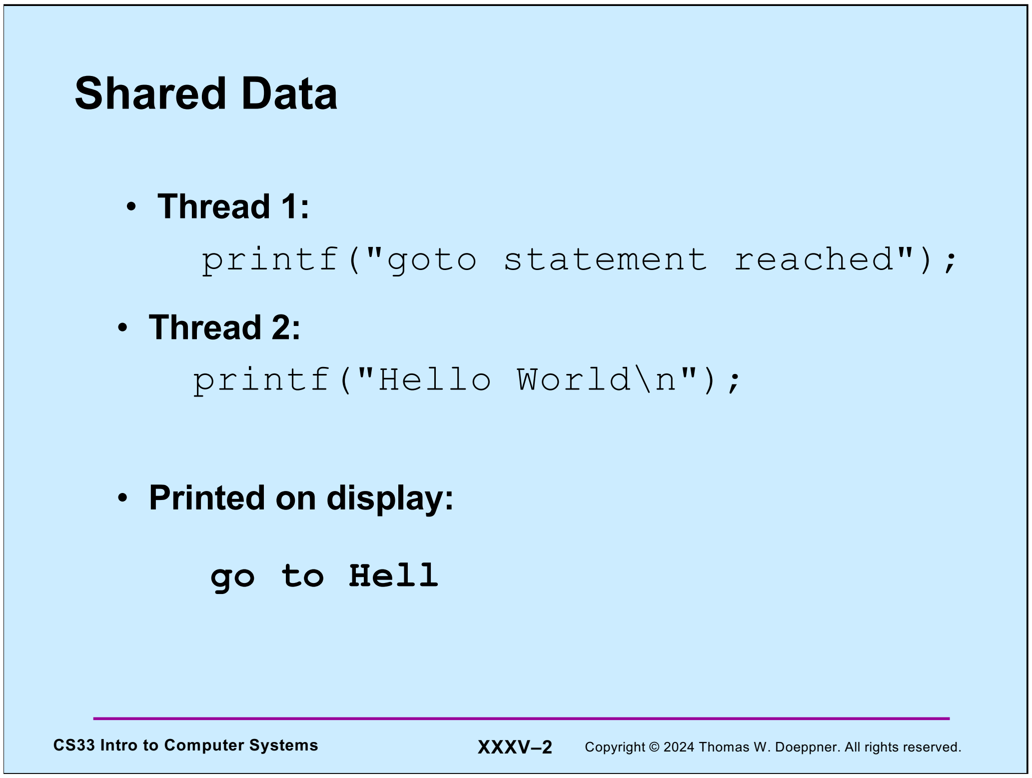
CS 33

Multithreaded Programming VI

Copyright © 2024 Thomas W. Doeppner. All rights reserved.



Shared Data

- **Thread 1:**

```
printf("goto statement reached");
```

- **Thread 2:**

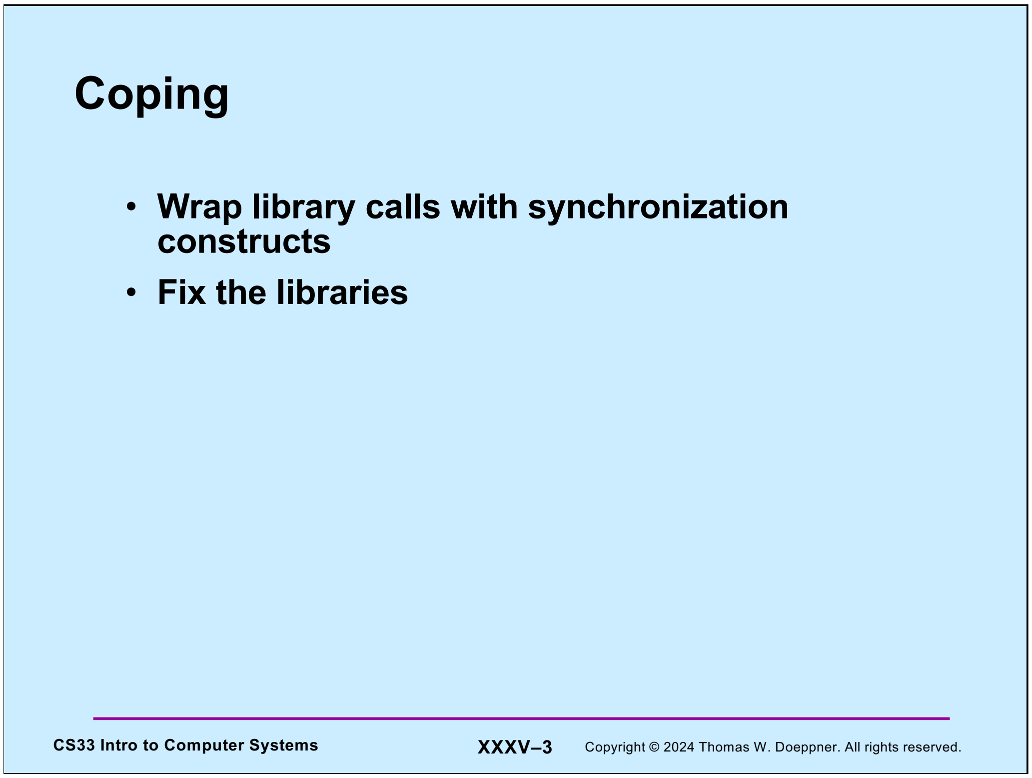
```
printf("Hello World\n");
```

- **Printed on display:**

```
go to Hell
```

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

Yet another problem that arises when using libraries that were not designed for multithreaded programs concerns synchronization. The slide shows what might happen if one relied on the single-threaded versions of the standard I/O routines.

