Synchronization Overview

- Learning Objectives:
  - Identify synchronization problems
  - Explain how synchronization problems arise and what bad things can go wrong.
  - Use pthreads, mutexes, and condition variables.
  - Define:
    - Mutual exclusion
    - Critical section
    - Race condition
    - Deadlock
    - Starvation

Deconstructing the Problem We Solved Tuesday

- Recall the problem we had Tuesday:
  - A parent wanted to wait for a child to exit, but it also wanted to avoid waiting forever.
- We had several unsatisfying solutions that left us vulnerable to race conditions.
- We then developed a solution using `select` that worked.
- In the exercises, we also developed solutions using `signalfd` and `pselect` that worked.

What's a Programmer to do?

- This is an instance of a general class of problems:
  - We want to check on an event
  - If the event has not happened, we want to wait for it
- We discovered that calls like `select`, `pselect`, and `signalfd` allow us to solve the problem, because they provide an atomic interface that lets us check on a condition and block without allowing something to happen between the check and block.
- The operating system implements these calls, guaranteeing the atomicity, because it controls when processes run.

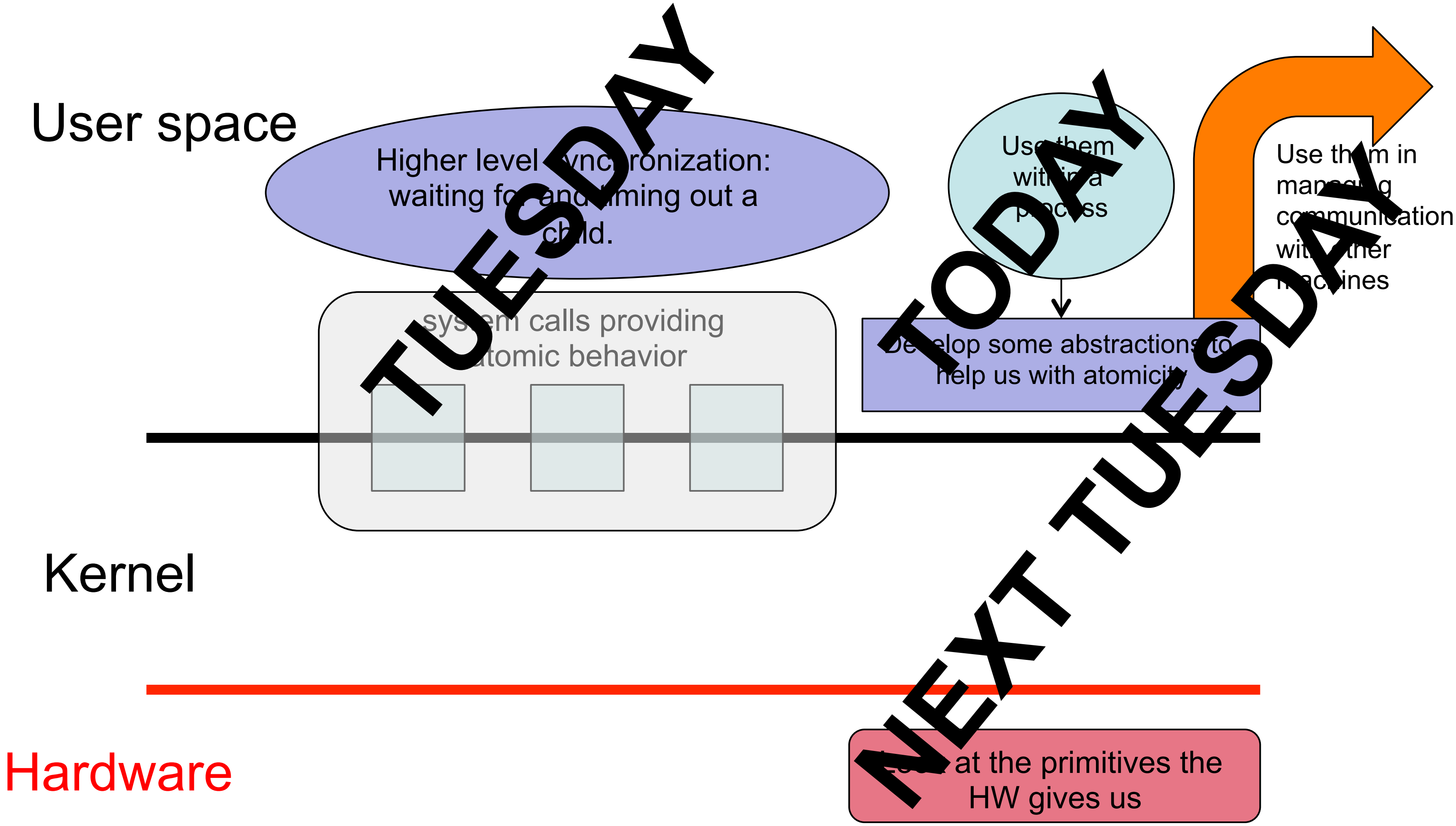
Providing Atomicity

- What if we had to ask the operating system to provide atomicity every time we needed it?

Providing Atomicity

- What if we had to ask the operating system to provide atomicity every time we needed it?
  - Could get expensive – recall that system calls are more expensive than regular function calls.
  - But wait – if you're synchronizing between two processes, doesn't the OS have to get involved when they switch anyway?
  - Maybe …
    - What if the two processes are running on different processors?
    - What if they are running on different machines?

