DATA STORAGE TECHNOLOGIES & NETWORKS

(CS C446, CS F446 & IS C446)

LECTURE 25– STORAGE

File Descriptor Locking

- Old mechanism (lock file)
  - Process would try to create a lock file. If creation succeeded, then update otherwise wait and try again
  - Drawback
    - CPU time waste (looping over attempting to create locks)
    - Locks left lying if system crashes
    - Super users are always permitted to create files
- Most general locking schemes allow multiple processes to update a file concurrently
- Whole file lock (BSD 4.2, 4.3)
  - Locks are inherited (fork())
  - Release locks only on the last close of a file

- Byte-range locks
  - Mandatory for POSIX standard
  - Release all locks held by a process on a file every time a close system call was done.
  - Problems
    - An application can lock a file then call library routine that opens, reads and closes the locked file.
    - Close call by library routine will release locks held by application
    - A process that have file write permission can only get exclusive lock
- BSD 4.4 support both byte – range and whole file locks
  - Byte range locks applied to processes, whole file locks applied to descriptors (descriptors are shared with child processes)
    - Whole file locks are inherited
    - Byte range locks are not inherited

- Locks
  - Mandatory locks
    - Enforced for every process without choice
    - Need some policy implementation in kernel
  - Advisory locks
    - Enforces for only those processes that request them
    - Effective only when all programs accessing a file use the locking mechanism
    - Requires cooperation of processes
    - Policy is left to the user program

- Cooperative programs (on range of bytes)
  - Advisory shared lock
    - Called Multiple readers lock
    - Multiple holders could simultaneously exist
  - Advisory exclusive lock
    - Called single writer lock
    - Only one process could hold the lock
  - Both shared and exclusive lock can not be present at the same time.
- Process blocks if the lock cannot be obtained immediately
- Implementation of a lock is per file system basis

- flock
  - **int flock(int *fd*, int *operation*);**
  - Apply or remove an advisory lock on the open file specified by *fd*. The argument *operation* is :
  - **LOCK_SH**
    - Place a shared lock. More than one process may hold a shared lock for a given file at a given time.
  - **LOCK_EX**
    - Place an exclusive lock. Only one process may hold an exclusive lock for a given file at a given time.
  - **LOCK_UN**
    - Remove an existing lock held by this process.

- Locks created by **flock**() are associated with an open file table entry
- Duplicate file descriptors created by fork() or dup refer to the same lock, and this lock may be modified or released using any of these descriptors
- Lock is released either by explicit **LOCK_UN** operation on any of these duplicate descriptors, or when all such descriptors have been closed.
- A process may only hold one type of lock (shared or exclusive) on a file. Subsequent **flock**() calls on an already locked file will convert an existing lock to the new lock mode.
- Locks created by **flock**() are preserved across an execve()
- A shared or exclusive lock can be placed on a file regardless of the mode in which the file was opened.

- **flock**() does not lock files over NFS. Use fcntl() instead
- **flock**() does not detect deadlock
- **flock**() is implemented as system call
- **flock**() places advisory locks only; given suitable permissions on a file, a process is free to ignore the use of **flock**() and perform I/O on the file.
- Converting a lock (shared to exclusive, or vice versa) is not guaranteed to be atomic
  - the existing lock is first removed, and then a new lock is established.
  - Between these two steps, a pending lock request by another process may be granted, with the result that the conversion either blocks, or fails if **LOCK_NB** was specified

- **fcntl – Duplicating a file descriptor**
  - int fcntl(int fd, int cmd);
  - int fcntl(int fd, int cmd, long arg);
  - int fcntl(int fd, int cmd, struct flock *lock);
- F_DUPFD (*long*) Find the lowest numbered available file descriptor greater than or equal to *arg* and make it be a copy of *fd*.
- This is different from dup2, which uses exactly the descriptor specified.
- On success, the new descriptor is returned.

- FD_CLOEXEC (*long*; since Linux 2.6.24)
  - Close on exec flag
  - If FD_CLOEXEC is 0 then file descriptor will remain open across an execve; otherwise it will be closed
- F_GETFD (*void*) Read the file descriptor flags
- F_SETFD (*long*) Set the file descriptor flags to the value specified by *arg*.
- F_GETFL (*void*) Get the file access mode and the file status flags
- F_SETFL (*long*) Set the file status flags to the value specified by *arg*.

Advisory locking

- F_GETLK, F_SETLK and F_SETLKW are used to acquire, release, and test for the existence of record locks (also known as file-segment or file-region locks).
- The third argument lock is a pointer to a structure that has at least the following fields (in unspecified order).

```
struct flock {
        short l_type;  /* Type of lock: F_RDLCK,F_WRLCK, F_UNLCK */
        short l_whence;  /* How to interpret l_start:
                            SEEK_SET, SEEK_CUR, SEEK_END */
        off_t l_start;   /* Starting offset for lock */
        off_t l_len;     /* Number of bytes to lock */
        pid_t l_pid;     /* PID of process blocking our lock
                            (F_GETLK only) */
};
```

l_whence, l_start, and l_len fields of this structure specify the range of bytes we wish to lock

- F_GETLK
  - Get the first lock that blocks the lock description pointed to by the third argument, *arg*, taken as a pointer to a *struct flock*
- F_SETLK
  - Set or clear a file segment lock according to the lock description pointed to by the third argument, *arg*, taken as a pointer to a *struct flock*
- F_SETLKW
  - This command is the same as F_SETLK except that if a shared or exclusive lock is blocked by other locks, the process waits until the request can be satisfied.

- When a shared lock has been set on a segment of a file, other processes can set shared locks on that segment or a portion of it.
  - A shared lock prevents any other process from setting an exclusive lock on any portion of the protected area.
  - A request for a shared lock fails if the file descriptor was not opened with read access.
- An exclusive lock prevents any other process from setting a shared lock or an exclusive lock on any portion of the protected area.
  - A request for an exclusive lock fails if the file was not opened with write access.

Mandatory locking

- Non-POSIX
- Mandatory locks are enforced for all processes
- To make use of mandatory locks, mandatory locking must be enabled both on the file system that contains the file to be locked, and on the file itself
- Mandatory locking is enabled on a file system using the "-o mand" option to mount(), or the MS_MANDLOCK flag for mount()
- Mandatory locking is enabled on a file by disabling group execute permission on the file and enabling the set-group-ID permission bit