Lecture C6: Multi-object synchronization and Deadlock

****

Review -- 1 min

****

Good news: we have a systematic way to write small synchronized programs

Application:	readers/writers	bounded_buffer	…	**invariants**
Abstractions:	semaphores	monitors		**mutex + scheduling**
Hardware:	test&set	interrupts off		**atomic read-modify-write**

advice: follow a consistent methodology

****

Outline - 1 min

****

What about larger programs? Programs with multiple shared objects?

2 problems: Safety, liveness

Safety: Design patterns

Liveness: Deadlock

- definition
- conditions for its occurrence
- solutions: breaking deadlocks, avoiding deadlocks
- efficiency v. complexity

Other hard (liveness) problems

  - priority inversion
  - starvation
  - denial of service

These problems are hard because whereas we were able to structure programs so that safety became a local property (e.g., we have modularity), these liveness issues have to do with global structure of program (e.g., no modularity)

The good news is that these problems are usually not as dangerous as safety bugs. As opposed to "intermittent bug", "The program stops with the evidence intact" [Lampson] (Usually not so bad; but occasionally catastrophic. Example: Mars Pathfinder.)

****

Preview - 1 min

****

catch up -- read scheduling policy; will not say much in lecture. May start memory virtualization on Thursday.

midterm
break
file systems
NOTE: Not sure if we will talk about scheduling policy…

****

Lecture - 20 min

****

1. Problems with threads

We've solved a really hard problem: how to safely coordinate access to a shared resource
Monitors give us a systematic, modular approach

This works great for problems that fit on a blackboard

Unfortunately there are other problems to threads programming that primarily arise in larger-scale programs.
We've shown how to coordinate actions *within* an object or module.
The challenge is to coordinate actions *across* modules.

2 issues
(1) safety: multi-object synchronization
(2) liveness: deadlock

Two problems where threads "break modularity" (literally, when one module calls into another, it has to know about the internal implementation details and make sure that both modules' synchronization mesh.)

1.1.1 The case against threads

Several prominent operating systems researchers have argued that one should almost never use threads because (a) it is just too hard to write multi-threaded programs that are correct and (b) most things that threads are commonly used for can be accomplished in other, safer ways.

I think they may go too far, but there is more than a grain of truth in their arguments.

The class web page has pointers to two documents that may interest you:

John Ousterhout "Why Threads Are A Bad Idea (for most purposes)."

Robert van Renesse "Goal-Oriented Programming, or Composition using Events, or Threads Considered Harmful"

These are important arguments to understand -- even if you disagree with them, they may point out pitfalls that you can avoid.

2. Multi-object synchronization

a->subtract($100)
b->add($100)

Even if individual actions atomic, sequence is not.

variation of same problem: **fine grained synchronization** within an object

Back to too much milk?

2.1 Example solution 1: Careful class design

-- Advantage over too much milk -- you can design API; you don't just need to expose atomic load/store

e.g., Apartment::checkForMilkAndSetNoteIfNeeded()

(DA: Still requires careful reasoning about how classes/objects interact)

2.2 Example solution 2: Serialization

Divide work into "tasks" (each a separate logical chunk of work)

Ensure that execution of set of tasks always produces a **serializable execution**

serializable execution -- an execution where tasks may execute concurrently, but where the result of each task is equivalent to the result that would have occurred if the tasks were executed one at a time in some serial order.

--> ensuring serializability allows one to reason about multi-step tasks as if each task executed alone.

ways to get serialization
(a) one big lock (trivial)

obvious -- gets serializability

DA: lock *everything* for long period of time. No concurrency.

