Input & Output

File Management

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

File Management Guidelines

- To manage a file it is always possible to use
  - C library functions
    - Are generally high level and easy to use
    - The code will be portable on non-Windows systems
    - Field and character-oriented funtions do not have direct Windows equivalent
    - Generic calls (ASCII and Unicode) can be easily used but the portability advantage will be lost
  - Windows functions
    - Enable advanced features
      - File security attributes, 32 versus 64-bit manipulation, file locking, directory manipulations, etc.
    - Optimize specific operations

File Management Guidelines

- File management in Windows includes
  - Basic file processing functions
    - CreateFile, ReadFile, WriteFile, CloseHandle
  - Random access functions
    - SetFilePointer (SetFilePointerEx), overlapped data structure, GetFileSizeEx
  - File locking
    - LockFileEx, UnlockFileEx

First form of threads (or processes) synchronization

Create a File

```
HANDLE CreateFile (
  LPCTSTR lpName,
  DWORD dwAccess,
  DWORD dwShareMode,
  LPSECURITY_ATTRIBUTES lpsa,
  DWORD dwCreate,
  DWORD dwAttrsAndFlags,
  HANDLE hTemplateFile
);
```

Numerous advanced options
(not fully described here)

- Returned value
  - A HANDLE to an open file object
  - INVALID_HANDLE_VALUE in case of failure

Create a File

```
HANDLE CreateFile (
  LPCTSTR lpName,
  DWORD dwAccess,
  DWORD dwShareMode,
  LPSECURITY_ATTRIBUTES lpsa,
  DWORD dwCreate,
  DWORD dwAttrsAndFlags,
  HANDLE hTemplateFile
);
```

- Parameters
  - lpName
    - Pointer to file name
    - Length limited to MAX_PATH
    - If "\\?\" is used as prefix it is possible to use name as long as 32K (UNICODE coding)
  - dwAccess
    - Specify the read and write access
      - Use GENERIC_READ or GENERIC_WRITE (the term "GENERIC" is somehow redundant)
    - Combine flags with the OR operator "|"
      - GENERIC_READ | GENERIC_WRITE

Create a File

```
HANDLE CreateFile (
  LPCTSTR lpName,
  DWORD dwAccess,
  DWORD dwShareMode,
  LPSECURITY_ATTRIBUTES lpsa,
  DWORD dwCreate,
  DWORD dwAttrsAndFlags,
  HANDLE hTemplateFile
);
```

- dwShareMode
  - File sharing mode
  - Bit-wise OR ("|") of table flags

Value	Action / Meaning
0	Cannot be shared. Not even the same process can open another handle.
FILE_SHARE_READ	Other processes can read concurrently
FILE_SHARE_WRITE	Other processes can write concurrently

Create a File

```
HANDLE CreateFile (
  LPCTSTR lpName,
  DWORD dwAccess,
  DWORD dwShareMode,
  LPSECURITY_ATTRIBUTES lpsa,
  DWORD dwCreate,
  DWORD dwAttrsAndFlags,
  HANDLE hTemplateFile
);
```

- lpsa
  - Usually NULL
  - It points to a SECURITY_ATTRIBUTES structure (advanced topic on security)
  - Alert: Everyone has full control of a newly created file

Create a File

```
HANDLE CreateFile (
  LPCTSTR lpName,
  DWORD dwAccess,
  DWORD dwShareMode,
  LPSECURITY_ATTRIBUTES lpsa,
  DWORD dwCreate,
  DWORD dwAttrsAndFlags,
  HANDLE hTemplateFile
);
```

- dwCreate
  - Create a file, overwrite existing file, etc.
  - There is no **append** mode
    - Set file pointer to the end of file

Value	Action / Meaning
CREATE_NEW	Fails if the file exists
CREATE_ALWAYS	An existing file will be overwritten
OPEN_EXISTING	Fail if the file does not exist
OPEN_ALWAYS	Open the file or create it if it doesn't exist
TRUNCATE_EXISTING	File length will be set to zero

Create a File

```
HANDLE CreateFile (
   LPCTSTR lpName,
   DWORD dwAccess,
   DWORD dwShareMode,
   LPSECURITY_ATTRIBUTES lpsa,
   DWORD dwCreate,
   DWORD dwAttrsAndFlags,
   HANDLE hTemplateFile
);
```

