CSCI 5103 Operating System Project 2

Group Member
Yutong Zhao 5593405
Qi Le 5674954

Lock vs SpinLock

Which lock provides better performance in your testing? Why do you think that is?
Answer: Test file is test_lock_vs_spinlock.cpp. Running command is in Makefile -> sudo make lock-vs-spinlock ./lock-vs-spinlock thread_num

lock vs spinlock										
Simple				Simple_yield				Complex		
thread	Lock(us)	SpinLock(us)		thread	Lock(us)	SpinLock(us)		thread	Lock(us)	SpinLock(us)
1	11	5		1	9	6		1	24	23
3	23	12		3	33	47432		3	73	66
10	57	34		10	96	716165		10	228	215
30	240	124		30	335	6958118		30	776	661
99	517	341		99	969	77634089		99	2294	2152

Table 1. Comparison between the Performance (Measured in us) of the Lock and SpinLock under different function and different number of threads

simple worker() function: Add 1 to a static variable
simple yield worker() function: Add 1 to a static variable and switch thread
long worker() function: Complete a for loop

Three different functions including simple worker() function, simple yield worker() function and long worker() function and varies numbers of threads have been tested to show the generalization performance of the two different locks.

From Table 1 Simple() function, we can find out that the SpinLock is a little bit faster than Lock because the SpinLock is simpler than Lock and SpinLock only runs for a short critical section, which does not need context switching.

From Table 1 Simple_yield() function, we can find out that the Lock out performance the SpinLock. Spinlock is very slow because all other workers must wait for multiple time slices before any worker completes the current work.

From Table 1 Complex() function, we can find that the SpinLock and the Lock have almost the same performance, which is unexpected. From previous observations, the result from the complex function would be similar to that of the Simple_yield function. One explanation would be that the for loop may be optimized by the compiler.

How does the size of a critical section affect the performance of each type of lock? Explain with results.

The Lock would have better performance when the critical section is longer. For small critical sections, the overhead of context switching in the Lock would be longer than the waiting time in the SpinLock. Furthermore, when the critical section's running time is less than time quantum, the Spinlock is generally faster; otherwise, the Lock is faster.

uthread is a uniprocessor user-thread library. How might the performance of the lock types be affected if they could be used in parallel by a multi-core system?

For multi-core system, if the expected time for the thread to wait for the lock is very short, which is shorter than the context switching time of the thread twice, it is cost-effective to use a SpinLock. If the expected time for the thread to wait for the lock for a long time, which is shorter than the context switching time of the thread twice, it is recommended to use a Lock.

Generally, the performance of Lock is better under the multi-core system (consider the case of n processors with 2n threads, the first n uses the same lock, and the last n uses another lock). Lock is suitable for scenes where the lock operation is very frequent and the critical section is large, and it has better adaptability.

For SpinLock, assuming there are two threads are running on two cores. Most of the time, only one thread can get the lock, so the other thread has been busy waiting on the core it is running, and the CPU occupancy rate is always 100%. The busy waiting would cost a huge amount of time. Moreover, we use Test-and-Set Lock for SpinLock, the time to execute a critical section would increase rapidly when the number of processors are increasing.

Are there any other interesting results from your testing?
Yes. We don't expect that the SpinLock would be so slow in the Simple_yield() function.

Synchronous vs Asynchronous I/O

Answer: Test file is test_lock_vs_spinlock.cpp. Running command is in Makefile -> sudo make sync-vs-async ./sync-vs-async io_thread_num io_rw_length comp_thread_num comp_work_load.

Testing Function:
ioworker() function: write and read files (sync/async). This function would test whether amount of I/O affect the performance of each type of I/O or not.
compworker() function: Compute the sum of several random numbers. This function would test whether amount of other available thread work affect the performance of each

type of I/O or not.

