Perforamce comparison between two I/Os and two Locks in Linux

Penny Wu(pwu56)

Abstract: This paper presents the performacnce difference between direct I/O and buffer I/O, between spin_lock and mutex in user-level applications that interact with the OS intensely. *For direct I/O and buffer I/O*, the conclusion is that using direct I/O typically results in the process taking longer to complete, especially for relatively small reads/writes. But in synchronous write application, direct I/O perform better. *For spin_lock and mutex*, a spinlock can be better when you plan to hold the lock for an extremely short interval and contention is expected to be rare. And with the increasing of threads number, the performance will decline for spin-lock.

Introduction: *Direct I/O* is a feature of the file system whereby file reads and writes go directly from the applications to the storage device, bypassing the operating system read and write caches. Direct I/O is used only by applications (such as databases) that manage their own caches. With direct I/O, I/O is performed directly to/from user-space buffers without being cached in the file system. An application invokes direct I/O by opening a file with the O_DIRECT flag. Alternatively, GFS can attach a direct I/O attribute to a file, in which case direct I/O is used regardless of how the file is opened.

Mutex provides one person to access a single resource at a time, others must wait in a queue. Once this person is done, the guy next in the queue acquire the resource. So access is serial, one guy after other. Too aggressive.Semaphore are useful if multiple instances (N) of a resource is to be shared among a set of users. As soon as all N resources are acquired, any new requester has to wait. Since there is no single lock to hold, there is as such no ownership of a semaphore. *Spinlock* is an aggressive mutex. In mutex, if you find that resource is locked by someone else, you (the thread/process) switch the context and start to wait (non-blocking).
Whereas spinlocks do not switch context and keep spinning. As soon as resource is free, they go and grab it. In this process of spinning, they consume many CPU cycles. Also, on a uni-processor machine they are useless and perform very badly.

Experiments:

- **Experimental platform:**

  # cores/processors: 5

  amount of physical memory: 32G

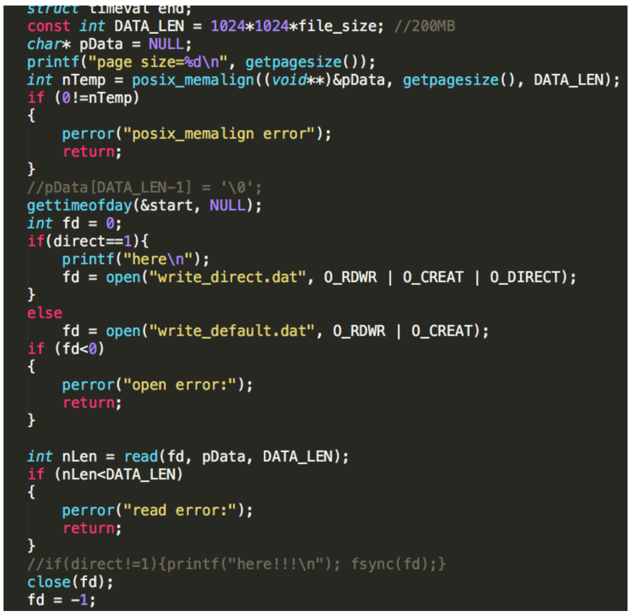
  OS version: Linux version 4.4.0-142-generic

  file system: ext3, ext4, nfs, tmpfs

- **Expriment 1: Local File system: Direct I/O vs. Default**

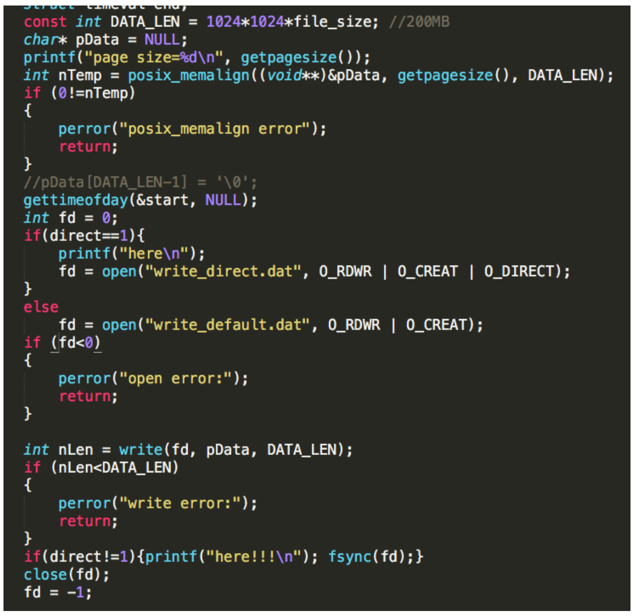
  The difference between Direct I/O and buffer I/O is that Direct I/O file reads and writes go directly from the applications to the storage device, bypassing the operating system read and write caches. In a word, Direct I/O is synchronous writes/reads, so I design application A only with writes operations, application B only with reads operations, application C with writes and reads operations, and application D with synchronnous writes operations using fsync() function after writes.