- dwAttrsAndFlags
  - 32 possibile different flags and attributes
  - Attributes are properties of the files themselves
  - The main flags are the following

Value	Action / Meaning
FILE_ATTRIBUTE_NORMAL	No other attributes are set
FILE_ATTRIBUTE_READONLY	Cannot write or delete
FILE_FLAG_OVERLAPPED	For asynch I/O
FILE_FLAG_SEQUENTIAL_SCAN	Provide performance hints
FILE_FLAG_RANDOM_ACCESS	Provide performance hints

Create a File

- hTemplateFile
  - Usually NULL
  - It can be a handle of an open file (opened in GENERIC_READ mode)
  - It forces CreateFile to use the same attributes of that file to create the new file

```
HANDLE CreateFile (
  LPCTSTR lpName,
  DWORD dwAccess,
  DWORD dwShareMode,
  LPSECURITY_ATTRIBUTES lpsa,
  DWORD dwCreate,
  DWORD dwAttrsAndFlags,
  HANDLE hTemplateFile
);
```

Guidelines

- There is an **OpenFile**() function
  - Don't use it
  - It's obsolete and for 16-bit applications
- Flags are associated with the file HANDLE
  - Different HANDLEs referring to the same file can have different flags
    - One HANDLE is "overlapped," another not
    - One HANDLE has FILE_FLAG_SEQUENTIAL_SCAN and another FILE_FLAG_RANDOM_ACCESS
  - Different Ts (Ps) can manage a file using the sme or different handles

Read a File

```
BOOL ReadFile(
  HANDLE hFile,
  LPVOID lpBuffer,
  DWORD nNumberOfBytesToRead,
  LPDWORD lpNumberOfBytesRead,
  LPOVERLAPPED lpOverlapped
);
```

Numerous advanced options
(not fully described here)

- Return
  - TRUE if the read succeeds
    - Even if no bytes were read due to an attempt to read past the end of file
  - FALSE indicates an invalid handle
    - A handle without GENERIC_READ access, etc.

Read a File

```
BOOL ReadFile(
  HANDLE hFile,
  LPVOID lpBuffer,
  DWORD nNumberOfBytesToRead,
  LPDWORD lpNumberOfBytesRead,
  LPOVERLAPPED lpOverlapped
);
```

- Parameters
  - hFile
    - File handle with GENERIC_READ access
  - lpBuffer
    - Memory buffer to receive the input data
  - nNumberOfBytesToRead
    - Number of bytes you expect to read

Read a File

```
BOOL ReadFile(
  HANDLE hFile,
  LPVOID lpBuffer,
  DWORD nNumberOfBytesToRead,
  LPDWORD lpNumberOfBytesRead,
  LPOVERLAPPED lpOverlapped
);
```

- *lpNumberOfBytesRead
  - Actual number of bytes transferred
  - Zero indicates end of file
- lpOverlapped
  - Points to the **OVERLAPPED** data structure
  - Often NULL
  - Not NULL for **random** file access

Write a File

```
BOOL WriteFile (
  HANDLE hFile,
  LPCVOID *lpBuffer,
  DWORD nNumberOfBytesToWrite,
  LPDWORD lpNumberOfBytesWritten,
  LPOVERLAPPED lpOverlapped
);
```

- Return
  - TRUE if the function succeeds
  - FALSE otherwise
- Parameters
  - See the **ReadFile** function

Close a File

```
BOOL CloseHandle (
  HANDLE hObject
);
```

- Return
  - TRUE if the function succeeds
  - FALSE otherwise
- This function is general purpose and will be used to close handles to many different object types

Convenience function to Copy a File

```
BOOL CopyFile (
  LPCTSTR lpExistingFile,
  LPCTSTR lpNewFile,
  BOOL fFailIfExists
);
```

- Copy an old file into a new one
- Parameters
  - lpExistingFile existing file name
  - lpNewFile new file name
  - If **fFailIfExists** is FALSE the source file will replace an existing file

Convenience function to Copy a File

```
BOOL CopyFile (
  LPCTSTR lpExistingFile,
  LPCTSTR lpNewFile,
  BOOL fFailIfExists
);
```

- This "convenience function"
  - It is easier to use
  - It provides better performance
  - It preserves the file's attributes, including time stamps

