

# **CS310 Operating Systems**

## **Lecture 28: Introduction to Semaphores and Deadlocks**

Ravi Mittal  
IIT Goa

# In the class we will study

- Semaphore Introduction
- Deadlock Introduction

# Acknowledgements !

- Contents of this class presentation has been taken from various sources. Thanks are due to the original content creators:
  - CS162, Operating System and Systems Programming, Sam Kumar, University of California, Berkeley,
  - Book: Operating Systems: Principles and Practice: Thomas Anderson and Michael Dahlin, Volume II, Chapter 5
  - Book: Modern Operating Systems, Fourth Edition, Andrew Tenenbaum, Herbert Bos, Pearson Publication
    - Chapter 2.3

# Reading

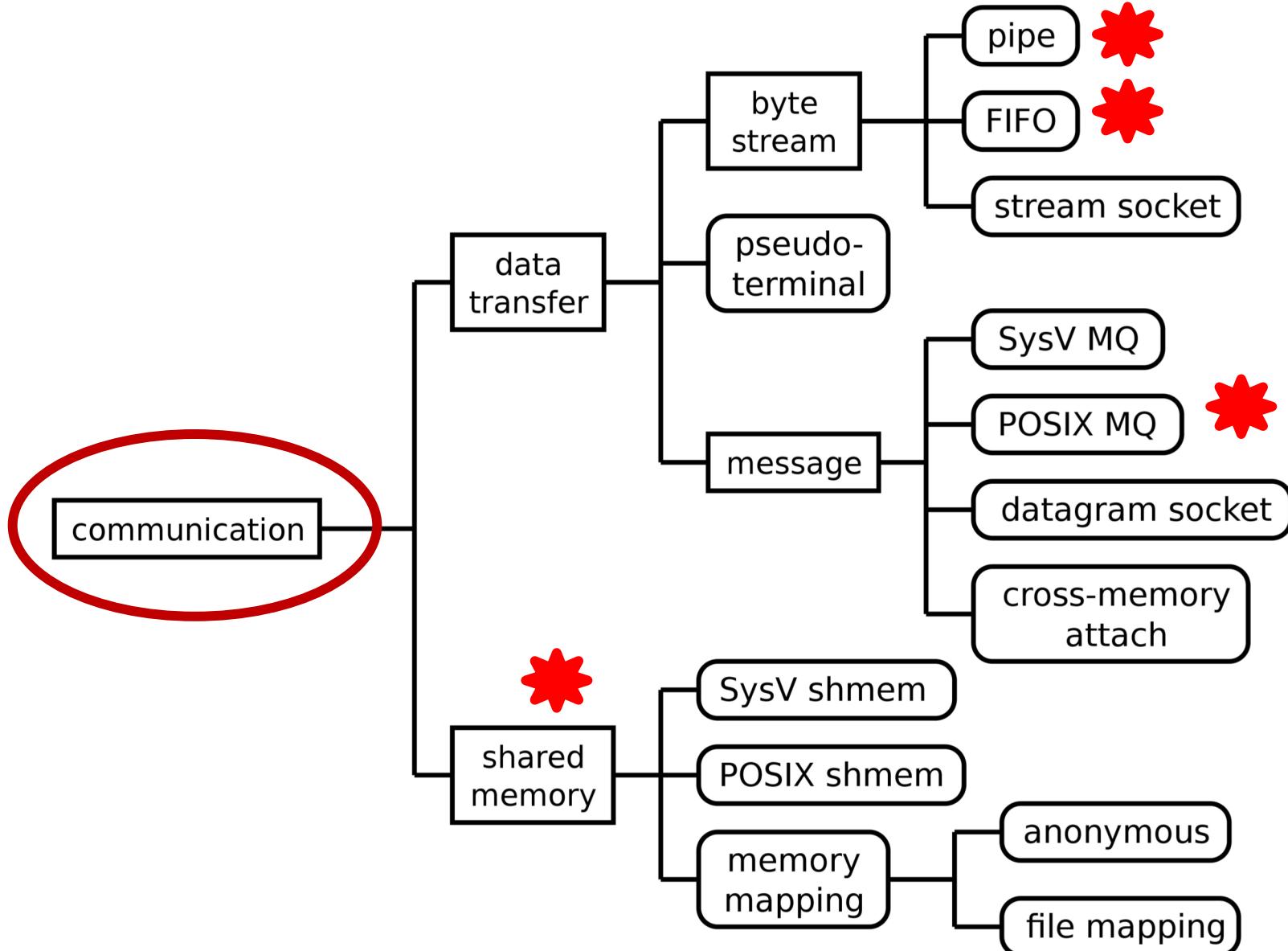
- Book: Operating System: Three Easy Pieces, by Remzi H Arpacı-Dusseau, Andrea C Arpacı-Dusseau
  - Chapter 31,  
<https://pages.cs.wisc.edu/~remzi/OSTEP/threads-semaphore.pdf>
- Book: Operating Systems: Principles and Practice: Thomas Anderson and Michael Dahlin, Volume II, Chapter 5.8

