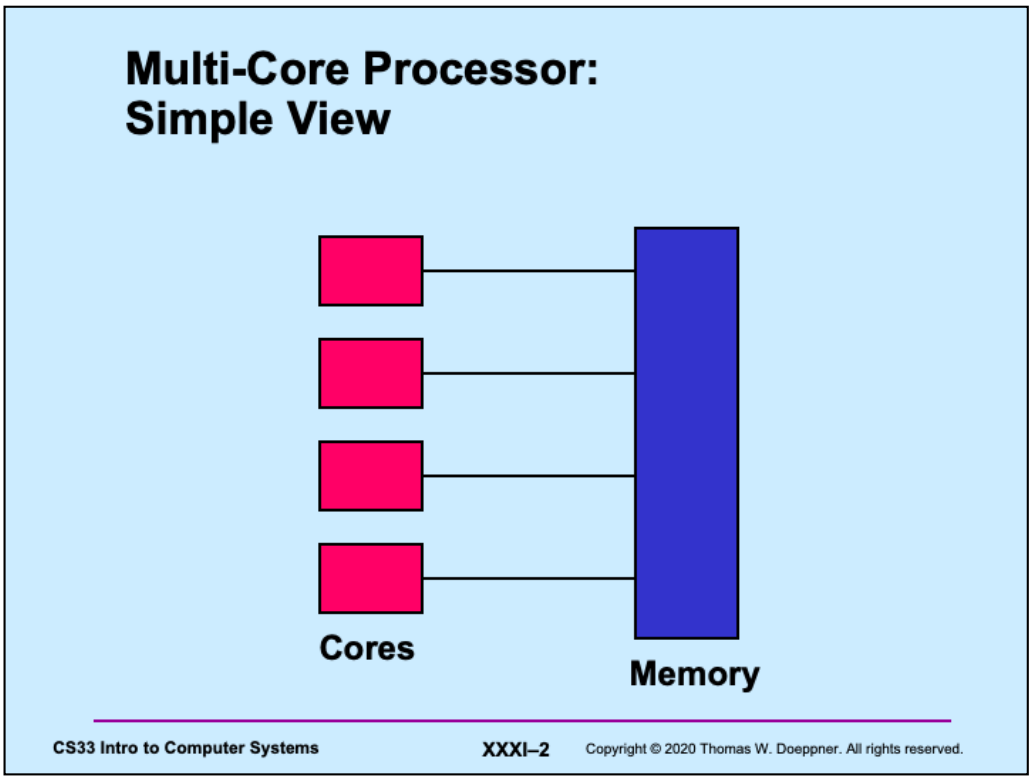
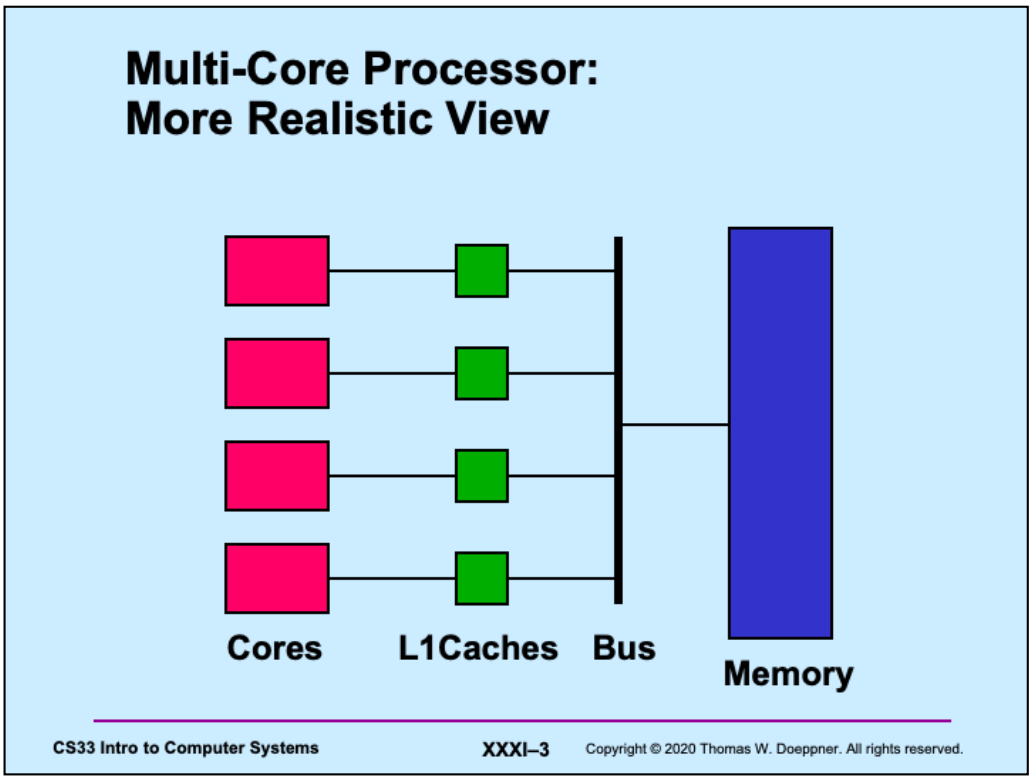
CS 33