Reads: Synchronous Writes(delete fsync() to normal writes):



```
struct timeval end;
const int DATA_LEN = 1024*1024*file_size; //200MB
char* pData = NULL;
printf("page size=%d\n", getpagesize());
int nTemp = posix_memalign((void**)&pData, getpagesize(), DATA_LEN);
if (0!=nTemp)
{
    perror("posix_memalign error");
    return;
}
//pData[DATA_LEN-1] = '\0';
gettimeofday(&start, NULL);
int fd = 0;
if(direct==1){
    printf("here\n");
    fd = open("write_direct.dat", O_RDWR | O_CREAT | O_DIRECT);
}
else
    fd = open("write_default.dat", O_RDWR | O_CREAT);
if (fd<0)
{
    perror("open error:");
    return;
}

int nLen = read(fd, pData, DATA_LEN);
if (nLen<DATA_LEN)
{
    perror("read error:");
    return;
}
//if(direct!=1){printf("here!!!\n"); fsync(fd);}
close(fd);
fd = -1;
```



```
struct timeval end;
const int DATA_LEN = 1024*1024*file_size; //200MB
char* pData = NULL;
printf("page size=%d\n", getpagesize());
int nTemp = posix_memalign((void**)&pData, getpagesize(), DATA_LEN);
if (0!=nTemp)
{
    perror("posix_memalign error");
    return;
}
//pData[DATA_LEN-1] = '\0';
gettimeofday(&start, NULL);
int fd = 0;
if(direct==1){
    printf("here\n");
    fd = open("write_direct.dat", O_RDWR | O_CREAT | O_DIRECT);
}
else
    fd = open("write_default.dat", O_RDWR | O_CREAT);
if (fd<0)
{
    perror("open error:");
    return;
}

int nLen = write(fd, pData, DATA_LEN);
if (nLen<DATA_LEN)
{
    perror("write error:");
    return;
}
if(direct!=1){printf("here!!!\n"); fsync(fd);}
close(fd);
fd = -1;
```

application_A(writes): Buffer I/O win.

Every direct I/O write causes a synchronous write to disk; unlike the normal cached I/O policy where the data is merely copied and then written to disk later. This fundamental difference can cause a significant performance penalty for applications that are converted to use direct I/O.

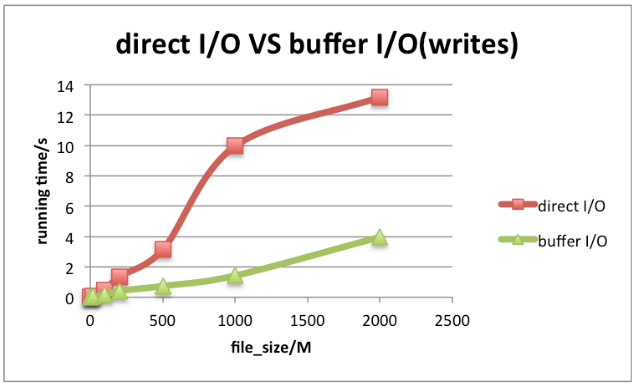


Figure 1. direct I/O VS buffer I/O(writes). When I only compare the writes operation between directI/O and buffer I/O. I found with the increase of file size, the performance decrease more aggressively for direct I/O than buffer I/O. And buffer I/O always performance better than direct I/O.

application_B(reads): Buffer I/O win.

Every direct I/O read causes a synchronous read from disk; unlike the normal cached I/O policy where read may be satisfied from the cache. This can result in very poor performance if the data was likely to be in memory under the normal caching policy.

