Operating Systems!

Concurrency: **Synchronization (Part 2)**

Prof. Travis Peters

Montana State University

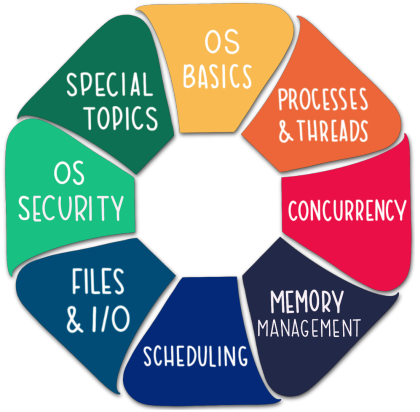
CS 460 - Operating Systems

Fall 2020

https://www.cs.montana.edu/cs460

Some diagrams and notes used in this slide deck have been adapted from Sean Smith's OS courses @ Dartmouth. Thanks, Sean!

Today



- Announcements
  - ~~Yalnix~~Project! Form teams *ASAP!* ☺

Travis has already helped a few "Free Agents" form team members! Great to have a group to discuss things (e.g., Accountability, PA2, Studying)

  - *Exam 1:* ***Friday [10/02/2020] – due @ 11:59 PM (MST)***
    - **Read over the exam coversheet BEFORE Friday!**
  - *PA2 due* ***Sunday [10/04/2020] @ 11:59 PM (MST)***
    - **USE the PA2 starter files! (You need to make *very minor updates* to the Makefile)**
  - *Office Hours*
    - **Extra Office Hours ~~Tuesday (8-11am)~~ + Thursday (8-11am)**
    - **NO OFFICE HOURS FRIDAY**

Today *(cont.)*

- *Last Time: Interleaving, Race Conditions, Naive Synchronization, …*
- *Last Time: OS Sync., Locks, Condition Variables, (Semaphores)*
- Agenda
  - Classic Concurrency Scenarios
    - Producer/Consumer, Readers/Writers, Sleeping Barber, Dining Philosophers
  - Example
  - Deadlock & Strategies

Classic Concurrency Scenarios

A Synopsis

Producer/Consumer Problem

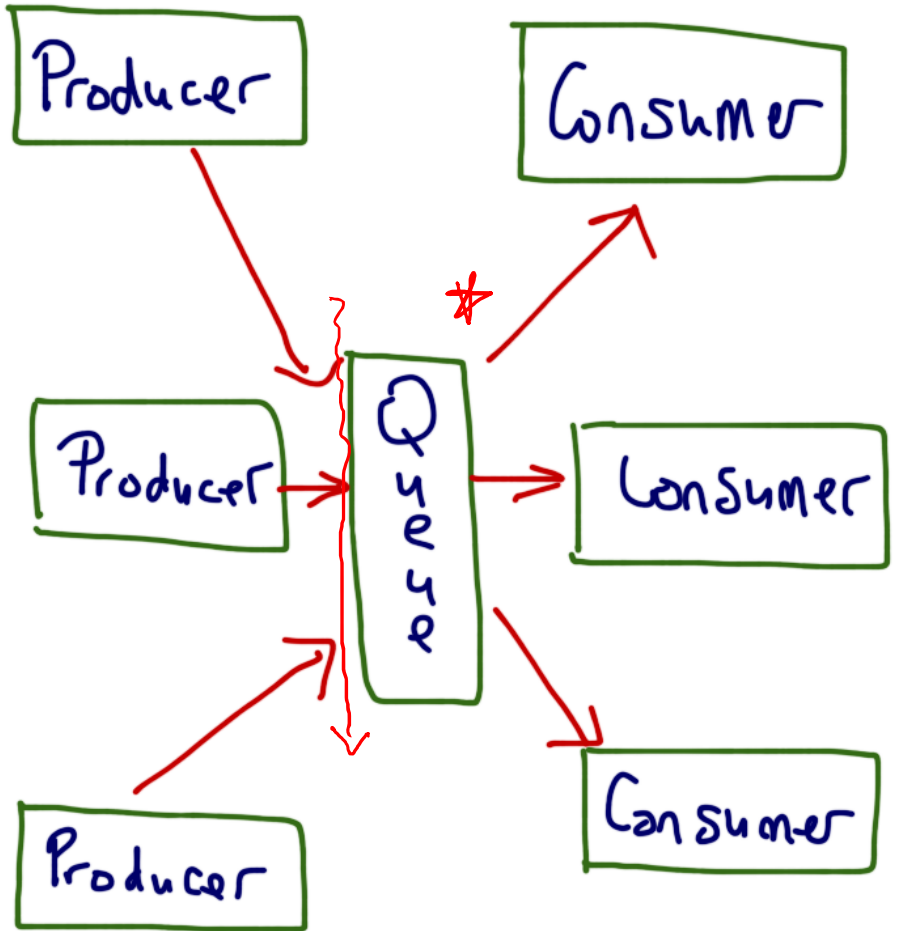
- Producers **produce** items
- Consumers **consume** items

- Store items in a **shared buffer**
  - What are some bad interleavings?
  - What if the buffer is full?
  - What if the buffer is empty?

