파일락(File Lock)은 파일의 일부분을 락할 수 있는 이유로 레코드락(Record Lock)이라고도 합니다.

파일락을 사용하게 되는 흐름은 다음과 같습니다.

|  |  |
| --- | --- |
| [상황] Pa와 Pb라는 프로세스(혹은 쓰레드)가 존재하며 F라는 파일을 공통으로 사용하려고 합니다. | |
| 1. Pa가 F를 사용하기 위하여 F를 오픈합니다.      3. Pa가 F에 대해 File Lock을 요청합니다.        6. F에 대한 Lock을 획득한 Pa는 F를 사용합니다.  7. F에 대한 Lock을 반환합니다. | 2. Pb가 F를 사용하기 위하여 F를 오픈합니다.    4. Pb가 F에 대해 File Lock을 요청합니다.  5. F에 대한 Lock을 획득하지 못하고 대기합니다.        8. F에 대한 Lock을 획득하고 이를 사용합니다.  9. F에 대한 Lock을 반환합니다. |

File을 Lock 하는 방법이 Windows와 Unix/Linux와 차이가 있습니다.

우선 Unix/Linux에서의 File Lock에 대해서 설명하겠습니다.

**1. Unix/Linux에서의 File Lock**

Unix/Linux계열에서는 파일을 Lock하기 위해서 lockf()함수와 fcntl()함수를 사용할 수 있습니다.

다음과 같은 원형을 갖고 있습니다.

int lockf(int fd, int cmd, off\_t len);

int fcntl(int fd, int cmd, struct flock \* lock);

두 함수의 차이는 lockf()는 쓰기락(Write Lock)과 읽기락(Read Lock)을 구별하지 않지만 fcntl()은 쓰기락과 읽기락을 구별합니다.

**1) int lockf(int fd, int cmd, off\_t len);**

lockf은 쓰기 가능으로 열린 파일기술자(fd)의 현재 위치부터 len만큼의 길이의 공간을 cmd에 따른 행동을 하게 됩니다.

cmd는 다음의 값이 될 수 있습니다.

- F\_ULOCK: Lock을 해제합니다.

- F\_LOCK: Lock을 요청합니다. 이미 Lock이 걸려 있다면 해제될 때까지 기다립니다.

- F\_TLOCK: Lock이 걸려 있는지 살피고 걸려있지 않다면 Lock을 겁니다. 단, Lock이 걸려있다면 기다리지 않고 -1을 리턴하고 errno에 EACCES(EAGAIN)값을 갖게 합니다.

- F\_TEST: Lock이 걸려 있는지 아닌지를 조사합니다. Lock이 걸려있지 않다면 0을 리턴합니다.

만약 걸려있다면 -1을 리턴하고 errno에 EACCES(EAGAIN)값을 갖게 합니다.

만약 len이 0이라면 현재 위치부터 뒷부분 모두를 락의 대상으로 하게 됩니다.

다음과 같은 소스로 Pa.c를 만듭니다.

|  |
| --- |
| #include <stdio.h>  #include <fcntl.h>  #include <unistd.h>  int main(int argc, char\* argv[])  {  int fd, ret;  fd=open("locktest", O\_RDWR);  ret=lockf(fd, F\_LOCK, 10L);  printf("Locked By Pa: %d\n", ret);  sleep(5);  close(fd);  printf("Released By Pa\n");  return 0;  } |

다음과 같은 소스로 Pb.c를 만듭니다.

|  |
| --- |
| #include <stdio.h>  #include <fcntl.h>  #include <unistd.h>  int main(int argc, char\* argv[])  {  int fd, ret;  fd=open("locktest", O\_RDWR);  ret=lockf(fd, F\_LOCK, 10L);  printf("Locked By Pb: %d\n", ret);  sleep(5);  close(fd);  printf("Released By Pb\n");  return 0;  } |

그리고 locktest 라는 파일을 현재 디렉토리에 만들어 놓습니다.

명령으로 다음과 같이 입력합니다.

cc -o Pa Pa.c  
cc -o Pb Pb.c  
Pa&  
Pb&

이렇게 실행하면 Pa가 먼저 locktest파일에 락을 걸고 5초가 지난후 Lock을 해제하면 Pb가 락을 거는 것을 확인하실 수 있습니다.

