Mutex Semantics, I

Even with "simple" lock there are some not-so-simple issues

What happens if ...

- Thread attempts to get lock that it (not another thread) already has?
- Thread drops lock it doesn't have?

Answers depend on mutex's "type"

Mutex Semantics, II

Possible mutex types:

- PTHREAD_MUTEX_NORMAL
- PTHREAD_MUTEX_ERRORCHECK
- PTHREAD_MUTEX_RECURSIVE
- PTHREAD_MUTEX_DEFAULT

PTHREAD_MUTEX_NORMAL man page says:

"Attempting to relock the mutex causes deadlock. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, undefined behavior results."

Mutex Semantics, III

`PTHREAD_MUTEX_ERRORCHECK` man page says:

"If a thread attempts to relock a mutex that it has already locked, an error shall be returned. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error shall be returned."

`PTHREAD_MUTEX_RECURSIVE` man page says:

"the mutex shall maintain ... a lock count ... Every time a thread relocks this mutex, the lock count shall be incremented by one. Each time the thread unlocks the mutex, the lock count shall be decremented by one. When the lock count reaches zero, the mutex shall become available for other threads to acquire. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error shall be returned."

Mutex Semantics, IV

`PTHREAD_MUTEX_DEFAULT` man page says:

"... attempting to recursively lock the mutex results in undefined behavior. Attempting to unlock the mutex if it was not locked by the calling thread results in undefined behavior. Attempting to unlock the mutex if it is not locked results in undefined behavior."

Mutex Semantics, IV

Q: Why so many possibilities for deadlock and undefined behavior?

A: "Rationale" section of man page says:

"Mutex objects are intended to serve as a low-level primitive from which other thread synchronization functions can be built. As such, the implementation of mutexes should be as efficient as possible, and this has ramifications on the features available at the interface.

The mutex functions and the particular default settings of the mutex attributes have been motivated by the desire to not preclude fast, inlined implementations of mutex locking and unlocking.

For example, deadlocking on a double-lock is explicitly allowed behavior in order to avoid requiring more overhead in the basic mechanism than is absolutely necessary."

Read/Write Lock Semantics

Common to have many readers and very few writers

Also common: long-lived readers and short-lived writers

With a read/write lock it is OK simultaneously to have:

- Any number of readers, or
- 1 writer

Pthreads Read/Write Locking Calls

`pthread_rwlock_rdlock`

`pthread_rwlock_tryrdlock`

`pthread_rwlock_timedrdlock`

`pthread_rwlock_wrlock`

`pthread_rwlock_trywrlock`

`pthread_rwlock_timedwrlock`

`pthread_rwlock_unlock`

Better names than "read/write" would be "shared/exclusive"

Tricky Cases

From `pthread_rwlock_wrlock` man page:

"The calling thread may deadlock if at the time the call is made it holds the read-write lock (whether a read or write lock)."

From `pthread_rwlock_rdlock` man page:

"The calling thread may deadlock if at the time the call is made it holds a write lock."

Note the word "may" – means it is implementation-dependent whether attempt to get a lock that's already held is (1) OK or (2) causes thread to deadlock with itself

Implementation Gotcha, I

Beware the optimizing compiler!

Consider this program, written in a high level language:

```
stmt A
stmt B
stmt C
```

Suppose those statements compile to these machine instructions:

```
A.1
A.2
B.1
B.2
C.1
C.2
```

Implementation Gotcha, II

Now consider this high level code:

```
pthread_mutex_lock
critical section
pthread_mutex_unlock
```

UNoptimized may compile to:

```
pthread_mutex_lock.1
pthread_mutex_lock.2     // this instruction gets lock
critical section.1
critical section.2
pthread_mutex_unlock.1
pthread_mutex_unlock.2
```


Implementation Gotcha, III

OPTIMIZED may compile to:

```
pthread_mutex_lock.1
critical section.1
pthread_mutex_lock.2     // this instruction gets lock
critical section.2
pthread_mutex_unlock.1
pthread_mutex_unlock.2
```

Beware: sometimes observe different behavior in code compiled at highest optimization level!

Some vendors sell compiler+library package where compiler knows which calls lock/unlock – optimization does not move instructions around those calls

Other higher-level Solutions

All implemented in language, library, or OS; all depend on locking at lower level

1. Semaphore
2. Monitor
3. Barrier
4. others ...

Q: Why needed—why isn't locking enough?

A: Busy-wait loop is very wasteful of execution time—these more sophisticated mechanisms put a wait-er to "sleep"

Semaphore

Semaphore = integer variable +
set of sleeping threads +
policy for removing from set +
two ATOMIC operations, P and V

This is a **counting semaphore**
(as opposed to **binary semaphore**)

P: "wait"
from Dutch *proberen te verlagen*,
"try to decrease"

V: "signal"
from Dutch *verhogen*, "increase"

Semaphore Semantics

```
P(val):

val--;
if (val < 0)
    then go to sleep


V(val):

val++;
if (val <= 0)
    then wakeup some sleeper
```

Both P and V are atomic — atomicity implemented using some underlying lock mechanism

Semaphore Value Conveys Information

if `val` < 0 (after P finishes or before V starts) ...

then $|$`val`$|$ threads are waiting

Semaphore Wakeup Semantics

Q: Which sleeper should V wake up?

A: Best left *unspecified*

If wakeup policy is specified, some programmers will write code that depends on it – then, if policy is changed, their code breaks

Alternate Definitions

Beware: there are other definitions of P and V

With Linux `sem_wait` and `sem_post` `val` never goes below zero

```
sem_wait(val):

if (val > 0) {
    val--;
} else if (val == 0) {
    go to sleep  // will be awakened when val > 0
    after awakened: val--
}


sem_post(val):

val++;
if (val becomes > 0)
    then wakeup some sleeper
```

Classic Synchronization Problems

1. Mutual exclusion
2. Producer/consumer
   - FIFO; one thread inserts, other deletes
3. Readers/writers
   - generalization of mutual exclusion
   - may have several readers or 1 writer (but not both) simultaneously in critical section
4. Dining philosophers
   - N threads, N resources
   - each thread must acquire 2 resources

Semaphore Example I: Mutual Exclusion

Each critical section written like this:

```
P(mutex);
<critical section>
V(mutex);
```

`mutex` initialized to 1

Thread passes thru P and enters critical section ONLY if `mutex` is decremented from 1 to 0; otherwise go to sleep & wait for some later V to cause wakeup

Example

One possible execution involving 4 threads:

```
0. mutex is initially 1 & no thread is in CS
1. Thread A executes P
[mutex now 0, thread A in CS]
2. Thread B executes P
[mutex now -1, thread A in CS, thread B asleep]
3. Thread C executes P
[mutex now -2, thread A in CS, threads B and C asleep]
4. Thread D executes P
[mutex now -3, thread A in CS, threads B, C, and D asleep]
5. Thread A executes V
[mutex now -2, no thread in CS, threads B, C, and D asleep]
6. Thread C selected for wakeup
[mutex now -2, thread C in CS, threads B and D asleep]
7. Thread C executes V
[mutex now -1, no thread in CS, threads B and D asleep]
8. Thread D selected for wakeup
[mutex now -1, thread D in CS, thread B asleep]
9. Thread D executes V
[mutex now 0, no thread in CS, thread B asleep]
10. Thread B selected for wakeup
[mutex now 0, thread B in CS]
11. Thread B executes V
[mutex now 1, no thread in CS]
```