- Variations
  - Infinite Buffer
  - Finite, Circular Buffer
  - one (or many) producers
  - one (or many) consumers

ls | grep

pipes



http://javarevisited.blogspot.com/2017/02/10-java-wait-notify-locking-and-synchronization-Interview-Questions-Answers.html

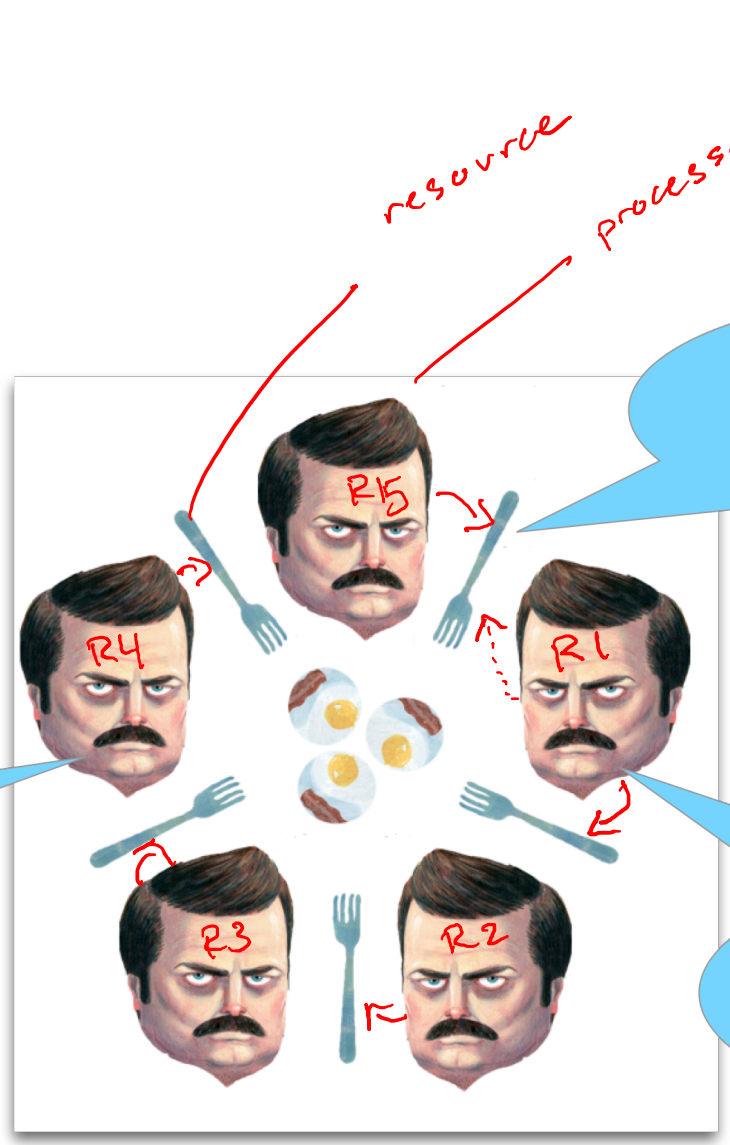
Readers/Writers Problem

- Each process is either a **reader** or a **writer**
- Both readers and writers share access to a data object (e.g., file, database)
- Multiple readers can access the data object simultaneously
- Each writer must have exclusive access (i.e., cannot share w/ readers OR any other writer)

What are some bad interleavings?

Dining Philosophers Problem

- No two **philosophers** (Rons) can hold the same **fork** at the same time
- No philosopher (Ron) should **starve** to death



Each Fork:
Who is holding me?

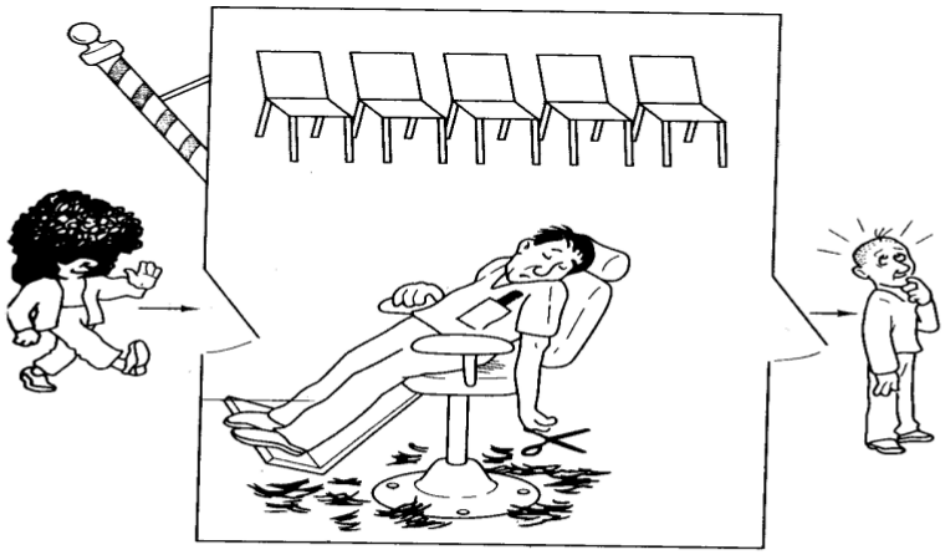
Each Ron:
I'm Hungry…
Give me bacon and eggs…

