

# Deadlocks

P. J. Denning

© 2022, P. J. Denning

# Deadlocks

- By using locks to avoid race conditions, we create a new problem: circular waiting on locks.
- Consider ATM example
- Transaction is program that performs a specific function on the database (e.g., making a deposit or withdrawal).
- Transaction includes locking the records to prevent races while updating them.

# Account Transfer Example

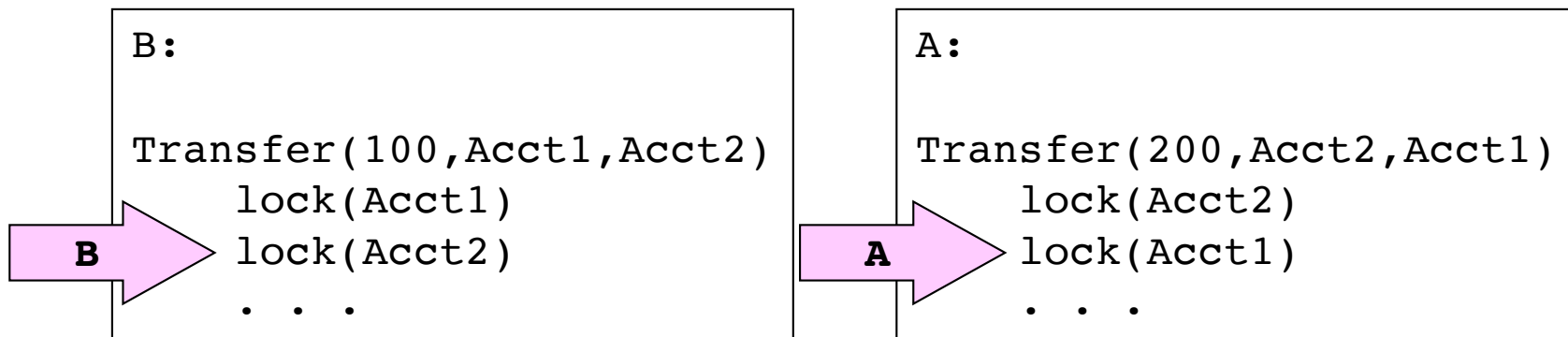
Transaction to transfer X dollars from account A to B.

```
Transfer(X, A, B)
  lock(A)
  lock(B)
  A = A - X
  B = B + X
  UNLOCK(B)
  UNLOCK(A)
  EXIT
```

What happens if Bob and Alice try this at the same time?

B: Transfer(100,Acct1,Acct2)

A: Transfer(200,Acct2,Acct1)



**DEADLOCK!**

# Conditions for Deadlock

- General conditions
  - Many concurrent tasks -- processes, threads, transactions
  - Tasks require resources to make progress, wait if unavailable
  - System has resources of different types (CPU, memory pages, disk sectors, files, records, locks, semaphores, etc.)
  - Each task holds some resources
- Necessary conditions
  - Tasks must wait for resources to be allocated
  - While task holds resource, it is not available for others
- Deadlock = circular wait among tasks.

# Two kinds of deadlock

- Signals (consumable resources)
  - Each task waiting for signal from another
  - Can't back out
- Units (reusable resources)
  - Each task waiting for another to release a resource unit
  - Can back out by aborting tasks and freeing up their resources

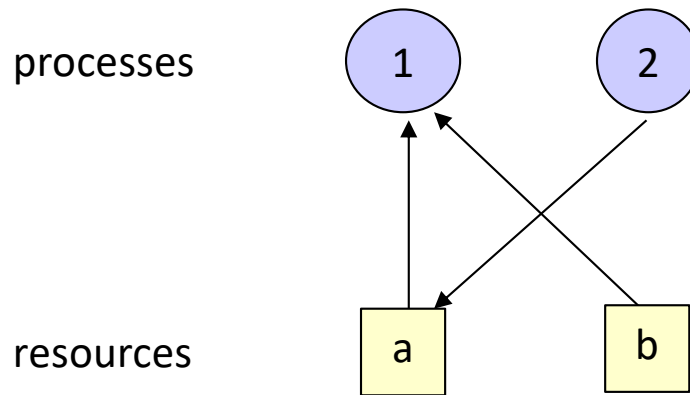
# Signal Deadlock

```
P1: ...  
    wait(a)  
    wait(b)  
    use objects a and b  
    signal(b)  
    signal(a)  
    ...
```

```
P2: ...  
    wait(b)  
    wait(a)  
    use objects a and b  
    signal(a)  
    signal(b)  
    ...
```

Initially, semaphores a and b are both 1.  
Deadlock if P1 and P2 complete their  
first waits before either attempts their  
second wait. See ATM account transfer example.