Example

Copy a file into an equivalent one

See previous section (and demo) for further comments

```
HANDLE hIn, hOut;
DWORD nIn, nOut;
TCHAR c;

hIn = CreateFile (argv[1], GENERIC_READ, 0, NULL,
  OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
hOut = CreateFile (argv[2], GENERIC_WRITE, 0, NULL,
  CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
if (hIn==INVALID_HANDLE_VALUE||hOut==INVALID_HANDLE_VALUE) {
    printf ("Error opening file.\n");
    return 1;
}

while (ReadFile(hIn,&c,sizeof(c),&nIn,NULL) && nIn > 0) {
  WriteFile (hOut, &c, sizeof(c), &nOut, NULL);
}

CloseHandle (hIn);
CloseHandle (hOut);
```

File Pointers

- Windows (like UNIX) indicates the **current** byte location in the file
  - The file pointer is associated with the HANDLE, not the file
    - For each handle (even to the same file) there is a file pointer

Pay attention to concurrent manipulation: 1 versus N threads R/W the same file

  - The pointer
    - Is initialized to zero by **CreateFile**
    - Advances with each read and write operation

File Pointers

- In Windows
  - It is possible to explicitly modify file pointers to perform **random walks** on the file
  - Random walks can be implemented using **two** different strategies
    - Setting the current position using a function **before** reading or writing with RF and WF system calls
      - SetFilePointer
      - SetFilePointerEx

      Obsolete (complex manipulation of 64-bit pointers) but still used
    - Setting the current position using the overlapped data structure **while** reading or writing

Setting File Pointers

```
DWORD SetFilePointer (
  HANDLE hFile,
  LONG lDistanceToMove,
  PLONG lpDistanceToMoveHigh,
  DWORD dwMoveMethod
);
```

32-LSBs

- Return
  - The low-order part (DWORD, unsigned) of the new file pointer
    - The high-order portion of the new file pointer goes to the DWORD indicated by **lpDistanceToMoveHigh** (if this parameter is non-NULL)
  - In case of error, the return value is 0xFFFFFFFF

32-MSBs

The return value can be a value or an error code … confused

File Pointers

```
DWORD SetFilePointer (
  HANDLE hFile,
  LONG lDistanceToMove,
  PLONG lpDistanceToMoveHigh,
  DWORD dwMoveMethod
);
```

- Parameters
  - hFile
    - Handle of an open file with read and/or write access
  - lDistanceToMove
    - LONG (32bits) signed distance to move or unsigned file position
  - *lpDistanceToMoveHigh
    - High-order portion of the move distance
    - Can be NULL for "small" files (<4GBytes)

32-LSBs

32-MSBs

File Pointers

```
DWORD SetFilePointer (
  HANDLE hFile,
  LONG lDistanceToMove,
  PLONG lpDistanceToMoveHigh,
  DWORD dwMoveMethod
);
```

- dwMoveMethod
  - Specifies one of the following modes

Value	Action / Meaning
FILE_BEGIN	Position from the start of file
FILE_CURRENT	Move pointer forward or backward
FILE_END	Position backward (or forward) from end of file

File Pointers with 64-bit Arithmetic

- With **SetFilePointer** file pointers are specified with two 32-bit parts
- For 64-bit file systems, file pointers are long 64 bits
  - Large files are increasingly important in many applications
  - However, many users will only require "short" (< 4GBytes) files
- Function **SetFilePointerEx** is the first of many "extended" functions
  - There is no consistency in the extended features or parameters

File Pointers with 64-bit Arithmetic

- **SetFilePointerEx** uses the LARGE_INTEGER data type for 64-bit file positions
- LARGE_INTEGERs are C **union** of
  - A LONGLONG type named QuadPart

  and two 32-bit quantities
  - A DWORD (32-bit unsigned integer) type named LowPart
  - A LONG (32-bit signed integer) type named HighPart

File Pointers with 64-bit Arithmetic

```
typedef union _LARGE_INTEGER {
  struct { DWORD LowPart; LONG  HighPart; };
  struct { DWORD LowPart; LONG  HighPart; } u;
  LONGLONG QuadPart;
} LARGE_INTEGER, *PLARGE_INTEGER;
```

LONGLONG QuadPart	
LONG HighPart	DWORD LowPart

64 bits

32 bits

32 bits

A **union** is a special data type available in C that allows to store different data types in the same memory area (overlapped, shared)

File Pointers with 64-bit Arithmetic

```
LARGE_INTEGER var;

var.QuadPart = ...
var.LowPart = ...
var.Highart = ...
```