Each Ron (Philosopher):
Which fork(s) are mine?

What are some bad interleavings?

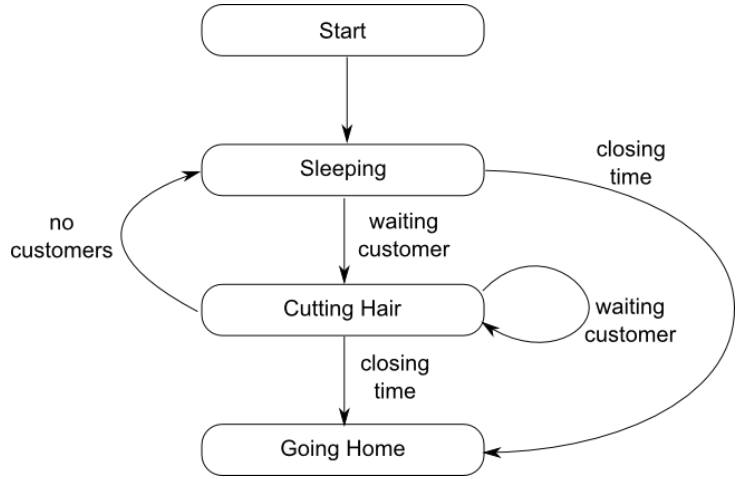
Sleeping Barber Problem

- 1 **barber**
  M **chairs**
  N **customers**



- No customers? ➔ barber takes a nap

- New customer arrives ➔ wake barber
  - What are some bad interleavings?
  - What if the barber is busy?
  - What if there are more waiting customers than available chairs?



Example

Walking through important aspects of the Readers/Writers problem

Readers/Writers

```
struct {
    data_t payload;
    int read_count;
    int write_count;
} reader_writer_t;
```

Readers/Writers

```
struct {
    data_t payload;
    int read_count;
    int write_count;
    lock_t lock;
} reader_writer_t;
```

Readers/Writers

```
struct {
    data_t payload;
    int read_count;
    int write_count;
    lock_t lock;
    cvar_t want_to_read;
    cvar_t want_to_write;
} reader_writer_t;
```

Readers/Writers

```
struct {
    data_t payload;
    int read_count;
    int write_count;
    lock_t lock;
    cvar_t want_to_read;
    cvar_t want_to_write;
} reader_writer_t;
```

```
// TO READ:
Acquire(lock);
while(writecount > 0)
    CvarWait(want_to_read);
readcount++;
Release(lock);

...do the reading...

Acquire(lock);
readcount--;
if (readcount == 0)
    CvarSignal(want_to_write);
Release(lock);
```

```
// TO WRITE:
Acquire(lock);
while ( (readcount > 0) || (writecount > 0) )
    CvarWait(want_to_write);
writecount++;
Release(lock);

... do the writing...

Acquire(lock);
writecount--;
CvarSignal(want_to_write); // why do we do this?
CvarBroadcast(want_to_read); // why do we do this?
Release(lock);
```