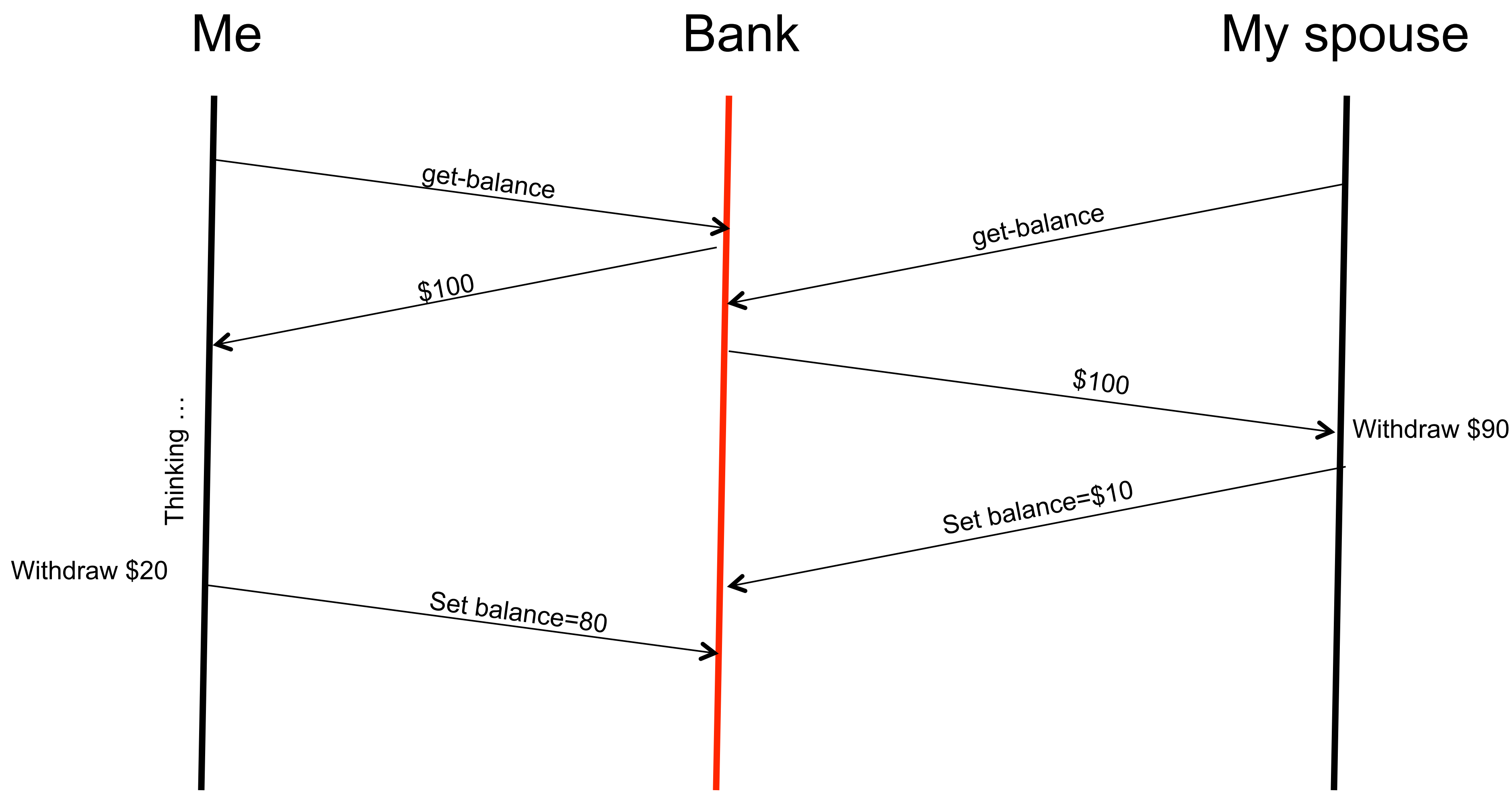
Where we are going



What problem are we solving?

- You have some shared state (e.g., a child's exit status).