# **Previous Classes on IPC and Synchronization**

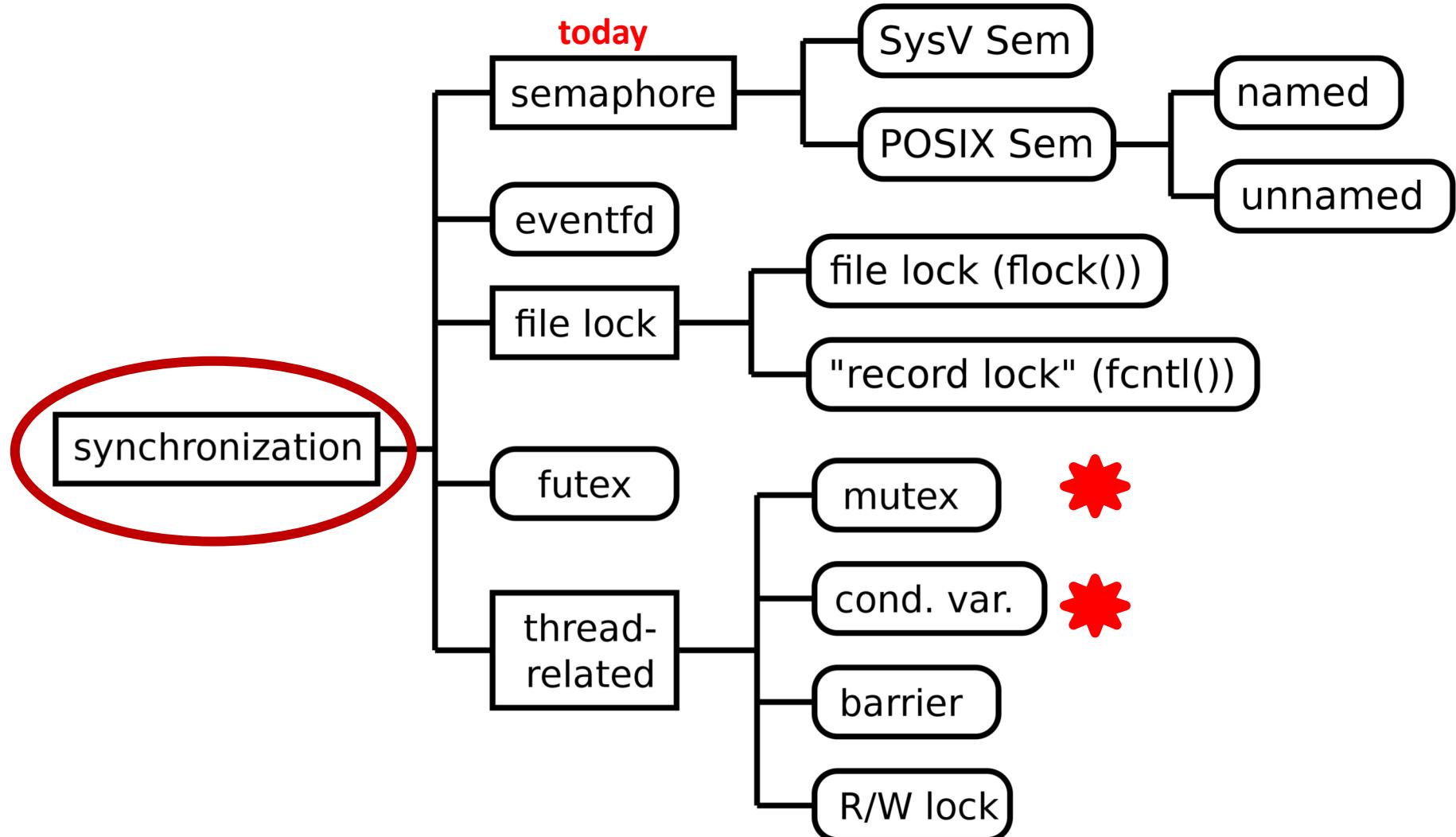
# So far we have studied

- Threads
- Processes
- Concurrent execution of Threads and Processes require
  - Communication
  - Synchronization
- Inter-process Communication methods
  - Message Passing
    - Message Queues
    - Pipes
    - Named Pipes or FIFO
  - Shared Memory
- Synchronization
  - Lock, Lock implantation: Test-and-Set, Compare-and-swap
  - Condition Variables