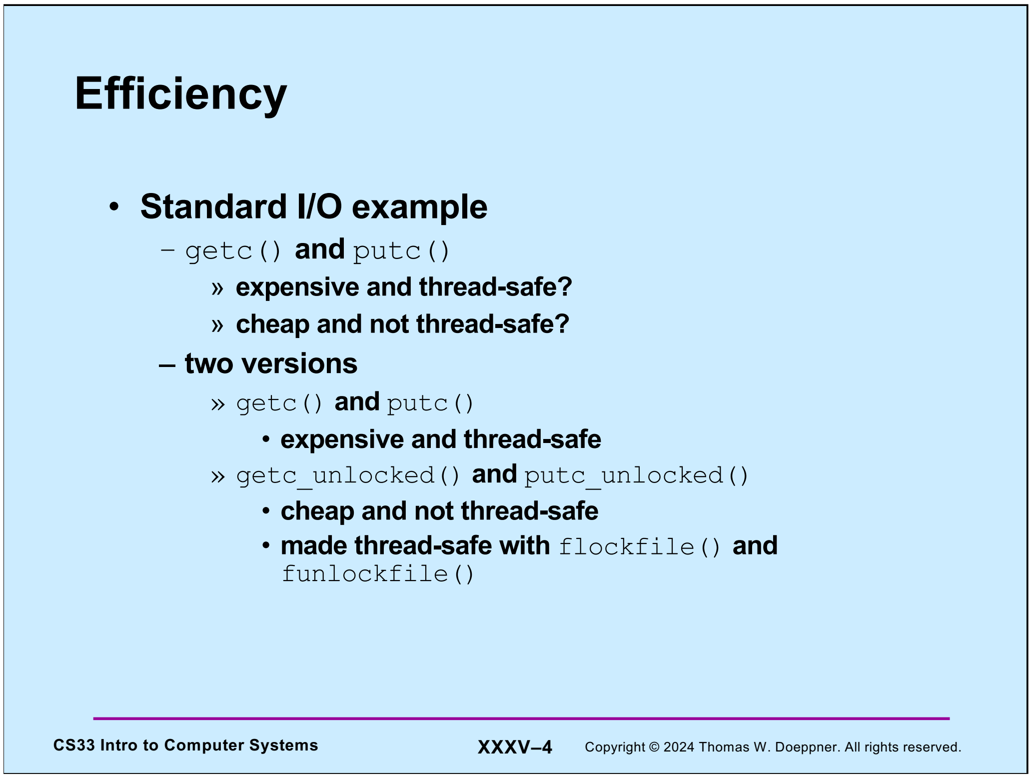


Coping

- Wrap library calls with synchronization constructs
- Fix the libraries

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

To deal with this **printf** problem, we must somehow add synchronization to **printf** (and all of the other standard I/O functions). A simple way to do this would be to supply wrappers for all of the standard I/O functions ensuring that only one thread is operating on any particular stream at a time. A better way would be to do the same sort of thing by fixing the functions themselves, rather than supplying wrappers (this is what is done in most implementations).



Efficiency

- **Standard I/O example**
  - `getc()` **and** `putc()`
    - » **expensive and thread-safe?**
    - » **cheap and not thread-safe?**
  - **two versions**
    - » `getc()` **and** `putc()`
      - **expensive and thread-safe**
    - » `getc_unlocked()` **and** `putc_unlocked()`
      - **cheap and not thread-safe**
      - **made thread-safe with** `flockfile()` **and** `funlockfile()`

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

After making a library thread-safe, we may discover that many functions have become too slow. For example, the standard-I/O functions **getc** and **putc** are expected to be fast — they are usually implemented as macros. But once we add the necessary synchronization, they become rather sluggish — much too slow to put in our innermost loops. However, if we are aware of and willing to cope with the synchronization requirements ourselves, we can produce code that is almost as efficient as the single-threaded code without synchronization requirements.

The POSIX-threads specification includes unsynchronized versions of **getc** and **putc** — **getc_unlocked** and **putc_unlocked**. These are exactly the same code as the single-threaded **getc** and **putc**. To use these new functions, one must take care to handle the synchronization oneself. This is accomplished with **flockfile** and **funlockfile**.

Efficiency

- **Naive**

```
for(i=0; i<lim; i++)
  putc(out[i]);
```

- **Efficient**

```
flockfile(stdout);
for(i=0; i<lim; i++)
  putc_unlocked(out[i]);
funlockfile(stdout);
```

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

What’s Thread-Safe?

- **Everything except**

asctime()
basename()
catgets()
crypt()
ctime()
dbm_clearerr()
dbm_close()
dbm_delete()
dbm_error()
dbm_fetch()
dbm_firstkey()
dbm_nextkey()
dbm_open()
dbm_store()
dirname()
dlerror()
drand48()
ecvt()
encrypt()
endgrent()
endpwent()
endutxent()
fcvt()
ftw()
gcvt()
getc_unlocked()
getchar_unlocked()
getdate()
getenv()
getgrent()
getgrgid()
getgrnam()
gethostbyaddr()
gethostbyname()
gethostent()
getlogin()
getnetbyaddr()
getnetbyname()
getnetent()
getopt()
getprotobyname()
getprotobynumber()
getprotoent()
getpwent()
getpwnam()
getpwuid()
getservbyname()
getservbyport()
getservent()
getutxent()
getutxid()
getutxline()
gmtime()
hcreate()
hdestroy()
hsearch()
inet_ntoa()
l64a()
lgamma()
lgammaf()
lgammal()
localeconv()
localtime()
lrand48()
mrand48()
nftw()
nl_langinfo()
ptsname()
putc_unlocked()
putchar_unlocked()
putenv()
pututxline()
rand()
readdir()
setenv()
setgrent()
setkey()
setpwent()
setutxent()
strerror()
strtok()
ttyname()
unsetenv()
wcstombs()
wctomb()

