Synchronization Overview

- Learning Objectives:
  - Define:
    - Mutual exclusion
    - Critical section
    - Race condition
    - Deadlock
    - Starvation
  - Identify a synchronization problem
  - Explain how synchronization problems arise and what bad things can go wrong.

What problem are we solving?

- You have some shared state.
- You need to be able to read/modify it and take action based on that state, knowing that someone else isn't doing the same thing.
- Examples from real life:
  - Two people who share a bank account must be able to use an ATM at the same time.
  - Two students wish to ask a single teaching fellow a question.
  - You want to do laundry, but the machine is occupied – you'd like to be notified when it's available.

Why is this hard?

You

```
B = get_balance();



// Withdraw $100
B = B - 100;

set_balance(B);
```

Your Banking Buddy

```

B = get_balance();
// Withdraw $100
B = B - 100;


set_balance(B);
```

Conceptual Building Blocks

- Mutual exclusion
  - Preventing concurrent access to *something*
    - A piece of code
    - A variable
  - Synchronization often provides mutual exclusion between threads (or processes).
- Critical sections
  - 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!

Bad Stuff Happens (1)

- Race condition:
  - When correctness depends on precisely how threads of control are interleaved (i.e., you get the synchronization wrong).
  - Produces unpredictable results (see bank example).
  - VERY difficult to debug
    - Typically you do not know there is a race condition until long after it has occurred.
    - 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.

Bad stuff happens (2)

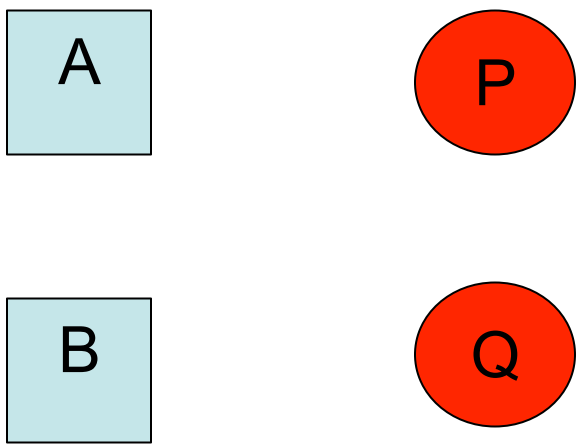
- Starvation
  - When a process blocks waiting for a resource but never gets it.
  - How can this happen?
    - Scheduling is non-deterministic.
    - Scheduling gives preference to some processes in a way that could lead to starvation of others.
  - Difficult to debug
    - Sometimes handy to always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

Bad stuff happens (3)

- Deadlock
  - The inverse of a race condition.
  - When two or more processes block each other so that no thread can make forward progress.
  - You can only have deadlock if the following conditions hold (conversely, if you can avoid at least one of these conditions, you can avoid deadlock):
    1. Resource is not preemptible (i.e., you can't make someone give it up temporarily while someone else uses it).
    2. Resource requires mutual exclusion.
    3. Someone holding a resource can block waiting for other resources.
    4. There exists a cycle in the graph with a directed edge between each a process and the process for which it is waiting. (This is called a "waits-for" graph – more details coming.)

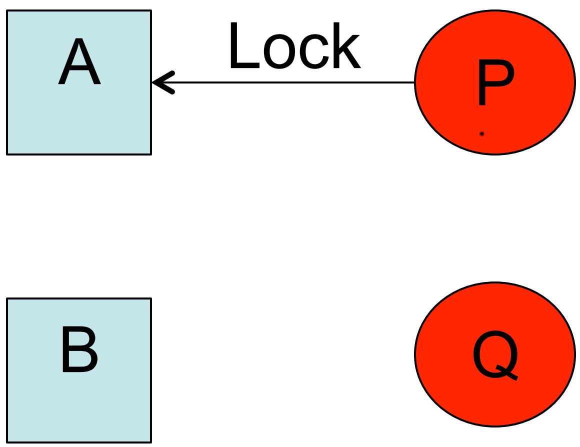
Visualizing Deadlock (1)

- Assume we have two processes and two objects.



