File and Record Locking

or, How to coordinate access to a file

- For regular files only

File and Record Locking

or, How to coordinate access to a file

- For regular files only
- applies to entire files or just to regions of files

File and Record Locking

or, How to coordinate access to a file

- For regular files only
- applies to entire files or just to regions of files
- Two methods of locking:

System V:	supports **mandatory** locking (ie. kernel enforced)
BSD/Posix:	**advisory** locking only (ie. cooperative)
Linux:	supports both advisory locking and mandatory locking

File and Record Locking

or, How to coordinate access to a file

- For regular files only
- applies to entire files or just to regions of files
- Two methods of locking:

System V:	supports **mandatory** locking (ie. kernel enforced)
BSD/Posix:	**advisory** locking only (ie. cooperative)
Linux:	supports both advisory locking and mandatory locking

- **read locks**: prevents overlapping write locks
  allows read locks by other processes

File and Record Locking

or, How to coordinate access to a file

- For regular files only
- applies to entire files or just to regions of files
- Two methods of locking:

System V:	supports **mandatory** locking (ie. kernel enforced)
BSD/Posix:	**advisory** locking only (ie. cooperative)
Linux:	supports both advisory locking and mandatory locking

- **read locks**: prevents overlapping write locks
  allows read locks by other processes
- **write locks**: prevents overlapping read and/or write locks (exclusive locking)

Locking, con’t.

- *Advisory locks are not enforced, but they may be checked for by cooperating processes*

Locking, con’t.

- *Advisory locks are not enforced, but they may be checked for by cooperating processes*
- Locks can be made down to single bytes!

Locking, con’t.

- *Advisory locks are not enforced, but they may be checked for by cooperating processes*
- Locks can be made down to single bytes!

Region currently has	*read lock*	*write lock*
no lock	ok	ok
1+ read locks	ok	denied
1 write lock	denied	denied

Locking, con't.

- *Advisory locks are not enforced, but they may be checked for by cooperating processes*
- Locks can be made down to single bytes!

Region currently has	*read lock*	*write lock*
no lock	ok	ok
1+ read locks	ok	denied
1 write lock	denied	denied

Advisory Locking Procedure

1. Try to set lock
2. If lock is acquired, read/write as desired
3. Release the lock

fcntl()

#include <unistd.h>

#include <fcntl.h>

int fcntl(int fd, int cmd,...)

	F_SETLK	sets locks, but does not block
cmd		

fcntl()

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd,...)
```

	F_SETLK	sets locks, but does not block
cmd	F_SETLKW	sets lock and blocks until lock is acquired

fcntl()

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd,...)
```

	F_SETLK	sets locks, but does not block
cmd	F_SETLKW	sets lock and blocks until lock is acquired
	F_GETLK	lock query – what process holds the locked file

fcntl()

#include <unistd.h>

#include <fcntl.h>

int fcntl(int fd, int cmd,...)

	F_SETLK	sets locks, but does not block
cmd	F_SETLKW	sets lock and blocks until lock is acquired
	F_GETLK	lock query – what process holds the locked file

3rd arg is a struct flock *pflk:

```
struct flock {
        short l_type;        type of lock
        short l_whence;      reference address for l_start
        off_t l_start;       offset from l_whence
        off_t l_len;         qty bytes in locked region
        pid_t l_pid;         pid of owning lock process
        }
```

fcntl()

#include <unistd.h>

#include <fcntl.h>

int fcntl(int fd, int cmd,...)

	F_SETLK	sets locks, but does not block
cmd	F_SETLKW	sets lock and blocks until lock is acquired
	F_GETLK	lock query – what process holds the locked file

3rd arg is a struct flock *pflk:

```
struct flock {
    short l_type;       type of lock
    short l_whence;     reference address for l_start
    off_t l_start;      offset from l_whence
    off_t l_len;        qty bytes in locked region
    pid_t l_pid;        pid of owning lock process
    }
```