- 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?



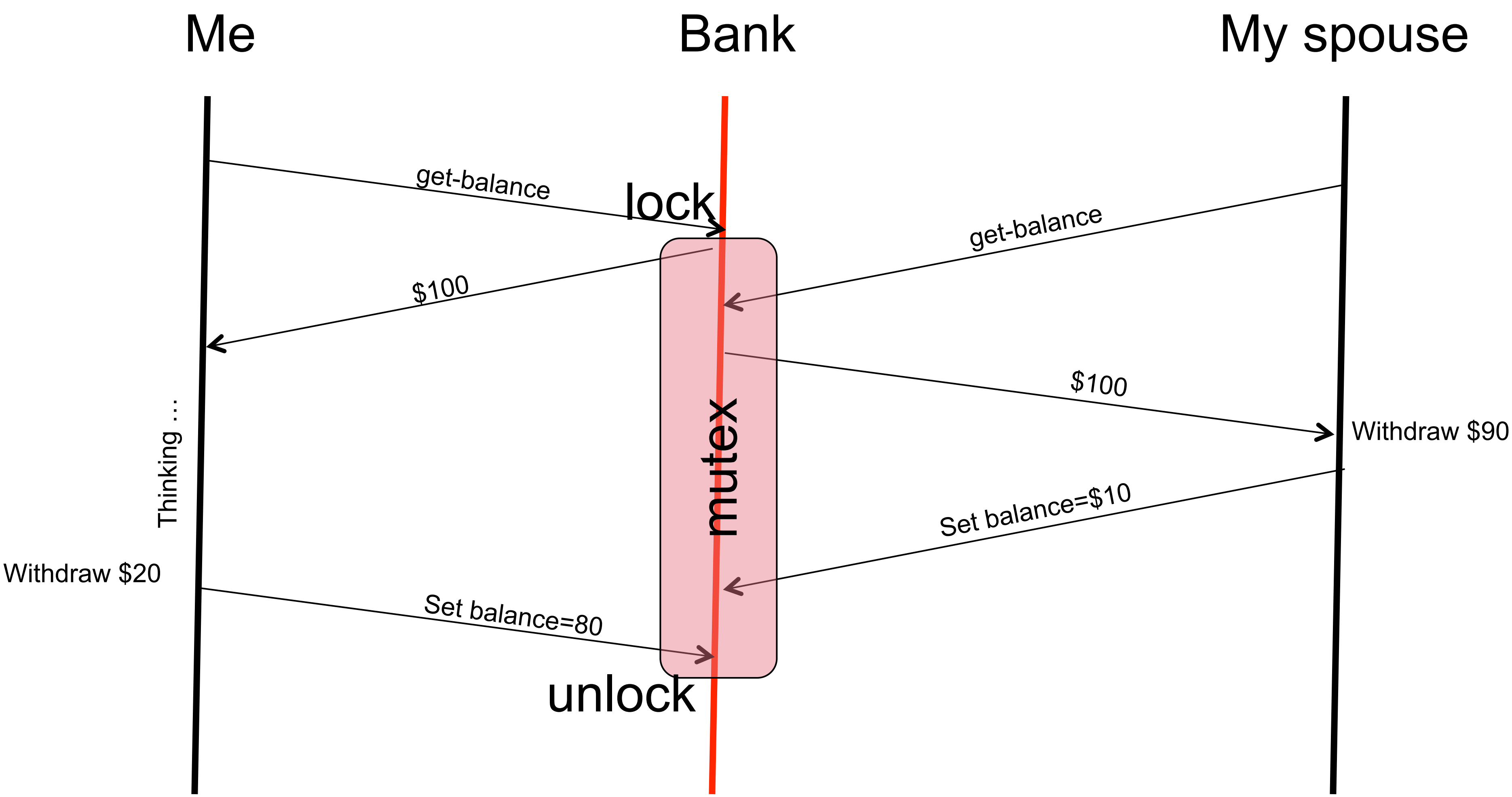
Bad Stuff Happens (1)

