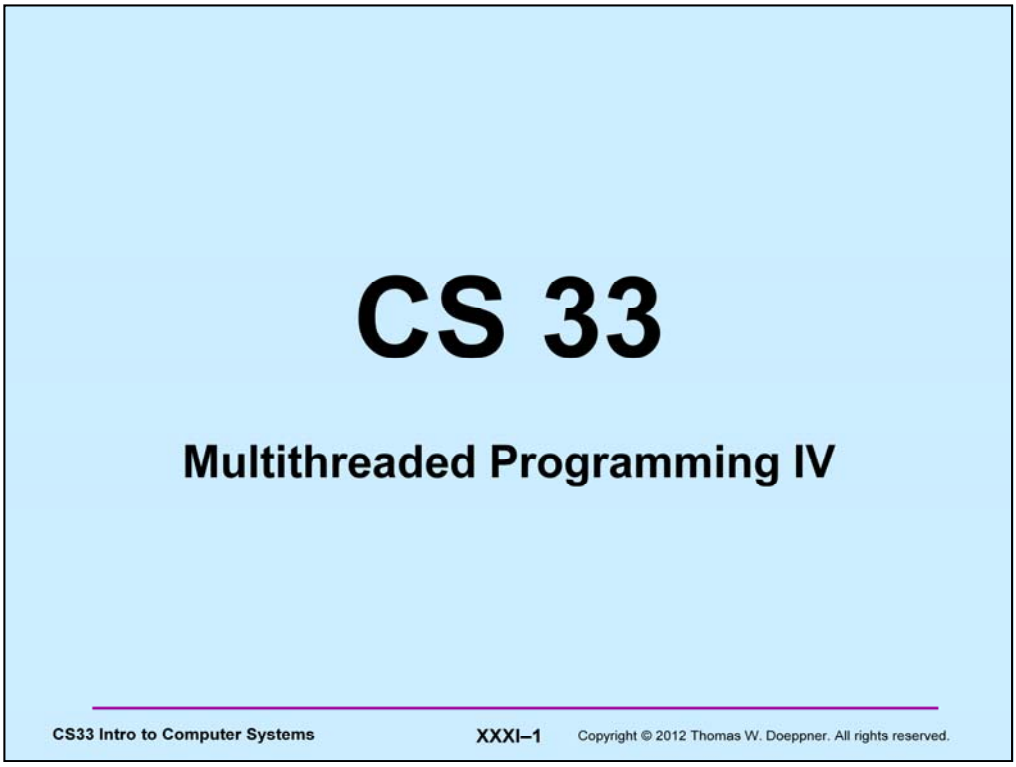
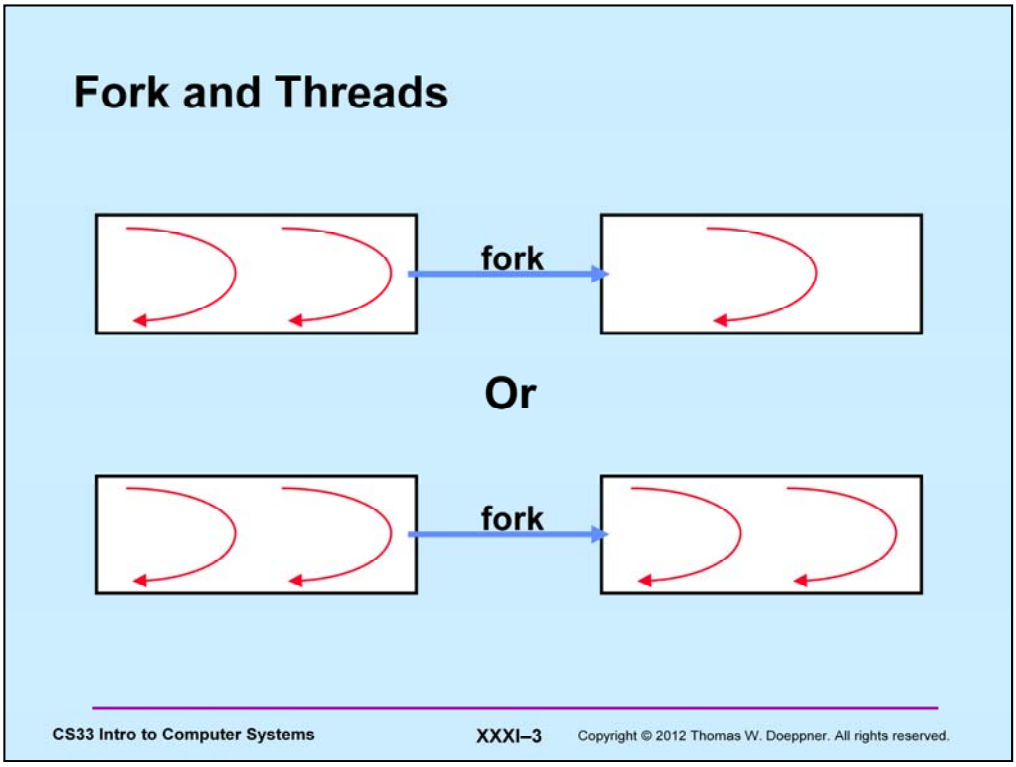


