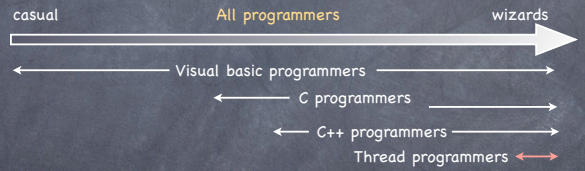
Advanced Synchronization and Deadlock

A house of cards?

- Locks + CV/signal a great way to regulate access to a single shared object…
- …but general multi-threaded programs touch multiple shared objects
- How can we atomically modify multiple objects to maintain
  - Safety: prevent applications from seeing inconsistent states
  - Liveness: avoid deadlock
    - a cycle of threads forever stuck waiting for one another

Contra Threads: Events

- John Ousterhout: "Why Threads Are a Bad Idea (for most purposes)"



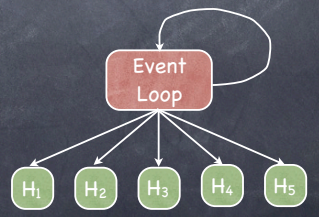
Contra Threads: Events

- John Ousterhout: "Why Threads Are a Bad Idea (for most purposes)"

casual
All programmers
wizards
Visual basic programmers
C programmers
C++ programmers
Thread programmers

Event-driven Programming

- No concurrency: one execution stream
- Register interest in events (callbacks)
- Wait for events; invoke (short-lived) handlers
- Complicated only for unusual cases
- Easier to debug



Multi-object synchronization

- Transfer $100 from account A to account B
  - A->subtract(100);
  - B->add(100);

  } Individual operations are atomic
  Sequence is not
- Fine-grain locking
  - Hash table:
    - put(key, value) value = get(key) value = remove(key)
    - one lock for whole table? one lock per bucket?
- Complexity vs Performance
  - Beware of premature optimizations!

Solutions: Careful class design

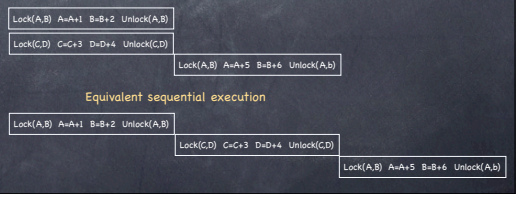
- You design the API!
  - Too Much Milk with 2 objects
    - Fridge Fridge::checkForMilk(); Fridge::addMilk()
    - Note Note::readNote(); noteWriteNote()
  - back to square one...
  - Instead
    - Fridge::checkForMilkAndSetNoteIfNeeded()
    - Fridge::addMilk()
- No panacea
  - still need to think carefully how objects interact

Solutions: Serialization

- Divide work into logically separate tasks
- Ensure serializable execution of tasks
  - tasks may execute concurrently...
  - ...but result of each task equivalent to what would be obtained if tasks executed one at a time in some serial order
- A few ways to get there
  - one big lock
  - lock-all/release-all
  - two phase locking

Solutions: Serialization

- Divide work into logically separate tasks
- Ensure serializable execution of tasks
  - tasks may execute concurrently...
  - ...but result of each task equivalent to what would be obtained if tasks executed one at a time in some serial order
- A few ways to get there
  - one big lock
  - lock-all/release-all
    - need to know all locks
  - two phase locking



Solutions: Serialization

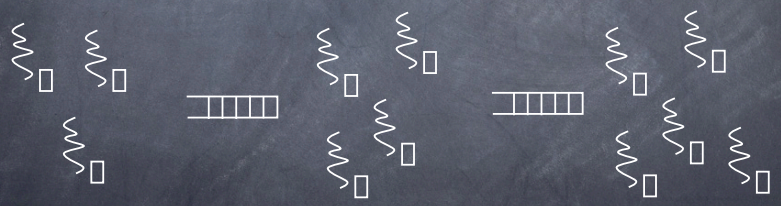
- Divide work into logically separate tasks
- Ensure serializable execution of tasks
  - tasks may execute concurrently...
  - ...but result of each task equivalent to what would be obtained if tasks executed one at a time in some serial order
- A few ways to get there
  - one big lock
  - lock-all/release-all
  - two phase locking
    - serializable

Phase 1
- acquire locks
  - upgrade reader to writer lock if necessary

Phase 2
- release locks
  - downgrade writer to reader lock if necessary

Solutions: ownership pattern

- Shared container
  - put things in; take them out; access them without a lock (own them)