위의 소스 중 Pb.c에서 lockf()의 cmd파라메터를 F\_TLOCK, F\_TEST등을 설정하여 똑같이 실행해 보고 ret값을 확인해 보세요.

또한 위의 예제에서는 lockf()의 cmd파라메터에 F\_UNLOCK을 사용하여 락 해제를 하지 않았습니다. 이렇듯 파일락이 걸린 파일기술자(fd)가 닫힘으로서 자동으로 해제됩니다.

**2) int fcntl(int fd, int cmd, struct flock \* lock);**

fcntl()은 lockf()와는 다르게 쓰기락과 읽기락을 구별합니다.

뭐 다들 아시겠지만 쓰기락과 읽기락의 차이의 차이를 이야기 한다면...

쓰기락이 걸린 파일의 해당 위치에는 어떠한 다른 락도 허용되지 않습니다. 반면에 읽기락이 걸린 경우 같은 위치에 또 다른 읽기락은 허용하지만 쓰기락은 허용되지 않습니다.

읽기락을 위해서는 O\_RDONLY나 O\_RDWR로 fd가 열려야 하며 쓰기락을 위해서는 O\_WRONLY나 O\_RDWR로 fd가 열려야 합니다.

cmd에 파일락과 관련해서는 다음과 같은 값이 사용됩니다.

- F\_GETLK: 락 정보를 얻는다.

- F\_SETLK: 락을 시도한다. (락을 걸 수 없다면 기다리지 않고 바로 리턴한다.)

- F\_SETLKW:  락을 시도한다. (락을 걸 수 없다면 락을 걸 수 있을 때 까지 기다린다.)

락의 형태와 영역을 정하기 위해 flock 구조체를 사용하게 되는데 flock구조체는 다음과 같이 생겼습니다.

struct flock{

    short l\_type;

    short l\_whence;

    long l\_start;

    long l\_len;

    short l\_pid;

}

l\_type은 적용하고자 하는 락의 형태로 F\_RDLCK(읽기락), F\_WRLCK(쓰기락), F\_UNLCK(락해제)를 지정할 수 있습니다.

l\_whence, l\_start, l\_len은 락을 적용할 파일을 위치를 지정하게 되는데....

l\_whence는 0(파일의 처음), 1(파일기술자의 현 위치), 2(파일의 맨끝)을 지정할 수 있습니다.

l\_start는 l\_whence에 대한 상대 위치로 락의 시작 위치를 지정합니다.

l\_len은 l\_start부터 락을 걸 영역을 byte크기로 지정합니다. 만약 0을 지정하면 l\_start부터 뒷부분 모두를 락의 대상으로 하게 됩니다.

l\_pid는 cmd에 F\_GETLK을 지정한 경우에만 유효하며 이전에 락을 건 프로세스의 번호가 저장되게 됩니다. 이때는 l\_type, l\_whence, l\_start, l\_len의 값들도 모두 이전 락을 건 프로세스의 정보로 변경되게 됩니다.

이번에는 fcntl()을 이용해서 Lock을 거는 예를 보입니다.

내용 자체는 위의 예제와 동일합니다.

다음과 같은 소스로 Pa.c를 만듭니다.

|  |
| --- |
| #include <stdio.h>  #include <fcntl.h>  #include <unistd.h>  int main(int argc, char\* argv[])  {  int fd;  FILE \*fp;  struct flock filelock;    fp=fopen("locktest", "r+b");  filelock.l\_type=F\_WRLCK;  filelock.l\_whence=SEEK\_SET;  filelock.l\_start=0;  filelock.l\_len=10;  fd=fileno(fp);  fcntl(fd, F\_SETLKW, &filelock);  printf("Locked By Pa\n");  sleep(5);  fclose(fp);  printf("Released By Pa\n");    return 0;  } |

다음과 같은 소스로 Pb.c를 만듭니다.

|  |
| --- |
| #include <stdio.h>  #include <fcntl.h>  #include <unistd.h>  int main(int argc, char\* argv[])  {  int fd;  FILE \*fp;  struct flock filelock;  fp=fopen("locktest", "r+b");  filelock.l\_type=F\_WRLCK;  filelock.l\_whence=SEEK\_SET;  filelock.l\_start=0;  filelock.l\_len=10;  fd=fileno(fp);  fcntl(fd, F\_SETLKW, &filelock);  printf("Locked By Pb\n");  sleep(5);  fclose(fp);  printf("Released By Pb\n");  return 0;  } |

그리고 locktest 라는 파일을 현재 디렉토리에 만들어 놓습니다.