Multithreaded Programming VI



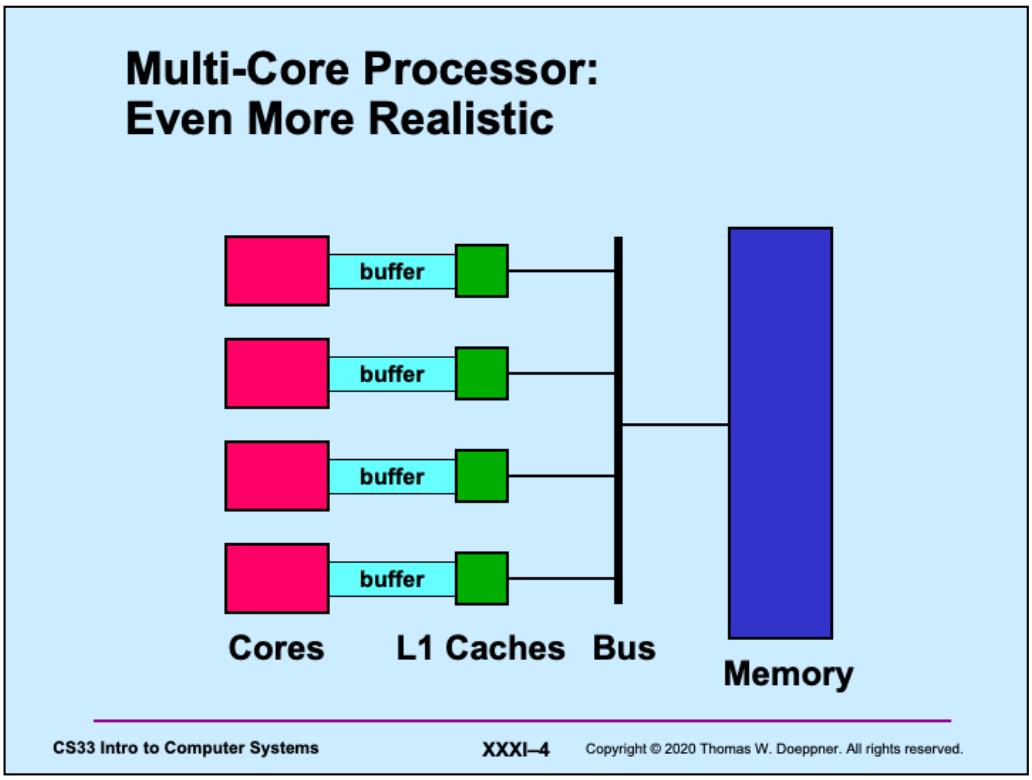
This slide illustrates the common view of the architecture of a multi-core processor: a number of processors are all directly connected to the same memory (which they share). If one core (or processor) stores into a storage location and immediately thereafter another core loads from the same storage location, the second core loads exactly what the first core stored.