(b) lock-all/release all

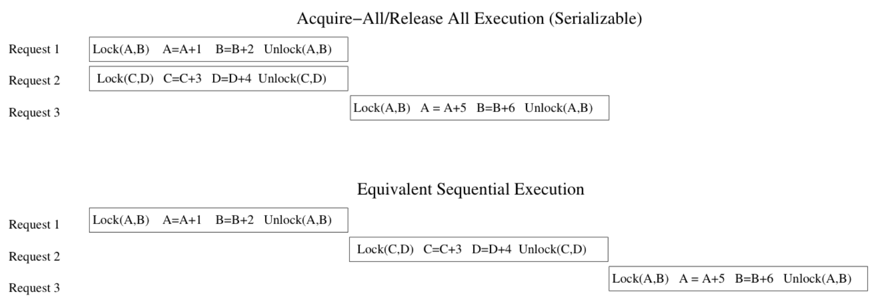

Acquire−All/Release All Execution (Serializable)
Request 1
Lock(A,B) A=A+1 B=B+2 Unlock(A,B)
Request 2
Lock(C,D) C=C+3 D=D+4 Unlock(C,D)
Request 3
Lock(A,B) A = A+5 B=B+6 Unlock(A,B)
Equivalent Sequential Execution
Request 1
Lock(A,B) A=A+1 B=B+2 Unlock(A,B)
Request 2
Lock(C,D) C=C+3 D=D+4 Unlock(C,D)
Request 3
Lock(A,B) A = A+5 B=B+6 Unlock(A,B)

Figure 5.2: Locking multiple objects using an acquire-all/release-all pattern results in a serializable execution that is equivalent to an execution where requests are executed sequentially in some order.

(c) two phase locking

Two phase locking
phase 1: Acquire locks (includes upgrading reader lock to writer lock)
phase 2: release locks (includes downgrading
[relate back one big lock]

allows concurrency
[e.g., 10 buckets, transfer A->B grabs A, grabs B, does transfer, releases B, releases A]

claim: any execution under two phase locking must be serializable

Summary -- serializability
+ Close to "atomic" semantics
- Limited -- Appropriate for lots of short tasks
  -- Cumbersome/limits program structure
- Conservative -- may limit concurrency/performance

[[Weakenings: "snapshot isolation", "repeatable reads", etc. Known anomolies. I don't now if there is a clean way to convince oneself that a particular weakening will be "OK" .... [but perhaps there is. I just don't know.]

2.3 Example solution 3: Ownership pattern

e.g., shared container (e.g., **hash table**) -- put things in, take them out (own them)

e.g., **work queue**

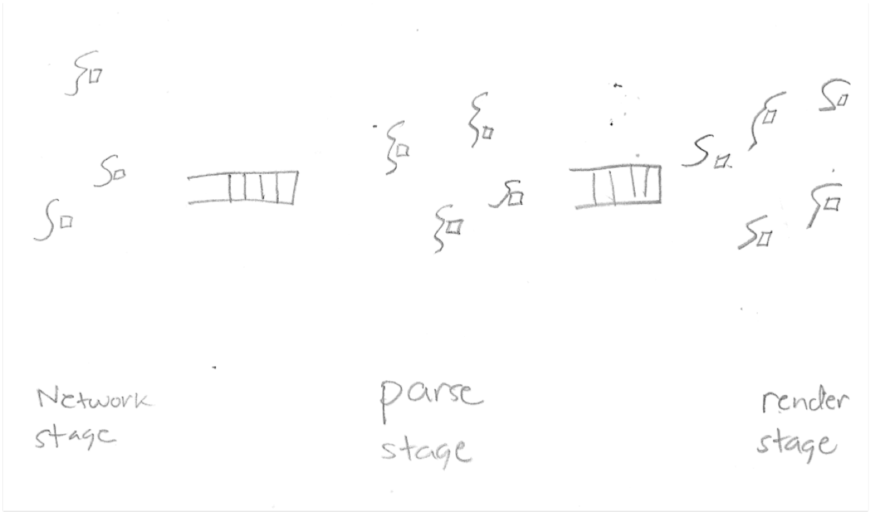


Figure 5.1: A multi-stage server based on the ownership pattern. In the first stage, one thread exclusively owns each network connection. In later stages, one thread is processing a given object at a time.

2.4 Example solution 4: Staged architectures

Each stage has local state and some threads that operate on it. No state shared across stages (except messages between them -- ownership pattern here...)