- Race condition:
  - When correctness depends on precisely how threads of control are interleaved.
  - Produces unpredictable results.
  - 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
  - We need to introduce some abstractions and mechanisms to implement those abstractions to deal with race conditions.

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!

Mutual exclusion/critical sections



Avoiding Race Conditions

- Here are some coding techniques to help you avoid race conditions:
  - You will use synchronization primitives to manage critical sections to achieve mutual exclusion.
  - 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)

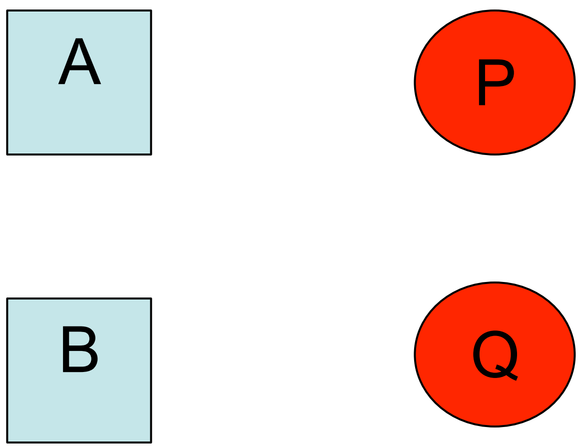
- Starvation
  - When a process blocks waiting for a resource but never gets it.
  - How can this happen?
    - Scheduling is non-deterministic.
    - Scheduling gives preference to some processes in a way that could lead to starvation of others.
  - Difficult to debug
    - Sometimes handy to always 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 agents block each other so that neither 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.)

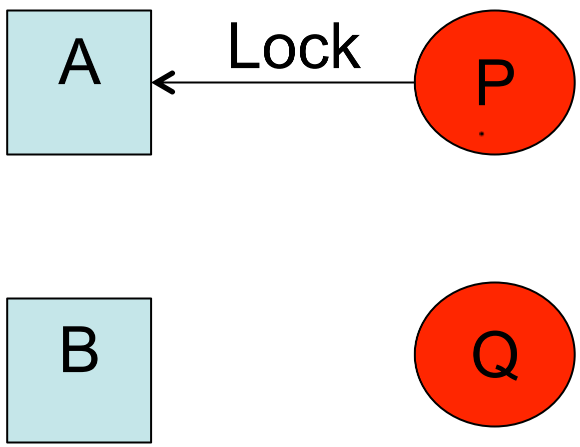
Visualizing Deadlock (1)

- Assume we have two processes and two objects.