Some systems also have the l_sysid field (system id)

fcntl(): struct flock

l_type	F_RDLCK	sets a read lock on a region
	F_WRLCK	sets a write lock on a region
	F_UNLCK	unlocks region

fcntl(): struct flock

l_type	F_RDLCK	sets a read lock on a region
	F_WRLCK	sets a write lock on a region
	F_UNLCK	unlocks region
l_whence, l_start	SEEK_CUR	current file ptr + l_start
	SEEK_SET	l_start
	SEEK_END	end-of-file ptr + l_start

fcntl(): struct flock

l_type	F_RDLCK	sets a read lock on a region
	F_WRLCK	sets a write lock on a region
	F_UNLCK	unlocks region
l_whence, l_start	SEEK_CUR	current file ptr + l_start
	SEEK_SET	l_start
	SEEK_END	end-of-file ptr + l_start
l_len	=0	start to ∞
	+	length of locked region in bytes
	-	formerly illegal
		allowed for Linux kernels 2.4+
		allowed for POSIX.1-2001
		locks l_start+l_len to l_start-1

Promotion and Splitting

lock promotion handling overlapping locks

Promotion and Splitting

lock promotion handling overlapping locks

- read locks are superceded by write locks

Promotion and Splitting

lock promotion handling overlapping locks

- read locks are superceded by write locks
- overlapping locks grow into one lock

  Example: lock[0,100] then lock[50,200] $\longrightarrow$ [0,200] *(one lock!)*

Promotion and Splitting

lock promotion handling overlapping locks

- read locks are superceded by write locks
- overlapping locks grow into one lock

  Example: lock[0,100] then lock[50,200] $\longrightarrow$ [0,200] *(one lock!)*

lock splitting unlocking a middle portion of a locked region results in the process holding two locks

Example: lock[0,150]

unlocking [50,100] $\longrightarrow$ lock[0,49] and lock[101,150] *(two locks!)*

Promotion and Splitting

lock promotion handling overlapping locks

- read locks are superceded by write locks
- overlapping locks grow into one lock

  Example: lock[0,100] then lock[50,200] $\longrightarrow$ [0,200] *(one lock!)*

lock splitting unlocking a middle portion of a locked region results in the process holding two locks

Example: lock[0,150]

unlocking [50,100] $\longrightarrow$ lock[0,49] and lock[101,150] *(two locks!)*

(Locks are associated with inodes)

lockf()

```
#include <unistd.h>

int lockf(int fd,int cmd,off_t len)
```

Purpose: to apply, test, or remove a POSIX lock on an open file

Very similar to flock(). Commands supported include:

F_LOCK set an exclusive lock

lockf()

```
#include <unistd.h>
int lockf(int fd,int cmd,off_t len)
```

Purpose: to apply, test, or remove a POSIX lock on an open file

Very similar to flock(). Commands supported include:

F_LOCK set an exclusive lock

F_TLOCK same as F_LOCK but the call never blocks; returns an error if the file is already locked

lockf()

```
#include <unistd.h>
int lockf(int fd,int cmd,off_t len)
```

Purpose: to apply, test, or remove a POSIX lock on an open file

Very similar to flock(). Commands supported include:

F_LOCK set an exclusive lock

F_TLOCK same as F_LOCK but the call never blocks; returns an error if the file is already locked

F_ULOCK unlock indicated section of file

lockf()

```
#include <unistd.h>

int lockf(int fd,int cmd,off_t len)
```

Purpose: to apply, test, or remove a POSIX lock on an open file

Very similar to flock(). Commands supported include:

F_LOCK set an exclusive lock

F_TLOCK same as F_LOCK but the call never blocks; returns an error if the file is already locked

F_ULOCK unlock indicated section of file

F_TEST test if locked

lockf()

```
#include <unistd.h>
int lockf(int fd,int cmd,off_t len)
```

Purpose: to apply, test, or remove a POSIX lock on an open file

Very similar to flock(). Commands supported include:

F_LOCK set an exclusive lock

F_TLOCK same as F_LOCK but the call never blocks; returns an error if the file is already locked

F_ULOCK unlock indicated section of file

F_TEST test if locked

- 0 = unlocked or this program holds the lock
- -1 = and errno=EAGAIN : lock held by another program

lockf()

```
#include <unistd.h>
int lockf(int fd,int cmd,off_t len)
```

Purpose: to apply, test, or remove a POSIX lock on an open file

Very similar to flock(). Commands supported include:

F_LOCK set an exclusive lock

F_TLOCK same as F_LOCK but the call never blocks; returns an error if the file is already locked

F_ULOCK unlock indicated section of file

F_TEST test if locked

0 = unlocked or this program holds the lock

-1 = and errno=EAGAIN : lock held by another program

Upon process termination, all file locks are released.

lockf()

```
#include <unistd.h>
int lockf(int fd,int cmd,off_t len)
```

Purpose: to apply, test, or remove a POSIX lock on an open file

Very similar to flock(). Commands supported include:

F_LOCK set an exclusive lock

F_TLOCK same as F_LOCK but the call never blocks; returns an error if the file is already locked

F_ULOCK unlock indicated section of file

F_TEST test if locked

> 0 = unlocked or this program holds the lock
>
> -1 = and errno=EAGAIN : lock held by another program

Upon process termination, all file locks are released.

