Synchronization Overview

- Topics
  - Assumptions
  - Common synchronization challenges
- Learning Objectives:
  - Discuss why synchronization is necessary.
  - Outline the common problems one encounters in designing and implementing properly synchronized code.

Assumptions

- You know why we need synchronization.
- You are familiar with the following concepts:
  - Mutual exclusion
  - Critical section
- You may not be familiar with:
  - Race condition
  - Deadlock
  - Starvation
- You have been exposed to and know how to use:
  - Locks
- Some of you have been exposed to and used:
  - Spinlocks
  - Semaphores
  - Condition variables
- You’ve never heard of this
  - Monitors

Review: Why Synchronize

- You have some shared state.
- You need to be able to read/modify it and take action based on that resource, knowing that someone else isn't doing the same thing.
- Examples:
    - Two people who share a bank account using the ATM at the same time.
    - Two processes trying to create files with the same name in the same directory at the same time.
    - A device and a user process trying to access data in the same memory locations at the same time.

Review: Mutual Exclusion

- Preventing concurrent access to something
  - A piece of code
  - A variable
- Synchronization often provides mutual exclusion between threads.

Review: Critical Section

- The piece of code to which we need to provide mutual exclusion.
- Typically the code that manipulates or examines shared state.
- Goal is to keep critical sections as short as possible.
- Clearly identifying critical sections is a good first step!

Race Condition

- When correctness depends on precisely how threads of control are interleaved (i.e., you get the synchronization wrong).
- Produces unpredictable results.
- VERY difficult to debug
    - Typically you do not know there is a race condition until long after the fact.
    - Non-deterministic, so you cannot easily reproduce it
- You should design carefully to avoid debugging race conditions; they can turn an hour of work into a lifetime of work.

Avoiding Race Conditions

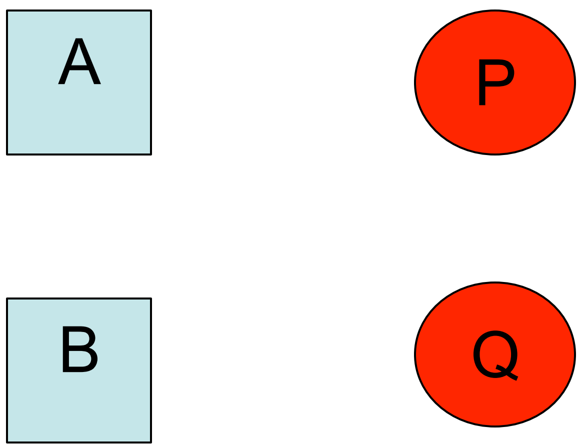
- Here are some coding techniques to help you avoid race conditions:
  - Make sure you always use the same synchronization primitive to access the same state.
  - Whenever possible encapsulate synchronization with manipulation (design synchronized APIs). Violate them at your own peril.
  - Document what primitives protect what resources.
  - Document assumptions about synchronization.
  - Review each other's designs and code.

Deadlock

- The inverse of a race condition.
- When two or more threads block each other so that no thread can make forward progress.
- Requirements:
    1. Resource is not preemptible (i.e., you can't make someone give it up).
    2. Resource requires mutual exclusion.
    3. Threads holding resources can block waiting for other resources.
    4. There exists a cycle in the graph with a directed edge between each a thread and the thread for which it is waiting.

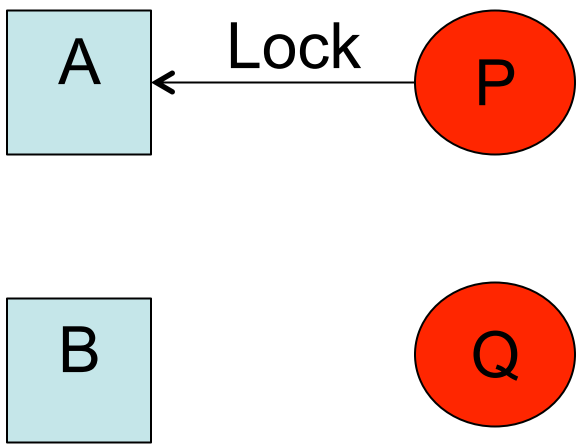
Visualizing Deadlock (1)

- Let's assume we have two threads and two objects.