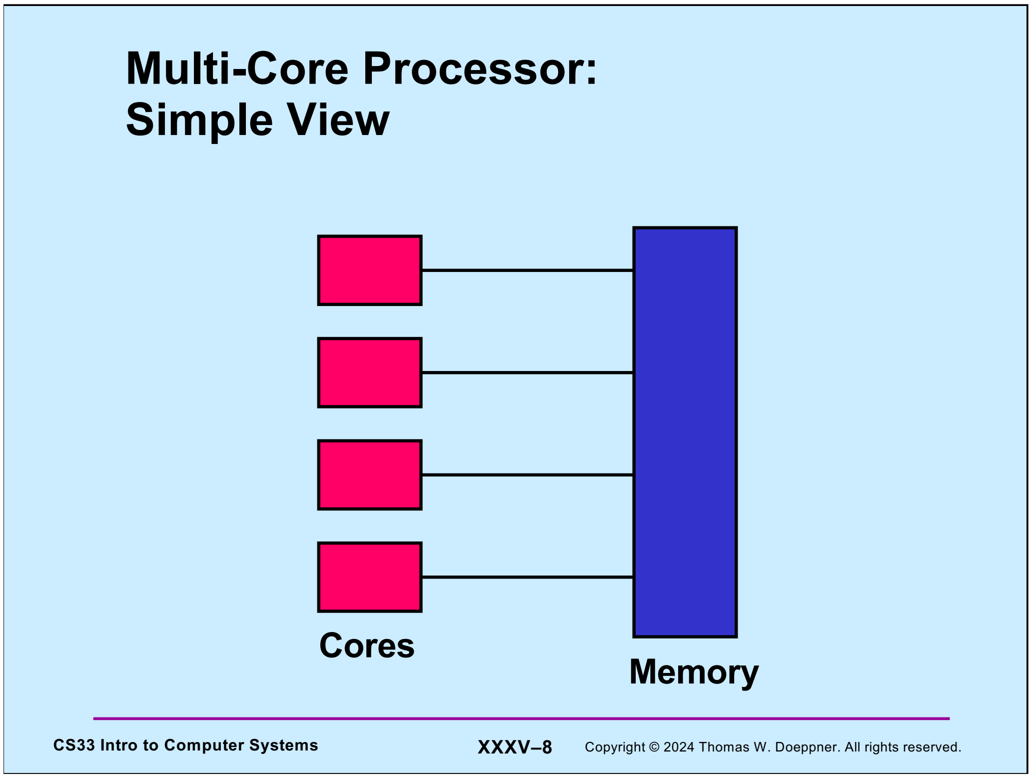
Copyright © 2024 Thomas W. Doeppner. All rights reserved.

According to IEEE Std. 1003.1 (POSIX), all functions it specifies must be thread-safe, except for those listed above.

Concurrency

- Real
  - many things happen at once
  - multiple threads running on multiple cores
- Simulated
  - things appear to happen at once
  - a single core is multiplexed among multiple threads
    - time slicing

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

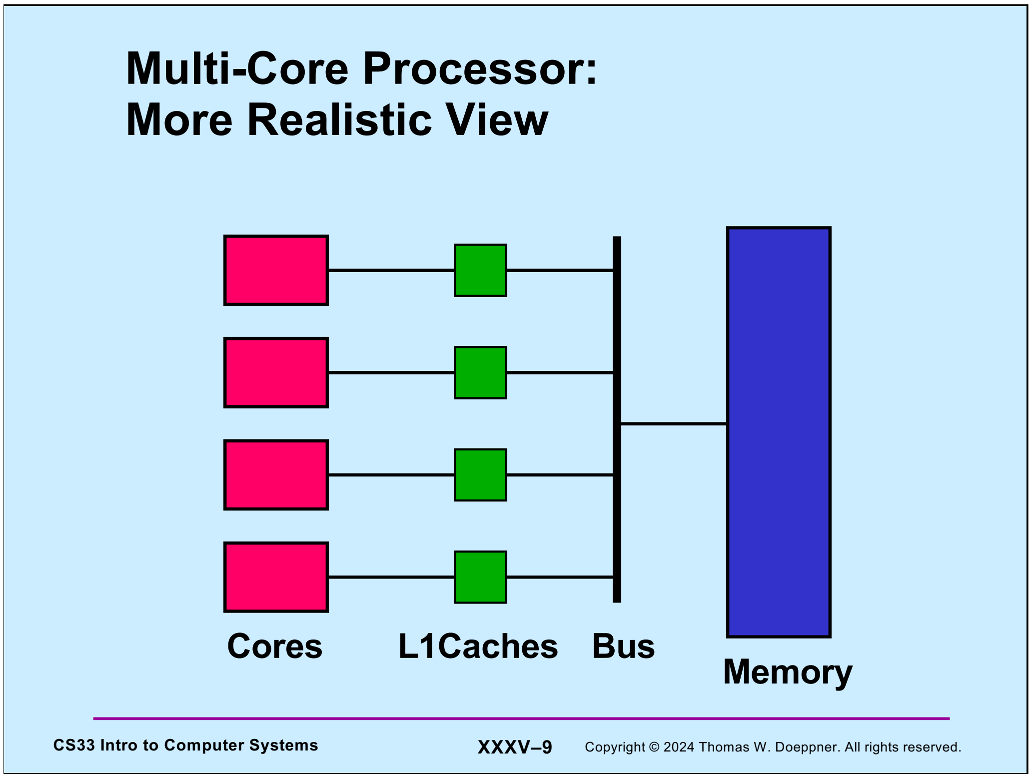


Multi-Core Processor: Simple View

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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.



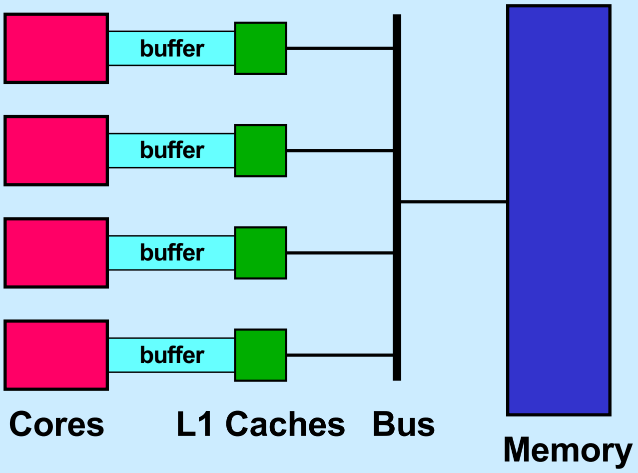
Multi-Core Processor: More Realistic View

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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 ensure 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.