Manipulate 64 bits

Get lower 32 bits

Get higher 32 bits

64 bits

32 bits

LONGLONG QuadPart	
LONG HighPart	DWORD LowPart

32 bits

Sometimes is useful to access 64 bits (i.e., address increment, etc.), sometimes it is useful to access two 32-bit fields

Function SetFilePointerEx

```
BOOL SetFilePointerEx (
  HANDLE hFile,
  LARGE_INTEGER liDistanceToMove,
  PLARGE_INTEGER lpNewFilePointer,
  DWORD dwMoveMethod
);
```

- Similar to **SetFilePointer** but requires
  - A large integer (liDistanceToMove) to set the required position
  - A large integer pointer (lpNewFilePointer) to return the actual position

Example

Update (read-modify-write) the same record within file hFile

```
. . .
record_t r;
LARGE_INTEGER FilePos;
PLARGE_INTEGER lpFilePos;
DWORD n, nRd, nWrt;
. . .

FilePos.QuadPart = n * sizeof (record_t);
SetFilePointerEx(hFile, FilePos, lpFilePos, FILE_BEGIN);
ReadFile(hFile, &r, sizeof(record_t), &nRd, NULL);
...

SetFilePointerEx(hFile, FilePos, lpFilePos, FILE_BEGIN);
WriteFile(hFile, &r, sizeof(record_t), &nWrt, NULL);
```

Set current position onto record n

Set file pointer to that position

Same position (**must** be reset)

Pay attention to share hFile

Overlapped Data Structure

- Windows provides another way to specify file position
- The final parameter of ReadFile and WriteFile is an overlapped data structure
  - This structure has offset fields to specify the starting position of the current read/write operation

Overlapped Data Structure

```
type def struct _OVERLAPPED {
  DWORD Internal;
  DWORD InternalHigh;
  DWORD Offset;
  DWORD OffsetHigh;
  HANDLE hEvent;
] OVERLAPPED;
```

- The overlapped structure has 5 data fields
  - Internal and InternalHigh
    - Those two fields are reserved
    - Do not use

Overlapped Data Structure

- Offset and OffsetHigh
  - Low order (32-LSBs)
  - High order address (32-MSBs)

New position is always set from "FILE_BEGIN"

- hEvent
  - Field is used with asynchronous I/O
  - Must be NULL

```
type def struct _OVERLAPPED {
  DWORD Internal;
  DWORD InternalHigh;
  DWORD Offset;
  DWORD OffsetHigh;
  HANDLE hEvent;
] OVERLAPPED;
```

Example

Define proper data-structure

Set those 2 fields

```
OVERLAPPED ov = { 0, 0, 0, 0, NULL };
record_t r;
LONGLONG n;
LARGE_INTEGER FilePos;
DWORD nRd, nWrt;
. . .
/* Update reference position (record n) */
FilePos.QuadPart = n * sizeof (record_t);
ov.Offset = FilePos.LowPart;
ov.OffsetHigh = FilePos.HighPart;
ReadFile(hFile, &r, sizeof(record_t), &nRd, &ov);
...
/* Update the record. */
...
WriteFile(hFile, &r, sizeof(record_t), &nWrt, &ov);
```

Use LARGE_INTEGER as before

Set position

Set position again (structure ov does not change)

Getting the File Size

- To append new record to the end of an existing file, it is enough to set
  - Offset and OffsetHigh to 0xFFFFFFFF, before performing a write operation
- Anyhow, to know the file size it is possible to use **SetFilePointerEx**
  - Set the position 0 bytes from the end of the file
  - Get the **lpNewFilePointer** returned

```
BOOL SetFilePointerEx (
  HANDLE hFile,
  LARGE_INTEGER liDistanceToMove,
  PLARGE_INTEGER lpNewFilePointer,
  DWORD dwMoveMethod
);
```

Getting the File Size

```
BOOL GetFileSizeEx (
  HANDLE hFile,
  PLARGE_INTEGER lpFileSize
);
```

- To know a file size in a more directed fashion
- Return value
  - FALSE in case of error
- Parameters
  - hFile is the file handle (of an already opened file)
  - lpFileSize the pointer to the 64-bit value representing the file size