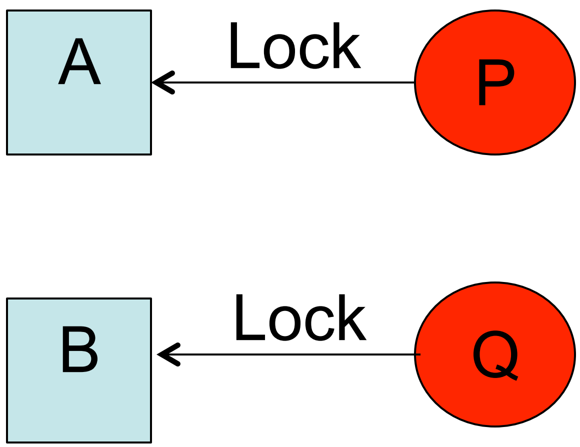
Visualizing Deadlock (2)

- Let's assume we have two threads and two objects.



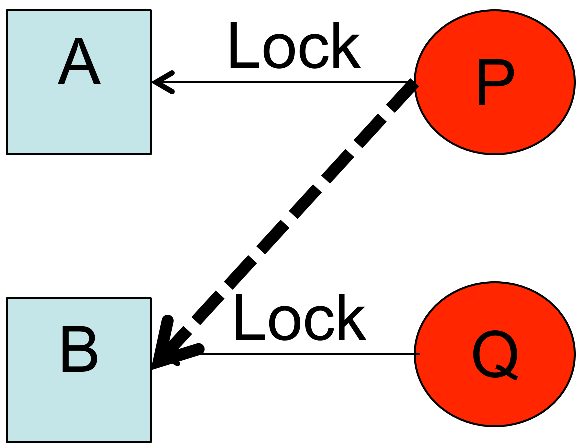
Visualizing Deadlock (3)

- Let's assume we have two threads and two objects.



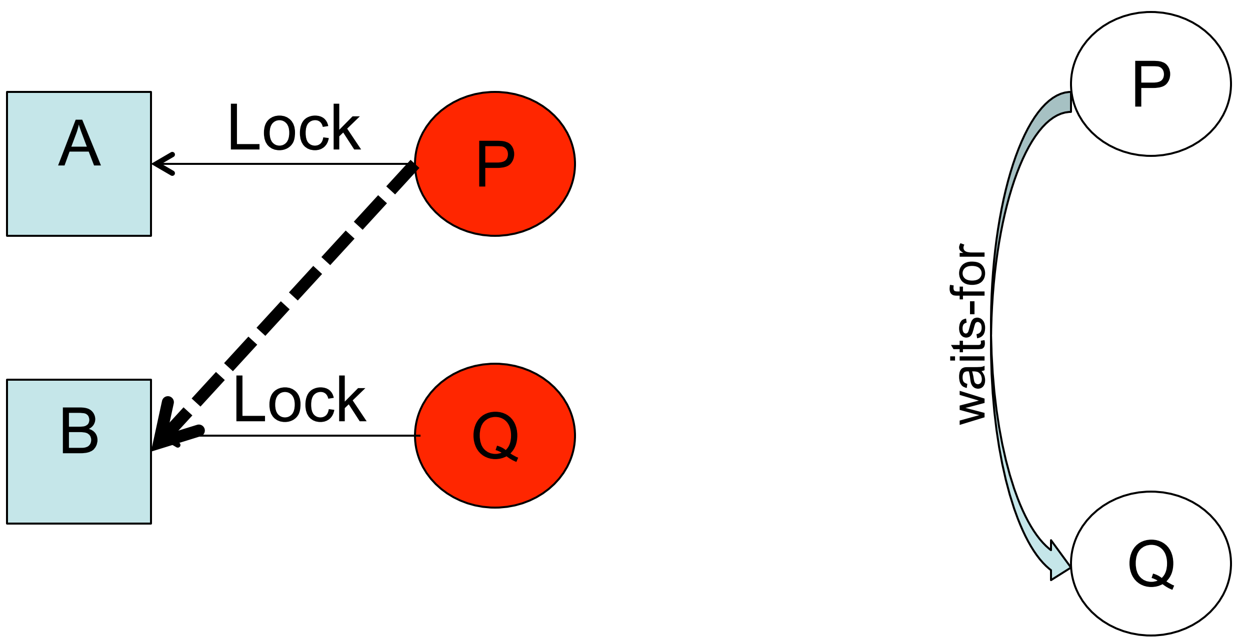
Visualizing Deadlock (4)

- Let's assume we have two threads and two objects.