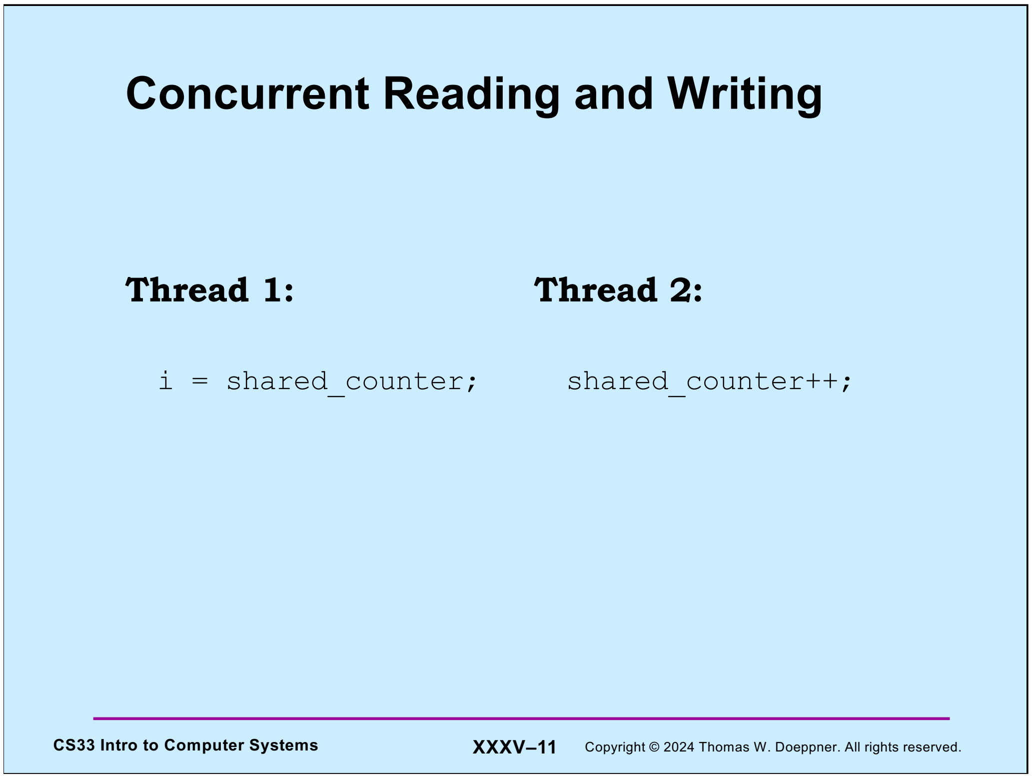
Multi-Core Processor:
Even More Realistic



Copyright © 2024 Thomas W. Doeppner. All rights reserved.

This slide shows an even more realistic model, pretty much the same as what we saw is actually used in recent 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).



Concurrent Reading and Writing

Thread 1:

```
i = shared_counter;
```

Thread 2:

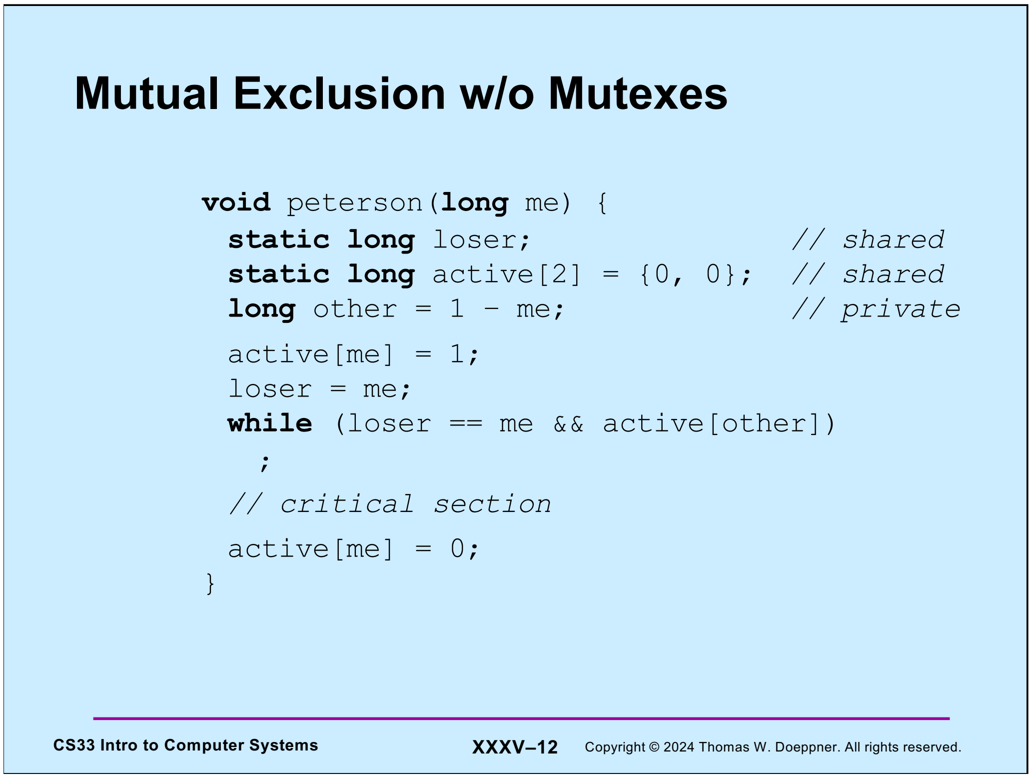
```
shared_counter++;
```

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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? (If it's done safely, then the value stored by thread 1 into I is either the value of shared_counter before it's incremented or its value afterwards).

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. If loads and stores are not atomic, it might be the case that the first four bytes of shared_counter are read, then the next four bytes.

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.