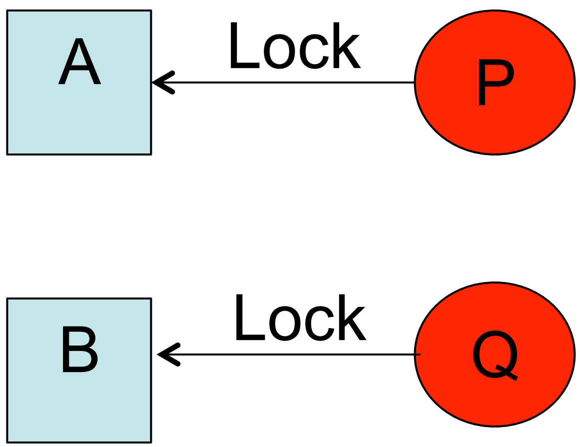
Visualizing Deadlock (2)

- Assume we have two processes and two objects.



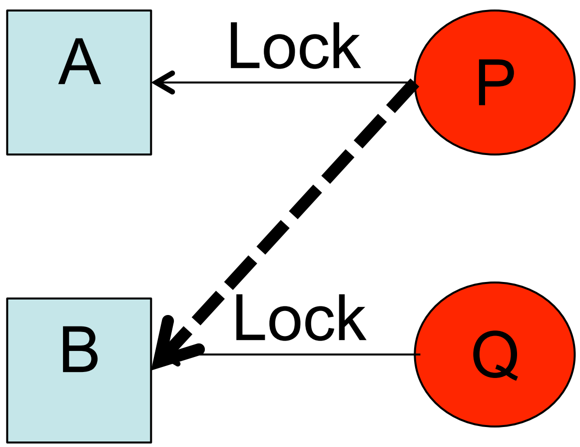
Visualizing Deadlock (3)

- Assume we have two processes and two objects.



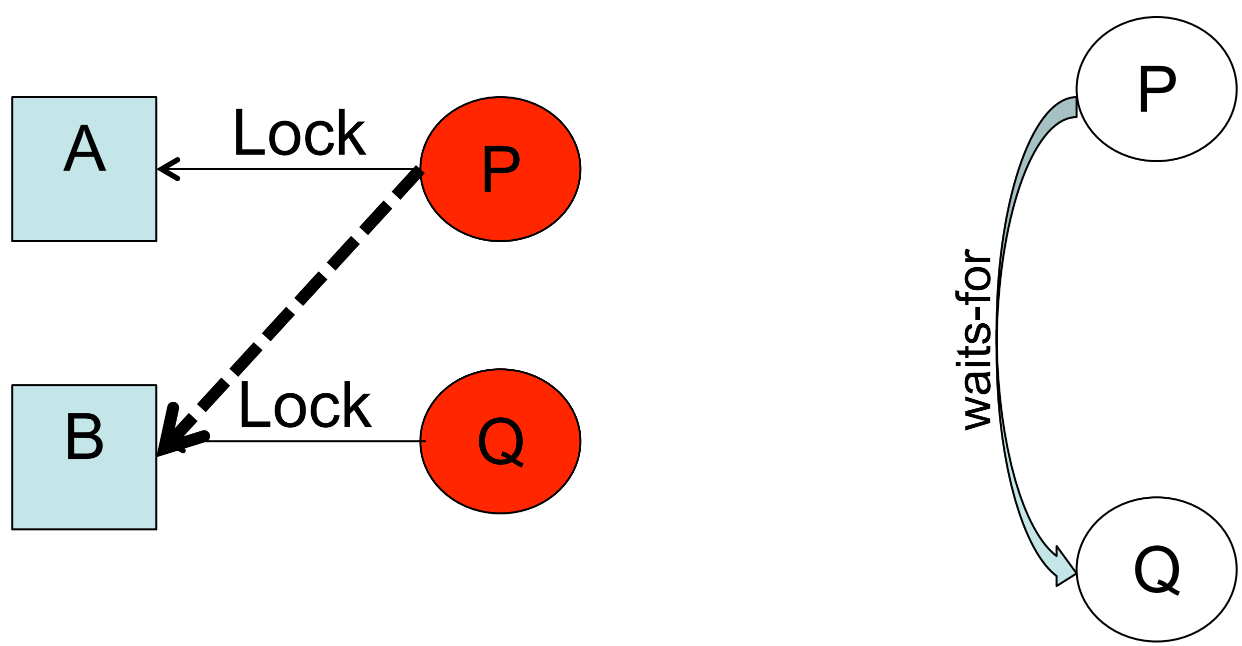
Visualizing Deadlock (4)

- Assume we have two processes and two objects.