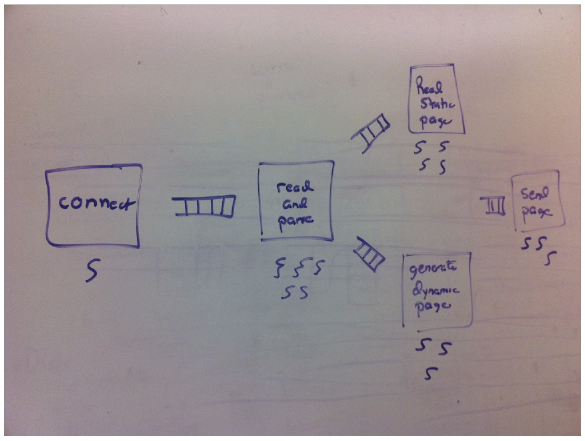


Figure 5.3: A staged architecture for a simple web server.

special case: **event processing** -- one thread per stage --> no locking needed

2.5 Example solution 5: Misc

Answers here not as cookbook as monitors/shared objects

People convince themselves that their programs work...

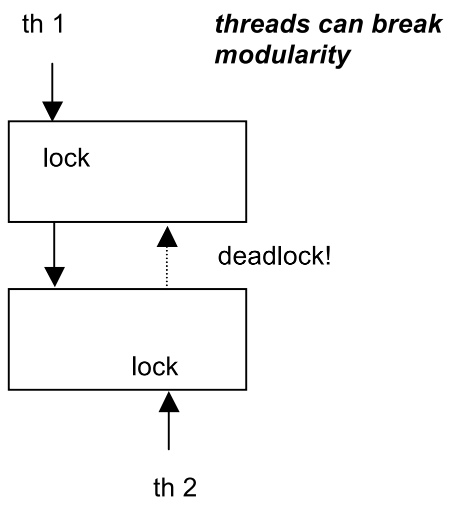
Are they informally convincing themselves that they do 2 phase locking/ownership/...? Are there other general answers? I need to think about this more...

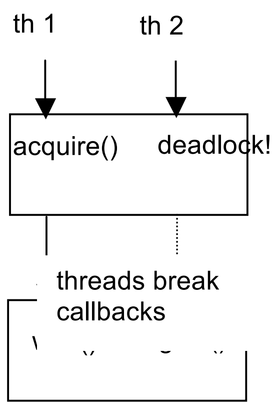
3. Deadlock: Definitions

Example: Dining lawyers

Deadlocks break modularity.

Two problems where threads "break modularity" (literally, when one module calls into another, it has to know about the internal implementation details and make sure that both modules' synchronization mesh.)





Warning: by the strictest definition, this case is not quite a deadlock. See appendix to today's notes.

threads break callbacks

3.1 Resources

threads – active; a schedulable execution context

resources – passive; things needed by thread to do its job (e.g. CPU, disk space, memory)

2 kinds of resources
Preemptable – can take it away (CPU)
Non-preemptable – must leave with thread
e.g. disk space – what would you think if I took space away from your files?

Lock/Mutual exclusion – a kind of resource
represents a set of data that a thread needs exclusive access to to do a job
QUESTION: is a lock pre-emptable or non-preemptable?

3.2 Starvation v. deadlock

starvation – thread waits indefinitely
(e.g. because some other threads are using resources)

deadlock – circular waiting for resources in which waiting threads cannot change state because the resources they have requested are held by other waiting threads

Deadlock implies starvation, but not vice versa

Deadlock example

```
Thread A          Thread B
x.Acquire();      y.Acquire();
y.Acquire();      x.Acquire();
```

4. Conditions for Deadlock

4.1 Motivation

- Deadlock can happen with any kind of resource
- Deadlocks can occur with multiple resources. Means you can't decompose the problem – can't solve deadlock for each resource independently

For example

- one thread grabs the memory it needs
- another grabs disk space
- another grabs the tape drive

each waits for the other to release

Deadlock can occur whenever there is waiting
Example: dining lawyers

Each lawyer needs two chopsticks to eat. Each grabs chopstick on the right first

What if all grab at same time? Deadlock.