The source code used in this lecture is available on the course web page.

Outline

- Threads and processes
- Shared-memory multiprocessors and caches
  - consistency
  - performance
- Alternatives to mutexes

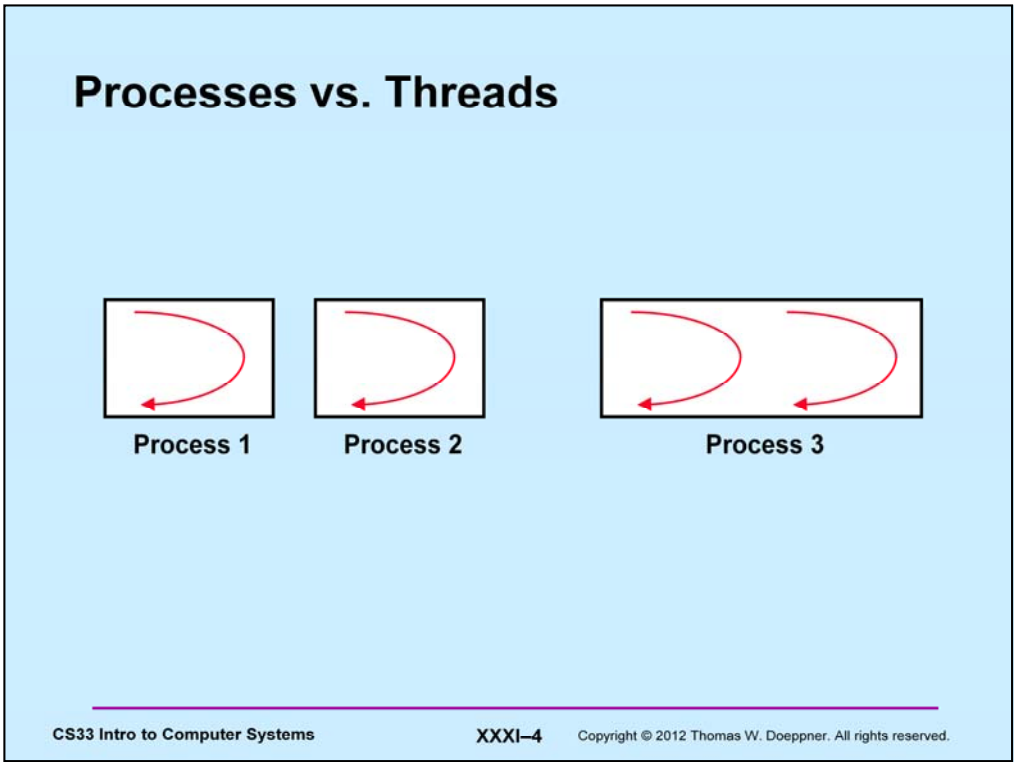


What happens when a thread in a multithreaded process calls *fork*? Clearly the child process's address space is a copy of the parent's, but what threads should appear in the child? Two possibilities are reasonable: all the parent's threads are replicated in the child, or just the thread that called *fork* is replicated in the child. Neither approach is ideal for all circumstances.

Suppose just one thread is created in the child. If the child process never calls *exec*, but uses its copy of the parent's address space, we could have a problem. Suppose some thread in the parent process, other than the one that called *fork*, has locked a mutex. This mutex is copied into the child process in its locked state, but no thread is copied over to unlock the mutex. Thus any thread in the child attempting to lock the mutex will wait forever. In this situation it makes the most sense for *fork* to replicate all of the threads of the process.

