

# CS 33

## Multithreaded Programming V

# Start/Stop



- Start/Stop interface

```
void wait_for_start(state_t *s) {
    pthread_mutex_lock(&s->mutex);
    while (s->state == stopped)
        pthread_cond_wait(&s->queue, &s->mutex);
    pthread_mutex_unlock(&s->mutex);
}

void start(state_t *s) {
    pthread_mutex_lock(&s->mutex);
    s->state = started;
    pthread_cond_broadcast(&s->queue);
    pthread_mutex_unlock(&s->mutex);
}
```

# Start/Stop



- Start/Stop interface

```
void wait_for_start(state_t *s) {  
    pthread_mutex_lock(&s->mutex);  
    while(s->state == stopped)  
        pthread_cond_wait(&s->queue,  
                        &s->mutex);  
    pthread_mutex_unlock(&s->mutex);  
}  
  
void start(state_t *s) {  
    pthread_mutex_lock(&s->mutex);  
    s->state = started;  
    pthread_cond_broadcast(&s->queue);  
    pthread_mutex_unlock(&s->mutex);  
}
```

## Quiz 1

You're in charge of designing POSIX threads. Should *pthread\_cond\_wait* be a cancellation point?

- a) no
- b) yes; cancelled threads must acquire mutex before invoking cleanup handler
- c) yes; but they don't acquire mutex

# Start/Stop



- Start/Stop interface

```
void wait_for_start(state_t *s) {
    pthread_mutex_lock(&s->mutex);
    pthread_cleanup_push(
        pthread_mutex_unlock, &s);
    while(s->state == stopped)
        pthread_cond_wait(&s->queue, &s->mutex);
    pthread_cleanup_pop(1);
}

void start(state_t *s) {
    pthread_mutex_lock(&s->mutex);
    s->state = started;
    pthread_cond_broadcast(&s->queue);
    pthread_mutex_unlock(&s->mutex);
}
```

## Cancellation and Conditions

```
pthread_mutex_lock(&m);
pthread_cleanup_push(pthread_mutex_unlock, &m);
while(should_wait)
    pthread_cond_wait(&cv, &m);

// ... (code perhaps containing other cancellation points)

pthread_cleanup_pop(1);
```

In this example we handle cancels that might occur while a thread is blocked within *pthread\_cond\_wait*. Again we assume the thread has cancels enabled and deferred. The thread first pushes a cleanup handler on its stack — in this case the cleanup handler unlocks the mutex. The thread then loops, calling *pthread\_cond\_wait*, a cancellation point. If it receives a cancel, the cleanup handler won't be called until the mutex has been reacquired. Thus we are certain that when the cleanup handler is called, the mutex is locked.

What's important here is that we make sure the thread does not terminate without releasing its lock on the mutex *m*. If the thread acts on a cancel within *pthread\_cond\_wait* and the cleanup handler were invoked without first taking the mutex, this would be difficult to guarantee, since we wouldn't know if the thread had the mutex locked (and thus needs to unlock it) when it's in the cleanup handler.

## A Problem ...

- In thread 1:

```
if ((ret = open(path,  
    O_RDWR) == -1) {  
    if (errno == EINTR) {  
        ...  
    }  
    ...  
}
```

- In thread 2:

```
if ((ret = socket(AF_INET,  
    SOCK_STREAM, 0)) {  
    if (errno == ENOMEM) {  
        ...  
    }  
    ...  
}
```

**There's only one errno!**

**However, somehow it works.**

**What's done???**

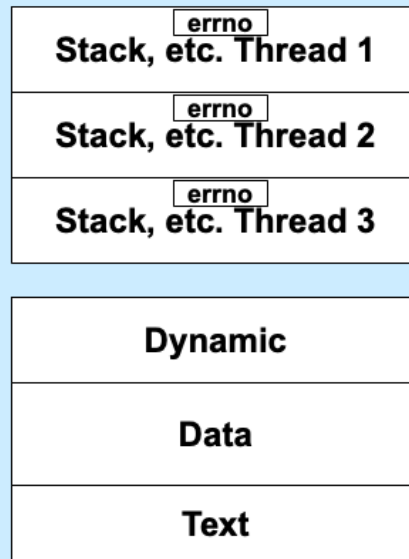
## A Solution ...

```
#define errno (*__errno_location())
```

