COSC441 Concurrent Programming

C11 Threading Basics

Richard A. O'Keefe

August 2, 2016

C11

- The C11 standard added threads to ISO C.
- Some libraries still haven't caught up (OSX).
- https://github.com/jtsiomb/c11threads provides a C11 interface to POSIX threads
- `_Atomic` and `_Thread_local` need compiler support
- Look for <threads.h>

Threads (1)

- `thrd_t` — implementation-defined type for identifying threads.
- `thrd_current()` — identifier for current thread
- `thrd_equal(`x`,`y`)` — do x and y refer to the same thread?
- `thrd_exit(`n`)` — cleans up this thread's thread-specific storage, sets result to n, and terminates thread.

Threads (2)

- `thrd_create`(&*id*, *fun*, *arg*) calls *fun(arg)* in a new thread, setting *id* to the identifier of that thread.
- No control over size or location of stack or any other property. *fun* returns int.
- `thrd_join`(*id*, &*res*) — waits for thread *id* to terminate, stores result in *res*
- `thrd_detach`(*id*) makes *id* unjoinable.

parbegin

The older `parbegin` s_1; ...; s_n `end` is
thrd_create(&t1, f1, a1); // f1(a1) does s_1
...
thrd_create(&tn, fn, an); // fn(an) does s_n
thrd_join(t1, NULL); // wait for t1 to finish
...
thrd_join(tn, NULL); // wait for tn to finish

Threads: missing operations

Some operations you might expect are missing.

- is thread x still alive?
- suspend thread x
- resume thread x
- kill thread x (in POSIX)
- enquire about properties of x

Mutexes: concepts

- A mutex is a queue of waiting threads.
- And an indication of whether (plain) or which (recursive) thread holds the mutex, if any.
- Recursive mutex has a counter.
- Support MUTual EXclusion.

Plain mutexes

- atomic lock(plain) {
  if plain is unlocked then mark plain as locked
  otherwise add this thread to the waiting queue
  }
- atomic unlock(plain) {
  it is an error if plain is not locked.
  if the queue is empty mark plain as unlocked
  otherwise wake one thread from the queue }
- Doing lock(&m); lock(&m); deadlocks the calling thread.

Recursive mutexes

- atomic lock(mutex) {
  if mutex's count is 0 set count to 1 and record this thread as the owner, otherwise if this thread is recorded as the owner, increment the count, otherwise add this thread to the waiting queue }
- atomic unlock(mutex) {
  it is an error if this thread is not the mutex's owner.
  decrement the count.
  if the count is 0 and any thread is waiting, wake a waiting thread. }
- Doing lock(&m); lock(&m); works.

Mutexes: interface (1)

- `mtx_t` — implementation-defined type of mutex objects (not pointers or identifiers). Do not copy/assign.
- `mtx_init`(&*mutex*, *type*) Do not call on mutex currently in use!
- *type* is `mtx_plain`, `mtx_timed` (allows timeouts), `mtx_plain|mtx_recursive` (can be repeatedly locked by same thread), or `mtx_timed|mtx_recursive`
- `mtx_destroy`(&*mutex*) — opposite of init. Do not call if still in use.
- No way to tell if mutex is still in use.

Mutexes: interface (2)

- `mtx_lock`(&*mutex*)
  lock the mutex; if you can't do it now, wait until you can.
- `mtx_trylock`(&*mutex*)
  if you can lock the mutex now, do so and return thrd_success, otherwise return thrd_busy.
- `mtx_unlock`(&*mutex*)
  unlock the mutex. (For recursive, might still own.)
- `mtx_timedlock`(&*mutex*, &*when*)
  like `mtx_lock` but gives up and returns `thrd_timedout` if the clock reaches *when* and not able to lock yet.

Barriers and ordering

- Locking and unlocking mutexes acts as a memory barrier. Pending loads and stores are completed, then the locking operation done, then new loads and stores can start.
- Actions in independent threads are generally unordered but init, (lock | trylock | timedlock | unlock)*, destroy operations on any one lock happen in some total order.

