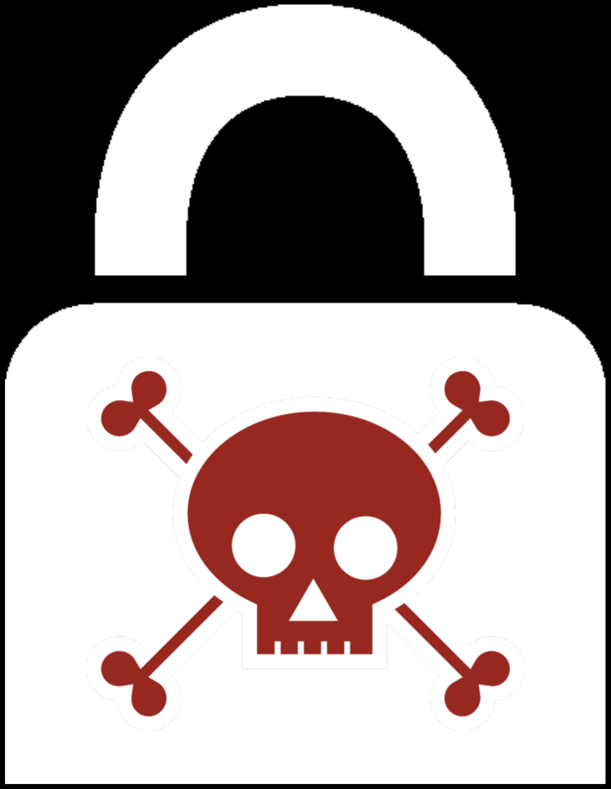
Ch. 10 Avoiding Liveness Hazards

Jesper Pedersen Notander

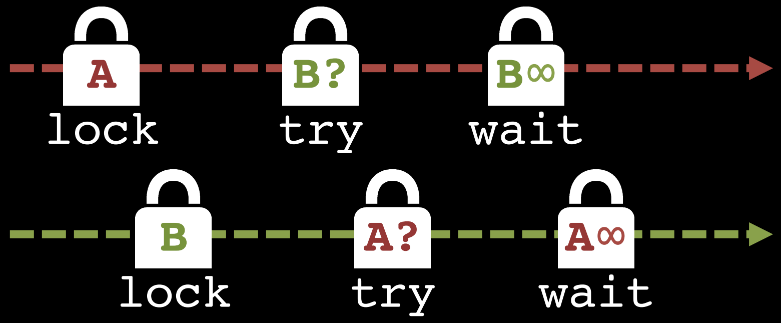
Liveness and Safety

- A liveness property,
  - *something good eventually happens.*
  - e.g. program termination.
- A safety property,
  - *something bad never happens.*
  - e.g. inconsistent shared states.
- Tension between liveness and safety.
- Protection → liveness hazards.

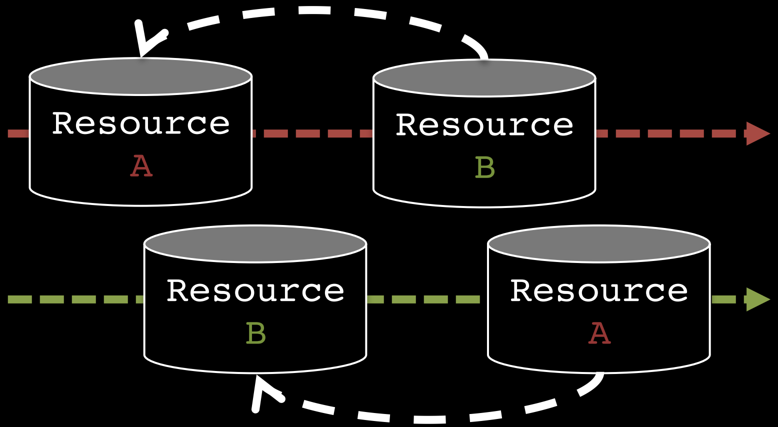


Deadlock

Lock Ordering Deadlock



Resource Deadlock



Lock Ordering Deadlock

```
doLeftRight(){
  synchronized( A ){
    synchronized( B ){
      // do something
}}}
```

```
doRightLeft() {
  synchronized( B ){
    synchronized( A ){
      // do something
}}}
```

- Beware nested synchronized blocks.
- Always same order → no deadlocks.

