Lecture 24 — Concurrency in File Systems

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

August 29, 2021

Concurrency in File Systems

We need to peel back the interface a bit and have at least a high-level understanding of their implementation.

File can be of arbitrary size (although in a particular file system there may be a limit).

They have to be allocated on disk according to some strategy.

Contiguous Allocation

Contiguous: a file occupies a set of contiguous blocks on disk.

So a file is allocated, starting at block b and is n blocks in size, the file takes up blocks $b, b+1, b+2, \ldots, b+(n-1)$.

This is advantageous, because if we want to access block b on disk, accessing $b+1$ requires no head movement, so seek time is nonexistent to minimal.

Contiguous Allocation

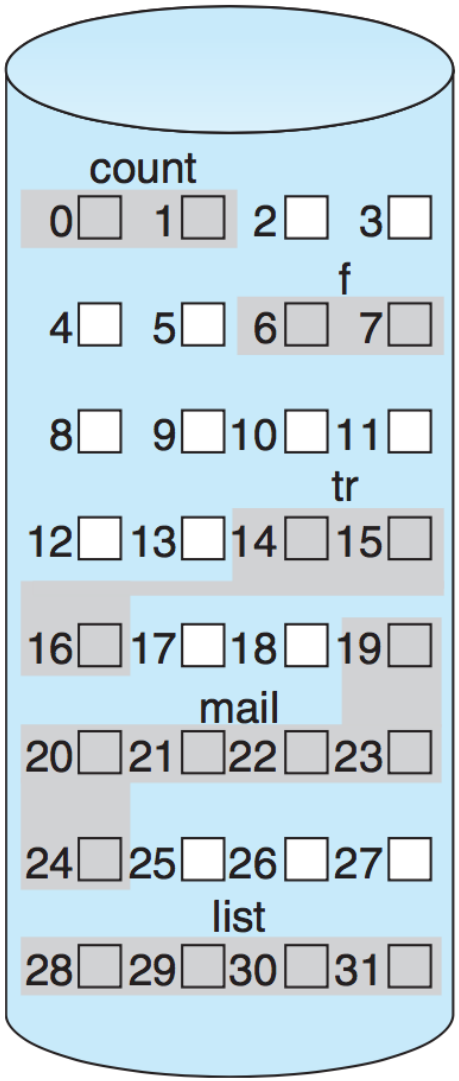
All that we need to maintain are b and n: the start location and length of the file.

Both sequential and direct access are very easy: the first block of a file is at b.

To access a block i at some offset into the file, it's at the base address b plus i.

Checking if the access is valid is also an easy operation: if $i < n$ then it is valid.

Contiguous Allocation



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Contiguous Allocation

If we need a memory block of size *N*:

(1) can we find a contiguous block of *N* or greater to meet that allocation?

(2) if there is more than one block, which one do we choose?

As before, we suffer the problem of external fragmentation, plus a bit of internal fragmentation in the last block of the file.

Contiguous Allocation

Another problem: how much space is a file going to take?

If it is just a copy-paste operation, the copy is the same size as the original.

When a user opens a new document, how big will it be?

If we allocate too little space, we may be able to tack on space at the end, or that block may be allocated, forcing us to move the file and reallocate it.

If the value we choose is too large, then significant space will be wasted for small files (and many files tend to be relatively small).

Linked Allocation

Linked allocation is a solution to the problems of contiguous allocation.

Instead of a file being all in consecutive blocks, we maintain a linked list of the blocks, and the blocks themselves may be located anywhere on the disk.

The directory listing just has a pointer to the first and last blocks (head and tail of the linked list).