# Communication



# Needs for Synchronization



# We will start with High level primitives

|                  |  |            |          |        |  |
|------------------|--|------------|----------|--------|--|
| Programs         | Shared Programs                            |            |          |        |  |
| Higher-level API | Locks                                      | Semaphores | Monitors | Others |  |
| Hardware         | Disable Ints Test&Set Compare&Swap, others |            |          |        |  |

- Today we will study Semaphores

# Semaphore Introduction



# Semaphore

- Semaphore = a synchronization primitive
  - Higher level of abstraction than locks
  - invented by Dijkstra in 1968, as part of the THE operating system
  - Synchronization tool that does not require busy waiting
    - So it doesn't waste CPU time
  - An integer value used for signaling among processes
- Fundamental Principle:
  - Two or more processes want to cooperate by means of simple signals
- Semaphores can be used both as **locks** and **condition variables**

# Semaphore Definition

- A semaphore is an object with an integer value
  - We can manipulate semaphore with two routines:
    - `sem_wait()`
    - `sem_post()`
- The initial value to the semaphore determines its behavior
  - So, we must initialize it to some value
- Initializing Semaphore:
  - `s`: semaphore
  - Initial value: 1
    - The third argument
- Second argument to `sem_init()` is set to 0
  - This indicates that the semaphore is shared between threads in a process
  - Other options: see man pages

```
1 #include <semaphore.h>
2 sem_t s;
3 sem_init(&s, 0, 1);
```

# Semaphore Definition

```
1 #include <semaphore.h>
2 sem_t s;
3 sem_init(&s, 0, 1);
```

- After initialization we can call `sem_wait()` or `sem_post()`
- `sem_wait()`
  - Returns right away – if the value of semaphore is 1 or more when `sem_wait()` was called, Enter critical section
  - If semaphore value was **zero or negative** --> caller to suspend execution
    - Waits for subsequent post – puts itself to sleep
    - If multiple calling threads may call `sem_wait()`
      - Gets queued waiting to be woken
- `sem_post()`
  - It simply increments the value of the semaphore
  - If there is any thread waiting to be woken, one is woken up
- The negative value of the semaphore = number of waiting threads

# Semaphore Definition

- The semaphore value is not seen by the users of semaphores
- `wait()` and `post()`

```
1 int sem_wait(sem_t *s) {
2     decrement the value of semaphore s by one
3     wait if value of semaphore s is negative
4 }
5
6 int sem_post(sem_t *s) {
7     increment the value of semaphore s by one
8     if there are one or more threads waiting, wake one
9 }
```

# Binary Semaphore (Locks)

```
1 sem_t m;  
2 sem_init(&m, 0, X); // initialize to X; what should X be?  
3  
4 sem_wait(&m);  
5 // critical section here  
6 sem_post(&m);
```

- What should be the value of X ?

# Binary Semaphore (Locks)

```
1 sem_t m;  
2 sem_init(&m, 0, X); // initialize to X; what should X be?  
3  
4 sem_wait(&m);  
5 // critical section here  
6 sem_post(&m);
```

- What should be the value of X ?
- Answer: 1

# First Case: Single Thread using semaphore

- Consider two threads
  - Only one thread is using Semaphore

| Value of Semaphore | Thread 0           | Thread 1 |
|--------------------|--------------------|----------|
| 1                  |                    |          |
| 1                  | call sem_wait()    |          |
| 0                  | sem_wait() returns |          |
| 0                  | (crit sect)        |          |
| 0                  | call sem_post()    |          |
| 1                  | sem_post() returns |          |

## Second Case: Two threads using a Semaphore

- Initial Value of semaphore is 1
- Thread 0 holds the lock using `sem_wait()`
- Thread 1 tries to enter critical section by calling `sem_wait()`
  - Thread 1 decrements the value of semaphore to -1 and waits by putting itself to sleep
  - It **relinquishes** the processor
- Thread 0 runs again
  - Eventually calls `sem_post()`
  - Increments semaphore to 0
  - Wakes up sleeping thread 1
- Thread 1 now enters critical section
  - Note that value of semaphore remains 0 and thread 1 has already decremented the semaphore .. It's not checked again