Child processes do not inherit file locks.

lockf()

```
#include <unistd.h>
int lockf(int fd,int cmd,off_t len)
```

Purpose: to apply, test, or remove a POSIX lock on an open file

Very similar to flock(). Commands supported include:

F_LOCK set an exclusive lock

F_TLOCK same as F_LOCK but the call never blocks; returns an error if the file is already locked

F_ULOCK unlock indicated section of file

F_TEST test if locked

- 0 = unlocked or this program holds the lock
- -1 = and errno=EAGAIN : lock held by another program

Upon process termination, all file locks are released.

Child processes do not inherit file locks.

(see filelock.c)

flock()

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

int flock(int fd, int operation)
```

Applies or removes advisory locks given a file descriptor.

flock()

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

int flock(int fd, int operation)
```

Applies or removes advisory locks given a file descriptor.

LOCK_SH	places a shared lock.

flock()

```
#include <sys/file.h>
int flock(int fd, int operation)
```

Applies or removes advisory locks given a file descriptor.

LOCK_SH	places a shared lock. More than one process may hold a shared lock at a time

flock()

#include <sys/file.h>

int flock(int fd, int operation)

Applies or removes advisory locks given a file descriptor.

LOCK_SH	places a shared lock.
	More than one process may hold a shared lock at a time
LOCK_EX	places an exclusive lock.

flock()

```
#include <sys/file.h>
int flock(int fd, int operation)
```

Applies or removes advisory locks given a file descriptor.

LOCK_SH	places a shared lock.
	More than one process may hold a shared lock at a time
LOCK_EX	places an exclusive lock.
	Only one process may hold an exclusive lock at a time

flock()

```
#include <sys/file.h>
int flock(int fd, int operation)
```

Applies or removes advisory locks given a file descriptor.

LOCK_SH	places a shared lock.
	More than one process may hold a shared lock at a time
LOCK_EX	places an exclusive lock.
	Only one process may hold an exclusive lock at a time
LOCK_UN	remove an existing lock held by this process

flock(), con't.

- Processes may block if an incompatible lock is held by another process

flock(), con't.

- Processes may block if an incompatible lock is held by another process
- Use bitwise or-ing with LOCK_NB to get non-blocking operations

flock(), con't.

- Processes may block if an incompatible lock is held by another process
- Use bitwise or-ing with LOCK_NB to get non-blocking operations
- Subsequent flock() calls on an already locked file converts the existing lock type to the new lock mode.

flock(), con't.

- Processes may block if an incompatible lock is held by another process
- Use bitwise or-ing with LOCK_NB to get non-blocking operations
- Subsequent flock() calls on an already locked file converts the existing lock type to the new lock mode.
- Locks created by flock()s are preserved across execve()

flock(), con't.

- Processes may block if an incompatible lock is held by another process
- Use bitwise or-ing with LOCK_NB to get non-blocking operations
- Subsequent flock() calls on an already locked file converts the existing lock type to the new lock mode.
- Locks created by flock()s are preserved across execve()
- A single file may not simultaneously have both shared and exclusive locks.

Mandatory Locks

- **Mandatory locks**: (*kernel enforced*) other processes cannot access file

Mandatory Locks

- **Mandatory locks**: (*kernel enforced*) other processes cannot access file
- To get a list of hard-disk partition devices and mounting types using the shell: `df -T`

Mandatory Locks

- **Mandatory locks**: (*kernel enforced*) other processes cannot access file
- To get a list of hard-disk partition devices and mounting types using the shell: `df -T`
- Mount the filesystem using the `mand` option:

```
mount -t ext3 -o mand,rw /dev/sda1 /somedir
```

Mandatory Locks

- **Mandatory locks**: (*kernel enforced*) other processes cannot access file
- To get a list of hard-disk partition devices and mounting types using the shell: `df -T`
- Mount the filesystem using the `mand` option:

```
mount -t ext3 -o mand,rw /dev/sda1 /somedir
```

- One may then enable certain files for mandatory locking:

```
chmod g+s,g-x filenames
```

Mandatory Locks

- **Mandatory locks**: (*kernel enforced*) other processes cannot access file
- To get a list of hard-disk partition devices and mounting types using the shell: `df -T`
- Mount the filesystem using the `mand` option:

  ```
  mount -t ext3 -o mand,rw /dev/sda1 /somedir
  ```
- One may then enable certain files for mandatory locking:

  ```
  chmod g+s,g-x filenames
  ```
- This command sets the `setgid` bit and unsets the `setuid` bit, which doesn't make sense otherwise.

