CS 33

Multithreaded Programming V
The use of the lock prefix causes “interlocked access” to memory: the memory bus is grabbed for a few cycles to insure that data can be read, modified, then written without anything happening in between. Note that this has a detrimental effect on overall system performance: while one core has “grabbed” the memory bus, other cores attempting to access memory must wait.
Alternatives to Mutexes:
Spin Locks

• Consider

    pthread_mutex_lock(&mutex);
    x = x+1;
    pthread_mutex_unlock(&mutex);

• A lot of overhead is required to put thread to
  sleep, then wake it up
• Rather than do that, repeatedly test mutex
  until it’s unlocked, then lock it
  – makes sense only on multiprocessor system
The source operand must be a register.
If \textit{cmpxchg} is used with the lock prefix, it provides a means for implementing a spin lock, as shown in the next slide.
In this code, a pointer to the spin lock is passed as an argument (in %edi). The `cmpxchg` instruction compares the spin lock’s value with the unlocked value (zero, which is in %eax). If equal (it is unlocked), then a 1 (the contents of %r10) is copied into the spin lock, thus setting it to locked. Otherwise the spin lock’s value (presumably 1) is copied into %r10 (and ignored). `cmpxchg` is called repeatedly until the spin lock is found to be 0 (unlocked) and then atomically set to 1 (locked).

Unlocking the spin lock is done by simply setting it to zero.
A problem with the approach of the previous slide is that `cmpxchg`, when executed as an interlocked instruction, is fairly expensive and disruptive to all processors, since it must grab the memory bus for a couple cycles. This is particularly a problem if it is executed repeatedly, as in the previous slide. What's done here is to “spin” using normal compare instructions, then, once the lock is found to be unlocked, to verify that it’s still unlocked and lock it using `cmpxchg`. This has the effect of minimizing the number of times the instruction is used, and thus minimizing the amount of time the memory bus is held.
Spin locks are a simple form of synchronization useful only on multiprocessors. The lock is represented by a single bit. To lock a spin lock, a thread repeatedly tests the lock until it finds it unlocked, then sets it to locked (atomically). The advantage over standard mutexes is that, if two threads are competing for access to a critical section, which both will hold only for a short period of time, it requires fewer instructions for one thread to “wait” for the other by repeatedly testing the lock a few times than it does for it to place a system call to put itself to sleep, then for the holder of the lock to place another system call to wake the sleeping thread up.

While spin locks were not part of POSIX 1003.1c, they were added to the most recent POSIX threads specification, 1003.1j.
Implementing Mutexes (1)

• Strategy
  – make the usual case (no waiting) very fast
  – can afford to take more time for the other case
    (waiting for the mutex)
Implementing Mutexes (2)

- Mutex has three states
  - unlocked
  - locked, no waiters
  - locked, waiting threads

- Locking the mutex
  - use cmpxchg (with lock prefix)
  - if unlocked, lock it and we’re done
    » state changed to locked, no waiters
  - otherwise, make “futex” system call to wait till it’s unlocked
    » state changed to locked, waiting threads

The actual solution involves some complications not apparent here — see http://people.redhat.com/drepper/futex.pdf for details.
Implementing Mutexes (3)

- Unlocking the mutex
  - if locked, but no waiters
    - state changed to unlocked
  - if locked, but waiting threads
    - futex system call made to wake up a waiting thread
In a naïve multithreaded implementation of malloc/free, there is one mutex protecting the heap, resulting in a bottleneck.
Solution 1

• Divvy up the heap among the threads
  – each thread has its own heap
  – no mutexes required
  – no bottleneck

• How much heap does each thread get?
Solution 2

- **Multiple “arenas”**
  - each with its own mutex
  - thread allocates from the first one it can find whose mutex was unlocked
    » if none, then creates new one
  - deallocations go back to original arena
Solution 3

- Global heap plus per-thread heaps
  - threads pull storage from global heap
  - freed storage goes to per-thread heap
    » unless things are imbalanced
    • then thread moves storage back to global heap
  - mutex on only the global heap
- What if one thread allocates and another frees storage?
Malloc/Free Implementations

- ptmalloc
  - based on solution 2
  - in glibc (i.e., used by default)
- tcmalloc
  - based on solution 3
  - from Google
- Which is best?
Test Program

const unsigned int N=64, nthreads=32, iters=10000000;
int main() {
    void *tfunc(void *);
    pthread_t thread[nthreads];
    for (int i=0; i<nthreads; i++) {
        pthread_create(&thread[i], 0, tfunc, (void *)i);
        pthread_detach(thread[i]);
    }
    pthread_exit(0);
}

void *tfunc(void *arg) {
    long i;
    for (i=0; i<iters; i++) {
        long *p = (long *)malloc(sizeof(long)*((i%N)+1));
        free(p);
    }
    return 0;
}
Compiling It …

% gcc -o ptalloc alloc.cc -lpthread
% gcc -o tcalloc alloc.cc -lpthread -ltcmalloc
The code was run on an Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz.
What’s Going On?