# Binary Semaphore (Locks) – Another case

| Val | Thread 0                    | State | Thread 1           | State |
|-----|-----------------------------|-------|--------------------|-------|
| 1   |                             | Run   |                    | Ready |
| 1   | call sem_wait()             | Run   |                    | Ready |
| 0   | sem_wait() returns          | Run   |                    | Ready |
| 0   | (crit sect begin)           | Run   |                    | Ready |
| 0   | <i>Interrupt; Switch→T1</i> | Ready |                    | Run   |
| 0   |                             | Ready | call sem_wait()    | Run   |
| -1  |                             | Ready | decr sem           | Run   |
| -1  |                             | Ready | (sem<0) → sleep    | Sleep |
| -1  |                             | Run   | <i>Switch→T0</i>   | Sleep |
| -1  | (crit sect end)             | Run   |                    | Sleep |
| -1  | call sem_post()             | Run   |                    | Sleep |
| 0   | incr sem                    | Run   |                    | Sleep |
| 0   | wake(T1)                    | Run   |                    | Ready |
| 0   | sem_post() returns          | Run   |                    | Ready |
| 0   | <i>Interrupt; Switch→T1</i> | Ready |                    | Run   |
| 0   |                             | Ready | sem_wait() returns | Run   |
| 0   |                             | Ready | (crit sect)        | Run   |
| 0   |                             | Ready | call sem_post()    | Run   |
| 1   |                             | Ready | sem_post() returns | Run   |

# Semaphore for Ordering

- We can use semaphores to order events in a concurrent program
- For example
  - A thread may wait for a list to become non-empty so that it can delete an item from it
- In general, we can use semaphores to develop programs where
  - One thread is *waiting* for something to happen
  - Another thread making that something happen and then *signaling* that it has happened
    - Thus, waking the sleeping thread
  - Similar to use of condition variable
- Example: Imagine a thread creates another thread and then wants to wait for it to complete its execution

# Example: A parent waiting for it's child

```
1  sem_t s;
2
3  void *child(void *arg) {
4      printf("child\n");
5      sem_post(&s); // signal here: child is done
6      return NULL;
7  }
8
9  int main(int argc, char *argv[]) {
10     sem_init(&s, 0, X); // what should X be?
11     printf("parent: begin\n");
12     pthread_t c;
13     Pthread_create(&c, NULL, child, NULL);
14     sem_wait(&s); // wait here for child
15     printf("parent: end\n");
16     return 0;
17 }
```

# Example: A parent waiting for it's child

- Execution of the program results prints:

Parent: begin

child

Parent: end

- Value of the semaphore should be 0

# Example: A parent waiting for its child

- Case 1

- Parent has created child but child has not run, yet (in ready queue)

---

| Val | Parent             | State | Child                   | State |
|-----|--------------------|-------|-------------------------|-------|
| 0   | create(Child)      | Run   | (Child exists, can run) | Ready |
| 0   | call sem_wait()    | Run   |                         | Ready |
| -1  | decr sem           | Run   |                         | Ready |
| -1  | (sem<0) → sleep    | Sleep |                         | Ready |
| -1  | Switch→Child       | Sleep | child runs              | Run   |
| -1  |                    | Sleep | call sem_post()         | Run   |
| 0   |                    | Sleep | inc sem                 | Run   |
| 0   |                    | Ready | wake (Parent)           | Run   |
| 0   |                    | Ready | sem_post() returns      | Run   |
| 0   |                    | Ready | Interrupt→Parent        | Ready |
| 0   | sem_wait() returns | Run   |                         | Ready |

# Example: A parent waiting for its child

- Case 2
  - Child gets to completion before parent gets chance to call `wait()`

| Val | Parent                   | State | Child                     | State |
|-----|--------------------------|-------|---------------------------|-------|
| 0   | create(Child)            | Run   | (Child exists; can run)   | Ready |
| 0   | <i>Interrupt</i> → Child | Ready | child runs                | Run   |
| 0   |                          | Ready | call sem_post()           | Run   |
| 1   |                          | Ready | inc sem                   | Run   |
| 1   |                          | Ready | wake(nobody)              | Run   |
| 1   |                          | Ready | sem_post() returns        | Run   |
| 1   | parent runs              | Run   | <i>Interrupt</i> → Parent | Ready |
| 1   | call sem_wait()          | Run   |                           | Ready |
| 0   | decrement sem            | Run   |                           | Ready |
| 0   | (sem ≥ 0) → awake        | Run   |                           | Ready |
| 0   | sem_wait() returns       | Run   |                           | Ready |