4.2 Conditions

Conditions for deadlock – without **all** of these, can't have deadlock:

1) limited access (for example, mutex or bounded buffer)
2) no preemption (if someone has resource, can't take it away)
3) multiple independent requests ("wait while holding")
4) circular waiting

****
Admin - 3 min
****

****
Lecture - 33 min
****

5. Solutions to deadlock

- Detect & fix
- Avoid

5.1 Detect deadlock and fix

```
scan graph
detect cycles
fix them          // this is the hard part
```

5.1.1 Detecting deadlock

No cycles → no deadlock exists

Cycle → deadlock **may** exist

- If one instance of each resource both necessary and sufficient condition
- If multiple instances, necessary condition, but not sufficient

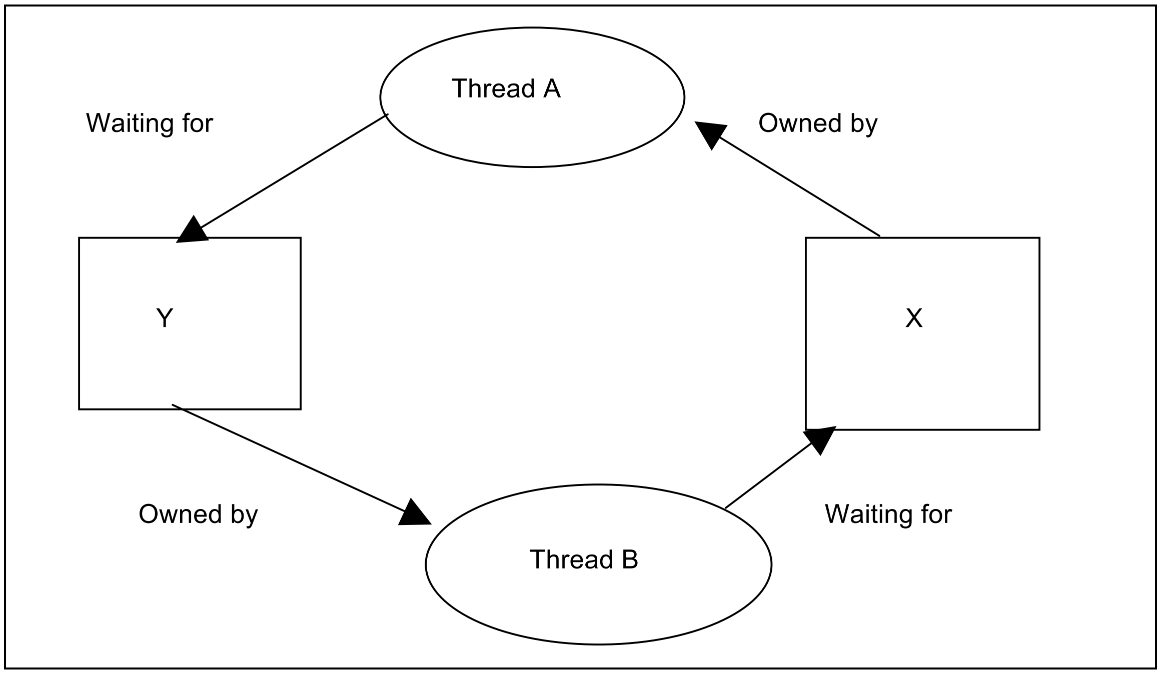
5.1.1.1 Resource allocation graph

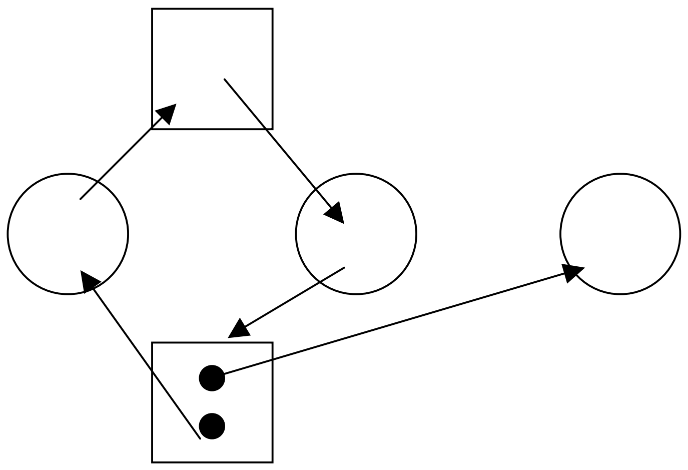
Square = resource

Multiple resources represented w/ multiple dots in square

Circle = thread

Arrows show dependency – “owned by”, “waiting for”





5.1.2 Ways to fix deadlock

Once you're in a deadlock, need to revoke resources to fix it

1) shoot thread; force it to give up resources
This isn't always possible – for instance, with a mutex, can't shoot a thread and leave the world in a consistent state

2) Roll back actions of deadlocked threads "transactions"

common database technique

DA: roll back work you've already done → inefficient?
DA: keeping state to allow roll back may involve overhead

5.2 Preventing deadlock

Key idea: Need to get rid of one of the four conditions

Warning: DA's – none of these are general; the more general ones tend not to be so simple or may significantly under-utilize resources (e.g., be too careful)

Example – avoiding deadlock in general is hard. Consider case with 3 resources A, B, C and 2 threads that access them: 1: ACB, 2: BCA

Thread 1 Thread 2

Grab A	**Grab B**
Grab C	wait for C
Wait for B	(A)

You could detect that when thread 1 grabs C it causes a deadlock, so don't let it grab C, but by then it's too late. In fact, you had to be smart enough to see deadlock coming 1 step earlier (once thread 1 grabs A, then we can't let thread 2 grab B!)

1) infinite resources
solves "limited access"

aka: ostrich algorithm...

2) No sharing – totally independent threads
