Review: Condition variables

- **Informs scheduler of which threads can run**
- **Typically done with *condition variables* or *semaphores***
- `struct cond_t;` **(**`pthread_cond_t` **or** `cv` **in OS/161)**
- `void cond_init (cond_t *, ...);`
- `void cond_wait (cond_t *c, mutex_t *m);`
  - Atomically unlock `m` and sleep until `c` signaled
  - Then re-acquire `m` and resume executing
- `void cond_signal (cond_t *c);`
  `void cond_broadcast (cond_t *c);`
  - Wake one/all threads waiting on `c`

Review: Semaphores [Dijkstra]

- **A *Semaphore* is initialized with an integer *N***
  - `sem_create(N)`
- **Provides two functions:**
  - `sem_wait (S)` (originally called *P*)
  - `sem_signal (S)` (originally called *V*)
- **Guarantees `sem_wait` will return only *N* more times than `sem_signal` called**
  - Example: If $N == 1$, then semaphore acts as a mutex with `sem_wait` as lock and `sem_signal` as unlock
- **Semaphores give elegant solutions to some problems**

Races

- **Synchronization used to prevent races**
- **Races can include:**
  - Data Race: accessing to shared variables without a lock
    - Clang/LLVM ThreadSanitizer can find data races easily
  - ABA Races: common concern with wait-free code
    - T1: reads n from the value of X
    - T2: sets X to m then back to n
    - T1: sees X is still n and continues
    - E.g. common in wait-free/lock-free stacks and queues
  - Locking, unlocking and relocking a resource without checking
  - TOCTTOU: Time of Check to Time of Use
    - Common class of security bugs
    - E.g., checking if a file exists, then opening it
      Another program can rename or delete the file inbetween

Races: Data Races

```
int foo;

void inc()
{
    foo++;
}
```

- **Two threads call** `inc()`
- **May drop increments or worse**
- **See:** How to miscompile programs with "benign" data races

Races: ABA

```
struct item {
  /* data */
  struct item *next;
};
typedef struct item *stack_t;

void atomic_push (stack_t *stack, item *i) {
  do {
    i->next = *stack;
  } while (!CAS (stack, i->next, i));
}

item *atomic_pop (stack_t *stack) {
  item *i;
  do {
    i = *stack;
  } while (!CAS (stack, i, i->next));
  return i;
}
```

Races: TOCTTOU

find/rm

readdir (“/tmp”) → “badetc”
lstat (“/tmp/badetc”) → DIRECTORY
readdir (“/tmp/badetc”) → “passwd”

unlink (“/tmp/badetc/passwd”)

Attacker

mkdir (“/tmp/badetc”)
creat (“/tmp/badetc/passwd”)

Races: TOCTTOU

find/rm	Attacker
	mkdir (“/tmp/badetc”)
	creat (“/tmp/badetc/passwd”)
readdir (“/tmp”) → “badetc”	
lstat (“/tmp/badetc”) → DIRECTORY	
readdir (“/tmp/badetc”) → “passwd”	
	rename (“/tmp/badetc” → “/tmp/x”)
	symlink (“/etc”, “/tmp/badetc”)
unlink (“/tmp/badetc/passwd”)	

- **Time-of-check-to-time-of-use [TOCTTOU] bug**
  - find checks that `/tmp/badetc` is not symlink
  - But meaning of file name changes before it is used

Deadlocks

```
/* Globals */
mutex_t la, lb;

Thread #1                        Thread #2
/* Both threads acquire a lock */
mutex_lock(&la);                 mutex_lock(&lb);
/* Deadlock: Both threads own a lock the other wants! */
mutex_lock(&lb);                 mutex_lock(&la);
```

- **Each thread tries to acquire the lock the other has**

Deadlocks

```
/* Globals */
mutex_t la, lb;

Thread #1                        Thread #2
/* Both threads acquire a lock */
mutex_lock(&la);                 mutex_lock(&lb);
/* Deadlock: Both threads own a lock the other wants! */
mutex_lock(&lb);                 mutex_lock(&la);
```

- **Each thread tries to acquire the lock the other has**
- **Solution: Obtain locks in the same order**

Deadlocks

```
/* Globals */
mutex_t la, lb;

Thread #1                          Thread #2
/* Both threads acquire a lock */
mutex_lock(&la);                   mutex_lock(&lb);
/* Deadlock: Both threads own a lock the other wants! */
mutex_lock(&lb);                   mutex_lock(&la);
```