Deadlock

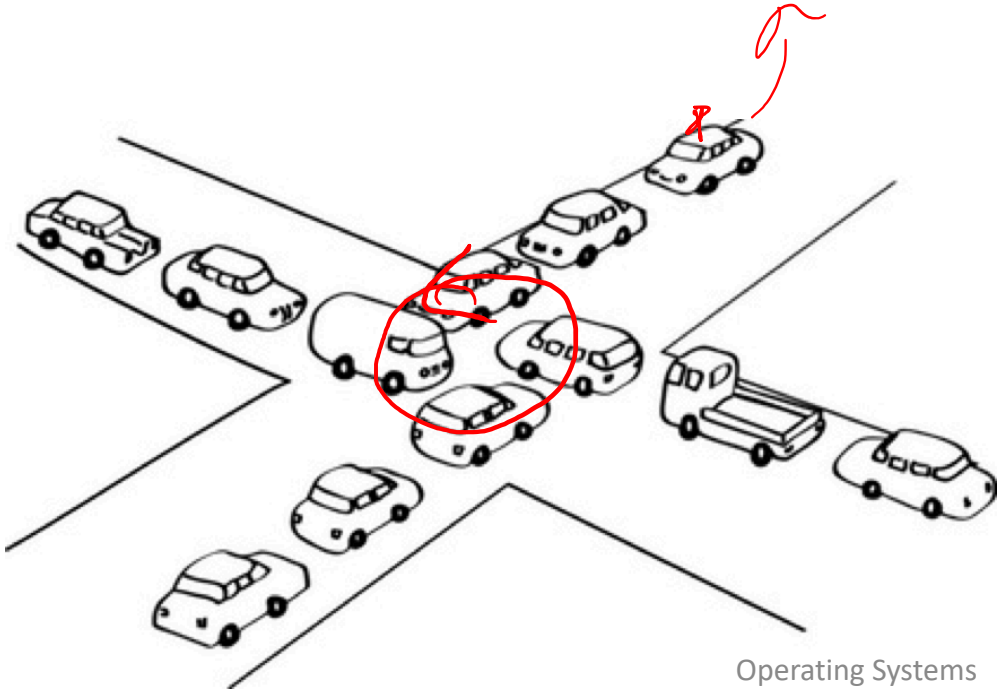
Deadlock

- The **permanent** blocking of a set of processes that either compete for system resources or communicate with each other.

- A set of processes is deadlocked when each process is blocked awaiting an event.

- We say permanent because none of the events is ever triggered…

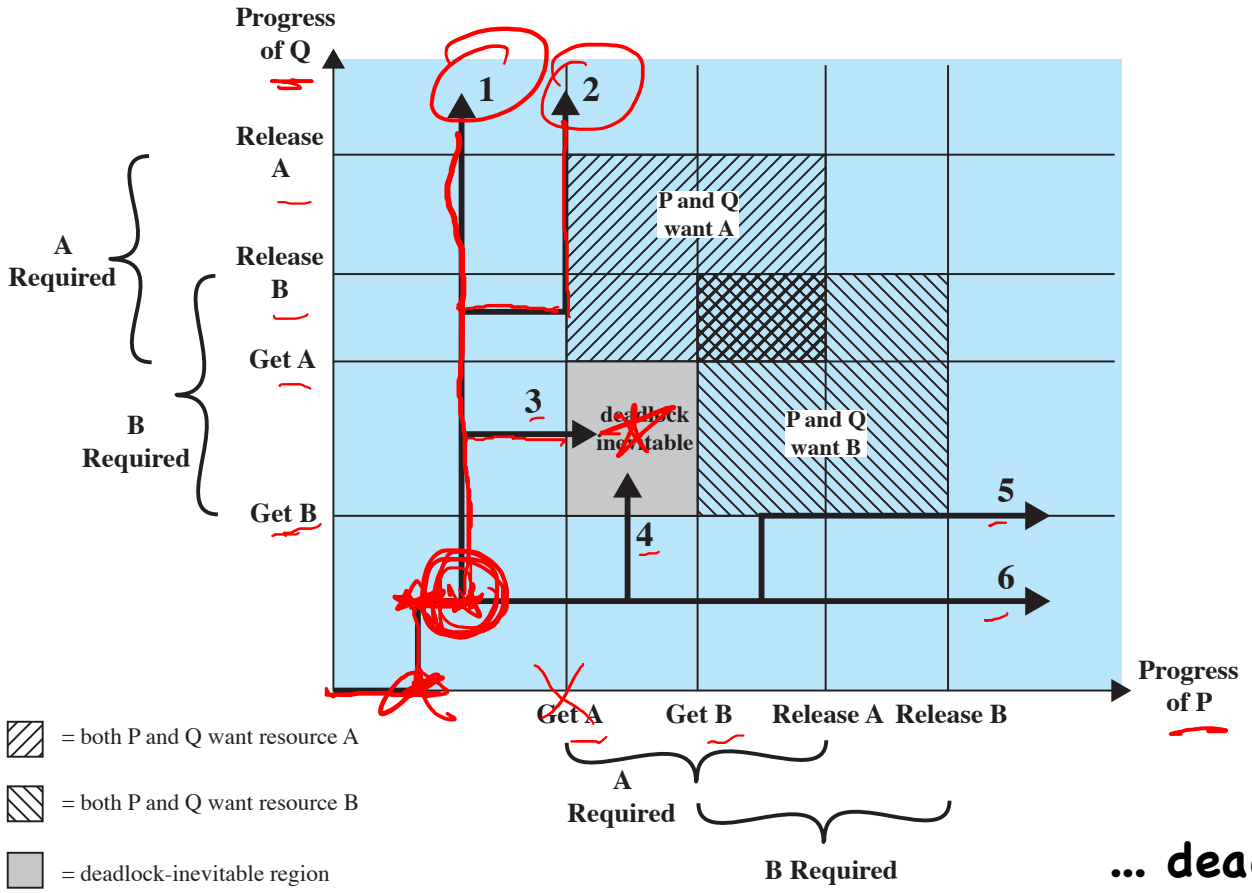
Unfortunately, there is no efficient solution in the general case…