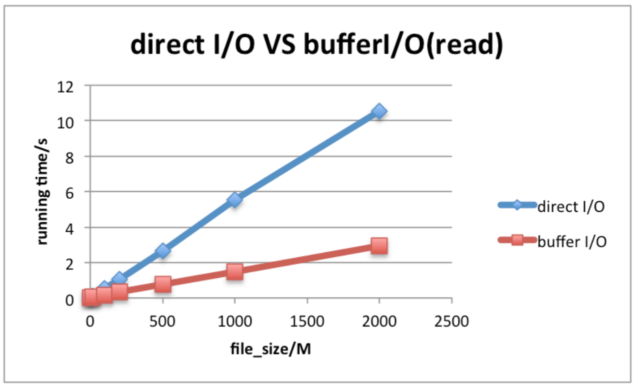


Figure 2. direct I/O VS buffer I/O(reads). When I only compare the read operation between directI/O and buffer I/O. I found with the increase of file size, the running of direct I/O increases aggressively, which means the performance of direct I/O decrease more aggressively than buffer I/O. And buffer I/O always performance better than direct I/O.

application_C(writes and reads): Buffer I/O win.

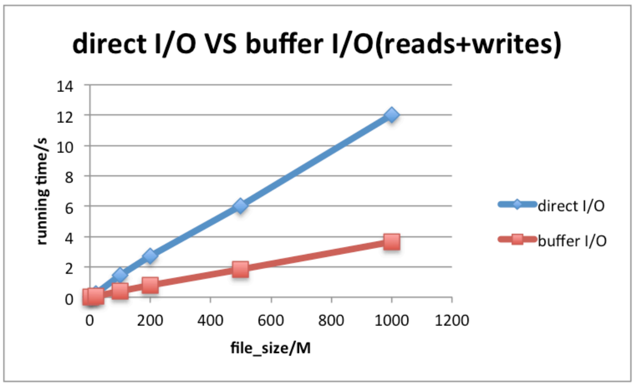


Figure 3. direct I/O VS buffer I/O(writes + read). When I compare the writes + read operation between directI/O and buffer I/O. I found that, with the increase of file size, the performance of direct I/O decreases more aggressively than buffer I/O. And buffer I/O always performance better than direct I/O.

application_D(sync writes): Direct I/O win.

When I add fsync() operation at the end of write() operation, actually the result/effect is same of Direct I/O and buffer I/O. But one different thing is direct I/O can bypass the operating system read and write caches. Thus, logically, direct I/O will perform better in this situation.

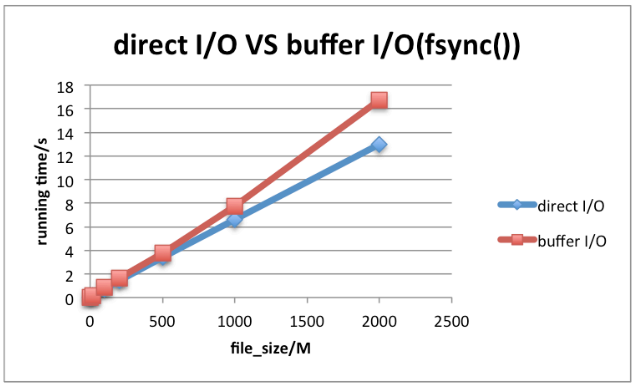


Figure 4. direct I/O VS buffer I/O(sync). When I compare the synchronous writes operation between directI/O and buffer I/O. I found that, with the increase of file size, the performance of direct I/O decreases almost as same as buffer I/O. And direct I/O always performance better than buffer I/O.

Conclusion:

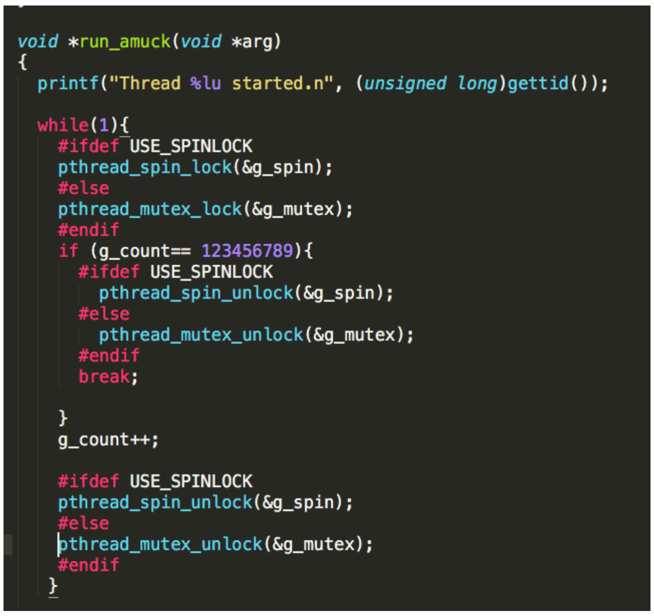
Buffer I/O wins when we compare writes and/or reads operations. It means that the cache of file system works and are very helpful for the writes and reads operation in users application.