Linked Allocation

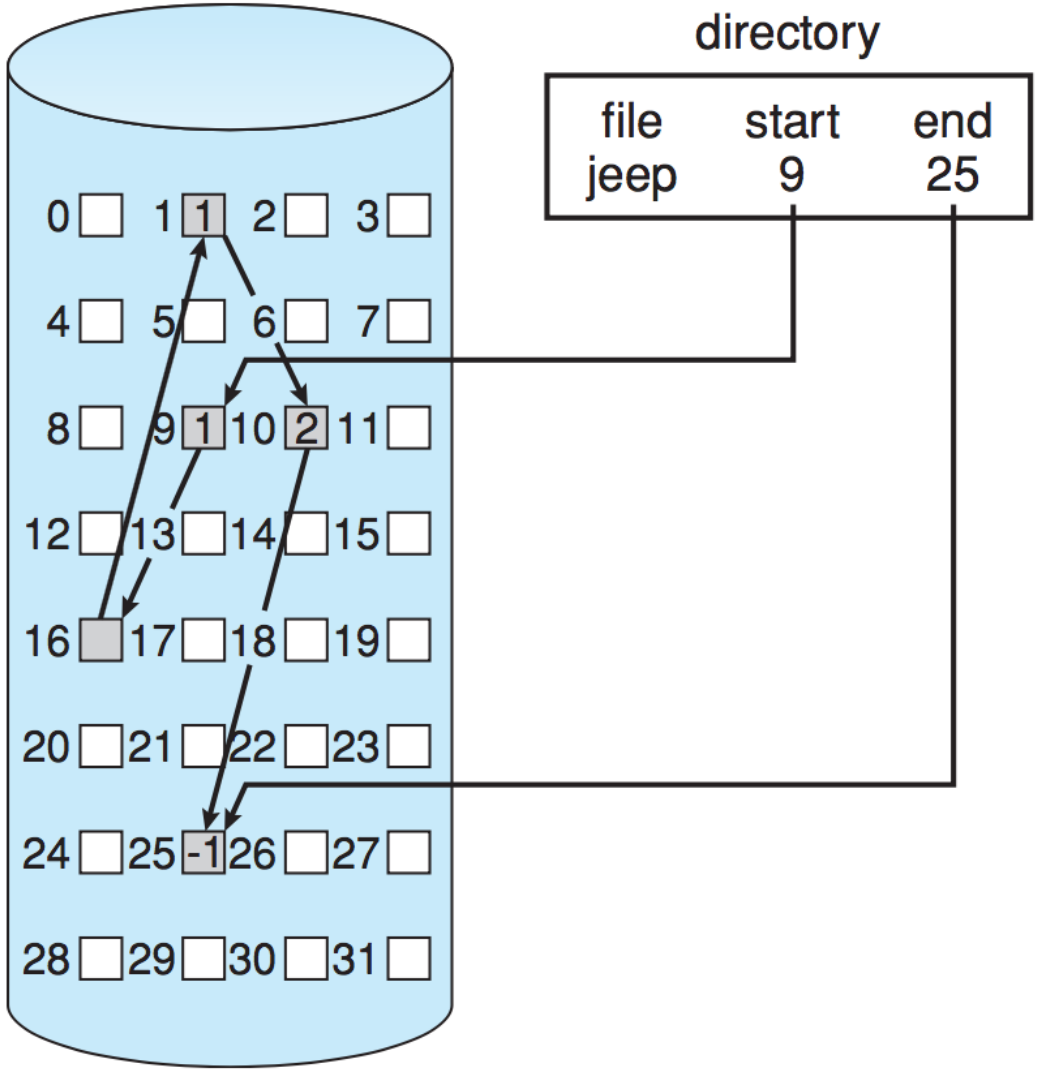
If a new file is created, it will be created with size zero and the head and tail pointers are null.

When a new block is needed, it can come from anywhere and will just be added to the linked list.

Thus, compaction and relocation are not really an issue.

Unfortunately, however, accessing block *i* of a file is no longer as simple as computing an offset from the first block; it requires following *i* pointers (a pain).

Linked Allocation



Linked Allocation

A possible solution to the problem of following so many pointers (and the overhead of maintaining so many) is to group up the blocks into clusters.

A cluster is comprised of, say, four blocks.

Then we waste less memory maintaining pointers and it improves disk accesses because there is less seeking back and forth to various disk locations.

Indexed Allocation

If we stuck to pure linked allocation, we still have the problem that accessing some part in the middle of the file is a pain.

We have to follow and retrieve a lot of pointers to the different blocks.

The idea of indexed allocation is to take all the pointers and put them into one location: an index block.

