizeof(val),SEEK_CUR);
  - write(fd,&val,sizeof(val));
- Needs 3 explicit system calls
- Copy of data between kernel and user I/O
- Memory Map (mmap) solution
  - access data file as in-memory data
  - *ptr += 10;

mmap() system call picture

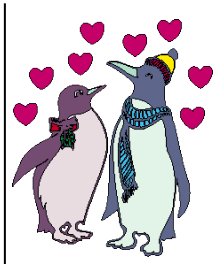


mmap() system call (I)



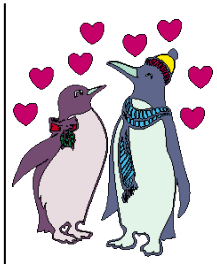
- #include <sys/types.h>
- #include <sys/mman.h>
- caddr_t mmap(addr,len,prot,flags,fd,off)
  - caddr_t addr; /* Should be 0 */
  - size_t len; /* Number of bytes to map */
  - int prot; /* Type of access */
  - int fd; /* File descriptor to map */
  - int off; /* Origin of map in fd */

mmap() system call (II)



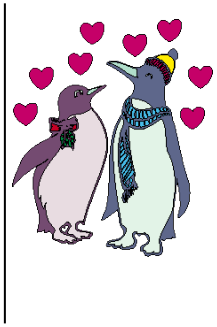
- int prot;
  - PROT_READ
  - PROT_WRITE
  - PROT_EXEC
  - PROT_NONE
- int flags;
  - MAP_SHARED: store will update the file
  - MAP_PRIVATE: store will create a private copy

mmap() system call (III)



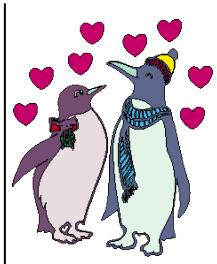
- The file must be opened (fd)
- mmap() returns the base pointer to access the data
- Returns -1 in case of failure (check errno)
- several processes can map the same file
  - A way to share data between processes
- mmap is used by UNIX to share libraries
  - Demonstration
    - % /usr/sbin/lsof -p PID
    - % strace COMMAND (linux)
      - *VERY USEFUL COMMAND FOR DEBUGGING*
      - *PRINT ALL SYSCALLS*

munmap() system call



- #include <sys/types.h>
- #include <sys/mman.h>
- int munmap(caddr_t addr, size_t len);
  - addr: origin of VM range to unmap
  - len: length of VM range
- Returns: 0 if sucess; -1 if failure
- Destroy the mapping between the file and the virtual memory address space
- Close do not call munmap.

msync() system call



- #include <sys/types.h>
- #include <sys/mman.h>
- int msync(caddr_t addr, size_t len, int flags);
  - addr: origin of VM range to unmap
  - len: length of VM range
  - flags:
    - MS_ASYNC: returns immediately
    - MS_INVALIDATE: invalidate the caching (broken on linux ?)
- Returns: 0 if sucess; -1 if failure
- Force the system to write mapped data to disk