But when we need synchronous writes, Direct I/O wins, because both of 2 I/O need to write data into disk, and direct I/O can bypass the operating system read and write caches.

- **Expriment 2: Concurrency: Spin-locks vs. mutexes**

The difference between Spin_locks and mutexes is that In mutex, if you find that resource is locked by someone else, you (the thread/process) switch the context and start to wait (non-blocking). Whereas spinlocks do not switch context and keep spinning. So the inetrval size and threads number is the most important factors to consider, because they are the key of the intense competition between processes/threads. Thus, I design application A with short interval, and comapare the performance of spin_lock and mutex, when I increase the threads number. I design application B with long interval, nd comapare the performance of spin_lock and mutex, when I increase the threads number.

Short interval(to do nothing but increment a counter): long interval(increment a counter for 100000 times):

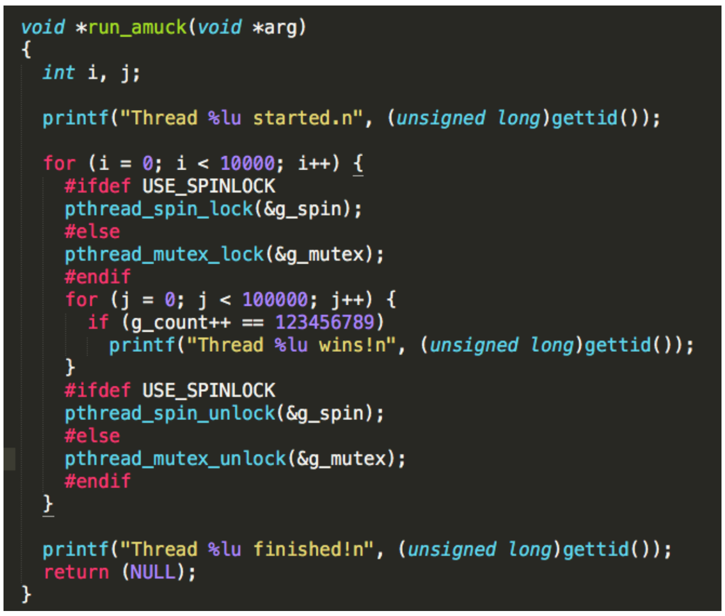


```
void *run_amuck(void *arg)
{
  printf("Thread %lu started.n", (unsigned long)gettid());

  while(1){
    #ifdef USE_SPINLOCK
    pthread_spin_lock(&g_spin);
    #else
    pthread_mutex_lock(&g_mutex);
    #endif
    if (g_count== 123456789){
      #ifdef USE_SPINLOCK
        pthread_spin_unlock(&g_spin);
      #else
        pthread_mutex_unlock(&g_mutex);
      #endif
      break;

    }
    g_count++;

    #ifdef USE_SPINLOCK
    pthread_spin_unlock(&g_spin);
    #else
    pthread_mutex_unlock(&g_mutex);
    #endif
   }
```



```
void *run_amuck(void *arg)
{
  int i, j;

  printf("Thread %lu started.n", (unsigned long)gettid());

  for (i = 0; i < 10000; i++) {
    #ifdef USE_SPINLOCK
    pthread_spin_lock(&g_spin);
    #else
    pthread_mutex_lock(&g_mutex);
    #endif
    for (j = 0; j < 100000; j++) {
      if (g_count++ == 123456789)
        printf("Thread %lu wins!n", (unsigned long)gettid());
    }
    #ifdef USE_SPINLOCK
    pthread_spin_unlock(&g_spin);
    #else
    pthread_mutex_unlock(&g_mutex);
    #endif
  }

  printf("Thread %lu finished!n", (unsigned long)gettid());
  return (NULL);
}
```

application_A(short interval): spin_lock I/O win 2 threads, when I increase threads, mutex win. This application demonstrate the differnence between direct I/O and buffer(default) I/O, when we have short interval(lock area).

