CS 33

Multithreaded Programming V
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

The use of the lock prefix causes “interlocked access” to memory: the memory bus is grabbed for a few cycles to insure that data can be read, modified, then written without anything happening in between. Note that this has a detrimental effect on overall system performance: while one core has “grabbed” the memory bus, other cores attempting to access memory must wait.
Alternatives to Mutexes:
Spin Locks

- Consider

```c
pthread_mutex_lock(&mutex);
new->next = list_ele->next;
list_ele->next = new;
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
Quiz 1

```c
pthread_mutex_lock(&mutex);
new->next = list_ele->next;
list_ele->next = new;
pthread_mutex_unlock(&mutex);
```

The thread that has locked the mutex might yield the processor to another thread due to time slicing. This will necessarily cause threads waiting for the mutex to wait longer. Will this result in those threads using significant additional processor time?

a) yes
b) no
The source operand must be a register.

If `cmpxchg` is used with the lock prefix, it provides a means for implementing a spin lock, as shown in the next slide.
In this code, a pointer to the spin lock is passed as an argument (in %rdi). The `cmpxchg` instruction compares the spin lock's value with the unlocked value (zero, which is in %rax). If equal (it is unlocked), then a 1 (the contents of %r10) is copied into the spin lock, thus setting it to locked. Otherwise the spin lock's value (presumably 1) is copied into %r10 (and ignored). `cmpxchg` is called repeatedly until the spin lock is found to be 0 (unlocked) and then atomically set to 1 (locked).

Unlocking the spin lock is done by simply setting it to zero.
A problem with the approach of the previous slide is that `cmpxchg`, when executed as an interlocked instruction, is fairly expensive and disruptive to all processors, since it must grab the memory bus for a couple cycles. This is particularly a problem if it is executed repeatedly, as in the previous slide. What’s done here is to “spin” using normal compare instructions, then, once the lock is found to be unlocked, to verify that it’s still unlocked and lock it using `cmpxchg`. This has the effect of minimizing the number of times the instruction is used, and thus minimizing the amount of time the memory bus is held.
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.

While spin locks were not part of POSIX 1003.1c, they were added to the most recent POSIX threads specification, 1003.1j.
Using Spin Locks

```c
pthread_spin_lock(&spinlock);
new->next = list_ele->next;
list_ele->next = new;
pthread_spin_unlock(&spinlock);
```
Quiz 2

```c
pthread_spin_lock(&spinlock);
new->next = list_ele->next;
list_ele->next = new;
pthread_spin_unlock(&spinlock);
```

The thread that has locked the spin lock might yield the processor to another thread due to time slicing. This will necessarily cause threads waiting for the spin lock to wait longer. Will this result in those threads using significant additional processor time?

   a) yes
   b) no
A Problem ... 

- In thread 1: 

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

- In thread 2: 

```c
if ((ret = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
    if (errno == ENOMEM) {
        ... 
    }
    ... 
}
```

There's only one errno!

However, somehow it works.

What's done???
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

- **Stack, etc. Thread 1**
- **Stack, etc. Thread 2**
- **Stack, etc. Thread 3**

- **Dynamic**
- **Data**
- **Text**
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

```plaintext
thread 1
  tsd[0]
  tsd[1]
  tsd[2]
  tsd[3]
  tsd[4]
  tsd[5]
  tsd[6]
  tsd[7]

thread 2
  tsd[0]
  tsd[1]
  tsd[2]
  tsd[3]
  tsd[4]
  tsd[5]
  tsd[6]
  tsd[7]
```
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 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)

```c
__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.
In this example, we put together a window package 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.

```c
typedef struct {
    wcontext_t win_context;
    int file_descriptor;
} w_t;
__thread static w_t my_win;

void *getWindow() {
    my_win.win_context = ... ;
    my_win.file_descriptor = ... ;
}

void *tfunc(void * arg) {
    getWindow();
    threadWrite("started");
    ...
    func2(...);
}

int threadWrite(char *buf) {
    int status = write_to_window(&my_win, buf);
    return(status);
}

void func2(...) {
    threadWrite("important msg");
    ...
}
```
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`.

```c
char *strtok(char *str, const char *delim) {
    static char *saveptr;

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

    return(&token);
}
```
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.
Here’s the thread-safe version of `strtok`.

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

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

    return(&token);
}
```
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.

Shared Data

- **Thread 1:**
  printf("goto statement reached");

- **Thread 2:**
  printf("Hello World\n");

- **Printed on display:**
  go to Hell
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).
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

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

• Efficient

```c
flockfile(stdout);
for(i=0; i<lim; i++)
  putc_unlocked(out[i]);
funlockfile(stdout);
```
According to IEEE Std. 1003.1 (POSIX), all functions they specify must be thread-safe, except for those listed above.
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 they are initialized properly for this purpose.
Cross-Process Synchronization

```c
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 initializes 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.
Here we once again solve the producer-consumer problem, this time with cross-process semaphores (which are initialized as such by setting the second \((p\text{shared})\) argument of \texttt{sem\_init} to 1).
Cross-Process Producer-Consumer (2)

```c
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 and is initialized to all zeroes. This region (because of the MAP_SHARED flag) will be shared with all of this process’s children. The file-descriptor argument should be -1.

Note that the effect of the call to `mmap` is to allocate storage for the buffer.
Here we initialize the buffer structure within the anonymous region. Note that the second argument of `sem_init` is set to 1, meaning that the semaphore is to be used by threads of multiple processes. The call to `fork` creates a child process that shares the buffer structure with its parent.
Here is the code that drives our example.

```c
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);
    }
}
```
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 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.
Shown on the slide is Peterson’s algorithm for handling mutual exclusion for two threads without explicit synchronization. (The \textit{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?

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
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 4

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

This works on sunlab machines.
a) true
b) false
```

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

  return(item);
}
```
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.
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/.