CS 33

Multithreaded Programming V

The source code used in this lecture is available on the course web page.

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

Memory Allocation

- Multiple threads
- One heap

Bottleneck?

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

- 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?

Solution 3

- 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

Malloc/Free Implementations

- ptmalloc
  - based on solution 3
  - in glibc (i.e., used by default)
- tcmalloc
  - based on solution 2
  - 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
```

Running It …

```
$ time ./ptalloc
real    0m5.142s
user    0m20.501s
sys     0m0.024s
$ time ./tcalloc
real    0m1.889s
user    0m7.492s
sys     0m0.008s
```


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 ./tcalloc
 …
% 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 = 8*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.

Test Program 2, part 2

```
int main() {
  long i;
  buffer_t b[npairs];
  for (i=0; i<npairs; i++) {
    b[i].nextin = 0;
    b[i].nextout = 0;
    sem_init(&b[i].empty, 0, BufSize/allocsPerIter);
    sem_init(&b[i].occupied, 0, 0);
    pthread_create(&b[i].pthread, 0, prod, &b[i]);
    pthread_create(&b[i].cthread, 0, cons, &b[i]);
  }
  for (i=0; i<npairs; i++) {
    pthread_join(b[i].pthread, 0);
    pthread_join(b[i].cthread, 0);
  }
  return 0;
}
```


The main routine creates *npairs* (16) of communicating pairs of threads.

Test Program 2, part 3

```
void *prod(void *arg) {
  long i, j;
  buffer_t *b = (buffer_t *)arg;
  for (i = 0; i<iters; i++) {
    sem_wait(&b->empty);
    for (j = 0; j<allocsPerIter; j++) {
      b->buf[b->nextin] = malloc(sizeof(int)*((j%N)+1));
      if (++b->nextin >= BufSize)
        b->nextin = 0;
    }
    sem_post(&b->occupied);
  }
  return 0;
}
```


To reduce the number of calls to *sem_wait* and *sem_post*, at each iteration the thread calls *new allocsPerIter* (1024) times.

Test Program 2, part 4

```
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;
}
```

Running It …

```
$ time ./ptalloc2
real    0m1.087s
user    0m3.744s
sys     0m0.204s
$ time ./tcalloc2
real    0m3.535s
user    0m11.361s
sys     0m2.112s
```


The code was run on a SunLab machine (an Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz).

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 ./tcalloc2
 …
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 93.86    6.604632          36    185731     45222 futex
 …
```