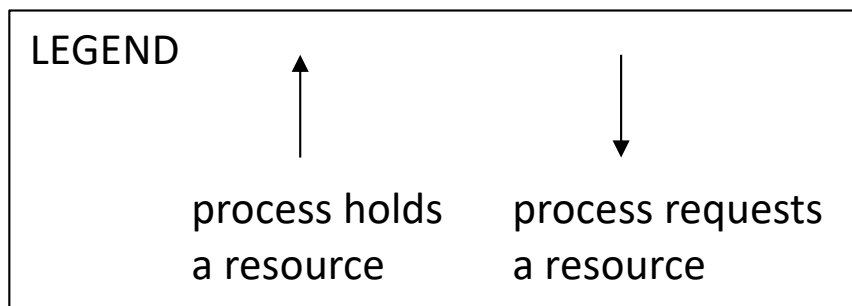
# Resource use graph



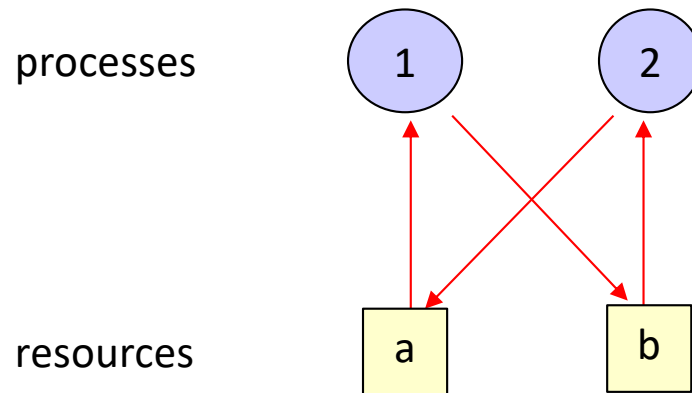
execution:

- P1 wait(a) -- pass – hold a
- P1 wait(b) -- pass – hold b
- P2 wait(a) -- wait for a

P1 releases a and b, allowing P2 to continue







execution:

P1 wait(a) -- pass – hold a

P2 wait(b) -- pass – hold b

P1 wait(b) -- wait for a

P2 wait(a) -- wait for b

Notice the circular wait:

P1-b-P2-a-P1

The cycle shows a deadlock

May not be easy to test for signal deadlocks:

```
T:  transaction(a,b)
    wait(a)
    wait(b)
    use objects a and b
    signal(b)
    signal(a)
    return
```

Deadlock may result if P1 calls T(X,Y) and P2 calls T(Y,X) at the same time. Now there are two conflicting tasks running in parallel, risking deadlock.

But no deadlock possible if both say T(X,Y) at same time.

# Two Phase Locking

Database systems use two-phase protocol:  
get all locks before updating; back out but do not wait.

Operation lock(r) returns lock value and sets lock  
(like TSL hardware instruction)

```
T:  transaction(a,b,c)
    if lock(a) then goto T
    if lock(b) then {unlock(a); goto T}
    if lock(c) then {unlock(a,b); goto T}
    update records a,b,c
    unlock(a,b,c)
    return
```

Two-phase protocol may have long busy-waiting period during times of heavy contention for shared records.

Can reduce waiting probability by inserting random delay before looping back after finding locked records.

# Resource Deadlocks

- Circular waits arising from waiting for new resources to be granted while holding other resources.
- Can be “backed out of” by aborting one or more of the deadlock processes and releasing their resources back to system pools.

# Banking Example

- Bank gives Alice credit limit \$100 and Bob \$200. Loan pool is \$250.
- Alice asks for \$90, bank grants. (Pool = \$160)
- Bob asks for \$160, bank grants. (Pool = \$0)
- Alice asks for \$10, waits.
- Bob asks for \$10, waits.
- Back out by getting either one to fully repay their loan

# Buffer Pool Example

- OS has a buffer pool used by processes when they want to send messages
- Can get a deadlock if a set of processes are all waiting for a buffer and they collectively hold all the buffers

