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

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

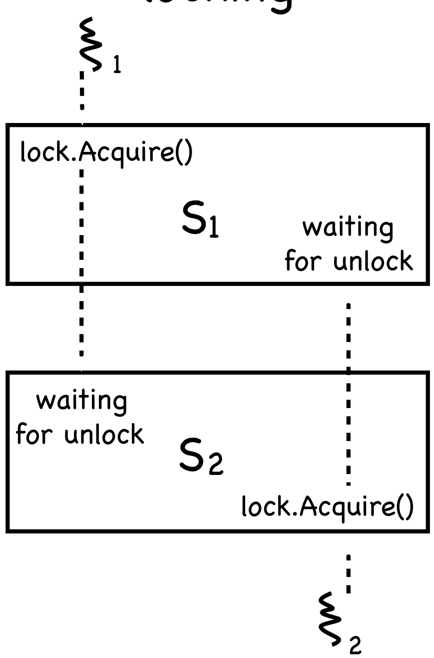
```
Producer1() {
emptyBuffer.acquire()
producerMutexLock.acquire()
  :
}
```

```
Producer2() {
producerMutexLock.acquire()
emptyBuffer.acquire()
  :
}
```

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



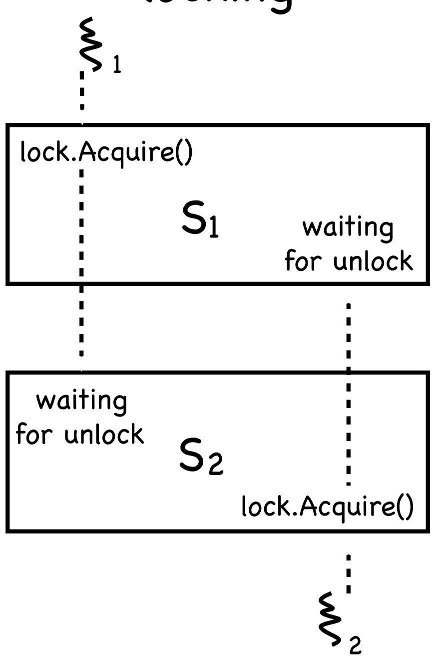

```
lock1.acquire()
...
lock2.acquire()
while (must wait)  {
        cv.wait(&lock2)
}
...
lock2.release()
...
lock1.release()
```

```
lock1.acquire()
...
lock2.acquire()
...
cv.signal()
lock2.release()
 ...
lock1.release()
```