- **Each thread tries to acquire the lock the other has**
- **Solution: Obtain locks in the same order**
- **In-practice: deadlocks can include file locks, CVs, semaphores**
- **Queuexfr Problem was to get you to think about deadlocks**

Deadlocks: Solutions

- **Lock Ranking/Ordering**
  - Always enforce a consistent ordering among locks
  - Often developer builds enforces extra lock ranking violations
  - VMware ESX: static ordering maintained by developers
  - FreeBSD: *witness* kernel option dynamically monitors lock order
- **Deadlock Detection**
  - Often expensive to run in production
  - E.g. might run detection when locks are asleep too long
  - Clang/LLVM ThreadSanitizer finds deadlocks in developer builds
- **Lock Order for arrays of locks**
  - Choose lock order sorted by address or array indices

Mesa vs. Hoare CVs

- **WARNING: Use Mesa CVs in CS350**
- **Mesa CVs:**
  - `void cond_wait (cond_t *c, mutex_t *m);`
    - ▷ Atomically unlock `m` and sleep until `c` signaled
    - ▷ Then re-acquire `m` and resume executing
  - `void cond_signal (cond_t *c);`
    `void cond_broadcast (cond_t *c);`
    - ▷ Wake one/all threads waiting on `c`
- **Hoare CVs:**
  - `void cond_wait (cond_t *c, mutex_t *m);`
    - ▷ Sleep until `c` signaled, then resume
    - ▷ Lock `m` is sent to/recieved from signalling thread
  - `void cond_signal (cond_t *c, mutex_t *m);`
    `void cond_broadcast (cond_t *c, mutex_t *m);`
    - ▷ Wake one/all threads waiting on `c` and pass `m` to it.

Mesa vs. Hoare CVs Continued

- **Mesa CVs:**
  - Possible race in reacquiring lock
  - Shared state must be rechecked
  - Simple implementation
  - Used by most languages/operating systems
  - Either suffer from double sleep/wakeup or requires wait morphing
- **Hoare CVs:**
  - Lock passed around (no race)
  - Shared state does not need to be rechecked
  - Complex implementation (requires modifying lock code)
  - Used in many books
  - Thread wakes up immediately, no double sleep/wakeup

Barriers

- **Another primitive: wait for `N` threads to complete**
- **Useful to gate phases of a program/computation**
- `struct barrier_t;` (`pthread_barrier_t`)
- `void barrier_init (barrier_t *b, int N);`
- `void barrier_destroy(barrier_t *b);`
- `void barrier_wait (barrier_t *b);`
  - Wait for `N` threads to reach the wait
  - Then allow `N` threads to proceed

Read/Write Locks

- **Allows multiple readers/single writer**
- `struct rwlock_t;` (`pthread_rwlock_t`)
- `void rwlock_init (rwlock_t *b, ...);`
- `void rwlock_destroy(rwlock_t *b);`
- `void rwlock_unlock (rwlock_t *b);`
- `void rwlock_rdlock (rwlock_t *b);`
- `int rwlock_tryrdlock (rwlock_t *b);`
  - Acquire read lock
- `void rwlock_wrlock (rwlock_t *b);`
- `int rwlock_trywrlock (rwlock_t *b);`
  - Acquire write lock
  - Blocks new readers and waits for readers to complete

Monitors

- Monitors are essentially mutex locks
- Often contain a condition variable like feature
- Variations exist in most modern programming languages
- Less error prone than mutexes

lock_guard in C++11

- **Automatically acquires/releases a C++11** `mutex`
- **No CV-like functionality**

```
#include <thread>
#include <mutex>

int foo;
std::mutex myMutex;

void inc() {
    std::lock_guard<std::mutex> lock(myMutex);
    foo++;
}
```

Monitors in Java

- Per-object monitors
- Java doesn't use CVs
- Uses a single wait queue for locks and notify API
- Queue supports mutex locks and `notify`/`notifyAll`/`wait`
- C# uses a similar implementation

```
public class SynchronizedCounter {
    private int foo;
    public synchronized void inc() {
        foo++;
    }
    public void dec() {
        synchronized (this) {
            foo--;
        }
    }
}
```