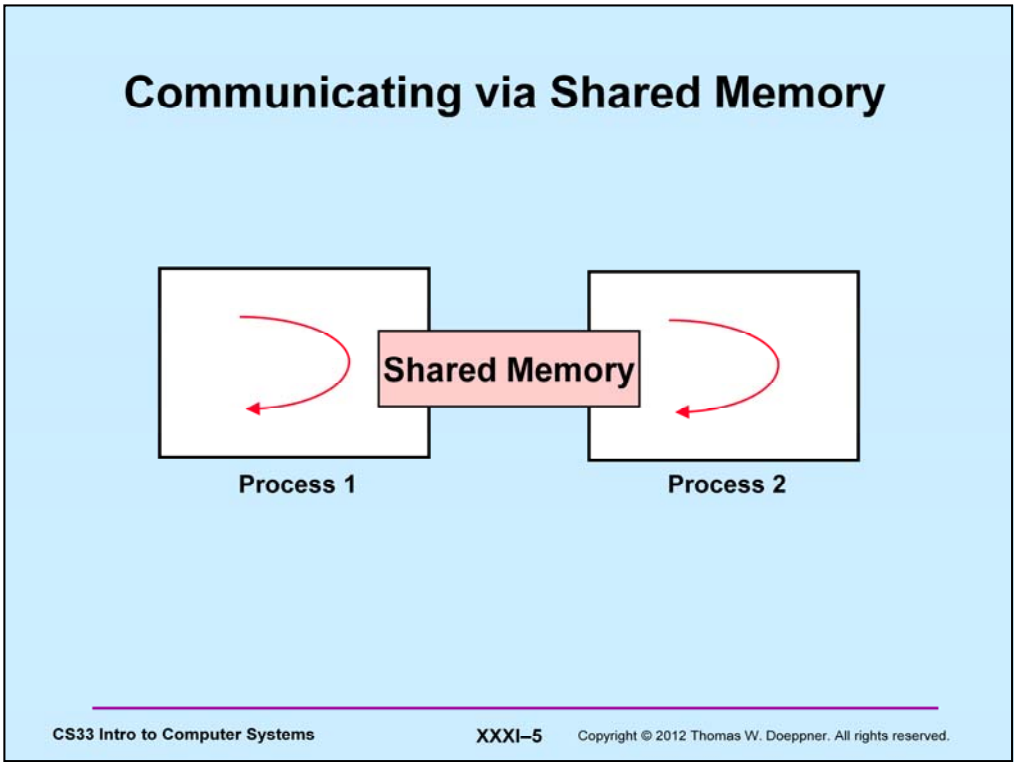
However, suppose that a thread in the child process immediately calls *exec*, which is what happens after most calls to *fork*. In this case, it would be pointless to replicate in the child process any thread other than the one which called *fork*.

Most versions of Unix provide only one form of *fork*, which duplicates only the thread that called it. To deal with the locking problems, one may employ *atfork handlers*: one calls *pthread_atfork* (any number of times) to register three functions (per call) that are called just before the *fork* takes place in the parent and just after the *fork* takes place in both parent and child. These routines are used to acquire locks before the call to *fork* and to release them afterwards — they insure that no thread that is not copied to the child process is holding any locks at the time of the *fork*.



When writing a concurrent program, one makes a decision: should one's program use multiple threads in one process or use multiple processes, each containing a thread? The former approach, as we argued in earlier, is much more efficient than the latter. However, if in the former approach there is a bug in the code executed by one of the threads, this thread can damage the other threads in the process. But in the latter approach, the threads are isolated from one another, so that one thread cannot directly damage the other threads, since each is in a separate address space.

Another issue that often arises is that one wants to arrange for cooperation between unrelated programs: you would like your program to interact with some other program, but the other program exists in binary form and you have no means for incorporating it into yours.



One can reduce the expense of communicating between threads of different processes by using shared memory that has been mapped into both processes. All of the POSIX-threads synchronization primitives may be used to synchronize threads of different processes, as long as the synchronization primitives are set up properly.

Cross-Process Synchronization

```
pthread_mutexattr_t mattr;
pthread_condattr_t cattr;
pthread_mutex_t mut;
pthread_cond_t cond;

pthread_mutexattr_init(&mattr);
pthread_condattr_init(&cattr);

pthread_mutexattr_setpshared(&mattr,
      PTHREAD_PROCESS_SHARED);
pthread_condattr_setpshared(&cattr,
      PTHREAD_PROCESS_SHARED);

pthread_mutex_init(&mut, &mattr);
pthread_cond_init(&cond, &cattr);
```


The slide shows how one sets up mutexes and condition variables to be used across processes. Of course, one must arrange so that the mutexes and condition variables reside in shared memory.