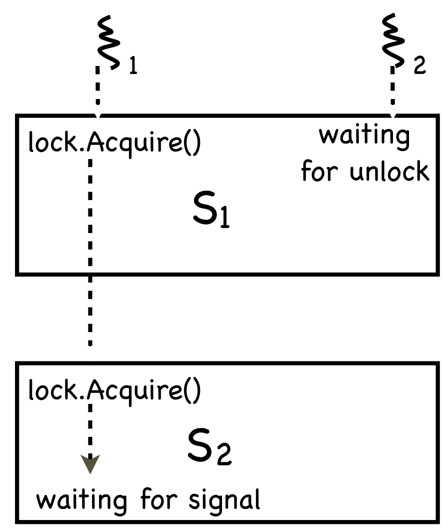
Deadlock

- 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



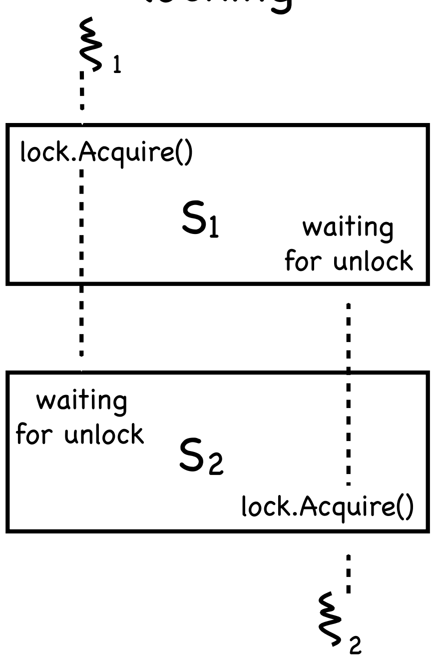
Nested waiting



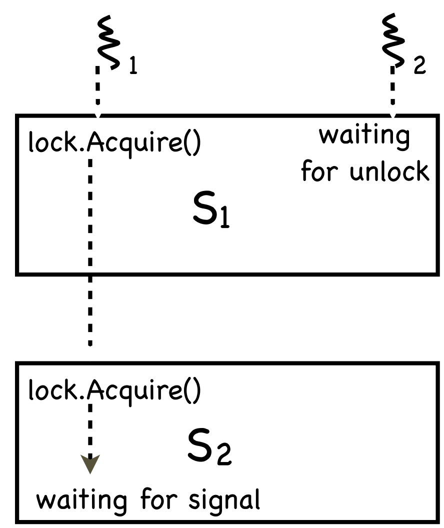
Deadlock

- 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

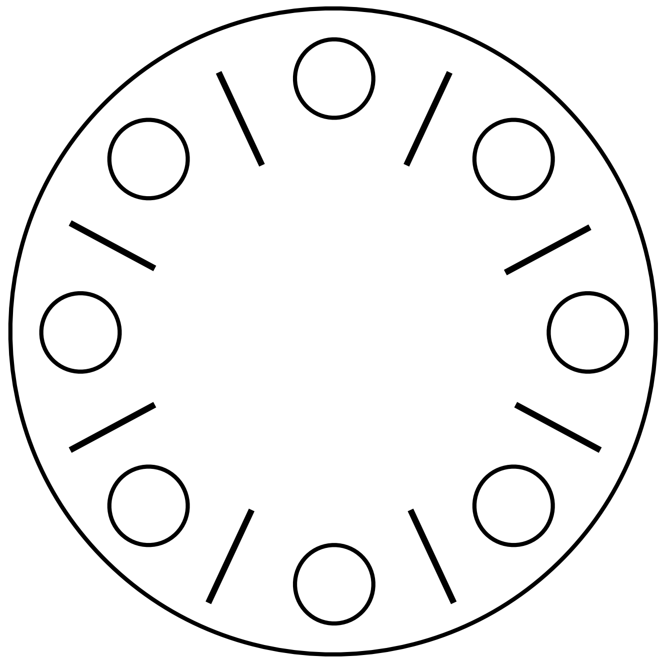
Mutually recursive locking



Nested waiting



Dining Philosophers

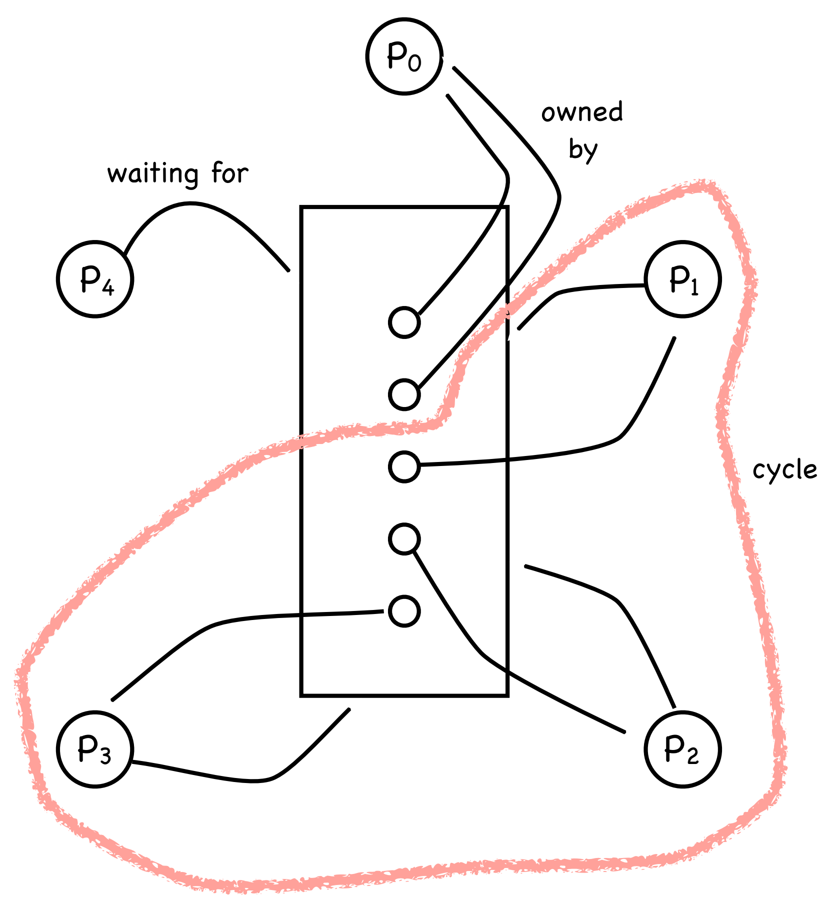


- N philosophers; N plates; N chopsticks
- If all philosophers grab right chopstick
  - deadlock!

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 T_{i+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”

Lock ordering

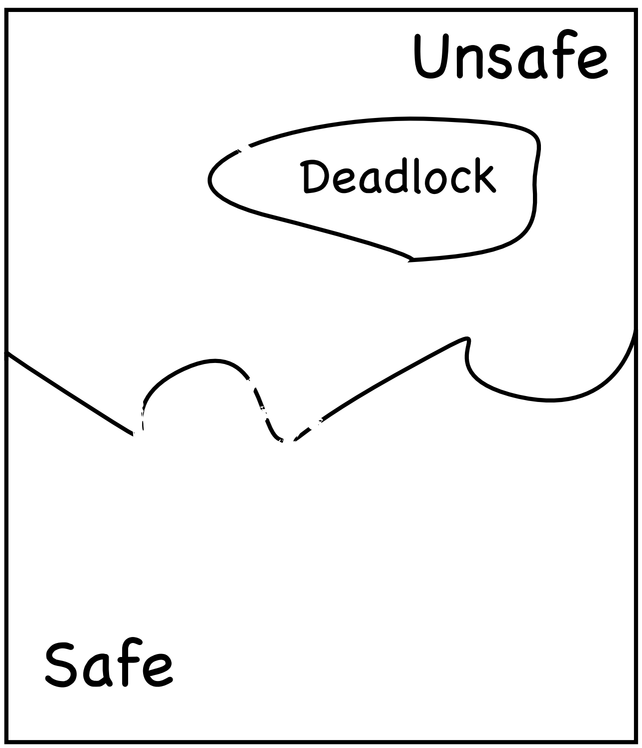
- A program code convention
  - Developers get together, have lunch, plan lock order
  - Usually reflects static assumptions about the structure of data
    - lock items in a list in order —what if order changes?
  - Nothing at compile time or run time prevents violating this convention!
    - Active research on making it better
      - Finding locking bugs
      - Automatically locking things properly
      - Transactional memory

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

  $MaxClaim_i = \quad Max_{ij}$

