CS 33

Multithreaded Programming V
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
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.
Spin Lock

- the spin lock is pointed to by the first arg (%rdi)
  - locked is 1, unlocked is 0
    .text
    .globl slock, sunlock
    slock:
    loop:
      movq $0, %rax
      movq $1, %r10
      lock cmpxchg %r10, 0(%rdi)
      jne loop
      ret
    sunlock:
      movq $0, 0(%rdi)
      ret

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.
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)) {
  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.

```c
#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`
## Process Address Space

<table>
<thead>
<tr>
<th>Stack, etc. Thread 1</th>
<th>Stack, etc. Thread 2</th>
<th>Stack, etc. Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic</td>
<td>Data</td>
<td>Text</td>
</tr>
</tbody>
</table>

- 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

```
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)`.

---

### 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
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.
Stacks

- Stack overflow
  - generate C code to check for it
  - not done by gcc
  - rely on OS to detect

Thread 1 Stack

Thread 2 Stack
Last Quiz!

- With respect to stack overflow, the OS
  a) can’t do anything
  b) can detect it in most cases
  c) can detect it in all cases
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`.
You’ll Soon Finish CS 33 …

• You might
  – celebrate

  – take another systems course
    » 32
    » 138
    » 166
    » 167

  – become a 33 TA
Systems Courses Next Semester

- **CS 32**
  - you’ve mastered low-level systems programming
  - now do things at a higher level
  - learn software-engineering techniques using Java, XML, etc.

- **CS 138**
  - you now know how things work on one computer
  - what if you’ve got lots of computers?
  - some may have crashed, others may have been taken over by your worst (and brightest) enemy

- **CS 166**
  - liked buffer?
  - you’ll really like 166

- **CS 167/169**
  - still mystified about what the OS does?
  - write your own!
The End

Well, not quite ...
Database is due on 12/16.
Most malloc rubrics will be released tonight.

Happy coding and happy holidays!