Lab: locks

In this lab you'll gain experience in re-designing code to increase parallelism. A common symptom of poor parallelism on multi-core machines is high lock contention. Improving parallelism often involves changing both data structures and locking strategies in order to reduce contention. You'll do this for the xv6 memory allocator and block cache.

Before writing code, make sure to read the following parts from the xv6 book :

- Chapter 6: "Locking" and the corresponding code.
- Section 3.5: "Code: Physical memory allocator"
- Section 8.1 through 8.3: "Overview", "Buffer cache layer", and "Code: Buffer cache"

```
  $ git fetch
  $ git checkout lock
  $ make clean
```

Memory allocator (moderate)

The program user/kalloctest stresses xv6's memory allocator: three processes grow and shrink their address spaces, resulting in many calls to `kalloc` and `kfree`. `kalloc` and `kfree` obtain `kmem.lock`. kalloctest prints (as "#test-and-set") the number of loop iterations in `acquire` due to attempts to acquire a lock that another core already holds, for the `kmem` lock and a few other locks. The number of loop iterations in `acquire` is a rough measure of lock contention. The output of `kalloctest` looks similar to this before you start the lab:

```
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #test-and-set 83375 #acquire() 433015
lock: bcache: #test-and-set 0 #acquire() 1260
--- top 5 contended locks:
lock: kmem: #test-and-set 83375 #acquire() 433015
lock: proc: #test-and-set 23737 #acquire() 130718
lock: virtio_disk: #test-and-set 11159 #acquire() 114
lock: proc: #test-and-set 5937 #acquire() 130786
lock: proc: #test-and-set 4080 #acquire() 130786
tot= 83375
test1 FAIL
start test2
total free number of pages: 32497 (out of 32768)
.....
test2 OK
start test3
child done 1
child done 100000
test3 OK
start test2
total free number of pages: 32497 (out of 32768)
```

```
.....
test2 OK
start test3
child done 1
child done 100000
test3 OK
```

You'll likely see different counts than shown here, and a different order for the top 5 contended locks.

`acquire` maintains, for each lock, the count of calls to `acquire` for that lock, and the number of times the loop in `acquire` tried but failed to set the lock. kalloctest calls a system call that causes the kernel to print those counts for the kmem and bcache locks (which are the focus of this lab) and for the 5 most contended locks. If there is lock contention the number of `acquire` loop iterations will be large. The system call returns the sum of the number of loop iterations for the kmem and bcache locks.

For this lab, you must use a dedicated unloaded machine with multiple cores. If you use a machine that is doing other things, the counts that kalloctest prints will be nonsense. You can use a dedicated Athena workstation, or your own laptop, but don't use a dialup machine.

The root cause of lock contention in kalloctest is that `kalloc()` has a single free list, protected by a single lock. To remove lock contention, you will have to redesign the memory allocator to avoid a single lock and list. The basic idea is to maintain a free list per CPU, each list with its own lock. Allocations and frees on different CPUs can run in parallel, because each CPU will operate on a different list. The main challenge will be to deal with the case in which one CPU's free list is empty, but another CPU's list has free memory; in that case, the one CPU must "steal" part of the other CPU's free list. Stealing may introduce lock contention, but that will hopefully be infrequent.