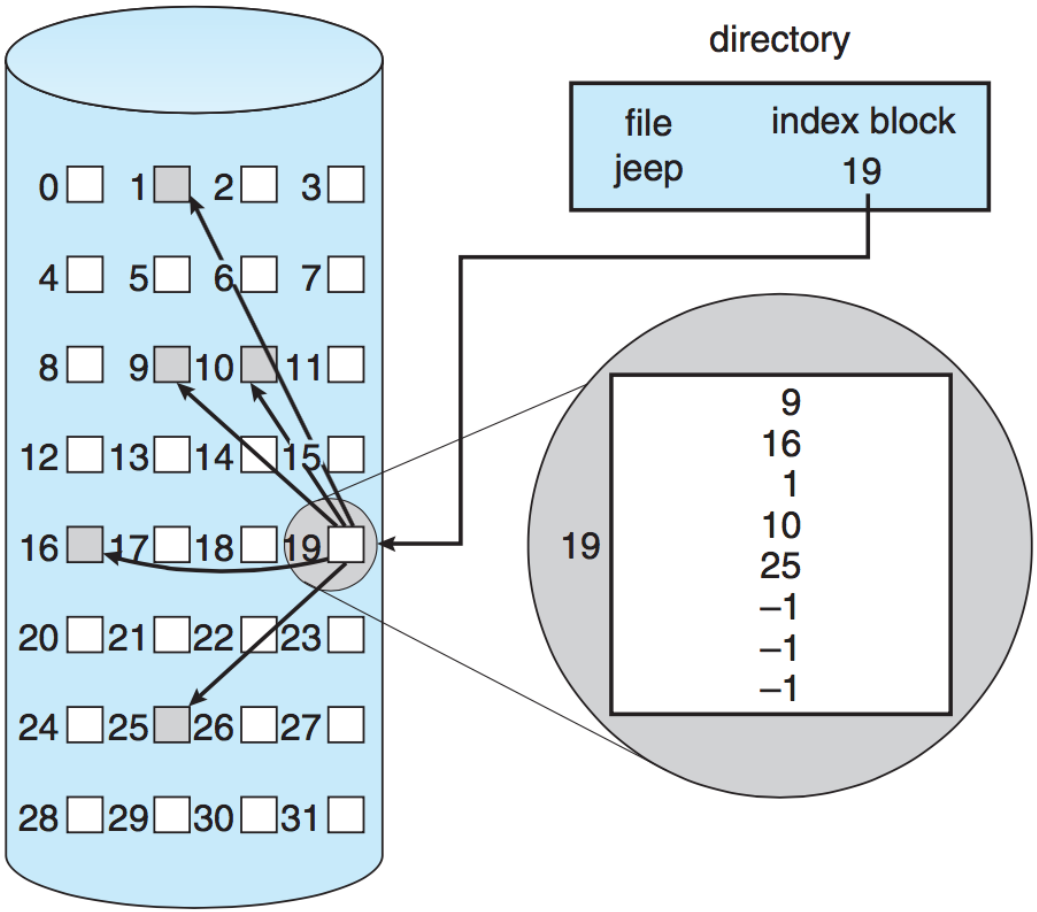
Indexed Allocation

So, the first block of the file contains a whole bunch of pointers.

To get to block *i*, just go to index *i* of the index block and we can get the location of block *i* much more efficiently than we could in linked allocation.

All pointers to blocks start as null, and when we add a new block, add its corresponding entry into the index block.

Indexed Allocation



Block Size

Like many of the other systems we have examined, there is a need to make a decision about the size of a block.

If a file needs only 1-2 blocks, one whole block is allocated for the pointers which contains only 1-2 entries.