Unfortunately, as we learned earlier in the course, things are not quite so simple.



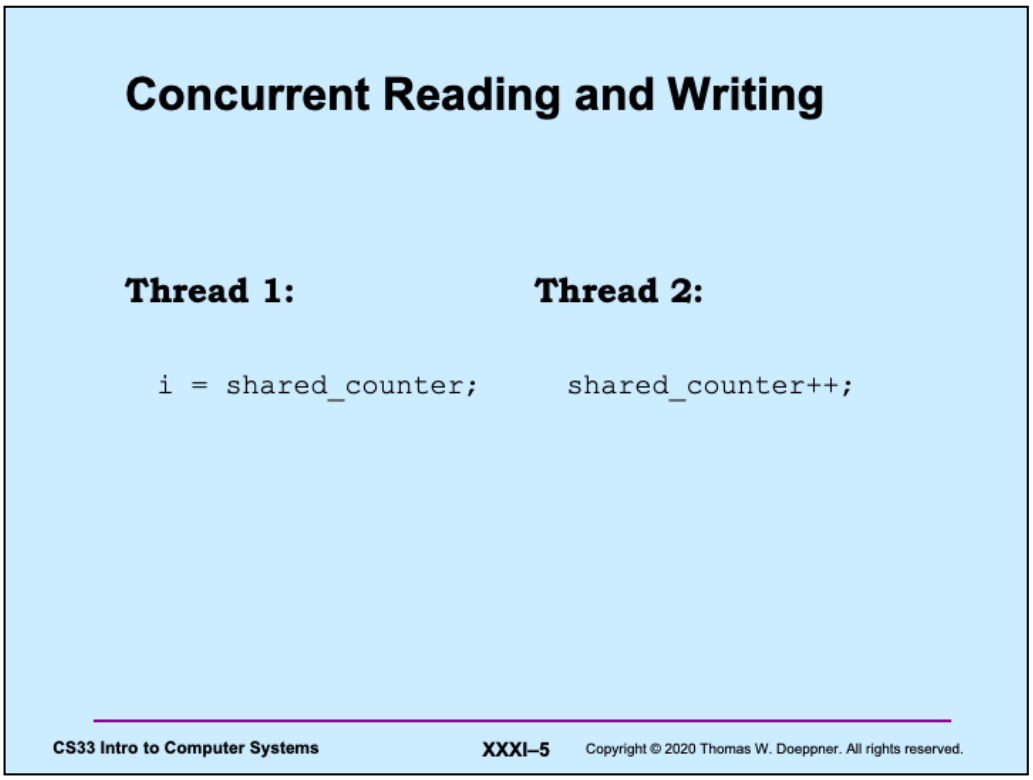
Real multi-core processors have L1 caches that sit between each core and the memory bus; there is a single connection between the bus and the memory. When a core issues a store, the store affects the L1 cache. When a core issues a load, the load is dealt with by the L1 cache if possible, and otherwise goes to memory (perhaps via a shared L2 cache). Most architectures have some sort of cache-consistency logic to insure that the shared-memory semantics of the previous page are preserved.

However, again as we learned earlier in the course, even this description is too simplistic.



This slide shows an even more realistic model, pretty much the same as what we saw is actually used in recent Intel processors. Between each core and the L1 cache is a buffer. Stores by a core go into the buffer. Sometime later the effect of the store reaches the L1 cache. In the meantime, the core is issuing further instructions. Loads by the core are handled from the buffer if the data is still there; otherwise they go to the L1 cache, and then perhaps to memory.

In all instances of this model the effect of a store, as seen by other cores, is delayed. In some instances of this model the order of stores made by one core might be perceived differently by other cores. Architectures with the former property are said to have *delayed stores*; architectures with the latter are said to have *reordered stores* (an architecture could well have both properties).



In this example, one thread running on one processor is loading from an integer in storage; another thread running on another processor is loading from and then storing into an integer in storage. Can this be done safely without explicit synchronization?

