

# CS 33

## Multithreaded Programming VII

# Implementing Mutexes

- **Strategy**
  - make the usual case (no waiting) very fast
  - can afford to take more time for the other case (waiting for the mutex)

# Futexes

- **Safe, *efficient* kernel conditional queueing in Linux**
- **All operations performed atomically**
  - `futex_wait(futex_t *futex, int val)`
    - » **if `futex->val` is equal to `val`, then sleep**
    - » **otherwise return**
  - `futex_wake(futex_t *futex)`
    - » **wake up one thread from `futex`'s wait queue, if there are any waiting threads**

# Ancillary Functions

- `int atomic_inc(int *val)`
  - **add 1 to** `*val`, **return its original value**
- `int atomic_dec(int *val)`
  - **subtract 1 from** `*val`, **return its original value**
- `int CAS(int *ptr, int old, int new) {`  
    `int tmp = *ptr;`  
    `if (*ptr == old)`  
        `*ptr = new;`  
    `return tmp;`  
}

# Attempt 1

```
void lock(futex_t *futex) {  
    int c;  
    while ((c = atomic_inc(&futex->val)) != 0)  
        futex_wait(futex, c+1);  
}
```

```
void unlock(futex_t *futex) {  
    futex->val = 0;  
    futex_wake(futex);  
}
```

# Quiz 1

```
void lock(futex_t *futex) {  
    int c;  
    while ((c = atomic_inc(&futex->val)) != 0)  
        futex_wait(futex, c+1);  
}
```

```
void unlock(futex_t *futex) {  
    futex->val = 0;  
    futex_wake(futex);  
}
```

**Why doesn't Attempt 1 work?**

- a) unlock fails to wake up a sleeping thread in certain circumstances
- b) the while loop in lock doesn't terminate in certain circumstances
- c) both of the above
- d) none of the above

# Attempt 2

```
void lock(futex_t *futex) {
    int c;
    if ((c = CAS(&futex->val, 0, 1) != 0)
        do {
            if (c == 2 || (CAS(&futex->val, 1, 2) != 0))
                futex_wait(futex, 2);
            while ((c = CAS(&futex->val, 0, 2)) != 0))
        }

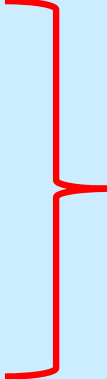
void unlock(futex_t *futex) {
    if (atomic_dec(&futex->val) != 1) {
        futex->val = 0;
        futex_wake(futex);
    }
}
```

## Quiz 2

**Does it work?**

- a) always
- b) except for pathological cases
- c) rarely
- d) never

# Memory Allocation

- Multiple threads
  - One heap
- 
- Bottleneck?**



# **Solution 0:**

**Use your malloc implementation but use mutexes to make it thread-safe**

**1) Use a single mutex to protect the heap**

- no concurrent access

**2) Use a mutex per block**

- concurrent access to the heap

# Quiz 3

**Solution 0.2 is not used because**

- a) Since the free list is circular, deadlock cannot be avoided**
- b) Since each core accesses memory via its private L1 cache, memory blocks used by one thread cannot be safely shared with others**
- c) There will be too many calls to lock and unlock mutexes, slowly things down a lot**
- d) Since there must be a mutex per block, too much memory is wasted**
- e) Something else**

# Not a Quiz

**How can it be done better?**

# 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?**
- **What if one thread frees memory malloc'd by another?**

# 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 only when needed
  - 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 frees memory malloc'd by another?**

# 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;
}
```

---



# Not a Quiz

**Which is fastest?**

- a) glibc (i.e., standard Linux)**
- b) Google**

# Compiling It ...

```
% gcc -o ptalloc alloc.c -lpthread
```

```
% gcc -o tcalloc alloc.c -lpthread -ltcmalloc
```

# Running It (2014) ...

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

# Running It (2024) ...

```
$ time ./ptalloc
real    0m0.558s
user    0m6.141s
sys     0m0.020s
$ time ./tcalloc
real    0m0.400s
user    0m4.458s
sys     0m0.008s
```

# What's Going On (2014)?

```
$ 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

```
...
```

# What's Going On (2024)?

```
$ strace -c -f ./ptalloc
```

```
...
```

% time	seconds	usecs/call	calls	errors	syscall
--------	---------	------------	-------	--------	---------

-----	-----	-----	-----	-----	-----
-------	-------	-------	-------	-------	-------

```
...
```

0.00	0.000000	0	1		futex
------	----------	---	---	--	-------

```
...
```

```
$ strace -c -f ./tcalloc
```

```
...
```

% time	seconds	usecs/call	calls	errors	syscall
--------	---------	------------	-------	--------	---------

-----	-----	-----	-----	-----	-----
-------	-------	-------	-------	-------	-------

```
...
```

0.38	0.000016	1	10		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;
```

# 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;
}
```



# 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;
}
```

# 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 (2014) ...

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

# Running It (2024) ...

```
$ time ./ptalloc2
real    0m1.594s
user    0m8.778s
sys     0m2.551s
$ time ./tcalloc2
real    0m7.089s
user    0m59.871s
sys     0m11.220s
```

# What's Going On (2014)?

```
$ strace -c -f ./ptalloc2
```

```
...
```

% time	seconds	usecs/call	calls	errors	syscall
--------	---------	------------	-------	--------	---------

```
...
```

```
-----
```

```
...
```

93.04	8.246196	117	70173	20775	futex
-------	----------	-----	-------	-------	-------

```
...
```

```
$ strace -c -f ./tcalloc2
```

```
...
```

% time	seconds	usecs/call	calls	errors	syscall
--------	---------	------------	-------	--------	---------

```
...
```

```
-----
```

99.92	47.796676	153	311012	7244	futex
-------	-----------	-----	--------	------	-------

```
...
```

# What's Going On (2024)?

```
$ strace -c -f ./ptalloc2
```

```
...
```

% time	seconds	usecs/call	calls	errors	syscall
--------	---------	------------	-------	--------	---------

-----	-----	-----	-----	-----	-----
-------	-------	-------	-------	-------	-------

98.55	55.917331	138	403494	108889	futex
-------	-----------	-----	--------	--------	-------

```
...
```

```
$ strace -c -f ./tcalloc2
```

```
...
```

% time	seconds	usecs/call	calls	errors	syscall
--------	---------	------------	-------	--------	---------

-----	-----	-----	-----	-----	-----
-------	-------	-------	-------	-------	-------

99.98	298.581838	149	2002633	22522	futex
-------	------------	-----	---------	-------	-------

```
...
```