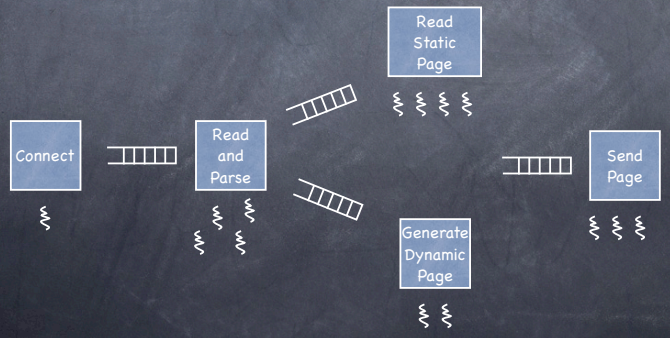
Network Stage
One thread/network connection

Parse Stage
One thread/object

Render Stage
One thread/object

Solution: staged architecture

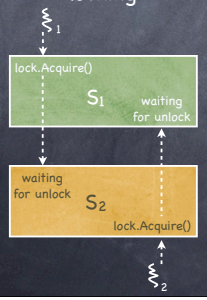
- Each stage has local state and some thread that operate on it
- No state shared across stages



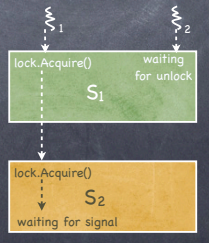
Deadlock

- A cycle of waiting among a set of threads, where each thread is waiting for some other thread in the cycle to take some action

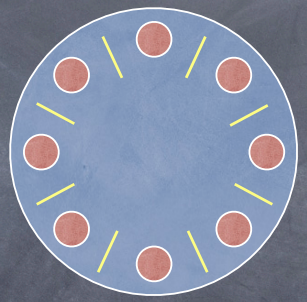
Mutually recursive locking



Nested waiting



Dining Philosophers

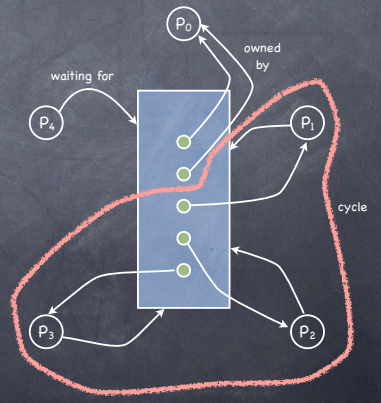


- N philosophers; N plates; N chopsticks
- If all philosophers grab right chopstick
  - deadlock!

Necessary conditions for deadlock

- Deadlock only if the all hold
  - Bounded resources
    - A finite number of threads can use a resource; resources are finite
  - No preemption
    - the resource is mine, MINE! (until I release it)
  - Wait while holding
    - holds one resource while waiting for another
  - Circular waiting
    - T_i waits for Ti+1 and holds a resource requested by T_{i-1}
    - sufficient if one instance of each resource

Not sufficient in general



Preventing deadlock

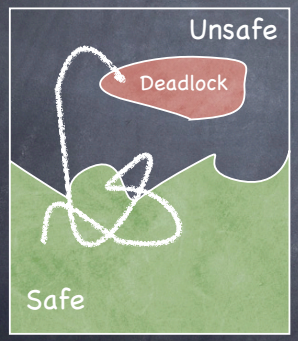
- Remove one of the necessary conditions
  - Provide sufficient resources
    - Removes "Bounded resources"
  - Preempt resources
    - Removes "No preemption"
  - Abort requests
    - Removes "Wait while holding"
  - Atomically acquire all resources
    - Removes "Wait while holding"
  - Lock ordering
    - Removes "Circular waiting"
    - Nested waiting?

Avoiding Deadlock: The Banker's Algorithm

E.W. Dijkstra & N. Habermann

- Sum of maximum resources needs can exceed the total available resources
  - if there exists a schedule of loan fulfillments such that
    - all clients receive their maximal loan
    - build their house
    - pay back all the loan
- More efficient than acquiring atomically all resources

Living dangerously: Safe, Unsafe, Deadlocked



A system's trajectory through its state space

- Safe: For any possible set of resource requests, there exists one safe schedule of processing requests that succeeds in granting all pending and future requests
  - no deadlock as long as system can enforce safe schedule
- Unsafe: There exists a set of (pending and future) resource requests that leads to a deadlock, for any schedule in which requests are processed
  - unlucky set of requests can force deadlock
- Deadlocked: The system has at least one deadlock

The Banker's books

- Max_{ij} = max amount of units of resource R_j needed by P_i
  - $MaxClaim_i = \sum_{j=1}^{m} Max_{ij}$
- $Alloc_{ij}$ = current allocation of R_j held by P_i
  - $HasNow_i = \sum_{j=1}^{m} Alloc_{ij}$
- $Avail_j$ = number of units of R_j available