# Kitchen Example

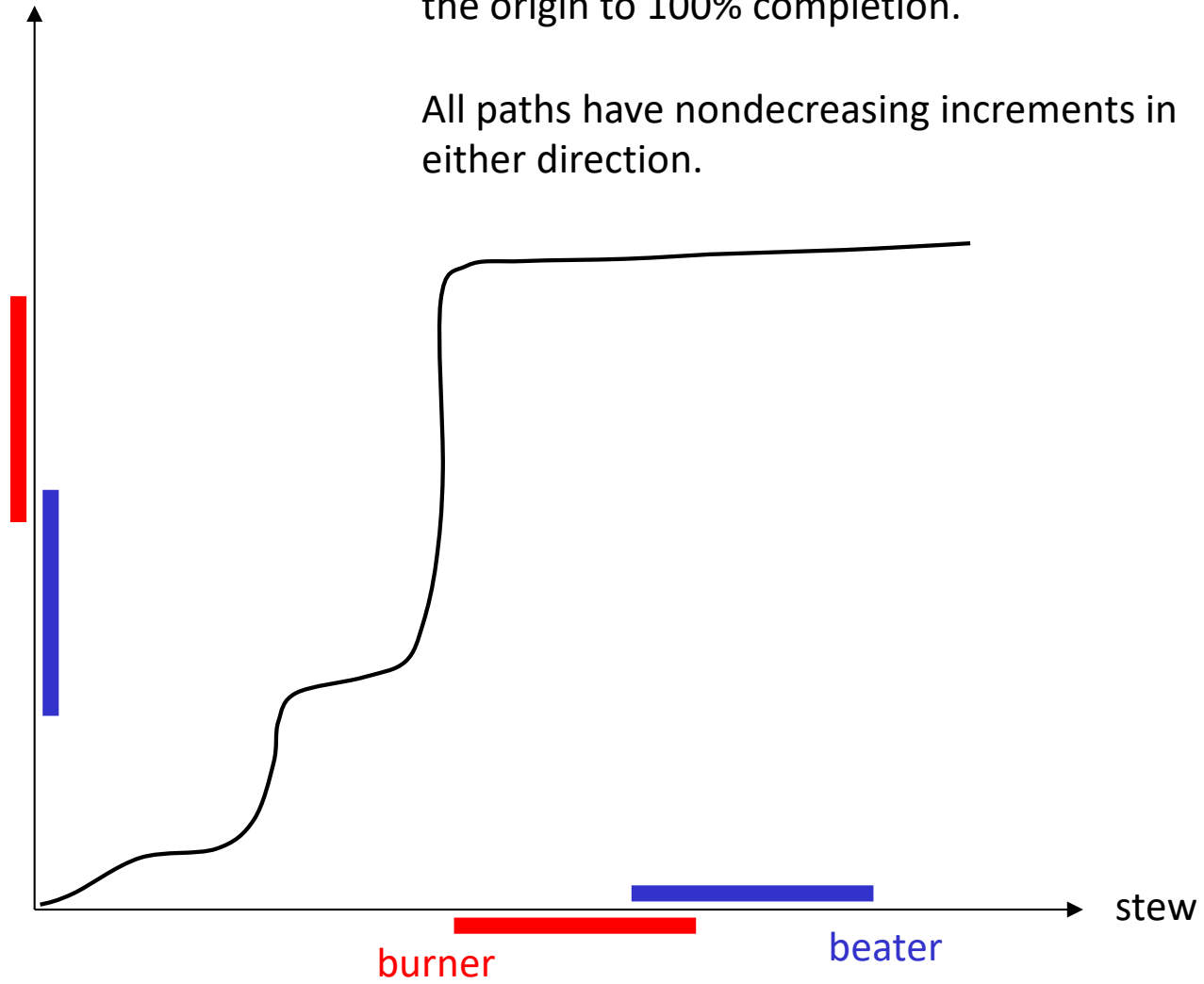
- Kitchen has two resources, burner and beater
- Chef has two tasks, make stew and pudding
- Stew recipe: start beating after placing on burner, continue beating for a few minutes after removing from burner.
- Pudding recipe: start beating before placing on burner, stop beating a few minutes after placing on burner.



pudding

There are many joint progress paths from the origin to 100% completion.