Dynamic Lock Ordering Deadlock

```
transaction(A, B) {
  synchronized(A){
    synchronized(B){
      // do the transaction
}}}
```

- Order unknown, defined by caller

Solution: Impose an Order

```
transaction(A, B) {
  if (#A > #B) {
    transfer(A, B);
  } else if (#A < #B) {
    transfer(B, A);
  } else {
    synchronized(T){
      transfer(A, B);
}}
```

```
transfer(X, Y) {
  synchronized(X) {
    synchronized(Y) {
      // perform a safe
      // transfer
}}}
```

Deadlock and Cooperating Objects

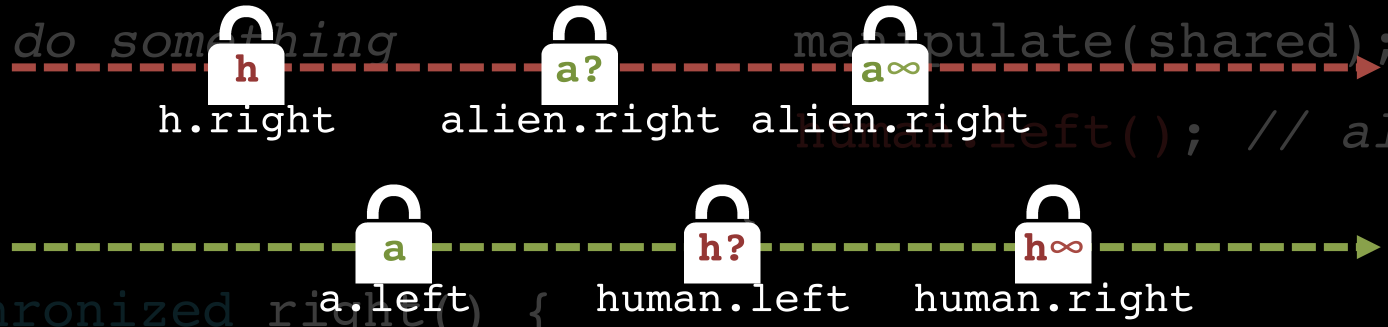
```
//class Human
synchronized left() {
  // do something
}

synchronized right() {
  manipulate(shared);
  alien.right(); // alien
}
```

```
//class Alien
synchronized left() {
  manipulate(shared);
  human.left(); // alien
}

synchronized right() {
  // do something
}
```

Deadlock and Cooperating Objects

```
//class Human
synchronized left() {
  // do something
}

synchronized right() {
  manipulate(shared);
  alien.right(); // alien
}
```

```
//class Alien
synchronized left() {
  manipulate(shared);
  human.left(); // alien

synchronized right() {
  // do something
}
```



