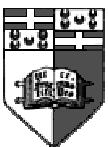




File Ownership and Permissions

- Every file has two distinct properties: the user and group ownership ID, and the file access permissions (`ls -l`).
- Ownership is specific to a certain user and group.
- Access permission may be to the user, group or/and everyone else.
- To access a file we need execute permissions inside the directory hosting the file.
- Directory read permissions merely lets us view the contents.

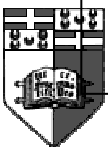




File Access Permissions

- We cannot create a file unless we have write permissions to the directory.
- The file ownership user ID and a group ID normally are the same as those of the creator.
- Access to a file is allowed when the effective user ID is equal to the file owner ID (or group IDs) or when permissions allow it.
- **root** is allowed access to all files

```
#include <unistd.h>
int access(const char *pathname,int mode);
    mode: R_OK, W_OK, X_OK,F_OK
    returns 0 if OK, -1 on error
```



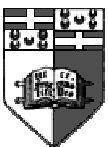


File Access Permissions (cont)

- *chmod* allows one to change the file access permissions of a file

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char *pathname, mode_t mode);
int fchmod(int fd, mode_t mode);
           returns 0 if OK, -1 on error
```

- *mode* is the same as *mode* in *open*.
- `S_ISVTX` sets sticky bit if **root**.
- One can change the ownership of a file, but then, unless one is **root**, privileged access rights are lost.
- This is performed using *chown()*, *fchown()* and *lchown()*.





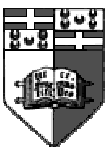
stat()

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *pathname, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *pathname, struct stat *buf);
```

```
struct stat {
    mode_t st_mode;
    ino_t st_ino;
    dev_t st_dev;
    dev_t st_rdev;
    nlink_t st_nlink;
    uid_t st_uid;
    gid_t st_gid;
    off_t st_size;
    time_t st_atime;
    time_t st_mtime;
    time_t st_ctime;
    long st_blksize;
    long st_blocks;
```

```
}
```

```
S_ISREG(mode_t st_mode);
S_ISDIR(mode_t st_mode);
S_ISLNK(mode_t st_mode);
    return 1 if YES, 0 if NO
```

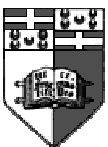




Soft and hard links

- A soft link is a file containing the path of another file. It has an inode and a data block.
- Most function calls, unless specified, follow symbolic links.
- Yet one can have two or more directory entries pointing to the same i-node number. This is a hard link.
- Only a superuser can create a hard link to another directory since it can cause loops.

```
#include <unistd.h>
int link(const char *existingpath,
         const char *newpath);
int unlink(const char *pathname);
int symlink(const char *actualpath,
            const char *sympath);
int readlink(const char *pathname,
             char *buf, int bufsize);
Return -1 on error
```



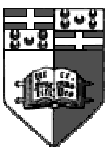


Directories

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

int mkdir(const char *pathname, mode_t mode);
int rmdir(const char *pathname);
DIR *opendir(const char *pathname);
struct dirent *readdir(DIR *dp);
void rewinddir(DIR *dp);
int close(DIR *dp);
int chdir(const char *pathname);
char *getcwd(char *buf, size_t size);
        return -1 or NULL on error
```

```
struct dirent {
    ino_t d_ino;
    char dname[NAME_MAX+1];
}
```





Other basic I/O calls

```
#include <unistd.h>
```

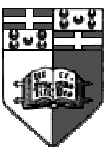
```
void sync(void);
```

```
void fsync(int fd);
```

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

```
int remove(const char *pathname);
```

```
int rename(const char *oldname,  
           const char *newname);
```

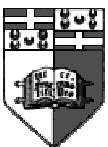




Record Locking

- When more than one process writes to the same file we have a race condition.
- With record locking we ensure deterministically which version of a file is the final version.
- These locks are advisory !!
- Locks are released when process terminates or when a locked file descriptor is closed (warning!!).
- Lock structures are held in the v-node entry.
- Mandatory locks are possible yet not portable (SUID on, SGID off).

	RDLCK	WRLCK
no lock	ok	ok
read lock	ok	no
write lock	no	no





Record Locking (cont)

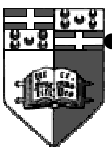
```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
```

```
int fcntl(int fd, int cmd, struct flock *flockptr);
           returns -1 on error
```