Cross-Process Producer-Consumer (1)

```
typedef struct buffer {
  char buf[BSIZE];
  sem_t filled;
  sem_t empty;
  int nextin;
  int nextout;
} buffer_t;
```


Here we once again solve the producer-consumer problem, this time with cross-process semaphores (which are initialized as such by setting the second (*pshared*) argument of *sem_init* to 1).

Cross-Process Producer-Consumer (2)

```
int main() {
  buffer_t *buffer;

  if ((buffer = mmap(0, sizeof(buffer_t),
       PROT_READ|PROT_WRITE,
       MAP_SHARED|MAP_ANONYMOUS, -1, 0))
       == (void *)-1) {
    perror("mmap");
    exit(1);
  }
```


Here we call *mmap* with the MAP_ANONYMOUS flag. This tells the system that we aren't mapping a file into the address space, but are mapping an *anonymous* region of memory that is not backed by a file. This region (because of the MAP_SHARED flag) will be shared with all of this process's children.

Cross-Process Producer-Consumer (3)

```
    buffer->nextin = buffer->nextout = 0;
    sem_init(&buffer->filled, 1, 0);
    sem_init(&buffer->empty, 1, BSIZE);

    if (fork() == 0)
      consumer_driver(buffer);
    else
      producer_driver(buffer);

    return 0;
}
```


Here we initialize the buffer structure within the anonymous region. Note that the second argument of *sem_init* is set to 1, meaning that semaphore is to be used by threads of multiple processes.

Cross-Process Producer-Consumer (4)

```
void producer_driver(
    buffer_t *b) {
  int item;

  while (1) {
    item = getchar();
    if (item == EOF) {
      produce(b, '\0');
      break;
    } else {
      produce(b, (char)item);
    }
  }
}
```

```
void consumer_driver(
    buffer_t *b) {
  char item;
  while (1) {
    if ((item = consume(b))
        == '\0')
      break;
    putchar(item);
  }
}
```

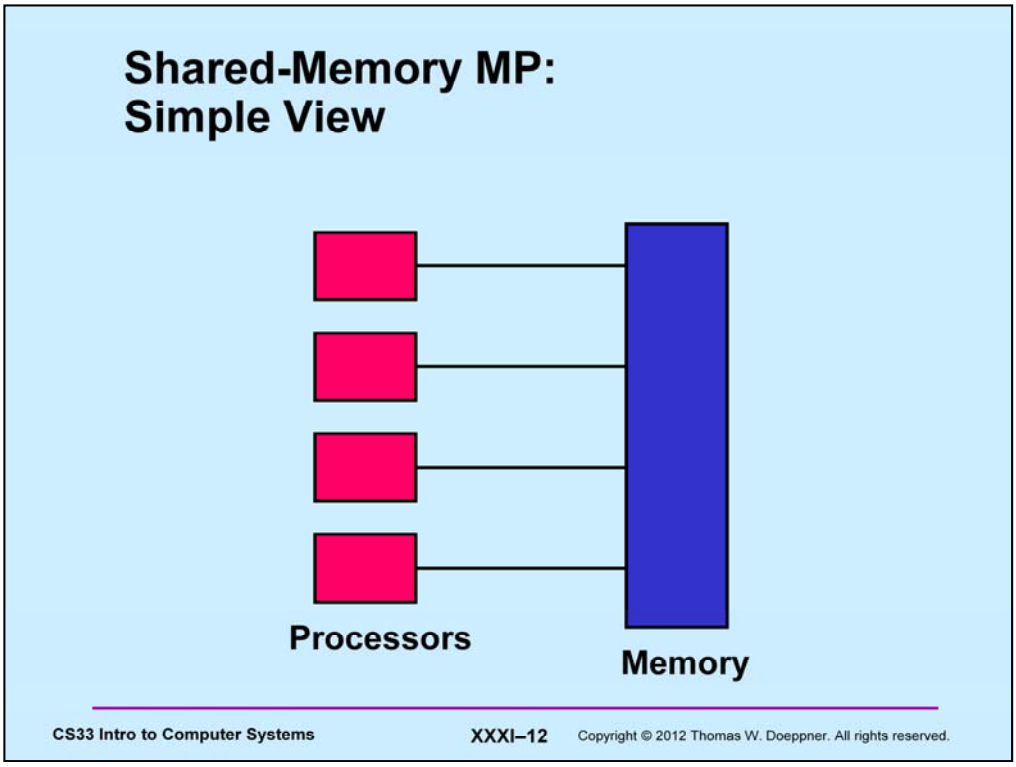

Here is the code that drives our example.