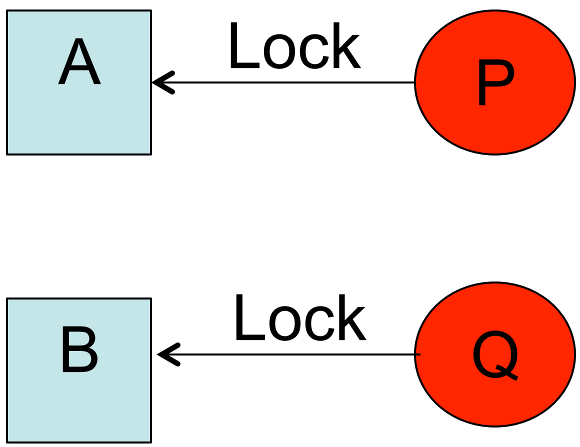
Visualizing Deadlock (2)

- Assume we have two processes and two objects.



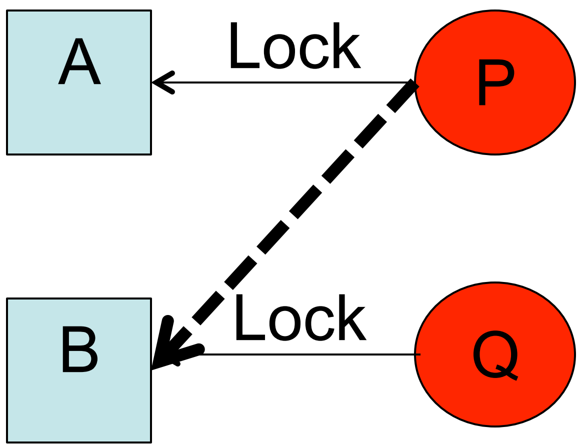
Visualizing Deadlock (3)

- Assume we have two processes and two objects.



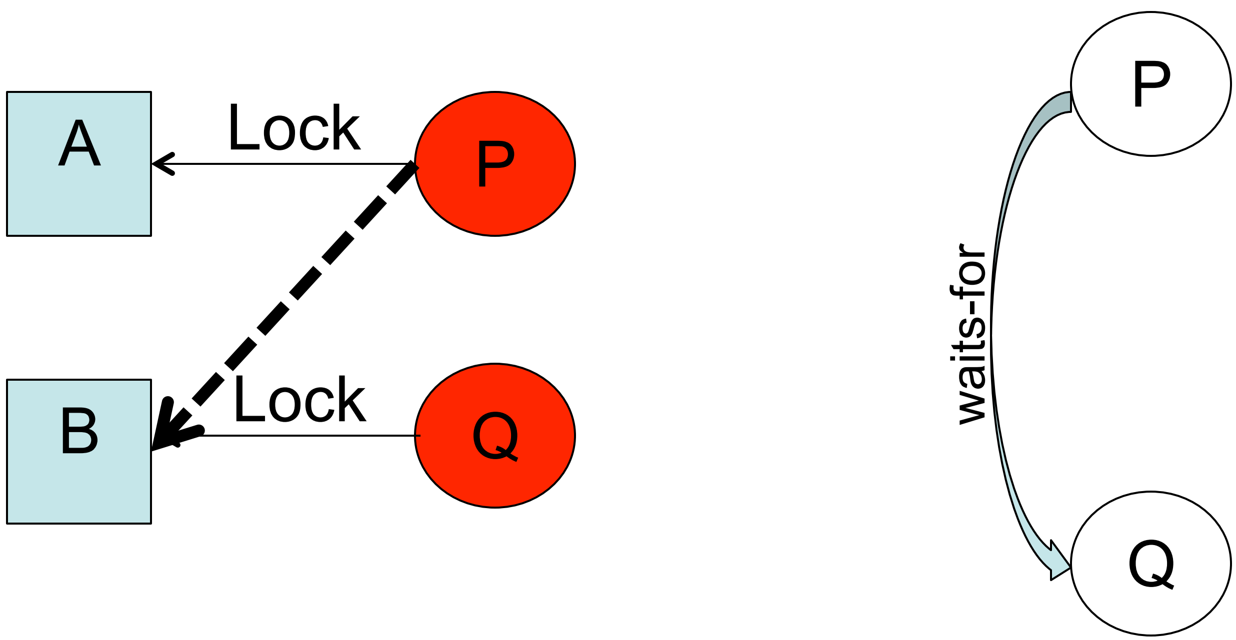
Visualizing Deadlock (4)

- Assume we have two processes and two objects.