```
struct flock {
    short l_type;
    off_t l_start;
    short l_whence;
    off_t l_len;
    pid_t l_pid;
}
```

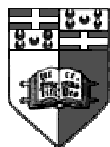
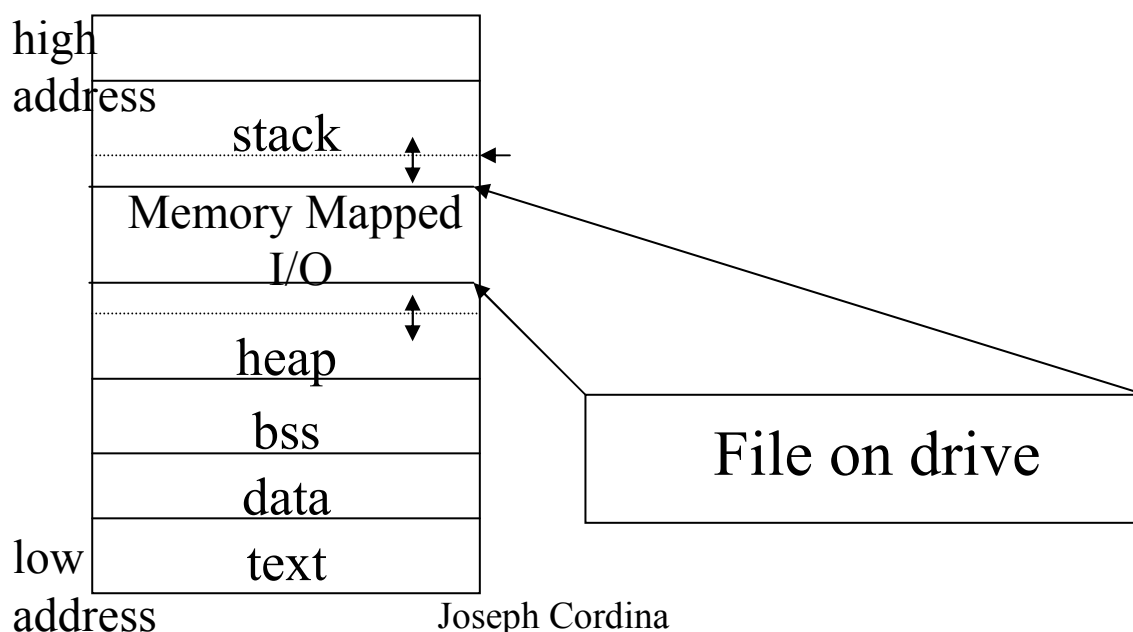
- *cmd*:
 - F_GETLCK: if ok *l_type* is set to F_UNLCK, otherwise *struct flock* is replaced with the relevant lock info.

- F_SETLCK: sets the lock by setting type to F_RDLCK or F_WRLCK.
- F_SETLCKW: Blocking version of F_SETLCK (DEADLOCK WARNING).
- *l_len* of 0 locks to end of file.



Unix Memory Mapped I/O

- UNIX allows us to map a region of a file into memory.
- One then accesses the file like any normal memory access; through pointers, etc.
- When a file is mapped, all the file contents is reflected inside the memory area.
- We can have a copy of the file for private use or have the kernel copy what we write to this mapped area onto the file.



low
address

Joseph Cordina

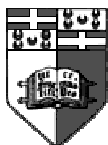




Memory Mapped I/O (cont)

```
#include <sys/types.h>
#include <mman.h>
caddr_t mmap(caddr_t addr, size_t len,
             int prot, int flag, int fd,
             off_t off);
                returns -1 on error
```

- *caddr_t* can be used as *char **.
- *mmap* returns the starting address of the mapped region.
- *prot* fields (must match opened file):
 - PROT_READ
 - PROT_WRITE
 - PROT_EXEC
 - PROT_NONE (!!)
- *addr* is set to 0 to allow the system choose the ideal place where to map the file area.
- *flag* fields
 - MAP_SHARED
 - MAP_PRIVATE

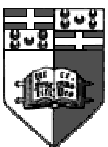




Memory Mapped I/O (cont)

- Memory mapping is ideal when accessing external devices which have on-board SRAM.
- Makes coding much neater.
- *munmap* does not flush the contents to disk. *munmap* is automatic on process termination but not on closing the file descriptor.

```
#include <sys/types.h>
#include <sys/mmap.h>
int munmap(caddr_t addr, size_t len);
```





Exercises

- Write your own `mkdir` and `rmdir`.
- Create a file and create two hard links to it and then two soft links to it (remember hard links are directories while soft links are merely files).
- Access one of your directories and display all the information of all the files and directories inside it using `stat()`.
- Open a file and memory map it. Then create a linked list inside it. Using another process try using the list again. You have now an easy way to store any data structure !! (Revise your typecasts)