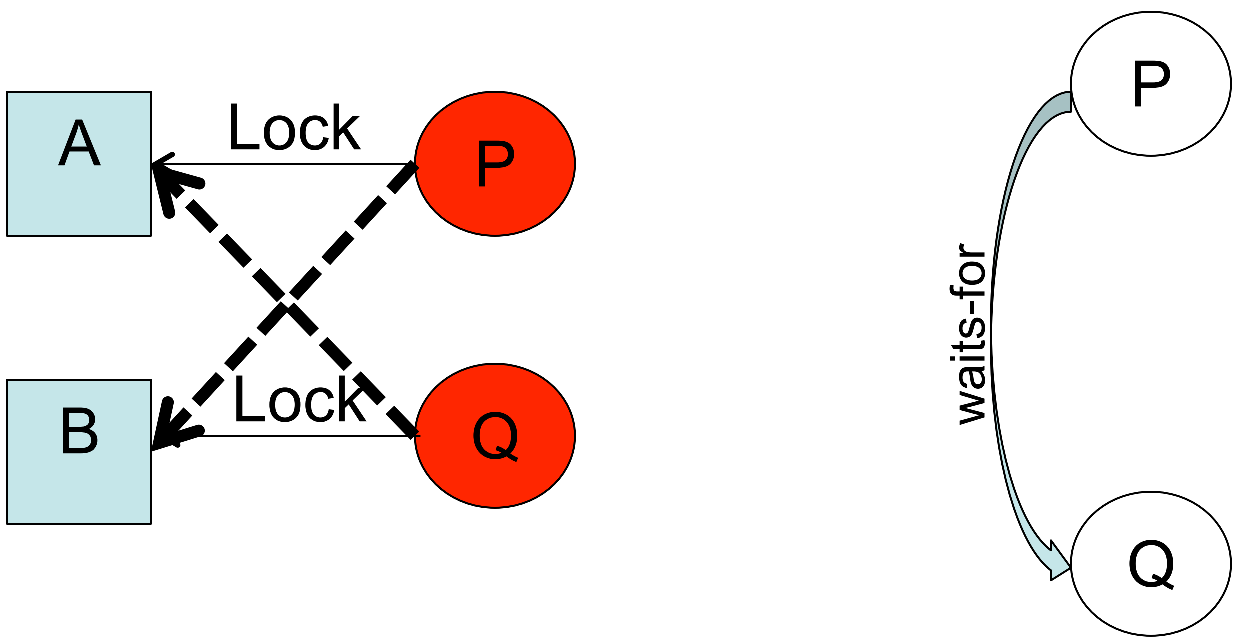
Visualizing Deadlock (5)

- Assume we have two processes and two objects.



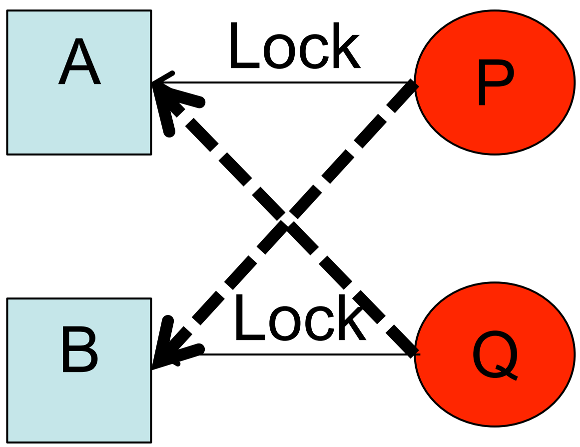
Visualizing Deadlock (6)

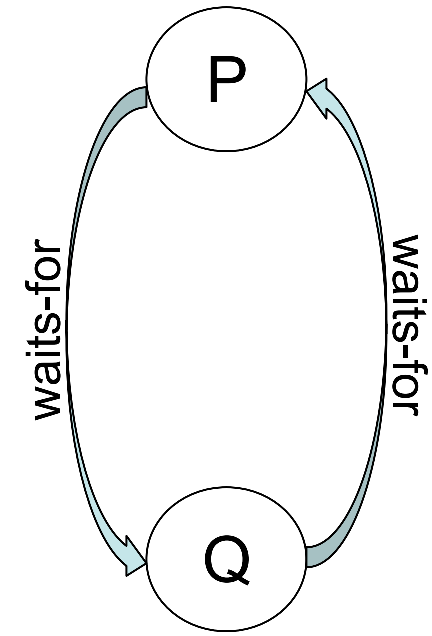
- Assume we have two processes and two objects.



