OPTIMIZING FOR WORKLOADS: ANALYSIS OF SYSTEM CALL BEHAVIOR

Nathan P. Petersen
University of Wisconsin - Madison
npetersen2@wisc.edu

Abstract

System calls frequently and necessarily occur in user applications that interact with the underlying operating system. Careful evaluation must be given to these system calls and their configurations to maximize application performance. Specific user workloads can have significant influence on the behavior and performance of system calls. This paper analyzes two common abilities of the operating system: persistent media interaction and synchronization primitives. Various user workloads are constructed to exercise these common abilities. Performance metrics are collected. Results are presented along with evaluation and recommendations for future user application optimizations when using system calls.

1 Introduction

Operating systems form the backbone of modern processing environments. Millions of lines of code are responsible for enabling programs to easily execute on a given machine. These programs must share the machine's resources, such as processor cores, volatile memory, peripheral devices, and persistent media. Virtualization and resource management is accomplished by the underlying operating system, which ensures the system operates correctly and in an efficient manner. Machine resources should be largely devoted to the user applications which mandates careful system design and implementation.

User applications must interact with the operating system to effectively use the machine's resources. This interaction is realized using a set of well-defined interfaces (APIs) which programs can call to instruct the operating system to perform required actions. Modern systems export hundreds of *system calls* to the user application which each have very specific actions they perform. There are five categories of system calls: process control, file manipulation, device manipulation, information maintenance and communication. By combining and mixing the usage of these various system calls, user applications can accomplish a wide variety of tasks on a given machine, but cannot take over control of the hardware or steal data from other programs. This paper analyzes a subset of two system call categories: file management and process control.

File management describes the set of system calls which deal with the underlying file system, which is responsible for storing any files the user creates in a reliable and efficient manner on the disks of the system. The user application cannot interact directly with the persistent storage media, therefore generic operations are exposed via the operating system's APIs. Actions that can be invoked from user applications include creating a new file, opening and closing a file, reading and writing to a file, and other auxiliary actions such as seeking through files and renaming them. By using this agreed upon interface, operating systems can abstract away the mechanics of the actual file system, enabling applications to easily manipulate and persistent data.

Process control refers to the operating systems ability to run programs at various times while maintaining each program's illusion that full machine capabilities are available. The operating system will tend to swap out tasks to provide fair usage of the physical processing resources. Modern hardware systems often have multiple physical processing cores which can be used to run different programs in parallel, or can be exploited by a single program for increased processing throughput. When a single program creates multiple threads of execution, careful thought must be placed on how these threads interact and share common data to ensure program correctness while simultaneously realizing the potential performance gains. The operating system provides various synchronization primitives which the application can use to maintain data consistency and algorithmic correctness.

1.1 Problems

Persistent media interaction can be a major source of performance problems for many data-intensive applications. Many modern operating systems use a buffering layer between the user application and the underlying filesystem, which allows for clever techniques to increase I/O throughput and reduce perceived latency, while also hiding alignment requirements from users. While the buffered I/O technique is better in the common case, it can lead to lower performance for some applications with specific workload characteristics. This motivates operating systems to include optional modes which disable the buffering layer and permit direct access to the underlying media. Direct access certainly has potential performance gains, but when used improperly, can severely decrease performance compared to the buffered I/O. The following sections explore this further.

Maximum processing throughput naturally demands concurrency and full utilization of the available processing resources. Critical sections of code, where only a single thread can execute at a time, are common in high performance concurrent applications. Modern operating systems provide simple locking primitives to enforce the single thread constraint. Two common lock types are the spin lock and mutex. Each lock type creates a barrier to critical sections, thus maintaining program correctness. However, spin locks and mutexes are internally implemented very differently, leading to significant performance differences in various common workloads. The following sections explore this further.

1.2 Approach

Multiple workloads are designed to exhibit the differing performance observed between buffered and direct I/O, as well as spinlocks and mutexes. These workloads are benchmarked on a typical user system and performance metrics are collected for analysis. Care is given to ensure the resulting data and conclusions are repeatable. When possible, workloads are designed to represent a wide range of use cases so more widely encompassing conclusions can be made. The rest of this paper describes the various experiments performed as well as the results. Conclusions are made which highlight the major observations which can be utilized when developing user applications.