Mandatory Locks

- **Mandatory locks**: (*kernel enforced*) other processes cannot access file
- To get a list of hard-disk partition devices and mounting types using the shell: `df -T`
- Mount the filesystem using the `mand` option:

  ```
  mount -t ext3 -o mand,rw /dev/sda1 /somedir
  ```
- One may then enable certain files for mandatory locking:

  ```
  chmod g+s,g-x filenames
  ```
- This command sets the `setgid` bit and unsets the `setuid` bit, which doesn't make sense otherwise.
- Note that, even with the `mand` option, only selected files may be locked.

Mandatory Locks

- **Mandatory locks**: (*kernel enforced*) other processes cannot access file
- To get a list of hard-disk partition devices and mounting types using the shell: `df -T`
- Mount the filesystem using the `mand` option:

  ```
  mount -t ext3 -o mand,rw /dev/sda1 /somedir
  ```
- One may then enable certain files for mandatory locking:

  ```
  chmod g+s,g-x filenames
  ```
- This command sets the `setgid` bit and unsets the `setuid` bit, which doesn't make sense otherwise.
- Note that, even with the `mand` option, only selected files may be locked.
- One may then use `lockf()` or `fcntl()` to lock the file's access to the current process

Mandatory Locks

- **Mandatory locks**: (*kernel enforced*) other processes cannot access file
- To get a list of hard-disk partition devices and mounting types using the shell: `df -T`
- Mount the filesystem using the `mand` option:

  ```
  mount -t ext3 -o mand,rw /dev/sda1 /somedir
  ```
- One may then enable certain files for mandatory locking:

  ```
  chmod g+s,g-x filenames
  ```
- This command sets the `setgid` bit and unsets the `setuid` bit, which doesn't make sense otherwise.
- Note that, even with the `mand` option, only selected files may be locked.
- One may then use `lockf()` or `fcntl()` to lock the file's access to the current process
- The `flock()` function will *not* trigger mandatory locking

Mandatory Locks

- **Mandatory locks**: (*kernel enforced*) other processes cannot access file
- To get a list of hard-disk partition devices and mounting types using the shell: `df -T`
- Mount the filesystem using the `mand` option:

  ```
  mount -t ext3 -o mand,rw /dev/sda1 /somedir
  ```
- One may then enable certain files for mandatory locking:

  ```
  chmod g+s,g-x filenames
  ```
- This command sets the `setgid` bit and unsets the `setuid` bit, which doesn't make sense otherwise.
- Note that, even with the `mand` option, only selected files may be locked.
- One may then use `lockf()` or `fcntl()` to lock the file's access to the current process
- The `flock()` function will *not* trigger mandatory locking
- Not even root can override file locks (although root can kill the process that holds the lock, indirectly removing them)

Mandatory File Locking

- turn on set-gid bit on file access permission

Mandatory File Locking

- turn on set-gid bit on file access permission
- turn off group execute bit on file access permission

Mandatory File Locking

- turn on set-gid bit on file access permission
- turn off group execute bit on file access permission

```
struct stat statbuf;
if(stat("FILENAME",&statbuf)) perror("stat: ");
else {
        int flag;
        flag= (statbuf.st_mode & ~(S_IXGRP)) |S_ISGID;
        if(chmod("FILENAME",flag)) perror("chmod: ");
        }
```

File Locking Dangers

Deadlock when two or more processes are each waiting for a resource that the other has locked.

File Locking Dangers

Deadlock when two or more processes are each waiting for a resource that the other has locked.

P_a	locks byte 0	P_b locks byte 1

File Locking Dangers

Deadlock when two or more processes are each waiting for a resource that the other has locked.

P_a	locks byte 0	P_b	locks byte 1
P_a	attempts to lock byte 1	P_b	attempts to lock byte 0

File Locking Dangers

Deadlock when two or more processes are each waiting for a resource that the other has locked.