solves ???

3) Don't allow waiting – how phone company avoids deadlock
solves ???

4) Preempt resources
example – can preempt main memory by copying to disk
solves??

5) Order resources
e.g., never grab lock A after grabbing lock B
common approach in programs – partial order across all locks
Make everyone use the same ordering in accessing resources

For example, all threads must grab locks in same order

```
x.Acquire()          x.Acquire()
y.Acquire()          y.Acquire()
```

Note: this works for locks. Does it work if a call to module Y can wait()?

6) make all threads request everything they'll need at the beginning
e.g. if you need 2 chopsticks grab both at same time (or don't grab any)
solves???

problem – predicting future is hard; tend to over-estimate resource needs (inefficient) (of course under-estimation leads to deadlock)

6) **banker's algorithm –** more efficient than reserving all resources on startup (due to Dijkstra)

Banker's algorithm allows the sum of maximum resource needs of all current threads to be greater than the total resources, as long as there is some way for all the threads to finish without getting into deadlock

a) state maximum resource needs in advance
b) allocate resources dynamically when resource is needed; wait if granting request would lead to deadlock (request can be granted if some sequential ordering of threads is deadlock free)

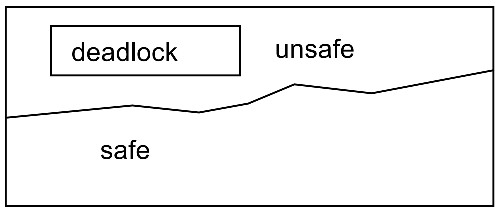
5.2.1 Key concept: safe state

safe state -- there exists some ordering of resource grants that guarantees all processes can complete w/o deadlock (e.g., OS can guarantee no deadlock will occur by granting resources in proper order)

If the system is in a safe state, then there exists a **safe sequence**

E.g., there is some ordering of processes 0..i s.t. job[0] can complete using the resources it has + available system resources; job[i] can complete with resources it has + available system resources + resources held by jobs[0..i-1] a "**safe sequence**"

Note: not all unsafe states must lead to deadlock -- (e.g., the applications could end up asking for fewer resources than they had originally planned to ask for)



All deadlock states are unsafe, but not all unsafe states are deadlocks.