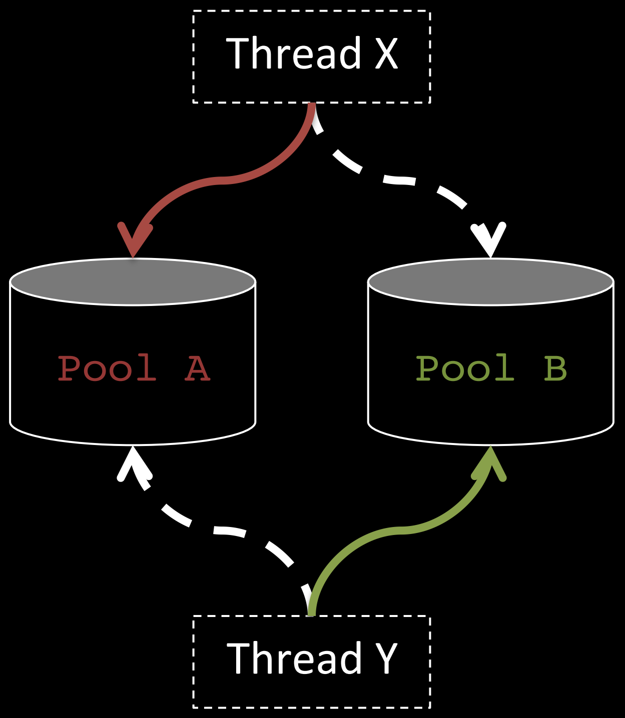
Solution: Open Calls

```
//class Human
synchronized left() {
  // do something
}

right() {
 synchronized(this) {
    manipulate(shared);
  }
  alien.right(); // open
}
```

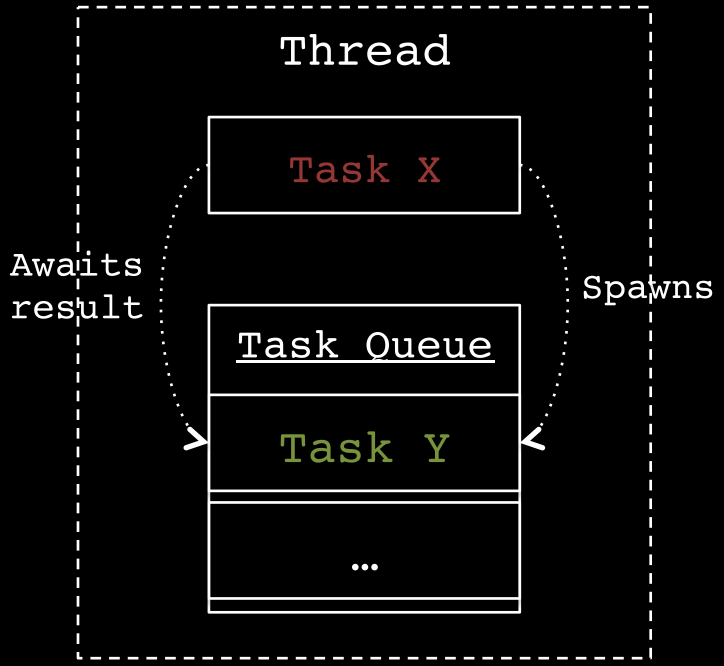
```
//class Alien
left() {
  synchronized(this) {
    manipulate(shared);
  }
  human.left(); // open
}

synchronized right() {
  // do something
}
```

Resource Deadlocks

Resource Pools



Thread-Starvation Deadlock



Avoiding and Analyzing Deadlocks

- Acquiring one lock at a time → no deadlocks.
  - Unfeasible → lock ordering must be in the design.
  - Or use explicit locks
    - `java.util.concurrent.locks`, `tryLock(long timeout)`
- Deadlocks analysis using *thread dumps*.
  - Triggered when sending `SIGQUIT` to the JVM.
  - Deadlock identification, less support with Lock.

Other Liveness Hazards

- Starvation
  - Denial of access to resources, e.g. CPU time
  - Thread priorities causes starvation
- Poor responsiveness
  - Not as severe as starvation
  - Heavy processes competing for CPU time

Livelock

- A thread that cannot progress, due to infinite retries of an action that always fail.
  - Common source of failure, error-recovery code.
- Or, multiple cooperating threads change state in a way that makes no further progress possible.
  - Solution: Introduce some randomness

Summary

- Synchronization give rise to liveness hazards.
- The most common hazard is lock ordering deadlock.
    - It must be handled already at design time.
    - Open calls is effective at minimizing this hazard.
- Other hazards mentioned are: resource deadlock, resource starvation, and livelock.

Thank

You