- **`__errno_location` returns an `int *` that's different for each thread**
  - thus each thread has, effectively, its own copy of `errno`

When you give gcc the `-pthread` flag, it, among other things, defines some preprocessor variables that cause some code in the standard header files to be compiled (that otherwise wouldn't be). In particular the `#define` statement given in the slide is compiled.

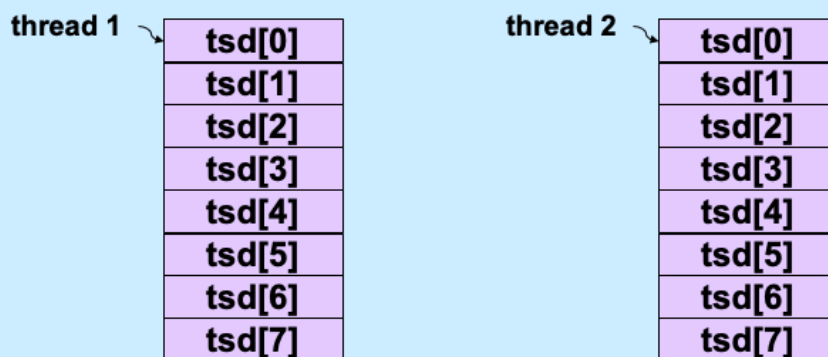
## Process Address Space





## Generalizing

- **Thread-specific data** (sometimes called **thread-local storage**)
  - data that's referred to by global variables, but each thread has its own private copy



## Some Machinery

- `pthread_key_create(&key, cleanup_routine)`
  - **allocates a slot in the TSD arrays**
  - **provides a function to cleanup when threads terminate**
- `value = pthread_getspecific(key)`
  - **fetches from the calling thread's array**
- `pthread_setspecific(key, value)`
  - **stores into the calling thread's array**

So that we can be certain that it's the calling thread's array that is accessed, rather than access the TSD array directly, one uses a set of POSIX threads library routines. To find an unused slot, one calls *pthread\_key\_create*, which returns the index of an available slot in its first argument. Its second argument is the address of a routine that's automatically called when the thread terminates, so as to do any cleanup that might be necessary (it's called with the key (index) as its sole argument, and is called only if the thread has actually stored a non-null value into the slot). To put a value in a slot, i.e., perform the equivalent of  $\text{TSD}[i] = x$ , one calls *pthread\_setspecific*(*i*,*x*). To fetch from the slot, one calls *pthread\_getspecific*(*i*).

## Beyond POSIX

### TLS Extensions for ELF and gcc

- Thread Local Storage (TLS)

```
__thread int x=6;
// Each thread has its own copy of x,
// each initialized to 6.
// Linker and compiler do the setup.
// May be combined with static or extern.
// Doesn't make sense for local variables!
```

ELF stands for “executable and linking format” and is the standard format for executable and object files used on most Unix systems. The `__thread` attribute tells gcc that the item being declared is to be thread-local, which is the same thing as thread-specific. A detailed description of how it is implemented can be found at <http://people.redhat.com/drepper/tls.pdf>.

## Example: Per-Thread Windows

```
typedef struct {  
    wcontext_t win_context;  
    int file_descriptor;  
} win_t;  
__thread static win_t my_win;  
  
void getWindow() {  
    my_win.win_context = ... ;  
    my_win.file_descriptor = ... ;  
}  
  
int threadWrite(char *buf) {  
    int status = write_to_window(  
        &my_win, buf);  
  
    return(status);  
}  
  
void *tfunc(void *arg) {  
    getWindow();  
  
    threadWrite("started");  
    ...  
  
    func2(...);  
}  
  
void func2(...) {  
    threadWrite(  
        "important msg");  
    ...  
}
```

In this example, we put together per-thread windows for thread output. Threads call *getWindow* to set up a window for their exclusive use, then call *threadWrite* to send output to their windows. Individual threads can now set up their own windows and write to them without having to pass around information describing their windows.