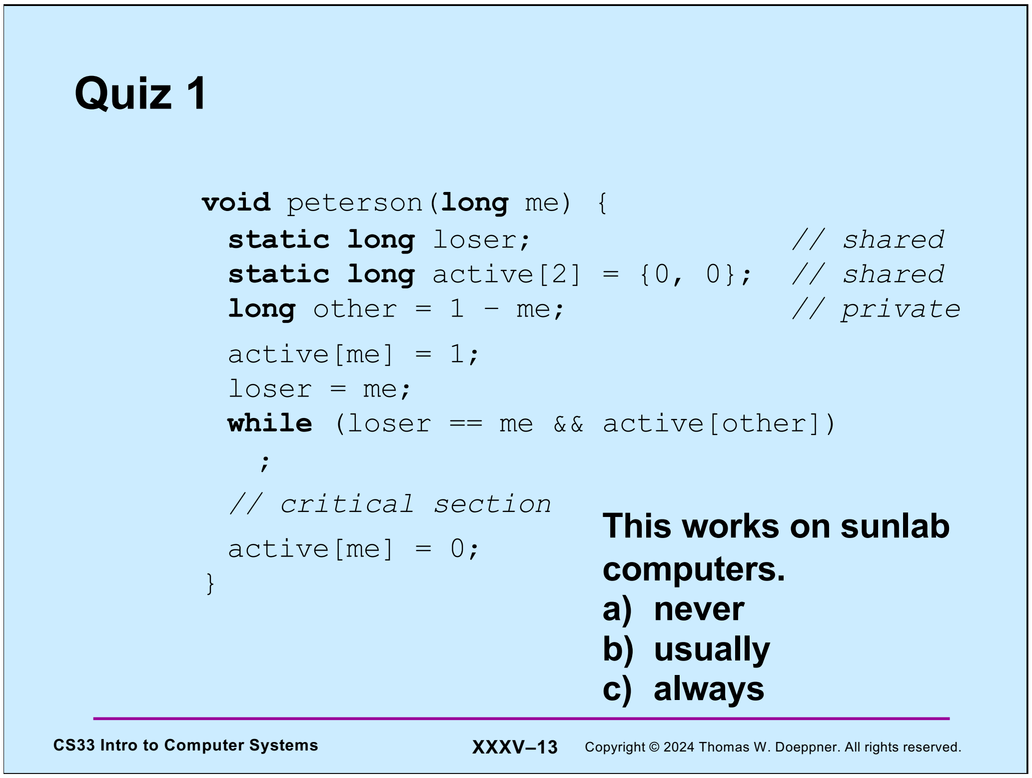
Mutual Exclusion w/o Mutexes

```
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;
}
```

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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.



Quiz 1

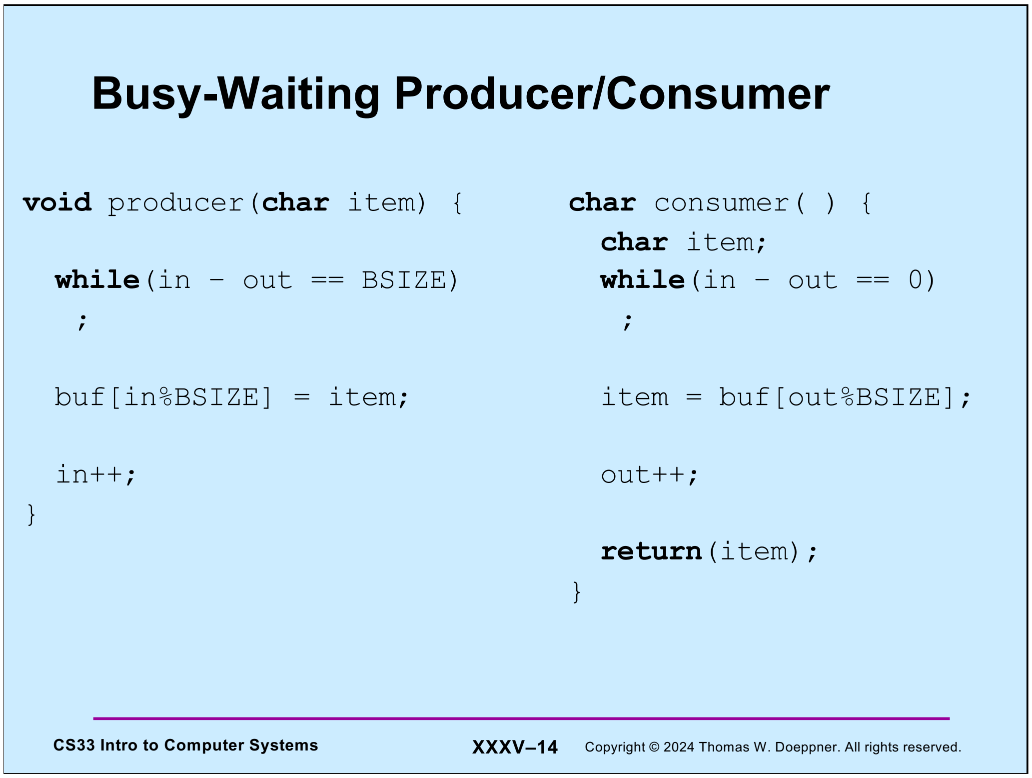
```
void peterson(long me) {
  static long loser;                  // shared
  static long active[2] = {0, 0};    // shared
  long other = 1 - me;                // private
  active[me] = 1;
  loser = me;
  while (loser == me && active[other])
    ;
  // critical section
  active[me] = 0;
}
```

This works on sunlab computers.

- **a) never**
- **b) usually**
- **c) always**

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

Sunlab computers (as do most modern computers) employ the delayed-store architecture.