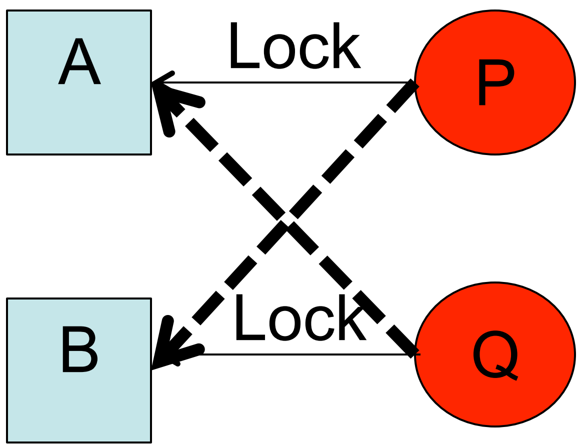
Visualizing Deadlock (5)

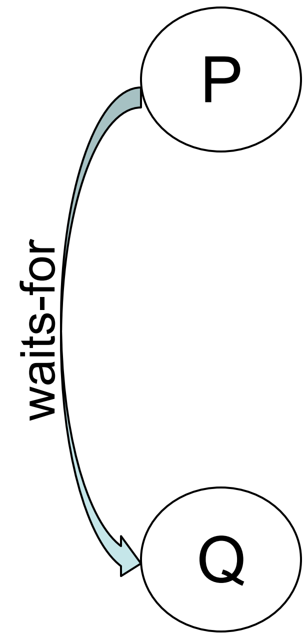
- Assume we have two processes and two objects.



Visualizing Deadlock (6)

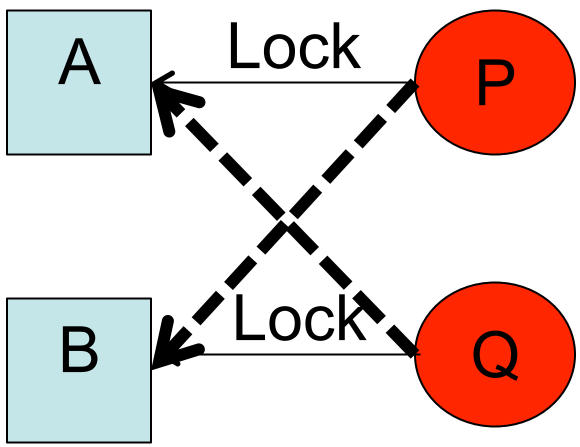
- Assume we have two processes and two objects.

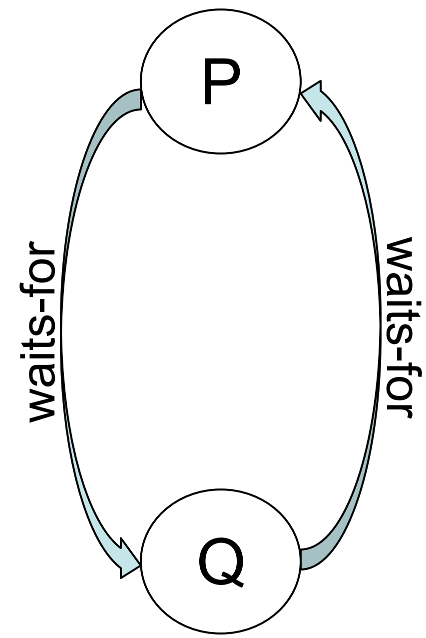




Visualizing Deadlock (7)

- Assume we have two processes 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).