- A request by P_k is safe if there is schedule $P_1, P_2, \ldots P_n$ such that, for all P_i, assuming the request is granted,

$$MaxClaim_i - HasNow_i \le Avail + \sum_{j=1}^{i-1} HasNow_i$$

An Example

- 5 processes, 4 resources

	Max				Alloc				Avail			
	R_1	R_2	R_3	R_4	R_1	R_2	R_3	R_4	R_1	R_2	R_3	R_4
P_1	0	0	1	2	0	0	1	2	1	5	2	0
P_2	1	7	5	0	1	0	0	0				
P_3	2	3	5	6	1	3	5	3				
P_4	0	6	5	2	0	6	3	2				
P_5	0	6	5	6	0	0	1	4				

- Is this a safe state?

An Example

- 5 processes, 4 resources

	Max				Alloc				Avail				MaxRequest			
	R_1	R_2	R_3	R_4	R_1	R_2	R_3	R_4	R_1	R_2	R_3	R_4	R_1	R_2	R_3	R_4
P_1	0	0	1	2	0	0	1	2	1	5	2	0	0	0	0	0
P_2	1	7	5	0	1	0	0	0					0	7	5	0
P_3	2	3	5	6	1	3	5	3					1	0	0	3
P_4	0	6	5	2	0	6	3	2					0	0	2	0
P_5	0	6	5	6	0	0	1	4					0	6	4	2

- Is this a safe state? P_1, P_4, P_2, P_3, P_5
  - While safe sequence does not include all processes:
    - Is there a P_i such that $MaxRequest_i \le Avail$?
      - if no, exit with unsafe
      - if yes, add P_i to the sequence and set Avail = Avail + $HasNow_i$
  - Exit with safe

An Example

- 5 processes, 4 resources

Max

	R_1	R_2	R_3	R_4
P_1	0	0	1	2
P_2	1	7	5	0
P_3	2	3	5	6
P_4	0	6	5	2
P_5	0	6	5	6

Alloc

	R_1	R_2	R_3	R_4
P_1	0	0	1	2
P_2	1	0	0	0
P_3	1	3	5	3
P_4	0	6	3	2
P_5	0	0	1	4

Avail

R_1	R_2	R_3	R_4
1	5	2	0

MaxRequest

	R_1	R_2	R_3	R_4
P_1	0	0	0	0
P_2	0	7	5	0
P_3	1	0	0	3
P_4	0	0	2	0
P_5	0	6	4	2

- P2 want to change its allocation to 0 4 2 0
- Safe?

An Example

- 5 processes, 4 resources

Max

	R_1	R_2	R_3	R_4
P_1	0	0	1	2
P_2	1	7	5	0
P_3	2	3	5	6
P_4	0	6	5	2
P_5	0	6	5	6

Alloc

	R_1	R_2	R_3	R_4
P_1	0	0	1	2
P_2	0	4	2	0
P_3	1	3	5	3
P_4	0	6	3	2
P_5	0	0	1	4

Avail

R_1	R_2	R_3	R_4
2	1	0	0

MaxRequest

	R_1	R_2	R_3	R_4
P_1	0	0	0	0
P_2	1	3	3	0
P_3	1	0	0	3
P_4	0	0	2	0
P_5	0	6	4	2

- P2 want to change its allocation to 0 4 2 0
- Safe?

Detecting Deadlock

- 5 processes, 3 resources

Alloc

	R_1	R_2	R_3
P_1	0	1	0
P_2	2	0	0
P_3	3	0	3
P_4	2	1	1
P_5	0	0	2

Avail

R_1	R_2	R_3
0	0	0

Pending

	R_1	R_2	R_3
P_1	0	0	0
P_2	2	0	2
P_3	0	0	0
P_4	1	0	2
P_5	0	0	2

- Given the set of pending requests, is there a safe sequence?
  - If no, deadlock

Detecting Deadlock

- 5 processes, 3 resources

Alloc

	R_1	R_2	R_3
P_1	0	1	0
P_2	2	0	0
P_3	3	0	3
P_4	2	1	1
P_5	0	0	2

Avail

R_1	R_2	R_3
0	0	0

Pending

	R_1	R_2	R_3
P_1	0	0	0
P_2	2	0	2
P_3	0	0	1
P_4	1	0	2
P_5	0	0	2

- Given the set of pending requests, is there a safe sequence?
  - If no, deadlock
- Deadlock triggered when request is formulated, not granted