Cross-Process Producer-Consumer (5)

```
void produce(buffer_t *b,
    char item) {
  sem_wait(&b->empty);
  b->buf[b->nextin] = item;
  b->nextin++;
  b->nextin %= BSIZE;
  sem_post(&b->filled);
}
```

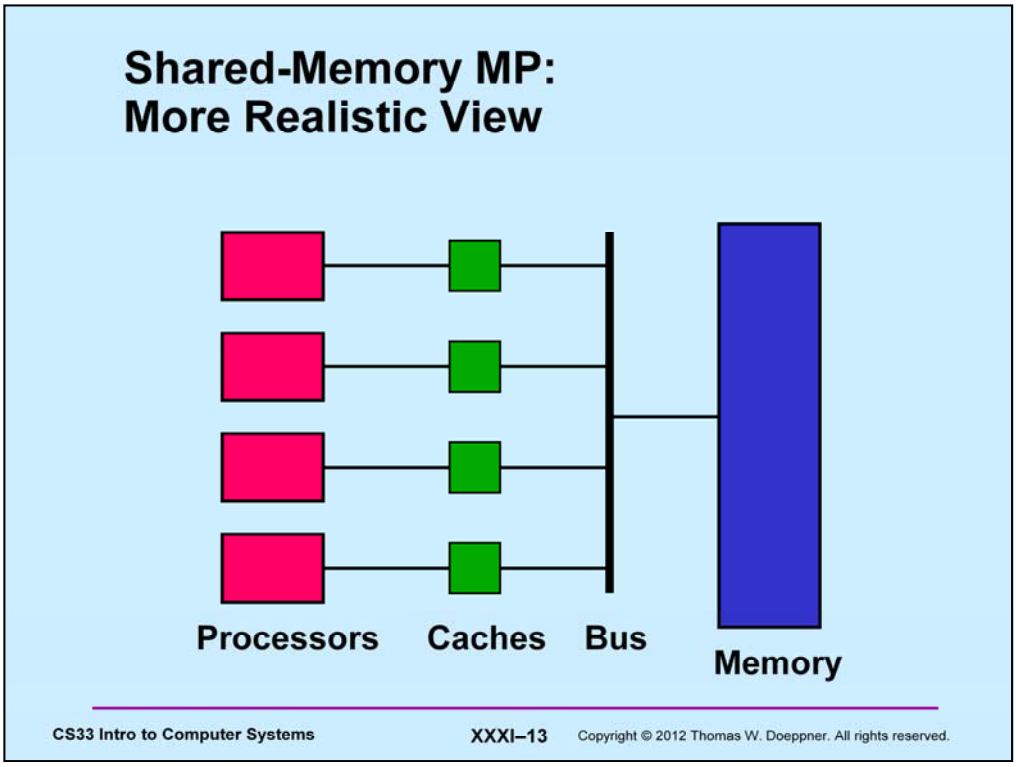
```
char consume(buffer_t *b) {
  char item;
  sem_wait(&b->filled);
  item =
      b->buf[b->nextout];
  b->nextout++;
  b->nextout %= BSIZE;
  sem_post(&b->empty);
  return item;
}
```


Finally, here's a minor variation of the solution that we saw earlier. This version handles only a single producer and a single consumer.