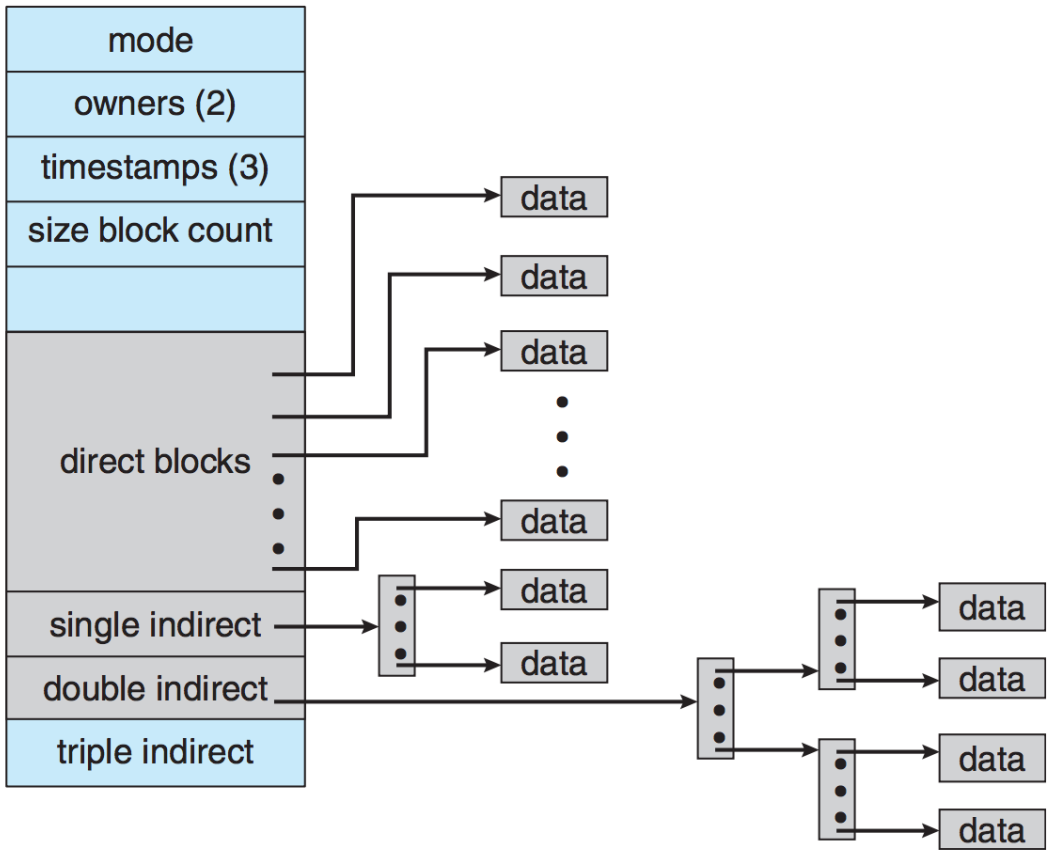
That suggests we want the index to be small, but what if we need more pointers than fit into one block? There are a few mechanisms for this.