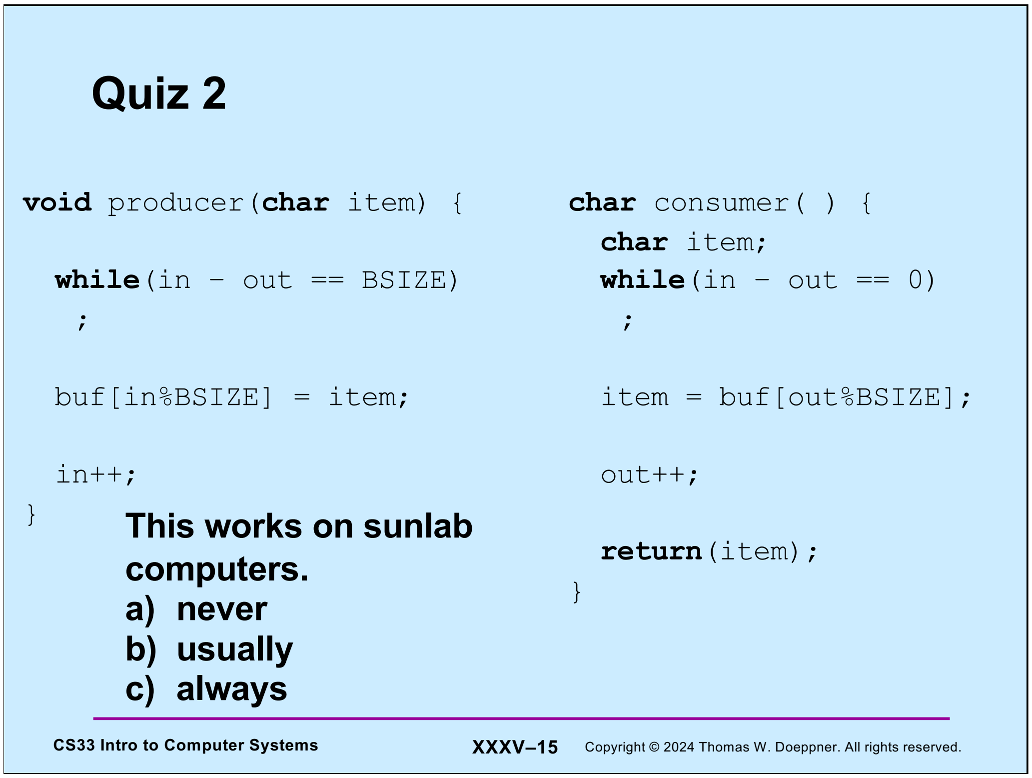
Busy-Waiting Producer/Consumer

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

  buf[in%BSIZE] = item;           item = buf[out%BSIZE];

  in++;                           out++;
}
                                  return(item);
                                }
```

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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 2

```
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 computers.

- **a) never**
- **b) usually**
- **c) always**

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

Sunlab computer have delayed stores.

Quiz 3

```
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 computers with reordered stores.

a) never
b) usually
c) always

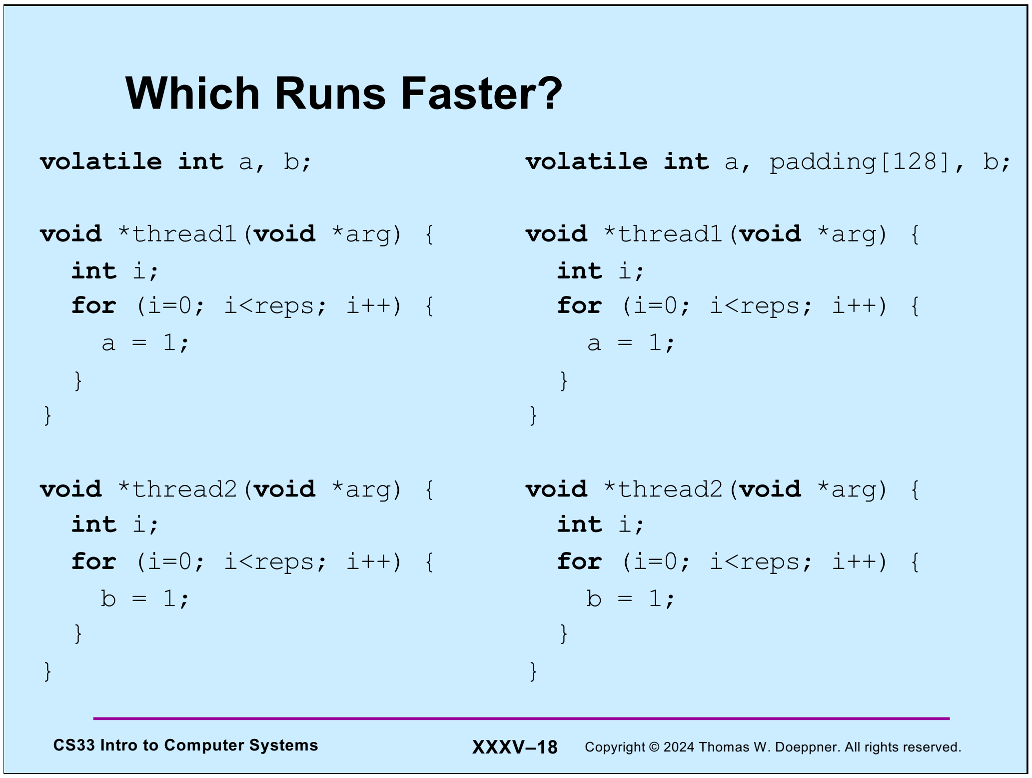
Copyright © 2024 Thomas W. Doeppner. All rights reserved.

Coping

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

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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.



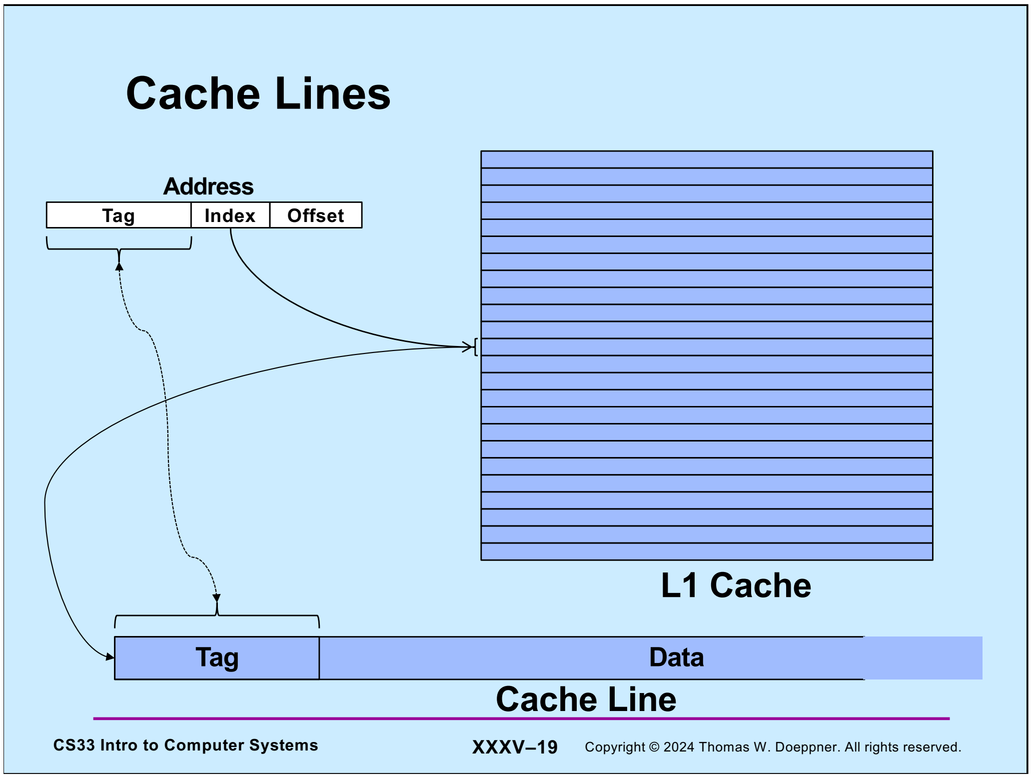
Which Runs Faster?