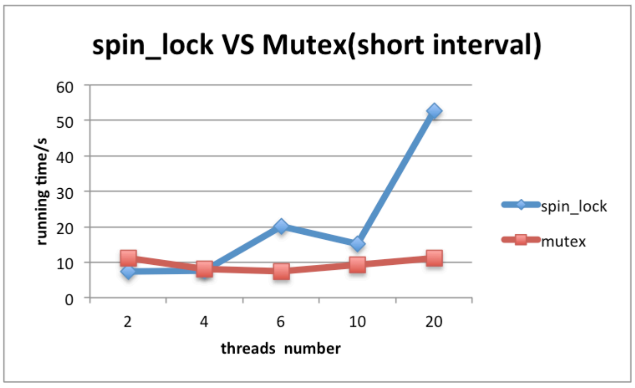


Figure 5. spin-locks vs mutexes(short interval), with the increasing of threads number. When we have only two threads, spin_lock perform better. However, with the increase of threads number, the performance of spin_lock decrease aggressively, and the performance of mutex are not influenced very much. So spin_lock perform better with short interval and 2 threads, but mutex perform better than spin_lock with many threads.

application_B(long interval): Mutex win. This application demonstrate the differnence between direct I/O and buffer(default) I/O, when we have long interval(lock area)

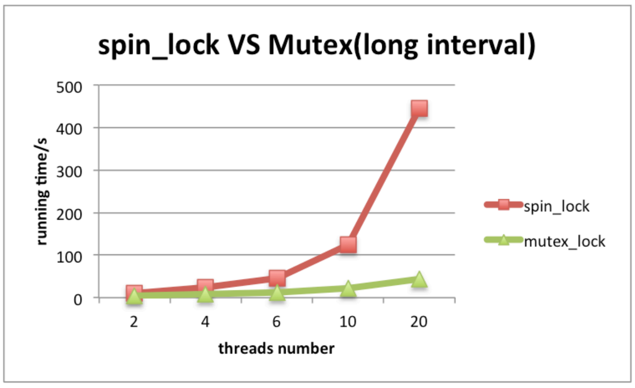


Figure 6. spin-locks vs mutexes(long interval), with the increasing of threads number. When we have only two threads, spin_lock perform worse than mutex. With the increase of threads number, the performance of spin_lock decrease aggressively, and the performance of mutex are not influenced very much. So mutex always perform better than spin_lock with long interval.

Conclusion:

Spin_lock performance decrease arrgressively when I increase the interval area and increase the threads number, while mutexes are not influenced very much.

When we have only two threads, as how bellow in Figure 7, spin_lock perform better than mutex with short interval, but nutex perform better than spin_lock with long interval. The reason is that, when we have only 2 threads and short interval, every thread will get the interval data quickly, spin_lock do not need to switch from waiting and working, but mutex need, so spin_lock will perform better. So, with long interval, there will be more intense competition between processes/threads with long interval and spin mode will damage the performance.

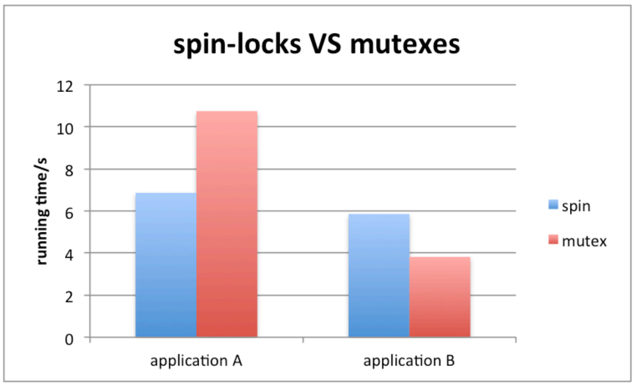


Figure 7. spin-locks vs mutexes(2 threads). When there are only 2 threads, spin_lock perform better than mutex with short interval, but nutex perform better than spin_lock with long interval. With long interval, more intense competition between processes.

But when we increase the number of threads, as show in Figure 5 and Figue 6, the running time of spin_lock will increase aggressively. The reason is that there will be more intense competition between processes with more threads, and ecery thread will nedd to wait for a long time to get the data, spin mode will dmage the performance.

Conclusions:

Every direct I/O write causes a synchronous write to disk, unlike the normal cached I/O policy where the data is merely copied and then written to disk later. This fundamental difference can cause a significant performance penalty for applications that are converted to use direct I/O. The primary benefit of direct I/O is to reduce CPU utilization for file reads and writes by eliminating the copy from the cache to the user buffer. When we need synchronous write for data, direct I/O will perform better than buffer I/O, because direct I/O can bypass the operating system read and write caches.

A spinlock can be better when you plan to hold the lock for an extremely short interval (for example to do nothing but increment a counter) and there are only 2 threads, and contention is expected to be rare, but the operation is occurring often enough to be a potential performance bottleneck. Another advantage of spin_lock is on unlock, there is no need to check if other threads may be waiting for the lock and waking them up. Unlocking is simply a single atomic write instruction. For the applications with long interval or have many threads, the performance of using spin_lock could be very bad.