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

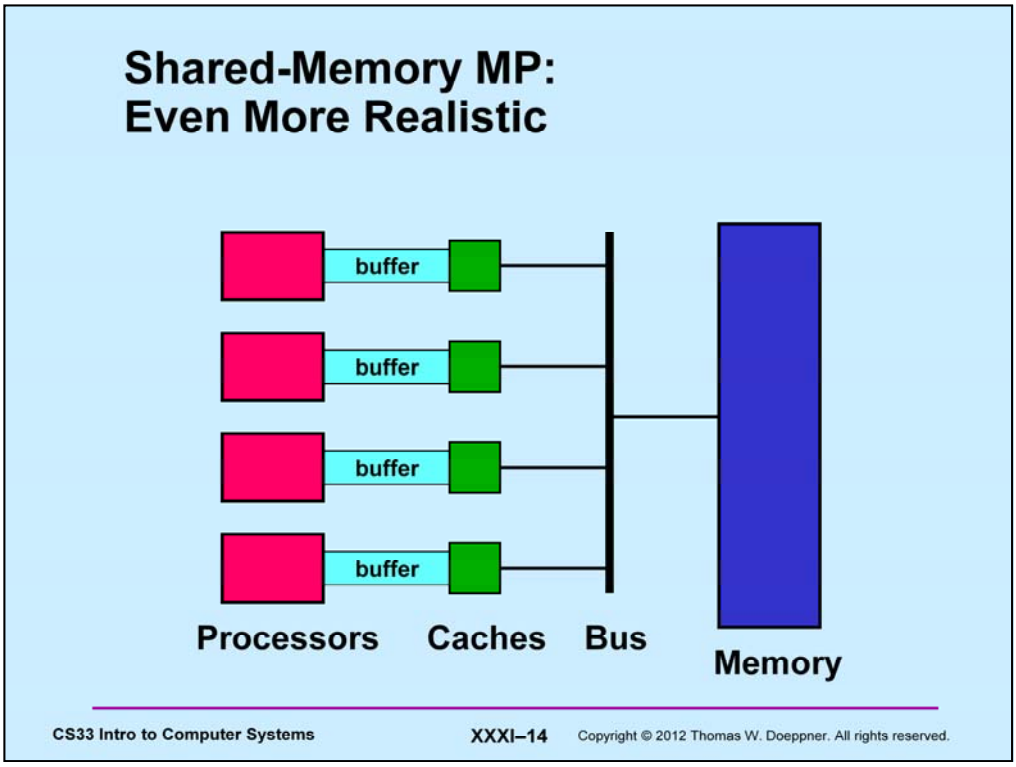
Unfortunately, as we learned earlier in the course, things are not quite so simple.



Real shared-memory multiprocessors have caches that sit between each processor and a common bus; there is a single connection between the bus and the memory. When a processor issues a store, the store affects the cache. When a processor issues a load, the load is dealt with by the cache if possible, and otherwise goes to memory.

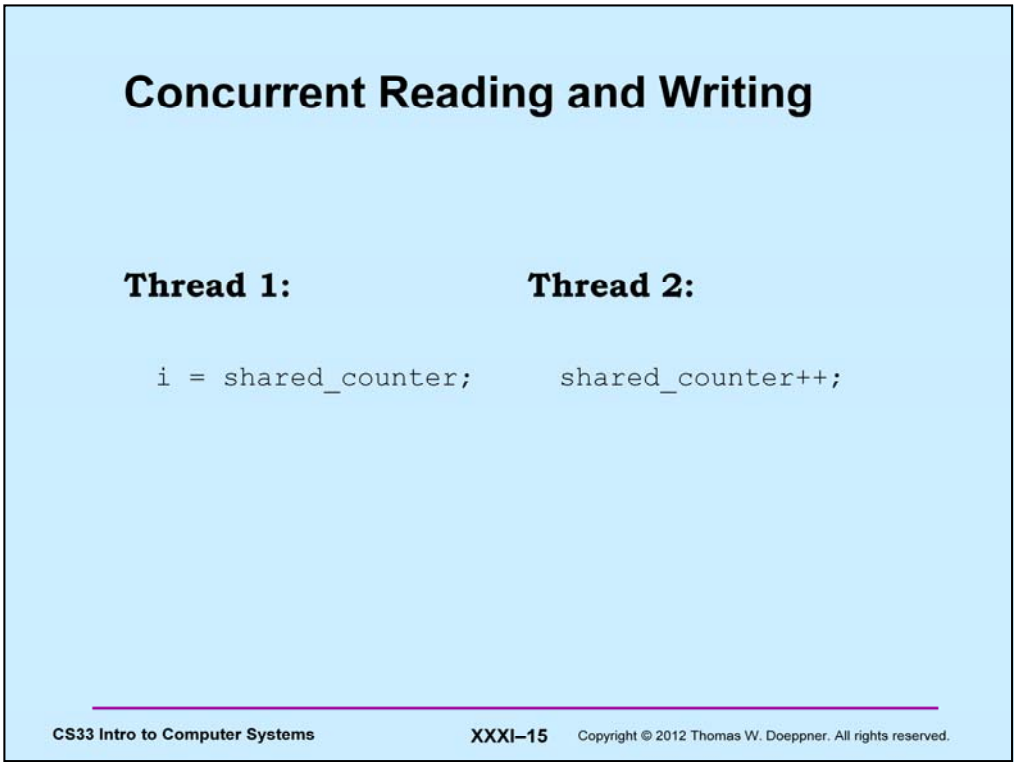
Most architectures have some sort of cache-consistency logic to insure that the shared-memory semantics of the previous page are preserved.