I/O and Synchronization

- An important aspect of concurrent programming is synchronization of access to shared objects such as files
- All previous input/output operations are **thread-syncronous**
  - The thread waits until input/output completes
  - To allow a thread to continue without waiting for an input/output operation to complete it is necessary to use asynchronous system calls

I/O and Synchronization

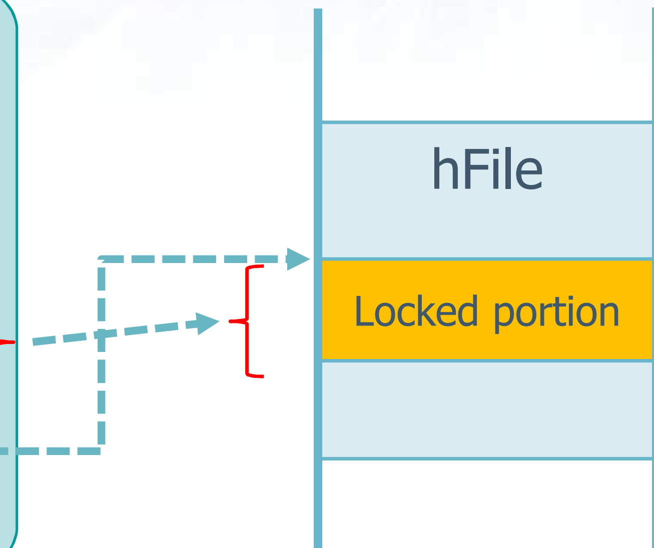
- File locking is a limited form of synchronization
- In Windows it is possible to lock a file so that no other P or T can access the same file area
- Lock **belongs** to a **process**, and it is possible to
  - Lock
    - An entire file
    - Part of a file
  - Obtain
    - A shared, i.e., multiple reader (read-only) access
    - An exclusive, i.e., single reader-writer, access

File Locking

- Conflicting locks cannot be created on a file
- Locks cannot overlap
- The logic to manipulate lock is the following
  - A process (or a thread within a process)
    - Gets a lock
    - Waits for a lock to become available
    - If it does not want to wait, it returns immediately
  - When more than one therad want to get a lock on (possibly) a different section of the file
    - Each thread must use a **different** file handles

Locking a File

```
BOOL LockFileEx (
  HANDLE hFile,
  DWORD dwFlags,
  DWORD dwReserved,
  DWORD nNumberOfBytesToLockLow,
  DWORD nNumberOfBytesToLockHigh,
  LPOVERLAPPED lpOverlapped
);
```



- **LockFileEx** locks a byte range in an open file
- Return
  - A non-zero value (TRUE), if it succeeds
  - A zero value (FALSE), if it fails

Locking a File

```
BOOL LockFileEx (
  HANDLE hFile,
  DWORD dwFlags,
  DWORD dwReserved,
  DWORD nNumberOfBytesToLockLow,
  DWORD nNumberOfBytesToLockHigh,
  LPOVERLAPPED lpOverlapped
);
```

- Parameters
  - hFile
    - Handle of an open file
    - The file must have an access such as
      - GENERIC_READ

      or
      - GENERIC_WRITE

Locking a File

```
BOOL LockFileEx (
  HANDLE hFile,
  DWORD dwFlags,
  DWORD dwReserved,
  DWORD nNumberOfBytesToLockLow,
  DWORD nNumberOfBytesToLockHigh,
  LPOVERLAPPED lpOverlapped
);
```

- dwFlags
  - Lock mode and how to wait for the lock to become available
  - It may get one or more of the following values
    - LOCKFILE_EXCLUSIVE_LOCK
      - If present, the request is for an exclusive (read-write) lock
      - Otherwise, the request is for a shared (read only) lock
    - LOCKFILE_FAIL_IMMEDIATELY
      - If present, specifies that the function should return immediately with a FALSE if the lock cannot be acquired
      - Otherwise, the call blocks until the lock becomes available

Locking a File

```
BOOL LockFileEx (
  HANDLE hFile,
  DWORD dwFlags,
  DWORD dwReserved,
  DWORD nNumberOfBytesToLockLow,
  DWORD nNumberOfBytesToLockHigh,
  LPOVERLAPPED lpOverlapped
);
```

- dwReserved
  - Reserved
  - Must be set to zero
- nNumberOf…Low
  - Low-order 32 bits

    of the length of the **byte range to lock**
- nNumberOfBytesLockHigh
  - High-order 32 bits of the length of the **byte range to lock**

nNumbeOfBytesToLockLow/High define the size (the number of bytes) of the locked region