On most architectures, the answer is yes. If the integer in question is aligned on a natural (e.g., eight-byte) boundary, then the hardware (perhaps the cache) insures that loads and stores of the integer are atomic.

However, one cannot assume that this is the case on all architectures. Thus a portable program must use explicit synchronization (e.g., a mutex) in this situation.

Mutual Exclusion w/o Mutexes

```
void peterson(long me) {
  static long loser;                // shared
  static long active[2] = {0, 0};   // shared
  long other = 1 - me;              // private
  active[me] = 1;
  loser = me;
  while (loser == me && active[other])
    ;
  // critical section
  active[me] = 0;
}
```


Shown on the slide is Peterson's algorithm for handling mutual exclusion for two threads without explicit synchronization. (The *me* argument for one thread is 0 and for the other is 1.) This program works given the first two shared-memory models. Does it work with delayed-store architectures?

The algorithm is from "Myths About the Mutual Exclusion Problem," by G. L. Peterson, Information Processing Letters 12(3) 1981: 115–116.

Busy-Waiting Producer/Consumer

```
void producer(char item) {
  while(in - out == BSIZE)
   ;
  buf[in%BSIZE] = item;
  in++;
}
```

```
char consumer( ) {
  char item;
  while(in - out == 0)
   ;
  item = buf[out%BSIZE];
  out++;
  return(item);
}
```


This example is a solution, employing "busy waiting," to the producer-consumer problem for one consumer and one producer.

This solution to the producer-consumer problem is from "Proving the Correctness of Multiprocess Programs," by L. Lamport, IEEE Transactions on Software Engineering, SE-3(2) 1977: 125-143.

Quiz 1

```
void producer(char item) {

  while(in - out == BSIZE)
    ;

  buf[in%BSIZE] = item;

  in++;
}
```

```
char consumer( ) {
  char item;
  while(in - out == 0)
    ;

  item = buf[out%BSIZE];

  out++;

  return(item);
}
```

This works on sunlab machines.

a) true

b) false

Coping

- Don't rely on shared memory for synchronization
- Use the synchronization primitives

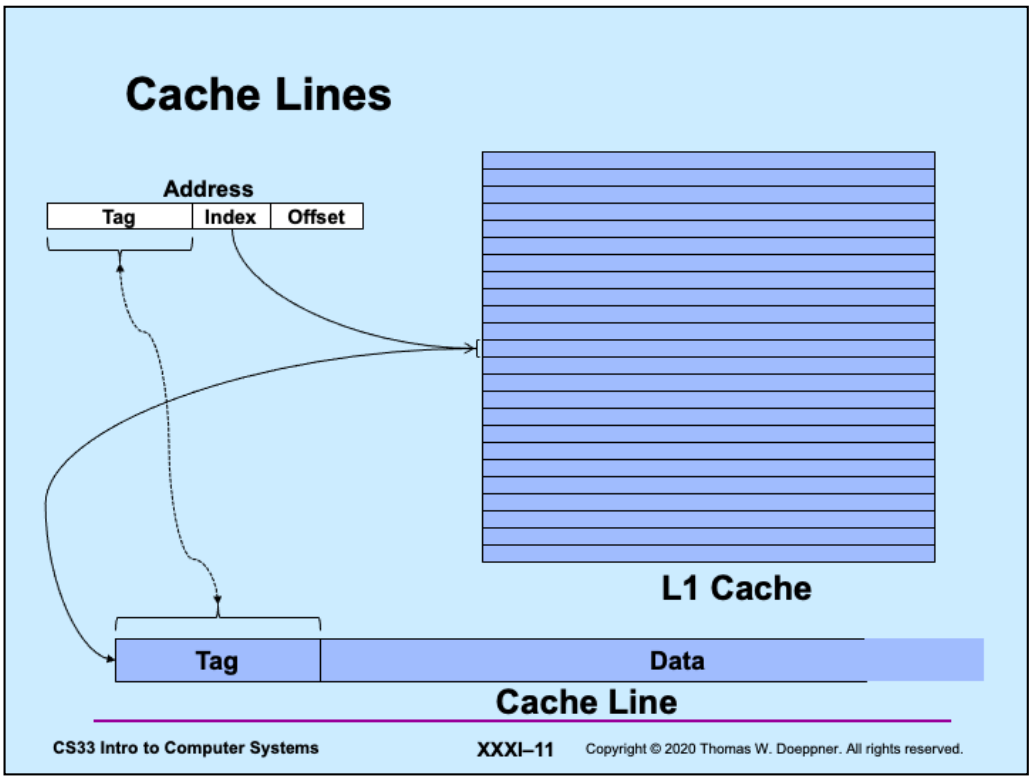
The point of the previous several slides is that one cannot rely on expected properties of shared memory to eliminate explicit synchronization. Shared memory can behave in some very unexpected ways. However, it is the responsibility of the implementers of the various synchronization primitives to make certain not only that they behave correctly, but also that they synchronize memory with respect to other threads.