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 the processor and the cache is a buffer. Stores by a processor go into the buffer. Sometime later the effect of the store reaches the cache. In the meantime, the processor is issuing further instructions. Loads by the processor are handled from the buffer if the data is still there; otherwise they go to the cache, and then perhaps to memory.

In all instances of this model the effect of a store, as seen by other processors, is delayed. In some instances of this model the order of stores made by one processor might be perceived differently by other processors. Architectures with the former property are said to have *delayed stores*; architectures with the latter are said to have *reordered stores*.



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., four-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(int me) {
  static int loser;                 // shared
  static int active[2] = {0, 0};    // shared
  int 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?

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 to the producer-consumer problem for one consumer and one producer. It works for the first two shared-memory models, and even for delayed-store architectures. But does it work on reordered-store architectures?

Coping

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

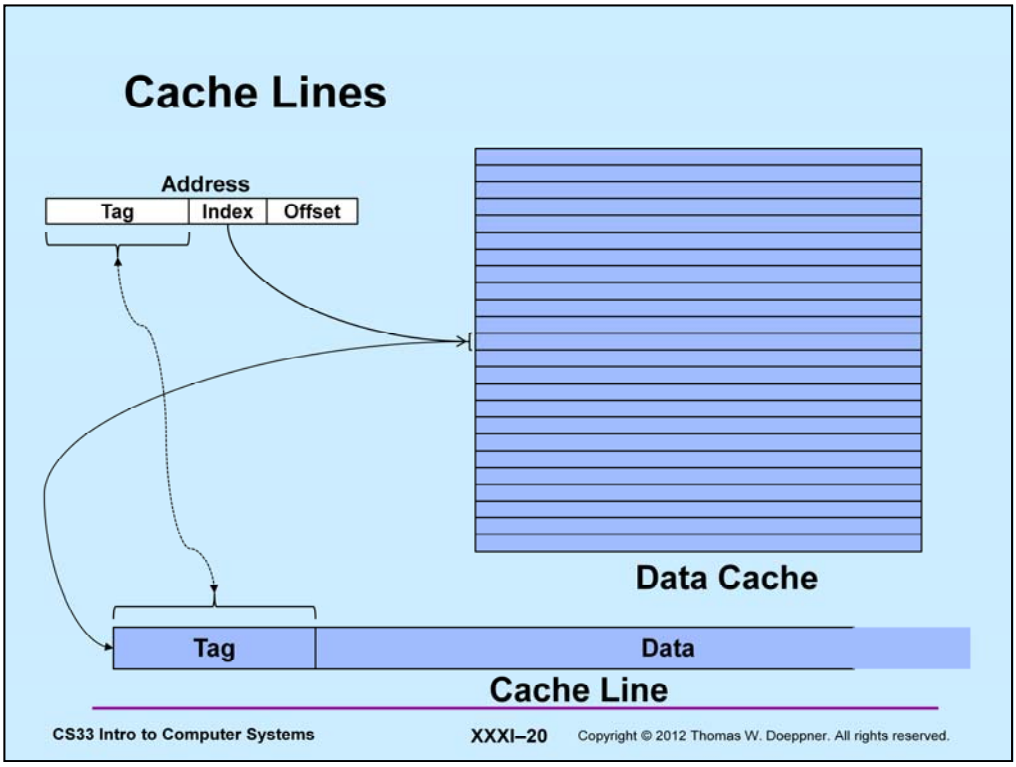
The point of the previous several slides is that one cannot rely on expected properties of shared memory to eliminate explicit synchronization. Shared memory can behave in some very unexpected ways. However, it is the responsibility of the implementers of the various synchronization primitives to make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

Which Runs Faster?