Your job is to implement per-CPU freelists, and stealing when a CPU's free list is empty. You must give all of your locks names that start with "kmem". That is, you should call `initlock` for each of your locks, and pass a name that starts with "kmem". Run kalloctest to see if your implementation has reduced lock contention. To check that it can still allocate all of memory, run `usertests sbrkmuch`. Your output will look similar to that shown below, with much-reduced contention in total on kmem locks, although the specific numbers will differ. Make sure all tests in `usertests -q` pass. `make grade` should say that the kalloctests pass.

```
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #test-and-set 0 #acquire() 42843
lock: kmem: #test-and-set 0 #acquire() 198674
lock: kmem: #test-and-set 0 #acquire() 191534
lock: bcache: #test-and-set 0 #acquire() 1242
--- top 5 contended locks:
lock: proc: #test-and-set 43861 #acquire() 117281
lock: virtio_disk: #test-and-set 5347 #acquire() 114
lock: proc: #test-and-set 4856 #acquire() 117312
lock: proc: #test-and-set 4168 #acquire() 117316
lock: proc: #test-and-set 2797 #acquire() 117266
tot= 0
test1 OK
start test2
```

```
total free number of pages: 32499 (out of 32768)
.....
test2 OK
start test3
child done 1
child done 100000
test3 OK
$ usertests sbrkmuch
usertests starting
test sbrkmuch: OK
ALL TESTS PASSED
$ usertests -q
...
ALL TESTS PASSED
$
```

Some hints:

- You can use the constant `NCPU` from kernel/param.h
- Let `freerange` give all free memory to the CPU running `freerange`.
- The function `cpuid` returns the current core number, but it's only safe to call it and use its result when interrupts are turned off. You should use `push_off()` and `pop_off()` to turn interrupts off and on.
- Have a look at the `snprintf` function in kernel/sprintf.c for string formatting ideas. It is OK to just name all locks "kmem" though.
- Optionally run your solution using xv6's race detector:

  ```
  $ make clean
  $ make KCSAN=1 qemu
  $ kalloctest
    ..
  ```

  The `kalloctest` may fail but you shouldn't see any races. If the xv6's race detector observes a race, it will print two stack traces describing the races along the following lines:

  ```
  == race detected ==
  backtrace for racing load
  0x000000008000ab8a
  0x000000008000ac8a
  0x000000008000ae7e
  0x0000000080000216
  0x00000000800002e0
  0x0000000080000f54
  0x0000000080001d56
  0x0000000080003704
  0x0000000080003522
  0x0000000080002fdc
  backtrace for watchpoint:
  0x000000008000ad28
  0x000000008000af22
  0x000000008000023c
  0x0000000080000292
  0x0000000080000316
  0x000000008000098c
  0x0000000080000ad2
  0x000000008000113a
  0x0000000080001df2
  0x000000008000364c
  ```

```
0x0000000080003522
0x0000000080002fdc
==========
```

On your OS, you can turn a back trace into function names with line numbers by cutting and pasting it into `addr2line`:

```
  $ riscv64-linux-gnu-addr2line -e kernel/kernel
  0x000000008000ab8a
  0x000000008000ac8a
  0x000000008000ae7e
  0x0000000080000216
  0x00000000800002e0
  0x0000000080000f54
  0x0000000080001d56
  0x0000000080003704
  0x0000000080003522
  0x0000000080002fdc
ctrl-d
kernel/kcsan.c:157
kernel/kcsan.c:241
kernel/kalloc.c:174
kernel/kalloc.c:211
kernel/vm.c:255
kernel/proc.c:295
kernel/sysproc.c:54
kernel/syscall.c:251
```

You are not required to run the race detector, but you might find it helpful. Note that the race detector slows xv6 down significantly, so you probably don't want to use it when running `usertests`.

Buffer cache (hard)

This half of the assignment is independent from the first half; you can work on this half (and pass the tests) whether or not you have completed the first half.

If multiple processes use the file system intensively, they will likely contend for `bcache.lock`, which protects the disk block cache in kernel/bio.c. `bcachetest` creates several processes that repeatedly read different files in order to generate contention on `bcache.lock`; its output looks like this (before you complete this lab):

```
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem: #test-and-set 0 #acquire() 33035
lock: bcache: #test-and-set 16142 #acquire() 65978
--- top 5 contended locks:
lock: virtio_disk: #test-and-set 162870 #acquire() 1188
lock: proc: #test-and-set 51936 #acquire() 73732
lock: bcache: #test-and-set 16142 #acquire() 65978
lock: uart: #test-and-set 7505 #acquire() 117
lock: proc: #test-and-set 6937 #acquire() 73420
tot= 16142
```

```
test0: FAIL
start test1
test1 OK
```

You will likely see different output, but the number of test-and-sets for the `bcache` lock will be high. If you look at the code in `kernel/bio.c`, you'll see that `bcache.lock` protects the list of cached block buffers, the reference count (`b->refcnt`) in each block buffer, and the identities of the cached blocks (`b->dev` and `b->blockno`).