Which Runs Faster?

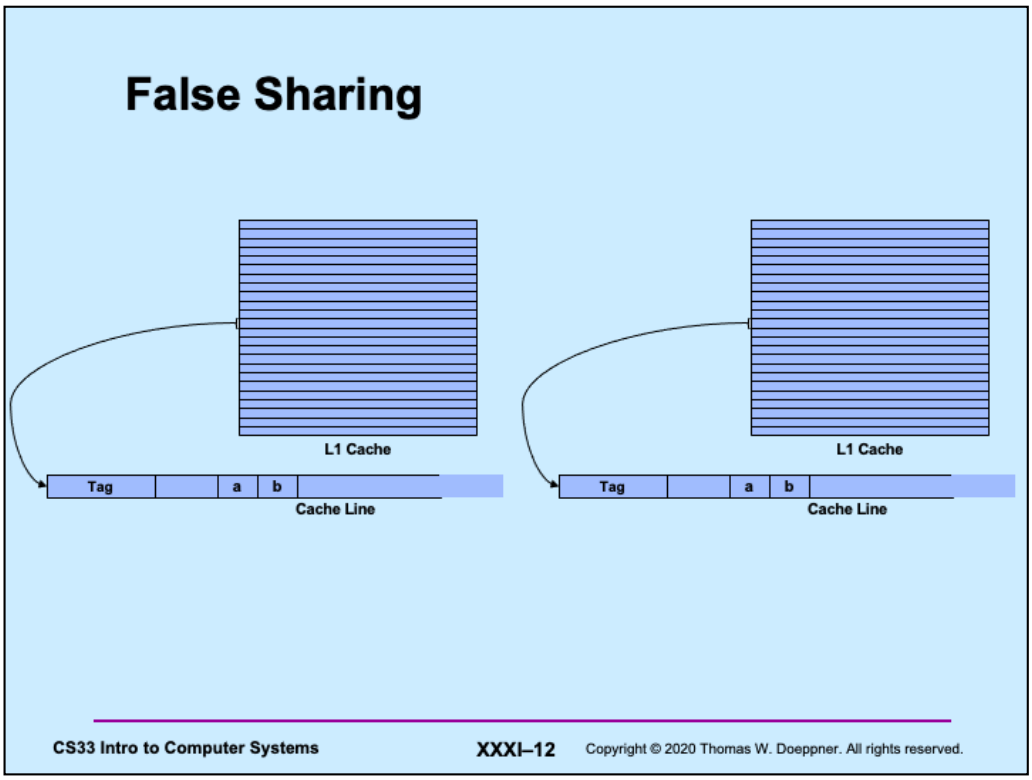
```
volatile int a, b;

void *thread1(void *arg) {
  int i;
  for (i=0; i<reps; i++) {
    a = 1;
  }
}

void *thread2(void *arg) {
  int i;
  for (i=0; i<reps; i++) {
    b = 1;
  }
}
```

```
volatile int a,
  padding[128], b;

void *thread1(void *arg) {
  int i;
  for (i=0; i<reps; i++) {
    a = 1;
  }
}

void *thread2(void *arg) {
  int i;
  for (i=0; i<reps; i++) {
    b = 1;
  }
}
```


Assume these are run on a two-processor system: why does the two-threaded program on the right run faster than the two-threaded program on the left?