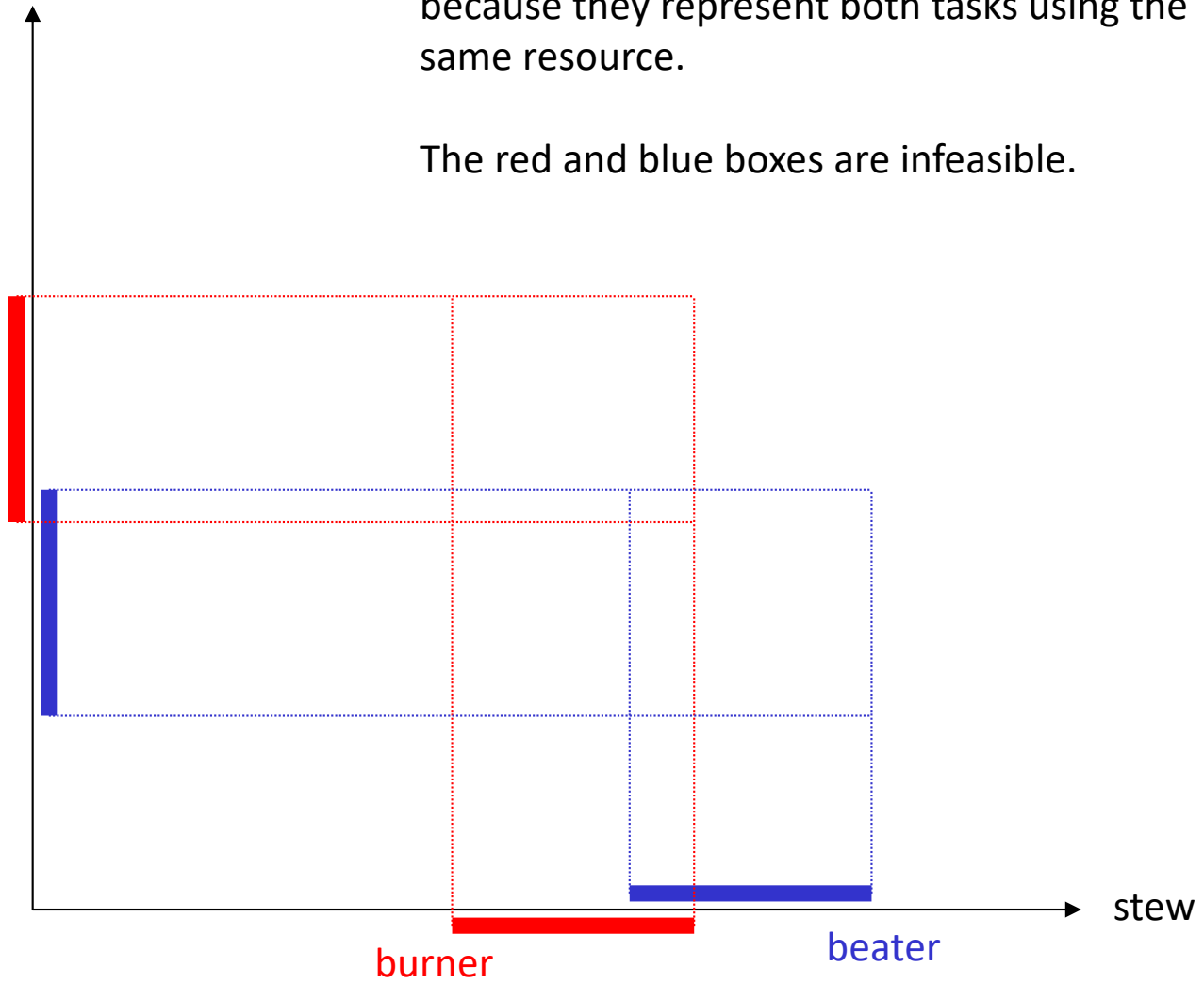
All paths have nondecreasing increments in either direction.



pudding

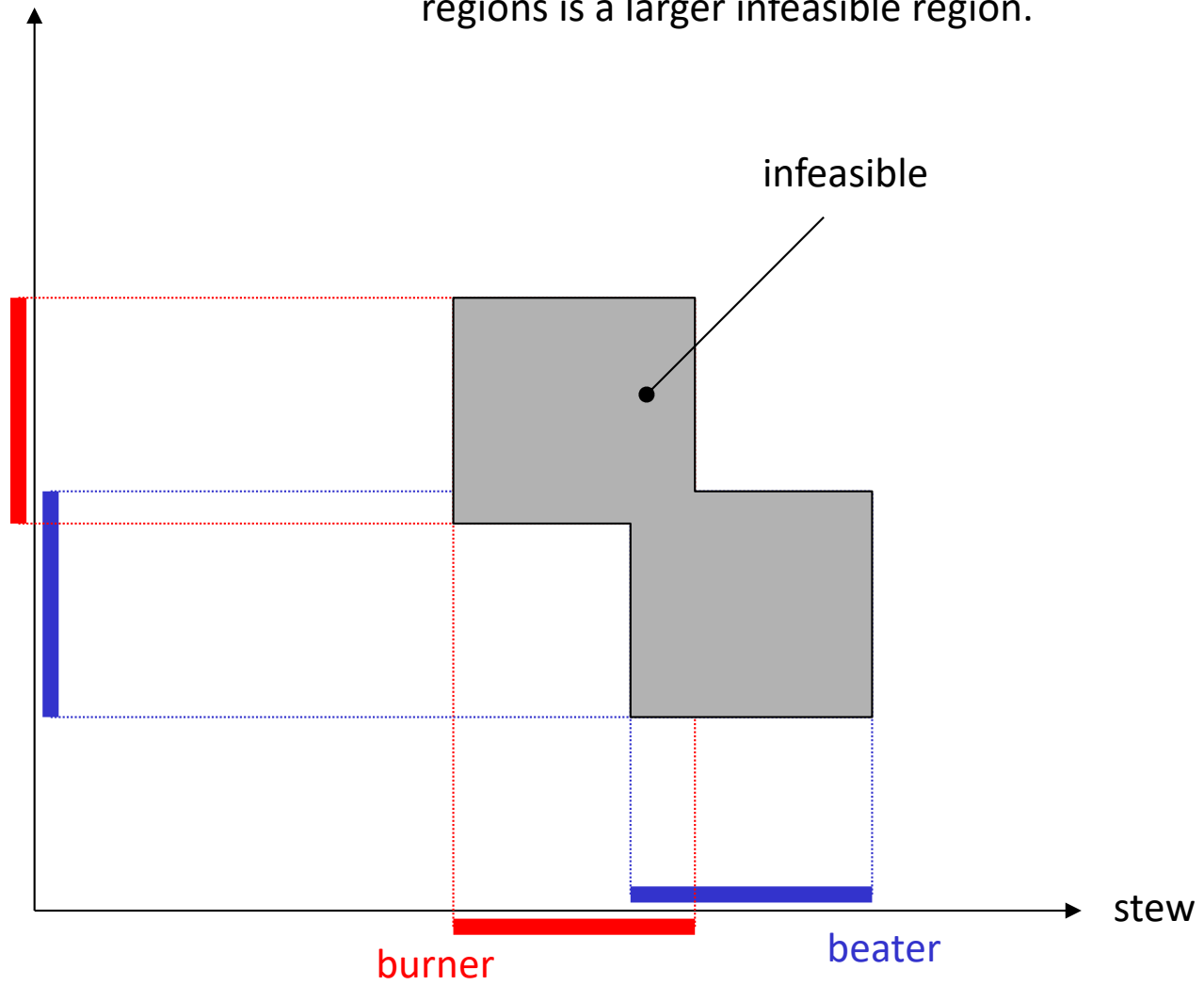
Some regions of progress space are infeasible because they represent both tasks using the same resource.