P_a	locks byte 0	P_b locks byte 1
P_a	attempts to lock byte 1	P_b attempts to lock byte 0

Unix will usually detect deadlocks and will choose a process (essentially randomly) for an error return.

File Locking Dangers

Deadlock when two or more processes are each waiting for a resource that the other has locked.

P_a	locks byte 0	P_b locks byte 1
P_a	attempts to lock byte 1	P_b attempts to lock byte 0

Unix will usually detect deadlocks and will choose a process (essentially randomly) for an error return.

- Process termination releases all locks held by that process

File Locking Dangers

Deadlock when two or more processes are each waiting for a resource that the other has locked.

P_a	locks byte 0	P_b locks byte 1
P_a	attempts to lock byte 1	P_b attempts to lock byte 0

Unix will usually detect deadlocks and will choose a process (essentially randomly) for an error return.

- Process termination releases all locks held by that process
- when a descriptor is closed, even though there may be duplicates, all locks on that file are released

File Locking Dangers

Deadlock when two or more processes are each waiting for a resource that the other has locked.

P_a	locks byte 0	P_b locks byte 1
P_a	attempts to lock byte 1	P_b attempts to lock byte 0

Unix will usually detect deadlocks and will choose a process (essentially randomly) for an error return.

- Process termination releases all locks held by that process
- when a descriptor is closed, even though there may be duplicates, all locks on that file are released
- Locks are never inherited via `fork()`

File Locking Dangers

Deadlock when two or more processes are each waiting for a resource that the other has locked.

P_a	locks byte 0	P_b locks byte 1
P_a	attempts to lock byte 1	P_b attempts to lock byte 0

Unix will usually detect deadlocks and will choose a process (essentially randomly) for an error return.

- Process termination releases all locks held by that process
- when a descriptor is closed, even though there may be duplicates, all locks on that file are released
- Locks are never inherited via `fork()`
- locks may be inherited by a new program across an `exec()` (Posix does not require this, but SysV and BSD do)

File Locking Dangers

Deadlock when two or more processes are each waiting for a resource that the other has locked.

P_a	locks byte 0	P_b locks byte 1
P_a	attempts to lock byte 1	P_b attempts to lock byte 0

Unix will usually detect deadlocks and will choose a process (essentially randomly) for an error return.

- Process termination releases all locks held by that process
- when a descriptor is closed, even though there may be duplicates, all locks on that file are released
- Locks are never inherited via fork()
- locks may be inherited by a new program across an exec() (Posix does not require this, but SysV and BSD do)
- Processes may use file locks to insure that only one copy of the program is running:
  open a file, write the process's pid to the file (see getpid()), write lock the file.

Directory API

```
#include <sys/stat.h>
#include <sys/types.h>
int mkdir(const char *pathname, mode_t mode)
```

Returns 0=success, -1=failure (use `perror()` for more details on errors)

Directory API

#include <sys/stat.h>

#include <sys/types.h>

int mkdir(const char *pathname, mode_t mode)

Returns 0=success, -1=failure (use perror() for more details on errors)

Possible failure reasons:

Directory API

#include <sys/stat.h>

#include <sys/types.h>

int mkdir(const char *pathname, mode_t mode)

Returns 0=success, -1=failure (use perror() for more details on errors)

Possible failure reasons:

invalid path	path name exceeds PATH_MAX

Directory API

#include <sys/stat.h>

#include <sys/types.h>

int mkdir(const char *pathname, mode_t mode)

Returns 0=success, -1=failure (use perror() for more details on errors)

Possible failure reasons:

invalid path	path name exceeds PATH_MAX
inadequate permission	directory already exists

Directory API

#include <sys/stat.h>

#include <sys/types.h>

int mkdir(const char *pathname, mode_t mode)

Returns 0=success, -1=failure (use perror() for more details on errors)

Possible failure reasons:

invalid path	path name exceeds PATH_MAX
inadequate permission	directory already exists
illegal mode	quote problems

Directory API

#include <sys/stat.h>

#include <sys/types.h>

int mkdir(const char *pathname, mode_t mode)

Returns 0=success, -1=failure (use `perror()` for more details on errors)

Possible failure reasons:

invalid path	path name exceeds PATH_MAX
inadequate permission	directory already exists
illegal mode	quote problems

- `S_ISGID` and `S_ISUID` are silently stripped from the `mode` if present.