OS can guarantee what happens in safe states. Process behavior determines what happens to unsafe states. --> so if OS wants to guarantee no deadlock, it can not let system into an unsafe state!

Idea – applications specify maximum possible resource demands
OS sees series of "aquire/release" resource
All OS can do to avoid deadlock is delay some of the requests
→ OS can control order that different applications progress
→ OS makes sure that at least one process can complete, then that a second one can complete, …

Note that OS must treat application as black box (or adversary) – must be conservative
(just b/c applications enter "unsafe state" doesn't mean a deadlock will occur, but OS can't take that chance…)

5.2.2 Algorithm:

```
//
// Invariant: the system is in a safe state
//
ResourceMgr::Request(ResourceID resource,
                     RequestorID thread){
     mutex.acquire();
     assert(system is in a safe state);

     while(the state that would result from
           giving resource to thread is not safe){
            cv.wait(&mutex);
     }
     update state by giving resource to thread
     assert(system is in a safe state);
     mutex.release();
}
```

Now the trick is: how can you tell if a state is safe?
➔ Determine if there is a safe sequence from the state

Each process states its max needs

```
Max[i,j] – max resource j needed by process i
Alloc[i,j] – current allocation of resource j
             to process i
Need[i,j] = Max[i,j] – Alloc[i,j]
Avail[j] – number of resource j available

TestSafe(Max[], Alloc[], Need[], Avail[]){
    Work[] = avail[]
    Finish[] = 0,0,0,… // Boolean; is process i finished?

    repeat{
        find i s.t. finish[i] = false and need[i] < work
        if no such i exists
           if finish[i] = true forall i return true
           else return false
        else
           work = work + alloc[i]
           finish[i] = true
    }
```

Example of Banker's algorithm with dining lawyers: chopsticks in middle of table
Deadlock free if when try to grab fork, take it unless it's the last one, and no one would have 2

What if k-handed lawyers?
Deadlock free if when try to grab fork: take it unless
 its the last one and no one would have k
 its the next to last one, and no one would have k-1
 …

Typically, a combination of techniques

5.3 Prudent engineering

If you are writing a large multi-threaded program

Consider overall program structure carefully. If possible:

- Use coarse grained locking ("one big lock" is often the right answer).
- Disciplined hierarchical structure (so you can order the locks); avoid up-calls

If your structure is poor, you have little hope.

Pairwise deadlock case 1: mutual waiting within monitor

- Lampson and Redell "Experience with Processors and Monitors in Mesa": "Localized bug in the monitor code…usually easy to locate and correct"

Pairwise deadlock case 2: Lock cycle across monitors

- Simplest solution: partial ordering across resources
- --> structure program to avoid mutually recursive monitors; avoid callbacks; avoid upcalls

Pairwise deadlock case 3: Nested monitors + wait

- LR: "Break [monitor] M into two parts: a monitor M' and an ordinary module O which implements the abstraction defined by M and calls M' for access to shared data. The call on [nested monitor] N must now be done from O rather than from within M"

Note: solutions to cases 2 and 3 break modularity and are not general

- They require knowledge of internals of other modules. *Can this module call me? Can this module call a module that calls me? Can this module wait?*
    - Target of call: no lock-->OK. Caller can continue to hold lock
    - Target of call: locks but never waits --> caller can continue to hold lock if partial ordering exists (e.g., if calee never calls back or higher)
    - Target of call locks and may wait --> dangerous to call while holding a lock
- Proposed rule: Manually release lock when calling another module
    - Still follow rule: release lock only at beginning/end of procedure
        - -->Wrapper procedures?
        - --> continuation style of programming?
        - Be careful not to assume anything stronger than invariant upon re-entry (danger: "implicit" reasoning based on "program counter")
    - This approach still requires careful thought and code structure (Andrew Birrell "Guide to programming with threads": "You should generally avoid holding a mutex while making an up-call (but this is easier said than done.)"
- Exceptions to rule:
    - **Callee uses no locks OR callee uses no condition variables and partial order exists**
    - Manually verify and hope invariant continues to hold?
    - Syntactic sugar (e.g., similar to `const`?)
    - Use a debugging version of lock, condition variables that detects "dangerous" patterns at run time?
    - Other exceptions? When is it safe to call a method that might wait?