- $Alloc_{ij}$ = current allocation of R_j held by P_i

  $HasNow_i = \quad Alloc_{ij}$

- $Avail_j$ = 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,

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

An Example

- 5 processes, 4 resources

Max

0	0	1	2
1	7	5	0
2	3	5	6
0	6	5	2
0	6	5	6

Alloc

0	0	1	2
1	0	0	0
1	3	5	3
0	6	3	2
0	0	1	4

Avail

- Is this a safe state?

An Example

5 processes, 4 resources

Max

0	0	1	2
1	7	5	0
2	3	5	6
0	6	5	2
0	6	5	6

−

Alloc

0	0	1	2
1	0	0	0
1	3	5	3
0	6	3	2
0	0	1	4

Avail

MaxRequest

0	0	0	0
0	7	5	0
1	0	0	3
0	0	2	0
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 $MaxRequest_i \leq Avail$?

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

0	0	0	0
0	7	5	0
1	0	0	3
0	0	2	0
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

0	0	0	0
1	3	3	0
1	0	0	3
0	0	2	0
0	6	4	2

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

Detecting Deadlock

- 5 processes, 3 resources. We no longer know Max.

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

0	0	0
2	0	2
0	0	0
1	0	2
0	0	2

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

Detecting Deadlock

- 5 processes, 3 resources. We no longer know Max.

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

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