Example: Two Competing Processes, Two Required Resources

Visualized with Joint Progress Diagrams

Not ALL execution paths lead to deadlock…



Resources: A B

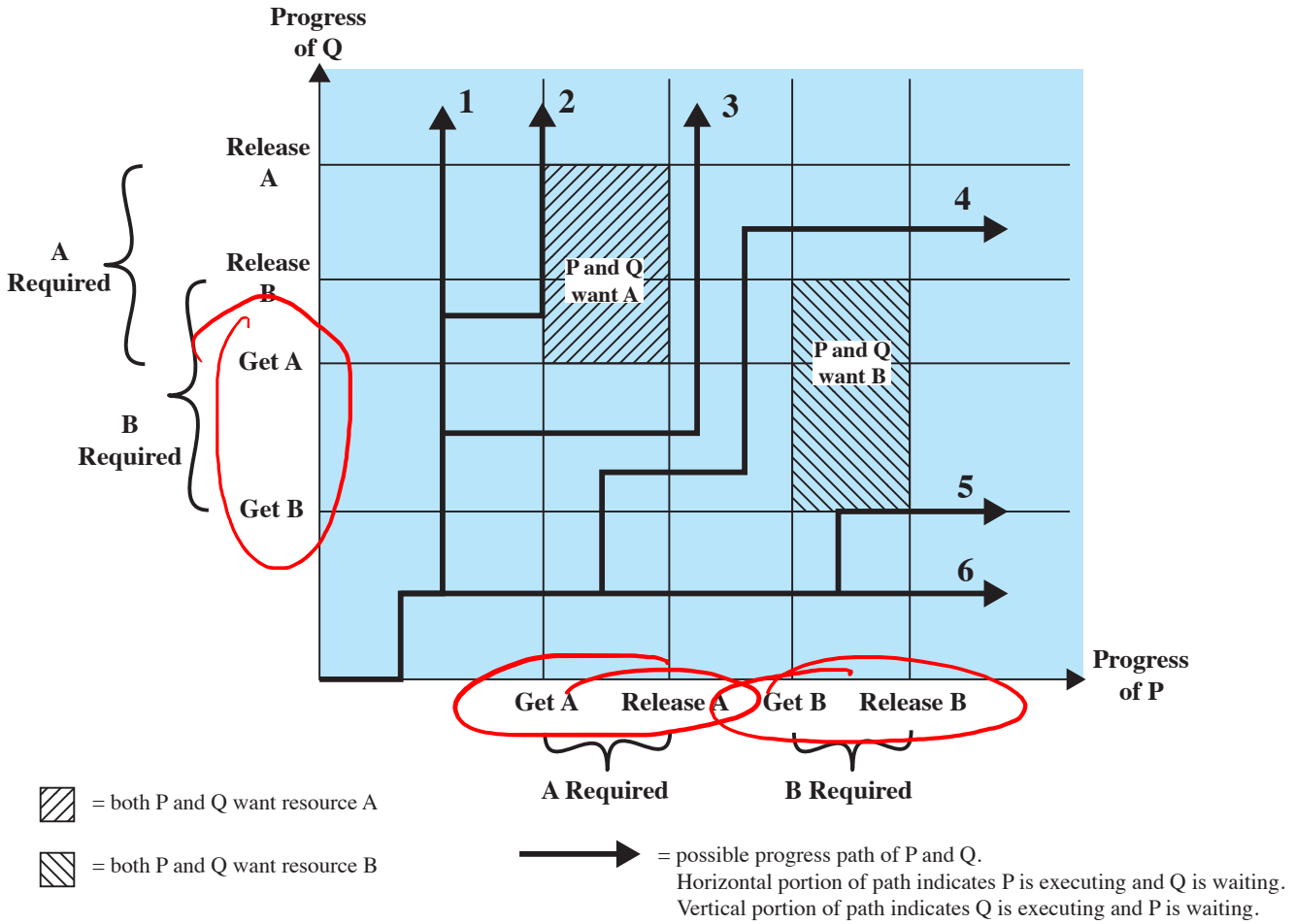
Process: Q P

… deadlock is only inevitable if both processes enter the "fatal region"

Example: Two Competing Processes, Two Required Resources

Visualized with Joint Progress Diagrams

If P doesn't require BOTH resources AT THE SAME TIME…



… NO DEADLOCK!