Mutexes: missing operations

There is no way to ask "lock exactly one of these and tell me which one".
You can use trylock to find out if a mutex is locked, but you can't ask how many other threads want it.
Deadlock diagnosis requires a graph of which threads hold/want which locks, but you can't access it.

Mutexes: missing language feature

A mutex exists to protect specific data.
There is no link between the data and the mutex in C.
Put the data in a `struct` and put the mutex in the same `struct`.

Conditions: concepts

The basic idea is that inside a critical region, we want to wait until some logical expression is true. To make it true, some other thread will have to get the lock. Ideally, we'd write `await` *expr*;
A "condition variable" is a queue of threads waiting for the same condition to become true of the same protected data.

Conditions: interface (1)

- `cnd_t` — implementation-defined type of condition objects (not pointers or identifiers). Do not copy/assign.
- `cnd_init`(&*cond*). Do not call on *cond* already in use!
- `cnd_destroy`(&*cond*). Opposite of init. Do not call if *cond* still in use.

Conditions: interface (2)

- `cnd_wait`(&*cond*, &*mutex*)
  atomically, add the current thread to the queue of threads in *cond*, and unlock *mutex*. When awoken, re-lock *mutex*.
- `cnd_timedwait`(&*cond*, &*mutex*, *when*)
  Like `cnd_wait` but if clock reaches *when*, wake up anyway and return `thrd_timedout`.
- `cnd_signal`(&*cond*)
  wake up one waiting thread
- `cnd_broadcast`(&*cond*)
  wake up all waiting threads

Conditions: beware

- signals/broadcasts are not counted; if there are no waiting threads when you issue a signal, it's as if it never happened.
- We expect signalling a condition to wake *one* thread. The POSIX specification for this operation says it wakes *at least one* thread. This concession is for multicore machines, and some libraries *do* wake multiple threads.

Thread-local variables

- `_Thread_local` *object-declaration*;
  for declaring variables such that each thread gets its own copy.
- can be combined with `static` (the name is local to this file, but every thread still has a copy)
- or `extern`, the default.
- <threads.h> defines `thread_local` as a synonym
- Such variables may be initialised.

Thread-specific storage (1)

- `tss_t` — implementation-defined type of thread-specific storage key type.
- `tss_create`(&*key*, *destructor*)
  creates a new thread-local variable at run time; *key* identifies this (relative to the calling thread). Values are void*, initially NULL. *destructor* is called to clean up.
- texttttss_destroy(&*key*)
  Reclaims storage used for this key in every thread. Does *NOT* call destructor.
- Destructor is called *ONLY* when thread exits.

Thread-specific storage (2)

- `tss_get`(&*key*)
  return void* value associated with *key* in calling thread.
- `tss_set`(&*key*, *value*)
  set value associated with *key* in calling thread, does not call destructor to clean up old value.
- Note: cannot get or set tss value in any other thread, or form a pointer to it so that could be done indirectly.
- Use `_Thread_local` if you can because that gets type-checked.

Lazy initialisation: single-threaded

```
static initialised = 0;
...
if (!initialised) {
  initialised = 1;
  do_initialisation();
}
```

Lazy initialisation: multi-threaded

```
static once_flag initialised = ONCE_FLAG_INIT;
...
call_once(&initialised, do_initialisation);
```

Atomic types and operations

- `_Atomic`(t) is a type specifier provided t is a type name other than a function, array, atomic, or qualified type.
- `_Atomic` may be used as a qualifier along with a type that is not a function or array type.
- Operations are in <stdatomic.h>

Atomic operations (simplified, 1)

- `atomic_store`(&*dst*, *val*)
- *val* = `atomic_load`(&*src*)
- *old* = `atomic_exchange`(&*dst*, *new*)
- if (`atomic_compare_exchange_weak`(&*dst*, &*old*, *new*) — LL+SC, _strong is CAS.

Atomic operations (simplified, 2)

- *old* = `atomic_fetch_`*op*(&*dst*, *operand*)
  _add like +=, _sub like -=, _and like &=, _or like |=, _xor like ^=, except that old value is returned, not new value.
- You can atomically update integers and pointers, but apparently not floats/doubles.