- How could programming tools help?
    - Language/compiler: "wontblock/mightblock" notation on method calls
    - Compiler/static checker – order of lock acquire
    - Runtime – track lock acquire order
    - Runtime – detect wait while holding other locks
    - …

6. Priority inversion

A related problem. Suppose thread A has high priority, thread B has medium priority, and thread C has low priority.
Then thread C acquires a lock
Thread A attempts to acquire the lock
Thread B is busy using the CPU

A waits for C
C waits for B

A is being delayed by a lower priority process?

Seems innocuous. This is why the Mars Pathfinder rover (Sojourner) took several days to get started.

Well known, common problem.

Solution
If C holds a lock and A is waiting on the lock, temporarily boost C's priority to A's (e.g., when I hold the lock, my priority is the max(priority of all threads waiting on the lock)

Note: this increases complexity of building locks
****

Summary - 1 min
****

7. What's hard about threads programming?

We started off with what seemed like a really hard problem, but came up with a reasonable solution (synchronization via monitors, etc.) What's the big deal?

In class we look at problems that fit on a blackboard. In life you have to deal with 100K-10M line programs. It makes a difference. Deadlock is one example – problems come from interactions among different critical sections.

"Doc, it hurts when I do this"
"Don't do that."
If you master the basics I've described, then 3 biggest challenges
(1) deadlock
(2) performance v. complexity -- temptation to do complex, fine-grained locking
[see below]
(3) performance tuning

For (1) and (2), I've found that the answer is high level design. Refactor until high level design is easy to reason about, easy to synchronize (rather than trying to do crazy complex things within bad design.)

For (3) -- hard. Can be difficult to identify culprit, let alone fix. (Tempted to say answer is, again, high level design. Often hard to get more than 2x, 4x, 8x on multi-threaded code (even on 32x processor). *real* speedups come when you slice program into independent pieces (e.g., mapreduce)... Still mulling how true this is...)

7.1 Performance v. complexity (correctness)

One big lock you hold for entire operation (simple, but slows you down)
v.
finer-grained locking (potentially faster, but more complex. More dangerous)

Example: hash table with conecurrent access
Option: one lock per table
    One lock per table + one lock per bucket in table
    One lock per table + one per bucket + one per element
Consider lock/unlock pattern for an operation like insert…

7.2 Synchronization bugs

Don’t hold/release locks when you should

Hidden sharing across modules:

e.g. – when a thread calls a library (e.g., printf, malloc) how do you know if you need to grab a lock?

(general solution is callee should use locks if it needs it, but that may add overhead for single-threaded programs)

Not protect all shared variables properly

e.g., performance v. complexity debate – as more clever fine-grained locking, increase chance to screw up

e.g., when port kernel to be multi-threaded, usually start with “one big lock” on entire kernel, then in next release per-module locks (with care to avoid pitfalls), then within module, etc.

Etc

Example

```
1)
P(s)            P(s)
…
V(s)            V(s)
…
V(s)

2)
lock(m)         a++
a++
unlock(m)

3)
lock(m)
…
unlock(n)

4)
lock(m)
…
if(…)
```

```
  return
…
unlock(m)
return;
```

Heisenbugs

Synchronization bugs are hard to detect and correct b/c hard to reproduce ("Heisenbugs" v. "Bohr bugs")

7.3 Deadlock

See above

Really a big problem in large systems – subsystem 1 calls subsystem 2 calls… How to enforce order of locking or whatever