The red and blue boxes are infeasible.



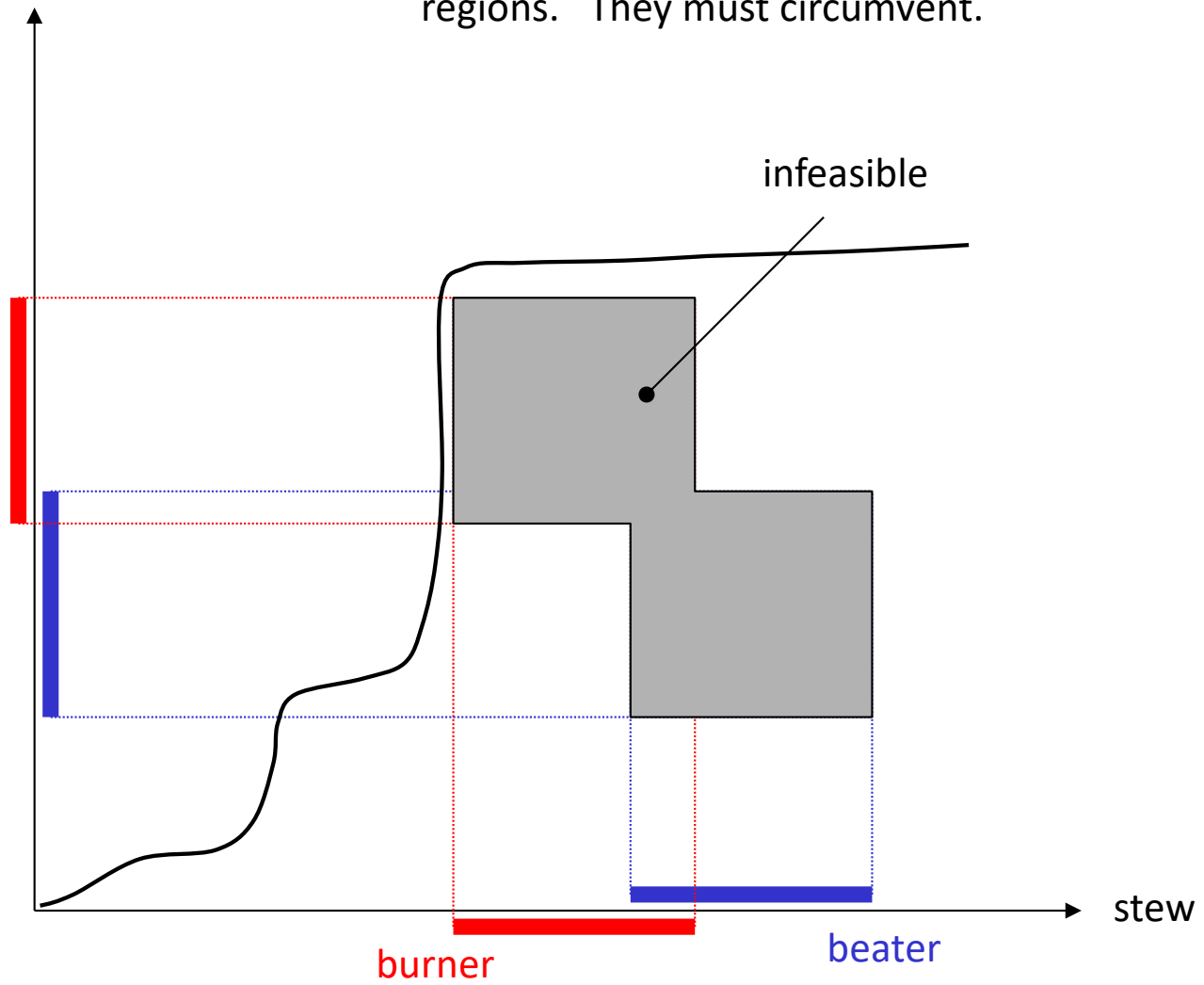
pudding

The union of the red and blue infeasible regions is a larger infeasible region.