Locking a File

```
BOOL LockFileEx (
  HANDLE hFile,
  DWORD dwFlags,
  DWORD dwReserved,
  DWORD nNumberOfBytesToLockLow,
  DWORD nNumberOfBytesToLockHigh,
  LPOVERLAPPED lpOverlapped
);
```

- lpOverlapped
  - Points to an OVERLAPPED data structure containing the offset of the **beginning** of the lock range
    - Offset is the low part offset
    - OffsetHigh is the high part offset
    - The HANDLE hEvent should be set to 0

```
type def struct _OVERLAPPED {
  ...
  DWORD Offset;
  DWORD OffsetHigh;
  ...
] OVERLAPPED;
```

lpOvelapped
defines the starting position (in term of bytes) of the locked region

Unlocking a File

```
BOOL UnlockFileEx (
  HANDLE hFile,
  DWORD dwReserved,
  DWORD nNumberOfBytesToLockLow,
  DWORD nNumberOfBytesToLockHigh,
  LPOVERLAPPED lpOverlapped
);
```

Any file lock is removed with a corresponding UnlockFileEx call

- The unlock must use exactly the same range as a preceding lock
- See **LockFileEx** for
  - Return value
  - Parameters
    - Notie that the field “DWORD dwFlags” is not present

Example

Define proper data-structures

Starting position for lock

Range size for lock

```
record_t ...;
HANDLE hFile;
LARGE_INTEGER filePos, fileReserved;
OVERLAPPED ov = {0, 0, 0, 0, NULL};

. . .

hFile = CreateFile (...);

. . .

filePos.QuadPart = n * sizeof (record_t);
fileReserved.QuadPart = m * sizeof (record_t);
```

n = number of records to skip
record_t = struct defining a file record

m = number of records to lock
record_t = struct defining a file record

Example

```
. . .
ov.Offset = filePos.LowPart;
ov.OffsetHigh = filePos.HighPart;
ov.hEvent = (HANDLE) 0;

LockFileEx (hFile, LOCKFILE_EXCLUSIVE_LOCK,
  0, fileReserved.LowPart, fileReserved.HighPart,
  &ov);
. . .

UnlockFileEx (hFile, 0, fileReserved.LowPart,
  fileReserved.HighPart, &ov);
```

Set overlapping data structure fields

Locking

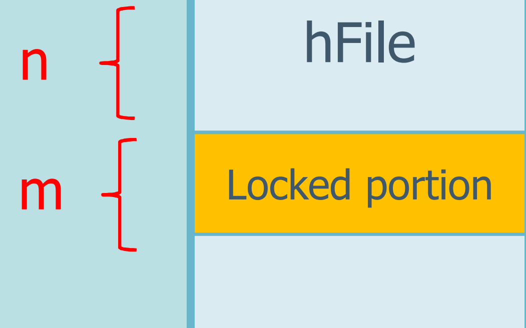
Exclusive Lock

Reserved Field

Starting at …

Size to lock

Unlocking



Guidelines

- Repeated Lock Request
  - If a lock is present
  - When a new lock request is granted or refused ?

Exiting Lock	Requested Lock Type	
	Shared Lock	Exclusive Lock
None	Granted	Granted
Shared lock	Granted	Refused
Exclusive lock	Refused	Refused

Guidelines

- I/O Request on a Lock
  - If a lock is present
  - When a new read or write opeation is granted or refused ?

Exiting Lock	Requested I/O Operation Read	Write
None	Succeeds	Succeeds
Shared lock	Succeeds	Succeeds for the lock owner. Refused otherwise
Exclusive lock	Succeeds for the lock owner. Refused otherwise	Succeeds for the lock owner. Refused otherwise

Guidelines

- Every successful file lock must be followed by a successful file unlock
  - There must be a 1-to-1 matchig between lock and unlock operations
- Locks cannot overlap
  - They would conflict
- It is possible to lock beyond the file's end
  - This process can be useful to extend the file
- A lock may fail if a portion of record is locked
  - The R/W operation will operate only when the portion is unlocked

Guidelines

- File locking can produce
  - Starvation
    - Thread A and B periodically obtain a shared lock whereas C is waiting forever for an exclusive lock
  - Deadlock
    - Thread A is waiting for B to unlock and vice-versa (on a different file region)