Many systems built with callbacks – almost invites cycles

7.4 Priority inversion (see above)

7.5 Starvation

If synchronization solution not well implemented a thread may starve (e.g., semaphore implemented in LIFO order)

7.6 Denial of service problems

Examples

- CS doesn't ensure progress
- Thread crashes in middle of CS
- Thread gets caught in infinite loop in CS
- Thread does not clean up after itself

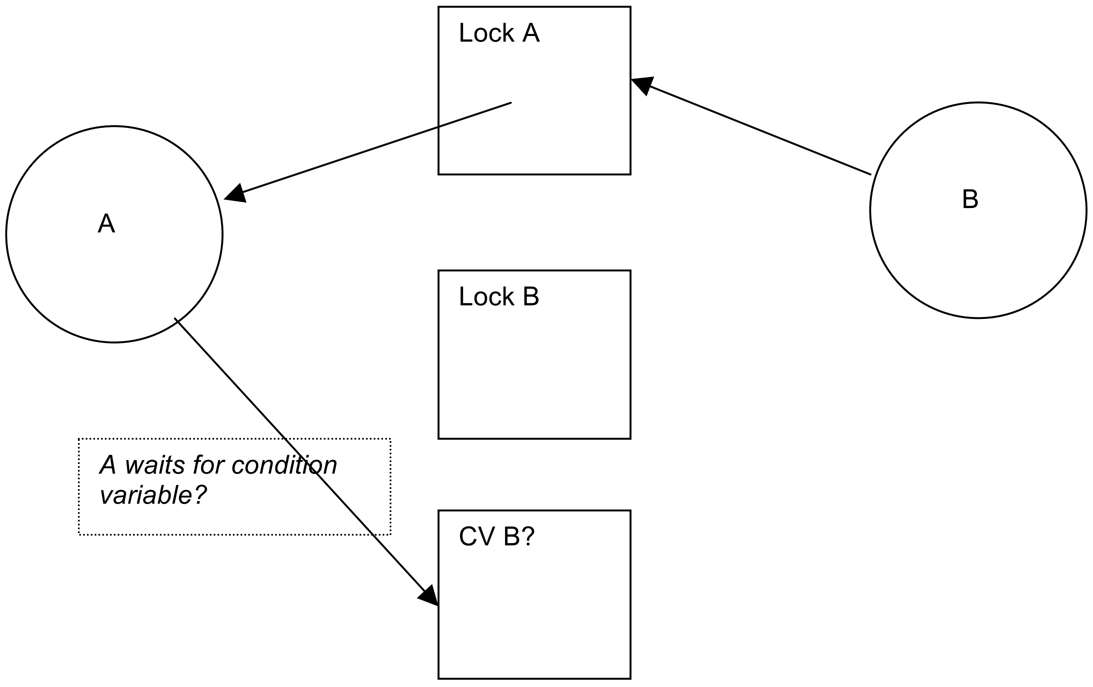
8. Appendix: Deadlock v. circular wait

Some definitions of deadlock hold that the "lock" is literal. That deadlock is what you have when you have circular waiting for locks/exclusive access to resources and that circular waiting that includes monitors can cause starvation but that they are not strictly speaking deadlocks.

I (and others such as Lampson and Redell, quoted above) find it more convenient to talk about both cases of circular waiting as deadlocks.

To see why the other point of view has some merit, consider the pairwise "deadlock" case of two threads calling though module A into module B. Thread 1 waits in module B (while holding A's lock) and waits for thread 2 to signal in module B; but thread 2 is stuck waiting for A's lock.

This certainly seems like deadlock. But try to draw the "waits for" graph. The final state



- Where is the cycle?
- This tool doesn't quite work for this case; this boxes, circles, and arrows tool (and related graph algorithms) work for lock-only deadlock, but not for mixed lock/cv deadlock.
- There still is a circular dependency. To stretch the point, B "holds" the signal that A "waits for" (so we could sort of add an arrow from CVB to B?). But, because condition variables capture higher level, more general scheduling constraints than

locks, it is not so easy to automate detection of cycles through condition variables (depends on program meaning.)