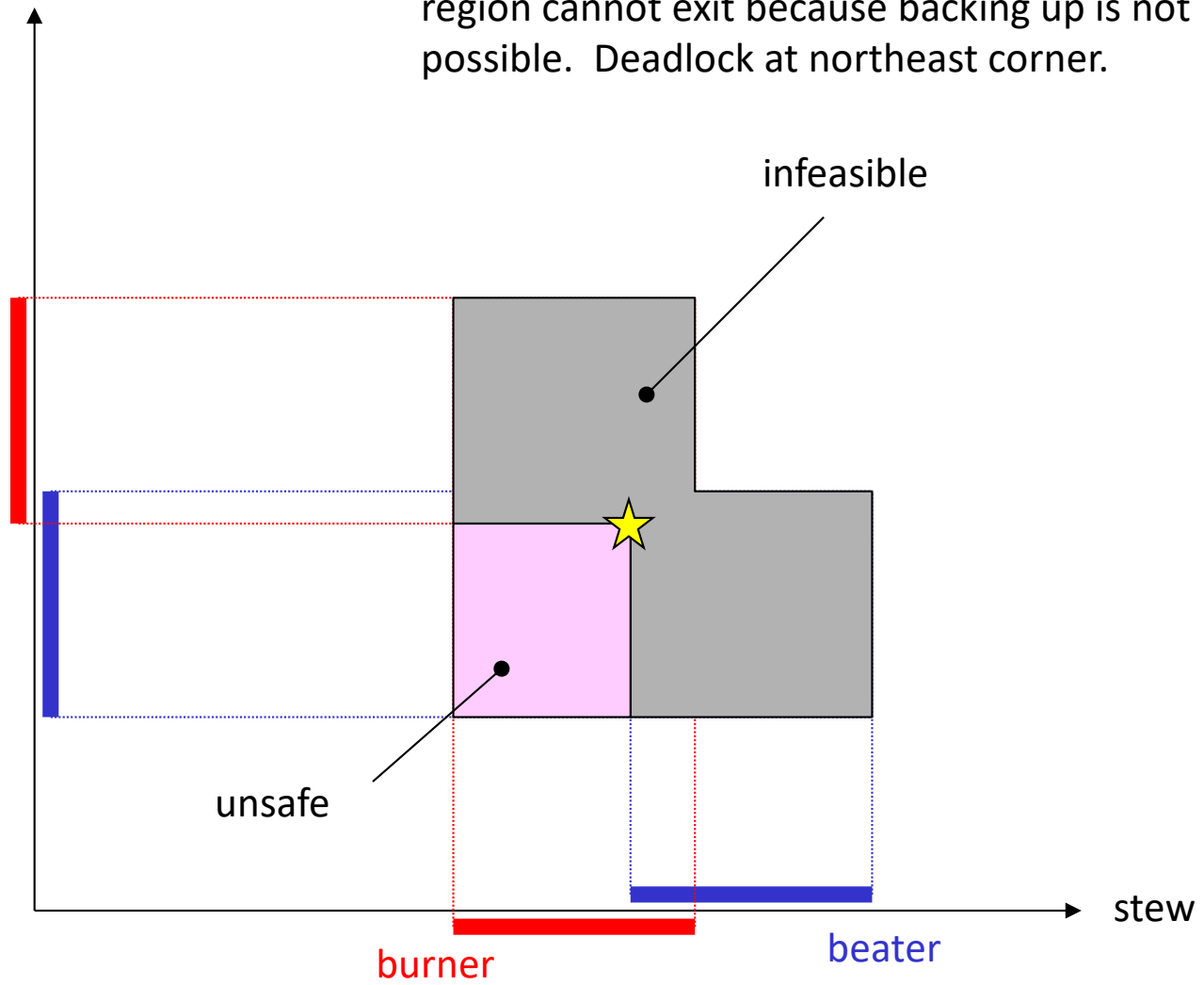
pudding

Progress paths cannot enter infeasible regions. They must circumvent.