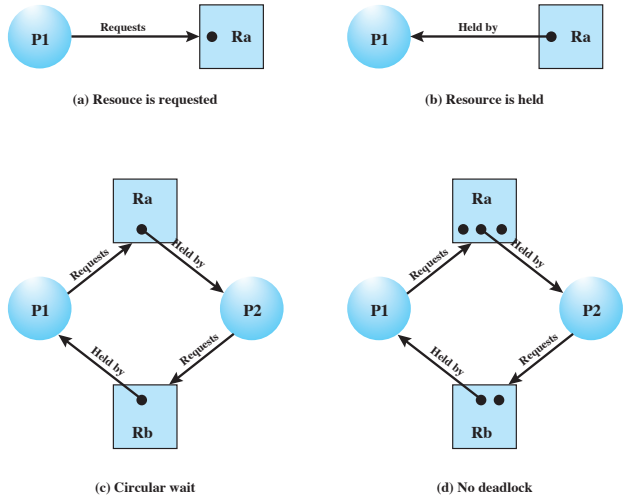
Strategies for Addressing Deadlock

No general solution... but we do have some good strategies!

- **Prevention*** — disallow/prevent one of the deadlock conditions; i.e., possibility of deadlock is excluded

- **Avoidance** — don't grant resource request if it might lead to deadlock

- **Detection** — grant resource requests when possible; periodically check for deadlock + take action to recover



The Conditions for Deadlock

For deadlock to occur, ***all 4 conditions must hold…***

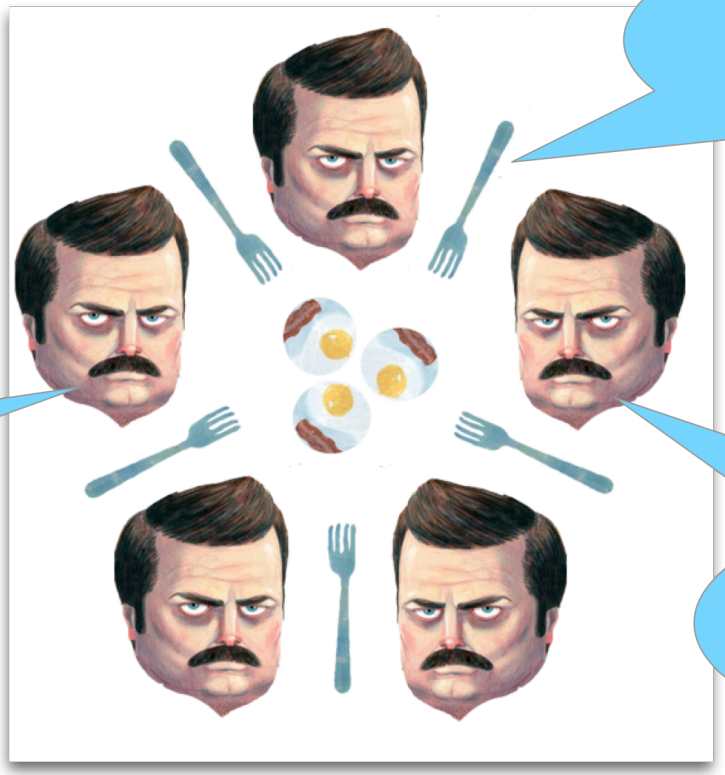
1. **Mutual Exclusion**
   Only 1 process can use a resource at a time. No other process may access that resource.
2. **Hold and Wait**
   A process may hold allocated resource while awaiting assignment of other resources.
3. **No Preemption**
   No resource can be forcibly removed from a process holding it.
4. **Circular Wait**
   A closed chain of processes exists s.t. each process holds at least one resource needed by the next process in the chain.

Condition for Deadlock:
1. Mutual Exclusion
2. Hold and Wait
3. No Preemption
4. Circular Wait

Dining Philosophers Problem

- No two **philosophers** (Rons) can hold the same **fork** at the same time

- No philosopher (Ron) should **starve** to death



Each Fork:
Who is holding me?

Each Ron:
I'm Hungry…
Give me bacon and eggs…

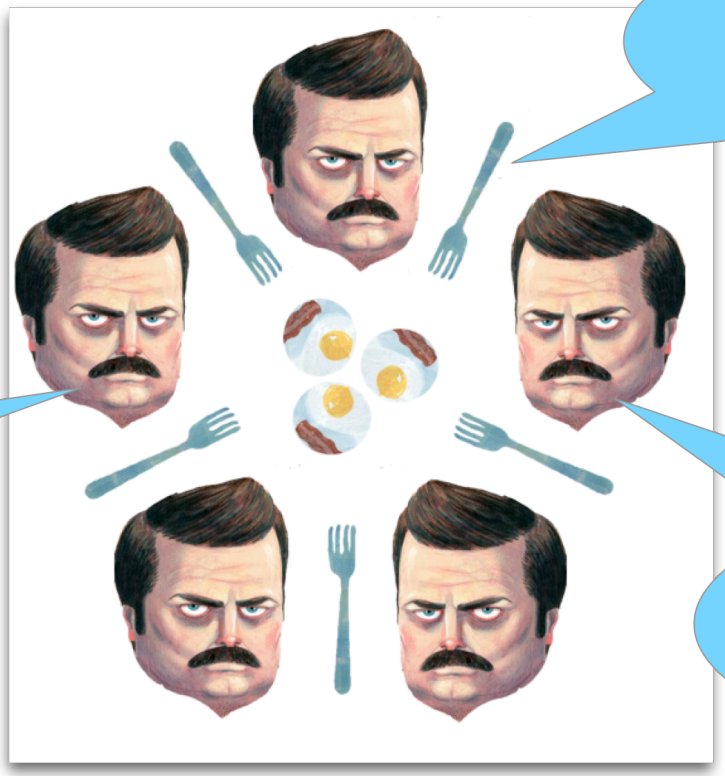
Each Ron (Philosopher):
Which fork(s) are mine?