2 Experiments

All experiments are performed on a consumer desktop PC running Linux 4.4.8. The PC has a quad-core Intel i7-2600K processor with hyper-threading to enable eight hardware threads of execution. System memory available is 24GB. The persistent media is a 1TB hard disk drive (HDD) from Western Digital with 7200 RPM rotational speed and 32MB cache. The filesystem is ext4.

2.1 Local File System

Performance of the local file system is critical to data-intensive applications. Some workloads implement caching at the application level, while others depend on file system caching techniques for high performance. These differences should be apparent in the ways in which the applications configure the system calls. For application layer caching, direct I/O can offer improved performance. By default, buffering is used which normally leads to much better performance, however data must be copied twice (once to the file system buffer, once to the user buffer). This overhead is not always an issue, but depends on the workload characteristics, as well as the maximum throughput of the underlying persistent media technology. For example, the relative speeds of in-memory copies versus hard disk drive random access throughput is hardly comparable; the HDD dominates. When faster devices are used for persistent media, such as solid state drives (SSDs) or even non-volatile memory (NVM), the extra copy overhead can start be become a significant performance bottleneck.

2.1.1 Workloads

Multiple workloads are created to examine different use cases and interactions with the file system and underlying disk. Workload #1 is created to evaluate sequential append performance. It is expected that for relatively small writes, the buffering layer will combine these sequential writes and issue larger block writes to increase throughput. Workload #2 is created to evaluate random read performance. It is expected that randomly reading via direct I/O will be slightly faster, as data is only copied once directly to the user buffer, not twice through the file system buffer. Workload #3 is created to evaluate single block overwrite performance. It is expected that direct I/O will actually outperform buffered I/O because direct copying to disk is used, and a flush of cache is performed between block writes. Workload #4 is created similar to workload #3, but reads the block directly after writing to it. It is expected that direct I/O will again be faster because of less redundant copying. See Listing 1 for implementations.

Workload #1: Sequential Appends

```
fd = open("file",
          TEST_FLAGS)

// PROFILE START

for 1 to 256:
 write(fd, buf, size)

syncfs(fd)

close(fs)

// PROFILE END
```

Workload #2: Random Reads

```
fd = open("file",
          O_DIRECT)

for 1 to 16MB:
  write(fd, "a", 1)

close(fd)

fd = open("file",
          TEST_FLAGS)

// PROFILE START

for 1 to numblks:
  lseek(fd, rand())
  read(fd, buf, size)

// PROFILE END
```

Workload #3: Single Block Overwrite

```
fd = open("file",
          TEST_FLAGS)

// PROFILE START

for 1 to 64:
  lseek(fd, 0)
  write(fd, buf, size)
  syncfs(fs)

close(fs)

// PROFILE END
```

Workload #4: Single Block R/W

```
fd = open("file",
          TEST_FLAGS)

// PROFILE START

for 1 to 64:
  lseek(fd, 0)
  write(fd, bufw, size)
  syncfs(fd)
  read(fd, bufr, size)

close(fd)

// PROFILE END
```

Listing 1: Workload implementation for buffered vs. direct I/O comparison. Red highlight indicates profiling section.

2.1.2 Results

Sequential Appends Figure 1 shows the throughput of the sequential appends as a function of block size. As block size increases, throughput also increases up to ~1MB block sizes. At this point, disk throughput is mostly saturated, so increasing block size does not greatly influence performance. Buffered I/O is clearly faster than direct I/O for block sizes ranging from 4KB to 2MB. For block sizes larger than 2MB, it is unclear which I/O method is faster; they are approximately equivalent.

Random Reads Figure 2 shows the throughput of random reads as a function of block size. It was expected that direct I/O would have slight performance advantages over buffered I/O, but the data show that the two methods are essentially equivalent. As block size increases, so does throughput.

Single Block Overwrite Figure 3 shows the relative performance of direct to buffered I/O for overwriting a single block as a function of block size. A value of 1.0 indicates both direct and buffered I/O have the same performance. For small block sizes, 4KB to 16KB, direct I/O achieves higher throughput. At 4KB block size, direct I/O is 25% faster than buffered I/O. As block size increases, performance becomes similar.

Single Block Read/Write Figure 4 shows I/O operations per second (IOPS) for reading and writing to one file block as a function of block size. Buffered I/O achieves slightly more IOPS than direct I/O, but the performance gap decreases as block size increases.