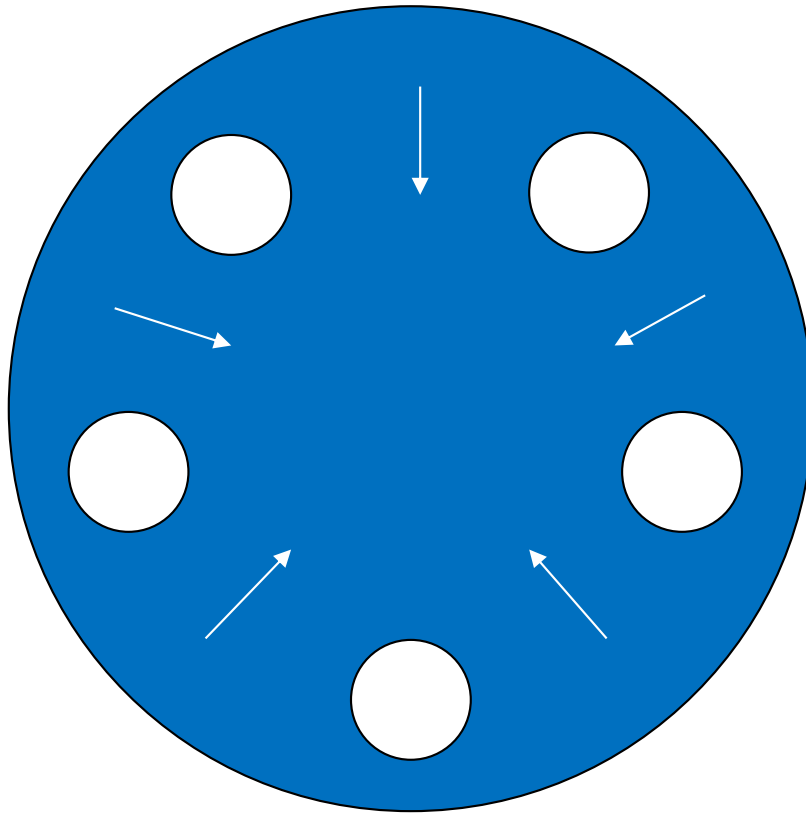


pudding

Any joint progress paths entering the unsafe region cannot exit because backing up is not possible. Deadlock at northeast corner.



# Philosophers Example

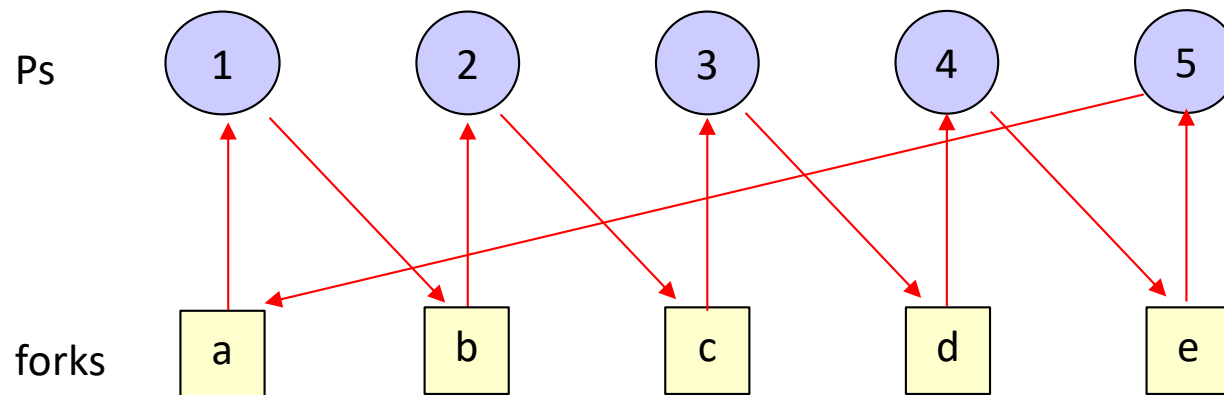


P comes to table  
Wait(left-fork)  
Wait(right-fork)  
Eats  
Release forks  
Departs

Deadlock occurs if all  
5 P's come at once and do  
their first wait together

How to prevent?

The DP problem is a stress  
test for proposed methods  
of preventing deadlock



DEADLOCK: each waiting for the next to release a fork, but all forks are in use

# Deadlock condition

For each deadlocked task, the set of unfilled requests is not covered by the sum of the available resources plus all resources held by non-deadlocked tasks.




# Basic Detection Algorithm

Set  $D = \{\text{all tasks}\}$

Set  $A = \text{vector of resources available}$

Find task  $i$  in  $D$  such that  $A$  covers  $\text{requests}(i)$ ;  
remove  $i$  from  $D$  and  
add  $\text{holdings}(i)$  to  $A$

That means this task  $i$  can complete; therefore pretend it finishes and releases all resources it holds



Repeat until no more tasks qualify for removal

Then  $D$  is the set of deadlocked tasks.