## Static Local Storage

```
char *strtok(char *str, const char *delim) {  
    static char *saveptr;  
  
    ... // find next token starting at either  
    ... // str or saveptr  
    ... // update saveptr  
  
    return(&token);  
}
```

An example of the single-thread mentality in early Unix is the use of static local storage in a number of library routines. An example of this is *strtok*, which saves a pointer into the input string for use in future calls to the function. This works fine as long as just one thread is using the function, but fails if multiple threads use it – each will expect to find its own saved pointer in *saveptr*, but there's only one *saveptr*.

## Coping

- **Use thread local storage**
- **Allocate storage internally; caller frees it**
- **Redesign the interface**

As the slide shows, there are at least three techniques for coping with this problem. We could use thread-local storage, but this would entail associating a fair amount of storage with each thread, even if it is not using *strtok*. We might simply allocate storage (via *malloc*) inside *strtok* and return a pointer to this storage. The problem with this is that the calls to *malloc* and *free* could turn out to be expensive. Furthermore, this makes it the caller's responsibility to free the storage, introducing a likely storage leak.

The solution taken is to redesign the interface. The “thread-safe” version is called *strtok\_r* (the *r* stands for *reentrant*, an earlier term for “thread-safe”); it takes an additional parameter pointing to storage that holds *saveptr*. Thus the caller is responsible for both the allocation and the liberation of the storage containing *saveptr*; this storage is typically a local variable (allocated on the stack), so that its allocation and liberation overhead is negligible, at worst.

## Thread-Safe Version

```
char *strtok_r(char *str, const char *delim,
               char **saveptr) {

    ... // find next token starting at either
    ... // str or *saveptr
    ... // update *saveptr

    return(&token);
}
```

Here's the thread-safe version of *strtok*.

## Shared Data

- **Thread 1:**

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

- **Thread 2:**

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

- **Printed on display:**

```
go to Hell
```

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**

To deal with this *printf* problem, we must somehow add synchronization to *printf* (and all of the other standard I/O routines). A simple way to do this would be to supply wrappers for all of the standard I/O routines 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 routines 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()`**