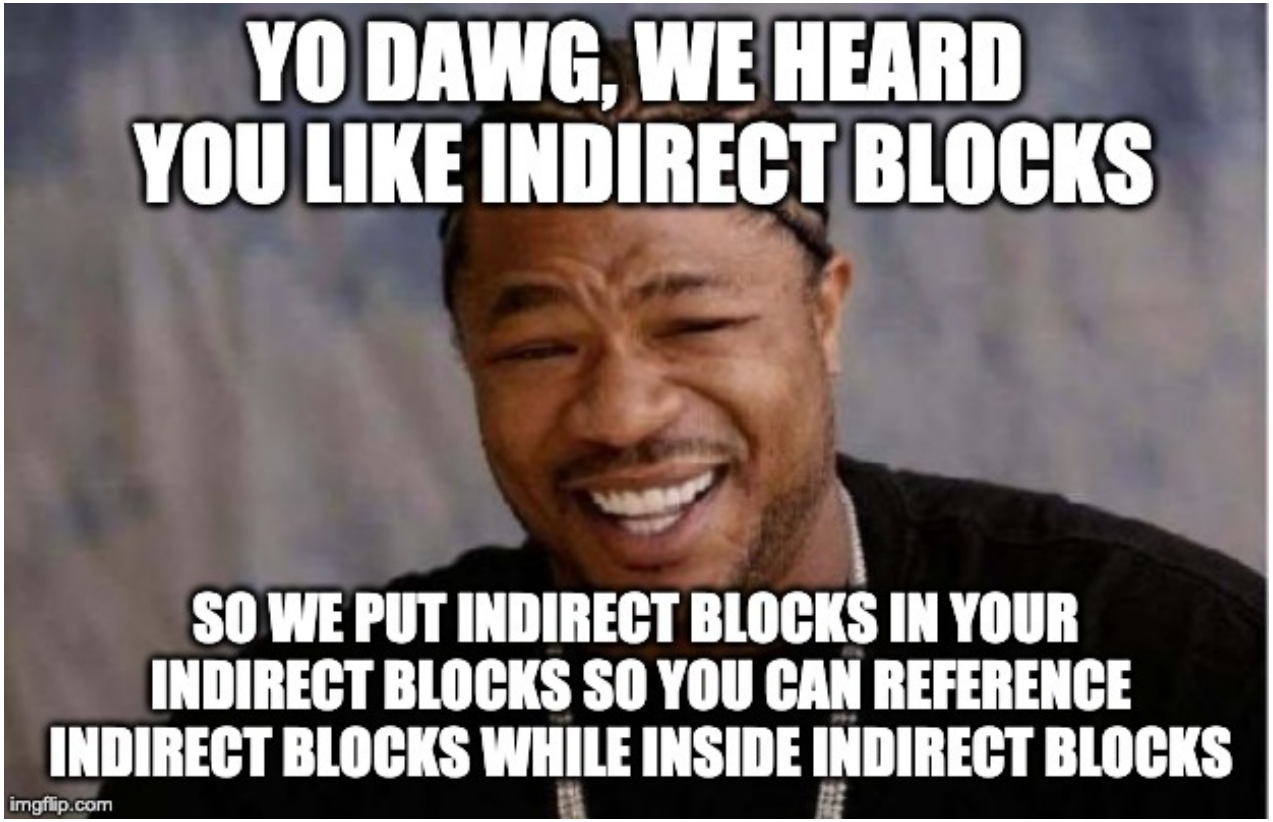
Block Size

What if we need more pointers than fit into one block?

1. **Linked Scheme**
2. **Multilevel Index**
3. **Combined Scheme**

A visual representation of an inode.




YO DAWG, WE HEARD
YOU LIKE INDIRECT BLOCKS
SO WE PUT INDIRECT BLOCKS IN YOUR
INDIRECT BLOCKS SO YOU CAN REFERENCE
INDIRECT BLOCKS WHILE INSIDE INDIRECT BLOCKS
imgflip.com

Partial Locking

Previously: `flock()` to lock a file, and this locks the entire file.

Using `fcntl`, we can lock only a part of a file.

This is referred to as record locking.

Partial Locking

Locking just a part of the file allows for more concurrency!

```
int fcntl( int file_descriptor, int command, ... /* struct flock * flockptr */ )
```

We need to provide one `struct flock` and a `command`.

Struct Flock

The `struct flock` has the following definition:

```
struct flock {
  short  l_type;   /* F_RDLCK, F_WRLCK, or F_UNLCK */
  short  l_whence; /* SEEK_SET, SEEK_CUR, or SEEK_END */
  off_t  l_start;  /* offset in bytes, relative to l_whence */
  off_t  l_len;    /* length, in bytes; 0 means lock to EOF */
  pid_t  l_pid;    /* returned with F_GETLK */
};
```

About `l_type`: the types of lock are read and write.

Compatibility matrix: reads are compatible with reads; writes with nothing.

To unlock, use `F_UNLCK`.

This is vulnerable to deadlock...

Struct Flock

The `struct flock` has the following definition:

```
struct flock {
  short  l_type;   /* F_RDLCK, F_WRLCK, or F_UNLCK */
  short  l_whence; /* SEEK_SET, SEEK_CUR, or SEEK_END */
  off_t  l_start;  /* offset in bytes, relative to l_whence */
  off_t  l_len;    /* length, in bytes; 0 means lock to EOF */
  pid_t  l_pid;    /* returned with F_GETLK */
};
```

`l_whence`: where does the offset begin?

It is possible for a locked region to extend past the end of the file.

Das war ein Befehl!

For `command`, our choices are [**?**]:

- `F_GETLK`
- `F_SETLK`
- `F_SETLKW`

Unlocking and Combining Locks

When unlocking a region, just as for locking, you can specify what part of the file you would like to unlock.

Partial unlocking is unusual, but why not?

The system will combine or split locks as appropriate,

Lock Usage Example

```
int write_lock_file( int file_descriptor ) {

  struct flock fl;
  fl.l_type = F_WRLOCK;
  fl.l_start = 0;
  fl.l_whence = SEEK_SET;
  fl.l_len = 0;

  return fcntl( fd, F_SETLK, &fl );
}

int unlock_file( int file_descriptor ) {

  struct flock fl;
  fl.l_type = F_UNLCK;
  fl.l_start = 0;
  fl.l_whence = SEEK_SET;
  fl.l_len = 0;

  return fcntl( fd, F_SETLK, &fl );
}
```

Checking if a given part of a file is locked:

```
int fd = open ( "example.txt", O_RDONLY );
struct flock lock;

lock.l_type = F_RDLOCK;
lock.l_start = 1024;
lock.l_whence = SEEK_SET;
lock.l_len = 256;

fcntl( fd, F_GETLK, &lock );
if ( lock.l_type == F_UNLCK ) {
        /* Lock is unlocked; we may proceed */
} else if ( lock.l_type = F_WRLOCK ) {
  /* File is write locked by a different process */
  printf( "File locked by process ID %d.\n", lock.l_pid );
  return -1;
}
```

Orders, Captain?