Even though simple, running this algorithm every time a deadlock is threatened would be very expensive

# Deadlock Coping Methods

- detection and recovery
- dynamic control of joint progress paths
- prior prevention -- negate the essential condition of resource waiting -- no circular wait possible

# Detection and Recovery

- If deadlock suspected, apply basic algorithm
- If algorithm stops with nonempty set of tasks, they are the deadlocked ones
- Abort the deadlocked tasks
- **Very expensive**

# Path Monitoring and Control

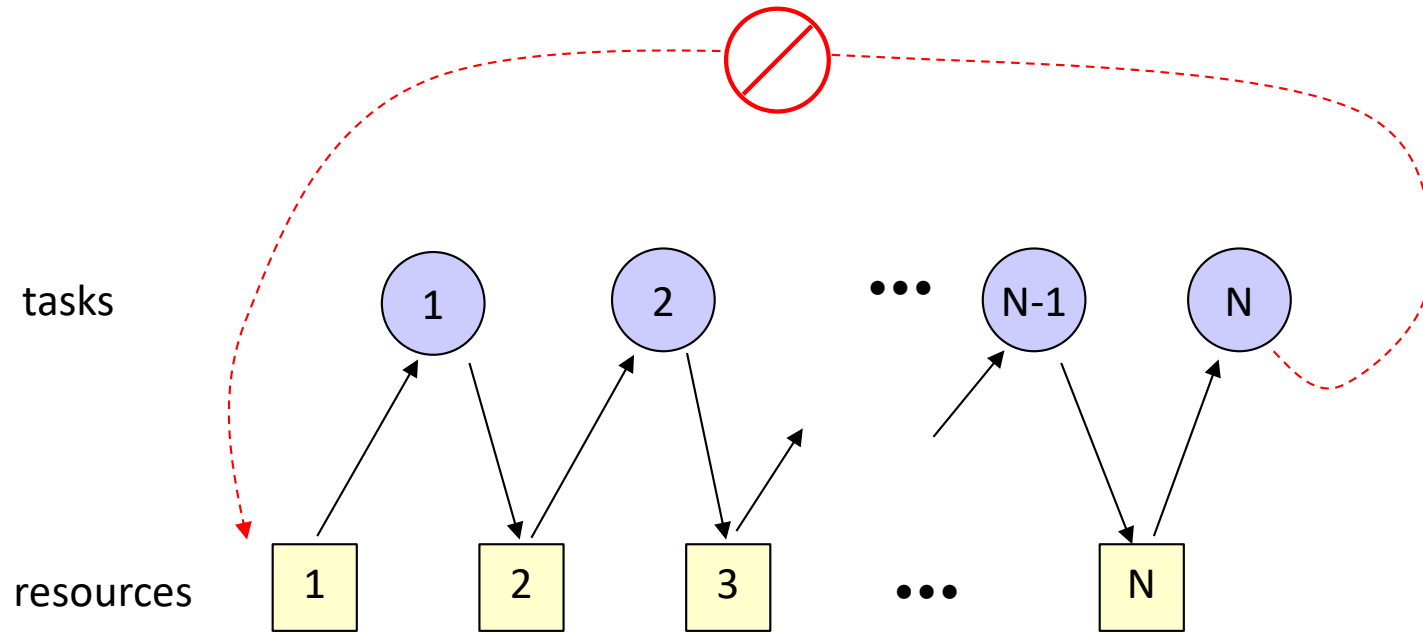
- Monitor joint progress path
- If a process's next move would enter unsafe region, block the move
- Finding a safe path is exhaustive search problem requiring detailed information about future process moves
- Beyond very expensive: intractable

# Prevention

- Design system so that the necessary condition of resource waiting is impossible
- Grant all resources before starting task
  - no additional requests allowed after start
  - two phase locking works this way
  - no deadlock because never wait
- Ordered resource protocol

# Ordered Resource Protocol

- ordered resource usage:
  - resources in groups 1,2,3,...
  - if task requests more, they must come from higher numbered group than any in which task holds resources



If  $N$  tasks are in circular wait, we can number them and the resources to get this diagram. Then task  $N$  must be holding the resource needed by task  $N-1$  and requesting the resource held by task 1. This violates the protocol and is impossible.

# Application to Transactions

- Program transactions to sort the incoming account numbers in ascending order
- Now all requests and waits will follow the ordering protocol



# Application to Banking

- Protocol inapplicable if all the loan money is in one pool ... subsequent requests violate the protocol
- However, it could work if pool were divided into multiple subpools ... get next increment from next pool
- Can also set loan pool size to be larger than sum of all lines of credit

# Application to Buffers

- Ordered protocol inapplicable because processes holding buffers can ask for more
- Can avoid running out of buffers by putting a limit  $B$  on how many buffers a process can hold; size the buffer pool at  $N * B$  buffers, where  $N$  is the maximum number of processes the system allows

# Application to Kitchen

- Ordered protocol is inapplicable
- Each task requires the two resources in a particular order ... the orders conflict
- Must run the tasks in serially ... completely cook one dish before starting the other

# Application to Philosophers

- Number forks 1,2,3,4,5
- Change acquisition protocol to
  - lower numbered fork first, higher numbered second
- Philosophers 1-4 find lower on left, upper on right
- Philosopher 5 finds lower on right, upper on left
- Now no deadlock is possible