Processors usually employ data caches that are organized as a set of cache lines, typically of 64 bytes in length. Thus data is fetched from and stored to memory in units of the cache-line size. Each processor has its own data cache.

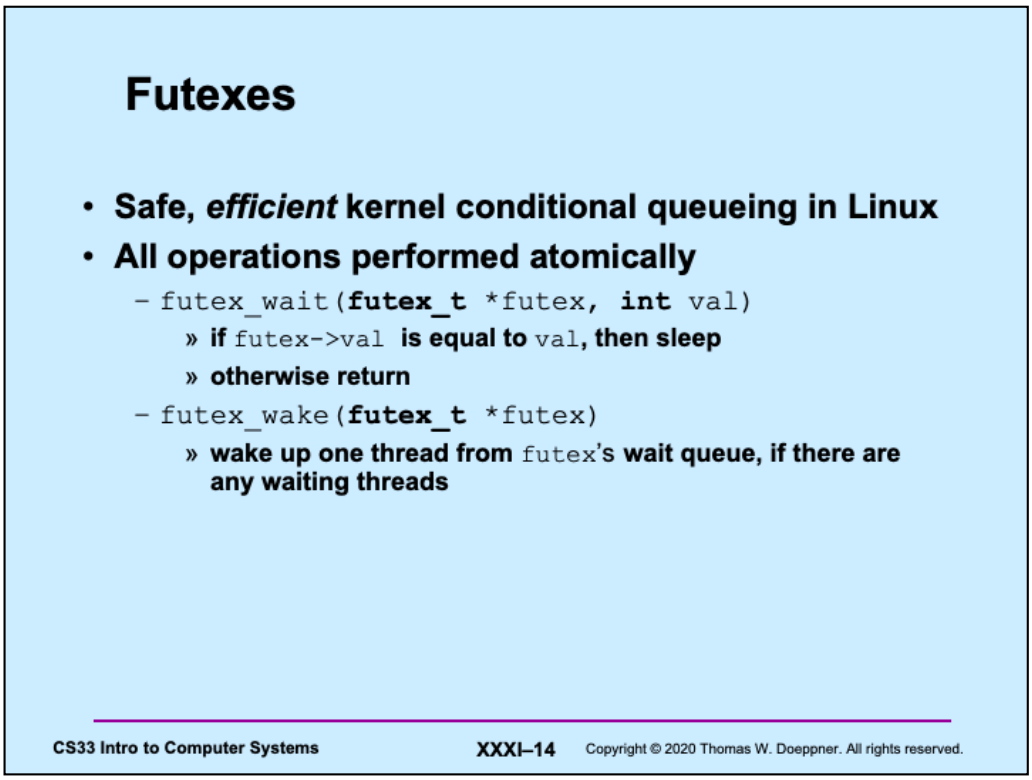


Getting back to our example: we have a two-processor system, and thus two data (L1) caches. If *a* and *b* are in the same cache line, then when either processor accesses *a*, it also accesses *b*. Thus if *a* is modified on processor 1, memory coherency will cause the entire cache line to be invalidated on processor 2. Thus when processor 2 attempts to access *b*, it will get a cache miss and be forced to go to memory to update the cache line containing *b*. From the programmer's perspective, *a* and *b* are not shared. But from the cache's perspective, they are. This phenomenon is known as *false sharing*, and is a source of performance problems.

For further information about false sharing and for tools to deal with it, see http://emeryblogger.com/2011/07/06/precise-detection-and-automatic-mitigation-of-false-sharing-oopsla-11/.

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)



For details on futexes, avoid the Linux man pages, but look at http://people.redhat.com/drepper/futex.pdf, from which this material was adapted. Note that there's actually just one *futex* system call; whether it's a *wait* or a *wakeup* is specified by an argument.