```
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;
  }
}
```

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

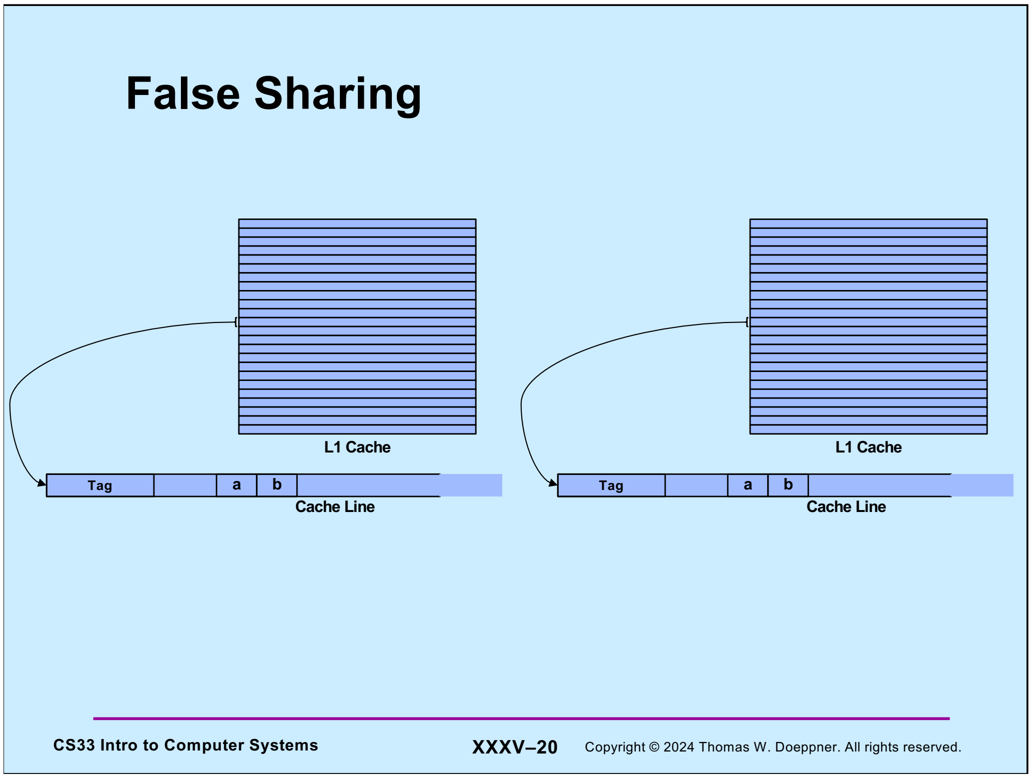
Assume these are run on a two-core prcessor: why does the two-threaded program on the right run faster than the two-threaded program on the left?



Cache Lines

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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.



False Sharing

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

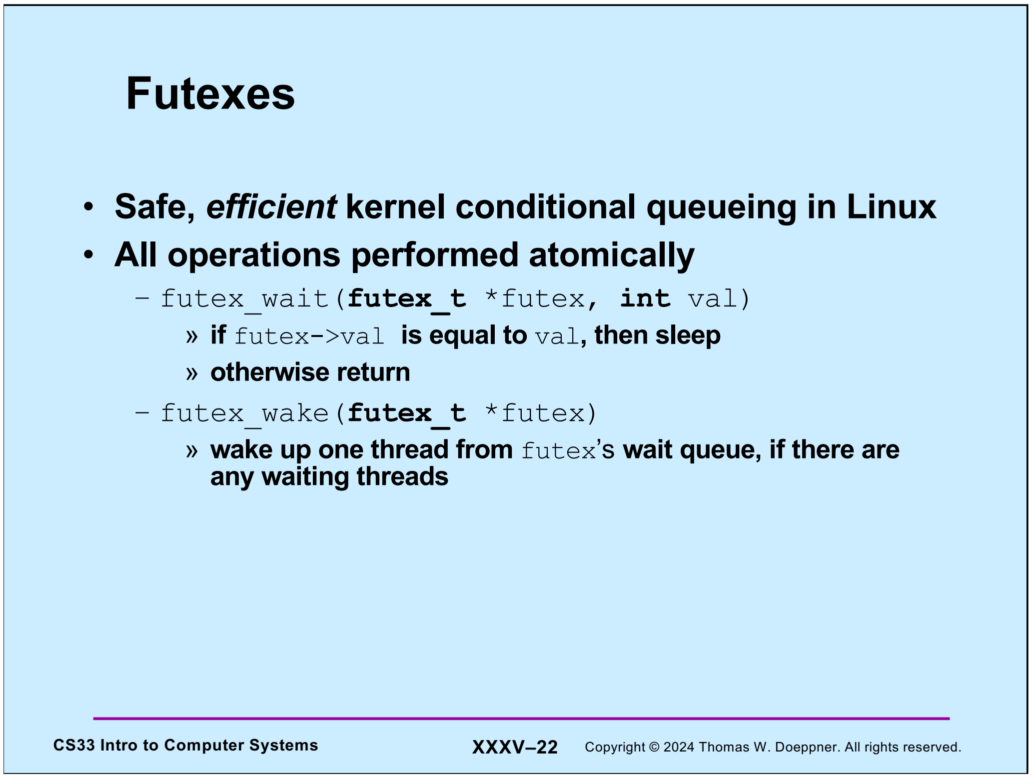
Getting back to our example: we have a two-processor system, and thus two data (L1) caches. If **a** and **b** are in the same cache line, then when either processor accesses **a**, it also accesses **b**. Thus if **a** is modified on processor 1, memory coherency will cause the entire cache line to 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)

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

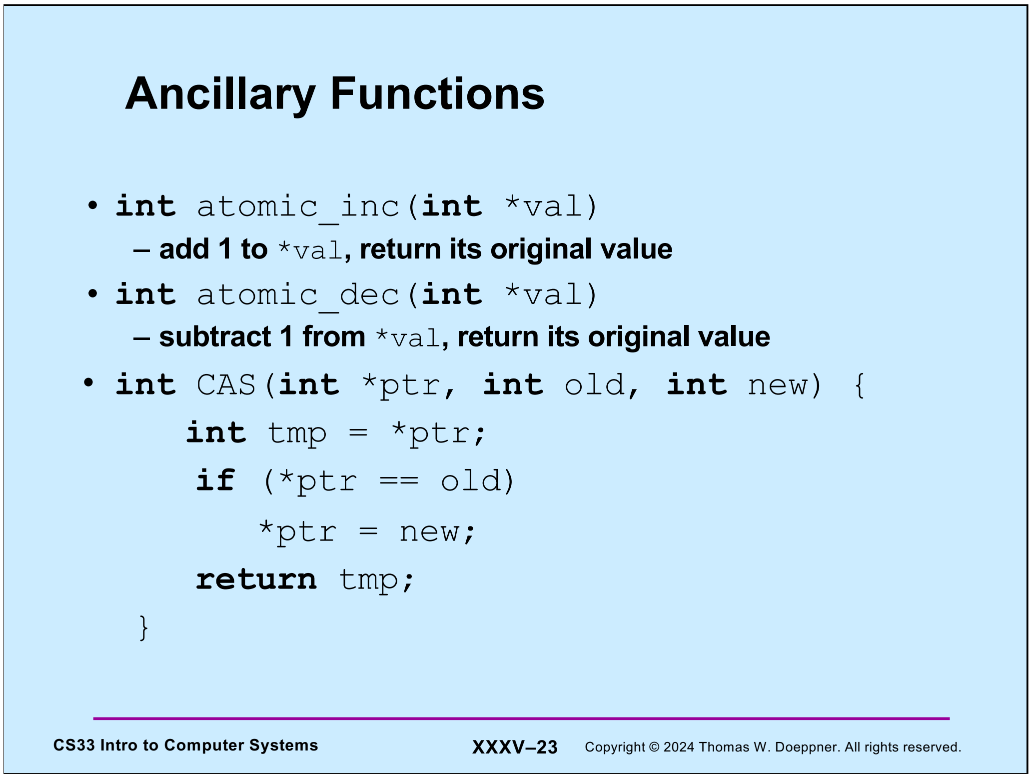


Futexes

- 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**

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

For details on futexes, avoid the Linux man pages, but look at http://people.redhat.com/drepper/futex.pdf, from which this material was obtained. 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 CAS(int *ptr, int old, int new) {
    int tmp = *ptr;
    if (*ptr == old)
        *ptr = new;
    return tmp;
}
```