What are some bad interleavings?

Dining Philosophers Problem

Condition for Deadlock:
1. Mutual Exclusion
2. Hold and Wait
3. No Preemption
4. Circular Wait

```
/* program      diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
     while (true) {
          think();
          wait (fork[i]);
          wait (fork [(i+1) mod 5]);
          eat();
          signal(fork [(i+1) mod 5]);
          signal(fork[i]);
     }
}
void main()
{
     parbegin (philosopher (0), philosopher (1), philosopher
(2),
          philosopher (3), philosopher (4));
     }
```



Each Fork:
Who is holding me?

Each Ron:
I'm Hungry…
Give me bacon and eggs…

Each Ron (Philosopher):
Which fork(s) are mine?

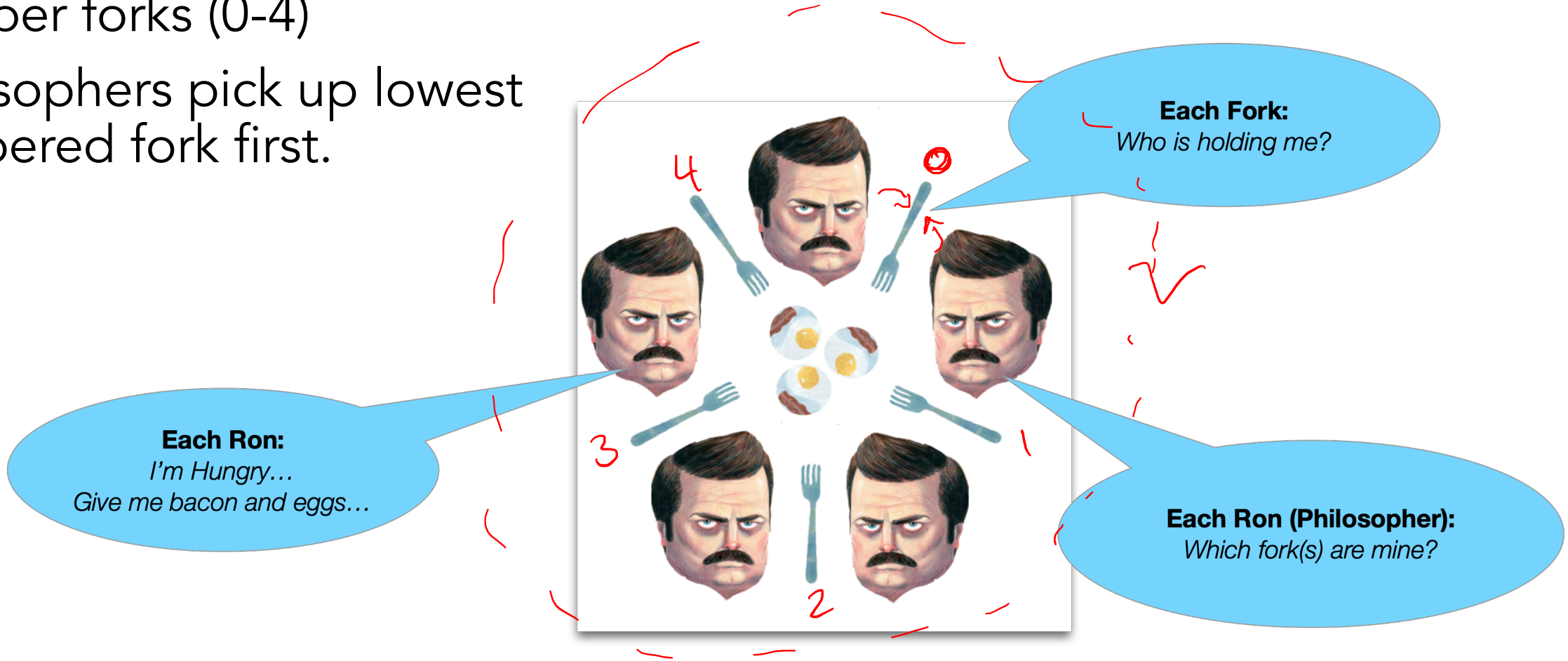
What are some bad interleavings?