IO Changing						Other Threads Changing		
IO = 1*x, COMP = 1 * 1			IO = x*10000000, COMP = 1 * 1			IO = 1*10000000, COMP = x * 100000		
x	Sync(us)	Async(us)	x	sync(us)	async(us)	x	sync(us)	async(us)
1	82	6870	1	902	7415	1	1284	4545
10	94	8674	3	195	8101	3	2755	6358
100	112	8228	10	2873	8918	10	6835	19658
1000	173	8482	30	10394	18446	30	20492	56192
10000	185	6350	90	208920	206022	90	57977	156638
100000	116	8882						

*Table 2. Comparison between the Performance (Measured in us) of the Synchronous I/O and Asynchronous I/O under different conditions. In the Table, IO = 1*x, 1 represents 1 I/O thread and x represents the length of each thread to read and write is x. IO = x*10000000, x represents x I/O threads and 10000000 represents the length of each thread to read and write is 10000000, COMP = x * 100000, x represents x computing threads and 100000 represents the length to be calculated for each thread is 100000*

Which I/O type provides better performance in your testing? Why do you think that is?

From Table 2, We can find out that when the amount of I/O thread is small, the Synchronous I/O is much faster than the Asynchronous I/O and provides better performance in all situation. The length of each thread to read and write affects the result slightly. But when the amount of I/O threads becomes larger and larger (90), the performance of both types of I/O becomes closer and closer. Following the trend, the Asynchronous I/O will provide better performance when the amount of I/O threads keep increasing. The reason is that the Asynchronous I/O does not block the execution of the following code, but the Synchronous I/O blocks the execution of the following code.

After testing the effect of other available threads, we find out that the Synchronous I/O has better performance than Asynchronous I/O. When the I/O is not intensive, we do not need to wait Synchronous I/O for a long time and the Synchronous I/O saves the time of thread switching.

How does the amount of I/O affect the performance of each type of I/O? Explain with results.

With the growth of I/O, the Synchronous I/O becomes faster than the Asynchronous I/O. The reason is that the Asynchronous I/O keeps almost the same performance when I/O is intensive while Synchronous I/O can only process one I/O at the same time, which leads to the great increase of waiting time.

How does the amount of other available thread work affect the performance of each type of I/O? Explain with results.

In our result, the complete time of Synchronous I/O increases from 1284(us) to 20492(us) at the first 30 threads and the complete time of Synchronous I/O increases from 20492(us) to 57977(us) at the next 60 threads. We can find out that with increasing amount of other available threads, the overhead impact brought by Synchronous I/O is getting smaller and smaller.

The complete time of Asynchronous I/O increases from 4545(us) to 56192(us) at the first 30 threads and the complete time of Asynchronous I/O increases from 56192(us) to 156638(us) at the next 60 threads. At beginning, the Asynchronous I/O has the extra load of thread switching. Then with increasing amount of other available threads, the effect of the extra load of thread switching is getting smaller and smaller.

The increase in completing time of Asynchronous I/O is larger than that of the Synchronous I/O. So, when the I/O is not intense, Synchronous I/O would be a better choice.

Priority Inversion

Explain your test and how your implementation works in the README file.

Priority ceiling is used in our code.

1. In thread_init() function and thread_create() function, we set the priority of the main thread and the priority of the new thread to RED priority.
2. Each time the current thread with free lock call the condition variable() or the current thread with empty waiting queue and the signaling queue when it calls the unlock(), the lock count of the thread would change.
3. In the timehandler() function, we check the lock count of the running thread. If the running thread owns lock, we would maintain its priority; otherwise, we would decrease its priority by one.

Test file is test_priority_inversion.cpp. Running command is in Makefile -> sudo make priority-inversion ./priority-inversion

In the test_priority_inversion.cpp, We create 3 threads: C, B and A. We set the priority of the 3 threads to GREEN, ORANGE and RED respectively. In the beginning, C holds the lock and A requires the lock. We set the running time of C longer than one slice to make sure that it can be preempted by the thread B. We expect the result would be C finishes first, then A, then B.

Other Test Files:

Test-zoo:

Testing content	All contents of lock() and condvar()
Testing method	See the comments in the code
Operation method	sudo make test-zoo
	./test-zoo 90
Expected result	1. When only printing zoo, directly accept all animals -> Meets broadcast expectations under hoared semantics
	2. Zoo still keeps all Room->signal running at the end

:

Test-async-io:

Testing content	Asynchronous IO
Testing method	1. Generate a file, write hello world, then read
	2. Ensure that the IO is asynchronous by printing the waiting information after the switch main thread.
Operation method	sudo make test-async-io
	sudo ./test-async-io
Expected result	1. The first line: Waiting for io
	2. Second line: The result read from the file