$ strace -c -f ./ptalloc
...
% time  seconds  usecs/call  calls  errors  syscall
-------- -------- -------- -------- -------- --------
100.00   0.040002  13   3007   520  futex
...

$ strace -c -f ./talloc
...
% time  seconds  usecs/call  calls  errors  syscall
-------- -------- -------- -------- -------- --------
... 0.00  0.000000  0   59   13  futex
...
Test Program 2, part 1

#define N 64
#define npairs 16
#define allocsPerIter 1024
const long iters = 9*1024*1024/allocsPerIter;
#define BufSize 10240

typedef struct buffer {
    int *buf[BufSize];
    unsigned int nextin;
    unsigned int nextout;
    sem_t empty;
    sem_t occupied;
    pthread_t pthread;
    pthread_t cthread;
} buffer_t;

This program creates pairs of threads: one thread allocates storage, the other deallocates storage. They communicate using producer-consumer communication.
The main routine creates \textit{n pairs} (16) of communicating pairs of threads.
To reduce the number of calls to `sem_wait` and `sem_post`, at each iteration the thread calls `malloc` `allocsPerIter` (1024) times.
void *cons(void *arg) {
    long i, j;
    buffer_t *b = (buffer_t *)arg;
    for (i = 0; i < iters; i++) {
        sem_wait(&b->occupied);
        for (j = 0; j < allocsPerIter; j++) {
            free(b->buf[b->nextout]);
            if (++b->nextout >= BufSize)
                b->nextout = 0;
        }
        sem_post(&b->empty);
    }
    return 0;
}
The code was run on a SunLab machine (an Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz).

```
$ time ./ptalloc2
real  0m1.087s
user  0m3.744s
sys   0m0.204s
$ time ./talloc2
real  0m3.535s
user  0m11.361s
sys   0m2.112s
```
What's Going On?

$ strace -c -f ./ptalloc2
...
% time seconds usecs/call calls errors syscall
------- ----------- ---------- ---------- ---------- ---------------
 94.96   2.347314   44  53653  14030    futex
...
$ strace -c -f ./talloc2
...
% time seconds usecs/call calls errors syscall
------- ----------- ---------- ---------- ---------- ---------------
 93.86   6.604632   36  185731  45222    futex
...
Running it Today ...

```
sphere $ time ./ptalloc

real    0m2.373s
user    0m9.152s
sys     0m0.008s
sphere $ time ./talloc

real    0m4.868s
user    0m19.444s
sys     0m0.020s
```
Running it Today ...

kui $ time ./ptalloc

real  0m2.787s
user  0m11.045s
sys   0m0.004s
kui $ time ./talloc

real  0m1.701s
user  0m6.584s
sys   0m0.004s
Running it Today ...

cslab0a $ time ./ptalloc

real    0m2.234s
user    0m8.468s
sys     0m0.000s

cslab0a $ time ./talloc

real    0m4.938s
user    0m19.584s
sys     0m0.000s
What’s Going On?

- **On kui:**
  - libtcmalloc.so -> libtcmalloc.so.4.1.0

- **On other machines:**
  - libtcmalloc.so -> libtcmalloc.so.4.2.2
However ...

cslab0a $ time ./ptalloc2

real  0m0.466s
user  0m1.504s
sys   0m0.212s
cslab0a $ time ./talloc2

real  0m1.516s
user  0m5.212s
sys   0m0.328s
You’ll Soon Finish CS 33 …

• You might
  – celebrate
  
  – take another systems course
    » 32
    » 138
    » 166
    » 167

  – become a 33 TA

CS33 Intro to Computer Systems XXXVII–31 Copyright © 2015 Thomas W. Doeppner. All rights reserved.
Systems Courses Next Semester

- **CS 32**
  - you’ve mastered low-level systems programming
  - now do things at a higher level
  - learn software-engineering techniques using Java, XML, etc.

- **CS 138**
  - you now know how things work on one computer
  - what if you’ve got lots of computers?
  - some may have crashed, others may have been taken over by your worst (and brightest) enemy

- **CS 166**
  - liked buffer?
  - you’ll really like 166

- **CS 167/169**
  - still mystified about what the OS does?
  - write your own!
The End

Well, not quite …
Database is due on 12/16.
We’ll do our best to get everything else back this week.
Happy coding and happy holidays!