명령으로 다음과 같이 입력합니다.

cc -o Pa Pa.c  
cc -o Pb Pb.c  
Pa&  
Pb&

결과도 위의 lockf()함수를 이용할 때와 똑같다는 것을 아실 수 있을 것입니다. 좀 다른 점이 있다면 open(), close()로 파일을 열지 않고 fopen(), fclose()를 사용하였으면 파일포인터를 파일기술자로 변환하는 fileno()함수를 사용하였다는 것입니다. 왜 그렇게 했냐구요? 그냥 하다보니 그렇게 됩습니다. 이유는 없어요. ^^;

**2. Windows에서의 File Lock**

Windows에서는 파일을 Lock하기 위해서 LockFile(), UnlockFile(), CreateFile() 함수를 사용할 수 있습니다.

다음과 같은 원형을 갖고 있습니다.

HANDLE CreateFile(  
  LPCTSTR lpFileName,                                       // pointer to name of the file  
  DWORD dwDesiredAccess,                                // access (read-write) mode  
  DWORD dwShareMode,                                     // share mode  
  LPSECURITY\_ATTRIBUTES lpSecurityAttributes,  // pointer to security attributes  
  DWORD dwCreationDisposition,                          // how to create  
  DWORD dwFlagsAndAttributes,                           // file attributes  
  HANDLE hTemplateFile                                     // handle to file with attributes to copy  
);

BOOL LockFile(  
  HANDLE hFile,                                                  // handle of file to lock  
  DWORD dwFileOffsetLow,                                   // low-order word of lock region offset  
  DWORD dwFileOffsetHigh,                                   // high-order word of lock region offset  
  DWORD nNumberOfBytesToLockLow,                  // low-order word of length to lock  
  DWORD nNumberOfBytesToLockHigh                   // high-order word of length to lock  
);

BOOL UnlockFile(  
  HANDLE hFile,                                                   // handle of file to unlock  
  DWORD dwFileOffsetLow,                                    // low-order word of lock region offset  
  DWORD dwFileOffsetHigh,                                    // high-order word of lock region offset  
  DWORD nNumberOfBytesToUnlockLow,                // low-order word of length to unlock  
  DWORD nNumberOfBytesToUnlockHigh                 // high-order word of length to unlock  
);

입력할 수 있는 아큐먼트가 꽤 많네요. 이 중 CreateFile()함수는 사용하는 용도가 아주 다양한데...

여기서는 파일락의 관점에서만 이야기 하도록 하겠습니다.

**1) HANDLE CreateFile(LPCTSTR lpFileName, DWORD dwDesiredAccess, DWORD dwShareMode, LPSECURITY\_ATTRIBUTES lpSecurityAttributes, DWORD dwCreationDisposition, DWORD dwFlagsAndAttributes, HANDLE hTemplateFile);**

CreateFile()함수는 함수명은 파일만들기이지만 파일을 만드는 것 뿐만 아니라 파일을 오픈하는데도 사용됩니다.

lpFileName은 오픈하고자 하는 파일의 경로입니다.

dwDesiredAccess는 파이을 어떤 형태로 열 것인지를 결정합니다. 사용할 수 있는 값으로는 0, GENERIC\_READ, GENERIC\_WRITE가 있습니다. 즉, 파일을 오픈할 때 읽고 쓰기 모드는 정하는 것입니다.

dwShareMode는 현재 CreateFile()함수에 의해 파일이 열렸을 경우 다른 프로세스나 쓰레드가 오픈된 파일에 접근할 수 있게 하는 아큐먼트입니다. 즉, 이 아큐먼트가 파일락에 관련된 것입니다.