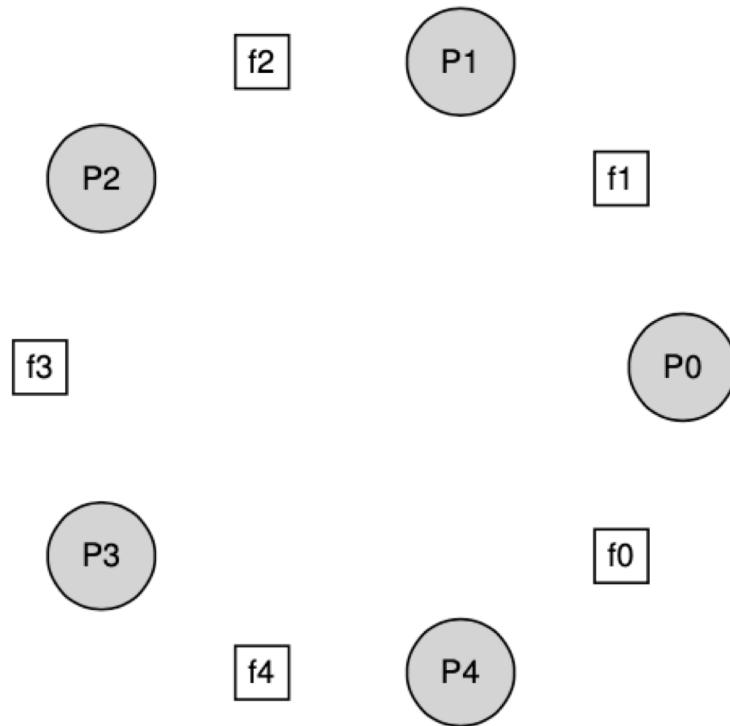
# Producer Consumer Problem

- We can easily solve producer consumer problem using semaphores
- Note that solution must avoid deadlock
  - Improper use of semaphores may cause both consumer and producer to block
  - Example: The consumer holds the mutex (semaphore) and is waiting for someone to signal. Producer could signal but it is waiting for the mutex.
    - Producer and consumer are stuck – waiting for each other
      - Deadlock
- Read details from Section 31.5 OSTEP book

# The Dining Philosophers

- Assume there are five “philosophers” sitting around a table
- Between each pair of philosophers is a single fork (and thus, five total)
- The philosophers each have
  - times where they think, and don’t need any forks
  - Times when they eat – They need both forks
    - One on their right and one on their left
- How do they solve it for contention?
- How can they synchronize?

# Dining Philosopher Problem



# Dining Philosopher problem

```
while (1) {  
    think();  
    get_forks(p);  
    eat();  
    put_forks(p);  
}
```

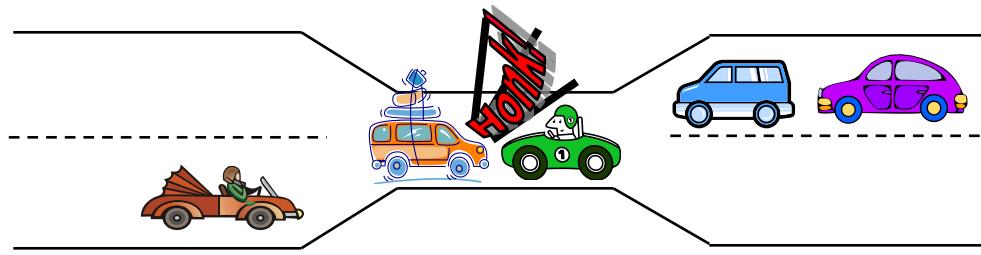
- How to write `get_fork()` and `put_fork()` such that
  - There is no deadlock
  - No philosopher starves
  - Concurrency is high (as many philosophers can eat at the same time as possible)

# **Deadlock**

# Example: Single-Lane Bridge Crossing



# Bridge Crossing Example



- Each segment of road can be viewed as a resource
  - Car must own the segment under them
  - Must acquire segment that they are moving into
- **Deadlock:** Two cars in opposite directions meet in middle
- **Starvation** (not deadlock): Eastbound traffic doesn't stop for westbound traffic

# Deadlock Definition

- A deadlock is a cycle of waiting among a set of threads, where each thread waits for some other thread in the cycle to take some action
- Deadlocks can happen due to many conditions

# Lecture Summary

- Semaphores are a powerful and flexible primitive for writing concurrent programs
- One could view semaphores as a generalization of locks and condition variables
- Deadlock can happen where each process/thread waits for some other thread in the cycle to take some action