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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);
}
```

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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

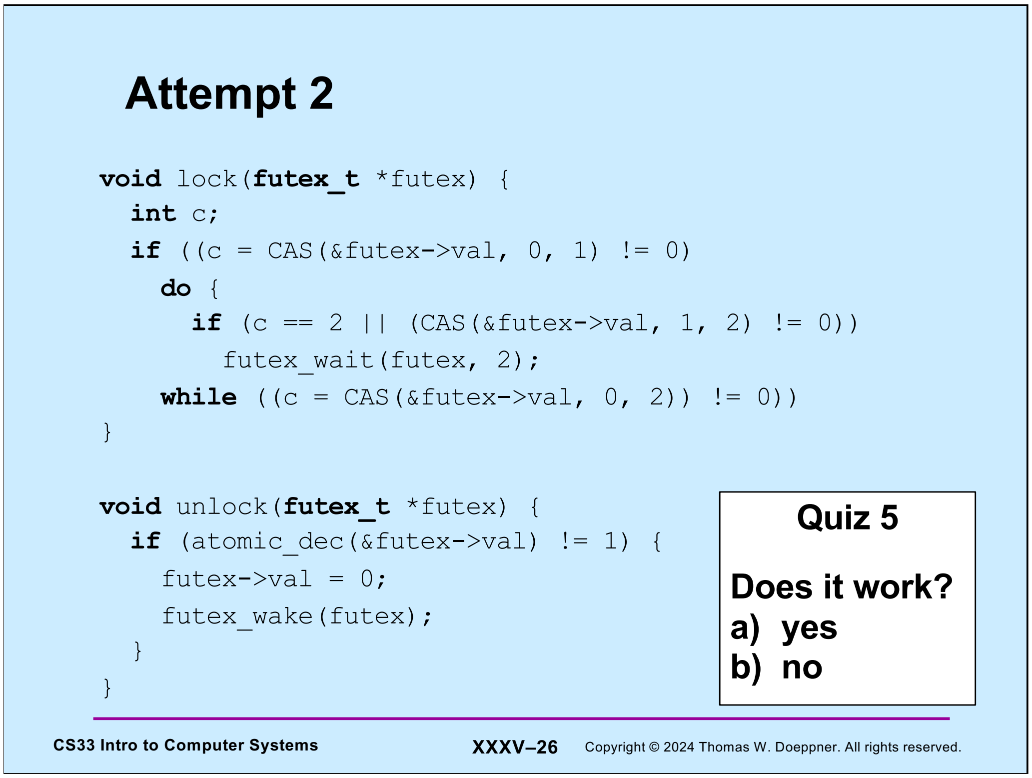
Quiz 4

```
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

Copyright © 2024 Thomas W. Doeppner. All rights reserved.



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 5

Does it work?

- **a) yes**
- **b) no**

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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.