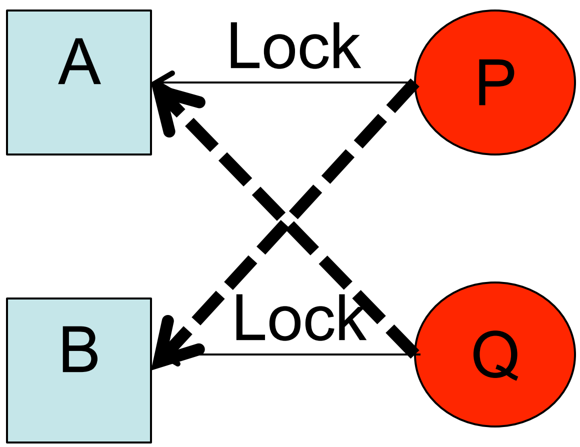
Visualizing Deadlock (5)

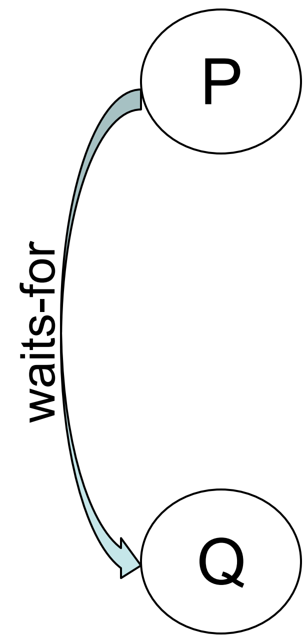
- Let's assume we have two threads and two objects.



Visualizing Deadlock (6)

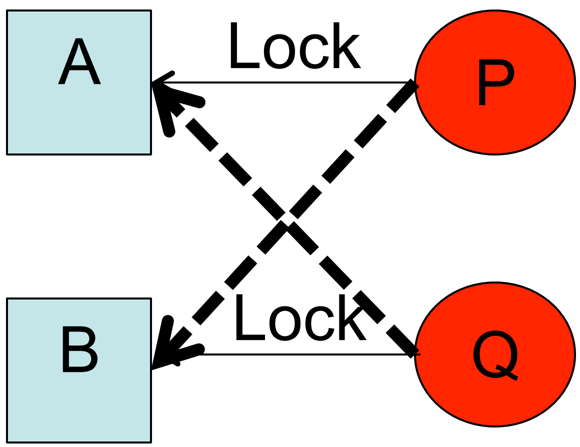
- Let's assume we have two threads and two objects.

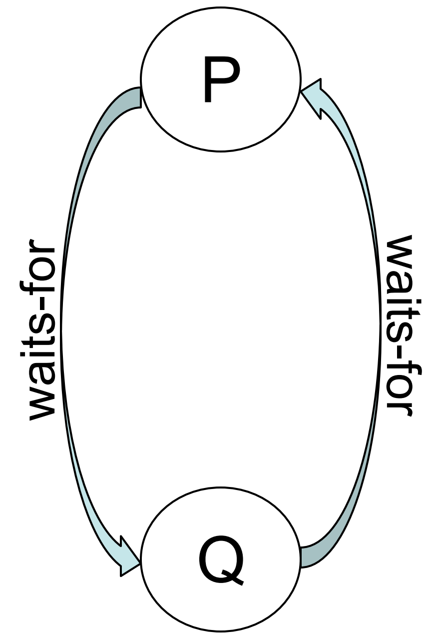




Visualizing Deadlock (7)

- Let's assume we have two threads and two objects.





Avoiding Deadlock

- Never acquire more than one resource at a time (somewhat inflexible).
- Always acquire resources in the same order (not always feasible, e.g., you don't know all the resources you need).
- Before waiting, check for deadlock and fail the operation if it would lead to a deadlock (might cause you to lose a lot of work).

Starvation

- When one (or more threads) is waiting for a resource but never gets it.
- How can this happen?
  - Scheduling is non-deterministic.
  - Scheduling gives preference to some threads in a way that could lead to starvation of others.
- Also difficult to debug
  - Sometimes handy to always have a simple backup FIFO scheduling discipline so you can determine if failure to run is a starvation problem or something else.