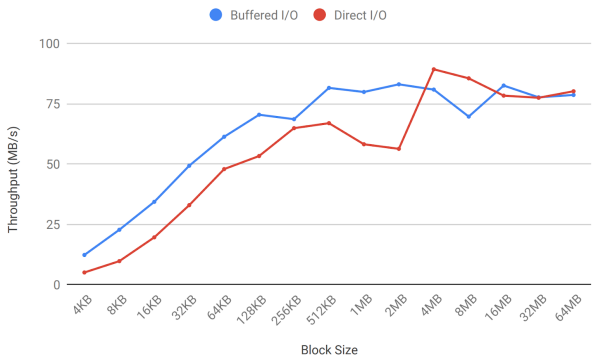

Figure 1: 256 sequential appends to file.
Data averaged over 50 runs.

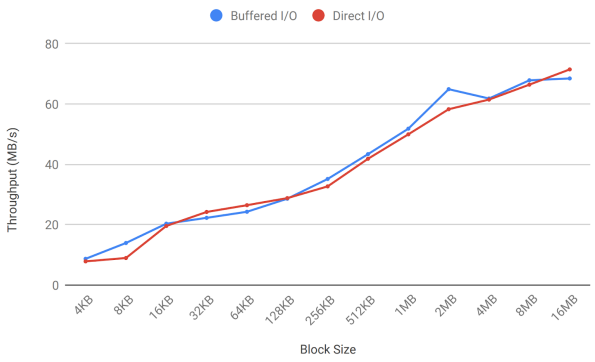

Figure 2: Randomly reading blocks from 16MB file.
Data averaged over 50 runs.

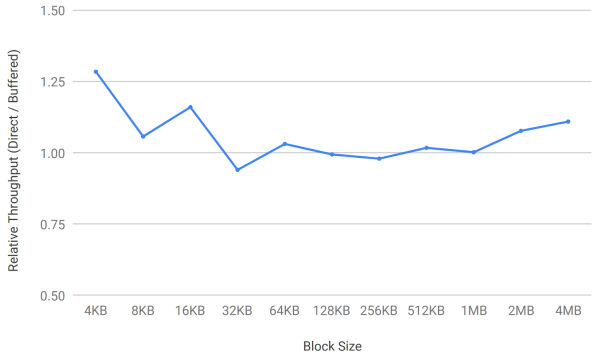

Figure 3: Overwriting single block 64 times.
Data averaged over 10 runs.

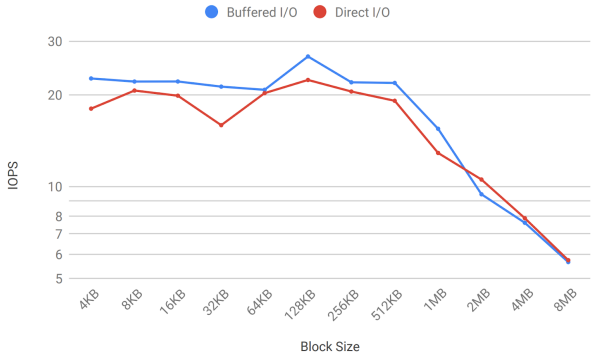

Figure 4: Single block write/read 64 times.
Data averaged over 10 runs.

2.1.3 Evaluation

For three out of the four tested workloads, buffered I/O performs at least as well as direct I/O. While the buffering layer requires double copying of all data, the throughput rates of hard disk drives are simply too small to notice this extra copying time, especially at small block sizes. At small block sizes, the file system is more performant if it combines reads or writes into larger requests. This is a result of the sheer increase in throughput for sequential disk accesses compared to random accesses.

The single workload that performed better for direct I/O than buffered I/O overwrote a single block many times. Bypassing the buffer cache and accessing the disk directly is beneficial in this instance because a flush call was made between every write. This forcibly slows down the buffering system and makes it perform like direct I/O, but with an additional copy for all data. Essentially, this is forcing bad performance onto the buffered I/O methodology and allows direct I/O to compete in terms of throughput.

2.2 Concurrency

While many different primitives exist to help applications manage concurrency, locks remain an easy-to-use solution. When one thread "holds" a lock, other threads are prevented from also holding the lock. Spinlocks are implemented such that the thread that fails to acquire the lock continually retries to get it. This wastes CPU time, but can result in better performance. Mutexes behave similarly to spinlocks at first, but after a few failed acquisitions, the thread is put to sleep until the lock is available. This allows the CPU to be used by other threads.