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).

Process = Address Space + Thread(s) (1)

- A process is composed of two parts:
  - A part that keeps track of "stuff": Address space
  - A dynamic part: Thread
- Address space:
  - The set of addresses (e.g., memory locations) to which a running computation has access.
  - Address spaces provide protection boundaries.

Process = Address Space + Thread(s) (2)

- A process is composed of two parts:
  - A part that keeps track of "stuff": Address space
  - A dynamic part: Thread
- Thread:
  - A logical flow of control
  - Execution state
- A process has one address space and one or more threads in it.
- Threads share the address space, i.e., memory, so you need to synchronize access to memory between threads.

Pthreads

- Pthreads is a standard interface to threads.
  - Specified by POSIX
- Includes APIs for different aspects of threads:
  - Thread routines (e.g., create, exit, join)
  - Attribute object routines (get and set thread attributes)
  - Mutex routines
  - Condition variable routines
  - Read/write lock routines
  - Per-thread context routines – manage per-thread data
  - Cleanup routines

Thread Routines

```
int pthread_create(pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start_routine)(void *),
    void *arg);
void pthread_exit(void *value_ptr);
pthread_t pthread_self(void);
int pthread_join(pthread_t thread,
    void **value_ptr);
```

Mutex Routines

```
int pthread_mutex_init(pthread_mutex_t *mutex,
    const pthread_mutexattr_t *attr);

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);

int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Screen Capture

- Let's look at pingpong.c (in synch1)
  - We have four threads (2 pings and 2 pongs); they are trying to alternate printing ping and pong, but are unsynchronized.
  - Let's see if we can add locks (mutexes) to make this work.
  - Solution is in pingpong-mutex.c

Condition Variables (CV)

- A construct designed to let you atomically check a condition and wait if the condition is not true.
- Paired with a mutex that protects the state that the condition checks.
- Interface
  - `cv_create (cv_destroy)`: Create (Destroy) a condition variable
  - `cv_wait`: block until the condition becomes true
  - `cv_broadcast`: wake eveyone waiting on this condition variable
  - `cv_signal`: wake one entity waiting on this condition variable
- Use case:
  - Want to run when a condition is true
  - Condition is typically simple
  - Need to check condition and wait atomically

CV Usage Pattern

- Usage:
  1. Acquire mutex
  2. Check condition
  3. If you need to wait on condition, call `cv_wait`.
  4. Once condition is true, decide if you want to `cv_signal` or `cv_broadcast` information to others.
  5. Release mutex
- Semantics:
  - Hoare semantics: If you wait on a condition, when you wake up you are **guaranteed that the condition is true**.
  - Mesa semantics: **No guarantees** when you wake; someone else may have beaten you to the punch.
  - **pthreads uses Mesa semantics; you must code accordingly.**
    - **Typically, this means that condition checks appear in a while loop.**

CV Example

- How might we do the, “Check if there is work on a work queue, and if so, let the server processes know.”

```
work_cv = create_cv();
work_mutex = create_mutex();
lock(work_mutex);
while (work queue is empty)
     cv_wait(work_cv,work_mutex);
// Now we can signal workers
cv_broadcast(work_cv);
unlock(work_mutex);
```

Condition Variable Routines

```
int pthread_cond_init(pthread_cond_t *cond,
    const pthread_condattr_t *attr);
int pthread_cond_wait(pthread_cond_t *cond,
    pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond,
    pthread_mutex_t *mutex,
    const struct timespec *abstime);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_destroy(pthread_cond_t *cond);
```

Screen Capture

- Let's now see if we can use CVs to make this a bit more efficient
  - Solution is in pingpong-cv.c