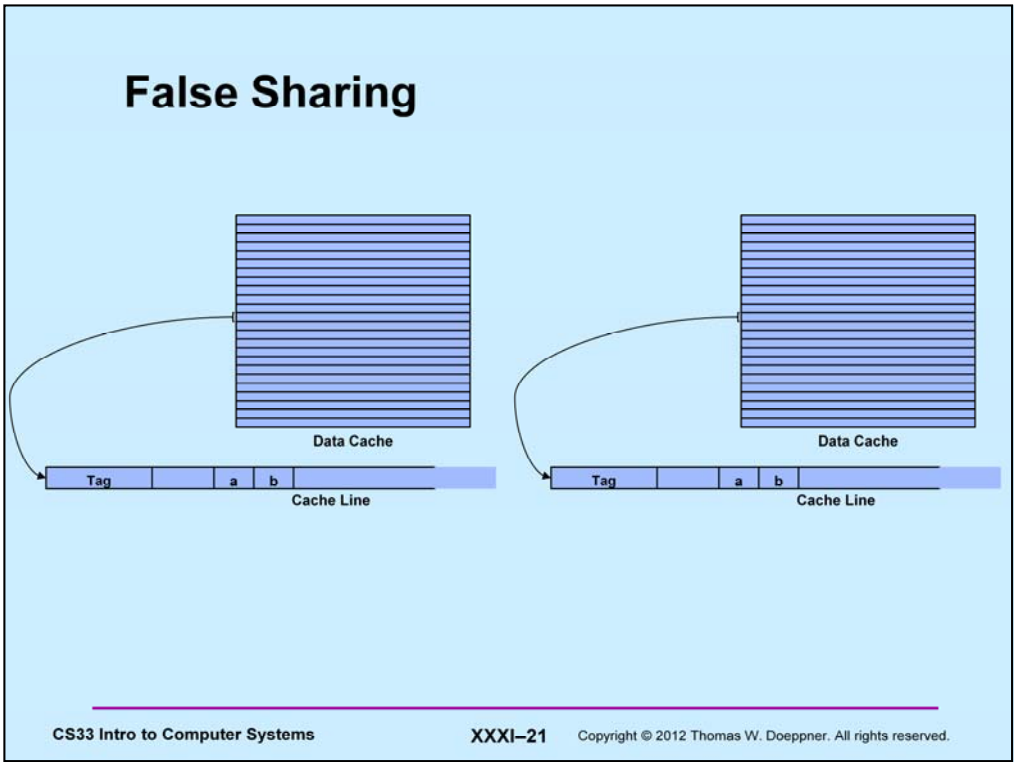
```
volatile int a, b;

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

void *thread2(void *arg) {
  int i;
  for (i=0; i<100000000; i++) {
    b = 1;
  }
}
```

```
volatile int a, padding[32], 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<100000000; 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 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/.

Alternatives to Mutexes:
Atomic Instructions

- Read-modify-write performed atomically
- *Lock prefix* may be used with certain IA32 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

```
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

Yet More From POSIX …

```
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);
```


Spin locks are a simple form of synchronization useful only on multiprocessors. The lock is represented by a single bit. To lock a spin lock, a thread repeatedly tests the lock until it finds it unlocked, then sets it to locked (atomically). The advantage over standard mutexes is that, if two threads are competing for access to a critical section, which both will hold only for a short period of time, it requires fewer instructions for one thread to "wait" for the other by repeatedly testing the lock a few times than it does for it to place a system call to put itself to sleep, then for the holder of the lock to place another system call to wake the sleeping thread up.

Adaptive Mutexes

- Putting a thread to sleep and waking it up are expensive operations
- Spin locks involve neither
- Spin locks monopolize the processor
  - not good if there are other threads to run
- Adaptive mutexes
  - wait by spinning for a short period
  - thread goes to sleep if it spins too long
  - works well if threads usually wait for just a short period

Syntax

```
pthread_mutexattr_t adaptive_attr;
pthread_mutexattr_init(&adaptive_attr);
pthread_mutexattr_settype(&adaptive_attr,
      PTHREAD_MUTEX_ADAPTIVE_NP);

pthread_mutex_t mut;
pthread_mutex_init(&mut, &adaptive_attr);

pthread_mutex_lock(&mut);
  // quick operation
pthread_mutex_unlock(&mut);
```


The slide shows how to specify and use an adaptive mutex. Note the "_NP" suffix on PTHREAD_MUTEX_ADAPTIVE_NP — it stands for "non portable," meaning that this feature is unlikely to exist on other implementations of POSIX threads.