> Modify the block cache so that the number of `acquire` loop iterations for all locks in the bcache is close to zero when running `bcachetest`. Ideally the sum of the counts for all locks involved in the block cache should be zero, but it's OK if the sum is less than 500. Modify `bget` and `brelse` so that concurrent lookups and releases for different blocks that are in the bcache are unlikely to conflict on locks (e.g., don't all have to wait for `bcache.lock`). You must maintain the invariant that at most one copy of each block is cached. When you are done, your output should be similar to that shown below (though not identical). Make sure 'usertests -q' still passes. `make grade` should pass all tests when you are done.

```
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem: #test-and-set 0 #acquire() 32954
lock: kmem: #test-and-set 0 #acquire() 75
lock: kmem: #test-and-set 0 #acquire() 73
lock: bcache: #test-and-set 0 #acquire() 85
lock: bcache.bucket: #test-and-set 0 #acquire() 4159
lock: bcache.bucket: #test-and-set 0 #acquire() 2118
lock: bcache.bucket: #test-and-set 0 #acquire() 4274
lock: bcache.bucket: #test-and-set 0 #acquire() 4326
lock: bcache.bucket: #test-and-set 0 #acquire() 6334
lock: bcache.bucket: #test-and-set 0 #acquire() 6321
lock: bcache.bucket: #test-and-set 0 #acquire() 6704
lock: bcache.bucket: #test-and-set 0 #acquire() 6696
lock: bcache.bucket: #test-and-set 0 #acquire() 7757
lock: bcache.bucket: #test-and-set 0 #acquire() 6199
lock: bcache.bucket: #test-and-set 0 #acquire() 4136
lock: bcache.bucket: #test-and-set 0 #acquire() 4136
lock: bcache.bucket: #test-and-set 0 #acquire() 2123
--- top 5 contended locks:
lock: virtio_disk: #test-and-set 158235 #acquire() 1193
lock: proc: #test-and-set 117563 #acquire() 3708493
lock: proc: #test-and-set 65921 #acquire() 3710254
lock: proc: #test-and-set 44090 #acquire() 3708607
lock: proc: #test-and-set 43252 #acquire() 3708521
tot= 128
test0: OK
start test1
test1 OK
$ usertests -q
  ...
ALL TESTS PASSED
$
```

Please give all of your locks names that start with "bcache". That is, you should call `initlock` for each of your locks, and pass a name that starts with "bcache".

Reducing contention in the block cache is more tricky than for kalloc, because bcache buffers are truly shared among processes (and thus CPUs). For kalloc, one could eliminate most contention by giving each CPU its own allocator; that won't work for the block cache. We suggest you look up block numbers in the cache with a hash table that has a lock per hash bucket.

There are some circumstances in which it's OK if your solution has lock conflicts:

- When two processes concurrently use the same block number. `bcachetest test0` doesn't ever do this.
- When two processes concurrently miss in the cache, and need to find an unused block to replace. `bcachetest test0` doesn't ever do this.
- When two processes concurrently use blocks that conflict in whatever scheme you use to partition the blocks and locks; for example, if two processes use blocks whose block numbers hash to the same slot in a hash table. `bcachetest test0` might do this, depending on your design, but you should try to adjust your scheme's details to avoid conflicts (e.g., change the size of your hash table).

`bcachetest`'s `test1` uses more distinct blocks than there are buffers, and exercises lots of file system code paths.