After making a library thread-safe, we may discover that many routines have become too slow. For example, the standard-I/O routines *getc* and *putc* are normally 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 routines, 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);
```

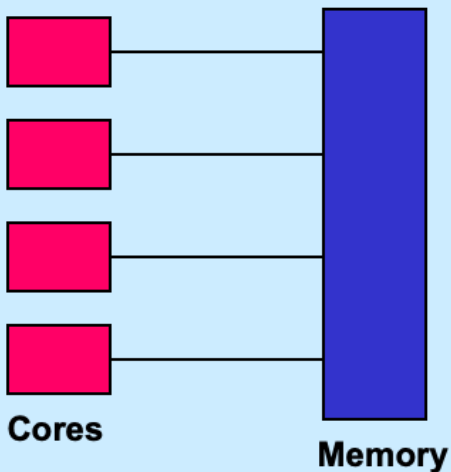
# What's Thread-Safe?

- Everything except

asctime()	ecvt()	gethostent()	getutxline()	putc_unlocked()
basename()	encrypt()	getlogin()	gmtime()	putchar_unlocked()
catgets()	endgrent()	getnetbyaddr()	hcreate()	putenv()
crypt()	endpwent()	getnetbyname()	hdestroy()	pututxline()
ctime()	endutxent()	getnetent()	hsearch()	rand()
dbm_clearerr()	fcvt()	getopt()	inet_ntoa()	readdir()
dbm_close()	ftw()	getprotobyname()	l64a()	setenv()
dbm_delete()	gcvt()	getprotobynumber()	lgamma()	setgrent()
dbm_error()	getc_unlocked()	getprotoent()	lgammaf()	setkey()
dbm_fetch()	getchar_unlocked()	getpwent()	lgammal()	setpwent()
dbm_firstkey()	getdate()	getpwnam()	localeconv()	setutxent()
dbm_nextkey()	getenv()	getpwuid()	localtime()	strerror()
dbm_open()	getgrent()	getservbyname()	lrand48()	strtok()
dbm_store()	getgrgid()	getservbyport()	mrnd48()	ttyname()
dirname()	getgrnam()	getservent()	nftw()	unsetenv()
dlopen()	gethostbyaddr()	getutxent()	nl_langinfo()	wcstombs()
drand48()	gethostbyname()	getutxid()	ptsname()	wctomb()

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

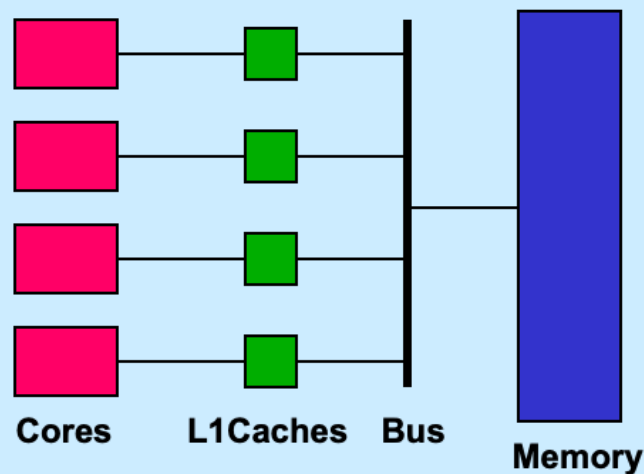
## Multi-Core Processor: Simple View



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.

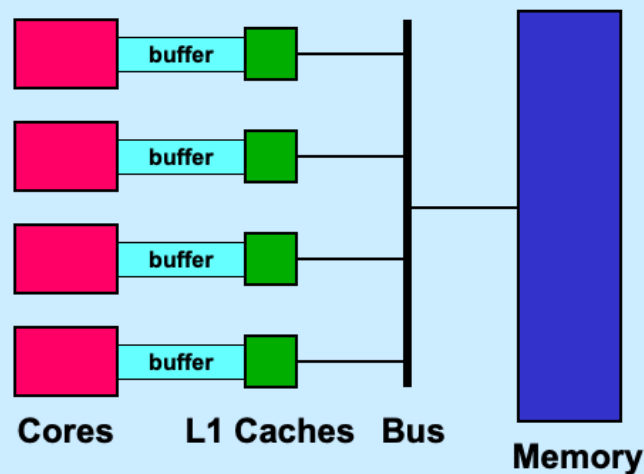
## Multi-Core Processor: More Realistic View



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.

## Multi-Core Processor: Even More Realistic



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).

# Concurrent Reading and Writing

**Thread 1:**

```
i = shared_counter;
```

**Thread 2:**

```
shared_counter++;
```

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.

## Quiz 2