Directory API

#include <sys/stat.h>

#include <sys/types.h>

int mkdir(const char *pathname, mode_t mode)

Returns 0=success, -1=failure (use perror() for more details on errors)

Possible failure reasons:

invalid path	path name exceeds PATH_MAX
inadequate permission	directory already exists
illegal mode	quote problems

- S_ISGID and S_ISUID are silently stripped from the mode if present.
- This function also creates the "." and ".." links necessary to make a usable directory (unlike mknode()).

Directory API

#include <sys/stat.h>

#include <sys/types.h>

int mkdir(const char *pathname, mode_t mode)

Returns 0=success, -1=failure (use perror() for more details on errors)

Possible failure reasons:

invalid path	path name exceeds PATH_MAX
inadequate permission	directory already exists
illegal mode	quote problems

- S_ISGID and S_ISUID are silently stripped from the mode if present.
- This function also creates the "." and ".." links necessary to make a usable directory (unlike mknode()).
- Directory records are specified by one of the following structures:

  BSD,linux: struct dirent   SysV: struct direct

  You may recollect that readdir() returns pointers to one of these on various unix systems.

rmdir()

```
#include <unistd.h>
int rmdir(const char *pathname);
```

- Removes directories (they must be empty, other than for “.” and “..”)

rmdir()

```
#include <unistd.h>
int rmdir(const char *pathname);
```

- Removes directories (they must be empty, other than for “.” and “..”)
- If the path is a symbolic link, it will not be followed

rmdir()

```
#include <unistd.h>
int rmdir(const char *pathname);
```

- Removes directories (they must be empty, other than for "." and "..")
- If the path is a symbolic link, it will not be followed
- If the directory link count goes to zero and no process currently has the directory open, then the directory's resources are freed

rmdir()

```
#include <unistd.h>
int rmdir(const char *pathname);
```

- Removes directories (they must be empty, other than for "." and "..")
- If the path is a symbolic link, it will not be followed
- If the directory link count goes to zero and no process currently has the directory open, then the directory's resources are freed
- Must have write permission on the directory

rmdir()

```
#include <unistd.h>
int rmdir(const char *pathname);
```

- Removes directories (they must be empty, other than for "." and "..")
- If the path is a symbolic link, it will not be followed
- If the directory link count goes to zero and no process currently has the directory open, then the directory's resources are freed
- Must have write permission on the directory
- Must have search (ie. exec) privileges for *every* component of the path

rmdir()

```
#include <unistd.h>
int rmdir(const char *pathname);
```

- Removes directories (they must be empty, other than for "." and "..")
- If the path is a symbolic link, it will not be followed
- If the directory link count goes to zero and no process currently has the directory open, then the directory's resources are freed
- Must have write permission on the directory
- Must have search (ie. exec) privileges for *every* component of the path
- May not remove the current directory

Device File API

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
int mknod(const char *pathname, mode_t mode, dev_t dev);
```

- Creates a file system node

Device File API

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
int mknod(const char *pathname, mode_t mode, dev_t dev);
```

- Creates a file system node
- Permissions are controlled by the process' `umask` item File types: File type must be one of S_IFREG, S_IFCHR, S_IFBLK, S_IFIFO, or S_IFSOCK (regular file); character special file or block special file; a FIFO (named pipe); or a Unix domain socket

Device File API

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
int mknod(const char *pathname, mode_t mode, dev_t dev);
```

- Creates a file system node
- Permissions are controlled by the process' `umask` item File types: File type must be one of S_IFREG, S_IFCHR, S_IFBLK, S_IFIFO, or S_IFSOCK (regular file); character special file or block special file; a FIFO (named pipe); or a Unix domain socket
- Fails if pathname already exists (or is a symbolic link)

Device File API

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
int mknod(const char *pathname, mode_t mode, dev_t dev);
```

- Creates a file system node
- Permissions are controlled by the process' `umask` item File types: File type must be one of S_IFREG, S_IFCHR, S_IFBLK, S_IFIFO, or S_IFSOCK (regular file); character special file or block special file; a FIFO (named pipe); or a Unix domain socket
- Fails if pathname already exists (or is a symbolic link)
- Must be superuser privileged...

Device File API

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
int mknod(const char *pathname, mode_t mode, dev_t dev);

- Creates a file system node
- Permissions are controlled by the process' `umask` item File types: File type must be one of S_IFREG, S_IFCHR, S_IFBLK, S_IFIFO, or S_IFSOCK (regular file); character special file or block special file; a FIFO (named pipe); or a Unix domain socket
- Fails if pathname already exists (or is a symbolic link)
- Must be superuser privileged...