Checking on things with F_GETLK is really for information purposes only.

“Read the value and then whatever operation you’d like to do next” is not atomic.

Instead, use the command F_SETLK and actually try to set the lock.

If -1 is returned then locking was not successful.

Or, if the plan is to wait, use F_SETLKW as one would expect.

Immutability is for Cowards

`fcntl` changes some values of the `struct lock`!

If you wanted to re-use it you need to make sure to reset it as appropriate.

You can use the same `struct lock` later to unlock the thing that you locked, just do so carefully.

SPLITTERS!

Sometimes the name you want is taken…

`lockf`: a simplified way of locking a file.

While `fcntl` is more flexible, sometimes all we need is the simple version.

Locking, Simplified

```
int lockf( int file_descriptor, int command, off_t length );
```

The command options can be:

- F_LOCK
- F_TLOCK
- F_ULOCK
- F_TEST

The length is an offset, and is based off the current position in the file.

If zero is provided then it locks the whole file.

Close up for the Day

The file is automatically unlocked when the file descriptor is closed.

And, on some systems `lockf` just calls `fcntl` but on some others they use different mechanisms.

So don’t mix and match.

If you lock a file with one function, unlock it with the matching one.

There are Rules

It is noteworthy that both kinds of lock are “advisory” only.

It only is really effective if everyone involved in accessing the shared resource follows the proper protocol and checks if access is permitted or not.

Mandatory locks do exist, but are hard to use and are not recommended.

The notes link to why you shouldn’t!

I am become Lock

We can use the very existence of a file as a way of controlling concurrency.

For example, `git` places a file `index.lock` in a particular directory to indicate that an operation is in progress.

Thus two different `git` clients do not operate on the same repository at the same time.

Open Sesame?

If we want to check, we just try to `open()` the file, but unless we are careful this can lead to a problem if two processes want to create the file.

If they both call `open`, they both might succeed. To get around this, we need to use the `flags` parameter

File Operations

```
int open(const char *filename, int flags);
int rename(const char *old_filename, const char *new_filename);
int remove(const char *filename);
```

When opening a file the following flags may be used for the `flags` parameter (and can be combined with bitwise OR, the | operator):

Value	Meaning
O_RDONLY	Open the file read-only
O_WRONLY	Open the file write-only
O_RDWR	Open the file for both reading and writing
O_APPEND	Append information to the end of the file
O_TRUNC	Initially clear all data from the file
O_CREAT	Create the file
O_EXCL	If used with O_CREAT, the caller MUST create the file; if the file exists it will fail

Lock-File Behaviour

Team-up `open` and `rename` to get lock-like behaviour between different programs that share nothing except a common file system.

The `open` call should be used to create the lock file, and fail if the file already exists.

If we want we can use `remove` to delete the lock file if we want to let the next process try, but there's an alternative option: `rename`.

The `rename` function is also atomic!

To lock: change the name; to unlock, change it back!

Using a File as a Lock

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <pthread.h>
#define NUM_THREADS 10

int lock_fd;
int shared = 0;

void* run( void* arg ) {
  int* id = (int*) arg;
  while( rename( "file.lock", "file.locked" ) == -1 ) {
    printf("Thread %d waiting.\n", *id);
  }
  printf("Thread %d in critical section.\n", *id);
  printf("Shared incremented from %d", shared);
  shared++;
  printf(" to %d.\n", shared);
  rename("file.locked", "file.lock"); /* Unlock */

  free( arg );
  pthread_exit(NULL);
}
```

Using a File as a Lock

```
void* writer( void* arg ) {
  /* Write data implementation not shown */
  pthread_exit(NULL);
}

int main( int argc, char** argv ) {
  lock_fd = open( "file.lock", O_CREAT | O_EXCL );
  if (lock_fd == -1 ) {
   printf( "File creation failed.\n");
   return -1;
  }

  pthread_t threads[NUM_THREADS];
  for (int i = 0; i < NUM_THREADS; i++) {
    int * id = malloc( sizeof( int ) );
    *id = i;
    pthread_create( &threads[i], NULL, run, id );
  }
  for (int i = 0; i < NUM_THREADS; i++) {
    pthread_join( threads[i], NULL );
  }
  close( lock_fd );
  remove( "file.lock" );

  return 0;
}
```