다른 쓰레드나 프로세스가 현재 파일을 삭제하거나 읽거나 쓸수 있게 하려면 다음과 같은 값을 OR로 연결하여 주어야 합니다. FILE\_SHARE\_DELETE, FILE\_SHARE\_READ, FILE\_SHARE\_WRITE.

만약 파일을 공유하기 싫다면 0을 주면 됩니다.

lpSecurityAttributes는 NTFS의 보안관련된 아큐먼트인데... 이게 좀 복잡하므로 현재는 NULL만을 입력하여 사용하도록 합니다.

dwCreationDisposition은 예외 사항시 어떻게 처리할 것인지를 결정합니다.

- CREATE\_NEW: 새로운 파일을 만듭니다. 만약 파일이 존재한다면 에러를 리턴합니다.

- CREATE\_ALWAYS: 파일이 존재하더라도 새로운 파일을 만듭니다.(존재하는 파일의 내용은 없어집니다.)

- OPEN\_EXISTING: 이미 존재하는 파일을 오픈합니다. 파일이 없다면 에러를 리턴합니다.

- OPEN\_ALWAYS: 파일을 엽니다. 만약 파일이 없다면 파일을 새롭게 만듭니다.

- TRUNCATE\_EXISTING: 파일을 엽니다. 단 열린 파일을 0바이트로 만듭니다.

dwFlagsAndAttributes은 파일 속성과 플래그를 지정하는 아큐먼트입니다.

이 아큐먼트에 사용할 수 있는 값들은 꽤 많습니다. 이는 다른 문서를 참조하시구요. 현재는 그냥 FILE\_ATTRIBUTE\_NORMAL을 지정하도록 합니다.

hTemplateFile은 템플릿 파일의 핸들을 지정합니다. 사용하지 않을 경우 NULL을 입력합니다.

CreateFile()은 파일을 정상적으로 열면 파일의 핸들을 리턴합니다. 단, 파일을 여는 동안 오류가 발생하였다면 INVALID\_HANDLE\_VALUE라는 값을 리턴합니다. 이렇게 오류가 난 경우는 GetLastError()를 통해 에러의 원인을 알 수 있습니다.

정상적으로 열린 파일핸들은 CloseHandle()함수에 의해 닫아져야 합니다.

CreateFile()에 의한 파일락은 UNIX에서 lockf()함수 처럼 파일 전체에 락을 거는 것입니다.

다음과 같은 소스로 Pa.c를 만듭니다.

|  |
| --- |
| #include <windows.h>  #include <stdio.h>  int main(int argc, char \*argv[])  {  HANDLE hFile;  BOOL bLock;  loop:  hFile=CreateFile("C:\\locktest",  GENERIC\_READ|GENERIC\_WRITE,  0,  NULL,  CREATE\_ALWAYS,  FILE\_ATTRIBUTE\_NORMAL,  NULL);  if(hFile==INVALID\_HANDLE\_VALUE){  Sleep(10);  goto loop;  }else{  printf("Locked By Pa\n");  Sleep(5000);  CloseHandle(hFile);  printf("Released By Pa\n");  }  return 0;  } |

다음과 같은 소스로 Pb.c를 만듭니다.

|  |
| --- |
| #include <windows.h>  #include <stdio.h>  int main(int argc, char \*argv[])  {  HANDLE hFile;  BOOL bLock;  loop:  hFile=CreateFile("C:\\locktest",  GENERIC\_READ|GENERIC\_WRITE,  0,  NULL,  CREATE\_ALWAYS,  FILE\_ATTRIBUTE\_NORMAL,  NULL);  if(hFile==INVALID\_HANDLE\_VALUE){  Sleep(10);  goto loop;  }else{  printf("Locked By Pb\n");  Sleep(5000);  CloseHandle(hFile);  printf("Released By Pb\n");  }  return 0;  } |

그리고 locktest 라는 파일을 C:\ 디렉토리에 만들어 놓습니다.

해보시면 아시겠지만 실행 결과는 유닉스에서 lockf()을 실행한 것과 같습니다.

주요하게 보셔야 하는 부분은 CreateFile()에서 세번째 아큐먼트를 0으로 주었다는 것 입니다.