To construct dev : device_id=(major<<8) | minor,

Device File API

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
int mknod(const char *pathname, mode_t mode, dev_t dev);

- Creates a file system node
- Permissions are controlled by the process' `umask` item File types: File type must be one of S_IFREG, S_IFCHR, S_IFBLK, S_IFIFO, or S_IFSOCK (regular file); character special file or block special file; a FIFO (named pipe); or a Unix domain socket
- Fails if pathname already exists (or is a symbolic link)
- Must be superuser privileged...

To construct	dev	:	device_id=(major<<8) | minor,
To extract	major	:	(device_id>>8) & 0xff, and

Device File API

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
int mknod(const char *pathname, mode_t mode, dev_t dev);
```

- Creates a file system node
- Permissions are controlled by the process' `umask` item File types: File type must be one of S_IFREG, S_IFCHR, S_IFBLK, S_IFIFO, or S_IFSOCK (regular file); character special file or block special file; a FIFO (named pipe); or a Unix domain socket
- Fails if pathname already exists (or is a symbolic link)
- Must be superuser privileged...

To construct	dev	: device_id=(major<<8) | minor,
To extract	major	: (device_id>>8) & 0xff, and
To extract	minor	: device_id & 0xff

FIFO File API

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

- The path is the name of the FIFO (named pipe).

FIFO File API

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

- The path is the name of the FIFO (named pipe).
- mode resembles open()'s mode.

FIFO File API

#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);

- The path is the name of the FIFO (named pipe).
- mode resembles open()'s mode.
- Use open/close/read/write/unlink to work with the FIFO.

FIFO File API

#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);

- The path is the name of the FIFO (named pipe).
- mode resembles open()'s mode.
- Use open/close/read/write/unlink to work with the FIFO.
- Writing to a FIFO that has no reading process will cause a SIGPIPE to be sent to the writing process

FIFO File API

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

- The path is the name of the FIFO (named pipe).
- mode resembles open()'s mode.
- Use open/close/read/write/unlink to work with the FIFO.
- Writing to a FIFO that has no reading process will cause a SIGPIPE to be sent to the writing process
- Normally an open for write will block (assuming that O_NONBLOCK is not used) until the FIFO gets opened for reading

Anonymous Pipes

#include <unistd.h>

int pipe(int pipefd[2])

This function creates an anonymous pipe; it does not create a file!

Anonymous Pipes

#include <unistd.h>

int pipe(int pipefd[2])

This function creates an anonymous pipe; it does not create a file!

pipefd[0]	read data from a FIFO with this

Anonymous Pipes

#include <unistd.h>

int pipe(int pipefd[2])

This function creates an anonymous pipe; it does not create a file!

pipefd[0]	read data from a FIFO with this
pipefd[1]	write data to a FIFO with this

Anonymous Pipes

#include <unistd.h>

int pipe(int pipefd[2])

This function creates an anonymous pipe; it does not create a file!

pipefd[0]	read data from a FIFO with this
pipefd[1]	write data to a FIFO with this

- Typically used by a parent process with a child process

Anonymous Pipes

#include <unistd.h>

int pipe(int pipefd[2])

This function creates an anonymous pipe; it does not create a file!

pipefd[0]	read data from a FIFO with this
pipefd[1]	write data to a FIFO with this

- Typically used by a parent process with a child process
- Creates a pipe, a unidirectional data channel (supports interprocess communication)

Anonymous Pipes

#include <unistd.h>

int pipe(int pipefd[2])

This function creates an anonymous pipe; it does not create a file!

pipefd[0]	read data from a FIFO with this
pipefd[1]	write data to a FIFO with this

- Typically used by a parent process with a child process
- Creates a pipe, a unidirectional data channel (supports interprocess communication)
- Data written to the write end of the pipe is buffered by the kernel until it is read by the read end of the pipe

Anonymous Pipes

#include <unistd.h>

int pipe(int pipefd[2])

This function creates an anonymous pipe; it does not create a file!

pipefd[0]	read data from a FIFO with this
pipefd[1]	write data to a FIFO with this

- Typically used by a parent process with a child process
- Creates a pipe, a unidirectional data channel (supports interprocess communication)
- Data written to the write end of the pipe is buffered by the kernel until it is read by the read end of the pipe

(see pipes.c)