Alternatives to Mutexes: Atomic Instructions

• Read-modify-write performed atomically
• *Lock prefix* may be used with certain IA32 and x86-64 instructions to make this happen
  – lock incr x
  – lock add %2, x
• It’s expensive
• It’s not portable
  – no POSIX-threads way of doing it
  – Windows supports
    » InterlockedIncrement
    » InterlockedDecrement
Alternatives to Mutexes: Spin Locks

• Consider

```c
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
Compare and Exchange

cmpxchg src, dest

- compare contents of %eax with contents of dest
  » if equal, then dest = src (and ZF = 1)
  » otherwise %eax = dest (and ZF = 0)
Spin Lock

• the spin lock is pointed to by the first arg (%edi)
  – locked is 1, unlocked is 0

.text
.globl slock, sunlock
slock:
loop:
  movl $0, %eax
  movq $1, %r10
  lock cmpxchg %r10, 0(%edi)
  jne loop
  ret
sunlock:
  movl $0, 0(%edi)
  ret
Improved Spin Lock

.text
.globl slock, sunlock

slock:
loop:
cmp $0, 0(%edi)  # compare using normal instructions
jne loop
movl $0, %eax
movq $1, %r10
lock cmpxchg %r10, 0(%edi)  # verify w/ cmpxchg
jne loop
ret

sunlock:
movl $0, 0(%edi)
ret
Yet More From POSIX …

```c
int pthread_spin_init(pthread_spin_t *s,
                 int pshared);
int pthread_spin_destroy(pthread_spin_t *s);
int pthread_spin_lock(pthread_spin_t *s);
int pthread_spin_trylock(pthread_spin_t *s);
int pthread_spin_unlock(pthread_spin_t *s);
```
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
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?
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

\[
\begin{align*}
\text{const unsigned int } & \quad N=64, \text{ nthreads}=32, \text{ iters}=10000000; \\
\text{int } & \quad \text{main() } \\
& \quad \text{void } * \text{tfunc(\text{void } *);} \\
\text{pthread_t } & \quad \text{thread[nthreads];}
\end{align*}
\]

\[
\begin{align*}
\text{for (int } & \quad i=0; \quad i<\text{nthreads}; \quad i++) \\
& \quad \text{pthread_create(\&thread}[i], 0, \text{tfunc, (\text{void } *)i);} \\
& \quad \text{pthread_detach(thread}[i]);
\end{align*}
\]

\[
\begin{align*}
\text{pthread_exit(0);} \\
\text{)}
\end{align*}
\]

\[
\begin{align*}
\text{void } * & \quad \text{tfunc(\text{void } *\text{arg}) } \\
& \quad \text{long } i;
\end{align*}
\]

\[
\begin{align*}
\text{for (i=0; } & \quad i<\text{iters}; \quad i++) \\
& \quad \text{long } *p = (\text{long } *) \text{malloc(sizeof(long)*((i}\%\text{N}+1))}; \\
& \quad \text{free(p);} \\
& \quad \text{return 0;}
\end{align*}
\]
Compiling It …

% gcc -o ptalloc alloc.cc -lpthread
% gcc -o tcalloc alloc.cc -lpthread -ltc_malloc
Running It ...

$ time ./ptalloc
real  0m5.142s
user  0m20.501s
sys   0m0.024s

$ time ./tcalloc
real  0m1.889s
user  0m7.492s
sys   0m0.008s
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;
Test Program 2, part 2

```c
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

```c
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

```c
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
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
...
Running it Today ...

sphere $ time ./ptalloc

real    0m2.373s
user    0m9.152s
sys     0m0.008s

sphere $ time ./tcalloc

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 ./tcallloc
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 ./tcallloc

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 ./tcalloc2

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
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!