2.2.1 Workload

The performance of the locking mechanisms depends on multiple factors. In this paper, attention is given to the time spent in the critical section of code, as well as the number of threads attempting to acquire the lock. A simple workload is constructed which creates an array of threads to do work. This work consists of 100 accesses of the critical section. At run time, a parameter is passed to the program to set the duration a thread spends in the critical section. By sweeping both the number of threads and the duration of time spent in the critical section, interesting results appear. See Listing 2 for implementation details.

```
critical() {
  lock(&lock)

  for 1 to WORK:
    NOP

  unlock(&lock)
}

void *work(void *arg) {
  count = 1
  while count <= 100:
    critical()
    count++
}
```

```
pthread_t threads[NUM]
lock_t lock

main() {
  init_lock(&lock)

  // PROFILE START

  for i 1 to NUM:
    pthread_create(&threads[i], NULL, work)

  for i 1 to NUM:
    pthread_join(threads[i], NULL)

  // PROFILE END
}
```

Listing 2: Workload implementation for spinlock vs. mutex comparison.

2.2.2 Results

Figure 5 shows a plot of relative performance of the spinlock to the mutex. Each cell in the plot contains a number which is the ratio of performance. A value of 1.0 indicates both the spinlock and mutex have identical performance, while a value of 2.0 indicates the spinlock version of the workload completed its task twice as fast as the mutex version. A wide range of time spent in the critical section was evaluated, ranging from about one instruction to about one million instructions. For each of these values, a range of thread counts were evaluated, ranging from one thread to 100 threads. Coloring is given to the plot to show areas of general performance. Green indicates values greater than one, meaning spinlocks perform better. Red indicates values less than one, meaning mutexes perform better.

2.2.3 Evaluation

The wide range of thread count and critical instruction count leads to interesting data which can help inform the decision of when to use spinlocks versus mutexes. General knowledge is that spinlocks are inefficient and waste CPU time, while mutexes allow efficient allocation of processing time, therefore should always be used. Figure 5 clearly shows that there is much more nuance than this initial idea.

For program thread counts less than or equal to the number of threads available on the given machine, spin locks *always* perform as well or better than mutexes, in some cases up to twice as well. This can be explained because mutexes must put a thread to sleep and then later reawaken it, which is a very expensive process. As long as there are not starved threads which require the CPU, having a thread spin on a lock and simply wait for it to become free leads to significantly higher performance. Naturally, as the number of threads is increased to more than the physical hardware supports, the threads must share the CPU. Programs that have critical sections with more than 1k instructions and more than the hardware allowed number of threads show that mutexes allow much higher

performance. This is because the blocked thread is put to sleep, which enables another thread to do work.

Finally, it can be seen that for workloads with more than eight threads but less than 1k critical section instructions, spinlocks perform very well. This could suggest that the operating system must run approximately 1k instructions to put a thread to sleep and wake it up. Therefore, just keeping the thread awake and spinning is better for performance.