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;
}
```


These functions are available on most architectures, particularly on the x86. Note that their effect must be *atomic*: everything happens at once.

Attempt 1

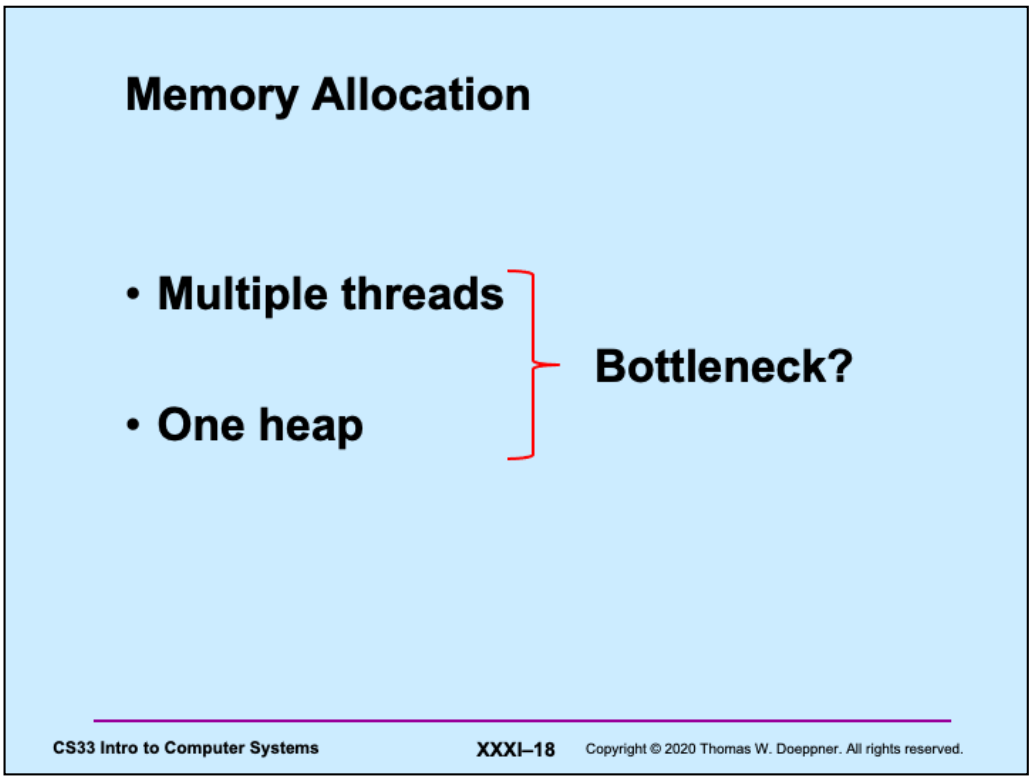
```
void lock(futex_t *futex) {
  int c;
  while ((c = atomic_inc(&futex->val)) != 0)
    futex_wait(futex, c+1);
}

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


If the futex's value is 0, it's unlocked, otherwise it's locked.

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);
  }
}
```


In this version, if the futex's value is 0, it's unlocked, if it's one it's locked and no threads are waiting for it; if it's greater than one it's locked and there might be threads waiting for it.



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
```


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
```


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
 …
```


strace is a system facility that supplies information about the system calls a process uses. The –c flag tell it to print the cumulative statistics after the process terminates. The –f flag tells it to include information on all threads and child processes.

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 function 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 malloc *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 (2014) …

```
$ 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
 …
```


Running it (2015) ...

```
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 (2015) ...

```
kui $ time ./ptalloc

real    0m2.787s
user    0m11.045s
sys     0m0.004s

kui $ time ./tcalloc

real    0m1.701s
user    0m6.584s
sys     0m0.004s
```


Running it (2015) ...

```
cslab0a $ time ./ptalloc

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

cslab0a $ time ./tcalloc

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 (2015) ...

```
cslab0a $ time ./ptalloc2

real    0m0.466s
user    0m1.504s
sys     0m0.212s

cslab0a $ time ./tcalloc2

real    0m1.516s
user    0m5.212s
sys     0m0.328s
```


It's 2020

- tcmalloc no longer exists
  - no explanation from Google, it's simply gone
- ptmalloc continues to improve