```
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 machines.**

- a) true**
- b) false**

## 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 3

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

**This works on sunlab machines.**

- a) true**
- b) false**

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

## 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;
    }
}

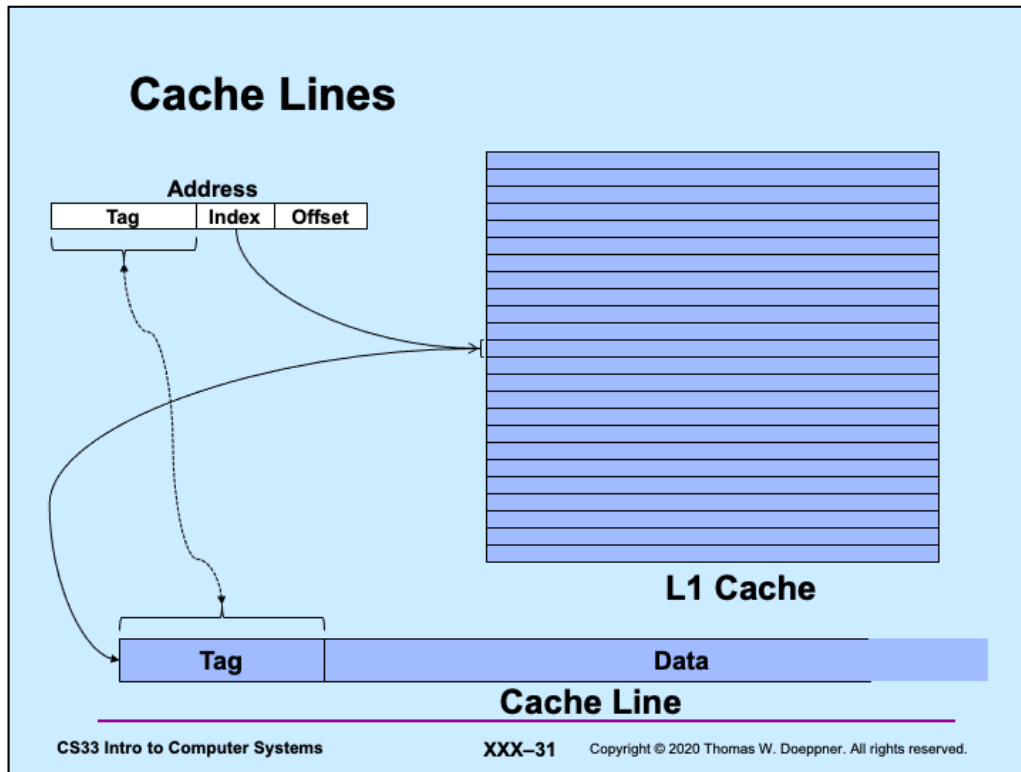
void *thread2(void *arg) {
    int i;
    for (i=0; i<reps; i++) {
        b = 1;
    }
}

volatile int a,
padding[128], b;

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

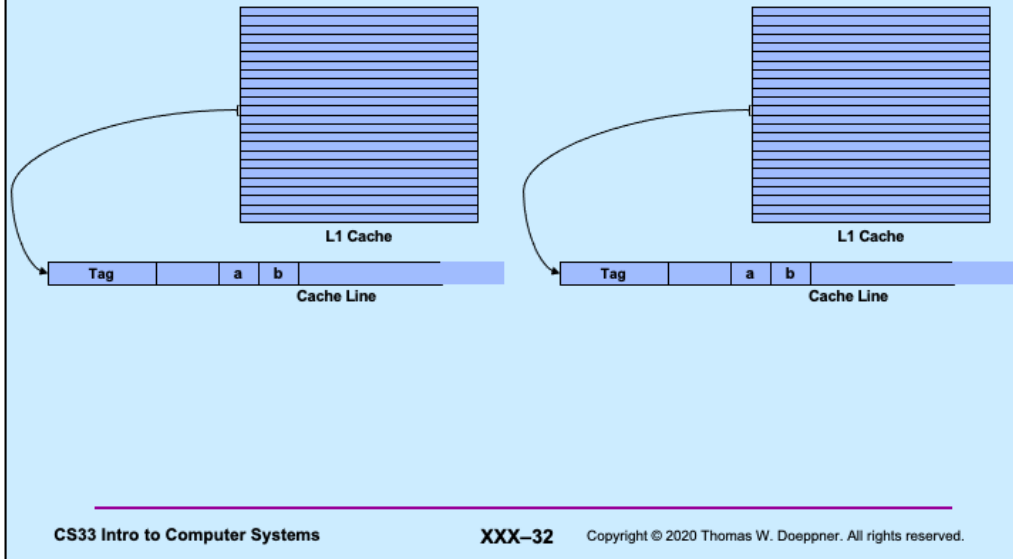
void *thread2(void *arg) {
    int i;
    for (i=0; i<reps; i++) {
        b = 1;
    }
}
```

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.

## False Sharing



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/>.