Condition for Deadlock:
1. Mutual Exclusion
2. Hold and Wait
3. No Preemption
4. Circular Wait

Dining Philosophers Problem

Solution 1: Eliminate Circular Wait

- Number forks (0-4)
- Philosophers pick up lowest numbered fork first.

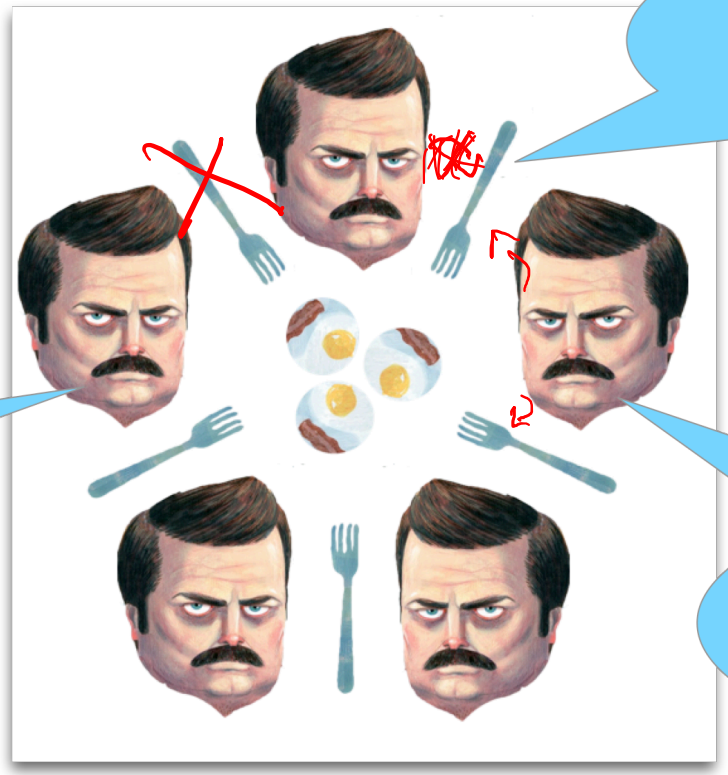


Condition for Deadlock:
1. Mutual Exclusion
2. Hold and Wait
3. No Preemption
4. Circular Wait

Dining Philosophers Problem

Solution 2: Eliminate Hold and Wait

- Must pick up both forks at one time
- No Ron can hold a fork unless they can also hold the other fork



Each Fork:
Who is holding me?

Each Ron:
I'm Hungry…
Give me bacon and eggs…

Each Ron (Philosopher):
Which fork(s) are mine?

Summary of Strategies for Approaching Deadlock

TL;DR Correctly addressing deadlock can be challenging – we need to consider all advantages/disadvantages

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	•Works well for processes that perform a single burst of activity •No preemption necessary	•Inefficient •Delays process initiation •Future resource requirements must be known by processes
		Preemption	•Convenient when applied to resources whose state can be saved and restored easily	•Preempts more often than necessary
		Resource ordering	•Feasible to enforce via compile-time checks •Needs no run-time computation since problem is solved in system design	•Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	•No preemption necessary	•Future resource requirements must be known by OS •Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	•Never delays process initiation •Facilitates online handling	•Inherent preemption losses

Reflection

Some topics and ideas to consider for assignments, and in general

Reflection

From **Birrell**: *"All shared mutable data must be protected by associating it with some mutex, and you must access the data only from a thread that is holding the associated mutex."*

Invariants

- Re: modifying complex data
- Not EVERY combination of field values are valid ALL THE TIME.
- An invariant is a Boolean function that describes which combinations are legal.
- An invariant is always true whenever someone is "looking."
- An invariant can be false when no one else is "looking."
- If the invariants "are too complicated to write down, you're probably doing something wrong."

Example: *A thread holds a lock, which provides mutual exclusion within a region of code that accesses shared data. That thread is free to modify the shared data, temporarily invalidating some invariant, so long as the invariant is restored before others can access the shared data.*

Reflection

Granularity

- How much should you lock?
- If you lock everything at a high-level, you get lots of needless conflicts, and potentially deadlock!
- If you lock everything at a low-level, you have more complexity, and a greater risk of getting it wrong.

Reflection

Two-Mutex (or Two-Cvar) Issues

- **Issue:** You can get deadlock! *(why?)*
- **Solution:** put the locks in an order, and require threads to grab them in order

- **Issue:** You can get race conditions! *(why?)*
  - w/ cvars at least...
  - even if you order them...
  - e.g., the first condition may no longer be true when you get through the second one
- **Solution:** consider testing BOTH conditions...