이렇게 하면 현재 오픈한 파일을 다른 쓰레드나 프로세스가 사용하지 못하게 하는 것입니다.

이 부분을 수정하면 읽으려고 하는 경우만 공유를 허용하게도 변경할 수 있습니다.

**2) BOOL LockFile(HANDLE hFile, DWORD dwFileOffsetLow, DWORD dwFileOffsetHigh, DWORD nNumberOfBytesToLockLow, DWORD nNumberOfBytesToLockHigh);**

위의 CreateFile()은 파일전체에 대한 락을 생성합니다. 엄밀하게 말하면 락이라기 보다는 공유를 하지 않는 것이죠? 어찌되었건 락의 효과는 얻을 수 있습니다. 이번에 이야기 할 LockFile()함수는 유닉스의 fcntl()과 같이 파일의 일부분만을 락킹하는 함수입니다.

아큐먼트는 다음과 같습니다.

hFile은 CreateFile()로 오픈한 파일의 핸들을 입력합니다.

dwFileOffsetLow와 dwFileOffsetHigh는 락을 걸 위치의 시작위치를 지정합니다.

nNumberOfBytesToLockLow와 nNumberOfBytesToLockHigh는 락을 걸 위치의 끝위치를 락을 걸 부분의 크기로 지정합니다.

보면 위치를 지정하는 아큐먼트의 변수명이 Low와 High의 쌍이라는 걸 아실 수 있을 겁니다.

Low는 High로 부터 Offset을 이야기 합니다. High는 대부분 0입니다. Low만으로 2Giga를 표현할 수 있습니다. 그래서 2Giga보다 작을 경우 High부분은 0인거죠.

LockFile()함수는 성공시에 0이외의 값을 리턴합니다. 만약 실패로 인해 0을 리턴할 경우 GetLastError()로 실패의 원인을 알 수 있습니다.

LockFile()에서 락을 걸 영역의 크기는 파일에 현재 존재하는 영역 밖을 지정할 수 있습니다.

즉, 파일의 현재 크기는 10바이트인데 50바이트부터 1000바이트까지를 락시킬 수 있다는 이야기입니다.

**3) BOOL UnlockFile(HANDLE hFile, DWORD dwFileOffsetLow, DWORD dwFileOffsetHigh, DWORD nNumberOfBytesToUnlockLow, DWORD nNumberOfBytesToUnlockHigh);**

 UnlockFile()은 LockFile()과 사용법이 같습니다. 단, 락을 거는게 아니라 락을 푸는것일뿐...

다음은 LockFile()함수를 이용한 샘플입니다.

다음과 같은 소스로 Pa.c를 만듭니다.

|  |
| --- |
| #include <windows.h>  #include <stdio.h>  int main(int argc, char \*argv[])  {  HANDLE hFile;  BOOL bLock;  loop:  hFile=CreateFile("C:\\locktest",  GENERIC\_READ|GENERIC\_WRITE,  FILE\_SHARE\_READ|FILE\_SHARE\_WRITE,  NULL,  CREATE\_ALWAYS,  FILE\_ATTRIBUTE\_NORMAL,  NULL);  if(hFile==INVALID\_HANDLE\_VALUE){  Sleep(10);  goto loop;  }else{  loop2:  bLock=LockFile(hFile, 0, 0, 0xFFFFFFFF, 0xFFFFFFFF);  if(bLock==0) {  Sleep(10);  goto loop2;  }  printf("Locked By Pa\n");  Sleep(5000);  UnlockFile(hFile, 0, 0, 0xFFFFFFFF, 0xFFFFFFFF);  CloseHandle(hFile);  printf("Released By Pa\n");  }  return 0;  } |

다음과 같은 소스로 Pb.c를 만듭니다.