Number Threads \ Number of Critical Instructions	1	10	25	50	75	100	250	500	750	1k	5k	10k	25k	50k	75k	100k	250k	500k	1M
1	0.8	1.1	1.1	0.8	1.0	1.0	1.0	1.0	0.8	1.0	1.0	1.1	1.1	1.1	1.0	1.1	0.9	0.9	0.9
2	1.2	1.1	1.5	1.5	1.6	1.6	1.5	1.3	1.0	1.1	1.0	1.0	1.2	1.5	1.2	1.4	1.5	1.4	1.3
3	1.1	1.2	1.5	1.5	1.6	1.9	1.9	1.4	1.1	1.2	1.1	1.0	1.4	1.8	1.8	1.7	1.6	1.6	1.3
4	1.1	1.2	1.5	1.7	1.6	1.8	1.5	1.4	1.2	1.2	1.1	1.0	1.3	2.0	1.8	1.9	1.7	1.5	1.2
5	1.0	1.1	1.7	1.9	1.8	1.8	1.6	1.3	1.4	1.3	1.1	1.1	1.6	2.1	1.6	1.8	1.8	1.4	1.3
6	0.9	1.0	1.7	1.9	1.8	2.0	1.4	1.4	1.3	1.3	1.1	1.1	1.6	1.9	1.8	1.9	1.8	1.5	1.2
7	0.9	1.0	1.8	2.1	1.9	1.8	1.7	1.3	1.3	1.3	1.1	1.1	1.4	1.9	1.9	1.9	1.7	1.6	1.2
8	0.9	1.0	1.8	1.9	1.9	1.9	1.5	1.4	1.4	1.2	1.1	1.1	1.3	1.8	1.9	1.8	1.8	1.5	1.2
9	0.9	1.0	1.8	1.9	1.9	1.7	1.5	1.4	1.4	1.3	1.1	1.0	1.4	1.9	1.7	1.8	1.7	1.4	1.2
10	0.9	1.1	1.9	2.1	1.9	1.9	1.5	1.4	1.4	1.3	1.1	1.0	1.5	1.9	1.7	1.9	1.7	1.4	1.2
15	0.9	1.1	1.8	1.8	1.8	1.6	1.2	1.4	1.5	1.3	1.0	1.0	1.0	1.5	1.0	1.5	1.5	1.2	1.0
20	1.0	1.0	1.6	1.4	1.3	1.4	1.6	1.1	1.3	1.4	1.0	0.9	0.8	1.3	0.9	1.4	1.1	1.0	0.8
25	0.9	1.0	1.6	0.9	1.2	1.4	1.2	1.4	1.4	1.2	0.9	0.8	0.7	1.0	0.7	1.2	1.1	0.8	0.6
30	0.9	1.0	1.6	1.0	1.1	1.3	1.1	1.3	1.4	1.4	0.8	0.7	0.6	0.9	0.6	1.1	0.8	0.7	0.5
35	0.8	1.0	1.9	0.8	1.2	1.6	1.3	1.3	1.4	1.3	0.8	0.6	0.5	0.7	0.9	1.0	0.8	0.6	0.5
40	0.9	1.0	1.5	0.8	1.0	1.1	1.4	1.2	1.2	1.2	0.7	0.5	0.5	0.7	0.6	0.8	0.6	0.5	0.4
45	0.9	1.0	1.7	0.7	0.9	1.3	1.5	1.1	1.2	1.2	0.6	0.5	0.3	0.5	0.7	0.7	0.6	0.5	0.3
50	0.9	1.0	1.8	0.7	0.9	0.8	1.4	1.1	1.3	0.9	0.6	0.4	0.3	0.6	0.6	0.6	0.5	0.4	0.3
60	0.9	1.0	1.6	0.6	1.0	1.1	1.3	1.2	1.1	1.1	0.5	0.4	0.3	0.4	0.6	0.4	0.4	0.3	0.2
70	0.9	1.0	1.2	0.6	0.8	1.0	1.4	1.2	1.2	0.8	0.4	0.4	0.2	0.4	0.5	0.4	0.4	0.3	0.2
80	0.9	1.6	1.9	0.6	0.8	1.1	1.3	1.3	1.1	0.9	0.4	0.3	0.2	0.3	0.4	0.4	0.3	0.2	0.2
90	0.9	1.6	1.9	0.5	0.8	0.9	1.5	1.2	0.9	0.9	0.3	0.3	0.2	0.3	0.3	0.3	0.3	0.2	0.2
100	0.9	1.0	1.9	0.6	0.8	0.7	1.2	1.0	0.9	0.6	0.3	0.2	0.2	0.2	0.3	0.3	0.2	0.2	0.2
	10,000 run average					100 run average			50 run average						10 run average				

Figure 5: Spinlock vs. mutex relative performance.
Values >1.0 indicate spinlock workload is faster than mutex workload
(e.g. 2.0 means workload throughput using spinlock is 2x faster than mutex).
Values are averaged over multiple runs.

3 Conclusions

Careful study of various workloads for file system operations as well as thread synchronization is performed. It is shown that for accessing persistent media such as hard disks, using buffered I/O generally leads to better performance. This is because the file system composes many small accesses into larger ones to exploit fast sequential accesses. Only for very specific workloads, when small amounts of data are flushed constantly to disk, does direct I/O offer any advantages over buffered I/O. Application developers must carefully study their workloads to determine if buffered or direct I/O should be used.

Both spinlocks and mutexes are tested in a wide range of combinations of thread count and critical section instructions. It is shown that for workloads with less than or equal number of threads than the physical hardware allows, using spinlocks can lead to higher performance (up to 2x). However, when critical section instruction count increases to above five thousand instructions and thread count rises above hardware supported amount, mutexes will perform better, as threads are put to sleep to make way for other threads to effectively use the CPU. Users must take into consideration the number of cycles in an application's critical section to determine appropriate lock type.