Here are some hints:

- Read the description of the block cache in the xv6 book (Section 8.1-8.3).
- It is OK to use a fixed number of buckets and not resize the hash table dynamically. Use a prime number of buckets (e.g., 13) to reduce the likelihood of hashing conflicts.
- Searching in the hash table for a buffer and allocating an entry for that buffer when the buffer is not found must be atomic.
- Remove the list of all buffers (`bcache.head` etc.) and don't implement LRU. With this change `brelse` doesn't need to acquire the bcache lock. In `bget` you can select any block that has `refcnt == 0` instead of the least-recently used one.
- You probably won't be able to atomically check for a cached buf and (if not cached) find an unused buf; you will likely have to drop all locks and start from scratch if the buffer isn't in the cache. It is OK to serialize finding an unused buf in `bget` (i.e., the part of `bget` that selects a buffer to re-use when a lookup misses in the cache).
- Your solution might need to hold two locks in some cases; for example, during eviction you may need to hold the bcache lock and a lock per bucket. Make sure you avoid deadlock.
- When replacing a block, you might move a `struct buf` from one bucket to another bucket, because the new block hashes to a different bucket. You might have a tricky case: the new block might hash to the same bucket as the old block. Make sure you avoid deadlock in that case.
- Some debugging tips: implement bucket locks but leave the global bcache.lock acquire/release at the beginning/end of bget to serialize the code. Once you are sure it is correct without race conditions, remove the global locks and deal with concurrency issues. You can also run `make CPUS=1 qemu` to test with one core.
- Use xv6's race detector to find potential races (see above how to use the race detector).

Submit the lab

This completes the lab. Make sure you pass all of the make grade tests. If this lab had questions, don't forget to write up your answers to the questions in answers-*lab-name*.txt. Commit your changes (including adding answers-*lab-name*.txt) and type make handin in the lab directory to hand in your lab.

Time spent

Create a new file, `time.txt`, and put in it a single integer, the number of hours you spent on the lab. Don't forget to `git add` and `git commit` the file.

Submit

You will turn in your assignments using the submission website. You need to request once an API key from the submission website before you can turn in any assignments or labs.

After committing your final changes to the lab, type **`make handin`** to submit your lab.

```
$ git commit -am "ready to submit my lab"
[util c2e3c8b] ready to submit my lab
 2 files changed, 18 insertions(+), 2 deletions(-)

$ make handin
tar: Removing leading `/' from member names
Get an API key for yourself by visiting https://6828.scripts.mit.edu/2022/handin.py/
Please enter your API key: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 79258  100   239  100 79019    853   275k --:--:-- --:--:-- --:--:--  276k
$
```

`make handin` will store your API key in *myapi.key*. If you need to change your API key, just remove this file and let **`make handin`** generate it again (*myapi.key* must not include newline characters).

If you run **`make handin`** and you have either uncomitted changes or untracked files, you will see output similar to the following:

```
 M hello.c
?? bar.c
?? foo.pyc
Untracked files will not be handed in.  Continue? [y/N]
```

Inspect the above lines and make sure all files that your lab solution needs are tracked i.e. not listed in a line that begins with **`??`**. You can cause `git` to track a new file that you create using `git add filename`.

If **`make handin`** does not work properly, try fixing the problem with the curl or Git commands. Or you can run **`make tarball`**. This will make a tar file for you, which you can then upload via our web interface.

- Please run `make grade` to ensure that your code passes all of the tests
- Commit any modified source code before running `make handin`
- You can inspect the status of your submission and download the submitted code at https://6828.scripts.mit.edu/2022/handin.py/

Optional challenge exercises

- maintain the LRU list so that you evict the least-recently used buffer instead of any buffer that is not in use.
- make lookup in the buffer cache lock-free. Hint: use gcc's `__sync_*` functions. How do you convince yourself that your implementation is correct?