|  |
| --- |
| #include <windows.h>  #include <stdio.h>  int main(int argc, char \*argv[])  {  HANDLE hFile;  BOOL bLock;  loop:  hFile=CreateFile("C:\\locktest",  GENERIC\_READ|GENERIC\_WRITE,  FILE\_SHARE\_READ|FILE\_SHARE\_WRITE,  NULL,  CREATE\_ALWAYS,  FILE\_ATTRIBUTE\_NORMAL,  NULL);  if(hFile==INVALID\_HANDLE\_VALUE){  Sleep(10);  goto loop;  }else{  loop2:  bLock=LockFile(hFile, 0, 0, 0xFFFFFFFF, 0xFFFFFFFF);  if(bLock==0) {  Sleep(10);  goto loop2;  }  printf("Locked By Pb\n");  Sleep(5000);  UnlockFile(hFile, 0, 0, 0xFFFFFFFF, 0xFFFFFFFF);  CloseHandle(hFile);  printf("Released By Pb\n");  }  return 0;  } |

 뭐 위의 소스가 다 그렇듯 위의 소스를 실행하면 지금까지의 다른 소스들과 같은 결과를 보실 수 있으실 겁니다.

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

Windows에서는 LockFile()함수와 함께 LockFileEx()와 UnlockFileEx()함수를 제공합니다. 차이는 LockFile()은 배타적락(Exclusive Lock) 만이 걸린다는 것이구요. LockFileEx()는 공유락(Shared Lock)도 걸 수 있다는 것입니다. 또 다른점은 오버랩된 입출력을 지원하고 블록 시킬 수 있다는 점이 다릅니다.

아큐먼트는 LockFile()과 비교해서 다른 점만 이야기 드리겠습니다.

dwFlags를 0으로 설정하면 공유락이 걸립니다. 배타적락을 걸고 싶으면 LOCKFILE\_EXCLUSIVE\_LOCK을 주어야 하고요 이미 락이 걸려있는 경우 블록 시키지 않으려면 LOCKFILE\_FAIL\_IMMEDIATELY을 주어야 합니다. 이 둘을 OR로 묶을 수도 있습니다.

dwReserved는 예약된 아큐먼트로 0을 주면 됩니다.

nNumberOfBytesToLockLow, nNumberOfBytesToLockHigh은 LockFile()의 것과 같습니다. 즉, 락을 걸 끝 위치의 크기를 나타내는데... 그럼 시작은 어디서 지정하느냐?

그건 제일 마지막 아큐먼트인 lpOverlapped에 지정합니다.

OVERLAPPED 구조체의 Offset과 OffsetHigh 변수를 이용하여 설정하면 됩니다.

나중에 예제를 참조해 보세요.

**5) BOOL UnlockFileEx(HANDLE hFile, DWORD dwReserved, DWORD nNumberOfBytesToUnlockLow, DWORD nNumberOfBytesToUnlockHigh, LPOVERLAPPED lpOverlapped);**

UnlockFileEx()는 LockFileEx()함수에 의해 걸린 락을 해제합니다.

아큐먼트의 사용법은 LockFileEx()의 것과 비교하면 아실거구요. dwReserved 만 예약된 값이므로 0을 준다는 것만 기억하시면 될 겁니다.

아래의 예를 참조하세요.

|  |
| --- |
| #include <windows.h>  #include <stdio.h>  int main(int argc, char\* argv[])  {  HANDLE hFile;  OVERLAPPED overlapped;  BOOL bRet;  overlapped.Offset=0;  overlapped.OffsetHigh=0;  hFile=CreateFile("C:\\locktest", GENERIC\_READ|GENERIC\_WRITE,  FILE\_SHARE\_READ|FILE\_SHARE\_WRITE, NULL, OPEN\_ALWAYS, 0, NULL);    if(hFile!=INVALID\_HANDLE\_VALUE){  printf("File is Opened\n");  bRet=LockFileEx(hFile, LOCKFILE\_EXCLUSIVE\_LOCK, 0, 0xFFFFFFFF, 0, &overlapped);  printf("Locked File[%d]\n", bRet);  Sleep(3000);  bRet=UnlockFileEx(hFile, 0, 0xFFFFFFFF, 0, &overlapped);  printf("Release File[%d]\n", bRet);  CloseHandle(hFile);  printf("Closed File\n");  }  return 0;  } |

위의 예제는 블록되는 LockFileEx()의 사용 예입니다.

뭐 이제 보시면 아시겠죠?

그럼 파일락은 여기까지.... ^^: